Developing Node.js Applications on IBM Cloud

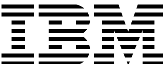
Ahmed Azraq

Mohamed Ewies

Ahmed S. Hassan

In partnership with
IBM **Skills Academy**

Redbooks



International Technical Support Organization

Developing Node.js Applications on IBM Cloud

December 2017

SG24-8406-01

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

Second Edition (December 2017)

This edition applies to IBM SDK for Node.js.

© Copyright International Business Machines Corporation 2017. 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 v
Trademarks vi

Preface vii
Authors vii
Now you can become a published author, too! viii
Comments welcome ix
Stay connected to IBM Redbooks x

Chapter 1. Developing a Hello World Node.js app on IBM Cloud 1
1.1 Getting started 2
1.1.1 Objectives 2
1.1.2 Prerequisites 2
1.1.3 Expected results 3
1.2 Architecture 4
1.3 Step-by-step implementation 4
1.3.1 Set up your IBM Cloud account 4
1.3.2 Log in to your IBM Cloud account 7
1.3.3 Create the Node.js application on IBM Cloud 8
1.3.4 Enable continuous delivery by using toolchain 10
1.3.5 Create a Hello World Node.js server 14
1.3.6 Add a module to the Node.js application 21
1.3.7 Stop the application 25
1.4 Exercise review 25

Chapter 2. Understanding asynchronous callback 27
2.1 Getting started 28
2.1.1 Objectives 28
2.1.2 Prerequisites 28
2.1.3 Background 28
2.1.4 Expected results 29
2.2 Architecture 30
2.3 Step-by-step implementation 31
2.3.1 Log in to your IBM Cloud account 31
2.3.2 Create the Node.js application on IBM Cloud 31
2.3.3 Enable continuous delivery 33
2.3.4 Integrate the Node.js app with the Watson Language Translator service 35
2.3.5 Access the Language Translator service from the Node.js app 41
2.3.6 Access the Language Translator service through a Node.js module 46
2.3.7 Stop the application 52
2.4 Exercise review 52

Chapter 3. Creating your first Express application 53
3.1 Getting started 54
3.1.1 Objectives 54
3.1.2 Prerequisites 54
3.1.3 Expected results 54
3.2 Architecture 56

© Copyright IBM Corp. 2017. All rights reserved.

3.3 Step-by-step implementation 57
3.3.1 Log in to your IBM Cloud account 57
3.3.2 Create the Node.js application on IBM Cloud 58
3.3.3 Create the Hello World Express application 62
3.3.4 Create a simple HTML view and organize the code 67
3.3.5 Integrate with Watson Natural Language Understanding service 74
3.3.6 Deploy the application and run it 81
3.4 Exercise review 88

Chapter 4. Building a rich front-end application by using React and ES6 89
4.1 Getting started 90
4.1.1 Objectives 90
4.1.2 Prerequisites 90
4.1.3 Background concepts 90
4.1.4 Expected results 97
4.2 Architecture 98
4.3 Step-by-step implementation 99
4.3.1 Log in to IBM Cloud 99
4.3.2 Clone the Express application from Git by using the Delivery Pipeline. 100
4.3.3 Create your first React page 106
4.3.4 Add a dynamic form to the page 110
4.3.5 Add more components to the form 115
4.3.6 Using the Fetch API to call the Node.js author service 121
4.4 Exercise review 124

Appendix A. Additional material 127
Locating the material on GitHub 127

Related publications 129
IBM Redbooks 129
Online resources 129
Help from IBM 129

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 will 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.

© Copyright IBM Corp. 2017. All rights reserved.

Trademarks

IBM, the IBM logo, and 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.

Global Business Services®	IBM Watson®	Redbooks (logo) ®
IBM®	Redbooks®	Watson™

The following terms are trademarks of other companies:

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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.

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

Preface

This IBM® Redbooks® publication explains how to create various applications based on Node.js, and deploy and run them on IBM Cloud. This book includes the following exercises:

- Develop a Hello World application in Node.js on IBM Cloud
- Use asynchronous callback to call an external service
- Create an Express application
- Build a rich front-end application by using React and ES6

During these exercises, you will perform these tasks:

- Create an IBM SDK for Node.js application.
- Write your first Node.js application.
- Deploy an IBM SDK for Node.js application on an IBM Cloud account.
- Create a Node.js module and use it in your code.
- Understand asynchronous callbacks and know how to use it to call an external service.
- Understand IBM Watson™ Language Translator service.
- Create a Hello World Express application.
- Create a simple HTML view for your application.
- Understand Express routing.
- Use third-party modules in Node.js.
- Understand IBM Watson® Natural Language Understanding service.
- Use a Git repository on IBM Cloud DevOps services.
- Understand Delivery Pipeline.
- Understand how to clone an IBM Cloud application.
- Use React to create interactive web pages.
- Understand the following concepts of ES6: Classes, arrow functions, and promises.

This book is for beginner and experienced developers who want to start coding Node.js applications on IBM Cloud.

Authors

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Ahmed Azraq is a Cloud Solution Leader in IBM Egypt. He has recently joined the Global IBM Cloud Services and Solutioning, East Hub organization. His primary responsibility is to help clients across the Middle East and Africa (MEA) and Asia Pacific to adopt IBM Cloud and IBM Watson. Since joining IBM in 2012, Ahmed has worked as a technical team leader, and architect in the IBM MEA Client Innovation Center, which is part of IBM Global Business Services® (GBS). Ahmed has several professional certifications, including Open Group IT Specialist, IBM Certified Solution Advisor - Cloud Reference Architecture, IBM Certified Application Developer - Cloud Platform, Java EE, IBM Business Process Manager, Agile development process, and IBM Design Thinking. Ahmed has delivered training on IBM Cloud, DevOps, hybrid cloud Integration, Node.js, and Watson APIs to IBM clients, IBM Business Partners, university students, and professors around the world. He is the recipient of several awards, including Eminence and Excellence Award in the IBM Watson worldwide competition Cognitive Build, the IBM Service Excellence Award for showing excellent client value behaviors, and knowledge-sharing award.

© Copyright IBM Corp. 2017. All rights reserved.

Ahmed has authored several IBM Redbooks publications, including *Building Cognitive Applications with IBM Watson Services: Volume 2 Conversation*, SG24-8394, and *Essentials of Cloud Application Development on IBM Bluemix*, SG24-83742.

Mohamed Ewies is a Certified Expert IT Specialist and IBM Certified Application Developer for Cloud Platform. He has 12 years of experience in developing enterprise applications in IBM Application Middleware Software. He worked as an Application Architect and Technical Team Lead on several large-scale projects. His technical experience includes Java EE, Web/Portal, Cloud, and Application Integration development. He worked on the architecture and implementation of several web applications and proofs of concept on IBM Cloud.

Ahmed S. Hassan has over 11 years experience in information technology. He worked as software developer and integration engineer for many projects in different industries, including electronic design automation, electronic payment, telecommunications, and travel and transportation. Ahmed is an IBM Certified Cloud Application Developer.

The project that produced this publication was managed by Marcela Adan, IBM Redbooks Project Leader, ITSO.

Thanks to the following author of the previous edition of this book:

Ahmed E. Marzouk
IBM Client Innovation Center, IBM Egypt

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

Andrea Emliani
Ossama Hakim
Juan Pablo Napoli
Denny Punnoose
IBM Skills Academy

Aya A. Fathy
Global Business Services, IBM Egypt

Khaled Sallam
Global Business Services, IBM Egypt

Uzma Siddiqui
IBM Hybrid Cloud, IBM US

Arlemi Turpault
IBM Digital Business Group, IBM UK

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 ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will 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, International Technical Support Organization
  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

1

Developing a Hello World Node.js app on IBM Cloud

IBM SDK for Node.js provides a stand-alone JavaScript runtime and server-side JavaScript solution for IBM platforms. It provides a high-performance, highly scalable, event-driven environment with non-blocking I/O that is programmed with the familiar JavaScript programming language. The IBM SDK for Node.js is based on the Node.js open source project.

The Eclipse Orion Web IDE is a web-based, integrated development environment (IDE) where you can create, edit, run, debug, and perform source-control tasks. You can seamlessly move from editing to running, submitting, and deploying.

In this chapter, you install the IBM SDK for Node.js on an IBM Cloud account. You develop a Node.js-based server application (by using the Eclipse Orion Web IDE) that responds to web browser requests.

This chapter contains the following topics:

- Getting started
- Architecture
- Step-by-step implementation
- Exercise review

© Copyright IBM Corp. 2017. All rights reserved.

1.1 Getting started

To start, read through the objectives, prerequisites, and expected results of this use case.

1.1.1 Objectives

Web developers write JavaScript applications to add interactivity to *client-side* web applications. As an interpreted scripting language, developers do not need to use compilers to write applications. The syntax of the programming language is simple enough for web developers with little programming experience to write simple applications.

IBM SDK for Node.js uses the JavaScript programming language for *server-side* applications. Instead of running scripts in a web browser, the node application interprets and runs JavaScript applications on a server. Node.js works on an event-driven model, which means it responds to events through callback functions that Node.js calls when an operation completes.

By completing the steps in this chapter, you install the IBM SDK for Node.js on an IBM Cloud account. You develop a server application that responds to web browser requests.

By the end of this chapter, you should be able to accomplish these objectives:

- Create an IBM SDK for Node.js application.
- Write your first Node.js application.
- Deploy an IBM SDK for Node.js application on an IBM Cloud account.
- Create a Node.js module and use it in your code.

1.1.2 Prerequisites

Before you start, be sure that you meet these prerequisites:

- A valid email account
- A workstation that has these components:
  - Internet access
  - Web browser: Google Chrome or Mozilla Firefox
  - Operating system: Linux, Mac OS, or Microsoft Windows

1.1.3 Expected results

The expected result of this exercise is to have a running Node.js application on IBM Cloud, as shown in Figure 1-1.

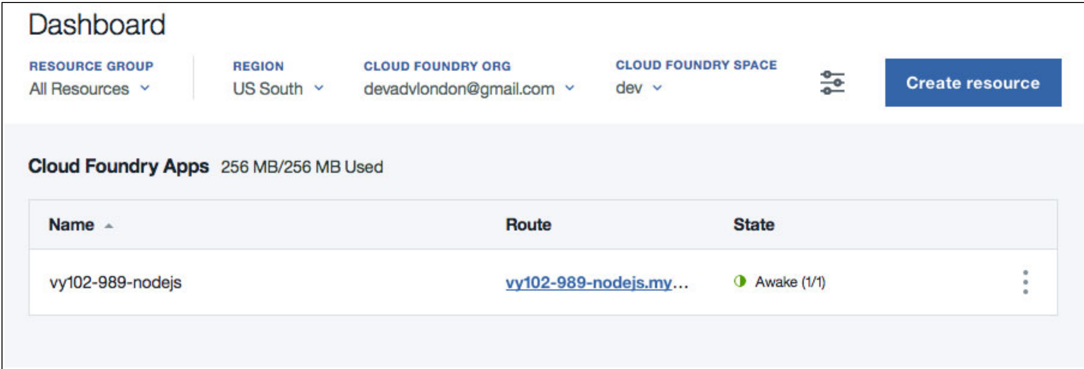


Figure 1-1 Expected results: Node.js app

This application is developed by using Eclipse Orion Web IDE. Eclipse Orion Web IDE is a web-based IDE where you can create, edit, run, debug, and perform source-control tasks.

The Web IDE is part of the IBM Cloud continuous delivery toolchains. Figure 1-2 shows the code.

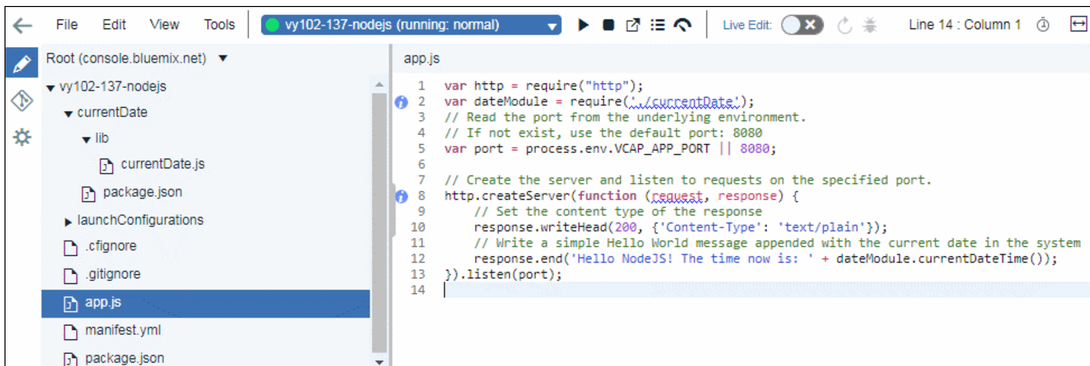


Figure 1-2 Expected results: Node.js code

The application's scope is to show a `Hello NodeJS!` message in the web browser for the user. It will also show the current system date by using a custom Node.js module that you will develop in this exercise. The output is shown in Figure 1-3.

Hello NodeJS! The time now is: Mon Jun 19 2017 23:07:05 GMT+0000 (UTC)

Figure 1-3 Expected results: Hello NodeJS! message and system date and time

1.2 Architecture

The architecture of the Node.js Hello World app is shown in Figure 1-4.

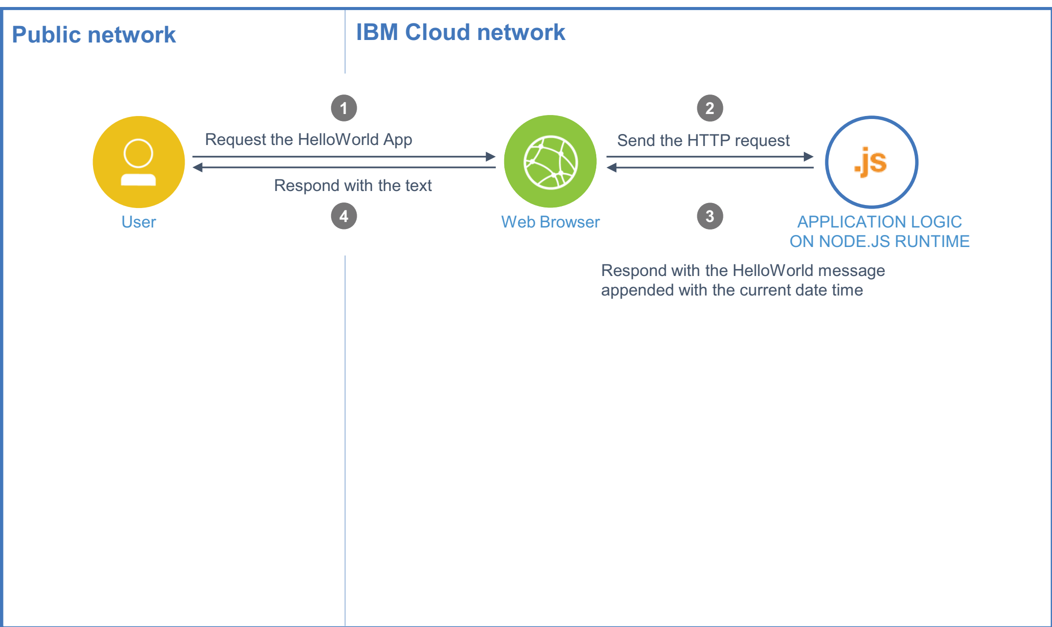


Figure 1-4 Architecture

The following steps explain the sequence of interactions between the components that are used in the exercise:

1. The user accesses the web application in a web browser through a URL provided by IBM Cloud.
2. The web browser sends the HTTP request to the deployed Node.js app in IBM Cloud.
3. The Node.js app listens to the incoming request and responds with a Hello World message that includes the current date and time.
4. The web browser shows the received message to the user.

1.3 Step-by-step implementation

This section describes how to implement the Hello World Node.js app.

1.3.1 Set up your IBM Cloud account

Register with IBM Cloud by providing a valid, unique email address. Your email address acts as your user name for IBM Cloud, and you provide a password of your choice. When you sign up to IBM Cloud, you are prompted for your demographic information (such as your name and company). An email is sent to the email account that you provide in the registration to confirm that your email account is valid and active.

Complete these steps to set up your IBM Cloud account:

1. Open the IBM Cloud console at `http://bluemix.net`.
2. Click **Create a free account**. You are presented with a page similar to Figure 1-5.

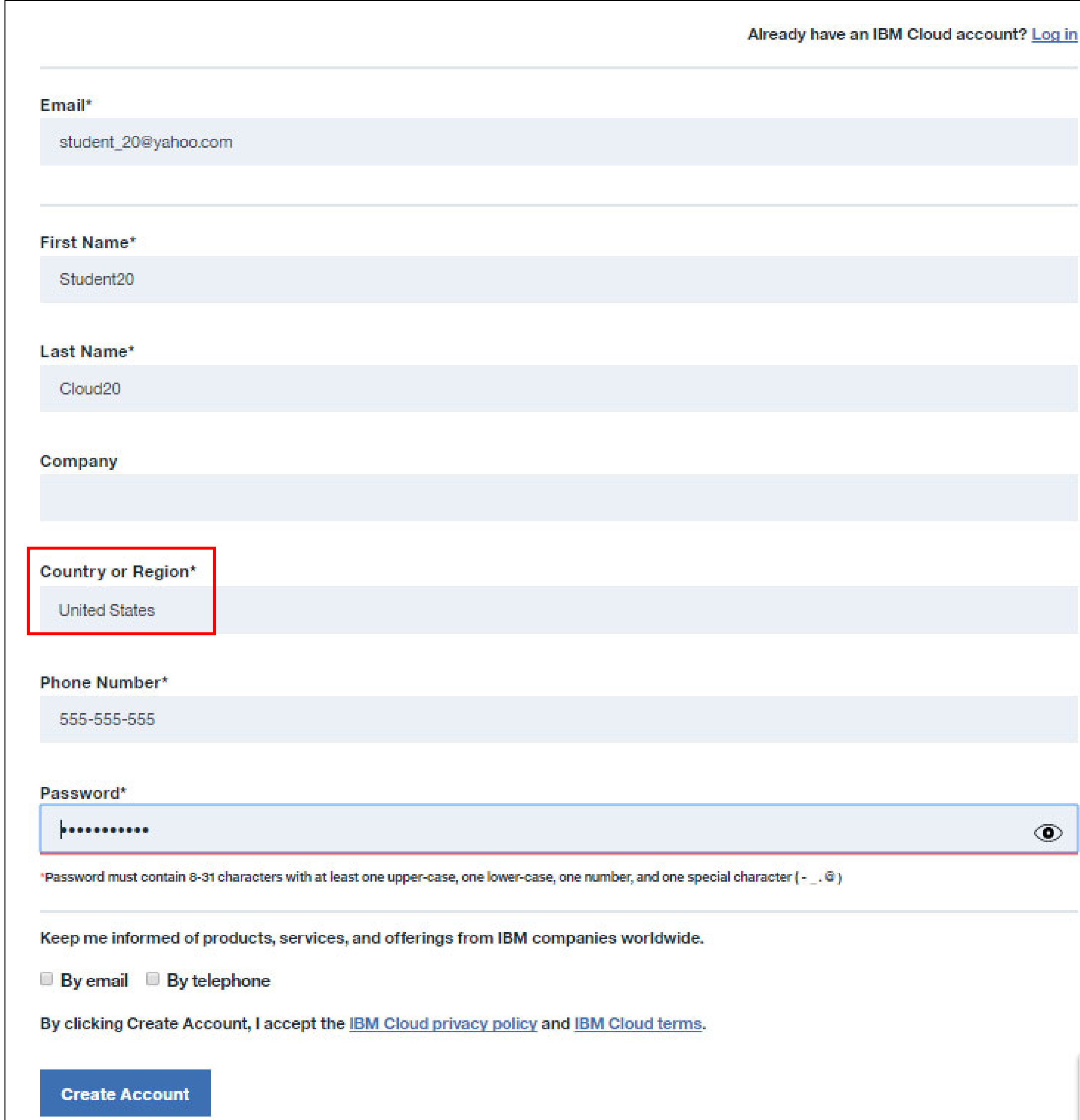


Figure 1-5 IBM Cloud Sign up pane

> **Important note:** Select **United States** for Country or Region. The exercises in this course were developed and tested in the IBM Cloud US South region. You must select United States to ensure that you create the resources in the US South region to be consistent with the services that are used during course development. IBM Cloud assigns you a region that is nearest to the country or region that you specify in your registration form.
>
> If you are physically located in a country that is closer to an IBM Cloud region other than the US South, the closest region might be selected when you log in. Every time you log in to IBM Cloud, check that the US South region was selected and, if not, switch the region to **US South**.

3. Complete the form with your personal information. Note that you must use a valid email address for this course because IBM Cloud sends you an email to verify your account.
4. Click **Create Account**. You are redirected to a page that looks similar to Figure 1-6. Close the page.

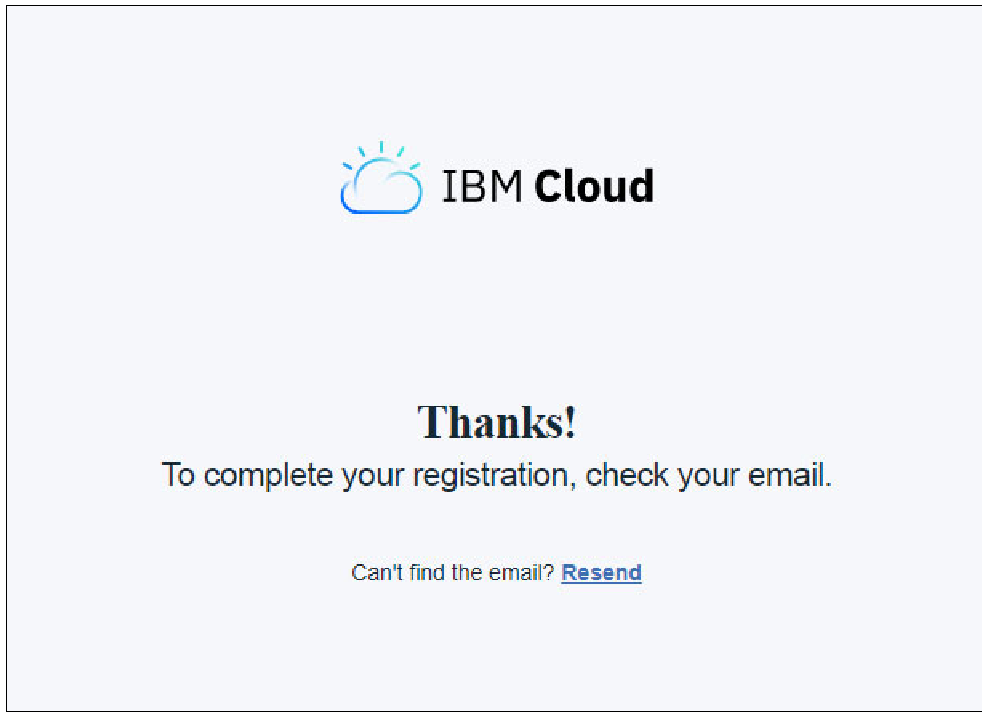


Figure 1-6 Email sent confirmation page

5. Check your email at the email account that you used to sign up to IBM Cloud. You will receive an email similar to the one shown in Figure 1-7.

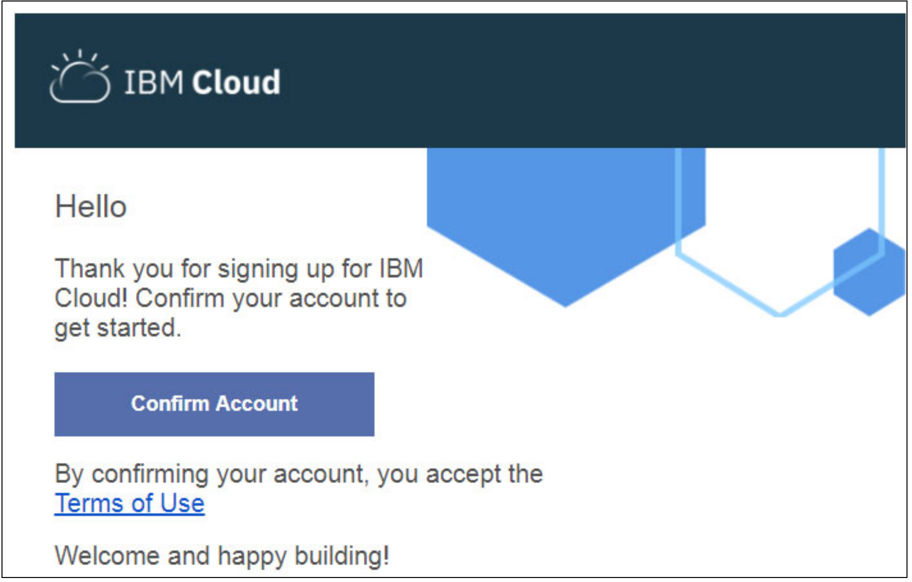


Figure 1-7 Confirm your account

6. Click **Confirm Account**. A page opens that explains that you have now activated your IBM Cloud account.
7. Close this browser or browser tab. You may proceed to the next step.

The page that confirms that your account was activated also includes a **Log in** link. Instead of following this **Log in** link, open a new browser window to experience the regular login to IBM Cloud.

1.3.2 Log in to your IBM Cloud account

Log in to your IBM Cloud account by completing these steps:

1. Open your web browser, enter the following web address, and press **Enter**:

   `https://bluemix.net`

2. The IBM Cloud login page opens (Example 1-8). Click **Log in** and provide your authentication credentials.

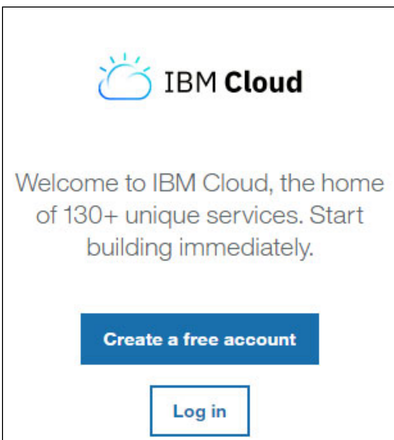


Figure 1-8 IBM Cloud login

1.3.3 Create the Node.js application on IBM Cloud

Create the Node.js app by using the SDK for Node.js runtime on IBM Cloud by completing these steps:

1. In the IBM Cloud Dashboard, click **Create resource** (Figure 1-9).

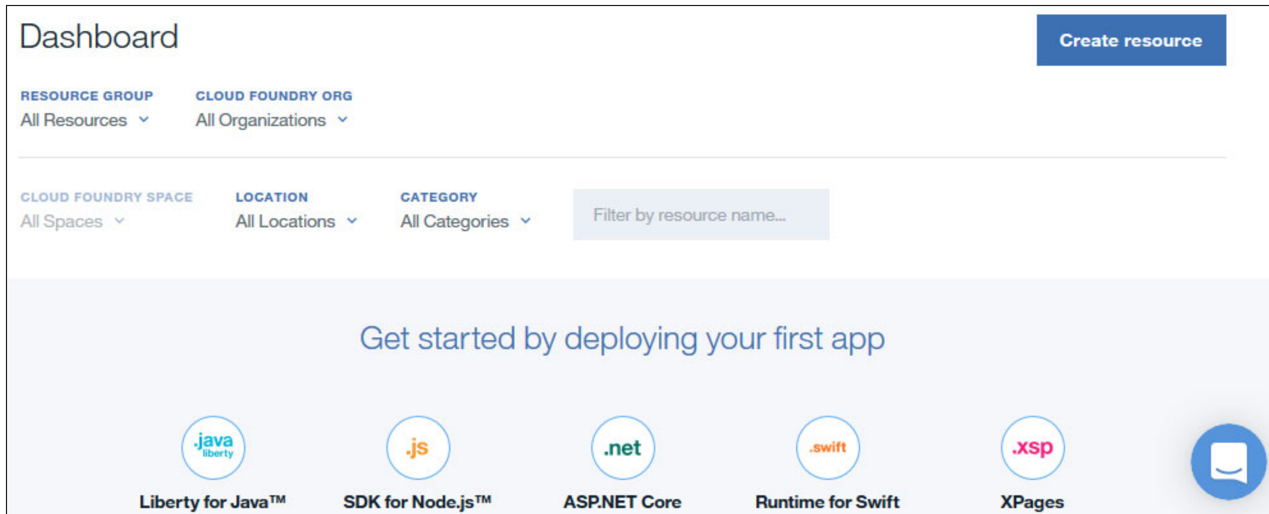


Figure 1-9 Creating the application

2. The IBM Cloud Catalog page opens. It lists the infrastructure and platform resources that can be created in IBM Cloud. Scroll down to the Cloud Foundry Apps section under Platform and click **SDK for Node.js** (Figure 1-10).

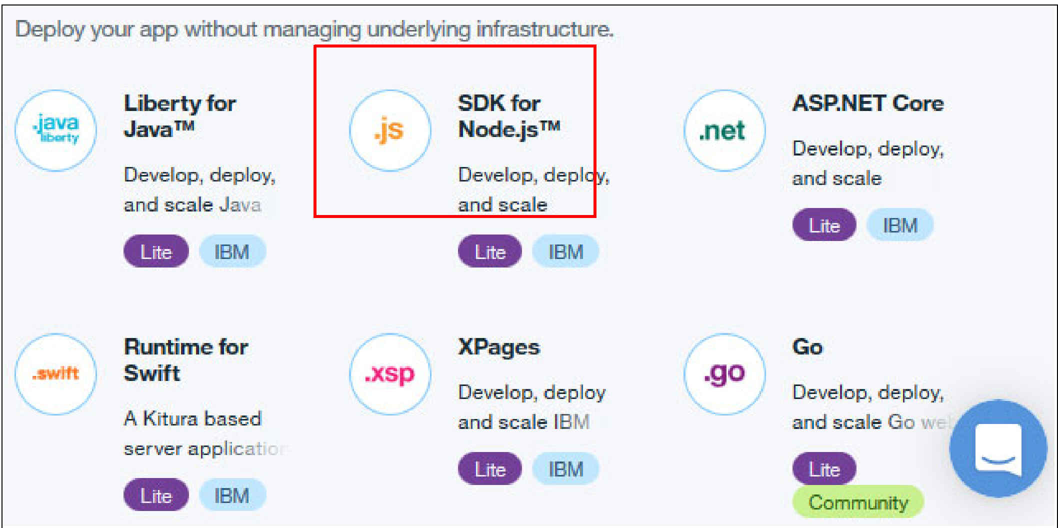


Figure 1-10 IBM Cloud catalog

3. In the **App name** field, enter `vy102-XXX-nodejs`. Replace `XXX` by three random characters that become your unique key (Figure 1-11). You will be using this unique key in the naming convention of this exercise.

   The **Host name** field is automatically populated with the same value as the app name.

   Keep the default values for the other fields.

   In the Pricing Plans section, select **128 MB**.

   > **Note:** If you are physically located in a country that is closer to an IBM Cloud region different from the US South, the closest region might be selected when you log in. Every time you log in to IBM Cloud, check that the US South region was selected and, if not, switch the region to US South.

   Click **Create**.

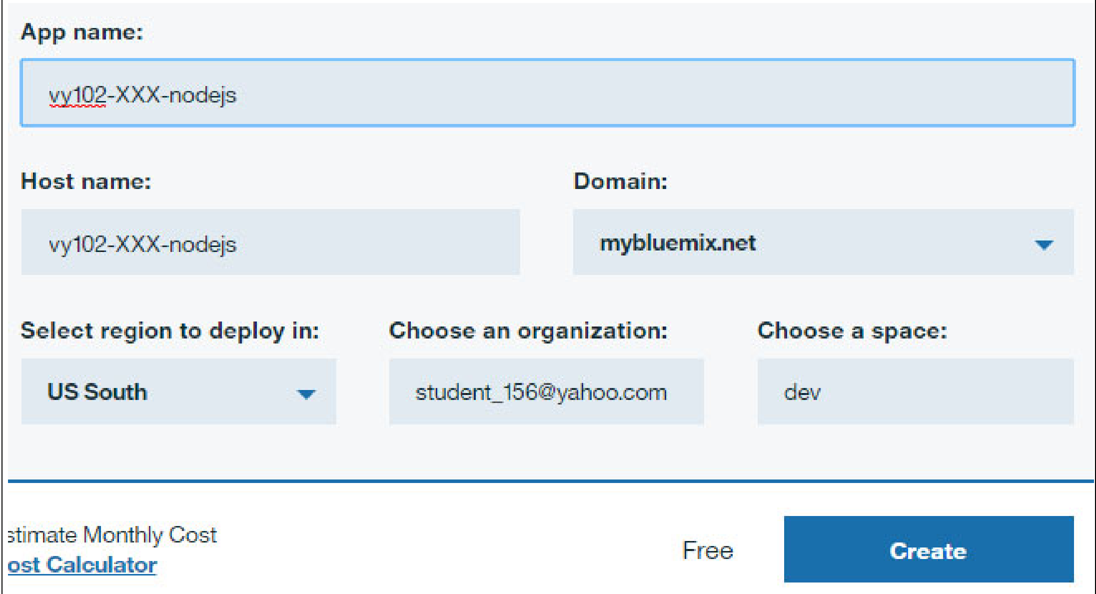


Figure 1-11 Creating the Node.js app

4. The Getting started page for the created application opens (Figure 1-12). The status for `vy102-XXX-nodejs` is shown as `Starting` until the application runs. Wait until the status changes to `This app is awake` (for IBM Cloud Lite accounts) or `Running` (for non-IBM Lite accounts).

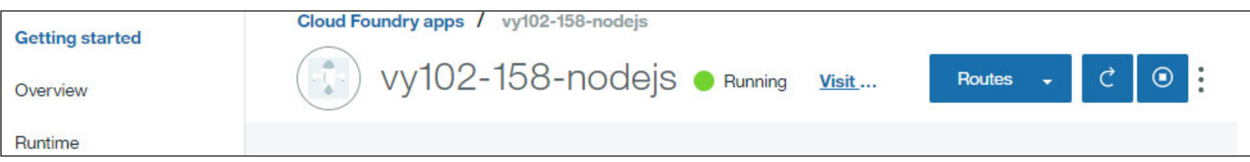


Figure 1-12 Created Node.js App

1.3.4 Enable continuous delivery by using toolchain

The Getting started page of your app shows instructions for accessing your app through the command-line interface (CLI). However, in this exercise, you use continuous delivery.

Enable continuous delivery for the Node.js app by completing these steps:

1. Click **Overview** on the left pane, scroll down to the Continuous delivery tile, and then click **Enable** (Figure 1-13).

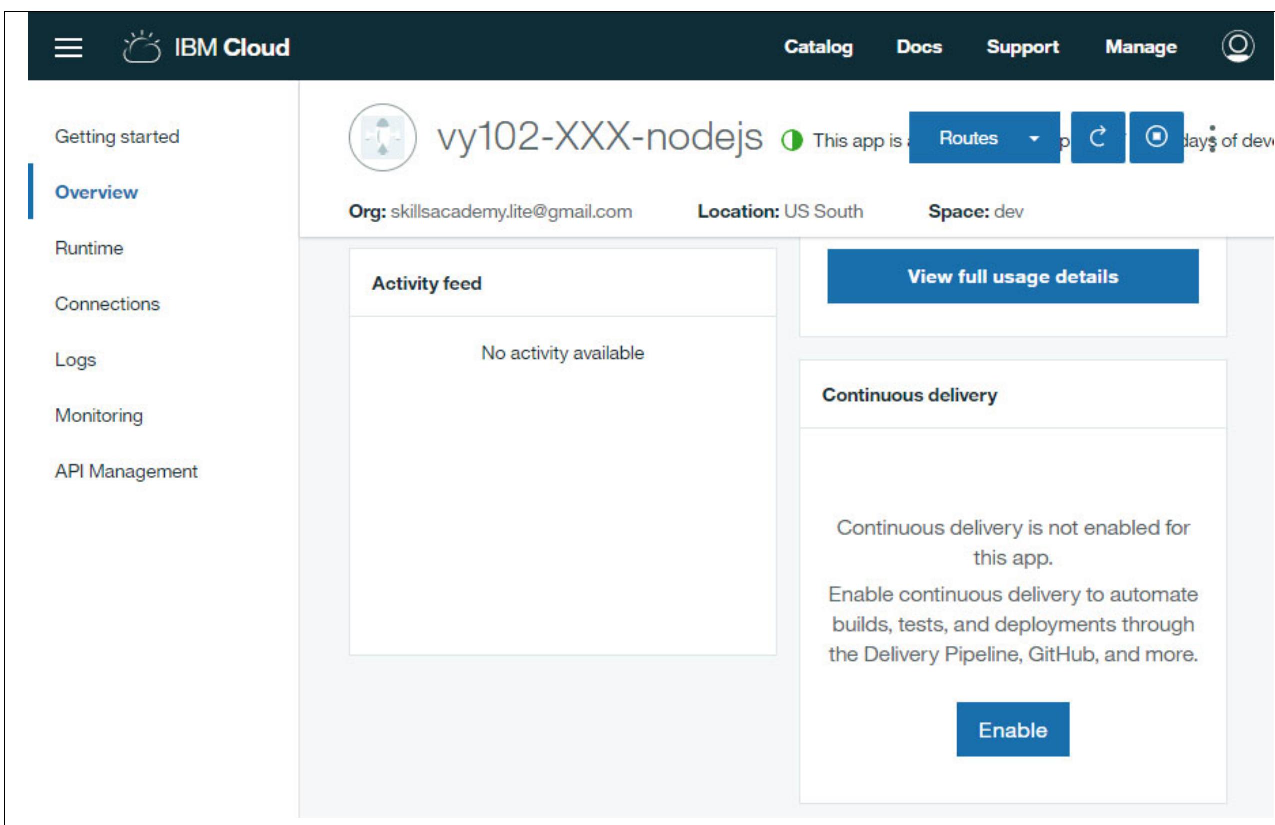


Figure 1-13 Enabling continuous delivery

2. A new Continuous Delivery Toolchain tab opens. This toolchain includes tools to develop and deploy the application.

   The **Toolchain Name** field is automatically populated. Keep the default values for the **Select Region** and **Choose an organization** fields (Figure 1-14).

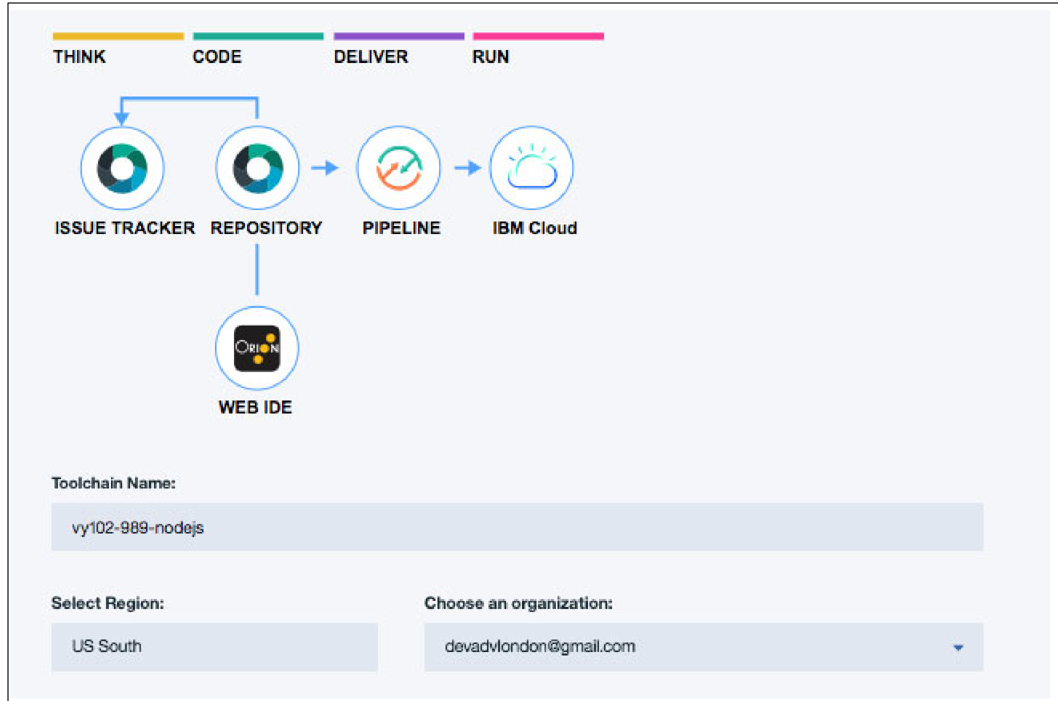


Figure 1-14 Toolchain page

Scroll down to see three main icons (Figure 1-15), which are described later:

– Git Repos and Issue Tracking.
– Eclipse Orion Web IDE.
– Delivery Pipeline.

The **Git Repos and Issue Tracking** icon is selected by default.

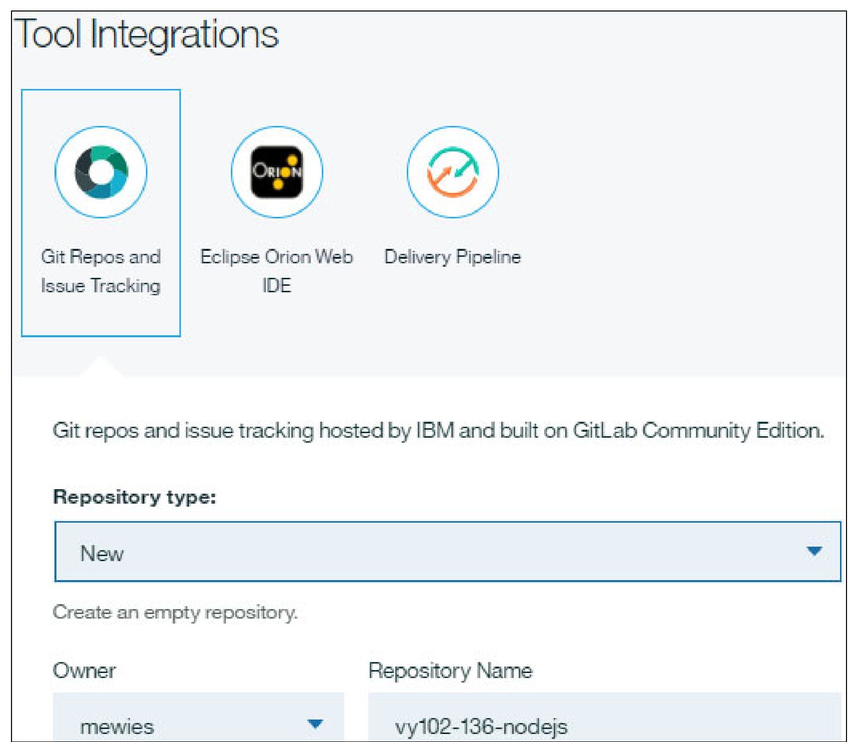


Figure 1-15 Three primary icons: Git Repos and Issue Tracking is selected by default

The **Repository type** field menu lists options to start from your code base:

New Start a new application.

Fork Start from a certain existing repository (you specify its URL), and then have a separate stream of the same repository.

Clone Clone an existing repository to create a new one.

Existing Link to an existing repository and continue working on it.

3. The default selection is **Clone**. For this exercise, select **New**, so that you can start your application from scratch.
4. For the other fields, keep their default values, then click **Create**.

5. A new page opens, showing the three main phases (Figure 1-16). A toolchain is a set of tool integrations that support development, deployment, and operations tasks. The UI to create a new toolchain groups the tools into the following phases:

THINK This phase is for planning the application by creating bugs, tasks, or ideas by using the Issue Tracker, which is part of the Git repository.

CODE This phase is for the implementation of the application by providing a GIT repository as source code management system, and a Web IDE (Eclipse Orion) to edit your code online. In the repository, you can specify whether to clone a repository or start from scratch by selecting **New** in the repository type.

DELIVER This phase is for configuring the delivery pipeline. It allows you to specify automatic build, deployment, and testing of your code after a developer pushes new code to the Git repository.

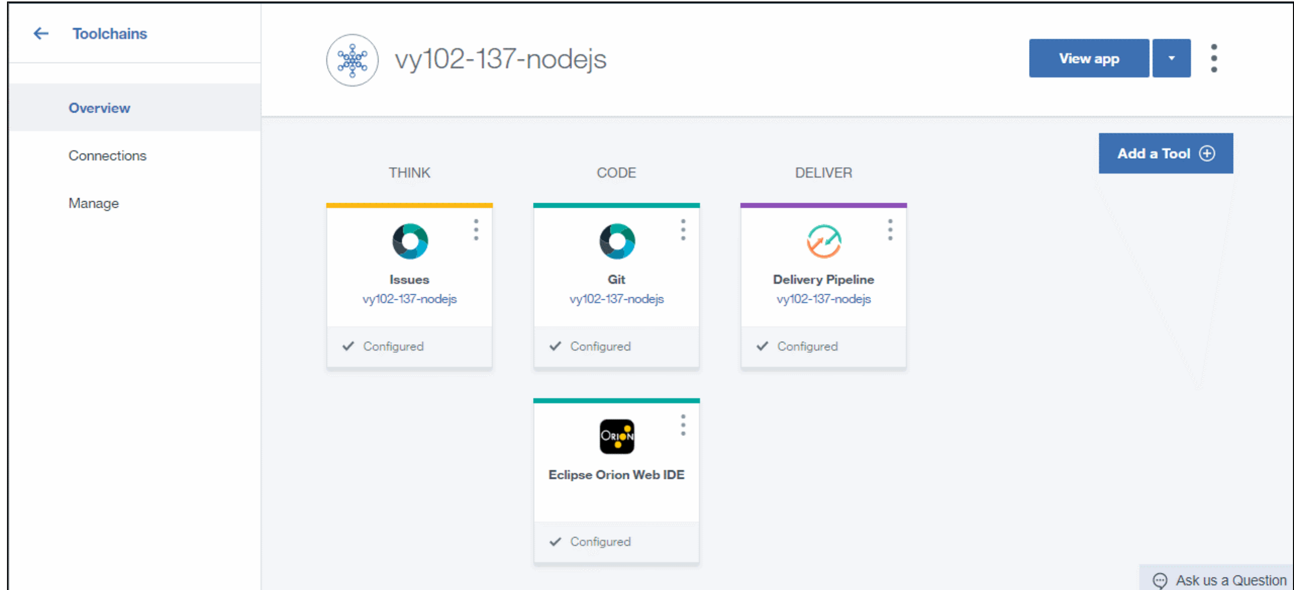


Figure 1-16 THINK, CODE, and DELIVER

1.3.5 Create a Hello World Node.js server

The following steps describe how to write Node.js code from Eclipse Orion Web IDE, and how to link this code to the Node.js app on IBM Cloud that you created in the previous sections:

1. On the Toolchain page, click the **Eclipse Orion Web IDE** icon (Figure 1-16 on page 13).
2. The page now shows the generated Node.js project through the Eclipse Orion Web IDE (Figure 1-17).

   The Eclipse Orion Web IDE is a browser-based development environment where you develop for the web. You can develop in JavaScript, HTML, and CSS with the help of content-assist, code-completion, and error-checking.

   The left side of the current page shows the project structure. Currently, no Node.js files are available. In the next steps, you create these files one by one.

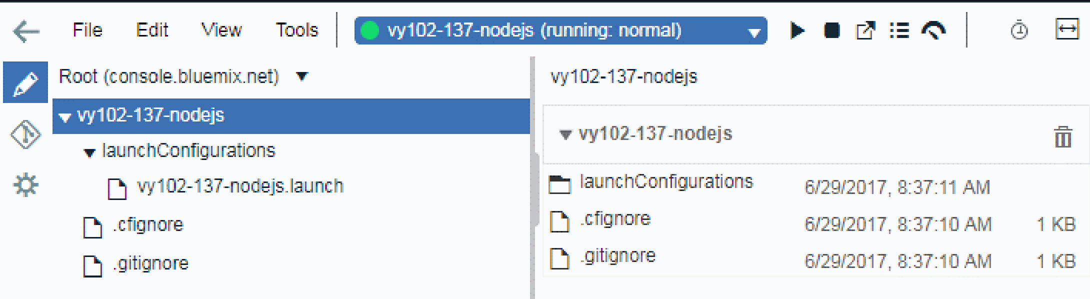


Figure 1-17 Eclipse Orion Web IDE

3. Right-click the root of the project (named `vy102-XXX-nodejs`) from the project structure on the left, and then select **New** → **File** (Figure 1-18).

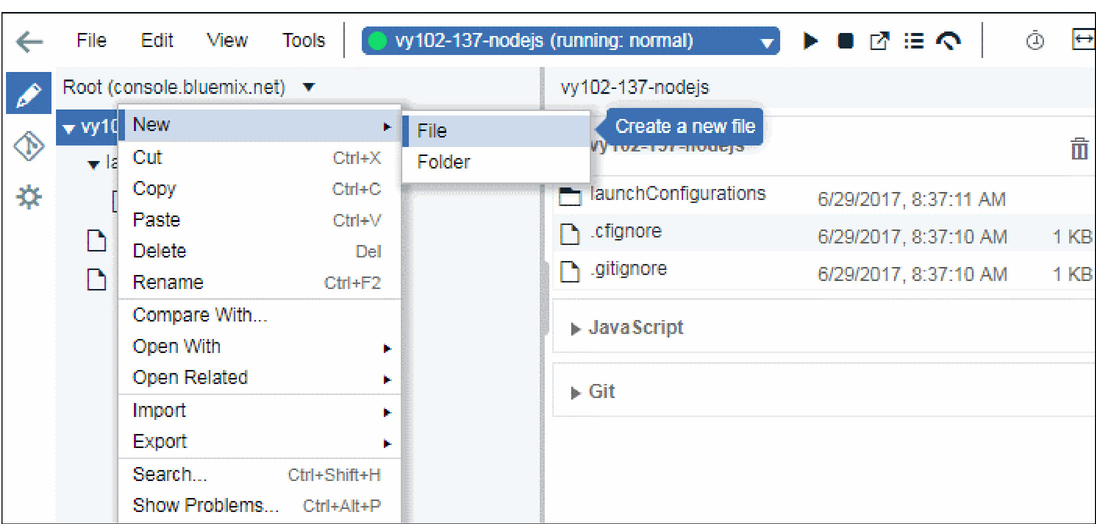


Figure 1-18 Creating a file

4. A text field is displayed (Figure 1-19). Type `manifest.yml` and then press **Enter**.

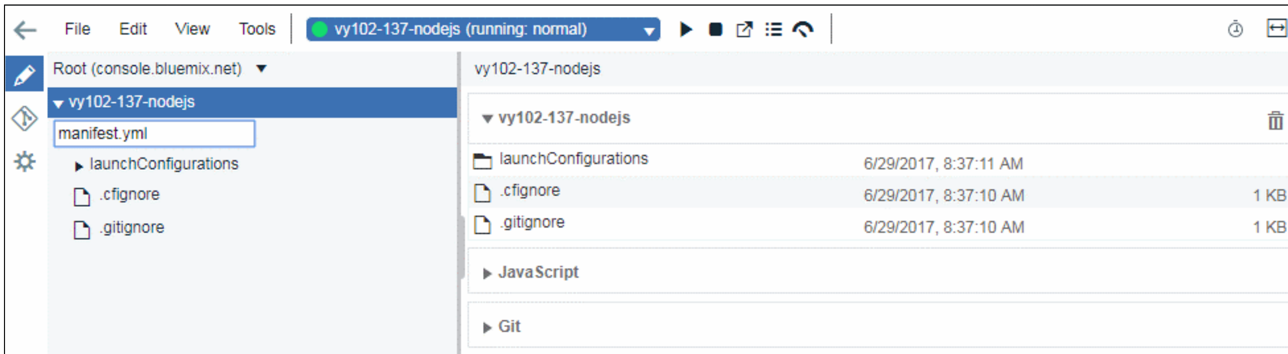


Figure 1-19 Creating the manifest.yml file

The `manifest.yml` file is now created (Figure 1-20).

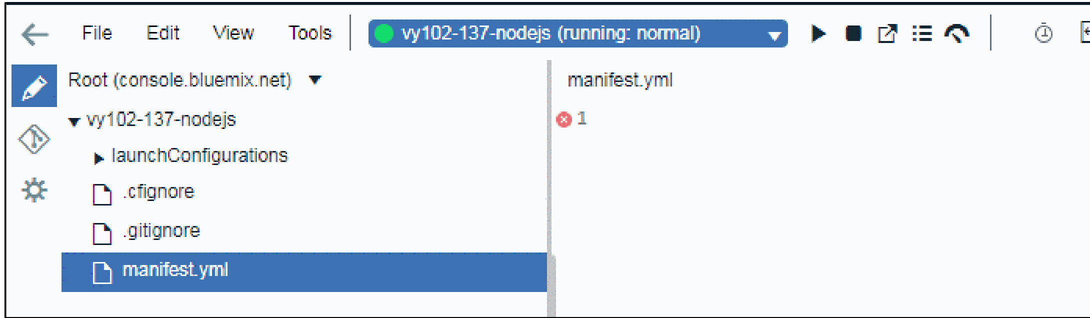


Figure 1-20 Viewing the manifest.yml file

The `manifest.yml` file contains information about the deployment of the application to IBM Cloud.

Add the code snippet from Example 1-1 to `manifest.yml`.

Example 1-1 Code snippet: Application details

```
applications:
- path: .
memory: 256M
instances: 1
domain: mybluemix.net
name: vy102-XXX-nodejs
host: vy102-XXX-nodejs
disk_quota: 1024M
```

Note: Replace the `XXX` with your unique key (three random characters) that you chose in step 3 on page 9 to give a unique name to your Node.js app and host.

Table 1-1 explains the meaning of each attribute in the `manifest.yml` file.

Table 1-1 manifest.yml file attributes

Attribute	Description
path	Indicates to Cloud Foundry the directory where the application is located.
memory	Specifies the memory limit for all instances of the application.
instances	Specifies the number of app instances that are needed for the application. A value of one instance is sufficient for this exercise.
domain	The Cloud Foundry domain to which you are deploying the application.
name	The name of the application that you specified when you created the Cloud Foundry app in IBM Cloud.
host	The subdomain where the application is available.
disk_quota	Specifies the allocation amount of disk space for the app instance.

5. If required, change the domain based on your IBM Cloud region as listed in Table 1-2.

Note: If you followed the instructions for this exercise, you should be in the US South region.

Table 1-2 IBM Cloud regions and domains

Region	Domain
US South	mybluemix.net
United Kingdom	eu-gb.mybluemix.net
Sydney	syd.mybluemix.net
Germany	eu-de.mybluemix.net

6. Save the file by clicking **File** → **Save**.
7. Right-click the root of the project (named `vy102-XXX-nodejs`), select **New** → **File**, name the file `package.json`, and then press **Enter**.

   The `package.json` file holds various metadata relevant to the project. For example, in Example 1-2, the “`start`” field specifies `app.js`, which is the starting point (entry JS file) for this application. The `package.json` file also specifies the dependencies on other Node.js modules.

   The `package.json` file is used by node package manager (NPM), which is the default package manager for the JavaScript runtime environment, Node.js.

   NPM provides two main functions:

   – Online repositories for Node.js packages/modules, which are searchable at the Node.js website at https://nodejs.org/en/.

   – A command-line utility to install Node.js packages, and perform version management and dependency management of Node.js packages.

   NPM accesses the `package.json` file to perform tasks such as registering the application by using the `name` field in the `package.json` file, making sure the dependencies in the `package.json` file are available in the Node.js online repository with the specified versions, and so on.

Insert the code snippet from Example 1-2 in the `package.json` file.

Example 1-2   Code snippet for package.json file

```
{
       "name": "NodejsStarterApp",
        "version": "0.0.1",
        "description": "A Hello World NodeJS sample",
        "scripts": {
      "start": "node app.js"
   }
}
```

The main attributes of the `package.json` file are described in Table 1-3.

Table 1-3   Main attributes

Attribute	Description
name	The name of your node application. The name and version together form an identifier that is assumed to be completely unique.
version	The version of the current application. The version must always be in the form of n.n.n (that is, 1.0 is not an allowed value).
description	A simple description about the application.
scripts	A dictionary property containing script commands that are run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point. In this case, the “start” event is needed to refer to the **`node app.js`** command.

Save the file by clicking **File** → **Save**.

8. Create a file and name it `app.js`. This is the entry JavaScript file for your application.
9. Insert the line from Example 1-3 into the `app.js` file.

Example 1-3   Add variable

```
var http = require("http");
```

Figure 1-21 shows the variable in IBM Cloud DevOps.

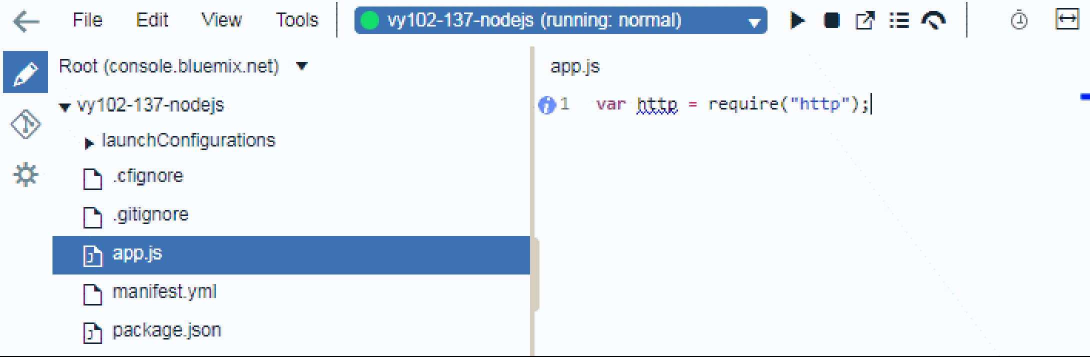


Figure 1-21   The app.js file: Importing the HTTP module

The `require` statement is used to import node modules that are managed by NPM.

The name of the built-in module, http, is managed by NPM. This module is mainly used to allow Node.js to transfer data over the Hypertext Transfer Protocol (HTTP).

10. Enter the code snippet from Example 1-4 to create the server. This server is expected to handle the HTTP requests coming from the client.

Example 1-4   Code snippet: Creating the server

```
// Read the port from the underlying environment.
// If not exist, use the default port: 8080
var port = process.env.VCAP_APP_PORT || 8080;

// Create the server and listen to requests on the specified port.
http.createServer(function (request, response) {

}).listen(port);
```

Your `app.js` file, which now looks like Figure 1-22, shows the createServer function.

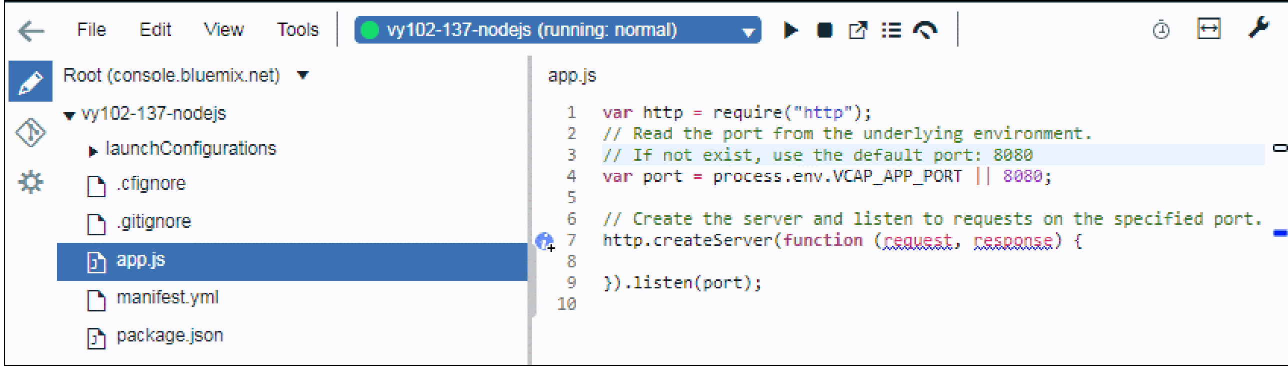


Figure 1-22   The createServer function

As shown, the HTTP module has a createServer callback function, which is responsible for creating a server. This server provides a callback that receives two parameters:

– `request`: This object contains the HTTP request details from the client. You should be able to read parameters from this request to use in your application.

– `response`: This object is created by the HTTP module and you add the response details to it. Then, the HTTP module sends the response object back to the client as an HTTP response to the original HTTP request from the client.

The server listens to a port variable. The port variable is set by the following value:

```
process.env.VCAP_APP_PORT
```

The `process.env` property returns an object that contains the user's environment. In this case, it contains properties related to the deployed application and its underlying IBM Cloud environment.

The `process.env.VCAP_APP_PORT` port value is provided by IBM Cloud automatically when you create the project.

If the `process.env.VCAP_APP_PORT` is null for some reason (this can happen if you run this application outside the IBM Cloud), then the port variable is set to `8080` (port 8080 is the default port for Node.js).

11. Inside the function of the createServer, add the code snippet from Example 1-5.

Example 1-5 Code inside createServer function

```
// Set the content type of the response
response.writeHead(200, {'Content-Type': 'text/plain'});
```

This line has the response's header details:

- Status code: Identifies whether the request was a success, a bad request, a forbidden request, and so on. The code for success is `200`.
- Content type: Identifies the type of content returned in the body of the response. The content type can be text/plain, text/html, and other types. In this scenario, content type is set as `text/plain`.

12. Add the code snippet from Example 1-6 to the end of the createServer function.

Example 1-6 Code snippet: Response - Hello NodeJS

```
// Write a simple Hello World message,
// which will be shown in the user's web browser
response.end('Hello NodeJS!');
```

The end function is used to add the text passed as an argument to the body of the response.

The `app.js` file now looks like the one in Figure 1-23, showing the added response.

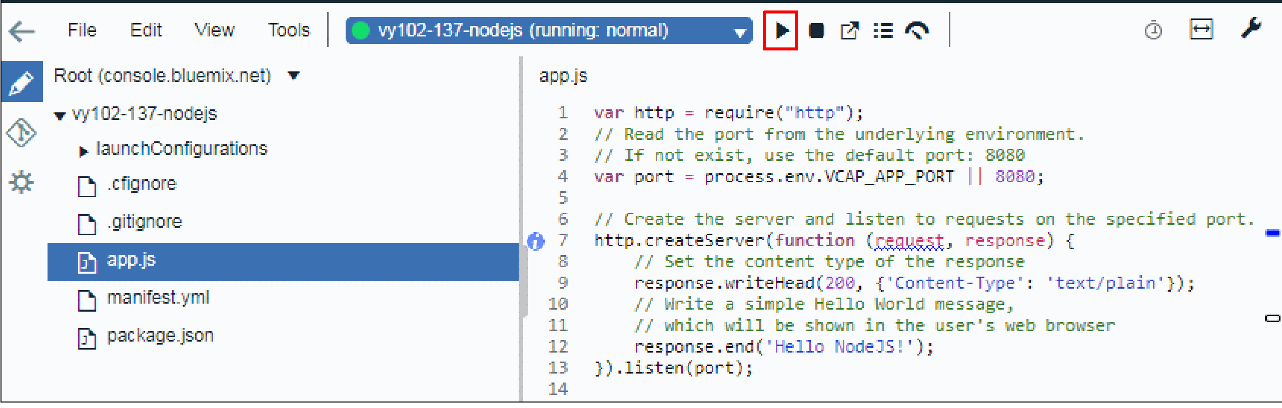


Figure 1-23 The app.js file: Added response

Save the file by clicking **File** → **Save**.

13. In the server toolbar, select **Create new launch configuration** from the drop-down list and click the "**+**" button to display the Edit Launch Configuration window. If you do not have the **Create new launch configuration** option, skip this step.

    In the Edit Launch Configuration window, ensure that the **Organization** is set to your email address, **Space** is set to `dev`, and click **Save**.

14. Now, you can click the play button (**Deploy the App from the Workspace**), highlighted in Figure 1-23. Clicking this button deploys the application to IBM Cloud.

    You might receive a notification warning you that your application will be redeployed. Click **OK** to confirm.

The deployment status indicates that deployment is in progress (Figure 1-24).

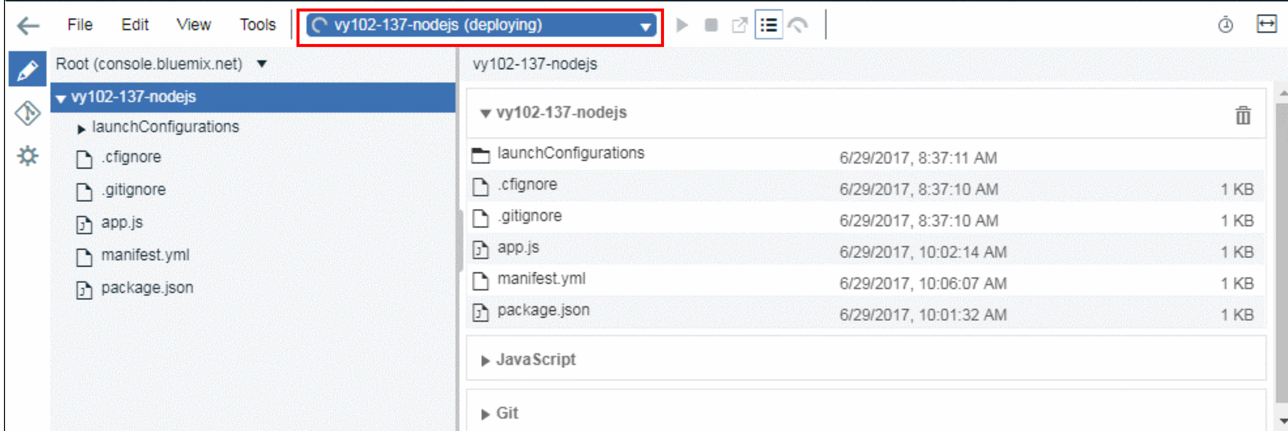


Figure 1-24 Pressing the Deploy the App from the Workspace button

15. After the deployment is complete, click **Open the Deployed App** (Figure 1-25).

> **Note:** If you receive an error message about a timeout, refresh the page.

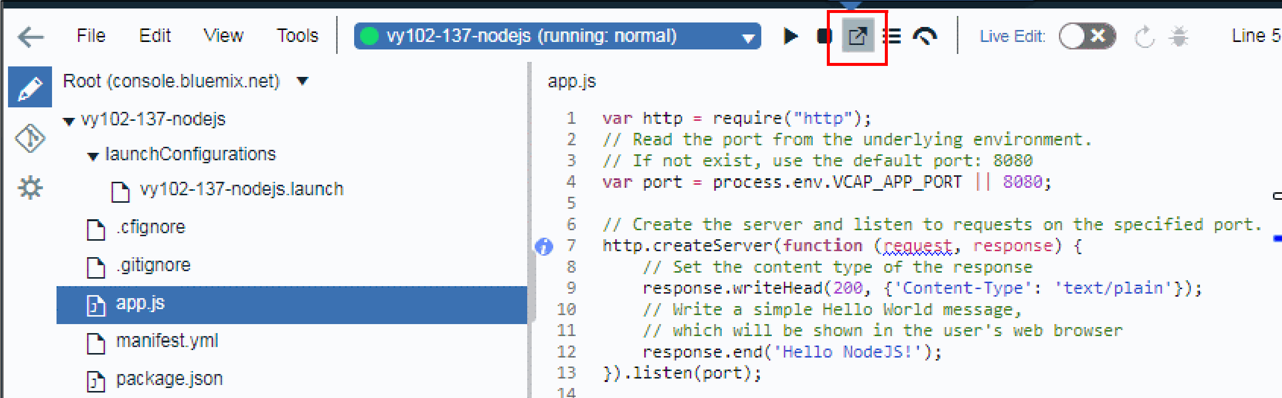


Figure 1-25 Opening the deployed app

A new tab opens and displays the `Hello NodeJS!` message.

1.3.6 Add a module to the Node.js application

In this section, you add a Node.js module to your application.

A module encapsulates related code into a single unit of code. Creating a module means moving the related functions into one file. This point is illustrated next with an example involving an application that is built with Node.js.

Several modules are built in Node.js. For example, the HTTP module is a built-in module. The following steps show how to create your own custom module:

1. Close the Running Application tab to return to Eclipse Orion.
2. Right-click the `vy102-XXX-nodejs` folder, and then select **New** → **Folder** (Figure 1-26).

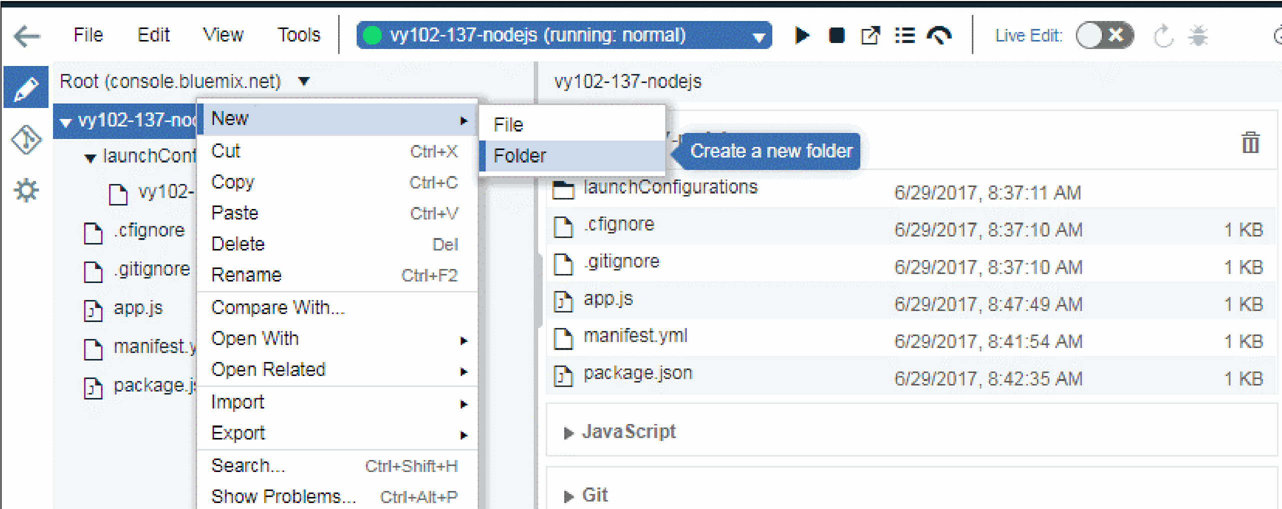


Figure 1-26 Creating a folder

3. Name the new folder `currentDate`.
4. In the `currentDate` folder, create a file named `package.json`.
5. In this `package.json` file, add the code snippet from Example 1-7.

Example 1-7 Code snippet: Add currentDate

```
{
    "name": "currentDate",
    "main": "./lib/currentDate"
}
```

The results are shown in Figure 1-27.

The figure shows that the new `package.json` file represents the new module:

- It has the usual name attribute and another attribute named `main`.
- This main attribute has the path of the JS file that contains the code of this module.
- The value of the main attribute is "`./lib/currentDate`". You need to create a `currentDate.js` file inside the `lib` folder, which you do in the next steps.

Save the file by clicking **File** → **Save**.

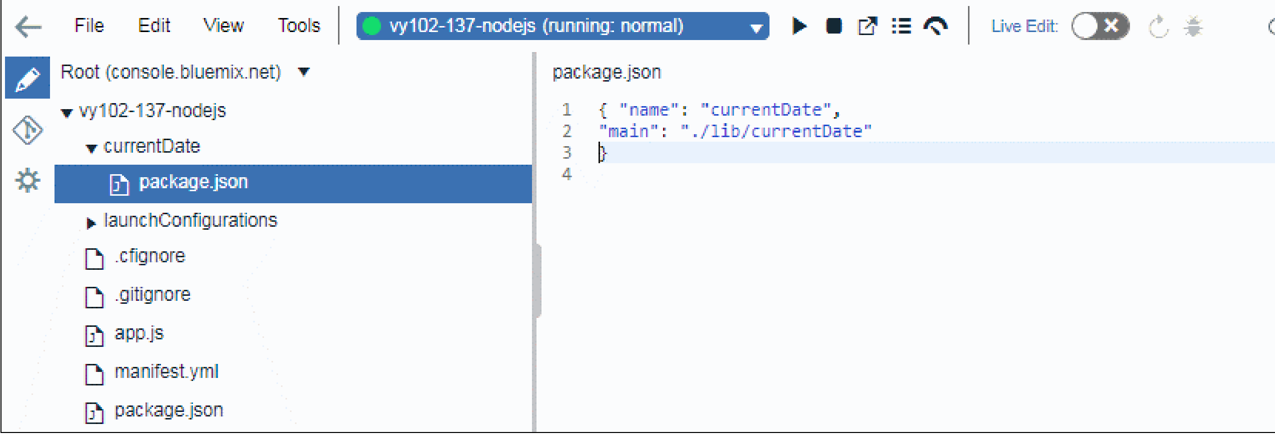


Figure 1-27 The currentDate module in the package.json file

6. In the `currentDate` folder, create a folder named `lib`.
7. In the `lib` folder, create a new file named `currentDate.js`. The result is shown in Figure 1-28.

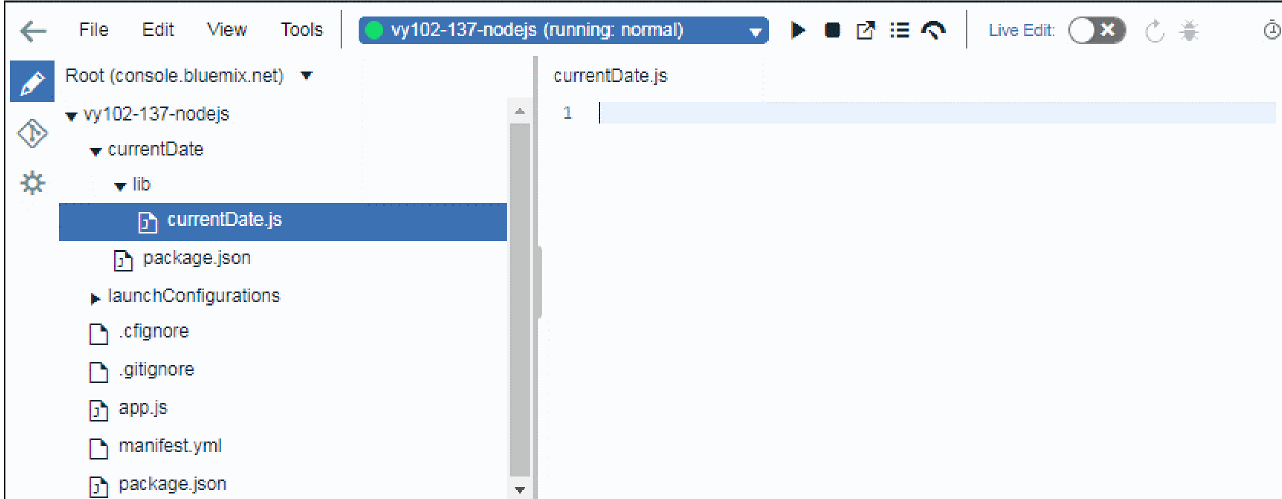


Figure 1-28 The currentDate module in the currentDate.js file

8. In the `currentDate.js` file, enter the code snippet from Example 1-8.

Example 1-8 Code snippet: Export current date time

```
exports.currentDateTime = function() {
    return Date();
};
```

This new currentDateTime function simply returns the current date by using Date().

This function is added to a variable named *exports*, which is used for exposing the function so it can be used by other modules in the system. It is a special object, which by default is included in every JS file in the Node.js application.

Your file now looks like the one shown in Figure 1-29.

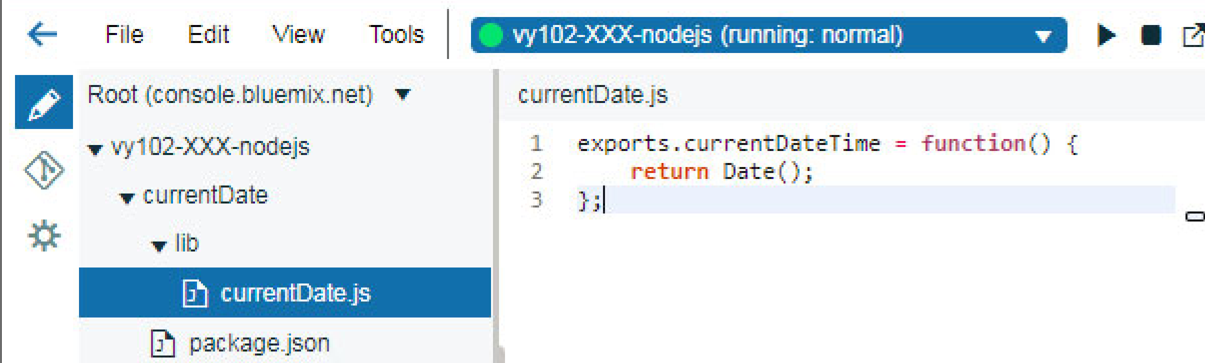


Figure 1-29 The currentDate module: Exporting the currentDateTime function

Save the file by clicking **File** → **Save**.

9. Open the `app.js` file and find the line that is shown in Example 1-9.

Example 1-9 Set variable http

```
var http = require("http");
```

Below that line, add the line from Example 1-10.

Example 1-10 Set variable dateModule

```
var dateModule = require('./currentDate');
```

This line imports the currentDate, which you just created, to your `app.js` file.

> **Note:** In the line, the period and forward slash (`./`) characters indicate that the currentDate module exists in the same folder as the folder that contains this `app.js` file.

10. Find the lines that are shown in Example 1-11.

Example 1-11 Write Hello world message

```
/ Write a simple Hello World message,
// which will be shown in the user's web browser
response.end('Hello NodeJS!');
```

Replace those lines with the lines from Example 1-12.

Example 1-12 Lines to use

```
// Write a simple Hello World message appended with the current date
response.end('Hello NodeJS! The time now is: ' + dateModule.currentDateTime());
```

The result is shown in Figure 1-30.

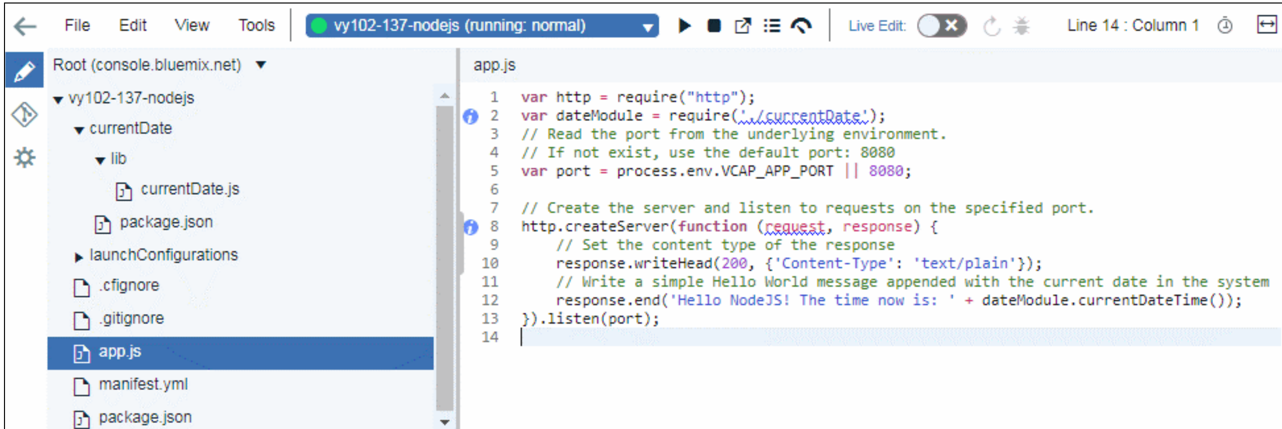


Figure 1-30 Using the currentDate module in the app.js file

All changes are automatically saved in Eclipse Orion. If the changes are not automatically saved or if you want to force a save, select **File** → **Save**.

To deploy, click the play button (**Deploy the App from the Workspace**), as shown in Figure 1-31. You might receive a notification warning you that your application will be redeployed. Click **OK** to confirm.

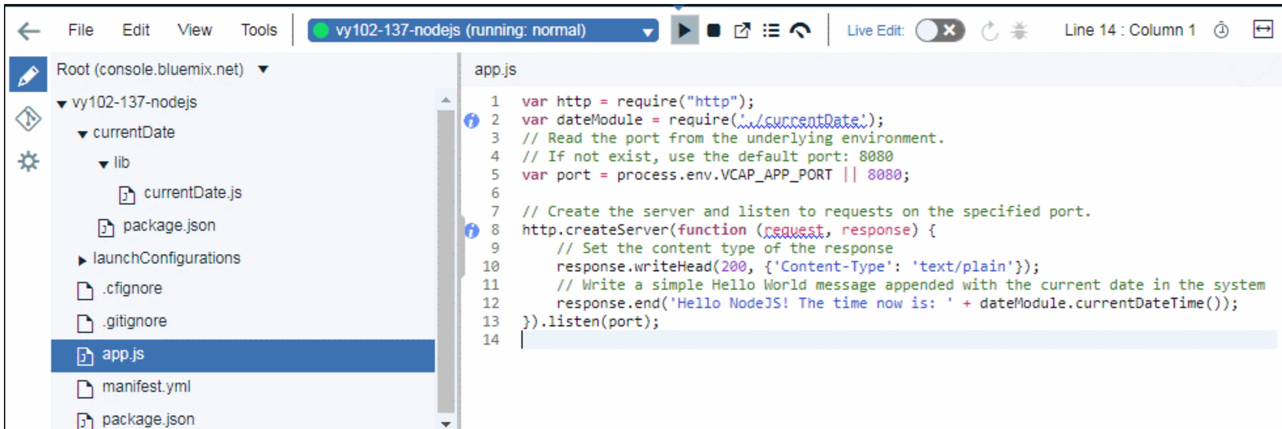


Figure 1-31 Deploying the application that is using the currentDate module

11. After deployment is complete, open the deployed application by clicking **Open the Deployed App** (Figure 1-32).

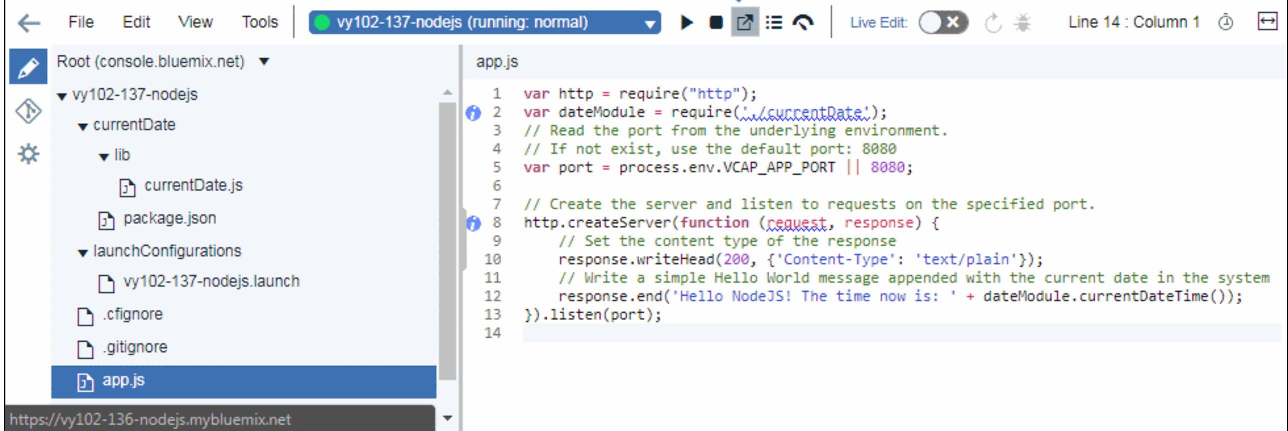


Figure 1-32 Opening the deployed application that is using the currentDate module

Your web browser now opens and displays the result (Figure 1-33).

```
Hello NodeJS! The time now is: Mon Jun 19 2017 23:07:05 GMT+0000 (UTC)
```

Figure 1-33 Results

1.3.7 Stop the application

IBM Cloud Lite account provides you with 256 MB of application memory for Cloud Foundry apps and 100 Cloud Foundry services.

To free the resources assigned to your application, you can either stop your application or delete it:

1. Stop your application by clicking the **Stop the App** icon (Figure 1-34).

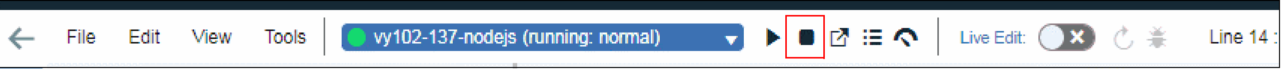


Figure 1-34 Stop the application

2. Close your web browser.

1.4 Exercise review

In this exercise, you accomplished the following goals:

- Created a Node.js App on the IBM Cloud environment.
- Used the Continuous Delivery Toolchain to develop and deploy the Node JS App.
- Wrote your first Node.js “Hello NodeJS!” application.
- Created a module in Node.js to wrap functionality and use it from other JS files.

2

Understanding asynchronous callback

This chapter discusses two important concepts: Asynchronous method invocation and callbacks:

- Asynchronous method invocation

  This is a design pattern where an invoker method does not block while waiting for the invoked method to finish processing and return a result. Instead, the invoked method is run in a separate thread and the invoker is notified when the result is ready.

- Callback function

  This is a function that is passed to another function as a parameter and the callback function is called and run within this other function.

An asynchronous callback has considerations that differ from a synchronous callback. This chapter describes those considerations.

This chapter shows how to use callback functions to call an external service. This exercise uses the IBM Watson Language Translator service in IBM Cloud. You will create a Node.js module that contains the logic for these calls.

This chapter contains the following topics:

- Getting started
- Architecture
- Step-by-step implementation
- Exercise review

© Copyright IBM Corp. 2017. All rights reserved.

2.1 Getting started

To start, read through the objectives, prerequisites, and expected results of this use case.

2.1.1 Objectives

By the end of this chapter, you should be able to understand asynchronous callbacks and be able to write the code in a Node.js application.

2.1.2 Prerequisites

Before you start, be sure that you meet these prerequisites:

- Basic knowledge of JavaScript
- An IBM Cloud account available at `https://console.bluemix.net/`.
- A workstation that has these components:
  - Internet access
  - Web browser: Google Chrome or Mozilla Firefox
  - Operating system: Linux, Mac OS, or Microsoft Windows

2.1.3 Background

In this exercise, you will use asynchronous callback functions.

A callback function is a function that is passed as a *parameter* to another function. The callback function is to be run *only after* the called function finishes running.

Example 2-1 shows an example of a callback function.

Example 2-1 Callback function example

```
setTimeout(function() {
    console.log("A");
}, 3000);

setTimeout(function() {
    console.log("B");
}, 2000);

setTimeout(function() {
    console.log("C");
}, 4000);

setTimeout(function() {
    console.log("D");
}, 1000);
```

The `setTimeout` function in Example 2-1 is a function that receives the following parameters:

- A callback function that should be run after a certain time interval.
- The time interval (in milliseconds) that should elapse before the callback function can run.

When calling the `setTimeout` function, Node.js is not waiting for the processing to finish. Instead, Node.js registers the callback function for it to be run after the response is returned.

When the code in Example 2-1 runs, the following output is returned in the console:

```
D
B
A
C
```

The result depends on the sequence of the `setTimeout` functions finishing their execution and not on the sequence of their declaration in the JavaScript file.

2.1.4 Expected results

This exercise shows how to use callback functions to call an external service (this exercise uses the Watson Language Translator service on IBM Cloud).

The logic of these calls will be in a Node.js module that you will create. Figure 2-1 shows a callback function to call the Language Translator service.

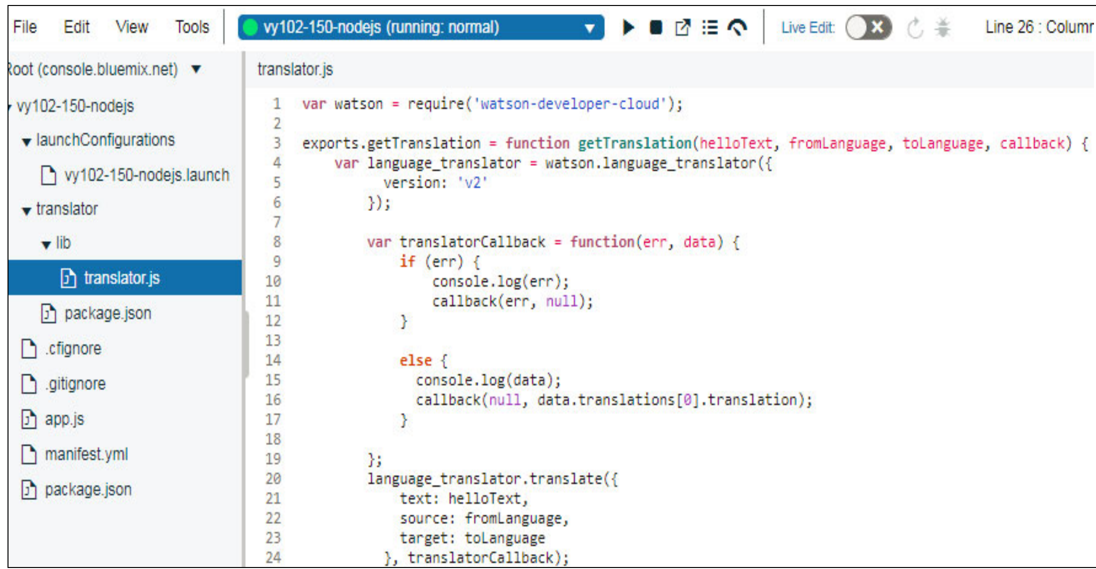


Figure 2-1 Expected results

The expected result of running this function is to see the translation of a word from English to Spanish using IBM Watson Language Translator service. (Figure 2-2).

```
Translation of Hello is Hola
```

Figure 2-2 Example results

2.2 Architecture

The architecture of the asynchronous callback in this chapter is shown in Figure 2-3.

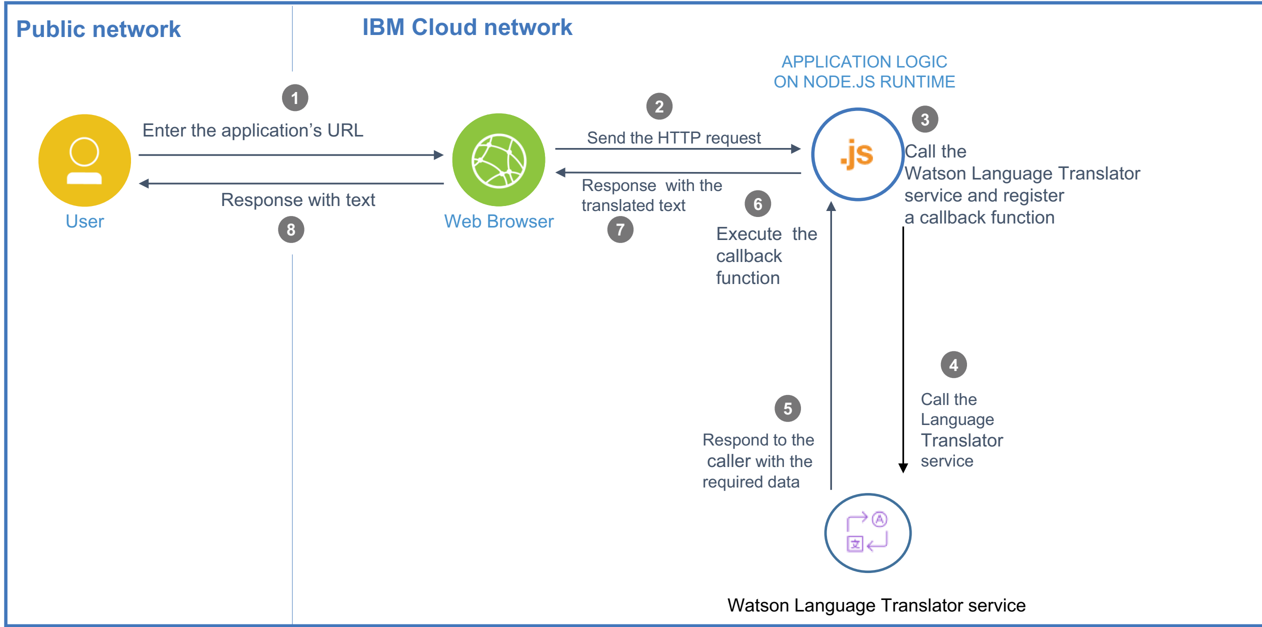


Figure 2-3 Asynchronous Callback sample architecture diagram

This exercise shows how to use callback functions in your Node.js application. In this scenario, the Node.js app accesses the IBM Watson Language Translator service in IBM Cloud. The flow that is shown in Figure 2-3 is as follows:

1. The user enters the application's URL in the web browser.
2. The web browser sends the HTTP request to the Node.js app that is deployed in IBM Cloud.
3. The Node.js app asynchronously calls the Language Translator service and registers a callback function that is to be called when the Language Translator service sends the response.
4. The Node.js app sends an HTTP request to the Language Translator service on IBM Cloud that is exposed as a REST API.
5. The Language Translator service responds to the HTTP request with the requested data (translated text).
6. The callback function (from step 3) is now run. This function responds to the HTTP request (from step 2).
7. The Node.js app sends the data to the web browser in the HTTP response.
8. The web browser displays a web page that shows the data to the user.

2.3 Step-by-step implementation

This section describes how to make asynchronous callback calls by using Node.js.

2.3.1 Log in to your IBM Cloud account

Follow these steps to log in to IBM Cloud:

1. Open your web browser, enter the following web address, and then press **Enter**:

   `https://bluemix.net`

2. The IBM Cloud login page opens (Figure 2-4). Click **Log in** and provide your account credentials.

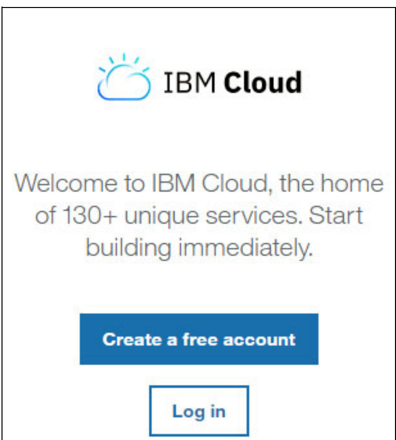


Figure 2-4 IBM Cloud login

2.3.2 Create the Node.js application on IBM Cloud

Create the Node.js app by using the SDK for Node.js runtime on IBM Cloud:

1. In the IBM Cloud dashboard, click **Create resource** (Figure 2-5).

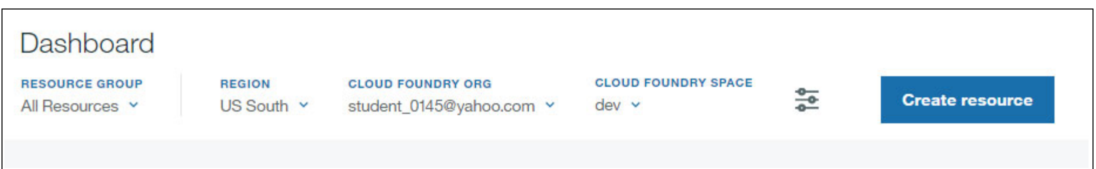


Figure 2-5 Create resource

2. The IBM Cloud Catalog page opens (Figure 2-6). It lists the infrastructure and platform resources that can be created in IBM Cloud. Scroll down to the Cloud Foundry Apps section and click **SDK for Node.js**.

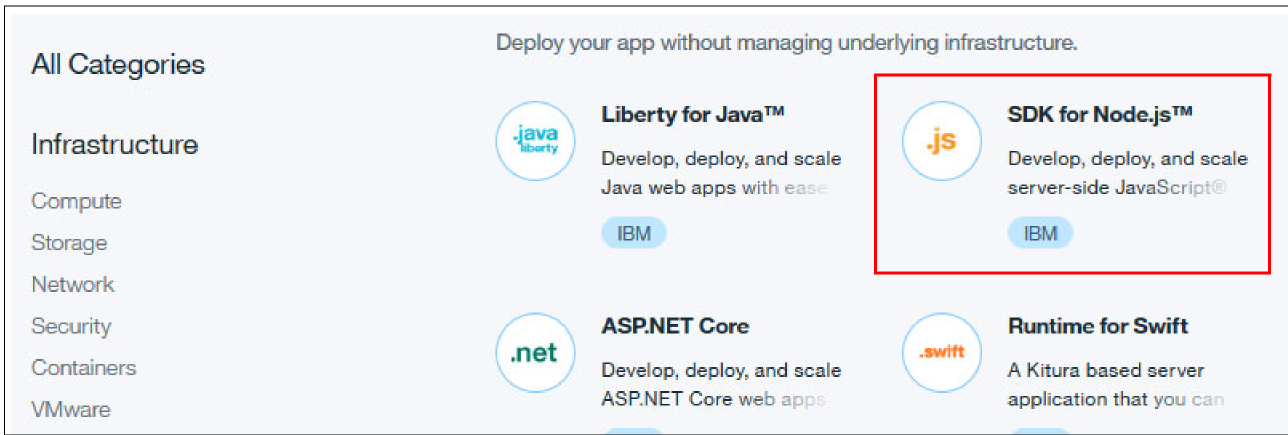


Figure 2-6 IBM Cloud Catalog

3. In the **App name** field, enter `vy102-XXX-nodejs`. Replace `XXX` with any three random characters, which become your unique key (Figure 2-7). Make sure that the name of this application is different from the name of the application that you created in Chapter 1, “Developing a Hello World Node.js app on IBM Cloud” on page 1. Take note of the random characters because you will be using your unique key in the naming convention of this exercise.

   The **Host name** field is automatically populated with the same value as the app name.

   Keep the default values for the other fields.

   Click **Create**.

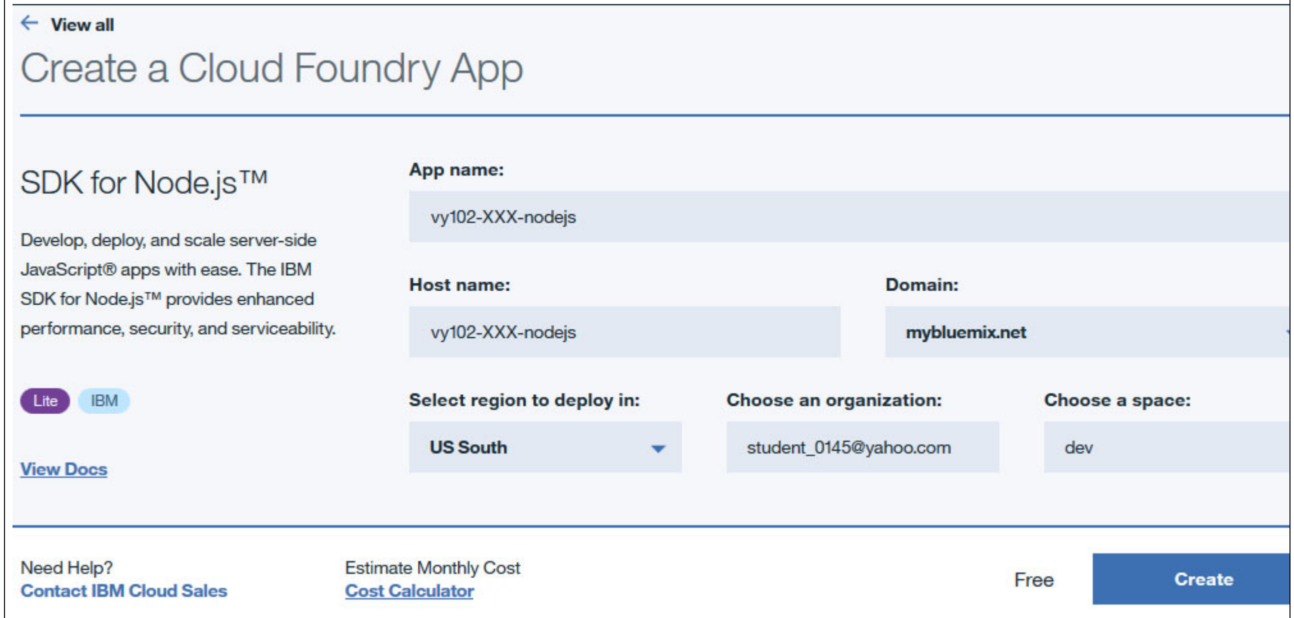


Figure 2-7 Creating the Node.js app

4. The Getting started page for the created application opens (Figure 2-8). For a while, the status for `vy102-XXX-nodejs` is shown as `Starting`. Wait until the status changes to `This app is awake` (for IBM Cloud Lite accounts) or `Running` (for non-IBM Lite accounts).

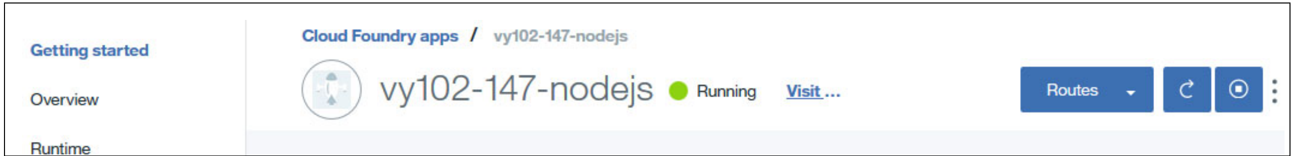


Figure 2-8 Created Node.js app

2.3.3 Enable continuous delivery

Enable continuous delivery for the Node.js app by completing these steps:

1. Click **Overview** on the left pane, scroll down to the Continuous delivery tile, and then click **Enable** (Figure 2-9).

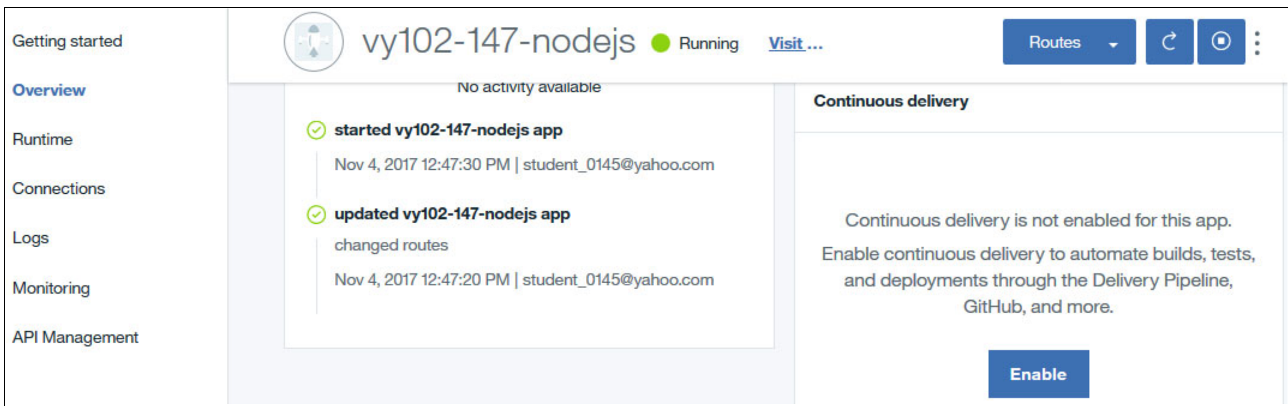


Figure 2-9 Enabling continuous delivery

A new continuous delivery toolchain tab opens (Figure 2-10).

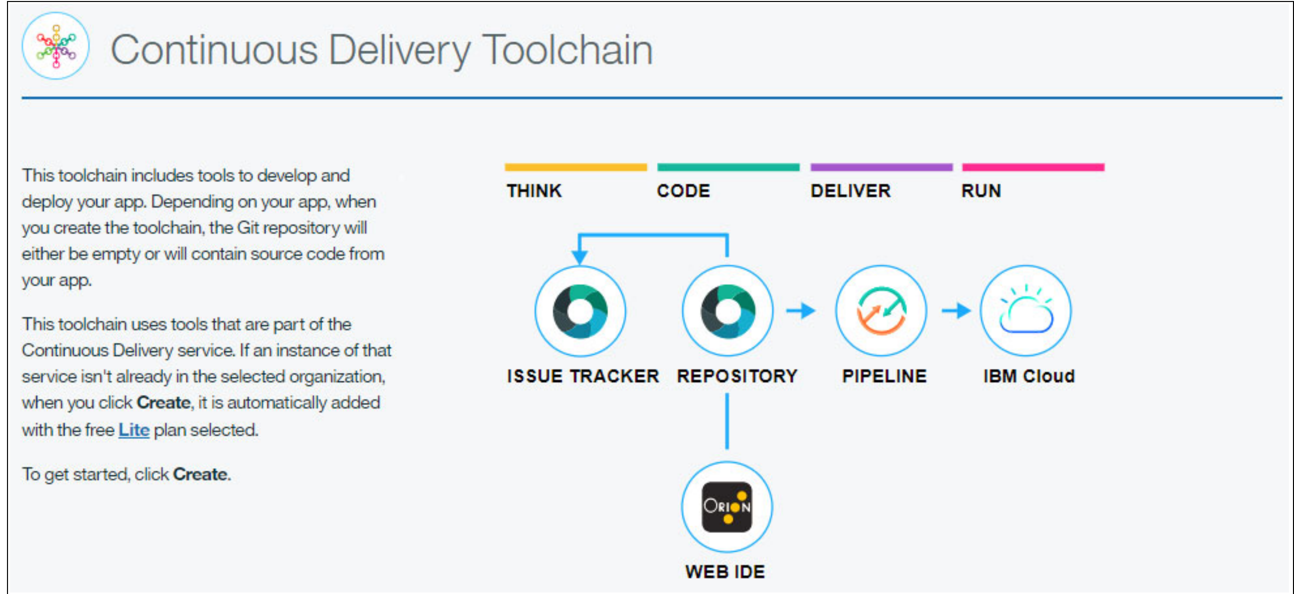


Figure 2-10 Toolchain creation page

Scroll down to see the three main icons shown in Figure 2-11:

– Git Repos and Issue Tracking
– Eclipse Orion Web IDE
– Delivery Pipeline

The **Git Repos and Issue Tracking** icon is selected by default.

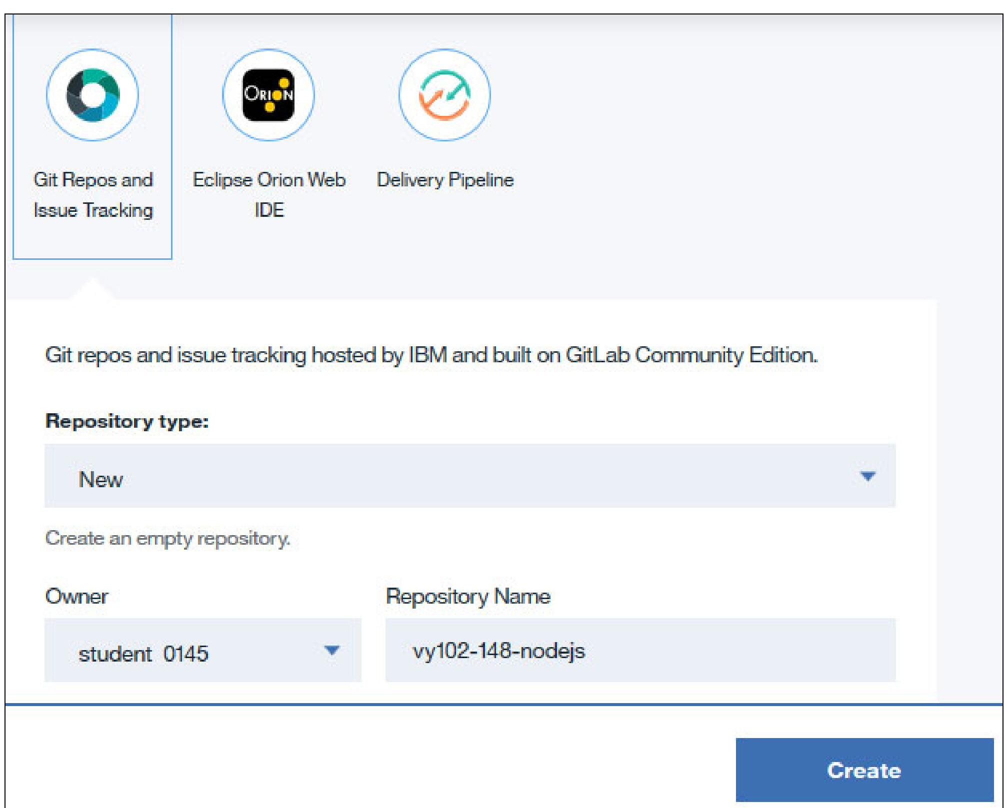


Figure 2-11 Three primary icons: Git Repos and Issue Tracking is selected by default

2. To create the Node.js app from scratch, select **New** from the **Repository type** menu, and then click **Create**. A new page opens (Figure 2-12).

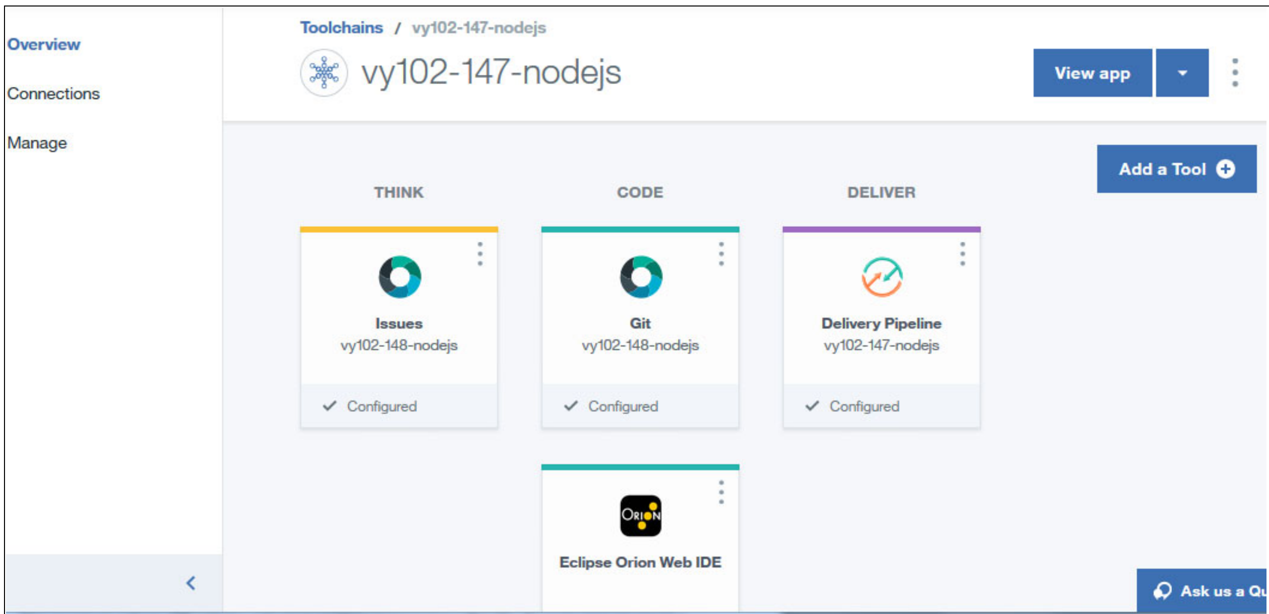


Figure 2-12 Create a toolchain page

2.3.4 Integrate the Node.js app with the Watson Language Translator service

An IBM Cloud service is a ready-for-use functionality that is hosted on IBM Cloud and can be accessed from your application over the internet. Examples of services are databases, message queues, and many others. In this exercise, you use the Watson Language Translator service.

Create a Language Translator service instance

In the following steps, you create an instance of the Watson Language Translator service that is hosted on IBM Cloud and access this service from your Node.js application:

1. Right-click **Catalog** and select **Open link in new tab** (Figure 2-13). A Catalog tab opens.

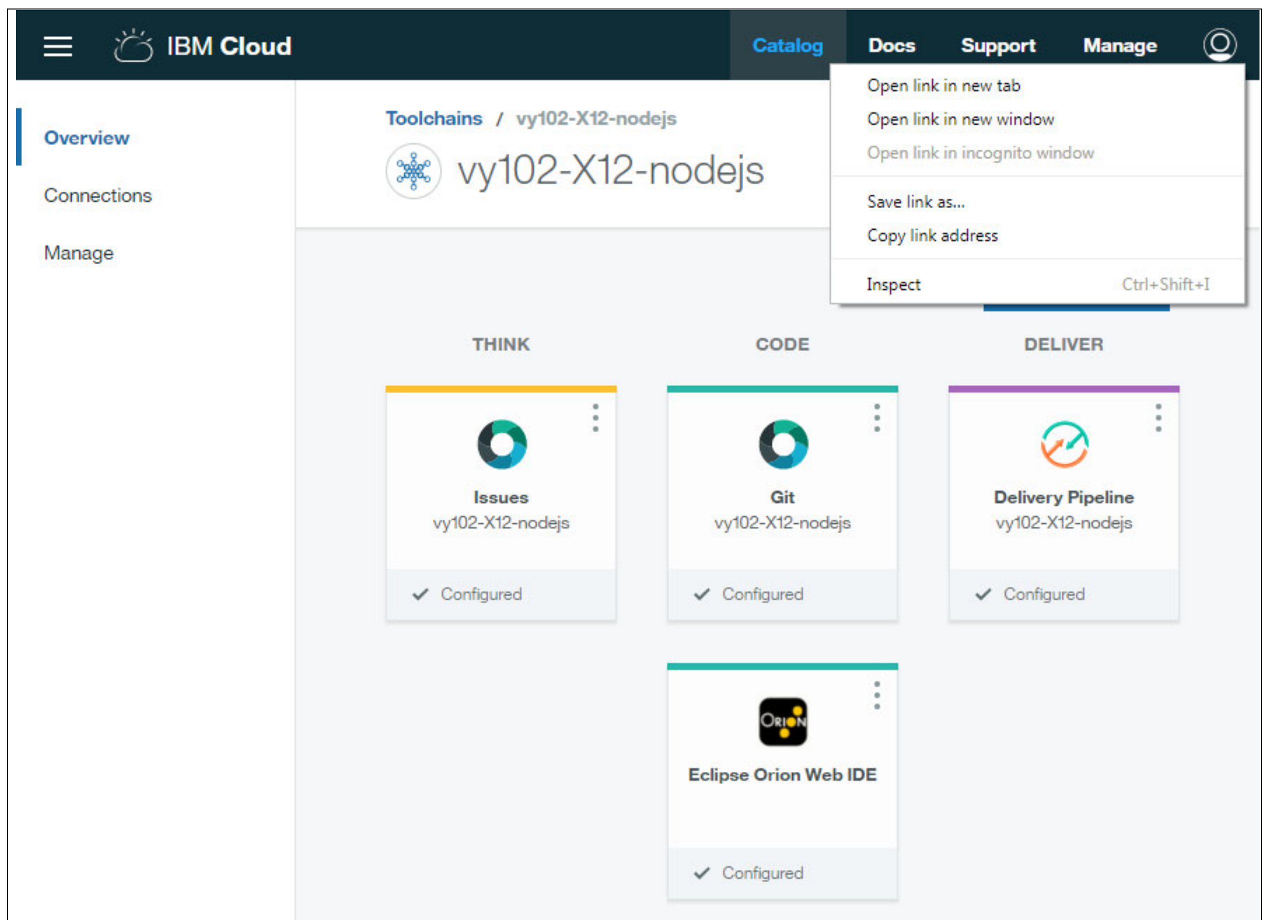


Figure 2-13 Open Catalog in new tab

2. In the filter field, search for the **Language Translator** service and click it (Figure 2-14).

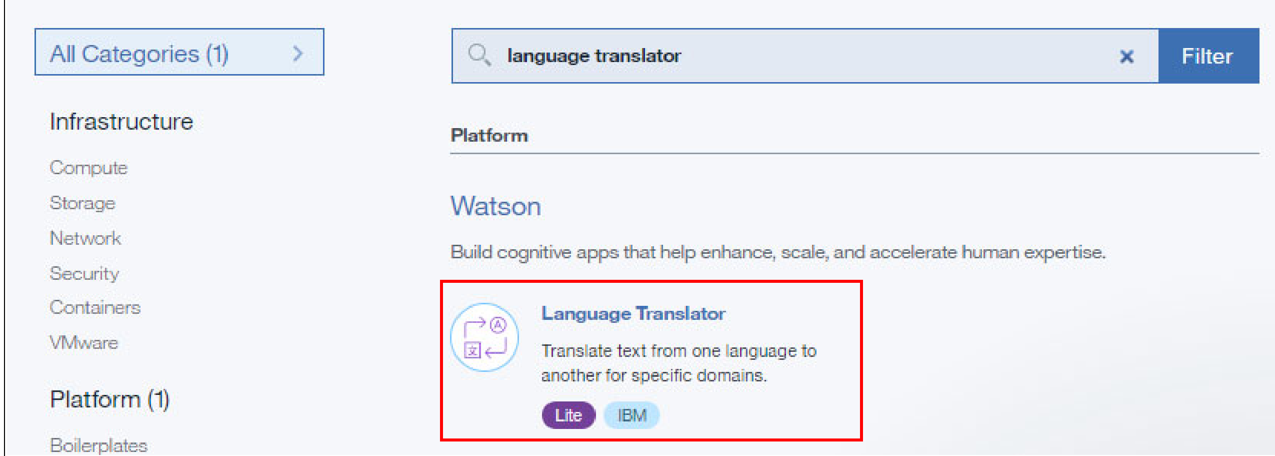


Figure 2-14 Language Translator service icon

Use this service to translate text from one language to another.

3. The Language Translator service page opens (Figure 2-15). Keep the default values and click **Create**.

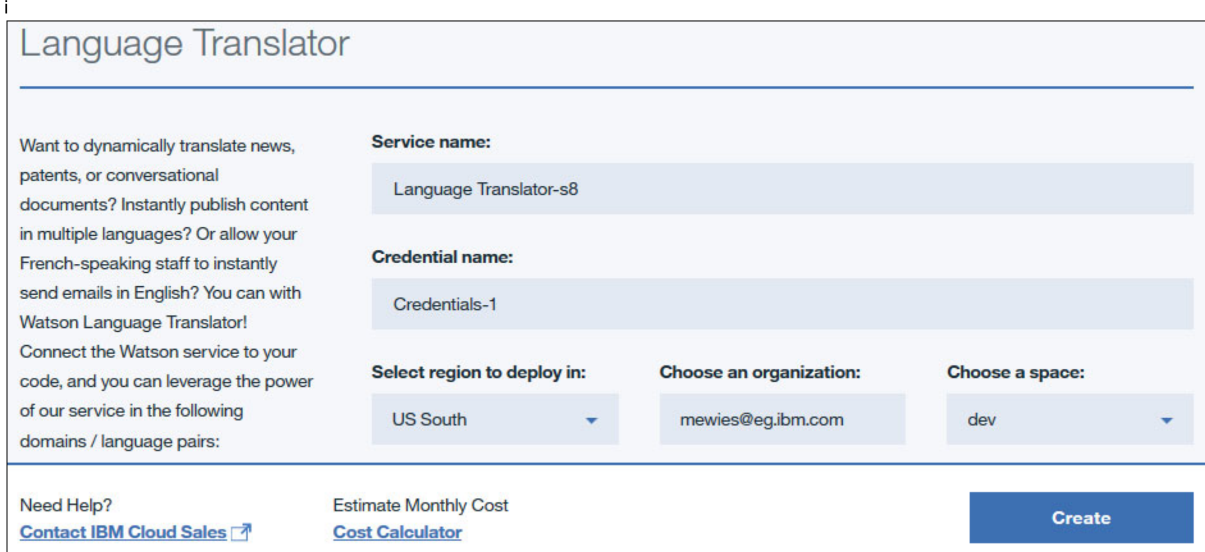


Figure 2-15 Creating the Language Translator service

Connect the Node.js app to the Language Translator service

To connect your Node.js app to the Language Translator service, complete these steps:

1. Click **Connections** on the left navigation bar, then click **Create Connection**.
2. From the list, select your Node.js application that you created in 2.3.2, "Create the Node.js application on IBM Cloud" on page 31, and click **Connect**.
3. A message prompts you to restage the application so that it can use the Language Translator service (Figure 2-16). Click **Restage**.

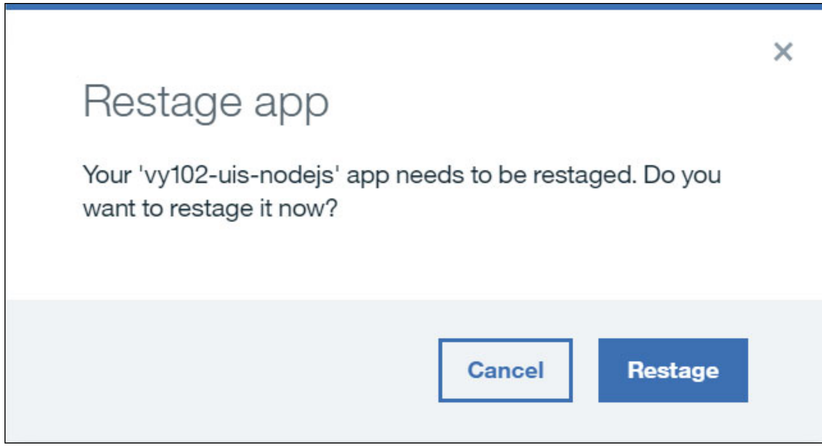


Figure 2-16 Restage after connecting to the Language Translator service

Obtain the Language Translator service credentials

The service credentials are required to access the service. To create and obtain the credentials for your Language Translator service instance, complete these steps:

1. In the Language Translator Service Details page, click **Service credentials** on the left navigation bar, then click **New Credential** (Figure 2-17).

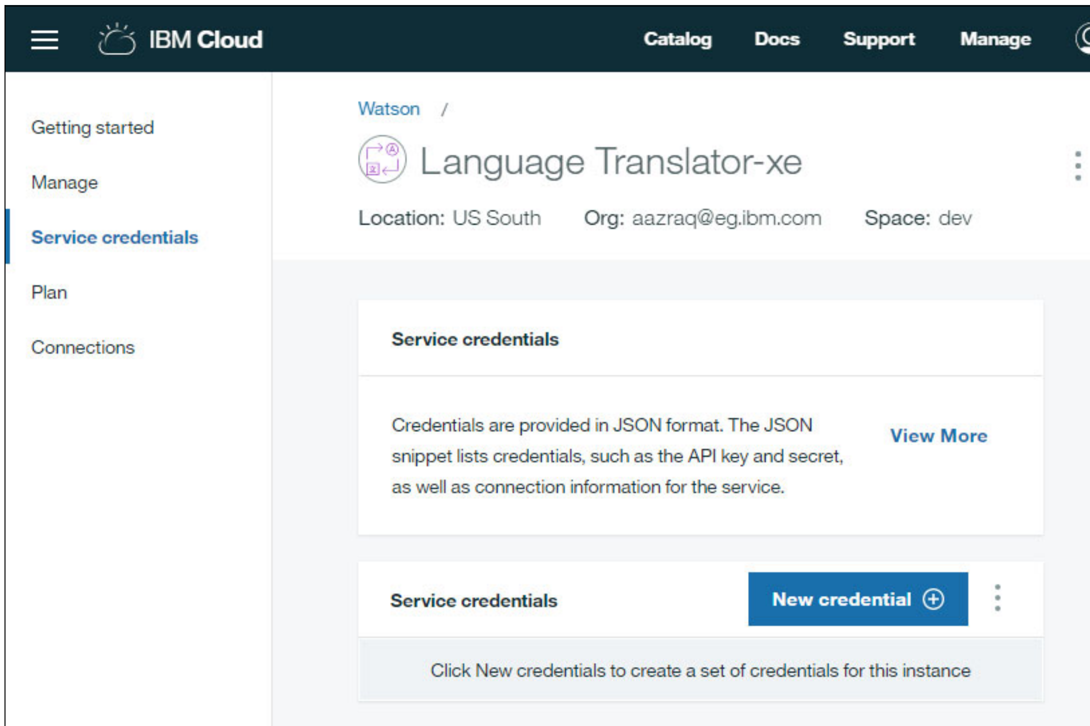


Figure 2-17 Service Credentials

2. The Add new credential window opens. Click **Add**.

3. Click **View credentials** to display the credentials. Figure 2-18 shows a JSON object that contains the service credentials. Click the **Copy** icon to copy the service credentials and paste them in a text editor. These credentials are needed to access the Language Translator service instance.

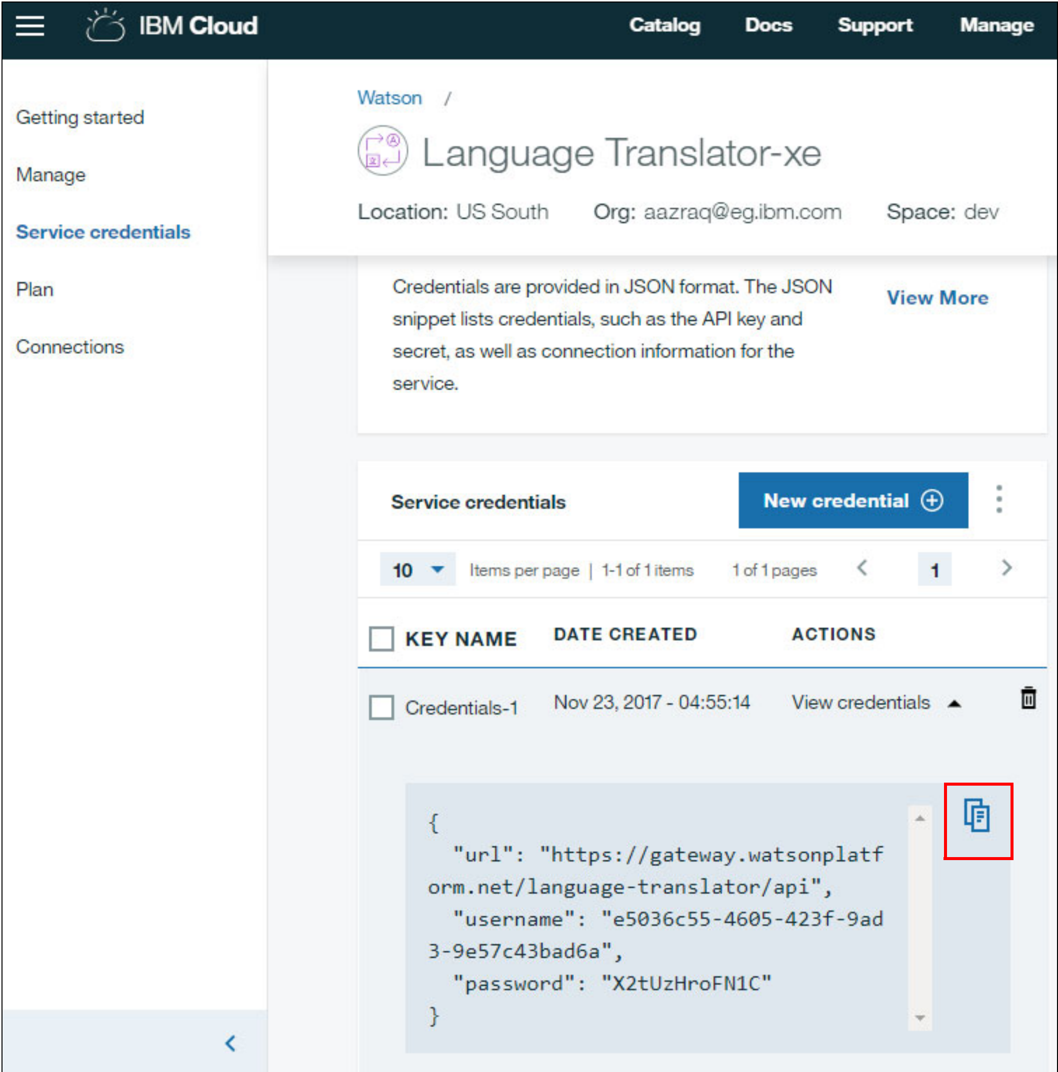


Figure 2-18 Language Translator service credentials

4. Close the Service Details browser tab.

Understand the Language Translator REST API

The Language Translator service provides REST APIs, which can be called by the applications linked to it. The following steps walk you through a demo showing you how the Language Translator REST API works:

1. Open a new web browser tab. Use the following web address to open the Language Translator for IBM Watson Developer Cloud API:

   https://language-translator-demo.ng.bluemix.net/

2. Scroll down to the Translate Text section. Select **English** for the input language, and **Spanish** for the Output language. Then, write `Hello` in the **Text input** field and press **Enter**. The **Text output** field shows the translation as `Hola` as shown in Figure 2-19.

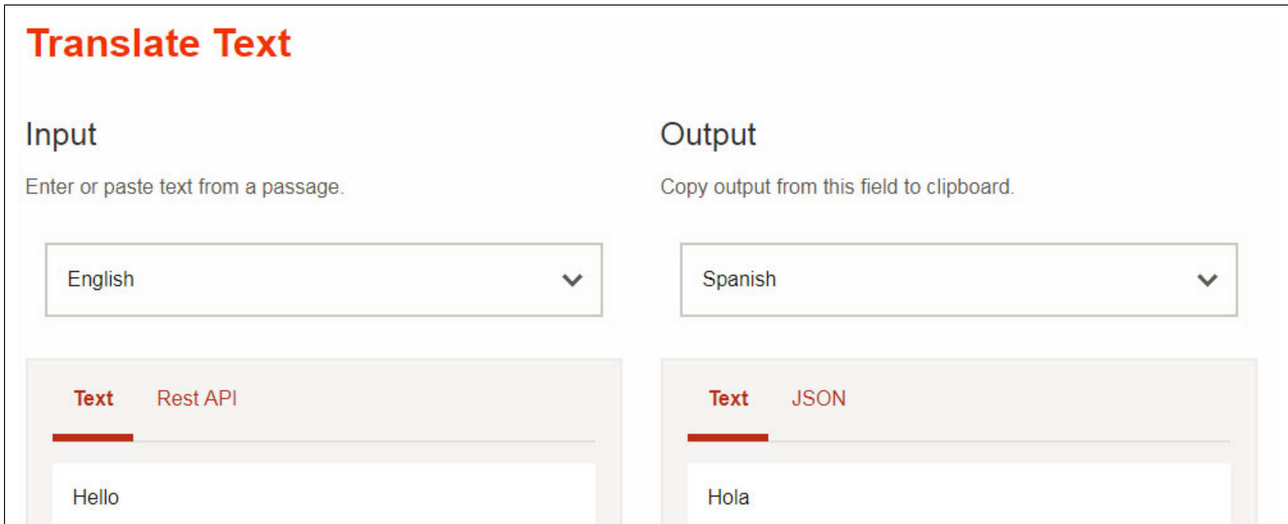


Figure 2-19 Language Translator For IBM Cloud API

3. In the Output section, click **JSON**. The JSON object retrieved from the API call is shown (Example 2-2).

Example 2-2 JSON object as response

```
{
{
  "translations": [
    {
      "translation": "Hola"
    }
  ],
  "word_count": 1,
  "character_count": 5
}
```

> **Note:** The `translations.translation` field contains the translated text result. You will learn how to read the content of this field in a Node.js app in the next section, 2.3.5, "Access the Language Translator service from the Node.js app".

4. Close the opened tab of the web browser.

2.3.5 Access the Language Translator service from the Node.js app

This section describes the steps to access the Language Translator service from a Node.js application.

1. On the View toolchain page, click the **Eclipse Orion Web IDE** icon.

   The browser shows the generated Node.js project in the Eclipse Orion Web IDE (Figure 2-20).

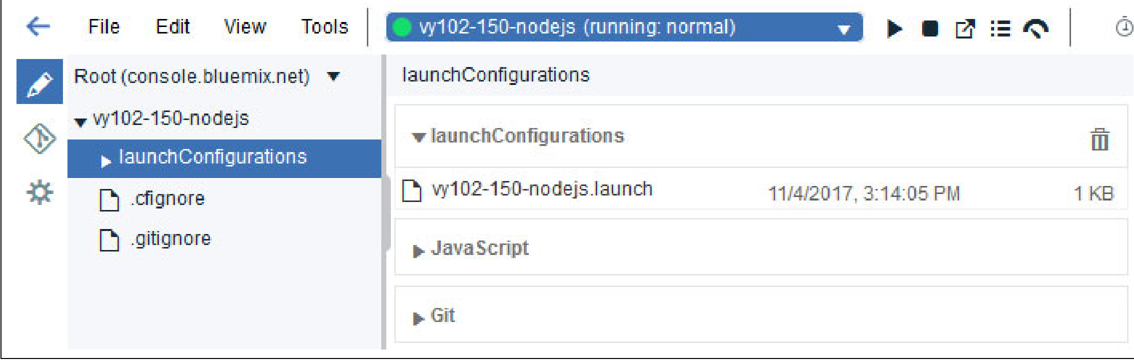


Figure 2-20 Eclipse Orion Web IDE

2. Right-click the root of the project (named `vy102-XXX-nodejs`), and select **New** → **File**. A text field is displayed. Type `manifest.yml` and then press **Enter**. The `manifest.yml` file is now created.
3. Add the code snippet from Example 2-3 to the `manifest.yml` file. Replace the `XXX` with your unique key (three random characters).

Example 2-3 Code snippet: manifest.yml file

```
applications:
- path: .
  memory: 256M
  instances: 1
  domain: mybluemix.net
  name: vy102-XXX-nodejs
  host: vy102-XXX-nodejs
  disk_quota: 1024M
```

4. If required, change the domain on based on your IBM Cloud region as listed in Table 2-1.

Table 2-1 IBM Cloud regions and domains

Region	Domain
US South	mybluemix.net
United Kingdom	eu-gb.mybluemix.net
Sydney	syd.mybluemix.net
Germany	eu-de.mybluemix.net

5. Create a file and name it `package.json`. Insert the code snippet from Example 2-4 into the `package.json` file.

   Notice the `dependencies` section, which includes the ready made packages to be used by the application. The package used in this exercise is `watson-developer-cloud`, which is the Node library used to access the Watson Developer Cloud services. You will use it in the exercise to access the Language Translator service.

   Example 2-4 Code snippet: package.json file

```
{
    "name": "NodejsStarterApp",
    "version": "0.0.1",
    "description": "App for understanding Callback",
    "scripts": {
        "start": "node app.js"
    },
    "dependencies": {
        "watson-developer-cloud": "^1.0.0"
    }
}
```

6. Create a file and name it `app.js`. In the `app.js` file, insert the code from Example 2-5.

   Example 2-5 Code snippet: Create http instance

```
const http = require('http');
```

   The `http` module is used for HTTP functions.

7. Next, insert the code shown in Example 2-6.

   Example 2-6 Code snippet: Create watson-developer-cloud instance

```
const watson = require('watson-developer-cloud');
```

   This line imports the `watson-developer-cloud` package. This package allows access to all of the Watson APIs, one of which is the Language Translator service that you use in this exercise.

8. Next, insert the code snippet from Example 2-7.

   Example 2-7 Code snippet: Create server to receive http request

```
var portNumber = process.env.VCAP_APP_PORT || 8080;
const server = http.createServer(handleRequests);
server.listen(portNumber, function() {
    console.log('Server is up!');
});
```

   This code creates a server to receive the `http` requests from the user. The function `handleRequests` is called whenever a request is received.

9. The next step is to implement the `handleRequests` function. Insert the code snippet from Example 2-8 at the end of the app.js file.

   Example 2-8 Code snippet: handleRequests function

```
function handleRequests(userRequest, userResponse) {

}
```

The `handleRequests` function has two parameters:

– The request (named `userRequest` in this example).
– The response (named `userResponse` in this example).

Figure 2-21 shows the `app.js` file up to this point in the exercise.

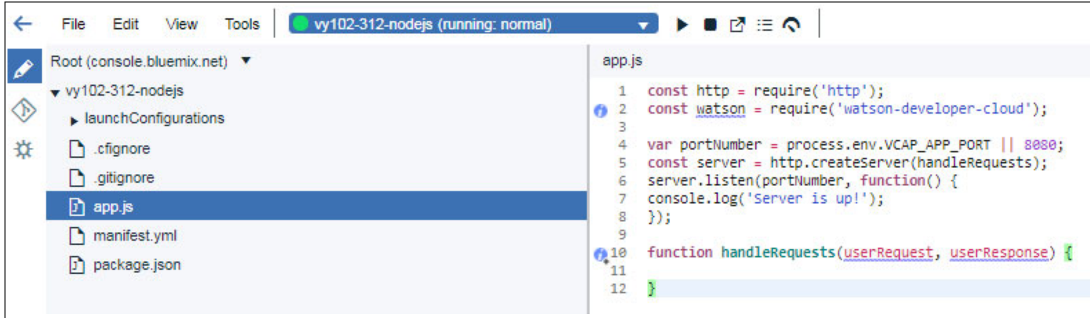


Figure 2-21 Watson developer cloud library, server, and handleRequests function

In the `handleRequests` function, add the following line for setting the header data:

```
userResponse.writeHead(200, {'Content-Type': 'text/plain'});
```

10. Below the `userResponse.writeHead` line, enter the code snippet from Example 2-9.

Example 2-9 Code snippet: Identify translation request message

```
var helloText = 'Hello';
var fromLanguage = 'en';
var toLanguage = 'es';
```

These variables contain the data to be sent in the request to the Language Translator service:

– `helloText` stores the text that will be translated.
– `fromLanguage` stores the source language for the translation.
– `toLanguage` stores the target language for the translation.

11. After the variables declaration in the function handleRequests, enter the code snippet from Example 2-10.

Example 2-10 Code snippet: Language Translator object

```
var language_translator = watson.language_translator({
   version: 'v2'
});
```

The `watson` module includes the `language_translator` API. This function expects a JSON object containing details about the connection to the Language Translator service such as the URL, username, and password.

However, because the Language Translator service is bound to the Node.js app through IBM Cloud, there is no need to specify these details as they are stored in VCAP_SERVICES. Only the version of the Language Translator service is specified here.

12. Enter the code snippet from Example 2-11.

Example 2-11 Code snippet: language_translator.translate

```
language_translator.translate({
   text: helloText,
   source: fromLanguage,
   target: toLanguage
}, callback);
```

The `language_translator.translate` function is called. The `translate` function expects two arguments:

– A JSON object. This JSON object contains the text, source, and target fields. As shown Example 2-11, the variables `helloText`, `fromLanguage`, and `toLanguage` are used for setting the values.

– A callback function. This function will be run *after* the Language Translator service finishes processing the request and returns the response to the Node.js app. While the Language Translator service is processing the request, the Node.js app registers the callback function so it can be run after the response is returned.

13. *Before* the line `language_translator.translate`, enter the code snippet in Example 2-12.

Example 2-12 Code snippet: language_translator callback function

```
var callback = function(err, data) {
   if (err) {
      console.log(err);
      userResponse.end('Error: ' + err);
   } else {
      console.log(data);
      userResponse.end('Translation of ' + helloText + " is " +
data.translations[0].translation);
   }

};
```

The callback function has two arguments:

– The `err` object, which is used if an error occurs in the function
– The `data` object, which contains the data returned by the Language Translator service.

The translation data can be sent in the `userResponse` object, which is the response object that contains the text to be returned to the web browser.

The `handleRequests` function should now look as shown in Example 2-13.

Example 2-13 Code snippet: handleRequests function complete

```
const http = require('http');
const watson = require('watson-developer-cloud');


var portNumber = process.env.VCAP_APP_PORT || 8080;
const server = http.createServer(handleRequests);
server.listen(portNumber, function() {
   console.log('Server is up!');
});


function handleRequests(userRequest, userResponse) {

   userResponse.writeHead(200, {
      'Content-Type': 'text/plain'
   });

   var helloText = 'Hello';
   var fromLanguage = 'en';
   var toLanguage = 'es';
   var language_translator = watson.language_translator({
      version: 'v2'
   });

   var callback = function(err, data) {
      if (err) {
         console.log(err);
         userResponse.end('Error: ' + err);
      } else {
         console.log(data);
         userResponse.end('Translation of ' + helloText + " is " +
data.translations[0].translation);
      }

   };

   language_translator.translate({
      text: helloText,
      source: fromLanguage,
      target: toLanguage
   }, callback);

}
```

14. If you see a warning message `Parameter 'userRequest' is never used` for the line function **`handleRequests(userRequest, userResponse)`**, click **Disable no-unused-params**.

15. Next, click **Create new launch configuration** and the “**+**” sign in the drop-down list. If you do not have the **Create new launch configuration** option, skip this step.

    In the Edit Launch Configuration window, ensure that the **Organization** is set to your email address, and **Space** is set to dev, and click **Save**.

16. Click the play icon (**Deploy the App from the Workspace**) to deploy the app.
17. After the deployment is complete, click the **Open the deployed app** icon.

    You see the output in your browser (Figure 2-22). The output shows the translation of the *Hello* text from English to Spanish.

```
Translation of Hello is Hola
```

Figure 2-22 Language Translator output

2.3.6 Access the Language Translator service through a Node.js module

In 1.3.6, "Add a module to the Node.js application" on page 21, you learned how to create a Node.js module. In this section, you will learn how to return a callback function in a module.

The following steps create a Node.js module, called `translator`, that will contain the logic for accessing the Language Translator service:

1. In the root path, create a new folder and name it `translator`.
2. In the `translator` folder, create a file named `package.json`.
3. Enter the code snippet in Example 2-14 in the `package.json` file. The `main` field contains the path of the JS file that has the Language Translator service code.

Example 2-14 Code snippet: package.json for Language Translator

```
{
    "name": "translator",
    "main": "./lib/translator",
    "dependencies": {
        "watson-developer-cloud": "^1.0.0"
    }
}
```

The `package.json` file in Example 2-14 represents the `translator` module.

4. In the `translator` folder, create a new folder named `lib`.

5. In the `lib` folder, create the `translator.js` file.

   The `translator` folder (and `translator.js` file) now look like Figure 2-23.

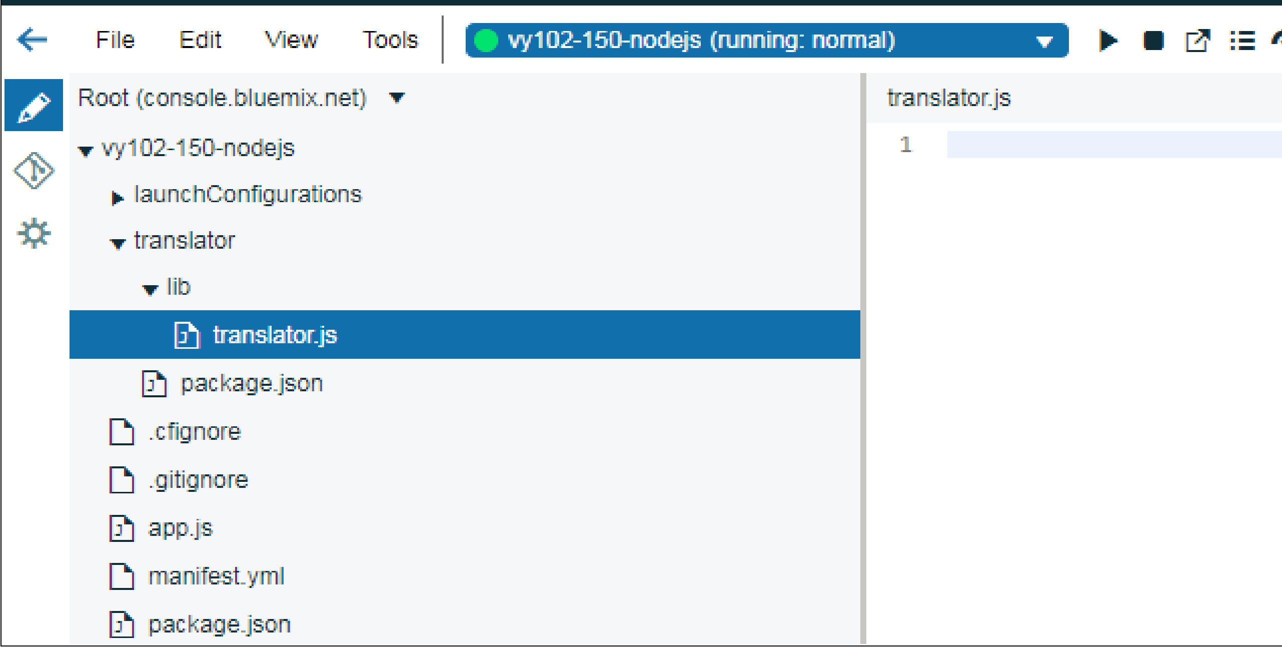


Figure 2-23 The translator.js file

   Leave the translator.js file empty for now.

6. Open the `app.js` file and remove all code in this file.
7. Add the code in Example 2-15 to the `app.js` file.

   Example 2-15 Code snippet: Importing http and translator

```
const http = require('http');
const translatorModule = require('./translator');
```

   The first line is for importing the `http` module. The second line is for importing the `translator` module that you created in the previous steps. You now import the `translator` module into the `app.js` file. You implement the `translator` module later.

8. Add the code snippet from Example 2-16 to the `app.js` file.

   Example 2-16 Code snippet: Declare text, from and to fields

```
var helloText = 'Hello';
var fromLanguage = 'en';
var toLanguage = 'es';
```

   These lines declare the variables for the input fields that you use when accessing the Language Translator service.

9. Below the `var toLanguage` line, add the code snippet from Example 2-17.

Example 2-17 Code snippet: Create server and listen on a port

```
var portNumber = process.env.VCAP_APP_PORT || 8080;
const server = http.createServer(handleRequests);
server.listen(portNumber, function() {
   console.log('Server is up!');
});
```

These lines create the server and listen on a certain port.

`createServer` receives the `handleRequests` callback function. You implement this function in the next step.

10. Add the code snippet from Example 2-18 for the `handleRequests` callback function.

Example 2-18 Code snippet: handleRequests

```
function handleRequests(userRequest, userResponse) {
   userResponse.writeHead(200, {
      'Content-Type': 'text/plain'
   });
}
```

The body has code that sets the header content.

11. Inside the `handleRequests` callback function, after the **`userResponse.writeHead(200, { 'Content-Type': 'text/plain' });`** function call, enter the code snippet from Example 2-19.

Example 2-19 Code snippet: getTranslation

```
translatorModule.getTranslation(helloText, fromLanguage, toLanguage, callback);
```

The code line in Example 2-19 calls the `getTranslation` function that should be defined in the Language Translator module.

This function has following parameters:

- `helloText`
- `fromLanguage`
- `toLanguage`
- `callback` function, which will be called after the `getTranslation` function finishes its execution

12. *Before* the `translatorModule.getTranslation()` line you just added, insert the code snippet from Example 2-20 for the callback function that is mentioned in the previous step.

Example 2-20 Code snippet: The callback function with translatorOutput

```
   var callback = function(error, translatorOutput) {
      if (error) {
         userResponse.end(error);
      } else {
         userResponse.end('Translation of ' + helloText + " is " +
translatorOutput);
      }

   };
```

The `callback` function is expected to have an `error` parameter that holds a value in case of errors. The other parameter is for the `translatorOutput`. If the `error` object is not null, the HTTP response is sent back to the user with this error. If no error occurs, the message containing the `translatorOutput` is sent in the HTTP response back to the user.

The `app.js` file now looks like the file in Figure 2-24.

```
app.js
1  const http = require('http');
2  const translatorModule = require('./translator');
3
4  var helloText = 'Hello';
5  var fromLanguage = 'en';
6  var toLanguage = 'es';
7
8  var portNumber = process.env.VCAP_APP_PORT || 8080;
9  const server = http.createServer(handleRequests);
10 server.listen(portNumber, function() {
11     console.log('Server is up!');
12 });
13
14 function handleRequests(userRequest, userResponse) {
15     userResponse.writeHead(200, {
16         'Content-Type': 'text/plain'
17     });
18
19     var callback = function(error, translatorOutput) {
20         if (error) {
21             userResponse.end(error);
22         } else {
23             userResponse.end('Translation of ' + helloText + " is " + translatorOutput);
24         }
25
26     };
27
28     translatorModule.getTranslation(helloText, fromLanguage, toLanguage, callback);
29 }
```

Figure 2-24 The app.js file using the translator module

Save the file by clicking **File** → **Save**.

13. Open the `translator.js` file. Add the code snippet from Example 2-21 to import the `watson-developer-cloud` module.

Example 2-21 Code snippet: Importing the watson-developer-cloud API

```
var watson = require('watson-developer-cloud');
```

14. Next, define and export the `getTranslation` data, which is used in the `app.js` file. The initial code looks like the code snippet shown in Example 2-22.

Example 2-22 Code snippet: Exporting the getTranslation function

```
exports.getTranslation = function getTranslation(helloText, fromLanguage, toLanguage, callback) {

};
```

This function receives the fields `helloText`, `fromLanguage`, `toLanguage`, and the `callback` function that is to be run when the `getTranslation` function finishes execution.

15. Inside the `getTranslation` function, add the code snippet from Example 2-23. This code is for accessing the Language Translator API.

Example 2-23 Code snippet: Access Language Translator API

```
  var language_translator = watson.language_translator({
    version: 'v2'
  });
```

Below the code in Example 2-23, add the code snippet from Example 2-24 for the actual call to the Language Translator service.

Example 2-24 Code snippet: language_translator translate function

```
language_translator.translate({
    text: helloText,
    source: fromLanguage,
    target: toLanguage
  }, translatorCallback);
```

16. *Before* the `language_translator.translate` line, add the code snippet from Example 2-25. This code defines the `translatorCallback` callback function.

Example 2-25 Code snippet: translatorCallback function

```
var translatorCallback = function(err, data) {

  };
```

The `translatorCallback` callback function is called after the response from the `language_translator.translate` function is returned.

The `translatorCallback function` contains two parameters:

– `err`: This parameter has a value if there is an error returned when the `language_translator.translate` function is run.

– `data`: This parameter has the returned response data from running the `language_translator.translate` function.

17.Inside the `translatorCallback` function, add the code snippet from Example 2-26.

Example 2-26 Code snippet: translatorCallback callback function details

```
if (err) {
          console.log(err);
          callback(err, null);
       } else {
          console.log(data);
          callback(null, data.translations[0].translation);
       }
```

The code first checks if an error occurred. In case of error, run the `callback` function, with the first parameter set to the `err` value and the second parameter, the data object, to null.

If `err` is null, then no errors occurred. In this case, the `data` object has the response data from calling the Language Translator service.

Now, the `callback` function can be run by setting the first parameter as null (because no errors occurred) and the second parameter as `data.translations[0].translation` to read the returned translation from the `data` JSON object. The JSON object structure is shown in Example 2-2 on page 40.

Figure 2-25 shows the `translator.js` file at this point.

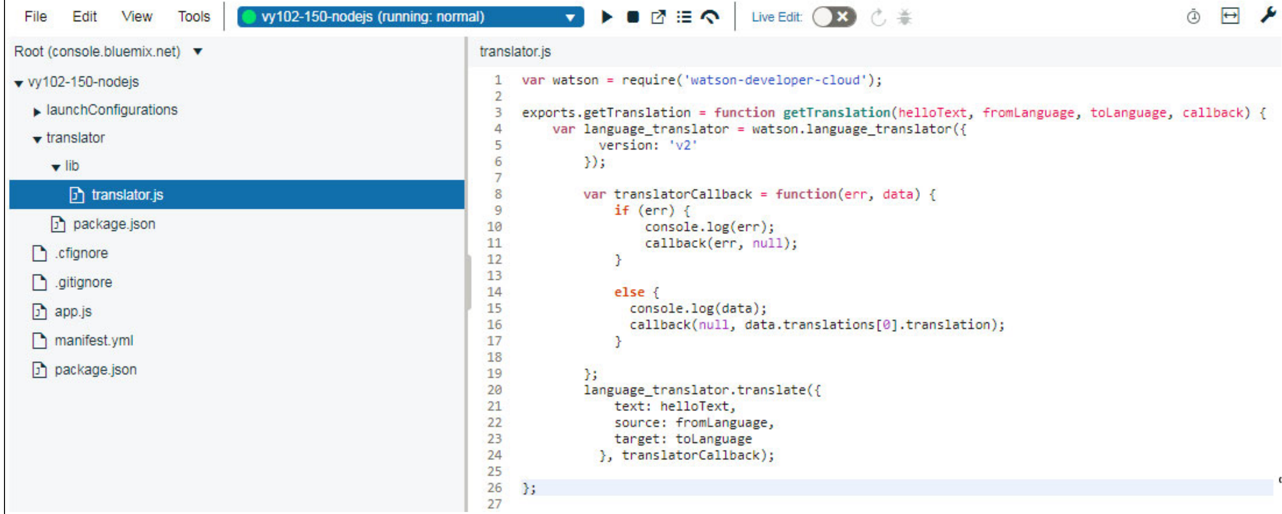


Figure 2-25 Translator module code

18.Click the play icon (**Deploy the App from the Workspace**) to deploy the app. Confirm if you are prompted to restart the app. After the deployment is complete, click the **Open the Deployed App** icon.

The window showing the translation is displayed (Figure 2-26).

```
Translation of Hello is Hola
```

Figure 2-26 Results of translator output

2.3.7 Stop the application

IBM Cloud Lite account provides you with 256 MB of application memory for Cloud Foundry apps and 100 Cloud Foundry Services.

To free the resources that are assigned to your application, you can either stop your application or delete it. To do so, complete these steps:

1. Stop your application by clicking the **Stop the App** icon(Figure 2-27).

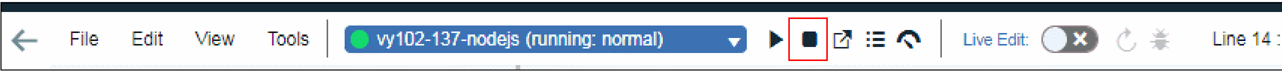


Figure 2-27 Stop the application

2. Close your web browser.

2.4 Exercise review

In this exercise you accomplished the following goals:

- Created the Language Translator service in IBM Cloud and connected it to your Node.js app.
- Used asynchronous callback functions in your Node.js app and learned how the callback function is run.
- Created a module in Node.js to call the Language Translator service and used it from other JS files.

3

Creating your first Express application

Express is a Node.js web framework. It allows rapid development of web applications. It provides an easy way to handle routing of an application by exposing REST APIs.

In this chapter, you create an application that uses the Express framework and the Watson Natural Language Understanding service to extract the author name from articles that are published on the web. You provide the web address (URL) of the article to the application, and it outputs the name of the author (or multiple names if the article has multiple authors).

This chapter contains the following topics:

- Getting started
- Architecture
- Step-by-step implementation
- Exercise review

© Copyright IBM Corp. 2017. All rights reserved.

3.1 Getting started

To start, read through the objectives, prerequisites, and expected results of this use case.

3.1.1 Objectives

Express is a Node.js web framework. It allows rapid development of web applications. It provides an easy way to handle routing of an application by exposing REST APIs.

By the end of this chapter, you should be able to accomplish these objectives:

- Create a Hello World Express application.
- Create a simple HTML view for your application.
- Understand Express routing.
- Use third-party modules in Node.js.
- Understand IBM Watson Natural Language Understanding service.
- Use a Git repository on IBM Cloud DevOps.
- Understand Delivery Pipeline.

3.1.2 Prerequisites

Before you start, be sure that you meet these prerequisites:

- Basic JavaScript skills
- Basic HTML skills
- An IBM Cloud account available at `https://console.bluemix.net/`.
- A workstation that has these components:
  - Internet access
  - Web browser: Google Chrome or Mozilla Firefox
  - Operating system: Linux, Mac OS, or Microsoft Windows

3.1.3 Expected results

You create an application that uses Express framework and Watson Natural Language Understanding service to extract the author name from any article that is published on the web. You provide the web address (URL) of the article, and the application outputs the name of the author (or multiple names if the article has multiple authors).

The application involves the following steps:

1. Figure 3-1 shows the starting page of the application. Enter the web address of any article, such as the following URL, and then click **Submit**. In this example, the URL of the article is as follows:

   ```
   http://www.forbes.com/sites/alexkonrad/2016/01/29/new-ibm-watson-chief-david-kenny-talks-his-plans-for-ai-as-a-service-and-the-weather-company-sale/
   ```

Figure 3-1 Watson Author Finder

2. The URL of the article is posted to the `/author` URL (Figure 3-2), then Watson Natural Language Understanding service is called to extract the author name (or names) from the article.

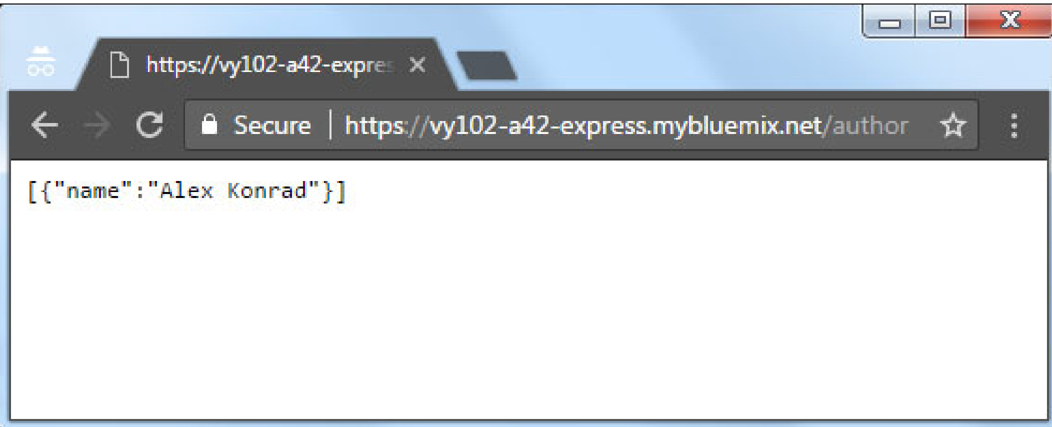


Figure 3-2 Watson Author Finder returned these results

3.2 Architecture

Figure 3-3 shows the components and runtime flow of the application.

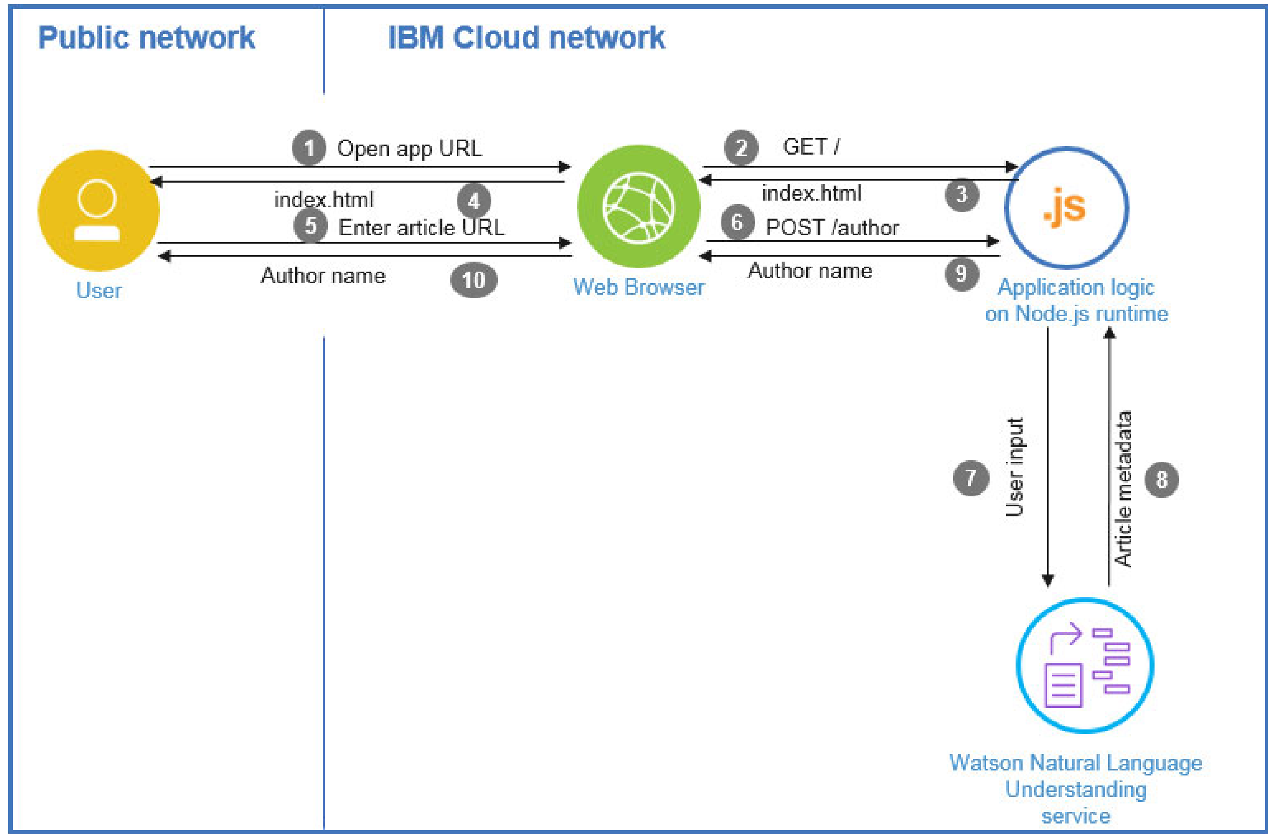


Figure 3-3 Architecture

The following steps explain the sequence of interactions between the various components in the exercise:

1. In a web browser, navigate to the application URL of this exercise:

   `http://vy102-`**`XXX`**`-express.mybluemix.net`

2. The web browser sends a GET request to the Node.js server. As mentioned in step 1, the application URL for this example is `http://vy102-XXX-express.mybluemix.net`. The path that follows this URL is a *route*, and there should be a handler for this route in the Node.js application.

   For example, if the user sends the following request, there should be a route called `'GET /sample'` in the Node.js application:

   `http://vy102-XXX-express.mybluemix.net/sample`

   This route can return a resource (HTML page, image, and so on), call a back-end service, or both.

   In step 1, the user requested the home page of this exercise's application. The browser sends a GET request to `http://vy102-XXX-express.mybluemix.net/`. That means that the `'GET /'` (root) route will be called.

3. The Express framework in Node.js returns the `index.html` file in response to the `'GET /'` route request.

4. The web browser shows the `index.html` page to the user. The `index.html` page contains a form that has one text box, and a **Submit** button. In the text box, the user can enter the URL of the article.
5. The user enters the article's URL, and then clicks **Submit**.
6. The web browser sends a POST request to the `/author` route with the article URL passed in the body.
7. Express framework in Node.js passes the article URL to Watson Natural Language Understanding service. Also, it requests that the metadata be returned.

> **Note:** The Watson Natural Language Understanding service uses natural language processing (NLP) to analyze semantic features of any text. The Natural Language Understanding service has many features, such as concepts, categories, emotion, entities, keywords, metadata, and sentiment.
>
> The feature that you use in this exercise is metadata.The feature gets document metadata, including author name, title, RSS and Atom feeds, prominent page image, and publication date.

8. The metadata of the article is returned by the Natural Language Understanding service.
9. Node.js filters the metadata to return only the author name (or names).
10. The author name (or names) of the article are returned to the user on the web browser.

3.3 Step-by-step implementation

This section provides details of how to implement the Hello World Node.js app by using the Express framework.

3.3.1 Log in to your IBM Cloud account

Log in to IBM Cloud by completing these steps:

1. Open your web browser, enter the following web address, and then press **Enter**:

   `https://bluemix.net`

2. The IBM Cloud login page opens (Figure 3-4). Click **Log in** and provide your authentication credentials.

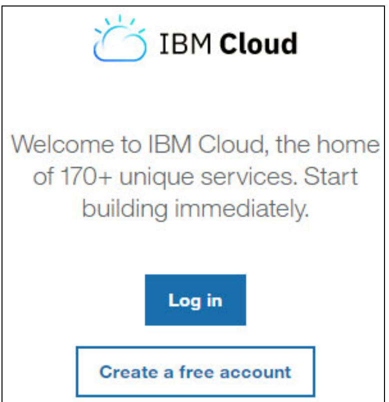


Figure 3-4 IBM Cloud login

3.3.2 Create the Node.js application on IBM Cloud

Complete these steps to create the Node.js app by using the IBM SDK for Node.js runtime on IBM Cloud and to enable continuous delivery for the application:

1. Click **Catalog** on the top bar and select **Cloud Foundry Apps** on the left pane.
2. Click **SDK for Node.js** (Figure 3-5).

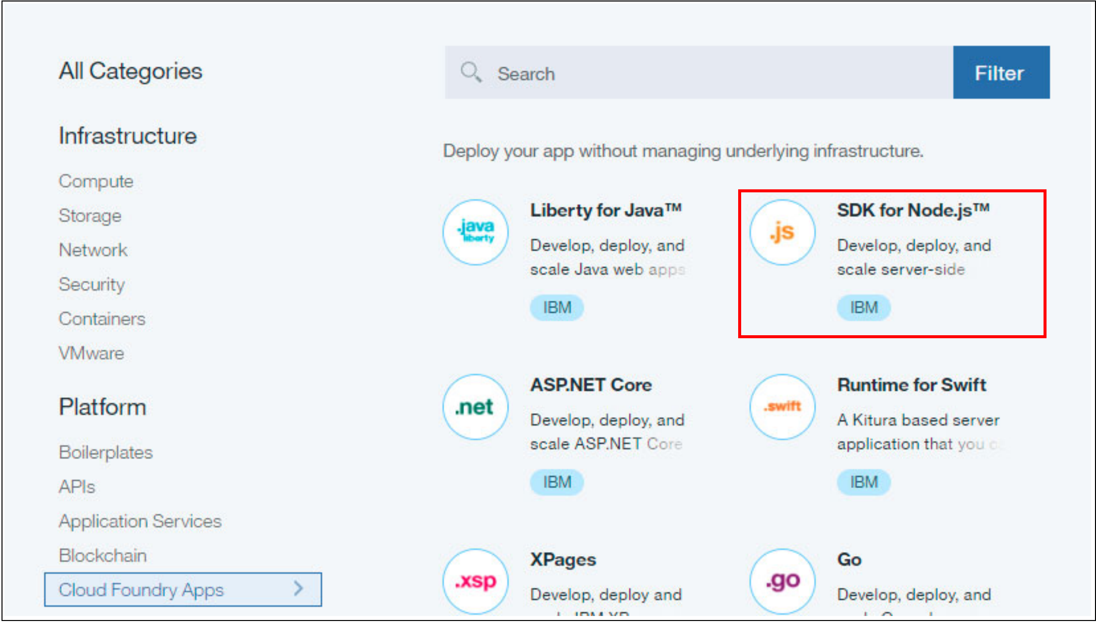


Figure 3-5 IBM Cloud Catalog: Cloud Foundry Apps

3. In the **App name** field, type `vy102-XXX-express` (Figure 3-6). Replace `XXX` with any three random characters. Click **Create**.

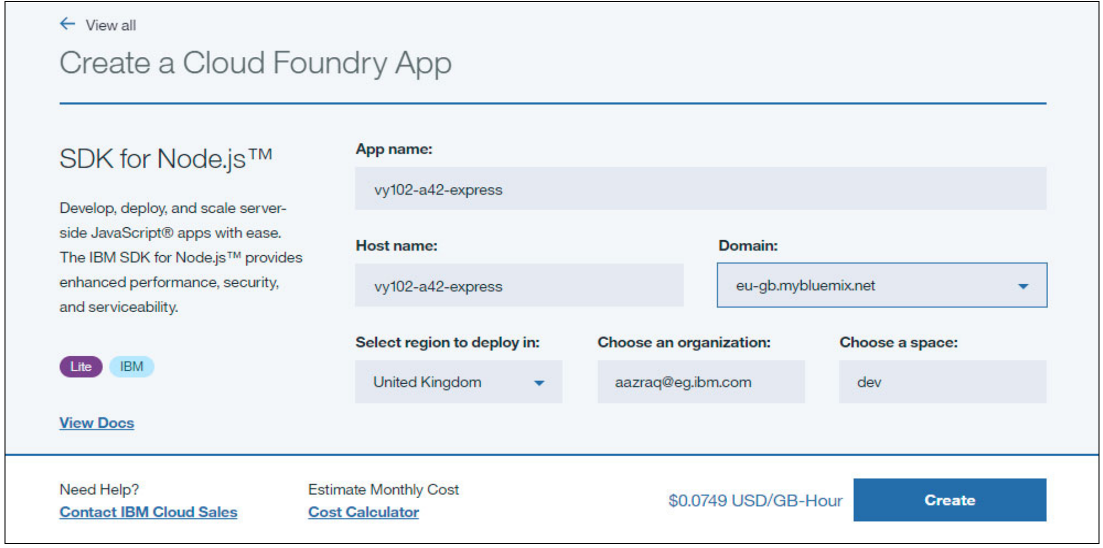


Figure 3-6 Create an SDK for Node.js application

4. In the next steps, you enable continuous delivery for this application. Application Details displays the Getting started page (Figure 3-7). Wait until the application is started and then click **Overview**.

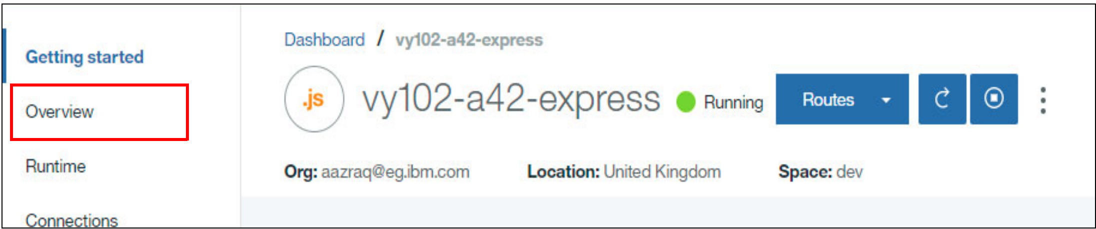


Figure 3-7 Application details: Getting started page

5. In the Continuous delivery tile of the Overview page, click **Enable** (Figure 3-8).

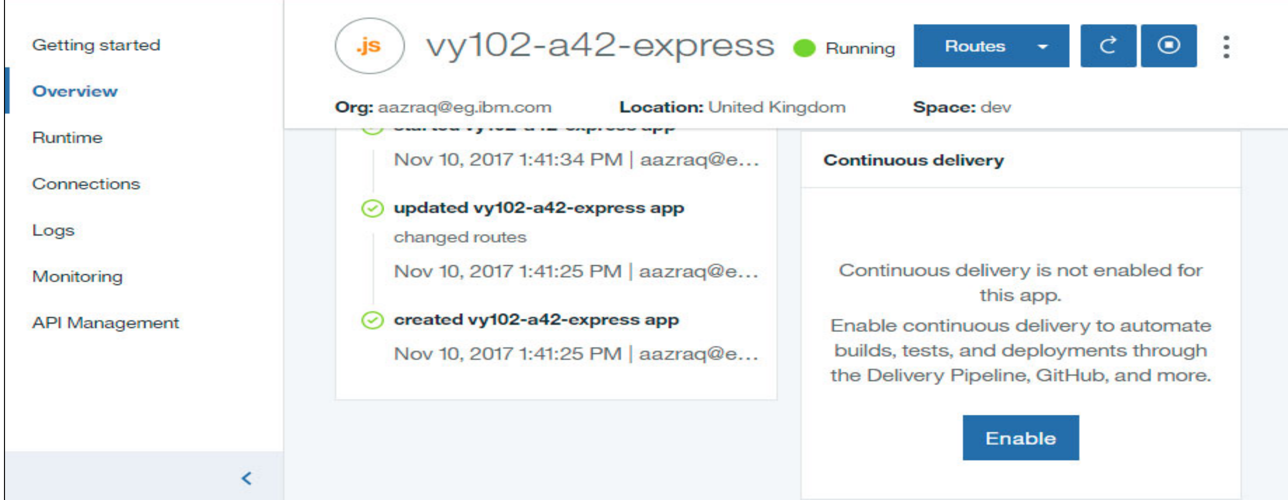


Figure 3-8 Application details: Overview page

6. A new tab opens (Figure 3-9) where you create a toolchain. In the **Repository type** field, select **New** to create an empty application, and then click **Create**.

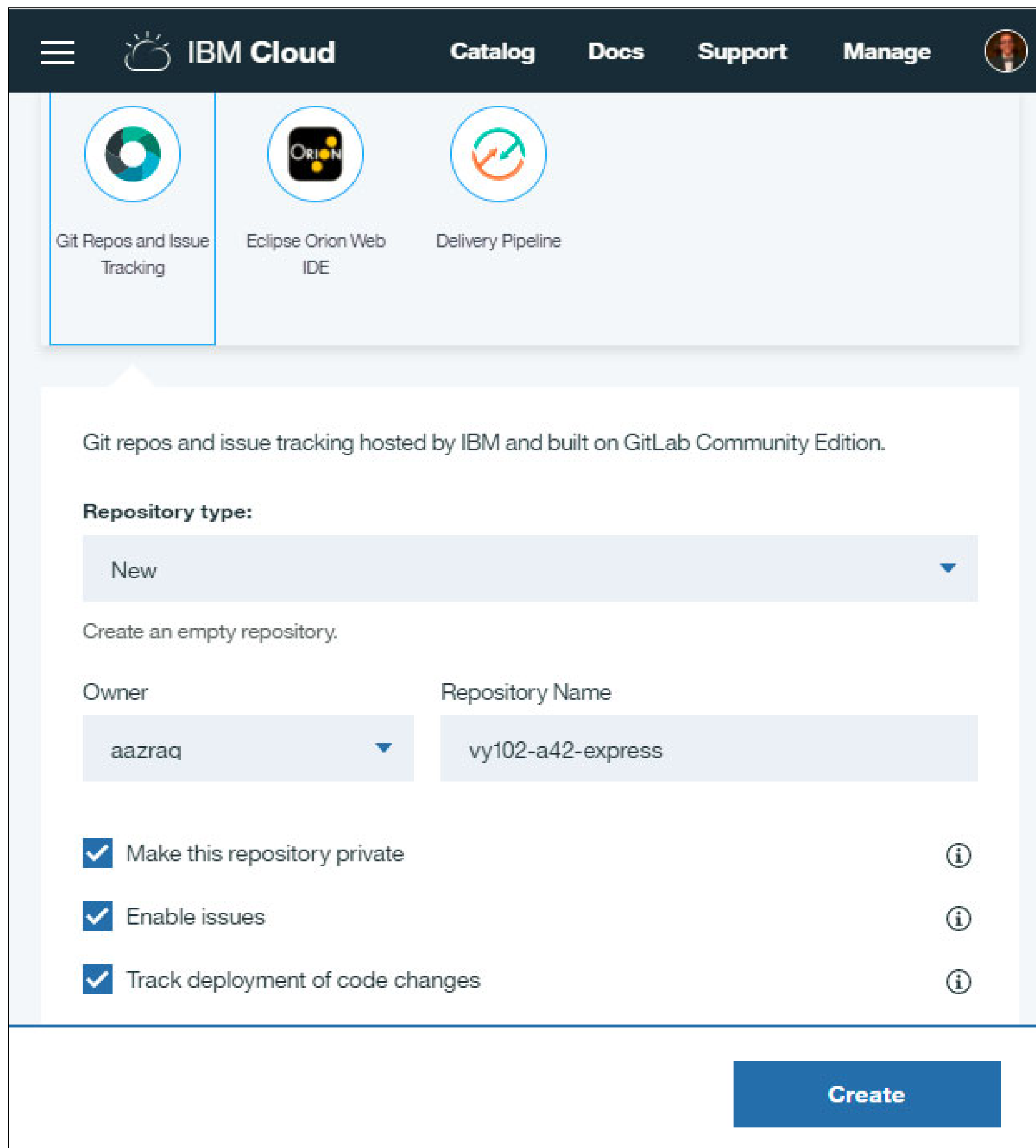


Figure 3-9 Create a toolchain

7. To start editing the code, click **Eclipse Orion Web IDE** (Figure 3-10).

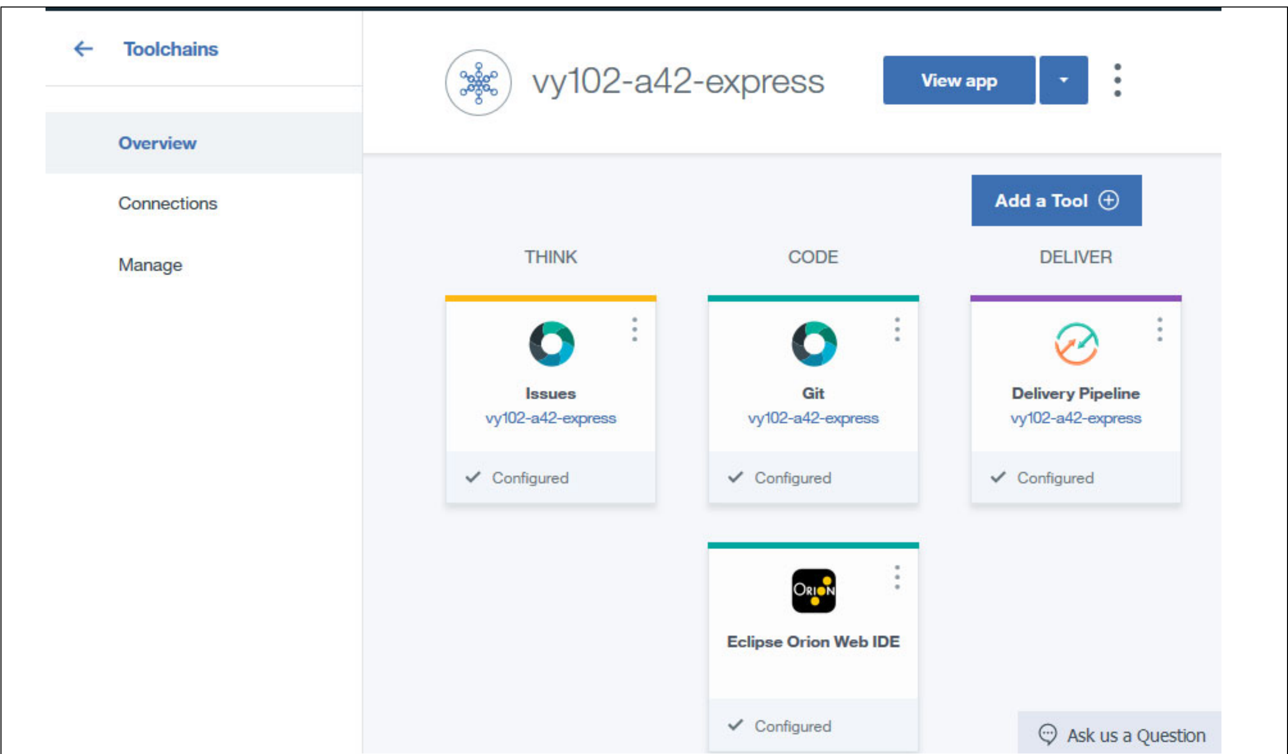


Figure 3-10 IBM Cloud DevOps toolchain

8. On the left, expand the twistie for `vy102-xxx-express` (highlighted in Figure 3-11).

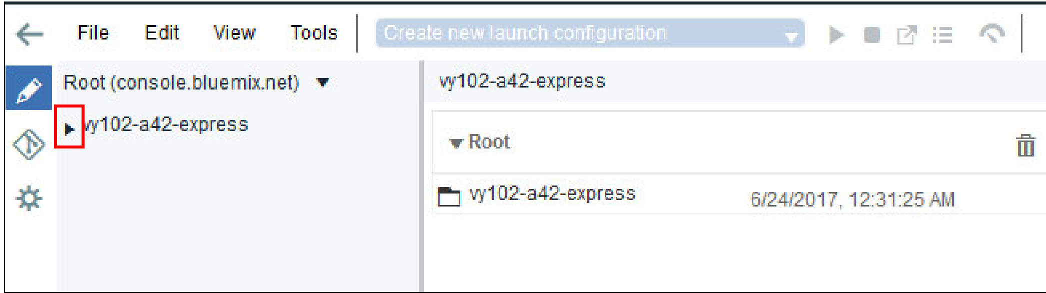


Figure 3-11 Eclipse Orion

Notice that an empty project is created (Figure 3-12).

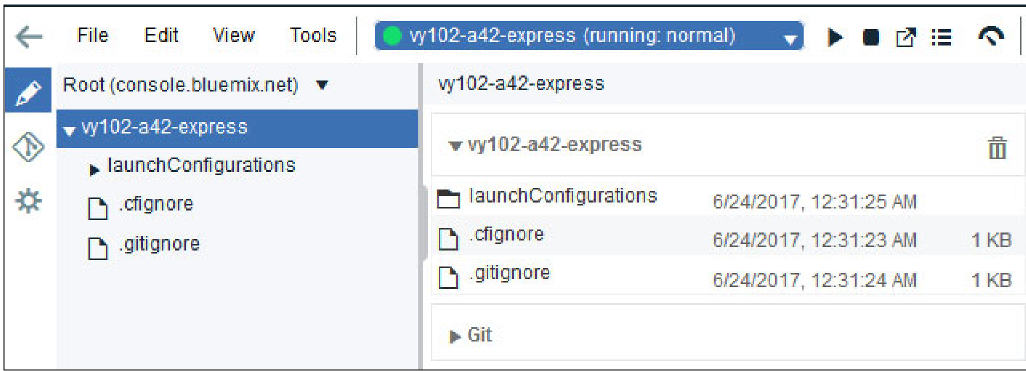


Figure 3-12 An empty project is created: vy102-xxx-express

3.3.3 Create the Hello World Express application

Express is a Node.js framework. It is used to simplify the creation of web applications on Node.js. The core component of Express is *routes*.

Route refers to the definition of application end points (Uniform Resource Identifiers (URIs)) and how they respond to client requests. Express supports the routing methods that correspond to the HTTP methods GET, POST, PUT, HEAD, DELETE, OPTIONS, and TRACE.

In the following steps, you create the Hello World Express application that returns the words "Hello Express!" in response to the GET / route request. The root (/) route is called whenever a user accesses the URL of the application. Also, you will create another route, `POST /author`, after the user sends a POST request to the `/author` resource. This route returns "Author Name" to the user.

1. Create the `package.json` file and add Express framework as one of its dependencies:
   a. Right-click `vy102-xxx-express` in the left navigation bar, and select **New** → **File**.
   b. Name the file `package.json` (Figure 3-13) and then press **Enter**.

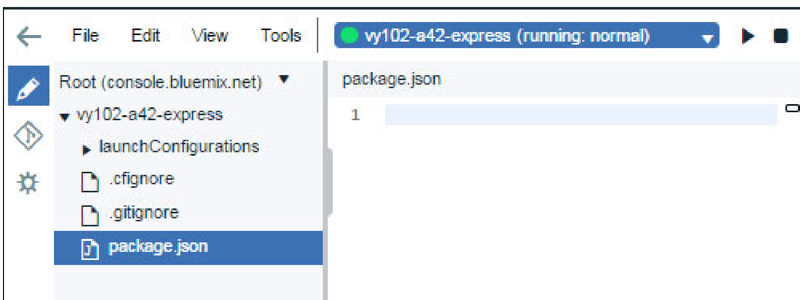


Figure 3-13 The empty package.json file

c. Copy the code snippet from Example 3-1 to the `package.json` file.

Example 3-1 Code snippet: package.json

```
{
    "name": "vy102-XXX-express",
    "version": "0.0.1",
    "description": "A sample express app",
    "dependencies": {
        "express": "4.*"
    }
}
```

d. Notice that Express version 4.x is added as a dependency. In the name, replace `XXX` with the three characters that you assigned as part of your application name (Figure 3-14).

```
package.json
1 {
2     "name": "vy102-a42-express",
3     "version": "0.0.1",
4     "description": "A sample express app",
5     "dependencies": {
6         "express": "4.*"
7     }
8 }
```

Figure 3-14 The package.json file with the Express version 4.x dependency

2. Create the `app.js` file. In `app.js`, you create an instance of Express and it will be the starting point of your application:

   a. In the navigation bar, right-click `vy102-xxx-express`, and then select **New** → **File**.

   b. Name the file `app.js` and press **Enter**. At this point, `app.js` is empty (Figure 3-15).

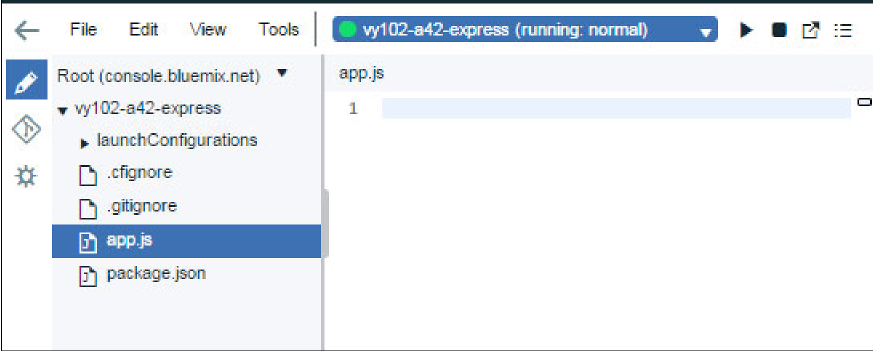


Figure 3-15 The empty app.js file

c. Copy the code snippet from Example 3-2 into the `app.js` file.

Example 3-2 Code snippet: Instantiate Express framework (GET and POST)

```
var port = process.env.VCAP_APP_PORT || 8080;

//Express Web Framework, and create a new express server
var express = require('express'),
   app = express();

// In case the caller calls GET to the root '/', return 'Hello Express!'.
app.get('/', function(req, res) {
    res.send('Hello Express!');
});

// In case the caller calls POST to /author, return 'Author name'
app.post('/author', function(req, res) {
    res.send('Author name');
});

// start server on the specified port and binding host
app.listen(port);
```

The code instantiates the Express framework, listening to the default port of IBM Cloud, and exposes two routes (Figure 3-16):

`GET /` Returns "Hello Express!" to the caller when the caller requests the root of the application.

`POST /author` When the caller issues a post request to `/author`, "Author name" is returned.

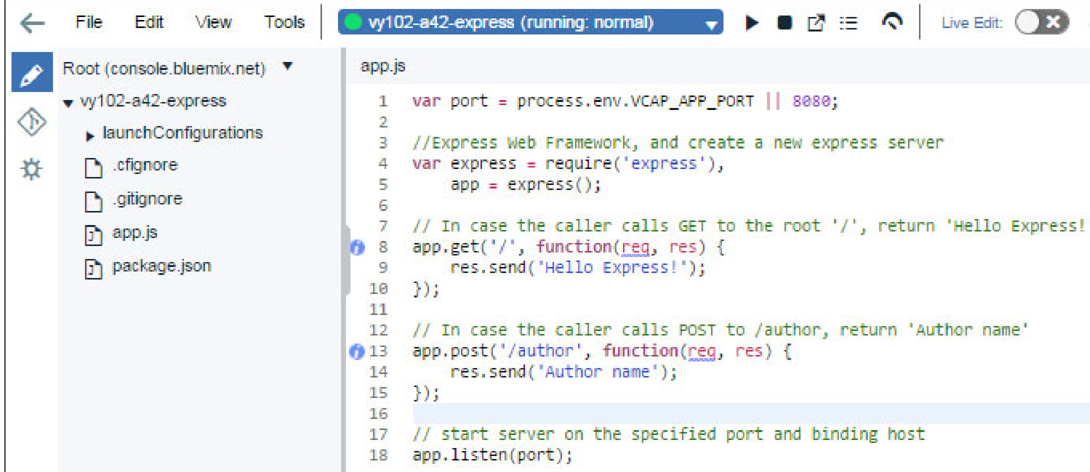


Figure 3-16 The app.js file with Express starter code

Note: You can usually press **Shift+Alt+F** to format the code.

3. Add the `manifest.yml` file with the domain and host, and configure it to run `app.js` after the Node.js server starts:

   a. Right-click `vy102-xxx-express` in the left navigation bar, and then select **New** → **File**.

   b. Name the file `manifest.yml` and press **Enter**.

   c. Copy the code snippet from Example 3-3 to the `manifest.yml` file. This code configures the domain, name, and memory of the application.

      In the `manifest.yml` file, replace `XXX` with the three characters you chose as part of your application name and host.

      Example 3-3 Code snippet: Add domain and host, and configure to run app.js

```
applications:
- path: .
  memory: 256M
  instances: 1
  domain: mybluemix.net
  name: vy102-XXX-express
  host: vy102-XXX-express

  disk_quota: 1024M
```

   d. If needed, change the domain on based on your IBM Cloud region as listed in Table 3-1.

      Table 3-1 IBM Cloud regions and domains

Region	Domain
US South	mybluemix.net
United Kingdom	eu-gb.mybluemix.net
Sydney	syd.mybluemix.net
Germany	eu-de.mybluemix.net

   e. Copy the following line to the end of the `manifest.yml` file. This line instructs IBM Cloud to start `app.js` after the application is staged.

```
command: node app
```

      Your `manifest.yml` now looks like the file in Figure 3-17.

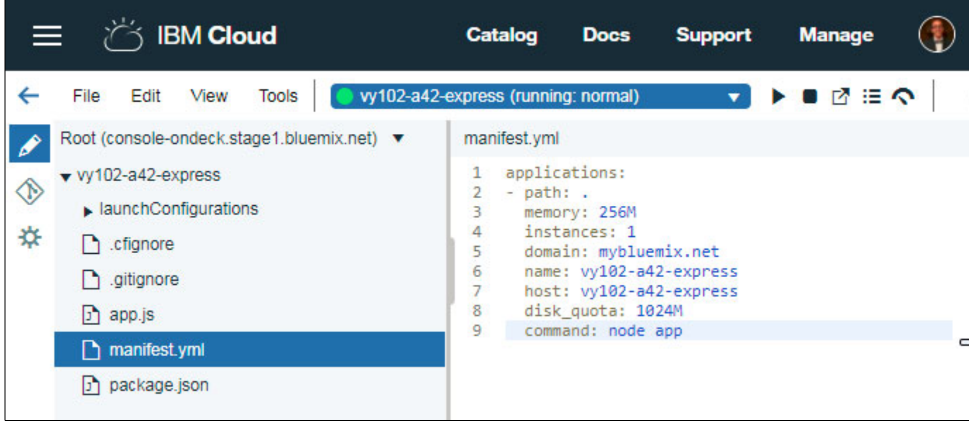


Figure 3-17 The manifest.yml file

The file is saved automatically. You can force save it by clicking **File** → **Save**.

4. Deploy the application from the workspace by completing these steps:

   a. In the server toolbar, select **Create new launch configuration** from the drop-down list. If you do not have the **Create new launch configuration** option, skip to step d.

   b. Click the “**+**” button to display the Edit Launch Configuration window.

   c. In the Edit Launch Configuration window, ensure that the **Organization** is set to your email address, and **Space** is set to dev, and click **Save**.

   d. Click the play icon (**Deploy the App from the Workspace**), which is highlighted in Figure 3-18. Then, click **OK** in the window that opens to confirm the action.

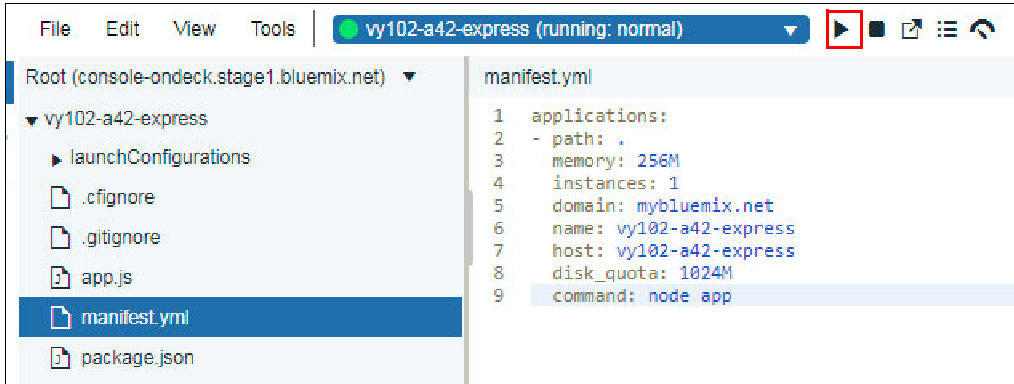


Figure 3-18 Deploy the application

   e. Notice that the Live Edit and debug options are visible on the top toolbar after the deployment is completed (Figure 3-19). The Live Edit and debug features are available only for the Node.js application. When IBM Cloud detects that you are creating a Node.js application (from the `manifest.yml` file), the two features become available on the toolbar.

> **Note:** IBM Cloud Lite account does not support the Live Edit feature. If you are using a Lite account, this option will not be displayed.

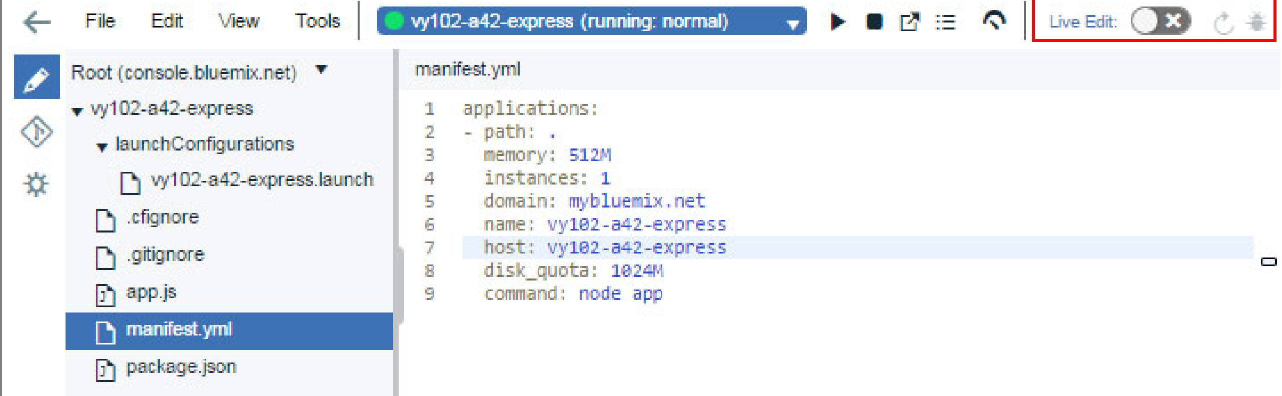


Figure 3-19 The Live Edit and debug features are now available

Note: While a Node.js application is in Live Edit mode, you can debug it on the Web IDE. You can edit code dynamically, insert breakpoints, step through code, restart the runtime, and more. You must use a Chrome browser to be able to use the debug feature.

5. Run the application by clicking the **Open the Deployed App** icon (Figure 3-20).

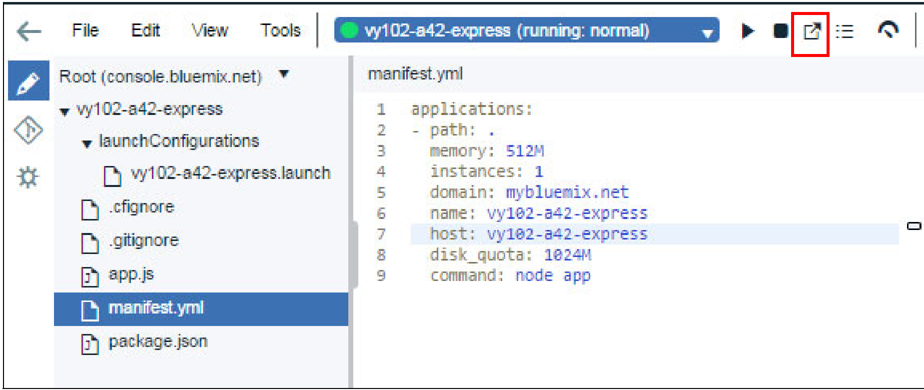


Figure 3-20 Open the application

A new tab opens with the application, and the “Hello Express!” text is returned (Figure 3-21).

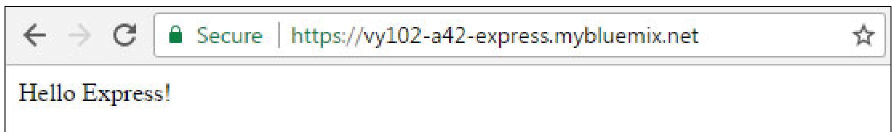


Figure 3-21 The Hello Express application

3.3.4 Create a simple HTML view and organize the code

In the next steps, you create a simple HTML page that has a form where the user enters the URL of the article. When the user clicks **Submit**, the article URL is posted to `/author` in the request body.

You will also organize the code by creating routing modules instead of having the routing handled in the `app.js` file.

Complete these steps:

1. Add `views/index.html` as the starting page (Figure 3-22):

   a. Close the browser tab where the application is running.

   b. Right-click `vy102-xxx-express` on the left bar, and select **New** → **Folder**.

   c. Name the folder `views`, and then press **Enter**.

   d. Right-click the `views` folder on the left bar, and select **New** → **File**.

   e. Name the file `index.html`, and then press **Enter**.

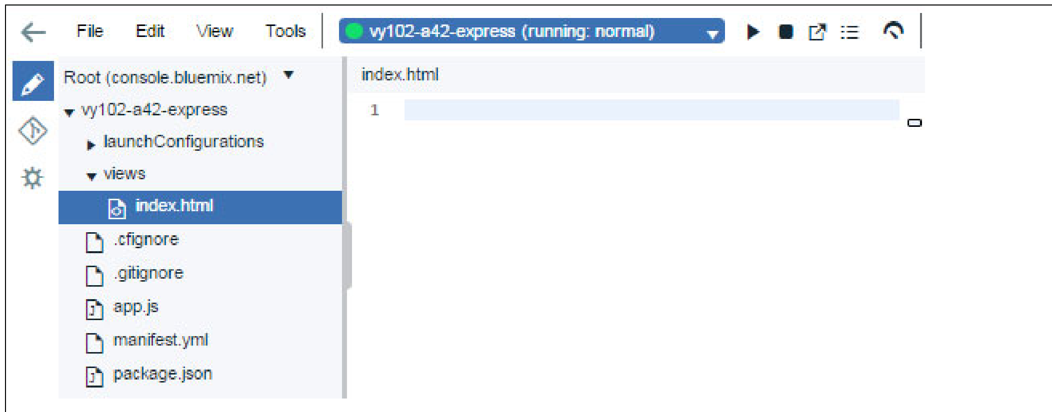


Figure 3-22 Empty index.html file

2. Copy the code snippet from Example 3-4 into the `index.html` file.

Example 3-4 Code snippet: HTML

```
<html>

<body>
    <h1 style="color:blue;">Watson Author Finder</h1>
    <p>To get information about the author of an article, enter the URL of that article.</p>
    <form action="author" method="post">
        <input type="text" name="url" />
        <input type="submit" value="Submit" />
    </form>
</body>

</html>
```

The HTML code indicates the following information:

- A heading that contains the words “Watson Author Finder” in blue.
- A paragraph that instructs the user what to do.
- A form that contains a text field and a **Submit** button. Upon submission, the parameters are submitted in the form of `x-www-form-urlencoded` in the body. In this code snippet, the only parameter is the URL of the article. The URL is submitted as a POST method to the `author` action, which triggers the POST /author route.

The `index.html` file now looks like the one shown in Figure 3-23.

```
1  <html>
2      <body>
3          <h1 style="color:blue;">Watson Author Finder</h1>
4          <p>To get information about the author of an article, enter the URL of that article.</p>
5          <form action="author" method="post">
6              <input type="text" name="url"/>
7              <input type="submit" value="Submit"/>
8          </form>
9      </body>
10 </html>
11 
```

Figure 3-23 The content of the index.html file

3. In `app.js`, change the root route (`GET /`) to send the `index.html` page to the caller:

   a. Open `app.js` from the left navigation bar.

   b. Add the following code snippet after the line that contains `app = express();`.

      This code snippet references the `path` module. The `path` module provides utilities for handling the directories, so it must point to the `index.html` file.

      ```
      var path = require('path');
      ```

   c. Update the callback function for `/ route`, which is the line after `app.get('/', function(req, res) {`. Change the code so that the callback function returns the `index.html` page instead of the words Hello Express!:

      ```
      res.sendFile(path.join(__dirname, 'views/index.html'));
      ```

      Also, change the comment before the line `app.get('/', function(req, res) {` to the following text:

      ```
      // In case the caller calls GET to the root '/',
      // return the content of index.html
      ```

   Figure 3-24 shows the `app.js` file with your updates.

```
▾ vy102-abc-express
    ▸ launchConfigurations
    ▾ routes
        author.js
        index.js
    ▸ views
    .cfignore
    .gitignore
    app.js
    manifest.yml
    package.json
```

```
1  var port = process.env.VCAP_APP_PORT || 8080;
2  
3  //Express Web Framework, and create a new express server
4  var express = require('express'),
5      app = express();
6  var path = require('path');
7  
8  // In case the caller calls GET to the root '/',
9  // return the content of index.html
10 app.get('/', function(req, res) {
11     res.sendFile(path.join(__dirname, 'views/index.html'));
12 });
13 
14 // In case the caller calls POST to /author, return 'Author na
15 app.post('/author', function(req, res) {
16     res.send('Author name');
17 });
18 
19 // start server on the specified port and binding host
20 app.listen(port);
```

Figure 3-24 Updated app.js

4. To be able to receive the request parameter, you must add a module named `body-parser`. The `body-parser` middleware module parses the data and populates the request object with the data under the `req.body` module. Complete these steps:

   a. *After* the line that contains `var path = require('path');`, add a reference to the third-party `body-parser` module:

```
var bodyParser = require('body-parser');

//parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }));
```

   b. Hover the cursor over `body-parser` and notice the warning message that is generated by Eclipse Orion Web IDE (Figure 3-25). The message indicates that the `body-parser` module is not defined in `package.json`.

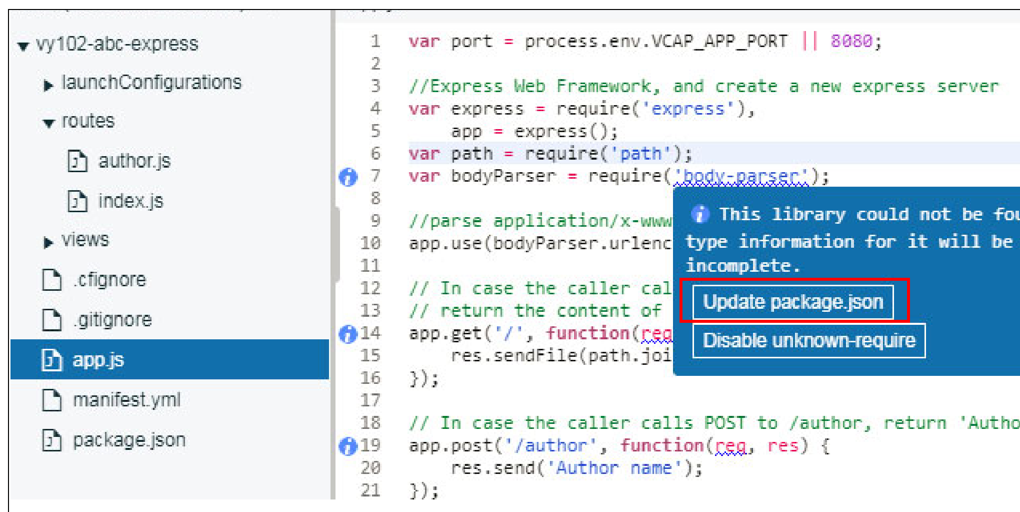


Figure 3-25 Missing module in package.json warning

   c. You can either update `package.json` manually to add dependencies to `body-parser` or click **Update package.json**, which updates the dependencies automatically.

   Open the `package.json` file and update the dependencies as shown in Example 3-5.

   > **Note:** Make sure that you add a comma after the `express` dependency line, as shown in the example

Example 3-5 Add body-parser

```
"dependencies": {
"express": "4.*",
"body-parser": "*"
}
```

   d. Open app.js and update the `callback` function for the POST /author route to send the URL to the user in response to `post /author` instead of sending `Author name`. Replace `res.send('Author name');` with the code snippet in Example 3-6.

Example 3-6 Code snippet: Send URL to user

```
res.send('You called the server requesting the author of the article: ' + req.body.url);
```

Change the comment before the line `app.post('/author', function(req, res) {` to the following text:

```
// In case the caller calls POST to /author, return the url of the article
```

e. Press **Shift** + **Alt** + **F** to format the code. Your updated `app.js` is shown in Figure 3-26.

```
vy102-abc-express
  launchConfigurations
  routes
    author.js
    index.js
  views
  .cfignore
  .gitignore
  app.js
  manifest.yml
  package.json
```

```
1  var port = process.env.VCAP_APP_PORT || 8080;
2
3  //Express Web Framework, and create a new express server
4  var express = require('express'),
5      app = express();
6  var path = require('path');
7
8  var bodyParser = require('body-parser');
9
10 //parse application/x-www-form-urlencoded
11 app.use(bodyParser.urlencoded({
12     extended: false
13 }));
14
15 // In case the caller calls GET to the root '/',
16 // return the content of index.html
17 app.get('/', function(req, res) {
18     res.sendFile(path.join(__dirname, 'views/index.html'));
19 });
20
21 // In case the caller calls POST to /author, return the url of the article
22 app.post('/author', function(req, res) {
23     res.send('You called the server requesting the author of the article: ' + req.body.url);
24 });
25
26 // start server on the specified port and binding host
27 app.listen(port);
```

Figure 3-26 Updated app.js file

5. In this step, you organize the code by moving all the routing to the routes module:

a. Right-click `vy102-xxx-express` on the left navigation bar, and then select **New** → **Folder**. Name the folder `routes` and press **Enter**.

b. Create `index.js` to handle all the routing related to the root resource, right-click the `routes` folder, select **New** → **File**, name the file `index.js`, and then press **Enter**.

c. Copy the code snippet from Example 3-7 and paste it into the `index.js` file.

Example 3-7 Code snippet: Using expres.Router

```
// index.js - Index route module
var express = require('express');
var router = express.Router();

//Provides utilities for dealing with directories
var path = require('path');

// Home page route
router.get('/', function (req, res) {
   res.sendFile(path.join(__dirname, '../views/index.html'));
});

module.exports = router;
```

This code snippet uses `express.Router`, which is introduced in Express 4 and provides an isolated instance of routes. It is used here to define an endpoint (URI) that handles the routing when a user sends a GET request to the home page of the application for this exercise.

The updated `index.js` file is shown in Figure 3-27.

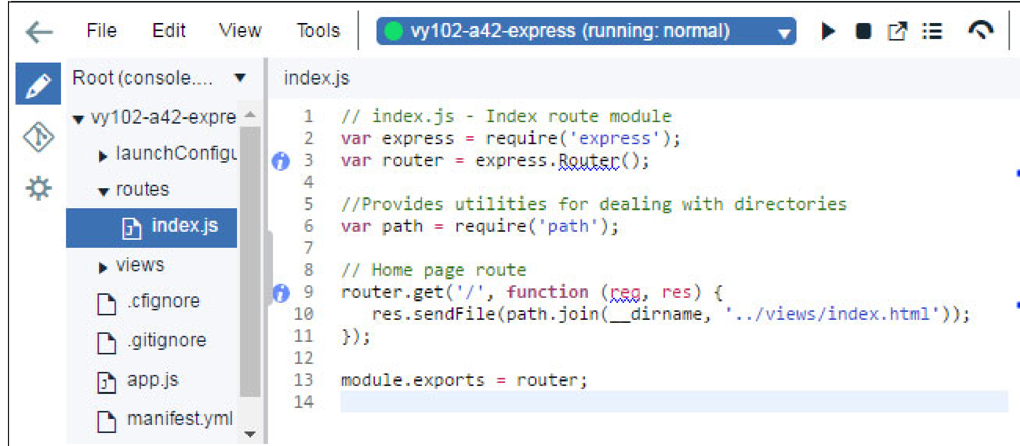


Figure 3-27 Updated index.js file

d. Create an `author.js` file to handle all the routing related to the /author resource. Right-click the `routes` folder, select **New** → **File**, name the file `author.js`, and then press **Enter**.

e. Copy the code snippet from Example 3-8 into the `author.js` file.

Example 3-8 Code snippet: Add to author.js file

```
// author.js - Author route module
var express = require('express');
var router = express.Router();

router.post('/', function (req, res) {
     res.send('You called the server requesting the author of the article: ' + req.body.url);
});

module.exports = router;
```

Figure 3-28 shows the updated `author.js` file.

Figure 3-28 Updated author.js file

f. Edit `app.js` to configure Express to use the index and author route modules:

   i. Click `app.js` on the left navigation bar.

   ii. Find lines of code between the comments `// In case the caller calls GET` and `// start server on the specified port and binding host.`

   iii. Replace those lines, including the comment `// In case the caller calls GET` with the code snippet from Example 3-9. Remove any unintended line breaks, especially in the comments, after pasting the code.

Example 3-9 Code snippet: Use app.js to bind routing modules

```
//Routes modules
var index = require('./routes'),
    author = require('./routes/author');

//In case the caller access any URI under the root /, call index route
app.use('/', index);

//In case the caller access any URI under /author, call author route
app.use('/author', author);
```

This example uses `app.js` to bind the routing modules that you defined previously with specific paths. The handling of the routing of any URI under root (/) is handled by the index routing module, and the handling of the routing of any URI under `/author` is handled by the author routing module.

Your updated `app.js` file is shown in Figure 3-29.

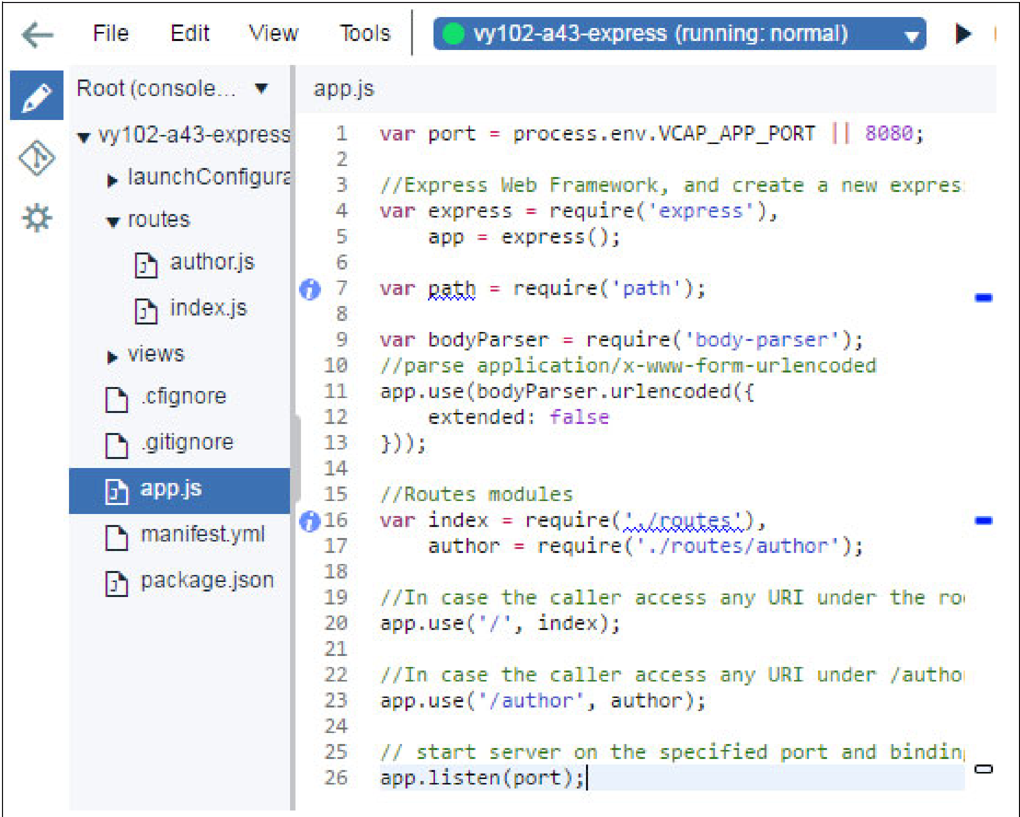


Figure 3-29 Updated app.js

6. Deploy the application and run it by completing these steps:
   a. Click the **Play** icon (**Deploy the App from the Workspace**) on the top toolbar to deploy the app. Confirm that you want to restart the app if prompted to do so.
   b. Wait for the deployment to complete.
   c. Click the **Open the Deployed App** icon on the top toolbar to run the application.

      The application opens in your browser (Figure 3-30).

Figure 3-30 Running the application

   d. In the text box, enter the URL for an article of your choice and click **Submit**.

      The route `POST /author` is then called (Figure 3-31).

      In this example, the following URL was used for testing:

```
https://www.forbes.com/sites/alexkonrad/2016/01/29/new-ibm-watson-chief-davi
d-kenny-talks-his-plans-for-ai-as-a-service-and-the-weather-company-sale
```

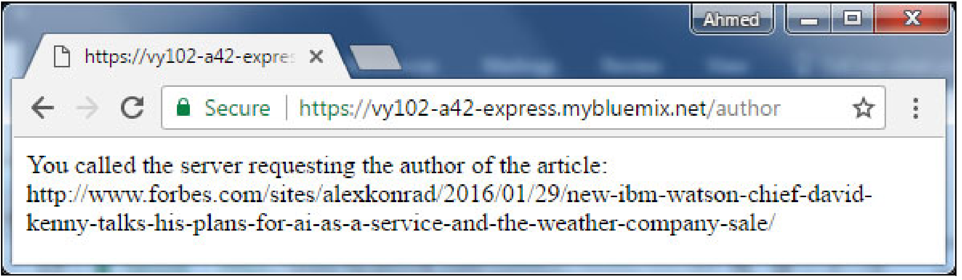


Figure 3-31 Author page

3.3.5 Integrate with Watson Natural Language Understanding service

The default Node.js framework includes only a minimal set of features. However, a large community of developers add to the Node.js framework through third-party libraries.

In this section, you extract the author name by calling the Watson Natural Language Understanding service. IBM Watson services provide REST APIs that you can use to add cognitive capabilities to your applications.

Natural Language Understanding uses natural NLP to analyze semantic features of any text, which can be plain text, HTML, or a public URL. Natural Language Understanding returns results for the features that you specify. The feature that you use in this exercise is metadata. It gets document metadata, including author name, title, RSS/Atom feeds, prominent page image, and publication date.

In Node.js, the Watson APIs are wrapped inside a third-party module that is named `watson-developer-cloud` that you will use in this exercise. The `watson-developer-cloud` module is developed and maintained by IBM. It is a third-party module because it is not developed as part of the Node.js foundation, and not packed into Node.js by default. To include the `watson-developer-cloud` module, you must add it as a dependency in the `package.json` file.

Complete these steps:

1. In this step, you add the Watson Natural Language Understanding service and bind it to the application.

   IBM Cloud services are a set of capabilities or functions delivered over the Internet that IBM Cloud hosts and manages. You can add services from the IBM Cloud catalog to your IBM Cloud application. Services provide a predefined endpoint that you can access from your application. The infrastructure for services is completely managed by IBM Cloud, and your app only must use the provided endpoint.

   You will bind the Natural Language Understanding service to your application so your application can use it by completing these steps:

   a. Close the running application tab and go back to the IBM Cloud DevOps page.

   b. Right-click **Catalog** and select **Open link in new tab** (Figure 3-32).

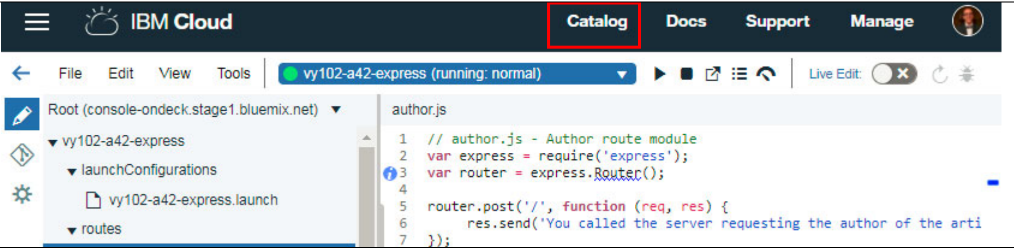


Figure 3-32 Eclipse Orion

c. In the IBM Cloud Catalog, click **Natural Language Understanding** (Figure 3-33).

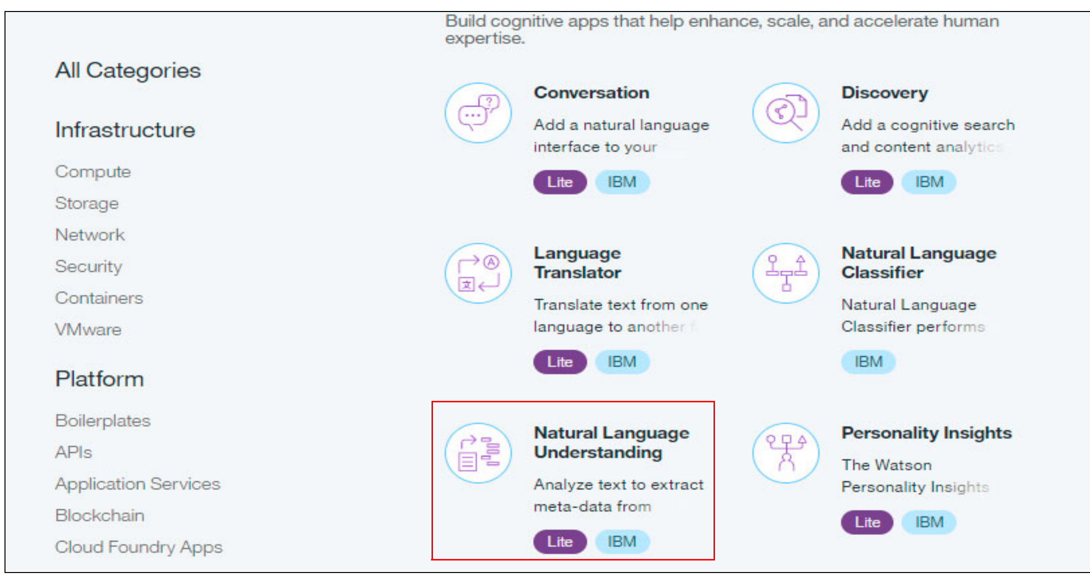


Figure 3-33 Catalog - Selecting Natural Language Understanding

The Natural Language Understanding page opens (Figure 3-34).

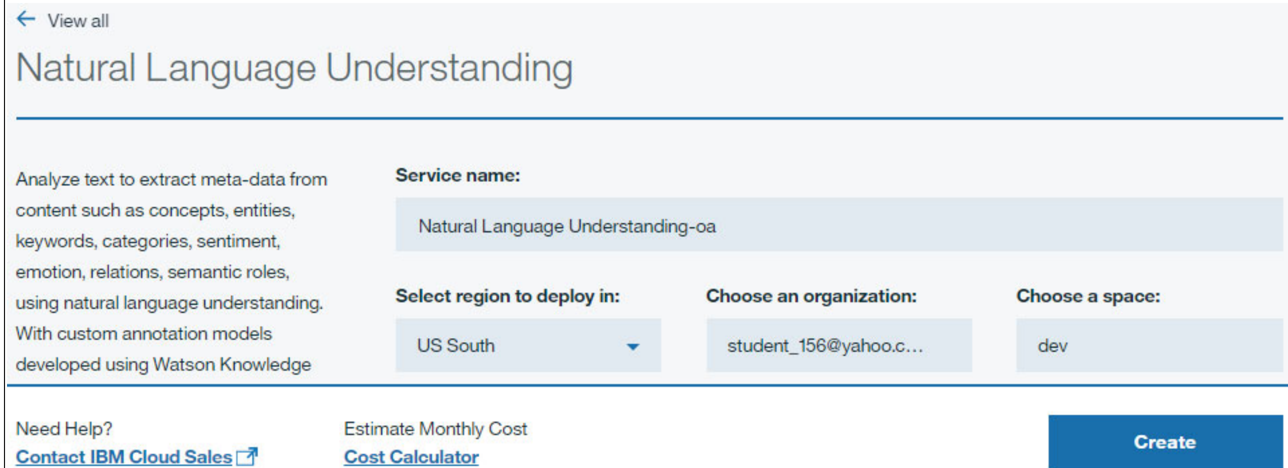


Figure 3-34 Natural Language Understanding creation page

d. Change the **Service name** field to `natural-language-understanding`. This is the name of the service.

e. Click **Create**.

f. Click **Connections** on the left pane.

g. Click **Create connection**, select your application **vy102-XXX-express**, and click **Connect** (Figure 3-35).

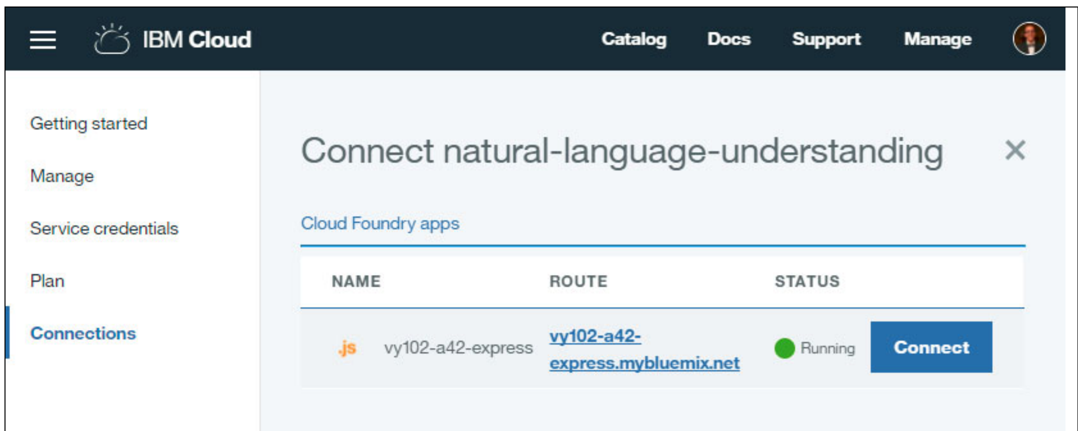


Figure 3-35 Connect the Natural Language Understanding service with the application

> **Important step:** You *must* perform this step because it binds the instance of the Natural Language Understanding service to the application so that your application can use the service.

h. To make the service available for use, click **Restage** on the dialog window that opens (Figure 3-36). Restaging the application means redeploying the application.

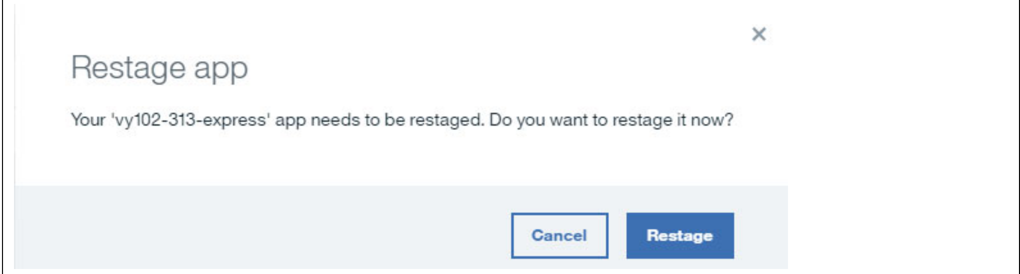


Figure 3-36 Restage app dialog box

i. Close the Service Details tab to return to IBM Cloud DevOps.

2. For integration with Watson, you will be using a module named `watson-developer-cloud`. Add a dependency for it in the `package.json` file:

   a. Open the `package.json` file.

   b. Add `watson-developer-cloud` as a dependency in `package.json`:

      i. Find the line “`body-parser`”: “`*`” (it can also be “`body-parser`”: “`latest`” if added automatically before). Add a comma (,) at the end of the line and press **Enter**.

      ii. On the new line, add the following code snippet:

```
"watson-developer-cloud": "2.25.1"
```

The updated `package.json` is shown in Figure 3-37.

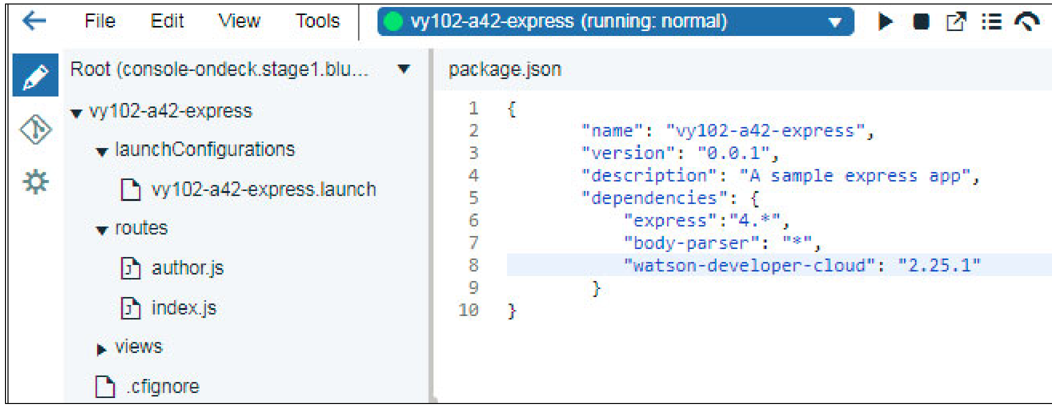


Figure 3-37 Updated package.json

3. Create a module named `articleServices` to handle all the business logic related to articles. This module has the function `extractArticleAuthorNames` that calls the Natural Language Understanding service, passing to it the article URL, and returning the list of author names. Complete these steps:

   a. Right-click **vy102-xxx-express** in the left navigation bar and then select **New** → **Folder**. Name the folder `services`, and then press **Enter**.

   b. Create `articleServices.js`. Right-click the `services` folder, select **New** → **File**, name the file `articleServices.js`, and press **Enter**.

   c. Create an instance of the third-party `node.js` module `watson-developer-cloud` by copying the code snippet from Example 3-10 to the `articleServices.js` file.

Example 3-10 Code snippet: Create instance of third-party Node.js module

```
// Watson Natural Language Understanding third party module
//Specify the release for the Natural Language Understanding service
var NaturalLanguageUnderstandingV1 =
require('watson-developer-cloud/natural-language-understanding/v1.js');
var natural_language_understanding = new NaturalLanguageUnderstandingV1({
   'version_date': NaturalLanguageUnderstandingV1.VERSION_DATE_2017_02_27
});
```

> **Unknown dependency:** Eclipse Orion might raise an `unknown dependency` warning message. You can ignore this warning because the parent module, `watson-developer-cloud`, was included previously. The `watson-developer-cloud` module contains the natural-language-understanding module.

   d. Create a function, `extractArticleAuthorNames` (Example 3-11) that calls the Natural Language Understanding service to extract the authors' names.

Example 3-11 Code snippet: extractArticleAuthorNames function

```
/*
 * Call Watson NLU Service to extract the list of author names for the requested article URL
*/
exports.extractArticleAuthorNames = function(req){
};
```

4. Prepare the parameters for calling the Natural Language Understanding service by placing the code snippet from Example 3-12 inside the `extractArticleAuthorNames` function.

Example 3-12 Code snippet: Parameters for calling Watson NLU

```
// url is the parameter passed in the POST request to /author
// It contains the URL of the article
// The metadata feature returns the author, title, and publication date.
    var parameters = {
        'url': req.body.url,
        'features': {
            'metadata': {}
        }
    };
```

5. Call the Natural Language Understanding service and return to the caller a callback function that contains the author's name. Complete these steps:

   a. Add `callback` as a parameter to the function:

   ```
   exports.extractArticleAuthorNames = function(req, callback){
   ```

   b. Call the analyze function from the `natural_language_understanding` third-party module. If the request succeeds, it returns a metadata object of the article. In this exercise, you are interested in the `authors` object in the metadata that returns the list of authors for the article.

   Copy the code snippet from Example 3-13 to the `extractArticleAuthorNames` function after the parameters variable initialization.

Example 3-13 Code snippet: Call the Natural Language Understanding service and return caller

```
// Call the Watson service and return the list of authors
    natural_language_understanding.analyze(parameters,  function(err, response) {
        if (err)
            callback(err,null);
        else
            callback(null,response.metadata.authors);
    });
```

6. If the URL passed is empty, return an error message to the user:

   a. Add the following error message as a constant after the initialization of the `natural_language_understanding` variable:

   ```
   //error message for missing URL
   const MISSING_URL_ERROR = 'URL not passed';
   ```

   b. Check whether the URL is empty. At the beginning of the `extractAuthorNames` function (line `exports.extractArticleAuthorNames = function(req, callback){}`, add the code snippet from Example 3-14 to return the error message if the URL is not defined.

   Example 3-14 Check URL

   ```
   //If the url is not passed, return error to the caller
   if(req===null||req.body===null||req.body.url===null){
      callback(MISSING_URL_ERROR,null);
      return;
   }
   ```

c. Press **Alt** + **Shift** + **F** to format the code.

The complete `articleServices.js` file now looks like the file in Figure 3-38.

File Edit View Tools vy102-mas-express (running: normal)

articleServices.js

```
1  //Natural Language Understanding Watson Third Party Module
2  // Specify the release number for the Watson NLU Service.
3  var NaturalLanguageUnderstandingV1 =
4      require('watson-developer-cloud/natural-language-understanding/v1.js');
5  var natural_language_understanding = new NaturalLanguageUnderstandingV1({
6      'version_date': NaturalLanguageUnderstandingV1.VERSION_DATE_2017_02_27
7  });
8
9  //error message for missing URL
10 const MISSING_URL_ERROR = 'URL not passed';
11
12 /*
13  * Call Watson NLU Service to extract the list of author names of the requested a
14  */
15 exports.extractArticleAuthorNames = function(req, callback) {
16     // If the url is not passed, return error to the caller
17     if (req === null || req.body === null || req.body.url === null) {
18         callback(MISSING_URL_ERROR, null);
19         return;
20     }
21
22     // url is the parameter passed in the post request to /author:
23     // It contains the URL of the article
24     // metadata feature of NLU returns the author of the webpage, the page
25     // title, and the publication date.
26     var parameters = {
27         'url': req.body.url,
28         'features': {
29             'metadata': {}
30         }
31     };
32
33     //Call the NLU Watson Service, and return the authors list
34     natural_language_understanding.analyze(parameters, function(err, response) {
35         if (err)
36             callback(err, null);
37         else
38             callback(null, response.metadata.authors);
39     });
40 };
```

Figure 3-38 The full code of the articleService.js file

7. Edit the author route to call `authorServices.extractArticleAuthorNames` instead of just returning the article URL by completing these steps:

   a. From the navigation bar, open the `routes` folder and select the `author.js` file.

   b. Add a reference to the `authorServices` module after the router variable initialization:

   ```
   var articleServices = require('../services/articleServices');
   ```

c. Edit / `route` to call the `extractArticleAuthorNames` function from the `articleServices` module. This process involves replacing the line `res.send('You called the server requesting the author of the article: ' + req.body.url);` with the code snippet from Example 3-15.

Example 3-15 Code snippet: Add call to extractArticcleAuthorNames function

```
articleServices.extractArticleAuthorNames(req, function(err, response) {
    if (err)
        res.status(500).send('error: ' + err);
    else
        res.send(response);
 });
```

The code indicates that in case of error, a 500 status code is returned to the user, which means Internal Server Error.

The updated `author.js` file is shown in Figure 3-39.

File Edit View Tools vy102-a43-express (running: live edit)

author.js

```
1  // author.js - Author route module
2  var express = require('express');
3  var router = express.Router();
4
5  var articleServices = require('../services/articleServices');
6
7  router.post('/', function(req, res) {
8      articleServices.extractArticleAuthorNames(req, function(err, response) {
9          if (err)
10             res.status(500).send('error: ' + err);
11         else
12             res.send(response);
13     });
14 });
15
16 module.exports = router;
```

Figure 3-39 Updated author.js

3.3.6 Deploy the application and run it

Other ways to deploy the application exist in addition to deploying it from Eclipse Orion Web IDE. In the next steps, you push the code to the Git repository, and then the IBM Cloud Delivery Pipeline automatically builds and deploys the code.

By default, enabling continuous delivery for a project creates a DevOps toolchain for your project. The toolchain includes a Git repository that is based on GitLab. *Git* is an open source change management system.

The Git repository perspective in the IBM Cloud DevOps Services web IDE supports common Git commands to manage your code. You can also develop your application on your own workstation and commit your changes to the Git repository with a standard Git client.

By default, IBM Cloud Delivery Pipeline services automatically run the build and deploy tasks when you commit changes to the Git repository.

Complete these steps:

1. Switch to the Git perspective by clicking the **Git** icon on the left toolbar (highlighted in Figure 3-40).

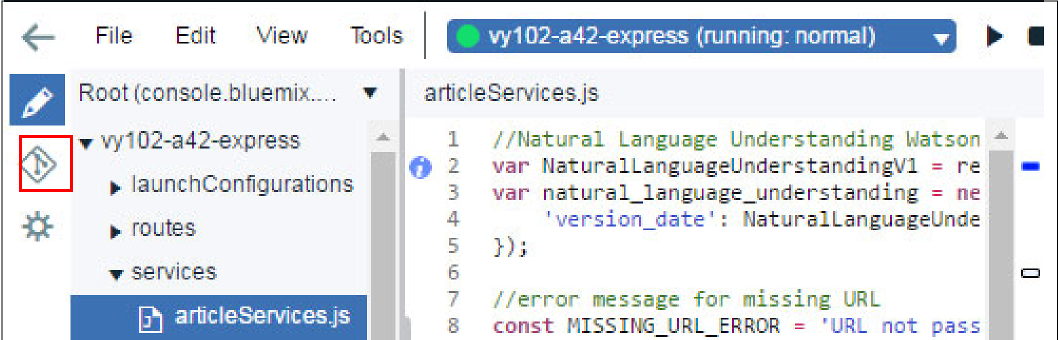


Figure 3-40 IBM Cloud DevOps

2. Notice that all the changed files are listed in the Working Directory Changes window (Figure 3-41). Enter a descriptive message for commit, for example `Watson Author Finder - Initial Code`, and then click **Commit**.

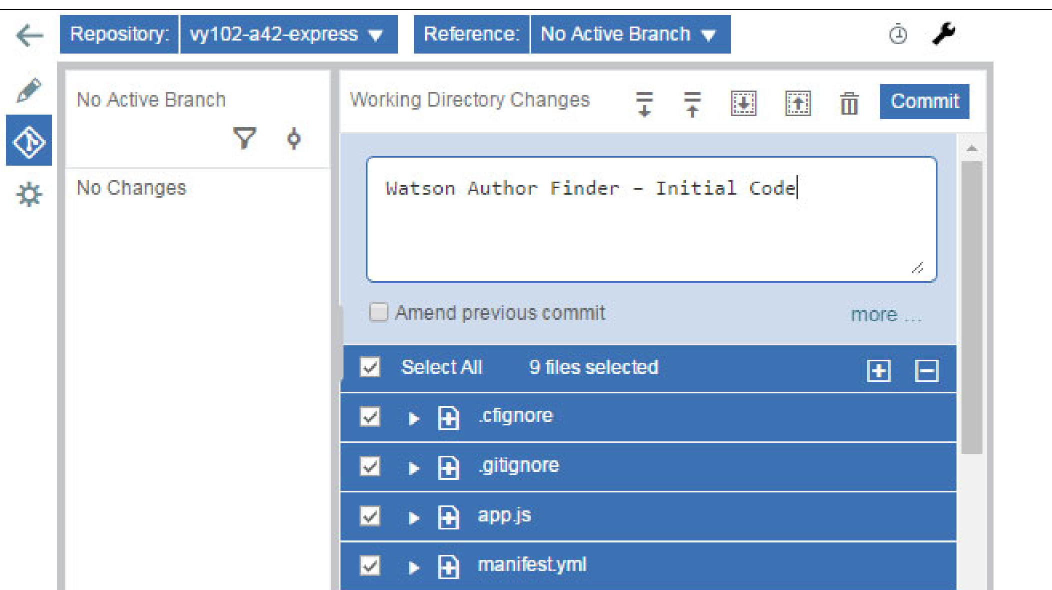


Figure 3-41 IBM Cloud DevOps: Git

3. Your change is displayed in the outgoing window (Figure 3-42). Click **Push**.

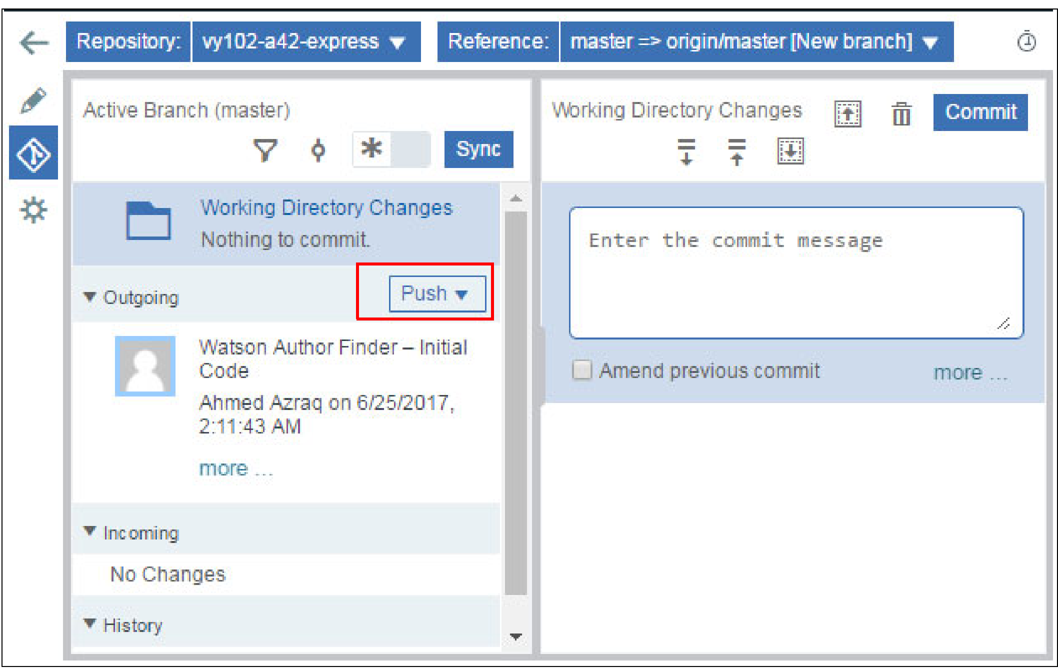


Figure 3-42 IBM Cloud DevOps: Git

4. After the application is pushed to Git, the Delivery pipeline automatically builds and deploys the application. Complete these steps:
   a. Close the IBM Cloud DevOps Git tab and return to the Application Details.
   b. Scroll down to the Continuous delivery tile and then click **View toolchain** (Figure 3-43).

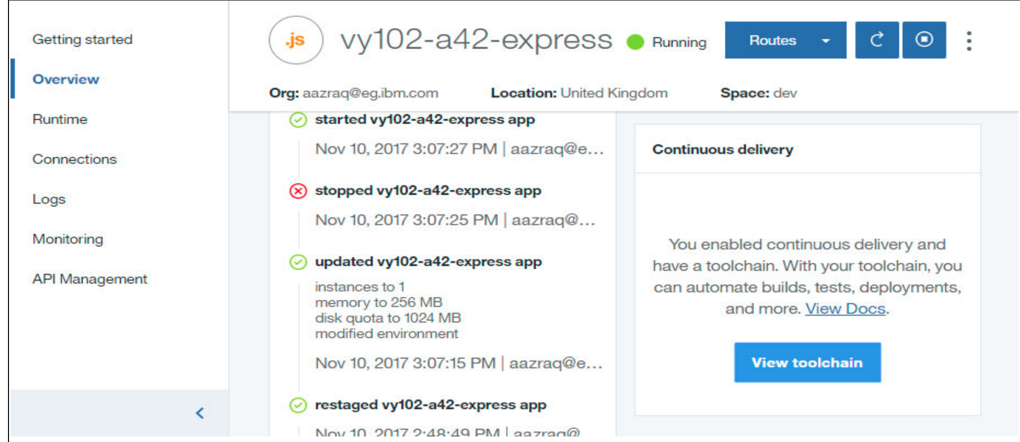


Figure 3-43 Application Details

c. Click **Delivery Pipeline** (Figure 3-44).

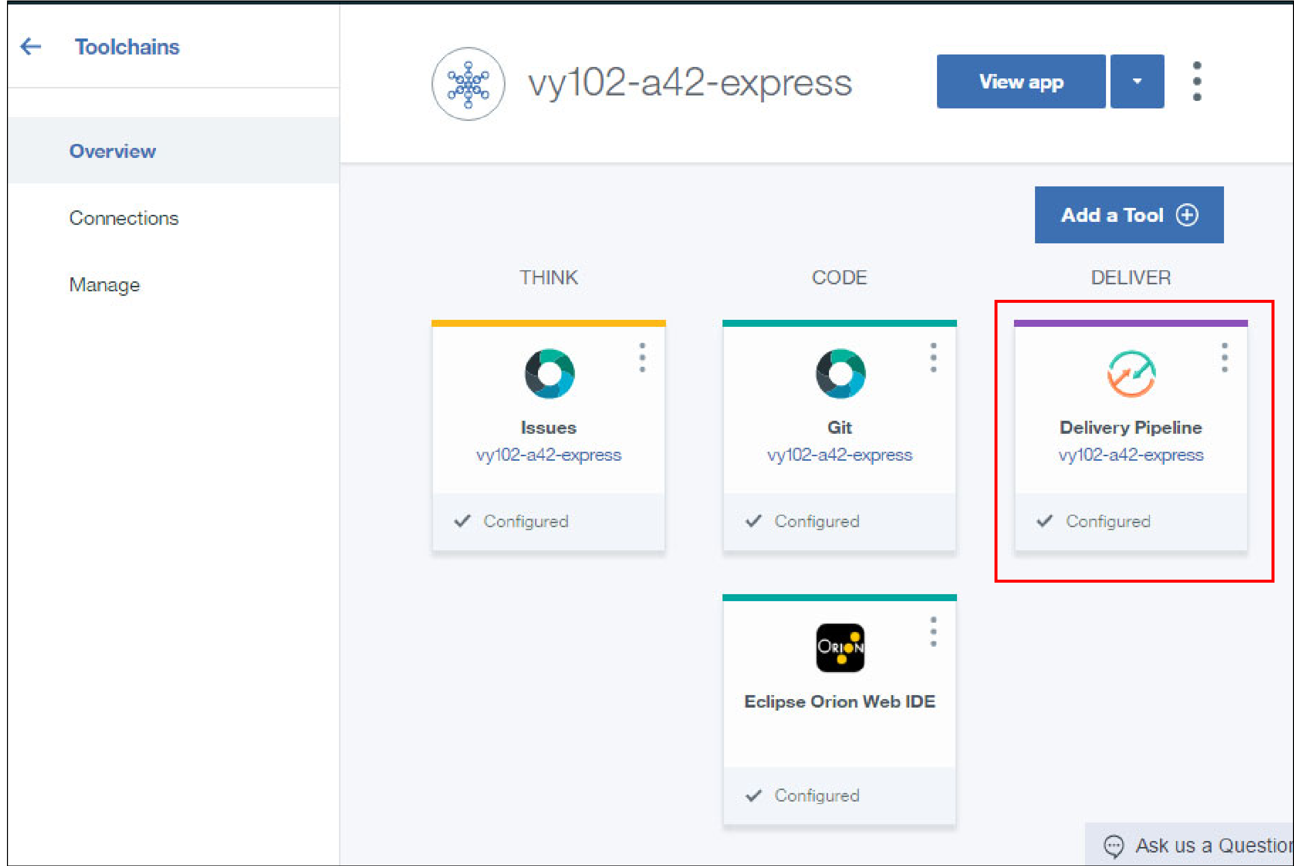


Figure 3-44 IBM Cloud DevOps toolchain

d. Wait until all the jobs at the Build Stage and Deploy Stage are completed. Figure 3-45 shows that the Deploy Stage is still running.

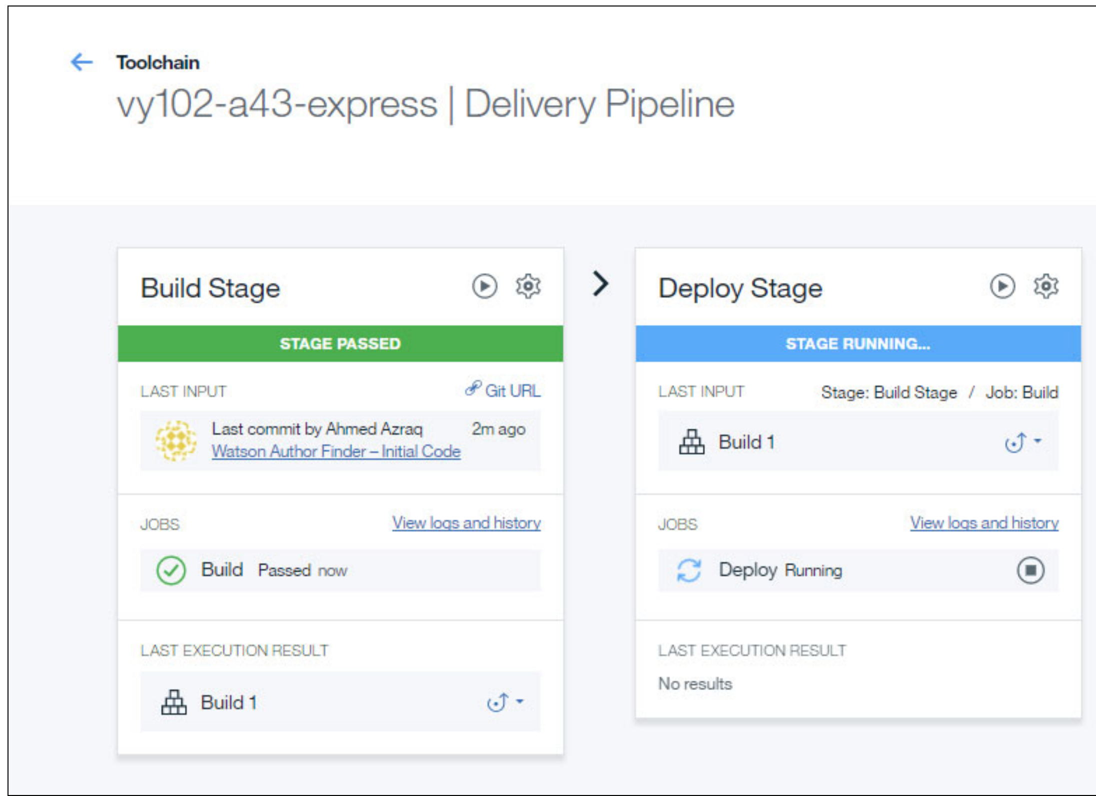


Figure 3-45 Delivery Pipeline: Deploy Stage is in-progress

Figure 3-46 shows that both stages are completed.

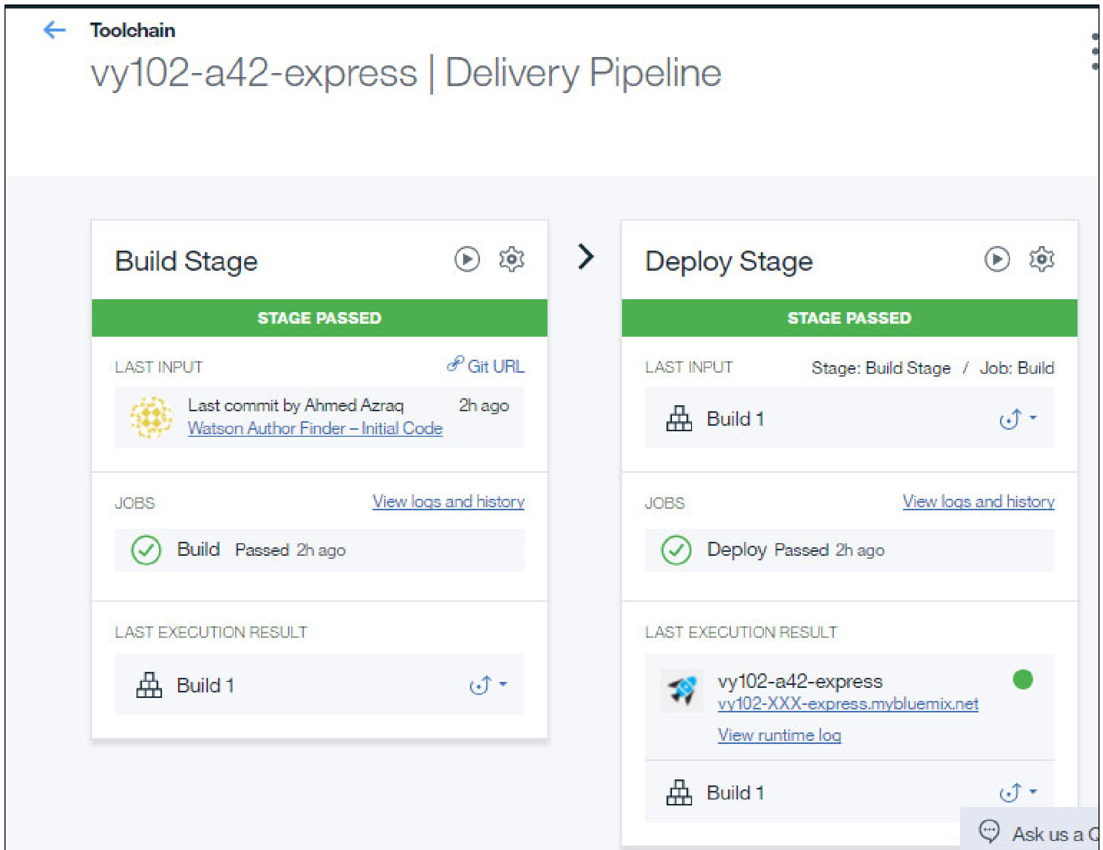


Figure 3-46 Delivery Pipeline: Build Stage and Deploy Stage passed

e. To run the application, click the URL of the last execution result of the Deploy Stage (Figure 3-47).

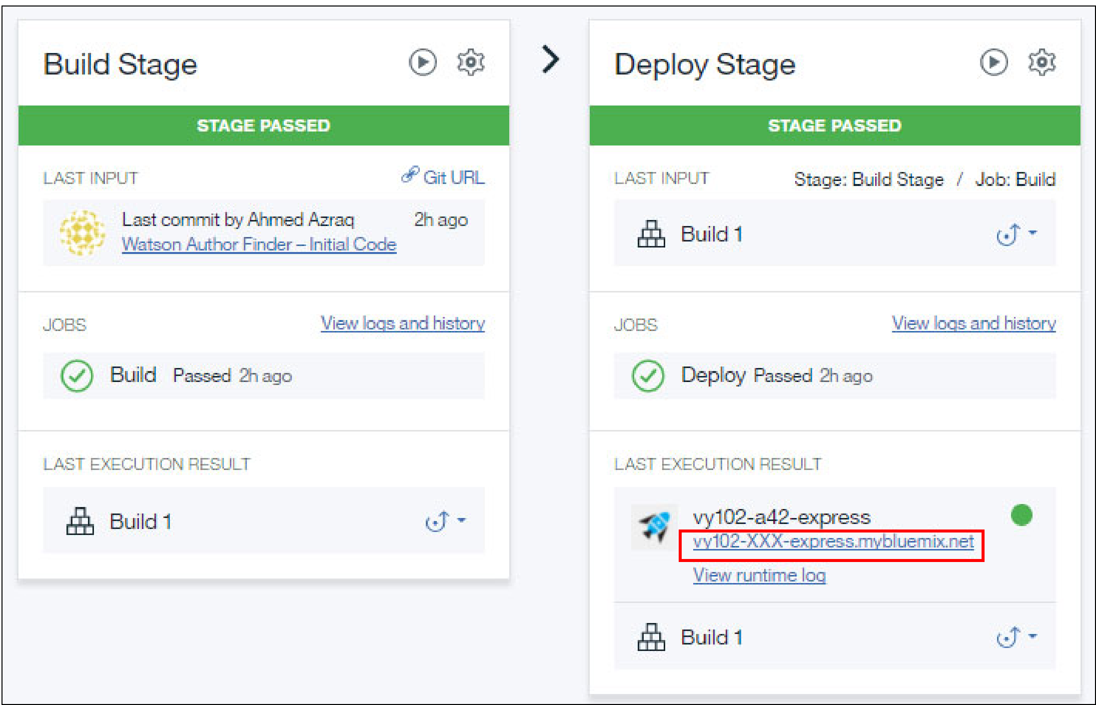


Figure 3-47 Delivery Pipeline: Click URL

f. Run the application by entering the URL of any article and then clicking **Submit** (Figure 3-48).

The following is an example of an article URL:

```
https://www.forbes.com/sites/alexkonrad/2016/01/29/new-ibm-watson-chief-davi
d-kenny-talks-his-plans-for-ai-as-a-service-and-the-weather-company-sale
```

Figure 3-48 Watson Author Finder application

The author name is retrieved from the Natural Language Understanding service (Figure 3-49).

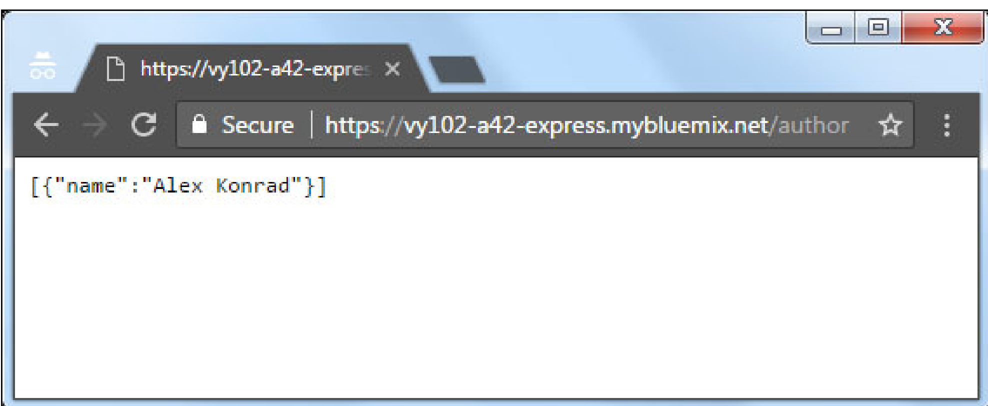


Figure 3-49 Watson Author Finder returned results

3.4 Exercise review

During this exercise, you accomplished the following goals:

- Created Hello World Express Application that includes two routes handling the URIs `GET /`, and `POST /author`. You should now understand the basics of Express Framework.
- Sent the index.html page to the caller of the GET URI. You should now know how to use Express to send an HTML page to the user in response to a route.
- Learned how to integrate a Node.js application with Watson Natural Language Understanding service.
- Organized the code into routes, views, and services. By following the steps, you should now know about preferred practices for organizing Express applications in Node.js.

4

Building a rich front-end application by using React and ES6

This chapter guides you through building an interactive and rich client-side application by using React. The app demonstrates the use of React components. It uses the *Fetch* API with ECMAScript 6 (ES6) to communicate with Node.js back-end services that call the IBM Watson Natural Language Understanding service to extract the authors of online articles that are selected by the user.

The sample application demonstrates the integration of the React front-end application with server-side Node.js services.

This chapter contains the following topics:

- Getting started
- Architecture
- Step-by-step implementation
- Exercise review

© Copyright IBM Corp. 2017. All rights reserved.

4.1 Getting started

To start, read the objectives, prerequisites, and expected results of this chapter.

4.1.1 Objectives

By the end of this chapter, you should be able to accomplish these objectives:

- Clone an IBM Cloud application.
- Use React to create interactive web pages.
- Use the Fetch API to interact with back-end web services.
- Understand the following concepts of ES6:
  - Classes.
  - Arrow functions.
  - Promises.

4.1.2 Prerequisites

Before you start, be sure that you meet these prerequisites:

- Basic JavaScript skills
- Basic HTML 5 skills
- An IBM Cloud account available at `https://console.bluemix.net/`.
- An understanding of Cloud DevOps basic concepts
- An understanding of Git basic concepts
- Access to a web browser: Google, Chrome, or Mozilla Firefox

4.1.3 Background concepts

In this exercise, you use ES6, React, and Bootstrap to build the front-end application. This section briefly introduces the concepts used.

ECMAScript 6

Most modern UI frameworks and new JavaScript APIs require an understanding of ES6. This exercise uses the ES6 Fetch API to call back-end services, classes to create the React Component, and the arrow function to simplify the code.

In 2015, the European Computer Manufacturers Association (ECMA), released a new version of the JavaScript standardization (ES6), which was considered a significant upgrade to the JavaScript language, since the standardization done in 2009, which was called ES5. ES6 introduced a set of new features and s*yntactic sugar*, to JavaScript language.

> **Syntactic sugar:** Syntactic sugar is a term used in programming languages to define the syntax introduced to make writing the code easier. It does not add new features. For example, sum = sum +1; //The syntactic sugar is **`sum++`**;.
>
> Both syntaxes are still valid and perform the same functions.

This chapter covers only four ES6 features:

- `let` keyword

  `let` is a keyword used to refer to a variable. It is similar to the `var` keyword, but the main difference is that the variable defined with `let` is visible only within the scope it is defined in. In contrast, `var` can be accessible outside its defined scope. Example 4-1 shows the difference between the `var` and `let` keywords:

  Example 4-1 Difference between var and let keywords

```
for (var i = 0; i < 10; i++) {
    console.log(i);
}
// The variable i is accessible here, although it was defined inside the for loop.
console.log(i);
for (let k = 0; k < 10; k++) {
    console.log(k);
}
// The following line is invalid, because k is not accessible outside the for loop.
console.log(k);
```

- Classes

  ES5 already included object-oriented and inheritance capabilities provided by `prototype`. ES6 introduces the `class` syntax, which is a syntactic sugar to make it easier to program.

  Example 4-2 shows how to define a class by using `prototype` in ES5. Notice that in ES5 a class is defined by using the `function` keyword, which is confusing.

  Example 4-2 Defining a class by using prototype in ES5

```
function Rectangle(x, y) {
    this.x = x;
    this.y = y;
}

Rectangle.prototype.draw = function() {
    // your rectangle drawing code goes here
}
// Initialize Rectangle.
var rect = new Rectangle(2,3);
```

  Example 4-3 shows how to define a class in ES6 by using the `class` syntax. This approach eliminates the confusion and is better aligned with object-oriented programming syntax.

  Example 4-3 Defining a class using ES6 syntactic sugar

```
class Rectangle {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    draw() {
        // your rectangle drawing code goes here
    }
}
let rect = new Rectangle(2,3);
```

- Arrow function

  Classical JavaScript function syntax does not provide flexibility for defining a function with just one statement when compared to defining a function with a longer body. Regardless of the number of statements in the function, you always must enter function () {}. The arrow function simplifies the syntax for a simple single-line function.

  Example 4-4 shows a simple function definition in ES5.

  Example 4-4 Defining the add function in ES5

```
var add = function(a, b) {
    return a + b;
}

console.log(add(3, 4));
```

  Example 4-5 shows the same function with a simplified arrow syntax in a single line.

  Example 4-5 Same example with ES6

```
let add = (a, b) => a + b;

console.log(add(3, 4));
```

- Promises

  Promises are a compelling alternative to callback functions when dealing with asynchronous code. Unfortunately, promises can be confusing. However, significant work has been done to bring out the essential advantages of promises in a way that is interoperable and verifiable.

  Note: There are several promises libraries, but the one introduced in ES6 is based on the Promises/A+ specification. For more information, see *Promises/A+* at `https://promisesaplus.com/`.

  The core component of a promise object is its `then` method. `then` takes two optional callback functions as arguments (*fulfillment value* and *rejection reason*). The `then` method is how you get the return value (known as the *fulfillment value*) or the exception thrown (known as the *rejection reason*) from an asynchronous operation.

  Example 4-6 shows calling an asynchronous function and handling the returned promise by using two callback functions, which are called `onFulfilled` and `onRejected` in this example.

  Example 4-6 Promises with ES6

```
var promise = doSomethingAync()
promise.then(onFulfilled, onRejected)
```

Example 4-7 and Example 4-8 illustrate the difference between handling callback functions in ES5 and ES6..

Example 4-7 shows a sample code written in ES5 to read a file. It uses asynchronous callback functions to process the file and handle the error.

Example 4-7 Reading files and handling errors in ES5

```
readFile(function(err, data) {
    if (err) {
        console.error(err);
    }
    console.log(data);
});
```

With ES6 promises, the code looks like in Example 4-8.

Note that the `data` argument in `console.log` and the `err` argument in `console.error` do not need to be specified. The code can be simplified as shown in the example because the interpreter implicitly passes the argument to functions that accept only one parameter.

Example 4-8 Reading files and handling errors in ES6

```
var promise = readFile()
promise.then(console.log, console.error)
```

You can pass the promise around and anyone with access to the promise can consume it using `then` regardless whether the asynchronous operation has completed or not. You are also ensured that the result of the asynchronous operation will not change for any reason because the promise will be resolved only once (either fulfilled or rejected).

> **Note**: It is helpful to think of `then` not just as a function that takes two callbacks (`onFulfilled` and `onRejected` in this example), but as a function that unwraps the promise to reveal what happened from the asynchronous operation. Anyone with access to the promise can use `then` to unwrap it.

React concepts

In this exercise, you use React to build a client-side application that takes the URL of an article published on the internet as input from the user and communicates asynchronously with a back-end Node.js service to extract the author of the article.

React is a framework for building client-side dynamic web applications. React uses dynamic data binding and virtual Document Object Model (DOM) to extend HTML syntax and to eliminate the need for code that keeps the UI elements synchronized with the application state. This code, sometimes referred to as *glue code*, serves solely to synchronize the state of the UI components with changes in the back-end application. It does not add business value capabilities to your solution. React saves you the time and effort needed to create this code.

Some of the basic concepts of React applications are as follows:

- Virtual DOM

  HTML pages use what is called DOM to render, traverse, and update UI elements in a responsive way. JavaScript code traverses the DOM tree to create, update, or hide HTML components. JavaScript traverses the DOM by using browser-specific application programming interfaces (APIs).

  Sometimes this approach creates compatibility issues across browsers. Also, some browsers implement the function more efficiently than others. For example, you might experience better performance when using Chrome to access a website compared to another browser.

  Figure 4-1 shows the DOM of the HTML page for the Watson Author Finder application developed in Chapter 3, “Creating your first Express application” on page 53.

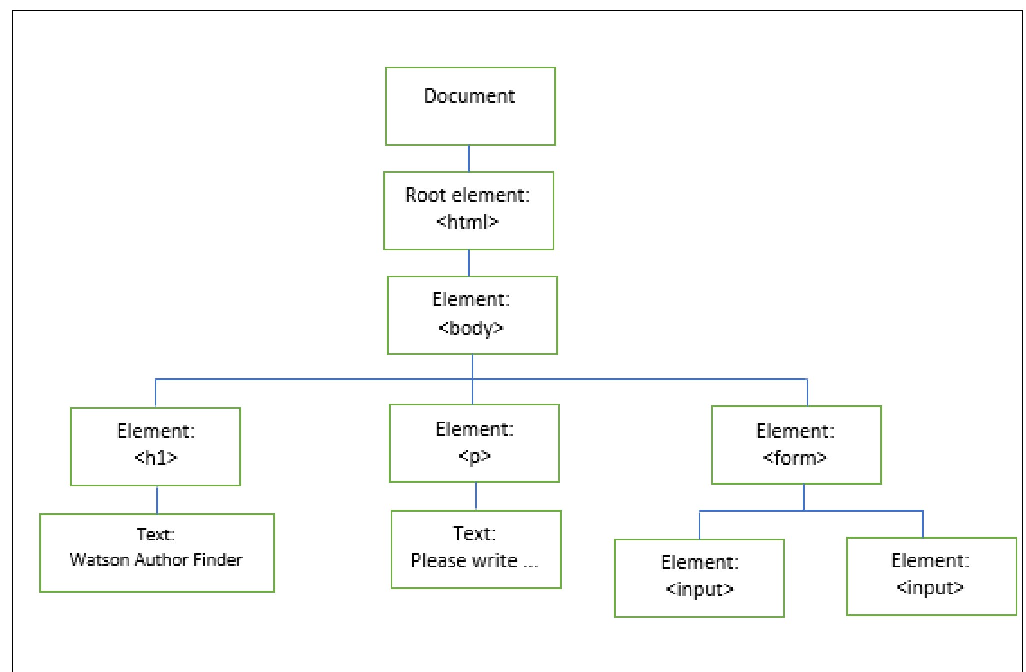


Figure 4-1 DOM of the HTML page for the Watson Author Finder application

Example 4-9 shows the HTML code represented by the DOM in Figure 4-1.

Example 4-9 Code snippet: HTML code for application Watson Author Finder

```
<html>
   <body>
      <h1 style="color:blue;">Watson Author Finder</h1>
      <p>To get information about the author of an article, enter the URL of that article.</p>
      <form action="author" method="post">
         <input type="text" name="url"/>
         <input type="submit" value="Submit"/>
      </form>
   </body>
</html>
```

React addresses this issue by using the concept of virtual DOM to provide an abstraction of the HTML DOM. With this approach, the responsibility for traversing the virtual DOM lies on the React framework, which makes it independent of the browser.

Example 4-10 shows the same HTML code written in React. Note the following in the code:

– The `AuthorCheckForm` class is equivalent to the `<form>` tag.
– The `InputUI` class is equivalent to the content of the form.

Example 4-10 HTML code in Example 4-8 written in React

```
class AuthorCheckForm extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return (
            <form action="author" method="post">
              <InputUI label='URL'/>
            </form>
        )
    }
}

class InputUI extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return (
            <div>
             <a>
              {this.props.label}
            </a>
            <input type="text" name="url"/>
            <input type="submit" value="Submit"/>
         </div>
        )
    }

}
```

- React components

  React components are reusable pieces of the UI. Each component extends the React class `component`, which has lifecycle methods that describe the behavior of the component. Conceptually, components are like JavaScript classes. Your implementation can override the `component` lifecycle methods to specify the behavior of each reusable piece.

  The `component` lifecycle methods are `constructor()` and `render()`. You initialize variables, including the `state`, in the `constructor()` method and write the HTML code in the `render()` method.

  Another important property of the component is that it can be used to keep the state of the UI.

Example 4-11 shows the basic skeleton of a React component.

Example 4-11 React component

```
   class InputUI extends React.Component {
       constructor(props) {
           super(props);
       }

       render() {
           return (<p> hello world </p>)
       }
}
```

- React components properties

  A React component property is similar to an attribute in an HTML tag. In React, you use a custom tag to define component properties instead of using custom attributes for an HTML tag. In Example 4-12, the component is `InputUI` and the property is `label`.

- React element

  The React element is similar to standard HTML tags, such as `<h1>`, `<body>`, and so on. The difference between React elements and HTML DOM elements is that React elements can also be used to render the components.

  Example 4-12 shows a React element rendering the InputUI component.

Example 4-12 React element

```
<InputUI label='URL' />
```

Example 4-13 shows React element example for a standard HTML tag.

Example 4-13 React element

```
<input type="submit" value="Submit"/>
```

- React state

  To make the React components dynamic, you initialize the state in the `constructor()` method and render the component data from the state. The state is updated by events, for example an Ajax response. React renders the components to display the updated state.

  States and virtual DOM complement each other to achieve the high performance of React. React uses the virtual DOM to compute what the DOM of the final web page should look like after the state of some elements is updated. The calculations happen in-memory on the virtual DOM. The result of this algorithm is how the HTML DOM should look.

- JavaScript XML (JSX)

  JSX simplifies code by enabling developers to write XML-like syntax in JavaScript code. JSX is commonly used in React, but current browsers do not support JSX out of the box. Writing code with JSX requires the use of the Babel JavaScript compiler (also known as *transpiler*) to convert JSX into JavaScript supported by the browsers.

  Example 4-14 shows creating an element in React by using JSX.

Example 4-14 Creating an element by using JSX

```
var sum = 3+2;
<input type=”text” value={sum}/>
```

Example 4-15 shows creating the same element in React without using JSX.

Example 4-15 Creating the same element using JavaScript

```
var sum = 3+2;
React.createElement("input",{"type":"text","value":{sum});
```

As a developer, you can program in React with JavaScript without using JSX. However, JSX is currently the preferred language for React programmers.

Bootstrap

Bootstrap is an HTML, CSS, and JavaScript front end for developing responsive web pages.

Bootstrap uses a *responsive grid layout*. Each Bootstrap page is arranged into nested rows and columns. Each row holds exactly 12 columns. Classes are assigned to columns and used to determine the width of the column at different screen sizes. For example, `div` tag with class col-md-6 takes exactly half the screen width (6/12) on screens of medium size and larger. A `div` with both classes `col-xs-12` and `col-md-6` takes the full screen width on extra small and small screens, and half the width on medium and large screens.

For more information, see the Bootstrap documentation at this URL:

https://getbootstrap.com/docs/4.0/getting-started/introduction/

4.1.4 Expected results

Figure 4-2 shows the Author Finder application displaying the list of authors extracted from an article published on the CNN website.

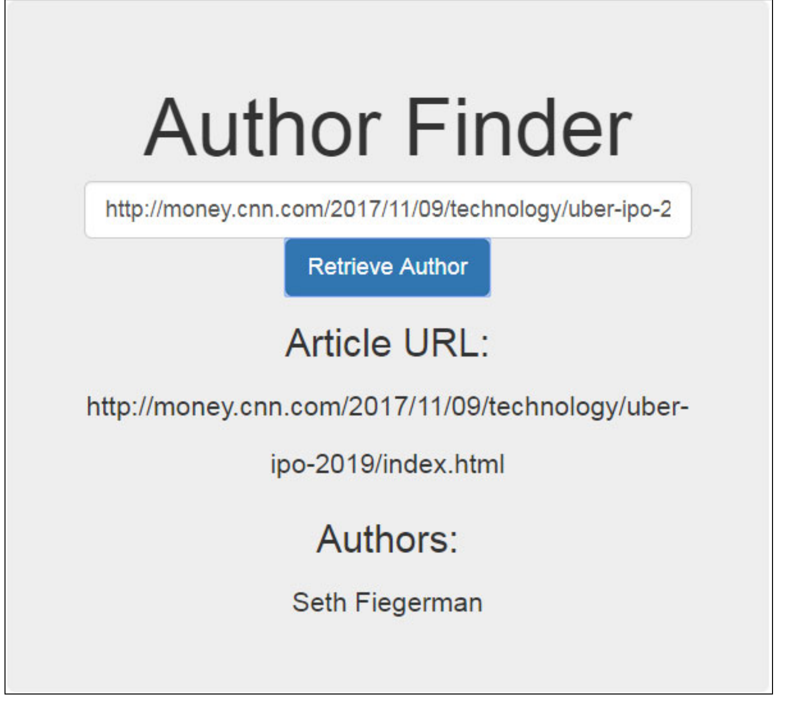


Figure 4-2 React app Watson Natural Language Understanding Author Finder

Here is how it works:

1. The user enters the URL of an article that has one or more authors.
2. The user clicks **Retrieve Author**.
3. React sends a request to the Node.js web service by using the Fetch API.
4. The web service calls the Watson Natural Language Understanding service to parse the article and extract the authors.
5. The result is passed back to the React state, which renders it in a responsive grid.

4.2 Architecture

Figure 4-3 shows the components and runtime flow of the application.

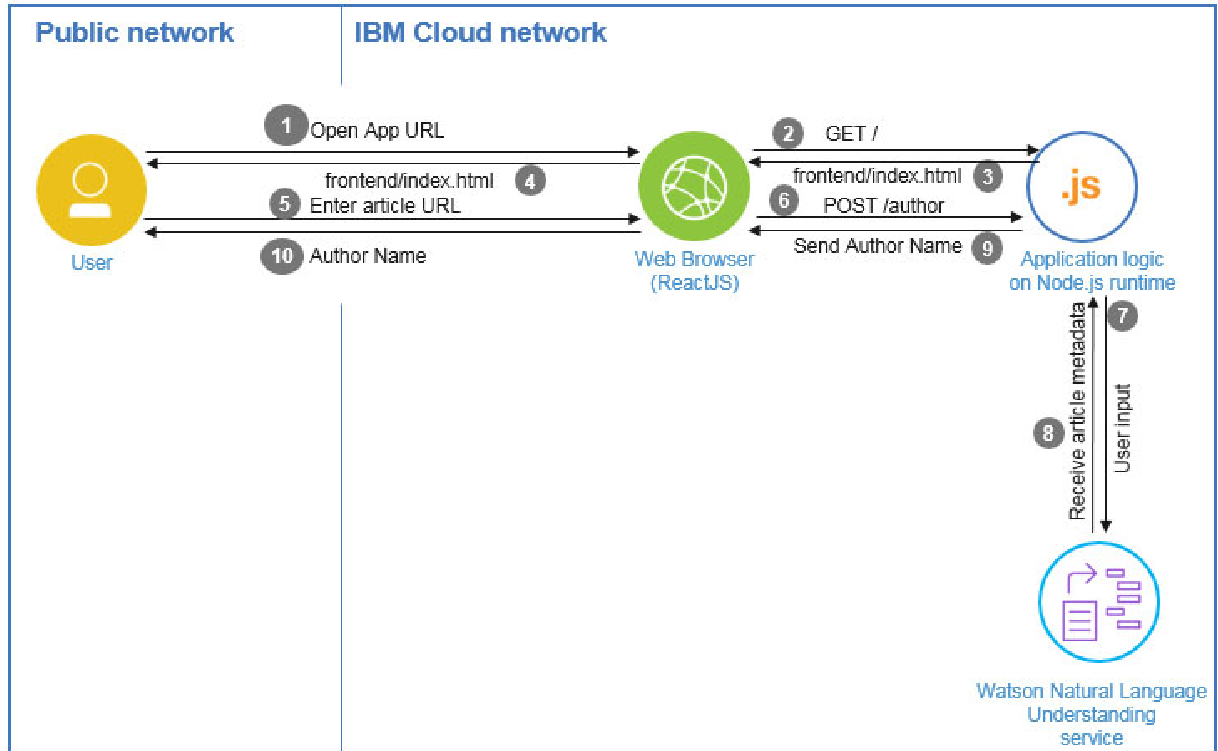


Figure 4-3 Architecture

Figure 4-3 describes these steps:

1. In a web browser, the user navigates to the application URL:

   `http://vy102-xxx-react.mybluemix.net`

2. The web browser sends a `GET /` request to the Node.js back-end application developed in Chapter 3, “Creating your first Express application” on page 53.
3. The Express framework in Node.js returns `frontend/index.html`.
4. The `frontend/index.html` page is returned to the user. The `index.html` page contains a form that has one text box and a **Submit** button. The text box is where the user enters the URL of an article.
5. The user enters the article URL and then clicks **Submit**.

6. The web browser sends a POST request to the `/author` route with the article URL passed in the body.
7. The Express framework in Node.js passes the article URL to the Watson Natural Language Understanding service. It also specifies that metadata should be returned.

> **IBM Watson Natural Language Understanding service:** This service uses natural language processing (NLP) to analyze semantic features of a text. Watson Natural Language Understanding service has many features, such as concepts, categories, emotion, entities, keywords, metadata, and sentiment.
>
> The feature used in this exercise is *metadata*. It retrieves document's metadata, including the author name, title, RSS/ATOM feeds, prominent page image, and publication date.

8. The metadata for the article is returned by the Watson Natural Language Understanding service.
9. Node.js filters the metadata to return only the author names.
10. The authors of the article are returned to the user through the web browser.

4.3 Step-by-step implementation

The implementation involves the following steps:

1. Clone the express application from Git using the Delivery Pipeline.
2. Create your first React page.
3. Add a dynamic form to the page.
4. Add more components to the form.
5. Use the Fetch API to call the Node.js author service.

4.3.1 Log in to IBM Cloud

To log in to IBM Cloud, complete these steps:

1. Open the IBM Cloud console in your web browser:

   `http://bluemix.net`

2. Click **Log in**.

3. Enter your IBM Cloud ID (Figure 4-4).

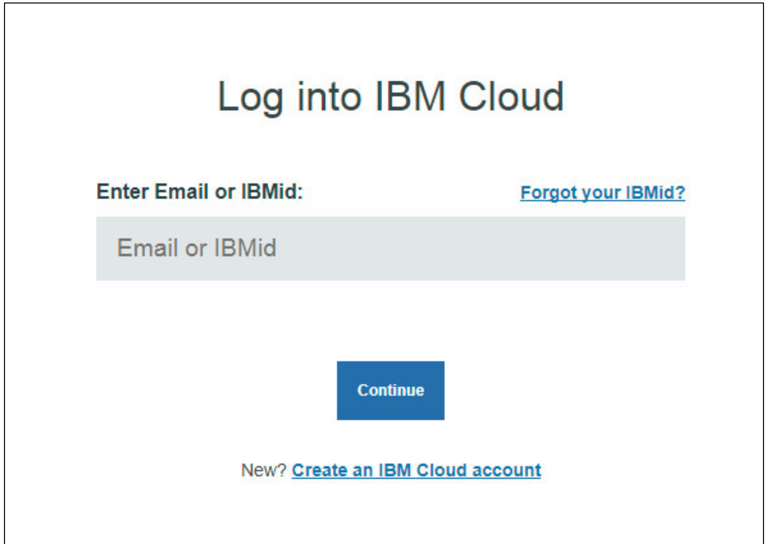


Figure 4-4 IBM Cloud login

4. Click **Continue**.
5. Enter your password and click **Log in**.
6. Confirm that your IBM Cloud account page opens.

4.3.2 Clone the Express application from Git by using the Delivery Pipeline

This application builds on the code that you developed in Chapter 3, “Creating your first Express application” on page 53. If you successfully finished the application in Chapter 3, “Creating your first Express application” on page 53, you can skip to “Create your first React page” on page 106.

Complete the following steps to quickly deploy the application:

1. Open the following link:

   https://console.bluemix.net/devops/setup/deploy?repository=https://github.com/ibm-redbooks-dev/vy102-XXX-express

2. On the page that opens (Figure 4-5), select the Delivery Pipeline tab. On the Delivery Pipeline tab, set the App name to `vy102-XXX-react` (replace `XXX` with your unique key). Keep all other settings unchanged and click **Deploy**.

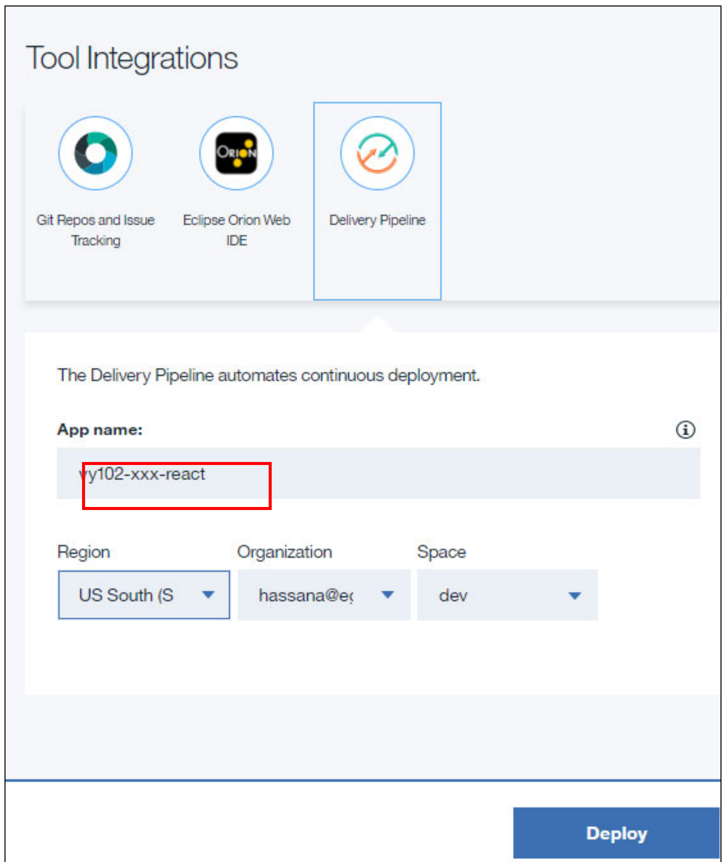


Figure 4-5 Create a Toolchain window

3. A new window opens (Figure 4-6). Click **Eclipse Orion Web IDE**.

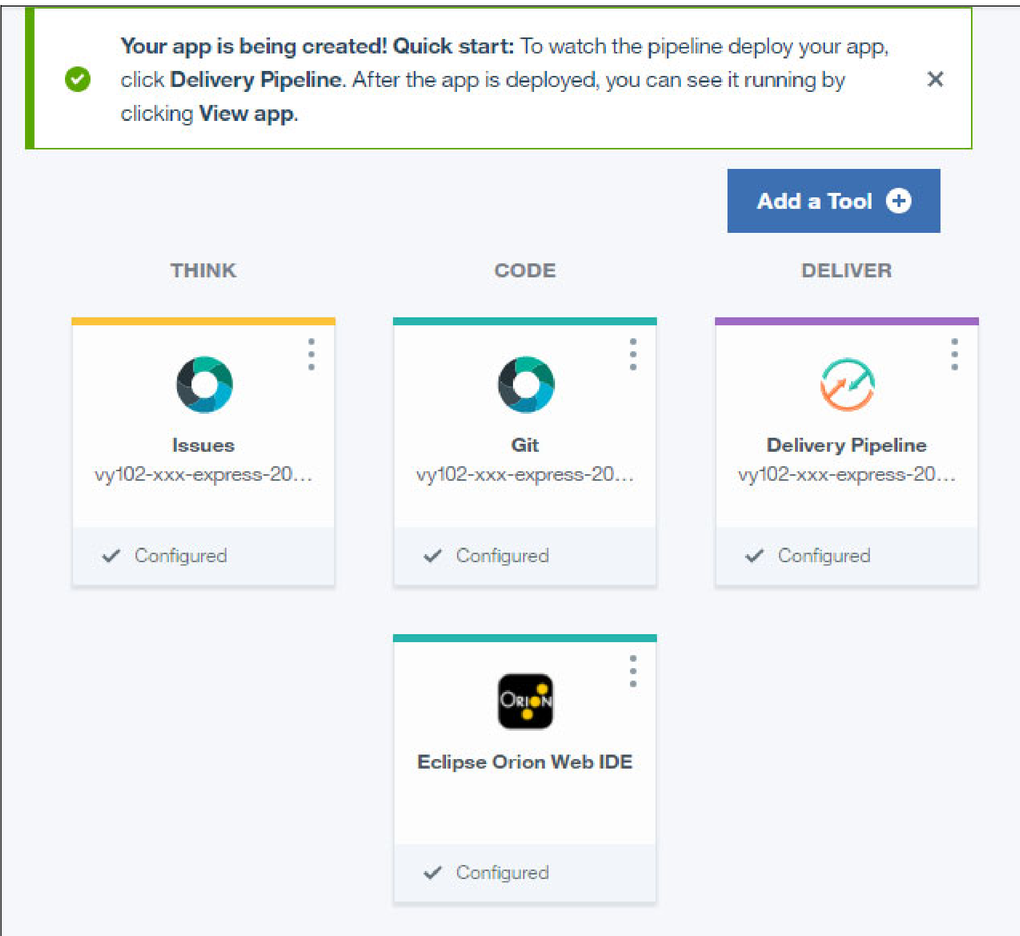


Figure 4-6 IBM Cloud toolchain: Select Eclipse Orion Web IDE

4. The Eclipse Orion Web IDE lists your project files (Figure 4-7). The source code from the application developed in Chapter 3, “Creating your first Express application” on page 53 is displayed in the online IDE because the new project was cloned from it.

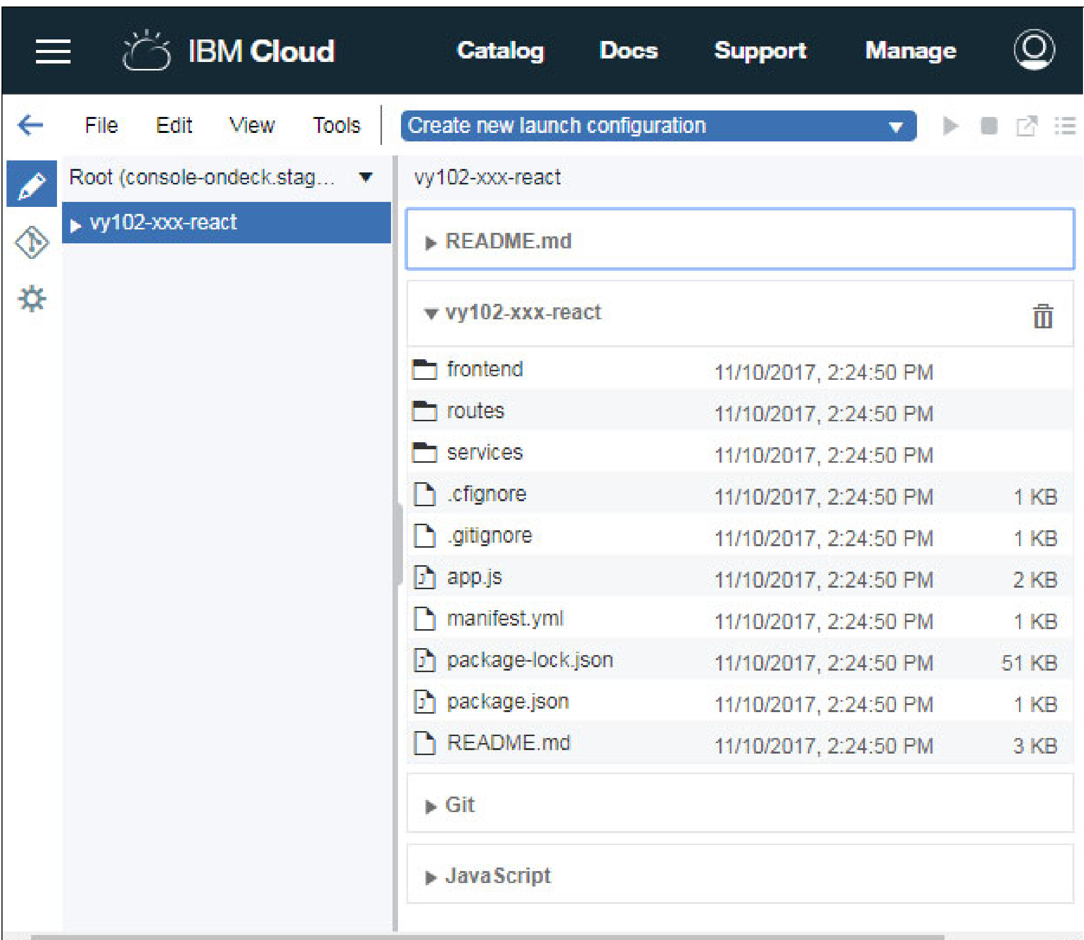


Figure 4-7 Eclipse Orion Web IDE

5. Make the following changes to the `manifest.yml` file (Figure 4-8 on page 104):
   - Change the application name and host on lines 10 and 11 to `vy102-xxx-react`. Replace xxx with your random key.
   - If needed, change the domain based on your IBM Cloud region as shown in Table 4-1.

The exercises in this book were tested in the US South region. If you use another IBM Cloud region and face any problem, register for a new IBM Cloud account and select **United States** for country or region as described in 1.3.1, "Set up your IBM Cloud account" on page 4.

Table 4-1 IBM Cloud regions and domains

Region	Domain
US South	mybluemix.net
United Kingdom	eu-gb.mybluemix.net
Sydney	syd.mybluemix.net
Germany	eu-de.mybluemix.net

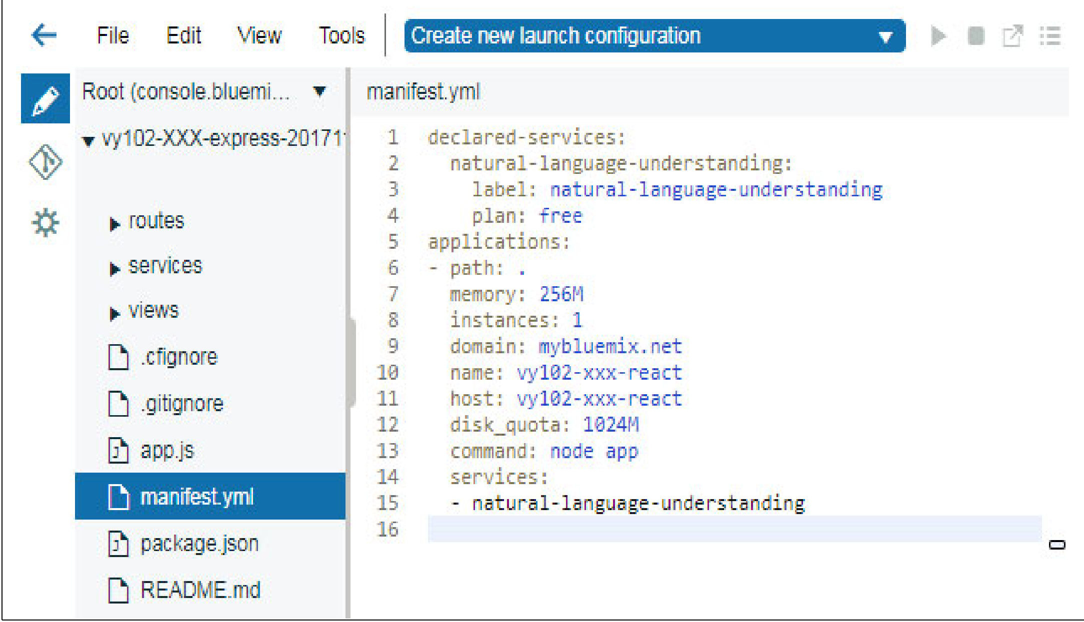


Figure 4-8 manifest.yml

6. Complete the following steps to configure the application to be deployed from Eclipse Orion IDE to IBM Cloud:
   a. Click **Create new launch configuration**, then press the **+** button (Figure 4-9).

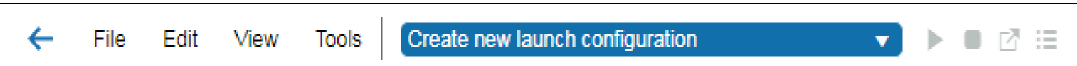


Figure 4-9 Eclipse Orion launch bar

b. In the Edit Launch Configuration window (Figure 4-10), keep all the settings as default and click **Save**. If your user is part of multiple organizations, choose the organization that has your email as its name. Note that application name and host values are defaulted from `manifest.yml`. If you change the host value, it overrides the value in `manifest.yml`.

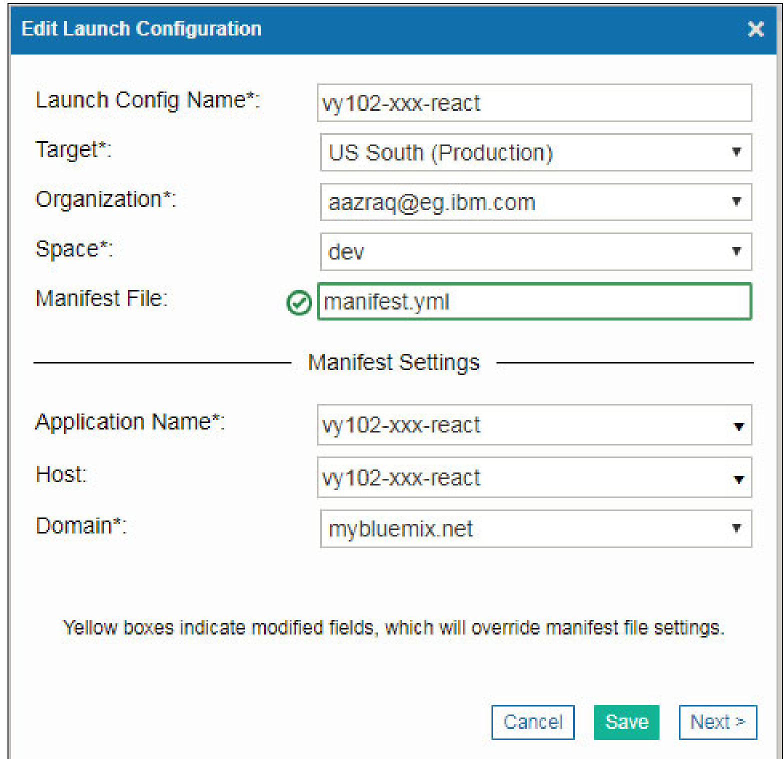


Figure 4-10 Edit Launch Configuration

c. Deploy the application by clicking the play icon, **Deploy the App from the Workspace**, which is highlighted in Figure 4-11.

Figure 4-11 Eclipse Orion launch bar

Troubleshooting: If you get the error `The app space binding to service is taken`, delete the deployed app from your IBM Cloud dashboard. Then redeploy the app from Eclipse Orion by clicking the play icon (**Deploy the App from the Workspace**).

7. Wait for the deployment to complete. To make sure that your application is running, either click the **Open the Deployed App** icon (highlighted in Figure 4-12) or go directly to your default route:

`https://vy102-xxx-react.mybluemix.net/`

Figure 4-12 Eclipse Orion launch bar

4.3.3 Create your first React page

React is a JavaScript-based front-end web application framework that is used primarily to develop single-page applications. The single-page application is a web page that loads one HTML page, and dynamically updates it as the user interacts with the app.

The entire React application, including HTML views, JavaScript files, and CSS files, run on the browser. In this exercise, to make these components available to the browser, all the client-side files must be located under a single root folder and must be served by the Express framework by using the `express.static()` method.

> **Important:** The application now uses two JavaScript frameworks: Express, which runs on the server side and React, which runs on the client side. From the perspective of Node.js, all the React code is a set of static text files that are passed to the browser without any processing. To avoid confusion between the client-side files and the server-side files, this exercise keeps all client-side files in a hierarchy inside the `frontend` folder.

Complete the following steps:

1. Start Eclipse Orion Web IDE for your application if it is not opened already.
2. Update the `app.js` file by adding the following lines after the initialization of the routes module:

```
//Serve the files in /frontend as static files
app.use(express.static(__dirname + '/frontend'));
```

The updated `app.js` file is shown in Figure 4-13.

```
1   var port = process.env.VCAP_APP_PORT || 8080;
2
3   //Express Web Framework, and create a new express server
4   var express = require('express'),
5       app = express();
6
7   var path = require('path');
8
9   var bodyParser = require('body-parser');
10  //parse application/x-www-form-urlencoded
11  app.use(bodyParser.urlencoded({
12      extended: false
13  }));
14
15  //Routes modules
16  var index = require('./routes'),
17      author = require('./routes/author');
18
19  //Serve the files in /frontend as static files
20  app.use(express.static(__dirname + '/frontend'));
21
22  //In case the caller access any resource under the root /, call index route
23  app.use('/', index);
24
25  //In case the caller access any resource under /author, call author route
26  app.use('/author', author);
27
28  // start server on the specified port and binding host
29  app.listen(port);
```

Figure 4-13 The added lines in the updated app.js file

The code in `app.js` initializes the Express framework and sets it to listen to all HTTP requests. It also sets Express to use the `/frontend` folder as its static content root. This setting means that when the user tries to access any URL in the application, Express tries to find the file that matches that path inside the `/frontend` folder.

If a file is found, it is returned to the user. If no static file matches the path that is requested by the user, Express tries to match the path to one of the routes. If no routes match, error 404 is returned.

> **Note:** Express tries to match static file paths first because the `express.static()` method is called in this code before the routes are registered. If the invocation order is reversed, Express checks the routes before looking up static files. In general, you should match static files before matching routes.

If the user does not specify a path, Express checks whether a file named `index.html` exists at the root of the `/frontend` folder. If that file exists, it is returned to users opening the domain without specifying a path. Essentially, the `/frontend/index.html` file is the home page, so the first order of business is to update that file to use React.

3. Create the folder `frontend` under the root of the project and create the file `index.html` inside it with the content shown in Example 4-16.

Example 4-16 Basic HTML page

```
<html>

<head>
    <title>React JS - Author Finder</title>
</head>

<body>

</body>

</html>
```

4. Reference React, and Babel in your HTML. This reference can be done by inserting the script lines shown in Example 4-17 inside the `<body>` tag.

Example 4-17 Fetching required libraries

```
   <!-- Loading the script in body is a recommendation to ensure faster loading,
        putting all scripts at the header will cause the page to wait till all scripts loaded
         Load React related files from internet -->
   <script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
   <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>

   <!-- Load babel JavaScript compiler from internet -->
   <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

5. Update the `/frontend/index.html` file to reference bootstrap CSS from the Content Delivery Network (CDN). This update can be done by inserting the `link` line shown in Example 4-18 inside the `<head>` tag right after the `</title>` tag.

> **CDN:** A CDN is a distributed system that hosts common CSS, JavaScript, and other web resources in a geographically dispersed group of servers so that the files are delivered to users from the server nearest to them.

Example 4-18 Fetching bootstrap stylesheets

```
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css"/>
```

6. Insert a simple HTML code that contains a `<div>` tag inside the `<body>` tag, right after the last `</script>` tag (Example 4-19).

   The React component is rendered in the `<div>` tag with `id="root"`.

   Notice the use of `col-sm-10`, `col-sm-offset-1`, and `text-center` bootstrap classes.

Example 4-19 <div> tag in HTML code

```
<div class="container">
   <div class="row">
      <!-- You do not need to write much html code, you will
            build the whole html on your JSPX code            -->
      <div class="col-sm-10 col-sm-offset-1 text-center" id="root">
      </div>
   </div>
</div>
```

7. With the `/frontend/index.html` file open in the IDE, press **Alt** + **Shift** + **F** to auto-format it. It should look like Example 4-20.

Example 4-20 Formatted /frontend/index.html file

```
<html>

<head>
   <title>React JS - Author Finder</title>
   <link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css" />
</head>

<body>
   <!-- Loading the script in body is a recommendation to ensure faster loading,
        putting all scripts at the header will cause the page to wait till all scripts
loaded
         Load React related files from internet -->
   <script crossorigin
src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
   <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>

   <!-- Load babel JavaScript compiler from internet -->
   <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

   <div class="container">
      <div class="row">
         <!-- You do not need to write much html code, you will
               build the whole html on your JSPX code            -->
         <div class="col-sm-10 col-sm-offset-1 text-center" id="root">
         </div>
      </div>
   </div>

</body>

</html>
```

8. You no longer need the `/views/index.html` file because you replaced it with the `/frontend/index.html` file. Delete the `/views/index.html` file and all references to it, including the `index.js` route and the code registering it in the `app.js` file. To do so, complete the following steps:

   a. Right-click the **views** folder and select **Delete** to delete the folder and the `index.html` file inside it.

   b. Right-click the **/routes/index.js** file and select **Delete** to delete it.

   > **Note:** Delete the `index.js` file only. Do *not* delete the `routes` folder.

   c. In `app.js`, locate and delete `index = require('./routes'),.` Do not delete `var` because it is still used to declare the `author`.

   d. In `app.js`, locate and delete the `app.use('/', index)` line, along with the comment line before it.

   Your app.js file should look like Figure 4-14.

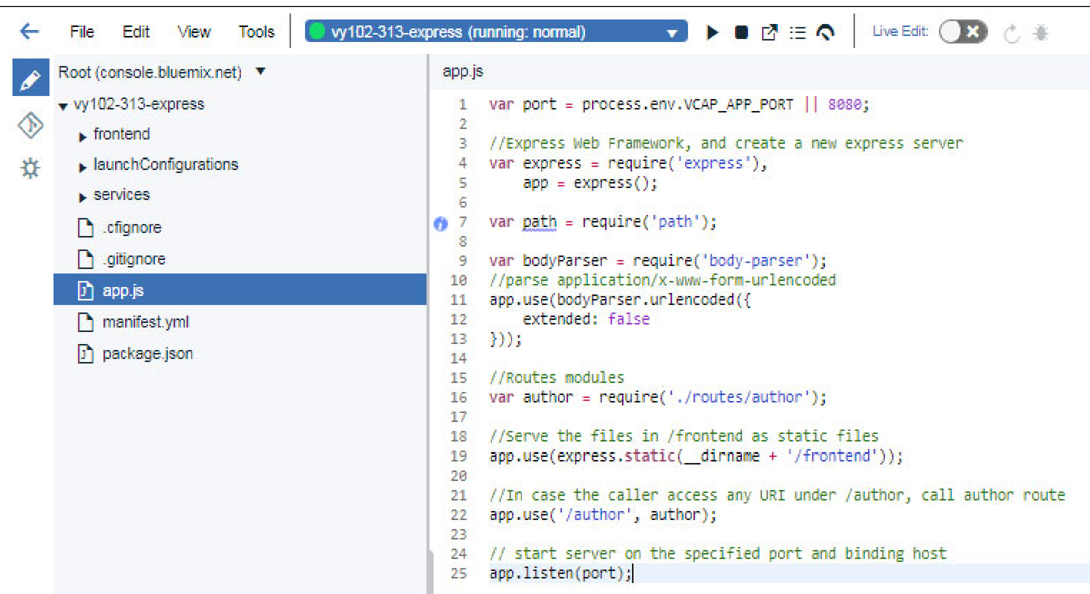


Figure 4-14 Updated app.js

4.3.4 Add a dynamic form to the page

Now, add a simple form to the page. The form consists of a text box, a label, and a button. The user enters the article URL in the text box and then clicks the button so that React calls the Node.js back-end service to extract the author's name.

For this step, create the form but do not call the Node.js service:

1. Under the `frontend` folder, create a folder called `components`. Create the `components.js` file in the `components` folder (Figure 4-15).

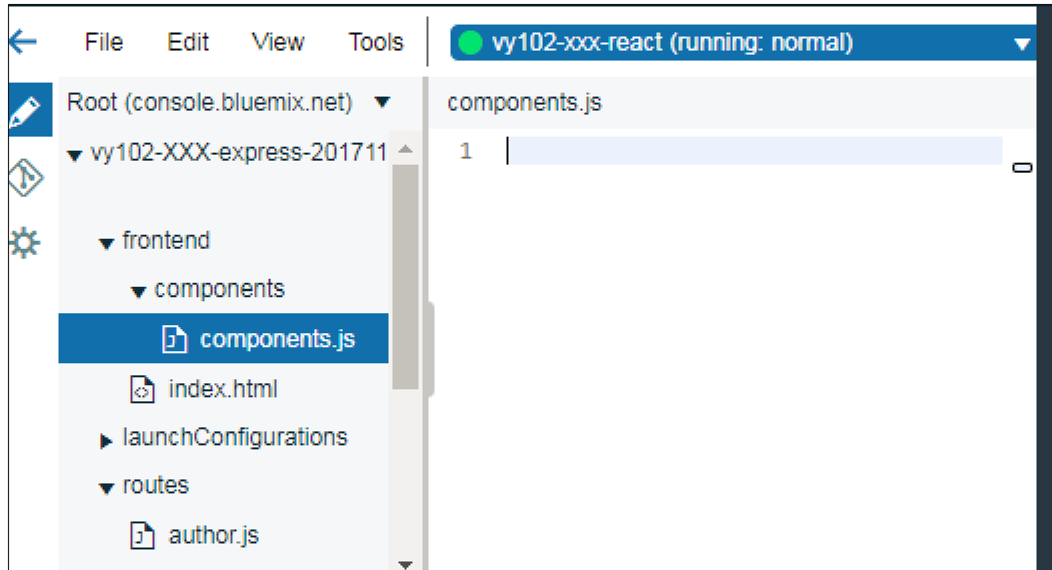


Figure 4-15 New components.js file

2. Inside the `components.js` file, create a class that extends `React.Component` as shown in Example 4-21.

Example 4-21 Code snippet: Class Container

```
class Container extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {}
}
```

3. Initialize the page state in the constructor by adding the code snippet from Example 4-22 to the constructor after the `super(props)` line.

Example 4-22 Code snippet: Class Container extends React.Component

```
this.state = {
            url: '',
            authors: [],
            inputVal: ''
};
```

4. After the constructor, add the `updateUrl()` method to update its state. Also, add the `getAuthor()` method with no implementation for now. It will be implemented later to call the back-end service to retrieve the authors (Example 4-23).

> **Note:** Format the code correctly after pasting, especially any wrapped around lines, and remove any unintentional line breaks in lines of code and comments.

Example 4-23 Adding the updateUrl() and getAuthor() methods

```
updateUrl(e) {
        this.setState({
            inputVal: e.target.value
        });
    }

    // getAuthor uses the fetch function to retrieve authors
    // from the REST service created in the previous exercise
    getAuthor() {}
```

5. Replace the render () method with the code snippet in Example 4-24. It contains html tags to render the author name and the button.

> **Note:** From this point on, do *not* use **Shift** + **Alt** + **F** to format the code in the `components.js file`. If you need to format the code, do it manually.
>
> Eclipse Orion Web IDE does not currently support JSX syntax formatting that is combining HTML and JavaScript code. You can use other code formatting tools such as *jsbeautifier* at `http://jsbeautifier.org/`. Select the **Support e4x/jsx syntax** option of *jsbeautifier* if you will use this tool to format JSX code in the examples of this chapter.

Example 4-24 Override the render() method

```
// Render is the core function behind React components.
// It defines components and elements in XML format.
// This is only feasible if using JSPX and Babel JavaScript compiler.
render() {
    return (
        <div class="jumbotron text-center">
                      <h1>Author Finder</h1>
                      <div id='input-form' class='text-center'>
                           <input type="text" class="form-control input-lg text-center"
onChange={e=>this.updateUrl(e)}  placeholder="Enter URL of Article here!"/>
                      </div>
                      <br/>
                      <button type="button" class="btn btn-primary btn-lg" disabled =
{this.state.inputVal.length===0} onClick={()=>{this.getAuthor()}}>Retrieve Author</button>
                  </div>
    )
}
```

6. Add the code snippet in Example 4-25 at the end, after the brace **}** closing the `Container` class. This code updates the HTML DOM tree, with the new component, starting from the `<div id="root">` element. The code inserts the Container component in the `<div id="root">` element shown in Example 4-20 on page 109.

Example 4-25 ReactDOM.render

```
ReactDOM.render(<Container />, document.getElementById("root"));
```

7. Your code now should look like Example 4-26.

Example 4-26 Complete code for Container class

```
class Container extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            url: '',
            authors: [],
            inputVal: ''
        };

    }

    updateUrl(e) {
        this.setState({
            inputVal: e.target.value
        });
    }

    // getAuthor uses the fetch function to retrieve authors
    // from the Rest service created in the previous exercise
    getAuthor() {}


    // Render is the core function behind React components.
    // It defines components and elements in XML format.
    // This is only feasible  if using JSPX and Babel JavaScript compiler.
    render() {
        return (
            <div class="jumbotron text-center">
                       <h1>Author Finder</h1>
                       <div id='input-form' class='text-center'>
                            <input type="text" class="form-control input-lg text-center"
onChange={e=>this.updateUrl(e)}  placeholder="Enter URL of Article here!"/>
                       </div>
                       <br/>
                       <button type="button" class="btn btn-primary btn-lg" disabled =
{this.state.inputVal.length===0} onClick={()=>{this.getAuthor()}}>Retrieve Author</button>
                   </div>
        )
    }
}

ReactDOM.render(<Container />, document.getElementById("root"));
```

8. Update `index.html` to import the React Container component by adding the following line in the `<body>` tag after the last `</script>` tag, as shown Figure 4-16.

```
<script type="text/babel" src="./components/components.js"></script>
```

Note that the script type is `text/babel`, not JavaScript, to indicate to the browser that it needs the support of the Babel compiler to translate this page.

```
1   <html>
2
3   <head>
4       <title>React JS - Author Finder</title>
5       <!-- load from the internet bootstrap style class that enhance the look of the the webpage -->
6       <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css" />
7   </head>
8
9   <body>
10      <!-- Loading the script in body is a recommendation to insure faster loading,
11           putting all scripts at the header will cause the page to wait till all scripts loaded
12           Load React related files from internet -->
13      <script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
14      <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
15
16      <!-- Load babel script pre-compiler from internet -->
17      <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
18
19      <script type="text/babel" src="./components/components.js"></script>
20
21      <div class="container">
22          <div class="row">
23              <!-- You do not need to write much html code, you will
24                    build the whole html on your JSPX code          -->
25              <div class="col-sm-10 col-sm-offset-1 text-center" id="root">
26              </div>
27          </div>
28      </div>
29
30  </body>
31
32  </html>
```

Figure 4-16 The /frontend/index.html file

9. Deploy the application by clicking the **Play** icon (**Deploy the App from the Workspace**). If you receive a pop-up message warning you that the application will be redeployed, click **OK** to confirm.

10. After the application deployment is complete, run the application by clicking the **Open the Deployed App** icon. The UI should look like Figure 4-17. Notice that the **Retrieve Author** button is disabled until the user starts writing in the text box.

Figure 4-17 Dynamic React form

4.3.5 Add more components to the form

Figure 4-18 represents the React components to be developed in this exercise.

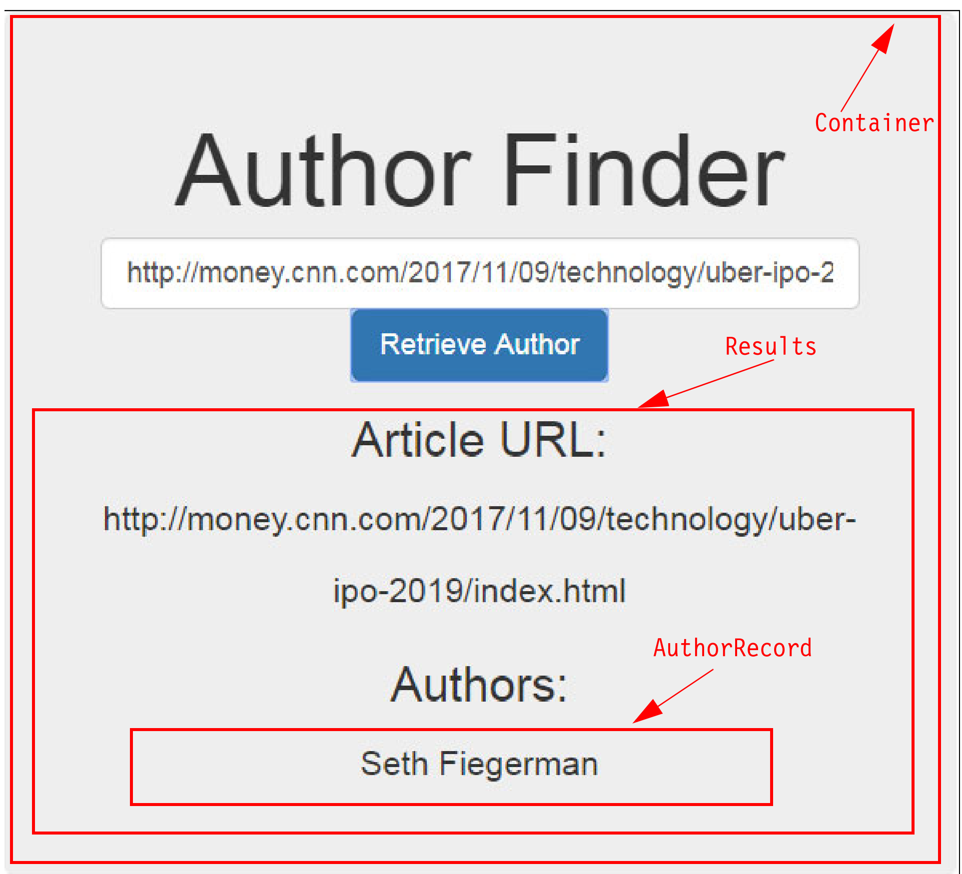


Figure 4-18 Component structure

In 4.3.4, "Add a dynamic form to the page" on page 110, you added an HTML form to the page as a React component called *Container*.

In this section, you will add the Results and the AuthorRecord components using these steps:

1. Open the `/frontend/components/components.js` file.
2. Copy the code snippet from Example 4-27 on page 116 into the `components.js` file, before `ReactDOM.render(<Container />, document.getElementById("root")).`

> **Note:** Do **not** use **SHIFT** + **ALT** + **F** to format the code in the `components.js` file. If you need to format the code, do it manually.

Example 4-27 shows a new component called Results. It displays the Article URL, and the list of extracted author names.

Example 4-27 Code snippet: Results Component in components.js

```
// Results: is another React component
// that creates the list of AuthorRecords
class Results extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {

        if (this.props.hide) {
            return null;
        }

        return (
            <div class='form-inline'>
                   <div class="row">
                        <div class="col-xs-12 col-md-3">
                            <h2>Article URL:</h2>
                        </div>
                        <div class="col-xs-12 col-md-9">
                            {this.props.url}
                        </div>
                    </div>
                    <div class="row">
                            <div class="col-xs-12 col-md-3">
                                <h2>Authors:</h2>
                            </div>
                            <div class="col-xs-12 col-md-9">
                                <div class="row">

                                </div>
                            </div>
                    </div>
            </div>
        )
    }
}
```

Note: If you see errors in the `components.js` file, ignore them. These errors are caused by combining HTML code with JavaScript code as this file follows JSX syntax.

3. Copy the code snippet from Example 4-28 into the `components.js` file after the `render()` method in the Results class.

   Example 4-28 shows a helper method called `renderAuthors()` to iterate over the authors retrieved from the back-end service by using `map()`.

   Notice that the AuthorRecord is not defined in the file. You are going to define it in another file in the next section. This is one of the powerful features of React Component because putting the AuthorRecord in a separate file makes it a reusable component.

Example 4-28 Code snippet: renderAuthors()

```
   /*
  Notice that the AuthorRecord is defined in a separate file. This is a
powerful feature of the React Component: Putting the AuthorRecord in a separate
file and making it a reusable component.
  */
renderAuthors() {
    /*
    When developing a component, you should capitalize it.
    Hence, you should use "AuthorRecord" instead of "authorrecord" to identify
    it as a component to React.
    */
    let authors = this.props.authors;
    return authors.map(a => {
        return <AuthorRecord author = {a}/>;
    });
}
```

4. Call the renderAuthors() method, within the div class=”row”. It is highlighted in **bold** in Example 4-29. Your final Results component should look like Example 4-29.

Example 4-29 Code snippet: Results component

```
// Results: is another React component
// It creates the list of AuthorRecords
class Results extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {

        if (this.props.hide) {
            return null;
        }

        return (
            <div class='form-inline'>
                   <div class="row">
                        <div class="col-xs-12 col-md-3">
                            <h2>Article URL:</h2>
                        </div>
                        <div class="col-xs-12 col-md-9">
                            {this.props.url}
                        </div>
                    </div>
                    <div class="row">
                            <div class="col-xs-12 col-md-3">
                                <h2>Authors:</h2>
                            </div>
                            <div class="col-xs-12 col-md-9">
                                <div class="row">
                                     {this.renderAuthors()}
                                </div>
                            </div>
                    </div>
            </div>
        )
    }

    /*
      Notice the AuthorRecord is defined in a separate file. This is a powerful feature of the React
Component: Putting the AuthorRecord in a separate file and making it a reusable component.
      */
    renderAuthors() {
        /* When developing a component, you should capitalize it.
        Hence, you should use "AuthorRecord" instead of "authorrecord" in order to identify it as a
component to React.
        */
        let authors = this.props.authors;
        return authors.map(a => {
            return <AuthorRecord author = {a}/>;
        });
    }

}
```

Note: The value of `url` and `authors` is retrieved from the state. Therefore, any event that updates `state.url` and `state.authors` causes the components to be rendered.

5. In the `Container` class below the <button> tag, add the `Results` element to render the Results component:

```
<Results url={this.state.url} hide={this.state.authors.length === 0}
authors={this.state.authors}/>
```

   This line includes the Results component element and these properties:

   – `url` retrieved from `state.url`
   – `authors` retrieved from `state.authors`.

   Figure 4-19 shows the Results component added to the Container class.

```
// Container: considr it the main component the descripe the whole page
class Container extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            url: "",
            authors : [],
            inputVal: ""
        }
    }

    updateUrl(e){
        this.setState({inputVal:e.target.value});
    }

    // getAuthor use fetch function, in order to retrieve authors for rest service we created in previous excersie
    getAuthor() {
    }

    // Render is the core function behind React components, define components and elements in XML format, only feasible
    render() {
        return (
            <div class="jumbotron text-center">
                <h1>Author Finder</h1>
                <div id='input-form' class='text-center'>
                    <input type="text" class="form-control input-lg text-center" onChange={e=>this.updateUrl(e)}  p
                </div>
                <br/>
                <button type="button" class="btn btn-primary btn-lg" disabled = {this.state.inputVal.length===0} onC
                <Results url={this.state.url} hide={this.state.authors.length === 0} authors={this.state.authors}/>

            </div>
          )
    }
}
```

Figure 4-19 Adding Results to the Container class

6. Create a file called `authorrecords.js` and place it in the `components` folder. `authorrecords.js` defines a new React component, but in a separate file.

7. Add the code snippet in Example 4-30 to the newly created file `authorrecords.js`.

> **Note:** Do *not* use **SHIFT** + **ALT** + **F** to format the code in the `authorrecords.js` file. If you need to format the code, do it manually.

Example 4-30 Code snippet: authorrecords.js

```
//AuthorRecord : a component defined to hold author names
class AuthorRecord extends React.Component{

    render() {
        return (
            <div class="row">
                 <div class="col-xs-12 col-md-6">
                    {this.props.author.name}
                </div>
            </div>
        );
    }
}
```

The code in Example 4-30 shows that you can define a component in a separate file, and use it in your web page. This approach allows you to reuse the component in another project by simply copying that file to your new project.

Figure 4-20 shows a snapshot of `component.js` rendering `AuthorRecord`.

```
components.js
53  // Results: is another React component that create hold the the list of AuthRecords
54  class Results extends React.Component {
55      constructor(props) {
56          super(props);
57      }
58
59      // Notice the AuthRecod not defined in that file, it is defined in separate file
60      // This is one of the powerful feautre of React Component, putting the AuthRecod in seperate file and make it
61      renderAuthors() {
62          let authors = this.props.authors;
63          // when developing compoenet you should make it capatilized "AuthRecord" not "authrecord" in order to make
64          return authors.map(a => {
65              return <AuthorRecord author= {a}/>;
66          });
67      }
```

Figure 4-20 Rendering AuthorRecord in component.js

8. To make the new component `AuthorRecord` accessible to the other component, import `authorrecords.js` in `index.html`. In `index.html`, add the following line before importing `components.js`:

```
<script type="text/babel" src="./components/authorrecords.js"></script>
```

Your `index.html` file should look like Figure 4-21.

```
<html>

<head>
    <title>React JS - Author Finder</title>
    <!-- load from the internet bootstrap style class that enhance the look of the the webpage
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.cs
</head>

<body>
    <!-- Loading the script in body is a recommendation to insure faster loading,
         putting all scripts at the header will cause the page to wait till all scripts loaded
         Load React related files from internet -->
    <script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></

    <!-- Load babel script pre-compiler from internet -->
    <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

    <script type="text/babel" src="./components/authorrecords.js"></script>
    <script type="text/babel" src="./components/components.js"></script>

    <div class="container">
        <div class="row">
            <!-- You do not need to write much html code, you will
                 build the whole html on your JSPX code          -->
            <div class="col-sm-10 col-sm-offset-1 text-center" id="root">
            </div>
        </div>
    </div>

</body>

</html>
```

Figure 4-21 The /frontend/index.html file

4.3.6 Using the Fetch API to call the Node.js author service

React is a framework to implement the view components in the model view controller (MVC) architecture. To retrieve resources from the back-end server, you need to use another library. `XMLHttpRequest` is commonly used to call REST services, but in this example you will use the Fetch API. It is a new API that provides a flexible feature set. It returns a promise that resolves the Response, which makes calling REST services simpler.

> You can use the Fetch API to call any REST API. All Watson APIs are exposed as REST services, which means that you can call the Watson Language Translator service, the Watson Natural Language Understanding services, or any other Watson service by using the Fetch API.
>
> In this part, you will call the `author` service that you developed in Chapter 3, “Creating your first Express application” on page 53.

To call the author service that was developed in Chapter 3, “Creating your first Express application” on page 53, complete these steps:

1. Open `/frontend/components/components.js` to update the `getAuthor()` method.
2. In the Container class, in `components.js`, replace `getAuthor()` with the code snippet in Example 4-31.

Example 4-31 Code snippet: /frontend/components/components.js file

```
getAuthor() {
   var myHeaders = new Headers();
   myHeaders.append("Content-Type", "application/x-www-form-urlencoded");
   fetch('/author', {
         method: 'POST',
         body: "url=" + this.state.inputVal,
         headers: myHeaders
      }).then(res => res.json())
      .then(data => this.setState({
         authors: data,
         url: this.state.inputVal
      }));
}
```

   This code sends an asynchronous POST request to `/author`. In Chapter 3, “Creating your first Express application” on page 53, you created an Express route that handles such requests.

   In Example 4-31 there are multiple `then` methods, which is called *promises chaining*. Because `then` returns a promise, it means that promises can be chained. This method is used to avoid nested callbacks.

   The code snippet in Example 4-31 defines the HTTP method as POST and sets the path to `/author`. It also passes the URL that is entered by the user in the `url` field in the body, and sets the content type header to `application/x-www-form-urlencoded`.

3. Deploy the application using the **Play** icon (**Deploy App from the Workspace**) from the server toolbar. If a window with `Stop and redeploy? Your application vy102-XXX-express will be redeployed` is displayed, click **OK**.
4. After the deployment is complete, start the application by clicking **Open the Deployed App** from the server toolbar.
5. To test the application, enter the following web address into the **URL** text box:

   `http://edition.cnn.com/2017/06/12/politics/hfr-dennis-rodman-north-korea/index.html`

The application now displays the complete list of authors (Figure 4-22).

Figure 4-22 Displaying the list of authors

6. Try resizing the window to be narrower (or open the application in your mobile device). Notice how the page *responds* by adapting its layout. The "Article URL" and "Authors" headings now appear above their respective results, as shown in Figure 4-23. This approach is called responsive design, and is made possible by using Bootstrap.

Figure 4-23 Mobile view

4.4 Exercise review

During this exercise, you achieved the following goals:

- Learned the basics of the React framework.

  You built a basic React application that uses React components to interact with the user. It uses the Fetch API to communicate with the server.

- Learned the basics of ES6.

  You used the class syntax, arrow functions, and promises with the Fetch API to call the back-end services

- Learned the basics of the Bootstrap framework.

  You used the Bootstrap framework responsive grid system to create a responsive page that has different layouts on various screen sizes.

- Learned how to use Express to serve static files and resources.

  You used the Express framework's `express.static` service to serve all the files that are hosted in the `/frontend` folder.

- Used Git to clone an existing project.

  You used Git to clone the source code from Chapter 3, "Creating your first Express application" on page 53 and used it as the basis for this exercise.

Additional material

This book refers to additional material that can be downloaded from the Internet.

Locating the material on GitHub

The source code that is associated with Chapter 3, “Creating your first Express application” on page 53 is available at the following GitHub location:

https://github.com/ibm-redbooks-dev/vy102-XXX-express

© Copyright IBM Corp. 2017. All rights reserved.

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 publications provide additional information about the topic in this document:

- *Essentials of Application Development on IBM Cloud*, SG24-8374

You can search for, view, download or order this documents and other IBM Redbooks, IBM Redpapers, Web Docs, draft and additional materials, at the following website:

http://www.redbooks.ibm.com/

Online resources

These websites are also relevant as further information sources:

- IBM Cloud console to sign up for an account

  https://bluemix.net
- Node.js

  https://nodejs.org/en/
- Express

  https://expressjs.com/
- React

  https://reactjs.org
- ECMAScript 2015 (ES6) Language Specification

  https://www.ecma-international.org/ecma-262/6.0/

Help from IBM

IBM Support and downloads

https://www.ibm.com/support/home/

IBM Global Services

https://www.ibm.com/services/index.html

© Copyright IBM Corp. 2017. All rights reserved.

Developing Node.js Applications on IBM Cloud

(0.2"spine)
0.17"<->0.473"
90<->249 pages

SG24-8406-01

ISBN 0738442852

Printed in U.S.A.

Get connected