

Accelerating Modernization with Agile Integration

Adeline SE Chun

Aiden Gallagher

Amar A Shah

Callum Jackson

Claudio Tagliabue

Iliya Dimitrov

James Blackburn

Joel Gomez

Kim Clark

Lee Gavin

Maria Menendez

Martin Evans

Mohammed Alreedi

Murali Sitaraman

Nick Glowacki

Shishir Narain

Timothy Quigly

Tony Curcio

Ulas Cubuk

Vasfi Gucer



 **Cloud**



IBM Redbooks

Accelerating Modernization with Agile Integration

January 2020

Note: Before using this information and the product it supports, read the information in “Notices” on page xi.

First Edition (January 2020)

This edition applies to IBM Cloud Pak for Integration Version 2019.2.3.

© Copyright International Business Machines Corporation 2020. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xi
Trademarks	xii
Preface	xiii
Authors	xiii
Now you can become a published author, too!	xix
Comments welcome	xix
Stay connected to IBM Redbooks	xix
Chapter 1. Introduction	1
1.1 Integration has changed	2
1.2 Audience and scope	3
1.3 Navigating the book	3
Chapter 2. Agile integration	5
2.1 Agile integration: A brief introduction	6
2.1.1 People: Decentralized integration ownership	7
2.1.2 Architecture: Delivery focused integration architecture	7
2.1.3 Infrastructure aspect: Cloud-portable integration infrastructure	9
2.2 The journey so far: SOA, ESBs, and APIs	9
2.2.1 The forming of the ESB pattern	9
2.2.2 What went wrong for the centralized ESB pattern	10
2.2.3 The API economy	12
2.3 Microservices	13
2.3.1 The rise of lightweight run times	13
2.3.2 Microservices architecture: A more agile and scalable way to build applications	14
2.3.3 Comparing SOA and the microservices architecture	17
2.3.4 Bi-modal IT and decentralization	19
2.3.5 Decentralization and integration versus point-to-point	20
2.4 The three aspects of agile integration	21
2.5 People: Decentralized integration ownership	21
2.5.1 Moving to a decentralized and business-focused team structure	23
2.5.2 Big bangs generally lead to big disasters	24
2.5.3 Decentralized integration ownership and decentralized infrastructures	25
2.5.4 Prioritizing project delivery first	25
2.5.5 Evolving the role of the architect	26
2.5.6 Automation: The key to consistency in decentralization	27
2.5.7 Producing multi-skilled developers	28
2.5.8 Conclusions on decentralized integration ownership	30
2.6 Architecture: Delivery-focused architecture	30
2.6.1 Consumer-centric API management	31
2.6.2 Fine-grained application integration	32
2.6.3 Application-owned messaging and events	33
2.6.4 Conclusions on delivery-focused architecture	34
2.7 Technology: Cloud-native infrastructure	35
2.7.1 Virtual machines, containers, and serverless	35
2.7.2 Cloud-native approach	36
2.7.3 Portability: Public, private, and multicloud	38
2.7.4 Conclusion on cloud-native integration infrastructure	39

Chapter 3. Agile integration: Capability perspectives	41
3.1 Capability perspective: API management	42
3.1.1 A brief history of API management	42
3.1.2 Cloud-native infrastructure	53
3.2 Capability perspective: Application integration	54
3.2.1 Moving to a cloud-native approach	55
3.2.2 Fine-grained deployment: Breaking up the ESB	55
3.2.3 Grouping integrations	56
3.2.4 Stateless components	58
3.2.5 Image-based deployment	58
3.2.6 Elastic, agnostic infrastructure and container orchestration platforms	60
3.2.7 Lightweight run times: How the modern integration run time has changed	62
3.2.8 Log-based monitoring	62
3.2.9 API intra-application communication	62
3.2.10 Event-driven architecture	63
3.2.11 Agile methods	63
3.2.12 Continuous Integration and Continuous Delivery and Deployment	63
3.2.13 DevOps	64
3.2.14 Creating integrations is becoming easier	64
3.2.15 Decentralizing integration ownership	65
3.2.16 Using integration run times in a microservices application	66
3.3 Capability perspective: Messaging and event streams	68
3.3.1 A brief history of asynchronous communication	68
3.3.2 Introducing messaging and event streams concepts	69
3.3.3 Differentiating capabilities	71
3.3.4 A detailed look at messaging	73
3.3.5 A detailed look at event streams	77
3.4 Capability perspective: Files and Business-to-Business	80
3.5 Hybrid and multicloud considerations	82
3.5.1 Multicloud: Multiple cloud services	82
3.5.2 Hybrid Cloud: Multiple deployment modes (public, private, and legacy)	82
3.5.3 Evolution of API deployment modes	82
3.6 Use cases driving hybrid and multicloud adoption	83
3.6.1 Multicloud strategy	83
3.6.2 Cloud bursting and scalability	83
3.6.3 Disaster recovery	83
3.6.4 Application affinity	83
3.6.5 Regional flexibility	83
3.6.6 Geographical high availability	83
3.7 References	84
Chapter 4. Cloud-native concepts and technology	85
4.1 Defining cloud-native	86
4.2 Key elements of cloud-native applications	87
4.2.1 Modular components	88
4.2.2 Preferring stateless	88
4.2.3 Immutable deployment	89
4.2.4 Elastic, agnostic infrastructure and container orchestration platforms	92
4.2.5 Lightweight run times	95
4.2.6 Log-based monitoring	95
4.2.7 API-led intra-application communication	96
4.2.8 The reprise of event-driven architecture	96
4.2.9 Agile methods	97

4.2.10	Continuous Integration and Continuous Delivery and Deployment.	97
4.2.11	Continuous Adoption.	98
4.2.12	DevOps.	99
4.3	Twelve-factor apps	100
4.3.1	Conclusion on 12-factor apps	104
4.4	Container technology: the current state of the art.	104
4.4.1	Containers.	105
4.4.2	Container orchestration.	107
4.4.3	Kubernetes primer	107
4.5	Cloud-native is not for everyone, nor for everything	111
4.6	Realizing the true benefits of containerization	112
4.7	Application boundaries in a container-based world.	113
4.7.1	Implicit and explicit boundaries	113
4.7.2	Why do application boundaries matter?	114
4.7.3	How should we choose the application boundaries?	115
4.8	Service mesh	119
4.8.1	Role of a service mesh	120
4.8.2	Service meshes and API management.	122
4.9	Cloud-native security – an application-centric perspective	128
4.9.1	Scope of this section.	128
4.9.2	Limitations of traditional security models	128
4.9.3	Challenges unique to cloud-native	130
4.9.4	Securing a cloud-native application	130
4.9.5	Hybrid solutions: Securing cloud to cloud and cloud to ground	135
4.10	The future of cloud-native	138
4.10.1	Are software-as-a-service applications serverless?	139
4.10.2	Function-as-a-service: a more accurate term for serverless?	140
4.10.3	Could any runtime be provided in a FaaS model?	140
4.10.4	FaaS for cloud-native?	140
4.10.5	Are there downsides to FaaS?	140
4.10.6	Conclusions on FaaS	141
	Chapter 5. IBM Cloud Pak for Integration	143
5.1	IBM Cloud Pak for Integration.	144
5.1.1	One platform supported by common services.	144
5.1.2	IBM Cloud Pak for Integration - benefits.	144
5.1.3	License flexibility for other non-containerized architectures	145
5.1.4	Getting access to IBM Cloud Pak for Integration for the exercises.	145
5.2	Red Hat OpenShift Container Platform	146
5.3	API Lifecycle: IBM API Connect	147
5.3.1	Key phases of the API Lifecycle	148
5.3.2	API Lifecycle components.	148
5.3.3	API lifecycle in combination with other capabilities.	150
5.3.4	Product deployment options	151
5.4	Integration security: IBM DataPower Gateway	152
5.4.1	Security Gateway	153
5.4.2	API Gateway	154
5.4.3	DataPower and agile integration.	154
5.5	Application integration: IBM App Connect.	155
5.5.1	User-aligned integration tooling	155
5.5.2	No-code RESTful integration services	156
5.5.3	Flexible integration patterns	156
5.5.4	Broad deployment options	156

5.5.5	Extended connectivity	156
5.5.6	Situational awareness with insightful and actionable notifications	157
5.5.7	Quick utilization of artificial intelligence (AI) services	157
5.5.8	Rapid visual orchestration of data and systems for API-driven architectures	157
5.5.9	Lightweight integration runtime for cloud native deployment	157
5.5.10	Grown from a trusted market leading product.	158
5.5.11	IBM App Connect on deployment options.	158
5.6	Enterprise Messaging: IBM MQ	159
5.7	Event Streaming: IBM Event Streams.	160
5.8	High-Speed File Transfer: IBM Aspera	161
5.8.1	Fast, Adaptive and Secure Protocol (FASP) technology	161
5.8.2	Aspera on Cloud	162
5.9	Service Mesh: Istio	164
Chapter 6.	Practical agile integration	169
6.1	Introduction	170
6.2	Application Integration to front a data store with a basic API	173
6.2.1	Db2 setup	177
6.2.2	Db2 table setup.	178
6.2.3	Swagger definitions.	183
6.3	Expose an API using API Management	190
6.3.1	Importing the API definition.	191
6.3.2	Configuring the API.	196
6.3.3	Merging two application flows into a single API	196
6.3.4	Add simple security to the API	202
6.4	Messaging for reliable asynchronous data update commands.	209
6.4.1	Enable create, update, delete via commands.	210
6.4.2	Deploy and configure Queue Manager	211
6.4.3	Queue manager configuration	225
6.4.4	DB commands implementation	242
6.4.5	Graphical data maps implementation	246
6.4.6	Policy definitions	250
6.4.7	BAR file creation	253
6.4.8	Override policies for environment specific values.	255
6.4.9	Global transaction coordination considerations	257
6.4.10	Conclusion	259
6.5	Consolidate the new IBM MQ based command pattern into the API	259
6.5.1	Defining the API data model	260
6.5.2	Paths	261
6.5.3	Securing the API	265
6.5.4	The Assembly	266
6.5.5	API testing.	268
6.5.6	API socialization	271
6.5.7	Conclusion	276
6.6	Advanced API security	276
6.6.1	Import the API into IBM API Connect	278
6.6.2	Configure the API	278
6.6.3	Add basic security to the API	278
6.6.4	Test the API	278
6.6.5	Securing the API Using OAUTH	278
6.6.6	External client testing	303
6.7	Create event stream from messaging	322
6.7.1	Creating a new event stream topic	324

6.7.2	Running the IBM MQ source connector	324
6.7.3	Configuring the connector to connect to IBM MQ	326
6.8	Perform event-driven SaaS integration	328
6.8.1	Scenario	329
6.8.2	IBM App Connect event-driven flow to Salesforce, Google and Slack SaaS applications	330
6.8.3	Prerequisites	331
6.8.4	Create flows	331
6.8.5	Test your flow	343
6.8.6	Conclusion	344
6.9	Implementing a simple hybrid API	345
6.9.1	Business scenario	345
6.9.2	Invoking existing APIs from IBM App Connect Designer	346
6.9.3	Solution overview	347
6.9.4	Preparing the external SaaS applications	348
6.9.5	Create simulated on-premises API flow	349
6.9.6	Create Hybrid API	357
6.9.7	Test the flows	364
6.9.8	First, test the simulated on-premises API	364
6.9.9	Final Hybrid API integrated testing	367
6.9.10	Conclusion	367
6.10	Implement event sourced APIs	367
6.10.1	Implementing the query side of the CQRS pattern	368
6.10.2	Event sourced programming - a practical example	369
6.10.3	How to do it?	373
6.10.4	Creating the flow in IBM App Connect	374
6.10.5	Mapping the received events to the output required	378
6.10.6	Sending the new payload to the database	380
6.10.7	Client applications	381
6.11	REST and GraphQL based APIs	381
6.11.1	IBM, GraphQL, and Loopback	383
6.11.2	LoopBack models and relationships	383
6.12	API testing	398
6.12.1	Create a test from an API request	399
6.12.2	Update the test case from a Swagger file and publish	403
6.12.3	Gain insights into API quality	408
6.13	Large file movement using the claim check pattern	410
6.13.1	Build the file transfer	411
6.13.2	Build an event-driven flow	420
	Chapter 7. Field notes on modernization for application integration	433
7.1	IBM App Connect adoption paths	434
7.1.1	Agile integration	434
7.1.2	Adoption path options	434
7.1.3	Conclusion	440
7.2	Splitting up the ESB: Grouping integrations in a containerized environment	441
7.2.1	What grouping do you have today?	442
7.2.2	Splitting by business domain	442
7.2.3	What about integrations that span business domains?	443
7.2.4	Grouping within a domain	444
7.2.5	Stable requirements and performance	445
7.2.6	Synchronous versus asynchronous patterns	446
7.2.7	Shared lifecycle	449

7.2.8	A worked example	450
7.2.9	Conclusion	451
7.3	When does IBM App Connect need a local MQ server?	452
7.3.1	Benefits of being dependency-free in container-based environments	452
7.3.2	When can we manage without a local MQ server?	454
7.3.3	Can I talk to multiple queues in the same transaction without a local MQ server?	456
7.3.4	Coordinating a two-phase commit requires a local MQ server	457
7.3.5	When else do I need a local MQ server?	459
7.3.6	Why do we have so many integrations that use server connections?	459
7.3.7	Conclusion	460
7.4	Mapping images to helm charts	460
7.4.1	Developing helm charts for Kubernetes	461
7.4.2	Upgrading (extending) helm charts	464
7.5	Continuous Integration and Continuous Delivery Pipeline using IBM App Connect V11 architecture	465
7.5.1	Continuous Integration Delivery and Deployment	467
7.5.2	Example pipeline - High-level concepts	468
7.5.3	CI/CD pipeline in depth	470
7.5.4	Considerations for CI/CD pipelines with IBM App Connect	471
7.5.5	Practical example	475
7.6	Continuous Adoption for IBM App Connect	485
7.6.1	What does Continuous Adoption apply to in IBM App Connect?	485
7.6.2	How can Continuous Adoption be implemented with IBM App Connect?	486
7.7	High Availability and Scaling considerations for IBM App Connect in containers	490
7.7.1	Overview	490
7.7.2	Scaling	490
7.7.3	High Availability	492
7.8	Migrating centralized ESB to IBM App Connect on containers	493
7.8.1	Overview	493
7.8.2	Considerations for IBM MQ based integrations in containers	494
7.8.3	Considerations for Http/WebServices based integration flows in containers	511
7.8.4	Considerations for integrations that interact with databases	512
7.8.5	Considerations for in containers	516
7.8.6	Considerations for TCP/IP based integrations in containers	522
7.8.7	Considerations for file-based integration in containers	524
7.8.8	Considerations for integrating IBM App Connect with IBM Event Streams	528
7.9	Splitting an integration across on-premises and cloud	534
7.9.1	Overview	534
7.9.2	Using callable flows with IBM App Connect Designer	535
7.9.3	Callable flows versus APIs	537
7.9.4	Cloud debugger for ACE on Cloud applications	537
	Chapter 8. Field notes on modernization for API lifecycle	539
8.1	Move from DataPower only to API Connect	540
8.1.1	Security Gateway	541
8.1.2	API gateway	541
8.1.3	Connectivity and mediation	542
8.1.4	DataPower and API Connect compared	542
8.1.5	DataPower to API Connect migration	542
8.2	Enterprise APIs across a hybrid or multicloud boundary	545
8.3	How many API Connect Clouds and Gateways	548
8.3.1	Separate API Clouds	548

8.3.2	Separate API Gateway Cluster	548
8.4	Organization, Catalog and Space responsibilities for APIs	549
8.5	Automated provisioning of a new API provider team	550
8.6	High availability and scaling on containers for API Management	553
8.6.1	High availability in a containerized environment	553
8.6.2	Scalability of API Connect in containerized environment	560
8.6.3	Conclusion	563
8.7	IBM API Connect API Test Pyramid	563
8.7.1	Practical Test Pyramid	563
8.7.2	Requirements	564
8.7.3	Test types	565
8.7.4	Automated testing	568
8.7.5	Conclusion	569
	Chapter 9. Field notes on modernization for messaging	571
9.1	Modernizing your messaging topology with containers	572
9.1.1	Typical existing topology	573
9.1.2	Removing local connections	574
9.1.3	Containerizing queue managers	575
9.1.4	Fine-grained queue manager deployment	576
9.1.5	Decentralization	578
9.1.6	Application containerization and operational consistency	580
9.2	IBM MQ availability	581
9.2.1	Kubernetes deployment styles: Deployments (replica sets) versus stateful sets	582
9.2.2	Multi-instance queue managers in containers	585
9.2.3	Further improving service availability with additional independent queue managers	586
9.2.4	Connection distribution	587
9.3	IBM MQ scaling	588
9.4	Automation of IBM MQ provisioning using a DevOps pipeline	594
9.4.1	Design for DevOps pipeline	594
9.4.2	Building a sample IBM MQ pipeline	598
	Appendix A. Additional material	619
	Locating the GitHub material	619
	Cloning the GitHub material	619
	Related publications	621
	IBM Redbooks	621
	Online resources	621
	Help from IBM	624

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

AIX®	IBM®	Redbooks (logo)  ®
Aspera®	IBM API Connect®	Terraform®
Cloudant®	IBM Cloud™	WebSphere®
DataPower®	IBM Cloud Pak™	z/OS®
DB2®	OpenWhisk®	
FASP®	Redbooks®	

The following terms are trademarks of other companies:

Evolution, are trademarks or registered trademarks of Kenexa, an IBM Company.

LoopBack, are trademarks or registered trademarks of StrongLoop, Inc., an IBM Company.

ITIL is a Registered Trade Mark of AXELOS Limited.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Ceph, OpenShift, Red Hat, are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

VMware, and the VMware logo are registered trademarks or trademarks of VMware, Inc. or its subsidiaries in the United States and/or other jurisdictions.

Other company, product, or service names may be trademarks or service marks of others.

Preface

The organization pursuing digital transformation must embrace new ways to use and deploy integration technologies, so they can move quickly in a manner appropriate to the goals of multicloud, decentralization, and microservices. The integration layer must transform to allow organizations to move boldly in building new customer experiences, rather than forcing models for architecture and development that pull away from maximizing the organization's productivity. Many organizations have started embracing agile application techniques, such as microservice architecture, and are now seeing the benefits of that shift. This approach complements and accelerates an enterprise's API strategy. Businesses should also seek to use this approach to modernize their existing integration and messaging infrastructure to achieve more effective ways to manage and operate their integration services in their private or public cloud.

This IBM® Redbooks® publication explores the merits of what we refer to as *agile integration*; a container-based, decentralized, and microservice-aligned approach for integration solutions that meets the demands of agility, scalability, and resilience required by digital transformation. It also discusses how the IBM Cloud Pak for Integration marks a significant leap forward in integration technology by embracing both a cloud-native approach and container technology to achieve the goals of agile integration.

The target audiences for this book are cloud integration architects, IT specialists, and application developers.

Authors

This book was produced by a team of specialists from around the world working in IBM Hursley Center.



Adeline SE Chun is a certified IT Specialist working for IBM Singapore as a ASEAN Technical Leader in Connectivity and Integration solutions. Adeline has over 20 years of experience in IT and specialized in Enterprise Connectivity and Integration Solutions. Prior to joining IBM Singapore in 2010, she worked as an IT architect for IBM Toronto Lab Services since 1997, implementing IBM WebSphere® e-Business solutions for worldwide banks, including HSBC, Banco Santiago, ING, ICBC Shanghai, Barclays Bank UK, ANZ Bank, and BBL Belgium. Using the WebSphere architecture, she implemented Teller, Call Center, and Internet banking solutions based on the IBM service-oriented architecture (SOA) methodology. As a technical leader, Adeline helps the sales teams across ASEAN to deliver technical solution workshops and proposals on Enterprise Integration solutions. Adeline is viewed as a senior technical consultant and has long standing relationships with IBM Customers in all Geographies. She is considered a Subject Matter Expert for MQ, IBM Integration Bus, MQ Telemetry Transport (MQTT) and other connectivity technologies across broad platform.



Aiden Gallagher is an Integration Consultant for the UK and Ireland working with API Connect, IBM App Connect, MQ and DataPower® products. He joined IBM in 2015 having graduated from Nottingham Trent University with a degree in Computer Science. His areas of expertise include Open Banking, Cloud Deployments and integration. He has written extensively on the history of integration, NodeJS, Agile and API Connect. You can find a list of his articles, node modules, achievements and projects here; <https://www.linkedin.com/in/aiden-gallagher-a5a698b7/>.



Amar Shah is a Serviceability Architect working with the IBM Application Integration support team. He is responsible for worldwide support for clients of IBM App Connect (formerly Integration Bus) and the serviceability enhancement of products. He is also a designated Lab Advocate for key clients and provides advice and consultancy on product solution and usage best practices. Amar Shah has been associated with IBM for the past 17 years, and holds a Master's degree in Software Systems from Birla Institute of Technology, Pilani, India.



Callum Jackson is a solution architect within the IBM Messaging Offering Management team. He works as a technical gateway between clients and the IBM development team, to understand client challenges, and how IBM Messaging and the wider IBM Integration portfolio can solve these. Prior to this, Callum spent several years within the IBM Lab Services organization, building Integration solutions across the IBM Integration portfolio.



Claudio Tagliabue is an IBM Cloud™ Solution Architect experienced in the world of Digital Transformation and the world of XaaS. He is excited by business change in the 21st Century and the consequent demands on technology. Making the "complicated" "business as usual" has been Tag's focus for the past 10 years. Through his work with clients as a domain expert for IBM's ground-breaking cloud technologies, Tag has gained significant insight into today's senior IT stakeholder motivations. Tag has worked in various IBM labs, and regularly presents at conferences around the world, focusing on Public and Private Cloud, Kubernetes (K8s), Terraform®, microservices, BPM, RPA, IaaS, PaaS, and software-as-a-service (SaaS). Tag also authored two IBM Redbooks publications. He can be followed on Twitter @ClaudioTag.



Iliya Dimitrov is an Early Experience Programs Manager for IBM App Connect in IBM UK. He has worked with the product since WebSphere Message Broker V7, focusing on Worldwide Early customers' engagements and enablement. He holds two Master's degrees in Electronics and Communication Systems Engineering from Technical University of Varna, Bulgaria and University of Portsmouth, United Kingdom. Iliya is an IBM Certified IT Specialist liaising with clients and internal development teams delivering education and helping Early Product adoption with the latest product's features.



James Blackburn is the Principal Architect for Integration and Middleware at Marks & Spencer, a leading British retailer. James is accountable for the integration roadmap, architecture and design of integration, API and middleware technology for Marks & Spencer, this spans all cloud and on-premises integration, message orientated middleware, managed file transfer, Business-to-Business (B2B) and Electronic Data Interchange (EDI), ETL, Events, Services and APIs. James holds a Master of Science degree in Software Engineering and has been working with IBM integration technology for over 12 years all centred around digital retail and commerce use cases. predominantly using MQ and App Connect.



Joel Gomez is a Sr. Integration Technical Specialist for North America with more than 27 years of experience in technical sales related to Application and Integration Middleware solutions, including MQ, App Connect, and API Connect. He holds a Masters degree in e-Commerce from Instituto Tecnológico y de Estudios Superiores de Monterrey in Mexico. His areas of expertise include Cloud Computing, SOA & API Economy, as well as Application Development, Web, Mobile & Microservices Technologies. In recent years he has focused on Integration for Healthcare creating the North America Integration for Healthcare User Group (<https://community.ibm.com/community/user/imwuc/communities/community-home?communitykey=32ce99a0-ddc0-4e99-a800-13a9f3563f38&tab=groupdetails>), and leading a monthly call with some of the largest healthcare providers and payers in North America, where they discuss topics like the Healthcare Pack, DICOM, HIPAA, HL7 and FHIR.



Kim Clark is a technical strategist for IBM Cloud Pak™ for Integration working as an architect providing guidance to the offering management team on current trends and challenges. He has spent the last couple of decades working in the field implementing integration and process related solutions. You can find Kim on LinkedIn: <https://www.linkedin.com/in/kimjulianclark?>



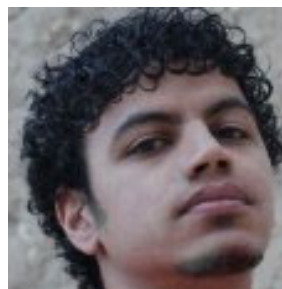
Lee Gavin is a Technical Specialist on the European Technical Sales team with IBM. She has many years experience advising customers and implementing integration solutions using IBM's integration capabilities. Prior to joining the European team she was a member of the World Wide Technical Sales team. Lee has written, delivered hands-on workshops and presented extensively over the years, on areas ranging from Hybrid Integration to Business Process Automation. In a previous life she was also a Project Leader with the International Technical Support Organization at the Hursley and Raleigh Centers and is the author of over a dozen IBM Redbooks.



Maria Menendez leads North America Technical Sales for IBM's Cloud Integration software portfolio. She has over 20 years of experience in integration and business process automation. Maria began her career as an IBM customer, implementing 3270 and 5250 screen scraping Java-based web solutions, then implementing MQ and WebSphere Business Integration and B2B solutions. At IBM, Maria has worked with customers in Central Region, North America and around the world in various technical sales and technical leadership roles in Integration, Business Process Automation and Analytics. She can be reached at <https://www.linkedin.com/in/maria-menendez-she-her-76a09b8/>.



Martin Evans is an Integration Architect working in the IBM Cloud Integration Expert Labs providing customers with integration consultancy services. He has been delivering integration solutions for over 15 years predominantly in the financial services sector and more recently for the retail industry.



Mohammed Alreedi is the IBM technical leader for integration in the Middle East and Africa (MEA). He is an IT certified technical specialist for Integration, Mobile and API Economy with expertise in full lifecycle API management, application integration, enterprise messaging, high speed file transfer and mobile backend-as-a-service. Mohammed Alreedi is responsible for providing strategic business, technical advice and recommendations for IBM MEA integration team. He is focusing now on "mega projects" to bridge the power of analytics and AI with the API experience which take the client through the journey of data monetization. He can be reached at <https://www.linkedin.com/in/alreedi>.



Murali Sitaraman is a Senior Technical Specialist in the Cloud Integration space in Switzerland. After his early career working on mobile internet (WAP) for a telco, Murali re-joined IBM in 2007 and has focused on the middleware- and integration space ever since. Over time he has gathered knowledge on Application Server, BPM, Business Rules, Business Event Management, Mobile Application Development, API Management and Cloud Integration. He has coauthored the product certification for the latest version of API Connect. He holds an MBA which enables him to translate tech-to-business - which he refers to as "another form of integration". Murali presents at events and conducts workshops on the topic of Integration Modernization. He shares his insights and views on [n <https://ch.linkedin.com/in/msit>](https://ch.linkedin.com/in/msit) and a the more personal side on twitter [@atech_e](https://twitter.com/atech_e).



Nick Glowacki is a technical evangelist for IBM Integration and advises large financial services organizations across the US who are investing on the leading edge. He has spent the last 5 years working with businesses who are transforming their integration architectures along a microservices journey. Nick leverages his experiences prior to IBM where he led the development and architecture of a microservices framework at a time prior to these techniques becoming. Across his career, he has held various other roles including developer, architect and integration security specialist. Nick can "go deep" on a number of technologies, including node, xsl, JSON, Docker, Solr, K8s, Java, SOAP, XML, C++, Docker, WebSphere Application Server, Filenet, MQ, API Connect, App Connect.



Shishir Narain is a certified IT Specialist working for IBM India. He has over two decades of experience in IT. He holds a master's degree in Industrial Management & Engineering from IIT Kanpur. He has worked in multiple large IT transformation projects as an IT Architect. His primary expertise is with IBM middleware products like WebSphere Application Server, MQ, App Connect, and API Connect - and he has written extensively on them. He can be reached at <https://www.linkedin.com/in/nshishir/>.



Timothy Quigly is an Integration Consultant working for IBM Cloud Expert Labs in the United Kingdom and Ireland. Tim works with customers throughout the UK and his focus is on developing DevOps approaches for integration using App Connect and MQ with Docker and K8s. Prior to Tim's life in consulting he worked as part of the IBM MQ development team in IBM's Hursley Labs, focusing on testing integration across Linux, Windows and IBM z/OS® environments. Tim holds four United States patents and has presented at conferences such as Impact and the WebSphere User Group. He can be reached at <https://uk.linkedin.com/in/timquigly>.



Tony Curcio has been focused on helping organizations leverage data and application integration technologies to maximize the impact of their digital assets to fuel business transformation for the past 20 years. After years of building solutions with customers, he joined the IBM Offering Management team where he led the strategy and roadmap for a set of integration, quality and governance technologies. Now as the Director of API Management and Gateway, Tony leads a worldwide team focused on helping customers through their digital transformation, spanning both cloud and on-premise topologies.



Ulas Cubuk has over 10 years of experience working with customers in Europe, Middle East and Africa across various industry verticals implementing middleware solutions. She has actively taken part in software development life cycle as a programmer, tester, analyst, configuration manager and team leader within varieties of GIS, ERP and banking projects. She holds a Master's degree in Geographical Information Systems and Environmental Engineering from Middle East Technical University. Ulas has recently joined the IBM Offering Manager team, where she helps set product direction based on her practical customer-focused experiences.



Vasfi Gucer is an IBM Redbooks Project Leader with the IBM International Technical Support Organization. He has more than 20 years of experience in the areas of systems management, networking hardware, and software. He writes extensively and teaches IBM classes worldwide about IBM products. His focus has been on cloud computing for the last 5 years. Vasfi is also an IBM Certified Senior IT Specialist, Project Management Professional (PMP), IT Infrastructure Library (ITIL) V2 Manager, and ITIL V3 Expert.

Thanks to the following people for their contributions to this project:

Mike Alley, Erik D. Anderson, Alan Glickenhause, Matt Lesher, Bruno Neves, Monica Raffaelli, Asim Siddiqui, Dan Temkin
IBM USA

Chris Phillips, Phil Coxhead, Andy Garratt, Rob Nicholson, Trevor Dolby, Rob Convery, Giacomo Chiarella, Richard Hine, Luca Floris, Chris Dudley, Simon Kapadia, Trevor Dolby, Matthew Clarke, Justin Deane, Alan Chatt, David Ware, Matt Roberts
IBM UK

Len Thornton
IBM Canada

Oliver Lucht
IBM Germany

Jeroen van Der Schot
IBM France

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an IBM Redbooks residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at: ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an email to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, IBM Redbooks
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:

<http://www.facebook.com/IBMRedbooks>

- ▶ Follow us on Twitter:

<http://twitter.com/ibmredbooks>

- ▶ Look for us on LinkedIn:

<http://www.linkedin.com/groups?home=&gid=2130806>

- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>

- ▶ Stay current on recent Redbooks publications with RSS Feeds:

<http://www.redbooks.ibm.com/rss.html>



Introduction

This chapter introduces agile integration and summarizes what we cover in this IBM Redbooks publication.

The following topics are covered in this chapter:

- ▶ Integration has changed
- ▶ Audience and scope
- ▶ Navigating the book

1.1 Integration has changed

Over the last few years, we have seen a tremendous acceleration in which customers are establishing digital transformation initiatives. In fact, International Data Corporation (IDC) estimated that by 2021, driven by line of business (LOB) needs, 70% of CIOs deliver *agile connectivity* through APIs and architectures that interconnect digital solutions from cloud vendors, system developers, start-ups, and others.¹ It is a staggering figure regarding the impact across all industries and companies of all sizes. A primary focus of this digital transformation is to build connected customer experiences across a network of applications that use data of all types.

However, bringing together these processes and information sources at the correct time and within the correct context has become increasingly complicated. Consider that many organizations aggressively adopted software-as-a-service (SaaS) business applications and spread their key data sources across a much broader landscape. Additionally, new data sources that are available from external data providers must be injected into business processes to create competitive differentiation.

Finally, AI capabilities, which are being attached to many customer-facing applications, require a broad range of information to train, improve, and correctly respond to business events. These processes and information sources must be integrated by making them accessible synchronously through APIs, propagated in near real time by event streams, and by using a multitude of other mechanisms.

The rise of the digital economy, like most of the seismic technology shifts over the past several centuries, fundamentally changed technology and business. The concept of “digital economy” continues to evolve. It was a section of the economy that was built on digital technologies but has evolved to become almost indistinguishable from the *traditional economy*. And it grows to include almost any new technology, such as mobile, the Internet of Things (IoT), cloud computing, and artificial intelligence.

At the heart of the digital economy is the basic need to connect disparate data. This need led to the rise of integration, which is the need to connect multiple applications, data, and devices. As a result, the system delivers the greatest insight to the people and systems who can act on it.

Agile integration architecture draws learning from hundreds of customer interactions and takes note of the dramatic changes that affect the integration landscape. The book looks at how customers are improving the agility of their integration landscape to align with the parallel advances occurring in their application delivery.

This book explores *agile integration* concepts in renewed detail and breadth, and is supplemented with practical examples and field notes that are gathered from the experiences of early adopters. Our aim is to collate meaningfully as much of the current theoretical and practical knowledge around integration modernization into one place to better enable IT leaders in their pursuit of digital transformation.

¹ IDC Reveals Worldwide CIO Agenda 2019 Predictions - <https://www.idc.com/getdoc.jsp?containerId=prUS44420918>

1.2 Audience and scope

Chapter 2, “Agile integration” on page 5 through Chapter 4, “Cloud-native concepts and technology” on page 85 cover relatively deep architectural territory. But we aim to define all terms and technologies that are mentioned so that it is suitable for any reader. Our expectation is that many readers are familiar with integration as a topic and have experience with one or more integration technologies. We move relatively quickly over topics that we expect that the reader knows.

Then, we focus on technology that empowers a modern integration journey. Chapter 5, “IBM Cloud Pak for Integration” on page 143 introduces the core integration capabilities. Chapter 6, “Practical agile integration” on page 169 is written such that the scenarios should be readable even where they involve capabilities in which you might not be a specialist. Chapter 7, “Field notes on modernization for application integration” on page 433 through Chapter 9, “Field notes on modernization for messaging” on page 571 are written for specialists, and they describe some of the common issues with typical modernization programs.

A single book cannot document integration modernization in its entirety, especially in a space that is under such rapid evolution. For more information about this topic, see the following resources:

- ▶ [Agile Integration](#)
- ▶ [API Management](#)
- ▶ [Application Integration](#)
- ▶ [Messaging and Events](#)

1.3 Navigating the book

Chapter 2, “Agile integration” on page 5 explores the effect that digital transformation has on both the application and integration landscape, and the limitations of previous techniques. We describe what led us to this point, the pros and cons of service-oriented architecture (SOA) and the enterprise services bus (ESB) pattern, the influence of APIs, and the introduction of the microservices architecture. We also describe the current state of agile integration.

The various capabilities that are associated with integration provide differing contributions to digital transformation. Chapter 3, “Agile integration: Capability perspectives” on page 41 explores three core technological perspectives: application integration, API management, and messaging and events. And it describes how their usage is changing in the context of agile integration.

Chapter 4, “Cloud-native concepts and technology” on page 85 provides a detailed description of what it means to build solutions by using cloud-native techniques. These concepts are the same for any type of component, so they have the same concerns for an application developer as they do for someone implementing integration.

Chapter 5, “IBM Cloud Pak for Integration” on page 143 provides an overview of IBM Cloud Pak for Integration, and describes recent advances that enable integration modernization.

Chapter 6, “Practical agile integration” on page 169 describes basic, practical scenarios that use integration capabilities that are based on common use cases for agile integration. This chapter explores expansion from simple exposure of a database, through managed and secure APIs, to implementing modern application patterns such as event sourcing and Command Query Responsibility Segregation (CQRS). We demonstrate different deployment options, such as using the administration capabilities within IBM Cloud Pak, deploying integration containers in your own cluster, and using the capabilities as managed services that are hosted in the cloud.

Chapter 7, “Field notes on modernization for application integration” on page 433 through Chapter 9, “Field notes on modernization for messaging ” on page 571 collate field notes that are based on the experiences of early adopters of these approaches, which are grouped again by application integration, API management, and messaging and events. As expected, there is plenty of discussion about the subtleties of migration to containers. So, we also describe the practicalities that relate to moving to cloud-native containers, such as changing team roles and responsibilities and strategies for testing.



Agile integration

This chapter begins with a brief description of agile integration. It then reviews the background of how several enterprises implement a centralized enterprise services bus (ESB). Then, it explores how to move forward to a more agile integration landscape that better enables a business to compete and potentially disrupt its market segment.

The following topics are covered in this chapter:

- ▶ Agile integration: A brief introduction
- ▶ The journey so far: SOA, ESBs, and APIs
- ▶ Microservices
- ▶ The three aspects of agile integration
- ▶ People: Decentralized integration ownership
- ▶ Architecture: Delivery-focused architecture
- ▶ Technology: Cloud-native infrastructure

2.1 Agile integration: A brief introduction

The pace of innovation in IT has changed dramatically. Iterations on requirements are in near real-time, prototypes are prepared in weeks or even days, and new mobile apps are made available in months. Application development techniques needed to keep pace by introducing new approaches such as microservices that enable teams to work more independently.

Integration is maturing towards a more API-led approach where interfaces that are easy to use enable teams to rapidly share data and functions.

However, there is much more subtlety to this change than is apparent. You must think differently about how you align the people and skillsets that relate to integration. You must consider how to ensure that integration components embrace new architectural tenets such as microservices and that they capitalize on the benefits of new infrastructure platforms such as containers.

As shown in Figure 2-1, agile integration addresses these issues in the detail, looking at how to modernize an integration landscape based on people, architecture, and infrastructure.

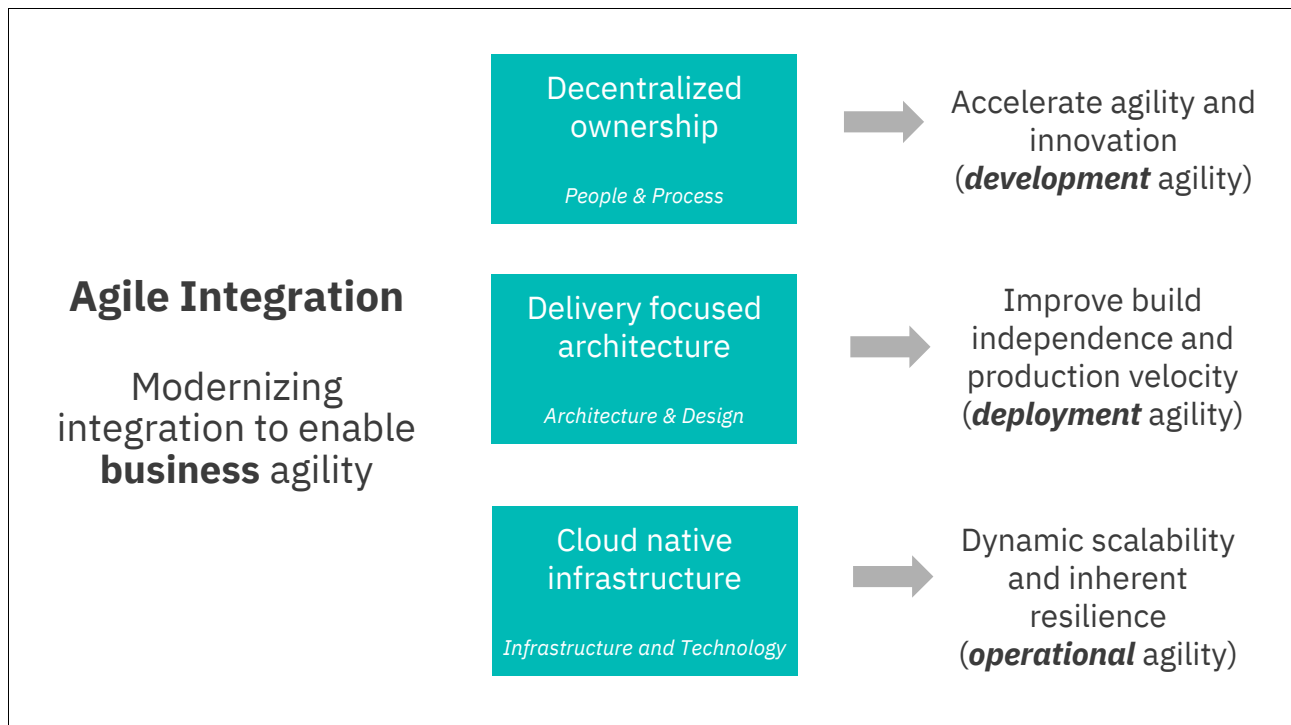


Figure 2-1 Three aspects of agile integration

Integration is the secret weapon behind the great innovations of today. Few new ideas are stand-alone applications. They always require data and functions from other applications within the enterprise and often from other enterprises.

However, the integration challenge is less about low-level connectivity than it is about the velocity of change. How quickly can ideas be transformed into production, or at least into prototypes, so that new niches can be leveraged?

This process requires highly empowered and autonomous teams that can self-provision the integration capabilities that they need wherever they are and still interact efficiently with other teams' capabilities.

Integration is prominent in enabling the *business agility* that is required to innovate faster than the competition. *Agile integration*, as shown in Figure 2-1, addresses this need by rethinking the approach to integration that is based on people, architecture, and technology. It results in a decentralized, microservices-aligned, and portable approach to integration:

- ▶ People and decentralized integration ownership: Improve *development agility* by empowering teams with integration capabilities so that they can innovate in real time.
- ▶ Architecture and delivery focused integration architecture: Improve *deployment agility* by using modern architectural and design practices such as API-led or event-driven integration with a focus on microservices applications.
- ▶ Technology and a cloud-portable integration infrastructure: Improve *operational agility* by using a platform-neutral and cloud-native approach to integration infrastructure.

We introduce these aspects at a high level in this chapter before going into detail into each one in subsequent chapters.

2.1.1 People: Decentralized integration ownership

In the past, a significant challenge for a service-oriented architecture (SOA) was that it tended to result in heavily centralized integration teams and a corresponding infrastructure to create the service layer. Although centralized teams certainly had their place, for example, making governance of quality easier, they also had downsides in terms of their ability to scale.

Because there was a limited set of skilled resources in the organization regarding integration technology, all projects depended on this centralized team. Although the team knew their integration technology well, they often did not understand the applications they were integrating, so conversion requirements might be slow and error-prone.

Many organizations prefer that the application teams own the creation of their own services, but the technology and infrastructure at the time did not enable that situation.

Today, organizations are moving to *fine-grained* integration deployment, which enables the distribution of the ownership of the creation and maintenance of the integrations.

With the evolution of integration technologies, it is now perfectly reasonable for business application teams to take on integration work and streamline the implementation of new capabilities. We describe this topic in 2.5, “People: Decentralized integration ownership” on page 21.

2.1.2 Architecture: Delivery focused integration architecture

In the architectural space, each integration capability contributes differently to improving integration agility.

Consumer-centric API management

An API-led integration strategy for connectivity between applications is now standard. API management furthered the ideas in SOA by refining and improving the standards around how interfaces are shared between applications and between enterprises in an API economy. A key lesson is the importance of creating and exposing APIs based on the needs of the API consumer.

APIs are now treated more like products than technical interfaces, so they must be marketable and potentially monetized. Promoting APIs by using slick developer portals that

enable self-subscription and convenient ways to learn and test the interface are critical to the success of any API.

In today's multicloud world, it is also critical to be able to administer, catalog, and secure APIs from a single place even if they are exposed on many different cloud endpoints, which further simplifying the consumer's experience.

Fine-grained application integration

In 2.4, “The three aspects of agile integration” on page 21, you explore why microservices concepts are popular in the application space. You see how those principles can be applied to the modernization of an integration architecture.

The centralized deployment of an integration hub or ESB pattern, where all integrations are deployed to a single, highly available (HA) pair of integration servers, has some benefits in terms of consistency, simplicity, and efficiency. However, centralized deployment might introduce a bottleneck for projects. Furthermore, any deployment to the shared servers runs a risk of destabilizing existing critical interfaces because this deployment is on a single software instance, so no individual project can choose to upgrade the version of the integration middleware to gain access to new features without impacting others.

You could break up the enterprise-wide ESB component into smaller more manageable and dedicated pieces. Perhaps in some cases you can even get down to one run time for each interface you expose. These *fine-grained integration deployment* patterns provide specialized, right-sized containers for improved agility, scalability, and resilience, and are different than the centralized ESB patterns.

Application-owned messaging and events

Asynchronous communication is even more relevant in today's geographically separated distributed solutions:

- ▶ Messaging, which was originally introduced to enable decoupled communication across disparate platforms, continues in that mission-critical purpose, but it also now has the same role in reliable communication across cloud boundaries.
- ▶ Events provide mechanisms to store an event history, which provides an alternative source of information about data changes and enable applications to listen selectively for notifications, and build local data stores suited to their needs.

However, it is no longer acceptable to wait on a highly specialized team to provision an asynchronous communication infrastructure. Teams must be able to self-provision and configure queues and topics for immediate use as part of their event-driven integration projects.

Templated and patternized mechanisms must be introduced to simplify provisioning tasks, and where possible provide them in as *in-place* multi-tenant managed services. The queue and topic configurations must be application-owned so that they can produce prototypes and iterate on solutions rapidly.

2.1.3 Infrastructure aspect: Cloud-portable integration infrastructure

Integration implies that key elements must be able to run on a multitude of different platforms to be in proximity to the systems with which they are integrating. As such, the run times that are used by these capabilities, whether they are mediation engines, API gateways, or queue managers of event brokers, must be designed for the greatest possible portability. They must run on a traditional platform and take advantage of cloud-native infrastructure, that is, they must be able to hand off the burden of many of their previously proprietary mechanisms for cluster management, scaling, and availability to the cloud platform in which they are running. This process entails more than running these mechanisms in a containerized environment: They must make the best use of the orchestration capabilities, such as Kubernetes (K8s) and many other common cloud standard frameworks.

There are multiple benefits of being able to run in a cloud-native style on portable platforms, including cost efficiencies from elastic scaling, rationalization of infrastructure skillsets, and consumer choice of cloud platforms.

2.2 The journey so far: SOA, ESBs, and APIs

Before you dive into agile integration, you first must learn more about the background of this topic. In this section, we briefly look at the challenges of SOA by taking a closer look at what the ESB pattern was, how it evolved, where APIs came onto the scene, and the relationship between the pattern and the microservices architecture.

2.2.1 The forming of the ESB pattern

As we started the millennium, we saw the beginnings of the first cross-platform protocol for interfaces. The internet and HTTP became ubiquitous, XML was being developed from HTML, and the SOAP protocols for providing synchronous web service interfaces were taking shape. Relatively wide acceptance of these standards hinted at a future where any system might discover and talk to any other system by using a real-time synchronous remote procedure call (RPC) without reams of integration code.

From this series of events, SOA was developed. The core purpose of SOA is to expose data and functions in systems of record through well-formed, simple-to-use, and synchronous interfaces, such as web services. Clearly, SOA was about more than just providing those services, and often involved some significant reengineering to align the back-end systems with the business needs, but the end goal was a suite of well-defined, common, and reusable services collating disparate systems. SOA enabled new applications to be implemented without the burden of deep integration every time because after the integration was done for the first time and exposed as a service, it could be reused by the next application.

However, this simple integration was a one-sided equation. We might have been able to standardize these protocols and data formats, but the back-end systems of record were typically old and had antiquated protocols and data formats for their current interfaces. Something was needed to mediate between the old system and the new cross-platform protocols.

This synchronous exposure pattern through web services was why ESB was introduced, as shown by the upper *integration* component in Figure 2-2. It is a centralized *bus* that can provide web *services* across the *enterprise*.

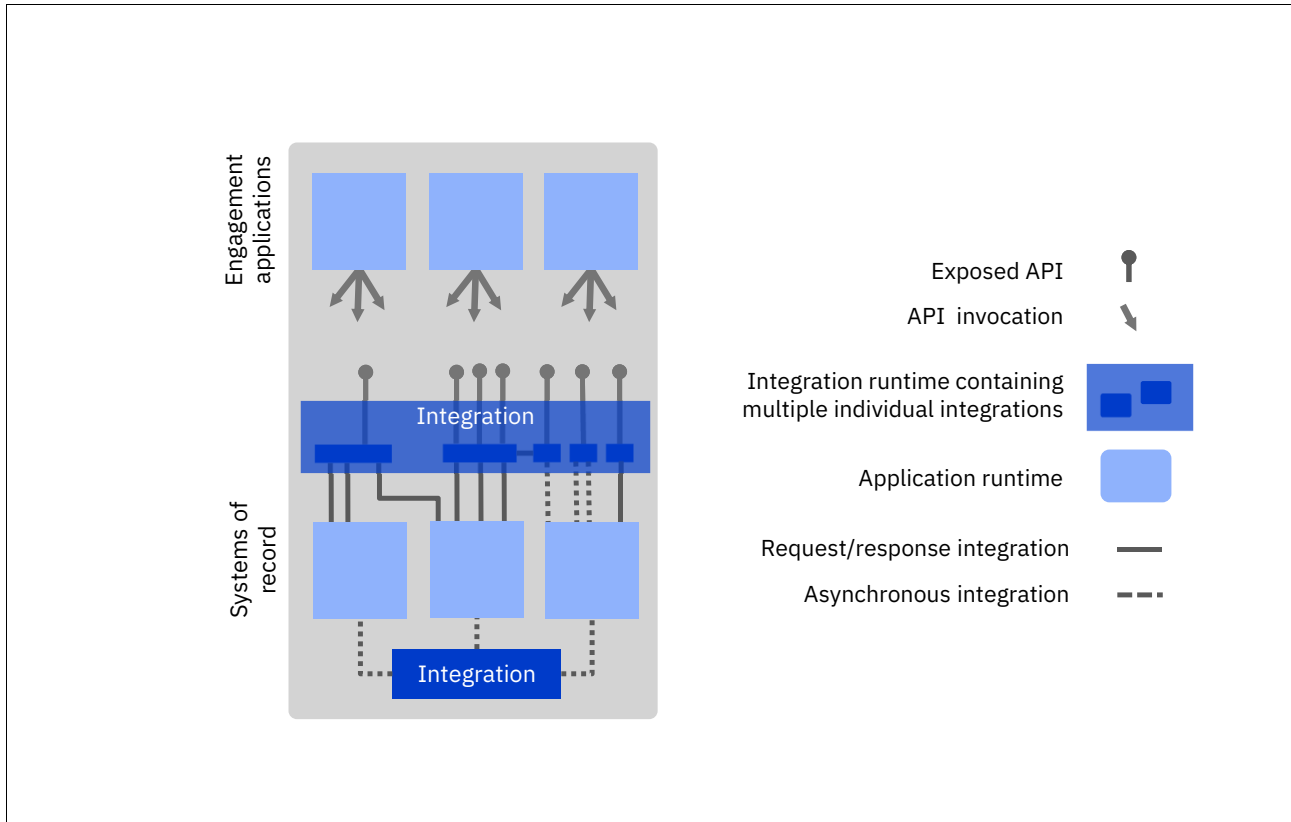


Figure 2-2 Centralized ESB: No gateway

We already had the technology (the integration run time) to provide connectivity to the back-end systems coming from the preceding hub-and-spoke pattern. These integration run times could simply be changed to offer integrations synchronously through SOAP and HTTP, and we would have our ESB.

2.2.2 What went wrong for the centralized ESB pattern

Although many large enterprises successfully implemented the ESB pattern, the term is often disparaged in the cloud-native space, especially in relation to the microservices architecture. It is seen as heavyweight and lacking in agility. What has happened to make the ESB pattern appear so outdated?

SOA turned out to be a more complex than just the implementation of an ESB for a host of reasons, such as who would fund such an enterprise-wide program. Implementing the ESB pattern itself is no small task.

The ESB pattern often took the “E” in ESB literally and implemented a single infrastructure for the whole enterprise, or at least one for each significant part of the enterprise. Tens or even hundreds of integrations might have been installed on a production server cluster, and if that was scaled up, they would be present on every clone within that cluster. Although this heavy centralization is not required by the ESB pattern itself, it was almost always present in the resultant topology.

There were good reasons for this situation, at least initially: hardware and software costs were shared, provisioning of the servers only had to be performed once, and due to the relative complexity of the software, only one dedicated team of integration specialists needed to be skilled up to perform the development work.

The centralized ESB pattern had the potential to deliver significant savings in integration costs if interfaces could be reused from one project to the next (the core benefit proposition of SOA). However, coordinating such a cross-enterprise initiative and ensuring that it would get continued funding and that the funding only applied to services that would be sufficiently reusable to cover their creation costs proved to be difficult. Standards and tools were maturing while the ESB patterns were being implemented, so the implementation cost and time for providing a single service were unrealistically high.

Note: ESB patterns have had issues ensuring continued funding for cross-enterprise initiatives because those costs do not apply specifically within the context of a business initiative.

Often, line of business (LOB) teams that were expecting a greater pace of innovation in their new applications became increasingly frustrated with SOA, and by extension the ESB pattern.

Some of the challenges of a centralized ESB pattern were:

- ▶ Deploying changes might potentially destabilize other unrelated interfaces running on the centralized ESB.
- ▶ Servers containing many integrations had to be kept running and patched live wherever possible.
- ▶ Topologies for HA and disaster recovery (DR) were complex and expensive.
- ▶ For stability, servers typically ran many versions behind the current release of software, which reduced productivity.
- ▶ The integration specialist teams often did not know much about the applications they were trying to integrate.
- ▶ Pooling of integration specialists resulted in more waterfall style engagement with application teams.

Note: A *waterfall* methodology is where an attempt is made to gather all the requirements up front, and then the implementation team works in isolation until they deliver the final product for acceptance.

- ▶ Service discovery was immature, so documentation quickly became outdated.

The result was that the creation of services by the SOA team was a bottleneck for projects instead of an enabler, which decreased the value of the ESB pattern.

The term *ESB* is an architectural pattern that refers to the exposure of services. Unfortunately, the term is often over-simplified and applied to the tools that are used to implement the pattern. This situation erroneously ties the static and aging centralized ESB pattern to the integration engines that have changed radically over time.

Modern application integration engines are more lightweight, easier to install and use, and can be deployed in more decentralized ways. As described in 2.4, “The three aspects of agile integration” on page 21, agile integration enables us to overcome the limitations of the ESB pattern by using these more modern integration tools in a model that allows for fine-grained deployment.

For more information about where the ESB pattern came from, and a detailed look at the benefits and the challenges that came with it, see [Digital Developer Conference: AI & Cloud](#).

2.2.3 The API economy

External APIs are an essential part of the online persona of many companies, and they are at least as important as its websites and mobile applications. Let’s take a brief look at how they evolved from mature internal SOA-based services.

SOAP-style RPC interfaces are complex to understand and use, and simpler and more consistent RESTful services that are provided by using JSON and HTTP became popular. But, the goal is the same: Make functions and data available by using standardized interfaces so that new applications can be built on top of them more quickly.

With the broadening usage of these service interfaces, more formal mechanisms for providing services were required. It quickly became clear that simply making something available over a web service interface or as a RESTful JSON and HTTP API was not enough.

The web service needed to be easily discovered by potential consumers who needed a path of least resistance for gaining access to it and learning how to use it. Additionally, the providers of the service or API needed to be able to place controls on its usage, such as traffic control and an appropriate security model. Figure 2-3 on page 12 demonstrates how the introduction of service and API gateways affects the scope of the ESB pattern.

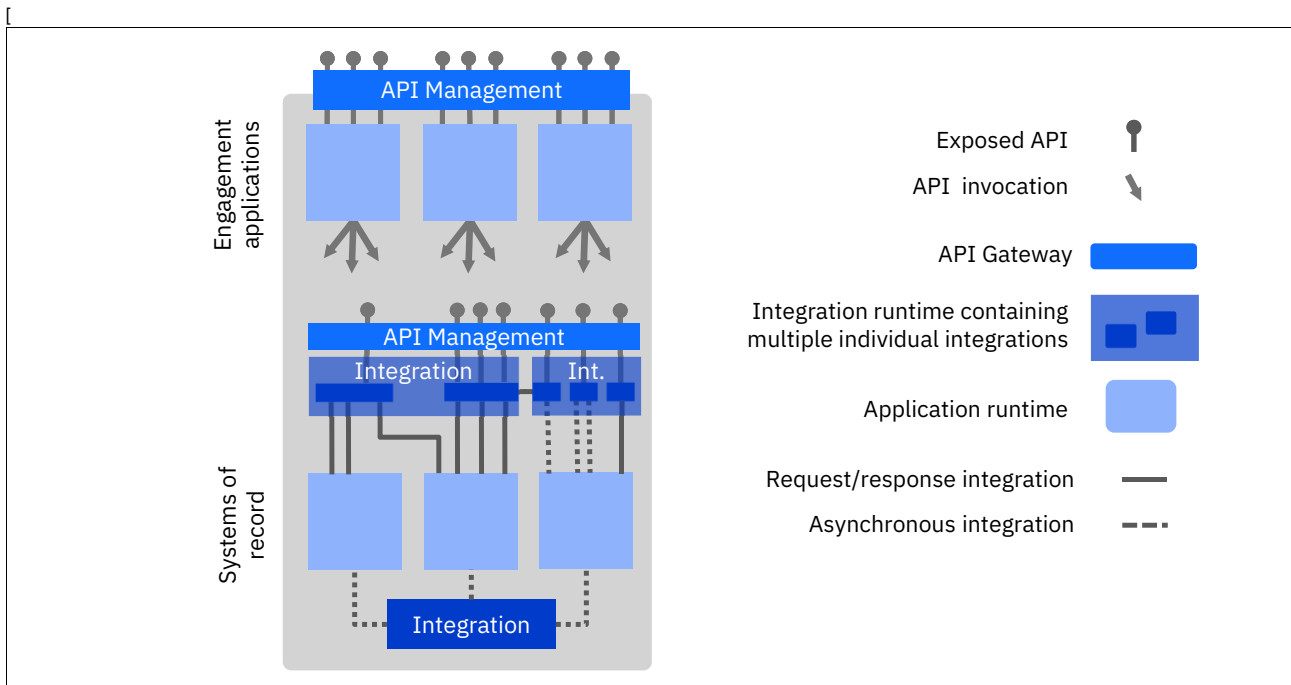


Figure 2-3 Introducing API management

The typical approach was to separate the role of service and API exposure into a separate gateway. These capabilities evolved into what is now known as API management and enabled simple administration of the service and API. The gateways could also be specialized to focus on API management-specific capabilities, such as traffic management (rate and throughput limiting), encryption and decryption, redaction, and security patterns. The gateways could also be supplemented with portals that describe the available APIs that enable self-subscription to use the APIs along with provisioning analytics for both users and providers of the APIs.

While logically the provisioning of APIs outside the enterprise looks like just an extension of the ESB pattern, there are significant infrastructural and design differences between externally facing APIs and internal services and APIs:

- ▶ From an infrastructural point of view, it is obvious that the APIs are being used by consumers and devices that may exist anywhere from a geographical and network point of view. As a result, it is necessary to design the APIs differently to account for the bandwidth that is available and the capabilities of the devices that are used as consumers.
- ▶ From a design perspective, you should not underestimate the difference in the business objectives of these APIs. External APIs are much less focused on reuse, much like internal APIs and services were in SOA, and more focused on creating services targeting specific niches of potential for new business. Suitably crafted channel-specific APIs provide an enterprise with the opportunity to radically broaden the number of innovation partners that it can work with (enabling crowd sourcing of new ideas), and they play a significant role in the disruption of industries. This realization caused the birth of what we call the *API economy*, which is described in [What is an API? and What is the API Economy?](#)

This progression toward more sophisticated API management is for many enterprises still very much “in-progress”. Moving to a cloud infrastructure is more than a replatforming exercise. By committing to a different way of designing applications along with the move to a container infrastructure, you can gain significant benefits in terms of agility, scalability, and resilience. A key element of agile integration is to introduce and evolve the use of API management with a focus on ensuring the APIs are consumer-centric. The cloud-native concept is described in Chapter 4, “Cloud-native concepts and technology” on page 85.

2.3 Microservices

No discussion about the current state of computing is complete without describing *microservices*. What is meant by the term, why is the concept useful, and how does it relate to similar sounding terms such as SOA? After you have a clear conceptual understanding of microservices, you can then appreciate what effect it might have in the integration space.

2.3.1 The rise of lightweight run times

Earlier, we covered the challenges of the heavily centralized integration run time: It is hard to safely and quickly make changes without affecting other integrations, expensive, and complex to scale.

These challenges were the same ones that application development teams were facing at the same time: The presence of bloated, complex application servers that contained too much interconnected and cross-dependent code on a fragile cumbersome topology that was hard to replicate or scale.

Ultimately, it was this common paradigm that led to the emergence of the principles of microservices architecture. As lightweight run times and application servers such as Node.js and IBM WebSphere Application Server Liberty were introduced (run times that started in seconds and had tiny footprints), it became easier to run them on smaller virtual machines (VMs), and then eventually within container technologies such as Docker.

2.3.2 Microservices architecture: A more agile and scalable way to build applications

To meet the constant need for IT to improve agility and scalability, a next logical step in application development was to break up applications into smaller pieces and run them in these new lightweight run times independent of each other. Eventually, these pieces became small enough that they were termed *microservices*.

If you take a closer look at microservices concepts (Figure 2-4), you see that they are more than simply breaking up things into smaller pieces. There are implications for architecture, process, organization, and more. Microservices are focused on enabling organizations to better use cloud-native technology advances to increase their pace of innovation.

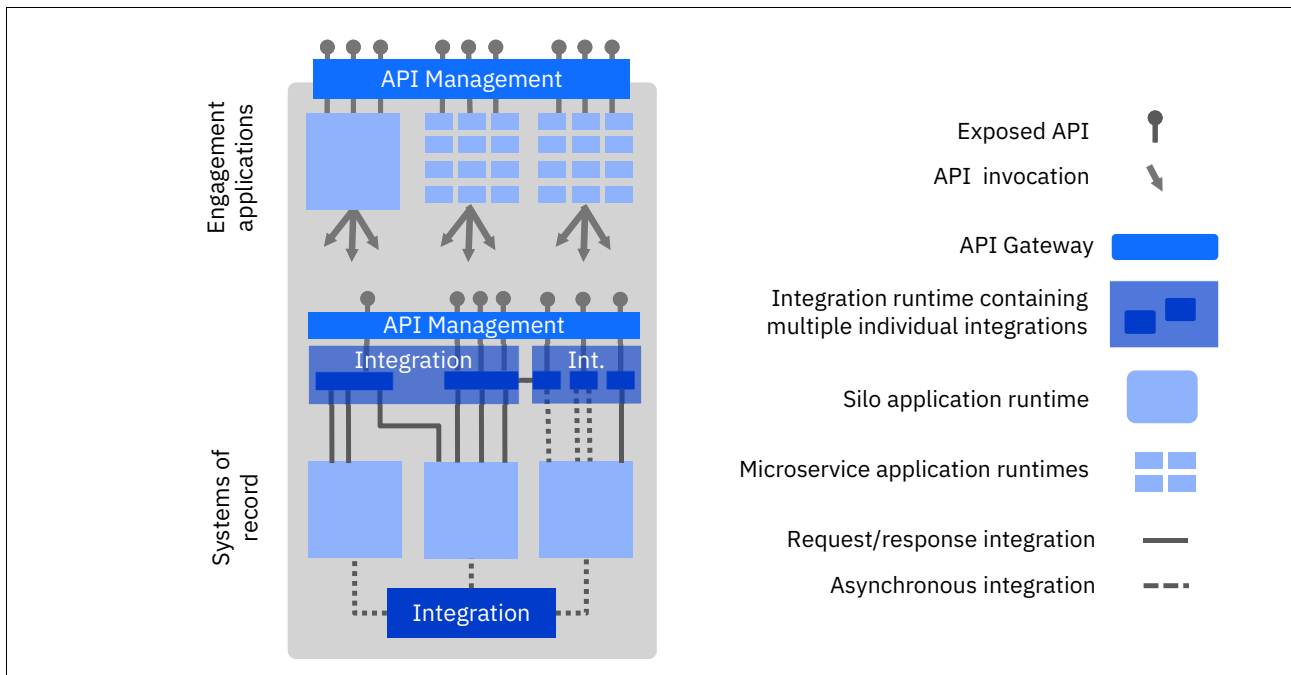


Figure 2-4 Microservices in the engagement layer

However, focusing on the core technological difference, these small independent microservices components can be changed in isolation to create greater agility, scaled individually to better use a cloud-native infrastructure, and managed to provide the resilience that is required by 24 x 7 online applications.

The core benefits of a microservices approach are:

- ▶ Greater agility: They are small enough to be understood in isolation and changed independently.
- ▶ Elastic scalability: Their resource usage can be tied to the business model.
- ▶ Discrete resilience: With suitable decoupling, changes to one microservice do not affect others at run time.

In theory, these microservices principles can be used anywhere. Where you see them most commonly is in the systems of the engagement layer, where greater agility is essential. However, they can also be used to improve the agility, scalability, and resilience of a system of record or indeed anywhere else in the architecture, as you will see as we describe agile integration.

Microservices as an architecture

Microservices is an alternative approach to structuring applications. Rather than an application being a large silo of code all running on the same server, an application is designed as a collection of smaller, independently running components. It could be described as an *application architecture*. As you see later, this is an important distinction when comparing it with SOA.

Microservices components are often made from pure language run times such as Node.js or Java, but they can be made from any lightweight run time. The key requirements include that they have a simple dependency-free installation, file-system-based deployment, start and stop in seconds, and strong support for a container-based infrastructure.

Note: IBM Cloud Pak for Integration, which is described in Chapter 5.1, “IBM Cloud Pak for Integration” on page 144, can fulfill these requirements.

The microservices architecture enables developers to better use cloud-native infrastructure and manage components by providing the resilience and scalability that are required by 24 x 7 online applications. It also improves ownership in line with DevOps practices where a team can take responsibility for a whole microservices component throughout its lifecycle and make changes at a higher velocity.

Microservices considerations

As with any new approach, there are challenges that can be overt or subtle. Microservices are a radically different approach to building applications. Here is a brief look at some of the considerations:

- ▶ Greater overall complexity: Although the individual components are potentially simpler and thus easier to change and scale, the overall application is inevitably a collection of highly distributed individual parts.
- ▶ Learning curve on cloud-native infrastructure: To manage the increased number of components, new technologies and frameworks are required, which include service discovery, workload orchestration, container management, logging frameworks, and more. Platforms are available to make this easier, but it is still a learning curve.

- ▶ Different design paradigms: The microservices application architecture requires fundamentally different approaches to design. For example, considering the use of eventual consistency rather than transactional interactions, or the subtleties of asynchronous communication to decouple components.
- ▶ DevOps maturity: Microservices require a mature delivery capability. Continuous Integration and Continuous Delivery and Deployment (CI/CD) and fully automated tests are a must. The developers who write code must be responsible for it in production. Build and deployment chains need significant changes to provide the right separation of concerns for a microservices environment.

The microservices architecture is not the solution to every problem. Because the microservices approach is complex, it is critical to ensure that the benefits that are outlined above outweigh the extra complexity. In many cases, existing enterprise solutions can and should continue running with a more traditional architecture. The investments have already been made in implementing the resiliency and scalability in the unique models of those tools. However, for suitably selected new solutions or for pockets of modernization within existing systems, microservices can provide an order of magnitude increase in benefits that are hard to achieve any other way.

Without question, microservices principles can offer significant benefits under the correct circumstances. However, choosing the correct time to use these techniques is critical, and getting the design of highly distributed components correct is not a trivial endeavor.

One of your challenges is deciding the shape and size of your microservices components. Another one is the design choices around the extent to which you decouple them. You must constantly balance practical reality with aspirations for microservices-related benefits. *In short, your microservices-based application is only as agile and scalable as your design is good, and your methodology is mature.*

Using microservices techniques in integration

Microservices architecture discussions are often heavily focused on alternative ways to build applications, but the core ideas behind it are relevant to all software components, including integration. Figure 2-5 shows the fine-grained integration deployment.

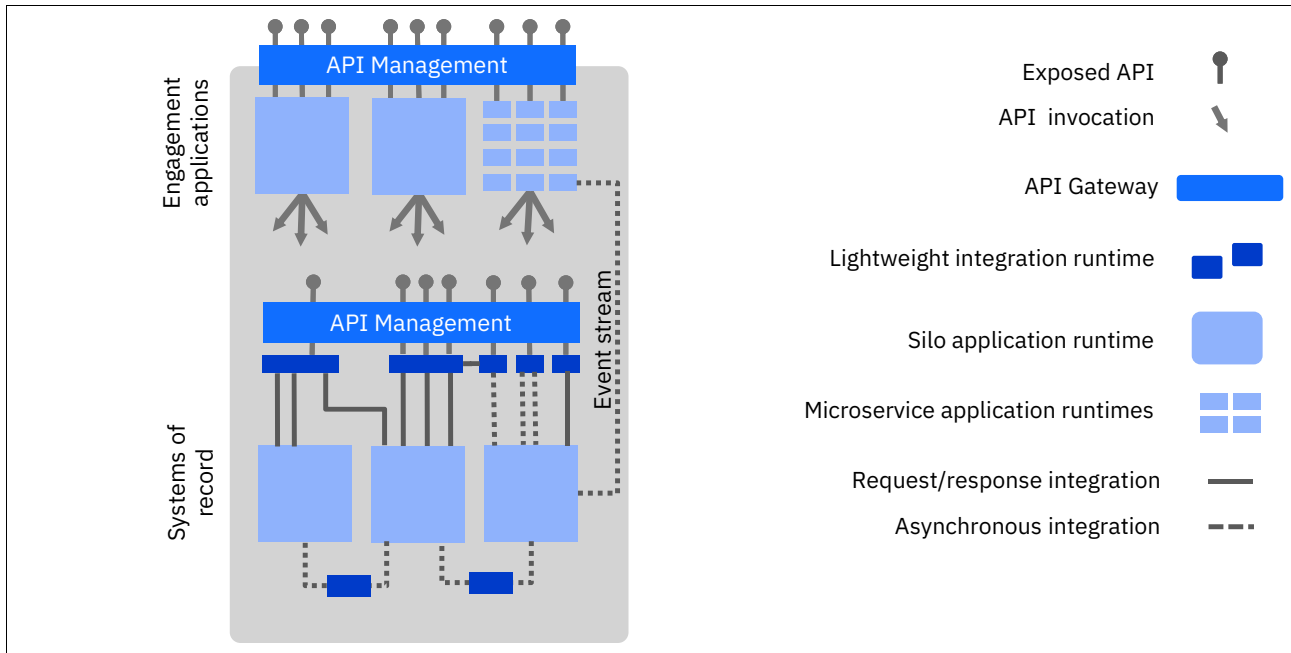


Figure 2-5 Fine-grained integration deployment

Clearly, you want to gain the benefits of agility, scalability, and resilience in your integration landscape, so a logical step is breaking up your centralized ESB pattern into separately deployable integration units. A complete refactoring to fine-grained integrations might be too much for some enterprises, but at least you should consider building new integrations as more granular, separately deployable pieces. We describe this topic in 3.2, “Capability perspective: Application integration” on page 54.

As you progress through this chapter, we consider how the concepts and techniques behind microservices can be used in the breaking up of ESB and the context of API management, and asynchronous communication over messaging, events, and files.

2.3.3 Comparing SOA and the microservices architecture

Microservices inevitably are compared to SOA in architectural discussions because they share many terms. However, this comparison is misleading because the terms apply to two different scopes.

Figure 2-6 on page 18 demonstrates how microservices principles are application-scoped, but SOA is an enterprise-scoped architecture.

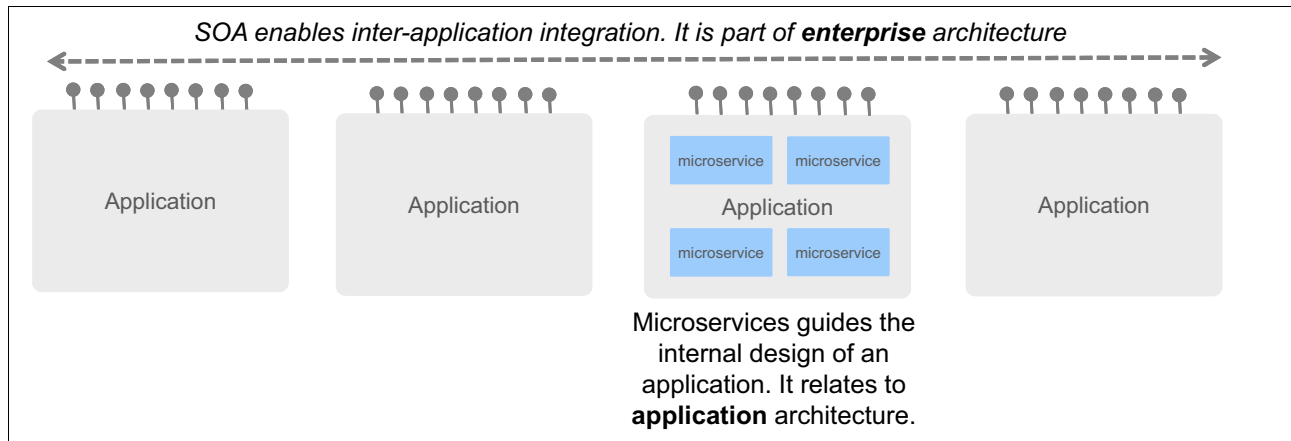


Figure 2-6 Comparing microservices and SOA

SOA is an enterprise-wide initiative to create reusable, synchronously available services and APIs, such that new applications can be created more quickly by incorporating data from other systems.

However, *microservices architecture* is an option for how you might choose to write an individual application in a way that makes that application more agile, scalable, and resilient.

It is critical to recognize this difference in scope because some of the core principles of each approach might be incompatible if applied at the same scope. For example:

- ▶ **Reuse:** In SOA, reuse of integrations is the primary goal, and at an enterprise level, some level of reuse is essential. In microservices architecture, creating a microservices component that is reused at run time throughout an application results in dependencies that reduce agility and resilience. Microservices components generally prefer to reuse code by copy and accept data duplication to help improve decoupling between one another.
- ▶ **Synchronous calls:** The reusable services in SOA are available across the enterprise by using predominantly synchronous protocols such as RESTful APIs. However, within a microservices application, synchronous calls introduce real-time dependencies, resulting in a loss of resilience, and also latency, which impacts performance. Within a microservices application, interaction patterns based on asynchronous communication are preferred, such as event sourcing where a publish/subscribe model is used to enable a microservices component to remain up to date on changes happening to the data in another component.
- ▶ **Data duplication:** A clear aim of providing services in an SOA is for all applications to synchronously get hold of, and change data directly at its primary source, which reduces the need to maintain complex data synchronization patterns. In microservices applications, each microservice ideally has local access to all the data it needs to ensure its independence from other microservices, and indeed from other applications, even if this means some duplication of data in other systems. Of course, this duplication adds complexity, so it must be balanced against the gains in agility and performance, but this is accepted as a reality of microservices design.

So, in summary, SOA has an enterprise scope and looks at how integration occurs between applications. Microservices architecture has an application scope, dealing with how the internals of an application are built. This is a relatively swift explanation of a much more complex discussion. For more information, see [Microservices, SOA, and APIs: Friends or enemies?](#)

However, you have enough of the key concepts to now delve into the various aspects of agile integration.

2.3.4 Bi-modal IT and decentralization

What becomes clear as you read 2.3, “Microservices” on page 13 is a deeper separation between the necessarily slower moving back-end systems of record compared to the radical agility that is required from the systems of engagement. It is hard to force a back-end system of record to move at the pace of innovation. It is much more realistic to enable using of data and functions from that system in new business solutions that are focused on a particular channel. The separation of these steady-speed and high-speed areas of IT is often termed *bi-modal* or *multi-modal* in cases where there are several levels of different velocities of development.

For the fast moving engagement tier to be successful, its teams must work autonomously and make decisions rapidly that are focused primarily on their specific business objective. This is clearly challenging in terms of overall governance of IT, but if done well, it can result in a much greater velocity of change. This distribution of decision-making power down to autonomous teams is known as *decentralization*, which is shown in Figure 2-7 on page 19.

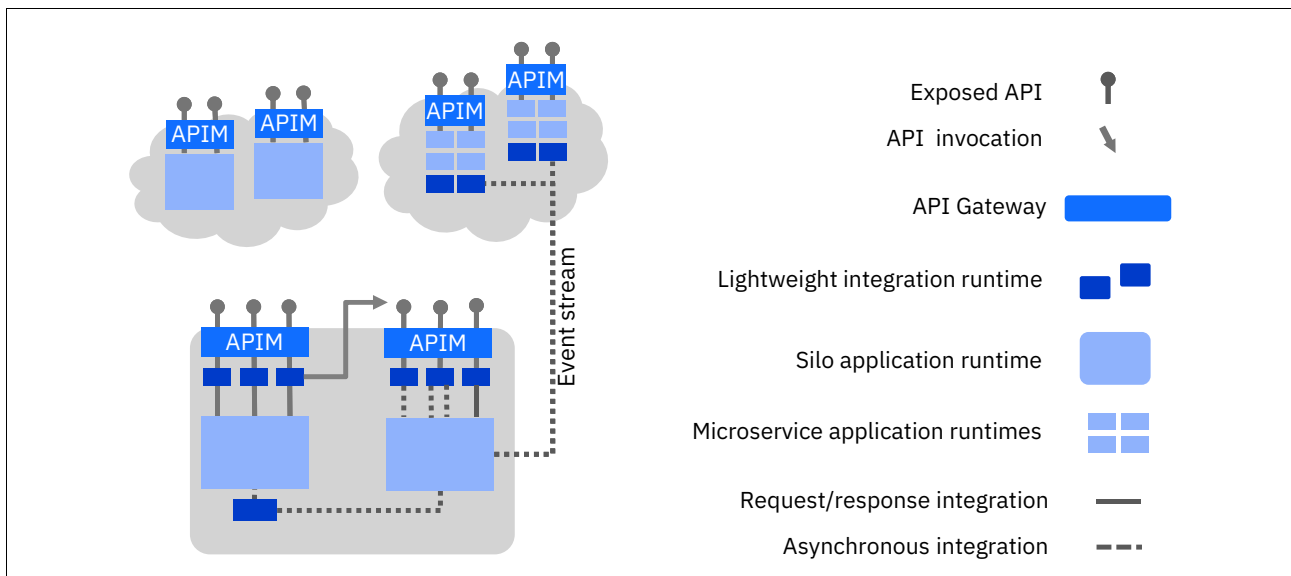


Figure 2-7 Decentralized integration ownership

Integration can help enable this decentralization in many ways, and also help retain distributed governance:

- ▶ From an application integration point of view, you can take the fine-grained integration deployment a step further and distribute the ownership of integrations to application teams. Application integration is described in 3.2, “Capability perspective: Application integration” on page 54.
- ▶ API management is perfectly aligned to enable teams to administer their own APIs rather than needing a central team to perform that role. It can also implicitly enforce company-wide standards on how those APIs are exposed to retain consistency. API management is described in 3.1, “Capability perspective: API management” on page 42.
- ▶ You might choose to use event streams to keep microservices components decoupled from back-end systems based on the modern patterns that are used in the microservices community, such as *event sourcing*. Event streams are described in 3.3, “Capability perspective: Messaging and event streams” on page 68.

2.3.5 Decentralization and integration versus point-to-point

Centralized integration and ESB patterns were introduced to gain control of the proprietary, wasteful, and complex integration that arose from having no common approach to integration. So, how is decentralization and fine-grained integration *not* a point-to-point solution?

Point-to-point solutions had many problems, such as varied interfacing protocols and application platforms that did not have the necessary technical integration capabilities. For each integration between two applications, you had to write new, complex, and integration-centric code for both the service consumer and the service provider.

Now, compare that situation to the modern, decentralized integration pattern. The interface protocols are simplified and rationalized such that many provider applications now offer RESTful APIs or at least web services, and most consumers are equipped to make requests based on those standards.

When applications cannot provide an interface over those protocols, powerful integration tools are available to the application teams to enable them to rapidly develop APIs and services by using primarily simple configurations and minimal custom code.

API-led integration: This is why API-led integration yields such benefits. API-led integration allows organizations to easily connect even the most complex systems and the consistency that they provide by exposing these endpoints as an API.

Along with wide-ranging connectivity capabilities to both old and new data sources and platforms, application integration tools also fulfill common needs such as data mapping, parsing and serialization, dynamic routing, resilience patterns, encryption and decryption, traffic management, security model switching, identity propagation, and more. All these needs are met primarily through simple configuration, which further reduces the need for complex custom code.

Because of the maturity of the complementary API management tools, you can provide those interfaces to consumers and the following functions:

- ▶ Make them easily discoverable by potential consumers.
- ▶ Enable secure, self-administered onboarding of new consumers.
- ▶ Provide analytics to understand usage and dependencies.
- ▶ Promote them as externally facing so they can be used by third parties.
- ▶ Potentially even monetize APIs by treating them as a product that is provided by your enterprise rather than just a technical interface.

In this more standards-based, API-led integration, there is little burden on either side when an application wants to use APIs that are offered from another provider application.

Of course, API management is only part of the picture. API management provides the standardized, secure, and discoverable exposure of an API, but what if the application in question does not provide an API, has the wrong granularity, is overly complicated, or has a complex security model? This is where application integration run times help by providing the tools to perform deep connectivity, unpick complex protocols, and compose multiple requests to produce an API that is appropriate to expose through an API management layer.

It is not point-to-point because this integration and surfacing of the API is done only once on the provider side for a capability. It can then be reused easily by multiple consumers, and its usage can be monitored and controlled in a standardized way.

2.4 The three aspects of agile integration

Over the next few sections, you explore agile integration based on its effects on people, architecture, and infrastructure:

- ▶ People - Decentralized integration ownership: Improve *development agility* by empowering teams with integration capabilities so that they can innovate in real time.
- ▶ Architecture - Delivery focused integration architecture: Improve *deployment agility* by using modern architectural and design practices such as API-led integration, microservices, and event-driven applications apply to integration.
- ▶ Infrastructure - Cloud portable integration infrastructure: Improve *operational agility* by using a platform-neutral, cloud-native approach to integration infrastructure.

2.5 People: Decentralized integration ownership

Any significant change must start with people. If the people are not aligned with a new way of working, they rapidly slip back into the old way of doing things. As ever, you can refer right back to Conway's Law (circa 1967): If we're changing the way we architect systems and we want it to stick, we also need to change the organizational structure.¹

Look at how enterprises are typically aligned from an integration perspective, then consider how you might progressively change them to a more agile approach. Although we propose an alternative model to centralized technology teams, we recognize that this model may not be right for all enterprises, and even within any enterprise it might be right only for some parts of that organization.

¹ [Conway's law](#)

Figure 2-8 on page 22 shows how in a traditional SOA architecture people are centralized into teams that are aligned to their technology stack.

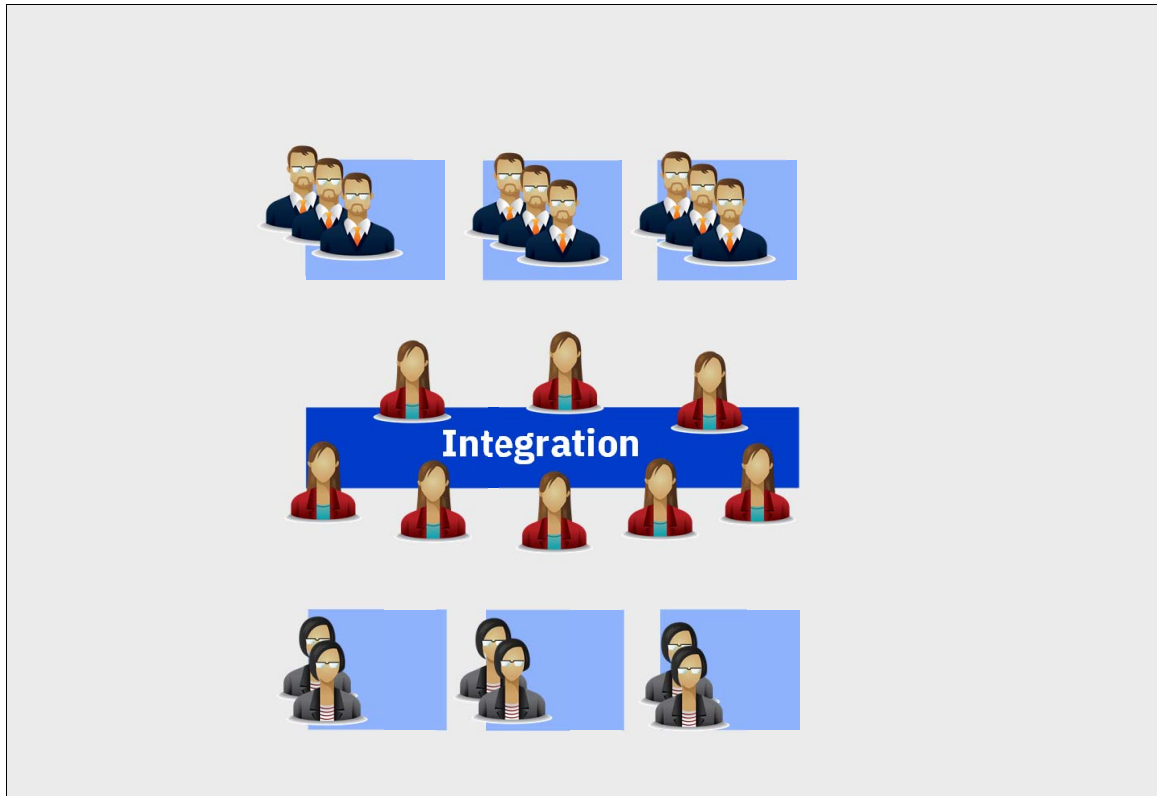


Figure 2-8 Centralized teams

A high-level organizational chart has the following characteristics:

- ▶ A front-end team, which focuses on user experience and creating UIs.
- ▶ An ESB team, which focuses on looking at existing assets that can be provided as enterprise assets. This team also focuses on creating the services that support the UIs from the front-end team.
- ▶ A back-end team, which focus on the implementation of the enterprise assets that are surfaced through the ESB. There are many teams here working on many different technologies. Some might provide SOAP interfaces that are created in Java, some provide COBOL copybooks that are delivered over IBM MQ, others create SOAP services that are exposed by the mainframe, and so on.

This is an organizational structure with an enterprise focus so that a company can rationalize its assets and enforce standards across a large variety of assets. The downside of this focus is that time to market for an individual project is compromised for the good of the enterprise.

A simple example of structure is a front-end team that wants to add a single new element to their screen. If that element does not exist on an existing SOAP service in the ESB, then the ESB team must be engaged, which impacts the back-end team, who also must make a change. The code changes at each level are simple and straightforward so that is not the problem.

The problem is allocating the time for developers and testers to work on it. The project managers must get involved to figure out who on their teams have the capacity to add the new element and how to schedule the push into the various environments. If you scale out, you also have competing priorities. Each project and new element must be vetted and prioritized, and this takes time. So, you are in a situation where there is much impact in terms of time for a simple and straightforward change.

The question is whether the benefits that you get by doing governance and creating common interfaces is worth the price you pay for the operational challenges. In the modern digital world of fast-paced innovation, you must think of a new way to enforce standards while allowing teams to reduce their time to market.

2.5.1 Moving to a decentralized and business-focused team structure

You are trying to reduce the time between the business “ask” and production implementation, knowing that you may rethink and reconsider how you implement the governance processes that were in place. Let’s now consider the concept of microservices, where you have broken down your technical assets into smaller pieces. If you do not consider reorganizing, you might make it worse. You introduce even more handoffs as the lines of what is an application and who owns what blur. You need to rethink how you align people to technical assets. Figure 2-9 on page 23 gives you a preview of what that new alignment might look like.

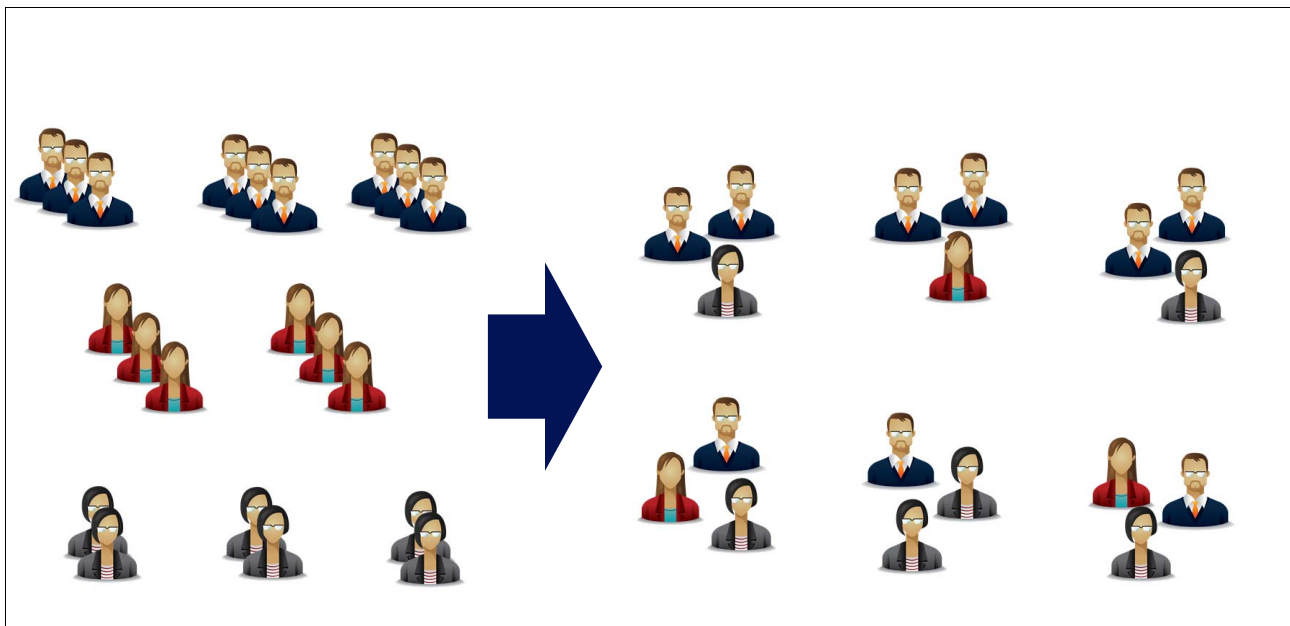


Figure 2-9 Decentralized teams

Instead of people being centrally aligned to the area of the architecture they work on, they are decentralized and aligned to business domains. In the past, you had a front-end team, services teams, back-end teams and so on. Now, you have many business teams, for example, an account team that works on anything that is related to accounts regardless of whether or the accounts involve a REST API, a microservice, or a user interface.

The teams must have cross-cutting skills because their goal is to deliver business results, not technology. To create that diverse skill set, it is natural to start by picking one person from the old ESB team, one person from the old front-end team, and another from the back-end team.

You do not have to do a massive reorg across the entire enterprise: You can do it application by application and piece by piece.

2.5.2 Big bangs generally lead to big disasters

The concept of “big bangs generally lead to big disasters” is not applicable to only code or applications. It is applicable to organizational structure changes. An organization’s landscape is a complex heterogeneous blend of new and old. It might have a “move to cloud” strategy, but is also contains stable heritage assets. The organizational structure continues to reflect that mixture. Few large enterprises have the luxury of shifting entirely to a decentralized organizational structure, and it is not wise to do so.

For example, if there is a stable application and there is nothing major on the roadmap for that application, it does not make sense to decompose that application into microservices or reorganize the team working on that application.

Decentralization need only occur where the autonomy it brings is required by the organization to enable rapid innovation in a particular area.

We certainly do not anticipate reorganization at a company level in its entirety overnight. The point here is that as the architecture evolves, so should the team structure working on those applications and the integration among them. If you look into the concerns and motivations of the people that are involved, they fall into two different groups, as shown in Figure 2-10.

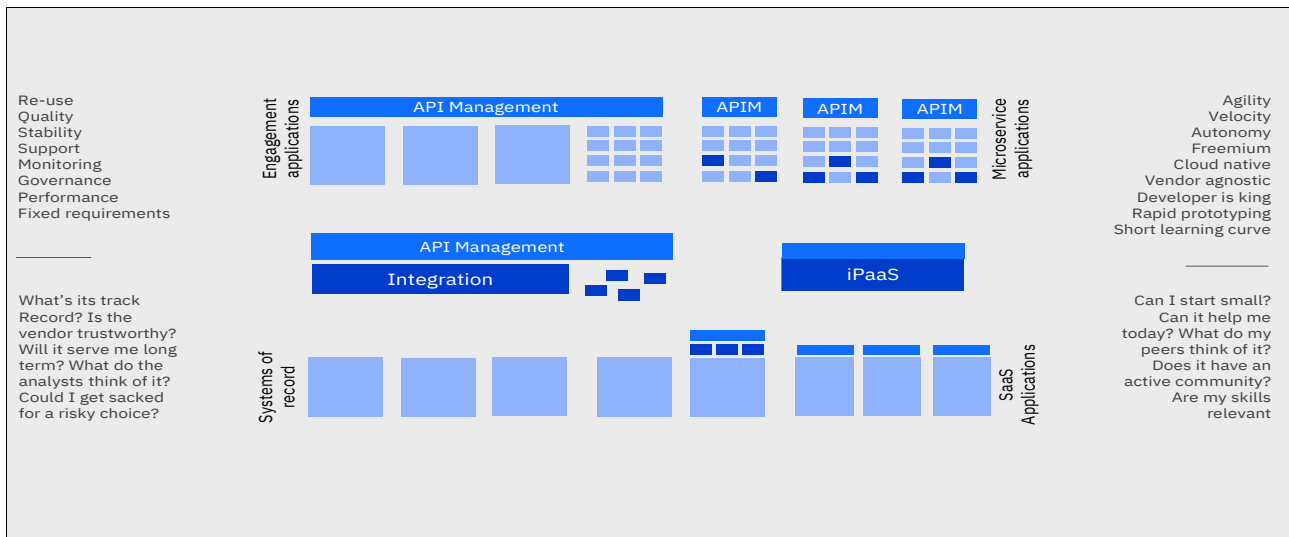


Figure 2-10 Different needs, and different motivations

If the architecture for an application is not changing and not foreseen to change, there is no need to reorganize the people working on that application. However, if you are targeting a significant increase in velocity of changes to production, then both the technology and the associated team structure must change.

2.5.3 Decentralized integration ownership and decentralized infrastructures

To reiterate, *decentralized integration is primarily an organizational change, not a technical one*. But does decentralized integration imply an infrastructure change? Possibly, but not necessarily.

The move toward decentralized ownership of integrations and their exposure does not necessarily imply a decentralized infrastructure. Although each application team clearly could have its own gateways and container orchestration platforms, it is not a certainty. The important thing is that they can work autonomously.

API management is commonly implemented with a shared infrastructure (an HA pair of gateways and a single installation of the API management components) with each application team directly administering their own APIs as though they had their own individual infrastructure. The same implementation can be done with the integration run times by having a centralized container orchestration platform on which they can be deployed and giving application teams the ability to deploy their own containers independently of other teams.

2.5.4 Prioritizing project delivery first

Now let us consider what this change does to an individual and what they are concerned about.

The first thing that you notice about the next diagram is that it shows both old and new architectural styles. This is the reality for most organizations. There are many existing systems that are older and more resistant to change, but are critical to the business. Although some of them might be partially or even completely reengineered or replaced, many remain for a long time. In addition, there is a new wave of applications being built for agility and innovation by using architectures such as microservices. There will be new cloud-based software-as-a-service (SaaS) applications being added to the mix too.

If you look into the concerns and motivations of the people that are involved, they fall into two different groups. A developer of traditional applications cares about stability and generating code for reuse and doing a large amount of up-front due diligence, but the agile teams have shifted to a delivery focus. Now, instead of thinking about the integrity of the enterprise architecture first and being willing to compromise on the individual delivery timelines, they are thinking about delivery first and willing to compromise on consistency.

Note: Agile teams are more concerned with the project delivery than they are with the enterprise architecture integrity.

Let's view these two conflicting priorities as two ends of a pendulum. There are negatives at the extreme end on both sides. On one side, you have *analysis paralysis* where all you are doing is talking and thinking about what you should be doing. On the other side, you have the *wild west* where all you are doing is blindly writing code with no direction or thought towards the longer-term picture. Neither side is correct, and both have grave consequences if allowed to slip too far to one extreme or the other.

There are several questions to consider:

- ▶ If you broke your teams into business domains and they are enabled and focused on delivery, how do you get some level of consistency across all the teams?
- ▶ How do you prevent duplicated effort?
- ▶ How do you gain some semblance of consistency and control while still enabling speed to production?

2.5.5 Evolving the role of the architect

The answer is to consider the architecture role. In the SOA model, the architecture team makes decisions in isolation. In agility integration, the architects are role-practicing architects. An example is shown in Figure 2-11 on page 26.

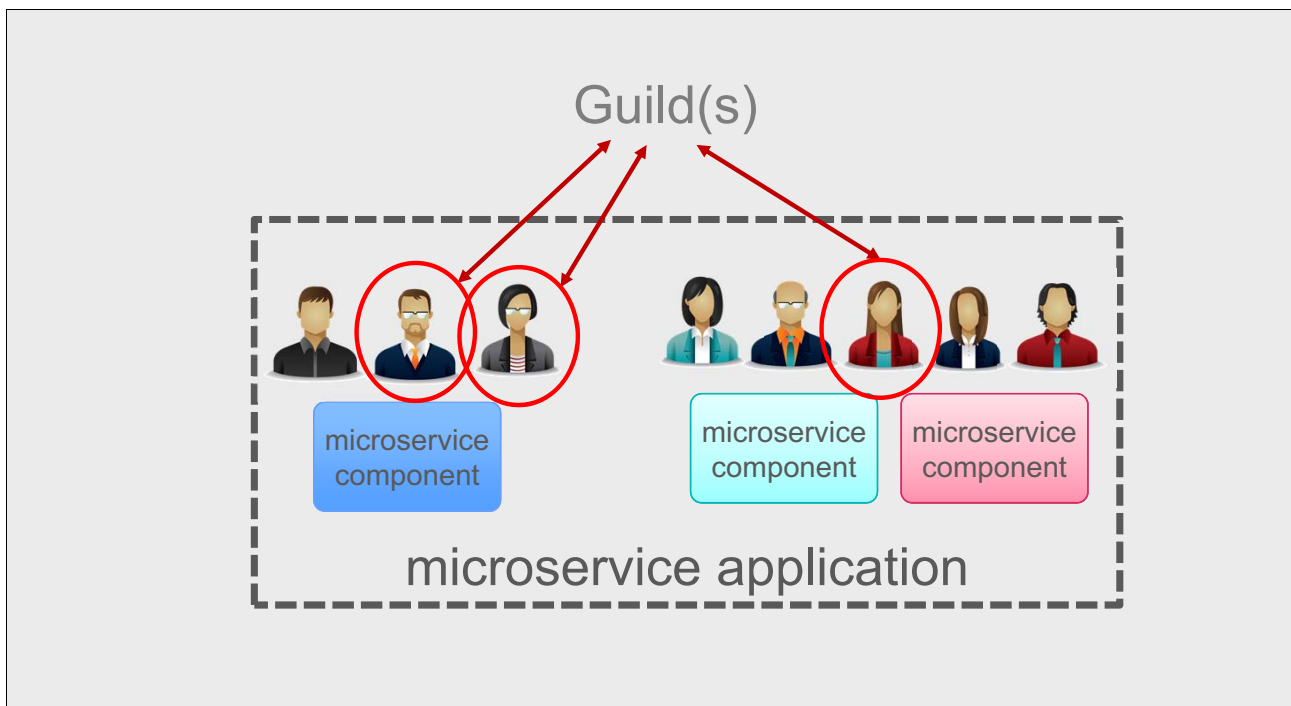


Figure 2-11 Guilds for governance

You have many teams and some of the members have a dual role. On one side, they are expected to be an individual contributor on the team, and on the other side they sit on a committee (or guild) that rationalizes what everyone is working on. They are creating common best practices from their work. They are creating shared frameworks, and sharing their experiences so that other teams do not blunder into traps they already encountered. In the SOA world, it was the goal to stop duplication and enforce standards before development started. In this model, the teams are empowered and the committee or guild's responsibility is to raise, address, and fix cross-cutting concerns during application development.

If there is a downside to decentralization, it is how to govern the multitude of different ways that each application team might use the technology. You encourage standard patterns of use and best practices, but autonomy can lead to divergence anyway.

If every application team creates APIs in their own style and convention, it can become complex for consumers who want to reuse those APIs. With SOA, attempts were made to create rigid standards for every aspect of how the SOAP protocol would be used, which inevitably made them harder to understand and reduced adoption. With RESTful APIs, it is more common to see convergence on conventions rather than hard standards. Either way, the need is clear: Even in decentralized environments, you still need to find ways to ensure an appropriate level of commonality across the enterprise. Of course, if you are already exploring a microservices-based approach elsewhere in your enterprise, then you are familiar with the challenges of autonomy.

Note: The practicing architect is now responsible for the execution of the individual team missions and the related governance requirements that cut across the organization.

Therefore, the practicing architect is now responsible for knowing and understanding what the committee has agreed to, encouraging their team to follow the governance guidelines, bringing up cross-cutting concerns that their team has identified, and sharing what they are working on. This role must be an individual contributor on one of the teams so that they feel the pain or benefit of the decisions made *by the committee*.

2.5.6 Automation: The key to consistency in decentralization

With the concept of decentralization comes a natural skepticism over whether the committee's influence is persuasive enough to enforce the standards they agreed to. Embedding the *practicing architect* into the team might not be enough.

Let's consider how the traditional governance cycle often occurs. It often involves the application team working through complex standards documents and having meetings with the governance board before the intended implementation of the application to establish agreement. Then, the application team proceed to development activities, normally beyond the purview of the governance team. On or near completion and close to the agreed production date, a governance review occurs.

Inevitably, the proposed project architecture and the actual resultant project architecture will be different. If the architecture review board had an objection, there almost certainly is not time to resolve it. Except for extreme issues (such as a critical security flaw), the production date typically goes ahead, and the technical debt is added to an ever-growing backlog.

Placing practicing architects in the teams encourages alignment. However, the architect is now under project delivery pressure, which might mean that they fall into the same trap as the teams originally did where they sacrifice alignment to hit deadlines. What more can you do with the practicing architect role to encourage the enforcement of standards?

The key ingredient for success in modern agile development environment is automation: automated build pipelines, automated testing, automated deployment, and more. The practicing architect must be actively involved in ways to automate the governance, which might be anything, such as automated code review, templates to build pipelines, or standard Helm charts to ensure that the target deployment topologies are homogeneous even though they are independent. In short, the focus is on enforcement of standards through frameworks, templates, and automation rather than through complex documents and review processes. Although the idea of using the technology to enforce standards is not new, the proliferation of open standards in the DevOps tool chain and cloud platforms in general is making it much more achievable.

Let's start with an example. Say that you have microservices components that issue HTTP requests. For every HTTP request, you want to log, by using a common format, how long that HTTP transaction took and the HTTP response code. Now, if every microservice did this differently, there would not be a unified way of looking at all traffic. Another role of the practicing architect is to build helper artifacts that are used by the microservices. Instead of the governance process being a gate, it is an accelerator because the architects are embedded in the teams, and work on code alongside of them. Now, the governance cycle is done with the teams, and instead of reviewing documents, the code is the document and the checkpoint is to make sure that the common code is being used.

Not all teams are created equally. Some teams are creaking APIs like a factory, some are thinking ahead to upcoming challenges, and some teams are a mix of the two. An advanced team that succeeds in finding a way to automate a particular governance challenge is a much more successful evangelist for that mechanism than any attempt to create it by a separate governance team.

Regarding the technical architect, it might seem that they have too many responsibilities. They are responsible for application delivery and a part of the committee, and now you are adding the additional responsibility of writing common code that is to be used by other application development teams. Is it too much?

A common way to offload some of that work is to create a dedicated team that is under the direction of the practicing architect who is writing and testing this code. The authoring of the code is not a huge challenge, but the testing of that common code is. The reason for placing a high value on testing is because of the potential impact to break or introduce bugs into all the applications that use that code. For this reason, extra due diligence and care must be taken to justify the investment in the additional resource allocation.

Your aim should be to ensure that general developers in the application teams can focus on writing code that delivers business value. With the architects writing or overseeing common components that naturally enforce governance concerns, the application teams can spend more of their time on value and less in governance sessions. Governance based on complex documentation and heavy review procedures is rarely adhered to consistently, but inline tools that are based on standardization happen more naturally.

2.5.7 Producing multi-skilled developers

Developers are expected and encouraged to be full stack developers and solve the business problem with whatever technology is required. This situation puts an incredible strain on each individual developer in terms of the skills that they must acquire. It is not possible for the developer to know every detail of each technology, so the infrastructure learning curve suffers. You are finding better ways to make infrastructural concerns the same among all products.

In the pre-cloud days, developers had to learn multiple aspects of each technology, as shown in Figure 2-12 on page 29.

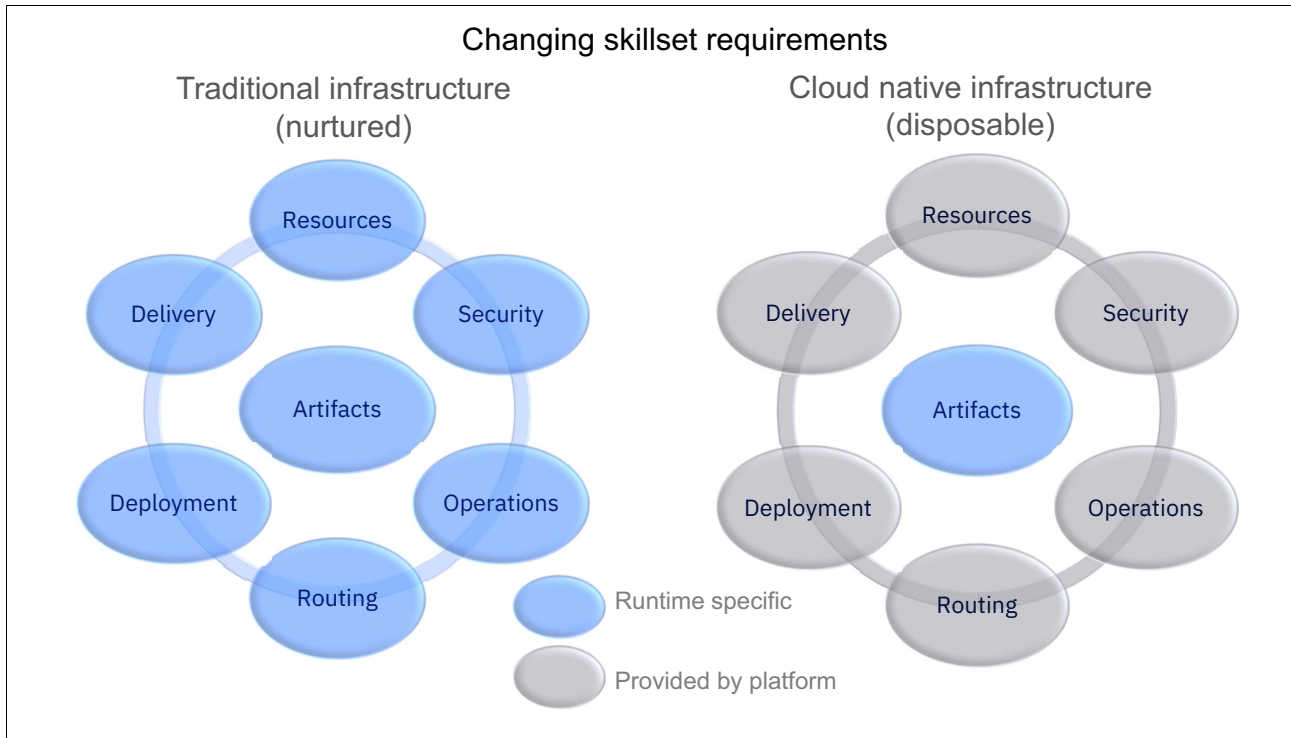


Figure 2-12 Changing skillset requirements

Note: With decentralization, developers can focus on what their team is responsible for, which is delivering business results by creating artifacts.

Each ellipse represents an area that the developer had to know and care about and understand the implications of their code on. They had to know for each technology how to install it; how many resources it needed; how to cater for HA, scaling, and security; how to create the artifacts; how to compile and build them; where to store them; how to deploy them; and how to monitor them at run time. All of these aspects were unique and specific to each technology, which means that you had to have technology-specific teams.

However, the common capabilities and frameworks of typical cloud platforms attempt to take care of many of those concerns in a standardized way so that the developer can focus on what their team is responsible for, which is delivering business results by creating artifacts.

The gray area represents areas that still need to be addressed but are now no longer at the front of the developer's mind. Standardized technology such as (Docker) containers, orchestration frameworks such as K8s, and routing frameworks such as Istio enable management of run times in terms of scaling, HA, deployment, and so on.

Furthermore, standardization in the way products present themselves by command-line interfaces (CLIs), APIs, and simple file system-based installation and deployment means that standard tools can be used to install, build, and deploy.

2.5.8 Conclusions on decentralized integration ownership

Decentralization does not fit every situation. It might work for some or parts of some organizations, but not for others. Application teams for older applications might not have the correct skill sets to take on the integration work. Integration specialists might need to be seeded into their team. This approach is a tool for potentially creating greater agility for change and scaling, but what if the application has been largely frozen for some time?

At the end of the day, some organizations find it more manageable to retain a more centralized integration team. The decentralized approach should be applied where the benefits are needed the most. That said, this style of decentralized integration is what many organizations and indeed application teams have always wanted to do, but they might have had to overcome certain technological barriers first.

The core concept is to focus on delivering business value and a shift from a focus on the enterprise to a focus on the developer. This concept has manifested in the movement from centralized teams to more business-specific ones, but also by more subtle changes such as the role of a practicing architect.

This concept is also rooted in technology improvements that are taking concerns away from the developer and dealing with them uniformly by using a cloud platform.

2.6 Architecture: Delivery-focused architecture

In agile integration, the architectural patterns are tuned towards enabling the business to deliver changes to production robustly at high velocity with an optimal use of resources. The integration occurs differently depending on each of the integration capabilities of API management, application integration, and events and messaging. Each capability changed to complement the other aspects of decentralized ownership and capitalize on cloud-native infrastructure.

Figure 2-13 on page 30 shows a summary of delivery-focused architecture concerns.

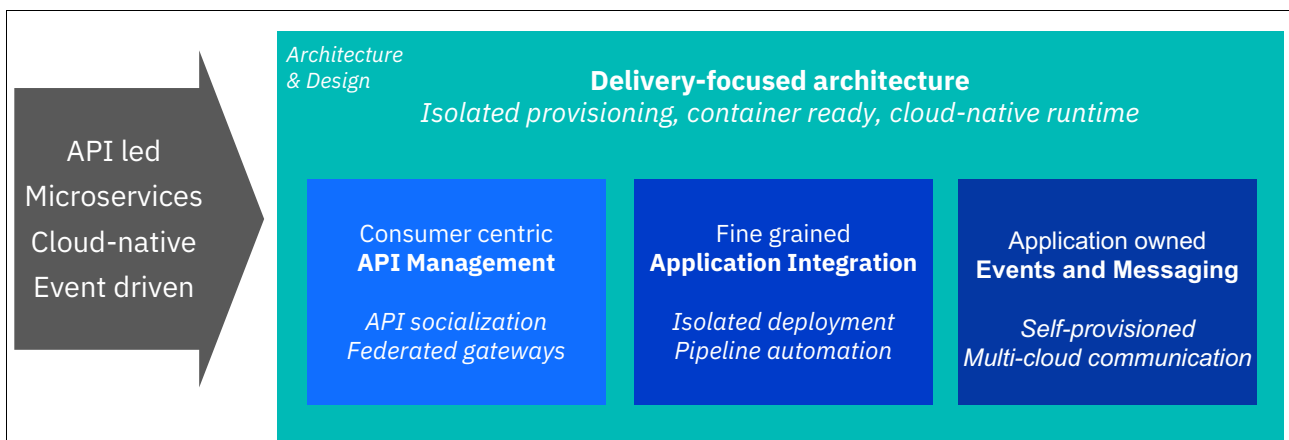


Figure 2-13 Summary of delivery-focused architecture concerns

2.6.1 Consumer-centric API management

The need to standardize the way applications interact with one another has been maturing for decades. We are now at a point where common standards, mostly based on the OpenAPI specification, are becoming almost ubiquitous for new interfaces.

However, to simply provide a technical interface, such as an API or a web service, is not enough to ensure that you get good value from it. You need a consumer-centric approach. You must ensure that the API is easy to find and appealing to use. The API should be treated more like a product that you want to market rather than a technical asset. This approach is clear for APIs that are exposed externally to the organization, and even monetized to create a source of revenue for the company.

However, you should also take a similar approach to APIs within the organization. APIs from one part of the organization should make themselves as appealing as possible to consumers from other parts of the organization. From a technical standpoint, to enable this consumer-centric approach, you need an API management capability.

API socialization

A key element of how you make APIs more consumer-centric is how you socialize them. Consumers of the interface must be able to easily discover it and subscribe to it. Providers must know who is using it and control their behavior. The broader business must have confidence that the API usage is suitably secured for each consumer group, and they might also want to monetize it in various ways. With API management, decentralized teams can administer their own APIs and provide consumers with developer portals for discovery and self-subscription.

Federated gateways

The implementations behind APIs are in many different platforms that potentially are in different clouds, and can even be run and managed by different cloud vendors. Although it might sometimes be possible to have a single gateway (for example, behind the firewall and used for internal APIs) route to all these different places, at a certain scale it becomes more appropriate to let each cloud location have its own gateway facility. However, to remain consumer-centric, you need to ensure a single point for discovery and subscription.

Furthermore, you want to ensure that you have consistent governance of APIs across all gateways, especially in terms of security policies. So, you need a topological model where the socialization aspects of APIs are provided by centralized, multi-tenant components but this centralized infrastructure controls many separate federated gateways. With this mode, you have the ideal balance between consumer convenience, physical optimization, and consistent governance.

Conclusions on consumer-centric API management

APIs are products that are marketed by the company providing them. For many companies, they are now more important than their traditional web presence. API management capabilities provide a set of features that focus on how APIs are exposed, and are fundamentally targeted at improving the experience for the consumer. However, you need an effective API strategy to ensure that the right APIs are chosen and are designed in a way that makes them attractive to potential consumers.

Moving to a cloud infrastructure is more than just a replatforming exercise. By committing to a different way of designing applications along with the move to a container infrastructure, you can gain significant benefits in terms of agility, scalability, and resilience. For more information, see Chapter 4, “Cloud-native concepts and technology” on page 85.

2.6.2 Fine-grained application integration

If the large centralized ESB pattern containing all the integrations for the enterprise is reducing agility, then why not break it up into smaller pieces? If it makes sense to build applications in a more granular fashion (for example, microservices), why not apply this idea to integration too? You could break up the enterprise-wide centralized ESB component into smaller, more manageable, and dedicated components. You might even be able to break it down to one integration run time for each interface you expose, although in many cases it is sufficient to group the integrations per component.

Fine-grained integration deployment draws on the benefits of a microservices architecture, that is, agility, scalability, and resilience:

- ▶ **Greater agility:** Different teams can work on integrations independently without deferring to a centralized group or infrastructure that can become a bottleneck. Individual integration flows can be changed, rebuilt, and deployed independently of other flows, enabling safer application of changes and maximizing speed to production.
- ▶ **Elastic scalability:** Individual flows can be scaled on their own so that you can take advantage of efficient elastic scaling of cloud infrastructures.
- ▶ **Discrete resilience:** Isolated integration flows that are deployed in separate containers cannot affect one another by stealing shared resources, such as memory, connections, or processors.

Figure 2-14 on page 32 shows the result of breaking up the ESB into separate, independently maintainable and scalable components.

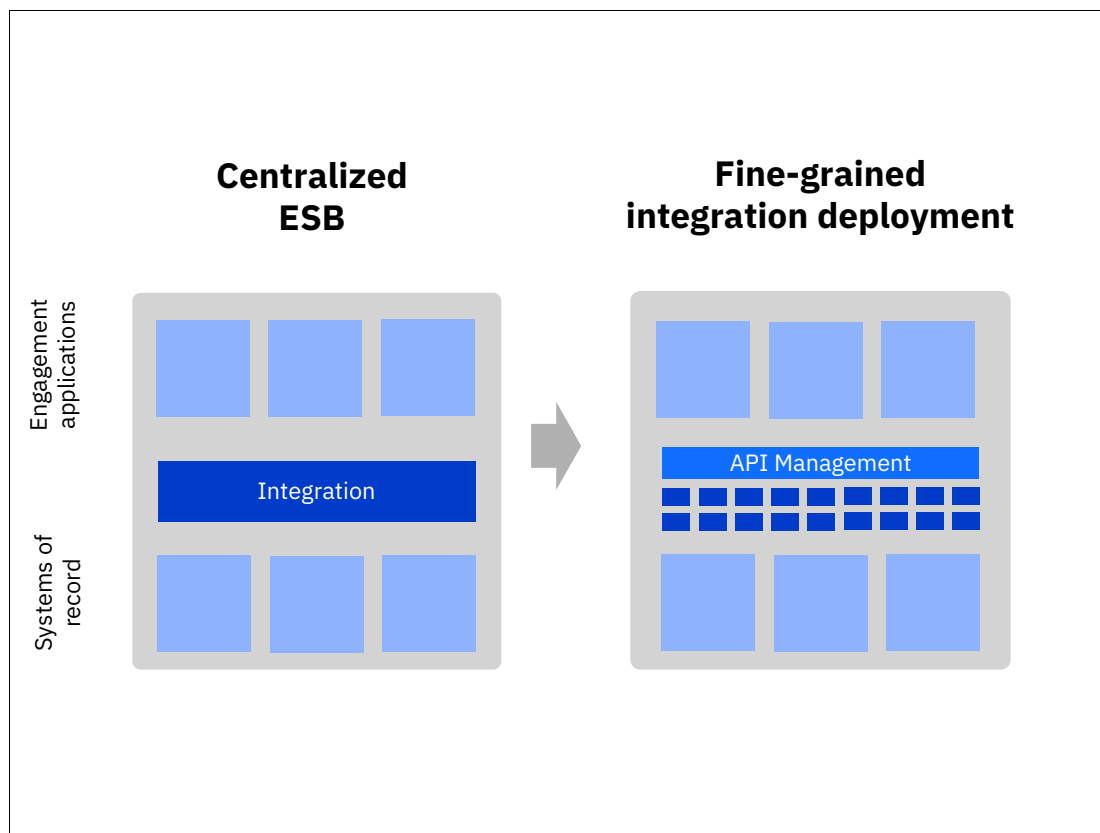


Figure 2-14 Breaking up the ESB

The heavily centralized ESB pattern can be broken up in this way, and so can the older hub-and-spoke pattern. This change makes each individual integration easier to change independently, and improves agility, scaling, and resilience.

Note: With fine-grained integration deployment, you can change an individual integration with complete confidence that you will not introduce any instability into the environment.

With this approach, you can change to an individual integration with complete confidence that you will not introduce any instability into the environment on which the other integrations are running. You can choose to use a different version of the integration run time, perhaps to take advantage of new features, without forcing a risky upgrade to all other integrations. You can scale up one integration independently of the others, making efficient use of the infrastructure, especially when using cloud-based models.

There are considerations with this approach, such as the increased complexity. Also, although this result can be achieved by using VM technology, the long-term benefits are greater if you use containers such as Docker and orchestration mechanisms such as K8s. Introducing new technologies to the integration team adds a learning curve. However, these challenges are the same ones that an enterprise already faces if they are exploring microservices architecture in other areas so that expertise might exist within the organization.

We typically call this pattern *fine-grained integration deployment* (a key aspect of agile integration) to differentiate it from other microservices application architectures. We also want to distinguish it from ESB, which is associated with the centralized integration architecture.

Conclusion on fine-grained integration deployment

With fine-grained deployment, you can reap some of the benefits of microservices architecture in your integration layer by enabling greater agility because of infrastructural decoupled components, elastic scaling of individual integrations, and an inherent improvement in resilience from greater isolation. For more information, see 3.2, “Capability perspective: Application integration” on page 54.

2.6.3 Application-owned messaging and events

Synchronous interaction patterns over HTTP are ubiquitous both within the enterprise and across the internet. From an application architecture point of view, assume that the infrastructure for them is present. However, this situation is still not true for asynchronous patterns that use events and messaging.

Asynchronous communication typically requires the storage of state for a period, and someone must own the storage of that state. For these reasons, asynchronous communication is not as commonly available as stateless HTTP communications. Asynchronous patterns must be supported by explicitly installed and managed components, which has been an inhibitor to agility in the past.

It was a relatively expensive and specialist task to configure and administer these asynchronous communication infrastructures. They tended to be centrally owned, and even the provisioning of a new queue or topic might involve specialists. This situation does not work efficiently with a decentralized and agile development where teams want to be self-sufficient.

Self-provisioning

Asynchronous communication capabilities must be reengineered to radically change the way they can be provisioned and maintained so that an application team that requires a queue or a new event topic can create one themselves and make it part of their pipeline.

Multi-tenant

Despite the fact that you are aiming for greater team autonomy, it does not necessarily follow that there must be separate application-owned infrastructure components. There are still options so that you can best use the underlying resources. Technically, you may create a fresh infrastructure for the queues and topics that are needed by an application, but that makes sense only at a certain scale. You also must be able to provision these capabilities easily on a multi-tenant infrastructure, whether self-managed, cloud-managed, or even on an appliance. Having a choice of consistency is critical to meet the different needs of the applications.

Multicloud

This multi-tenant aspect becomes more important in a multicloud environment where asynchronous communication is required among many different cloud domains that are run by different cloud vendors. Being able to use capability on any cloud makes it easier to rapidly satisfy the needs of teams, projects, and solutions that are distributed across multiple cloud domains.

Conclusions on application-owned messaging and events

The focus for events and messaging is on enabling them to be more *application-owned* capabilities that can be self-provisioned and self-administered by the application teams themselves.

There is still a place for centrally provisioned messaging and events infrastructure, such as for high-performance and critical use cases. In these cases, the goal is to use those same self-provisioning features to automate the standardized aspects of this role by using templates so that more time can be spent on more critical activities such as performance tuning.

For more information, see 3.3, “Capability perspective: Messaging and event streams” on page 68.

2.6.4 Conclusions on delivery-focused architecture

In each technology area, you saw that the architectural choices must be heavily centered on how to ensure more effective delivery of solutions:

- ▶ APIs are built with the needs of consumers in mind, not just as technical interfaces.
- ▶ Integrations are fine-grained so that they can be built and deployed in isolation.
- ▶ Messaging and events capabilities are self-provisionable by application teams.

2.7 Technology: Cloud-native infrastructure

To be effective in moving to an agile integration, you must take full advantage of the advances in modern infrastructure platforms. They can provide significant benefits in the efficiency of resources long term, and they offer superior approaches to the deployment and management of components.

Cloud in this context can be misleading because it does not necessarily imply deployment to a public cloud infrastructure. The core difference is that in this case you are building highly stateless, disposable, and lightweight components. This design ensures that cloud-style infrastructures such as containers and potentially serverless can provide maximum benefit.

This is not just a replatforming exercise: It means building components differently. You are using a *cloud-native approach* to the design and configuration of components that you deploy.

2.7.1 Virtual machines, containers, and serverless

Containers are the most common approach to software virtualization for new solutions, but they are a step within a longer evolution.

Virtualization hides software from its physical computing environment by putting it into a software wrapper, which makes it more portable. Virtualization began with the abstraction of operating systems from the hardware on which they run (VMware). The provisioning of infrastructure by using VMs is ubiquitous, and it is now the exception for software to be installed directly on baremetal servers. Containers take virtualization further by enabling a more fundamental abstraction from the underlying infrastructure and providing a more lightweight and portable virtualization model.

You should expect to see this journey towards greater abstraction from the operating system too (as provided by containers). As shown in Figure 2-15, abstraction will continue to the point where you abstract the visibility of the language and product run times themselves, as in serverless computing or *function-as-a-service (FaaS)*.

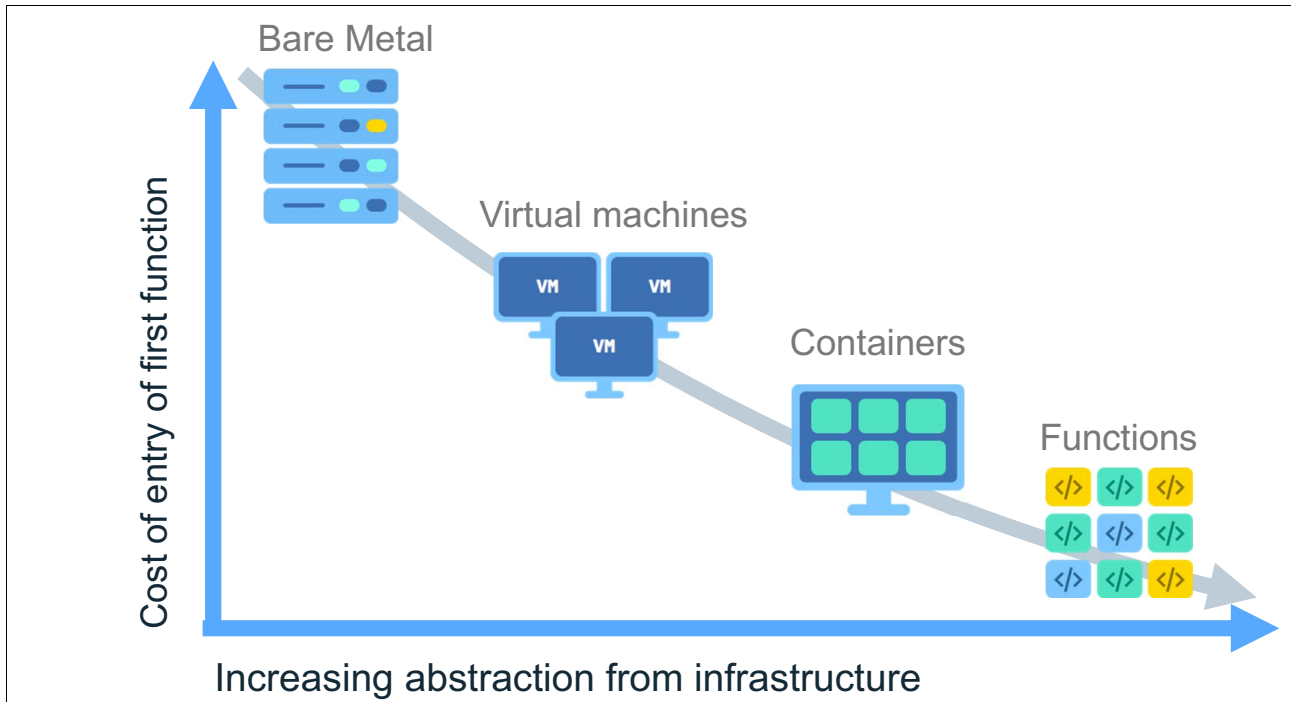


Figure 2-15 The road to containers

We dedicate several sections to container technology in 2.7.2, “Cloud-native approach” on page 36, so for now we describe the cloud-native approach.

2.7.2 Cloud-native approach

Cloud-native is a collection of common practices for creating agile, scalable, and resilient applications. Some of the key practices are:

- ▶ Modular components
- ▶ Stateless components
- ▶ Image-based deployment
- ▶ Lightweight run times
- ▶ Elastic, agnostic infrastructure
- ▶ Log-based monitoring
- ▶ API-led intra-app communication
- ▶ Event-driven architecture
- ▶ Agile methods
- ▶ CI/CD
- ▶ DevOps

Chapter 4, “Cloud-native concepts and technology” on page 85 describes these concepts, so here we describe a common analogy that is used to articulate the behavior of components that are designed in a cloud-native style.

Livestock not pets

Let's take a brief look at where the concept of *livestock not pets* came from before we describe how to apply it in the integration space.

Note: This phrase often is referred to as the [cattle not pets](#) approach. However, because cows are considered sacred in some cultures, in this book we have chosen to use the term “livestock” in place of cattle to refer to animals reared in large herds rather than nurtured individually.

When servers took weeks to provision and minutes to start, it was fashionable to boast about how long you could keep your servers running without failure. Hardware was expensive, and the more applications that you could pack onto a server, the lower your running costs were. HA was handled by using pairs of servers, and scaling was vertical by adding more cores to a machine. Each server was unique, precious, and treated like a *pet*.

Times changed, and hardware is now virtualized. With container technologies such as Docker, you can reduce the surrounding operating system to a minimum so that you can start an isolated process in seconds at most.

Using a cloud-based infrastructure, scaling can be horizontal by adding and removing servers or containers and adopting more cost-effective pricing models. You can now deploy thin slivers of application logic on minimalist run times into lightweight independent containers. Furthermore, rather than running a pair of servers as with traditional HA, it is trivial to run many servers in containers to limit the effects of one container failing. By using container orchestration frameworks such as K8s, you can introduce or dispose of containers rapidly to scale up or down workload. These containers are treated more like a herd of *livestock*.

Integration pets: The traditional approach

Let's examine what the common “pets” model looks like. In the analogy, if you view a server (or a pair of servers that attempt to appear as a single unit) as indispensable, it is a pet. In the context of integration, this concept is similar to the centralized integration topologies that the traditional approach used to solve enterprise application integration (EAI) and SOA use cases.

Table 2-1 on page 37 shows the characteristics of pets.

Table 2-1 Characteristics of pets

Characteristics of pets	How they are applied to a centralized or traditional integration context
Manually built	Integration hubs are often built only one time in the initial infrastructure stage. Scripts help with consistency across environments, but are mostly run manually.
Managed	The hub and its components are directly and individually monitored during operation with role-based access control (RBAC) to allow administrative access to different groups of users.

Characteristics of pets	How they are applied to a centralized or traditional integration context
Hand-fed	The hub is nurtured over time, for example, by introducing new integration applications, and changes to OS and software maintenance levels. As part of this process, new options and parameters are applied, changing the overall configuration of the hub. Thus, even if the server started out being based on a defined pattern, gradually the running instance becomes more bespoke with each change in comparison to the original installation.
Server pairs	Typically, pairs of nodes provide HA. Great care is taken to keep these pairs running and back up the evolving configuration. Scalability is coarse-grained and achieved by creating more pairs or adding resources so that existing pairs can support more workloads.

Table 2-2 shows the characteristics of livestock.

Table 2-2 Characteristics of livestock

Characteristics of livestock	How they are applied to an agile integration context
Elastic scalability	Integrations are scaled horizontally and allocated on demand in a cloud-like infrastructure.
Disposable and recreatable	Using lightweight container technology encourages changes to be made by redeploying amended images rather than by nurturing a running server.
Starts and stops in seconds	Integrations are run and deployed as more fine-grained entities and take less time to start.
Minimal interdependencies	Unrelated integrations are not grouped. Functional and operational characteristics create colocation and grouping.
Infrastructure as code	Resources and code are declared and deployed together.

2.7.3 Portability: Public, private, and multicloud

One of the major benefits of using a cloud-native architecture is portability. The goal of many organizations is to run containers anywhere and move freely between a private cloud, various vendors of a public cloud, or a combination of them.

Cloud-native platforms must ensure compatibility with standards such as OpenAPI, Docker, and K8s if this portability is to be a reality for consumers. Run times must be designed to take full advantage of the standardized aspects of the platforms.

An example might be data security. Assume that a solution has sensitive data that must remain on-premises at this point. However, regulations and cloud capabilities might mature such that it might move off-premises at some point. If you use cloud-native principles to create your applications, then you have much greater freedom to run those containers anywhere.

Other examples might include development and testing in one cloud environment and production in a different one, or using a different cloud vendor for a DR facility.

Whatever the reason, we are at a point where applications can be more portable than ever before. And this fact also applies to the integrations that enable us to use their data.

These integrations must be deployable to any cloud infrastructure and enable the secure and efficient spanning of multiple cloud boundaries.

2.7.4 Conclusion on cloud-native integration infrastructure

Moving to a cloud infrastructure is more than a replatforming exercise. By committing to a different way of designing applications along with a move to a container infrastructure, you can gain significant benefits in terms of agility, scalability, and resilience. For more information about cloud-native, see Chapter 4, “Cloud-native concepts and technology” on page 85.



Agile integration: Capability perspectives

This chapter describes agile integration from the perspective of the integration capabilities: API management, application integration, and messaging and events. You see how the concepts that are described in Chapter 2, “Agile integration” on page 5 fit into each of these technology areas.

The following topics are covered in this chapter:

- ▶ Capability perspective: API management
- ▶ Capability perspective: Application integration
- ▶ Capability perspective: Messaging and event streams
- ▶ Capability perspective: Files and **Business-to-Business**
- ▶ Hybrid and multicloud considerations
- ▶ Use cases driving hybrid and multicloud adoption

3.1 Capability perspective: API management

Let's briefly consider how API management (or sometimes called *Full Lifecycle API management*) relates to the three aspects of agile integration:

- ▶ People and process - Decentralized ownership: API management's goal is to enable both API providers and API consumers to be as autonomous as possible. Ideally, it enables them to configure, expose, discover, and consume APIs without relying on the enterprise architecture community.
- ▶ Architecture and design - Delivery led architecture: API management is in itself an architectural pattern that better enables delivery by being *consumer-centric* at its core, which means that API management is primarily concerned with enabling APIs for consumers. As such, it provides discovery, self-subscription, traffic management, secure connectivity, metrics, and more to ensure that the API is seen as needed by consumers. The API is a product in its own right.
- ▶ Technology and infrastructure - Cloud-native infrastructure: There are two sides to this aspect. The API management capability itself must be able to run on a cloud-native infrastructure such as containers if required. However, there is a more subtle side to this aspect in that API-based communication is fundamental to cloud-native applications. So, we look at when and where it is appropriate to insert an API management gateway into, for example, a microservices application.

The act of introducing API management is fundamental to integration modernization. However, even though the concepts of API management are relatively mature, it does not mean every enterprise has implemented it. Many organizations are on a digital transformation journey but they are starting within a service-oriented architecture (SOA). Their challenge is knowing what it will look like to add API management their environment. So, we are going to begin with a rough history that is similar to the one Chapter 2, "Agile integration" on page 5, but with a particular focus on the evolution of API management. This history helps you see where and why enterprises might be where they are today and what the future might look like for them.

3.1.1 A brief history of API management

When synchronous web services were made available when SOA was common, it became clear that providing an interface for consumers to call was only part of the story. Consumers needed to find that interface, learn how to use it, and self-subscribe so that they could use it securely.

Service-oriented architecture

SOA initially addressed this challenge by adding a service registry (Figure 3-1), which was a separate component of the integration run time that was used to implement the web service. It was hard to keep the definitions in the registry synchronized with the implemented interfaces of the integration run time.

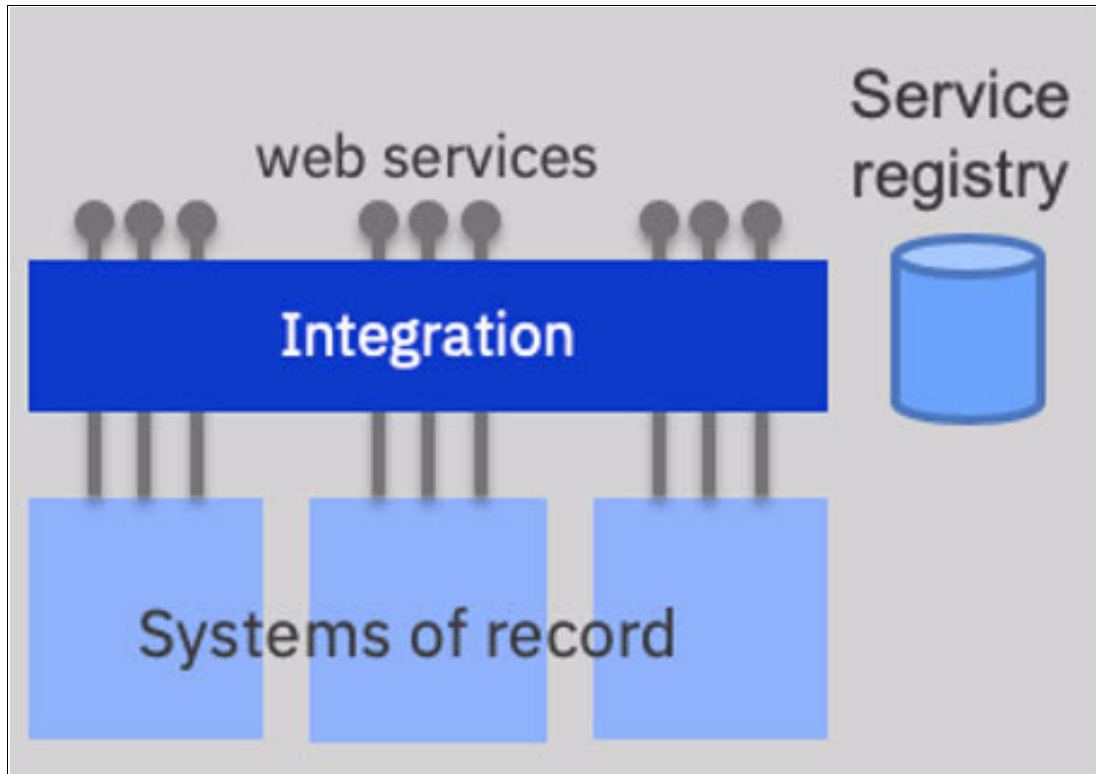


Figure 3-1 Service-oriented architecture

Some attempts were made to enable cross-synchronization and the integration run time to call the service registry at run time. However, because they are two separately designed components, potentially even from different vendors, synchronization was challenging. Worse, some use cases involved a call out to the service registry during the service invocation, which introduced another potential point of failure and increased latency at run time.

External exposure of services

Around this time, we started to see enterprises exploring exposure of interfaces beyond the boundary of the organization. There are much more challenging security implications in this situation, so the idea of a secure gateway in the DMZ was introduced, as shown in Figure 3-2 on page 44.

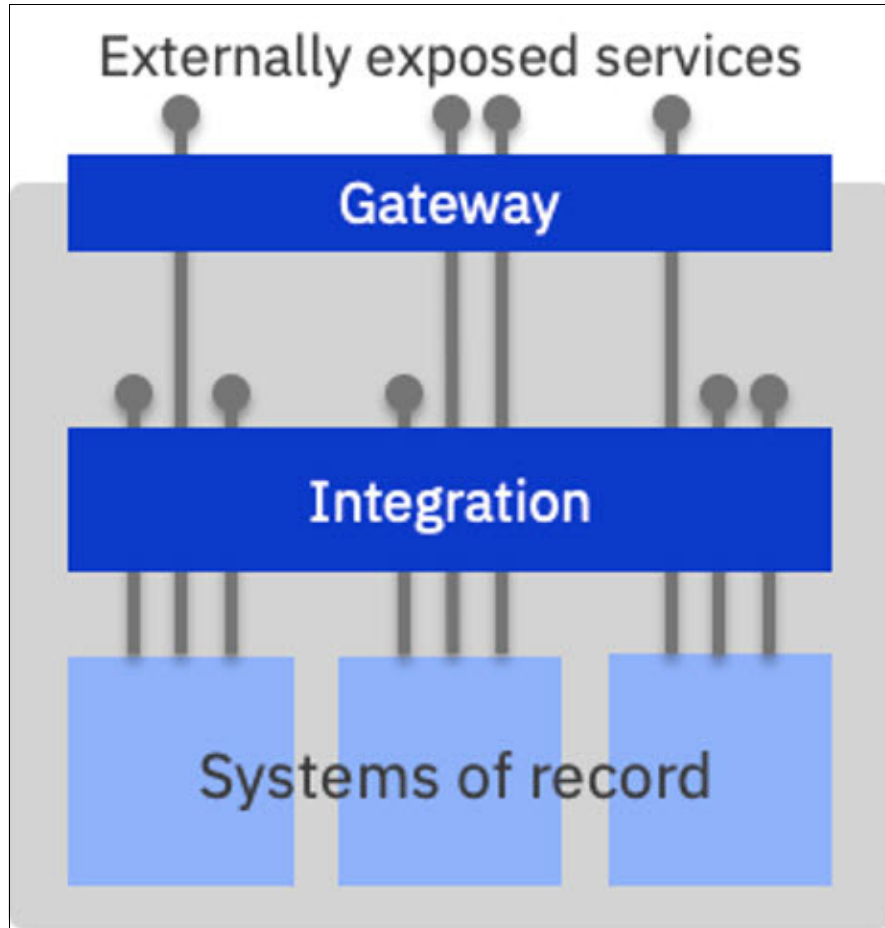


Figure 3-2 Externally exposed services

This gateway had only one job: to securely expose the interfaces on the internet, handling whatever security model was required, and protecting against other forms of attacks, such as denial of service, and even payload-based attacks, such as XML threats. It then passed the request on the enterprise's network to an integration engine that performed the integration to the back-end system's protocols, performed data mapping, and did any composition of invocations that were required to form the response. This situation encouraged a separation of duties between exposure (on the gateway) and integration (in the integration run time) so that each component could focus on what it does best.

From web services to RESTful APIs

As often happens as technologies mature, web services became more complex to implement. Eventually, a new protocol was introduced, *APIs*, which use JSON payloads over HTTP. APIs became popular for external exposure because the consuming application (such as single page web applications and mobile apps) preferred the simpler JavaScript native JSON as a data format to the more complex XML trees. However, there was something else different about these interfaces. They used what was termed a *RESTful interaction pattern* that uses the natively HTTP verbs to mirror the common create, read, update, and delete against data entities. These patterns were easy to consume by the new applications, but often the back-end systems were a long way from this model. To provide the needed RESTful interface, as shown in Figure 3-3 on page 45.

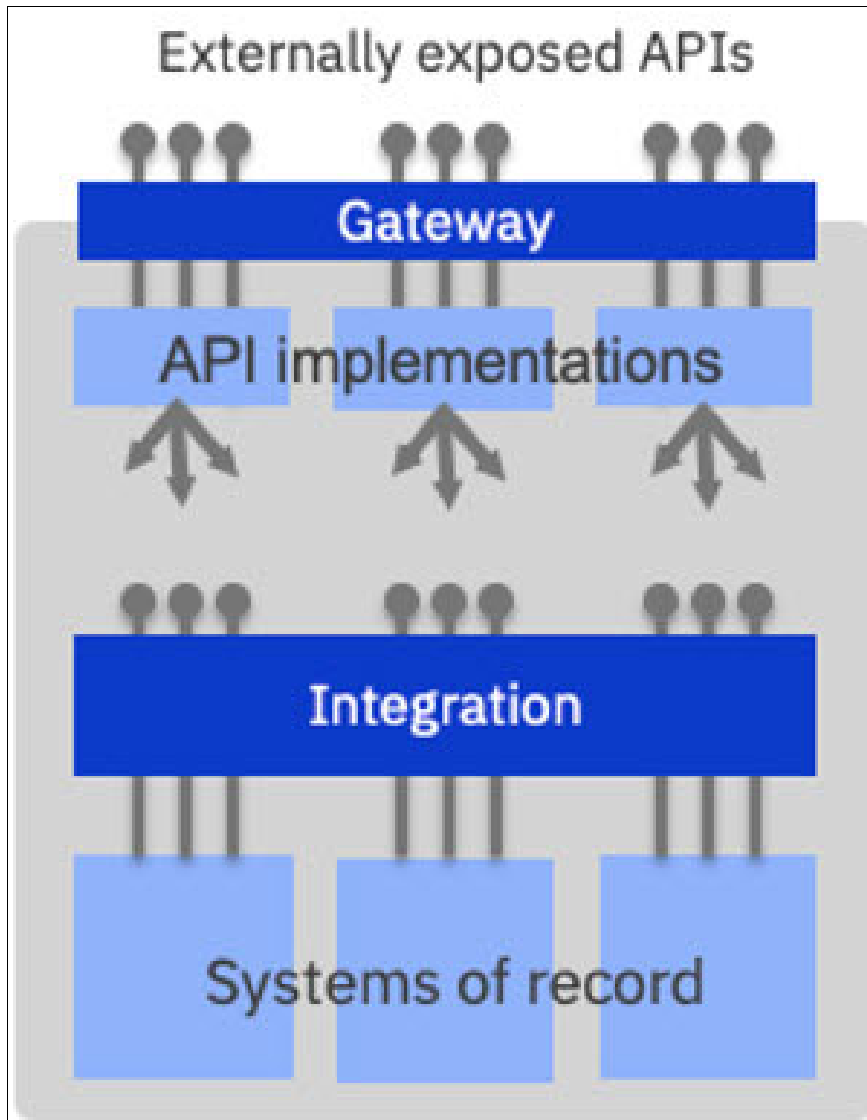


Figure 3-3 RESTful APIs

Soon, we saw the introduction of an engagement tier to create these RESTful APIs by converting and enriching the back-end data to produce this consumer-focused interface. The engagement tier was there, but it was seen as the presentation tier where consumer-facing web applications were built. Now, it has an extra role in creating consumer-facing APIs.

The introduction of API management

It was around this point that the API economy grew. These consumer-focused APIs could be made available to external innovators to rapidly create disruptive applications and break into new markets. However, this situation created challenges:

- ▶ How does an organization keep up with the number of consumers wanting to be onboarded to use the new APIs?
- ▶ How should they monitor and even limit their usage to avoid rogue applications over-using the service.
- ▶ How do they enable new communities to quickly learn how to use the APIs, and enable working in collaboration with those communities to improve the API designs?

In some instances, there was even a need to monetize (charge for use of) the API.

Figure 3-4 on page 46 provides an overview of API management.

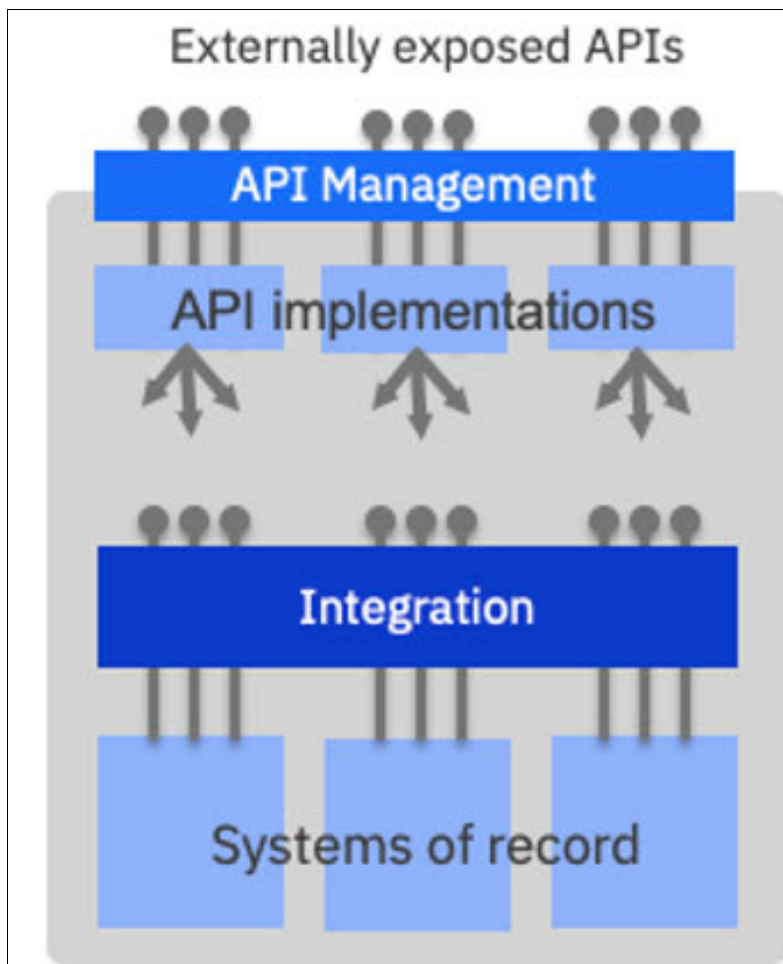


Figure 3-4 API management

To cater for these exposure-based needs of the consumers, rather than return to the challenges of a separate service registry product, the API gateway vendors took on the challenge of incorporating the registry alongside the gateway. The separate service registry became the incorporated API catalog, and API management was born, as shown in Figure 3-5 on page 47.

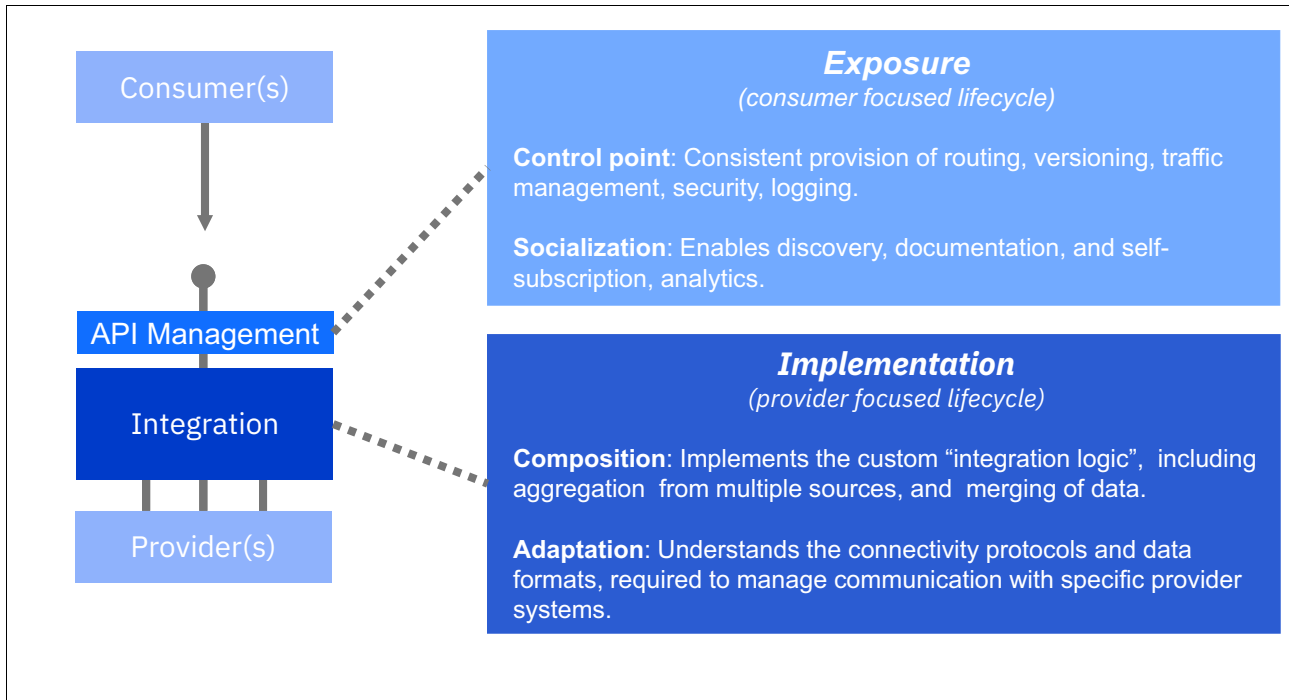


Figure 3-5 Differentiating exposure from implementation in API-led integration

The API management capability enables API providers to create or import an interface specification, provide the necessary configuration for a security model, and document how to use it. The API provider can then decide who should be able to see and subscribe to the API and what service they can expect, for example, in terms of throughput rate. At the designer's request, the API management capability then simultaneously configures the API gateway, makes the documentation available through a portal site, provides facilities for self-subscription to the API, and collects data on the usage.

API management within the enterprise boundary

After the components of API management matured and the benefits were clear, enterprises wanted those same benefits within their organizations. If they could have that level of convenience for consumers of APIs outside the enterprise boundary, surely they wanted that situation for the broader set of APIs that they wanted to expose internally, as shown in Figure 3-6.

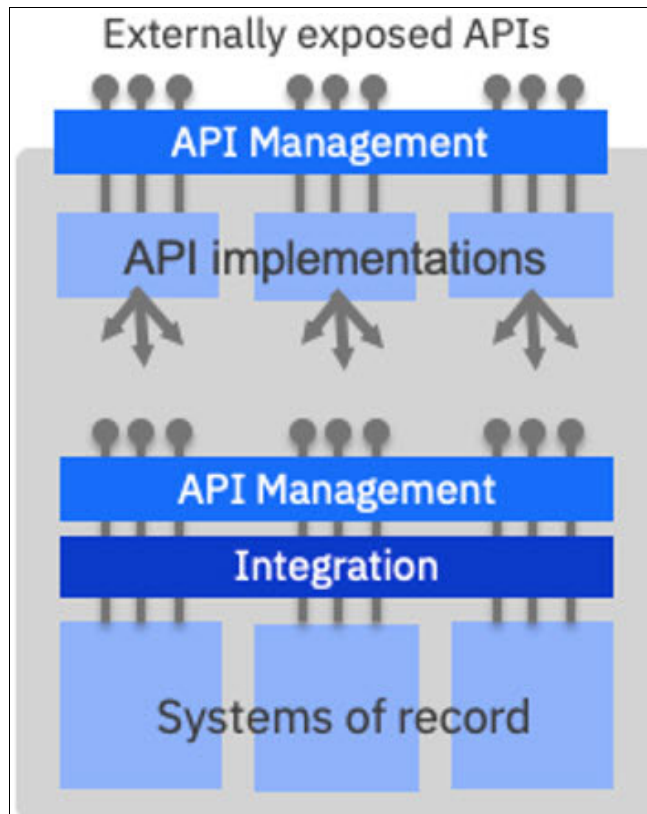


Figure 3-6 API management within the enterprise boundary

As such, the API management capability often has gateways both at the edge of the enterprise and within it. You could see this situation as the maturing of SOA because it provides easily reusable services. Some organizations continue to refer to internally exposed interfaces as services for that reason regardless of whether they use RESTful JSON and HTTP, or SOAP-based web service protocols.

API management: More than a gateway

So far, API management has been represented as little more than a gateway through which invocations pass, but for most implementations it is multiple separate but highly coordinated components. Only the gateway lies on the invocation path of the APIs.

The API gateway is responsible for the following items:

- ▶ Composition: Implements the custom *integration logic*, including aggregation from multiple sources and merging of data.
- ▶ Decoupling and routing: Acting as a proxy to the actual implementation to protect the consumer from change. Requests can be rerouted to new versions of an API implementation or have rules that are based on the incoming message headers or content to decide which version to route to.

- ▶ **Traffic management:** Policies the incoming requests, for example, in terms of throughput. Again, this might be a single throughput rate for all consumers or it might depend on which consumer is calling.
- ▶ **Security:** Protects the API from being used by non-subscribers by using authentication and an authorization mechanism that is based on which roles are allowed access to which APIs. Security can be different for various subscribers, especially for APIs exposed within the organization and those exposed to the general public.
- ▶ **Conversion:** Simple conversions can smooth movement between versions of the API implementations when the data model changes. If the gateway's conversion capabilities are high performance, you might want to perform more complex conversion of data models. You might use the gateway to perform switches between common wire formats, such as between XML and JSON. Anything more complex should be done in a lower level of the architecture.

These items imply that the gateway has significant knowledge of the APIs it is routing, throttling, securing, and converting. Where did that knowledge come from? Who configured the gateway? This is where the core benefit of API management comes in. Gateways have been around for a long time and are rich in functions. As such, they require specialist knowledge to configure them. However, the API exposure use is a simpler subset of that function.

Figure 3-7 gives an overview of how the gateway connects to the API manager and other components.



Figure 3-7 API management: More than just a gateway

In API management, there is a separate component on which you can define and administer the APIs that is called the *API manager*, which is responsible for the following items:

- ▶ **API design:** Enabling API providers to build APIs or create them based on provided information, such as the *Open API Specification* (previously Swagger).
- ▶ **Access management:** Prepare rules about who can discover, subscribe to, and access the API after it is made available.
- ▶ **Policy administration:** Assign policies about throughput, security models, and more about the APIs. You might base these policies on the type of subscription that the consumers choose.

- ▶ **Gateway configuration:** After the APIs and their associated access management and policies are prepared, the API manager can push them to the gateway. The API provider requires only minimal knowledge of the gateway because the API manager uses well-tested patterns to perform the configuration. As such, the consistency is higher, and the likelihood of performing an erroneous deployment that might destabilize other APIs is reduced.
- ▶ **Usage analytics:** During run time, the gateway captures events relating to its usage, which are interpreted and placed in an analytics store. The API provider can retrieve them to evaluate the usage of the APIs for diagnostics, planning, and charging.

But how do developers discover and learn about the APIs in the first place? The consumer-facing view of the API catalog is the *developer portal*, which is responsible for the following items:

- ▶ **API discovery:** When APIs are published to the API gateway, they are also published to the API catalog on the developer portal so that the gateway and portal are always synchronized. Consumers can come to the developer portal to search for APIs that are related to particular data resources and explore their documentation before they commit to using them in their applications. They see only the APIs that are appropriate for their role.
- ▶ **Self-service:** After a consumer has found the API that they want to use, they can use the Developer Portal to directly subscribe to it. They receive any keys and secrets that are necessary to call the API. This process should be as automated as possible so that new users can get onboard with minimal impact on the API provider and still make their invocations as a known consumer so that their usage can be tracked and controlled.
- ▶ **Account usage analytics:** Just as the provider has an interest in the usage statistics, so does the consumer. They want to know that the API is performing well. If they are being charged, they want to see what volumes of invocations they are using.

With the combination of the tightly synchronized API gateway, API manager, and API portal, you can achieve the consistent consumer-centric administration of APIs that is required by autonomous development teams.

Decentralized API ownership on a centralized API management infrastructure

Modern application development practices mandate a more “decentralized” approach to improve productivity and agility by giving teams more autonomy to self-serve. What does that really mean from an API management perspective? Should the decentralized approach be applied down at the infrastructure level, increasing the number of gateways and other components so that each team has their own? Does it mean that application teams should independently use the shared capabilities of API management in an isolated, multi-tenant fashion?

This section explores the deployment of the API management components by using modern application development methods to see what decentralized integration ownership really means in this context.

An overly simplistic view of decentralization might lead us to the conclusion that every team or component might need its own dedicated API management infrastructure. At a minimum, this would mean an API gateway for each API implementation, as shown in Figure 3-8.

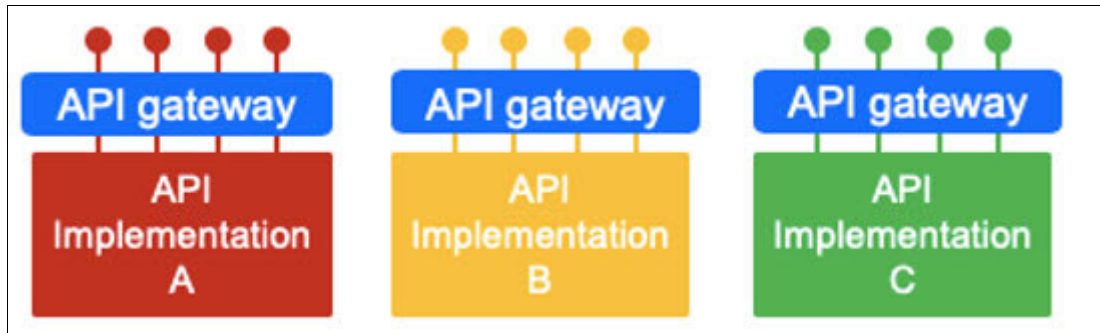


Figure 3-8 Dedicated API management infrastructure

It is easy to see how this might quickly get out of hand. Does each gateway need its own API manager and Developer Portal? It would be painful if each team had to manage their own API management infrastructure alongside their implementation. Furthermore, the mapping between exposed APIs on the gateway and the underlying implementations might not be one to one. An API's coverage might spread across many different implementations. An API might be exposed in many different forms to satisfy different channels or types of consumer. How would you split up the gateways? Figure 3-8 suggests that in the case of API management that you should not be looking at decentralization from an infrastructure perspective.

You might notice that we deliberately used the phrase *decentralized integration ownership*. We are intentionally emphasizing that it is the ownership that must be decentralized among teams and not necessarily among the infrastructure as well. In the case of API management, the thing you are decentralizing is the ownership over the ability to administer APIs (the provider perspective) and the ability to discover, subscribe, and use them (the consumer's perspective). API management deliberately designed to make it easier to decentralize ownership of the APIs so that providers and consumers could administer their own APIs without referring to a central team.

We already described that distributing the infrastructure, for example, with a separate gateway for each implementation, adds unnecessary complexity. As shown in Figure 3-9, could you have one API management infrastructure across all APIs and still provide decentralized ownership?

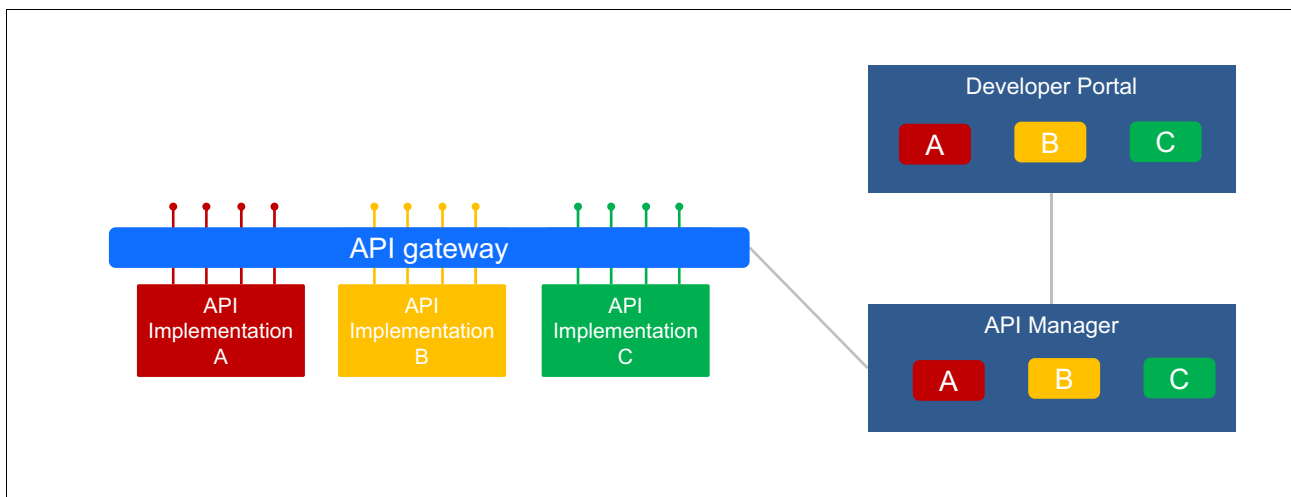


Figure 3-9 Decentralized API ownership on a centralized API management infrastructure

A good API management capability should provide strong multi-tenant capabilities so that it can expose APIs from multiple separate implementations and provide good isolation. For example, managing heavy traffic through one API (and perhaps limiting it in relation to the policy that is defined for its consumers) should have no effect on the performance characteristics of any other APIs also passing through the gateway concurrently.

Equally, there should obviously be zero chance of any leakage of runtime data between APIs. This should be true on the gateway and components. There should be no risk that the people configuring one set of APIs by using the API management user interface can make changes or see the other API configurations that are present. The portal access should separately control sets of portal pages for each independently administered API. The rules as to who can view and subscribe to the APIs also must be separate for each API. So, each API is defined, managed, and administered only by the team that created it, and is made available to only those consumers that they deem appropriate. It is a true multi-tenant architecture.

There is nothing technically stopping us from having a separate gateway for each API implementation, but because you have the multi-tenancy capabilities, there is little benefit in doing so. The configuration, runtime data, and performance characteristics are already suitably isolated and independently administrable. You might be concerned about a high level of availability and not want to run the risk of being affected by downtime that is caused by other APIs. However, to define and expose an API, you are not delivering code, only a configuration, so it should be hard for someone defining an API to cause instability on the API management platform.

Furthermore, you expect the API management platform to provide sufficient redundancy and have fast enough component recovery times that any outages that did occur would be barely noticeable.

Perhaps you should be thinking of API management more as a facility that is available to your applications rather than as a part of the applications themselves. An analogy is a network infrastructure. You typically do not expect each application to have its own network routers, for example, because they are an infrastructural capability that you share from the platform on which your environment sits.

Federated gateways and centralized management

Although a gateway per implementation is clearly too granular, there are some reasons that you might want or need to break up some of the API management infrastructure.

A typical reason to split gateways is because of lines of business (LOBs). Each LOB might want an independent gateway for their part of the organization for many reasons, such as independence and decoupling, or it might be because of practical issues, such the different LOBs having different network domains.

Looking from a much broader architectural perspective, many organizations today have a *hybrid* infrastructure, where they have components that are spread between on-premises, and also have a *multicloud* infrastructure. Their infrastructure is spread across multiple cloud vendors. Each of these platforms might come with its own API management capability, but if you adopt all of them, the technical diversity would be constantly increasing. It would also make for a disjointed consumer experience if APIs were served from radically different portals and API catalogs.

Figure 3-10 on page 53 shows an overview of federated gateways and centralized management.

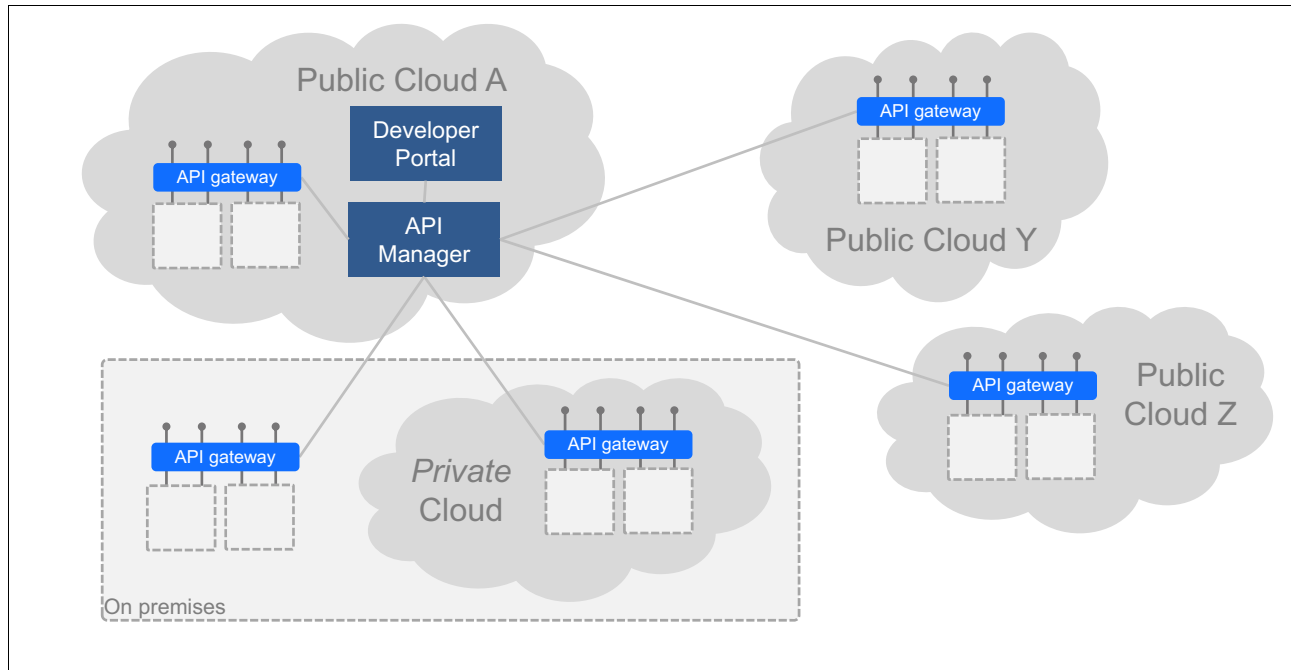


Figure 3-10 Federated API gateways and centralized management

To achieve discoverability and reuse objectives, it is much better to pick one API management capability that can manage multiple gateways (one on each of the platforms) from a central API management facility, which then can publish definitions to a single portal that consumers use to explore and subscribe to all available APIs. It would be only the URLs of the APIs themselves that might reveal that they are on different gateways (and even that could be hidden if required).

Thus, you have the best of all worlds:

- ▶ The simplicity of the API management capability enables *decentralized ownership* so that teams can own the administration of their own APIs.
- ▶ The multi-tenancy of the API management facility can be provided by a *centralized infrastructure* to simplify the overall architecture.
- ▶ The architecture is divided into components such that, for example, it can have multiple federated gateways that are managed centrally.

For more information about this topic, see [Is API management a centralized or decentralized approach?](#)

3.1.2 Cloud-native infrastructure

It might seem like we have forgotten one of the aspects of agile integration: *cloud-native infrastructure*. So far, we have not mentioned containers or cloud-native in relation to API management.

The reality is that from an API consumer or provider perspective, you should be treating API management as a centralized and multi-tenant capability. You should not need to concern ourselves with how it is built internally. It is the API management software vendor who should be ensuring that the technology scales and provides the necessary availability.

Over time, you expect an API management capability to take advantage of and run on a cloud-native infrastructure.

There is one more topic regarding API management and cloud-native to describe: the relationship between API management and a service mesh. However, because it is not an integration modernization topic, and we have not described a service mesh is, we will describe application boundaries and the service mesh in Chapter 4, “Cloud-native concepts and technology” on page 85.

Conclusion on API management in agile integration

API management and the consumer-centric focus are critical to decentralized ownership of APIs. We also described if and when it is appropriate for the infrastructure to be decentralized. Finally, we considered how a cloud-native infrastructure can be applied to the software vendors design of API management components.

3.2 Capability perspective: Application integration

Agile integration, when viewed from an application integration perspective, primarily focuses on the breaking up of the centralized ESB pattern into discrete integrations.

Let’s consider at a high level how this topic relates to the three aspects of agile integration:

- ▶ Decentralized ownership: These smaller, isolated integrations can have different owners. Application teams can choose to own and potentially implement their own integrations.
- ▶ Delivery-led architecture: Integrations can be deployed in more fine-grained groups through automated pipelines to enable Continuous Integration and Continuous Delivery and Deployment (CI/CD).
- ▶ Cloud-native infrastructure: Providing integrations as images to a container orchestration platform enables standardized deployment, administration, elastic scalability, portability, and more.

Figure 3-12 on page 56 shows the summary of the three phases: Traditional, fine-grained, and decentralized.

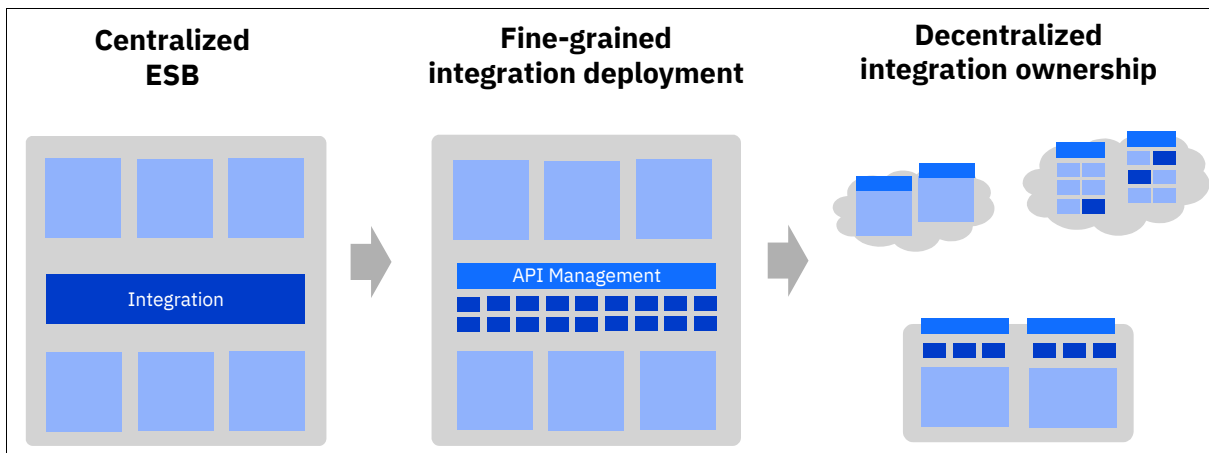


Figure 3-11 Summary of the three phases: traditional, fine-grained, and decentralized

You must do groundwork such as breaking up the ESB, improving your automation of processes, and moving to container infrastructure before you can change the ownership

model for your existing integrations. For some new integrations, you might have some options.

3.2.1 Moving to a cloud-native approach

Most of the changes that are required when moving towards more fine-grained and decentralized integration align with cloud-native principles.

The term *cloud-native* refers to a different approach to designing, building, deploying, and administering components. If done well, it provides *implementation agility, infrastructure optimization, discrete resilience, and platform portability*.

On a practical level, it encompasses the following items:

- ▶ Modular components
- ▶ Stateless components
- ▶ Image-based deployment
- ▶ Elastic, agnostic infrastructure
- ▶ Lightweight run times
- ▶ Log-based monitoring
- ▶ API-based intra-app communication
- ▶ Event-driven architecture
- ▶ CI/CD
- ▶ DevOps
- ▶ Agile methods

We touch on all of these items as we explore modernization of application integration. If you are only vaguely familiar with cloud-native or any of these terms, see Chapter 4, “Cloud-native concepts and technology” on page 85, which explains them in more detail.

We begin a description about breaking up the ESB into more, fine-grained *modular components*.

3.2.2 Fine-grained deployment: Breaking up the ESB

Because of the potential benefits of breaking up applications into microservices, you should consider breaking up the ESB into many separate modules so that each is on its own separate run time with just a small collection of integrations inside.

Figure 3-12 shows a fine-grained deployment.

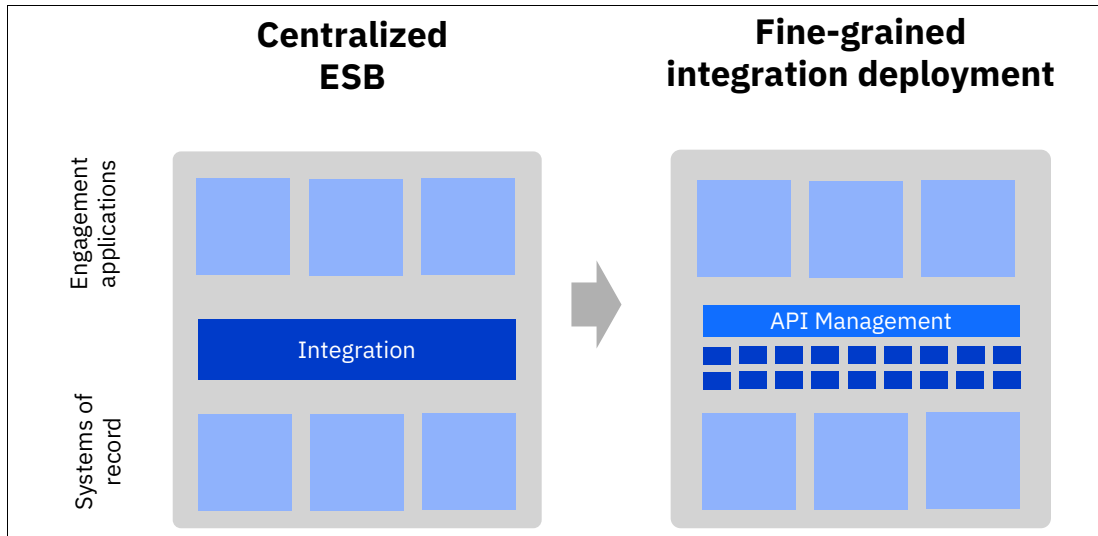


Figure 3-12 Fine-grained deployment: Breaking up the ESB

Figure 3-12 also shows the concept of API management because it is common that when you break up the ESB that you need more sophisticated exposure of APIs, and the two activities are complementary. For more information, see 3.1, “Capability perspective: API management” on page 42.

Let’s consider what benefits you can gain from moving to this more fine-grained model:

- ▶ Independent deployment: Discrete integrations can be deployed with certainty that they will not introduce instability to other integrations. For example, if a new integration included some custom Java libraries that introduce a memory leak or another issue that caused its integration run time to fail, it will not bring down any other integrations when it fails.
- ▶ Runtime version independence: Integrations can be paired with whichever version of the run time that they require, perhaps by using the latest Fix Pack for specific features. There is no need to wait for the next planned runtime upgrade based on the schedules, and you do not need to worry about regression testing of all the other integrations.
- ▶ Minimal dependencies: You need install only the dependencies that are required for the integrations. For example, if a local IBM MQ server is not required with the integration server, it does not need to be present. This situation reduces installation time, start-up time, memory footprint, and overall complexity.

3.2.3 Grouping integrations

Although you might potentially separate each integration into a separate run time, it is unlikely that such an approach would make sense. It is common to hear that enterprises have hundreds of integrations running in their centralized ESB. Moving to hundreds of independent servers would be unnecessarily fine-grained.

The real goal is to ensure that unrelated integrations are not housed together, that is, a middle ground with group-related integrations together (Figure 3-13) can be sufficient to gain many of the benefits.

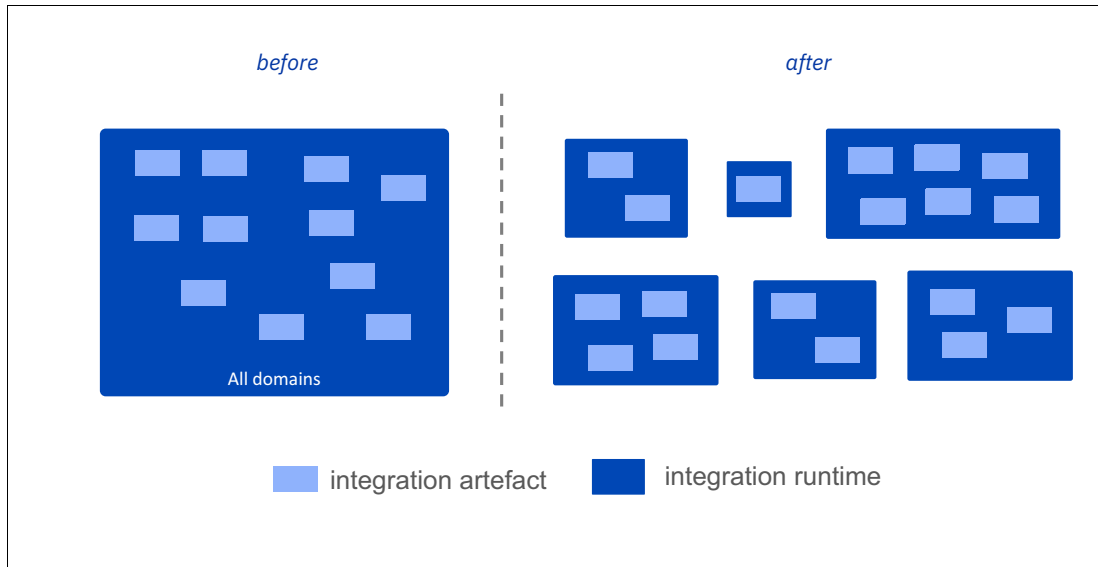


Figure 3-13 Grouping integrations

You target the integrations that need the most independence and break them out on their own. Keep together flows that, for example, share a common data model for cross-compatibility. In a situation where changes to one integration must result in changes to all related integrations, the benefits of separation might not be so relevant unless there are other overriding reasons, such as differing scaling requirements.

The grouping of integrations is covered in greater depth with specific reference to the decision points in 5.3, “API Lifecycle: IBM API Connect” on page 147.

So far, we are talking about the act of separating integrations into more fine-grained groups, which does not necessarily imply a change of infrastructure from, for example, the current virtual machines (VMs) to a container platform. For example, if we were describing IBM App Connect, we could be splitting integrations across integration servers (previously known as *execution groups*), so no containers are required. If you are not ready to make the move to container technology, something like this action might be a valuable step in the correct direction, with its own inherent benefits, and it makes any later move to containers easier and more natural.

Deciding at what point to make the move to containers is important, and should be based on the required benefits, not just on the desire to use modern technology. In 7.1, “IBM App Connect adoption paths” on page 434, we describe this topic in detail in the context of IBM App Connect. However, this chapter is product-neutral, so we look at containers from a purely architectural perspective for now.

As you progress through this section, you take more steps toward a cloud-native approach for application integration. Many of these steps are more beneficial on a container-based platform.

Breaking down the ESB into smaller components so that they are more independent and running them in lightweight containers is part of being cloud-native, but there is more to it than that.

3.2.4 Stateless components

Statelessness of the integration components is critical if a container orchestration capability such as Kubernetes (K8s) administers them. To be stateless, integrations must not hold or persist any state as a result of running an integration that is visible only to them, which is necessary for future processing of invocations. To do otherwise inhibits K8s from scaling the integration (especially scaling down). Similarly, it would make recovering from failure in an integration replica more difficult because any replacement would have to be able to see the same persisted data.

Integrations do not use one of the most common forms of statefulness (stateful sessions) because they rarely have any notion of user state. However, they do have other forms of state that should be eliminated.

One of the most common states is local message queues. If a component has a local message queue that persists messages specifically for that replica, then if the component is stopped, these messages become orphaned or lost. Fortunately, there is a relatively straightforward solution because most modern queuing systems can now switch to remote queues that have a lifecycle that is independent from the integration and potentially can be moved to a more shared pattern so that queues are not dedicated to a particular instance of an integration. These issues are subtle, as is anything involving persistent state and containers, and you find plenty more resources relating to this topic in later sections.

Not all integrations can be stateless, and there are cases where you must hold a state by using K8s concepts such as Persistent Volume Claims, which enable reattaching and even sharing a persistent state, and StatefulSets, which enable us to manage replicas that need more control over what happens when they scale or are reinstated.

Many integrations are inherently stateless, and more can be made stateless with relatively minor refactoring. For those integrations that must remain stateful, they can still be containerized, and they are better off, but you might have to compromise on some of the cloud-native benefits.

3.2.5 Image-based deployment

Image-based deployment is a mechanism for doing file-based packaging of all local dependencies into an immutable image. The image contains the integration artifacts, the integration product run time, and enough of the operating system for it to be run independently in a container environment.

Figure 3-14 shows image-based integration deployment.

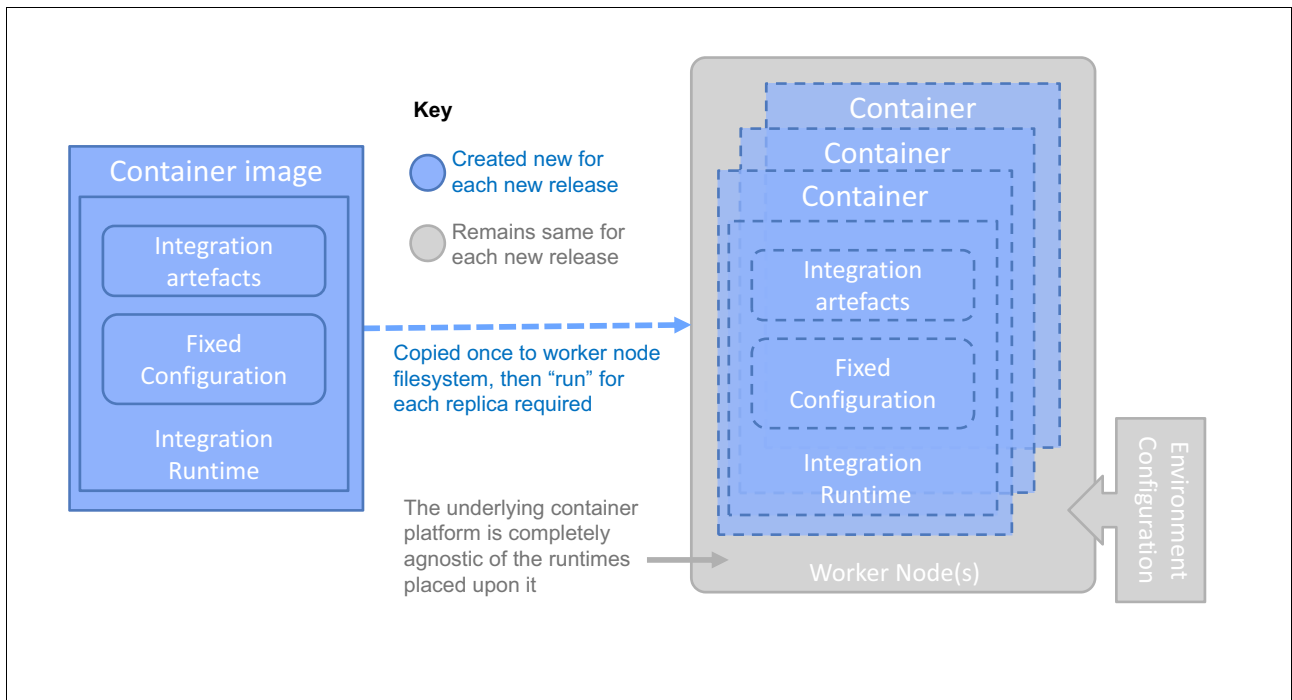


Figure 3-14 Image-based integration deployment

Packaging top to bottom dependencies makes for more consistent and rapid deployment. It also makes it possible for container environments to deploy and administer integrations without having to know anything about the integration product. This consistency improves testing confidence, enables simpler re-creation of environments for, for example, functional and performance testing, and simplifies problem diagnosis.

Let's briefly compare image-based deployment to traditional deployment. In traditional deployment, you create an integration server with all the dependencies, fix packs, and configuration that is required for all the integrations that will be deployed to it. Many of these steps are done by running proprietary installation and configuration commands of the operating system and integration server. Then, you deploy your integration again by running a proprietary command on a running live integration server along with any additional configuration. Few of these actions can be performed by a neutral container orchestration capability.

With image-based deployment, integration servers are not administered when they are live. The installation of the product and its fix packs and the addition of application artifacts are done by laying files down on the file system and capturing it as a "container image". The image will have further metadata within it, such as what process within the image to run on start and which ports to open. From this point, the image is a black box. You do not have to know anything about what kind of run time or integrations are inside it to start it.

3.2.6 Elastic, agnostic infrastructure and container orchestration platforms

What we described in terms of image-based deployment is how Docker containers work. It is their *black box* nature that allows a neutral container orchestration system like K8s to deploy and administer them consistently without any product-specific knowledge.

Although this is an architectural chapter, a concrete example is essential here. IBM Integration Bus (the predecessor to IBM App Connect) required an *integration node* (or *broker* in older versions) to look after the set of *integration servers* (or *execution groups* as they were previously known) that performed the integrations. Most traditionally deployed products had something similar.

Creating an integration topology entailed explicitly installing both the integration node and several integration servers and configuring them to know about one another. Then, you had to configure the integration node to perform load-balancing across the integration servers or independently set up a pair of HTTP load balancers to spread the incoming calls across them. You had to know a lot about the topology of IBM Integration Bus and the installation configuration to set up and maintain a production worthy high availability (HA) topology.

In this new cloud-native approach with the latest version of IBM App Connect, there is no need for an integration node when running in a containerized deployment. You can still create a traditional topology on VMs with an integration node, as described in 7.1, "IBM App Connect adoption paths" on page 434, but it is not needed in a container platform.

By using K8s, you can *deploy* container images regardless of what they contain. K8s can scale, load balance, provide fault tolerance, and perform updates. You provide a policy about its non-functional characteristics, for example, a minimum number of replicas for initial HA and performance characteristics, when to create more replicas, and up to what maximum. If a container fails, K8s reinstates it. If the existing containers reach a CPU threshold, more replicas are created. All the necessary load-balancing and health-checking is done automatically for you.

Figure 3-15 on page 61 shows traditional versus cloud-native integration models.

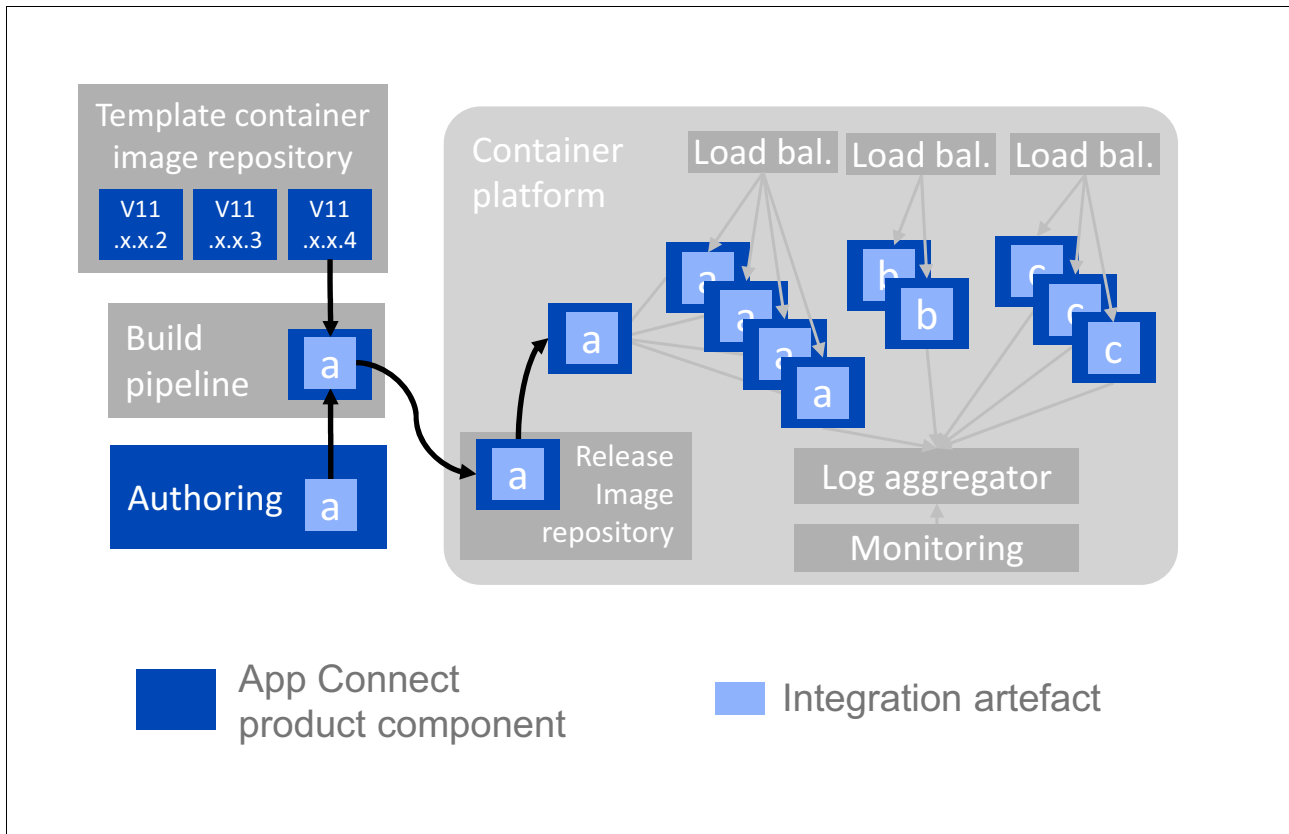


Figure 3-15 Traditional versus cloud-native integration models

This is a radically different way of building and running an integration topology. The benefit is that it enables a generic platform such as K8s to instantly create a topology based on any image that is provided to it, and autonomously provides HA and scaling. Better still, if you must re-create *exactly* that setup in another environment, K8s has everything that it needs to do so. There is no risk of configuration divergence. All environments are started from the same set of images, configuration, and policy.

Regarding changes, we improved the application component design. For example, assume that there is an important new fix pack out with a security patch for the run time. You cannot make changes on a running server because your live container will be out of sync with the image repository and you can no longer easily re-create the environment.

Instead, think back to our description of image-based deployment. All changes should be done by creating a container image, placing it in the image repository, and then instructing K8s to refresh it in whichever environment it must be applied. K8s starts a new set of replicas based on the new image, then shuts down the current containers until the new configuration is in place. K8s did not need to know anything proprietary about how to deploy applications to the run time in the container; it just creates a new container from the new image just like it did originally.

Breaking up an ESB into more modular fine-grained integration components might seem like it adds more complexity to your infrastructure. You can now see that much of the creation and management of these discrete topologies for each integration is done for us by K8s, so it no longer seems so daunting. Furthermore, adopting container orchestration reduces the deployment and administration skills operational teams need because they largely are consistent across any solution that sits on a container-based platform. This *operational consistency* becomes all more important in the decentralized ownership of integration.

3.2.7 Lightweight run times: How the modern integration run time has changed

K8s can perform deployments, create resilient topologies, self-heal, and scale elastically, but your runtimes must be lightweight, that is, they ideally must be a single process that pulls everything it needs from the file system on start and requires no further online configuration.

For this reason, integration run times have evolved in recent years. Let's have look at some of the key improvements that enable the characteristics we discussed:

- ▶ **Rapid start-up:** The run times are a single process that runs in containers such as Docker. They are lightweight enough that they can be started and stopped in seconds, and can be easily administered by orchestration frameworks such as K8s.
- ▶ **Dependency free:** The run times no longer have hard dependencies on databases or message queues that must be installed alongside them, although they are still adept at connecting to them remotely.
- ▶ **File system installation, configuration, and deployment:** The run times can be installed and configured by laying their binary and configuration files on a file system and starting them. Furthermore, there is no longer a need to deploy integrations to a live running integration server. The integrations artifacts (maps, flows, configuration, and others) can be laid down on the file system. This situation is ideal for the layered file systems of Docker images and the automated pipelines that are essential for CI/CD.

For more information, see 5.5, "Application integration: IBM App Connect" on page 155.

Now, we continue with a few more cloud-native characteristics and see how they are handled by modern integration run times.

3.2.8 Log-based monitoring

You should not be connecting directly to individual containers, so monitoring cannot be done by direct connection. Instead, the integration run times are enabled to report their status by using standardized logging, which can then be aggregated by a monitoring stack on the container.

3.2.9 API intra-application communication

Cloud-native components should communicate by using runtime-neutral protocols, the most common of which is RESTful APIs. Exposing and calling APIs is part of an integration's basic function.

3.2.10 Event-driven architecture

In some cases, it is necessary to reduce runtime coupling between cloud-native components by using asynchronous protocols, patterns such as event sourcing, and Command Query Responsibility Segregation (CQRS). Even-based brokers such as those based on Apache Kafka are popular in this space. Traditional messaging still has a place as a cross-platform communication style that provides transitionally assured delivery. Connectivity over these protocols and transports is basic function for a modern integration engine.

3.2.11 Agile methods

There is little point in changing your entire architectural approach and moving to a container-based infrastructure if you then persist with an outdated waterfall approach to implementation.

Note: A *waterfall* methodology is where an attempt is made to gather all the requirements up front, and then the implementation team works in isolation until they deliver the final product for acceptance.

Agile is a broad term that encompasses lessons that were learned over many decades of software development around how to work more collaboratively with the business, working in short sprints with the aim of delivering some value early, and then reaffirming the direction of travel.

Integration is no different in this respect, and it benefits from the fact that most integrations are naturally discrete pieces of work. Lessons that were learned from the SOA era confirm that spending too long over-analyzing the perfect set of reusable services is time that is lost. Picking a handful of likely candidates, building them out for immediate value, and then iterating to improve them is a much more likely route to success.

There are good reasons that it was more difficult to work in an agile way in the past. The concepts were there and progressively honed over the years in various forms, but they also needed the technology to catch up. Now, you live in an era where individual decoupled components can be rapidly and built, tested, and deployed without the impact of long up-front infrastructure projects to environments in place. Now, agile is much more feasible because the technology has caught up and the methods are there, but there is still one more thing that must be sorted before you can move at the speed of the business: processes and people.

3.2.12 Continuous Integration and Continuous Delivery and Deployment

Development process automation is a foundation of agile delivery. Agility is defined by how fast you can go through the cycle of understanding a requirement, implementing that requirement, and then getting feedback on that requirement before going back into the loop. You cannot afford for the implementation part of that loop to be impeded by a slow and complex build and deploy processes.

You must automate and streamline the pipeline process for building, in our case, the container images from our latest version of an integration, which must include automated quality assurance and testing. This process is known as *Continuous Integration* (CI).

You also must automate the process of delivering and potentially even deploying that new image into the relevant environments. This process is known as *Continuous Delivery and Deployment* (CD).

The good news is that the steps you have taken to make the run time more suited to containers have also made both these tasks simpler. If you recall, installing and configuring the integration run time is now a matter of file copy commands, as is the addition of the integration artifacts. These artifacts can be programmed into easily understood build pipeline scripts. These scripts can be triggered based on events from the source control, resulting in a container image that is ready for deployment. Deployment is now something that based on additional policy information, which K8s already has.

We are over-simplifying, but the point is that the complexities of creating CI/CD pipelines are reduced after we take on the elements of the cloud-native approach.

An example of CI/CD pipelines for App Connect is shown in 7.5, “Continuous Integration and Continuous Delivery Pipeline using IBM App Connect V11 architecture” on page 465.

3.2.13 DevOps

Last but not least, we come to people, or more specifically, how people collaborate across teams and which teams they are in. To keep that close link between what we implement and what the business sees and uses in production, we must narrow the gap between development and operations.

You must get away from the “throw it over the wall” mentality that can occur between heavily separated development and operations teams and give both sides an investment in each other’s success. *This situation is DevOps.*

The same potential separations exist in the integration space as they do in application development, where DevOps was first introduced.

On a technical level, we can say that through image-based deployment and the ability to repeatedly deploy discrete fine-grained integration components, cloud-native techniques ensure a much greater consistency between what was tested during development and what is live in production. But, DevOps is not just about technology.

People are the key to success in DevOps: How they are organized, incentivized, motivated, and more. It is beyond the scope of this book to delve into how to achieve DevOps successfully, but we should mention one particular role change that is prevalent.

Operations teams, which used to have their hands full focusing on performing manual procedures to create and configure environments and reactively chasing down problems based on alerts, now have an opportunity for a more interesting role. As platforms such as K8s take away much of the drudgery of building and maintaining environments, the operations teams are turning their hands to automation. They automate anything that they see often enough to understand a pattern. They are no longer operations staff: They are Software Reliability Engineers.

3.2.14 Creating integrations is becoming easier

If you have followed all the elements of a cloud-native approach, then you have discarded the proprietary mechanisms for HA, scaling, deployment, monitoring, and other system administration tasks. The teams can use generically available skills for container orchestration for building, deploying, and administering their implementations.

This standardization extends beyond container orchestration into ubiquitous source code repositories such as GitHub and build tools such as Jenkins.

Ideally, the only new skills you need to pick up to use another run time is how to build its artifacts, whether that is writing code for a language run time or building mediation flows for an integration engine. After you finish building your artifacts, everything else is done in a similar way across all runtime types.

Furthermore, the increasing enhancements to tools for integration mean that many interfaces can be built by configuration alone. Combine this practice with the addition of templates for common integration patterns and integration best practices that are burned into the tools that further simplify the task of building integrations.

As such, not all integration implementations require deep integration specialists. Having them certainly reduces the recruiting challenge for the integration team, but it also opens another possibility: Does the central integration team still need to build and administer all of the integrations, or might some potentially be taken on by the application teams themselves?

3.2.15 Decentralizing integration ownership

If you have successfully broken up the integrations into separate decoupled pieces that can be administered at run time by using standard container platform skills, you have an opportunity to distribute those integrations differently from an ownership and administration point of view as well.

As shown in Figure 3-16, decentralization means allowing the administration and potentially even the creation and maintenance of integration artifacts to be owned directly by application teams rather than by a centralized team.

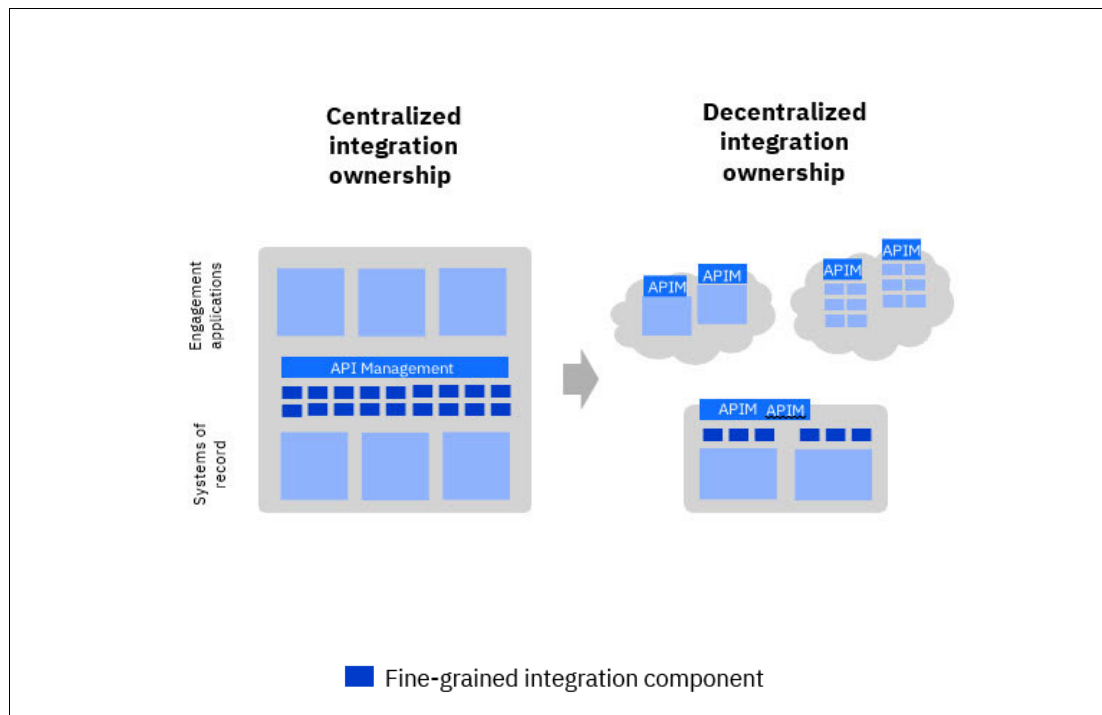


Figure 3-16 Decentralizing integration ownership

It is important to recognize that decentralization is a significant change for most organizations. For some, it might be too different to take on board and they might have valid reasons to remain centrally organized in relation to integration. For large organizations, it is unlikely it will happen consistently across all domains. It is much more likely that only specific pockets of the organization will move to this approach, where it suits them culturally and helps them meet their business objectives, and many of the integrations will remain centralized.

So, we will likely in most organizations see a mixture of both halves of Figure 3-16 on page 65. However, some application teams always wanted to own their integrations, so the drive for some decentralization is present.

There are many potential advantages to this decentralized integration approach:

- ▶ **Application domain knowledge:** A common challenge for separate SOA teams was that they did not understand the applications they were offering through services. The application teams know the data structures of their own applications better than anyone.
- ▶ **Agility:** Fewer teams will be involved in the end-to-end implementation of a solution, reducing the cross-team chatter, project delivery timeframe, and inevitable waterfall development that typically occurs in these cases.

Let’s reinforce that point we made in the introduction of this section. Although decentralization of integration offers potential unique benefits, especially in terms of overall agility, it is a departure from the way many organizations are structured today. The pros and cons must be weighted carefully, and it might be that a blended approach where only some parts of the organization take on this approach is more achievable.

3.2.16 Using integration run times in a microservices application

In 3.2.15, “Decentralizing integration ownership” on page 65, we spoke about application teams creating and administering integrations, but in Figure 3-16 on page 65, we implied we that application teams look after the back-end systems of record. What if the team taking on the integrations is a microservices application team building a system of engagement application?

Figure 3-17 shows implementing microservices with integration.

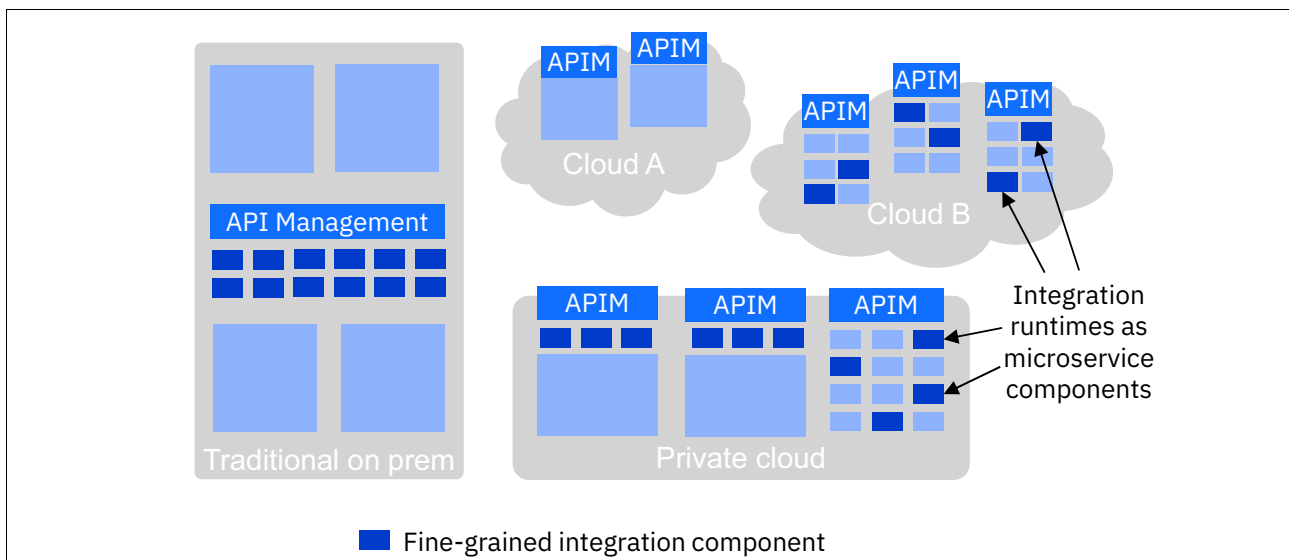


Figure 3-17 Implementing microservices with integration

One of the key benefits of the microservices architecture is that you can use multiple different run times that are suited for the different jobs. For example, one run time might be focus on the user interface and is based on Node.js and many UI libraries. Another run time might be more focused on a particular need of the solution, such as a rules engine or machine learning. All applications must get data in and out, so you expect an integration run time too, which is a specialized run time.

It is common to find microservices components in an application whose responsibilities are primarily focused on items like integration. For example, what if a microservices component implemented an API that performed a few invocations to other systems, collated and merged the results, and responded to the caller? That process is something an integration tool would be good at. A simple graphical flow that shows which systems you are calling and might let you easily find where the data items are merged, and provides a visual representation of the mapping, is easier to maintain than hundreds of lines of code.

Let's look at another example. There is a resurgence of interest in messaging in the microservices world because of the popularity of patterns, such as event-sourced applications and using eventual-consistency techniques. So, you probably find many microservices components that do little more than take messages from a queue or topic, do a little conversion and perhaps enrichment, and then push the result into a data store. However, they might require many lines of code to accomplish this task. An integration run time can perform the task with easily configurable connectors and graphical data mapping so that you do not have to understand the specifics of the messaging and data store interfaces.

The integration run time is a lightweight component that can be run in a cloud-native style. Therefore, it can easily be included *within* microservices applications rather than just being used to integrate among them.

When using this approach, an inevitable question is “Am I introducing an ESB into a microservices application?” It is an understandable concern, but as you may recall from the earlier definitions, an integration run time is *not* an ESB. It is just one of the architectural patterns of which the integration run time can be a part.

ESB is the heavily centralized, enterprise-scope architectural pattern that is described in 2.2, “The journey so far: SOA, ESBs, and APIs” on page 9. Using a modern lightweight integration run time to implement integration-related aspects of an application, deploying each integration independently in a separate component is different from the centralized ESB pattern. So the answer is *no*: By using a lightweight integration run time to containerize discrete integrations, you are *not* re-creating the centralized ESB pattern within your microservices application. You are using the most productive tool for the job in hand.

In the past, it was difficult for application developers to use integration because the integration tools were not part of the application developer's toolbox. Deep skills were required in the integration product and in the associated integration patterns. Now, integrations are easier to create and maintain.

Now that applications are composed of many fine-grained components that can be based on a polyglot of different run times, you may use the correct run time for each task. Where integration-like requirements are present, you can choose to use an integration run time.

3.3 Capability perspective: Messaging and event streams

In this section, we describe the messaging and event streams from a capability perspective.

3.3.1 A brief history of asynchronous communication

Asynchronous communication has been present in IT since the beginning. Synchronous communication was nearly impossible among disparate platforms. Files and messaging infrastructures were the only way to move data among them.

In recent years, retrieving information by using stateless request and response over HTTP became ubiquitous by using web services (SOAP/HTTP) and more recently RESTful APIs by using JSON and HTTP. Although these services enable many data communication scenarios, there are still a significant proportion that requires a richer and more asynchronous approach.

The renewed importance of messaging

All enterprises need robust, secure, and reliable ways to move data asynchronously.

The challenge is more relevant and complex. Instead of moving data from one application and operating system to another one, you now have the additional challenge of moving among geographically separate cloud locations that are run by different vendors. Furthermore, because of the increasing number of connected personal devices, sensors, and increasingly fine-grained components, the number of consumers of these asynchronous messages and events is increasing.

Introducing event streams

The term *events* hints at an important progression in this space. You must make a clearer distinction between two different use cases, which for our purposes here we call messaging and events:

- ▶ Messaging was originally introduced to enable reliable, once only, decoupled delivery across disparate platforms. It continues that mission-critical purpose, but now also takes the same role in reliable communication across cloud boundaries.
- ▶ Events focus on the publish/subscribe pattern. This pattern is possible with messaging, but modern use cases come with radically different requirements than those of the past and require a different implementation. You now must handle dramatic increases in both the number of events occurring and subscribers listening for those events. Furthermore, the stateless nature of many consumers of events introduces the need to retain an event history.

We describe in detail the technical and architectural differences between these two complementary types of asynchronous communication later in this section, but first you must consider how the requirements for messaging and event streams are changing from an organizational ownership point of view.

Application-owned messaging and events

We described how many enterprises are experiencing a move towards more decentralized ownership of IT so that they can prototype and iterate on solutions more rapidly. It is no longer acceptable to have to wait on a highly specialized team to provision and administer message queues and event stream topics. To innovate more rapidly, these decentralized application development teams must be able to self-provision and administer the capabilities that they need on demand.

To enable teams to self-provision without deep knowledge of the technologies, we must introduce simplified mechanisms that are based on using templates and patterns for the provisioning tasks, which ensures consistency of implementation and a governable and maintainable landscape.

Cloud-native infrastructure for messaging and events

A key opportunity to simplify the deployment of messaging and event topologies is the introduction of container platforms, which you can use package the software and its configuration so that it can be deployed in a standardized way for a container orchestration platform such as K8s. As K8s skills become more ubiquitous, this approach will simplify the way that messaging and events technologies are deployed and maintained.

Because messaging and events implicitly involve the storage of state, their deployment in containers is more subtle than that of stateless components. Therefore, we must use more complex deployment types such as the StatefulSet, and elastic scalability becomes a slightly more challenging proposition. There are still many benefits to be gained by exploring containers in this context, for example, in terms of operational consistency, portability, and more.

3.3.2 Introducing messaging and event streams concepts

At a high level, messaging and event streaming technology appear to have overlapping capabilities because they both can be used for the same core asynchronous interaction patterns. However, on a deeper review of the capabilities of each technology, it becomes clear that they achieve these patterns in different ways to serve different purposes, and it is critical to select the correct technology for the job.

Primary asynchronous interaction patterns

Let's clearly define these high-level asynchronous interaction patterns before we delve into each of the two technologies and compare their usage:

- ▶ **Fire and forget:** A requesting application sends a message or event for processing to another application. The requesting application wants to ensure that the message or event that was sent might in the future submit another request to determine the status of the original request, but this is a separate interaction that might not be done by using messaging or events.

By using this pattern, requesting applications can submit messages and events to be stored by the message and event server and then continue with other processing. The target application processes the message or event when it is ready to do so.

Figure 3-18 shows the fire and forget pattern.

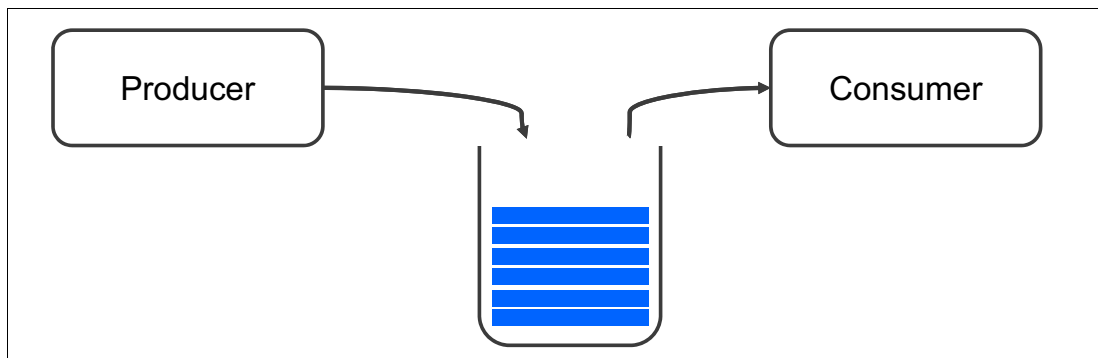


Figure 3-18 Fire and forget pattern

- ▶ Decoupled request and reply: Similar to a synchronous call over a protocol such as HTTP, a requesting application sends a request message or event to a target application and requires a response. However, in contrast to synchronous HTTP calls, the requesting application can choose to continue processing and be called with a response when one is available. Messaging facilitates this task because the location and availability of the target application can be decoupled from the requesting application.

Figure 3-19 shows a decoupled request and response.

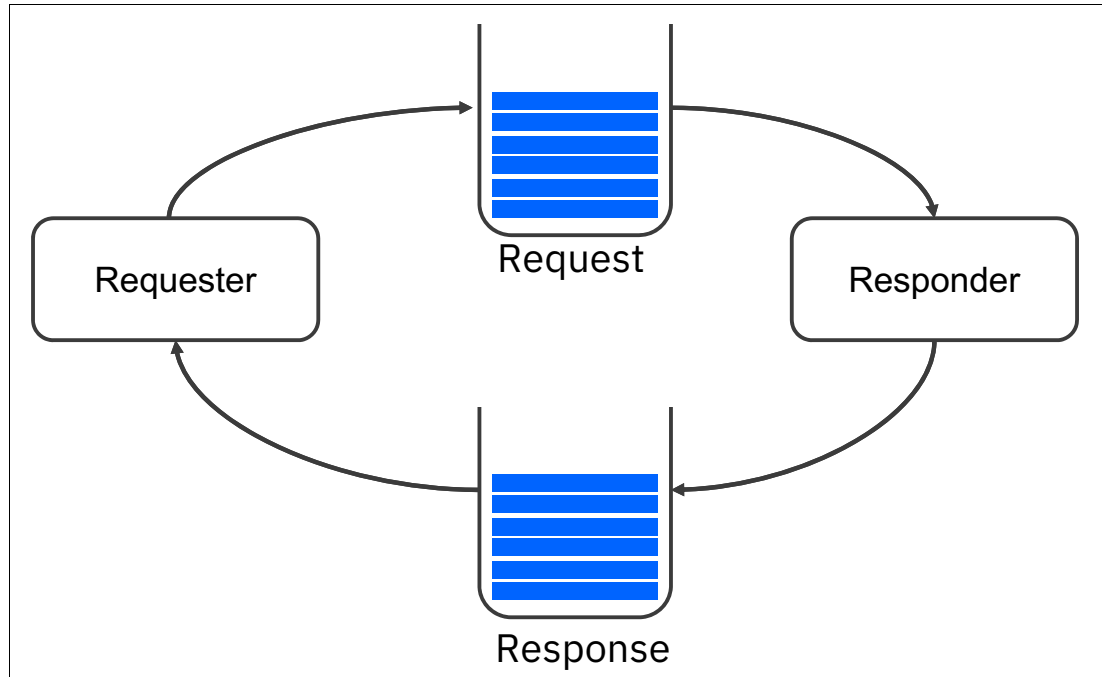


Figure 3-19 Decoupled request and response

- ▶ Publish/subscribe: The previous two interaction styles have a one-to-one relationship between the requesting and target applications. In other scenarios, it is often desirable to send messages and events to multiple target applications. For example, a message or event that is published with an airlines flight change might be of interest to the passenger mobile application, the itinerary application, and many others. The publish/subscribe interaction pattern provides this capability. Applications that publish define a topic where they put their messages, and that same topic is used by applications that are interested in consuming those messages. The list of subscribers is dynamic. It can change over time, as you might suspect when other applications would benefit from that same information.

Figure 3-20 on page 71 shows the publish/subscribe pattern.

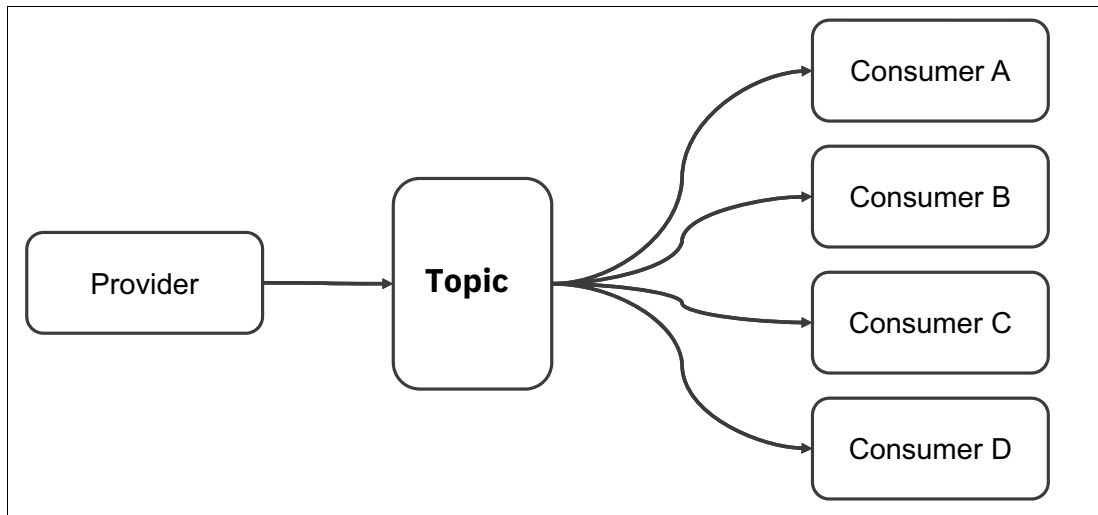


Figure 3-20 Publish/subscribe pattern

3.3.3 Differentiating capabilities

Let's now consider the core differentiating capabilities of each technology to highlight why these technologies are different.

Messaging providers differentiating capabilities

The following list summarizes messaging providers and their differentiating capabilities:

- ▶ **Transient data:** Data is stored only until a consumer processes the message or it expires. The data does not need to be persisted longer than required, and it is beneficial from a system resource point of view that it does not.
- ▶ **Request/reply:** Although messaging operations are often fire and forget, they can be request and reply too. Messaging technology should support this pattern of interaction. Request and reply are something that event brokers are not suited to because it is hard to send a response event back to only a specific subscriber.
- ▶ **Targeted reliable delivery:** Messaging instances can be connected together to a messaging network. Clients send a targeted message by using their local messaging instance, and the message is transported to the target messaging instances transparently from the client regardless of the distance. This targeting can use logical addressing, and in many instances it is preferable. Messages should also be delivered by using a suitable level of reliability. Different messages have different levels, but often once-and-once-only, ensured, and transactional behavior is critical.

Event streaming differentiating capabilities

The following list summarizes event streaming differentiating capabilities:

- ▶ **Stream history:** When retrieving events, consumers are interested in the most recent and historical events. There are many instances when this information is valuable, such as retrieving the historical trends of a system's availability. Therefore, every event must be appended, and made available to consumers, and depending on the configuration, a certain number or volume of events are stored before removal.
- ▶ **Scalable subscription:** An event stream can be consumed by many subscribers with limited impact as the number of subscriptions increases.

- ▶ **Immutable data:** When an event is placed in the stream history, it cannot be changed or removed. It is *immutable* data. Therefore, consumers can rely on the assumption of consistent replay, which reduces the complexity of replicating the data from a consistency point of view.

Although event streaming technology can be forced to implement a messaging solution, or a messaging provider can be used to implement an event solution, this produces an anti-pattern. They both facilitate the communication of data between systems, but the underlying capabilities and usage of the technology is different.

Publish/subscribe considerations

Sections “Messaging providers differentiating capabilities” on page 71 and “Event streaming differentiating capabilities” on page 71 provide a high-level view of the difference between messaging and event streaming technology. Experienced users see the apparent overlap in the publish/subscribe capability. Although they both support publish/subscribe at a high level, there are important technical differences. It is beyond the scope of this book to do a comprehensive comparison, but the following four capabilities provide a useful illustration and best practices to consider:

- ▶ **Event history:** Does the solution need to be able to retrieve historical events either during normal and failure situations? Within a messaging-based publish/subscribe model, the event is published to a subscriber, and after it is received, it is the subscriber’s responsibility to store this information. There are certain situations where the publish/subscribe model can retain the last publication, but it is unusual to use the messaging solution to store historical events. For Apache Kafka, storing event history is fundamental to the architecture; the only question is how much and for how long. In some use cases, it is critical to store this history, and events are a likely solution, and in other cases it might be undesirable from a system resources and security point of view, so messaging is more appropriate.
- ▶ **Fined-grained subscriptions:** When a topic is created in Apache Kafka, it creates one or more partitions within the solution. This is a fundamental architectural concept within the underlying Apache Kafka infrastructure, and is key to why Kafka can scale to handle a massive number of events. Each partition uses up resources, and it is normally advisable to limit the number of topics to hundreds or maybe thousands within a single cluster.

Messaging publish/subscribe has a more flexible mechanism, where the topic can be a hierarchical structure, such as travel; flights; airline; flight number; and seat, allowing more selective subscription points. Subscribing applications can select the events at varying levels of granularity. In addition, messaging publish and subscribe selectors can be used to further refine the events of interest. For applications subscribing, it means that they are far less likely to receive events that are irrelevant to them in the case of messaging publish/subscribe, and an event streaming application that wants only a small proportion of the events likely needs a discarding filter to be applied early in the processing.

- ▶ **Scalable subscription:** If 100 consumers subscribe to all events on a topic, a messaging technology must create 100 messages for each published event (except for multicast publish/subscribe) so that it can be aware of which ones were read by which subscribers. Each of them are stored and, if required, persisted to disk by using system resources.

In the case of event streams, the event is written once, and each subscription has an index corresponding to where they are in the event history. As the number of subscribers increases, there is little more work for the event server to do, so it can scale efficiently. Messaging is a highly scalable technology, so it depends on the number of events that are emitted by the publisher and the number of subscribers whether it is an important deciding factor.

- ▶ Transactional behavior: Both messaging and event streams provide transactional APIs to process events. However, the flexibility and capabilities that are provided by certain messaging solutions (such as IBM MQ) are wider and more mature than the ones that are provided by Apache Kafka. Often in a publish/subscribe solution, the hardened transactional behavior of messaging might not be as critical as in a messaging queuing scenario. For more information about transactional considerations, see [Does Apache Kafka do ACID transactions?](#)

3.3.4 A detailed look at messaging

A typical enterprise environment includes numerous applications that must communicate with each other to provide business value. Messaging enables a decoupled architecture where an intermediary is placed between two applications, systems, or services for communication. This component is often referred to as a *messaging provider*. The messaging provider achieves decoupling by providing *queues* onto which applications can place messages for later retrieval by target applications. This asynchronous communication between the systems means that the applications no longer depend on each other’s availability. Messaging also brings more qualities of service, such as reliable delivery, enhanced security, and workload distribution.

Some examples of messaging providers include:

- ▶ IBM MQ
- ▶ ActiveMQ
- ▶ RabbitMQ

A messaging solution has two key components: The *messaging server* that manages the messages, and a *messaging client* that consumes and produces messages. This messaging client is used by the application for communicating. Messaging providers offer messaging clients in different languages, for example, Java, .NET, C, Golang, and Node.js. The messaging client provides a simple library that enables applications to send and receive messages. Modern messaging solutions also provide a REST-based API to lower the barrier of entry so that application developers may interact with the messaging solution by using HTTP instead of requiring a messaging client. A mature messaging solution also typically supports many operating systems.

When a client communicates with a messaging provider, it can take the role of either or both of the *requesting service* and *providing service*. Requesting services send a PUT message to a queue, and providing services receive a GET message from a queue.

Figure 3-22 on page 75 shows the messaging basic interaction.

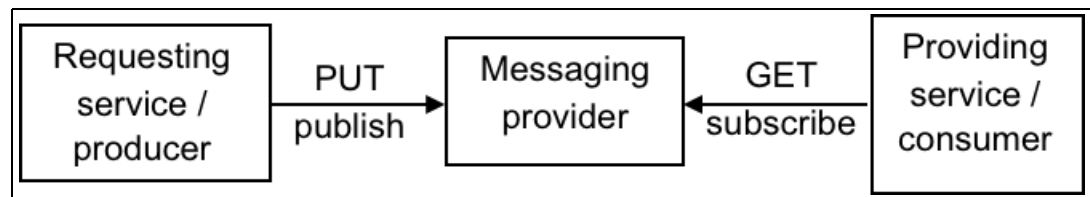


Figure 3-21 Messaging basic interaction

Messaging benefits

Benefits such as reliable, secure, and scalable communication are automatically provided by a messaging infrastructure. Let's discuss these aspects in more detail:

- ▶ **Decoupled communication:** Communicating that uses messaging and queuing removes tight coupling between applications. Other technologies such as REST APIs tightly couple the requester and responder, meaning that the immediate availability of the responder directly affects the requester. Messaging provides an intermediary between the two applications, which means that they are free to operate independently. Therefore, application developers can write less error-processing code.
- ▶ **Reliable message delivery:** Messaging ensures that business-critical information is not lost or duplicated, and is delivered to the recipient once, which removes the need for the application to implement de-duplication or loss-prevention logic.
- ▶ **Security:** Encryption of the messages in flight at a transport layer (TLS), and message content (payload), including when *at rest* can be configured within the messaging platform, removes the responsibility from the application developer. This scenario can be achieved end to end, which separates it from simplistic HTTPS-based encryption on synchronous calls. HTTPS encryption must be terminated, typically as it passes through the DMZ. Because messaging provides its own transport channels, it also means that security characteristics must be finalized independently of the application logic, so no changes to the code are required.
- ▶ **Workload management:** When applications receive a high volume of requests, those applications can become overloaded and non-responsive. Application developers often implement defensive code to protect applications from this scenario. The messaging server provides a buffer by allowing work to be temporarily stored until the application is ready to process. Instead of the application being flooded with too much work, it instead pulls work as it becomes able to process. This drastically simplifies management of workload through the application.

Teams are under increased pressure to deliver enterprise-grade applications on ever more aggressive deadlines to meet the needs of the business. Using messaging can bring qualities of service that otherwise would require significant complex coding and configuration to achieve.

Messaging protocols

Messaging providers support a range of communication protocols for exchanging messages. Here is some guidance about which protocol to use in which circumstances:

- ▶ **Proprietary protocol:** For example, IBM MQ often provides the best performance because it is explicitly designed and implemented for the platform. It also provides access to unique features that may be available on only IBM MQ, such as the coordination of two-phase commit transactions.
- ▶ **Specialized protocols for particular use cases:** For example, IBM MQ Telemetry Transport (MQTT) was created for low bandwidth, Internet of Things (IoT), and publish/subscribe based interactions.
- ▶ **Open Standard Protocol:** For example, Advanced Message Queuing Protocol (AMQP) is an OASIS open standard protocol for interacting with a wide range of messaging providers. Its focus is on interoperable communication, which allows flexibility in the future to simplify the effort of migrating between messaging providers.
- ▶ **HTTP REST-based APIs:** The messaging server can expose an HTTP REST-based API and remove the need for a messaging client. Application developers are normally programming in languages that have built-in library support for HTTP, which lowers the barrier to entry, but it does make it harder to ensure a *once only* delivery of messages because HTTP is not a transactional protocol.

To support these protocols, many programming languages provide language-specific messaging APIs, for example, Java Messaging Service (JMS). JMS defines the interfaces, and the vendors provide the Java libraries so that an application developer can work with the messaging solution without having to know the underlying messaging protocol.

Messaging in agile integration

Messaging provides several capabilities that support an enterprise-class agile integration. Although it is beyond the scope of this book to describe messaging comprehensively, it is important to highlight common use cases where messaging is required:

- ▶ Decoupled communication: Messaging enables applications to decouple communication, relieving the apps from burdens such as routing, target application availability, and secure communication. The messaging infrastructure provides these capabilities. A source application sends a message to a logical address (called a queue), and the messaging infrastructure delivers this message to the target application. At the time of sending, the application might be unavailable either due to a failure or planned maintenance window. Without messaging, a temporary failure of the target application has an immediate impact on the calling application.
- ▶ Pulling requests: Messaging enables a pattern where target applications pull workload from the messaging infrastructure. Compare this to the more typical direct connections of HTTP where work is distributed from a load balancer and pushed directly to the target applications.

As shown in Figure 3-22, messaging *protects applications during burst workloads* when the inbound workload is temporarily above the throughput that the servers can handle, so requests are buffered. Depending on the scenario, they can be buffered with a time to live to prevent stale messages being processed late.

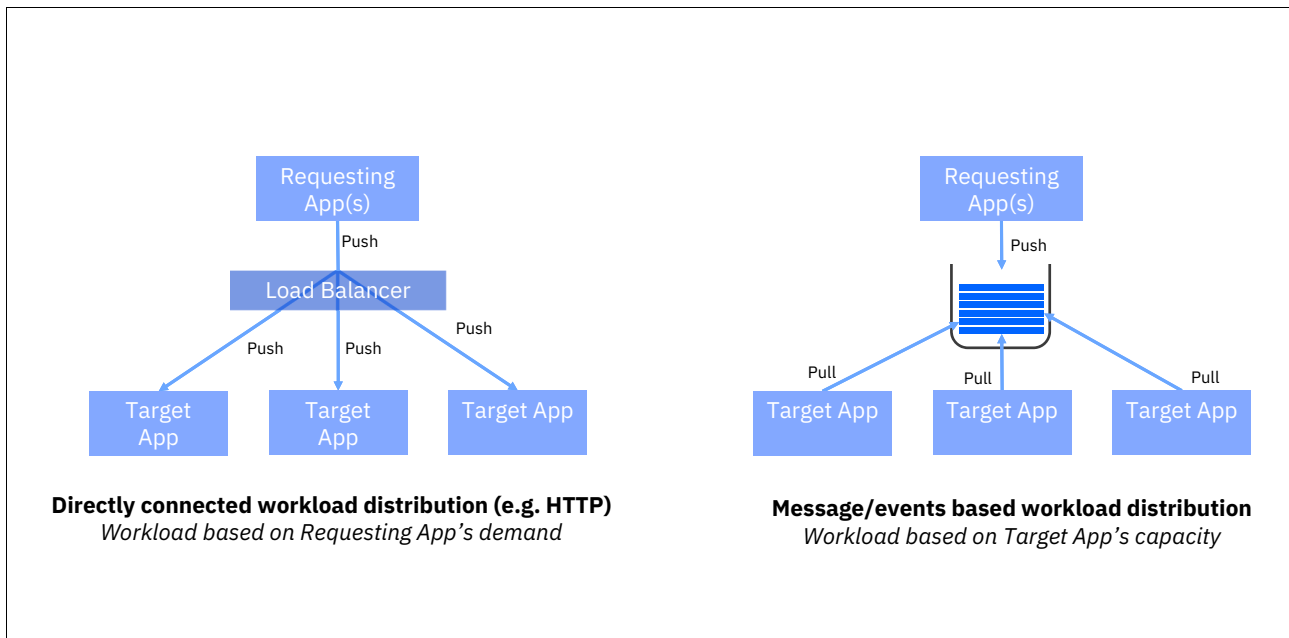


Figure 3-22 Messaging protects applications during burst workloads

- ▶ **Communication for business-critical applications:** Some operations within an integration architecture do not require ensured delivery, for example, read operations or calls to idempotent services where the caller can retry on failure. The convenience of HTTP is a natural protocol for these types of interaction due to its universal availability across platforms, operating systems, and applications. However, when completing non-idempotent data-changing transactions on business-critical applications, HTTP has limitations that can cause frustration and confusion for your clients, and a damaged reputation for the business. We have all experienced the situation when we are completing an order online when suddenly the order goes into an indeterminate state and it is unclear whether the order has occurred. In a stateless integration architecture based on end to end HTTP, recovering from fragile networks and application availability that cause the above situations is challenging if not unfeasible. Messaging technologies such as IBM MQ provide ensured delivery, where situations that can cause an end to end failure are reduced, and there is improved recoverability in these cases.
- ▶ **Hybrid multicloud business critical communication:** Within a traditional enterprise with a single data center, your business-critical communication between applications is bound to the data center. As enterprises embrace the cloud, this leads to a hybrid multicloud enterprise with applications hosted across on-premises, public, and private clouds. When communicating between clouds, the network becomes the weak link. Here, reliable cross-cloud messaging is critical, enabling applications within each cloud to confidently write messages locally within their own cloud and know that the messaging framework will solve the problem of reliably getting them to their destination cloud once.

Figure 3-23 shows multicloud messaging.

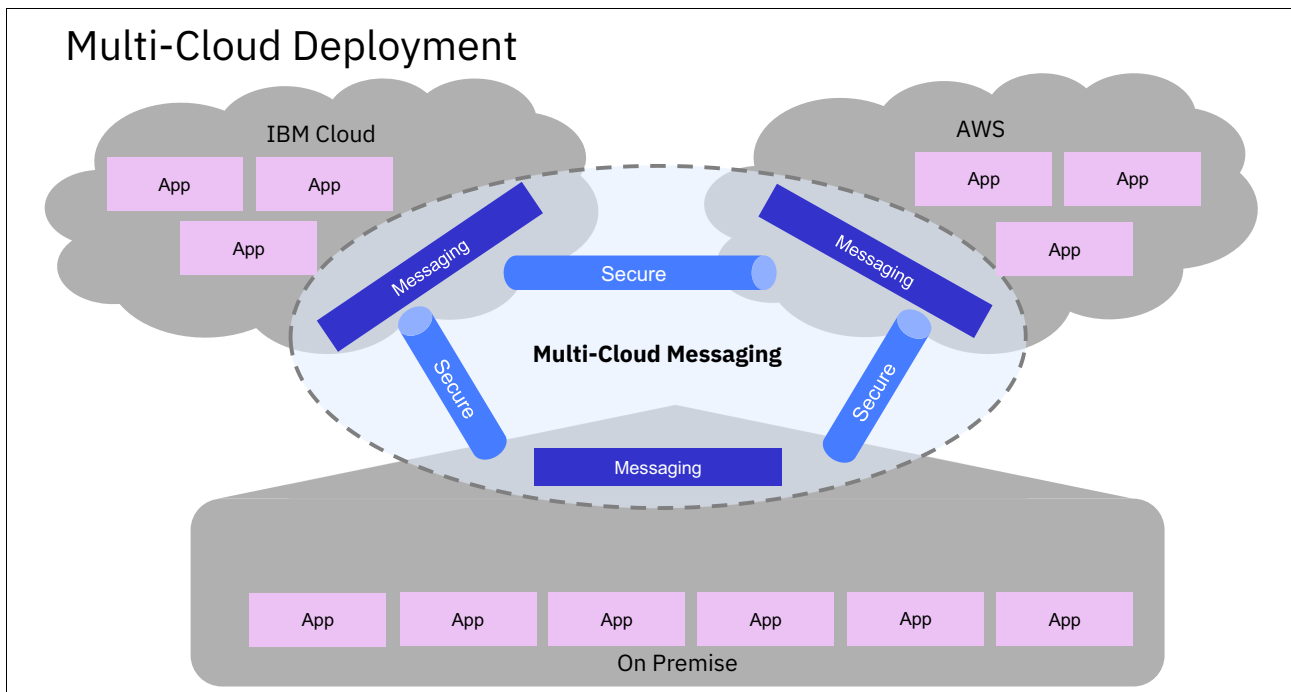


Figure 3-23 Multicloud messaging

3.3.5 A detailed look at event streams

With event streaming technology, an enterprise can collect, store, and distribute events across all their applications at massive scale. When an application interacts with an event stream technology, it may be either or both of the following items:

- ▶ Providers: Emits and publishes events.
- ▶ Consumers: Reads events.

Consumers of events do not necessarily want to read all events passing through the event streaming installation, so publishers of events specify a *topic*, and subscribers listen only to the topics in which they are interested.

There are two key components to any event streaming technology: A *server* that stores events and manages the topics, and a *client* that allows applications to interact as a *provider* or *consumer*.

Several technologies provide event streaming capabilities. The market is leaning toward Apache Kafka as the *de facto* standard. Apache Kafka is an open source project that was originally created by LinkedIn and donated to the community in 2011. Some vendors provide commercially supported versions of Kafka; our implementation is IBM Event Streams. IBM is also a key contributor to the Apache Kafka open source project as committers.

Capabilities for an event streaming technology

The following list highlights three core capabilities for an event streaming technology:

- ▶ Stream history: Events that are emitted in an event stream topic are stored and are *not* removed when a subscriber receives the event. The topic can be considered a stream history of all the events that are emitted, and allows subscribers to rewind to different locations of the topic. Topics are configured with persistence, and the stream history is stored so it can be recovered in a failure. The size of the stream history is configurable based on the scenario that is required.
- ▶ High performance: Event streaming technology is designed to handle millions of events a second. This performance is available to the producers who are creating the event and the consumers.
- ▶ Scalable subscription: From a consumption point of view, event streaming technology is designed so that increases in the number of subscribers to a topic has a minimal impact on the resources.

Figure 3-24 on page 78 shows event stream scalable consumption.

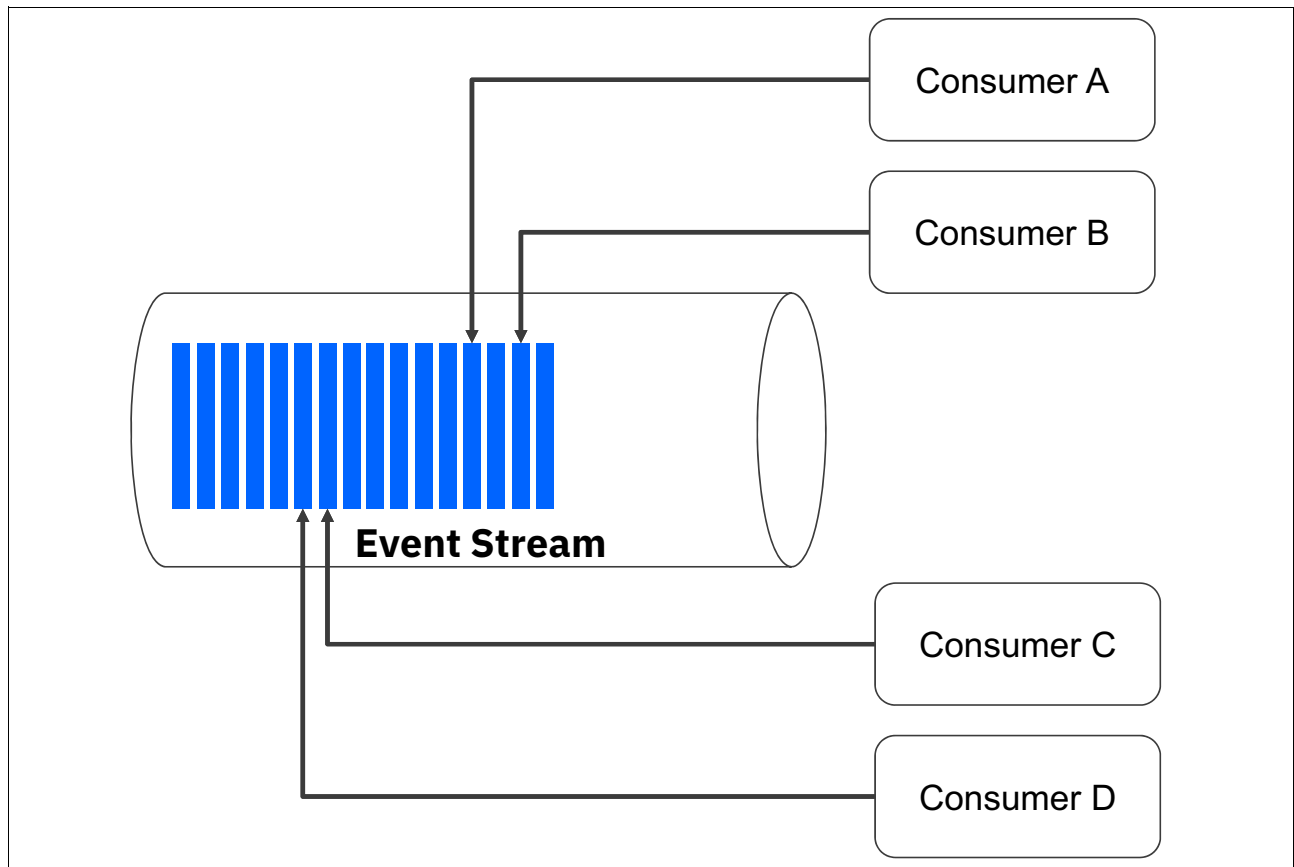


Figure 3-24 Event stream scalable consumption

- **Decoupled communication:** Communicating by using events removes tight coupling between applications in much the same sense that messaging does. Other technologies such as REST APIs tightly couple the requester and responder, meaning that the immediate availability of the responder directly affects the requester. Event streaming provides an intermediary between the two applications, which means that they are decoupled. This configuration allows application developers to write components that have fewer dependencies on other components at run time.

Event streaming in agile integration

Let's look at some of the use cases where event streaming can be used to improve agility in modern application landscapes:

- **Enable responsive cloud-native applications:** Cloud-native applications are being built across the enterprise in public and private cloud environments. These applications need access to business data that might be on-premises. To deliver a responsive application, this business data must be available locally instead of calling to on-premises for each piece of data. A local event stream within the cloud can provide this capability. It acts as a local data store from which the cloud-native applications can build their own data models, decoupling these applications from the core business applications on-premises.

Figure 3-25 shows enabled responsive cloud-native applications

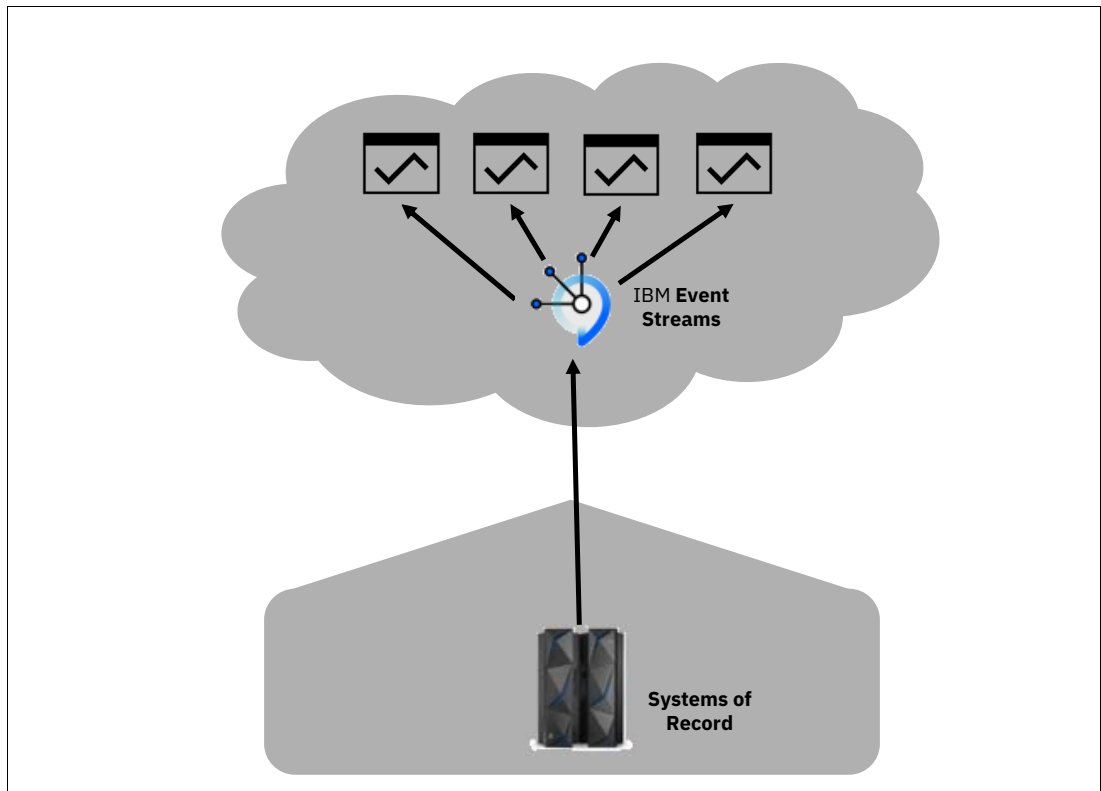


Figure 3-25 Enabling responsive cloud-native applications

- ▶ Adding real-time data streams to power reactive applications: An enterprise is rich with events, and many applications already are creating events. However, without an event streaming infrastructure, there is no facility to effectively collect, manage, and distribute these events, which inhibit business agility that can be achieved in an event-driven integration model. Figure 3-26 shows the addition of real-time data streams.

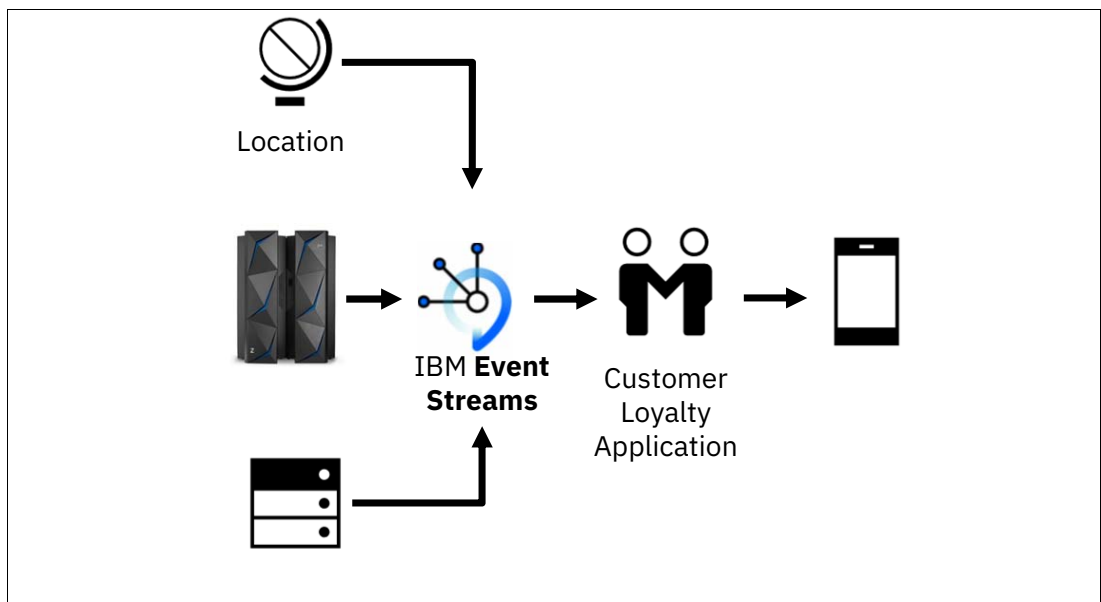


Figure 3-26 Adding real-time data streams

- ▶ Gaining insight from historical data: By listening to and processing an event stream, you can gain insights based on time-based event patterns. Furthermore, because event streams also provide an event history, you can use this history to train machine learning models to progressively evolve the sophistication of the insights, as shown in Figure 3-27.

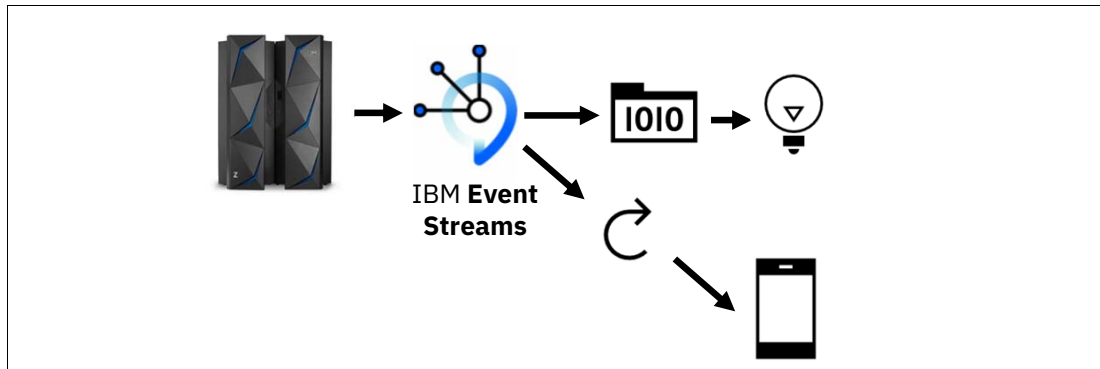


Figure 3-27 Machine learning with event streams

3.4 Capability perspective: Files and Business-to-Business

Modern approaches to integration are often focused around synchronous exposure over web services and RESTful APIs. It is easy when focused on these patterns to forget that for many enterprises, especially in sectors such as retail, there is still significant reliance on traditional Business-to-Business (B2B) integration, such as Electronic Data Interchange (EDI).

Can an API-led strategy be used for B2B integration? To what extent can you enable the same type of collaborations over APIs between business partners and the enterprise as are done by using EDI over a Value Added Network (VAN)?

Let's explore this situation from a few different perspectives: architecture, time, and cost.

- ▶ Architecture

B2B data exchanges through EDI have three core technological elements: transport protocol, data transformation, and partner management. They also are present in an API management capability, such as IBM API Connect, in a slightly different form.

- ▶ Time:

- Rapid onboarding. New partners must be onboarded quickly. Although partner management has been a core part of B2B and EDI for some time, API management technology has in recent years placed a focus on developer community enrolment to support rapid data access and exchange.
- Development time. Most modern applications already come with an API by default to enable them to be more quickly incorporated into new solutions. Furthermore, most modern languages can call APIs natively.
- Real-time information exchange. Traditionally, B2B and EDI patterns were focused around supporting messages for purchasing and supply chains. They are often batch-based in nature. With an increased business need for real-time data, for example, to manage critical path supply chain decisions, APIs better enable real-time data exchanges between applications and business partners.

- ▶ **Cost:** Dedicated B2B networks can be costly to implement from scratch. An API-led approach to B2B allows a shared infrastructure that servers both digital application APIs and B2B APIs.

[Should Business APIs Replace EDI?](#) goes into more detail about the decision criteria for whether an enterprise might replace EDI with APIs.

We can certainly debate the merits of migrating some or all of an enterprises' partners over to (REST) API interactions. However, traditional integration methods such as EDI are not going to disappear just because enterprises' desire to undergo digital transformation. We must also consider how we might modernize the existing B2B interactions.

A common alternative transport protocol is IBM MQ, which is often used in B2B. However, using IBM MQ assumes that your business partner also uses IBM MQ (or a compatible messaging client that IBM MQ supports, such as JMS). The use cases for IBM MQ connectivity are described in 3.3, "Capability perspective: Messaging and event streams" on page 68.

You can use EDI payloads over IBM MQ or HTTP transport and then use App Connect to transform the EDI payload. For an example of using EDI standards within IBM App Connect, see [IBM Integration Bus tutorial: Modeling UN/EDIFACT data using DFDL schemas](#).

EDI can also be performed over HTTP by using Applicability Statement 2 (AS2). However, you likely need a stateful data store to store the Message Disposition Notifications (MDNs).

However, there will still be circumstances where your business partner must use a traditional EDI message set and a VAN. For this scenario, there are a couple of alternative ways that a B2B pattern might be implemented to support an agile integration:

- ▶ Use a managed service or software-as-a-service (SaaS) product.

The simplest way is using a specialist B2B product as a SaaS or managed service solution. The EDI message mappings and a secure (VAN) connection to the business partners are then provided by the service.

There are many providers of B2B and EDI solutions in the marketplace that provide a managed or SaaS offering, such as [IBM B2B](#).

The EDI mappings are stored within the SaaS service. The key connectivity consideration is then how the B2B SaaS product connects to the core enterprise. You can do so by using standard protocols such as IBM MQ or HTTP.

- ▶ IBM DataPower Gateway

Within the IBM Integration Technology portfolio, IBM DataPower Gateways can be used as a physical appliance, virtual appliance, or run in a container. One of the IBM DataPower Gateway variants include support for native B2B EDI protocols, along with an internal B2B store.

The use cases for implementing IBM DataPower Gateway are described in 5.4, "Integration security: IBM DataPower Gateway" on page 152, and using DataPower Gateway supports a use case where EDI and APIs are supported on the same technology architecture for B2B integration.

3.5 Hybrid and multicloud considerations

Hybrid cloud and multicloud often are used interchangeably. The main difference is that with hybrid cloud, an organization uses a combination of deployment modes, and with multicloud, an organization uses multiple cloud services from more than one cloud provider vendor. However, it is becoming more common across companies to adapt both of these approaches.

3.5.1 Multicloud: Multiple cloud services

Multicloud is a cloud approach to support multiple cloud services from more than one vendor. By using it, companies can use the strength and unique offerings from different public cloud vendors to achieve portability without cloud platform lock-in, for example, running API Connect on both IBM public cloud and Amazon Web Services (AWS).

3.5.2 Hybrid Cloud: Multiple deployment modes (public, private, and legacy)

Hybrid cloud is a platform for applications and infrastructure that integrates traditional IT with a combination of public, private, or managed cloud services. In all its forms, hybrid cloud facilitates flexibility and portability for applications and data. With hybrid cloud, companies can more effectively manage speed and security, latency, and performance. Every application and service can be deployed and managed where it makes the most sense.

For more information, see [What is multicloud?](#)

3.5.3 Evolution of API deployment modes

In the early days of the API economy era, APIs were classified as internal and external based on how they were consumed:

- ▶ **External APIs:** External APIs present the API provider or business an opportunity to share certain data sets, services, and capabilities with developers to use the business's assets to develop innovative new applications and allow for existing applications and services to be modified. These APIs can be open to any developer who wants to sign up or open only to select business partners to have greater control over how the data is used.
- ▶ **Internal APIs:** Organizations use APIs internally or privately to develop new ways of operating and managing their business. These internal APIs can be developed to more efficiently process internal documents, manage processes, share information, account for assets, and other business processes to drive increased productivity. Businesses also use internal APIs to build publicly available applications. Internal APIs are the predominant category of APIs because most APIs start privately inside organizations and later evolve for public or partner access with some rules and restrictions.

Internal and external APIs followed a similar deployment pattern in a traditional DataPower workload where external facing DataPower services are hosted in a secure network zone and internally in a private or internal network zone.

- ▶ **Cloud-based APIs** are the new emerging classification that is the natural evolution of the external APIs as certain data sets, services, and capabilities are moved across multiple cloud environments, either born directly on cloud or migrated. Therefore, it is crucial to manage these APIs.

3.6 Use cases driving hybrid and multicloud adoption

Here are some use cases to review by clients considering hybrid or multicloud adoption.

3.6.1 Multicloud strategy

Customers want to use the strength and unique offerings from different cloud vendors, but also want to have a consistent operation and runtime environment so that they can achieve portability without cloud platform lock-in.

3.6.2 Cloud bursting and scalability

If you have private cloud environments running on-premises and want to expand the cluster or private cloud to an external infrastructure only in certain special conditions (such as load testing) or for a bursting workload, hybrid and multicloud topologies can meet these needs.

3.6.3 Disaster recovery

Because the same workload can be easily and quickly provisioned, external cloud providers can be a great place to act as a disaster recovery (DR) data center.

3.6.4 Application affinity

Generally, API back ends are distributed across multiple on-premises and off-premises clouds. Therefore, you should prefer to deploy APIs near systems of records to reduce latency.

3.6.5 Regional flexibility

Exposing APIs from different geographical areas can expand your business into new regions because customers expect fast response (low latency) and secure access to APIs from anywhere in the world. Geographical distribution of API requests from a physically closer cloud unit reduces latency and ensures adherence to local policies that require certain data to be physically present within the area or country.

3.6.6 Geographical high availability

There is a limited value in deploying APIs to another region if the back-end application is not present in that region or consumers accessing APIs from different regions. In that respect, geographical HA targets both application affinity and regional flexibility concerns and increases global availability profile and performance of APIs.

3.7 References

For more information about new material that is related to agile integration, see [Agile integration architecture: Useful links](#).

The following resources also might be useful:

- ▶ Agile integration eBooklet
<http://ibm.biz/agile-integration-ebook>
- ▶ The fate of the ESB
<http://ibm.biz/agile-integration-links>
- ▶ Microservices, SOA, and APIs: Friends or enemies?
<http://ibm.biz/agile-integration-links>
- ▶ The hybrid integration reference architecture:
<http://ibm.biz/HybridIntRefArch>



Cloud-native concepts and technology

Throughout the preceding chapters we have regularly mentioned cloud-native and related terms. Although we briefly qualify those terms as they appear, we recognize that some readers might be relatively new to cloud-native concepts. Therefore, the beginning of this chapter provides an overview of the core cloud-native concepts and technologies.

For readers familiar with the core concepts, the latter part of the chapter considers some of the typical challenges, such as the definition of application boundaries, more advanced technologies such as the service mesh, and a look into the future of the cloud-native infrastructure.

The following topics are covered in this chapter:

- ▶ Defining cloud-native
- ▶ Key elements of cloud-native applications
- ▶ Twelve-factor apps
- ▶ Container technology: the current state of the art
- ▶ Cloud-native is not for everyone, nor for everything
- ▶ Realizing the true benefits of containerization
- ▶ Application boundaries in a container-based world
- ▶ Service mesh
- ▶ Cloud-native security – an application-centric perspective
- ▶ The future of cloud-native

4.1 Defining cloud-native

Cloud-native concepts existed before the term itself came into use. The beginning of cloud-native was when public cloud vendors began providing easy and affordable access to elastic instances of compute power. How can you write applications to capitalize on the flexibility of this new infrastructure, and what business benefits can you achieve?

Cloud-native methods and technology are still evolving, but the core business objectives that cloud-native applications set out to achieve remained the same. The business benefits of cloud-native are:

- ▶ **Agility:** Enable rapid innovation that is guided by business metrics.
- ▶ **Resilience:** Target continuous availability that is self-healing and downtime-free.
- ▶ **Scalability:** Provide elastic scaling and limitless capacity.
- ▶ **Cost:** Optimize the costs for compute and human resources.
- ▶ **Portability:** Enable free movement between locations and providers.

Note that the “cloud” in cloud-native refers to the public cloud and the infrastructure that is rapidly self-provisioned, elastically scalable, and has apparently limitless capacity. Cloud-native solutions may be built on-premises or in dedicated environments, but it is the cloud-based approach to platform provisioning that is important.

Important: Microservices are associated with cloud-native, but they are not equivalent. Microservices architecture is a concrete example of how to build applications in a cloud-native style.

A key reference is the principle of 12-factor applications. Although the 12 factors make no claim to be all encompassing guidelines for cloud-native, they are one of the earliest examples of how to write applications with a cloud-native intent.

“The twelve-factor app is a methodology for building software-as-a-service (SaaS) apps that:

- ▶ Use *declarative* formats for setup automation, to minimize time and cost for new developers joining the project;
- ▶ Have a clean *contract* with the underlying operating system, offering *maximum portability* between execution environments;
- ▶ Are suitable for *deployment* on modern *cloud platforms*, obviating the need for servers and systems administration;
- ▶ *Minimize divergence* between development and production, enabling *Continuous Deployment* for maximum agility;
- ▶ And can *scale up* without significant changes to tooling, architecture, or development practices.”¹

Despite its age (2011), this list encapsulates many of the key concepts and guidelines for cloud-native, and the 12 factors themselves still have relevance even if the specific terms we use have evolved in the past few years. It is a useful backdrop, and we describe the 12 factors in 4.3, “Twelve-factor apps” on page 100.

We should recognize the importance of contributions from [early adopters in cloud-native](#).

¹ Source: <https://www.12factor.net/>

Adopting cloud-native practices enabled companies like Netflix to radically innovate at a pace that their more traditionally architected competitors could not, which contributed directly to their sustained market leading status today.

Let's look at what makes something a cloud-native application.

4.2 Key elements of cloud-native applications

Cloud-native brings together elements that have been developing for years, and in some cases decades. Each one of them are beneficial on their own, but together these benefits multiply and deliver on the business benefits of cloud-native:

- ▶ **Modular components:** Small, loosely coupled, and minimal interdependencies. They enable greater agility and optimization of resources, and provide for more specific resilience models. An example is the microservices architecture, which aligns and overlaps with many of the other cloud-native characteristics.
- ▶ **Prefer stateless:** Ideally, components can be re-created by starting a fresh copy of an image. Provides elastic scalability and reduces environment configuration divergence.
- ▶ **Image-based deployment:** Lightweight file-based packaging of all local dependencies into an immutable image. Improves testing confidence, enables simpler re-creation of environments for functional and performance testing, and simplifies problem diagnosis.
- ▶ **Lightweight runtimes:** The run time starts and stops in seconds or less, reacts instantly to workload demands, and enables fast recovery from failure and overall optimization of resources.
- ▶ **Elastic, agnostic infrastructure:** Self-service, self-scaling, and self-healing infrastructure that runs all workloads in the same way. Removes provisioning processes. Optimizes resource costs. Simplifies component deployment and scaling. Improves portability.
- ▶ **Log-based monitoring:** Monitoring by analysis of centralized logs. No direct connectivity to live run times. Enables centralized standards-based monitoring capabilities and simplifies the run times.
- ▶ **API-led intra-app communication:** APIs for communication among the modular components of the application. Standards-based and runtime agnostic, so it enables common routing and load balancing and the use of a polyglot of runtime types.
- ▶ **Event-driven architecture:** Asynchronous patterns where deep runtime decoupling is required. Improves availability and reduces response times.
- ▶ **Agile methods:** Short, business-led iterations from requirements to implementation. Keeps the direction of travel in line with business expectations.
- ▶ **Continuous Integration and Continuous Delivery and Deployment (CI/CD):** Automation of the building, testing, and deployment of code to environments. Facilitates rapid fine-grained iterations on changes with immediate feedback. Improves consistency of code and testing. Reduces deployment risk.
- ▶ **DevOps:** Near merging of development and operations. Ensures the route to live is short, and feedback from production is correctly prioritized.

Let's look at each of these items in more detail.

4.2.1 Modular components

Until relatively recently, to use hardware and software resources efficiently it was necessary to build software in large siloed blocks of code. More recent developments in technology, containers being a key enabler, have made it realistic to break up applications into smaller pieces and run them in lightweight run times. Eventually, these pieces became small enough that they deserved a name, and they were termed *microservices*. These small independent microservices components can be changed safely in isolation, scaled individually, and managed more ruthlessly.

The core benefits of a more fine-grained microservices approach are:

- ▶ **Greater agility:** They are small enough to be understood in isolation and changed independently.
- ▶ **Elastic scalability:** Each component can be scaled individually by using efficiently a cloud-native infrastructure so that their resource usage can be tied to the success of the business model.
- ▶ **Discrete resilience:** With suitable decoupling, changes to one microservice do not affect others at run time. So, they can provide the resilience that is required by 24/7 online applications.

Without question, microservices principles can offer significant benefits under the correct circumstances. However, choosing the correct time to use these techniques is critical, and getting the design of highly distributed components correct is non-trivial. Deciding the shape and size of your microservices components is only part of the story; there is an equally critical set of design choices around how fine-grained the components should be and the extent to which you decouple them. You must constantly balance practical reality with aspirations for microservices-related benefits. In short, your microservices-based application is only as agile and scalable as your design is good and your methodology is mature.

In this section, we focused on the fine-grained aspect of microservices. However, to be successful with microservices architecture requires a much broader scope than simply breaking applications up into smaller pieces. There are implications for architecture, process, organization, and more, many of which are also elements of cloud-native, so we cover them in later sections.

The microservices architecture is often inappropriately compared to service-oriented architecture (SOA) because they share words in common and seem to be in the same conceptual space. However, they relate to different scopes. Microservices is about *application architecture*, and SOA is about *enterprise architecture*. This distinction is critical, and is explored further in [Microservices versus SOA: How to start an argument](#).

4.2.2 Preferring stateless

Clear separation of state within the components of a cloud-native solution is critical. It enables the orchestrating platform to manage the components in an optimal way. There is a strong preference that components should be stateless, and for those that are not, their needs must be specified to the platform.

Knowing which components are truly stateless enables the platform to manage them differently. For example, it can more ruthlessly start and stop components as required. More importantly, it means the platform is free to create many replicas of the component and distribute workload across them. The platform can route requests to any existing or new replica and expect the same results.

From a design point of view, statelessness means no changes are made to the internal configuration or the data that is held by a component after it starts that makes it different from any other replica. Nothing should be saved uniquely to a replica that it must remember if it were stopped and then started again because this impairs the orchestration platform's ability to manage replicas.

A simple example of the differences is that stateful components are harder to scale, especially scaling down, because state must be re-located. Availability is another example. If a whole node (for example, a whole operating system instance) in an environment failed, then it becomes much harder (and slower, and possibly even impossible) to re-instate the components that were running on that node to a new healthy node.

Examples of statefulness include:

- ▶ **Session affinity:** Expecting a specific user or consumer's requests to come back to the same component on their next invocation, typically due to specific data caching. This state used to be relatively common in web applications but is now discouraged, instead, push the state to the client or to back-end systems.
- ▶ **Local message queues:** If a component has a local message queue that cannot be seen by other components, this is component-specific state. If the component stops, these messages become orphaned. So, remote queues should be used where possible.

This concept should not be confused with whether the component interacts with a downstream system that holds a state. For example, a component might interact with a database or a *remote* message queue that persists state. However, that does not make our component stateful just because the database or message queue that it talks to holds a state. It is just passing stateful requests onto a downstream system. If no state is held in the component itself, it is still considered stateless.

The component becomes stateful only if it were, for example, to cache the data in the back-end database *within* the component so that future requests can respond quicker by not needing to go all the way to the back-end system. Now, you would have a stateful component because different replicas might have different data that is cached. You can quickly see how this statefulness would make the component harder to manage. For the cached data to be of any real value, you must ensure that requests that are likely to use the cached data go back to the same replica. This is an example of *session affinity*. Suddenly, the orchestration platform could not just perform simple load-balanced routing. Instead, it would need to know which of the replicas has cached which pieces of data. You can imagine this affinity might become even more important and complex when you consider how you would ensure that the cache is kept in sync with the back-end systems.

Ultimately, some components in a cloud-native environment must be stateful; the state must be stored somewhere. The point is that the designer of a component should be clear about whether it is stateful or stateless, and should understand the design, deployment, scaling, and availability issues that might arise should their component be stateful. Platforms such as Kubernetes (K8s) have separate mechanisms for handling stateless and stateful components. We describe this topic later in this chapter.

4.2.3 Immutable deployment

Component deployment should be immutable, that is, the component should be deployed from a known pre-built image. Following its initial startup, it should then not be further configured at runtime. Changes to the configuration should be done by creating an image and replacing the components with new ones based on that new image.

Image-based deployment is a mechanism for file-based packaging of all local dependencies into an immutable image. The image contains more than the code that is required to run the component; it also contains the language runtime and enough of the operating system for it to be run independently within a container environment. This approach improves testing confidence, enables simpler re-creation of environments for functional and performance testing, simplifies problem diagnosis, and contributes to the simplicity of elastic scaling.

In Figure 4-1, you can see what is deployed in a release into a traditional versus an image-based deployment.

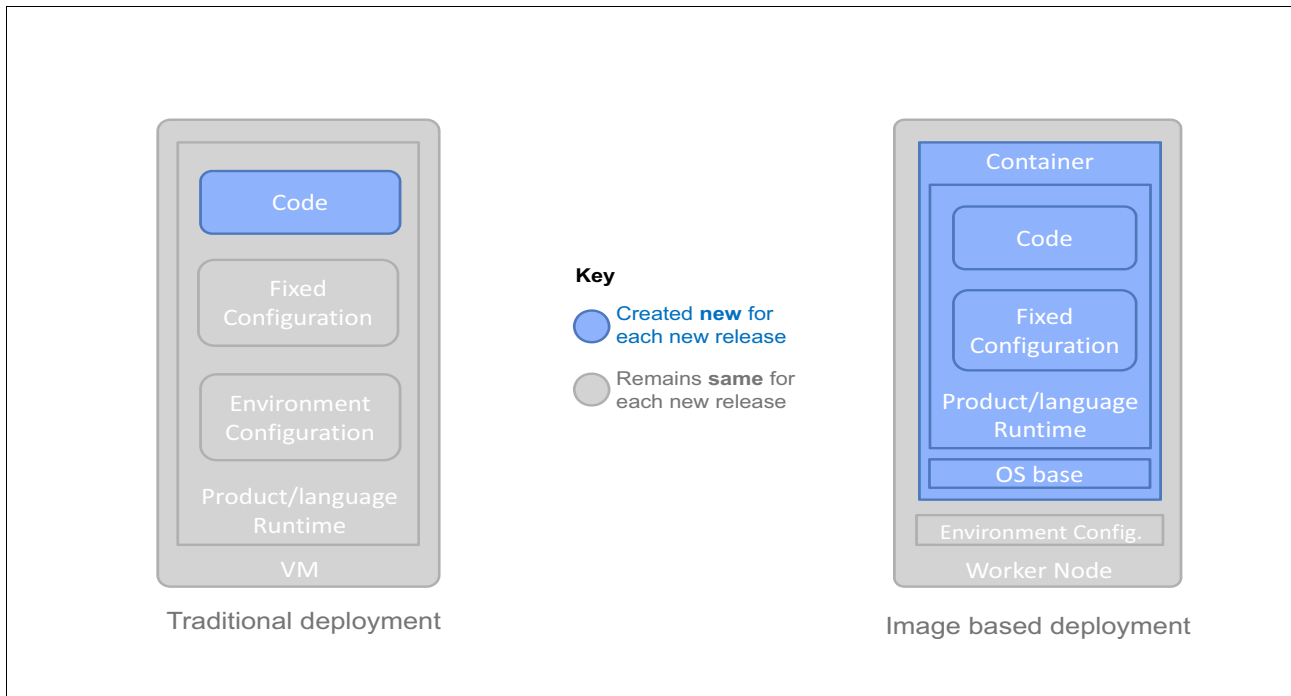


Figure 4-1 Traditional versus an image-based deployment

It is easy to see why image-based deployment has the opportunity to create much more consistent deployment results because all the dependencies are packaged alongside the code. Here are some practical examples that illustrate Figure 4-1:

- ▶ **Code:** This is the code that you write and deploy as a unit. In Java, this is the Java in your JAR, EAR, or WAR file. In Node.js, this is the JavaScript in JS files. In IBM App Connect, this is the flows, maps, and more in your BAR file.
- ▶ **Fixed configuration:** These are the dependencies on which your code relies. If your code is making an HTTP call, this is the HTTP package that you are using. If you are using a database connection, these are the ODBC or JDBC classes. It also includes any configuration details that do not change from one environment to another.
- ▶ **Environment configuration:** These are the details that are expected to change from environment to environment (they are different in a test environment from what they are in production). An example here is if you are integrating with something over HTTP, this configuration is the HTTP URL. If you are connecting to a database, then this configuration is the host, port, user name, and password.
- ▶ **Runtime:** This is what is running your code. It can be either your Node.js run time, Java JRE, Liberty server, or in our case the App Connect run time. It is the run time that interprets and runs your coded artifacts.

Notice how in the traditional environment that an enormous amount is assumed to be already present in the environment and (hopefully) correctly installed and configured. The operating system, the product and language run time, and the fixed configuration must all be present before you can deploy an application to it.

In the cloud-native approach, your image supplies the application code and all its dependencies, including the product and run time, including its dependencies within the operating system. You can be sure that what you deliver to any environment is always consistent.

To understand a bit more about the differences, let us look at how you can build and nurture a traditional server. We typically build the server by starting with a raw operating system, starting the operating system, installing the run time, adding any necessary fix packs, and performing any necessary environmental configurations. We might perform some verification tests on this server, and then start deploying our applications.

Figure 4-2 shows an overview of this process.

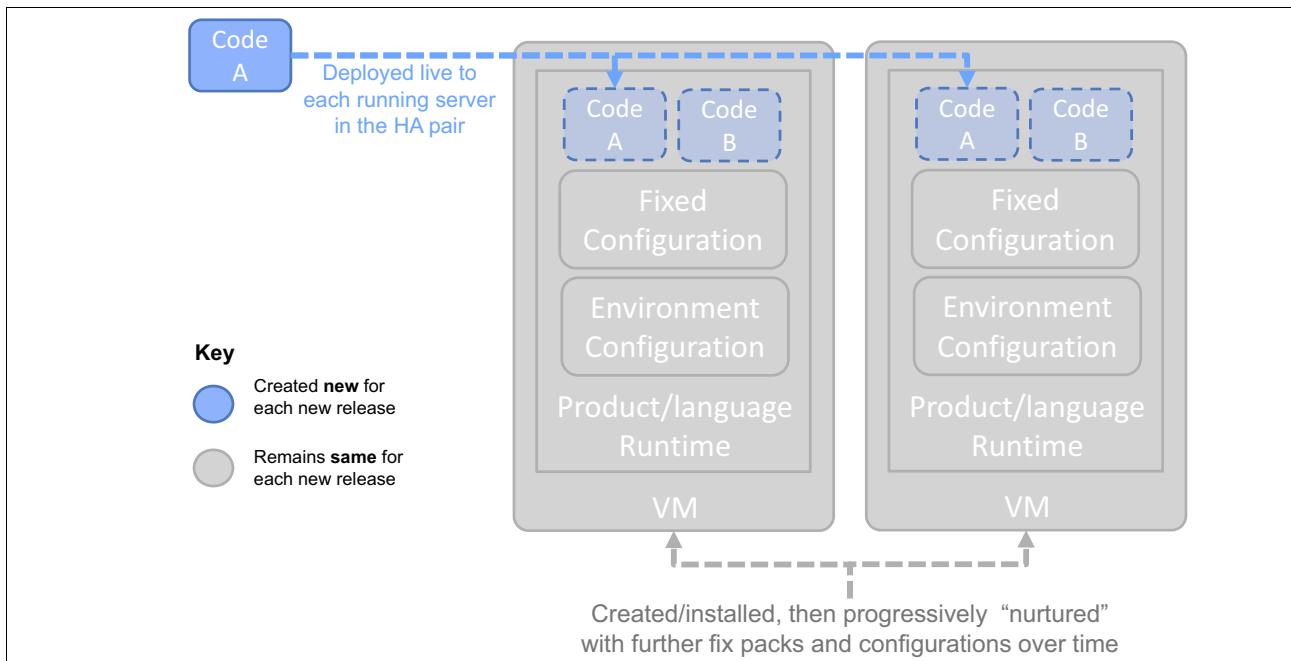


Figure 4-2 Building and nurturing a traditional sever

All that installation and deployment work had to happen before you could service a single request. Over time, you might decide to add further operating system upgrades or fix packs to the run time. All these actions had to be done with the server running, and required special commands to be run to install or deploy things to the live server.

What are the challenges of this approach? The build of the server requires many steps that involve running proprietary commands on a live running server. These are complex commands, and even if they are automated it is hard to be 100% sure that after the sequence of commands you end up with exactly the same server configuration in each of a high availability (HA) pair or across multiple environments, such as development, test, and production. When further ongoing configuration changes and deployments are performed, they can potentially result in configuration divergence between environments. This situation can lead to the classic diagnosis problem of "Well, it worked in my environment!" Creating an exact replica of an environment becomes more difficult the more *nurtured* the environment is.

The alternative is an image-based deployment, as shown in Figure 4-3. Servers are not administered when they are live. The installation of the product and its fix packs and the addition of applications artifacts are ideally done by laying down files on the file system. This action can then be captured as a “container image” and stored in a repository.

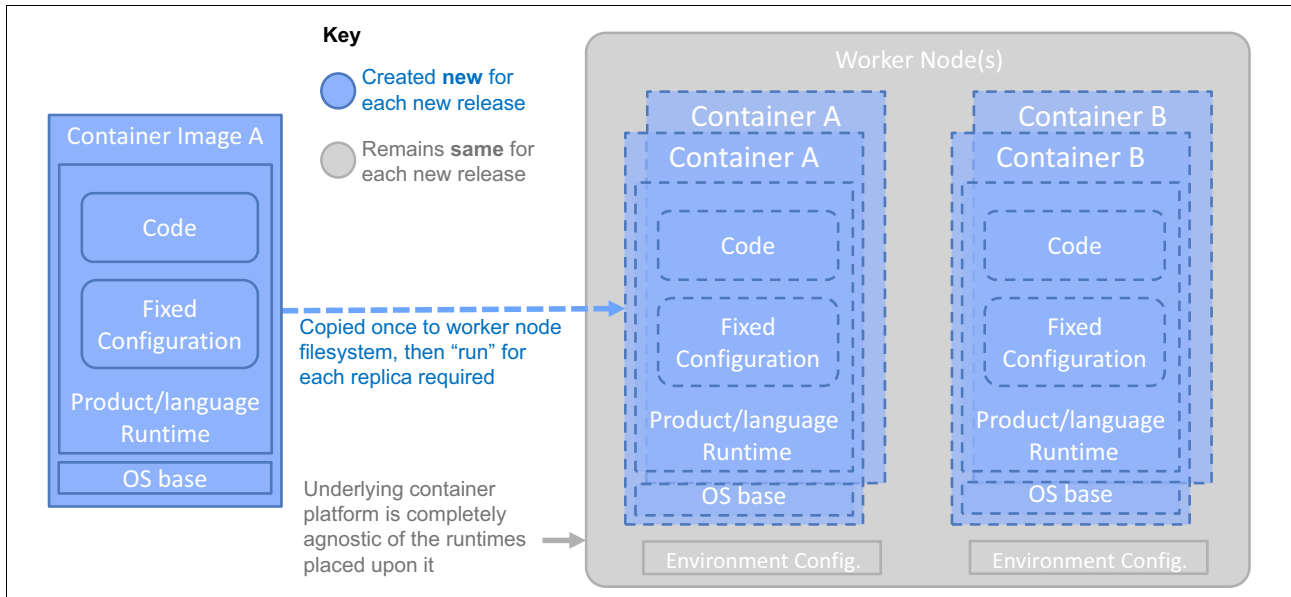


Figure 4-3 Image-based deployment

The image has further metadata, such as what process within the image to run on start and which ports to open. From this point, the image is a black box. You do not need to know anything about what kind of run time or application code is inside it to start it. If it fails, you can reliably and rapidly start a new one. You can consistently start several instances of the same image to scale it out if it is designed as a stateless component. (This is how container technologies such as Docker work.)

Building this type of image is relatively trivial compared to a traditional installation process because you are adding things to a file system. So, automating this build process is straight-forward (for more information about pipelines, see 7.5, “Continuous Integration and Continuous Delivery Pipeline using IBM App Connect V11 architecture” on page 465 and 7.6, “Continuous Adoption for IBM App Connect” on page 485). You can also build images from existing images. For example, you can create a base image with the product run time, a standard set of fix packs, and a configuration that you plan to use for all components. You can then create further images starting from that base image that includes specific applications.

4.2.4 Elastic, agnostic infrastructure and container orchestration platforms

Consider how most traditional servers are deployed as a cluster. How do they handle HA? How are their replicas managed, and how is the load spread across them? How are ongoing deployments and fix packs handled? There is often a product specific *deployment manager* to handle these tasks. For a traditional WebSphere Application Server, it was called exactly that.

Figure 4-4 shows an overview of traditional servers deployed in a cluster.

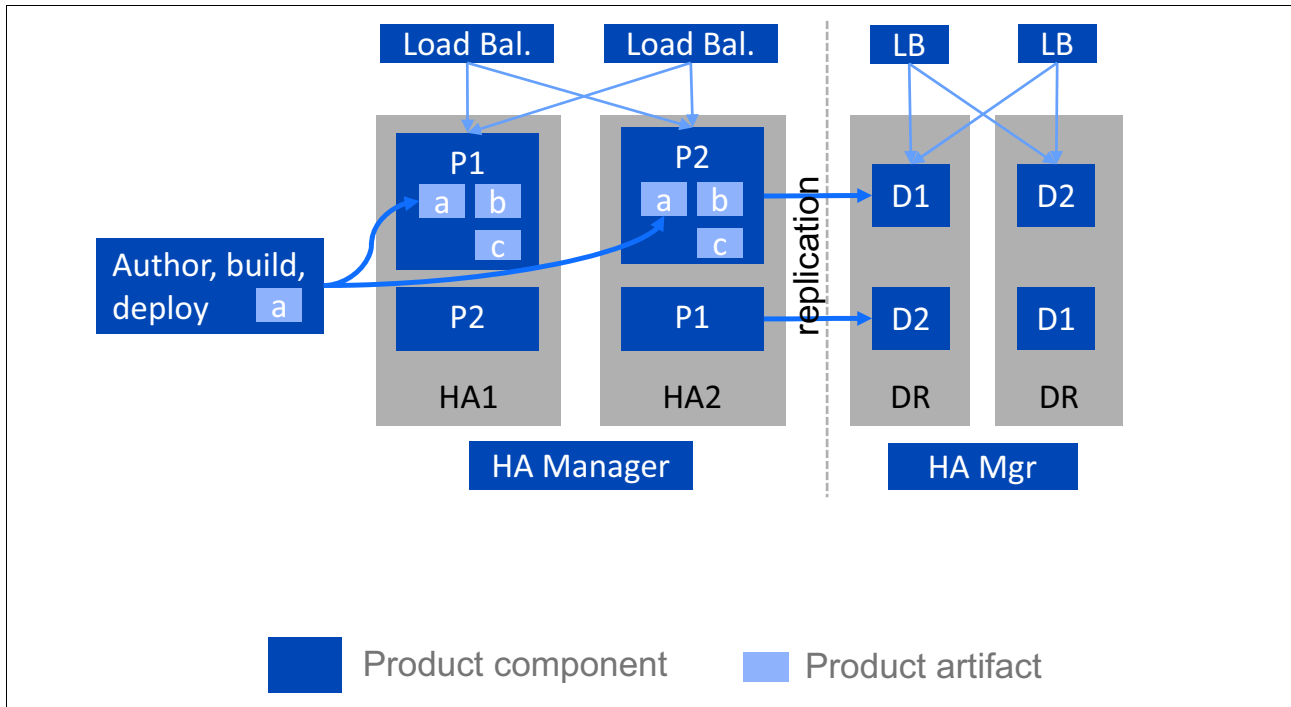


Figure 4-4 Deploying traditional servers as a cluster

As part of your complex installation process, you also had to install the deployment manager and tell it where to find the application servers it was going to administer. Then, for all future configurations and deployments, you used the deployment manager. You also might have to set up a pair of HTTP load balancers to spread the incoming calls across your replicas. In short, you had to know much about WebSphere Application Server to set up and run a production worthy HA topology.

In the cloud-native approach, you no longer need the deployment manager. An orchestration engine such as K8s works with standard containers to manage administration without knowing anything about the internals of a container image. This is the typical programming model for the new lightweight runtime of WebSphere Application Server Liberty when it runs in containers.

Figure 4-5 shows the cloud-native approach to this task.

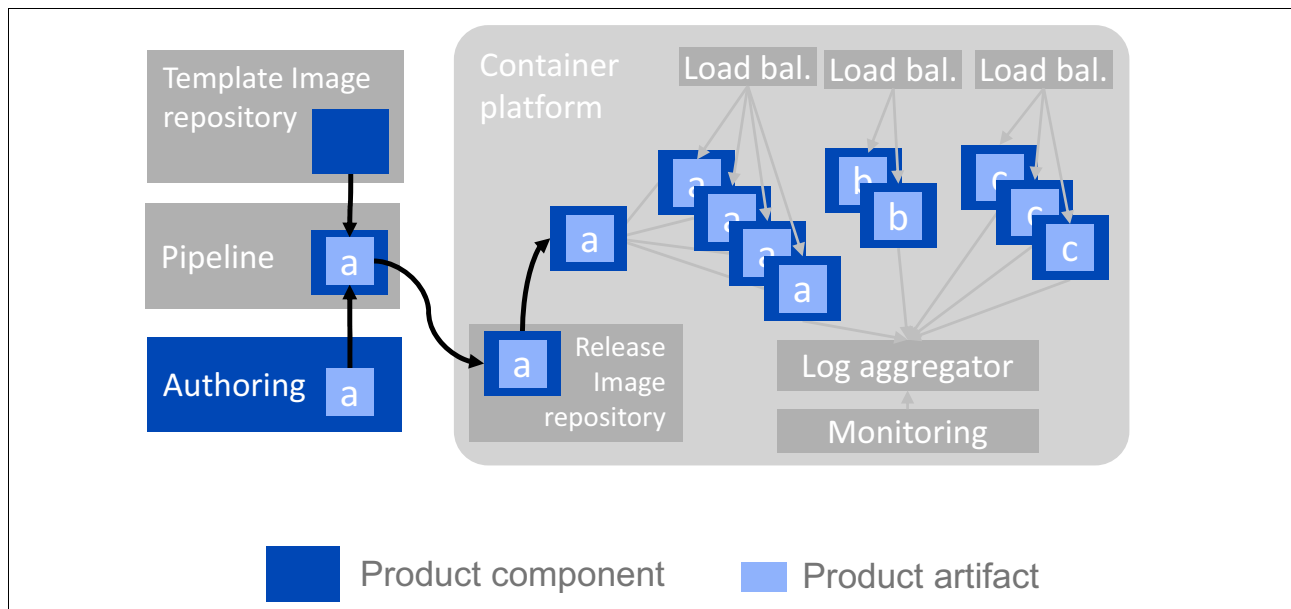


Figure 4-5 New cloud-native approach: No deployment manager

You can instruct K8s to “deploy” the image, and give it some further standards that are based on policy for its non-functional characteristics, for example, a minimum number of replicas for initial HA and performance characteristics, and when to create more replicas and up to what maximum. Furthermore, K8s provides and automatically configures all the necessary load balancing and health checking of containers for you. So, HA is standard feature in K8s. If a container fails, K8s re-instates it.

This is a different way of building and running a topology. The benefit is that it enables a generic platform such as K8s to create instantly a topology that is based on any image that is provided to it, and autonomically and automatically provide HA and scaling for it. If you must re-create *exactly* that setup in another environment, K8s has everything it needs to do so. There is no risk of configuration divergence. All environments are started from the same set of images, configuration, and policy.

However, giving K8s a container image is not enough for it to set up and manage the topology. You need to provide more information about how many replicas to create and where to place them. This information is in a further set of configuration files that are typically in the form of *Helm Charts* and *Operators*. We describe these concepts in more detail later in the chapter.

Here is another process that the cloud-native approach can do: What if you want to change something, for example, you improved the application component design or there is an important new fix pack out with a security patch for the run time. *You cannot make changes to a running server.* You cannot attach to a server and run commands against it. If you do, you lose all your cloud-native benefits because your live container goes out of sync with the image repository and you can no longer easily re-create the environment.

Instead, you should use *image-based deployment*, where any and all changes are done by creating a container image, placing it in the image repository, and then instructing K8s to refresh it in whichever environment it needs to be applied to. K8s starts a new set of replicas based on the new image, and then shuts down the current containers until the new configuration is in place. K8s did not need to know anything proprietary about how to deploy applications to the run time in the container; it just creates a container from the new image.

Adopting container orchestration impacts the ways in which your teams interact with the environment and the solution. The deployment and administration skills they need are container and K8s skills, which largely are consistent across any solution that sits on a container-based architecture. As more solutions are moved to containers, your teams gain efficiencies and consistencies that are impossible to achieve in a traditional infrastructure.

You probably now are starting to see some dependencies across different aspects of the cloud-native infrastructure. For example, container orchestration only works if you embrace image-based deployment, and it is much simpler if your components are stateless. It also is a more nimble platform if the containers use lightweight run times.

4.2.5 Lightweight run times

Clearly, modern run times must work well in a cloud-native deployment. Let's look at the typical characteristics of a modern lightweight run time:

- ▶ **Rapid start:** The run time is a single process with no dependencies, and it can be started and stopped in seconds.
- ▶ **File-based installation and configuration:** The run time can be installed and configured by placing their binary and configuration files on a file system and starting them.
- ▶ **File-based application deployment:** There is no need to deploy application code to a live running server. The application artifacts can be placed on the file system.
- ▶ **Small footprint:** The run time has a smaller footprint and contains only the necessary dependencies, which reduces both its disk and memory footprint.

The responsiveness and efficiency of the container orchestration platform and the agility of the build pipeline for rapid development iterations depends on these characteristics. The better suited the run time, the greater the benefits from a cloud-native infrastructure.

4.2.6 Log-based monitoring

You should not connect directly to individual servers, so monitoring is not done by connecting to a live running server. Instead, the servers report by using logging, which is aggregated by the platform to provide a monitoring view, as shown Figure 4-5 on page 94. Direct monitoring techniques cannot keep up with the constantly changing number of containers. It is not appropriate to expect every container to accept regular monitoring requests.

Note: This description is a slight oversimplification because there are some things that are done by contacting running containers. Key examples include simple health checks, which are used by the container orchestration platform to determine whether the container is functioning correctly and replace it if required. There are also some monitoring techniques such as the scraping that are used by capabilities like Prometheus that may directly connect to individual containers, but there are subtleties to this pattern that are beyond the scope of this chapter.

4.2.7 API-led intra-application communication

A cloud-native infrastructure encourages more modular deployment, but those fine-grained components must communicate among themselves. These calls work across the network. Around the same time as the rise in microservices, consensus was forming around RESTful JSON and HTTP APIs as a standard means for remote synchronous connectivity between distributed components. In cloud-native applications, the general assumption is that components expose their capabilities as APIs so that intercomponent calls are straightforward.

There should be a difference between calls across components within the same *ownership boundary (application boundary)* and to components in another ownership boundary. The former are local simple calls, but the latter are cross-enterprise calls that need a more formal gateway or full API management. For more information about this topic, see 3.1, “Capability perspective: API management” on page 42. We describe the differences between API management and a newer capability known as a *service mesh* in 4.8, “Service mesh” on page 119.

4.2.8 The reprise of event-driven architecture

Although it is not new, an event-driven architecture is an alternative to APIs to pass information between fine-grained components. If you always use APIs to retrieve data from other components, you are dependent at run time on their availability and performance, and indeed all links in the invocation chain between the two components.

Figure 4-6 shows the communication among microservices within an application.

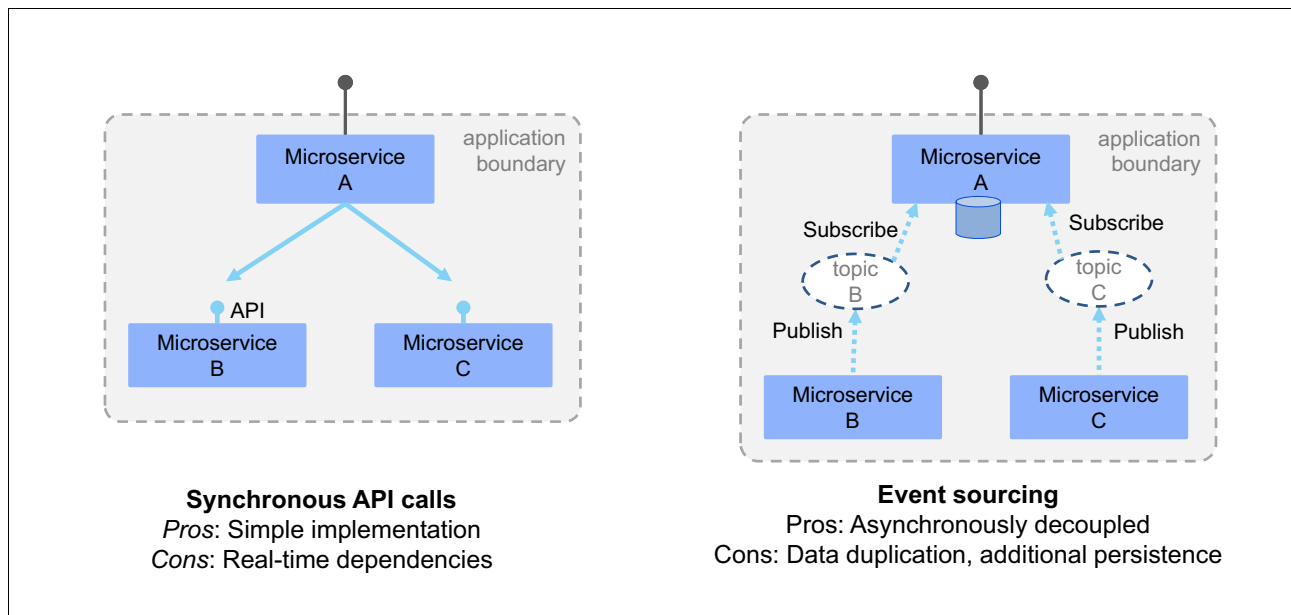


Figure 4-6 Communication among microservices within an application

Events that use a publish/subscribe pattern can more effectively decouple themselves from other components. Components that own data publish events about changes to their datastore (created, updates, and deletes). Other components that need that data listen to the event stream and build their own local datastore so that when they need the data, they have a copy. This process is described as *event sourcing*.

Although event-based patterns improve availability and performance, they do have a downside: They result in copies of the data that must be managed and synchronization with the master data. There are various techniques for resolving this issue, but because of the nature of asynchronous communication, they all result in the data being *eventually consistent* rather than immediately consistent. So, event-based design should be used only where the non-functional requirements dictate it.

4.2.9 Agile methods

To deliver changes more effectively, there must be synergy between agile methodologies and a cloud-native approach. They both enable empowered (decentralized) teams to achieve rapid change cycles that are more closely aligned with the business.

Agile methods are contrasted to older more *waterfall*, methodologies, where an attempt is made to gather all the requirements up front, and then the implementation team works in isolation until they deliver the final product for acceptance. Although this method enables the implementation team to work unhindered by change requests, in today's rapidly changing business environment the final delivery is likely to be out of sync with the current business needs.

Agile methodologies use iterative development cycles and regular engagement with the business to ensure that projects stay focused on the business goals. The aim is to constantly correct the course of the project as measured against real business needs. Work is broken up into business relevant features that can then be prioritized more directly by the business for each release cycle. This benefit to the business is when they accept that there cannot be a precise plan for what will be delivered over the long term but that they can prioritize what is built next.

We encourage you to explore further material about agile methods at your leisure.

4.2.10 Continuous Integration and Continuous Delivery and Deployment

You cannot achieve the level of agility that you want unless you reduce the time that it takes to move new code into production. It does not matter how agile your methods are if the underlying processes are slow. If your feedback cycle is stilted, you cannot react to changes in business needs in real time.

Arguably, the easiest processes to automate are those that are associated with building, testing, and deploying code. *Continuous Integration* (CI) refers to changes that are committed to the source code repository and are instantly and automatically built, quality checked, integrated with dependent code, and tested. CI provides developers with instant feedback on whether their changes are compatible with the current codebase. It also encourages regular commits to ensure that you stay in line with other changes happening to related components.

Many of the things that we described so far are enablers for CI. For example, image-based deployment enables simpler and consistent build pipelines. Furthermore, the creation of more modular, fine-grained, decoupled, and stateless components simplify the automation of testing.

CD has two possible interpretations with subtle but important differences: CD either stands for *Continuous Delivery* or *Continuous Deployment*. Continuous Delivery takes the output from CI and performs all the preparation that is necessary for it to be deployed into the target environment, but it *does not* deploy it because a final approval step is required before code goes into major environments. For many circumstances, Continuous Delivery is as far as full automation can go because the risks that are associated with destabilization of major environments is too high. However, for some environments, *Continuous Deployment* to the target environments is possible, and it is beneficial because of the further reduction in the iteration cycle time.

The dependency ensurance that is provided by container images and the simplicity of their deployment through an orchestration platform is key to providing confidence in automated delivery and deployment. The consistent approach to HA and scalability of the platform provides further comfort that deployments behave as expected.

The extent to which automation is achievable or appropriate across CI/CD processes varies both by organization and project. You must ensure that the technology and the architecture do not block automation. You decide the appropriate level of governance of key environments and whether you can build sufficient trust in automated processes.

4.2.11 Continuous Adoption

CI/CD approaches are well documented and increasingly common, although they are not ubiquitously adopted. There is another “C” for continuous that is an increasing priority for consumers of software: *Continuous Adoption (CA)*.

Cloud-native was initially led by innovations in the public cloud, but many components and tools for cloud-native applications are becoming available for private cloud deployments too.

One of the challenges for enterprises operating a private cloud is to deliver a cloud-native development and runtime platform that keeps up with the accelerated pace of innovation in the underlying software components and tools. Public clouds offer these components “as a service” on recent version levels, and private cloud operators must do the same. Some differences always exist and are necessary, like isolation, governance, and connectivity to traditional IT. However, if private cloud services are lagging in capability, enterprise developers either suffer from a lack-of-innovation penalty or simply bypass their enterprise’s private cloud and expand *shadow IT*.

Vendors and open source communities increased the pace of production of software. At IBM, flagship products like IBM Cloud Paks are on a Continuous Delivery lifecycle, which provides at least quarterly or even monthly upgrades, and this is a pattern that is present across the industry. This pace provides enterprises faster access to technology innovations and essential security fixes. Concurrently, those producers cannot provide support for more variations (versions) of an offering and typically support a limited number of versions behind the most current one. For example, a minor K8s version is released every three months and only the most recent three are supported by the community.

Although there is plenty to inspire the producer community to accelerate, what has not received much attention so far is how consumers deal with this increased pace of upgrades to the software they use. Moreover, in many IT organizations, technology upgrades are considered harmful. They do not seem to bring sufficient business value, and change creates risk. In the face of reducing budgets, upgrades are pushed out until the end of what the provider of the technology supports and beyond. However, the technical debt that is built up must be paid for at some point. The scale of the problem leads many enterprises to be burdened with vast obsolescence management programs.

Meanwhile one kind of technology update is often mandated: security updates. Under the increasing threat of cybercrime, enterprises enforce rapid compliance with the latest security patches, which sometimes leads to a forced upgrade of the entire software package.

CA takes a proactive approach so that software consuming enterprises can stay up to date and out of support trouble. It is now common practice to have code changes triggering builds, integration, testing, and even deployment in a *Continuous Delivery* approach. Enterprises should automate similar CA pipelines that are triggered when vendors or communities release new upgrades. After the CA capability exists, enterprises can still *throttle* the rate of adoption to match their appetite for risk.

Until recently, the risk of change in IT led to stringent change control and a cautious attitude towards upgrades. However, as recent studies² on DevOps show, more lightweight change control with more automation drive better software delivery performance, including lower deployment failure rates. As software is delivered in smaller increments by providers, the perceived risk that is associated with the pain of upgrades reduces significantly.

With CA, enterprises remove the need for significant migrations programs.

Furthermore, the move toward more fine-grained and modular components, which are exemplified in the microservices architecture, further simplifies the scope of components, which reduces the risk when implementing CA of new versions of the underlying dependencies.

In summary, the main reasons for considering CA are:

- ▶ Delivering the latest innovations to developers for optimal productivity
- ▶ Maintaining security compliance
- ▶ Ensuring support availability from providers
- ▶ Avoiding shadow IT
- ▶ Reducing the risk of upgrades
- ▶ Avoiding technical debt

For more information about CA, see [Continuous Adoption - keeping current with accelerating software innovations](#).

4.2.12 DevOps

Intrinsically tied to agile methods and CI/CD is a change in the roles of the people that are involved in the lifecycle of the components.

There used to be separate development and operations roles. Developers were not allowed near the production environment, and operations staff had little exposure to the process of software development. This situation resulted in many challenges, such as the challenges and priorities of each group. Code was not written with the realities of production environments in mind, which resulted in problems that were hard to diagnose and resolve. Operations teams, in an effort to protect their environments, often introduced quality gates that further slowed the path to production, which made it even harder for developers to see how their code reacts in that environment.

DevOps takes the approach that we should constantly strive to reduce and possibly remove the gap between development and operations so that they feel more aligned, and ideally see themselves as a single team with the same objective.

² Forsggren, et al. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*, IT Revolution Press, 2018, ISBN 1942788339

Clearly, the shortening of the path between development and production by using CI/CD is a key part, as is the iterative- and business-focused nature of agile methods. It also means changing the type of work that people do. Software developers should play an active role in looking after production systems rather than just creating new functions. The operations staff should focus on ways to automate monotonous tasks so that they can move on to higher value activities, such as creating more autonomically self-healing environments. This particular role change is often referred to as a Site Reliability Engineer to highlight the fact that they too are software engineers.

In the perfect DevOps world, there would be only a single team consisting of DevOps engineers. Few organizations reach anything close to that point, but many gain significant benefit by keeping a focus on reducing the gap between the teams and introducing system reliability engineer roles.

4.3 Twelve-factor apps

The idea of a *12-factor app* is associated with the concepts of cloud-native, containerization, and microservices architecture. The 12 factors pre-date the mainstream use of container technology and microservices. That context is important because it means that we should see the 12 factors for what they are: A well-written and long-lived set of guidelines for implementing decoupled, scalable, and maintainable application components, but not as a definitive list for containerization, microservices or cloud-native concepts.

With that caveat in mind, in this section we provide:

- ▶ A brief summary of the 12 factors because they overlaps with many of the elements of cloud-native.
- ▶ Bring the 12 factors up to date by relating them to current technologies that are described in this book, such as APIs and containerization.
- ▶ Show how tightly the 12 factors are interrelated to demonstrate how important it is to commit to a significant change in approach if you are to gain the benefits of the 12 factors.

Factor 1 - Codebase: One codebase tracked in revision control with many deployments

A *codebase* is defined as a collection of code that is used to build an application. To encapsulate the application, the codebase should be stored in only one version control repository, such as Git.

Although it is sometimes necessary to share code between applications, each application should have its own codebase. Break the shared code into a separate repository that multiple applications can share. This shared code can then be imported as a versioned library dependency, as described in “**Factor 2 - Dependencies: Explicitly declare and isolate dependencies**”).

The codebase should be the same even when deployed to different environments, as described in “**Factor 3 - Configuration data: Store environment-specific configurations in environment variables**” on page 101).

Factor 2 - Dependencies: Explicitly declare and isolate dependencies

The additional code libraries your application uses are known as *dependencies*. Most languages these days have some form of dependency or package management tool that you use to specify which libraries and versions of libraries that your application relies on. A 12-factor app *never* relies on the implicit existence of system-wide packages, such as curl, so these dependencies also must be declared.

The build process can then isolate them in a self-contained executable, as described in “**Factor 5 - Build, release, run: Strictly separate build and run stages**” on page 102, which in most modern cloud-native applications means code within a container image.

Factor 3 - Configuration data: Store environment-specific configurations in environment variables

Configuration data is everything that varies between deployment environments. It is now known as *environment configuration*. A common example is the credentials that are required for remote resources, as described in “**Factor 4 - Backing services: Treat backing services as remotely attached resources**” on page 101).

How you communicate with your database (for example, over what protocol) always should remain constant. However, which database you use often is different on each system, for example, the host names, the credentials, the log level, and others might be different.

A 12-factor app must not contain any environment configuration; instead, it picks up the configuration from *environment variables*. Environment variables are easy to change between deployments without changing any code. Furthermore, unlike configuration files, there is less chance of them accidentally being checked into the code repository.

Anything that is sensitive should be kept out of the repository. A good litmus test for whether you are following these rules is to ask the following question: Can you make this code repository *open source* without exposing anything sensitive or critical?

K8s provides ConfigMaps and Secrets as places to store environment-specific configurations. These locations can be made available inside the containers as environment variables or files.

Factor 4 - Backing services: Treat backing services as remotely attached resources

A *backing service* is anything the application consumes over the network for normal operation. Examples include datastores (for example, IBM DB2® or MongoDB), and messaging and queueing systems (such as IBM MQ).

A 12-factor application should be able to swap out the local backing services for remote ones with no code changes. There should be no distinction between local and remote backing services.

The location and credentials of backing services should be stored in environment variables, as described in “**Factor 3 - Configuration data: Store environment-specific configurations in environment variables**” on page 101.

In containers, essentially anything that is not in the container can be contacted only remotely because it is difficult (and unwise) to attempt to use in-memory invocations on resources on the local machine. So, this model is implicitly enforced, and all outbound requests from a container look like remote requests even if they are to another container on the same machine.

Factor 5 - Build, release, run: Strictly separate build and run stages

In a 12-factor app, the code is transformed during its deployment process by using three stages that are strictly separated:

- ▶ The build stage converts a code repository into an executable bundle that includes the code and all its dependencies. This bundle is known as a *build*.
- ▶ The release stage takes the build and combines it with the deployments's current environment configuration, which is described in “**Factor 3 - Configuration data: Store environment-specific configurations in environment variables**” on page 101. The resulting release is ready for immediate execution in the execution environment.
- ▶ The run stage starts the app in the execution environment.

The most important separation is between build and run to ensure that it is impossible to make changes to the code at run time and simplify rollback if it is required.

In container technology that uses modern language run times, the build stage can be as simple as placing product and language files on to the file system and creating from them a container image. That image is then the immutable executable that is deployed to each environment.

Factor 6 - Processes: Run the app as one or more stateless processes

This factor is really about the statelessness of the application process. Twelve-factor should not assume that any state is held between invocations. Any data that might be needed in the future must be stored in a stateful backing service such as a database.

There can be many instances where a process running for scaling has a high chance that a future request will be served by a different process. Even when running only one process, a restart (triggered by code deployment, a configuration change, or the execution environment relocating the process to a different physical location) wipes all local (for example, memory and file system) states.

You can see that this statelessness is embedded into the container model. Containers are ephemeral: When they are restarted, a new container is created, and all local memory and files are lost. We see the importance of this point when we talk about scaling in “**Factor 8 - Concurrency: Scale out by using the process model**”.

Factor 7 - Port binding: Export services through port binding

Your application process should expose its function only over a defined URL scheme that is bound to a port, which commonly means HTTP-based APIs that are based on the OpenAPI specification, but other protocols also can be used. Any libraries that are required to perform this exposure should be included as part of the application.

The standardized port-binding approach means that one app can become a resource (see “**Factor 4 - Backing services: Treat backing services as remotely attached resources**” on page 101) for another app by providing the URL to the backing app as an environment variable (see “**Factor 3 - Configuration data: Store environment-specific configurations in environment variables**” on page 101) for the consuming app.

Factor 8 - Concurrency: Scale out by using the process model

Because you build our application as a stateless process (see “**Factor 6 - Processes: Run the app as one or more stateless processes**” on page 102), you can defer its operational management to the surrounding platform. The modern parallel here is with containers running as a single process, and container orchestration platforms such as K8s that provide agnostic operational management of the container replicas.

The platform can manage availability by simply restarting the process (container) whenever there are free resources, and scaling by introducing more replicas. This process reduces the burden on the application code, which can then focus on the application logic.

Factor 9 - Disposability: Maximize robustness with fast startup and graceful shutdown

The 12-factor app processes must be disposable, which means that they can be started or stopped at a moment's notice. This facilitates fast elastic scaling (see "**Factor 8 - Concurrency: Scale out by using the process model**" on page 102), rapid deployment of code or configuration changes, and robustness of production deployments through rapid reinstatement. Processes should strive to minimize startup time, ideally to a few seconds, and ensure that even on rapid shutdown their termination is graceful.

The lightweight nature of containers means that the language, product run time, or an isolated environment can be started rapidly. Container orchestration then makes the most of the lightweight nature of these ideally stateless (see "**Factor 6 - Processes: Run the app as one or more stateless processes**" on page 102) containers to scale and relocate them rapidly as necessary to meet the non-functional needs of the application.

Factor 10 - Dev/Prod parity

Historically, there were substantial gaps between environments, such as the time that is taken for new code to reach production, the different people and roles that are involved in working the different environments, and even the choice of the backing service technology changing from one environment to another.

Twelve-factor apps are designed for Continuous Deployment by keeping the gap between development and production small, which implies good dependency management and isolation (see "**Factor 2 - Dependencies: Explicitly declare and isolate dependencies**" on page 101), automation of build and deployment (see 4.2.10, "Continuous Integration and Continuous Delivery and Deployment" on page 97), overlapping the roles between development and operations (see 4.2.12, "DevOps" on page 99), and choosing backing services that can be used as effectively in development as in production. Containerization helps with this objective because the container image provides a nearly platform-neutral program with all its dependencies included. The same image can be run on a developer's machine as easily as it can in a production environment.

Factor 11 - Logs: Treat logs as event streams

All applications produce logs, which are a stream of time-ordered events that provide information about the running process. In traditional implementations, the application is concerned with where and how those logs are sent, stored, and analyzed.

In a 12-factor app, logs should be passed unbuffered to the standard output. The environment should decide how to aggregate, route, whether and where to store, and how to visualize them.

This approach simplifies the application (see "**Factor 1 - Codebase: One codebase tracked in revision control with many deployments**" on page 100 and "**Factor 2 - Dependencies: Explicitly declare and isolate dependencies**" on page 101) and enables standardization of the approach to logging across the platform. Furthermore, because the application has no knowledge of how its logs are used, no code changes are required if the logging technology changes.

K8s does not provide a standardized logging framework, but managed distributions such as Red Hat OpenShift do, most commonly implementing the Elasticsearch, Logstash, and Kibana (ELK) stack. This stack provides aggregation of logs from multiple types of containers, which enables cross-component visualization and problem diagnosis.

Implicit in this factor is the fact that it is against 12-factor principles to attempt to attach to the running application to assess its status for monitoring purposes, especially in K8s where containers are constantly being added and removed (see “**Factor 9 - Disposability: Maximize robustness with fast startup and graceful shutdown**” on page 103). The only way to know what happened within the application is by processing the historic log stream because the container might no longer be present.

Factor 12 - Admin processes: Run admin/management tasks as one-off processes

Throughout the lifetime of your application, you must run one-off processes to migrate your database, clean up data, or run a console for introspection. These processes should be run in environments as long-running processes to ensure that there are no issues that might occur from a different environment.

These one-off processes run against a release by using the same codebase (see “**Factor 1 - Codebase: One codebase tracked in revision control with many deployments**” on page 100), dependencies (see “**Factor 2 - Dependencies: Explicitly declare and isolate dependencies**” on page 101) and the same configuration (see “**Factor 3 - Configuration data: Store environment-specific configurations in environment variables**” on page 101) as your application.

For example, do not run migrations as a separate codebase working directly against a database. Instead, have them as part of your code base and let them use the same data access path that you normally use in the application.

4.3.1 Conclusion on 12-factor apps

The [12-factor documentation](#) is excellent and should be treated as the source of truth regarding this topic, but it is the starting point for understanding cloud-native applications.

Another useful reference for understanding the 12 factors in a modern context is [Michael Elder’s regrouped version of the 12 factors](#).

4.4 Container technology: the current state of the art

Cloud-native is a theoretical concept. Today, most cloud-native applications are written by using container technology, which was a distant concept not too long ago. Soon, we might be discussing serverless computing.

So, for now, containers are the most important technology in this space. For that reason, in this section we conceptually explain what containers are, and then describe the critical related technology of *container orchestration*, including a brief description of the primary technology in this space: K8s.

4.4.1 Containers

Containers are a modern approach to software virtualization. Virtualization abstracts software from its physical computing environment by putting it into a software wrapper, which makes the software more portable.

Virtualization began with the abstraction of operating systems from the hardware on which they run, as popularized by VMware. The provisioning of infrastructure by using virtual machines (VMs) has become near ubiquitous, and it is now the exception for software to be installed directly on baremetal servers. Containers take this abstraction a step further by enabling a more fundamental abstraction from the underlying infrastructure and providing a much more lightweight and portable virtualization model, as shown in Figure 4-7.

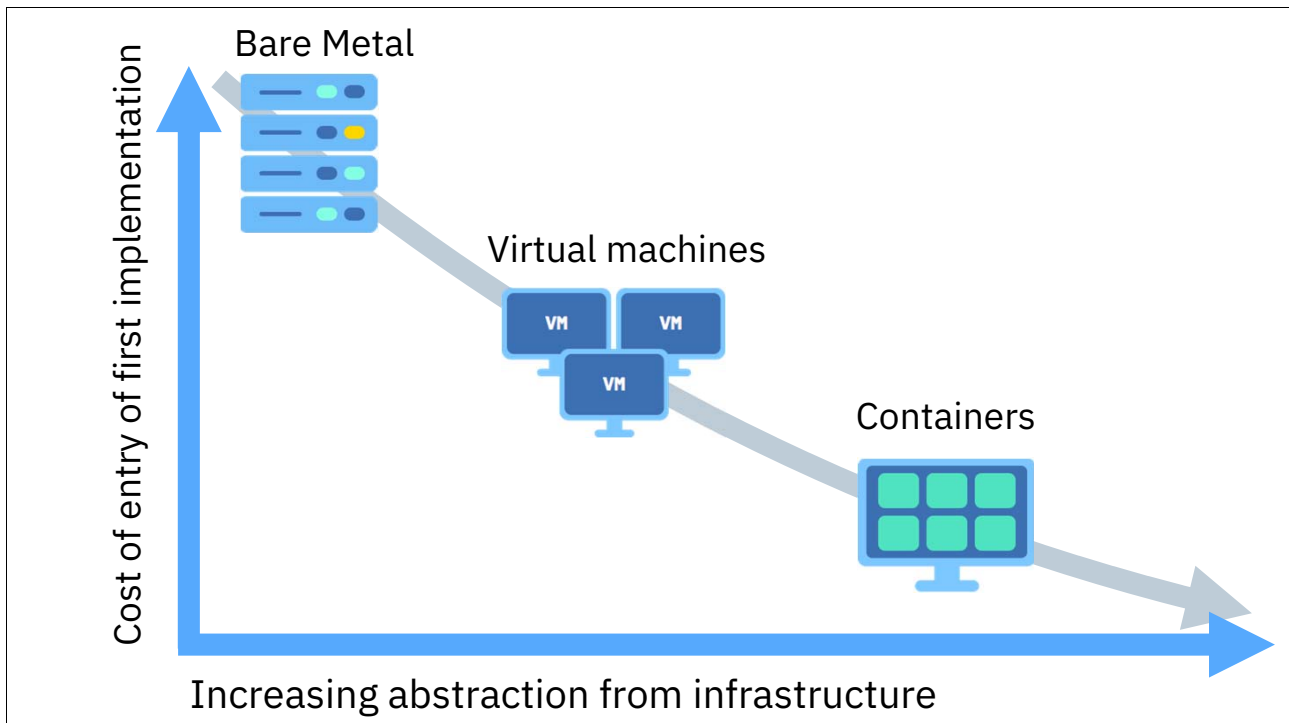


Figure 4-7 Abstraction from infrastructure

Containers further reduce the time, cost, and complexity of setting up a new environment on which to run code.

A *container* is a file that packages together application code along with all the libraries and other dependencies that it needs to run. By packaging together applications, libraries, environment variables, other software binary and configuration files, a container ensures that it has everything that is needed to run the application regardless of the operating environment in which the container runs.

Figure 4-8 shows the difference between VMs and containers.

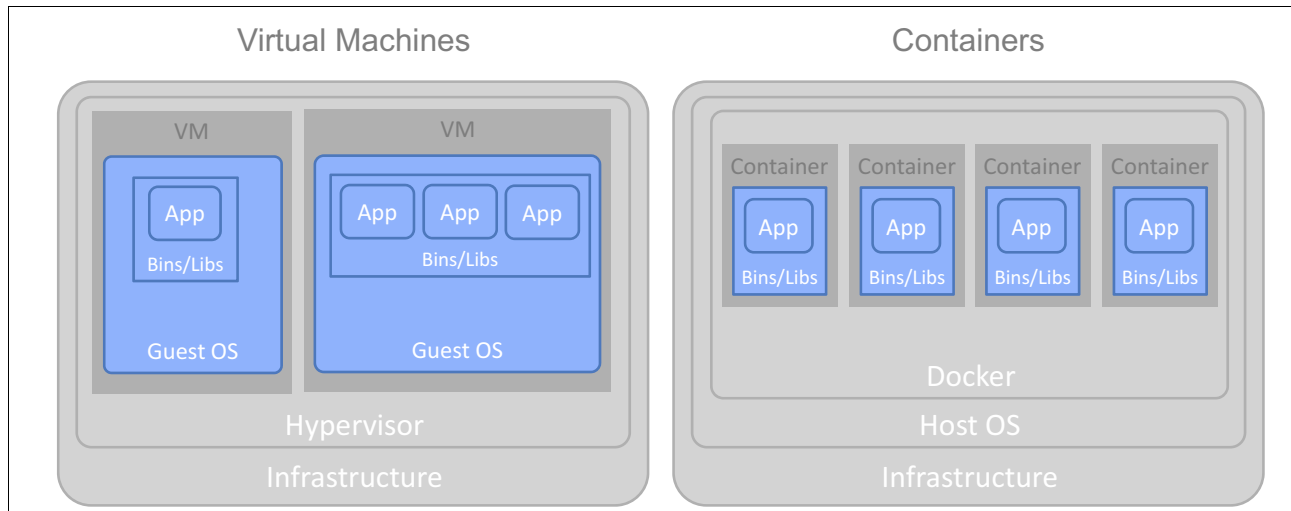


Figure 4-8 Difference between virtual machines and containers³

A key characteristic of a container is that it is small and fast because it uses some of the underlying host operating system's resources to run rather than containing a whole OS of its own. As such, many more containers can be placed on hardware than VMs. Furthermore, their lightweight nature enables a radically different approach to how they are managed and scaled, which align perfectly with the needs of cloud-native applications.

Containerization is the act of readying an application for distribution in a container by packaging its various runtime components together. These components include the relevant configuration files, libraries, and software dependencies. The result is a container image that can then be run on a container platform. For more information about this process, see this [YouTube video](#).

The following sections describe the basic container concepts.

Container image

A container image is the base for every container that you want to run. Container images are built from a *Docker file*, which is a text file that defines how to build the image and what build artifacts to include in it, such as the app, the app's configuration, and its dependencies. Images are normally made from other images, making them quicker to build. Let someone else do the bulk of the work on an image and then tweak it for your use.

Container

A container is a running instance of a container image. You can make multiple containers from the same image. For example, you might do this to create a set of replicas to scale a container horizontally. Every container is created from an image, so a container is a packaged app with all of its dependencies, and the app can be moved between environments and run without changes. Unlike VMs, the running container does not need to virtualize a device, its operating system, and the underlying hardware. Only the app code, run time, system tools, libraries, and settings are packaged inside the container. Containers run as isolated processes on Linux compute hosts and share the host operating system and its hardware resources. This approach makes a container much more lightweight, portable, and efficient than a VM, and many more containers can be run on a hosts' resources than can VMs.

³ Source: <https://www.docker.com/what-container>

Image registry

An image registry is a place to store, retrieve, and share container images. Images that are stored in a registry can either be publicly available (public registry) or accessible by a small group of users (private registry).

4.4.2 Container orchestration

As container adoption increases, so does the need for *container orchestration*. Containers are small and easy to reproduce consistently, and companies tend to use many of them. Applications that are written by using, for example, a microservices architecture are broken down into many separate components that can run in containers. Furthermore, you can run many identical container images alongside each other to scale an application or introduce resiliency.

When the number of containers within an organization grows, this situation requires a new way of managing software. Traditionally, companies manage fewer physical or virtual servers and look after each one with the objective of keeping them running for as long as possible. However, containers can and should be managed more as disposable resources, and they can be created and deleted quickly to manage scaling and failover scenarios.

Container orchestration capabilities make it possible to handle containers at scale. Container orchestration is the process of managing each container throughout its lifecycle and encompasses the following additional functions:

- ▶ Provisioning
- ▶ Redundancy
- ▶ Health monitoring
- ▶ Resource allocation
- ▶ Scaling and load balancing
- ▶ Moving between physical hosts
- ▶ Security isolation

Today's most popular container orchestration technology is K8s, and at the time of writing, it is the one that most cloud vendors are focused on. Other container orchestrators include Apache Mesos and Nomad.

A good overview of why container orchestration is needed is shown in this [YouTube video](#).

4.4.3 Kubernetes primer

[Kubernetes](#) (abbreviated as "K8s," where "8" stands for the 8 characters between "K" and "s.") is an open source system for automating deployment, scaling, and management of containerized applications.

The name Kubernetes originates from Greek, meaning helmsman or pilot. A deeper introduction to what K8s is and is not can be found here:

<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

The K8s platform provides an isolated and secure app platform for managing containers that is portable, extensible, and self-healing in case of failovers.

K8s is based on a variety of distinct "objects". Objects are persistent entities in the K8s system. K8s uses these entities to represent the state. Working with K8s primarily means managing objects. Object can be managed via imperative commands or configuration, or via declarative configuration. More information on advantages and disadvantages of different

object management types can be found here:

<https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/>

K8s objects are divided in basic abstractions, like a pod, or a service, and higher-level abstractions, called Controllers, which are built upon basic objects and provide additional functionality. We describe all of the most important abstractions later in this section.

From an architectural perspective, a K8s platform is deployed as a cluster. A K8s cluster consists of one or more compute hosts that are called *worker nodes*. Worker nodes are managed by a K8s master node that centrally controls and monitors all K8s resources in the cluster. When you deploy workload, the K8s master decides which worker node to deploy the workload resources on, accounting for the deployment requirements of that workload and the available capacity in the cluster.

To understand K8s, we need to have some understanding of its objects and controllers, and their behavior. Some of the main K8s objects and controllers are described in the remainder of this section.

Namespace

Namespaces are a way to divide cluster resources between multiple users. Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each K8s resource can be in only one namespace.

Keep in mind that namespaces do not enforce physical or even virtual segregation or isolation of resources.

Pod

It is useful to recognize that a container is not a K8s object. Containers are always placed within a pod and it is the pod that is deployed, run, and managed by K8s. Pods represent the smallest and simplest deployable units in a K8s cluster, and are used to group containers that must be treated as a single unit. In most cases, each container is deployed in its own pod. However, an app might require a container and other helper containers to be deployed into one pod. So, those containers can be addressed by using the same private IP address. K8s can use different container runtimes to run containers in pods, with CRI-O (the name derives from Container Runtime Interface plus Open Container Initiative) as the default. But Docker and Container are also being supported.

Services and Ingress

A service is a K8s object that provides network connectivity to a group of one or more pods. Thanks to this service, consumers never need to be aware of the actual private IP address of the pods, which of course might be changing regularly. A service can be used to make an app available within your cluster or to outside of the cluster.

When a service provides network connectivity to more than one pod, it also provides basic round-robin load balancing across those pods.

Services can have different types, which determine their exposure mechanism. The two primary types for K8s services are as follows:

- ▶ **ClusterIP:** Exposes the Service on a cluster-internal IP. This service type makes the Service reachable only from within the cluster.
- ▶ **NodePort:** Exposes the Service on each Node's IP at a static port (the NodePort). From outside the cluster, the service can be reached on <NodeIP>:<NodePort>, where the

NodePort can either be assigned randomly by K8s at service creation time, or assigned manually or programmatically.

Services can then be further exposed by using an Ingress. Ingress is not a Service type, but it acts as the entry point for a cluster. It consolidates routing rules into a single resource as it can expose multiple services under the same IP address. An ingress controller is required if you choose to expose services as Ingress.

Labels and Selectors

Labels are key/value pairs that are attached to specify identifying attributes of those objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object. Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in K8s.

Volume and Persistent Volume

On-disk files in a container are ephemeral (anything they store to their “local file system” cannot be guaranteed to survive a restart). This condition presents some problems for non-trivial applications that run in containers:

- ▶ When a container crashes, K8s restarts it, but the files are lost.
- ▶ When you run containers together in a pod, it is often necessary to share files between those containers.

A K8s Volume abstraction solves both problems. A volume is a piece of storage that is managed outside of the K8s clusters, and is accessible to the containers in a pod. How that storage comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.

In this publication, the types of volumes discussed are as follows:

- ▶ NFS (Network File System): An NFS volume allows an existing Network File System share to be mounted into a pod.
- ▶ RBD (Rados Block Device): An RBD volume allows a dynamically generated Ceph block device (<https://docs.ceph.com/docs/master/>) to be mounted into a pod.

For a full list of K8s volume types refer to:

<https://kubernetes.io/docs/concepts/storage/volumes/>

Additionally, volumes have different access modes, often dependent on the volume type:

- ▶ ReadWriteOnce – the volume can be mounted as read-write by a single node
- ▶ ReadOnlyMany – the volume can be mounted read-only by many nodes
- ▶ ReadWriteMany – the volume can be mounted as read/write by many nodes

When the volume becomes a resource available to and manageable by the K8s cluster, it is called a Persistent Volume (PV).

The consistence and persistence of the data on the storage layer is not K8s’ responsibility: it is typically delegated to either the storage technology, or to the containerized applications.

Persistent Volume Claim

A Persistent Volume Claim (PVC) is a request for storage by a consumer, typically a Deployment (described in “Deployment” on page 110) or a StatefulSet (described in

“StatefulSet” on page 110). Conceptually, a Persistent Volume Claim is similar to a pod: pods consume node resources, while PVCs consume PV resources. PVC can request specific PV type, size, and access modes. When a claim request is created, K8s finds all the available Persistent Volumes that match the request, and that have an equal or bigger storage size, and binds them to the request. A bound Persistent Volume cannot be used by other objects until it is released.

If no matching PVs are found, claims remain unbound indefinitely. Claims are bound as matching volumes become available.

ReplicaSet

A ReplicaSet’s purpose is to maintain a stable set of identical pods (a replica) running at any given time. As such, it is often used to guarantee the availability of a specified number of identical pods. While ReplicaSets were originally created to be used independently, today they are mainly used by Deployments as a mechanism to orchestrate pod creation, deletion, and updates.

Deployment

A deployment is a controller that owns other objects (pods) and controllers (ReplicaSets) that are required to run an app, and updates them and their pods via declarative, server-side rolling updates. When a *desired state* is described in a Deployment, the Deployment controller changes the actual state to the desired state at a controlled rate.

Deployments can use Persistent Volume Claims to univocally bind its pods and ReplicaSets to volumes. If a pod in the replica set crashes and is restarted, the original volume is mounted back onto the new pod, which allows the pod to recover its state, if it had persisted that state in the volume.

StatefulSet

A StatefulSet manages the deployment and scaling of a set of pods, and provides guarantees about the ordering and uniqueness of these pods. It is in essence a particular type of Deployment, for applications that require one or more of the following capabilities:

- ▶ Stable, unique network identifiers.
- ▶ Stable, persistent storage.
- ▶ Ordered, graceful deployment and scaling.
- ▶ Ordered, automated rolling updates.

Similarly to a Deployment, a StatefulSet can — and most often does — use Persistent Volume Claims to univocally bind its pods to volumes. In addition, StatefulSets can assign volumes to single pods in the set.

A useful comparison between Deployments and StatefulSets is provided in Table 4-1.

Table 4-1 Comparison between Deployments and StatefulSets Affinity and anti-affinity rules

Topic	StatefulSet	Deployment
Replica Identification	Ordinal support – Pod name consistent across failure	No trivial mechanism – Pod name only exists for lifetime of pod.
Pod Management	<ul style="list-style-type: none"> ▶ Ordered – Completes one by one ▶ Parallel – Action completed across all replicas in parallel 	<ul style="list-style-type: none"> ▶ RollingUpdate – Pods incrementally updated ▶ Recreate - All pods are killed before new ones are created

Topic	StatefulSet	Deployment
Updates & Scaling	<ul style="list-style-type: none"> ▶ Rolling Updates ▶ Scaling ▶ Roll Back – Custom process 	<ul style="list-style-type: none"> ▶ Rolling Updates ▶ Scaling ▶ Rolling Back
Persisted Volumes	<ul style="list-style-type: none"> ▶ Volume per pod – Using volumeClaimTemplates ▶ Volume shared between pods – Using Persistent Volume Claims 	Volume shared between pods – using Persistent Volume Claims
Number of pods running	K8s assures that no more than the maximum number of pods are running, unless it is overridden.	The number of pods can be higher or lower than the number of replicas in specified (normally higher)
Single resilient container	Partial – On node failure the container is not automatically restarted.	Container restarted by K8s after configurable delay.

Many controllers allow the specification of affinity and anti-affinity rules, and constraint of object behavior with other objects with particular labels. For example, you can prevent a pod from being deployed on a particular node. The affinity/anti-affinity feature specifies a wide range of constraints that can be expressed:

Rules can be “soft”/“preference” rather than a hard requirement. So, if the scheduler cannot satisfy it, the pod will still be scheduled constrain against labels on other pods running on the node are allowed, rather than against labels on the node itself, which allows rules about which pods can and cannot be colocated

The list in this section is not intended to be exhaustive. Instead, it provides a reference for the K8s concepts that are used in the practical chapters of this Redbooks publication. A more detailed description of these concepts is provided in the K8s public documentation: <https://kubernetes.io/docs/concepts/#kubernetes-objects>

4.5 Cloud-native is not for everyone, nor for everything

All of the preceding benefits sound valuable. Who wouldn't want them, right? Who wouldn't want to be able to innovate at the speed of the business, optimize the cost of infrastructure, ensure that their applications had the highest possible availability, and not be locked into any particular vendor or technology? It's the CIO's, and indeed the CEO's dream.

However, any good CIO, and more so the CFO, knows that all *improvements* come at a cost. Moving to a cloud-native approach costs in time and money. For sure, it has the potential to provide enormous benefits in the long term enabling the business to remain competitive, and indeed innovate at a pace that brings them to the front of the pack. But it requires investment up front, with returns being seen only somewhere in the middle. The more dramatic gains are seen after the system is well established.

Therefore, it should be clear that there are very few companies that adopt cloud-native wall to wall. This is all the more true for large long-standing enterprises with a large IT landscape. Therefore, we need to clearly recognize *where* we need the benefits of cloud-native, *which* benefits we need, and then plan accordingly, applying the improvements that make the right difference for our particular context.

This likely means adopting these practices in pockets of the enterprise only, especially at first. Applying it to specific applications, initiatives, technology areas and indeed teams. It also means considering what are our priorities, and indeed more granularly, what are the specific

priorities of those pockets of the enterprise – they may not all be the same. As such, we can focus on the right actions to ensure we get the right type of benefits at the earliest point in the initiative.

It is beyond the scope of this book to delve into this in much detail, but in the next section, we summarize and reference some material that might help in this prioritization

4.6 Realizing the true benefits of containerization

It might be tempting to view containerization simply as an infrastructure upgrade. Perhaps likening it to hardware upgrades of the past, or even compared to the move to VMs over the last decade or so. *This would be a mistake.* Such a simplistic view of containers often leads to a “lift-and-shift” approach, which misses the whole premise of what containers bring to the table. At the very least, it results in poor execution of the containerization exercise. At worst, it results in the failure of an entire modernization program.

Before we start sounding too negative, let us recognize that containerization offers wide-reaching benefits across the following areas:

- ▶ *Agility and productivity:* Accelerated development, improved consistency across environments, and empowered autonomous teams improve productivity and quality.
- ▶ *Fine-grained resilience:* Independent deployment of HA components remove single points of failure.
- ▶ *Scalability and infrastructure optimization:* Fine-grained dynamic scaling and maximized component and resource density make the best use of infrastructure resources.
- ▶ *Operational consistency:* Homogeneous administration of heterogeneous components reduces the range of skillsets that are required to operate the environments.
- ▶ *Component Portability:* Portability across nodes, environments, and clouds ensures choice when you select platforms.

Figure 4-9 shows the benefits of containerization.

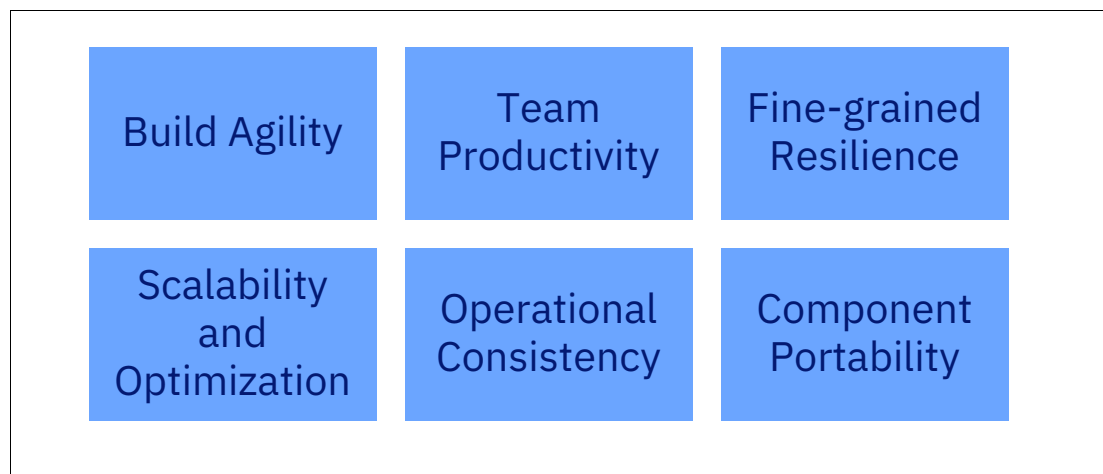


Figure 4-9 Benefits of containerization

You should not be surprised to see similarities in the first few of these benefits when you compare them to the benefits of microservices architecture described earlier in the chapter. The two initiatives are related, though not intrinsically linked.

You certainly can achieve the benefits that are mentioned earlier in this document. But you must recognize that a move to containers is an opportunity for broader modernization. This means making additional changes such as those listed here.

- ▶ Fine-grained components
- ▶ Container orchestration
- ▶ Disposable components
- ▶ Pipeline automation
- ▶ Image-based deployment
- ▶ Infrastructure as code
- ▶ Organizational decentralization
- ▶ Agile development methodology
- ▶ Self-service developer experience

Many of these are similar, or even the same as the themes we spoke about in our description of cloud-native. So again, there is unsurprisingly a significant overlap between cloud-native and containers. For this reason, we do not expand on them here, but instead refer you to the following article for more details:

<https://developer.ibm.com/series/benefits-of-containers>

The most important take away here is that you can assess which of the containerization benefits are most important to you. Then, you can prioritize which steps to focus on first, ensuring a greater likelihood of early gains, and indeed success overall.

4.7 Application boundaries in a container-based world

In traditional software design approaches, people often refer to *IT applications*. An IT application can be loosely defined as a set of software-delivered capabilities that have a specific application in a user context.

As an example, a word processor helps an author to write a book. In this example:

- ▶ The *user* is the author.
- ▶ The *user context* is writing a book.
- ▶ The *IT application* is the word editor.

From here onwards, we refer to *IT applications* simply as *applications*.

4.7.1 Implicit and explicit boundaries

Historically, from an implementation perspective, the functions of an application were contained in one large silo of code. That is often called a monolithic application.

Monolithic applications were deconstructed slightly into tiers but this was more to separate different types of code (for example separate presentation code, business logic, and data access). Still, each tier contained code that relates to functions that span right across the application.

Within those tiers, a well-designed application is broken down internally into separate components that are responsible for each of the application's functions. However, they still all run in the same application server runtime.

Software development and architecture approaches like 12-factor and microservices started to advocate for those components to become truly independent. Ideally, they would be completely decoupled from each other down to the network level. This is shown in Figure 4-10.

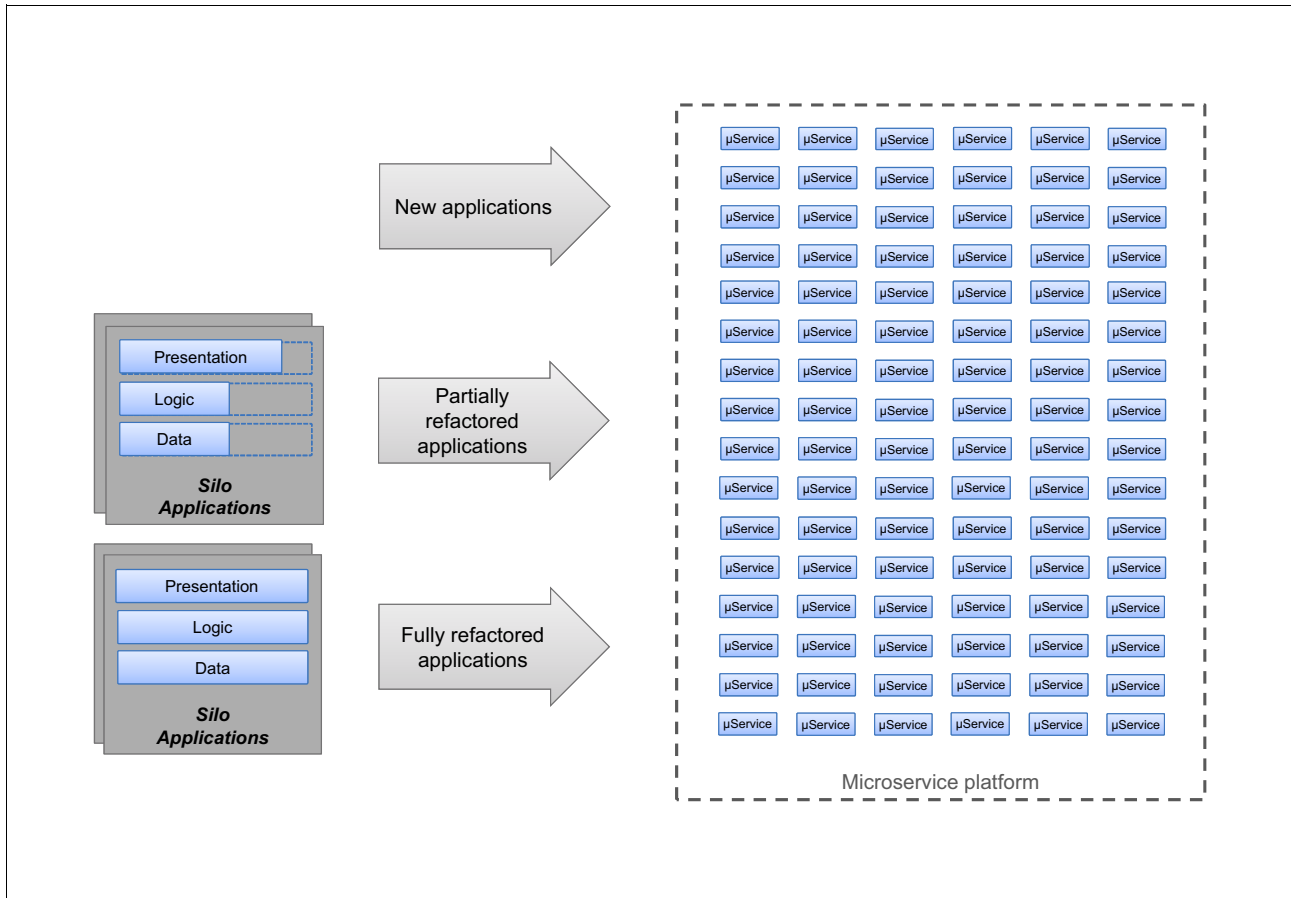


Figure 4-10 Separate components that run in the same application server runtime

With these changes in application architectures, an issue started to surface: how to define an application boundary.

For monolithic applications the logical boundary of the application corresponded with the physical implementation of the application itself. After the application components become truly decoupled and independently deployed, the boundary is no longer implicitly present. Therefore, if a boundary must exist to group all the application components in a single logical construct, that boundary must be explicitly identified and designed in.

4.7.2 Why do application boundaries matter?

It might be tempting to think that we could dispense with application boundaries altogether and simply manage business functions granularly. However, anyone with experience of even a moderately sized IT landscape recognizes that clear ownership is required at a much more

coarse-grained level than individual business functions. This applies at all stages of the lifecycle of an application – analysis, design, implementation, and operational running.

Application boundaries are important, especially in the context of integration. Figure 4-11 shows the difference between interactions within an application and those across an application boundary.

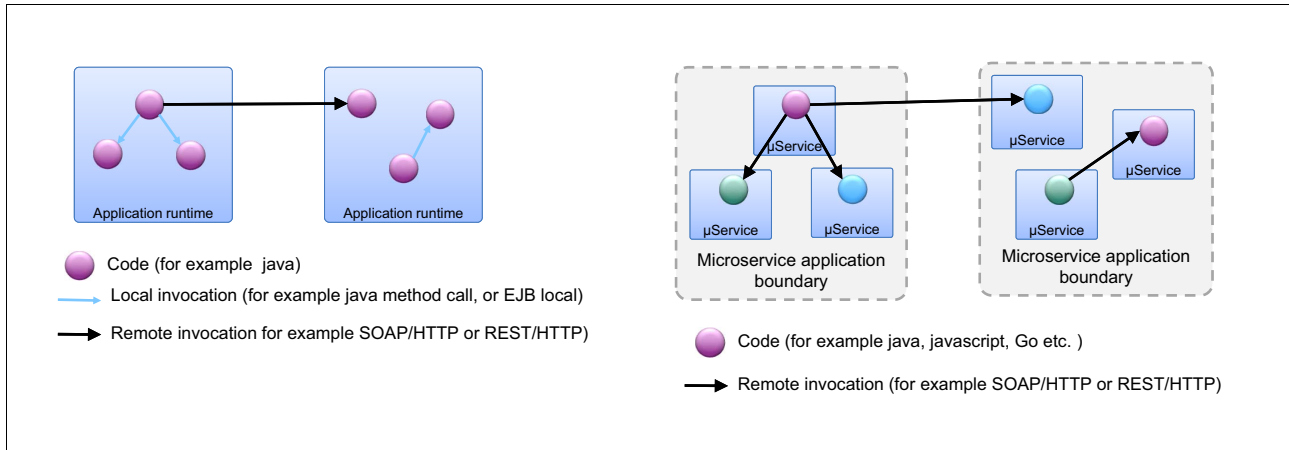


Figure 4-11 Difference between interactions within an application, and those across an application boundary

We discuss later that boundaries are primarily about ownership. Within a boundary, a relatively close-knit group of people will be involved in implementing and/or administering the components. They already know of one another's interfaces where there is probably a level of trust between them that means they can offer interfaces without going through a complex security model. In many cases, there might be relatively few components that use a particular interface. So if it needs to be changed, the impact is minimal.

Interactions across application boundaries are very different. We would need to be able to discover that the interfaces exist in the first place. Since it is owned by a separate group, that group will want to have control and insight over who is using it. Since it is likely that it has multiple consumers, changes will need to be performed in a more measured way with formal deprecation and versioning policies.

So, it quickly becomes clear that we need to decide what the boundaries are. That way, we know when we are crossing them, and we can institute appropriate guidance and control that respond to the differences across those boundaries.

4.7.3 How should we choose the application boundaries?

In a cloud-native world, it is easy to see that there are various ways to define application boundaries.

A possible approach to define an application boundary would be to look at its domain. A software development approach called Domain Driven Design (DDD) advocates developing applications and projects based on their domain: a sphere of knowledge, influence, or activity. However, multiple applications might pertain to the same domain. So, a domain might be too coarse-grained to be used as a boundary for a single application.

https://en.wikipedia.org/wiki/Domain-driven_design.

Note: At this point, it is useful to borrow from Martin Fowler's thinking on this topic:

Applications are social constructions:

- ▶ A body of code that's seen by developers as a single unit
- ▶ A group of functionalities that business customers see as a single unit
- ▶ An initiative that those with the money see as a single budget

All of these are social things. We can draw application boundaries in hundred arbitrarily different ways. But it's our nature to group things together and organize groups of people around these groups. There's little science in how this works, and in many ways these boundaries are drawn primarily by human inter-relationships and politics rather than technical and functional considerations. To think about this more clearly I think we have to recognize this uncomfortable fact.^a

a. Excerpt from <https://martinfowler.com/bliki/ApplicationBoundary.html>

Following Martin Fowler's idea, we assume that people's view of an application is not directly related to its architectural design, nor its technology implementation. Instead, they view the effect that the application has on people that interact with it: its developers, its users, and its sponsor.

In essence, an application is defined by its *owners*.

Therefore, defining the *ownership boundary* for an application would help to identify the parties, architectural components, and technology implementations that pertain to that application. See Figure 4-12 on page 116.

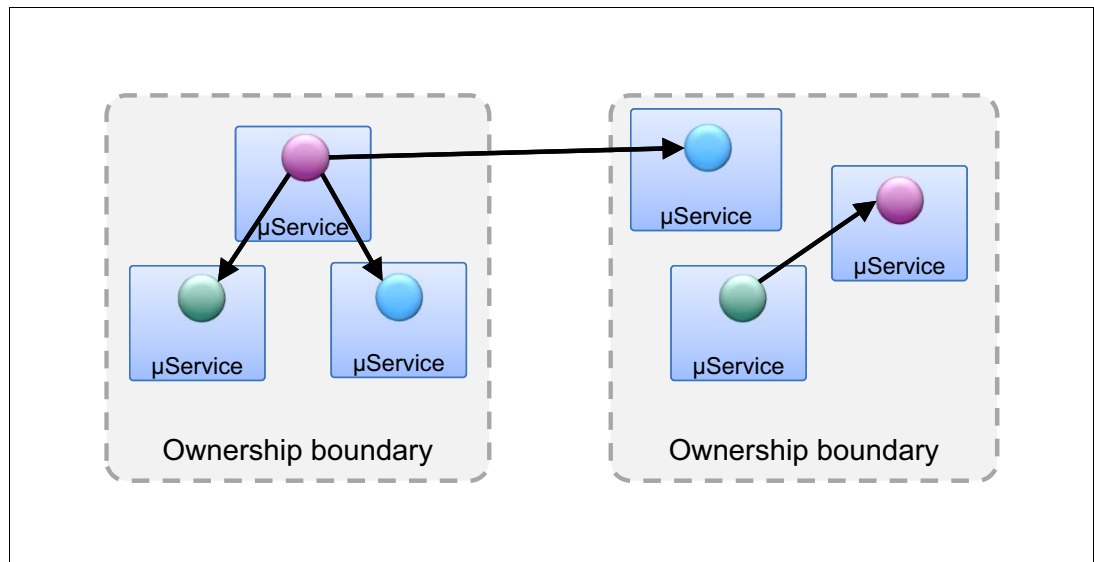


Figure 4-12 Ownership boundary for an application

Therefore, an application boundary can essentially be defined by the ownership boundary for that application.

It is also worth noting that, within an enterprise, multiple application ownership boundaries could be part of a single domain. See Figure 4-13.

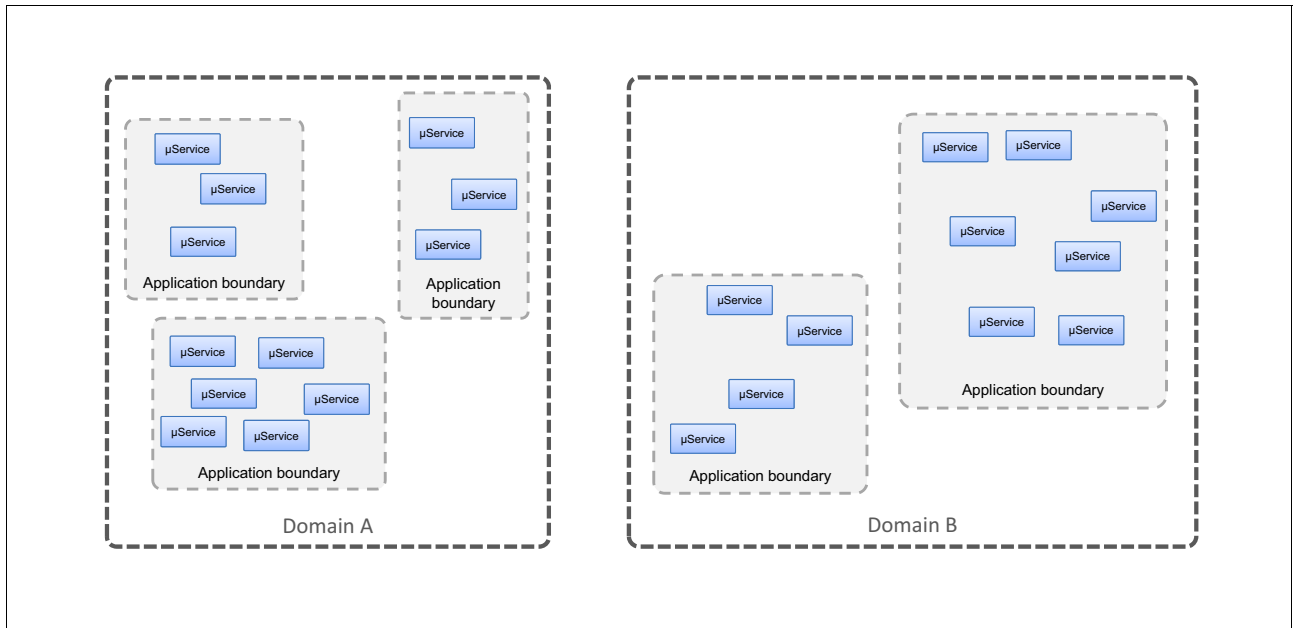


Figure 4-13 Multiple application ownership boundaries part of a single domain

The agreed-upon definition of an application boundary also allows to take a close look at the APIs that are exposed by each application component, for simplicity identified as a microservice. Most microservices components make their capabilities available via an interface such as RESTful HTTP/JSON based APIs. Just as the number of microservices components on the network increases, so does the number of exposed APIs on those components. See Figure 4-14 on page 117.

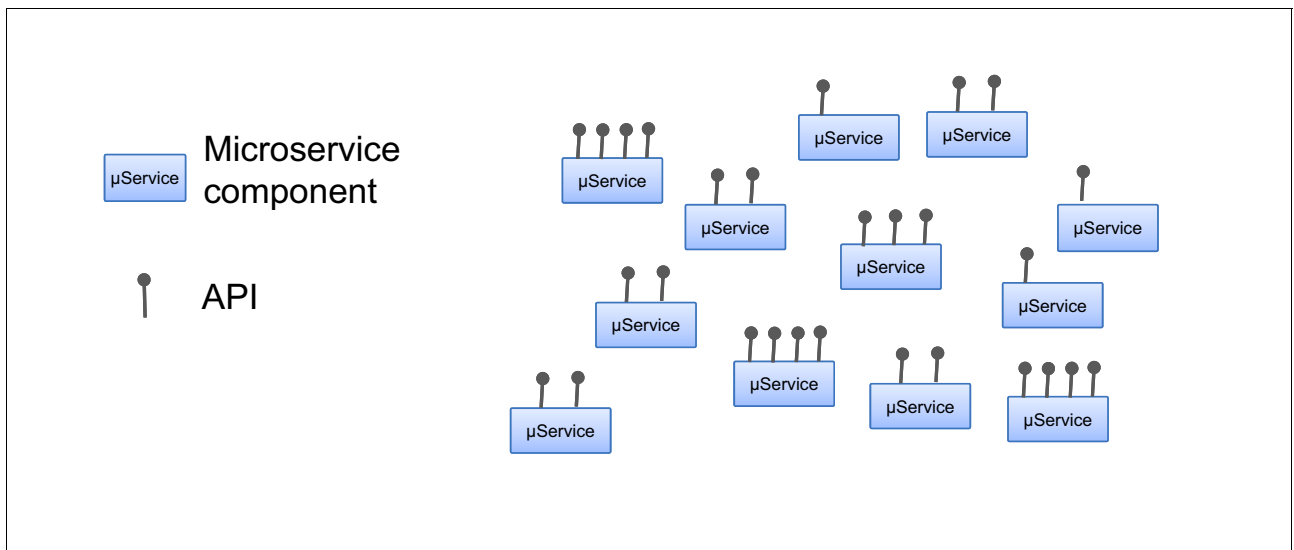


Figure 4-14 Microservices capabilities made available via APIs - 1

How can consumers find the APIs they want from the overwhelming set available? Which are APIs that should be re-used in other contexts?

Ideally microservices components are completely independent. But, in reality, some are really used only in the narrow context of an application. And some others are intended for much

wider reuse across the enterprise, and perhaps beyond. However, technically they all look the same. See Figure 4-15 on page 118.

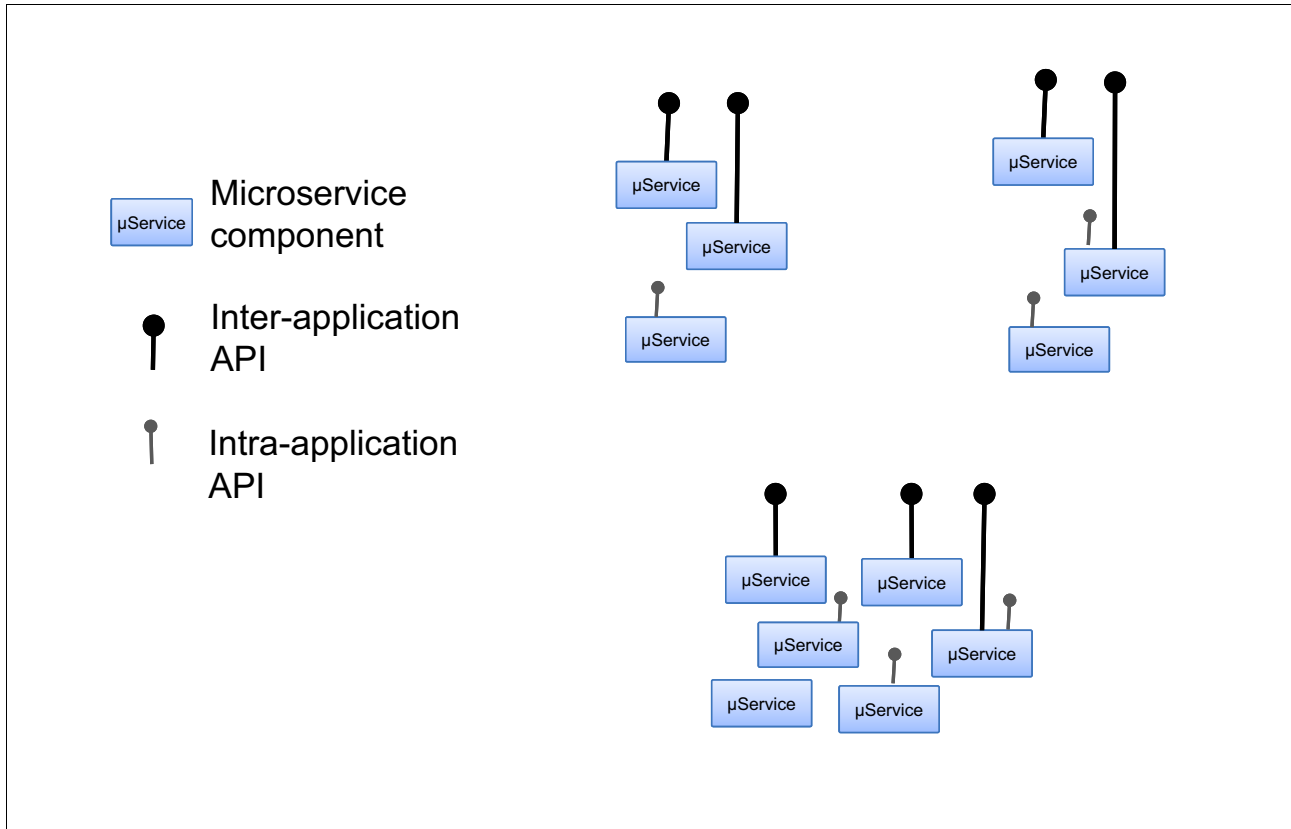


Figure 4-15 *Microservices capabilities made available via APIs - 2*

Intra-application communication (within an application) is logically different from inter-application communication (across different applications). Although they may both be performed by using web APIs, their scope is radically different, as can be their implementation

It becomes clear that a good way to manage such a large number of components is to recur to the notion of the application boundary that was introduced earlier. Components within the boundary should be able to talk to one another's APIs at will, and then make some APIs available only beyond the boundary

Without the notion of an application boundary, there is no enforceable distinction between an intra-application API, and an inter-application API. And more importantly, there is no indication of ownership and accountability. See Figure 4-16 on page 119.

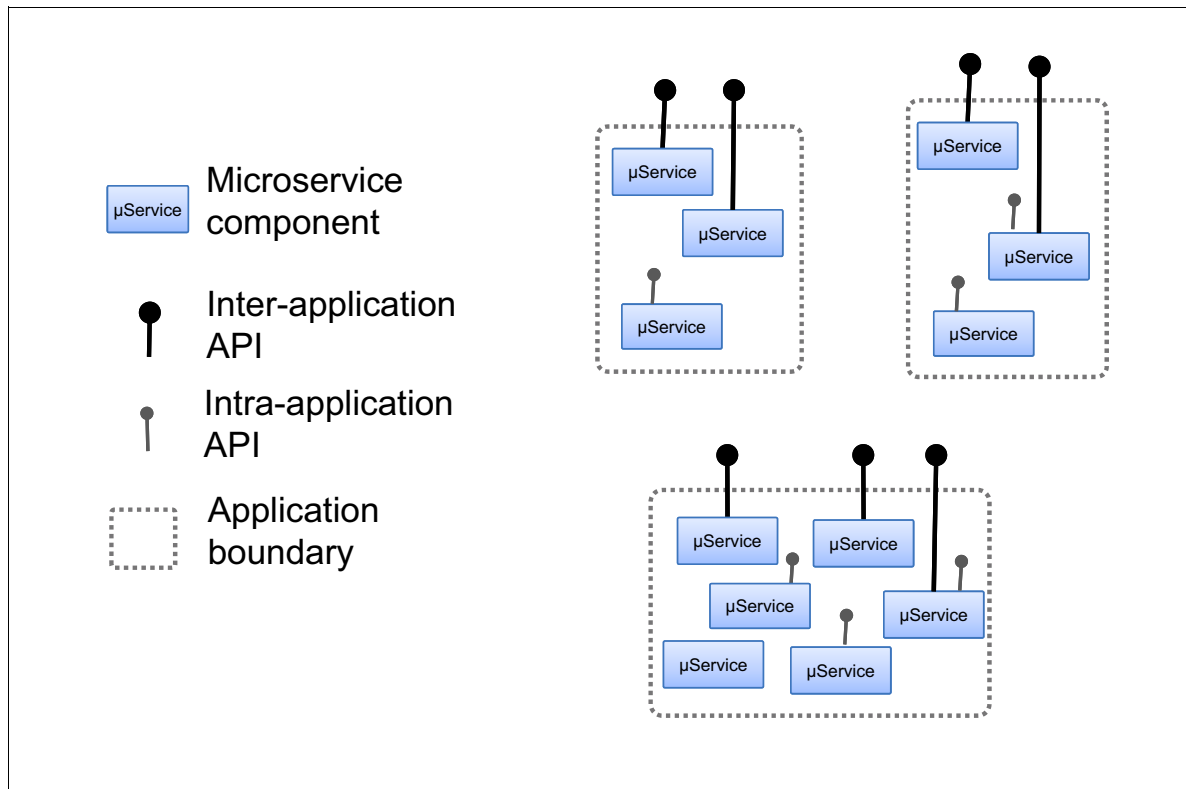


Figure 4-16 Microservices capabilities made available via APIs - 3

Therefore, application boundaries allow you to group components with an owner at the group level in addition to the owners at the component level. That way, you ensure a consistent design of the overall application. And as a result, you also can differentiate between these issues:

- ▶ Communication within the boundary, which should be “light touch”
- ▶ APIs exposed beyond the boundary, which are destined for broader reuse by consumers outside the ownership of the boundary, or even its domain.

Later in this chapter we look at technology-based solutions that help define and enforce application boundaries. Furthermore, we would expect to see 3.1, “Capability perspective: API management” on page 42 used to expose and manage APIs beyond across these application boundaries, though perhaps not within them. We expand on this when we discuss the purpose of the service mesh in relation to API management in the next section.

4.8 Service mesh

In traditional applications, communication patterns are usually built into application code and service endpoint configuration is usually statically defined per environment.

As mentioned in the previous section, as application componentization grows and applications become more cloud-native, so does the number of components on the network. These components, often referred to by the much overloaded term *services*, typically expose APIs to be consumable by other services.

Therefore, there is a requirement for an additional infrastructure layer that helps managing communication within complex applications that consist of a large number of distinct services. Such a layer is usually called a service mesh.

It is important to note that a service mesh is not an overlay network. A service mesh simplifies and enhances how service communicate over the network that is provided by the underlying platform.

4.8.1 Role of a service mesh

The aim of a service mesh is to abstract the complexity of inter-service routing away from applications and their components, and to manage it at cloud-native infrastructure level. As such, a service mesh is a cloud-native infrastructure capability that handles communication between services.

At a high level, a service mesh is responsible for:

- ▶ Providing efficient communication between services,
- ▶ Abstracting the mechanism for reliable request/response delivery from the application code.
- ▶ Handling network failures by using, for example, configurable re-try patterns.
- ▶ Providing visibility and control of the service-to-service communication.
- ▶ Simplifying secure communication between services.

More specifically, typical functional requirements for a service mesh are:

- ▶ Service discovery
- ▶ Service registry
- ▶ Traffic management
- ▶ Traffic encryption
- ▶ Observability and traceability
- ▶ Authentication and authorization
- ▶ Failure recovery

Often a service mesh also enables more complex operational capabilities, like A/B testing and canary rollouts (both described in <https://martinfowler.com/bliki/CanaryRelease.html>), rate limiting, access control, and end-to-end encryption. From a non-functional point of view, all these capabilities are available to applications and their components without affecting the application code. As a result, developers can leverage them without having to instrument their code.

Finally, a typical building block of cloud-native infrastructure is a container orchestration platform, such as K8s. For this reason, it is expected from a service mesh to be able to interact natively with K8s controllers and resources, and to enhance their functionality, when it comes to service-to-service communication.

Service registry

Modern, cloud-native applications are often highly distributed and use a large number of components or microservices that are loosely coupled. In a dynamic, cloud environments placement of any service instance can change at any time. So, there is a need for a repository that holds current information about which services are available and how they can

be reached. This repository is called *service registry*, and it is used for the service discovery that we mention shortly.

K8s provides implicit service registry by using DNS. When a controller alters K8s resources — for example starting or stopping some pods — it also updates the related service entries.

A service mesh can use K8s' DNS, or implement a separate service registry.

Service discovery

Container orchestration platforms are constantly starting and stopping components for resilience and scaling purposes. As a result, it becomes all the more challenging to be able to find instances of the service at runtime. The dynamic routing of the service mesh makes this even more complex, so the mesh needs highly effective mechanisms for runtime service discovery.

Service meshes offer two types of service discovery:

- ▶ **Client-side discovery:** A consumer that requests a service network location gets all service instances from the service registry, and it decides which one to contact.
- ▶ **Server-side discovery:** A consumer sends a request to a proxy, which routes the request to one of the available service instances.

A service mesh is expected to provide at least one of these two types of service discovery.

It is worth noting that this usage of the term “service discovery” is not universal, but specific to service mesh technologies. In an API management context, service discovery is focused on how implementers find what services are available at design time. So, for example, implementers find services in a developer portal, rather than at run time as defined here for a service mesh.

Traffic management

When there are multiple instances of a target service available, the incoming traffic should be load balanced between them. K8s natively implements basic round-robin load-balancing functionality, for inbound traffic (known as *ingress*): connecting from outside the mesh to services within the mesh.

A service mesh can greatly enhance the traffic management options available in K8s, providing more sophisticated and fine-grained traffic management options:

- ▶ Reverse proxy capabilities
- ▶ Percentage-based and context-based routing: canary release, A-B testing (<https://martinfowler.com/bliki/CanaryRelease.html>)
- ▶ Advanced failure handling: see the section “Failure recovery” on page 122.

In a service mesh, traffic management can be controlled and scoped at different levels:

- ▶ Ingress: connecting from outside the mesh to services within the mesh
- ▶ Intra-mesh: services within the mesh invoking other services within the mesh
- ▶ Egress: services within the mesh invoking services outside of the mesh.

Traffic encryption

Securing traffic between components is an onerous task if performed by the application developer, involving deep knowledge of security mechanisms and introducing issues such as certificate management. We need the platform to take these challenges of the developers hands.

In K8s internal data traffic can either be all plain or all encrypted using IPSec. This is a good start, but is not granular or flexible enough for many solutions.

A service mesh allows dynamic encryption of traffic to and from specific services, based on policies, and it does not require any changes to the application code. Adopted encryption mechanisms are typically using mutual TLS, with the service mesh being responsible for key and certificate provisioning and rotation.

Observability and traceability

In a similar way to which application logs are traced for debugging purposes, there is benefit in tracing traffic between services. K8s does not provide tracing capabilities for traffic between services.

A service mesh provides this capability, allowing all traffic to be traced and visualized, without any modification to the application code. Typically, a service mesh can interface with backends via a number of telemetry adapters, including Prometheus (Metrics), Jaeger (Tracing), and Kiali (Mesh visualization).

Access control

K8s, by default, defines network policies that govern which pods can communicate with one another, but implementation of network policies is done at the Container Network Interface (CNI) network level.

A service mesh enhances access control up to Layer 7 of the networking stack, providing both authentication and authorization at service level.

Failure recovery

When it comes to service connectivity, by default K8s does not provide failure recovery capabilities.

A service mesh provides several failure recovery features and patterns, such as:

- ▶ Retries – reattempting requests that fail based on a retry policy.
- ▶ Timeouts – Releasing calling thread resources after a specified period.
- ▶ Circuit breaker pattern – Stopping requests for a cool off period when a downstream system seems to be in trouble (<https://martinfowler.com/bliki/CircuitBreaker.html>).
- ▶ Fault injection – Enabling faults to be deliberately introduced into invocation chains during testing in order to explore their repercussions.

4.8.2 Service meshes and API management

As previously discussed in this section, a service mesh is an infrastructure layer that helps managing communication within complex applications that consist of a large number of distinct services.

This service-to-service communication is typically communication between microservices.

Thus far, we have made no differentiation between these microservices. However, microservices are typically grouped into something similar to the application boundaries that would have been there, were they written as a traditional siloed application. This boundary is of great importance, because it is only within this boundary that we would expect to see a service mesh in use. Let's understand why.

In 4.7, “Application boundaries in a container-based world” on page 113, we gave a definition of an application as a set architectural components and technology implementations within a domain that is part of the same ownership boundary. Consequentially, we used the ownership boundary to define the application boundary.

The application boundary was then used to differentiate between intra-application communication (which goes between microservices within one application boundary) and inter-application communication (which goes across different application boundaries). See Figure 4-17.

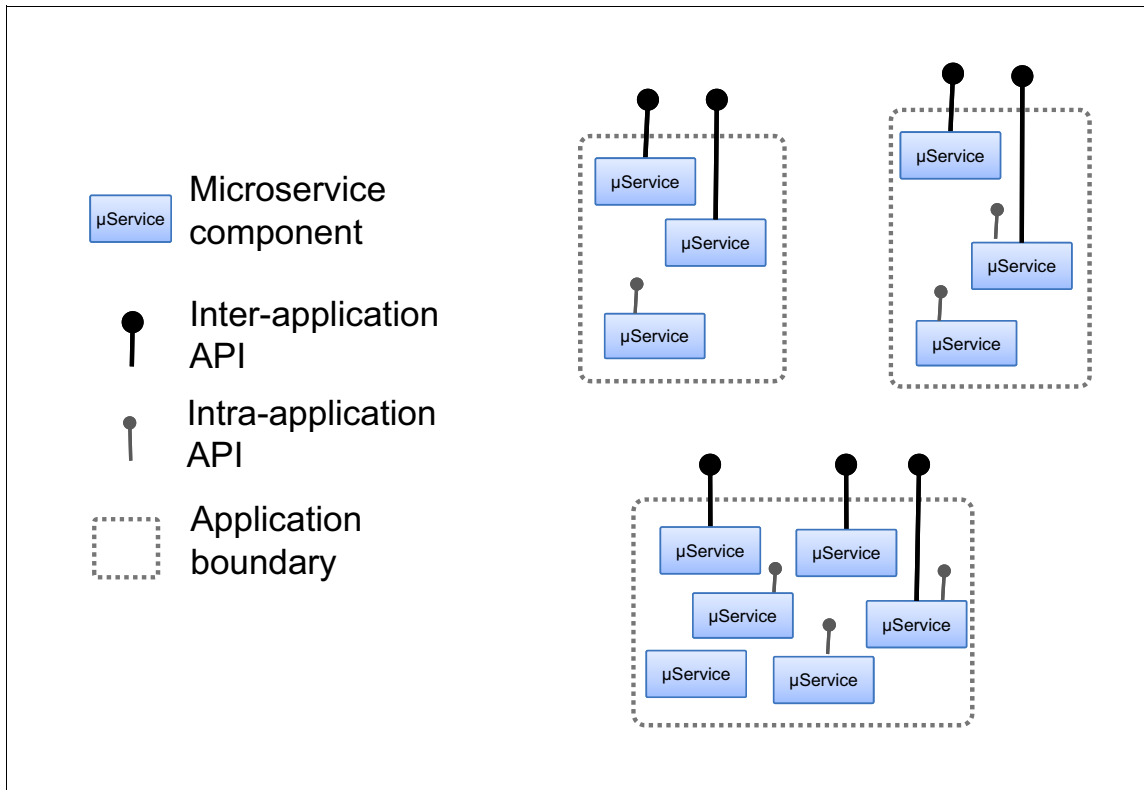


Figure 4-17 Application boundaries

In this section, we discuss how the difference between these two types of communication reflects into technology implementations.

A first step to explicitly expose specific APIs beyond an application boundary would be to use a formal exposure mechanism: for example, a network-accessible K8s service - via ingress or node port.

But how do the owners of the microservices make definitions of the APIs that they want to expose easily discoverable? How do other consumers explore what APIs are available? How will access to the APIs be administered?

A more formal exposure mechanism, like an API gateway, could fulfill those additional requirements. See Figure 4-18 on page 124.

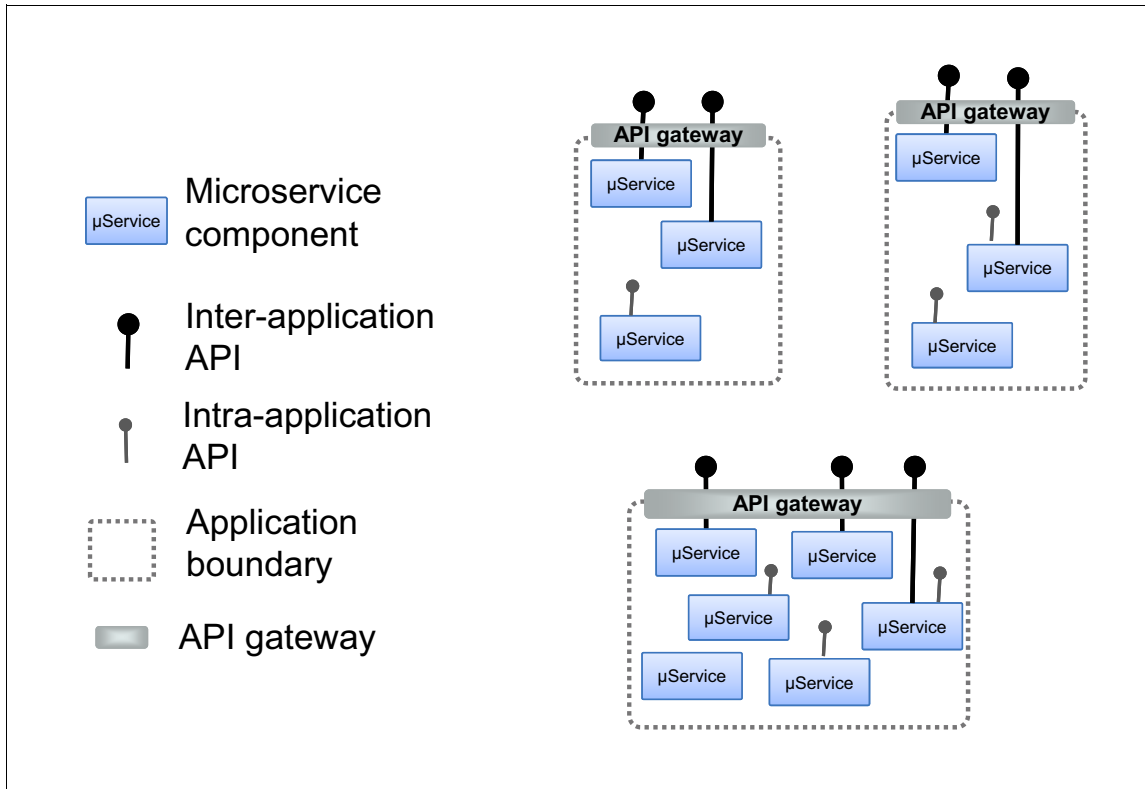


Figure 4-18 Application boundaries with API gateway

The usage of an API gateway enforces a physical implementation of what so far was only a logical boundary, based on the social and business notion of the application itself: it mandates that communication across applications goes through a formal layer, which is the sole gateway to what's wrapped within the ownership boundary. As a direct consequence, other microservices that are owned by a different group — and therefore outside of the boundary — should be able to access application components only via this gateway.

What about the advantages that are provided by a service mesh when it comes to service-to-service communication?

They can all be leveraged — if the mesh is used for communication within the application boundary — for what we have called intra-application communication. See Figure 4-19 on page 125.

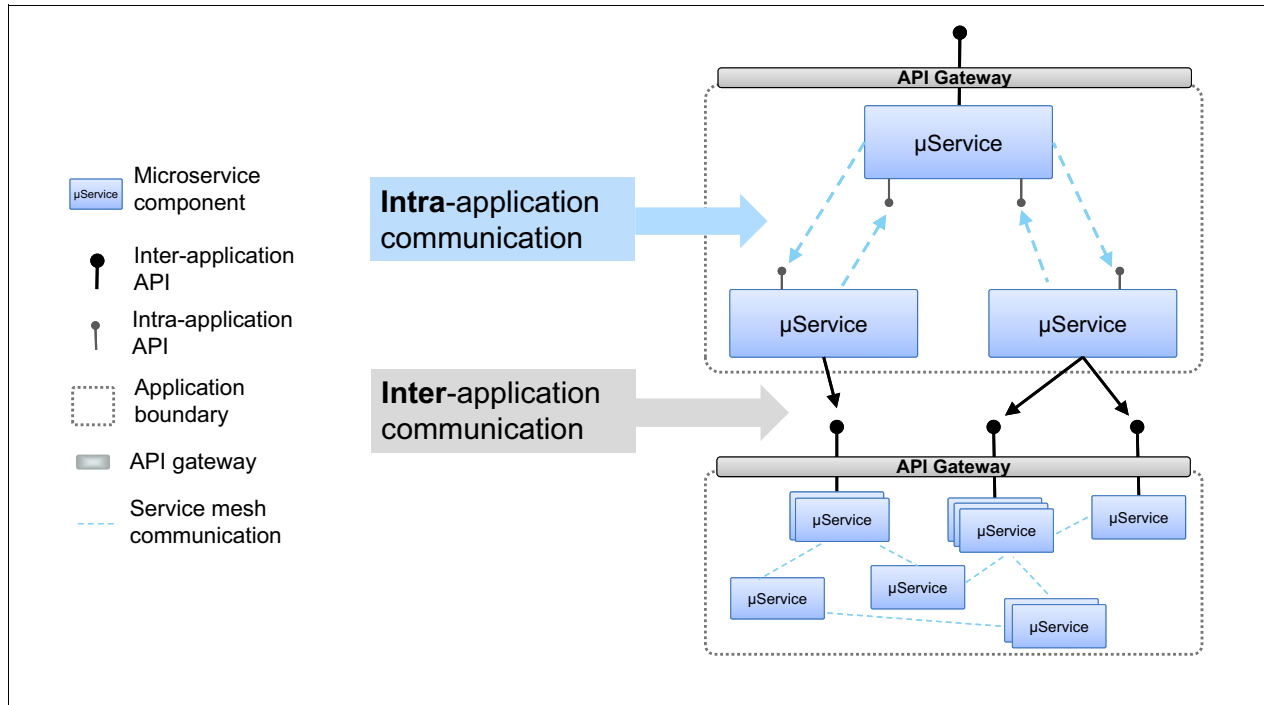


Figure 4-19 Intra-application versus inter-application communication

With this approach, when it comes to inter-application communication within each application boundary we can still benefit from the key capabilities of a service mesh:

- ▶ Service discovery
- ▶ Traffic management
- ▶ Traffic encryption
- ▶ Observability and traceability
- ▶ Authentication and authorization
- ▶ Failure recovery

For inter-application communication, services that will be exposed outside of the mesh – via the mesh ingress and egress gateways – and guarded by an API gateway. This approach enforces the application boundary, of which the mesh itself has no notion.

An API gateway also comes with additional benefits to the application:

- ▶ Gateway load balancing
- ▶ Gateway throttling
- ▶ Service mesh entry point security (for example generating tokens)
- ▶ Policy enforcement
- ▶ Abstraction from internal communication protocols

Additionally, if the API gateways are part of an API management solution, that solution can control how the application makes itself available to other applications within the mesh, outside of the mesh, and even beyond the enterprise. It allows consumers to explore the APIs available via a developer portal, and to subscribe to an appropriate plan of use for the APIs they want to use. Then, at run time, it enables the application owners to recognize, control and perhaps even bill differing types of consumers of the APIs. It ensures that all consumers

are identifiable, and can be managed separately in terms of traffic, access control, and more. It also ensures that the underlying implementations remain abstracted from the way that the APIs are ultimately exposed, performing any necessary routing and translation along the way. API management is discussed in more detail in 3.1, “Capability perspective: API management” on page 42 and Chapter 5.3, “API Lifecycle: IBM API Connect” on page 147.

In summary, a service mesh and an API management solution operate on two logically orthogonal planes:

The mesh is a capability that is distributed across the microservices platform. However, its use should be scoped such that it is used to take care of fine-grained, intra-application communication. This is often called *East-West* traffic.

The API management solution sits on the edge of the mesh. It provides gateways as scoped and secured entry points into the mesh to control access to the applications’ internal components, thus managing inter-application communication. This communication is often called *North-South* traffic.

Figure 4-20 shows East-West versus North-South traffic.

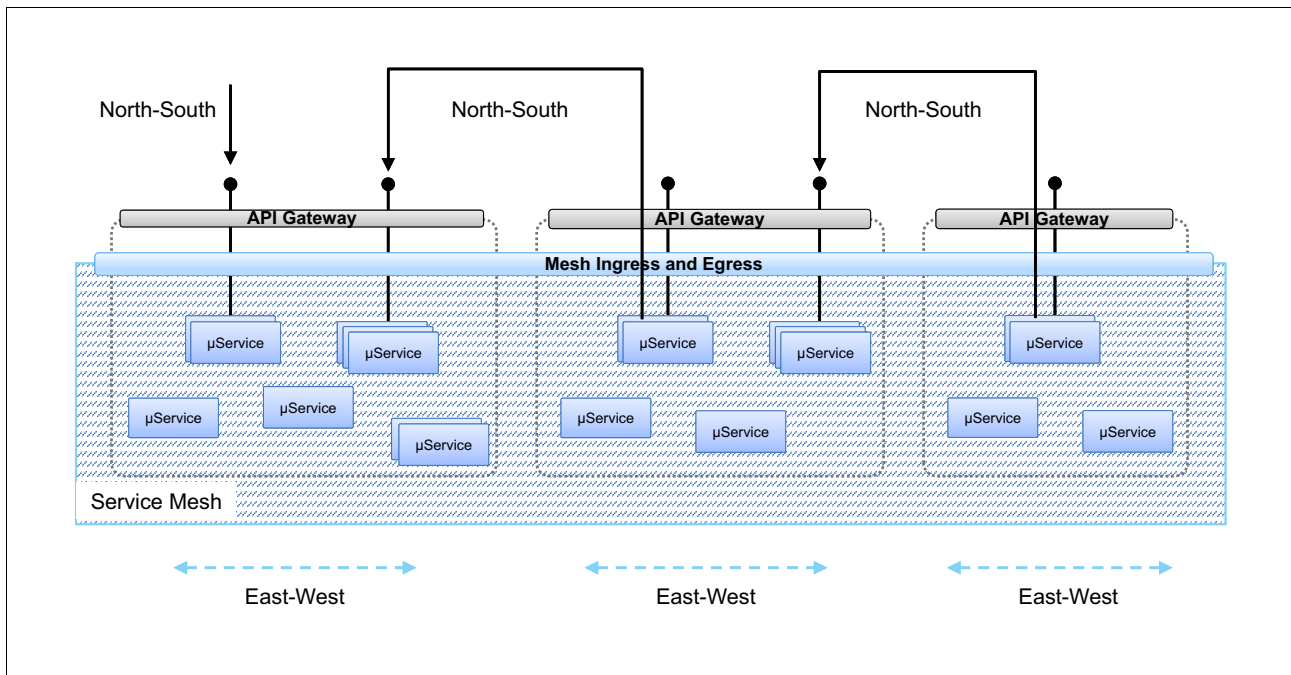


Figure 4-20 East-West versus North-South traffic - 1

It is important to notice that East-West traffic applies only to intra-application communication. The mesh might be physically distributed across a microservices platform that hosts multiple applications across different domains. Nonetheless, it is still the ownership boundary that dictates what type of interaction is taking place.

When an application boundary is present, the consumption of its APIs by components in other applications should not go directly via the mesh. This rule applies even when the consumer happens to be another application that is leveraging the same underlying mesh for its inter-application communication. Inter-application communication is always by nature North-South, and should go through the API gateway. See Figure 4-21 on page 127.

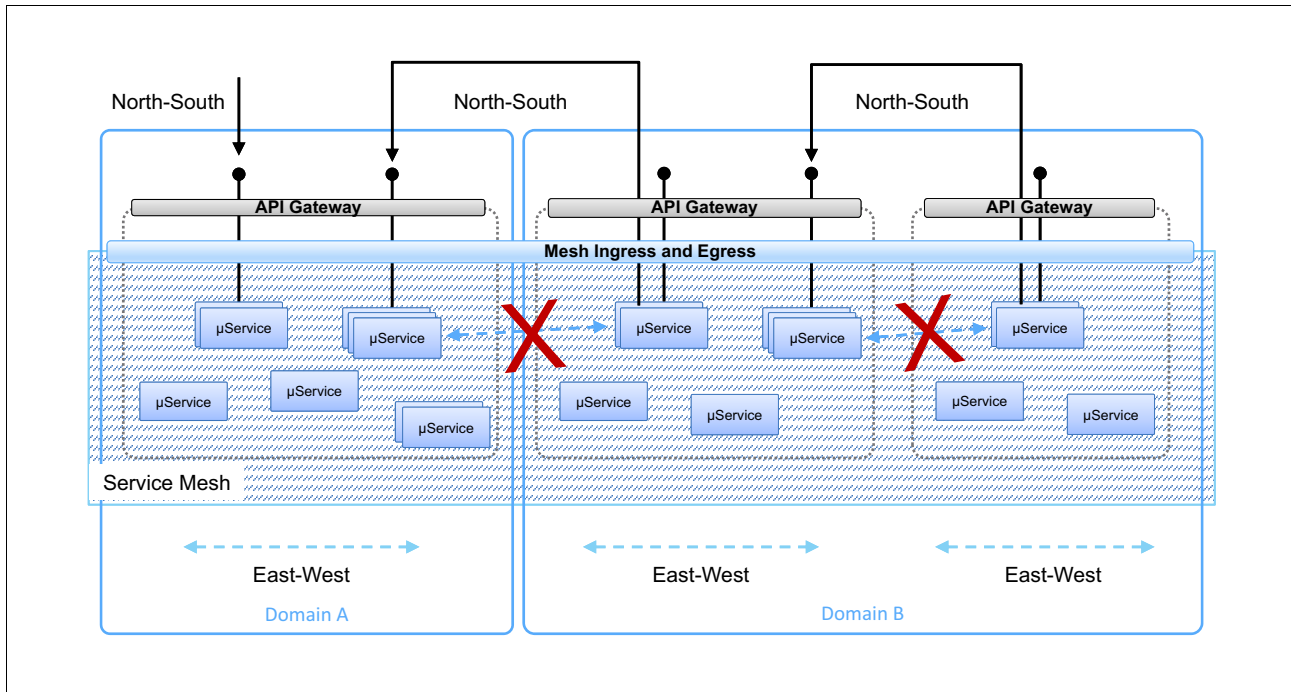


Figure 4-21 East-West versus North-South traffic - 2

Based on this distinction, it is easy to see where a service mesh and an API management solution differ (Table 4-2).

Table 4-2 Differences between a service mesh and an API management solution

Topic	Service mesh	API management
Traffic flow	Handles East-West / intra-application traffic	Handles North-South / inter-application traffic
Role regarding services	Manages and controls service inside the microservices platform	Exposes services and makes them easily consumable to other applications and consumers
Role regarding internal resources	Focuses on brokering internal resources	Maps external traffic to internal resources
Location	Sits between the network and application, without business notion of the application	Exposes APIs or edge services to serve the application's business function

However, although API Management and service mesh are logically orthogonal, their contact point is in their respective gateways:

- ▶ The API gateway for API management
- ▶ The Ingress and Egress gateways for the service mesh

It is at these touch points that the two collaborate, to provide consistent management when it comes to the numerous shared concerns across the two solutions:

- ▶ Shared definition of application APIs
- ▶ A shared registry of components within the application boundary
- ▶ Consistent traffic enforcement rules
- ▶ Consistent access control and authorization

- ▶ Securing of the service mesh
- ▶ Policy creation and propagation
- ▶ End-to-end traceability and observability

Commercially available API Management solutions are evolving to provide a tighter integration with the most popular service mesh implementations. As a result, those shared aspects can be *defined* in API Management yet *enforced* via the service mesh when they cross the application boundary. Specific technology solutions like IBM API Connect and Istio are described in more detail in Chapter 4, “Cloud-native concepts and technology” on page 85.

For more detailed discussion around service meshes and API management refer to the following material:

- ▶ <https://developer.ibm.com/apiconnect/2018/11/13/service-mesh-vs-api-management/>
- ▶ <https://developer.ibm.com/apiconnect/wp-content/uploads/sites/23/2018/07/API-and-Microservice-Management-Side-by-Side-PART-1-1.pdf>
- ▶ <https://blog.christianposta.com/microservices/api-gateways-are-going-through-an-identity-crisis/>
- ▶ <https://www.youtube.com/watch?v=53o15ESfpdM>

4.9 Cloud-native security – an application-centric perspective

Earlier in this chapter we looked at what makes cloud-native applications different from traditional applications. In this section, we’re going to consider what knock on effects this very different style of application implementation has on security.

4.9.1 Scope of this section

Security is a huge topic and we cannot hope to do it justice in this short section. Therefore, we focus on the security around application level interactions, because that is where the components of an integration portfolio have most relevance.

From a cloud-native application’s standpoint, we primarily look at:

- ▶ **Inbound interactions:** How do other applications talk to our cloud-native components?
- ▶ **Inter-component interactions:** How do our components talk to one another?
- ▶ **Outbound interactions:** How do our components talk to applications beyond our cloud-native application’s boundary, especially system still residing on-premises?

Notice that the preceding focus relies heavily on the idea that there is still a concept of an application boundary. This is true, despite the fact that we have potentially broken up what is within it into discrete components (for example microservices) that are separate runtimes that are sitting on the network. For more on this concept see 4.7, “Application boundaries in a container-based world” on page 113.

4.9.2 Limitations of traditional security models

The strongest security barrier in most organizations has typically been the de-militarized zone (DMZ) and its associated firewalls. This provided a double strength barrier between the internal networks on-premises and the public internet. The assumption was that the biggest

concern came from the significantly higher number and more unknown threats from the outside.

This resulted in a way of building security into applications that assumed things about the security of this “on-premises” environment. Common examples would include the assumption that the internal network was relatively safe from attack so components could be placed there with minimal security.

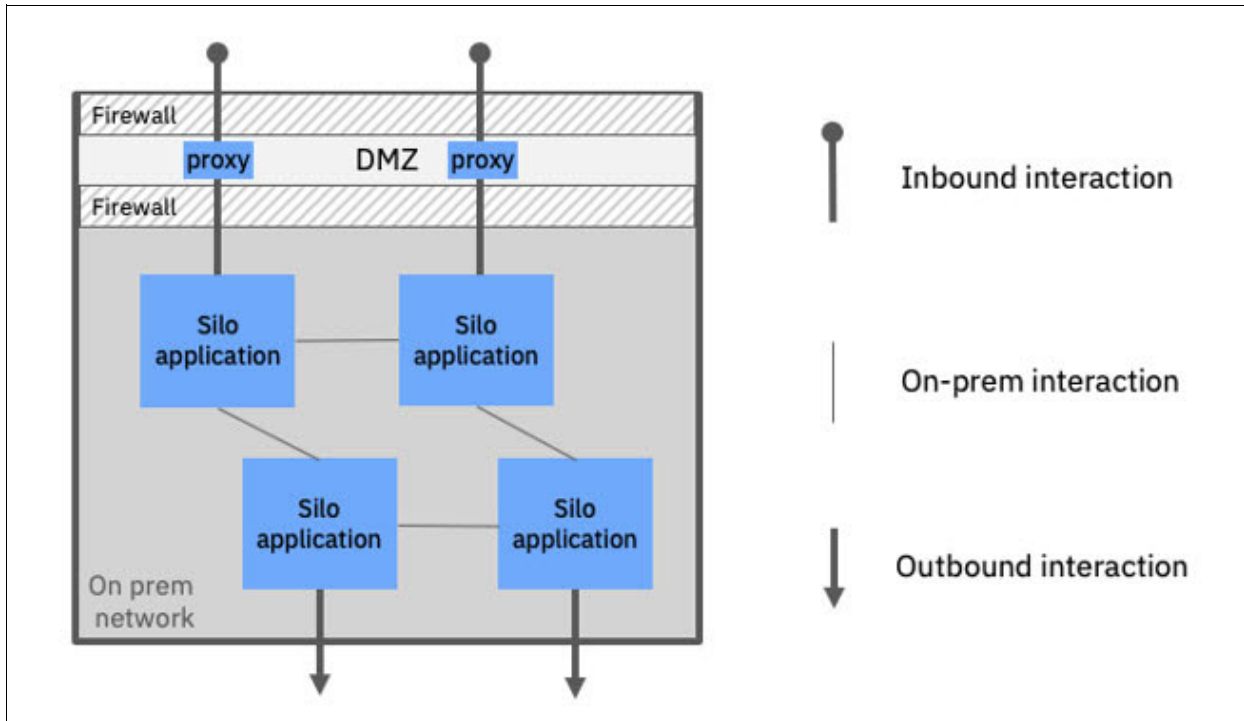


Figure 4-22 Building security into applications

Figure 4-22 implies that the applications on the internal network have a more relaxed interaction security. To be fair, most applications do of course implement some form of security pattern that involves a need have credentials in order to connect. But they do not have the high garden walls that are provided by the DMZ. Nor do they have firewalls to protect against broader types of attack such as denial of service.

We are now in an era where new solutions are increasingly developed and run in public cloud. Furthermore, any solutions that are built on-premises currently should consider that they might be moved into the public cloud at some point. This has multiple implications:

- ▶ If our application is running in a public cloud, we must consider deep security as a fundamental part of our application.
- ▶ We are likely to need to expose our capabilities to other applications that are calling us over the public internet.
- ▶ We are likely to need data from other applications that are accessible only over the public internet.

Even solutions that are to remain on-premises, we should be thinking differently about security. It has been shown that many of the most serious threats actually materialize internally rather than from the public cloud, so treating on-premises as a safe zone is inherently dangerous.

We noted earlier in this chapter that “cloud-native” does not mean “built on public cloud”, it means “built using cloud-based principles”. However, if we build a cloud-native solution, we should build it to defend against the potential for both internal and external threats. This is especially true of security considerations. We need to build our solution such that all of its components are appropriately robust to attack. And any barrier to the public internet that might be required, we put in place as part of the solution, rather than simply assume that it is provided by the environment that we are in.

4.9.3 Challenges unique to cloud-native

Cloud-native applications are different from tradition applications in many ways as discussed earlier in Chapter 2, “Agile integration” on page 5. Notably, they are more fine-grained, resulting in more components to secure, each with an attack surface of its own. The application might be deployed on a public cloud infrastructure. And this completely different way of deploying and administering the application makes it clear that we have some new security challenges that must be addressed.

- ▶ **Fine-grained components:** As components increasingly become more fine-grained, interactions that would have been within a runtime are now distributed across the network. Indeed, the number of permutations of communication paths has dramatically increased. There are many more potential communication paths between these components, but not all are valid. Effectively, we could say that all these fine-grained components have a potential unique attack surface that we need to consider. We want to make only the correct subset of interaction paths possible, but without introducing a significant administration overhead.
- ▶ **Public cloud:** These components might be deployed to public cloud infrastructures. How exposed are they to the public internet? We might build initially on private cloud. But we would want to design in a way that was ultimately portable to public cloud. So, we need to consider this aspect from the beginning.

4.9.4 Securing a cloud-native application

We now explore an approach to security for cloud-native applications focused as mentioned earlier in the scoping section on application level interactions. It comes in four elements:

- a. Establish application boundaries
- b. Isolate the application
- c. Use API management to guard the front door

We describe each of these in more detail in the following subsections.

Part A. Establish application boundaries

As noted in 4.7, “Application boundaries in a container-based world” on page 113 there is significant value in considering how to group our fine-grained cloud-native components. We used to have a natural grouping that is based on the coarse-grained application components. Now we have broken these up into more fine-grained microservices components.

If we don’t decide on some way to group these together, we are left with the task of defining security at the level of every single component. This task would quickly become challenging, if not impossible. From many perspectives, and especially that of security, it would be much easier to administer these components if application boundaries were defined. This would allow us to apply definitions as the application group level rather than individually. See Figure 4-23 on page 131.

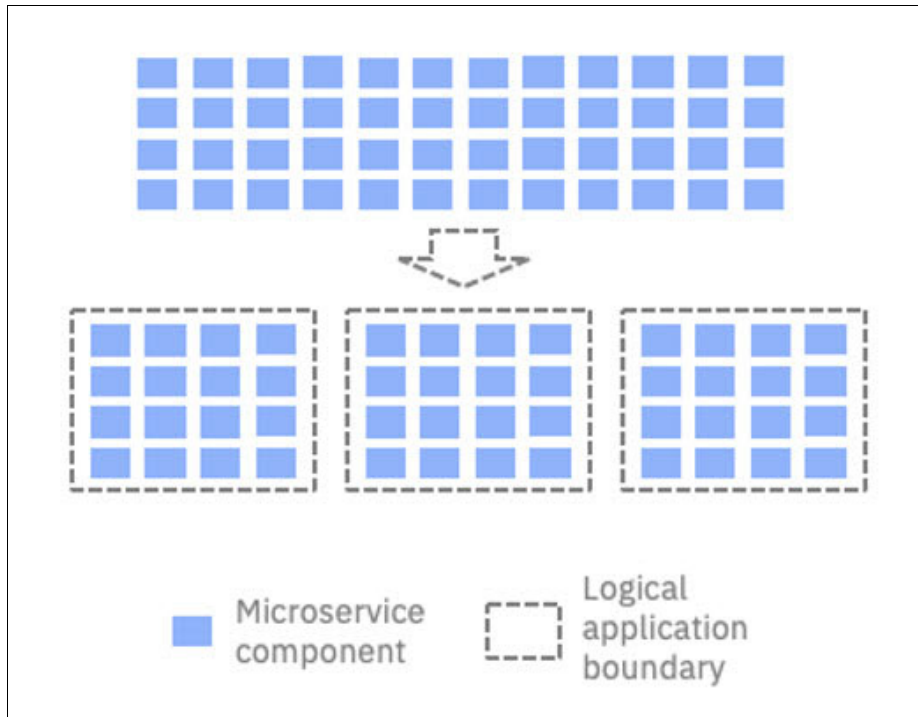


Figure 4-23 Application boundaries

While application boundaries are not essential to this approach, it does make it much easier to decide where to use what type of security techniques.

Part B. Isolate the application

In the previous section, we very deliberately used the term “logical” application boundary because there is nothing yet enforcing that boundary. In reality, the components still float around insecurely on the network.

The first thing that we need to do it to lock down the individual components. It is likely we are building our cloud-native solution in a container-based environment. This has the benefit that containers by default expose no interfaces.

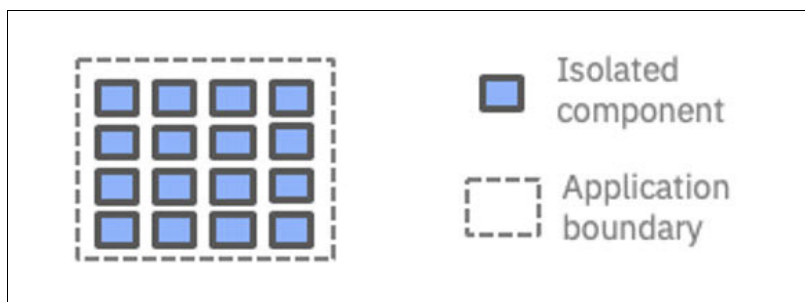


Figure 4-24 Isolated components

If we have not explicitly exposed ports in the Dockerfile the containers are inaccessible even within the K8s cluster as shown in Figure 4-24 on page 131.

Within the application boundary, some of the components need to talk to one another. We need to open up at least some of the ports on the components. So, we need to work out how to avoid those becoming available outside the scope of the application. One option, as shown

in Figure 4-25, would be to have all the components of a given application on a dedicated network.

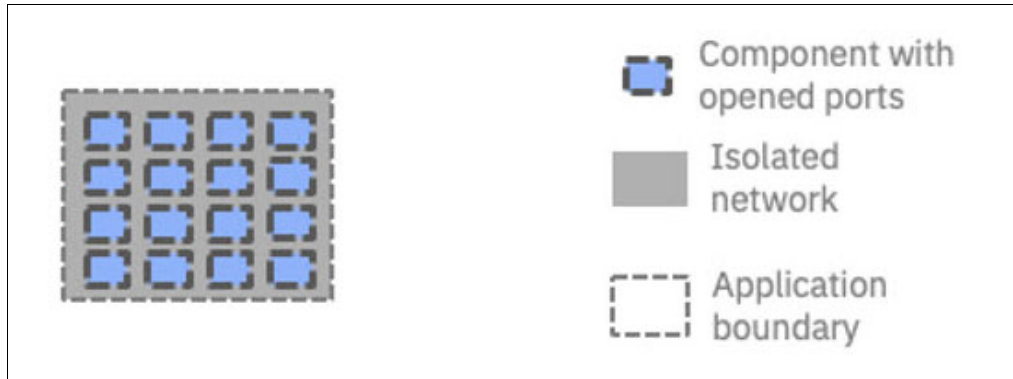


Figure 4-25 Components with open ports, protected by an application-specific network

A container that is deployed within a K8s cluster is not visible beyond the cluster by default, which is a good start. However, it would be visible to any other container on the cluster, even if it were in a different logical application. Clearly it would be quickly unsustainable to stand up a new K8s cluster each time we built a new application. Furthermore, we would lose out on the opportunity for infrastructure optimization between applications, which is one of the key benefits of using containers in the first place.

To group containers within a cluster into network spaces for each logical application, we could use something like namespaces and associated network policies. Or we could use a software defined networking capability such as Calico.

We have somewhat oversimplified the situation here. In reality, containers on K8s are actually exposed via constructs such as NodePorts and clusterIPs. But we do not have the time to go to that level of detail in this section, and besides, it is an area that is still maturing. Furthermore, there are alternatives as we discuss next.

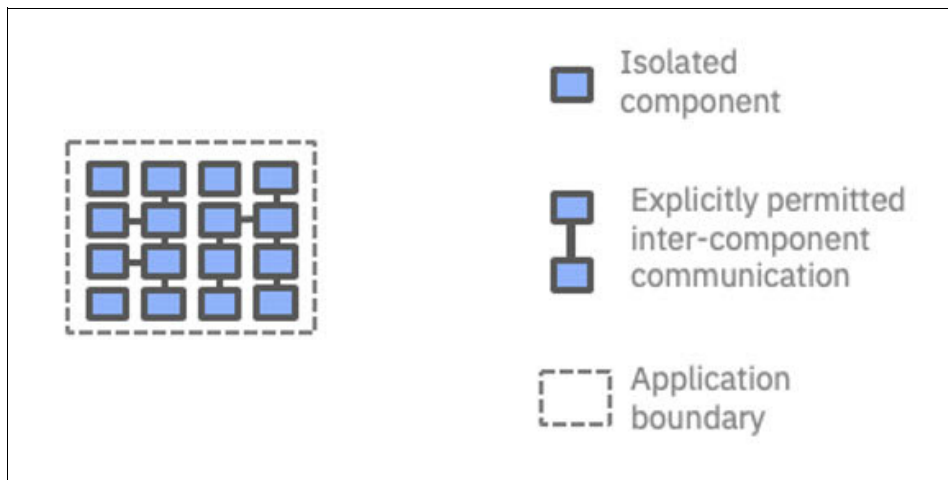


Figure 4-26 Explicitly permitted connections

What if we avoid trying to create an application-wide sub network. Instead, we can create explicit secure connections between the components within our application that need to talk to each other as in Figure 4-26.

Clearly this requires more individual configuration effort than the previous mechanism. But it does ensure a more well-defined behavior for the application. And this allows the application to be deployed to any network and be self-secure. There are mechanisms such as the services mesh Istio that we described earlier. Istio can make the preceding approach more straightforward. For example, it enables mutual TLS to be defined declaratively as part of the overall application code.

Part C. Use API management to guard the front door

So, now we have managed to lock down the application boundary, and nothing can reach our components. However, there are some interactions that we do want to reach our components, the most obvious example being any APIs that our overall application might want to expose beyond its boundary.

We would want to expose these APIs in a fairly formal and well-governed way and make them easy for our consumers to find. Ideally they would want to search for them in a developer portal where they could also then subscribe to use them. Therefore, it is likely that this would be the place where we would want to use an API management facility. See Figure 4-27 on page 133.

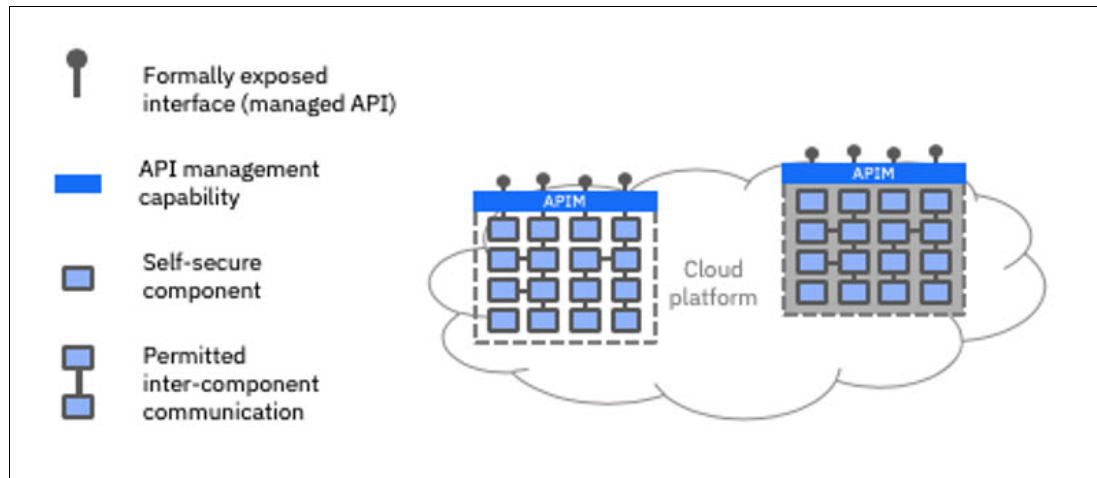


Figure 4-27 Use API management to guard the front door

We have discussed the capabilities of API management in detail already in 3.1, “Capability perspective: API management” on page 42. We also discussed how API management should be used on the application boundary in 4.7, “Application boundaries in a container-based world” on page 113, so we need not repeat that here.

Let’s remind ourselves of the core capabilities that a good API-management product should be bringing to the table from this particular application security viewpoint:

- ▶ Highly secure and performant, DMZ ready gateway
- ▶ Support for authorization delegation frameworks such as OAuth that can be used with tokens such as JWT to enable assertion and integrity (discussed next)
- ▶ Throttling of APIs independently per consumer
- ▶ Payload level encryption and signature

Basic encryption on the wire

The most common API protocols for new APIs are REST (Representational State Transfer) that uses JSON data format (or sometimes XML). However, there are still many instances of the older web services protocol on SOAP (Simple Object Access Protocol). Both of these are

typically (though not always) performed over HTTP. It has become commonplace for all HTTP APIs to be provided over Transport Layer Security (TLS) to encrypt the communication channel. This is an important first line of defense and accomplishes a basic level of confidentiality. But further encryption might be needed to provide more specific privacy for specific elements of the payload.

Authentication

Authentication relates to establishing identity. In the case of APIs, it tells us *who* is calling the API. Identity is present at two levels for APIs

- ▶ **Application:** What is the *application* that is making the calls? The application is the direct consumer of the APIs – the one that actually makes the calls.
- ▶ **End user:** Who is the *end user* who uses the application? This is the indirect user of the API, always using it via an intermediary application. However, it is their data that is being accessed and/or their access privileges that are being used.

We need these identities for a variety of reasons, as in these examples:

- ▶ To verify that their role allows them to perform the requested operation.
- ▶ To enable returning data that is related specifically based on their identity. We explore this example soon, when we discuss *authorization*.
- ▶ To enable us to record profiling information about their usage of the APIs. We can later view this data in the aggregate, through analytics. This data might help us to better design future APIs, or make decisions about how to promote products, or it could even be used to monetize the APIs by billing for their use.

The application identity is the primary identity and that might be enough for some use cases such as generic searches against data. The main aim here is generally to stop rogue or malicious applications from using the APIs. API management capabilities typically enable the use of an application key or client ID that the consumer receives when they first subscribe to the API in the portal. This is used in combination with a secret that they receive at the same time. The key/secret pair allows the API gateway to authenticate the identity of an application.

The *end-user identity* is required any time that we want to do something that is user specific. For example, before we access a user's account information, or purchase something on their behalf, we first need to know that they are who they say they are. Various authentication protocols exist for establishing end-user identity, a common example being OpenID.

API gateways can be configured to perform user authentication directly or as part of a broader flow such as OAuth, which we discuss in the next section.

Authorization

An authenticated identity alone just tells us who someone is, not what they can do. *Authorization* refers to what actions a given identity is authorized to perform, and in turn, what data they are allowed to access.

There are many options for specific consumer authorization. One of the most commonly used today is OAuth, which is an open standard for authorization and delegation. It is based on the exchange of an authorization token instead of a password, and it enables access by the application on your behalf. It is worth noting that it is now OAuth 2.0 that is typically implemented. The preceding OAuth 1.0 is increasingly less commonly supported. OAuth 2.0 is not backwardly compatible with OAuth 1.0.

Clearly, OAuth requires an identity before it can be used to enable authorization and this often happens as part of the higher-level flow, but it is done outside of OAuth itself. A common example of an OAuth flow would be where you want to enable an application to post

messages on social media. At the beginning of the flow, the application would send you to a Facebook authorization page. To get to that screen on Facebook, you must log in to Facebook itself (authentication). Facebook then can present a page to you where you agree to let the application post on your behalf (authorization). You are then directed back to the application, which now has the specific authorization to post only messages and nothing. Furthermore, it had not sight of your username/password on Facebook at any time during the flow.

For simplicity, we have skipped over some of the subtlety of the flow, to focus on the key points as seen from the user's perspective. Behind the scenes, there are many important details such as how Facebook confirms the identity of the application, safe authorization token retrieval, token expiry, token encryption, and so on. OAuth is a sophisticated protocol as are all the related standards; complex to learn and easy to implement insecurely. API management products aim to simplify these challenges as much as possible, enabling the setup of OAuth flows through guided configuration. For an example, see 6.5.3, "Securing the API" on page 265.

Tokens

The OAuth protocol does not specify the format of the tokens that are used to hold authorization information.

A common choice is the JSON Web Token (JWT). Ultimately a JWT is just a convenient way for transferring a token of data and it is used in many different security-related use cases. They can be digitally signed to ensure its integrity, and they are self-describing/self-contained.

One of the key benefits of tokens such as JWT is that they are self-signed. As a result, they can be validated without having to go back to the originator, thus reducing the number of network interactions needed. This becomes particularly important in cloud-native applications where there are many more fine-grained components that each need to validate the tokens that they receive.

However, one of the downsides of the self-contained token is that it is harder to revoke. After it is out in the wild, its self-validating nature makes it appear legitimate even if it is not. So, it is susceptible to man-in-the-middle attacks. Of course, the best defense against this is to reduce the self-contained time so that it expires in as quickly as possible (based on the use case). In other words, minimize the attack-opportunity window.

IBM API Connect provides combined OAuth and JWT capabilities which are discussed in detail in 6.5.3, "Securing the API" on page 265.

4.9.5 Hybrid solutions: Securing cloud to cloud and cloud to ground

In this final subsection, we consider how cloud-native applications should connect to other applications beyond its boundary.

The first consideration is how we enable a network path between our cloud-native applications that are spread to other clouds, or to our existing system of record on-premises (on the ground).

For illustration, consider the following deployment landscape where we have applications that are spread across three network zones: On-premise, IBM Cloud, and Google Cloud.

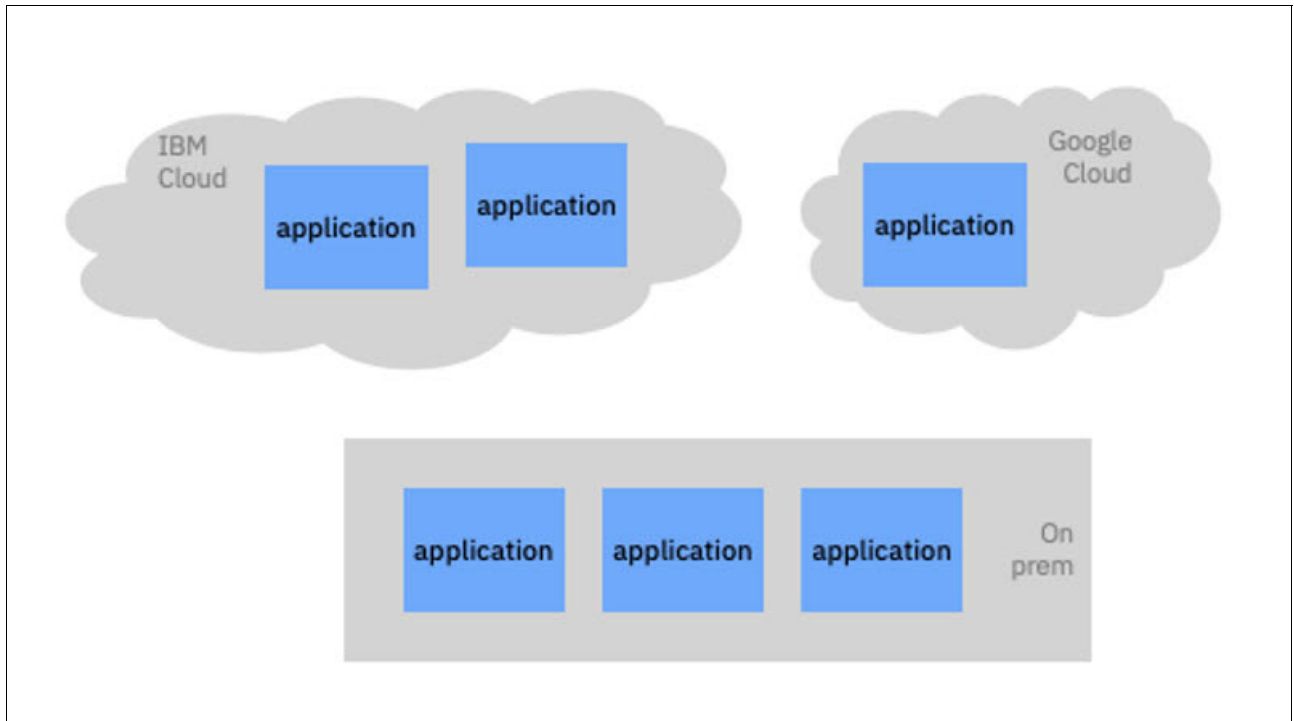


Figure 4-28 Isolated network zones in vendor clouds and on-premises

By default, each is in its own separate network zones that do not allow any inbound communication.

As shown in Figure 4-28, from a network standpoint we have options to provide connectivity between network zones. These options primarily depend on whether the target applications are accessible directly over the internet.

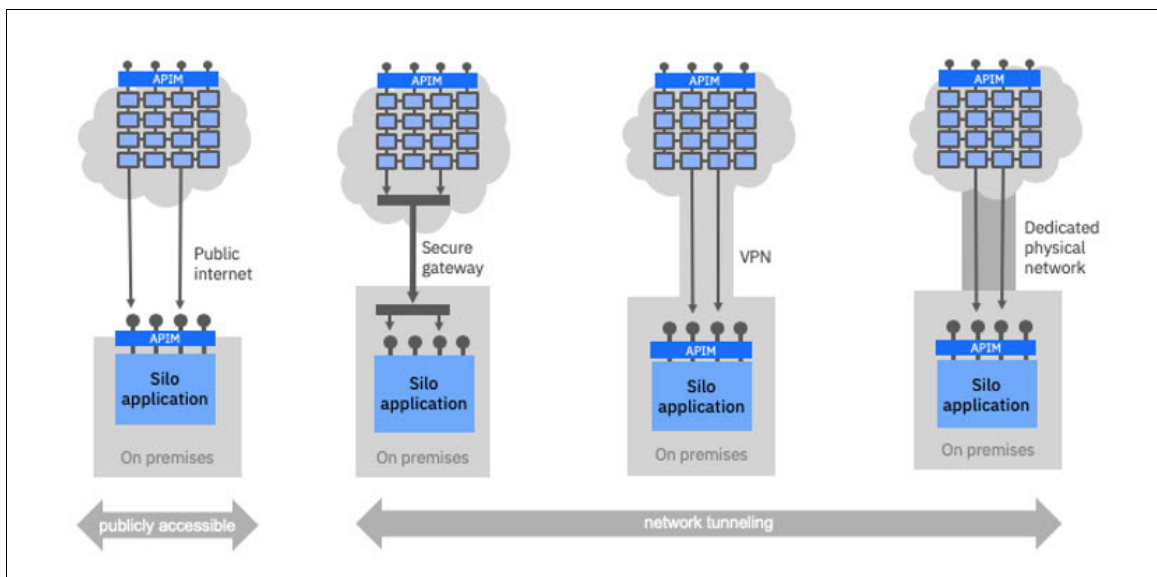


Figure 4-29 Proving connectivity across different networking zones

In Figure 4-29 on page 136 we have shown cloud to ground connectivity, for example, hybrid connectivity from the cloud to on-premise. But at this conceptual level, the options would be largely similar for cloud to cloud.

Target applications exposed on the internet

Although many protocols might be exposed on the internet, the most common for reusable capabilities today is exposure as APIs as we have discussed earlier in this chapter. From the perspective of the consumer – in our case the cloud-native application, this is the easiest way for us to consume an application outside our boundary.

It assumes of course that we have direct access to the internet. This is common, but not a given. There might be good reasons to restrict direct internet access to some applications.

This option is essentially as secure as the security model imposed by the exposed system. In the previous section, we discussed potential security models for APIs, with API keys/secrets and OAuth as our example. This scenario would likely be further secured at the infrastructure level by using further layers of network infrastructure and software such as these:

- ▶ Firewalls
- ▶ Proxy
- ▶ Servers
- ▶ Load Balancer

This is the ideal exposure mechanism for wider reuse. Consumers can discover APIs using a public portal, and self-subscribe to use them immediately without any involvement of other teams on either side. However, from the providers perspective there is some significant effort involved — procedurally and infrastructurally — to ensure that the APIs are exposed with appropriate security for public reuse. There are circumstances where this extra work and/or risk is unacceptable for a given solution.

Network tunneling for applications that are only privately accessible

The alternative to making applications public accessible is to make a private networking path between the applications. Common examples include:

- ▶ **Encryption tunnels:** These provide point-to-point secure communication between networks. Traffic is encrypted and transmitted across the internet. IBM Secure Gateway on IBM Cloud is an example. This explicitly exposes endpoints within the source network that map to a location in the target network. From a security point of view, it has the advantage of being more specific about what is exposed across the tunnel. But therefore, it has the disadvantage of requiring new configuration for each target/destination involved.
- ▶ **VPN:** These are network devices or software components that create a secure tunnel between networks, across the internet. Components can then see one another directly across the two networks, with the VPN connection (rather than the application) taking responsibility for ensuring it is encrypted across the internet. By default, a VPN, just like dedicated connectivity mentioned next, effectively turns the two zones into a bigger single “virtual” network. Clearly this makes things very simple – everything can see everything. However, from a security point of view this might not be desirable. Further configuration effort might then be required to lock down resources appropriately.
- ▶ **Dedicated connectivity:** Telecommunication providers supply dedicated physical connectivity between the two locations. Communication flows across the telecommunication provider’s private network, with no use of the internet. This likely enables the highest bandwidth options, but likely comes at a higher cost than the other two options and requires physical infrastructure changes. This effectively joins the two zones together as if they were one network. As discussed previously, this simplifies connectivity, but possibly at the cost of increased risk by exposing more attack surfaces.

Product specifics on tunnelling

There are a few special considerations for specific components of the IBM Cloud Pak for Integration, both of which fall into the “tunneling” category:

- ▶ *IBM MQ* provides Internet Pass-Through (IPT) which is a SupportPac for IBM MQ, which provides proxying capability for MQ traffic. It provides the capability to proxy the MQ traffic as is, or convert into an HTTP protocol to meet certain security policies. IPT also provides the capability to write extension points so logic can be applied to traffic. This is possible only due to IPT understanding the MQ protocol. Network devices commonly need to be configured to allow communication to the IPT server at the network boundary. IPT can be used in combination of network devices where appropriate and needed.
- ▶ *IBM DataPower* is a hardened security appliance and therefore completes some of the same capabilities that are highlighted in the network devices section. Therefore, it might be appropriate to delegate some of the network device security aspects to the IBM DataPower appliance.

Considerations for Network Tunneling

Generally, the Network Tunneling solution would be established by the network and security infrastructure teams when a new cloud environment is added into the enterprise. The exception to this is the Encryption Tunnel that is often a bespoke solution that can be quickly established for a point-to-point secure solution. The focus of the Encryption Tunnel is allowing connectivity from a single remote environment into another environment. Therefore, Encryption Tunnels are often not ideal for a multicloud scenario where the number of environments can grow or change.

4.10 The future of cloud-native

At the beginning of this chapter, we spoke about how we had moved to increased abstraction from hardware, from VMs, and then more recently moved to containers that abstract us even from which operating system we are running on.

We have discussed how containers bring us many benefits, a key one being in terms of operational consistency. We need only one infrastructural skillset to manage multiple different types of product runtimes, where before they each had their own intricacies such as HA, scaling and so on.

These are big steps forward in terms of infrastructural simplicity and efficiency, and they significantly reduce the time and cost to provision infrastructure for new projects. Clearly, we have moved into a territory where implementation teams can provision their own infrastructure when they need it. See Figure 4-30 on page 139.

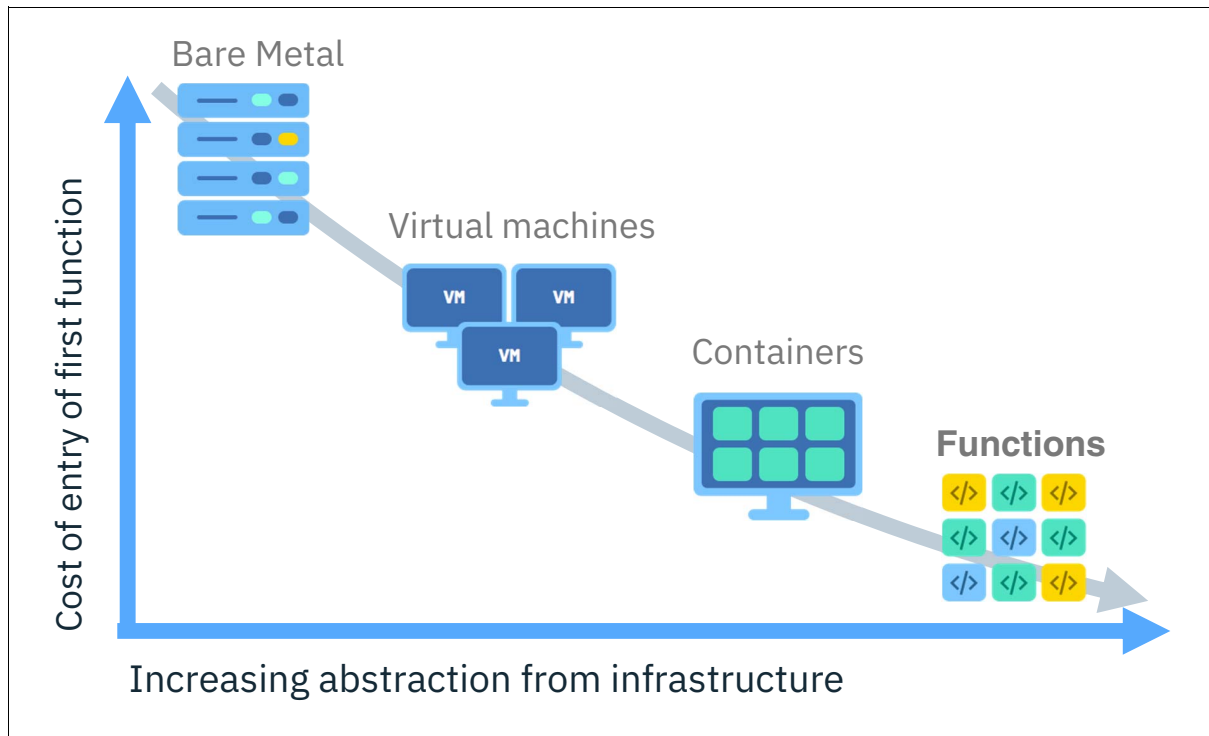


Figure 4-30 Increasing abstraction from infrastructure

However, we are not at the end of the road yet, there is certainly more that can be done. We still need to look after the deployed containers. We are still very aware of how many we are running, and how they are scaled and secured.

What if we could completely abstract ourselves even from the management of the runtimes themselves. What if we could just the implementation technology, write the code, and then give it to the infrastructure to run it, keep it available, scale it, secure it. In effect, this is the model that “serverless computing” has adopted – although the name *serverless* has fallen quickly out of favor for reasons we explain shortly.

4.10.1 Are software-as-a-service applications serverless?

Traditionally, “serverless” means that the users of a capability do not need to have any notion of the servers required to run it. This definition has been around pretty much forever in the computing industry. Looking at today’s technology, any web-based multi-tenant application that we use daily could be described as “serverless”. We could list hundreds of SaaS application as examples from Gmail to GitHub, to Workday, Salesforce, and any number of social media applications, and so on.

As a user who subscribes to any of those services, we have no notion of how many “servers” are working for us in the background. If we ask the application to do more work for us, it scales up invisibly in the background. Our cost model (if we pay for subscription at all) is based on user-focused notions of usage, for example, number of users, number of jobs submitted, number of invocations made, amount of data transferred, and so forth. Clearly, we could find examples of that model as far back in computing as you care to look. However, *this is not what is typically being referred to when we speak of serverless*. Obviously, this term can easily be misconstrued. So, a new term was quickly introduced: *function-as-a-service* (FaaS).

4.10.2 Function-as-a-service: a more accurate term for serverless?

The term serverless really became mainstream only around 2015 as you can see from a quick look at Google Trends:

<https://trends.google.com/trends/explore?date=all&geo=US&q=serverless>

Back in 2015, serverless was specifically attached to the idea of being able to write code (a *function*) and then ask an online capability to run that code for you instantly. Any scaling was provided invisibly in the background. After the idea was out, key vendors quickly came up with hosted services. Amazon Lambda was the first, but was quickly followed by Microsoft Azure Functions, Google Cloud Functions, and finally IBM Cloud Functions, which were based on the open source project in Apache OpenWhisk®. The next chapter focuses on Knative, which standardizes the way we achieve the serverless model in K8s.

As you can tell from the naming of these capabilities, it quickly became more technically appropriate to label this technology “function-as-a-service (FaaS)”. However, if you plot FaaS, the term doesn’t rank particularly high. Clearly, “serverless” is still very much in use.

<https://trends.google.com/trends/explore?date=all&geo=US&q=serverless,microservices,cloud%20native>

4.10.3 Could any runtime be provided in a FaaS model?

In theory, yes. For users, the big difference with the model is as follows:

- ▶ That the user pays only when a function is running.
- ▶ The runtime must dynamically and invisibly manage increases in load.

For vendors, the efficiency of the FaaS model depends on this: the runtime must scale up rapidly (from zero load) and down just as fast. This is of course an oversimplification. Many issues affect how easy it is to provide an isolated process space, how code is loaded onto the runtime and more.

4.10.4 FaaS for cloud-native?

It is easy to see how this model is very attractive to cloud-native applications. Look through our list of cloud-native elements at the beginning of the chapter. You quickly see how it ticks all the boxes, providing benefits in increased agility, fine grained deployment, auto-scaling, discrete resilience, and infrastructure optimization. It is perhaps the ultimate in terms of enabling a cost model for your infrastructure that scales precisely with the success of your business model.

4.10.5 Are there downsides to FaaS?

FaaS models clearly aren’t suited to every problem. Some of the current problems relate to the maturity of the technology and will improve over time, such as the ability to monitor and perform fault diagnosis on a platform that you have little access to. Some problems are more architecturally fundamental, such as the increased latency for some scenarios.

Like any model, FaaS has pluses and minuses. Security is a case in point:

- ▶ FaaS raises many of the usual cloud-based concerns over where data is kept, who has access to production, what is encrypted, how well is it protected from the public internet, and so on.

- ▶ It seems reasonable to expect that a FaaS vendor's be highly effective and trustworthy in this space, and that they should have more experience putting the right security patterns in place than you do.
- ▶ Furthermore, it is in the vendor's interest to keep up to date on software versions across all aspects of the platform, and do that with minimal or no disruption to users.

These qualities are akin to the CA that we discussed earlier, but with the FaaS vendor taking on the challenge rather than you.

4.10.6 Conclusions on FaaS

Ultimately whether FaaS is right for your particular initiative is going to depend on the preceding factors and more. FaaS models are sufficiently attractive that they continue to increase in use. Likewise, the different types of runtimes that become available on FaaS platforms is likely to increase.

In relation to the topic of this book, it is interesting to consider what a FaaS type model would look like for integration. For example, IBM App Connect is already available as a fully online and managed service. You are charged for the invocations/events passing through and have no notion of how many servers might be satisfying that load underneath. Furthermore, IBM API Connect can be purchased as a managed service whereby you pay based on the number of API invocations. Therefore, the FaaS (serverless) model is already present in the integration space, and very well suited to it.



IBM Cloud Pak for Integration

This chapter provides an overview of IBM integration. The capabilities can be purchased collectively as the IBM Cloud Pak for Integration, or individually for any particular capability. Each of the products is detailed in the following sections:

- ▶ IBM Cloud Pak for Integration
- ▶ Red Hat OpenShift Container Platform
- ▶ API Lifecycle: IBM API Connect
- ▶ Integration security: IBM DataPower Gateway
- ▶ Application integration: IBM App Connect™
- ▶ Enterprise Messaging: IBM MQ
- ▶ Event Streaming: IBM Event Streams
- ▶ High-Speed File Transfer: IBM Aspera®
- ▶ Service Mesh: Istio

5.1 IBM Cloud Pak for Integration

IBM Cloud Pak for Integration is a simple, complete solution to support a modern, multicloud-capable approach to integration. The Cloud Pak for Integration allows organizations to power their digital transformation initiatives. With IBM Cloud Pak for Integration, you can apply the appropriate organizational models and governance practices to support agile integration, simplify the management of your integration architecture, and help lower costs.

Using a container-based approach, IBM Cloud Pak for Integration gives businesses complete choice and agility to deploy workloads on-premises and on private and public clouds to extend scale, flexibility, and security as needed. It provides the modularity and scalability that is required from a modern integration platform to provide the right integration pattern at the right time.

The Cloud Pak for Integration contains the following critical integration capabilities:

- ▶ **API Lifecycle:** Create, secure, manage, share, and monetize APIs across clouds while you maintain continuous availability. Take control of your API ecosystem and drive digital business with a robust API strategy that can meet the changing needs of your users.
- ▶ **Application and Data Integration:** Integrate all of your business data and applications more quickly and easily across any cloud, from the simplest SaaS application to the most complex systems. You avoid concerns about mismatched sources, formats, or standards.
- ▶ **Enterprise Messaging:** Simplify, accelerate, and facilitate the reliable exchange of data with a flexible and security-rich messaging solution that's trusted by some of the world's most successful enterprises. Ensure you receive the information you need, when you need it — and receive it only once.
- ▶ **Event Streaming:** Use Apache Kafka to deliver messages more easily and reliably and to react to events in real time. Provide more-personalized customer experiences by responding to events before the moment passes.
- ▶ **High-Speed Data Transfer:** Send, share, stream, and sync large files and data sets virtually anywhere, reliably and at maximum speed. Accelerate collaboration and meet the demands of complex global teams, without compromising performance or security.
- ▶ **Secure Gateway:** Create persistent, security-rich connections between your on-premises and cloud environments. Quickly set up and manage gateways, control access on a per-resource basis, configure TLS encryption and mutual authentication, and monitor all of your traffic.

5.1.1 One platform supported by common services

The Cloud Pak for Integration is one of several Paks that IBM makes available (<https://www.ibm.com/cloud/paks>). Each Pak is built as an enterprise-ready, containerized software offering that leverages Red Hat OpenShift as the Kubernetes underlying platform.

5.1.2 IBM Cloud Pak for Integration - benefits

Beyond the key benefits IBM provides in all Cloud Paks, the integration platform further extends those benefits in four key areas:

- ▶ **Consumability:** The Cloud Pak for Integration includes a *platform navigator*, which is a single entry-point for users. It allows users fast access to all of the integration capabilities. Through this unified experience, it is simple to navigate across any of the product

capabilities. Deployment, management, and monitoring are done consistently across all integration capabilities.

- ▶ **Productivity:** The asset repository allows users to share integration assets across various integration capabilities in the platform. For example, an OpenAPI specification that is stored in the repository can be directly imported within the API management user interface.
- ▶ **Monitoring:** The underlying container platform takes care of collecting the logs from all components in use. The benefit is that all logs are in one place and the monitoring and dashboards are built upon this information.
- ▶ **Tracing:** By using standards-based OpenTracing, the IBM Cloud Pak for Integration is able to provide a consolidated tracing experience whereby the passage of an invocation or message can be followed as it passes through any of the integration capabilities. For example, a consumer might call an API that is exposed by IBM API Connect. This API in turn calls IBM App Connect in order to perform an aggregation across multiple systems. We might communicate with one of those systems over IBM MQ. The Pak's integrated diagnostics tooling allows us to trace the entire call from end to end.

5.1.3 License flexibility for other non-containerized architectures

Organizations are on different journeys toward integration modernization, and not everyone is ready to embrace a containerized deployment architecture. For this reason, IBM Cloud Pak for Integration continues to provide and support traditional software deployment options for custom bare-metal and virtual images. Regardless of that choice, the key integration capabilities of IBM Cloud Pak for Integration are available in all topologies.

5.1.4 Getting access to IBM Cloud Pak for Integration for the exercises

Many of the chapters in this book assume that the reader has access to an IBM Cloud Pak for Integration instance.

If one is not currently available to you, IBM provides ways of provisioning instances on demand:

- ▶ Through a free-of-charge demonstration environment:
<https://www.ibm.com/demos/collection/Cloud-Pak-for-Integration/>
- ▶ Through a paid-per-usage enterprise environment:
<https://cloud.ibm.com/catalog/content/ibm-cp-integration>

You can benefit from a basic familiarity with Kubernetes concepts, which are discussed in 4.4.3, “Kubernetes primer ” on page 107.

In order to do the exercises based on IBM Cloud Pak for Integration you need to,

- ▶ Install the Cloud Pak command line interface (CLI), which is available at:
https://www.ibm.com/support/knowledgecenter/SSGT7J_19.4/cloudctl/3.2.3/install_cli.html
- ▶ Install the OpenShift command line interface (CLI), which is available at:
<https://cloud.ibm.com/docs/openshift?topic=openshift-openshift-cli>
- ▶ Have access credentials to the IBM Cloud Pak for Integration instance.
- ▶ Know the IP addresses of the Kubernetes Nodes that run the IBM Cloud Pak for Integration instance.

5.2 Red Hat OpenShift Container Platform

The IBM Cloud Pak for Integration standardizes on Kubernetes as the container orchestration platform. In particular, the IBM Cloud Pak for Integration makes an informed choice to use the Red Hat OpenShift Container Platform as the underlying Kubernetes platform, and ships it as part of its installation.

In 4.4.2, “Container orchestration” on page 107 the importance of container orchestration was discussed. In particular, this book focused on Kubernetes as today’s most widely adopted container orchestrator.

Kubernetes is an advanced technology when it comes to container orchestration. However, it is generally recognized that Kubernetes on its own is not a complete cloud-native microservices platform. Additional capabilities that go beyond container orchestration are typically required for a full cloud-native platform.

In this section, we describe the open Kubernetes-based cloud-native microservices platform: Red Hat OpenShift Container Platform.

Red Hat OpenShift Container Platform (<https://www.openshift.com/products/container-platform>), or RHOCP, is a platform based on Red Hat Enterprise Linux and Kubernetes that focuses on deploying and managing containerized, enterprise-grade workloads.

RHOCP is designed to run in multiple scenarios:

- ▶ On physical and virtual machines, on and off premises
- ▶ On private and public Infrastructure as a Service
- ▶ As a managed service by public cloud vendors, such as IBM and Microsoft Azure.

Red Hat OpenShift Container Platform, as shown in Figure 5-1 on page 147, extends Kubernetes with a set of capabilities that create a holistic, cloud-native platform for containerized applications:

- ▶ A container runtime that is compliant with the Open Container Initiative (<https://www.opencontainers.org/>): (CRI-O <https://cri-o.io/>)
- ▶ A container registry
- ▶ Monitoring, metering, and logging services
- ▶ Deployable application runtimes
- ▶ A set of application lifecycle management services
- ▶ An automated operations framework, based on operators (<https://coreos.com/operators/>)
- ▶ A service mesh based on Istio (see 5.9, “Service Mesh: Istio ” on page 164)

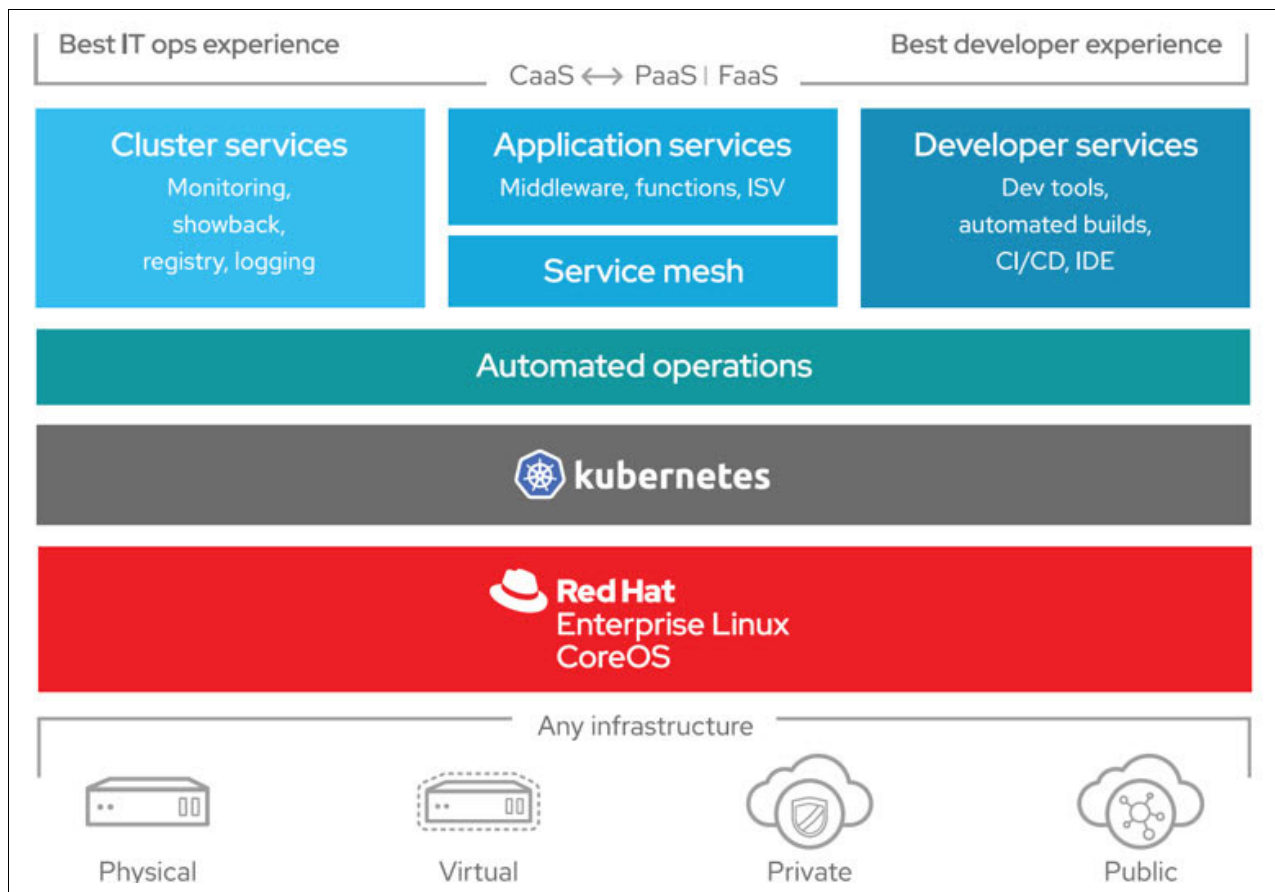


Figure 5-1 Red Hat OpenShift Container Platform

These components are preintegrated and tested together for unified operations. RHOCP also provides capabilities for hybrid cloud deployments. It can be used across on-premises and public cloud infrastructures, enabling a hybrid approach to how applications can be deployed as a self-managed solution.

After the cluster and applications are deployed, lifecycle management for these components, consoles for operators and developers, and security throughout the entire lifecycle become critical. Red Hat OpenShift Container Platform offers automated installation, upgrades, and lifecycle management for every part of the container stack — the operating system, Kubernetes, and cluster services and applications. The result is a secure, Kubernetes-based cloud-native platform, which removes the hindrance of manual and serial upgrades, or downtime.

Additionally, the platform integrates tightly with standard continuous integration/continuous delivery (CI/CD) tools, and OpenShift’s built-in workflows and tools, for security-focused application builds.

5.3 API Lifecycle: IBM API Connect

IBM Cloud Pak for Integration includes integrated API management capabilities with tooling for all phases of the API lifecycle. Key steps of the API lifecycle include create, secure, manage, socialize, and analyze. It includes the ability to deploy in complex, multicloud topologies (both on-premises and on cloud). The latest version also provides enhanced

experiences for developers and cloud administrators. The API Lifecycle capabilities are also sold separately as *IBM API Connect*.

API Lifecycle has two main focuses: the first is providing best in class API management tooling, and the second is having a cloud native solution. This allows users to create, manage, and secure applications that are deployed across a variety of on-premises and cloud environments.

5.3.1 Key phases of the API Lifecycle

The following explains the key phases in the API lifecycle in more detail.

- ▶ **Create:** Develop and write API definitions from an API development environment, eventually bundling these APIs into consumable products, and deploying them to production environments.
- ▶ **Secure:** Leverage the best-in-class API Gateway, gateway policies, and more, to manage access to your APIs and back-end systems.
- ▶ **Manage:** Governance structures are built in to the entire API lifecycle, from managing the view/edit permissions of APIs and Products being deployed, to managing what application developers can view and subscribe to when APIs are deployed.
- ▶ **Socialize:** Leverage an advanced Developer Portal that streamlines the onboarding process of application developers, and can be completely customized to an organization's marketing standards.
- ▶ **Analyze:** Developers and Product Managers alike are given the ability to understand their API traffic patterns, latency, consumption, and more to make data driven insights into their API initiatives.

5.3.2 API Lifecycle components

IBM API Connect includes four major components: API Manager, Analytics, Developer Portal, and Gateway. These four components can be deployed in a variety of hybrid and multicloud topologies. The infrastructure can either be deployed and managed by an IBM team in an IBM Cloud environment. Or it can be deployed and managed by the customers in their own dedicated environment or third-party cloud. There is also the option for having hybrid scenarios, for example, with the API Connect Reserved Instance Offering users are able to have their API Manager and Developer Portal running in the IBM Cloud, but then place remote gateways next to their back-end services.

In addition to the preceding point, there is a Cloud Manager to manage the IBM API Connect topology and a developer toolkit for offline API definition.

API Manager

The API Manager provides a user interface that facilitates promotion and tracking of APIs that are packaged within Products and Plans. API providers can move the Products through their lifecycle, and manage the availability and visibility of APIs and Plans.

Catalogs and Spaces are created in the API Manager to act as staging targets through which APIs, Plans, and Products are published to consumer organizations. API providers can stage their Products to Catalogs or Spaces, and then publish them to make the APIs in those Products visible on a Developer Portal for external discovery.

To control access to the available API management functions, users in the provider organization can be set up in the API Manager UI with assigned roles and permissions. API

providers can also use the UI to manage the consumer organizations that sign up to access their APIs and Plans. Additionally, developer communities can be created as a way of grouping together a collection of consumer organizations to whom a particular set of Products and Plans can be made available.

The API Manager UI also includes functions to manage the security of the API environment, and provides access to analytics information about API invocation metrics within customizable dashboard views.

API Gateways

Gateways enforce runtime policies to secure and control API traffic, provide the endpoints that expose APIs to the calling applications, and provide assembly functions that enable APIs to integrate with various endpoints. They also log and report all API interactions to the analytics engine, for real-time and historical analytics and reporting. The gateway that is used is the IBM DataPower Gateway, which is an enterprise API Gateway that is built for departments and cross-enterprise usage. This gateway provides a comprehensive set of API policies for security, traffic management, mediation, acceleration, and non-HTTP protocol support. It can be deployed as a virtual or physical appliance and supports multiple catalogs per instance or cluster. The DataPower Gateway can handle enterprise-level complex integration, and supports containers for flexible runtime management.

IBM DataPower Gateway is described in more detail in 5.4, “Integration security: IBM DataPower Gateway” on page 152.

Developer Portal

The Developer Portal provides a customizable self-service web-based portal to application developers to explore, discover, and subscribe to APIs.

When API providers publish APIs in the API Manager, those APIs are exposed in the Developer Portal for discovery and usage by application developers in consumer organizations. Application developers can access the Developer Portal UI to register their applications, discover APIs, use the required APIs in their applications (with access approval where necessary), and subsequently deploy those applications.

The Developer Portal provides additional features, such as forums, blogs, comments, and ratings, for socialization and collaboration. API consumers can also view analytics information about the APIs that are used by an application, or used within a consumer organization.

API Analytics

IBM Cloud Pak for Integration provides the capability to filter, sort, and aggregate your API event data. This data is then presented within correlated charts, tables, and maps, to help you manage service levels, set quotas, establish controls, set up security policies, manage communities, and analyze trends. API analytics is built on the Kibana V5.5.1 open source analytics and visualization platform, which is designed to work with the Elasticsearch real-time distributed search and analytics engine.

Cloud Manager

The API Connect Cloud Manager component is used to manage the API Connect on-premises cloud. The Cloud Administrator uses this UI to:

- ▶ Define the cluster of Management servers, Gateway servers, and containers that are required in the cloud, and configure the topology.
- ▶ Manage (modify, move, remove, restart, reboot) the servers in the cloud.
- ▶ Monitor the health of the cloud.

- ▶ Define and manage the provider organizations that develop APIs. (Assigned managers or owners of provider organizations can also complete this task.)
- ▶ Define additional cloud administrators, or set up users with roles that enable access to specific capabilities.
- ▶ Add user registries for authenticating users and securing APIs, and configure the secure transmission of data (for example, through websites).

The developer toolkit

The developer toolkit provides the tools for modeling, developing, and testing APIs and LoopBack® applications. The developer toolkit is installed locally, for offline API development. It includes a command line interface (CLI). It also incorporates LoopBack, an open source Node.js framework to enable rapid creation of Node.js implementations.

API developers use the API management functions in the API Manager or the CLI to create draft API definitions for REST and SOAP APIs, or for OAuth provider endpoints that are used for OAuth 2.0 authentication. The API definitions can be configured to add the API to a Product, add a policy assembly flow (to manipulate requests/responses), and to define security options and other settings. APIs can then be tested locally before they are published, to ensure that they are defined and implemented correctly.

For more information on IBM API Connect, see the IBM Knowledge Center:

https://www.ibm.com/support/knowledgecenter/SSMNED_2018/com.ibm.apic.overview.doc/api_management_overview.html

5.3.3 API lifecycle in combination with other capabilities

When we make a statement that IBM API Connect is *deployed* into a particular 3rd party cloud, we might actually mean a few different things. For example, the topology might be customer managed (in the same way that it would be, if it were on-premises), or it might be managed by IBM. It might be a public managed platform where it is shared with other customers, or it might be dedicated/reserved for the customer. In this section, we walk through the various possibilities of what deploying to cloud can mean, and their relative benefits.

API lifecycle capabilities such as those in IBM API Connect were introduced specifically to focus on the management and secure exposure of APIs. This is described in some detail in 3.1.1, “A brief history of API management” on page 42. As such, the API gateway should deliberately focus on only a very specific subset of integration capabilities to back-end systems, generally limited to HTTP based RESTful and web service based interactions. IBM DataPower Gateway’s capabilities are somewhat broader than this as described in the next section, but still, we would generally expect to use some additional, architecturally separate integration capability to perform deep integration to systems that cannot surface APIs themselves.

For this reason, IBM API Connect is often used in combination with IBM App Connect, which provides the richest set of application integration capabilities. The capabilities cover a broad range of protocols, composition and mapping, and data formatting possibilities. This is also part of the Cloud Pak for Integration and is described in 5.5, “Application integration: IBM App Connect” on page 155.

Specifically, for mainframe connectivity it is worth considering the pros and cons of the many options that are available — including z/OS Connect EE — to surface mainframe data and functions as APIs. An excellent reference on this topic is IBM Redpaper *IBM Z Integration*

5.3.4 Product deployment options

When we make a statement that IBM API Connect is *deployed* into a particular 3rd party cloud, we might actually mean a few different things. For example, the topology might be customer managed (in the same way that it would be, if it were on-premises), or it might be managed by IBM. It might be a public managed platform where it is shared with other customers, or it might be dedicated/reserved for the customer. In this section, we walk through the various possibilities of what deploying to cloud can mean, and their relative benefits.

IBM API Connect installations are provided in Container/Kubernetes and OVA VMware form factors to run virtually anywhere. Keep in mind that OVA installations are based on Kubernetes internally, giving customers some of the benefits of a cloud-native solution without having to install their own Kubernetes environment.

Therefore, IBM API Connect can be deployed with Kubernetes, IBM Cloud Pak for Integration, and OVA (VMWare) and Red Hat OpenShift for maximum flexibility to enable a true hybrid or multicloud adoption.

IBM API Connect is available in three different deployment options described in this list:

1. IBM Managed API Connect on IBM Cloud

IBM API Connect is available on IBM Cloud in three different options, which gives you different flexibility and isolation options:

- a. **IBM Cloud Public:** This format leverages the advantages of hosting the services on the public IBM Cloud architecture. The Public cloud includes the following key features:
 - No additional hardware requirements.
 - No special firewall requirements.
 - Easy to pair with other IBM Cloud services.
- b. **IBM Cloud Dedicated:** Cloud Dedicated is a Cloud environment that is set up specifically for your organization and is managed by the IBM operations team. The Cloud dedicated deployment includes the following key features:
 - No additional hardware requirements.
 - A single-tenant environment that enables a higher level of stability.
 - More security because you are removed from the public cloud.
- c. **IBM API Connect Reserved Instance:** This option offers an individual API Connect instance that runs on IBM-managed infrastructure. Reserved Instance provides value by balancing the flexibility of a shared in Reserved Instance infrastructure with the isolation of a single-tenant deployment based on the topology and functionality of API Connect. Reserved Instance offers the following key features:
 - Common log-in with other IBM services that use IBMid.
 - Isolation from other users of the IBM Cloud public service.
 - Managed, monitored, and operated by the API Connect operations team.
 - Deployed across multiple pods within the datacenter-zone for resilience.
 - Optionally deployed as a High Availability (HA) deployment with a 99.95% SLA, for an additional cost.

2. Customer Managed API Connect on-premises or on any cloud

You have the flexibility to deploy on any cloud infrastructure by using these tools:

- Kubernetes
- Red Hat OpenShift (with the option to use the consolidated administrative user interface provided by the IBM Cloud Pak for Integration)
- VMware (also Kubernetes inside)

The table at the following site provides a convenient comparison of these different options:

https://www.ibm.com/support/knowledgecenter/en/SSMNE2_2018/com.ibm.apic.overview.doc/rapic_overview_apic_formats.html

5.4 Integration security: IBM DataPower Gateway

The Integration Security component helps provide security, control, integration, and optimized access to a full range of mobile, web, application programming interface (API), service-oriented architecture (SOA), B2B, and cloud workloads. As noted above, it is used as the API Gateway component within API Lifecycle, but it has much broader capabilities than just API exposure as we discuss in this section. The Integration Security capabilities are also sold separately as IBM DataPower Gateway. It is available in physical, virtual, cloud, Linux, and Docker form factors.

The Integration Security capabilities are extensively used in the industry and its usage can be broadly categorized into the following patterns:

- ▶ Security Gateway placed between the firewalls, as shown in the next figure
- ▶ API gateway, both as internal and external gateway
- ▶ Providing connectivity and mediation services in the internal network, close to the systems of record

Figure 5-2 shows DataPower usage patterns.

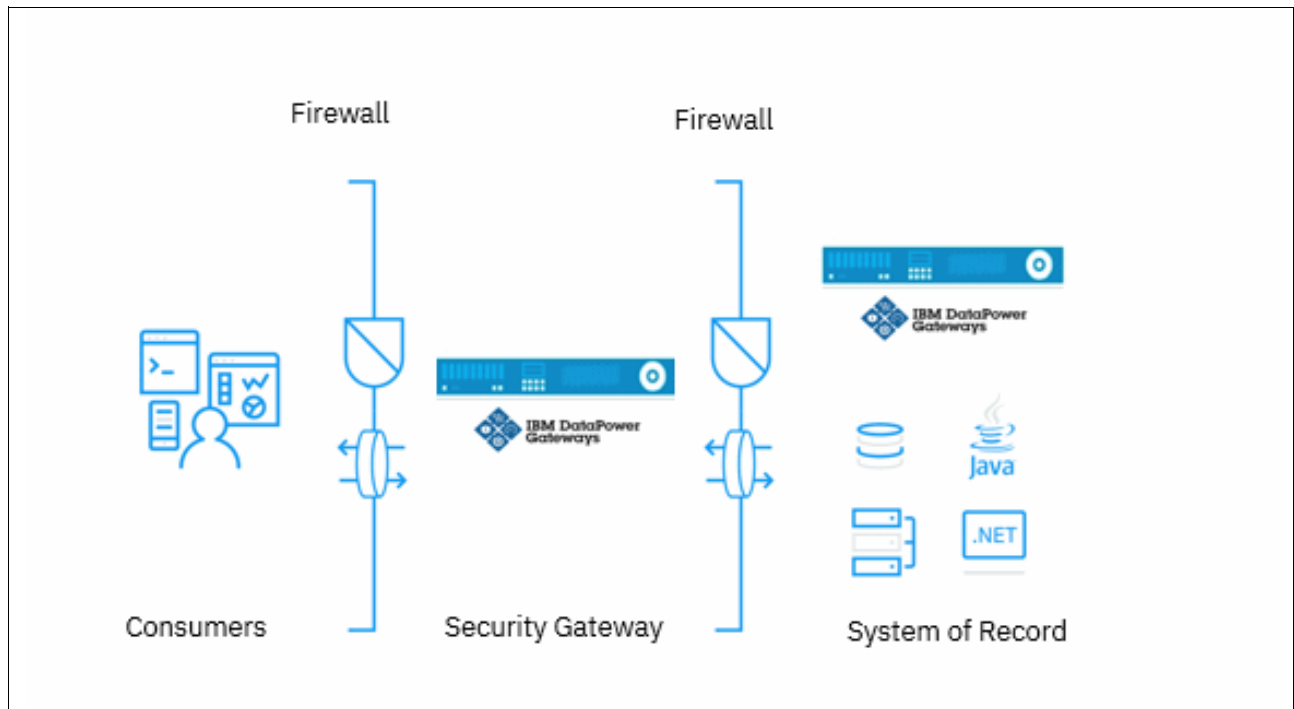


Figure 5-2 DataPower usage patterns

Let's discuss the patterns in detail.

5.4.1 Security Gateway

DataPower is purpose-built, DMZ-ready, and is used for converged policy enforcement and consistent security policies across business channels. As a result, you reduce operating costs and improve security. DataPower can,

- ▶ Protect against unwanted access, denial of service attacks, and other unwanted intrusion attempts from the network
- ▶ Verify identity of network users (identification and authentication)
- ▶ Protect data and other system resources from unauthorized access (authorization)
- ▶ Protect data in the network by using cryptographic security protocols:
 - Data endpoint authentication
 - Data origin authentication
 - Message integrity
 - Data confidentiality
- ▶ Provide proxying and enforcement:
 - Terminate incoming connection
 - Terminate transport-level security (SSL/TLS offload)
 - Threat protection
 - Enforce service level agreement policies
 - Inspect message content and filter (Schema validate)

DataPower also supports important security standards like OIDC, OAuth, WS-Security, JWT. As it is based on configuration-driven policy creation, it provides extensible platform to provide first grade security to the enterprise applications with minimum development effort.

For example, in Figure 5-2 on page 153 the DataPower in the DMZ provides consistent and advanced security gateway for the downstream applications. Without the secure gateway, these applications would have to develop enterprise grade security in order to expose or use external services. The overhead of building and managing of hardened security module can be avoided by using a secure gateway.

5.4.2 API Gateway

As an API Gateway the integration security capabilities are used for exposing internal APIs to consumers. It acts as a façade for the internal APIs as it has features like:

- ▶ Web services infrastructure needed to support highly secure data routing with daily high volume and sensitive nature of information.
- ▶ Centralized service governance and policy enforcement, as discussed in the above point.
- ▶ Service level monitoring (SLM) to protect your services and applications from over-utilization and enforce quota.
- ▶ Application optimization, which leverages dynamic runtime conditions to distribute based on topology and workload.

5.4.3 DataPower and agile integration

DataPower as a gateway inevitably performs a certain level of the underlying integration. But from an architectural point of view it would be unwise for it to be attempting to do all integration patterns. We need to consider what are its sweet spots, and where we should be delegating to deeper integration capabilities. Lets look at some of the integration capabilities that are provided by the gateway, and then when the crossover should occur to other technologies.

- ▶ **Security Gateway:** DataPower is obviously an excellent choice for the security aspects of a gateway, providing controlled, restricted access to downstream systems. Indeed, the physical appliance is often chosen to provide security gateway functionality as delivers up to 2X the performance of IBM DataPower Gateway (IDG). It has tamper proof hardware and software and can have a factory-installed hardware security module (HSM). An HSM provides secure storage for RSA keys and accelerates RSA operations. Due to these important security features, physical devices are preferred in the production environments.
- ▶ **API gateway:** With the rise in federated middleware and decentralized ownership, there is a marked increase in applications wanting to expose their APIs to other applications via their own infrastructure. Building enterprise grade custom API gateway is neither feasible nor desirable. IBM API Connect is a scalable API platform that allows creation and management of APIs securely across clouds. Section 5.3, “API Lifecycle: IBM API Connect” on page 147 discusses in detail the differences between DataPower and IBM API Connect and the benefits that a broader API management solution brings.
- ▶ **Deep integration:** DataPower provides configuration led easy path for integrating with various Systems of Record via out-of-the-box adapters for a selection of technology protocols, easy transformation and routing rules. However, DataPower it is not a general purpose integration capability. IBM App Connect is the premier integration solution covering the broadest range of protocols and mediation capabilities. In the next section, we explore IBM App Connect in more detail. There are also capabilities such as IBM z/OS

Connect that make mainframe data and functions available over REST HTTP APIs. These can then be future exposed by an API gateway such as IBM API Connect, for example in order to add API management capabilities. For more information on the options for connecting to mainframes, see the IBM Redpaper *IBM Z Integration Guide for Hybrid Cloud and the API Economy*, REDP5319 (<http://www.redbooks.ibm.com/abstracts/redp5319.html?Open>).

5.5 Application integration: IBM App Connect

IBM Cloud Pak for Integration includes a market-leading application integration capability. It enables the implementation of API and event-driven integrations and provides extensive adaptation to on-premises and cloud-based applications. It provides tooling that is optimized to the users' skillsets, so that they can be productive in a matter of hours and achieve real results in days. Powerful underlying capabilities facilitate the implementation of even the most complex integration patterns. As a result, data can be moved quickly, accurately and robustly. The Application Integration capabilities are also sold separately as *IBM App Connect* (and previously *Integration Bus*).

IBM Cloud Pak for Integration delivers a range of application integration capabilities and features including these:

- ▶ The ability to build and expose APIs through a no-code approach that can be easily managed through an API Lifecycle (discussed in 5.3, “API Lifecycle: IBM API Connect” on page 147)
- ▶ Extended connectivity across Cloud Services, Software as a Service (SaaS), cloud platforms, and on-premises applications
- ▶ Lightweight integration runtimes for cloud native and container-based deployment
- ▶ Deployment options that can enable clients to achieve a balance between control, management overhead, and budget
- ▶ New, simple tooling for all styles of user that works together to expose and integrate enterprise systems

5.5.1 User-aligned integration tooling

The Application Integration features include an upgraded desktop user interface that is coupled with new award-winning browser-based tooling. Together, they bring together the teams that own and manage the data with those that have the context to apply it. Digital businesses rely on data that is delivered in the right context, to the right client touch point, at the required time. The tooling provides frictionless access to enterprise data at the front line of the clients' business, whether those individuals are in IT or citizen integrators in the line of business.

It provides new integrated tooling experiences for the spectrum of users across the digital enterprise:

- ▶ For the core IT teams that manage the key systems and packaged applications, there is a rich tooling experience to support all styles of interaction, powerful mapping, parsing, and transformation. A broad range of functions — including built-in unit testing and the ability to perform pre-deploy validation, alongside linked browser-based tooling for the line-of-business teams — ensures that both developers and non-technical users can rapidly build integration without the need for code.

- ▶ Knowledge workers and citizen integrators in lines of business can take advantage of the simpler, configuration-based *designer* tooling to connect applications in the cloud. Alternatively, they can innovate on-premises applications for themselves to automate information and process flows by using a no-code approach while they take advantage of the multi-tenant, Cloud Service runtime.
- ▶ Integration specialists can choose to use the new web-based tooling to build simple things quickly, or use the full Integrated Development Environment (IDE) toolkit to tackle more detailed and challenging requirements.

Users of all experience benefit from accelerators, such as templates for common integration and industry-specific-use cases.

5.5.2 No-code RESTful integration services

Cloud Pak for Integration supports building integration flows through a no-code approach and exposes those flows as RESTful APIs without having to be an API development expert. These APIs can be seamlessly brought into the API Manager to handle the management, securing, and socializing to development teams. Whether building cloud scale, resilient applications from the ground up or taking the opportunity to use existing technologies as part of a serverless architecture, enterprises can rely on the rapid build and access to the systems at the right time.

5.5.3 Flexible integration patterns

There is broad support in a single platform for API and event-driven technologies to complement integrations that support batch data movement, Electronic Data Interchange (EDI), and service-oriented architecture (SOA). This helps to ensure that existing investments in integration can be used where they are optimal, while they support the rapid build of new use cases.

5.5.4 Broad deployment options

Two form factors — software and a managed cloud service — provide an extensive range of deployment options that include on-premises, to private data centers, dedicated instances on public cloud, on Red Hat OpenShift, through to the public cloud providers, including of course our own IBM Cloud. It also includes a fully managed IBM Cloud service. The runtime continues to serve virtual machine and bare metal installation while also being significantly optimized for running in containers.

5.5.5 Extended connectivity

Connectivity options across cloud service applications, cloud platforms, and existing on-premises applications provide pre-packaged connectivity to a wide range of Cloud services, Software as a Service (SaaS) applications (100+), and cloud platforms. These options complement the robust existing set of connectors for packaged applications that include:

- ▶ Customer relationship management (CRM)
- ▶ Enterprise resource planning (ERP)
- ▶ Marketing and Human Capital Management (HCM)
- ▶ Files

- ▶ Databases
- ▶ Messaging systems
- ▶ Mainframe applications¹

5.5.6 Situational awareness with insightful and actionable notifications

This new capability empowers knowledge workers to easily consume data in enterprise systems and cloud services, and then proactively detects business situations of interest to remove information blind spots. Key to the ability for any enterprise to act immediately, non-technical users can quickly and easily build flows to detect events of interest and provide notifications that communicate the key pieces of information that are required for them to make an informed decision. Based on that insight, users can then quickly select the right, next-best-action that should be carried out.

5.5.7 Quick utilization of artificial intelligence (AI) services

The cloud-based tooling enables enrichment of data in flight and the implementation of digital agents by using pre-built connectors to IBM Watson's cognitive services. AI capabilities can be trivially embedded into clients' integrations to perform sentiment analysis, translation, or to pull out key aspects of a document narrative by using rich cognitive and natural language services. It can also integrate directly with conversation services to quickly build out chat bots.

5.5.8 Rapid visual orchestration of data and systems for API-driven architectures

The strength of an API relies heavily on how well it is composed to bring existing sources of information together. The offering provides a range of relatable tooling experiences that support users of multiple types and skills sets to perform that composition, with everything from simple, graphical flow design right through to deep grammatical controls. Users, whether business-focused or technical, can create and expose innovative APIs from wherever data it is located.

5.5.9 Lightweight integration runtime for cloud native deployment

IBM Cloud Pak for Integration delivers revolutionary changes to the way that application integration assets are deployed to the integration runtime. This massively shortens and simplifies build pipelines and offers rapid deployment of new artifacts, fast start-up times and elastic scaling, and availability configurations. The result is to create a runtime that is more CPU and memory efficient, truly cloud-native and aligns with the principles of microservices. As such, large centralized ESB installations often containing 100s of integrations can be broken down into small granular sets of micro integrations each running in their own individually managed and scaled containers.

Users who deploy the software can take advantage of these features:

- ▶ Simple file-system-based, dependency-free installation, and deployment that is ideally suited to Docker images.

¹ For more information on the options connecting to mainframe, see the IBM Redpaper *IBM Z Integration Guide for Hybrid Cloud and the API Economy*, REDP5319 (<http://www.redbooks.ibm.com/abstracts/redp5319.html?open>)

- ▶ Easy scaling and management by using orchestration frameworks such as Kubernetes, alongside other components within a modern architecture.
- ▶ Ensures consistency of fixed integration server settings between environments, yet enables simple overriding of settings that should vary by environment.
- ▶ Truly, cloud-native components that are tailored for use in container technologies, such as Docker that run under a Kubernetes framework.

5.5.10 Grown from a trusted market leading product

IBM App Connect builds upon the robust and proven IIB runtime that is trusted by thousands of clients over the past 18 years or more to run their mission-critical, application integration projects. During this period, the offering continually grew to allow clients to embrace new technologies — such as Kafka and Loopback bridge across cloud and on-premises architectures with a hybrid runtime — and adopt open standards through the delivery of OpenAPI features.

The connectivity and tooling options extend the wide variety of data formats and application that are supported and include standards-based formats, such as:

- ▶ eXtensible Markup Language (XML)
- ▶ Data Format Description Language (DFDL)
- ▶ JavaScript Object Notation (JSON)
- ▶ Industry formats and standards, such as:
 - Health Level 7 (HL7)
 - The Society for Worldwide Interbank Financial Telecommunication (SWIFT)
 - ISO8583
 - Custom formats

An extensive range of operations (there are around 100 functions on the palate of the toolkit) can be performed on data, such as routing, filtering, and enrichment.

Wherever the applications are on-premises, on cloud, or both, these flexible integration capabilities can support the users' choice of solution architectures, which include:

- ▶ Service-oriented
- ▶ RESTful
- ▶ Event-oriented
- ▶ Data-driven
- ▶ File-based (batch or real-time)

The new capabilities that are delivered in IBM App Connect unify and extend the capabilities of the IIB family with those of IBM App Connect Professional in a single offering. These capabilities better serve the demands of clients who are integrating applications and data to compete in today's economy. IBM App Connect is the official successor to the IIB family of offerings.

5.5.11 IBM App Connect on deployment options

This sections outlines the platforms on which IBM App Connect is available.

- ▶ **Container-base installation:** IBM App Connect's integration server (formally known as IBM Integration Bus) has been supported on containers since 2015.

- **IBM Cloud Pak for Integration:** The simplest way to deploy IBM App Connect's integration servers, is to use the facilities provided by IBM Cloud Pak for Integration. Through a consistent user interface shared with the other capabilities within the Cloud Pak, users can directly upload an integration definition (a "bar" file) for deployment. IBM Cloud Pak for Integration then takes care of how to build that into a container image, deploy it, and also manage and monitor it.
- **Red Hat OpenShift and other container platforms:** IBM App Connect's integration server is supported for use in any container platform that runs OCI (Open Container Initiative) based containers. Template container images, Dockerfiles, and Helm Charts are provided to simplify setup as documented in Chapter 7, "Field notes on modernization for application integration" on page 433.
- ▶ **Operating system installation:** IBM App Connect V 11 can be installed directly on the following operating systems:
 - Windows
 - Linux
 - IBM AIX® – IBM App Connect 11.0.0.5 and later
- ▶ **Managed service on IBM Cloud:** IBM App Connect on IBM Cloud is a fully managed integration platform on IBM Cloud with a broad range of capabilities to connect different applications. It provides "enterprise - wide" connectivity options for deep integration needs.

Clients can use the IBM App Connect Toolkit to build integration assets that are packaged and deployed to an Integration Server.

The IBM App Connect Toolkit can be obtained by downloading the Developer Edition of IBM App Connect, which is included with the Cloud Service and allows development of integrations.

5.6 Enterprise Messaging: IBM MQ

As a key part of IBM Cloud Pak for Integration, IBM MQ is the gold standard for enterprise messaging. It makes life easier for developers by supporting multiple operating systems, hybrid cloud environments, agile development processes, and microservices architectures with an all-in-one messaging backbone. The technology has been proven across industry use cases for over 26 years and used by 85% of the Fortune 100².

It enables applications and services to communicate reliably without calling each other directly, introduces process independence into the application architecture, and improves fault tolerance and reliability throughout the system. IBM's enterprise messaging enhances connectivity, simplifies resilience, and is perfectly suited for distributed and diverse computational architectures. Unlike most messaging solutions, IBM MQ can be used to ensure once-and-once-only delivery. Others offer at most once or at least only, introducing the potential for message loss or duplication.

IBM has provided enterprise messaging in containers since 2015 and certifies container technologies such as Kubernetes and OpenShift to allow managed production grade deployments. Its lightweight nature makes it a natural fit for cloud native environments, allowing runtimes to start up in seconds. High availability is provided as standard, cloud-native features such as Uniform Clusters make scaling of environments easy. As clients expand from their traditional on-premises environments to a hybrid multicloud deployment,

² <https://www.ibm.com/products/mq>

IBM MQ environments are expanded to provide the mission-critical communication across this hybrid multicloud environment.

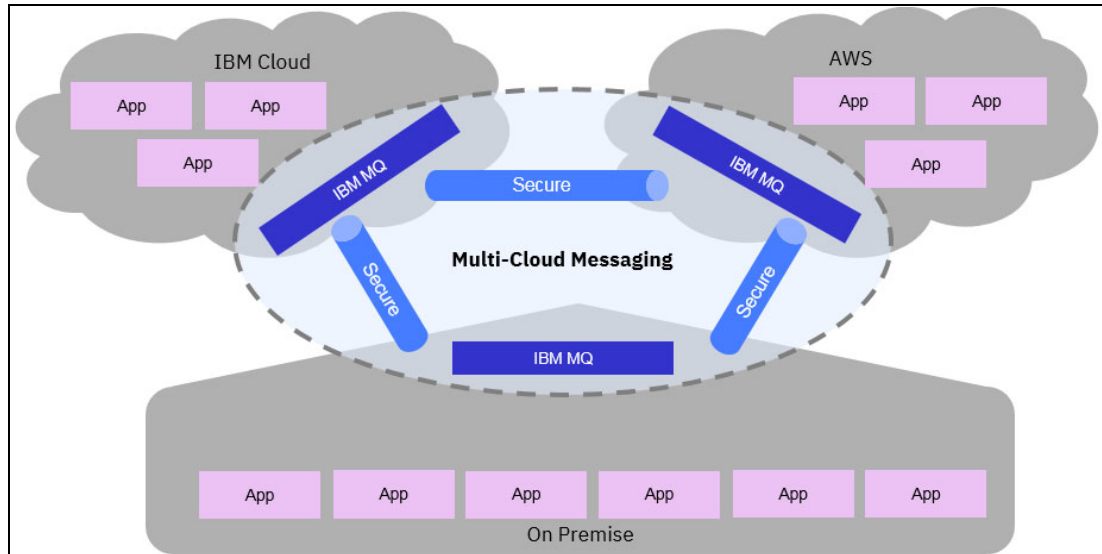


Figure 5-3 IBM MQ Multi-Cloud for mission-critical communication

Messaging is traditionally administered by dedicated teams within IT. “As-a-service” delivery lets enterprises create self-service portals that enable line of business (LOB) or individual users to request changes to the messaging infrastructure independently. For example, they can create or delete queues or provision new resources for applications.

Implementing MQ as a service can enable enterprises to increase agility and efficiency, since resource provisioning can be accomplished faster and more dynamically. It can also improve usability and internal consistency. Messaging middleware naturally works well within serverless or microservices architectures common in cloud-native development. It’s also being offered in a cloud-hosted service model. In this model, your MQ handles all the provisioning, installation, and maintenance of your messaging infrastructure, and is hosted in the cloud (for example IBM and AWS clouds).

To discover more about mastering IBM MQ, go to the Learn MQ site here:
<https://developer.ibm.com/messaging/learn-mq/>

5.7 Event Streaming: IBM Event Streams

IBM Cloud Pak for Integration includes Apache Kafka as the event streaming capability. IBM builds on the open source technology and enhances the ease of use to be truly ready for enterprise deployments. This is provided on public and private cloud environments to meet the needs of clients. IBM was the first supplier of a managed Apache Kafka offering in 2015 on the IBM Cloud. IBM Event Streams on IBM Cloud provides both multi-tenant and single tenant options, and this experience of running a managed service has been rolled into our on-premises private cloud offering. This capability is also sold separately as IBM Event Streams. In either model, it can run on-premise as a fully containerized offering, allowing the deployment to benefit from cloud-native best practices such as easy installation, management, and scaling of the solution. Out-of-the-box high-availability is pre-configured, so you can replicate streams across the globe for disaster recovery, thereby satisfying the requirements for mission-critical use.

Apache Kafka scales to handle millions of messages a second, suitable for any organizations needs. However, deploying a production environment can be daunting with the configuration of Kafka brokers, zoo keepers, administration agents, and so forth. IBM Event Streams has been purpose built for simplicity, with the initial production setup only a few clicks to complete. As with all supported IBM products, IBM Event Streams also provides IBM 24x7 support to provide the safety net organizations need.

To learn more about getting started with IBM Event Streams, go to *Getting Started Guide on IBM Cloud Docs* at <https://cloud.ibm.com/docs/services/EventStreams/index.html>.

5.8 High-Speed File Transfer: IBM Aspera

The high-speed file transfer capability delivers time-critical transport of very large files and data sets — such as high-definition broadcast videos and high-quality advertising footage — to many global endpoints. It offers highly scalable server software that supports thousands of concurrent transfer sessions and multiple client options for initiating high-speed transfers including the web browser plug-in, desktop and point-to-point clients and command line. This capability is also sold separately as IBM Aspera.

This file transfer software enables centralized control over network-wide transfers, nodes, and users with complete visibility into the high-speed transfer environment. It provides a dashboard overview of all activity and multilevel views of the performance of individual transfers and node activity. In addition, it lets users create automated one-time or recurring transfers including multipoint *smart transfers* that can later be copied, modified, and reused.

With direct-to-cloud technology and built-in clustering, Aspera can provide faster performance and automatically scale transfer capacity. Enterprise-grade security includes a powerful access control model, encryption in transit and at rest, and data-integrity verification.

Key features include:

- ▶ **High-speed access to data in a hybrid cloud:** View and transfer files across almost any cloud or on-premises storage system with a single interface.
- ▶ **Fast, reliable and secure file sharing and exchange:** Drag and drop large files and folders to send or share data with any authorized individual or group of users.
- ▶ **Package delivery to anyone around the world:** Deliver packages of files to recipients anywhere. Include external users in package delivery and content-submission requests.
- ▶ **Data migration to, from and between clouds:** Migrate massive volumes of unstructured data at high speed across all market-leading cloud platforms.
- ▶ **High-speed transfer-enabled applications:** Use published APIs to extend existing web and mobile applications to include fast file transfer.
- ▶ **Customized reports and auditing:** Gives users and administrators control over individual transfer rates and bandwidth sharing, and full visibility into bandwidth utilization. It maintains comprehensive logging for customized reports and auditing and allows administrators to monitor and control bandwidth utilization. The adaptive rate-control feature enables fast, automatic discovery of bandwidth capacity and its full utilization, while remaining fair to other traffic.

5.8.1 Fast, Adaptive and Secure Protocol (FASP) technology

Built on the patented IBM Aspera FASP® technology, Aspera can move data up to hundreds of times faster than FTP and HTTP, regardless of file size, transfer distance, or network

conditions. It allows you to fully utilize your available bandwidth, without impacting other traffic on the network.

This file transfer software offers a breakthrough transfer protocol that leverages an existing WAN infrastructure and commodity hardware. It achieves speeds up to hundreds of times faster than FTP and HTTP and eliminates the fundamental bottleneck of conventional file transfer technologies.

A detailed description of the IBM Aspera FASP technology can be found at the following link:

<https://www.ibm.com/downloads/cas/7D3KBL9Z>

5.8.2 Aspera on Cloud

Aspera on Cloud is IBM Aspera's on-demand SaaS offering for global content transfer and exchange.

Using Aspera on Cloud, organizations can store and readily access files and folders in multiple cloud-based and on-premises storage systems. Sharing among users is as easy as browsing or dragging-and-dropping, regardless of where the files are located, freeing collaboration from traditional boundaries among colleagues in both local and remote locations.

Aspera on Cloud also uses IBM Aspera's FASP protocol, which overcomes the limitations of other file-transfer technologies. By moving large data sets at maximum speed, reliably and securely — regardless of network conditions, physical distance between sites, and file size, type, or number — Aspera on Cloud enables a new world of collaboration, sharing, and content delivery.

Aspera on Cloud highlights include the following:

- ▶ Domain security within the application
- ▶ Intuitive file management tools
- ▶ Multiple permission or access levels for individual files and folders, configurable per user or group
- ▶ Ability to send content to multiple recipients by using an intuitive, email-like interface
- ▶ Simple sharing of files and folders
- ▶ Configurable permissions to share with and send to outside users who do not have an Aspera on Cloud account
- ▶ Comprehensive administrative console
- ▶ Support for IBM Aspera Transfer Service, enabling automated self-scaling of transfer capacity to maximize performance while minimizing cost
- ▶ RESTful Platform API (including users, permissions, transfers, and reporting)

Intuitive file sharing and delivery

Move files across on-premises and multicloud environments with an easy-to-use interface that simplifies file uploads, downloads, sharing, and distribution. Easily drag and drop files and folders to transfer to any storage location. Assemble files into a digital package, and use an email-like interface to send to one or more recipients. Organize your files and users into secure collaboration workspaces. Enable users to submit data to a shared content-submission portal. See Figure 5-4 on page 163.

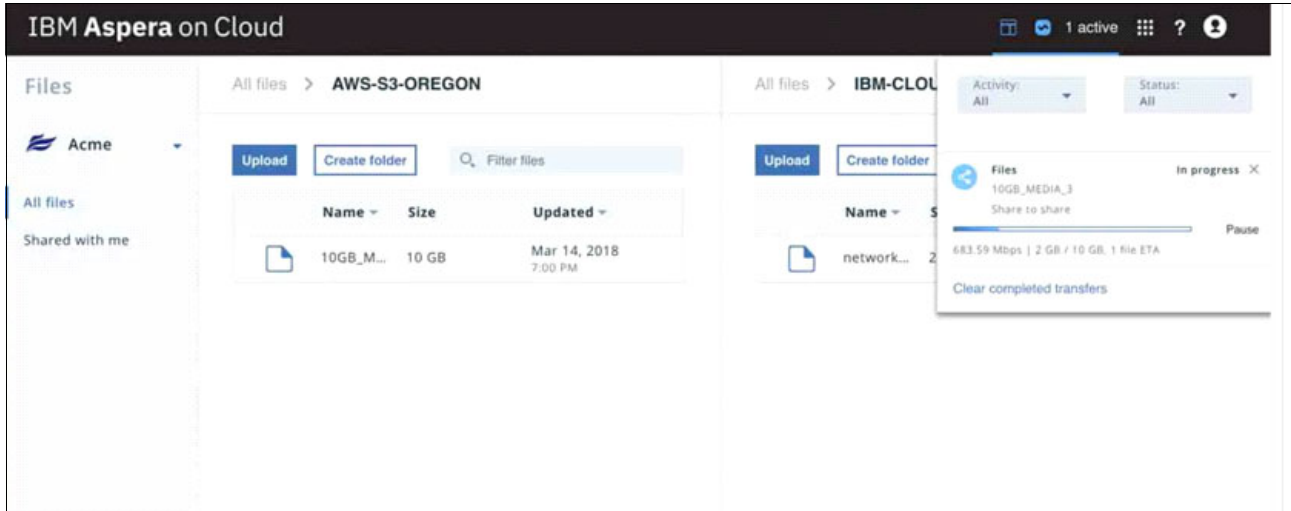


Figure 5-4 Intuitive file sharing and delivery

Central administration of hybrid environments

Connect to all your cloud and on-premises storage and remotely manage your transfer nodes with a single easy-to-use interface. Store and readily access files and folders in multiple cloud-based and on-premises storage systems. Configure access to transfer nodes that are located in your data center or on any of the following market-leading cloud platforms: IBM Cloud, Amazon Web Services (AWS), Azure and Google. Establish network policies that govern how transfer nodes interact with each other. See Figure 5-5.



Figure 5-5 Central administration of hybrid environments

Real-time visibility and control of transfers

Monitor transfer activities in real time, while embedding your brand into every communication and web asset. Manage transfer activities, storage usage and digital packages in real time. Monitor activity logs and service alerts. Manage membership in workspaces, user groups and

shared inboxes. Easily create a uniquely branded web presence by customizing email templates and logos to match your brand identity. See Figure 5-6.

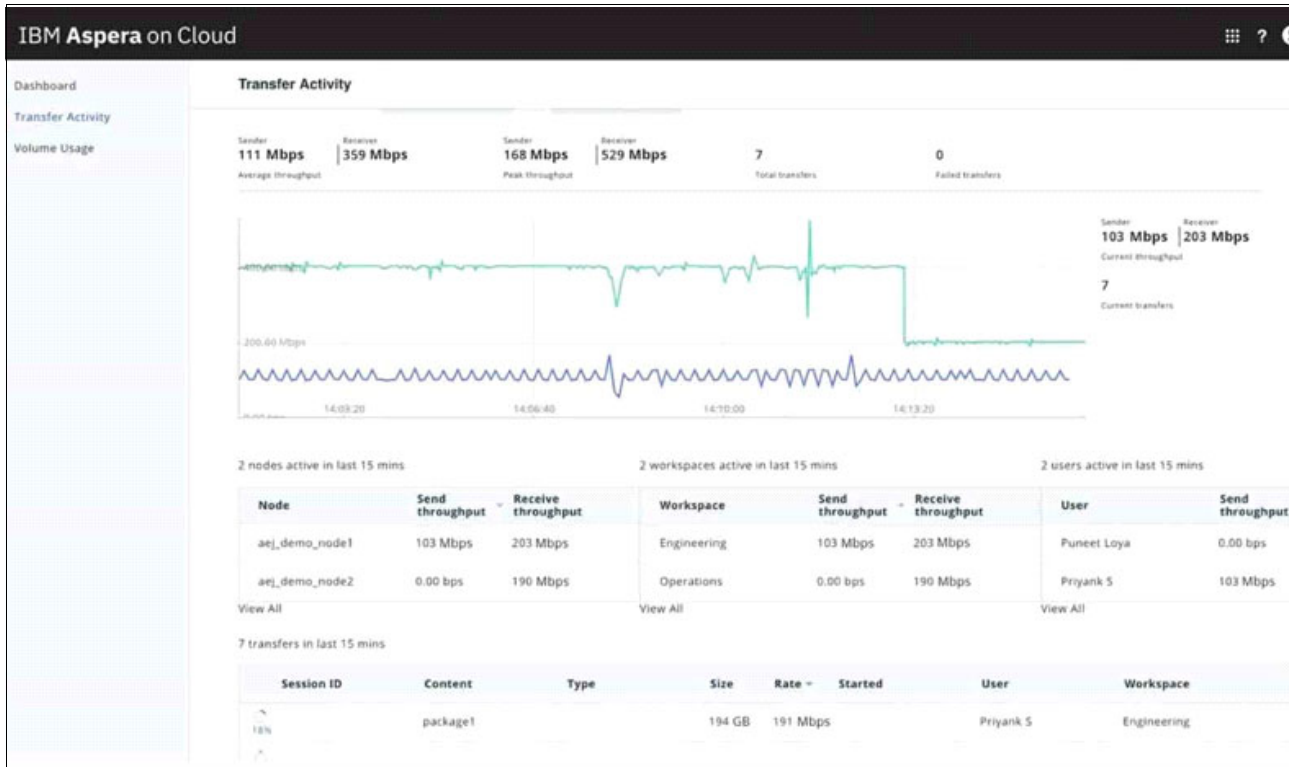


Figure 5-6 Real-time visibility and control of transfers

5.9 Service Mesh: Istio

The IBM Cloud Pak for Integration and Red Hat OpenShift Container Platform provide a Service Mesh based on Istio. At a high level, service mesh can appear to have some overlap points with other integration components. So, we have chosen to briefly describe it here for clarity. See also 4.8, “Service mesh” on page 119, where we discuss its role conceptually and especially in relation to API Management.

Istio (<https://istio.io>) is one of the most popular technology implementations for a service mesh. Istio is a Kubernetes-compatible open platform for providing a uniform way to integrate microservices, manage traffic flow across microservices, enforce policies and aggregate telemetry data.

From a high-level functionality perspective, Istio allows you to connect, secure, control, and observe your containers as they run on Kubernetes.

In this section, we go specifically into detail on the Istio service mesh. For more information on the broader architectural role of the Service Mesh, and specifically its comparative relationship to API management, see 4.8, “Service mesh” on page 119.

Connect

Istio provides traffic management for services. The traffic management function includes:

- **Intelligent routing:** The ability to perform traffic splitting and traffic steering over multiple versions of the service

- ▶ **Resiliency:** The capability to increase micro services application performance and fault tolerance by performing resiliency tests, error and fault isolation and failed service ejection.

Secure

Istio implements a Role-based Access Control (RBAC) which allow a specific determination on which service can connect to which other services. Istio uses Secure Production Identity Framework for Everyone (SPIFFE) to identify the ServiceAccount of a micro service uniquely and use that to make sure that communications are allowed.

Control

Istio provides a set of policies that allows control to be enforced based on data collected.

Observe

While enforcing policies, Istio allows observing your microservices through integrated telemetry, monitoring, tracing, and logging.

An Istio service mesh is logically split into a data plane and a control plane.

- ▶ The *data plane* is composed of a set of intelligent proxies (*Envoy*) deployed as sidecars. These proxies mediate and control all network communication between microservices along with *Mixer*, a general-purpose policy and telemetry hub.
- ▶ The *control plane* manages and configures the proxies to route traffic. Additionally, the control plane configures Mixers to enforce policies and collect telemetry.

Figure 5-7 on page 166 shows the Istio service mesh architecture.

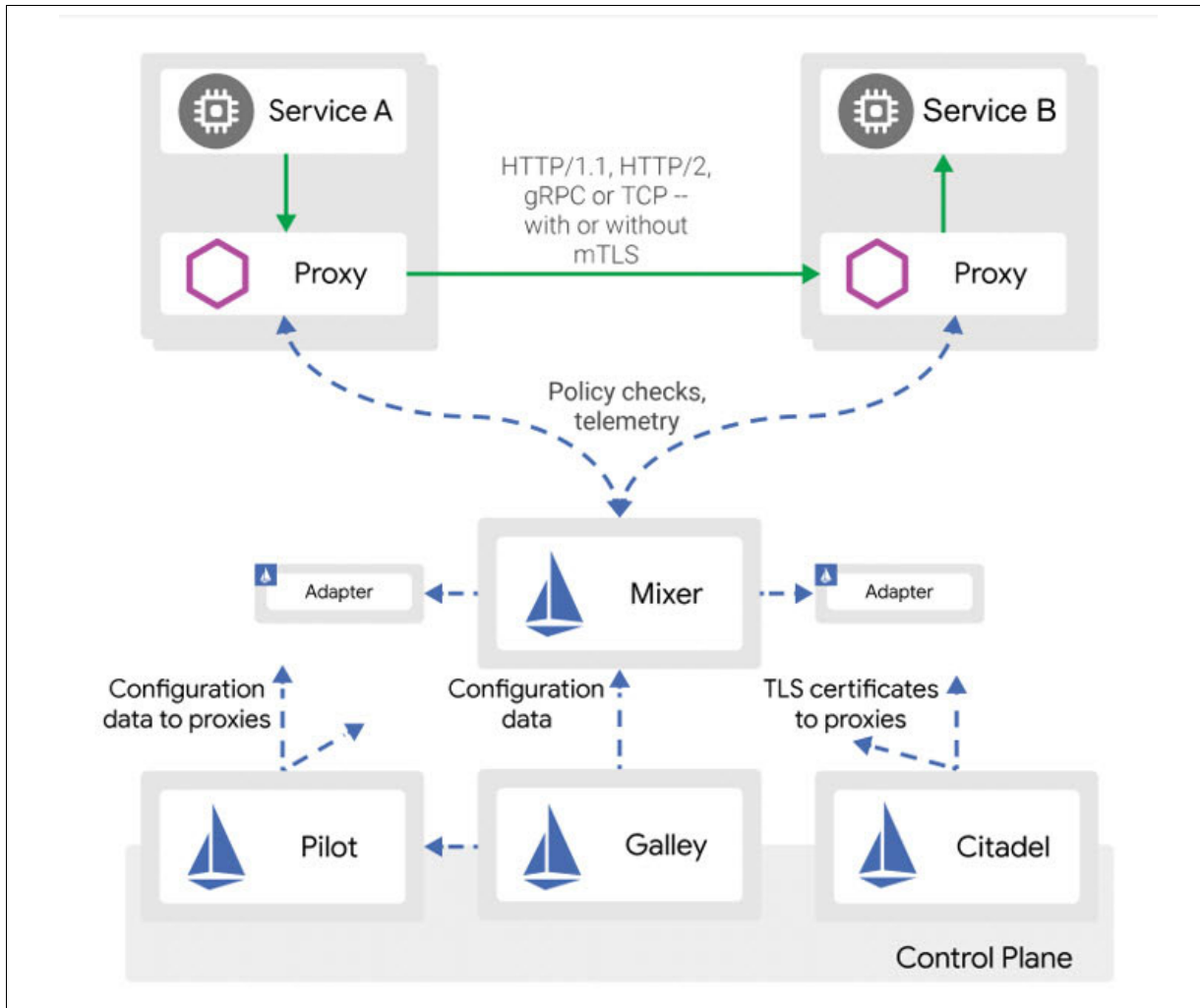


Figure 5-7 Istio service mesh architecture

The Istio control and data plane are made up by the following components:

Envoy: sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services. The proxies form a secure microservice mesh providing a set of functions like discovery, layer-7 routing, circuit breakers, policy enforcement and telemetry recording/reporting functions.

Mixer: central component that is leveraged by the proxies and microservices to enforce policies such as authorization, rate limits, quotas, authentication, request tracing and telemetry collection.

Pilot: a component responsible for configuring the proxies at run time.

Citadel: a centralized component responsible for certificate issuance and rotation. Citadel also deploys node agents responsible for certificate issuance and rotation.

Galley: central component for validating, ingesting, aggregating, transforming, and distributing config within Istio.

In the remainder of this section, a more detailed description the Istio components is provided, (based on the community definition available here:

(<https://istio.io/docs/concepts/what-is-istio/>). That way, the reader can understand how each component contributes to Istio's ability to connect, secure, control, and observe microservices.

Envoy

Istio uses an extended version of the Envoy proxy. Envoy is a high-performance proxy that is developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. Istio leverages Envoy's many built-in features, for example:

- ▶ Dynamic service discovery
- ▶ Load balancing
- ▶ TLS termination
- ▶ HTTP/2 and gRPC proxies
- ▶ Circuit breakers
- ▶ Health checks
- ▶ Staged rollouts with percentage-based traffic split
- ▶ Fault injection
- ▶ Rich metrics

Envoy is deployed as a sidecar container alongside your application container. It is deployed in the same Kubernetes pod such that it is always present and shares the same lifecycle. This deployment allows Istio to extract a wealth of signals about traffic behavior as attributes. Istio can, in turn, use these attributes in Mixer to enforce policy decisions, and send them to monitoring systems to provide information about the behavior of the entire mesh.

The sidecar proxy model also allows the addition of Istio capabilities to an existing deployment with no need to rearchitect or rewrite code.

Mixer

Mixer enforces access control and usage policies across the service mesh, and collects telemetry data from the Envoy proxy and other services. The proxy extracts request level attributes, and sends them to Mixer for evaluation.

Mixer includes a flexible plug-in model. This model enables Istio to interface with a variety of host environments and infrastructure backends. Thus, Istio abstracts the Envoy proxy and Istio-managed services from these details.

Pilot

Pilot provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing (like A/B tests, canary rollouts), and resiliency (timeouts, retries, circuit breakers).

Pilot converts high level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at run time. Pilot abstracts platform-specific service discovery mechanisms and synthesizes them into a standard format that any sidecar can use, as long as that sidecar conforms with the Envoy data plane APIs.

Citadel

Citadel provides strong service-to-service and end-user authentication with built-in identity and credential management. Citadel can be used to upgrade unencrypted traffic in the service mesh. Using Citadel, operators can enforce policies based on service identity rather than on network controls. Istio's authorization feature also allows you to control who can access services within the mesh.

Galley

Galley validates user-authored Istio API configuration on behalf of the other Istio control plane components. Over time, Galley will take over responsibility as the top-level configuration ingestion, processing, and distribution component of Istio. It will be responsible for insulating the rest of the Istio components from the details of obtaining user configuration from the underlying platform (for example Kubernetes).



Practical agile integration

In this chapter we progressively build up a scenario that demonstrates use of the integration capabilities to solve some common modern application design challenges using agile integration techniques.

We begin by exposing data from a traditional data source over RESTful APIs. Then, we gradually build up to more sophisticated ways of making that same data, and data from other applications, available via modern microservice-style components.

We have arranged each section to be completely independent of the others, so there is no need to work through the sections in order. Each section is self-sufficient, and provides everything needed to build it out.

Note: The solution in each section is complete for a specific scenario. However, none of these solutions forms a global, end-to-end solution for agile integration.

This chapter has the following sections. See 6.1, “Introduction” on page 170 for a detailed description of each section:

- ▶ Introduction
- ▶ Application Integration to front a datastore with a basic API
- ▶ Expose an API using API Management
- ▶ Messaging for reliable asynchronous data update commands
- ▶ Consolidate the new IBM MQ based command pattern into the API
- ▶ Advanced API security
- ▶ Create event stream from messaging
- ▶ Perform event-driven SaaS integration
- ▶ Implementing a simple hybrid API
- ▶ Implement event sourced APIs
- ▶ REST and GraphQL based APIs
- ▶ API testing
- ▶ Large file movement using the claim check pattern

6.1 Introduction

The scenarios for this chapter are based on a common theme we see with customers. We often begin with a requirement from the business to provide access to a back-end system by fronting it with an API. However, this apparently simple requirement grows in complexity over time, typically to cater to increasingly challenging non-functional requirements such as performance and availability.

Our scenario begins with the basic requirement to make the data from a table in a traditional database available as an API. In our scenario this is a simple single table that holds information about "Products." But in a real scenario it could be multiple tables that are joined in various ways to provide product catalog information. We achieve this integration in its most basic form in section 6.2, "Application Integration to front a data store with a basic API" on page 173. Deliberately, we do this using the fine-grained cloud native style deployment that is fundamental to agile integration as discussed earlier in this book. This ensures that integrations are isolated from one another so they can be changed and scaled independently. This improves agility, resilience, and optimization of the underlying resources.

As shown in Figure 6-1, we then look at improving the exposure of the API, using API management to make the API more discoverable, enable consumers to self-subscribe to use APIs, and enable us to track and control usage of the APIs. This is the topic of 6.3, "Expose an API using API Management" on page 190.

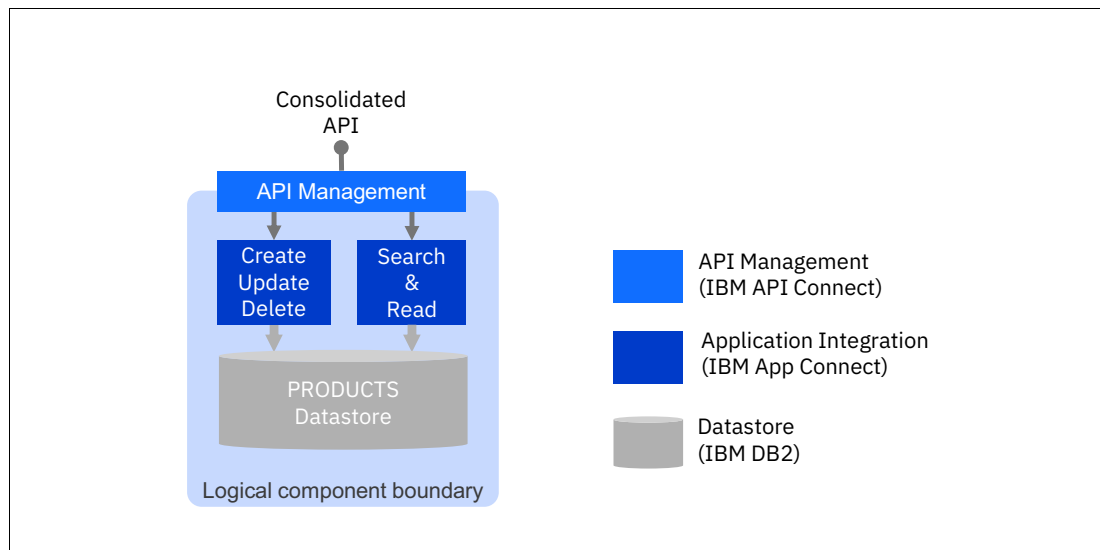


Figure 6-1 Improving the exposure of the API using API management

As a result, we now have a self-contained business component that provides API-based access to product data, that can easily be reused to bring that data into new solutions.

Next comes the non-functional requirements. Although our integrations have been designed in a cloud native way such that they can scale, and indeed scale independently, the back-end database is still a bottleneck. There are times when the number of updates being made to the Product table is affecting the performance of the reads on that table. Indeed, the writes themselves start to take more time. The effect on the user experience of applications based on this API is becoming noticeable, reducing customer satisfaction. With our single-table scenario, clearly these performance issues would be unlikely to occur. But we can imagine a real multi-table scenario — with searches performing multi table joins, and updates locking

multiple tables in order to perform transactional updates with integrity — where performance issues of this type soon become an issue. We decide to tackle this problem on two fronts.

► **Change the interaction pattern for updates to be asynchronous:**

We change the way updates are performed such that they are done asynchronously after the requests has been acknowledged. Consumers of the API then get an immediate response to assure them that the updates will occur, enabling a much more responsive user interface. Furthermore, it means we can now throttle the rate at which we apply those updates to the database such that they have less effect on the performance of reads. We enable this in 6.4, “Messaging for reliable asynchronous data update commands” on page 209. We provide a route to performing updates asynchronously in a “fire-and-forget” pattern by placing them in a “command store” (in our case, IBM MQ), then responding immediately back to the caller with an acknowledgment. We then make this accessible to a broader audience in 6.5, “Consolidate the new IBM MQ based command pattern into the API” on page 259 by bringing the asynchronous update back into the HTTP based API. Effectively we hide the use of IBM MQ behind the scenes. It should be noted that we have now introduced *eventual consistency* (rather than *immediate consistency*). Updates don't occur at the time of the update request. The applications using the API need to take this into account in their design, but as a result they can enjoy the performance improvements.

► **Provide a read-optimized datastore:**

Now that Product data is so easily available through our API, it is being used in many new and innovative ways. Unfortunately, the way that the data is stored in the current database is poorly suited to the types of queries now being performed. These new queries perform slowly even with our move to asynchronous updates as the issues are more related to how data is aggregated. This may simply be because the new consumers want a very different representation of the Product data. Or it could be because it needs to be combined with other data such as Price before it is useful. To solve this, we decide to implement a new datastore, that is specifically optimized for these new queries. We of course need to keep this new datastore in sync with the original master Product database. To do this, in 6.7, “Create event stream from messaging” on page 322 we show how we can keep a note of all the changes that happen to the Product database and place them in an event store, which in our case is provided by IBM Event Streams. These updates can then be asynchronously applied to the read-optimized datastore as discussed in 6.10, “Implement event sourced APIs” on page 367. Note this further exacerbates the *eventual consistency* between the updates to the main database, and reads from the read-optimized datastore. However, since users of our API have already had to learn to code for this behavior when we separated out the commands, it should have minimal effect on their applications.

Figure 6-2 on page 172 shows the enhanced integration pattern with asynchronous updates and optimized reads.

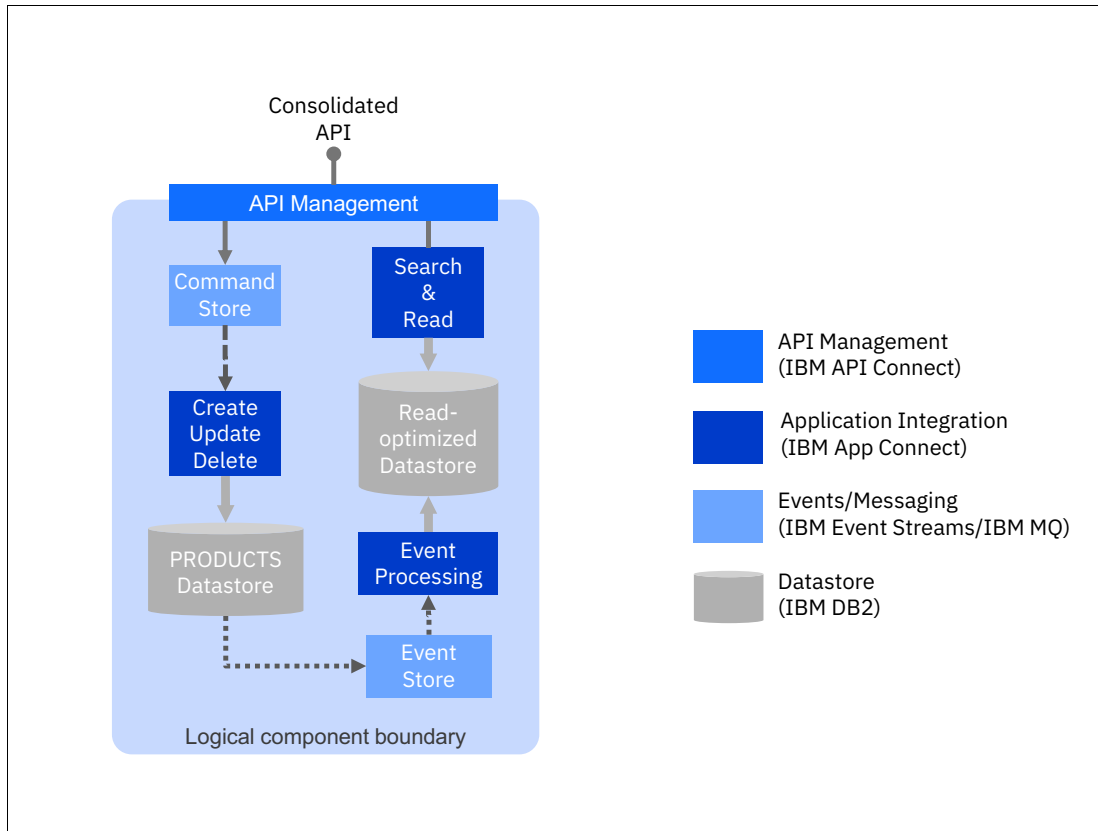


Figure 6-2 Performance and availability improvements through eventual consistency patterns

From an integration standpoint, the patterns we've introduced are many decades old. Yet, modern application developers might see this set of patterns implemented together and recognize that it is an implementation of CQRS. CQRS stands for Command Query Responsibility Segregation. It essentially means providing independent paths for commands (create, update, delete) and queries (reads, searches, and so on), just as we have done in our scenario. Because this concept is familiar to many developers, we have tried to use the associated terms in our scenario where appropriate.

From the outside, our logical Product component looks much the same as it did in our first iteration, enabling access to Product data via an API. However, using agile integration techniques:

- ▶ The API is now more easily discovered, used, and controlled.
- ▶ The implementation is more performant and scalable.
- ▶ The patterns give us flexibility to rapidly implement new requirements without destabilizing what we have.

So, we have had a detailed look at how integration capabilities can be involved in the implementation of a reusable business component, from a simple API exposure through to a full CQRS-based implementation. Next, we look at how these business components might interact with one another.

In 6.9, "Implementing a simple hybrid API" on page 345, we look at our new API from the consumers point of view. We consider how much easier exposing something as an API makes the creation of new solutions. In this case we enable a non-integration specialist to build a new API based on existing APIs in order to create a further unique capability.

Going back to the events that we were using internally within the component, let's consider how they might also be valuable outside the component. The events might become a reusable capability in the same way that our API is. Exposing events beyond the component would enable, for example, a separate application to use the same *event-sourced* programming models in their own implementations, maintaining their own read-optimized data stores. In 6.8, "Perform event-driven SaaS integration" on page 328, we extend this thought. We consider an example of how non-integration specialists could use events from our component as triggers on new integrations with modern Software-as-a-Service applications.

We then return to our exposed API and consider how we might want to improve that exposure as we make it available to broader audiences. We explore these issues:

- ▶ How to implement the OAuth security model to enable authentication to be handled by a separate provider ([6.6, "Advanced API security" on page 276).
- ▶ How to introduce alternative API exposure styles such as GraphQL to give consumers more flexibility in how they consume the data (6.11, "REST and GraphQL based APIs" on page 381).
- ▶ How to perform effective, repeatable API testing to ensure that the API behavior remains consistent as we add enhancements (6.12, "API testing" on page 398).

Finally, we consider what to do when the data we need to move between applications is not appropriate for APIs or events. A common example in modern applications is files such as video media. It makes no sense for these large files to travel over an API, or events due to their size, often greater than a Gigabyte. In these circumstances a more logical approach is the *claim check pattern* explored in 6.13, "Large file movement using the claim check pattern" on page 410. In that example, we store the object in a place that the cloud can reference, then pass only the reference. Of course, the file's content must move to its destination eventually. So, we also discuss the benefits of FASP (the Fast and Secure Protocol) for getting large data across significant network distances.

Note: For some of the following exercises, we use IBM Cloud Pak for Integration. If you do not currently have access to an environment, see section 5.1.4, "Getting access to IBM Cloud Pak for Integration for the exercises" on page 145.

6.2 Application Integration to front a data store with a basic API

Important: Since the writing of this IBM Redbooks publication, the IBM Cloud Pak for Integration has embraced Kubernetes Operators (<https://coreos.com/operators/>). This significantly simplifies how components such as an Integration Server are installed and maintained, extending the features provided by Helm. There is more information and an excellent video demonstrating this new capability here:

<https://developer.ibm.com/integration/blog/2020/06/28/ibm-app-connect-operator-1-0-is-now-available/>

It does unfortunately mean that some of the instructions describing the deployment of App Connect Enterprise in containers within this section are now out of date, and will need to be adapted to the use of operators. We may well look to update the book, but in the mean time, refer to the product documentation to find information on the new features.

In this section, we demonstrate how to use IBM App Connect to expose a datastore as a RESTful API. The key points implemented here are as follows:

- ▶ *Fine grained, container-based deployment of integrations*, enabling independent maintenance, elastic scaling, and isolated resilience.
- ▶ *Code free data mapping* from a REST data model to database table definition.
- ▶ *Configuration-based protocol conversion* from HTTP to JDBC.

The objective here is not to show in detail how to create the integrations themselves since there is plenty of existing material on building integrations. The key thing to note is that no code is required for these simple integrations, just a simple integration flow that contains a map.

Instead, we want to focus on what it looks like to deploy these to a cloud native style. As such, we begin the exercise with a blank cloud environment, and we deploy the integrations directly to it. There is no preparatory stage of building a shared infrastructure as we would have traditionally. Instead, each integration that is deployed provides its own discrete integration runtime, and the rest (such as HA and scaling) is provided by the container orchestration platform.

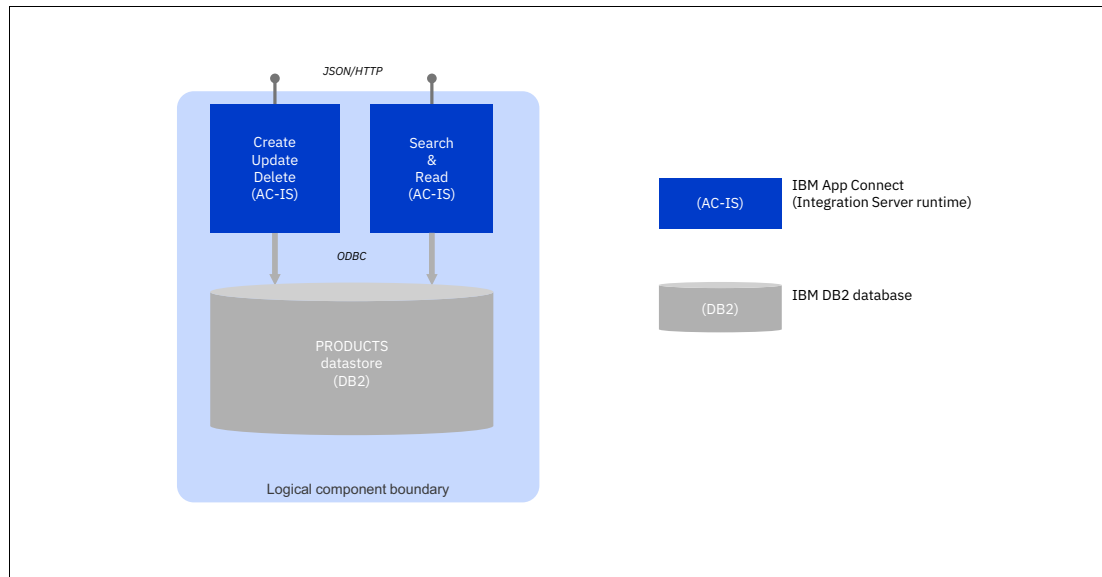


Figure 6-3 Deployment of the API across two separate containers

We deliberately demonstrate deployment of this API across two separate containers as shown in Figure 6-3. That way, you see that they could be changed and scaled separately and could have different resilience models.

We create two REST applications in the IBM App Connect Toolkit. One is for Create, Update, and Delete (commands) to a table on IBM Db2 called Products. The other deals exclusively with Read (queries). The flows for the two REST applications are already built and available at this GitHub site:

<https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/tree/master/chapter6/6.1>

The applications can be opened in the Toolkit by importing the following files:

- ▶ https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/6.1/database_operations_PI.zip

- ▶ https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/6.1/database_query_PI.zip

Alternatively, you can build them yourself as described here:

https://www.ibm.com/support/knowledgecenter/en/SSTTDS_11.0.0/com.ibm.gdm.doc/cm28851_.htm

Each REST application includes:

- ▶ JDBC Connection to a Db2 database that uses a policy
- ▶ A Db2 database called PRODUCTS, with a schema called PRDCTS and a table called Products as defined in the Products data model in https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/products_data_model.json
- ▶ Swagger that describes the data model for each path
- ▶ Unique sub flows for each relevant operation (Create, Read, Update, Delete)

Each subflow has an input, a mapping node and an output. Figure 6-4 shows the REST API operation subflow.

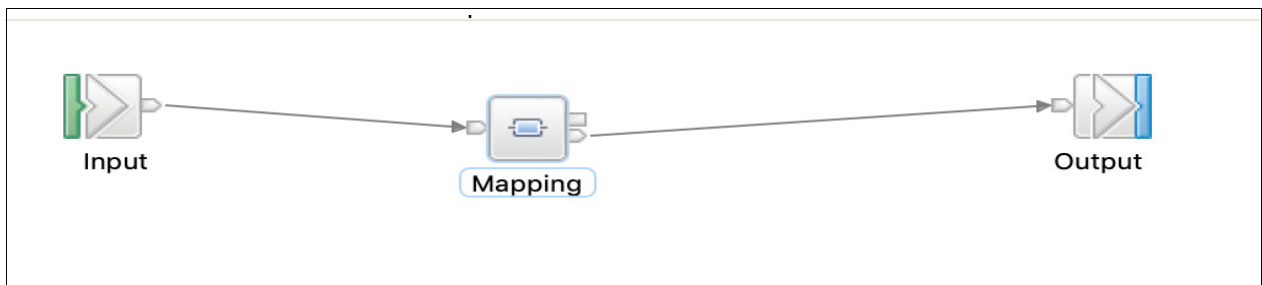


Figure 6-4 REST API Operation Subflow

Figure 6-5 shows the Queries flow.

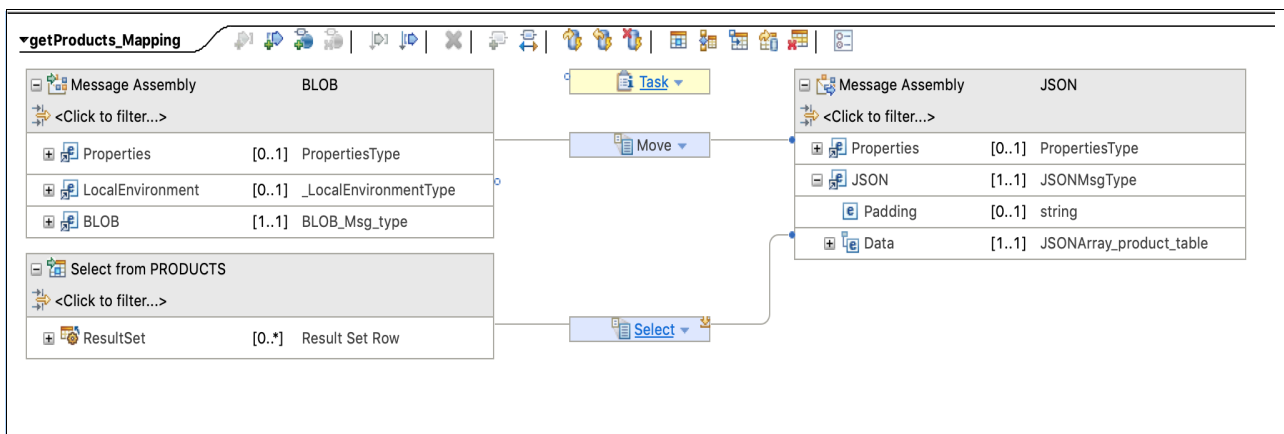


Figure 6-5 Get Product Operation Mapping Node

Figure 6-6 on page 176 through Figure 6-8 on page 177 show the Command flows.

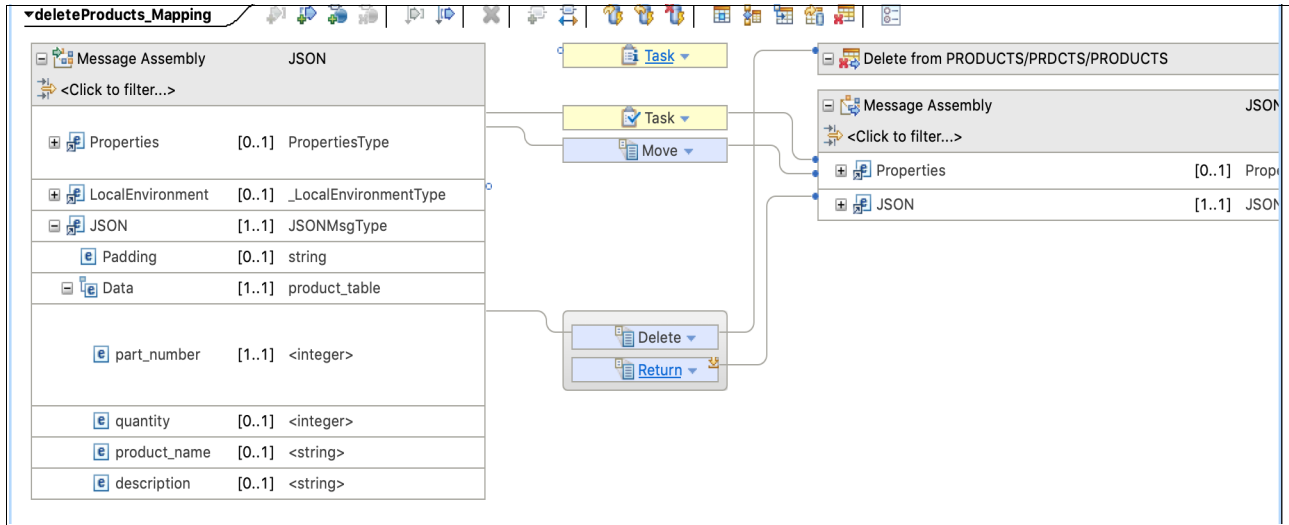


Figure 6-6 Delete Product Operation Mapping Node

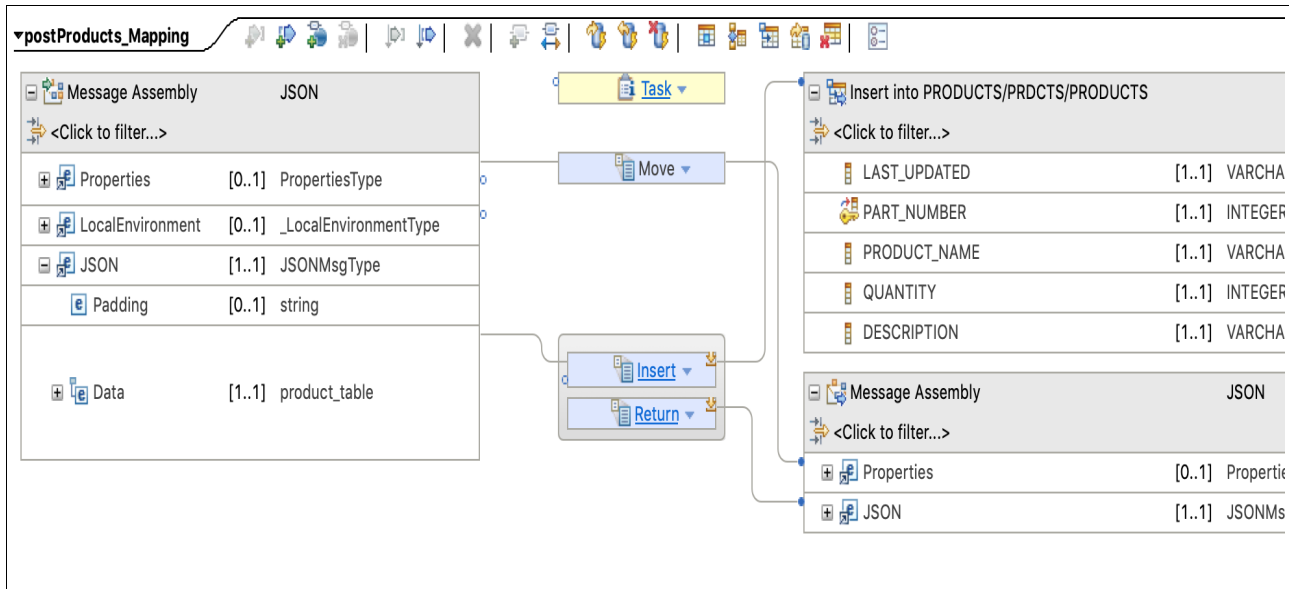


Figure 6-7 Post Product Operation Mapping Node

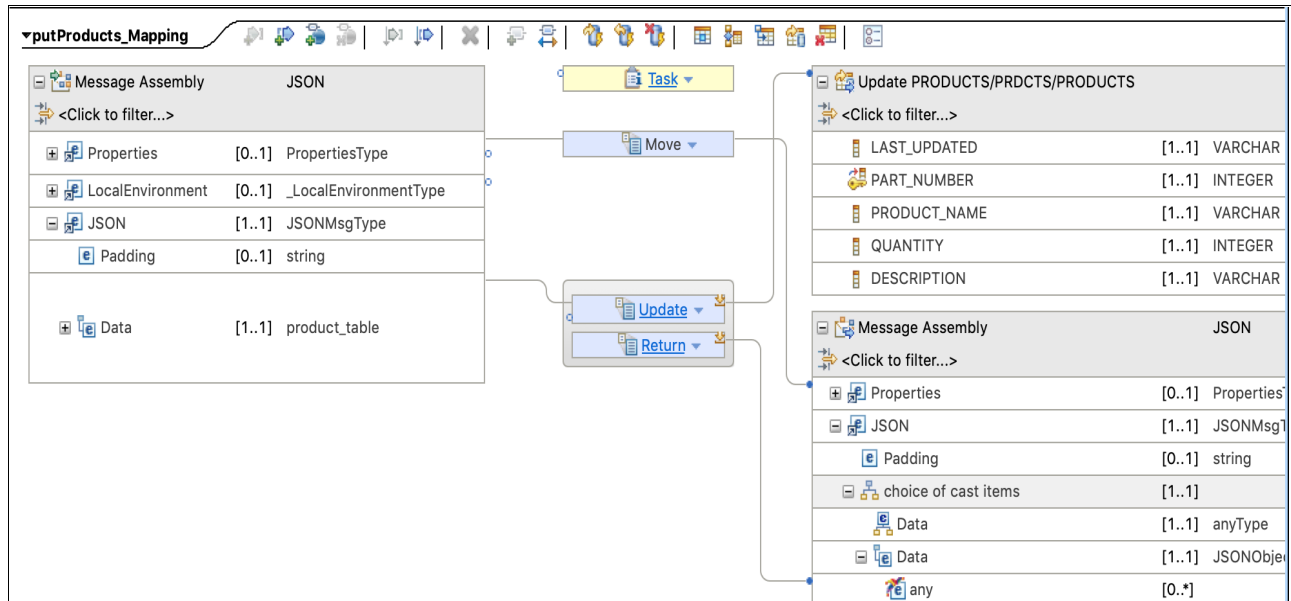


Figure 6-8 Put Product Operation Mapping Node

6.2.1 Db2 setup

For the application in this section to function correctly, a Db2 database needs to be set up for the IBM App Connect flow to operate against.

Next we must create a Db2 instance using the following instructions:

<https://github.com/IBM/Db2/tree/develop/deployment>

1. Configure the instance with the following parameters:
 - a. Choose a release name.
 - b. Select a namespace.

Note: This value does not need to be the same namespace where the IBM App Connect container will be deployed.
 - c. Set the Database Name to PRODUCTS.
 - d. Set the Db2 instance name to db2inst1.
 - e. Provide a Password for Db2 instance, for example passw0rd.
 - f. Persistence is optional.
 - g. Do not select the Db2 HADR mode.
2. After the deployment is complete, observe that ports 50000 and 55000 are both exposed via two types of Kubernetes services, NodePort and ClusterIP in the OpenShift console,

Note: For a distinction between NodePort and ClusterIP service types, refer to section 4.4.2, “Container orchestration” on page 107.

3. You can use Db2-compatible database client to connect to the Db2 instance via the NodePort service, and to verify that the database called PRODUCTS has been successfully created. In the next section, we use the IBM App Connect Toolkit in the Database Development view.

In the remainder of the chapter, we connect to the Db2 instance by using a policy definition in IBM App Connect. The definition leverages the ClusterIP service, as described in 6.4.6, “Policy definitions” on page 250.

6.2.2 Db2 table setup

To set up a Db2 table for the samples in this chapter, we use the IBM App Connect Toolkit in the Database Development view to connect to the Db2 instance that we spun up in 6.2.1, “Db2 setup” on page 177.

The following tasks are performed on the IBM App Connect Toolkit version 11.0.0.5.

1. Open the toolkit and navigate to the Database Development View. See Figure 6-9 on page 178.

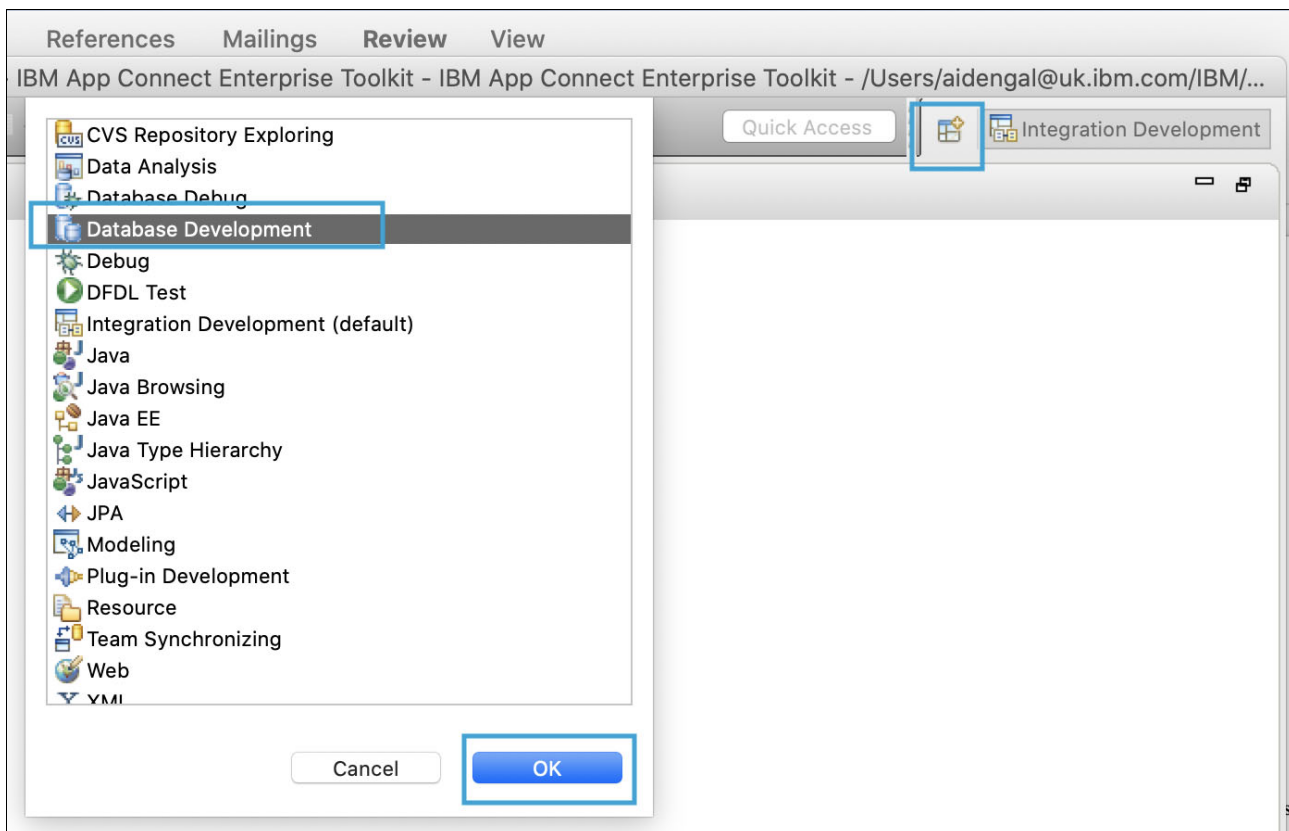


Figure 6-9 Switch to Database Development View

2. In the Data Source Explorer, right-click **Database Connections**, and select **New**. See Figure 6-10.

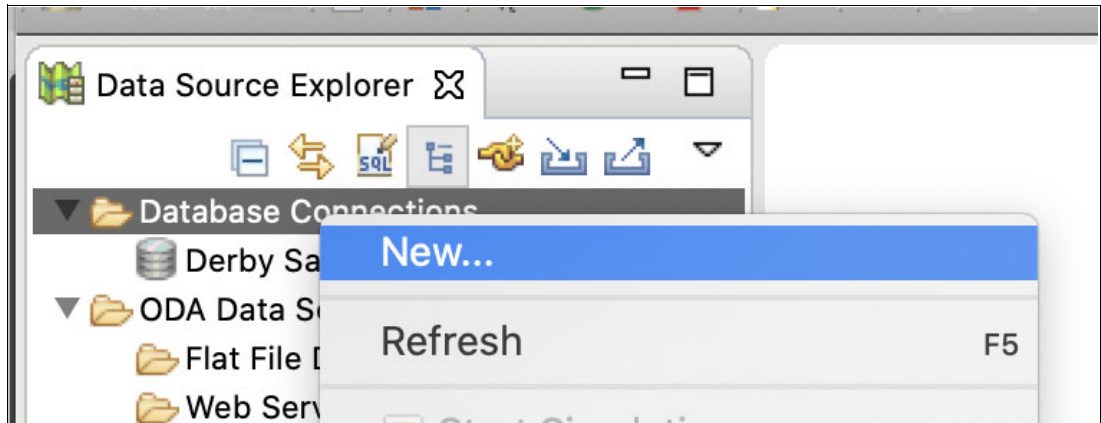


Figure 6-10 New Database Connection

3. In the connection details, select the **DB2 for Linux, UNIX, and Windows** option in the menu and put the database details from the previous section. Remember to use the NodePort service exposing the container port 50000 for the Db2 instance.

The service is typically exposed on the Proxy Node IP address of Fully Qualified Domain Name. See Figure 6-11 on page 180.

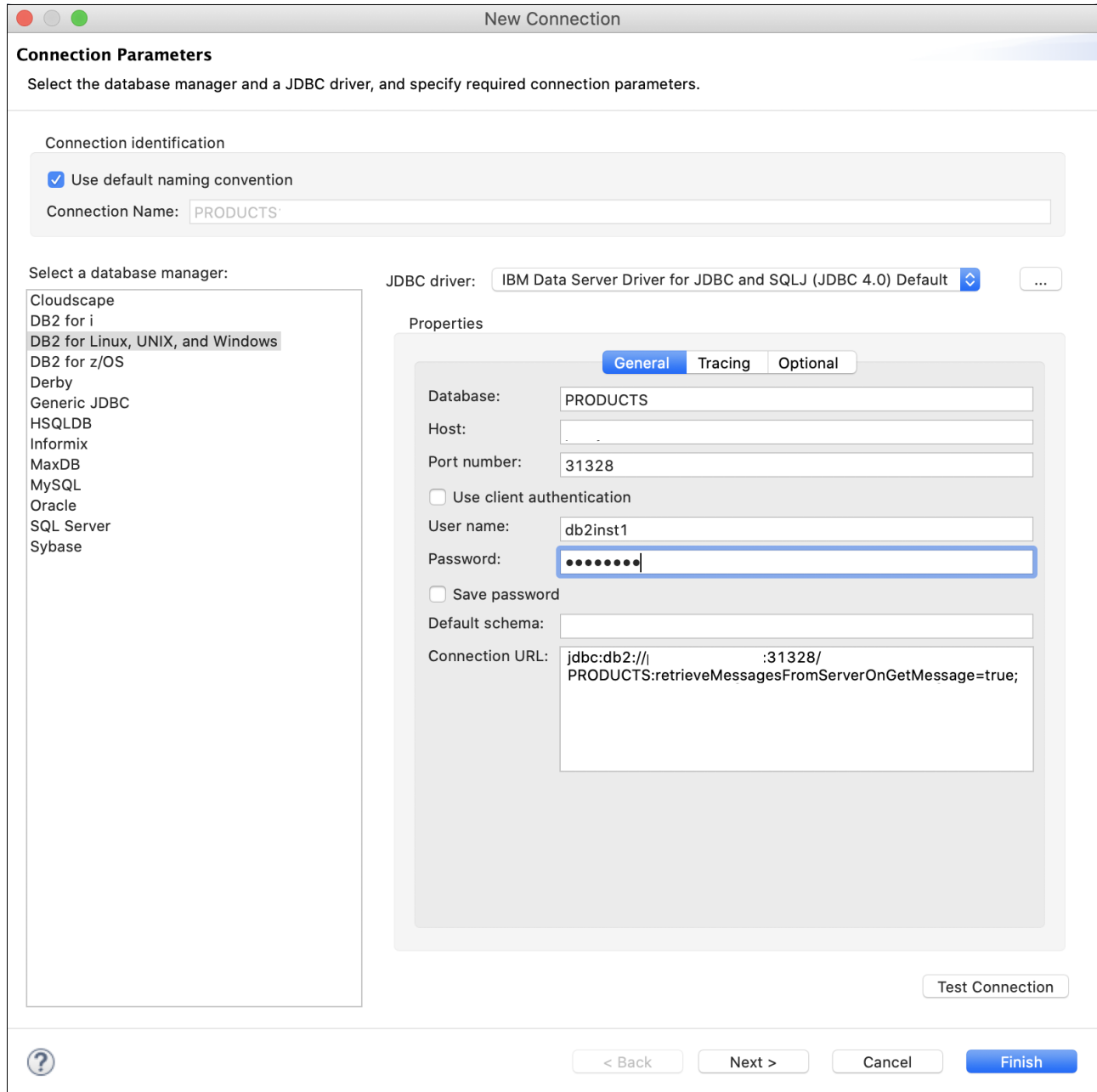


Figure 6-11 Connection details

4. The Products database connection now exists in the Database Connections sidebar. Right click **Products** and select **New SQL Script**. See Figure 6-12 on page 181.

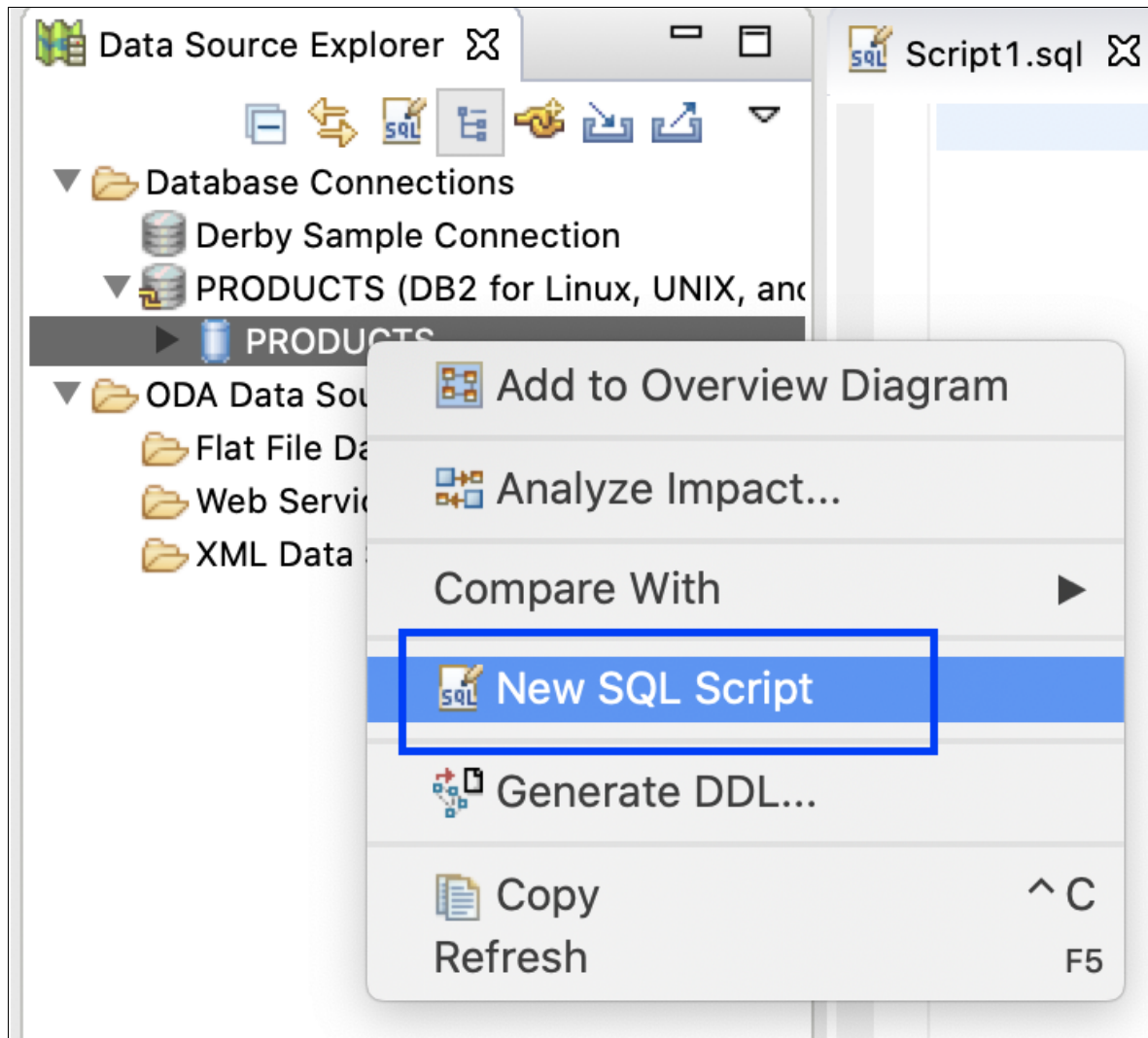


Figure 6-12 New SQL Script

5. To create a PRODUCTS table in the PRODUCTS database, type the following SQL command shown in Example 6-1 into the newly opened SQL script.

Example 6-1 SQL script

```

CREATE TABLE PRODUCTS
(
  last_updated varchar(255) NOT NULL,
  part_number int NOT NULL,
  product_name varchar(255),
  quantity int,
  description varchar(255),
  PRIMARY KEY (part_number)
);

```

6. Right click inside the SQL script and select **Run SQL**. See Figure 6-13 on page 182.

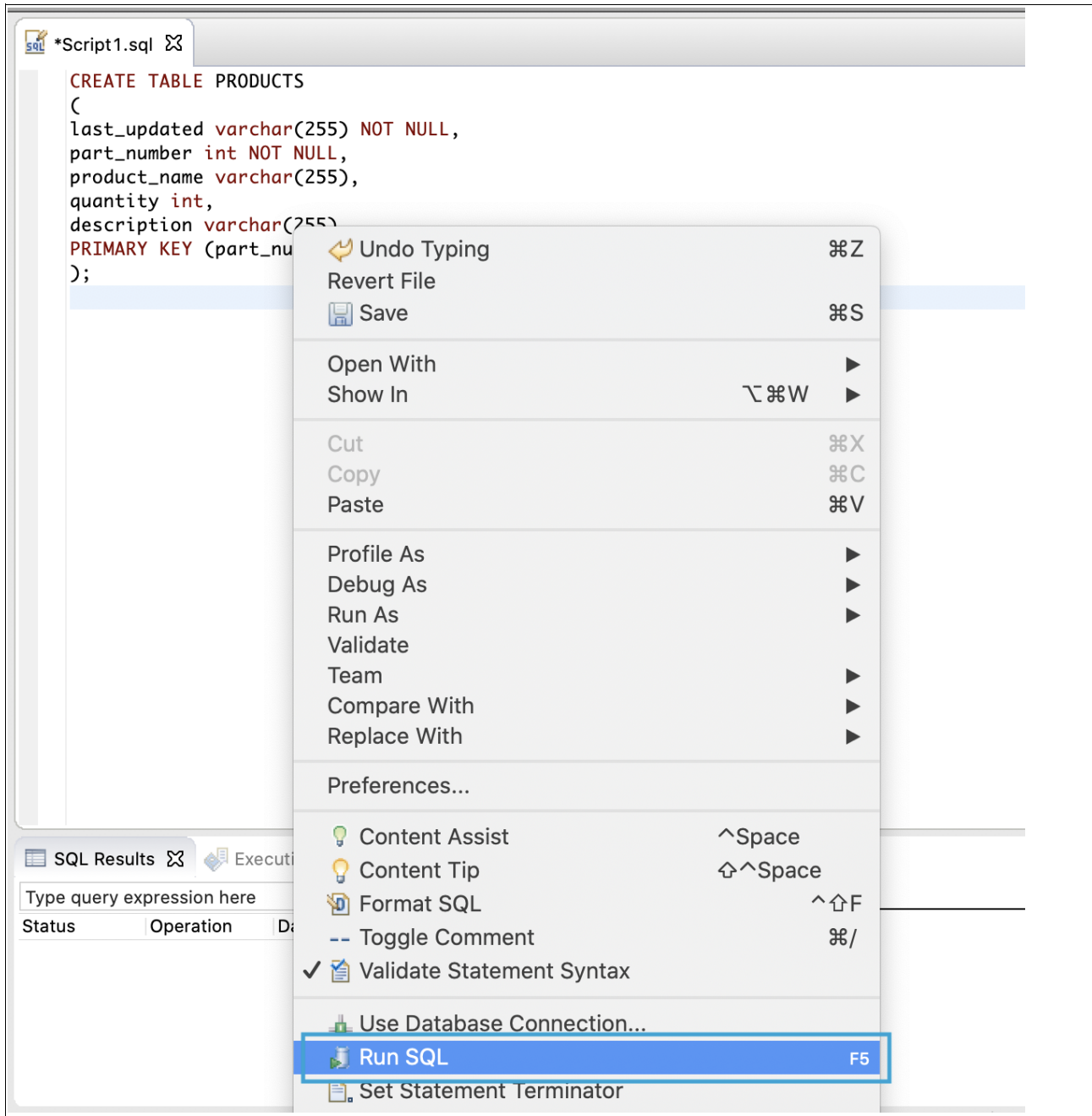


Figure 6-13 Run SQL

This should result in a success report as shown in Figure 6-14 on page 183.

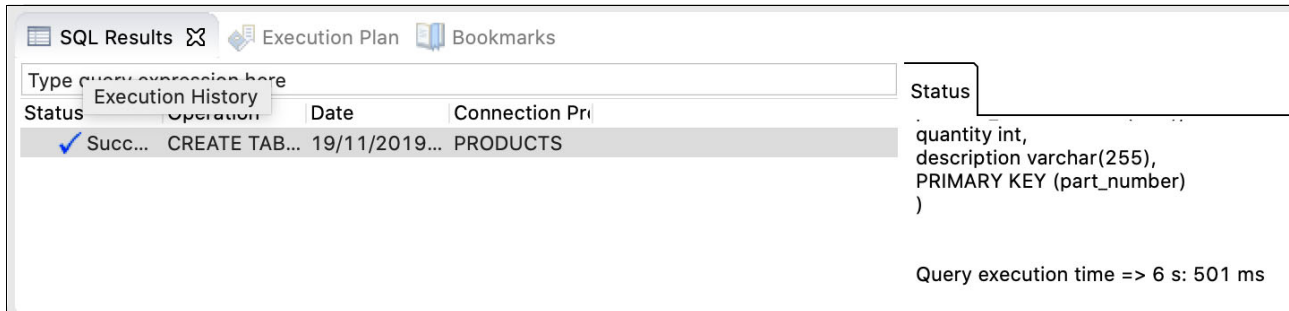


Figure 6-14 - SQL success report

- You can now insert a new row into the table using the following SQL command and again selecting the Run SQL command as previous.

```
INSERT INTO PRODUCTS ( last_updated, part_number, product_name, quantity,
description)
VALUES ('2019-08-01T09:57:34.265Z', 12, 'duck', 100, 'a waterbird with a broad
blunt bill, short legs, webbed feet, and a waddling gait');
```

- We can check that the entry has been successfully inserted by using the SELECT SQL command.

```
SELECT * FROM PRODUCTS
```

Figure 6-15 shows the SELECT SQL query result.

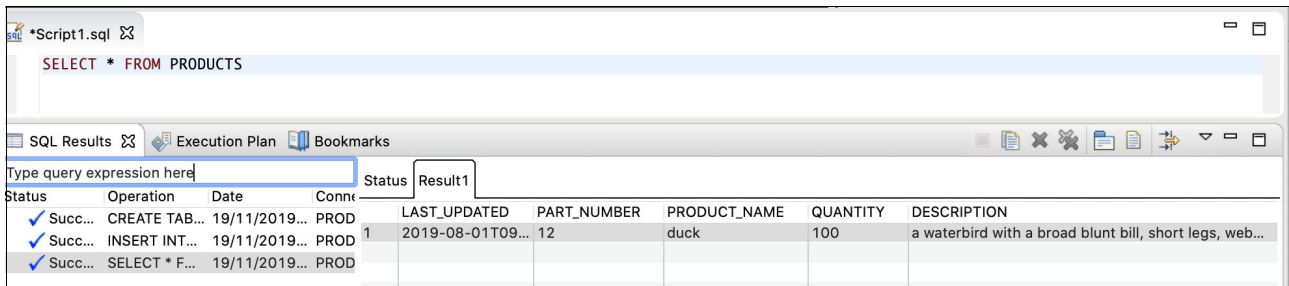


Figure 6-15 SELECT SQL query result

6.2.3 Swagger definitions

In this section, we package the project files into a BAR file for each application as shown in the following web pages:

- ▶ https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/6.1/database_operations.bar
- ▶ https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/6.1/database_query.bar

Then we create a new server for each. In this chapter we want to use the user interface to achieve this. Later, we see how this can be automated by using pipeline deployment in section 7.5, “Continuous Integration and Continuous Delivery Pipeline using IBM App Connect V11 architecture” on page 465.

- Log in to the IBM Cloud Pak for Integration instance and the IBM App Connect dashboard. In this view we click **Add server** as shown in Figure 6-16.

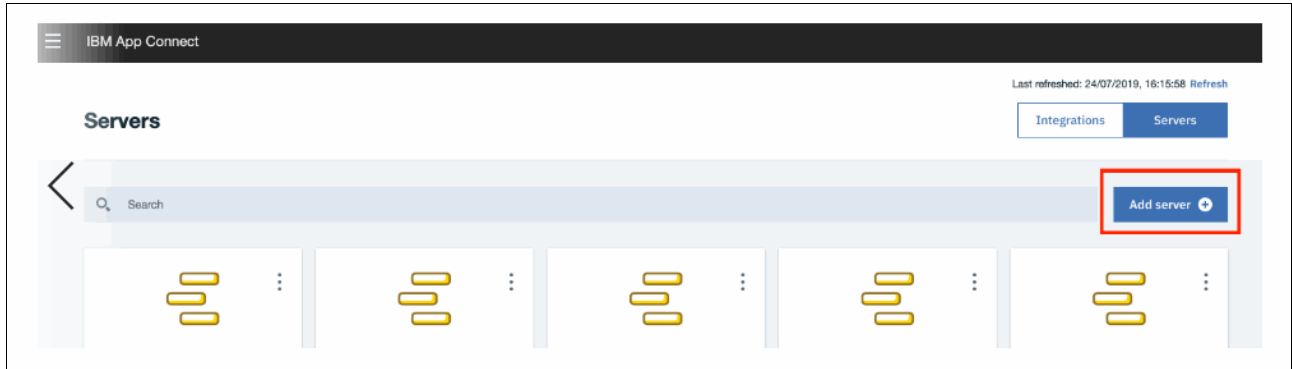


Figure 6-16 Add server in the IBM App Connect dashboard

2. We now select the BAR file that we want to import, select **Add a BAR file**. See Figure 6-17.

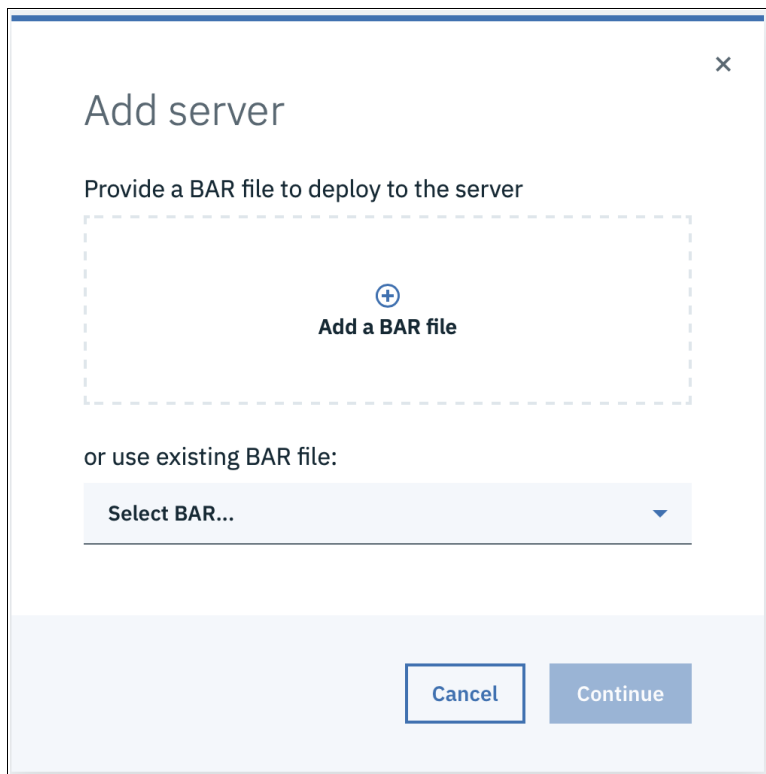


Figure 6-17 Add a BAR file

3. Next, select the BAR file from the local directory where it is saved and select **Choose**. See Figure 6-18 on page 185.

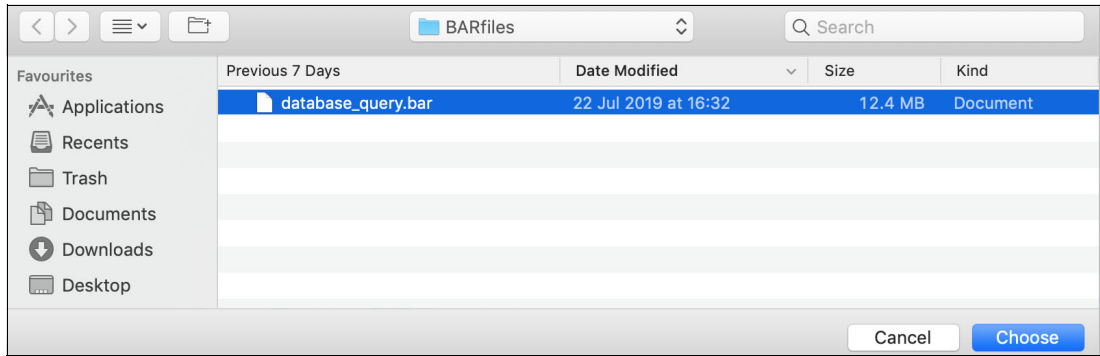


Figure 6-18 Choose the relevant BAR file

4. The BAR file is displayed in the user interface, and you can confirm that the correct file has been uploaded, then select **Continue**. See Figure 6-19.

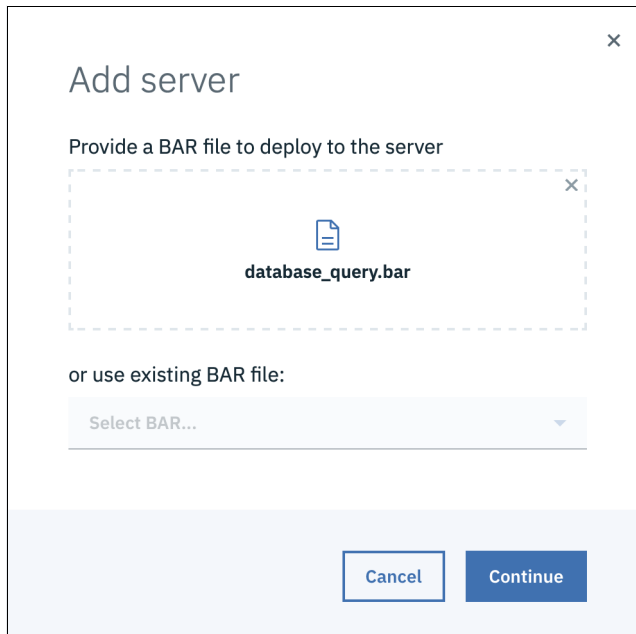


Figure 6-19 BAR file shown in the user interface

5. In the next screen we copy the **Content URL** and select **Configure release**. See Figure 6-20 on page 186.

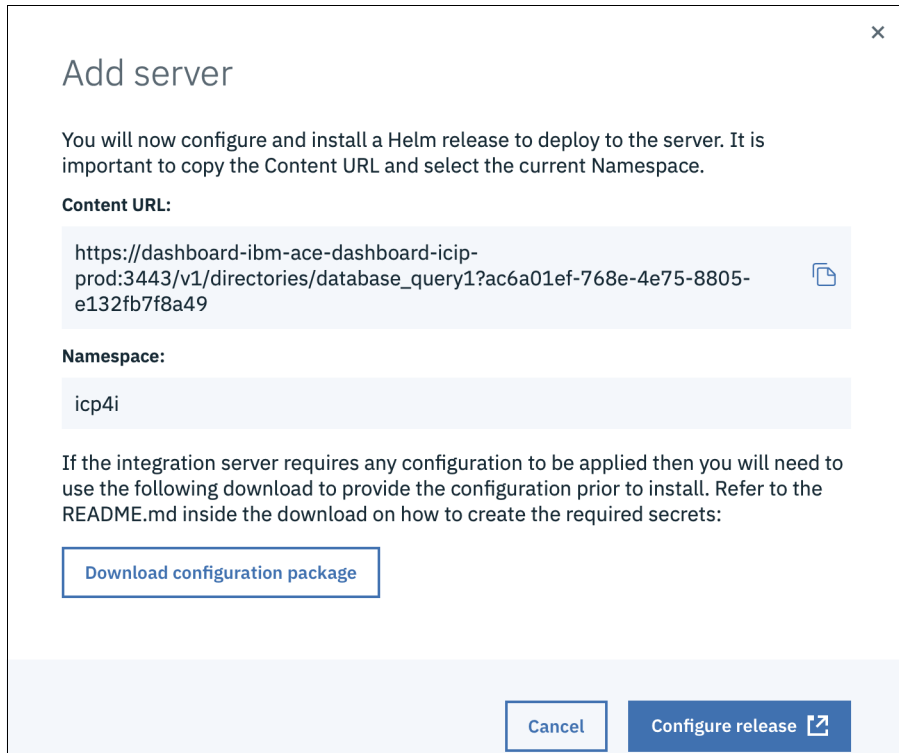


Figure 6-20 Get Content URL (ConfigureReleaseCopyContentURL)

- The new page describes the helm chart to be used to deploy the new server into, confirm the Cloud Pak version and select **Configure**. See Figure 6-21 on page 186.

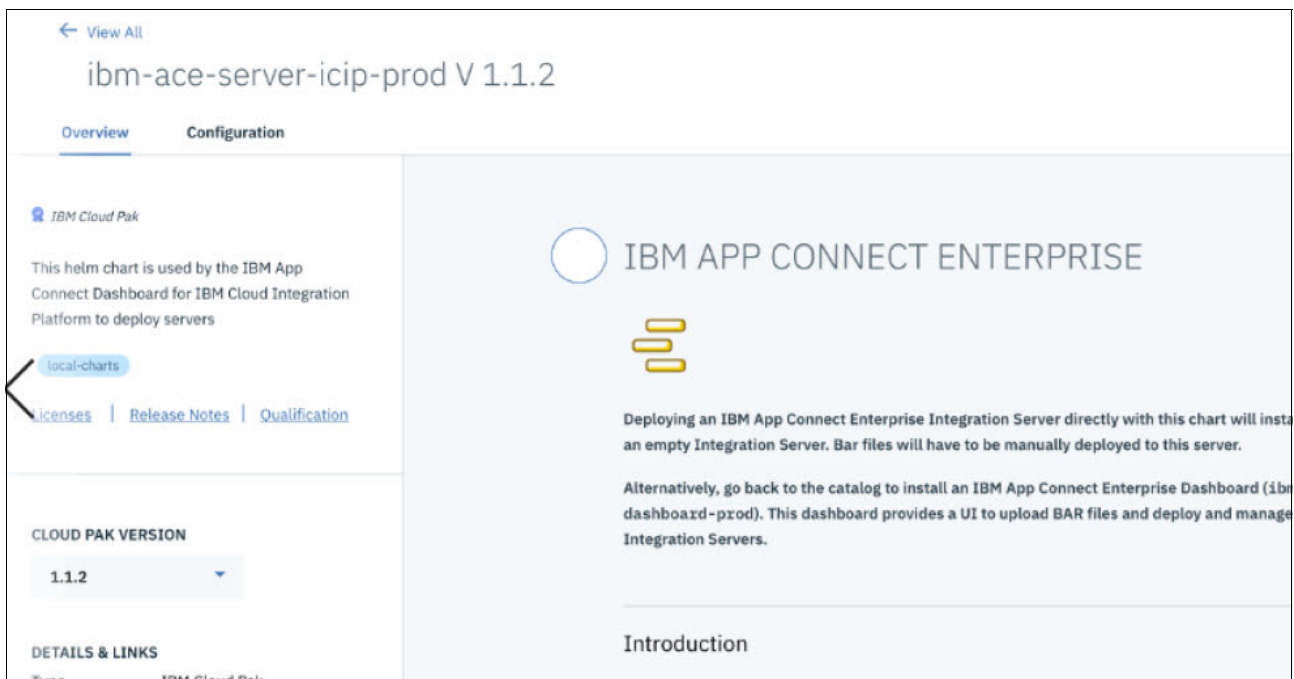
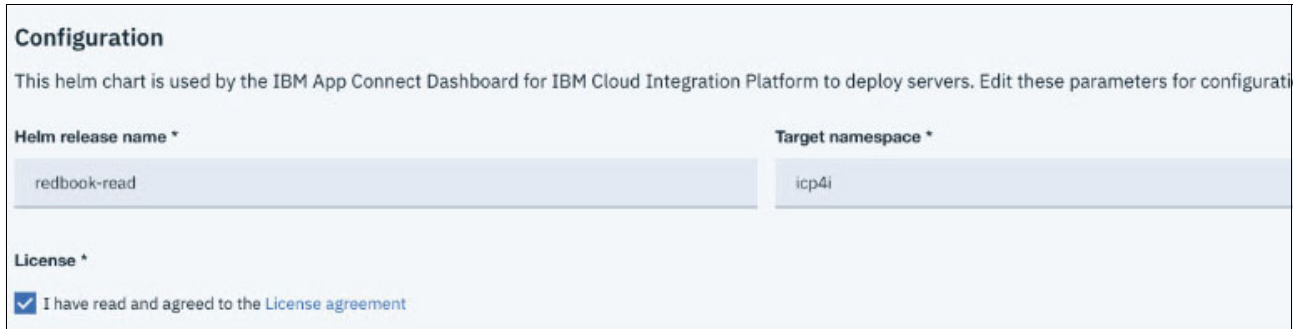


Figure 6-21 IBM App Connect Helm Chart

- Type a Helm release name such as redbook-read for the database query BAR and redbook-commands for the database commands BAR. In the Target namespace select

icp4i before you read and accept the license agreement by checking, **I have read and agreed to the license agreement**. See Figure 6-22 on page 187.



Configuration

This helm chart is used by the IBM App Connect Dashboard for IBM Cloud Integration Platform to deploy servers. Edit these parameters for configuration.

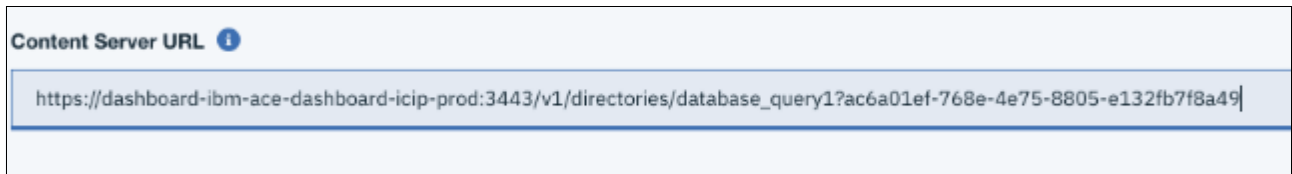
Helm release name * **Target namespace ***

License *

I have read and agreed to the [License agreement](#)

Figure 6-22 Helm release name, target namespace and accepted license agreement

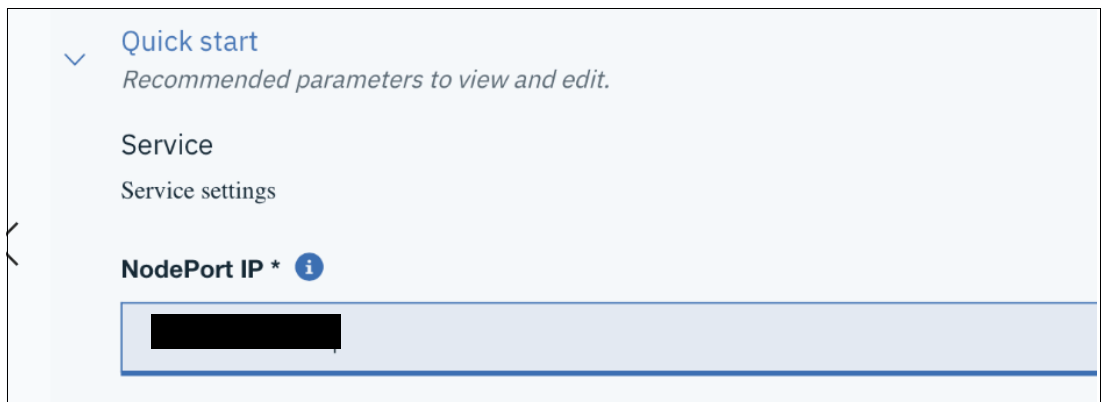
8. Paste the previously copied Content URL into the **Content Server URL** box. See Figure 6-23 on page 187.



Content Server URL ⓘ

Figure 6-23 Pasted Content URL

9. Type the IP address for the Master or Proxy node of the IBM Cloud Pak for Integration instance into NodePort IP box. See Figure 6-24 on page 187.



Quick start
Recommended parameters to view and edit.

Service
Service settings

NodePort IP * ⓘ

Figure 6-24 Deployment IP

10. For this example we need only a single replica. So, set the **Replica count** to '1' and ensure that the **Local default Queue Manager** is deselected as shown in Figure 6-25 on page 188.

Local default Queue Manager

Architecture scheduling preference * File system group ID

amd64 Enter value

Replica count The name of the secret to create or to use that contains the server configuration

1 Enter value

Figure 6-25 Replica Count 1 and Queue Manager deselected

11. After double checking the configuration, select **Install**.
12. A message confirming the starting of the installation is displayed. The message can be tracked in the helm release by redirecting through the **View Helm Release** button. See Figure 6-26 on page 188.

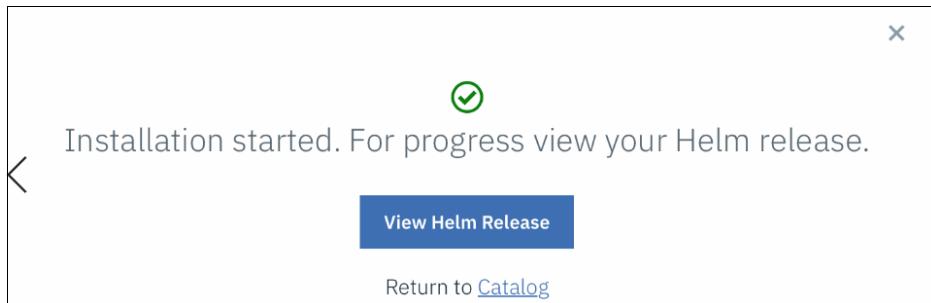


Figure 6-26 Helm Deployment successful start

13. The new services for the helm release is displayed. To see further details, including the exposed ports for the server, select the link **redbook-read-ibm-ace-server-icjp-prod** or **redbook-commands-ibm-ace-server-icjp-prod** depending on the BAR file that is being deployed. See Figure 6-27 on page 188.

Service				
NAME	TYPE	CLUSTER IP	EXTERNAL IP	PORT(S)
redbook-read-ibm-ace-server-icjp-prod-ace-metrics	ClusterIP	10.0.247.254	<none>	9483/TCP
redbook-read-ibm-ace-server-icjp-prod	NodePort	10.0.234.66	<none>	7600:32335/TCP,7800:31016/TCP,7843:31369/TCP

ServiceAccount	
NAME	SECRETS

Figure 6-27 Two services for the deployed IBM App Connect server

14. In Figure 6-28 on page 189 we can see the HTTP server exposed port (31016).

Created	0 minutes ago
Type	NodePort
Labels	app=ibm-ace-server-icip-prod,chart=ibm-ace-server-icip-prod,heritage=Tiller,release=redbook-read
Selector	app=ibm-ace-server-icip-prod,release=redbook-read
Cluster IP	10.0.234.66
External IP	-
Load balancer IP	-
Port	webui 7600/TCP; ace-http 7800/TCP; ace-https 7843/TCP
Node port	webui 32335/TCP ace-http 31016/TCP ace-https 31369/TCP
Session affinity	None

Figure 6-28 Exposed ports for the service

15. We can then use the port to create a test url to make a GET on the redbook-read server. In an API Connect Test and Monitor select a **GET** operation `http://<ibm_cloud_ip>:<http_port>/database_query/v1/products` which returns the records from the PRODUCTS table in the database. See Figure 6-29 on page 189.

The screenshot shows the IBM API Connect Test and Monitor interface. The URL bar contains a GET request to `http://[redacted]:31016/database_query/v1/products`. The response is displayed in a JSON array format:

```
[
  {
    last_updated: "07-23-2019T08:27:24.050Z",
    part_number: 12,
    quantity: 80,
    product_name: "duck",
    description: "a waterbird with a broad blunt bill, short legs, webbed feet, and a waddling gait"
  },
  {
    last_updated: "07-23-2019T08:33:25.231Z",
    part_number: 20,
    quantity: 8,
    product_name: "red lorry",
    description: "large vehicle for moving goods"
  },
  {
    last_updated: "07-23-2019T09:46:21.951Z",
    part_number: 1200,
    quantity: 8,
    product_name: "red blue lorry",
    description: "large vehicle for moving cats"
  },
  {
    last_updated: "07-22-2019T16:04:03.922Z",
    part_number: 201,
    quantity: 8,
    product_name: "red blue lorry",
    description: "large vehicle for moving cats"
  }
]
```

Figure 6-29 Returned output from the PRODUCTS table

It is worth considering that the policy used in both BAR files is dependent on its reference to the datastore. This must be updated if the port of the database changes. It is possible to

deploy the BAR directly to an existing empty server through the Toolkit and as the system gains maturity automated builds based on code repository pushes can be deployed.

We now have two BAR files deployed onto the Cloud that enable us to read/add/delete/update rows in the database. These bar files can be independently maintained and scaled, and can have separately defined availability models.

6.3 Expose an API using API Management

This section shows you how to expose an API using API Management, which brings the following benefits. (Not all of these issues are explored in detail in the example.)

- ▶ **Discovery:** Enable consumers to find the APIs they need, understand their specifications, learn how to use them, and experiment with them before committing to use them.
- ▶ **Self-subscription:** Allow consumers to self-subscribe to use the API using a revocable key.
- ▶ **Routing:** Hide the exact location of the API implementation, and enable version based routing.
- ▶ **Traffic management:** Provide throttling of inbound requests to the API on a per-consumer basis
- ▶ **Analytics:** Provide both consumers and providers with information regarding their API usage, response times and more.

In the first exercise we created the two basic implementations of the REST APIs. We now want to bring those together into a single consolidated API to simplify usage for the consumer. Furthermore, we want to provide API management capabilities as shown in Figure 6-30.

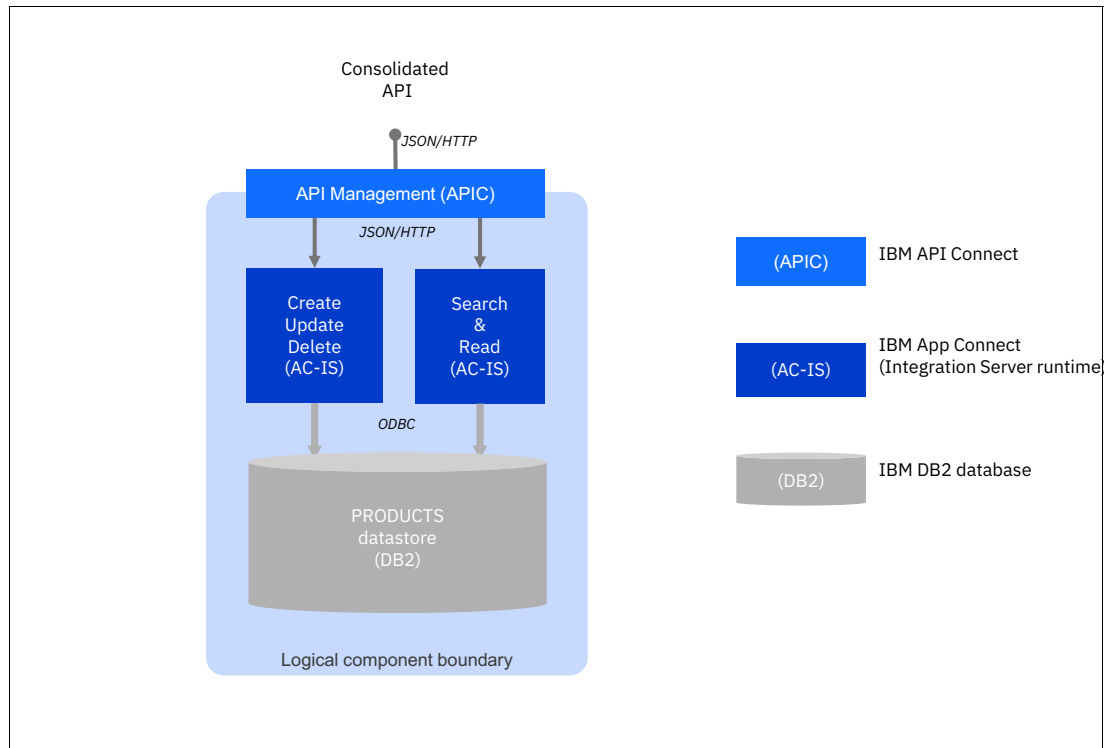


Figure 6-30 Providing API management capabilities

Having created the implementation of REST APIs in the previous section we now want to expose them through an API Management system. This gives the ability to manage how the APIs are consumed, how traffic can be limited and how exposure to external parties is properly handled.

6.3.1 Importing the API definition

There are two methods for exporting an IBM App Connect REST API to IBM API Connect.

Pushing from IBM App Connect to IBM API Connect

The first option is to make use of the **Push REST APIs to API Connect...** functionality which is available from the App Connect dashboard. This is documented in the IBM Knowledge Center:

https://www.ibm.com/support/knowledgecenter/SSTDS_11.0.0/com.ibm.etools.mft.doc/bn28905_.htm

Note: This functionality currently is possible in stand-alone instances of IBM App Connect to any given API Connect instance. Be aware that at the time of writing there was a limitation in IBM Cloud Pak for Integration deployments that meant a callback 'POST' to the IBM App Connect Server is not available.

Importing the API definition file

The second option is to import the API definition file manually.

Perform the following steps to import API to the Developer Workspace:

1. First, you must create the API. To do that, click on **Develop APIs and Products**. See Figure 6-31 on page 191.

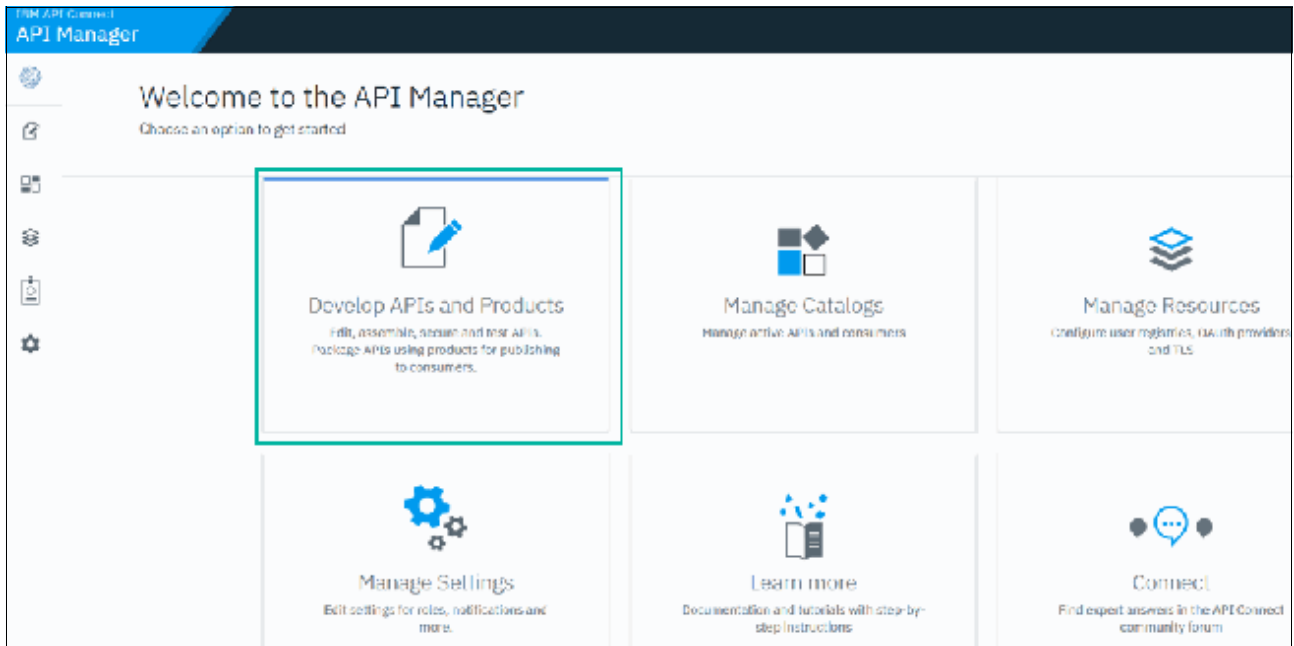


Figure 6-31 API Manager main page

2. Click on **Add** → **API**. See Figure 6-32.

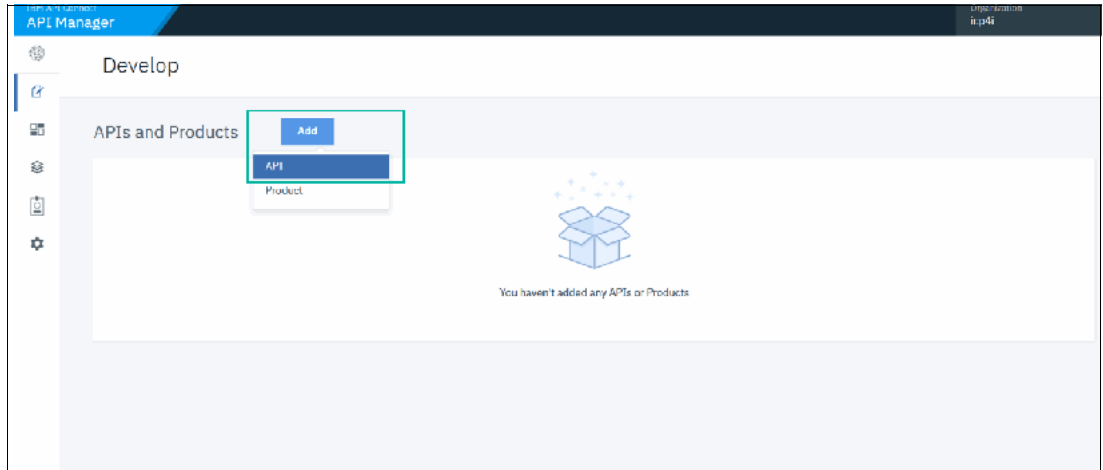


Figure 6-32 Import API to the developer workspace 1

3. Select **From existing OpenAPI service**. See Figure 6-33 on page 192.

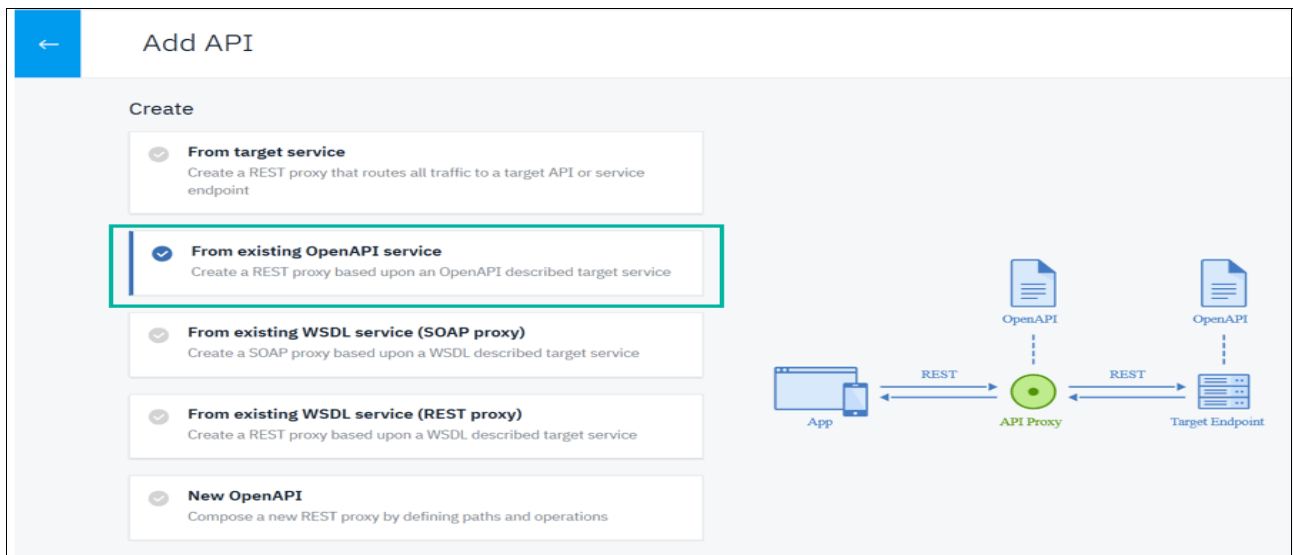


Figure 6-33 Import API to the developer workspace 2

4. Click **Next**. See Figure 6-34.

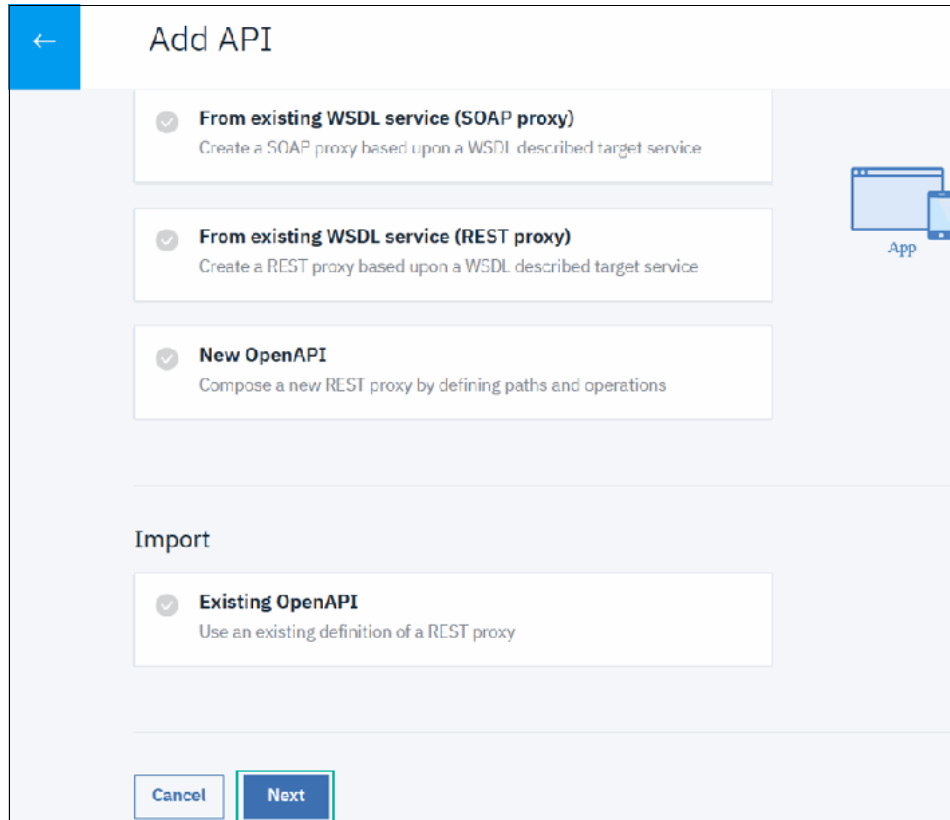


Figure 6-34 Import API to the developer workspace 3

Note: Download the swagger definition from https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/database_operations_swagger.json.

5. Click **Browse** and choose the Swagger definition that you have downloaded from GitHub. See Figure 6-35.

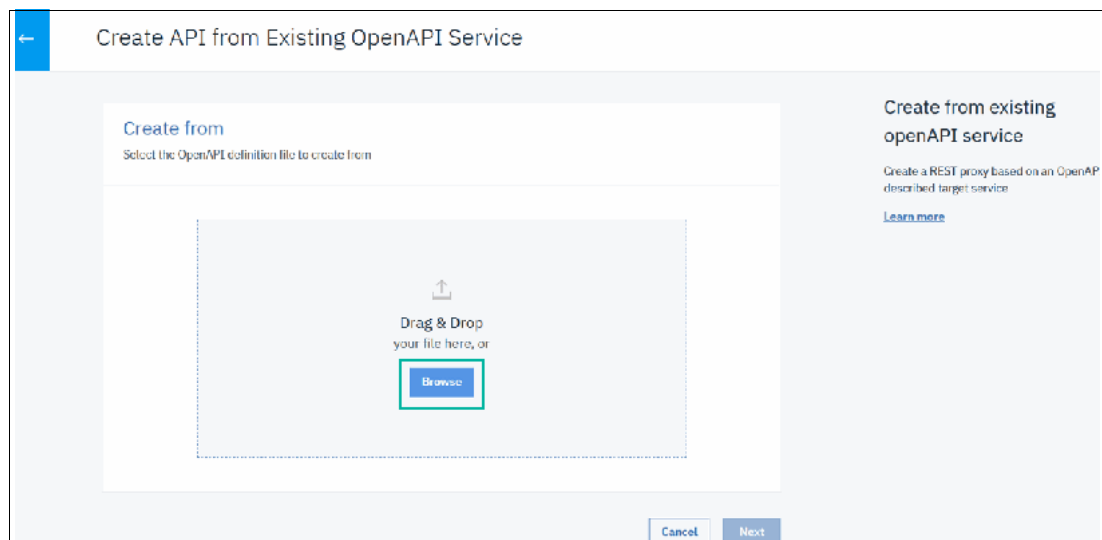


Figure 6-35 Import API to the developer workspace 4

6. Click **Next**. See Figure 6-36.

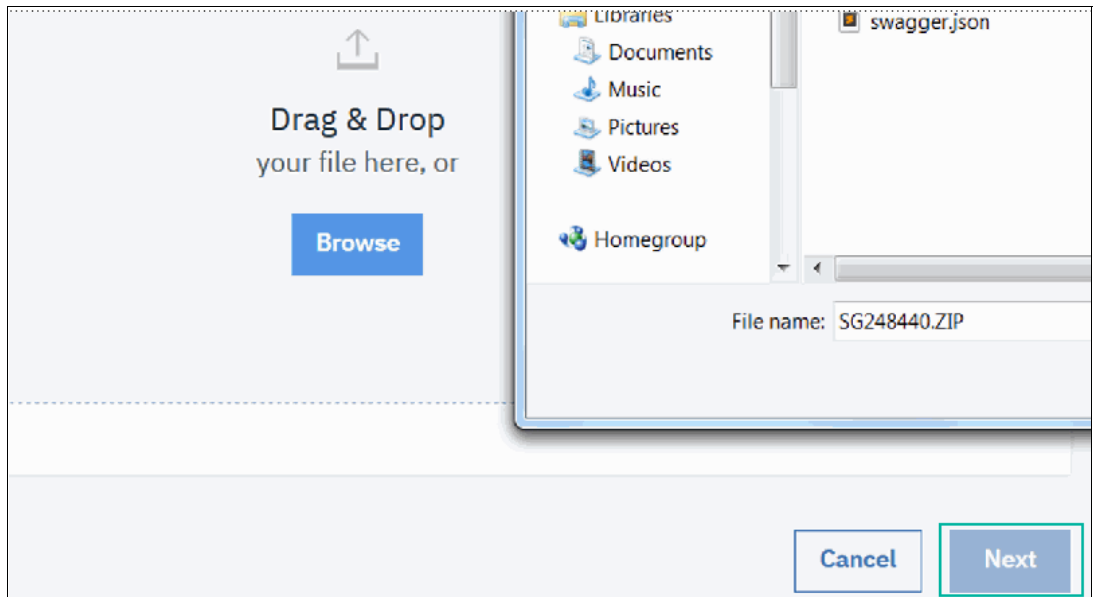


Figure 6-36 Import API to the developer workspace 5

7. The API should be imported successfully. Click **Next**. See Figure 6-37 on page 194.

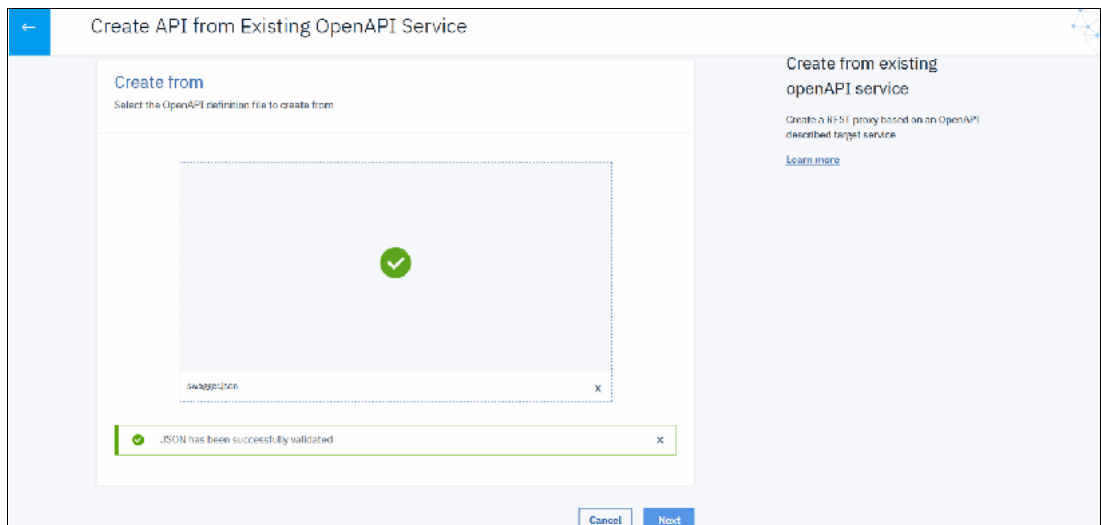


Figure 6-37 Import API to the developer workspace 6

8. Accept the defaults and click **Next**. Figure 6-38.

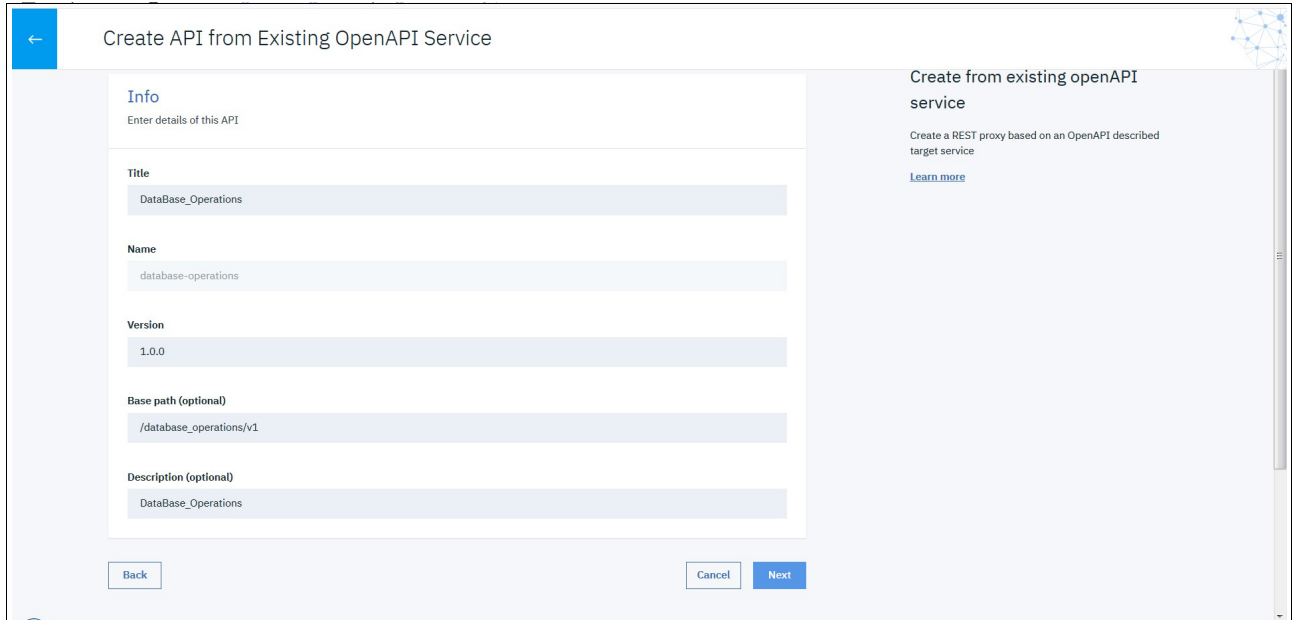


Figure 6-38 Import API to the developer workspace 7

9. Do not check **Activate API** and click **Next**. See Figure 6-39 on page 195.

Descriptions:

Secure using Client ID - Select this option to require an Application to provide a Client ID (API Key). This causes the X-IBM-Client-Id parameter to be included in the request header for the API. When this option is selected, you can then select whether to limit the API calls on a per key (per Client ID).

CORS - Select this option to enable cross-origin resource sharing (CORS) support for your API. This allows your API to be accessed from another domain.

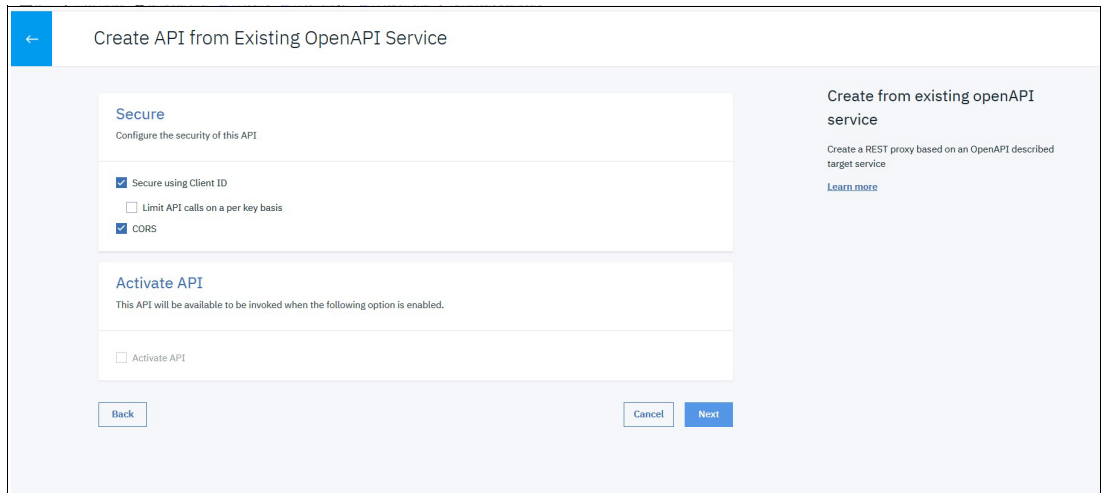


Figure 6-39 Import API to the developer workspace 8

6.3.2 Configuring the API

In this section we edit the API to include two different APIs from two microservices for the same business function. From a consumer point of view, it looks like a single API but it is actually connected to two different microservices at the back end.

6.3.3 Merging two application flows into a single API

We have now deployed the Commands swagger into the API Management service of API Connect, either through the App Connect push functionality or through a direct import of the API definition into API Connect.

This API and Product now displays the command operations for the database (Add, Update, Delete), but not the query (Read). We add this query manually by navigating to the API in API Connect.

1. Navigate to the API Manager API Definition in API Connect for `database_operations` and select **Paths** (Figure 6-40 on page 196) and the **/products** (Figure 6-41 on page 196) path name.

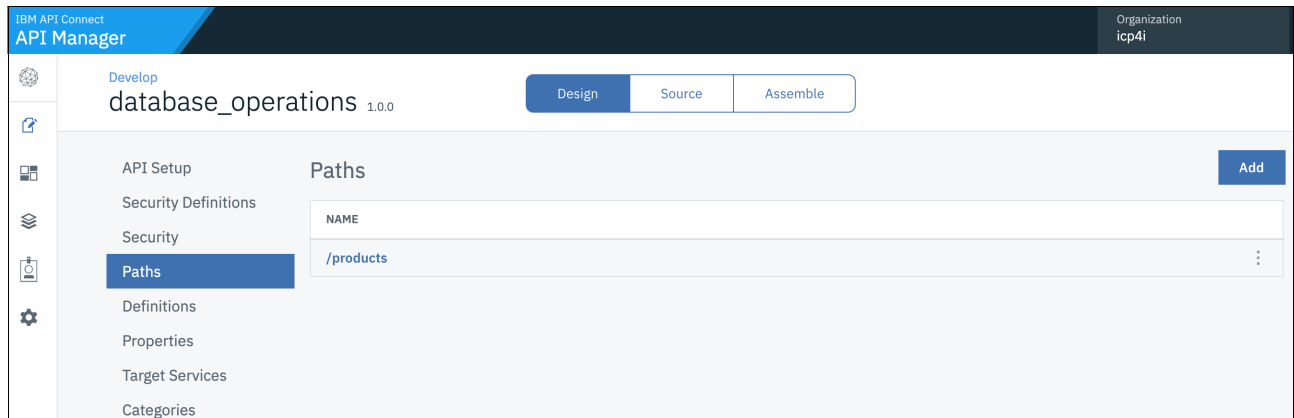


Figure 6-40 API Definition in API Connect

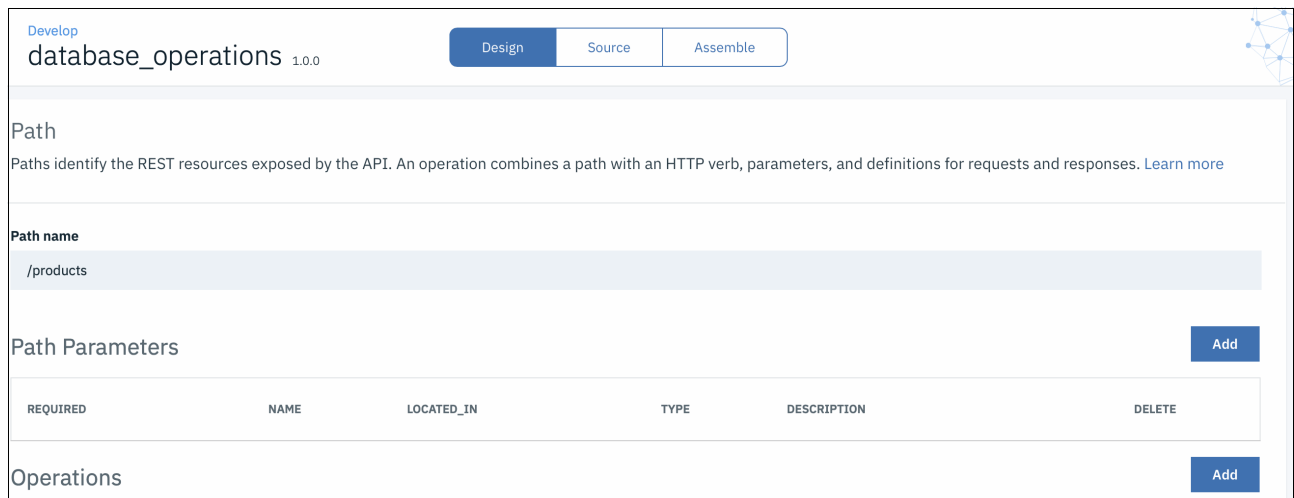


Figure 6-41 API definition in API Connect in Path

2. In the operations section, we select the **Add** button to include a get operation by toggling ON the **get** option and selecting **Add**. See Figure 6-42 on page 197.



The screenshot shows a dialog box titled "Add Operation". Inside the dialog, there is a section titled "Operations (optional)". Below this title is a list of seven HTTP methods, each with a checkbox to its left. The methods and their checkbox states are: GET (checked), PUT (checked), POST (checked), DELETE (checked), OPTIONS (unchecked), HEAD (unchecked), and PATCH (unchecked).

Figure 6-42 Add the GET operation

3. Select the **Get** operation and give an **Operation Id** like `getProducts` and a **Description** like `getProducts`. See Figure 6-43 on page 198.

Paths

/products

Operation

Operation Id (optional)

getProducts

Summary (optional)

Tags (optional)

Description (optional)

Retrieve a products

Figure 6-43 Set OperationID and Description

4. In the same page toggle **ON** the following; Override API Produce Types, application/json, Override API Consume Types, application/json. See Figure 6-44 on page 199.

Override API Security Definitions
 Override API Produce Types

Produces (optional)

application/json
 application/xml

Add media type (optional)

Override API Consume Types

Consumes (optional)

application/json
 application/xml

Figure 6-44 Set the Produces and Consumes types

- On the same page, we go to **Response** and select **Add** before you set Status Code to 200, Schema to object and Description to “The operation was successful”. See Figure 6-45.

Response Add			
STATUS CODE	SCHEMA	DESCRIPTION	DELETE
200	object	The operation was successful.	

Figure 6-45 Create a Response

- Remember to **Save** the API before navigating to the **Assemble** view of the designer.
- In the assemble view, we need to ensure that there is a **Switch** object to point to each of the four operations. If not, drag and drop the **Switch** object from the side panel into the assembly flow. See Figure 6-46 on page 200.



Figure 6-46 Add a Switch statement

8. We add a case condition for each of the path parameters. Click on the **Switch** in the assembly, and click the **+ Case** button 3 times to give four conditions. Select each to **Add the operation** from a list of available operations, one to each case. See Figure 6-47.

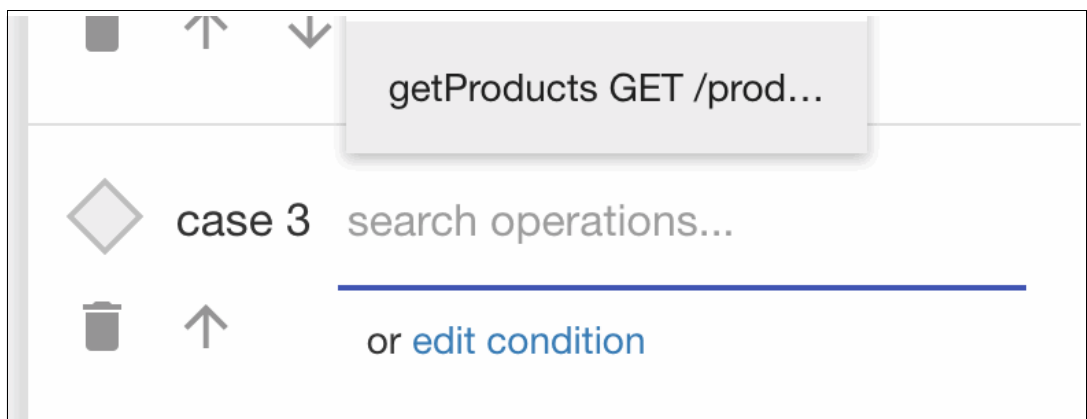


Figure 6-47 Add Cases for each operation

9. For each operation we include a proxy for the POST, PUT, and DELETE operations and an invoke for the GET. The differences are described here: <https://chrisphillips-cminion.github.io/apiconnect/2017/07/17/Proxy-and-Invoke-What-is-the-difference-in-API-Connect.html>.) Also see Figure 6-48.

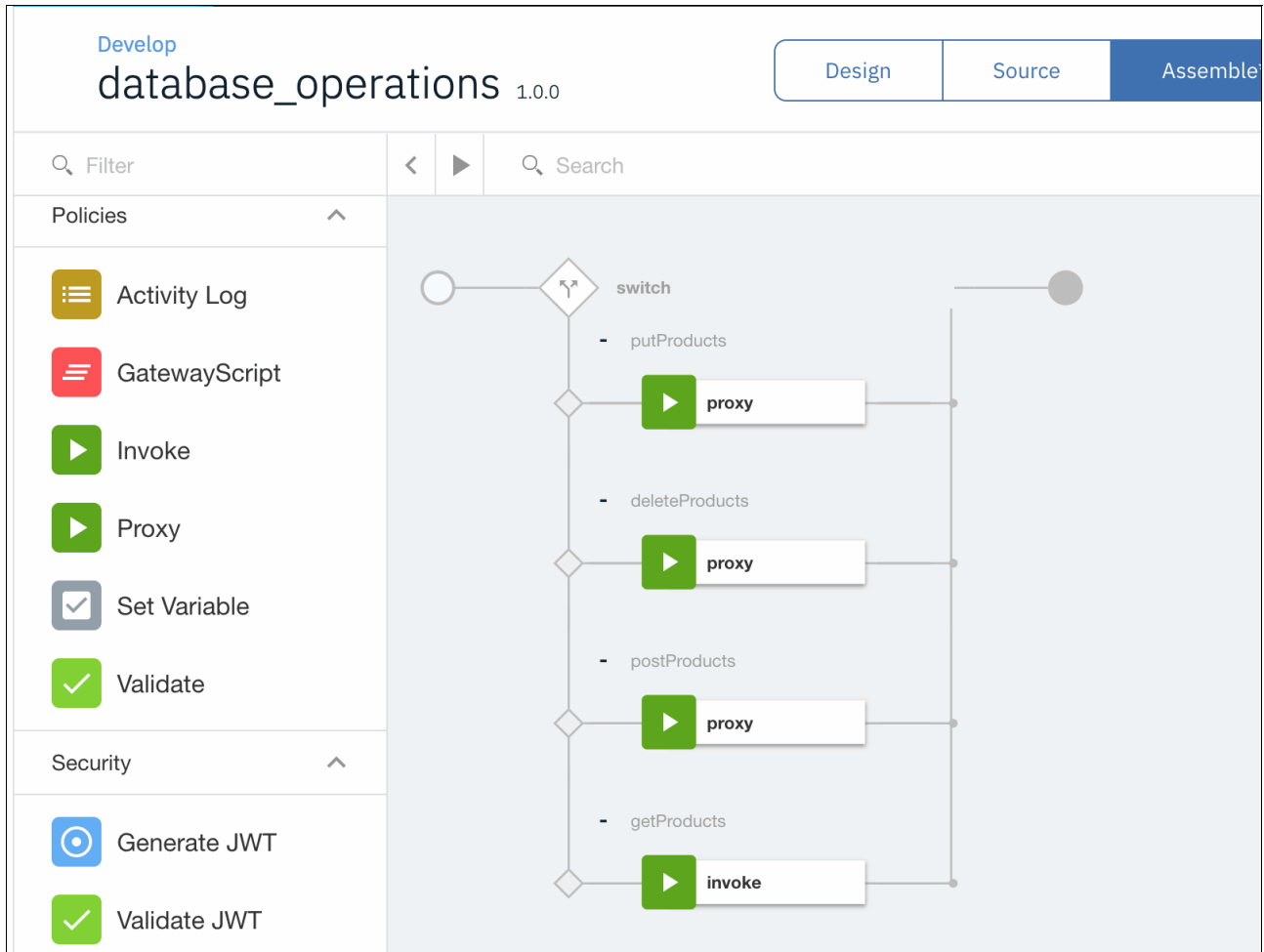


Figure 6-48 Add Proxy and Invokes for each assembly path

10. In the POST, PUT, and DELETE proxies we include the **ACE Commands Endpoint** and the GET invoke includes the **ACE Query Endpoint**. See Figure 6-49 on page 202.

The screenshot shows a configuration window for an API endpoint. The title bar includes a green play button icon and the text 'invoke', along with window control icons. The main area is divided into sections: 'Title' with the value 'invoke', 'Description' (empty), 'URL' with the value 'http://[redacted]/database_query/v1/products', and 'TLS Profile' with the value '(none)'. A tooltip for the URL field reads 'The URL to be invoked.' and a tooltip for the TLS Profile field reads 'The TLS Profile to use for the secure transmission of data.'

Figure 6-49 Define the required endpoints

11. Click **Save**.

Now that the API is ready to use, we can look at becoming the consumer of this API.

6.3.4 Add simple security to the API

In this section, we show you how to secure the API.

Configure API key security

In this section we will:

1. Define the API with simple security like API key and API secret
2. Publish the product
3. Test the API

Define simple security

Perform the following steps:

1. On the API page, click **Security Definitions** then click **Add**. See Figure 6-50.

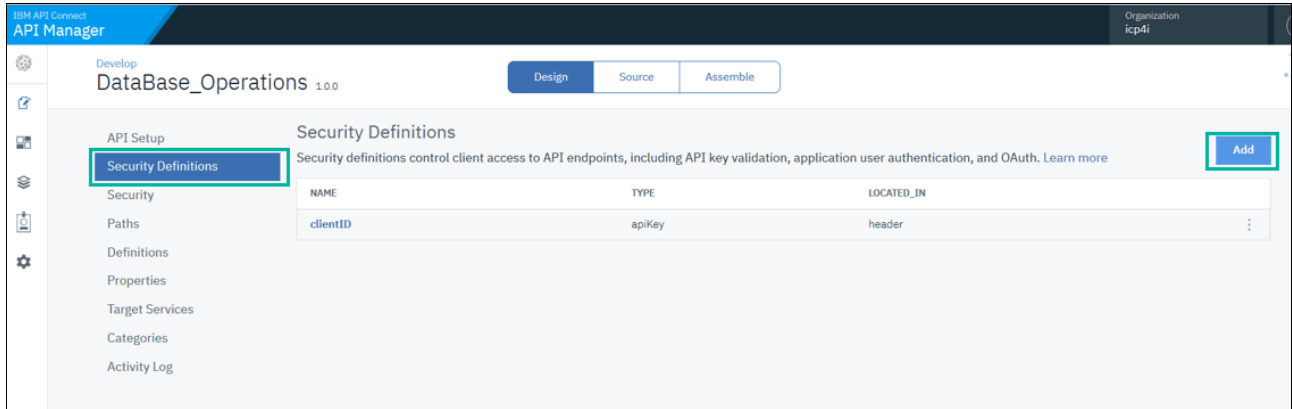


Figure 6-50 Defining the security 1

2. Under name type **secret**, choose **APIKey** for the Type and Located in Header. Then click **Save**. See Figure 6-51.

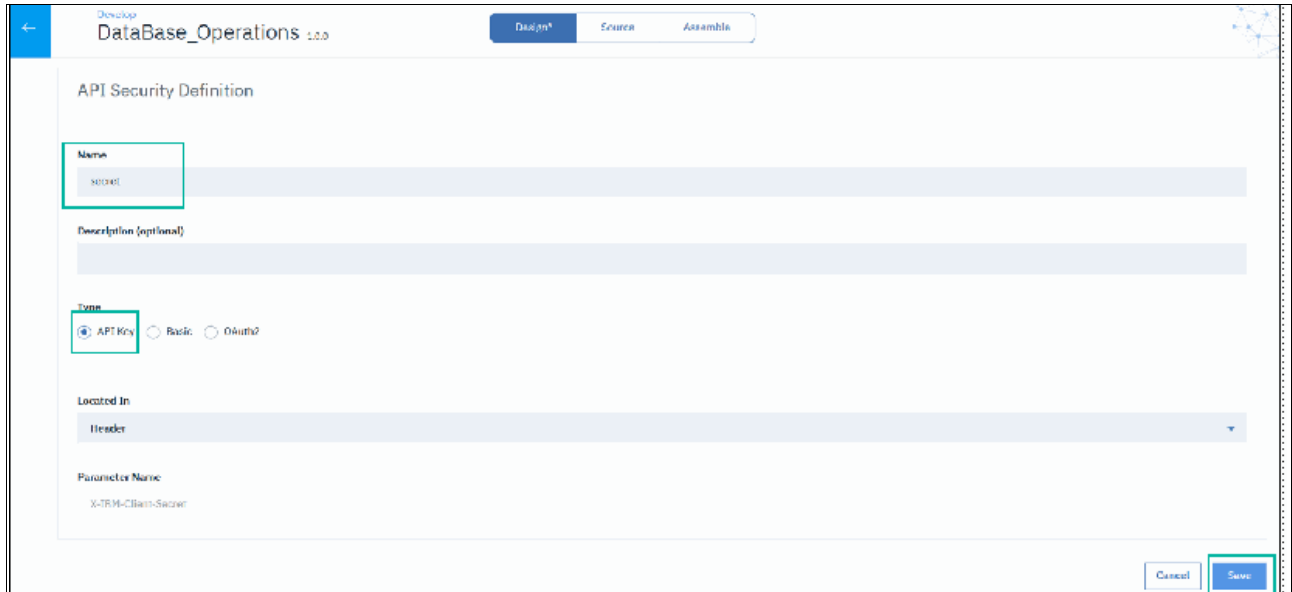


Figure 6-51 Defining the security 2

Your definitions should look like Figure 6-52 on page 204.

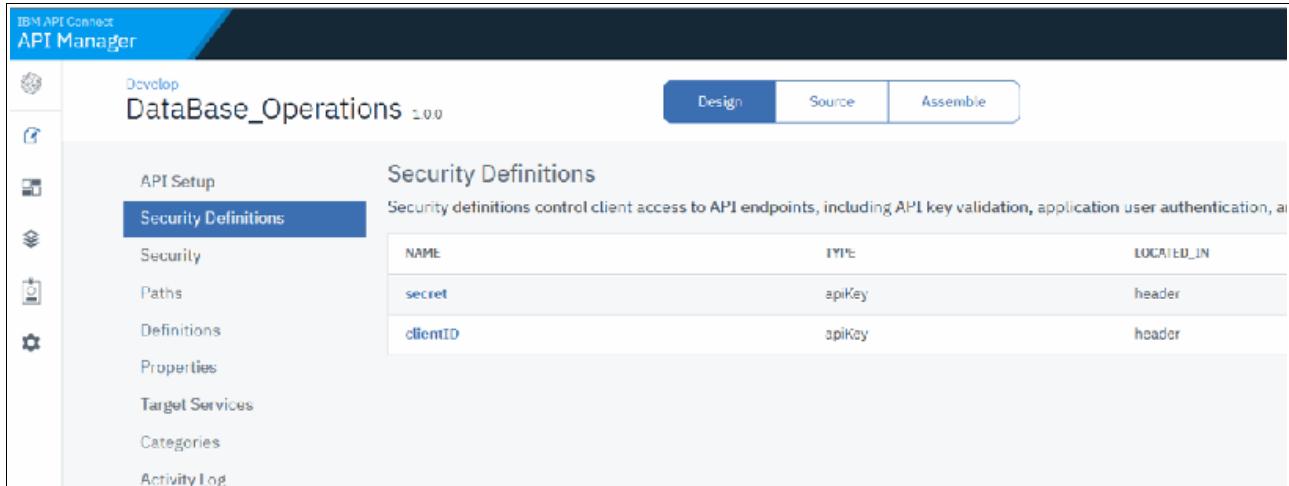


Figure 6-52 Defining the security 3

Publish the product

Perform the following steps to publish the product:

1. Click **Develop** from the left side menu, then click the ellipsis (...) beside the API that you want to publish and click **Publish**. See Figure 6-53.

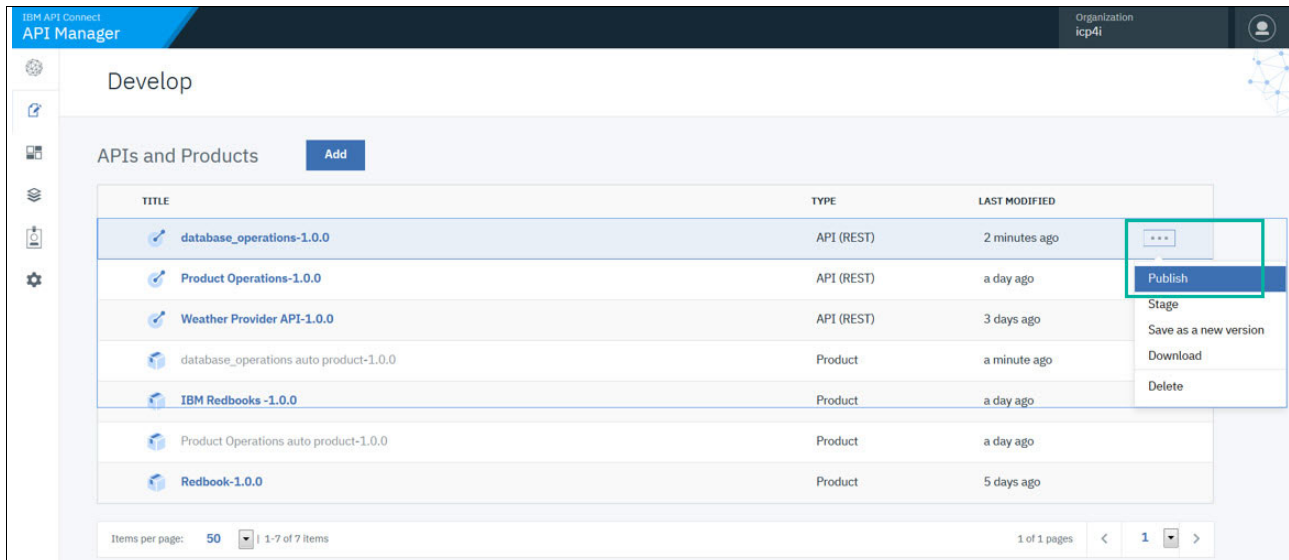


Figure 6-53 Publishing the product 1

2. Choose **New Product** and type in database_operation_product. Then click **Next**. See Figure 6-54.

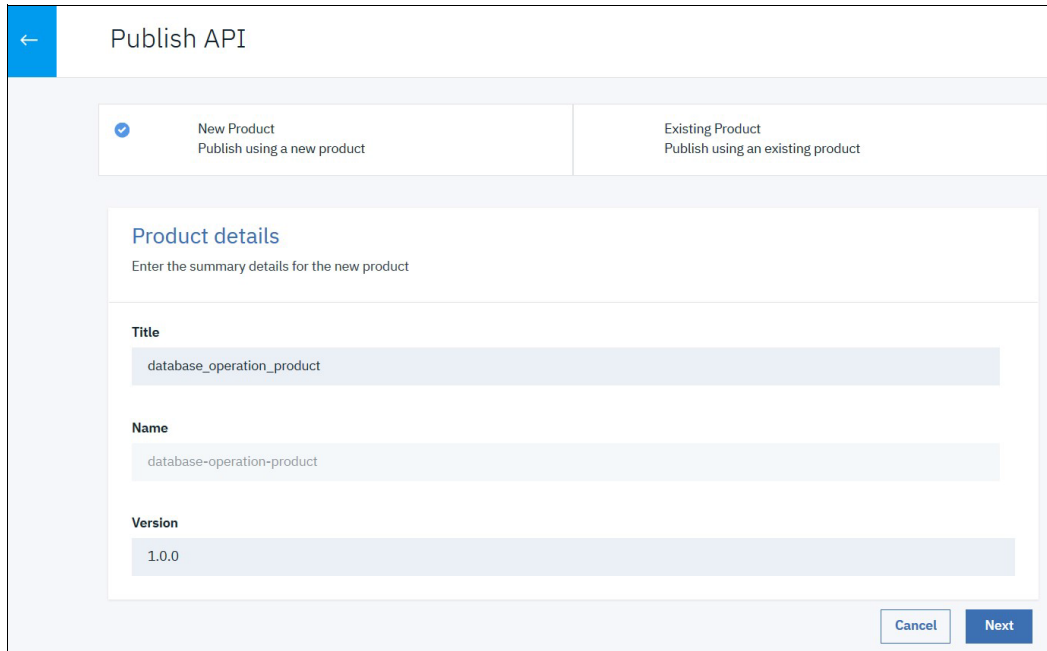


Figure 6-54 Publishing the product 2

3. Click **Publish** as shown in Figure 6-55.

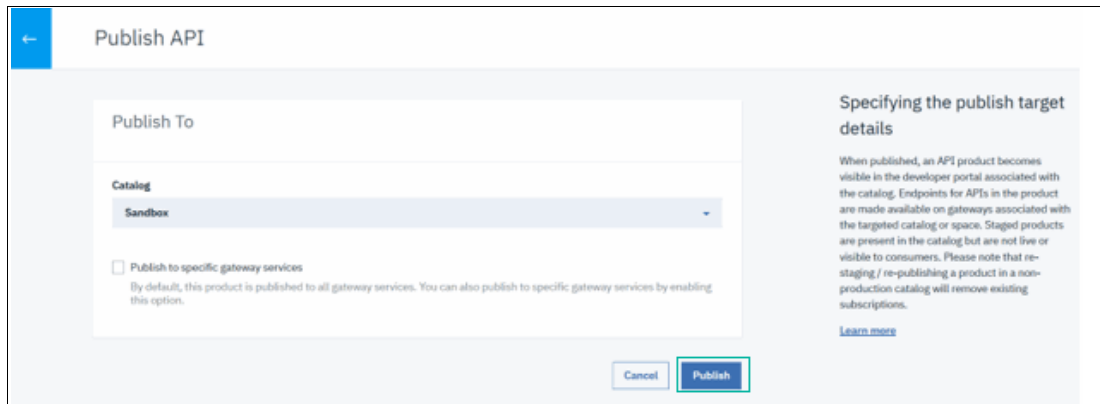


Figure 6-55 Publishing the product 3

Test the product

1. Now you can test the API. Go to **Assemble** and click the highlighted box in Figure 6-56 on page 206.

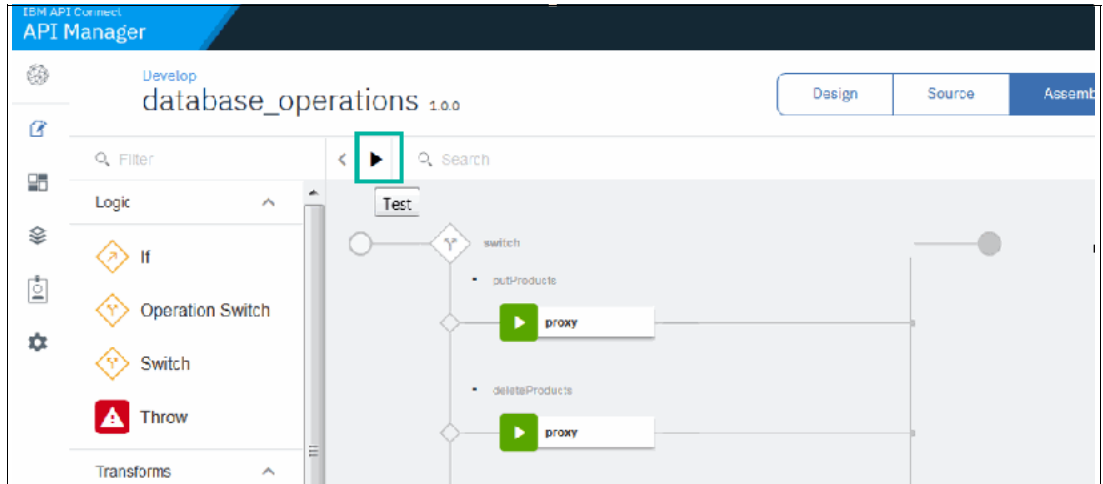


Figure 6-56 Testing the API 1

2. Click **Activate API** as shown in Figure 6-57.

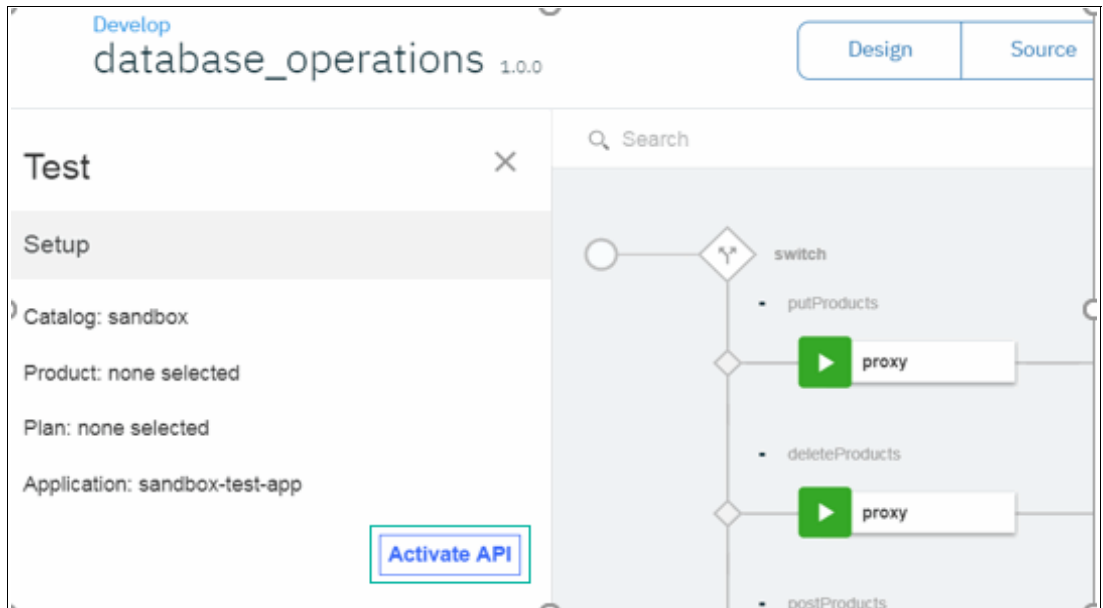


Figure 6-57 Testing the API 2

3. Choose the operation to test. Here, we try the get operation. See Figure 6-58.

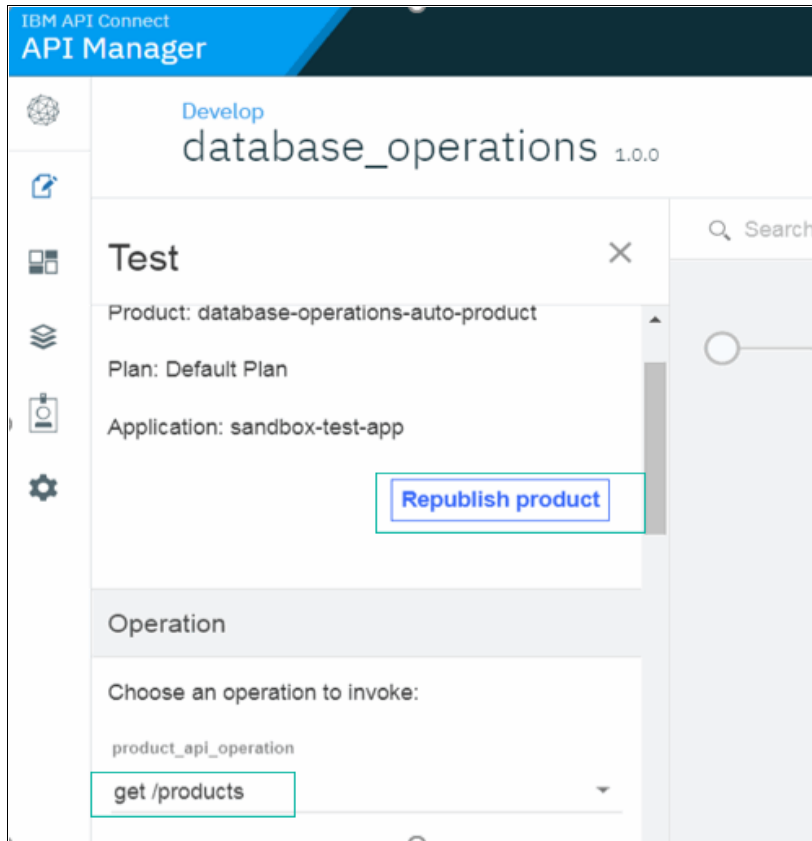


Figure 6-58 Testing the API 3

4. Click **Invoke**. You receive the response from the back end with 200 OK. See Figure 6-59 on page 208.

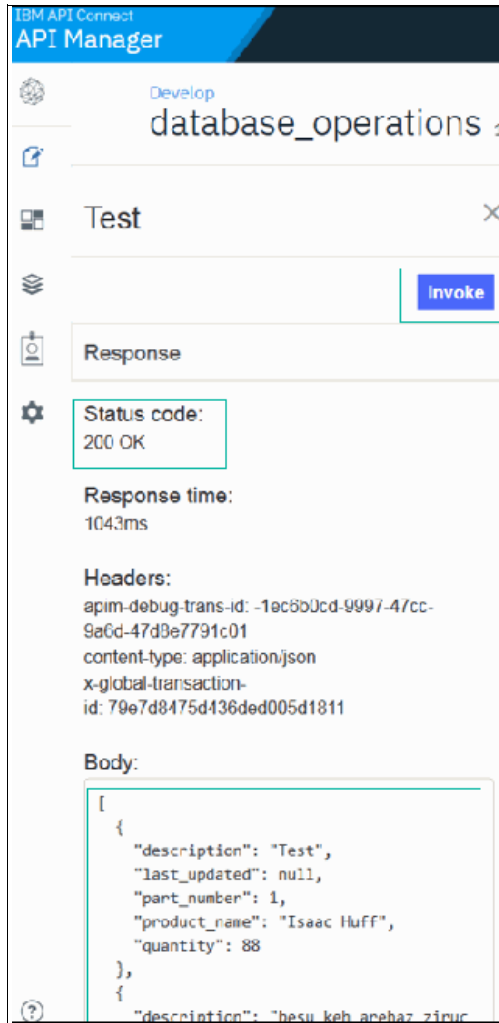


Figure 6-59 Testing the API (Response)

This was only a simple security using the API Key.

We have shown a basic invocation of the API using the internal testing mechanism. Of course, real consumers would first need to discover the API through the developer portal, then subscribe to use it. We cover this more formal discovery and subscription in “Subscribing to products” on page 296.

6.4 Messaging for reliable asynchronous data update commands

Important: Since the writing of this IBM Redbooks publication, the IBM Cloud Pak for Integration has embraced Kubernetes Operators (<https://coreos.com/operators/>). This significantly simplifies how components such as an Integration Server are installed and maintained, extending the features provided by Helm. There is more information and an excellent video demonstrating this new capability here:

<https://developer.ibm.com/integration/blog/2020/06/28/ibm-app-connect-operator-1-0-is-now-available/>

It does unfortunately mean that some of the instructions describing the deployment of App Connect Enterprise in containers within this section are now out of date, and will need to be adapted to the use of operators. We may well look to update the book, but in the mean time, refer to the product documentation to find information on the new features.

In previous sections, we explored how to move existing centralized ESB based integrations into an Agile Integration paradigm. To do this, we broke the different integrations apart and exposed them through API management. This is a good start, but as part of your overall application modernization strategy, new integration patterns will also emerge.

A good example is event-based programming models for updates. For reasons such as performance or availability of a data source, you might decide to move toward these models and away from traditional synchronous data updates.

In the first section, we deliberately split the traditional CRUD (create, read, update, delete) into separate models for commands. The separate models make changes to data (create, update, delete) and to the other operations that query (search/read) data, to enable them to be changed and scaled independently. However, they were still synchronous in nature, dependent on the datastore's performance and availability.

We can make use of this separation and independently refactor the change operations to be asynchronous, without touching the query path. In our case, we do this by introducing IBM MQ instead of HTTP as the mechanism for the update. Just to be clear what we mean by this, we are not talking about changing only the transport from HTTP to IBM MQ. We are also changing the interaction pattern from request/response to fire and forget. This way, after a request has been made to change data, we can respond immediately to the calling system that the request has been received. We do not have to wait for it to be completed. So, we are no longer dependent on the back-end systems availability or performance. IBM MQ's assured delivery means that we can be confident that it will eventually happen. Furthermore, we can throttle and control when the updates are applied, so that in busy periods they do not affect the performance of queries.

Clearly this model introduces challenges. We don't know exactly when the update will occur. And there might be other updates from other consumers, too. So, we can never be entirely sure of the status of the data in the back-end system. Nowadays we use the term *eventual consistency* to describe this situation. Clearly it is better suited to some business scenarios than others. In our example, we decided that the increased availability and response time on updates to our "product" data, and the potentially more consistent performance on queries, are more important than knowing that the data is 100% consistent all the time.

The *CQRS* (Command Query Responsibility Segregation) pattern has become popular in recent years. Data changes (commands) and reads (queries) are treated as separate

implementations, to improve reliability and performance. The integrations for these two halves were already separate, but they were both acting synchronously on the same data source. What we are doing in this section can be described as implementing the "command" part of this pattern. In other words, we change the synchronous data changes into a series of asynchronous commands. In later sections, we look at creating even more separation on the "query" side.

Figure 6-60 on page 210 illustrates this pattern.

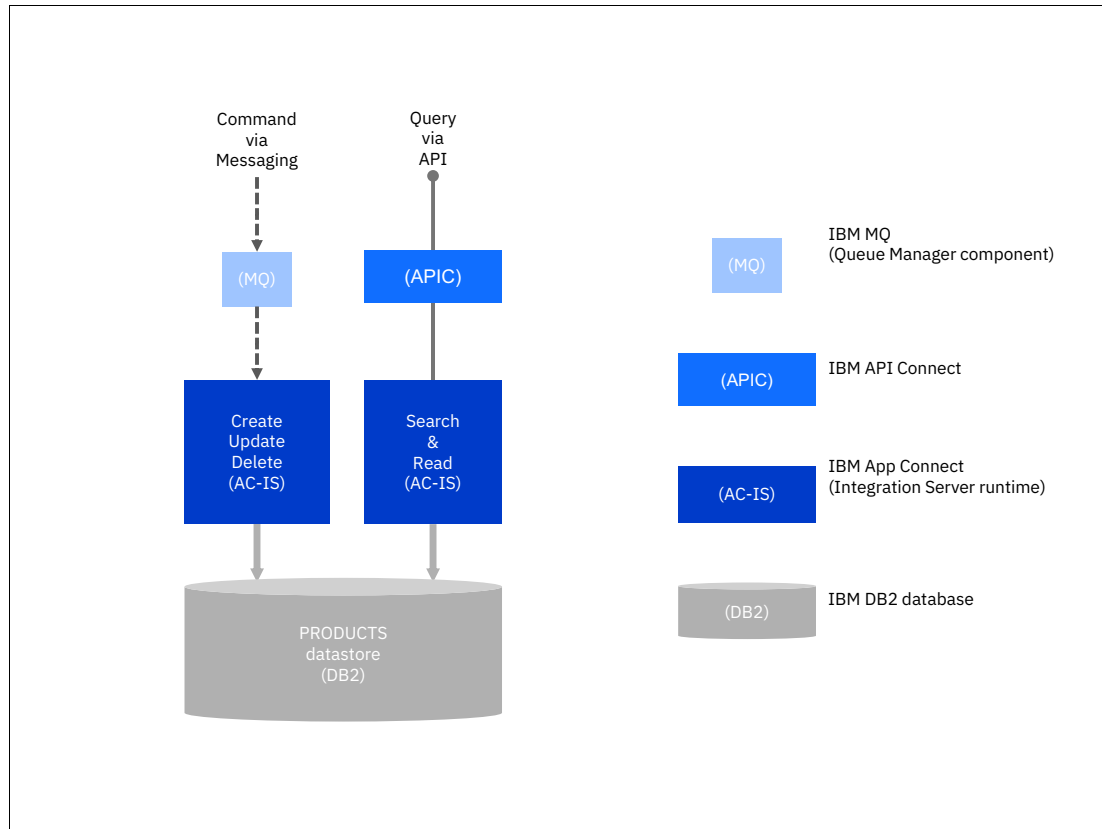


Figure 6-60 Command and Query pattern

In this section, we explore how you can use IBM MQ and IBM App Connect to implement the Command side of this pattern.

6.4.1 Enable create, update, delete via commands

One of the first choices you must make when implementing this side of the pattern is which protocol mechanism to use for sending the commands to the corresponding component.

Because Commands represent a specific action that must occur, we use a one-way "PUT" to an IBM MQ queue. This approach allows us to decouple the requester from the implementation. At the same time, it provides a reliable messaging platform that allows event collaboration among the different services. IBM MQ's assured "exactly once" delivery of messages is ideal here. And its ability to participate in a transaction with a database offers even more options as we discuss later.

For a clean design, you need three queues that represent each one of the commands. In addition, it is recommended to have two extra queues, one to store potential errors and a

second one to log the activity. The second queue can be replaced by any other logging framework that is available in the platform, but we use it here for illustration purposes.

In the next section we show you how to create an IBM MQ queue manager, and the necessary queues for this part of the solution. However, you might already have an existing IBM MQ queue manager (or already have the skills to create one). Example 6-2 on page 211 shows the list of IBM MQ commands that you need to create the corresponding queues and the needed authorization records:

Example 6-2 List of IBM MQ commands

```
DEF QLOCAL(DB.LOG)
DEF QLOCAL(DB.ERROR)
DEF QLOCAL(DB.CREATE) BOTHRESH(1) BOQNAME(DB.ERROR)
DEF QLOCAL(DB.UPDATE) BOTHRESH(1) BOQNAME(DB.ERROR)
DEF QLOCAL(DB.DELETE) BOTHRESH(1) BOQNAME(DB.ERROR)
SET AUTHREC PROFILE('DB.LOG') OBJTYPE(Queue) principal('user11') AUTHADD(ALL)
SET AUTHREC PROFILE('DB.ERROR') OBJTYPE(Queue) principal('user11') AUTHADD(ALL)
SET AUTHREC PROFILE('DB.CREATE') OBJTYPE(Queue) principal('user11') AUTHADD(ALL)
SET AUTHREC PROFILE('DB.UPDATE') OBJTYPE(Queue) principal('user11') AUTHADD(ALL)
SET AUTHREC PROFILE('DB.DELETE') OBJTYPE(Queue) principal('user11') AUTHADD(ALL)
```

Notice that we are taking advantage of the backout feature in IBM MQ to handle potential poison messages. Poison messages are ones that cannot be processed by the receiving system for some reason, but that we do not want to lose until we have had the opportunity to review them. This is also useful for transactional requirements that are discussed later in the section.

6.4.2 Deploy and configure Queue Manager

Now that you have decided to use an asynchronous messaging model, we can leverage the messaging capabilities that are provided by the Cloud Pak for Integration via IBM MQ.

1. Start creating a new instance. For that you can go to the Platform Navigator page and select **Add new instance** from the MQ tile as shown in Figure 6-61.

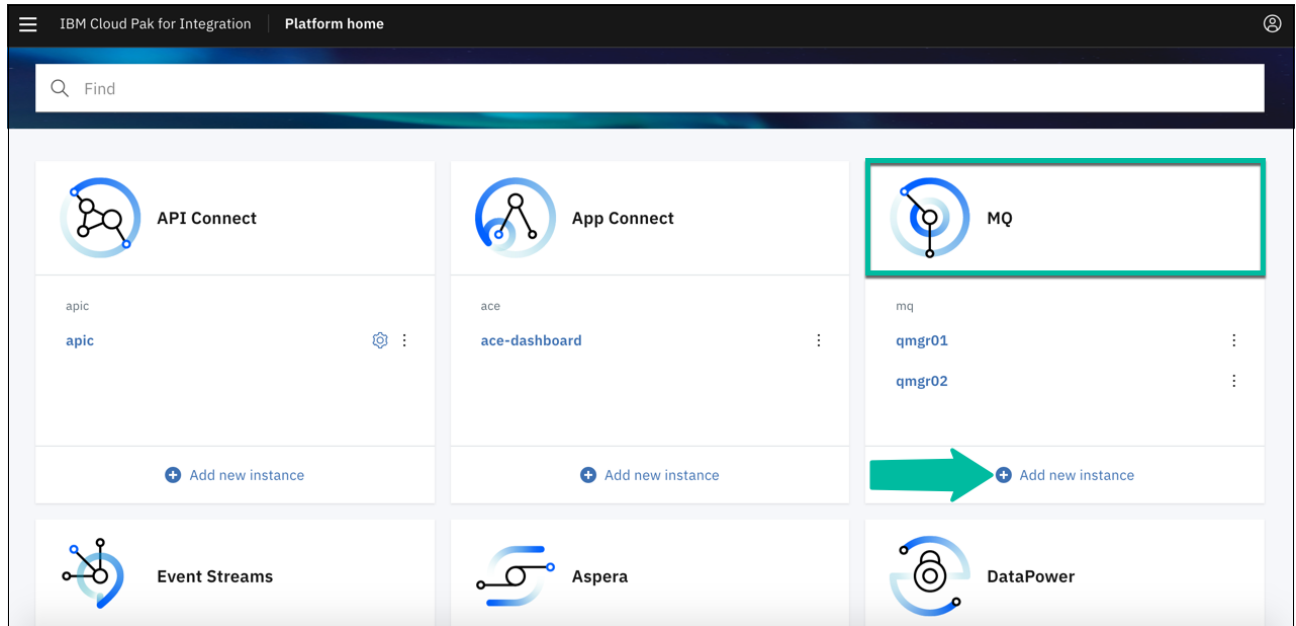


Figure 6-61 Creating a new IBM MQ instance - 1

2. You see the following pop-up window (Figure 6-62 on page 212) that provides a brief explanation about some prerequisites for deployment of IBM MQ. This is something that was usually configured at installation time, but you can check with your administrator, as suggested, to validate. After you confirm that your Cloud Pak for Integration is properly configured you can click **Continue**.

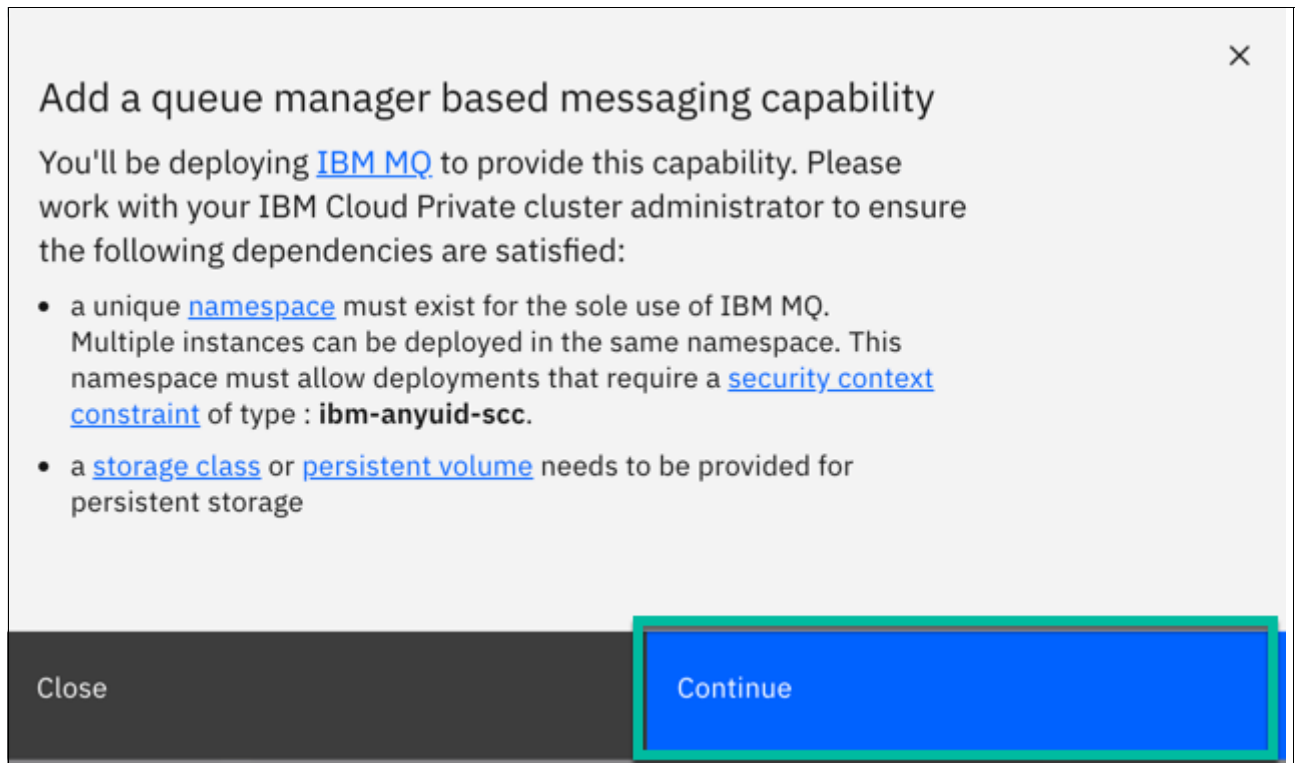


Figure 6-62 Creating a new IBM MQ instance - 2

3. This launches the helm chart that guides you through the deployment process. In the first section of the form, you are required to enter the name of the Helm release, and the namespace and cluster where the queue manager will be deployed. For this scenario, we used the following values:

- ▶ **Helm release name:** mqicp4
- ▶ **Target namespace:** mq
- ▶ **Target cluster:** local-cluster

You need to check the license box where you confirm that you have read and agreed to the licensing agreement. Figure 6-63 on page 213 shows the information:

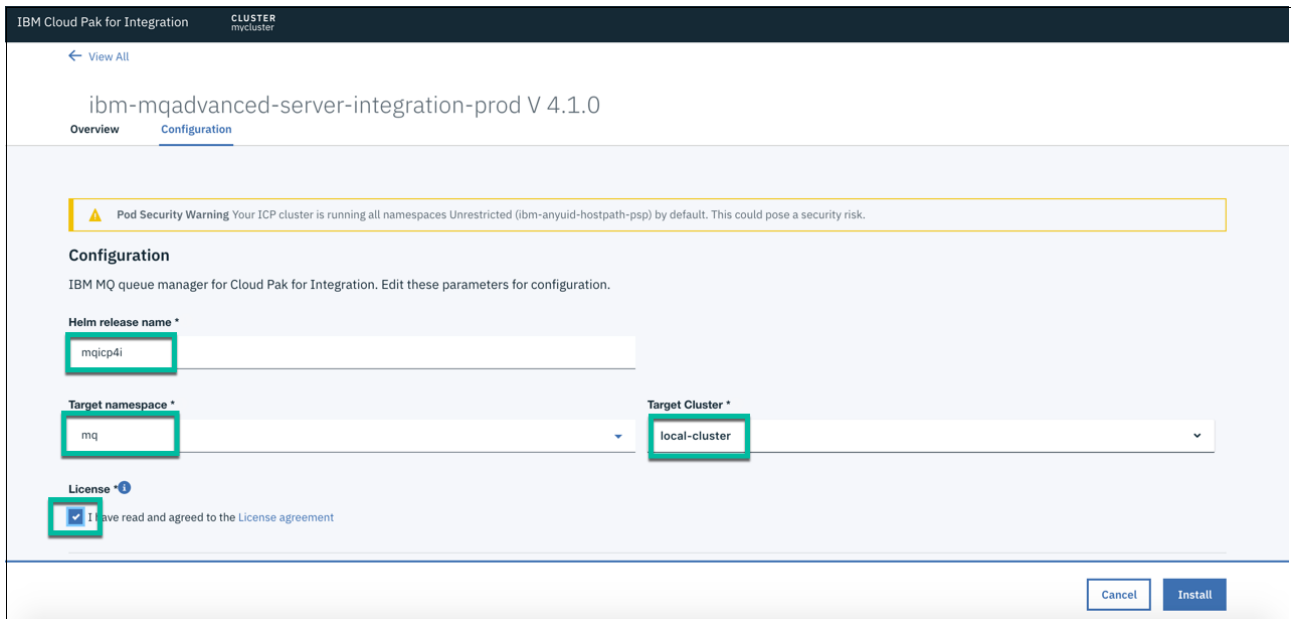


Figure 6-63 Creating a new IBM MQ instance - 3

4. You can scroll down to access the next set of fields starting with Pod Security. In this section you need to provide only the FQN of the proxy that gives access to your cluster. See Figure 6-64.

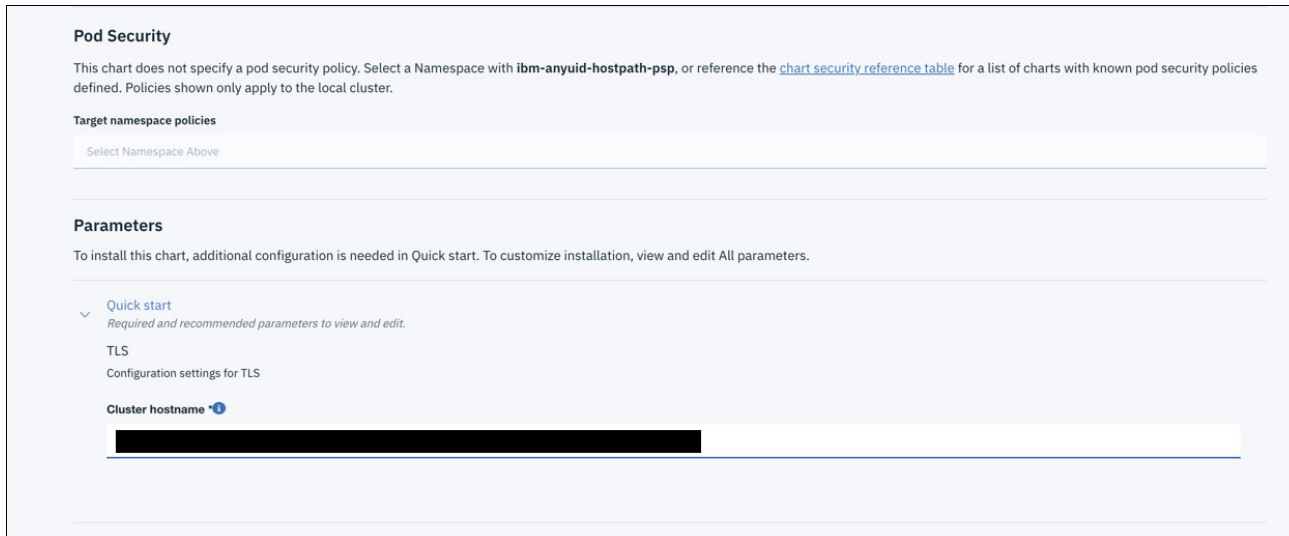


Figure 6-64 Creating a new IBM MQ instance - 4

Then you scroll down and expand the All Parameters section to review and modify the rest of the parameters. You can clear the **Production Usage** field, because this deployment is for test purposes. You can accept the default values for “Image repository” and “Image tag”, unless you want to use your own image. We discuss this scenario later on. Enter the value of the secret with the credentials to access your registry in order to be able to pull the images. In our test environment it is called `entitled-registry`. And we recommend that you select **Always** for the image pull policy, so that you are sure to always get the most recent image in the registry. But you can use the other options if needed. Figure 6-65 on page 214 shows the screen with the values:

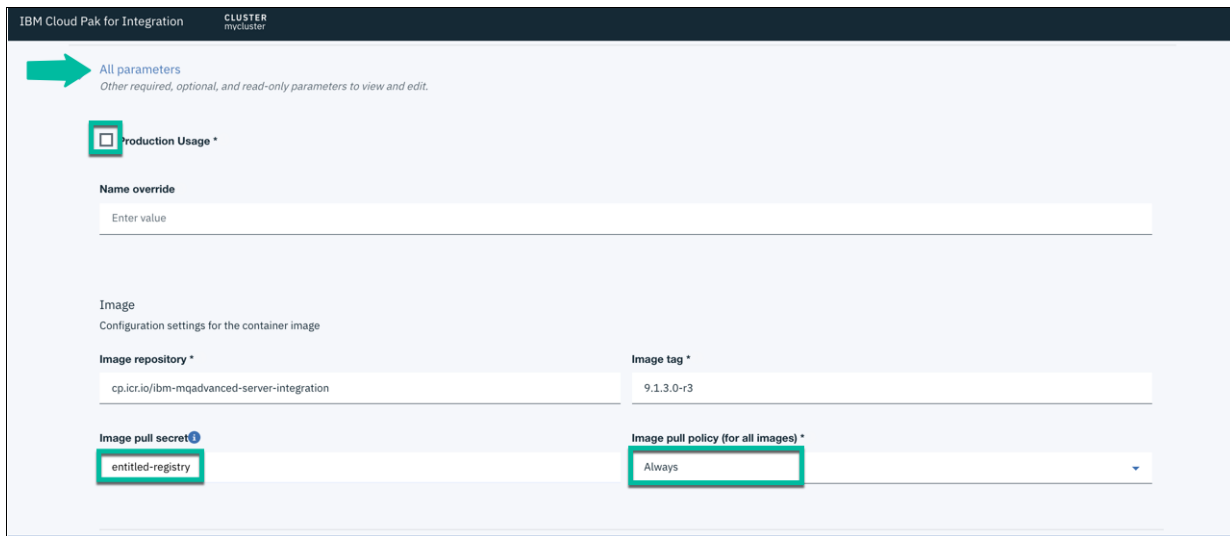


Figure 6-65 Creating a new IBM MQ instance - 5

5. You can keep the IBM Cloud Pak for Integration section with the default values as shown in Figure 6-66.

IBM Cloud Pak for Integration CLUSTER
mycluster

IBM Cloud Pak for Integration
Configuration settings for IBM Cloud Pak for Integration

Namespace where the platform navigator is installed *

integration

Single sign-on
Configuration settings for single sign-on

Registration image repository * **Registration image tag ***

cp.icr.io/ibm-mq-oidc-registration 2.1.2

Web admin users *

admin

Unique user identifier *

sub

Figure 6-66 Creating a new IBM MQ instance - 6

- In the next section, select the **Generate Certificate** checkbox to get a new certificate for the queue manager. The cluster hostname field is prepopulated with the value that you entered in the first part of the form. Figure 6-67 on page 215 shows the form:

IBM Cloud Pak for Integration CLUSTER
mycluster

TLS
Configuration settings for TLS

Generate Certificate

Cluster hostname * **Secret name**

[Redacted] Enter value

Metadata
Additional metadata to be added to resources

Additional labels

Enter object in YAML syntax:
- key: value

Figure 6-67 Creating a new IBM MQ instance - 7

- The next section in the form is the particular relevance for IBM MQ. By definition the storage in a container is ephemeral. In other words, if for some reason the pod where the container is running dies, the storage that is reserved for the container is also destroyed. And that behavior doesn't fit well with a resource manager like IBM MQ. With MQ, you can

have persistent messages that should be preserved in case of a server or queue manager failure.

The good news is that specialized elements allow you to externalize the storage that is assigned to a pod (container). Therefore, you preserve the information (queues, messages, and so on) that is created by the queue manager.

The specific field to configure this is called “Enabled persistence.” For the test scenario we have cleared this field, but for a production environment you probably must enable it. After you decide to enable this option, you can dynamically allocate the required storage. For this, you must select the **Use dynamic provisioning** box. In our case, we can clear the box, because we didn’t enable persistent storage.

If you do not want to use dynamic provisioning, you can still enable persistence, but you must create the persistence volume claim (PVC) beforehand and provide the corresponding name. If you opted for dynamic provisioning you must provide the proper storage class name. Figure 6-68 on page 216 shows the four field reflecting our assumption that no persistence is required.

IBM Cloud Pak for Integration CLUSTER mycluster

Persistence
Configuration settings for Persistent Volumes

Enable persistence *

Use dynamic provisioning ⓘ

Data PVC
Configuration settings for the main Persistent Volume Claim

Name *
data

Storage Class name
Enter value

Size *
2Gi

Figure 6-68 Creating a new IBM MQ instance - 8

8. The next section gives you the option to separate the Logs and Queue Manager configuration settings in different persistence volume claims. This approach is similar to what you would do with a regular queue manager and the file system that is associated. But in this case everything is parameterized. In our case, we left the boxes cleared because we decided not to enable persistence storage. See Figure 6-69 on page 217.

Log PVC
Configuration settings for the transaction logs Persistent Volume Claim

Enable separate storage for transaction logs *

Name * Storage Class name

Size *

QM PVC
Configuration settings for the queue manager data Persistent Volume Claim

Enable separate storage for queue manager data *

Name * Storage Class name

Figure 6-69 Creating a new IBM MQ instance - 9

- The next section will allow you to assign the resources (CPU and memory) that will be assigned to the queue manager. For testing purposes we will use the default values, but for a production environment you can do a sizing exercise to assign the values that fit your needs. For the security section you can use the default values as shown in Figure 6-70 on page 217.

Resources
Configuration settings for specifying required resources

CPU limit * Memory limit *

CPU request * Memory request *

Security
Configuration settings for security

File system group

Supplemental groups

Enter array in YAML syntax:
- item1
- item2

Figure 6-70 Creating a new IBM MQ instance - 10

- Make sure the box for the last parameter around security named **Initialize volume as root** is checked to avoid any issue concerning access to the file system that is assigned to the container. See Figure 6-71.

IBM Cloud Pak for Integration CLUSTER
mycluster

Initialize volume as root *

Queue manager
Configuration settings for the Queue Manager

Queue manager name ¹

mqicpd1

Enable multi-instance queue manager *

PKI
Certificates to be added to the queue manager

Keys

```
-
  name: default
  secret: {secretName: ibm-mq-tls-secret, items: [tls.key, tls.crt]}
```

Figure 6-71 Creating a new IBM MQ instance - 11

11. You can leave the rest of the form with the default values as shown in Figure 6-72 and Figure 6-73 on page 219.

Log
Configuration settings for the container logs

Error log format

JSON

Enable debug log output *

Metrics
Configuration settings for Prometheus metrics

Enable metrics *

Liveness probe
Configuration settings for the MQ liveness probe, which checks for a running Queue Manager

Initial delay (seconds) Period (seconds)

90 10

Figure 6-72 Creating a new IBM MQ instance - 12

Readiness probe
Configuration settings for the MQ readiness probe, which checks when the MQ listener is running

Initial delay (seconds)	Period (seconds)
<input type="text" value="10"/>	<input type="text" value="5"/>
Timeout (seconds)	Failure threshold
<input type="text" value="3"/>	<input type="text" value="1"/>

Operations Dashboard configuration
Settings for the IBM Cloud Pak For Integration Operations Dashboard

Enable Operations Dashboard *

OD agent image repository	OD agent image tag
<input type="text" value="cp.icr.io/icp4i-od-agent"/>	<input type="text" value="1.0.0"/>
OD agent liveness probe initial delay (seconds)	OD agent readiness probe initial delay (seconds)
<input type="text" value="10"/>	<input type="text" value="10"/>

Figure 6-73 Creating a new IBM MQ instance - 13

12. After you review all the parameters, click on Install to start the deployment. See Figure 6-74 on page 219.

IBM Cloud Pak for Integration CLUSTER
mycluster

OD agent image repository	OD agent image tag
<input type="text" value="cp.icr.io/icp4i-od-agent"/>	<input type="text" value="1.0.0"/>
OD agent liveness probe initial delay (seconds)	OD agent readiness probe initial delay (seconds)
<input type="text" value="10"/>	<input type="text" value="10"/>
OD collector image repository	OD collector image tag
<input type="text" value="cp.icr.io/icp4i-od-collector"/>	<input type="text" value="1.0.0"/>
OD collector liveness probe initial delay (seconds)	OD collector readiness probe initial delay (seconds)
<input type="text" value="10"/>	<input type="text" value="10"/>
OD tracing instance namespace	
<input type="text" value="Enter value"/>	

Figure 6-74 Creating a new IBM MQ instance - 14

13. After a few moments you receive the following message indicating that the deployment has started. See Figure 6-75.

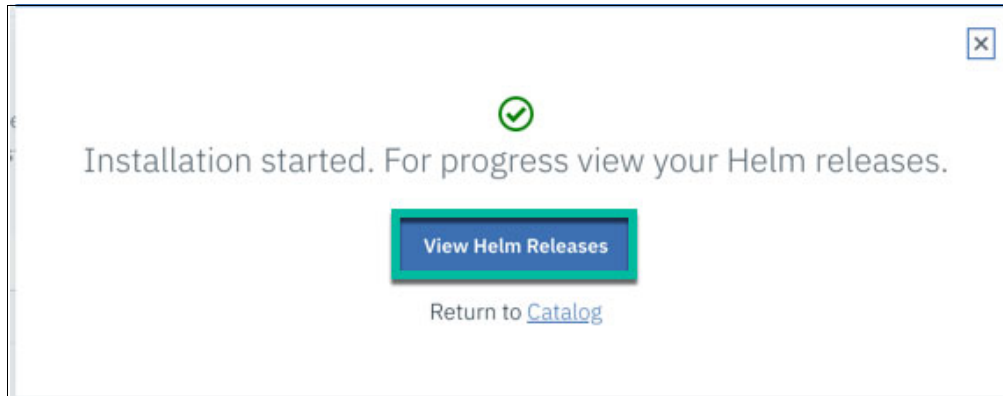


Figure 6-75 Creating a new IBM MQ instance - 15

14. Next we will monitor the progress of the deployment to confirm that there are no errors and start working with the queue manager. For that, you click on the **View Helm Release** button from the previous pop up window.

This action takes you to the Cloud Pak Foundation view for Helm Releases. After the new window is open, look for the name used for your deployment. If you are using the names suggested in the book it will be “mqjcp4i”, as shown in Figure 6-76 on page 220.

Helm Releases

Search

Name	Cluster Name	Namespace	Status	Chart Name	Current Version	Updated
qmgr03	local-cluster	mq	Deployed	ibm-mqadvanced-server-integration-prod	4.1.0	4 minutes ago
mqjcp4i	local-cluster	mq	Deployed	ibm-mqadvanced-server-integration-prod	4.1.0	10 days ago
ace-java	local-cluster	ace	Deployed	ibm-ace-server-icp4i-prod	2.1.0	12 days ago
ace-dashboard	local-cluster	ace	Superseded	ibm-ace-dashboard-icp4i-prod	2.1.0	30 days ago
ace-designer	local-cluster	ace	Deployed	ibm-ace-server-icp4i-prod	2.1.0	1 months ago
rf9ad2183d2	local-cluster	apic	Deployed	dynamic-gateway-service	1.0.39	1 months ago
r307b84ffe1	local-cluster	apic	Deployed	apic-analytics	2.0.0	1 months ago
rbc357bd8b	local-cluster	apic	Deployed	apic-portal	2.0.0	1 months ago
r09aaf73f9	local-cluster	apic	Deployed	apiconnect	2.0.0	1 months ago
r9a3cf2a2d0	local-cluster	apic	Deployed	cassandra-operator	1.0.0	1 months ago
apic	local-cluster	apic	Deployed	ibm-apiconnect-icp4i-prod	1.0.2	1 months ago

Figure 6-76 Creating a new IBM MQ instance - 16

15. After you find the deployment, click on the name to get the detail. In the new screen, scroll down to check the different objects that are part of the deployment. The one that we will explore in more detail is the StatefulSet, which includes the Pod with the actual queue manager process. But before moving to the next screen, write down the commands provided in the Notes section so that you can get the connection information to the queue manager. We will need this information when we work on the integration project.

See Figure 6-77 through Figure 6-80 on page 222.

View All

mqicp4i

Deployed Launch

Details and Upgrades

CHART NAME ibm-mqadvanced-server-integration-prod NAMESPACE mq	CURRENT VERSION 4.1.0 <small>Installed: November 21, 2019 → Release Notes</small>	AVAILABLE VERSION 5.0.0 ▲ <small>Released: November 28, 2019 → Release Notes</small>	<input type="button" value="Upgrade"/> <input type="button" value="Rollback"/>
---	--	---	---

Job

Name	COMPLETIONS	DURATION	Age
mqicp4i-ibm-mq-registration	1/1	14s	10d

Pod

Name	READY	Status	RESTARTS	Age
------	-------	--------	----------	-----

Figure 6-77 Creating a new IBM MQ instance - 17

Pod

Name	READY	Status	RESTARTS	Age
mqicp4i-ibm-mq-0	1/1	Running	0	10d
mqicp4i-ibm-mq-registration-qd2hc	0/1	Completed	0	10d

Role

Name	Age
mqicp4i-ibm-mq	10d

RoleBinding

Name	Age
mqicp4i-ibm-mq	10d

Service

Name	TYPE	Cluster IP	External IP	Port(s)	Age
------	------	------------	-------------	---------	-----

Figure 6-78 Creating a new IBM MQ instance - 18

Service

Name	TYPE	Cluster IP	External IP	Port(s)	Age
mqicp4i-ibm-mq-metrics	ClusterIP	172.21.172.141	<none>	9157/TCP	10d
mqicp4i-ibm-mq	NodePort	172.21.1.251	<none>	9443:31670/TCP,1414:31736/TCP	10d

ServiceAccount

Name	SECRETS	Age
mqicp4i-ibm-mq	2	10d

StatefulSet

Name	Desired	Current	Age
mqicp4i-ibm-mq	1	1	10d

Notes

```
Get the MQ Console URL by running these commands:
export CONSOLE_PORT=$(kubectl get services mqicp4i-ibm-mq -n mq -o jsonpath='{.spec.ports[?(@.port=="9443")].nodePort}')
export CONSOLE_IP=$(kubectl get configmap ibmcloud-cluster-info -n kube-public -o jsonpath='{.data.proxy_address}')
```

Figure 6-79 Creating a new IBM MQ instance - 19

mqicp4i-ibm-mq 2 10d

StatefulSet

Name	Desired	Current	Age
mqicp4i-ibm-mq	1	1	10d

Notes

```
Get the MQ Console URL by running these commands:
export CONSOLE_PORT=$(kubectl get services mqicp4i-ibm-mq -n mq -o jsonpath='{.spec.ports[?(@.port=="9443")].nodePort}')
export CONSOLE_IP=$(kubectl get configmap ibmcloud-cluster-info -n kube-public -o jsonpath='{.data.proxy_address}')
echo https://$CONSOLE_IP:$CONSOLE_PORT/ibmmq/console

Get the MQ connection information for clients outside the cluster by running these commands:
export MQ_PORT=$(kubectl get services mqicp4i-ibm-mq -n mq -o jsonpath='{.spec.ports[?(@.port=="1414")].nodePort}')
export MQ_IP=$(kubectl get configmap ibmcloud-cluster-info -n kube-public -o jsonpath='{.data.proxy_address}')
echo $MQ_IP:$MQ_PORT

The MQ connection information for clients inside the cluster is as follows:
mqicp4i-ibm-mq.svc:1414
```

Figure 6-80 Creating a new IBM MQ instance - 20

16. After you click on the **StatefulSet** link you see the following screen (Figure 6-81 on page 223) with the details. From there you can drill down in the pod to review the events it has produced.

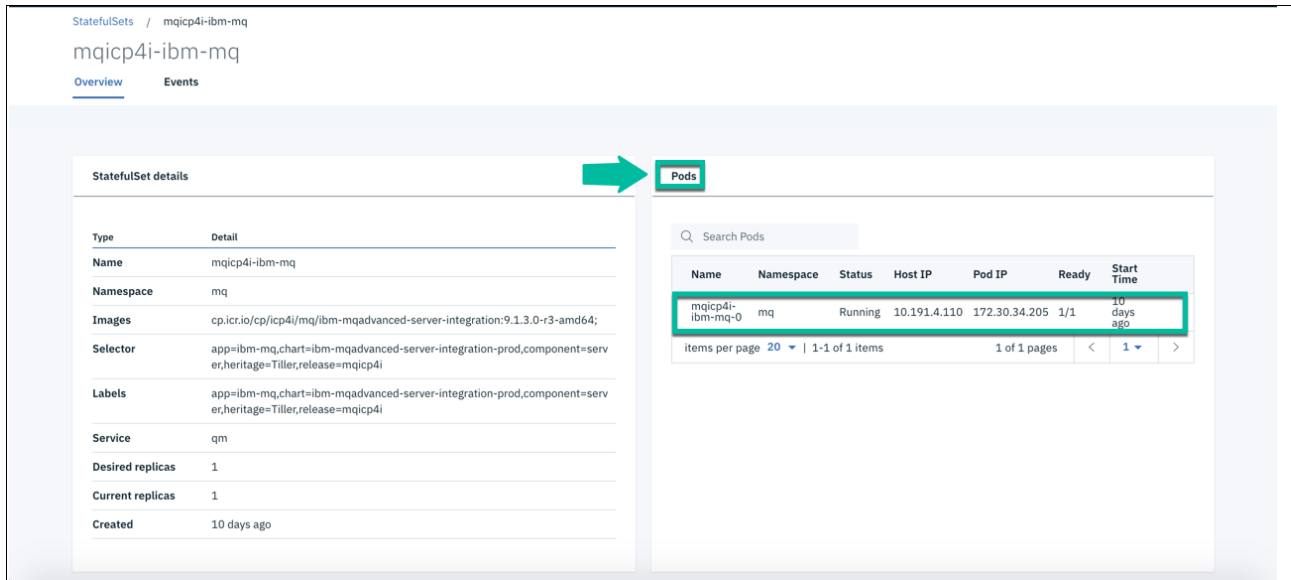


Figure 6-81 Creating a new IBM MQ instance - 21

17. In the Pod screen, check the Status to confirm that the queue manager is running. You could have seen the status from previous screens, but we want to show how you can navigate to the pod for potential troubleshooting situations to review the events that were produced during startup. See Figure 6-82 on page 223.

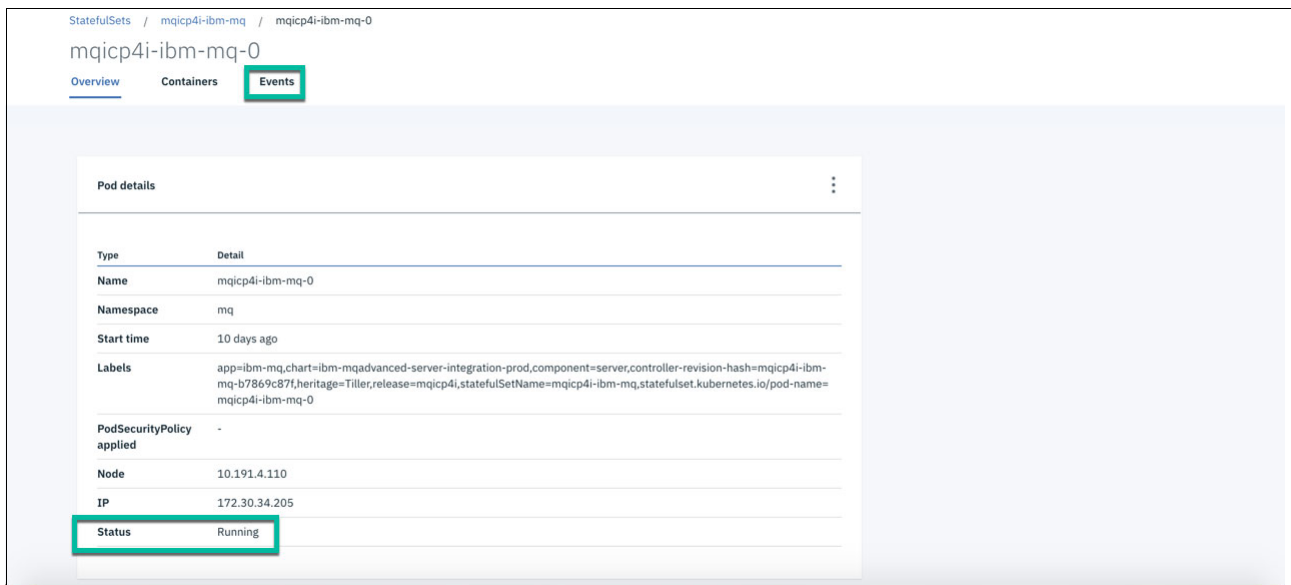


Figure 6-82 Creating a new IBM MQ instance - 22

18. When you have confirmed the queue manager is up and running, go back to the window where you initiated the deployment and click **Done** to close the pop-up window.

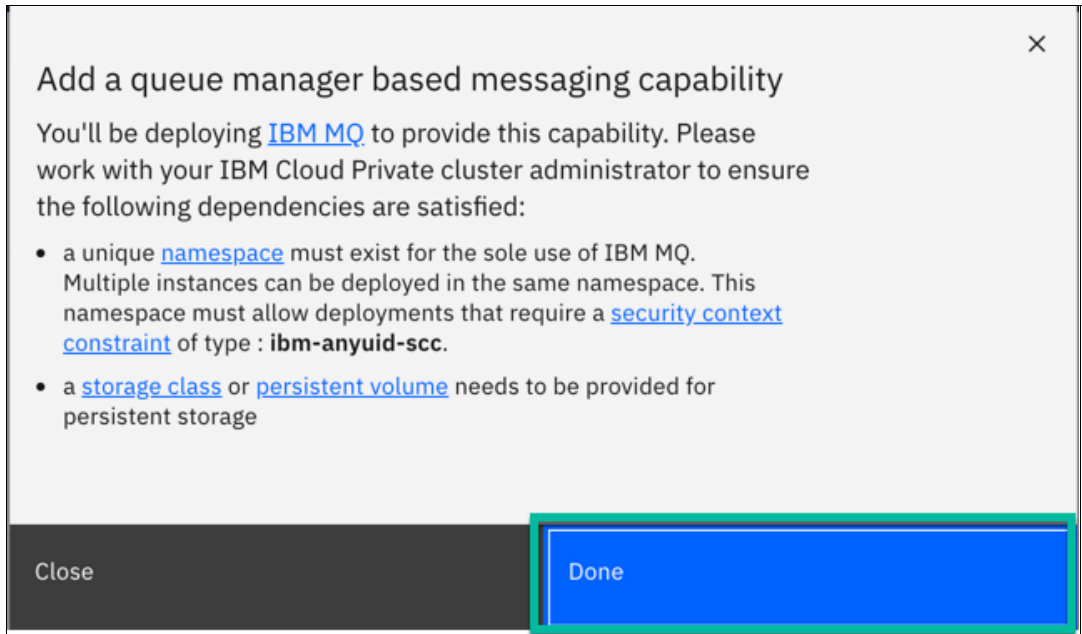


Figure 6-83 Creating a new IBM MQ instance - 23

19. After you close the pop-up window, you see that the new queue manager is displayed in the MQ tile.

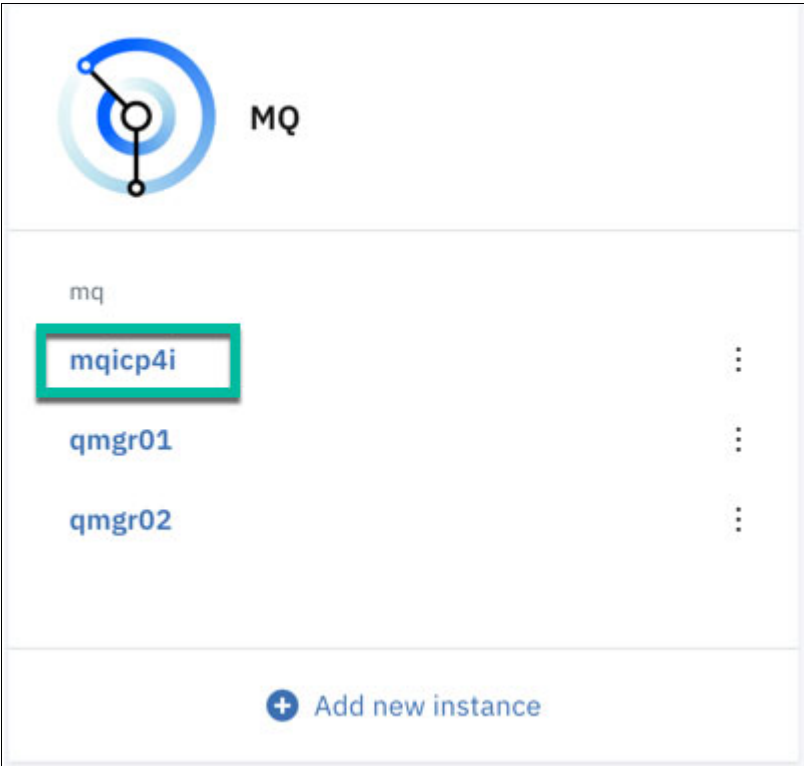


Figure 6-84 Creating a new IBM MQ instance - 24

6.4.3 Queue manager configuration

Now that the queue manager is up and running, you configure it with the objects that are required by the integration project. The objects were listed in the previous section as MQSC commands. In order to show several alternatives, we use the new MQ Web UI to create the same objects.

1. After you click in, the queue manager name is displayed in the MQ tile with all the queue managers you have available in the Cloud Pak for Integration. You are taken to the initial queue manager web UI. As you can see, only the Local Queue Manager widget is available. To configure the required objects, you need to add some other widgets. Click the **Widget** button as shown in Figure 6-85 on page 225.

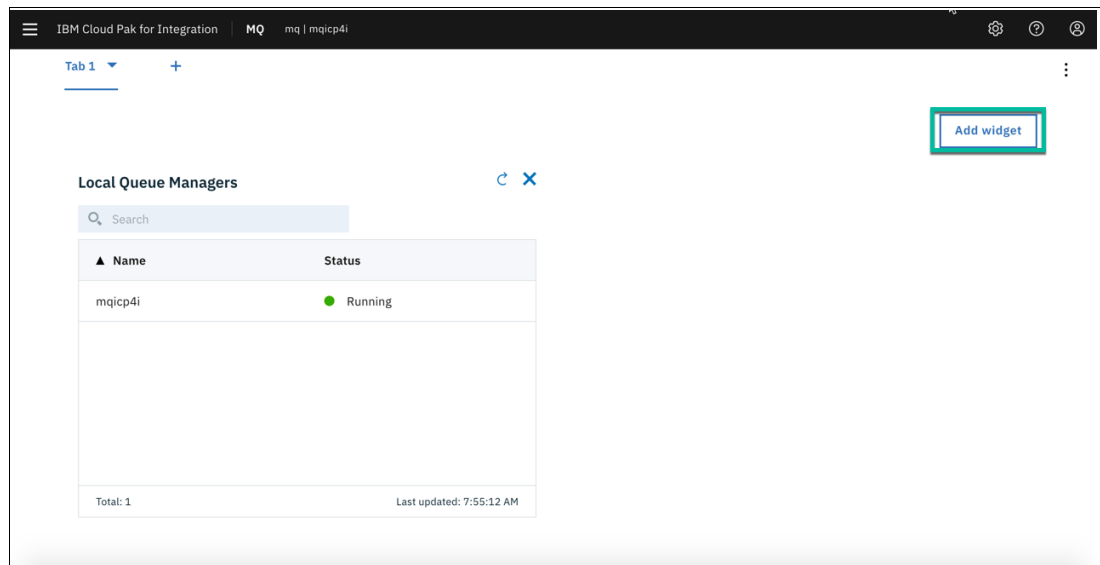


Figure 6-85 Queue manager configuration -1

2. The Add a new widget dialog is displayed where you can select the different IBM MQ objects that you want to administer. Click the **Queues** item as shown in Figure 6-87.

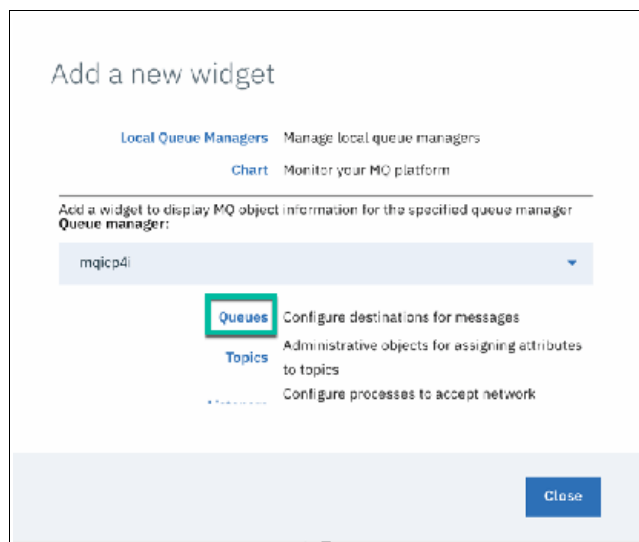


Figure 6-86 Queue manager configuration -2

3. The corresponding widget is added to the administration console as shown in Figure 6-87.

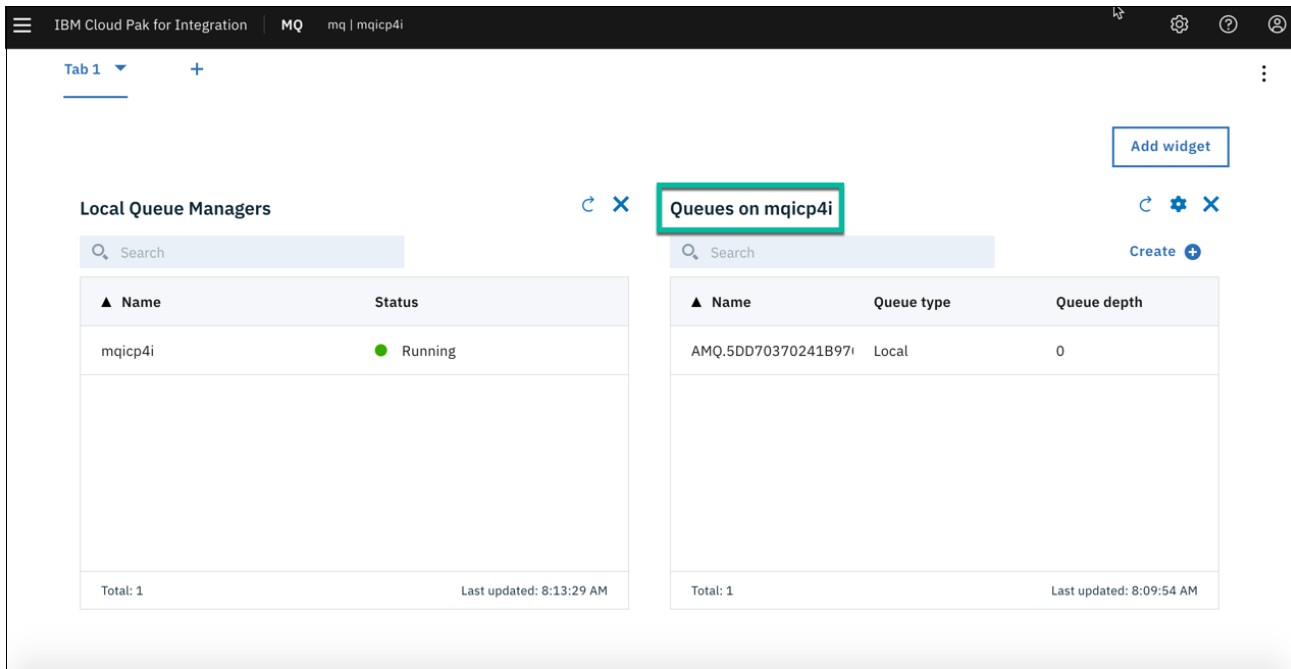


Figure 6-87 Queue manager configuration -3

4. Repeat the same process to add the Listener and Channel widgets. The web UI will look like Figure 6-88 after you have added the widgets.

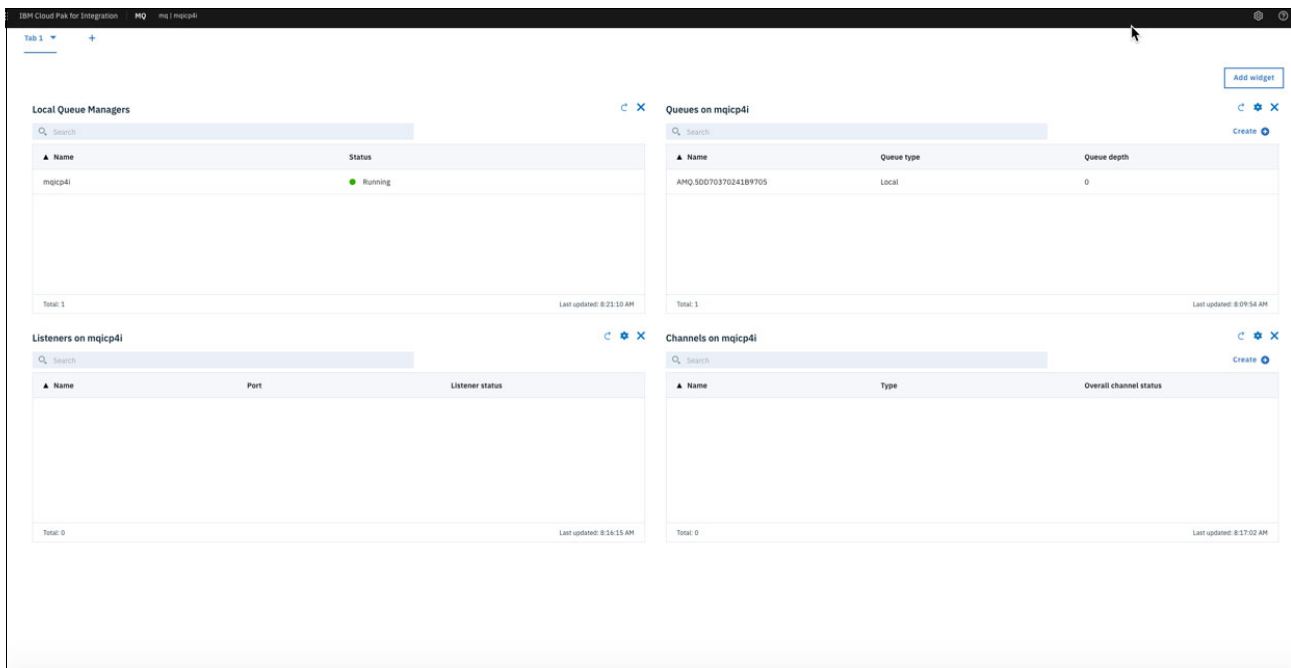


Figure 6-88 Queue manager configuration -4

5. We won't create any additional Listener, but we have added the widget to validate the default listener was properly configured when we deployed the queue manager. To do this you hover over the gear icon in the Listener widget and click on it to configure it, as shown in Figure 6-89 on page 227.

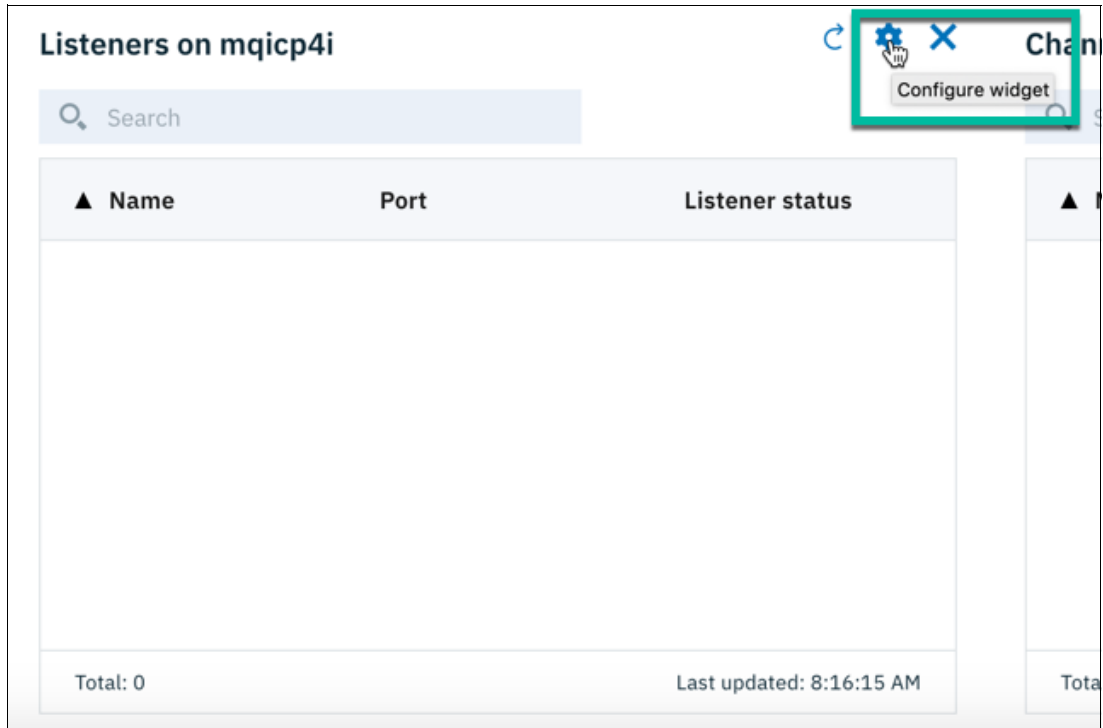


Figure 6-89 Queue manager configuration -5

6. The Listeners configuration dialog is displayed, where you select **Show System objects**, and then click **Save** as shown in the next figure.

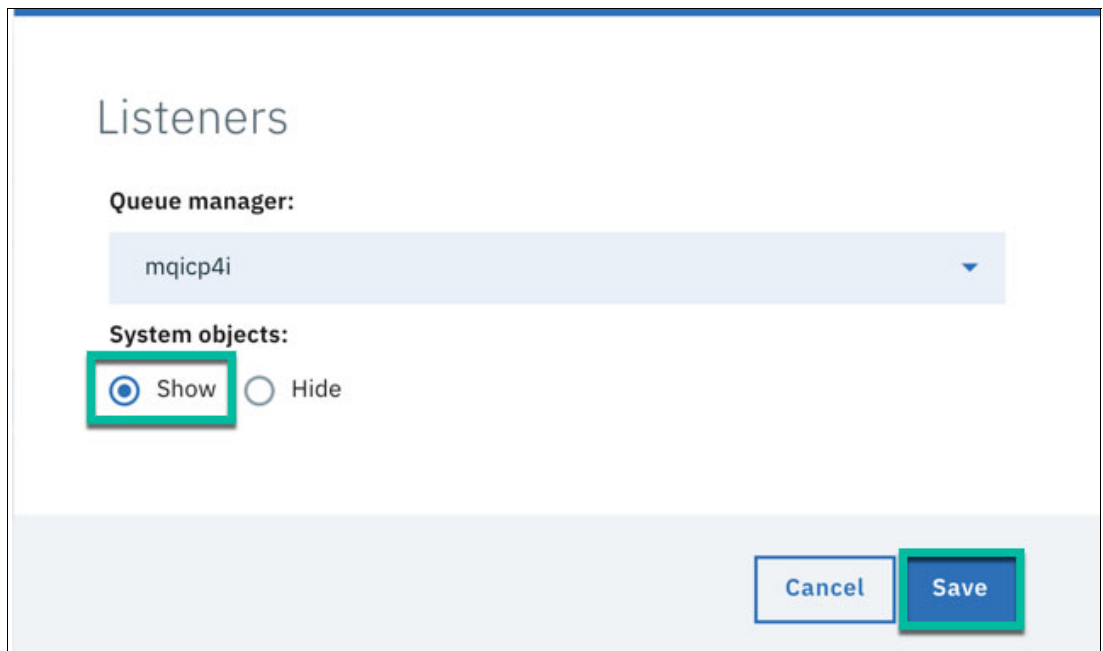


Figure 6-90 Queue manager configuration -6

7. The system objects are displayed, and you can see the default listener in the known port 1414 is already running. See Figure 6-91 on page 228.

Listeners on mqicp4i ↻ ⚙️ ✕

🔍 Search

▲ Name	Port	Listener status
SYSTEM.DEFAULT.LISTEN	0	● Stopped
SYSTEM.LISTENER.TCP.1	1414	● Running

Total: 2 Last updated: 8:33:19 AM

Figure 6-91 Queue manager configuration -7

Now that we know the listener is up and running, we can proceed to create the required Server Channel. This channel allows the connection between our integration flow running in IBM App Connect with the queue manager that we just deployed. To do this, click the **Create** button in the Channels widget as shown in Figure 6-92 on page 229.

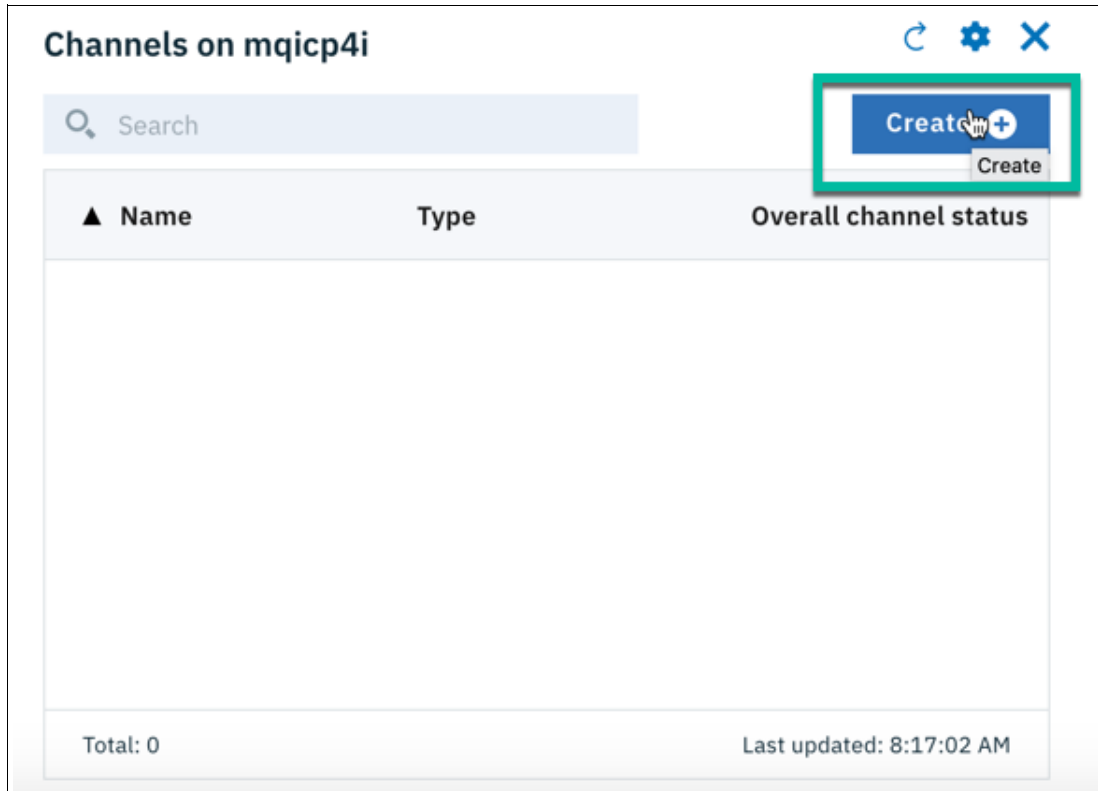


Figure 6-92 Queue manager configuration -8

8. The Create a Channel dialog box is displayed. In the window, select the type of channel to create to adjust the fields that we need to provide. In this case, we choose the Server-connection. Then, enter the name of the channel, in this case ACE.TO.MQ. But you can use another name. Just be sure to write it down, because you will need this value when you configure the IBM MQ policy in your integration flow. Finally, click **Create**. Figure 6-93 on page 230 illustrates the process.

Create a Channel

Channel name: * ⓘ

ACE.TO.MQ

Channel type: *

Server-connection

Cancel Create

Figure 6-93 Queue manager configuration -9

9. After a moment the widget will be updated to show the newly created channel. Notice that the channel is in an Inactive status. This status is normal, because we haven't deployed yet the integration flow that will use the channel. (You have the option to come back after you deploy the Integration flow to confirm that the status has changed to Active. See Figure 6-94 on page 231.

The screenshot shows a web interface titled "Channels on mqicp4i". At the top right, there are icons for refresh, settings, and close. Below the title is a search bar with a magnifying glass icon and the text "Search". To the right of the search bar is a "Create" button with a plus sign icon. The main content is a table with three columns: "Name", "Type", and "Overall channel status". The table contains one row with the following data: "ACE.TO.MQ" under Name, "Server-connection" under Type, and "Inactive" under Overall channel status, accompanied by a red dot icon. At the bottom of the table, there is a summary row showing "Total: 1" on the left and "Last updated: 8:48:52 AM" on the right.

Name	Type	Overall channel status
ACE.TO.MQ	Server-connection	Inactive

Total: 1 Last updated: 8:48:52 AM

Figure 6-94 Queue manager configuration -10

Now we can define the queues to use in our integration project. Similar to the channel, move to the Queues widget and click on the **Create** button as shown in Figure 6-95 on page 232.

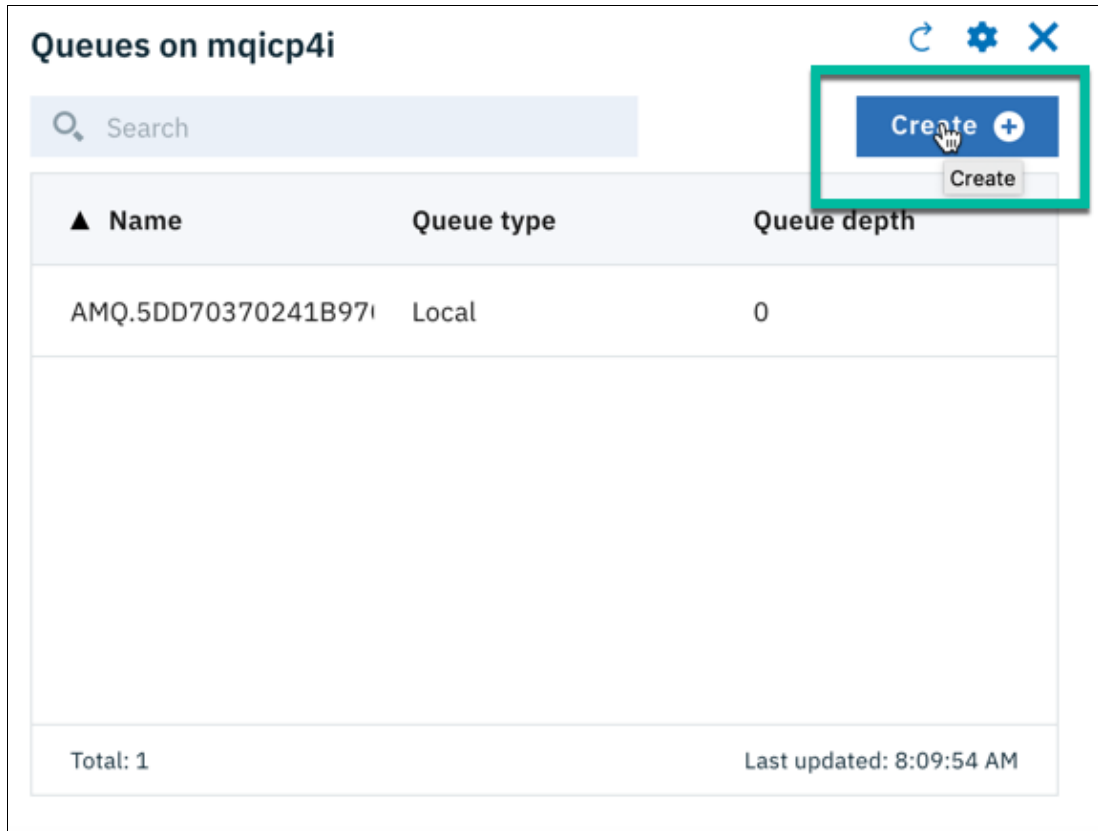


Figure 6-95 Queue manager configuration -11

10. The Create a Queue dialog is displayed where you provide the name and queue type for the definition of the required objects. Using the first queue in the list, we enter **DB.LOG** as the Queue name and **Local** as the Queue type. Then click the **Create** button as shown in Figure 6-96.

Create a Queue

Queue name: * i

DB.LOG

Queue type:

Local Remote Alias Model

Cancel Create

Figure 6-96 Queue manager configuration -12

11. After a moment the Queues widget is updated to include the newly created queue. See Figure 6-97.

Queues on mqicp4i

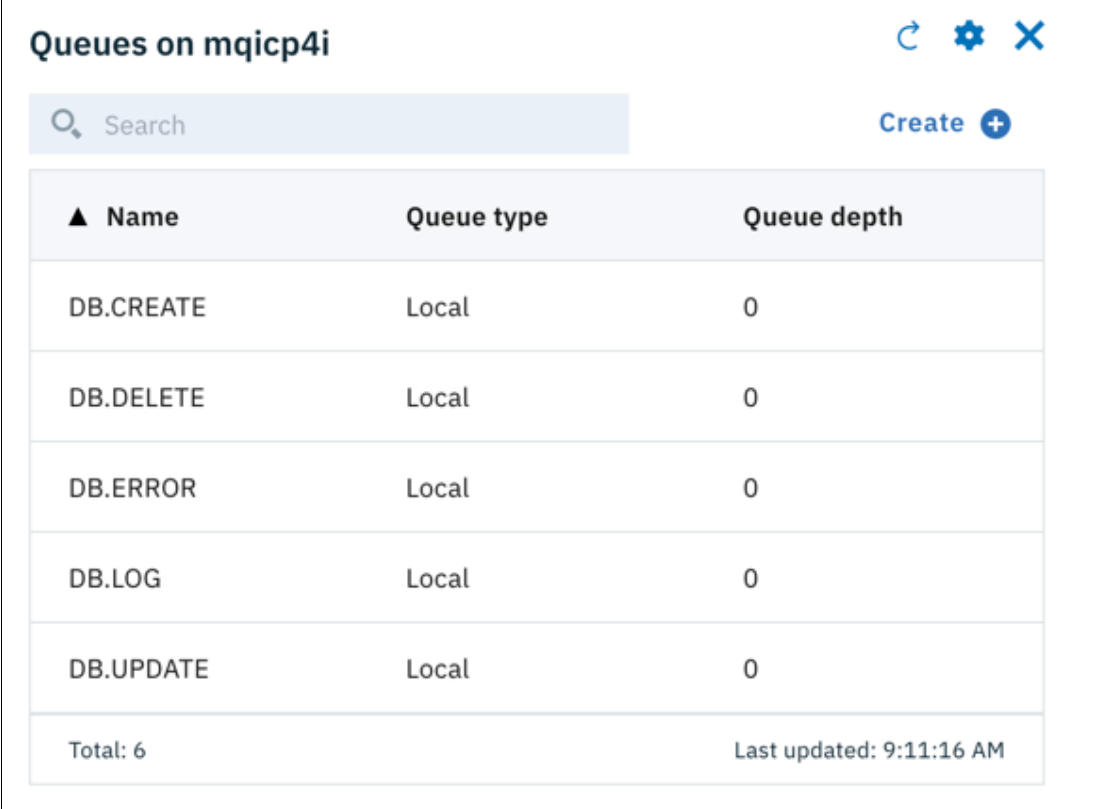
Search Create +

Name	Queue type	Queue depth
AMQ.5DD70370241B971	Local	13
DB.LOG	Local	0

Total: 2 Last updated: 9:05:40 AM

Figure 6-97 Queue manager configuration -13

12.Repeat the same process to create the rest of the queues. As mentioned before, the queue names are: DB.ERROR, DB.CREATE, DB.UPDATE, and DB.DELETE. All of them being local queues. At the end, you see something like Figure 6-98 on page 234.



▲ Name	Queue type	Queue depth
DB.CREATE	Local	0
DB.DELETE	Local	0
DB.ERROR	Local	0
DB.LOG	Local	0
DB.UPDATE	Local	0
Total: 6		Last updated: 9:11:16 AM

Figure 6-98 Queue manager configuration -14

13.To complete the configuration, we need to update the definition for the three queues that will process the commands to handle errors and cope with potential poison messages. In the Queues widget, select the **DB.CREATE** queue, hover over the Properties icon, and click on it as shown in Figure 6-99 on page 235.

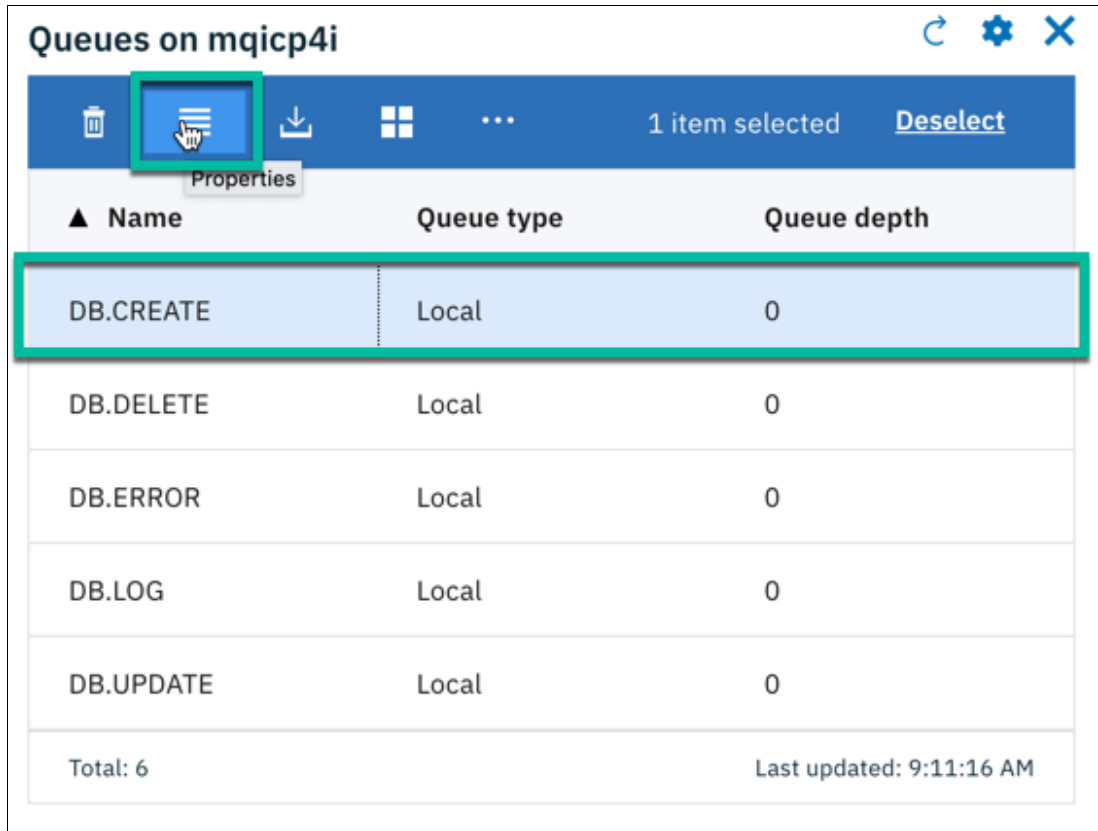


Figure 6-99 Queue manager configuration -15

14. This opens the Properties window for this particular queue. You can explore all the parameters, but for our scenario, we will move to the Storage section where we will update the fields Backout requeue queue and Backout threshold. Specifically, we use the **DB.ERROR** queue that we created in the previous step and assign a value of 1 to the threshold. Depending on your situation, you could use a different value for the threshold, but for the sample scenario we will consider an error after the first attempt. After you enter the values, click **Save** to update the queue as shown in Figure 6-100 on page 236.

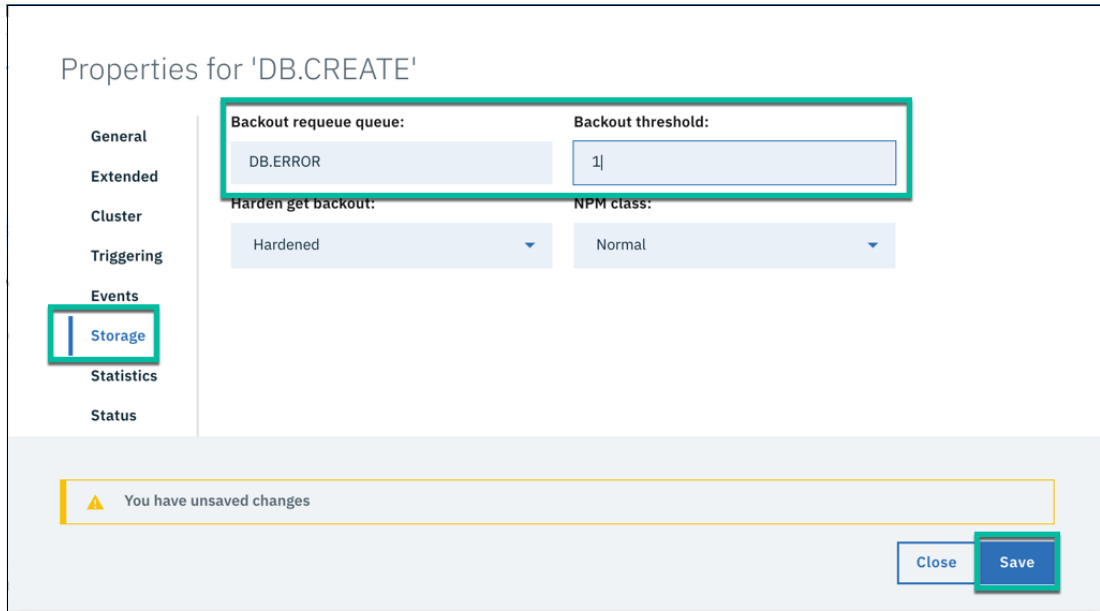


Figure 6-100 Queue manager configuration -16

15. The warning message that you have seen regarding unsaved changes now changes to a new message stating that the properties have been saved. Now you can click **Close** to return to the Queues widget and proceed to update the other two queues that we are missing.

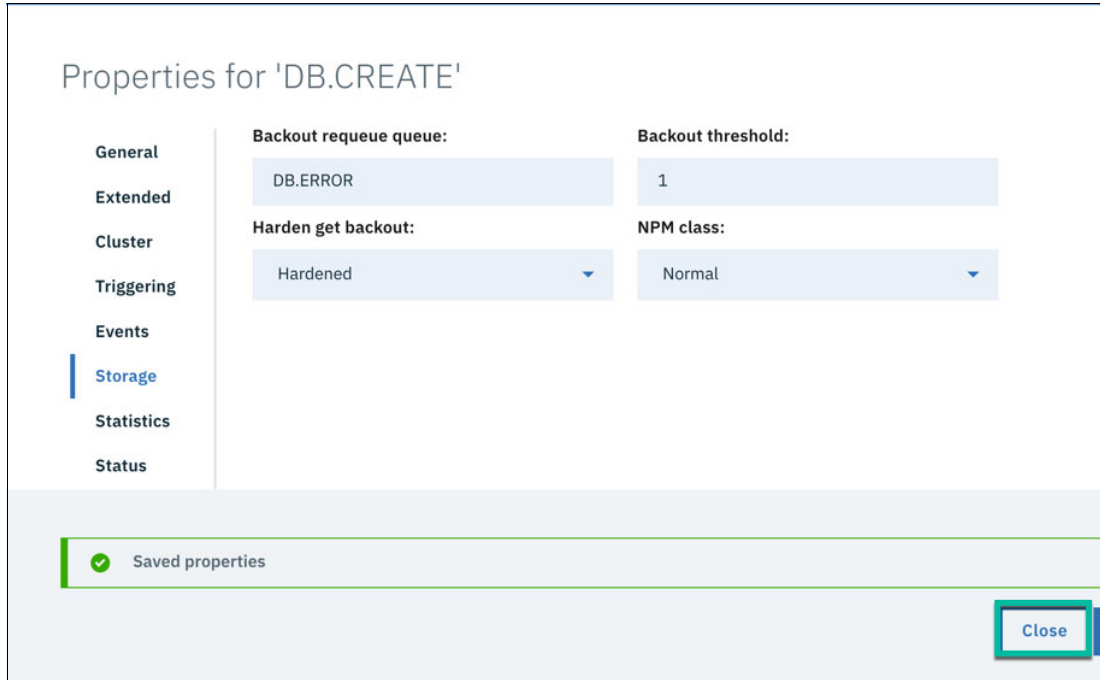


Figure 6-101 Queue manager configuration -17

To complete the configuration, repeat the process for queues DB.DELETE and DB.UPDATE.

16. After you update the queues you have all the objects that are required for the scenario. However, due to the security changes that were recently introduced by IBM MQ, you also

need to create the corresponding Authority Records. These records allow a user to interact with the queues.

Since this is a demonstration scenario, instead of working with Authority Records we show how to disable Connection Authentication at the queue manager level. Keep in mind, this is only for testing purposes. Disabling security is not recommended in a production environment.

Note: You can check 9.4, “Automation of IBM MQ provisioning using a DevOps pipeline” on page 594. That section describes how to create a queue manager using DevOps, alongside with the MQSC commands that are listed in the previous section to include the required security as part of your configuration.

17. To disable connection security checking, you need to modify the queue manager configuration. To do that, select the queue manager name from the Local Queue Managers widget and hover over the Properties button, clicking it as shown in Figure 6-102.

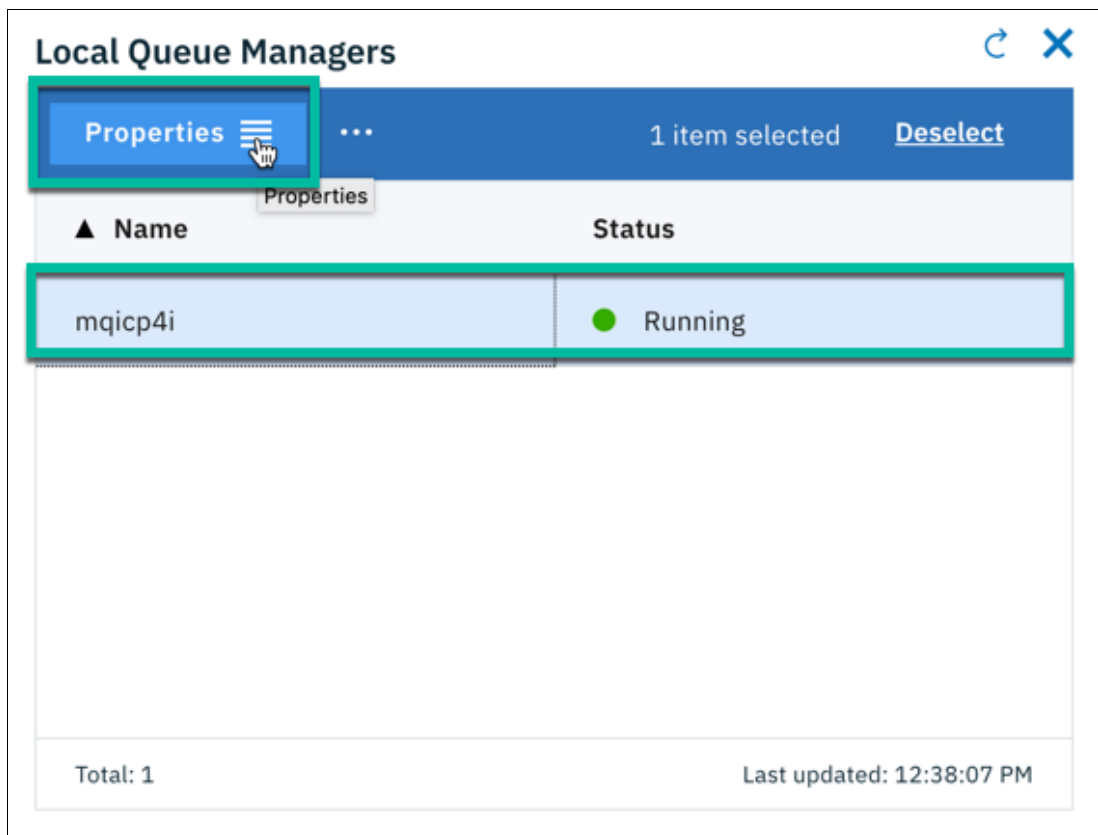


Figure 6-102 Queue manager configuration -18

18. This click opens the queue manager properties window. Navigate to the **Communication** section and change the CHLAUTH records field to the **Disabled** value. Then click **Save** to update the property has shown in Figure 6-103 on page 238.

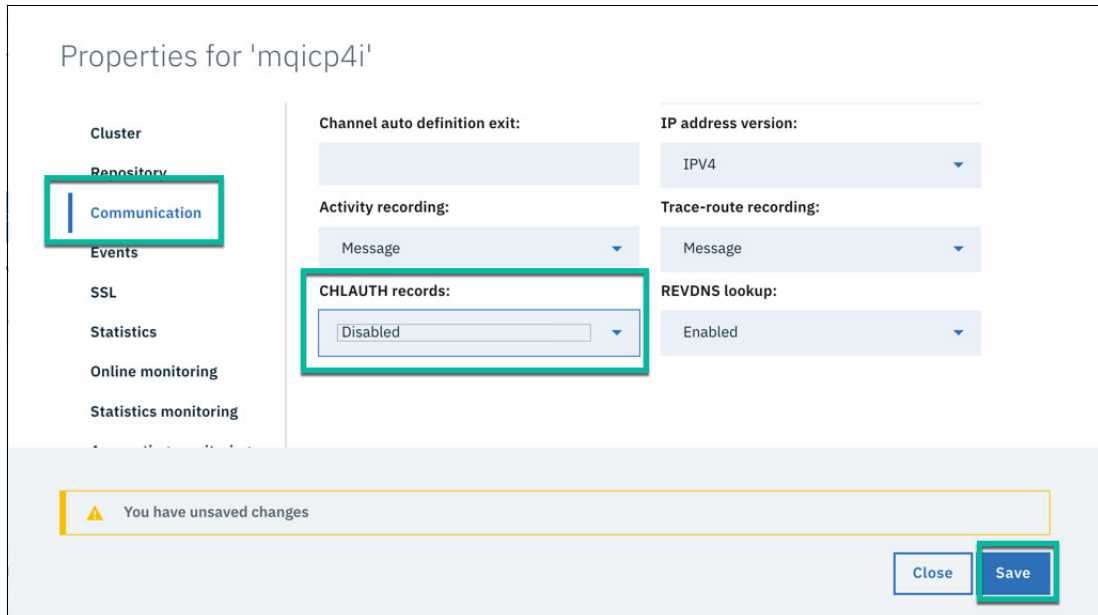


Figure 6-103 Queue manager configuration -19

19. This action removes the warning message and confirms that the changes were applied. Now you can click **Close** as shown in Figure 6-104.

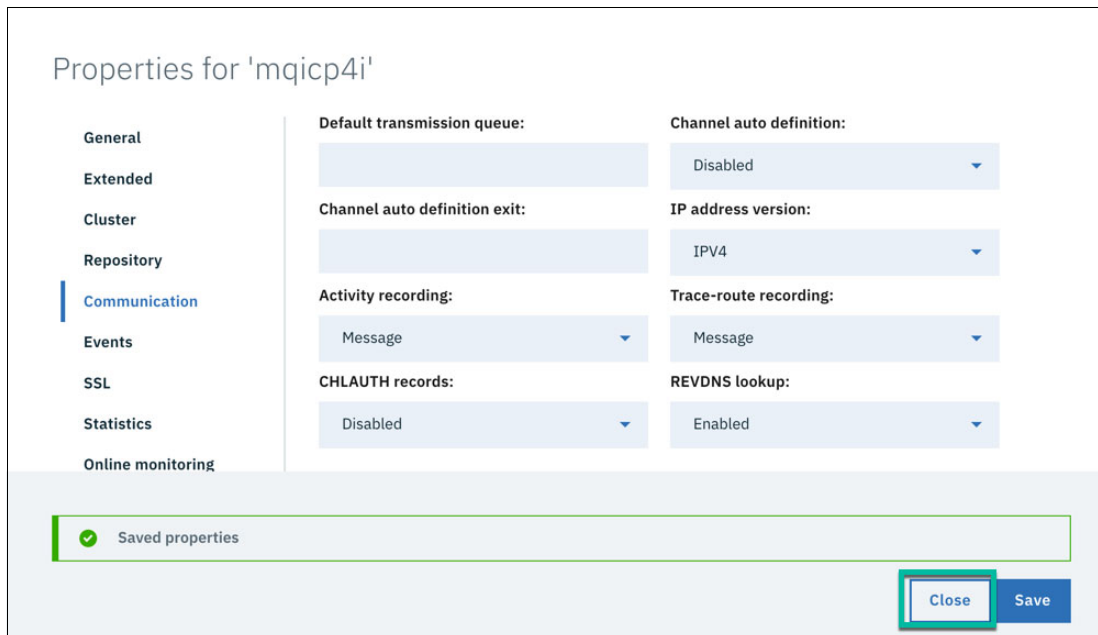


Figure 6-104 Queue manager configuration -20

20. To avoid potential issues using an administrator user we also disable **Client Connection checking**. Don't forget that we do this for simplicity reasons, but this is not recommended in a production environment.

First, we add the Authentication Information widget and we configure to Show System objects as we explained before. This adds the tile to the web UI that is shown in Figure 6-105 on page 239.

Authentication Information on mqicp4i ↻ ⚙ ✕



Delete  Properties  ... 1 item selected Deselect	
▲ Name	Type
SYSTEM.DEFAULT.AUTHINFO.CRLLDAP	CRL LDAP
SYSTEM.DEFAULT.AUTHINFO.IDPWLD/	IDPW LDAP
SYSTEM.DEFAULT.AUTHINFO.IDPWOS	IDPW OS
SYSTEM.DEFAULT.AUTHINFO.OCSP	OCSP
Total: 4 Last updated: 12:52:31 PM	

Figure 6-105 Queue manager configuration -21

- After the system objects are displayed in the widget, select **SYSTEM.DEFAULT.AUTHINFO.IDPWOS** and hover over the Properties menu and click on it as shown in Figure 6-106 on page 240.

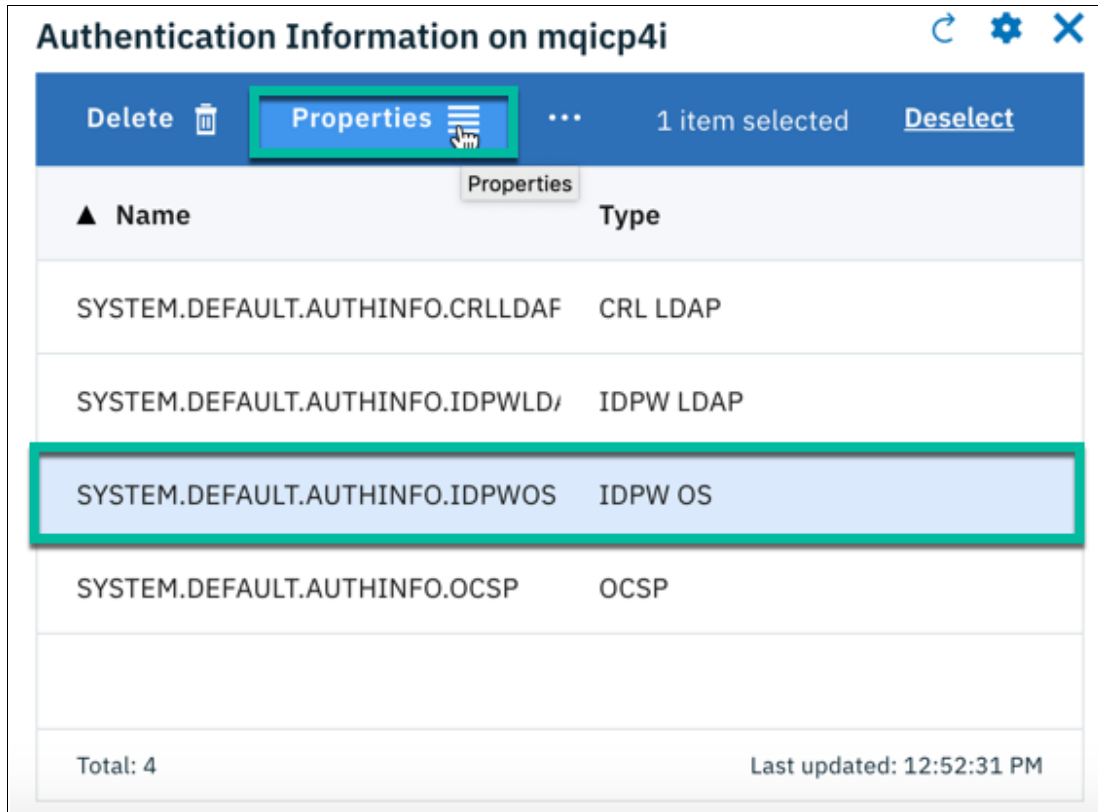


Figure 6-106 Queue manager configuration -22

22. In the Properties window, navigate to the User ID + password section, and modify the value of the Client connections field to None. Then, click **Save** as shown in Figure 6-107.

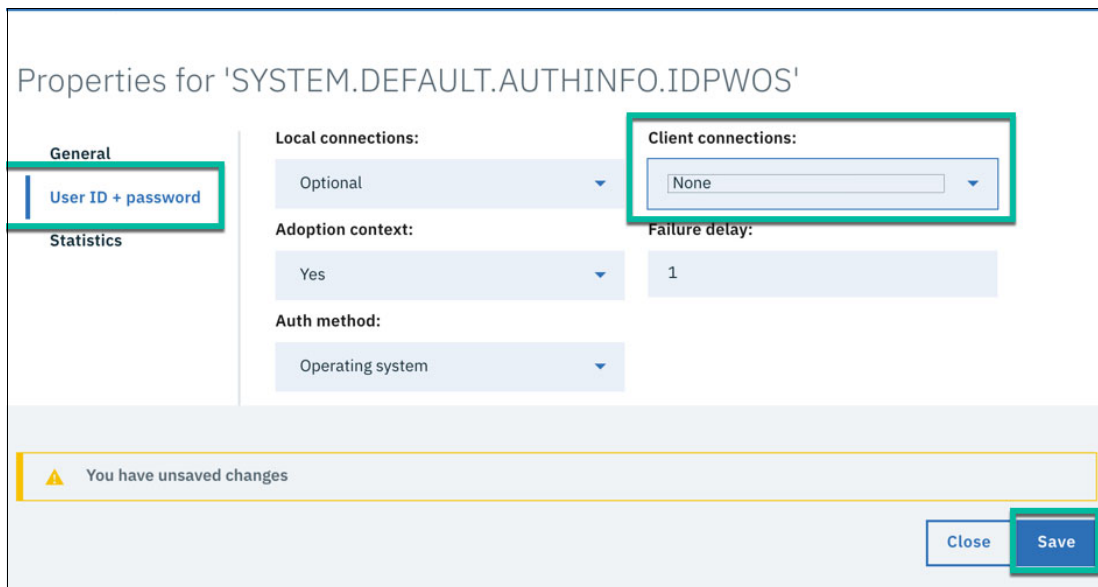


Figure 6-107 Queue manager configuration -23

23. This removes the warning message about unsaved changes and confirms that the properties have been saved. Now you can click **Close** as shown in Figure 6-108 on page 241.

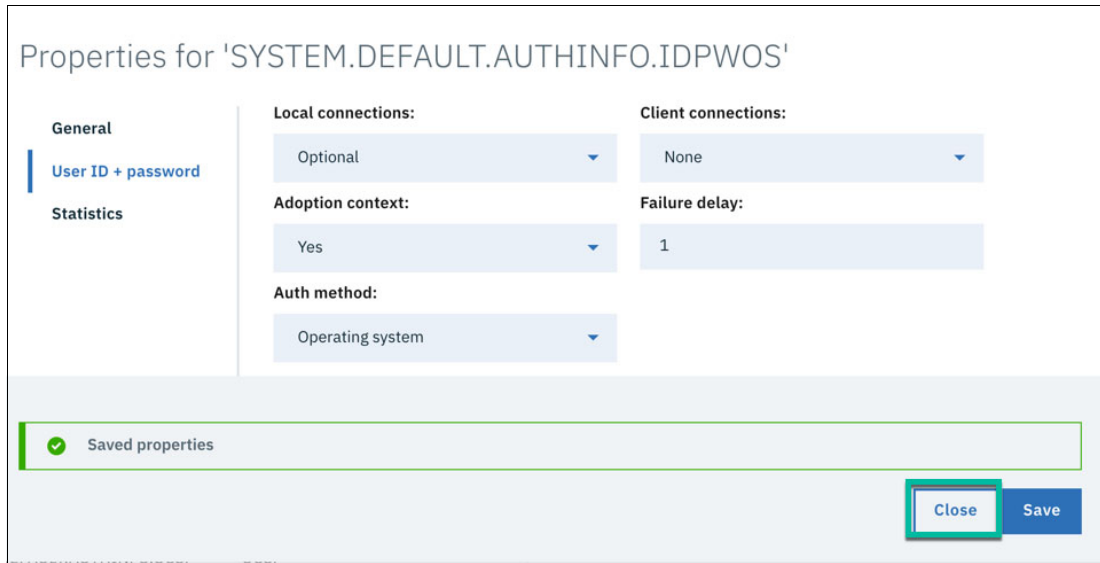


Figure 6-108 Queue manager configuration -24

24. This takes you to the main administration page and now you can proceed to the final step before you continue with the integration flow development. We need to refresh security to ensure that the changes we made take effect. Scroll as needed to make the Local Queue Manager widget visible. Hover over the ellipsis (...) in the upper menu bar to display the other menu and select the Refresh security option as shown in Figure 6-109.

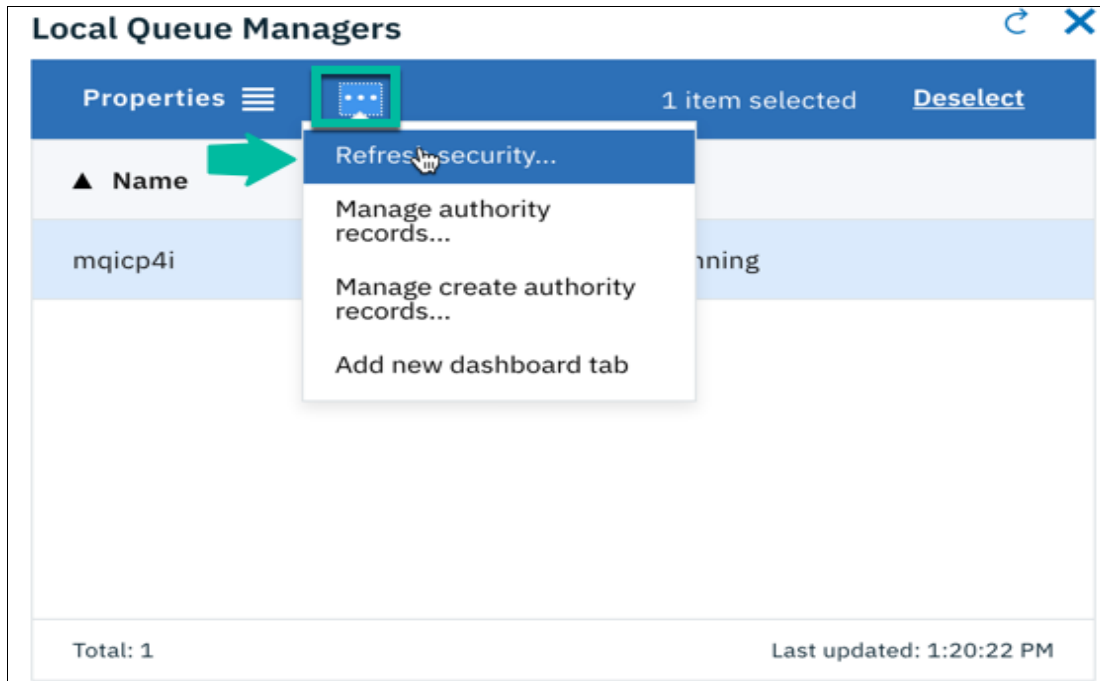


Figure 6-109 Queue manager configuration -25

25. In the new Refresh security dialog click the **Connection authentication link** as shown in Figure 6-110.

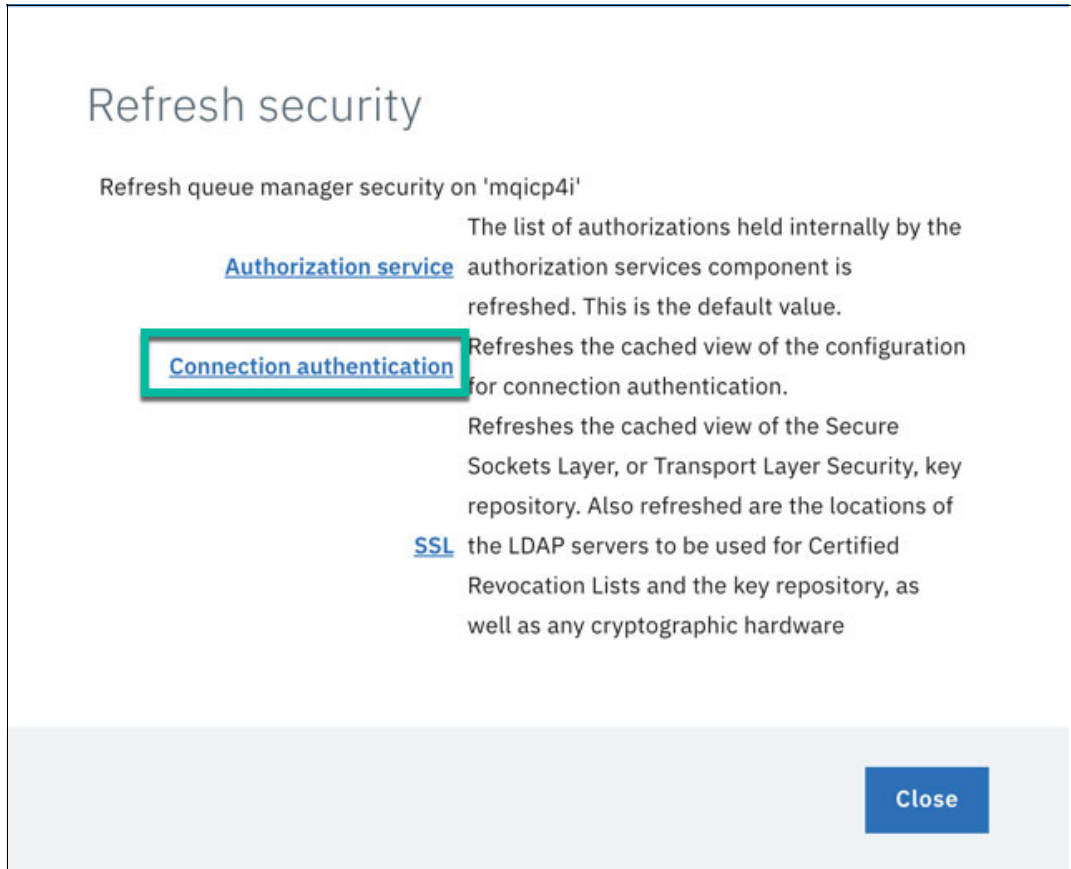


Figure 6-110 Queue manager configuration -26

26.A message in the upper part of the Web UI indicates that queue manager security was refreshed successfully. Click the X to the right side of the message to dismiss it, as shown in Figure 6-111.

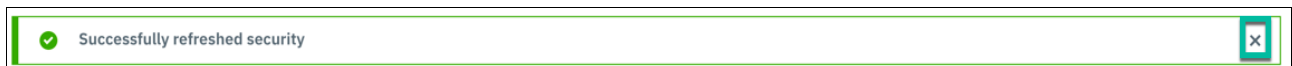


Figure 6-111 Queue manager configuration -27

We are now ready to start the implementation of the integration flow.

6.4.4 DB commands implementation

In the initial design we will leverage the fact that IBM App Connect doesn't require a local queue manager any more. And we will connect to a central queue manager that acts as the messaging backbone for the whole environment that has been configured with the corresponding persistence volumes to handle high availability. Later on, we will explore the need to support two-phase commit (2PC) and what changes are required to address this requirement. For now, this is the logical representation of the solution.

Figure 6-112 shows the IBM Cloud Pak for Integration Cluster.

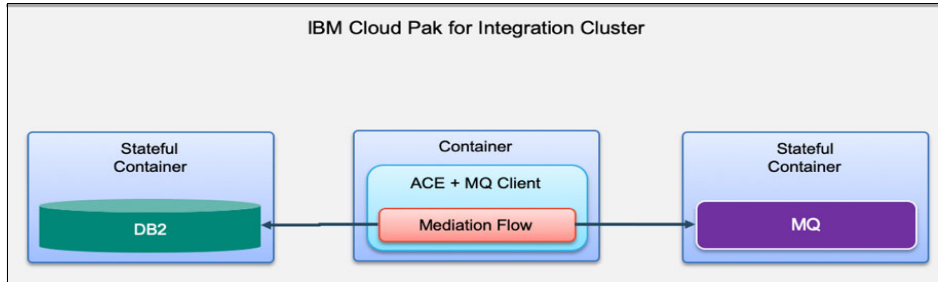


Figure 6-112 IBM Cloud Pak for Integration Cluster

The advantage of using IBM App Connect and IBM MQ to build what is effectively a microservice component is that you can leverage the existing skills in your integration community. As you will see, the design of the integration flow in IBM App Connect uses the same core concepts that you have used in the past to interact with IBM MQ and a database.

You can use the steps outlined in 6.2.2, “Db2 table setup” on page 178 to create the resources needed in the Toolkit to interact with a database. As a reference, in the sample implementation we are presenting here, you need a database connection, a database project, and the corresponding database definition as shown in Figure 6-113 and Figure 6-114 on page 244.

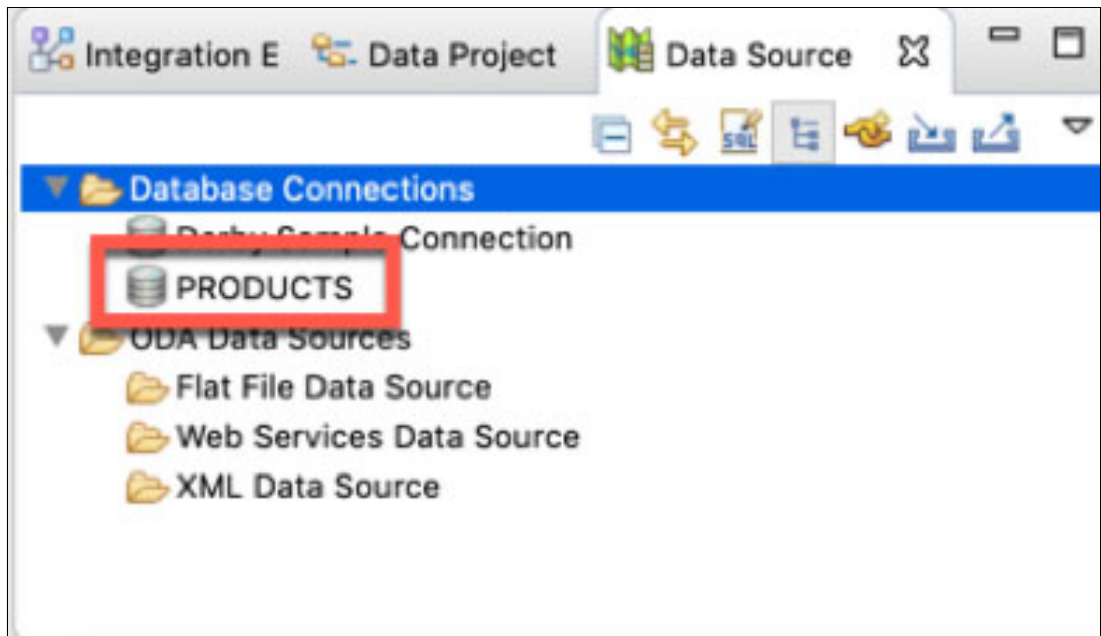


Figure 6-113 Database connection -1

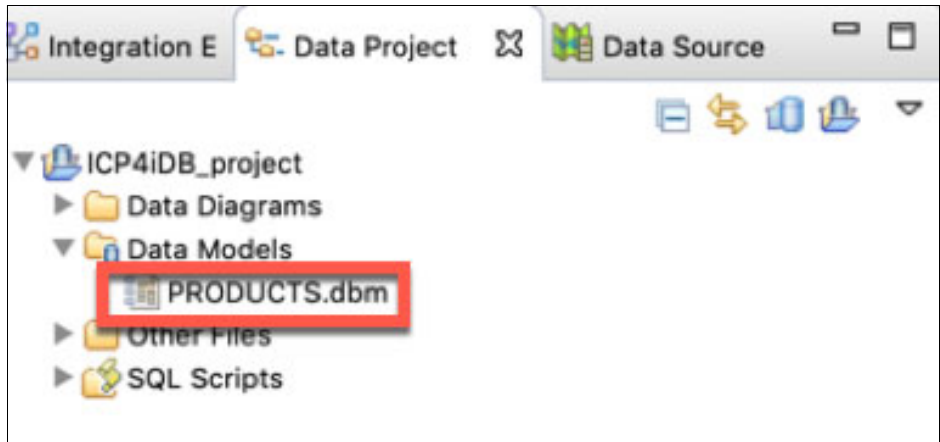


Figure 6-114 Figure 6-115 Database connection -2

Figure 6-115 shows the structure of the application.

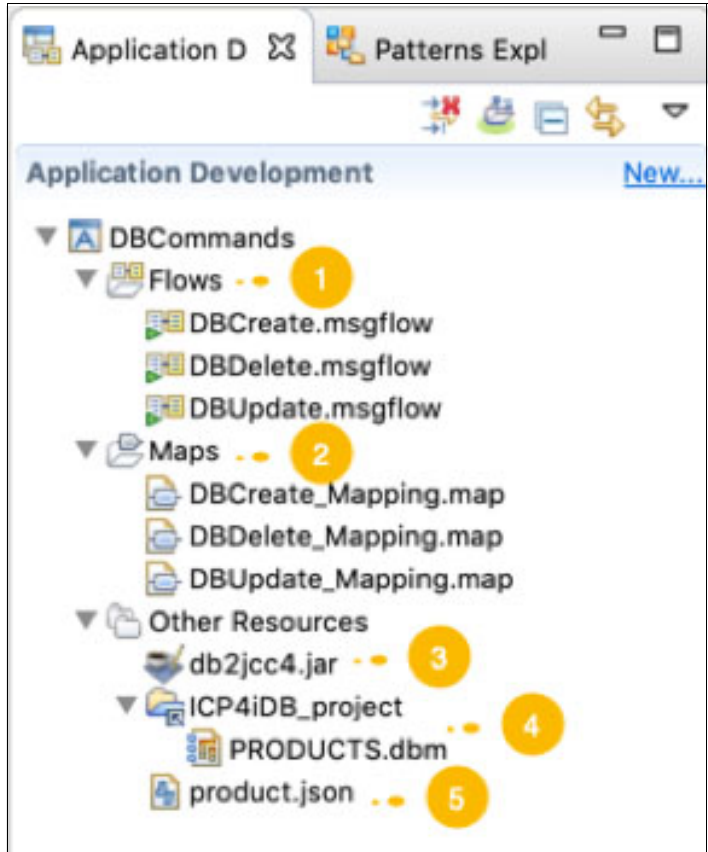


Figure 6-115 Structure of the application

We have three integration flows, one for each command. The interaction with the database will leverage the database capabilities in the graphical map node. Therefore we have a map for each one of the commands as well.

We will also take advantage of the new capabilities in IBM App Connect to include the jdbc driver as part of the BAR file. So we have included it in the application to minimize external dependencies.

We need to include the reference to the database project so we can use the database definitions in the graphical maps.

In this case we are going to use JSON as the data format to receive the data that will be processed by the commands. So we need to create and include the corresponding JSON schema to simplify the mapping in the map nodes as well.

Example 6-3 shows the JSON schema that is used in the product.json file, which maps the database data model used in this scenario.

Example 6-3 JSON schema used in the product.json file

```
{
  "$id": "https://example.com/person.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Product",
  "type": "object",
  "properties": {
    "lastUpdate": {
      "type": "string"
    },
    "partNumber": {
      "type": "integer"
    },
    "productName": {
      "type": "string"
    },
    "quantity": {
      "type": "integer"
    },
    "description": {
      "type": "string"
    }
  }
}
```

Note: This is the bare-minimum information that is required to create the JSON schema. In a real-life scenario you might need to extend it.

The three flows follow the same basic model as the one depicted in Figure 6-116 on page 245. The differences will be in the queue name that is used in the MQ Input Node called *DB Command*, and of course in the logic inside the Graphical Map Node called *Process Action*.

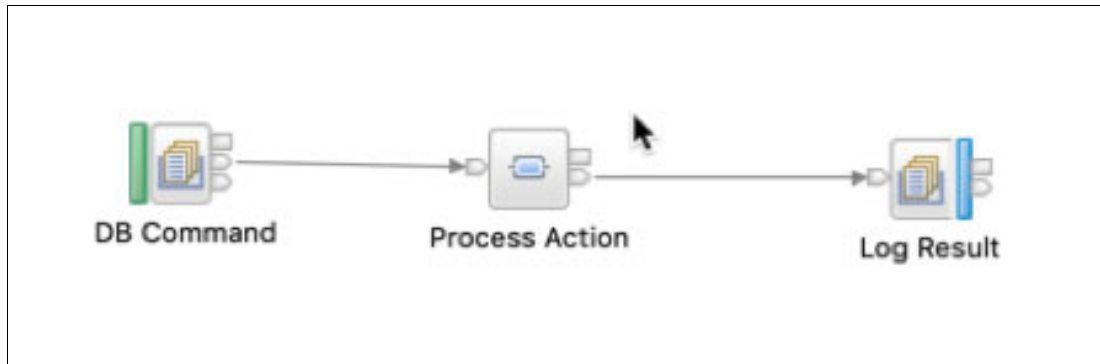


Figure 6-116 The flows

Note: The purpose of the scenario is to show the core principles to implement the commands. No error handling is included beyond the backout configuration of the queues.

To enable the connectivity to the queue manager and the database we will use the new policies introduced in IBM App Connect that will be included with the BAR file as well. In this way, we minimize external dependencies and fit better in the containerized world of agile integration. We will discuss the policies in the next section.

6.4.5 Graphical data maps implementation

IBM App Connect provides several ways to interact with databases, including ESQL, Java, and graphical data maps (GDM) among others. For this scenario, we decided to use GDMs to avoid writing any code. But in a real-life scenario you can rely on any of the other options, depending on your particular needs.

Create command

Figure 6-117 shows how the map for the Create command looks like.

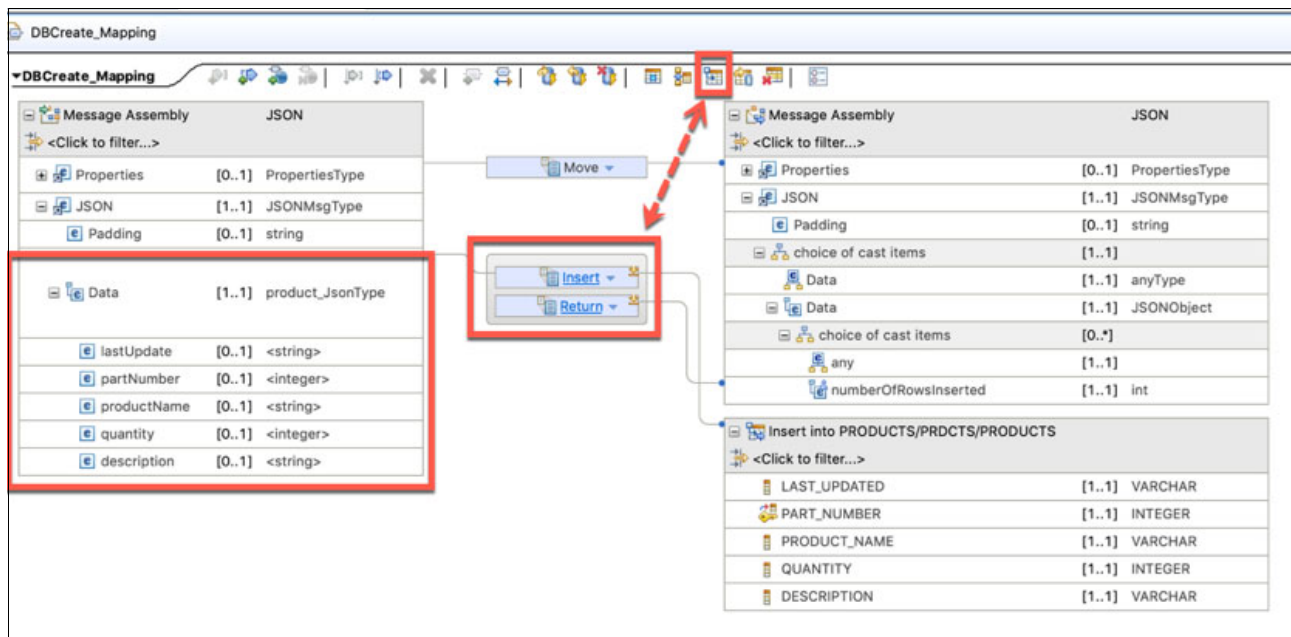


Figure 6-117 Map for the Create command

As shown in Figure 6-117, we are using the Insert a row into a database table function. It is important to mention that we were able to get the data structure for the input message since we included the JSON schema file as part of the project. Additionally we are mapping the return result from the insert to a single field that will be logged, but here you could add any other information you need.

And this is how the actual mapping looks like, as you can see it is a simple and straight forward mapping. The advantage of using similar names in your data models is that you can take advantage of the “auto map” feature available in GDM. This option is highlighted in Figure 6-118.

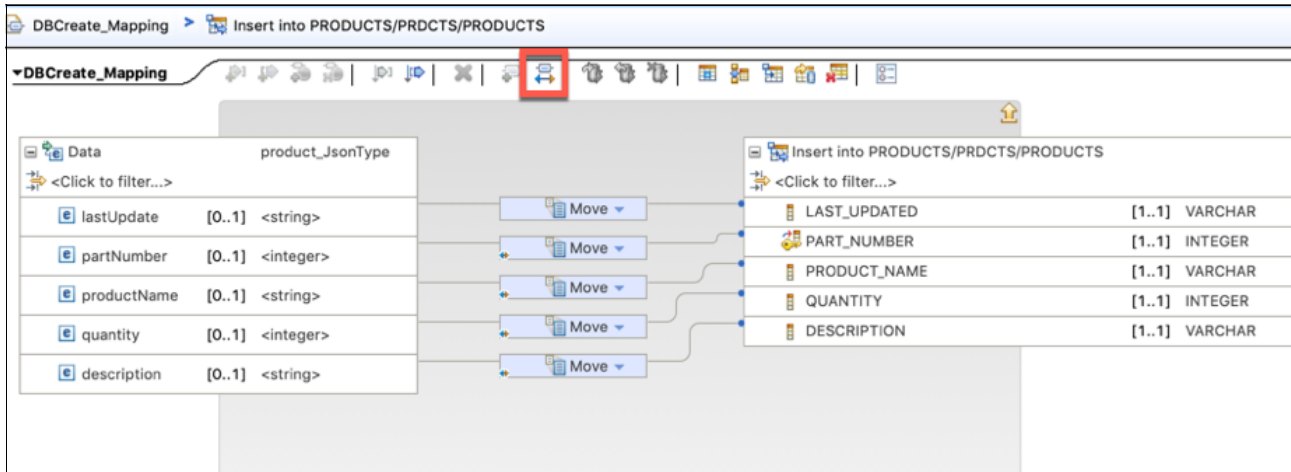


Figure 6-118 Auto-map feature

Update command

Figure 6-119 shows the map for the Update command.

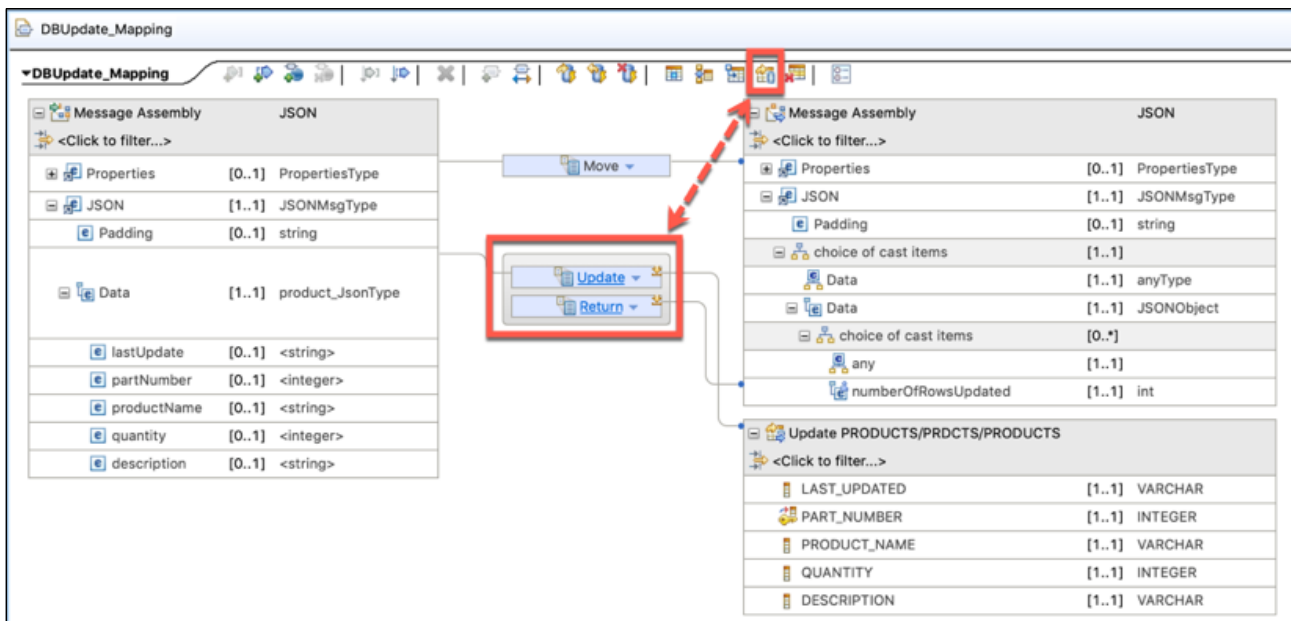


Figure 6-119 Map for the Update command

In this case, we are using the Update a row into a database table function. Instead of using straight moves, we are evaluating if each field is present to proceed to do the actual move, this way we avoid undesired consequences. The map looks like in Figure 6-120 on page 248.

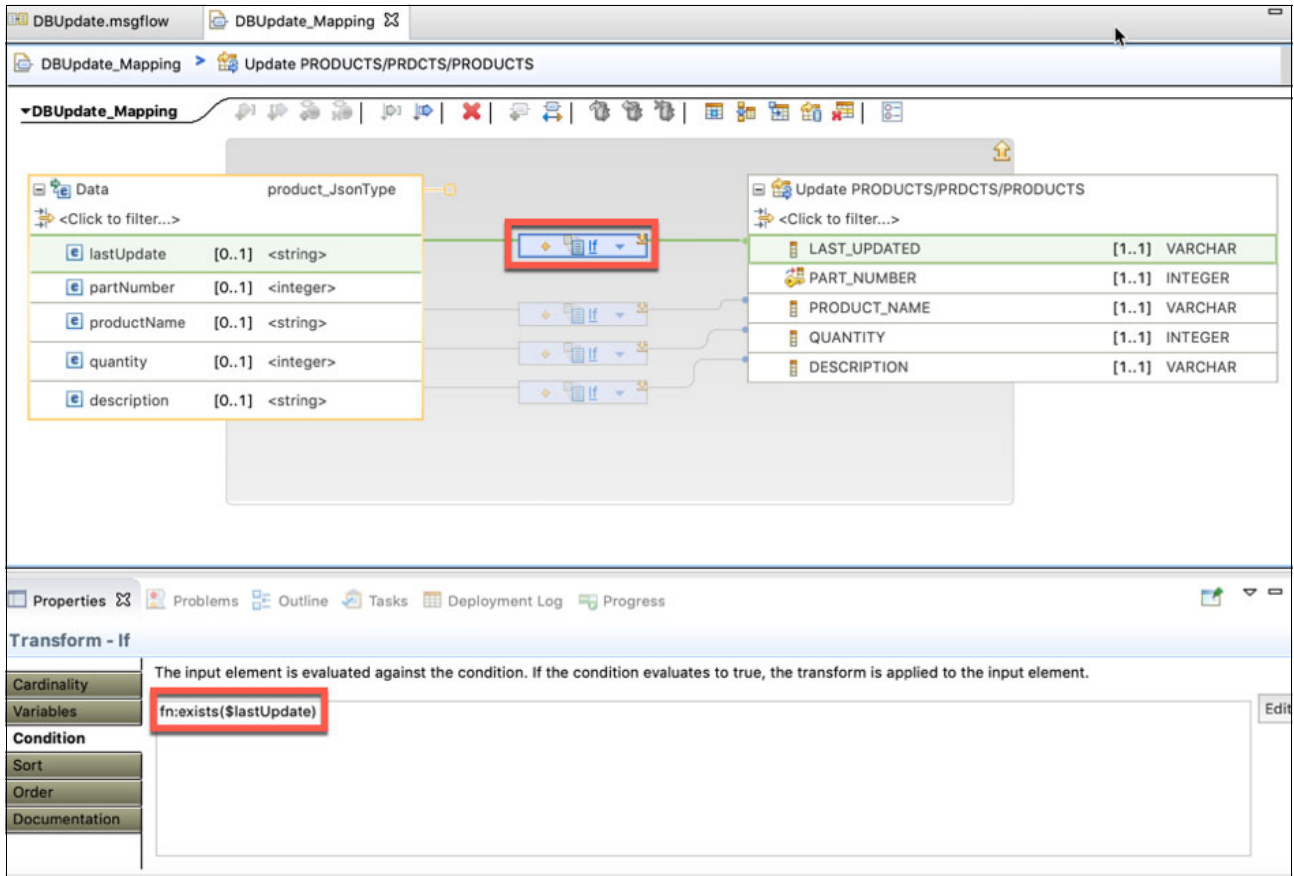


Figure 6-120 Update a row into a database table function

The other important difference versus the Insert map is that for the update we have included a “where” clause. This clause is based on the partNumber field that is provided in the input message, which corresponds to the primary key in the table. The definition looks like Figure 6-121 on page 249.

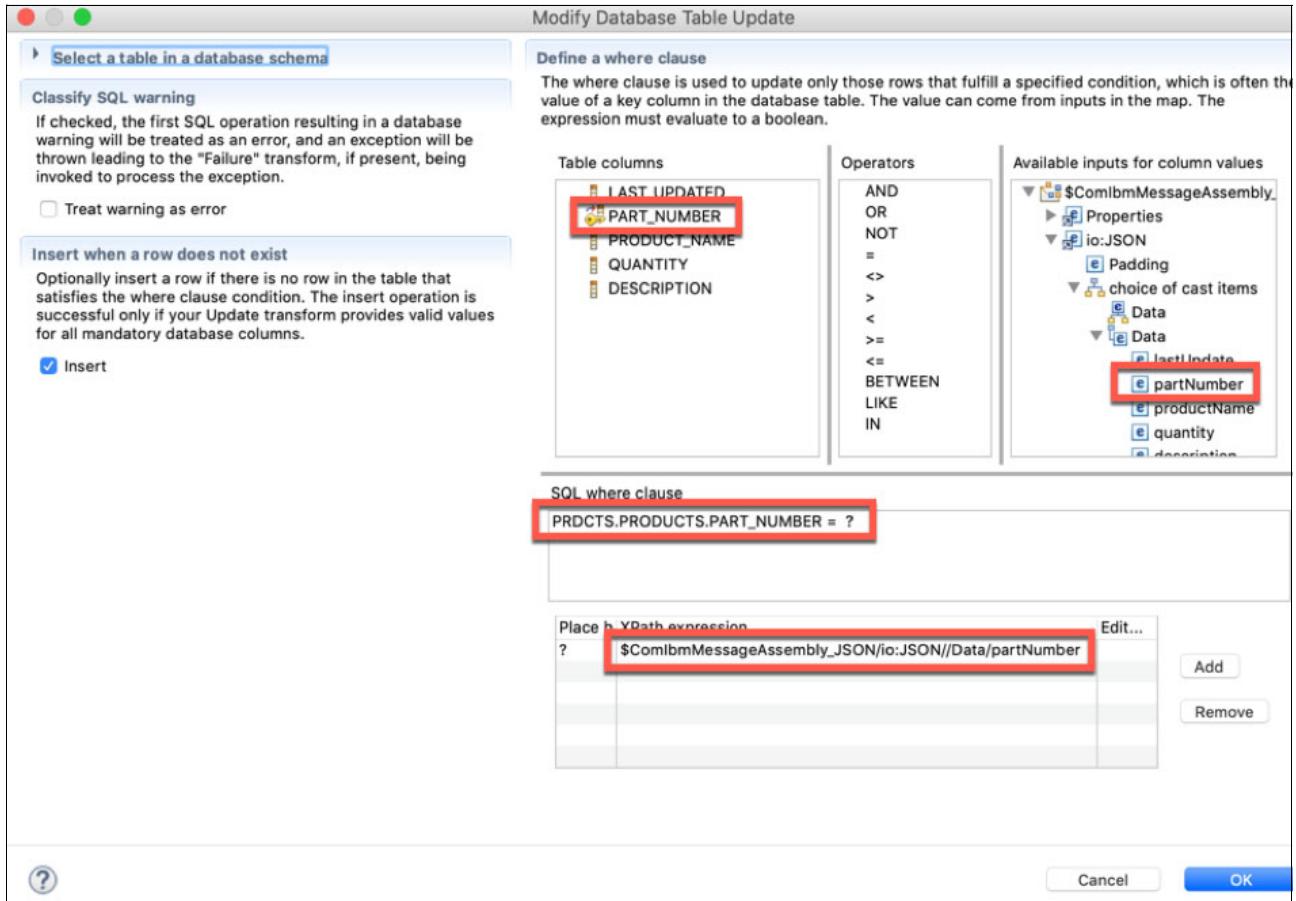


Figure 6-121 Modify Database Table Update

Delete command

Figure 6-122 shows the map for the Delete command.

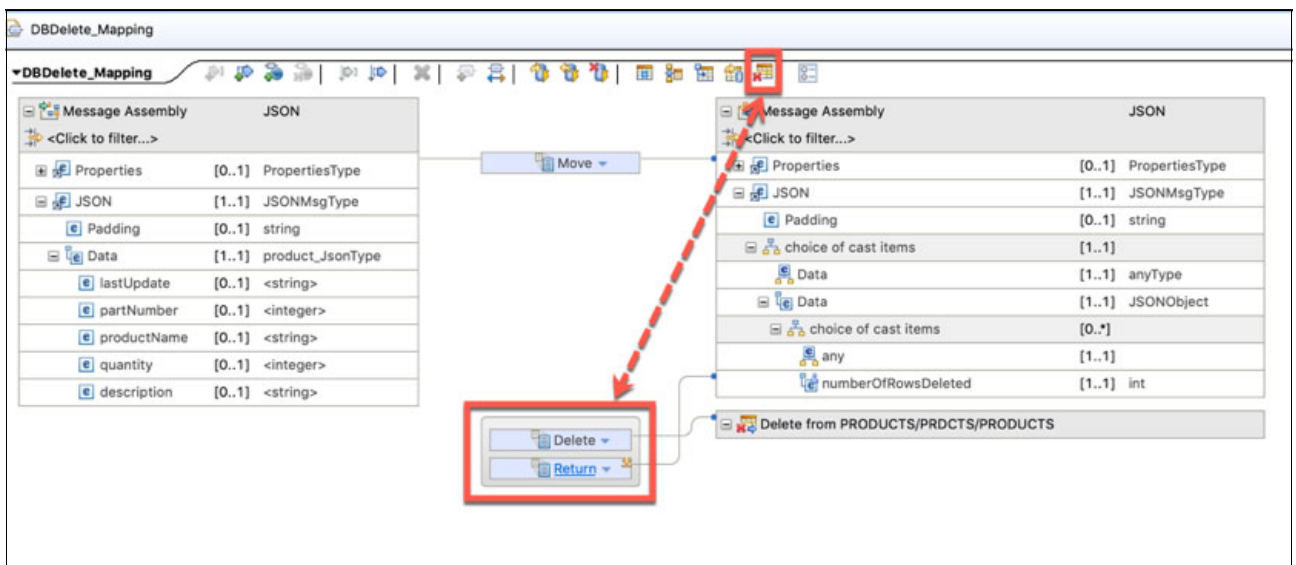


Figure 6-122 Map for the Delete command

In this case, no data mapping is needed. We just need to set the “where” clause properly to the corresponding input field, so that we delete the right record. Figure 6-123 shows how the configuration would look.

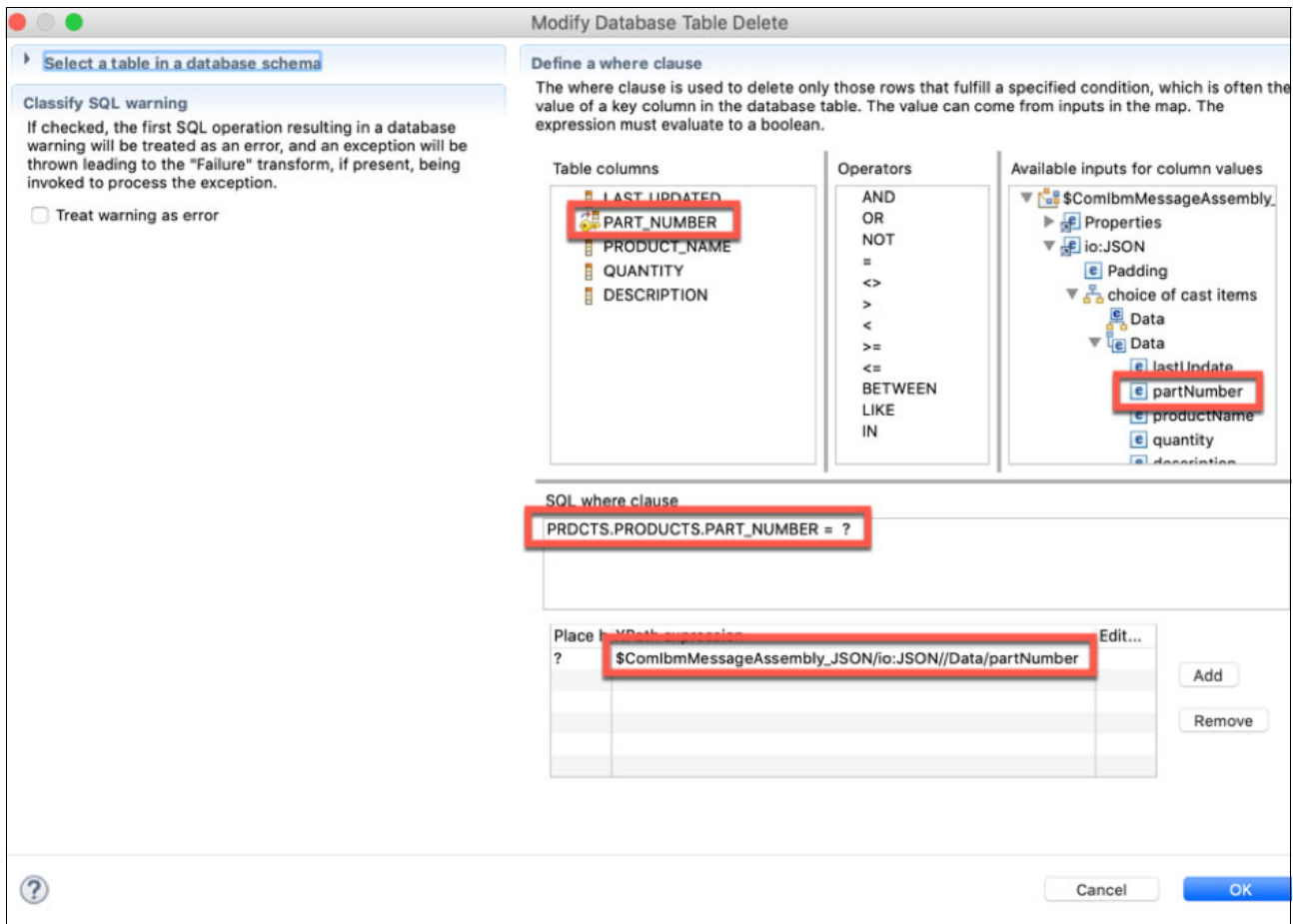


Figure 6-123 Modify Database Table Delete

6.4.6 Policy definitions

After the Integration Flow is ready, the other important elements are the policies associated with the two external resources that are required. Figure 6-124 on page 251 shows the definitions for both policies in the corresponding Policy Project.

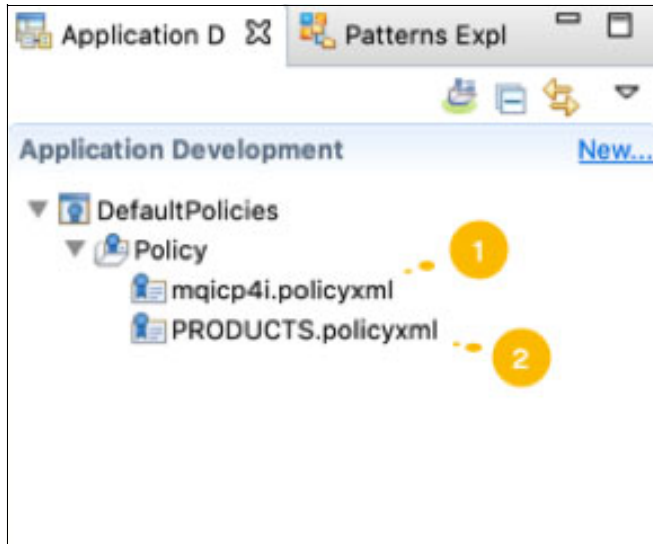


Figure 6-124 Policy definitions

In Figure 6-124 1 corresponds to the IBM MQ Endpoint policy and 2 corresponds to the JDBC Provider policy.

The properties for the IBM MQ Endpoint are the shown in Figure 6-125.

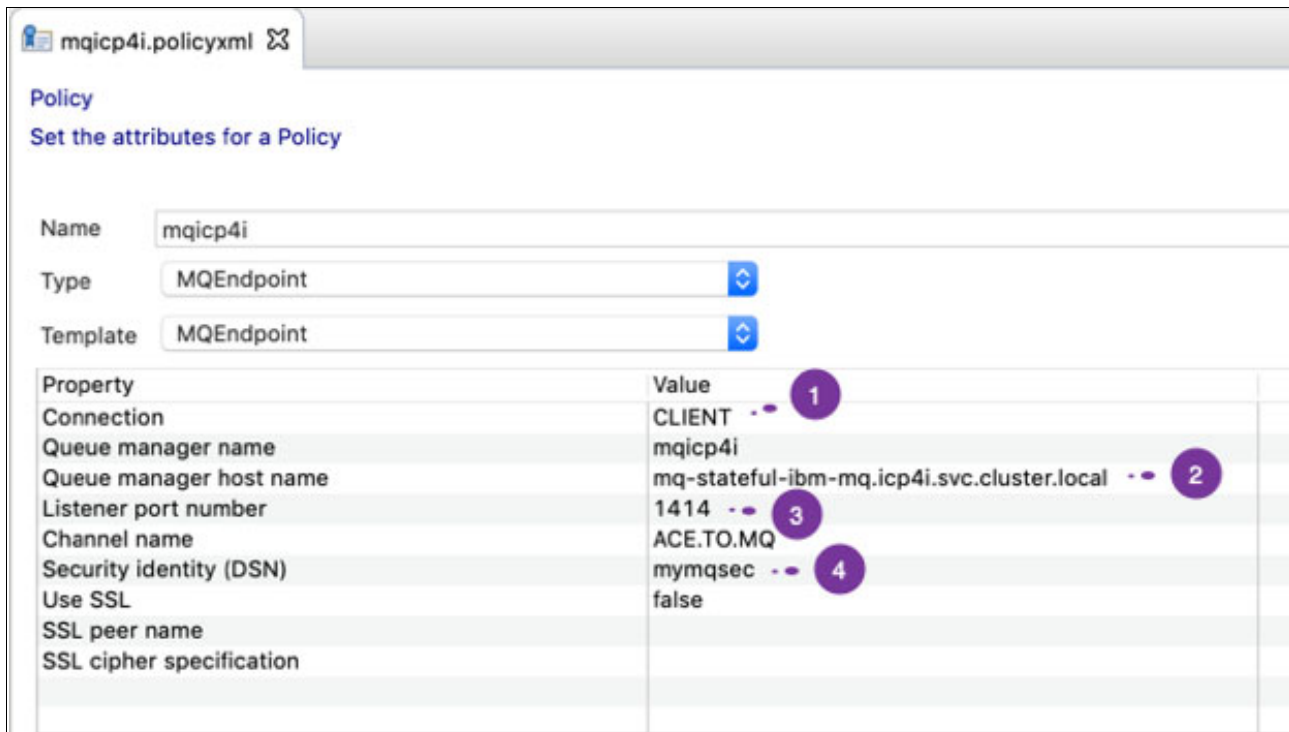


Figure 6-125 MQ Endpoint policy properties

Some relevant points about the policy are:

1. This corresponds to a client connection since we are connecting to a remote queue manager as mentioned above.

2. The queue manager hostname corresponds to the internal DNS value of the Cloud Pak for Integration Cluster since the Integration Server is running in the same cluster. If for some reason the queue manager would be running outside the cluster, you use the corresponding value here.
3. The same applies to the listener port. Inside the cluster the queue manager is listening in the default 1414 port. But if you wanted to access the same queue manager from a different location you would need to use the corresponding Node Port value.
4. The security identity corresponds to the secret that was defined in 6.5.3, “Securing the API” on page 265.

The properties for the JDBC Provider are the following shown in Figure 6-126.

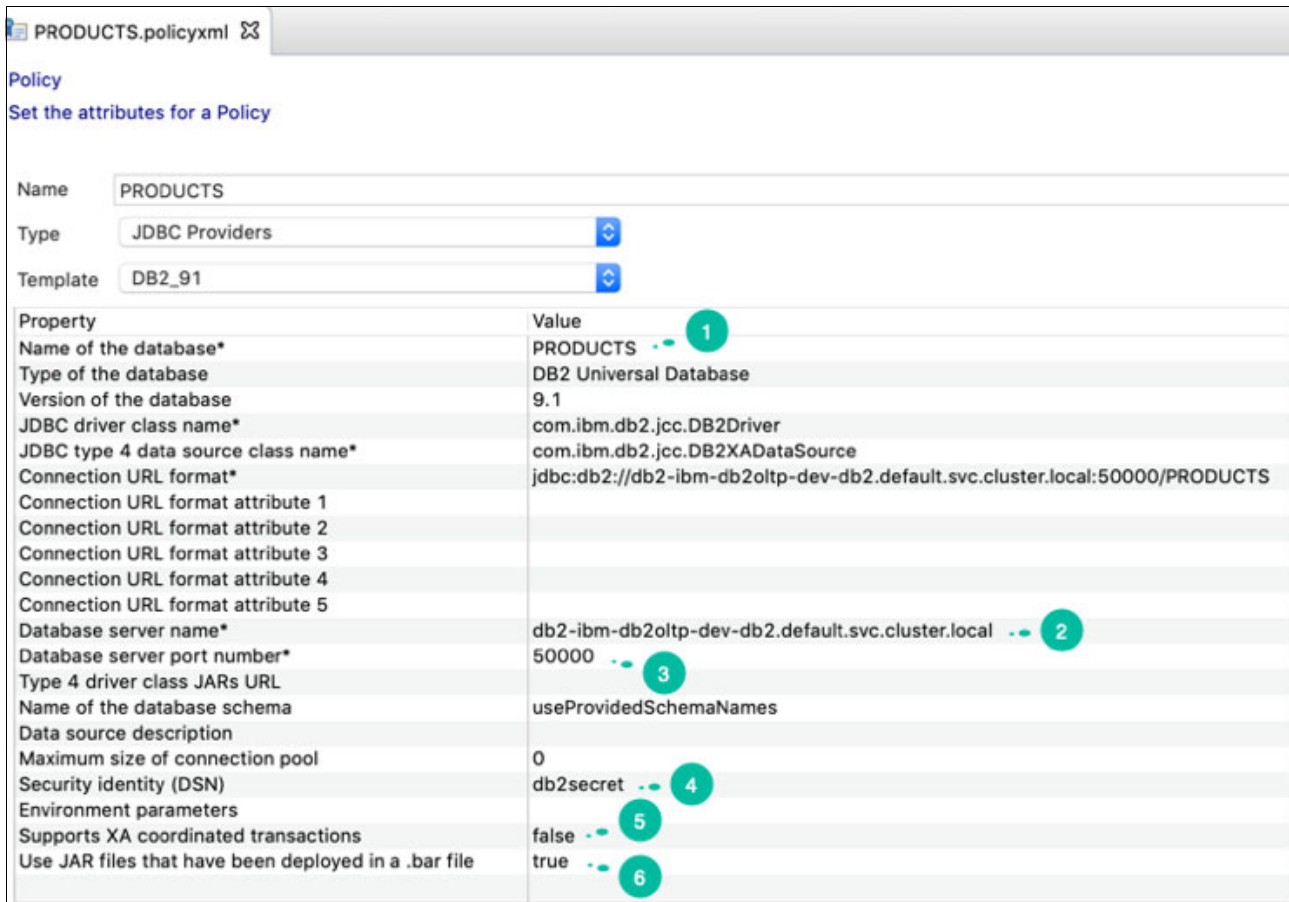


Figure 6-126 JDBC Provider policy properties

In this case, the highlights are as follows:

- ▶ The name of the policy must match the name of the database, as required by the GDM.
- ▶ The server name used is the one used inside the cluster, similar to what we did with the queue manager. If you wanted to access the server from outside the cluster, then the cluster IP address or equivalent DNS must be used.
- ▶ The port value is the same, internally Db2 is listening in the default port, but if your integration server would be running outside the cluster you must use the Node Port value instead.
- ▶ For the initial scenario we have not enabled global transaction coordination. In the next section we will discuss when you might want to enable this feature.

- ▶ As mentioned before, we have included the jdbc JAR file as part of the project as it is good cloud native practice to avoid external dependencies. To use the driver, we need to enable this property to make sure that the Integration Server uses the embedded driver.

6.4.7 BAR file creation

After you have developed the integration application and configured the associated policies, you can proceed to prepare the corresponding BAR file.

1. Start selecting the application as in Figure 6-127.

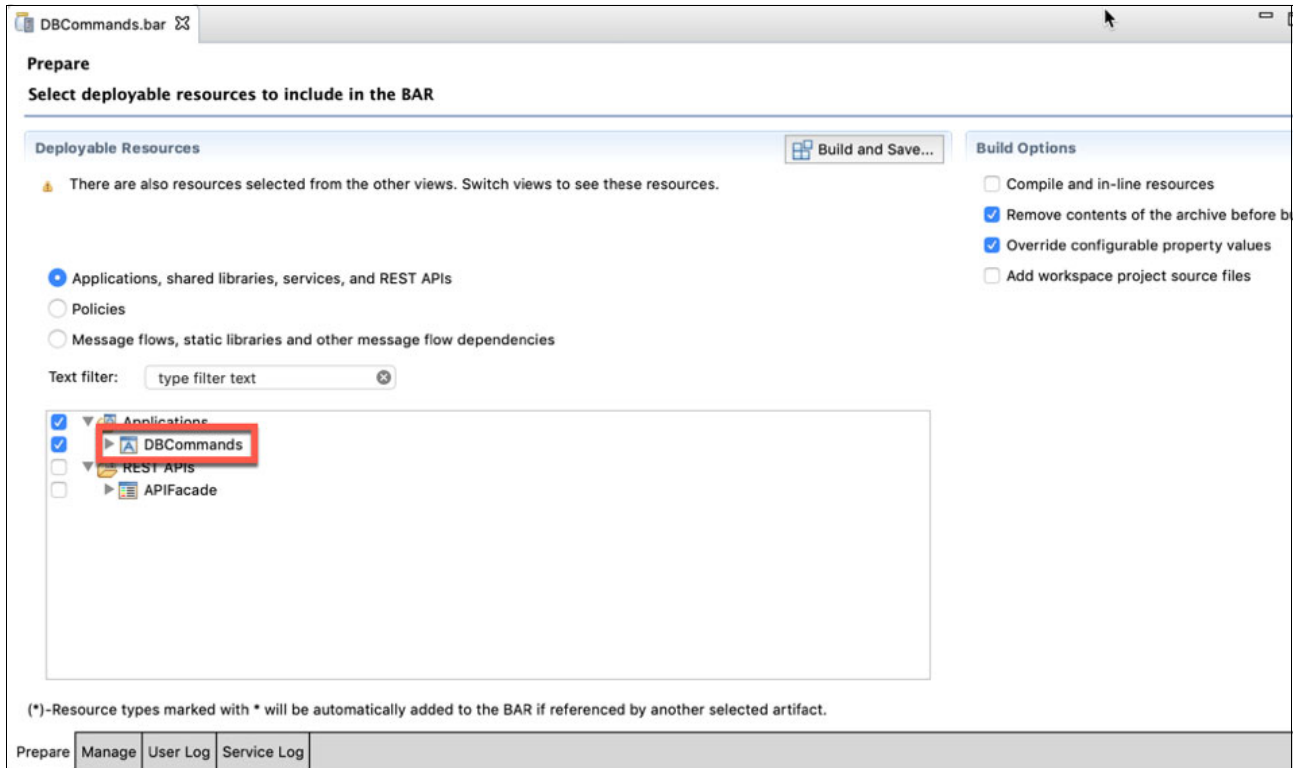


Figure 6-127 BAR file creation -1

2. Include the Policies project. Remember, this is one of the changes introduced with IBM App Connect. Policies replaced Configurable Services to provide a stateless configuration model that also allows you to include the policy with your BAR file. We take advantage of this feature now, to minimize external dependencies and fit better in the agile integration paradigm. See Figure 6-128 on page 254.

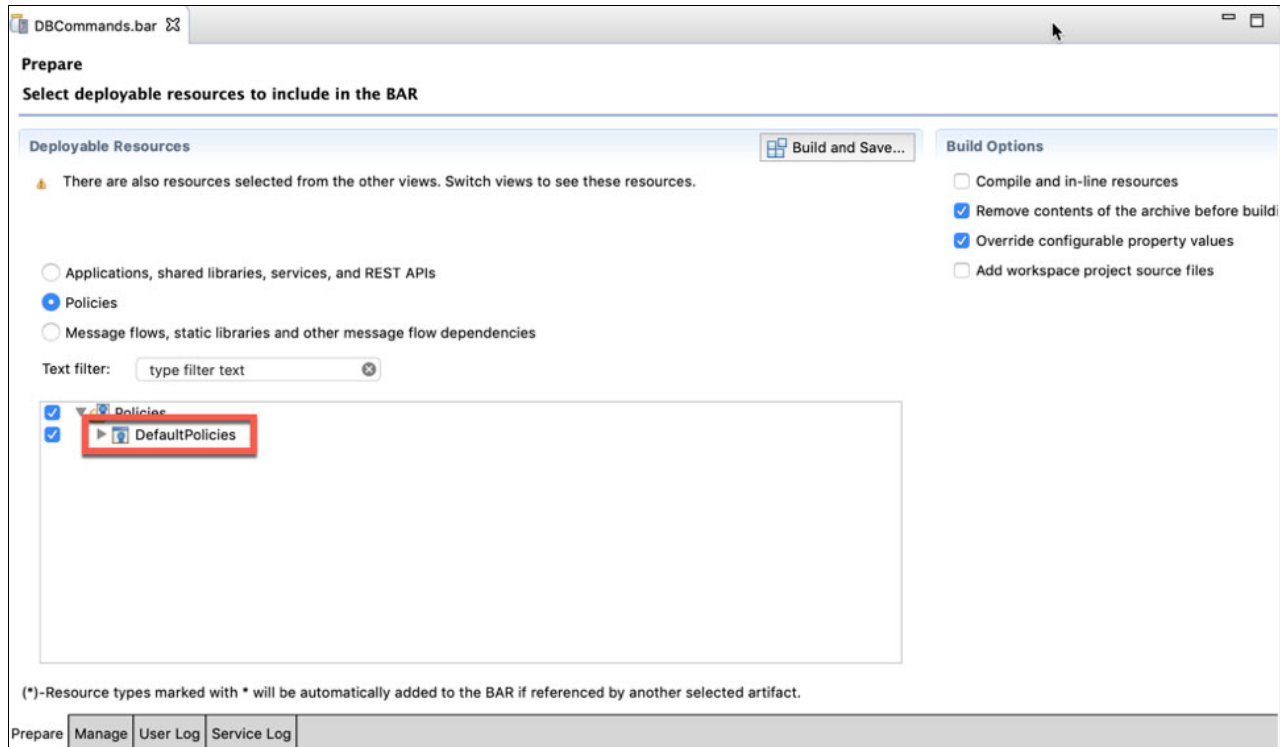


Figure 6-128 BAR file creation -2

3. After you have included both resources, the BAR file content will look like Figure 6-129 on page 255 and you can proceed to build the BAR file.

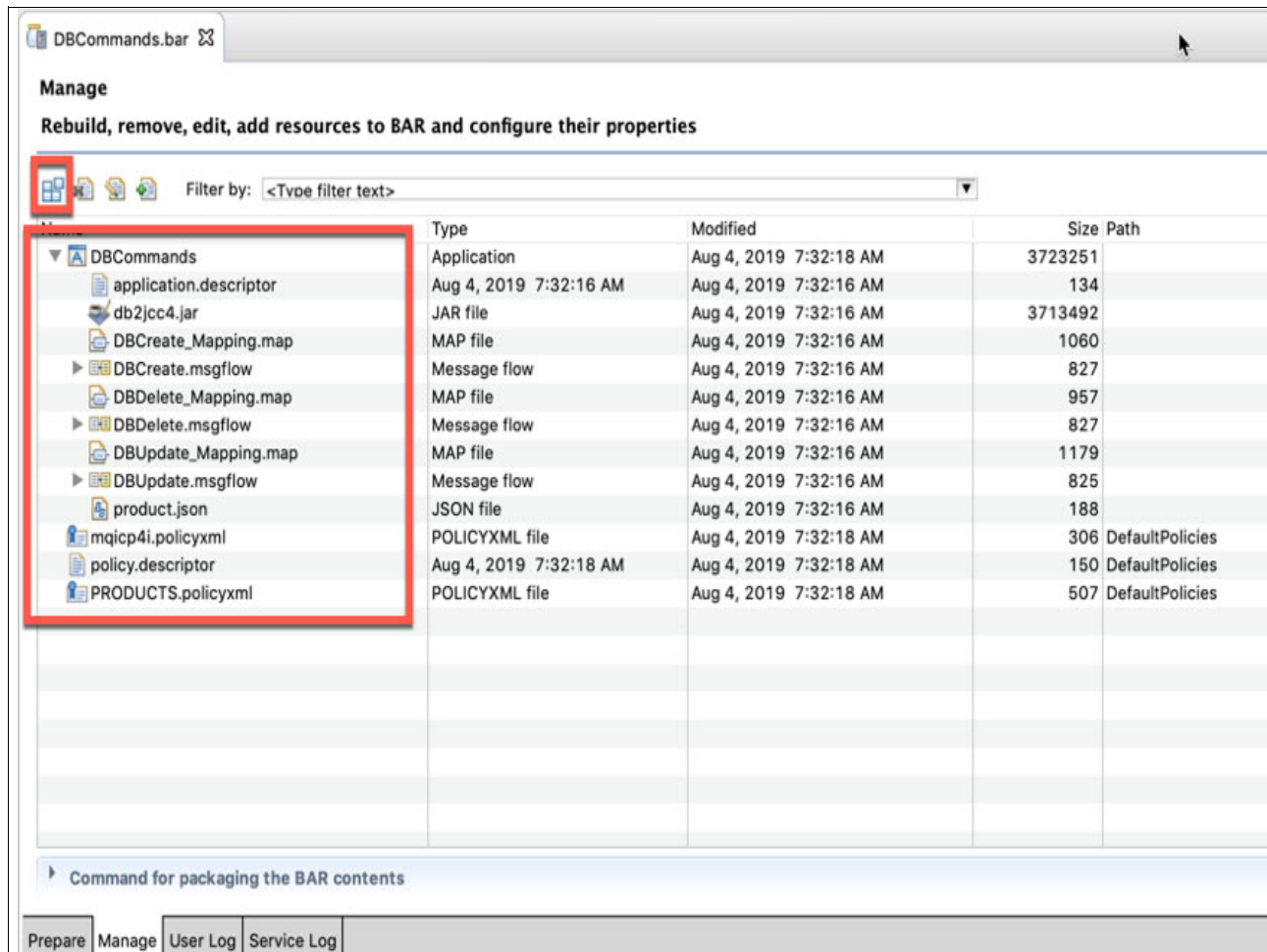


Figure 6-129 BAR file creation -3

- At the end, you have the BAR file that you can deploy into the IBM Cloud Pak for Integration using the Application Integration dashboard. You can check 6.2, “Application Integration to front a data store with a basic API” on page 173 for the details in the steps needed to complete the deployment.

6.4.8 Override policies for environment specific values

IBM App Connect gives you the option to embed the policies where you have configured your end points in a single BAR file. However, there will always be circumstances where you want to have the flexibility to override some values. In our example, for instance, we need to override the queue manager hostname and the database server name. That way, we can use the same integration solution in multiple environments — like production and quality assurance — without having to create multiple BAR files. We want to be able to treat the bar file as the unchanging source code and just override the environment-specific values each time.

To handle this situation, IBM Cloud Pak for Integration gives you the option to provide a set of properties in the form of *secret keys*. You can use the keys at deployment time to override values in the policies. The only consideration is that you need to create the secret before you perform the deployment since you will use the name of the secret when you configure the deployment.

1. As part of the deployment process you are asked to provide the BAR file and then you get the following pop-up window (Figure 6-130 on page 256). There, you have the opportunity to download a configuration package that includes the instructions for creating the secret for the deployment.

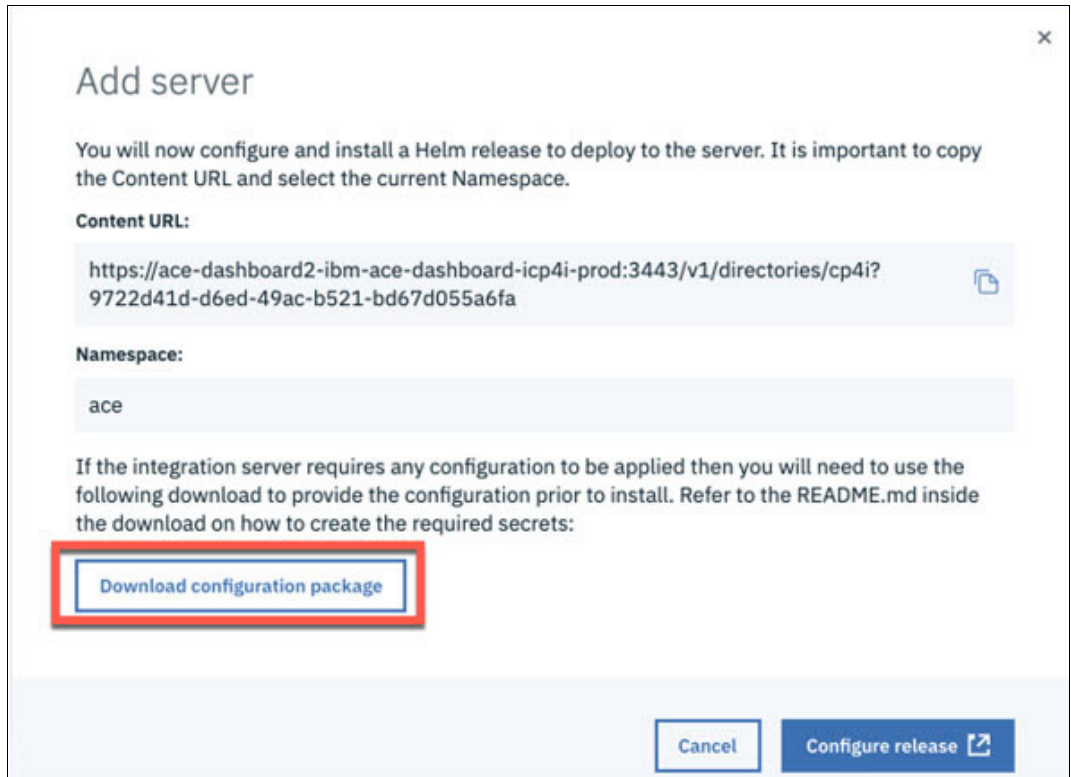


Figure 6-130 Download configuration package

2. The file you download is called `config.tar.gz`, which provides empty files for all the things that you can pass within a secret to Kubernetes for IBM App Connect to pick up on start-up. It also provides a script to generate the secret that we will use later. The content of the `config.tar.gz` is shown in Figure 6-131.

```

-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 adminPassword.txt
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 appPassword.txt
-rwxr-xr-x. 1 1000 1000 3819 Jul 2 04:42 generateSecrets.sh
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 keystore-mykey.crt
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 keystore-mykey.key
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 keystore-mykey.pass
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 keystorePassword.txt
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 mqsc.txt
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 odbc.ini
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 policyDescriptor.xml
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 policy.xml
-rw-r--r--. 1 1000 1000 4118 Jul 2 04:42 README.md
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 serverconf.yaml
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 setdbparms.txt
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 truststoreCert-mykey.crt
-rwxr-xr-x. 1 1000 1000 0 Jul 2 04:42 truststorePassword.txt

```

Figure 6-131 `config.tar.gz` file contents

As you can see, there are multiple files within config.tar.gz you can configure but for this particular scenario the relevant elements are policyDescriptor.xml and policy.xml.

3. We now need to copy the policy information across from our IBM App Connect Toolkit workspace into the files in the folder where we untared the config.tar.gz file. In this sample, we focus on the JDBC policy.
 - a. Copy the content of the JDBC policy file (PRODUCTS.policyxml) from your IBM App Connect Toolkit Workspace, and paste it into the policy.xml file.
 - b. Copy the content of the policy descriptor file (policy.descriptor) from your IBM App Connect Toolkit workspace, and paste it into the policyDescriptor.xml file.
4. You can now proceed to generate the secret using the following command, which is also included with the package:

```
./generateSecrets.sh <config-secret>
```

Note that in order to execute the command successfully you need to be logged in to your OCP cluster and using the right project, by default it should be *ace*.

5. Finally, you use the secret that you created when you were asked to configure the deployment. The field that must include the secret created is highlighted in Figure 6-132. For more details about the full deployment process for a BAR file, see 6.2, “Application Integration to front a data store with a basic API” on page 173.

The screenshot shows a configuration form for an Integration Server. The form has several input fields. The field 'The name of the secret to create or to use that contains the server configuration' is highlighted with a red border and contains the text '<config-secret>'. Other fields include 'Integration Server name', 'List of key aliases for the keystore', 'List of certificate aliases for the truststore', 'Name of the default application', and 'File system group ID', all with 'Enter value' placeholders.

Figure 6-132 secret file

6.4.9 Global transaction coordination considerations

The design used above achieves the desired goal of implementing the commands to Create/Update/Delete a data source using the *Command Query Responsibility Separation (CQRS) paradigm*. However, there is an aspect that needs to be considered if the implementation needs to assure consistency among the resource managers that are involved. In this case Db2 and MQ are involved.

As discussed previously, the scenario involves receiving a message with the command instructions and the data to modify the database. Then we are just sending a result message to another queue for logging purposes. Now imagine, there is a business requirement to guarantee that the logging messages are consistent with any change to the database in case there is a failure in any of the two resource managers.

In other words, we need to treat the whole flow as a single unit of work. If we cannot successfully put the log message in the queue, then we need to roll back the change that we made to the database in the previous step. When we want to make consistent changes across separate resources such as a database and a queue, this is known as *Global Transaction Coordination* or *Two-Phase Commit (2PC)*. The good news is that this is something that IBM App Connect has supported for many years even in its incarnations (such as IBM Integration Bus, WebSphere Message Broker). However, you need to take into account some considerations in the container world, whenever you need to address such a requirement.

IBM App Connect relies on IBM MQ to act as the global transaction coordinator, as explained in the IBM Knowledge Center article titled “*Message flow transactions*”:

https://www.ibm.com/support/knowledgecenter/en/SSTTDS_11.0.0/com.ibm.etools.mft.doc/ac00645_.htm

If you need to have 2PC, instead of using a remote queue manager, you must create your container using an image that includes both IBM App Connect and a local IBM MQ server.

Refer to the section “*When IBM App Connect needs a local queue manager*” for additional details. A high-level diagram for this scenario is shown in Figure 6-133.

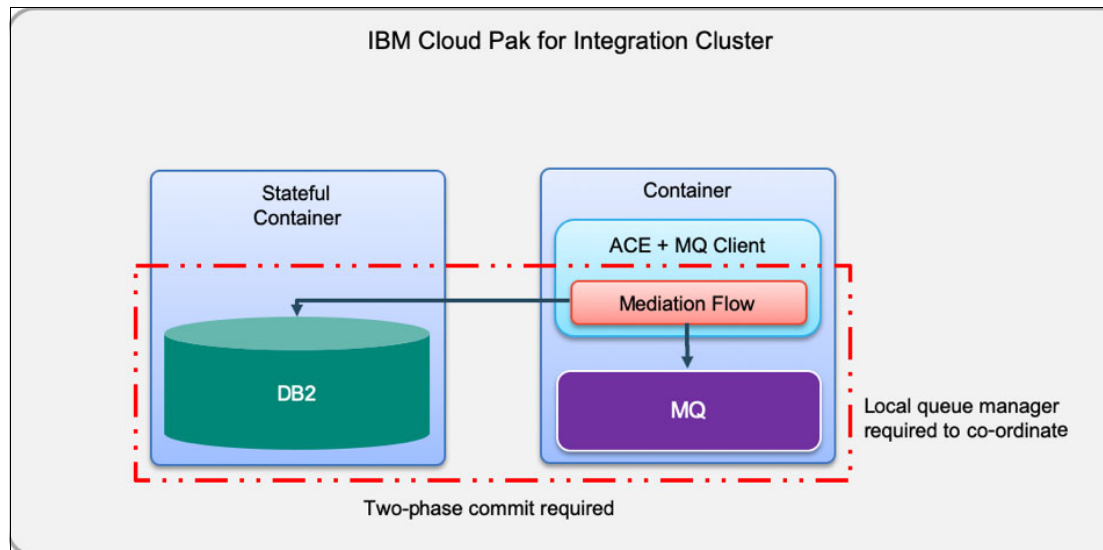


Figure 6-133 High-level diagram for the scenario

In this case, you would need to change the JDBC Provided policy to enable “Support for XA coordinated transactions” and make sure that the driver you are using is a JDBC Type 4. Additionally, two extra configuration changes are required to the associated local queue manager:

1. Modify the qm.ini file associated with the queue manager to include a stanza entry for the database with the following format (Example 6-4 on page 258):

Example 6-4 qm.ini file

```

XAResourceManager:
  Name=<Database_Name>
  SwitchFile=JDBCSwitch
  XAOpenString=<JDBC_DataSource>
  ThreadOfControl=THREAD
  
```

in our case it would look like this:


```
XAResourceManager:  
  Name=SAMPLES  
  SwitchFile=JDBCSwitch  
  XAOpenString=SAMPLES  
  ThreadOfControl=THREAD
```

2. Set up queue manager access to the switch file by creating a symbolic link to the switch files that are supplied in the IBM App Connect installation directory. The command would be something like the following:

```
ln -s /opt/ibm/ace-11/server/lib/libJDBCSwitch.so /var/mqm/exits64/JDBCSwitch
```

For additional details you can check the IBM App Connect Knowledge Center article titled “Configuring a JDBC type 4 connection for globally coordinated transactions”:

https://www.ibm.com/support/knowledgecenter/SSTTDS_11.0.0/com.ibm.etools.mft.doc/ah61330_.htm

With this in mind, it is important to mention that at the moment of writing this IBM Redbooks publication there is no way to inject those changes at deployment time of the IBM App Connect and IBM MQ container. Therefore, if you need to implement two-phase commit for your integration flows, you must use kubectl in order to access the pod you have deployed. Then, make the changes directly there. However, IBM’s labs are exploring the best option to handle this scenario in the future to have a more natural way to implement it. We will provide an update in the form of a technote after a different approach is defined.

6.4.10 Conclusion

In this section we have demonstrated how you can use IBM App Connect and IBM MQ as part of the IBM Cloud Pak for Integration to implement the “command” side of the Command Query Separation pattern to interact with IBM Db2 using one-phase commit and leverage your existing skills on IBM App Connect and IBM MQ, but we also highlighted the new functionality that is introduced in IBM App Connect to fit better in the agile integration paradigm. And finally we discussed the considerations for implementation of a two-phase commit.

6.5 Consolidate the new IBM MQ based command pattern into the API

In 6.4, “Messaging for reliable asynchronous data update commands” on page 209 we discussed how to use IBM App Connect and IBM MQ to implement the Command side of the Command Query Responsibility Segregating (CQRS) pattern. Using a fire-and-forget pattern over a messaging transport such as IBM MQ was a good option to address the performance issues related to slow response times, and provide availability that is not tied to that of the database.

However, there is an issue with this approach. It requires that the consumer of this service must have the ability to talk to IBM MQ. This requires IBM MQ specific client-side libraries, and indeed the knowledge to use them. We could reduce the knowledge burden on the developer by using the standards-based JMS library to talk to IBM MQ. That way, they would not need to know IBM MQ. But JMS itself is still a reasonably complex interface to learn if you have not used it before.

A good alternative is to make the act of putting the message on an IBM MQ queue available via RESTful API. This provides perhaps the lowest barrier to implementation for most developers, regardless of the programming language they are using.

Notice that we don't have to choose API or IBM MQ, we can have both. We can use the direct IBM MQ-based interface for consumers of the service that would prefer the improved reliability that the IBM MQ Client can provide compared to an HTTP-based API.

In this section we explore how to expose the IBM MQ based "Command" implementation as a RESTful API façade. We keep the messaging layer in place beneath the API to continue to decouple the interaction with the data source and retain the benefits of the command pattern. Figure 6-134 on page 260 shows the extended model.

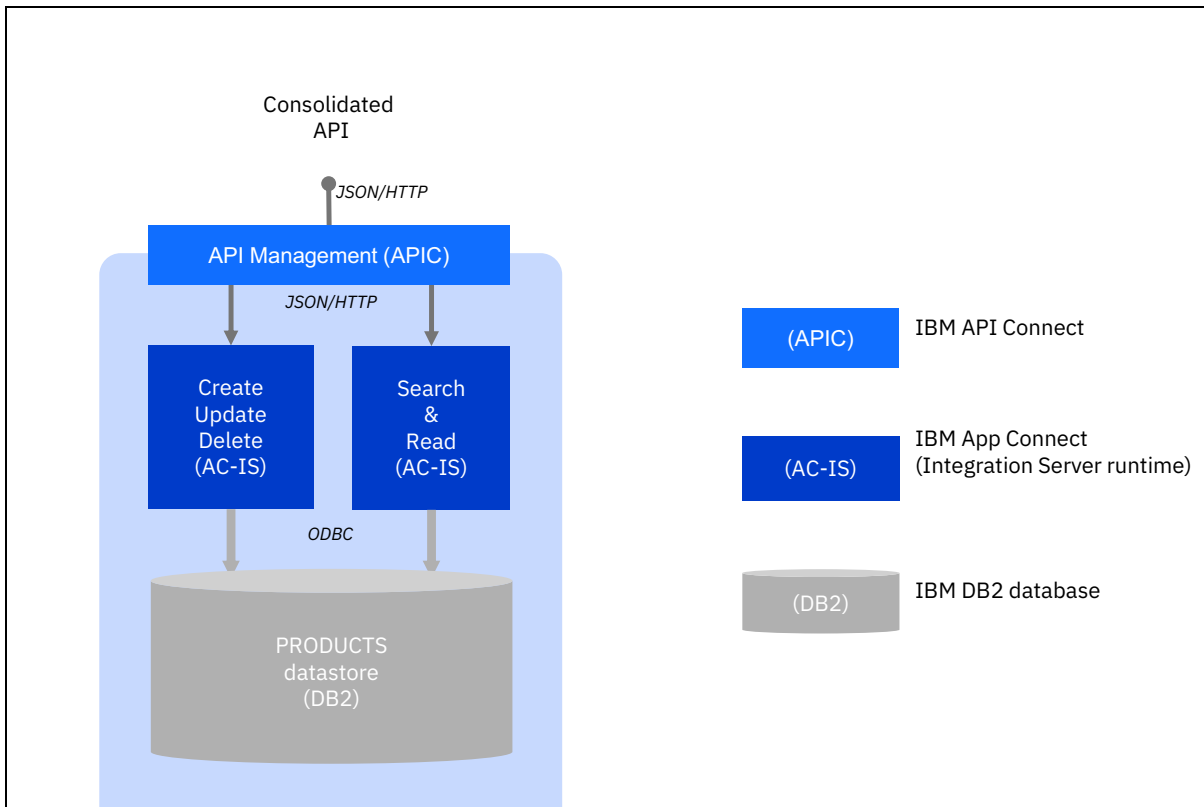


Figure 6-134 Command pattern exposed as an API

As in previous sections, we will use IBM API Connect (APIC) to expose the API providing discovery, access control, traffic management, governance, and security as discussed in 6.3, "Expose an API using API Management" on page 190.

APIC uses the OpenAPI specification formerly known as Swagger to model the API.

6.5.1 Defining the API data model

The first thing we need to include is the data models — known as "Definitions" in the API world — that will be supported by the API. We will use this information when we work in other areas of the API. This is equivalent to the product.json file we used in IBM App Connect. However, in this case you create the definition as part of the API, because we want the API specification be self-explanatory and to use it as documentation as well, as shown in Figure 6-135 on page 261.

Definitions Add		
API request and response payload structures are composed using OpenAPI schema definitions.		
NAME	TYPE	DESCRIPTION
product	object	⋮
responseMsg	object	⋮

Figure 6-135 Product definitions

The details for the product definition are shown in Figure 6-136.

Edit definitions

Name:

Type:

Description (optional):

Properties Add

PROPERTIES NAME	PROPERTIES TYPE	PROPERTIES EXAMPLE	PROPERTIES DESCRIPTION	DELETE
lastUpdate	string	2019-08-01T14:33:47Z	Date last time product was updated	🗑️
partNumber	integer	300	Product identifier	🗑️
productName	string	laptop	Name of the product	🗑️
quantity	integer	10	Number of items in stock	🗑️
description	string	Workstation for SCP42	Description of the product	🗑️

Additional properties Cancel Save

[Edit schema](#)

Figure 6-136 Product definitions- edit

6.5.2 Paths

Now that we have the definitions, we can move to the paths that will represent the location the API can be invoked. To keep a similar approach to the integration flows created in IBM App Connect, we are using three different paths, one for each command. But we can just as easily have one path because we are using different HTTP verbs for each command anyway. Figure 6-137 on page 261 shows the corresponding configuration.

Paths Add	
NAME	
/create	⋮
/update/{partNumber}	⋮
/delete/{partNumber}	⋮

Figure 6-137 Product configuration

Each path will have one operation. In the case of the create command, the convention is to use HTTP POST as shown in Figure 6-138.

The screenshot shows the configuration for a new API path. The path name is `/create`. Under the 'Operations' section, a single operation named 'POST' is listed and highlighted with a red box. The interface includes 'Add', 'Cancel', and 'Save' buttons.

Figure 6-138 POST operation

Some of the key aspects of the POST operation are highlighted in Figure 6-139 on page 262.

This screenshot details the configuration for the POST operation. Key aspects are highlighted with red boxes and numbered callouts:

- 1:** 'Add media type (optional)' field.
- 2:** 'REQUIRED' checkbox, which is checked.
- 3:** 'LOCATED IN' dropdown menu, set to 'body'.
- 4:** 'TYPE' dropdown menu, set to 'product'.
- 5:** 'SCHEMA' dropdown menu, set to 'responseMsg'.

Other visible settings include 'Override API Produce Types' and 'Override API Consume Types', both checked, with 'application/json' selected as the media type. The response description is 'The operation was successful.' with a status code of 200.

Figure 6-139 Key aspects of the POST operation

As shown in Figure 6-139, as part of the API definition you can specify the data type you will support. In this case we will continue with JSON as we did with the IBM App Connect

implementation (1). Since this is a create command, we have marked the input data as mandatory (2). The location of the data is the body of the request (3). The type is the product definition that we included before (4). Finally, we also specify the schema for the response message. In this case is a simple response, but in a real-life scenario this can be as complex as needed (5).

For the update command, the HTTP operation will be a PUT. In some implementations, API developers prefer to use PATCH, but usually that depends on the scope of the update, in other words, the whole resource or individual fields. In this case we will keep both scenarios under PUT, but you can expand the scenario to include PATCH on your own, based on the information provided here.

Our path configuration is as follows. However, in contrast to the create path, we have added the partNumber as a parameter, so that we can identify which product we want to update.

The screenshot shows a configuration interface for an API path. At the top, it says "Path" and "Paths identify the REST resources exposed by the API. An operation combines a path with an HTTP verb, parameters, and definitions for requests and responses. [Learn more](#)".

Under "Path name", there is a text input field containing "/update/{partNumber}", which is highlighted with a red box.

Below that is a "Path Parameters" section with an "Add" button. It contains a table with the following columns: REQUIRED, NAME, LOCATED_IN, TYPE, DESCRIPTION, and DELETE.

Next is an "Operations" section with an "Add" button. It contains a table with a "NAME" column, where "PUT" is listed and highlighted with a red box.

At the bottom right, there are "Cancel" and "Save" buttons.

Figure 6-140 Path configuration

Because we included the partNumber as part of the path you can see it is included as a new parameter for the operation. But it is located in the path instead of the body. See Figure 6-141 on page 264.

Parameters Add

REQUIRED	NAME	LOCATED IN	TYPE	DESCRIPTION	DELETE
<input checked="" type="checkbox"/>	partNumber	path	int 32	Product identifier	
<input checked="" type="checkbox"/>	body	body	product	The request body for the operation	

Response Add

STATUS CODE	SCHEMA	DESCRIPTION	DELETE
200	responseMsg	The operation was successful.	

Cancel Save

The path for the Delete command uses the corresponding HTTP Delete operation to process the request as shown in Figure 6-141 on page 264.

Path

Paths identify the REST resources exposed by the API. An operation combines a path with an HTTP verb, parameters, and definitions for requests and responses. [Learn more](#)

Path name

`/delete/{partNumber}`

Path Parameters Add

REQUIRED	NAME	LOCATED_IN	TYPE	DESCRIPTION	DELETE
----------	------	------------	------	-------------	--------

Operations Add

NAME	
DELETE	

Cancel Save

Figure 6-141 Path for the Delete command

In the case of the Delete operation, we need only the partNumber to identify the record to delete, so the configuration is shown in Example 6-5 on page 267.

Parameters						Add
REQUIRED	NAME	LOCATED IN	TYPE	DESCRIPTION	DELETE	
<input checked="" type="checkbox"/>	partNumber	path	string			

Response				Add
STATUS CODE	SCHEMA	DESCRIPTION	DELETE	
200	object	The operation was successful.		

6.5.3 Securing the API

A key benefit of using IBM API Connect to create and manage the API is the fact that you can add security to the API. Instead of giving the responsibility to the API developer, you can have a central location where you can enforce certain security policies that are applied globally regardless of the actual API back end and how it was implemented.

In this case we have added API key validation with Client Id and Client Secret. But APIC supports other options, including OAuth, which is a common approach in the API world, and is described in a later section. See Figure 6-142 on page 265.

Security Definitions			Add
Security definitions control client access to API endpoints, including API key validation, application user authentication, and OAuth . Learn more			
NAME	TYPE	LOCATED_IN	
clientSecret	apiKey	header	
clientIdHeader	apiKey	header	

Figure 6-142 Security definitions -1

To enable the security definitions that are specified in 6.5.3, “Securing the API” on page 265, you use the Security Tab. There, you have the opportunity to check which definitions that you want to use as part of the security policies. In this case we have enabled both. See Figure 6-143.

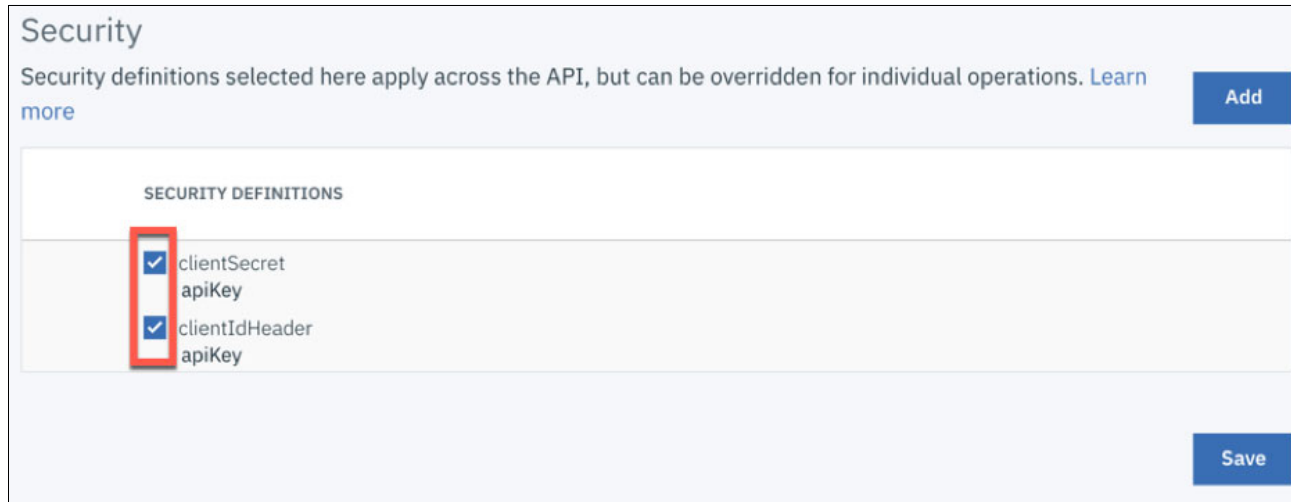


Figure 6-143 Security definitions -2

6.5.4 The Assembly

There are other properties that you can set as part of the API definition. But we will leave them with the defaults, and we will move to the “brains” of the API. The Assembly is where you will define the logic that your API will execute for each operation. It can be as simple as a proxy that invokes an existing API implemented already, for instance in IBM App Connect. But you can also take advantage of the many functions (known as policies) available in IBM API Connect to build an orchestration as complex as needed.

In this case, we are going to use the Operation Switch. It automatically allows you to create a case for each of the operations we have configured, with a GatewayScript that includes the functionality to interact with IBM MQ directly via the `urlopen` module. Additional details can be found in the APIC v2018 Knowledge Center article titled “*urlopen module*”:

https://www.ibm.com/support/knowledgecenter/SS9H2Y_7.7.0/com.ibm.dp.doc/urlopen_js.html#urlopen.targetformq

Note that we could have used the new messaging REST API that is available in IBM MQ v9.1, and we would not have needed to use any GatewayScript at all. However, we know that not all the customers have migrated to this version yet. So we decided to show a more generic approach that can be used immediately with previous versions of IBM MQ. If you are interested to explore the messaging REST API that is introduced in IBM MQ v9.1, read the Knowledge Center article titled “*Messaging using the REST API*”:

https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.1.0/com.ibm.mq.dev.doc/q130940_.htm

With this in mind, the Assembly to implement the API will look like Figure 6-144.

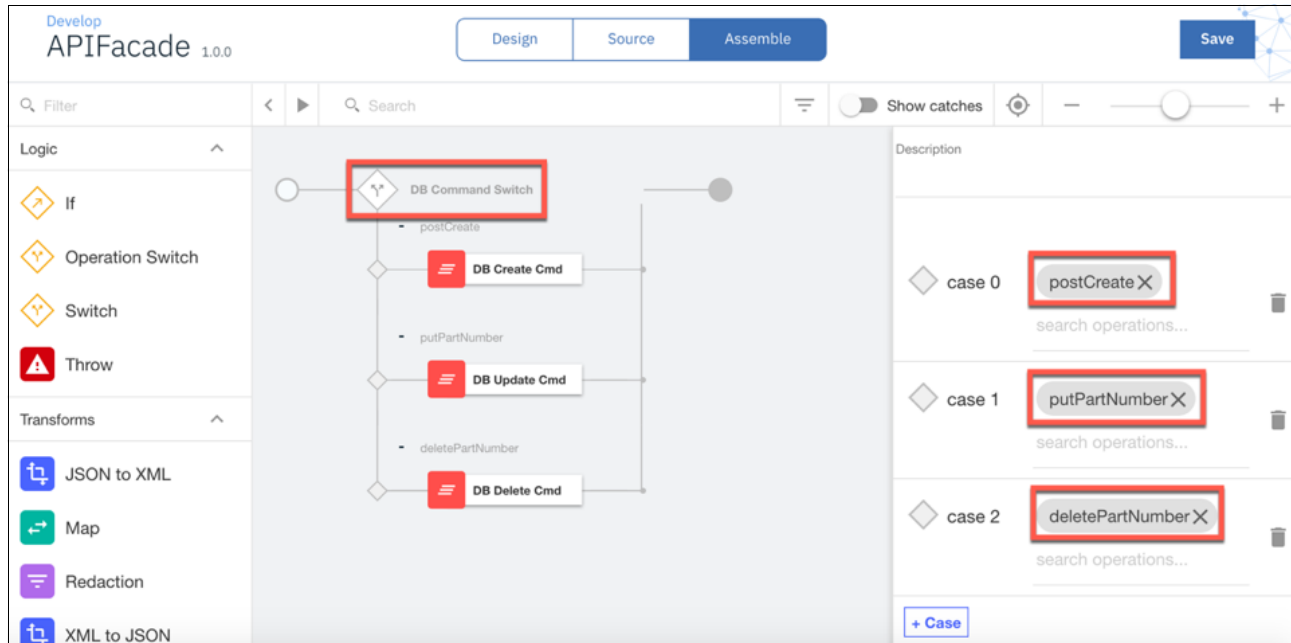


Figure 6-144 The Assembly to implement the API

The code details for each one of the gateway scripts are included in Example 6-5 below.

Example 6-5 Code details

```

DB Create Cmd:
var urlopen = require ('urlopen');
var putData = apim.getvariable('request.body');
var options =
{
  target:
'mq://XXX.XXX.XXX.XXX:YYYYY?QueueManager=mqicp4i;UserName=user11;Channel=ACE.TO.MQ;ChannelTimeout=3000;'
+
'ChannelLimit=2;Size=100000;MQCSPUserId=user11;MQCSPPassword=ZZZZZZZZZZ;RequestQueue=DB.CREATE;TimeOut=10000',
  data : putData,
  headers : { MQMD : {
    MQMD: {
      StructId : { $ : 'MD' } ,
      Version : { $ : '1' } ,
      Format: { $ : "MQSTR" }
    }
  }
};
urlopen.open (options, function (error, response) {} );
apim.setvariable('message.body',{ "statusMsg": "Command to create row was received successfully" });
DB Update Cmd:
var urlopen = require ('urlopen');
var putData = apim.getvariable('request.body');
putData.partNumber = apim.getvariable('request.parameters.partNumber');
var options =
{
  target: 'mq://
XXX.XXX.XXX.XXX:YYYYY?QueueManager=mqicp4i;UserName=user11;Channel=ACE.TO.MQ;ChannelTimeout=3000;'

```

```

        + 'ChannelLimit=2;Size=100000;MQCSPUserId=user11;MQCSPPassword=
ZZZZZZZZZZ;RequestQueue=DB.UPDATE;TimeOut=10000',
    data : putData,
    headers : { MQMD : {
        MQMD: {
            StructId : { $ : 'MD' } ,
            Version : { $ : '1' } ,
            Format: { $ : "MQSTR" }
        },
    }
};
urlopen.open (options, function (error, response) {} );
apim.setvariable('message.body',{ "statusMsg": "Command to update row was received successfully" });
DB Delete Cmd:
var urlopen = require ('urlopen');
var putData = apim.getvariable('request.parameters');
var options =
{
    target: 'mq://
XXX.XXX.XXX.XXX:YYYYY?QueueManager=mqicp4i;UserName=user11;Channel=ACE.TO.MQ;ChannelTimeout=3000;'
    + 'ChannelLimit=2;Size=100000;MQCSPUserId=user11;MQCSPPassword=
ZZZZZZZZZZ;RequestQueue=DB.DELETE;TimeOut=10000',
    data : putData,
    headers : { MQMD : {
        MQMD: {
            StructId : { $ : 'MD' } ,
            Version : { $ : '1' } ,
            Format: { $ : "MQSTR" }
        },
    }
};
urlopen.open (options, function (error, response) {} );
apim.setvariable('message.body',{ "statusMsg": "Command to delete row was received successfully" });

```

6.5.5 API testing

After we are satisfied with the API we can test it right there in the Assembly which provides productivity benefits for the API Developer.

1. To enter in test mode, you click **Play** in the assembly diagram.

This will open the Test section where you will have the opportunity to Republish the Product in case you have made any update recently. You might notice that the user interface refers to *Product* and not API. A Product is an artifact that allows you to package many products to help manage multiple APIs where you can define rate limits among other things. See Figure 6-145.

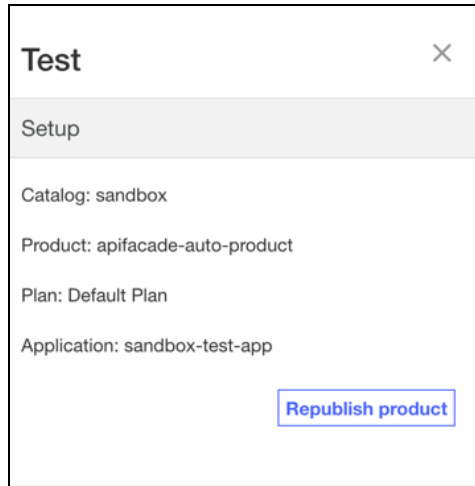


Figure 6-145 API testing -1

2. After you have republished the product if needed, you can select the operation you want to test. Let's start creating a new product. See Figure 6-146.

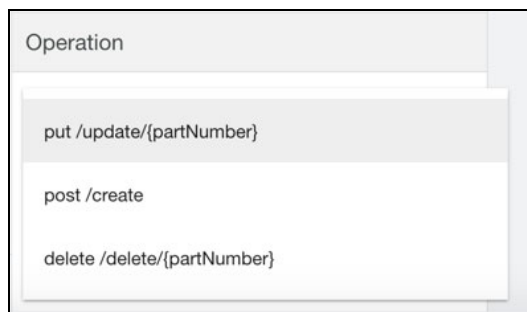


Figure 6-146 API testing -2

3. After that a new section is opened where you are required to enter the security information we defined before. See Figure 6-147 on page 269.

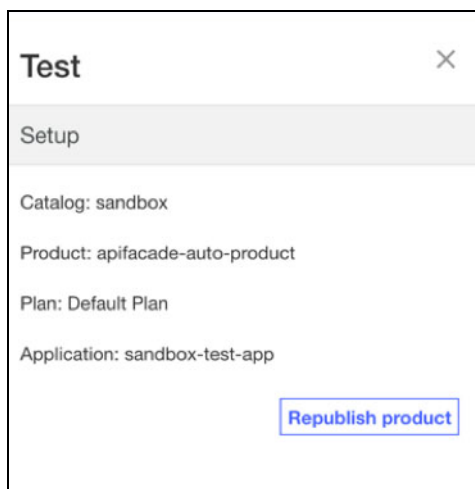


Figure 6-147 API testing -3

In this case, the tool creates test credentials to simplify the process. But when this is promoted to another tiers, the App Developers are required to obtain their own API Keys.

4. When you continue scrolling down, you will be able to enter the parameters required to invoke the API. The tool allows you to generate test data. If you prefer, you can type or copy and paste some sample test data, you might have already available. See Figure 6-148.



Figure 6-148 API testing -4

5. After you have entered all the information you can invoke the API, and you have the opportunity to repeat the invocation multiple times if needed. See Figure 6-149 on page 270.

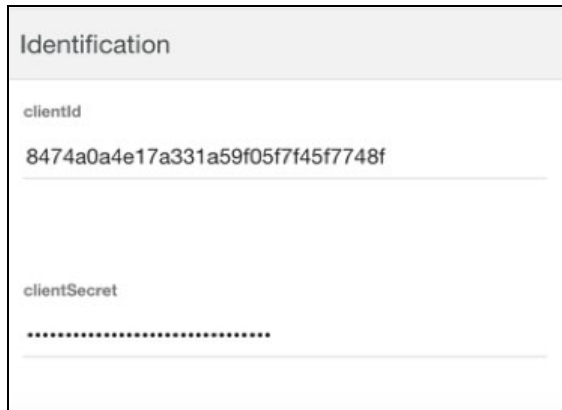


Figure 6-149 API testing -5

6. After a moment the result of the API invocation is returned, where you can see the status code and the body of the response message. In this case we can see a successful invocation (200) as well as the response message we defined in the gatewayscript code. See Figure 6-150.

The screenshot shows an API testing interface. At the top, there is a section labeled "parameters". Below this, there are two dropdown menus. The first is labeled "Content-Type" and has "application/json" selected. The second is labeled "Accept" and also has "application/json" selected. Below the dropdowns, there is a section titled "The request body for the operation" with a sub-label "body *". The request body is a JSON object:

```
{ "lastUpdate": "2019-08-01T14:31:33.473Z",  
  "partNumber": 20,  
  "productName": "laptop",  
  "quantity": 5,  
  "description": "Workstation for ICP4I"}
```

 At the bottom of the interface, there are two buttons: "Show schema" and "Generate". The "Generate" button is highlighted with a red border.

Figure 6-150 API testing -6

6.5.6 API socialization

At this point your API is ready to be consumed by App Developers. The App Developers will use another component of the APIC platform known as the *Developer Portal* where they can search for available APIs, get all the required information to invoke the API, register the App that will be used to consume the API and request the API keys associated with the App.

For simplicity here you have a screen capture of the Developer Portal (Figure 6-151).

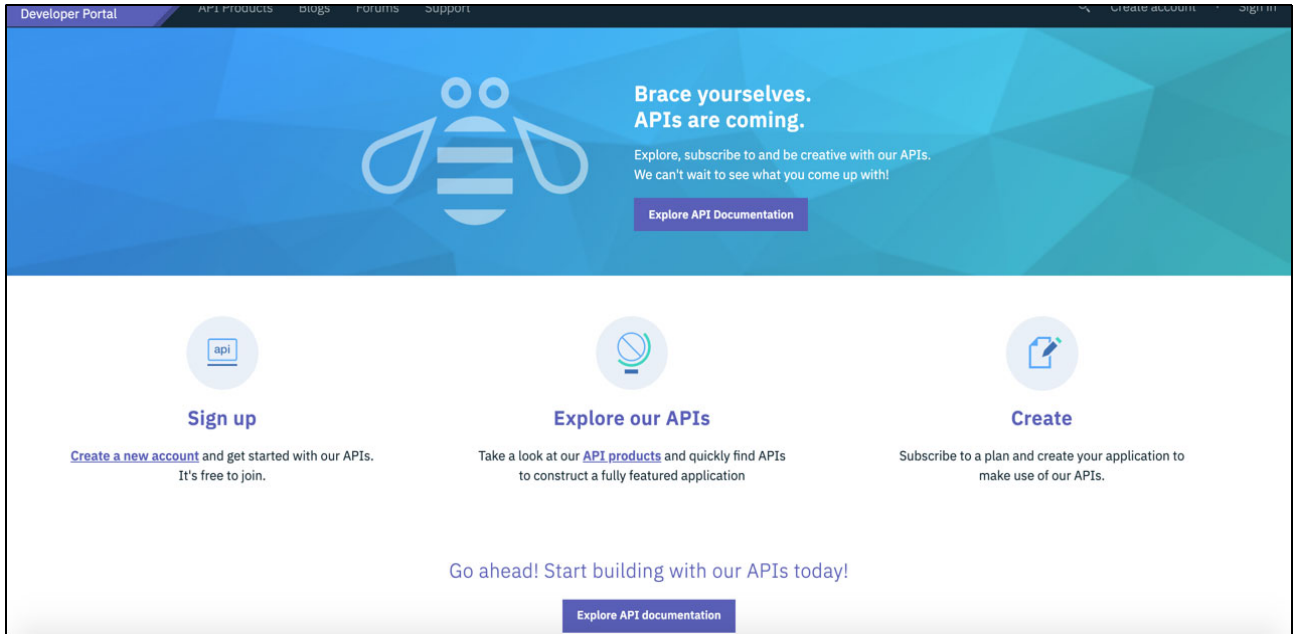


Figure 6-151 Developer Portal

1. Searching the catalog, we can find the API that we have defined. It is associated to an “auto” product because we leverage the automatic publication. But if needed we could create a different product to control and configure all the aspects of the product.
2. If you click on the **API name**, you see the API details in Figure 6-152.

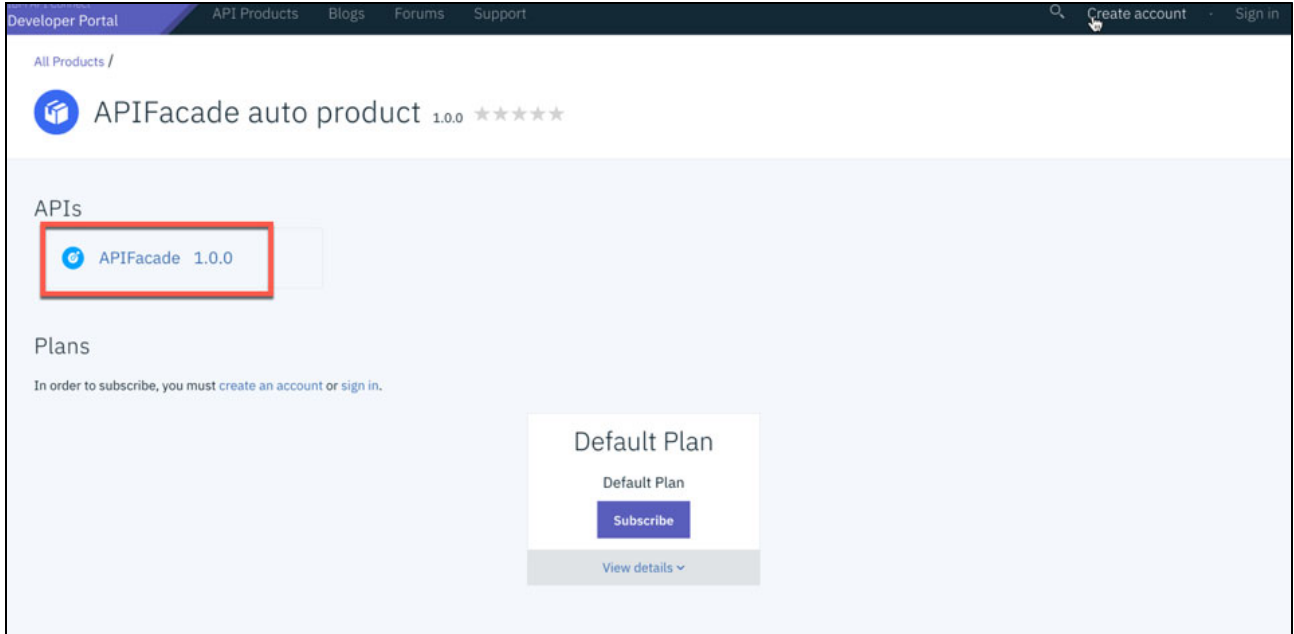


Figure 6-152 APIFacade 1.0.0

Figure 6-153 on page 273 shows the API details.

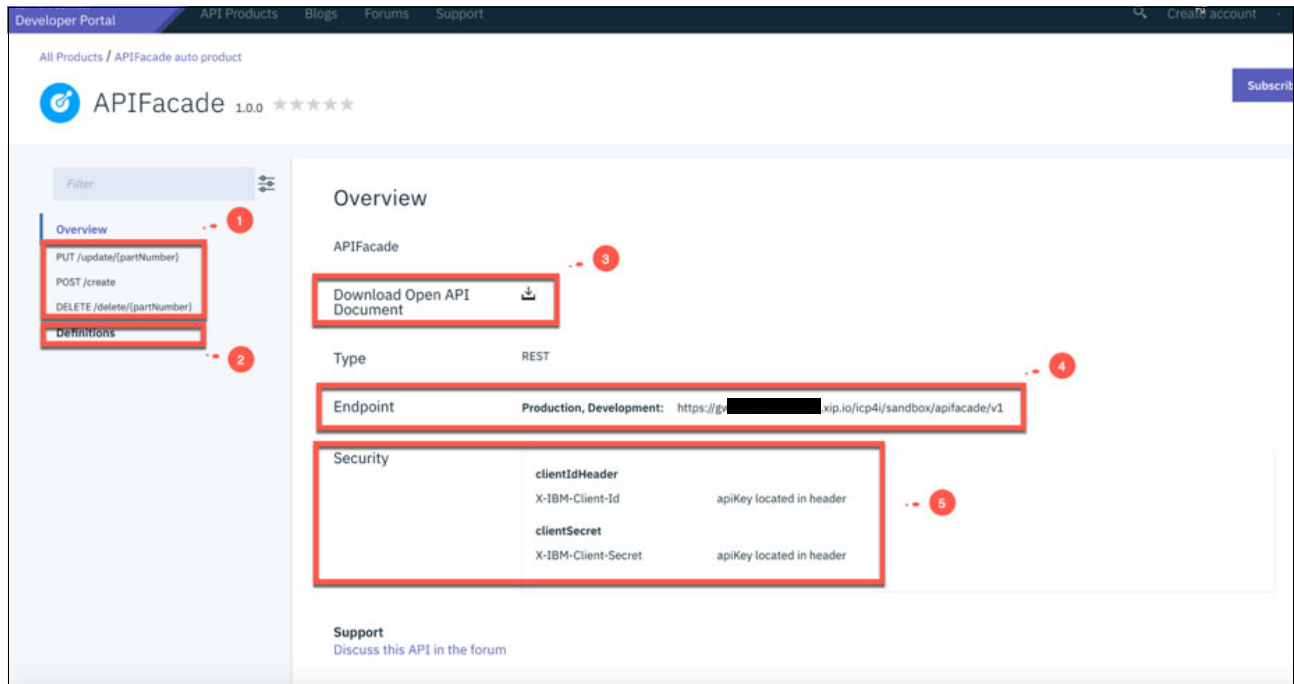


Figure 6-153 APIFacade overview

3. As shown in Figure 6-153:

You will find the three operations we defined for the API (1). It also includes the information about the data models used by the API (2).

You have the opportunity to download the OpenAPI document for the API to use it when developing the App that will consume this API (3).

The API includes information about the actual endpoint that you will need to use to invoke the API (4). It also includes the security information (5), so the App Developer is aware right away of the security requirements associated with the API.

4. If you select the **Definitions**, you get the details about the data models as shown in Figure 6-154 on page 274. This is why you should include as much information as possible at the time of the API creation to serve as documentation. Consider including the JSON schema and also an example.

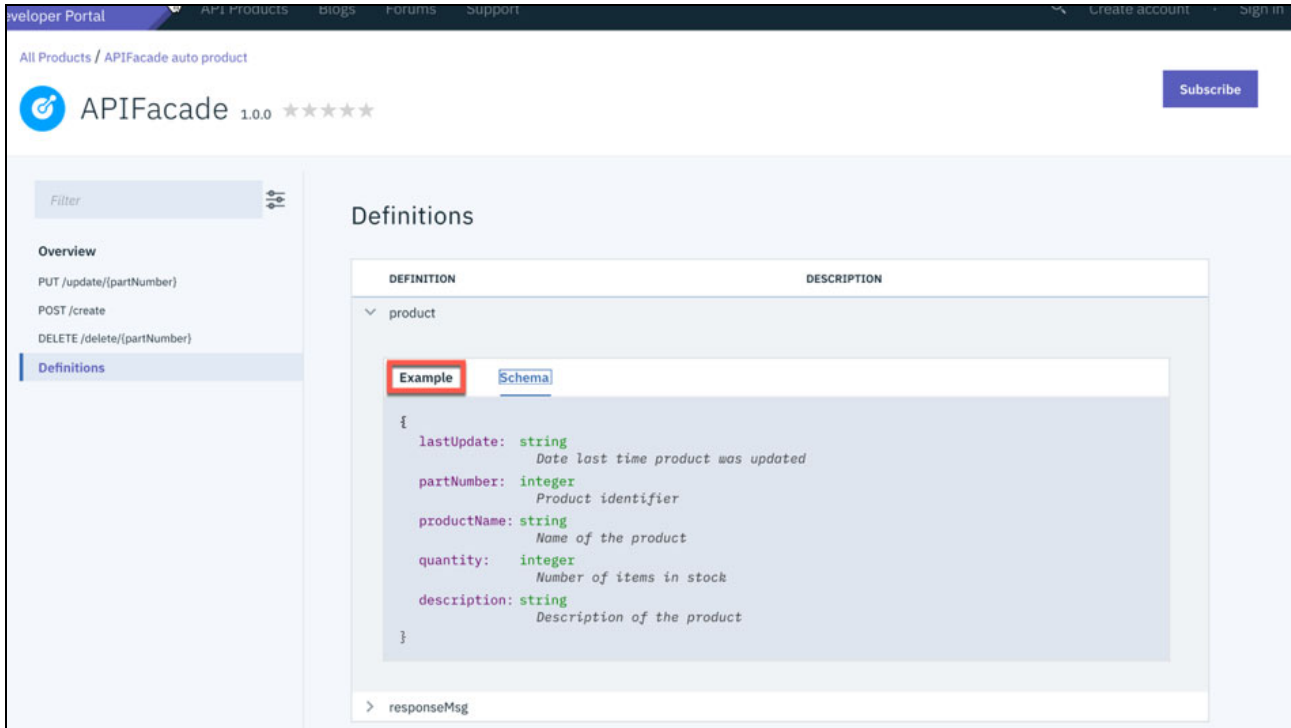


Figure 6-154 Definitions

- When you select an operation you are presented with the corresponding details, and you can even try it directly from the portal to improve the App Developer productivity. See Figure 6-155.

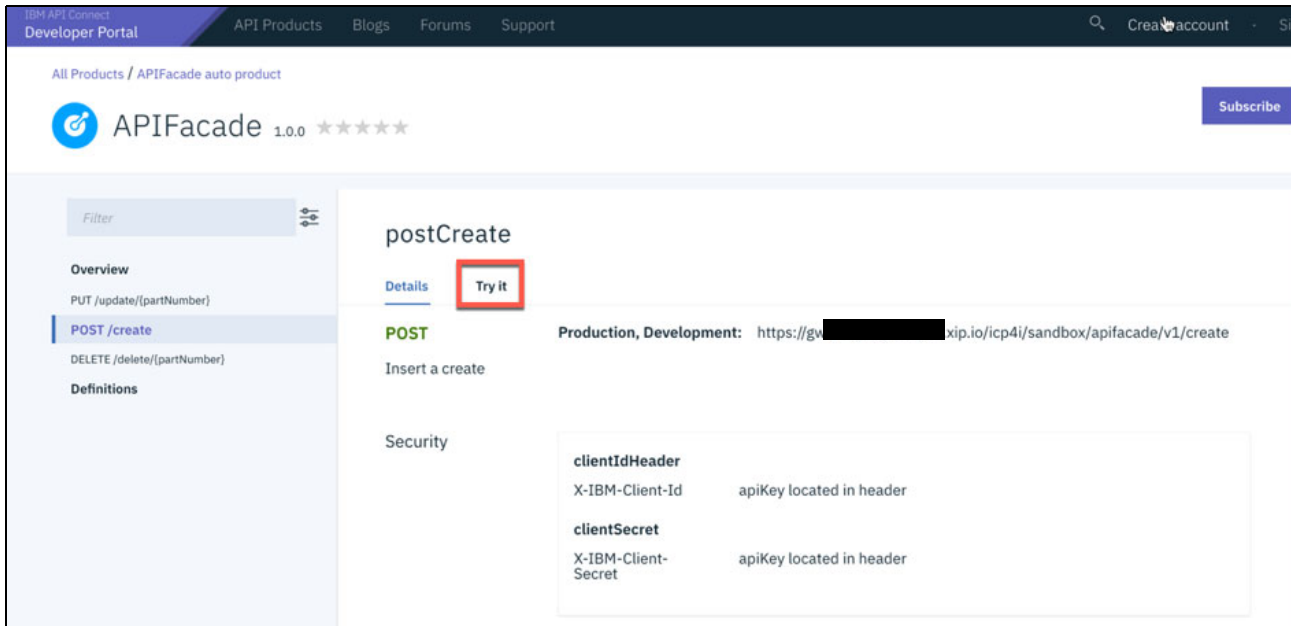


Figure 6-155 Try it

- When you continue scrolling down you will find the rest of the information about the parameters. Almost at the bottom, you will find code snippets in many languages,

including Java, Node, and Swift among others, helping the App Developer to accelerate the development process. See Figure 6-156 on page 275.

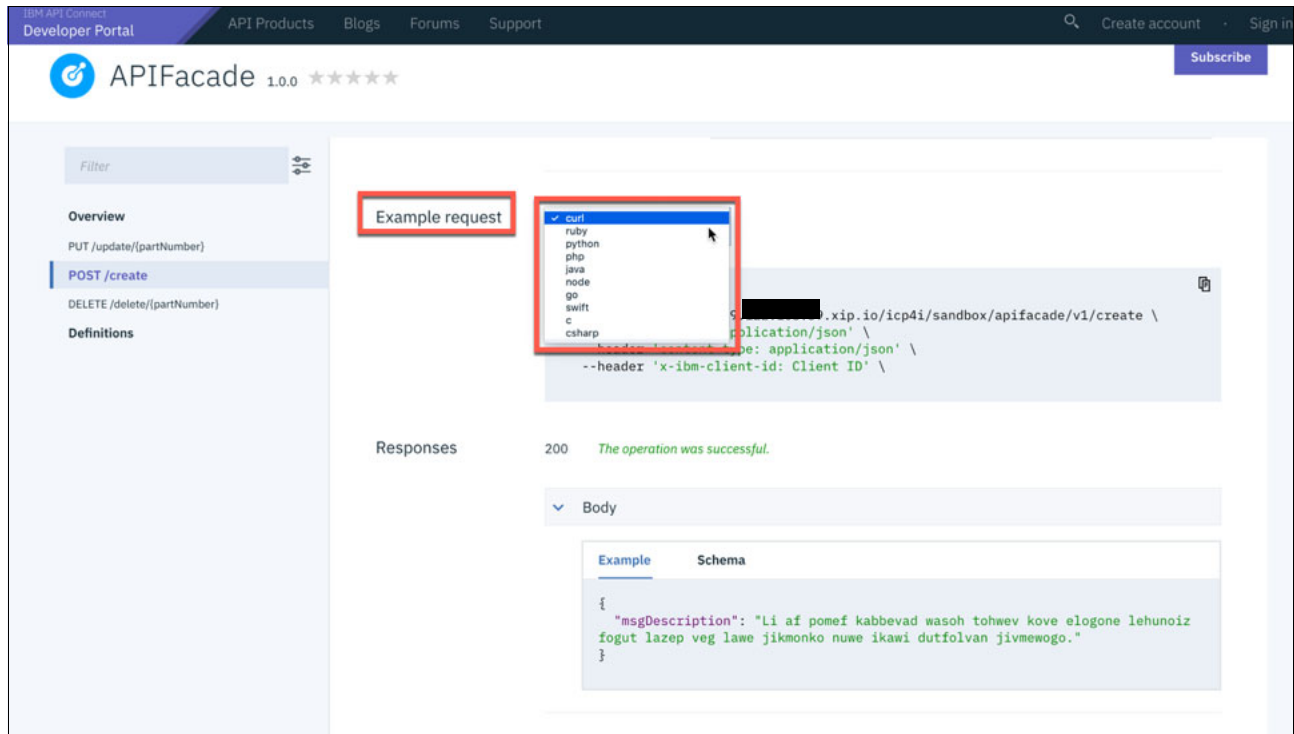


Figure 6-156 Example request

7. When you explore the **Try It** tab you see the screen to invoke the API. The whole process has been documented in a different section of this document. But let's give special attention to the message that states, "Log in to try this API", which is a key security aspect of APIC.

The goal is that anyone could easily find all the APIs that you have created. But in case they want to actually try it, they will need to sign in to the Portal. And in case they do not have an account, they can create one with the self-service capabilities. It is important to mention that all these capabilities are configurable. As a result, you can make the APIs visible to everybody to enable the self-service functionality and everything in between. See Figure 6-157 on page 276.

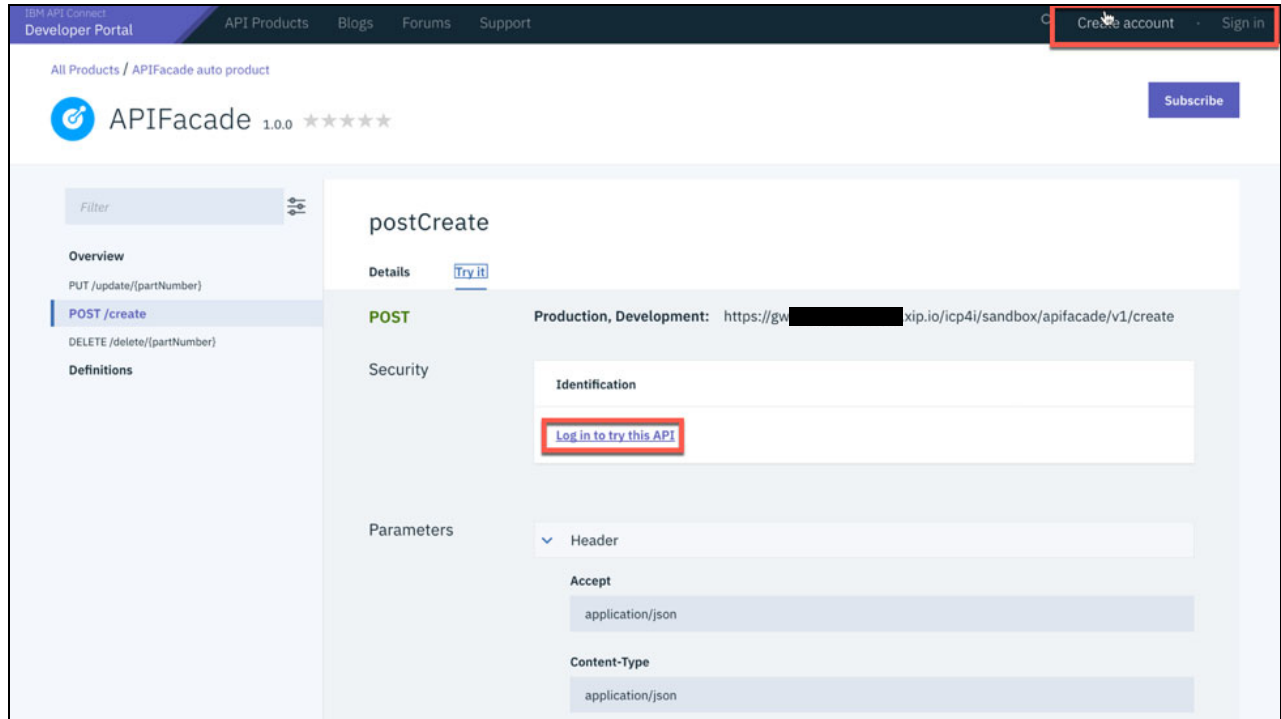


Figure 6-157 Sign in

6.5.7 Conclusion

In this section we have demonstrated how you can extend the Command side of the Command Query Responsibility Segregation (CQRS) pattern to expose the functionality as an API. In this way, you remove the IBM MQ dependency from the client side. But you continue to have the benefits that IBM MQ provides, plus all the API Management capabilities provided by APIC, including API security enforcement and API socialization among others.

6.6 Advanced API security

API management has enabled us to effectively hide the implementation from consumers. They see the implementation as a black box component with only one way in, via the API gateway as shown in Figure 6-158 on page 277.

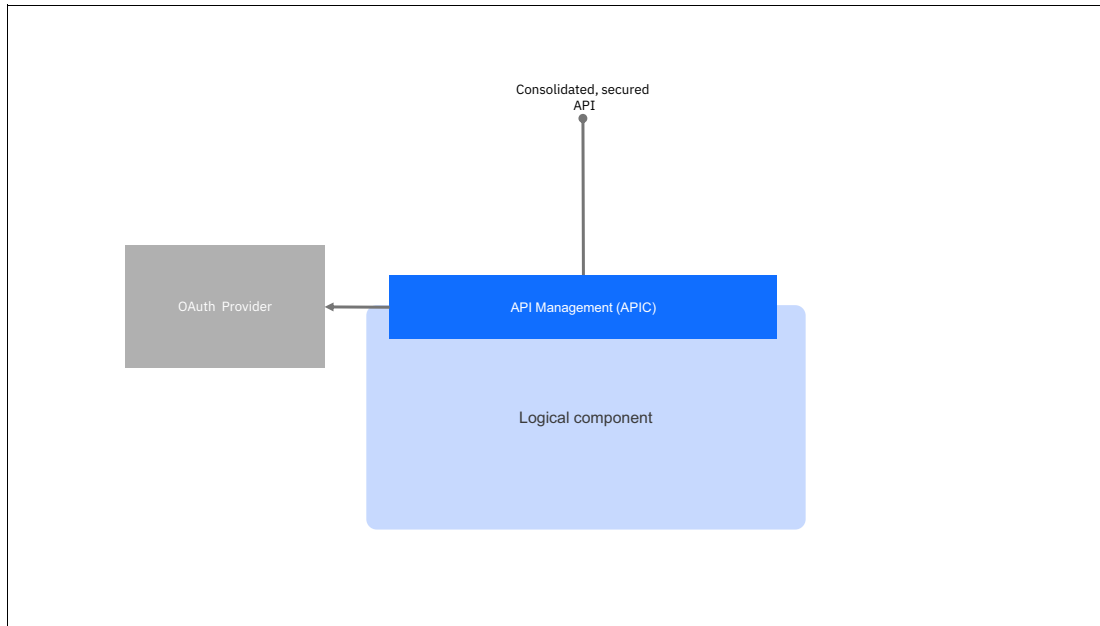


Figure 6-158 OAuth-based API security

We can now choose from a range of options to add sophistication to how we secure access to the API. In this section we will enable OAuth to control access.

IBM API Connect supported by DataPower provides advanced security features, which include but are not limited to, OAuth, JWT, encryption, throttling, etc. As described in section 4.9, “Cloud-native security – an application-centric perspective” on page 128 we can cover many different use cases.

We will demonstrate how to secure your API with OAuth token. We will also discuss the JWT token generation.

First you must identify which scheme you want your flow to follow. For more information, see the following article:

https://www.ibm.com/support/knowledgecenter/en/SSMNE5.0.0/com.ibm.apic.toolkit.doc/tutorial_apionprem_security_oauth.html

To choose an OAuth scheme. You must first establish whether your implementation is considered public or confidential. This will narrow your choices to three schemes. A brief outline of each scheme and the characteristics of the three public and three confidential schemes follows.

In our example here, we will use a Confidential (For internal application) scheme with password flow to demonstrate how to secure the API.

A confidential scheme is suitable when an application is capable of maintaining the secrecy of the client secret. This is usually the case when an application runs in a browser and accesses its own server when it gets OAuth access tokens. As such, these schemes make use of the client secret.

In Figure 6-159 we are showing the overall scenario, in which the developer initiates the request using one of the available channels (mobile, web), then the application uses the confidential scheme to obtain the token from the gateway. Then if authorized, the application calls the back-end microservice to get the data.

Figure 6-159 is an overview of the full scenario for this use case.

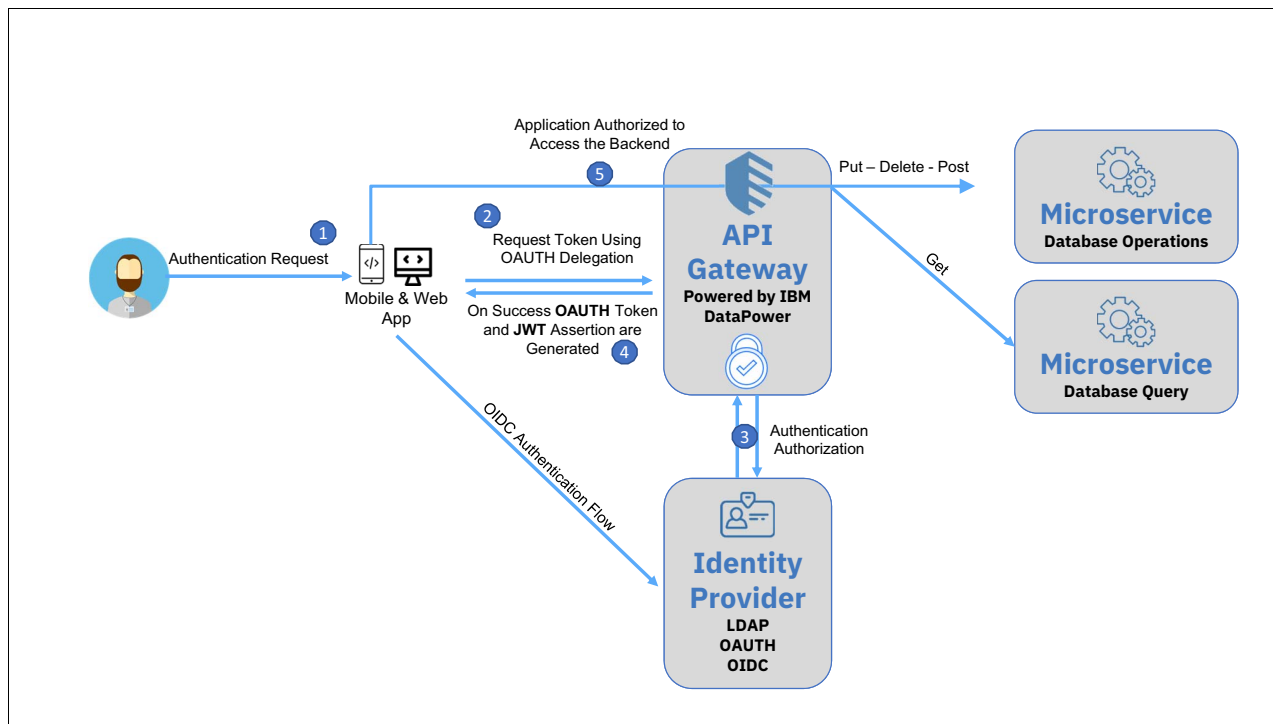


Figure 6-159 Scenario overview

6.6.1 Import the API into IBM API Connect

We described this procedure in the preceding section 6.3.1, “Importing the API definition” on page 191.

6.6.2 Configure the API

We described this procedure in the preceding section 6.3.2, “Configuring the API” on page 196.

6.6.3 Add basic security to the API

We described this procedure in the preceding section 6.3.4, “Add simple security to the API” on page 202.

6.6.4 Test the API

We described this procedure in the preceding section “Test the product” on page 205.

6.6.5 Securing the API Using OAUTH

Securing the API with OAUTH is divided into two parts, first the user repository and second the token issuer. They each could be using different systems or could be within the same platform.

In this demo we are going to utilize and create a DataPower Basic Authentication service that is using the DataPower Information file. Figure 6-160 on page 279 shows our scenario.

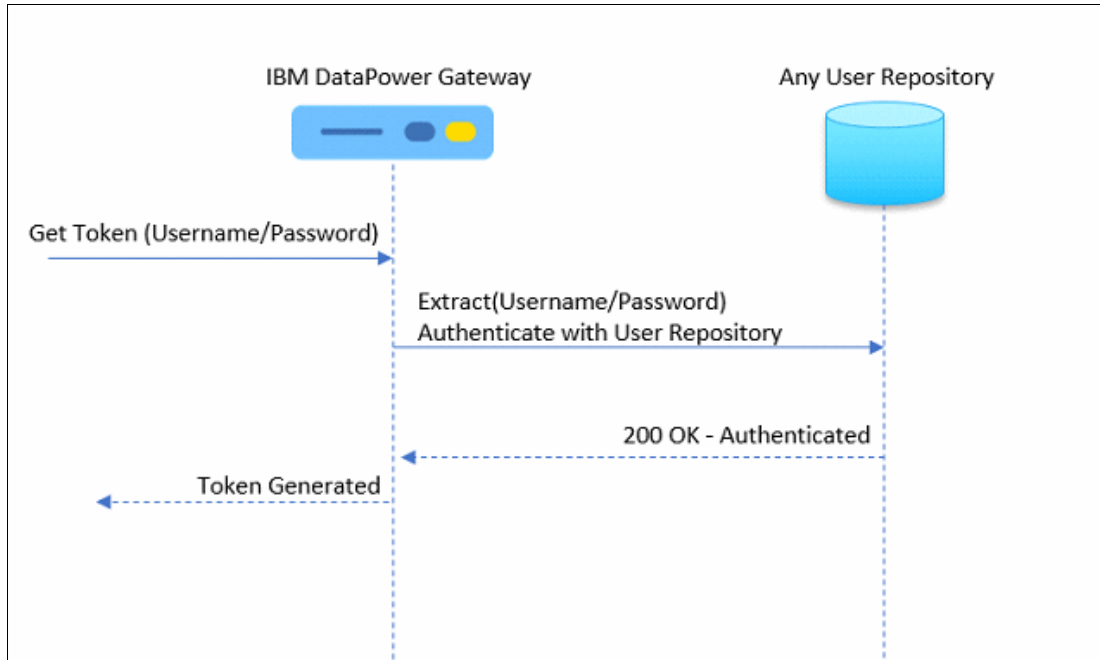


Figure 6-160 Our scenario

Importing the DataPower Auth URL

Perform the following steps to import the configuration into IBM DataPower that will enable it to offer a basic authentication service over a URL.

1. Import the following XML Firewall to IBM DataPower.

<https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/IBMRedBookDPAAuth.zip>.

To do that, **Log in** to the IBM DataPower interface, then click **Import Configurations**.

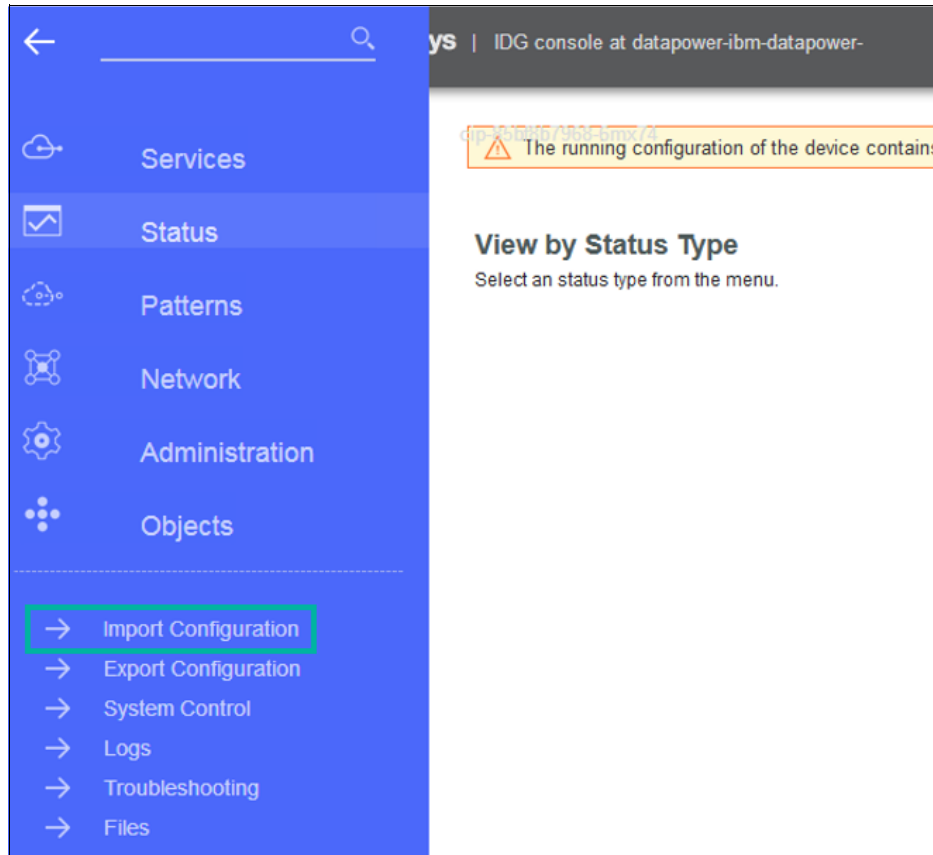


Figure 6-161 DataPower OAUTH URL Import 1

2. Browse for the downloaded file then click **Next** → **Select all** → **Import** → **Close**.

For step-by-step details, refer to 6.7, “Create event stream from messaging” on page 322.

API Connect OAUTH configurations

Back in IBM API Connect, we want to configure our API to use OAUTH. First it will need a user registry. We will use the DataPower service that were created in the previous step.

The user registry can be any type of user registry like an LDAP based registry, Auth URL or the local registry of the platform.

Create user registry

Perform the following steps in IBM API Connect:

1. Click on **Resources** → **User Registries** and then click on **Create**. See Figure 6-162.

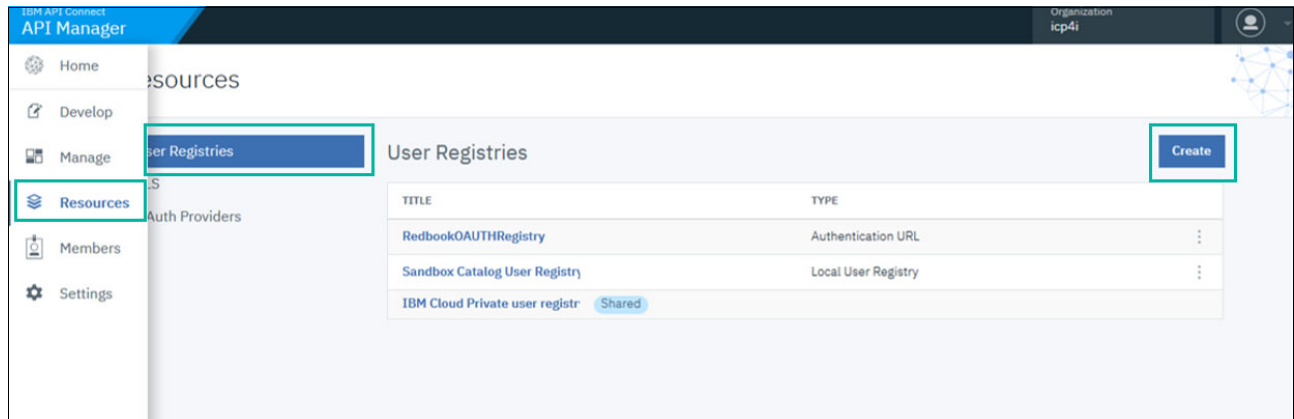


Figure 6-162 Creating user registry 1

2. You will have three options
 - Authentication URL User Registry (This is our option with DataPower Auth Service)
 - LDAP User Registry
 - Local User Registry

Choose **Authentication URL User Registry** and click **Next**. See Figure 6-163 on page 281.

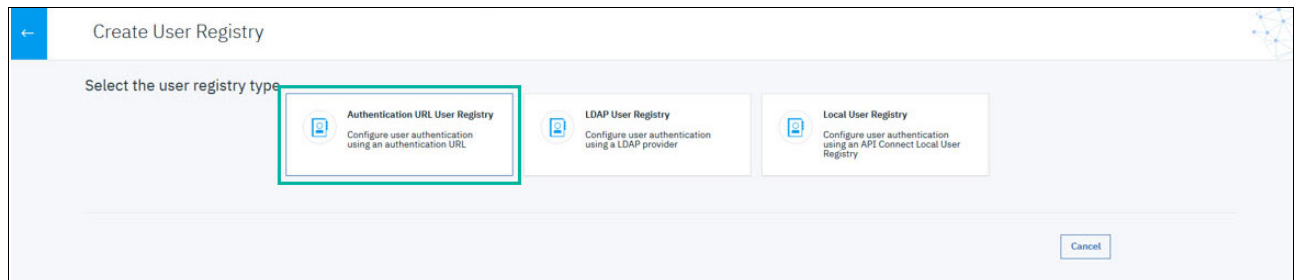


Figure 6-163 Creating user registry 2

3. Type in:
 - Title: RedbookOAUTHRegistry
 - URL: DataPower Authentication URL
 Click **Save**. See Figure 6-164.

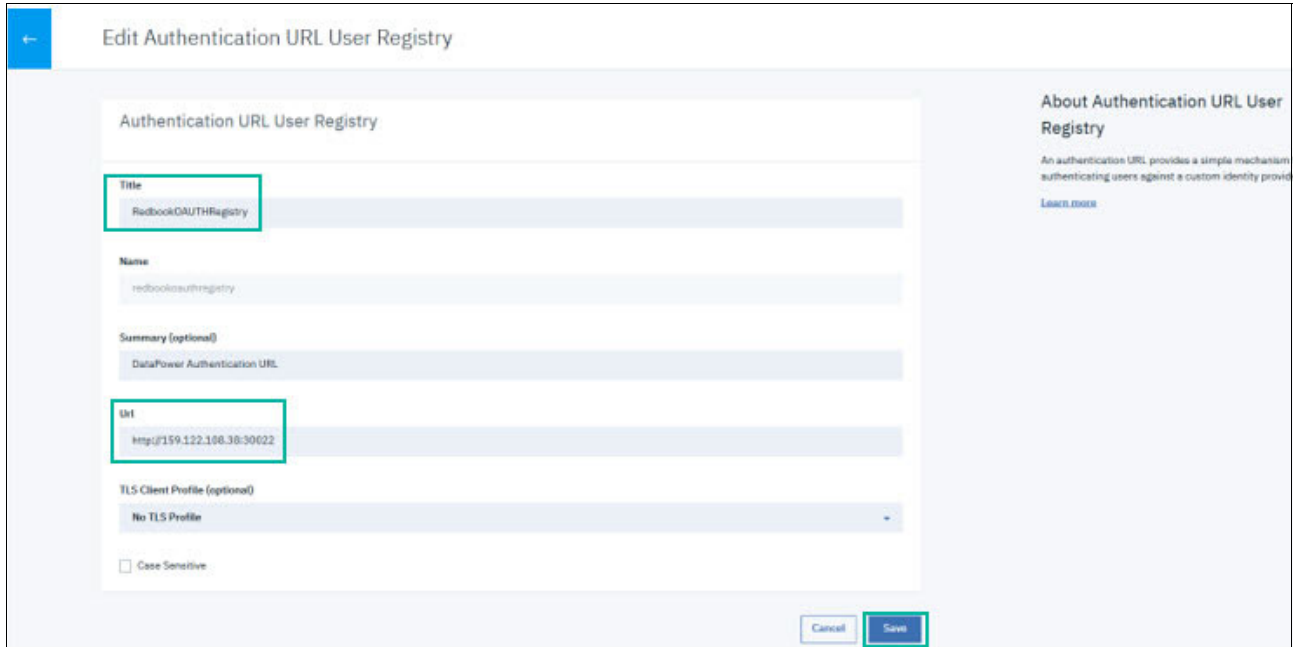


Figure 6-164 Creating user registry 3

Create OAuth Provider

IBM API Connect provides a native OAUTH provider. Perform the following steps to create an OAUTH provider:

1. Under **Resources**, click on **OAuth Providers**, then click on **Add** and choose **Native OAuth provider**. See Figure 6-165.

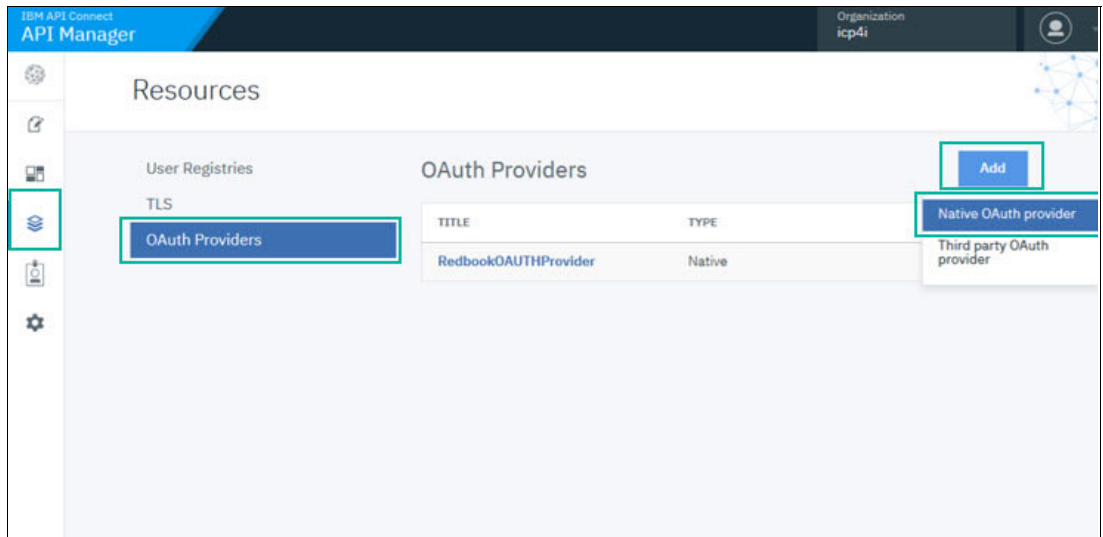


Figure 6-165 Creating user registry 4

2. Fill up the title RedbookOAUTHProvider and use **DataPower V5** then click **Next**. See Figure 6-166 on page 283.

Figure 6-166 Creating user registry 5

3. There are different options for the supported grant types:

- **Implicit:** An access token is returned immediately without an extra authorization code exchange step.
- **Application:** Application to application. Corresponds to the OAuth grant type “Client Credentials.” Does not require User Security.
- **Access code:** An authorization code is extracted from a URL and exchanged for an access code. Corresponds to the OAuth grant type “Authorization Code.”
- **Resource owner password:** The user’s username and password are exchanged directly for an access token, so can be used only by first-party clients.

Choose **Resource owner - Password** to exchange the basic authentication credentials with the server to get the token. We are choosing this because we want capture to the username and password to obtain the OAUTH token.

Then click **Next**. See Figure 6-167 on page 284.

← Create Native OAuth Provider

Configuration

Authorize path
/oauth2/authorize

Token path
/oauth2/token

Supported grant types

Implicit
 Application
 Access code
 Resource owner - Password

Supported client types

Confidential
 Public

Back Cancel Next

Figure 6-167 Creating user registry 6

4. Define the scopes that you want to use for your API, this could be the base path of your API, in this case it will be database_operations then click **Next**. See Figure 6-168.

← Create Native OAuth Provider

Scopes

Add scopes for this OAuth Provider. Add

NAME	DESCRIPTION	DELETE
database_operations	API Scope	

Back Cancel Next

Figure 6-168 Creating user registry 7

5. Choose the following options:
 - Collect credentials using: Basic Authentication
 - Authenticate application users using: RedbookOAUTHRegistry
 - Authorize application users using: Authenticated
Then click **Next**. See Figure 6-169 on page 285.

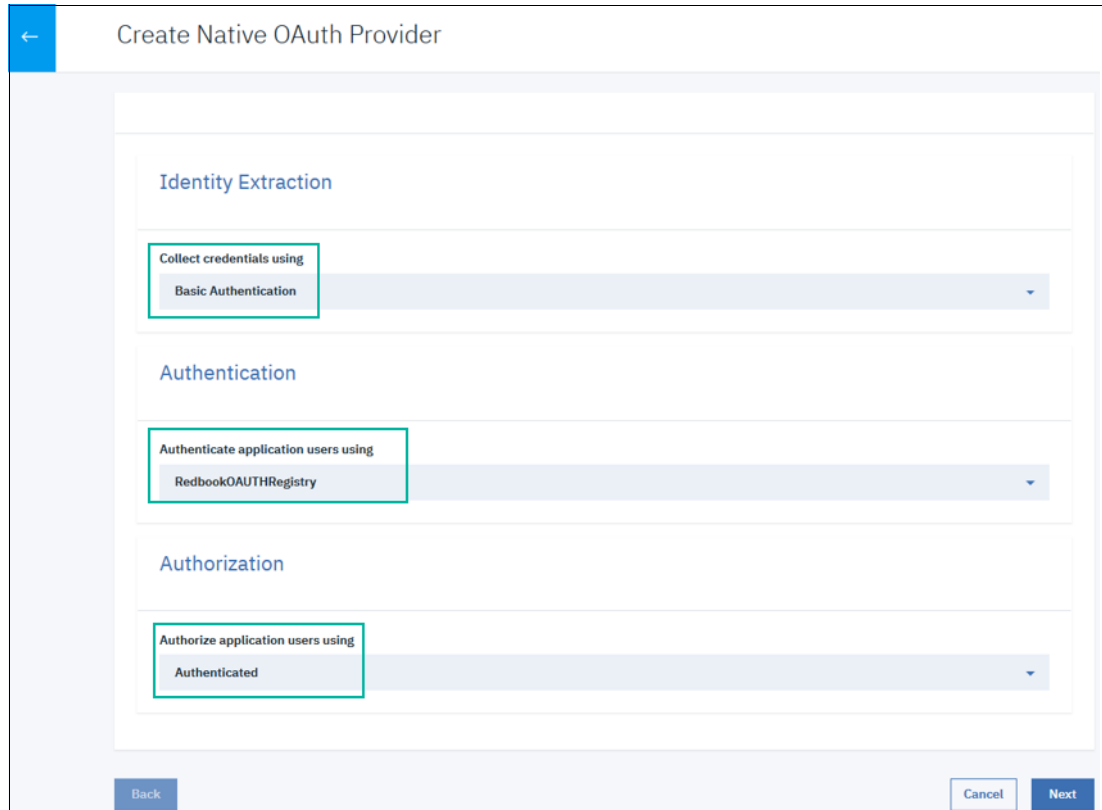


Figure 6-169 Creating user registry 8

- An option to create a sample user registry will be available if you do not have any configured user registry. See Figure 6-170.

Tip: If you didn't create the registry in the previous step, you can simply click **Create Sample User Registry**.

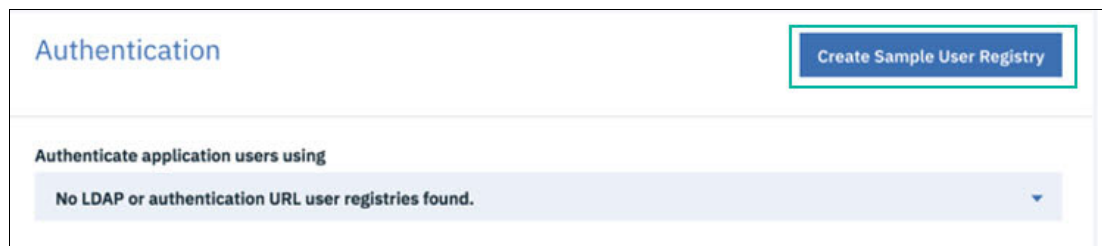


Figure 6-170 Creating user registry 9

- Click **Finish**. See Figure 6-171 on page 286.

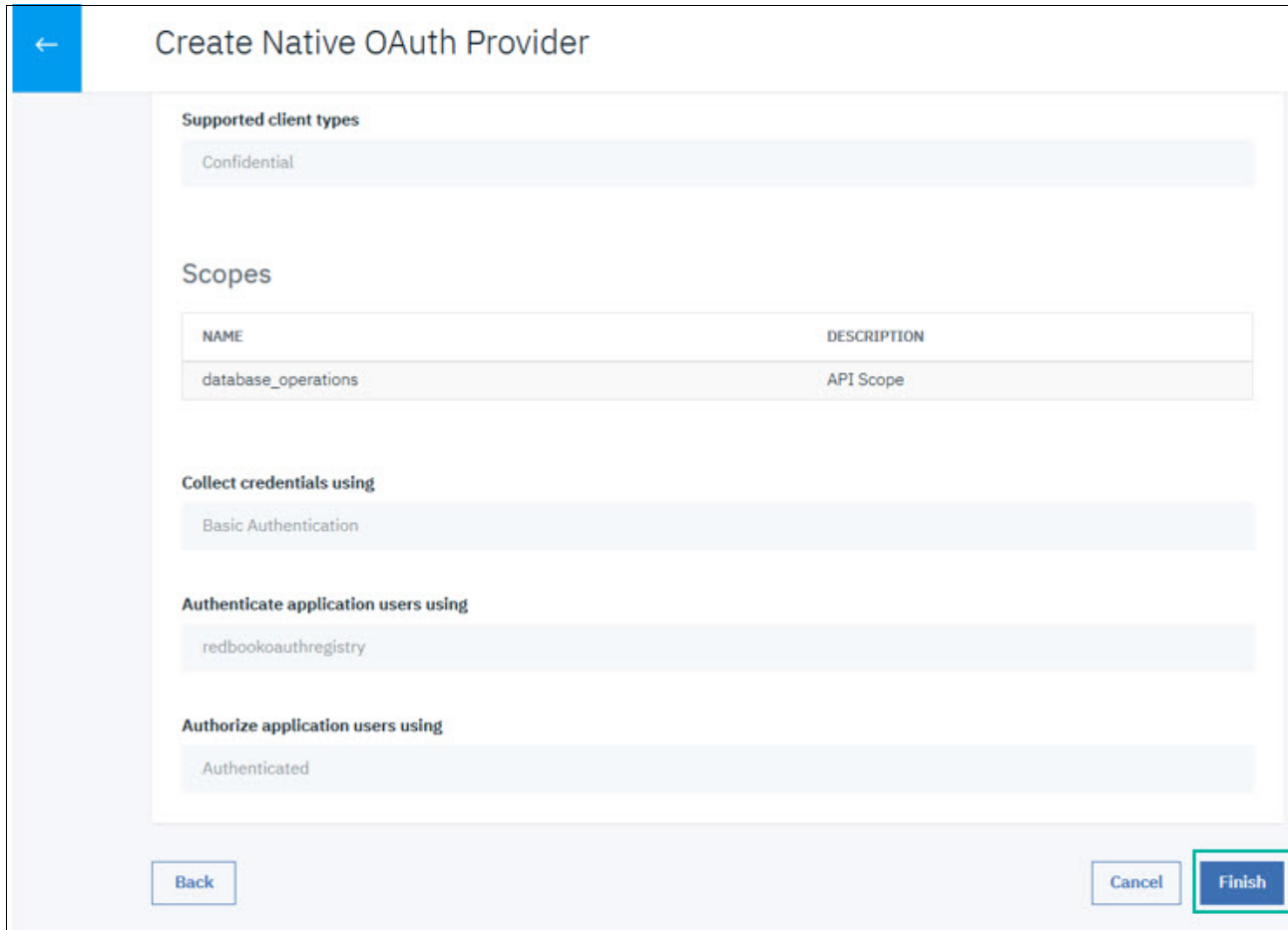


Figure 6-171 Creating user registry 10

Next, we must add the OAuth Provider to the gateway catalog and the API that we need to secure.

Adding the OAuth Provider to the gateway catalog

Perform the following steps:

1. From the left side menu click **Manage** and choose **Sandbox**. See Figure 6-172.

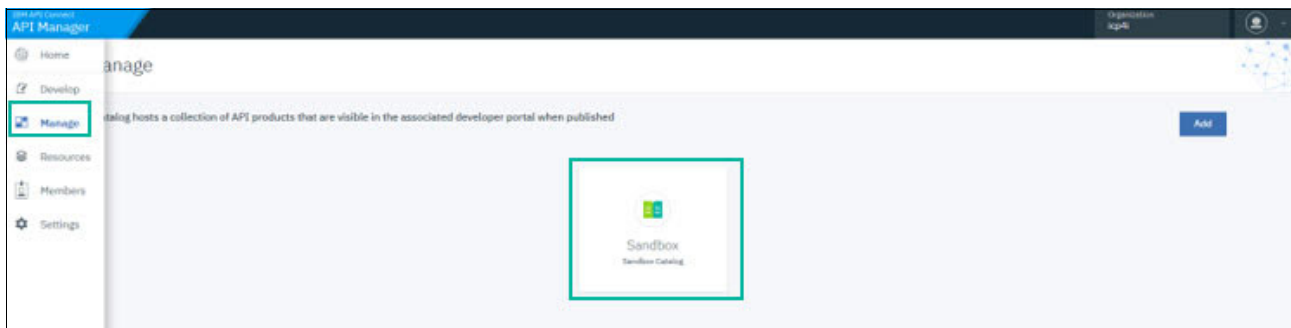


Figure 6-172 Adding the OAuth provider to the gateway 1

2. Again (inside the Sandbox Page) from the left side menu click on **Settings**. See Figure 6-173 on page 287.

Tip: You must be inside the Sandbox page to see the Catalog Settings.

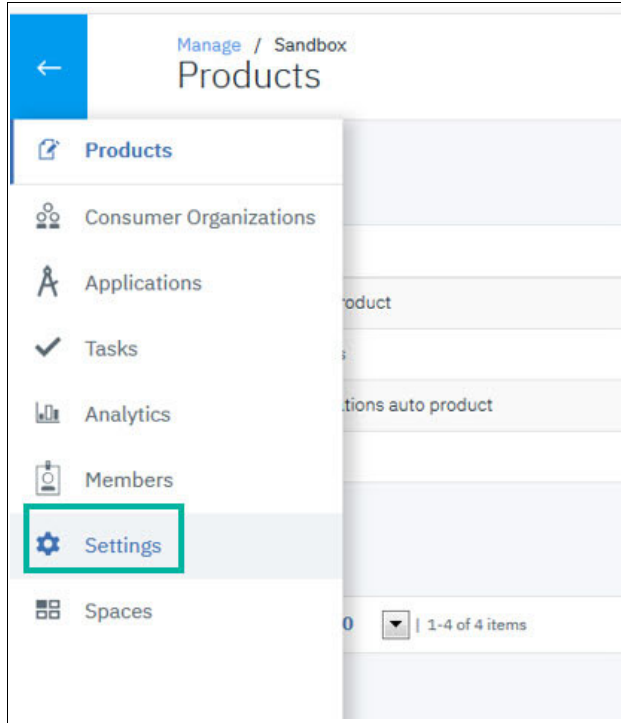


Figure 6-173 Adding the OAuth provider to the gateway 2

3. Click **API User Registries** then click **Edit**. See Figure 6-174.

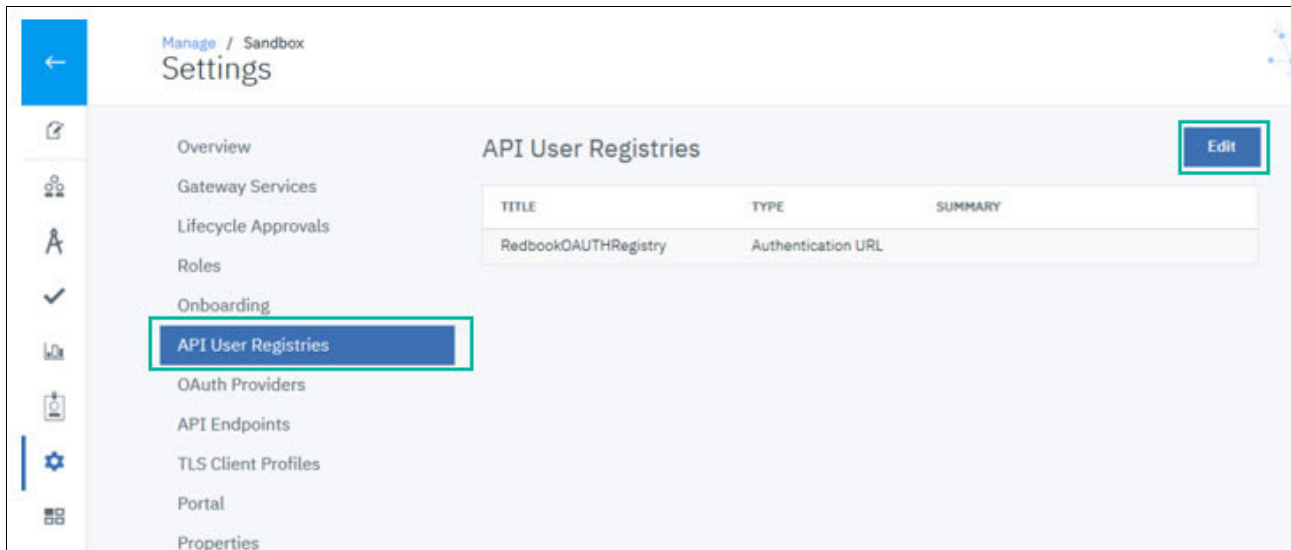


Figure 6-174 Adding the OAuth provider to the gateway 3

4. Check the **RedbookOAUTHProvider** and click **Save**. See Figure 6-175.

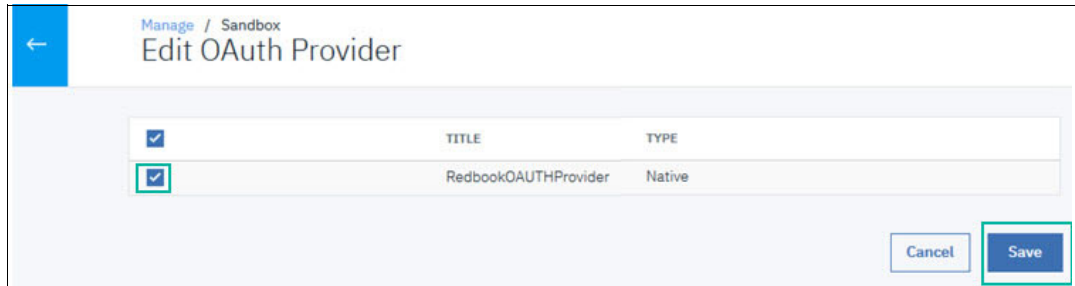


Figure 6-175 Adding the OAuth provider to the gateway 4

Adding the OAuth definition to the API

Perform the following steps:

1. From the left side menu, click on **Develop**, then choose the API you want to secure. See Figure 6-176.

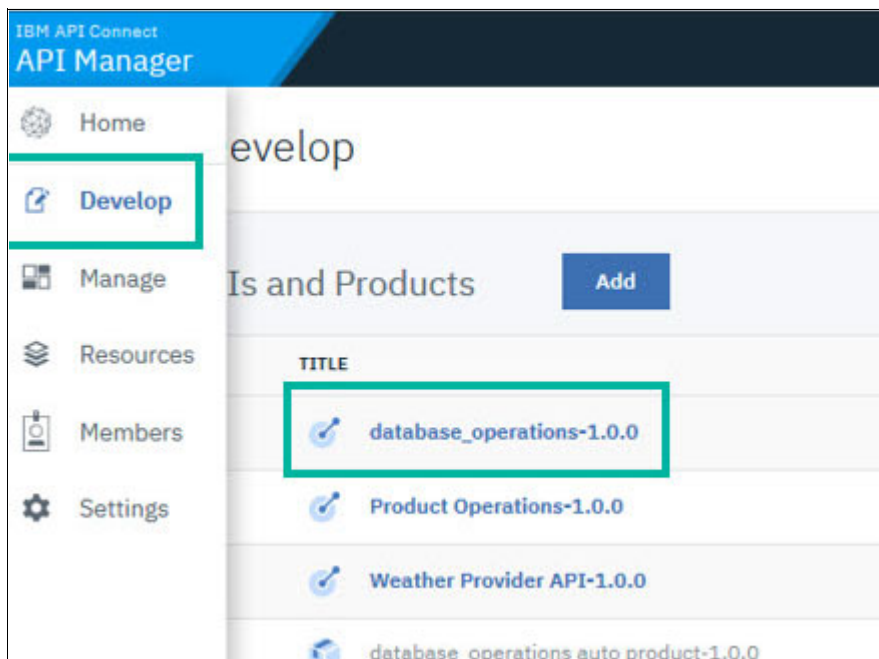


Figure 6-176 Adding the OAuth provider to the gateway 1

2. Click **Security Definitions** then click **Add**. See Figure 6-177.

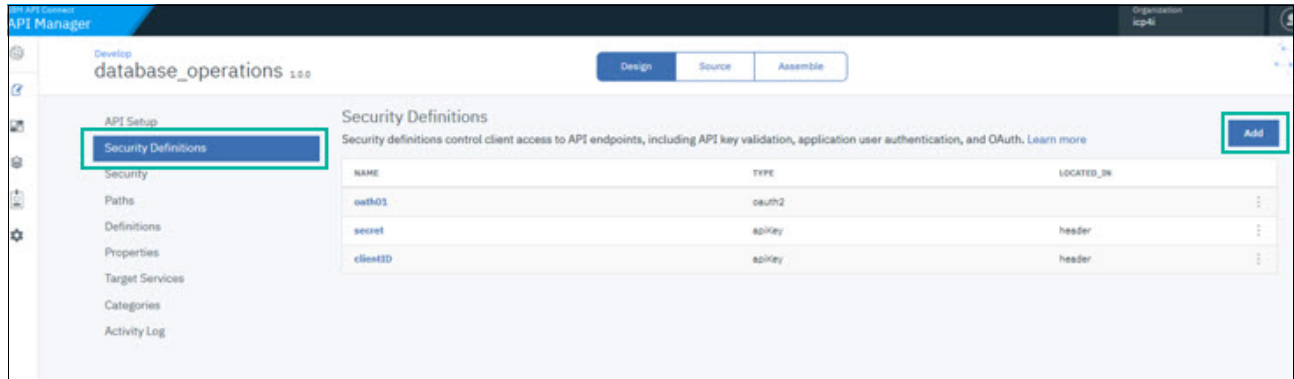


Figure 6-177 Adding the OAuth provider to the gateway 2

3. Fill in the following values:
 - Name: oauth01
 - Type: OAuth2
 - OAuth Provider: RedbookOAUTHProvider
 - Scopes: database_operations
- Then click **Save**. See Figure 6-178.

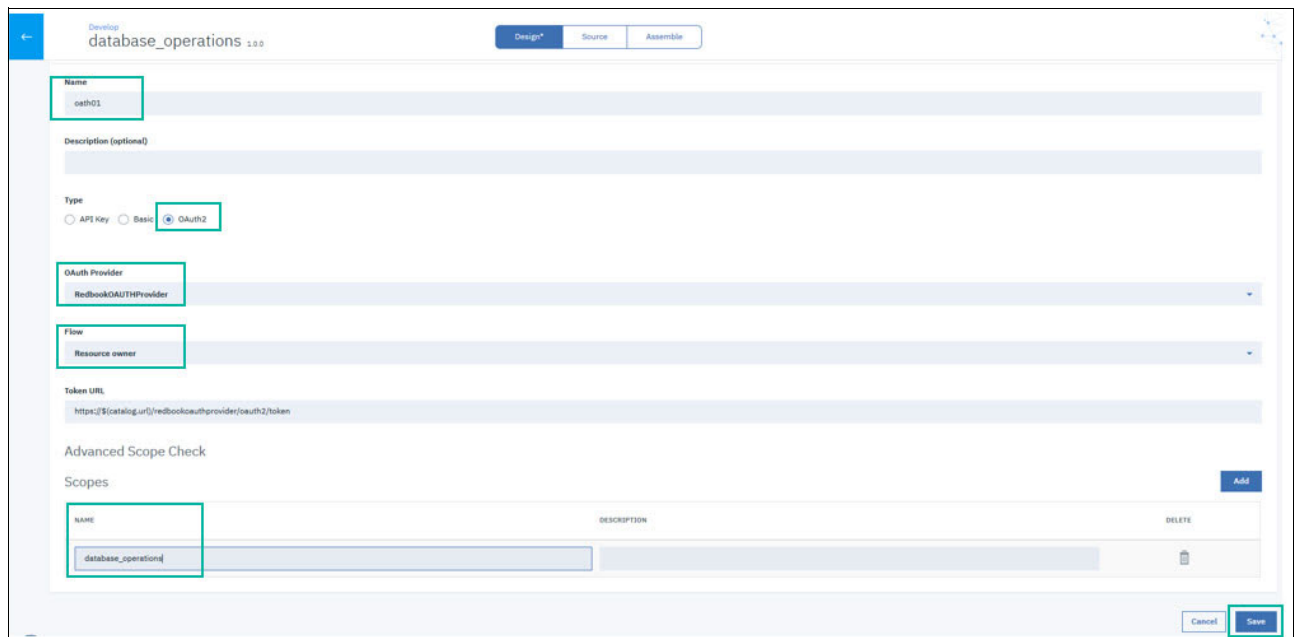


Figure 6-178 Adding the OAuth provider to the gateway 3

4. Next, click **Security** and check the **oauth01** and the scope **database_operations** then click on **Save**. See Figure 6-179.

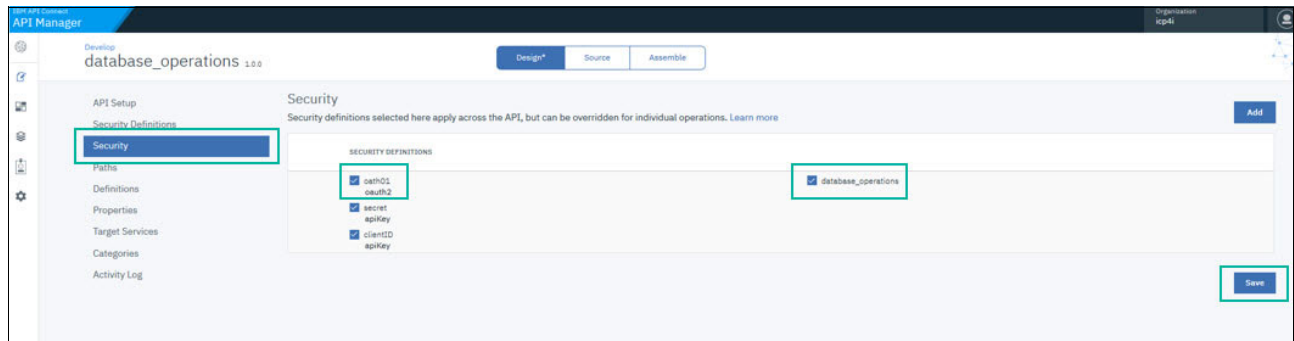


Figure 6-179 Adding the OAuth provider to the gateway 4

API discovery and testing

Now you can publish the API for testing. Follow the steps in “Publish the product” on page 204 for publishing the product.

You have three ways of testing the APIs:

- ▶ Using the Automated Application Subscriptions in the API Manager (Discussed in “Publish the product” on page 204)
- ▶ Using the Developer Portal Application Subscriptions
- ▶ Using an external REST tool like Postman client

First let’s go through the Developer Portal application subscriptions.

1. You must register in the Developer Portal. To do that, click **Create account**, then fill in your information and finally click **Sign Up**. See Figure 6-180 on page 291.

The image shows a web form for signing up on the API Developer Portal. The form is titled "Sign up" and includes the sub-header "Sign up with Sandbox Catalog User Registry". The form fields are as follows:

- Username ***: IBM
- Email address ***: User@IBM.com
- First Name ***: IBM
- Last Name ***: User
- Consumer organization ***: RedbookOrg
- Password ***: [Redacted with dots]
- Confirm password ***: [Redacted with dots]
- Image CAPTCHA ***: XPA55

Below the password fields, there is a green progress bar and the text "Password strength: Strong". A green box contains the message: "Your password meets the password policies required for this site". At the bottom of the form is a blue "Sign up" button.

Figure 6-180 API Discovery and testing 1

2. Upon a successful registration, you receive a success message indicating that you will receive an email with the activation link as shown in Figure 6-181.

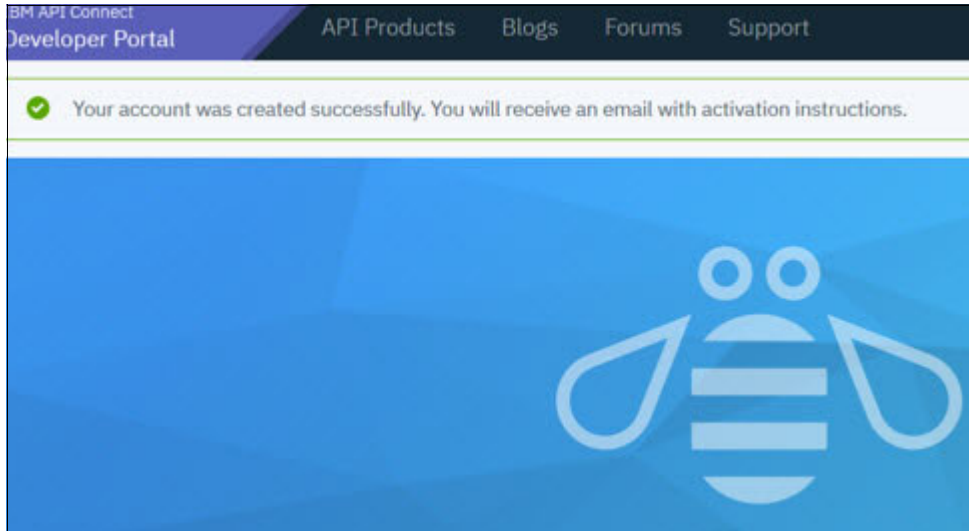


Figure 6-181 API Discovery and testing 2

3. Click the link in the received email. See Figure 6-182.

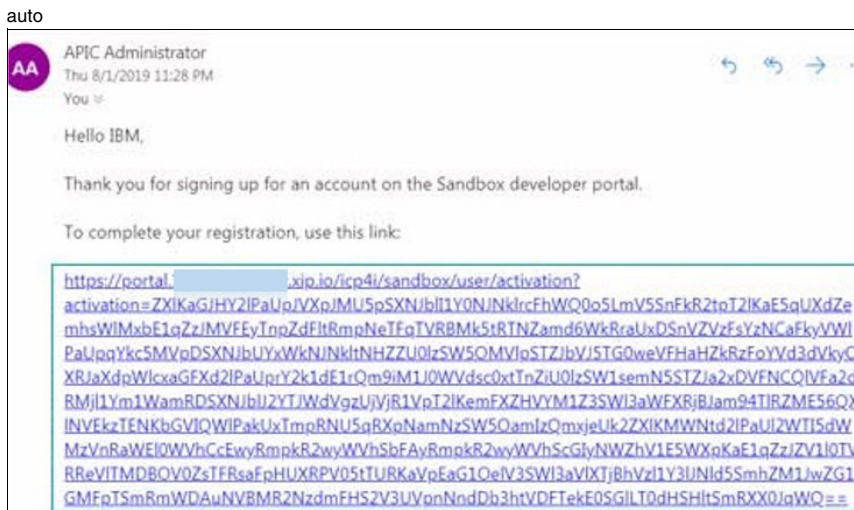


Figure 6-182 API Discovery and testing 3

Now you should see the message indicating that your account has been activated. See Figure 6-183.

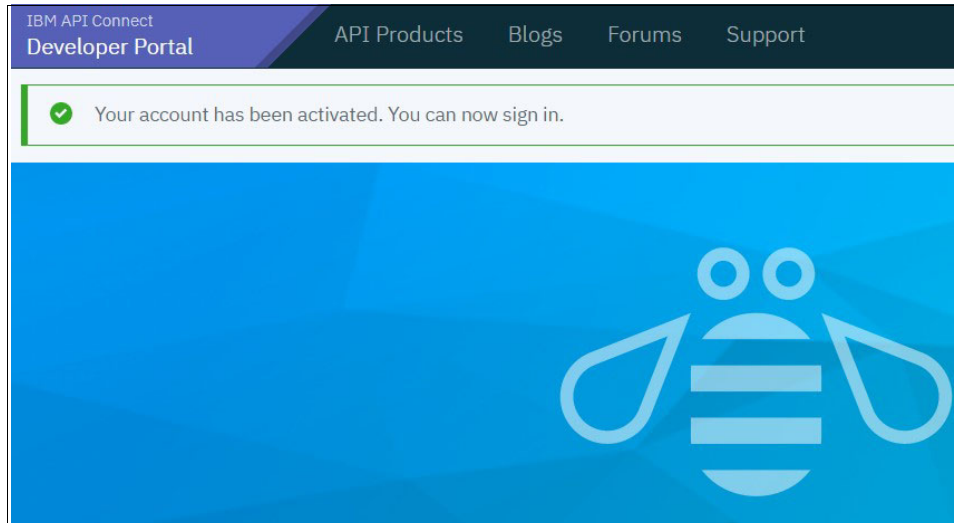


Figure 6-183 API Discovery and testing 4

4. After activating your account click **Sign in**. See Figure 6-184.

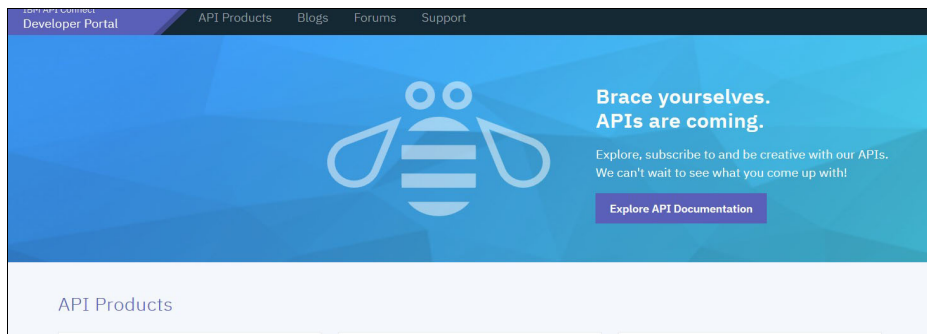


Figure 6-184 API Discovery and testing 5

5. Provide a username and password as shown in Figure 6-185.

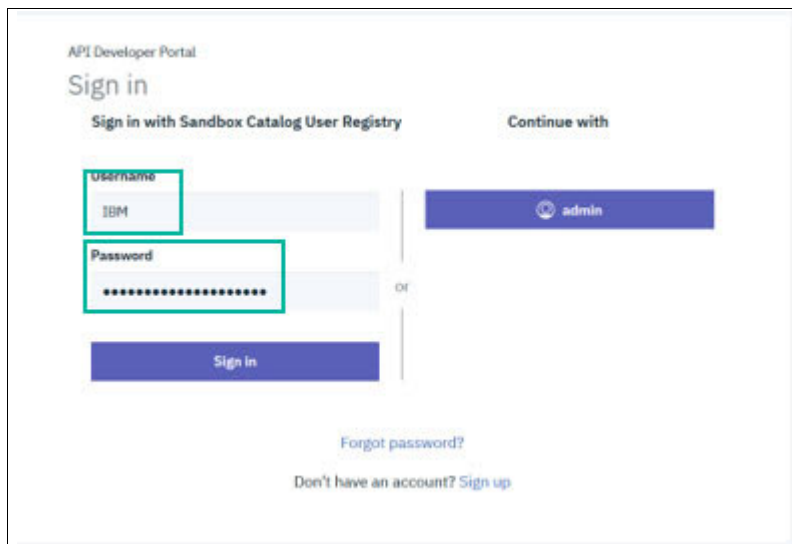


Figure 6-185 API Discovery and testing 6

6. Now you can see that you are logged in to the created organization during the registration process. See Figure 6-186.

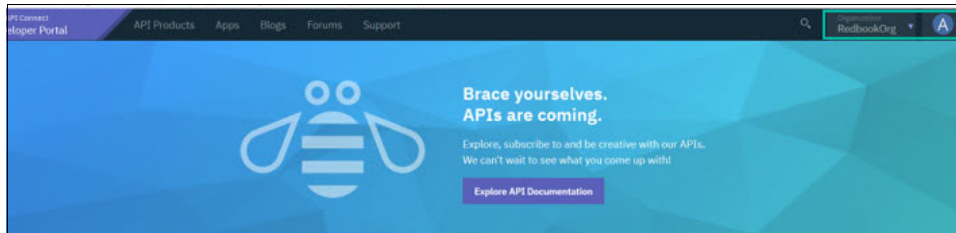


Figure 6-186 API Discovery and testing 7

7. Click **Apps** then click **Create new app**. See Figure 6-187.



Figure 6-187 API Discovery and testing 8

8. Now fill in as shown here:

- Title: RedbookTestApp
- Description: IBM Redbooks Testing Application
- Application OAuth Redirect URL(s): <http://www.oauth.com/redirect>

Then click **Submit**. See Figure 6-188.

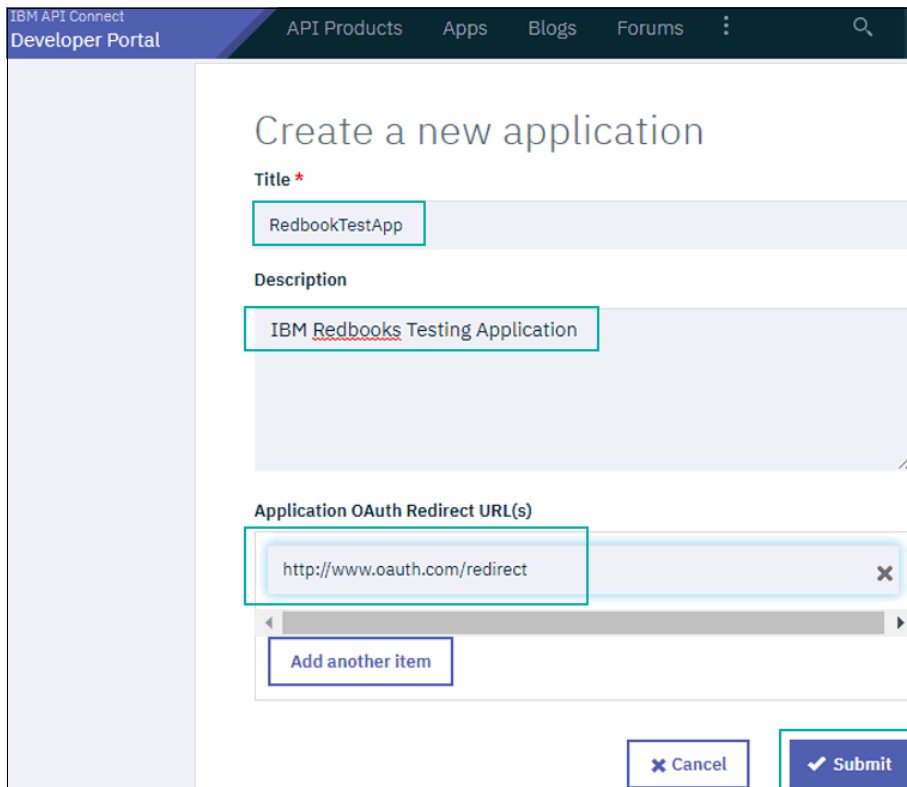


Figure 6-188 API Discovery and testing 9

9. Next page (Figure 6-189) will show the Key and Secret. Note that the secret is viewed *only once*. Therefore, you must copy it and keep it for your records.

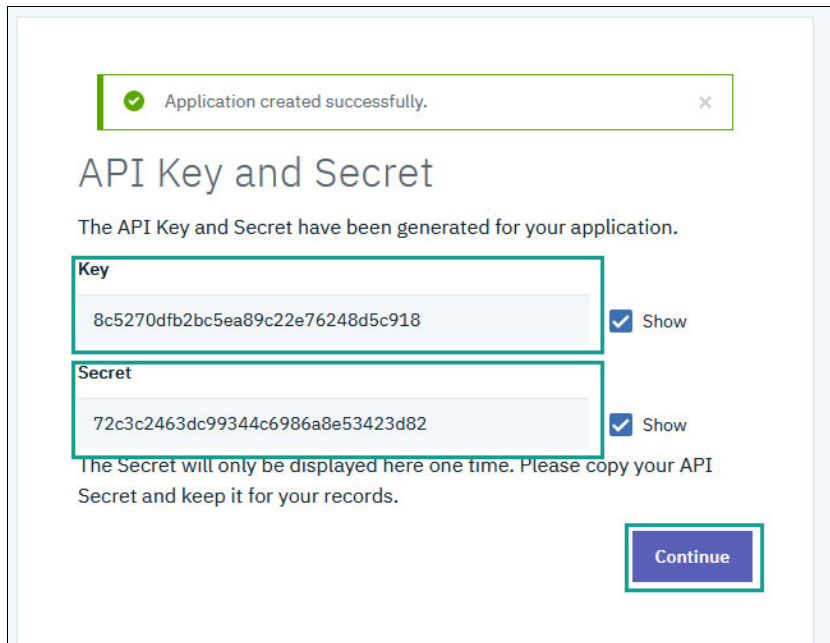


Figure 6-189 API Discovery and testing 10

10. Now choose the **RedbookTestApp** from the Apps menu. See Figure 6-190.

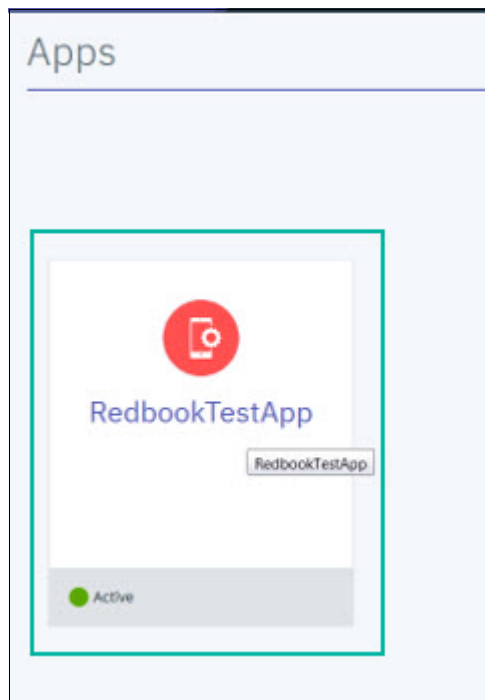


Figure 6-190 API Discovery and testing 11

11. You can add picture of your application from the application options **Upload image**. See Figure 6-191.

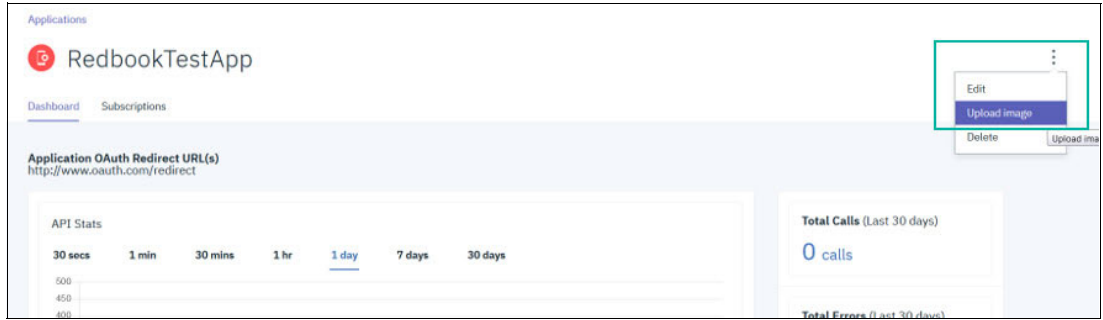


Figure 6-191 API Discovery and testing 12

12. **Browse** for the required picture then click **Submit**. See Figure 6-192 on page 296.

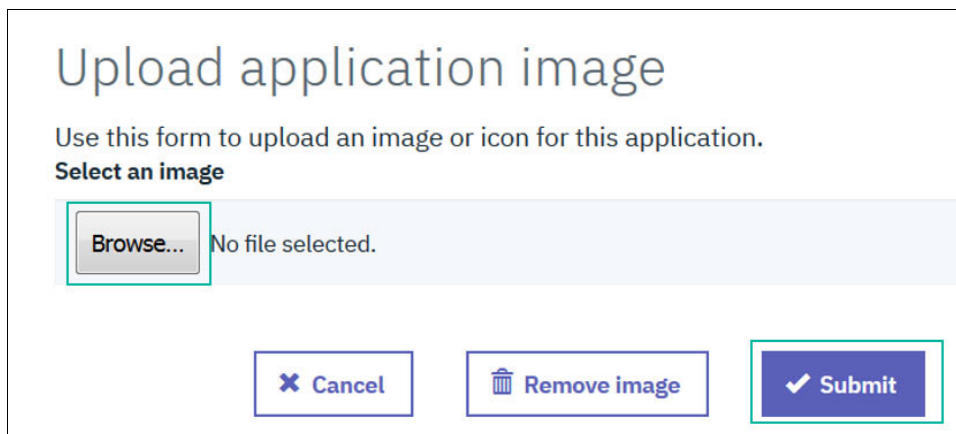


Figure 6-192 API Discovery and testing 13

Now you can see the newly uploaded application image in Figure 6-193.

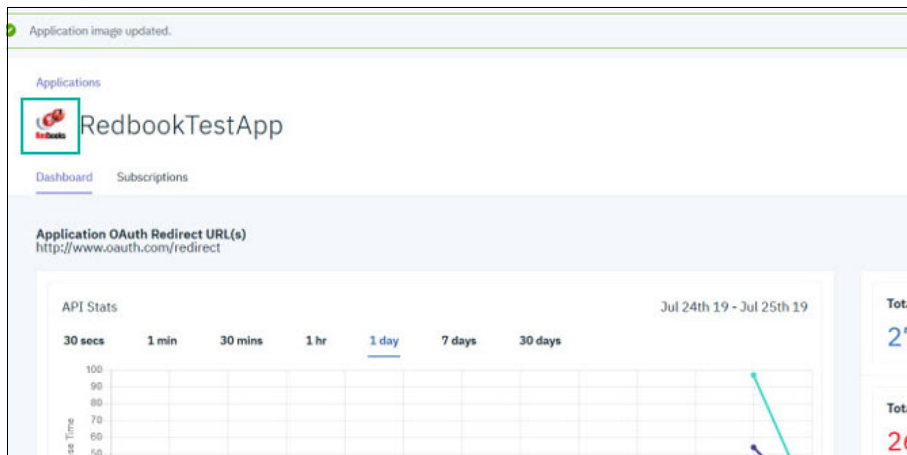


Figure 6-193 API Discovery and testing 14

Subscribing to products

Perform the following steps for subscribing to products:

1. Click **Subscriptions** then click **Why not browse the available APIs?** See Figure 6-194 on page 297.

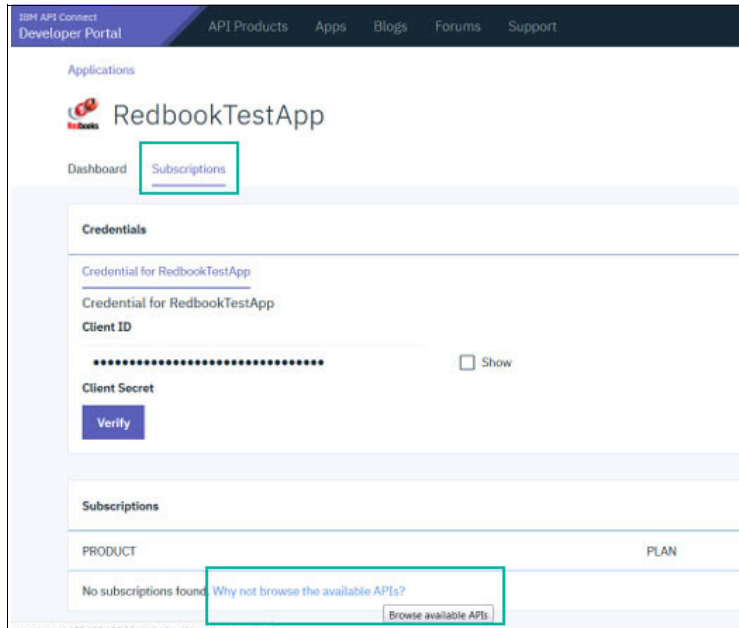


Figure 6-194 Product subscriptions 1

2. Choose the published product for **database_operations**. See Figure 6-195.

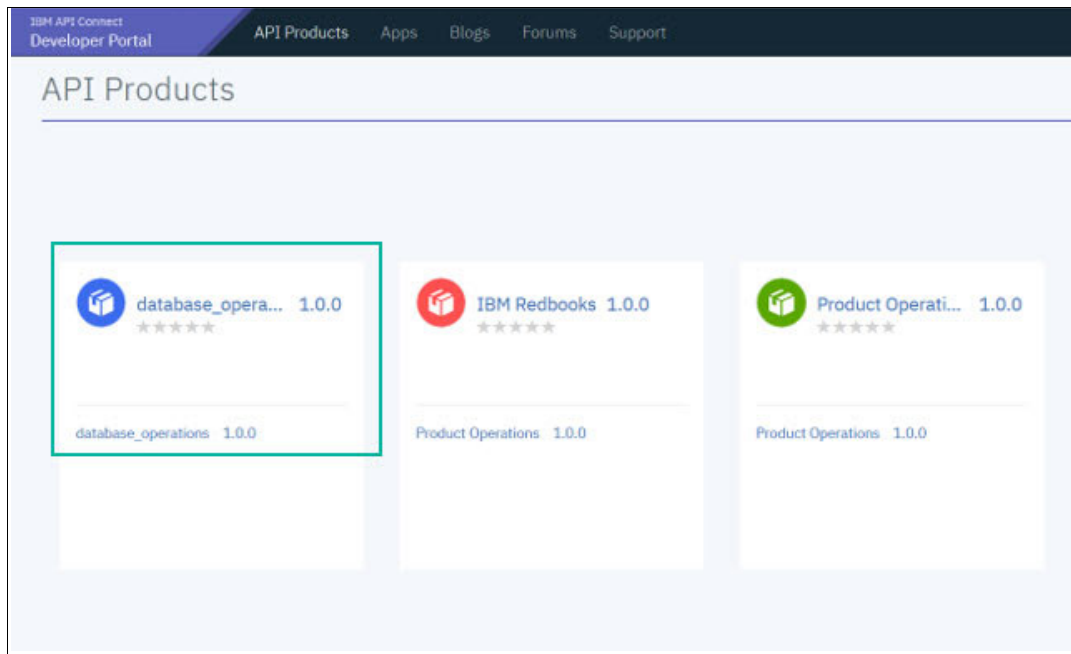


Figure 6-195 Product subscriptions 2

3. Click **Subscribe**. See Figure 6-196 on page 298.

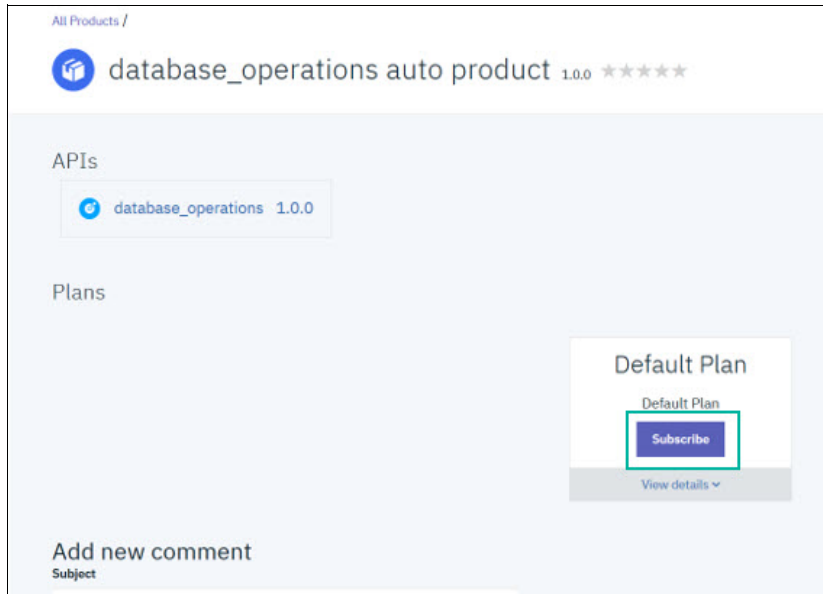


Figure 6-196 Product subscriptions 3

4. Select **RedbookTestApp**. See Figure 6-197.

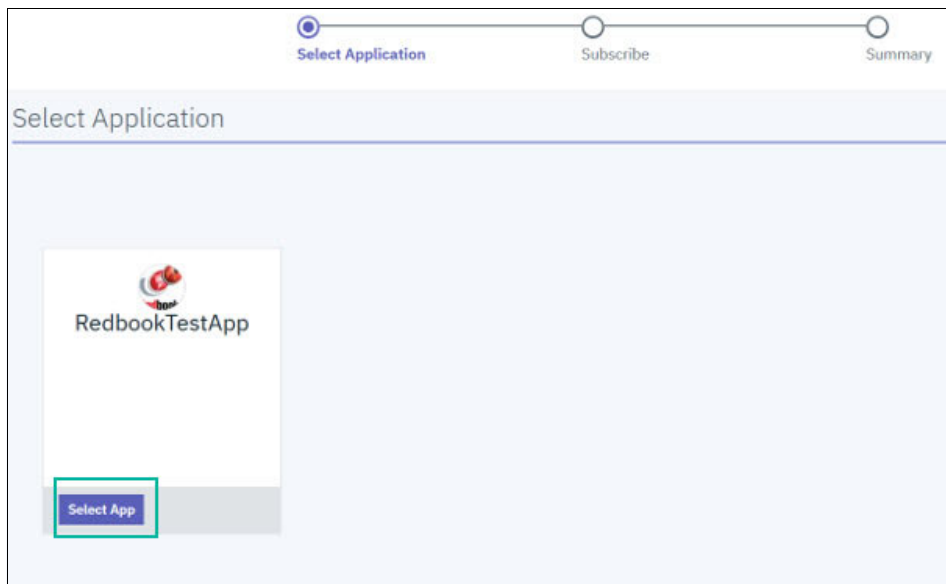


Figure 6-197 Product subscriptions 4

5. Click **Next**. See Figure 6-198 on page 299.

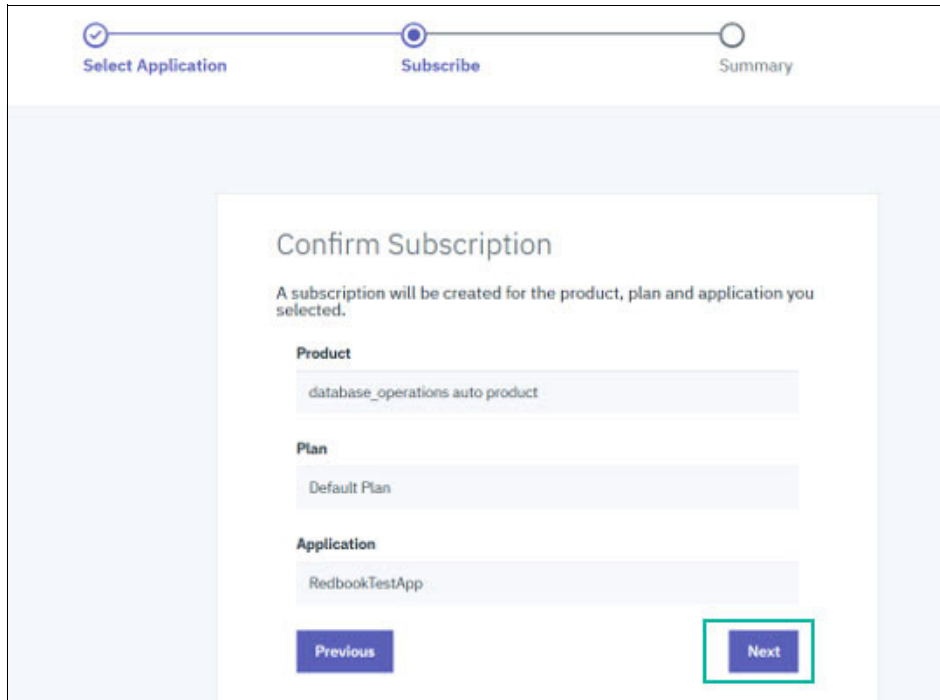


Figure 6-198 Product subscriptions 5

6. Click **Done**. See Figure 6-199.

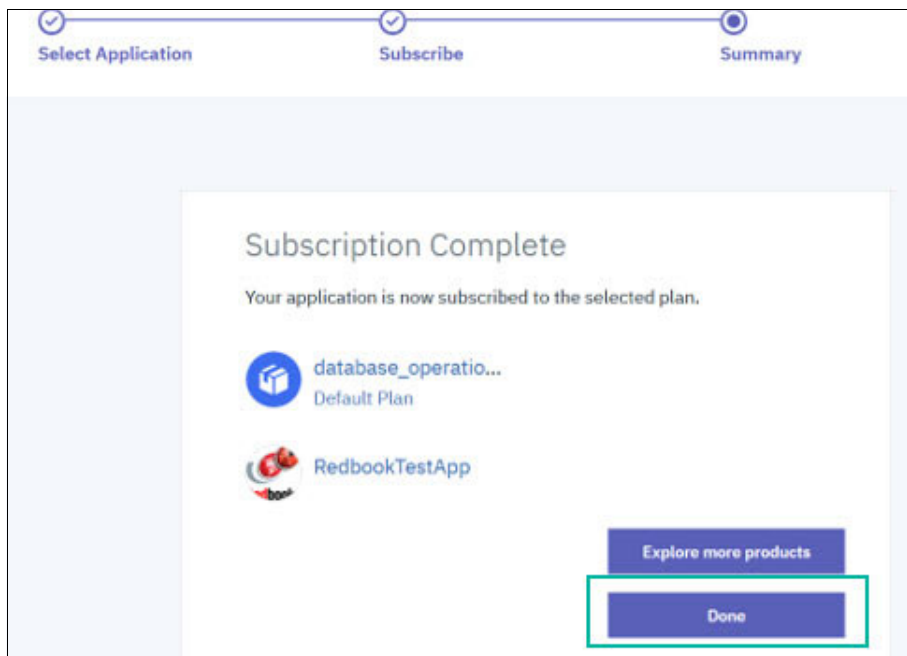


Figure 6-199 Product subscriptions 6

7. To test the api, click the product link under the product page. See Figure 6-200 on page 300.

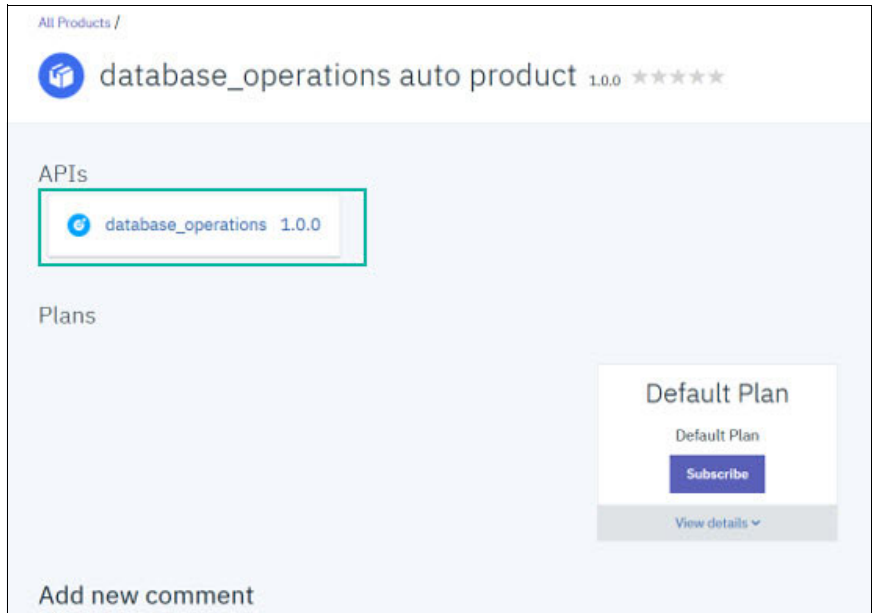


Figure 6-200 Product subscriptions 7

This will take you to the product explorer where you will be able to try the api. Under details, you can see all the api related operations and artifacts.

8. Choose **GET /products** then click **Try it**. See Figure 6-201.

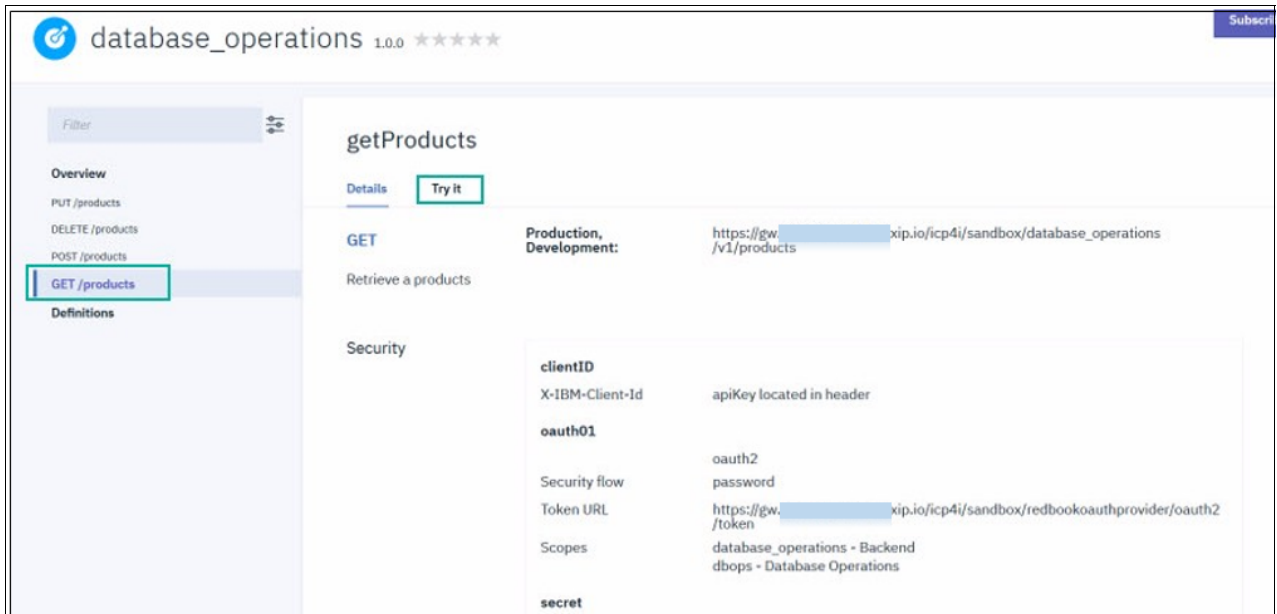


Figure 6-201 Product subscriptions 8

9. Fill in the following options:

- Client ID: choose the created application
- Client Secret: use the application client secret that was shown in the previous step
- Username (**DataPower Auth**): BookUser
- Password: BookUser

- Scopes: check database_operations
Then click **Get Token**. See Figure 6-202.

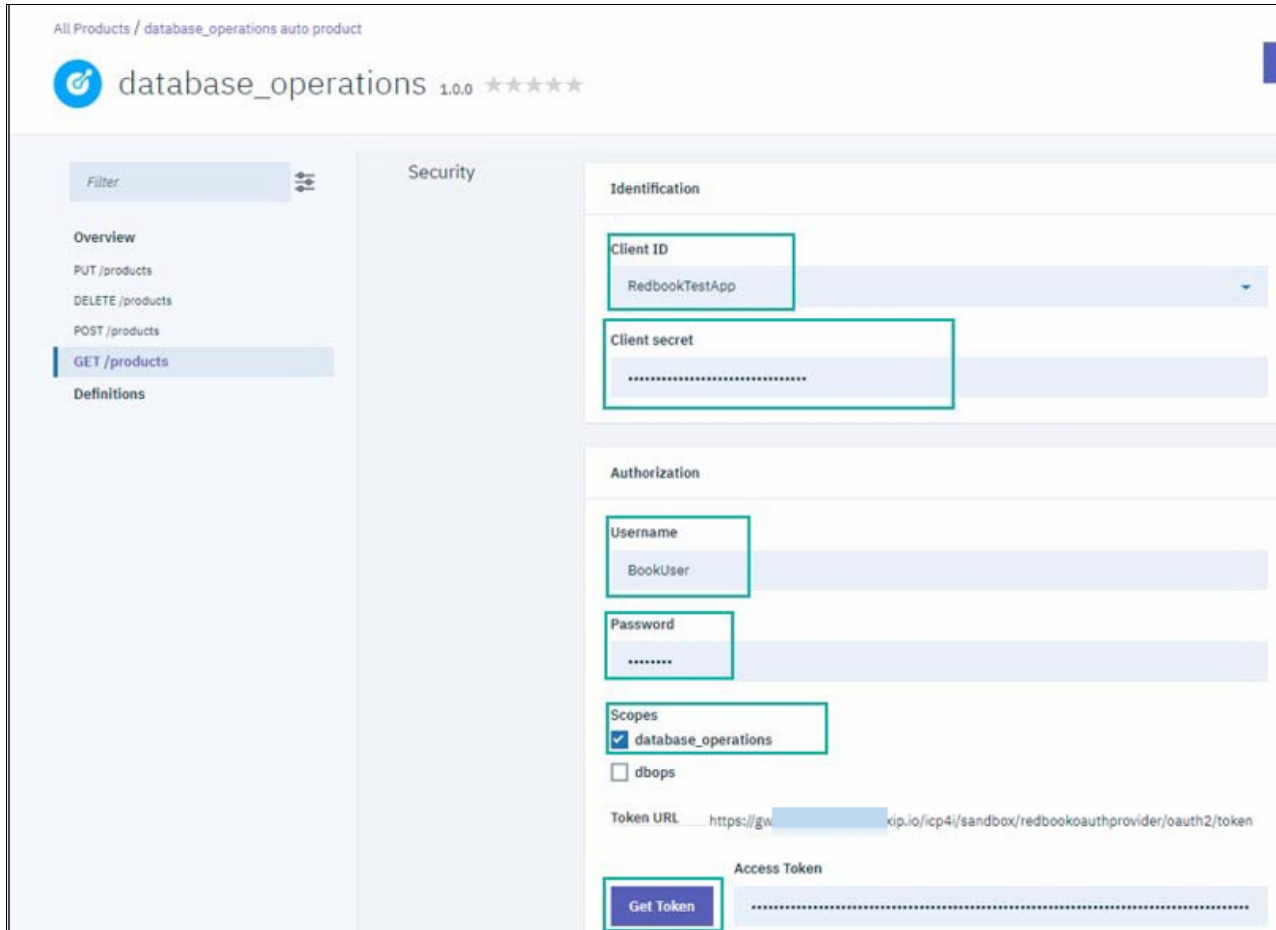


Figure 6-202 Product subscriptions 9

10. Scroll down the page then click **Send**. You receive the response **200 OK** from the back end. See Figure 6-203 on page 302.

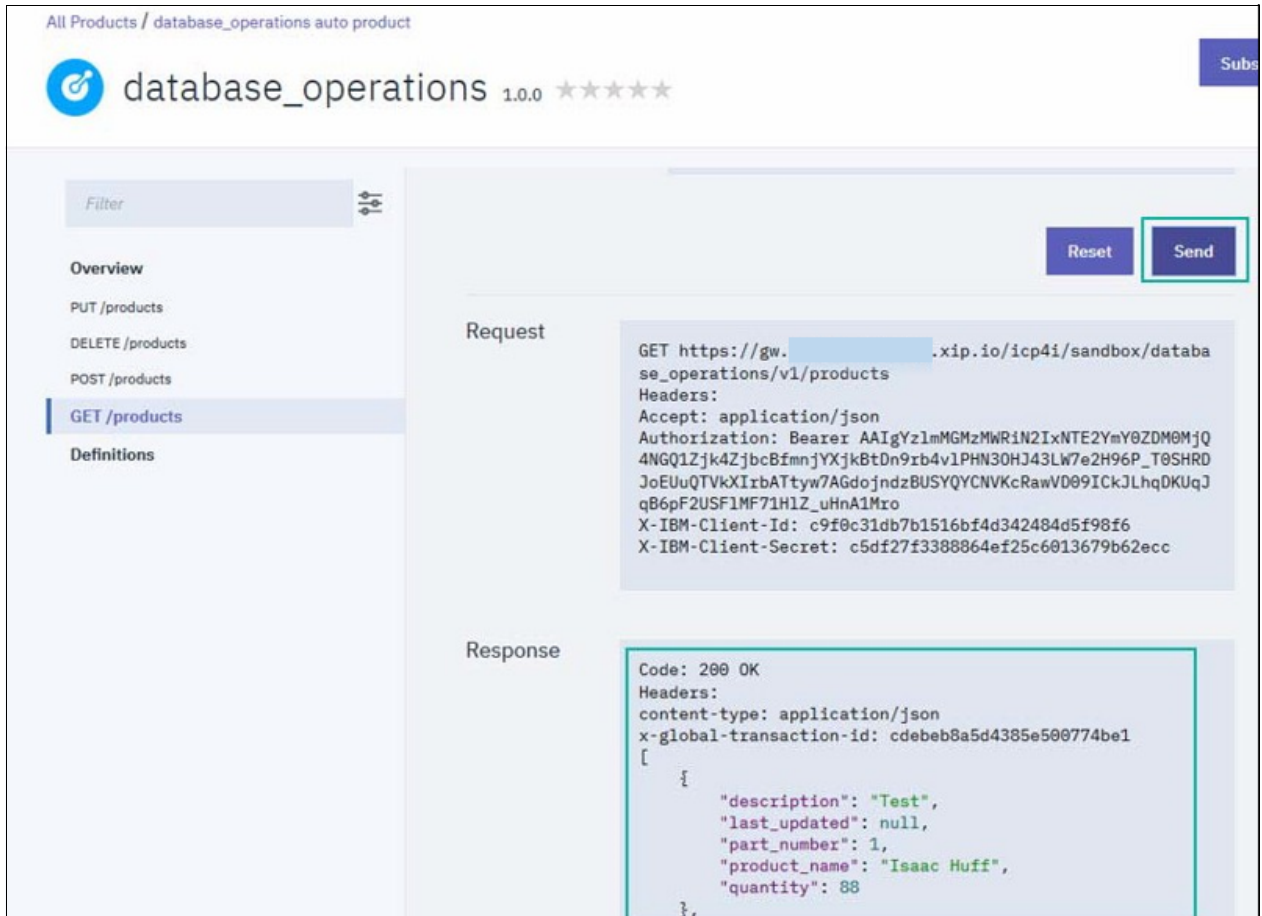


Figure 6-203 Product subscriptions 10

11. Providing a wrong credentials will result in an Invalid grant message as shown in Figure 6-204 on page 303.

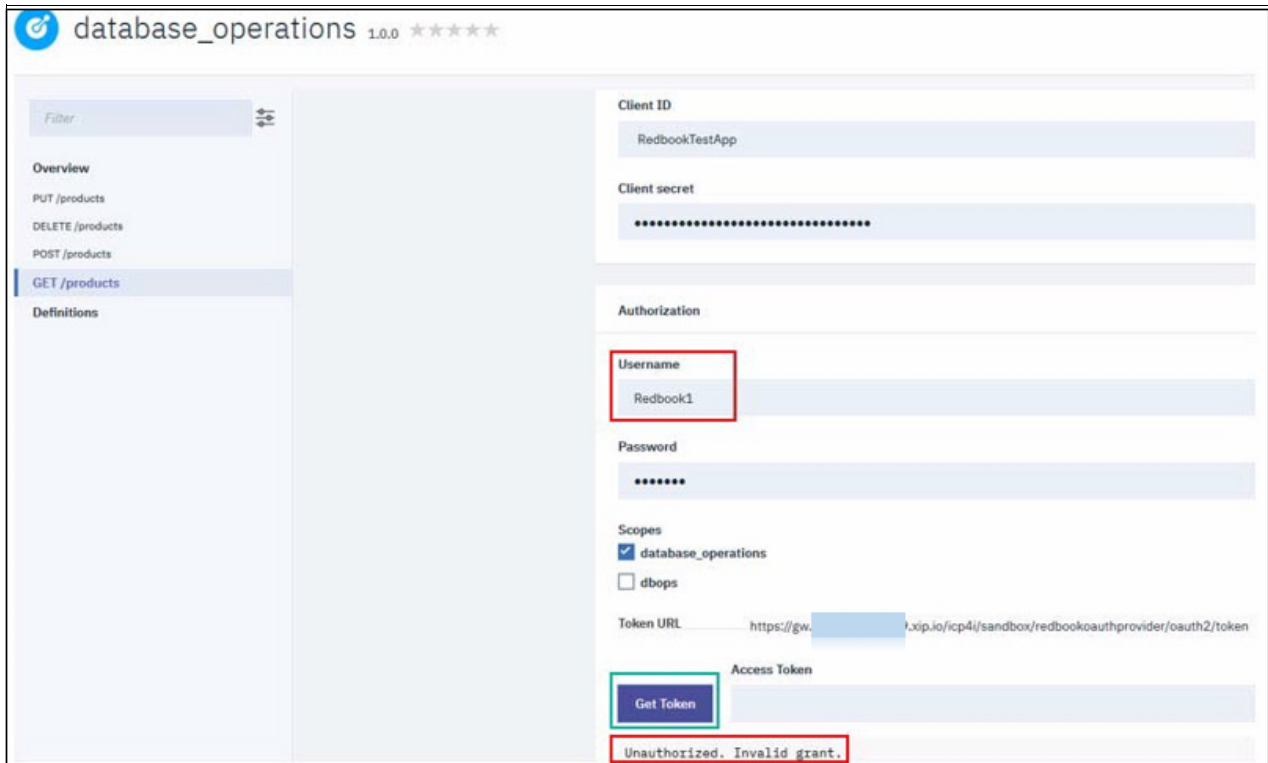


Figure 6-204 Product subscriptions 11

6.6.6 External client testing

If you wonder how to test it from Postman, first, you must obtain the token, then you can call the API.

To get the OAuth token url you can:

1. Get the Sandbox URL from **Sandbox** → **Settings** → **API Endpoints** as shown in Figure 6-205.

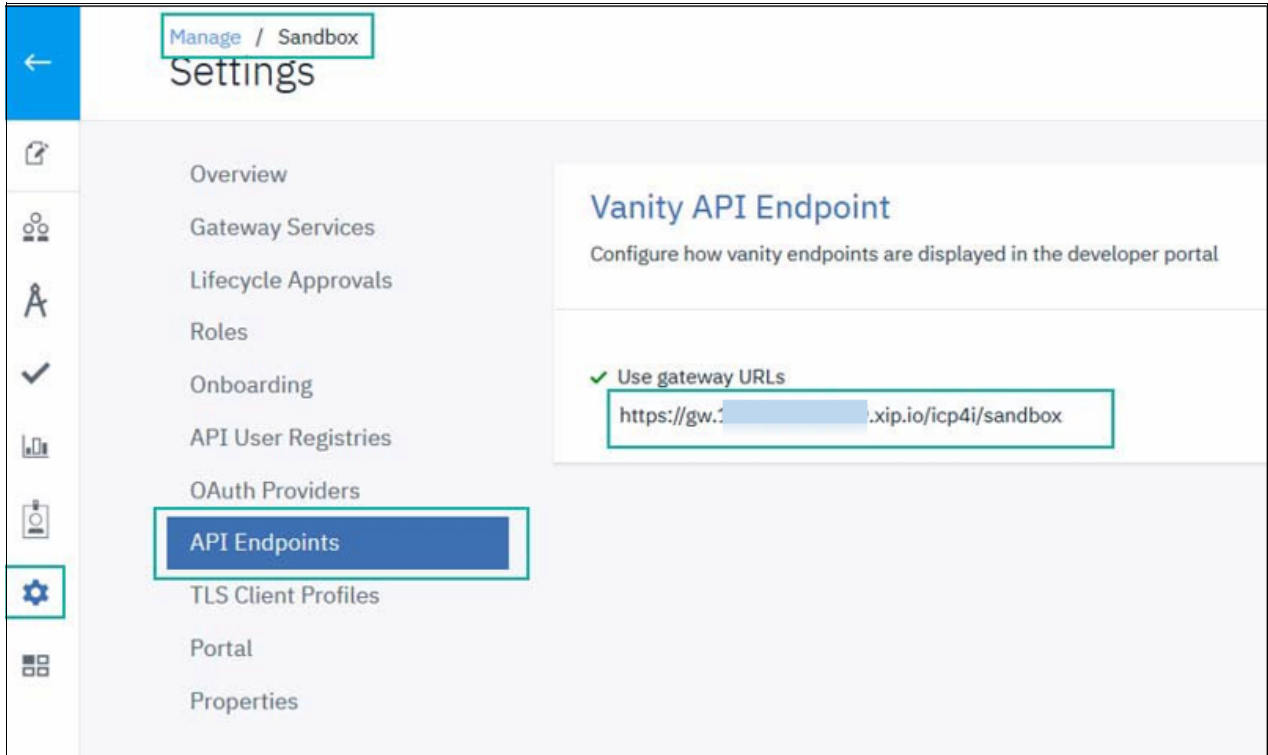


Figure 6-205 API external client testing 1

2. Get the OAuth URI base path from **Resources** → **OAuth Providers** → **RedbookOAuthProvider** → **Info**. See Figure 6-206.

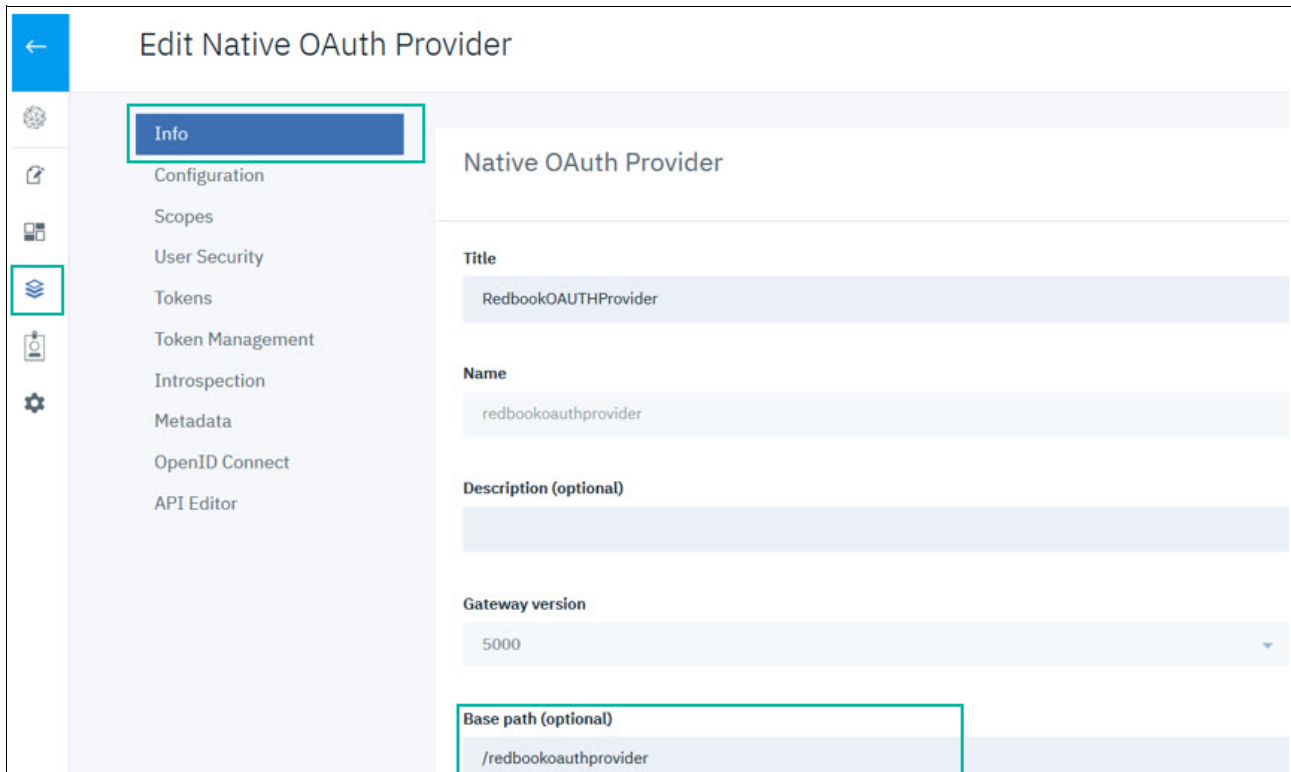


Figure 6-206 API external client testing 2

3. Get the Token path from **Resources** → **OAuth Providers** → **RedbookOAuthProvider** → **Configuration**. See Figure 6-207.

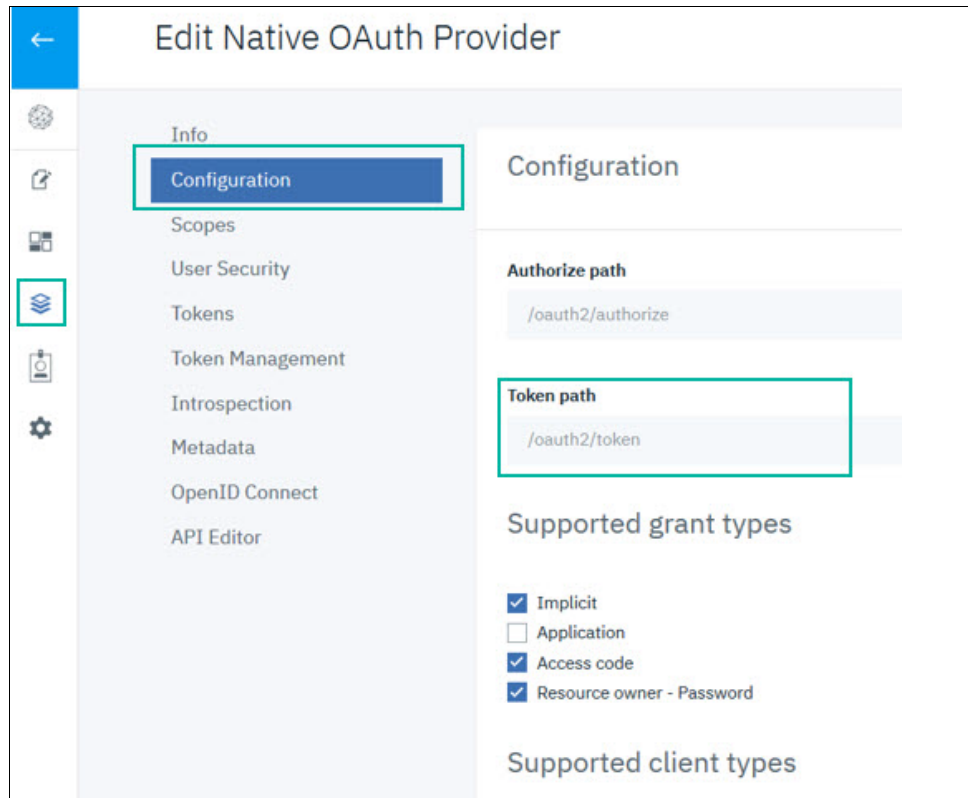


Figure 6-207 API external client testing 2

4. You must provide:
- grant_type: password
 - client_id: your subscribed application id
 - client_secret: your subscribed application secret
 - scope: the OAuth scope defined in the oauth provider and the API
 - username: (DataPower Auth URL username)
 - password: (DataPower Auth URL password)

The response should contain the token in the response body. You can find the details in Figure 6-208.

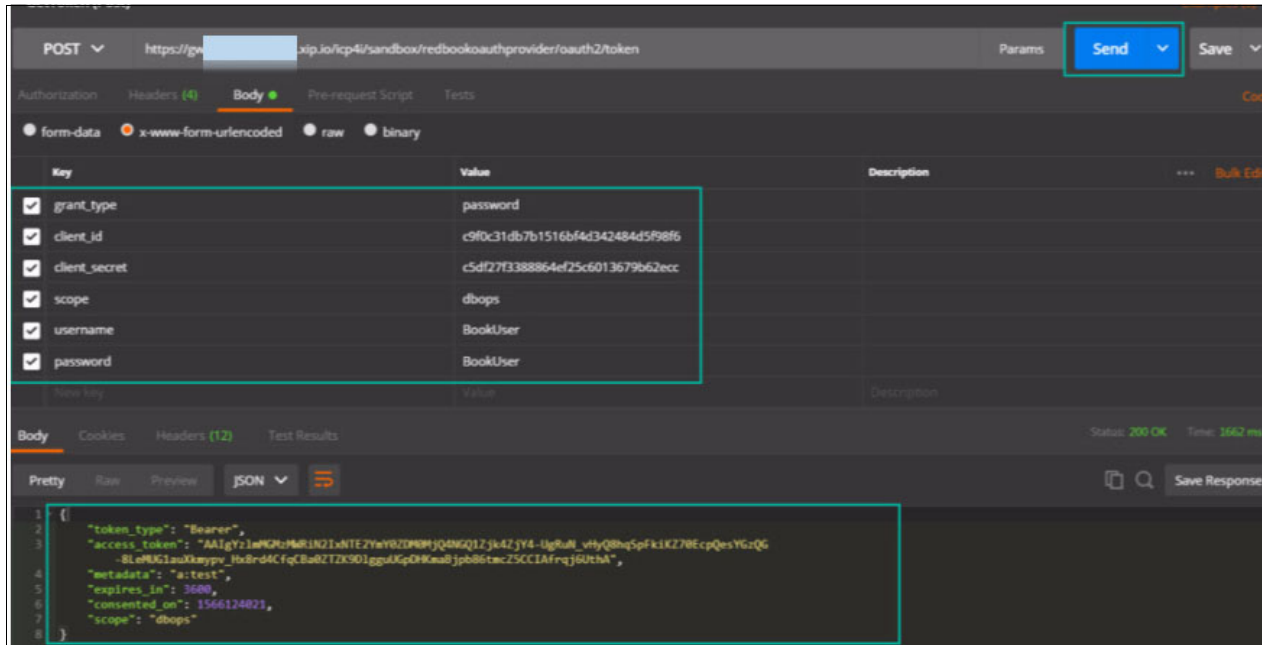


Figure 6-208 API external client testing 3

5. If you provided the wrong username or password you receive a **401 Unauthorized** “invalid_grant” response. Refer to Figure 6-209 on page 306.

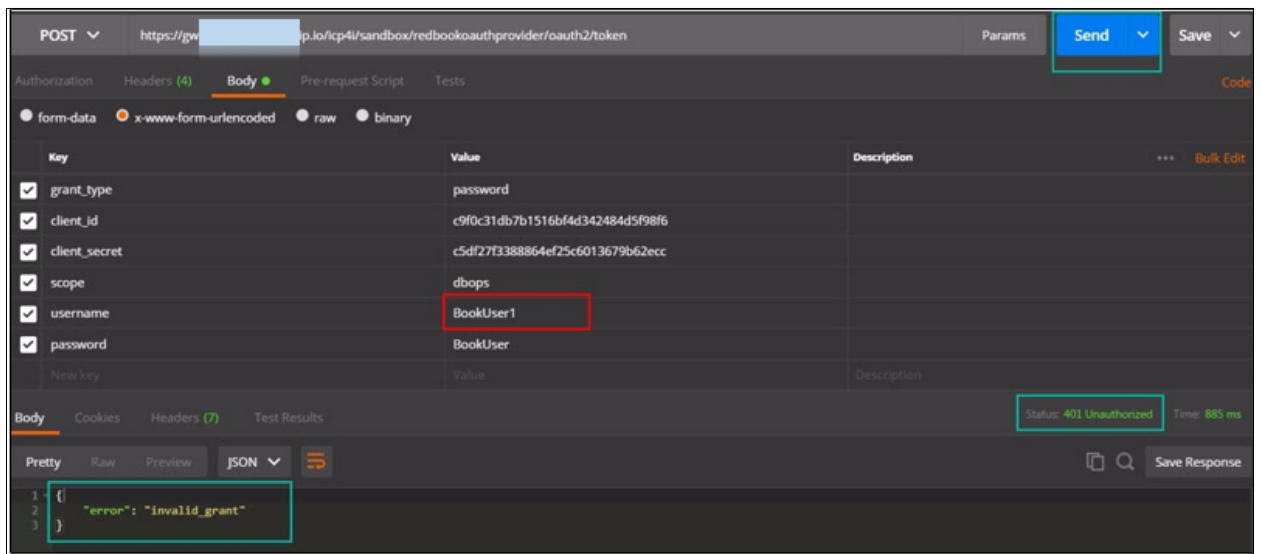


Figure 6-209 API external client testing 4

6. Next, use the API URL to test it, you must provide,
 - ▶ X-IBM-Client-Id
 - ▶ X-IBM-Client-Secret
 - ▶ Authorization (Type as Bearer)

The response is **200 OK**, and you should receive the same response that you received in the developer portal. See Figure 6-210 on page 307.

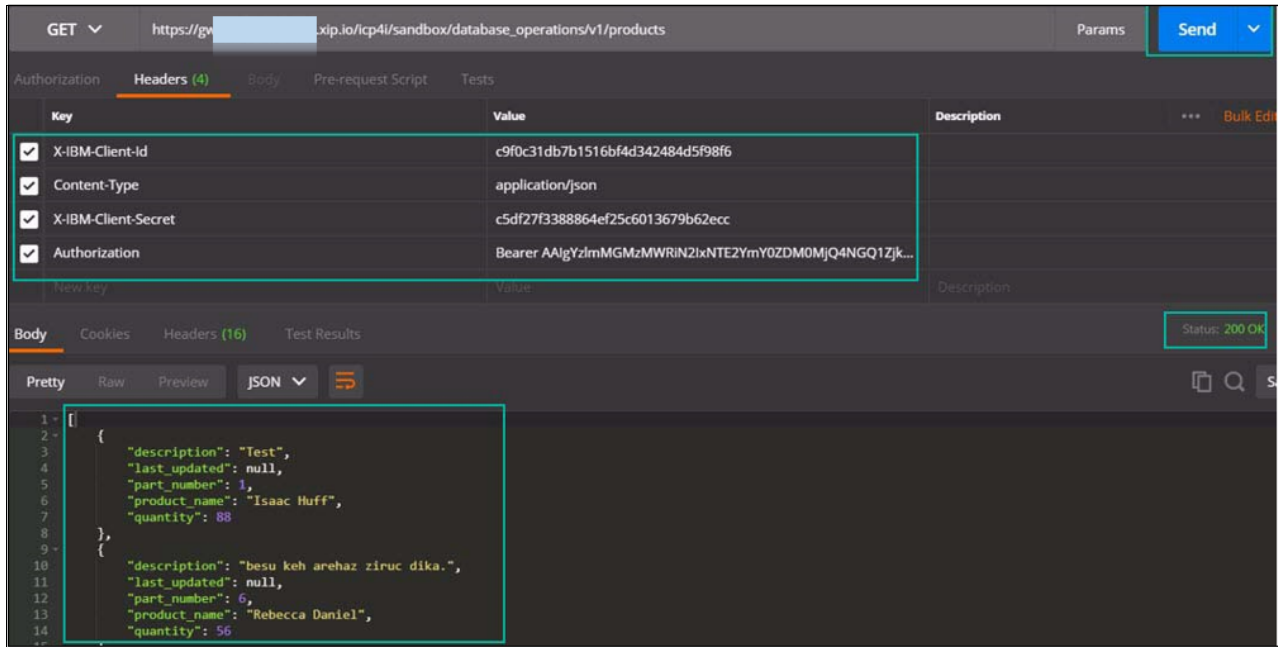


Figure 6-210 API external client testing 5

7. Providing wrong credentials will result in **401 Unauthorized** response. See Figure 6-211.

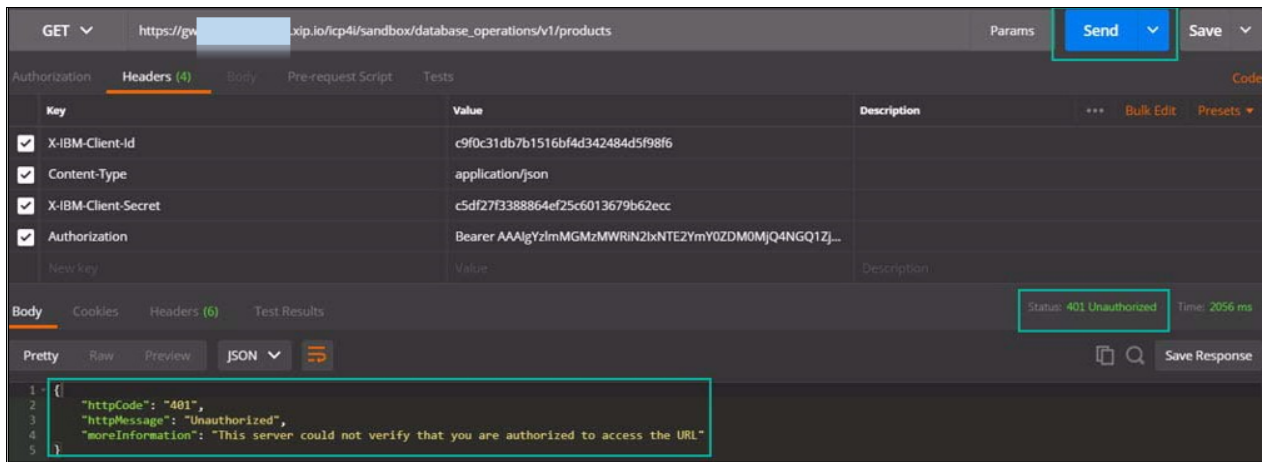


Figure 6-211 API external client testing 6

Creating the DataPower Auth URL *Optional* (Step-by-Step)

To simplify the tutorial, and to show some of the IBM DataPower capabilities, we are going to create an Auth URL. This URL could be changed to any public Auth URL.

You can use an AAA information file in the following processing phases of an AAA policy: authentication, credential mapping, resource mapping, and authorization. For more information see the example XML file here:

<https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/AAAInfoFile.xml>.

Perform the following to create the DataPower Auth URL:

1. Open the DataPower interface and go to: **Services** → **XML Firewall** → **New**. See Figure 6-212.

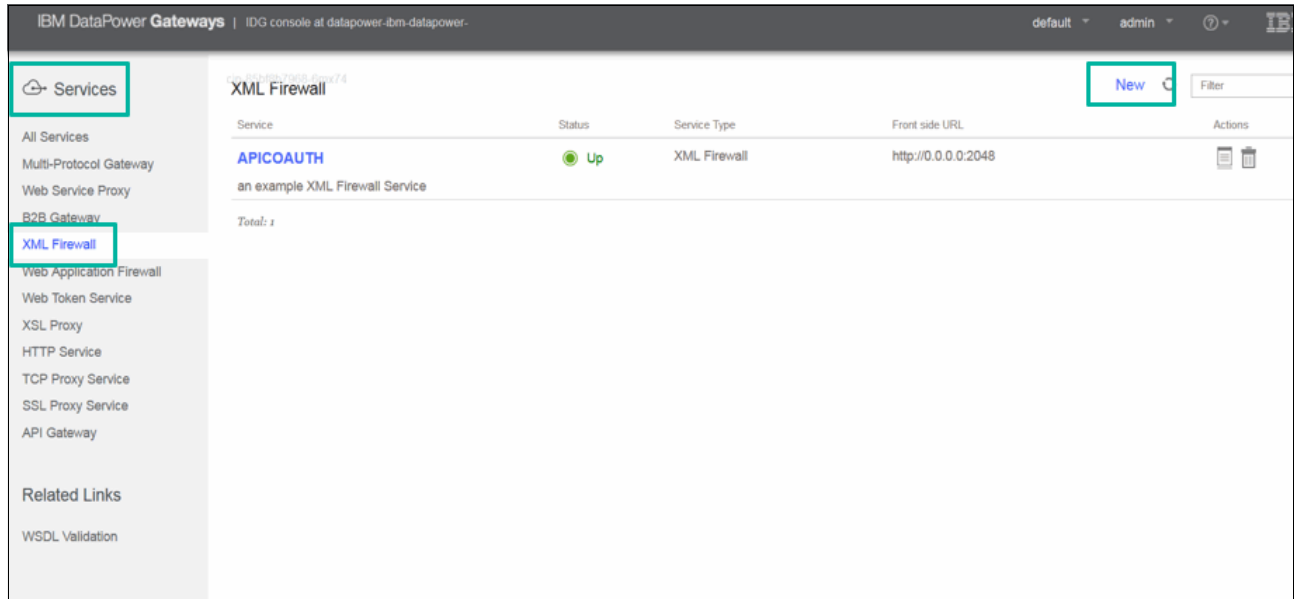


Figure 6-212 Configuring the DataPower XML firewall 1

Change the properties to:

- Firewall name: RedbookAuth
- Firewall Type: Loopback
- Request Type: JSON
- Port: You can leave it as default or change it to your preference

2. Then click on the plus sign (+) next to Processing Policy. See Figure 6-213.

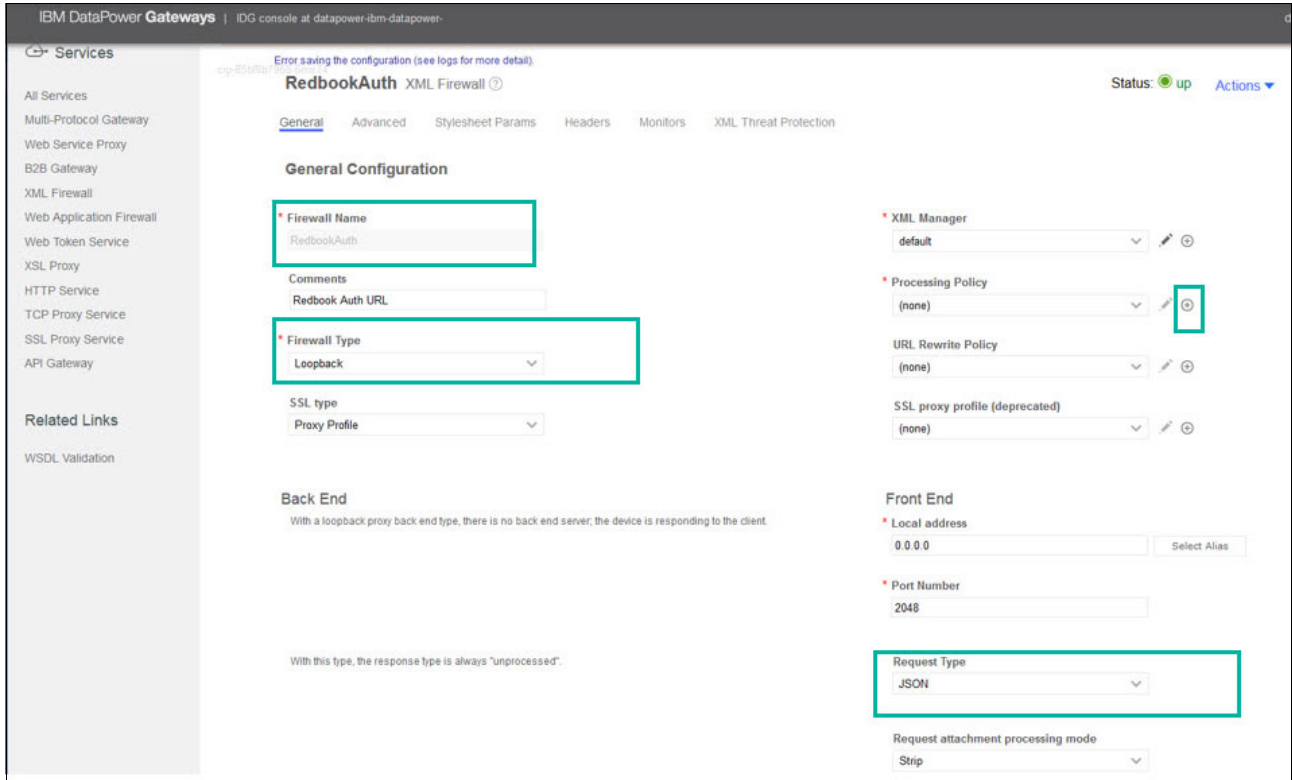


Figure 6-213 Configuring the DataPower XML firewall 2

3. Enter the policy name and choose **Client to Server** then click on **New Rule**. See Figure 6-214.

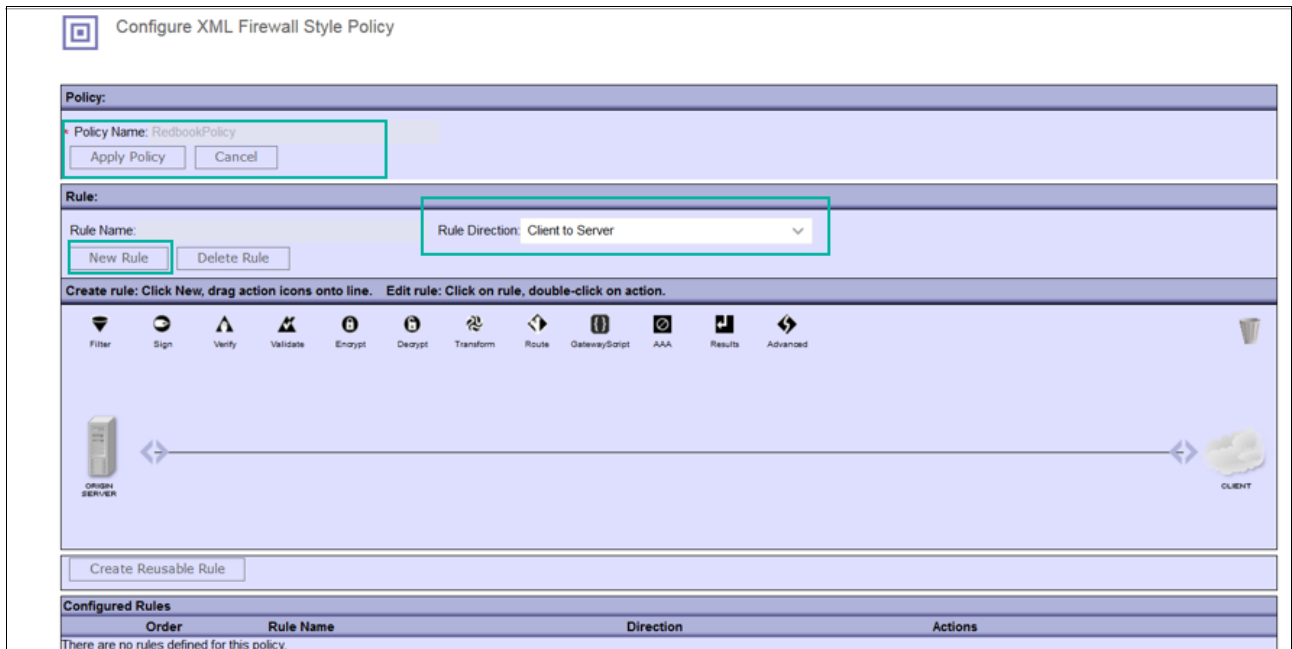


Figure 6-214 Configuring the DataPower XML firewall 3

4. Double click on highlighted (=) to configure it as shown in Figure 6-215.

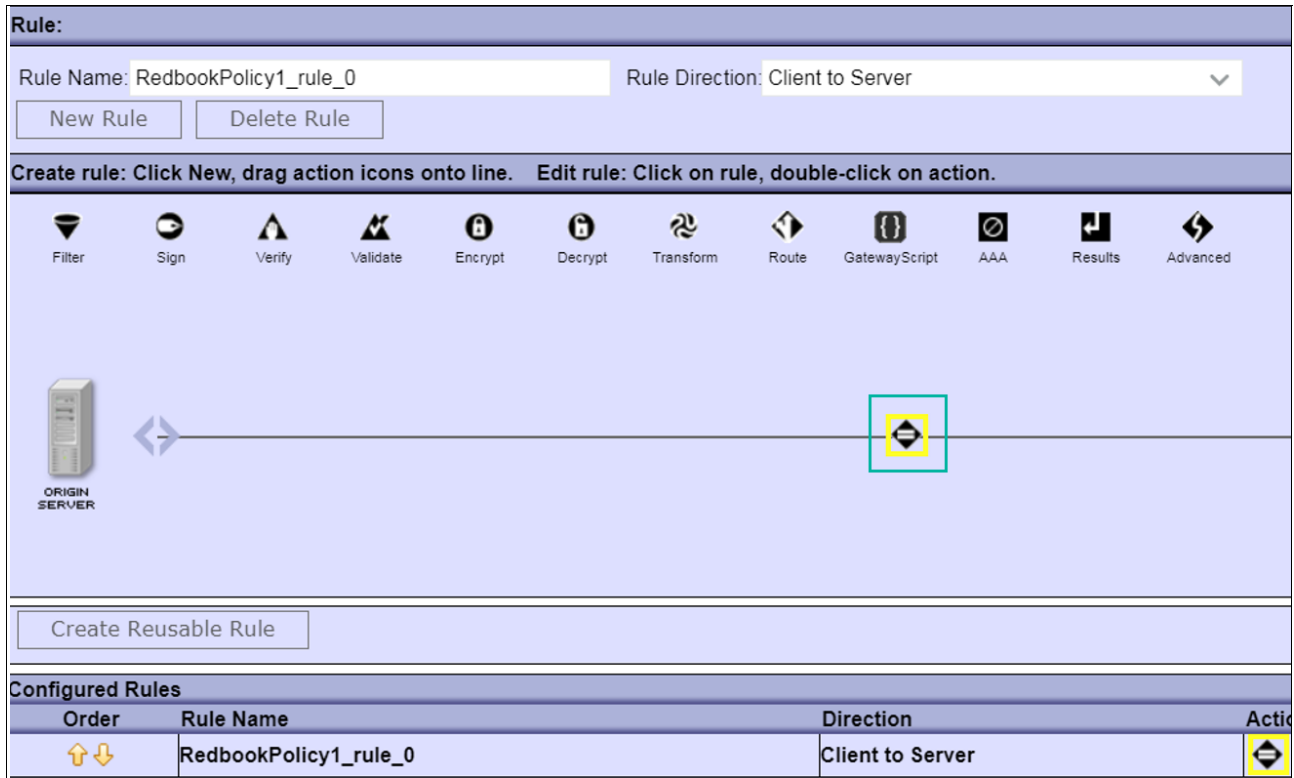


Figure 6-215 Configuring the DataPower XML firewall 3

5. Click on the (+) next to Matching Rule. See Figure 6-216.

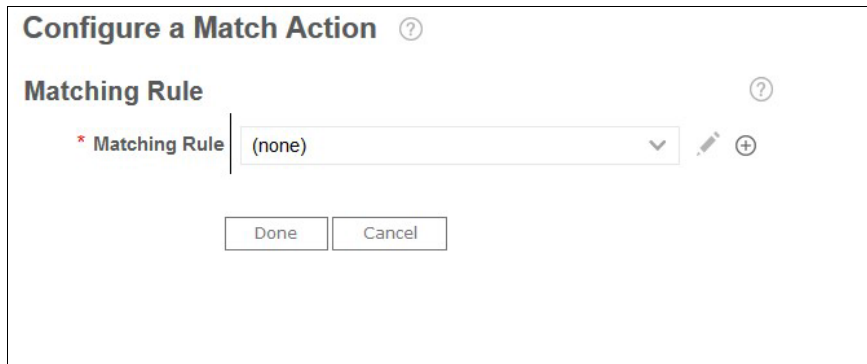


Figure 6-216 Configuring the DataPower XML firewall 5

6. Change the name to **MatchAll** then click on **Add** under Rules. See Figure 6-217 on page 311.

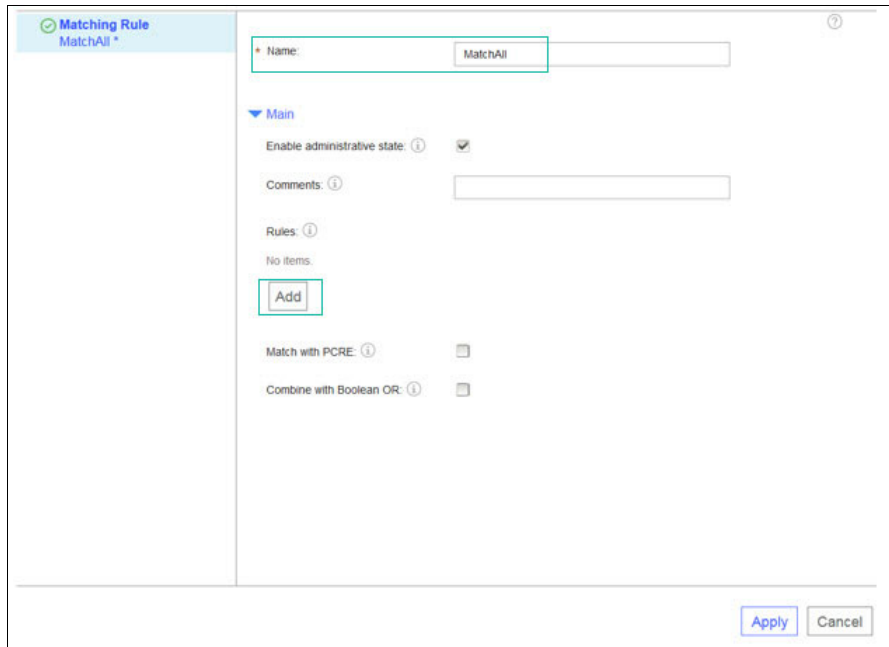


Figure 6-217 Configuring the DataPower XML firewall 6

7. Choose the Matching type to be **URL** and in the value use *****, click on **Apply** then **Done**. See Figure 6-218.

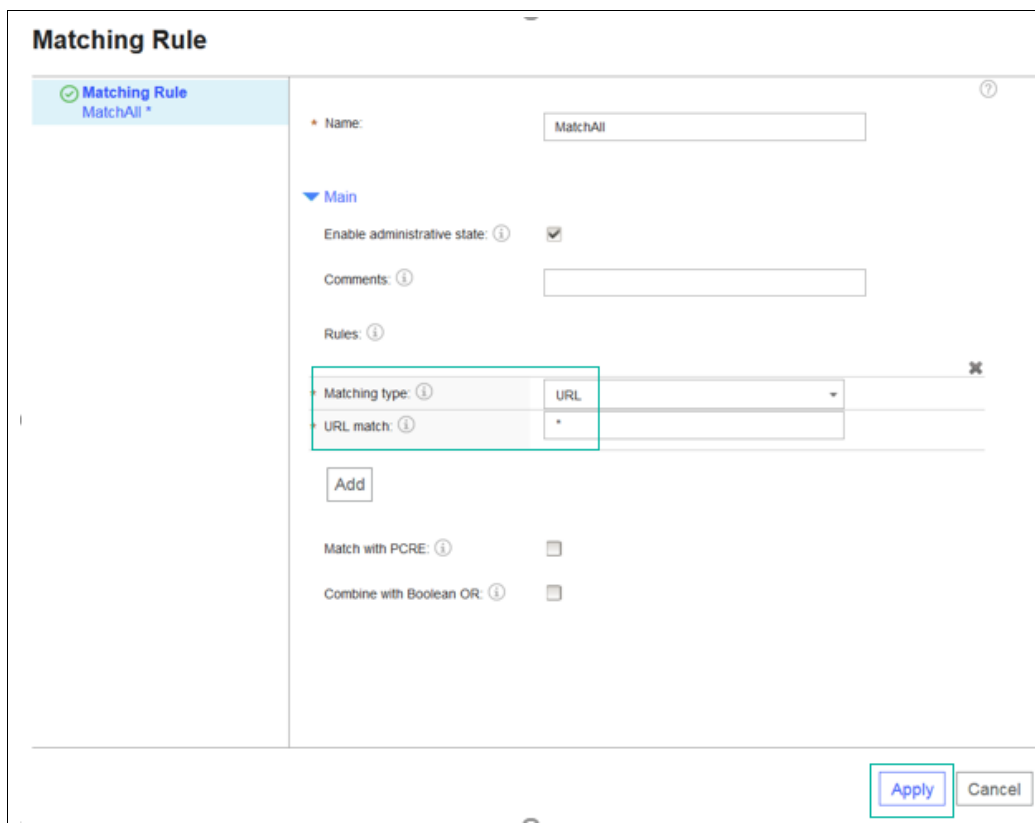


Figure 6-218 Configuring the DataPower XML firewall 7

8. Now drag the **AAA** policy to the flow and double click on it to configure it. See Figure 6-219.

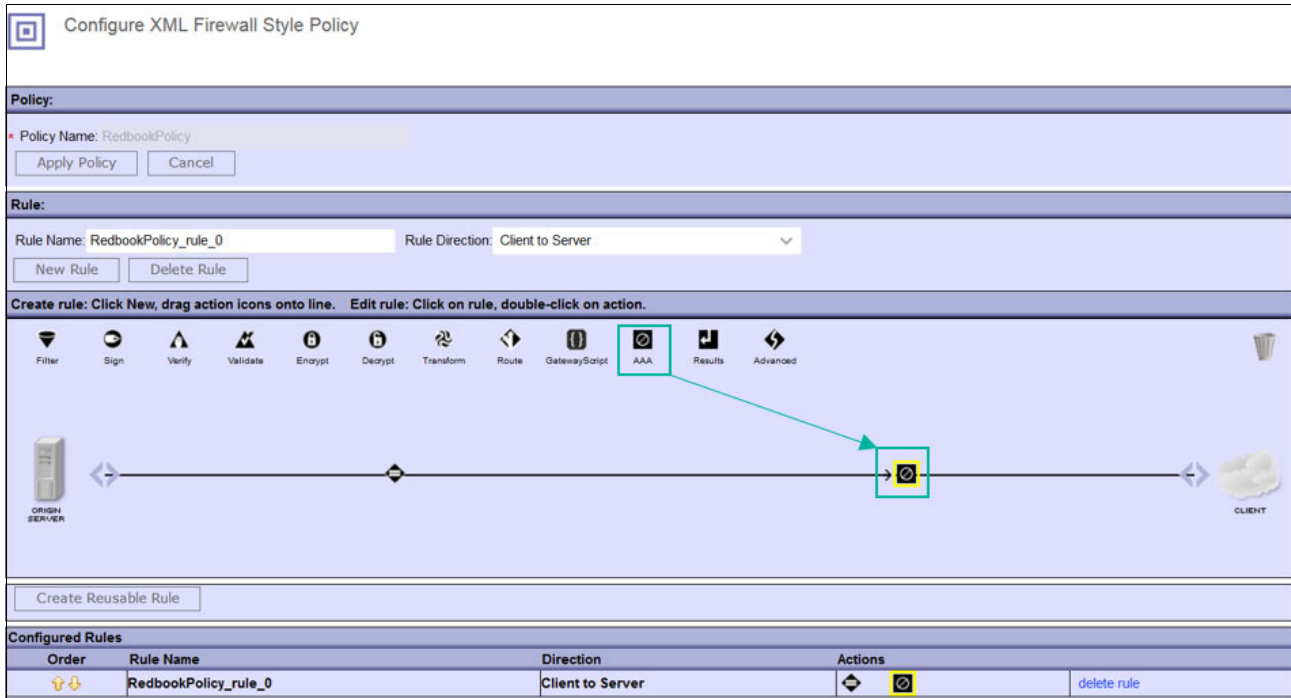


Figure 6-219 Configuring the DataPower XML firewall 8

9. Click + next to the AAA policy as shown in Figure 6-220.

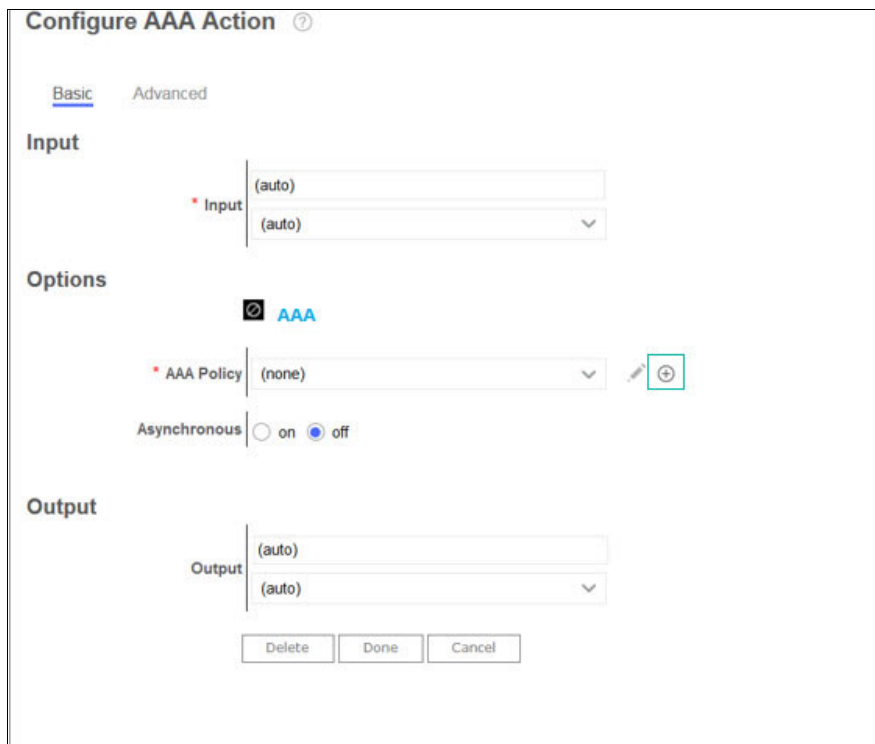


Figure 6-220 Configuring the DataPower XML firewall 9

10. Now type **RedbookPolicy** in the name and click **Create**. See Figure 6-221 on page 313.



Figure 6-221 Configuring the DataPower XML firewall 10

11. Choose **HTTP Authentication Header** and click **Next**. See Figure 6-222.

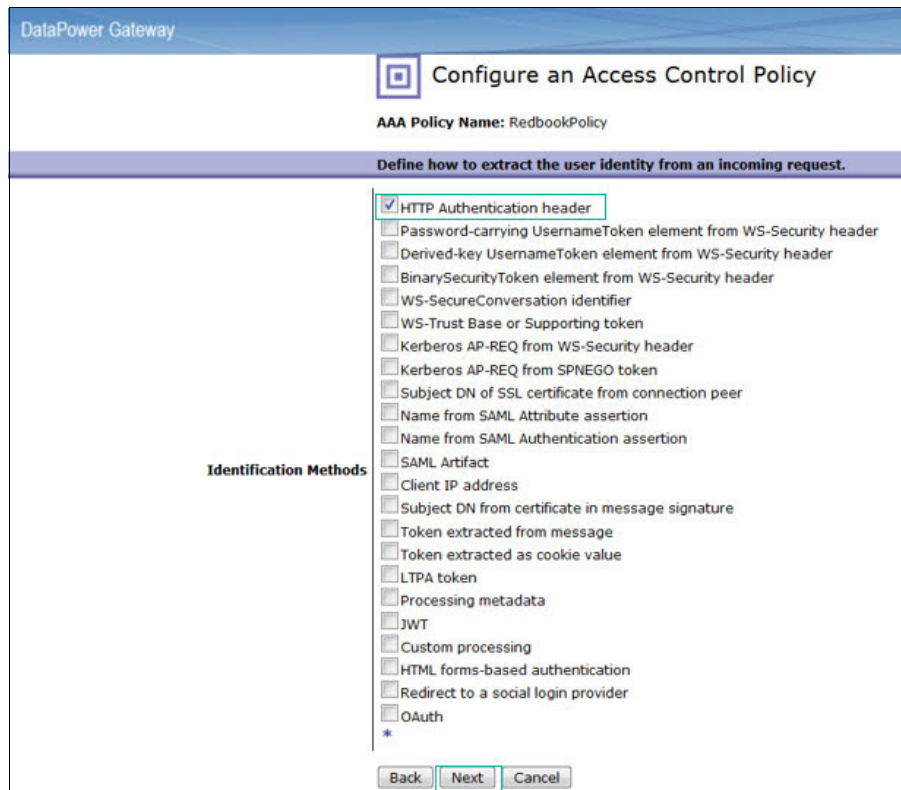


Figure 6-222 Configuring the DataPower XML firewall 11

12. Choose **Use AAA Information File** and click **Upload**. Use the sample file AAAInfoFile.xml from this web page:
<https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/AAAInfoFile.xml>.
Click **Next**. See Figure 6-223 on page 314.

Note: The file contains different usernames and passwords. You can use the first one, which is BookUser/BookUser.

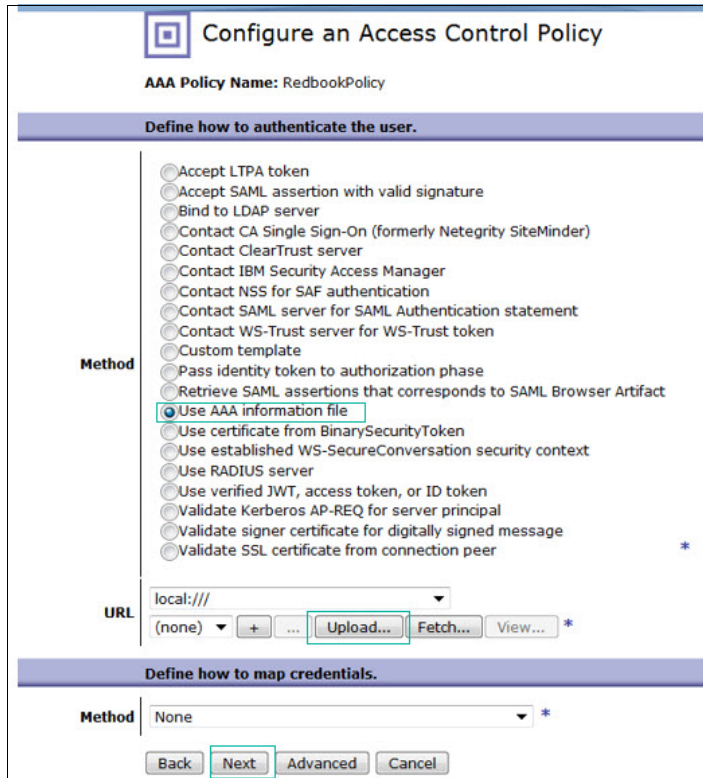


Figure 6-223 Configuring the DataPower XML firewall 12

13. Choose **Local name of request element** (this will extract the required domain for authorization), then click **Next**. See Figure 6-224.

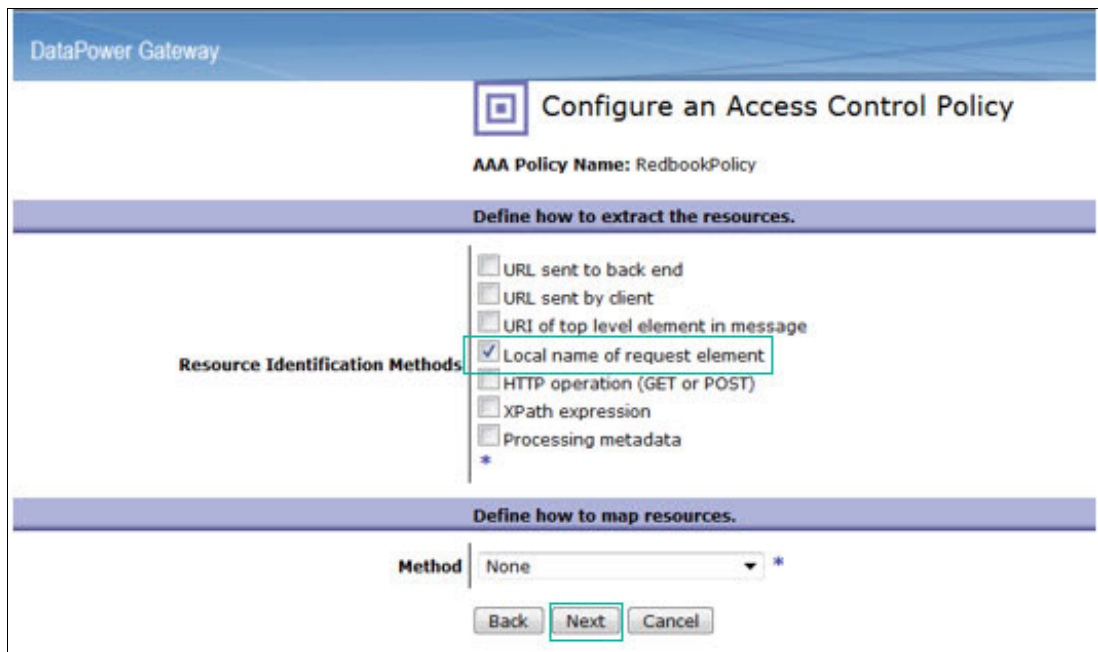


Figure 6-224 Configuring the DataPower XML firewall 13

14. Choose **Allow any authenticated client** then click **Next**. See Figure 6-225 on page 315.

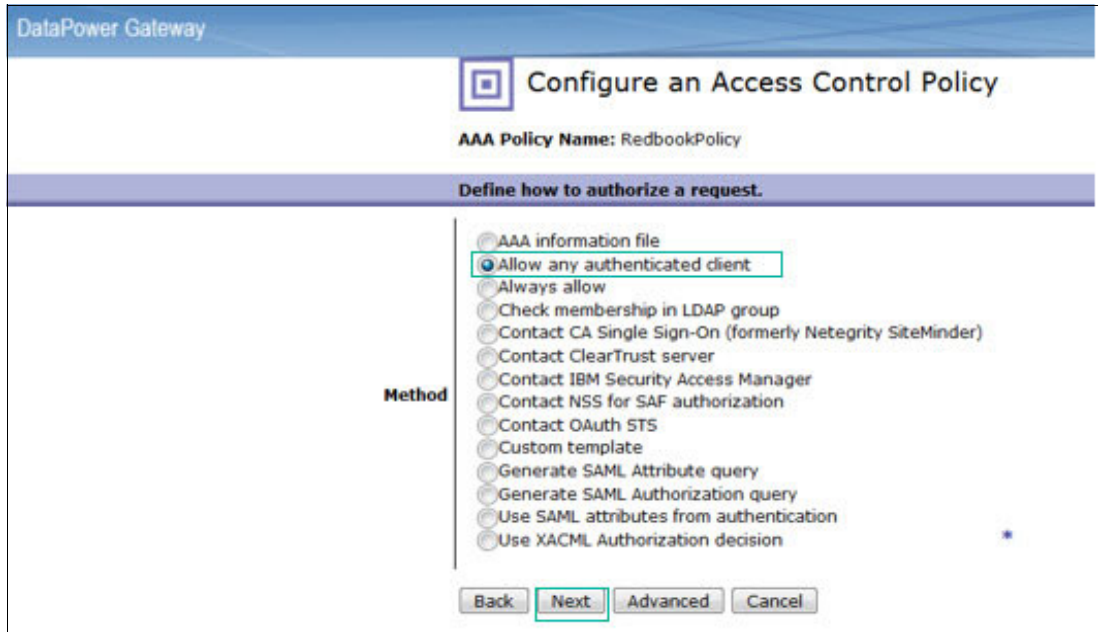


Figure 6-225 Configuring the DataPower XML firewall 14

15. Now you can see many different options of monitoring, logging, and post-processing features. Go with the default and click on **Commit** as shown in Figure 6-226 on page 316.

MONITORS	
Authorized Counter	(none) ▼ + ...
Rejected Counter	(none) ▼ Reject Counter Tool
Logging	
Enable Logging of Allowed Access Attempts	<input checked="" type="radio"/> on <input type="radio"/> off *
Log Level for Allowed Access Attempts	information ▼
Enable Logging of Rejected Access Attempts	<input checked="" type="radio"/> on <input type="radio"/> off *
Log Level for Rejected Access Attempts	warning ▼
Choose any postprocessing.	
Run Custom Post Processing	<input type="radio"/> on <input checked="" type="radio"/> off *
Send SAML Logout Message (SAML 2.0 only)	<input type="radio"/> on <input checked="" type="radio"/> off
Include a WS-Security Kerberos AP-REQ token	<input type="radio"/> on <input checked="" type="radio"/> off
Process WS-Trust SCT STS Request	<input type="radio"/> on <input checked="" type="radio"/> off
Add WS-Security UsernameToken	<input type="radio"/> on <input checked="" type="radio"/> off
Generate an LTPA Token	<input type="radio"/> on <input checked="" type="radio"/> off
Generate a Kerberos SPNEGO token	<input type="radio"/> on <input checked="" type="radio"/> off
Request TFIM Token Mapping	<input type="radio"/> on <input checked="" type="radio"/> off
Generate an ICRX token for z/OS Identity Propagation	<input type="radio"/> on <input checked="" type="radio"/> off
Generate SAML assertion with only Authentication statement	<input type="radio"/> on <input checked="" type="radio"/> off
Generate SAML assertion or response	<input type="radio"/> on <input checked="" type="radio"/> off
Generate a JSON Web Token	<input type="radio"/> on <input checked="" type="radio"/> off
Dynamic Configuration	
Dynamic configuration type	None ▼ *
<input type="button" value="Back"/> <input checked="" type="button" value="Commit"/> <input type="button" value="Cancel"/>	

Figure 6-226 Configuring the DataPower XML firewall 15

16. Click on **Done**. See Figure 6-227 on page 317.

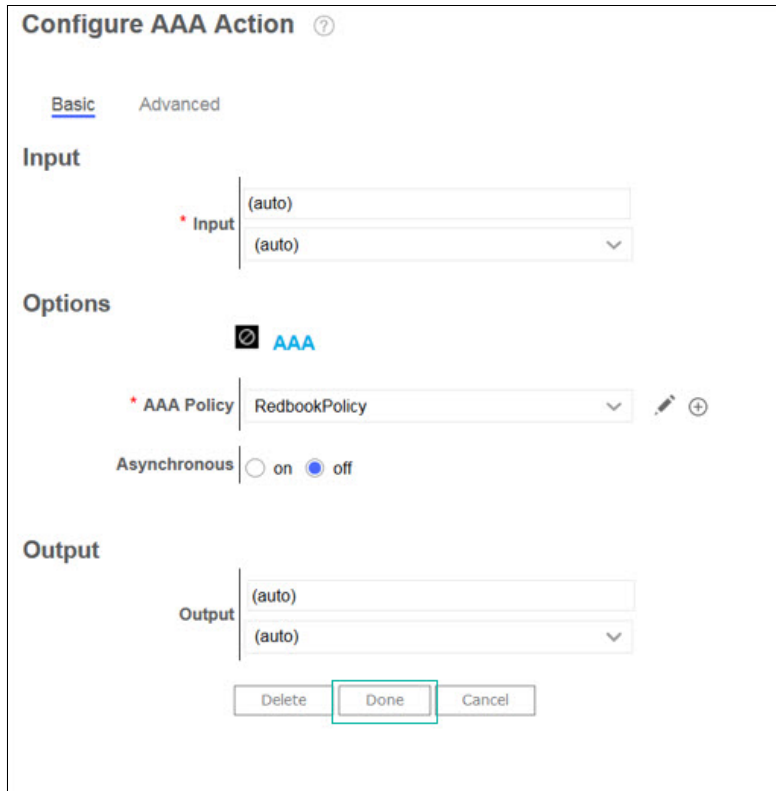


Figure 6-227 Configuring the DataPower XML firewall 16

17. Now drag the **transformation node** to the flow and **double-click** it to configure it. See Figure 6-228.

This step is to create a custom response for the authentication.

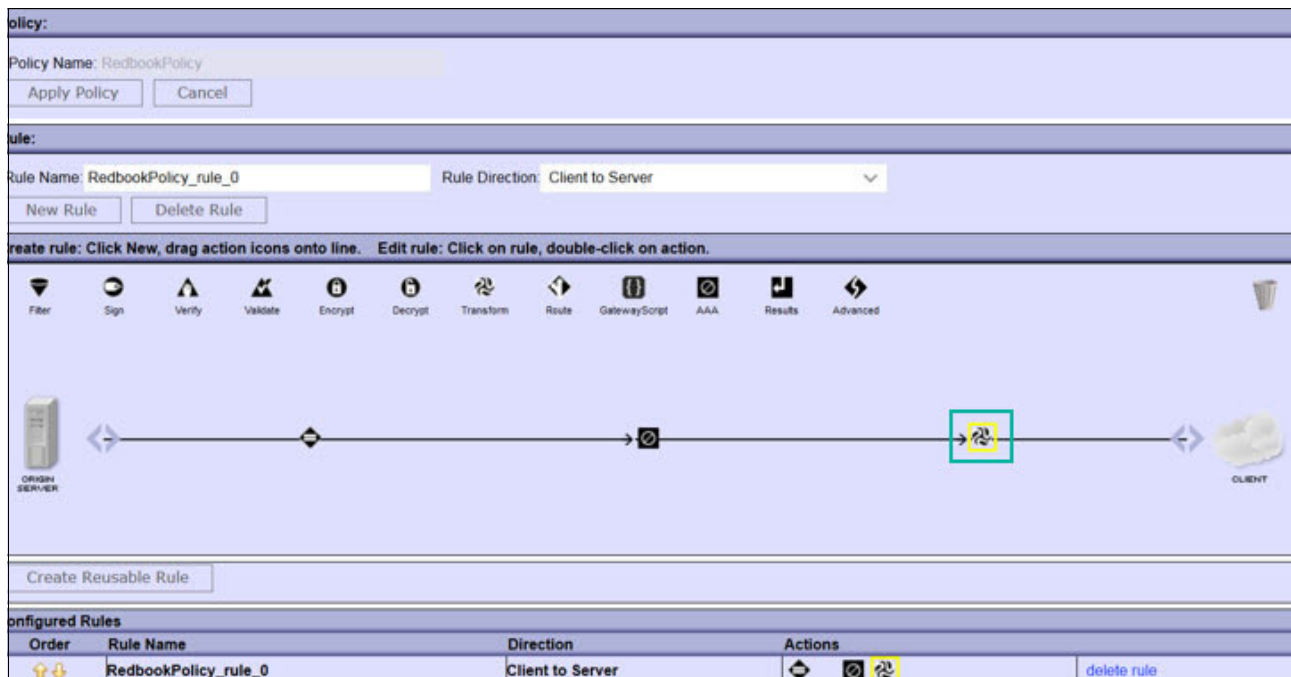


Figure 6-228 Configuring the DataPower XML firewall 17

18. Click on **Upload** and use the file `DataPowerAuthResponse.xslt` from this web page: <https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration/blob/master/chapter6/DataPowerAuthResponse.xslt>
Then click **Done**. See Figure 6-229.

The screenshot displays the 'Options' configuration panel for a DataPower XML firewall. The main heading is 'Transform with XSLT style sheet'. Under the 'Use Document Processing Instructions' section, four radio buttons are listed: 'Transform binary', 'Transform with a processing control file, if specified', 'Transform with embedded processing instructions, if available', and 'Transform with XSLT style sheet', which is selected and highlighted with a red box. Below this, the 'Transform File' section shows a dropdown menu with 'local:///'. The selected file is 'DataPowerAuthResponse.xslt', which is also highlighted with a red box. To the right of the file name are buttons for 'Upload...' (highlighted with a red box), 'Fetch...', 'Edit...', 'View...', and 'Var Builder'. The 'URL Rewrite Policy' is set to '(none)'. The 'Asynchronous' option is set to 'off'. The 'Output' section at the bottom has two dropdown menus, both set to '(auto)', and buttons for 'Delete', 'Done', and 'Cancel'.

Figure 6-229 Configuring the DataPower XML firewall 18

19. Finally, drag the **result node** to the end of the flow. See Figure 6-230 on page 319.

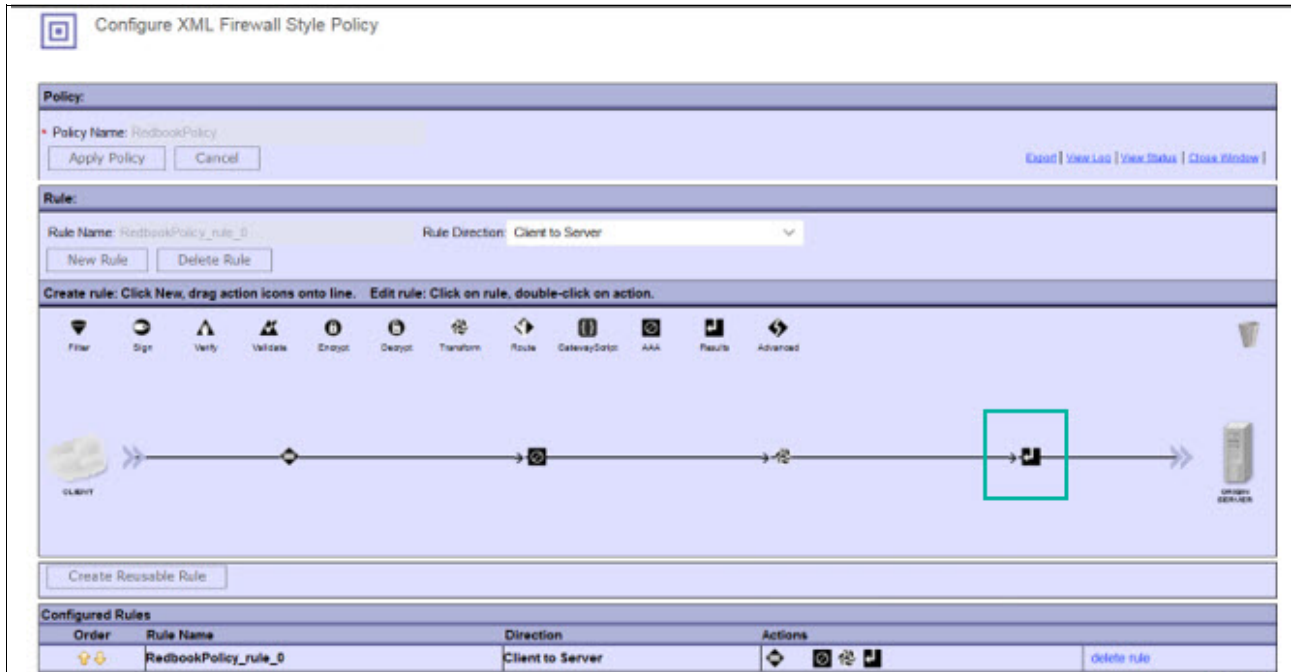


Figure 6-230 Configuring the DataPower XML firewall 19

20. Click **Apply Policy** then **Close** the window. See Figure 6-231.

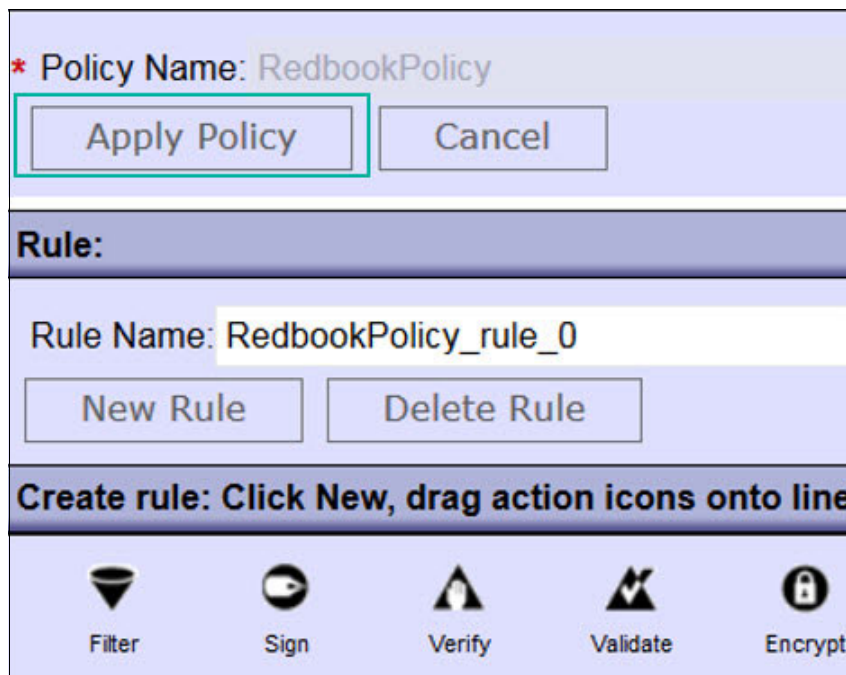


Figure 6-231 Configuring the DataPower XML firewall 20

21. Click **Apply** again on the main XML Firewall then click **Save Changes**. See Figure 6-232 on page 320.

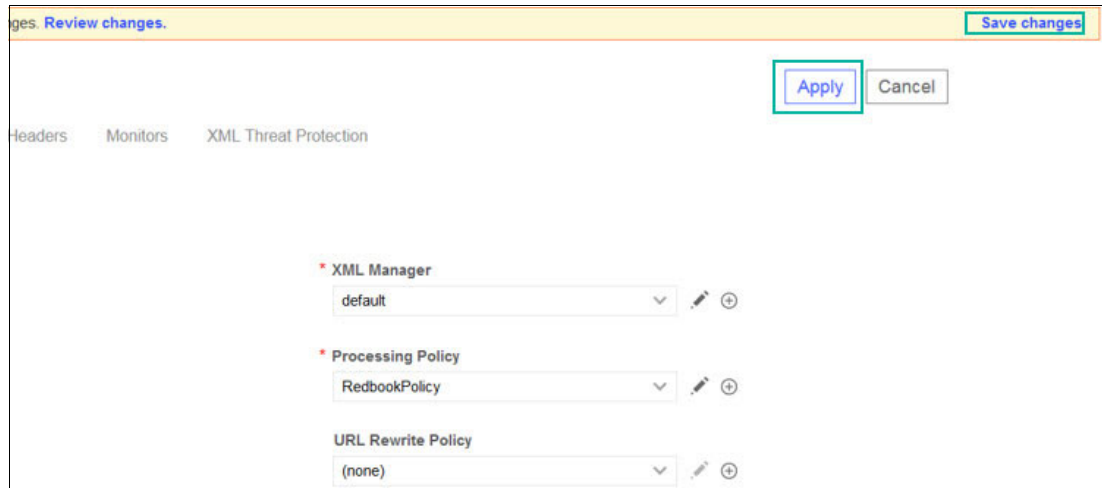


Figure 6-232 Configuring the DataPower XML firewall 21

22. Click **Advanced** and set:

- Disallow GET (and HEAD) to **off**
- Process Messages Whose Body Is Empty to **on**

Then click **Apply** as shown in Figure 6-233 on page 321.

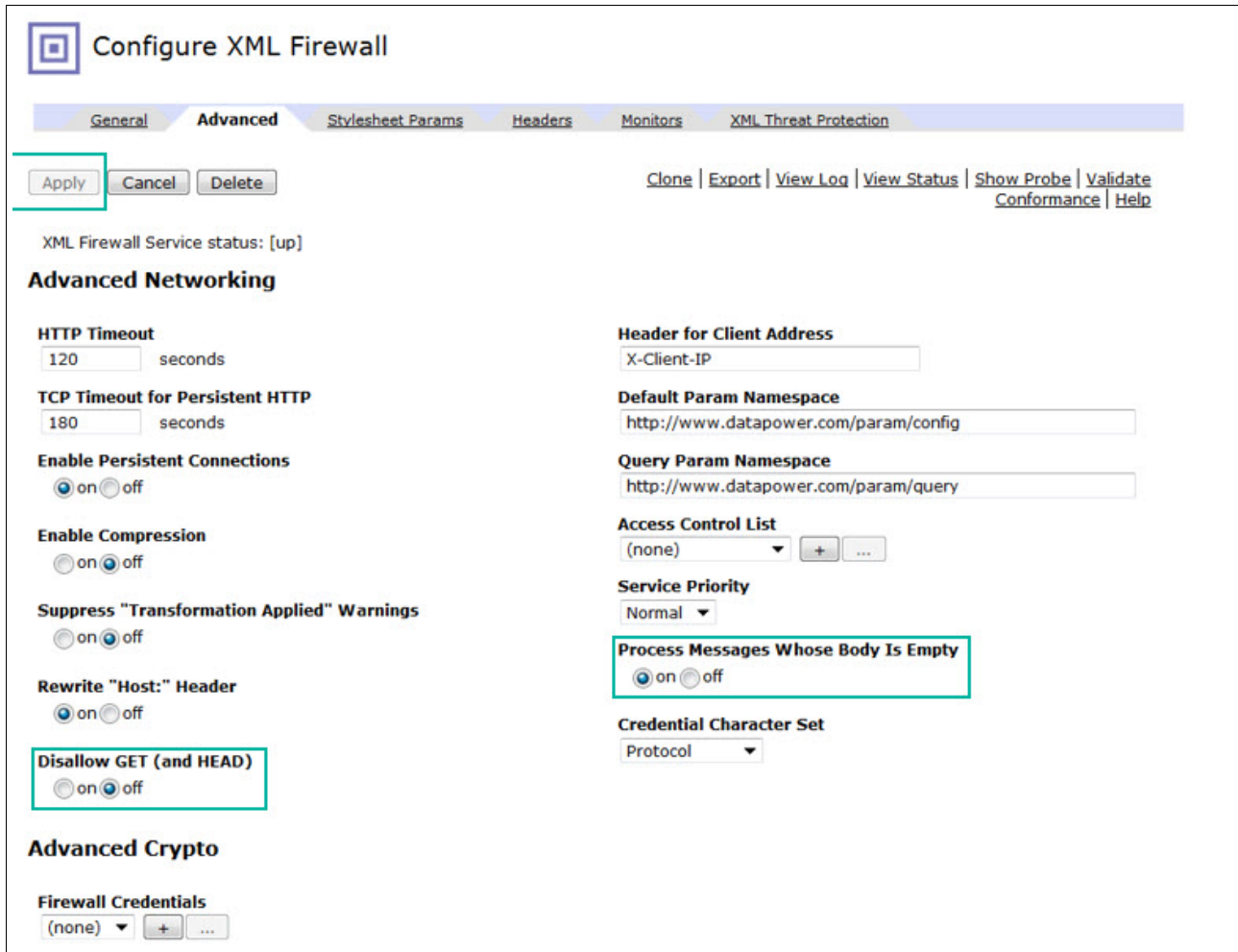


Figure 6-233 Configuring the DataPower XML firewall 22

23. To test that the Authentication URL is working, use the DataPower IP and the XML Firewall Port in the browser. You should be prompted to enter the username and password. Type the username and password BookUser/BookUser.

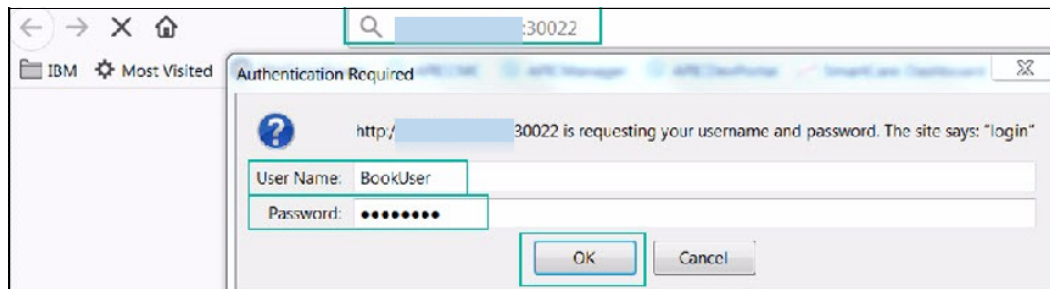


Figure 6-234 Configuring the DataPower XML firewall 23

24. When you click on **Sign In**, you should receive the SOAP response as shown in Figure 6-235 on page 322.

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<soapenv:Envelope xmlns:ws="http://ws.rcbj.com/" xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:auth>
      <return>accepted</return>
    </ws:auth>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 6-235 Configuring the DataPower XML firewall 24

Now you have a working authentication URL in the DataPower.

6.7 Create event stream from messaging

APIs have emerged as the simplest way to make data and functions accessible from back-end systems, and their role will remain fundamental to integration for some time to come. However, most modern applications have a need to respond to events as they happen, in real time.

An API is a "pull" pattern whereby the consumer knows only the state of the back-end system when they choose to pull state from it via the API. As such, it may be some time before a consumer becomes aware a change has occurred. We could of course "poll" the API at intervals to check the state of the back-end system. But the more regularly we poll, the less efficient from a resource point of view this pattern becomes.

Imagine the mobile banking applications of all customers regularly polling the bank to check the balance in order to check whether the customer has gone into overdraft. It would obviously be better to simply be only notified each time the balance changed, or better still only when an overdraft occurred. Therefore, we also need a mechanism to "push" notifications from the back-end systems up to the consumer applications.

These real time notifications of events are just one example of where a "push" model is a common requirement. We consider another example later in 6.10, "Implement event sourced APIs" on page 367.

Clearly, passing messages asynchronously isn't in any way new. IBM MQ was invented for this very purpose over 25 years ago, and indeed we could easily use IBM MQ for this purpose. Indeed, we demonstrate IBM MQ as a source of events to a cloud integration in 6.8, "Perform event-driven SaaS integration" on page 328. However, modern applications come with a subtly different set of requirements. Their requirements might be better suited to an alternative form of asynchronous communication known as "event streams", and typified by Apache Kafka, on which IBM Event Streams is built.

We have already discussed in detail the differences between traditional messaging such as IBM MQ, and event streaming technologies such as Kafka in 3.3, "Capability perspective: Messaging and event streams" on page 68 so we need not repeat that here. Suffice to say there is definitely a place for both in a solution, and indeed they can often be used alongside one another in a complementary fashion.

There are many different ways to create an event stream from a back-end system. In this section we are going to take advantage of the fact that in many organizations, a substantial IBM MQ infrastructure is already in place. For them, the simplest way to create an event

stream may well be to simply listen for IBM MQ messages and publish their payload to an event stream (as shown in option a, in Figure 6-236 on page 323).

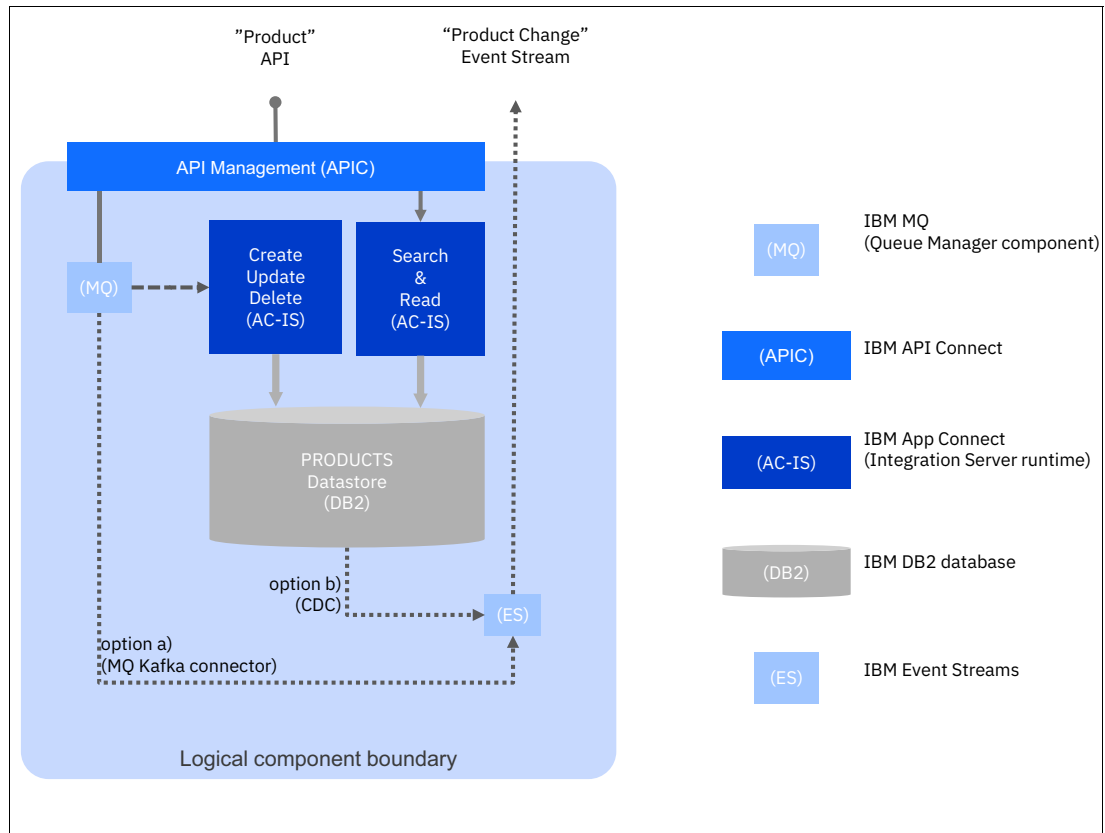


Figure 6-236 Create event stream from messaging

Option a) fits neatly into our existing scenario as we already have an IBM MQ queue that receives all data change events (creates, updates, deletes) and that's what we implement in this section.

However, it is worth noting that there will be circumstances where an input queue might not contain all the events that occur in the database. For example, imagine if we had left in place the original synchronous mechanism for performing database changes alongside the new IBM MQ based asynchronous one. Perhaps older consumers were unwilling to change to the new interaction pattern. Or perhaps they required the synchronous interaction so that they could confirm that an update had been completed. In this case, the messages on the IBM MQ queue would represent only a subset of the actual data changes that occur in the database. In these situations we would need to capture the data changes at source using one of the database replication capabilities available. Examples for Db2 are here:

https://www.ibm.com/support/knowledgecenter/en/SSTRGZ_11.4.0/com.ibm.idr.frontend.doc/pv_welcome.html?cp=SSEPGG_11.5.0

Let's return to our example, where there is an IBM MQ topic that we can use to publish events to IBM Event Streams (IBM's Kafka implementation). We are going to assume that the IBM MQ topic already exists. That way, we can focus on how to create a new subscriber to the existing IBM MQ topic, then leveraging the Kafka Connect source connector for IBM MQ.

6.7.1 Creating a new event stream topic

First, we create a new topic in IBM Event Streams:

1. Begin by logging in to your instance of Event Streams.
2. Next, create a topic as shown in Figure 6-237 on page 324.

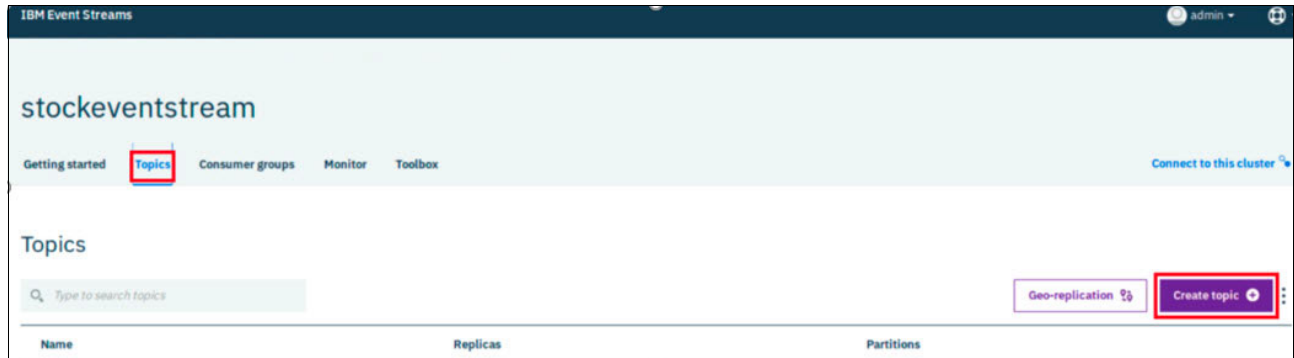


Figure 6-237 Create a topic

3. Specify a name, the number of partitions message retention and replicas that are necessary or your use case and volumes.
4. Then click the **Connect to this cluster** button as shown in Figure 6-238.

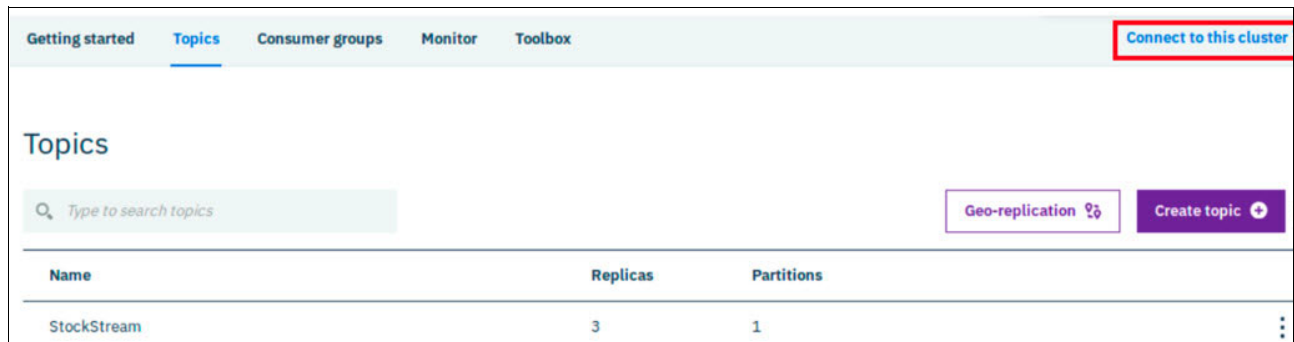


Figure 6-238 Click Connect to this cluster button

5. Save the API Key for later use.

6.7.2 Running the IBM MQ source connector

You can use the IBM MQ source connector to copy data from IBM MQ into IBM Event Streams or Apache Kafka. The connector copies messages from a source IBM MQ queue to a target Kafka topic.

Kafka Connect can be run in stand-alone or distributed mode. We cover steps for running the connector in distributed mode in a Docker container. In this mode, work balancing is automatic, scaling is dynamic, and tasks and data are fault-tolerant. For more details on the difference between stand-alone and distributed mode, see the explanation of Kafka Connect workers.

Prerequisites

The connector runs inside the Kafka Connect runtime, which is part of the Apache Kafka distribution. IBM Event Streams does not run connectors as part of its deployment, so you need an Apache Kafka distribution to get the Kafka Connect runtime environment.

Ensure you have IBM MQ v8 or later installed.

Note: These instructions are for IBM MQ v9 running on Linux. If you're using a different version or platform, you might have to adjust some steps slightly.

Setting up the queue manager

These instructions set up an IBM MQ queue manager that uses its local operating system to authenticate the user ID and password. The user ID and password you provide must already be created on the operating system where IBM MQ is running.

1. Log in as a user authorized to administer IBM MQ, and ensure that the IBM MQ commands are on the path.
2. Create a queue manager with a TCP/IP listener on port 1414: `crtmqm -p 1414 <queue_manager_name>`
For example to create a queue manager called QM1 use `crtmqm -p 1414 QM1`.
3. Start the queue manager: `strmqm <queue_manager_name>`
4. Start the runmqsc tool to configure the queue manager: `runmqsc <queue_manager_name>`
5. In runmqsc, create a server-connection channel: `DEFINE CHANNEL(<channel_name>) CHLTYPE(SVRCONN)`
6. Set the channel authentication rules to accept connections that require a userid and password:
 - a. `SET CHLAUTH(<channel_name>) TYPE(BLOCKUSER) USERLIST('nobody')`
 - b. `SET CHLAUTH('*') TYPE(ADDRESSMAP) ADDRESS('*') USERSRC(NOACCESS)`
 - c. `SET CHLAUTH(<channel_name>) TYPE(ADDRESSMAP) ADDRESS('*') USERSRC(CHANNEL) CHCKCLNT(REQUIRED)`
7. Set the identity of the client connections based on the supplied context (the user ID):
`ALTER AUTHINFO(SYSTEM.DEFAULT.AUTHINFO.IDPWOS) AUTHTYPE(IDPWOS) ADOPTCTX(YES)`
8. Refresh the connection authentication information: `REFRESH SECURITY TYPE(CONNAUTH)`
9. Create a queue for the Kafka Connect connector to use: `DEFINE QLOCAL(<queue_name>)`
10. Authorize the IBM MQ user ID to connect to and inquire the queue manager: `SET AUTHREC OBJTYPE(QMGR) PRINCIPAL('<user_id>') AUTHADD(CONNECT,INQ)`
11. Authorize the IBM MQ user ID to use the queue: `SET AUTHREC PROFILE(<queue_name>) OBJTYPE(Queue) PRINCIPAL('<user_id>') AUTHADD(ALLMQI)`
12. Stop the runmqsc tool by typing `END`.

For example, for a queue manager who is called **QM1**, with user ID **alice**, creating a server-connection channel called **MYSVRCONN** and a queue called **MYQSOURCE**, you run the following commands in `runmqsc` (Example 6-6):

Example 6-6 Sample commands

```
DEFINE CHANNEL(MYSVRCONN) CHLTYPE(SVRCONN)
SET CHLAUTH(MYSVRCONN) TYPE(BLOCKUSER) USERLIST('nobody')
SET CHLAUTH('*') TYPE(ADDRESSMAP) ADDRESS('*') USERSRC(NOACCESS)
```

```
SET CHLAUTH(MYSVRCONN) TYPE(ADDRESSMAP) ADDRESS('*') USERSRC(CHANNEL)
CHKCLNT(REQUIRED)
ALTER AUTHINFO(SYSTEM.DEFAULT.AUTHINFO.IDPWOS) AUTHTYPE(IDPWOS) ADOPTCTX(YES)
REFRESH SECURITY TYPE(CONNAUTH)
DEFINE QLOCAL(MYQSOURCE)
SET AUTHREC OBJTYPE(QMGR) PRINCIPAL('alice') AUTHADD(CONNECT,INQ)
SET AUTHREC PROFILE(MYQSOURCE) OBJTYPE(QEUE) PRINCIPAL('alice') AUTHADD(ALLMQI)
END
```

The queue manager is now ready to accept connection from the connector and get messages from a queue.

6.7.3 Configuring the connector to connect to IBM MQ

The connector requires details to connect to IBM MQ and to your IBM Event Streams or Apache Kafka cluster. You can generate the sample connector configuration file for Event Streams from either the UI or the CLI. For distributed mode, the configuration is in JSON format and in stand-alone mode it is a .properties file.

The connector connects to IBM MQ using a client connection. You must provide the following connection information for your queue manager:

- ▶ The name of the IBM MQ queue manager.
- ▶ The connection name (one or more host and port pairs).
- ▶ The channel name.
- ▶ The name of the source IBM MQ queue.
- ▶ The user name and password if the queue manager is configured to require them for client connections.
- ▶ The name of the target Kafka topic.

Using the UI

Note: The following instructions relate to IBM Cloud Private as that was what was available at the time of writing. For OpenShift the concepts will be largely the same.

Use the UI to download a .json file that can be used in distributed mode.

1. Log in to your IBM Event Streams UI.
2. Click the **Toolbox** tab and scroll to the Connectors section.
3. Go to the **Connecting to IBM MQ?** tile, and click **Add connectors**.
4. Click the **IBM MQ connectors** link.
5. Ensure that the MQ Source tab is selected and click **Download MQ Source Configuration**. Another window is displayed.
6. Use the relevant fields to alter the configuration of the MQ Source connector.
7. Click **Download** to generate and download the configuration file with the supplied fields.
8. Open the downloaded configuration file and change the values of `mq.user.name` and `mq.password` to the username and password that you used to configure your instance of IBM MQ.

Using the CLI

Use the CLI to download a .json or .properties file that can be used in distributed or stand-alone mode.

1. Log in to your cluster as an administrator by using the IBM Cloud Private CLI:

```
cloudctl login -a https://<Cluster Master Host>:<Cluster Master API Port>
```

The master host and port for your cluster are set during the installation of IBM Cloud Private.

2. Run the following command to initialize the Event Streams CLI on the cluster:

```
cloudctl es init
```

3. Run the **connector-config-mq-source** command to generate the configuration file for the MQ Source connector.

For example, to generate a configuration file for an instance of IBM MQ with the following information: a queue manager called QM1, with a connection point of localhost(1414), a channel name of MYSVRCONN, a queue of MYQSOURCE and connecting to the topic TSOURCE, run the following command:

```
cloudctl es connector-config-mq-source --mq-queue-manager="QM1"  
--mq-connection-name-list="localhost(1414)" --mq-channel="MYSVRCONN"  
--mq-queue="MYQSOURCE" --topic="TSOURCE" --file="mq-source" --json
```

Note: Omitting the **--json** flag will generate a mq-source.properties file that can be used for stand-alone mode.

4. Change the values of mq.user.name and mq.password to the username and password that you used to configure your instance of IBM MQ.

The final configuration file will resemble what you see in Example 6-7.

Example 6-7 Final configuration file

```
{  
  "name": "mq-source",  
  "config": {  
    "connector.class":  
"com.ibm.eventstreams.connect.mqsource.MQSourceConnector",  
    "tasks.max": "1",  
    "topic": "TSOURCE",  
    "mq.queue.manager": "QM1",  
    "mq.connection.name.list": "localhost(1414)",  
    "mq.channel.name": "MYSVRCONN",  
    "mq.queue": "MYQSOURCE",  
    "mq.user.name": "alice",  
    "mq.password": "passw0rd",  
    "mq.record.builder":  
"com.ibm.eventstreams.connect.mqsource.builders.DefaultRecordBuilder",  
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",  
    "value.converter": "org.apache.kafka.connect.storage.StringConverter"  
  }  
}
```

A list of all the possible flags can be found by running the command **cloudctl es connector-config-mq-source --help**. Alternatively, see the sample properties file at

<https://github.com/ibm-messaging/kafka-connect-mq-source/tree/master/config> for a full list of properties you can configure, and also see <https://github.com/ibm-messaging/kafka-connect-mq-source> for all available configuration options.

Downloading the MQ source connector

Perform the following steps:

1. Log in to your IBM Event Streams UI.
2. Click the **Toolbox** tab and scroll to the **Connectors** section.
3. Go to the Connecting to IBM MQ? tile, and click **Add connectors**.

Ensure that the MQ Source tab is selected and click Download MQ Source JAR, this will download the MQ Source JAR file.

Configuring Kafka Connect

IBM Event Streams provides help with getting a Kafka Connect environment.

1. Follow the steps at <https://ibm.github.io/event-streams/connecting/setting-up-connectors> to get Kafka Connect running. When adding connectors, add the MQ source connector that you downloaded earlier.

2. Verify that the MQ source connector is available in your Kafka Connect environment:

```
$ curl http://localhost:8083/connector-plugins  
[{"class":"com.ibm.eventstreams.connect.mqsource.MQSourceConnector","type":"source","version":"1.1.0"}]
```

3. Verify that the connector is running. For example, If you started a connector called mq-source:

```
$ curl http://localhost:8083/connectors  
[mq-source]
```

4. Verify the log output of Kafka Connect includes the following messages that indicate the connector task has started and successfully connected to IBM MQ:

```
INFO Created connector mq-source  
INFO Connection to MQ established
```

Send a test message

1. To add messages to the IBM MQ queue, run the **amqspout** sample and type in some messages:

```
/opt/mqm/samp/bin/amqspout <queue_name> <queue_manager_name>
```

2. Log in to your IBM Event Streams UI.
3. Navigate to the Topics tab and select the connected topic. Messages appear in the message browser of that topic.

6.8 Perform event-driven SaaS integration

With the vast adoption of SaaS applications, there is an inevitable need to integrate these either to keep data in sync, or to progress business processes. It is no longer feasible to expect a centralized integration team to keep up with these new integration demands.

Business teams need an easy, guided, intuitive, data driven integration tooling that they can use themselves, without having to refer to a central team of integration specialists. They need an agile integration tooling that is of low complexity, and highly productive with extensive list of built-in connectors to integrate sales, marketing and CRM applications.

IBM App Connect Designer is a browser-based, all-in-one integration tool for connecting applications, integrating data from on-prem service to Cloud, and building and invoking APIs. You can build flows that recognize and respond to new events, or batched or events, or are triggered based on a scheduler, and deploy them within minutes on IBM Cloud.

It should be noted that while IBM App Connect can be purchased separately, users of IBM App Connect now have access to Designer in order to use the vast array of SaaS connectors, under their current license agreement.

6.8.1 Scenario

In this section, we illustrate how IBM App Connect Designer can be used to help a Store Warehouse business team to be productive adopting SaaS integration to connect to Salesforce, Slack and Gmail, based on the receipt of an IBM MQ message as shown in Figure 6-239.

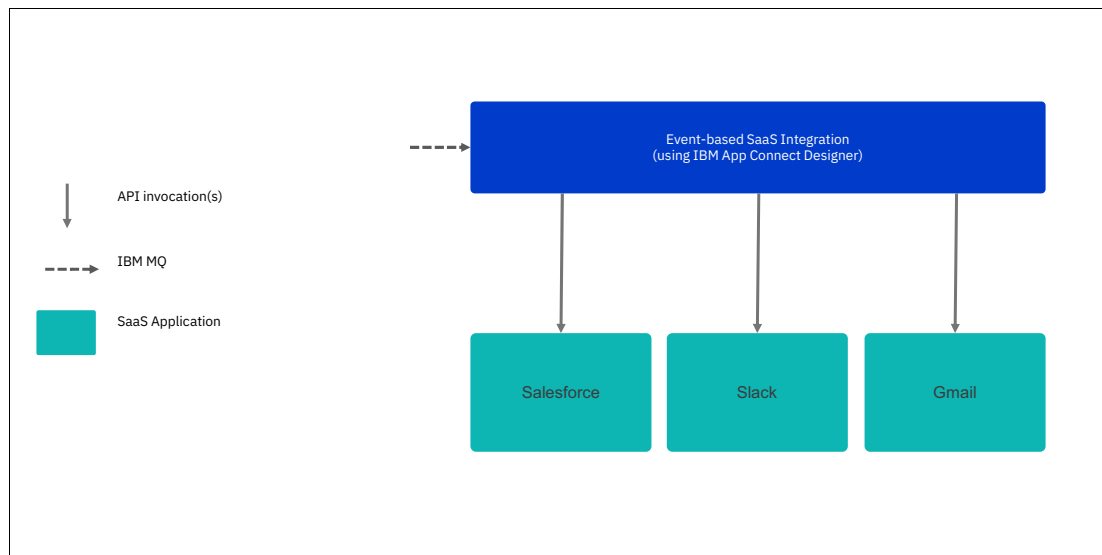


Figure 6-239 Perform event-driven SaaS integration

Currently, the store uses a manual process to track who are the customers that have ordered this stock, generate the customer list for confirmed orders, and send emails to customers with orders in pending status. This manual order process involves multiple different steps, accessing to different data repositories to do update and report generation.

The business team want to be able to automate the order tracking process whenever new stock has arrived. They have recently adopted Salesforce SaaS as their CRM to help improve customer shopping experience. Using IBM App Connect Designer, the team is now able to automate the order process on the fly using an Event Driven flow. When new stock has arrived, it will trigger a flow via an IBM MQ messaging that contains the product code. The flow retrieves all the orders from Salesforce that has this product ordered, with associated customer information. Then the list is processed according to order status. If the status is activated, a row will be generated to Google Sheet. Otherwise, send an email to inform the customer about stock arrival. An instant Slack message will also be generated if the email

address is missing. The flow is completed with an automatic notification being sent after the order list is processed.

6.8.2 IBM App Connect event-driven flow to Salesforce, Google and Slack SaaS applications

This tutorial assumes that you have signed up for free or trial accounts for Salesforce developer, Gmail, and Slack or that you have business accounts. In addition, you have already registered for a free IBM Cloud user ID, where you can access a full catalog of IBM Cloud solutions, including IBM App Connect Designer and IBM MQ on Cloud, which are used in this tutorial.

Additional references:

- ▶ How to use IBM App Connect with Salesforce:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-salesforce/>

- ▶ How to use IBM App Connect with Gmail:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-gmail/>

- ▶ How to use IBM App Connect with Slack:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-slack/>

The steps in the next sections guide you through the creation of the following event flow that is shown in Figure 6-240 and Figure 6-241.

Note: For readability, we split the event flow in two figures.

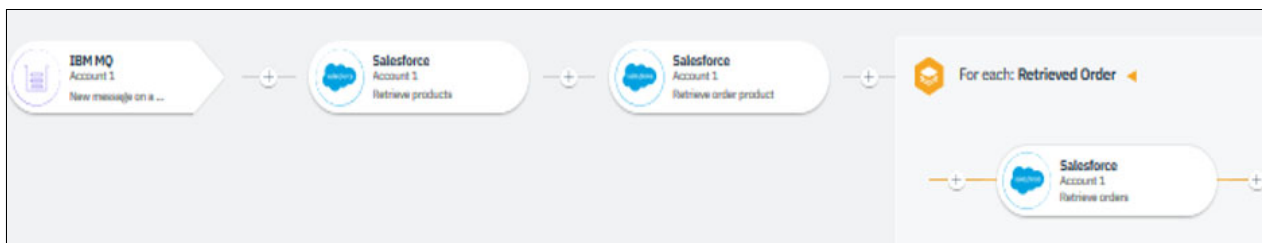


Figure 6-240 Completed SaaS Integration event flow - 1

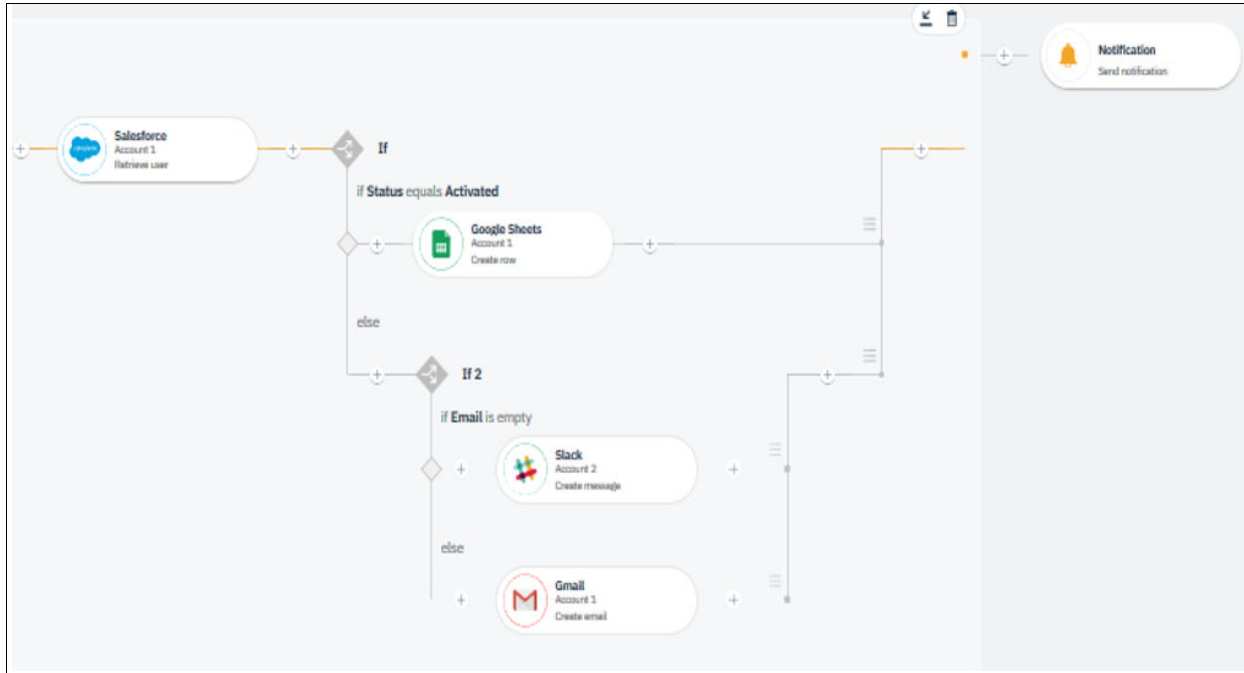


Figure 6-241 Completed SaaS Integration event flow - 2 (Continued from the previous figure)

6.8.3 Prerequisites

This scenario requires Salesforce objects: Account, Product, Order and Contract to be created or already existed. Accounts will have valid contract which link to products and orders. You should also have an account with Google, to generate the process order list in Google sheet and using Slack for instant messaging. Using IBM MQ message as our event flow trigger, that is, upon arrival of a message on the queue, the IBM App Connect flow will be initiated and run.

If you are new to IBM MQ on Cloud, refer to the following tutorial to learn and create a queue for use with the IBM App Connect flow.

Note: To access IBM MQ on Cloud Tutorial refer to the following link:
<https://www.ibm.com/cloud/garage/dte/tutorial/tutorial-mq-ibm-cloud>.

6.8.4 Create flows

Perform the following steps to create the flows for this tutorial:

1. Log in to IBM MQ console on the IBM MQ on Cloud instance:

As shown in Figure 6-242 you should have created a queue manager QM1, with these default local queues. The queues are running and waiting for messages to arrive, in this scenario, it will be the product ID that has arrived in the warehouse. We will put a message in the queue when we perform the flow testing:

▲ Name	Queue type	Queue depth
DEV.DEAD.LETTER.QUEU	Local	0
DEV.QUEUE.1	Local	0
DEV.QUEUE.2	Local	0
DEV.QUEUE.3	Local	0
Total: 4		Last updated: 5:03:08 PM

Figure 6-242 QM1 on IBM Cloud

2. Log in to App Connect Designer.
3. From the Dashboard, click **New** → **Event-driven flow**, and name it as Process Salesforce Orders. Figure 6-243 shows the IBM App Connect dashboard.

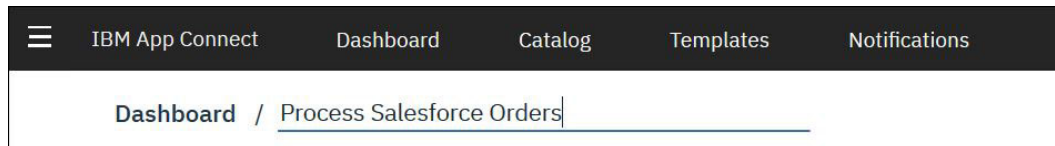


Figure 6-243 ProcessSalesforce Orders

4. Select **MQ with new message arrive on queue** as the event that is to trigger the flow. An IBM MQ connection will be created using the account you added. See Figure 6-244.

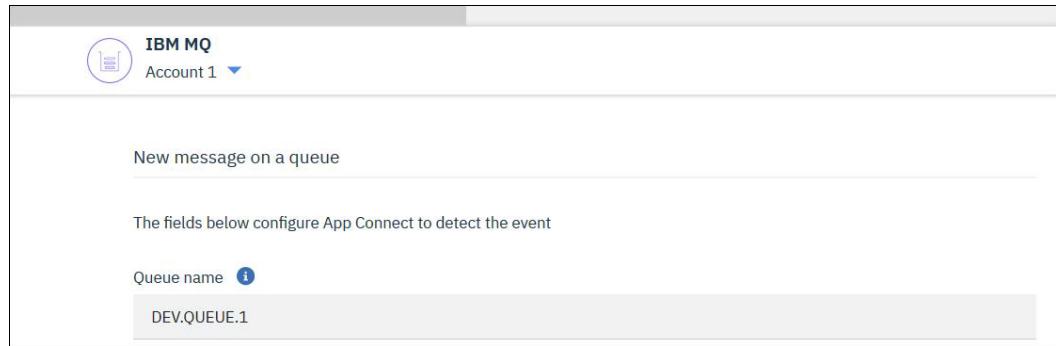


Figure 6-244 Using IBM MQ as the trigger application

5. In the following steps we create integration to different SaaS applications, Salesforce, Gmail and Slack using the IBM App Connect Designer’s out-of-the-box connectors, to complete the order process automation.
 - a. Explanation: To retrieve the stock description and customer information, we need to connect to the **Salesforce** → **Retrieve Orders** application and the **Salesforce** → **Retrieve Users** application. In Salesforce, Objects are linked via relationship. In this case, Order Product is linked to Orders via Product_ID, and Owner_ID is linked to Account (Customer Information). Hence, there is no need to write a complex lookup query to join various objects to get to the required data. IBM App Connect Designer simply and easily accesses to these records by invoking the exposed Application’s API and out-of-the-box connectors.

Figure 6-245 shows the completed Salesforce flow design which we will construct in the next steps.



Figure 6-245 Retrieve Order and Customer information

Tip: Refer to the Salesforce website to understand Object Relationship: https://help.salesforce.com/articleView?id=overview_of_custom_object_relationships.htm&type=5.

- b. Choose **Salesforce** → **Orders** → **Retrieve Products** as the next application after MQ. Set a condition for the data retrieval where you want to retrieve the product details using the product ID:
- c. Click **Add condition** and then select **Order ID** from the drop-down list.
- d. Leave the operator as equals. Then in the adjacent field, select Message data from the list.
- e. Set the maximum number of items to retrieve as 10. See the following note.
- f. If no item is found, we want to continue the flow and issue a “204: No content status code 1”.

Note: You can easily modify the flow to handle larger quantity of new stock arrival, and run it as a scheduled batch job. Batch processes are optimized for handling much larger volumes of data than the standard retrieve action. More information on batch processing:

<https://developer.ibm.com/integration/blog/2018/03/16/introducing-batch-processing-in-ibm-app-connect/>.

Figure 6-246 shows Salesforce Retrieve products.

The screenshot displays the configuration for a 'Retrieve products' action in a Salesforce flow. At the top, it shows the 'Salesforce' account. The main configuration area includes a 'Where' clause with the field 'Product Code' set to 'equals' 'Message data'. Below this is an 'Add condition' button. The 'Maximum number of items to retrieve' is set to 10. Underneath, there are two sections: 'If the limit is exceeded:' with options 'Exit the flow with an error' and 'Process 10 items from the collection' (selected); and 'If no items are found:' with options 'Continue the flow and issue a '204: No content' status code' (selected) and 'Exit the flow with a '404' error code'.

Figure 6-246 Retrieve Product information for the arrival product code

- g. Next, we want to generate a list of orders from Salesforce that contains this product. Repeat the previous step. But this time, choose **Salesforce** → **Orders** → **Retrieve Order Product** using the Product ID retrieved from previous step (b). Product ID is the internal Salesforce generated ID that link the Product to the Orders. See Figure 6-247 on page 335.

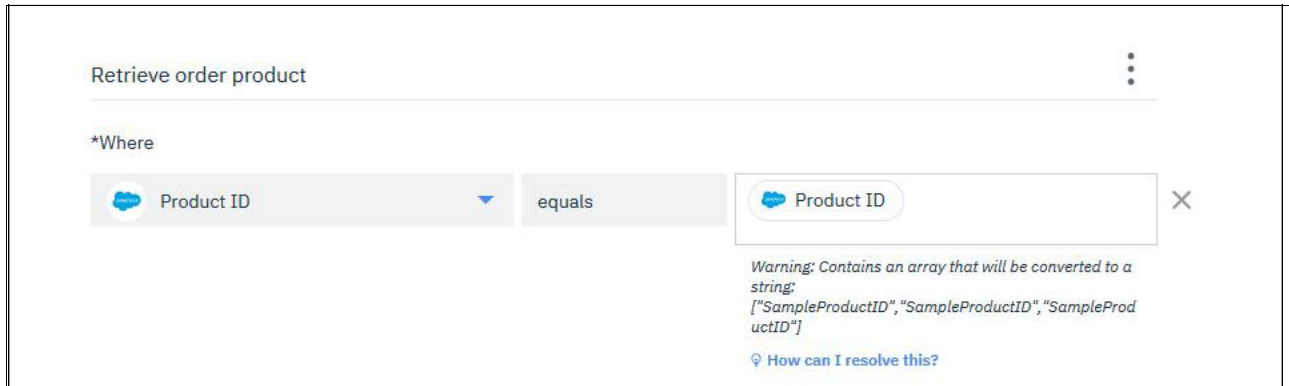


Figure 6-247 Retrieve order product

- h. To process all the orders from the retrieved list, we use For each loop, under the Toolbox option. The collection of items to process is called Salesforce/Retrieve order product/OrderProducts.
- i. Provide a display name to the For each loop, use **Retrieved Order**.
- ii. Accept the default, **Process all the items sequentially**.
- iii. Choose **Process all other items and continue the flow**.

Figure 6-248 shows the For each loop definition.

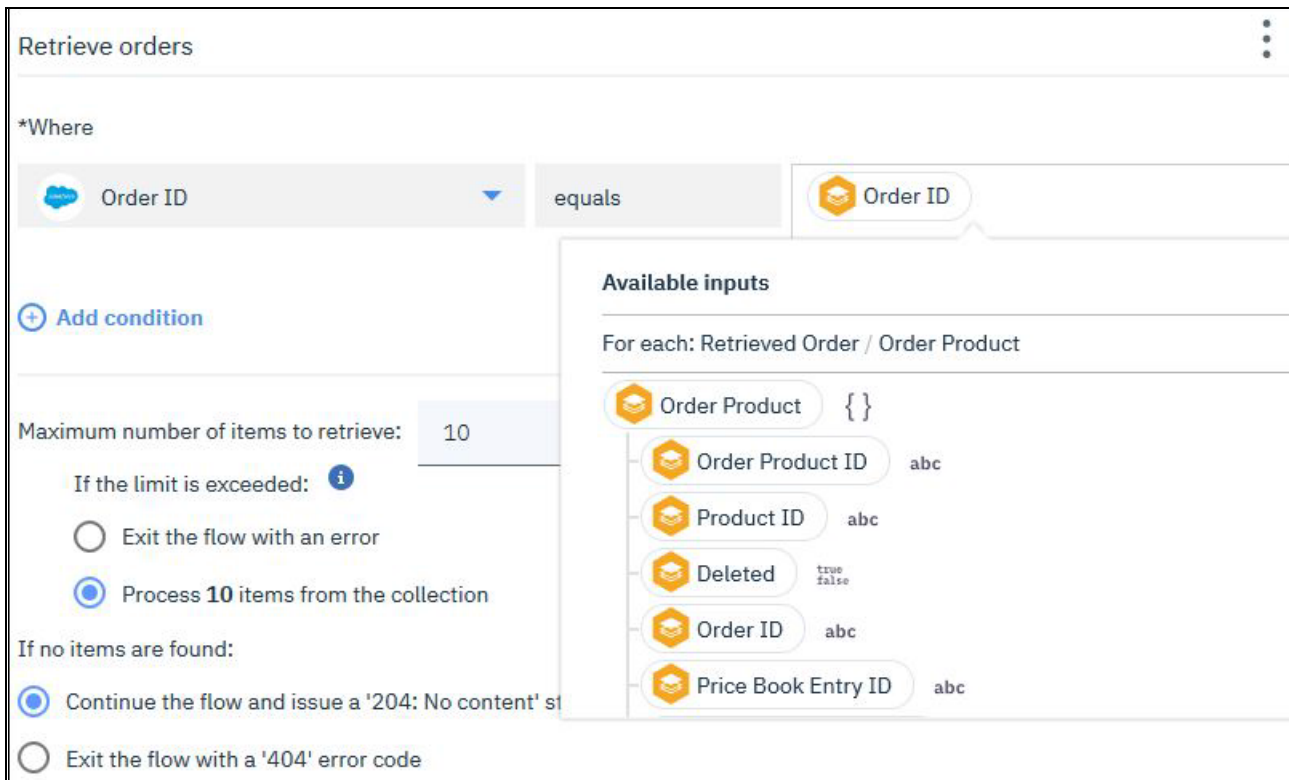


Figure 6-248 Process each order

- i. For each retrieved order, we want to retrieve the product description from the Salesforce application that is called Retrieve Orders and customer information from the Salesforce application that is called Retrieve Users.

- i. Choose **Salesforce** → **Orders** → **Retrieve orders** as the next application in the For loop. Set a condition for the data retrieval where you want to retrieve the product details using the order ID:
- ii. Click **Add condition** and then select Order ID from the drop-down list.
- iii. Leave the operator as equals. Then in the adjacent field, click the **Insert Reference** button and select **Order ID (\$Foreachitem.OrderId)**. See Figure 6-249.

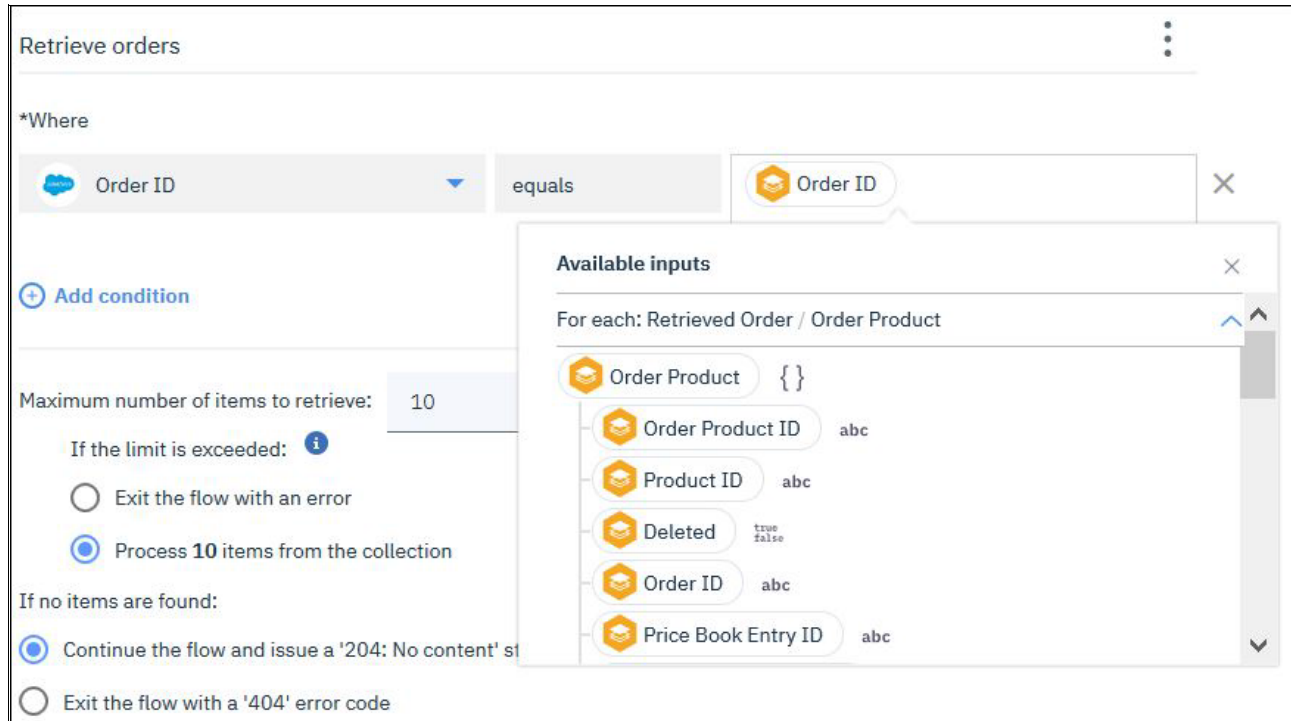


Figure 6-249 Select Order ID

- iv. Set the Maximum number of items to retrieve as 10.
- v. If no item is found, we want to continue the flow and issue a “204: No content status code 1” message.
- vi. To retrieve customer information, we choose **Salesforce** → **Orders** → **Retrieve user** as the next application to connect. Follow the preceding steps, and fill in the information as shown in Figure 6-250 on page 337.

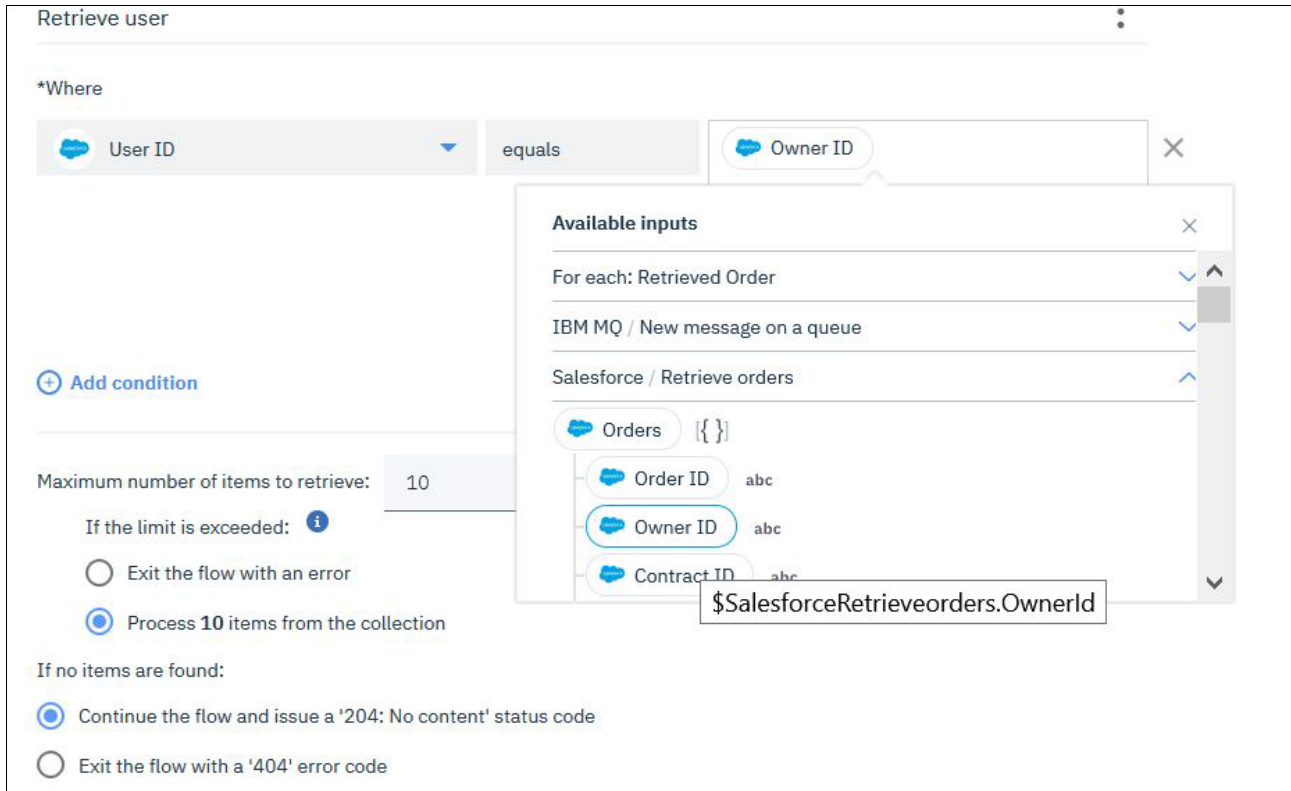


Figure 6-250 Retrieve customer information for the retrieved order

- j. For our final action, we want to process orders based on order status. We want to generate a Google Sheets spreadsheet that contains all the orders with Activate status for sending to the warehouse processing. For the orders status that are in Draft, we want to inform customers via email on stock arrival. Otherwise, send an instant message on Slack for orders that have missing email address.

Figure 6-251 shows the completed Nested IF conditions.

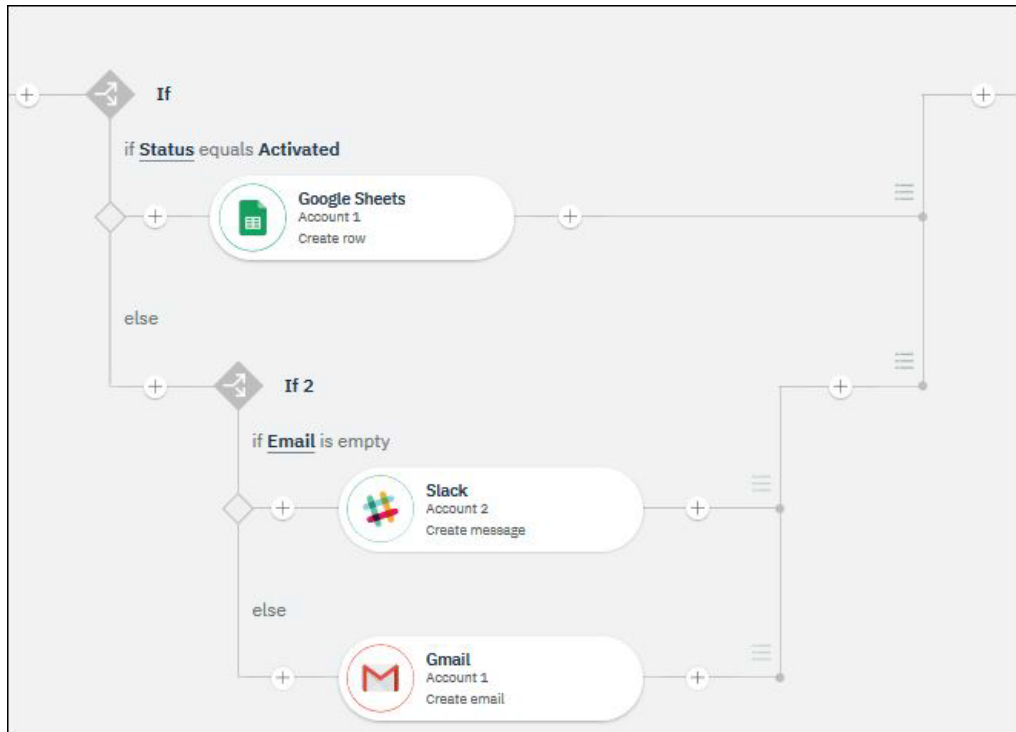


Figure 6-251 Process orders based on order status

- k. We need to define some conditional logic to make that decision, so add an If (conditional) node to your flow. Click the plus (+), open the Toolbox tab, then select **If (conditional)**. In the If node dialog box, configure the “if” branch.
 - i. In the first field of the “if” statement, expand **Salesforce** → **Update** or create contact response. Then, select **Status** (\$SalesforceRetrieveorders.Status) from the insert reference drop-down and type **Activated** next to equals. See Figure 6-252.

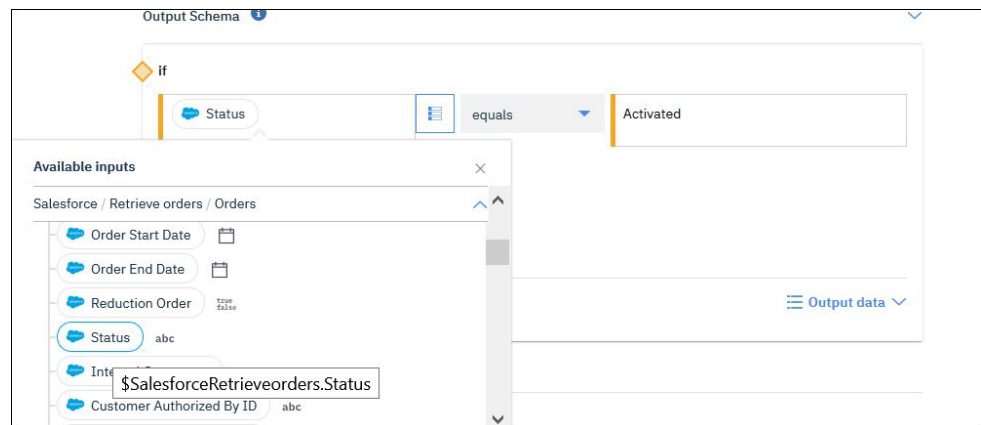


Figure 6-252 Selecting orders that are Activated

- ii. Click (+) and select **Generate Google Sheet Create** row to capture the orders as the next action. If you haven't already connected a Google sheets account, click **Connect to Google Sheets** and follow the instructions to allow IBM App Connect to connect to your Google Sheets account.

- iii. Select the **Google Sheets spreadsheet** (and then the worksheet) that you configured with the column headings: Customer Name, Order Product ID, Qty and Date.
- iv. For each field that you want to populate, click the **Insert a reference** icon, then select the Salesforce field that contains the data that you want to transfer to Google Sheets. The first field will be customer name (`$SalesforceRetrieveuser.FirstName`) complete the fields as shown in Figure 6-253 on page 339.

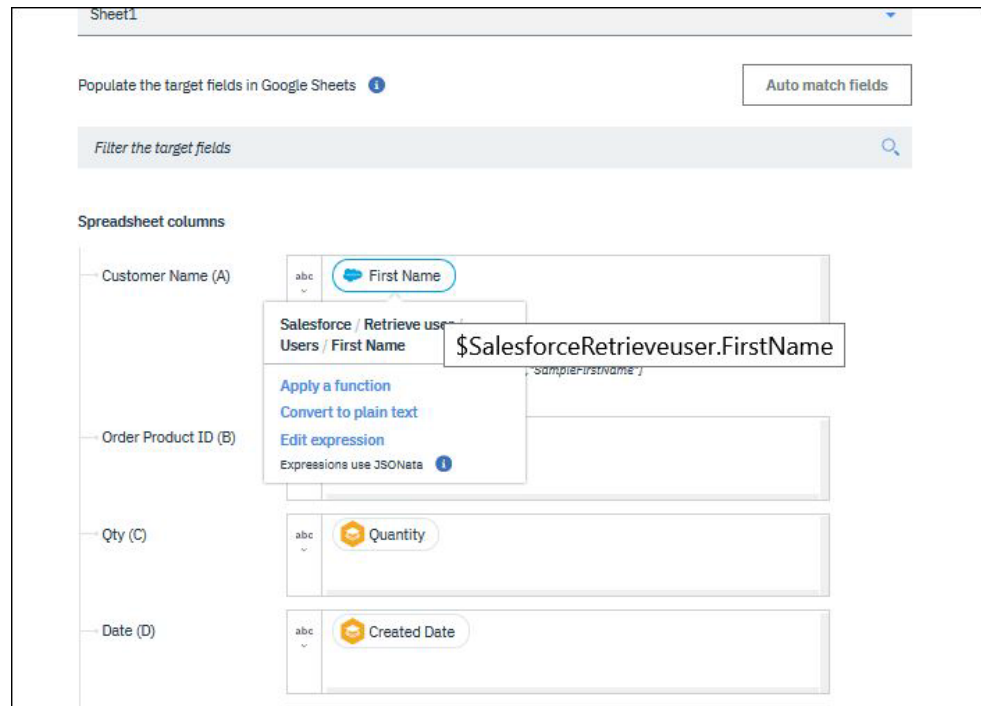


Figure 6-253 Create a Google Sheet row for activated orders

- I. In the Else branch, click **Add else if** to create a second IF condition from the Toolbox, it will be automatically called if2. In this condition, we check the email ID if it is present or missing in the Salesforce record. Populate the If condition as shown in Figure 6-254 on page 340.

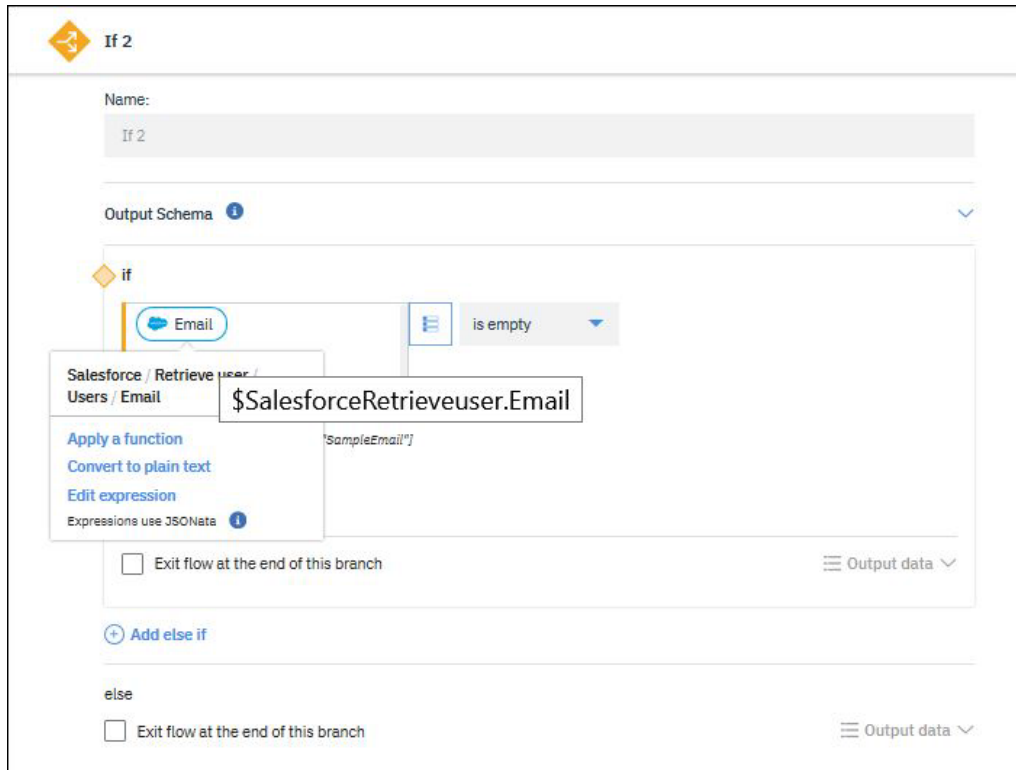


Figure 6-254 Check email ID is empty

- i. Click the (+) and then select **Slack** as your next application.
- ii. Select **Message** → **Create Message** as the Slack action.
- iii. If you haven't already connected to a Slack account, click **Connect to Slack** and follow the instructions to allow IBM App Connect to connect to your Slack account.
- iv. Select the channel that you want to post the message to. For this tutorial we have chosen *test appcon* so that only authorized users in that channel can see the message. Next, we format the message content in the text box using the order information we retrieved from Salesforce, namely **Order ID** and **Order Name**.

Figure 6-255 on page 341 shows the formatted Slack message.

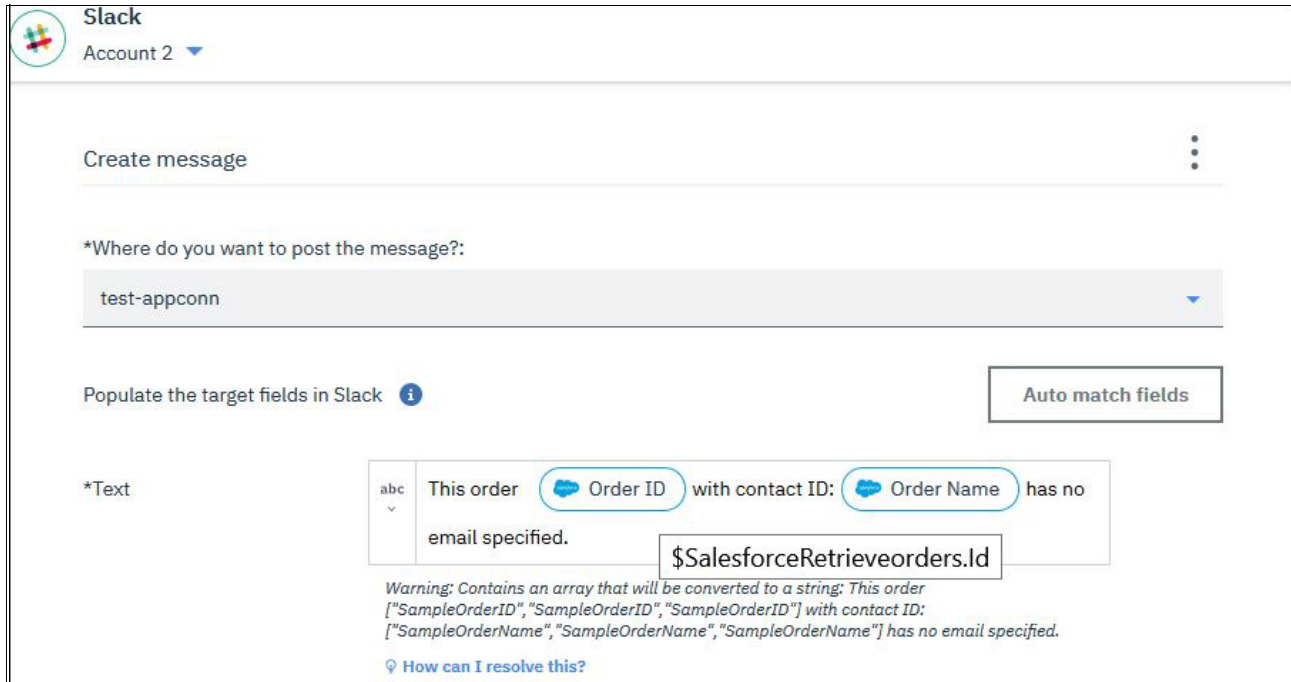


Figure 6-255 Slack instant message

- v. Complete the Else action for if2 condition by connecting to Gmail as next action to inform customers of stock arrival. Click the (+) and then select Gmail > Create message as the action that IBM App Connect should use to send an email to inform customer of stock arrival.
- vi. If not already connected, click **Connect to Gmail** and then specify the values that will allow IBM App Connect to connect to your Gmail account. Then click **Connect**.
- vii. Complete the Gmail fields as follows:
 - **To:** Click within the field and then click **Insert a reference**. From the list, expand **SalesforceRetreiveUser** → **Users** and select **Email**.
 - **Subject:** Your order has arrived.
 - **Body:** Type *Dear* and add a trailing space., click **Insert a reference** icon and select **FirstName** under **SalesforceRetreiveUser** → **Users** and add a comma (,) after firstname. Start a new line and type “Your order item” and select **message data** from Insert a reference icon list, then type “has arrived. Please let us know if you would like to confirm the order. Thanks.”

Figure 6-256 on page 342 shows the Gmail message.

Create email ⋮

Populate the target fields in Gmail Auto match fields

***To** abc Email

Warning: Contains an array that will be converted to a string: ["SampleEmail","SampleEmail","SampleEmail"]
[How can I resolve this?](#)

Subject abc Your order item has arrived

Body abc Dear First Name,
Your ordered item Message data has arrived. Please let us know if
you would like to confirm the order .
Thanks.

Figure 6-256 Compose Gmail

6. The flow ends with a notification sent to the IBM App Connect Designer dashboard to inform that all orders for the product ID have been processed. *The product id represents the message data that was put into our mq message payload that trigger our SaaS integration flow at the beginning.*

Figure 6-257 shows the notification sent to the IBM App Connect dashboard.

🔔 **Notification**

***Title:**

Orders have been processed

Description:

abc All orders for product Message data have been processed.

\$Trigger.msgPayload

Figure 6-257 Notification to the IBM App Connect dashboard

6.8.5 Test your flow

When you have successfully created the flow, you will see the running instance **Process Salesforce Orders** on the IBM App Connect dashboard.

Figure 6-258 shows the flow instance in Running state.

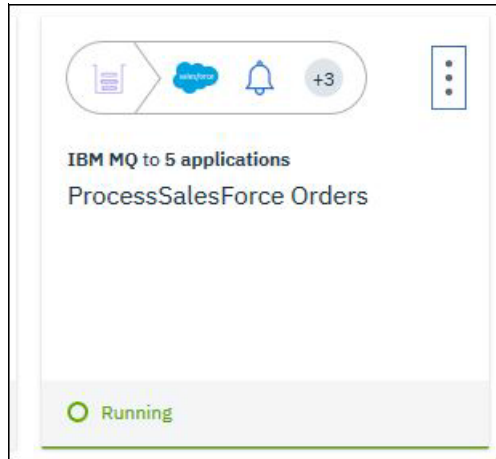


Figure 6-258 Process Salesforce Orders

1. To test the flow, we first ensure that the related objects are existing in Salesforce with the product code that represent the new stock, this means Orders and Account are linked via an internal generated product ID to Orders and Order Product. If this is not already existed, see step 3 in section 6.8.4, “Create flows” on page 331. You can also refer to https://trailhead.salesforce.com/en/content/learn/modules/field_service_maint/field_service_maint_assets.
2. New stock has arrived via an IBM MQ message:
 - a. Log in to IBM MQ console on IBM Cloud. Put a message into DEV.QUEUE.1 that represent the new stock that has arrived.
 - b. The IBM App Connect flow Process Salesforce Orders will be triggered. A notification on the IBM App Connect dashboard will be generated showing that all orders have been successfully processed for the product code that you put into the IBM MQ message.
3. Orders List is generated as a Google Sheet:
 - a. Log in to Google, and check the Google Sheet generated for the Order list. Note that all the Order Product IDs should be the same, this is the internal Salesforce product ID. Figure 6-259 shows the generated list.

	A	B	C	D
1	Customer Name	Order Product ID	Qty	Date
2	James Brown	8022v00000LlqxIAAB	10	2019-07-18T15:50:24.000+0000
3	Diane Pierce	8022v00000LlqxIAAB	10	2019-07-18T15:50:24.000+0000
4	Tan Lin	8022v00000LlqxIAAB	10	2019-07-18T15:50:24.000+0000
5	Jose Lopez	8022v00000LlqxvAAB	4	2019-07-18T16:12:07.000+0000

Figure 6-259 Sample generated OrderList

4. Notify customer of new stock arrival via email:
 - a. Log in to the email account of the email address that was specified in the customer email of the Salesforce order. You will see a email to notify you that the new stock has arrived.

Figure 6-260 shows the email to customer.

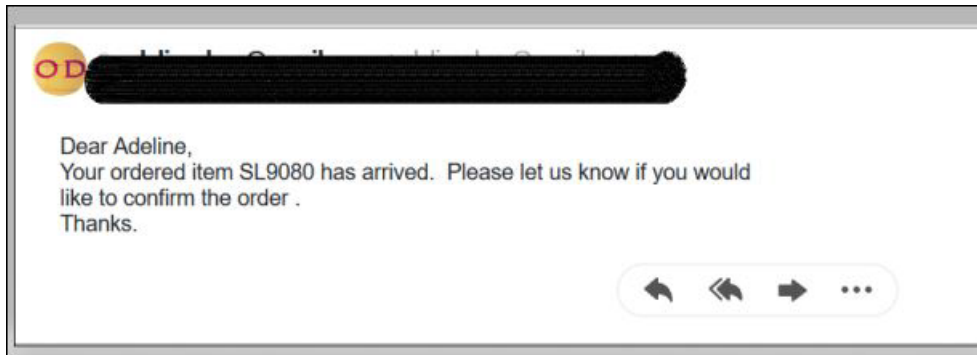


Figure 6-260 Generated email to customer

5. Instant messages for missing email address, and notification of flow completion:
 - a. You will receive instant messages in Slack notifying you on orders with missing email address, if any. In this case, we do not have any missing email address in the Salesforce Orders.
 - b. A notification message will also be shown on the IBM App Connect dashboard to inform you of the successful completion of the order processing for the new stock.

6.8.6 Conclusion

You have seen how quickly the business team is able to process all the orders upon arrival of the new stock, using the intuitive user interface of IBM App Connect Designer, connecting various SaaS applications with extensive list of connectors.

By using IBM App Connect Designer to act on events and automate process through SaaS application connectors, business teams can reduce time to market and improve ROI.

Business requirements are built and delivered independently on a scalable, secured, and industry-based, standard IBM Cloud platform.

6.9 Implementing a simple hybrid API

To deliver engaging customer experiences, the business team needs to provide APIs that combine data from multiple sources. They want to explore ideas for these new APIs without the cost and time implications of a full IT project. They do this by using integration specialists' APIs that already exist for the data that they need. This section shows how IBM App Connect Designer enables non-integration specialists to implement hybrid APIs that are based on a composite of existing on-premises and SaaS-based APIs.

6.9.1 Business scenario

The business team are exploring new innovative API possibilities that would combine existing APIs with data from the SaaS applications they use. They want to be able to prototype these without the need for an integration specialist from the central team.

Using IBM App Connect Designer, you can easily create cloud managed APIs that enable you to call out to an on-premises API, and aggregate data between enterprise systems and SaaS applications.

We will continue to use the store warehouse for our business scenario. Currently, the store has its stock inventory and CRM systems on-premises, exposing the data via API. They have recently adopted ServiceNow SaaS for incident management to improve customer experience. ServiceNow is a cloud-based platform that supports service management for all departments of your business including IT, human resources, facilities, field service, and more.

The business team would like to automate the initiation of incident tracking in ServiceNow whenever a stock product is under recall due to a fault. By utilizing an existing on premises API that provides a customer list with the product installed, the business team will aggregate the data using an IBM App Connect Designer API flow to generate new incidents into its ServiceNow application. This hybrid composition will be exposed as an API such that it can be initiated by other applications. See Figure 6-261 on page 346.

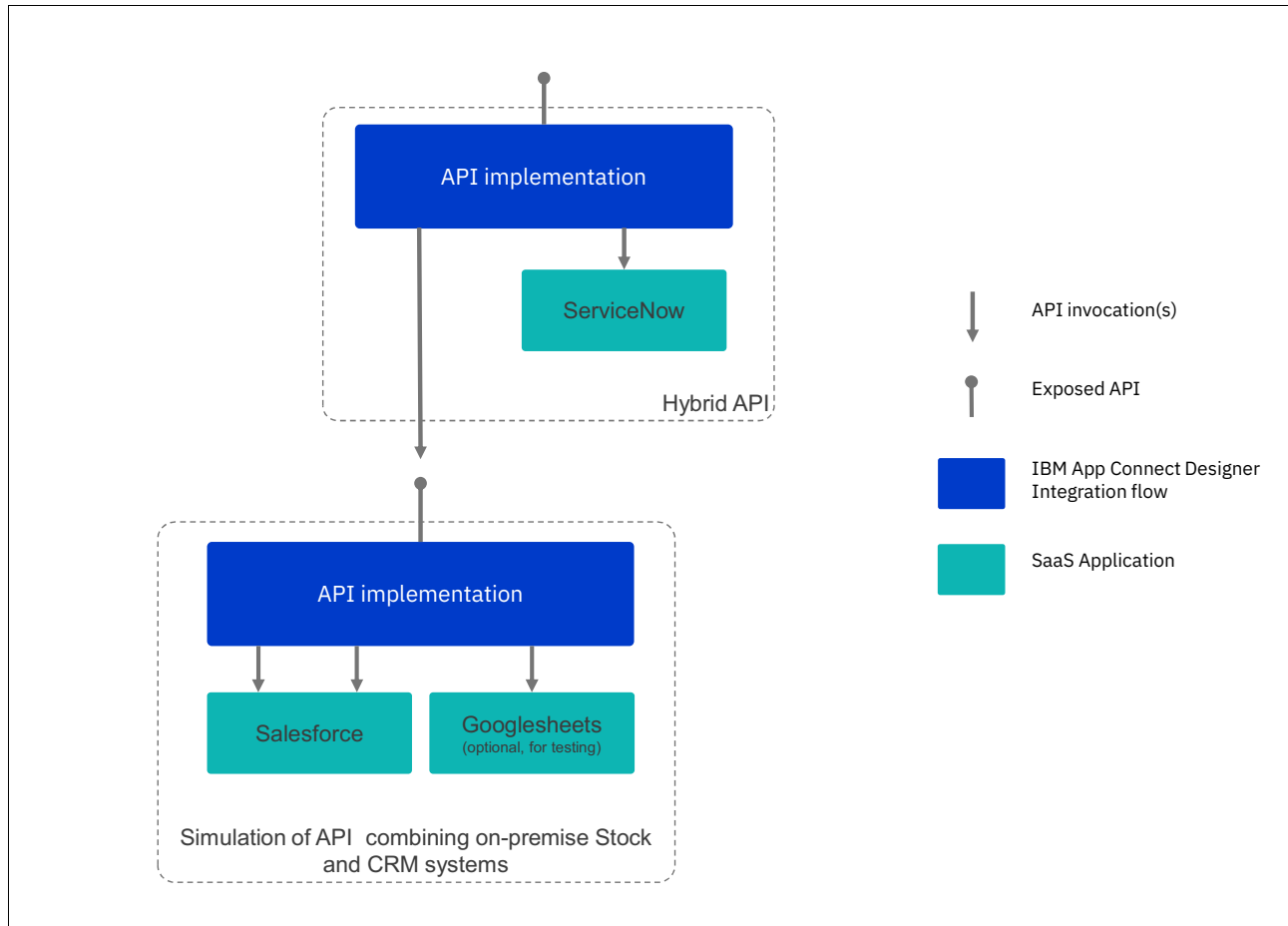


Figure 6-261 Perform event-driven SaaS integration

6.9.2 Invoking existing APIs from IBM App Connect Designer

This scenario invokes downstream systems via APIs. IBM App Connect enables invocation of existing APIs in a number of ways.

- ▶ Invoking imported API definitions for external APIs
- ▶ Invoking operations known to and managed by IBM App Connect
- ▶ Using raw HTTP connectivity

Note: To use an API in a flow, you must first connect IBM App Connect to the API by using the security scheme that is configured for that API. This will be explained in 6.8.4, “Create flows” on page 331.

Using APIs imported from OpenAPI documents

You can import OpenAPI documents that contain API definitions into IBM App Connect. Then you can call the API from a flow. The OpenAPI Specification previously known as Swagger document, is a definition format that is written in JSON or YAML for describing REST APIs. You describe the API details such as available endpoints and operations, authentication

methods, and other information in this document. Each imported document is added as an API to the IBM App Connect API catalog,

Tip: Refer to this blog for a step-by-step guide to import OpenAPI into IBM App Connect Designer:

https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-openapi/#openapi_add-app.

Using shared APIs that are managed natively in IBM Cloud

You can access to a set of APIs that are shared within your IBM Cloud organization. The shared APIs are available in the IBM App Connect catalog of applications and APIs. These shared APIs are managed natively in IBM Cloud by the API management solution IBM API Connect. Shared APIs can come from many sources, including those that are created using integrated IBM Cloud services such as the IBM App Connect service, APIs that are exposed by Cloud Foundry applications, and APIs that incorporate OpenWhisk actions that are created using Cloud Functions.

Tip: This blog provides details on How to use IBM App Connect with shared APIs in IBM Cloud:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-sharedapis-in-ibm-cloud/>.

Using the invoke method for the HTTP application

Another way to invoke an API from the flow is to use the **HTTP Invoke Method** node under the Applications tab. You can use IBM App Connect to pass key data from an app into an HTTP “invoke” action that calls out to an HTTP endpoint, and then pass data that is returned from the HTTP response into other apps in the flow.

There are network and security setups that you need to consider before you use the HTTP invoke method, for accessing API end points that resides in a private network. You should read and follow the detail setup instructions as described in the blog: *How to use IBM App Connect with HTTP* before your flow design.

Tip: You can find the blog “*How to use IBM App Connect with HTTP*” here:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-http/>.

6.9.3 Solution overview

The completed scenario flows consist of two separate IBM App Connect flows, namely Recall List and Hybrid API, as shown in the following figures. The on-premises API called RecallList will be simulated using IBM App Connect Designer API flow. The second IBM App Connect Designer API called Hybrid API will create ServiceNow incidents for all the recalled customers that are retrieved as response from RecallList.

You can easily modify the simulated flow to reflect the actual on-premises API that your organization has. You must register and share the API (in this case, RecallList) in IBM App Connect Designer Catalog and trigger it in the Hybrid API flow. Steps for doing this have been described in section 6.9.2, “Invoking existing APIs from IBM App Connect Designer” on page 346.

Figure 6-262 shows the simulated on-premises API Recall List.

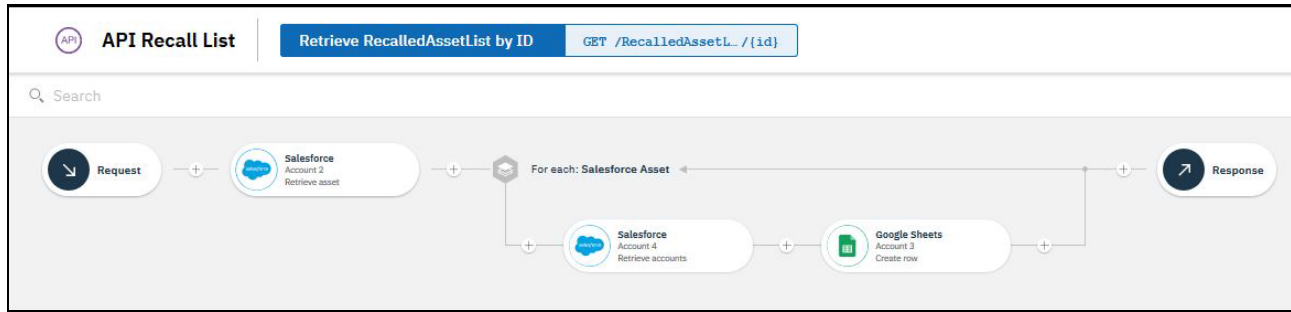


Figure 6-262 Recall List API

Figure 6-263 on page 348 shows the Hybrid API.

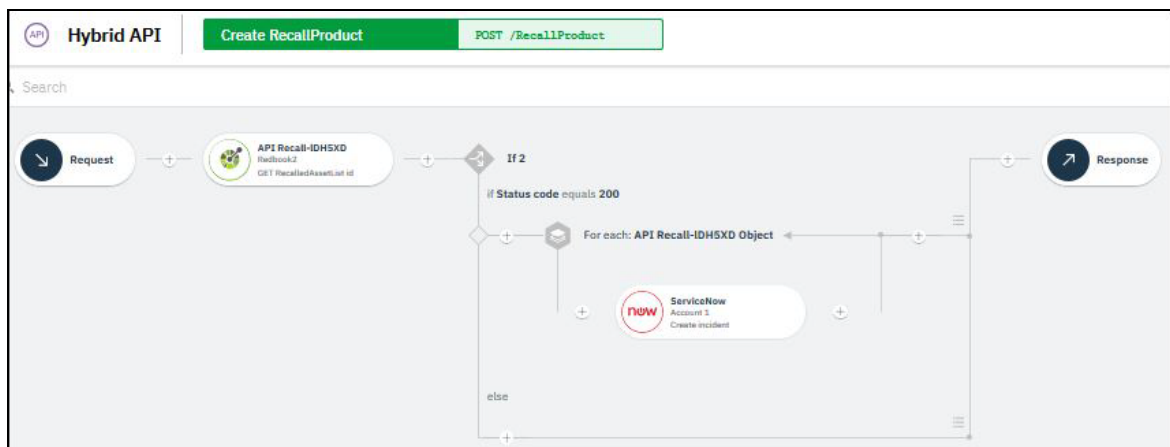


Figure 6-263 Hybrid API

6.9.4 Preparing the external SaaS applications

This tutorial assumes that you have signed up for free or trial accounts for Salesforce developer, GoogleSheet and ServiceNow accounts or that you have business accounts. In addition, you have already registered for a free IBM Cloud user ID, where you can access a full catalog of IBM Cloud solutions, including IBM App Connect Designer and API Connect, which will be used in this tutorial.

Additional references:

- ▶ How to use IBM App Connect with Salesforce:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-salesforce/>

- ▶ How to use IBM App Connect with ServiceNow:

<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-servicenow/>

This scenario requires Salesforce Accounts and Asset object to be created in the Salesforce system. A Salesforce asset represents customer purchased or installed products. The

on-premises API Recall List will access Salesforce system to retrieve all the customers based on a recall product ID, and generate this list of recalled customers.

Note: To understand Salesforce Account and Asset object relationship refer to the following document:

https://trailhead.salesforce.com/en/content/learn/modules/field_service_maint/field_service_maint_assets.

In the second flow, called Hybrid API, you should have a ServiceNow account with a running instance to view the new incidents that are created for the recalled customers. If you are not familiar with this, refer to section 6.8.2, “IBM App Connect event-driven flow to Salesforce, Google and Slack SaaS applications” on page 330 for more details on IBM App Connect integrate with ServiceNow.

6.9.5 Create simulated on-premises API flow

Perform the following steps to create the flow:

1. Log on to IBM App Connect Designer.
 - a. On the Dashboard in IBM App Connect Designer, click **New** → **Flows** for an API.
 - b. Enter a name that identifies the purpose of your flow, for example APIRecallList.
 - c. Create the model named RecallAssetList. This defines the object you are working with; in this case, we are generating a list of customer records.
 - d. On the Create Model panel, there are two tabs: a Properties tab and an Operations tab. Properties are required to define the structure of the object that the API will work with. Use product name as **ID**, fill in all the properties as shown in Figure 6-264 on page 349.

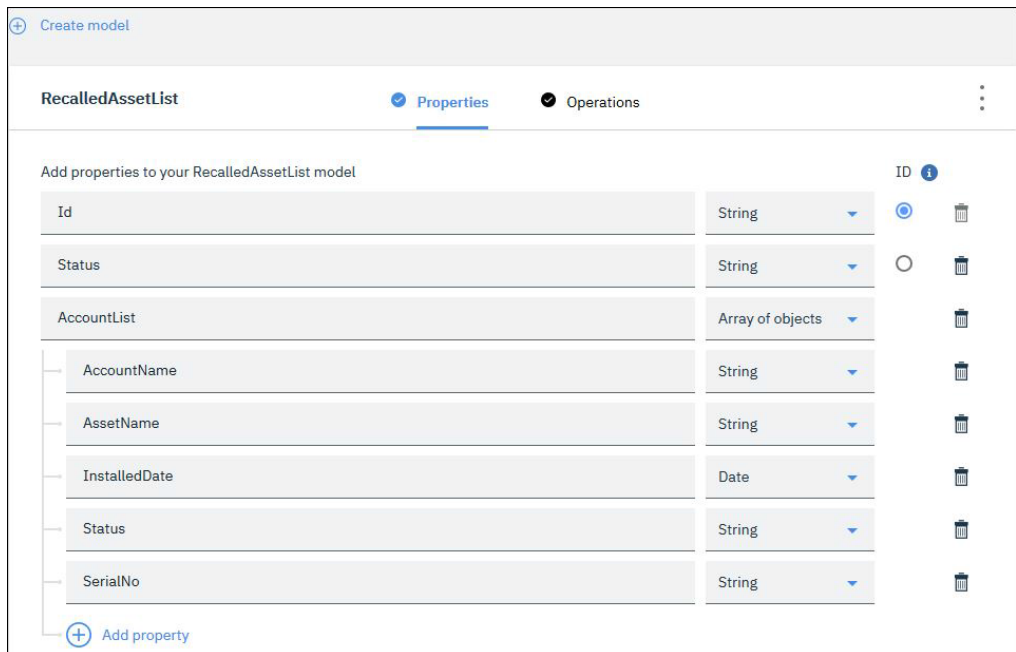


Figure 6-264 Properties defined for RecallAssetList

- e. The Array of Objects called Account List, which consists of AccountName, AssetName, InstalledDate, Status and SerialNo, represents the API Response.

Note: A property that is set as the ID might indicate that your flow must return this property when creating an object. Or it might indicate that the property must be sent in a request to update or retrieve an object by using its ID. You can use ID against only one property.

2. To define how the API will interact with the objects, click **Operations**. You can add operations to:
 - ▶ Create an object.
 - ▶ Retrieve an object by using its unique ID or by using a filter.
 - ▶ Update or create an object (by using its ID or a filter), where the object is updated if it exists, or created if it doesn't.
3. Define **GET** as the Operation of the request and click within the Select an operation to add drop-down list, and then select **Retrieve RecalledAssetList by ID**. The **GET/RecalledAssetList/{id}** will be automatically generated.
4. Click **Implement flow** to implement the API operations.

Figure 6-265 shows the APIRecallList operation.

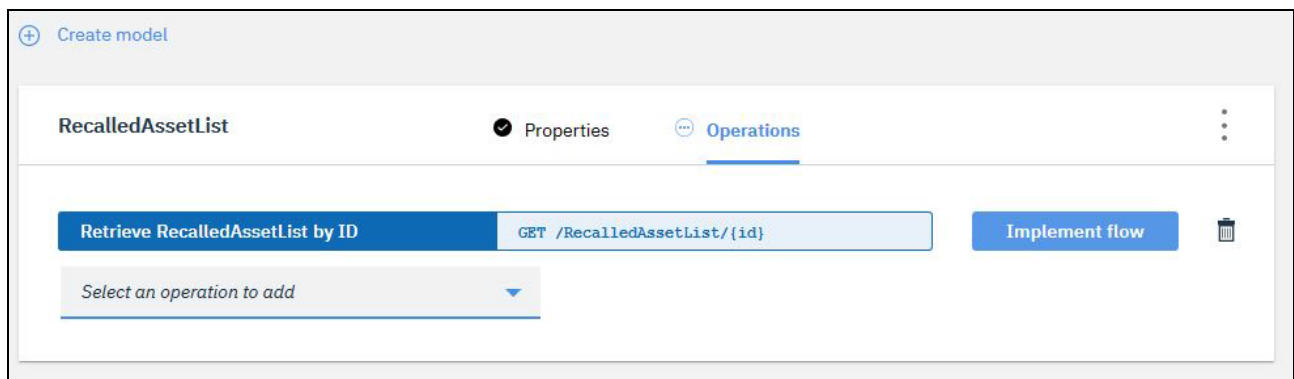


Figure 6-265 Retrieve RecalledAssetlist by ID

5. You will see a basic flow in the flow editor, with a Request node, a Response node, and a space to add one or more target applications. We want to retrieve the list of customers that have purchased the recalled product, from the **Salesforce Retrieve Asset** object. This will return a list of **Account ID** that we will use to retrieve customer records next.
6. Retrieve all the Account ID records that have the recalled product:
 - a. If you haven't already connected IBM App Connect to Salesforce, specify the name and password of your Salesforce account. Click (+), Select **Salesforce** → **Assets** → **Retrieve assets**. Click **Add a Condition** to specify the selection condition. Under the Where* clause choose **Asset Name** from the drop-down list, and click **Insert a Reference** on the adjacent field, and select **ID** (this is the request ID field that we have defined in the properties).
 - b. Click **Add Condition** to add **AND** clause. Select **Status** from the drop-down list, and type **Installed** on the adjacent field.

Figure 6-266 on page 351 shows the Retrieve Asset condition.

The screenshot shows the configuration for the 'Retrieve asset' action in a Salesforce flow. At the top, it identifies the context as 'Salesforce Account 2'. The main configuration area is titled '*Where' and contains two filter conditions:

- Condition 1: 'Asset Name' (field) equals 'Id' (value).
- Condition 2: 'Status' (field) equals 'Installed' (value).

Below the conditions, there is an 'Add condition' button. Further down, the 'Maximum number of items to retrieve' is set to 10. Underneath this, there are two sections for handling edge cases:

- If the limit is exceeded:**
 - Exit the flow with an error
 - Process 10 items from the collection
- If no items are found:**
 - Continue the flow and issue a '204: No content' status code
 - Exit the flow with a '404' error code

Figure 6-266 Retrieve assets that are under recalled

Leave the defaults for the rest of the fields on the form.

7. Using the Asset List that was generated, our next action is to build the list of customers that own these assets from **Salesforce Retrieve Accounts** object using the Account Name. (This field is returned as one of the data from the Asset List.)
8. Add the **For each loop** under the Toolbox option as the next action, **For each:Salesforce Asset**:
 - a. On the Input tab, provide a display name to the For each loop, use Salesforce Asset.
 - i. The collection of items to process is called Assets.
 - ii. Accept the default. Process all the items sequentially.
 - iii. Choose **Process** all other items and continue the flow.

Figure 6-267 on page 352 shows the For Each input construct.

For each: Salesforce Asset

Input **Output**

*Select the collection of items to process:

Assets

Display name: \$SalesforceRetrieveasset

For each: Salesforce Asset

Collection processing options:

Process all items in the collection sequentially

Process items in parallel in any order (optimized for best performance)

If an error occurs while processing an item:

Process all other items and continue the flow

Exit the flow with an error

Figure 6-267 Process Asset List

- b. Click (+) and select **Salesforce** → **Accounts** → **Retrieve Accounts** as the next action inside the for loop,
 - i. Click **Add a condition** for our selection of accounts, in the Where* clause choose Account Id from the drop-down list, and click **Insert a Reference** on the adjacent field, and select **Foreach Salesforce Asset/Asset/Account Id**.
 - ii. We want to process 10 items in this tutorial, hence click **Exit** the flow with an error, if the maximum is exceeded. (For bulk data processing, you should be using Scheduler jobs).
 - iii. If no item is found, exit the flow with the '404' error code.

Figure 6-268 on page 353 shows the selection condition.

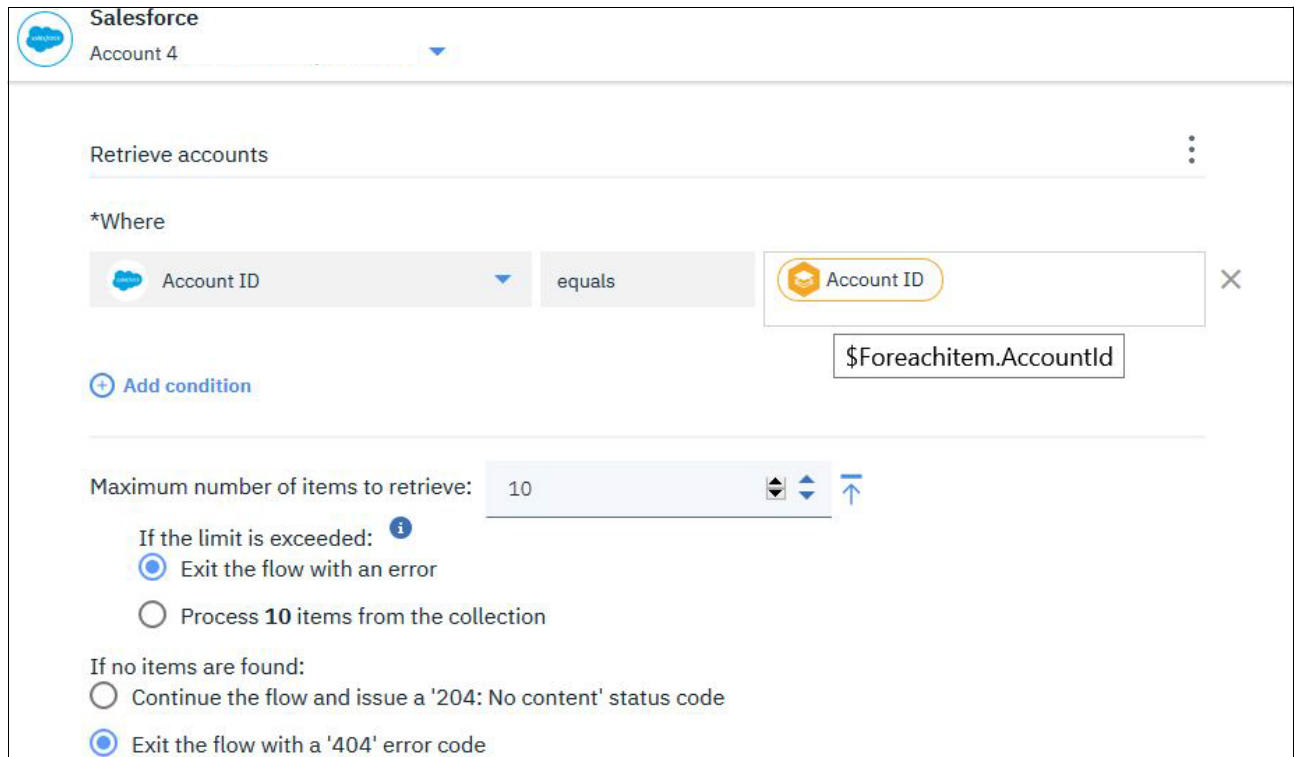


Figure 6-268 Retrieve Customer Account Details

- c. Next, we define the Output mappings of the customer information we required for the API response as follows:
 - i. Click the **Output** tab to capture the customer records into an array
 - ii. On the Edit Properties panel, perform the mappings by clicking **Insert Reference** for the following fields and choose the respective properties as shown here.
 - Account Name: **SalesforceRetrieveAccount.Name**
 - Asset Name: **Request URL parameters / Object/ id**
 - Installed Date: **For each: Salesforce Asset / Asset / Install Date**
 - Status: **For each: Salesforce Asset / Asset / Status**
 - Serial Number: **For each: Salesforce Asset / Asset / Serial Number**
 - iii. When the final item has been processed, the complete output made available is an array of the mapped output objects we defined above. The completed mapping is shown in Figure 6-269 on page 354.

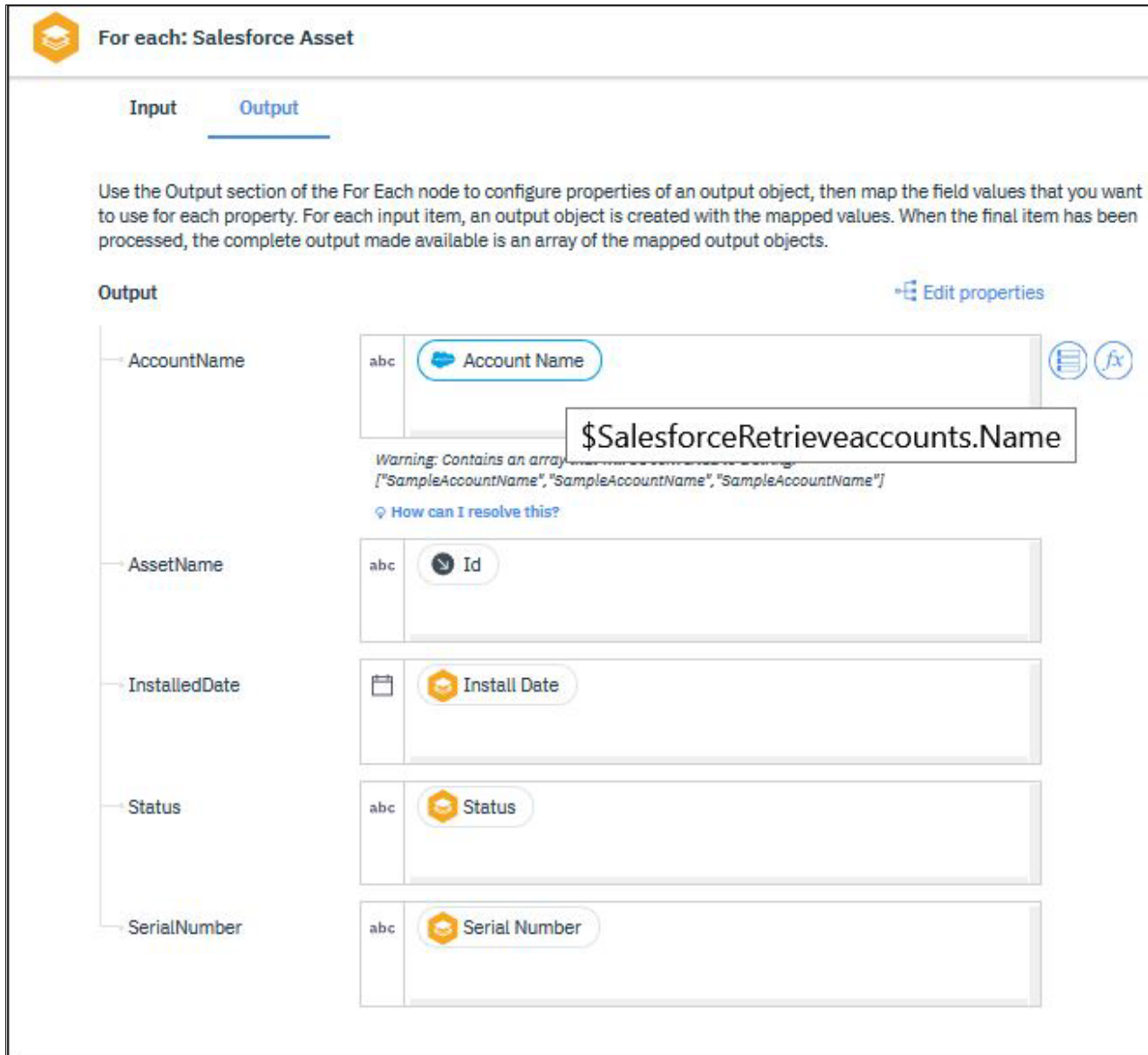


Figure 6-269 Capture the customer information into an array

9. The next step is generating the customer list to Google Sheet and this is optional. For testing purposes, you can remove this action after validating the customer data and verified that the flow runs successfully.
 - a. Click (+) and select **Generate Google Sheet Create** row to capture the orders as the next action. If you have not already connected a Google Sheets account, click **Connect to Google Sheets** and follow the instructions to allow IBM App Connect to connect to your Google Sheets account.
 - b. Select the **Google Sheets** spreadsheet (and then the worksheet, which is called Recalled Customer List in this tutorial) that you configured with the column headings: Account Name, Asset Name, Installed Date, Status, Serial Number.
 - c. Map the spreadsheet columns by clicking **Insert Reference** for each of the following fields and choosing the property that is listed here:
 - Account Name: **SalesforceRetreiveAccount.Name**
 - Asset Name: **Request URL parameters / Object / id**

- Installed Date: **For each: Salesforce Asset / Asset / Install Date**
- Status: **For each: Salesforce Asset / Asset / Status**
- Serial Number: **For each: Salesforce Asset / Asset / Serial Number**

Figure 6-270 shows the Google Sheet column mappings.

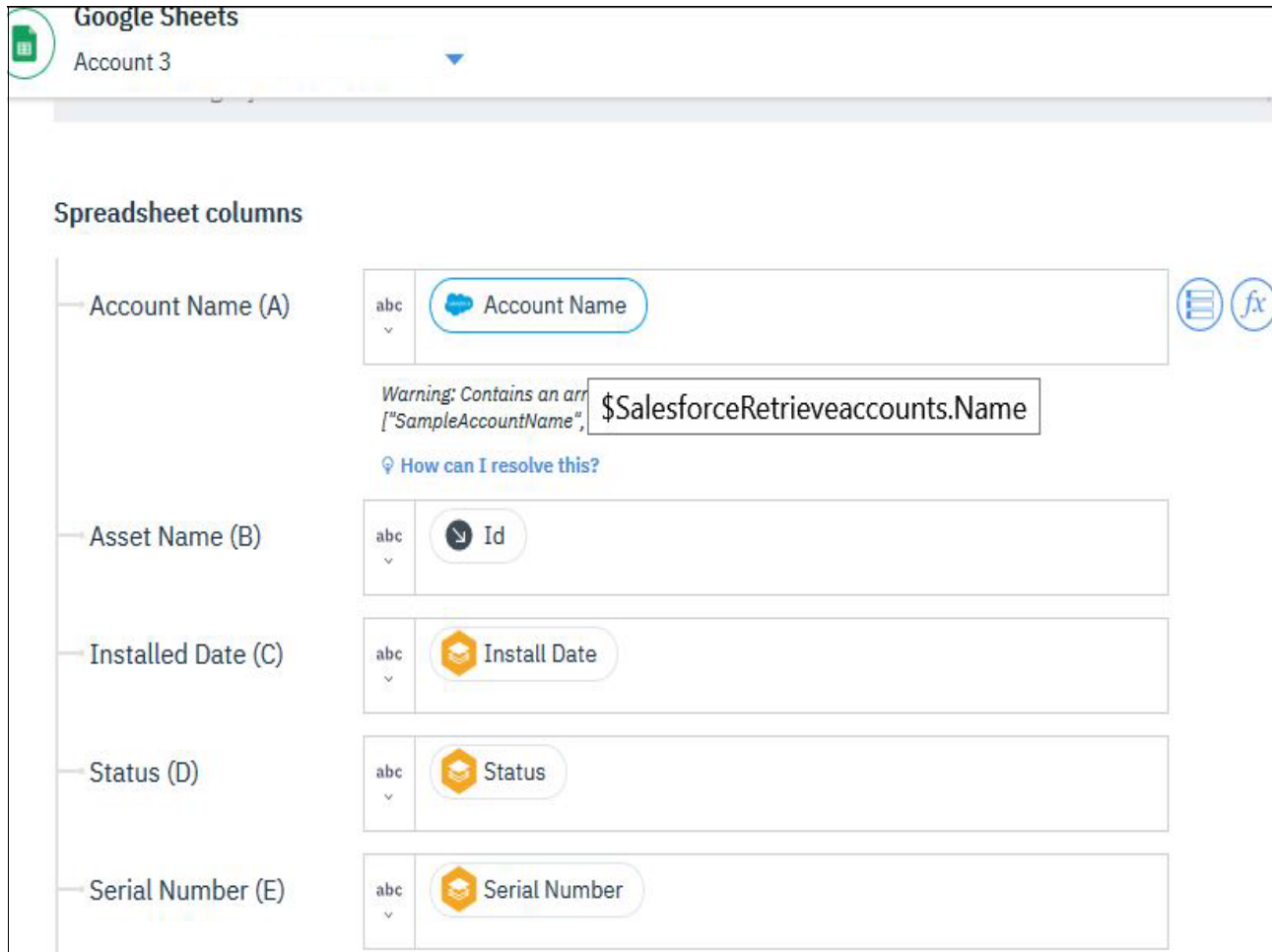


Figure 6-270 Google Sheet with list of Recalled Customers

10. Finish the flow by mapping the API response body using the array generated from the For each loop:
 - a. Click the **Response node** in the flow, and map the output fields as follows by clicking **Insert reference**:
 - ID: **Request URL parameters / Object / id**
 - Status: **Type Report generated for <ID>**
 - Account List: **For each: Salesforce Asset Output / Array**
 - Account Name: **Parent mapping item: Array / Array / Output/ AccountName**
 - Asset Name: **Parent mapping item: Array / Array / Output/ Asset Name**
 - Installed Date: **Parent mapping item: Array / Array / Output/ Install Date**
 - Status: **Parent mapping item: Array / Array / Output/ Status**
 - Serial Number: **Parent mapping item: Array / Array/ Output/ Serial Number**

Note: In the Response header section, you can choose your own response code mapping. The following response codes are returned for the different operations:

- ▶ Create operations return a response code of 201 (record created).
- ▶ Retrieve operations return a response code of 200 (record retrieved).

Replace or create operations return a response code of 200 (record replaced) or 201 (record created).

Figure 6-271 on page 356 shows the API response mappings.

The screenshot displays a 'Response' configuration window. At the top, there is a header with a right-pointing arrow icon and the text 'Response'. Below this, the configuration is organized into several sections:

- *Status Code:** A field with a dropdown menu showing '123' and a text input field containing '200'.
- Response body:** A section containing several field mappings:
 - Id:** A dropdown menu showing 'abc' and a button labeled 'Id' with a right-pointing arrow icon.
 - Status:** A dropdown menu showing 'abc' and a text input field containing 'Report is generated for' followed by a button labeled 'Id' with a right-pointing arrow icon.
 - AccountList:** A button labeled 'Array' with a right-pointing arrow icon.
 - AccountName:** A dropdown menu showing 'abc' and a button labeled 'AccountName' with a right-pointing arrow icon.
 - AssetName:** A dropdown menu showing 'abc' and a button labeled 'AssetName' with a right-pointing arrow icon.
 - InstalledDate:** A date picker icon and a button labeled 'InstalledDate' with a right-pointing arrow icon.
 - Status:** A dropdown menu showing 'abc' and a button labeled 'Status' with a right-pointing arrow icon.
 - SerialNo:** A dropdown menu showing 'abc' and a button labeled 'SerialNumber' with a right-pointing arrow icon.

Figure 6-271 Response data for the API RecallList

- b. You have completed the on-prem simulated API. Click **Done** to return to your model.
- c. From the options menu (:), click **Start API**.
- d. We will perform an integrated test upon completion of the Hybrid API.

6.9.6 Create Hybrid API

Back to our business scenario, the business team wants to be productive quickly by reusing an existing on-premises API, and build a new API. The hybrid API will help the team to automate the creation of customer cases in their newly adopted SaaS application, ServiceNow, aggregating data from the existing API. The ServiceNow incidents will be created for all the affected customers who have purchased and installed the product under recalled.

Adopting the hybrid AP will help the team to monitor and provide better customer support with all the information at one place.

In the following steps, we will create new ServiceNow incidents for all the customers that are in the recall list (data from existing API). Using IBM App Connect Designer, this can be achieved in three simple processes:

1. Invoke RecallList API.
2. For each customer record:
 - a. Create a ServiceNow incident.
 - b. Capture a new incident number.
 - c. Complete until all records are processed.
3. Map API Response data to return new incidents created in ServiceNow.

Create the flow

Perform the following steps to create the flow:

1. Click **New flows for an API** and choose first action to be **API** instead of Application.
 - a. On the Dashboard in IBM App Connect Designer, click **New** → **Flows** for an API.
 - b. Enter a name that identifies the purpose of your flow, called it **RecallProduct**.
 - c. Create the model property as **RecallProductName**, this defines the object that you are working with. In this case, we are using this ID to retrieve the list of recalled customer records and create a corresponding ServiceNow incident for each customer.
2. Next, click the **Operation** tab to define the API operation. Select **CreateRecallProduct** from the drop-down list. The POST/RecallProduct will be automatically generated. Click **Implement flow** to continue. See Figure 6-272 for details.

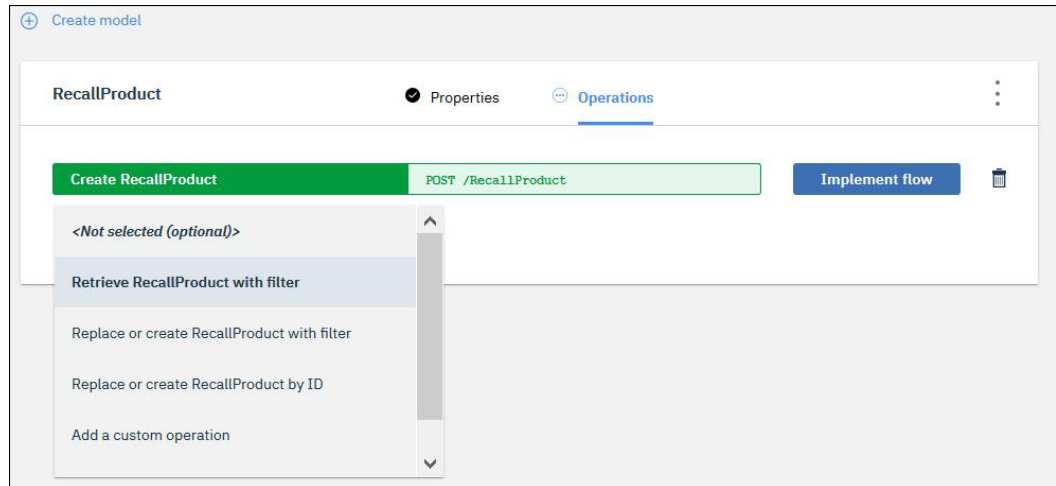


Figure 6-272 RecallProduct Operation

3. To add the target application to the flow, click (+) and choose API tab. You will choose the RecallList API (which you created as our simulated on-premises API) and select **GET / RecalledAssetList /{ id}** to continue.

Figure 6-273 shows the Add the API RecallList as the first action.

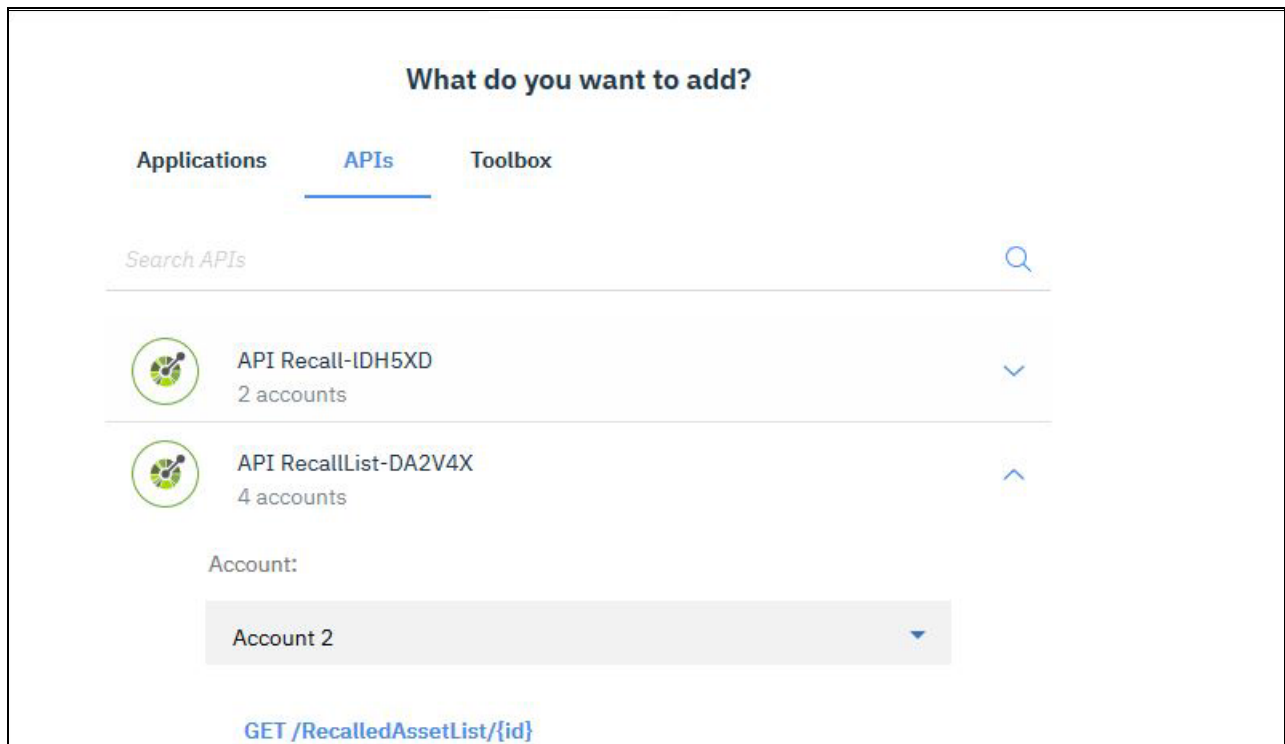


Figure 6-273 Select APIRecall List

4. The input to the flow will be a string that contains the Recall product name. We define it in the ID field by clicking the **Insert Reference** button and selecting **Request body parameters / Object / Object / RecallProductName**. See Figure 6-274.

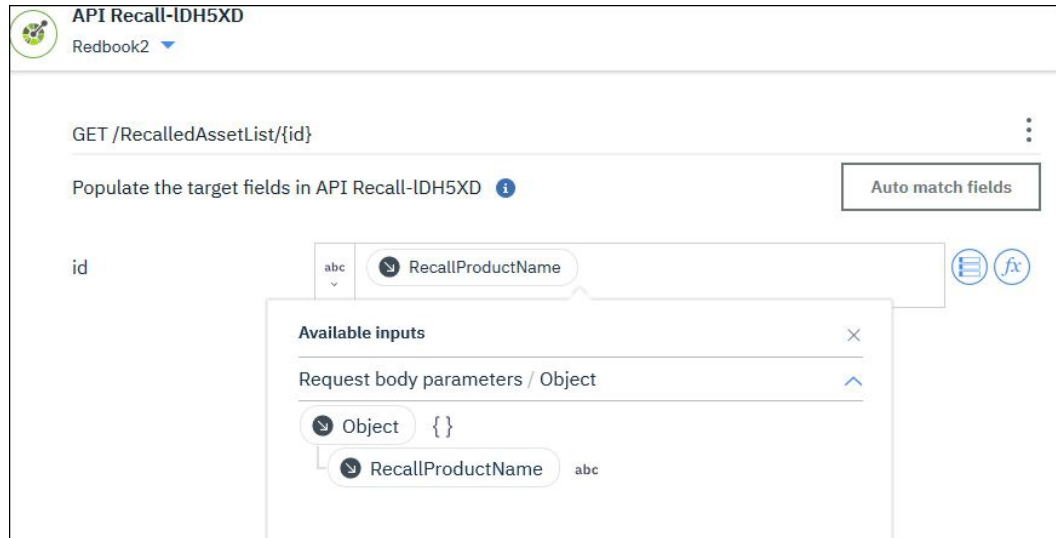


Figure 6-274 Define the input field for invoking API Recall List

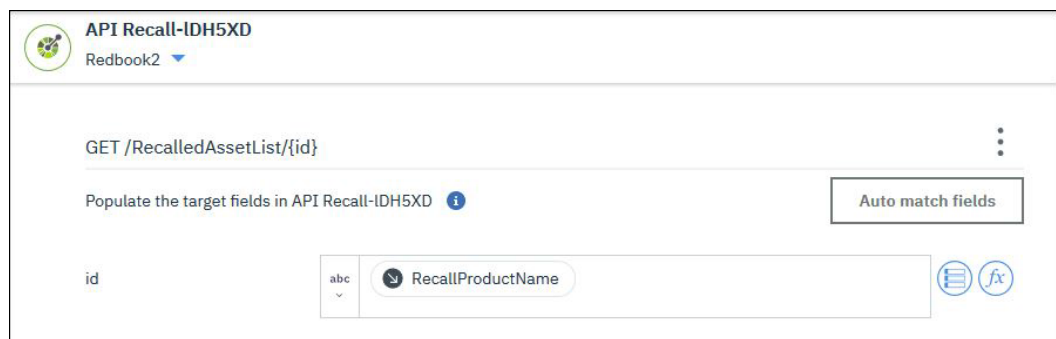


Figure 6-275 Invoke RecallList API using product name

5. Add an If condition under the toolbox to check that the response status is successful as the next action.

Note: In order to pass data from a nested “If / For loop” to the rest of the flow, we need to define the output schema properties, and map the output data. In our scenario, we want to capture the list of new ServiceNow incidents created for all the recall customers and compose it as our API response. We will perform this step after we complete the For loop, when the list of customer ServiceNow incidents has been generated and made available for data mapping.

6. Next, we process the list of recall customers using the For each loop. And for each customer we want to create a new ServiceNow incident, and capture the created incident number into an array called **IncidentCreated**. Figure 6-276 on page 360 shows the flow design.

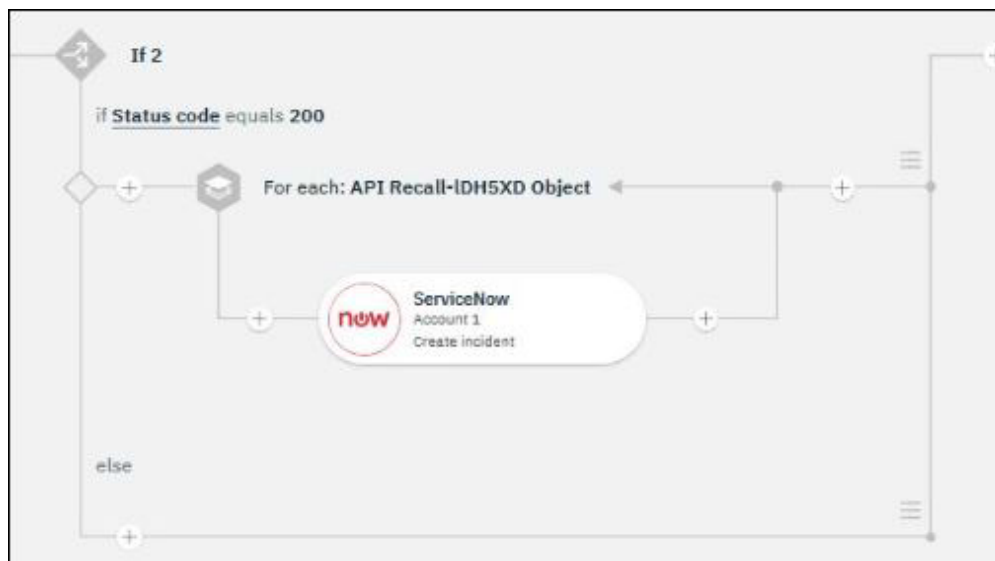


Figure 6-276 Create ServiceNow Incidents

7. Choose For each under the Toolbox option as the next action. Use the **AccountList** (retrieved from RecallAPIList) object as the **Input** → **Select** the collection of items to process: as the **Input** to the For each loop. Leave the defaults for the rest of the fields.

Figure 6-277 shows the process all the Recall Records window.

The screenshot shows the configuration for the 'For each: API Recall-IDH5XD Object' action. The 'Input' tab is selected. Under the heading '*Select the collection of items to process:', the 'AccountList' item is chosen. A dropdown menu is open, showing the following text: 'API Recall-IDH5XD / GET RecalledAssetList id / RecalledAssetList_findById_model / response / 200: Request was successful / AccountList'. Below this, there are options to 'Apply a function' and 'Edit expression'. At the bottom, there are two radio button options: 'Process items in parallel in any order (optimized for best performance)' (which is unselected) and 'Process all other items and continue the flow' (which is selected). There is also an option 'Exit the flow with an error' which is unselected.

Figure 6-277 For each Input property

8. For each retrieved customer record, we want to create a new Incident case in ServiceNow application. Choose **ServiceNow create incident** as the next application in the For each loop. If you have not already connected to your ServiceNow instance, choose the account and click **Connect**. After it is connected, the Populate the target fields in ServiceNow screen will display, ready for you to create the incident record details.
9. Populate the new incident record with the information we retrieved from the RecallAPI List array, we will populate only a subset of the ServiceNow fields, as follows:
 - Caller: Type Hybrid API
 - Category: Type Hardware
 - SubCategory: Click within the field and then click Insert a reference button, From the list, expand **Request body parameters** → **Object** and select **RecallProductName**.
 - Scroll down to the field called Short Description. and type Product Serial Number: and click the **Insert Reference** button to select the object **Foreach API recall object / Object {}**. This object gives us the collection of the record information, but we want to extract only the Serial No.
 - To do the extraction, click the **object {}** on the target field. This will change the object display to **{{Foreach2item}}**, and choose **Edit Expression**. Append **Serial No** to the end of the field, the final expression will look like **{{Foreach2item.SerialNo}}**.
 - WatchList: Type Ticket created for: and repeat the previous two steps to extract **Account Name** from the **Foreach API recall object / Object {}**. The final expression will be **{{Foreach2item.AccountName}}**.

Figure 6-278 shows the ServiceNow field mappings for Called / Category / SubCategory.

The screenshot shows the 'Create incident' form in ServiceNow. At the top, there is a 'ServiceNow' header with the 'now' logo and 'Account 6' selected. Below the header, there is a 'Create incident' title and a 'Populate the target fields in ServiceNow' section with an 'Auto match fields' button. A search bar labeled 'Filter the target fields' is present. The form contains several fields with dropdown menus and reference buttons:

- Number:** A dropdown menu with 'abc' selected and a reference button.
- Opened at:** A dropdown menu with 'abc' selected.
- Caller:** A dropdown menu with 'abc' selected and the text 'Hybrid API' displayed.
- Category:** A dropdown menu with 'abc' selected and the text 'Hardware' displayed.
- Subcategory:** A dropdown menu with 'abc' selected and a reference button labeled 'RecallProductName'.

Figure 6-278 ServiceNow Incident detail 1 of 2 display fields

Figure 6-279 shows the ServiceNow field mappings for Short Description and Watch List.

*Short description	abc	Recalled Proudct SerialNo: {{{ \$Foreach2item.SerialNo }}}}
Watch list	abc	Ticket created for: {{{ \$Foreach2item.AccountName }}}}

Figure 6-279 ServiceNow Incident detail 2 of 2 display fields

10. Now, we are ready to define the Output Schema of the For loop and If condition to capture the new ServiceNow incidents generated, to include in our API response.

- a. Go back and click the For construct, and select the **Output** tab, click **Add a property** to specify the structure of the data as it will appear after the 'For each' node. The field name will be IncidentCreated and its type is String.
- b. After creating the structure we choose 'Edit mappings' to select the data we want to appear in our new output collection. Here we use the ID from ServiceNow to populate the **IncidentCreated** field. See Figure 6-280 on page 362 for the field details.

The screenshot shows the configuration for a 'For each: API Recall-IDH5XD Object' node. The 'Output' tab is active, displaying a property named 'IncidentsCreated' with a value of 'new Incident ID'. A context menu is open over the value, showing options like 'ServiceNow / Create incident / incident / Incident ID', 'Apply a function', 'Convert to plain text', and 'Edit expression'. The interface also includes an 'Edit properties' button and a note about using the Output section to configure properties of an output object.

Figure 6-280 Mappings of the IncidentCreated field

- i. Next we will complete the output schema and data for the If condition. Go back to the If construct, and click **Output Schema**, type IncidentsCreated in the property field (click **Add a property** if it is not already there).
- i. Click on the **Output data**, you will see the IncidentCreated label that is created for you, and ready for data mappings. Click on the **insert a reference button**, select **For each: API Recall-IDH5XD Object Output / Array / Output / IncidentsCreated**, this is the array of all the ServiceNow incidents that are generated as output data from the For loop.

See Figure 6-281 on page 363 for the field mapping details.

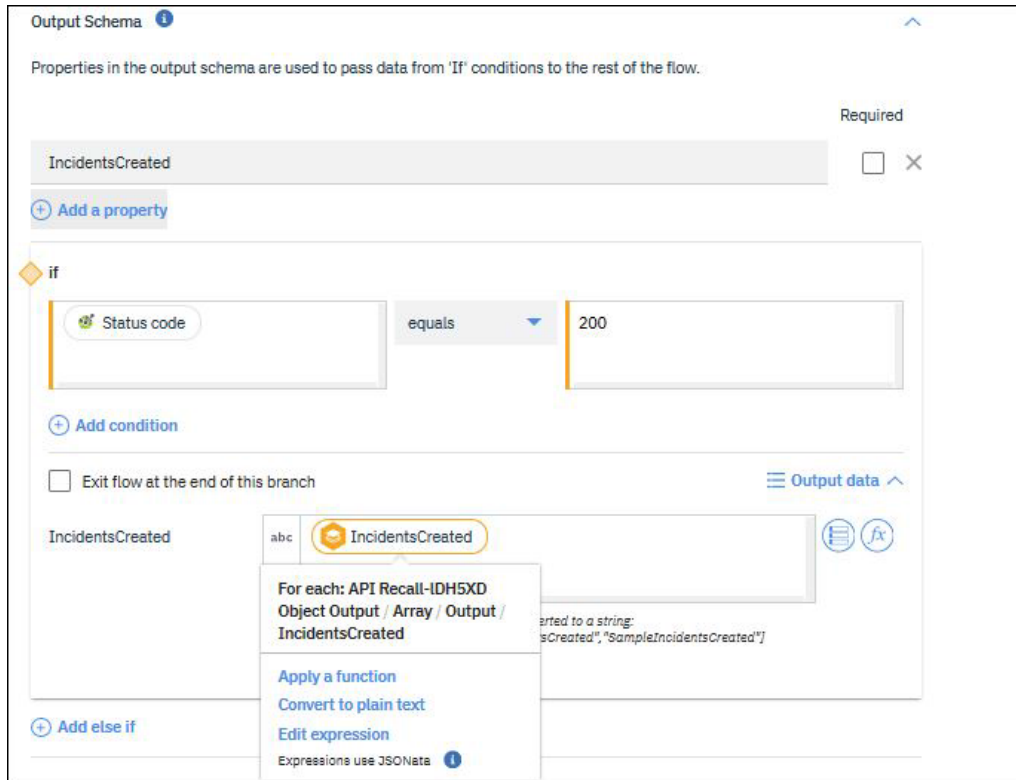


Figure 6-281 If condition Output Schema and Output Data mappings

Tip: You can create additional field capture in the Output Schema and Output Data in the For loop or If condition. To learn more, refer to the following document:

<https://developer.ibm.com/integration/blog/2018/05/15/process-data-node-ibm-app-connect/>

In addition, you can apply JSONATA function to transform your data to the desired format including concatenation of different fields, date formats, string conversion:

<https://developer.ibm.com/integration/docs/app-connect/creating-managing-event-driven-flows/completing-fields-action/applying-jsonata-functions/>

11. In our final step, we map the API Response data to return new incidents created in ServiceNow, using the array we created in the preceding step.
 - a. Click **Response node**. Map the Response Header and Body as shown in Figure 6-282 on page 364.

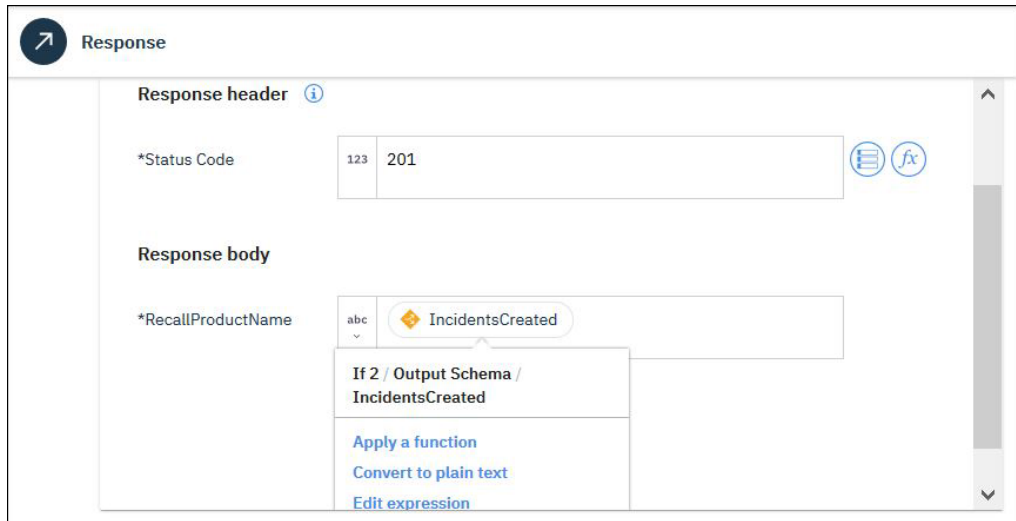


Figure 6-282 Hybrid API response

6.9.7 Test the flows

There are two APIs that we need to perform integrated testing. These are 1) an existing on-premises API to generate the recall list and 2) a Hybrid API that invokes the on-premises API to create ServiceNow for all the recall customers.

You might be using an existing on-premises API instead of our simulated on-premises API, which we built in the previous steps. In that case, you should have already gotten familiar with the section 6.9.2, “Invoking existing APIs from IBM App Connect Designer” on page 346, which explains the different methods for invoking the API in IBM App Connect. In our testing, we will use the API Sharing Outside of Cloud Foundry Organization method inside the built-in API portal.

This tutorial assumed that there are Salesforce Accounts with linked Assets are already existed in the Salesforce system. This represents the products that the customer has purchased and installed. If it does not exist, follow the steps in 6.8, “Perform event-driven SaaS integration ” on page 328 to understand the steps involved in setting this up.

6.9.8 First, test the simulated on-premises API

Follow the steps described here for testing of the simulated on-premises API Recall List:

1. On the dashboard, click the **API Recall List**. After it is loaded, click the **Menu** at the upper right, and click **Start API**.
2. Click the **Manage** tab and scroll down to the **Sharing Outside of Cloud Foundry organization** section (in the lower part of the page).
3. Click **Create** API key, give the key a descriptive name; for example: **RecallAssetList**, and then click **Create**.
4. Click the **API Portal** link. This opens the API in an API portal window with the API request and response information, plus some response data.
5. To invoke your API in the API portal, click **Try it**.

- Using the product code as the recalled product (Platinum in our tutorial) enter it on the Model id field under Parameters/Id* and click **Call operation**. Figure 6-283 on page 365 shows the API Call operation.

The screenshot shows an API client interface for a GET request. The URL is `https://service.au.apiconnect.ibmcloud.com/gws/apigateway/api/861023f9aa3f7d3042b766ebb22382d483635504daa5d20b7c04ddf403d2285e/aSykwM/RecalledAssetList/{id}`. There are links for "Examples" and "Try it". Under "Type Headers", the "Accept" header is set to "application/json". Under "Parameters", the "id *" parameter is set to "Platinum". A "Call operation" button is located at the bottom right.

Figure 6-283 Using product ID Platinum to invoke the Recall API

- This will invoke the API flow. You will receive the code 200 OK as a successful response, with a list of customers with production information and a status-completed statement generated by the flow: "Id": "Platinum", "Status": "Report is generated for Platinum" below the response.

Figure 6-284 on page 366 shows partial response details and status from RecallList API.

```

{
  "AccountName": "Adeline Chun",
  "AssetName": "Platinum",
  "InstalledDate": "2019-07-09",
  "SerialNo": "A123",
  "Status": "Installed"
},
{
  "AccountName": "RedBook",
  "AssetName": "Platinum",
  "InstalledDate": "",
  "SerialNo": "T12345",
  "Status": "Installed"
}
],
"Id": "Platinum",
"Status": "Report is generated for Platinum"
}

```

Figure 6-284 Recall customer and product information

8. If you have incorporated the Google Sheet in your flow to be used for testing purpose, you will see the following rows generated as output. **Note:** This information will be exactly the same as our API response data, which is displayed on the API portal.

Figure 6-285 on page 366 shows customer and product information on Google Sheet.

	A	B	C	D	E
1	Account Name	Asset Name	Installed Date	Status	Serial Number
2	Susan Lim	Platinum	2019-07-11	Installed	R567
3	Dan Par	Platinum	2019-07-11	Installed	B124
4	Lester James	Platinum	2019-07-09	Installed	A123
5	RedBook	Platinum		Installed	T12345
6					

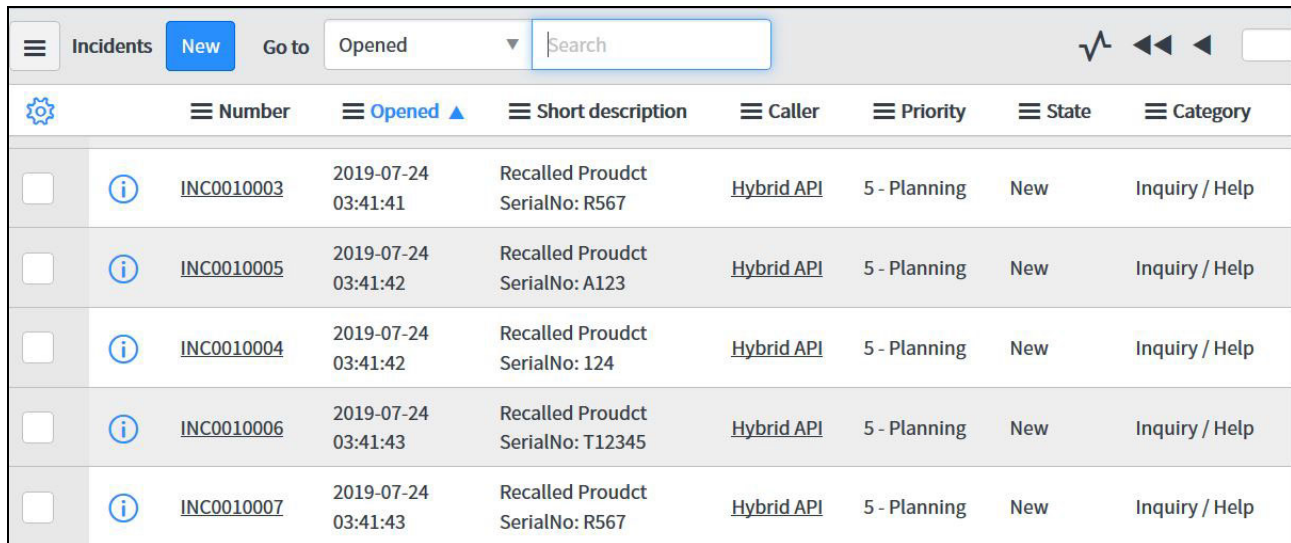
Figure 6-285 Google Sheet output

Congratulations! You have successfully implemented the Recall API.

6.9.9 Final Hybrid API integrated testing

We are now ready to invoke the Hybrid API flow to automate the ServiceNow incidents creation for all the customers under the recall program

1. On the dashboard, click on **HybridAPI** flow, ensure it is running, if it is stopped, start the API by clicking the **Start API** to start it.
2. Follows the steps number 2 through 5 in the previous section 6.9.8, “First, test the simulated on-premises API” on page 364.
 - ▶ Using the product ID as the recalled product (SLA9080 in our tutorial) enter it on the Parameter/body/id and click **Call operation**. The flow will return a success response with a list of the new generated incidents ids.
3. Connect to your ServiceNow instance, and you will see all the newly created incidents with Caller name Hybrid API. Figure 6-286 on page 367 shows ServiceNow incidents created for all the recall customers.



	Number	Opened	Short description	Caller	Priority	State	Category
<input type="checkbox"/>	INC0010003	2019-07-24 03:41:41	Recalled Proudct SerialNo: R567	Hybrid API	5 - Planning	New	Inquiry / Help
<input type="checkbox"/>	INC0010005	2019-07-24 03:41:42	Recalled Proudct SerialNo: A123	Hybrid API	5 - Planning	New	Inquiry / Help
<input type="checkbox"/>	INC0010004	2019-07-24 03:41:42	Recalled Proudct SerialNo: 124	Hybrid API	5 - Planning	New	Inquiry / Help
<input type="checkbox"/>	INC0010006	2019-07-24 03:41:43	Recalled Proudct SerialNo: T12345	Hybrid API	5 - Planning	New	Inquiry / Help
<input type="checkbox"/>	INC0010007	2019-07-24 03:41:43	Recalled Proudct SerialNo: R567	Hybrid API	5 - Planning	New	Inquiry / Help

Figure 6-286 ServiceNow incidents

4. You have successfully implemented the Hybrid API flow.

6.9.10 Conclusion

The business team has become productive with the adoption of IBM App Connect Designer to fulfill their new request. The data now passes between an imported API and designer flows, automatically, in real time. All these steps are done using configuration and data-mapping techniques, without any need for coding, and without the need of an integration specialist.

Reusing APIs from existing environments in IBM App Connect Designer enables the exploration of new APIs in minutes or hours, rather than days or months.

6.10 Implement event sourced APIs

Modern applications have extremely challenging requirements in terms of response times, and availability. In our always-on, always-online world, most mobile applications must be

ready for use at any time of day, any day of the year. And the responsiveness of the interactive experience should be exceptional.

However, any significant application requires data, and not only data it owns, but data from other systems too. The exercises that we have done in the previous sections make it clear that the most obvious and straightforward way to retrieve data from other systems is via an API. However, an API is a real-time synchronous interaction pattern. The system at the other end of that interaction must be always available, and it must be suitably performant at all times. Where this availability and responsiveness is critical, we may need to look to alternative ways to provide more effective access to data.

6.10.1 Implementing the query side of the CQRS pattern

For an API to be highly responsive and available, it really needs a separate, read-optimized data store. The data store can hold the data in the most efficient form for the typical queries that we receive, as shown in Figure 6-287 on page 368. As noted in the introduction, a separate data store has little value for our simple, single-table example. On the other hand, if the "products" were actually a multi-table object you can imagine how this data store could provide significant read optimizations.

This new data store needs to be populated and kept up to date, and must not be coupled into the main database from a runtime perspective. The obvious solution is to asynchronously update it from the event stream we created in 6.7, "Create event stream from messaging" on page 322.

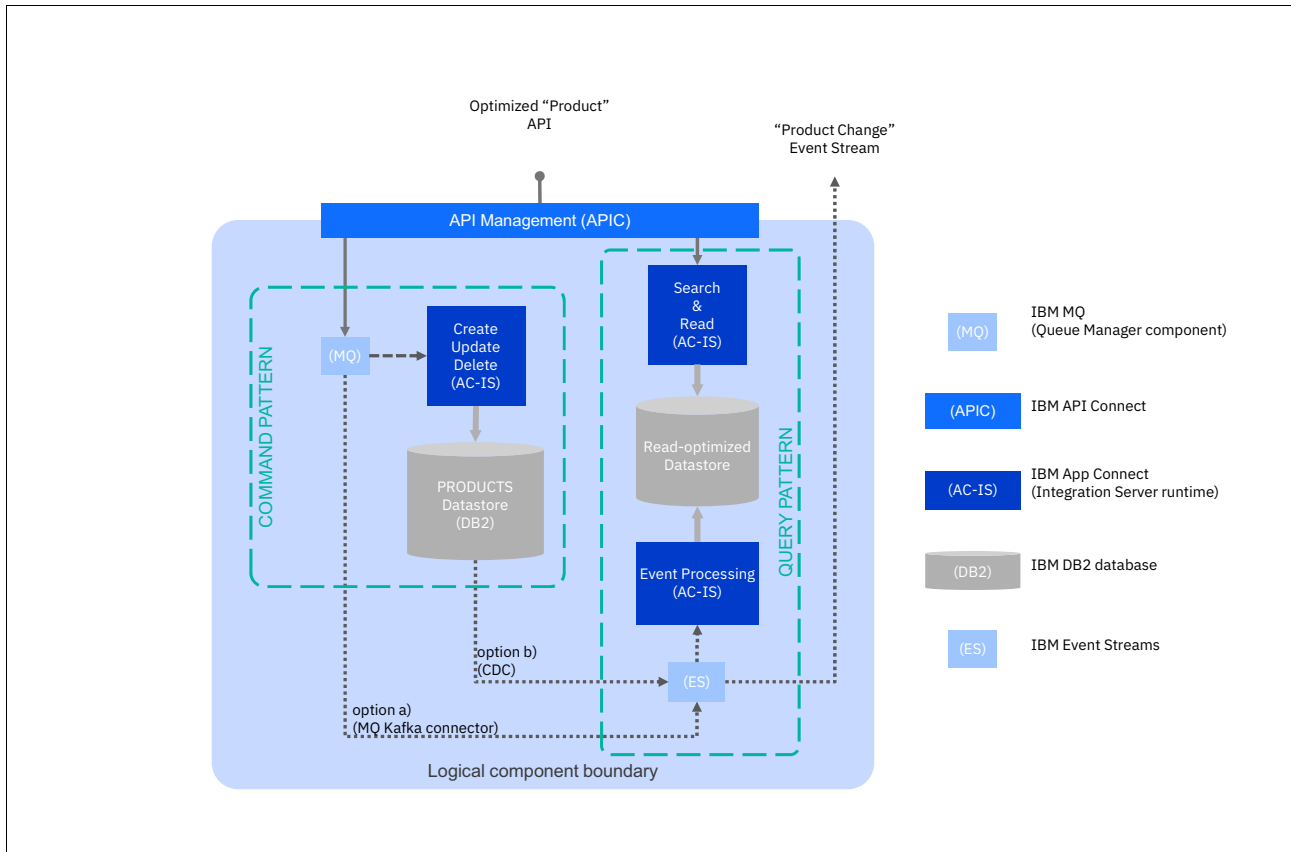


Figure 6-287 Implementing the query side of the CQRS pattern

In previous sections, we already explored how moving to an asynchronous pattern can help for data changes (the "Command" side of CQRS). In our case, the data was submitted over IBM MQ rather than over HTTP. In this section, we consider how we could also use asynchronous patterns to assist with reads (queries).

We are now effectively building out the "Query" side of the CQRS pattern.

6.10.2 Event sourced programming - a practical example

We have seen how the event stream we created in 6.7, "Create event stream from messaging" on page 322 can be used to improve the performance and availability within the boundary of our "Product" business component. This is powerful in itself already. But for the practical part of this section, we want to take this a little further. We want to show how the event stream can also be further reused in other components that implement distributed "event sourced" patterns.

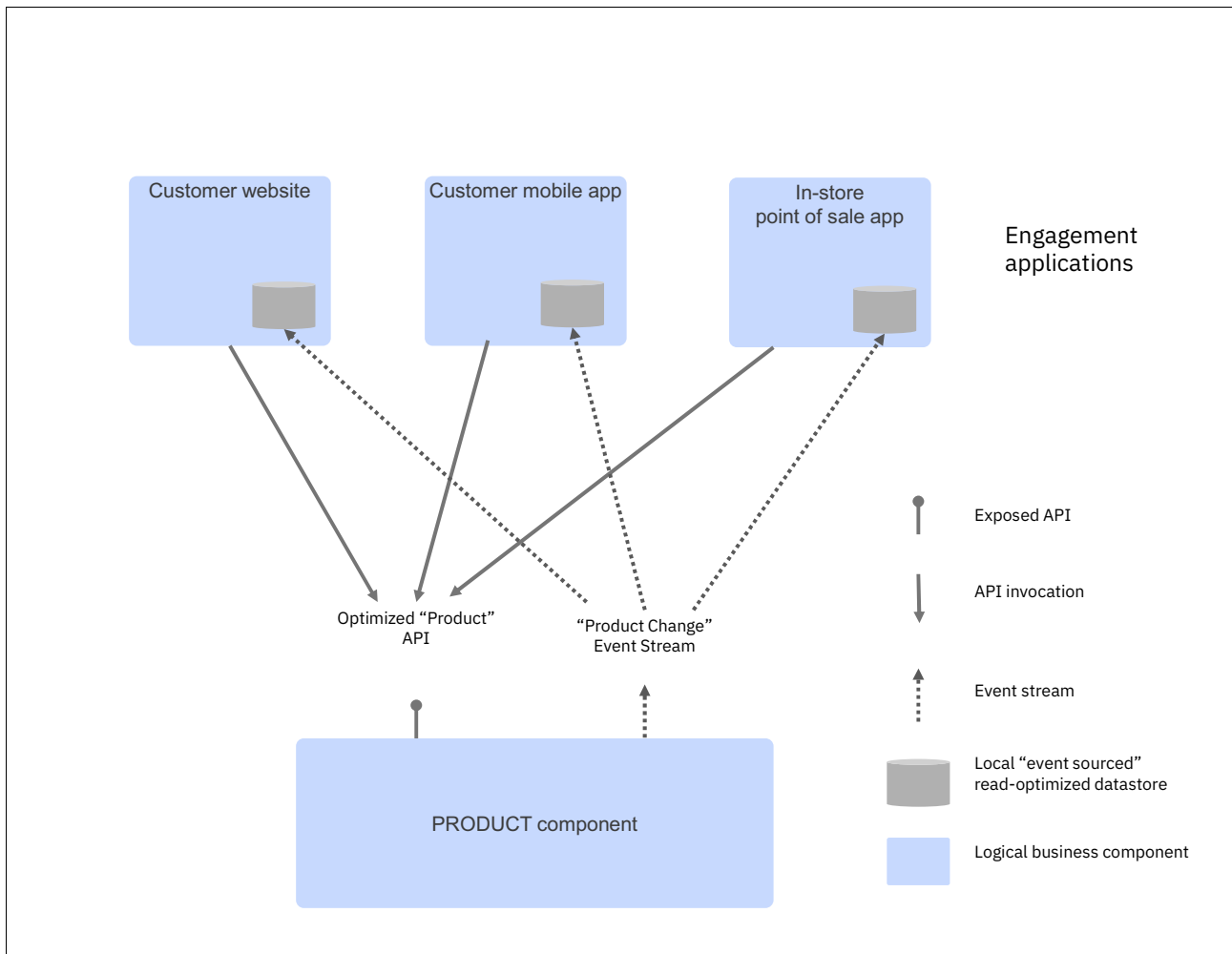


Figure 6-288 Multiple applications reusing both the APIs and the event streams for best customer experience

Figure 6-290 on page 372 shows how multiple engagement applications might each need to build their own read-optimized data stores to serve their specific needs.

The engagement applications can then use a combination of:

- ▶ APIs for simplicity of the programming model.

- ▶ Event streams where they specifically need real-time notifications.
- ▶ Event sourced local data stores where they need complete control over the availability and performance data access.

In the practical example in this section we're going to consider the back-end of a mobile application ("back-end for front end" or BFF) that has a requirement for a local read-optimized data store for Product information. We're going to add a further requirement that it regularly needs to serve up pricing information alongside Product information. However, the pricing information lives in a separate back-end database.

The BFF could of course retrieve the Product and Pricing information using two separate API calls to the respective components, as we did in the earlier example using 6.9.3, "Solution overview" on page 347. However, we can imagine the overall latency starting to escalate, not to mention the dependency on combined availability. Instead, we will introduce a new data store local to the BFF as shown in Figure 6-289, containing both Product and Pricing in exactly the form the BFF needs it.

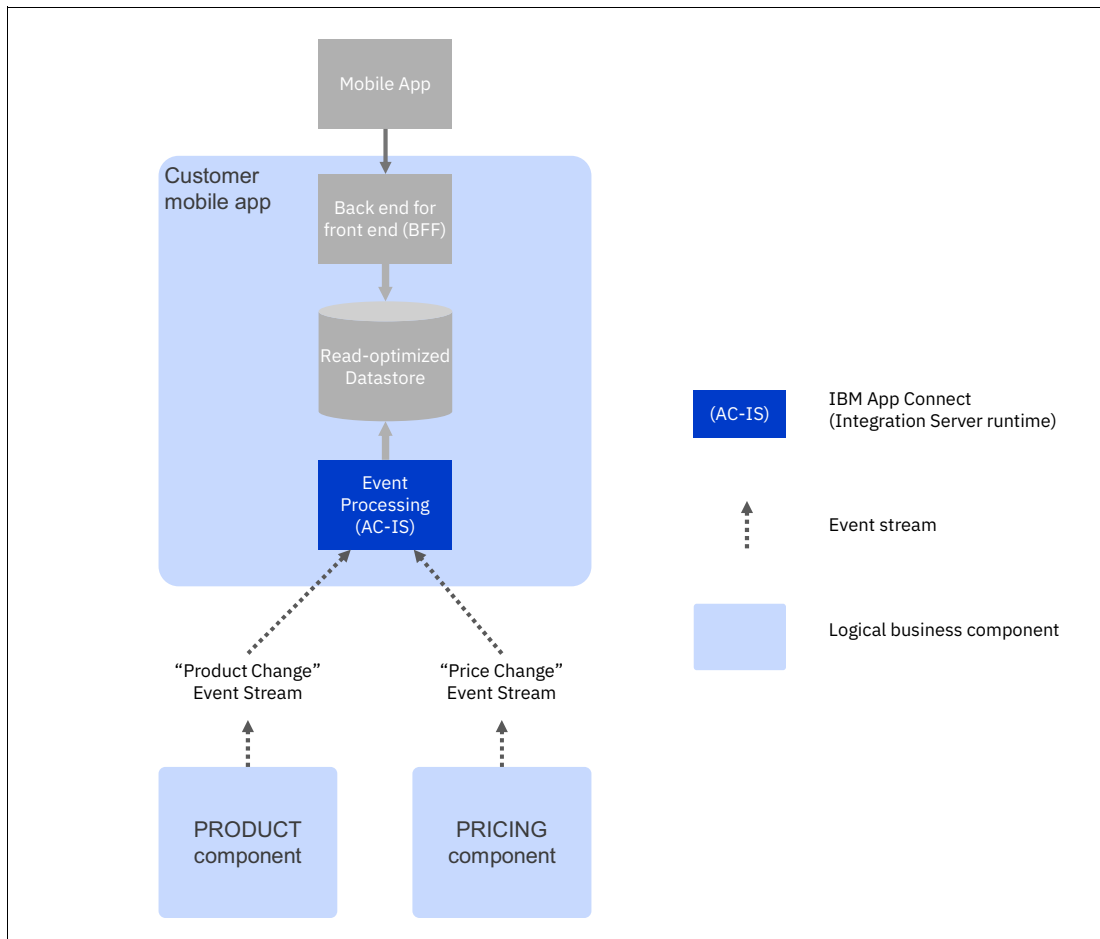


Figure 6-289 Consolidating event streams from multiple sources in order to create a combined read-optimized store.

The mobile application team that implements the BFF will have ownership over that data store, so they can ensure it meets their availability needs. And they can ensure that the data is stored within it in the most optimized form for the type of queries they need to do. They can also change it at will, as they are the only ones using it.

Clearly we need to populate this data store and keep it in synchronization with the back-end systems. To do this we will listen to event streams from the back-end systems and translate these into our local data store. As per the title of this section, this is known as the "event sourced" pattern.

In this scenario we will implement the following features:

1. An IBM App Connect flow running in Integration Server will listen on two separate topics on IBM Event Streams. These topics inform about a change in quantity or price in the two back-end systems for Product and Price data. We will assume these were created in the way discussed in 6.7, "Create event stream from messaging" on page 322.
2. As soon as a change of product or price information is received the information will be picked by a flow in IBM App Connect.
3. Within IBM App Connect the data is merged into a schema that combines the two different schemas of the topics. This 'new' document is then pushed to the IBM Cloudant® database (noSQL).

The scenario has the following pattern:

The existing API's read operation doesn't meet the increasingly demanding non-functional requirements around low-latency responses and high availability. This is because the reads are done on the back-end data store, which was not designed for the volumes of requests we are now seeing nor for the level of availability we require. Furthermore, there is a need to combine the back-end data with information from other sources too.

To overcome this limitation and provide a good customer experience, the goal is to introduce a new API that provides the combined information much more responsively. As a result, the client's non-functional requirements are met and they do not need to make multiple separate API requests.

We could create this new API by performing a composition across the two existing APIs as we did in the earlier example using IBM App Connect Designer in 6.9.3, "Solution overview" on page 347. However, as we will soon see, the solution to this requirement is more technically complex and requires more complex capabilities. So we will use the IBM App Connect runtime and Toolkit to perform the task.

The challenge is that a simplistic composition of APIs would further reduce the response time and availability. Instead we will introduce a new data store local to the new API implementation. This will hold the combined data in exactly the form we need it. Therefore, it will be more performant, and we will have complete control over its availability.

Clearly we need to populate this data store and keep it in synchronization with the back-end systems. To do this, we will listen to event streams from the back-end systems and translate these into our local data store. As per the title of this section, this is known as the "event sourced" pattern.

This circles back to the basics of API management to think *outside-in*, or put another way, *consumer first*. The question to ask is "What would an App Developer need for his app and users?" rather than saying "We have service X, that provides data sets A,B and C and this is what we expose". This new API is heavily consumer focused, providing the best possible experience for a specific category of consumers using the API.

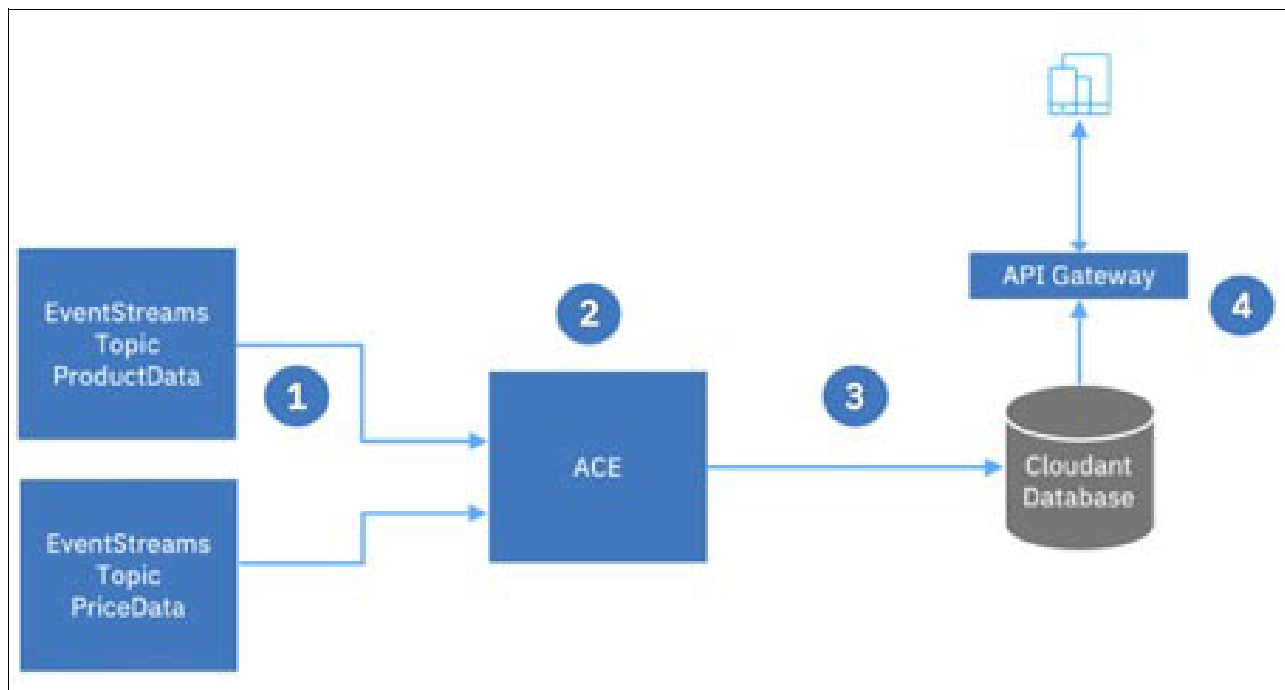


Figure 6-290 Event sourced API

What will be happening in the scenario that is shown in Figure 6-290 is:

1. IBM App Connect will listen on two separate topics on IBM Event Streams - these topics inform upon a change in quantity or price in the two back-end systems for Product and Price data. We will assume these were created in the way that is discussed in 6.7, "Create event stream from messaging" on page 322.
2. As soon as a change of product or price information is received the information will be picked by a flow in IBM App Connect.
3. Within IBM App Connect the data is merged into a schema that combines the two different schemas of the topics. This 'new' document is then pushed to the IBM Cloudant database (noSQL).
4. The content of the database can then be accessed via an API, which is ideally managed by API Connect to protect the database.

Figure 6-291 shows the data flow in this scenario.

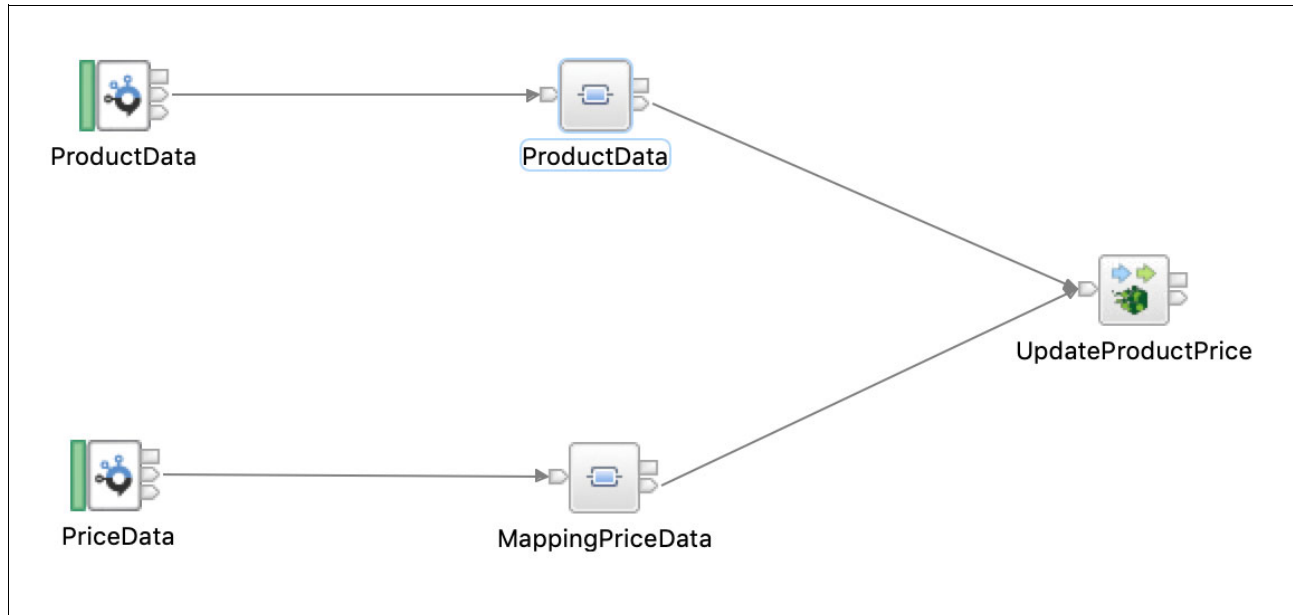


Figure 6-291 IBM App Connect data flow

6.10.3 How to do it?

Let's look at it in reverse order and start with the database.

The scenario described is created with IBM Cloudant Database leveraging this helm chart at <https://github.com/maxgfr/ibm-cloudant>

You are free to use any other database that meets your requirements.

The helm chart is installed on the same Kubernetes platform as the Cloud Pak for Integration for the purpose of illustrating this sample. The database could as well be on IBM Cloud (search for Cloudant in the IBM Cloud catalog <https://cloud.ibm.com/catalog>) or another NoSQL database on any other private or public cloud.

The IBM Cloudant database can be reached the IP and Port, which can be verified by calling the IP and Port. It is essential to get the networking part sorted, so this doesn't obstruct you in building the scenario. See Figure 6-292 on page 374.

The screenshot shows an API client interface with the following details:

- Method:** GET
- URL:** http://{CloudantURL}:{Port}/productprice/1234
- Response Status:** HTTP Status: 200
- Duration:** 46ms
- Response Body (JSON):**

```
{
  "_id": "1234",
  "_rev": "2-41f0af6eb49d3816afc4ae289b361006",
  "part_number": 1234,
  "product": {
    "product_name": "HtTzZ.",
    "quantity": 840,
    "description": "this is a product description"
  },
  "price": {
    "price": 964,
    "currency": "USD"
  }
}
```

Figure 6-292 Calling IBM Cloudant database

You will need a user and password to create a database on your IBM Cloudant instance.

Create a database via the Endpoint using the following API and operation:

PUT `http://{endpoint}:{port}/$DATABASE?partitioned=false`

For sake of simplicity in this book we are not partitioning the database.

You can refer to the Cloudant documentation for details

<https://cloud.ibm.com/docs/services/Cloudant?topic=cloudant-databases>

Note: Cloudant is not prescriptive on what format your documents have that you put in. Any content and structure will be stored in the database.

6.10.4 Creating the flow in IBM App Connect

Create a Kafka Consumer in the IBM App Connect Toolkit.

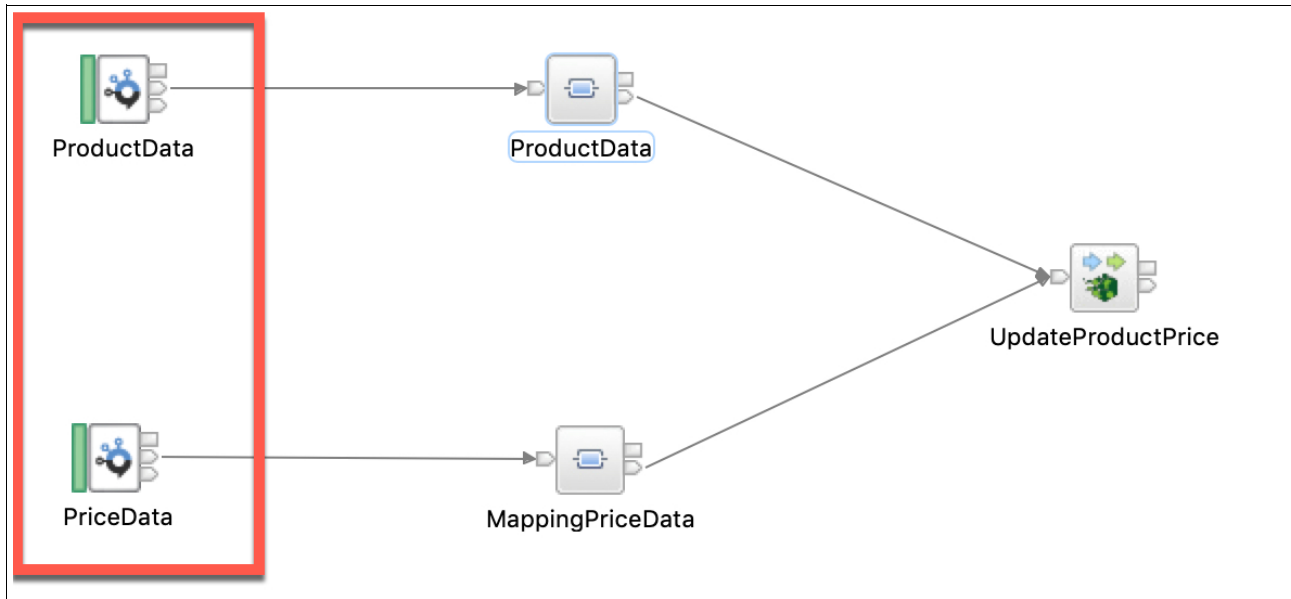


Figure 6-293 Kafka Tutorial

To get this done there are two options:

- Use the tutorials where one tutorial is specifically to create a Kafka Consumer. See Figure 6-294. OR

📖 Tutorial Gallery

Here you can explore and learn about App Connect Enterprise using tutorials.
What are you interested in?

Tool Capabilities

Explore App Connect Enterprise concepts by following simple tutorials

- Using a REST API to manage a set of records
- Using a HTTP Input to drive a message flow
- Using a Mapping node to access a Lookup table that is stored in the Global Cache
- Using a JavaCompute node to split up a large input file
- Using a WorkLoad Management Policy to restrict the maximum throughput of a flow
- Using a LoopBack Request node to insert data into a Cloudant database
- Using a SalesforceRequest node to retrieve records from Salesforce.com
- Using a Sequence node to inject a sequence number into a message
- Using a Resequence node to reorder messages going through a flow
- Using the KafkaProducer and KafkaConsumer nodes with a Kafka topic

Figure 6-294 Kafka Tutorial

- Follow this blog post:

<https://developer.ibm.com/integration/blog/2018/10/01/connecting-app-connect-enterprise-and-ibm-integration-bus-to-ibm-event-streams>

where the second section is for IBM App Connect V11.

Important: At the time of finalizing this document, there is a correction to be made. Of course, the truststore values in the server.conf.yaml must be uncommented and filled in. But also the keystore values must be uncommented and filled in, as shown in Figure 6-295, where **password** = the password of the truststore and **es** is the name of the IntegrationServer (the full path name is obfuscated).

```
keystoreType: 'JKS'  
keystoreFile: ' /es/truststore/es-cert.jks'  
keystorePass: 'es::password'  
truststoreType: 'JKS'  
truststoreFile: ' /es/truststore/es-cert.jks'  
truststorePass: 'es::password'
```

Figure 6-295 Correction in server.conf.yaml

A successful creation of the Kafka Consumer looks like this in the logs of the IntegrationServer that you can run from your IBM App Connect Toolkit. See Figure 6-296.

```
Application 'Kafka2Cloudant' has been changed successfully.  
The source 'Kafka2Cloudantproject.generated.bar' has been successfully deployed.  
Integration server finished with Configuration message.
```

Figure 6-296 Kafka Consumer creation

If something is not complete or not working, you would see an error message like “Kafka2Cloudant encountered a failure and could not start.”

It is a good practice to use a File Output-Node at the beginning as a replacement for your database. Also, you should have a consumed message being written to the local file-system to verify the working of the consumers.

To test the consumption of events from IBM Event Streams it is easiest to create a starter application (as shown in Figure 6-297 on page 377 and Figure 6-296 on page 376).

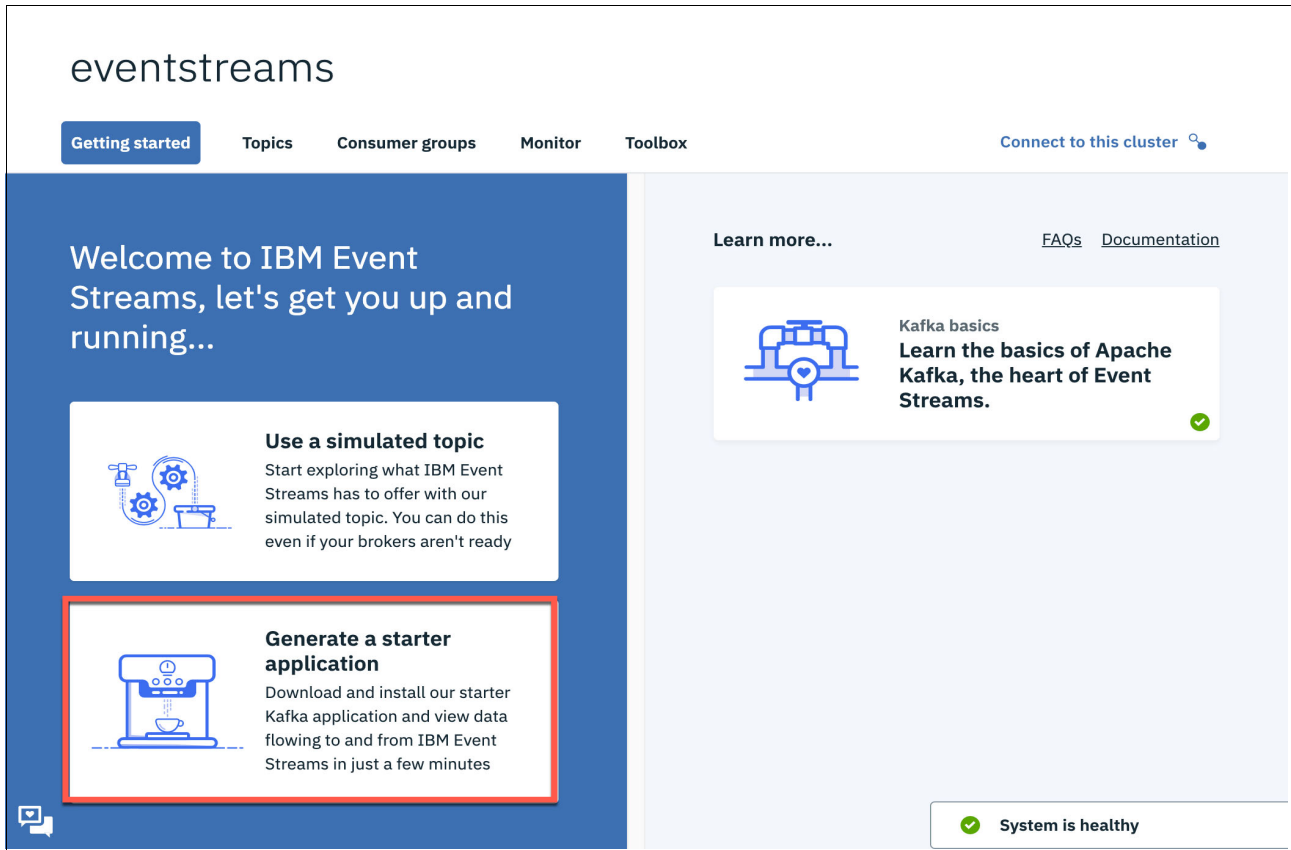


Figure 6-297 Test the consumption of events from IBM Event Streams -1

← Getting started

Generate starter application

Application name ⓘ

IBMRedbooks

What do you want this application to be able to do?

Produce messages

Consume messages

Which topic shall we connect with?

You can choose to create a new topic, or you can choose an existing topic to use in this application

Create topic ⓘ

Choose existing topic ⓘ

Name your topic

PriceData × ▾

[Just looking for the instructions?](#)

Figure 6-298 Test the consumption of events from IBM Event Streams -2

Because a starter application can consume or produce events only in connection with one topic, two applications must be created.

6.10.5 Mapping the received events to the output required

To have a level of standardization it is recommended to follow a structure and combine the information from the product and price-events into one structure to insert it into the Database.

The common denominator between the price and the product information is the part-number. See Figure 6-299 on page 379.


```

{} products_data_model.json ▶ ...
1  {
2    "last_updated": "07-21-2019T12:32:01.322Z",
3    "part_number": 12,
4    "product_name": "duck",
5    "quantity": 100,
6    "description": "a waterbird with a broad blunt bill,
7  }
8

{} price_data_model.json ▶ ...
1  {
2    "last_updated": "07-21-2019T12:32:01.322Z",
3    "part_number": 12,
4    "price": 129,
5    "currency": "USD"
6  }
7

```

Figure 6-299 The price and the product information

The calling application that needs the information on a certain product would be calling for a product ID or "part_number" as it is named in the preceding samples.

It is worth knowing that Cloudant uses an ID on its own to search and identify documents. That ID can be overridden to be the same like the part_number, which saves a lot of work later on.

Note: Be aware of capabilities of the database that can improved life for you and the API consumer (for example, the app developer).

The new payload looks like Figure 6-300 - the schema chosen.

```

1  {
2    "_id": "54",
3    "part_number": 54,
4    "product": {
5      "product_name": "OurProduct",
6      "quantity": 840,
7      "description": "This product is produced in our own factories and sourced locally"
8    },
9    "price": {
10     "price": 25,
11     "currency": "USD"
12   }
13 }

```

Figure 6-300 The schema

The schema definitions for all three payloads need to be in the IBM App Connect project to be leveraged, for example in a message map. As before it is an easy approach to check the tutorial gallery of the IBM App Connect Toolkit for a sample as shown in Figure 6-301 on page 380.

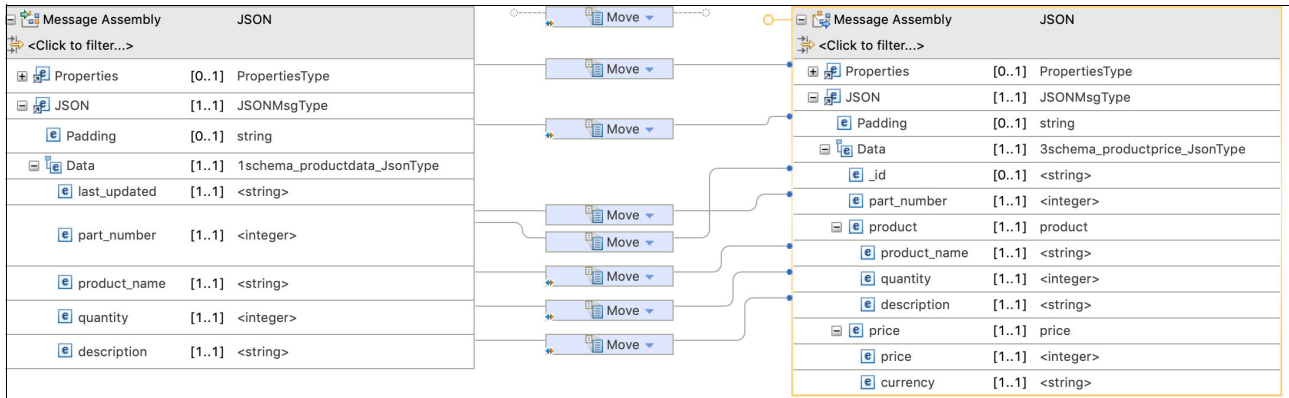


Figure 6-301 Schema definitions

6.10.6 Sending the new payload to the database

Cloudant provides two ways to update entries:

- ▶ Via the loopback connector
- ▶ Via a REST API

The loopback connector is the recommended approach, as it provides you with the mechanisms to update entries in the database. In contrast, the REST API requires a search for an item with the **_id** that equals the **part_number**. Then it maps that to a new payload in a POST request that also contains the revision information (with the flag **_rev**). Otherwise, a POST operation fails for a CouchDB / CloudantDB. See Figure 6-302.

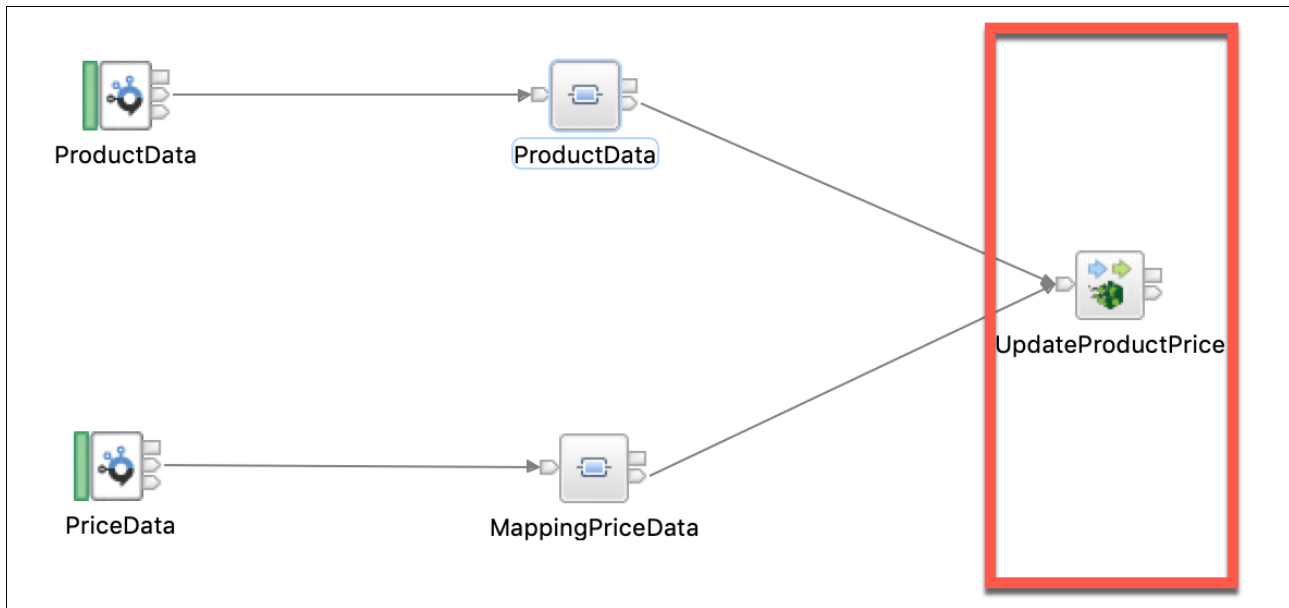


Figure 6-302 IBM App Connect flow - update db

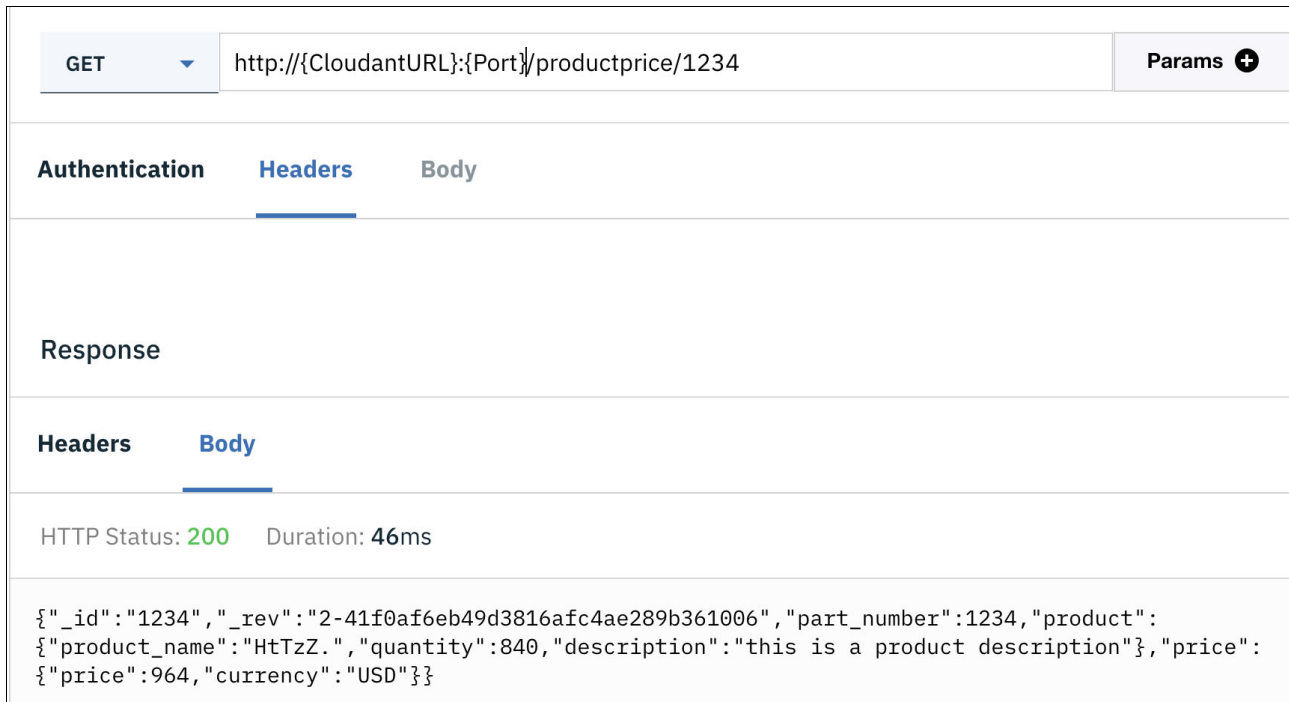
After all is done, deploy the flow onto your Integration Server and create events using the starter applications of IBM Event Streams.

After these events have been consumed and pushed to your database, you will be able to search for the product by leveraging **_id** (which is the **part_number**).

Note: Procedures for updating data in a database vary, so there might be different approaches that you can take.

6.10.7 Client applications

As shown in Figure 6-303, the data is now in our Cloudant database and can be called by any application that uses the combined data sets.



The screenshot shows a REST client interface with the following details:

- Method: GET
- URL: `http://{CloudantURL}:{Port}/productprice/1234`
- Params: +
- Authentication: (empty)
- Headers: (empty)
- Body: (empty)
- Response: (empty)
- Headers: (empty)
- Body: (empty)
- HTTP Status: 200
- Duration: 46ms
- Response Body (JSON):

```
{ "_id": "1234", "_rev": "2-41f0af6eb49d3816afc4ae289b361006", "part_number": 1234, "product": { "product_name": "HtTzZ.", "quantity": 840, "description": "this is a product description" }, "price": { "price": 964, "currency": "USD" } }
```

Figure 6-303 the data is in our Cloudant database

6.11 REST and GraphQL based APIs

When building out a consumer-focused API such as the one in the previous section, we need to ensure we offer the best possible experience for the users of the API. The API is after all a product, and we want it to be appealing for first time users, and also “sticky.” In other words, we want the consumers to continue to use it long into the future.

Therefore, we need to ensure we provide an API that is well suited to the consumer channel we are targeting, which may well mean embracing different styles of API exposure. In this section we will explore an alternative style of API known as GraphQL, which has subtle, but important differences from the more common RESTful style.

REST or (Representational State Transfer) is an architectural style of building a defined set of operations for the interface (Web Service). It is used to send data over HTTP. It is known by its simplicity of using the underlying HTTP verbs, for example GET, POST, PUT and DELETE. It is the most common API exposure style in use today.

GraphQL is a new architectural style for building an API that was mainly developed to overcome some of the architectural limitations of the RESTful APIs.

There are two key limitations in RESTful interfaces that GraphQL addresses:

- ▶ **Data relationships.** RESTful APIs are very granular. You must retrieve each type of data resource separately. An example would be an author and her books. One author may have more than one book. The authors are retrieved separately to the books. To get the books of each author, you must first call the author API, then you will retrieve each author's book using another separate API call. Furthermore, the two separately retrieved data models may then need to be combined by the caller. So, we need to do multiple invocations, and we had to then merge the data together ourselves.
- ▶ **Data filters.** In REST API you always get the full payload in the response. In our example, you get all the data fields about the author or the particular book. This is known as over fetching. What if you need only part of the response? Perhaps you only needed to know the "titles" of the author's books, but not the rest of the book information? This can be critical where resource data payloads are large and they are going over low-bandwidth connections - the typical reality for a mobile application for example. This filtering aspect also applies to being able to perform more complex queries on the lists we retrieve such that we do not bring back all the records in a list. For example what if we only wanted to retrieve books published before 1997.

Figure 6-304 shows the REST API flow.

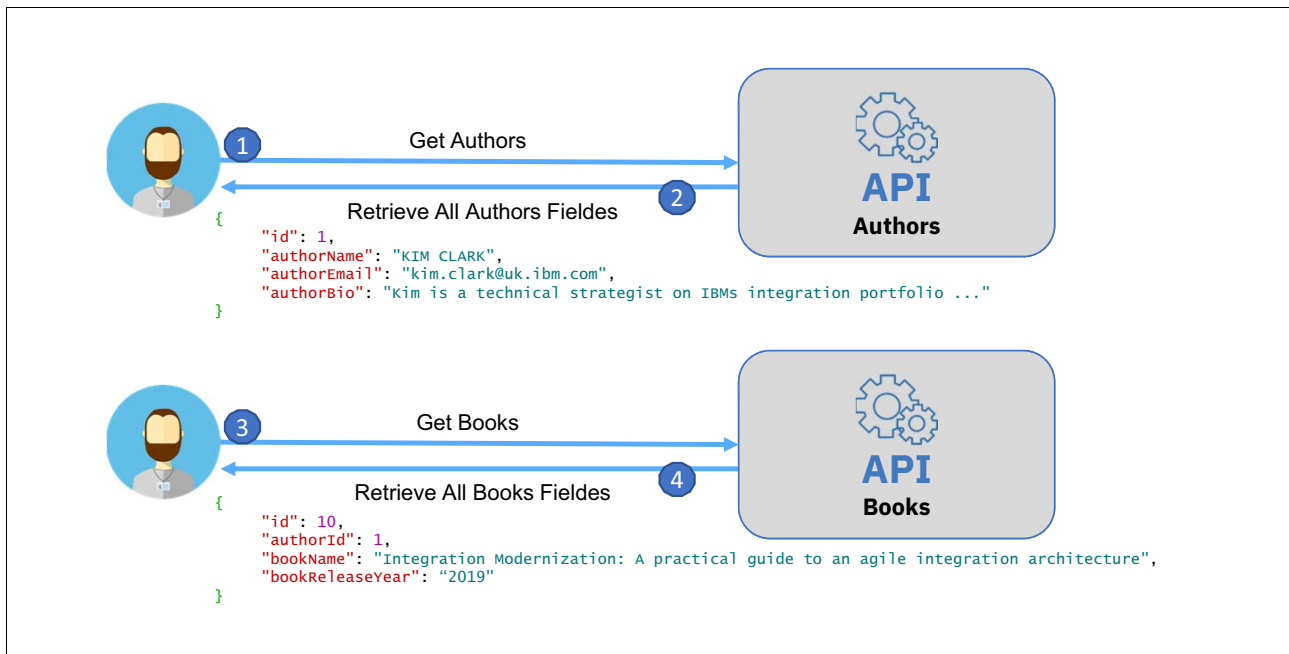


Figure 6-304 REST API flow

To overcome these limitations, we have seen different projects in the market such as OData (Open Data Protocol), GraphQL, and the LoopBack framework. In this document, we focus on GraphQL because its popularity is growing rapidly.

It is important to note that GraphQL is not a REST API replacement, it is only an alternative. REST and GraphQL often co-exist in the same project depending on the type of interfaces exposed.

However, we will first explore the open source LoopBack project, which is integrated with IBM API Connect. We previously mentioned Loopback as an alternative to GraphQL, because it does indeed provide a convention for API exposure that resolves the REST limitations noted above. However, Loopback is more than that. LoopBack is a Node.js based framework that provides a way to rapidly create a REST API implementation from a data source. We will use this in our example to create a sample REST API. We can then briefly explore how the LoopBack augmentations to REST enable GraphQL-like interaction. Finally, we will show how we can place a Wrapper around our Loopback-based REST API (or indeed any Open API Specification based API) to convert it into GraphQL.

6.11.1 IBM, GraphQL, and Loopback

IBM has a strong commitment to open source. We are an active member of the GraphQL community, and the original creator of the now open source “OpenAPI-to-GraphQL” project used at the end of this section.

“We are pleased to join the new GraphQL Foundation as a founding member to help drive greater open source innovation and adoption of this important data access language and runtime for APIs.” – Juan Carlos Soto, VP Hybrid Cloud Integration and API Economy, IBM.

IBM is also the owner and primary contributor to the open source LoopBack framework used in this example to rapidly create API implementations.

6.11.2 LoopBack models and relationships

LoopBack is a highly extensible, open-source Node.js framework based on Express that enables you to quickly create APIs and microservices that are composed from back-end systems such as databases and SOAP or REST services.

Individual models are easy to understand and work with. But in reality, models are often connected or related. When you build a real-world application with multiple models, you’ll typically need to define relations between models. For example:

- A customer has many orders and each order is owned by a customer.
- A user can be assigned to one or more roles and a role can have zero or more users.
- A physician takes care of many patients through appointments. A patient can see many physicians too.

With connected models, LoopBack exposes as a set of APIs to interact with each of the model instances and query and filter the information based on the client’s needs.

You can define the following relationships (called relations in LoopBack) between models:

- BelongsTo
- HasOne
- HasMany
- HasManyThrough
- HasAndBelongsToMany
- Polymorphic
- Embedded (embedsOne and embedsMany)

One of the most useful LoopBack capabilities is that it has an automated build wizard to connect to different databases, create the model's relationships, and perform filtering that is similar to the GraphQL. Figure 6-305 on page 384 shows LoopBack and GraphQL.

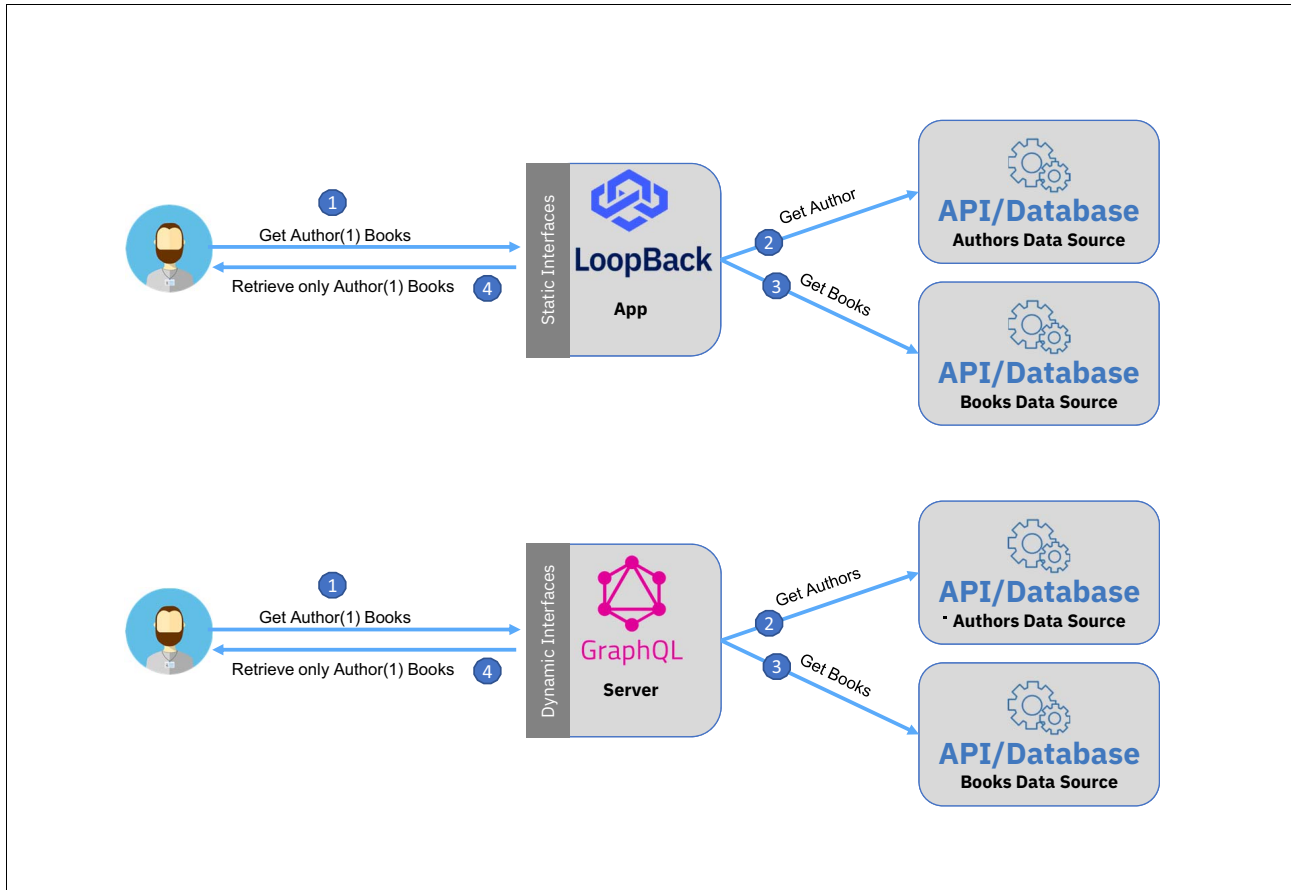


Figure 6-305 LoopBack and GraphQL

A key difference between LoopBack and GraphQL is that LoopBack uses the URL to specify the relations and the filters. In contrast, GraphQL puts that as part of the payload body as shown in Figure 6-306.

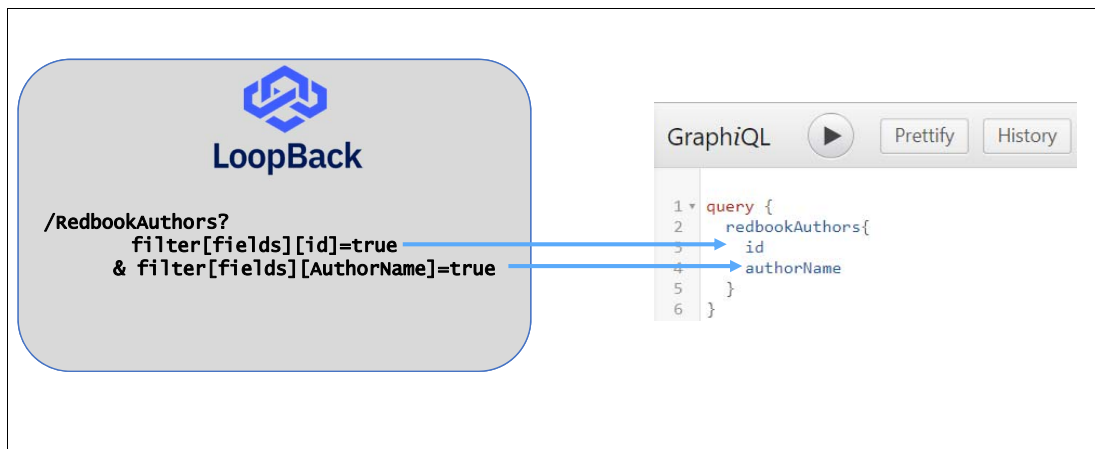


Figure 6-306 LoopBack REST API mapping to GraphQL

We will now:

1. See the creation of an API implementation using the IBM API Connect LoopBack framework and test the out-of-the-box filters, relations and model creation that LoopBack provides.
2. Create a GraphQL wrapper to expose the created loopback REST API as a GraphQL service.

LoopBack REST API creation

Let's start by creating the LoopBack application that exposes a data source as an API and feeds it with sample data, and then try some of the filters and database join (model relation).

Prerequisites:

1. Install NPM using the NodeJS installer from (<https://nodejs.org/en/download/>)
2. Install API Connect CLI that has the LoopBack framework out-of-the-box from IBM fix central (<https://www-945.ibm.com/support/fixcentral/swg/selectFixes?parent=ibm~WebSphere&product=ibm/WebSphere/IBM+API+Connect&platform=All&function=all>)

Note: You may also use the open source LoopBack framework available at <https://loopback.io/doc/en/lb4/Getting-started.html>.

Having the LoopBack framework embedded with IBM API Connect is an advantage. It enables the developers to develop APIs and microservices faster and deploy it to IBM Public Cloud or OpenShift.

To further understand the database relation that we are aiming for, see Figure 6-307.

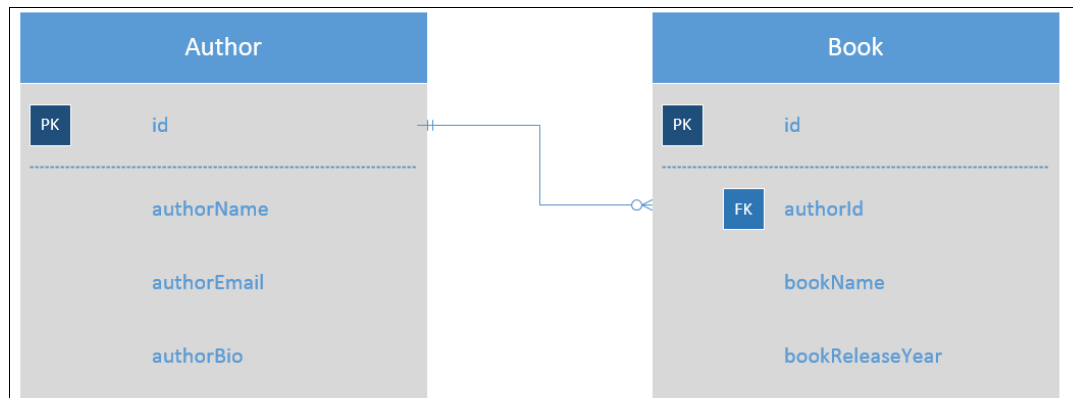


Figure 6-307 Entity relationship diagram

Creating the application with LoopBack is simple and straight forward and can be done in five steps:

1. Create the LoopBack application.
2. Define the data source.
3. Create the Authors and Books models.
4. Create the model's relations.
5. Test the API using LoopBack embedded explorer.

Create the LoopBack application

Perform the following steps:

1. Type in the command in Example 6-8

Example 6-8 Creating the LoopBack application 1

```
apic lb app
```

2. Enter the details to the wizard as Example 6-9.

Example 6-9 Creating the LoopBack application 2

```
? What's the name of your application? RedbookApp
? Enter name of the directory to contain the project: RedbookApp
? What kind of application do you have in mind? empty-server (An empty LoopBack
API, without any configured models or datasources)
```

3. This will create a new directory called “RedbookApp” that contains all the application data.

Navigate to the folder using the following command in Example 6-10 on page 386.

Example 6-10 Creating the LoopBack application 3

```
cd RedbookApp
```

Define the data source

We will define an in-memory data source which will allow us to use the file system folders as a data repository. See Example 6-11 shows the command and Example 6-12 on page 386 shows the output.

Example 6-11 Defining the data source 1

```
apic lb datasource
```

Example 6-12 Defining the data source 2

```
? Enter the datasource name: memorydb
? Select the connector for memorydb: In-memory db (supported by StrongLoop)
? window.localStorage key to use for persistence (browser only):<Leave Blank>
? Full path to file for persistence (server only): mydata.json
```

Create authors and books models

Perform the following steps:

1. To create a model using the LoopBack framework type the following command as shown in Example 6-13.

Example 6-13 Defining the data source 3

```
apic lb model
```

2. Use Example 6-14 to answer the wizard for creating the Author model.

Example 6-14 Defining the data source 4

```
? Enter the model name: Author
? Select the datasource to attach Author to: memorydb (memory)
? Select model's base class PersistedModel
? Expose Author via the REST API? Yes
```

? Custom plural form (used to build REST URL):<Leave Blank>
? Common model or server only? **common**
Let's add some Author properties now.

Enter an empty property name when done.

? Property name: **authorName**
 invoke loopback:property
? Property type: **string**
? Required? **Yes**
? Default value[leave blank for none]: **<Leave Blank> or Use your name**

Let's add another Author property.

Enter an empty property name when done.

? Property name: **authorEmail**
 invoke loopback:property
? Property type: **string**
? Required? **No**
? Default value[leave blank for none]: **<Leave Blank> or Use your email**

Let's add another Author property.

Enter an empty property name when done.

? Property name: **authorBio**
 invoke loopback:property
? Property type: **string**
? Required? **No**
? Default value[leave blank for none]:**<Leave Blank>**

Let's add another Author property.

Enter an empty property name when done.

? Property name:**<Press Enter to Exit the Wizard>**

3. Use Example 6-15 to answer the wizard for creating the **Book** model.

Example 6-15 Defining the data source 5

? Enter the model name: **Book**
? Select the datasource to attach Book to: **memorydb (memory)**
? Select model's base class **PersistedModel**
? Expose Book via the REST API? **Yes**
? Custom plural form (used to build REST URL):
? Common model or server only? **common**
Let's add some Book properties now.

Enter an empty property name when done.

? Property name: **authorId**
 invoke loopback:property
? Property type: **number**
? Required? **Yes**
? Default value[leave blank for none]:**<Leave Blank>**

Let's add another Book property.

Enter an empty property name when done.

? Property name: **bookName**
 invoke loopback:property
? Property type: **string**

? Required? **Yes**
? Default value[leave blank for none]: **<Leave Blank>**

Let's add another Book property.
Enter an empty property name when done.

? Property name: **bookReleaseYear**
 invoke loopback:property
? Property type: **string**
? Required? **No**
? Default value[leave blank for none]: **2019**

Let's add another Book property.
Enter an empty property name when done.
? Property name:**<Press Enter to Exit the Wizard>**

Now you have created two models. Next let us create the relation between the two models.

Create the model relation

1. Start by the following command in Example 6-16 to create a model relation.

Example 6-16 Creating the model relation 1

```
apic lb relation
```

2. Use Example 6-17 to create the model relation.

Example 6-17 Creating the model relation 2

? Select the model to create the relationship from: **Author**
? Relation type: **has many**
? Choose a model to create a relationship with: **Book**
? Enter the property name for the relation: (books) **books**
? Optionally enter a custom foreign key: **authorId**
? Require a through model? **No**
? Allow the relation to be nested in REST APIs: **No**
? Disable the relation from being included: **No**

Testing the API

Now that we have created the app, data source, model, and relations, let us test the API.

1. To run the application on your local host type as in Example 6-18.

Example 6-18 Testing the API 1

```
node .
```

2. You will get the localhost url to access the API Explorer. See Example 6-19.

Example 6-19 Testing the API 2

```
Web server listening at: http://localhost:3010  
Browse your REST API at http://localhost:3010/explorer
```

3. The API Explorer is an embedded testing that allows you to test your application during development. Use your browser to access the LoopBack API Explorer as shown in Figure 6-308 on page 389.

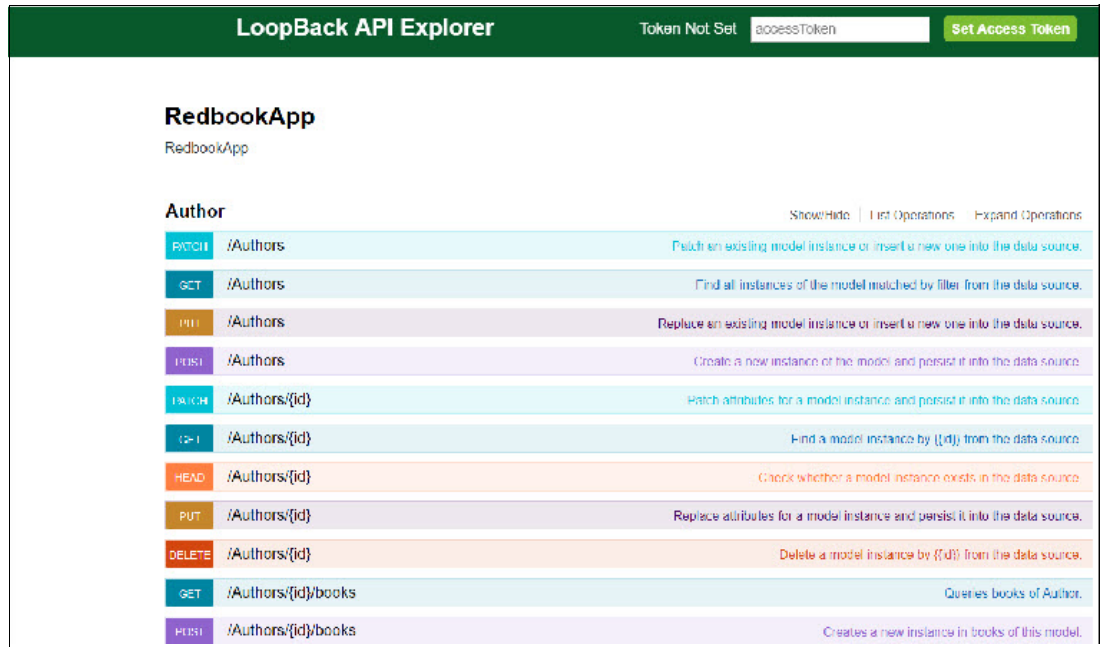


Figure 6-308 LoopBack API Explorer

4. You can test your APIs now using the explorer. However, we don't have any data on the local store, so let's add some data. Click **POST** operation and complete the fields as shown in Example 6-20.

Example 6-20 Testing the API 3

```
{
  "authorName": "Kim Clark",
  "authorEmail": "kim.clark@uk.ibm.com",
  "authorBio": "Kim is a technical strategist on IBMs integration portfolio..."
}
```

5. Then click **Try it out!** as shown in Figure 6-309 on page 390.

LoopBack API Explorer Token Not Set Set Access Token

POST /Authors Create a new instance of the model and persist it into the data source.

Response Class (Status 200)
Request was successful

Model | Example Value

```
{
  "authorName": "Mohammed Alreedi",
  "authorEmail": "[REDACTED]@sa.ibm.com",
  "authorBio": "string",
  "id": 0
}
```

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
data	<pre>{ "authorName": "Kim Clark", "authorEmail": "[REDACTED]@uk.ibm.com", "authorBio": "Kim is a technical strategist on IBMs integration portfolio..." }</pre>	Model instance data	body	Model Example Value

Parameter content type: application/json

Try it out!

```
{
  "authorName": "Mohammed Alreedi",
  "authorEmail": "[REDACTED]@sa.ibm.com",
  "authorBio": "string"
}
```

Figure 6-309 Testing the API 4

6. You see the **200 successful operation** message. See Figure 6-310 on page 391.

Try it out! [Hide Response](#)

Curl

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{ \
  "authorName": "Kim Clark", \
  "authorEmail": "██████████@uk.ibm.com", \
  "authorBio": "Kim is a technical strategist on IBMs integration portfolio..." \
}' 'http://localhost:3010/api/Authors'
```

Request URL

http://localhost:3010/api/Authors

Response Body

```
{
  "authorName": "Kim Clark",
  "authorEmail": "██████████@uk.ibm.com",
  "authorBio": "Kim is a technical strategist on IBMs integration portfolio...",
  "id": 2
}
```

Response Code

200

Figure 6-310 Testing the API 5

- After the first successful POST, you can check the local storage file on the same application directory that was created during the data source creation step (mydata.json) See Figure 6-311.

This PC > Local Disk (C:) > IBM > RedbookApp

Name	Date modified	Type	Size
client	8/4/2019 8:22 AM	File folder	
common	8/4/2019 8:47 AM	File folder	
node_modules	8/4/2019 8:23 AM	File folder	
server	8/4/2019 8:22 AM	File folder	
.editorconfig	8/4/2019 8:22 AM	EDITORCONFIG File	1 KB
.eslintignore	8/4/2019 8:22 AM	ESLINTIGNORE File	1 KB
.eslintrc	8/4/2019 8:22 AM	ESLINTRC File	1 KB
.gitignore	8/4/2019 8:22 AM	Text Document	1 KB
.yo-rc.json	8/4/2019 8:22 AM	JSON File	1 KB
apic.exe	7/16/2019 4:36 PM	Application	76,788 KB
mydata.json	8/4/2019 10:17 AM	JSON File	1 KB
package.json	8/4/2019 9:06 AM	JSON File	1 KB
package-lock.json	8/4/2019 8:23 AM	JSON File	109 KB

Figure 6-311 Testing the API 6

- You can also copy and paste the following Example 6-21 if you want to add the data without using the API Explorer.

Example 6-21 Sample data

```
{
  "ids": {
    "Author": 2,
    "Book": 4
  },
  "models": {
    "Author": {
      "1": "{\"id\":1,\"authorName\":\"Mohammed Alreedi\",\"authorEmail\":\"malreedi@sa.ibm.com\",\"authorBio\":\"Mohammed Alreedi is the MEA Technical Integration Leader\"}",
      "2": "{\"id\":2,\"authorName\":\"Kim Clark\",\"authorEmail\":\"kim.clark@uk.ibm.com\",\"authorBio\":\"Kim is a technical strategist on IBMs integration portfolio...\"}"
    },
    "Book": {
      "1": "{\"authorId\":2,\"bookName\":\"IBMRedbooks1\",\"bookReleaseYear\":\"2017\",\"id\":1}",
      "2": "{\"authorId\":1,\"bookName\":\"IBMRedbooks2\",\"bookReleaseYear\":\"2018\",\"id\":2}",
      "3": "{\"authorId\":2,\"bookName\":\"IBMRedbooks3\",\"bookReleaseYear\":\"2019\",\"id\":3}"
    }
  }
}
```

Note: You must restart the server to pick-up the modified mydata.json file.

9. After the server restart, you can test the GET operation and check the result. See Figure 6-312 on page 393.

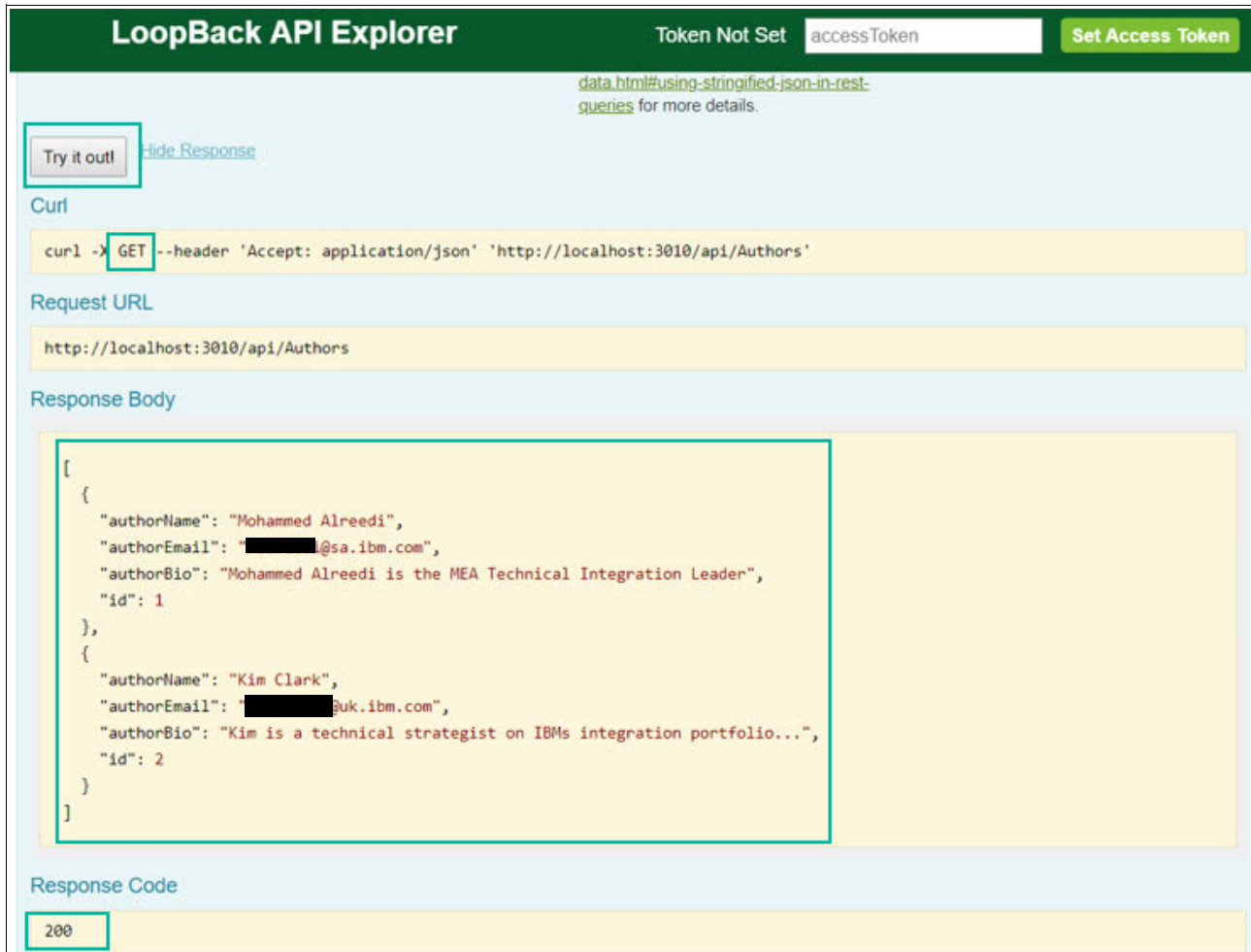


Figure 6-312 Testing the API 6

LoopBack join and filters

In section “LoopBack REST API creation” on page 385 we have created a model relation which is basically a data join. To demonstrate this let’s take an example of retrieving all the author books.

You see an API created out-of-the-box to get the books based on an authorId. Let’s try to use that.

1. Click **GET /Authors/{id}/books**. Enter 1 for the ID, then click **Try it out!** See Figure 6-313 on page 394.

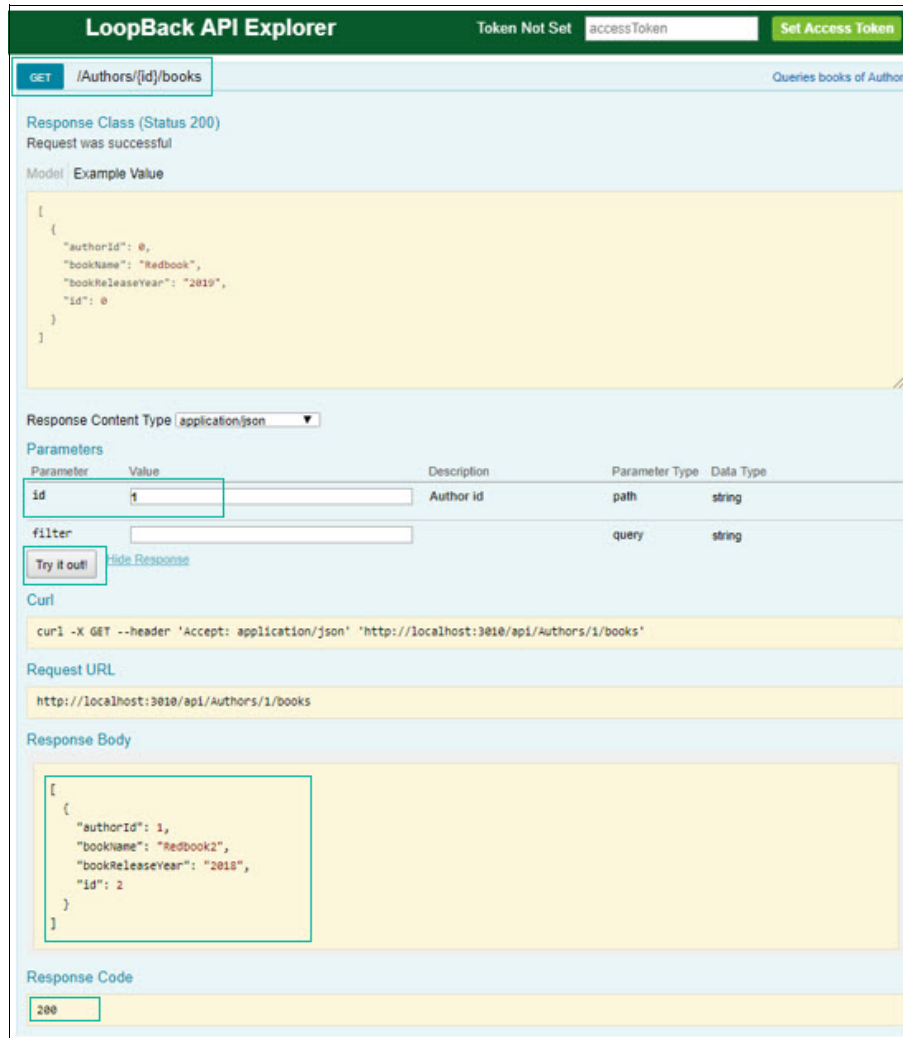


Figure 6-313 Testing the API 7

2. You can also test the same from any external tool or even the web-browser. To do that, use the link in Example 6-22.

Note: Do not forget to change the port if you have a different server port.

Example 6-22 API testing link

<http://localhost:3010/api/Authors>

Figure 6-314 shows the API testing result.

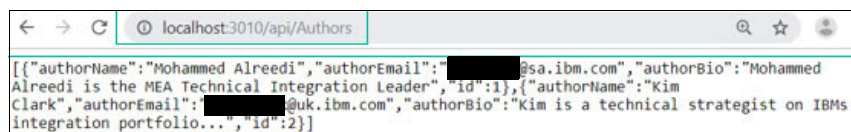


Figure 6-314 API testing result

3. Next is filters. Filters specify criteria for the returned data set. LoopBack supports the following kinds of filters:

- Fields filter
- Include filter
- Limit filter
- Order filter
- Skip filter
- Where filter

More information about filters can be found here:

<https://loopback.io/doc/en/lb3/Querying-data.html>

Let's try some filters such as applying a filter to show only authorName.

4. Open your browser and type the string that you see in Example 6-23.

Example 6-23 API testing link with filters 1

`http://localhost:3010/api/Authors?filter[fields][authorName]=true`

This will retrieve only the name of all authors. See Figure 6-315.



Figure 6-315 API testing link with filters 2

5. Another filter that can be used is the **where** filter, based on a specific goal like finding all books that were released after 2017.

Example 6-24 API testing link with filters 2

`http://localhost:3010/api/Books?filter[where][bookReleaseYear][gt]=2017`

Figure 6-316 shows the *API testing result*.

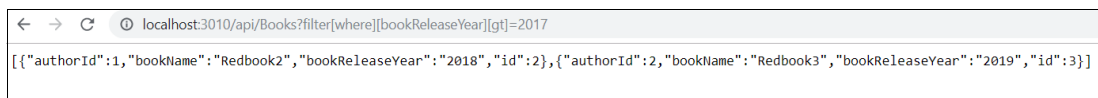


Figure 6-316 API testing result

6. You can also combine two filters together as seen in Example 6-25.

Example 6-25 API testing link with filters 3

`http://localhost:3010/api/Books?filter[where][bookReleaseYear][gt]=2017&filter[fields][bookName]=true`

Figure 6-317 on page 395 shows the result.

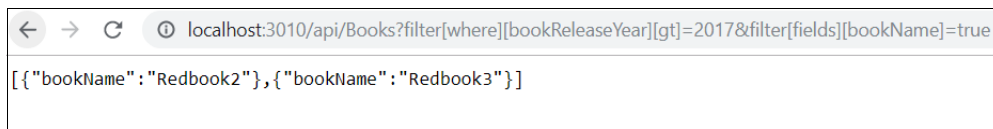


Figure 6-317 API testing result

Generate GraphQL out of OAS based API

Perform the following steps:

1. OpenAPI-to-GraphQL can be used either as a library, or via its Command Line Interface (CLI) to quickly get started. To install the OpenAPI-to-GraphQL CLI, run the indicated command Example 6-26.

Example 6-26 Generating the graphql API 1

```
npm i -g openapi-to-graphql-cli
```

2. OpenAPI-to-GraphQL relies on the OpenAPI Specification (OAS) of an existing API to create a GraphQL interface around that API.
3. To create a GraphQL wrapper, first download the open api definition or swagger, use the following link in the browser

Example 6-27 Download the swagger.json link

```
http://localhost:3010/explorer/swagger.json
```

Note: if you are using LoopBack V4, you will use openapi.json instead of swagger.json.

4. **Save** the swagger.json to your local disk, name it RedbookAppDef.json, and then add the following server IP address at the end of the “RedbookAppDef.json” file.

Example 6-28 Adding the server url in the swagger.json

```
'  
  "servers": [  
    {  
      "url": "http://localhost:3010/api/"  
    }  
  ]  
'
```

The file end should look like Figure 6-318.

```

    "Author": {
      "properties": {
        "authorName": {
          "default": "Mohammed Alreedi",
          "type": "string"
        },
        "authorEmail": {
          "default": "malreedi@sa.ibm.com",
          "type": "string"
        },
        "authorBio": {
          "type": "string"
        },
        "id": {
          "type": "number",
          "format": "double"
        }
      },
      "required": [
        "authorName"
      ],
      "additionalProperties": false
    }
  ],
  "servers": [
    {
      "url": "http://localhost:3010/api/"
    }
  ]
}

```

Figure 6-318 The file after adding the server url

5. Make sure your LoopBack application is running because the GraphQL service that we are creating is only a wrapper to the created LoopBack API.
6. After OpenAPI-to-GraphQL is installed and the OAS/Swagger is obtained, you can create and start the GraphQL server from the folder in which OpenAPI-to-GraphQL is installed. Run the following comment shown in Example 6-29.

Example 6-29 Run the graphql server

```
openapi-to-graphql RedbookAppDef.json
```

Figure 6-319 shows the *command output*.

```

"numOps": 43,
"numOpsQuery": 15,
"numOpsMutation": 28,
"numQueriesCreated": 15,
"numMutationsCreated": 26
}
GraphQL accessible at: http://localhost:3000/graphql

```

Figure 6-319 Command output

Now you can access the GraphQL interface using the URL
`http://localhost:3000/graphql`

7. Open your browser and navigate to the above URL. Use the following query to retrieve all the authors.

Example 6-30 GraphQL query example 1

```
query{
  authors{
    authorName
    authorEmail
    authorBio
  }
}
```

Figure 6-320 shows the result of the query.

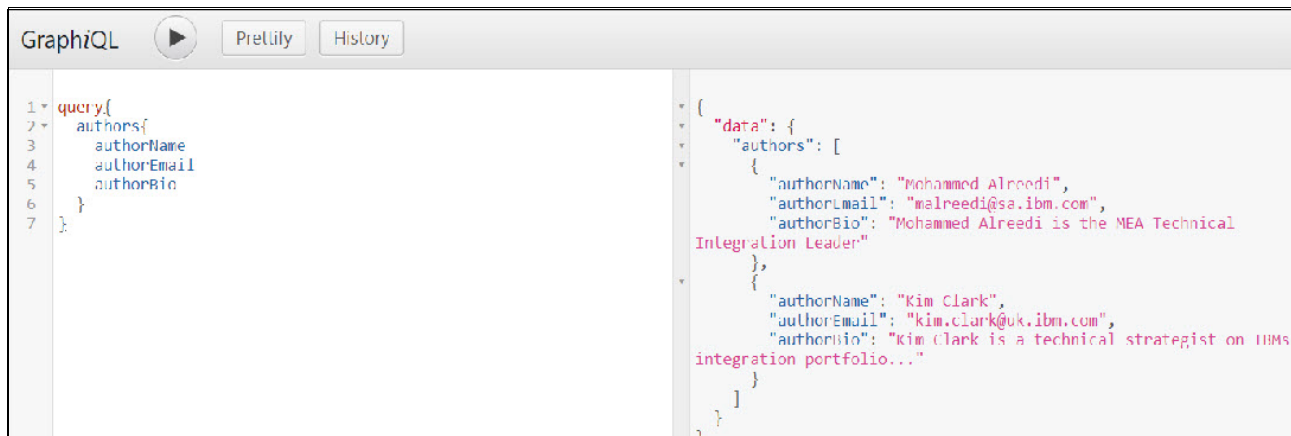


Figure 6-320 GraphQL query result

With that we have concluded the GraphQL wrapper creation which will give us the ability to build a GraphQL API based on a RESTful API.

6.12 API testing

A key element to cloud native based agile delivery is automation. How else could you reduce the code to production cycle time? But of course we must also retain quality, so an essential part of that automation is test automation. We need to be able to rapidly capture, and amend test cases that we want to include in our pipeline processes.

This section covers the specific step-by-step implementation of the creation and maintenance of an API test. There is a broader discussion on API testing strategy in 6.5.5, “API testing” on page 268.

For the implementation of the scenario, *IBM API Test and Monitor* on IBM Cloud is used.

API Test and Monitor provides the following three options to create and update test cases:

- ▶ Create from an API request. In this option, the test is generated automatically from the response that’s returned from the endpoint.

- ▶ Create from an existing specification file. In this option, you are not dependent on the completion of the API development or connectivity, having the specification file will be sufficient to generate the test cases.
- ▶ Create using the visual or code test editor. This option gives you the flexibility to build or update your test assertions with a visual or code editor at any phase of your API project. Especially if you adopt test driven requirements approach your test cases will directly reflect, eventually your test cases will trace to the API definition and implementation.

In this scenario, you are going to combine all three approaches.

6.12.1 Create a test from an API request

This is the simplest way to create a test. Most of the work is done for you by generating the test artifacts based on an example invocation of the API. Complete the following steps to create a test from an API request:

1. Log in **IBM API Connect Test and Monitor** from the following page (<https://ibm.biz/apitest>) as shown in Figure 6-321.

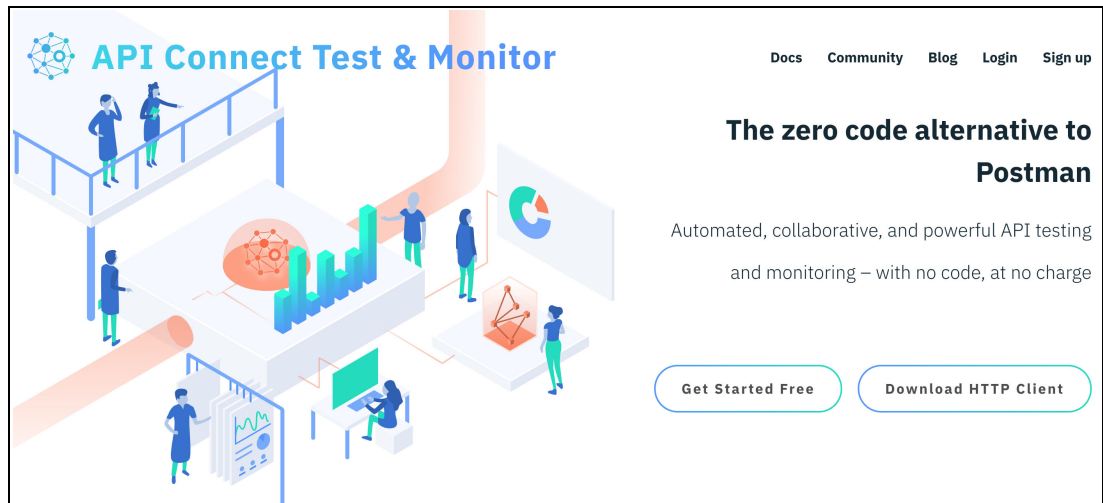


Figure 6-321 API Test and Monitor Login page

2. Open **HTTP Client** as shown in Figure 6-322 on page 400.

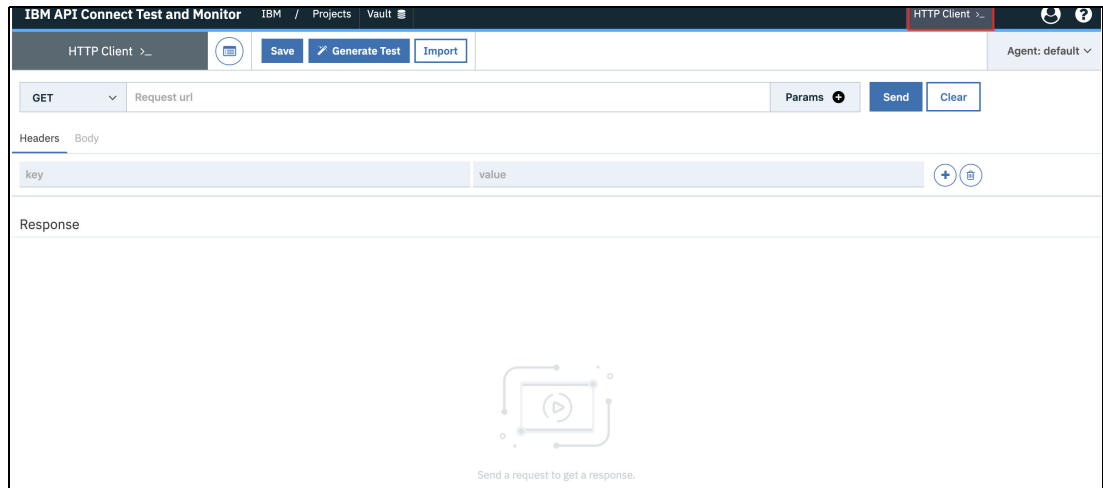


Figure 6-322 Open HTTP Client

3. Configure and send an API test request as shown in Figure 6-323.
 - a. In the upper left section of the **HTTP Client**, select the type of request (GET, POST, PUT, PATCH and DELETE) that you want from the drop-down menu.
 - b. Complete the **Request url** field with the API endpoint URL.
 - c. Click **Params** icon if it requires to add parameters to the API endpoint URL.
 - d. Select the **Headers** tab to supply request HTTP headers.
 - e. Select **Body** tab to configure the request body.
 - f. Click **Send** in the upper section of the HTTP Client. The response from the endpoint is displayed in the lower section of the page.

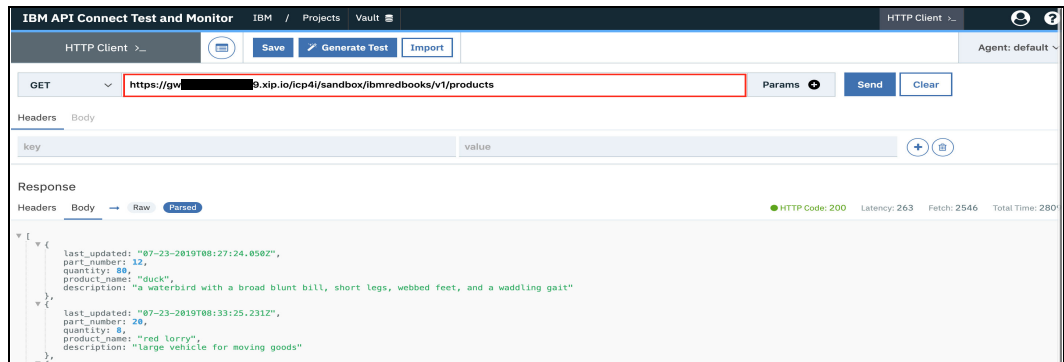


Figure 6-323 Calling an API using HTTP Client

4. Click **Generate Test** icon as shown in Figure 6-324.

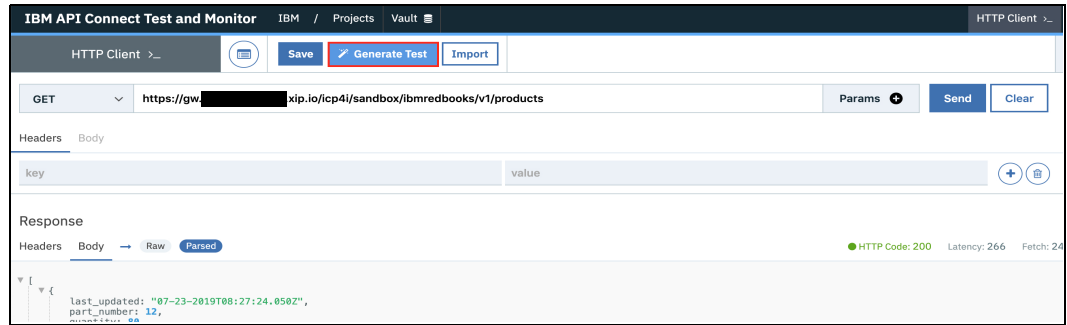


Figure 6-324 Generate Test

- a. Generate Test window pops up. Enter the name of the test to be generated. From the **Save to Project** drop-down menu, select a project name, or select **Create new project** to create your own project. Click the **Confirm** icon to save the test and start the test generation as shown in Figure 6-325.

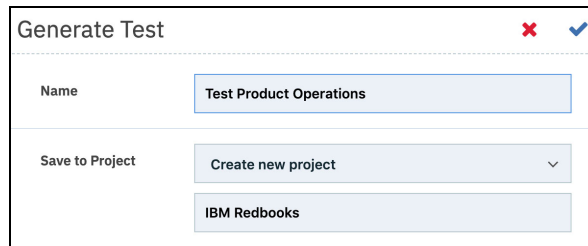


Figure 6-325 Generate Test and the Project

- b. After test generation is complete, the **All set!** page is displayed. Click **Close** to continue as shown in Figure 6-326 on page 401.

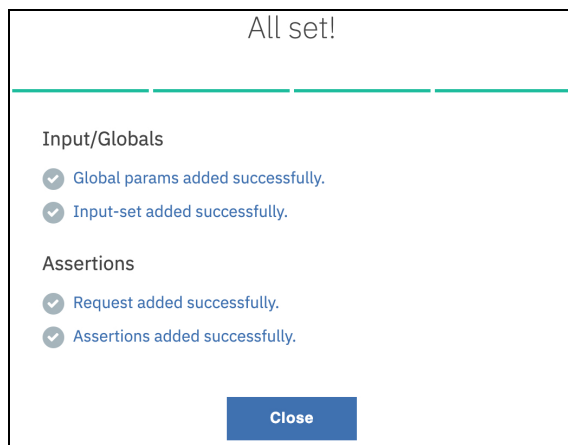


Figure 6-326 Confirmation of Test generation

5. You are now in the test editor, called the **Composer**. In the Composer you can view all the assertions that are generated automatically as shown in Figure 6-327. Click **Save and exit**.

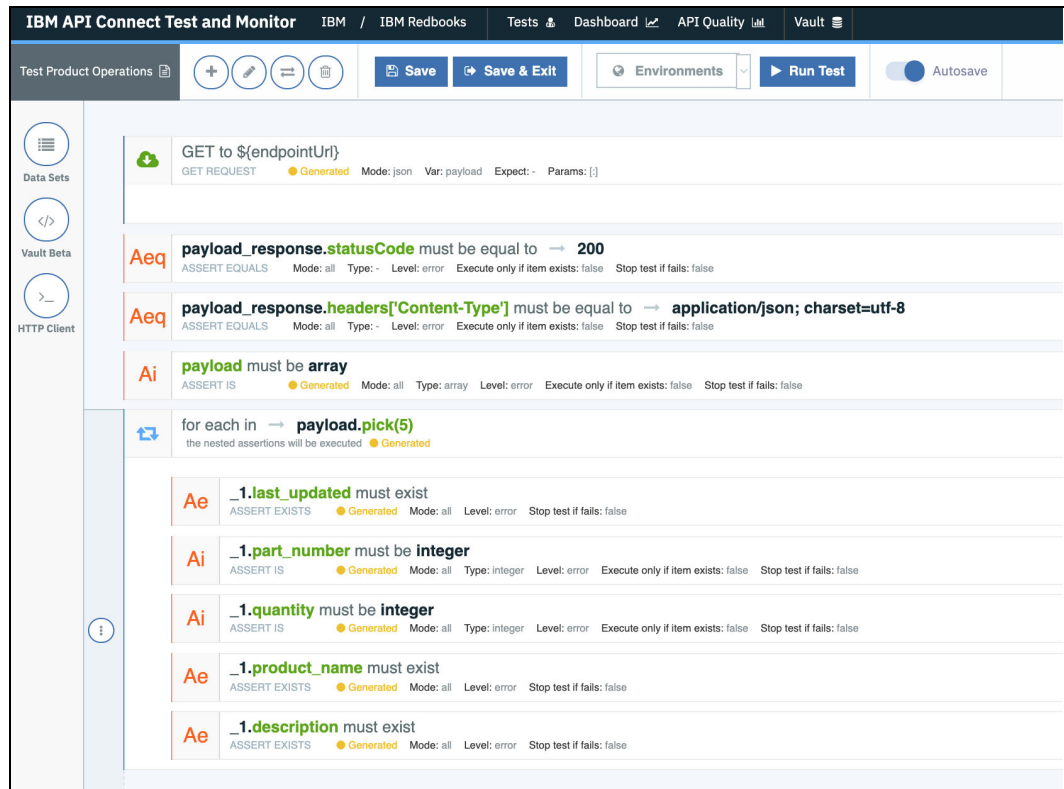


Figure 6-327 Test Composer

Note: The test can be edited by using either the code (text) editor, or the visual (graphical) editor. Toggle between these two editors by clicking the CODE and VISUAL tabs in the upper right of the Composer. The visual editor is the default editor. The code editor is shown in Figure 6-328 on page 402.

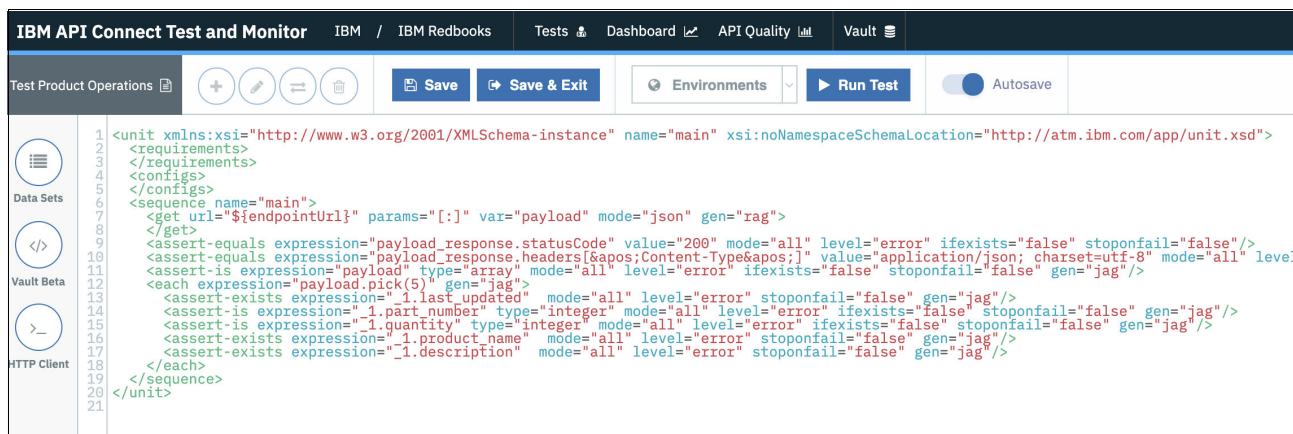


Figure 6-328 Test Code editor

So far you have created a test from an existing API request.

6.12.2 Update the test case from a Swagger file and publish

Generally, APIs are developed in a more agile manner, as the requirements for APIs evolves rapidly. Therefore, updating the test cases for APIs quickly and easily is essential to keep up with chase of agile development approaches. So let us say if the API definition is amended with a new data model or operation, you don't need to re-create the tests from scratch. You simply need to enrich the test case with the new API definition as shown in this section. Complete the following steps to update the test case with a Swagger file:

1. Navigate to **Tests** page, a list of the tests that have been created in the project are displayed. Click **Edit** icon to update the draft test as shown in Figure 6-329.

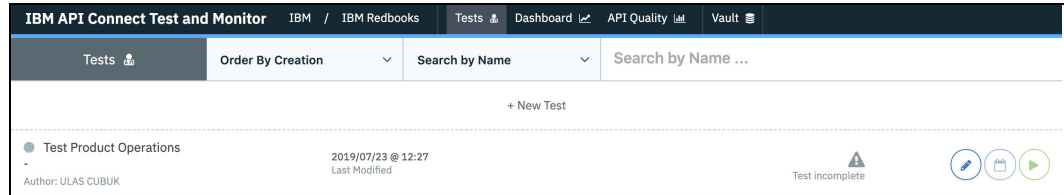


Figure 6-329 Test page

2. Test editor page opens. Click **Build from SPEC** icon to update the test case as shown in Figure 6-330.



Figure 6-330 Build from SPEC

3. Upload the spec file and click **Save** icon as shown in Figure 6-331 on page 404.

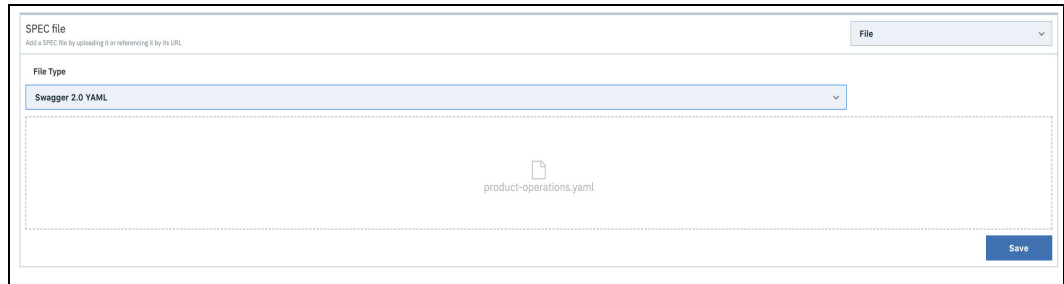


Figure 6-331 Upload the spec file

4. Select the **API operation** from the drop-down box and click **Merge** icon to amend the changes as shown in the next figure. Test Composer page opens.

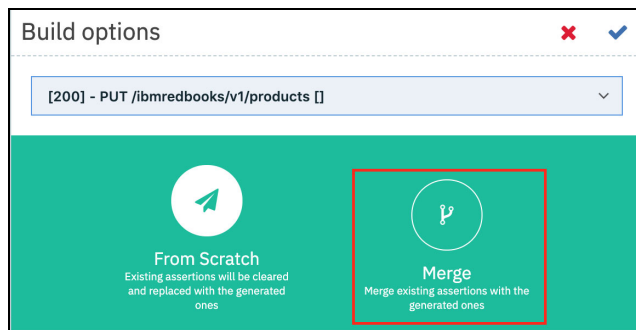


Figure 6-332 Build from SPEC options

You can repeat this step to merge more operations exposed by the API into the existing test case.

5. Expand the **Data Sets** panel from the left and add a new global parameter for test data. When the test was generated, it extracted the values that you provided into a variable as shown in Figure 6-333 on page 405.

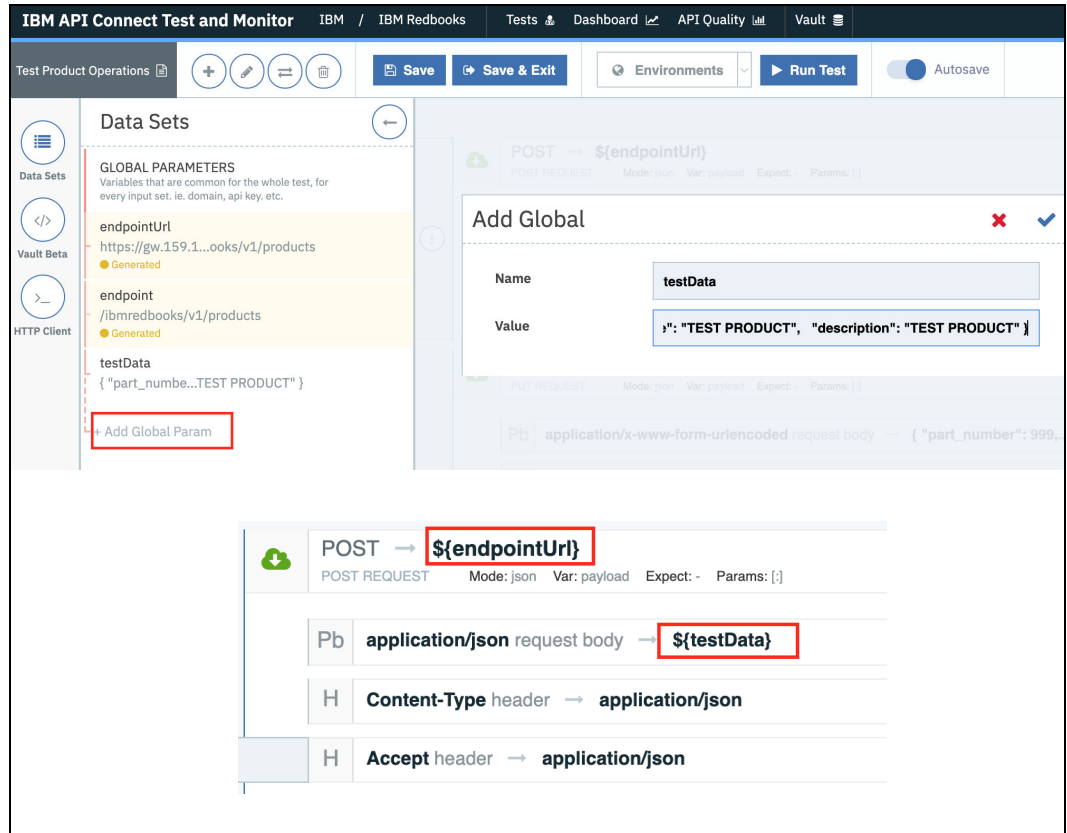


Figure 6-333 Global Data Sets

- Rearrange the test components to be executed in the correct order. As shown in Figure 6-334 on page 405, you first inject a test data, then update and retrieve it, and lastly you delete it.

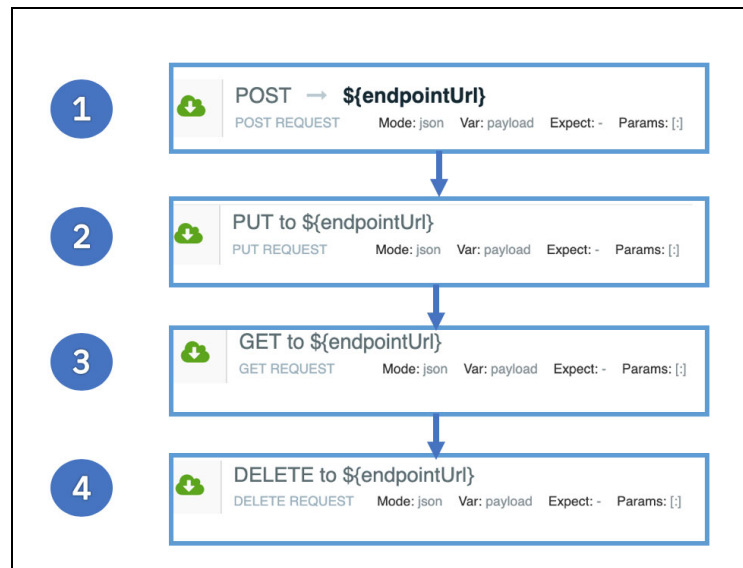


Figure 6-334 Test execution order

- Click **Run Test** icon to confirm that it works correctly as shown in Figure 6-335.

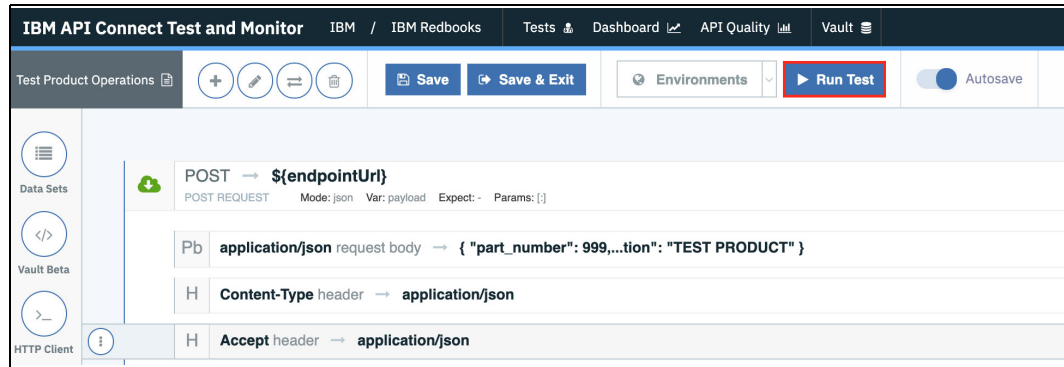


Figure 6-335 Run the test

- a. If there was an error, the test report that is generated can help you diagnose the error. An example error report as shown in Figure 6-336 on page 406.

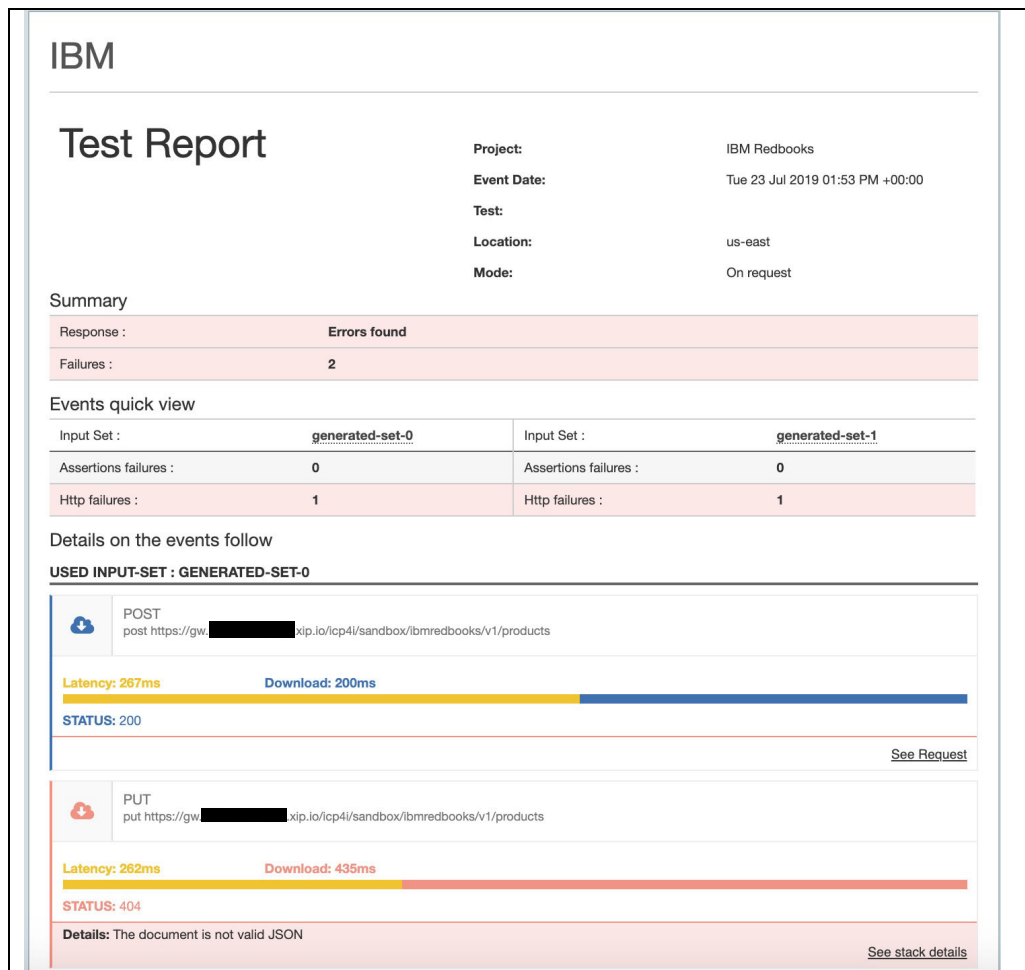


Figure 6-336 Error Report example

- b. If the test passes, the test report generated would again show the details of each execution step.
8. After you have completed the configuration of your test, click the **Publish** icon to publish the test as shown in Figure 6-337 on page 407.



Figure 6-337 Publish Test

Note: You need to publish a test to be able to schedule it to run automatically. After you have published a test, you can continue to work on the test in the Composer without affecting the published test. Later, you can publish the test again to update the test that you previously published.

9. After the test is published, you can navigate to **Tests** page and click the **Run** icon to trigger the test as shown in Figure 6-338 on page 408.

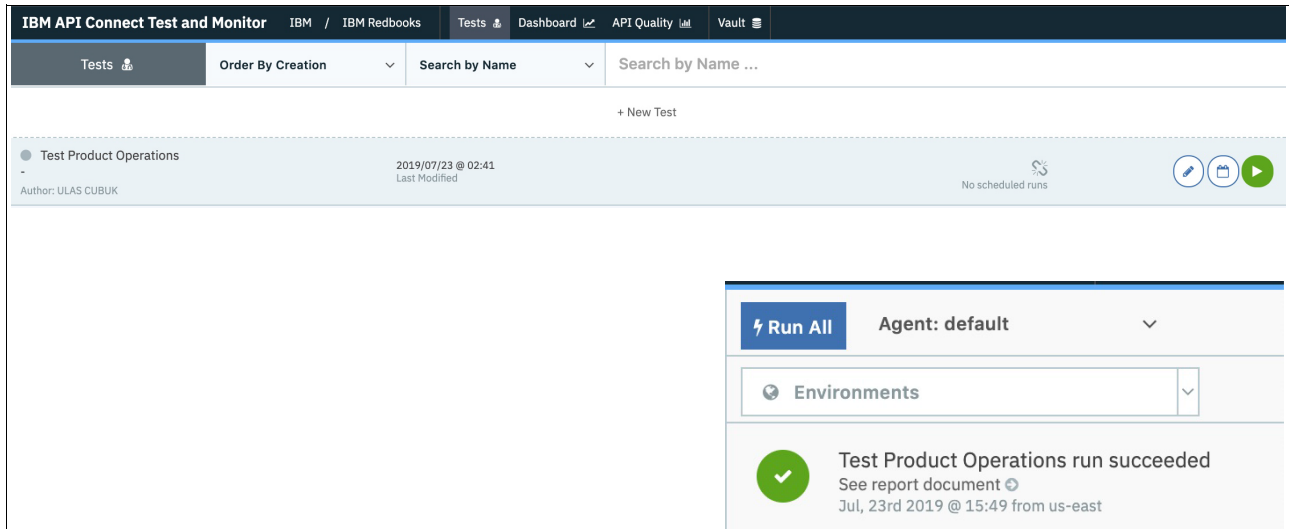


Figure 6-338 Tests page

6.12.3 Gain insights into API quality

Dissemination of test results with the interested parties and having historical and comparable view on the test data is as important as performing the test itself. Without the key understanding of the test results, any effort to fix the defects would be pointless. IBM API Test and Monitor provides a dashboard to view all the individual test run results. That way, you can use the failure cases as the basis of diagnosis. Also it generates functional and performance test reports to give you insight into quality over time. Follow these instructions to view test results and API quality reports:

1. Navigate to **Dashboard** page to view test results. Dashboard provides three different views to analyze your test results:

Logs View show the status of the test runs where you can add filters based on timeframe, tag, location, and success/failure. You can also reach the test report of each individual test case and share the results with a wider audience.

- a. Select the **Logs View** as shown in Figure 6-339 on page 408.

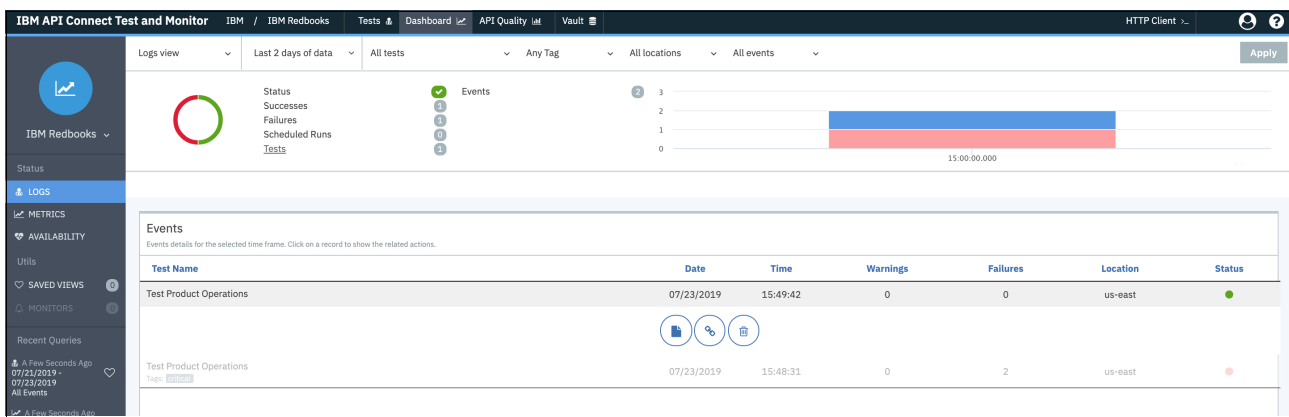


Figure 6-339 Test logs

- b. Metric View shows the footprint of individual test requests for the selected timeframe, endpoint, and location. Select the **Metrics view** as shown in Figure 6-340.

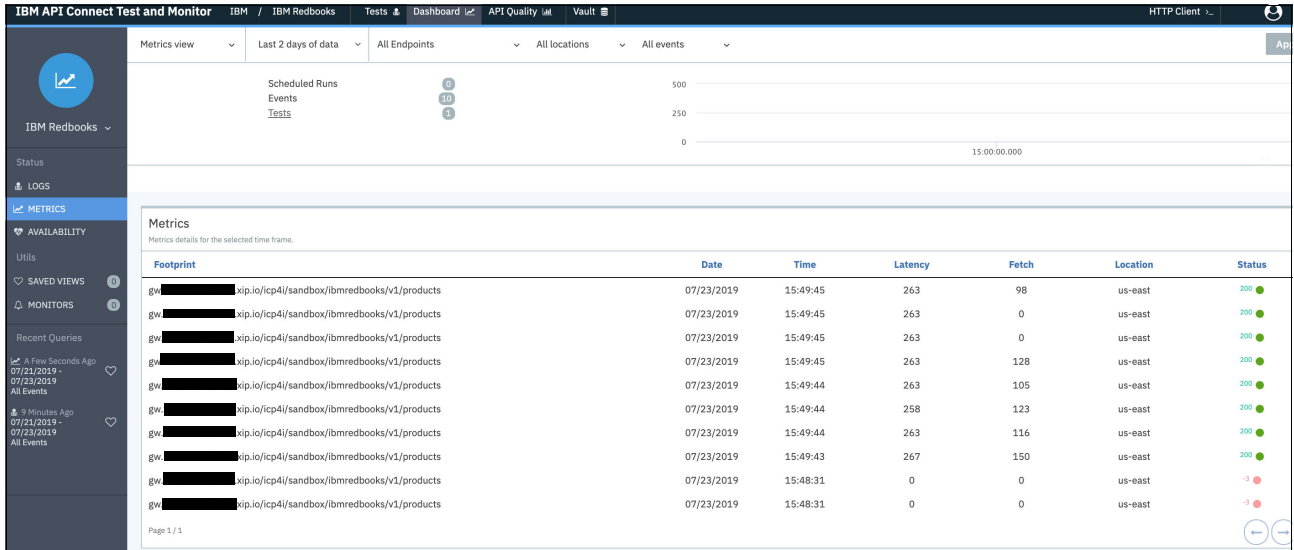


Figure 6-340 Test metrics

c. Availability View shows the availability details of the API Endpoints for selected timeframe and location. Select the **Availability view** as shown in Figure 6-341 on page 409.

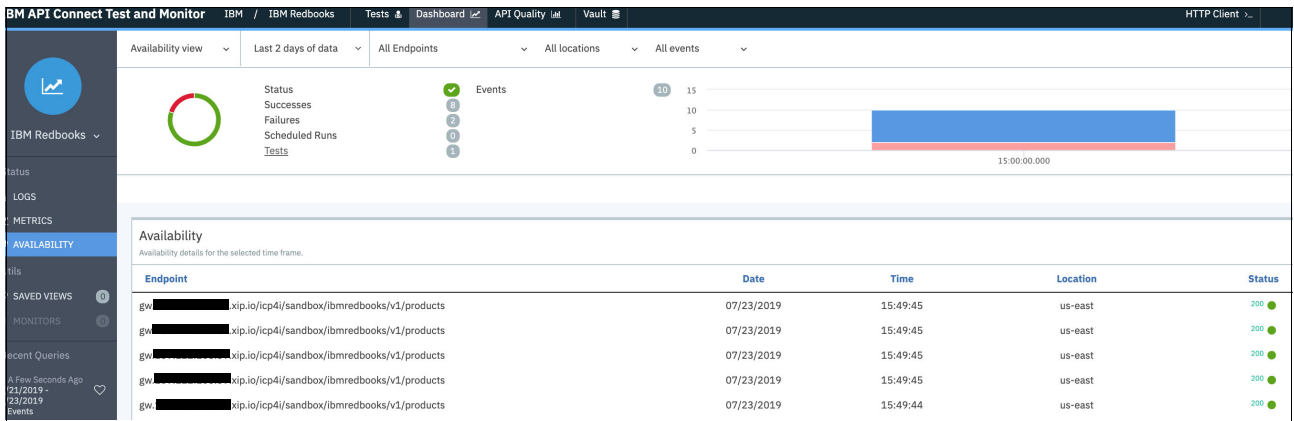


Figure 6-341 Test availability

2. Data on API quality can be used as an indicator of API consumption. Analyze the uptime, performance, and failures of the API based on all the test runs to gain actionable insights and diagnose errors. Navigate to **API Quality** page to view the functional and performance reports.

a. Select **Functional view** as shown in Figure 6-342 on page 410.

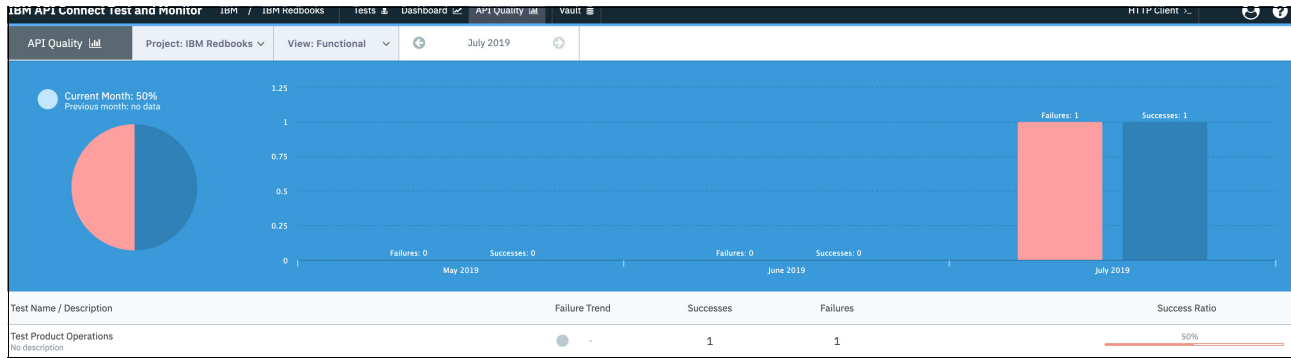


Figure 6-342 API Quality Functional view

b. Select **Performance** view as shown in Figure 6-343.

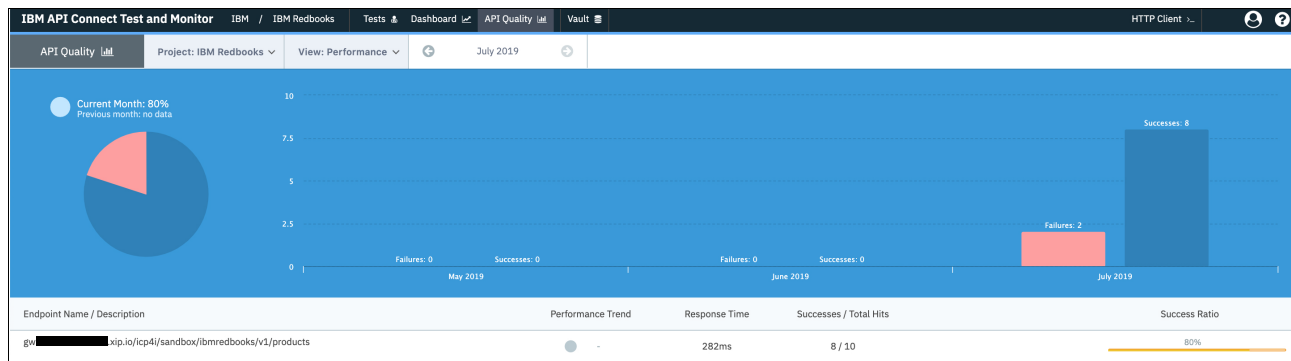


Figure 6-343 API Quality Performance view

6.13 Large file movement using the claim check pattern

A messaging-based architecture, at some point, must be able to send, receive, and manipulate large messages. Such messages may contain anything, including images (for example, video), sound files (for example, call-center calls), text documents, or any kind of binary data of arbitrary size.

Sending such large messages to the message bus directly is not recommended, because large messages can also slow down the entire solution. Also, messaging platforms are usually fine-tuned to handle huge quantities of small messages. Finally, many messaging platforms have limits on message size, so you may need to work around these limits for large messages.

The claim check pattern is used to store message data in a persistent store and pass a Claim Check to subsequent components. These components can use the Claim Check to retrieve the stored information. For a real-world example of this, think of flight travel. When you check your luggage, you receive a claim check. When you reach your destination, you hand in only the claim check with which you can reclaim your luggage.

In this scenario we will be implementing a claim check pattern for the sending and receiving of a large file. Basically, this pattern means splitting a large message into a claim check and a payload. The claim check is sent to the messaging platform (in this case IBM App Connect) and the payload is stored by an external service (in this case IBM Aspera on Cloud).

6.13.1 Build the file transfer

As mentioned, for the file transfer portion of this scenario we will use IBM Aspera on Cloud. IBM Aspera on Cloud enables fast, easy, and secure exchange of files and folders of any size between end users, even across separate organizations, in both local and remote locations. Importantly for our scenario, using the Aspera on Cloud service means that the recipient of the file does not need to be running an Aspera Transfer Server, but is sent a link with which to retrieve the file they have been sent in an email-like workflow. This link will form the basis of our claim check.

Sign up for a free trial

Perform the following steps:

1. If want to build this scenario for yourself, you will need to sign-up for a free trial of IBM Aspera on Cloud. Use this following link and sign up for the free trial:

<https://www.ibm.com/cloud/aspera>

See Figure 6-344.

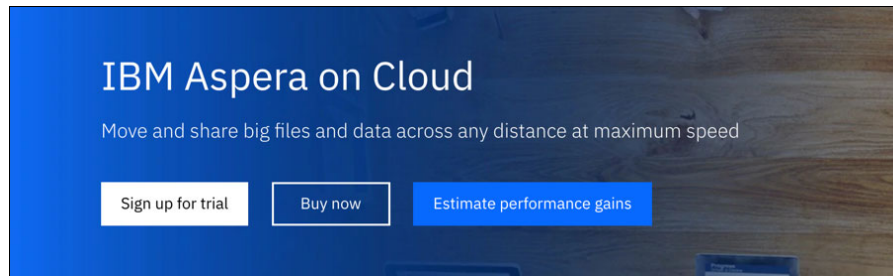


Figure 6-344 Sign up for trial

2. After you have received your notification, log on and follow the instructions to download and install IBM Aspera Connect (This will allow transfer of files via the browser interface). See Figure 6-345.

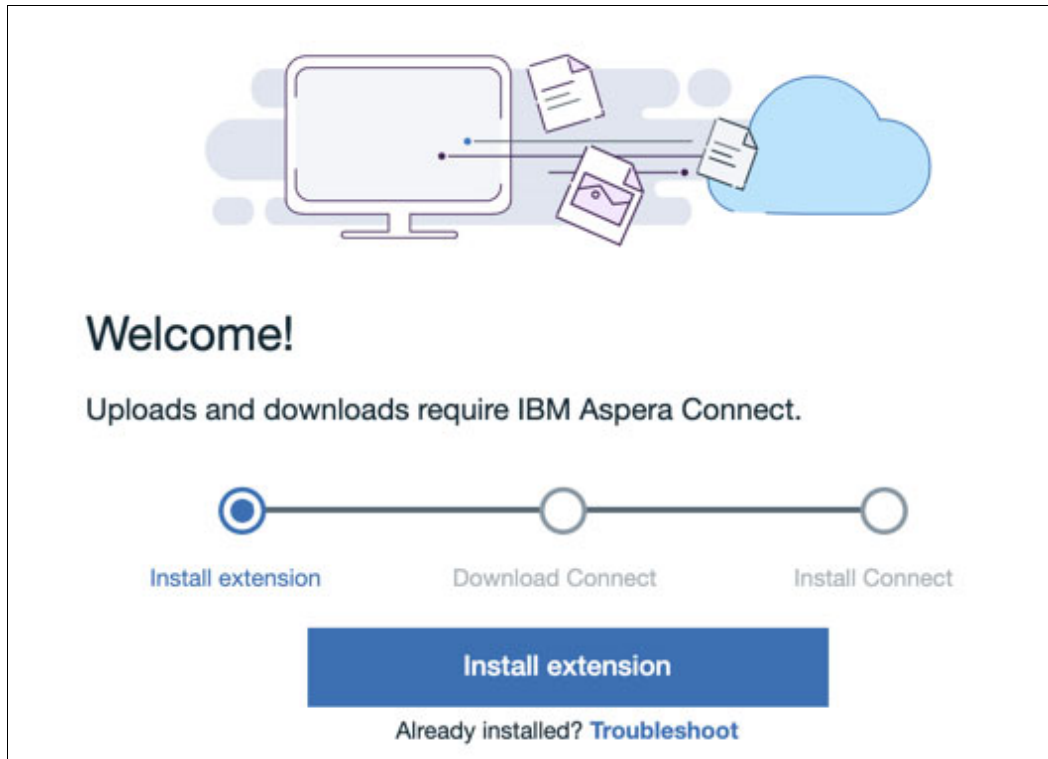


Figure 6-345 Install IBM Aspera Connect

3. After this has been completed you will see the home page. Here, you may select which of the apps you want to be your permanent landing area (For this scenario we chose **Files**). See Figure 6-346.

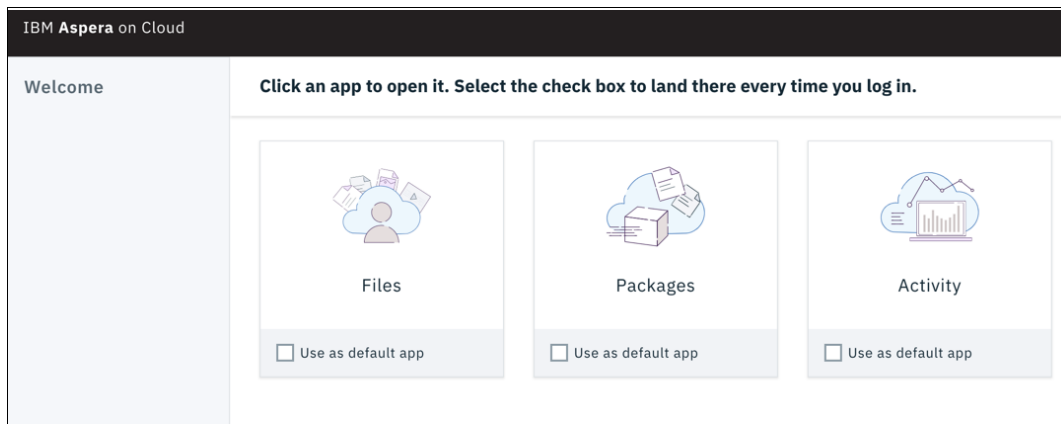


Figure 6-346 Choose a default app

4. To navigate between one app and another, click the box of dots in the upper right of the screen. See Figure 6-347 on page 412.



Figure 6-347 Navigation

As shown in Figure 6-348, this will show the apps that you have access to.

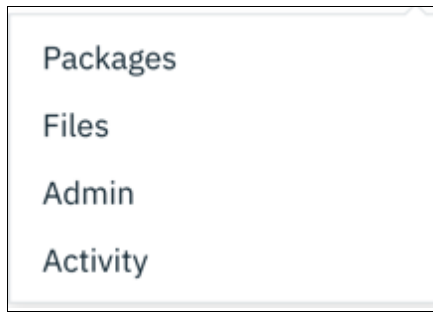


Figure 6-348 Apps

5. Select the **Admin app** and on the left side, select the **Workspaces** section. For our scenarios we will create a new workspace. The workspace is a collaborative arena for those working together, on a given project, for example, or perhaps in a department or division.

The workspace also acts as a security realm. Workspace members can send and share content freely to and with members of the same workspace. For our simple example, we could simply use the Default Workspace, but let us create one for the book.

6. Select **Create new**.

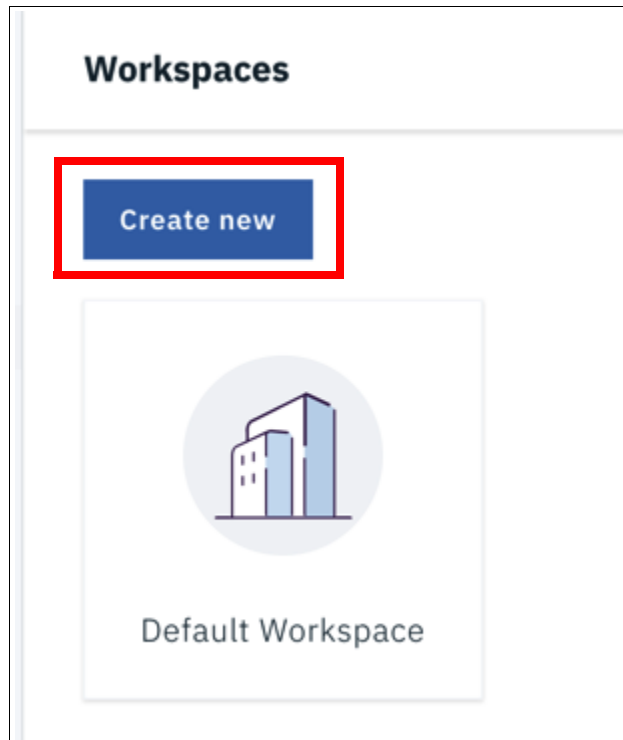


Figure 6-349 Create new workspace

7. We need to give the workspace and name (IBM Redbooks) and if you like, upload a logo for the workspace (we have used the IBM Redbooks logo). You will also see the default node that the files will be stored in (do not change this). See Figure 6-350.

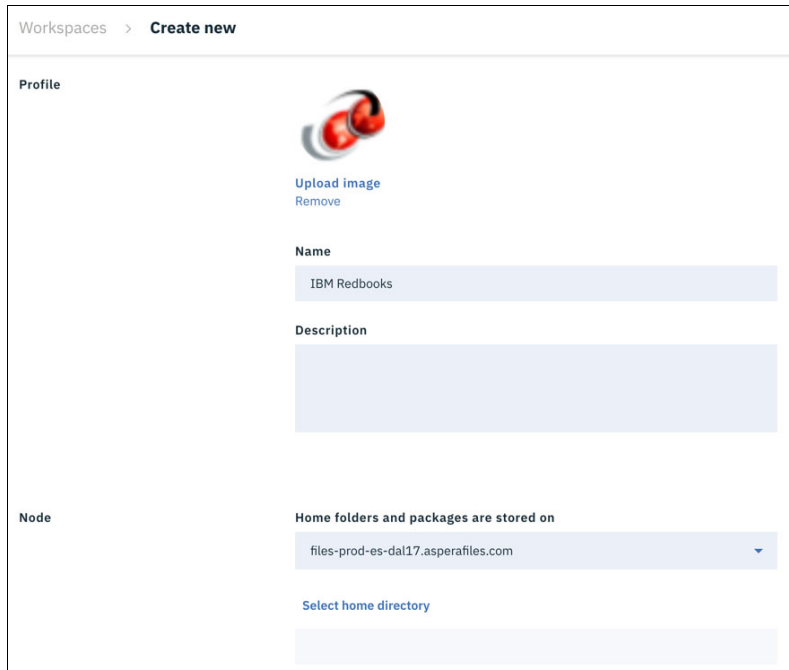


Figure 6-350 Redbooks workspace

8. Click the **Applications** tab and ensure that the Files and Packages are shown as active.

In the Packages app, a package (also called a digital package) is a collection of digital assets (such as files, folders, video, images). You gather them to send to one or more individuals or user groups, or to an Aspera on Cloud shared inbox. You can also attach files and folders in your Files app as you create a digital package in the Packages app. Simply right-click the items in your Files app to include in the package and select the option to send. You can add content to a package you initiate from the Files app. See Figure 6-351.

In our scenario we will be sending only a single file through the File application.

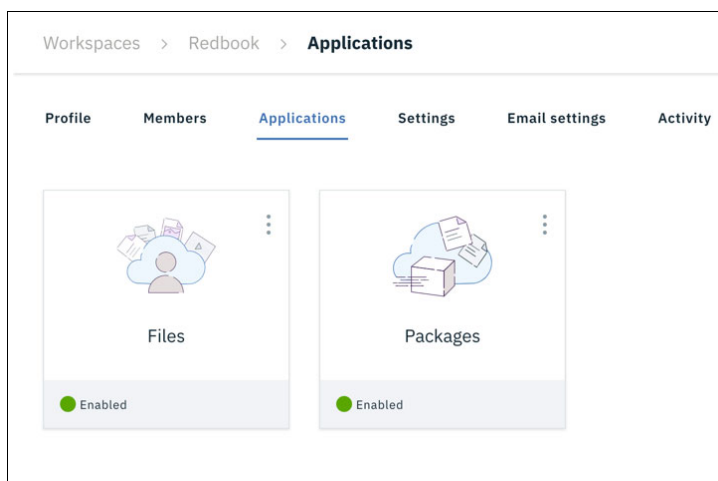


Figure 6-351 Applications

9. Now click the **Settings** tab. We will not be changing any of the settings but there is one here that we need to take notice of for later in the scenario. See Figure 6-352.

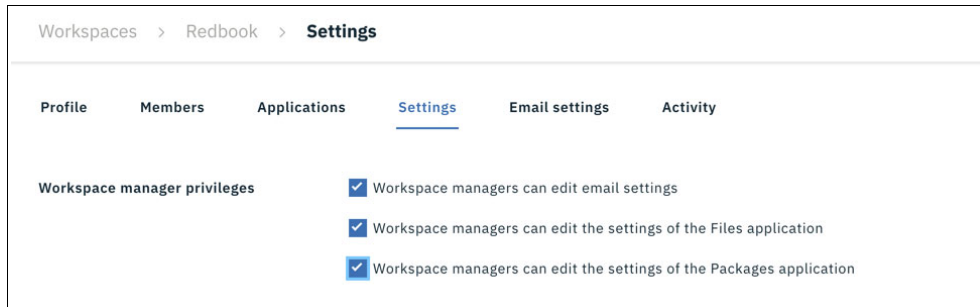


Figure 6-352 Settings

10. Click **Email Settings and Templates**. Notice that the Subject prefix is IBMAsperaOnCloud. This will come into play in the event flow part of the scenario. See Figure 6-353.

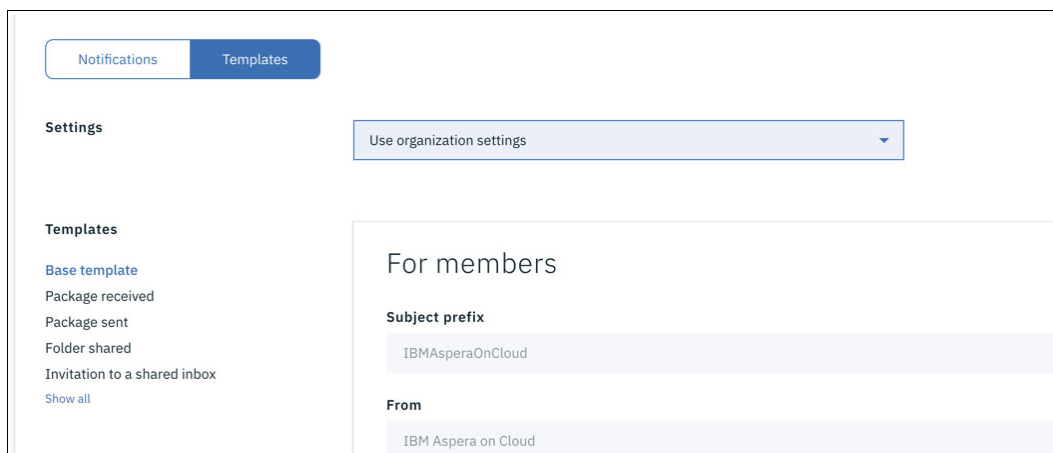


Figure 6-353 Email templates

11. Using the navigator, move to the Files application. Click the **IBM Redbooks** workspace. See Figure 6-354 on page 415.

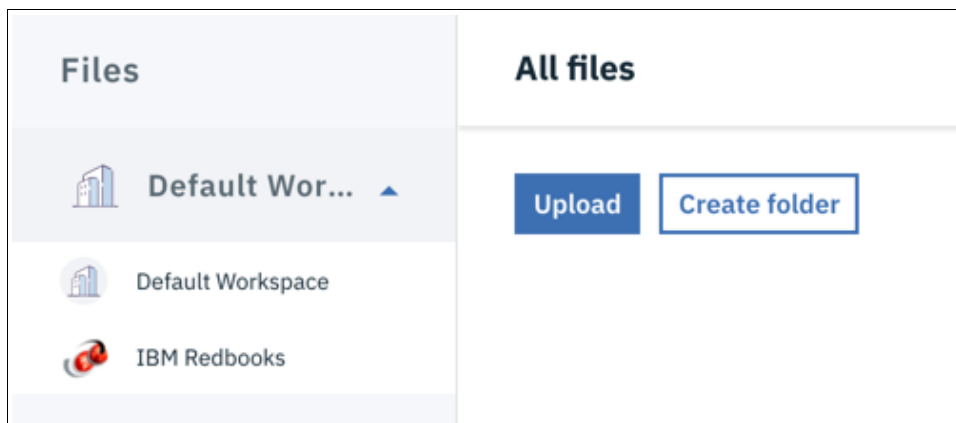


Figure 6-354 Files and workspaces

12. We will create a folder to contain our files for transfer. Click **Create Folder**. Enter a name for the new folder (we chose **forTransfer**). See Figure 6-355.

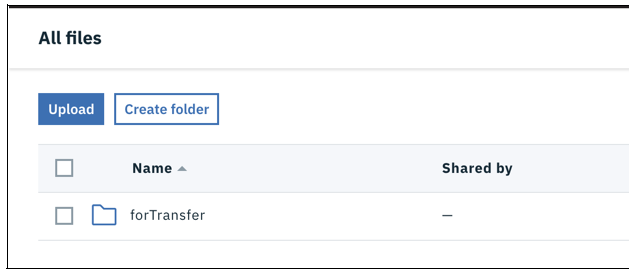


Figure 6-355 New folder

We are now ready to upload the file and send it to the recipient.

Upload a file

Perform the following steps to upload a file:

1. Click **New folder** to open it.
2. Select **Upload** → **Files**. See Figure 6-356.

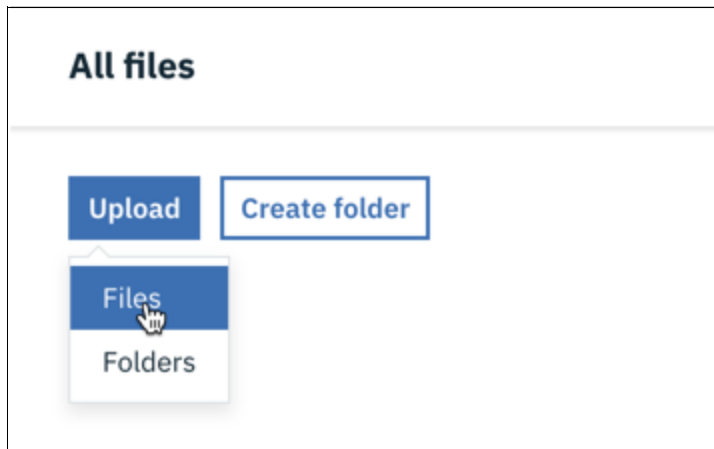


Figure 6-356 Upload files

3. This will open a pop-up window (Figure 6-357 on page 417), where you can select the file to be transferred. Which file that you decide send in not important for this scenario, but a file of a decent size would be ideal.

Tip: If you are using a Mac, you may find that the window to select the file is in the background.

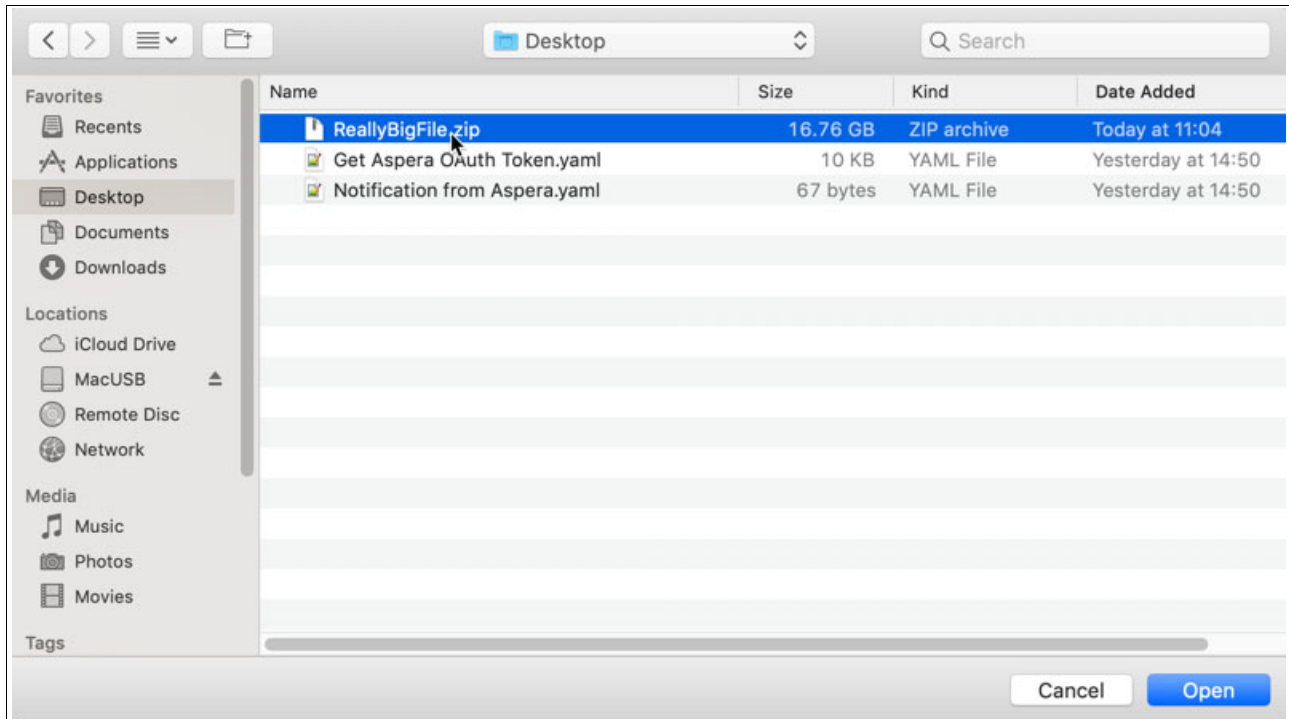


Figure 6-357 Select a file to upload

4. The upper right area of Figure 6-358 shows that there is one active transfer. (This is your upload.)

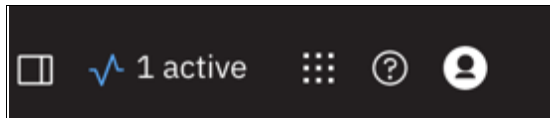


Figure 6-358 Active transfer

5. Click the **Transfer icon** to see the current progress of the upload.

We can see the activity of the upload by clicking the dotted box again and selecting **Activity**.

On this dashboard we see information about all of the transfer, but specifically the current transfer in progress.

Tabs on the left side of this dashboard provide good information for our scenario regarding volume usage and file access available. See Figure 6-359 on page 418.

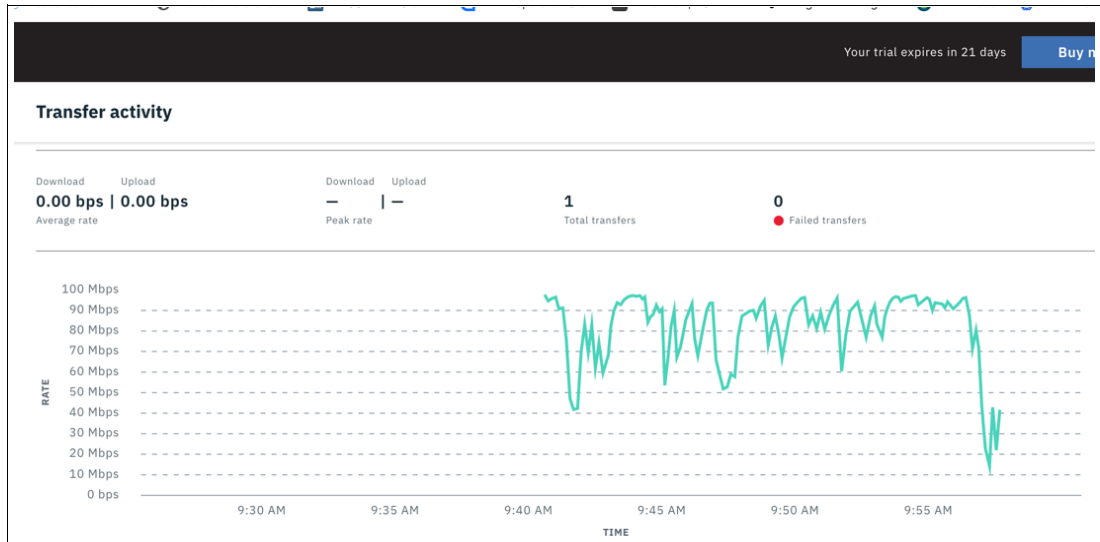


Figure 6-359 Upload transfer activity

- Go back to the **Files** application. Click the **IBM Redbooks** workspace and drill down to the current file to see the progress. After the file has been successfully uploaded, you can see the various options available for processing of this file. See Figure 6-360.

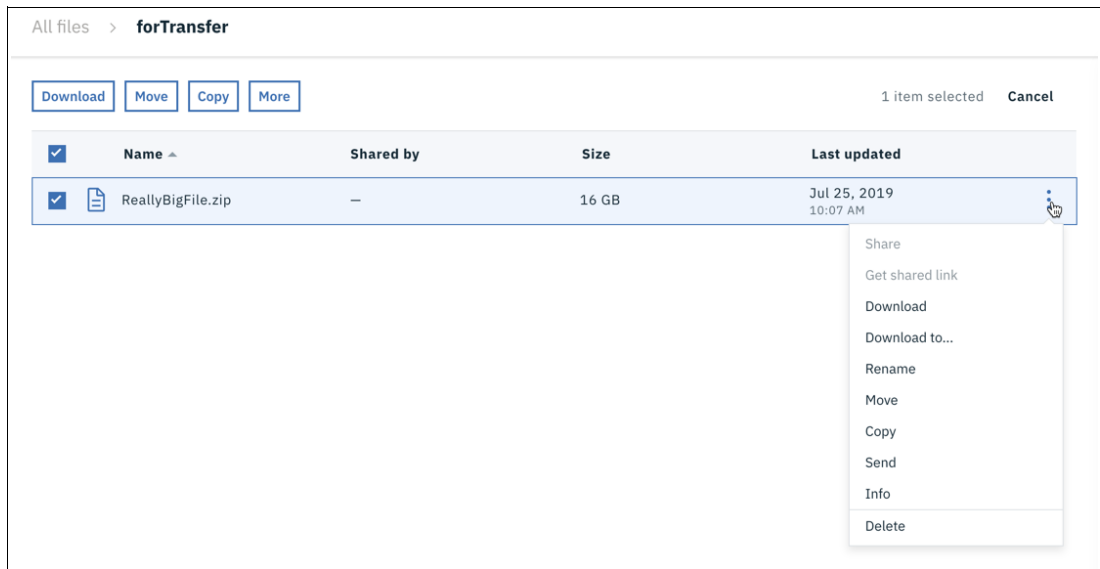


Figure 6-360 Options for the file

We are now ready to send the file to the intended recipient.

Send a file

If this scenario, we send the file to a Gmail address. We plan to use the Gmail application connector in IBM App Connect in the next part of the scenario.

If you want to build the second half of the scenario but do not have a Gmail account, get one now.

- Using the options menu that was shown earlier, select **Send**.

A pop-up window will appear where the details of the file and recipient are to be entered.

2. Enter the Gmail address of the intended recipient, add a subject that will let the recipient know what has been sent. As shown in Figure 6-361 the file should already be listed there.

Tip: If you select the wrong file and the file is not the one you intended, simply remove that file (using the x) and drag and drop the correct file from the list of uploaded files.

3. Optionally, you can add a message for the recipient (in our scenario, we added a message to remind the recipient that the file is large). Part of the claim-check pattern means that the recipient is alerted to the arrival of the file (that is, they receive the claim check). However, it may not be the most suitable time for them to download such a large file. They may want to process the file at a more appropriate time.
4. Additionally, you can choose to password protect the file (if it contains sensitive information) and request that someone is notified if the package is available and/or downloaded). These are in the Options section, but we will not be using them for our scenario.

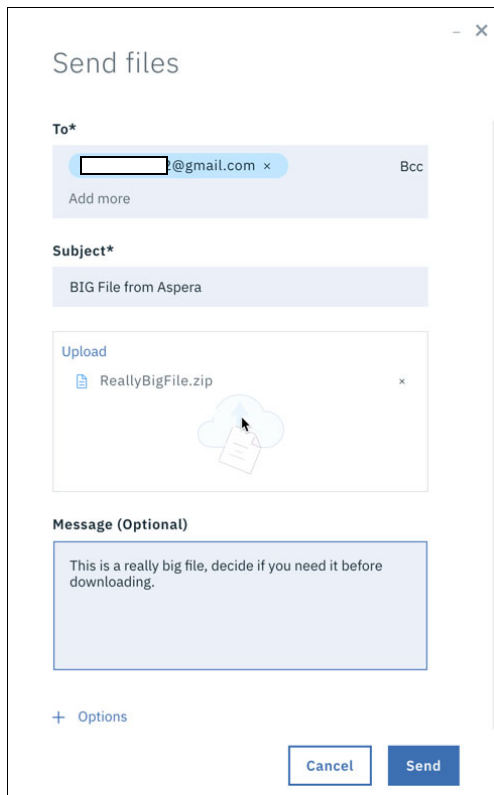


Figure 6-361 Send the file

5. Click **Send**.
6. If we now view the Activity monitor again (via the indicator in the upper section of the screen), we see that the file transfer is being processed. See Figure 6-362 on page 420.

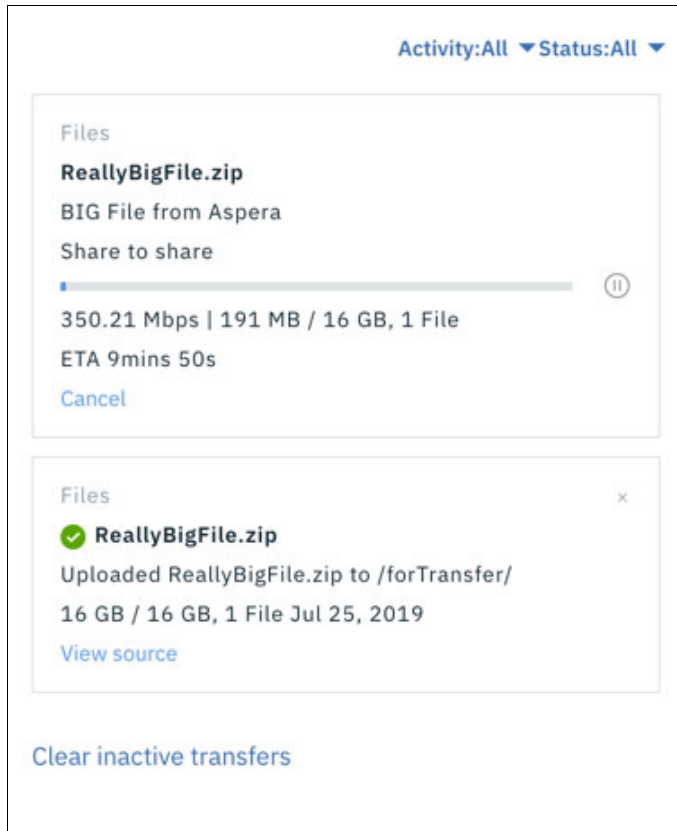


Figure 6-362 File transfer being processed

When the transfer has been completed, we will see an email in the inbox of the recipient.

6.13.2 Build an event-driven flow

In this part of the scenario we will be using IBM App Connect. For many people, simply going and reading email periodically may be a standard way to work. However, as you build out your new digital infrastructure to respond to ever-changing customer demands, you need to be able to capture new events, contextually and in near real time. By orchestrating the right follow on actions through workflows, you immediately react to new events. In this way, you ensure that your customers and employees have a positive experience.

IBM App Connect allows you to utilize smart connectors to capture events from your systems designed on event-driven architectures. Unique capability and situational tooling then empowers your teams that have the context to apply such data. These teams can rapidly build and change integrations as their needs shift, enabling the business to move at the speed of its customers.

Imagine that the file that we sent in 6.13.1, “Build the file transfer” on page 411 was an urgent file that is required to resolve a customer situation. It would be ideal if the customer service representative could automatically be alerted to the fact that the file has arrived and could immediately take respond.

Sign up for a free trial

In this part of the scenario we will be utilizing two of the smart connectors in IBM App Connect. As you will have read previously, there are over 100 to choose from.

If you have not already done so, sign-up for a free trial on the IBM Cloud.

In 6.13.1, “Build the file transfer” on page 411, you signed up for a Gmail account. You also need to sign up for a Slack trial account if you do not already have one.

Slack is a collaboration hub where you and your team can work together to get things done.

You can sign-up for a trial account here:

<https://slack.com>

You will need to define the following elements:

- ▶ **Slack workspace:**
A workspace is a shared hub made up of channels, where team members can communicate and work together.
- ▶ **Slack channel:**
In Slack, work happens in channels. You can create channels based on teams, projects or even office locations. Members of a workspace can join and leave channels as needed. You can create a channel that has you as the only member, for receiving your file notifications.
- ▶ **Slack application:**
Apps and integrations are the tools that will help you to bring your existing workflows into Slack.

The Slack App Directory has thousands of apps that you can integrate into Slack. See Figure 6-363.

IBM App Connect is one of these apps. Instructions to create the app can be found at:

<https://get.slack.help/hc/en-us/articles/360001537467-A-guide-to-apps-and-the-App-Directory>

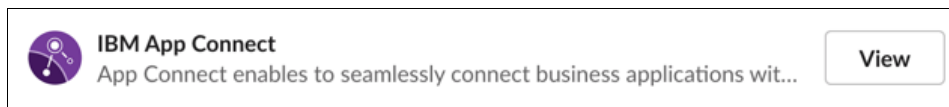


Figure 6-363 Integration into Slack

After you have your Gmail, Slack and IBM App Connect service on the IBM Cloud, we are ready to build the event-driven flow.

Build the flow

Perform the following steps to build the flow:

1. Log on to the IBM App Connect service on the IBM Cloud.
To enable IBM App Connect to interact with the applications (Gmail and Slack) you need to first connect your credentials for these accounts. (This can also potentially be done as you build the flow, but it is easier to get it out the road up front).
2. Go to the **Catalog** tab and select **Applications**. Here you can either scroll down to find the Gmail application or type in a filter for it. See Figure 6-364 on page 422.

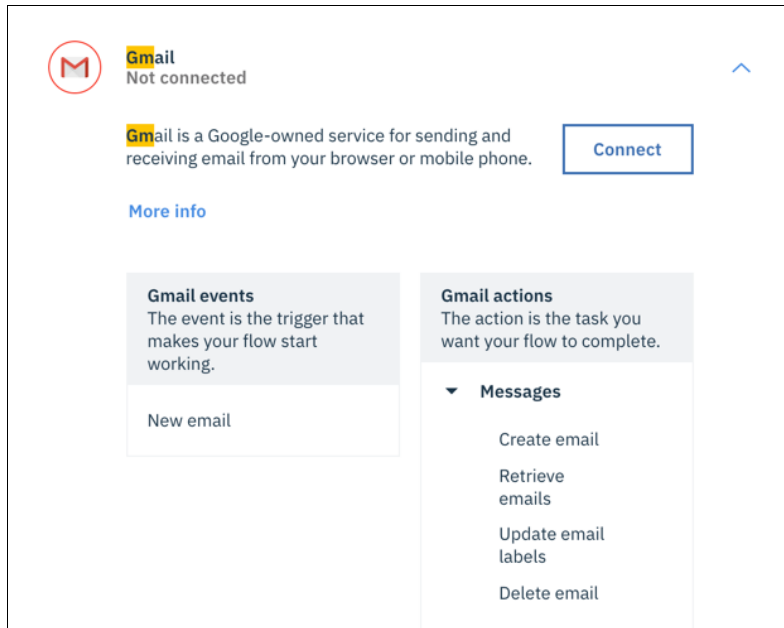


Figure 6-364 Connect Gmail

3. Click **Connect**, follow the instruction to log on to Google (if you are not already logged on). See Figure 6-365.

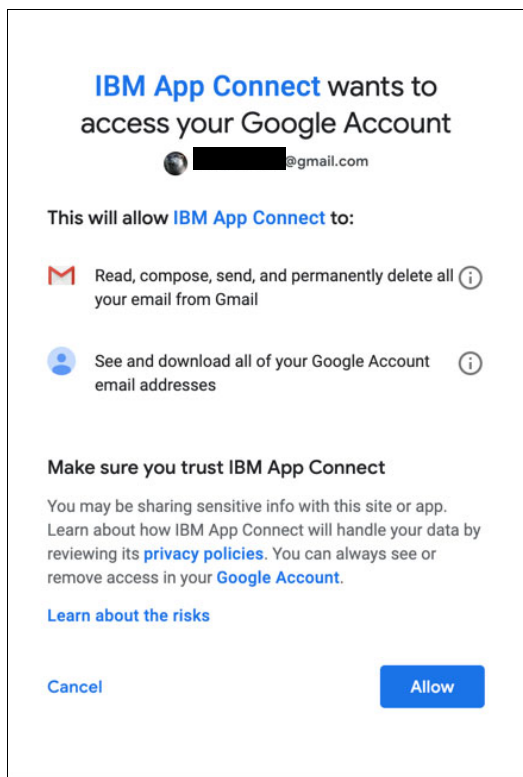


Figure 6-365 Connect account

4. Allow IBM App Connect access to your Gmail account.

After this has been completed, you will see a pop-up message that your account is connected and you can check the details. See Figure 6-366 on page 423.

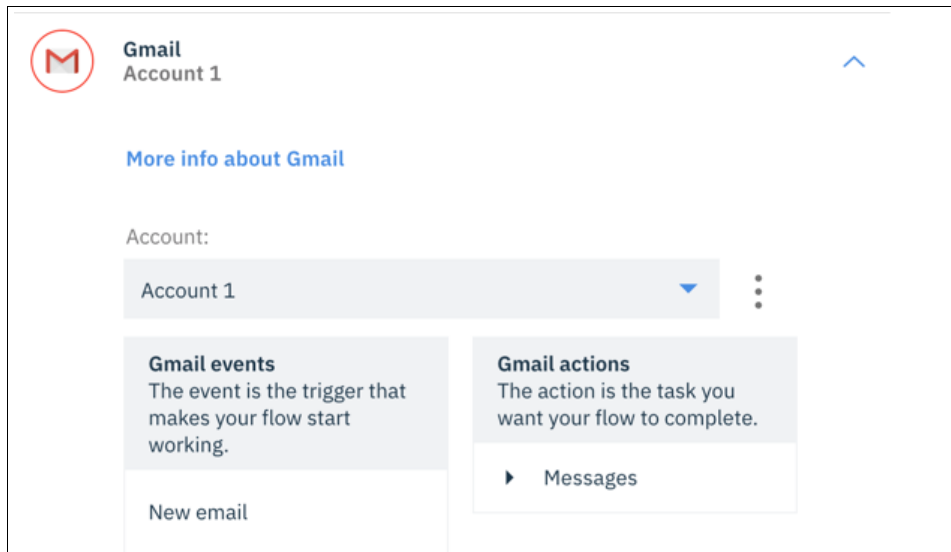


Figure 6-366 Connected Gmail account

5. Repeat the procedure for your Slack information, and agree to connect. See Figure 6-367.

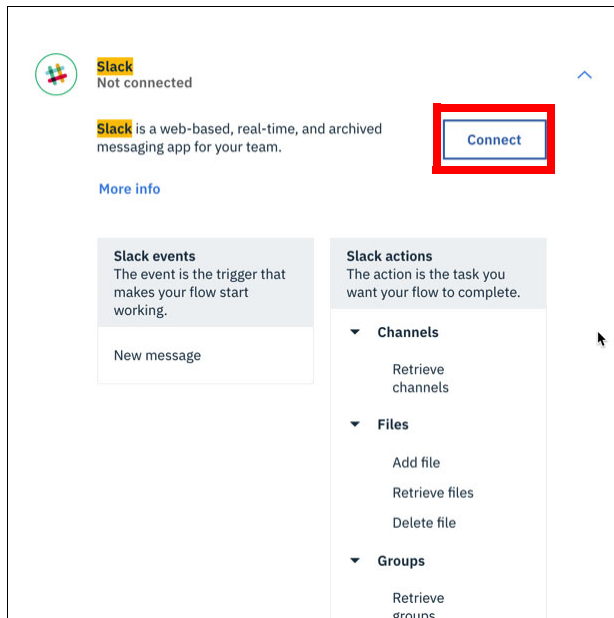


Figure 6-367 Connect Slack

6. You will, again, receive a pop-up indicating that your account is connected. See Figure 6-368 on page 424.

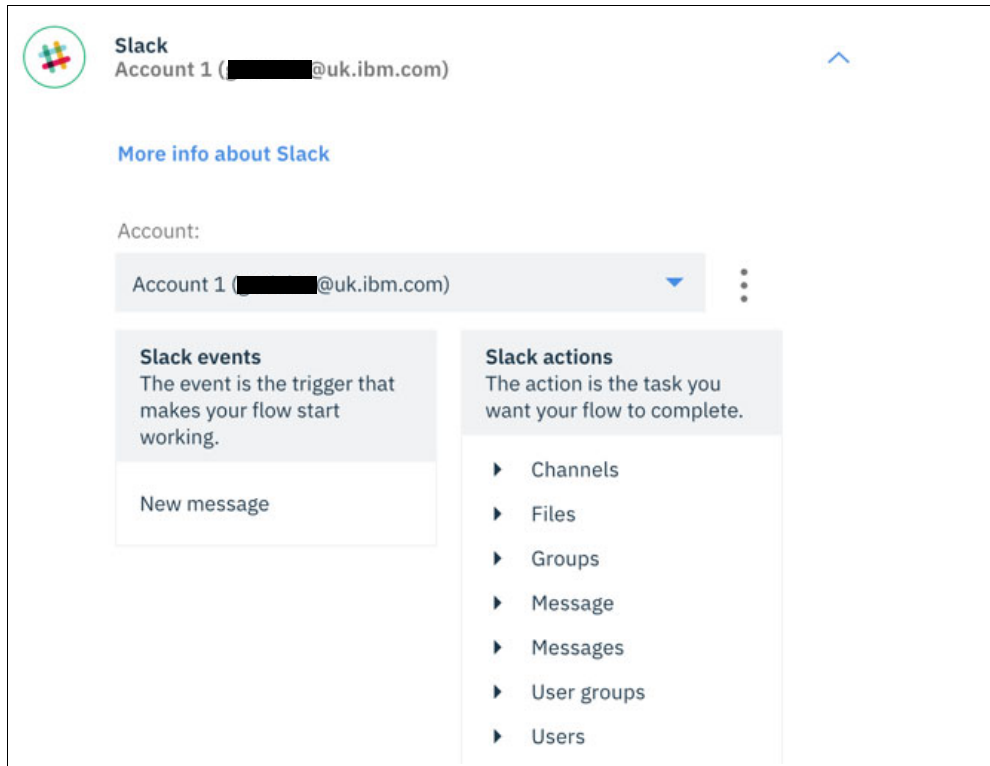


Figure 6-368 Slack account connected

7. Click to go back to the **Dashboard**.

As we mentioned, we will be building an event-driven flow. That is, a flow that kicks off in response to something noteworthy happening in an application.

8. Select **New** → **Event-driven flow**. See Figure 6-369.

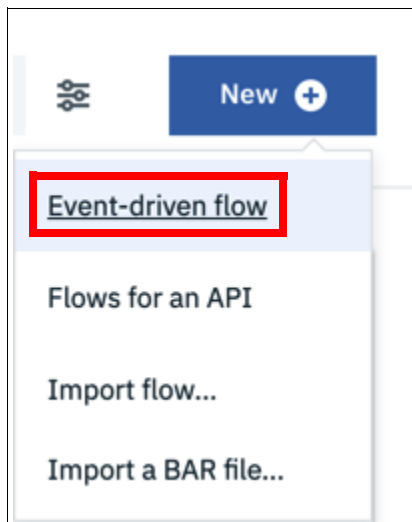


Figure 6-369 New event-driven flow

9. A pop-up will appear for us to build the flow. First, you add the application where the event occurs (in this case, the event is the arrival of an email).

10. Select the **Gmail** application. See Figure 6-370 on page 425.

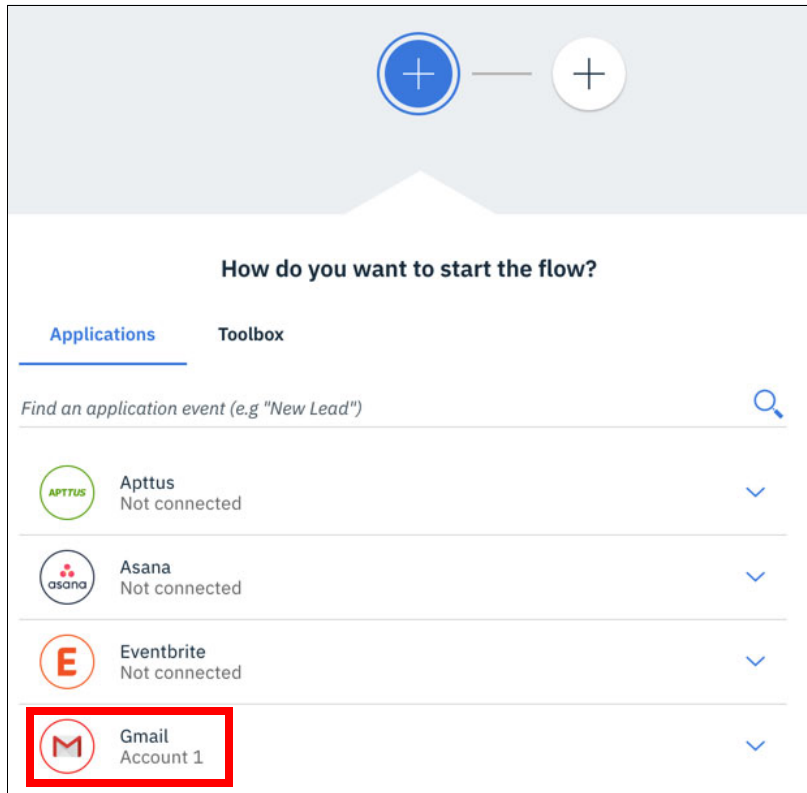


Figure 6-370 How do you want to start the flow

11. We need to select the Account (you may have only one Gmail account connected, so it will default to this account) and the application event (in this case, it is the arrival of a new email).

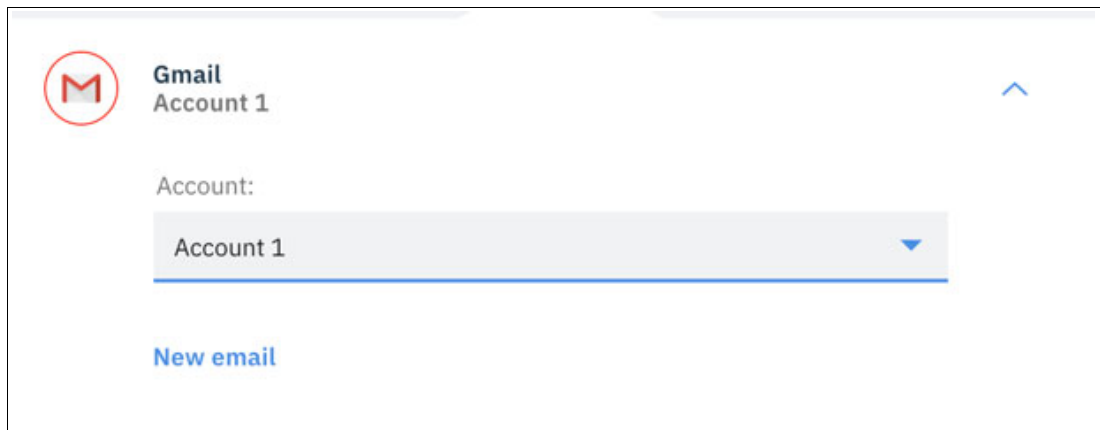


Figure 6-371 Configure event parameters

12. We then need to define what happens next after the event has been detected and received. In this case, simply putting a message to Slack is a good start.

13. Click on the **blue circle** and select **Application**. After the event has been received, you may choose to do any number of other before passing on to the destination application things. For example, you could invoke an API or use any of the utilities in the toolbox to manipulate or filter the data. More on this later.

14. Select the **Slack** application. We now see the various operations that can be performed in Slack. In this scenarios, we will be creating a message that is sent to Slack. See Figure 6-372.

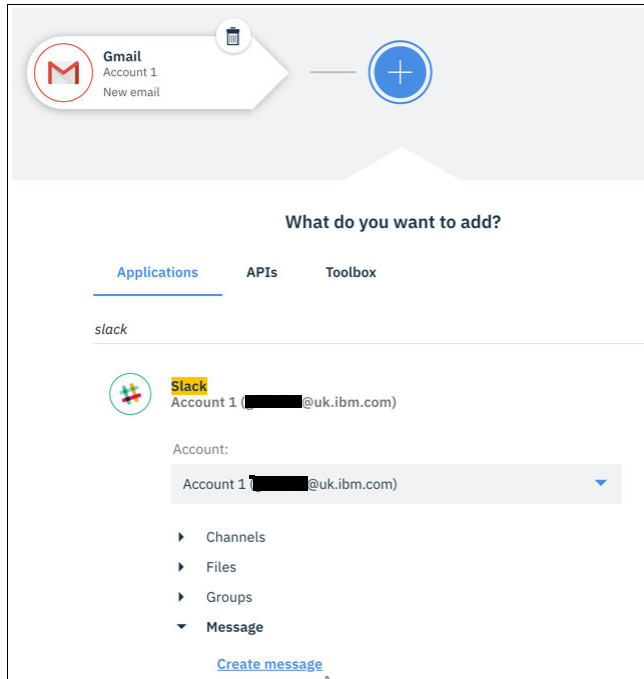


Figure 6-372 Create message in Slack

We now have the source event application and the target application in the flow as shown in Figure 6-373.

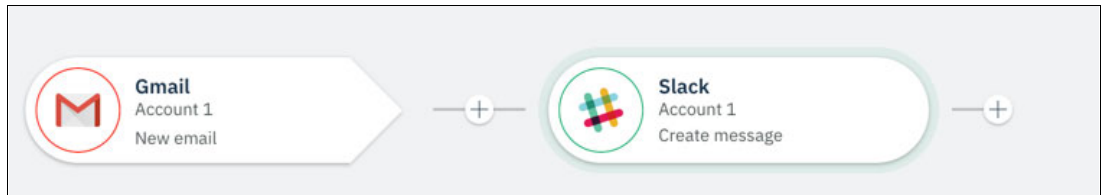


Figure 6-373 Applications

We could run this flow by providing only one additional configuration detail:
What is the message that we want to post to Slack?

15. Click the **Slack** application in the flow. In the lower half of the screen, the parameters for the application are shown in Figure 6-374 on page 427.

The one mandatory piece of configuration that is missing is the text (the message).

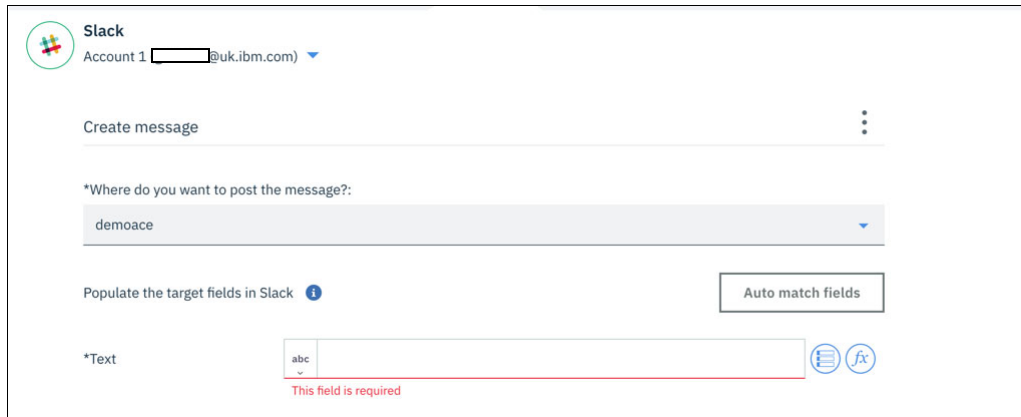


Figure 6-374 What's the message?

16. Click the **blue icon** next to the field to show all inputs that are available for the text. One of the nice design features about IBM App Connect is that at any point in a flow, all data of any operation or application that has come before is available for use. In this case, we have only the email itself, so we see a list of all of the parts of the email.

17. Select the **Body** from the list.

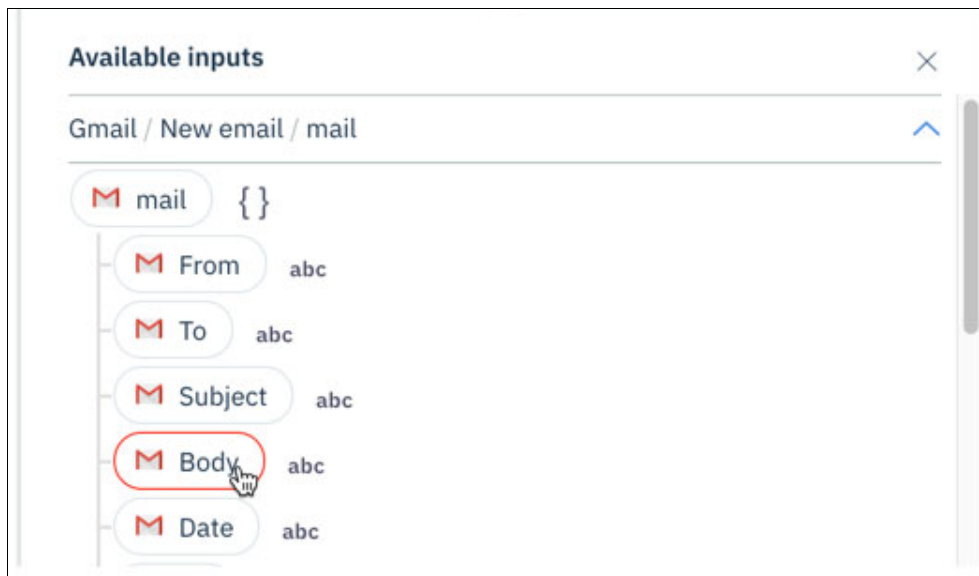


Figure 6-375 Insert Body

Now, the flow is correct and complete, and we could stop here and start the flow. However, not all emails to that account are necessarily file transfer notifications, so we need to filter out anything not relevant.

18. Click the **+** sign between the two applications. Select the **Toolbox** in the pop-up that appears. Here we see many different things that we might want to do with the data we are processing (conditional logic, array handling, parsing and so on).

19. In this scenario we can use the conditional If tool. Click this tool. See Figure 6-376 on page 428.

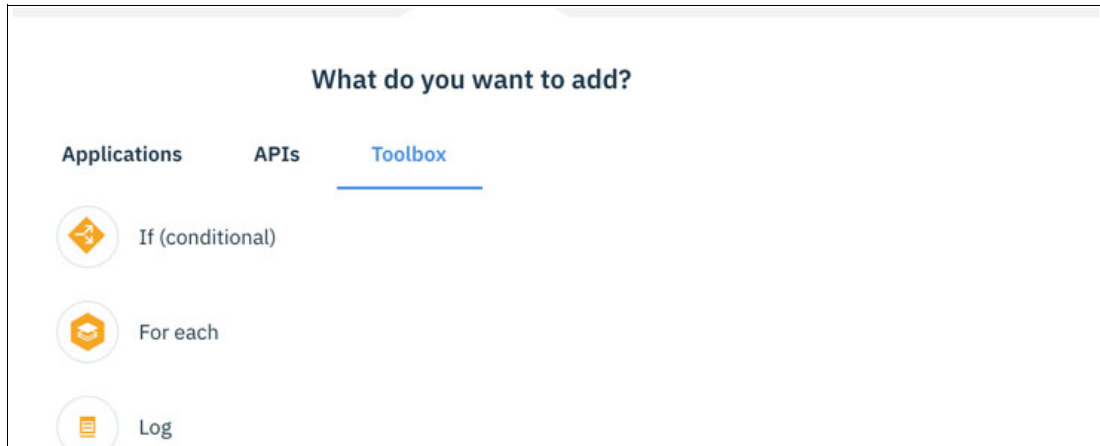


Figure 6-376 Add conditional logic

20. An If block appears in the flow. You can ignore the error flag as this simply indicates that there is more configuration required. See Figure 6-377.

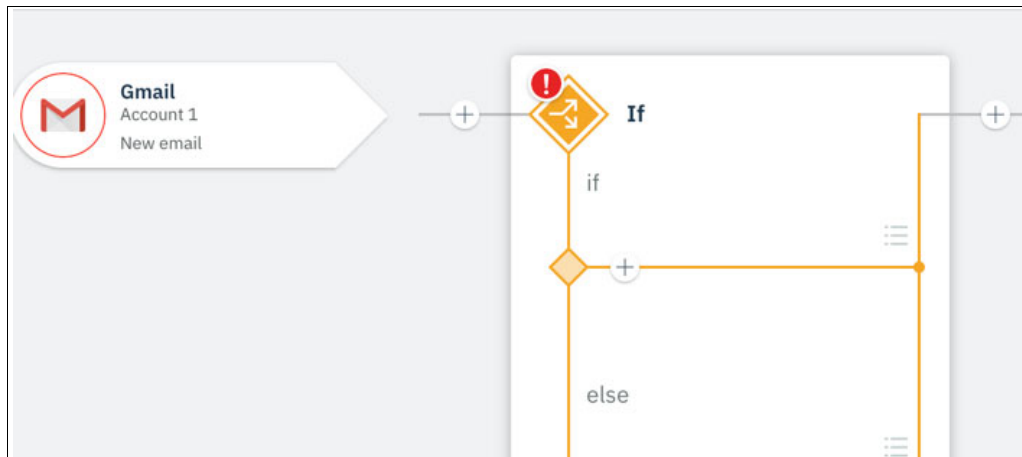


Figure 6-377 If block

The one mandatory piece of configuration we need to add is the if condition. This is shown in Figure 6-378 on page 428.

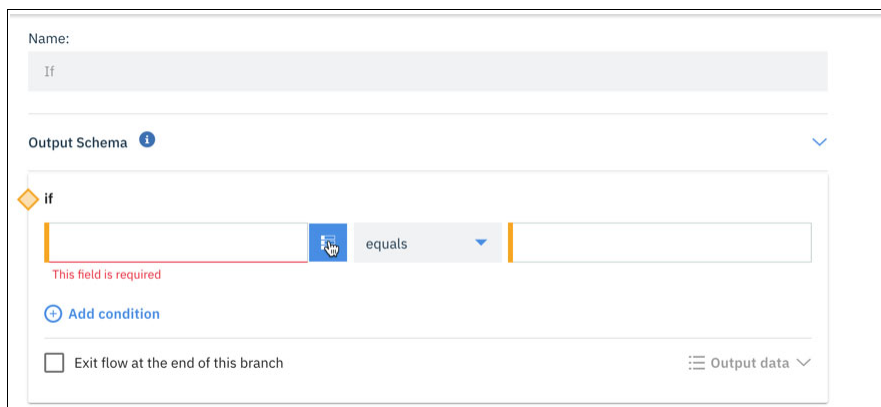


Figure 6-378 Needs an if condition

You might recall that earlier in this section (step 10), we took note of the default Subject prefix that would be put on the email to the recipient. We will now filter to only process emails that contain that prefix. For this we need to use a `$contains()` string function.

21. In the if condition box, type `co`.

22. This will pop-up an auto-complete helper, where we see String functions. Open the list and select the **\$contains()** function. See Figure 6-379.

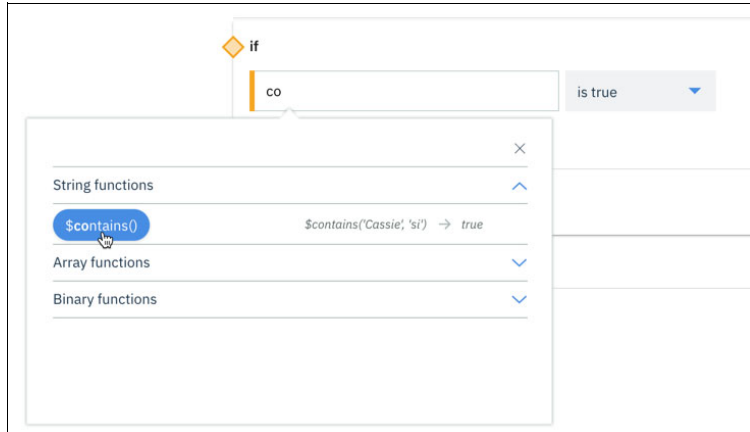


Figure 6-379 Select the function

23. We now see that the function has been inserted into the condition, we merely have to complete it. Before we do that, use the drop-down list to change the condition from is equal to is true.

24. Click the **string** and use the blue box at the side of the condition to pop up the available input. This time, select the **Subject** of the email. See Figure 6-380 on page 429.

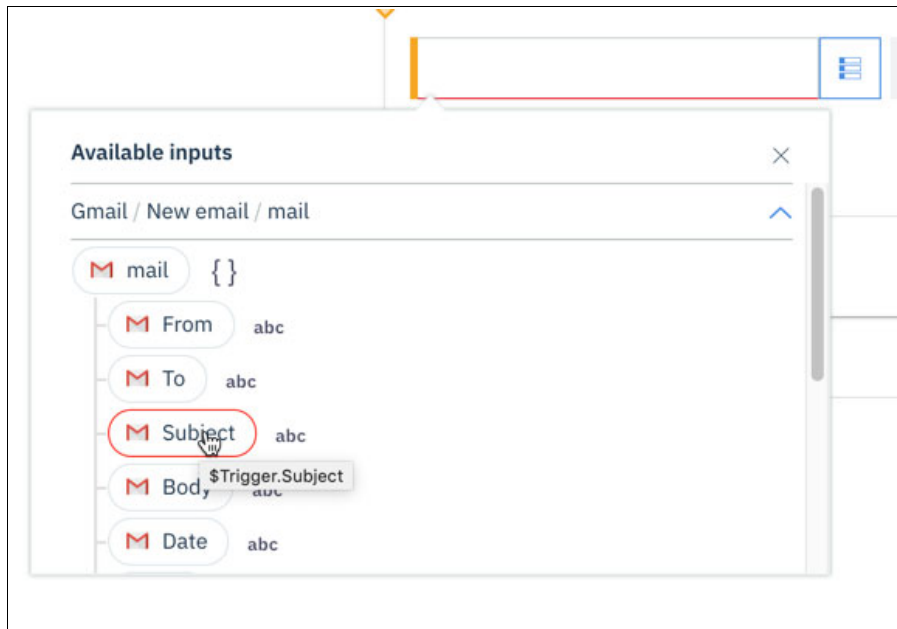


Figure 6-380 Fill condition

25. We now need to fill the character string to match, this is “[IBMASpera0nCloud]”. Simply position the cursor after the double quotation marks, and enter the text as shown in Figure 6-381.

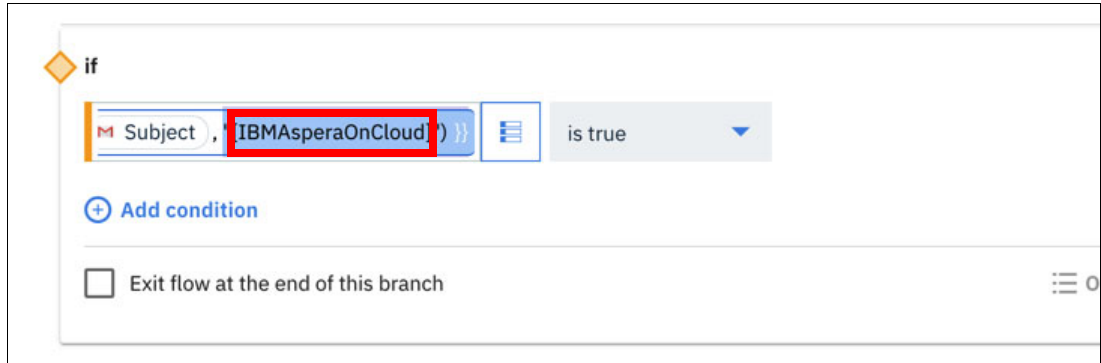


Figure 6-381 Completed condition

We need to move the call to Slack from outside the If block to inside so that we send only a message to Slack for the relevant emails.

26. Drag the **Slack** application from outside the IF block to inside as shown in Figure 6-382 on page 430.

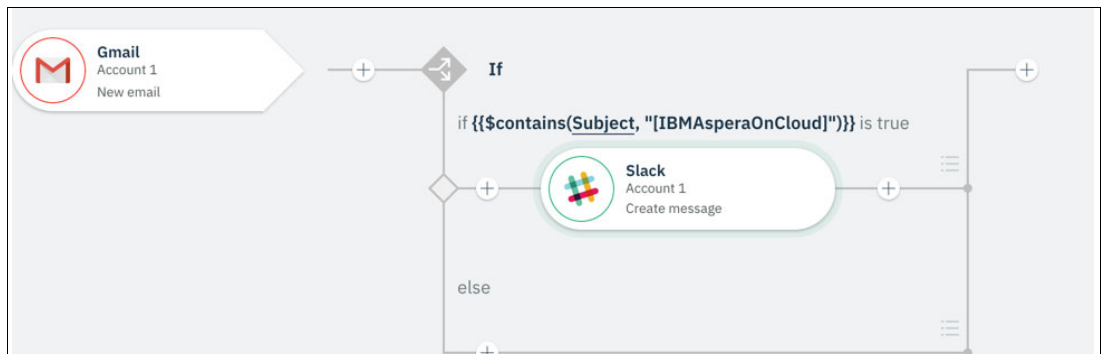


Figure 6-382 Completed flow

27. Last but not least, give the flow a meaningful name (we chose Notification from Aspera). See Figure 6-383.

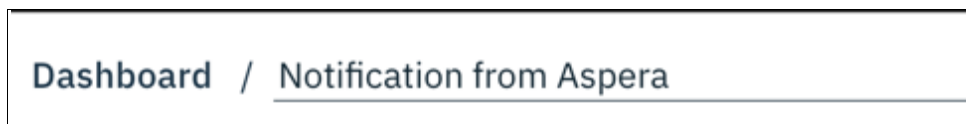


Figure 6-383 Flow name

28. If everything has been done correctly, we see a pop-up indicating that the flow is ready to be started (for example ready to start receiving and processing events). See Figure 6-384.

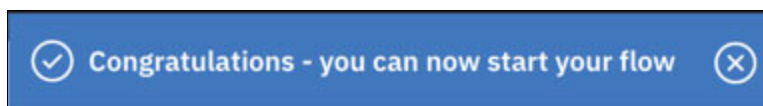


Figure 6-384 Congratulations, flow complete

The flow can be started either from here or from the main Dashboard.

29. Go back to the **Dashboard**. Right click the **hamburger menu** and select **Start**. See Figure 6-385 on page 431.

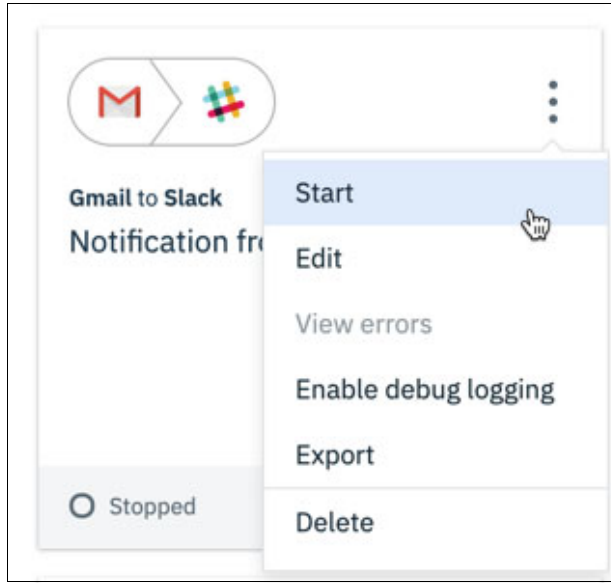


Figure 6-385 Start flow

We are now ready to test the flow.

Test the flow by sending another file

Perform the following steps for testing the flow:

1. Go back to IBM Aspera on Cloud and repeat the file transfer that you performed before.
2. After the transfer has completed, go back to the IBM App Connect dashboard.
3. If the flow has successfully received the event notification from Gmail and processed it, you will see a green tick in the lower right of the tile for the flow. See Figure 6-386.

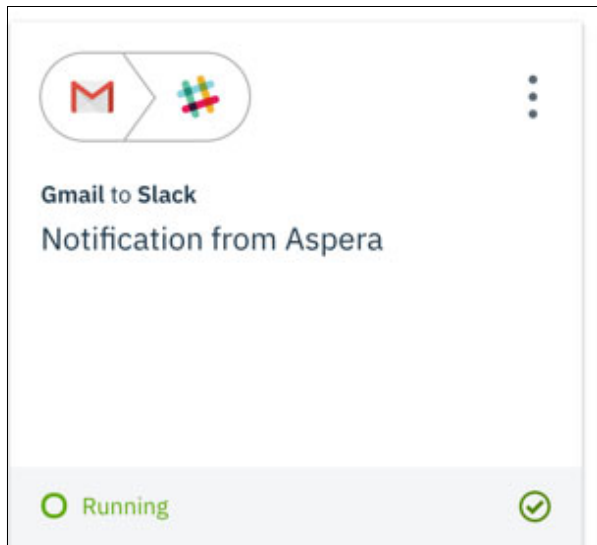


Figure 6-386 Successful completion

4. You should also see a notification from Slack that a message has been received. Open the message received. We see that IBM App Connect has given us the details of the email and also a link (containing a token) which is our claim check. Thus, allowing the recipient

to download the file when it makes sense to do so, and without having to have an IBM Aspera account or IBM Aspera transfer server. See Figure 6-387 on page 432.

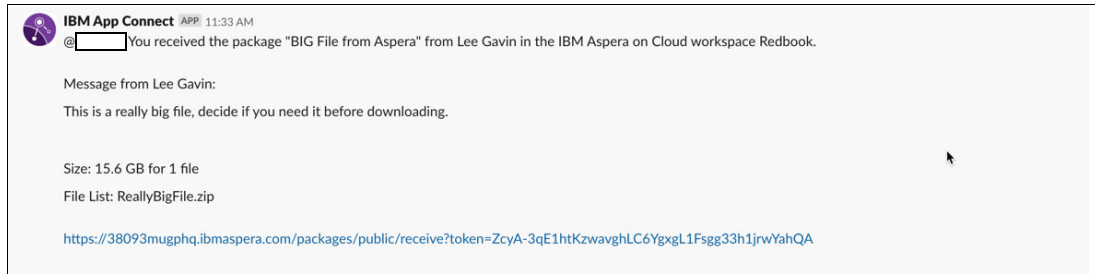


Figure 6-387 Go and get the file

We could further refine this scenario but adding or removing some of the information from the email body using the toolbox in IBM App Connect. We could also have processed the file, which would potentially make sense for a small file. However, the claim check pattern enables us to deal with much larger files in an “offline” fashion rather than blocking our fast-moving messaging platforms.

Obviously, this is a very simplistic scenario and only scratches the surface of what you can do with IBM Aspera on Cloud. For more information, go to this web site:

<https://ibm.ibmaspera.com/help/>



Field notes on modernization for application integration

In this chapter, we explore various guidance points for customers who plan to modernize their application integration landscape. The target audience for this chapter is individuals who are already skilled in IBM App Connect (or its predecessors such as IBM Integration Bus, WebSphere Message Broker). We explore at a product level what it means to move to a more agile integration style. For example, such a move includes a move to container technology, refactoring to more fine-grained deployment, and automation of build pipelines.

This chapter has the following sections:

- ▶ IBM App Connect adoption paths
- ▶ Splitting up the ESB: Grouping integrations in a containerized environment
- ▶ When does IBM App Connect need a local MQ server?
- ▶ Mapping images to helm charts
- ▶ Continuous Integration and Continuous Delivery Pipeline using IBM App Connect V11 architecture
- ▶ Continuous Adoption for IBM App Connect
- ▶ High Availability and Scaling considerations for IBM App Connect in containers
- ▶ Migrating centralized ESB to IBM App Connect on containers
- ▶ Splitting an integration across on-premises and cloud

7.1 IBM App Connect adoption paths

IBM App Connect V11 combines the existing, industry-trusted IBM Integration Bus (IIB) software with new cloud based composition capabilities, including connectors to a host of well-known SaaS applications. However, a more fundamental change is the continued focus on enabling container-based deployment of the on-premises software runtime. But V11 does not mandate a move to containers. Customers can continue deploying workloads in the more centralized ESB pattern if that is their preference. In this section, we focus on deployment options for the on-premises software, comparing the traditional centralized ESB topology with the alternative containerized deployment in order to understand the pros and cons of each path.

7.1.1 Agile integration

Figure 7-1 summarizes the progressive phases of integration modernization defined by agile integration.

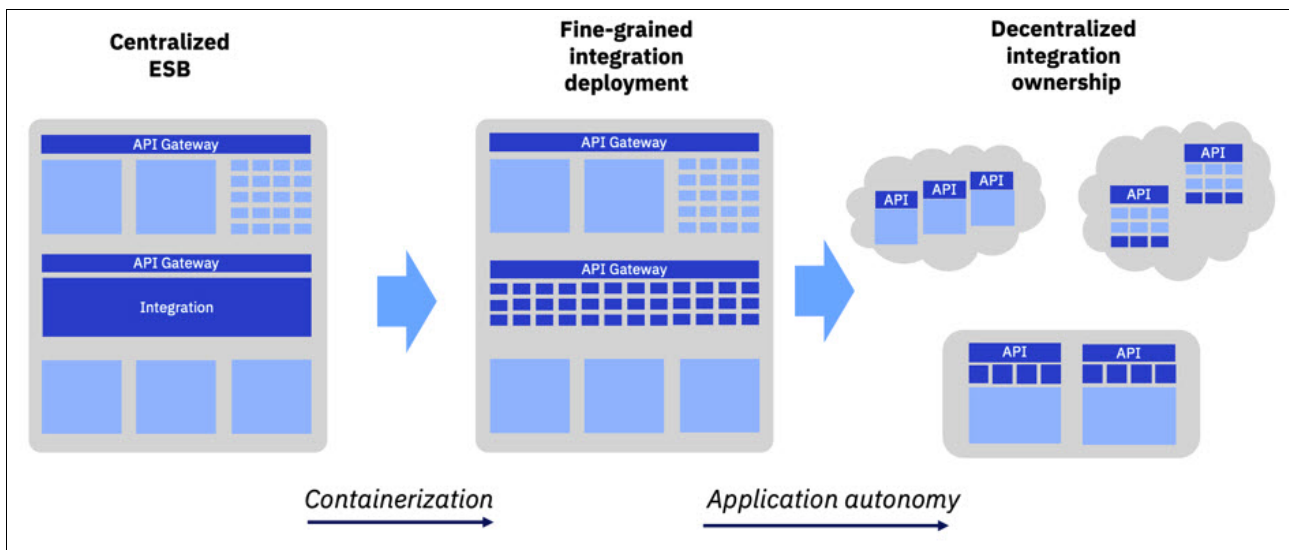


Figure 7-1 Moving progressively to agile integration

Agile integration has the potential to dramatically improve the velocity at which connectivity is delivered, and the discrete resilience and elastic scalability of isolated integrations. Much of this is achieved initially by moving to a more fine-grained deployment of the integrations themselves making it possible to change integrations more independently. After integrations can be deployed independently, this lays the foundations for decentralized ownership whereby the ownership of the integrations moves from a central integration specialist team out to the application teams. Enterprises vary in how far down this road they need to travel and at what pace.

7.1.2 Adoption path options

So, what does this mean for users who have been running workloads on previous versions of IBM Integration Bus? When we upgrade to IBM App Connect, how can we best prepare ourselves to leverage agile integration benefits?

Some amount of fine-grained deployment can be achieved by using existing capabilities. But to gain the most effective isolation between integrations implies the need to move to container-based technology. However, as we see, this is more than just a replatforming exercise. To gain the most significant benefits we need to move to a truly cloud-native style of deployment with impacts how teams build, deploy, administration, and monitor their integrations. Some enterprises want to take more gradual steps, staging their way to cloud-native, rather than jumping in with both feet.

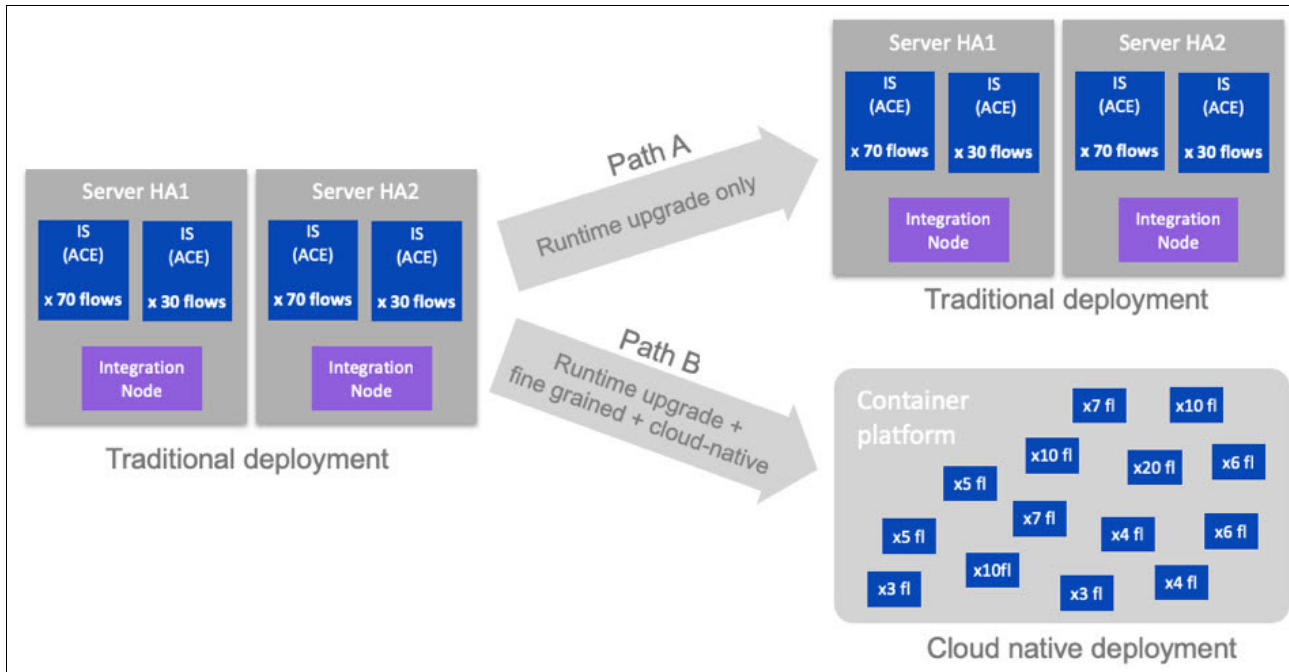


Figure 7-2 IBM App Connect adoption paths

We start with the conceptually simplest upgrade we could do, raising the level of the runtime to IBM App Connect v11, but still on our existing traditional topology (Path A). We then see how much we can push that toward the benefits of agile integration. And we can also see at what point we need to make the more significant changes toward a true cloud-native deployment on containers (Path B).

Path A: Runtime upgrade only (preserve existing topology)

On this path we simply upgrade the runtimes of the Integration Server and Integration Node, and the developer Toolkit. The core topology remains the same.

IBM App Connect v11 does not mandate a move to container infrastructure. It can still be deployed in this traditional topology by using Integration Nodes to administer the Integration Servers just as we did in prior versions of IBM Integration Bus. Let's also assume that at least some of the workloads require a local MQ server, so that also needs to be present in the topology.

Even this simple upgrade path brings core runtime and tooling enhancements. Examples depend on what version you are moving from, but might include:

- ▶ Simpler file system-based installation
- ▶ Removed hard dependency on local MQ server
- ▶ New capabilities such as the Group node
- ▶ Toolkit now supported on MacOS

- ▶ New admin console
- ▶ New web UI for Record and Replay functionality
- ▶ Consolidated configuration through properties files: `server.conf.yaml`, policies
- ▶ Access to a wide range of cloud connectors to access well known software as a service applications and many more.

Therefore, Path A has the following benefits in comparison to Path B:

- ▶ runtime-only upgrade
- ▶ minimal learning curve on new technology
- ▶ minimal mandatory changes to the build and deploy pipeline

A traditional integration topology looks something like the following (Figure 7-3 on page 437). Interestingly, a similar diagram could be drawn for just about any product deployed in a non-cloud native way such as an application server for instance.

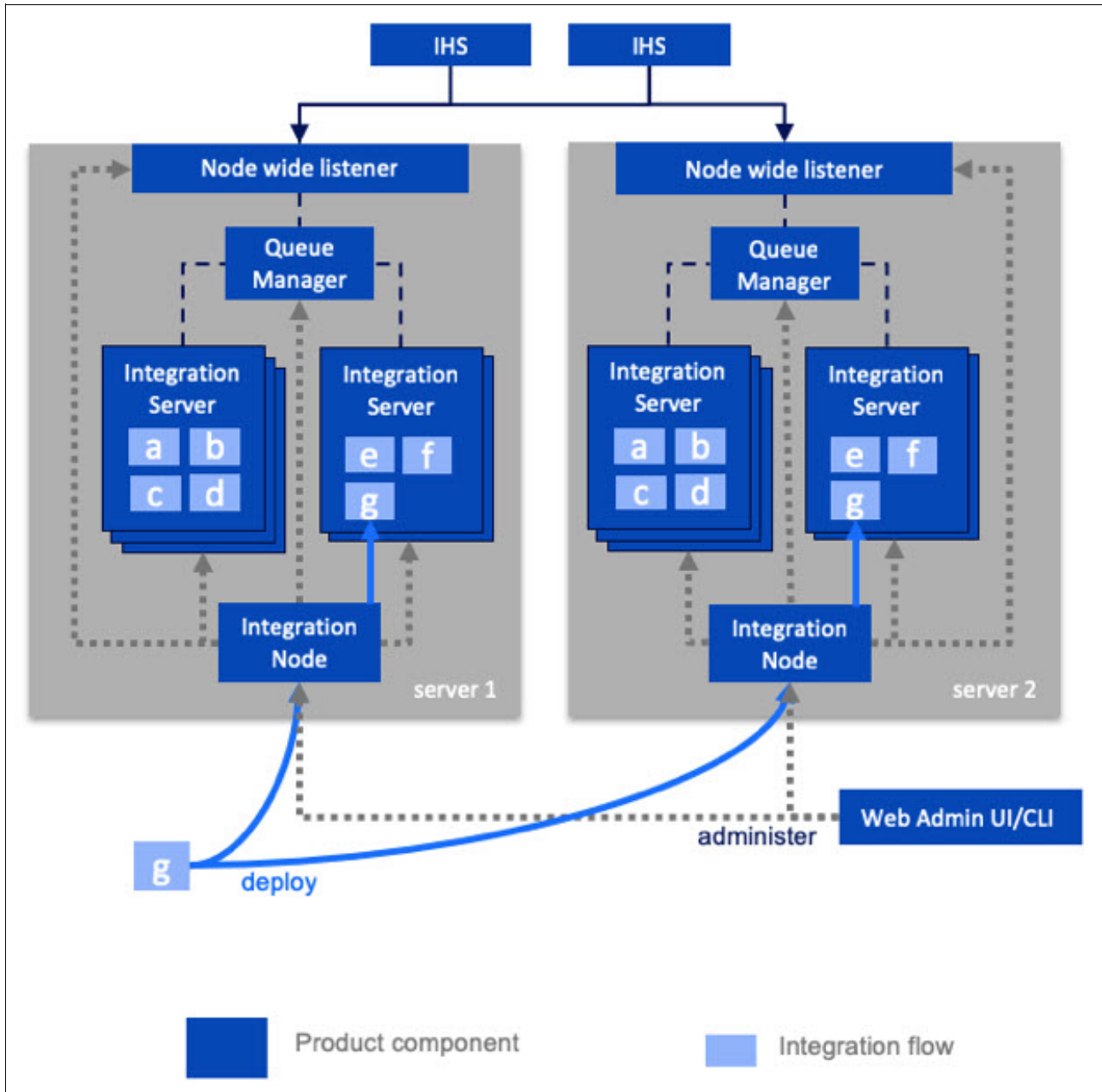


Figure 7-3 Example of a traditional deployment (IHS stands for IBM HTTP Server)

Perhaps the first thing to notice is just how much of the diagram is dark blue (labeled “product components” in the key). These are the elements that require product-specific skill to install and administer.

Note: Customers who have upgraded from v10 already have these skills.

It is also worth noting that it is a fixed topology – it defines a specific high availability pair.

These traditional topologies have some limitations in comparison to their cloud-native equivalents. Let’s have a look at some key aspects:

- ▶ **Isolation:** Any changes that affect the running server, whether fix pack upgrades, introduction of new integrations, changes to existing integrations, or configuration changes to the server, carry risk as they affect all integrations that run on the shared servers. We must either carry that risk or mitigate it with potentially significant amounts of regression testing across our integrations.
- ▶ **Scalability:** The topology can be extended only through manual configuration. More CPU could be added to server HA1 and HA2. But it would likely still require careful scheduling, and there is clearly a physical limit to the amount of CPU that can be added. Adding a “server HA3” would typically be a manual exercise, as would removing it when the extra load is no longer present.
- ▶ **High availability:** There is a significant amount of the topology to be built beyond that of the installation of runtimes and much of this relates to high availability. Consider how much of the preceding diagram is dark blue (labeled “Product component” in the key), which means it needs to be explicitly installed and configured. This includes setting up of load-balancing within and across nodes, enabling high availability and disaster recovery. This must be repeated for each environment such as development, test, production. Significant custom work to create each environment. But also, there is also a genuine risk that configurations in each environment become out of synch. Certainly, there are ways to automate and patternize installations, but this in itself is additional work.

It is worth recognizing that even with this traditional topology it is possible to make some moves towards the benefits of agile integration. For example, it is already possible to split a large installation into a number of separate Integration Servers, each containing a subset of the integrations and administered via a single Integration Node. Many large installations are likely to be already using this method of grouping. This provides some level of isolation between sets of integrations, but not the deep decoupling that would be offered by containers described later in Path B. They also enable some degree of independent scaling of integrations, but not to the extent of automatically and elastically provisioning new hardware resources that you would get in a container orchestration framework.

Perhaps the most striking point is that in this traditional installation, administration and deployment all require specialist skills. Container-based environments aim to standardize those skills and make those skills transferable across technologies such that the only specialist skill required, is the one that matters: *how to build artifacts*. In our case that would mean the building of integration flows. Everything else should be done by using common tools and capabilities that are common across all the technologies.

We now look at the cloud-native path. And we take a deeper look at what benefits a more fine-grained deployment on containers might bring if we adhere to a true cloud-native deployment style.

Path B: Cloud-native deployment (including runtime upgrade)

On this path, we switch completely to a container platform, but more than that, we embrace true cloud-native style with associated benefits. Figure 7-4 on page 439 is a simplistic example of the key elements of a cloud-native topology:

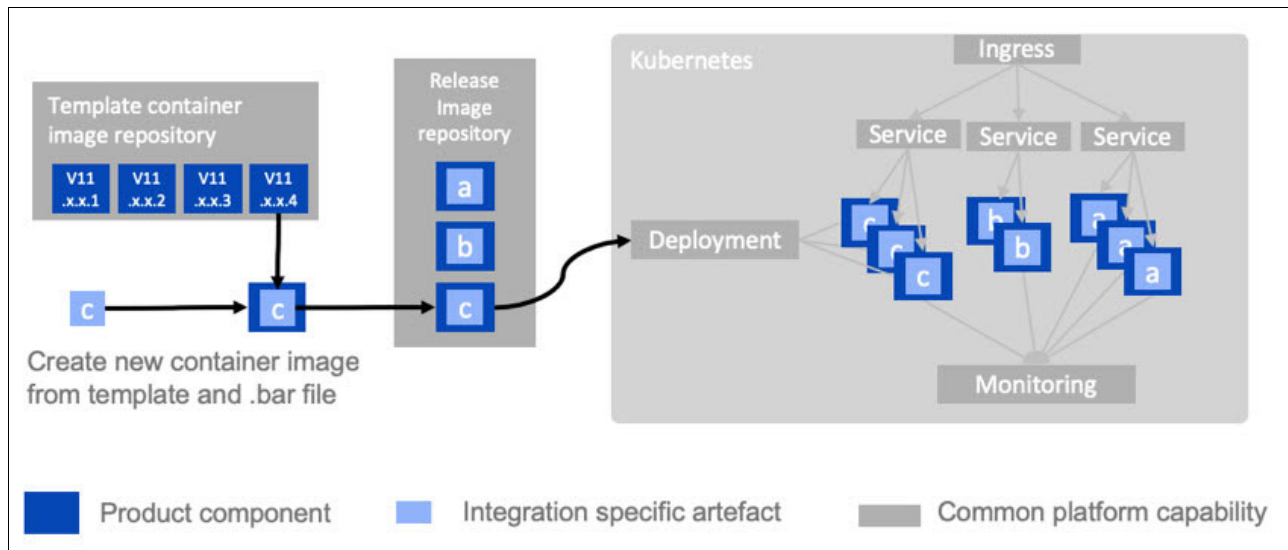


Figure 7-4 Example of a cloud native deployment

Compare this diagram with the diagram for the traditional topology from Path A. The first thing to notice is the smaller amount of dark-blue product-specific components. This is possible because much of their role is now performed in a standardized way by the container orchestration platform. Indeed the remaining dark-blue boxes are simply references to standardized container image templates (and Helm Chart templates) from which they were built. This demonstrates the level of consistency that a containerized approach provides.

Let's begin by comparing those same three aspects we discussed at the end of the previous section, then we look more broadly at other benefits of a cloud native approach.

- ▶ **Isolation (fine grained deployment):** Containers are truly isolated from one another, almost as if they were separate operating system instances. The integrations can be split across multiple containers so that changes to the integration flows — and also changes to fix pack versions of the IBM App Connect runtime — affect only a very small number of integrations within a given container.
- ▶ **Scalability (policy-based elastic auto-scaling):** Container orchestration frameworks provide elastic scaling capabilities out-of-the-box in a standardized way. They enable automated dynamic changes to the number of replicas of containers based on defined workload policies.
- ▶ **High availability (auto reinstatement):** In a containerized world there are standardized ways to declaratively define an HA topology (Helm Charts). Furthermore, the components that enable the high availability such as load balancers and service registries do not need to be installed or configured because they are a fundamental part of the platform. High-availability policies are built into Kubernetes, and these can be customized through standard configuration.
- ▶ **Visibility (platform-based monitoring):** With fine-grained deployment comes a larger number of components deployed. Container orchestration systems offer a single way to view the health of components across all types of runtime deployed (in other words, not just integration). Furthermore, common standards such as ELK stacks and Prometheus are coming to the surface as effective ways to add deeper monitoring capabilities. These are often built into commercial container orchestration offerings such as OpenShift.
- ▶ **Repeatable, rapid topology creation (infrastructure as code):** In a container orchestration environment, you don't build topologies yourself. The topology requirements are defined declaratively in files that can be stored alongside the code. This ensures that

the integrations are always deployed onto a topology that suits their needs. Helm Charts are currently the most common mechanism for providing the logical definition of the requirements from the deployment topology. For example, they define how it should respond to changes in workload, then leave the orchestration framework to work out how to build that out and keep it running.

- ▶ **Cross-environment consistency (image-based deployment, declarative configuration):** Containers enable us to draw together the operating system, product binaries, configuration and the (integration) code into a single immutable image. Furthermore, we can combine that with the infrastructure-as-code to define the topology. We are then assured that we are deploying exactly the same thing to every environment from development right through to production.
- ▶ **Pipeline automation (file system-based runtime installation and artifact deployment):** The IBM App Connect integration runtime can be installed, and integration flows deployed simply by laying their files on the file system. This significantly reduces the specialism required in creation and maintenance of an automated build pipeline to create container images, and significantly improves the image preparation time.
- ▶ **Operational consistency (common platform administration):** A broader benefit of container platforms is that the skills required for operation of the environment are the same across all product and language runtimes – not just integration runtimes such as IBM App Connect. One of the aims of containers is encapsulation: that they should all look essentially the same from the outside. This allows their deployment, scaling, monitoring, and administration in general to be done by using standard container platform capabilities with little need for knowledge of what's inside the container. This encourages consistent practices on operations across the landscape and reduces the number of skill sets that need to be maintained.
- ▶ **Portability (standards-based container technology):** It could be argued that the level of standardization around container technology makes it simpler to move components between platforms, and indeed between your own infrastructure and other cloud infrastructure. However, this is a rapidly changing technology space, and careful choices are required so that components remain portable. Based on those careful choices, OpenShift allows you clear portability across various cloud and on-premises environment choices.
- ▶ **Decentralization (technical autonomy to business teams):** A truly cloud-native approach allows developers to focus on the creation of artifacts by simplifying build and deployment, and standardizing the surrounding administration and operational needs. This in turn makes it possible for teams to get up to speed on new languages and technology more quickly. In the past, we tended to have centralized teams, specialized on specific technologies. Relying on more standardization of the surrounding platform, it is now reasonable to allow teams to become multi-skilled across a range of build technologies rather than relying solely on centralized teams.
- ▶ **Shift-left (automated, production aligned testing):** Thanks to immutable containers and the consistency of topology build, developers can build tests and environments that more closely resemble the non-functional aspects of the production environment. As a result, these qualities of service are built in and tested right from the start.

7.1.3 Conclusion

Containerization is not mandatory for moving to v11 of IBM App Connect. You can upgrade the runtime only. In the new version, you retain the existing Integration Node/Integration Server topology, and benefit from many fundamental enhancements.

However, containerization and the associated move to a more cloud-native approach has many advantages. Among these are simpler deployment build pipeline, isolation/decoupling between integrations, consistency across environments, portability, standardized administration and monitoring, and common capabilities that enable non-functional characteristics such as scaling and high availability.

7.2 Splitting up the ESB: Grouping integrations in a containerized environment

The agile integration approach, as shown in Figure 7-1 on page 434, ideally involves moving to a container-based infrastructure, allowing integrations to be deployed more independently. At the extreme end of this approach we could imagine a separate container for each and every integration. It is technically possible to do this, but not advisable. Doing so would likely result in a more numerous and complex collection of components to administer. It is more advisable to go for a more measured approach, with groups of integrations in each container. The container would have only one integration runtime process, in keeping with good practice, but that integration runtime would have a group of integrations that are loaded into it. So, we clearly have some decisions to make around how we want to group our integrations together, and we discuss this topic in this section.

To put this in practical terms, it is not uncommon to hear of centralized ESBs that run 100s of discrete integrations across a highly available pair of servers. It is easy to see the issues and risks that could arise when you deploy integrations and update the currently running integrations. Furthermore to take advantage of new features, any upgrades to the underlying integration runtime in require at least some regression testing across all the integrations present. Also, and potentially some downtime might be needed to do the upgrade. See Figure 7-5.

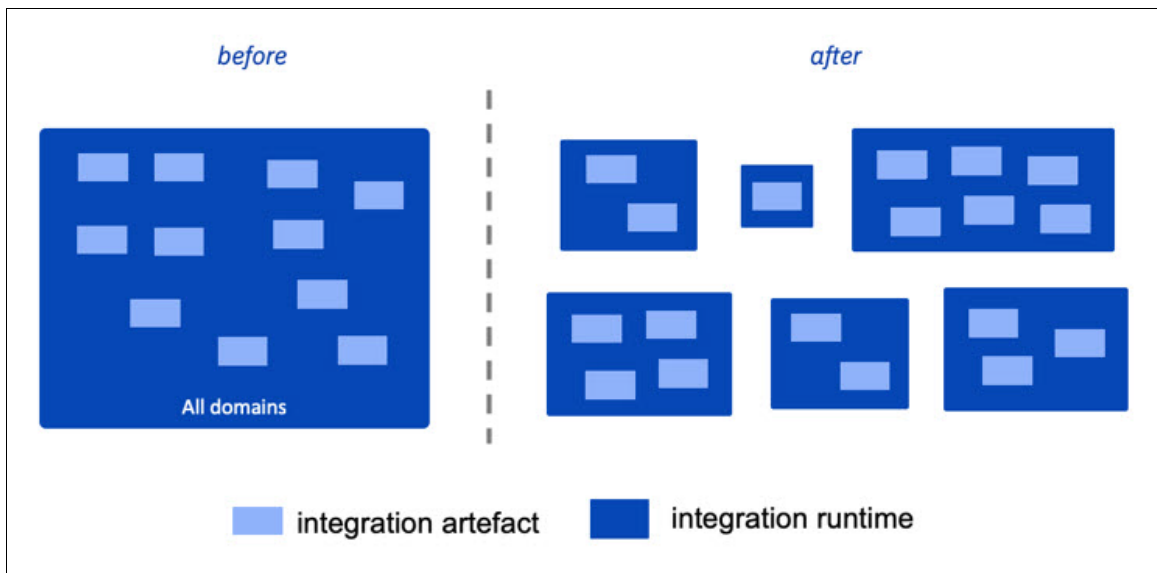


Figure 7-5 Basic grouping

In a more decentralized and containerized architecture, we might move a group of integrations in each container. These integrations could then be maintained independently, and the underlying integration runtimes patched independently, with no risk to integrations in other containers. This then offers benefits of agility, productivity, elastic dynamically optimized scalability, and more fine-grained resilience models associated with agile integration.

Ultimately, this brings faster time to market, reduced system downtime, and more cost-effective use of resources.

So, what criteria should we use to decide which ones could/should stay together, and which ones must be separated from one another?

7.2.1 What grouping do you have today?

Many businesses have been grouping their integrations into separate Integration Servers (*execution groups* as they used to be called prior to v9 of the product). You might have already evolved a good strategy for grouping of integrations into suitably independent groups. It is certainly wise to take that as your starting point. If it has been providing sufficient decoupling of deployments to date, this might be all that you need. As they say, *“If it ain’t broke, don’t fix it!”*

However, for many, this type of separation has not yet occurred. Or perhaps it occurred as a result of more hasty tactical decision, and revisiting the grouping design would be wise.

7.2.2 Splitting by business domain

The simplest place to start is business domains. Integrations owned (created and maintained) by radically different parts of the business would be better separated from one another. That way, they have less chance of affecting one another whether at deployment or runtime. See Figure 7-6.

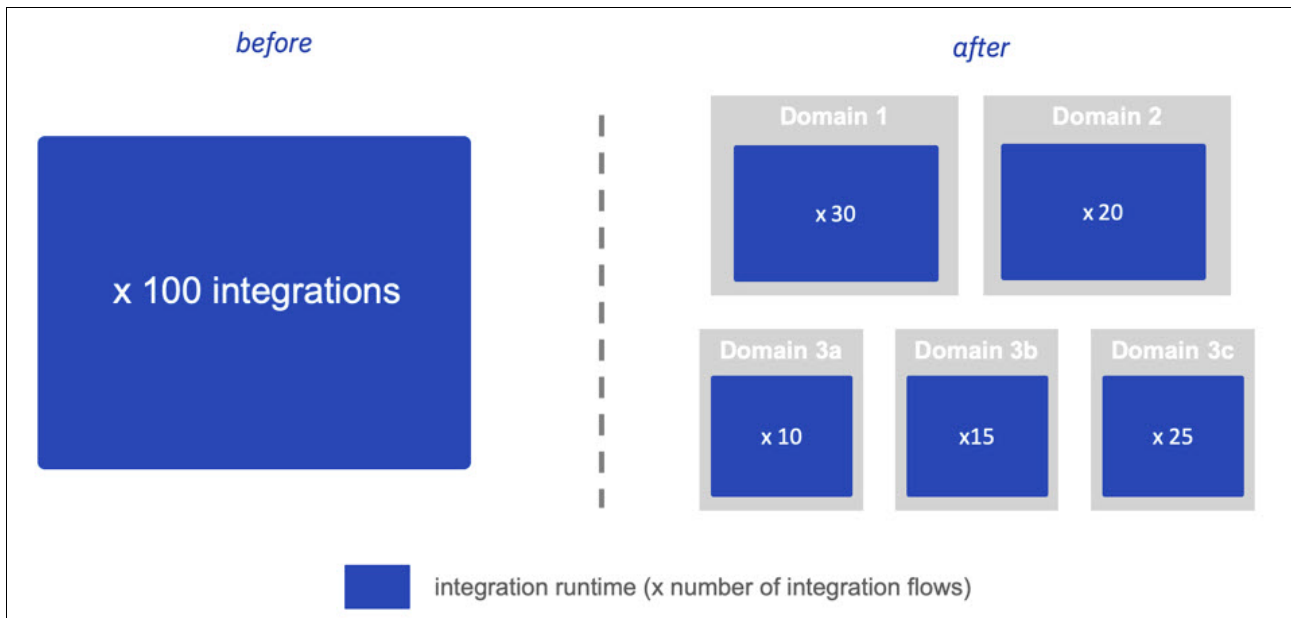


Figure 7-6 x 100 integrations - Domains

An obvious example of a very coarse-grained split of this type would be in an insurance company that has completely different business domains that handle general insurance, life insurance/pensions, and health insurance. There would be little advantage in sharing infrastructure among integrations that are built for these very separate domains. Indeed, it is likely that such a coarse-grained split will already be evident, with each business domain a having separate enterprise service bus infrastructure.

If we plan to move to containers, we can retain this strong separation between business domains by using, for example, Kubernetes namespaces and network policies to separate them at a network level. This might be supplemented by container-based software defined networking such as Calico. At the extreme end we, could of course have different Kubernetes clusters for complete separation even at the infrastructural level.

The same concept could then be followed within a domain to group integrations that into subdomains (3a, 3b, 3c). So continuing with the insurance analogy, within the general insurance domain, we might see a natural split across motor insurance, house insurance, travel insurance and so on. Splitting by subdomain is useful if the integrations are genuinely owned (created and administered) by very different business groups, and as such would benefit from being handled independently. However, there might be little benefit in splitting buildings insurance from contents insurance if they are looked after by the same team of people. We would need to look for another reason to subdivide them as we discuss next.

7.2.3 What about integrations that span business domains?

While business domains provide a convenient way to subdivide our integrations, we should recognize that integration by its very nature often crosses business domains.

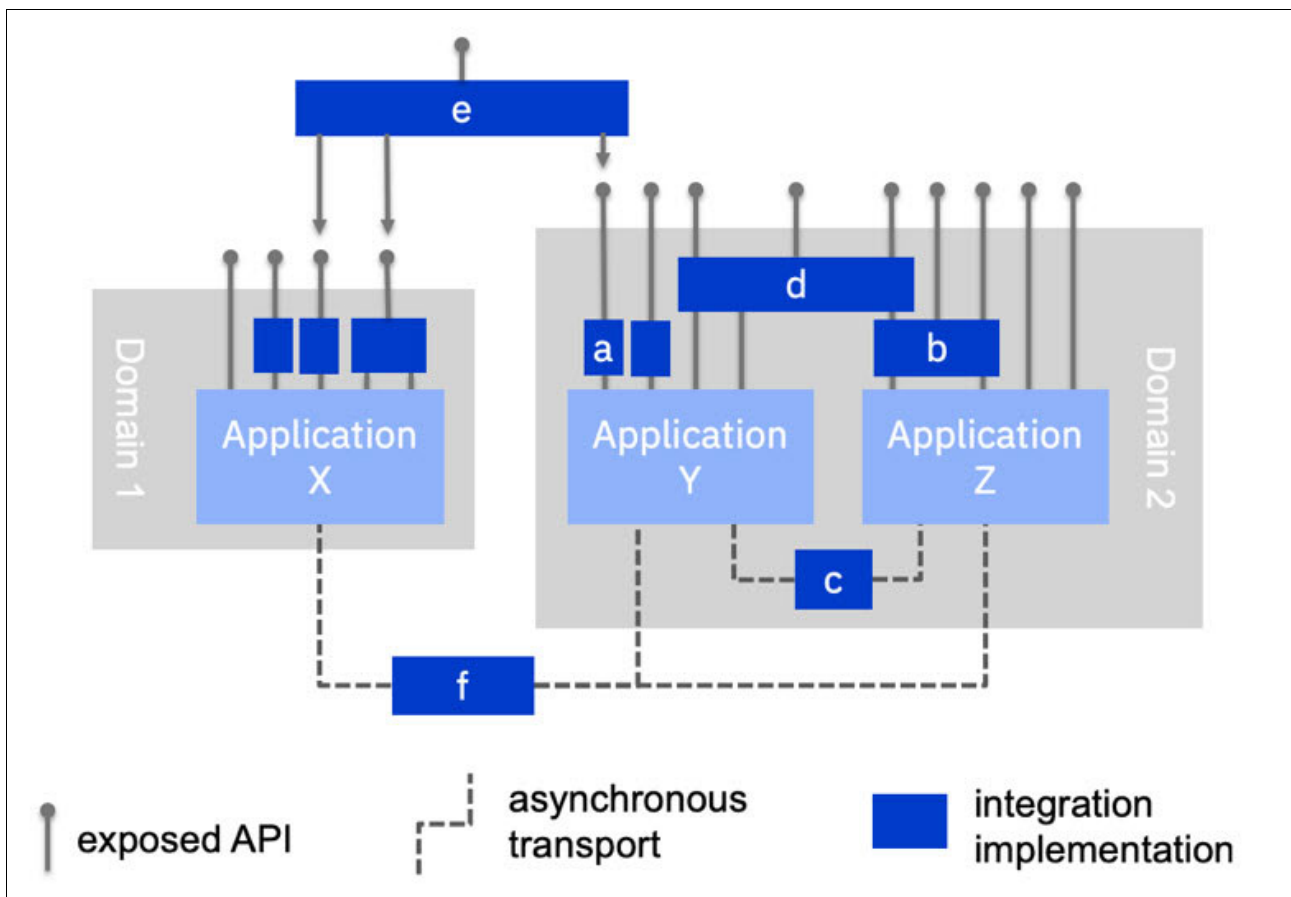


Figure 7-7 Grouping across domains

In Figure 7-7, integrations **a**, **b**, **c** and **d** are contained within a domain. However, an integration such as **e** is an API implementation that aggregates calls across multiple domains. For example, an API that aggregates all the insurance contracts owned by a single individual,

across all business domains. So, it might collate details about their travel, car, building, life, and health insurance to provide a single view of our relationship with the customer.

Another common example is **f**, an integration that propagates events from one domain into another. A common scenario is synchronizing business data such as a customer's address that might be duplicated in applications in different domains.

Ideally, we would want to allocate cross-domain integrations like **e** and **f** in their entirety to a specific domain even though they have contact with other domains.

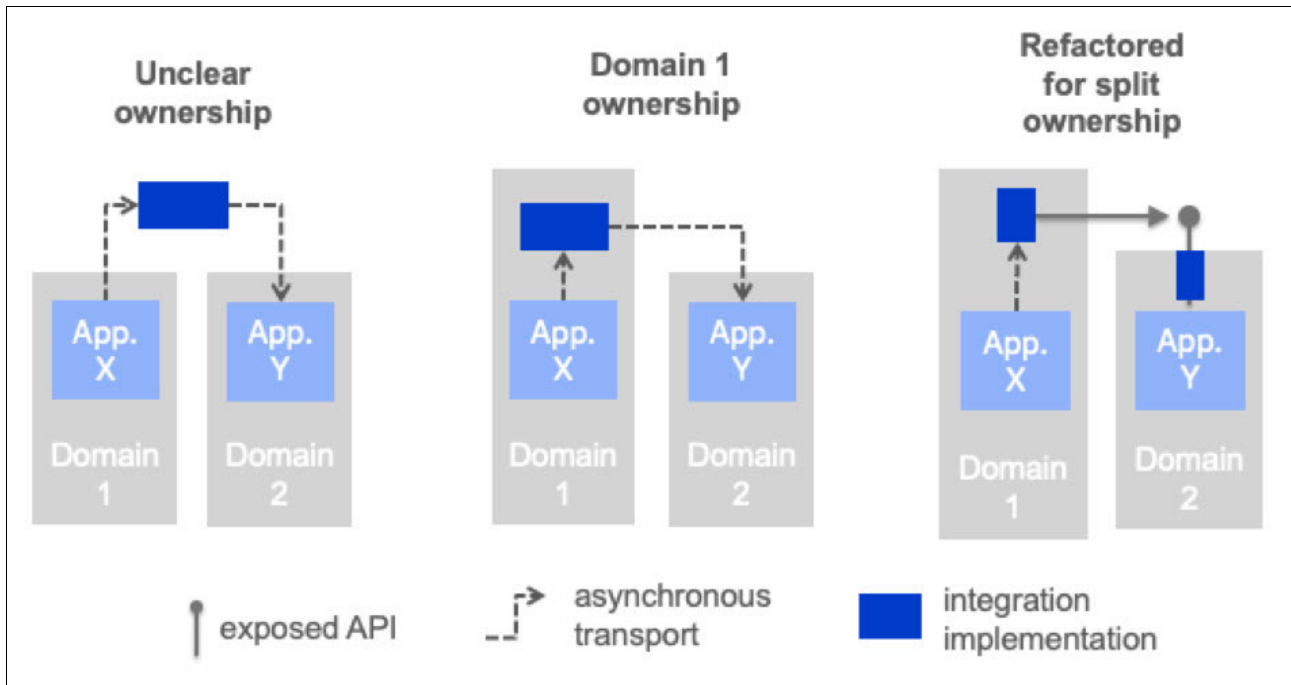


Figure 7-8 Splitting integrations

In some cases, the right approach might even be to split the integration into several parts each of which have a more clear ownership within a domain. In Figure 7-8, Application Y uses an integration to formally expose an API for general reuse. Domain 1 then has a listener integration that receives events from Application X, and it then propagates the event data to Application Y via its new API.

Tip: Grouping the applications with the component that has the highest likelihood of change that would impact this component would help align the lifecycle of each.

Forcing ownership decisions for these crosscutting integrations is not such a bad thing. All components in an architecture need an owner if they are to be maintained effectively over the long term.

7.2.4 Grouping within a domain

Having performed an initial split of the integrations based on business domains, we now need to look for reasons why integrations might need to be grouped within those domains. In the following diagram there are some possible reasons that we might group integrations together. In the remainder of this section, we explore the pros and cons of each.

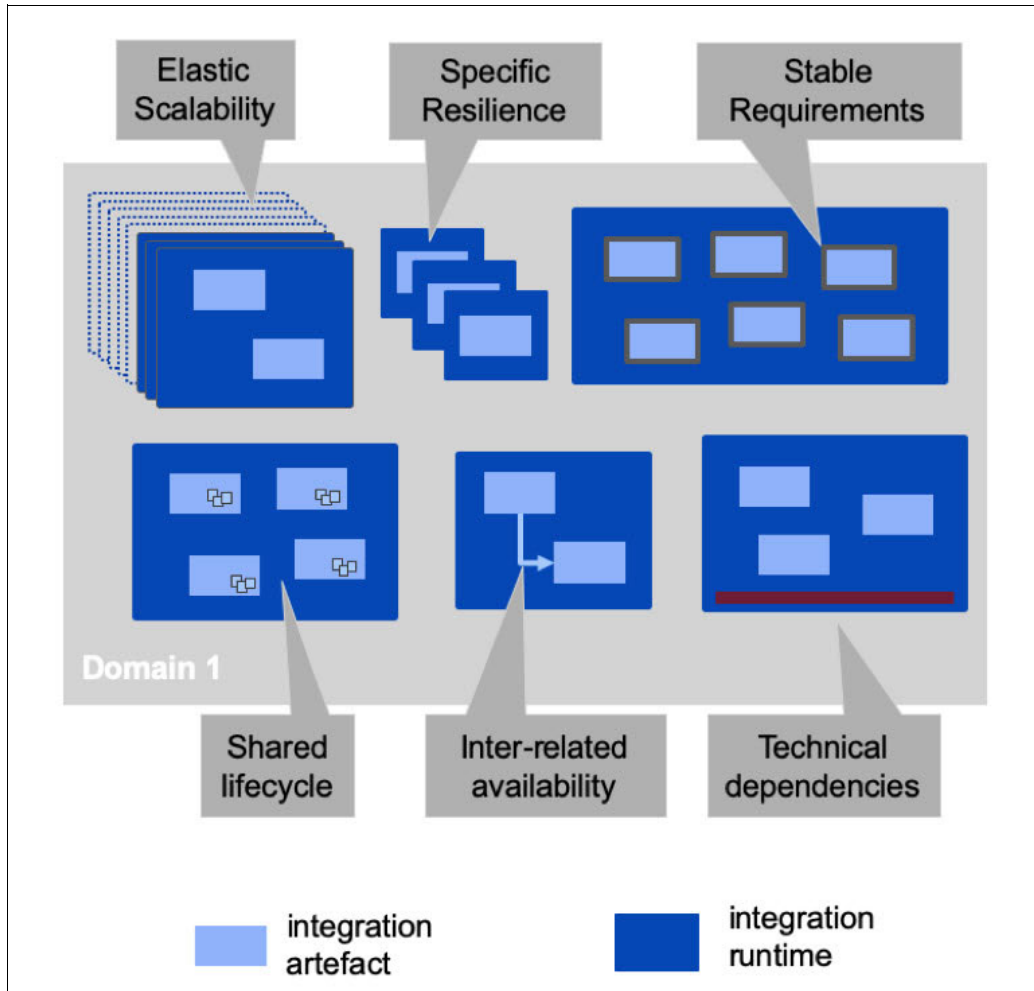


Figure 7-9 Grouping summary

7.2.5 Stable requirements and performance

Maybe the most obvious reason to group integrations would be if they are stable from a requirements and performance point of view. Perhaps few if any changes have been required on these integrations for several years, and the workload they serve is well known and predictable. There would seem little point in separating these integrations into individual containers.

The only thing against keeping them together might be robustness. If one of the integrations were to suffer a failure that had an effect on the overall runtime, it might bring the rest of the integrations down with it. However, given that integrations have matured over several years, most of their usage permutations have already been explored. In other words, if they were going to have a catastrophic failure, it would probably already have happened. Furthermore, if they've been living together on the same server until now, we can assume we are comfortable the availability they provide. As such we can be reasonably confident that they can live alongside each other and retain the same level of service as we currently have.

Technical dependencies

There might be sets of integrations that all rely on a key runtime dependency. The most obvious example is those that need a local MQ Queue Manager to be present within the container, but we will look at some other examples, too.

Local MQ server dependency

The hard dependency on a local MQ server was removed in v10. However, many interfaces were written by using MQ server bindings, because it could be assumed that a default local queue manager was present. Many of these could be refactored to use client bindings and thereby not have a dependency on a local queue manager. But some interfaces will continue to require a server binding due to the nodes that are used in the flow, or their transactional requirements.

See Figure 7-10.

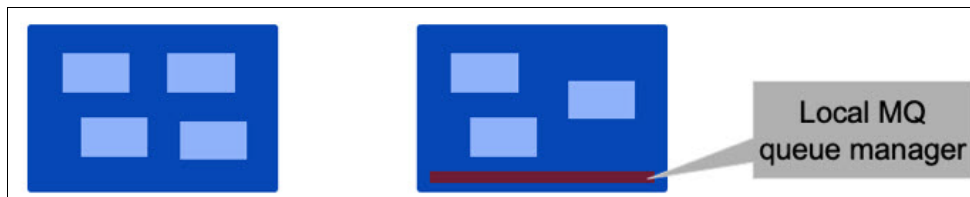


Figure 7-10 Local MQ queue manager

Those requiring a local MQ server typically also require a persistent volume. As a result they will be more restricted in terms of how dynamically they can scale up and down. We should at the very least separate out the integrations that do not need a local MQ server such that they can enjoy elastic scalability and faster startup times.

Refer to section 7.3, “When does IBM App Connect need a local MQ server?” on page 452 for more details on this topic.

7.2.6 Synchronous versus asynchronous patterns

There are a huge number of different integration patterns, but at a very high level they can almost always split up into two core types. See Figure 7-11.

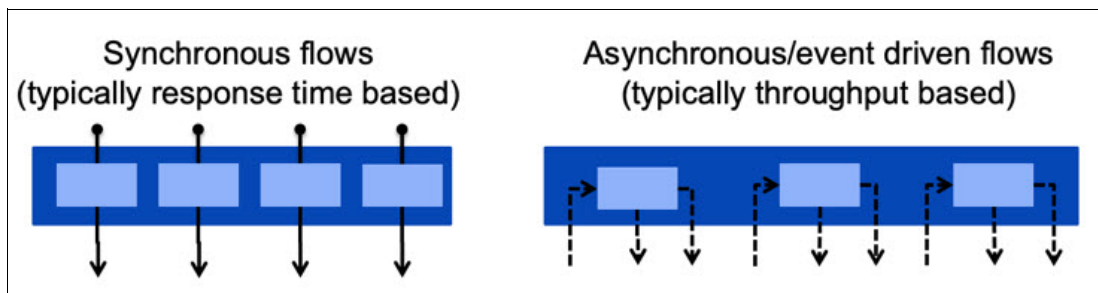


Figure 7-11 Synchronous versus asynchronous patterns

- **Request-response (blocking):** The caller waits for the integration to occur because they need the final result to continue. This applies to most web services or REST APIs. Minimizing “response time” is critical for these interactions as the caller is blocked until the integration completes.

- ▶ **Fire-forget (non-blocking):** Those that react to asynchronous fire-and-forget events. The initiator of the event does not block waiting for a response. The focus for these interactions is maximizing “throughput” – the rate at which events are processed.

Notice that the preceding terms refers to the overall interaction pattern, not to the transport being used. For example, request/response calls can be over transports such as HTTP, but equally over messaging transports such as IBM MQ. Equally you could do a fire-forget style interaction over either transport too.

We should aim to separate these to core types, because they likely require very different configuration with regard to aspects such availability and scaling.

Cross dependencies

If an integration is completely dependent on the availability of other integrations, that might make a case for them to be deployed together. There are of course other ways to handle this situation. See Figure 7-12.

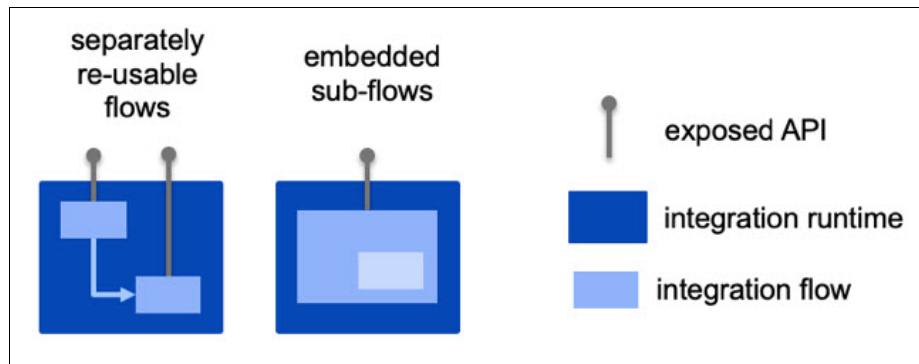


Figure 7-12 Subflows

The related integrations might be better combined by using sub flows within the same integration. However, there are times when the flows need to be separate, perhaps because they not only call one another but are also independently callable.

Scalability

With modern internet facing applications it is often very difficult to predict future workload. If a mobile application is successful it might reach enormous numbers of users. Container orchestration platforms enable elastically increasing the number of replicas and then reduce them after the load decreases. Integrations that are likely to need to scale together could be placed in the same container so that they can scale together via the same replication policy.

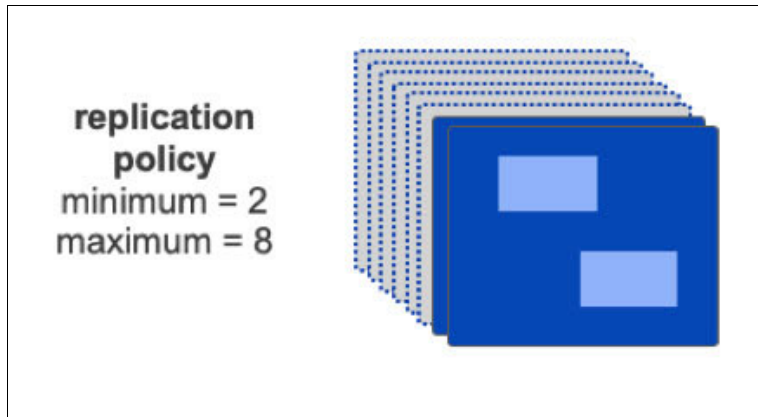


Figure 7-13 Replication

This might help with preemptive, efficient scaling; rather than scaling based on the throughput of just one interface, we could react to the sum of the throughput on all related interfaces.

However, there are downsides too. We can no longer maintain the integrations independently, nor can we provide differing scaling policies if we find that some of the integrations react differently to load.

Resilience

We might have very high availability requirements such, for example "five nines" (99.999% availability) where only 5 mins of downtime a year is acceptable. To achieve this goal, we might need to ensure that a significant number of replicas are always running. The more replicas available, the less likely a requester is to experience downtime should any individual instance fail. See Figure 7-14.



Figure 7-14 Availability

We should note of course that the number of replicas is only part of any high availability story. We also need to ensure that no single points of failure in any of the underlying resources. For example, the replication policy would also have to ensure that replicas were spread across multiple physical nodes, and of course make sure that the underlying systems behind the integration have appropriate availability themselves.

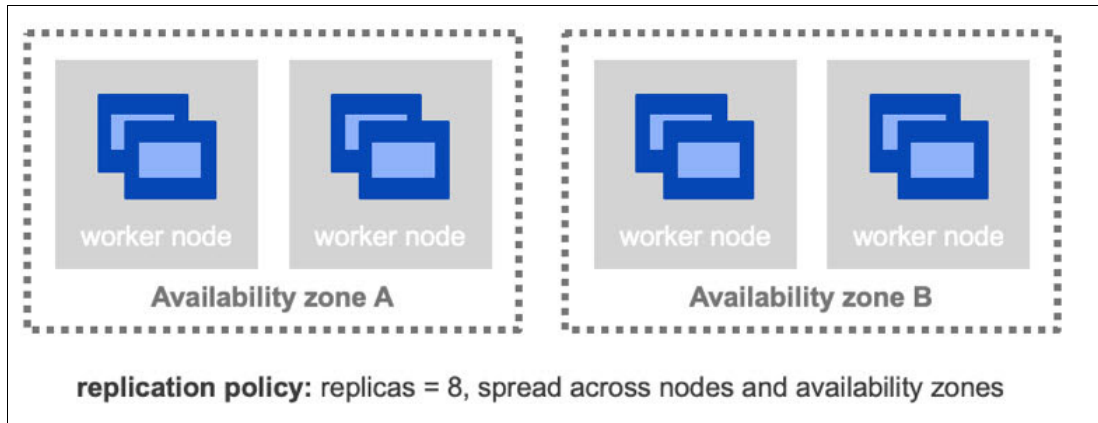


Figure 7-15 Zones

It certainly makes sense to isolate those integrations that have significantly higher availability requirements such that they can have a separate replication policy. However, we should recognize that there are diminishing returns on grouping too many integrations together in this situation. Let's consider that each integration has some probability that it could cause an outage of the whole container and all the integrations within it. The overall probability of an outage is then potentially greater than the integrations would have faced if they were deployed on their own. The more integrations in the group, the worse the availability. Ultimately for very specific high-availability requirements, a separate runtime (and therefore container, and indeed pod) for each integration is probably required.

7.2.7 Shared lifecycle

Do we have a set of integrations that always get maintained together, and released into production in synchronization with one another? These might be a good candidate to be grouped together in the same container. This might indeed make ensuring a consistent release across the set easier.

Often, this can occur where integrations use shared data models (Figure 7-16). By this we mean that they all use a single data model that must remain in step across these integrations. This sometimes happens because of data model that implements a particular versioned industry standard. Integrations that are locked in step on the same data model often have the same release cycle for major changes that accompany changes to the underlying data model version.

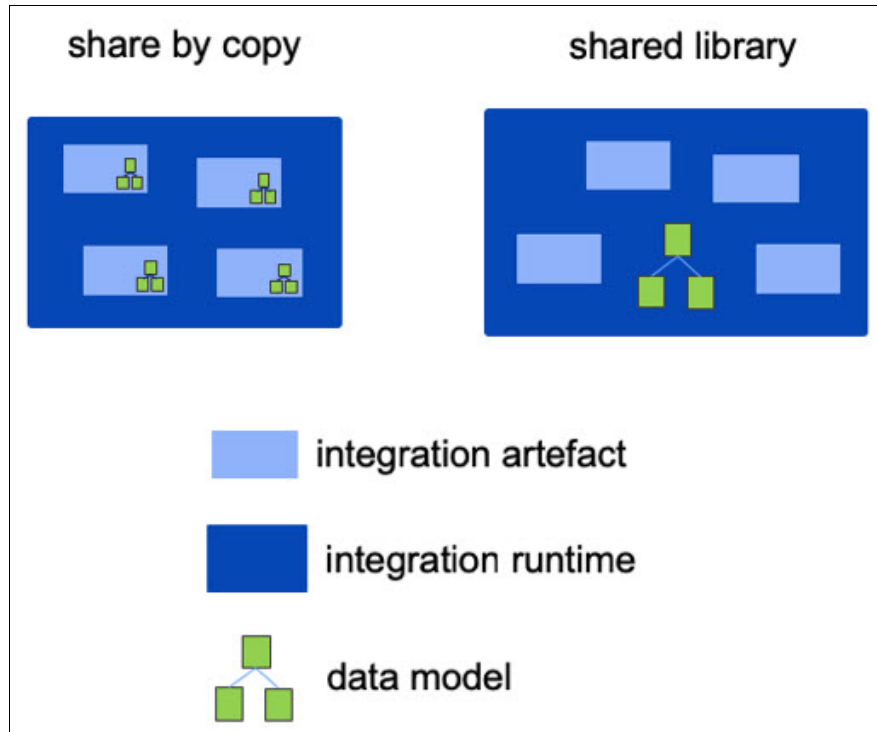


Figure 7-16 Data models

If the integrations are deployed to the same runtime, it makes it easier to deploy them consistently. Furthermore, we can also consider using a shared library within the integration runtime so we have to update only one copy and we can ensure consistency across the integrations.

Another reason for synchronized data models across integrations might be that they are tied to the release cycle of the application whose data they expose. For example, changes to an integration to incorporate extra data fields from a system of record, might require changes on the system or record itself as well as within the integration. Multiple integrations might need to change in the same release cycle as the system of record, so it might make sense for them to be deployed together.

However, the existence of a shared data model doesn't guarantee a shared lifecycle for the integrations. Although changes to data models are a very common cause of changes to integrations there are certainly plenty of other reasons integrations might need to be updated. We need to look at the history of changes to see whether there is a clear trend of the integrations being released together.

7.2.8 A worked example

Although we didn't announce it at the start, we have actually deliberately walked through the criteria for grouping integrations in roughly priority order. Let's walk through what that might look like with a fictional, but vaguely realistic example.

Let's take our insurance company example from earlier and imagine they have a centralized enterprise service bus that currently contains 100 integrations. We need to decide how best to group them into containers.

Initially we look at the core business domains, who we assume would prefer to have full ownership of the integrations that pertain solely to their aspect of the business. We find that 20 of the services relate to the Life and Pensions business domain, 40 relate to the various General Insurance policies (such as car insurance, building insurance), and 25 relate to Healthcare Insurance. We group these integrations by their business domains. The remaining 15 are cross domain services that relate to providing a single view of customer by calling the services of the other domains. We decide to bring these integrations together into a new “Customer” domain. In some cases, we refactor them where a portion of it really belonged to one of the other business domains.

We then look at each domain in turn for further opportunities to group services. Within the General Insurance domain for example we see that there are a number of synchronous integrations that retrieve data in real-time to enable status inquiries on customers insurance products, policies people have taken out, and the claims they have made. These all need to be tuned such that they provide a rapid response time for customers who use the web portal and mobile application. Thus, highly elastic scaling is required. Then there are a number of asynchronous integrations that for example process the regular payment transactions for the policies. These need to be tuned for overall throughput because they must complete before the end of day batch jobs begin. They are carefully scaled to ensure they make best use of processing capabilities in the back-end systems. But we must never overload them as that would reduce efficiency and reduce the throughput rate.

Next we notice that of all the synchronous integrations, “get quote” has a particularly critical response time. If it doesn’t respond within 2 seconds, its results are not even included by the insurance broker sites. Therefore, we decide to bring out that integration into its own separate container such that we can ensure it is as lightweight as possible for such that we can configure it for rapid elastic scaling.

We also notice that when customers are buying a new policy, they are very sensitive to outages during the application process. They will very likely go to a competitor insurance if any type of outage occurs while they are in the middle of a purchasing decision. Therefore, we choose to put all the synchronous integrations that relate to the buyer journey into a separate container so we can configure differently. We set a minimum of six replicas, spread across three availability zones, to make them appear as robust as possible even in the face of individual outages. The most critical of the integrations we break out into a container of its own to further reduce the probability of being affected by an outage of any sort.

Finally, we look at the remaining large group of integrations and spot that there are a number of them that are bound to a specific version of a payments related data model. The model is dictated by the payments partner, so if it changes, it will change for all related integrations. Looking back at the history of changes for these integrations, it seems the only times that have been changed in the last few years have been for changes to the payments data model. We choose to group them together in a container because we can see that we will almost always maintain and deploy them at the same time.

Of the remaining integrations, we leave them grouped together except for a few that are related to a pilot project we are working on which is forcing very regular changes. We separate these out so they can be independently amended at a rapid pace.

7.2.9 Conclusion

There are many different criteria that we might choose to help us break up a large number of integrations on a traditional enterprise services bus into a collection of more lightweight, decoupled containers.

We have discussed a variety of common criteria based on functional (domains) versus non-functional (scaling, availability), volatility versus stability, and various forms of dependencies (such as MQ, data models).

What's clear is that some of these criteria are overlapping so we have discussed the importance of establishing priorities, beginning with high-level business ownership and working down to more pragmatic concerns.

We have provided a simple framework for making your own decisions about grouping of integrations as you move on your containerization journey. However, we recognize that every enterprise is functionally different, with a differently evolved landscape, and different business priorities.

We should of course always aim to design our integrations such that they are strongly independent of one another. This way we can easily change our mind about the grouping decisions and implement that change with minimal risk.

7.3 When does IBM App Connect need a local MQ server?

Note: The following section was correct at the time of writing the book. However, changes are anticipated in the relatively near term that will further reduce the need for a local MQ server for IBM App Connect. For the latest information on this, check the blog post from which this section was created, which will be updated to reflect any changes when they occur.

<https://developer.ibm.com/integration/blog/2018/06/21/ace-need-local-mq-server/>

Back in 2015, with v10 of IBM App Connect (at that time known as IBM Integration Bus), the hard dependency on a locally installed IBM MQ server was removed. Since that time, we have been able to install IBM App Connect completely independent of MQ making the installation and the resultant topology lighter and simpler. With the move toward more granular installation of integrations, especially in containers, knowing exactly when we can exploit this more lightweight installation has become all the more relevant.

In this section, we explore when we can install IBM App Connect independently and, conversely, under what circumstances might we still need a locally installed IBM MQ server? Put another way, when do we really need a local IBM MQ server as opposed to when will it be sufficient to connect to remote MQ servers?

There are two very different reasons IBM App Connect makes use of IBM MQ:

- ▶ As an asynchronous messaging provider
- ▶ As a co-coordinator for global (two-phase commit) transactions

As explained later in this article, it is only the latter use of MQ that is the real driver for a locally installed MQ server, but read on as this use might be less common than you thought.

7.3.1 Benefits of being dependency-free in container-based environments

The question of whether IBM App Connect requires a local MQ server becomes more pertinent as we move to container-based deployment and we explore installation of integrations in a more granular way. Rather than having a single installation that contains

every integration, we can move to deploying small groups of integrations in isolated containers.

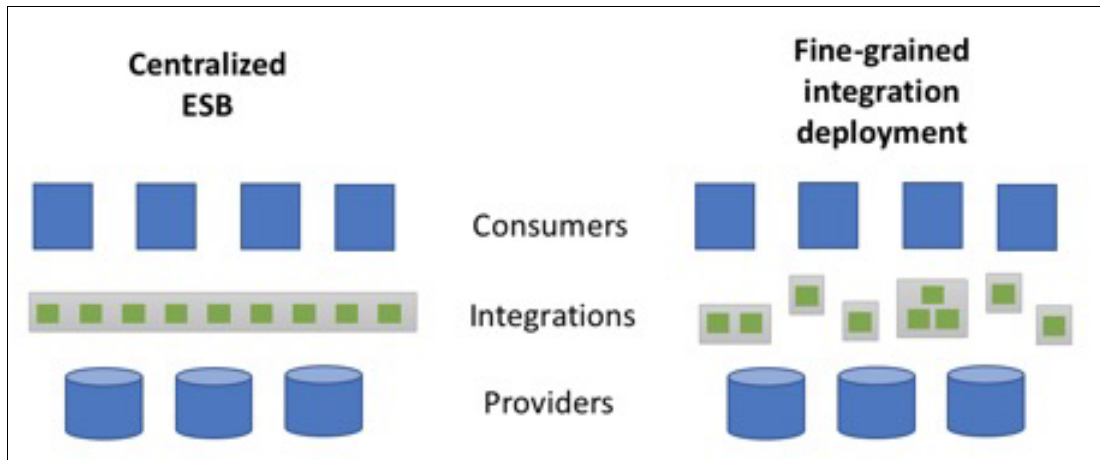


Figure 7-17 Local MQ for IBM App Connect -1

When you combine container technologies such as Docker with container orchestration facilities such as Kubernetes, you can rapidly stand up a discrete set of integrations within a container in a scalable, secure, and highly-available configuration. This represents a radically different approach to the implementation of integrations, potentially providing greater agility, scalability and resilience.

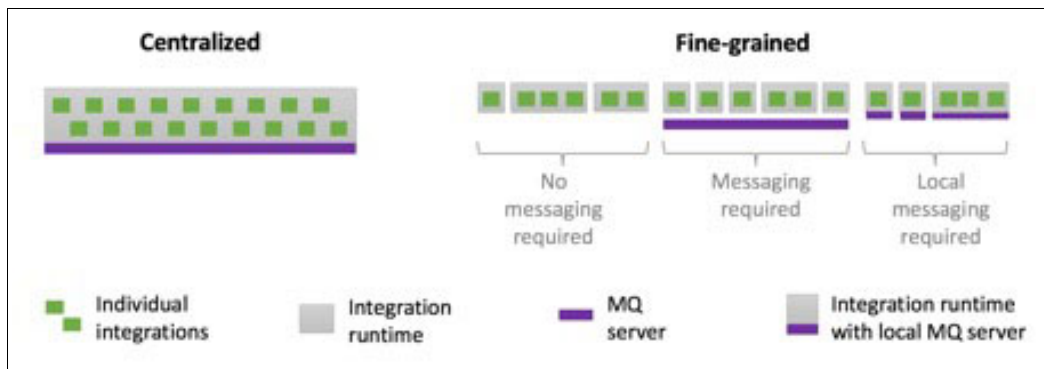


Figure 7-18 Local MQ for IBM App Connect -2

In the more traditional centralized architecture, the infrastructure must have a local MQ server, even if only a small number of the integrations require it.

In the more modern fine-grained architecture, we can decide whether a local IBM MQ server is required based on the specific needs of the small set of integrations it contains.

There are significant benefits in being able to stand up an integration server that does not need a local IBM MQ server. These benefits become particularly pronounced in a container-based environment such as Docker:

- ▶ The size of the installation is dramatically reduced, and thereby the size of the Docker image. This reduces build times due to the reduced image creation time, and reduces deployment times as a smaller image is transported out to the environments.
- ▶ The running container uses significantly less memory as it has no processes associated with the MQ server. Cloud infrastructure that is used for container-based deployment is

often charged on the basis of memory rather than CPU, so this can have a significant impact on running cost.

- ▶ Start-up of a container is much faster because only one operating system process is started; that of the integration engine. This improves agility by reducing test cycle time, and improves resilience and elastic scalability by being able to introduce new runtimes into a cluster more rapidly.
- ▶ MQ holds its message data on persistent volumes, and specific servers need access to specific volumes within the MQ topology. If IBM App Connect has a local MQ server, it becomes locked into this topology. This makes it more complex to elastically add new servers to handle demand dynamically. Once again, this makes it harder to take advantage of the cost benefits of elastic cloud infrastructure.

7.3.2 When can we manage without a local MQ server?

Clearly, the simplest case where a local MQ server is not needed is where we have a flow that does not put or get IBM MQ messages. There are around 100 nodes that can be used to create flows in IBM App Connect, and only perhaps a dozen of those involve MQ. So, there are plenty of integrations that can be created without MQ at all. We can expose and invoke RESTful APIs and SOAP-based web services, perform database transactions, read and write files, connect to enterprise applications like SAP, Siebel and much more, without the need for a local MQ server. Indeed, IBM App Connect now enables, since v11, connectivity to over 100 cloud-based SaaS applications too.

What if we are using IBM MQ, or another transactional resource (for example a database)? We should start by making a clear statement: *We do not need a local queue manager for IBM App Connect to communicate with IBM MQ queues.* IBM MQ provides an MQ client that enables messages to be placed on and retrieved from remote queues by using single-phase commit transactions, without the need for a local MQ server.

A single-phase commit is the transactional protocol used when our transaction spans only one transactional resource. For example, we want to work only with messages on a single queue manager, or perform actions on a single database. With a single-phase commit transaction, the actual transaction is essentially happening within the one resource we are talking to.

To use client-based connections when you create flows in IBM App Connect, you simply choose the client setting on the configuration of the MQInput, MQOutput, MQGet, or MQReply nodes. Alternatively, you can use an MQEndpoint policy to consistently apply the *use client-based connections* feature across multiple nodes in flows along with other MQ properties.

So, if a given flow uses only client connections to MQ, no local MQ server is required, as shown in Figure 7-19 on page 455.

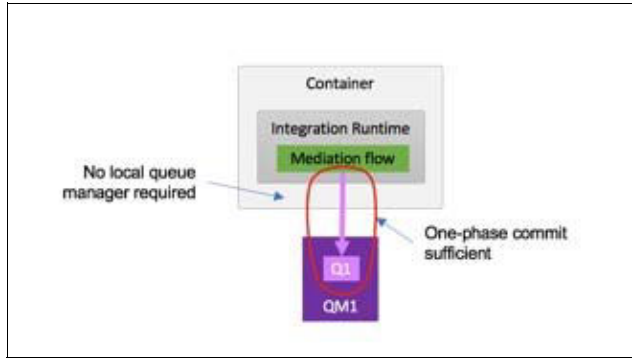


Figure 7-19 Single transaction with an MQ queue

As can be seen from Figure 7-20 the same is true of any interaction with a single transactional resource, such as a database.

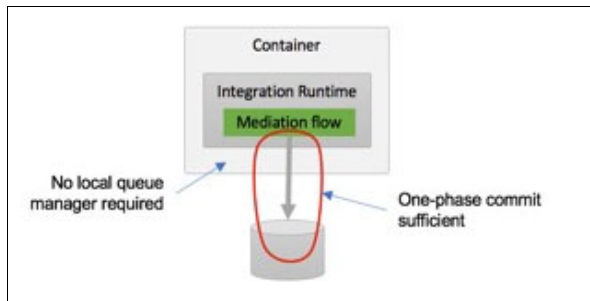


Figure 7-20 Single transaction with a database

Indeed, we can even interact with multiple transactional resources as shown in Figure 7-21. However, this is possible only if we don't need them to be combined into a single unit of work (for example if one fails, they all rollback to where they started).

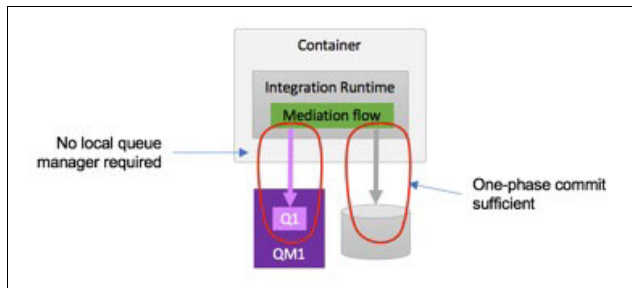


Figure 7-21 Separate transactions to a database and a queue

Assume that we do want to combine multiple resources into a single unit of work as shown in Figure 7-22. In that case, we require a local MQ server to coordinate the required two-phase commit transaction.

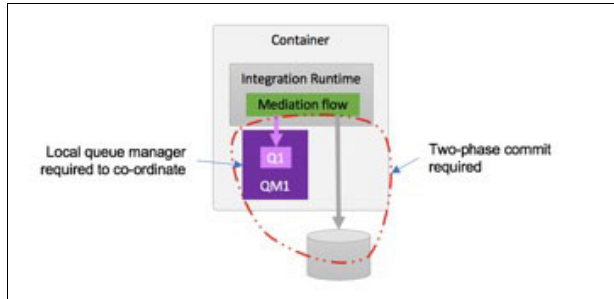


Figure 7-22 Combined transaction to a database and a queue

7.3.3 Can I talk to multiple queues in the same transaction without a local MQ server?

Starting with the simplest case, we can perform multiple MQ updates in the same transaction via a client connection as long as they are all on the same queue manager as in Figure 7-23 on page 456.

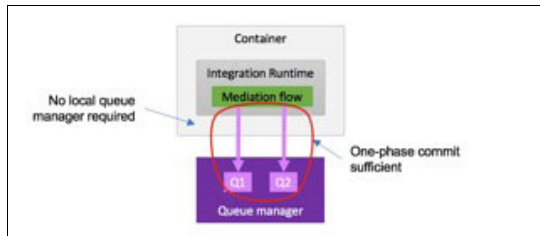


Figure 7-23 One transaction updating two queues in the same queue manager

The complete set of interactions with all queues can be committed (or rolled back) together. This is because this interaction is performed by using only a single-phase commit because the queue manager is itself only a single-resource manager.

To be clear, multiple updates to queues on the same queue manager do not necessarily require a local queue manager, and can be done over a client connection.

If we do have to talk to two separate queue managers there are three options as shown in Figure 7-24), Figure 7-25 on page 457 and Figure 7-26 on page 457. Let's look at each one separately.

In Figure 7-24, we update each queue individually. Because each transaction is a separate one-phase commit transaction, no local MQ server is required.

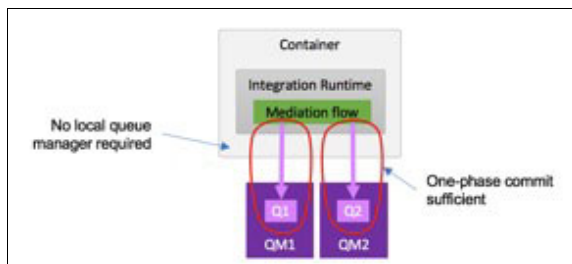


Figure 7-24 Two separate transactions to two queues in different queue managers

However, there is a small risk like the one we saw for diagram c. Something could happen in the middle of the flow such that the first transaction occurs but the second one doesn't. If this risk would be a concern — and we really need to treat the updates as a single unit of work — then we need to consider one of the other methods as shown in Figure 7-25 on page 457 and Figure 7-26 on page 457.

As shown in Figure 7-25, it is possible to configure an MQ topology. That way, IBM App Connect can perform actions across queues that reside on multiple different queue managers, still without needing a local MQ server.

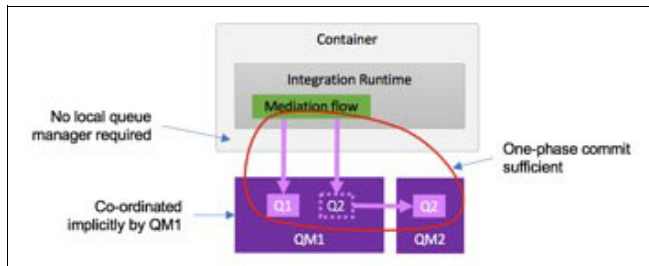


Figure 7-25 An update to two queues through use of a remote queue

The technique here is to directly connect one queue manager to the Integration Server (we can call this a *concentrator*), and make all of the queues that are involved in the transaction available on that. This is done by setting up remote queue definitions on the concentrator queue manager for the queues that reside on other queue managers. IBM App Connect can then “see” all the queues via the concentrator queue manager, and perform the multi-queue interaction over a single-phase commit, which can be done with the client connection. The concentrator queue manager then takes on the task of performing the more complex transactional and persistent behavior across multiple queue managers, using standard MQ channels to achieve the desired coordinated results.

7.3.4 Coordinating a two-phase commit requires a local MQ server

If you look back at the preceding diagram, you see that a local MQ server is required because we want IBM App Connect to combine changes to two (or more) separate resources. For example, see the queue and database in Figure 7-22 on page 456 or between two separate queue managers as in diagram Figure 7-26 that follows.

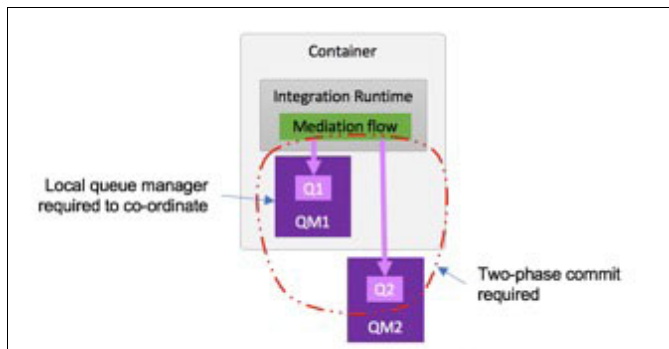


Figure 7-26 A coordinated transaction to two queues in two different queue managers

Something needs to act as a transaction manager across both resources. IBM MQ can be a transaction manager in a two-phase commit transaction on behalf of IBM App Connect, but only if it is locally installed.

The transaction manager then performs what is known as a two-phase commit, where the overall transaction is broken down into two phases:

- ▶ A prepare phase, where each of the resources makes a promise to complete the work if asked.
- ▶ A separate commit phase, where all the resources are requested to complete their individual units of work.

If any one of the resources is unable to complete in a reasonable time, then the transaction manager can request a rollback of all the involved resources.

It is for this two-phase commit transaction coordination that an IBM MQ server must be installed locally to IBM App Connect for scenarios d) and h).

What are the alternatives to a two-phase commit transaction?

A two-phase commit is not as commonly used as you might expect. This fact is surprising because a truly atomic transaction across two or more resources helps ensure data consistency. However, the reality is as follows:

- ▶ It is complex to set up, requiring the additional transaction managers for coordination of the overall transaction
- ▶ It requires the resources that are involved to trust a totally independent transaction manager. They must trust that management will be efficient with the use of locks against the resource, between the prepare and commit phases.
- ▶ It introduces considerable complexity to architect for disaster recovery, and even for high-availability configurations. Two-phase commit requires complete consistency across both the transaction logs of the transaction manager, and all the distributed resources involved. Architecting this consistency even in disaster recovery situations can be very difficult.

It is very rare to see two-phase commit between two separate systems owned (and funded) by fundamentally different parts of an organization. It is sometimes found within a single solution, where all resources (for example database and queues) are owned by the same team or part of the same product. Even in this situation, designers are often looking for alternatives.

Furthermore, modern RESTful APIs, which are increasingly becoming the predominant way that distributed systems talk to one another, work over the HTTP(S) protocol. They are not transactional on any level, let alone able to take part in two-phase commit. So, if in the future we are able to design complex solutions involving multiple systems that are available only over RESTful APIs, we will need alternative approaches to distributed transactions.

For circumstances where it is preferable to look at alternative designs to two-phase commit, here are a couple of commonly used designs:

- ▶ **Retries and idempotence:** For many scenarios, it is possible to ensure that the operations on the target systems are idempotent. For example, if the request was to process a payment, then if the same payment were submitted twice, it would still result in only one payment being processed. With idempotent target systems, we might be able to remove the need for two-phase commit. We might simply ensure that we perform retries until success occurs. This provides eventual consistency as opposed to absolute consistency. However, that approach might not be appropriate for your use case.
- ▶ **Saga or compensation pattern:** The saga pattern was introduced in 1987 as a way to handle distributed transactions across independent systems. It works by bringing together atomic actions, either by chaining them, or by having a central orchestrator. If one of the systems should fail, then a compensating action is performed on all those systems that

have already processed the event, in a non-transactional way. This pattern has been implemented in many forms over the intervening years. Business Process Execution Language (BPEL) is an example of an orchestration-based saga implementation. Equally, chains of message flows – each one asynchronously leading into the next – could also be set up to perform a choreography style saga implementation. There is a resurgence of interest in the saga pattern in recent years due to the highly distributed nature of microservice applications.

So there certainly are alternatives (such as those by design that were previously mentioned), and also more practical product specific mechanisms (such as that discussed earlier for handling updates to queues across multiple queue managers). Ultimately, whether we choose an alternative to two-phase commit will depend on how much benefit we believe we stand to gain from the increased simplicity of the IBM App Connect topology.

7.3.5 When else do I need a local MQ server?

In addition to the use cases discussed so far, at the time of writing a small collection of features in IBM App Connect require a local MQ server. These are the aggregation nodes, the timeout nodes, and the Collector, Sequence and Resequence nodes. We are deliberately discussing this separately from the preceding for two reasons:

- ▶ New versions of these nodes are being developed and introduced into App Connect in the relatively near term to remove the dependency on local MQ for common scenarios. Depending on your migration timescales, these current dependencies might not actually present an issue.
- ▶ Although these nodes currently require local MQ, they do not necessarily imply the same stateful persistence that would be required for the preceding two-phase commit scenarios. For example, an aggregation node could be gathering information from a number of resources in parallel. The node might not need persistent queues in order to respond to a caller in real-time. Queues in this instance are just being used for efficiency of multi-threading, not for their assured delivery properties. So, we might need a local MQ server, but have a low level of concern about persistence. Therefore, we could use a simpler, more stateless topology.

So, in short, the existing dependency on local MQ for these nodes is quickly becoming less important.

7.3.6 Why do we have so many integrations that use server connections?

It is common to find older integration flows that use server connections to a local MQ even when they are not required. This is largely because a local MQ server in the past could be assumed to be present. It also provided some extra comfort at a time when networks outages were more common, because you know that a message would at least reach the local queue and the messaging system would eventually take care of delivery. Today, networks are generally more reliable and the need for local MQ servers is much reduced. Indeed, many customers are significantly simplifying their MQ topologies, moving to more centralized options such as MQ Appliances to host all their queue managers in high-availability pair. So, perhaps with some minor reconfiguration, many existing integrations could simply use client-based connections instead of server connection, and they would no longer have a dependency on a local queue manager.

7.3.7 Conclusion

As we have seen, many scenarios can be achieved without the need for a local queue manager. The single main exception is where IBM App Connect is required to perform a two-phase commit transaction. However, there are viable alternatives to the two-phase commit. This type of commit is not mandatory for scenarios with multiple queue managers or combined database and queue updates. The benefits of being able to independently administer and scale our IBM App Connect and IBM MQ topologies could make those alternative patterns very attractive. Add the fact that with a more fine-grained integration deployment approach, we don't have to build a one-size fits all topology. We can instead just introduce local MQ server for the integrations where it is most needed.

7.4 Mapping images to helm charts

Important: Since the writing of this IBM Redbooks publication, the IBM Cloud Pak for Integration has embraced Kubernetes Operators (<https://coreos.com/operators/>). This significantly simplifies how components such as an Integration Server are installed and maintained, extending the features provided by Helm. There is more information and an excellent video demonstrating this new capability here:

<https://developer.ibm.com/integration/blog/2020/06/28/ibm-app-connect-operator-1-0-is-now-available/>

It does unfortunately mean that some of the instructions describing the deployment of App Connect Enterprise in containers within this section are now out of date, and will need to be adapted to the use of operators. We may well look to update the book, but in the mean time, refer to the product documentation to find information on the new features.

In the example scenario we built out in chapter 6, we deployed integrations by using the new convenience capabilities of the consolidated user interface for the IBM Cloud Pak for Integrations (see section 6.2, “Application Integration to front a data store with a basic API” on page 173). However, some more advanced users will likely want more control over the deployment specifics. So, they might want to deploy their own images, by using their own topology definitions (helm charts). This section discusses what is available from IBM to accelerate this approach.

Note: From a future-looking point of view, we should note that a further complementary option known as a Kubernetes Operator (<https://kubernetes.io/docs/concepts/extend-kubernetes/operator>) is also becoming important in this space. However, at this time, it is appropriate to focus on the current technology of helm charts.

In a container orchestration environment, topologies are not built by their users. The topology requirements are defined declaratively in files that can be stored alongside the code. This ensures that the integrations are always deployed onto a topology that suites their needs.

Helm charts are currently the de facto standard for providing the logical definition of the requirements from the deployment topology. For example, they define how it should respond to changes in workload, then leave the orchestration framework to work out how to build that out and keep it running.

In a containerized approach, the container consistency is being achieved by referencing to standardized container image templates (and helm chart templates).

A key objective in the approach for standardized container-image templates for IBM App Connect is to reduce the number of Helm Chart templates, while ensuring that the required use cases have been covered.

In Figure 7-27 we see a need for three primary starting point images for IBM App Connect.

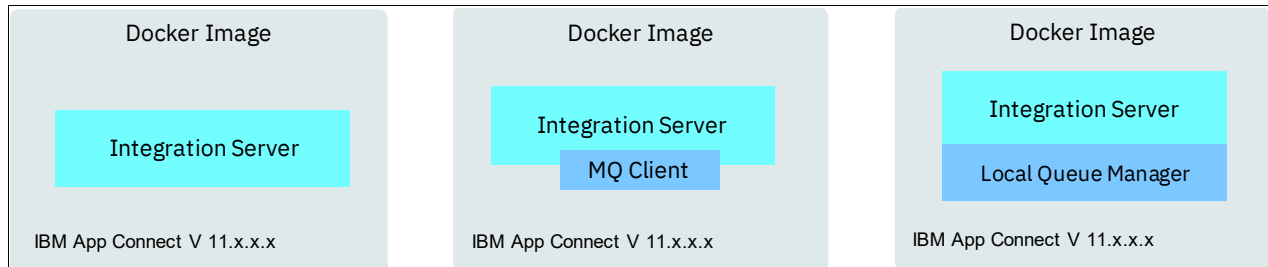


Figure 7-27 Three primary starting point images for IBM App Connect

The first two, contain the core IBM App Connect runtime binaries (V 11.x.x.x), and optionally including the MQ Client depending whether it is required. These two images provide the smallest possible footprint and startup time, keeping resource usage to a minimum, while satisfying the highly stateless use cases.

The last image in Figure 7-27 will apply for a use case, where an MQ queue manager is required locally. Typically, for reasons of two-phase commit transactionality and including for some of the more complex message dependent nodes, a different image would be the starting point. This would typically (though not always) infer a persistent volume on which to store messages and transaction logs, resulting in a different set of deployment patterns.

IBM provides a set of images with IBM App Connect based on the preceding scenarios. The available images can be accessed on the Docker Hub (<https://hub.docker.com/r/ibmcom/ace/>) and can be used straight away.

If you want to further customize these images for your own purposes, the Dockerfile from which they are built is available on GitHub: (<https://github.com/ot4i/ace-docker/blob/master/ubi/Dockerfile.aceonly>)

7.4.1 Developing helm charts for Kubernetes

IBM provides a helm chart repository that is available at <https://github.com/ot4i/ace-helm> that contains a stable repo with publicly available versions of many IBM software products. IBM also encourages others to publish their software charts in the community section of the repo.

To help build quality helm charts, the repository includes detailed recommendations about how to build a proper helm chart. The following repository document and IBM Redbooks refer to IBM Cloud Private, but most of the considerations around customizing helm charts apply to Cloud Paks, too:

- ▶ (<https://github.com/IBM/charts/blob/master/GUIDELINES.md#developing-helm-charts-for-ibm-cloud-private>)
- ▶ A detailed guide for building helm charts is covered in an IBM Redbooks *IBM Cloud Private Application Developer's Guide*, SG248441.

IBM Cloud Pak for Integration provides helm charts out-of-the-box that can be configured to point to the three primary starting point images as described.

In order to create your own helm chart, several methods can be used. The following section outlines the process of generating and deploying a helm chart based on a Docker image using the simplest method: the use of the Helm CLI command, using based on a helm template that is provided.

Docker image

Create your own Docker Image or use one of the provided set of images on Docker Hub. As discussed earlier, this Image can contain one of the preceding variations with the required version of IBM App Connect. In addition, the Docker Image can contain your BAR file, which streamlines the containerization approach.

1. Create an empty image stream in your project for the image by using `oc create image stream`:

```
$ oc create imagestream myimagestream
```
2. In order to be able to push this Docker image to the OCP (OpenShift Container Platform) registry, tag your image with ICP cluster information. For example:

```
# docker tag ace:11.0.0.0 mycluster.icp:8500/default/ace_bar:11.0.0.0
```
3. Push the image to the OCP registry. For example.

```
# docker login mycluster.icp:8500  
# docker push mycluster.icp:8500/default/ace_bar:11.0.0.0
```
4. Navigate to the ICP admin console and verify that the Docker image is now shown at **Cluster Console** → **Builds** → **Image Streams**.

Create your own helm chart

We now have an image that can be used with a helm chart for Application Integration.

1. Start by downloading the base code of the helm package for IBM App Connect from GitHub:
<https://github.com/ot4i/ace-helm>
It is expected that this base Helm template will be updated and tailored by users, based on their requirements.
2. Unpack the downloaded .zip file (**ace-helm-master.zip**) and navigate to subdirectory to get the `Chart.yaml` file.
3. Edit the `Chart.yaml` file to give a unique name to the helm chart that you want to create. For example: `name: ibm-ace-bar-dev`.
4. Edit the `values.yaml` to provide the name of the Docker image that you just pushed to the local OCP registry in the preceding section. For example:

```
image:  
# repository is the container registry to use, which defaults to IIB docker  
registry hub image  
repository: mycluster.icp:8500/default/ace_bar  
# tag is the tag to use for the container repository  
tag: 11.0.0.0
```
5. To verify that the helm chart directory name and structure are correct, run the `lint` command as shown below. Make sure that the name of the top-level directory where the helm package files are stored, matches with the name of the chart specified in the `Chart.yaml` file:

```
# helm lint ibm-ace-bar-dev
==> Linting ibm-ace-bar-dev
Lint OK
1 chart(s) linted, no failures
```

6. Now package the helm chart:

```
# helm package ibm-ace-bar-dev
Successfully packaged chart and saved it to:
/root/ACE/ace-helm-master/ibm-ace-bar-dev-1.0.0.tgz
```

7. Load the helm chart in the Cloud Pak catalog:

Note: This requires the Cloud Pak CLI to be installed as specified in 5.1.4, “Getting access to IBM Cloud Pak for Integration for the exercises” on page 145.

```
# cloudctl login -a https://mycluster.icp:8443 --skip-ssl-validation
# kubectl load-helm-chart --archive ibm-ace-bar-dev-1.0.0.tgz --clustername
mycluster.icp
Loading helm chart
{"url":"https://icp-management-ingress:8443/helm-repo/charts/index.yaml"}
OK
Synch charts
{"message":"synch started"}
OK
```

8. The chart that was just published will now be seen in the Cloud Pak admin console, in the Catalog. Use ‘local-charts’ as a filter in the Catalog menu.

Using helm chart metadata

Cloud Pak catalog provides some other features to make the helm charts easier to use. One of these features is formatting and validating the values that the chart makes available for users to customize.

You can define metadata for the parameters in *values-metadata.yaml* file that is placed in root directory of the chart (same as *values.yaml*). The file *values-metadata.yaml* defines the metadata for values defined in *values.yaml*. It should mirror the same structure of *values.yaml*, except that instead of specifying a value for the leaf property, define a property named *__metadata*.

If the *values-metadata.yaml* file is not present, the UI continues to display all parameters that are declared in *values.yaml* based on the type inference. See Example 7-1.

Example 7-1 *values.yaml*

```
example:
  __metadata:
    label: Example
    description: New Version
  stringField:
    __metadata:
      label: String field
      type: string
      required: true
  numberField:
    __metadata:
      label: Number field (with validation)
```

```
type: number
required: true
```

7.4.2 Upgrading (extending) helm charts

As your application evolves, the application images and helm chart changes versions. Helm provides a convenient way to upgrade existing releases with new content. The same command is used for two purposes:

Upgrade to new version of helm chart (for example, to define new Kubernetes resources).

Update the resources with new values (without changing the helm chart version).

To upgrade the release, run the following command:

```
helm upgrade <release> <chart>
```

Helm upgrade can take new values that are specified with

```
-f <new_values.yaml>
```

or using

```
--set key=value option.
```

You must specify only the new values to be used for the upgrade. All of the previous values that are used before in release are carried forward. For example, run the following command to update only the container image tag that is used by the chart:

```
helm upgrade <release> <chart> --set image.tag=v1.0.1
```

To avoid this behavior and reset all the values to default values that are included in the chart, add `--reset-values`.

If any of the values change, the pod specification template within deployment definition (for example, the image tag as shown in the preceding section), triggers creating a ReplicaSet. If the update strategy that is specified in deployment is RollingUpdate, the deployment controller gradually scales up new ReplicaSet and scales down old ReplicaSet. This process ensures that the required number of application pods is available in any point. As the result, the application should be upgraded in place, without a downtime that is visible for users. An alternative update strategy is to Recreate. In that case, the old ReplicaSet and pods are deleted and a new one is created.

Customizing helm charts

To allow users to customize or extend the chart deployment, it is possible to replace static parts within a template with references to the content of the values.yaml file. The values.yaml file can define whatever names you want by following a hierarchical YAML structure.

In the following example we show customization of a helm chart by adding additional ports.

1. In the `service.yaml` file you can add a section under 'ports', to specify a new port with a name 'switch1'. The value of the port number will be taken from the values.yaml file (values.service.switch1). Example 7-2.

Example 7-2 Customization of a helm chart

```
spec:
ports:
  - port: {{ .Values.service.switch1 }}
```

```
targetPort: {{ .Values.service.switch1 }}
protocol: TCP
name: switch1
```

2. In the values.yaml file, under the 'service' part in the file, along the default ports, a port for 'switch1' is defined, which will be picked up by service.yaml file.

```
service:
  type: NodePort
  webuiPort: 7600
  serverlistenerPort: 7800
  serverlistenerTLSPort: 7843
  switch1: 9010
```

3. Finally, in order to see the fields in the helm charts WebUI for the newly added port the values-metadata.yaml file should also define 'switch1' port as a parameter.

```
service
  switch-1:
    __metadata:
      label: "switch-1 port"
      description: "This is port number example"
      type: "number"
      immutable: true
      required: false
```

7.5 Continuous Integration and Continuous Delivery Pipeline using IBM App Connect V11 architecture

Important: Since the writing of this IBM Redbooks publication, the IBM Cloud Pak for Integration has embraced Kubernetes Operators (<https://coreos.com/operators/>). This significantly simplifies how components such as an Integration Server are installed and maintained, extending the features provided by Helm. There is more information and an excellent video demonstrating this new capability here:

<https://developer.ibm.com/integration/blog/2020/06/28/ibm-app-connect-operator-1-0-is-now-available/>

It does unfortunately mean that some of the instructions describing the deployment of App Connect Enterprise in containers within this section are now out of date, and will need to be adapted to the use of operators. We may well look to update the book, but in the mean time, refer to the product documentation to find information on the new features.

This section looks at IBM App Connect in the context of CI/CD pipelines. The theory of CI/CD pipelines is explored, building upon 4.2.10, "Continuous Integration and Continuous Delivery and Deployment" on page 97, leading into an example pipeline that displays a practical implementation.

When we talk about modernization of your Integrations, an important topic for discussion is "pipelines". Many IT organizations already have an existing Pipeline implementation in one form or another. These pipelines can be triggered either manually or automatically and as an input it could serve a source or a whole BAR file.

When discussing Pipelines, the key terminology we use, is defined below:

- ▶ Continuous Integration (CI) – this refers to build and test, whenever new code is available. This type of testing will show if the new code has had any impact or broken anything.
- ▶ Continuous Delivery (CD) – this phase refers to when we are pushing to production from a continuous stream. After the code is decided to be ready, it is pushed to production.
- ▶ Continuous Deployment – similar to the continuous delivery, this outlines an automated push to production, without human intervention, where the code is stopped going live only by failed tests.

We see three main stages in running your Pipeline when you use IBM App Connect V11 architecture.

Development

As the Pipeline structure usually involves multiple products, this stage is expected to cover your development work towards building the components required. For example, as shown in the figure below, the development work might include creation of MQ resources, BAR files and packaging. Further, Secrets and Docker files are being developed, and finally Helm Chart development and storing. Figure 7-28 on page 466.

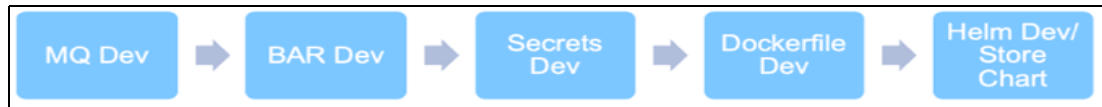


Figure 7-28 Pipeline development

Build

The Build part of the Pipeline includes the build, validation and storing of the BAR file, followed by the build, storing and validation of the Image. We see that some of the steps can be combined as shown in the figure below, depending on the build approach. See Figure 7-29 on page 466.



Figure 7-29 Pipeline build

Deploy

The creation of secrets might already be complete in the final steps of the Pipeline because they were created as part of the development process. The final stage includes the update of the Helm Chart Values and its deployment. See Figure 7-30.

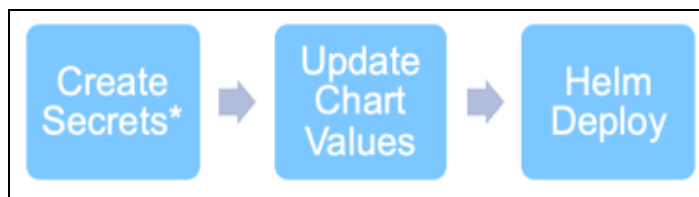


Figure 7-30 Pipeline deployment

Please refer to 7.5.5, “Practical example” on page 475 for a practical Pipeline scenario built during the stages that were described previously.

7.5.1 Continuous Integration Delivery and Deployment

CI/CD pipelines have increasingly been adopted by many organizations, as the switch to DevOps implementations has been made from traditional SOA and waterfall implementations. As an integration tool, there are specific considerations that need to be made when you incorporate IBM App Connect into a CI/CD pipeline.

To help us understand these considerations, it is helpful to briefly explore the different meanings of CI/CD.

<https://www.ibm.com/blogs/cloud-computing/2018/11/27/continuous-integration-vs-continuous-delivery/>

CI: Continuous Integration

Continuous Integration is a DevOps practice where developers commit code into source control. This is combined with a process comprised of automated tools that check and verify this code each time a commit is made. The benefits of this approach are that code changes from multiple developers can be tested together more frequently, meaning that problems can be spotted far earlier.

CD: Continuous Delivery

Continuous Delivery (CD) is centered around the delivery of working applications and updates to these applications (such as bug fixes and new features) to consumers as quickly and safely as possible. Because the delivery is 'continuous' there are no fixed release cycles, instead code is produced and is made available to be pushed to production as soon as it is ready.

CD: Continuous Deployment

Continuous Deployment is the automation of releases to production. In this case only the failure of a testing stage in the Continuous Deployment pipeline prevents the deployment of new code to production.

IBM App Connect: Continuous Delivery versus Deployment

As an integration technology IBM App Connect will, in most cases, be deployed into production environments that are not naturally suited to a continuous deployment solution. Consideration needs to be given to a number of factors prior to deploying new integration services to a production environment, both from a technical and business point of view. For example, an organization might choose to only perform production updates in specific time windows when customer traffic is reduced in order to reduce risk.

On the technical side, dependencies will need to be appropriately managed. Validation is needed to ensure that the systems to which a new integration service connects are at the correct versions in production. Failure to ensure this could lead to errors and to the need to back out an update.

The CD model produces new integration services and makes them available for deployment to production via a separate process. This model is a more flexible approach that fits more naturally with integration scenarios and IBM App Connect. However, the examples in the previous paragraphs illustrate why the amount of CD that is used will be different for each organization. Some will prefer a higher level of automation in production, and others will prefer more control and flexibility.

7.5.2 Example pipeline - High-level concepts

Figure 7-31 describes the stages of an example CI/CD pipeline for IBM App Connect. These have been highlighted in different logical stages that could be split out into smaller sub pipelines depending on the level of granularity required. The section explains the different stages:

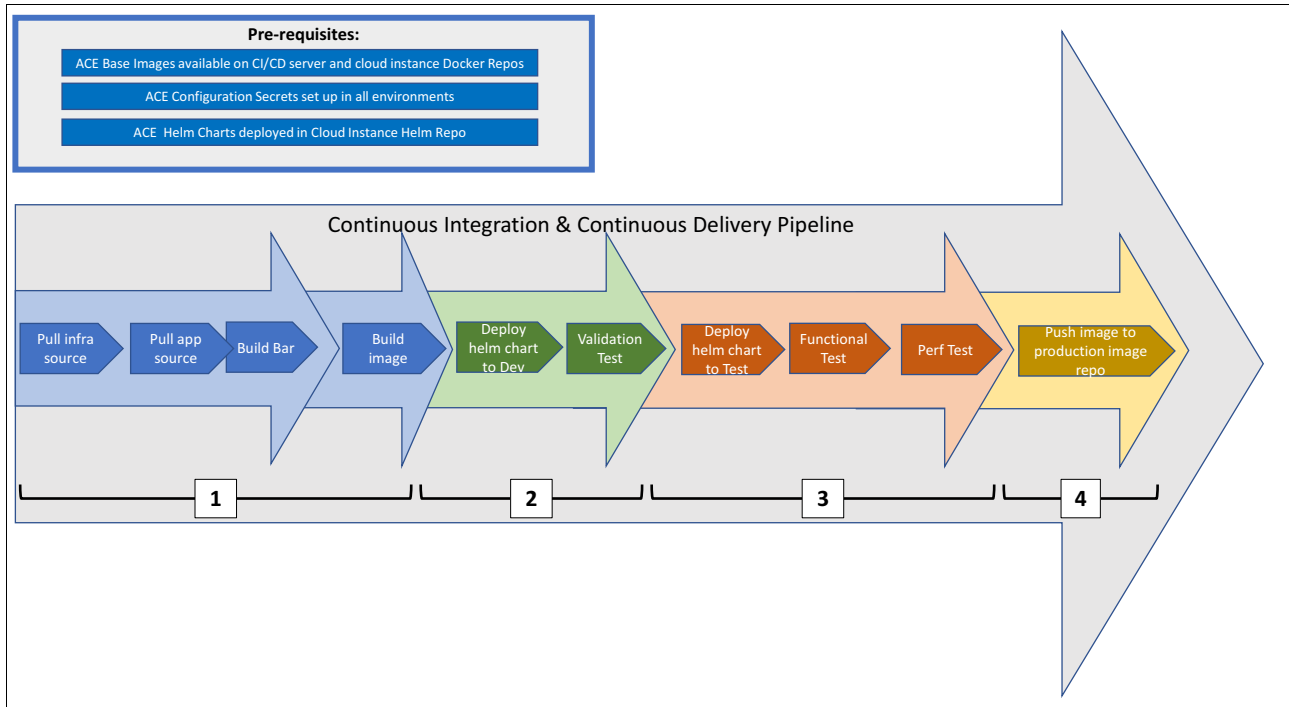


Figure 7-31 CI/CD pipeline stages for IBM App Connect

Prerequisites

The pipeline has the following prerequisites:

- ▶ IBM App Connect base images (on top of which the microservice images are created) are available on the CI/CD automation server and the cloud instance.
- ▶ IBM App Connect configuration secrets have been created in each Kubernetes environment where the microservice will be deployed.
- ▶ IBM App Connect helm chart has been deployed the cloud instance Helm repository.

Section 1

This section looks at pulling down the source code and the creation of deployable artifacts, be this a BAR file or Docker image.

Pull infrastructure source

This stage pulls down the infrastructure source. This includes resources that describe the pipeline and run the pipeline build (for example a *Jenkinsfile*, which is a type of declarative pipeline), scripts and other resources that are run by the pipeline and resources — other than application code — that are used to build the IBM App Connect microservice image (such as a copy of the *ace-docker-master* folder from **ot4i**).

Pull Application Source

This stage pulls down the application source code. This source is comprised of resources that are packaged into a BAR file, such as message flows, subflows, schemas, swagger files, and ESQL files. If the `mqsipackagebar` command is used in your pipeline to create your BAR file then any Java applications and message sets must already be compiled and checked into the application source code repository here:

https://www.ibm.com/support/knowledge/en/SSTDS_11.0.0/com.ibm.etools.mft.doc/bc31730_.htm

Build bar

This stage takes the application source code pulled down in the preceding stage and turns it into the broker archive file. In the example pipeline, this is affected by running an IBM App Connect base Docker image locally on the CI/CD automation server. A directory with the application source is mounted into the container, and the `mqsipackage` BAR command is run to create the BAR. After the BAR file has been retrieved from the mount this local container is stopped and deleted.

In this stage, a push of the BAR file to an artifact repository could happen; however, for the example pipeline this has not been implemented.

Build image

The *Build Images* stage builds a new Docker image from the IBM App Connect base image. In this stage the BAR file is retrieved (be this from the CI/CD automation server file system, or external artifact repository) then it makes it available for the Dockerfile that build the image. The result of this process is an IBM App Connect microservice image. This image is stored in the Docker registry on the CI/CD automation server and also pushed to the Docker repository on the cloud instance, where the Helm deploy will be able to use it.

Section 2

This section deals with the deployment of the microservice to the first environment, in this case a development environment, and the running of a test.

Deploy helm chart to dev

Perform the `helm update` command to install (the `-i` flag is specified) or, if the deployment already exists, update the deployment of the microservice. Prior to `helm upgrade` command a new `values.yaml` file is created, which references the IBM App Connect microservice image created in Section 1. This new `values.yaml` is supplied as an argument to the `helm upgrade` command with the `-f` parameter.

Configuration information is picked up from a secret created in the Kubernetes namespace, prior to the invocation of the pipeline (see the prerequisites).

Verification test

This stage runs a script that sleeps to wait for the container and the Integration Server to initialize. Then the script runs a `curl` invocation against the Integration Server's administration port. It checks for a correct response which indicates that the Web UI would be rendered in a web browser.

The verification test stage can also include the scanning of the new image for vulnerabilities. On IBM Cloud Pak this happens automatically when images are pushed the Docker registry on the Cloud Pak instance.

Section 3

This section looks at deployment to a test environment and the running of automated tests.

Deploy helm chart to test

This stage performs the same set of steps as the deployment to the development environment. The only differences are that the helm upgrade command will either be pointed at a different Kubernetes namespace, or at a different Kubernetes cluster, depending on the configuration of the environments.

The values.yaml used for this deployment could be altered to increase the number of replicas of the microservice pod, so that heavier non-functional testing can be performed.

Once again, configuration information is picked up from a secret created in the Kubernetes namespace prior to pipeline being invoked.

Functional test

This stage runs a functional test against the deployed microservice. In the example pipeline, this stage has been left blank, as it is merely to display where this type of test could fit into the CI/CD pipeline with IBM App Connect. You might want to spin up a container on Kubernetes to host a test harness. Containers could also be provisioned to host stubs as part of the testing.

Test cases could be included with the application source (pulled down in stage 1) or these could be retrieved from a central test case repository.

Performance test

This stage runs a performance test. Once again, this stage has been left blank in the example pipeline.

Section 4

Sets out the steps that happen at the end of the pipeline.

Push image to production image repo

This stage shows that, at the completion of the CI/CD pipeline, the tested IBM App Connect microservice Docker image is pushed to a registry, making it available for production deployment. This illustrates that this is a Continuous Delivery pipeline, whose purpose is to make images that are ready for production deployment, not a pipeline that does the deployment to the production environment.

7.5.3 CI/CD pipeline in depth

Figure 7-32 on page 471 shows the concepts of the CI/CD pipeline, up to the first container deployment in greater depth.

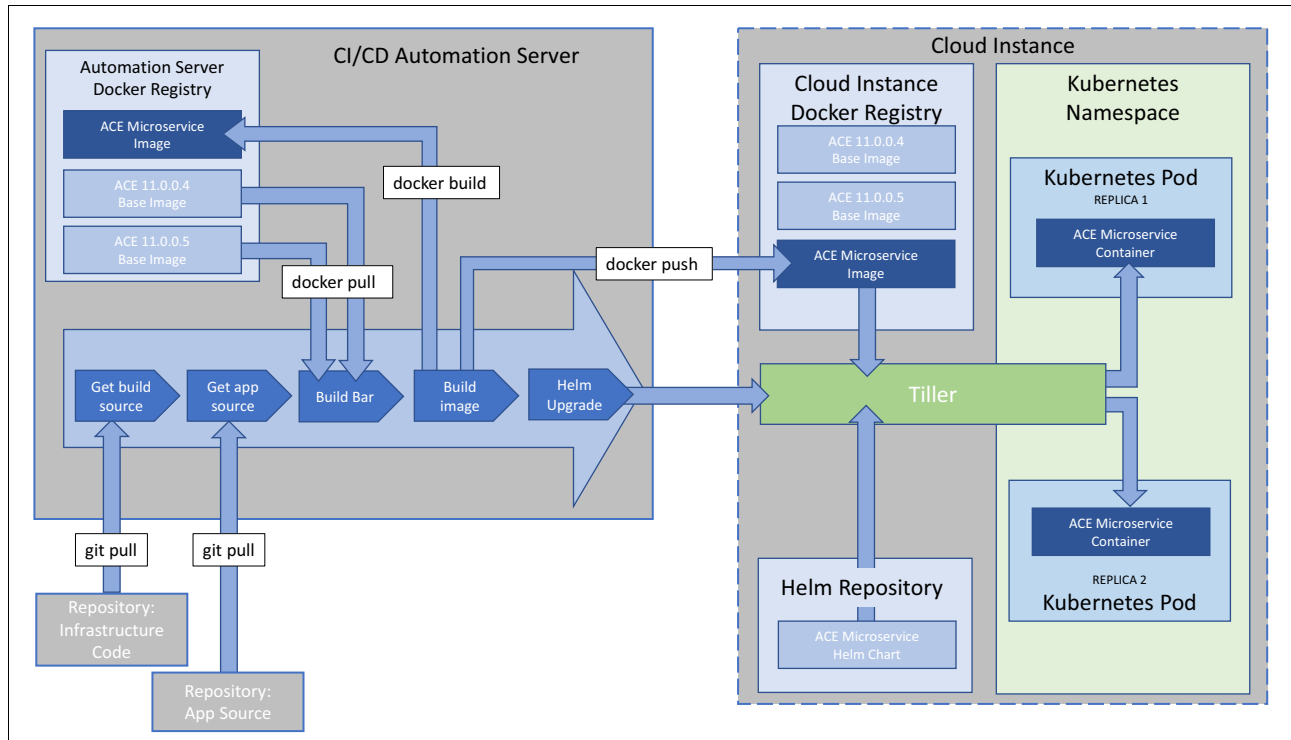


Figure 7-32 In depth look at the CI/CD pipeline: deployment to first environment

The key understandings from this diagram are:

- ▶ The infrastructure source and application source are stored separately in different repositories. These perform different roles and this is discussed later in this section.
- ▶ The build of the BAR file relies on an IBM App Connect base image being available on the CI/CD Automation Server.
- ▶ The IBM App Connect microservice image created by the pipeline is stored locally on the CI/CD Automation Server and pushed to the Cloud Instance
- ▶ The helm upgrade command instructs the Helm Tiller to deploy the IBM App Connect helm chart, resulting in the microservice IBM App Connect image being deployed in a Kubernetes Pod.

7.5.4 Considerations for CI/CD pipelines with IBM App Connect

In this section we discuss considerations for CI/CD pipelines with IBM App Connect.

Triggering the CI/CD pipeline

With automation servers such as Jenkins, you can trigger a build of the pipeline in a multitude of ways. Triggering builds based on commits to the application source code repository master branch (and/or other branches of your choice) is one way to approach this. Other options include the use of a cron job or by polling source code management on a certain interval and running a build if changes are detected.

The answer of the question “*How and when should I trigger my pipeline?*” is one that will vary from one organization to the next, affected by the capacity of their CI/CD automation server to run builds and the number of changes they are making to their source code each day.

The example pipeline created for this book is invoked manually.

Source code: infrastructure and application

The preceding diagrams indicate two separate steps for source code. Firstly, the infrastructure source and, secondly, the application source. Both of these should be stored in separate source code repositories. These are distinct artifacts and treating them as such allows for a more controllable and scalable CI/CD solution.

Separating infrastructure source code from application source code reduces complexity. Changes to infrastructure and applications inevitably move at different speeds. Where applications can be updated multiple times in a single day, infrastructure normally remains comparatively constant. Organizations are unlikely to modify their pipeline structure at the same rate that changes are made to applications.

What artifacts can be built, stored, and reused?

The first stage of a continuous delivery pipeline will normally consist of retrieving the relevant source code, for the infrastructure and applications. It will then go on to produce a series of artifacts that can be stored, deployed and propagated through the pipeline to a repository from which it can be deployed to production.

So far, three main artifacts have been identified that can be built and reused as part of a pipeline solution:

- ▶ BAR files
- ▶ Docker images containing an IBM App Connect microservice
- ▶ The working directory

BAR files

The BAR file represents the traditional unit of deployment for IBM App Connect and its predecessors. It remains a valuable artifact to store as part of a CI/CD pipeline. It would be a reasonable step for BAR files created by the pipeline to be pushed to an artifact repository, this way BAR files created by the pipeline could be made available to support personnel, both internal to the organization and to IBM support. This has the advantage that BAR file can be deployed to support personnel's locally defined Integration Servers. Additionally storing BAR files in an artifact repository means that they can be pulled down from the repository and deployed independently to an IBM App Connect instance in non-cloud environments. This could be especially useful in organization operating a hybrid-cloud environment where some integration middleware remains on-premises.

It would make sense that BAR files are pushed to a third repository separate to the infrastructure and application source code repositories. This allows a clear delineation between source code and built code. Artifact repositories such as JFrog Artifactory and Nexus have both been used for this purpose in some implementations.

While storing BAR files that have been created by the pipeline is useful, by itself this does not allow us to take full advantage of IBM App Connect's move to a containerized architecture.

Docker image containing IBM App Connect microservice

Docker images containing IBM App Connect and a deployed application, the microservice image, can now be considered the unit of deployment for a CI/CD pipeline. As part of the example pipeline in this section the IBM App Connect microservice image is stored in two places: in a Docker registry on the automation server and in a Docker registry on the cloud instance. The example also shows that the image can be stored in a further *production Docker registry* after it has successfully progressed through the CI/CD pipeline.

The rationale behind focusing on the microservice image as the central artifact in the CI/CD pipeline is that this is the artifact that gets deployed and ran in production. This follows on from the ideas discussed in the Continuous Adoption sections of this book. In this scenario

developers will locally test their code against IBM App Connect base images. The CI/CD pipeline will then build a microservice image. This image eventually ends up being deployed to production. Images produced this way can be deployed to different environments as the organization deems necessary. All the infrastructure to run the application (except for of the configuration secrets and any config maps) is included in this self-contained image.

Support engineers can pull down microservice images and run them locally to inspect their behavior without having to worry about differing environment setups. The container they pull down will have IBM App Connect that runs on the same platform and same fix pack as the container that has seen an issue in production.

While this approach does have advantages in keeping a stable unit of deployment, it does place additional upskilling requirements on support and development teams. Both these teams will have to gain fundamental Docker skills. In some cases this might not be possible, perhaps due to contractual reasons, where support is handled by a third party, or simply due to a lack of time and resources. The third approach provides a model that has been used successfully, where Docker skills are not widely available,

The working directory

Starting with IBM App Connect version 11 the concept of a working directory for the Integration Server was introduced. This directory contains all the configuration information for the Integration Server process in addition to all the files for deployed BAR files. The **mqscreateworkdir** command is used to create this directory:

https://www.ibm.com/support/knowledgecenter/en/SSTTDS_11.0.0/com.ibm.etools.mft.doc/createworkdircmd_.htm

A pipeline stage could consist of a Jenkins slave container — in which IBM App Connect is installed (for example: a base image) — being run on the CI/CD automation server. The **mqscreateworkdir** command would then be run to create a working directory. Optionally, configuration could be deployed into this working directory, such as a new `server.conf.yaml` file and policy files (these can also be deployed via Kubernetes secrets). The BAR file would then be unpacked into the run directory by using the **mqsibar** command. At this stage the working directory can be compressed and exported out of the slave container, creating a reusable and deployable artifact. This can be stored in a repository in the same way that was mentioned for BAR files.

Later in the CI/CD pipeline (or in a separate pipeline entirely) the working directory is retrieved from the repository and uncompressed into the IBM App Connect microservice Docker container as part of that container's image build.

An advantage of this approach is that support personnel can pull down compressed working directories and then unpack them onto their workstation's local file system. A simple Integration Server command then can be run to start the Integration Server. Unlike with the Docker image-centric approach, support engineers do not need to run Docker locally on their workstation. This reduces the initial investment required to move to an IBM App Connect on Cloud setup. However, it does not offer the same level of assurance that developers and support personnel will work with the closest artifact as possible to what is deployed in production. For example, a working directory might be created against one fix pack version of IBM App Connect. But a support engineer might pull it down to their workstation and run the Integration Server with a different fix pack level. With this implementation additional care must be taken to manage these factors. For example, you can implement a manual process to ensure that developers use an IBM App Connect base image on their workstation that is at the latest fix-pack level.

How many pipelines?

The example pipeline provided in this chapter shows a single pipeline that manages the progression from source code up to a production Docker repository. This is a valid pattern; however, it is not the only one. Other patterns might include having the creation of the BAR file in a separate pipeline from the creation of the microservice image. The advantage here is that images can be built selectively - with a specific BAR file (not every BAR file build needs to result in the building of an image). This also opens the opportunity for the deployment of multiple BAR files into one microservice image. Further CI/CD pipelines might be required in order to orchestrate the deployment of an integration test environment, in which multiple IBM App Connect microservices might be deployed alongside other types of services.

Credentials - secrets and vaults

There are several configuration objects for IBM App Connect that require credentials and other sensitive information to be provided:

- ▶ Policies can contain credentials for establishing secure connections to data sources like JDBC
- ▶ Keystores
- ▶ Truststores
- ▶ mqsisetdbparms storing credentials for keystores and data sources like ODBC
- ▶ Web user account IDs and passwords

The IBM App Connect helm charts on ot4i show how you can generate a Kubernetes secret to hold the preceding information:

<https://github.com/ot4i/ace-helm/tree/master/ibm-ace/scripts>

The use of a secret to hold this information does mean that any user with access to the Helm Tiller can access the contents of the secret (<https://github.com/helm/helm/issues/2196>). Therefore, in order to proceed with this model, appropriate steps should be taken to ensure that only administrators or cluster administrators have access to the tiller.

Products like Hashicorp Vault provide extra functionality to enable the creation and maintenance of secrets.

Testing in the CI/CD pipeline

Automated testing is a central part of the CI/CD pipeline. It is of critical importance to the whole endeavor, making sure that code releases actually work and do not cause any breakages further down the line.

Advantages of automated testing in the CI/CD pipeline

<https://smartbear.com/learn/automated-testing/the-continuous-development-pipeline/>

The idea of automated testing is not new. To many organizations this approach has been part of their test cycle for a number of years. However, the introduction of containerization and the CI/CD pipeline does offer the opportunity to run these tests on a far more frequent basis. And the environment is more controlled than is normally the case in an on-premises setup.

For example, in an on-premises setup, automated tests might fail due to the incorrect setup of a shared test environment, where artifacts for the previous test run were not cleaned up properly. The risk of such a situation happening with containers is reduced, enabling a far tighter feedback loop - test setup takes far less time, meaning that tests can be run more frequently.

It is important to mention this stage that manual testing, of course, still has a place. It still plays a valuable part in many test cycles. Exploratory testing, for example, is difficult to automate, and will remain a manual task.

Types of testing in the CI/CD pipeline

Different organizations will have varying ideas and thoughts on what types of testing to include in the CI/CD pipeline for their IBM App Connect microservices. Such tests could include a simple verification test (check that the IBM App Connect container is running), integration test, component test and performance test. The decision will be based on the answer to this question: What testing does your organization believe needs to be performed to make a microservice image ready for production repo?

Notice that *Unit test* has been left out of this list. In the example provided with this chapter, it is assumed that developers have performed unit testing locally on their workstation, for example, by using the IBM App Connect Flow Exerciser.

7.5.5 Practical example

As part of this book an example CI/CD pipeline for IBM App Connect has been developed.

This scenario was developed with a Jenkins server installed on an Ubuntu virtual machine. The Ubuntu instance sat outside of the Kubernetes cluster to which it was deploying the helm chart. The Kubernetes cluster was hosted on IBM Cloud Pak for Integration, but the concepts are equally applicable to Red Hat OpenShift.

Prerequisites

Your Jenkins server must have the following elements:

- ▶ Docker installed locally.
- ▶ Install the Cloud Pak and OpenShift command-line interfaces (CLIs) as specified in 5.1.4, “Getting access to IBM Cloud Pak for Integration for the exercises” on page 145.
- ▶ The jenkins user (the user that your Jenkins server runs as) must be able to run Docker commands on the server. In this example the jenkins user was given `NOPASSWD sudo` permission so that it could run Docker commands. In a live environment an alternative strategy should be employed to make sure that the appropriate security levels are in place.
- ▶ You must have uploaded an IBM App Connect only base Docker image to your OCP registry. This can be built based on the instructions on the ACE-docker ot4i Git repository: <https://github.com/ot4i/ace-docker>
- ▶ IBM App Connect override-able configuration must be deployed in a secret on your Kubernetes workspace prior to running this pipeline. The documentation on the following web page shows how to do this by using a script supplied by IBM (see section “*Installing a sample image*” and the `generateSecrets.sh` script.): <https://github.com/ot4i/ace-helm/blob/master/ibm-ace/README.md>

Note: At the time of writing the secret name must be `ace-secret-dev`, as this is what is hard-coded in the `value.yaml`.

Restrictions

The example uses a non-secure mechanism for administering credentials, intended only for use in a sandbox environment. The Jenkins server on which the example was developed had

three files defined locally (not in source control), with permission locked down to the Jenkins user only. These files are:

- ▶ `git.txt` - holds a personal access token for GitHub. Used when the pipeline pulls down the application source.
- ▶ `oc-login.sh` - logs the shell into the Cloud Pak instance and sets up the Helm and Kubernetes command-line environment.
- ▶ `dockerlogin.sh` - logs the shell into the Docker registry hosted on the Cloud Pak instance.

In order to run the pipeline you will need to create your own solution for managing these credentials.

Links

Two GitHub repositories exist as part of this example pipeline:

- ▶ <https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration-CICD-infra>
This repository contains the infrastructure source code and the Jenkinsfile.
- ▶ <https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration-CICD-appsrc-db2>
This repository contains sample application code, to create a microservice that connects to a Db2 database.

Perform a `git clone` to make your own copies of these repositories and then push them to your own Git source control.

Scripts

The infrastructure source code repository contains a number of resources that describe and execute the pipeline.

Jenkinsfile

The Jenkinsfile (a type of declarative pipeline) has nine stages that correspond to the parts of a CI/CD pipeline build that were discussed earlier in this section:

1. Clone - pulls the infrastructure source code from GitHub to a working directory on the Jenkins server.
2. Build and Verify BAR File - runs the `pullSource.sh` and `barBuild.sh` scripts
3. Build Microservice Docker Image - runs the `dockerBuild.sh` script
4. Deploy Microservice to Dev Environment - runs the `deployChart.sh` script
5. Verification Test - runs the `pingService.sh` script
6. API-Test - runs the `apiTest.sh` script
7. Deploy Microservice to Test Environment - example stage that does not run. Would invoke `deployChart.sh` to deploy to a different Kubernetes namespace
8. Functional Test - example stage for a functional test
9. Performance Test - example stage for a performance test
10. Commit to Production Docker Repo - example stage to show when a test IBM App Connect microservice Docker image would be pushed to a production registry, ready for deployment.

The Jenkinsfile has the following variables that must be configured depending on the microservice that is to be built:

- ▶ **applicationName** - name of the application, default of 'ace-db2-test-01'
- ▶ **applicationSourceRepoName** - name of the Git repository containing the IBM App Connect application source, default of 'ace-db2-app-source-02'

- ▶ **helmChartName** - default of 'ibm-ace-server-icip-prod'
- ▶ **helmChartVersion** - default of '1.1.2-icp4i-jenkins-01'
- ▶ **aceBaseImageName** - base IBM App Connect Docker image to build FROM (also used to run container for BAR file build), defaults to 'ace-base-11004'
- ▶ **aceBaseImageTag** - tag of IBM App Connect base image, default of latest

pullSource.sh

Downloads IBM App Connect application source code from a Git repository specified in the `applicationSourceRepoName` variable in the Jenkinsfile.

barBuild.sh

Builds a BAR file from source code retrieved by `pullSource.sh`, by temporarily running an IBM App Connect container locally on the Jenkins server with the source code mounted into that container.

dockerBuild.sh

Builds a new Docker image FROM the base image, adding in the BAR file built by `barBuild.sh`.

deployChart.sh

Performs Helm deployment. A new `values.yaml` file is created for each deployment, referencing the image made for the build. The secret containing the IBM App Connect override-able information is currently hard coded in the `values.yaml` as 'ace-secret-dev'.

pingService.sh and apiTest.sh

These scripts perform two tests, one to ping the service's admin port and one to make a series of calls to the API by using `curl`.

Set up the pipeline in Jenkins

After you have installed the prerequisites and configured a solution to manage credentials to your application source code repository, Docker registry and your Cloud Pak CLI tools you can proceed to set up the example pipeline in Jenkins.

1. Sign in to your Jenkins server and on the home page click **New Item**. See Figure 7-33 on page 478.

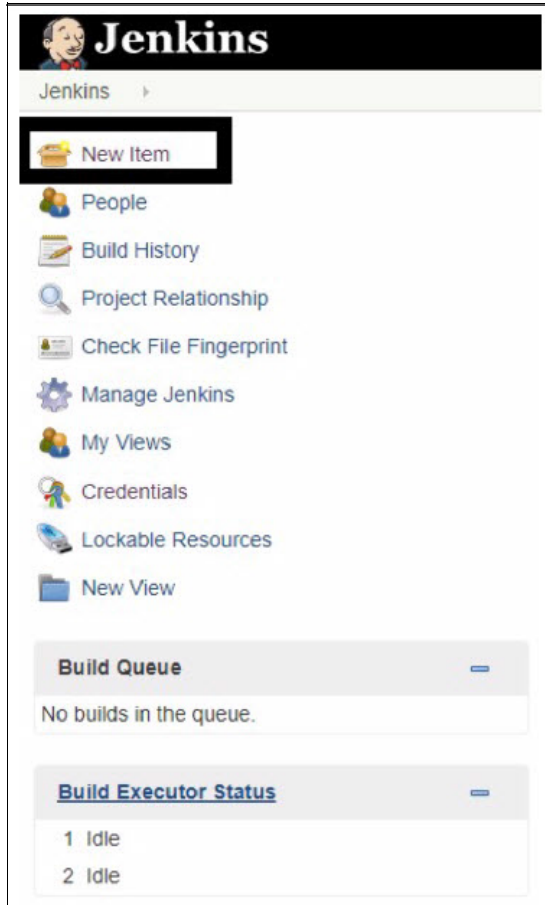


Figure 7-33 Set up the pipeline in Jenkins -1

2. On the next panel, enter a sensible name, select **Pipeline** and click **OK** as shown in Figure 7-34 on page 479.

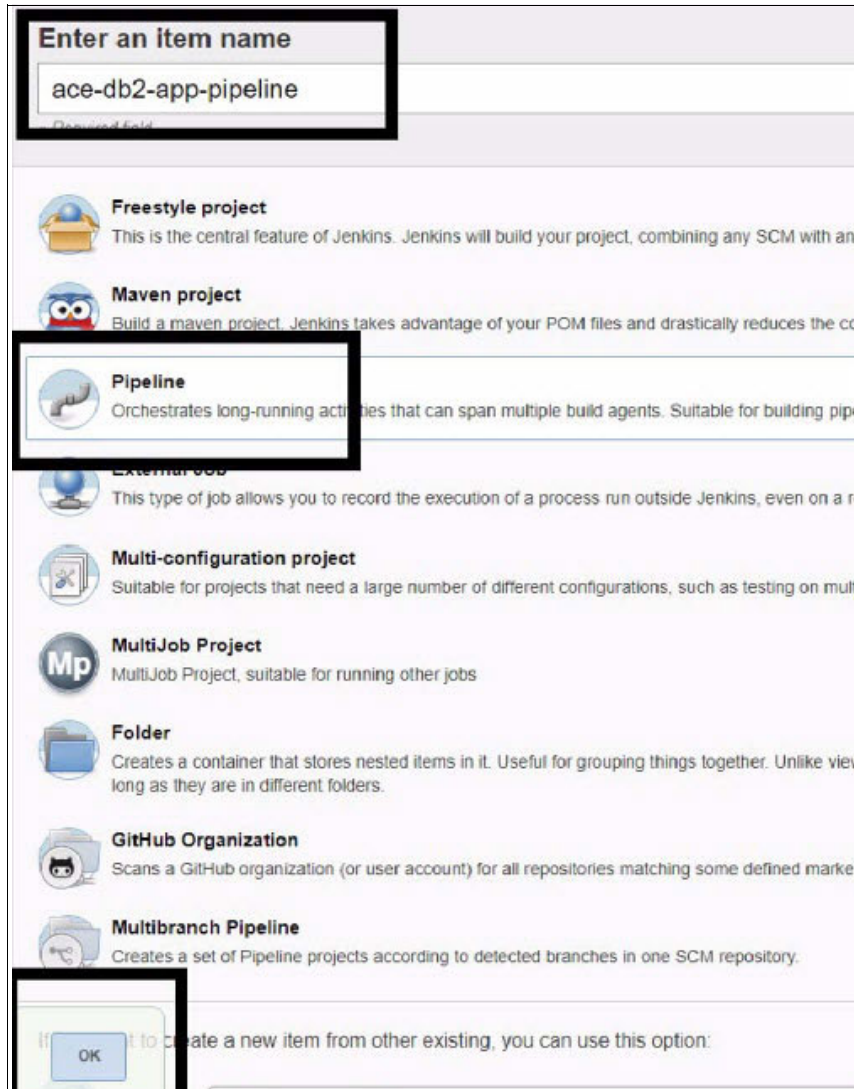


Figure 7-34 Set up the pipeline in Jenkins -2

3. The next page displays. Scroll down to the **Pipeline** section. In the Definition field click the drop-down menu and select **Pipeline Script from SCM**. See Figure 7-35.

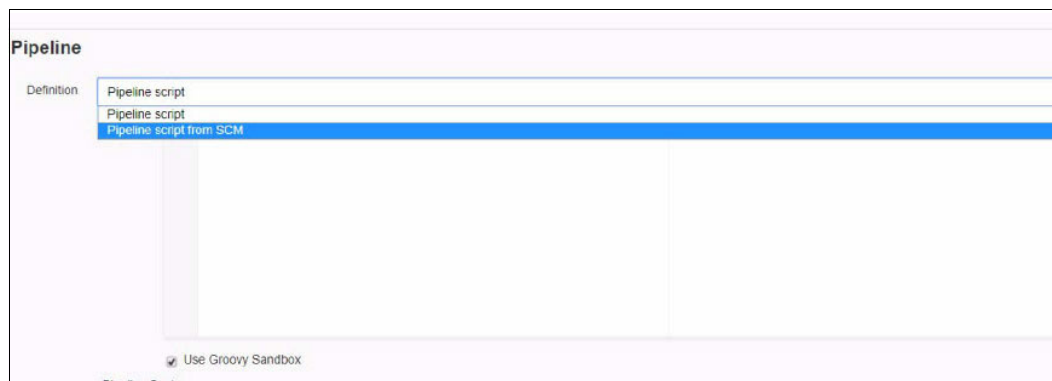


Figure 7-35 Set up the pipeline in Jenkins -3

4. When this has been selected a new section appears. Configure the following options:

- ▶ **SCM: Git**
- ▶ **Repository URL:** add in the URL for your copy of the infrastructure repository.
- ▶ Select the Jenkins credentials you have configured to allow connectivity to the Git repository.

If you do not already have this configured you can click **Add**. Select the Username and Password option, where the username is your username for the Git Repository and the password is a Git personal access token.

- ▶ Leave the branch specifier as ***/master**.
- ▶ Specify the Scriptpath as **Jenkinsfile**.

5. Click **Save**. See Figure 7-36.

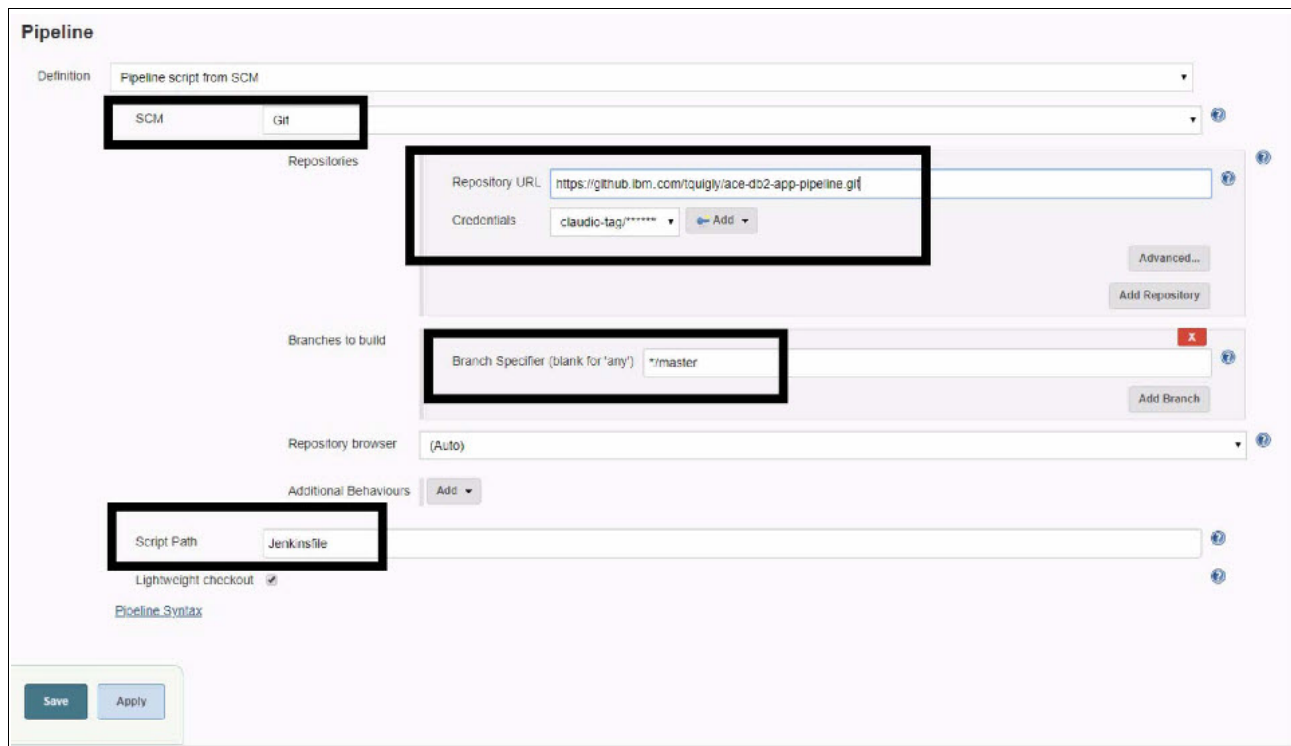


Figure 7-36 Set up the pipeline in Jenkins -4

Your pipeline's home page is displayed.

Running the example pipeline

1. To run the example pipeline click **Build Now**. See Figure 7-37 on page 481.

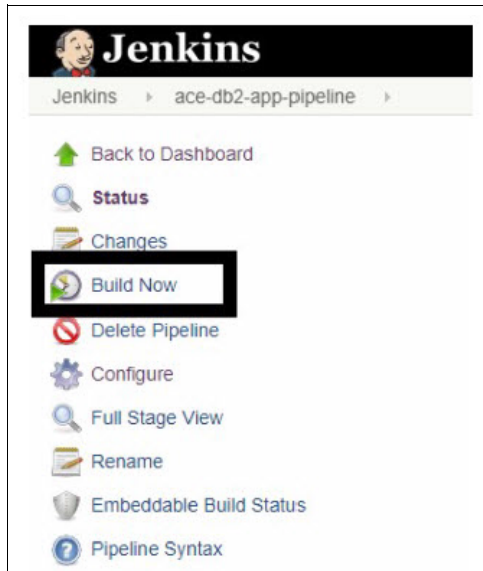


Figure 7-37 Run the pipeline in Jenkins -1

The build will be initiated and it will display in the **Build History** section of the build's home page as shown in Figure 7-38.

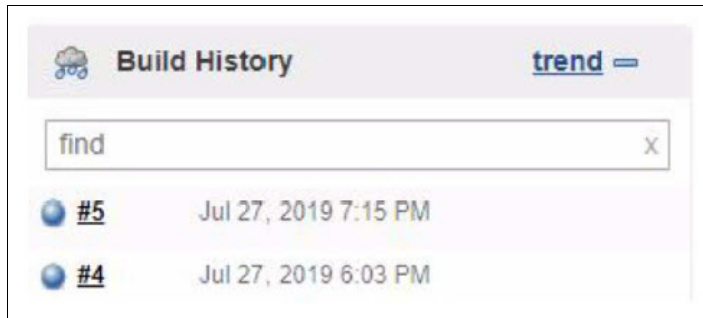


Figure 7-38 Run the pipeline in Jenkins -2

2. Click the build number to see more information about the build. The build page will be displayed. Click **Console Output** to see the log from the build as shown in Figure 7-39 on page 482.

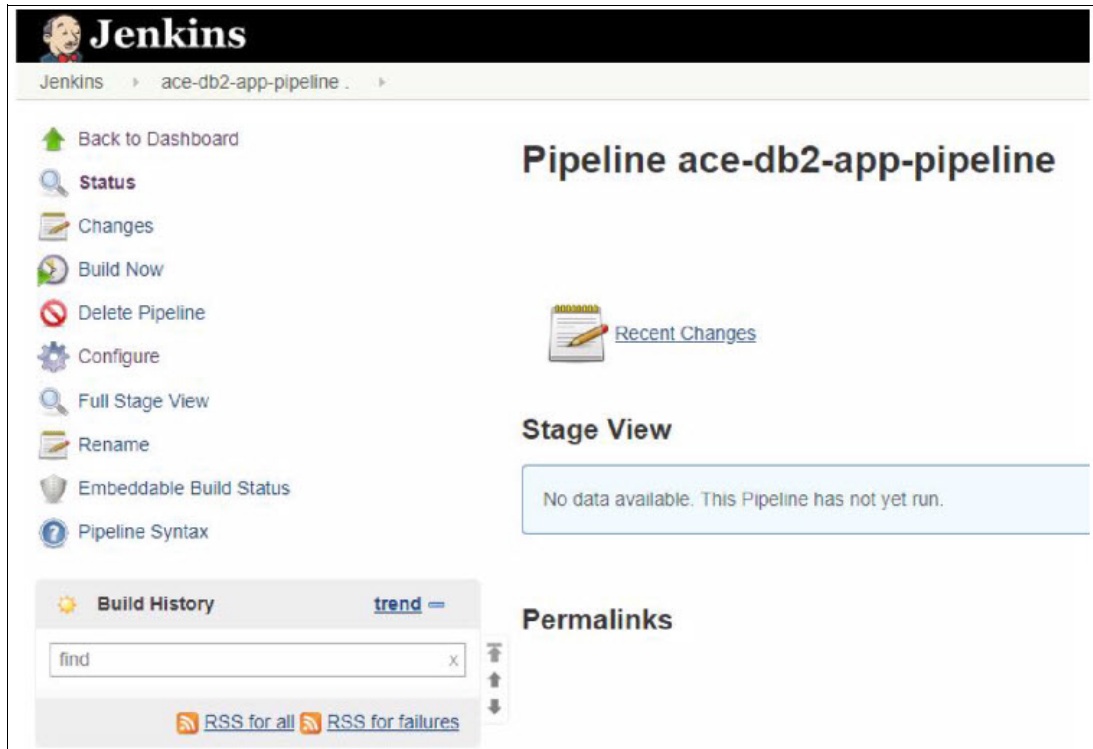


Figure 7-39 Run the pipeline in Jenkins -3

3. This will display a log file from the build, where you will be able to see the output of the various scripts executed in the build. For example, in the image we can see the execution of the `barBuild.sh` script in Figure 7-40 on page 483.


```

+ chmod 700 barBuild.sh
[Pipeline] sh
+ ./barBuild.sh ace-db2-app-source-02 ace-base-11004 latest
sourceRepo is: ace-db2-app-source-02
baseImageName is: ace-base-11004
baseImageTag is: latest
sourceDir is: ace-db2-app-source-02-5/ace-db2-app-source-02
thisDir is: /var/lib/jenkins/workspace/ace-db2-app-pipeline
barTag is: satjul27191602cest2019.5
bar_file is: database_query.satjul27191602cest2019.5.5.bar
appName is: database_query
total 4
drwxr-xrwx 6 jenkins jenkins 4096 Jul 27 19:16 ace-db2-app-source-02
total 16
-rwxr-xrwx 1 jenkins jenkins 23 Jul 27 19:16 README.md
drwxr-xrwx 2 jenkins jenkins 4096 Jul 27 19:16 DefaultPolicies
drwxr-xrwx 3 jenkins jenkins 4096 Jul 27 19:16 database_query
drwxr-xrwx 2 jenkins jenkins 4096 Jul 27 19:16 database_queryJava
Spin up ACE container and mount in source folder
sudo docker run --name aceserverbarbuild --detach --env LICENSE=accept --env ACE_SERVER_NAME=AC
mount type=bind,src=/var/lib/jenkins/workspace/ace-db2-app-pipeline/ace-db2-app-source-02-5/ace
source-02/,dst=/home/aceuser/app-source mycluster.icp:8500/icp4i/ace-base-11004:latest
eb66f579b1dbf0c09da9cbf5835d77f51a96db279090e1c458c34e5ba395dfa5
barBuildContainerID is: eb66f579b1db
mqsiprofile repetition disallowed
Successfully added file 'gen/database_query.msgflow' to the BAR file.
Successfully added file 'restapi.descriptor' to the BAR file.
Successfully added file 'getProducts.subflow' to the BAR file.
Successfully added file 'swagger.json' to the BAR file.
BIP1849W: The application 'database_query' references project 'Product' which does not exist.
Successfully added file 'database_queryJava-old.jar' to the BAR file.
Successfully added file 'db2jcc4.jar' to the BAR file.
BIP1849W: The application 'database_query' references project 'Products' which does not exist.
Application file 'database_query.appzip' successfully added to the BAR file.
BIP1863W: Command completed with warnings.
-rw-rw-rw- 1 icpuser 1001 12351608 Jul 27 19:16 /var/lib/jenkins/workspace/ace-db2-app-pipeline
source-02-5/ace-db2-app-source-02/database_query.satjul27191602cest2019.5.5.bar
eb66f579b1db
eb66f579b1db
Bar file database_query.satjul27191602cest2019.5.5.bar would be pushed to repository
barBuild.sh completed building bar file database_query.satjul27191602cest2019.5.5.bar
[Pipeline] }

```

Figure 7-40 Run the pipeline in Jenkins -4

The overall status of the build is displayed on pipeline home page as shown in Figure 7-41 on page 484.



Figure 7-41 Run the pipeline in Jenkins -5

4. Access the OpenShift Container Platform **Web UI** → **Go to Application Console** → **Select the namespace** where the helm chart was deployed. Go to **Applications** → **Services** → **Select the service** that contains the word "jenkins". Note the NodePort mapping to the ACE web UI. See Figure 7-42.

Route / Node Port	Service Port	Target Port	Hostname	TLS Termination
30857	→ 7600/TCP (webui)	→ 7600	none	none
31668	→ 7800/TCP (ace-http)	→ 7800	none	none
31861	→ 7843/TCP (ace-https)	→ 7843	none	none

Figure 7-42 Run the pipeline in Jenkins -6

5. The Web UI will be accessible at the following address: `https://<cluster IP/FQDN>:<Web UI NodePort>`. The IBM App Connect Web UI will launch in a browser session. The application deployed as part of the pipeline will be displayed on the Web UI. See Figure 7-43.

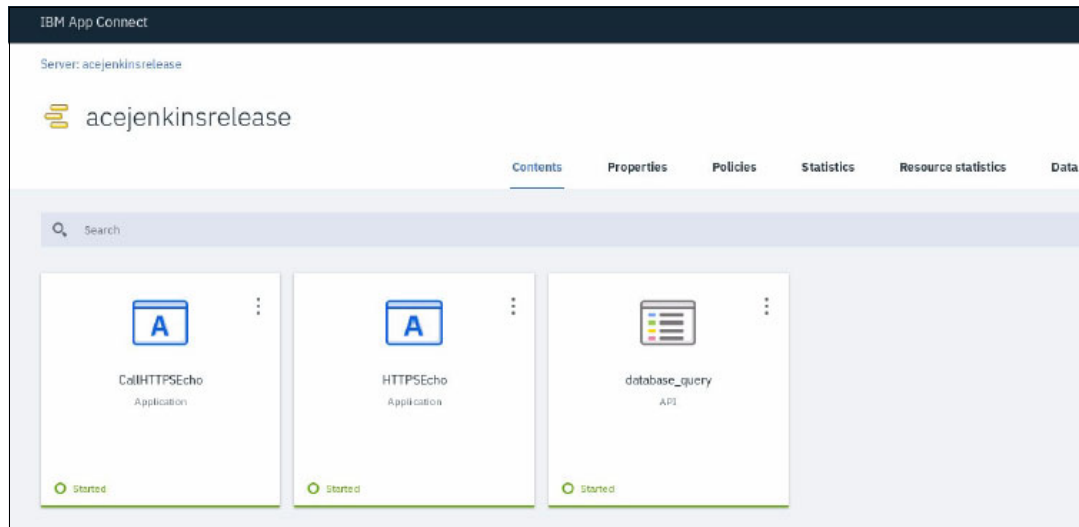


Figure 7-43 Run the pipeline in Jenkins -7

7.6 Continuous Adoption for IBM App Connect

Continuous Adoption, also known as *Evergreening*, is the automation and regular implementation of updates and changes to software and packages. Where Continuous Integration and Continuous Delivery normally have been applied to applications developed by an organization, Continuous Adoption applies to the infrastructure and dependencies upon which those applications are deployed.

The rationale and benefits of Continuous Adoption are discussed in greater detail in 4.2.11, “Continuous Adoption” on page 98.

Additionally, the following article is a useful resource to learn more about Continuous Adoption in the context of Integration Middleware:

<https://medium.com/@jvdschot/continuous-adoption-keeping-current-with-accelerating-software-innovations-33233461181a>

7.6.1 What does Continuous Adoption apply to in IBM App Connect?

In the context of IBM App Connect, Continuous Adoption encompasses the regular adoption of the following artifacts:

- ▶ Fixpacks
- ▶ APAR
- ▶ New versions

With on-premises installations, the application of new fix packs, APARs and moving to new versions present significant process challenges. Upgrades to new fix packs, for example, generally are applied to all integration servers on an integration node in one go. This requires a maintenance window to be agreed to across all business teams, so that all Integration servers on the Integration Node can be restarted to use the new binaries. In many organizations windows of this type can be pushed back in favor of releasing new functionality instead. The result is that updates to the infrastructure accumulate until a “must fix” situation is encountered.

Applying fix-packs and APARs, as well as migrating to new product versions, under these circumstances, can be highly pressurized. Keeping the infrastructure up to date is increasingly associated with slowing down deliveries and not with unlocking new functionality or keeping the system in optimum health.

The move to IBM App Connect in a container-based architecture provides an opportunity to turn this paradigm on its head. It does this by adopting new fix-packs, APARs, and product versions through the same CI/CD pipeline that produces an organization’s containerized microservices.

7.6.2 How can Continuous Adoption be implemented with IBM App Connect?

Implementation of Continuous Adoption can be achieved by the combination of ‘base’ Docker images, containing a generic IBM App Connect installation, plus a CI/CD pipeline which layers IBM App Connect integrations on top of this base.

Base Images and Application Images

A base Docker image of IBM App Connect is a container image that contains no integration code. It is built at a specific IBM App Connect fix pack level, with any relevant APARs and PTFs installed into it.

The role of the base image is to provide the foundation on top of which images containing an integration can be built. This is achieved by layering Docker images. Two separate Dockerfiles exist for this purpose: one for the base build and one for the integration image build, which references the image that is created from the first Dockerfile (Table 7-1).

Table 7-1 Example IBM App Connect image architecture

Image type	Explanation
Base image	Product binaries installed at specific fix pack level. Any APAR fixes not included in fix pack also installed. IBM management scripts (container management and health checks)
Application image	Built on top of the base image by using ‘FROM’ statement in Dockerfile. Application code (via BAR file or copy of compiled code to working directory). ^a

a. Run-time configuration is not stored in the Docker image. Instead, it is stored in Kubernetes secrets and loaded into the container at startup.

The IBM App Connect Docker section on **ot4i** (Open Technologies for Integration) provides a template for this type of architecture:

- ▶ <https://github.com/ot4i/ace-docker> - shows how to build a container image without any application being deployed. This can be considered a ‘base’ image.
- ▶ <https://github.com/ot4i/ace-docker/tree/master/sample> - shows how to build an image deploying sample applications on top of the base image.

Figure 7-44 on page 487 shows an example FROM statement inside a Dockerfile for building an application image.

```
Executable File | 12 lines (6 sloc) | 233 Bytes
1 # Build integration image FROM a base image with a specific fix pack
2 FROM mycluster.icp:8500/icp4i/ace-base-11004:latest
3
4
5 USER aceuser
6 RUN env
7
8 COPY ace-docker-master/sample/bars_aceonly /home/aceuser/bars
9
10
11 RUN ace_compile_bars.sh
```

Figure 7-44 Sample IBM App Connect application image Dockerfile

Base images should be produced in your organization each time a new fix pack for IBM App Connect is released. These base images should then be pushed to a Docker registry that is accessible by developer's laptops and the Kubernetes platform

APARs and base images

When an APAR fix needs to be applied before the next fix pack a new base image will need to be created. This new base image, with the APAR fix installed, can be created one of two ways:

- ▶ From the latest base image
- ▶ Created as part of a new base image.

FROM the base image

In this method, an image containing the APAR fix is created FROM the latest base image. This avoids creating a new base image from the ground up, saving storage. Additionally this provides flexibility, as the APAR fix can be applied to different fix pack base images that your organization has available.

For an example of how to build IBM App Connect base images with APAR fixes installed, see the following blog post:

<https://developer.ibm.com/integration/docs/app-connect-enterprise/tutorials/installing-fixpacks-fixes-iib-ace-containers/>

Created as part of a new base image

In this method, the APAR fix is expanded into the tar file that contains the IBM App Connect binaries. A brand-new base image is then created by using this modified binaries file.

To see details on how to create a base image with an APAR fix applied with this method see the IBM App Connect Docker section on the ot4i GitHub:

<https://github.com/ot4i/ace-docker/blob/master/README.md>

Navigate to the section “*Building a container image which contains an IBM Service provided fix for ACE*”. The following is detailed:

“You may have been provided with a fix for IBM App Connect Enterprise by IBM Support. This fix will have a name of the form 11.0.0.X-ACE-LinuxX64-TF12345.tar.gz. In order to apply this fix, follow these steps.”

1. On a local system extract the IBM App Connect archive `tar -xvf ace-11.0.0.5.tar.gz`.
2. Extract the fix package into expanded IBM App Connect installation `tar -xvf /path/to/11.0.0.5-ACE-Linux64-TF12345.tar.gz --directory ace-11.0.0.5`.
3. Tar and compress the resulting IBM App Connect installation `tar -cvf ace-11.0.0.5_with_IT12345.tar ace-11.0.0.5 gzip ace-11.0.0.5_with_IT12345.tar`.
4. Place the resulting `ace-11.0.0.5_with_IT12345.tar.gz` file in the `deps` folder and when building with the `build-arg` to specify the name of the file: `--build-arg ACE_INSTALL=ace-11.0.0.5_with_IT12345.tar.gz`.

Application Development with Continuous Adoption

The use of Docker containers makes it a lot easier for developers to test code locally on their workstation in an environment that is as close as possible to production. See Figure 7-45.

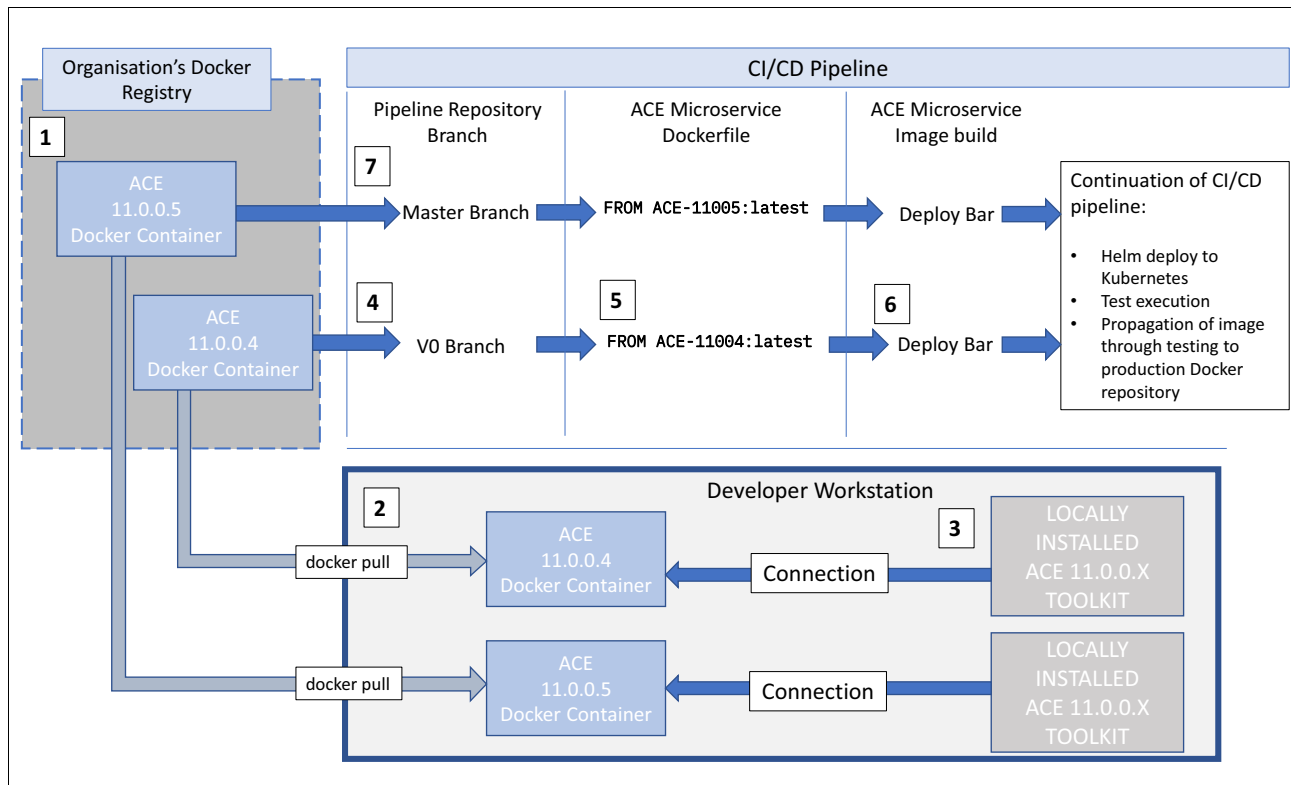


Figure 7-45 Overview of Continuous Adoption with IBM App Connect, for development and CI/CD pipeline

The use of Docker containers makes it a lot easier for developers to test code locally on their workstation in an environment that is as close as possible to production.

Developers can pull down IBM App Connect base images from the organization's Docker repository and run them locally on their workstations. The IBM App Connect Toolkit is still used to develop integrations. However, instead of using a locally defined Integration Server to test their code, developers can connect their Toolkit to the Integration Server in the base image container that was pulled to their workstation.

These base images provide the basis of the microservice images that eventually get deployed to production. So, the application development and local testing is done against an Integration Server that is far closer to production (and all the route-to-live environments) than might otherwise be the case.

CI/CD pipelines for IBM App Connect with Continuous Adoption

In the context of Continuous Adoption, the CI/CD pipeline provides the mechanism for new integration images to be built on top of a range of base images. Then, the set of images is tested and propagated to a production Docker registry (from which the production deployment of new integrations is managed).

Further information on the CI/CD pipeline can be found in section 7.5.1, “Continuous Integration Delivery and Deployment” on page 467.

Figure 7-45 shows the stages of Continuous Adoption for development and deployment with IBM App Connect:

1. Base Docker Images are created by the organization each time a new IBM App Connect fix pack is made available by IBM.
2. Developers issue a Docker pull command to retrieve a copy of the image to their workstation.
3. Developers install the latest version of the IBM App Connect Toolkit onto their workstation and connect these to running instances of the IBM App Connect container. That way, code is developed against the latest fix pack and against an IBM App Connect instance that is as close as possible to production.
4. New branch is created each time a new base image arrives.
5. IBM App Connect integration Docker image builds contain a FROM statement at the start of their Dockerfiles. The statement references the IBM App Connect base image that corresponds to that branch.
6. Continuous Delivery pipeline continues to deploy and test image until it is pushed to production Docker registry.
7. The latest fix pack sits on the master branch. In this example, when fix pack 11.0.0.6 arrives, the 11.0.0.6 will be the new master and 11.0.0.5 will move to a branch called V1.

Further considerations for Continuous Adoption

The following are further considerations for Continuous Adoption:

Microservice versioning and naming

Each organization will likely have its own existing versioning and naming system for applications prior to the implementation of Continuous Adoption. These should be modified, after the following points have been considered:

- ▶ Can you tell which base image each integration image is built upon from its name or tag?
An example image and tag could be: <app-name>:<base image fix pack><timestamp>
ace-db2-app:1100407302019
- ▶ Does your organization consider an integration built against a new base image, but without any integration code changes, to be a minor or major version change?
- ▶ Will you deploy a new Kubernetes service, with a unique name (for example with a minor or major version in the name) for each new microservice?

Below is an example of versioning that is used to show how new base images can be treated in a versioning system, relative to other types of code changes:

- ▶ Version 0.1 - built against ace-base-11004 - initial app
- ▶ Version 0.1 - built against ace-base-11004 - non-functional code update 1
- ▶ Version 0.2 - built against ace-base-11005 - non-functional code update 1
- ▶ Version 0.3 - built against ace-base-11005 - non-functional code update 2
- ▶ Version 1.0 - built against ace-base-11005 - functional code update 1
- ▶ Version 2.0 - built against ace-base-vNext-FP1 - functional code update 1

In this example non-functional code changes (that do not change the api) and updates to the fix pack used in the base image constitute a minor version change. Functional updates to the code (which change the api) and a base image, which contains a new version of IBM App Connect, constitute a major version change.

Depending upon what is decided, you might have to modify the IBM helm charts for IBM App Connect. For example, you might do this to modify the service name, which by default is the same as the application name.

Service rollout

Consideration needs to be given to how new versions of integrations are deployed to test and to production environments. In order to continuously adopt integrations built on new base images, a reliable mechanism will need to be used to migrate requests from one version of the microservice to another. The use of a service mesh like Istio can allow the easy canary deployment of new services and is discussed in section 3.4 with an example shown in section 4.1.3.

7.7 High Availability and Scaling considerations for IBM App Connect in containers

This section discusses considerations regarding High Availability and scaling of IBM App Connect in containers.

7.7.1 Overview

Containerizing your applications brings the key benefit of being able to elastically scale to meet demand. The driving force for moving your applications to containers should not be expectations of improving how the applications run, but more to enable elastic scalability, which containers allow us to achieve.

When the load on your integration server increases due to an increased volume of messages, one of the impacts you would observe is to the CPU utilization and the throughput rate. In such cases, you might want to scale up your integration flows horizontally to cater for the additional load. That way, the CPU utilization is within limits and eventually improves the message throughput rate. Also, when the peak load times are over and the message volumes are less, you would want to scale down the number of integration servers to save on CPU and memory resources.

To address this requirement, Kubernetes provides scaling policies that allow applications to adapt to changing conditions, for example, auto-scaling based on the % CPU utilization for a given deployment. As traffic and workload increases for an application, scaling the application allows it to keep up with user demand. Running multiple instances of an application distributes the traffic to all of them. Additionally, after multiple instances of an application running are available, rolling updates can be performed without downtime.

7.7.2 Scaling

Because we can run IBM App Connect in a Docker container in a Kubernetes environment, we can take advantage of one of the main benefits of the Kubernetes container orchestration engine. Specifically, we can use dynamic scheduling of containers to greater reliability and stability to distributed applications.

As discussed in previous sections, there are three main IBM App Connect container images and Figure 7-46 on page 491 outlines possible topologies that can be enabled through helm charts and run by Kubernetes.

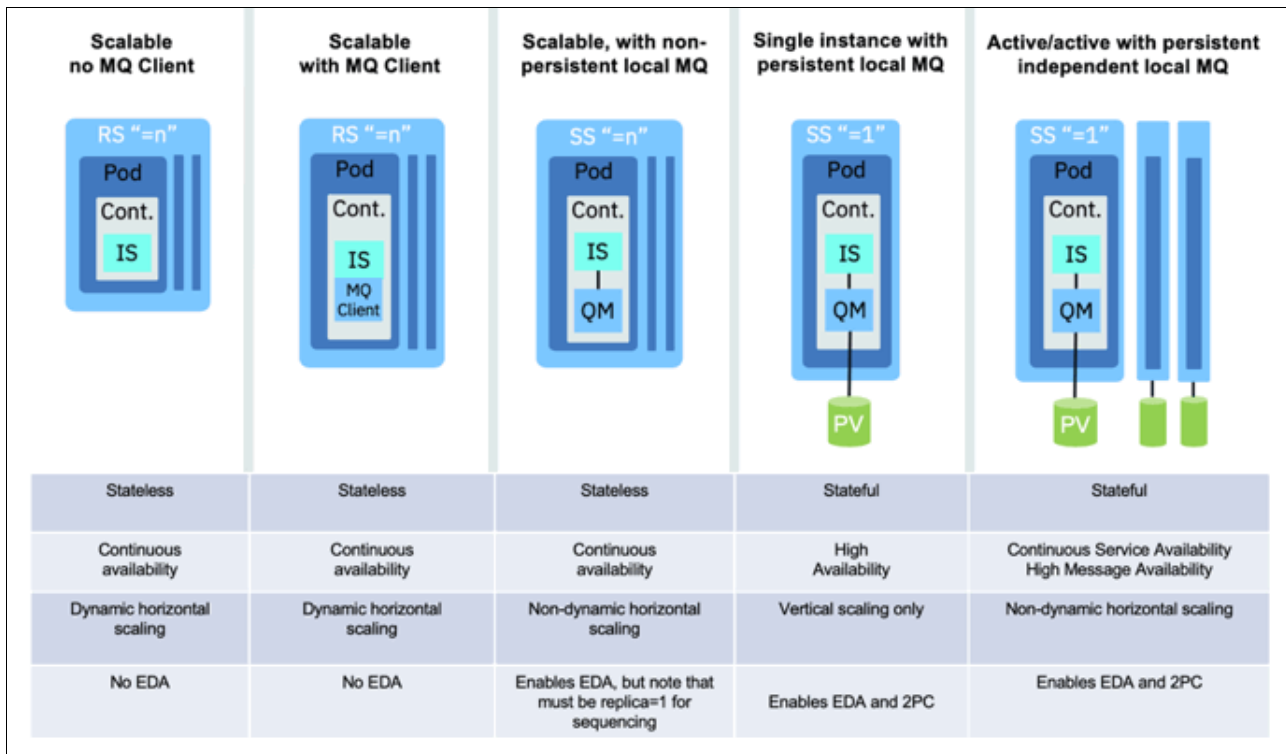


Figure 7-46 Topologies for IBM App Connect Container Images

We can consider two broad topologies for the container images;

- ▶ In the first topology, the first two container images from the left in the preceding figure have only the IBM App Connect Integration Server or include an MQ Client to provide the dynamic scalability for the pods. The Replica Set in this case is shown as 'n'. Replica Set is the mechanism that is built into the Kubernetes environment and is used to replicate the Pods. Using these two images allows completely elastic scaling, based on the number of replicas that we have set.

This topology is completely stateless. There is no need to store state to the disk if you are restarting the containers. For many modern integrations, this approach provides completely sufficient scaling, for example, when exposing REST APIs. The responsibility of storing the state falls to MQ, even when we use MQ Client. So no state is being held in the Integration Server.

- ▶ In the second topology, shown on the right half of the preceding figure, the image contains a local Queue Manager. This means that it be persisting some messaging on local disk.

However, multiple scenarios are possible here:

- ▶ We can still take an approach for elastic scaling with non-persistent local MQ, as shown. No persistent storage is provided if the Node fails and needs to be restarted. Yet this approach can still be sufficient, for example, if we do a 'read' with an Aggregation node. The 'read' can be initiated again after the restart.
- ▶ The next approach represents a typical topology we see today. It could be statically scaled, similar to an HA configuration, where the scaling is vertical by adding more instances.

- ▶ The last topology on the right shows an advanced use case where you might want to scale up your underlying MQ, for example, to increase the throughput. By using Stateful sets, each instance will have its own IP address and we can load-balance between them. However, more manual scaling of the environment will be needed because this scenario requires a local Queue Manager.

As you move towards modernizing your Integrations, you might find that the larger number of your Integrations do not need a local Queue Manager. So you can move them to the horizontal elastically-scaling domain.

7.7.3 High Availability

One of the foundational assumptions about an application's production readiness is that it must meet the requirements for availability. Although designing applications for high availability is out-of-scope for this book, one aspect of availability (the ability to handle dynamic demand or scalability) is particularly relevant to cloud native applications.

With the ability to run IBM App Connect in Docker and Kubernetes environments we can take advantage of one of the main benefits of the Kubernetes container orchestration engine, for example how it brings greater reliability and stability to distributed applications, through the use of dynamic container scheduling.

In a containerized world there are standardized ways to declaratively define an HA topology (Helm Charts). Furthermore, the components that enable the high availability such as load balancers and service registries do not need to be installed or configured because they are a fundamental part of the platform. High-availability policies are built into Kubernetes, and these can be further customized by using standard configuration techniques.

At the application level it is the pods that provide high availability. Kubernetes allows you to run multiple pods (redundancy) and in the event of one of the pods or containers failing, Kubernetes will spin up a replacement pod. This way you can ensure the availability of your services at all times.

When configuring an IBM App Connect helm chart, we can define the values for the configurable parameters of your helm chart. Along with various Integration Server related parameters, you will find an option called '*Replica Count*'. This is the count that represents the number of pods of your application will have running all the time. You can set this value to 1 or more. By defining the number of replicas for your IBM App Connect deployment, Kubernetes platform ensures that the defined number of pods are always running.

7.8 Migrating centralized ESB to IBM App Connect on containers

Important: Since the writing of this IBM Redbooks publication, the IBM Cloud Pak for Integration has embraced Kubernetes Operators (<https://coreos.com/operators/>). This significantly simplifies how components such as an Integration Server are installed and maintained, extending the features provided by Helm. There is more information and an excellent video demonstrating this new capability here:

<https://developer.ibm.com/integration/blog/2020/06/28/ibm-app-connect-operator-1-0-is-now-available/>

It does unfortunately mean that some of the instructions describing the deployment of App Connect Enterprise in containers within this section are now out of date, and will need to be adapted to the use of operators. We may well look to update the book, but in the mean time, refer to the product documentation to find information on the new features.

In this section, we discuss the practical considerations for migrating your centralized ESB topology into more cloud native style deployment by using containers.

7.8.1 Overview

A centralized ESB can simplify consistency and governance of implementation. However, many organizations have more fluid and dynamic requirements to manage. And one part of the organization might be under pressure to implement integration based on cloud-native technologies and agile methods that are being used in other parts of the organization.

A more fundamental change is the continued focus on enabling container-based deployment of the on-premises runtime. Containerization is not mandatory for v11 of IBM App Connect. You can upgrade the runtime only, retaining your existing Integration Node/Integration Server topology, and benefit from many fundamental enhancements in the new version. However, containerization and the associated move to a more cloud-native approach has many advantages. The advantages include simpler deployment build pipeline, isolation/decoupling between integrations, consistency across environments, portability, standardized administration and monitoring, and common capabilities to enable non-functional characteristics such as elastic scaling and high availability and self-healing.

Several other benefits of cloud native deployment are already discussed in 7.1.2, “Adoption path options” on page 434. Another important aspect to consider while migrating to containerized environment is the grouping criteria for integrations. Section 7.2, “Splitting up the ESB: Grouping integrations in a containerized environment” on page 441 describes grouping criteria that you can use to decide which integrations could/should stay together, and which ones must be separated from one another.

Runtime work directory structure changes in IBM App Connect V11

The architecture change in IBM App Connect V11 that introduces independent integration server has brought in several changes to the runtime directory structure. Therefore, it is important to understand how the runtime resources and configuration in IIB maps to IBM App Connect V11. For example, the registry, resource manager configuration, integration node/server configuration, configurable services in IIB are now defined in *server.conf.yaml* file. Policies replace configurable services in IBM App Connect. The deployed artifacts are stored under the *run* directory.

Figure 7-47 on page 494 shows how the runtime work directory structure of IIB maps to IBM App Connect.

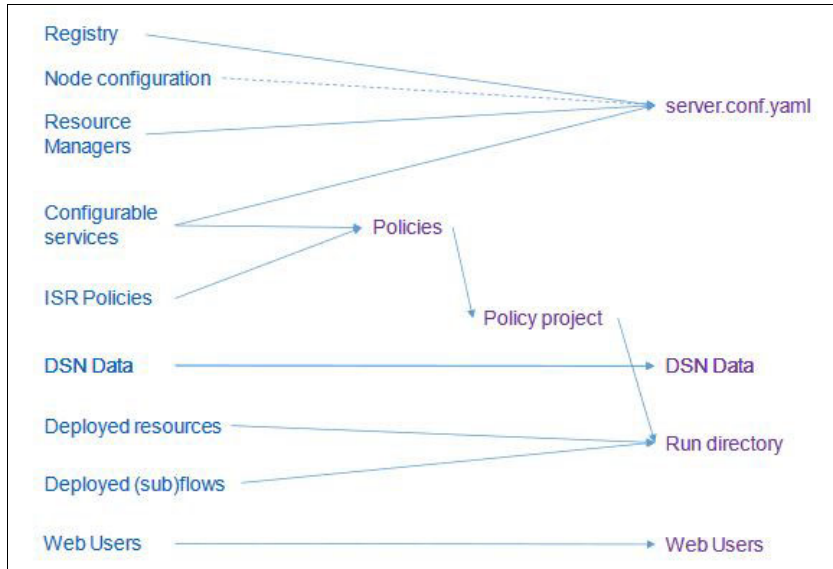


Figure 7-47 Resource mapping between IIB and IBM App Connect

In the subsequent sections of this chapter we primarily focus on the containerization aspects of IBM App Connect and discuss configuration requirements for some of the commonly used endpoints in integration flows.

7.8.2 Considerations for IBM MQ based integrations in containers

The question of whether IBM App Connect requires a local Queue Manager becomes more pertinent as we move to container-based deployment and we explore installation of integrations in a more granular way. There are two very different reasons IBM App Connect makes use of IBM MQ:

- ▶ As an asynchronous messaging provider
- ▶ As a co-coordinator for global (two-phase commit) transactions, aggregation nodes, EDA nodes.

In 7.3, “When does IBM App Connect need a local MQ server?” on page 452 we have already discussed in detail this topic:

When do we need a local Queue Manager and when can we manage without it?

So now, you are in a position to decide what option you want to use to deploy your set of integration flows. Then, you can determine the next steps for deployment. In the following section we describe the configuration and procedure for both the options. Specifically, we describe these examples: 1) Deployment of an integration flow in a container with an associated local queue manager and 2) Integration flow in a container that accesses a remote queue manager by using MQ client connection.

Let’s first look at the different options of Docker images you have with IBM App Connect and MQ.

Docker Image Options for IBM App Connect and MQ

Figure 7-48 shows a chart with various combinations of IBM App Connect Integration server Docker images in context of MQ and the key characteristics for each of the options. This can help in deciding which image type to be used for deploying your integration flows.

	Docker Image Integration Server	Docker Image Integration Server MQ Client	Docker Image Integration Server Local Queue Manager
MQ connectivity	Yes (HTTP API)	Yes (client binding)	Yes (server binding)
1PC for MQ	No	Yes	Yes
EDA nodes	No	No	Yes
2PC	No	No	Yes
Horizontally scalable	Yes	Yes	Yes (with loss of sequencing)
Persistent volume	Not required	Not required	Required (if durability desired)
Start up	Fast	Fast	Slower
Disk space	Smallest	Medium	Largest

Figure 7-48 Docker image options for IBM App Connect and MQ

Connection protocol between IBM App Connect and MQ

Figure 7-49 shows how IBM App Connect and MQ communicate with each other depending on the deployment pattern, for example IBM App Connect and MQ within the same container or IBM App Connect and MQ in different containers.

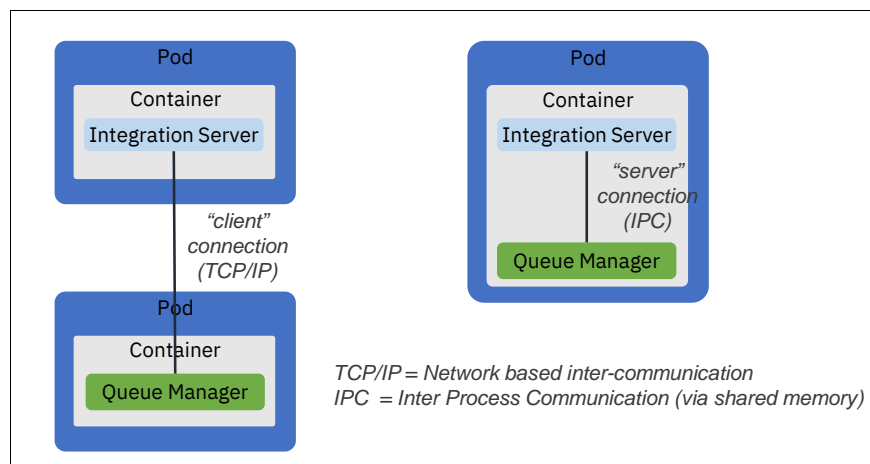


Figure 7-49 IBM App Connect and MQ connection protocol

Now that we have reviewed the key fundamentals of IBM App Connect and MQ connectivity and deployment options, let's go through the step-by-step procedure to deploy an integration flow with an associated local queue manager and another example with integration flow that accesses a remote queue manager by using MQ client connection.

Deploying an Integration Server with Local Queue Manager

If your use case requires a local MQ server for your integration flows, you will require to deploy an IBM App Connect and MQ Server in the same container. In this section, we

illustrate the step-by-step procedure for deploying an IBM App Connect integration server with a local queue manager by using a helm chart.

1. From your IBM Cloud Pak for Integration, log in to IBM App Connect dashboard, and click **Add Server** as shown in Figure 7-50.



Figure 7-50 Deploy new integration server from IBM App Connect dashboard

2. Select an existing BAR file if it was previous added to the dashboard or add a new BAR file as shown in Figure 7-51 on page 496.

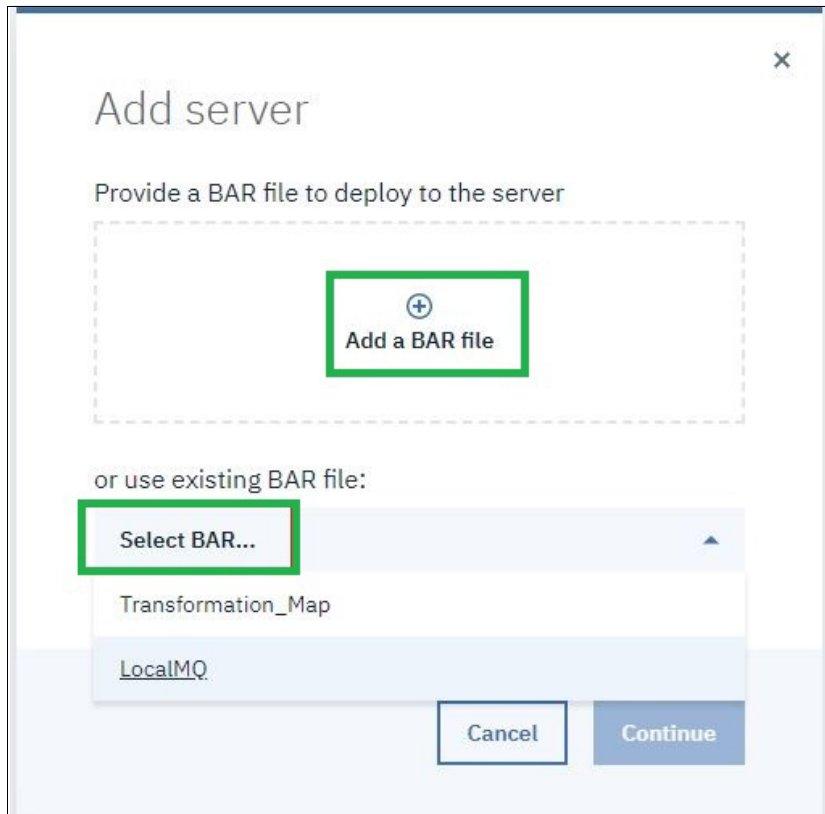


Figure 7-51 Add new BAR file or select existing BAR file for deployment

3. Select the Namespace to which you want to deploy your integration server and copy the Content URL as shown in Figure 7-52 on page 497.

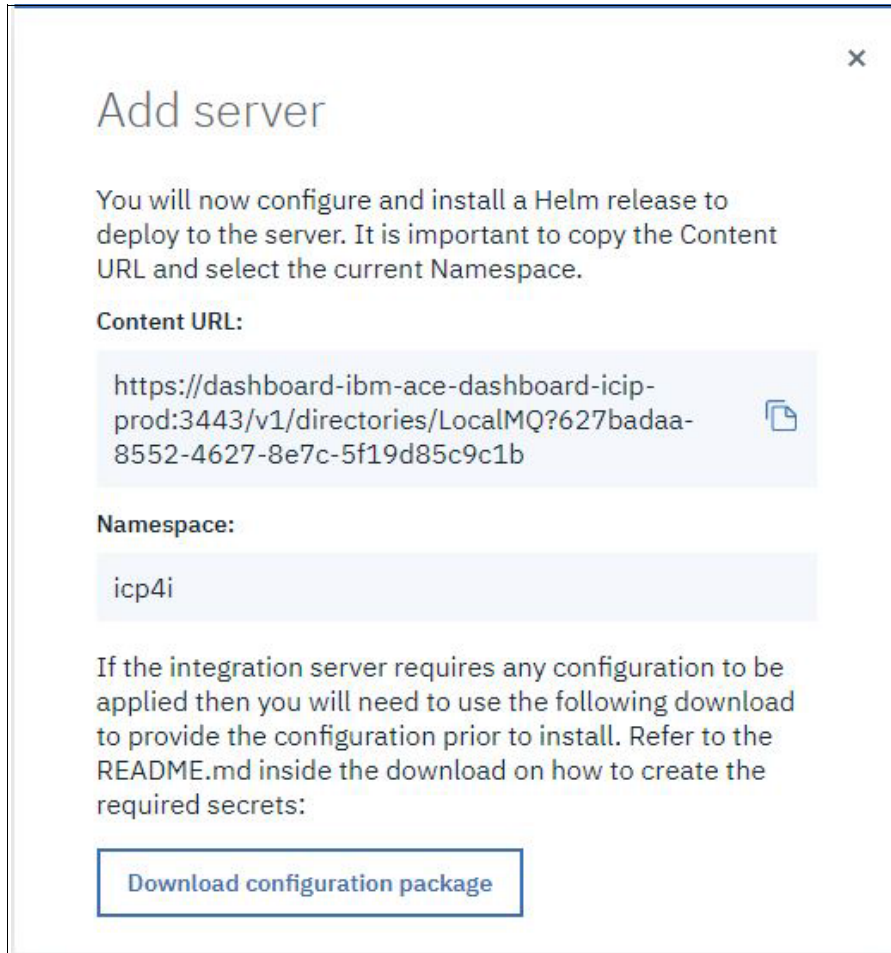


Figure 7-52 Content URL with the location of BAR file

4. Click **Download configuration package**. It downloads a config.tar.gz file.
5. Save the file on your system and unpack it. It provides a set of files as seen in Figure 7-53 on page 497, which you can use for dynamic configuration of your IBM App Connect integration server.

```
README.md
adminPassword.txt
appPassword.txt
config.tar.gz
generateSecrets.sh
keystore-mykey.crt
keystore-mykey.key
keystore-mykey.pass
keystorePassword.txt
mqsc.txt
odbc.ini
policy.xml
policyDescriptor.xml
serverconf.yaml
setdbparms.txt
truststoreCert-mykey.crt
truststorePassword.txt
```

Figure 7-53 List of files in config package

Table 7-2 on page 498 description of files in the config package.

Table 7-2 Description of the files

File	Description
adminPassword.txt	An admin password to set for the MQ queue manager
appPassword.txt	An app password to set for the MQ queue manager
keystore-mykey.crt	A text file that contains a certificate file in PEM format. This will be imported into the keystore file, along with the private key. The file name must be the alias for the certificate in the keystore, with the suffix .crt.
keystore-mykey.key	A text file that contains a private key file in PEM format. This be imported into the keystore file, along with the certificate. The filename must be the alias for the certificate in the keystore, with the suffix .key
keystore-mykey.pass	If the private key is encrypted, then the passphrase can be specified in a file with the file name of alias with the suffix .pass.
keystorePassword.txt	A password to set for Integration Server's keystore.
mjsc.txt	An mjsc file to run against the queue manager
odbc.ini	An odbc.ini file suitable for the Integration Server to use when connecting to a database.
policy.xml	Contains policies to apply.
policyDescriptor.xml	Policy descriptor file.
serverconf.yaml	Server.conf.yaml for integration server. It gets copied to overrides directory of integration server.
setdbparms.txt	Multi-line file that contains <ResourceName> <userid> <password> to pass to mqsisetdbparms command.
truststoreCert-mykey.crt	A text file that contains a certificate file in PEM format. This will be imported into the truststore file as a trusted certificate authority's certificate. The file name must be the alias for the certificate in the key store, with the suffix .crt. The alias must not contain any whitespace characters.
truststorePassword.txt	A password to set for IntegrationServer's truststore.
generateSecrets.sh	A script to generate Kubernetes secrets on the basis of the preceding files.

6. If you want to create some local MQ queues that your integration flows can use for GET or PUT operations, update the mjsc.txt with the list of **mjsc** commands to define the local queues. For example, to create local queues called 'IN' and 'OUT', update the mjsc.txt file as:

```
DEFINE QLOCAL('IN')
DEFINE QLOCAL('OUT')
```

7. There are several other files in the downloaded configuration package such as *serverconf.yaml* and *odbc.ini*, which customer can update, depending on their other configuration requirement.
8. After the required config files are updated on your local filesystem, you need to generate the Kubernetes secret to wrap them in the same namespace as integration server is being deployed to.

You can generate a Kubernetes secret by using the *generateSecrets.sh* script that is provided in the config package. The *generateSecrets.sh* script will take all the files in that directory and turn them into a Kubernetes *secret*. When the Docker container starts and the container is deployed, it loads the secret and extracts the parts. And it puts them into the appropriate paths of integration server work directory, such as placing *server.conf.yaml* in the *overrides* directory of the integration server's workdir.

At the time of writing this book, the **generateSecrets.sh** script in the configuration package did not have a command-line option to specify the namespace in which you want to create the secret. So you might need to modify the *generateSecrets.sh* script to include command-line parameter for namespace. For example, modify the *generateSecrets.sh* script as:

```
TARGET_SECRET_NAMESPACE=$2
```

```
kubectl create secret generic ${TARGET_SECRET_NAME} ${SECRET_ARGS} -n  
$TARGET_SECRET_NAMESPACE
```

Ensure that you have logged in to your kubectl environment and then run the script as `generateSecrets.sh <secret name> <namespace>`

Run following command to confirm that the secret has been successfully created

```
kubectl get secrets -n <namespace>
```

9. Now return to the helm chart configuration. Click **Configure release** as shown in Figure 7-54 on page 499.

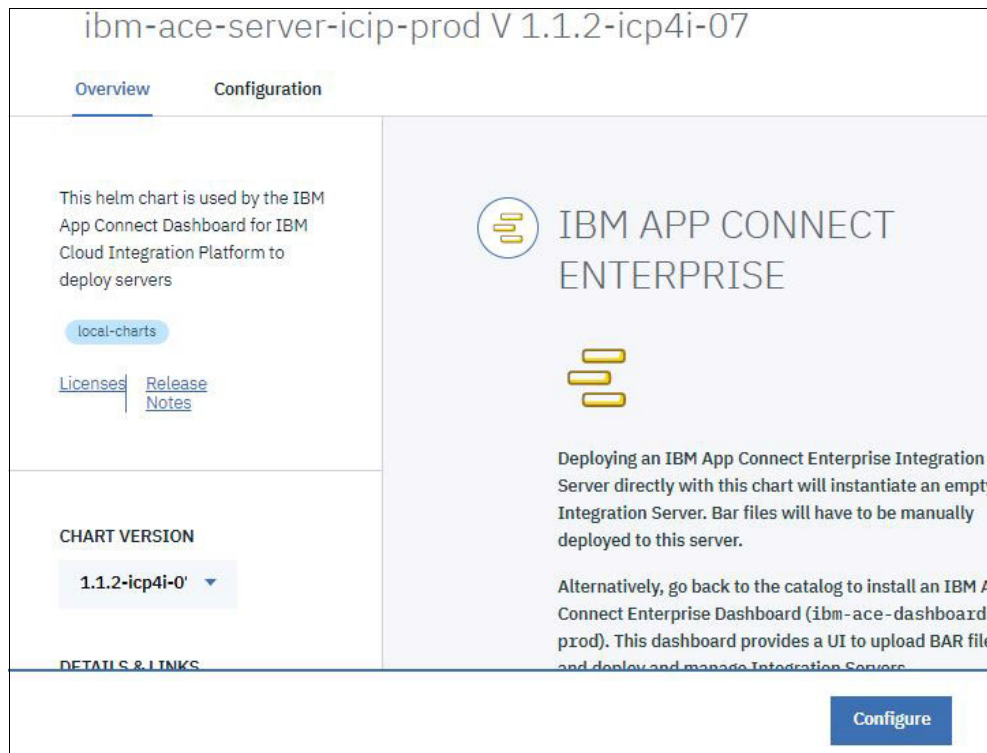


Figure 7-54 Configure the helm chart

10. Provide the name for the helm release and specify target namespace. Accept the license as shown in Figure 7-55.

ibm-ace-server-icip-prod V 1.1.2-icp4i-07

Configuration

Configuration

This helm chart is used by the IBM App Connect Dashboard for IBM Cloud Integration Platform to deploy servers. Edit these parameters for configuration.

Helm release name * ⓘ

Target namespace * ⓘ

License * ⓘ I have read and agreed to the License agreement

Figure 7-55 Helm Release name and Target namespace

11. Paste the Content Server URL that was copied in Step 3. Select the checkbox **Local default Queue Manager**. As shown in Figure 7-56 update the text field corresponding to the secret for integration server configuration with the secret that we had created in Step 8 by using the generateSecrets.sh script.

All parameters
Other configurable, optional, and read-only parameters.

Content Server URL

https://ace-dashboard-ibm-ace-dashboard-dev:3443/v1/directories/LocalMQ?cb1e7e7e-82a3-

Production usage

Local default Queue Manager

Architecture scheduling preference * **File system group ID**

amd64 Enter value

Replica count

1

The name of the secret to create or to use that contains the server configuration

localmqsecrets

Figure 7-56 Content Server URL and Secrets for Integration Server

12. Enter the name for your local queue manager and integration server and click **Install**. See Figure 7-57 on page 502.

Queue manager settings
Settings for the Queue Manager (only applies when running with a Queue Manager)

Queue manager name
mylocalqm

Integration Server
Define configuration for the Integration Server

Integration Server name
ace-mq-server

List of key aliases for the keystore
Enter value

List of certificate aliases for the truststore
Enter value

Name of the default application
Enter value

Cancel **Install**

Figure 7-57 Define the name for Queue manager and Integration Server

13. Check the status in helm release to ensure that the chart has been deployed successfully. See Figure 7-58.

← View All

ace-localmq ● Deployed Launch ▾

UPDATED: July 18, 2019 at 10:17 AM

Details and Upgrades

CHART NAME	CURRENT VERSION	AVAILABLE VERSION	
ibm-ace-server-icip-prod	1.1.2-icp4i-07	1.1.2-icp4i-07	Upgrade
NAMESPACE			Rollback
icp4i	Installed: July 18, 2019 → Release Notes	Released: July 16, 2019 → Release Notes	

Pod

NAME	READY	STATUS	RESTARTS	AGE	
ace-localmq-ib-92e8-0	1/1	Running	0	2m	View Logs

Figure 7-58 Helm Release status

14. You can also check the status of your queue manager from the command line by using kubectl commands.

- a. First get the name of the pod as shown in Figure 7-59.

```
kubectl get pods -n <namespace>
```

```
PS C:\> kubectl get pods -n icp4i
NAME                                READY   STATUS    RESTARTS   AGE
ace-localmq-ib-92e8-0              1/1     Running   0           54m
```

Figure 7-59 listing the deployed pods

- b. Then exec inside the container as shown in Figure 7-60.

```
kubectl exec -it <pod name> -n <namespace> /bin/bash
```

```
PS C:\> kubectl exec -it ace-localmq-ib-92e8-0 -n icp4i /bin/bash
(ACE_11:)root@ace-localmq-ib-92e8-0:/home/aceuser# dspmq
QMNAME(mylocalqm)                                STATUS(Running)
(ACE_11:)root@ace-localmq-ib-92e8-0:/home/aceuser#
```

Figure 7-60 Queue manager status by logging in to the container

15. You can check the status of the integration server from the IBM App Connect dashboard as shown in Figure 7-61 on page 503.

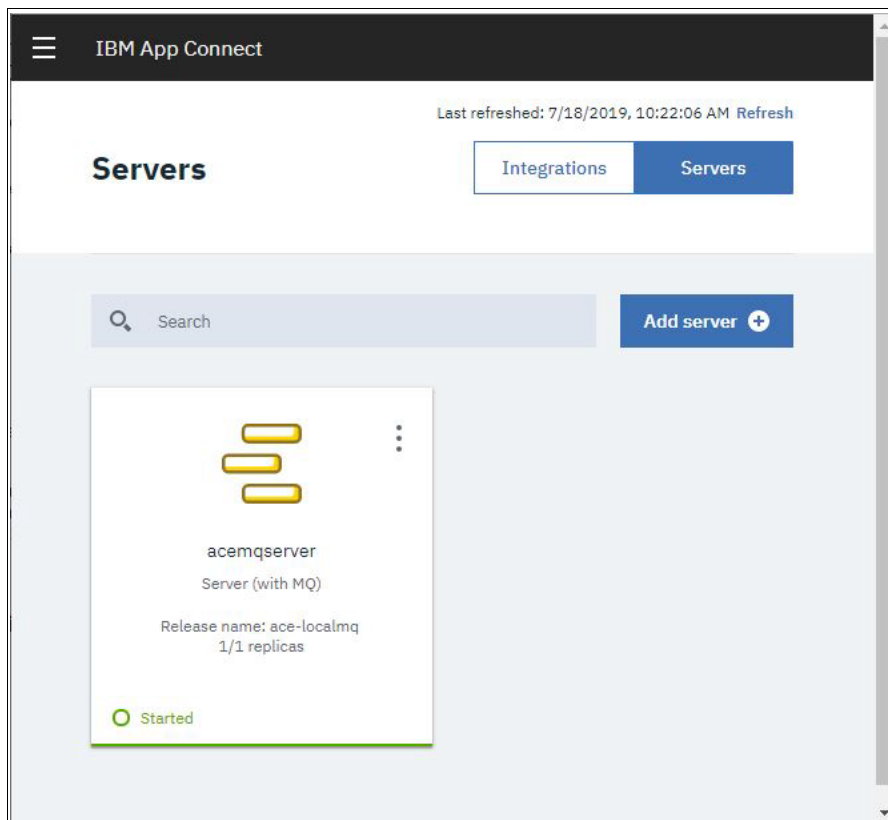


Figure 7-61 IBM App Connect dashboard showing the deployed integration server

Deploying an Integration Server with MQ client connection properties

Greater flexibility was introduced in IBM Integration Bus Version 10.0 in its interactions with IBM MQ; IBM App Connect Version 11.0 maintains this enhanced flexibility.

Consider a scenario where you are migrating from IBM Integration Bus to IBM App Connect v11, specifically for containerized deployment. In this scenario, you might want to configure your IBM App Connect Version 11.0 deployment to take advantage of the flexibility benefits that are described earlier in this chapter.

It might be possible to simplify your system so that your message flows interact with the remote queue managers, which might simplify the topology that you must manage. This simplification requires that you redesign your message flows and your topology. And it goes beyond just a migration of your existing solution. However, you might want to include these activities as part of your migration plans.

You can build a customized IBM App Connect Docker image by combining MQ client libraries. The detailed instructions are available at the following OT4I GitHub repository under section *Build an image with IBM App Connect and MQ Client*: <https://github.com/ot4i/ace-docker>.

You require MQ Client libraries in order to deploy the MQ transport-based integration flows. For the purpose of demonstrating the IBM App Connect and remote MQ client connectivity, we use IBM App Connect-MQServer image. It inherently enables us with all the necessary MQ client libraries to deploy MQtransport based integration flows. However, we connect to a remote MQ server that runs in another container. The instructions described below will remain same if you were to try this with “IBM App Connect and MQ Client” built image.

1. Find the information of your MQ instance that you intend to connect to. Your MQ instance could be running locally on-premises or in the container. For demonstration, we take an MQ instance that runs in our Cloud Pak for Integration platform. Figure 7-62 shows the Platform Navigator of Cloud Pak for Integration with couple of MQ instances.

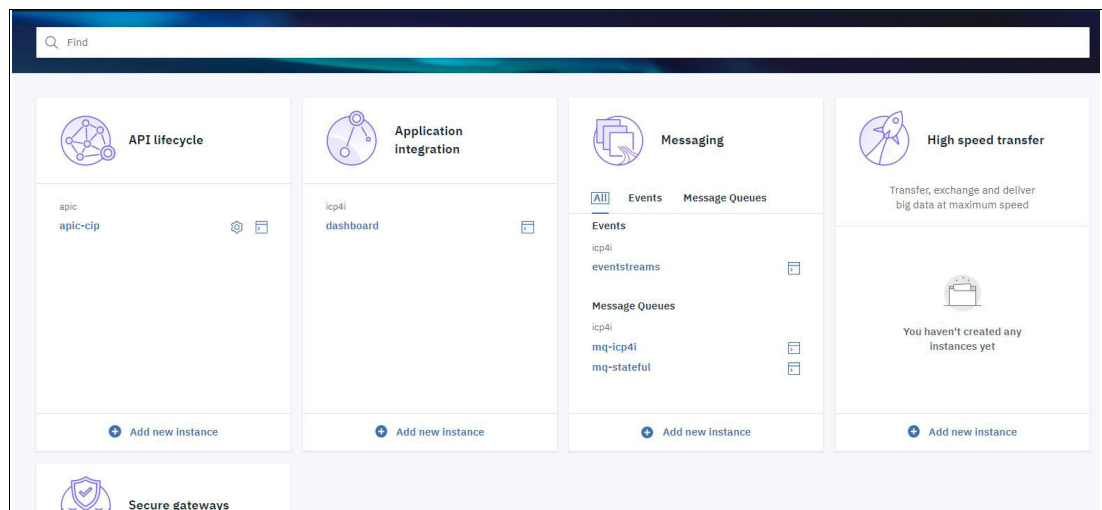


Figure 7-62 IBM Cloud Pak for Integration Platform Navigator

2. Select the MQ instance that you want to connect to from the Messaging tile as shown in Figure 7-63

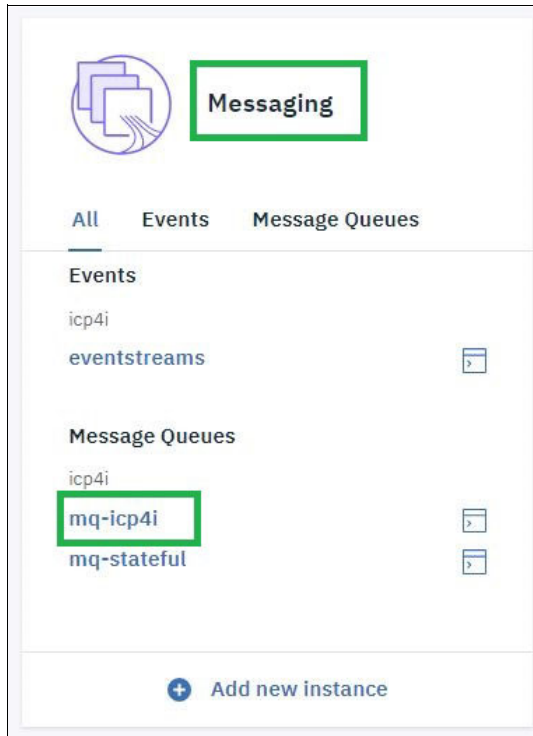


Figure 7-63 MQ instance in platform navigator

3. It will take you to the MQ WebAdmin console as shown in Figure 7-64. To establish the connectivity from your IBM App Connect message flows to the queue manager that runs remotely on another container, you require the connection information of the target queue manager. Click the highlighted options as shown in Figure 7-64 to obtain the Connection Information of your queue manager that runs in the container.

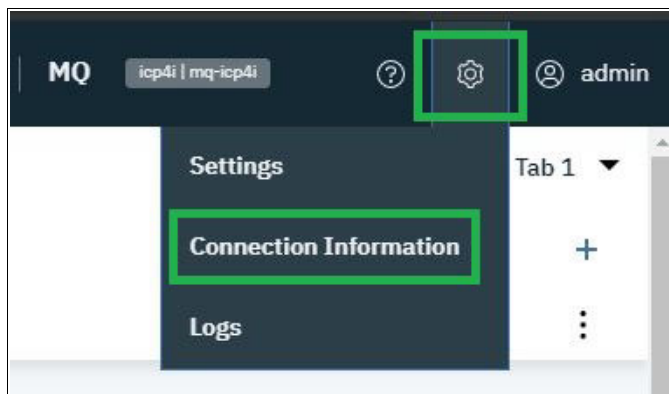


Figure 7-64 MQ Web console options

4. The **Connection Information**, that is the hostname and listener port information of your queue manager will be displayed as shown in Figure 7-65.

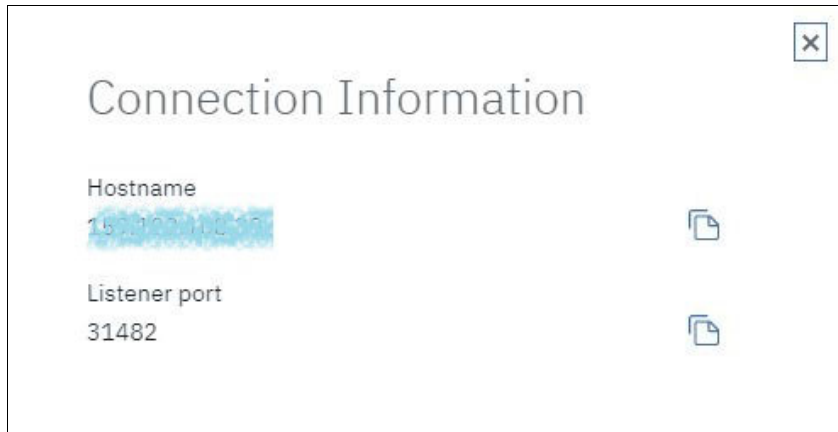


Figure 7-65 Connection information for Queue manager

- Now in the IBM App Connect Toolkit, configure MQ nodes of your integration flow with these hostname and Listener port details. Figure 7-66 on page 506 shows the properties configured on the MQInput Node. If you will be using other MQ based nodes like MQGET, MQPUT, you can follow similar configuration on respective node properties.

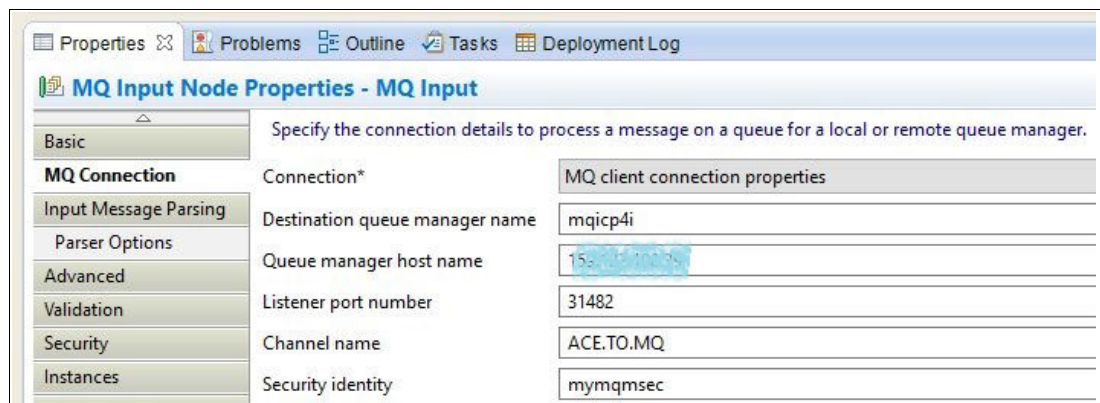


Figure 7-66 Configuring MQ Client Connection through node properties in Toolkit

- Alternatively you can create a MQEndpoint Policy with necessary details and configure the MQInput node to use the policy as shown in Figure 7-67 on page 507.

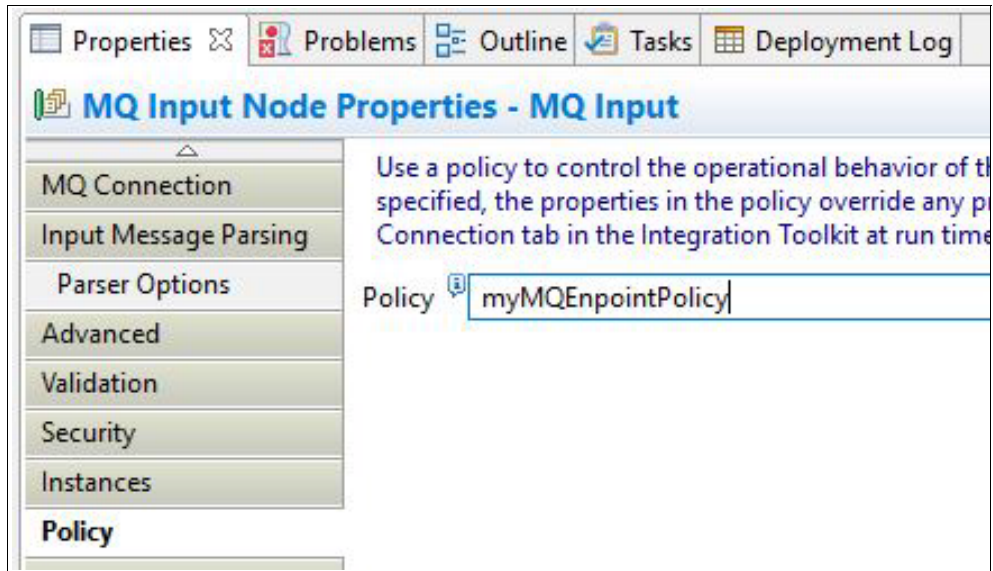


Figure 7-67 Configuring MQ Client connection by using MQEndpoint policy in Toolkit

7. You can create the MQ Endpoint policy in Toolkit or use the template as shown in Example 7-3.

Example 7-3 MQ Endpoint policy template

```
<?xml version="1.0" encoding="UTF-8"?>
<policies>
  <policy policyType="MQEndpoint" policyName="myMQEndpointPolicy"
policyTemplate="MQEndpoint">
    <connection>CLIENT</connection>
    <destinationQueueManagerName>mqicp4i</destinationQueueManagerName>
    <queueManagerHostname>mq-stateful-ibm-mq</queueManagerHostname>
    <listenerPortNumber>1414</listenerPortNumber>
    <channelName>ACE.TO.MQ</channelName>
    <securityIdentity>mymqmsec</securityIdentity>
    <useSSL>>false</useSSL>
    <SSLPeerName></SSLPeerName>
    <SSLCipherSpec></SSLCipherSpec>
  </policy>
</policies>
```

Description of the fields in policy file:

- **policyName** should match with the value mentioned on the Node property in Toolkit
- **Connection** value should be set to CLIENT
- **queueManagerHostname** to be set to host or IP address of the system where the remote queue manager is running.
 - If the queue manager is deployed in the container in the same namespace as IBM App Connect integration server, you can use the Kubernetes service name of the MQ instance. In the preceding example, we use the Kubernetes service name as our MQ server instance is deployed in the container in the same namespace.

```
<queueManagerHostname> mq-stateful-ibm-mq </queueManagerHostname>
```

- If the MQ server instance is deployed in another namespace you can use the fully qualified service name. For example

```
<queueManagerHostname>mq-stateful-ibm-mq.icp4i.svc.cluster.local</queueManagerHostname>
```

Here,

Kubernetes service -> mq-stateful-ibm-mq

namespace -> icp4i

- **listenerPortNumber** is the port number which queue manager listener is listening on
 - **channelName** is the SVRCONN channel between IBM App Connect and MQ server.
 - Set the **securityIdentify** field if you intend to have an authentication process for MQ client connections.
8. Use this policy file shown in preceding example for creating Kubernetes secrets by using the *generateSecrets.sh* script. We will define this secret while deploying the helm chart. If you have defined any *securityIdentity* then update the *setdbparms.txt* file with necessary details so that they too are included in the secrets.
 9. On MQ server side, using the MQ WebAdmin UI, create a SVRCONN channel with the same name as specified on the MQInput node property or in the MQEndpoint policy.

Figure 7-68 shows the SVRCONN channel 'ACE.TO.MQ' and its status.

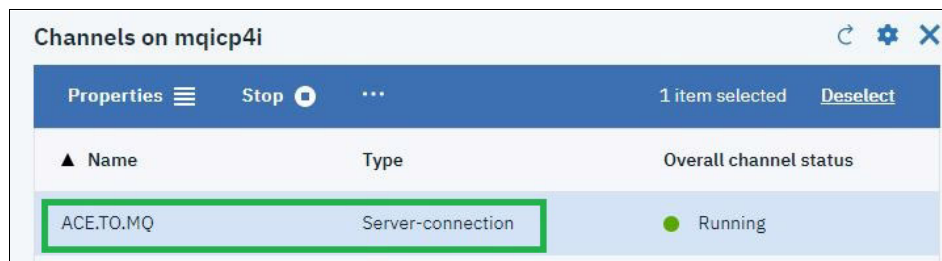


Figure 7-68 MQ SVRCONN channel

10. Now we deploy the integration flow by using the IBM App Connect dashboard. In the helm chart configuration, we select **Local Queue Manager** option. We do not intend to use local queue manager. However, we do this to have the MQ client libraries available inside the IBM App Connect container so that we can support the deployment of MQ-based nodes in IBM App Connect integration server. See Figure 7-69 on page 509.

Parameters

To install this chart, some additional configuration is recommended in Quick Start. If further customization is desired, view All parameters.

> Quick start
Recommended parameters to view and edit.

∨ All parameters
Other configurable, optional, and read-only parameters.

Content Server URL

https://dashboard-ibm-ace-dashboard-icp-prod:3443/v1/directories/RemoteMQ?d1898a7f-414d-418a-802c-312c6eeefccd

Production usage

Local default Queue Manager i

Figure 7-69 Local Queue manager option in IBM App Connect helm chart

11. Update the secrets field of server configuration with the name of the secret created in previous step number 8. See Figure 7-70.

Architecture scheduling preference *	File system group ID
amd64	Enter value
Replica count	The name of the secret to create or to use that contains the server configuration
1	remotemqsecrets
Docker image Specify the image to run. The image is selected based on whether MQ is enabled.	
Docker image for ACE only *	Docker image for ACE with MQ *
mycluster.icp:8500/icp4i/ibm-ace-server-prod-11005	mycluster.icp:8500/icp4i/ibm-ace-mq-server-prod
Configurator docker image *	Image tag *
mycluster.icp:8500/icp4i/ibm-ace-icp-configurator-prod	11.0.0.4a

Figure 7-70 Secrets field for integration server in IBM App Connect helm chart

12. We clear the **Enable Persistence** option as we are going to use MQ in the container only as a client. Therefore, we do not need to persist any messages. See Figure 7-71 on page 510.

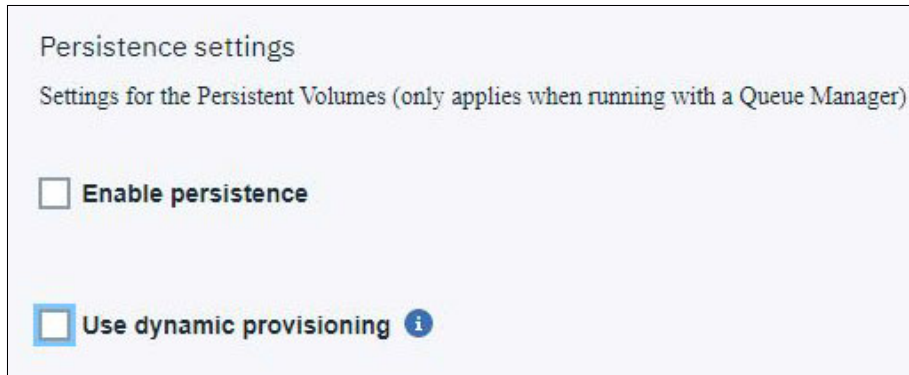


Figure 7-71 Persistence settings

13. After the deployment is done, confirm that the integration flow has successfully connected to remote Queue manager and has started listening for the messages on the input queue. You can review the logs of your integration server in the logging service like Kibana.

Note: You might encounter the `mqrc=2035 MQRC_NOT_AUTHORIZED` error while connecting from IBM App Connect container to MQ server. This happens when your IBM App Connect container is running with a userid that is not present on MQ container or is not a member of mqm group. So you might want to create this IBM App Connect user on MQ container and assign it to mqm group. Alternatively, you could set up some authentication mechanism like LDAP authentication on MQ server and authorize the IBM App Connect user through LDAP services.

14. On successfully connection to remote queue manager, you can observe the *Open input count* on the queue properties in MQ Web Console. A nonzero value indicates that a client application has opened the queue for reading messages. See Figure 7-72 on page 510.

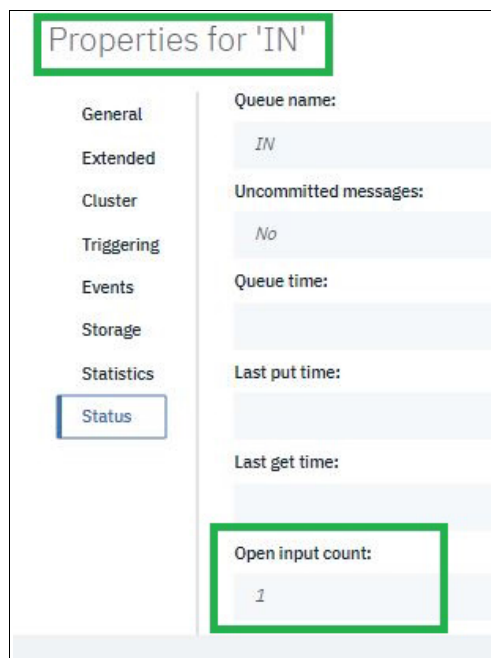


Figure 7-72 Queue properties

15. Now execute the integration flow by putting a message on the input queue.

7.8.3 Considerations for Http/WebServices based integration flows in containers

If your integration flow is designed the SOAPInput or HttpInput node, one of the key configurations to consider while running in containers is the URL on which the service is listening for client requests. The default port on which integration server listens for Http traffic is 7800 and for Https traffic default port is 7843. When Integration server is deployed in a container that runs in Kubernetes, these ports are exposed via NodePort service so that external clients can connect via NodePort service. You can obtain the port-mapping information from **OpenShift Container Platform - Cluster Console** → **Networking** → **Services** → **ServicePort Mapping**.

Figure 7-73 shows a sample configuration of an integration flow with HttpInput node.

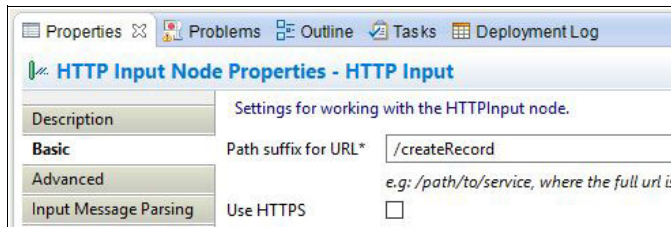


Figure 7-73 HttpInput Node property with service URL

Figure 7-74 on page 511 shows the post-deployment status, where the Kubernetes Nodeport service exposed Http port 7800 as Nodeport 31628 and Https port 7843 as Nodeport 30064.

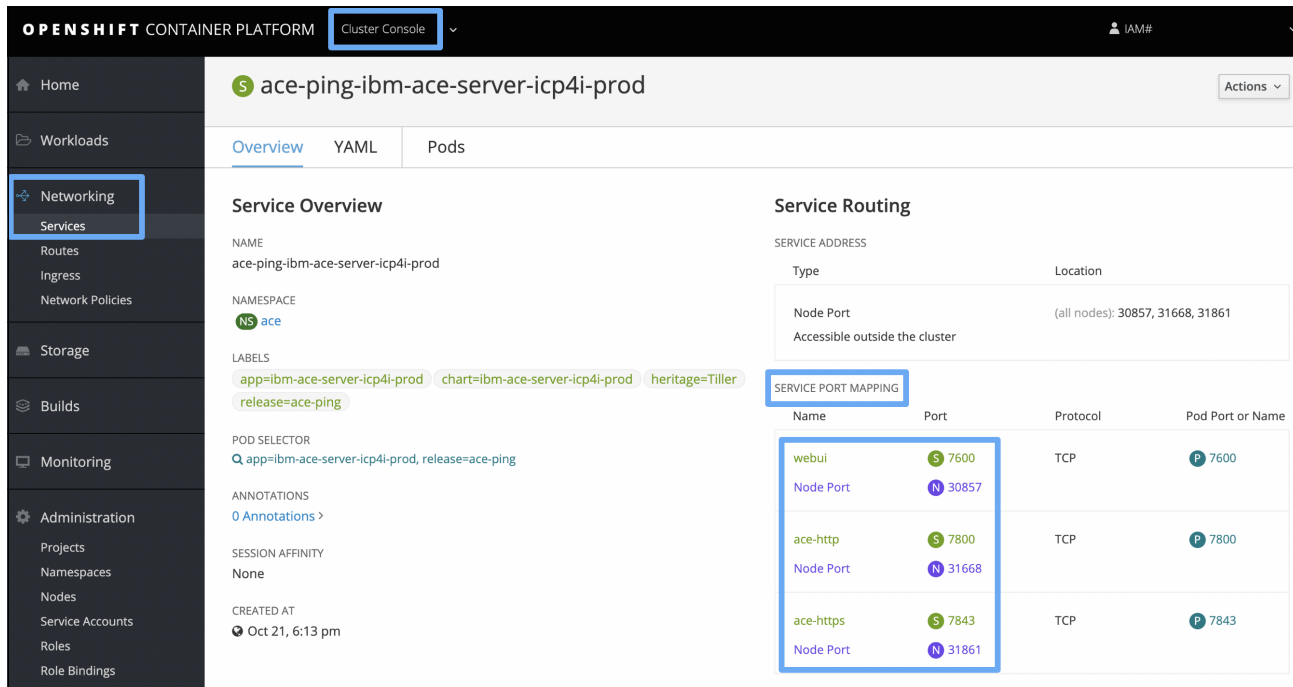


Figure 7-74 Http, https node port mapping

If your client applications are connecting from outside the Kubernetes cluster, they will need to invoke service on URL in following format

`http://<cluster proxy IP>:<Nodeport>/serviceName.`

For example, the preceding URL can be written as follows:

```
http://xxx.xxx.xxx.xxx:31628/createRecord
```

Figure 7-75 shows a Webservice URL configuration for an HttpRequest node of an IBM App Connect flow that runs outside of container platform. The configuration invokes a deployed web service (another IBM App Connect integration flow with HttpInput node) on the container platform.



Figure 7-75 Webservice URL configuration to invoke a service on container platform

Tip: You can also configure a load balancer to route the external client requests to the appropriate nodeports within the cluster. A similar approach is described for MQ in Chapter 9, “Field notes on modernization for messaging ” on page 571.

If your client applications are running within the Kubernetes cluster, you can invoke the service by using the URL format shown in Figure 7-76 on page 512.

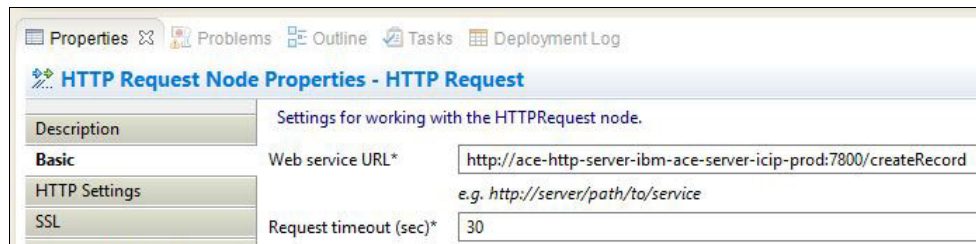


Figure 7-76 Webservice URL using internal service name and port

For example:

- If the service is running in the same namespace as requesting client
`http://ace-http-server-ibm-ace-server-icp-prod:7800/createRecord`
- If service is running in the different namespace as requesting client, then use a fully qualified service name as
`http://ace-http-server-ibm-ace-server-icp-prod.icp4i.svc.cluster.local:7800/createRecord`

7.8.4 Considerations for integrations that interact with databases

If you want to access databases from your deployed message flows in containers, you can connect to a database through a JDBC or ODBC interface. In this section we discuss the configuration requirements for each of these interfaces.

Connecting to database via ODBC interface

IBM App Connect supports the databases that are listed in IBM App Connect system requirements. ODBC drivers for certain database providers (which are based on Wire Protocol) are supplied and installed with IBM App Connect. However, for some database providers like DB2 you must install the client libraries alongside IBM App Connect to enable the connection to the database server. So depending on your target database server, you might need to customize IBM App Connect Docker image to include database client libraries.

- ▶ If you are using the Wire Protocol ODBC driver that ships with the IBM App Connect for databases like Oracle, Sybase, SqlServer no further customization is required for the IBM App Connect Docker image to enable database connectivity. You can proceed to the following section on ODBC configuration steps.
- ▶ The default IBM App Connect Docker image does not include DB2 client libraries. So if your integration flows need to access DB2 database, you need to create a new IBM App Connect Docker image by including DB2 client in it. DB2 client libraries can be obtained from <http://www.ibm.com/support/docview.wss?uid=swg21385217>.

ODBC configuration steps

To access a database via ODBC interface, you require `odbc.ini` file and database access credentials (userid, password) to be set via `mqsisetdbparms`. When you run IBM App Connect integration server in a container in Kubernetes, you need to configure the `odbc.ini` and security credentials via Kubernetes secrets. This is explained through following steps:

1. Create new integration server from IBM App Connect dashboard in the Cloud Pak for Integration.
2. Download the configuration package as shown in Figure 7-77

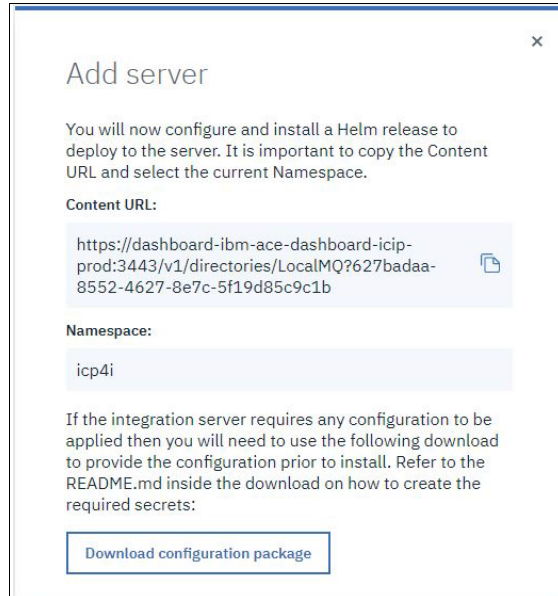


Figure 7-77 Configuration package for integration server

3. Extract the package on your local file system. You will see an `odbc.ini` file in the extracted files.
4. If you are migrating from previous version of IIB or moving from IBM App Connect on-premises to containers, you can copy your existing `odbc.ini` file or create a new DSN stanza in the `odbc.ini`.
5. Ensure that you retain the name of the `odbcini` file as `odbc.ini`.

- The installation path of IBM App Connect in a container in IBM Cloud Pak for Integration platform is `/opt/ibm/ace-11/`. Therefore, modify the `odbc.ini` file as shown in Example 7-4 below to reflect the correct path of the odbc driver in IBM App Connect container. Make the similar change to the `InstallDir` path under mandatory [ODBC] stanza.

Example 7-4 `odbc.ini` file changes for Driver and `InstallDir` path

```
[USERDB]
Driver=/opt/ibm/ace-11/server/ODBC/drivers/lib/UKora95.so

#####
##### Mandatory information stanza #####
#####

[ODBC]
InstallDir=/opt/ibm/ace-11/server/ODBC/drivers
UseCursorLib=0
IANAAppCodePage=4
UNICODE=UTF-8
```

Note: Only a partial snippet of `odbc.ini` file configuration is shown in the preceding example. Refer to the product Knowledge Center for the complete DSN stanza for respective database providers.

- In the Downloaded configuration package, you will find a file named `setdbparms.txt`. Associate your DSN with the user ID and password as shown in Example 7-5.

Example 7-5 Format for credentials in `setdbparms.txt`

```
#<Resource Name> <Userid> <Password>
USERDB user1 passwd1
```

This is equivalent to running `mqsisetdbparms` command for an on-premises integration server.

- Create the Kubernetes secret by using the `generateSecret.sh` script, which is available in the downloaded configuration package. The syntax and considerations for running `generateSecret.sh` is already described in previous section.

```
$ generateSecret.sh myodbcsecret icp4i
```

- In the helm chart, specify this secret in the field for integration server secret configuration as shown in Figure 7-78.



Figure 7-78 Kubernetes secrets for odbc configuration

- After the helm chart is deployed successfully verify the database operations by processing a message through the flow.

Connecting via JDBC interface

You might include a DatabaseRetrieve, DatabaseRoute, JavaCompute, Mapping, or Java user-defined node in a message flow, and interact with a database in that node. In this case,

you must define a JDBC Providers policy to provide the integration server with the information that it needs to complete the connection. Some important considerations about naming the policy are documented in IBM Knowledge Center.

https://www.ibm.com/support/knowledgecenter/en/SSTTDS_11.0.0/com.ibm.etools.mft.doc/ah61310_.htm

To set up a JDBC provider for type 4 connections for message flows running in a container in Kubernetes environment, complete the following steps.

1. Use the *Policy editor* in the IBM App Connect Toolkit to create a *JDBC Providers* policy and choose the template for your chosen database type. The template provides some default values, but you must change some of them to create a viable definition.

Figure 7-79 shows an example of DB2 JDBC provider policy.

Policy

Set the attributes for a Policy

Name: PRODUCTS

Type: JDBC Providers

Template: DB2_91

Property	Value
Name of the database*	PRODUCTS
Type of the database	DB2 Universal Database
Version of the database	10.5
JDBC driver class name*	com.ibm.db2.jcc.DB2Driver
JDBC type 4 data source class name*	com.ibm.db2.jcc.DB2XADataSource
Connection URL format*	jdbc:db2://db2-ibm-db2oltp-db2-dev.default.svc.cluser.local:50000/PRODUCTS
Connection URL format attribute 1	
Connection URL format attribute 2	
Connection URL format attribute 3	
Connection URL format attribute 4	
Connection URL format attribute 5	
Database server name*	db2-ibm-db2oltp-db2-dev.default.svc.cluser.local
Database server port number*	50000
Type 4 driver class JARs URL	
Name of the database schema	useProvidedSchemaNames
Data source description	
Maximum size of connection pool	0
Security identity (DSN)	db2secret
Environment parameters	
Supports XA coordinated transactions	false
Use JAR files that have been deployed in a .bar file	true

Figure 7-79 JDBC provider policy

2. Define *Security identity* field in the jdbc providers policy. This property specifies a unique key to identify a DSN entry, which provides the user ID and password credentials that are required to connect to the database system.

3. Create secrets for IBM App Connect configuration by using the same security identity name in the setdbparms.txt file in the following format:

```
jdbc:<security identity name> username password
```

For example,

```
jdbc::db2secret db2inst1 mypasswd
```

4. Define the *Connection URL* for the target database system. You can specify the Kubernetes service name and port number if your database instance is running in a container in Kubernetes. On the other hand, if your database is running on-prem you can follow the usual convention of target hostname/IP and port combination.

If database is running in a container, you can use the following connection URL format:

- If the database container is running in the same namespace as IBM App Connect server

```
jdbc:db2://<kubernetes service name>:<database port>/DatabaseName
```

For example,

```
jdbc:db2://db2-ibm-db2oltp-db2-dev:50000/PRODUCTS
```

- If database container is running in different namespace to IBM App Connect server then specify the fully qualified service name.

```
jdbc:db2://< service name>.<namespace>.svc.cluster.local:<database port>/DatabaseName
```

For example,

```
jdbc:db2://db2-ibm-db2oltp-db2-dev.default.svc.cluster.local:50000/PRODUCTS
```

- If database is running outside of container platform, then use following format

```
jdbc:db2://HostName_or_IP:portNumber/databaseName
```

5. Generate Kubernetes secrets by using the **generateSecrets.sh** script available in the downloaded configuration package:

```
$generateSecrets.sh myjdbcsecrets icp4i
```

6. Use this secret while deploying the helm chart as shown in Figure 7-80 on page 516.

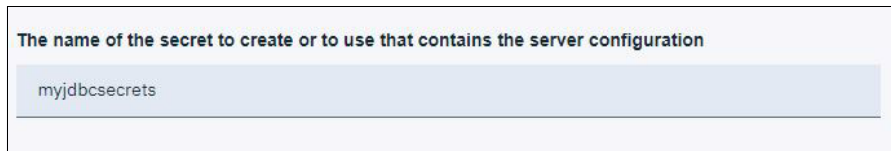


Figure 7-80 jdbc secrets config in helm chart

7. After the deployment is successful, process the message through the integration flow to confirm the database operation.

7.8.5 Considerations for in containers

If your callable flows are both deployed to the same integration server they can communicate with each other as soon as you deploy them. If you are splitting processing between different integration servers or different containers your flows communicate by using a Switch server and connectivity agents. The Switch server is a special type of integration server that routes data. The connectivity agents contain the certificates that your flows require to communicate securely with the Switch server. The connectivity agents must be running in the integration servers where you have deployed your integration flows.

The calling flow includes a `CallableFlowInvoke` or `CallableFlowAsyncInvoke` node, which calls a `CallableInput` node in a second (callable) flow. The calling flow uses a combination of the application name and the endpoint name of the `CallableInput` node to identify which callable flow to call.

Often you want to set up an environment where a Switch server and connectivity agents all run in the container environment. In this case, you will require one container, at a minimum, to act as a Switch server, and two containers act as connectivity agents.

Figure 7-81 on page 517 shows an example arrangement of a Switch server and callable flows. As shown in the figure, Container1 hosts a switch server, Container2 hosts a message flow that has a `Callable Invoke` node and Container3 hosts a message flow that has `Callable`

Input and Callable Reply nodes. The communication between the Callable Invoke flow that runs in Container2 and the CallableInput flow that runs in Container3 is routed via Switch Server that runs in Container1.

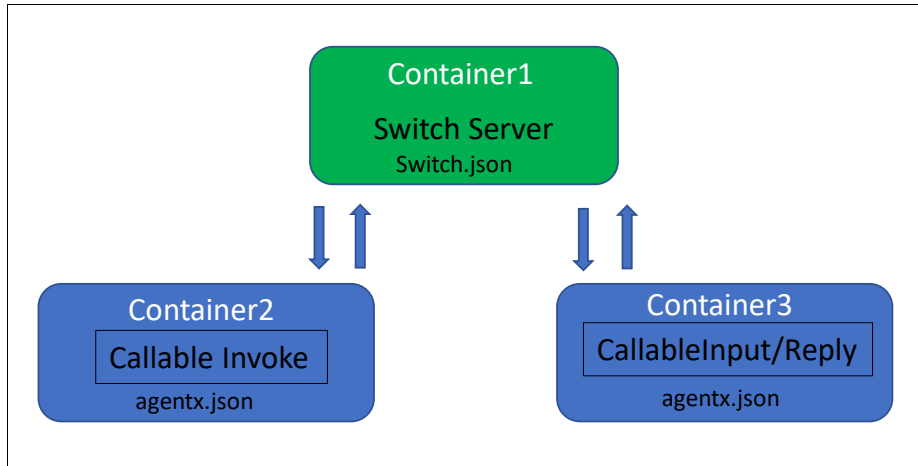


Figure 7-81 Components of in containers

The high-level steps required to set up an environment for callable flows in containers are as follows:

1. Generate the configuration files for Switch Server and connectivity agents
2. Deploy an Integration Server with Switch configuration file
3. Deploy two (or more) Integration Servers with Connectivity agent files

We describe each of these steps in more detail here.

Generate the configuration files for switch server and connectivity agents

You can create the required configuration files for switch and agent components by using the **iibcreateswitchcfg** command

```
iibcreateswitchcfg -hostname <IP address of the server where switch server will run>
```

This command creates two JSON configuration files, *switch.json* and *agentx.json*. The *switch.json* file is used to create the *Switch server*. The *agentx.json* is used to configure the *connectivity agent* for each integration server where your callable flows are deployed.

Deploy integration server in a container with switch configuration file

You start the switch server by deploying the *switch.json* file to an integration server in a container that runs in Kubernetes.

Switch server listens on three ports 9010, 9011, 9012 over which the connectivity agents establish the secure communication. Therefore, when you deploy a Switch server in a container you need to expose these ports externally by using Kubernetes service so that connectivity agents can make connection over these exposed ports. At the time of writing this book, the standard helm chart available with Cloud Pak for Integration did not have an option to expose these additional ports. Therefore, we customize the helm chart to expose these additional ports by using the NodePort service. In future releases of Cloud Integration Pack this might become available as a standard configuration. In that case, the customization of the helm chart will no longer be required.

Refer to 7.4.2, “Upgrading (extending) helm charts” on page 464 which describes a general procedure on how to customize a helm chart.

Following are the high-level steps for deploying an integration server with Switch server component.

1. Download sample IBM App Connect Docker image from <https://github.com/ot4i/ace-docker/>.
2. Download the source of existing helm package from the helm chart overview panel as shown in Figure 7-82 on page 518.

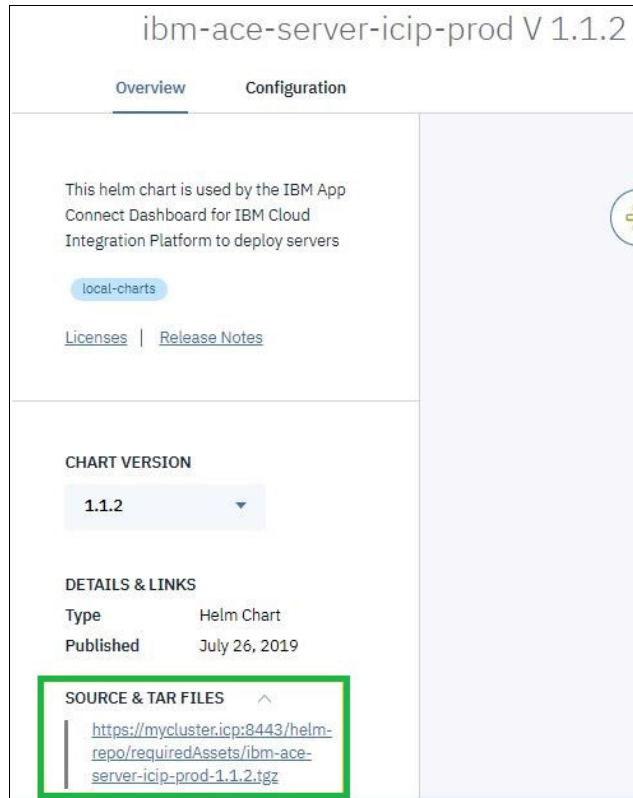


Figure 7-82 Helm chart source files

3. Create a Docker image for IBM App Connect with necessary modifications described below to host the switch server.
 - a. Unpack the Docker package that you downloaded from OT4I in step 1.
 - b. Create a folder with the name 'agent' under sample/initial-config directory.
 - c. Copy the **switch.json** file that you created to the *agent* directory by using the **iibcreateswitchcfg** command.
 - d. Add following lines to the Dockerfile *ubi/Dockerfile.aceonly* such that the switch.json file is packaged in the Docker image during Docker build.

```
RUN mkdir /home/aceuser/initial-config/agent && chown aceuser:aceuser /home/aceuser/initial-config/agent
```

```
COPY *.json /sample/initial-config/agent /home/aceuser/initial-config/agent
```
 - e. Build the Docker image.

```
docker build -t ace11005 --build-arg
ACE_INSTALL=11.0.0-ACE-LINUX64-FP0005.tar.gz --file ubi/Dockerfile.aceonly
.
```

You can also refer to the instruction on <https://github.com/ot4i/ace-docker> to ensure that you get the most current syntax for building the IBM App Connect Docker images.

- f. Log in to Docker registry and push the Docker image to your private registry of your Cloud Pak for Integration platform.

```
docker login mycluster.icp:8500
```

```
docker tag ace11005:latest
mycluster.icp:8500/icp4i/ibm-ace-server-prod:11.0.0.5
```

```
docker push mycluster.icp:8500/icp4i/ibm-ace-server-prod:11.0.0.5
```

4. Create a helm chart by modifying `values.yaml` and `service.yaml` files to include additional fields for the switch ports that we want to expose.
 - a. As shown in Example 7-6 we add the new fields named `switch*` under service type `NodePort` in `values.yaml` file.

Example 7-6 Additional service ports in values.yaml

```
service:
  type: NodePort
  webuiPort: 7600
  serverlistenerPort: 7800
  serverlistenerTLSPort: 7843
  switchport1: 9010
  switchport2: 9011
  switchport3: 9012
```

- b. Add corresponding entries in `service.yaml` as shown in Example 7-7.

Example 7-7 New port fields in service.yaml file

```
- port: {{ .Values.service.switchport1 }}
  targetPort: {{ .Values.service.switchport1 }}
  protocol: TCP
  name: switchport1
- port: {{ .Values.service.switchport2 }}
  targetPort: {{ .Values.service.switchport2 }}
  protocol: TCP
  name: switchport2
- port: {{ .Values.service.switchport3 }}
  targetPort: {{ .Values.service.switchport3 }}
  protocol: TCP
  name: switchport3
```

- c. Update `chart.yaml` file to specify the new version number of the chart. For example:

```
version: 1.1.2-icp4i-07
```
 - d. Package and upload the helm chart. For this you will need to have the Cloud Pak CLI installed as specified in 5.1.4, “Getting access to IBM Cloud Pak for Integration for the exercises” on page 145.

For example:

```
helm package ibm-ace-server-icip-prod
```

```
cloudctl catalog load-chart -a ibm-ace-server-icip-prod-1.1.2-icip4i-07.tgz
```

5. Now deploy the integration server to host Switch server by selecting the new helm chart version from the drop-down list as shown in Figure 7-83.

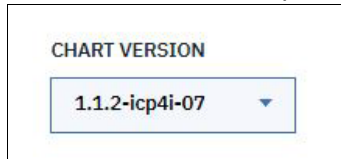


Figure 7-83 Helm chart version selection

6. Select the Docker image that we created in step 3 and select the appropriate image tag as shown in Figure 7-84.

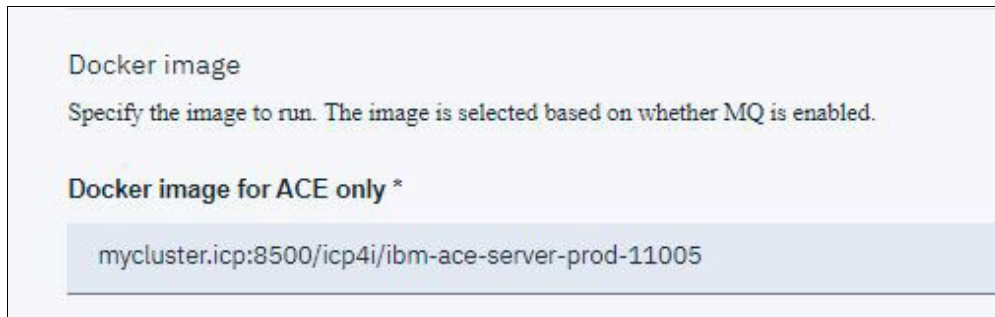


Figure 7-84 Docker image field

Note: In future releases of ICP4I, you might be able to add the agent and switch files dynamically via Kubernetes secrets generated by using the generateSecrets.sh script. In this case, you will not have to build a custom Docker image and helm chart for deploying Switch and Connectivity Agent components.

7. Upon successful deployment of helm release, you will find messages as shown in Example 7-8 in the integration server log (through Kibana logging service). The console output shows the port numbers that the Switch server is listening on.

Example 7-8 Console output for switch server

```
Starting switch with config folder: '/home/aceuser/ace-server/config/iibswitch/switch'  
The Switch server has started listening for agent requests on the back side port '9010'.  
The Switch server has started listening for agent requests on the front side port '9011'.  
The Switch server has started listening for HTTPS administration requests on port '9012'.  
The integration server component 'switch' has been started.  
component started: "switch"
```

8. From the services details of the helm release find out the corresponding externally exposed port numbers as shown in Figure 7-85.

Port	webui 7600/TCP; ace-http 7800/TCP; ace-https 7843/TCP; switch1 9010/TCP; switch2 9011/TCP; switch3 9012/TCP
Node port	webui 32167/TCP ace-http 31368/TCP ace-https 32003/TCP switch1 30804/TCP switch2 30971/TCP switch3 32129/TCP

Figure 7-85 Node port service details for switch agents

Deploy containers with connectivity agent

In order to deploy connectivity agents in container, perform following steps:

1. Update *Switch server URL* in *agentx.json* file to include external port number corresponding to the switch server port 9011. Figure 7-85 shows the port mapping of the services in OCP console and you can observe that the port 9011 maps to 30971.

Example 7-9 shows the snippet of the *agentx.json* file after modification:

Example 7-9 *agentx.json* configuration for switch server URL

```
{
  "name" : "agentx",
  "switch" : {
    "url" : "wss://xxx.xxx.xxx.xxx:30971",
  }
}
```

2. Prepare a new Docker image by following the similar procedure as described in previous section for building the image for Switch server. In this case, instead of *switch.json* file, place this modified *agentx.json* file in *agent* folder and build the Docker image.

Tag the Docker image as:

```
mycluster.icp:8500/icp4i/ibm-ace-server-agent-prod:11.0.0.5
```

3. Push the Docker image to the private registry.
4. Deploy the BAR file that has *CallableInput* node by using the helm chart. Make sure you specify the newly created Docker image in the helm chart Docker image parameter field. See Figure 7-86.

Docker image

Specify the image to run. The image is selected based on whether MQ is enabled.

Docker image for ACE only *

```
mycluster.icp:8500/icp4i/ibm-ace-server-prod-11005
```

Figure 7-86 Docker image customized for Connectivity Agent

5. After connectivity agent makes successful connection with switch server, you will notice following messages in console log

Example 7-10 Console output for agentx component

The connection agent for remote callable flows has established a connection to the Switch server with URL 'wss://xxx.xxx.xxx.xxx:30971'.
The integration server component 'agentx' has been started.
component started: "agentx"

6. Deploy another integration server with the BAR file that has CallableInvoke node by using the same procedure as above.
7. Now invoke your message flow that contains CallableInvoke node and verify that callable flow has returned the expected response.

7.8.6 Considerations for TCP/IP based integrations in containers

You use a TCP/IP node in an IBM App Connect integration flow to create a server or client connection to a raw TCP/IP socket, and to send or receive data over the connection to or from an external application. The TCPIPServerInput node listens on a port and when a client socket connects to the port, the server socket creates a connection for the client. For the TCP client applications to make connection requests on TCP Server, the TCP Server port must be exposed. In a Kubernetes container environment, this is done via Node port service.

Note: At the time of writing this book, the standard helm chart available as part of the Cloud Integration Pack does not have an option to expose user-defined TCP/IP ports externally. Section 7.4.2, “Upgrading (extending) helm charts” on page 464 describes a procedure on how to extend a helm chart. In future releases of Cloud Integration Pack, this might become available as a configuration option such that the customization of the helm chart will no longer be required.

You can also refer to 7.8.3, “Considerations for Http/WebServices based integration flows in containers” on page 511. It gives an example of how to add new fields for TCPIP port via Nodeport service in the IBM App Connect helm chart.

1. You configure the TCPIPServer node properties as shown in Figure 7-87.

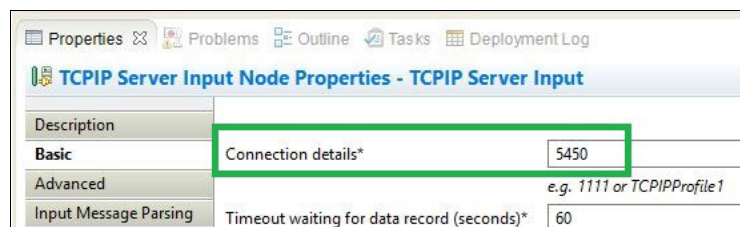


Figure 7-87 TCPIP Server Input Node connection details

After deploying such integration flow in a container that runs in Kubernetes environment, you can obtain its nodeport service details. For example:

- Kubernetes service: *ace-tcpip-server-icp4i-prod*
- Nodeport: *32190/TCP* (externally exposed nodeport for the port 5450 in the container)

Now the TCP Client flows or external TCPIP client applications can connect to this TCPIPServer by using one of the following methods:

2. Use Cluster Proxy IP and NodePort if your TCPIP client is running outside of Kubernetes cluster.

<Cluster proxy IP address>:32190

Figure 7-88 shows the connection details for a TCPIP Client Input of an integration flow that is acting as a client application. Similar connection configuration is applicable for other non-IBM App Connect TCP Client applications.

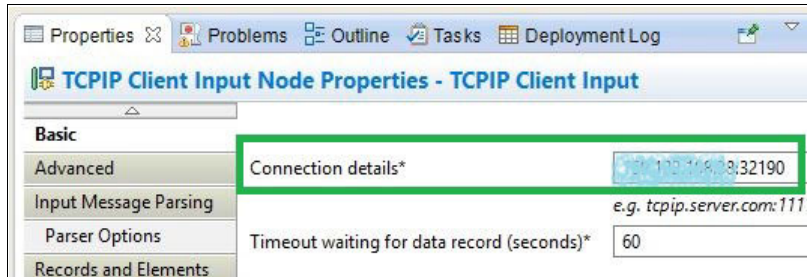


Figure 7-88 TCPIP Client Nodeport connection property

3. Use Service name and Port number if your TCPIP client is running within the Kubernetes cluster.

- **Example 1:** A container that hosts TCPIP Client Applications/integration flows is running in the same namespace as the container that hosts TCP Server integration flow. You can use following connection format:

<service name>:<port number>

Example code for Example 1:

ace-tcpip-server-icp4i-prod:5450

Figure 7-89 shows an example of connection details on TCPIP Client Input node.

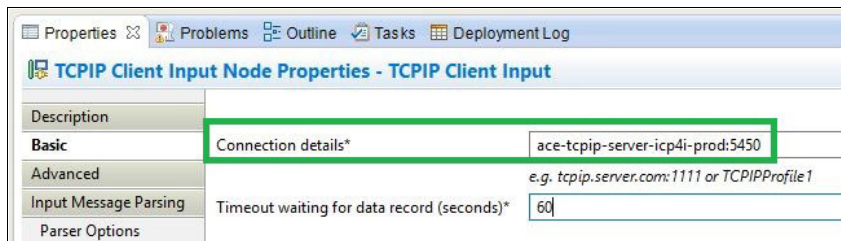


Figure 7-89 TCPIP Client connection using service name and Port number

- **Example 2:** If a container that hosts TCPIP Client Applications/integration flows is running in a different namespace to the container that hosts TCP Server integration flow. You can use fully-qualified service name in the following format:

<service name>.<namespace>.svc.cluster.local:<port number>

Example code for Example 2:

ace-tcpip-server-icp4i-prod.icp4i.svc.cluster.local:5450.

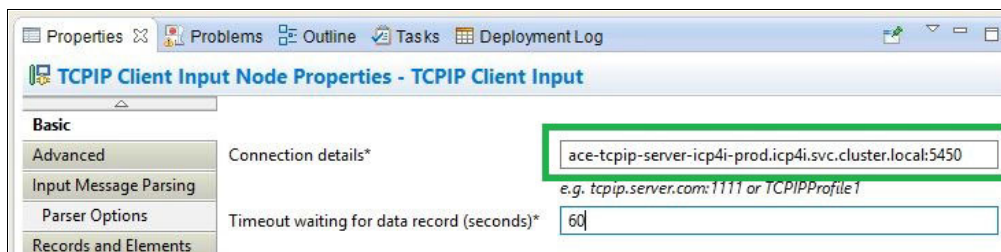


Figure 7-90 Connection details with fully-qualified service name

If you intend to use TCPIP Client policy to configure the connection details, you can use similar connection information as described above in the Hostname and Port fields of the policy file.

7.8.7 Considerations for file-based integration in containers

If your IBM App Connect application (integration flows) is based on File Nodes that consume files from local file system directory and if you intend to move such integration flows to a container, you can use one of the following options:

- ▶ Make the local file system available to the container by using Persistent Volume Claim.
- ▶ Refactor the integration flow to consume the files over remote transfer protocol.

In this section we discuss the considerations for each of these approaches to assist you in making appropriate choice that suit your business needs.

Using Persistent Volume Claim to access local file system

Kubernetes Persistent Volume (PV) and Persistent Volume Claim (PVC) provides a mechanism to share a directory from your local system to Kubernetes container. For more information on PVs and PVCs, refer to 3.2, “Capability perspective: Application integration” on page 54. In your Container Platform you can create Persistent Volumes of desired size, type and storage class. The IBM App Connect helm chart can be customized to provide an option for a Persistent Volume (for non-MQ use cases). At deployment time, the Helm release will then create a Persistent Volume Claim to bind an available Persistent Volume in the container platform to the IBM App Connect Deployment.

At the time of writing this book, the standard IBM App Connect helm charts available as part of the Cloud Integration Pack do not provide an option to bind to a PVC except for the use with Local queue manager. However, it is possible to extend the existing IBM App Connect helm chart to include an option to attach to a PVC. Figure 7-91 on page 525 shows a possible arrangement of the components for the File-based integrations in containers for a NFSv4 compliant file system storage.

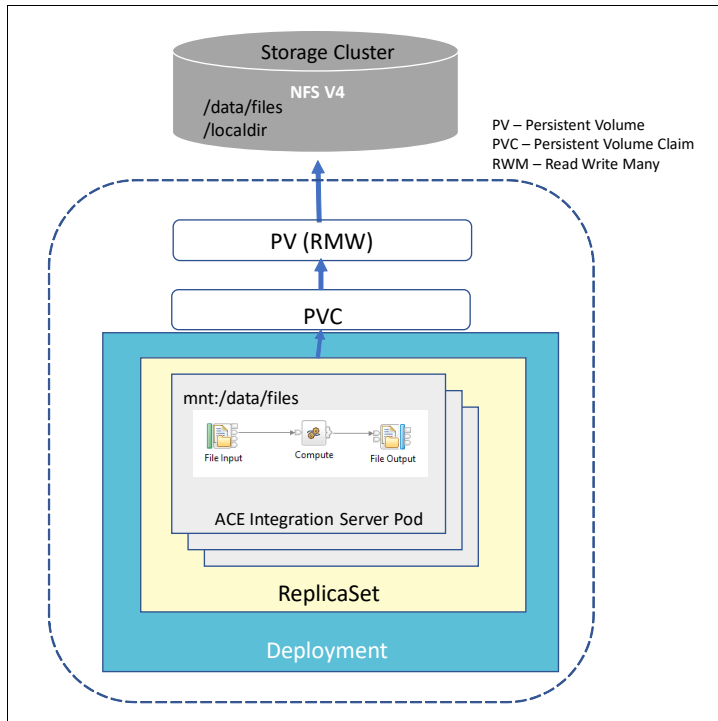


Figure 7-91 Accessing external files from IBM App Connect container

Notice that we have shown the configuration as a *Deployment* rather than a *Stateful set*. *Deployment* creates a *ReplicaSet* that then creates one or more *Pods*. In this configuration only one *PVC* will be created that all the pods will share. The *Persistent Volume* should support *Read-Write-Many* to allow *Integration Flows* that runs in multiple *Pods* to access the same folder. Such feature is typically provided by file system like *NFS*. Following are the guidelines for deploying your integration flows by using the *Deployment* and *PVC* option.

- ▶ Ensure that your host system has a file system that supports *Read-Write-Many* (for example *NFSv4*).
- ▶ Mount the shared path on each node in the cluster. Later, when you scale your flows, *Deployment* and *Pods* could be scheduled on any node within your cluster. Therefore, you must have that mount point available for the *Pods* to access the files. As an example, assume that the shared path is */data/files*.

You use this shared path in your *File Nodes* properties. For example, if you want to use *FileInput* node to read the files, define */data/files* path as *Input Directory* on the *FileInput* Node properties. Ensure that files arrive on this path in our local file system.

- ▶ Create a *Persistent Volume (PV)* in *Kubernetes* cluster by using *NFS* and *Read-Write-Many* option
- ▶ In your customized helm chart, select the option to create a *Persistent Volume Claim (PVC)* which, at deployment time, will bind an available persistent volume to the *IBM App Connect* deployment.

Upon successful deployment of helm release, the shared path on the local file system is now accessible to the *File Nodes (FileInput, FileOutput, FileRead)* that runs in a container.

Refactor the integration flow to consume the files over remote transfer protocol

In addition to reading or writing a file on local file system, the file nodes in IBM App Connect can read a file or write to on a remote directory by using FTP, FTPS, or SFTP protocol. In the previous section we discussed how you can migrate/enable your file-based integration flows. Those flows were designed to read/write files from/to local filesystem to run in a container environment by using Persistent Volume Claim (PVC). This approach requires you to provision a storage class that supports Read-Write-Many mode of operation and bind a deployment to a Persistent Volume. If you cannot adopt this option, then you can use the following alternative approach: Access the files over remote transfer protocol as depicted in Figure 7-92 on page 526.

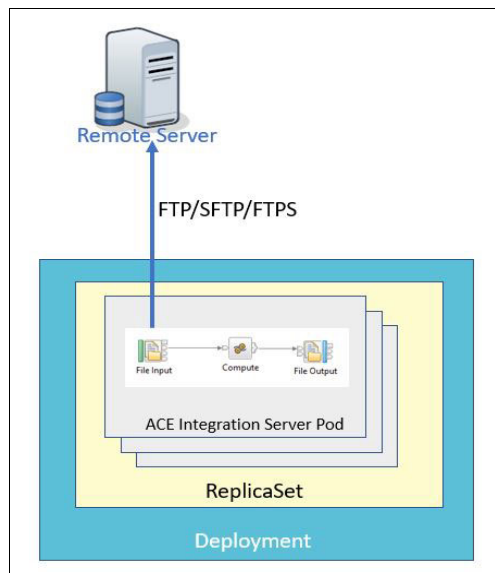


Figure 7-92 File Nodes Remote Connection configuration

This approach might require you to refactor your integration flows to use Remote Transfer option on File Nodes and ensure that the remote transfer services are available on target server.

Following are the configuration steps to use File nodes in Remote Transfer mode:

- ▶ Select *Remote Transfer* option on FileInput Node as shown in Figure 7-93
- ▶ Select the *Transfer Protocol* from the options in drop-down list - FTP, SFTP, FTPS
- ▶ Enter the server and port number where files are to be written or read from.
- ▶ Provide the security identity that you created for accessing remote server. This security identity must be defined in `setdbparms.txt` while creating the Kubernetes secret key. In `setdbparms.txt` the format of the entry will be as given below. For example, for an SFTP mode: `sftp::sftpsecret userid password`
- ▶ Directory on the remote server where the file is accessed.

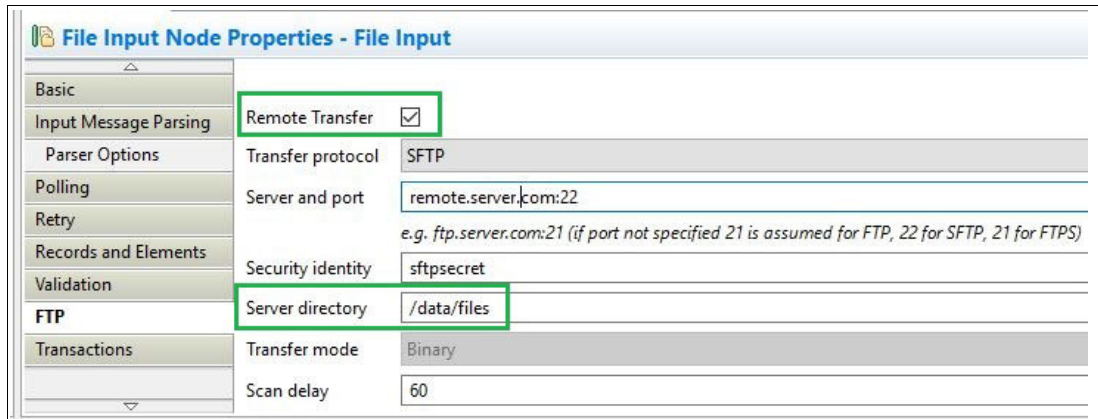


Figure 7-93 FileInput Node remote transfer configuration

- In the remote transfer mode, Fileinput node first moves the files from the remote server to the local directory and then processes it. Therefore, as shown in Figure 7-94 on page 527 you need to provide a *local input directory* to hold the file while it is being processed. This input directory must be on the local file system where the Integration Server is running. In the container environment the local directory for an Integration Server will be the directory on a file system inside the container.

This directory should have read/write access for the IBM App Connect user. In the example shown in Figure 7-94, /tmp is defined as the Input Directory. With this configuration, the file on the remote server will first get transferred to the /tmp directory on the local file system inside the container. You can also specify the file name or pattern to filter out the files that you want to process.

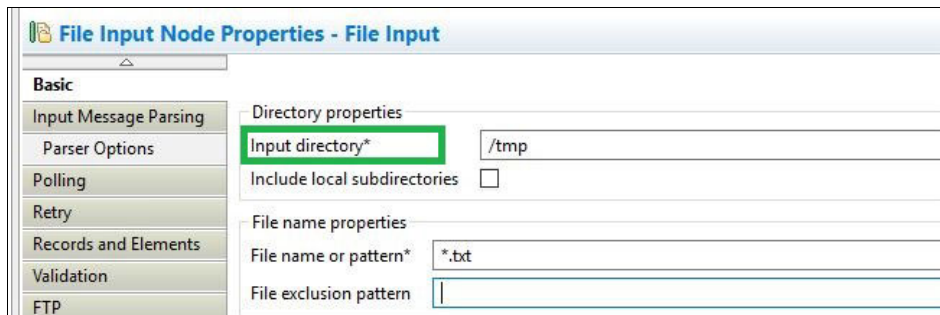


Figure 7-94 FileInput node Basic tab properties

- After you have modified your integration flow as described above, create the BAR file, Kubernetes secret and proceed with the deployment via helm chart.

Important: In Remote Transfer mode, the file is deleted from the remote server after it is transferred to the local directory by the FileInput node for processing. The FileInput node creates an *mqsitransitin* subdirectory in the input directory. The *mqsitransitin* subdirectory holds and locks the input files while they are being processed. If an integration server crashes while processing the file, the partially processed file can be recovered from the *mqsitransitin* subdirectory of the local input directory and can be submitted for reprocessing.

In the Kubernetes environment, if the IBM App Connect Integration Server Pod crashes, a new replacement Pod is spawned by Kubernetes from the base Docker image. The data or state is lost when the container/Pod dies because the file system inside the container is temporary. Therefore, you might lose the partially processed file because it is held in the local directory inside the container in the following scenario:

You are processing a file inside container in Remote Transfer mode, and the container /Pod crashes for some reason.

Therefore, when you use the Remote Transfer option for processing files, you need to ensure that you have a backup copy of the files on the remote server. That way, the files can be resubmitted for processing in the event of failure due to pod or integration server crash.

For file-based integration, Remote Transfer option is recommended instead of consuming the files via file system mounts that use persistent volume claim (PVC).

7.8.8 Considerations for integrating IBM App Connect with IBM Event Streams

IBM Event Streams is a scalable, distributed, high-throughput message bus, which supports a number of client protocols including Kafka. You can use the KafkaProducer and KafkaConsumer nodes in IBM App Connect to receive messages from and send messages to IBM Event Streams.

The following steps provide detail on how to configure a connection from IBM App Connect to IBM Event Streams that runs within the Cloud Pak for Integration platform. Similar steps can be used for connecting to an external event streams instance.

Extract event stream connection information

1. To obtain the connection information of Event Stream instance in the IBM Cloud Pak for Integration cluster, click **Connect to this cluster** option as shown in Figure 7-95.

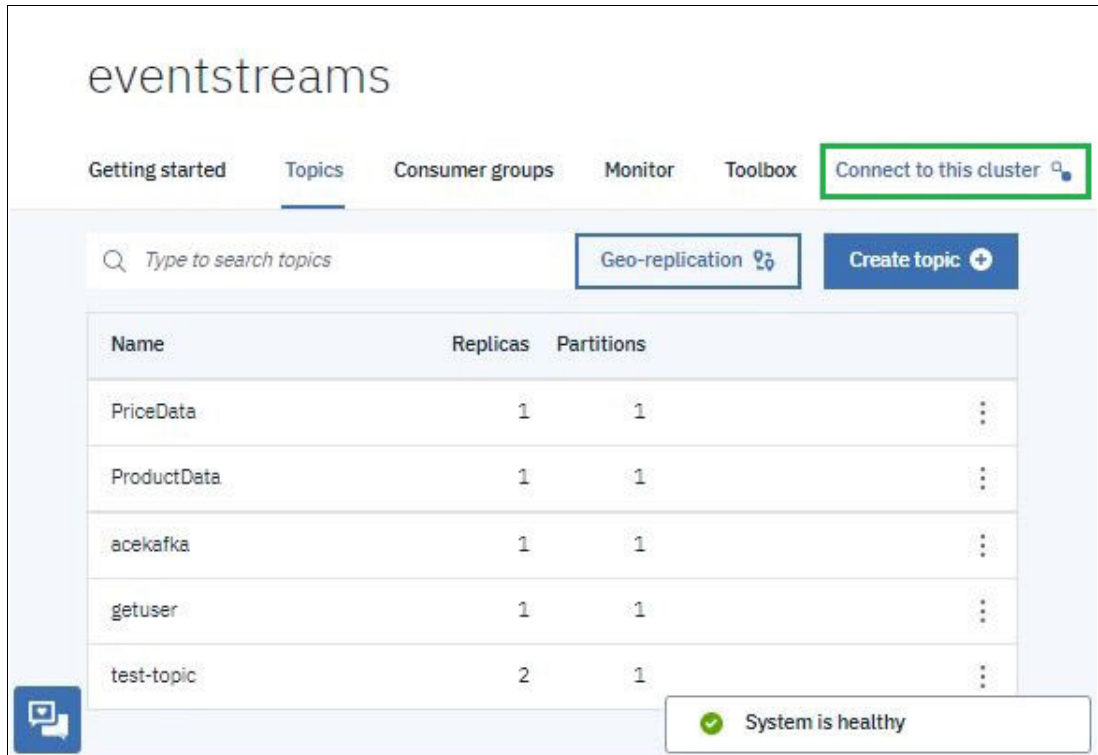


Figure 7-95 Event Stream cluster connection and topics

2. Copy and download the *Bootstrap server* information as shown in Figure 7-96 on page 530. You use this information while you configure the *KafkaProducer* and *KafkaConsumer* nodes in IBM App Connect integration flow. In order for the Kafka clients — such as IBM App Connect Integration flow — to connect to Event Stream instance with the SASL_SSL security protocol, you must have the certificates. Download the *PEM certificates* by clicking the option as shown in Figure 7-96 on page 530.

Cluster connection

[Connect a client](#)
[Sample code](#)
[Geo-replication](#)

To connect an application or tool to this cluster, you will need the address of a bootstrap server, a certificate and an API key.

Bootstrap server

Your application or tool will make its initial connection to the cluster using the bootstrap server.

27.6c.7a9f.ip4.static.sl-reverse.com:30418

Certificates

A certificate is required by your Kafka clients to connect securely to this cluster.

Java truststore
Use this for a Java client

Truststore password: password

PEM certificate
Use this for anything else

API key

To connect securely to Event Streams, your application or tool needs an API key with permission to access the cluster and resources such as topics.

● ○ ○ ○

Name your application

Provide a name for your application

Connecting your tool or application will generate a service ID using your application name, a service policy and an API key in IBM Cloud Private. Additional API keys can be generated in the IBM Cloud Private Cluster Management Console or CLI.

[IBM Cloud Private Cluster Management Console](#)

Figure 7-96 Bootstrap server and Certificate information

3. Make note of the Truststore password as this will be required while you configure the IBM App Connect secrets.
4. Create API Key by using the options as shown in Figure 7-97 on page 531.

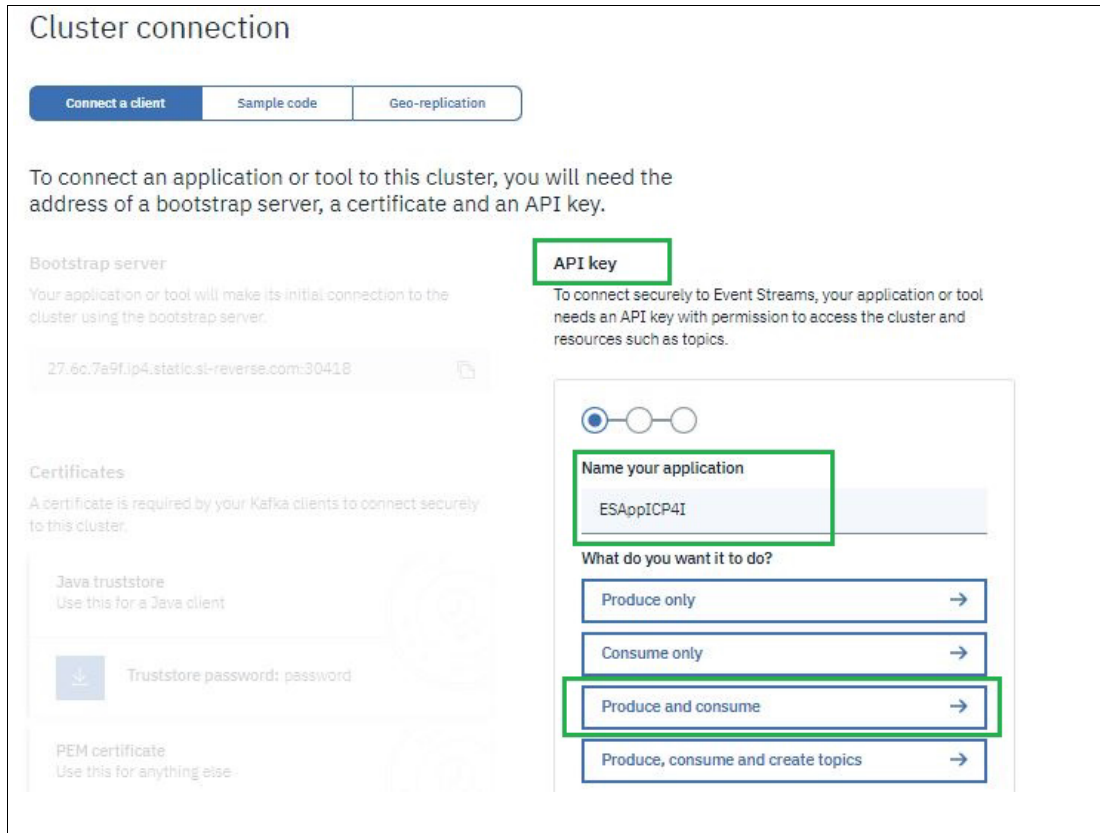


Figure 7-97 API Key creation options

5. Click **Generate API Key**.

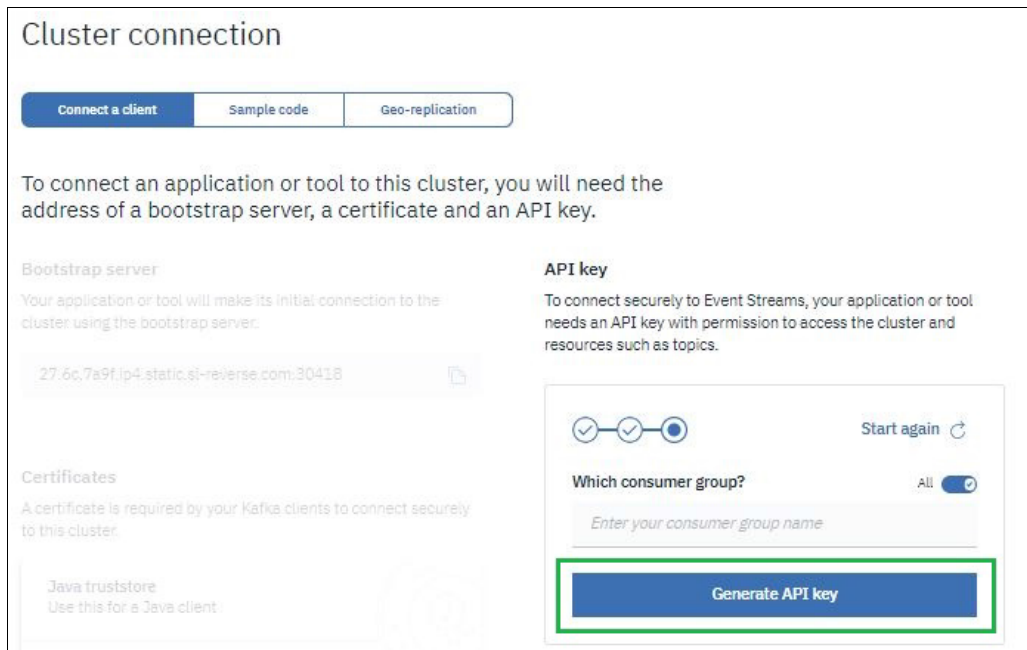


Figure 7-98 Generate API Key

6. Copy or download the API Key.

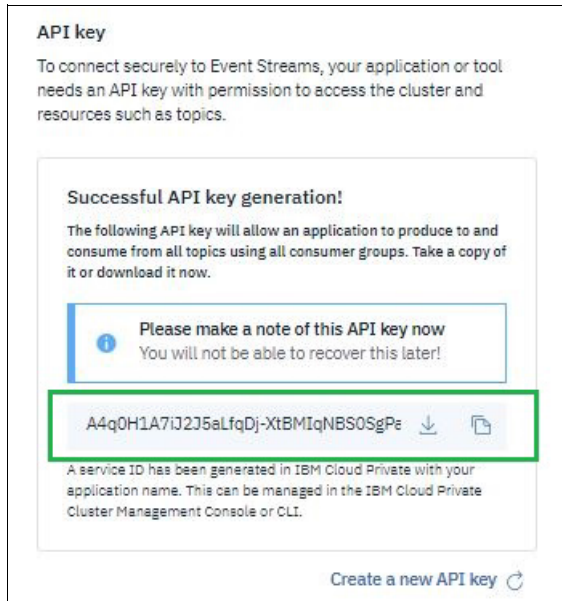


Figure 7-99 API Key for Accessing Event Stream

Configuration for IBM App Connect integration flow

Follow these steps to configure IBM App Connect integration flow:

1. In IBM App Connect Integration Toolkit, configure KafkaConsumer Node with Topic name and Bootstrap server as shown in Figure 7-100.

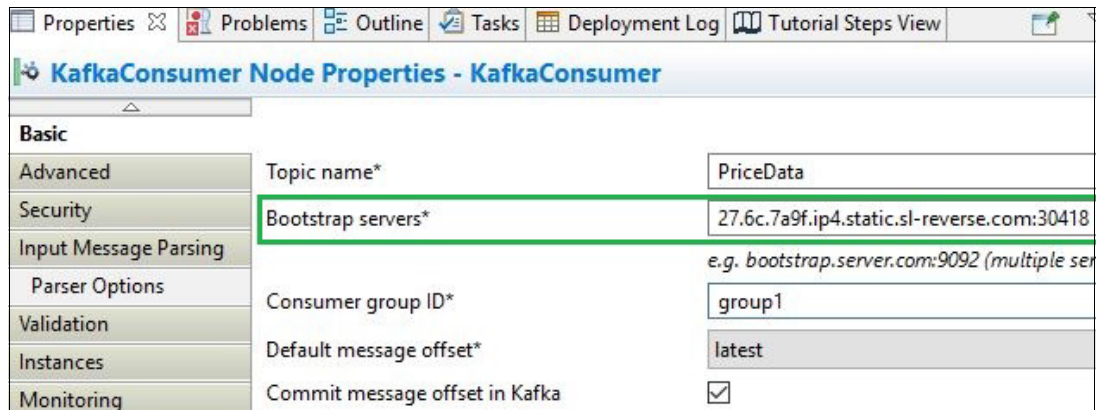


Figure 7-100 KafkaConsumer Node property configuration

2. You can configure the KafkaProducer or KafkaConsumer node to authenticate by using the user ID and password. You must set the Security protocol property on the node to SASL_SSL and SSL protocol to TLS v1.2 as shown in Figure 7-101 on page 533.

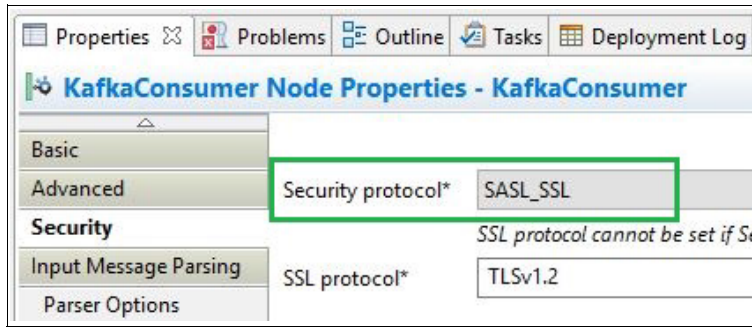


Figure 7-101 Security Protocol configuration

3. Similarly, you can configure KafkaProducer node as with Bootstrap server, Topic and Security protocol as shown in Figure 7-102.

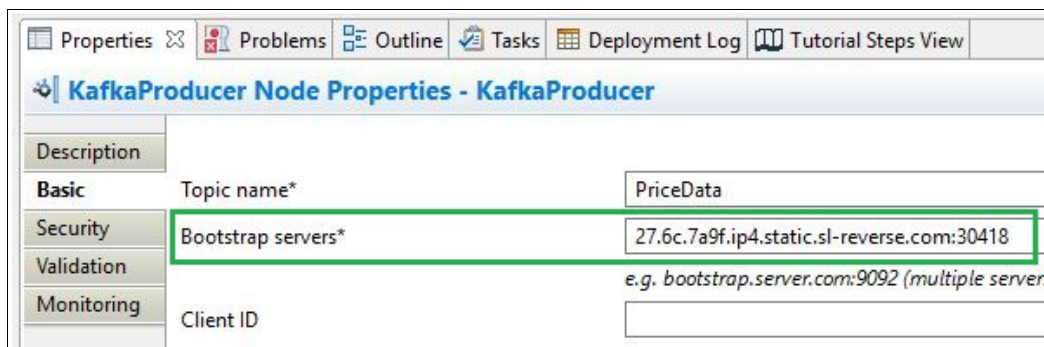


Figure 7-102 KafkaProducer node configuration

Generate a secret object for connecting from IBM App Connect helm release to the Event Streams instance

Next we need to generate a secret object for connecting from IBM App Connect helm release to the Event Streams instance.

1. Navigate to the directory where Secrets configuration package is extracted.
2. Update serverconf.yaml file to specify the security credentials parameter for accessing TrustStore under the JVM resource.

```
ResourceManagers:
  JVM:
    truststorePass: setdbparms::truststore
```

3. Copy the API Key from the previously downloaded es-api-key.json file to the setdbparms.txt file.

Note: Copy only key, not the quotation marks around it.

Add a row for security credentials for truststore access. You can use any dummy userid. Use the password obtained from the EventStream connection information.

```
kafka::KAFKA token A4q0H1A7iJ2J5aLfQDj-XtBMIqNBS0SgPaMtWxcjIPty
setdbparms::truststore dummy password
```

4. Extract the certificate from the previously downloaded the es-cert.pem file.

```
openssl x509 -in es-cert.pem -text
```

5. Copy the certificate, including the lines -----BEGIN CERTIFICATE----- and -----END CERTIFICATE----- and paste it to both files: **truststore-Cert-mykey.crt** and **keystore-mykey.crt**
6. Run **generateSecret.sh**.

```
./generateSecrets.sh my-kafka-secret icp4i
```

Deploy the BAR file

Perform the following to deploy the BAR files:

1. On the IBM App Connect dashboard, navigate to the Servers tab and click **Add server**.
2. Select BAR file and click **Continue**.
3. Copy the Content URL to the clipboard and click **Configure release**.
4. Helm chart opens. **Click Configure**.
5. Along with all usual configurations, ensure that you enter the secret name as defined previously in the step for generating secrets for accessing Event Streams. In our case, secret name is *my-kafka-secret* as shown in Figure 7-103.



Figure 7-103 Helm chart - Kafka Secret

6. On successful deploy of helm release, verify the connectivity with Kafka broker by processing the message through IBM App Connect integration flow.

7.9 Splitting an integration across on-premises and cloud

This section discusses available features in IBM App Connect that enables hybrid Integration between Cloud and on-premises deployments.

7.9.1 Overview

When you are building your hybrid or multicloud infrastructure, you must make an important architectural decision about splitting your integrations between these environments. IBM App Connect makes this possible with callable flows. A message flow can call another flow directly, both deployed on different platforms or environments, or indeed between on-premises and cloud.

A message flow can complete many different actions, if those actions are computation-intensive, you can split them from the main flow and complete them somewhere else. Callable flows also facilitate reuse because they can be called by multiple message flows. You can split your flows between different applications in the same integration server, or between different integrations servers, which can also be in different integration nodes. Callable flows must be in applications. You cannot deploy in libraries or integration projects.

In a hybrid integration scenario, some parts of a message flow might logically belong in a specific location. For example, if your flow queries an on-premises database, performance is better if that part of the flow remains on-premises. But if part of your flow queries a website multiple times, performance might be improved by running that part of the flow in the cloud.

You can call the cloud-based flow from your on-premises flow, and the flow in the cloud does not use any of your on-premises resources.

If you are splitting processing between different integration servers, your flows communicate by using a Switch server technology and connectivity agents. As shown in Figure 7-104, the Switch server communicates to the appropriate connectivity agents. The Switch server is a special type of integration server that routes data. You cannot deploy anything to the Switch server.

The connectivity agents contain the certificates that your flows require to communicate securely with the Switch server. The connectivity agents for connecting to a Switch server, must be running in the integration servers where you have deployed your on-premises message flows.

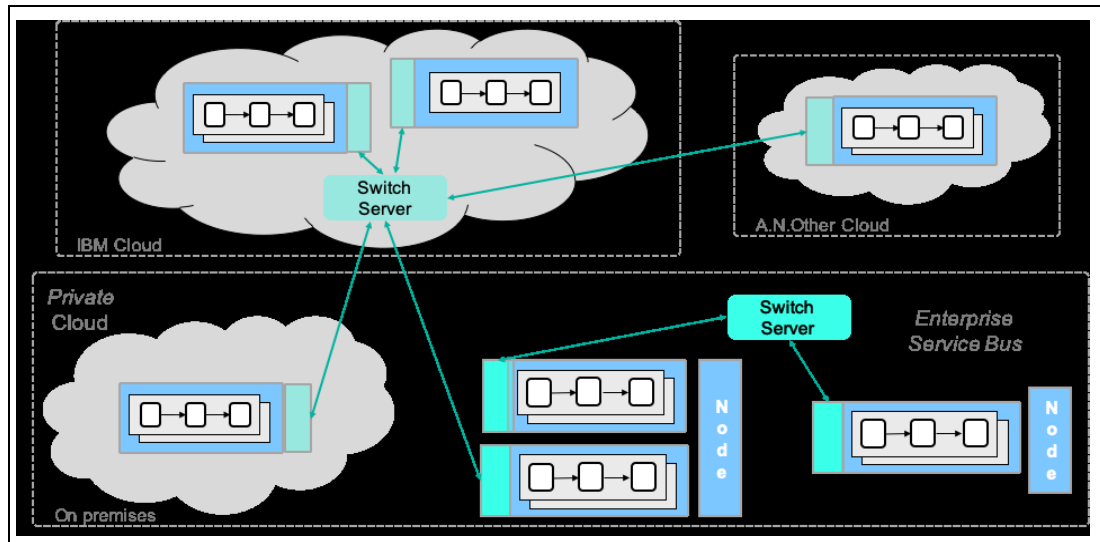


Figure 7-104 Callable message flows

Notice that if both of your callable flows are both deployed on IBM Cloud, they can communicate with each other as soon as you deploy them as the Switch server is automatically started for you. You do not need to set up communication between them. Similarly, you also do not need to set up communication if both flows are deployed to the same IBM App Connect integration server.

You can split processing synchronously between message flows by using the *CallableFlowInvoke* node in the calling flow, and *CallableInput* and *CallableReply* nodes in the callable flow. Alternatively, you can split processing asynchronously between message flows, by using the *CallableFlowAsyncInvoke* node in the calling flow, *CallableInput* and *CallableReply* nodes in the callable flow, and the *CallableFlowAsyncResponse* node in the response flow. You can also choose to share data between the flows that contain these asynchronous nodes (the calling flow and the response flow) by storing and retrieving data in the *UserContext* folder in the environment tree.

7.9.2 Using callable flows with IBM App Connect Designer

Flows created on the cloud in IBM App Connect Designer have the ability to call flows on App Connect integration servers in other environments. Here are some examples of the powerful integrations that are now possible:

- ▶ Integrate events from cloud-based applications like Salesforce, Workday, and Marketo with on-premises packaged applications like SAP. This allows the SAP system part of the integration to be run on-premises to remove the need to expose the SAP system directly to the internet. All communications with SAP are controlled by the callable flow that runs on-premises. Also, by not directly integrating the cloud integration to the SAP system, it reduces the number of network interactions that are required to one call from the cloud to the running callable flow.
- ▶ Integrate IBM App Connect Designer flows into on-premises transports like IBM MQ, file systems (including managed file transfer), and databases without having to allow direct access from the cloud to these transports. Each transport can be accessed by constructing a flow for an IBM App Connect integration server, and making it available as a callable flow.
- ▶ Allow the exchange of data from cloud-based formats that are based in JSON or XML to proprietary message formats like EDIFact, COBOL structures, or CSV. You can use callable flows to convert between these formats using the rich-message-parsing technology that is available in IBM App Connect Enterprise.

Figure 7-105 outlines how the use of callable flows authored in both IBM App Connect Toolkit and Designer can enrich your Hybrid Integration implementations.

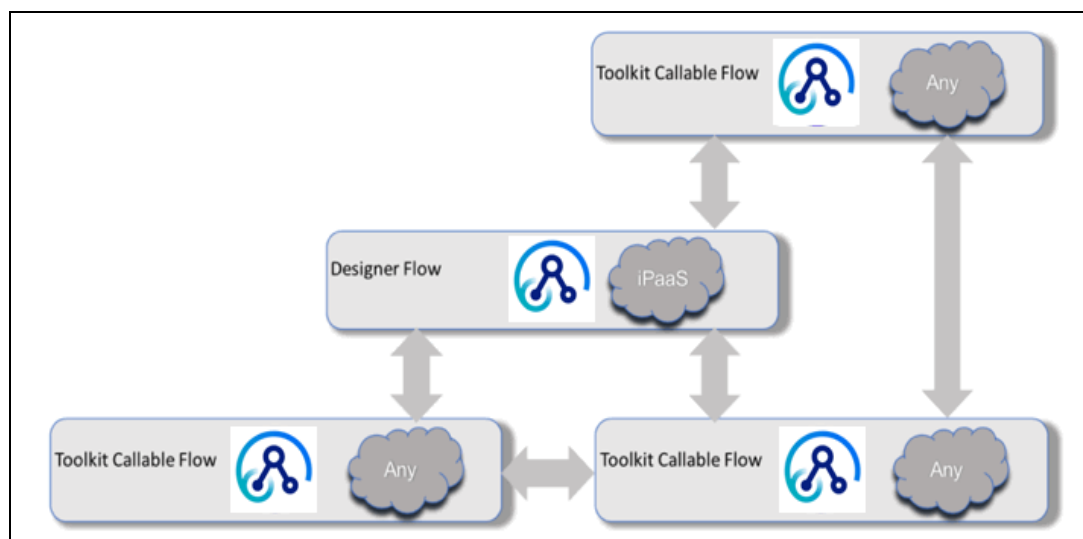


Figure 7-105 Callable flows between IBM App Connect Toolkit and Designer authored flows

If you need to call a callable flow in IBM App Connect Toolkit from Designer flow you can do that by using an application name that is identical to the IBM App Connect on IBM Cloud flow name, with the endpoint name set to default.

There's no need to copy any definitions from App Connect Toolkit to Designer or vice versa. After the IBM App Connect on IBM Cloud flow is started, it can be discovered and then executed by any flow that's configured with the correct application name and endpoint name.

As shown in Figure 7-105 on page 536, you can call Toolkit flows from other Toolkit flows. As discussed in the previous section, this is achieved by using the Switch server technology. The Switch server can be deployed either in IBM App Connect managed runtime (iPaaS) or deployed on your own cloud.

Access to Cloud Connectors

The expanded capability of the callable flows allows on premises IBM App Connect flows to make use of the growing selection of connectors to software as a service (SaaS) applications. This enables seamless hybrid integration where it is easy to construct integrations that combine function in the cloud and on-premises.

For example, this enables enterprise IT practitioners to augment data in SaaS applications that are used by line of business (LoB) users and developers. And they can retrieve data from SaaS applications for use in enterprise IT activities.

Figure 7-106 shows a simple IBM App Connect flow that uses cloud-based connectors for applications like Marketo and Eventbrite, and uses callable flows, which can utilize IBM App Connect integration servers on-premises.

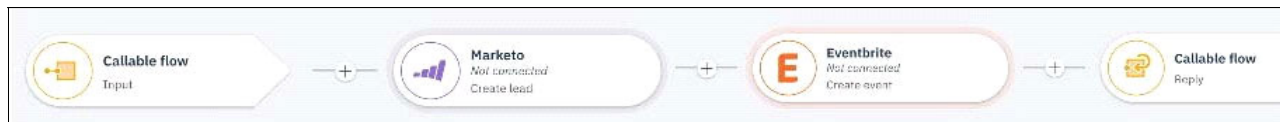


Figure 7-106 Simple IBM App Connect flow

The following link provides an example where a flow in IBM App Connect Enterprise (ACE) updates on-premises enterprise data:

<https://developer.ibm.com/integration/docs/app-connect-enterprise/tutorials/sharing-data-processing-premises-activities-cloud-saas-applications-using-callable-cloud-flow/>

The example then calls an event-driven flow in IBM App Connect on IBM Cloud to pass enterprise data to SaaS applications and to get data from SaaS applications for processing on-premises.

7.9.3 Callable flows versus APIs

IBM App Connect has very good REST support so REST can allow splitting processing between two flows. What is more, IBM App Connect integration server has a REST node specifically for calling cloud-based integrations.

However, in many cases callable flows are a better choice:

- ▶ Do not require opening any on-premises TCP/IP ports
- ▶ Flows can be moved seamlessly between on-premises and cloud servers without the calling flows being changed.
- ▶ Requires less skill to understand how to call and implement a callable flow.
- ▶ Removes the need to integrate the IBM App Connect integration servers.
- ▶ Allows for future enhancements of your message flows that could not be simply added on top of REST.

7.9.4 Cloud debugger for ACE on Cloud applications

Message flows are routinely deployed in IBM App Connect integration servers on IBM Cloud. When you work with these message flows, you might need to troubleshoot or investigate data as it travels through those flows.

Now, you can use Cloud Debugger to remotely debug your message flows. This new feature utilizes the Switch technology and the connectivity agents, which enables visual debugging through an on-prem installation of IBM App Connect integration server.

Figure 7-107 shows the different agents available to connect to the Switch server, with highlighted Agent C, which enables the visual debugging on-premises.

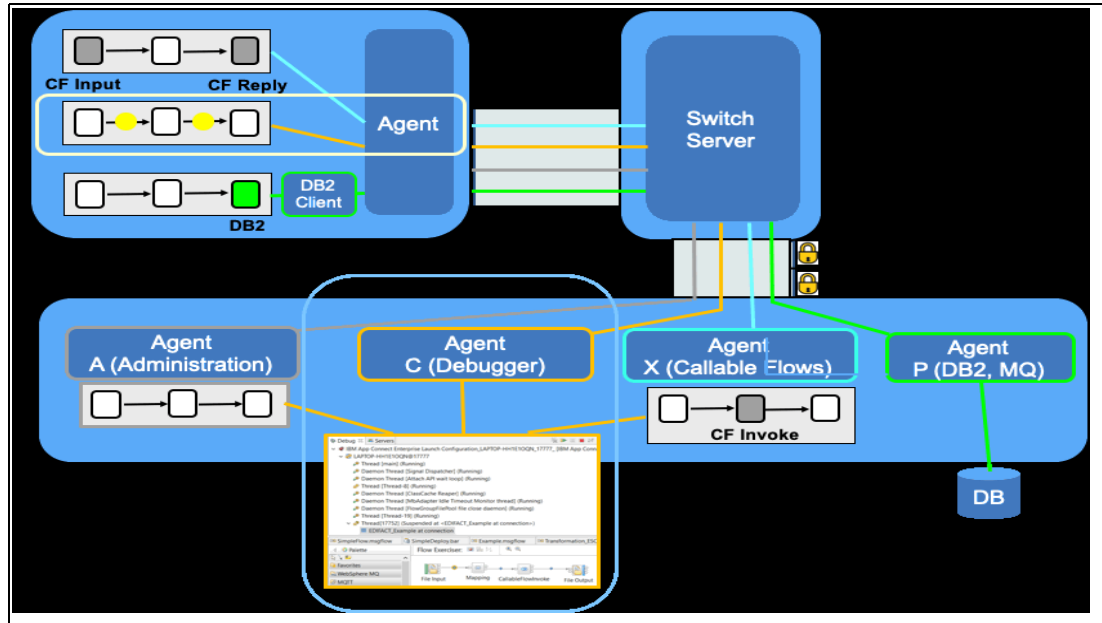


Figure 7-107 Different agents available to connect to the Switch server

The Switch Server ensures secure communication between the Cloud deployment and on-premises IBM App Connect Toolkit. This hybrid approach brings all the benefits of the Toolkit debugger, so you can easily set break points and view the data as the messages step through the message flow.

See the following blog post (and linked articles) for practical guidance on setting up your hybrid environment and debugging message flow that runs on IBM App Connect on Cloud: (<https://developer.ibm.com/integration/blog/2019/04/30/remotely-debug-your-enterprise-integrations-in-ibm-app-connect/>)



Field notes on modernization for API lifecycle

In this chapter, we explore some best practices for customers planning to introduce API management into their integration landscape. The target audience for this chapter is those currently acquiring skills in IBM API Connect who want to understand some of the broader architectural concerns. We explore at a product level what topics around hybrid cloud boundaries, gateway location, automated provisioning, high availability, and testing.

This chapter has the following sections:

- ▶ Move from DataPower only to API Connect
- ▶ Enterprise APIs across a hybrid or multicloud boundary
- ▶ How many API Connect Clouds and Gateways
- ▶ Organization, Catalog and Space responsibilities for APIs
- ▶ Automated provisioning of a new API provider team
- ▶ High availability and scaling on containers for API Management
- ▶ IBM API Connect API Test Pyramid

8.1 Move from DataPower only to API Connect

In this section, we look at the situation where an enterprise might be using an IBM DataPower Gateway alone in order to expose APIs. Then, we compare that to the benefits they would gain by moving to IBM API Connect (APIC) to provide a complete API management solution. We begin by looking at the capabilities of IBM DataPower Gateway in isolation.

IBM DataPower Gateway helps provide security, control, integration, and optimized access to a full range of mobile, web, application programming interface (API), service-oriented architecture (SOA), B2B and cloud workloads.

IBM DataPower Gateway is available in physical, virtual, cloud, Linux, and Docker form factors.

DataPower is extensively used in the industry and its usage can be broadly categorized into the following patterns:

- ▶ Security Gateway, as shown in the figure, placed between the firewalls
- ▶ API gateway, both as internal and external gateway
- ▶ Providing connectivity and mediation services in the internal network, close to the system of record

Figure 8-1 shows DataPower usage patterns.

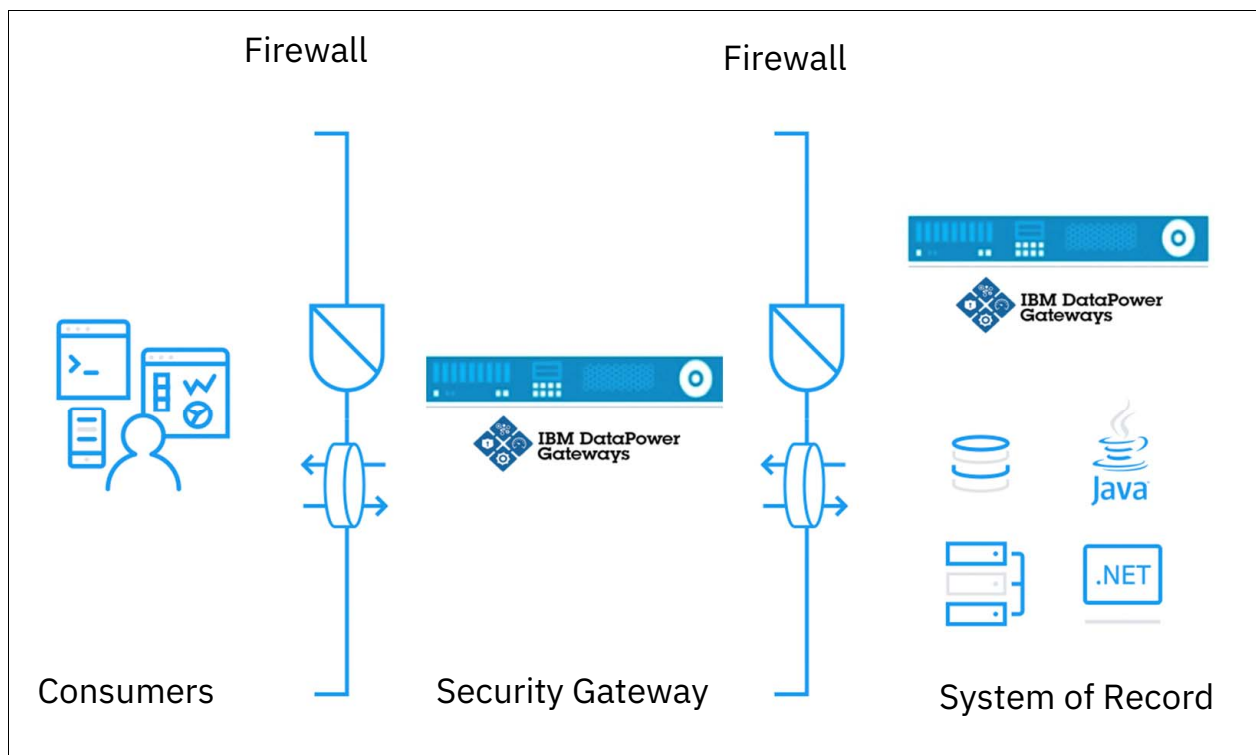


Figure 8-1 DataPower usage patterns

Let's discuss the patterns in detail.

8.1.1 Security Gateway

DataPower is purpose-built, DMZ-ready and is used for both policy enforcement and consistent security policies across business channels. As a result, you reduce operating costs and improve security. DataPower can,

- ▶ Protect against unwanted access, denial of service attacks, and other unwanted intrusion attempts from the network
- ▶ Verify identity of network users (Identification and Authentication)
- ▶ Protect data and other system resources from unauthorized access (Authorization)
- ▶ Protect data in the network by using cryptographic security protocols:
 - Data endpoint Authentication
 - Data origin Authentication
 - Message integrity
 - Data confidentiality
- ▶ Provide proxying and enforcement:
 - Terminate incoming connection
 - Terminate transport-level security (SSL/TLS offload)
 - Threat protection
 - Enforce service level agreement policies
 - Inspect message content and filter (schema validate)

DataPower also supports important security standards like OpenID Connect, OAuth, WS-Security, JWT. Because it is based on configuration-driven policy creation, it provides an extensible platform to provide first grade security to enterprise applications with minimum development effort.

For example, in Figure 8-1 on page 540 the DataPower in the DMZ provides a consistent and advanced security gateway for the downstream applications. Without DataPower, these applications would have to develop their own enterprise grade security in order to expose or use external services. The overhead of building and maintaining such security modules is avoided by using DataPower.

8.1.2 API gateway

DataPower is also often used for exposing internal APIs to consumers. It acts as a facade for the internal APIs as it has features like:

- ▶ Web services infrastructure needed to support highly secure data routing with daily high volume and sensitive nature of information.
- ▶ Centralized service governance and policy enforcement.
- ▶ Service level monitoring (SLM) to protect your services and applications from over-utilization.
- ▶ Application optimization, which uses dynamic runtime conditions to distribute based on topology and workload.

8.1.3 Connectivity and mediation

DataPower is specialized to its task as a low latency gateway, and as such, it focuses on optimizing a very specific set of protocols, transports, and data formats. It excels at HTTP based traffic, with a particular focus on XML and JSON data formats, and then further specializes on the routing, security, and traffic management aspects expected from a gateway. It should not be seen as a general-purpose mediation and connectivity capability. It does have broader capabilities including database, mainframe, IBM MQ, JMS, FTP and more, but such integrations can quickly become much more complex and ill-suited architecturally and technically to a gateway component. As such they will be better suited to other integration components.

IBM App Connect is the premier integration solution for providing complex mediation. It enables development of integrations by using a configuration, not coding approach, with premade integration templates, and rich connectors to speed development time.

8.1.4 DataPower and API Connect compared

DataPower includes features that allow it to work as an API gateway and compliments API Connect, which is custom-built to solve the challenges of enterprise API management. API Connect adds the following additional features to the DataPower gateway services:

- ▶ API lifecycle management (for example active, retire, stage).
- ▶ API governance (for example best practices, security, access, versioning).
- ▶ Analytics, dashboards, third party data offload for usage analysis.
- ▶ API socialization based in a portal that allows self-service for developer community.
- ▶ API developer toolkit, which provides tools API developers need for modelling, developing, and testing APIs and LoopBack applications, and then publishing them to API Connect.

8.1.5 DataPower to API Connect migration

In the API gateway pattern, organizations use DataPower as a facade for internal services built on top of system of record. Figure 8-2 shows such a scenario where consumers use DataPower as an API gateway to expose services available on system of records.

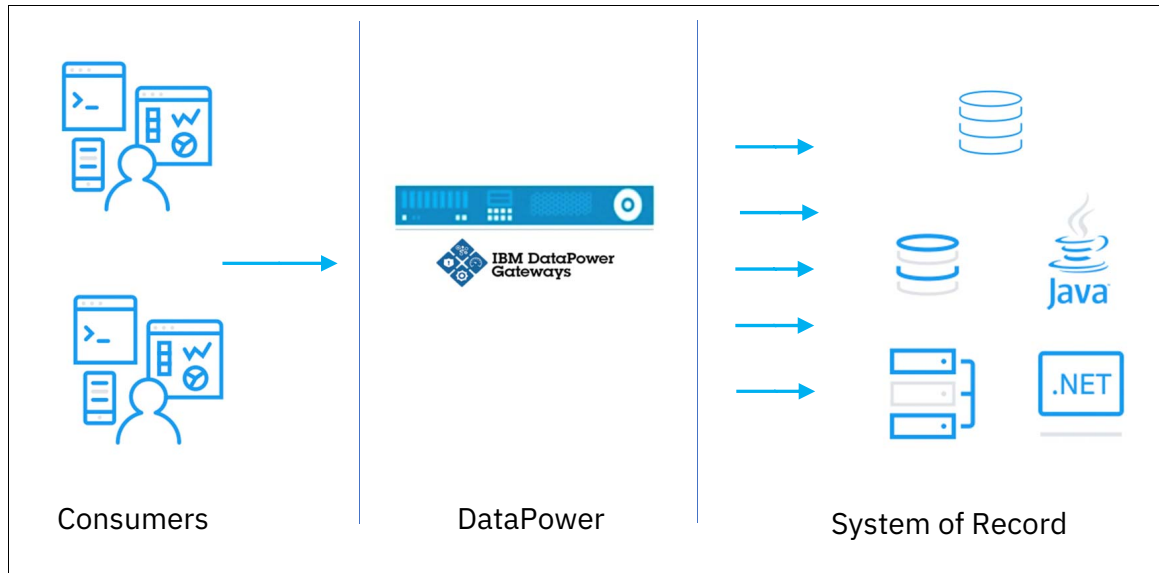


Figure 8-2 DataPower as API gateway

- ▶ It is difficult to onboard new API subscriber without lots of hand holdings. The current API details like endpoints, request/response message format, error conditions, test messages, SLAs are not easily available.
- ▶ It is difficult to tell which subscribers are really using the API and how often, without building a custom solution.
- ▶ It is not possible to differentiate between business-critical subscribers versus low value subscribers.
- ▶ It is difficult to manage changes in the API.
- ▶ There is no dynamic scaling that is built into the solution, which often means making hardware investment for max load or worst availability scenario.
- ▶ Service registry is often not in sync with latest service as the process is manual.
- ▶ APIs might not be following consistent security rules.
- ▶ It is not easy to move from WSDL to REST-based services.

A modernized solution can be built that uses API Connect and microservices as shown in Figure 8-3 on page 544.

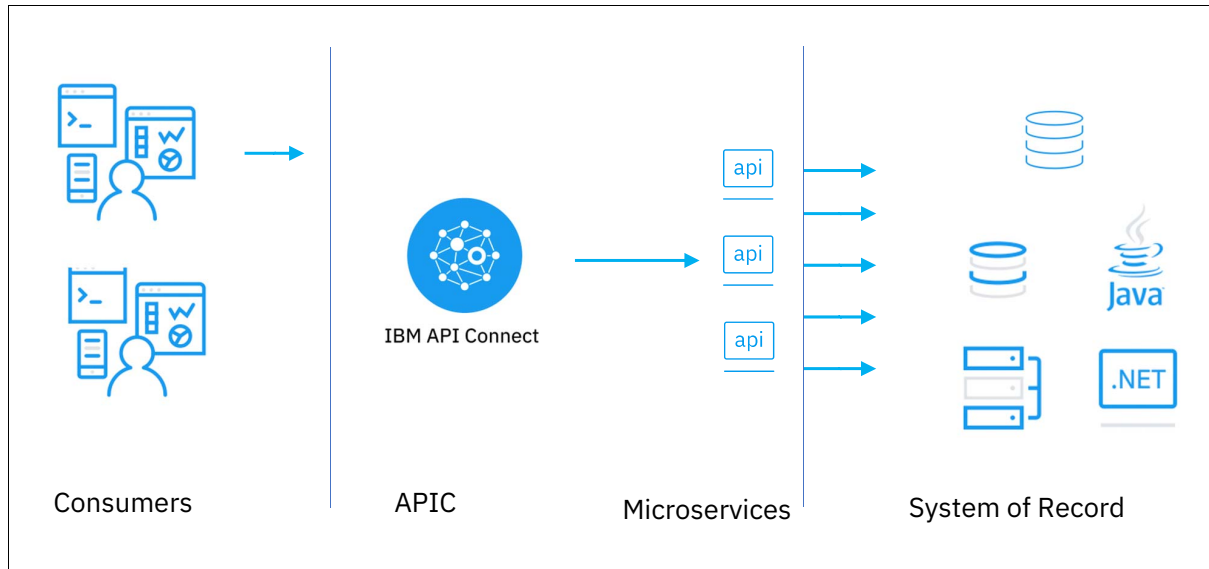


Figure 8-3 Using API Connect as API gateway

- ▶ API Connect provides a Developer Portal, which enables smooth onboarding of new API subscriber. The portal provides all the relevant information that is required for using the API.
- ▶ API Connect Analytics can be used to analyze event data through visual charts. It also allows the offloading of data to third-party systems like HTTP, Elasticsearch, Apache Kafka, Syslog. They can be used to derive API usage data.
- ▶ API Connect allows the packaging of APIs as product and plans. These can be used to provide differentiated services to consumers.
- ▶ API Connect provides API lifecycle management (for example Draft, Staged, Published, Deprecated, Retired, Archived) out of the box.
- ▶ API Connect can be deployed on cloud to leverage the elasticity, flexibility, and scalability of cloud.
- ▶ API Connect supports both WSDL and REST, supports industry standard Swagger specifications and supports modern security frameworks like OAuth, OIDC, JWT.

The preceding scenario describes the advantages of using API Connect over DataPower. The real value can be derived by building API platform with API Connect where different business value streams can host their APIs on this platform via self-service. This can be achieved through:

- ▶ Rules-based API governance (discovery, naming standards, security patterns, versioning, approvals).
- ▶ Automated CI/CD that uses tools like Jenkins and UrbanCode.
- ▶ Automated testing.
- ▶ Modern deployment practices like blue/green, A/B, canary deployments.

Conclusion

In this section, we examined various DataPower usage patterns. Also we discussed scenarios where the primary use case is API exposure. In such cases, API Connect provides a complete API management solution alongside DataPower, as opposed to using DataPower in isolation.

8.2 Enterprise APIs across a hybrid or multicloud boundary

In this section, we explore the transition from a solely on-premises API management topology to a topology that includes both on-premises and one or more cloud-based deployment destinations.

Before we do that though, we need to carefully define a couple of terms that we will use throughout the section.

► **API Connect topology** (in place of *API Cloud*)

IBM API Connect documentation defines the concept of an *API Cloud*, which essentially refers to a single installation of the components of IBM API Connect. It is made up of one management service, which is associated with one or more gateway services and zero or more portal services and analytics services. However, in this section we will use the term “cloud” when referring to a cloud destination such as IBM Cloud, or a third-party cloud such as AWS. As such, we will instead use the term *API Connect topology* in place of *API Cloud* to avoid any confusion around the term “cloud”.

► **Availability zone**

An availability zone is a group of logical or physical data centers that are connected via a low latency connection. To achieve that low latency, the whole zone must be within a relatively small geographical boundary. In order to ensure availability even during the occurrence of extreme events, the API Connect topology might need to be spread across availability zones.

Availability zones are a known construct within API Connect and you can add as many availability zones into your API Connect topology as needed depending on business requirements.

Availability zones can contain one or more Gateway services, Analytics service, and Portal service. But there is only one Management service, which administers the availability zones that are used by an API Connect topology.

When configuring your availability zones and services, the recommended practice is that the gateway, analytics, and portal services do not communicate across availability zones. Also, a single component service must not span multiple clouds or Kubernetes clusters. So for example, any given gateway service must live within a Kubernetes cluster, which is itself within *one cloud environment* (for example AWS, IBM Cloud, Google Cloud Platform). Only the Management service can communicate across availability zones, providing flexibility in deployment scenarios. See Figure 8-4 on page 546.

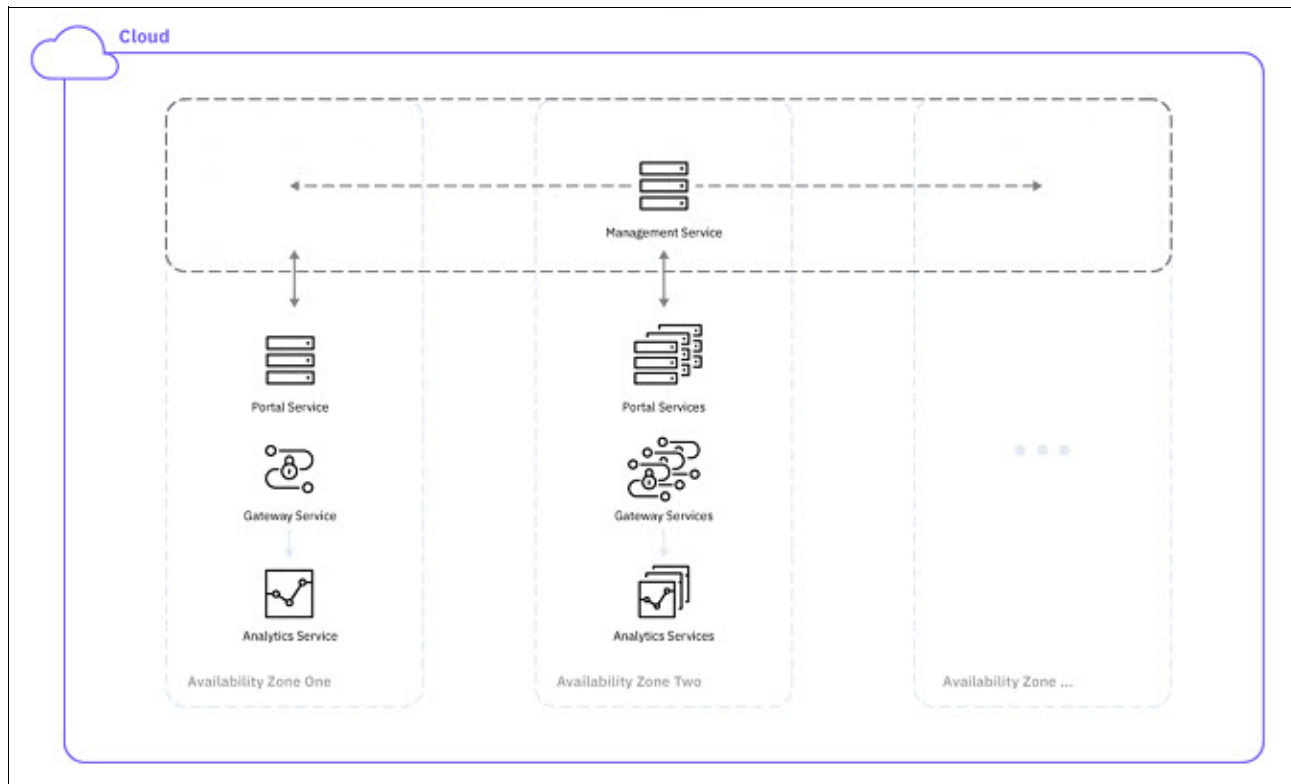


Figure 8-4 IBM API Connect spread across availability zones

Hybrid and multicloud deployment options

There are many permutations of how API Connect topologies can be spread across on-premises and cloud destinations. We discuss two of the most common. Many of the other permutations can be derived by using combinations these options.

- ▶ Option 1: One API Connect topology, with federated cloud deployment destinations
- ▶ Option 2: Multiple API Connect topologies, across multiple cloud destinations

Option 1: One API Connect topology, with federated deployment destinations

API Connect components (Gateway, Analytics, Portal) can be deployed across several clouds for regional flexibility. Or the components can be near systems of records to reduce latency, while still being governed by a centralized Management component.

This pattern, as shown in Figure 8-5, has the following characteristics:

- ▶ Different target applications are present across the multiple clouds and environments.
- ▶ API Gateway services are colocated with target back-end applications to reduce latency.
- ▶ Web and mobile applications consume the applications via their geographically disperse API gateways.
- ▶ Developer Portals can be separated, or centralized depending on API characteristics exposed from different clouds (for example different Developer Portals for internal and external APIs)
- ▶ For API Analytics, there are two options:
 - Colocate analytics in same AZ or Cloud for security and networking latency reasons.
 - Centralize to a single API analytics service.

See Figure 8-5.

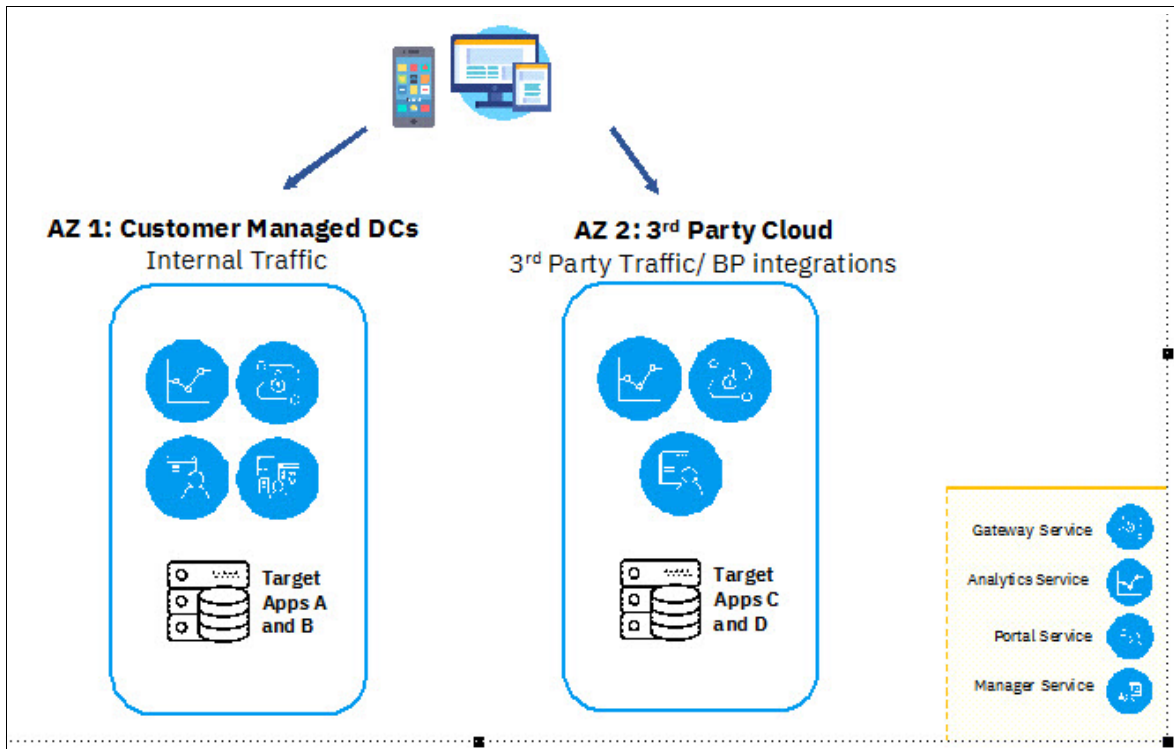


Figure 8-5 Hybrid cloud topology pattern

Option 2: Multiple API Connect topologies, across multiple cloud destinations

You might not want to retain a centralized management across multiple cloud locations as we have done in the first two options. Instead, you could have a separate API Connect topology in each location. This makes sense when the different cloud regions have different ownership within the organization. Perhaps they are different domains of the business, and do not want any dependencies on other domains. See Figure 8-6 on page 548.

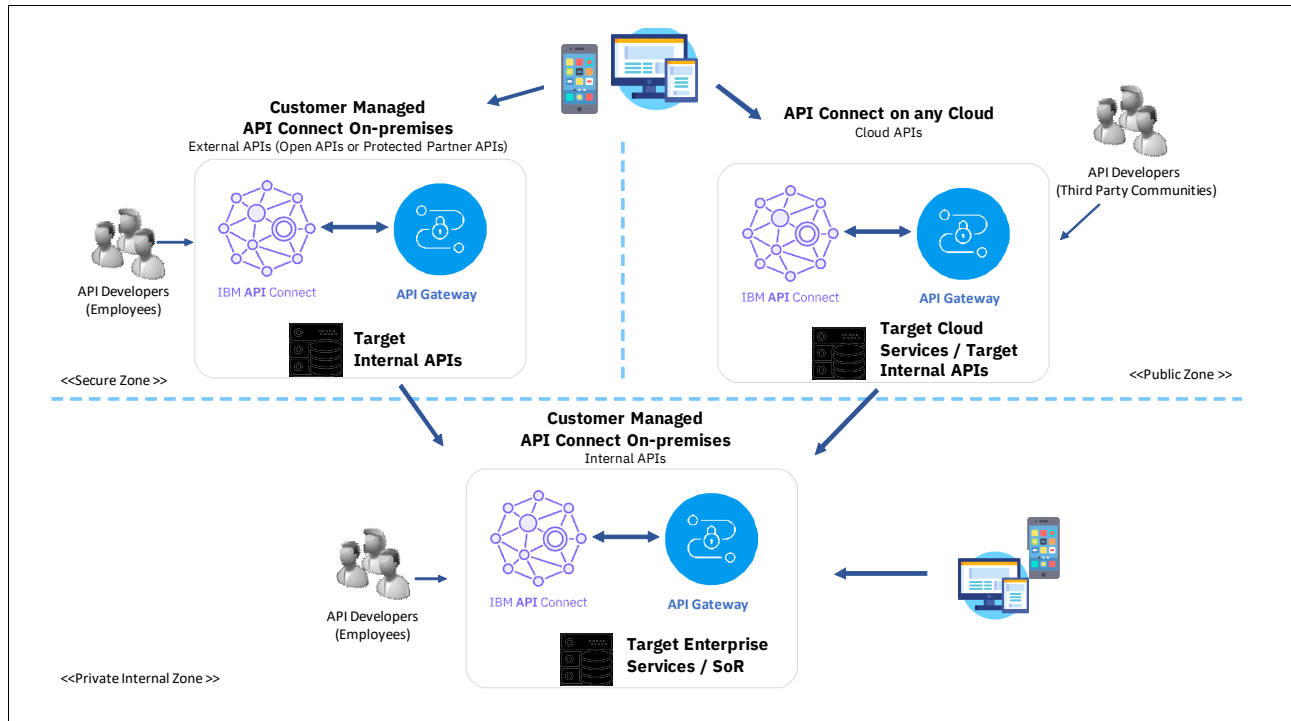


Figure 8-6 Build new API Connect Cloud

8.3 How many API Connect Clouds and Gateways

Determining when to create new API Connect Clouds and Gateway clusters is a separate consideration to determining when to scale or when to consider multi cloud deployments. Organizations should consider a number of factors from impact of cloud upgrades, logging, availability through to the level of coupling between teams.

8.3.1 Separate API Clouds

There are various requirements that might lead to the creation of a new team on a separately provisioned cloud. Cloud-level separation allows for:

- ▶ Decoupling of platform upgrade requirements across API Provider teams.
- ▶ Separation of Cloud management concerns, which might include funding, management team skills and management team capacity.
- ▶ Separation of logging and analytics between teams.
- ▶ Varying security requirements of API Provider teams, which might mean overly restrictive procedures and protocols.
- ▶ Supporting varying or conflicting non-functional requirements.

8.3.2 Separate API Gateway Cluster

There are various requirements that might lead to the creation of a new channel for API traffic. Each new DataPower Service in the Cloud Manager relates to a new gateway domain. That means you have isolation of traffic to the cluster of gateways that this configuration is pushed to. Gateway separation allows for:

- ▶ Isolation of traffic with differing availability, throughput, and security requirements.
- ▶ Decoupling of teams who require global policies to be applied.
- ▶ Decoupling of gateway upgrade requirements across API Provider teams.
- ▶ Separation of exposure between internal and external zones, including endpoints and ports.
- ▶ For non-functional requirements around application affinity such as connecting to zones with lower latency.

8.4 Organization, Catalog and Space responsibilities for APIs

API Provider teams will be distributed within API Connect on one of three levels: Organization, Catalog, or Space. An Organization contains Catalogs, which can be optionally split into Spaces. There are certain considerations that will help system architects make these decisions.

The article at the following URL discusses what actions can be performed at each layer and the benefits this brings to organizing teams, including the roles and responsibilities that might be assigned:

<https://developer.ibm.com/apiconnect/2019/07/18/organizingteamsinapic/>

Table 8-1 shows the decision matrix.

Table 8-1 Decision matrix

One portal for all API Exposure?	Centrally managed/support ed Infrastructure?	Isolated and independent development teams?	Isolation of billing, security, roles, and registration?	Recommendation
y	y	n	n	Catalog
y	y	y	n	Spaces
n	n	y/n	y	Organization
n	y	n	n	Catalog
n	y	y	n	Spaces

Each of the four factors affects the decision on how to split teams in API Connect. This is because the results can affect the outward appearance of the brand to API consumers, the risk that is exposed in production environments, and the ability of teams to interoperate in a decoupled way.

- ▶ **One Portal:** Organizations wanting to expose and manage all their APIs from a single location want a single portal, which has a one-to-one mapping with a Catalog.

A single portal means a simpler route to live as only one site needs updating. It means greater architectural simplicity when thinking of API exposure. It also means that login processes and portal user management are in a single location. API applications will all show in a single location for the API consumer, improving consistency of look and feel for all users.

Portal separation is preferred in some scenarios, for example internal APIs and external APIs. You can handle visibility and subscribability in a single Catalog by using communities. And multiple portals can be customized to give the impression of being a single portal.

- ▶ **Centrally managed and supported Infrastructure:** Where a single team owns the API Connect Cloud (including gateway definitions, gateway mapping and TLS profiling), all options are available for splitting of teams. However, when there is no centralized team exists, API Provider teams need to manage their APIs. This means that each team should have their own organization if teams are intended to be isolated and decoupled.

It is worth noting that teams that exist in their own organization will have their own portals. Yet, they might still be tightly coupled to other teams on the same cloud for upgrades, certificate management, and gateway provisioning.

- ▶ **Isolated and independent development teams:** Teams can manage their API lifecycle without interaction and in some cases awareness of other teams. By setting permissions and roles at any of the levels managing the API lifecycle without interaction can be achieved, but this ties in with other considerations like one portal.

Even when teams are isolated, there might still need coordination from a central team to keep consistent API naming conventions, exposure requirements, for example only to select organization types, and deployment timelines.

- ▶ **Isolation of billing, security, roles, and registration:** Certain aspects that are defined at the organization layer would need to be considered and might include how API Provider teams log in to API Connect, how billing is configured, and whether TLS Profiles are available to other teams in a tightly restricted API management environment.

These four factors along with the roles, responsibilities, and actions that are detailed for each API Connect level need to be considered in unison for a clear and scalable approach when splitting teams. The use of the matrix looks to set recommendations at a high level, but each organization has different needs and as such one size will not fit all.

8.5 Automated provisioning of a new API provider team

As new API Provider teams join the API Connect Cloud, you need the ability to quickly provision new deployment targets. Typical targets include a new Organization with a Catalog, a new Catalog in an existing Organization, or a new Space in an existing Catalog.

New teams have a variety of requirements. They need administrators and owners that are assigned, the fully qualified deployment target must be passed to API deployment teams, new TLS profiles might need to be added for both upstream and downstream authentication, new gateway clusters must be created, new communities and elementary tests must be completed to confirm that the setup was successful.

Manual process

Initially a manual process might be used to quickly onboard teams that want to use API Connect. This manual process is likely to be communication heavy and requires both infrastructure and API deployment teams for task completion and verification.

Typical tasks from an API Connect perspective that might need completing will be as follows (Table 8-2).

Table 8-2 Manual process - Typical tasks

Step	Task	Team	Automatable
1	All configurations defined and confirmed by all relevant parties	Infrastructure team, Architects, API Provider Team	No

Step	Task	Team	Automatable
2	Setup new Gateway Cluster (If applicable)	Infrastructure Team	Yes with user interaction
2a.	Build and configure the IBM DataPower appliance	Infrastructure Team	Yes
2b	Create a TLS profile and configure the gateway cluster in the cloud manager adding the relevant servers	Infrastructure Team	Yes
3	Create the new provider deployment target (Organization/Catalog/Space as applicable)	Infrastructure Team	Yes
4	Assign defined administrator/owner of the new deployment target	Infrastructure Team	Yes
5	Configure the deployment target to be associated with the relevant gateway cluster	Infrastructure Team	Yes
6	Deploy a simple API with dynamically determined endpoints and complete tests	API Provider Team	Yes
7	Add other users to the deployment target - API Developer, Lifecycle Authorities, Analytics Viewers	Infrastructure Team	Yes
8	Add predefined artifacts to the deployment target for example downstream TLS Profiles, policies	Infrastructure Team	No
9	Send deployment target details to relevant teams for deployment pipeline	Infrastructure Team API Provider Team	Yes
10	Set up Developer Portal (If applicable)	Infrastructure Team	Yes
10a	Enable and configure Developer Portal with validation	Infrastructure Team	Yes
10b	Set up admin user and add relevant members to the portal	Infrastructure Team	Yes

This process might take anywhere from a couple of hours to several days depending on the teams that are involved and whether all the requirements have been met.

Automated process

As sophistication within an API Connect team grows, there will be an ability to automate the process of provisioning new teams. This might use existing APIs exposed within the product, or will use deployment tools such as Kubernetes automation processes.

https://openwhisk.ng.bluemix.net/api/v1/web/API%20Connect%20Native_apic-on-prem/default/index.http

In order to call REST APIs on the API Connect Cloud a client application must be registered, a scope defined, and a bearer token obtained. From here, as much of the pipeline can be automated as possible. The new process with automation is described in Table 8-3.

Table 8-3 Automated process

Step	Task	Automation
1	All configurations defined and confirmed by all relevant parties	N/A
2	Create the new Provider deployment target (Organization/Catalog/Space as applicable)	Later automation of the gateway service requires an organization to be created. If the deployment target is a new Organization this step must be completed first; POST /cloud/orgs POST /orgs/{org}/catalog POST /catalogs/{org}/{catalog}/spaces
3	Set up new Gateway Cluster (If applicable)	
3a.	Build and configure the IBM DataPower appliance	The creation of Gateway services is a little subtle at the moment ^a . We anticipate making improvements in this area if automation of this type becomes commonplace.
3b	Create a TLS Profile and configure the gateway cluster in the cloud manager adding the relevant servers	Create TLS profile: POST /orgs/{org}/tls-client-profiles Add Gateway Service to the cloud manager: POST/orgs/{org}/availability-zones/{availability-zone}/gateway-services
4	Assign defined administrator/owner of the new deployment target	Depending on the deployment target, assign owners: POST /orgs/{org}/transfer-owner POST /catalogs/{org}/{catalog}/transfer-owner POST /spaces/{org}/{catalog}/{space}/transfer-owner Depending on the deployment target - Assign an administrator: POST /orgs/{org}/members POST /catalogs/{org}/{catalog}/member-invitations POST /catalogs/{org}/{catalog}/members POST /spaces/{org}/{catalog}/{space}/member-invitations POST /spaces/{org}/{catalog}/{space}/members
5	Configure the deployment target to be associated with the relevant gateway cluster	For all three deployment targets the catalog should be associated: POST /catalogs/{org}/{catalog}/configured-gateway-services For spaces: POST /spaces/{org}/{catalog}/{space}/configured-gateway-services
6	Deploy a simple API with dynamically determined endpoints and complete tests	Deploy API: POST /orgs/{org}/drafts/draft-apis Complete API Test. Delete the API after test complete: DELETE /orgs/{org}/drafts/draft-apis//orgs/{org}/drafts/draft-apis
7	Add other users to the deployment target - API Developer, Lifecycle Authorities, Analytics Viewers	POST /orgs/{org}/member-invitations/{member-invitation}/register POST /orgs/{org}/member-invitations/{member-invitation}/accept
8	Add predefined artifacts to the deployment target, that is, downstream TLS Profiles, policies	Deploy relevant objects for example: POST /catalogs/{org}/{catalog}/configured-tls-client-profiles POST /catalogs/{org}/{catalog}/configured-gateway-services/{configured-gateway-service}/global-policies
9	Send deployment target details to relevant teams for deployment pipeline	Automatically send API Deployment target based on the deployment target type, and the names given for each, as in these examples: POST /catalogs/{org}/{catalog}/stage POST /catalogs/{org}/{catalog}/publish

Step	Task	Automation
10	Set up Developer Portal (If applicable)	
10a	Enable and configure Developer Portal with validation	PUT /catalogs/{org}/{catalog}/settings
10b	Set up admin user and add relevant members to the portal	New Consumer Organizations can be set up to host API Provider users in the new Portal: Create an Org: POST /consumer-api/orgs Invite new users: POST consumer-api/orgs/{org}/member-invitations POST consumer-api/orgs/{org}/members

a. Gateway Services need to be deployed independently using the subsystem definitions for the gateway leveraging the apicup installer, requiring specific editing of namespace and image registry. This can then be deployed ready for automated addition to the API Connect Cloud as above

The guideline provided for automation shows that REST APIs are available on the API Connect Cloud in order to sufficiently automate the provisioning of API Provider teams. Individual use cases might vary and the assembly of the different REST APIs into a process need customization into some form of pipeline.

Rollback

In the event of a rollback, there are two options available. The first option is to use backup configurations of the system to bring the system back to before the changes were made. However, this risks losing API, product, and membership data, which would be a heavy-handed approach.

The second option is a fix forward strategy. This can be achieved by using the available REST APIs to GET all the relevant resources that might be touched by the automation or manual process. Given an event that requires a rollback, the system can be restored to its previous configuration (active Organizations and Catalogs). You can use either a manual process or further automation that resolves the configuration to a pre-change state.

We can now provision new API Provider teams using both manual and automated steps in API Connect.

8.6 High availability and scaling on containers for API Management

As discussed in the introductory chapters, agile practices in integration are giving rise to new patterns of deployment. We need the API management infrastructure to be able to fully use the cloud-native infrastructure, both from a high availability point of view and also to enable scaling. In this section, we explore how deployment on containers enables API Connect to provide a more nuanced availability and scalability model. This uses the underlying standards-based Kubernetes platform to look after the infrastructure orchestration, simplifying installation and maintenance of the overall topology.

8.6.1 High availability in a containerized environment

API Connect leverages Kubernetes as a container orchestration platform and also the Quorum concept (described in “Quorum” on page 555) to provide a sophisticated high

availability model. This can withstand many common failure scenarios and still provide the best possible service under the circumstances.

Notice that this section discusses only the availability of API Connect running on a Kubernetes platform. Clearly there is also a need to ensure that the core components of the Kubernetes platform itself are configured to a suitable degree of availability (for example the Master, Management and Proxy). This is already explained in detail in Kubernetes documentation (<https://kubernetes.io/docs/home/>). Kubernetes platform is highly available. The point here is to ensure that it provides a level of availability that you need. That availability must be aligned with the availability you need for the components such as API Connect that you intend to run on Kubernetes.

Before we discuss high availability any further, let's highlight two important topics:

APIC components

First let's refresh our knowledge of the different components within API Connect. As we will see later, they each behave differently in the various availability scenarios. API Connect consists of four components as shown in Figure 8-7 on page 554.

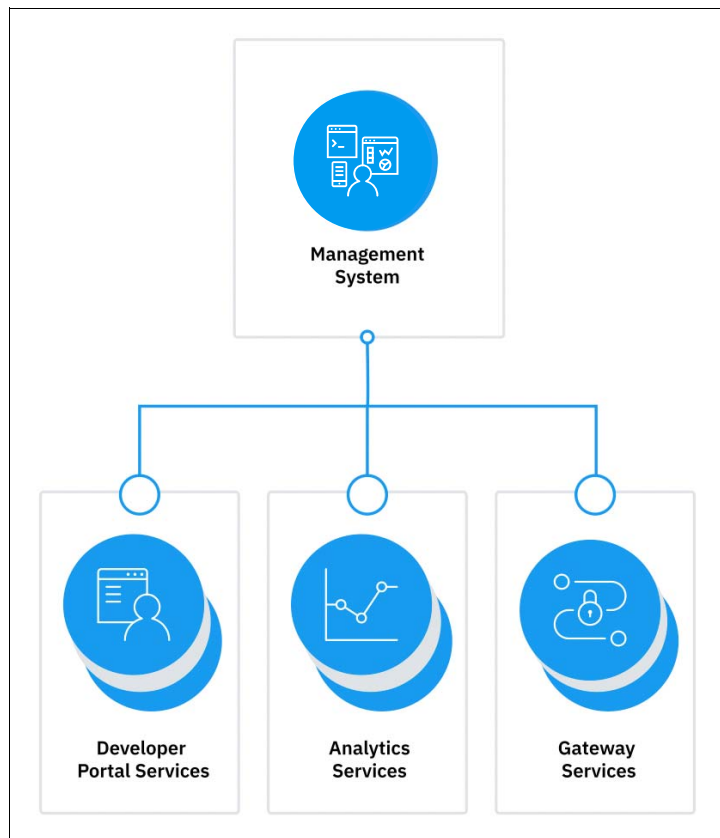


Figure 8-7 Components of API Connect

- ▶ *Management System* is responsible for creation, publication, and management of APIs. It is also used for managing the API Connect cloud infrastructure.
- ▶ *Developer Portal Services* is where consumers can explore and subscribe APIs, create applications, and generate credentials.
- ▶ *Analytics Services* is used for monitoring API usage and responses across the gateway service.

- ▶ *Gateway Services* is used for enforcing API traffic, rate limits, throttling, and security.

Gateway services, Developer Portal services, and Analytics services are scoped to single availability zone unlike Management system, which can communicate across availability zones in an API cloud topology.

Figure 8-8 on page 555 shows the context in which the different components are used.

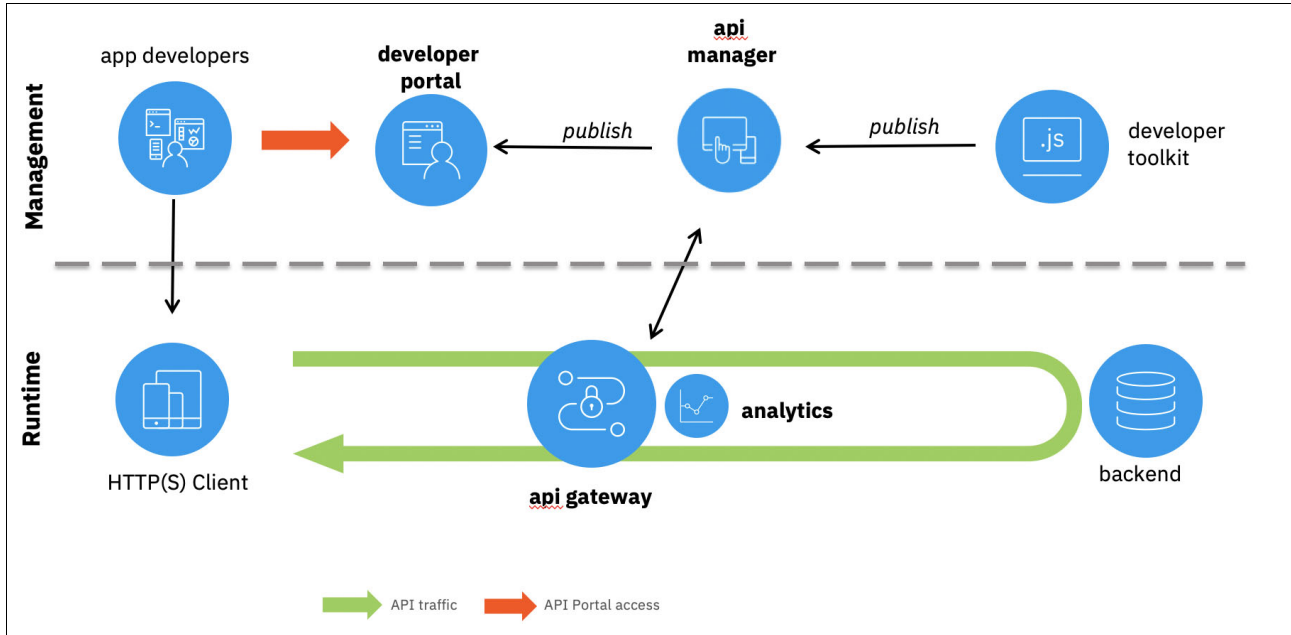


Figure 8-8 API Connect components - a runtime view

Quorum

Next let's discuss the Quorum concept. This is a well-established model for ensuring that we can cater for the loss of a node within a cluster without losing the integrity of the cluster altogether. Without a methodology like Quorum, distributed applications like APIC can suffer *split brain*. Split brain describes the scenario when the communication breaks down between subsets of nodes in a cluster. As a result, it becomes impossible for each of the subsets to know whether the other subsets are still running. As such there is a danger that multiple subsets might believe they are the master of key elements of cluster-related state. Were more than one subset to change that state, this typically would result in inconsistencies that would be impossibly hard to reconcile. Quorum defines the number of operational nodes in a cluster that are required to perform cluster level operations, thus ensuring there is only ever one master of the state.

API Connect when deployed in containerized runtime uses Quorum to provide high availability, and yet also retain data consistency in relation to configuration and other state required by the topology.

Quorum can be defined as $Q = n - \text{floor}((n - 1)/2)$ where:

- ▶ Q is the count of nodes required to maintain quorum,
- ▶ n is the total number of nodes and floor function rounds down the number to its integer. For example, 4.7 becomes 4 after rounding.

If there is a cluster with four nodes, then the quorum requirements would be as follows:

$$\begin{aligned}
 Q &= 4 - \text{floor}(3/2) \\
 &= 4 - \text{floor}(1.5) \\
 &= 4 - 1 \\
 &= 3
 \end{aligned}$$

This implies that for a four-member cluster we can lose only one node and with the second node failure, we lose quorum.

Five-member cluster would again need three nodes as in the preceding formula. In other words, it can lose up to two nodes (40% fault tolerance) without losing quorum while a four-node cluster can lose only one node (25% fault tolerance).

It can be noticed that the formula is favorable for odd number of members. So, it is advisable to go for odd number of members of nodes as they increase level of fault tolerance. Figure 8-9 shows the pictorial representation of a node failure and its impact on quorum.

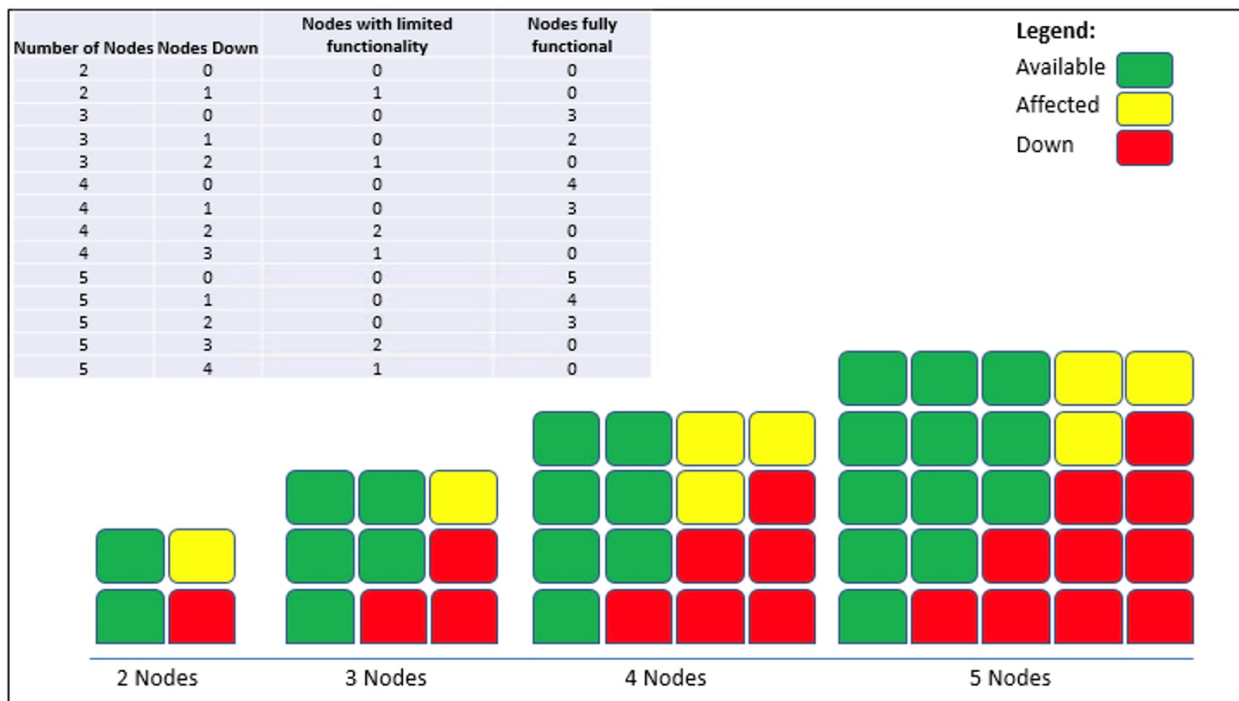


Figure 8-9 Node failure and its effect on Quorum

As it can be seen in Figure 8-9, high availability can't be implemented in less than a three-node setup.

What happens to API Connect components when quorum is lost?

The API Connect components are backed by different data store technologies and the behavior of the component is driven by their data store.

- ▶ Management system continues to serve traffic and supports read transactions but will not allow any changes to configuration, for example publication of new APIs, adding new users, creation of applications.
- ▶ Developer Portal would be unavailable preventing consumers from viewing and subscribing to the APIs.
- ▶ Analytics would allow viewing existing data but new data won't be ingested.

- ▶ Gateway Server would continue to process existing API traffic but will not be able to support:
 - API publications
 - Onboarding new applications or subscriptions
 - Rate limiting (quota enforcement)
 - Token revocation

Now that we have described API Connect components and the quorum concepts, let's cover few sample topologies.

Sample topologies

We will discuss some common topologies and assess their high availability capability.

Single cluster, single location setup

Figure 8-10 shows the single cluster, single location setup.

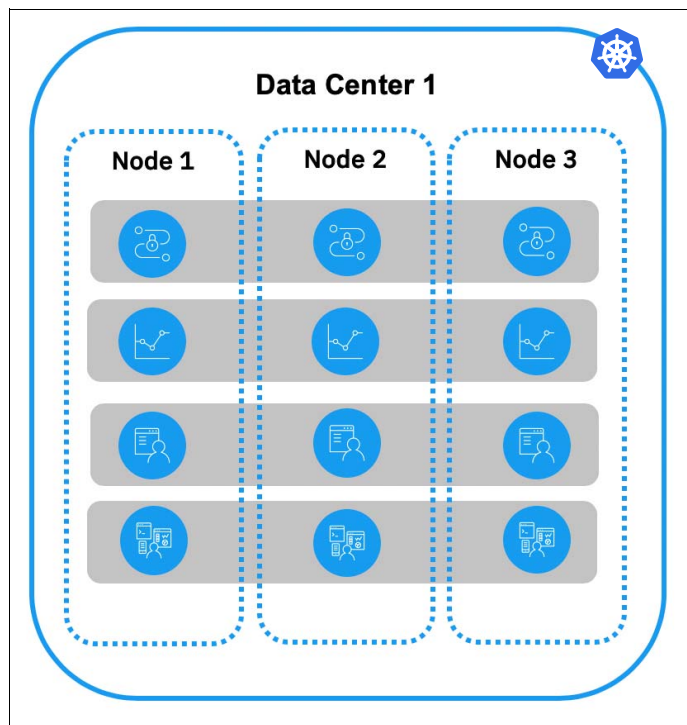


Figure 8-10 Single cluster, single location topology

This is the simplest API Connect high availability topology. Assuming that the components are placed as shown in Figure 8-10, let's evaluate this solution against common failures:

- ▶ **Node failure:** Single node failure will not have any impact, but if two nodes fail then quorum would be lost and the components will behave as described earlier.
- ▶ **Data center failure:** Application will not be available.

Advantages of single cluster setup are as follows:

- ▶ Simple single Kubernetes cluster setup.

Concerns of single cluster setup are as follows:

- ▶ Latency of < 30 ms is required between the nodes.

- ▶ Single data center failure will cause complete outage.

Two-cluster, two-location, Active-Passive setup

Let’s now discuss the topology that has two Kubernetes clusters at two locations as shown in Figure 8-11.

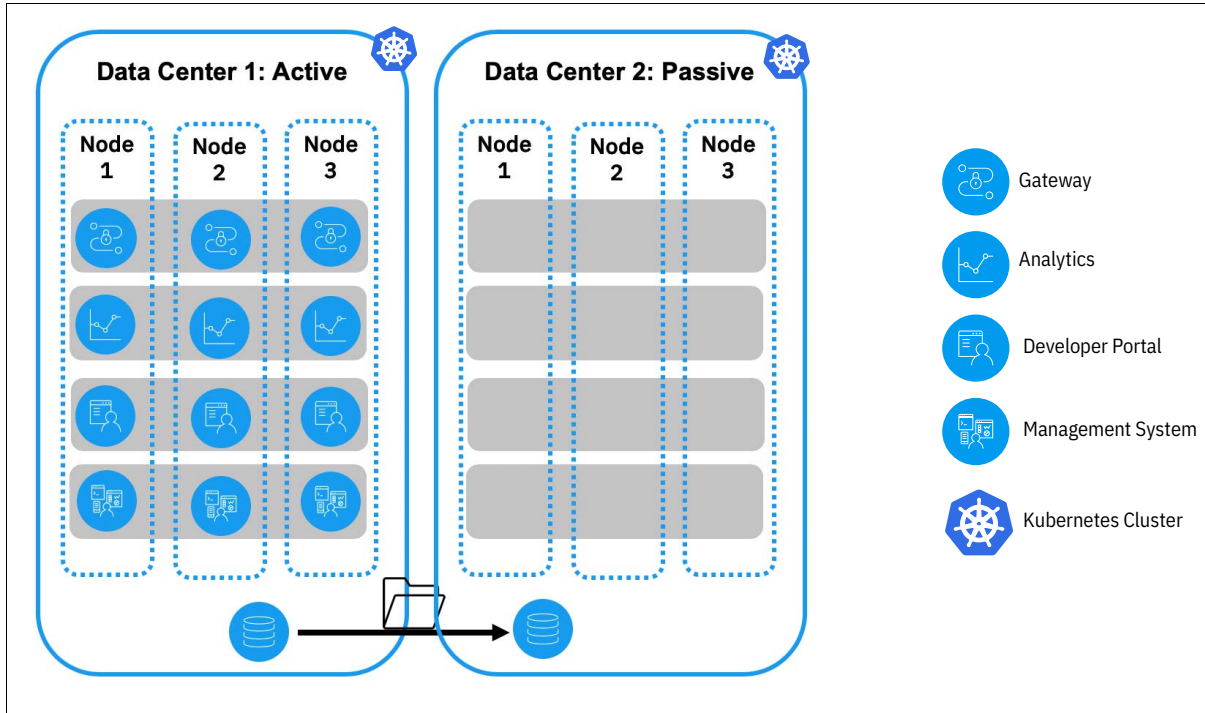


Figure 8-11 Two cluster Active-Passive, two location topology

To overcome the single data center failure point, we can plan for two data center deployments, each having its own Kubernetes cluster and in active passive mode. The secondary data center would have to be kept preconfigured and ready to have instances of the services that are deployed through regular data sync up. In case of a primary data center failure, a secondary data center would be built with the last primary backup. This means that the customer would have to wait while the secondary data center is being enabled and only then can normal operations resume.

Let’s evaluate this solution against common failures:

- ▶ **Node failure:** Single-node failure will not have any impact, but if two nodes fail then quorum would be lost, and the components will behave as described earlier.
- ▶ **Data center failure:** In the event of primary data center failure, normal operations will be impacted until the secondary data center is brought up back online. The backup and recovery should be designed to meet recovery time and recovery point objectives of the organization.

The following are the advantages of this setup:

- ▶ Simple Kubernetes cluster setup at each site.
- ▶ It can survive single data center outage.

The following are the challenges of this setup:

- ▶ Latency of < 30 ms is required between the nodes within the same data center.

- ▶ Regular backup and syncing between the data centers is required.
- ▶ Backup and recovery processes should meet the Recovery Point Objective for data center failures.
- ▶ Unused spare capacity.

Single cluster, three location setup

Figure 8-12 shows the single cluster, three location setup.

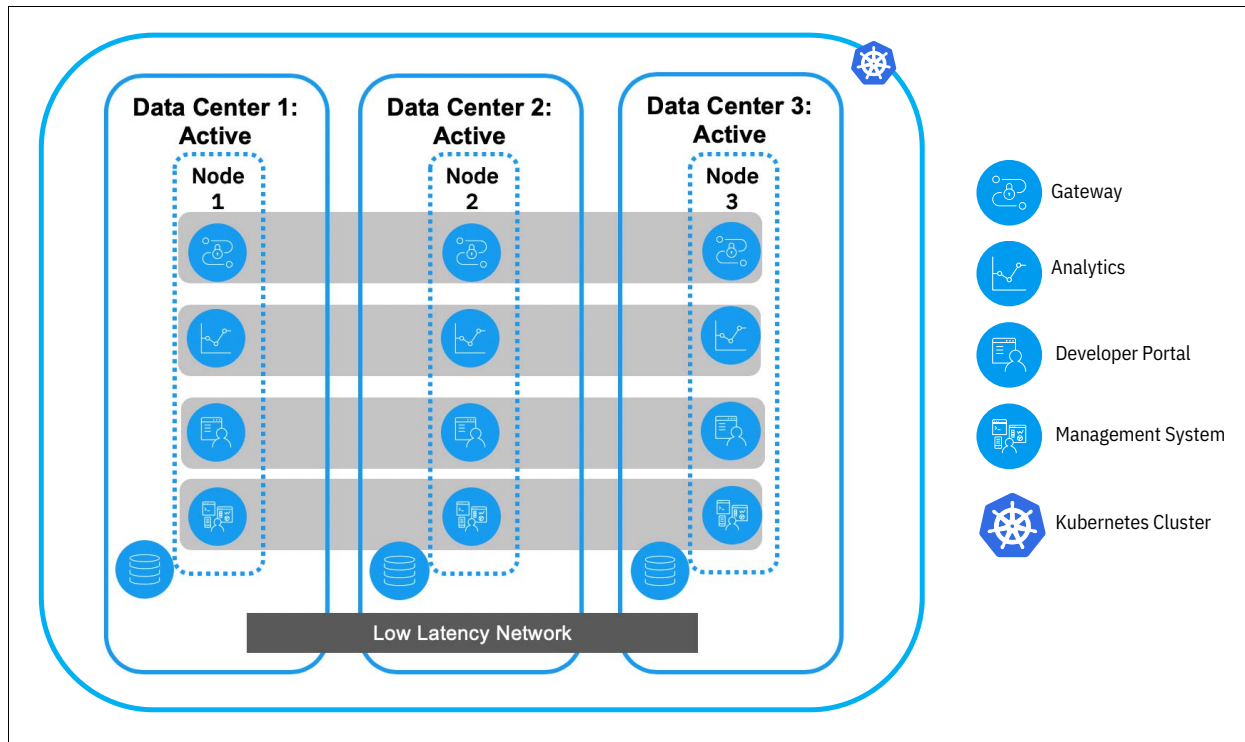


Figure 8-12 Single cluster, three location topology

This setup expands on the first suggested topology by keeping the three nodes of the single Kubernetes cluster in different location to avoid single data center outage scenario. Let's see how this solution fares against.

- ▶ **Node failure:** Single node failure will not have any impact, but if two nodes fail then quorum would be lost.
- ▶ **Data center failure:** Like node failure, single data center failure will not have any impact, but if two data centers fail then quorum would be lost.

The following are the advantages of this setup:

- ▶ It can survive data center outage.
- ▶ It is IBM recommended topology from availability perspective.

The following are the concerns for this setup:

- ▶ Latency of < 30 ms is required between the data centers.

We covered few deployment scenarios and evaluated its high availability. It is beyond scope of this book to discuss the all the variations, but the principles discussed earlier can be extended to evaluate any topology.

8.6.2 Scalability of API Connect in containerized environment

API adoption rates are sometimes unpredictable and can lead to mismatch in the required versus the available infrastructure, so scalability is important. API Connect components can be scaled up or down independently as per customer requirement.

Note: The scalability that is discussed in this section applies to the scaling of the API management exposure layer and not to the implementation layer, which needs to be scaled independently.

APIC scalability is based on Kubernetes and provides operational consistency through standardized Kubernetes commands.

APIC can be scaled by using:

- ▶ Manual scaling through commands
- ▶ Horizontal Pod Autoscaler

Manual scaling

Manual scaling can be done in two ways:

- ▶ **The required API Connect component can be scaled by using its StatefulSet.**

Use the following command to find the statefulset names:

```
kubectl get statefulset -n <namespace> | grep gateway-service
```

Then, use the following command to scale in or out with the number of replicas that you require:

```
kubectl scale statefulset <StatefulSet name from above> --replicas=<desired no of replicas> -n <namespace>
```

For example,

```
kubectl scale statefulset r62bf86f4e0-dynamic-gateway-service --replicas=3 -n apic
```

- ▶ **This can also be achieved by using the console.**

As an example see Figure 8-13 and Figure 8-14 on page 561.

Name	Namespace	Desired	Current	Created
r9ad2183d2-dynamic-gateway-service	apic	2	2	12 days ago
r307b84ffe1-analytics-storage-data	apic	1	1	
r307b84ffe1-analytics-storage-master	apic	1	1	
rccb357bd8b-apic-portal-db	apic	1	1	12 days ago

Figure 8-13 Scaling a stateful set (i)

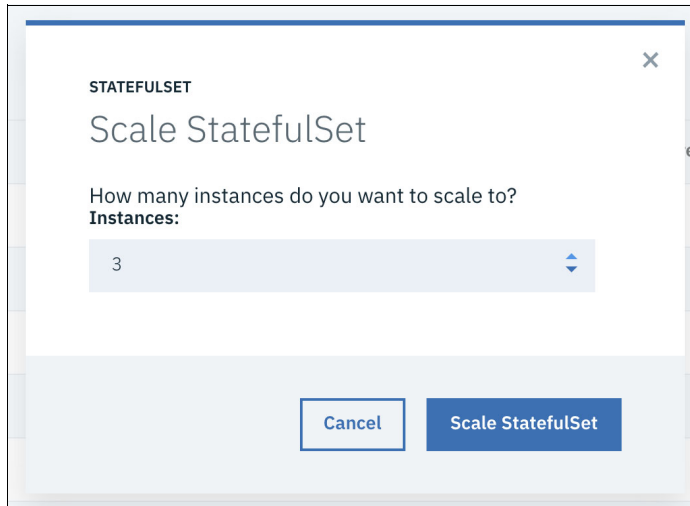


Figure 8-14 Scaling a stateful set (ii)

Horizontal Pod Autoscaler

Horizontal Pod Autoscaler (HPA) can be used to monitor the resources and scale the replicas automatically.

API Connect provides inbuilt policies that can be used for scaling. They can also be defined to suit project needs. For example, the YAML as shown in Example 8-1, creates an HPA, which spins an additional instance of gateway if the CPU usage goes beyond 5%.

Example 8-1 YAML example

```

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: gateway-hpa
  namespace: apic
spec:
  maxReplicas: 3
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: rf9ad2183d2-dynamic-gateway-service
  targetCPUUtilizationPercentage: 5

```

Figure 8-15 shows how to create custom scaling policy.

Scaling Policies						apic
Name	Namespace	Reference	CPU%	Min Replicas	Max Replicas	
gw-hpa	apic	rf9ad2183d2-dynamic-gateway-service	5	2	3	

Figure 8-15 Scaling policy

This policy will spin a new instance of gateway when the CPU usage goes beyond 5%.

Existing scaling policies can also be modified through console to scale a component. See Figure 8-16.

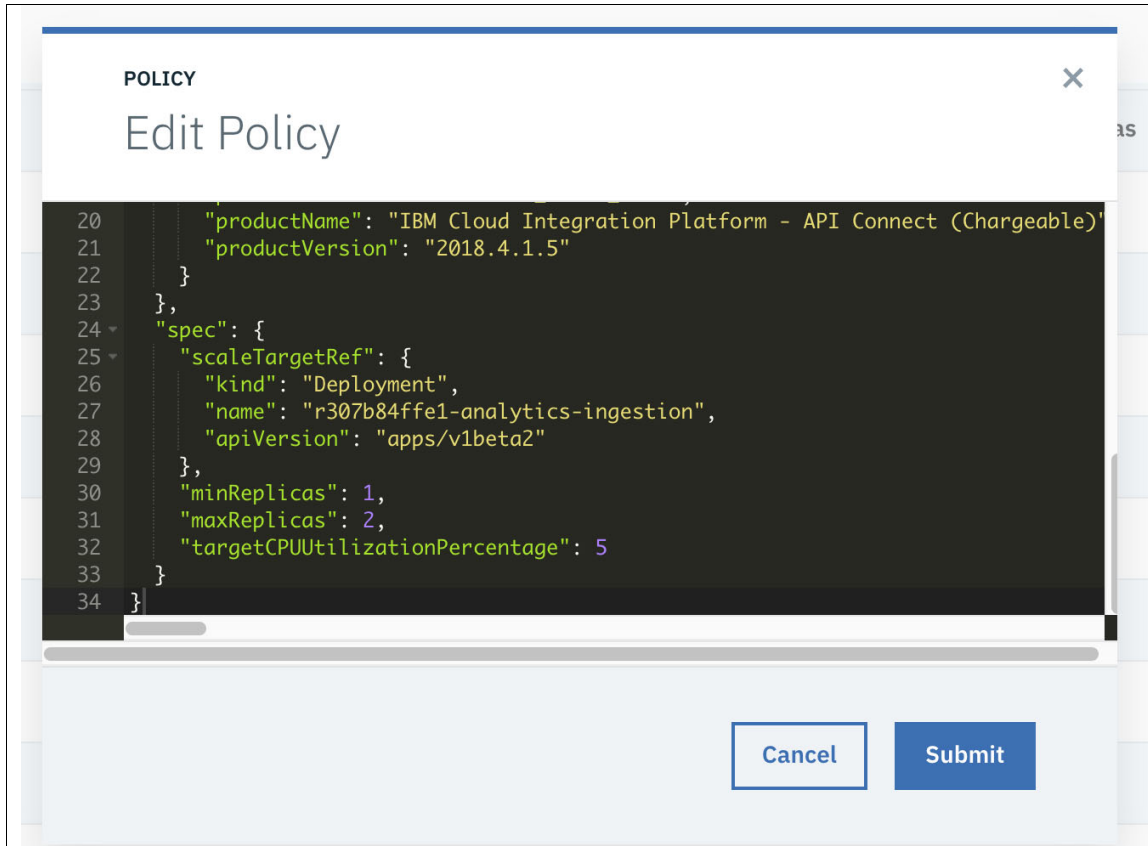


Figure 8-16 Edit scaling policy

Figure 8-17 on page 562 is a screen capture of scaling due to the preceding policy, as seen in logs.

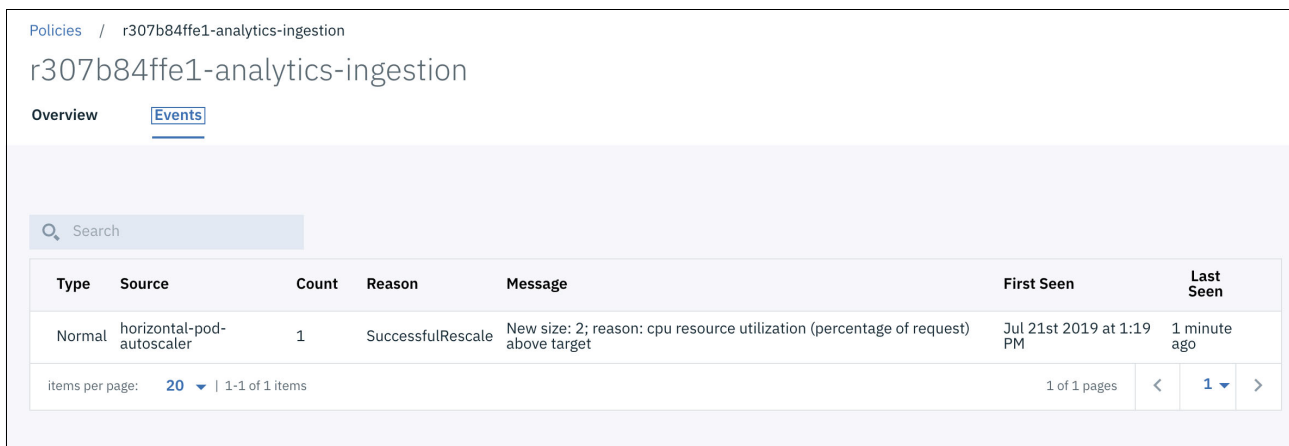


Figure 8-17 Scaling logs

The scaling can also be verified by checking the pods as shown in Figure 8-18.


```

user12:~$ kubectl get pods -n apic -o wide | grep analytics-ingestion
r307b84ffe1-analytics-ingestion-885b6ccc-9sctd      1/1      Running   1      12d      10.1.160.69      10.135.148
r307b84ffe1-analytics-ingestion-885b6ccc-trq4l     1/1      Running   0      29m      10.1.145.234     10.135.148
user12:~$

```

Figure 8-18 Component after scaling up

Scaling using APICUP

apicup is the installer for API Connect on containers. In order to scale out with it, `apiconnect-up.yaml` (found under `apicinstall` folder) can be used. Run the following command to increase the replica count in the config.

```
apicup subsys set <subcomponent name> replica-count=<desired no of replicas>
```

Then deploy the updated config by using,

```
apicup subsys install <subcomponent name>
```

For example, with debug logs

```
apicup subsys install mgmt --out mgmt-out --debug
```

Important: High-availability and scalability can be addressed only in a holistic way across the application landscape. API Connect resilience would be meaningless if other components of the solution are not equally resilient, such as back-end microservices API, ancillary applications, and databases.

8.6.3 Conclusion

In this section, we discussed high availability and scalability concepts as applicable to APIC and then also discussed few sample deployment topologies. We also described how API Connect can be scaled both manually and automatically.

The following are additional references:

- ▶ APIC white paper 1.0.8 <https://www.ibm.com/downloads/cas/30YERA2R>
- ▶ Link to APIC2018 Infocenter:
https://www.ibm.com/support/knowledgecenter/en/SSMNED_2018/mapfiles/getting_started.html

8.7 IBM API Connect API Test Pyramid

The *Practical Test Pyramid* is a common methodology for implementing a testing strategy. In this section, we consider how this applies to an IBM API Connect solution.

8.7.1 Practical Test Pyramid

The *Practical Test Pyramid*, developed by Mike Cohn¹, describes initiating lots of quick, low-level, *cheap tests* on the smallest possible unit to give greater confidence that end-to-end flows will work. Whereas more expensive, time consuming, and disruptive test cases should be fewer in number and demonstrate connectivity with other systems or demonstrate that all unit tests work together.

¹ Mike Cohn, *Succeeding with Agile*, Addison-Wesley Professional, 2009

In Martin Fowler's publication, *The Practical Test Pyramid* (<https://martinfowler.com/bliki/TestPyramid.html>), he talks about the categorization of the tests. He says that the words used to describe them can become conflated and detract from the general aim of creating systems that work. In our experience, the following usage of the pyramid best applies to an API Test Pyramid (see Figure 8-19 on page 564).

- **Unit Test:** A unit test denotes the smallest unit of API testing in a request being sent and a response received. Both must match service requirements.
- **Integration Test:** Often known as the *Component Test*, the use of integration as the test keyword shows that these tests should focus on APIs calling independent resources such as a database.
- **Consumer Test:** APIs are created by providers for the use of consumers. The consumer has an interaction with the whole system that includes any user interface, thus consumer testing applies to full end to end tests.

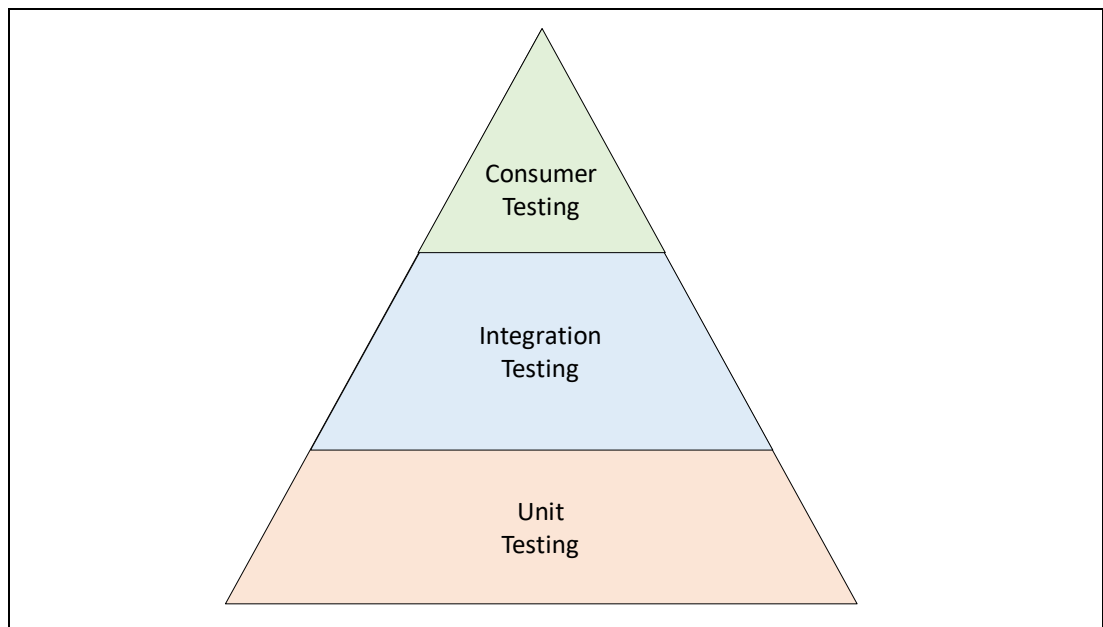


Figure 8-19 The test pyramid in the context of IBM API Connect

8.7.2 Requirements

It should not be left to the implementer of the API to dream up tests from thin air. They must be based on business requirements. Without this, the whole basis of testing is undermined.

Before any API project begins, there should be a clear description of the expected outcomes. What the project is trying to achieve and what is the minimum request and response that must be delivered in order for the API calls to be successful? These expected outcomes are the requirements of the project, without which no tests can be written and thus no APIs can be written.

Requirements state explicitly what is and is not permitted which allows both positive and negative testing to be completed and might be based on industry and enterprise standards and specifications.

Tests that are written without requirements will either use existing code and implementation as a source of truth or will rely on the developers view of what should be occurring. It is also

possible for these tests to be disputed, because there is no individual source of truth for what the implementation should be doing to meet business needs.

8.7.3 Test types

In this section, we explore how different test stages in the API Test Pyramid apply to an IBM API Connect solution.

Unit tests

The primary question when discussing Unit Testing is what makes up the smallest, testable, self-contained component, as derived from the requirements. A unit test for an API should *not* call a back-end service. It should *not* rely on making requests from outside the API Connect domain and should be executable in any simple IBM API Connect environment.

The smallest testable unit for API Connect APIs consist of the four components that make an API:

- ▶ Its address
- ▶ Its operation
- ▶ Request data
- ▶ Response data

An API has a primary data representation that is called a resource, which is the conceptual mapping to a set of entities. The *address* of the resource is displayed as a *Uniform Resource Identifier (URI)* which is crucial to making an API call.

Addresses will have tests such as;

- ▶ Does an exposed address return a correct result?
- ▶ Are unspecified addresses correctly handled?
- ▶ Are misspelled addresses correctly handled?

An *operation* is the method of a REST API that determines what action to perform on the resource. The four main HTTP verbs that define the operation are; GET (read), PUT (update), POST (create), and DELETE (delete).

Operation Unit Tests make requests to valid addresses to ensure that the API exposes only the verbs that are outlined in the requirement. This test is simply about operation exposure.

Simple security tests might also be applicable here like the absence of tokens, client IDs, or certificates for secure endpoints. But they are unlikely to include more advanced tests such as OAuth, which would be more applicable in integration tests as multiple calls are needed.

Request data normally refers to a list of headers and the body of data that is passed with an operation to an address. Tests at this level focus on whether the correct data is provided, the correct headers are used, and that incorrect headers or malformed data are handled correctly.

Response data normally refers to how the resource call will respond to the request. The data usually includes a status code, headers, and a body. Tests at this level aim to ensure that valid and expected output is received when certain requests are passed into the system. Additional tests will be used for the handling of both successful and failure scenarios.

The simplest unit test topology is shown in Figure 8-20 on page 566.

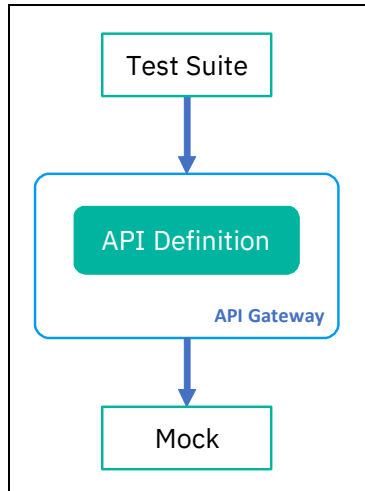


Figure 8-20 Unit test topology

Unit testing GatewayScript fragments

From the API Connect perspective, this is the lowest logical unit. Deeper testing has little benefit. While lower-level testing within the API of specific fragments of gateway code logic might provide some assurance, having quickly deployable unit tests that implicitly exercise that same gateway code can give the same level of confidence. Testing the inner workings of the API is irrelevant if the correct response is achieved.

Moreover, lower-level testing adds extra overhead in order to implement a unit testing codebase and runtime environment, which might inadvisably lead developers to test outside of a DataPower runtime. It is important to note that gateway programming model is security that is hardened and enriched with DataPower-specific functions where you access and manipulate the variables in the API context during execution. As such, any attempt to test fragments of GatewayScript in an independent runtime is likely to be invalid.

Unit testing of APIs in API Connect should be concerned with the unit only, which in IBM API Connect is processing of a request and the expected response.

Ideally, you do unit tests on a smaller runtime environment (such as using Docker image for DataPower). That way, the tests can be isolated to a single change, which would help to build consistent pipeline deployments. Using a shared environment brings operational challenges.

Unit tests should not connect to real back-end systems, because they must be able to run independently. If you connect to real backends for unit tests, you increase the runtime dependencies of the unit tests.

Integration tests

After you confirm that the API's work in isolation, you can perform integration tests. These tests can confirm whether connectivity with other services works and that the results are handled and remain valid for the requirements.

The concept of *Contract Testing* is discussed in *The Practical Test Pyramid* and is applicable to API Connect API testing as well. Connecting services have similar request/response testing as performed for APIs.

By mapping expected requests to expected responses, each integrated service has a contract of integration. If the way that the API behaves changes, the providing service should notify the consumers as part of the release.

Integration tests look to make calls to connected systems such as a database, microservice, or load-balancing service. Each connection should be handled independently without multiple linked tests across several systems and services. This might mean exposing unit test environments to other systems so they can perform isolated integration tests between the IBM API Connect layer and the singular service it is calling.

These tests should try to call services that have no dependencies on other services. While the API Connect layer has no control of the services they call, it is better to call a service whose own back-end service is mocked for these tests. That way, the integration is affected by only the integrated services.

Sample integration test topologies are shown in Figure 8-21.

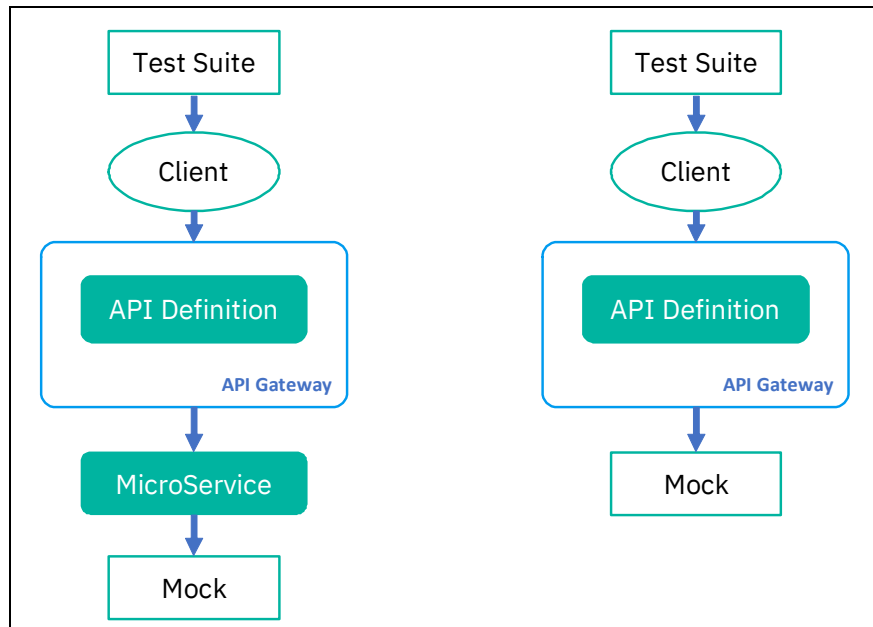


Figure 8-21 Integration test topologies

Testing of integration in API Connect is the one-to-one communication of the service immediately before or immediately after the API unit.

Consumer tests

Also known as end-to-end testing, these tests reach each service of the system as a full flow in the same way that consumers do. Unit testing has already covered a wider API base and integration tests that are already connected to each connected service. So, you should limit consumer testing to only what you expect the API consumer to complete.

At this stage, consider independent users who do not know how the API works internally and are not directly involved in development or lower-level testing of the component parts. This type of test case uses the system like an API consumer does, to see if there are issues outside of the current test scenarios. This concept is known as *Exploratory testing*.

Consumer tests might also include application creation, API subscription, and API test call through user interfaces. This should include testing of how the system is used by API consumers. The goal can be to ensure that the themes, buttons, and selection boxes all work as intended in a process known as *User Interface (UI) testing*.

While UI tests cover a lot of the usability tests, a thorough end-to-end testing should be conducted. This looks at user journeys through the system. The tasks include creating an

application and subscribing APIs, logging in to the system, and performing a series of linked API calls such as an Oauth2 flow.

The consumer test topology is depicted in the Figure 8-22.

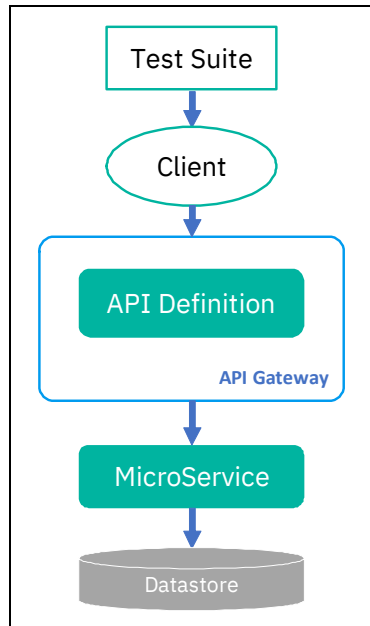


Figure 8-22 Consumer test topology

Testing at the consumer level looks at the full end to end journey experienced by the consumer. Each flow should behave as the consumer expects including API responses, user interfaces that might be required and redirection to other services.

8.7.4 Automated testing

It is possible to perform automated testing at each layer of the pyramid with initial testing on as updates to the API definition are pushed to a code repository. Build tests might include:

- ▶ YAML validation tests
- ▶ JavaScript validation
- ▶ JavaScript Linting - checks against a custom set of specifications such as line length
- ▶ API deployment
- ▶ Test application subscriptions
- ▶ Complete run of unit tests
- ▶ Complete run of integration tests
- ▶ Automatable consumer tests

Following successful completion of the test suite in one environment (for example development) the artifact can be made ready for combined testing in integration tests. In those tests potential changes to multiple APIs can be tested at the integration and consumer level by running the automated tests against the System Integration Environments.

All non-breaking APIs will be pushed (in an automated way) to non-functional tests. At this stage, they once again follow the automated test procedures before the APIs failing performance tests are removed from the release.

The remaining changes are pushed through into User Acceptance Testing (UAT) environment. At this stage, you also complete the automated tests, but with additional manual exploratory test scenarios. And you carry out these tests in the most production-like environment possible, before the same process proceeds into production.

The high-level process diagram of automated testing is shown in Figure 8-23 on page 569.

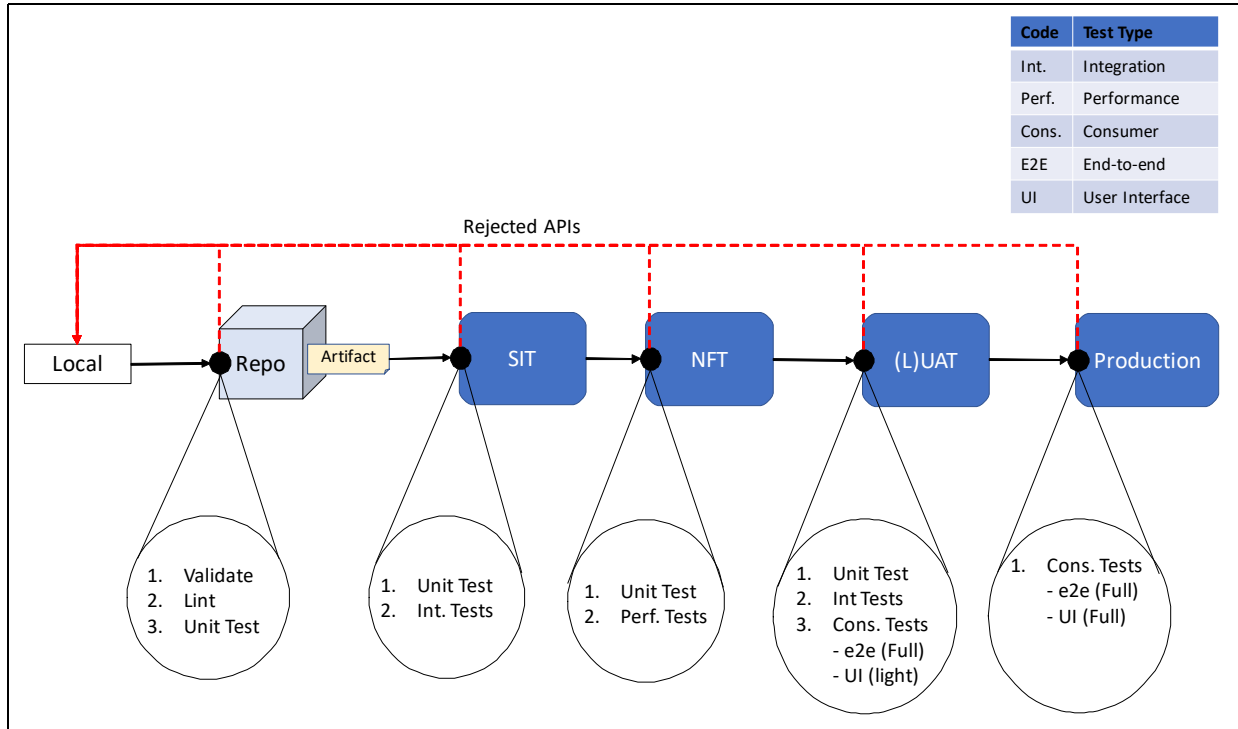


Figure 8-23 Automated testing

8.7.5 Conclusion

In this section, we have explored how The Practical Test Pyramid can be used in the context of IBM API Connect. An overview has been given on the three types of testing: unit testing, integration testing, and consumer testing.

It is important to consider what makes up the independent *unit* of an API, which we have described as a request and response that use the IBM API Connect and IBM DataPower Gateway in isolation.

Additional stages such as the integration testing and consumer testing are simply names that best describe the actions that apply in this context. Descriptions for them follow the general rules that are defined in the *Practical Test Pyramid*.



Field notes on modernization for messaging

This chapter explores the typical stages that customers go through as they progressively modernize their IBM MQ environment. You learn how best to implement IBM MQ in a container orchestration platform in order to leverage its inherent availability capabilities. We also discuss IBM MQ scaling and automation of IBM MQ provisioning.

This chapter has the following sections:

- ▶ Modernizing your messaging topology with containers
- ▶ IBM MQ availability
- ▶ IBM MQ scaling
- ▶ Automation of IBM MQ provisioning using a DevOps pipeline

9.1 Modernizing your messaging topology with containers

IBM MQ first provided support for containers in 2015, and rapidly embraced container orchestration technologies such as Kubernetes and OpenShift to allow managed production grade deployments. IBM MQ has continuously innovated over the years such that today it is just as at home in traditional deployments as it is in cloud native environments. Runtimes start-up in seconds and are arguably the most performant on the market. Container orchestration provides high availability as standard. When this is combined with cloud native features recently added to IBM MQ such as *uniform clusters*, scaling is simple and natural within a container environment.

However, as we made clear in Chapter 4, “Cloud-native concepts and technology” on page 85, there is much more to moving to containers than simply a change in infrastructure. For each technical area there are different subtleties to be considered as we move towards containerization. In this section we explore the typical stages that customers go through as they progressively modernize their IBM MQ environment.

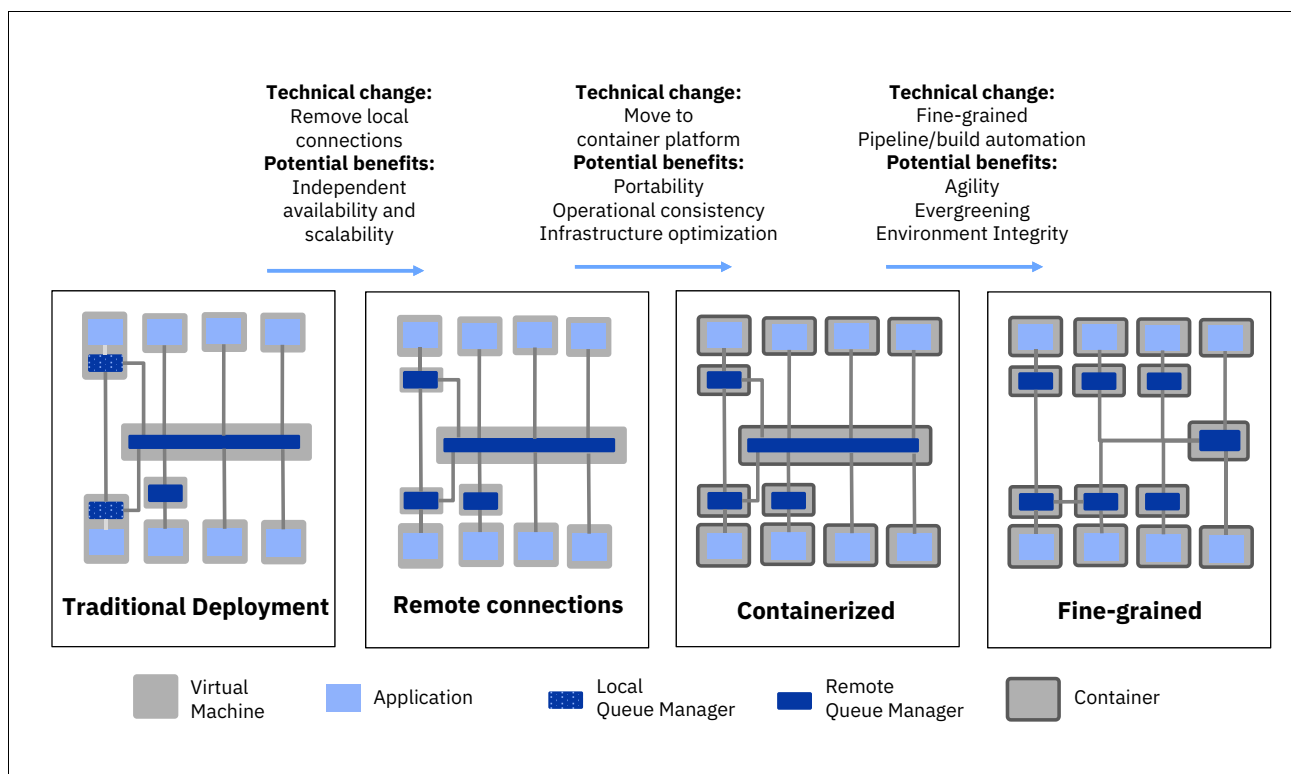


Figure 9-1 Modernizing an IBM MQ infrastructure using containers across three discrete stages

As summarized in Figure 9-1 this section describes how an IBM MQ estate can be modernized by using containers across three discrete stages.

- ▶ **Remove local connections:** Ensuring that IBM MQ is running separately of the applications it serves such that its availability and scaling is independent.
- ▶ **Containerized queue managers:** Move the queue managers into containers to gain greater standardization, and consistency on an operational level, especially if applications are also moving to containers.
- ▶ **Fine grained queue managers:** Reducing the dependencies between sets of queues by separating them into discrete queue managers and potentially allowing application teams greater ownership.

It should be noted up front that while these are common progressions based on real customers, of course every customer's environment is different, and it might make sense for you to do things in a different order to we depict here. As such it is more important to understand the reasons behind each stage than to be too focused on the order.

We begin by introducing a typical IBM MQ infrastructure, then walk through each of the preceding stages to see how it evolves.

9.1.1 Typical existing topology

Figure 9-2 on page 573 is an illustration of a typical current IBM MQ infrastructure:

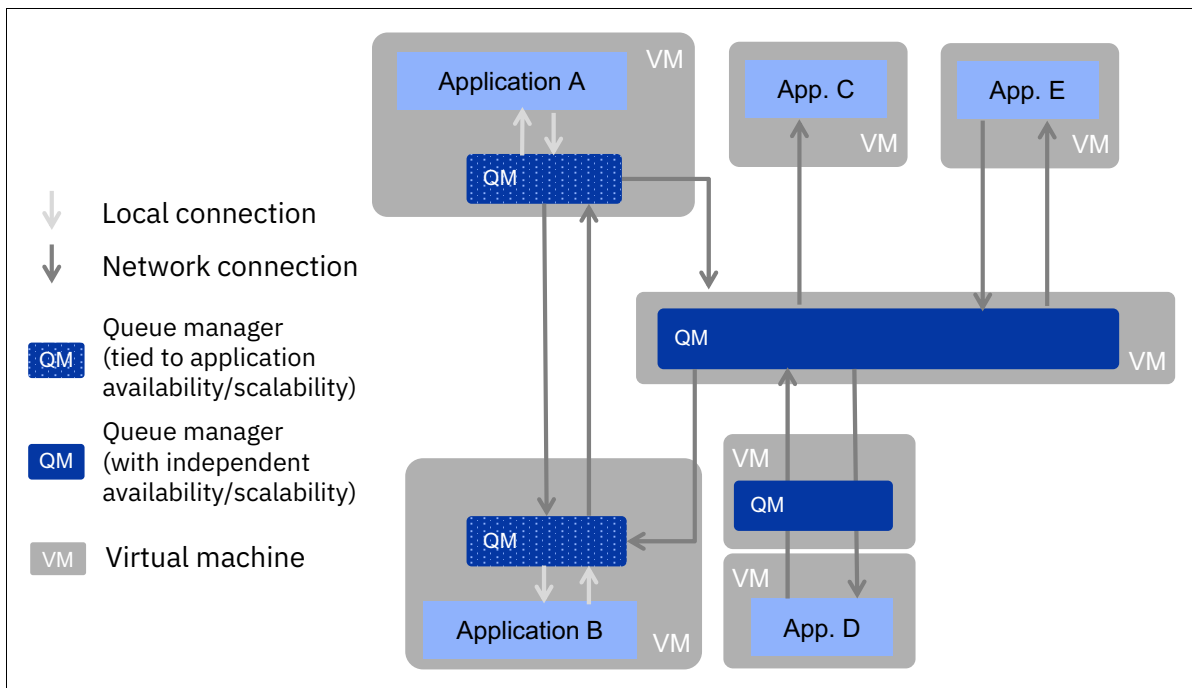


Figure 9-2 Typical traditional IBM MQ environment

An existing IBM MQ estate might include many queue managers as depicted in Figure 9-2. Some installed locally alongside applications (for example Applications A and B), some separate, but logically associated with applications (as shown with application D), and also central IBM MQ instances. Although deployed differently, often all of the IBM MQ instances are controlled by a central IBM MQ team.

The existing IBM MQ estate will include different types of connectivity between the applications and IBM MQ. IBM MQ supports two options for connecting to a queue manager:

- ▶ **Local connections:** Applications that have a locally installed queue manager on the same operating instance can connect by using local inter-process communication. This is called a *server binding*.
- ▶ **Remote connections:** Applications running on a separate operating system can communicate remotely over a network connection. In the IBM MQ documentation this is called a *client binding*, but in this document we will use the product-agnostic term *remote*.

Tip: Technically this binding can also be used to connect to a local queue manager, but as you would expect, the connection traverses the network.

The locally installed instances might be there for a specific reason such as to enable coordination of global transactions, or to enable them to function even when they are disconnected from the network. However, in some cases it will simply be for historical reasons. In the past, due to less reliable networking, locally installed IBM MQ was the norm. For early versions of IBM App Connect (up to what was then called IBM Integration Bus v10), a local IBM MQ server was a hard dependency on installation. Interestingly, the administration of these local instances might still be performed by the central IBM MQ administration team, due to the specialized knowledge that is required.

As networks became trustworthy, more recent applications would be more likely to use a remote queue manager such that it could be administered separately more easily. Applications might still have a dedicated instance of IBM MQ per application, such as for Application D. This might be due to the particular requirements that the application team had for isolation, performance, or security for example. See Figure 9-3.

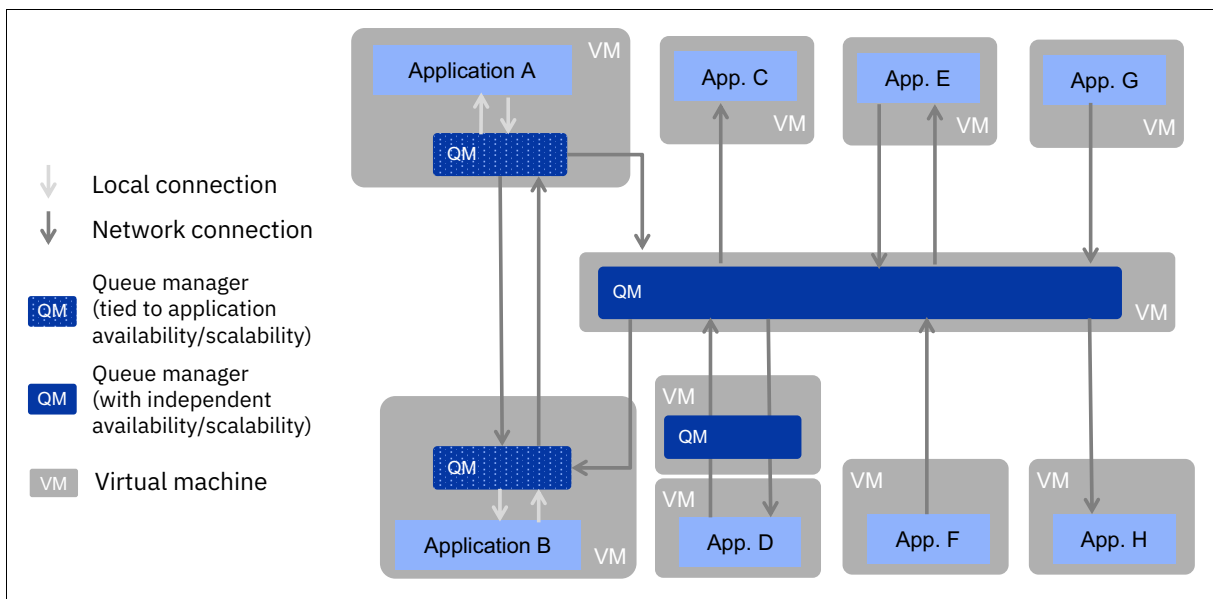


Figure 9-3 Adding queues to the shared queue manager as we introduce more applications

However, as the number of applications increases, we often see shared queue managers that host queues for multiple applications, to provide a perceived simplification in administration.

In addition, some or all of these instances might not be included in an IBM MQ cluster. However, for simplicity we will leave clustering until later when we discuss availability, scaling, and routing.

9.1.2 Removing local connections

As noted, the need for local connections (server bindings) has diminished, as the robustness of remote connections has increased dramatically. Removing the need for a local IBM MQ installation has many advantages. It is one less thing for the application team to deal with when configuring their application. Furthermore, they no longer need to concern themselves with how to scale IBM MQ and the application together. See Figure 9-4.

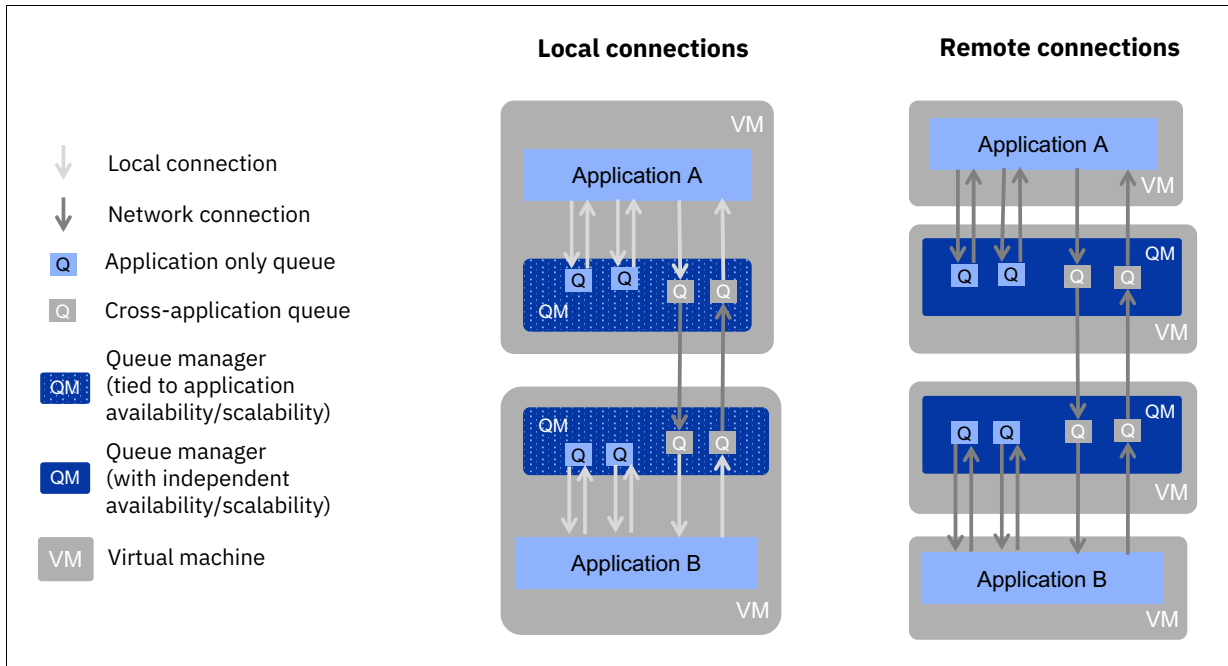


Figure 9-4 Removing local connections, and moving local queue managers to separate VMs

This also turns out to be an excellent first step in the direction of moving to containers, because a preferred practice is that a container runs only one process. This enables containers to be administered more simply by orchestration environments such as Kubernetes. So, having isolated the queue managers from the applications they are in a better position to be containerized.

9.1.3 Containerizing queue managers

The next logical step is to take the existing queue managers and move them each into a container of their own as shown in Figure 9-5 on page 576.

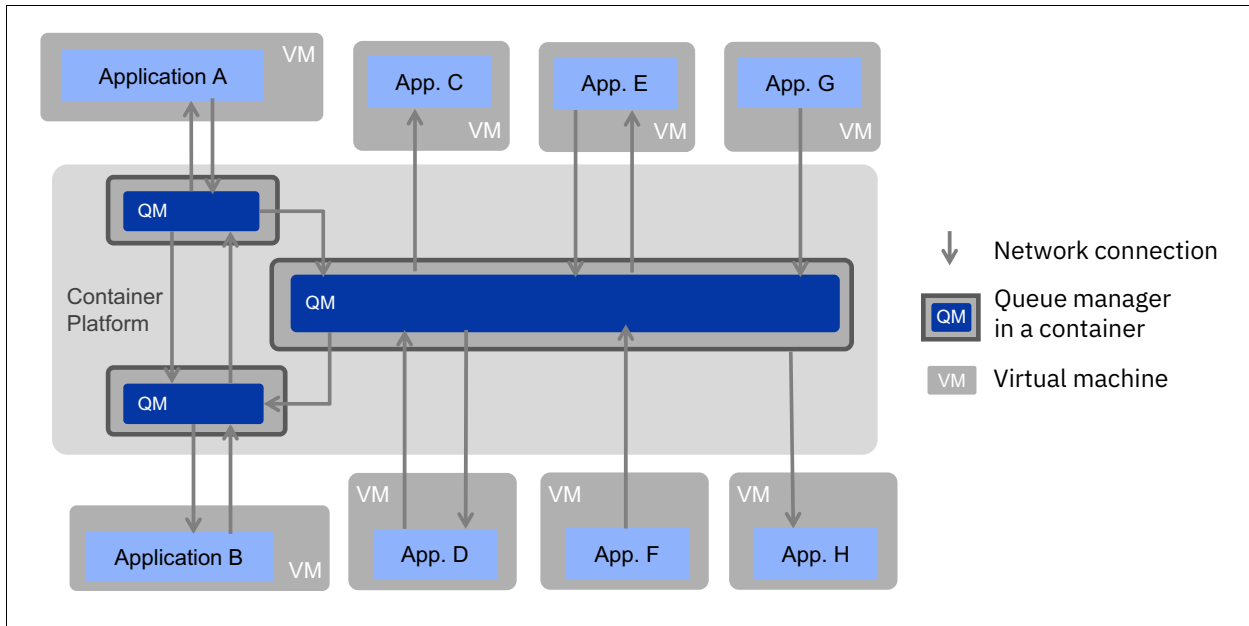


Figure 9-5 Basic, coarse-grained adoption of containers

Figure 9-5 deliberately does not show the detailed configuration of IBM MQ from an availability and scalability viewpoint, as we will discuss the considerations here in more detail later.

9.1.4 Fine-grained queue manager deployment

As noted, many enterprises have evolved to have centralized, shared IBM MQ deployment, where one or more queue managers have been deployed together on highly available infrastructure. This setup is often deliberately created to assure a central enforcement point for the IBM MQ administrators to govern and maintain. Initially this might appear ideal, but there are several issues:

- ▶ If multiple applications use the same queue manager, maintenance must be scheduled across application teams. Dependencies across multiple teams are challenging to arrange and often cause additional effort and delays.
- ▶ If an application uses messaging resources on the same machine as other applications there are limited amounts of isolation that can be provided. For instance, there is no easy mechanism to limit the CPU that one application can use compared to another application.

Applications want control over their IBM MQ resources, and this requires isolation to avoid application teams from affecting each other. For instance, a team might want access to the latest features of IBM MQ, while others do not. The use of a container orchestration platform enables isolated fine-grained IBM MQ resources to be created and managed in a standardized way. This is true, even though they ultimately run in completely separate containers as shown in Figure 9-6 on page 577.

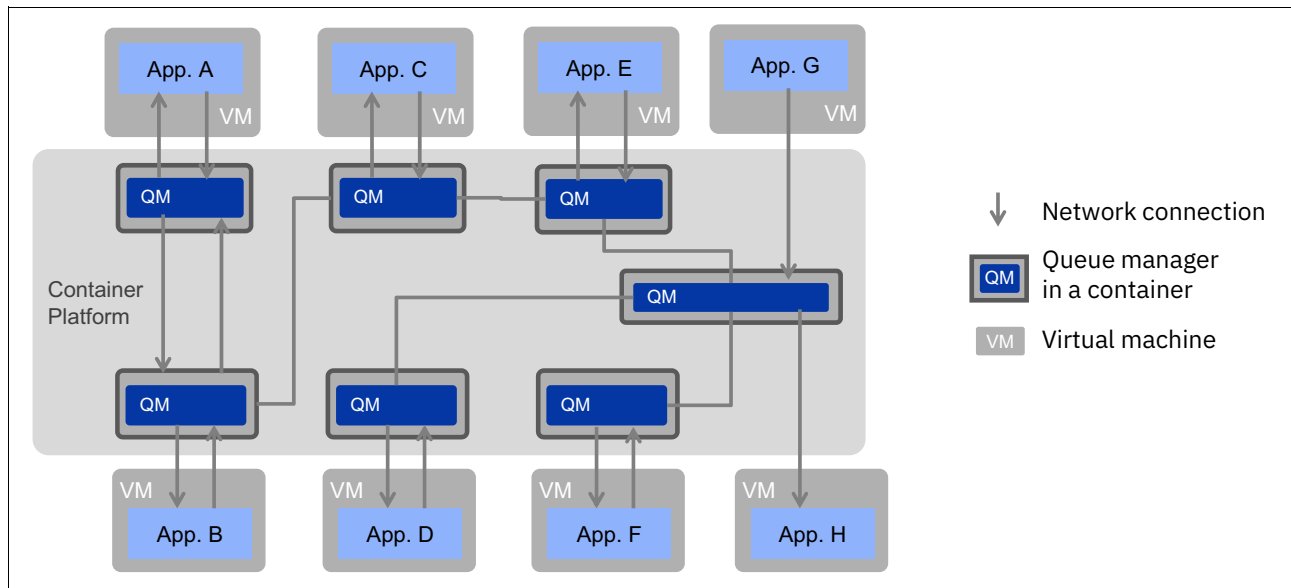


Figure 9-6 Fine-grained queue managers

Fine-grained queue managers mean that individual applications have separate queue managers. This setup overcomes the preceding situation, because isolation containers are provided for each application.

The fine-grained architecture will increase the number of queue managers within the enterprise, which introduces some new considerations.

- ▶ **Disposability:** When you develop cloud-native components, the component instance is designed and built to be disposable and immutable. Traditionally IBM MQ queue managers were treated as build once, and then continuously nurture from that point forward. An IBM MQ queue manager is indeed a stateful application. But that does not mean that an IBM MQ queue manager runtime itself cannot be disposed of, when no longer needed. Likewise, this runtime can also be spun up, on demand. This characteristic is especially useful in testing scenarios. Special considerations are needed for assured delivery scenarios. But overall, there are many cases where a more “disposable” deployment style is appropriate for IBM MQ.
- ▶ **Maintenance:** Having an effective mechanism to centrally manage and automate maintenance, becomes critical or the administration overhead would become overwhelming. We need to provide continuous adoption (see 4.2.11, “Continuous Adoption” on page 98) for the IBM MQ runtime. IBM MQ has two delivery streams: the traditional long-term support (LTS) and the continuous delivery (CD) stream that provides early access to new features. If we want the latest features and want to be up to date on security patches, CD stream would be the ideal choice. The resultant image can then be rolled out across the deployments either automatically or based on a minimal approval step. Ideally, we limit any downtime experienced by the application, by providing multiple logically identical instances of IBM MQ, and allow an application to connect to any. This could use the new “uniform cluster” capability that was mentioned previously, to provide additional capabilities during upgrading and failover scenarios. We look at this in more detail in 9.4, “Automation of IBM MQ provisioning using a DevOps pipeline” on page 594.
- ▶ **Monitoring:** Deploying on a container orchestration platform (such as OpenShift) allows for centralized monitoring across all instances that are deployed to the platform. Cross-component tracing and diagnostics can be achieved for instances of other related components, too.

9.1.5 Decentralization

Modernizing the people and process is critical to assuring a successful modernization. Typically IBM MQ has been the responsibility of a central IT team. They built, configured, and maintained the MQ infrastructure on behalf of the application teams — a highly centralized ownership model that is shown in Figure 9-7.

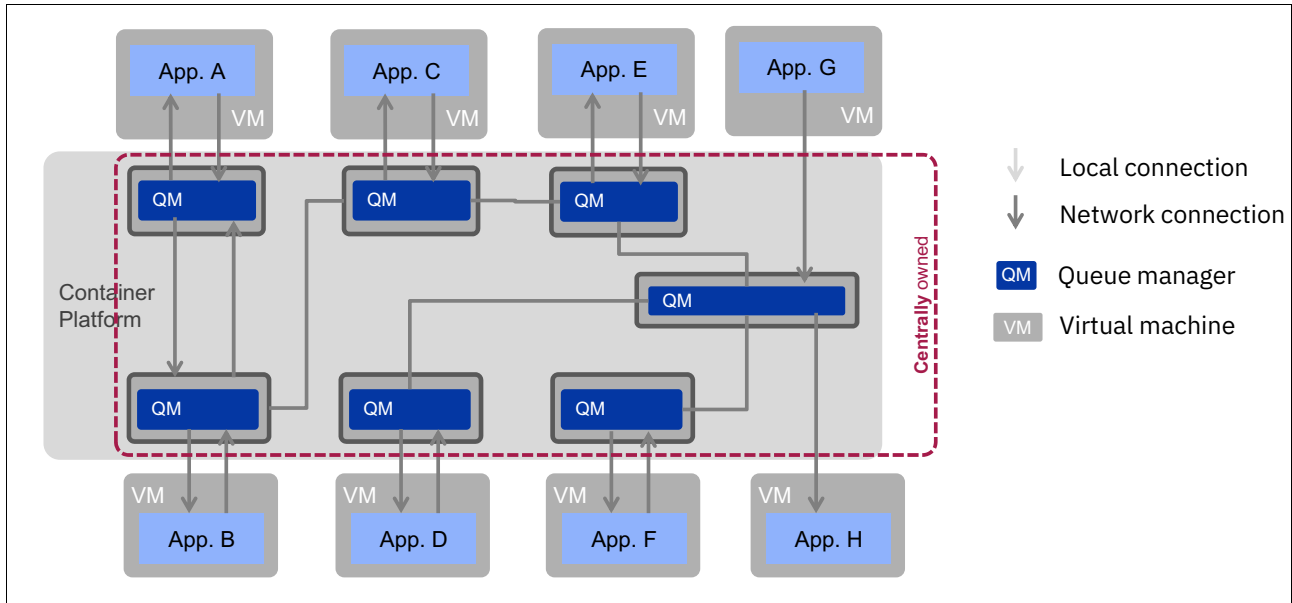


Figure 9-7 Highly centralized ownership model

This created friction, as the development teams want more control to change the configuration, while the central IT team felt they are bombarded by requests of development teams. A common way to address this tension is to enable a more agile deployment model that offers the potential to have more decentralized ownership of MQ resources where appropriate:

- ▶ **Application-owned IBM MQ resources:** Development teams would like to move to a more autonomous ownership model. They want access to update the IBM MQ resources to complete day-to-day activities, such as creating queues, topics and subscriptions, and potentially even to create their own queue managers. The precise capabilities a development team would have access to, and within which environments (development, test, pre-production, and production) would depend on each organization.
- ▶ **Self-service:** When a development team starts a project that requires messaging, they will want a sandbox environment to kickstart their development. Traditionally the central IT team provided a sandbox and might have required several days or weeks to do so. Now, sandbox deployment is often automated and available to the development teams through a self-service portal, for example. As a result, development teams can immediately provision an IBM MQ queue manager instance, often within a matter of minutes.
- ▶ **DevOps pipeline:** In early-stage environments such as development, the creation of IBM MQ instances and objects might be completed using web portals. However, this practice will quickly mature to be driven by a DevOps pipeline. This pipeline will allow application teams to load definitions of IBM MQ resources into a code repository alongside their application code. And then the deployment is automatically built, deployed, and tested with their application.

This more decentralized approach to messaging capabilities is highlighted in Figure 9-8.

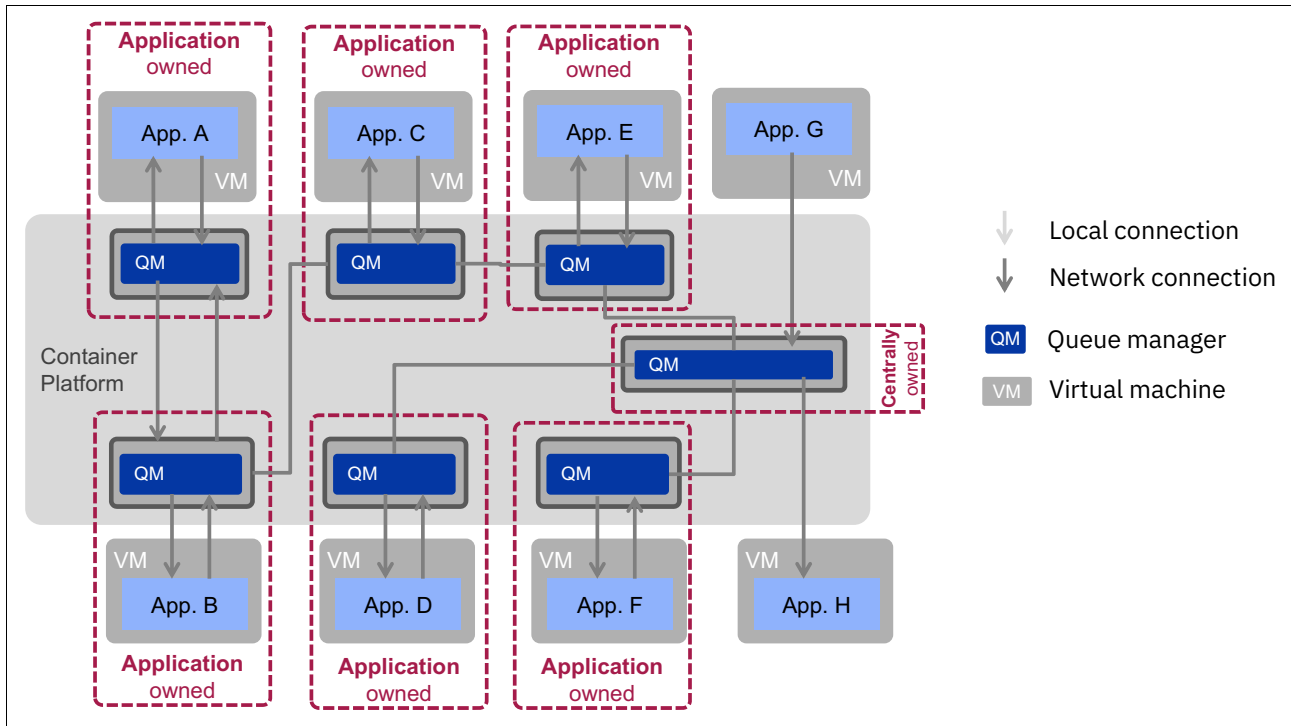


Figure 9-8 Decentralized ownership

Clearly there are few organizations where the application teams would willingly take on the entire role of looking after their own IBM MQ estate. In addition, queues form a core part of the enterprise’s communication infrastructure and as such, we would want to ensure a level of governance over how they are deployed and configured.

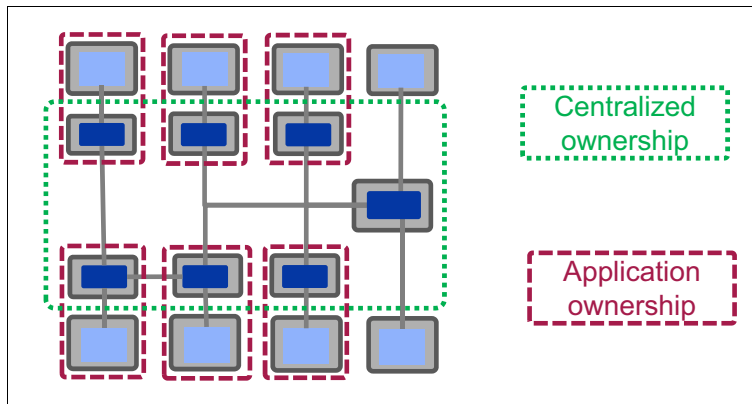


Figure 9-9 The reality - a combination of both centralized and application-based ownership

Typically we see a natural split of responsibilities between central and application teams as shown in Figure 9-9. The central teams focus on ways to templatize and automate the messaging capabilities. The application teams use those patterns and templates to create the queues that they require. Let’s briefly look at that in a little more detail.

The central team’s key concern is that the IBM MQ infrastructure is rolled out in a well-controlled and governed way based on good practice. They would focus on the creation of templates and patterns to use for the creation of queue managers and their accompanying channel definitions. This might for example provide different templates for different usages, such as “t-shirt sizes” for small/medium/large usage profiles (as we do on IBM MQ on Cloud),

to simplify choices. Along with that would come defaults that relate to aspects such as log configuration, what persistent volume capabilities to use, and so on.

The application team would then have ownership of the actual creation of queue managers, queues and topics based on those patterns and templates. They would then be responsible for the runtime administration of the queues, and for consumer-specific requirements such as the level and type of security to place on the queues.

So, the traditional central messaging team would no longer perform all IBM MQ provisioning onto a shared infrastructure. They would instead build out *Messaging as a Service* for other teams (for example non-messaging engineers from application teams) to consume via self-service.

It is worth noting that to enable self-service effectively requires a very different way of working for the central team. There needs to be a high degree of patternization. As a result, queue topologies can be built in simply, predictably, and most importantly, through automated pipelines. Furthermore, roles of at least some of the central team would need to move more towards that of Site Reliability Engineers to ensure continued improvements in scalability, reliability and automation of the administration of the environments. The payback for this of course is that the patternization and automation enables the central IT team to retain some level of governance across an increasingly decentralized landscape.

We explore pipeline-based deployment of IBM MQ in more detail in 9.4.1, “Design for DevOps pipeline” on page 594.

9.1.6 Application containerization and operational consistency

Throughout this section so far, we have focused on modernization of the IBM MQ infrastructure in isolation. However, it is likely that the applications themselves are also undergoing modernization. For example, we might be refactoring an application into a number of smaller microservice components, and it is likely that these too are destined to be run in containers.

Consider a scenario in which we have already moved to the use of remote queue managers rather than local ones. In turn, we use remote clients, rather than local connections. So, from a connectivity standpoint, it makes little difference whether the application is containerized or not. They simply continue to use the client binding as they did before.

However, there is another aspect that might make a combined move to containers for both application and IBM MQ more attractive. A move to containers is more than just an infrastructure change, it also brings in standardization of many other things that were previously specific to each product. A container orchestration capability such as Kubernetes provides common mechanisms for topology deployment, workload routing, high availability, health checking, scaling, handling of credentials and more. Indeed, commercial versions of Kubernetes often standardize much more, such as monitoring, security and logging. We call this *operational consistency* and it essentially means that the operational skills required to look after the environment are consistent across products, simplifying management of a diverse technology portfolio. See Figure 9-10.

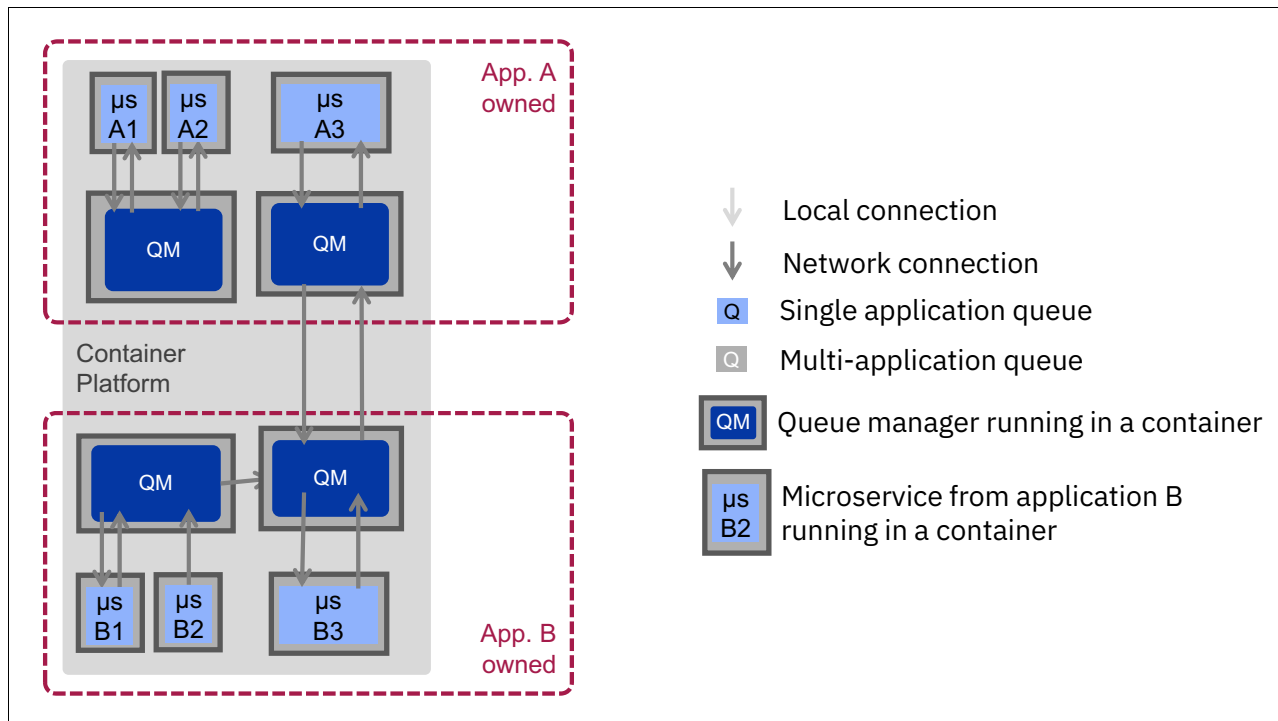


Figure 9-10 Containerized integration and applications that bring operational consistency

9.2 IBM MQ availability

Within this section we discuss how best to implement IBM MQ in a container orchestration platform in order to leverage its inherent availability capabilities.

First, we should recognize that the topic of availability has a subtle nuance when applied to messaging. There are two aspects: *message* and *service* availability.

- ▶ **Message availability:** Message availability relates primarily to the ability to perform message “gets” (as opposed to “puts”) and specifically to whether *all* currently stored messages are available. With IBM MQ distributed, a message is stored on exactly one queue manager. If that queue manager becomes unavailable, you temporarily lose access to the messages it holds. To achieve high *message availability*, you need to be able to recover a specific failed instance of a queue manager and its message as quickly as possible.
- ▶ **Service availability:** Service availability helps ensure that you can always “put” to a queue, and you can always “get” *some* messages (even if a subset of messages is temporarily unavailable). You can achieve *service availability* by having multiple instances of queues for client applications to use on different queue managers. When continuous availability is discussed, it is usually in relation to *service availability*.

When you architect for availability, it is critically important to target the type of availability that is most important for your specific application needs.

9.2.1 Kubernetes deployment styles: Deployments (replica sets) versus stateful sets

Kubernetes provides two approaches for managing sets of containers: *deployments* (which use replica sets in the background) and *stateful sets*.

- ▶ **Deployments (based on replica sets):** As the name suggests Deployments that are based on replica sets allow Kubernetes to create “replicas” of the container. In this situation, the containers are considered completely indistinguishable. So, they can be created and destroyed by Kubernetes at will. As such they cannot be uniquely identified and addressable. The precise number of replicas is purely a target for Kubernetes to aim for, and there might be more or less than that number at any given time. They can have persistent storage attached, but it is typically shared across all replicas. As such it should not be used for replica-specific state.
- ▶ **Stateful sets:** Stateful sets are different in that they are created explicitly with sequential names, and each one can be addressed directly. The number of replicas is precise, and Kubernetes will aim to ensure all replicas requested (and no more) are present. Should one of the containers be lost, it will be reinstated under the same name, with the same instance of persistent storage attached.

The replication performed by the preceding types of sets actually happens at the level of a Kubernetes “Pod” rather than on the containers themselves. However, in this instance there is a one-to-one relationship between queue manager, container, and pod. So, for simplicity in this short section we just talk in terms of containers.

From an IBM MQ perspective, there is one condition that is critical regardless of which of the preceding high availability types that we are targeting. For any given queue manager state that is held on shared storage, there cannot be more than one *active* queue manager. This restriction is common in stateful components to ensure data integrity. This means that we need to use a specific type of deployment mechanism on Kubernetes. For completeness, we should briefly note that for IBM MQ on mainframe, we actually can share persistent storage across two active queue managers by using “shared groups.” But that is unique to that platform. Here, we are discussing non-mainframe installations only, where there can only be one active queue manager.

Replica sets allow the platform to perform very elastic scaling of the component based on the replication settings provided. However, during this highly dynamic creation and deletion of replicas, the replica count is approximate rather than exact. As such, even if we have only asked for one replica, we can for a brief moment have two replicas running. As we know, in IBM MQ it is not acceptable for multiple containers to represent the same queue manager.

Even if we were to set up persistence such that each replica had its own separate storage, there are other reasons why we cannot use replica sets. Primarily this is because they do not provide specific addresses to the individual replicas, which is required in many circumstances. The following are common examples:

- ▶ Applications often require connectivity to a specific queue manager to assure request messages are processed. This would require a separate network address for each queue manager.
- ▶ Capabilities such as the IBM MQ uniform cluster — which we will discuss later — depend on a dedicated endpoint for each queue manager.
- ▶ Transactional recovery depends on the ability to reestablish a connection to the same logical queue manager after a failure.

- JMS connectivity sometimes creates two separate connections to a queue manager for a single logical connection. Kubernetes might cause these two connection attempts to be distributed across different queue manager replicas, resulting in a failure of the interaction.

Figure 9-11 shows reinstatement on container failure for replica sets and stateful sets.

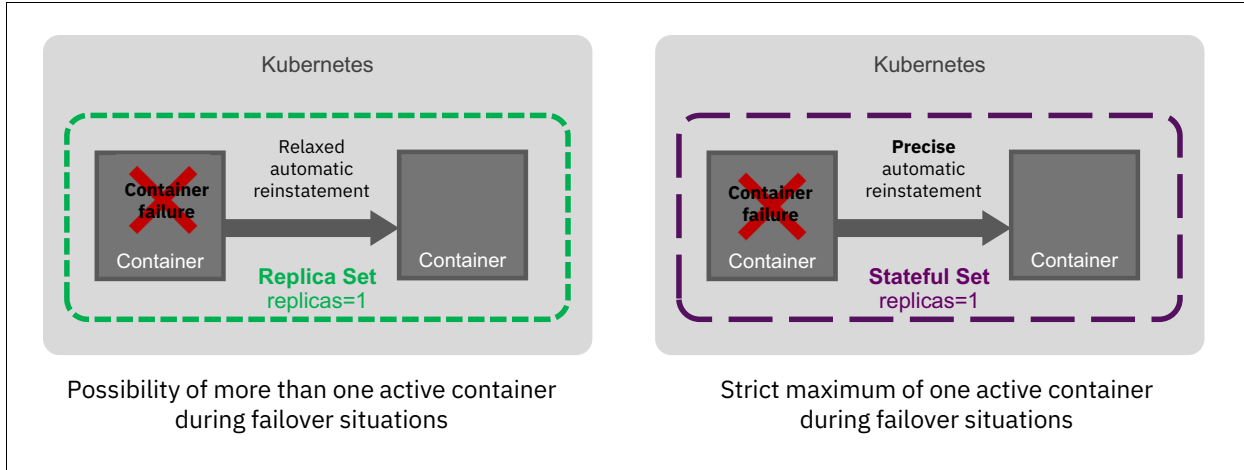


Figure 9-11 Replica sets and stateful sets – reinstatement on container failure

Kubernetes' stateful sets overcome this by providing a much more stringent management of their containers. They ensure that the replica count is a maximum value. So, for example, we only ever have one queue manager running if the replica count is 1. The stateful sets also enable the specific replicas within the set to be addressable. As a result, in nearly all cases, we use stateful sets to deploy IBM MQ in containers.

Single resilient queue manager pattern

We'll begin with the simple cold-standby pattern, as translated into Kubernetes. This has become known as the *single resilient queue manager pattern*, as shown in Figure 9-12.

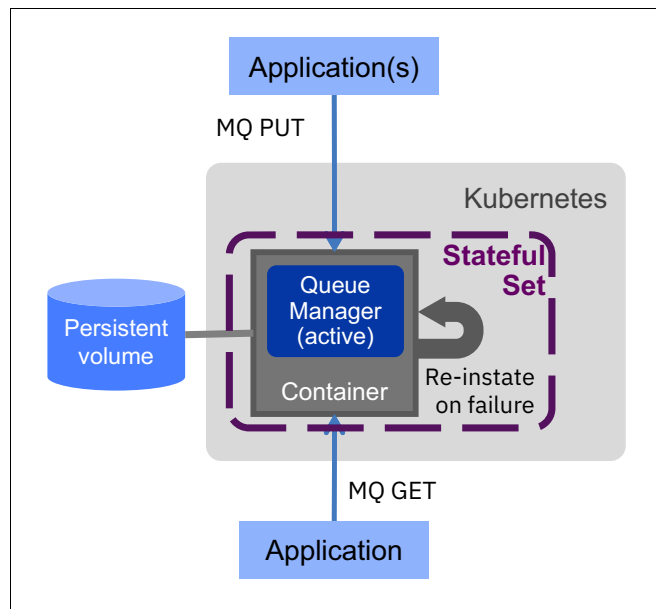


Figure 9-12 Single resilient queue manager (equivalent of traditional cold standby)

This pattern uses Kubernetes in-built availability capabilities to ensure that one and only one instance of the queue manager is always running. Should the queue manager fail, Kubernetes will reinstantiate the container on a node within the Kubernetes cluster that has sufficient resources. The queue manager data is stored on a persistent volume to ensure that it is not lost when the container instance is shut down, and that it can be attached to the new container as it is started up.

This would seem to be a complete solution were it not for a subtlety in the way that Kubernetes handles Stateful Sets under specific circumstances – namely a failure of a worker node, as explained in the next section.

Worker node failures and stateful sets

A *node* in Kubernetes terminology is simply a machine instance (for example a virtual machine) in the Kubernetes cluster. Kubernetes clusters contain various types of nodes but the ones we most care about in this section are “worker nodes”. These are the nodes on which our containers can run.

The simplest failure scenario for a container is that we lose the container instance, but the worker node continues to run. Assuming there are still enough resources available, Kubernetes spins up a new instance of the container. Potentially it places the instance in the same node, reattaches it to the persistent volume, and makes the queue manager available again.

However, there will obviously be circumstances when a worker node itself fails, bringing down all its running containers too. Kubernetes will then be forced to seek out a new node on which to start the container. For deployments (replica sets), this happens automatically, and by the magic of the orchestration platform, largely invisibly to the users of the container.

Figure 9-13 shows reinstatement on node failure for replica sets and stateful sets.

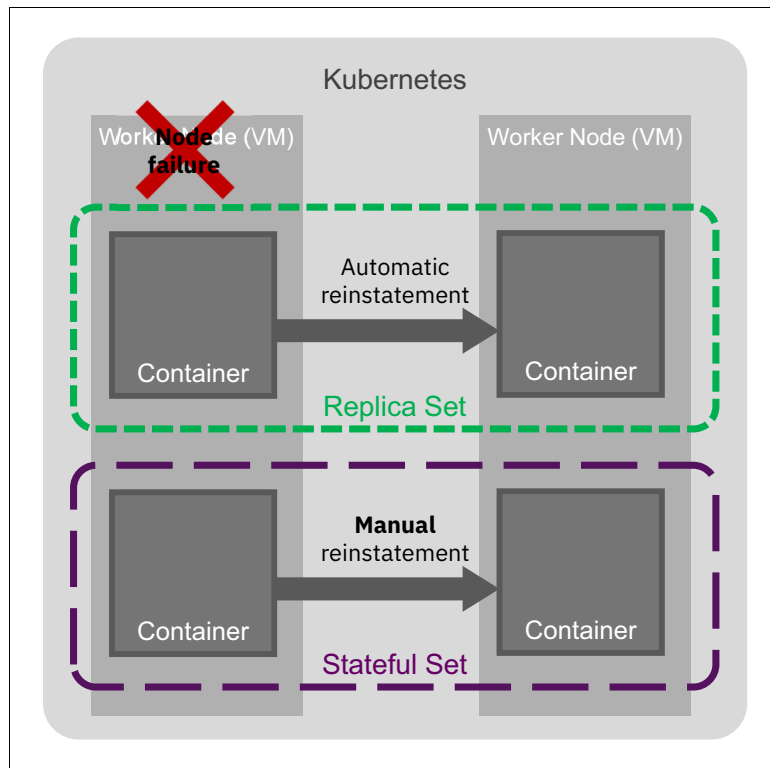


Figure 9-13 Replica sets and stateful sets – reinstatement on node failure

Stateful sets promise to be stringent on the number of container instances that they instantiate. However, they need to be absolutely certain that the container on the current node had really disappeared. This requires confirmation of the node failure, which it turns out, is surprisingly difficult to ascertain. What if the node is still running, and performing work but Kubernetes just can't communicate with it? For this reason, currently Kubernetes will not start the container automatically on another node. It waits for an external actor such as a system administrator to confirm whether it is safe to restart the container by issuing a manual delete command to force the failover.

While it is obviously good that Kubernetes is cautious in this situation, it makes it difficult for us to provide a fully automated highly available environment. There are some third-party solutions that offer resolutions to how to see whether a node is truly down, but they are not part of a default Kubernetes deployment. In fact, we have a mechanism already available in IBM MQ that can help resolve this issue.

9.2.2 Multi-instance queue managers in containers

It turns out that IBM MQ's multi-instance queue manager capability is already a great fit for solving this problem. It enables us to rapidly spot that a queue manager is no longer running on the original skill and know with confidence that it is safe to start another one in its place.

Figure 9-14 shows the container-based multi-instance queue manager concept.

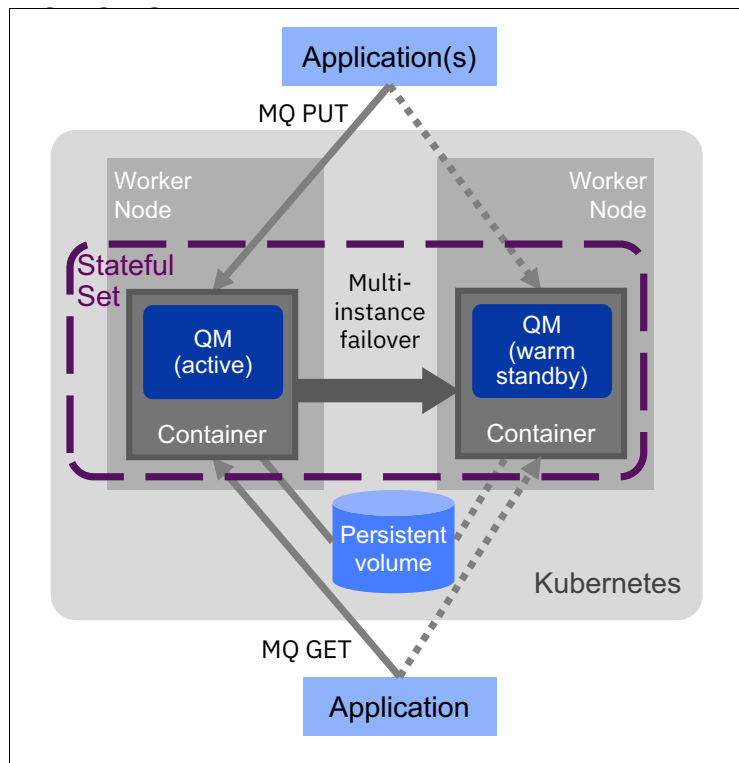


Figure 9-14 Container-based multi-instance queue manager

We can create a stateful set that contains exactly two, multi-instance queue managers that share the same persistent volume. We did not tell Kubernetes which worker node to put each container on. As a general rule, we do not want to do this. We prefer that the platform make that decision based on the available resource on the nodes. Instead of assigning nodes, we provide affinity guidance that says we want each of the containers within any given stateful set to be on different nodes to ensure availability.

IBM MQ's multi-instance queue managers automatically check the file locks on the persistent volume at startup to establish if another queue manager is active. If the file lock has been taken, the second instance will sit waiting in (warm) standby mode. Should there be a failure of the active container — or indeed, of the whole worker node on which it sits — the file lock is released, and the standby queue manager immediately becomes active and is available to service requests. The manual activity required to reinstate the original queue manager still needs to be done, but it is no longer on the critical path.

In the last paragraph we mentioned the importance of file locking on the shared storage. To make this work, your persistent volume provider must provide the following features:

- ▶ The ability to perform ReadWriteMany (RWX) (that is, active on more than one pod at a time), as opposed to ReadWriteOnce
- ▶ The RWX file system also must have suitable file locking semantics to support IBM MQ's multi-instance requirements

Both of these requirements are specific to the multi-instance queue manager capability and are not provided by all persistent volume providers, so you do need to check the details carefully.

9.2.3 Further improving service availability with additional independent queue managers

Although the failover speed of multi-instance queue managers is rapid, often in the order of only a few seconds, it is still measurable. A very brief outage still might be perceived by messaging clients. The only way to provide an almost completely seamless transition on failover would be to have another queue manager already active.

We can have only one fully active queue manager against any given set of queue data (on non-mainframe platforms). As a result, the queue manager that you maintain in this state would have to be completely independent, with separate queue data (although with queues of the same name). Our application could then immediately reconnect to the second queue manager if our current queue manager fails.

From a *service availability* point of view, this would ensure an even higher availability for both message “puts” and “gets”. It's worth noting that the “message availability” remains the same. There will now always be at least one queue manager available to retrieve messages from (service availability). However, the messages on the failed queue manager will still have to wait for the multi-instance failover to complete before they become available (message availability).

So, let's take a look at how that might look if deployed onto Kubernetes. We would deploy two separate stateful sets. As before, each stateful set will have a pair of queue managers, one of which will become active on startup, and one in warm standby mode.

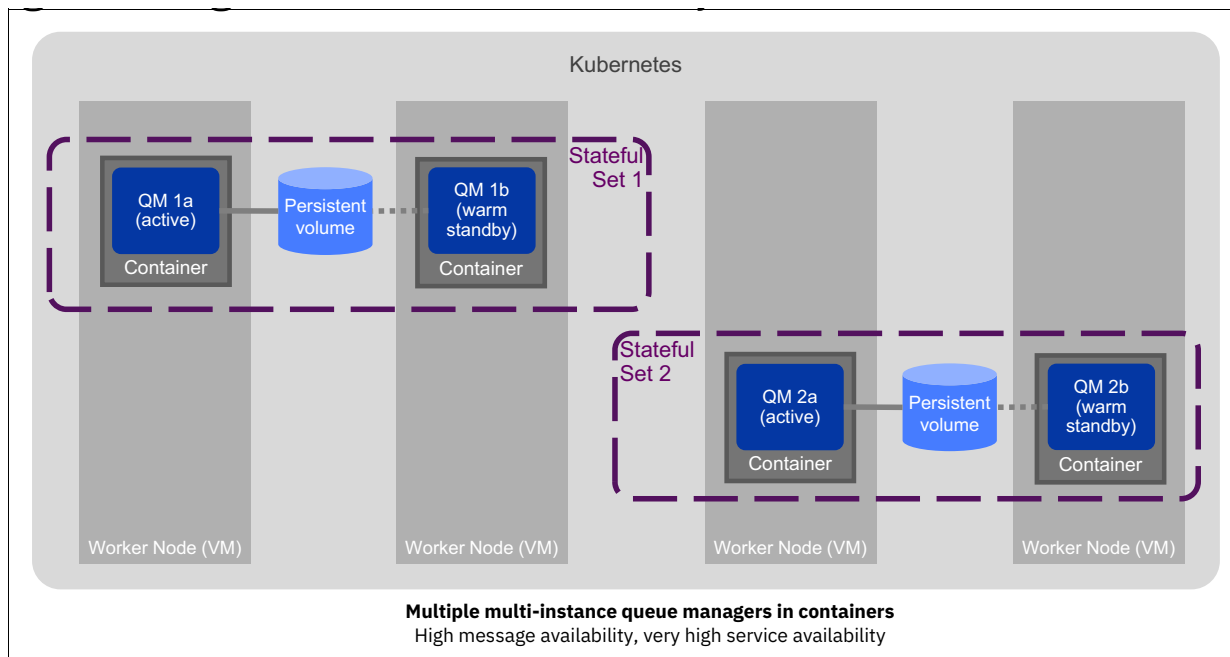


Figure 9-15 Achieving very high service availability

Kubernetes then the sets to worker nodes with sufficient resources, for example, as shown in Figure 9-16 on page 588.

9.2.4 Connection distribution

Closely tied to high availability is connection distribution. It doesn't matter how quickly a queue manager is made available if no clients connect to it. We need to consider how the clients know what queue managers are available, and how they decide which one to connect to, and under what circumstances they should reconnect in order to improve the connection distribution.

Techniques have existed for some time to provide IBM MQ clients with a list of queue managers. The typical approach is a *Client Channel Definition Table (CCDT)* which can be directly set on the IBM MQ client, or referenced remotely for easier group configuration. This ensures that if a connection to a queue manager is broken, the client can try alternative queue managers until it finds a working connection. Up until recent versions of IBM MQ, after connections are made, they are typically not dynamically redistributed unless a connection fails, and so there is no guarantee that workload is well balanced.

However, a recent advance known as "uniform clusters" (introduced in IBM MQ 9.1.2) takes this one step further by enabling intelligent dynamic, workload-based rerouting of applications connections to queue managers. This caters for the following scenarios:

- ▶ With multiple IBM MQ queue managers, how are application instances evenly distributed across IBM MQ instances?
- ▶ As the number of IBM MQ queue managers is increased, how can it be assured that application instances will be evenly distributed across the IBM MQ instances?
- ▶ In planned maintenance or failure situations how can application instances be redistributed to the remaining IBM MQ queue managers, and then rebalanced when the original queue manager becomes available again?

To configure an IBM MQ uniform cluster a set of equivalent queue managers are deployed, linked together into an IBM MQ Cluster and labeled as a uniform cluster. This gives them the permission to start to chat among themselves, sharing knowledge of which application instances are connected where. With that information, the cluster can start to tackle the preceding problems.

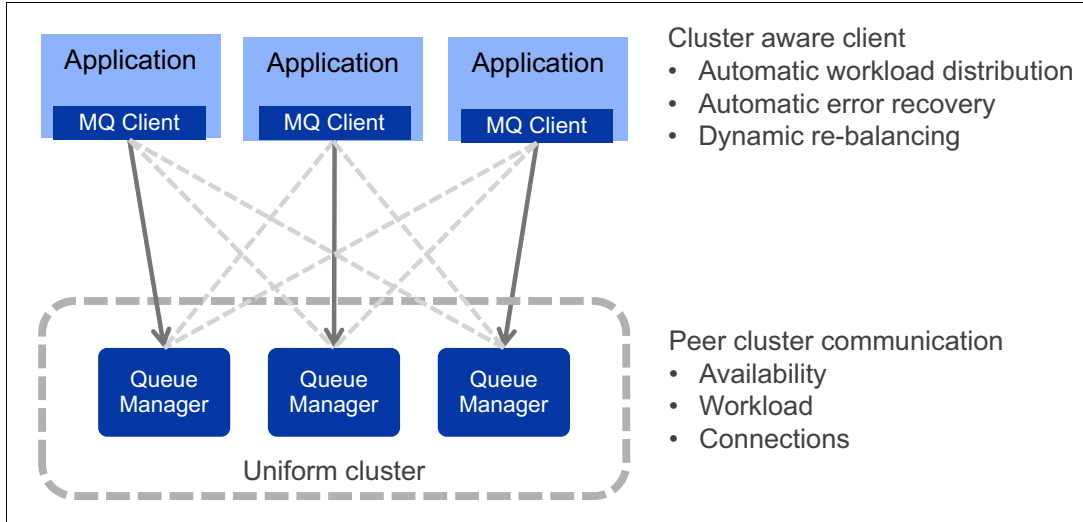


Figure 9-16 Uniform cluster

Figure 9-16 shows queue managers in the uniform cluster that detect an imbalance of applications across the individual queue managers. When imbalance occurs, they automatically move an application's connection from one queue manager to another, reestablishing a balance. The imbalance detection is continually running, making incremental improvements upon the initial balancing and also those times when queue managers are being stopped and started due to maintenance and scaling.

Container environments constantly and dynamically start and stop container replicas based on changes in workload, recovery from failures, and in order to roll out updated container images. IBM MQ's uniform cluster technology ensures that client applications continuously remain evenly distributed across the available IBM MQ container replicas.

We delve more into uniform clusters in the 9.3, "IBM MQ scaling" on page 588.

9.3 IBM MQ scaling

Container technology provides the opportunity to scale deployments by replicating the number of active containers. A load-balancing component commonly sits in front of the active containers and distributes the work across the available instances. IBM MQ has been proven to scale in the most extreme scenarios, this section will look at the common options and how these translate to a container environment.

Traditional horizontal scaling of IBM MQ

Prior to containers, IBM MQ has been repeatedly proven in the most extreme scenarios, by the most demanding of users across the world, while providing the highest qualities of message service. It ensures that no persistent message data is lost, no matter what the problem.

That's not to say that every IBM MQ system behaves like this, thought around the design and how applications use it to maximize the scalability and availability is required. Small deployments of IBM MQ in the past commonly focused on a single queue manager approach.

Figure 9-17 shows single IBM MQ instance topology.

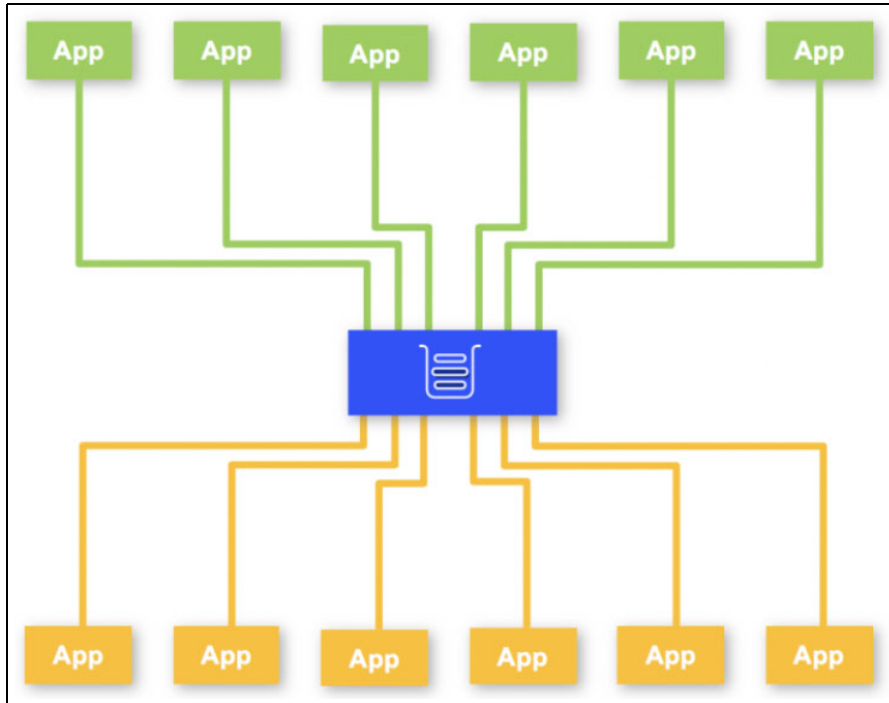


Figure 9-17 Single IBM MQ instance topology

With this model steps, you can take steps to make that single queue manager as available as physically possible. Perhaps you would use one of IBM MQ's high availability capabilities, such as multi-Instance described in 9.2.2, "Multi-instance queue managers in containers" on page 585. This does not avoid those times when the queue manager needs to be restarted or reconfigured. At those times, you have an outage as the queue manager fails over, even if just for a few seconds.

Continuous availability looks to avoid an outage of the whole messaging system differently, an architecture that avoids a single queue manager as the route for all messages. Many clients have built continuous available IBM MQ systems to do just that, these use multiple active queue managers and distribute the messaging workload across them. And it is not enough just to worry about the messaging layer, applications also need to remove any single points of failure, so that means multiple active instances of the applications as shown in Figure 9-18 on page 590.

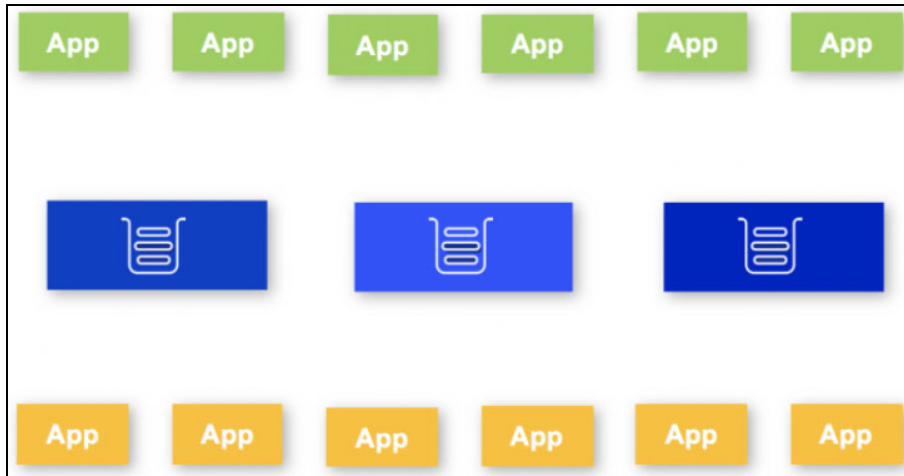


Figure 9-18 Multiple active instances

On z/OS, users will probably be aware of Queue Sharing Groups, and how they help on that platform by exposing centrally shared queues through those multiple queue managers. But on another platform, where such a capability doesn't exist, having multiple queue managers means multiple copies of the same queue, each with a different set of messages.

For new applications, the multi queue manager pattern really should be the starting point and built into the applications expectations from the start. However, adapting existing applications to this pattern that have relied on a single queue manager being the target for all their messages might require extra work. In fact, it might require application changes if the logic is relying on things that are fundamentally counter to highly available solutions, for example global ordering of all message data.

One option to achieve better availability is to create multiple *cloned* environments, and to manually pin application instances across them. See Figure 9-19 on page 590.

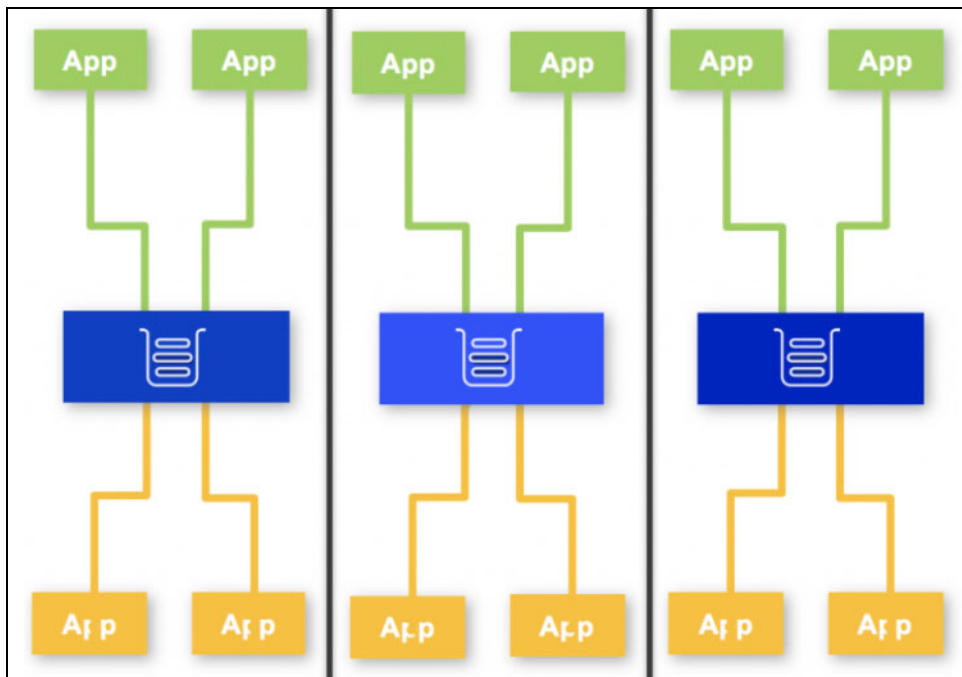


Figure 9-19 Applications tightly coupled to queue manager instances

This means that when a single queue manager stops, only a proportion of the messaging stops. This is an improvement on the single queue manager but not such good news for those instances of the application that are pinned to the stopped queue manager. It can also become a problem to manage these static configurations, constantly needing to reassess how they're configured to ensure they're still meeting the requirements.

An even better approach is to decouple the application instances from the individual queue managers entirely, allowing them to connect to any of them. This is shown in Figure 9-20 on page 591.

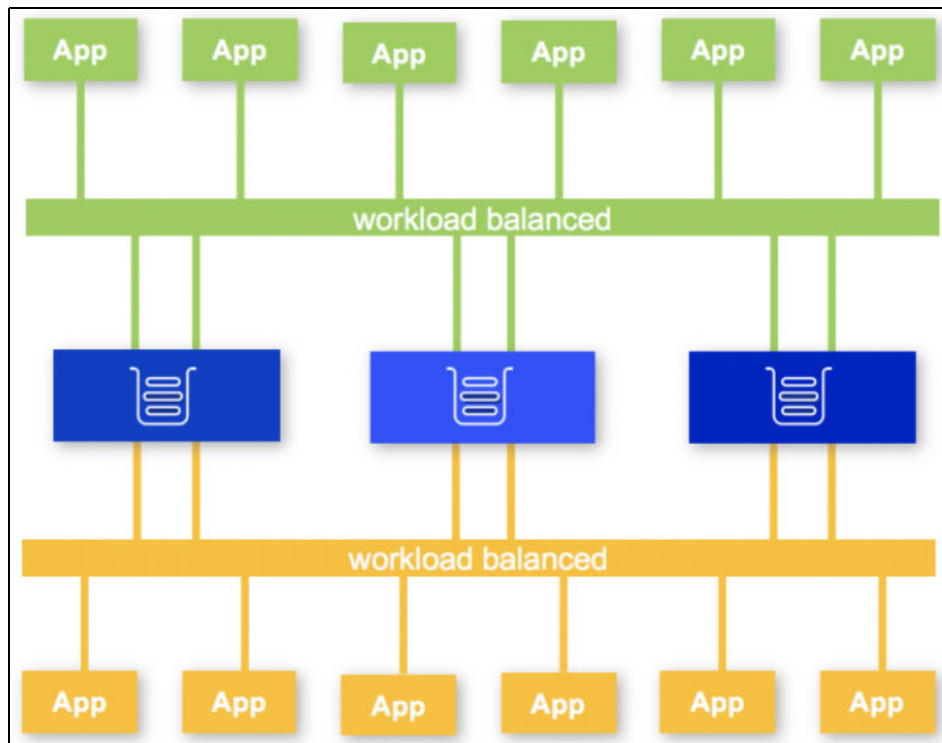


Figure 9-20 Decoupled queue managers and applications

This pattern adheres to the principle that messaging workload is distributed across a set of queue managers. Also, this loosely coupled pattern also brings in the fundamental requirement that applications connect over TCP/IP. Nothing new there, IBM MQ has had specific capabilities to help decouple client connections from a specific queue manager, for example:

- ▶ Client Channel Definition Table (CCDT) queue manager groups to randomize the application connections.
- ▶ Dynamic CCDT retrieval over HTTP/FTP to ensure that applications always have the latest IBM MQ connectivity details.
- ▶ Auto reconnect of applications to hide system and network outages from the applications.
- ▶ IBM MQ Clustering allows queue managers to be connected into a network and resources (such as queues) *advertised* in the cluster. This reduces the administration overhead of IBM MQ networks.

The Uniform Cluster pattern

For the preceding queue manager pattern to succeed, an application needs to treat all those queue managers as equivalent. This means that the application can connect to any one of them and expect to behave the same, with the same queues, topics, security configuration, and so on. Or to put it another way, these queue managers need to be uniform. This is why

IBM MQ has introduced a new phrase to its catalog, the *Uniform Cluster*. In IBM MQ 9.1.2, queue managers can be aware that they're being used for this exact purpose. And this solves some of the challenges that users would have had to address:

How do I ensure that my multiple application instances are evenly distributed across those queue managers? Also, how do I ensure that every queue manager has at least one consuming application for each queue?

If these questions are left to chance with a CCDT to load balance, there are no guarantee of success (Figure 9-21 on page 592).

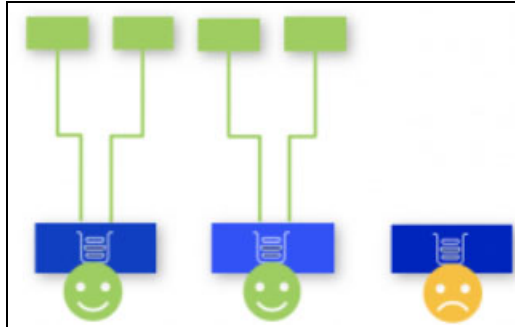


Figure 9-21 Unbalanced connections with traditional CCDT configuration

If a queue manager is stopped for maintenance, applications can reconnect to a remaining queue manager (for example with auto reconnect). But how do applications connect back to that stopped queue manager after it is restarted? See Figure 9-22.

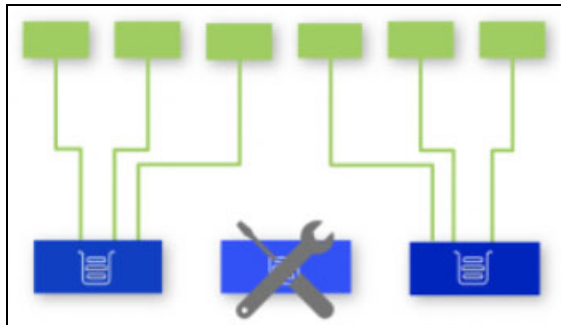


Figure 9-22 Maintenance of queue managers

If a configuration change brings in a new queue manager, perhaps to scale up, how does it get a fair proportion of connected applications? See Figure 9-23.

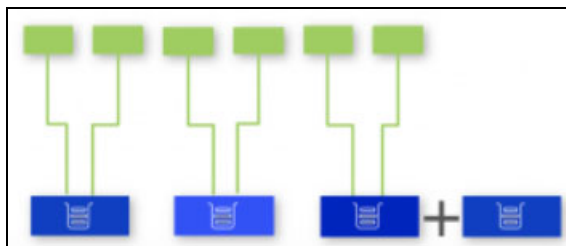


Figure 9-23 Scaling out with additional queue managers

To date, the preceding issues have typically been solved either through increased application logic, or through manual operations being performed. With the release of IBM MQ 9.1.2 on

the distributed platforms (Linux, IBM MQ Appliance, Windows, AIX) the preceding questions can be answered with the introduction of Uniform Clusters.

A set of equivalent queue managers is used and linked together in their own IBM MQ Cluster and configured to be a Uniform Cluster. This gives them the permission to start to chat among themselves, sharing knowledge of which applications are connected where. With that information, the cluster can start to tackle the preceding problems. The queue managers in the Uniform Cluster detect an imbalance of applications across the individual queue managers and automatically move an application's connection from one queue manager to another, reestablishing a balance. This is continually happening, solving that initial balancing and also those times when queue managers are being stopped and started due to maintenance and scaling. See Figure 9-24 on page 593.

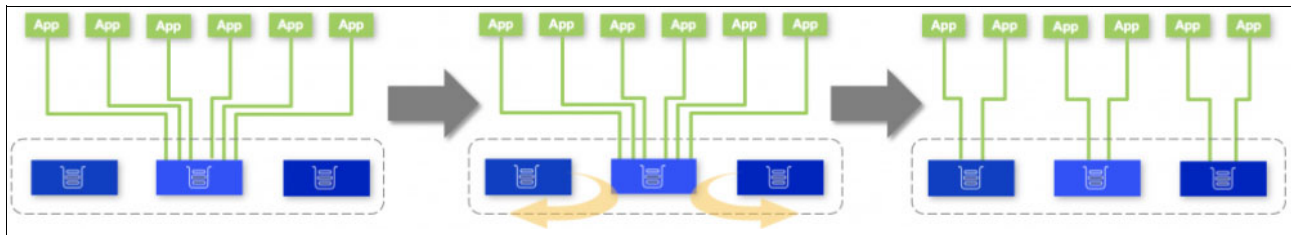


Figure 9-24 Uniform cluster rebalancing

Adopting uniform clusters with containers

As a component is scaled out in a container platform, additional replicas are provided. Load is commonly distributed across active replicas by using a load-balancing component, and in HTTP scenarios where the connection can be short lived, over time a fair distribution is spread across the available instances.

IBM MQ scenarios can be different where the connection between a client and Queue Manager is established and maintained for an extended period of time. These differences are often deliberate to improve the performance of the overall solution. This means that the standard load-balancing approach for a container environment will not work for load running connections, and uniform cluster capability should be used in these scenarios. Each member of a uniform cluster needs a unique entry point for applications to directly access the queue manager (for example a network address). This is used by the uniform cluster client-balancing logic to assure that each queue manager has the correct number of clients attached, and complete any re balancing as needed. Therefore no container load-balancing component should be placed in front of a uniform cluster. Instead, the IBM MQ client should be fully in control of the uniform cluster load balancing. This is illustrated in Figure 9-25.

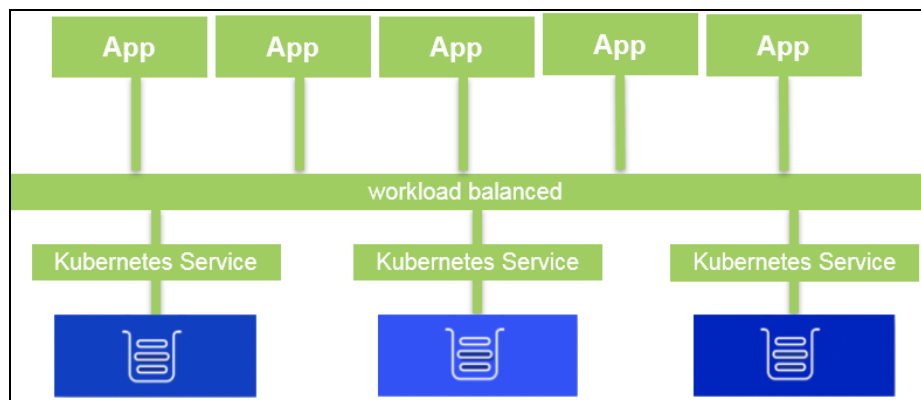


Figure 9-25 Uniform cluster within containers

As discussed, IBM MQ has been repeatedly proven in the most extreme scenarios. And the new uniform cluster capability has simplified the administration activities that are required for many use cases. Uniform clusters are deliberately designed to work within a container environment. With customizations (as mentioned earlier) around the distribution of traffic across replicas, uniform clusters can work naturally within containers.

9.4 Automation of IBM MQ provisioning using a DevOps pipeline

Customers are adopting a decentralized IBM MQ architecture, which can lead to an increased number of IBM MQ instances. Often container orchestration platforms such as Kubernetes are used to streamline the monitoring and management. Automating the deployment and maintenance of IBM MQ is central to providing the agility required for a modern IBM MQ estate. Therefore, a pipeline is critical within an organization.

9.4.1 Design for DevOps pipeline

The DevOps technology that is used might differ depending on the customer, but the high-level approach remains the same. The scenario is that within an organization there are many application development teams. Each of these want an individual IBM MQ instance to be deployed and any updates to either the configuration of IBM MQ or software level should cause a build to occur.

Typically, the DevOps process is separated into four pipelines:

1. **IBM MQ Base Pipeline:** this pipeline generates the core base IBM MQ container image that is used for the downstream pipelines. In certain instances, this can be omitted as IBM MQ Advanced and Cloud Pak for Integration provides a certified container that can be used as is. Other clients might want to define their own container image and can use our publicly available GitHub repository as a starting point for their own container image. This is depicted in Figure 9-26 on page 595.

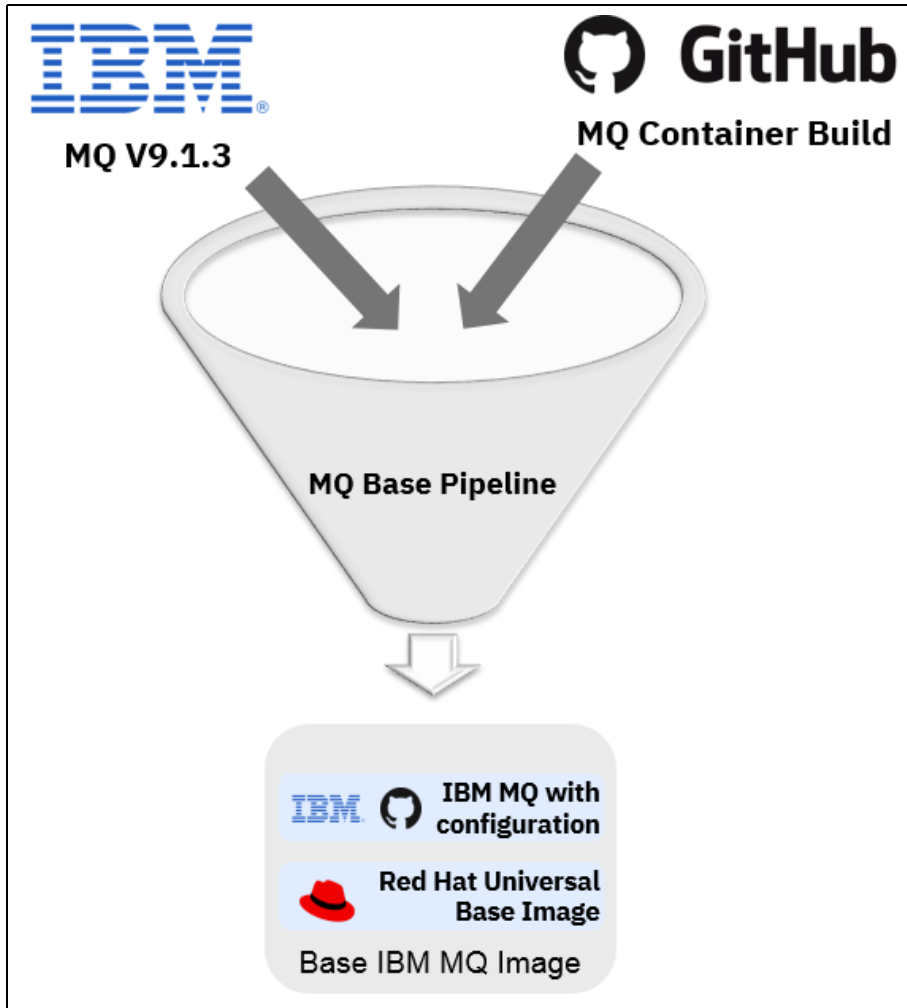


Figure 9-26 IBM MQ Base Pipeline

2. **Enterprise Pipeline** - enterprises commonly have IBM MQ standards around their deployments such as channels, performance characteristics and channel security configuration. This configuration should be added to the *Base IBM MQ Image* and can be included by defining MQSC configuration and layer into `/etc/mqm`. This is depicted in Figure 9-27 on page 596:

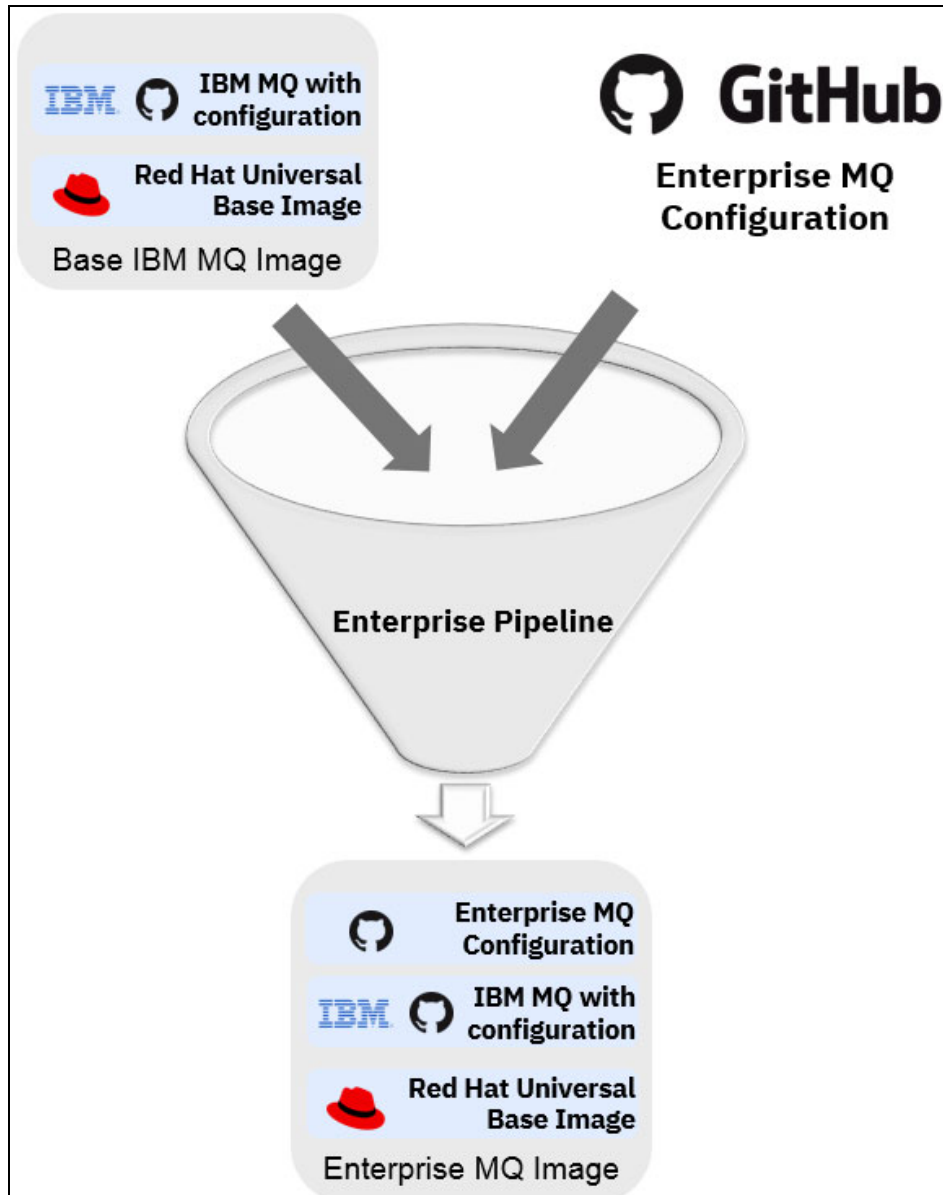


Figure 9-27 Enterprise Pipeline

3. **Application Pipeline:** application teams own their own messaging resources, and the pipeline is configured to process any updates. The messaging resources can either be written within an MQSC file, or within a simplified structure and pre-processed by the pipeline into a valid MQSC file. Typical resources include queues, topics and security access configuration. This is depicted here:

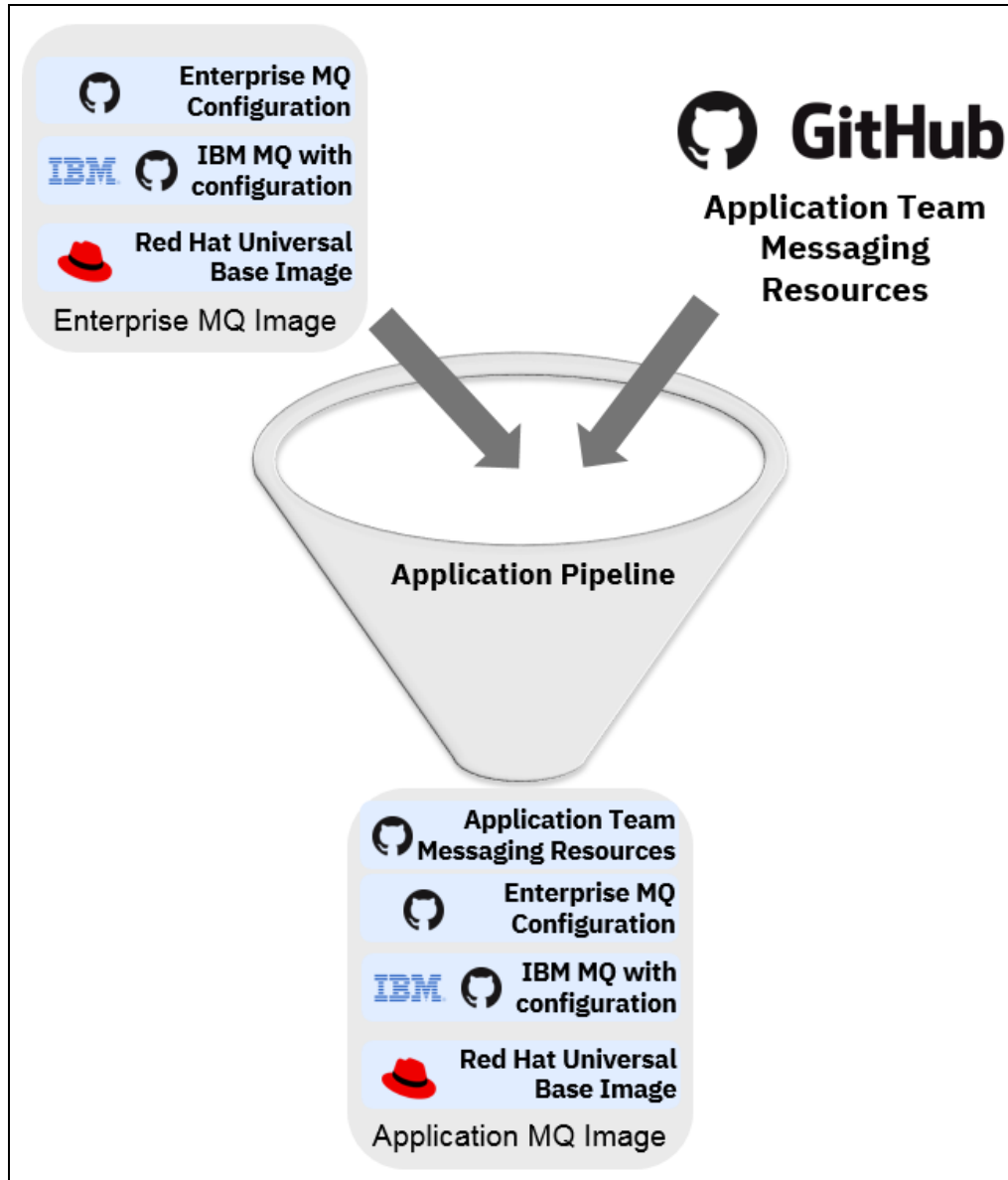


Figure 9-28 Application Pipeline

4. **Deploy to environment:** After the application IBM MQ image is built, this can be deployed to a container orchestration platform. Unless completed in previous stages, security information such as TLS certificates will be associated with the deployment. Depending on the level of maturity, this may involve these activities:
- Deploying to a central development environment
 - Completing automated unit testing against the deployed image
 - Deploying to a testing environment
 - Complete additional automated release testing (functional, non-functional)
 - Deploy to a production environment

The end to end DevOps pipeline is shown in Figure 9-29 on page 598:

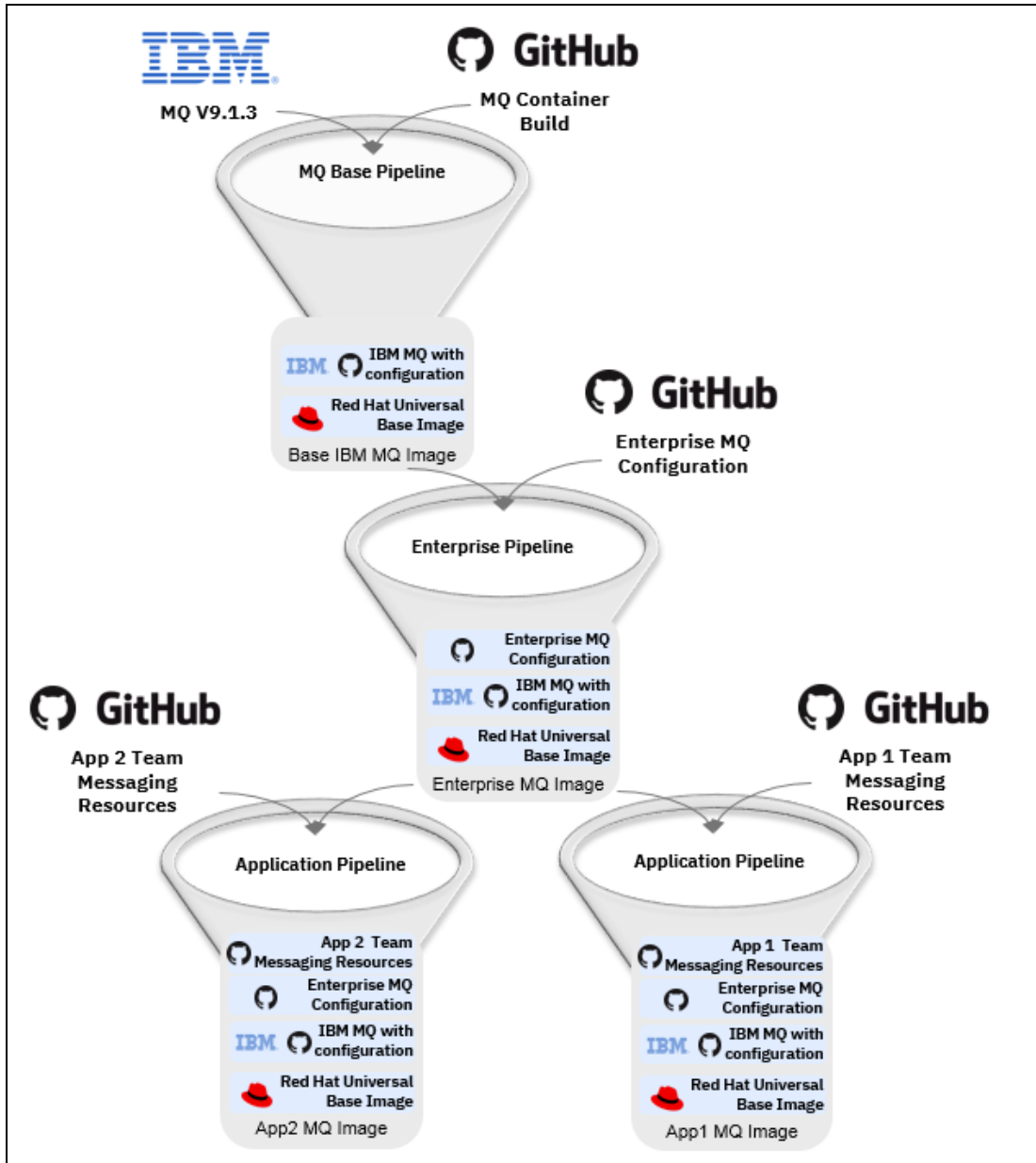


Figure 9-29 End-to-end pipeline flow

The key aspect to highlight is that each of the pipelines are triggered based on changes in the preceding layers, and the associated GitHub repositories. This allows any changes to ripple through the pipeline layers and generate new images for all the associated applications.

9.4.2 Building a sample IBM MQ pipeline

This section will explain how an end to end pipeline flow can be created by using freely available tooling on a fresh installation of Ubuntu. It was deliberately decided to use an Ubuntu image instead of a complete Cloud Pak for Integration installation to allow developers to reproduce on their own laptops. The steps required to prepare the Ubuntu image have been included for completeness.

Prepare the Ubuntu image

These instructions have been developed in a freshly installed image of Ubuntu 16.04, with Docker Community Edition, and Jenkins for the automation. A similar process can be completed on other Linux releases, but any adaptation of these instructions is an exercise that is left up to the reader.

Install docker

Perform the following steps to install docker:

1. Log in to the Ubuntu image and open a terminal window. Ensure that the system is up to date by running,

```
sudo apt-get update
```

2. Install the tools and repository for Docker to be installed:

```
sudo apt-get install apt-transport-https ca-certificates curl  
software-properties-common
```

```
sudo curl -fsSl https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add  
-
```

```
sudo add-apt-repository "deb [arch=amd64]  
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

3. Install Docker by using the following commands:

```
sudo apt-get update
```

```
sudo apt-get install docker-ce
```

4. Create a user for Jenkins to use, and configure access to run Docker by using the following commands:

```
sudo adduser jenkins
```

```
sudo usermod -aG docker jenkins
```

```
sudo usermod -aG docker <primaryUser>
```

where the <primaryUser> is the user that you are currently using,

```
sudo systemctl restart docker
```

Install the dependences for the Jenkins and the IBM MQ Container build

Perform the following steps, while still logged in to Ubuntu:

1. Install Git by using the following command:

```
sudo apt-get install git-core
```

2. The Jenkins agent requires a JRE to be installed to orchestrate the pipeline. Run the following command:

```
sudo apt-get install default-jre
```

3. GNU Make is used by the IBM MQ build process. This can be installed by running the following command:

```
sudo apt install gcc
```

4. Install and start ssh by running the following commands:

```
sudo apt-get install openssh-server
```

Starting Jenkins

Perform the following steps to start Jenkins:

1. Jenkins provides a docker container that simplifies the configuration and setup. This will be used during this exercise. Start the container by running the following command:

```
docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home
jenkins/jenkins:1ts
```

Important: When you run the preceding command, you might see the following error:

```
docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home
jenkins/jenkins:1ts
```

```
docker: Got permission denied while trying to connect to the Docker daemon
socket at unix:///var/run/docker.sock: Post
http://%2Fvar%2Frun%2Fdocker.sock/v1.40/containers/create: dial unix
/var/run/docker.sock: connect: permission denied.
```

See 'docker run --help'.

This can occur for several reasons including these:

- ▶ The user has not been added to the docker group, as specified in the Install docker section.
- ▶ Docker has not been restarted as specified in the Install docker section.
- ▶ The terminal needs to be refresh, use ssh <user>@localhost, and reattempt the command.

2. Open a web browser and navigate to <http://localhost:8080> this will request an initial password, run the following command to display the password:

```
sudo cat
/var/lib/docker/volumes/jenkins_home/_data/secrets/initialAdminPassword
```

3. Paste the password into the console and click **Continue**: See Figure 9-30 on page 601.

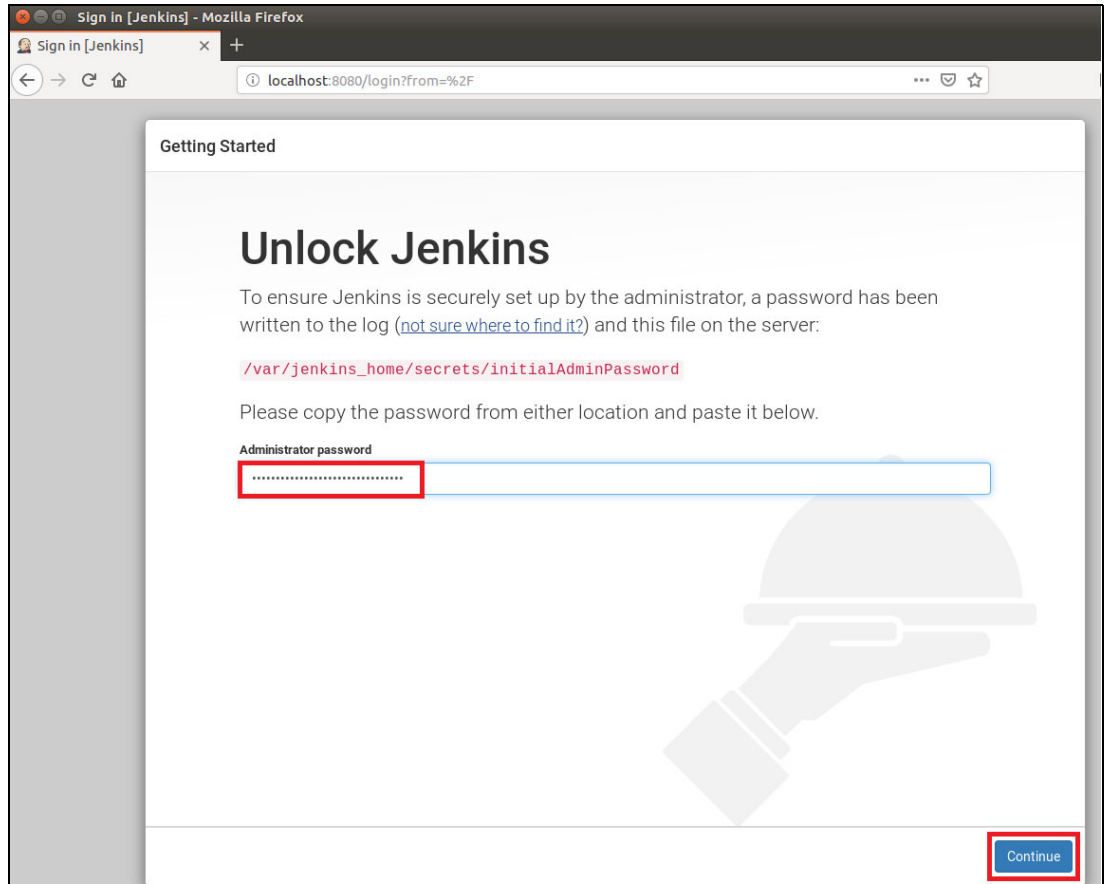


Figure 9-30 Unlock Jenkins

4. The available installation options will be shown (Figure 9-31). Select the **Install suggested plugins** option as shown in Figure 9-31.

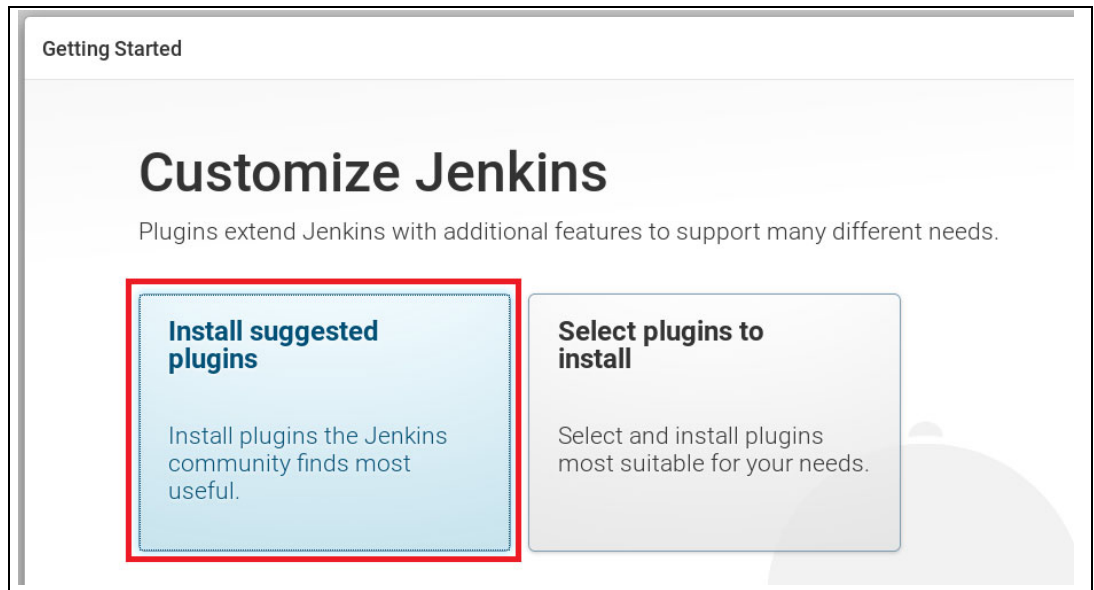


Figure 9-31 Set up Jenkins with the standard plugins

5. Jenkins installs the plugins, and after completion, requires the definition of a new admin user. Fill in the following details:
 - Username: admin
 - Password: Passw0rd!
 - Confirm Password: Passw0rd!
 - Full name: Administrator
 - E-mail address: admin@admin.com

Click **Save and Continue**. See Figure 9-32.

The screenshot shows the 'Getting Started' section of the Jenkins installation wizard. The main heading is 'Create First Admin User'. Below this, there are five input fields, each with a red box around its content: 'Username: admin', 'Password: Passw0rd!', 'Confirm password: Passw0rd!', 'Full name: Administrator', and 'E-mail address: admin@admin.com'. At the bottom left, it says 'Jenkins 2.176.2'. At the bottom right, there are two buttons: 'Continue as admin' and 'Save and Continue', with the latter being highlighted by a red box.

Figure 9-32 Create first admin user

6. On the final configuration screen click **Save and Finish**.
7. Jenkins is ready to use, click **Start using Jenkins**.

Configuring Jenkins to use the host machine for build

Jenkins can be configured with agents that are used to run the pipelines. It is likely that in customer build environments, separate node machines would be used. For this exercise the host machine (Ubuntu) will be used as the prerequisites have been installed. To use, the host machine it needs to be configured, and this is referred to as configuring an agent.

1. Select **Jenkins** → **Manage Jenkins** → **Manage Nodes**:

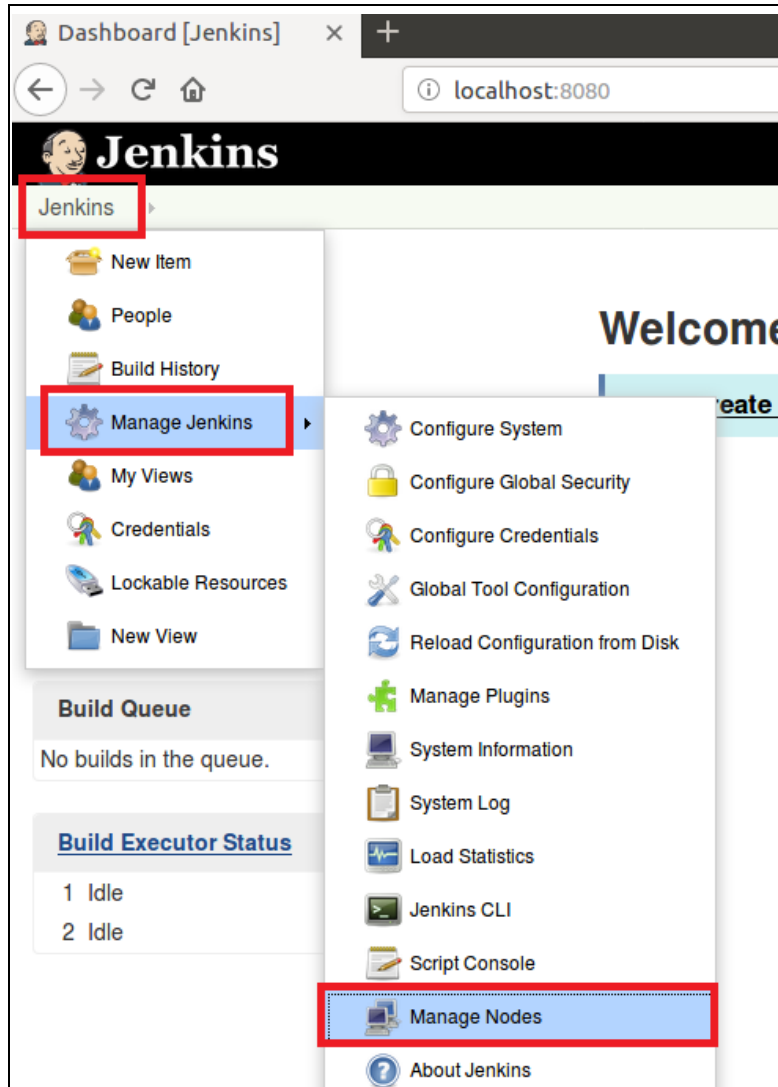


Figure 9-33 Manage nodes

2. Click **New Node**:

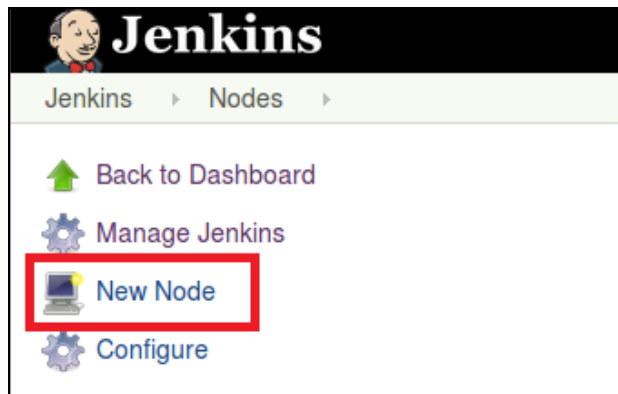


Figure 9-34 New Node

3. In the Node name field type **HostMachine**, select the **Permanent Agent** checkbox and click **OK**. See Figure 9-35 on page 604.

Node name

Permanent Agent

Adds a plain, permanent agent to Jenkins. This is called "permanent" because Jenkins doesn't provide higher level of integration with these agents, such as dynamic provisioning. Select this type if no other agent types apply — for example such as when you are adding a physical computer, virtual machines managed outside Jenkins, etc.

Figure 9-35 Specify node name of the agent

4. Fill in the following details as shown in Figure 9-36:
 - Remote root directory: /home/jenkins
 - Labels: HostMachine
 - Host: <Host IP Address> in our case this was 192.168.246.200

Name

Description

of executors

Remote root directory

Labels

Usage

Launch method

Host

Figure 9-36 Basic configuration of Node Agent

5. New credentials need to be configured to connecting to the host machine. Click on the **Add** drop-down that is associated with the **Credentials** field, and select **Jenkins**:



Figure 9-37 Add credentials for host machine

6. Fill in the credentials for the Jenkins user that was created in the previous steps. See Figure 9-38.

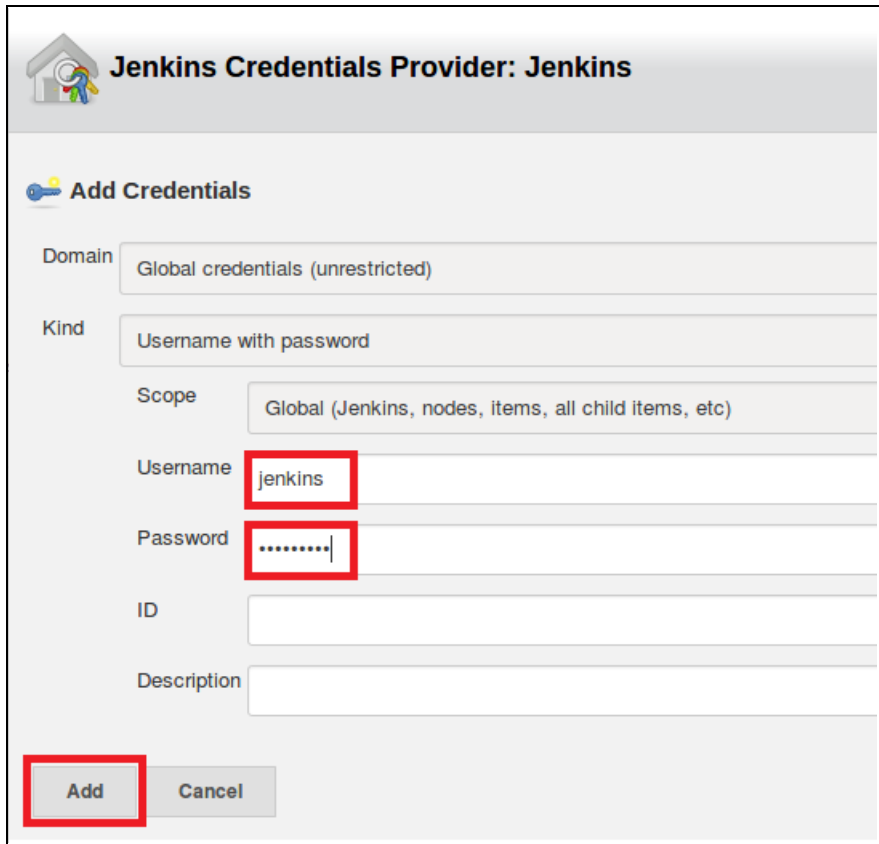


Figure 9-38 Jenkins user details

7. Within the original configuration page complete the following additional configuration (Figure 9-39 on page 606):
 - Credentials: jenkins/*****
 - Host Key Verification Strategy: Non verifying Verification StrategyClick **Save**.

Launch method: Launch agent agents via SSH

Host: 192.168.246.200

Credentials: jenkins/***** Add

Host Key Verification Strategy: Known hosts file Verification Strategy

Availability: Keep this agent online as much as possible

Node Properties

Disable deferred wipeout on this node

Environment variables

Tool Locations

Save

Figure 9-39 Finalize configuration of host

The Nodes table should be updated with the HostMachine and shown as **In sync**. See Figure 9-40.

S	Name ↓	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	HostMachine	Linux (amd64)	In sync	177.44 GB	3.99 GB	177.44 GB	902ms
	master	Linux (amd64)	In sync	177.44 GB	3.99 GB	177.44 GB	0ms
Data obtained		18 sec	17 sec	17 sec	17 sec	17 sec	17 sec

[Refresh status](#)

Figure 9-40 Host machine in sync

This completes the configuration of the agent.

Configure the base IBM MQ pipeline

This section will build the first pipeline for building the base IBM MQ container image. The public IBM MQ container GitHub repository will be used.

1. Select **Jenkins** → **New Item**. See Figure 9-41 on page 607.



Figure 9-41 Create new item in Jenkins

2. Enter **BaseMQPipeline** as the name, select **Freestyle project** and click **OK**:

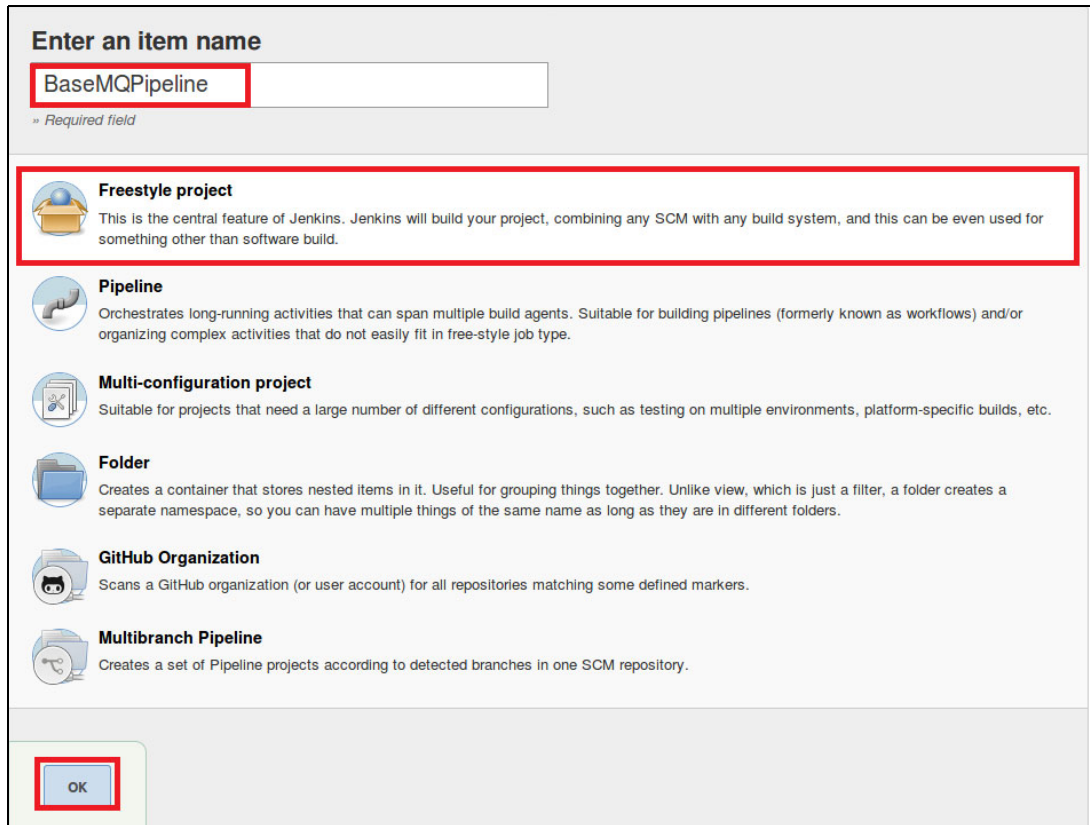


Figure 9-42 Creating base MQ pipeline

3. Within the general section select these options:

- GitHub project: <https://github.com/ibm-messaging/mq-container>
- Restrict where this project can be run: HostMachine

See Figure 9-43 on page 608.

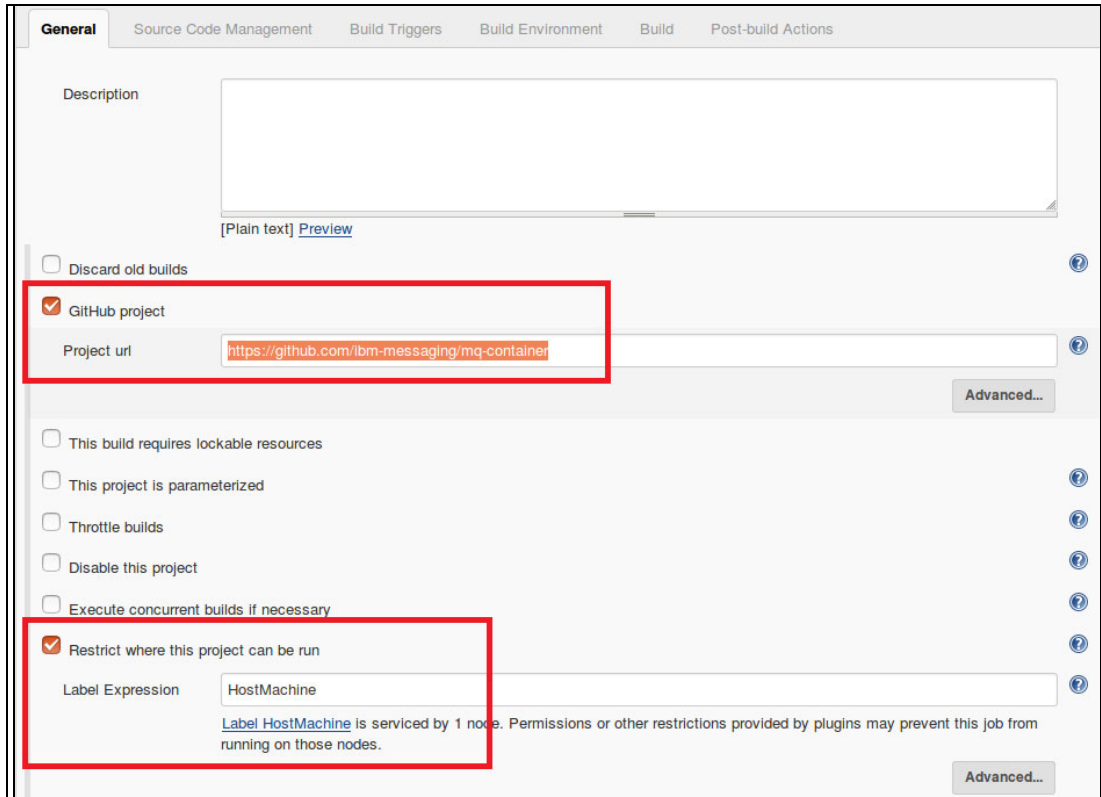


Figure 9-43 Configure the general properties

4. Within the Source Code Management section fill in the following details:
 - Select **Git**
 - Repository URL: <https://github.com/ibm-messaging/mq-container>
 - Branches to build: 9.1.3

Important: The preceding example references the 9.1.3 branch, because it is recommended that **master** NOT be used. Over time, it is expected that the 9.1.3 branch will be removed. For information on the available branches, see the following repository: <https://github.com/ibm-messaging/mq-container/branches>

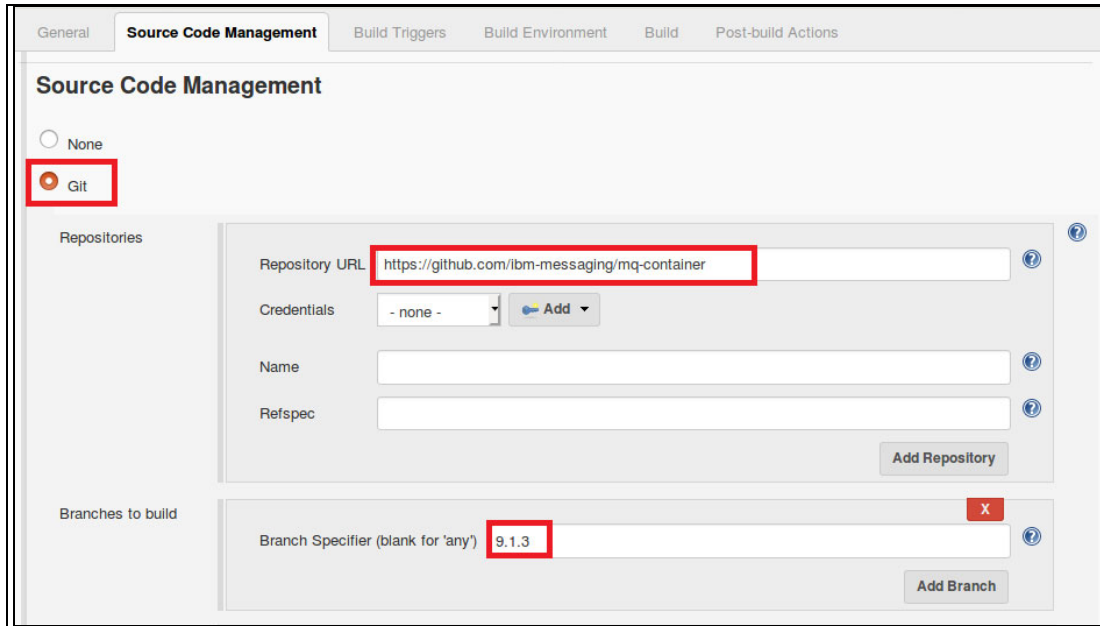


Figure 9-44 Configure the source code management

5. Within the **Built Environment** section select the **Delete workspace before build starts** as shown in Figure 9-45.



Figure 9-45 Configure the workspace to be removed before building

6. Within the **Build** section click **Add build step** and select **Execute shell**. See Figure 9-46.

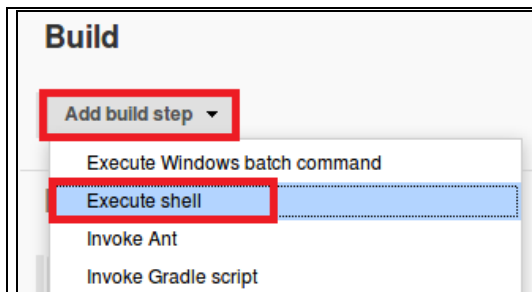


Figure 9-46 Create a new Execute shell

7. Copy the following lines into the shell box:

```
make build-devserver
docker tag mqadvanced-server-dev:9.1.3.0-amd64 mqdev-server:latest
```

Important: The preceding are the current build commands for IBM MQ 9.1.3. These steps regularly change. If you are building a different level, you find up-to-date information here: <https://github.com/ibm-messaging/mq-container/blob/master/docs/building.md>

8. Click **Save** to complete the configuration.
9. Click **Build Now** to verify that the pipeline successfully works. See Figure 9-47.



Figure 9-47 Test the build for base MQ pipeline

Important: The build process includes the ability to download the IBM MQ Developer edition. Depending on the network connection this can cause the build to take minutes to complete. There is an option to download and copy into the build directory. For example, you can include the following lines in the shell script:

```
mkdir downloads  
cp /home/student/Downloads/mqadv_dev913_linux_x86-64.tar.gz downloads
```

After the build is complete, click the build name, and select the **Console Output** to confirm that this is correct.

Configure the enterprise pipeline

This section will build the second pipeline for building the enterprise IBM MQ container image. To simulate an enterprise configuration a GitHub repository has been created here, (<https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration>)

with the content in `/chapter9/MyOrgMQContainer/`. This content is used here:

1. Select **Jenkins** → **New Item**. See Figure 9-48 on page 610.




Figure 9-48 Create new item in Jenkins


2. Enter **Enterprise Pipeline** as the name, select **Freestyle project** and click **OK**. See Figure 9-49.


Enter an item name


EnterprisePipeline


» Required field


 **Freestyle project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

 **Pipeline**
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

 **Multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

 **Folder**
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

 **GitHub Organization**
Scans a GitHub organization (or user account) for all repositories matching some defined markers.

 **Multibranch Pipeline**
Creates a set of Pipeline projects according to detected branches in one SCM repository.

If you want to create a new item from other existing, you can use this option:


 Copy from

Figure 9-49 Creating Enterprise Pipeline

3. Within the general section as shown in Figure 9-50 on page 612 select these options:
 - GitHub project:
<https://github.com/IBMRredbooks/SG248452-Accelerating-Modernization-with-Agile-Integration>
 - Restrict where this project can be run: HostMachine

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Description

[Plain text] [Preview](#)

Discard old builds

GitHub project

Project url:

This build requires lockable resources

This project is parameterized

Throttle builds

Disable this project

Execute concurrent builds if necessary

Restrict where this project can be run

Label Expression:

[Label HostMachine](#) is serviced by 1 node. Permissions or other restrictions provided by plugins may prevent this job from running on those nodes.

Figure 9-50 Configure the general properties

4. Within the Source Code Management section fill in the following details (Figure 9-51):
 - Select **Git**
 - Repository URL:
https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration

Source Code Management

None

Git

Repositories

Repository URL:

Credentials: [Add](#)

[Advanced...](#)

[Add Repository](#)

Branches to build

Branch Specifier (blank for 'any'):

[Add Branch](#)

Repository browser:

Additional Behaviours: [Add](#)

Figure 9-51 Configure the source code management

5. Within the **Build Triggers** section, click **Build after other projects are built** and enter **BaseMQPipeline**. See Figure 9-52.

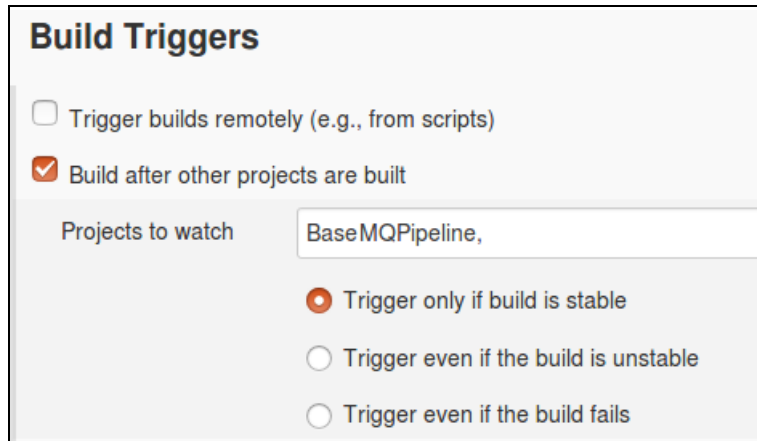


Figure 9-52 Trigger based on base IBM MQ pipeline

6. Within the **Built Environment** section as shown in Figure 9-53 select the **Delete workspace before build starts**:



Figure 9-53 Configure the workspace to be removed before building

7. Within the **Build** section click **Add build step** and select **Execute shell**:

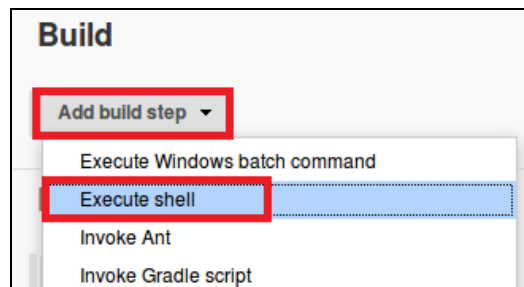


Figure 9-54 Create a new Execute shell

8. Copy the following lines into the shell box:

```
cd chapte9/MyOrgMQContainer
docker build --tag myorg-mq:${BUILD_NUMBER} .
docker tag myorg-mq:${BUILD_NUMBER} myorg-mq:latest
```

9. Click **Save** to complete the configuration.
10. Click **Build Now** to verify that the pipeline successfully works. After the build is complete, click the build name, and select the **Console Output** to verify.

Configure the Application Pipeline

This section will build the final pipeline corresponding to an applications IBM MQ instance. Similar to the previous pipeline a GitHub repository has been created for illustration.

1. Select **Jenkins** → **New Item**. See Figure 9-55.



Figure 9-55 Create new item in Jenkins

2. Enter **ApplicationPipeline** as the name, select **Freestyle project** and click **OK**. See Figure 9-56.

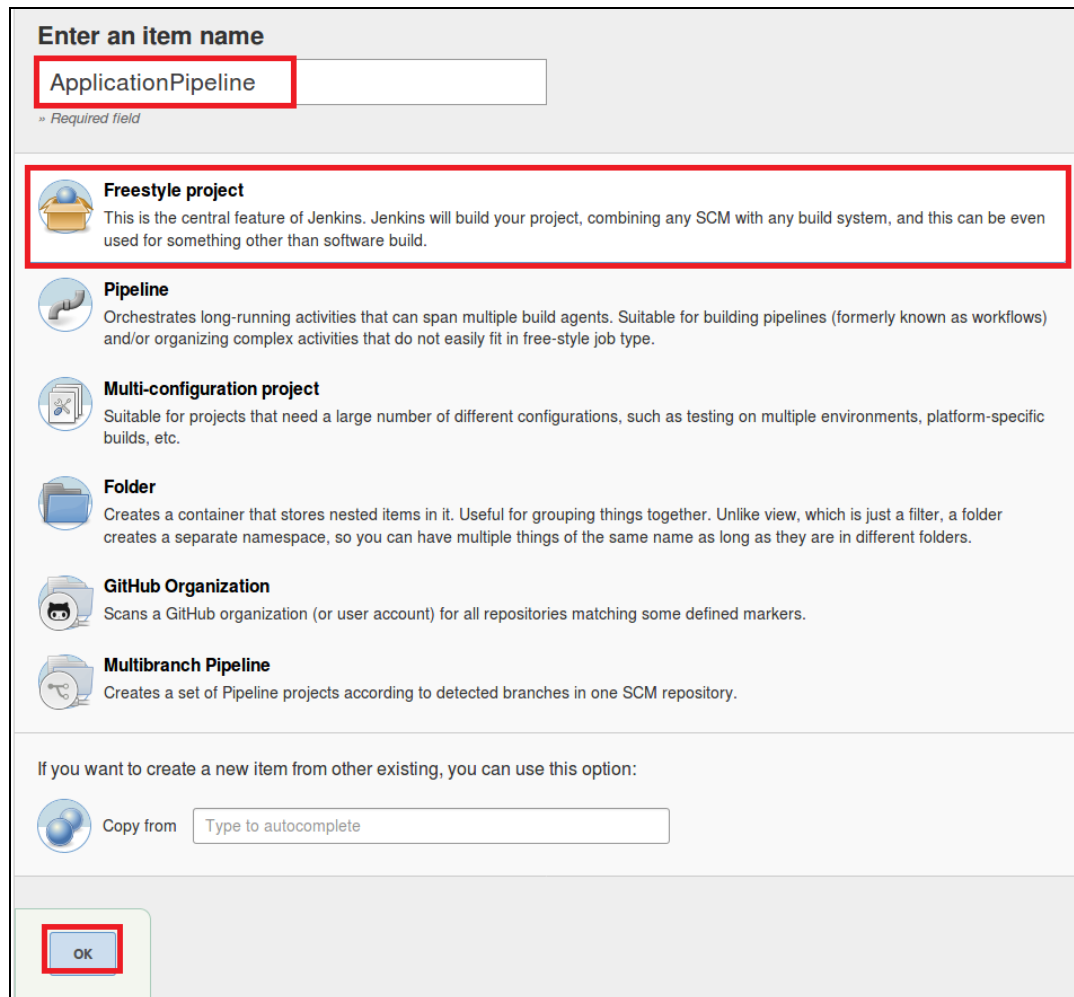
A screenshot of the Jenkins 'Enter an item name' form. The form has a title 'Enter an item name' and a text input field containing 'ApplicationPipeline'. Below the input field, there is a list of project types: 'Freestyle project' (selected and highlighted with a red box), 'Pipeline', 'Multi-configuration project', 'Folder', 'GitHub Organization', and 'Multibranch Pipeline'. At the bottom of the form, there is an 'OK' button (highlighted with a red box) and a section for 'Copy from' with a text input field.

Figure 9-56 Creating base IBM MQ pipeline

3. Within the general section, select (Figure 9-57) these options:

- GitHub project:
https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration
- Restrict where this project can be run: **HostMachine**

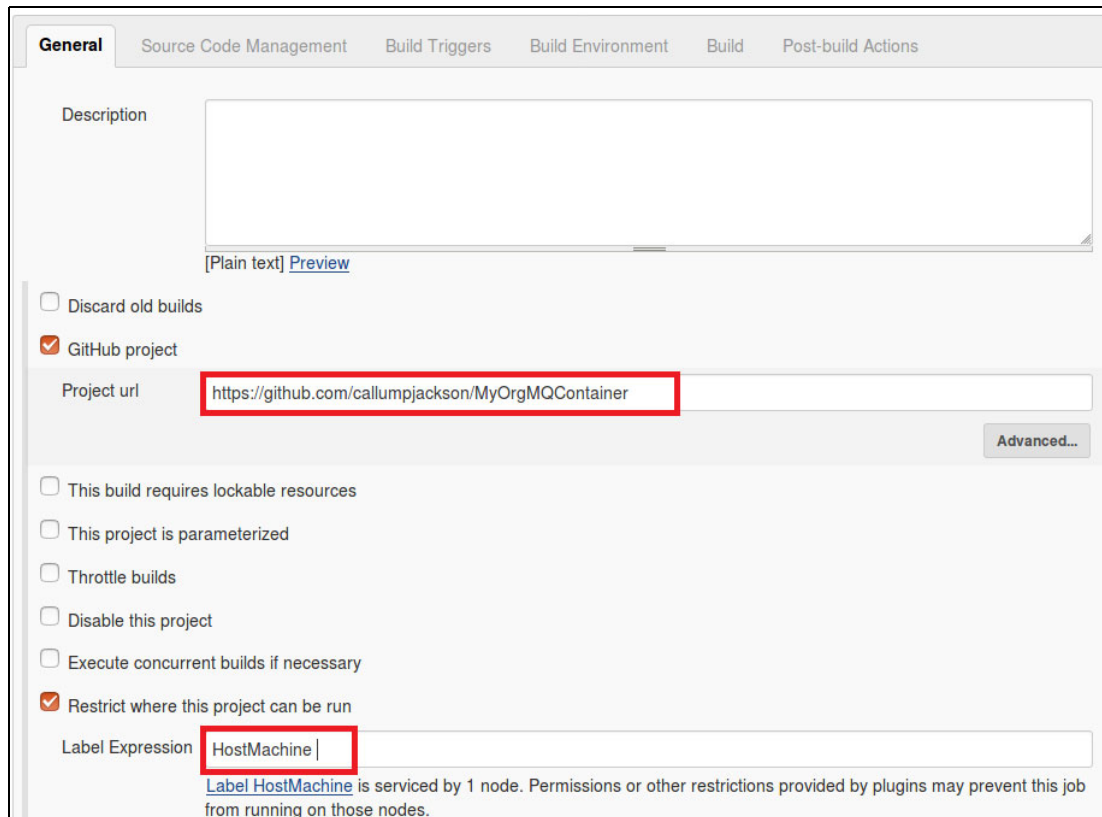


Figure 9-57 Configure the general properties

4. Within the Source Code Management section fill in the following details (Figure 9-58):
 - Select **Git**
 - Repository URL:
https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration

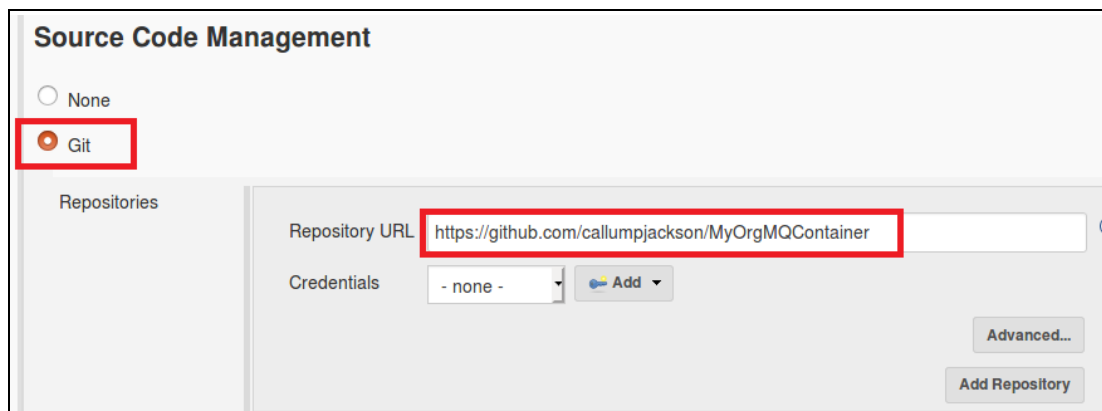


Figure 9-58 Configure the source code management

5. Within the **Build Triggers** section, click **Build after other projects are built** and type **EnterprisePipeline** (Figure 9-59 on page 616):

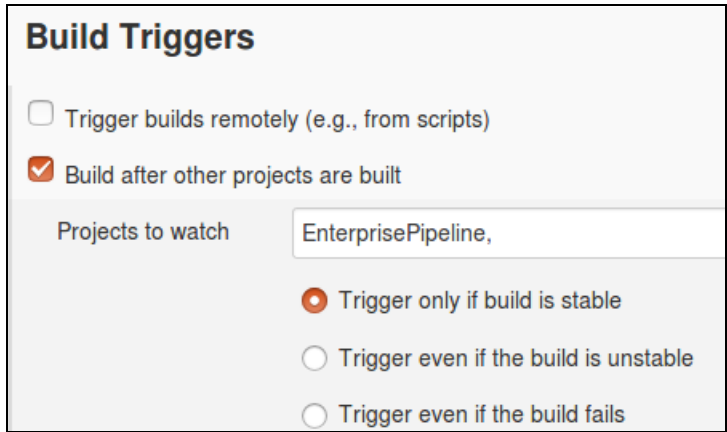


Figure 9-59 Triggering the pipeline from the previous

- In the **Built Environment** section, select the **Delete workspace before build starts** (Figure 9-60):



Figure 9-60 Configure the workspace to be removed before building

- Within the **Build** section click **Add build step** and select **Execute shell** (Figure 9-61):

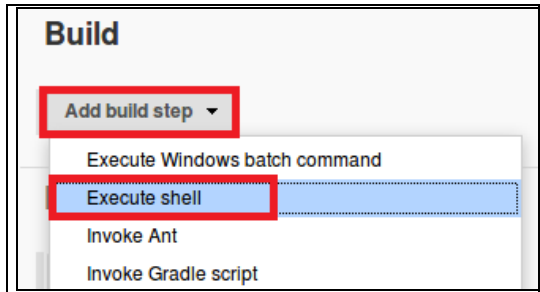


Figure 9-61 Create a new Execute shell

- Copy the following lines into the shell box:


```
cd /chapte9/Dev1Team
docker build --tag app1-mq:${BUILD_NUMBER} .
docker tag app1-mq:${BUILD_NUMBER} app1-mq:latest
```
- Click **Save** to complete the configuration.
- Click **Build Now** to verify that the pipeline successfully works. After the build is complete, click the build name, and select the **Console Output** to verify.

Verification of the build process

After all the pipelines have been created this can be verified by starting the BasePipeline, which will trigger the next pipeline automatically until an application container is built.

- Return to the Jenkins dashboard and kick-off a new build for the BaseMQPipeline as shown in Figure 9-62.

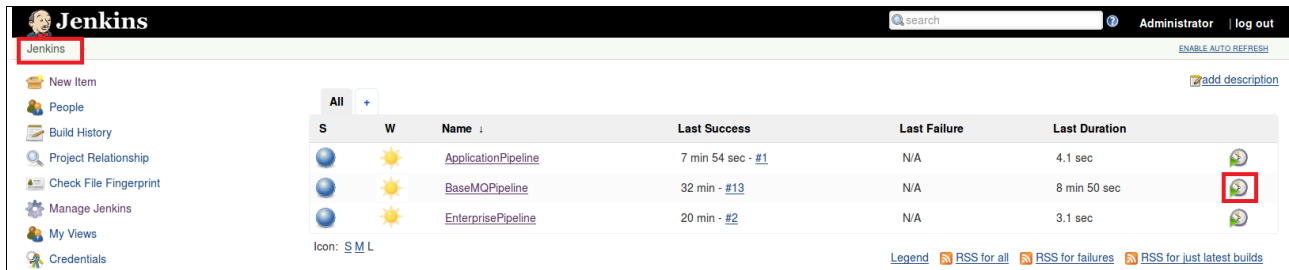


Figure 9-62 Verify the entire pipeline

- After the build has completed, the **Last Success** time should have been updated. See Figure 9-63.

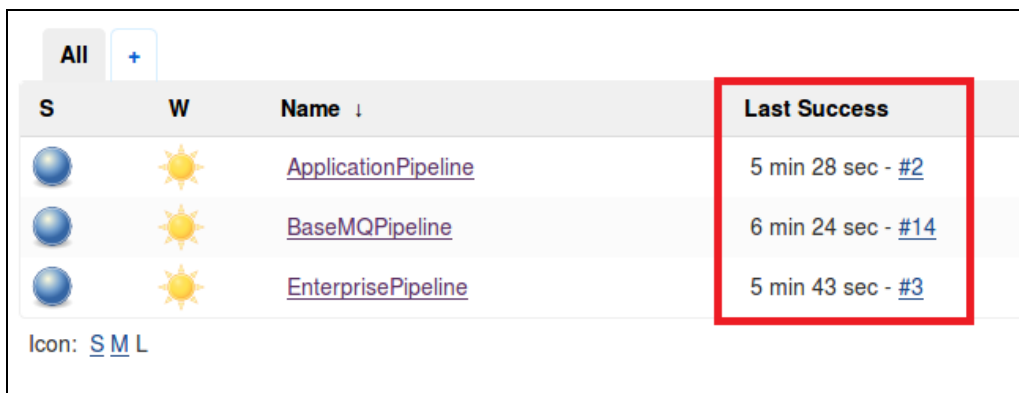


Figure 9-63 Updated build of the entire pipeline

- The latest created container image will now be started, run the following command from a new terminal to start the queue manager:

```
docker run --env LICENSE=accept --env MQ_DEV=false --env MQ_QMGR_NAME=QM1
--publish 1414:1414 --publish 9443:9443 --detach app1-mq:latest
```

Refer to the IBM MQ container documentation here:

<https://github.com/ibm-messaging/mq-container/blob/master/docs/usage.md> for the documentation on the parameters.

- The preceding command will start the container. To view the running containers, run the following command as shown in Figure 9-64.

```
docker ps
```

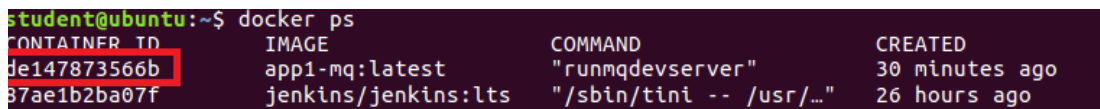


Figure 9-64 List the running containers

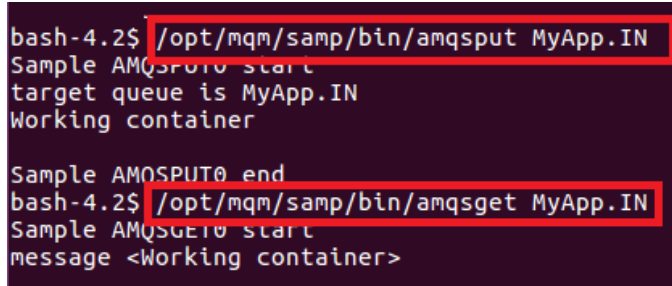
- To verify the behavior of the container a message will be sent and received. An easy mechanism to accomplish this is to use sample IBM MQ utilities shipped with the product. Therefore, exec into the container image by using the Container ID from your output as follows:

```
docker exec -it <CONTAINER_ID> /bin/bash
```

For example, `docker exec -it de147873566b /bin/bash`

6. As part of the build a single queue that is called `MyApp.IN` was created. The IBM MQ samples will be used to send and receive a message, run the following commands (Figure 9-65 on page 618):

```
/opt/mqm/samp/bin/amqsput MyApp.IN  
/opt/mqm/samp/bin/amqsget MyApp.IN
```



```
bash-4.2$ /opt/mqm/samp/bin/amqsput MyApp.IN  
Sample AMQSPUT0 start  
target queue is MyApp.IN  
Working container  
  
Sample AMQSPUT0 end  
bash-4.2$ /opt/mqm/samp/bin/amqsget MyApp.IN  
Sample AMQSGET0 start  
message <Working container>
```

Figure 9-65 Send and receive a message

The preceding verification could be built into the pipeline, so this is automated and would avoid any manual interaction.

Important: This example pipeline built an IBM MQ queue manager without securing the channels with TLS. Configuring the channel with TLS would be set in the IBM MQ configuration via MQSC and stored in `/etc/mqm`.

In addition to setting the channel to use TLS the IBM MQ queue manager would also need to have access to the TLS certificates and keystore. This could be achieved by storing the certificates in locations such as the following examples: a separate filesystem, a cloud key vault mounted into the container at startup, a series of Kubernetes secrets, or even in the container image.

The directory locations and certificate types are documented in the IBM MQ container repository here:
<https://github.com/ibm-messaging/mq-container/blob/master/docs/usage.md#supplying-tls-certificates>



A

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the GitHub material

The web material that is associated with this book is available in softcopy on the internet from the IBM Redbooks GitHub location:

<https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration>.

Cloning the GitHub material

Complete the following steps to clone the GitHub repository for this book:

1. Download and install Git client if not installed from [this web page](#).
2. Run the following command to clone the GitHub repository:

```
git clone
https://github.com/IBMRedbooks/SG248452-Accelerating-Modernization-with-Agile-Integration.git.
```


Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- ▶ *An Architectural and Practical Guide to IBM Hybrid Integration Platform*, SG24-8351
- ▶ *IBM Cloud Private Application Developer's Guide*. SG248441
- ▶ *IBM Z Integration Guide for Hybrid Cloud and the API Economy*, REDP5319

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- ▶ IBM Cloud Pak for Integration Knowledge Center:
<https://www.ibm.com/support/knowledgecenter/en/SSGT7J/overview.html>
- ▶ Agile integration page:
<https://www.ibm.com/cloud/integration/agile-integration>
- ▶ Agile integration eBooklet:
<http://ibm.biz/agile-integration-ebook>
- ▶ API management blog:
<https://developer.ibm.com/apiconnect/blog>
- ▶ Application integration blog:
<https://developer.ibm.com/integration/blog>
- ▶ Messaging and events blog:
<https://developer.ibm.com/messaging/blog>
- ▶ Microservices architecture:
<http://ibm.biz/MicroservicesVsSoa>
- ▶ Cattle not pets approach:
<http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle>
- ▶ ESB patterns:
<http://ibm.biz/FateOfTheESBPaper>

- ▶ API Connect webcast:
<https://developer.ibm.com/apiconnect/2018/12/10/api-management-centralized-or-decentralized>
- ▶ IBM developer works article on whether an enterprise might look to replace EDI with APIs:
<https://developer.ibm.com/apiconnect/2018/06/25/should-business-apis-replace-ed-i/>
- ▶ EDI standards within IBM App Connect:
<https://github.com/ot4i/dfdl-edifact-tutorial>
- ▶ Hybrid Cloud: Multiple deployment modes:
<https://www.ibm.com/blogs/cloud-computing/2018/10/24/what-is-multicloud>
- ▶ Microservices, SOA, and APIs: Friends or enemies?:
<https://developer.ibm.com/tv/microservices-frenemies>
- ▶ The hybrid integration reference architecture:
<http://ibm.biz/HybridIntRefArch>
- ▶ How to write applications with a cloud-native intent (12factor):
<https://www.12factor.net/>
- ▶ cloud-native approach (Netflix example):
<https://www.cloudcomputing-news.net/news/2017/aug/11/netflix-exemplar-blueprint-cloud-native-computing>
- ▶ Continuous Adoption:
<https://ibm.biz/ContinuousAdoption>
- ▶ Kubernetes website:
<https://kubernetes.io/>
- ▶ Ceph Block documentation:
<https://docs.ceph.com/docs/master/>
- ▶ Benefits of containers:
<https://developer.ibm.com/series/benefits-of-containers>
- ▶ A/B testing and canary rollouts- martinFowler.com:
<https://martinfowler.com/bliki/CanaryRelease.html>
- ▶ CircuitBreaker- martinFowler.com::
<https://martinfowler.com/bliki/CircuitBreaker.html>
- ▶ Michael Elder's regrouped version of 12factors:
<https://medium.com/ibm-cloud/kubernetes-12-factor-apps-555a9a308caf>
- ▶ The Practical Test Pyramid by Martin Fowlers:
<https://martinfowler.com/bliki/TestPyramid.html>
- ▶ How to build a proper helm chart for IBM Cloud Private environments:
(<https://github.com/IBM/charts/blob/master/GUIDELINES.md#developing-helm-charts-for-ibm-cloud-private>)
- ▶ Considerations for naming JDBC Providers policy - Knowledge Center link::
https://www.ibm.com/support/knowledgecenter/en/SSTDS_11.0.0/com.ibm.etools.mft.doc/ah61310_.htm

- ▶ Sample IBM App Connect docker image:
<https://github.com/ot4i/ace-docker/>
- ▶ Base code of the helm package for IBM App Connect from GitHub:
<https://github.com/ot4i/ace-helm>
- ▶ Push REST APIs to API Connect - Knowledge Center link::
https://www.ibm.com/support/knowledgecenter/SSTTDS_11.0.0/com.ibm.etools.mft.doc/bn28905_.htm
- ▶ Configuring a JDBC type 4 connection for globally coordinated transactions - Knowledge Center link::
https://www.ibm.com/support/knowledgecenter/SSTTDS_11.0.0/com.ibm.etools.mft.doc/ah61330_.htm
- ▶ urlopen module - Knowledge Center link::
https://www.ibm.com/support/knowledgecenter/SS9H2Y_7.7.0/com.ibm.dp.doc/urlopen_js.html#urlopen.targetformq
- ▶ Messaging using the REST API - Knowledge Center link:
(https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.1.0/com.ibm.mq.dev.doc/q130940_.htm)
- ▶ How to secure an API by using OAuth 2.0 - Knowledge Center link:
https://www.ibm.com/support/knowledgecenter/en/SSFS6T/com.ibm.apic.toolkit.doc/tutorial_apionprem_security_0Auth.html
- ▶ AAA information files - - Knowledge Center link:
https://www.ibm.com/support/knowledgecenter/en/SS9H2Y_7.5.0/com.ibm.dp.doc/aaa_aaainfofile_contents.html
- ▶ How to use IBM App Connect with Salesforce:
(<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-salesforce/>)
- ▶ How to use IBM App Connect with Gmail:
<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-gmail/>
- ▶ How to use IBM App Connect with Slack:
<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-slack/>
- ▶ MQ on Cloud tutorial:
<https://www.ibm.com/cloud/garage/dte/tutorial/tutorial-mq-ibm-cloud>.
- ▶ Batch processing tutorial:
<https://developer.ibm.com/integration/blog/2018/03/16/introducing-batch-processing-in-ibm-app-connect/>
- ▶ How to use IBM App Connect with Salesforce:
(<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-salesforce/>)
- ▶ How to use IBM App Connect with ServiceNow:
<https://developer.ibm.com/integration/docs/app-connect/how-to-guides-for-apps/use-ibm-app-connect-servicenow/>

- ▶ IBM Cloudant Database scenario helm chart:
(<https://github.com/maxgfr/ibm-cloudant>)
- ▶ Salesforce Account and Asset object relationship:
https://trailhead.salesforce.com/en/content/learn/modules/field_service_maint/field_service_maint_assets.
- ▶ IBM Cloud catalog:
(<https://cloud.ibm.com/catalog>)
- ▶ Cloudant documentation:
<https://cloud.ibm.com/docs/services/Cloudant?topic=cloudant-databases>
- ▶ IBM API Connect Test and Monitor:
<https://ibm.biz/apitest>
- ▶ Open source LoopBack framework:
<https://loopback.io/doc/en/lb4/Getting-started.html>
- ▶ IBM Aspera free trial link:
<https://www.ibm.com/cloud/aspera>
- ▶ Slack free trial link::
<https://slack.com>
- ▶ Organization, Catalogue and Space responsibilities for APIs:
<https://developer.ibm.com/apiconnect/2019/07/18/organizingteamsinapic/>
- ▶ Kubernetes documentation:
<https://kubernetes.io/docs/home/>
- ▶ API Connect 2018 Whitepaper:
<https://www.ibm.com/downloads/cas/30YERA2R>
- ▶ IBM API Connect Version 2018 documentation:
https://www.ibm.com/support/knowledgecenter/en/SSMNE2_2018/mapfiles/getting_started.html
- ▶ Service meshes and API management:
▶ <https://developer.ibm.com/apiconnect/2018/11/13/service-mesh-vs-api-management/>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



Redbooks

Accelerating Modernization with Agile Integration

SG24-8452-00

ISBN 0738458368

(1.5" spine)
1.5" <-> 1.998"
789 <-> 1051 pages



Redbooks

Accelerating Modernization with Agile Integration

SG24-8452-00

ISBN 0738458368

(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



Redbooks

Accelerating Modernization with Agile Integration

SG24-8452-00

ISBN 0738458368

(0.5" spine)
0.475" <-> 0.873"
250 <-> 459 pages



Redbooks

Accelerating Modernization with Agile Integration

(0.2" spine)

0.17" <-> 0.473"

90 <-> 249 pages

(0.1" spine)

0.1" <-> 0.169"

53 <-> 89 pages



Accelerating Modernization with Agile

SG24-8452-00

ISBN 0738458368

(2.5" spine)
2.5" <-> mmn.n"
1315 <-> mmm pages



Accelerating Modernization with Agile Integration Agile Integration Architecture

SG24-8452-00

ISBN 0738458368

(2.0" spine)
2.0" <-> 2.498"
1052 <-> 1314 pages





SG24-8452-00

ISBN 0738458368

Printed in U.S.A.

Get connected

