WebSphere. software

IBM

IBM WebSphere Telecommunications Web Services Server Programming Guide

- Architecture and component overview
- Design considerations and best practices
- Creating a custom service implementation

John Bergland
Rajasekhar Durvasula
Jochen Kappel
Hailong Luo
Bala Sivasubramanian

Redbooks

ibm.com/redbooks

Archived

Archived

International Technical Support Organization

IBM WebSphere Telecommunications Web Services Server Programming Guide

September 2008

Archived

SG24-7589-00

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

Archived

First Edition (September 2008)

This edition applies to IBM WebSphere Telecommunications Web Services Server Programming Guide Version 6.2.

© Copyright International Business Machines Corporation 2008. 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

© Copyright IBM Corp. 2008. All rights reserved.

Archived

Archived

Notices

This information was developed for products and services offered in the U.S.A.

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 give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: 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 states 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 Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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

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.

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 the names and addresses used by an actual business enterprise 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.

© Copyright IBM Corp. 2008. All rights reserved.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
Alerts®
DB2®
Domino®
IBM®
IMS™
Lotus®
NetView®
OS/2®
Rational®
Redbooks®
Redbooks (logo) ®
Tivoli®
WebSphere®
Workplace™

The following terms are trademarks of other companies:

Direct Connect, the AMD Arrow logo, and combinations thereof, are trademarks of Advanced Micro Devices, Inc.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

EJB, Enterprise JavaBeans, J2EE, Java, JavaBeans, JavaServer, JDBC, JDK, JMX, JSP, JVM, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

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

Archived

Preface

This IBM® Redbooks® publication is a programming guide that provides developers with the information they need to create new Web service implementations for IBM WebSphere® Telecommunications Web Services Server.

IBM WebSphere Telecommunications Web Services Server allows you to expose high-level Web service interfaces to network services for third parties.

Third parties are typically external service providers, customers, or organizational divisions that want to develop new services that integrate with the service provider network infrastructure. Web service interfaces provide access to service capabilities in a programming language and technology independent way. Each Web service interface can have multiple back-end implementations for connecting with a service provider's environment. For example, a Web service interface may connect to a service provider's network through the Session Initiation Protocol (SIP), using a Parlay Connector through a Parlay Gateway, through native service provider protocols, or using custom integrated services.

In this IBM Redbooks publication, we provide specific references, best practices, guidance, and implementation examples for programming IBM WebSphere Telecommunications Web Services Server components and customize it for your organization's particular needs. More specifically, we discuss the following items within the context of a common example scenario:

- Working with the Service Policy Manager and creating a custom policy
- Working with the Access Gateway to make modifications to a default mediation flow, create a custom mediation primitive, or create a completely new mediation flow from scratch
- Creating a custom Parlay X Service Implementation, which is based on creating the Publish ( ) operation within a Presence Supplier interface.

The team that wrote this book

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

Rajasekhar Durvasula is an Advisory IT Architect with BPTSE Team, based at IBM India Software Labs. Raj carries over 10 years of rich experience in application development, product development, and delivery services. Raj is a team member of Industry Business Partner Technical Strategy & Enablement, focusing on the Asia Pacific region. Raj leads the design and deployment of large scale Enterprise Application Integration projects in telecom vertical. His interest areas are IMS™, SDP, SOA, Business Integration, and Project Methodologies. He works with IBM IMS Service Plane, USCE & SDP, as well as IBM SOA Foundation products.

© Copyright IBM Corp. 2008. All rights reserved.

Jochen Kappel is a Senior IT Architect in the IBM Telecom Solutions Lab in Europe. He has a 22-year experience in building solutions for wireless and wireline telecom operators, focusing on service creation, next generation networks, and service delivery platforms. Jochen's areas of expertise include end-to-end systems integration, application architecture and development, and technical team leadership. He is an IBM certified Telecommunications IT Architect. and holds a Diploma in Electrical Engineering and Telecommunications from the Technical University Darmstadt, Germany.

Hailong Luo is an Advisory IT Specialist and Field Technical Sales Specialist (FTSS) in IBM China. He currently serves in a lead role in supporting Huawei with their telecom applications. Hailong has been the team lead for strategic POC efforts for Huawei, using his skills on both IBM WebSphere Telecommunications Web Services Server and WebSphere Process Server. Hailong is one of the earliest members of Open Partnership Center (OPC), acting as a WebSphere team leader and establishing a strong team to support Business Partners. In addition to his in-depth knowledge of IBM WebSphere Telecommunications Web Services Server and WebSphere Process Server, he has experience in SOMA, SOA, and system architecture.

Bala Sivasubramanian is an IBM Accredited Senior IT Specialist for Business Applications from IBM GBS US Wireless Data Services and communication practice. He is a Board Member of Advisory & Senior Accreditation for IT Specialist. He has more than eight years of experience in the IT Industry and has specialized in telco industry for more than five years. His expertise includes AIX®, Linux®, WebSphere products, and programming in Java™, J2EE™, Web Services, developing BPEL, and XML. He focuses primarily on designing, developing, and integrating solutions that use Java, J2EE, Web Services, BPEL, and JMS. He is also a Sun™ Certified Java Programmer and SEI PSP Level 5 Engineer from SEI, Carnegie Melon University, Pittsburgh. Additionally, he holds a degree in Computer Engineering from Madurai Kamaraj University, TamilNadu, India.

John Bergland is a project leader at the ITSO, Cambridge Center. He manages projects that produce IBM Redbooks that focus on IBM WebSphere technology. Before joining the ITSO in 2003, John worked as an Advisory IT Specialist with IBM Software Services for Lotus® (ISSL), specializing in Notes and Domino® messaging and collaborative solutions.

Co-authors of foundation material for this IBM Redbooks publication

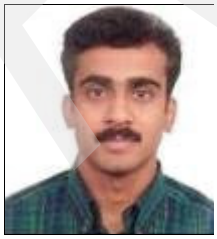
This IBM Redbooks publication is based upon much of the work from the following people, who produced both the *IBM WebSphere Telecommunications Web Services Server 6.2 Service Platform System Level Design Document*, and the *IBM WebSphere Telecommunications Web Services Server Programming Guide Version 6.2.*

Scott Broussard is a Senior Software Engineer and IBM Master Inventor with 15 plateau works in SWG AIM WebSphere Service Provider Development. He has worked on the IBM WebSphere Telecommunications Web Services Server and related software for the past nine years. He has worked on the Java JVM™ team with a focus on the windowing features of the Java runtime across several IBM platforms. He has also worked in the OS/2® Multimedia project in the area of user interfaces involving the Workplace™ Shell and OpenDoc. Scott holds a BS degree in Computer Science from the University of Missouri at Rolla.

Michael Gilfix works in architecture and strategy for IBM software group's WebSphere Business Process Management (BPM) and connectivity product portfolio, helping customers to build, automate, and derive insight from their business applications and processes. Prior to that, he worked in IBM software group's telecom industry solutions. He was the lead architect for the IBM WebSphere Telecommunications Web Services Server product from its inception through several product releases. Before that, he was an architect on the IBM SIP stack, which has been integrated and shipped with WebSphere Application Server as of Version 6.1.

Tim Tedford is an Advisory Software Engineer at the IBM Development Lab in Austin, Texas. He has over 20 years of experience, including 10 years with IBM, in software development, support, and system integration in the financial and telecom areas. His current area of expertise is in the development of Service Oriented Architecture (SOA) and WebSphere Enterprise Service Bus (WESB) based implementations, Web services design and development, and advanced Java and J2EE technologies. He is currently responsible for the development and support of the Access Gateway component of the IBM WebSphere Telecommunications Web Services Server product, which contains a set of mediation primitive plug-ins that extends the Mediation Flow Editor of the IBM WebSphere Integration Developer product and a set of mediation module flows that provides support policy-driven traffic monitoring, capture, authorization, and management capabilities for the Parlay X V2.1 Service Implementations provided with IBM WebSphere Telecommunications Web Services Server. He holds a bachelor's degree in Computer Science from Worcester State College.

Velma Pavlasek is an Advisory Programmer and member of the IBM WebSphere Telecommunications Web Services Server Web Services Server team. She has been a member of the IBM WebSphere Telecommunications Web Services Server team for three years, and has been at IBM for thirty years. During the IBM Redbooks publication residency, Velma helped the team to achieve a deeper understanding of the Common Components.



Dhandapani Shanmugam is a Staff Software Engineer within the IMS IBM WebSphere Telecommunications Web Services Server Development team. His education includes an MS in Software Systems & BE - ECE. Dhandapani has eight years of IT experience, of which more than four years have been with IBM-ISL. He has been working on IBM WebSphere Telecommunications Web Services Server for the past three releases. Currently, he is part of the IBM WebSphere Telecommunications Web Services Server Development team in ISL Bangalore.

Special acknowledgement to the following people for their contributions:

Thank you to the following people for their sponsorship and support of this effort, together with technical guidance.

- **Larry Irvin** - Project sponsor. IBM Software Group, Application and Integration Middleware Software, BLM, SWG Communication Sector Solutions, IBM, Austin, TX
- **Cristi Nesbitt Ullmann**- IMS Chief Programmer, IBM Software Group, Application and Integration Middleware Software, IBM, Austin, TX.
- **Catherine Camillone** - IP Multimedia Subsystem Development Team Lead, IBM Software Group, Application and Integration Middleware Software, IBM, Austin, TX
- **Glen Bartels** - Program Director, WebSphere Service Provider Development. IBM Software Group, Application and Integration Middleware Software.
- **Anthony Wrobel** - Lead Architect, IMS Applications, IBM Software Group, Application and Integration Middleware
- **Jeffrey Martin** - Technical Enablement Specialist: AIM.SOA Implementation, IBM Software Group, Application and Integration Middleware Software
- **Jon Etkins** - IT Support Specialist, Austin ITSO, IBM Sales & Distribution, ibm.com

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

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 in one of the following ways:

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

  ibm.com/redbooks
- Send your comments in an e-mail to:

  redbooks@us.ibm.com

Archived

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Archived

Archived

1

Introduction to the IBM WebSphere Telecommunications Web Services Server

IBM WebSphere Telecommunications Web Services Server allows you to expose high-level Web service interfaces to network services for third parties.

Third parties are typically external service providers, customers, or organizational divisions that want to develop new services that integrate with the service provider network infrastructure. Web service interfaces provide access to service capabilities in a programming language and technology independent way. Each Web service interface can have multiple back-end implementations for connecting with a service provider's environment. For example, a Web service interface may connect to a service provider's network through the Session Initiation Protocol (SIP), using a Parlay Connector through a Parlay Gateway, through native service provider protocols, or using custom integrated services.

In this IBM Redbooks publication, we provide specific references, best practices guidance, and implementation examples for programming IBM WebSphere Telecommunications Web Services Server components to customize it for your organization's particular needs. More specifically, it discusses the following within the context of a common use case:

- Working with the Service Policy Manager and creating a custom policy
- Working with the Access Gateway to make modifications to a default mediation flow, create a custom mediation primitive, or create a completely new mediation flow from scratch
- Creating a custom Parlay X Service Implementation, which is based on creating the Publish () operation within a Presence Supplier interface

This chapter provides an overview of the IBM WebSphere Telecommunications Web Services Server and describes how it can provide business value. Additionally, this chapter introduces the common use case we refer to throughout this IBM Redbooks publication.

© Copyright IBM Corp. 2008. All rights reserved.

1.1 Overview of IBM WebSphere Telecommunications Web Services Server

IBM WebSphere Telecommunications Web Services Server enables service providers to provide third parties secure, reliable, and policy driven access to telecom network capabilities. Third-party application developers can enhance consumer and enterprise applications by utilizing valuable service provider network services, such as messaging, location, presence, and call handling through open standards-based Web Services. IBM WebSphere Telecommunications Web Services Server's access gateway function provides a common control point for service providers to define, manage, and enforce policies and Service Level Agreements (SLA) for third-party services and subscribers. The service provider controls which applications and which users have access to which network services and under what conditions.

Figure 1-1 illustrates the emphasis on service exposure, enabling *secure, policy-driven* delivery of telecom network capabilities through industry-standard Web services.

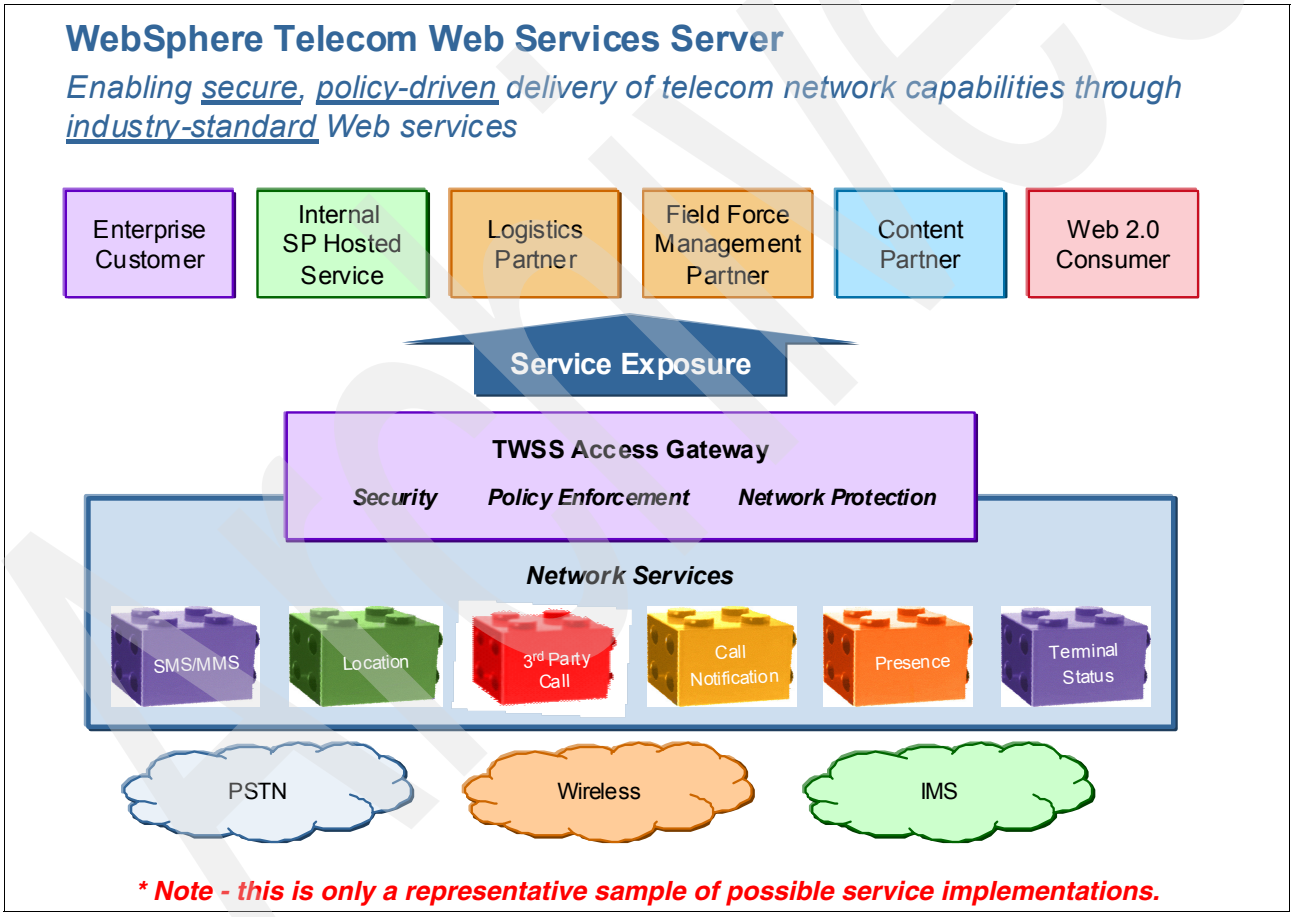


Figure 1-1 Enabling secure, policy-driven delivery of telecom network capabilities through Web services

1.2 Business value and positioning

IBM WebSphere Telecommunications Web Services Server delivers value by allowing service providers to expose services in a secure. reliable, and policy controlled way. By providing capabilities to expose network services, this allows for the service providers to potentially host new revenue generating services, and to capitalize on innovative third-party services.

1.2.1 Revenue generating services

The changing telecommunications market is driving telecom service providers to look for new revenue generating services. Their business is changing from a relatively few hosted services, to hundreds, if not thousands, of services. Most of these services are provided by third-party partners and service providers. New services will be rapidly developed from easy to use fragments of configurable service functions to create highly personalized services:

- *Offer new revenue generating services and business models* by providing third parties with industry standard defined Web services access to network service capabilities.
- *Retain high value customers and third-party service providers* with flexible policy driven service level agreements at the service, customer set, or individual subscriber level.
- *Protect underlying network resources* from unauthorized access and overload with secure, policy-based access, and effective traffic management capabilities.

1.2.2 Innovative third-party services

IBM WebSphere Telecommunications Web Services Server abstracts core network functionality, opening it to the large community of IT application developers and providing the following benefits:

- *Isolation from telecom network complexity*: The Telecom Web Services Service-Orientated Architecture enables third-party developers to focus their creativity on business function and service creation, rather than low-level implementation.
- *Simplification of skill requirements*: Network services are abstracted as Web services, enabling a large audience of Web developers to access service functions without requiring detailed infrastructure knowledge.
- *Protection from evolving network technologies*: Service provider networks are experiencing rapid change. While telecom Web service implementations may require changes to address network evolution, applications using these Web Services remain the same.

1.3 Functional component description

This section describes the primary components that make up the IBM WebSphere Telecommunications Web Services Server and the functions that they perform. Subsequent chapters in the IBM Redbooks publication provide a more in-depth examination of each component.

The IBM WebSphere Telecommunications Web Services Server is the foundation for managing third-party Web service access and provides the platform for telecom Web service implementations. The IBM WebSphere Telecommunications Web Services Server is comprised of two major components:

- The front-end Telecom Web Services access gateway
- The back-end service implementations

1.3.1 Access gateway - providing secure, policy-driven, and third-party access

The access gateway provides policy-driven traffic monitoring, capture, authorization, and management capabilities that are enforced for each Web service request using knowledge of the requestor, target service, and invoked operation.

The access gateway is built upon the WebSphere Enterprise Service Bus (ESB) product. This provides flexibility for constructing tailored Web service messaging processing in accordance with the service provider's network policies. The access gateway provides pluggable ESB components, each designed to serve a specific purpose. Specific details of each of the ESB components, the mediation primitives, is provided in Chapter 4, “Design considerations for Access Gateway flows - base mediation flows and the mediation primitives” on page 67.

1.3.2 Web service implementations based on Parlay X V2.1 Web Services standards

IBM WebSphere Telecommunications Web Services Server Service provides Web service implementations based on the Parlay X V2.1 Web Services standards for the Web service interface abstraction IMS network functions, including:

- Parlay X V2.1 Third Party Call, supporting the WebSphere V6.1 SIP application server call management function
- Parlay X V2.1 Call Notification, supporting the WebSphere V6.1 SIP application server call management function
- Parlay X V2.1 Presence for WebSphere Presence Server
- Parlay X V2.1 Terminal Status for WebSphere Presence Server

To facilitate the rapid development of new service implementations, reusable components are provided that can be shared by all service implementations. The common services are utilized by the Parlay X V2.1 Web service implementations provided in the IBM WebSphere Telecommunications Web Services Server and are available for use in developing additional implementations through IBM services. The key common services components are discussed in Chapter 6, “Common components” on page 203.

Note: For a complete list of all the Web service implementations based on the Parlay X V2.1 Web Services standards provided with IBM WebSphere Telecommunications Web service Server, refer to the IBM WebSphere Telecommunications Web Services Server Information Center at:

`http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp`

1.4 Introduction to the sample scenario described in this IBM Redbooks publication

Throughout this IBM Redbooks publication, we use a common example scenario in order to illustrate the procedure and the best practices for customizing the IBM WebSphere Telecommunications Web Services Server components, including the Service Policy Manager, the access gateway, and the development of a custom service implementation that gets deployed on the IBM WebSphere Telecommunications Web Services Server Service platform. The use of this example scenario helps to provide context within the following key areas of customization:

- Creation of a custom policy to determine if SIP addressing conversion needs to occur
- Modifying the default mediation flow for the Access Gateway by adding a custom primitive which, based on the custom policy, transforms the address for SIP addressing
- Creation of a custom service implementation that publishes the Presence status for a SOAP request.

Figure 1-2 illustrates the conceptual overview of this sample scenario.

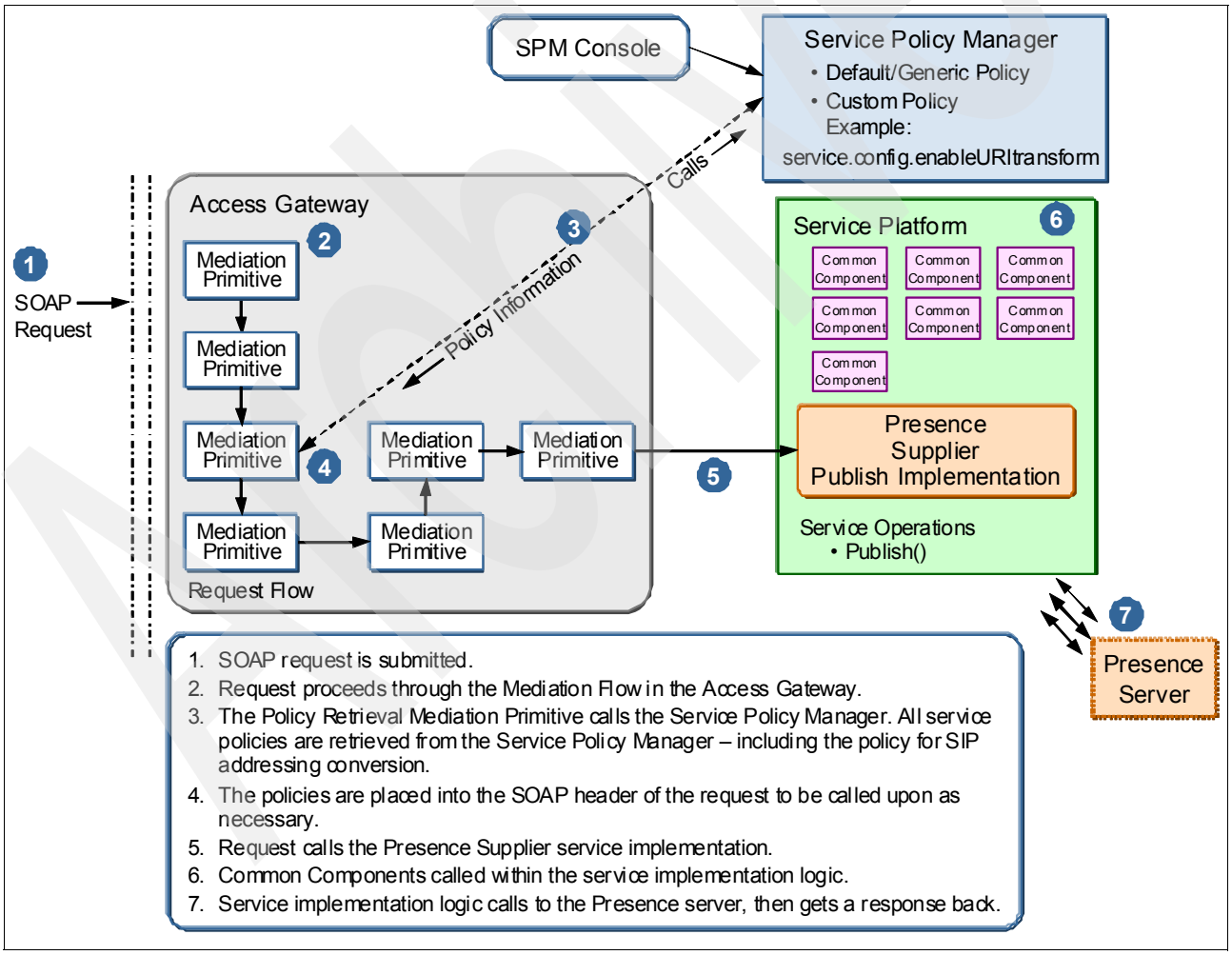


Figure 1-2 Conceptual overview of the use case

1.5 Roadmap to the chapters in this IBM Redbooks publication

Figure 1-3 illustrates a visual guide to the chapters in this IBM Redbooks publication.

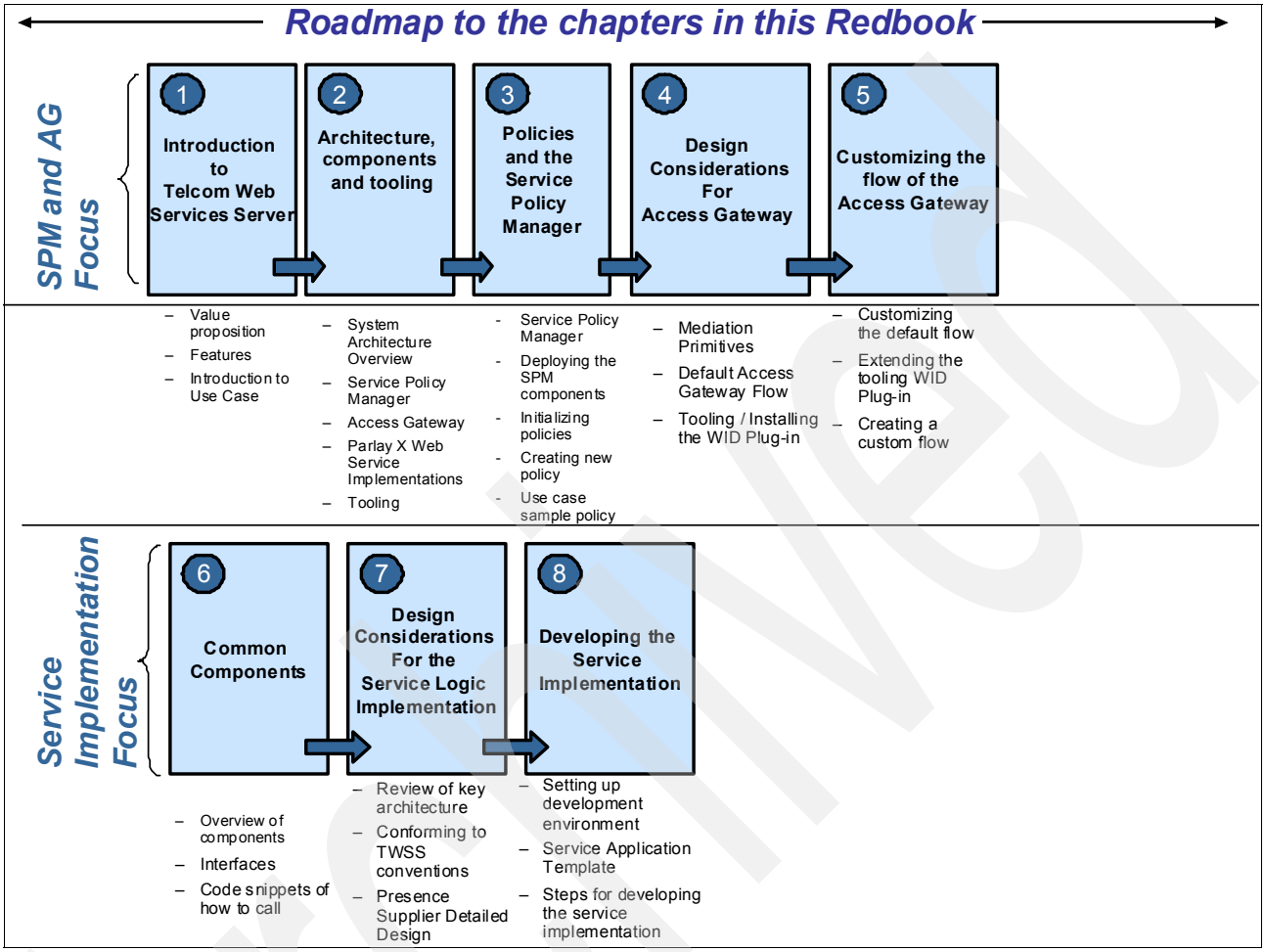


Figure 1-3 Overview of the chapters and topics covered in this IBM Redbooks publication

- This chapter provides an overview of the IBM WebSphere Telecommunications Web Services Server and describes how it can provide business value. Additionally, this chapter introduces the common use case we refer to throughout this IBM Redbooks publication.
- Chapter 2, “Architecture, components, and tooling” on page 9 discusses the architecture of IBM WebSphere Telecommunications Web Services Server. We provide both an overall system architecture overview, and examine how this product fits into the larger IMS architecture. We then discuss each of the components that make up the IBM WebSphere Telecommunications Web Services Server, namely the Service Policy Manager, the Access Gateway, and the Service Platform. Finally, we examine the key tooling components required to work with the IBM WebSphere Telecommunications Web Services Server.
- Chapter 3, “Working with service policies and the Service Policy Manager” on page 33 discusses service policies and the role of the Service Policy Manager in IBM WebSphere Telecommunications Web Services Server. It begins with an overview of the Service Policy Manager, discussing both the role of the runtime component and the Service Policy

Manager console. It also discusses how to deploy and configure each of the subcomponents.

- Chapter 4, “Design considerations for Access Gateway flows - base mediation flows and the mediation primitives” on page 67 provides an overview of the default mediation flow provided with IBM WebSphere Telecom Communications Server and discusses each of the default mediation primitives included.[1] We also discuss how to install the WebSphere Integration Developer Tooling required to begin working with the default mediation flow in the Access Gateway.
- Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127 describes how to develop and customize an Access Gateway Flow. In particular it covers the following customizations associated with the mediation flows:
  - Customizing the Default Mediation Flow by creating and adding a new mediation primitive into the default flow,
  - Extending the WebSphere Integration Developer Tooling Environment by converting a custom mediation primitive to a WebSphere Integration Developer Plug-in
  - Creating a new Mediation Flow from the beginning
- Chapter 6, “Common components” on page 203 discusses the common components provided with the Service Platform for IBM WebSphere Telecommunications Web Services Server. The common components are unique to IBM WebSphere Telecommunications Web Services Server in that they provide value and flexibility for customizing service implementations. Since the custom components are exposed through Web service implementations, this allows for substitution without requiring modifications to service implementation code. Common components are shared within the application server by all deployed service implementations.
- Chapter 7, “Design considerations for the service implementation” on page 227 introduces you to the design aspects of Service Implementations on Service platform component of IBM WebSphere Telecommunications Web Services Server. These design principles may be used as reference by architects and developers working on custom service implementations.
- Chapter 8, “Developing the service implementation” on page 251 describes the approach and steps required to develop the custom service implementation from our common sample scenario, namely the Presence publish operation. This chapter begins with a review of the development configuration requirements to begin creating the service implementation, then proceeds into the key steps for developing a Parlay X Web service. It discusses the core logic of the service implementation code, then discusses how to unit test and ultimately deploy and test the completed sample.

[1] For this IBM Redbooks publication, we discuss the mediation primitives provided with IBM WebSphere Telecommunications Web Services Server Version 6.2. For the latest information on mediation primitives provided with the product, also refer to the Information Center at:
http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

Archived

2

Architecture, components, and tooling

This chapter discusses the architecture of IBM WebSphere Telecommunications Web Services Server. We provide both a system architecture overview, and discuss each of the components that make up the IBM WebSphere Telecommunications Web Services Server, namely the Service Policy Manager, the Access Gateway, and the Service Platform. Finally, we examine the key tooling components required for customization of the IBM WebSphere Telecommunications Web Services Server.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

2.1 IBM WebSphere Telecommunications Web Services Server System architecture overview

IBM WebSphere Telecommunications Web Services Server enables service providers to expose high-level Web service interfaces for network services for third-party access.

Third parties are internal and external customers of service providers. They can be different internal divisions within an organization that wish to develop new services based on a core infrastructure function or external customers that wish to integrate new or enhance existing services. The Web service interfaces provide technology-agnostic access to service capabilities; each Web service interface can have multiple implementations (also referred to as instantiations or flavors) in a given service provider environment, providing access to IMS services through SIP, PSTN functionality through a Parlay/OSA gateway, Direct Connect™ access to network protocols, or custom integrated services. IBM WebSphere Telecommunications Web Services Server provides a middleware infrastructure for managing Web service access and provides an environment for hosting Web service API implementations.

The use of high-level Web service APIs as a means of exposing functions to third parties offers the following benefits:

- Isolation from evolving network technologies: With the emergence of the IMS architecture as a viable implementation standard, service provider networks are experiencing rapid change. Web services APIs provide a convenient, open standards-based abstraction for accessing service function. While Web service API implementations might require changes to address network evolution, Web service API application client code can remain the same.
- Focus on business functions: The Web services SOA architectural model promotes the definition of coarse-grained Web service interfaces that expose business-level functions. This enables third-party integrators to focus on service creation, rather than low-level implementation.
- Simplification of skill requirements: The use of high level Web service APIs reduces cost for deployment of new integrated services by enabling a large audience of Web programmers to access service function without requiring detailed infrastructure knowledge.

Figure 2-1 on page 11 illustrates an overview of the IBM WebSphere Telecommunications Web Services Server architecture, emphasizing how it is constructed from three key components:

- Telecom Web Services Access Gateway
- Service Policy Manager
- Service Implementations

The role and functionality of each of these three components is discussed in much greater detail in 2.1.3, “Components that make up the IBM WebSphere Telecommunications Web Services Server” on page 13.

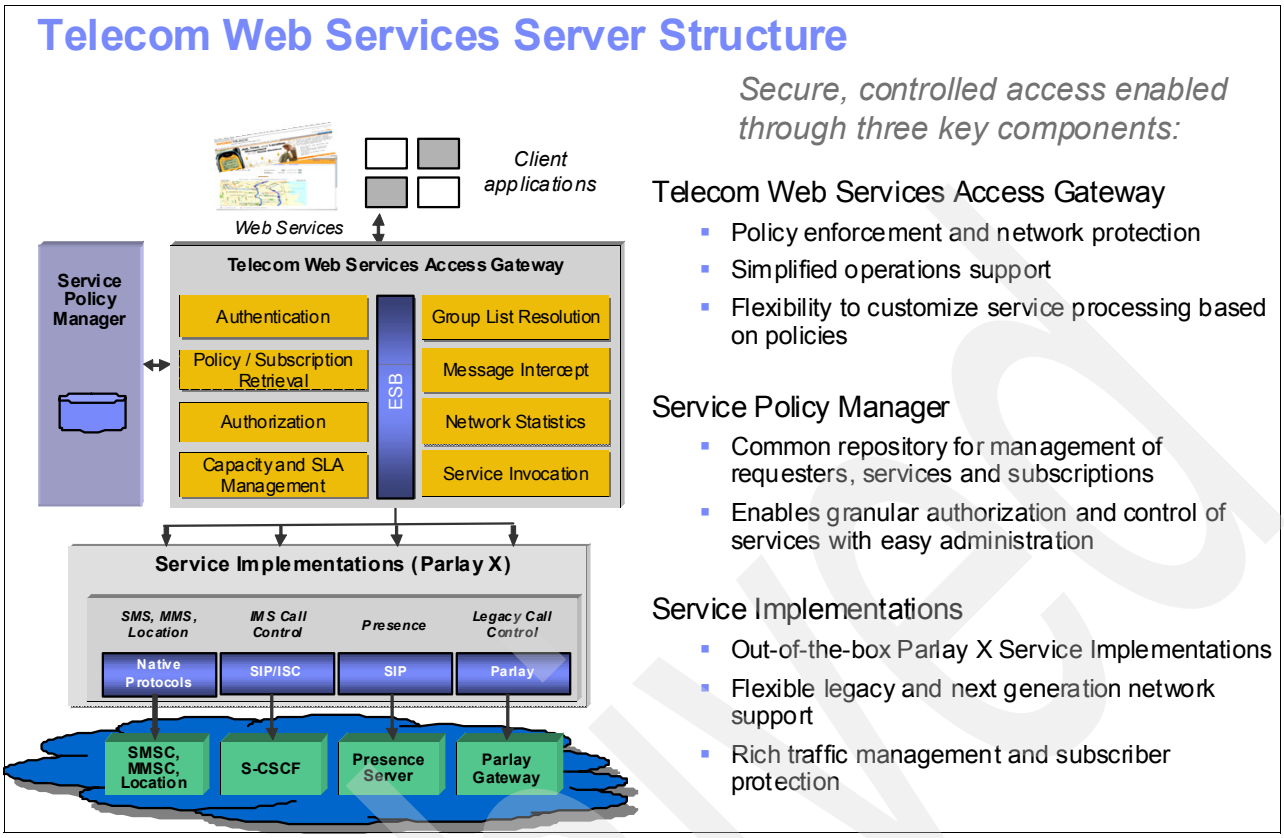


Figure 2-1 Architectural overview for IBM WebSphere Telecommunications Web Services Server

2.1.1 The role of Direct Connect and Parlay connector based services

The IBM WebSphere Telecommunications Web Services Server architecture enables Parlay X and other Web services to be efficiently implemented through the use of various other telecom protocols. This provides a standardized bridge from industry standard Web Services to existing lower level telecom network protocols. The IBM WebSphere Telecommunications Web Services Server infrastructure supports three broad categories of back-end network connectivity, including IP Multimedia Subsystem (IMS), Parlay/OSA, and Direct Connection to various telecom network elements.

IBM WebSphere Telecommunications Web Services Server runs on WebSphere V6.1, which provide a converged HTTP/ SIP (Session Initiation Protocol) container. Through the use of the SIP container, IBM WebSphere Telecommunications Web Services Server service implementations can communicate with SIP and other IMS devices and components. This protocol enables interoperability with call sessions and well as other IMS system components, such as the S-CSCF, the Presence Server, and the XML Document Management Server.

IBM WebSphere Telecommunications Web Services Server also enables connectivity to a variety of network elements that are common within a Telecom Service Provider environment. IBM WebSphere Telecommunications Web Services Server connects to these elements directly using the appropriate network protocol, such as SMPP for short messaging, MM7 for multimedia messaging, MLP for location, and potentially others. IBM WebSphere Telecommunications Web Services Server can utilize the Web container and JCA Connector Architecture to connect to these network elements.

IBM WebSphere Telecommunications Web Services Server also enables connectivity to a Parlay/OSA (Open Service Architecture) Gateway, which provides a connectivity to wireline and wireless network infrastructures. These implementations are needed for call control related Web Services (third-party call, call handling, terminal status, and so on) for earlier networks, but can also be used for location, SMS/MMS messaging, accounting, and other functions. The IBM WebSphere Telecommunications Web Services Server Parlay Connector component provides flexibility configuration to enable an IBM WebSphere Telecommunications Web Services Server Web service implementation to be a Parlay client application and integrate with a Parlay Gateway that is deployed in the service provider environment. Parlay communication is done over IIOP/CORBA communications and involves a sophisticated authorization strategy through the Parlay Framework.

The Parlay Gateway provides interoperability with service providers that own and operate PSTN networks. The Direct Connect capabilities provide an optimal integration between IBM WebSphere Telecommunications Web Services Server and common network elements found in a service provider environment and generally are associated with data services, such as SMS/MMS messaging, location, Presence, and so on. Both the Parlay Gateway and Direct Connect capabilities enable IBM WebSphere Telecommunications Web Services Server to integrate with existing networks, and provide a migration step for IBM WebSphere Telecommunications Web Services Server to integrate with emerging next generation SIP/IMS based networks.

2.1.2 The role of IBM WebSphere Telecommunications Web Services Server in IMS architecture

Before discussing the specific architectural components that comprise IBM WebSphere Telecommunications Web Services Server, let us examine where IBM WebSphere Telecommunications Web Services Server fits into the greater IP-Multimedia Subsystem (IMS) architecture.

The IP-Multimedia Subsystem (IMS) describes the next generation architecture for implementing IP-based telephony and multimedia services.

The IMS architecture describes functional relationships between network elements that provide call processing and service request routing at the core of the service provider network. The interfaces between network elements are based on IETF standard technologies, particularly the Session Initiation Protocol (SIP) and Diameter protocols. SIP is a signaling protocol used to establish media sessions between network entities. The goal of IMS is to enable service providers to rapidly create and deploy new services. The use of standard Internet technologies allows service providers access to a larger pool of technical resources and commoditized deployment platforms. Figure 2-2 on page 13 illustrates the role of IBM WebSphere Telecommunications Web Services Server within a logical view of the IMS architecture.

Archive

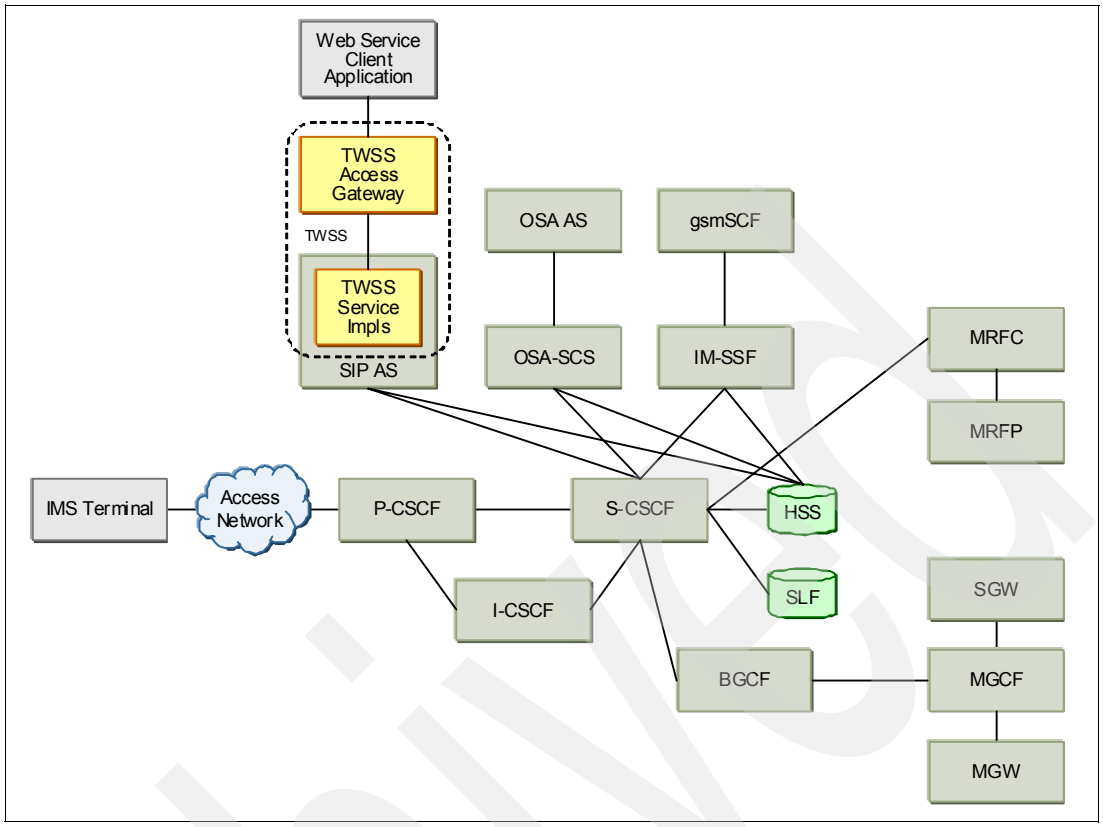


Figure 2-2 IBM WebSphere Telecommunications Web Services Server role as AS in IMS architecture

Figure 2-2 is a typical logical view of the IMS network functional areas. Some of the relevant acronyms for IBM WebSphere Telecommunications Web Services Server are expanded here:

P-CSCF Proxy Call Server Control Function
I-CSCF Interrogating Call Server Control Function
S-CSCF Serving Call Server Control Function
HSS Home Subscriber Server
SIP AS Session Initiation Protocol Application Server

An in-depth description of the function of these components is provided in the IMS Spec - 3GPP 23.228, found at:

`http://www.3gpp.org/ftp/Specs/html-info/23228.htm`

2.1.3 Components that make up the IBM WebSphere Telecommunications Web Services Server

IBM WebSphere Telecommunications Web Services Server is divided into three functional areas:

- Service Policy Manager

  The Service Policy Manager is primarily used by the Service implementations and ESB component to coordinate on policy information.

- Telecom Web Services Access Gateway

  Telecom Web Services Access Gateway provides policy-driven traffic monitoring, message capture, authorization, and management capabilities. These services are

provided at the application layer, and they are enforced for each Web service request using knowledge of the requester, target service, and invoked operation.

- Service Platform, which contains the Web service implementations.

  The Service Platform facilitates service exposure through standardized interfaces.

Figure 2-3 provides an overview of IBM WebSphere Telecommunications Web Services Server architecture.

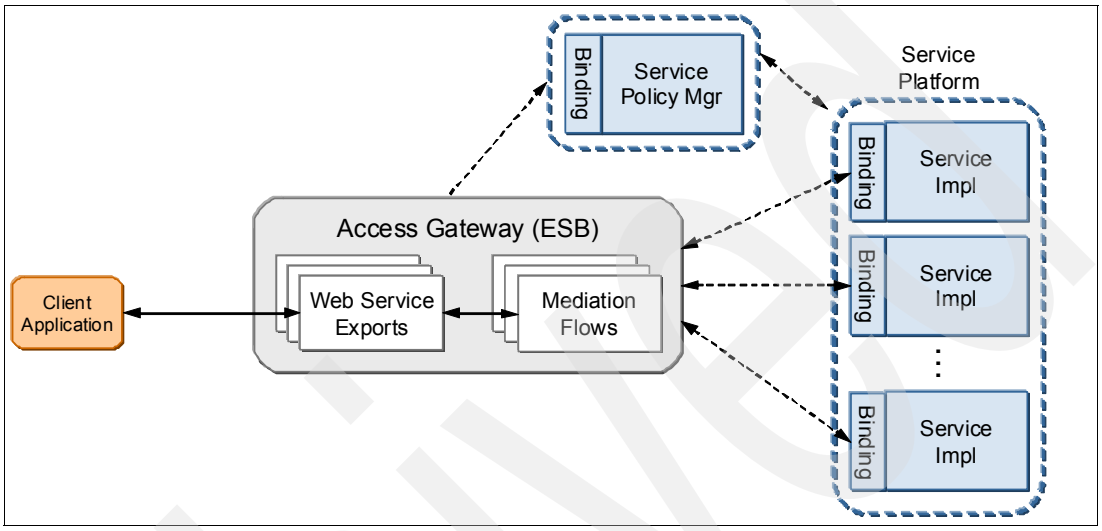


Figure 2-3 Overview of the three IBM WebSphere Telecommunications Web Services Server components

2.2 Service Policy Manager

The Service Policy Manager is used by Service Implementations and ESB components to coordinate policy information. It provides a structured data store and administrative interfaces for managing requesters, services, policies, and subscriptions within IBM WebSphere Telecommunications Web Services Server. The Web service implementations consist of an application server with enhanced support for telecom integration functions. The IBM WebSphere Telecommunications Web Services Server Product provides support for Parlay X V2.1, which is an industry standard set of interfaces to telecom functions, and other nonstandard Web service interfaces. Parlay X Web service implementations can adapt arbitrary Web service interface definitions and back-end network element implementations.

2.2.1 Underlying Service Policy Manager definitions in the context of IBM WebSphere Telecommunications Web Services Server

The Service Policy Manager is based upon service policies, requestors, services, and subscriptions. This section helps to define what each of these is and what role it plays in regard to the Service Policy Manager.

Requester

In IBM WebSphere Telecommunications Web Services Server, a requester is uniquely identified entity that wishes to access a service. The requester is the actual security entity obtained from WebSphere global security. A requester can be a user (like a subscriber), or an authority, or a company such as a large telco service provider. For example, a company (large

telco service provider), as a requester, can fetch service policies for a subscriber that it manages. During the run time, a requester could be a current user principal, or another user the current user principal delegates.

Service

A service is a collection of self-contained functions that perform a defined task and expose it through a well known interface. Users of the service only need to know what it does and what the interface is; the user does not need to know how it is implemented or deployed. Operations are important, but are sometimes considered part of the service.

Subscription

A subscription is a relationship between a requester and a service or an operation. The subscription is Policy Manager's authorization mechanism to grant a requester the permission to access and personalize a service in the form of policies. A requester must subscribe to the service before a policy can be added to the service by the requester.

Service policy

A policy is a configuration value that is associated with a particular scope of operation determined by the requester, service, and operation being executed. The configuration values can be of various data types, and control behavior of IBM WebSphere Telecommunications Web Services Server components within the Access Gateway or Service Platform. For example, presence service may have a presence subscription timeout policy, presence notification duration, and frequency policies. Third-party call service may have a call session timeout and a charging method (online or offline) policy. A service policy can often be used to describe QOS, or quality of service, for example, minimum and maximum bit-rate, as well as service availability rate. Note that not all policies relate to QoS.

2.2.2 Key features of the SPM and architectural overview

Highlights of the Service Policy Manager include:

- Highly structured data storage to store definitions of requesters, services, subscriptions, and policies
  - Hierarchical policy structure
  - Policies are name/value pairs
    - Example: requester.service.Authorized, value=true.
- Simplified setup and administration of requester groups and services
  - Flexible policy granularity
  - Global polices or policy by requester, service, or operation
- Policies used to personalize service by requester supporting features, such as:
  - Requester groups (Gold, Silver, and Bronze)
  - Geo-localization to enable centralized access with local service execution
    - Example: Select SMS service implementation based on the requester's country (for example, UK).
  - Authorization and Service Execution
    - Example: Select the SMS ConfirmDelivery operation.

Figure 2-4 illustrates both the architecture of the Service Policy Manager within IBM WebSphere Telecommunications Web Services Server and shows how a process flow executes when a Web service request enters the system, and then is routed to the service implementations.

Service implementations store dynamically-updatable, service-specific policies into a policy manager system, and the policy/subscription mediation primitive fetches policies from the policy manager system and populates SOAP headers with policy information. This policy information is then passed along with the request to downstream mediation primitives and back-end service implementations for policy-based decision making during service execution.

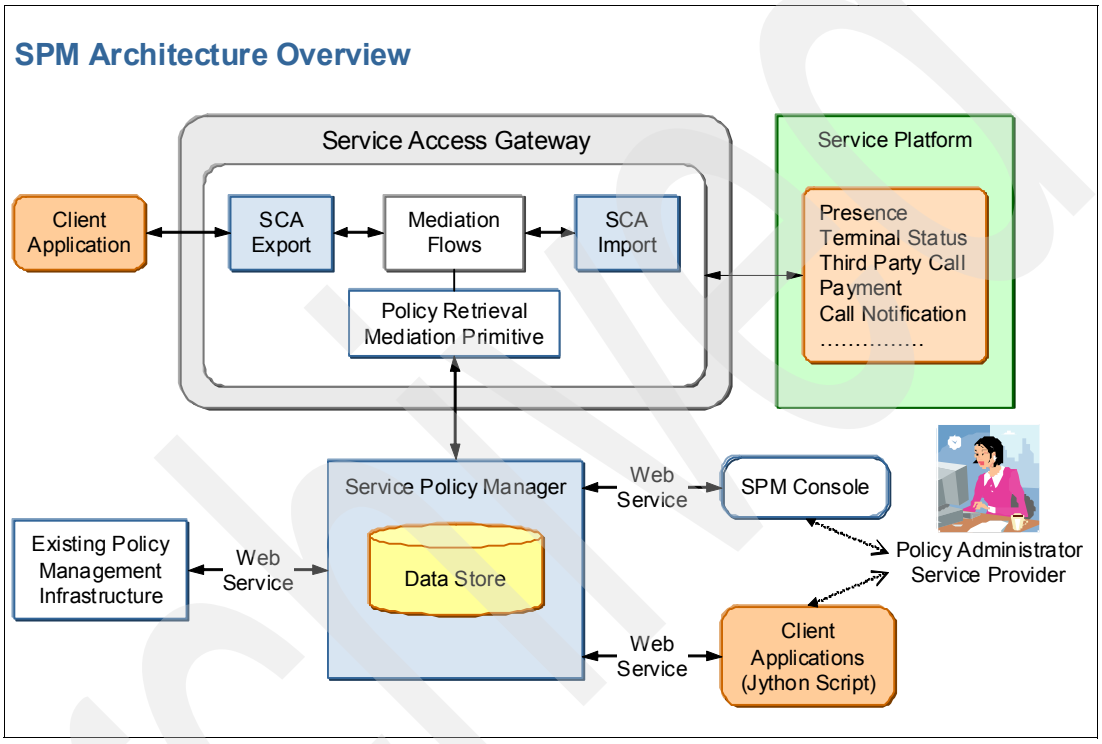


Figure 2-4 How a process flow executes when a Web service request enters the system

Note: A detailed discussion on how to deploy the SPM and work with policies, including how to create a custom policy, is provided in Chapter 3, “Working with service policies and the Service Policy Manager” on page 33.

2.3 Telecom Web Services Access Gateway

Telecom Web Services Access Gateway consists of a telecom-added function on top of WebSphere Enterprise Service Bus and provides policy-driven traffic monitoring, capture, authorization, and other message processing capabilities. These services are provided at the application layer, and are enforced for each Web service request using knowledge of the requester, target service, and invoked operation.

Highlights of the Access Gateway include:

- Uses Enterprise Service Bus to provide policy-driven authorization, SLA enforcement, and management capabilities.

  These services are enforced for each Web service request using knowledge of the requester, target service, and invoked operation.

- Access Gateway provides flexibility for the construction of tailored/customized message processing logic.
  - In accordance with service provider network policies/requirements.
  - WebSphere Integration Developer tooling allows the service provider to model its unique network policies, enable new features, and deploy customized logic into Access Gateway.

2.3.1 Telecom Web Services Access Gateway architecture

Telecom Web Services Access Gateway acts as a process-driven, message processing element. *Process flows* are used to capture and implement common business logic for service requests.

Telecom Web Services Access Gateway acts as a message processing intermediary. All Web service requests and responses pass through Telecom Web Services Access Gateway. Telecom Web Services Access Gateway consists of a telecom-specific function added on top of the WebSphere ESB platform. This telecom function is provided in the form of *mediation primitives* that can be used to construct *mediation flows* containing message processing logic.

Figure 2-5 illustrates a logical view of the Telecom Web Services Access Gateway architecture.

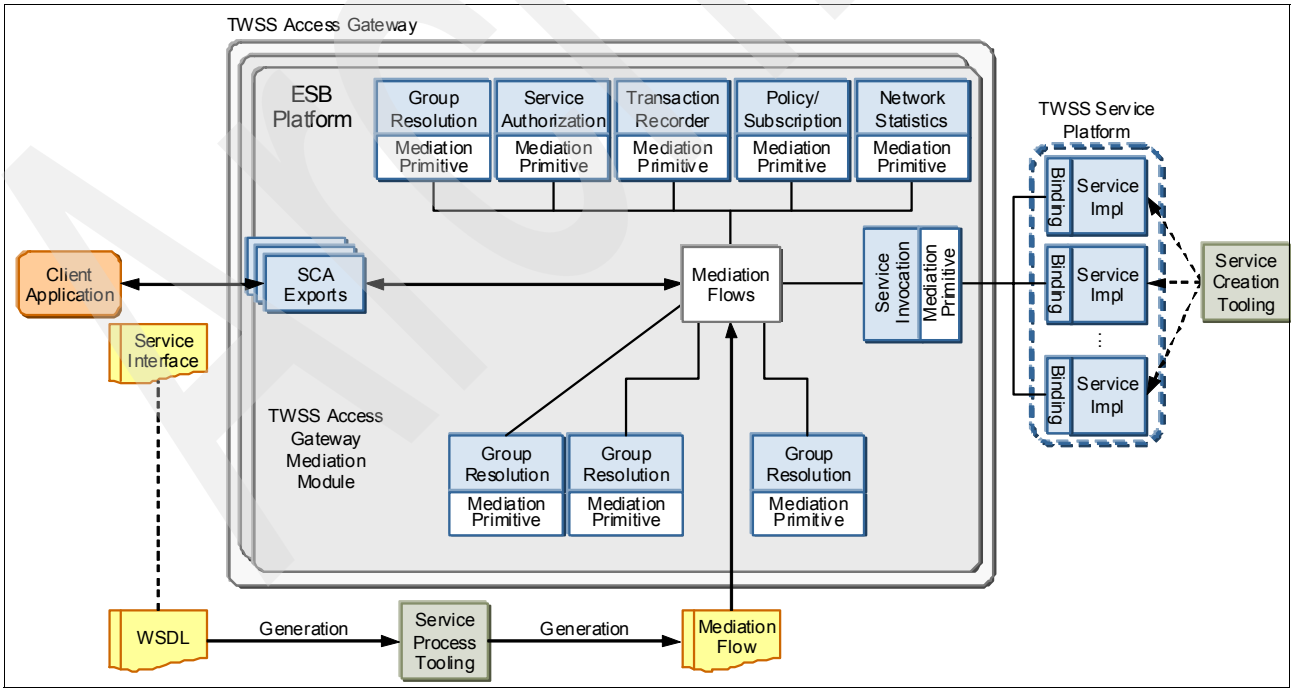


Figure 2-5 IBM WebSphere Telecommunications Web Services Server Access Gateway architecture

The execution of message processing logic within WebSphere ESB is orchestrated by a mediation flow module. The flow definition is produced within the WebSphere Integration Developer tooling. The flow definition artifacts are used by WebSphere Integration Developer to generated J2EE deployable code, in the form of an EAR, for deployment on top of the WebSphere ESB run time. Flow definitions describe the wiring between interfaces and mediation primitives; mediation primitives are units of logic that correspond to message processing elements in the visual flow programming environment. Examples of mediation primitives are a message logging primitive, a policy retrieval primitive, or a group resolution primitive.

Note:

- Details on how to install the WebSphere Integration Developer Plug-in to begin working with the Mediation flow artifacts is described in 4.5, “Tooling / WebSphere Integration Developer Plug-in” on page 88.
- Technical details on how to customize Access Gateway flows and create custom mediation primitives is discussed in depth in Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127.

Role of the Access Gateway

All Web service requests and responses pass through and are inspected by Telecom Web Services Access Gateway. Telecom Web Services Access Gateway acts as an intermediary for all communication between client and service endpoints, processing both incoming requests to Parlay X Web service implementations and outgoing requests made by Parlay X Web service implementations. Access gateway mediation primitives have access to the SOAP message in its entirety, including headers and body. This allows mediation primitives to determine the initiating requester, target service, and invoked operation. In addition, information is communicated to other Telecom Web Services Access Gateway mediation primitives and back-end components using SOAP headers. This ensures that information is available for any external Web service requests that need to include this information. For example, policy data is passed in the form of SOAP headers between mediation primitives and relayed to Parlay X Web service implementations during invocation of them.

Flow of information - from client operation request to invoking the appropriate service implementation

Upon receipt of an application client operation request and after evaluating mediation flow logic for the incoming request, Telecom Web Services Access Gateway creates its own back-end request to the appropriate service implementation. This invocation is performed using runtime selection of the back-end service implementation based on policy information. Upon receipt of the response from the service implementation, Telecom Web Services Access Gateway evaluates mediation flow logic on the returned response and then issues an appropriate response to the client application. For Parlay X Web service implementations that are capable of issuing outbound requests, initiated through a notification service or through other means, these implementations must also send outbound requests through Telecom Web Services Access Gateway to be processed in a similar fashion.

All communication between the external third-party client application and Telecom Web Services Access Gateway is done using SOAP/HTTP(S) using document literal encoding. This provides maximum interoperability between endpoints, as literal encoding has the widest platform support and interoperability characteristics. Communication between Telecom Web Services Access Gateway and back-end service implementation uses SOAP over the most appropriate transport. Both HTTP and secure HTTPS transports can be used for delivering the SOAP message. WS-security can be enabled for mediation module flows within the standard WebSphere Integration Developer tooling.

Role of mediation primitives

Telecom Web Services Access Gateway components are provided in the form of additional mediation primitives that extend WebSphere ESB and plug into the WebSphere Integration Developer Tooling Environment. Each mediation primitive is designed to serve a specific, well-scoped purpose; the preference is create a bunch of different variations of a mediation primitive over a larger, more generalized primitive. This allows the system integrator to choose the most appropriate primitive when designing a mediation flow. This aligns with the visual programming primitives within WebSphere Integration Developer tooling; the system integrator selects the mediation primitive most appropriate for a given situation when creating a mediation flow. This approach also encourages code reuse; rather than trying to create generalized logic, each primitive can excel at its particular specialty. As mediation primitives appear in the WebSphere Integration Developer tooling palette, over time this palette can expand to include additional telecom function, resulting in a highly industry-specific business logic creation environment.

Note: A description and functional specification for each of the default mediation primitives provided with IBM WebSphere Telecommunications Web Services Server V6.2 is provided in Chapter 4, “Design considerations for Access Gateway flows - base mediation flows and the mediation primitives” on page 67.

Mediation modules follow the SCA component model used within WebSphere ESB. The SCA component model is described within the WebSphere ESB Information Center documentation. Each mediation model contains a one or more SCA exports that correspond to a Services Description Language (WSDL) interface definition and are exposed as a Web service to external client requesters. Mediation modules can also contain SCA imports for accessing back-end functions. In the current WebSphere ESB implementation, each import must be associated with a single endpoint, whose value can be configured statically during deployment of the mediation module. As a result, an additional mediation primitive is provided with Telecom Web Services Access Gateway to enable runtime, policy-driven selection of back-end services.

The WebSphere ESB platform provides flexibility for the construction of tailored message processing logic in accordance with service provider network policies. This logic is typically unique to each service provider network and can also vary according to the particular service being invoked. The mediation primitive programming model offers a point of extensibility for the creation of customer-specific function, providing a well-defined programming model that furthers the capabilities of the WebSphere Integration Developer Tooling Environment. IBM WebSphere Telecommunications Web Services Server provides a set of default mediation flows as part of its component deliverables. Service providers can choose to use this flow as-is or to construct a customized flow and mediation primitives using WebSphere Integration Developer.

2.4 Service Platform and the Web service implementations

IBM WebSphere Telecommunications Web Services Server hosts the Parlay X Web service implementations that maps Web service interfaces to service provider network functions. The term service implementation is used to refer to the implementation code of the Web service API; a single Web service interface can have multiple service implementations that provide a different mapping of the same API to service provider network function. IBM WebSphere Telecommunications Web Services Server consists of an WebSphere Application Server instance, WebSphere Application Server Version 6.1 or higher, for SIP protocol support with additional Telecom Web Services service implementations applications deployed on the instance. Telecom Web Services service implementations provide support function for Parlay

X Web service implementations and are accessible through Web services. The components are intended to be accessed using local Web service invocations; this allows WebSphere Application Server to perform local optimizations and reduce the cost of Telecom Web Services service implementations Web service calls. Thus, in a cluster deployment, each Parlay X Web service implementations WebSphere Application Server instance within will have these Telecom Web Services service implementations installed locally.

IBM WebSphere Telecommunications Web Services Server focuses on the use of Parlay X APIs as the Web service interface abstraction for telecom network function, but the architecture is general enough to provide infrastructure services for any kind of Web service implementation.

The Parlay X Web service implementations provide access to IMS, Parlay, and other telecom network elements

2.4.1 Parlay X Web service implementations architecture

IBM WebSphere Telecommunications Web Services Server provides a development platform, execution environment, and common set of components to facilitate rapid development of Web service implementations.

The term service implementation describes an implementation of any Web service API. An individual Web service API can have more than one service implementation, with a different back end for each API realization. The Parlay X Web service APIs are the first instantiation of services. Consequently, parts of the design can include a function that provides specific support for Parlay X services, but the architecture is general enough to provide infrastructure services for any kind of Web service implementation.

Figure 2-6 on page 21 provides an overview of IBM WebSphere Telecommunications Web Services Server architecture, with a specific focus on the Service Platform.

Archived

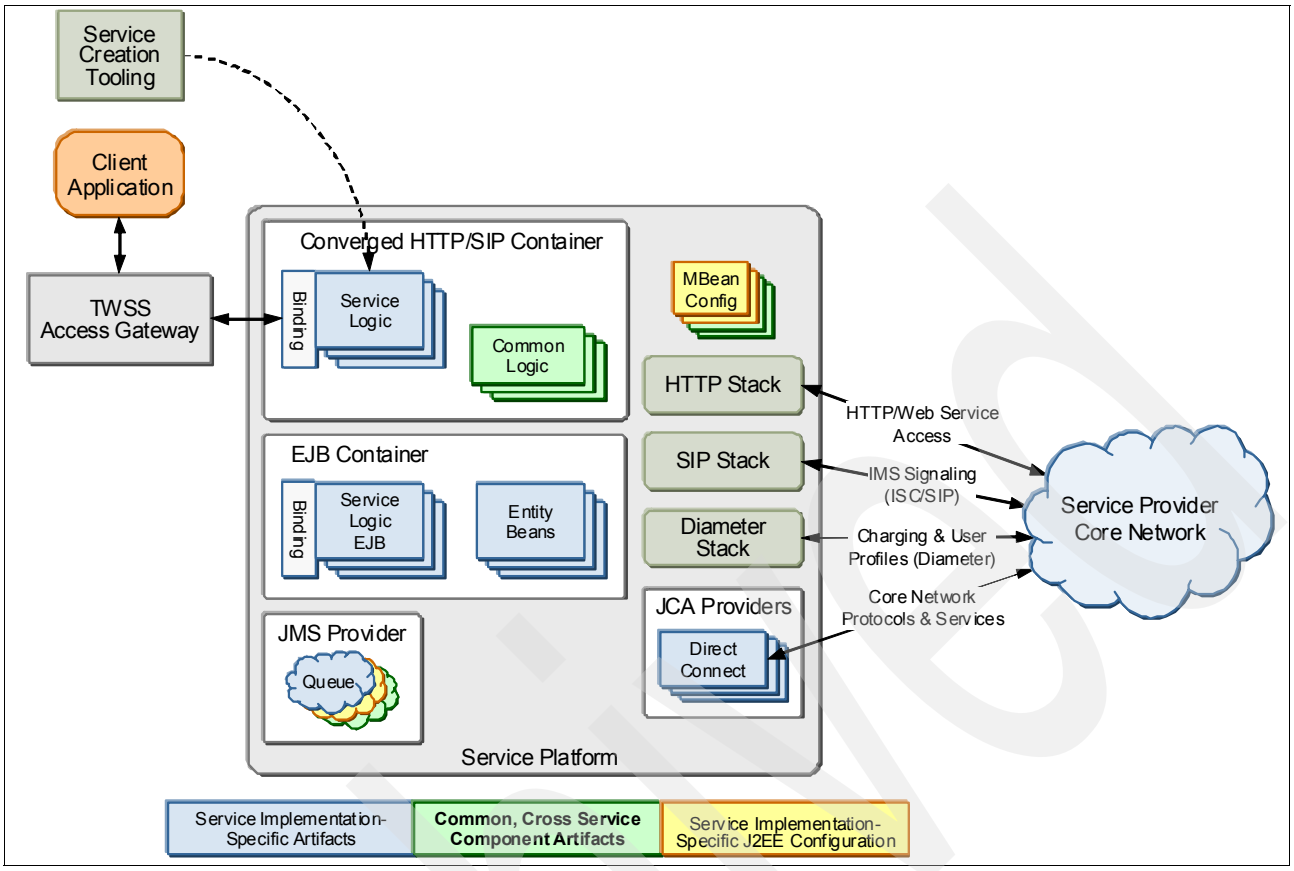


Figure 2-6 IBM WebSphere Telecommunications Web Services Server service platform architecture

All Web service requests and responses pass through and are inspected by Telecom Web Services Access Gateway before being passed on for processing. Telecom Web Services Access Gateway acts as an intermediary for all communication between client and service endpoints, processing both incoming requests to Parlay X Web service implementations and outgoing requests made by Parlay X Web service implementations. For each incoming request, Telecom Web Services Access Gateway consults policy information from the Subscription Management system in order to select the service endpoint and invoke the appropriate service implementation. Parlay X Web service implementations provide a bridge between the Web service API and core network, typically initiating one or more network-level activities for each request. For Parlay X Web service implementations capable of issuing outbound requests, either provisioned through a notification service or through other means, these implementations must send outbound requests through Telecom Web Services Access Gateway to be processed in a similar fashion.

Communication between Telecom Web Services Access Gateway and Parlay X Web service implementations uses SOAP messages over HTTP. The HTTP transport should be used when tighter latency response times are required. Document literal is the preferred SOAP message encoding format. It allows maximum interoperability between Web service clients. Document literal encoding should be used as the encoding format when generating all Web service proxy stubs. The architectural design focuses primarily on the use of document literal encoding over HTTP. HTTPS communication is also assumed to be an alternative transport to HTTP where HTTP is mentioned within this document, but requires additional deployment-time configuration.

IBM WebSphere Telecommunications Web Services Server consists of a WebSphere Application Server (WebSphere Application Server) environment with support for IMS protocols. The converged HTTP/SIP Servlet container is used for hosting Web and SIP application logic. SIP Servlets can interact with the IMS SIP signaling elements according to the 3GPP ISC interface. The IMS Connector stack provides a client API for accessing Diameter services for integration of security, user profile, and accounting functionality. For direct access to network protocols for which a stack in base WebSphere Application Server does not exist, the Java Connector Architecture (JCA) framework provides a means of extending the application server environment with additional protocol stacks. Finally, the Java Messaging Service (JMS) provided with base WebSphere Application Server can be used for asynchronous, latency-insensitive, and high throughput communication between application components.

Parlay X Web service implementations are based primarily on the Web container programming model for Web service implementation bindings and common service components. JAX-RPC Servlets are the recommended Web service implementation binding, although Parlay X Web service implementations are free to choose the method that is most efficient for the given application (for example, use of EJB™ bindings if transactions are necessary).

Common components

In addition to these WebSphere Application Server facilities, Telecom Web Services service implementations contain support function for Web service implementations. Each of these components has an associated Web Services Description Language (WSDL) interface and is Web Services accessible. This allows for substitution and customization of Telecom Web Services service implementations by deploying a different implementation of the Telecom Web Services service implementations interface and selecting the new implementation through its new endpoint. Telecom Web Services service implementations should be accessed using local Web service invocations. This allows for WebSphere Application Server to perform some in-process optimizations, reducing calling impact. As a result, within a clustered configuration, each WebSphere Application Server instance within the cluster will have Telecom Web Services service implementations deployed locally so that they can be accessed by Parlay X Web service implementations using local Web service invocations.

Note: Common components are discussed in greater detail in Chapter 6, “Common components” on page 203.

All components are to be bundled as individual J2EE enterprise application archive (EAR) files for deployment within the WebSphere Application Server environment. Parlay X Web service implementations EAR files should contain all service logic and runtime dependencies. Telecom Web Services service implementations should each have their own EAR file that contains the component logic. The use of EAR packaging promotes separation between individual service implementations, as well as access to Telecom Web Services service implementations. The use of EAR packaging also promotes substitution of interfaces: each interface implementation (for example, a SIP or SMPP implementation of an SMS service or a customized common component implementation) will have a corresponding EAR file. This implementation can be selected at runtime by using the appropriate endpoint information for the invocation.

Figure 2-7 on page 23 provides an overview of Telecom Web Services service implementations, deployed in EAR form on top of a WebSphere Application Server instance. Some of these components exist to facilitate development of Parlay X Web service implementations; new Parlay X Web service implementations should choose to include whichever Telecom Web Services service implementations are appropriate.

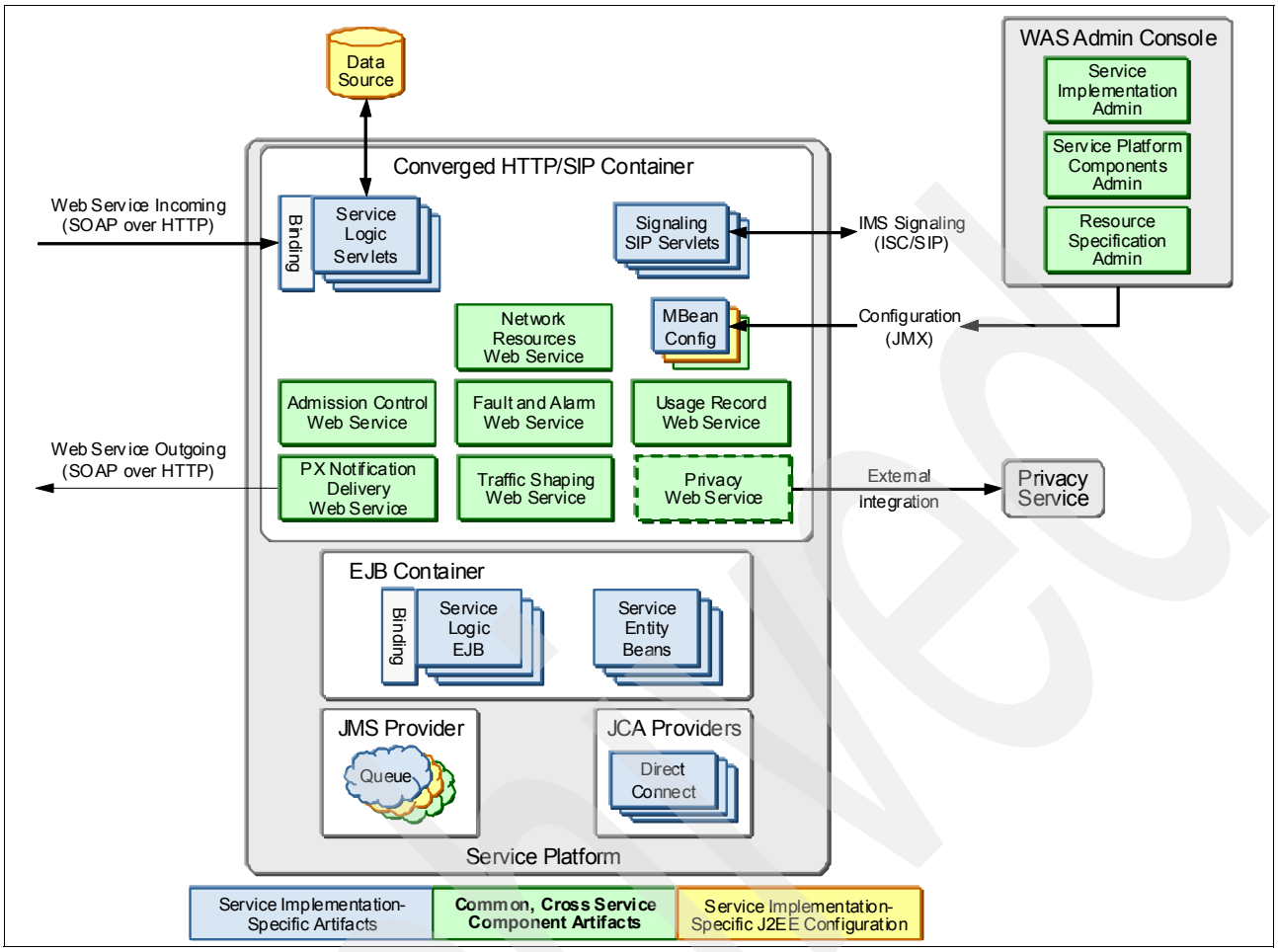


Figure 2-7 IBM WebSphere Telecommunications Web Services Server service platform and common components

When creating a Web service implementation, you must select a concrete binding to the J2EE programming model. Parlay X Web service implementations architecture favors a Web container programming model. The recommended choice of service implementation bindings is a Servlet-based binding, such as JAX-RPC Servlets, although service implementations are free to choose the most efficient access strategy for their particular application. The Servlet threading model favors short execution of Servlet methods. Thus, Parlay X Web service implementations should avoid performing large chunks or high latency work within Servlet methods. This should be addressed either through design of the Web service interface or through use of JMS queues within the design of the service implementation.

The service logic Servlets contains the glue code for each service that invokes the appropriate Telecom Web Services service implementations. While each service has unique requirements that influence the chosen architecture, the general suggested architecture for creating Parlay X Web service implementations is to use a service controller model, where the service is broken into two main parts: a front-end controller that contains logic that is common to all potential implementations of a Web service interface and a back end that provides network-specific logic. This enables substitution of the back end for different network access contexts; however, the level of substitutability is at the Web service abstraction and corresponding EAR file. Thus, this separation should be maintained at the code level and each access context should be built as a separate EAR. This architecture is particularly useful for Direct Connect protocol function, where each back end can make use of a different network protocol to perform the desired function. This architecture is also implied by service

implementations that make use of SIP Servlets, where the HTTP interactions are used to control SIP protocol logic.

2.4.2 Parlay services

Parlay/OSA APIs, also known as Open Systems Architecture APIs, enable you to expose network capabilities as IT interfaces. IBM WebSphere Telecommunications Web Services Server comprises Parlay connector, a component that allows to deploy services as Parlay client applications. These services are capable of interacting with a Parlay Gateway and eventually establish communication with a Parlay service capability function, such as, for example, Parlay V4.2 specification, Part 14, Presence and Availability Management SCF.

Note: To learn more about Parlay V4.2 specifications, refer to the link below:

`http://portal.etsi.org/docbox/TISPAN/Open/OSA/Parlay42.html`

2.4.3 Direct Connect services

Telecom services can be implemented to interact with network elements directly in some cases. Telecom Web service Server comprises Web services that implement Short Message Peer to Peer (SMPP), Multimedia Messaging (MM7), Mobile Location Protocol (MLP), and XML Capability Application Part (XCAP) application protocols out-of-box. All these implementations utilize J2EE Connectors (JCA) as the medium to connect to respective network elements. The WebSphere Application Server, which is the underlying application server environment for IBM WebSphere Telecommunications Web Services Server, supports a J2EE connector adhering to the JCA V1.5 specification. The JCA programming model offers extensive facilities for defining a wide range of interfaces for protocol communication and allows for creation of container managed threads for performing lower level message and data processing tasks.

Note: JCA connector architecture has limited support for failover within WebSphere Application Server. The backup connector instances can be supplied within the cluster, but these instances cannot share state directly; a JMS queue can be used internally to ease the process of failover. However, without additional state, the connector might not be able to provider proper failover behavior. Thus, it is the responsibility of a higher level service to ensure that failover requirements are met and interface as a low-level element with the J2EE connector.

As a guideline to implement Direct Connect implementation using a J2EE connector, consider the following situations:

- When the service implementation needs to access a protocol that is not natively supported within the WebSphere Application Server protocol stack, specifically the byte-level protocols.
- When there is a need for handling asynchronous, inbound notifications from the network interfaces that cannot be handled using J2EE messaging components directly.

When there is a need for a service implementation to access a proprietary APIs for tight integration with a older network element or COTS application. Such APIs sometimes require a different transaction model than that can be provided by an application server environment. Such integration with the network interfaces or applications is encapsulated into a JCA connector. This approach isolates the remainder of the service logic from the specifics of integration.

2.5 Extensibility

The following section provides an overview of possibilities for extending and customizing both the Access Gateway and the service implementation.

2.5.1 Telecom Web Services Access Gateway extensibility

The mediation primitive approach of WebSphere ESB provides a well defined mechanism for packaging and managing delivered components, while enabling customized logic to be introduced in a non-invasive manner.

Mediation primitives should be designed to serve a specific purpose. Thus, a system integrator can select the most appropriate mediation primitive within the tooling for the task at hand. The WebSphere Integration Developer tooling also provides an extensible mechanism for adding additional mediation primitives into the tooling palette. This provides a telecom-specific flow creation environment and easy access to telecom-specific functions during flow customization.

Recognizing that message processing flows will be unique for each network's policies and the organization requirements for each service provider, Telecom Web Services Access Gateway tooling approach provides an environment where message processing logic can easily be customized. IBM WebSphere Telecommunications Web Services Server includes a set of default flows that encompass a standard set of message processing logic. These flows are delivered both in binary form and in a form that can be imported into the WebSphere Integration Developer tooling for customization. The intent of the binary form is to provide functions out-of-the-box, while the flow can then be easily customized and extended for the target environment.

Note: Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127 provides actual examples in which the Access Gateway can be extended and customized to meet your organization’s specific needs.

Archived

Figure 2-8 highlights the role of WebSphere Integration Developer tooling within the IBM WebSphere Telecommunications Web Services Server architecture. Telecom Web Services Access Gateway components are delivered as binary mediation primitives that plug in to the WebSphere ESB run time. Message processing logic consists of a process model called a mediation flow that describes the interactions between mediation primitives (through wiring of input and output terminals) and Web Services Description Language (WSDL) interfaces. Each WSDL interface is associated with an SCA export or import. An SCA export is a callable interface, while an SCA import is an invocation to another interface. Each SCA export or import is given a binding, which determines how the interface can be called. SOAP/HTTP Web service bindings are used within the IBM WebSphere Telecommunications Web Services Server architecture.

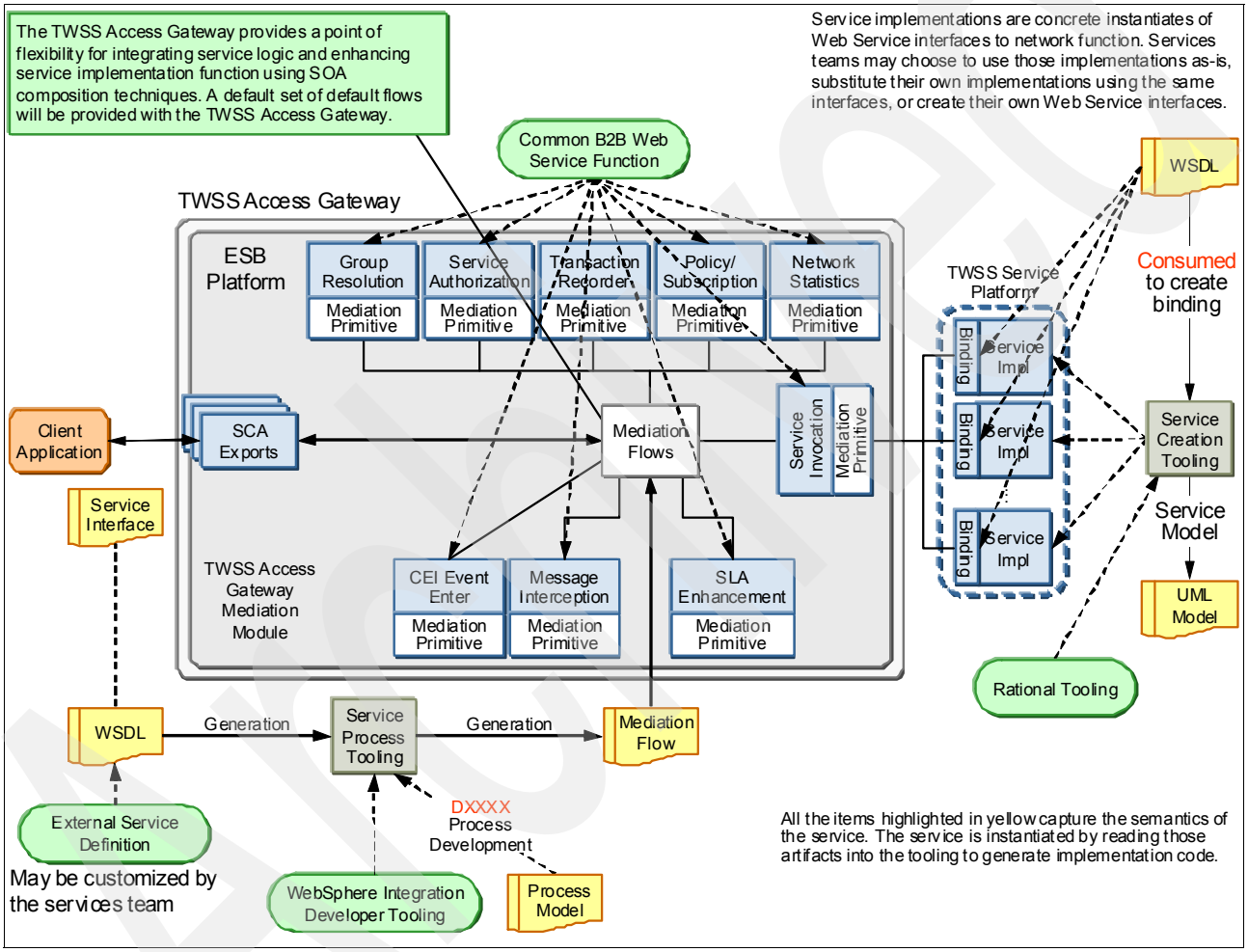


Figure 2-8 IBM WebSphere Telecommunications Web Services Server Access Gateway Architecture and the role of WebSphere Integration Developer tooling

During execution of the mediation flow, a data object (called a Service Mediation Object (SMO)) is passed between mediation primitives. This data object is a business object representation of the Web service SOAP request; Telecom Web Services Access Gateway components use a loosely coupled SOAP model for passing data between components, but include additional information within SOAP headers for processing by downstream elements. This data is also included within the back-end request to Telecom Web Services service implementations.

To create deployable code, the WebSphere Integration Developer tooling generates J2EE staging projects (such as Web and EJB projects) from the flow description artifacts. These staging projects are then bundled into a J2EE EAR for execution within the WebSphere ESB server. Additional configuration and deployment of the flow can then be achieved through the WebSphere ESB administration console.

Mediation primitives within Telecom Web Services Access Gateway are intended to be policy-driven. In a typical flow, the policy mediation primitive calls out to a service policy Management system that contains requester-specific policy data for tailoring execution flow. This policy information is inserted into the SOAP headers for processing by downstream elements. Additional Telecom Web Services Access Gateway mediation primitives should look for policy information to provide any required configuration information. Mediation primitives can also possess static properties that can be configured within the tooling on the properties page; however, such properties can only be statically set within the tooling and thus should only be used for default or flow-context specific data.

2.5.2 Parlay X Web service implementations extensibility

Telecom Web Services service implementations are callable through a Web service interface that is defined using the Web Services Description Language (WSDL) interface. The WSDL for a particular service must match between the Access Gateway and its Service Implementation running on the Service Platform.

Implementations of the WSDL interfaces are designed to solve a specific, narrowly scoped goal. A single interface can have multiple implementation flavors, for example, performing the task specified within the interface through different means. The appropriate implementation flavor is selected at run time through the appropriate endpoint; this allows for substitution of interface implementations through configuration, without requiring changes to the code.

Within IBM WebSphere Telecommunications Web Services Server, there are two basic categories of components: Parlay X Web service implementations and Telecom Web Services service implementations. Each of these categories corresponds to a potential substitution point. For example, an SMS interface can be substituted with a service implementation that provides SMS functionality over SIP or one that provides a similar function over SMPP and Telecom Web Services service implementations can be substituted with customer-unique implementations. Since components are bundled as individual EARs, an interface can be substituted by deploying a new EAR containing the new implementation and then setting the invocation endpoint to point to the new EAR.

The rationale behind this approach is that generalization of component function is difficult, while creating domain specific components is much easier to get right and typically results in higher performance code. The best implementation can then be selected at run time according to the requirements of the service being delivered.

This approach also increases the independence of additional Parlay X Web service implementations, and Telecom Web Services service implementations can be built with and maintained by reducing the coupling between components.

The substitution methodology also has advantages for serviceability: a minor upgrade can be performed by deploying a new EAR containing the new version and changing the endpoint to correspond to the new implementation. A IBM WebSphere Telecommunications Web Services Server architectural requirement is that each component read in its set runtime configuration at the start of each call flow (or sequence of interactions) and that such information remain constant throughout the lifetime of the call flow. This supports the substitutability concept, allowing for maintenance without interruption.

Substitution can be done at on a component basis for minor upgrades, and for major release levels, a complete system substitution should be done by setting up a new system in parallel, and then routing a new workload to the new system.

Note: Chapter 8, “Developing the service implementation” on page 251 discusses the approach and process for developing a custom service implementation.

2.6 Tooling

The following section describes the tooling required to work with IBM WebSphere Telecommunications Web Services Server. There are three areas (see Figure 2-9) of development activities around IBM WebSphere Telecommunications Web Services Server, each one requiring its specific set of tools.

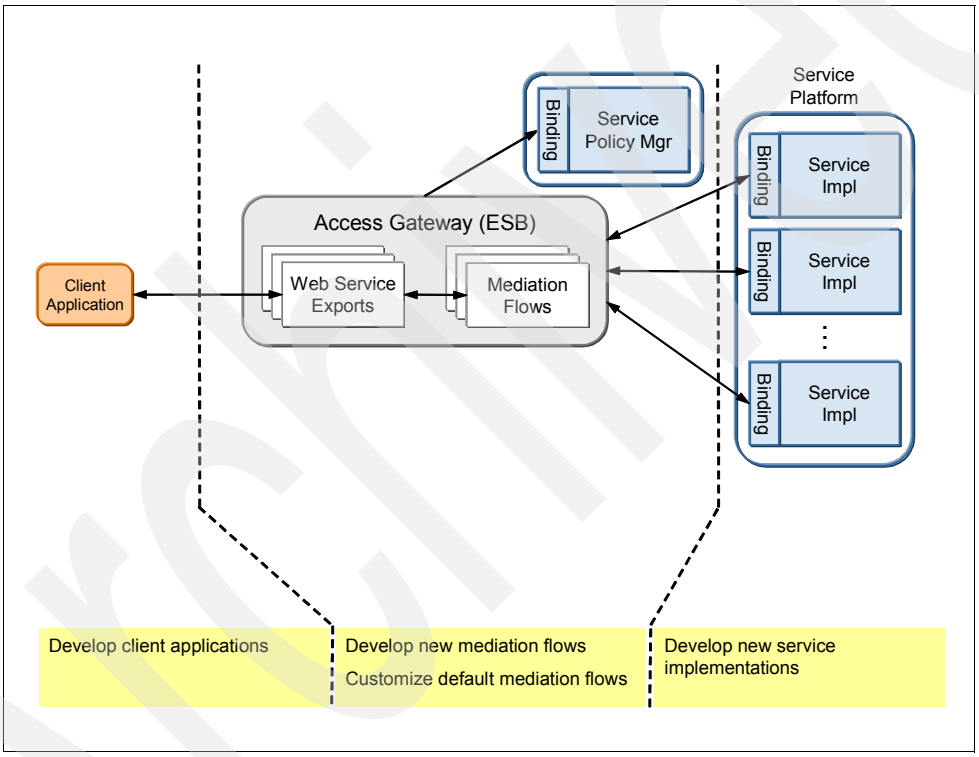


Figure 2-9 Development activities

2.6.1 Tooling for developing Service Implementations

The Rational® Application Developer v7 (RAD) development environment is used to develop service implementations.

Rational Application Developer provides a single, comprehensive development environment designed to meet a variety of development needs, from Web interfaces to server-side applications, from individual development to advanced team environments, from Java development to application integration. Rational Application Developer is part of the Rational Software Development Platform series of products, which is a platform of products all built on Eclipse, which is an open-source platform for creating application development tools. Each product in the Rational desktop family offers the same integrated development environment

(IDE). The differences among these products reflect which plug-in tools are available in each configuration. The IBM Rational Application Developer is a full suite of development, analysis, and deployment tools for rapidly implementing J2EE applications, Enterprise JavaBeans™, portlets, and Web applications.

A summary of the key features of Version 7 that are relevant to developing service implementations or client applications are:

- Full support for Java EE V1.4, Java SE V5.0, and IBM WebSphere Application Server V6.1.
- Based on Eclipse V3.2.
- Application Developer V7.0 supports Java 5. There is tooling for such features as Web tooling.
- Web Tooling.
  - The Web Diagram Editor is rewritten to leverage the Graphical Modeling Framework (GMF).
  - Drag and drop functionality (from the Palette) in the Web Diagram Editor updates the diagram and (behind the scenes) generates appropriate code (keeping diagram and code in-sync).
- JavaServer™ Faces (JSF):
  - Full support for JSF V1.1.
  - New version of the IBM JSF Widget Library (JWL), including AJAX-like behavior.
  - Support for JSF portlet bridge.
  - Support for standard JSF only mode (which excludes usage of IBM-specific JSF components) as well as support for third-party JSF components.
  - Support for multiple faces configuration files.
- Test Server environments.
  - Test environments included for WebSphere Application Server V6.1, V6.0, V5.1, and WebSphere Portal V6.0, V5.1, and WebSphere Express V5.1.
  - Integration with IBM WebSphere Application Server V6.0 for deployment, testing, and administration is the same (test environment, separate install, and Network Deployment edition).
- XML.
  - Updated support for XML and XSLT tooling.
  - Updated support for XML schema editing, including visual modeling.
  - Updated support for XML schema to Java code generation.
- Web services.
  - Series of usability improvements in Web services development (improved skeleton merge for top-down Web service creation, simplified editing of WSDL and XML schema, and remote WSDL validation).
  - Complex schema support with SDO.
  - Enhanced support for XSD.
  - Support for WSDL and XSD modeling.

Archived

- Debugging.
  - Support for debugging WebSphere Jython administration scripts.
  - Support for DB2® V9.0 Stored Procedure Debug.

2.6.2 Tooling for developing mediation flows

The WebSphere Integration Developer environment is used to customize the Parlay X message processing flows. Customizing the flows is optional and customization depends on the needs of your environment. WebSphere Integration Developer tooling is also necessary to create custom flows for non-Parlay X Web services.

The IBM WebSphere Integration Developer is a role-based development environment based on the Eclipse V3.0 platform. It can be used in conjunction with other Rational and WebSphere tools. Each user has a unique tooling perspective based on their role (for example, J2EE developer, business analyst, or integration developer).

To simplify and accelerate the development of integrated applications, this environment provides a layer of abstraction that separates the visually-presented components you work with from the underlying implementation.

You can use WebSphere Integration Developer to graphically model and assemble mediation components from mediation primitives, and assemble mediation modules from mediation components:

- If the interface for SCA mediation components are not imported, you can use the Simplified Interface Editor to create the interface. You can use this editor to specify and edit interfaces (operations and parameters) of mediation modules.
- You can use the Mediation Flow Editor to map between operations on the endpoints of a mediation to define the set of mediation flows needed for this application. You can use a set of predefined mediation primitives to visually compose a mediation flow.
- You can use the Business Object Editor to construct the messages that are used in mediations.
- You can use other editors to extend the development environment to meet your business needs, for example:
  - Create and edit custom mediation primitives, and add them to the Mediation Flow Editor.
  - Create and edit message descriptors.

Applications created using the WebSphere Integration Developer conform to a number of industry-wide standards. These include:

- J2EE Connector Architecture is used for connectivity.
- Java Message Service (JMS) is used for asynchronous messaging, and in cases where guaranteed delivery of data is required.
- Simple Object Access Protocol (SOAP) is used for integrating Web Services.
- Web Services Description Language (WSDL) is used for describing services.

In addition to the business process and integration, WebSphere Integration Developer also provides support for development of mediation services. Mediation services intercept and modify messages that are passed between existing services (providers) and clients (requesters) that want to use those services. Mediation modules are typically deployed on the WebSphere Enterprise Service Bus.

For example, mediation flows can be used to find services with specific characteristics that a requester is seeking and to resolve interface differences between requesters and providers. For complex interactions, mediation primitives can be linked sequentially. Typical mediations include:

- Transforming a message from the sending service to a format that the receiving service can process
- Conditionally routing a message to one or more target services based on the contents of the message
- Augmenting a message by adding data from a data source

In addition to these mediations that are included with the base tool, the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer plug-in offers several additional mediation primitives that are essential to handle IBM WebSphere Telecommunications Web Services Server service requests, such as:

- Adding IBM WebSphere Telecommunications Web Services Server headers to the request
- Retrieving the relevant policies for an incoming request and adding them to the message header

2.6.3 Tooling for developing client applications

The whole IBM WebSphere Telecommunications Web Services Server environment is based on a Service-Oriented Architecture design. As such, the client applications are only dependent on the definition of the Web service interfaces that are exposed by IBM WebSphere Telecommunications Web Services Server. The client applications can be implemented in any technology that allows you to send and receive SOAP messages.

A suitable set of tools to build the client side applications is provided by the IBM Rational Application Developer, which is covered in 2.6.1, “Tooling for developing Service Implementations” on page 28.

However, if your applications are built on the standard Parlay X Web Services API, this can save valuable development efforts by using the WebSphere Telecom Toolkit. This toolkit is an Eclipse plug-in to the Rational Application Developer.

The WebSphere Telecom Toolkit has two features, namely Telecom Web Services feature and the IMS Enablement feature. The Telecom Web Services feature provides a complete environment to develop and test Telecom client applications using Parlay X V2.1 APIs. The feature includes wizards, cheat sheets, snippets, samples, and a simulator that emulates a real telecommunication network. The IMS Enablement feature provides a collection of tools for developing other IP Multimedia Subsystem applications.

Using the Telecom Web Services feature of the WebSphere Telecom Toolkit gives you a fast start to build Parlay X based applications as well as to run tests independent of the back-end services.

Archived

Table 2-1 provides the complete set of tools provided by the Telecom Web Services feature. The tools are categorized into application development tools and application testing tools. The application development tools include the telecom application templates, snippets, WSDL import wizard, cheat sheets, and telecom Web samples. The Application Testing tools include the simulator configuration Editor, the simulator and the runtime views.

Table 2-1 WebSphere Telecom Toolkit features

Feature	Description
Application templates	The Telecom Web Application Template sample provides a template Web project to develop new applications. The template, when imported into the Rational Application Developer workspace, creates Web and Enterprise Application projects. The Web project contains all the required Parlay X API jars in its classpath.
Cheat Sheets	The telecom cheat sheets provide guided steps to create a new application using the telecom Web application template sample.
WSDL Import wizard	The Parlay X V2.1 WSDL import wizard is used to import Parlay V2.1 WSDLs into an existing project.
Snippets	The telecom snippets are used to insert predefined working code into a Java class or Java Server Page. There are snippets to call various Parlay X Web services, which include: ► Short Message Service ► Multimedia Message Service ► Terminal Location ► Terminal Status ► Accounting ► Payment ► Notification Administration ► Third Party Call ► Audio call ► Group Management ► Presence
Application Samples	The telecom samples built using the toolkit can be deployed and tested on the Web services client simulator. The toolkit also provides a tutorial with step-by-step guidance on how to build the sample.
Simulator configuration	The toolkit ships a default simulator configuration along with the telecom samples. The user can create a custom simulator configuration using the Telecom Simulator Configuration wizard. The simulator configuration editor is used to edit a user created custom simulator configuration.
Simulators	The Web Services Client Simulator emulates a Parlay X gateway and provides a test suite to test user developed Parlay X Web applications without the need of a real network. The simulator uses a configuration file to configure its test data.
Runtime views	The simulator runtime views display the simulator runtime data while the simulator is in operation.

Archive

3

Working with service policies and the Service Policy Manager

This chapter discusses service policies and the role of the Service Policy Manager in IBM WebSphere Telecommunications Web Services Server. It begins with an overview of the Service Policy Manager, discussing both the role of the runtime component and the Service Policy Manager console. It also discusses how to deploy and configure each of the subcomponents.

With a foundation in place for the Service Policy Manager, it then discusses policies. We begin with a discussion about how to initialize base policies required for the Access Gateway and default mediation flow. Next, we discuss how to create a new policy.

Finally, this chapter provides a sample use case on creating a policy for SIP address conversion.

Specific topics in this chapter include:

© Copyright IBM Corp. 2008. All rights reserved.

3.1 Focus of this chapter within the context of the common use case

This chapter focuses on three aspects for setting up the initial policies and creating a custom policy.

- "Deploying the Service Policy Manager components" on page 37
- "Initializing policies" on page 51
- "Creating a new policy" on page 53

This chapter also includes a sample service policy for SIP adressing conversion, discussed in the following sections:

- "Sample for Service Policy Manager - SIP addressing conversion" on page 61
- "Use case realization" on page 62

Figure 3-1 illustrates the primary focus of this chapter within the context of the common use case.

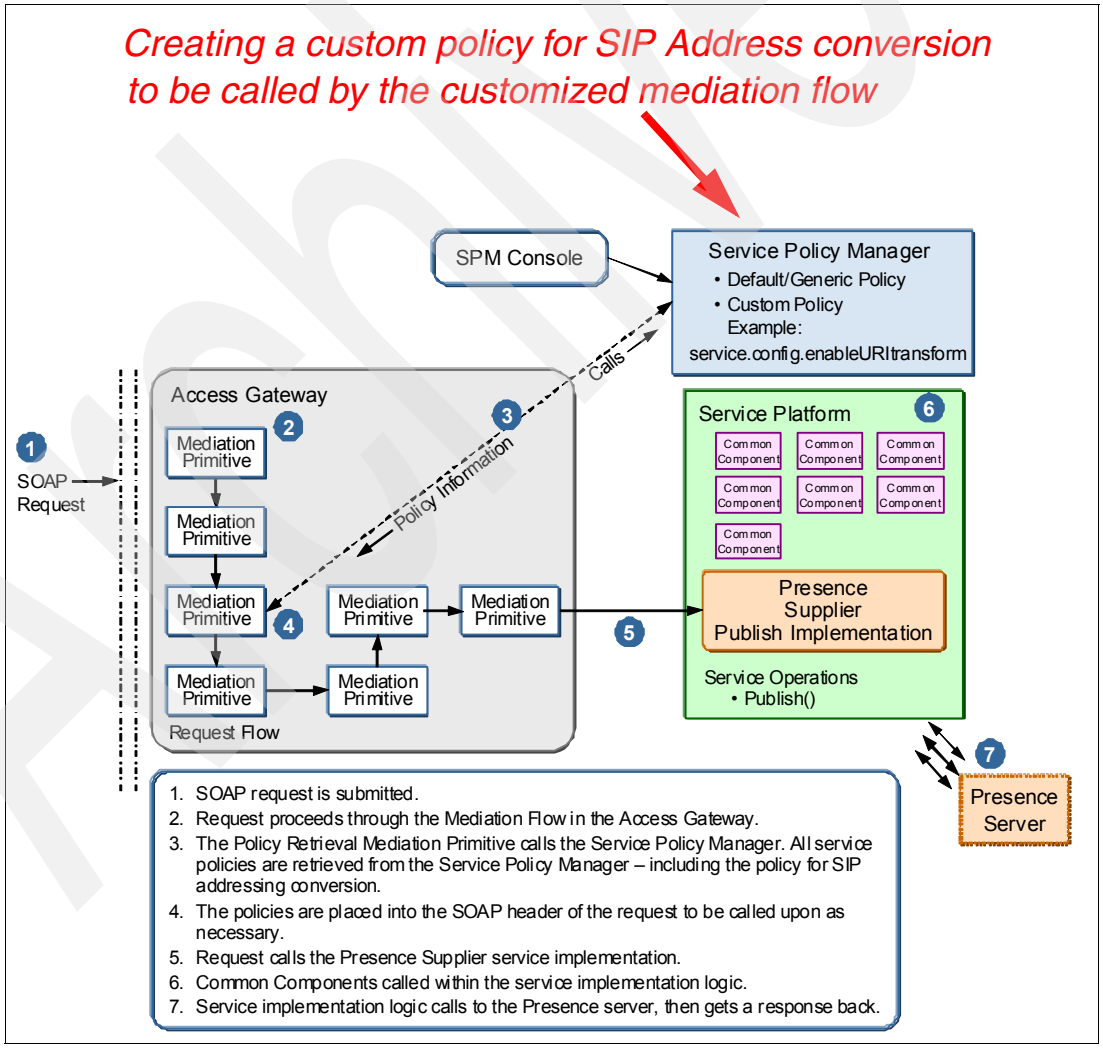


Figure 3-1 Creating a custom policy for SIP Address conversion called by a mediation flow

3.2 Overview of policies

The Service Policy Manager provides management, storage, and retrieval functions for the policy configuration data, and the runtime data used to customize service delivery for a given requester. It manages hierarchical policy definitions and service subscriptions for requesters. The key benefits of this component are a common database for the management of requesters, services and subscriptions, support for attaching policies to managed entities, and hierarchical resolution of policies enabling rich policy resolution with easy administration.

Using policy management capabilities, an enterprise-level administrator manages definitions of third-party requesters, service definitions, and service relationships; they also personalize the services provided to groups and to individual requesters in a way that is scalable.

A service policy defines a piece of runtime configuration data for a particular service. Service policies can be associated directly with requesters (as part of a subscription) to provide personalization of service delivery. It is defined as a name and a value, with the interpretation of the service policy performed by message processing or application logic.

You access Service Policy Manager from your applications through a Web service, which enables Service Policy Manager to be deployed and managed independent of the applications that utilize it. Service Policy Manager uses a Web services interface and must be integrated with the service provider environment. Service Policy Manager recognizes policies as data, which it manages and stores, but the services that define and use these data are solely responsible for interpreting it.

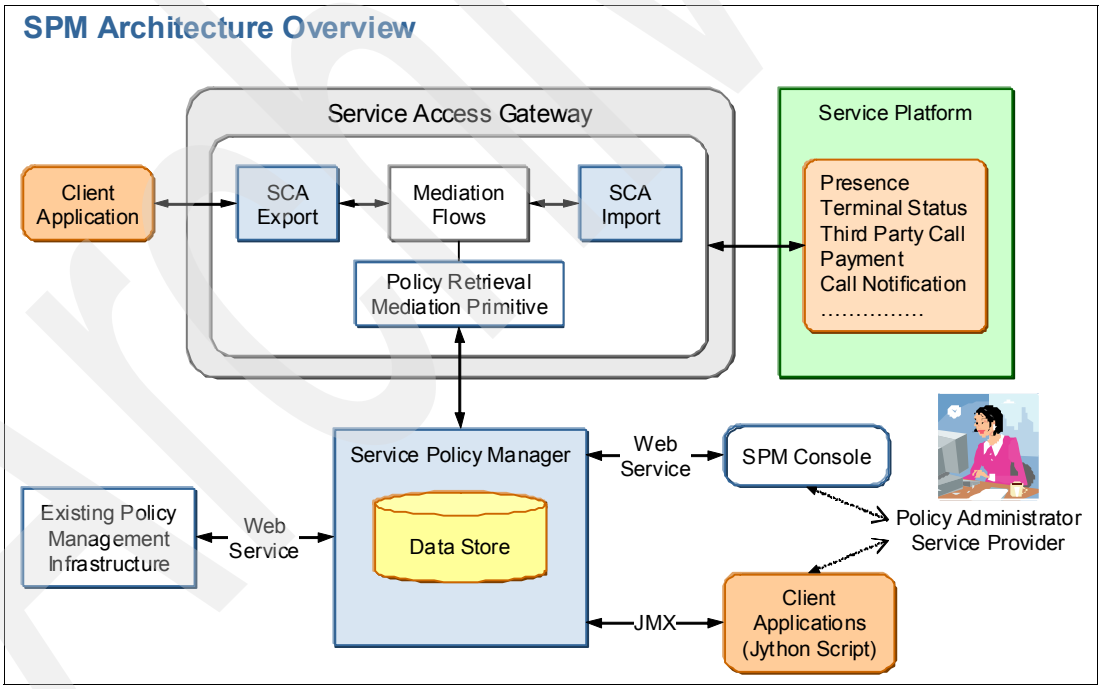


Figure 3-2 Service Policy Manager architecture overview

3.3 Overview of the Service Policy Manager

The Service Policy Manager is split into two parts:

- The runtime subcomponent
- The console

The following section gives an overview of each component, and then discusses how to deploy each one.

3.3.1 SPM runtime component

The runtime subcomponent provides the actual storage, management, and policy resolution capabilities. Its functions can be accessed through two sets of interfaces, both of which are available either through Web services or through Jython scripting (using the WebSphere Application Server wsadmin tool), or through MBeans that can be accessed within wsadmin. The wsadmin tool now supports the Jython scripting for advanced automation capabilities.

The policy access interface is used by non-administrative applications to look up policies. It is a set of administrative interfaces through which requesters, services, subscriptions, data types, and policy values can be managed

The Service Policy Manager resolves policy values using a hierarchical algorithm. The requester and service information can be organized into a tree hierarchy that allows for groupings of requesters and services. The subscriptions can be set within the requester tree scope, and policies can be set within the requester and service tree scopes. The lookup process is hierarchical, allowing requesters and services that are lower in the tree to inherit subscriptions and policy values from their parents.

Each service represents an interface that is managed within the Service Policy Manager. Multiple service implementations, or back-end implementations, can be registered within the Service Policy Manager, and policies can be administered across the service as a whole or within the context of each unique back end.

A special value of ALL can be used to apply the policy across one element of the scoping. For example, defining a policy at a scope of (ALL, myservice, ALL), will define an attribute or value pair for the service myservice across all requesters and operations. The Service Policy Manager uses a hierarchical resolution algorithm to determine the final set of policy attribute or value pairs for a given service context.

3.3.2 SPM console

The Service Policy Manager console provides an administrative view of the requesters, services, operations, subscriptions, and policies that are managed by the Service Policy Manager.

IBM WebSphere Telecommunications Web Services Server Service Policy administrators use this console application to create, modify, and remove operational policy data that controls the behavior of the IBM WebSphere Telecommunications Web Services Server. This information is defined within various scopes. The scopes are indexed in two ways: by the requester name of the application invoking the system, and by the service identifier and operation name of the service being invoked.

Archived

This administration console communicates with the Service Policy Manager run time through a Web service invocation. The endpoint URL of the Web service is configured here. It enables this console application to communicate with the Service Policy Manager where the policy data is stored.

3.4 Deploying the Service Policy Manager components

The following section discusses how to deploy both the runtime component and the console for the Service Policy Manager.

Note: Before you begin, you must make sure the Service Policy Manager applications have been installed. First, perform an initial check within the file system for SPM_Runtime.ear and SPM_Console.ear.

These files always appear in <$WAS_install_root>/installableApps/TWSS-Base/ directory.

Check the IBM WebSphere Telecommunications Web Services Server InfoCenter for Service Policy Manager Application installation guidance.

Also, you need to initialize the database for Service Policy Manager. Check the IBM WebSphere Telecommunications Web Services Server InfoCenter for guidance about installing database initialization files for Service Policy Manager and how to connect to the database:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

Use the WebSphere administrative console to prepare the cluster environment for deployment of Service Policy Manager applications. Refer to the WebSphere Application Server Network Deployment InfoCenter for information about creating the cluster:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

3.4.1 Deploying the Service Policy Manager runtime component

The Service Policy Manager runtime component is a single enterprise application that must be deployed in the cluster. You can install and deploy the application on the deployment manager. Then, you can synchronize all nodes in the cluster to replicate the application across the cluster.

1. Launch the WebSphere Administrative Console in the Web browser by using the address `http://<IP Address> :<Port number>/ibm/console/`. Enter the User ID and click the **Login** button.
2. In the navigation window, select **Applications** → **Install New Applications** (Figure 3-3).

Figure 3-3 WebSphere Application Server navigation window - Applications

3. Locate the SPM_Runtime.EAR file, select **Show me all installation options and parameters**, and click **Next** (Figure 3-4).

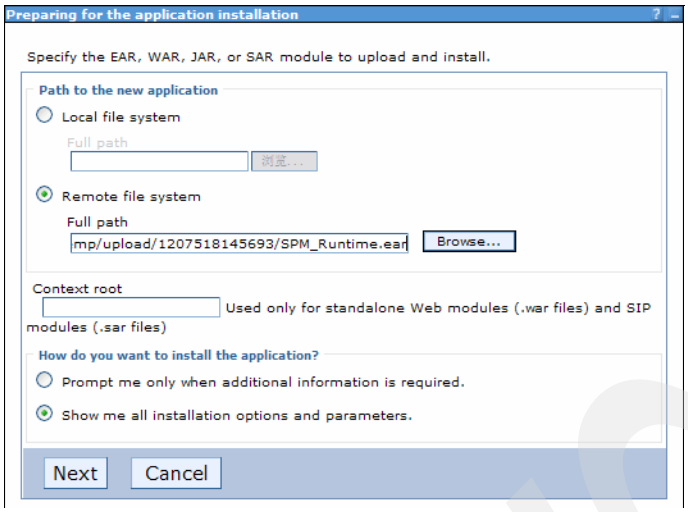


Figure 3-4 Preparing for the application installation

4. In the Preparing for the application installation window, keep the defaults and click **Next**. (Figure 3-5).

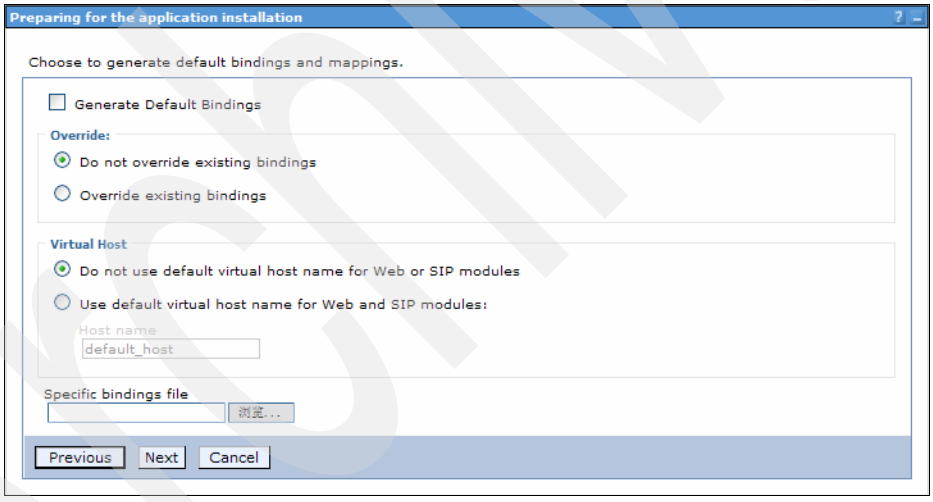


Figure 3-5 Preparing for the application installation

5. In the Select installation options window, keep the defaults and click **Next** (Figure 3-6 on page 39).

Archive

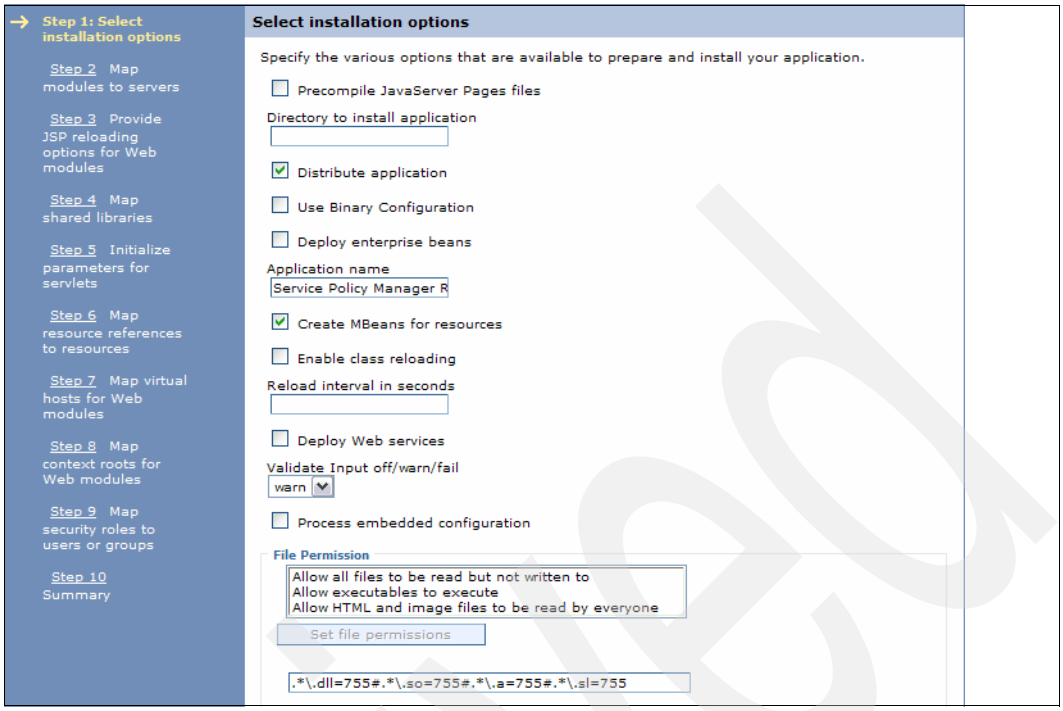


Figure 3-6 Deploy the Service Policy Manager runtime component step 1

6. In the Map modules to servers window, map all of the modules to the cluster that you created for deployment of Service Policy Manager applications (Figure 3-7).

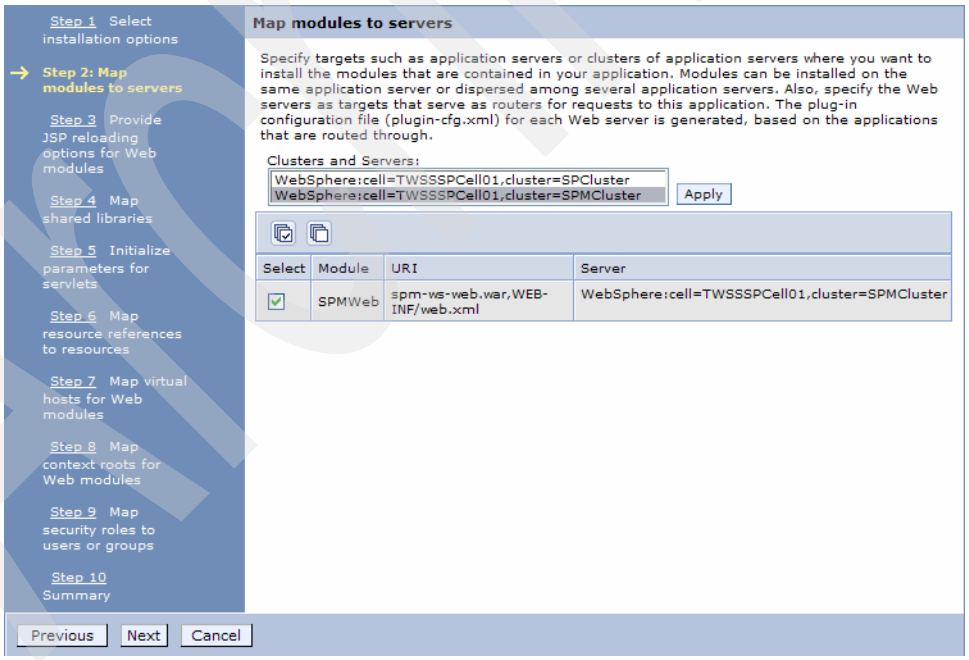


Figure 3-7 Deploy the Service Policy Manager runtime component step 2

7. In the Provide JSP™ reloading options for Web modules window, keep the defaults and click **Next** (Figure 3-8).

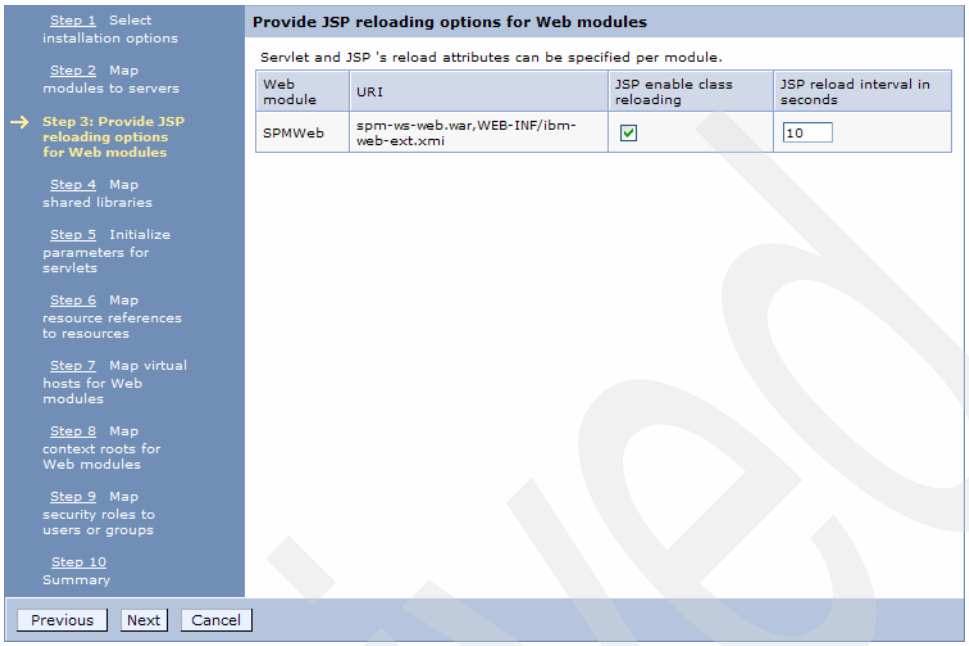


Figure 3-8 Deploy the Service Policy Manager runtime component step 3

8. In the Map shared libraries window, keep the defaults and click **Next** (Figure 3-9).

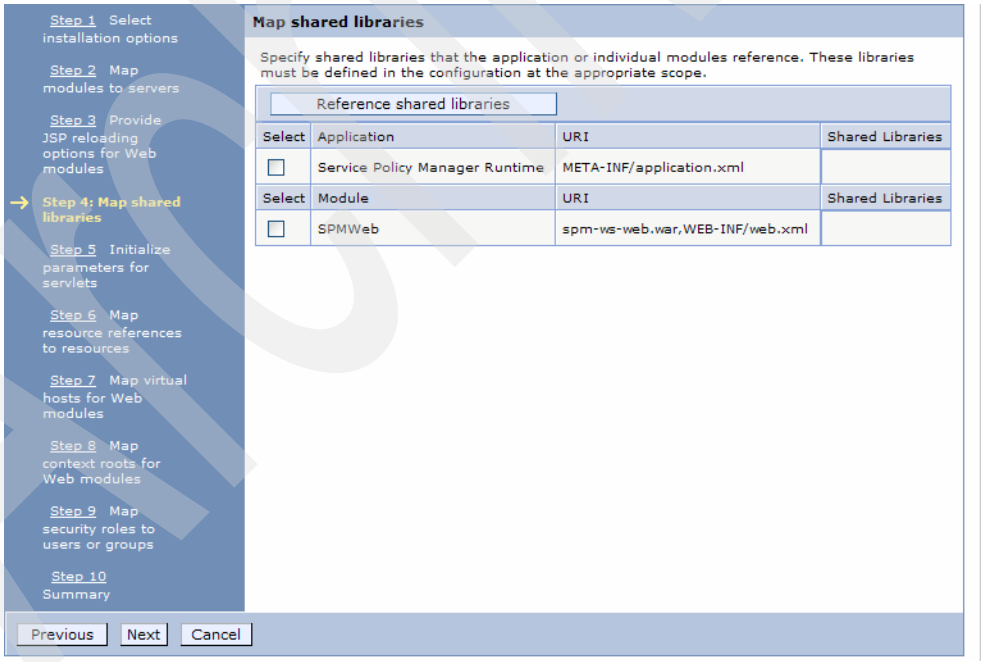


Figure 3-9 Deploy the Service Policy Manager runtime component step 4

9. In the Initialize parameters for servlets window, keep the defaults and click **Next** (Figure 3-10 on page 41).

Archived

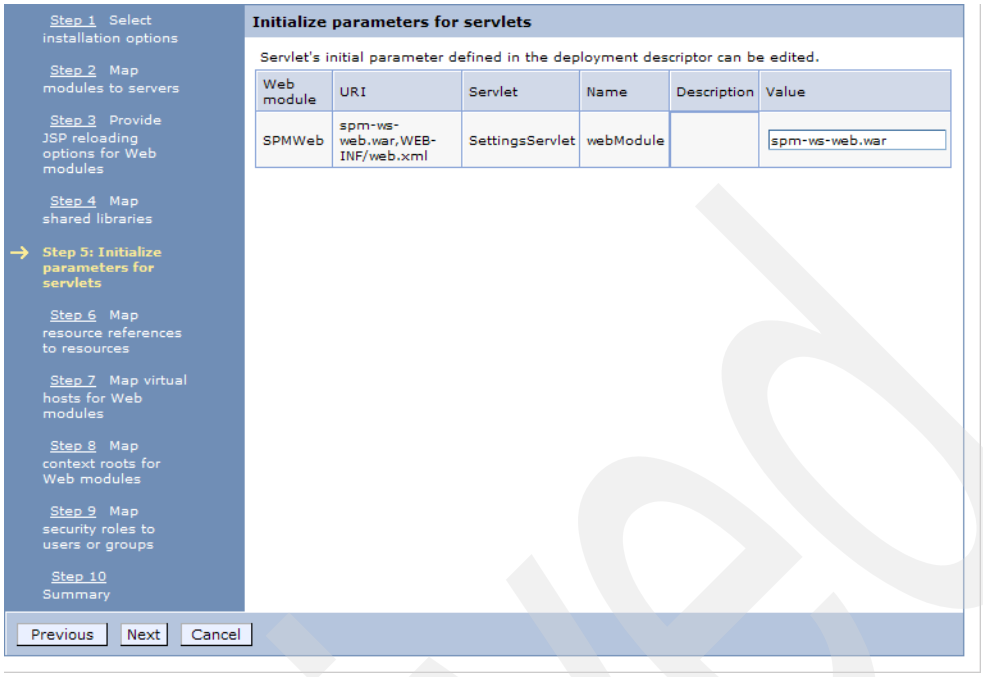


Figure 3-10 Deploy the Service Policy Manager runtime component step 5

10. In the Map resource references to resources window:
    a. Select the **uSe default method** radio button.
    b. Select the **Authentication data entry** from the drop-down list, select the modules, and click the **Apply** button.
    c. Click Next (Figure 3-11).

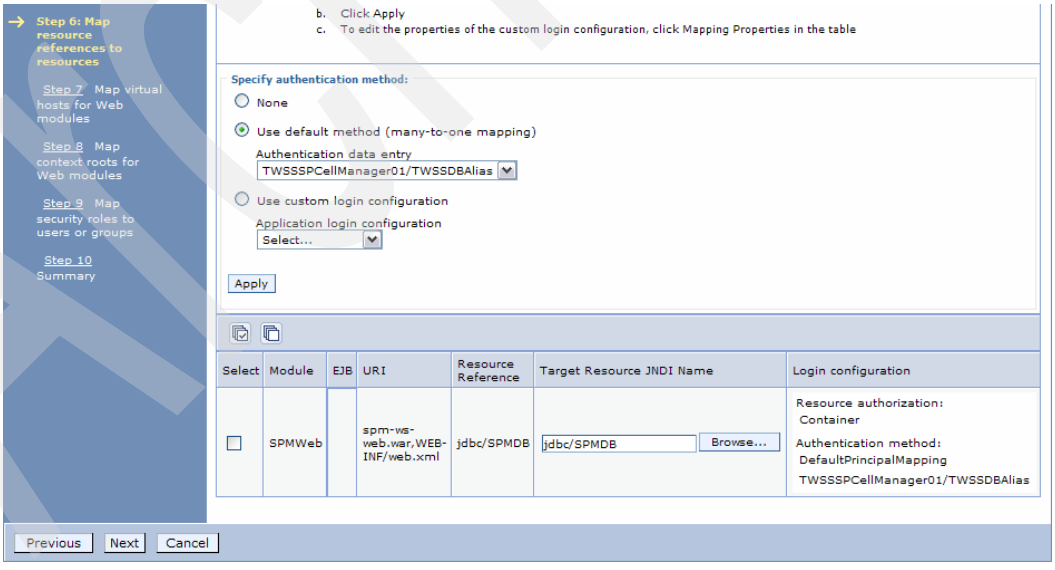


Figure 3-11 Deploy the Service Policy Manager runtime component step 6

Archived

11. In the Map virtual hosts for Web modules window, keep the default values and click **Next** (Figure 12).

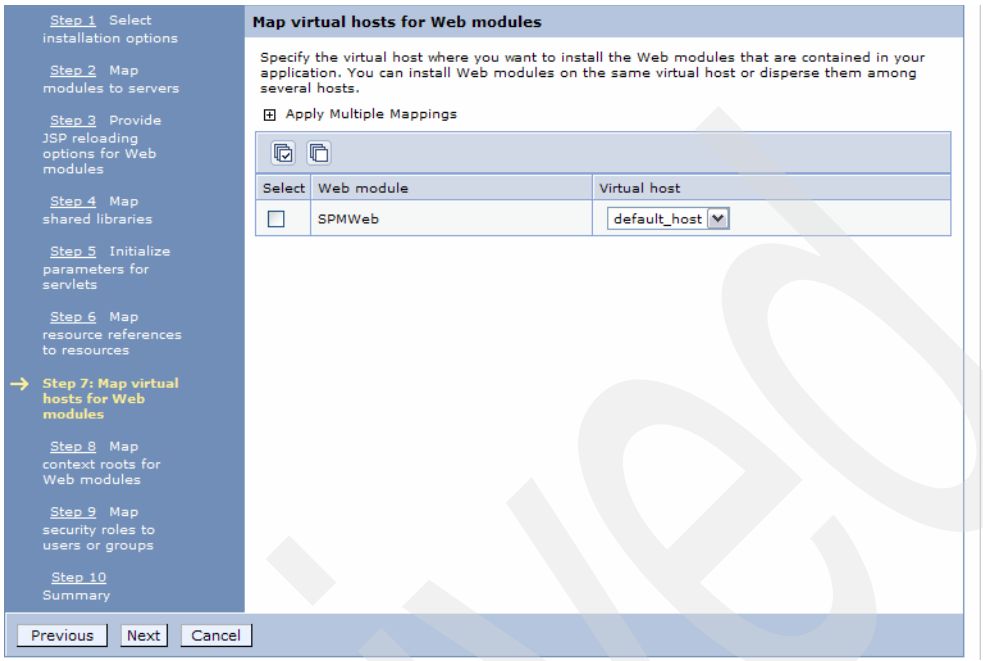


Figure 3-12 Deploy the Service Policy Manager runtime component step 7

12. In the Map context roots for Web modules window, keep the defaults and click **Next** (Figure 3-13).

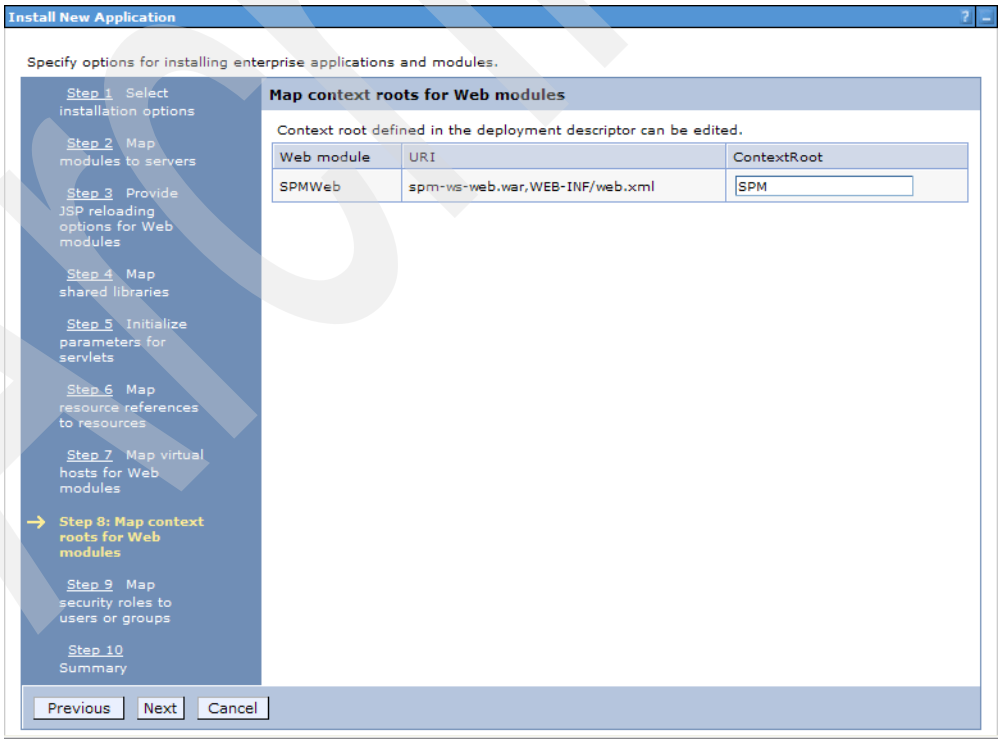


Figure 3-13 Deploy the Service Policy Manager runtime component step 8

Archived

13. In the Map security roles to users or groups window, map the PolicyAdministrator to the desired users, groups, or both. This is the role used to manage requester, service, and policies. On the other hand, you can just select the **Everyone** option for no security (Figure 14 on page 44).

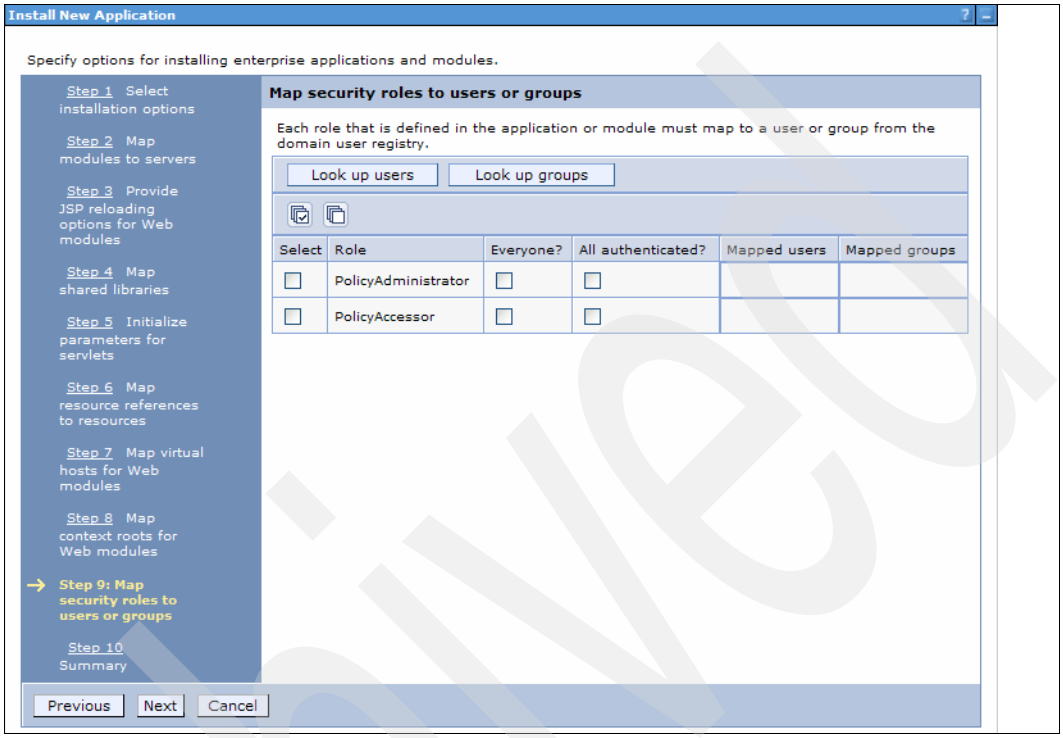


Figure 3-14 Deploy the Service Policy Manager runtime component step 9

Archived

14. Review the summary information and click **Finish** (Figure 3-15).

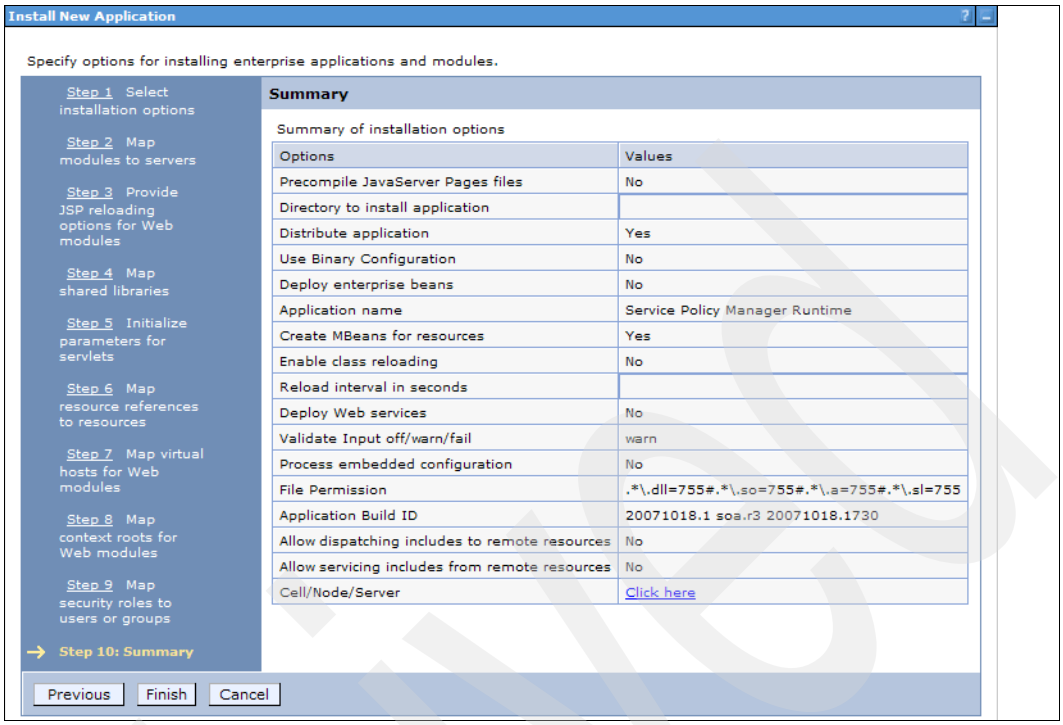


Figure 3-15 Deploy the Service Policy Manager runtime component step 10

15. When finished, click **Save** to apply the configuration (Figure 3-16).

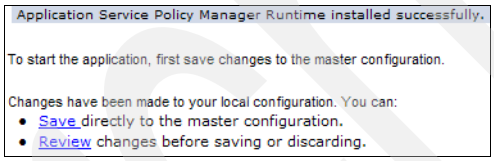


Figure 3-16 Deploy the Service Policy Manager runtime component step 11

3.4.2 Deploying the Service Policy Manager console

The Service Policy Manager console provides a graphical Web interface for using the runtime subcomponent to manage various entities. You can deploy the console either in stand-alone fashion or within a full portal runtime environment. When you deploy it in the cluster environment, you can install and deploy the application on the deployment manager, and then you can synchronize all nodes in the cluster to replicate the application across the cluster. This deployment is similar to the deployment of the Service Policy Manager runtime component:

1. Launch the WebSphere Administrative Console in a Web browser using the address `http://<IP Address> :<Port number>/ibm/console/`. Enter the User ID and click the **Login** button.
2. In the navigation window, select **Applications** → **Install New Applications**.
3. Locate the SPM_Console.EAR file, select **Show me all installation options and parameters**, and click **Next** (Figure 3-17 on page 45).

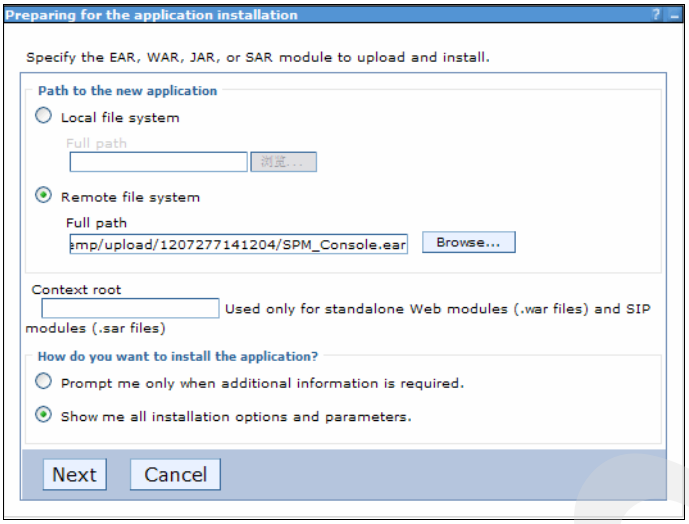


Figure 3-17 Preparing for the application installation

4. In the Application Security Warnings window, click **Continue**.
5. In the Step 1: Select installation options window, keep the defaults and click **Next**.
6. In the Map modules to servers window, map all of the modules to the cluster which you created for deployment of Service Policy Manager applications (Figure 3-18).

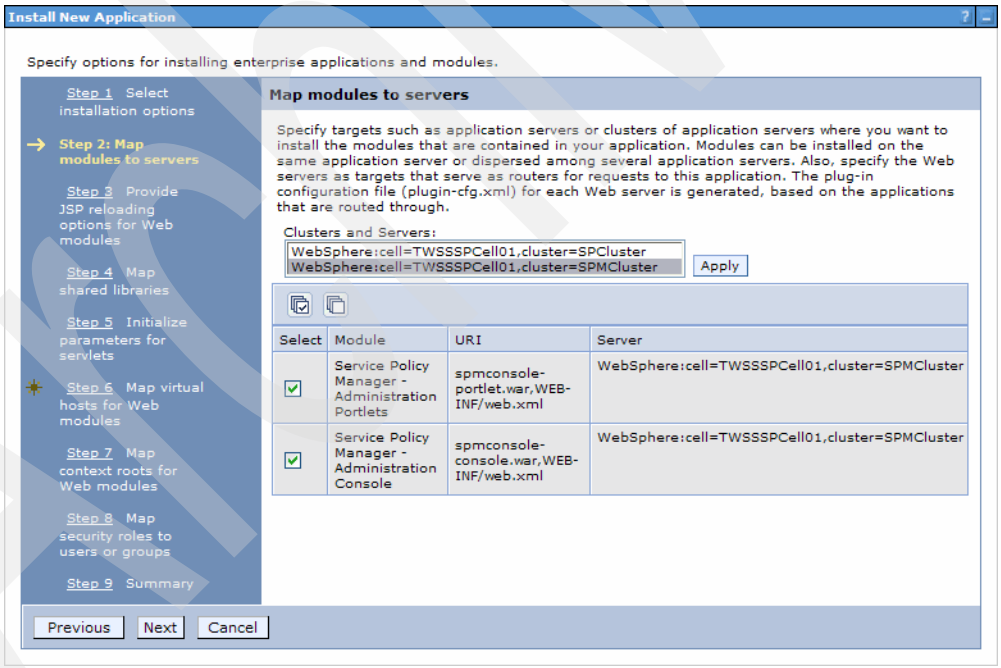


Figure 3-18 Deploy the Service Policy Manager console step 2

7. In the Provide JSP reloading options for Web modules window, keep the defaults and click **Next**.
8. In the Map shared libraries window, keep the defaults and click **Next**.

9. In the prompt Application Resource Warning window, click **Continue** (Figure 3-19).

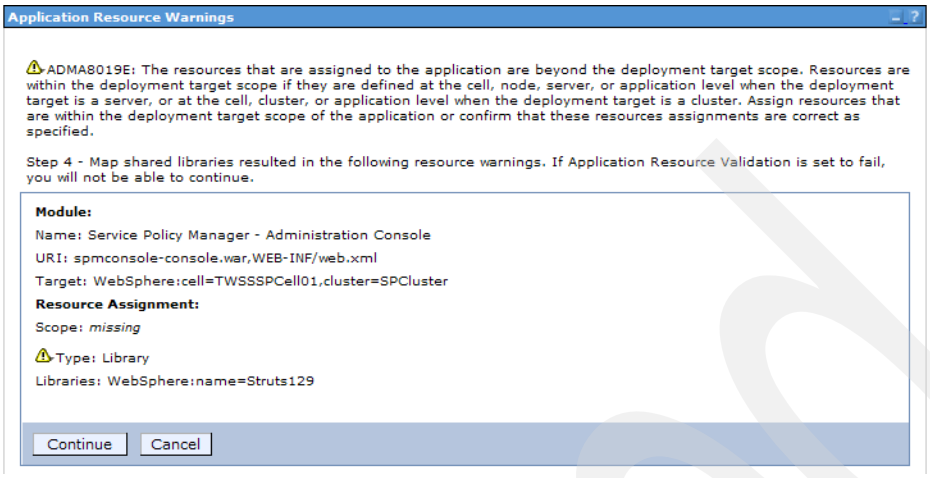


Figure 3-19 Deploy the Service Policy Manager console step 4

10. In the Initialize parameters for servlets window, keep the defaults and click **Next**.
11. In the Map virtual hosts for Web modules window, keep the defaults and click **Next**.
12. In the Map context roots for Web modules window, keep the defaults and click **Next**.
13. In the Map security roles to users or groups page window, map the PolicyAdministrator to the desired users, groups, or both. This is the role used to add policies. On the other hand, you can just select the **Everyone** option for no security (Figure 3-20 on page 47).

    To look up *users*, perform the following steps:

    a. Click the **Look up users** button.

    b. On the prompt page, click **Search** to retrieve all available users.

    c. Select the users and move them into the Selected list.

    d. When done, click **OK** (Figure 3-21).

Archived

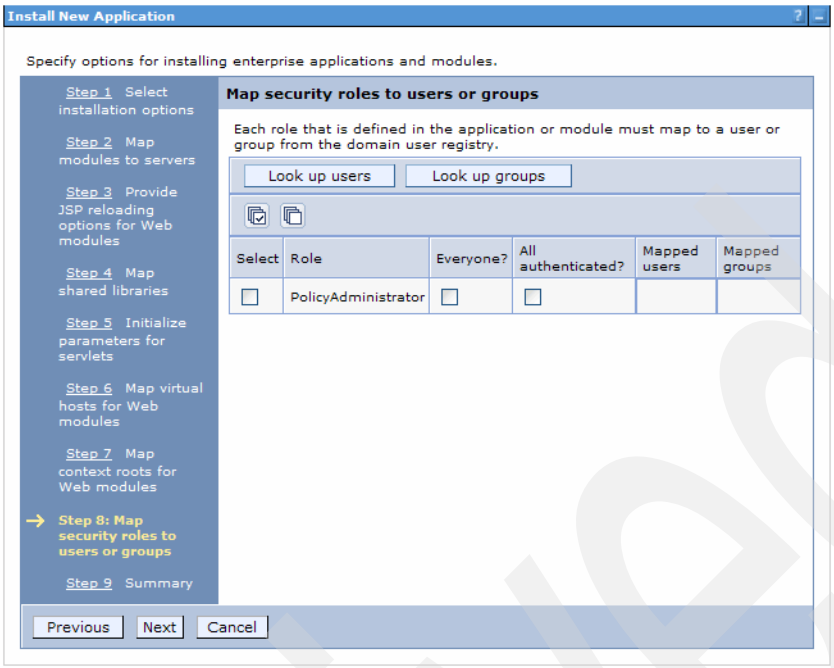


Figure 3-20 Deploy the Service Policy Manager console step 8

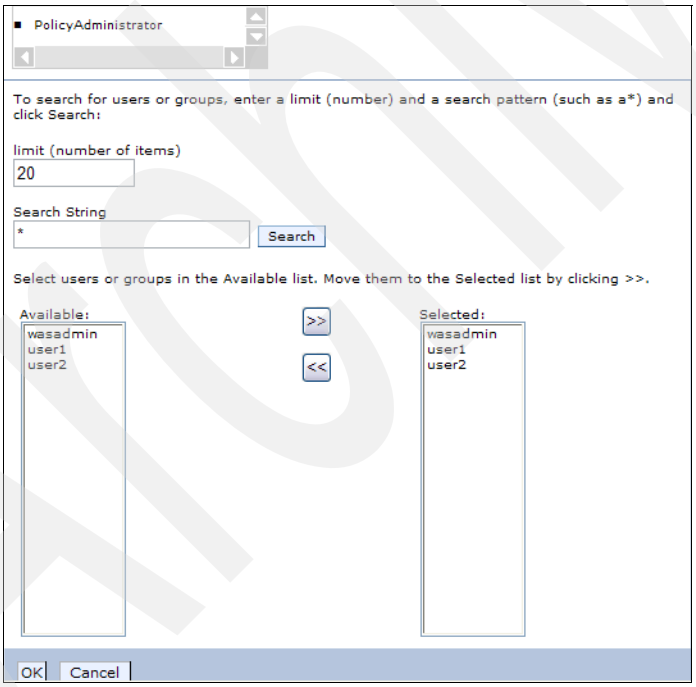


Figure 3-21 Select users for Service Policy Manager

Archived

14. To look up and add groups, perform the following steps;
    a. Click **Look up groups** in the prompt window.
    b. Click **Search** to retrieve all available groups.
    c. Select the groups and move them into the Selected list.
    d. When you are done, click **OK** (Figure 3-22).

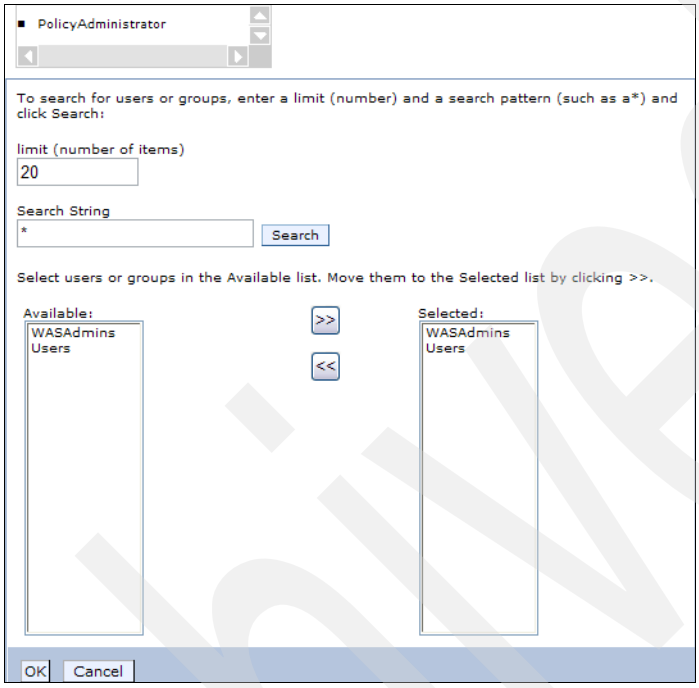


Figure 3-22 Select groups for Service Policy Manager

    e. After you have selected users and groups for the module, check the **All authenticated** check box (Figure 3-23 on page 49).

Archived

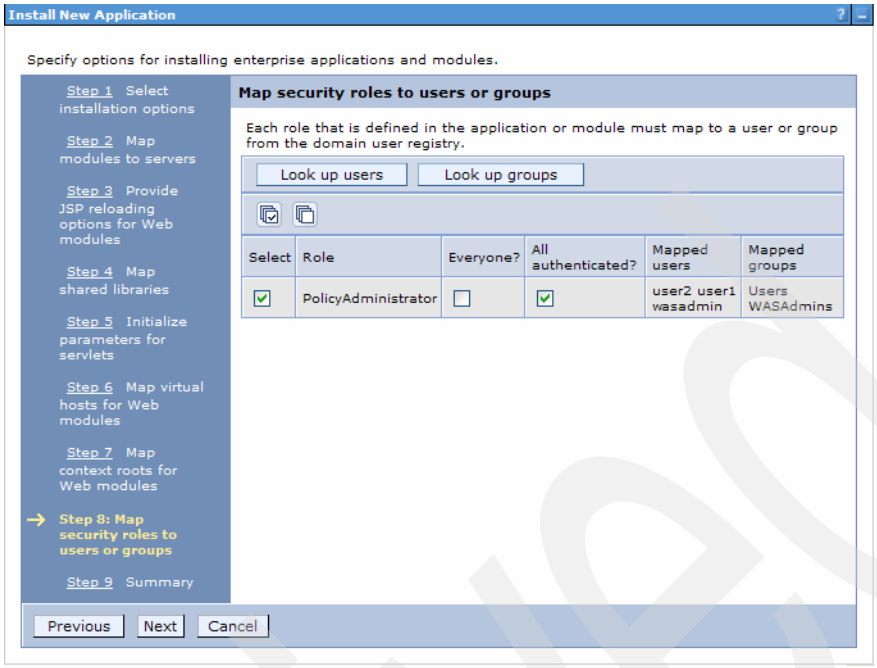


Figure 3-23 Selected users and groups

15. Review the summary information and click **Finish**.
16. When finished, click **Save** to apply the configuration (Figure 3-24).

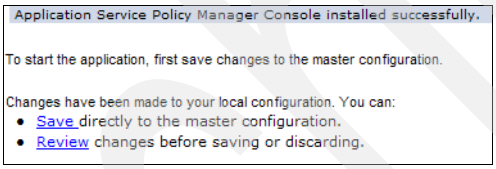


Figure 3-24 Save to master configuration

17. You will see the window shown in Figure 3-25, which indicates a successful save and synchronization.

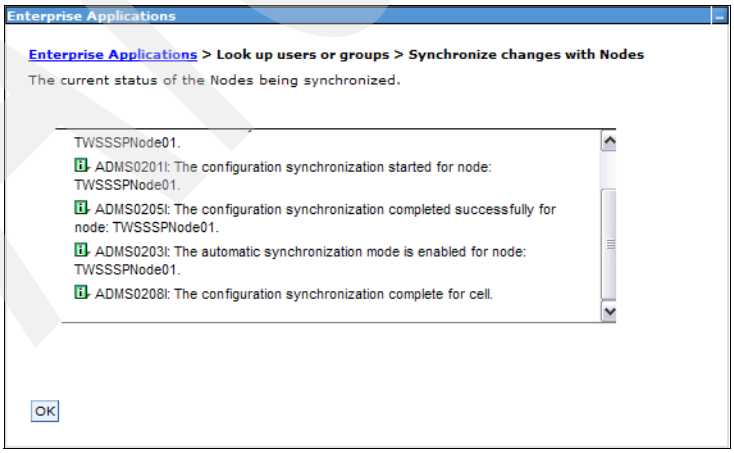


Figure 3-25 Save and synchronization success

Archive

3.4.3 Start the Service Policy Manager applications

After the deployment of Service Policy applications, that, Service Policy Manager runtime component and Service Policy Manager console, you can now use the WebSphere Application Server Administrative Console to start the applications:

1. Launch the WebSphere Administrative Console in the web browser with the address `http://<IP Address> :<Port number>/ibm/console/`. Enter the User ID and click the **Login** button.
2. In the navigation window, select **Applications** → **Enterprise Applications**.
3. Select both Service Policy Manager Console and Service Policy Manager Runtime, then click **Start** (Figure 3-26).

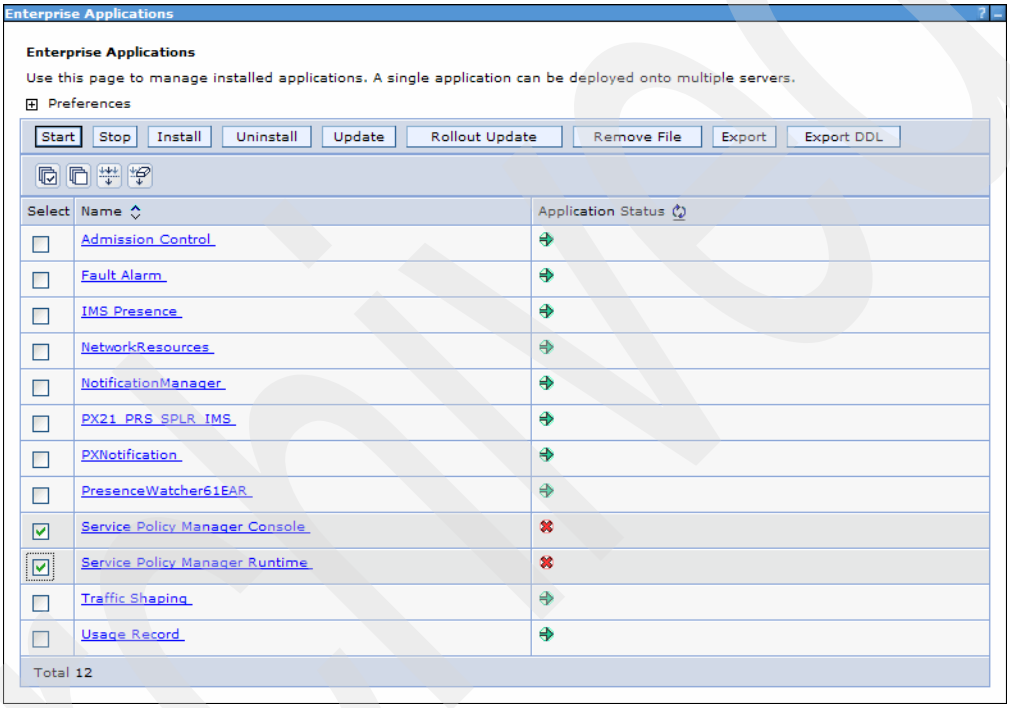


Figure 3-26 Select the applications to start

4. You will get the message indicating that you are successful, similar to the one shown in Figure 3-27.

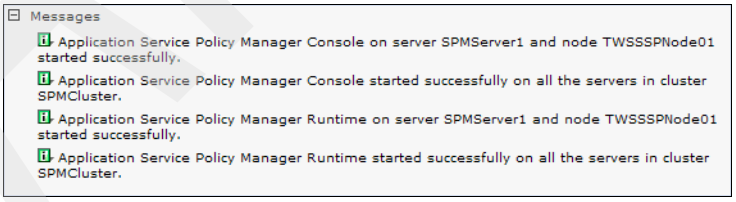


Figure 3-27 Application success started

3.5 Initializing policies

Note about the specific version this section is based on: This IBM Redbooks publication is using IBM WebSphere Telecommunications Web Services Server V6.2 to describe these concepts. It also goes into detail on some user tasks that are overlapping with the InfoCenter, such as the steps for installing SPM EARs, and configuration steps, such as running scripts to initialize policies.

At the time this IBM Redbooks publication was written, IBM WebSphere Telecommunications Web Services Server V6.3 was also close to completion, but had not yet been formally released. All the concepts and interfaces are consistent, however, and some installation and configuration steps have been improved in V6.3. Therefore, some of these details are changed to match the context of V6.3. Refer to the InfoCenter for the latest and most complete installation and configuration steps associated with the version that you are using.

You can refer to the InfoCenter at the following URL:

`http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp`

However, for the screen captures in this IBM Redbooks publication, and for the sake of readability, we specifically used IBM WebSphere Telecommunications Web Services Server V6.2. We have included a description of the configuration steps necessary for the use case.

After deploying the Service Policy Manager applications, that is, Service Policy Manager runtime component and Service Policy Manager console, you can run scripts to initialize policies for the Access Gateway and for each of the Web service implementations you plan to deploy.

3.5.1 Initialize the basic policies

You need to initialize the base policies so that the Access Gateway can get them to communicate with back-end service implementations. The base policy scripts for Access Gateway should be located under <install root>/installableApps/TWSS-Base/scripts directory.

Note: The Service Policy Manager applications, that is, Service Policy Manager runtime component and Service Policy Manager console, should be running when you run the policy scripts.

1. Log in to the host where the policy script has been installed.
2. Change the directory to <WAS ROOT>/bin.
3. Initialize the first default policies by running the following command:

```
./wsadmin.sh -lang jython -f <WAS
ROOT)/installableApps/TWSS-Base/scripts/ag_pol.py
```

4. Check the message on the screen, making sure the initialization is successful:

```
WASX7209I: Connected to process "dmgr" on node twss01CellManager01 using SOAP
connector;  The type of process is: DeploymentManager
*sys-package-mgr*: processing new jar,
'/opt/IBM/WebSphere/AppServer/lib/startup.jar'
```

```
*sys-package-mgr*: processing new jar,
'/opt/IBM/WebSphere/AppServer/lib/bootstrap.jar'
----------------------------------------
[req=ALL,svc=ALL,op=ALL,name=message.sla.LocalServiceRate,value=0,type=Numeric]
-> Success
[req=ALL,svc=ALL,op=ALL,name=message.sla.LocalOperationRate,value=0,type=Numeri
c] -> Success
```

5. Initialize other default policies by running the following command:

```
./wsadmin.sh -lang jython -f <WAS
ROOT)/installableApps/TWSS-Base/scripts/ag_px21_services.py
WASX7209I: Connected to process "dmgr" on node linuxCellManager01 using SOAP
connector;  The type of process is: DeploymentManager
[Fri Nov 23 14:59:47 2007] SOAC7528I: Operation definition successfully
created.
[Mon Mar 03 15:20:24 2008] SOAC7528I: Operation definition successfully
created.
[Mon Mar 03 15:20:24 2008] SOAC7527I: Defined new operation:
service=[http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface
],operation=notifySubscription,enabled=True,description=[]
[Mon Mar 03 15:20:24 2008] SOAC7528I: Operation definition successfully
created.
[Mon Mar 03 15:20:24 2008] SOAC7527I: Defined new operation:
service=[http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface
],operation=subscriptionEnded,enabled=True,description=[]
[Mon Mar 03 15:20:24 2008] SOAC7528I: Operation definition successfully
created.
```

Note: The policy script can be run many times. In the case where the policy already exists, you will get a message stating that the `policy already exists` on the second or further successful initialization.

3.5.2 Initialize the additional policies for specified Web service implementations

The IBM WebSphere Telecommunications Web Services Server product supports a large number of Web service implementations, many of which are based on Parlay X specifications. At this point, we will use “Parlay X Presence over SIP/IMS” for our demonstration.

1. Got to the IBM WebSphere Telecommunications Web Services Server information center at `http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp` for the list of Policy scripts for the Web service implementations, as shown in Figure 3-28 on page 53.

Table 1. Policy scripts for the Web service implementations

Web service	Script name
Parlay X Call Notification over SIP/IMS	px21_cn_ims_pol.py
Parlay X Presence over SIP/IMS	px21_prs_ims_pol.py
Parlay X Terminal Status over SIP/IMS	px21_ts_ims_pol.py
Parlay X Third Party Call over SIP/IMS	px21_tpc_ims_pol.py
Parlay X Payment over Usage Records/CEI	px21_pmt_postpaid_pol.py
Parlay X Address List Manager over XCAP	px21_grp_ims_pol.py
WAP Push over SMPP	wap10_push_smpp_pol.py
Parlay X SMS over SMPP	px21_sms_smpp_pol.py
Parlay X Terminal Location over MLP	px21_tl_mlp_pol.py
Parlay X Multimedia Messaging over MM7	px21_mm_mm7_pol.py
Parlay X Terminal Status over Parlay	px21_ts_parlay_pol.py
Parlay X Terminal Location over Parlay	px21_tl_parlay_pol.py
Parlay X SMS over Parlay	px21_sms_parlay_pol.py

Figure 3-28 Policy script list for service implementation

2. Initialize the default policies for Parlay X Presence over SIP/IMS:

   ```
   ./wsadmin.sh -lang jython -f <WAS
   ROOT)/installableApps/TWSS-Service/scripts/px21_prs_ims_pol.py
   ```

3. Check the output message and make sure the initialization is successful.

3.6 Creating a new policy

Policy information in the incoming request may override the default policy configuration. If the policies' information in the header is more restrictive than the policies stored in the Service Policy Manager, the policy information in the header will override the stored policies.

3.6.1 Create default policy

The IBM WebSphere Telecommunications Web Services Server includes default policies for Access Gateway, and many of the Web service implementations sets. You initialize the default policies by running scripts after you install the Service Policy Manager. Once initialized, you should configure the default policies and set the correct value for each policy.

1. Reference to the IBM WebSphere Telecommunications Web Services Server InfoCenter at http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp for the default policies (see Figure 3-29 on page 54).

Archive

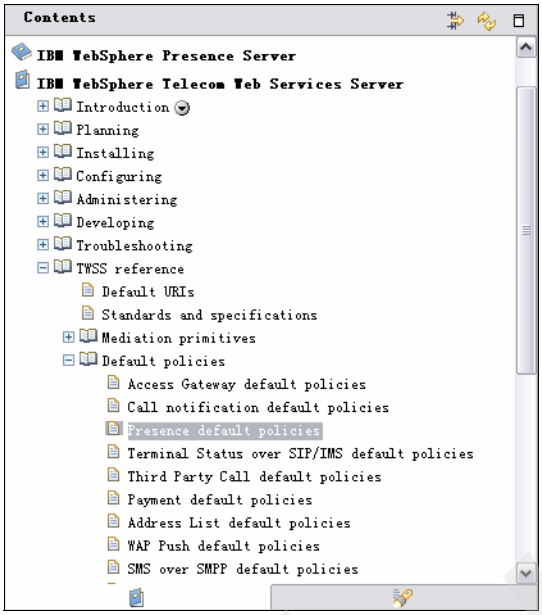


Figure 3-29 IBM WebSphere Telecommunications Web Services Server reference for default policies

2. Launch the Service Policy Manager Console in the web browser using the address `http://<IP Address>:<Port number>/spm/console/`. Enter the user ID and click the **Login** button (Figure 3-30).

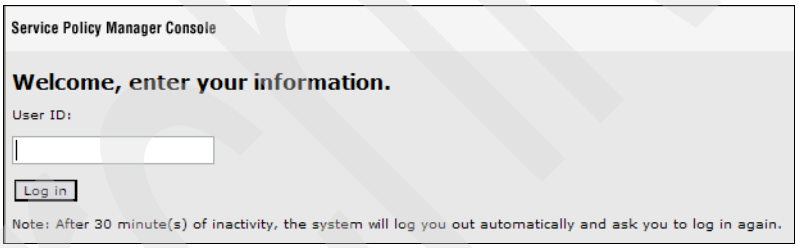


Figure 3-30 Service Policy Manager Console launch window

3. In the Service Policy Manager connection details window (see Figure 3-31 on page 55):
   a. Select the connection URL.
   b. Input the password.
   c. Select the connection timeout time from the drop-down list box.
   d. Click **Connect**.

Archived

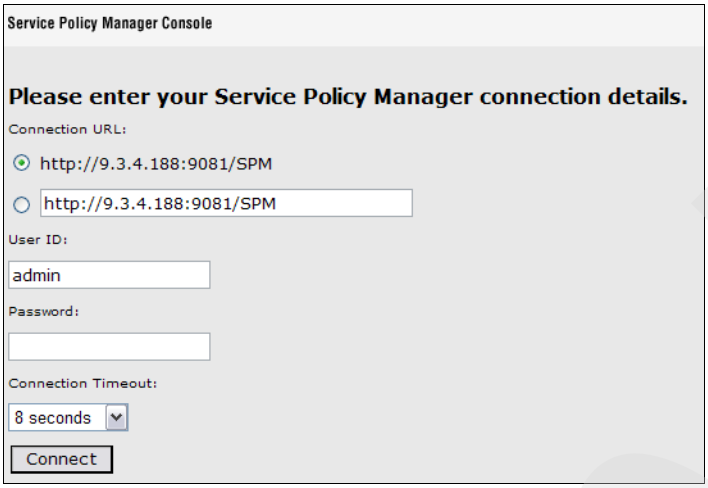


Figure 3-31 Service Policy Manager Console connect window

4. The Service Policy Manager console Welcome window opens (see Figure 3-32).

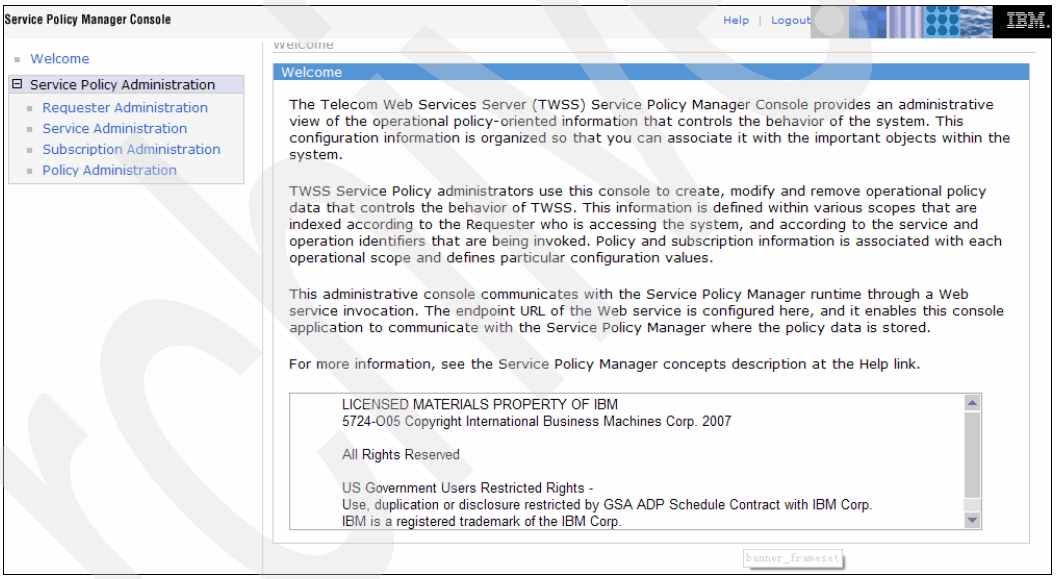


Figure 3-32 Service Policy Manager Console Welcome window

Archived

5. In the navigation window, click **Policy Administration** (see Figure 3-33).

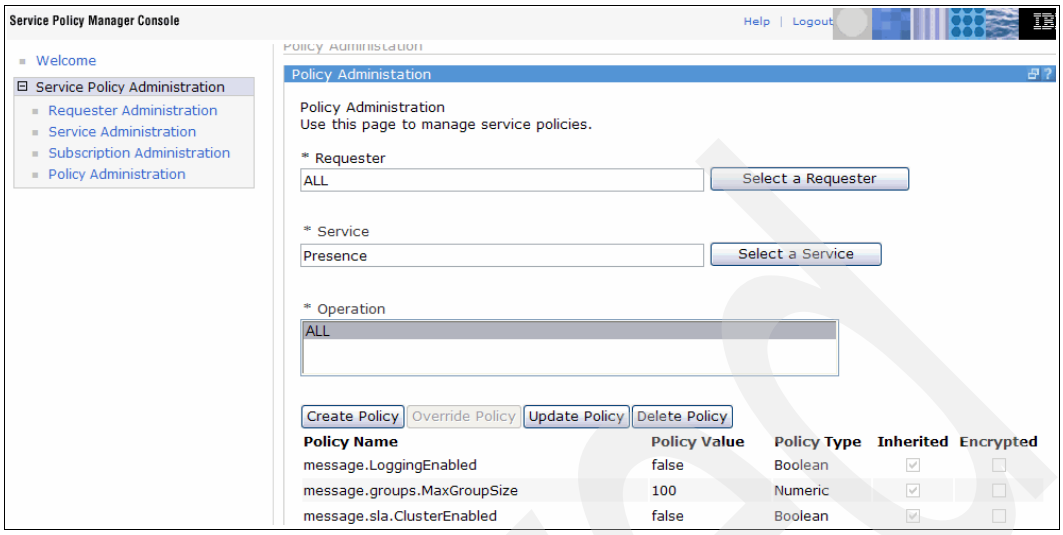


Figure 3-33 Policy Administration window

6. In the Policy Administration window (see Figure 3-34):
   a. Click the **Select a Service** command button.
   b. Expand **Parlay X** in the service tree view.
   c. Select the Presence.
   d. Select the interface in the Type Service listview.
   e. Click **Create Policy**.

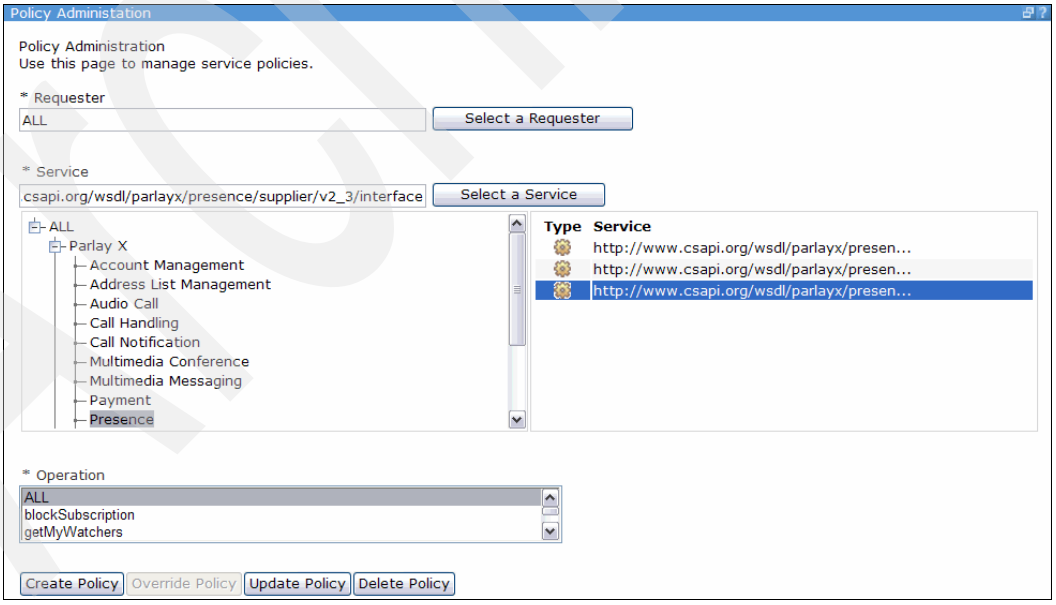


Figure 3-34 Policy Administration window for presence services

7. In the Create Policy window (see Figure 3-35):
   a. Input the Policy Name.
   b. Select the Policy Type.
   c. Input the Policy Value.
   d. Click **Create Policy**.

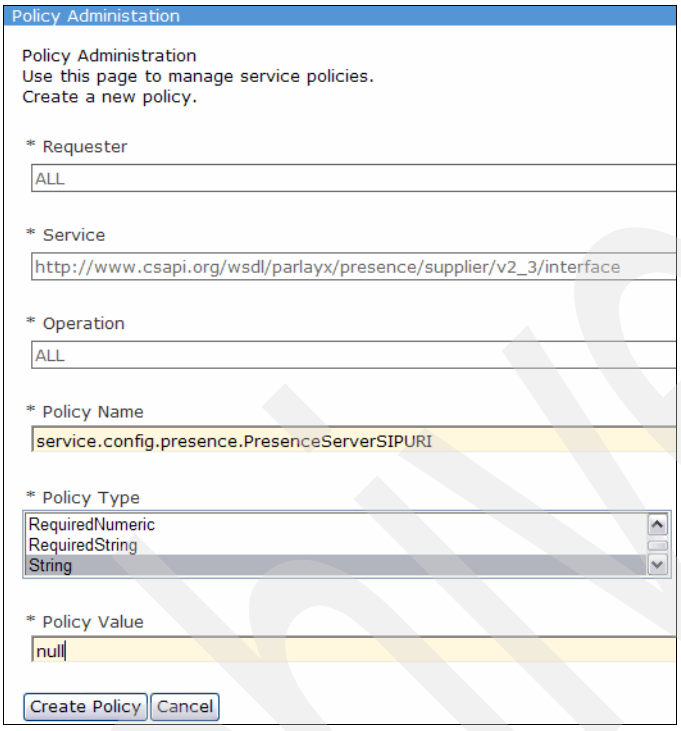
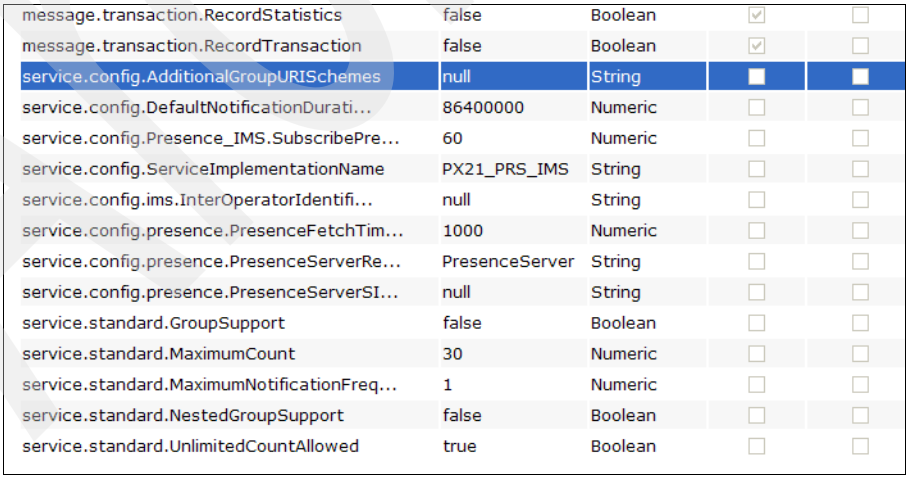


Figure 3-35 Create new policy window

8. Repeat steps 6 through 7 to create all the required default policies (see Figure 3-36).



message.transaction.RecordStatistics	false	Boolean	☑	☐
message.transaction.RecordTransaction	false	Boolean	☑	☐
service.config.AdditionalGroupURISchemes	null	String	☐	☐
service.config.DefaultNotificationDurati...	86400000	Numeric	☐	☐
service.config.Presence_IMS.SubscribePre...	60	Numeric	☐	☐
service.config.ServiceImplementationName	PX21_PRS_IMS	String	☐	☐
service.config.ims.InterOperatorIdentifi...	null	String	☐	☐
service.config.presence.PresenceFetchTim...	1000	Numeric	☐	☐
service.config.presence.PresenceServerRe...	PresenceServer	String	☐	☐
service.config.presence.PresenceServerSI...	null	String	☐	☐
service.standard.GroupSupport	false	Boolean	☐	☐
service.standard.MaximumCount	30	Numeric	☐	☐
service.standard.MaximumNotificationFreq...	1	Numeric	☐	☐
service.standard.NestedGroupSupport	false	Boolean	☐	☐
service.standard.UnlimitedCountAllowed	true	Boolean	☐	☐

Figure 3-36 Default Presence policies

Archived

Important: To configure the Access Gateway to invoke Web service implementations, a policy (service.Endpoint) must be created for the specific service on the Service Policy Manager.

This policy must be created for all requesters, for all operations, and for a specific service. You can define it for both all and unauthenticated requesters, but it needs to be set in the scope of each Web service implementation.

3.6.2 Enabling a new requestor for IBM WebSphere Telecommunications Web Services Server

In order to enable a new requestor for IBM WebSphere Telecommunications Web Services Server, you need to first to create a new requestor and a new subscription. Both processes are defined in these next sections, within the context of creating the custom policy.

The relationship between a *requester* and a *service* or an *operation* is managed through a *subscription*. The subscription is Policy Manager's authorization mechanism to grant a requester the permission to access and personalize a service in the form of policies. A requester must subscribe to the service before a policy can be added to the service by the requester.

A requester ID must be made known to the WebSphere user registry (LDAP or other) so that it is a known identity that can be authenticated when security is enabled. Additionally, there are some roles that the user should be mapped to. Typically, you should set up a group in the user registry for all client requesters so that roles, such as PolicyAccessor, can be mapped to it. Then, when adding a new user, the action can be done as a member of the group. Creating a similar requester group in the SPM Console, under authenticated, and then putting user1 into that group, is desirable.

3.6.3 Creating the custom policy

Policies can be defined at different scopes:

- Global
- Service-specific
- Operation-specific
- Requester-service specific
- Requester-specific scopes

Policies defined in high level scope can be overridden by the same policy defined in a lower level.

Create new requester

To create a new requester, do the following steps:

1. Log in to the Service Policy Manager Console and, in the navigation window, click **Requester Administration** (Figure 3-37 on page 59).

Figure 3-37 Requester Administration window

2. In the **Requester Administration** window, click **Create Requester**.
3. In the Create Requester window, select the requester type from the Type drop-down list box, select the Parent for the new requester, input the Request name, input the Description, check the **Enabled** check box, and click **Create Requester** (Figure 3-38).

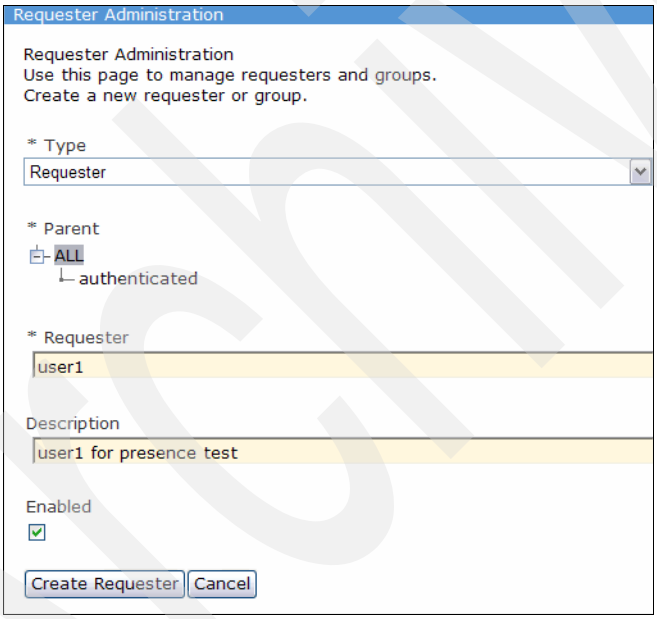


Figure 3-38 Create new requester

Archived

4. Repeat step 2 through 3 to add all the users that are required (Figure 3-39).

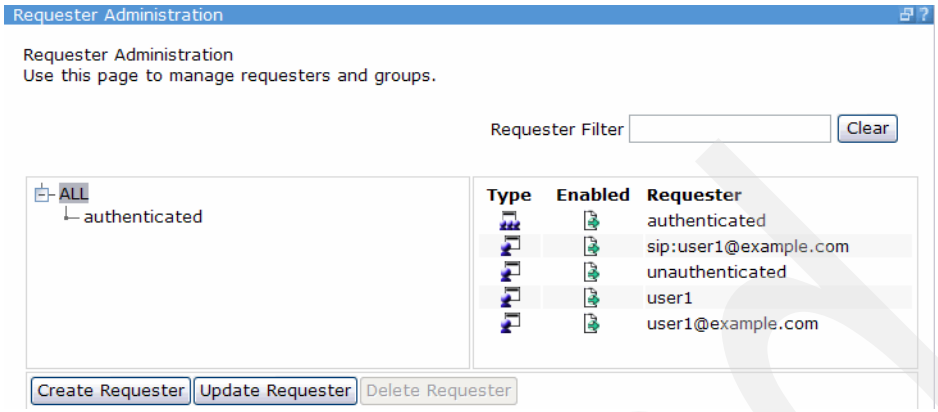


Figure 3-39 List of all requesters

Create new subscription

A subscription is a relationship between a requester and a service or an operation. The subscription is Policy Manager's authorization mechanism to grant a requester the permission to access and personalize a service in the form of policies. A requester must subscribe to the service before a policy can be added to the service by the requester. Do the following steps to create a new subscription:

1. Log in the Service Policy Manager Console, and in the navigation window, click **Subscription Administration**.
2. In the Subscription Administration window, select the requester from the Requester tree, and click **Create Subscription** (Figure 3-40).

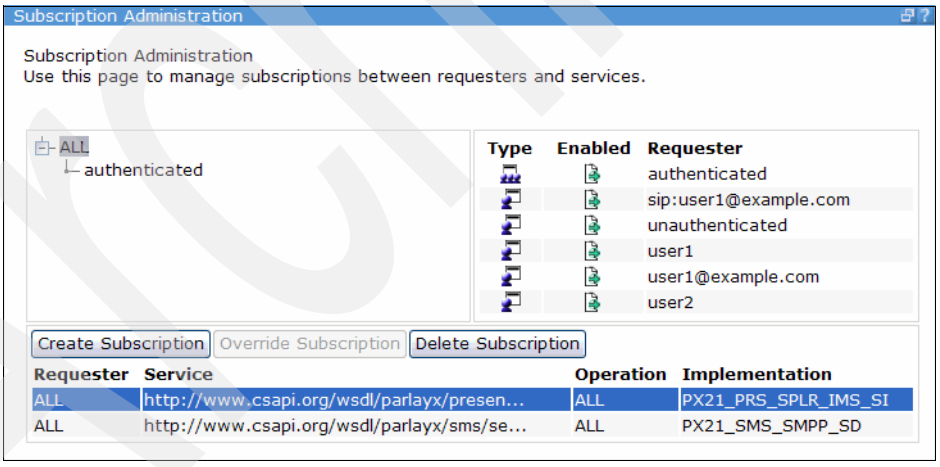


Figure 3-40 Subscription Administration window

3. In the create subscription window, click the **Select a Service** button, select the service in the service tree, select the interface from the Type Service list view, select the operation from the Operation list view, check that the Implementation is correct, and click **Create Subscription** (Figure 3-41 on page 61).

Archive

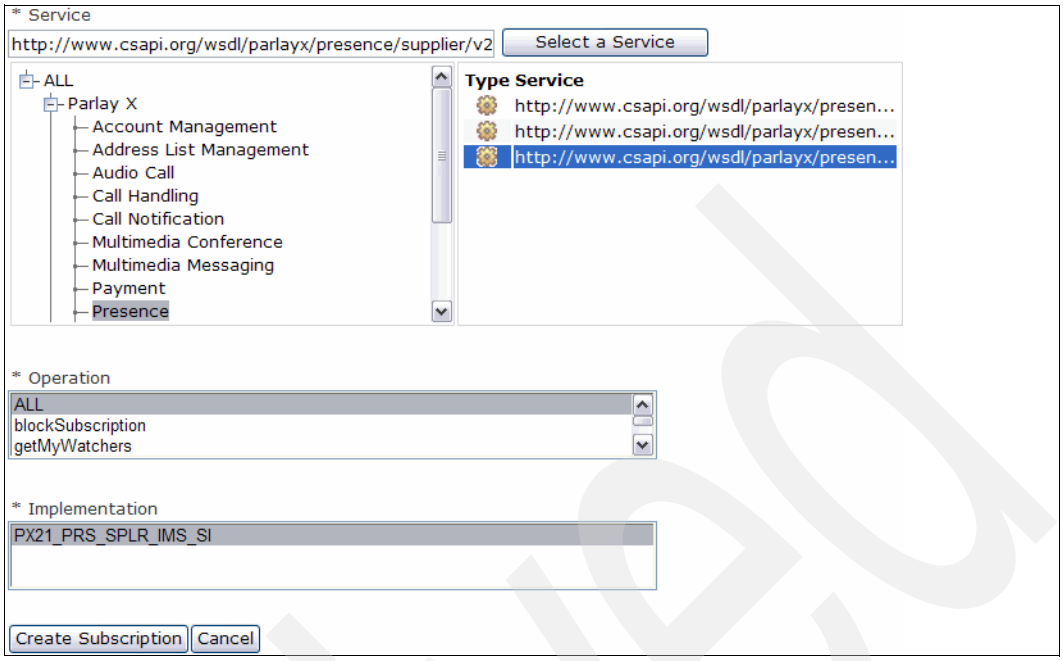


Figure 3-41 Create new subscription

3.7 Sample for Service Policy Manager - SIP addressing conversion

In this section, we discuss how to create a sample service policy for SIP addressing conversion. First, we describe the use case and proposed functionality for the service policy, and then give step-by-step instructions on how to create the policy.

3.7.1 Use case description for new policy to be created

Policy limits apply to a three-level hierarchy:

- Requester
- Service
- Operation

Requester limits govern all traffic from a specific requester across all services. Service limits constrain the rate of requests for all operations executed against that service, regardless of the distribution of requests to the individual operations.

Through policy attributes, policy information contains dynamic runtime configuration information that is used as input for decision logic during service execution. Policy information allows personalization of service execution because policy information can be resolved uniquely for each (requester, service, and operation) tuple. The policy configuration information required is unique to each service implementation and is chosen at design time. Each service implementation should uniquely identify its service name using the convention (service name)_(category/flavor) when defining new policies.

If we had a situation where we needed to transform the format of an URI during a Web service request, we may decide to do that transformation in an Access Gateway mediation primitive, and utilize a policy to control its operation.

Archived

In this case, we create a specific policy to control the transformation:

1. We create two requesters, in this case, user1, sip:user1@example.com, and so on
2. We create a Boolean policy, that is, service.config.enableURItransformation, to publish the operation of presence service.

When the client mediation flow retrieves the TRUE value of the policy, it will change the soap message content, that is SOAPHeader → requesterID, and transform the requesterID from user1 to sip:user1@example.com, which is the SIP standard format.

Figure 3-42 illustrates this conversion.

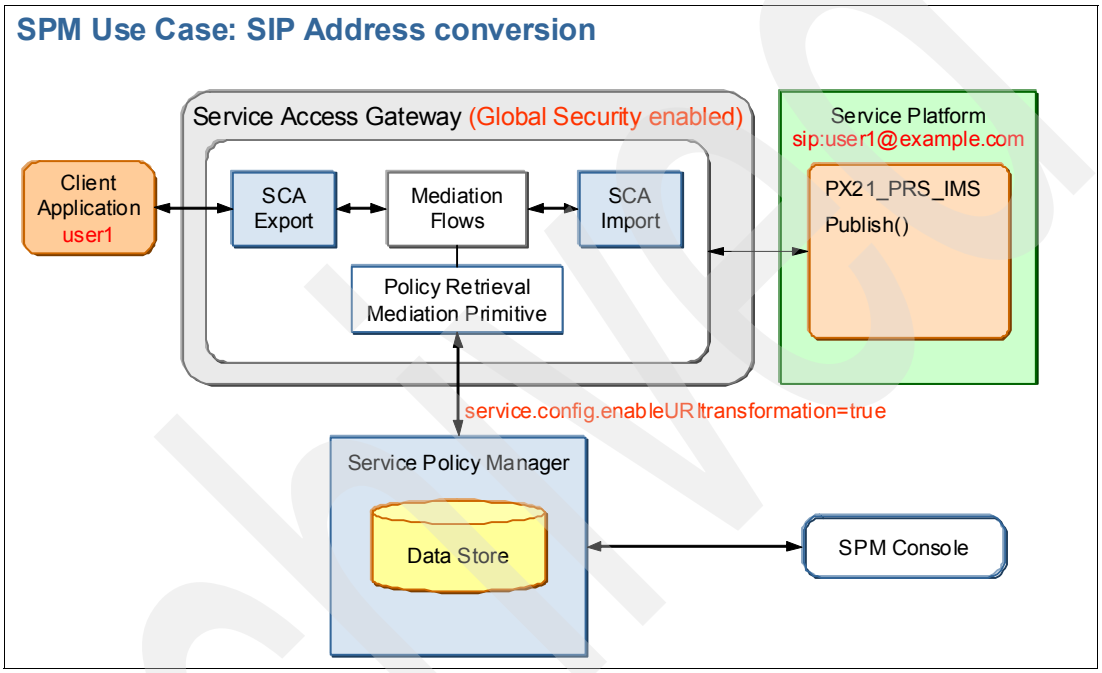


Figure 3-42 Service Policy Manager Use Case

3.7.2 Use case realization

As described previously, this sample SIP Addressing conversion requires us to create two requesters and a policy on Service Policy Manager console. To do these tasks, do the following steps:

1. Log in the Service Policy Manager Console, and in the navigation winnow, click **Requester Administration**.
2. In the Requester Administration window (Figure 3-43 on page 63), click **Create Requester**.
3. In the Create Requester window:
   a. Select **Requester** from the Type drop-down list box,
   b. Select **All** from the Parent listview,
   c. Input “user1” as the Request name,
   d. Input the description.
   e. Check the Enabled check box.
   f. Click **Create Requester**.

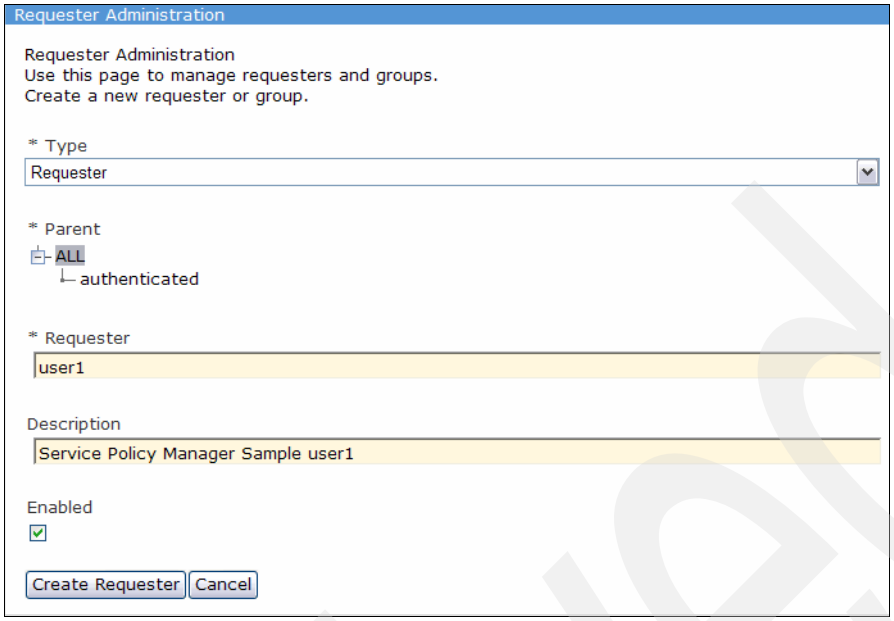


Figure 3-43 Create sample user1

4. In the Requester Administration window, click **Create Requester** again.
5. In the **Create Requester** window (Figure 3-44):
   a. Select **Requester** from the Type drop-down list box.
   b. Select **All** from the Parent listview.
   c. Input “sip:user1@example.com” as the request name.
   d. Input the description.
   e. Check the **Enabled** check box.
   f. Click **Create Requester**.

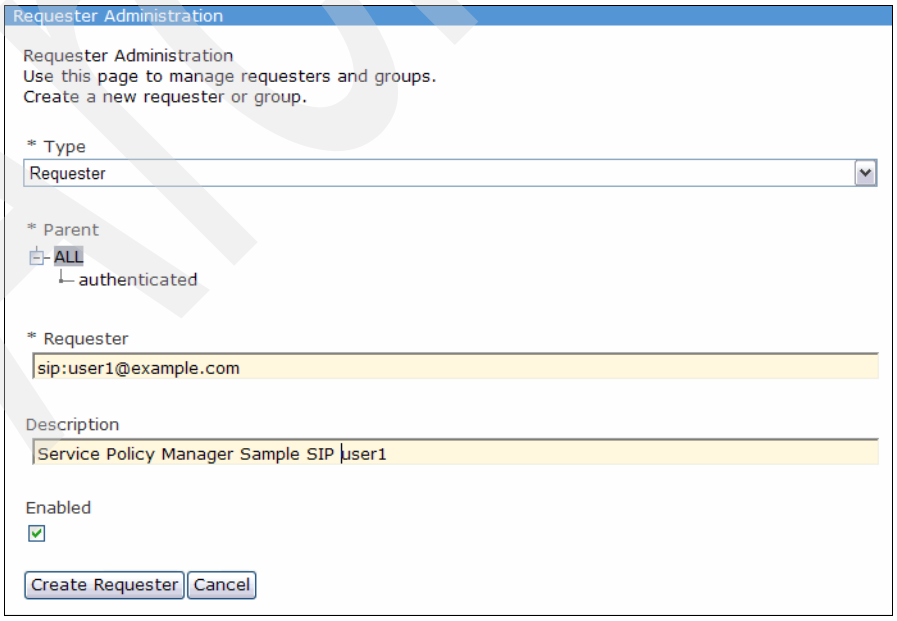


Figure 3-44 Create sample SIP user1

6. In the navigation window, click **Policy Administration**.
7. In the **Policy Administration** window:
   a. Click the **Select a Service** button.
   b. Expand **Parlay X** in the service tree.
   c. Select **Presence**.
   d. Select the supplier interface in the Type Service listview.
   e. Select **Publish** in the Operation box.
   f. Click **Create Policy**.
8. In the create policy window (Figure 3-45):
   a. Input service.config.enableURItransformation as the Policy Name.
   b. Select **Boolean** from the Policy Type box.
   c. Input false as the Policy Value,
   d. Click **Create Policy**.

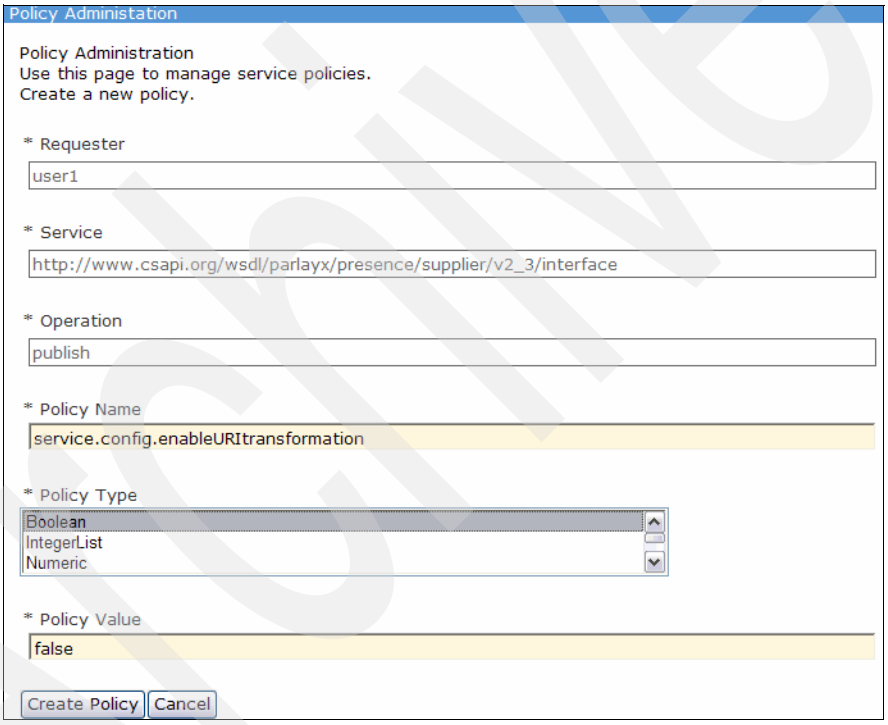


Figure 3-45 Create sample policy for the specify operation

9. Congratulations, you have finished the policy settings for the sample. You can now check that the requester and policy that you created are correct.

3.7.3 Test the sample

To test the sample, we need the application running on the Access Gateway to retrieve the policy and make the change in the SOAP message.

Archived

The test the application through Access Gateway, do the following steps:

1. Enable Global Security and add “user1” to the user registry.
2. Open the trace so that we can receive the detailed SOAP message.
3. Deploy the client side media flow application, which should make the change in the SOAP message.

After a successful test, you should get the following message from the trace.log file, which indicates that the "service.config.enableURItransformation" policy value is true. The requesterID will be changed from `user1` to `sip:user1@example.com`.

```
<SOAPHeader>
        <requesterID>user1</requesterID>
        <policies>
          <policy attribute="service.config.enableURItransformation"
value="true"/>
        </policies>
      </value>
    </SOAPHeader>
```

After it is modified by the mediation flow, the requester ID has been changed as follows:

```
<SOAPHeader>
        <requesterID>sip:user1@example.com</requesterID>
        <policies>
          <policy attribute="service.config.enableURItransformation"
value="true"/>
        </policies>
      </value>
    </SOAPHeader>
```

We have not actually created a sample here, but only defined a policy, run an existing flow, and utilized trace to verify that the policy is actually made available to the flow. The custom flow that extends this use case is defined in Chapter 4, “Design considerations for Access Gateway flows - base mediation flows and the mediation primitives” on page 67 and illustrated in Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127.

Archived

Archived

4

Design considerations for Access Gateway flows - base mediation flows and the mediation primitives

IBM WebSphere Telecommunications Web Services Server Access Gateway, built on WebSphere Enterprise Service Bus, provides flexibility for tailored/customized message processing logic.

- Mediation primitives are small, goal driven pieces of message processing logic that can be combined and rearranged in different configurations.
- A default flow or “wiring” of the mediation primitives is provided for out-of-the box implementations The default message processing flow for the Access Gateway makes use of the various mediation primitives in a set sequence.

IBM WebSphere Telecommunications Web Services Server Access Gateway logic can be customized according to service provider network policies through tailoring of ESB mediation flows. WebSphere Integration Developer tooling can be used to create customized mediation flows. Ultimately, this reduces service creation and deployment time while lowering the necessary skill level for service development.

In this chapter, we provide an overview of the default mediation flow provided with the IBM WebSphere Telecommunications Web Services Server V6.2 and discuss each of the default mediation primitives included. We also discuss how to install the WebSphere Integration Developer tooling required to begin working with the default mediation flow in the Access Gateway.

Once the developer understands how to work with the *default* mediation flow, we discuss how to proceed with *customizations* in Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127.

© Copyright IBM Corp. 2008. All rights reserved.

4.1 Introduction to Access Gateway and mediation flows

Telecom Web Services Access Gateway acts as a message processing intermediary. All Web service requests and responses pass through Telecom Web Services Access Gateway. Telecom Web Services Access Gateway consists of a telecom-specific function added on top of the WebSphere ESB platform. The telecom function is provided in the form of mediation primitives that can be used to construct mediation flows containing message processing logic.

The execution of message processing logic within WebSphere ESB is orchestrated by a mediation flow module. The flow definition is produced within the WebSphere Integration Developer tooling.

IBM WebSphere Telecommunications Web Services Server includes a set of default flows, one for each Parlay X interface. It is a model of a typical service provider processing function. Each default flow invokes the mediation primitives provided with the Access Gateway in a set sequence. Each default flow is designed to support the accounting of requests, service or operation level authorization, message capture for regulatory purposes, and traffic level enforcement.

Figure 4-1 on page 69 illustrates the concept of how a mediation flow serves to support and manage requests, authorization, traffic level enforcement, and logging when required. Note that WebSphere Integration Developer is for creating and editing the flows. The flows are actually *managed* through the administration console by editing the SCA Module Properties.

Archived

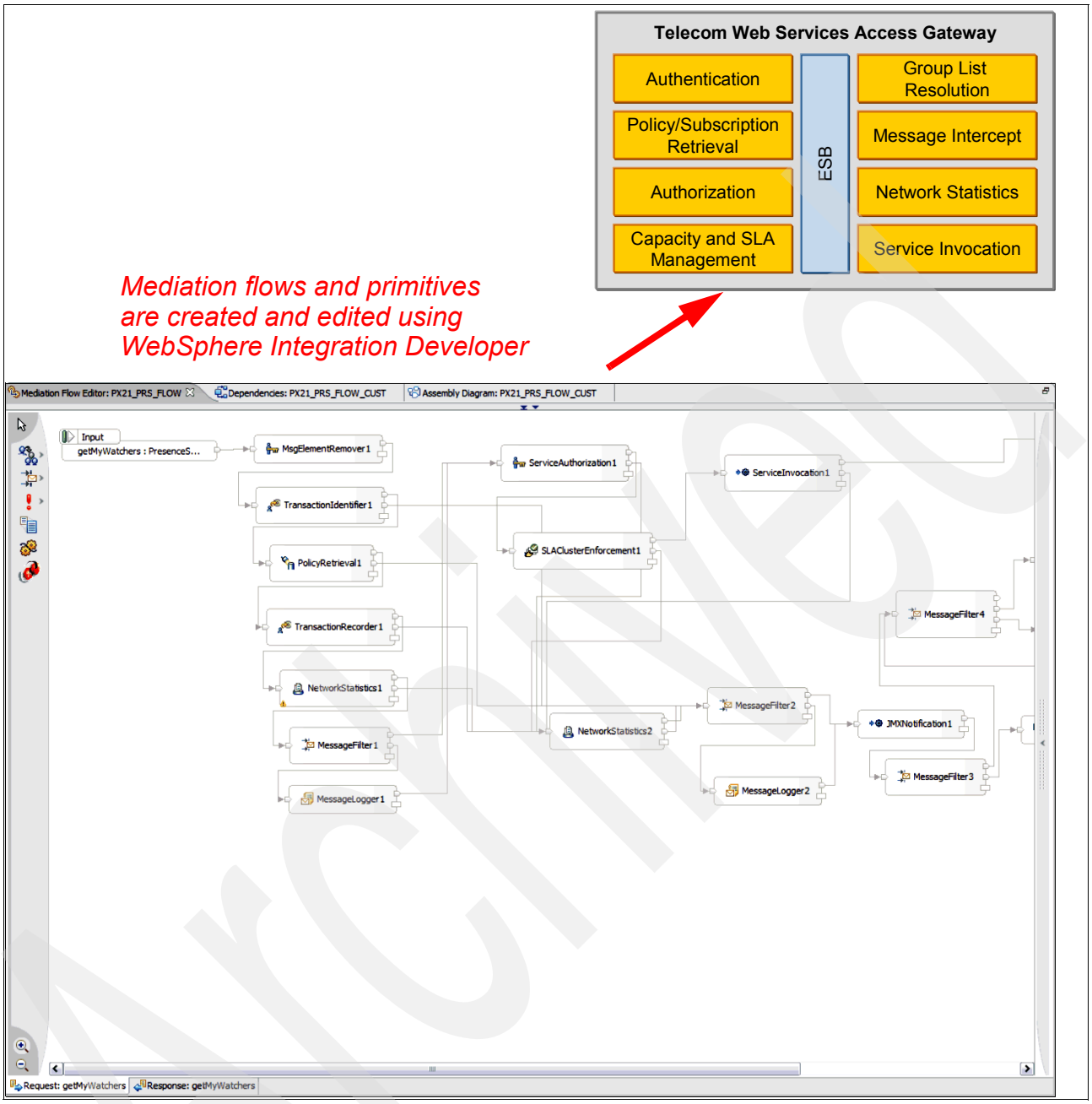


Figure 4-1 Out-of-the-box Access Gateway components and default processing flow

4.2 Mediation primitives

Mediation primitives plug in to the WebSphere Integration Developer tooling palette. They can be used as primitive operations when creating Access Gateway mediation flows. A set of default flows is provided when you install the Access Gateway. To create a customized logic unit, you can utilize the WebSphere Integration Developer tooling to assemble flows among the different components.

Archived

Figure 4-2 illustrates several of the primitives provided with the default mediation flow for the Presence Publish service installed with the WebSphere Integration Developer Plug-in. (The installation of the WebSphere Integration Developer Plug-in is discussed in 4.5, “Tooling / WebSphere Integration Developer Plug-in” on page 88.)

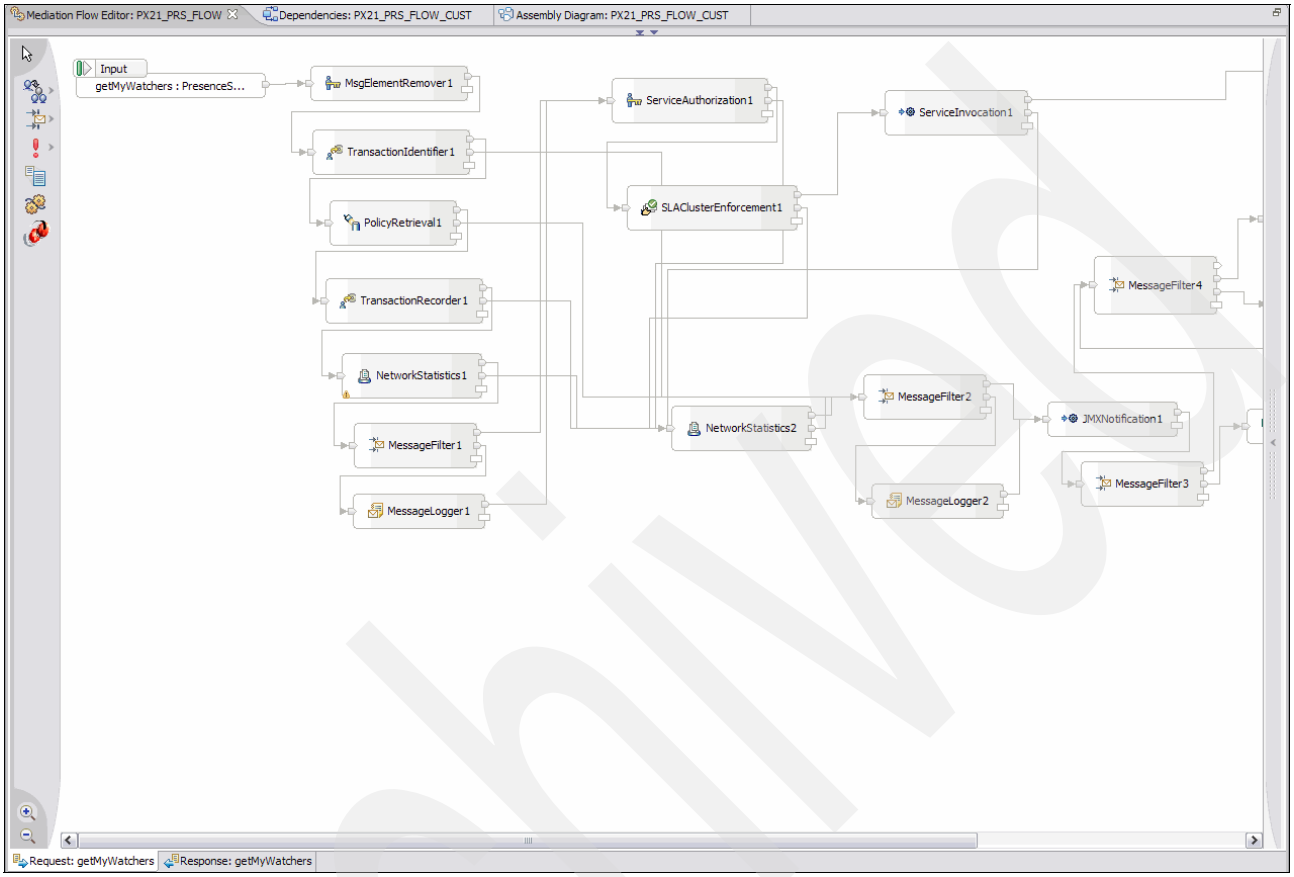


Figure 4-2 Example set of mediation primitives provided with the WebSphere Integration Developer plug-in

Access Gateway mediation primitives consist of Java code that implements a component interface for the Access Gateway. A mediation primitive is a logical unit that has a failure terminal and a number of input and output terminals.

Data objects are passed during execution of an Access Gateway flow between I/O terminals. The data objects use a representation called Service Message Object (SMO) for the Simple Object Access Protocol (SOAP) message being processed for a Web service invocation. The SMO object arrives at a mediation primitive's input terminal during execution flow, which allows control to be passed on to the primitive's mediate method.

The mediation primitive can then process the message and determine where to place the response on its defined output terminal, according to the semantics of that terminal. Mediation flows are created by wiring or connecting the input and output terminals.

Mediation primitives are the primary programming model for the Access Gateway subcomponents. Mediation primitives use a pipeline architecture. You can insert additional mediation primitives anywhere in the flow, change the order of execution (subject to some constraints), and so on. The pipeline architecture is based on the SOAP processing model, where additional SOAP headers may be added, modified, or removed prior to passing execution to the next downstream element.

Individual mediation primitives have unique semantics or expectations, which define the input at its terminal and the data which must be present in order to execute its logic. Adding additional headers to the SOAP message for processing downstream elements, or requiring upstream primitives to insert certain headers, are examples of unique semantics. The semantics or expectations may place added constraints on how a component is used within a mediation flow.

4.2.1 Mediation primitives used by the Access Gateway

This section lists the mediation primitives which are *mandatory* for a base Access Gateway configuration, and those which are *optional*.

Mandatory mediation primitives

The following components are considered mandatory for a base Access Gateway configuration and flow:

- Transaction Identifier mediation primitive
- Policy Retrieval mediation primitive
- Service Invocation mediation primitive

The mandatory components provide base services that support mediation primitives:

- The transaction identifier mediation primitive records information about the transaction within a table that is typically referenced by other mediation primitives.
- The Policy Retrieval mediation primitive retrieves policy data based on the requester, service, and operation being called. This data is used as decision parameter within the mediation primitive's execution flow.

Optional plug-ins used by the default Access Gateway flow

The following components are optional plug-ins and are used by the default Access Gateway flow:

- Network Statistics mediation primitive
- Message Logger mediation primitive
- Service Authorization mediation primitive
- Transaction recorder mediation primitive
- SLA Enforcement mediation primitives
- Group Resolution mediation primitive (Parlay X-specific)
- JMX™ Notification mediation primitive
- CEI Event Emitter mediation primitive
- Message Element Remover mediation primitive (required on the response flow to be the last mediator, namely, at the end of the flow.)
- Fault Transformation mediation primitive (required in the fault/error flow)

Finally, the following mediation primitive is part of the WebSphere ESB product offering and is included in the default flows:

- Message Interceptor mediation primitive

Archived

4.3 Default Access Gateway flow

Default flows are provided with Telecom Web Services Access Gateway for each Parlay X interface. The default flows are provided in binary form as an EAR that can be deployed directly to WebSphere ESB and as a project interchange that can be imported into the WebSphere Integration Developer Tooling Environment for customization. The default flows are designed to support accounting of requests, service/operation level authorization, message capture for regulatory purposes, and traffic level enforcement according to requester service level agreements.

Figure 4-3 illustrates the logic provided as part of the default flows.

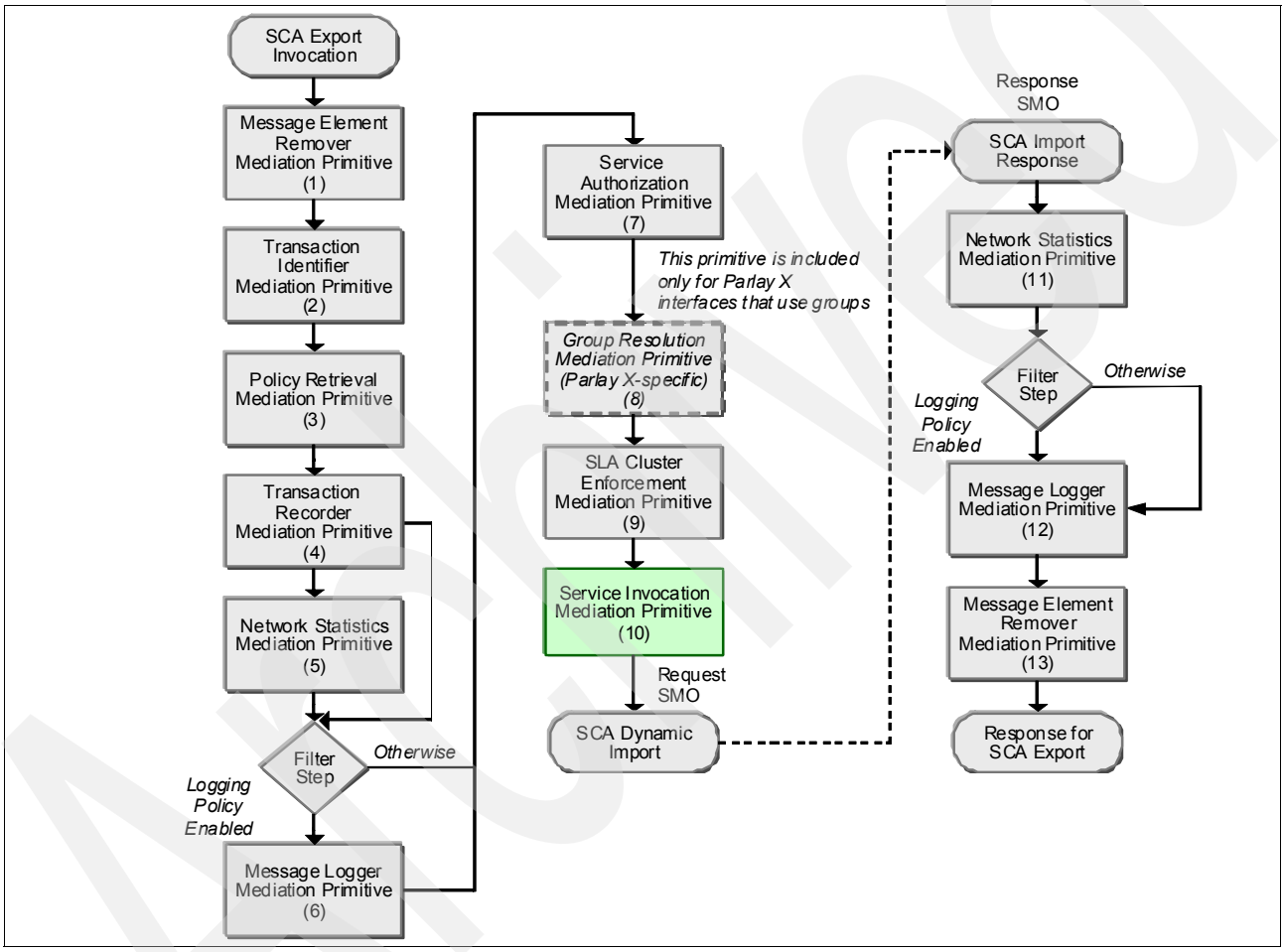


Figure 4-3 Default Access Gateway message processing flow logic

Note: This section is intended to provide an overview of the functionality of the different mediation primitives provided in the default flow and to give an understanding of the overall sequence of the flow between primitives. Details on how to perform specific customizations for mediation primitives is provided in Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127.

Analysis of sequence and actions in the Default Flow

Upon receipt of a Web service request, the following mediation primitives are executed in sequence:

1. Message Element Remover mediation primitive: Remove SOAP Header elements selectively based on the configuration of the Access Gateway.
2. Transaction Identifier mediation primitive: Record information about the transaction, such as the unique ID, requester, service, and operation. This information will be referenced by other accounting data. If no transaction ID is provided in a SOAP header from an upstream mediation primitive (that is, the transaction ID generation mediation primitive), then the WebSphere ESB message ID will be used as the transaction ID. The ID will then be added to the SOAP message header for downstream elements.
3. Policy Retrieval mediation primitive: Retrieve the policy/subscription information from the Service Policy Management system. This calls the specific service policy requested. The resulting information is inserted as SOAP headers into the SMO to be passed between mediation primitives.
4. Transaction Recorder mediation primitive: If transaction recording is enabled by policy attributes, then the information about the transaction, such as the unique ID, requester, service, and operation, is recorded to a database table
5. Network Statistics mediation primitive: If network statistics are being recorded for the particular service operation, statistics are recorded about the request.
6. Message Logger mediation primitive: If messages should be captured for the particular service operation, then the message is logged to a database. This is done using the message recording mediation primitives provided with WebSphere ESB.
7. Service Authorization mediation primitive: If applicable, authorization based on requester, service requested, and operation is verified.
8. Group Resolution mediation primitive: Optional only for Parlay X flows that use groups. Expands all group URIs into a flat list of member URIs by resolving groups through the Group List Management Service. This component will perform recursive resolution of groups to expand nested sub-groups. The list of targets in the operation will then be replaced with the flat, full list of member URIs.
9. SLA Enforcement mediation primitive: If SLA monitoring and enforcement is enabled, then monitoring information is adjusted and the statistics information checked to ensure that a request can be processed given the constraints.
10. Service Invocation mediation primitive: The service invocation mediation primitive is then called to select the appropriate back-end service and perform the back-end Web service invocation. Following this mediation primitive, the SMO is sent through an Import call to the back-end service.
11. Network Statistics mediation primitive: When the response is received, the mediation response flow executes. The flow starts with the network statistics mediation primitive. If network statistics are being recorded for the particular service operation, then statistics are recorded about the response.
12. Message Logger mediation primitive: If messages should be captured for the response, then the message is logged to a database using the WebSphere ESB provided mediation primitive.
13. Message Element Remover mediation primitive: The Message Element Remover checks the user principal against the requester exception list. If the requester is not in the list, the message element remover removes the IBM WebSphere Telecommunications Web Services Server headers element (using an appropriate XPath selector), if IBM WebSphere Telecommunications Web Services Server headers exist.

Upon completion of the flow, the response object is returned to the SCA export invocation as the Web service response.

Error handling within the flow

If a failure occurs during the execution of the request portion of the flow, then the fault handling segment of the flow is invoked. When handling a Web service response, the response is always returned to the requester despite faults in the flow. This ensures consistency with the back-end system. Errors during a request flow are signified by the output of the mediation primitive placed on its fault terminal. This fault terminal is wired to the fault processing segment of the flow, which returns the error response to the requester.

The Access Gateway performs a transformation step to convert a fault to a WSDL defined error type in a fault situation. This step is performed just prior to returning the error response to the Web service client, and it matches the client expected output from the WSDL interface.

The default fault handling flow segment executes the following mediation primitives in sequence (see Figure 4-4 on page 75):

1. The network statistics mediation primitive is called to log statistics about the fault event.
2. The message interceptor mediation primitive is called to log the actual fault object being returned to the requester.
3. The JMX Notification mediation primitive will emit a JMX Notification with the error information.
4. The CEI event emitter mediation primitive is called to emit a common base event (CBE). By emitting CBEs, the fault information can be picked up by external security/monitoring systems, such as Tivoli® products.
5. An XSLT transformation is performed to create an appropriate Web Services Description Language (WSDL) default fault from the SMO. This provides an appropriate fault response for the caller. The XSLT transformation mediation primitive is provided with the WebSphere ESB platform.

Archive

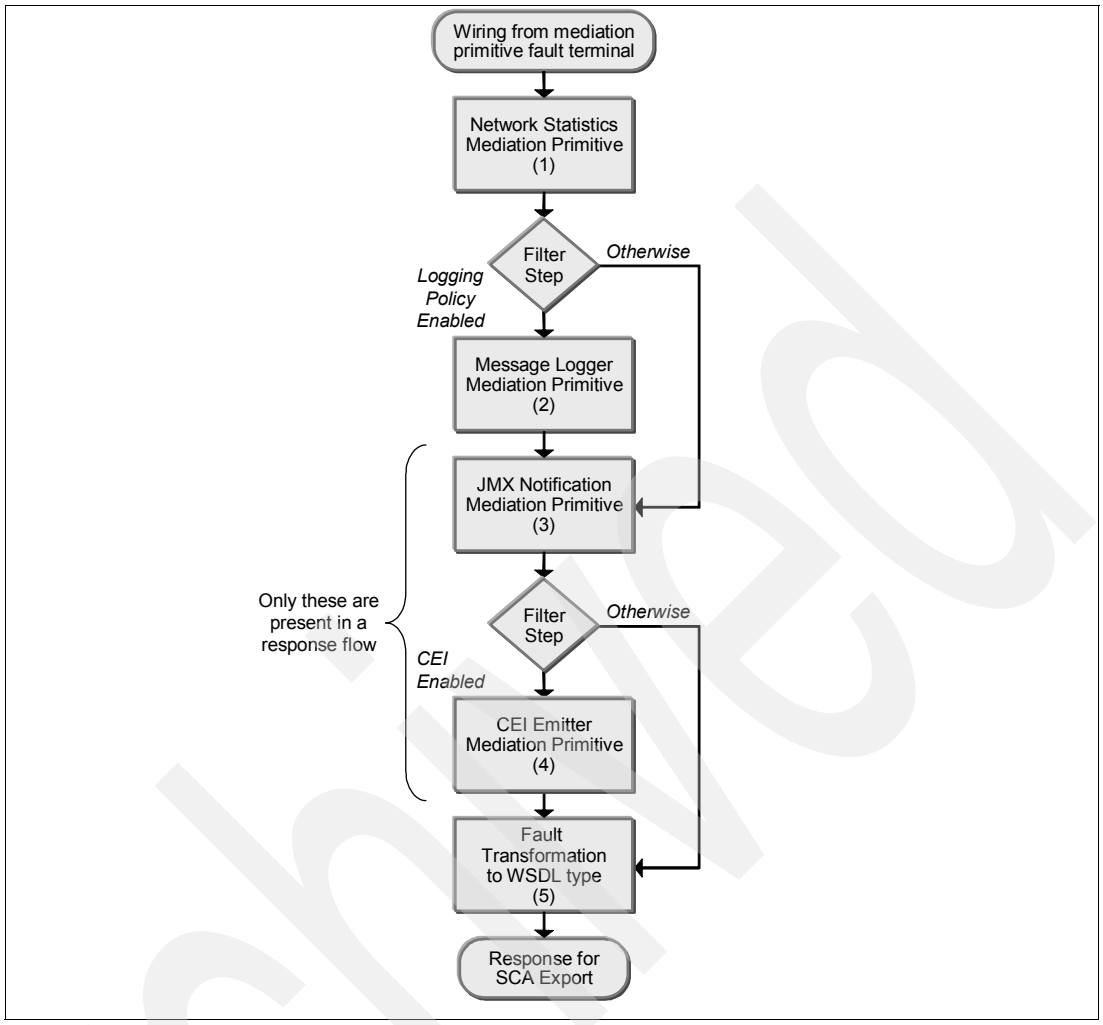


Figure 4-4 Fault handling flow for default Access Gateway flow

Any outgoing requests that originate from within a service implementation, that is, Web service notifications, must also pass through Telecom Web Services Access Gateway. These outbound requests are subject to standard message processing logic. This allows for logging, recording of network statistics, and other policy-driven acts on outbound notifications.

4.4 Working with Telecom Web Services Access Gateway mediation primitives

In order to add additional mediation primitives or customize a flow, it is important to understand the function of each primitive and its interaction with messages.

Each mediation primitive executes message processing logic on the SOAP request as the message passes through the flow. Some mediation primitives record data about the request and others insert headers to be used by downstream mediation primitives.

Archived

Note: In addition to the information referenced below, see also the IBM WebSphere Telecommunications Web Services Server 6.2 Information Center at http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp.

4.4.1 Message Element Removal mediation primitive

The Message Element Removal mediation primitive compares the authenticated requester identity with the exception list property to determine if twssHeaders should be removed or not.

This component inserts the following SOAP header for use by downstream components, provided the headers have not already been inserted by upstream components:

```
<twss:twssHeaders>
<twss:globalTransactionID>
<!-- Global transaction ID that can be used for correlated
   logged transactions. If this header already exists from
   an upstream mediation primitive, then this header will
   be preserved and that transaction ID used for logging.
   Otherwise, the WebSphere ESB message ID will be used. -->
 </twss:globalTransactionID>
<twss:requesterID>
<!-- The requester ID. If this header already exists, then
   it will be preserved and its contents used for logging.
   Otherwise, the WebSphere Application Server user principal will be
used. If the
   requester is unauthenticated (that is, security is turned
   off), then the special requester "UNAUTHENTICATED" will be
   used. -->
</twss:requesterID>
</twss:twssHeaders>
```

Note that for outbound notification, the service platform has already inserted the requesterID within the message. In that case, only the globalTransactionID header is added.

Faults and Alarms

- SOAC4023: No value was specified for the {2} element in twssHeaders.

4.4.2 Transaction Recorder mediation primitive

The Transaction Recorder mediation primitive logs information about the Web service request to a database for accounting purposes and for reference by other mediation primitive tables. The mediation primitive also inserts information about the transaction into the SOAP headers.

Added SOAP headers

```
<twss:twssHeaders>
<twss:globalTransactionID>
<!-- Global transaction ID that can be used for correlated
   logged transactions. If this header already exists from
   an upstream mediation primitive, then this header will
   be preserved and that transaction ID used for logging.
   Otherwise, the WebSphere ESB message ID will be used. -->
 </twss:globalTransactionID>
```

```
<twss:requesterID>
<!-- The requester ID. If this header already exists, then
   it will be preserved and its contents used for logging.
   Otherwise, the WebSphere Application Server user principal will be
used. If the
   requester is unauthenticated (that is, security is turned
   off), then the special requester "UNAUTHENTICATED" will be
   used. -->
</twss:requesterID>
 …
</twss:twssHeaders>
```

Table 4-1 illustrates the fields and each of their associated attributes contained in the transactions database.

Table 4-1 Transactions database table definition

Field	Key	Data Type	Contents
TRANSACTIONID	PK	VARCHAR(127)	Unique transaction identifier
REQUESTER		VARCHAR(250)	Requester identity
SERVICE		VARCHAR(250)	The service being invoked
SERVICEOPERATION		VARCHAR(250)	The service operation being invoked
ADDTIME		TIMESTAMP	Transaction creation time

Faults and Alarms

- Illegal Transaction ID Fault: This fault will be generated when attempting to insert a non-unique transaction ID. This indicates that an entry in the transactions table already exists for this ID. The fault is placed on the fault terminal and a JMX notification is generated.
- Data Store Not Available Alarm: Indicates that an error occurred while accessing the database to write out transaction information. A fault is placed on the fault terminal and a JMX notification is generated.

4.4.3 Policy Subscription mediation primitive

The policy or subscription mediation primitive fetches policy information from a Service Policy Management system and populates SOAP headers with policy information. This policy information is then passed along with the request to downstream mediation primitives and back-end Parlay X Web service implementations for policy-based decision making during service execution.

Archived

Used SOAP headers

```
<twss:twssHeaders>
   …
   <twss:requesterID>
      <!-- Used for the lookup of the requester's policies. If this
         header is missing, "UNAUTHENTICATED" is assumed. -->
   </twss:requesterID>
   …
</twss:twssHeaders>
```

Added SOAP headers

```
<twss:twssHeaders>
   …
   <twss:policies>
      <twss:policy  attribute="attrib_name"  value="attrib_value"/>
      <twss:policy  attribute="attrib_name"  value="attrib_value"/>
      …
   </twss:policies>
   …
</twss:twssHeaders>
```

Policy interface

This component uses the Service Policy Manager access interface to perform policy resolution.

Faults and Alarms

- Policy subsystem Not Available: An error occurred while contacting the policy management system. A fault is placed on the fault terminal and a JMX notification is generated.
- Policy Retrieval Fault: An error occurred retrieving policy information for a particular (requester, service, or operation) tuple. A fault is placed on the fault terminal and a JMX notification is generated.

4.4.4 Network Statistics mediation primitive

The Network Statistics mediation primitive records message entry and exit information. The statistics are stored in a database for use by network operations. This information can be used to construct traffic summaries for network analysis and capacity planning.

Table 4-2 shows the data type and default value for a network statistics service policy.

Table 4-2 Network statistics service policies

Name	Type	Description
Message.statistics.RecordStatistics	Boolean: true or false	Statistics recording enabled. Default value: true

Table 4-3 shows the fields and their respective attrtibutes contained within the NETWORKSTATISTICS database table definition.

Archived

Table 4-3 NETWORKSTATISTICS database table definition

Field	Key	Data type	Contents
ENTRYID	PK	VARCHAR(127)	Unique identifier for the entry.
TRANSACTIONID	FK	VARCHAR(127)	Unique transaction identifier. Note there can be multiple entries under a given transaction ID.
MESSAGETYPE		CHAR(1)	I: inbound; O:outbound; F: fault.
EVENTTYPE		VARCHAR(250)	The event type, indicating at what point in the execution this statistic is being recorded.
EVENTTIME		TIMESTAMP	Time of message.

Table 4-4 shows the fields and their respective attrtibutes contained within the TRANSACTIONS database table definition.

Table 4-4 TRANSACTIONS database table definition

Field	Data Type	Contents
TRANSACTIONID	VARCHAR(127)	Unique transaction identifier.
REQUESTER	VARCHAR(250)	Requester identity.
SERVICE	VARCHAR(250)	The service being invoked.
SERVICEOPERATION	VARCHAR(250)	The service operation being invoked.
MESSAGETYPE	CHAR(1)	I: inbound; O: outbound; F:fault.
EVENTTYPE	VARCHAR(250)	The event type indicating at what point in the execution this statistic is being recorded.
EVENTTIME	TIMESTAMP	Time of message.

Faults and Alarms

- Data Store Not Available Alarm: Indicates an error occurred accessing the database for writing out transaction information. A fault is placed on the fault terminal and a JMX notification is generated.

4.4.5 Service Authorization mediation primitive

The Service Authorization mediation primitive provides fine-grained authorization for access to services. It functions as an application layer authorization service that looks at a policy attribute that indicates whether, for a given requester, access to the service or operation can succeed. The service authorization component follows simple block or pass semantics: All authorization policies must be allowed for the request to be processed.

Table 4-5 shows the service policies and their data types within the Service Authorization mediation primitive.

Table 4-5 Service policies

Name	Type	Description
requester.service.Authorized	Boolean	Access to a service allowed for requester. Default value: true.
requester.operation.Authorized	Boolean	Access to perform a service operation allowed for requester. Assumed Default value: true.
requester.AnonymousAccessAllowed	Boolean	Indicates whether anonymous requests should be allowed to pass. Default value: true.

4.4.6 Parlay X Group Resolution mediation primitive

For Parlay X Web service implementations that can accept group URIs within a list of targets for a given operation, the group resolution mediation primitive resolves group URIs within those operations to their member URIs. The expanded groups are placed into the SOAP headers for downstream processing.

This component will act as a Parlay X V2.1 Address List Management interface client requester to perform the actual resolution. This component uses the Parlay X V2.1 Address List Management Group::queryMembers() interface to resolve groups. The endpoint is configurable in the ESB Administration console under SCA Modules.

Added SOAP headers

```
<twss:twssHeaders>
   …
   <twss:operationTargets>
      <!-- The number of targets for this operation once all groups
         have been resolved. Duplicates are included in this count.
         Downstream compnents should assume a default value of '1'
         If this header is not present. -->
   </twss:operationTargets>
   <twss:resolvedGroups>
      <twss:GroupList groupURI="glmgroup:name@domain">
         (member)member1@domain(/member>
         (member)member2@domain(/member>
         …
      </twss:GroupList>
      <twss:GroupList groupURI="glmgroup:other@domain">
         …
      </twss:GroupList>
      …
   </twss:resolvedGroups>
   …
</twss:twssHeaders>
```

Faults and Alarms

- Group Resolution Sub-system Not Available Alarm: An error occurred contacting the Address List Management service. A fault is placed on the fault terminal and a JMX notification is generated.
- Group Resolution Fault: An error occurred while resolving a group. A fault is placed on the fault terminal and a JMX notification is generated.

4.4.7 SLA Enforcement mediation primitives

The SLA Enforcement mediation primitives measure the system use by requester to enforce policy-driven service level agreements.

There are two instances (or versions) of this mediation primitive: one to track SLA statistics on a per-server basis and the other to track SLA statistics across the cluster of WebSphere ESB instances. In a load-balanced environment with a random distribution of requests, the local enforcement mediation primitive can be used as a light-weight version of cluster enforcement. The cluster enforcement scheme provides a higher threshold for limiting inbound requests by implementing a distributed reservation algorithm using WebSphere Application Server high availability manager facilities.

Table 4-6 shows the service policies and their data types within the local enforcement mediation primitive.

Table 4-6 Service policies for the local enforcement mediation primitive

Name	Type	Description
message.sla.LocalEnabled	Boolean	SLA measurements enabled. Default value: true.
message.sla.LocalWeight	Integer	Weight for the tuple (requester, service, or operation). Default value: 0.
message.sla.LocalRequesterRate	Integer	The number of requesters per second to admit in the system for this requester. Default value: 0 (denied).
message.sla.LocalServiceRate	Integer	The number of service requests per second to admit in the system for this requester. Default value: 0 (denied).
message.sla.LocalOperationRate	Integer	The number of operation requests per second to admit in the system for this requester. Default value: 0 (denied).

Archived

Table 4-7 shows the service policies and their data types within the cluster enforcement mediation primitive.

Table 4-7 Service policies for the cluster enforcement mediation primitive

Name	Type	Description
message.sla.ClusterEnabled	Boolean	SLA measurements enabled. Default value: true.
message.sla.ClusterWeight	Integer	Weight for the tuple (requester, service, or operation). Default value: 0.
message.sla.ClusterRequesterRate	Integer	The number of requesters per second to admit in the system for this requester. Default value: 0 (denied).
message.sla.ClusterServiceRate	Integer	The number of service requests per second to admit in the system for this requester. Default value: 0 (denied).
message.sla.ClusterOperationRate	Integer	The number of operation requests per second to admit in the system for this requester. Default value: 0 (denied).

Used SOAP headers

```
<twss:twssHeaders>
   …
   <twss:requesterID>
      (!-- The requester ID. If this headerismissing,then
      "UNAUTHENTICATED" is assumed. -->
   </twss:requesterID>
   <twss:operationTargets>
      <!-- Used to calculate the weighting or cost of this request. If
         this header is not provided, then a default of '1' is
         assumed. -->
   </twss:operationTargets>
   …
</twss:twssHeaders>
```

4.4.8 Transaction Identifier mediation primitive

The Transaction Identifier mediation primitive examines SOAP headers to determine if an upstream mediation primitive has supplied a global transaction ID or requester ID for a Web service request.

If no global transaction ID exists, the Transaction Identifier mediation primitive generates a Universal Unique Identifier (UUID) (RFC 4122) and inserts it in a globalTransactionID SOAP header.

If no requester ID exists, the Transaction Identifier mediation primitive gets the WebSphere Application Server user principal for the request and inserts that ID into a requester SOAP header. Because downstream mediation primitives typically use these headers, place the Transaction Identifier mediation primitive at the start of custom access gateway flows.

Policy configuration

This mediation primitive uses the following policies for runtime configuration:

- None

Mediation primitive properties

- None

Upstream SOAP headers

The following SOAP header elements are expected from upstream mediation primitives:

- None

Added SOAP headers

The SOAP header elements shown in Example 4-1 are added or modified for downstream mediation primitives.

Example 4-1 SOAP header elements for downstream mediation primitives

```
<twss:twssHeaders>
  ...
  <twss:globalTransactionID>
    <!-- Global transaction ID that can be used for correlating
         transactions. If this header already exists from an
         upstream mediation primitive, this header is
         preserved and that transaction ID is used for logging.
         Otherwise, a UUID is used. -->
  </twss:globalTransactionID>
  <twss:requesterID>
    <!-- The requester ID. If this header already exists,
         it is preserved and used as the requester identity.
         Otherwise, the WAS user principal is used. If the
         requester is unauthenticated (for example, security is turned
         off), the special requester "unauthenticated" is
         used. -->
  </twss:requesterID>
  ...
</twss:twssHeaders>
```

Message handling

Messages that are successfully processed by the Transaction Identifier mediation primitive are passed to the output terminal of the mediation primitive. If an error occurs while processing the message, the message is redirected to the fault terminal.

The Service Message Object (SMO) data object transient context ("context/transient/exceptionType") indicates whether a service-related or policy-related exception occurred. For the Transaction Identifier mediation primitive, this context is always set to service.

Archived

The fault information is set in the SMO headers, as indicated in Table 4-8.

Table 4-8 Fault information for message handling set in the SMO headers

SMO header (represented by XPath)	Contents
ServiceMessageObject/context/failInfo/failureString	The full message text that represents the fault situation with substituted variables. For example, SOAC4025E: Error occurred.
ServiceMessageObject/context/failInfo/origin	The name of the mediation primitive class that originated the fault.
ServiceMessageObject/SOAPFaultInfo/faultcode	The IBM WebSphere Telecommunications Web Services Server message code representing the fault situation. For example, SOAC4025E.
ServiceMessageObject/SOAPFaultInfo/faultstring	The full message text that represents the fault situation with substituted variables. For example, SOAC4025E: Error occurred.

4.4.9 JMX Notification mediation primitive

The JMX Notification mediation primitive is used to emit Java Management Extensions (JMX) notifications through the WebSphere Application JMX infrastructure. WebSphere Application Server provides plug-points that IBM and third-party software use to receive those notifications and gather management-related information. The JMX Notification mediation primitive gathers information about access gateway processing faults from the Service Message Object (SMO) headers according to the SMO schema.

SMO headers

The headers used by other access gateway mediation primitives are described in Table 4-9.

Table 4-9 SMO headers

SMO header (represented by XPath)	Content
ServiceMessageObject/context/failInfo/failureString	The full message text that represents the fault situation with substituted variables. For example, SOAC4025E: Error occurred.
ServiceMessageObject/context/failInfo/origin	The name of the mediation primitive class that originates the fault.
ServiceMessageObject/SOAPFaultInfo/faultcode	The IBM WebSphere Telecommunications Web Services Server message code that represents the fault situation. For example, SOAC4025E.
ServiceMessageObject/SOAPFaultInfo/faultstring	The full message text that represents the fault situation with substituted variables. For example, SOAC4025E: Error occurred.
ServiceMessageObject/context/transient/exceptionType	The exception type element under the transient context may contain one of two values: service or policy. These values represent the context of the fault, indicating whether it was caused by the operation of the service or a policy-related action.

The notification contains the detailed fault string with the origin element used as the originating class for the JMX notification.

When running in a secured environment, the JMX Notification mediation primitive must have access to the administrative user name and password to have sufficient privileges to emit notification events. These credentials can be configured as part of the mediation primitive configuration properties.

Policy configuration

This mediation primitive uses the following policies for runtime configuration:

- None

Mediation primitive properties

This mediation primitive uses the configuration properties shown in Table 4-10. These properties can be modified using WebSphere Integration Developer tooling. Properties that are promoted can be configured using the Integrated Solutions Console.

When running in a secured environment, the JMX Notification mediation primitive must have access to the administrative user name and password to have sufficient privileges to emit notification events. These credentials can be configured as part of the mediation primitive configuration properties.

Table 4-10 Mediation primitive configuration properties

Property	Type	Promoted?	Description
adminUserName	string	Yes	The administrative user name used to run as the administrative user principal when emitting JMX notifications. This user name is needed only when security is enabled. Default: (blank).
adminPassword	string	Yes	The administrative user password used to run as the administrative principal when emitting JMX notifications. Encrypt the password with the AdminTool provided with IBM WebSphere Telecommunications Web Services Server: java -jar AdminTool.jarencryptpassword Default: (blank).

Upstream SOAP headers

The following SOAP header elements are expected from upstream mediation primitives:

- None

Added SOAP headers

The following SOAP header elements are added or modified for downstream mediation primitives:

- None

Message handling

Messages that are successfully processed by the JMX Notification mediation primitive are passed to the output terminal of the mediation primitive. If an error occurs while processing the message, the error is logged for tracing.

Archived

4.4.10 Service Invocation mediation primitives

The Service Invocation mediation primitive performs runtime selection of the back-end service implementation to invoke. A policy attribute is used to provide the endpoint of the back-end service. The service invocation mediation primitive also provides a session propagation function between the calling Web service client and back-end Web service.

Table 4-11 shows the service policies and their data types within the service invocation mediation primitives.

Table 4-11 Service policies for service invocation mediation primitives

Name	Type	Description
service.Endpoint	String	The endpoint to invoke for this service. This policy attribute corresponds to the service provider's service to implement mapping for the (requester, service, or operation) tuple. Default value: Raises a fault if this attribute does not exist.
service.notification.Username	String	The user name used for transport-level security in delivering outbound notifications. Default value: no transport-level security if not present.
service.notification.Password	String	Contains an encrypted password used for transport-level security in delivering outbound notifications. The password should be encrypted with the administration tool provided with IBM WebSphere Telecommunications Web Services Server. Default value: no transport-level security if not present.

Used SOAP headers

Parlay X Web service implementations might need to act as Web service client requesters to external third-party entities to deliver notifications. These notifications pass through Telecom Web Services Access Gateway for processing. An additional header must be added that indicates the endpoint of the final notification delivery destination. If this header is present, its endpoint overrides any policies.

```
<twss:twssHeaders>
   …
   <twss:notification>
      <twss:destination>
         <!-- The notification endpoint destination -->
      </twss:destination>
   </twss:notification>
   …
</twss:twssHeaders>
```

Faults and Alarms

- No Service Endpoint Policy Alarm: The policy attribute indicating the back-end endpoint to use is missing. A fault is placed on the fault terminal and a JMX notification is generated.

4.4.11 CEI Event Emitter mediation primitives

The common event infrastructure (CEI) provides a means for applications to emit events in a common format that can be filtered, categorized, and monitored by external applications.

The common format used is the common base event (CBE), an extensible XML format for events. The CEI Event Emitter mediation primitive uses the CEI client API to emit CBEs that contain information about the SOAP fault being returned to the requester. The SOAP fault contains within the body the application exception that initiated the fault. The mediation primitive makes use of CEI event filtering to determine whether or not to emit the fault. Filtering is configured through CEI administration.

shows the service policies and their data types within the CEI Event Emitter mediation primitive.

Table 4-12 Service policies for CEI Event Emitter mediation primitive

Name	Type	Description
message.cei.Enabled	Boolean	CEI event emission enabled. Default value: true.

Table 4-13 shows the CBE Extensions.

Table 4-13 CBE Extensions

Name	Type	Description
soapHeaders	Element	All SOAP headers at the time of the fault as sub elements.
faultCode	String	SOAP fault code.
faultString	String	Human readable SOAP fault string.
faultActor	String	SOAP actor that caused the fault.
detail	String	Application-specific error information.

Faults and Alarms

- CEI Emission Alarm: An error occurred while sending the fault information through CEI. A JMX notification is generated.

4.4.12 Custom mediation primitives

The WebSphere Integration Developer tooling allows the creation of custom primitives through Java snippets. A custom mediation primitive gets placed on the canvas and a code editor allows for the insertion of Java mediation code.

Archive

The process for creating a custom mediation primitive is discussed thoroughly, together with an example implementation, in Chapter 5, “Developing and customizing a custom Access Gateway flow” on page 127.

4.5 Tooling / WebSphere Integration Developer Plug-in

A custom access gateway flow is created using WebSphere Integration Developer and the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Access Gateway mediation primitive plug-ins. These are available as part of the IBM WebSphere Telecommunications Web Services Server product and are provided on a separate image from the IBM WebSphere Telecommunications Web Services Server runtime components. The process is similar to creating a standard WebSphere Integration Developer Mediation Module with a few specific steps to include IBM WebSphere Telecommunications Web Services Server requirements.

The flow definition is produced within the WebSphere Integration Developer tooling. The following section discusses how to install the WebSphere Integration Developer Plug-in so that you may begin working with the default mediation flows provided with the IBM WebSphere Telecommunications Web Services Server product.

4.5.1 IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-in installation

This section describes the steps for downloading and installing the WebSphere Integration Developer toolkit.

Note: The WebSphere Integration Developer Plug-ins Installer is a separate CD image that is part of the IBM WebSphere Telecommunications Web Services Server product offering.

Also note that the specific instructions listed here pertain to IBM WebSphere Telecommunications Web Services Server V6.2. For different versions of IBM WebSphere Telecommunications Web Services Server, refer to the information center for complete installation instructions at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

Do the following steps:

1. Download the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-in (TWSS_WIDPlugin_C14CAEN.zip). Open the zip file and extract the contents (Figure 4-5).

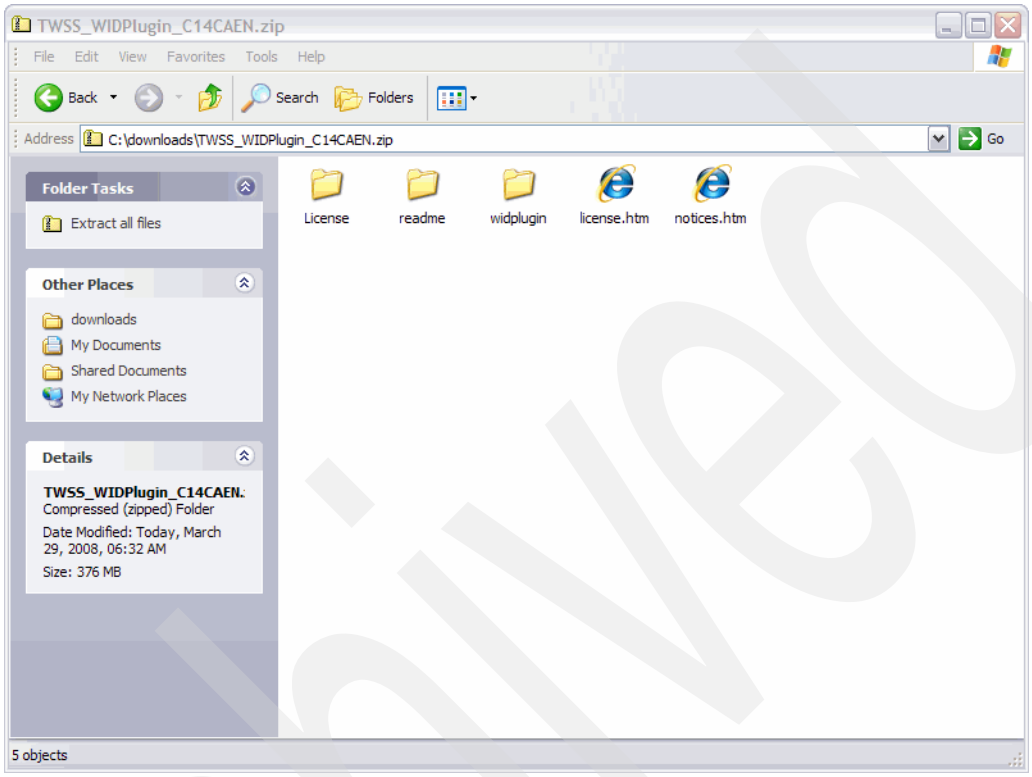


Figure 4-5 Contents of the WebSphere Integration Developer Plug-in .zip file

Archived

2. After extracting the zip file's contents, the contents will appear as shown in Figure 4-6 on page 90.

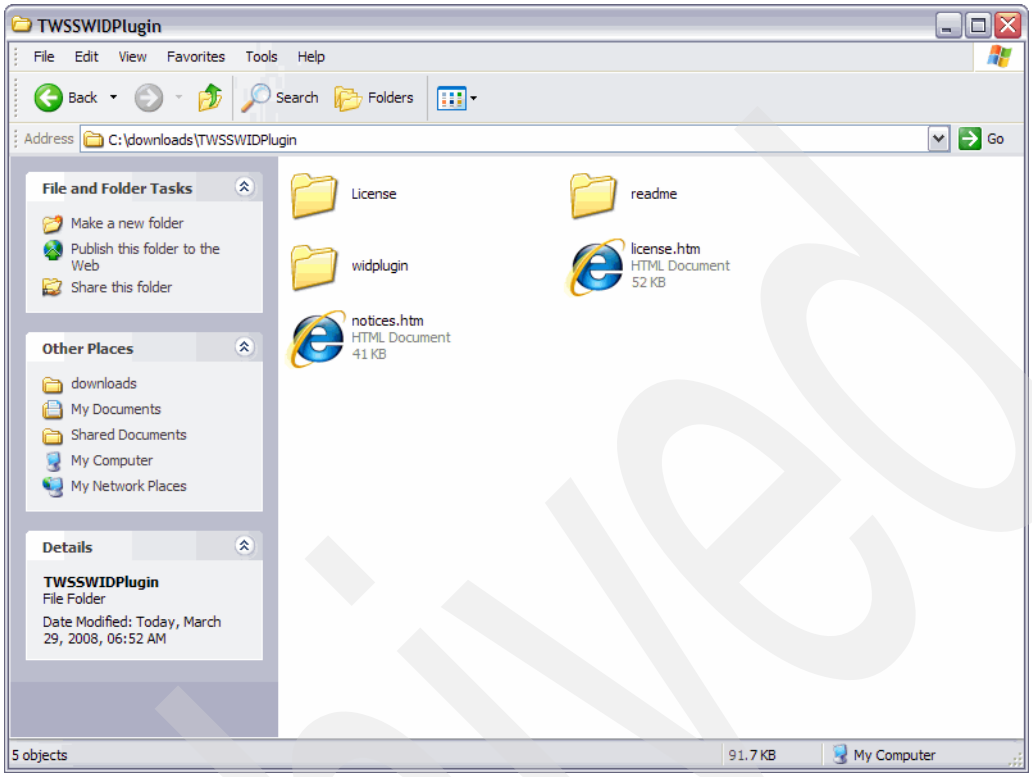


Figure 4-6 Contents of the WebSphere Integration Developer Plug-In

3. Open the WIDPlugin folder and navigate to folder that represents the operating system (OS) where WebSphere Integration Developer has been installed (Figure 4-7).

Archived

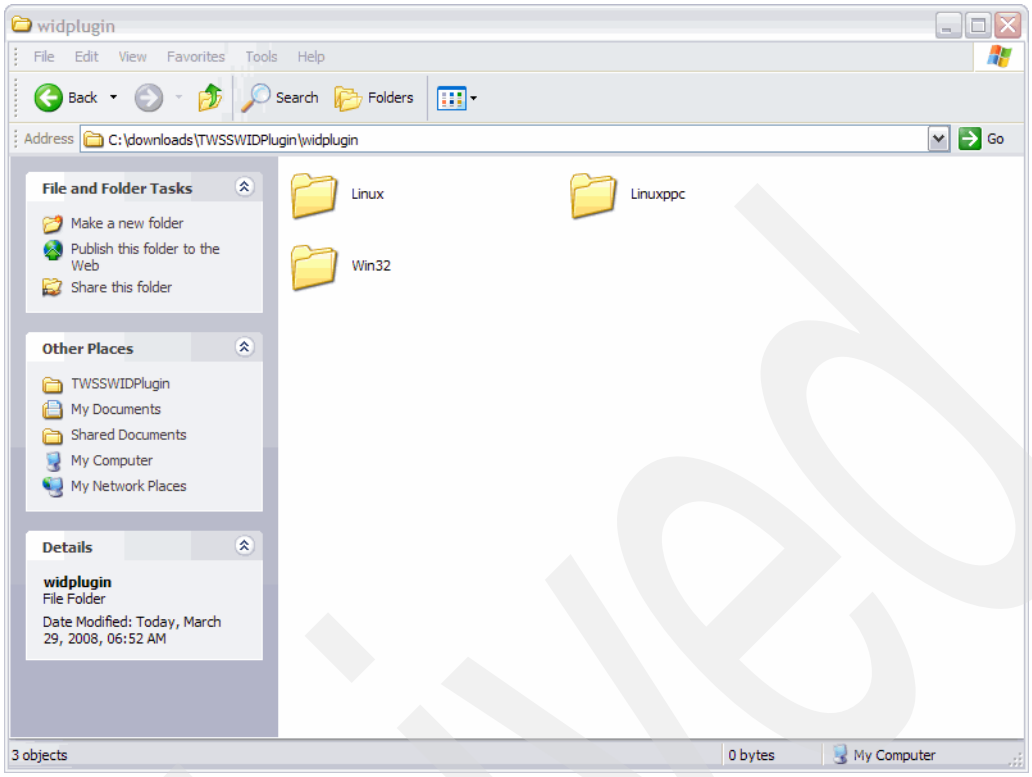


Figure 4-7 Selecting the proper OS for WebSphere Integration Developer Plug-in

4. Once selected, navigate to the setup.exe file to initiate the installation (Figure 4-8).

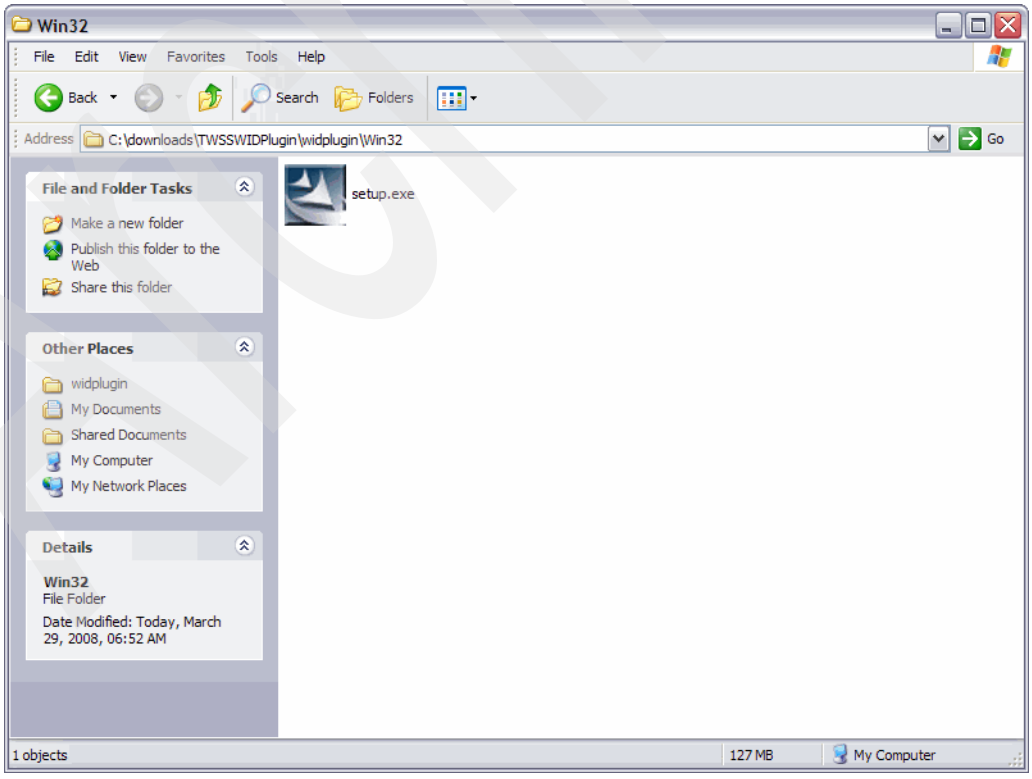


Figure 4-8 Setup executable for WebSphere Integration Developer Plug-in

Archived

5. Launch the setup.exe and select the Language in the drop-down menu. Click **OK** to continue the installation of IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-in Installer (Figure 4-9).

Figure 4-9 Selecting language

Archived

6. The IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-in Installer will take few minutes to initialize. Click **Next** on the initial windows that present you with the license agreement and provide an overview about the functionality of the WebSphere Integration Developer plug-in (see Figure 4-10). Click **Next** until you are prompted to enter a directory location for installing the plug-in.

Figure 4-10 Series of windows relating to the license agreement and beginning the installation

7. Browse the directory where you have installed the WebSphere Integration Developer, for example, choose the WebSphere Integration Developer Home installation directory. Click the **Next** button to proceed, as shown in Figure 4-11.

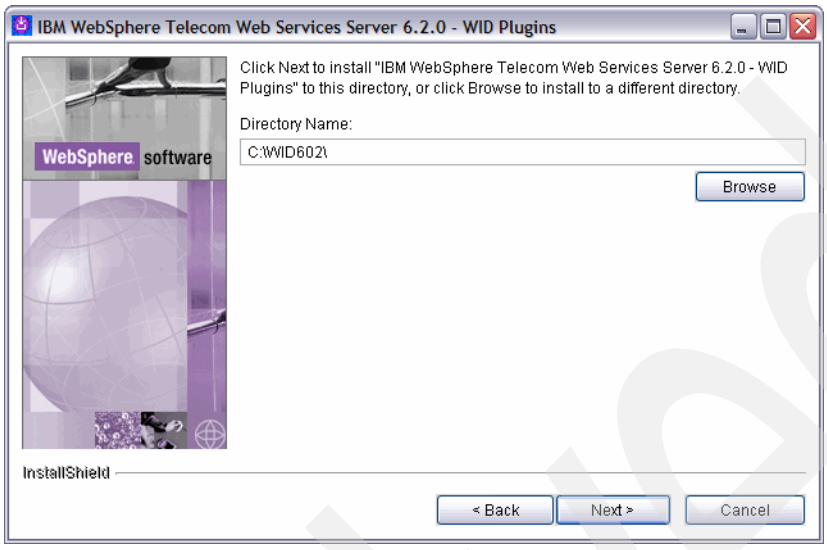


Figure 4-11 Selecting the installation directory

8. Select the features for IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer plug-ins you would like to install by enabling the check box next to **IBM WebSphere Telecom Web Services Server IMS WID Plug-ins** and Click **Next** (Figure 4-12).

Figure 4-12 Selecting the features

9. Verify the installation path and features that you have selected to install. Click **Next** (Figure 4-13 on page 95).

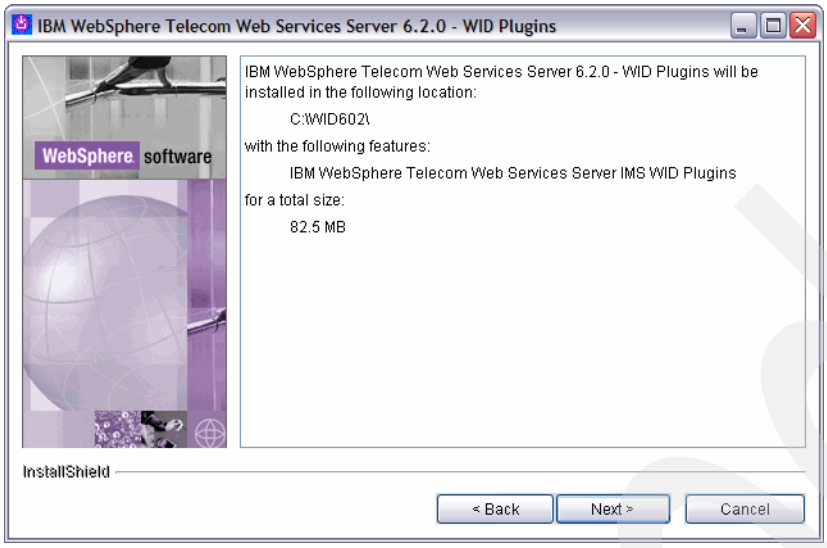


Figure 4-13 Verifying the installation path

10. Finally, the installation will begin. This will take several minutes to complete (Figure 4-14). Click **Finish** once the installation is complete.

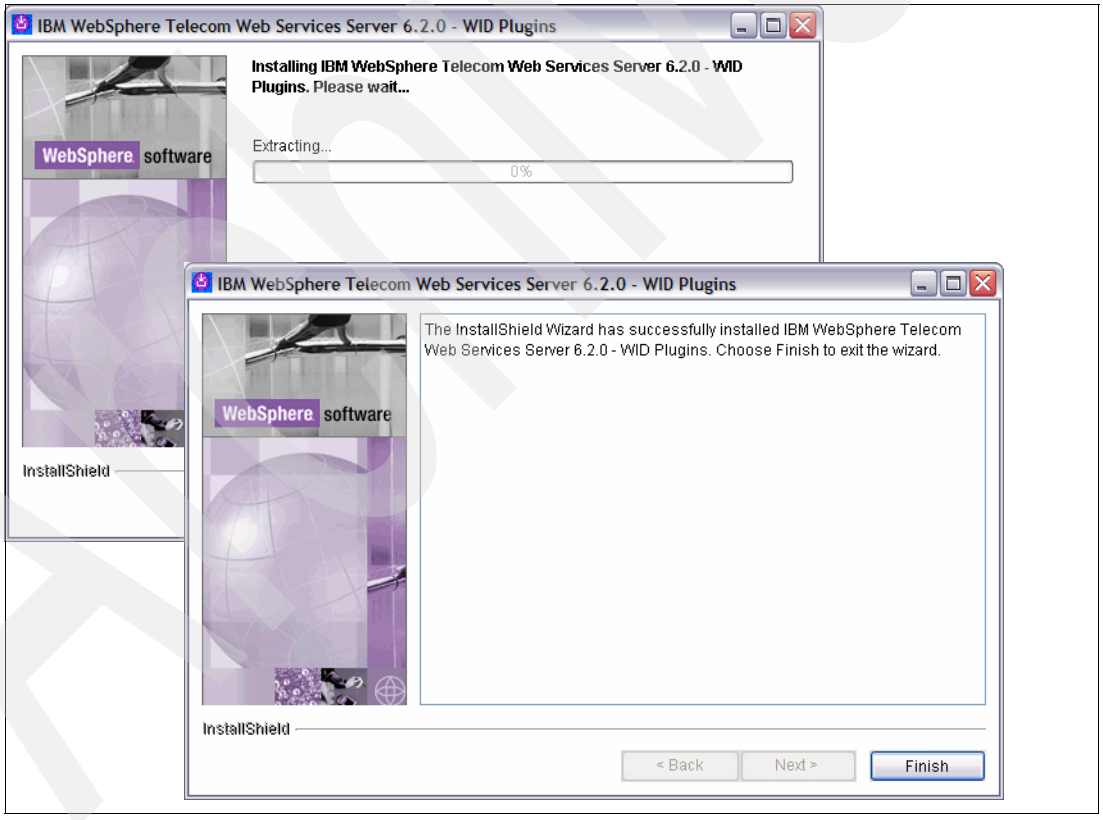


Figure 4-14 Installation in progress and completion

Verifying the installation of the WebSphere Integration Developer Plug-in

The following section describes several steps to validate the installation of this plug-in.

Important: After installing or updating any mediation primitive plug-ins for WebSphere Integration Developer, you must restart WebSphere Integration Developer using the -clean option (**`C:\WID61\wid.exe -clean`**) for the plug-ins to load.

The actual mediation primitive plug-ins are installed in the <WID_home>\wstools\eclipse\plugins directory, as shown in Figure 4-15.

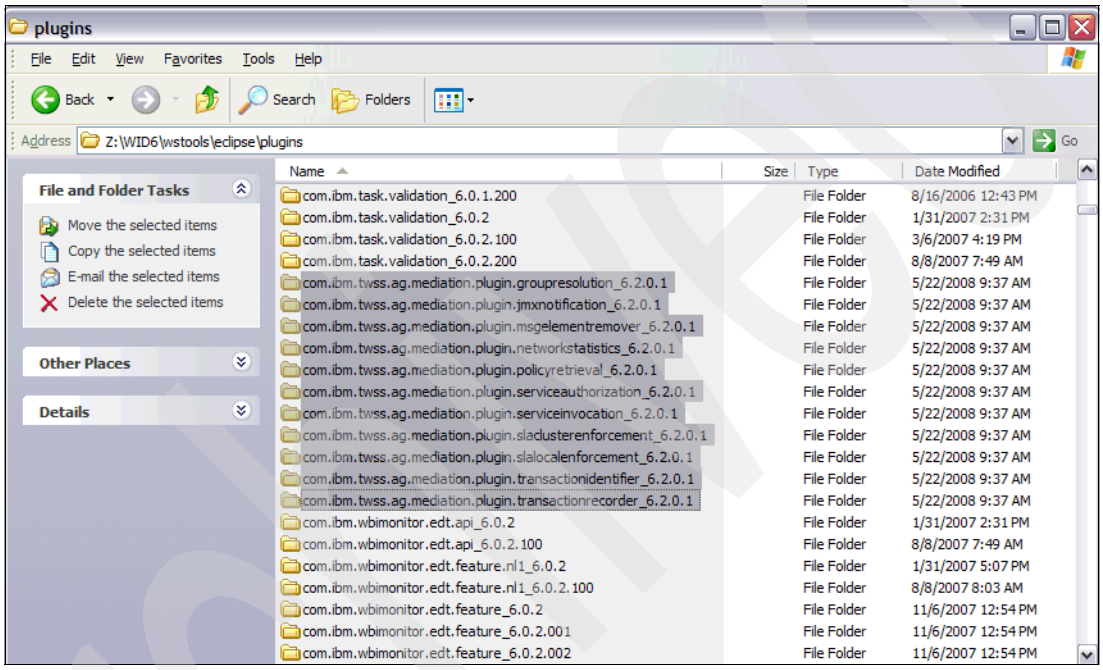


Figure 4-15 Mediation primitive plug-ins

The AG default flows are installed in the TWSS folder within the <WID_HOME>\TWSS directory.

To verify the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-in installation, follow these steps:

11. Navigate to the WebSphere Integration Developer Home directory and search for the folder name TWSS (Figure 4-16).

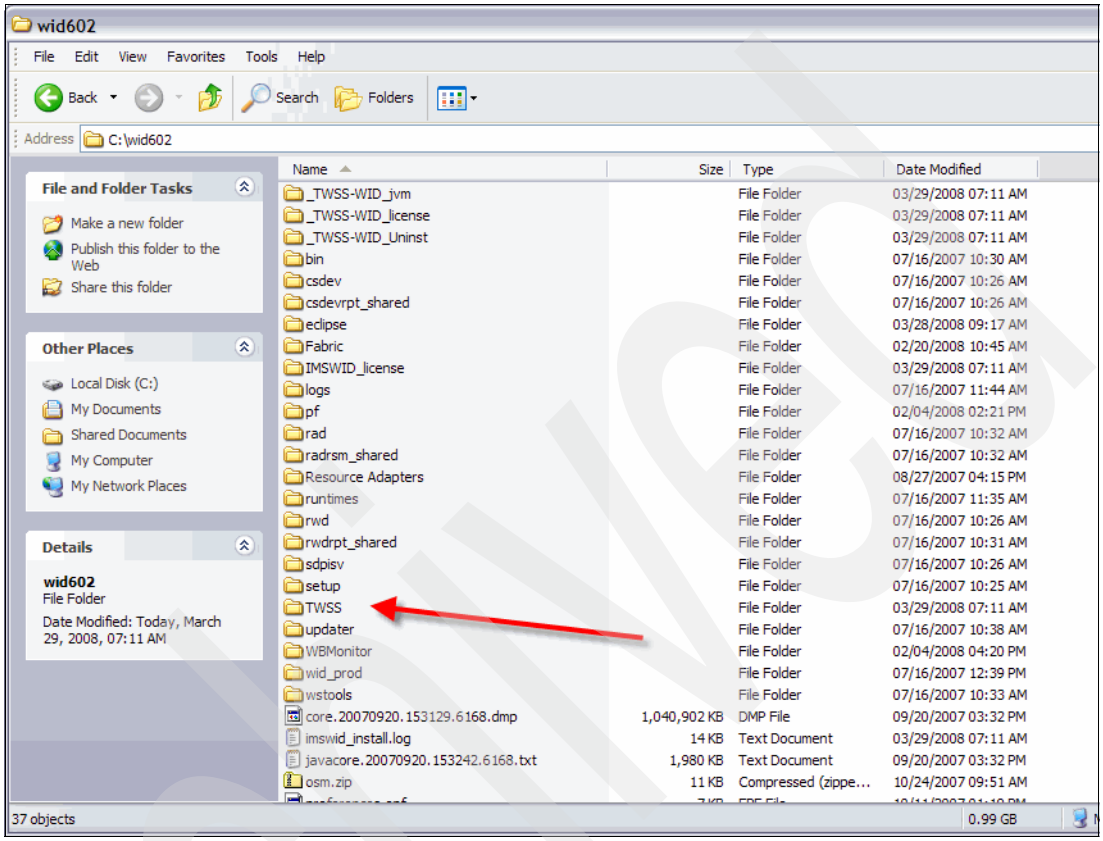


Figure 4-16 Verifying installation location

Archive

12. Open the TWSS Folder. This is the directory that contains the default mediation flows, as shown in Figure 4-17.

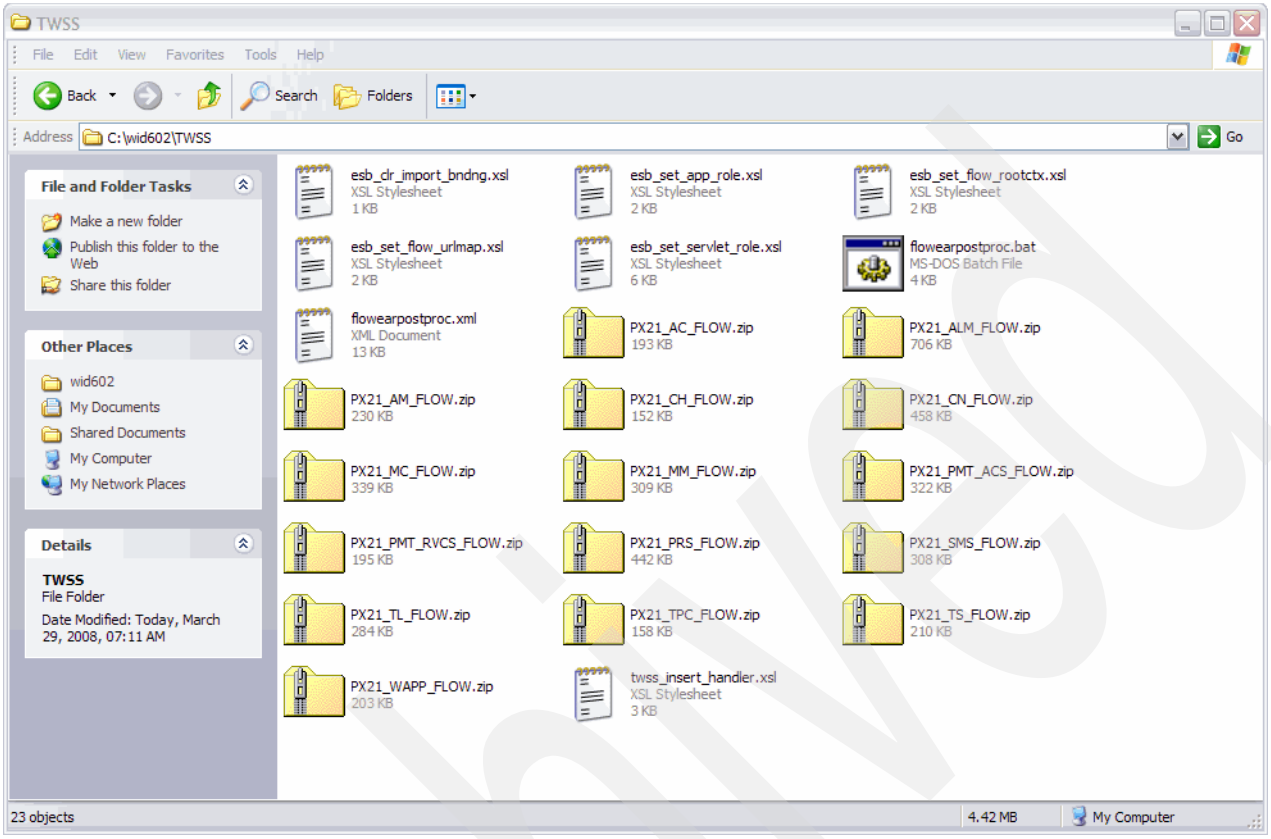


Figure 4-17 Default flows directory

The file names have the form PX21_xx_FLOW.zip, where xx is an identifier of the Parlay X service. For example, the project interchange file for Address List Management (ALM) is PX21_ALM_FLOW.zip.

We will use the existing presence mediation flow (PX21_PRS_FLOW) as a sample and the basis for our sample scenario throughout the remainder of the IBM Redbooks publication. This PX21_PRS_FLOW is shown in Figure 4-18.

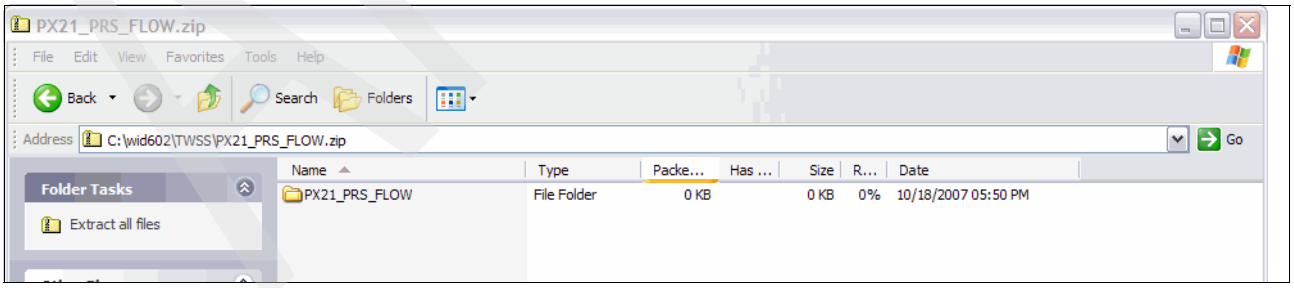


Figure 4-18 File for existing default Presence mediation flow

4.6 Using the default mediation flow in WebSphere Integration Developer

To successfully import and begin using the default Presence mediation flow in WebSphere Integration Developer, perform the following steps:

1. Launch the WebSphere Integration Developer (Figure 4-19).

Figure 4-19 WebSphere Integration Developer splash window

2. Browse to the workspace where you would like to import the default flow mediation flow. Click **OK** (Figure 4-20).

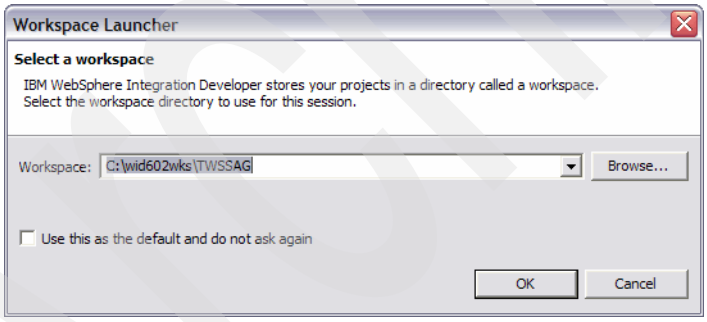


Figure 4-20 Selecting a workspace

Archive

3. Once the WebSphere Integration Developer is opened, it will take us to the default window, which looks similar to Figure 4-21.

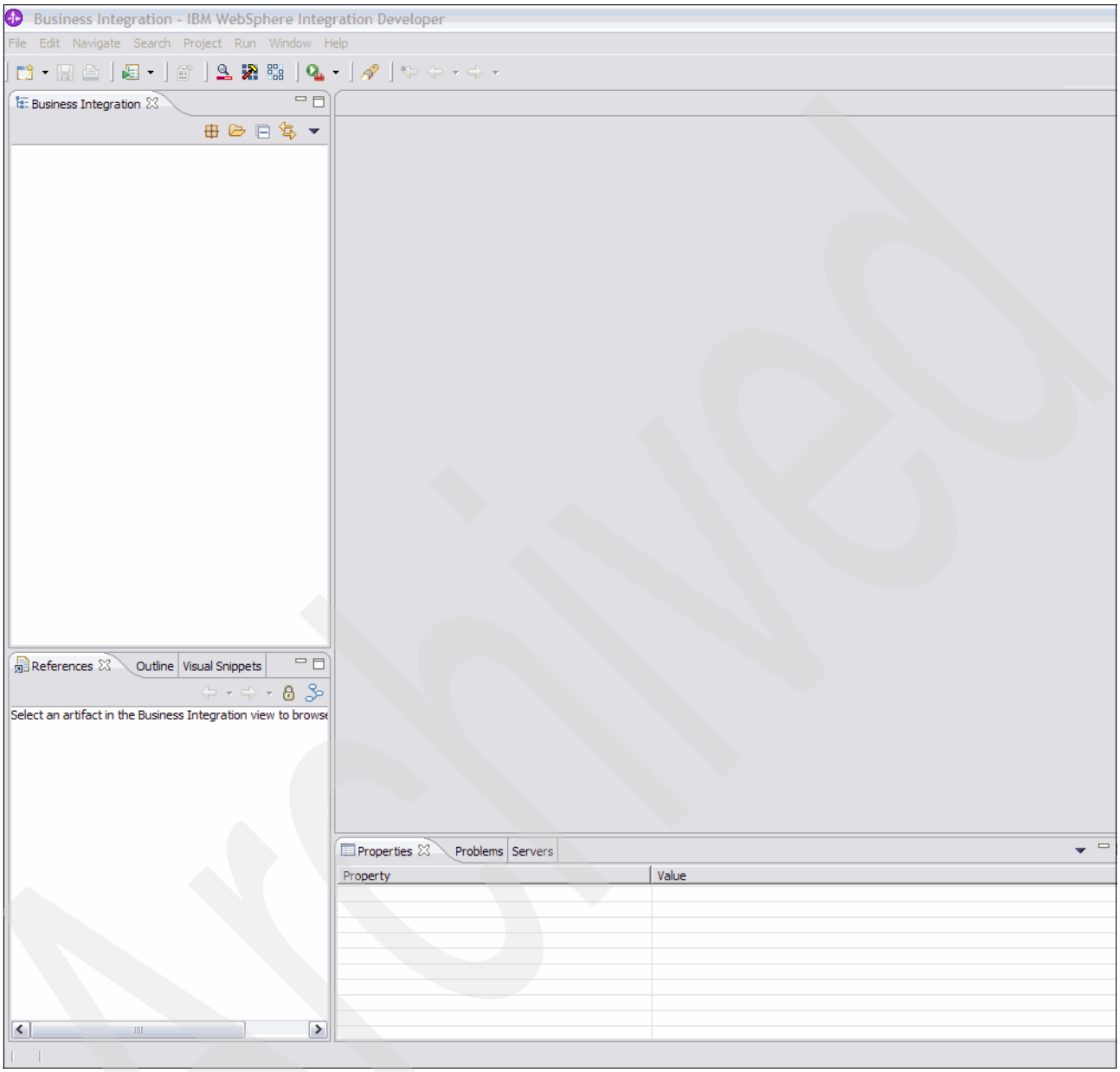


Figure 4-21 Initial workspace for WebSphere Integration Developer

4. As we need to import the default flow mediation, right-click the **Business Integration** perspective, which pops up a window. Select the option **Project Interchange** to import the default mediation flow. Click **Next** (Figure 4-22).

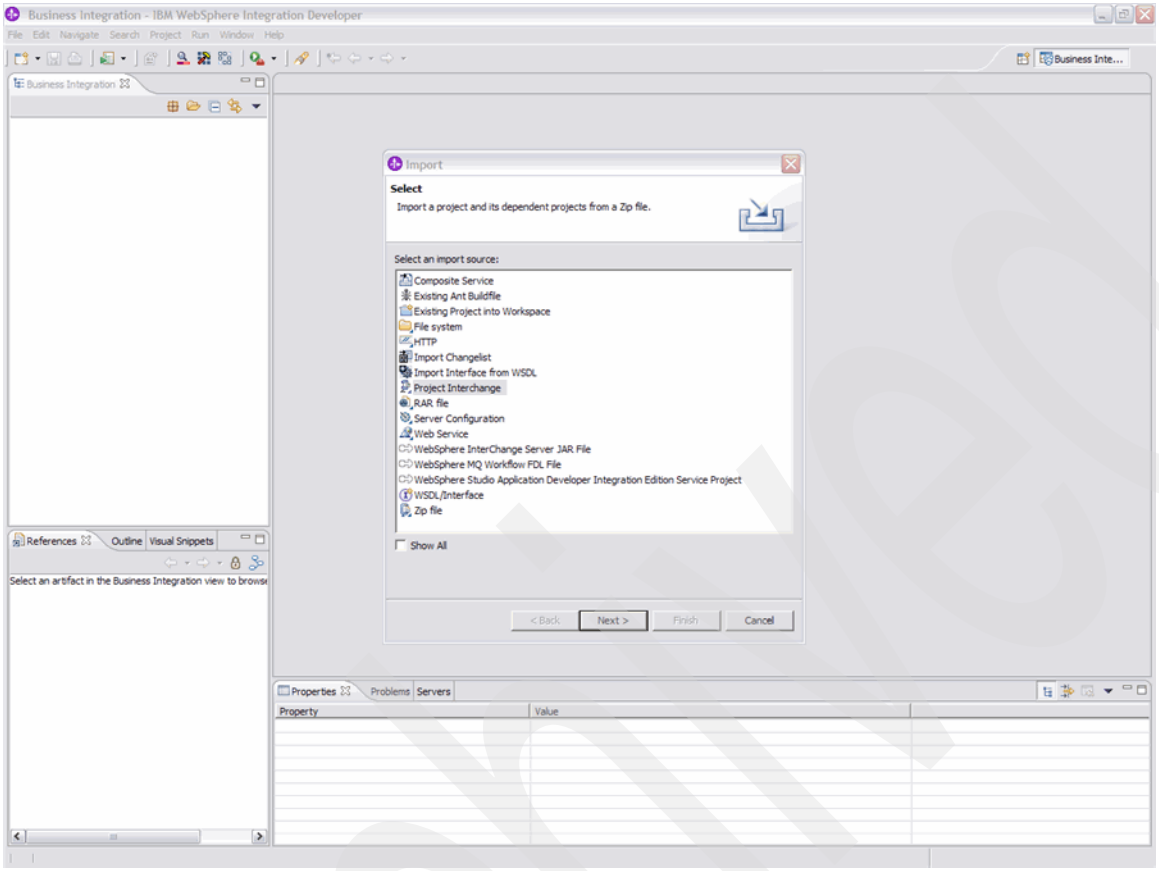


Figure 4-22 Preparing to import the default flow mediation

Archive

5. Click the **Browse** button and navigate to the folder that contains the default mediation flow, for example, C:\<WID_HOME_DIRECTORY>\TWSS\. Click **Finish** (Figure 4-23).

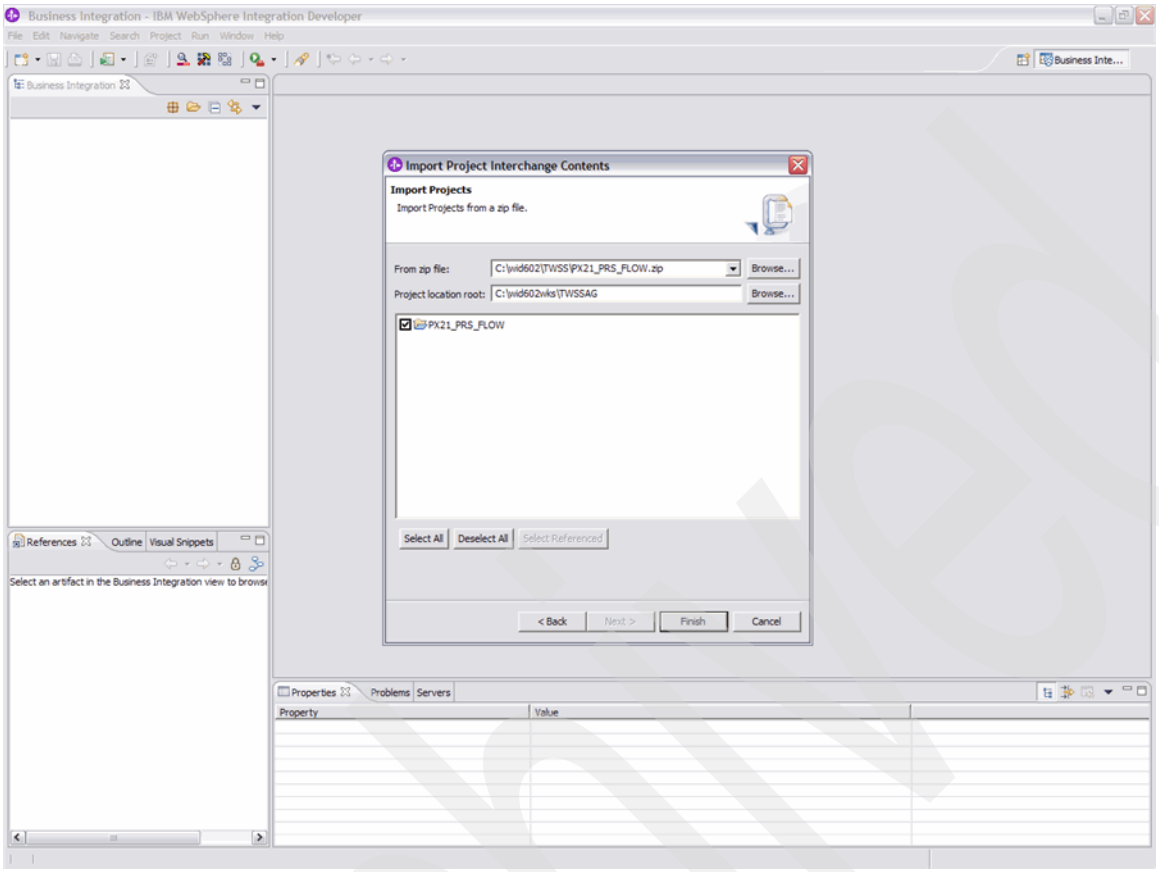


Figure 4-23 Import Project Interchange Contents window

Archive

6. The default mediation flow will open, displaying the Business Perspective, as shown in Figure 4-24.

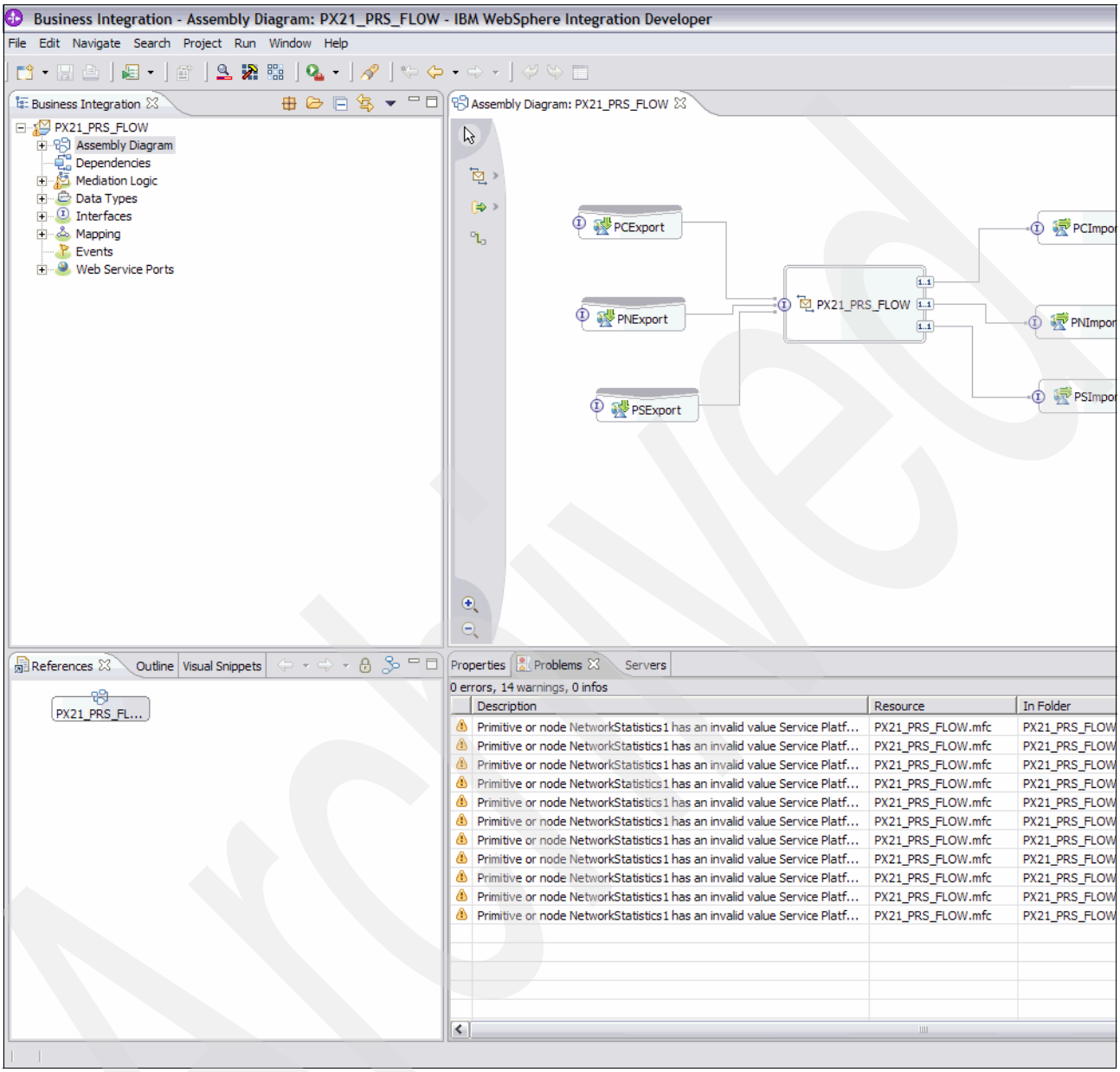


Figure 4-24 WebSphere Integration Developer mediation flow

7. Double-click **PX21_PRS_FLOW**, which will open the mediation flow and expose the visual representation of the Parlay X interfaces (Figure 4-25).

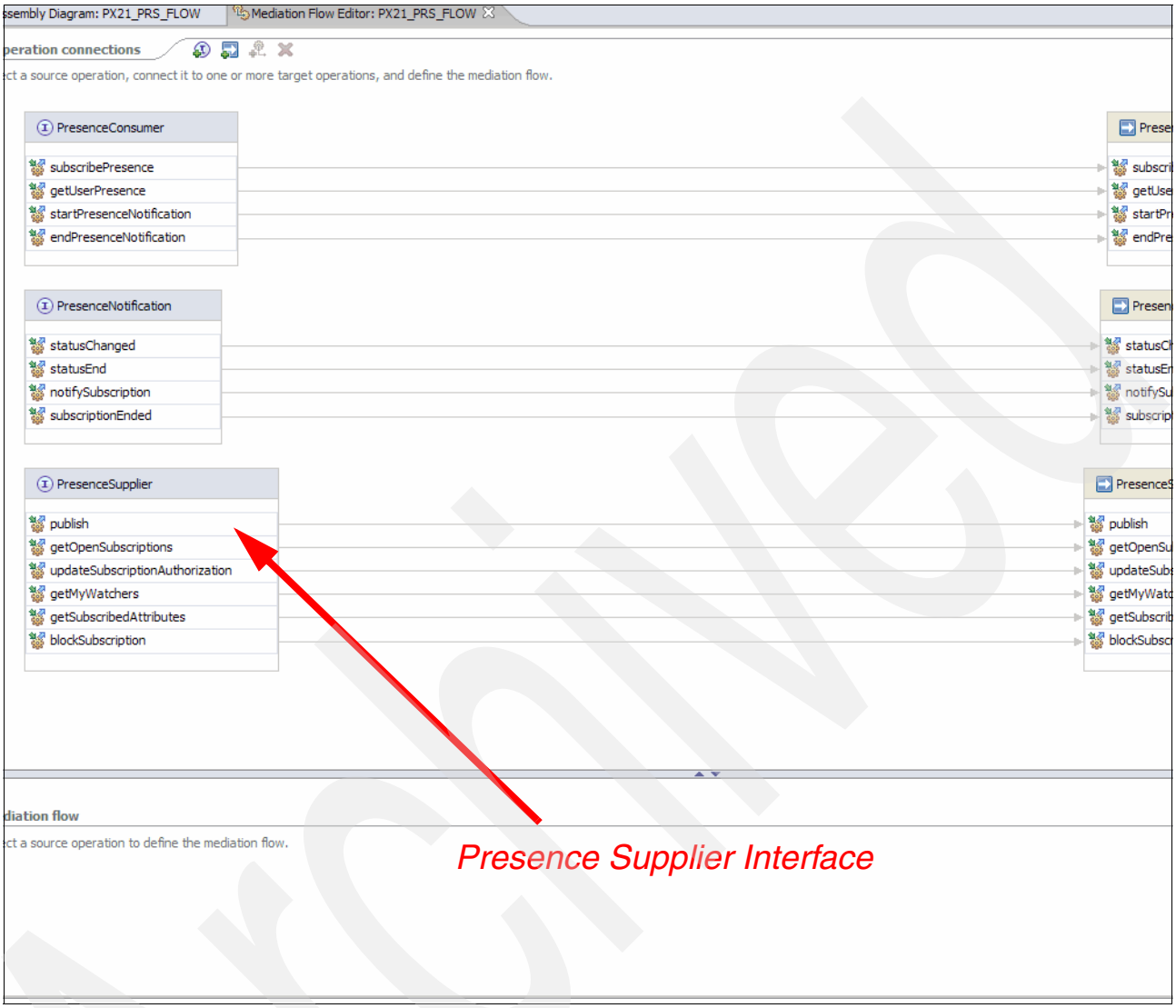


Figure 4-25 Visual representation of the Parlay X interfaces

Once the mediation flow loaded successfully, we need to build the mediation flow within the context of a project.

Building the default mediation flow within the context of a project

1. Select **Project** in the toolbar and click **Build Project**.
2. After the project builds successfully, look at the Problem tab for any errors, as shown in Figure 4-26 on page 105.

> **Note:** In Figure 4-26 on page 105, these are *normal warnings*, not *errors*. Why do we see these at this time? This is an ActivationSpec warning that, during the initial deploy, shows as a warning in the admin console. You can ignore these warnings and move on.

Archived

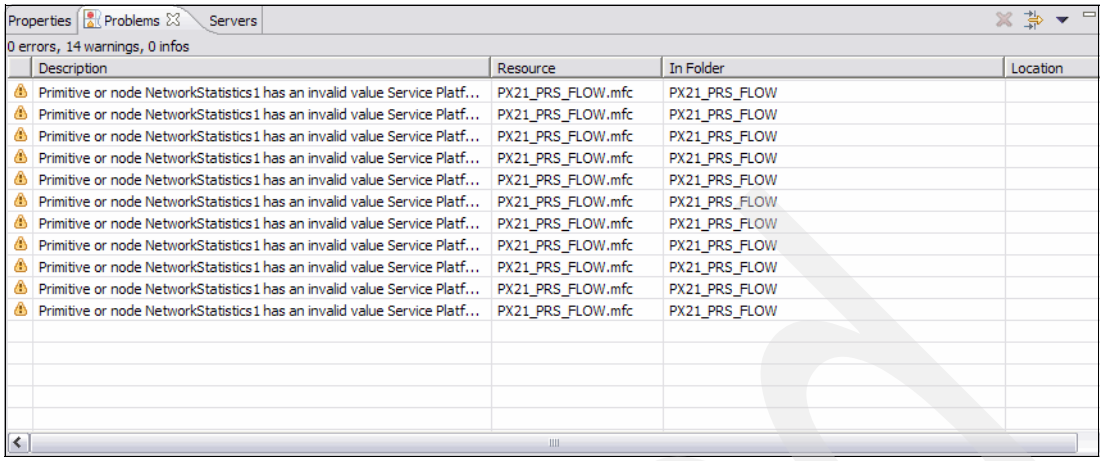


Figure 4-26 Problems reported during project creation

4.6.1 Deploying the default mediation flow

1. Right-click the project **PX21_PRS_FLOW** and select the **Export** feature. Once the Export feature is selected, as list of options appear. Highlight the EAR file and click **Next** (Figure 4-27).

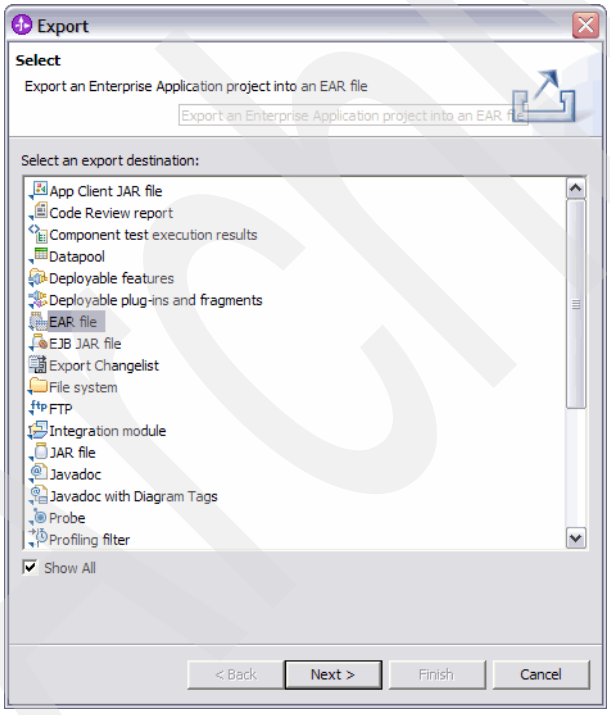


Figure 4-27 Export window

2. Select the EAR Project **PX21_PRS_FLOWApp** from the drop-down menu and browse the destination where you want to keep the EAR, for example, - c:\ PX21_PRS_FLOWApp.ear.

3. Click **Finish** (Figure 4-28).

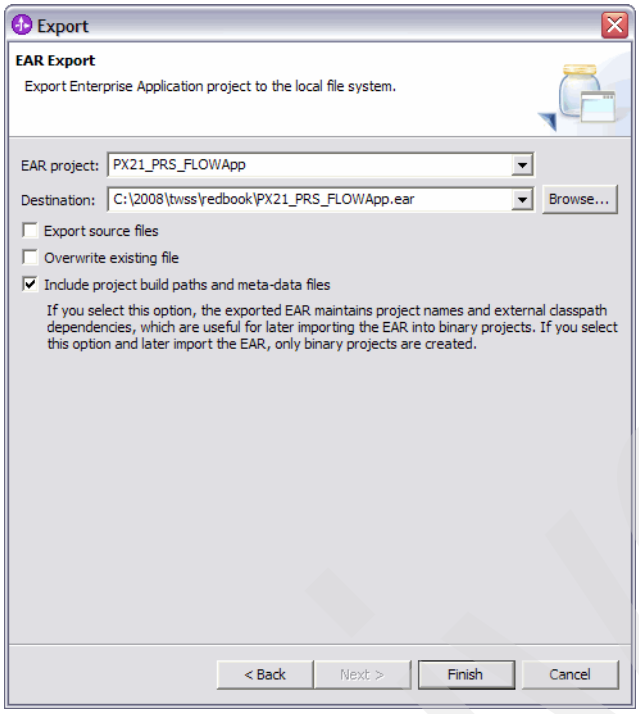


Figure 4-28 EAR Export window

Archived

4. Launch the IBM WebSphere Telecommunications Web Services Server Access Gateway console in the web browser using the address `http://<IP Address> :< Port number>/ibm/console/`. Enter the User ID Admin and click the **Login** button, as shown in Figure 4-29.

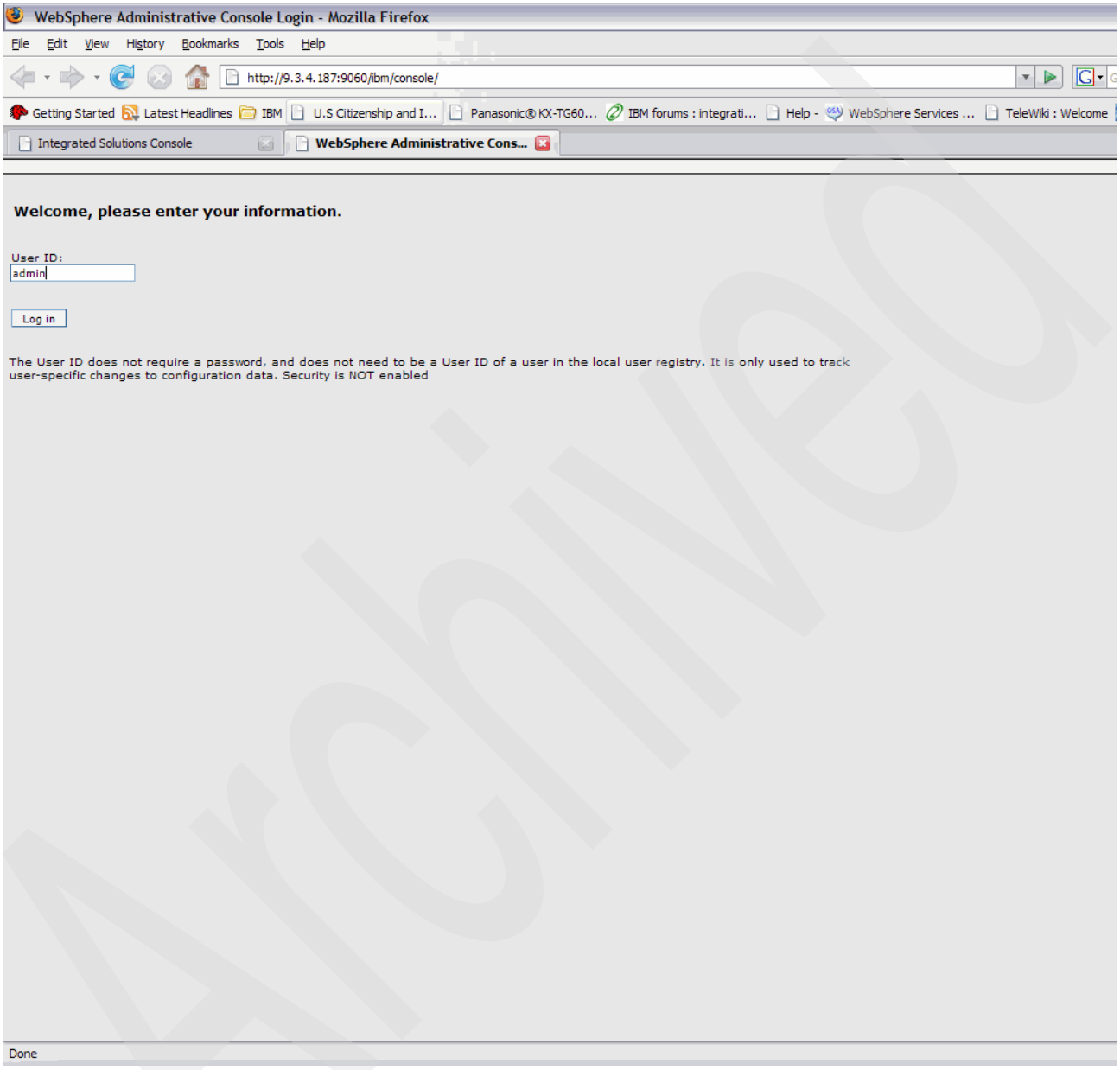


Figure 4-29 Logging into the Administrative Console

Archived

5. Once you have logged in, the IBM WebSphere Telecommunications Web Services Server Console Access Gateway home page will appear, as shown in Figure 4-30.

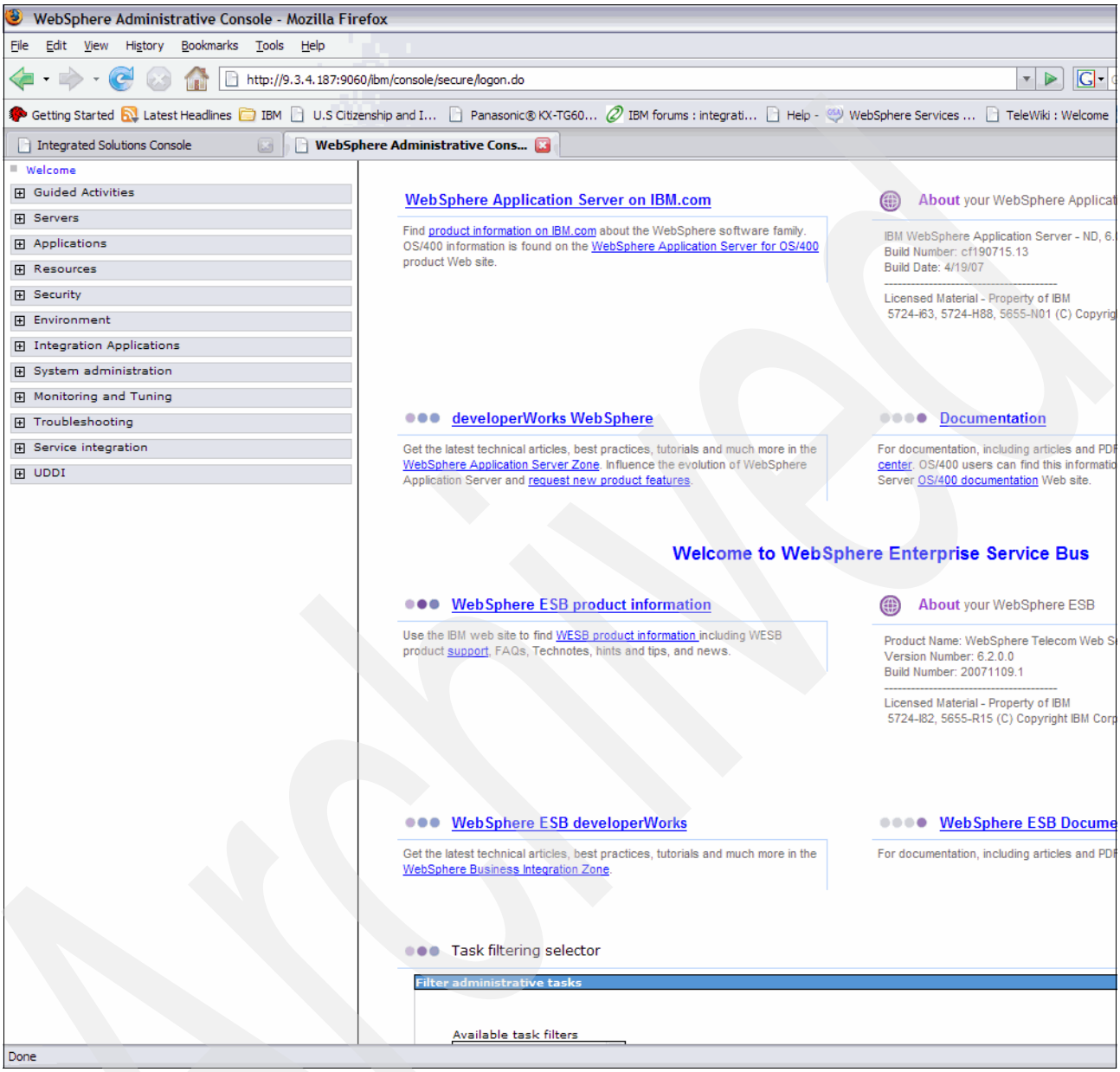


Figure 4-30 IBM WebSphere Telecommunications Web Services Server Console Access Gateway home page

6. Click **Applications** and select **Install New Application**. You will be prompted to select the default mediation flow EAR, which is exported from WebSphere Integration Developer (Figure 4-31 on page 109).

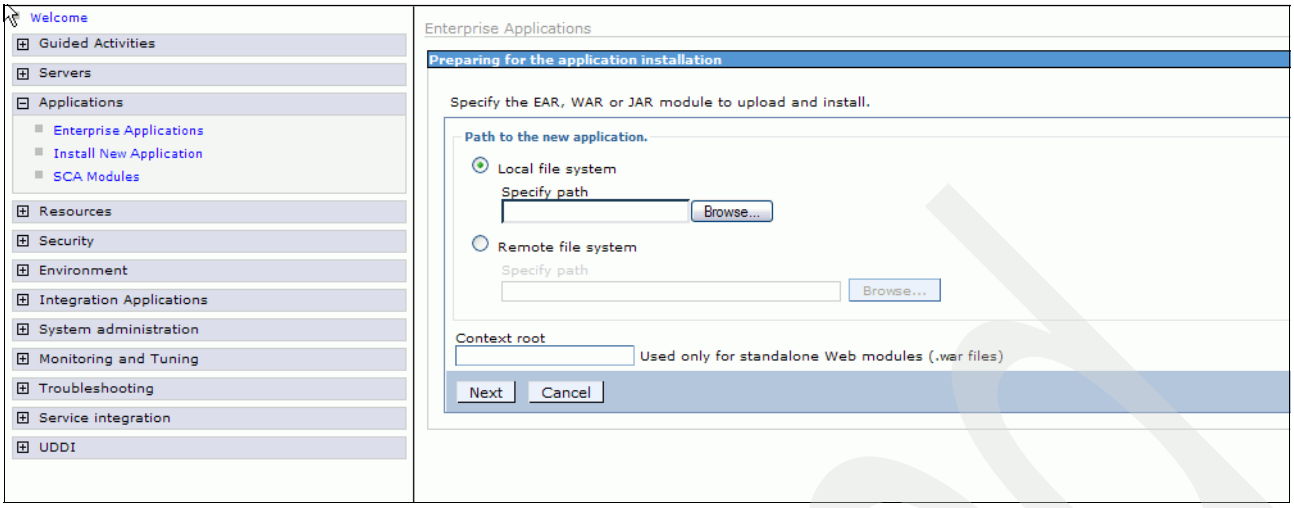


Figure 4-31 Enterprise Applications window

7. Select the **Local File system** radio button and click **Browse** to navigate the meditation flow EAR. Click **Next** (Figure 4-32).

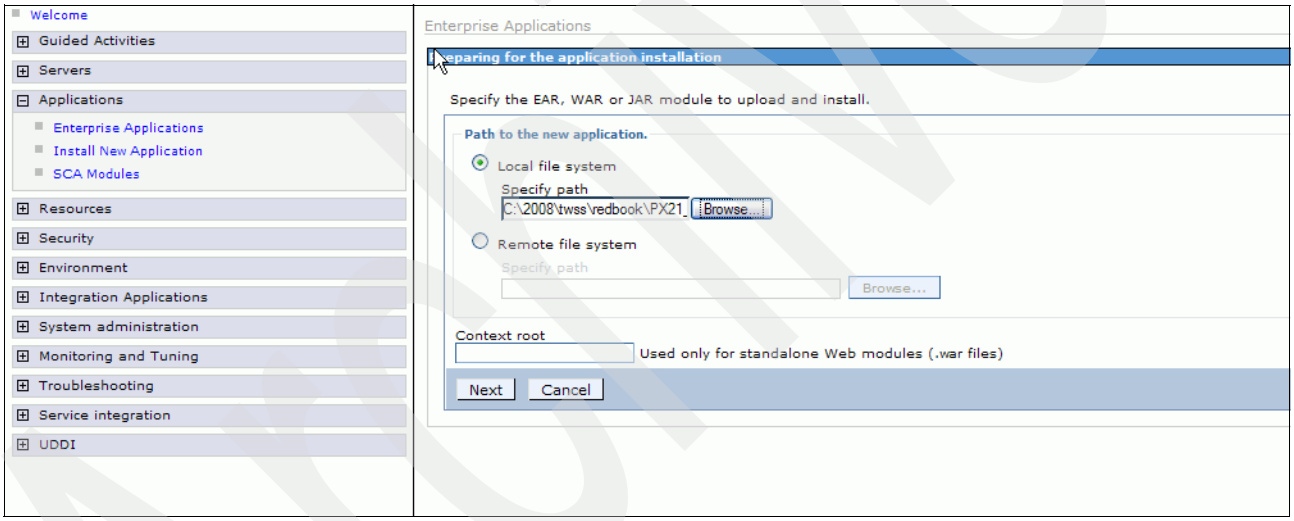


Figure 4-32 Preparing for the application installation

8. In this window, you can specify several general options that determine how WebSphere Application Server installs your application. Specifically, it allows you to:
   - Specify a prefix for beans.
   - Select whether to override default bindings.
   - Specify whether to use a default host name for Web modules.

Leave the default options as is and click **Next** (Figure 4-33).

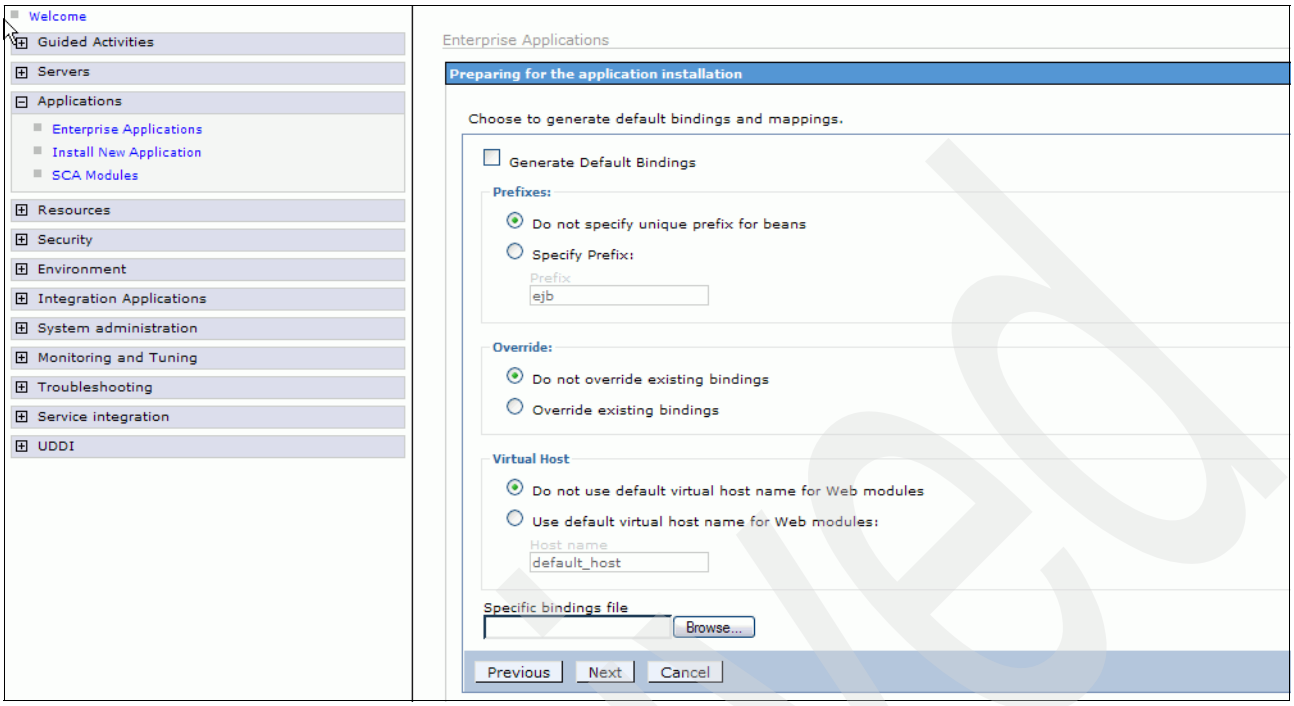


Figure 4-33 Preparing for the application installation window

9. In this next step, you have the option to select installation option settings. Use the default options and click **Next**, as shown in Figure 4-33. Meanwhile, some explanation about the different options here is helpful.

> **Note:** Table 4-14 provides a description of the installation options and how these affect the installation. Use the default options and click **Next**, as shown in Figure 4-33.

Use this window to specify options for the installation of an application onto a WebSphere Application Server deployment target. The default values for the options are used if you do not specify a value. After application installation finishes, you can specify values for many of these options from an enterprise application settings page.

The Select installation options window is the same for the application installation and update wizards.

Installation options are described in Table 4-14.

Table 4-14 Description of installation options

Installation option	Description	Notes
Pre-compile JSP	Specify whether to precompile JavaServer Pages (JSP) files as a part of installation. The default is not to precompile JSP files.	Data type - Boolean Default - false

Installation option	Description	Notes
Distribute application	Specifies whether the product expands application binaries in the installation location during installation and deletes application binaries during uninstallation. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified.	► On *single-server* products, the binaries are deleted when you uninstall and save changes to the configuration. ► On *multiple-server* products, the binaries are deleted when you uninstall and save changes to the configuration and synchronize changes. ► If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs. ► Important: If you disable this option and you do not copy and expand the application binaries to the nodes, a later saving of the configuration or manual synchronization does not move the application binaries to the nodes for you. Data type - Boolean Default - false
Use binary configuration	Specifies whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the deployment.xml file (default), or those located in the enterprise application resource (EAR) file. This Use binary configuration field is the same as the Use metadata from binaries setting on an Enterprise Application settings page.	Data type - Boolean Default - false

Archive

Installation option	Description	Notes
Deploy enterprise beans	Specifies whether the EJBDeploy tool runs during application installation. The tool generates code needed to run enterprise bean (EJB) files.	You must enable this setting in the following situations: ► The EAR file was assembled using an assembly tool such as Rational Application Developer or WebSphere Integration Developer and the EJBDeploy tool was not run during assembly. ► Enabling this setting might cause the installation program to run for several minutes. Data type - Boolean Default-true
Application name	Specifies a logical name for the application. An application name must be unique within a cell and cannot contain an disallowed character. This Application name field is the same as the Name setting on an Enterprise application settings page.	An application name cannot begin with a period (.), cannot contain leading or trailing spaces, and cannot contain non-alphabetic characters, such as “/”, “$”, “*”, “%”, and so on. Data type - String
Create MBeans for resources	Specifies whether to create MBeans for resources such as servlets or JSP files within an application when the application starts. The default is to create MBeans.	Data type - Boolean Default - true
Enable class reloading	Specifies whether to enable class reloading when application files are updated.	The default is not to enable class reloading. Data type - Boolean Default - false
Reload interval in seconds	Specifies the number of seconds to scan the application's file system for updated files. The default is the value of the reloading interval attribute in the IBM extension (META-INF/ibm-application-ext.xmi) file of the EAR file.	► The reloading interval attribute takes effect only if class reloading is enabled. ► To enable reloading, specify a value greater than zero (for example, 1 to 2147483647). To disable reloading, specify zero (0). The range is from 0 to 2147483647. ► This Reload interval in seconds field is the same as the Reloading interval setting on an Enterprise Application settings page. Data type - Integer Unit - Seconds Default - 3

Installation option	Description	Notes
Deploy Web services	Specifies whether the Web services deploy tool wsdeploy runs during application installation. The tool generates code needed to run applications using Web services.	The default is not to run the wsdeploy tool. You must enable this setting if the EAR file contains modules using Web services and has not previously had the wsdeploy tool run on it, either from the Deploy menu choice of an assembly tool or from a command line. Data type - Boolean Default -false
Validate input off/warn/fail	Specifies whether WebSphere Application Server examines the application references specified during application installation or updating and, if validation is enabled, warns you of incorrect references or fails the operation. An application typically refers to resources using data sources for container managed persistence (CMP) beans or using resource references or resource environment references defined in deployment descriptors. The validation checks whether the resource referred to by the application is defined in the scope of the deployment target of that application.	Select off for no resource validation, warn for warning messages about incorrect resource references, or fail to stop operations that fail as a result of incorrect resource references. This Validate input off/warn/fail field is the same as the Validation setting on an Enterprise Application settings page. Data type - String Default - warn
Process embedded configuration	Specifies whether the embedded configuration should be processed. An embedded configuration consists of files such as resource.xml and variables.xml.	When selected or true, the embedded configuration is loaded to the application scope from the .ear file. If the .ear file does not contain an embedded configuration, the default is false. If the .ear file contains an embedded configuration, the default is true. Data type - Boolean Default - false

Archive

Use the default options and click **Next**, as shown in Figure 4-34.

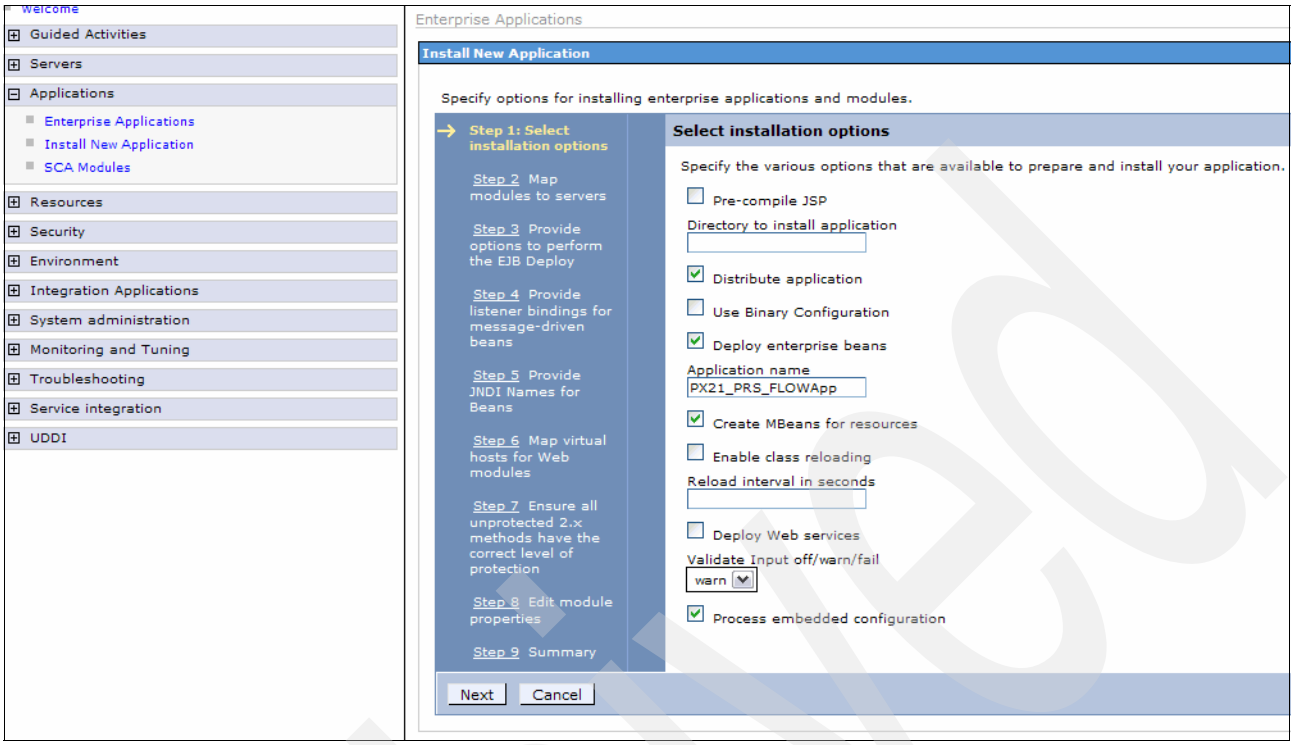


Figure 4-34 Installation options window

10. In this next step, we focus on selecting servers and mapping modules to the servers' settings. This window allows you to specify deployment targets where you want to install the modules contained in your application. Modules can be installed on the same deployment target or dispersed among several deployment targets:

    – On single-server products, a deployment target can be an application server or Web server.

    – On multiple-server products, a deployment target can be an application server, cluster of application servers, or Web server.

    To view this administrative console window, select **Applications** → **Enterprise Applications** → **application_name** → **Map modules to servers**. This window is similar to the Map modules to servers window on the application installation and update wizards.

    In this window, each *module* must map to one or more desired targets identified under *Server*.

    Details on how to change the settings are described in Table 4-15 on page 115.

Archived

Table 4-15 Details for selecting servers and mapping modules to servers' settings

Function	Steps	Notes
To change a mapping	► In the list of mappings, select the **Select** check box beside each module that you want mapped to the same target(s). ► From the Clusters and Servers drop-down list, select one or more targets. Select only appropriate deployment targets for a module. Modules that use WebSphere Application Server Version 6.x features cannot be installed onto a Version 5.x target server. ► Click **Apply**.	Use the Ctrl key to select multiple targets. For example, to have a Web server serve your application, press the Ctrl key and then select an application server and the Web server together. The plug-in configuration file plug-in-cfg.xml for that Web server will be generated based on the applications that are routed through it.
Clusters and servers	Lists the names of available deployment targets. This list is the same for every application that is installed in the cell. From this list, select only appropriate deployment targets for a module. You can install an application, enterprise bean (EJB) module or Web module developed for a Version 5.x product on a 5.x or 6.x deployment target. If the module supports J2EE V1.4, calls a V6.x API, or uses a V6.x feature, then you must install the module on a V6.x deployment target.	► Does not support Java 2 Platform, Enterprise Edition (J2EE) V1.4. ► Does not call any V6.x runtime application programming interfaces (APIs). ► Does not use any V6.x product features.
Module	Specifies the name of a module in the installed (or deployed) application.	►
URI	Specifies the location of the module relative to the root of the application (EAR file).	►
Server	Specifies the name of each deployment target to which the module currently is mapped.	To change the deployment targets for a module, select one or more targets from the Clusters and Servers drop-down list and click **Apply**. The new mapping replaces the previous mapping.

Archive

Click **Select All Items** and click the **Apply** button. Later, the Server field will be updated. Click **Next** to continue, as shown in Figure 4-35.

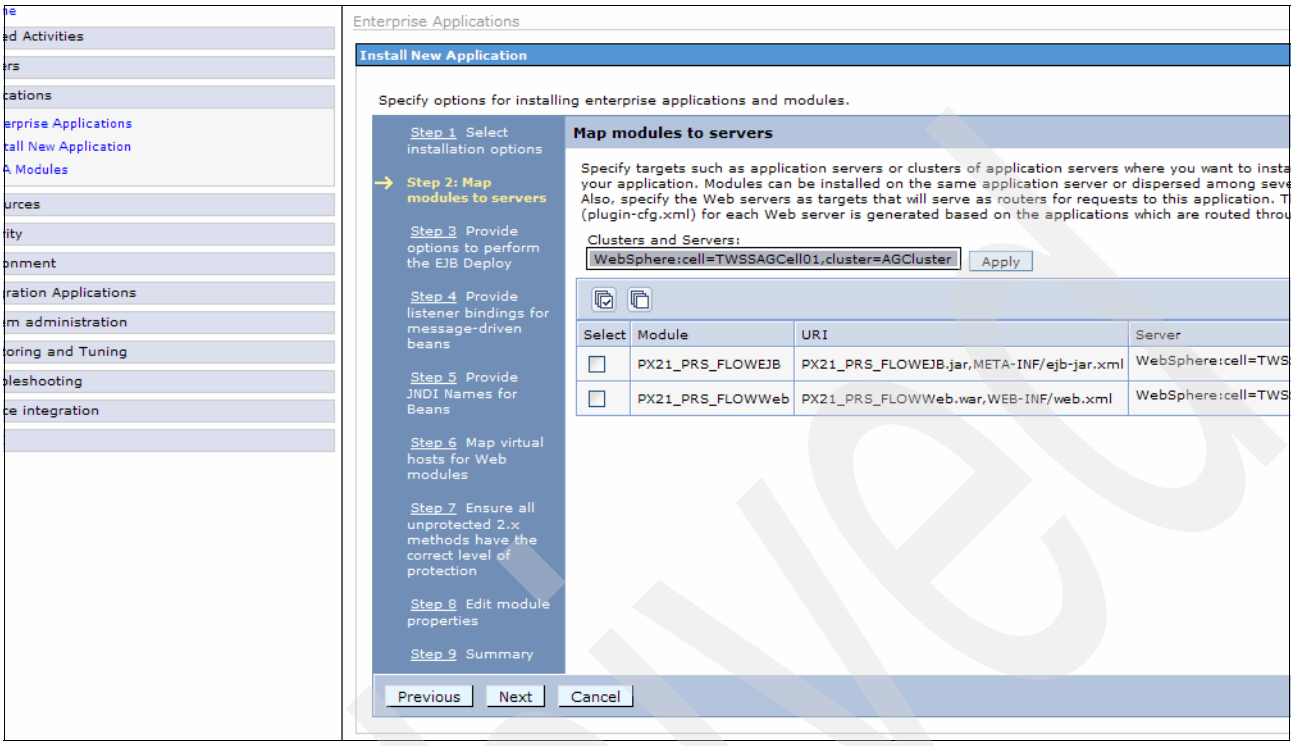


Figure 4-35 Specify deployment targets where you want to install the modules

11. This next window provides options to perform the EJB Deploy. If the Deploy enterprise bean setting is enabled on the Select installation options window, then you can specify options for the EJB deployment tool on the Provide options to perform the EJB Deploy window. In this window, you can specify extra class paths, RMIC options, database types, and database schema names to be used while running the EJB deployment tool. The tool is run on the EAR file during installation after you click **Finish**. Refer to man page for the **`ejbdeploy`** command for more information about this topic.

    Use the default values and click **Next**, as shown in Figure 4-36 on page 117.

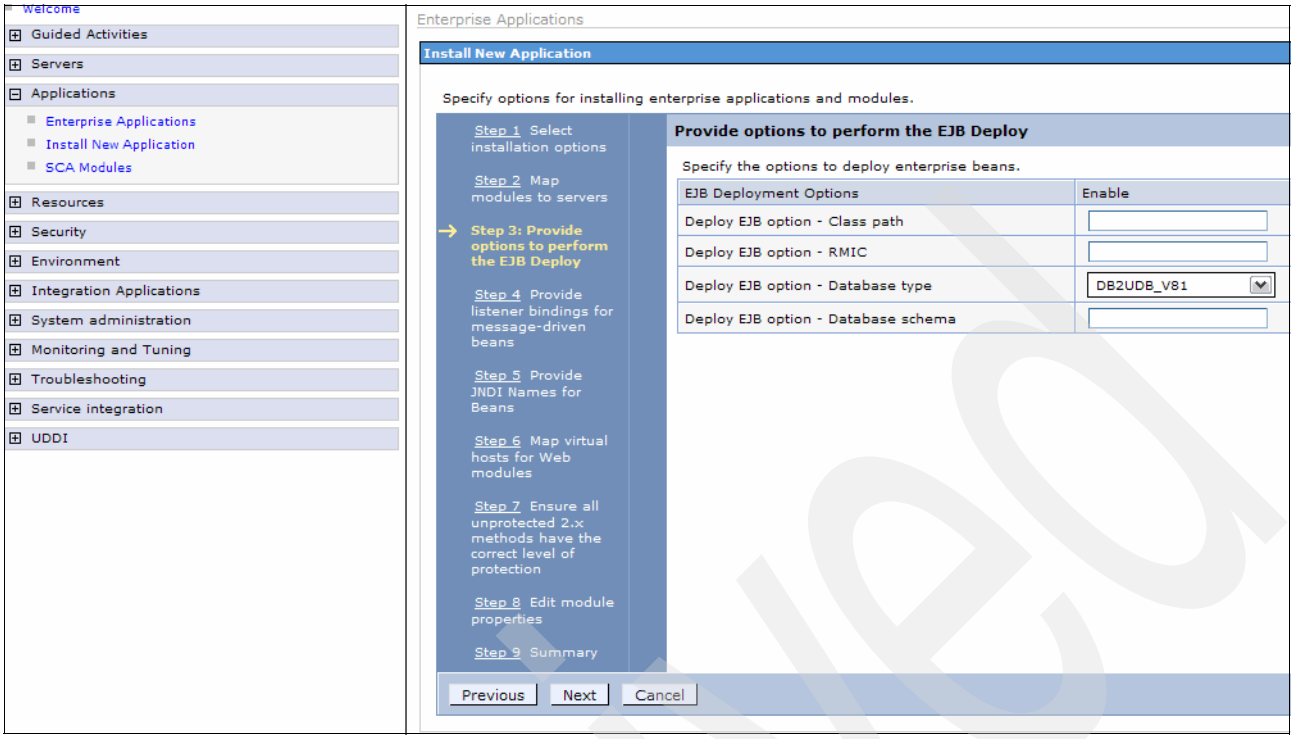


Figure 4-36 Provide options to perform the EJB Deploy

12. Figure 4-37 on page 118 provides listener bindings for message-driven beans.

   – Message-driven beans

     For each message-driven bean, you must specify a queue or topic to which the bean will listen. A message-driven bean is invoked by a Java Messaging Service (JMS) listener when a message arrives on the input queue that the listener is monitoring. A deployer specifies a listener port or JNDI name of an activation specification as defined in a connector module (.rar file) under WebSphere Bindings on the Beans page of an assembly tool EJB deployment descriptor editor. An example JNDI name for a listener port used by a Store application might be StoreMdbListener. The binding definition is stored in IBM bindings files such as ibm-ejb-jar-bnd.xmi. If a deployer chooses to generate default bindings when installing the application, the install wizard assigns JNDI names to incomplete bindings.

     - For EJB V2.x message-driven beans deployed as JCA V1.5-compliant resources, the install wizard assigns JNDI names corresponding to activationSpec instances in the form eis/MDB_ejb-name.
     - For EJB V2.x message-driven beans deployed against listener ports, the listener ports are derived from the message-driven bean <ejb-name> tag with the string Port appended.

   – During application installation using the administrative console, you can specify a listener port name or an activation specification JNDI name for every message-driven bean on the Provide Listener Ports or activation specification JNDI name for messaging beans panel. A listener port name must be provided when using the JMS providers: Version 5 default messaging, WebSphere MQ, or generic. An activation specification must be provided when the application's resources are configured using the default messaging provider or any generic J2C resource adapter that supports inbound messaging. If neither is specified, then a validation error is displayed after you click **Finish** in the Summary window. Also, if the module containing the

message-driven bean is deployed on a V5.x deployment target and a listener port is not specified, then a validation error is displayed after you click **Next**.

– After application installation, you can specify JNDI names and configure message-driven beans on console pages by selecting **Resources** → **JMS** → **JMS Providers** or under **Resources** → **Resource Adapters**.

Use the default values and click **Next** (Figure 4-37).

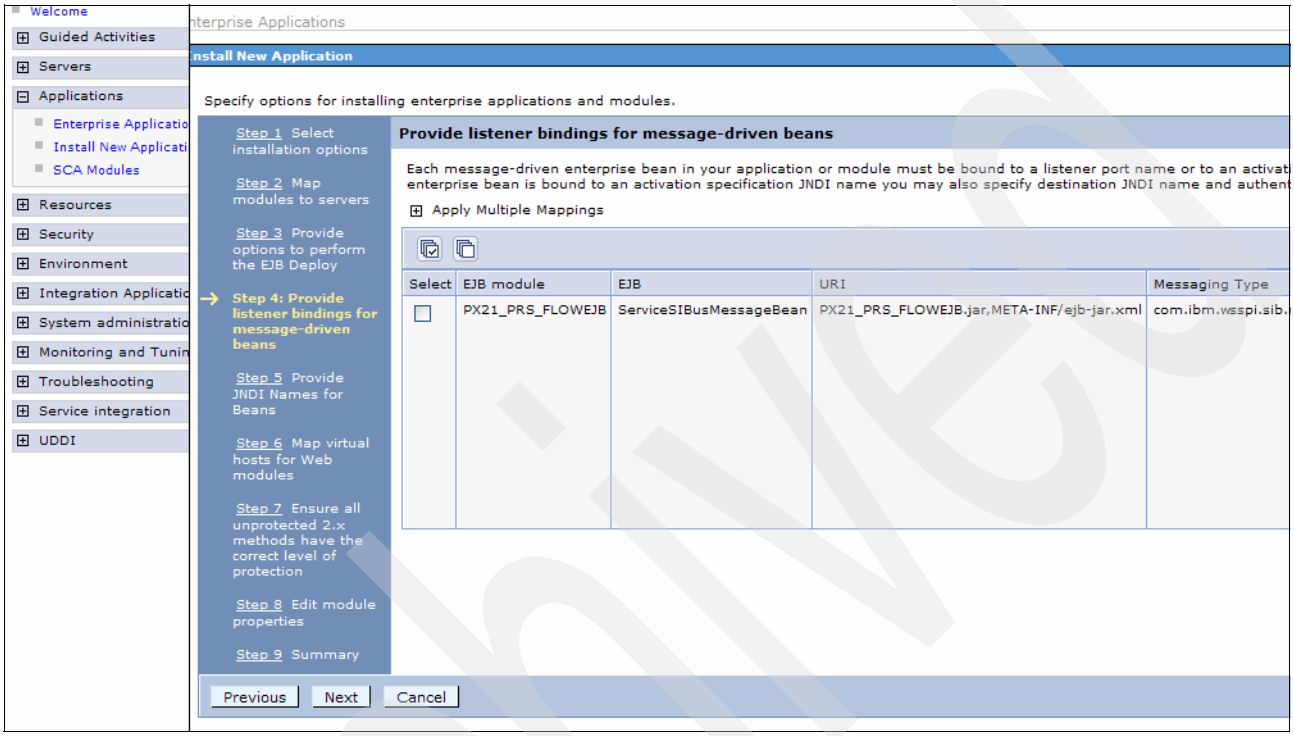


Figure 4-37 Provide listener bindings for message-driven beans

13. The next window alerts you to any application Application Resource warnings. Use the default values and click **Continue** (Figure 4-38).

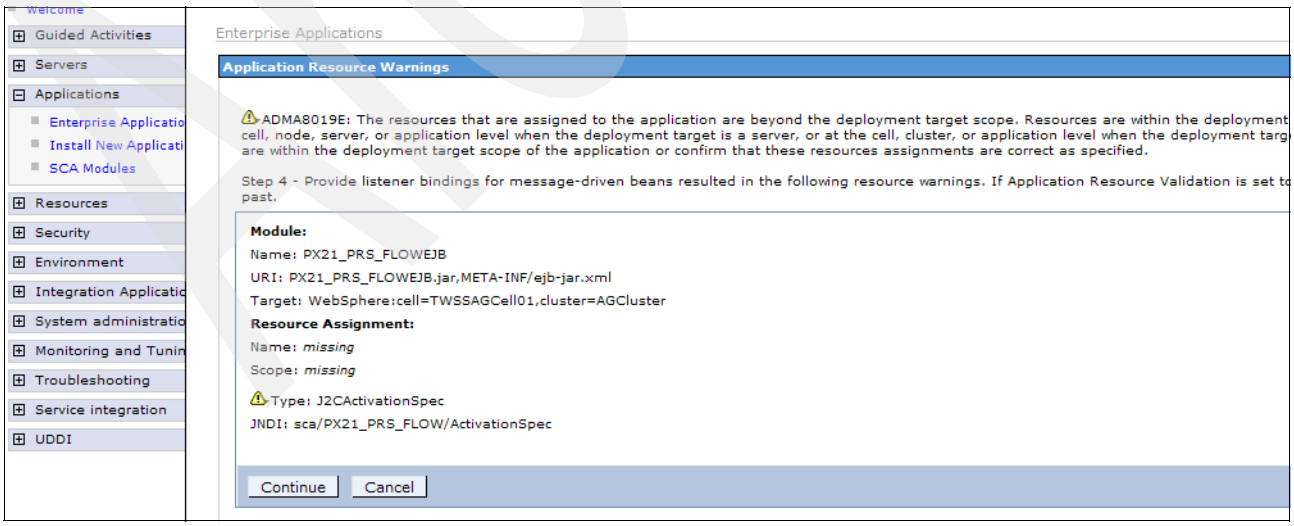


Figure 4-38 Application Resource Warnings

14. Leave the default values and click **Next** (Figure 4-39).

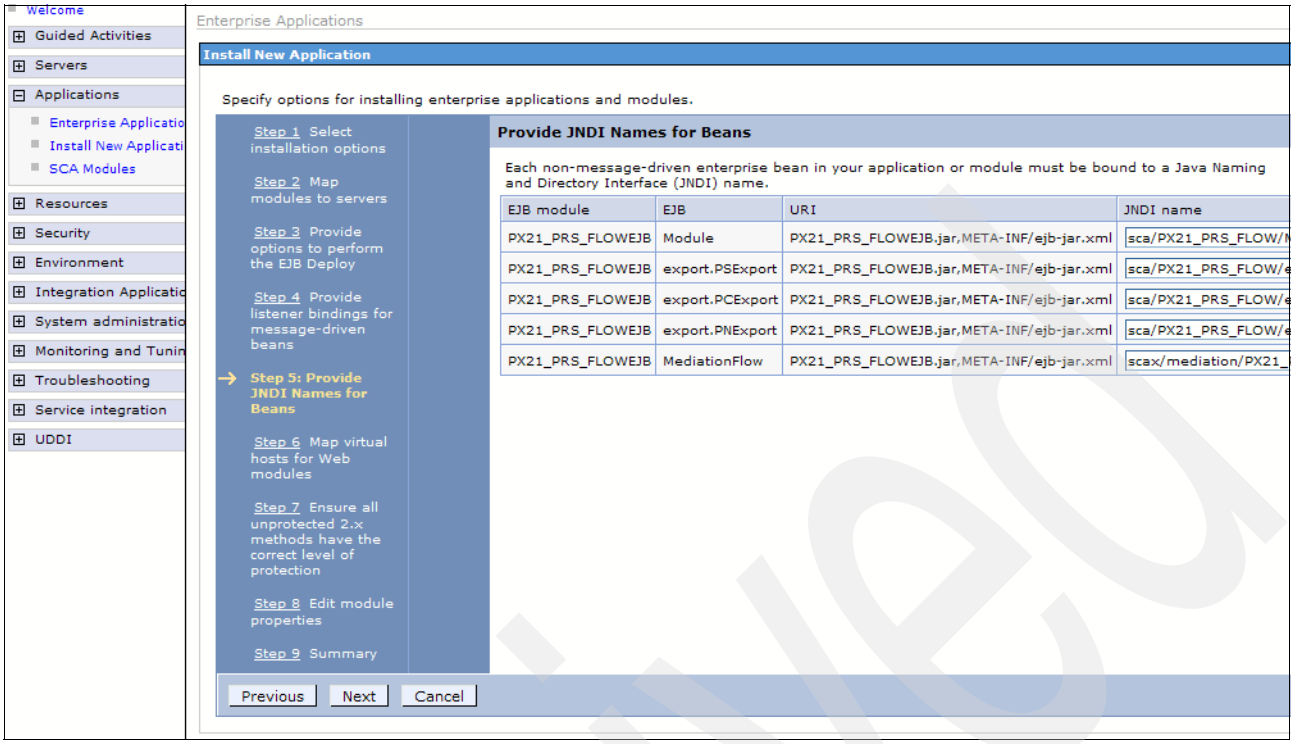


Figure 4-39 Install New Application

15. Use the default values and click **Next** (Figure 4-40).

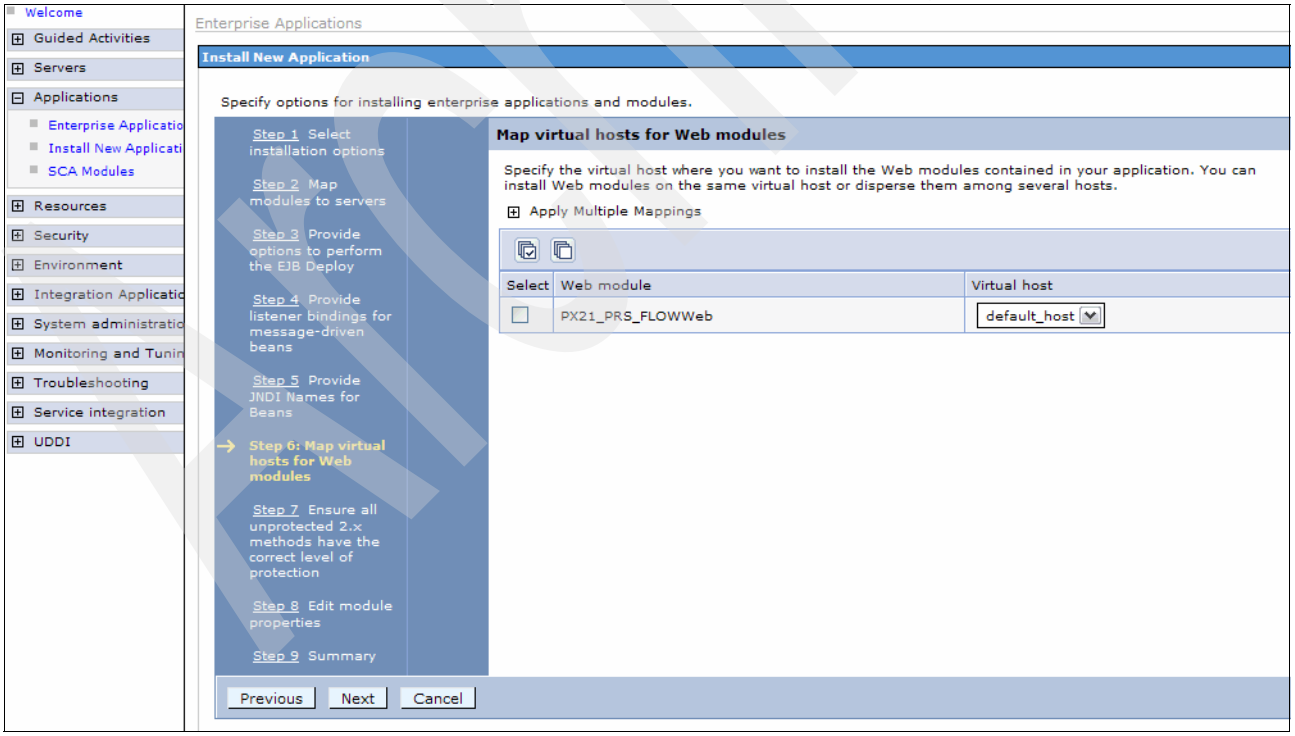


Figure 4-40 Map virtual hosts for Web modules

16. Use the default values and click **Next** (Figure 4-41).

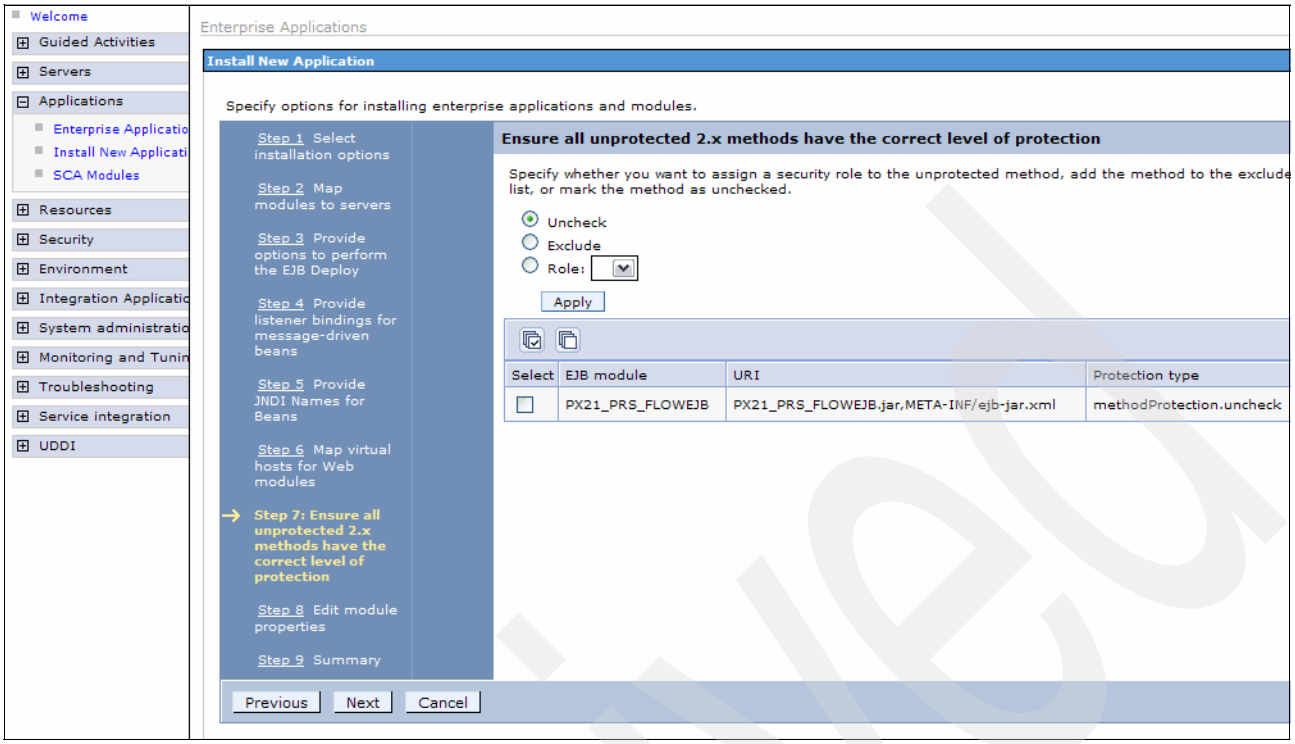


Figure 4-41 Ensure all unprotected 2.x methods have the correct level of protection

17. Use the default values and click **Next** (Figure 4-42 on page 120).

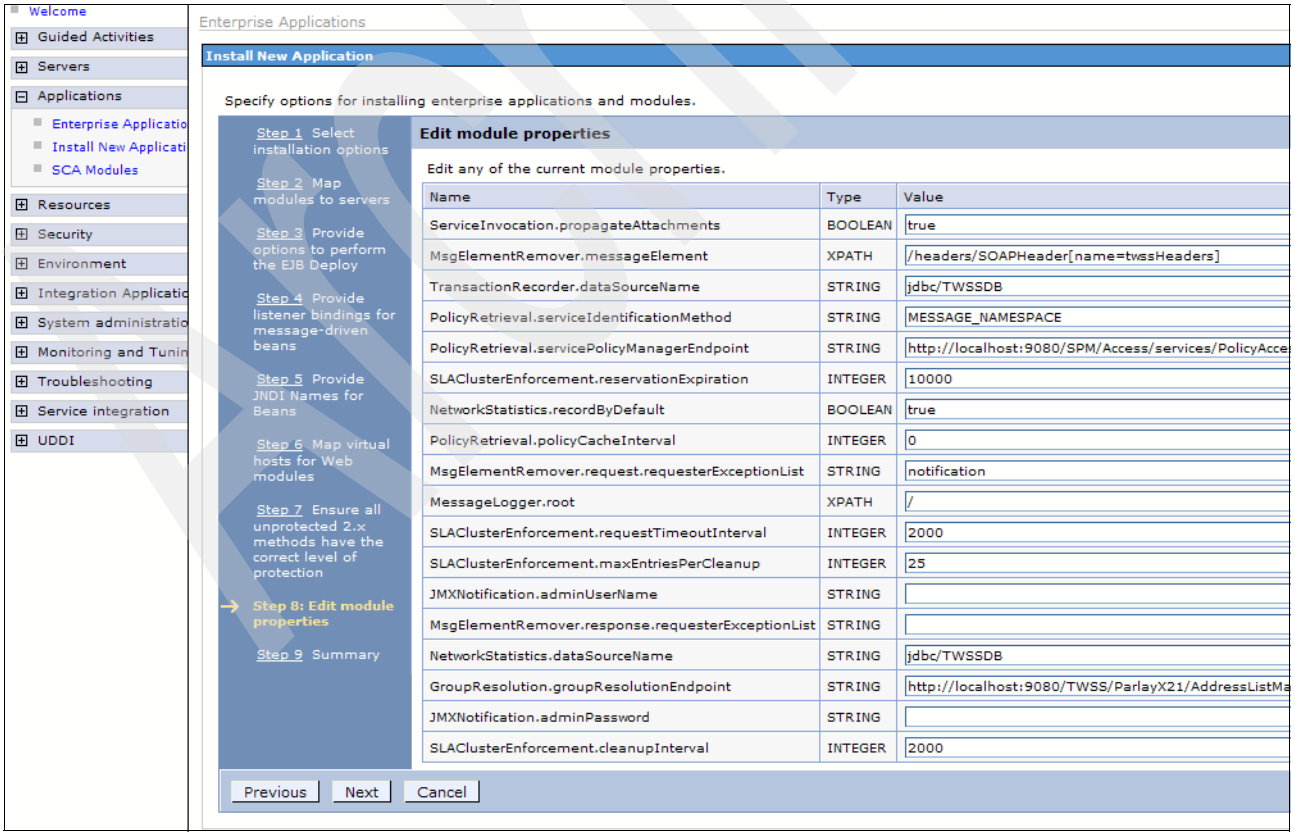


Figure 4-42 Edit module properties

18. Use the default values and click **Finish** (Figure 4-43).

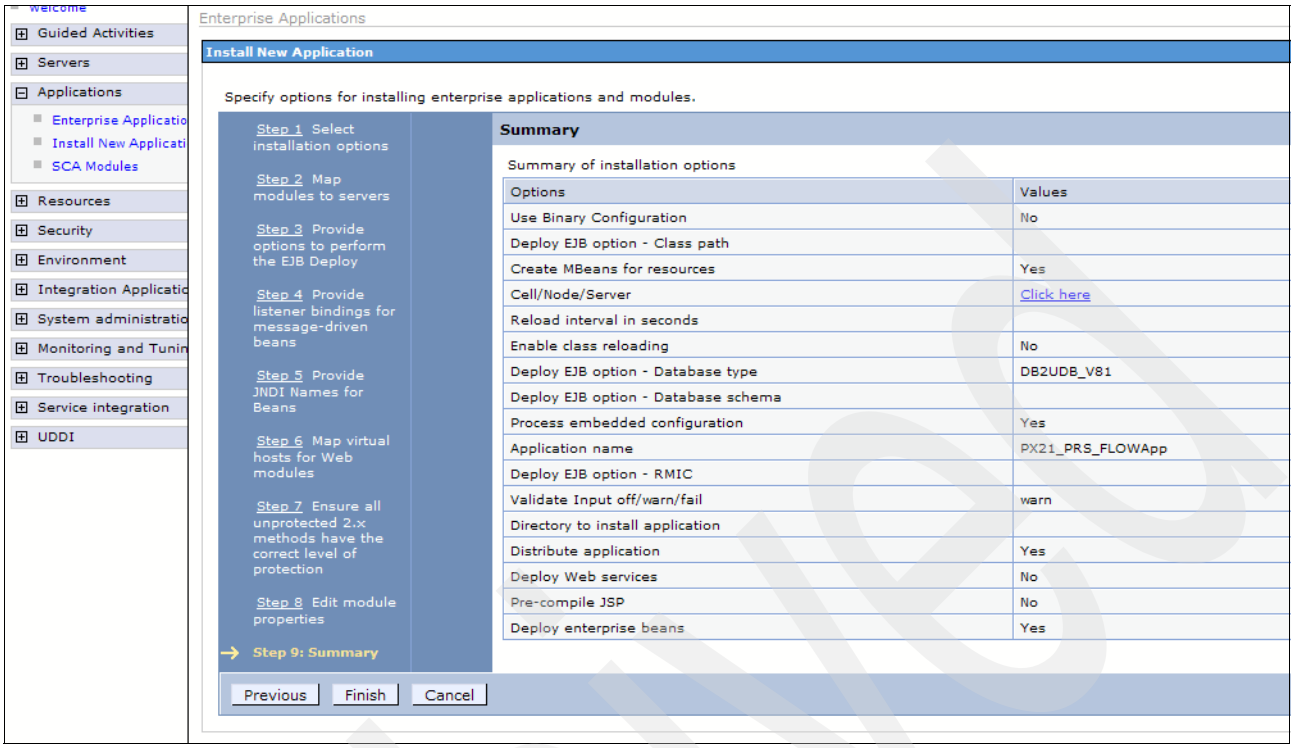


Figure 4-43 Summary

19. The default mediation flow will be installed, as shown in Figure 4-44.

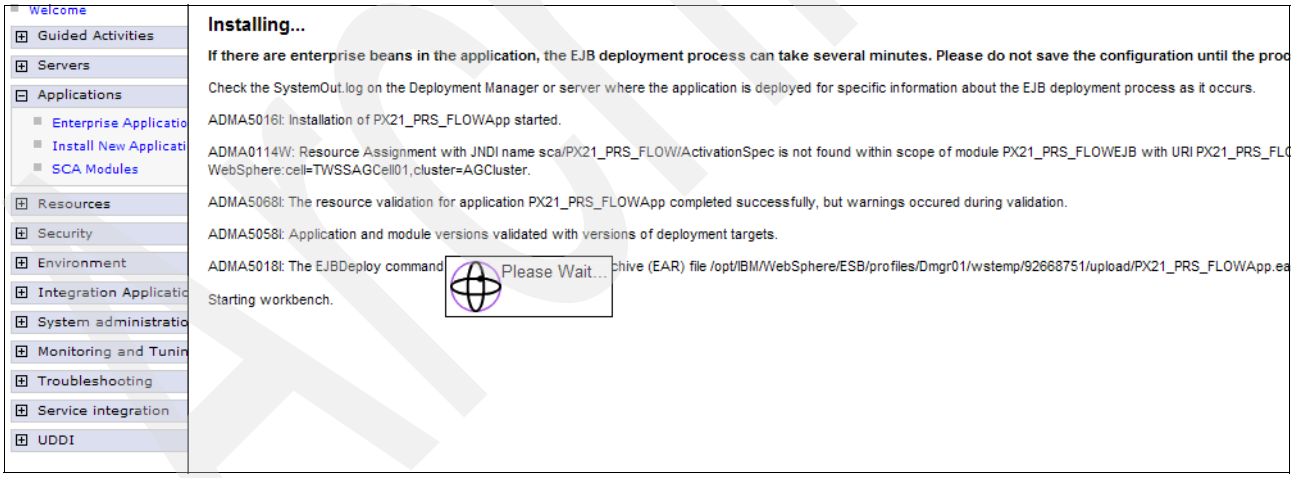


Figure 4-44 Default mediation flow installation

20. Once the default mediation flow application installs successfully, click **Save to Master Configuration** (Figure 4-45).

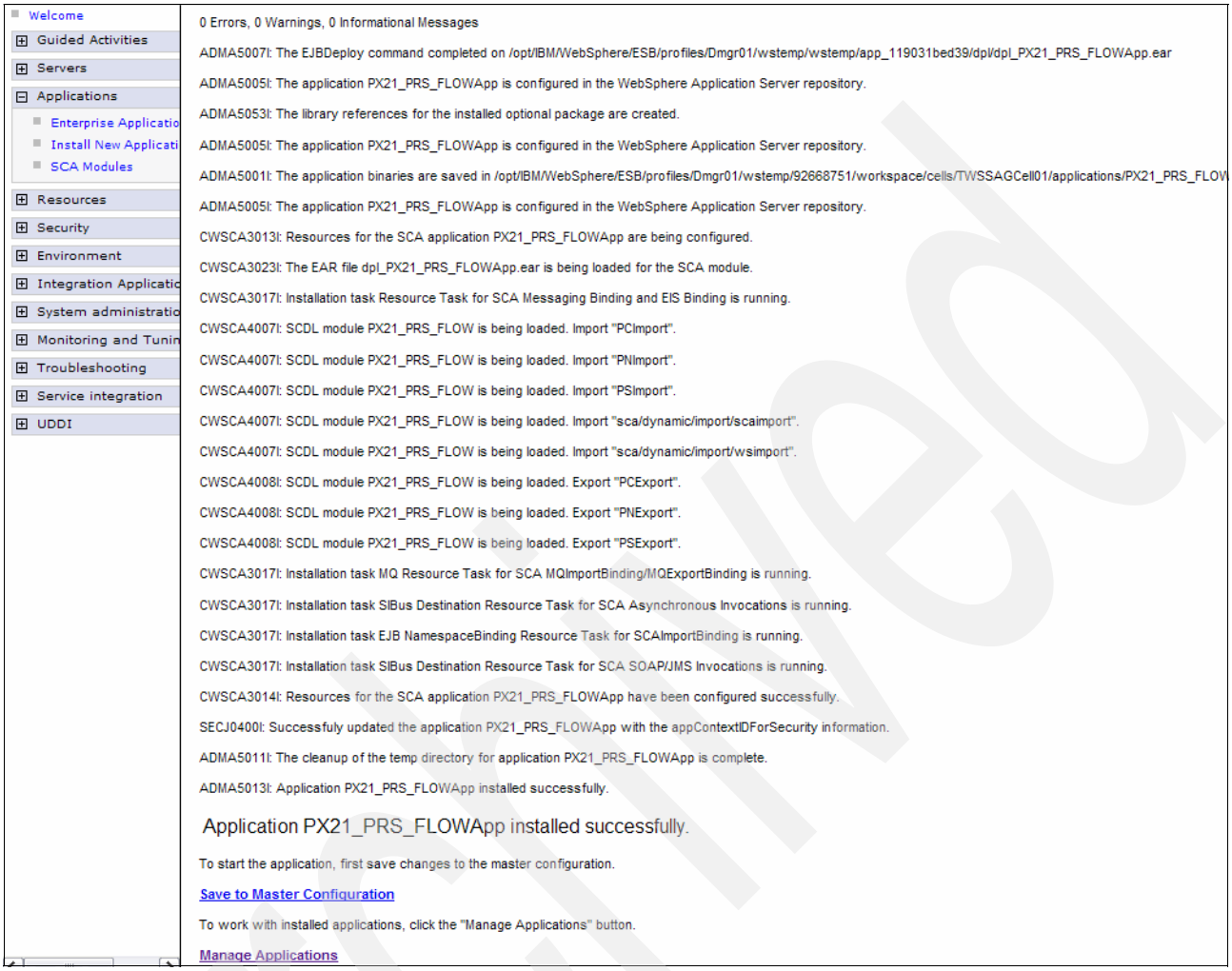


Figure 4-45 Save to Master Configuration

21. Use the default options and click the **Save** button (Figure 4-46).

Archived

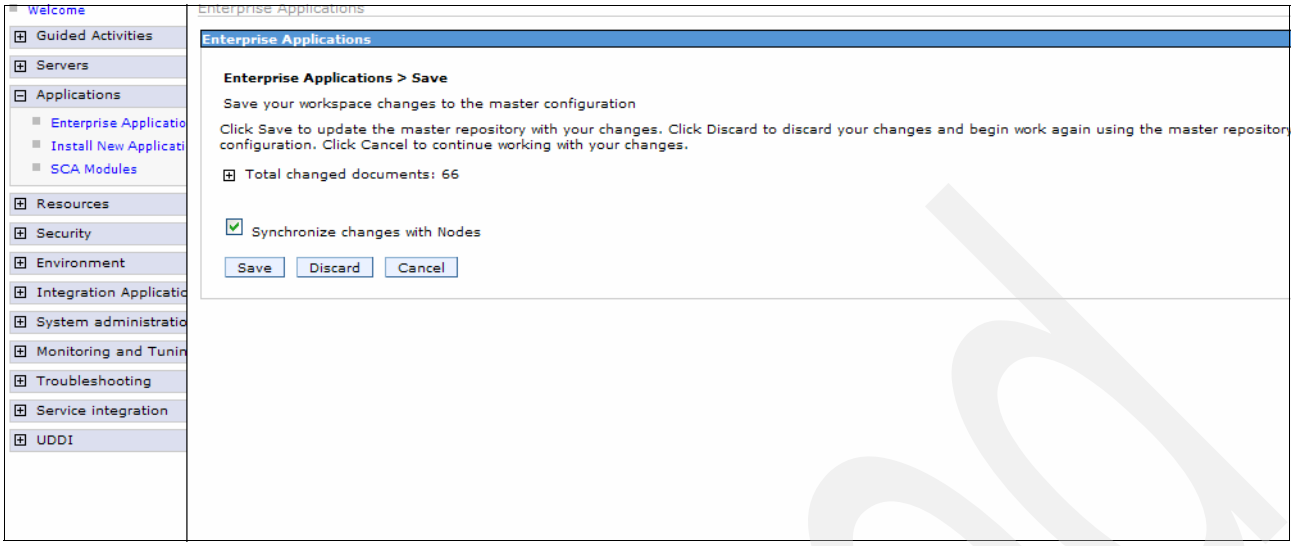


Figure 4-46 Enterprise Applications Save

22. Click **Applications** and select **Enterprise Applications**. All the applications will be listed on the right side of the window where you can see that the default flow has been installed (Figure 4-47).

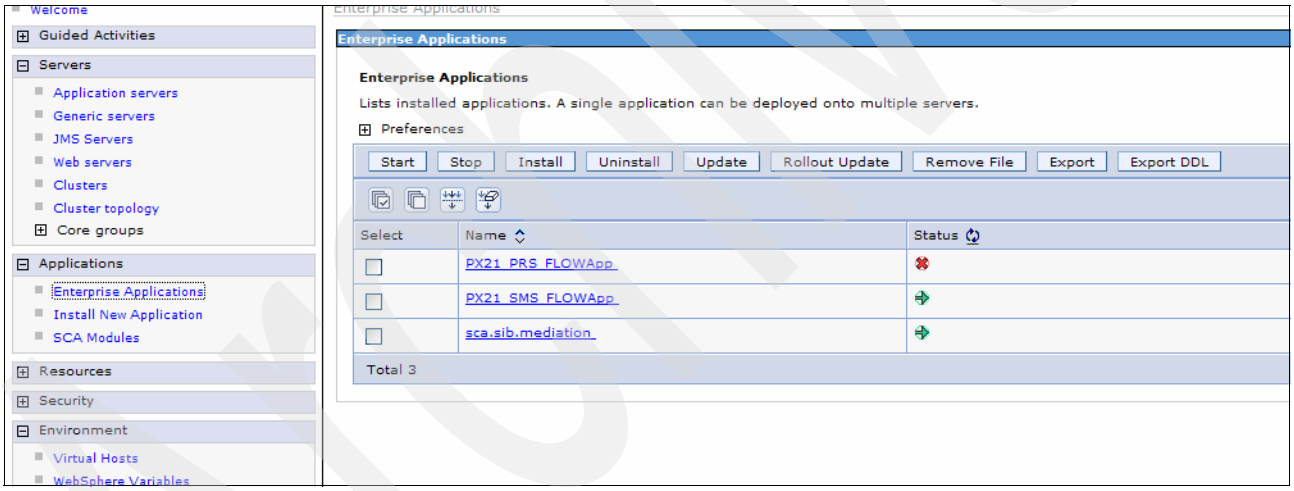


Figure 4-47 Enterprise Applications installed

Archived

23. Select the application **PX21_PRS_FLOWApp** and click the **Start** button (Figure 4-48).

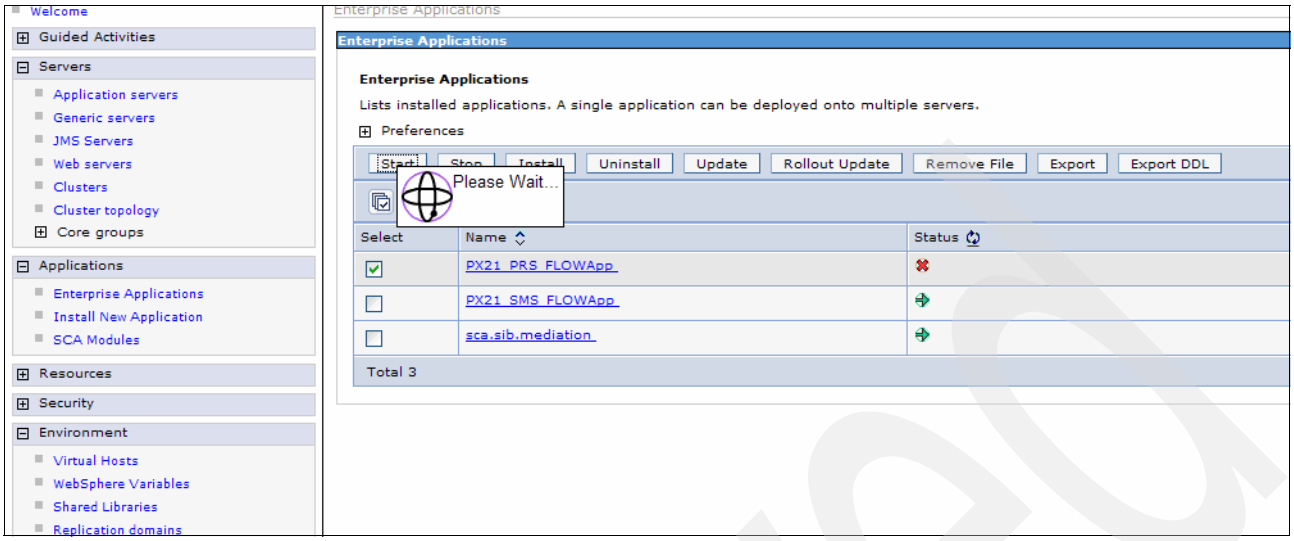


Figure 4-48 Enterprise Applications starting

24. Once the application starts successfully, the window shown in Figure 4-49 will appear.

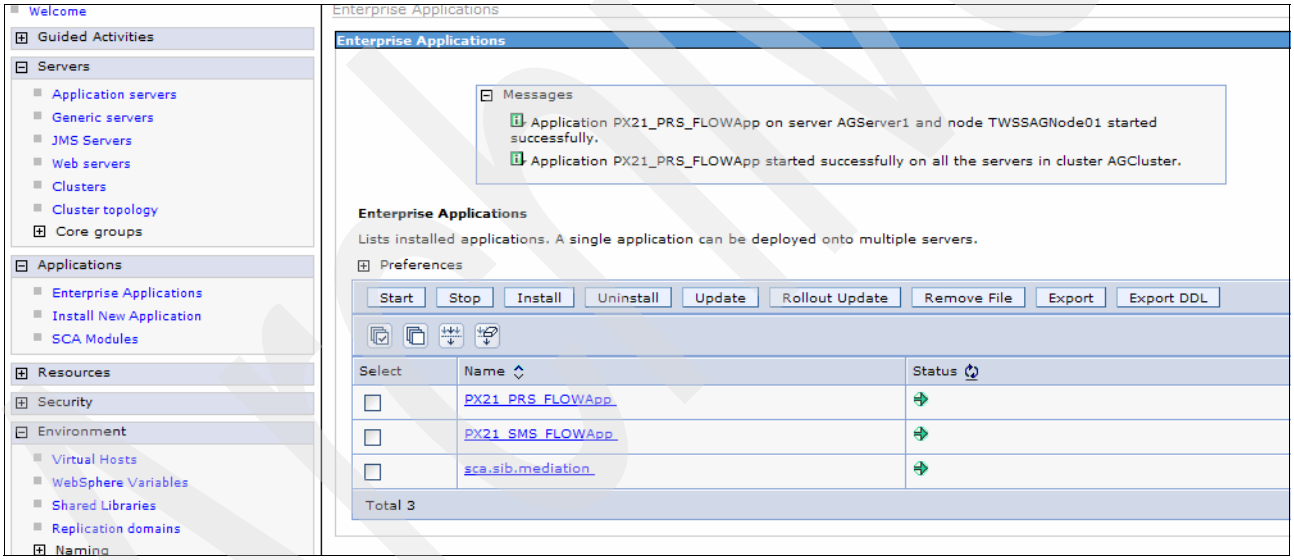


Figure 4-49 Enterprise Applications started

25. Navigate to IBM WebSphere Telecommunications Web Services Server Access Gateway machine and look for log files that the Application started successfully (Figure 4-50 on page 125).

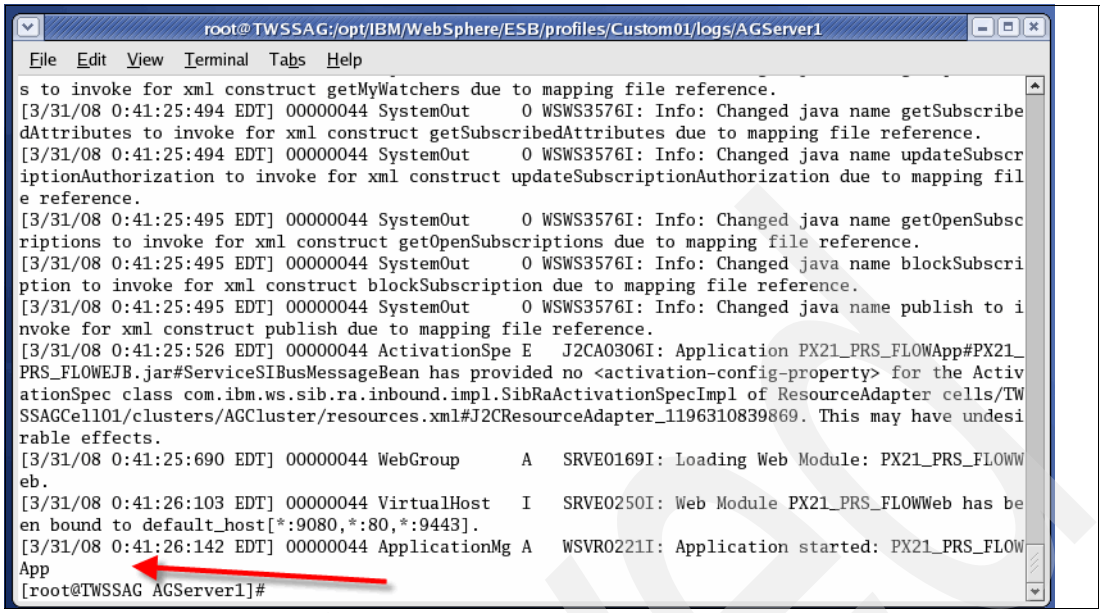


Figure 4-50 Log files started

Archive

Archived

5

Developing and customizing a custom Access Gateway flow

This chapter describes how to develop and customize an Access Gateway flow. More specifically, it covers the following customizations for the Access Gateway:

- Customizing the default mediation flow by creating and adding a new mediation primitive into the default mediation flow
- Extending the WebSphere Integration Developer Tooling Environment by converting the custom mediation primitive to a WebSphere Integration Developer Plug-in
- Creating a new mediation flow from scratch

Note: The sample code for these customizations are available for download. Refer to Appendix E, "Additional material" on page 399 for detailed instructions on how to download and work with this sample code.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

5.1 The focus of this chapter within the context of the common use case

This chapter focuses on three aspects for customizing the mediation flow in the Access Gateway.

- “Customize a default mediation flow” on page 128
- “Extending the WebSphere Integration Developer Tooling Environment” on page 153
- “Develop a custom mediation flow” on page 173

Figure 5-1 illustrates the primary focus of this chapter within the context of the common use case.

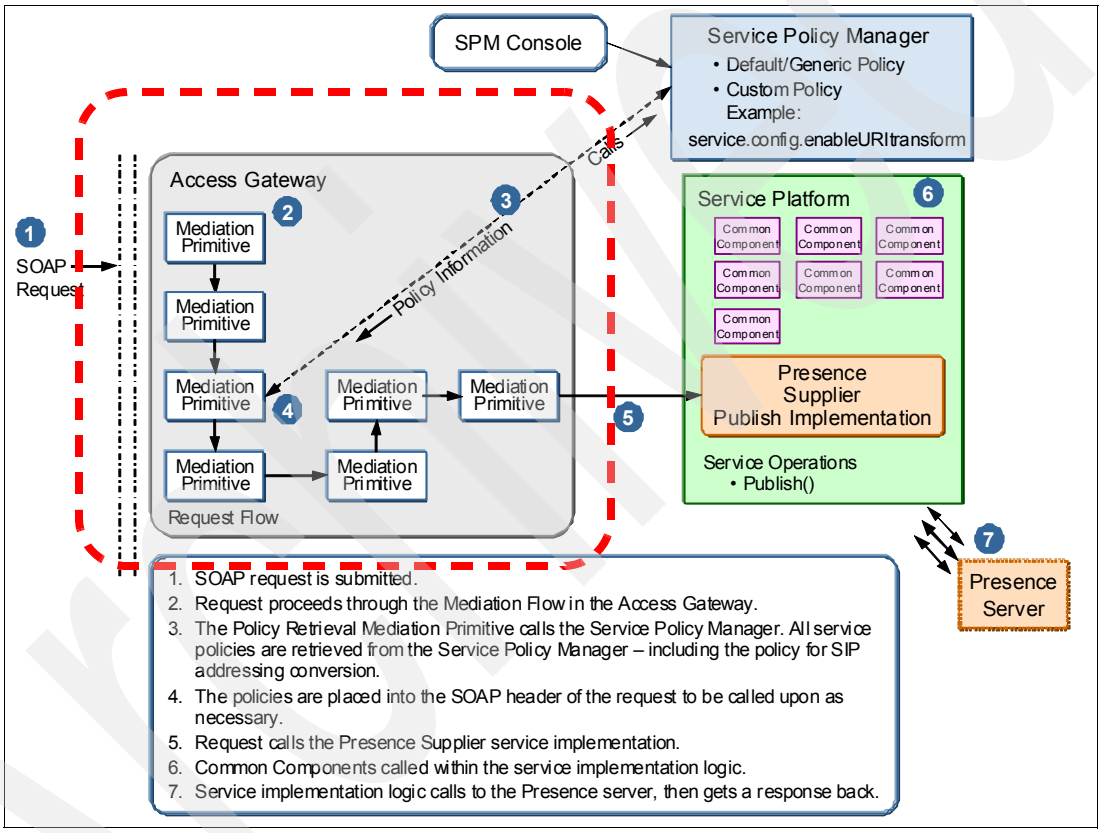


Figure 5-1 Overview of the focus of this chapter

5.2 Customize a default mediation flow

This section explains how to customize one of the default mediation flows by adding a custom mediation primitive to the flow. We use one of the flows that is provided as part of the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer plug-in, namely the Parlay X PresenceSupplier interface.

The key activities are shown in Figure 5-2 on page 129.

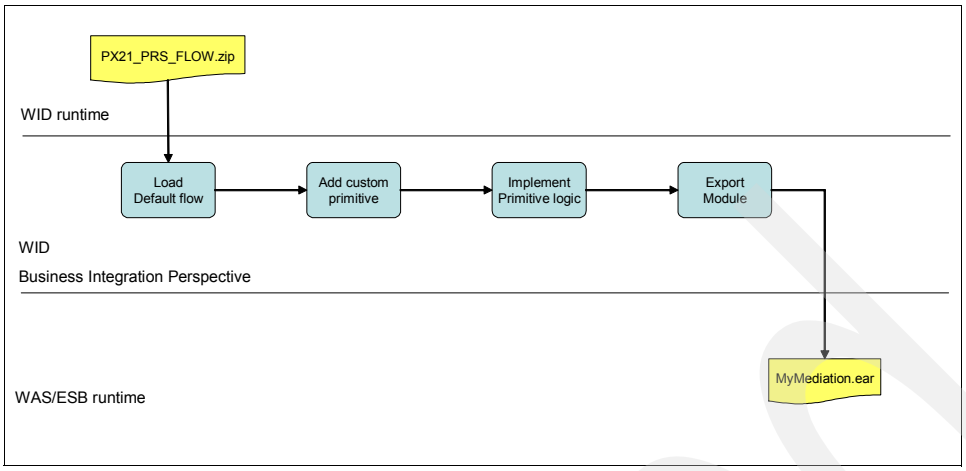


Figure 5-2 Development process to customize a default flow

First, we import the PX21_PRS_FLOW.zip project interchange file into our development environment. We then will add a custom primitive to the flow and implement a sample business logic. Lastly, we will export the flow and deploy it to the runtime environment.

We will provide only a brief description of those steps that have already been described in previous chapters.

5.2.1 Design guidelines for custom mediation primitive

WebSphere Integration Developer tooling allows the creation of custom primitives through Java snippets. A *custom mediation primitive* gets placed on the canvas and a code editor allows for the insertion of Java mediation code. This method is well documented in the WebSphere Integration Developer Information Center.

This methodology provides a quick and easy way for the creation of a custom mediation primitive. However, it is limited in terms of reuse. Each project must import the custom mediation primitive call and recreate the addition of the custom primitive. Custom mediation primitives can also be packaged in such a way as to extend the WebSphere Integration Developer tooling palette. This allows for greater reuse of custom primitive code and is described in 5.3, “Extending the WebSphere Integration Developer Tooling Environment” on page 153.

Mediation primitives implement a pipelined architecture, where information can be passed between primitives in the form of SOAP headers. The following are guidelines for the design and implementation of custom mediation primitives to best fit into this architecture. Consider the following:

- Mediation primitives should contain narrowly defined, well scoped functions. This model encourages different mediation primitives for different variations of function, rather than a complex configuration. The best primitive can be chosen (or substituted) to meet a specific need when constructing a flow.
- Try to minimize the upstream assumptions of mediation primitives. The aim of mediation primitives is to be able to control their point of execution within the flow. By minimizing assumptions about headers from elements higher in the pipeline, flexibility of use will be gained.

Properties and policies are two means to configure the behavior of your mediation primitive and thus increase flexibility. Each of them has it specific area of use that needs to taken into account when designing your primitive:

Properties Mediation primitive properties are defined in the properties group file of the mediation primitive plug-in. These properties can be defined as "promotable" or not. Properties that are not promotable can only be set using the WebSphere Integration Developer tooling environment. Promotable properties that are promoted can be modified using the WebSphere ESB Administration Console after the mediation module has been deployed. Both of these property types would be used for properties that are common for the mediation module irrespective of the requester, service, and operation that may get invoked, such as a datasource name, spm endpoint, and so on. See 5.3.2, "Generate the mediation meta-data" on page 161 for details and examples about the definition of properties.

Policies Another option is to define a service policy that is retrieved by the IBM WebSphere Telecommunications Web Services Server mediation primitive PolicyRetrieval and stored in the IBM WebSphere Telecommunications Web Services Server SOAP header. This type of property would be useful if the value of the policy needs to be set based on the requester, service, and operation getting invoked. For example, the policy message.LoggingEnabled is useful if you need trace data for a given requester having issues trying to access a particular service and operation. Then you can turn on tracing for the individual requester, service, or operation tuple while disabling tracing for all others. Handling of policies by a custom primitive is covered in "Read IBM WebSphere Telecommunications Web Services Server policies" on page 138 and "Add a new policy" on page 139.

There are four options for the implementation of a custom primitive. Table 5-1 describes these options and provides some information that can be used to decide which option to pick.

Table 5-1 Design options for custom mediations

Option	Description	Pros (+) and Cons (-)
Custom mediation - Implementation: Visual	Use a custom mediation. Implement the business logic using the built-in visual snippet editor. This option is not covered in this book.	► (-) Limitations in WebSphere Integration Developer V6.0.2 regarding imports. ► (-) Support for complex business logic. ► (-) Mediate-method scope only (no other custom methods).
Custom mediation - Implementation: Java	Use a custom mediation. Implement the business logic using the build-in Java editor. This option is described in 5.2, "Customize a default mediation flow" on page 128.	► (-) Limitations in WebSphere Integration Developer V6.0.2 regarding imports. Requires fully specified class names. ► (-) Support for complex business logic. ► (-) Mediate-method scope only (no other custom methods).

Option	Description	Pros (+) and Cons (-)
Custom mediation - Implementation: Invoke	Use a custom mediation. Implement the business logic in a separate class that is invoked by the custom mediation. This class offers an SCA export. This option is not covered in this IBM Redbooks publication.	► (+) Complex logic execution off-loaded from flow processing. ► (+) Reusability. ► (+) Easy to migrate to a WebSphere Integration Developer plug-in.
Custom primitive and WebSphere Integration Developer plug-in	Define a plug-in. Implement the business logic in a separate class. This option is described in 5.3, “Extending the WebSphere Integration Developer Tooling Environment” on page 153.	► (+) Reusability. ► (+) Flexibility. ► (-) Slightly more effort to design and implement.

5.2.2 Key custom primitive functions

In this section, we will provide code samples for some of the key functions you will most likely need to implement in a custom mediation primitive.

A custom primitive provides information to downstream mediation primitives or to the service implementation by enriching or modifying the headers of the Service Message Object, as shown in Figure 5-3.

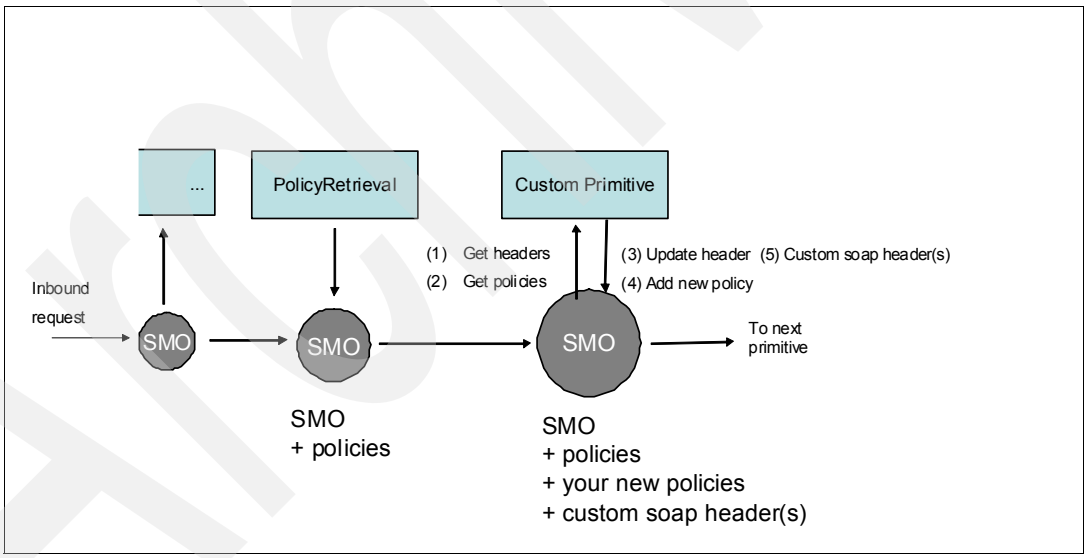


Figure 5-3 Custom primitive key functions

A custom service primitive receives a Service Message Object (SMO) that contains the body for the original request and the headers that have been added by all primitives and handlers that intercepted the flow before it reaches the custom primitive.

It can interact with the SMO in five ways:

1. Read headers to get information required to execute the primitive's business logic.
2. Read the policies to enforce customizable behavior of the primitive's business logic.
3. Update header elements.
4. Create new IBM WebSphere Telecommunications Web Services Server header policy elements to pass new data to the downstream primitives and service implementations.
5. Create custom SOAP header(s) to pass new data to the downstream custom primitives and service implementations.

> **Note:** As the IBM WebSphere Telecommunications Web Services Server header is defined by a WSDL, it is not possible to add new elements to it during run time. Simple name/value pair data can be added to the IBM WebSphere Telecommunications Web Services Server header by adding it as policy data. Additional SOAP header(s) can also be used to pass additional data to downstream custom mediation primitives or Service Implementations by importing custom WSDL into the mediation module and having a custom mediation primitive create and add the custom SOAP header(s) to the message.

In the following paragraphs, we will introduce code snippets that demonstrate how to read headers as well as how to update header elements.

All non-static properties that are required by a custom primitive should be either defined as policies or promotable properties to enable easy runtime modifications (see 5.2.1, "Design guidelines for custom mediation primitive" on page 129). In this section, we will look at the use of policies by a custom primitive. The PolicyRetrieval primitive, which is one of the mandatory primitives in a flow, adds all policies that are defined for the requester, service, and operation through to the SMO header. Code snippets in the following paragraphs provide examples on how to retrieve the policies from the SMO header as well as to add new policies.

In a final paragraph, we introduce code snippets to cover fault handling inside a custom primitive.

Service Message Object (SMO) structure

Service message objects (SMOs) are enhanced Service Data Objects (SDOs). SMO provides an abstraction layer for processing and manipulating messages exchanged between services. The SMO contains a representation of the following groups of data:

- Context information (data other than the message payload).
- Header information associated with the message. For example, Java Message Service (JMS) headers if a message has been conveyed using the JMS API.
- Body: The business payload of the message. The payload is the application data exchanged between service endpoints.

Figure 5-4 on page 133 shows the structure of the SMO.

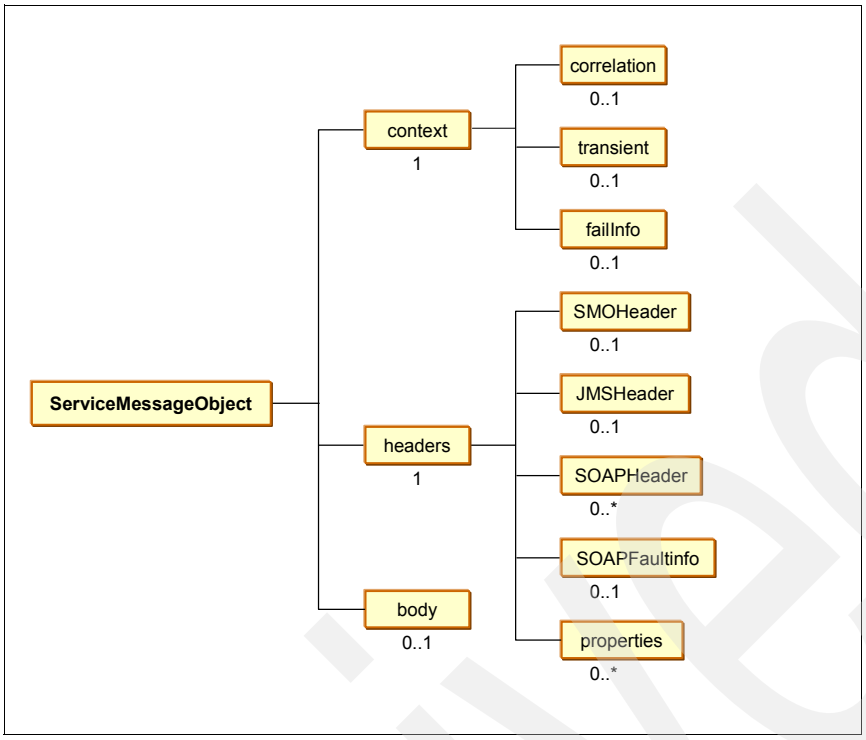


Figure 5-4 The Service Message Object structure

Each part of the SMO serves a different purpose:

- Context: Allows data that is not part of the message payload to be passed between mediation primitives.
  - Correlation: The correlation context can link a specific request message with its response.
  - Transient: Transient context is restricted to a single message flow, and cannot link requests and responses.
  - Failinfo: Represents exception information for use when a fail terminal is wired.
- Headers: Header information associated with the message.
  - SMOHeader: Models generic header fields (for example, version)
  - SOAPHeader: Models the SOAP headers.
  - SOAPFaultInfo: Models the SOAP fault information.
  - Properties: Models a list of properties whose names are not fixed from one message instance to the next.
- Body: Contains the message payload (the application data exchanged between service endpoints) as, for example, defined in the Parlay X WDLS.

Note: For a detailed description of the SMO structure, refer to the WebSphere Integration Developer V6.0.2 InfoCenter.

Archived

Information that is specific to IBM WebSphere Telecommunications Web Services Server request handling is added to a SOAP header called “twssHeaders”. Figure 5-5 shows the basic structure of the IBM WebSphere Telecommunications Web Services Server header.

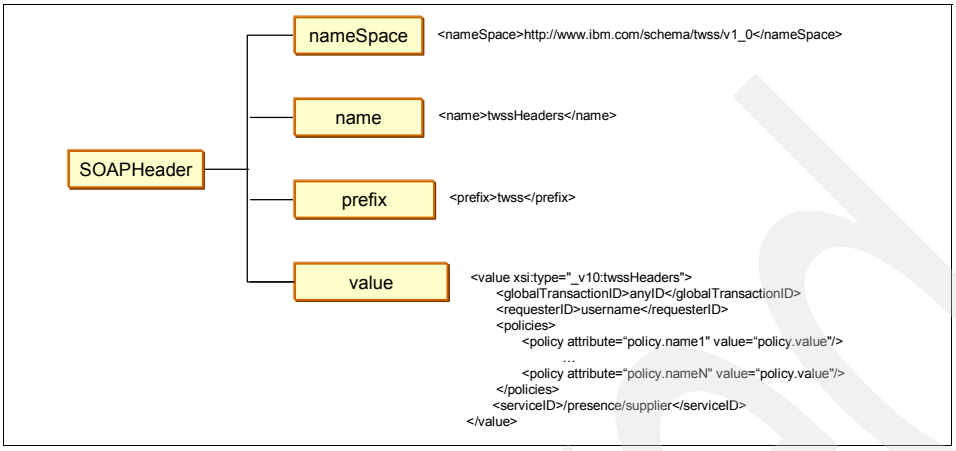


Figure 5-5 The IBM WebSphere Telecommunications Web Services Server Header structure

The value element of the IBM WebSphere Telecommunications Web Services Server header contains some key information required for processing the request:

- globalTransactionID: This is the unique identifier of the transaction.
- requesterID: This element contains the ID that has been used to authenticate the requester (for example, a user name). The requesterID element is always present. If security is disabled on the access gateway, then the special “unauthenticated” value is used.
- policies: This is a list of policies. The policy is one entry per policy that is retrieved from the policy manager. Each entry contains a policy name and policy value.
- serviceID: This is the URI of the service interface.

Example 5-1 is a sample of a real SMO as documented in the trace of an incoming request.

Example 5-1 Example of a Service Message Object

```
<ServiceMessageObject:smo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ServiceMessageObject="http://www.ibm.com/websphere/sibx/smo/v6.0.1"
xmlns:_local="http://www.csapi.org/schema/parlayx/presence/supplier/v2_3/local"
xmlns:_pX2="http://PX21_PRS_FLOW" xmlns:_v10="http://www.ibm.com/schema/twss/v1_0"
xmlns:interface="http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface">
  <context>
    <transient xsi:type="_pX2:ExceptionType"/>
  </context>
  <headers>
    <SMOHeader>
      <MessageUUID>3251235D-0119-4000-E000-6AE2C0A80166</MessageUUID>
      <Version>
        <Version>6</Version>
        <Release>0</Release>
        <Modification>2</Modification>
      </Version>
      <MessageType>Request</MessageType>
    </SMOHeader>
    <SOAPHeader>
```

```
      <nameSpace>http://www.ibm.com/schema/twss/v1_0</nameSpace>
      <name>twssHeaders</name>
      <prefix>twss</prefix>
      <value xsi:type="_v10:twssHeaders">
        <globalTransactionID>3251235D-0119-4000-E000-6AE2C0A80166</globalTransactionID>
        <requesterID>sip:user2@example.com</requesterID>
        <policies>
          <policy attribute="service.config.enableURItrace" value="true"/>
          <policy attribute="message.groups.MaxGroupSize" value="100"/>
          <policy attribute="service.config.presence.supplier.PresenceServerURI" value=""/>
          <policy attribute="requester.anonymousAccessAllowed" value="true"/>
          <policy attribute="message.sla.LocalOperationRate" value="0"/>
          <policy attribute="service.config.enableURItransformation" value="true"/>
          <policy attribute="service.standard.NestedGroupSupport" value="false"/>
          <policy attribute="message.sla.ClusterWeight" value="0"/>
          <policy attribute="message.sla.LocalEnabled" value="false"/>
          <policy attribute="service.config.requestIDorig" value="user2"/>
        </policies>
<serviceID>http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface</serviceID>
      </value>
    </SOAPHeader>
  </headers>
  <body xsi:type="interface:PresenceSupplier_publishRequest">
    <publish>
      <_local:presence>
        <lastChange>2002-11-11T11:11:11.0Z</lastChange>
        <note>hi this is my new status</note>
        <typeAndValue>
          <UnionElement>Activity</UnionElement>
          <Activity>Travel</Activity>
        </typeAndValue>
      </_local:presence>
    </publish>
  </body>
</ServiceMessageObject:smo>
```

Required imports

The functions described in this section will require you to import certain classes from the WebSphere Enterprise Service Bus run time, as shown in Example 5-2.

Example 5-2 Required imports

```
/* base Java classes */
import java.util.List;
import java.util.logging.Logger;
import java.util.logging.Level;

/* IBM WebSphere Enterprise Service Bus classes -  emf.jar */
import commonj.sdo.DataObject;

/* IBM WebSphere Enterprise Service Bus classes - wbiBOS.jar */
import com.ibm.websphere.bo.BOFactory;

/* IBM WebSphere Enterprise Service Bus classes - sca.jar */
import com.ibm.websphere.sca.ServiceManager;
```

Archived

```
/* IBM WebSphere Enterprise Service Bus classes - smobo.jar */
import com.ibm.websphere.sibx.smobo.ServiceMessageObject;
import com.ibm.websphere.sibx.smobo.ServiceMessageObjectFactory;
iimport com.ibm.websphere.sibx.smobo.ContextType;
import com.ibm.websphere.sibx.smobo.HeadersType;
import com.ibm.websphere.sibx.smobo.FailInfoType;
import com.ibm.websphere.sibx.smobo.SOAPFaultInfoType;
import com.ibm.websphere.sibx.smobo.SOAPHeaderType;
mport com.ibm.ws.sibx.smobo.util.XMLSerialisationHelper;

/* IBM WebSphere Enterprise Service Bus classes - sibx.mediation.engine.jar */
import com.ibm.wsspi.sibx.mediation.InputTerminal;
import com.ibm.wsspi.sibx.mediation.OutputTerminal;
import com.ibm.wsspi.sibx.mediation.MediationBusinessException;
import com.ibm.wsspi.sibx.mediation.MediationConfigurationException;
import com.ibm.wsspi.sibx.mediation.esb.ESBMediationPrimitive;
```

Read SMO headers

The code snippet in Example 5-3 shows how to extract the IBM WebSphere Telecommunications Web Services Server headers from the SOAP header.

Example 5-3 Extract IBM WebSphere Telecommunications Web Services Server SOAP headers

```
/* Extract the HeadersType data object which will contain all headers*/
HeadersType headertype = ((ServiceMessageObject) message).getHeaders();

/* Extract All SOAP headers attached to SMO */
List soaphdrs = headertype.getSOAPHeader();

/* From list of SOAP headers , get the IBM WebSphere Telecommunications Web Services Server Soap
header data object, named "twssHeaders" */
DataObject twssDO = getSoapHeader(soaphdrs, "twssHeaders");
```

First, you extract all headers from the Service Message Object into a HeadersType object.

The HeadersType class implements methods to extract each of the various headers (see “Service Message Object (SMO) structure” on page 132) into a list.

The methods to extract headers are:

- SMOHeaderType getSMOHeader()
- List getSOAPHeader()
- SOAPFaultInfoType getSOAPFaultInfo()
- List getProperties()
- JMSHeaderType getJMSHeader() (if JMS has been used as transport mechanism)
- MQHeaderType getMQHeader() (if MQ has been used as transport mechanism)

In our example, as we are interested in the contents of the SOAP header, we will use the getSOAPHeader method to extract the list.

Archived

We will retrieve the IBM WebSphere Telecommunications Web Services Server headers from this list and create a DataObject holding them. In Example 5-3 on page 136, this is done by the small helper method getSoapHeader. This method takes the list of SOAP headers and a string that identifies the specific header we are interested in as input.

An example implementation of this function is shown in Example 5-4.

Example 5-4 Extract the IBM WebSphere Telecommunications Web Services Server headers from the SOAP header

```
private DataObject getSoapHeader( List soapHeaderList, String headerName) {
   SOAPHeaderType soapHeader = null;

   if ( soapHeaderList != null ) {

      for (int i=0; i<soapHeaderList.size(); i++) {
         soapHeader = (SOAPHeaderType)soapHeaderList.get(i);
         if (soapHeader.getName().equals(headerName)) {
            return (DataObject)soapHeader.getValue();
         }
      }
   }
   return null;
}
```

In this sample, we loop through each header in the list of SOAP headers until we find the element named “twssHeaders”. This is returned as DataObject.

Read and update a IBM WebSphere Telecommunications Web Services Server header element

The examples provided in this paragraph assume that we have extracted the IBM WebSphere Telecommunications Web Services Server headers from a message to an object named twssDO (see Example 5-3 on page 136).

A DataObject offers these key methods to access and manipulate the elements:

- void setString(String property, String value): Set a property with a string value.
- String getString(String property): Get the current value of a property as a string.
- boolean isSet(String property): Check if the property is set.
- void unset(String property): Empty the property.

Note: The get... and set... methods are available for various types.

In Example 5-5, we use the isSet and getString methods to retrieve the requesterID element from the IBM WebSphere Telecommunications Web Services Server header.

Example 5-5 Get a IBM WebSphere Telecommunications Web Services Server header element

```
if (twssDO.isSet("requesterID")){
   String requesterID = twssDO.getString("requesterID");
}
```

Archived

Note: The property can be specified as a simple XPath expression. The property “policy/policies” will get the list of policies from the header (see also Example 5-7 on page 138).

To update a IBM WebSphere Telecommunications Web Services Server header element, we use the set-methods shown in Example 5-6.

Example 5-6 Update a IBM WebSphere Telecommunications Web Services Server header element

```
if (twssDO.isSet("requesterID")){

   // write new value to header
   twssDO.setString("requesterID",newRequesterID);

}
```

Read IBM WebSphere Telecommunications Web Services Server policies

The examples provided in this paragraph assume that we have extracted the IBM WebSphere Telecommunications Web Services Server headers from message to an object named twssDO (see Example 5-3 on page 136).

Policies are a means to configure the behavior of a custom primitive. As such, it is important to understand how to extract a policy from the IBM WebSphere Telecommunications Web Services Server header.

The custom mediation needs to know which policy it is looking for. There is no generic mechanism to filter the policies unless certain naming conventions have been defined for all policies that are in scope of the mediation.

In Example 5-7, we assume that the custom mediation knows which policy to retrieve.

Example 5-7 Read IBM WebSphere Telecommunications Web Services Server policies

```
// Extract the policies from the IBM WebSphere Telecommunications Web Services
Server Header

List policiesList = null;
boolean isEnabled = false;

// if the IBM WebSphere Telecommunications Web Services Server policies header is
set, get the list of policies
if ( twssDO.isSet(“policies“) ) {
   policiesList = twssDO.getList(“policies/policy“);
}

// get the setting for the Address Transformation policy
isEnabled = getBooleanPropertyValue(policiesList,”service.config.myPolicy”,true);
```

In the example, we retrieve the list of all policies from the header using the DataObjects getList-method. Then we retrieve the policy we are looking for from this list. This is encapsulated in another method, which is shown in Example 5-8 on page 139.

Example 5-8 Scan policylist

```
public static boolean getBooleanPropertyValue(List policiesList, String
policyName, boolean defaultVal ) {

   DataObject policy = null;

   for (int x=0; x < policiesList.size(); x++ ) {
      policy = (DataObject)policiesList.get(x);
      if ( policy.get("attribute").equals(policyName)) {
         return policy.getBoolean("value");
      }
   }
   return defaultVal;
}
```

The method expects a list of policies as input as well as the name of the policy to find and a default value to use in case the policy is not found.

Add a new policy

Adding a new policy is the recommended way for a custom primitive to pass on additional information to downstream components.

The code snippet in Example 5-9 shows how to add a new policy to the IBM WebSphere Telecommunications Web Services Server header.

Example 5-9 Add a new policy to the IBM WebSphere Telecommunications Web Services Server header

```
// assumes the policies have already been extracted into policiesList

// get a Business Object
ServiceManager serviceManager = new ServiceManager();
BOFactory bof = (BOFactory) serviceManager.locateService("com/ibm/websphere/bo/BOFactory");

// create the new policy element
DataObject newPolicy = bof.createByElement("http://www.ibm.com/schema/twss/v1_0", "policy");
newPolicy.set("attribute", "service.config.requestIDorig");
newPolicy.set("value", requesterID);

// add new policy to the list
policiesList.add(newPolicy);
```

The new policy needs to be instantiated as new business object to be added to the list of existing policies (the one that had been retrieved from the header in the previous paragraph).

First, we use the ServiceManager to instantiate a Business Object Factory.

Among others, the Business Object Factory provides a method called createByElement to create a new business object. This method takes two arguments:

- The namespace of the target business object, which is "`http://www.ibm.com/schema/twss/v1_0`" for the IBM WebSphere Telecommunications Web Services Server Headers.
- The name of the element ("`policy`").

Each policy is described by a name and a value. We use the DataObject “set” method to add both the attribute (the policy name) and the value.

We define the name of the policy to be **“**`service.config.requestIDorig`**”**. The value is set to the requesterID, which initially had been retrieved from the header (see Example 5-5 on page 137).

We add the policy DataObject to the list of policies. Thus, the new policy is added to the IBM WebSphere Telecommunications Web Services Server headers when the outgoing message is created for this custom primitive.

Handle terminals

A custom primitive needs to instantiate OutputTerminal objects in order to fire messages to these terminals. Example 5-10 shows the required code to create a standard out as well as an optional fault terminal.

Example 5-10 Get the output terminals

```
// gets the out and fault terminals from the mediation services

OutputTerminal outTerminal = getMediationServices().getOutputTerminal(“out“);

OutputTerminal faultTerminal = getMediationServices().getOutputTerminal(“fault”);
```

Note: The names of the output terminals must match the ones defined in the Terminal Properties of your custom primitive.

The OutputTerminal class implements a fire method that takes a DataObject as parameter. Example 5-11 shows a code snippet that fires a message to the outTerminal.

Example 5-11 Fire a message to an output terminal

```
if (outTerminal != null) {
   outTerminal.fire(message);
}
```

Fault handling

IBM WebSphere Telecommunications Web Services Server mediation primitives that can issue faults have an additional output terminal added named *fault*. As shown in “Service Message Object (SMO) structure” on page 132, the SMO comprises two elements that carry fault information:

- The context contains a faultinfo element.
- The SOAPFaultinfo header.

To send a fault, create a FailInfoType object and a SOAPFaultInfo object, add them to the SMO, and fire a message on the fault terminal.

To handle a fault for Parlay X service requests, the fault terminal should be wired to the IBM WebSphere Telecommunications Web Services Server fault flow. A simplified version is shown in Figure 5-6 on page 141.

Archived

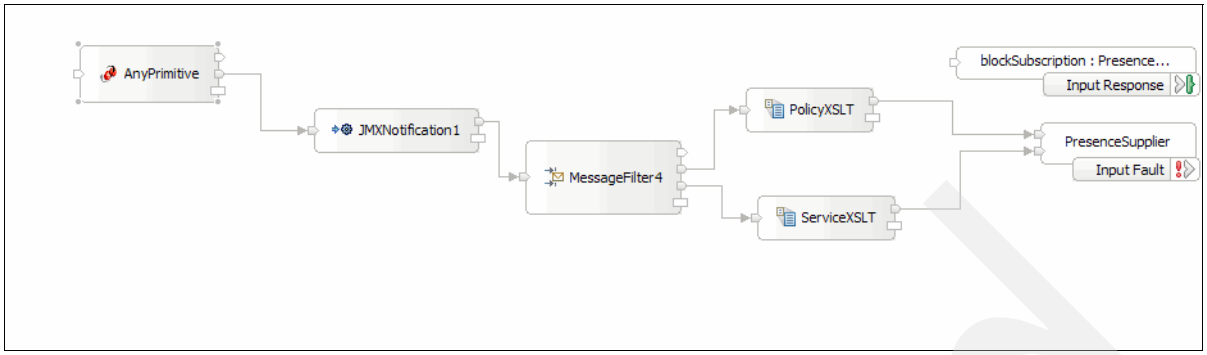


Figure 5-6 Default fault flow

Starting with the primitive that has fired a message to the fault terminal, this message triggers a JMX notification. Driven by policies, the fault may be logged and a CEI event may be emitted. Both have been left out in the above flow to focus on the essentials.

The IBM WebSphere Telecommunications Web Services Server Fault flow then contains a message filter on the SMO context/transient/exceptionType (a BusinessObject added to all the IBM WebSphere Telecommunications Web Services Server mediation module projects). The exceptionType should be set to `service` or `policy` depending on whether a ServiceException or PolicyException fault will be returned.

The PolicyXSLT adds a body that contains a PolicyException to the SOAP message, while the ServiceXSLT adds a body with a ServiceException.

Example 5-12 shows a sample code snippet that we used in our custom primitive to create the fault headers and fire a message to the fault output terminal.

Example 5-12 Fault handling

```
// set the exceptionType in the messae context. Allowed values are “service” or
“policy”

message.set("context/transient/exceptionType", “service”);
// set faultCode and faultMsg
String faultCode = “CustomFault0001”;
String faultMsg = “Detailed description about the fault that occured”;

//get the context from the message
ContextType contextType = ((ServiceMessageObject)message).getContext();

// add Fail Info and Soap fault to SMO
FailInfoType failInfo = getFailInfoTypeObject(faultMsg);
contextType.setFailInfo(failInfo);

SOAPFaultInfoType soapfaultInfo = getSoapFaultInfoTypeObject(faultMsg, faultCode);
headerType.setSOAPFaultInfo(soapfaultInfo);

//fire message to fault terminal
if (faultTerminal != null) {
   faultTerminal.fire(message);
}
```

In the example, we set the ExceptionType of the transient context to “service” to indicate that the failure has been caused by the service logic. Remember that the other option would be “policy”.

We then define a custom fault code and the text that describes the fault. Custom fault codes can be used; just assign the detailed text, and do not use getExceptonText.

We get the context from the message by calling the getContext method of the message. The context is required in order to add the faultinfo in a subsequent step using the setFailInfo method of the context object.

To create the FailInfoType object, which needs to be added to the context, we invoke the method shown in Example 5-13.

Example 5-13 .Create a FaultInfoType object

```
      private FailInfoType getFailInfoTypeObject( String failMsg) {

         FailInfoType failinfo =
            ServiceMessageObjectFactory.eINSTANCE.createFailInfoType();

         failinfo.setOrigin(this.getClass().getName());
         failinfo.setFailureString(failMsg);

         return failinfo;
      }
```

Similarly, we have implemented a method to create a SoapFaultInfo object. This code is shown in Example 5-14.

Example 5-14 Create a SoapFaulInfoType object

```
      private SOAPFaultInfoType getSoapFaultInfoTypeObject( String faultMsg,
String faultCode ) {

         SOAPFaultInfoType soapfaultinfotype =
            ServiceMessageObjectFactory.eINSTANCE.createSOAPFaultInfoType();

         soapfaultinfotype.setFaultcode(faultCode);
         soapfaultinfotype.setFaultstring(faultMsg);

         return soapfaultinfotype;
      }
```

We use the setSOAPFaultInfo provided by the HeadersType class to add this header to the message.

5.2.3 Use case description for the customization

In order to demonstrate as many of the SMO interactions as possible, we have created a fictitious use case. This use case is based on the default mediation flow for the Parlay X PresenceSupplier publish operation.

The requirements for the mediation flow are as follows:

- The mediation flow transforms user names into SIP URIs. The user name is provided by the incoming request.
- The SIP URI follows this pattern: sip:username@exmaple.com.
- It is possible to enable/disable the transformation.
- The original user name is kept as a reference.

To fulfill these requirements, we add an AddressTransformation custom primitive to the default flow, as shown in Figure 5-7.

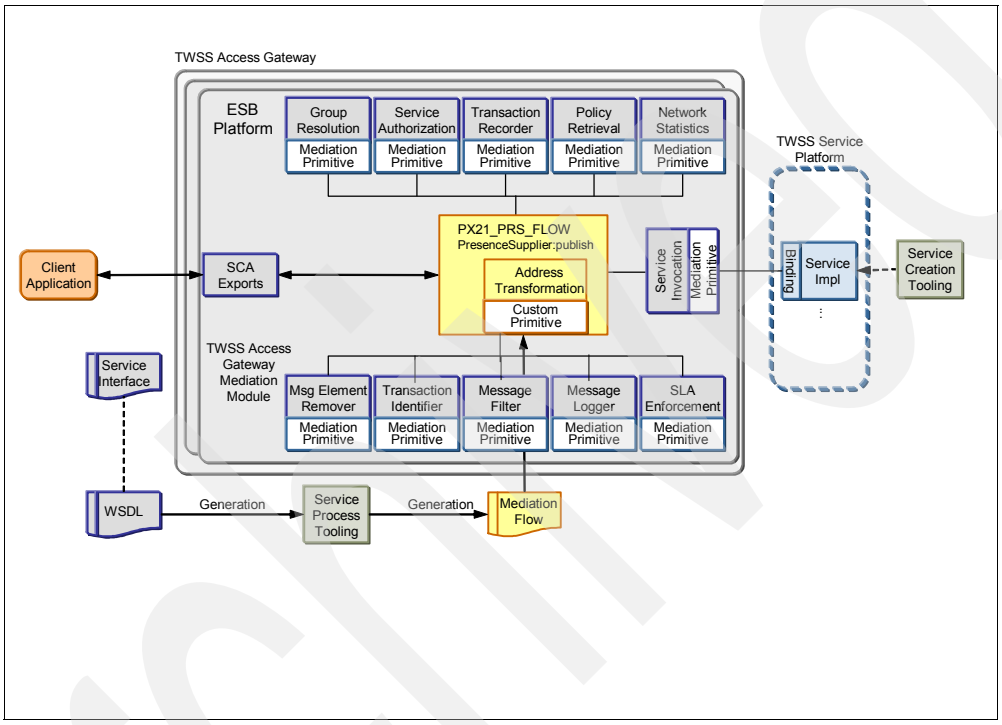


Figure 5-7 Customized Presence Supplier Mediation Module

Archive

The default flow, as contained in the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer plug-in (see 4.3, “Default Access Gateway flow” on page 72), needs to be modified, as shown in Figure 5-8.

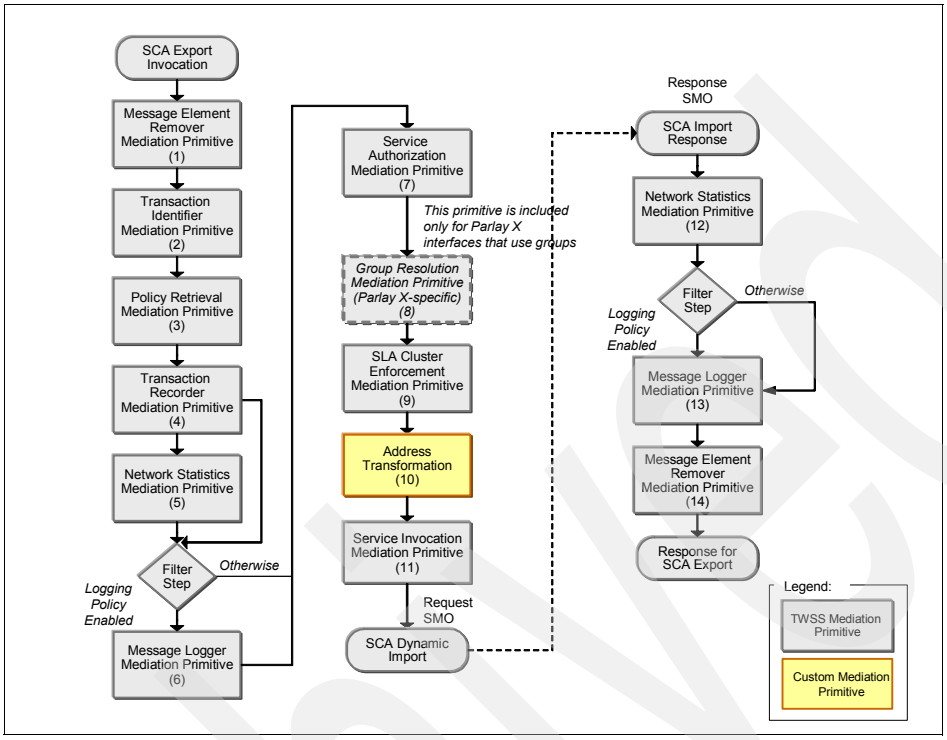


Figure 5-8 Customized request flow

The Address Transformation custom primitive implements this business logic:

1. It retrieve the enableURItransformation policy from the SMO header.
2. If enableURItransformation is true, then it:
   a. Gets the requesterID header from the SMO header.
   b. Creates a new requesterID with the format sip:requesterID@example.com.
   c. Updates the requesterID header with the new value.
   d. Creates a new policy named service.config.requesterIDorig that contains the original value of requesterID.

Note: As discussed in 5.2.1, “Design guidelines for custom mediation primitive” on page 129, there are various options to implement these requirements. We have decided to implement a custom primitive that handles all requirements.

One alternative option would be to use a similar pattern as applied to the MessageLogger in the default flows, as follows:

1. Insert a MessageFilter into the flow to retrieve and enforce the enableURItransformation policy.
2. Insert a AddressTransformation custom primitive that just provides the transformation.

The custom primitive shall be inserted into the flow at the very end, just before the endpoint is selected and the service implementation is called.

Note: The use case requires global security to be switched on for the application server. This results in the user name, which is used to authenticate the incoming request to be added to the SOAP header as *requesterID*.

5.2.4 Load the default flow

As our goal is to modify the mediation flow of the PresenceSupplier publish request, we first need to load the default flow into WebSphere Integration Developer (for a detailed description, see 4.6, “Using the default mediation flow in WebSphere Integration Developer” on page 99):

1. Start WebSphere Integration Developer and select **File** → **Import**. In the Import window, select **Project Interchange**. Click **Next**.
2. Browse to the WID_INSTALL/TWSS directory and select the archive **PX21_PRS_FLOW.zip**. Click **Finish** to start the import.
3. You should now switch to the Business Integration perspective. The import project structure is shown in Figure 5-9.

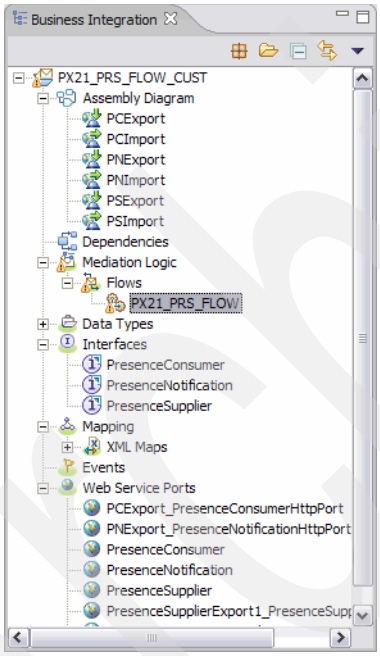


Figure 5-9 The imported default flow

5.2.5 Adding the new mediation primitive to the mediation flow

In this section, we will add a custom mediation to the default request flow:

1. In the project outline (see Figure 5-9), expand **Mediation Logic**, then **Flows**, and double click the flow **PX21_PRS_FLOW**. This opens the mediation flow editor (see Figure 5-10 on page 146).
2. In the upper pane, scroll down and locate the PresenceSupplier interface. Select the publish operation. The request flow is now displayed in the lower pane.

Archived

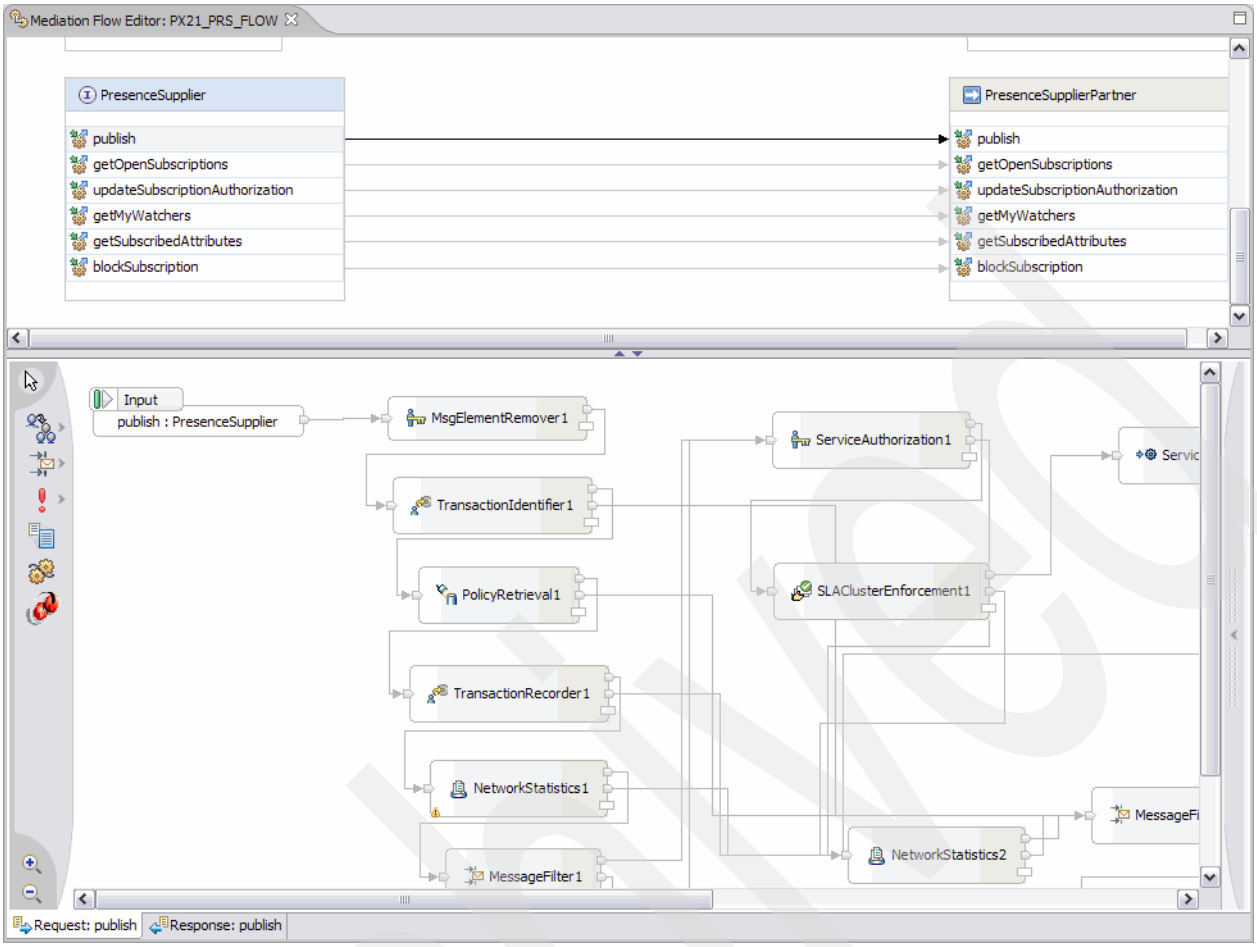


Figure 5-10 Default request flow for PresenceSupplier publish operation

In our use case, we must insert the custom mediation at the very end of the flow, that is, between the SLAClusterEnforcement and the ServiceInvocation primitive.

3. Click the **Custom Mediation** primitive in the flow editor's tool palette and drag it onto the canvas. This will create a CustomMediation1 primitive. Change its name to *AddressTransformation*.
4. Disconnect the existing connection at the place where we want to insert our custom mediation. Select the wire between the output terminal of the SLAClusterEnforcement and the input terminal of the ServiceInvocation primitive and delete it.
5. Now insert the AddressTransformation into the flow. Click the Output terminal of the SLAClusterEnforcement primitive and drag the wire to the Input terminal on the AddressTransformation primitive.
6. Click the Output terminal of the AddressTransformation primitive and drag the wire to the Input terminal on the ServiceInvocation primitive.
7. Click the Fail terminal of the AddressTransformation primitive and drag the wire to the Input terminal on the NetworkStatistics primitive.

Your flow now should look like Figure 5-11 on page 147.

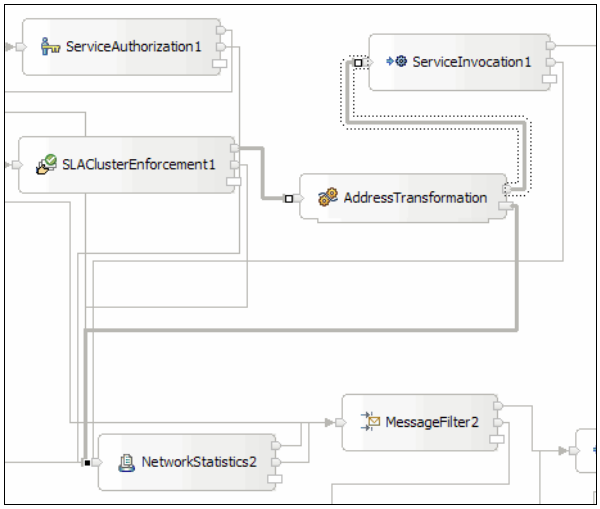


Figure 5-11 AddressTransformation inserted into the flow

5.2.6 Develop the mediation primitive logic

Using the snippets described in 5.2.2, “Key custom primitive functions” on page 131, we are now able to develop the business logic of our custom mediation:

1. Select the **AddressTransformation** primitive.
2. In the Properties tab, click **Details**.
3. For Implementation, select **Java**.
4. Copy the code listed in Example 5-15 to the scroll box in the Details tab.

Example 5-15 Custom logic

```
final String policyName = "service.config.enableURItransformation";
final boolean polURIdefault = false;
boolean polURI = polURIdefault;

System.out.println("Custom Primitive AddressTransformation: method entered");

// get the Service Messge Object (SMO) from the input terminal
com.ibm.websphere.sibx.smobo.ServiceMessageObject smo =
(com.ibm.websphere.sibx.smobo.ServiceMessageObject)input1;

// now get the HeadersType opbject from the SMO. This contains all headers (SMO, SOAP, ...)
com.ibm.websphere.sibx.smobo.HeadersType headers = smo.getHeaders();

// get all SOAP headers from the SMO
java.util.List soapHdrs = headers.getSOAPHeader();
commonj.sdo.DataObject twssHdrs = null;

// extract the IBM WebSphere Telecommunications Web Services Server headers from the list of
SOAP headers
if ( soapHdrs != null ) {
   for (int i=0; i<soapHdrs.size(); i++) {
      com.ibm.websphere.sibx.smobo.SOAPHeaderType soapHdr =
(com.ibm.websphere.sibx.smobo.SOAPHeaderType)soapHdrs.get(i);
      if (soapHdr.getName().equals("twssHeaders")) {
```

```
         twssHdrs = (commonj.sdo.DataObject)soapHdr.getValue();
       }
   }
} else {
   System.out.println("Custom Primitive AddressTransformation: no soap headers found");
}

// get the policies from the IBM WebSphere Telecommunications Web Services Server Header
java.util.List policies = null;
if ( twssHdrs.isSet("policies") ) {
    policies = (java.util.List)twssHdrs.get("policies/policy");
} else {
   System.out.println("Custom Primitive AddressTransformation: no policies found");
}

commonj.sdo.DataObject policy = null;
System.out.println("Custom Primitive AddressTransformation: policy "+policyName+" set to
default:"+polURIdefault);


for (int x=0; x < policies.size(); x++ ) {
   policy = (commonj.sdo.DataObject)policies.get(x);
   if ( policy.get("attribute").equals(policyName)) {
      polURI = policy.getBoolean("value");
      System.out.println("Custom Primitive AddressTransformation: policy "+policyName+" found,
value: "+policy.getBoolean("value"));
     }
}

// service.config.enableURItrace = false
if (polURI) {
   // now get the requester ID and enhance it means:   user =>  user@example.com
   if ( twssHdrs.isSet("requesterID") ) {
    String reqID = twssHdrs.get("requesterID").toString();
    reqID = "sip:"+reqID+"@example.com";
    twssHdrs.setString("requesterID",reqID);
      System.out.println("Custom Primitive AddressTransformation: new requesterID
is:"+twssHdrs.get("requesterID").toString());
   } else {
      System.out.println("Custom Primitive AddressTransformation: no requesterID found");
   }
}

System.out.println("Custom Primitive AddressTransformation: method exited");
return input1;
```

Archived

5.2.7 Assemble the EAR

Once any custom tooling is done in WebSphere Integration Developer, using either the PX_ESB_xx_FLOW.zip files or your custom flows, the ESB flow application EAR is exported from WebSphere Integration Developer. Before this EAR is ready for execution on the ESB Server, some customizations in the generated code are necessary. The ESB Installer now provides an Ant task to automate these customizations. This flowearpostproc tool is provided by the IBM WebSphere Telecommunications Web Services Server product to provide automation for some specific customizations that are necessary in our standard flows. For each Flow EAR, the deployer should run the flowearpostproc.bat, which uses the flowearpostproc.xml.

Due to the limitations in the WebSphere Integration Developer tooling related to modifications of generated deployment descriptors getting overwritten when WebSphere Integration Developer regenerates, the use of any modifications in the generated deployment descriptors of the IMS IBM WebSphere Telecommunications Web Services Server components is also not supported.

The IMS IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-ins installer contains a flowearpostproc.bat file that can apply a specific set of customizations to the exported EAR (either delivered in the Access Gateway Installer or exported from the WebSphere Integration Developer tool).

In particular, the use of security that requires the customization of the deployment descriptors of the Access Gateway Flow Web Services, such as web.xml and application.xml, is not supported using WebSphere Integration Developer or the embedded ESB server. The flowearpostproc.bat file can be used to update an ESB Flow EAR deployment descriptors so that security roles and URL constraints are applied, and then when the EAR is installed on the stand-alone ESB Server, security can be used.

Note: Under WebSphere Integration Developer (WID_HOME\eclipse\plugins\org.apache.ant_1.6.2), there is an Ant plug-in. This version of Ant can be used to execute these customizations. This tool is provided in the WebSphere Integration Developer Plug-ins Installer, and the flowearpostproc.zip if an iFix is delivered. Make sure you have got at least the version provided with IBM WebSphere Telecommunications Web Services Server V6.2 IFix 0004 Fix Pack and beyond.

Note: The flowearpostproc tool delivered with IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer plug-ins is only used with IBM WebSphere Telecommunications Web Services Server Access Gateway default flows after they have been imported into WebSphere Integration Developer, customized or not, and then exported back out.

This tool is not intended for use on any other flow EARs and any attempt to use the tool on any other flow EAR is not supported.

To create your deployable archive, follow these steps:

1. In WebSphere Integration Developer, select **File** → **Export**. In the Export selection window (Figure 5-12), select **EAR** and click **Next**.

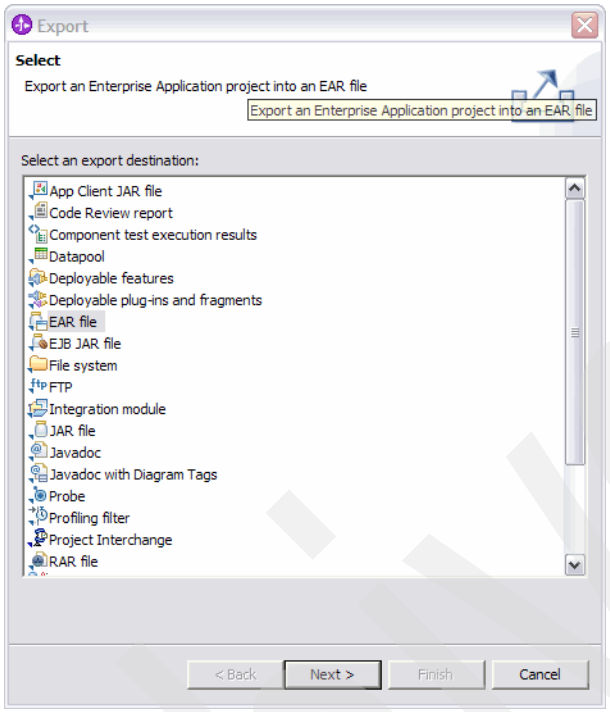


Figure 5-12 Export window

2. In the EAR export window, select the project you want to deploy and the destination directory for the exported EAR file (Figure 5-13).

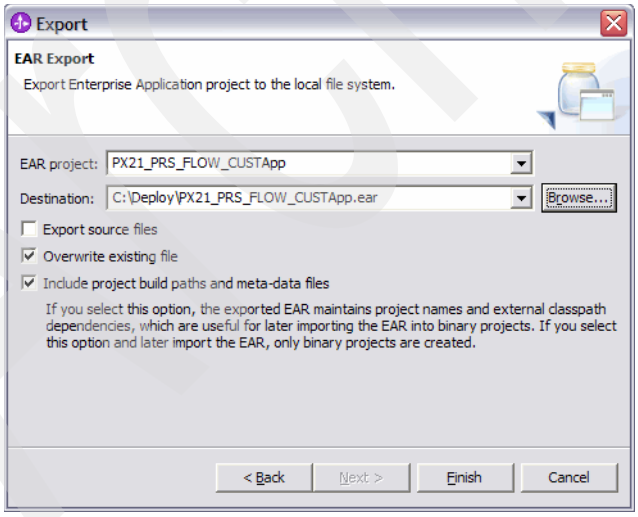


Figure 5-13 EAR export window

You now need to apply the customizations to the exported EAR file by running the post processing tool. In order to run the flowearpostproc.bat to update deployment descriptors, context root, and so on, follow these procedures.

Archived

Set WID_HOME= to point to your WebSphere Integration Developer installation. The flowearpostproc.bat command arguments are shown in Table 5-2.

Table 5-2 Arguments for flowearpostproc.bat

Argument	Description	Example
-buildfile	The XML ant build file	flowearpostproc.xml
-Dear.in	The name of the EAR input file	MyFlowApp.ear
-Dear.dir	Working directory containing the EAR file	C:\\MyDir
-Dflow.name	The name of the mediation flow. It must match the capitalization used for the project name, not EAR name.	PX21_PRS_FLOW

For example, run the command (must be one continuous line) shown in Example 5-16.

Example 5-16 EAR post processing command

```
flowearpostproc.bat -buildfile flowearpostproc.xml -Dear.in="PX_ESB_ALM_FLOW.ear"
-Dear.dir="C:/Deploy" -Dflow.name=PX_ESB_ALM_FLOW
```

Note: The customizations that are made in the EAR by flowearpostproc.xml are:

- The Application Roles and URL constraints are defined for security.
- The Web Root Context is defined for Web service URIs.
- The Web service URIs are defined.

To process the EAR file, follow these steps:

1. Open a command-line window.
2. Go to the WID_HOME/TWSS directory (where the flowearpostproc.bat file is located).
3. Enter this command:

```
flowearpostproc.bat -buildfile flowearpostproc.xml
-Dear.in="PX21_PRS_FLOWCUSTApp.ear" -Dear.dir="C:/Deploy"
-Dflow.name=PX21_PRS_FLOW
```

Note: The flowearpostproc is only provided when customizing one of the IBM WebSphere Telecommunications Web Services Server Access Gateway *default* flows. Each AG default flow project interchange file has a flowearpostproc.properties file that contains values used by the flowearpostproc process that are specific to this flow. This post-processing step is not required for your own custom, non-default flows.

Archive

5.2.8 Deploy the EAR to the runtime environment

Follow the instructions provided in 4.6.1, “Deploying the default mediation flow” on page 105 to install the EAR file on the server.

To run the custom mediation flow, you need to enable global security on the server. In order for your application to receive the requesterID in the SOAP headers, be sure to set the following parameters during the installation of the application:

- In Step 7, “Map security roles to users/groups”, of the installation configuration, check **All Authenticated** (Figure 5-14).

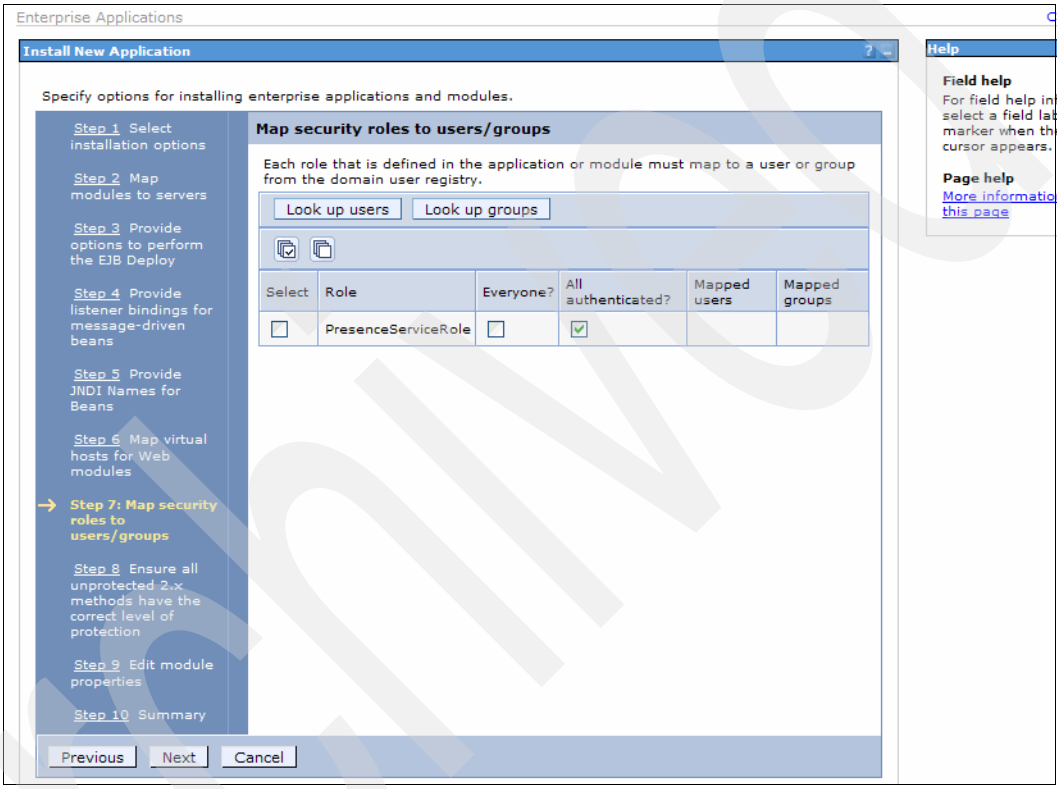


Figure 5-14 Map security roles to users/groups

Archive

- In Step 8, click **Role** and select the **PresenceServiceRole** (Figure 5-15).

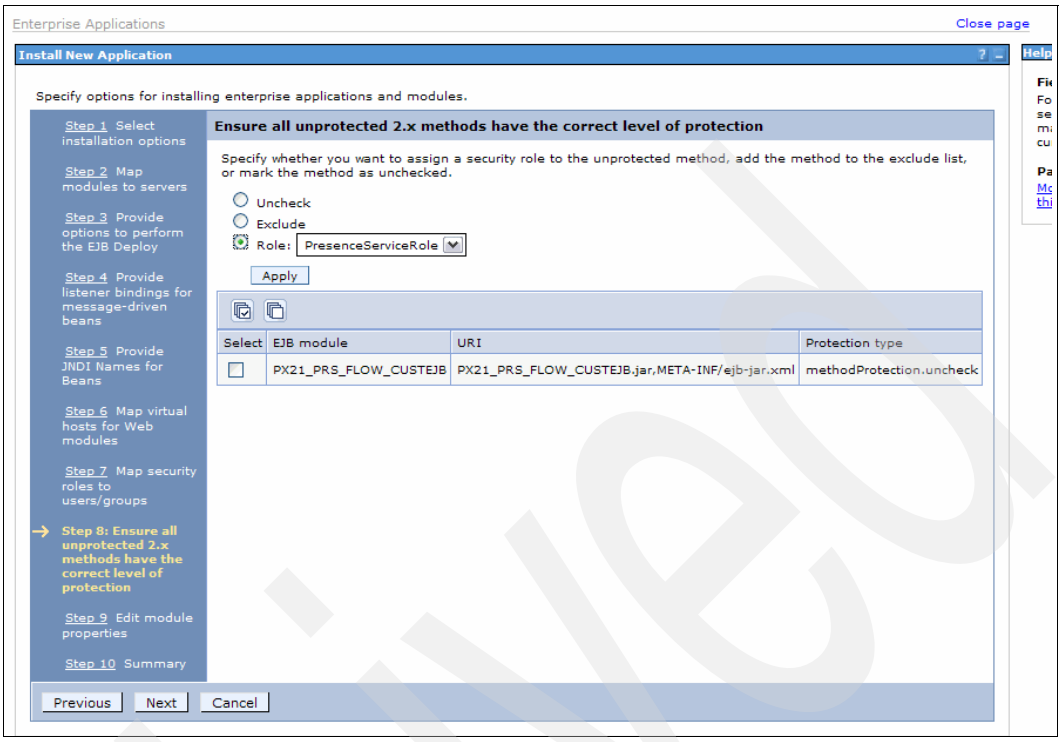


Figure 5-15 Ensure correct level of protection

5.3 Extending the WebSphere Integration Developer Tooling Environment

A more reusable approach to creating custom mediation primitives is to package mediation primitives so that they plug into the WebSphere Integration Developer tooling palette. The palette is the toolbar to the left of the assembly diagram window within the WebSphere Integration Developer tooling.

Eclipse plug-in code must be generated and an icon included for the tooling palette.

Note: The plug-ins are not automatically loaded; they are only loaded when WebSphere Integration Developer is started with the -clean option. In addition, the mediation primitive plug-ins are installed into the eclipse plugins directory of the wid tooling directory (wstools), so the full path is <WID_HOME>\wstools\eclipse\plugins.

If you want to test the plug-in, this requires creating a JAR that contains the mediation primitive runtime code and placing that code within the WebSphere Application Server class path.

Mediation primitives that extend the tooling palette can also have static properties that can be set on the primitives. These properties are described within the Eclipse plug-in descriptor and appear within the properties view below the assembly diagram canvas. Properties can be set on each primitive instance that is added into the canvas. Property values can be promotable or non-promotable. If defined as promotable, their values can be set or changed during run

Archived

time using the WebSphere ESB administration console (see also 5.2.1, “Design guidelines for custom mediation primitive” on page 129).

Figure 5-16 introduces the key steps and artifacts of the plug-in development process.

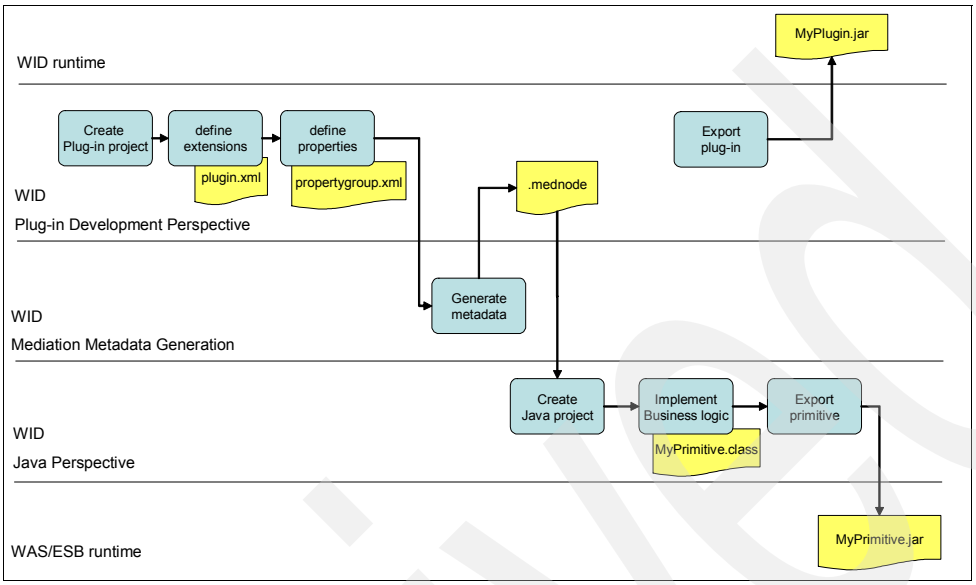


Figure 5-16 The plug-in development process

There are two main sets of activities:

- Describe the plug-in and its properties that represent your custom primitive on the tool using WebSphere Integration Developer and the plug-in development environment.

  Note: Make sure you have the plug-in development capability enabled in WebSphere Integration Developer. Select **Window** → **Preferences**. Expand **Workbench** and click **Capabilities**. Make sure that the **Eclipse Developer** entry is checked.

- Develop the business logic of your custom primitive using the Java development perspective of WebSphere Integration Developer.

The following section will guide you step by step through each of the activities in Figure 5-16 to build a custom primitive plug-in.

Note: In addition to the information mentioned here, we also recommend referencing the WebSphere Integration Developer InfoCenter, which contains detailed descriptions on contributing your own plug-in:

`http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/topic/com.ibm.wbit.help.sib.mediation.ui.doc/topics/rcustspi.html`

5.3.1 Create a plug-in

In this section, we will create a mediation primitive Eclipse plug-in. This plug-in will provide a new mediation primitive to the WebSphere Integration Developer mediation flow editor palette.

Create the plug-in project

1. Using WebSphere Integration Developer, open the plug-in development perspective by selecting **Window** → **Open** → **Perspective Other...** and choose **Plug-in Development** from the offered list.

Note: If you do not see Plug-in Development in the list, check the **Show all** option.

2. Create a plug-in project by selecting **File** → **New** → **Plug-in Project**. The New Plug-In Project window opens (Figure 5-17).

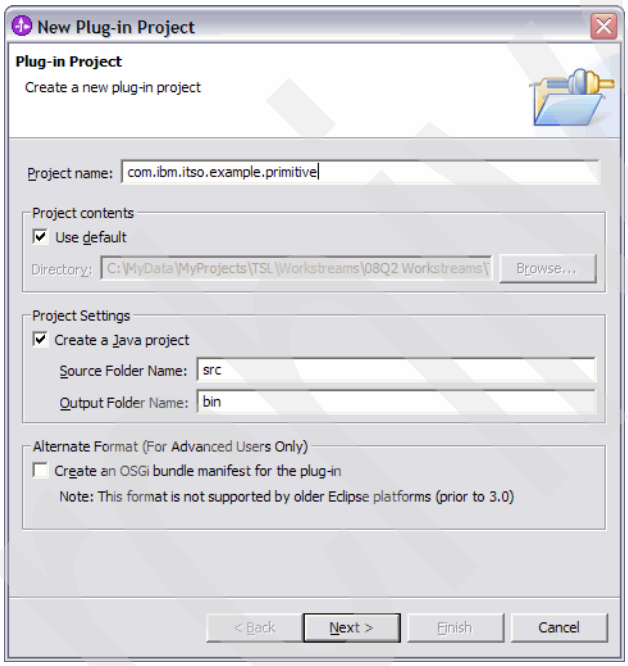


Figure 5-17 New Plug-In Project window

3. Enter com.ibm.itso.example.primitive as the project name. Keep the default options, and click **Next**.

4. Uncheck the option to generate a Java class (see Figure 5-18) and click **Finish**.

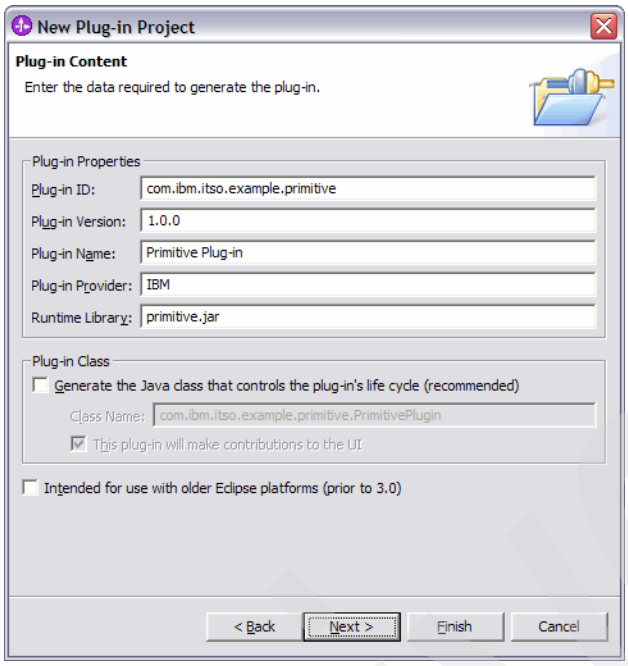


Figure 5-18 New plug-in content

The new project is created and the content is displayed in the Package Explorer.

Edit plugin.xml

Create these extensions in the plugin.xml:

- com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveHandlers

  Defines the terminals of your primitive, and identifies the properties file where the primitive's properties are defined and the .mednode file that you will generate in 5.3.2, “Generate the mediation meta-data” on page 161.

- com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveUIContribution

  Builds on medationPrimitiveHandler and adds information to contribute the medationPrimitiveHandler to the Mediation Flow editor's palette.

1. Open plugin.xml in the Plug-in Manifest editor, and switch to the Extensions page.
2. Click **Add**. This opens the New Extension wizard (Figure 5-19 on page 157).

Archived

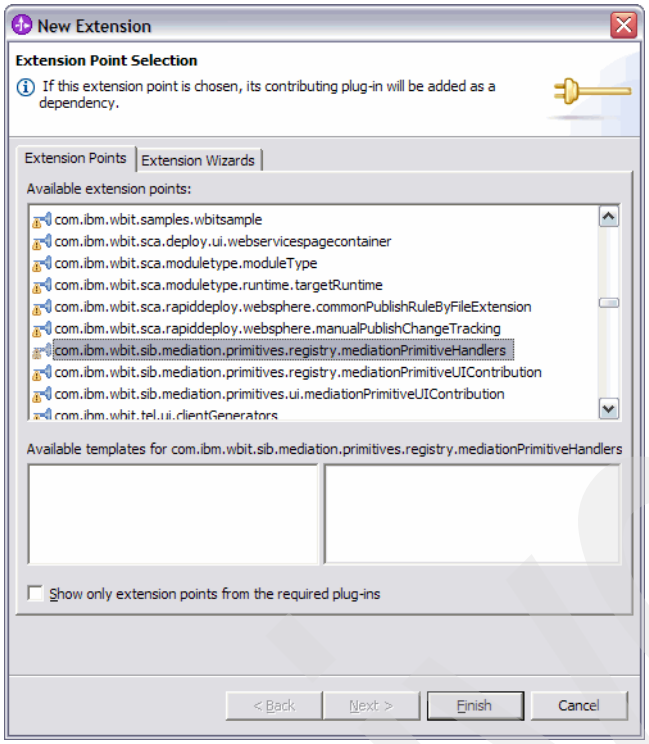


Figure 5-19 New Extension

3. Uncheck **Show only extension points from required plug-ins** to view the list of plug-ins.
4. Select **com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveHandlers**. Click **Finish**. An entry appears in the All Extensions list (Figure 5-20).

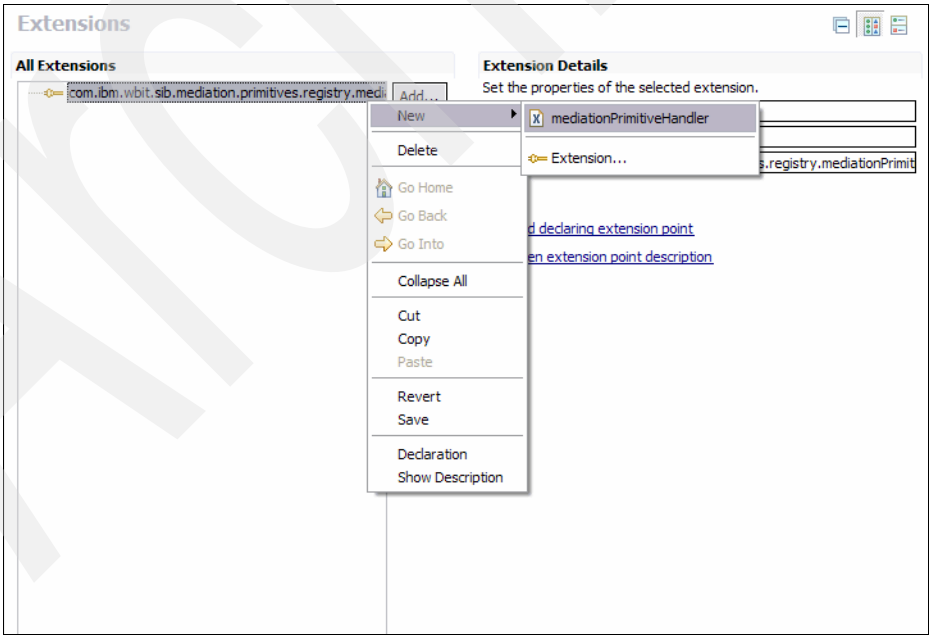


Figure 5-20 Add New Handler

Archived

5. In Figure 5-20, right-click **com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveHandlers** and select **New** → **medationPrimitiveHandler**. A medationPrimitiveHandler is added for the extension.
6. Select the handler and set its properties, as shown in Figure 5-21.

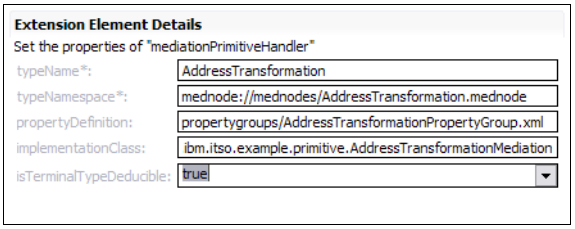


Figure 5-21 Extension Element Details window

Note: The typeNamespace must begin with mednode://mednodes, and end with FileName.mednode. A file of this name will be created when we generate the mediation metadata.

7. Add short and long description and terminal categories for the handler.
8. Select the handler, right-click it, and select **New...** → **shortDescription**.
9. Click **Body Text** at the bottom of the Extensions page. Enter a short description for the primitive (“Custom MediationPrimitive for AddressTransformation”). Click **Add**.
10. Similarly, add text for the longDescription (“This mediation transforms requesterIDs into SIP URIs”).
11. Add the input terminal for the handler by selecting it, right-click it, and select **New...** → **terminalCategory**.
12. In Extension Element Details, select **input** as the type and enter “in” as the name.
13. Add the output terminal for the handler by selecting it, right-click it, and select **New...** → **terminalCategory**.
14. In Extension Element Details, select **output** as the type and enter “out” as the name.
15. Add the fault terminal for the handler by selecting it, right-click it, and select **New...** → **terminalCategory**.
16. In Extension Element Details, select **output** as type and enter “fault” as the name.
17. Add the fail terminal for the handler by selecting it, right-click it, and select **New...** → **terminalCategory**.
18. In Extension Element Details, select **fail** as the type and enter “fail” as the name.

Note: The fail terminal MUST be named fail.

Figure 5-22 on page 159 shows the fully specified handler.

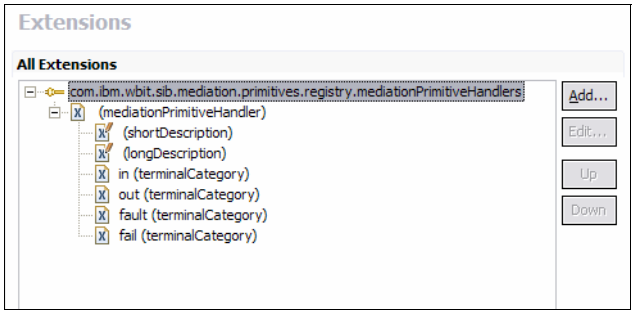


Figure 5-22 Mediation primitive handler with details

19. We need to add another extension to specify the UI properties. Click **Add...** to open the New Extension window.
20. In the New Extension window, select **com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveUIContribution** from the list and click **Finish**.
21. IOn the extension page, right-click **com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveUIContribution** and select **New** → **mediationPrimitiveUIContribution**.
22. Set the properties of mediationPrimitiveUIContribution, as shown in Figure 5-23, and save them.

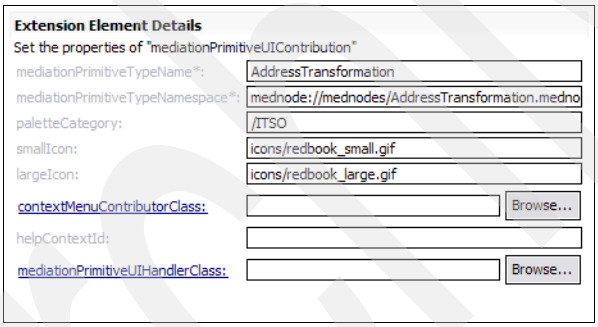


Figure 5-23 Extension Element Details for the UI contribution

Note: Enter just a / for the paletteCategory to place your custom primitive icon in the root of the palette. Enter /ITSO to create a new group of ITSO primitives.

Create the required folders in the plug-in project

We need to create additional folders for the plug-in project. One folder will hold the images of the icons used to represent the custom mediation primitive in the flow editor. The second folder contains the propertygroup-files, which describe the properties of the custom primitive.

Do to following steps to create the folders:

1. Create a folder named *icons* in the plug-in project. Select the plug-in project, right click it, and select **New** → **Folder**. Enter icons as the Folder Name.
2. Place your icons into the new folder. The small icon (16x16) will appear on the palette. The large icon (32 x 32) will appear on the canvas. Icon file names must match the ones you have specified in Figure 5-23.

Archived

3. Create a folder named *propertygroups* in the plug-in project, and create a property group XML file in this folder by selecting **File** → **New** → **Other** → **XML** → **XML**. Click **Next**. Choose **Create an XML file from scratch**. Click **Next**. Name the file AddressTransformationPropertyGroup.xml and click **Finish**.

Note: The propertygroup filename must match the one you have specified in Figure 5-21 on page 158.

Your project structure should now look similar to the one shown in Figure 5-24.

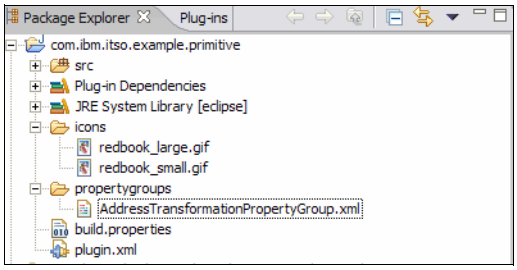


Figure 5-24 Plug-in project structure

Edit the propertygroup file

In AddressTransformationPropertyGroup.xml file, we will describe the properties of this new primitive, so as to derive the UI for the properties view Details page in the mediation flow editor. The Java class, which later implements the business logic of the primitive, must have getter and setter methods that correspond to each of these properties.

For our AddressTransformation primitive, we will just add one property, sipDomain, that enables the mediation flow developer to specify the domain name, that will be appended to the requester ID to build the SIP URI. The property's default is set to example.com.

Add the required properties to AddressTransformationPropertyGroup.xml by entering the XML code shown in Example 5-17.

Example 5-17 AddressTransformationPropertyGroup.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<pg:BasePropertyGroups name="AddressTransformationPropertyGroups"
resourceBundle="TwssRedbookExamples"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:pg="http://www.ibm.com/propertygroup/6.0.1">

   <propertyGroup name="AddressTransformationPropertyGroup"
   xsi:type="pg:BasePropertyGroup" >

   <!-- domain name using String Property -->
      <property name="sipDomain" displayName="Domain Name"
      defaultValue="example.com" required="true" propertyType="String"
      xsi:type="pg:ConstraintSingleValuedProperty">
         <description>
            Your domain name, required to build SIP URIs
         </description>
      </property>

   </propertyGroup>
```

```
</pg:BasePropertyGroups>
```

Do not forget to save your project.

Note: When promoting properties, the WebSphere Integration Developer tooling generates an Alias name and Alias value for the promoted property. These aliases are what is visible in the ESB Admin console. The generated names are always made unique by added numeric values to the names. If you are adding the same mediation primitive to the flow several times (in different operations), then you can end up with a multiple of aliases for the same property.

We recommend that you change the alias name to be consistent for all operations if you want to manage a single property from the Admin Console.

Note: You will find the specification of the propertygroup.xml in the WebSphere Integration Developer InfoCenter found at:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r1mx/index.jsp

5.3.2 Generate the mediation meta-data

In this section, we will generate the mediation metadata (.mednode file) for the mediation primitive using the Mediation Metadata Generation view. The .mednode file contains the runtime representation of the mediationPrimitiveHandlers and must be placed at the root of the Java project that you create in 5.3.2, “Generate the mediation meta-data” on page 161.

Launch the runtime workbench from the plug-in development perspective:

1. From the menu, select **Run** → **Run As** → **Runtime Workbench**. If necessary, create a new configuration by selecting **Runtime Workbench** and clicking **New**. Accept the defaults. Click **Run**. This will open a new Eclipse IDE.
2. In the new IDE that is launched, select **Window** → **Show view** → **Other** → **Mediation Development** → **Mediation Metadata Generation**.

Archive

3. The mediation primitives that you created are displayed here. Use this view to generate a .mednode file for your primitive. Select the primitives that you want and click **Generate**. A status of OK indicates that the .mednode file was successfully generated (Figure 5-25).

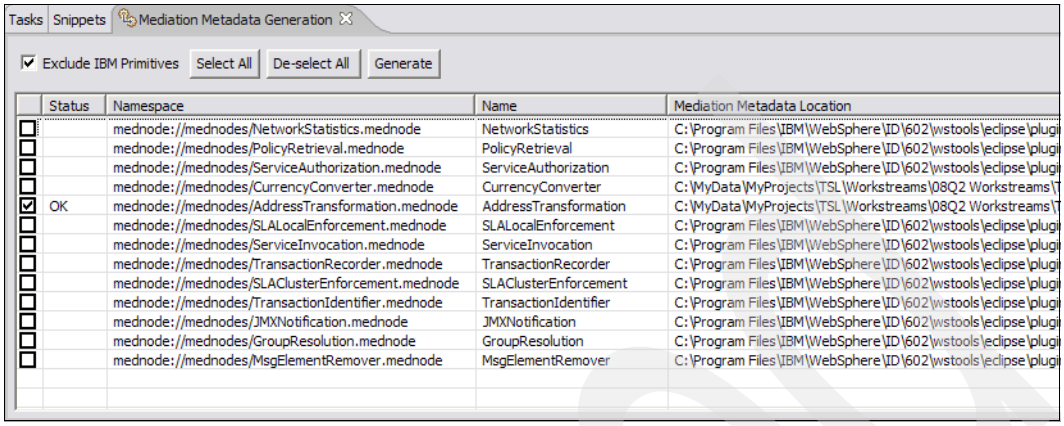


Figure 5-25 Mediation Metadata Generator

A file named AddressTransformation.mednode is created in the plug-in project's mednodes folder. You may need to refresh your view to see this file, as shown in Figure 5-26.

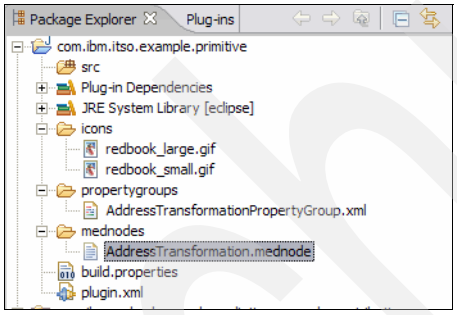


Figure 5-26 The generated .mednode file

5.3.3 Develop the mediation business logic

Create a Java project, and write the code to implement your mediation primitive.

Write your Java implementation code for the mediation primitive in the inherited mediate() method. The mediate method takes an InputTerminal and a DataObject. Use the InputTerminal only if you have multiple input terminals. The DataObject is your message. You can use the getters and setters in the DataObject interface to read and write the values in your messages, identified through an XPath expression. DataObject is part of the Service Data Object (SDO) emerging standard. This message parameter can also be cast to a ServiceMessageObject (SMO) in the com.ibm.websphere.sibx.smobo package as part of the Service Message Object APIs. This interface is useful for accessing individual sections of the Service Message Object, such as the body, context, and headers. See the examples in 5.2.2, “Key custom primitive functions” on page 131 for more details on how to work with the SMO.

Follow these step to create the Java project:

1. Create a Java project by selecting **File** → **New** → **Project** → **Java Project**. Click **Next**. This opens the New Java Project window (Figure 5-27 on page 163).

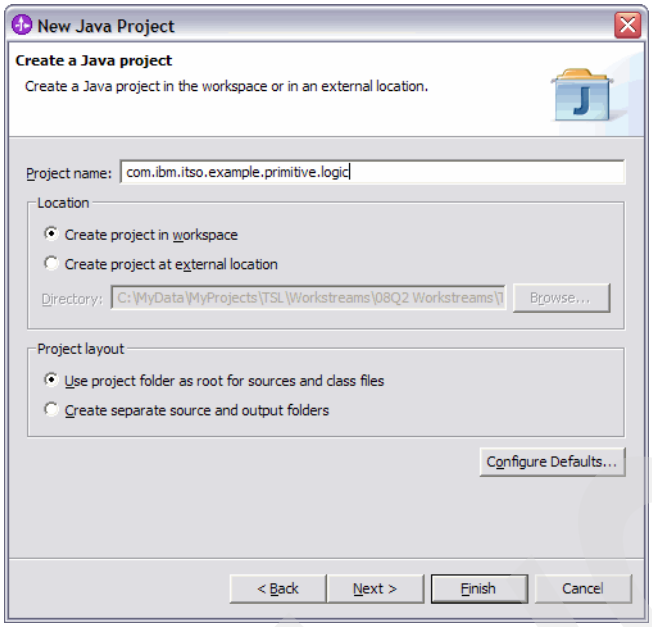


Figure 5-27 New Java Project

2. Enter the project name, com.ibm.itso.example.primitive.logic, and click **Next**.
3. Switch to the Libraries page, and click **Add Library**. In the Add Library window (Figure 5-28), select **WebSphere ESB Server v6.0** and click **Next**.

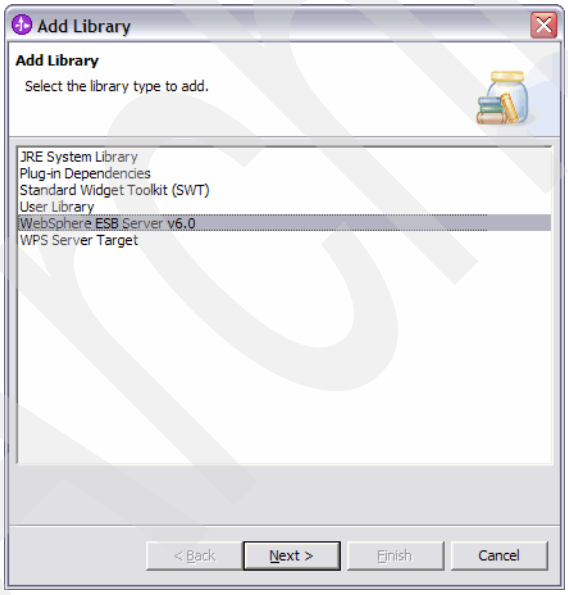


Figure 5-28 Add Library

Archived

4. In the next window (Figure 5-29), check the option to configure WebSphere ESB Server's classpath. Click **Finish**.

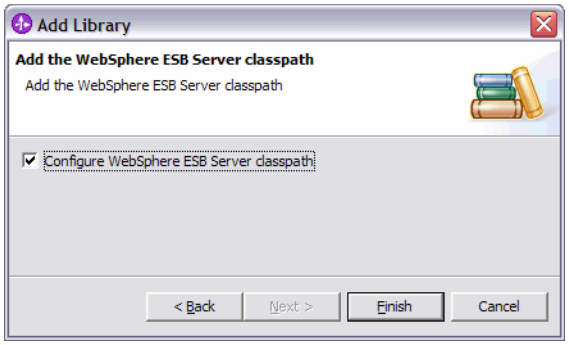


Figure 5-29 Add classpath

5. Click **Finish** in the New Java Project window (Figure 5-30) to create the project.

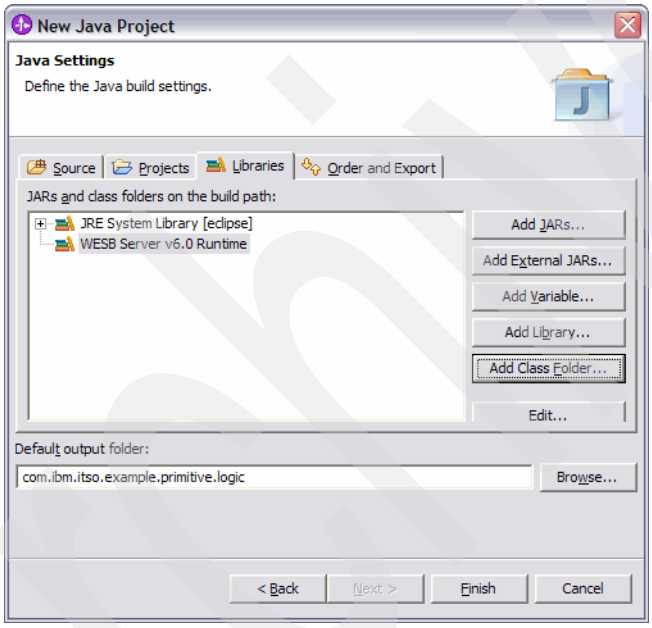


Figure 5-30 Java project settings

6. Copy AddressTransformation.mednode from the plug-in project (see “Generate the mediation meta-data” on page 161) to the root of the Java project.
7. Switch to the Java perspective. Select the Java project and select **File** → **New** → **Class**. This opens the New Java Class window (Figure 5-31 on page 165).

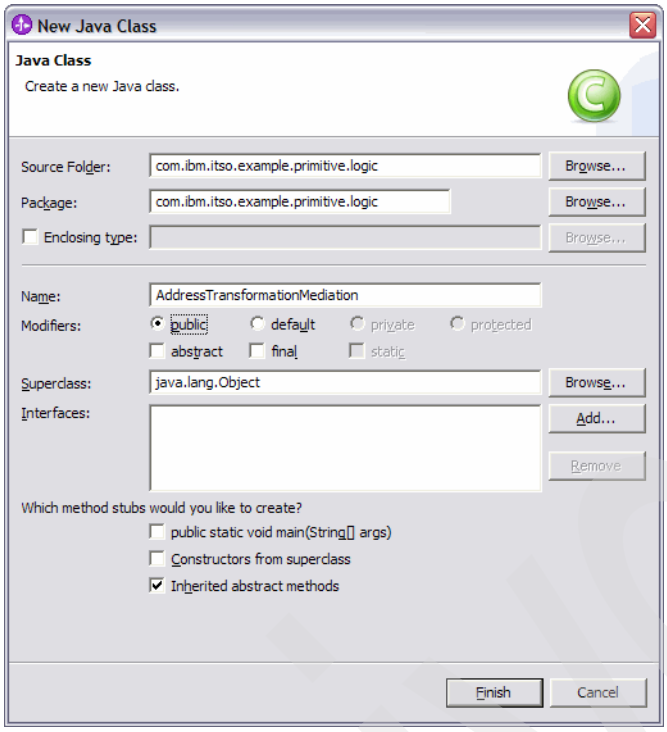


Figure 5-31 New Java class

8. Enter com.ibm.itso.example.primitive.logic as the package name and AddressTransformationMediation as the class name. Click **Finish**.

Your Package Explorer now contains two projects with a structure, as shown in Figure 5-32.

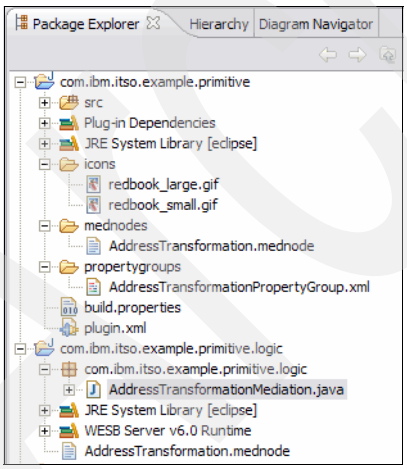


Figure 5-32 Plug-in and implementation project structure

9. Double click the AddressTransformationMediation.java file to open the Java editor.
10. The source code for this class is provided in the additional materials of this IBM Redbooks publication. (See Appendix E, “Additional material” on page 399 for details on how to download this material.) It makes use of all the snippets introduced in 5.2.2, “Key custom primitive functions” on page 131. Once downloaded, just copy the Java code to this Java file.

5.3.4 Deploy the plug-in to the tool environment

Deploy your plug-in so that your mediation primitives appear in the Mediation Flow Editor palette:

1. Open the plug-in project's build.properties file in the Build Properties Editor, and make sure that the mednodes, icons, and propertygroups folders are selected under Binary Build (Figure 5-33).

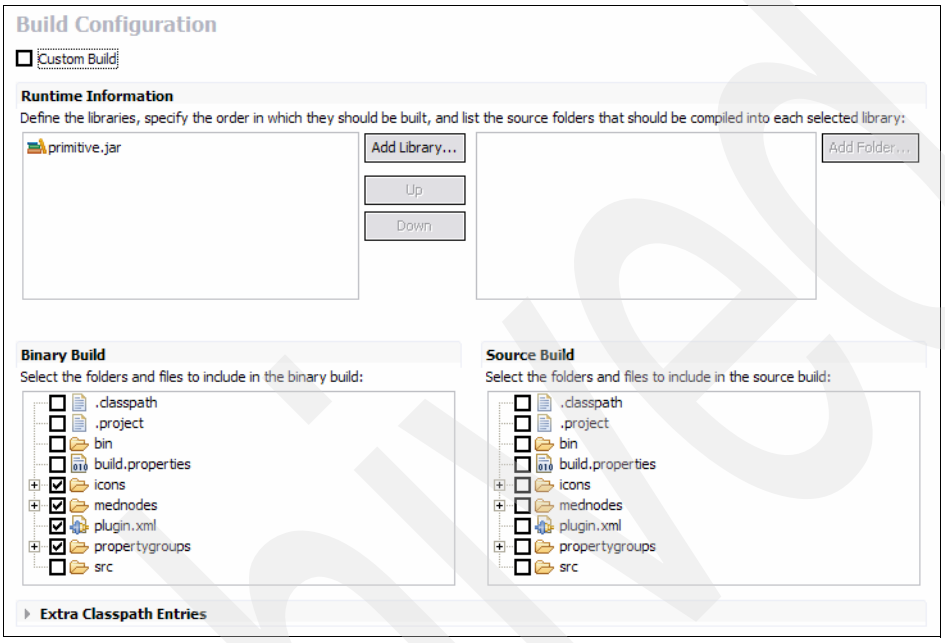


Figure 5-33 Build properties

2. To export the project, right-click the plug-in project and select **Export** → **Deployable plug-ins and fragments**. Click **Next** to open the Export plug-ins window (Figure 5-34 on page 167).

Archived

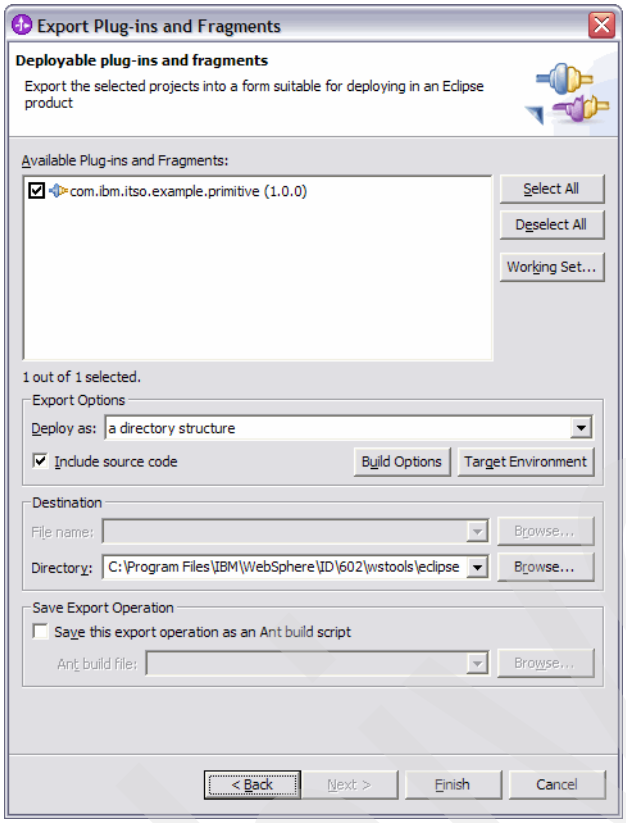


Figure 5-34 Export plug-ins and fragments window

3. Select **Deploy as a directory structure** and specify the destination directory WIDInstall\wstools\eclipse. Click **Finish** to start the export.
4. Shut down WebSphere Integration Developer and start it using the -clean option.
5. Open a mediation flow component in the Mediation Flow editor. The AddressTransformation mediation primitive's icon should appear in the palette ( ).

Archived

5.3.5 Deploy the primitives to run time

The Java implementation of the plug-in needs to be deployed to the WebSphere ESB runtime environment:

1. Export the Java project as JAR. Right-click the Java project and select **Export** → **JAR file** and click **Next**. This opens the JAR Export window (Figure 5-35).

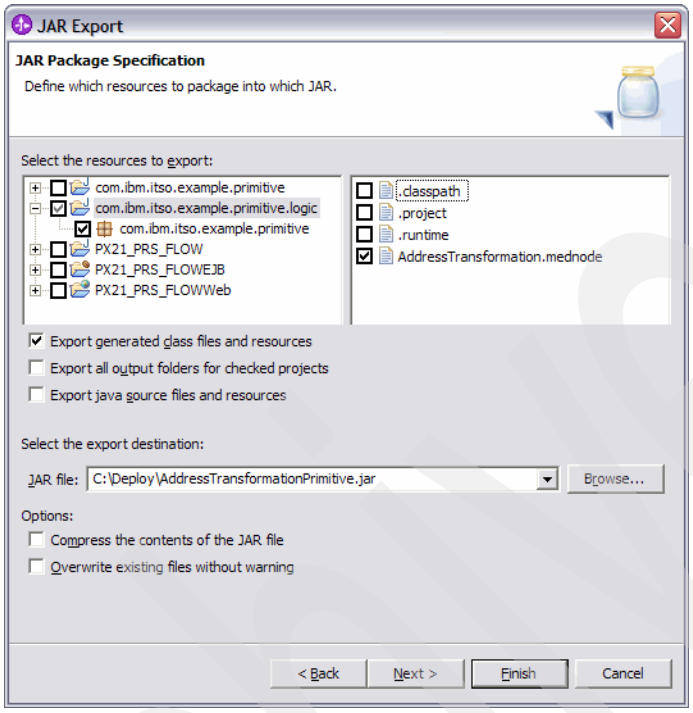


Figure 5-35 Export JAR window

2. In the root folder, select **.mednode** only, but keep the Java class selected. Export the jar to WESB_HOME/lib/ext. Click **Finish** to start the export.

Note: If you intend to test your primitive in a mediation flow using the universal test environment, you need to export the JAR file to WIDInstall\runtimes\bi-v6\lib\ext.

5.3.6 Using the plug-in to customize a flow

Now let us customize a default mediation flow using the AddressTransformation primitive we just have developed and deployed.

We base our sample on the same use case in 5.2.3, “Use case description for the customization” on page 142:

1. Load the default PresenceSupplier Mediation module into WebSphere Integration Developer, as described in 5.2.4, “Load the default flow” on page 145.
2. Open the flow editor and select the PresenceSupplier publish operation. The flow is displayed in the editor now.
3. Select the **Address Transformation**  primitive in the palette and drag it onto the flow editor.

4. First, disconnect the existing connection at the place where we want to insert our custom primitive. Click the output connection of the SLAClusterEnnforcement primitive and delete it.
5. Now connect the custom primitive into the flow. Click the Output terminal of the SLAClusterEnforcement primitive and drag the wire to the Input terminal on the AddressTransformation primitive.
6. Click the Output terminal of the AddressTransformation primitive and drag the wire to the Input terminal on the Service Invocation primitive.
7. Click the Fault terminal of the AddressTransformation primitive and drag the wire to the Input terminal on the NetworkStatistics primitive.
8. Select the **AddressTransformation primitive** and click the **Details** tab in the properties view to adjust the properties (Figure 5-36). You will see the custom property “Domain Name” with its default, as specified in “Edit the propertygroup file” on page 160.

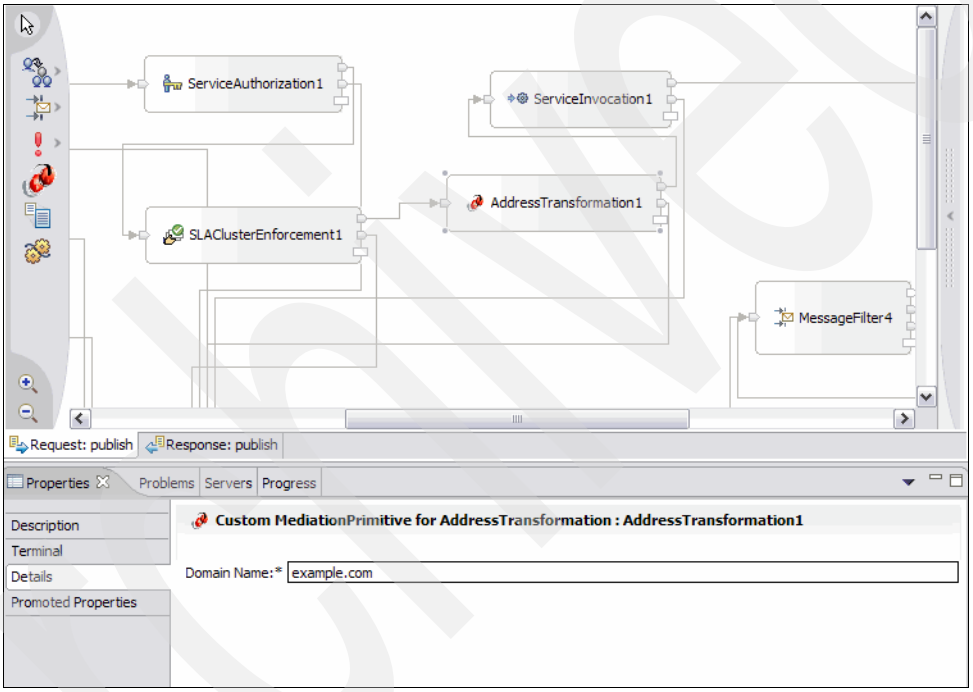


Figure 5-36 AddressTransformation primitive inserted into the flow

You are now ready to export the customized flow and deploy it to the run time to test it (see 5.2.7, “Assemble the EAR” on page 149 and 5.2.8, “Deploy the EAR to the runtime environment” on page 152 for reference).

5.3.7 Test your custom flow and primitive

This section introduces some test cases using the primitive and the flow we have just created. The prerequisites to run this flow are:

- A test client (for example, WebSphere Integration Developer or any other SOAP test tool)
- A working service implementation of the PresenceSupplier publish operation
- A working presence server and XDMS server
- A working Service Policy Manager used by the access gateway to retrieve policy data

Once you have deployed your mediation module to the server run time, you are ready to send SOAP requests to test the flow.

For our tests, we have used a publish request, as shown in Example 5-18.

Example 5-18 Sample publish request message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:loc="http://www.csapi.org/schema/parlayx/presence/supplier/v2_3/local">
   <soapenv:Header/>
   <soapenv:Body>
      <loc:publish>
         <loc:presence>
            <lastChange>2002-11-11T11:11:11</lastChange>
             <note>testing my primitive</note>
            <typeAndValue>
               <UnionElement>Activity</UnionElement>
               <Activity>Travel</Activity>
            </typeAndValue>
         </loc:presence>
      </loc:publish>
   </soapenv:Body>
</soapenv:Envelope>
```

Now let us have a look at how our custom primitive behaves by running some test cases.

We will look at the snapshot of traces that shows:

- The input message, when received by the AddressTransformation primitive
- Any information written to the log by the AddressTransformation primitive
- The input message, when received by the subsequent Service Invocation primitive

To increase readability, we have removed some of the headers in the message log that are not important for the test cases.

Test case 1- Custom policy defined (false)

The policy service.config.enableURItransformation is set to false.

The resulting entries in trace.log are shown in Figure 5-37 on page 171.

Archived

```
[4/10/08 5:36:52:290 EDT] 0000004f AddressTransf 3   mediate Entered method
[4/10/08 5:36:52:292 EDT] 0000004f AddressTransf 3   mediate Input message: {0}<?xml version="1.0" encoding="UTF-8"?>
<ServiceMessageObject:smo ">
  <context>
    <transient xsi:type="_pX2:ExceptionType"/>
  </context>
  <headers>
    <SMOHeader>
      <MessageUUID>37B1D68E-0119-4000-E000-4FB4C0A80166</MessageUUID>
      <Version>
        <Version>6</Version>
        <Release>0</Release>
        <Modification>2</Modification>
      </Version>
      <MessageType>Request</MessageType>
    </SMOHeader>
    <SOAPHeader>
      <nameSpace>http://www.ibm.com/schema/twss/v1_0</nameSpace>
      <name>twssHeaders</name>
      <prefix>twss</prefix>
      <value xsi:type="_v10:twssHeaders">
        <globalTransactionID>37B1D68E-0119-4000-E000-4FB4C0A80166</globalTransactionID>
        <requesterID>user1</requesterID>
        <policies>
          <policy attribute="service.config.enableURItransformation" value="false"/>
        </policies>
        <serviceID>http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface</serviceID>
      </value>
    </SOAPHeader>
  </headers>
  <body xsi:type="interface:PresenceSupplier_publishRequest">
  </body>
</ServiceMessageObject:smo>
[4/10/08 5:36:52:293 EDT] 0000004f AddressTransf 3   getBooleanPropertyValue service.config.enableURItransformation found, v
[4/10/08 5:36:52:293 EDT] 0000004f AddressTransf 3   mediate Policy set to NOT transform Addresses
[4/10/08 5:36:52:294 EDT] 0000004f AddressTransf 3   mediate Exiting method
```

Figure 5-37 Test case 1 trace

The AddressTransformation primitive does not change the requestID. The message is forwarded to the service implementation, which responds with a fault when it receives an invalid SIP URI.

The response is shown in Example 5-19.

Example 5-19 Test case 1 response

```
<soapenv:Envelope">
   <soapenv:Header/>
   <soapenv:Body>
      <soapenv:Fault>
         <faultcode
xmlns:p826="http://www.csapi.org/schema/parlayx/common/v2_1">p826:ServiceException
</faultcode>
         <faultstring>SOAX7551</faultstring>
         <detail encodingStyle="">
            <_v21:ServiceException
xmlns:_v21="http://www.csapi.org/schema/parlayx/common/v2_1">
               <messageId>SOAX7551</messageId>
               <text>SOAX7551E:Caught Exception in _ method for Requester URI
_.</text>
               <variables>publish</variables>
               <variables>user1</variables>
            </_v21:ServiceException>
         </detail>
      </soapenv:Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

Archived

Test case 2 - Custom policy defined (true)

This test case covers the actual address transformation.

The trace that is written by the AddressTransformation primitive is shown in Figure 5-38.

```
[4/10/08 4:35:27:869 EDT] 00000068 AddressTransf 3   mediate Entered method
[4/10/08 4:35:27:871 EDT] 00000068 AddressTransf 3   mediate Input message: {0}<?xml version="1.0" encoding="UTF-8"?>
<ServiceMessageObject:smo ">
  <context>
    <transient xsi:type="_pX2:ExceptionType"/>
  </context>
  <headers>
    <SMOHeader>
      <MessageUUID>3779614F-0119-4000-E000-4CE9C0A80166</MessageUUID>
      <Version>
      </Version>
      <MessageType>Request</MessageType>
    </SMOHeader>
    <SOAPHeader>
      <nameSpace>http://www.ibm.com/schema/twss/v1_0</nameSpace>
      <name>twssHeaders</name>
      <prefix>twss</prefix>
      <value xsi:type="_v10:twssHeaders">
        <globalTransactionID>3779614F-0119-4000-E000-4CE9C0A80166</globalTransactionID>
        <requesterID>user1</requesterID>
        <policies>
          <policy attribute="service.config.enableURItransformation" value="true"/>
        </policies>
        <serviceID>http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface</serviceID>
      </value>
    </SOAPHeader>
  </headers>
  <body xsi:type="interface:PresenceSupplier_publishRequest">
  </body>
</ServiceMessageObject:smo>
[4/10/08 4:35:27:871 EDT] 00000068 AddressTransf 3   getBooleanPropertyValue service.config.enableURItransformation found, va
[4/10/08 4:35:27:872 EDT] 00000068 AddressTransf 3   mediate Exiting method
[4/10/08 4:35:27:873 EDT] 00000068 ServiceInvoca 3 mediate Entered method
```

Figure 5-38 Test case 2 - AddressTransformation

The message that is received by the subsequent primitive (the Service Invocation) is shown in Figure 5-39.

```
[4/10/08 4:35:27:872 EDT] 00000068 AddressTransf 3   mediate Exiting method
[4/10/08 4:35:27:873 EDT] 00000068 ServiceInvoca 3 mediate Entered method
[4/10/08 4:35:27:874 EDT] 00000068 ServiceInvoca 3 mediate Input message: <?xml version="1.0" encoding="UTF-8"?>
<ServiceMessageObject:smo">
  <context>
    <transient xsi:type="_pX2:ExceptionType"/>
  </context>
  <headers>
    <SMOHeader>
      <MessageUUID>3779614F-0119-4000-E000-4CE9C0A80166</MessageUUID>
      <Version>
      </Version>
      <MessageType>Request</MessageType>
    </SMOHeader>
    <SOAPHeader>
      <nameSpace>http://www.ibm.com/schema/twss/v1_0</nameSpace>
      <name>twssHeaders</name>
      <prefix>twss</prefix>
      <value xsi:type="_v10:twssHeaders">
        <globalTransactionID>3779614F-0119-4000-E000-4CE9C0A80166</globalTransactionID>
        <requesterID>sip:user1@example.com</requesterID>
        <policies>
          <policy attribute="service.config.enableURItransformation" value="true"/>
          <policy attribute="service.config.requestIDorig" value="user1"/>
        </policies>
        <serviceID>http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface</serviceID>
      </value>
    </SOAPHeader>
  </headers>
  <body xsi:type="interface:PresenceSupplier_publishRequest">
  </body>
</ServiceMessageObject:smo>
```

Figure 5-39 Test case 2 - Service Invocation

Archived

As we can see, the incoming requesterID “user1” has been transformed to the “sip:user1@example.com” format. The original value has been added as a new policy.

The resulting response from the service implementation is shown in Example 5-20.

Example 5-20 Test case 2 response

```
<soapenv:Envelope">
   <soapenv:Header/>
   <soapenv:Body>
      <_local:publishResponse
xmlns:_local="http://www.csapi.org/schema/parlayx/presence/supplier/v2_3/local"/>
   </soapenv:Body>
</soapenv:Envelope>
```

Test case 3 - no custom policy defined

In this test case, the custom flow should basically behave as described in “Test case 2 - Custom policy defined (true)” on page 172.

As the custom policy cannot be found in the header, the custom primitive will use a predefined default value, which in this test case has been set to true.

Figure 5-40 shows the output of the primitive, which indicates that a default value has been used.

```
</ServiceMessageObject:smo>
[4/10/08 6:45:27:871 EDT] 00000068 AddressTransf 3   getBooleanPropertyValue service.config.enableURItransformation
                                                                                not found, using default value (true)
[4/10/08 6:45:27:872 EDT] 00000068 AddressTransf 3   mediate Exiting method
[4/10/08 6:45:27:873 EDT] 00000068 ServiceInvoca 3 mediate Entered method
[4/10/08 6:45:27:874 EDT] 00000068 ServiceInvoca 3 mediate Input message: <?xml version="1.0" encoding="UTF-8"?>
<ServiceMessageObject:smo">
```

Figure 5-40 Test case 3 trace

5.4 Develop a custom mediation flow

This section shows how to develop a new mediation flow from scratch. We cover the full development life cycle of a new mediation flow for IBM WebSphere Telecommunications Web Services Server, starting by designing the solution, developing and testing the flow, and finally assembling and deploying the executables to the runtime environment.

5.4.1 Design guidelines for new mediation flows

A custom access gateway flow is created using the IBM WebSphere Integration Developer and the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Access Gateway mediation primitive plug-ins. These are available as part of the IBM WebSphere Telecommunications Web Services Server product and are provided on a separate image from the IBM WebSphere Telecommunications Web Services Server runtime components. The process is similar to creating a standard WebSphere Integration Developer Mediation Module with a few specific steps to include IBM WebSphere Telecommunications Web Services Server requirements.

The mandatory components provide base function for an Telecom Web Services Access Gateway flow and are required by downstream mediation primitives.

The following mediation primitives are mandatory within a base Access Gateway configuration and flow:

Message Removal The message removal mediation primitive removes specific SOAP headers that may have been passed into the Access Gateway by the client application.

Transaction Identifier The transaction identifier mediation primitive determines the requester of the message and the global transaction ID for the message.

Policy Retrieval The policy retrieval mediation primitive retrieves policy data for an operation and populates the policy SOAP headers within the data object passed between mediation primitives. This policy data is used to drive the execution logic of the flow.

Service Invocation Performs runtime selection of which back-end service implementation to invoke.

The following components are optional plug-ins and are used by the default Telecom Web Services Access Gateway flow. Use of the service invocation mediation primitive is highly recommended for flows that invoke Telecom Web Services service implementations:

Network Statistics Records Web service request/response entry and exit information.

Transaction Recorder Writes information about a Web service request passing through the Access Gateway to a database table for accounting purposes. (See the note below for additional discussion on how this differs from Message Logger.)

JMX Notification Used to emit Java Management Extensions (JMX) notifications by means of the WebSphere Application JMX infrastructure.

Service Authorization Provides fine-grained authorization for access to Web services.

SLA Enforcement Controls the type of traffic admitted into the network for individual requesters.

Group Resolution (Parlay X-specific) Resolves group URIs to a collection of member URIs.

The following mediation primitives are provided as part of the WebSphere WebSphere Integration Developer/ ESB product offering:

Message Logger Performs logging. Using the Flow Editor in WebSphere Integration Developer, modify the Root property of the mediation primitive to specify the XPath expression to specify the part of the SMO to be logged.

XSL Transformation You can use the XSLT mediation primitive to change the headers or the body of your messages.

CEI Event Emitter Emits Common Event Infrastructure (CEI) events.

Archived

Note: How is Transaction Recorder different from Message Logger?

- Transaction Recorder: Logs information about the Web service request to a database for accounting purposes and has a description of the transaction for other accounting data.
- Message Logger: Logs an XML transcoded copy of the Service Message Object (SMO). This mediation primitive contains a property named 'Root', which represents an XPATH statement defining the scope of the message to be logged. You can specify /, /body, /headers, or your own XPath expression. / refers to the complete SMO, /body refers to the body section of the SMO, and /headers refers to the headers of the SMO. If you specify your own XPath expression, the part of the SMO you specify is processed. The message to be logged is converted to XML from the point specified by Root.

Tip: How can the Message Logger be customized to only record what the customer needs?

The incoming SOAP message could contain some confidential information that the customer may not be in a position to log, so it become necessary to customize this component in order to record only, for example, the header information received from the user.

This can be accomplished using the Flow Editor in WebSphere Integration Developer and modifying the Root property of the mediation primitive to specify the XPath expression to specify the part of the SMO to be logged.

5.4.2 Use case description for the new mediation flow

For our example, we assume that an operator has implemented an IMS based conferencing service, which he now wants to expose to third parties, so that they can make use of this functionality when building new applications.

We assume the conferencing service implementation is available and exposes a Web service interface. In the next sections, we demonstrate how to build a custom mediation flow that:

- Exports the conferencing Web service interface.
- Applies default primitives to incoming requests.
- Applies default primitives to the responses.
- Forwards the request to the service implementation.

Figure 5-41 shows the IBM WebSphere Telecommunications Web Services Server Access Gateway and the artifacts we use or create in this section.

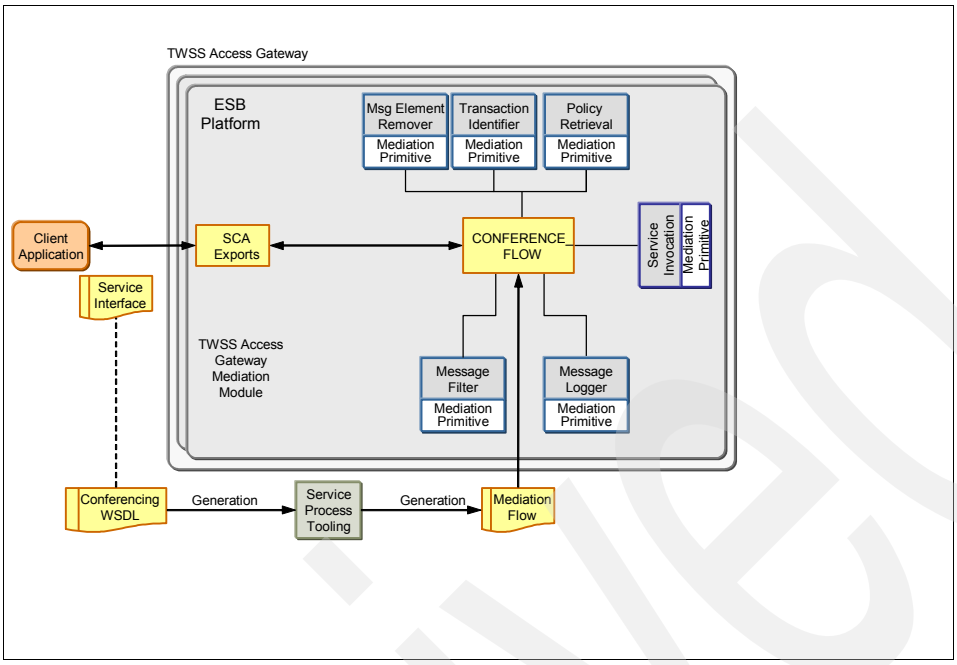


Figure 5-41 A custom mediation flow for a conferencing service

Let us assume the following scenario: Your company has developed an IMS-based conferencing service. The service exposes a simple Web service interface to start immediate conference calls through callback calls to all participants.

Product management has decided to offer access to this service to third parties. The plan is to expose the current interface using the already existing IBM WebSphere Telecommunications Web Services Server infrastructure.

Development has build a service implementation and has provided you with the WSDL file of the service. As a next step, it needs to build a custom mediation flow based on the following requirements:

- No authorization is required for the incoming requests.
- The requests shall be logged based on policy settings.
- The service implementation endpoint shall be defined by a policy.

The following request and response flow shown in Figure 5-42 on page 177 has been designed and needs to be implemented.

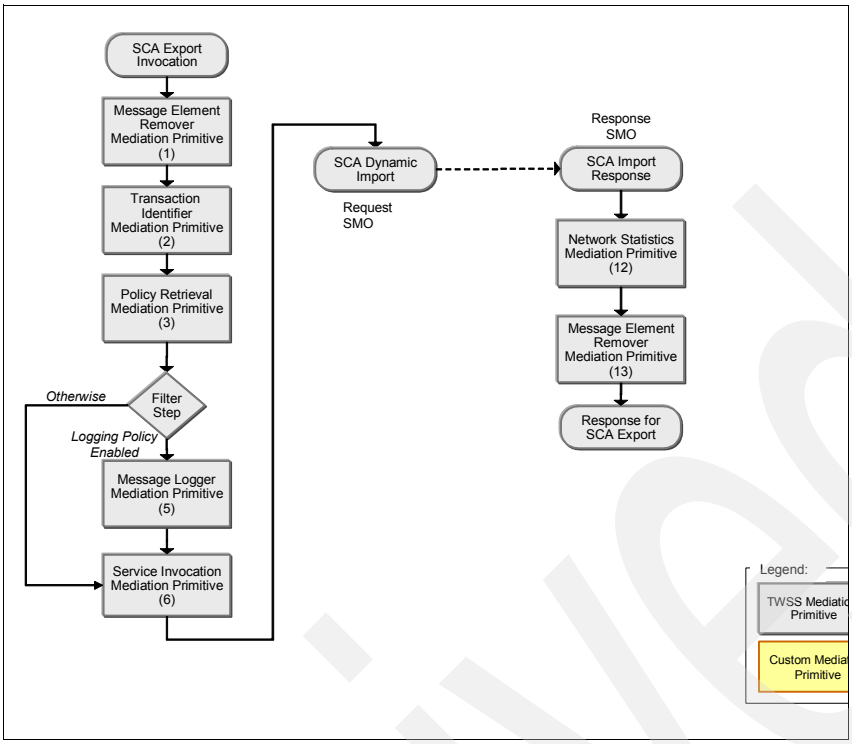


Figure 5-42 Mediation flow for the custom mediation module

5.4.3 Create a new mediation module

Use the WebSphere Integration Developer Tooling Environment to create a new mediation module.

Follow these steps to create a mediation module:

1. In the WebSphere Integration Developer menu, select **File** → **New** → **Mediation Module**. If Mediation Module is not available in the New menu, select **File** → **New** → **Other...** to select Mediation Module from the list, as shown in Figure 5-43 on page 178.

Archived

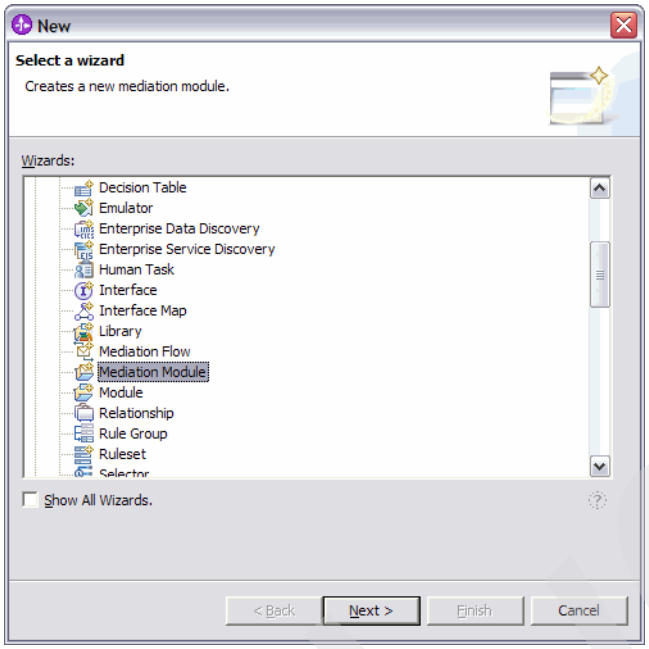


Figure 5-43 Select Mediation Flow Wizard

2. In the New Mediation Module wizard (Figure 5-44):
   - Enter CONFERENCE_FLOW as Module Name.
   - Ensure WebSphere ESB is the chosen target run time.
   - Check **Create Mediation Flow Component**.

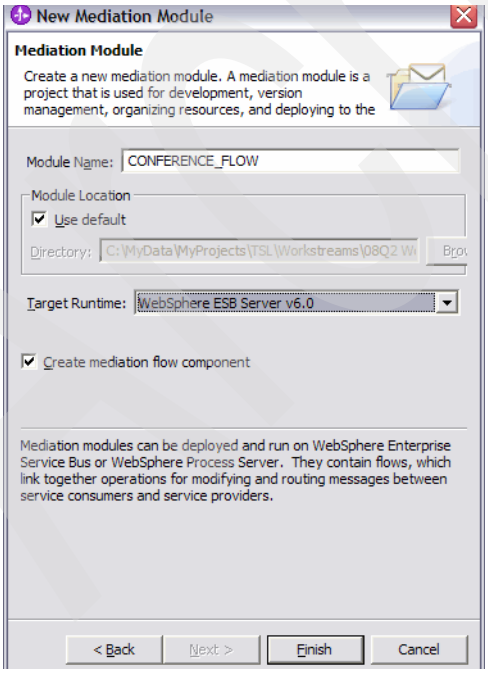


Figure 5-44 New Mediation Module Wizard

3. Click **Finish**.
4. Change to the Business Integration perspective.

5.4.4 Import WSDLs

Import the WSDL files that you are going to be using. In our example, this will be the WSDL that defines the interface to our multi-party conference call service.

To import the interface definitions of the target service, follow these steps:

1. In the Business Integration perspective, right-click **Interfaces** and click **Import...**. This will open the Select Import window (Figure 5-45).

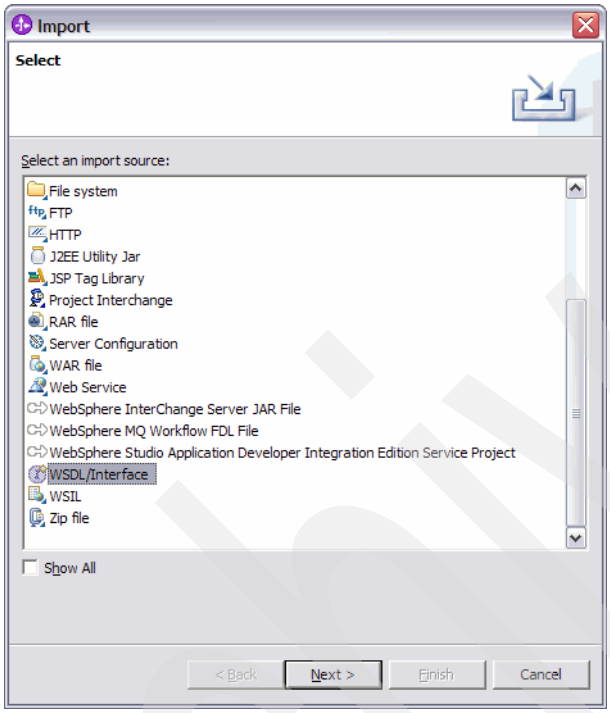


Figure 5-45 Select Import

Archived

2. Select **WSDL/Interface** and click **Next**. This will display the WSDL/Interface Import window (Figure 5-46).

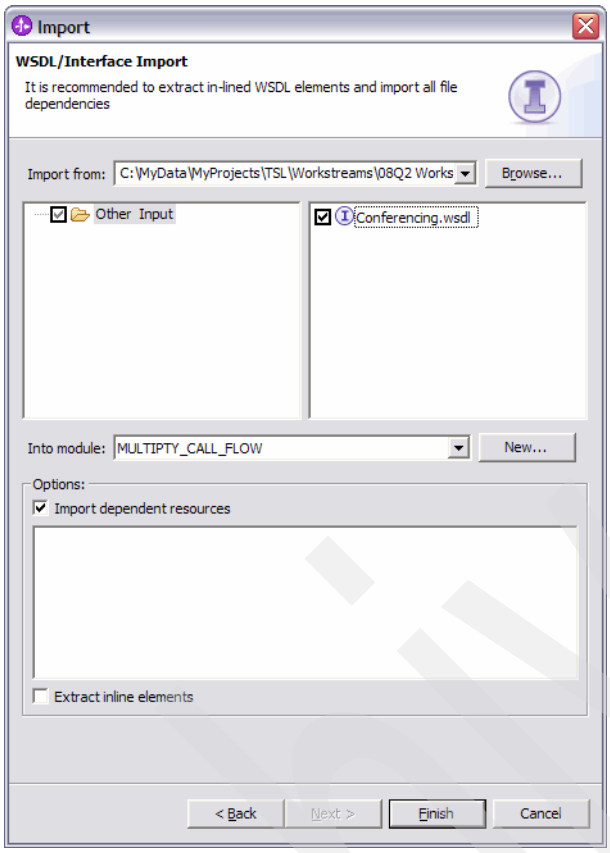


Figure 5-46 WSDL/Interface Import

3. Click **Browse...** and browse to the directory containing your WSDL file.
4. On the left side, expand the directory and select your WSDL file on the right side.
5. Click **Finish**.

5.4.5 Create the ExceptionType business object

We must manually create the business object exceptionType that the IBM WebSphere Telecommunications Web Services Server mediation primitives use in a flow. It is included in the Service Message Object XML structure that flows among mediators. exceptionType is used in a fault flow to return exceptions that occur in mediation primitives back to the caller. All exceptions must be wrapped with Web Services Description Language (WSDL)-defined faults. The exceptionType object has a single string attribute that is assigned the value service or policy by a mediator when an exception occurs. These exceptions may not be defined in your chosen WSDL; however, the object still must exist when using IBM WebSphere Telecommunications Web Services Server mediation primitives.

In the Business Integration perspective, you will see your new mediation module in the left-most panel and under the main project node, a few sub-nodes.

1. Right-click **Data Types**. Select **New** → **Business Object**. This starts the New Business Object wizard (Figure 5-47 on page 181).

Archived

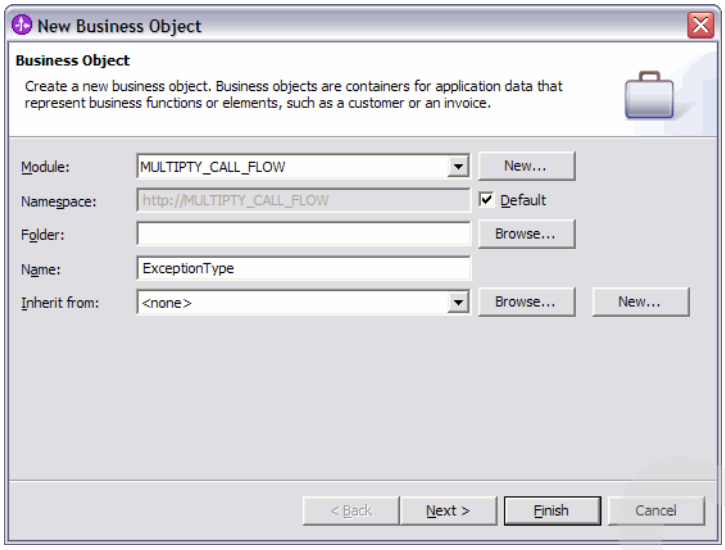


Figure 5-47 New Business Object wizard

2. Enter ExceptionType as the Name.
3. Click **Finish**.
4. After the wizard has created the new business object (it should appear in the business integration view), the business object editor opens. Select the business object **ExceptionType** and click the **Add attribute** icon on the action bar of the editor.
5. Change the attribute name to exceptionType.

You should see a business object that contains one attribute (Figure 5-48).

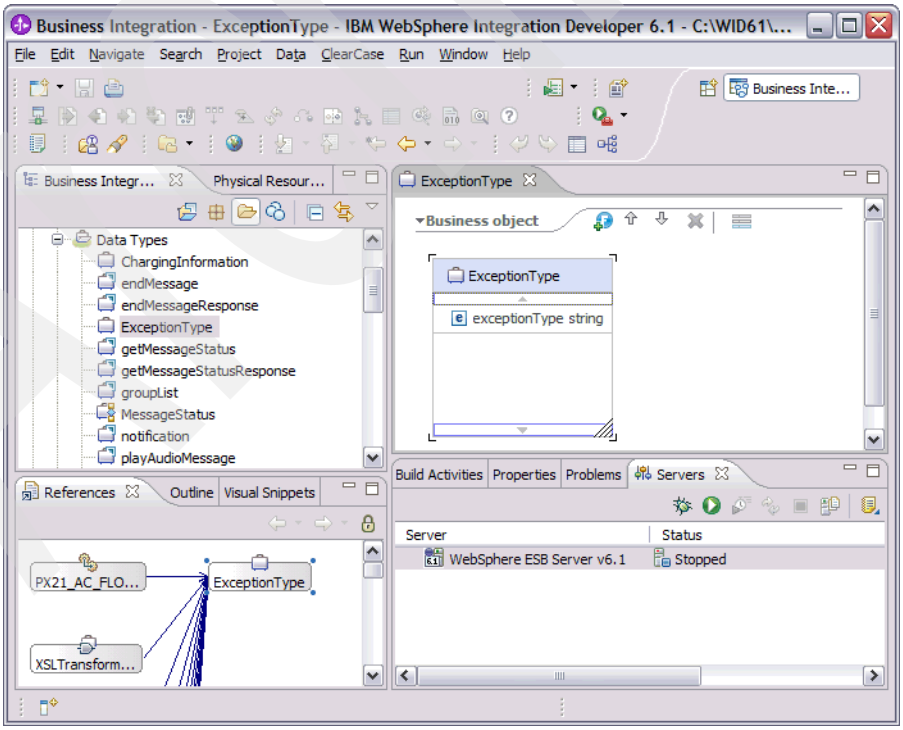


Figure 5-48 ExceptionType business object

6. Select **File** → **Save** to save your workspace.

5.4.6 Import IBM WebSphere Telecommunications Web Services Server SOAP header data types

The mediation primitives all access or modify SOAP headers in the Service Message Object (SMO) during a call flow. The IBM WebSphere Telecommunications Web Services Server mandatory mediation primitives enrich the SOAP header to convey additional information (for example, the policies) to the downstream primitives. The IBM WebSphere Telecommunications Web Services Server SOAP Header Data Types must be imported into the tool to be available to the primitives.

To import the IBM WebSphere Telecommunications Web Services Server SOAP Header data types, follow these steps:

1. In the Business Integration perspective, right-click **Data Types**.
2. Click **Import...**.
3. Select **WSDL/Interface** and click **Next.** This opens the Import window (Figure 5-49).

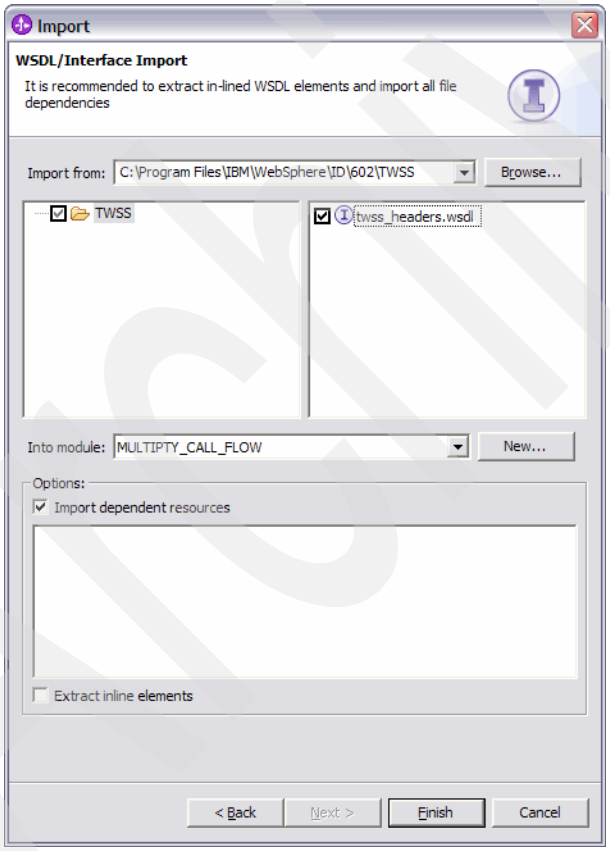


Figure 5-49 WSDL/Interface Import

4. Click **Browse...** and navigate to the ${WID_HOME}\TWSS directory where the file twss_headers.wsdl resides and select it. It is delivered in the IBM WebSphere Telecommunications Web Services Server WebSphere Integration Developer Plug-ins installer or the latest updates. Put a check in the box next to twss_headers.wsdl.

Archived

Note: You may need to extract one of the default flow archives to get the twss_headers.wsdl file.

5. Click **Finish**.

You will see the IBM WebSphere Telecommunications Web Services Server data types defined in that WSDL under the Data Types node in your project (Figure 5-50).

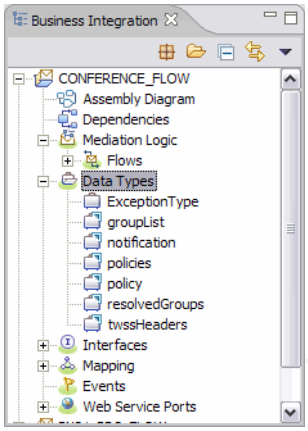


Figure 5-50 Imported data types

5.4.7 Create the assembly

We will use the assembly diagram to specify the SCA imports and exports and to connect these to the mediation flow. The assembly editor is the component of WebSphere Integration Developer where individual components are customized and wired to each other.

Archived

To connect the components of the Mediation Module, perform the following steps:

1. Double-click the Assembly Diagram sub-node under the top project node. The Assembly Editor opens with a default mediation flow component, as show in Figure 5-51.

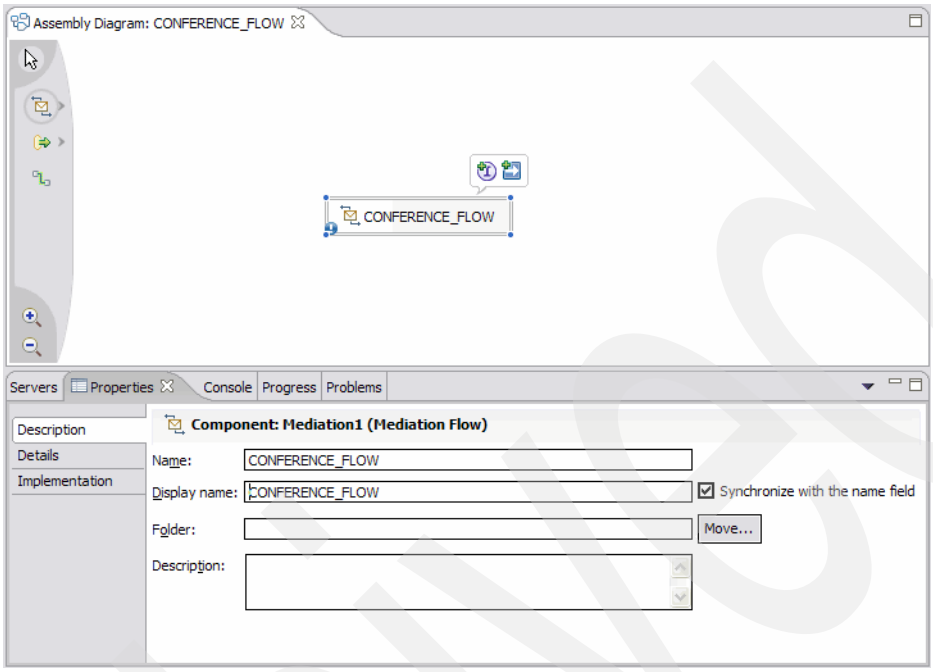


Figure 5-51 Assembly Diagram

2. Perform the following steps for each Web service interface you have imported:

   a. The mediation flow component needs to provide an interface to connect the SCA export to. The SCA export represents the interface that is exposed by the flow. Right-click the flow component and select **Add** → **Interface** to add an interface. This will open the Add Interface selection window (Figure 5-52 on page 185).

Archived

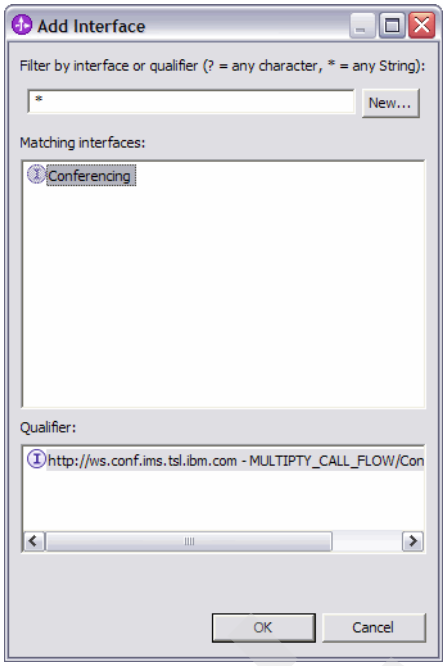


Figure 5-52 Add Interface

b. In the Add Interface window, select the interface you want to add from the list of imported interfaces. In our case, this is the Conferencing interface. Click **OK** to add the interface to the mediation flow. The flow component will have an interface added.

Archived

c. The mediation flow component needs also to provide a reference to connect the SCA import to. The SCA import represents the interface that is exposed by the service implementation. Right-click the flow component again and select **Add** → **Reference** to add a reference. This will open the Add Reference window (Figure 5-53).

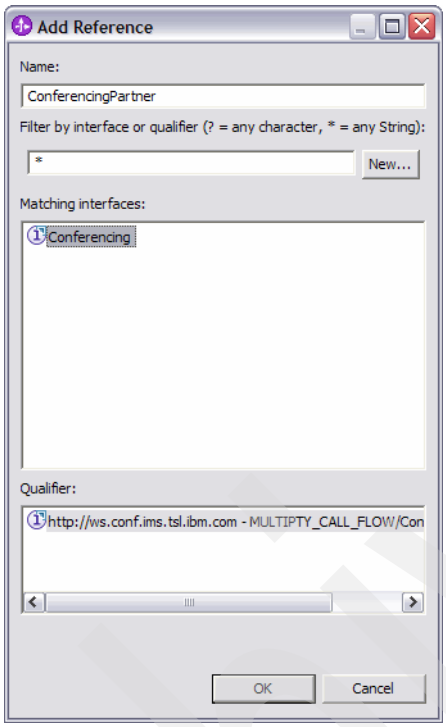


Figure 5-53 Add Reference

d. In the Add Reference window, select the interface you want to add from the list of imported interfaces. In our case, this is the Conferencing interface. Click **OK** to add the reference to the mediation flow. The flow component will have an output terminal for that interface type. The mediation flow component now should look like Figure 5-54.

Figure 5-54 Mediation flow with interface and reference

e. From the list of interfaces in the Business Integration view, drag and drop the interface used in step a on page 184 (the conferencing interface) to the Assembly Diagram editor. The Component Creation window is displayed (Figure 5-55 on page 187).

Archived

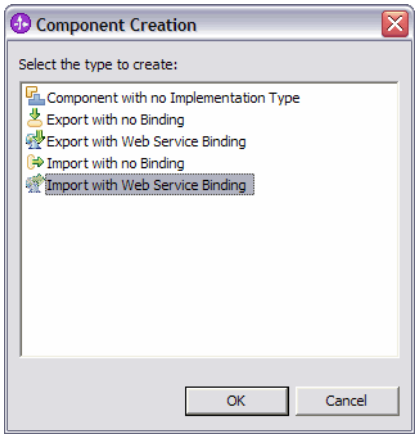


Figure 5-55 Component Creation

f. In the Component Creation wizard, select **Import with Web service Binding** and click **OK**.

g. In Web service Import Details window (Figure 5-56), leave **Use an existing web service port** selected and click **Browse...** to open the Service Port Selection window (Figure 5-57 on page 188).

Figure 5-56 Web service Import Details

Archived

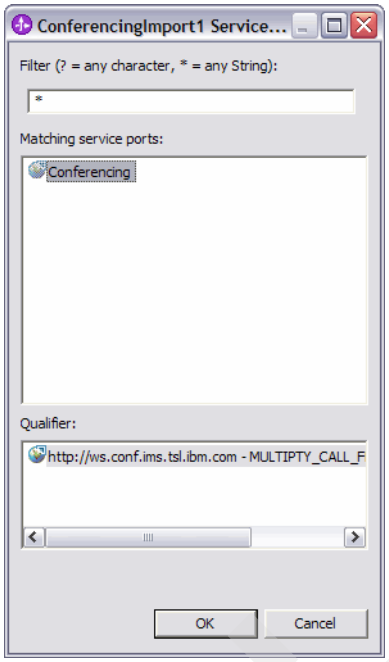


Figure 5-57 Service Port Selection

h. Select the conferencing WSDL, which defines the Web service interface and click **OK**. This creates the ConferencingImport1 SCA import component.

i. Click the Output terminal 1..1 of the mediation flow, drag the wire to the Input terminal of the Import you just created, and drop it there. Your Assembly Diagram should now look similar to Figure 5-58.

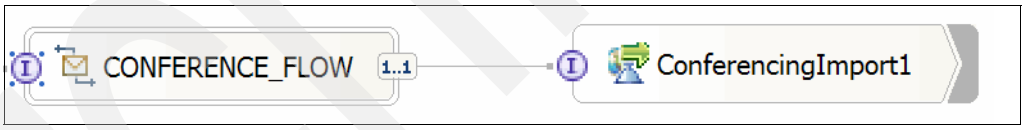


Figure 5-58 Mediation flow connected to SCA import component

j. Right-click the mediation flow component and select **Generate Export... → Web service Binding**. The Transport Selection window opens (Figure 5-59).

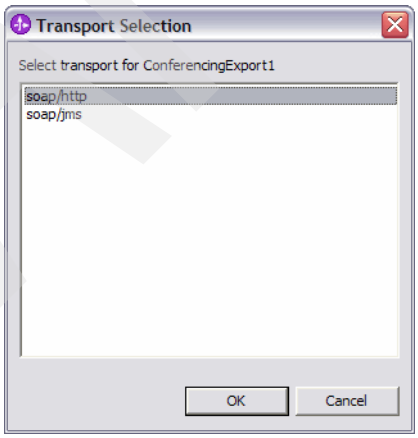


Figure 5-59 Transport Selection

Archived

k. Select the **soap/http** transport. The tooling auto-generates a binding file for the export, which is represented by the SCA export component in the Assembly Diagram, which now looks like Figure 5-60.

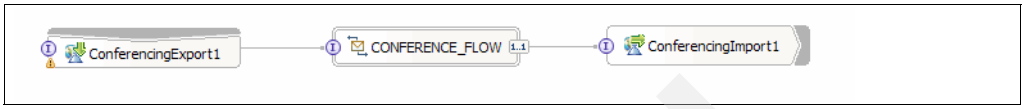


Figure 5-60 Final assembly diagram

l. Right-click the mediation flow component again, and click **Generate Implementation**. This opens the Generate Implementation window (Figure 5-61).

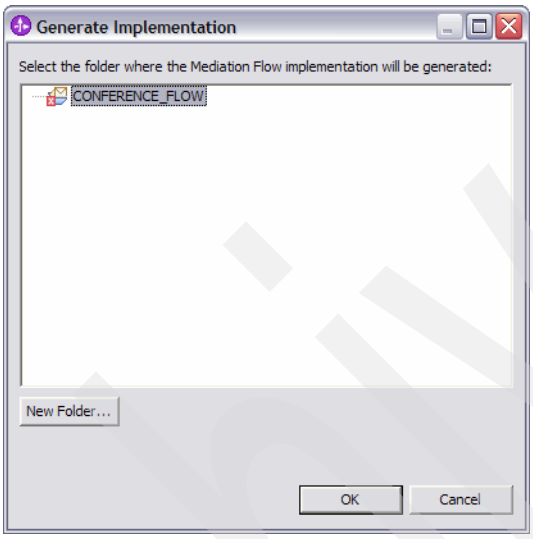


Figure 5-61 Generate Implementation

Archived

m. Select the current project folder, or create a new one if desired, and click **OK**. This opens the mediation flow editor, where you will create your flow (Figure 5-62).

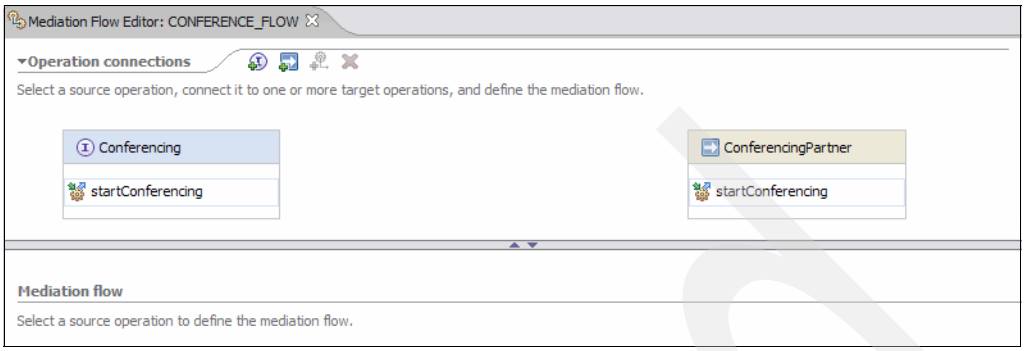


Figure 5-62 Mediation Flow Editor

5.4.8 Construct the mediation flow

In the mediation flow editor, there are two panes. The top pane, Operations connections, shows the interfaces on the left and references on the right. Each operation on the left must be connected to its respective reference on the right. To do this, hover over the operation on the interface you want to connect, click the connector (Figure 5-63), and drag it onto the operation of the reference to connect both together.

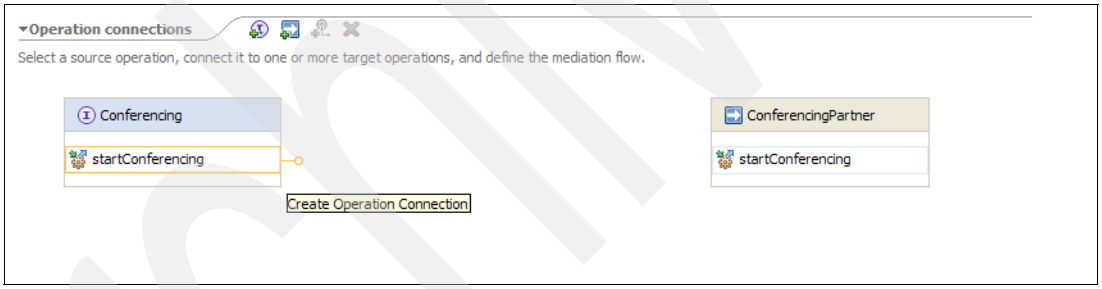


Figure 5-63 Connect export with import operations

Once the connection is in place and an operation is selected, the flow for that operation can be built in the bottom panel.

The bottom panel is activated when an operation is selected in the top panel.

There are several nodes present in the flow by default, as shown in Figure 5-64 on page 191.

Archive

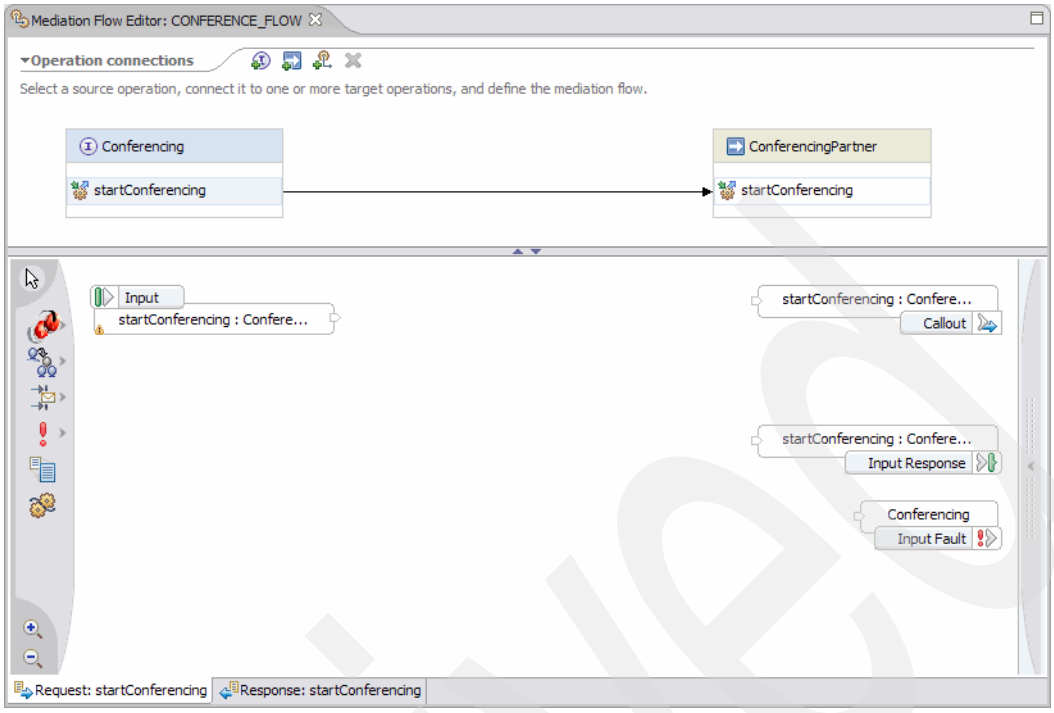


Figure 5-64 Default Mediation Flow nodes

Where:

- Input node: This is where the message flow originates.
- Callout node: This is the final node in the request flow; it connects to the back-end Web service.
- Input Response node: Sends a response message directly back to the caller without invoking the back-end service.
- Input Fault Nodes: If there are Services Description Language (WSDL) defined faults, there will be fault nodes as well.

The bottom panel contains the mediation primitive palette on the left side. The palette provides access to groups of primitives or atomic primitives:

- ►Group of IBM WebSphere Telecommunications Web Services Server mediation primitives
- ►Group of ESB mediation primitives
- ►Group of ESB exception handling primitives
- ►XSLT transformation primitive
- ►Custom mediation primitive
- ►Your custom primitive added as plug-in

The group of IBM WebSphere Telecommunications Web Services Server mediation primitives comprises 11 atomic primitives:

1. Group Resolution
2. JMX Notification
3. Message Element Remover
4. Network Statistics

5. Policy Retrieval
6. Service Authorization
7. Service Invocation
8. SLA Cluster Enforcement
9. SLA Local Enforcement
10. Transaction Identifier
11. Transaction Recorder

The group of ESB primitives covers these functions:

- Message Filter
- Database Lookup
- Message Logger
- Event Emitter
- Message Element Setter
- Endpoint Lookup

The third group provides two primitives to handle failure situations:

- Fail
- Stop

Primitives can be added to the flow by dragging/dropping the primitive icons onto the editor.

Build the request flow

To build your custom mediation request flow, you need to execute the following steps:

1. Register the IBM WebSphere Telecommunications Web Services Server business object with the flow to include the object in the Service Message object.
   a. Click on the Input node and view its properties in the Details tab.
   b. Click **Browse** to add the ExceptionType object as the Transient Context. This will open the Data Type Selection window (Figure 5-65 on page 193).

Archived

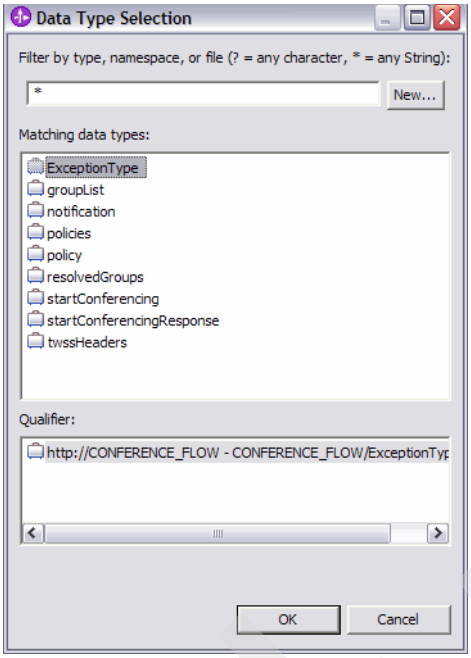


Figure 5-65 Data Type Selection window

c. In the Data Type Selection window, select the **ExceptionType** and click **OK**. This adds the ExceptionType to the transient context of the input message.

To build the request flow, add the primitives you have defined in your design (see 5.4.1, “Design guidelines for new mediation flows” on page 173) to your flow.

2. The Message Element Remover primitive.

a. Select the **Message Element Remover** primitive in the palette and drag it onto the flow editor.

b. Click the Output terminal of the Input Node and drag the wire to the Input terminal on the Message Element Remover primitive. Your mediation flow should look like the one in Figure 5-66. WebSphere Integration Developer automatically assigns the message type to the input and output terminals.

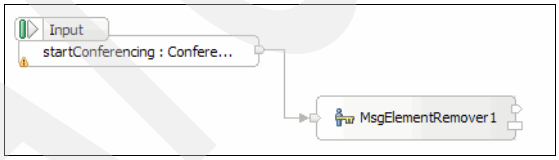


Figure 5-66 Wire mediation primitives

Note: WebSphere Integration Developer automatically assigns the message type to the input and output terminals. You can verify the assigned message type. Select the primitive and then select the **Terminal** tab in the Properties view. Expand the input or output terminal categories and click the terminal (the default is in or out). The message type is displayed on the right. Click **Change...** in case you need to modify the message type.

Archived

Note: The Message Element Remover primitive by default contains a XPath expression to remove any IBM WebSphere Telecommunications Web Services Server Headers from the SOAP header (/headers/SOAPHeader[name=twssHeaders]). Click the **Details** tab in the Properties view to verify the XPath. Click **Custom XPath...** to open the XPath Editor in case you need to change the default expression.

3. The Transaction Identifier primitive.
   a. Select the **Transaction Identifier** primitive in the palette and drag it onto the flow editor.
   b. Click the Output terminal of the Message Element Remover primitive and drag the wire to the Input terminal on the Transaction Identifier primitive.
4. The Policy Retrieval primitive.
   a. Select the **Policy Retrieval** primitive in the palette and drag it onto the flow editor.
   b. Click the Output terminal of the Transaction Identifier primitive and drag the wire to the Input terminal on the Policy Retrieval primitive.
5. The Message Filter primitive.
   a. Select the **Message Filter** primitive in the palette and drag it onto the flow editor.
   b. Click the Output terminal of the Policy Retrieval primitive and drag the wire to the Input terminal on the Message Filter primitive.

   The message filter routes the message to the Message Logger primitive based on a policy. To implement this behavior, we first need to add an additional output terminal to the Message Filter.
   c. Select the **Message Filter** primitive and then click the **Terminal** tab in the Properties view.
   d. Right-click **Output terminal** and select **Add Output Terminal**. This opens the New Dynamic Terminal window (Figure 5-67).

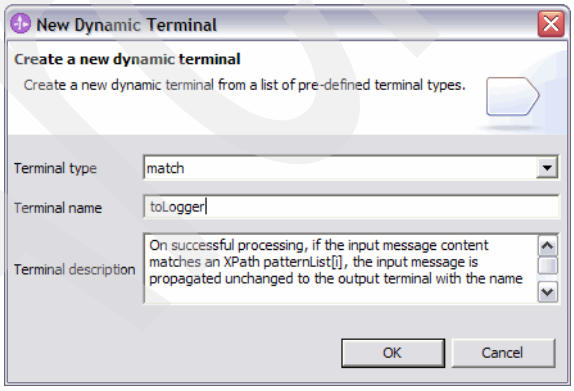


Figure 5-67 New Dynamic Terminal

   e. Enter toLogger as the Terminal Name and click **OK**. This will add a second output terminal to the Message Filter primitive (Figure 5-68 on page 195).

Figure 5-68 Message filter with two output terminals

f. To define the filter logic, click the **Details** tab in the Properties view.

g. Click **Add...**. This opens the Add/Edit Properties window (Figure 5-69).

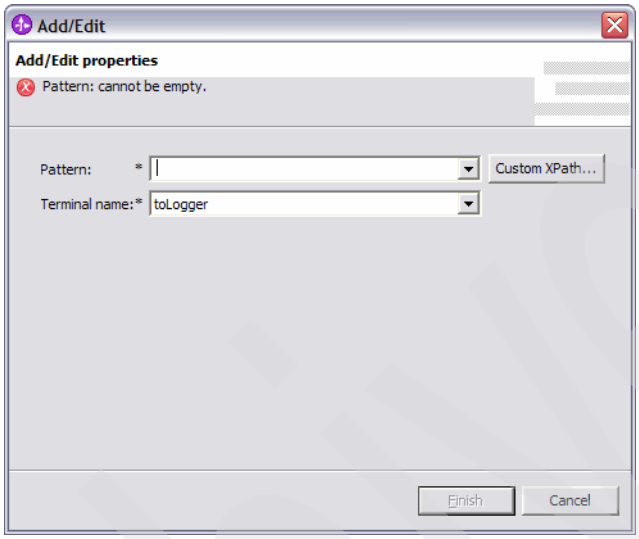


Figure 5-69 Add/Edit properties

Archived

h. In the Add/Edit properties window, select toLogger as the Terminal name. Click **Custom XPath...** to evaluate the policy. This opens the XPath Expression® Builder (Figure 5-70).

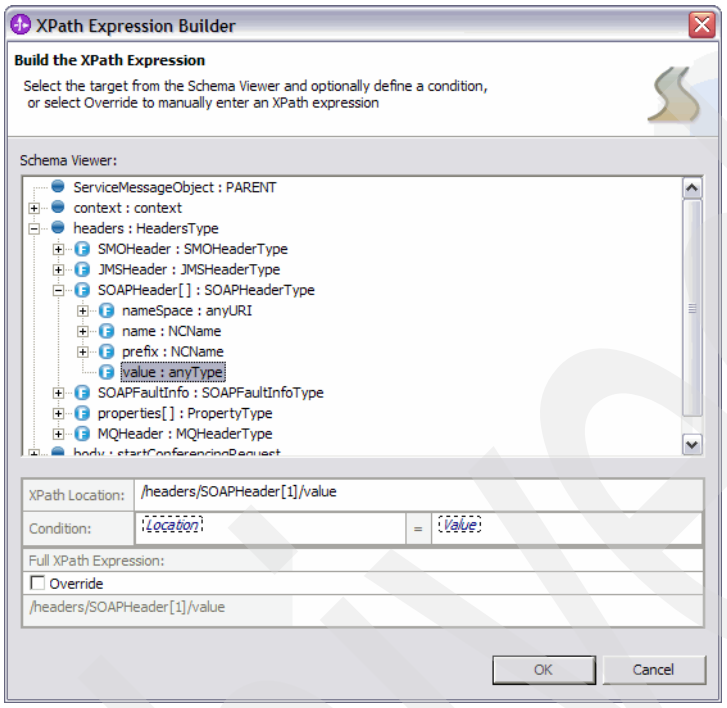


Figure 5-70 XPath Expression Builder

i. In the XPath Expression Builder, check **Override** and enter //headers/SOAPHeader[name="twssHeaders"]/value/policies/policy[@attribute="message.LoggingEnabled"]/@value="true" as the expression. Click **OK**.

j. In the Add/Edit properties window, click **Finish**.

6. The Message Logger primitive

   a. Select the **Message Logger** primitive in the palette and drag it onto the flow editor.

   b. Click the toLogger Output terminal of the Message Filter primitive and drag the wire to the Input terminal on the Message Logger primitive.

   Note: By default, the message body will be logged. You can modify the default and specify which parts of the message shall be logged. Select the **Message Logger** primitive and click the **Details** tab in the Properties view. The Root attribute contains an XPath expression that defines the part of the message that will be logged. The default is /body.

7. The Service Invocation primitive.

   a. Select the **Service Invocation** primitive in the palette and drag it onto the flow editor.

   b. Click the default Output terminal of the Message Filter primitive and drag the wire to the Input terminal on the Service Invocation primitive.

   c. Click the Output terminal of the Message Logger primitive and drag the wire to the Input terminal on the Service Invocation primitive.

8. Click the Output terminal of the Service Invocation primitive and drag the wire to the Input terminal on CallOut node.

Your main request flow is now finished and looks like the one shown in Figure 5-71.

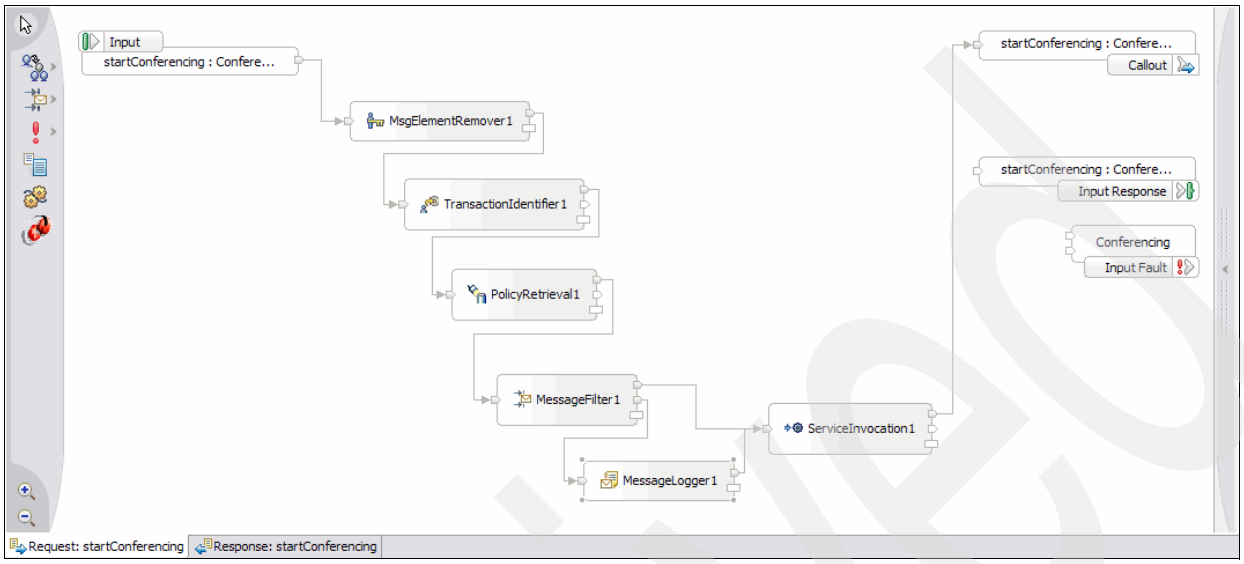


Figure 5-71 Request flow

Build the fault flow

Here we add the flows that handle the faults.

To convey the fault, we need convert the startConference request message type to a startConference_faultMsg type. This is required because the input terminal of the Conference Input Fault node expects a startConference_faultMsg message type. To convert the message type, we insert a XSLT transformation mediation primitive into the flow (see “Fault handling” on page 140 for more information).

Follow these steps to add the transformation primitive:

1. Add and connect the XSL Transformation primitive.
   a. Select the **XSL Transformation** primitive in the palette and drag it onto the flow editor.
   b. Click the Output terminal of the XSL Transformation primitive and drag the wire to the Input terminal on the Input Fault node.

Archive

c. Click the Output terminal of the Service Invocation primitive and drag the wire to the Input terminal on the XSL Transformation primitive. Your flow should look like Figure 5-72.

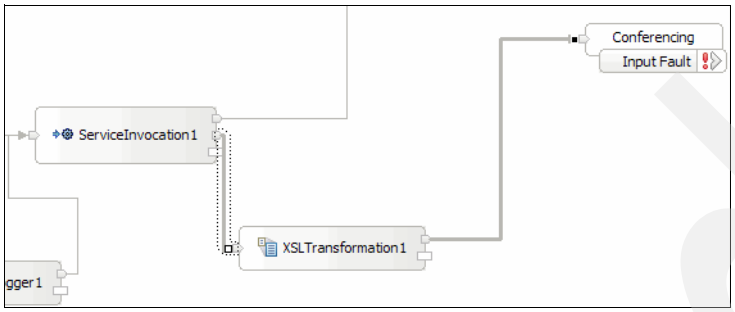


Figure 5-72 XSL transformation wired to the flow

2. Configure the transformation.
   a. Select the **XSL Transformation** primitive. Click the **Details** tab in the Properties view.
   b. Click **New** to open the XSLT Mapping window (Figure 5-73).

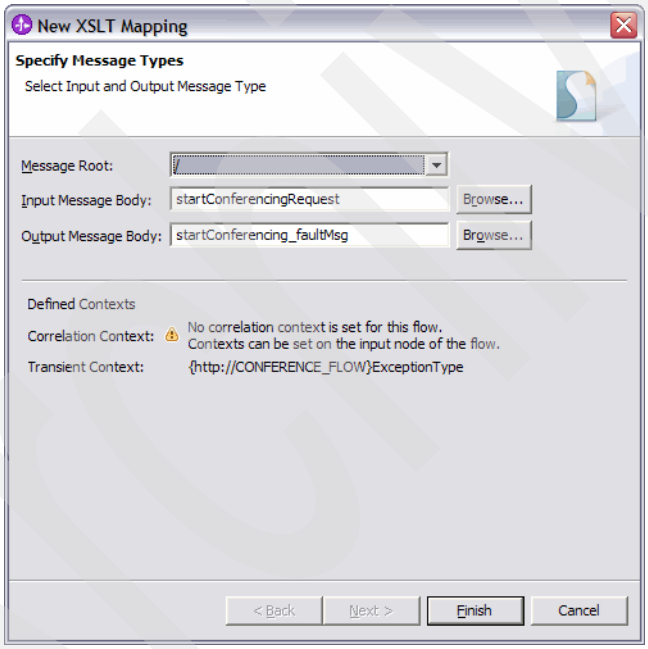


Figure 5-73 New XSLT Mapping

   c. Select **/** as Message Root, **startConferencingRequest** as Input, and **startConferencing_faultMsg** as output.
   d. Click **Finish**. A new XML Map is created (Figure 5-74), assigned to the XSL Transformation, and the XML to XML Mapping editor opens.

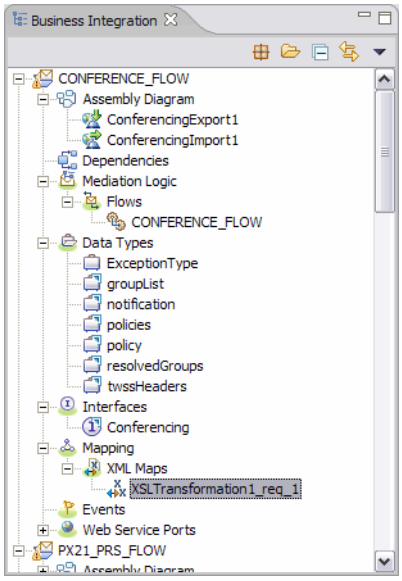


Figure 5-74 XML Map

e. In the source pane of the Mapping editor (Figure 5-75 on page 199) expand **smo** → **ServiceMessageObject** → **context** → **failInfo** → **failureString**.

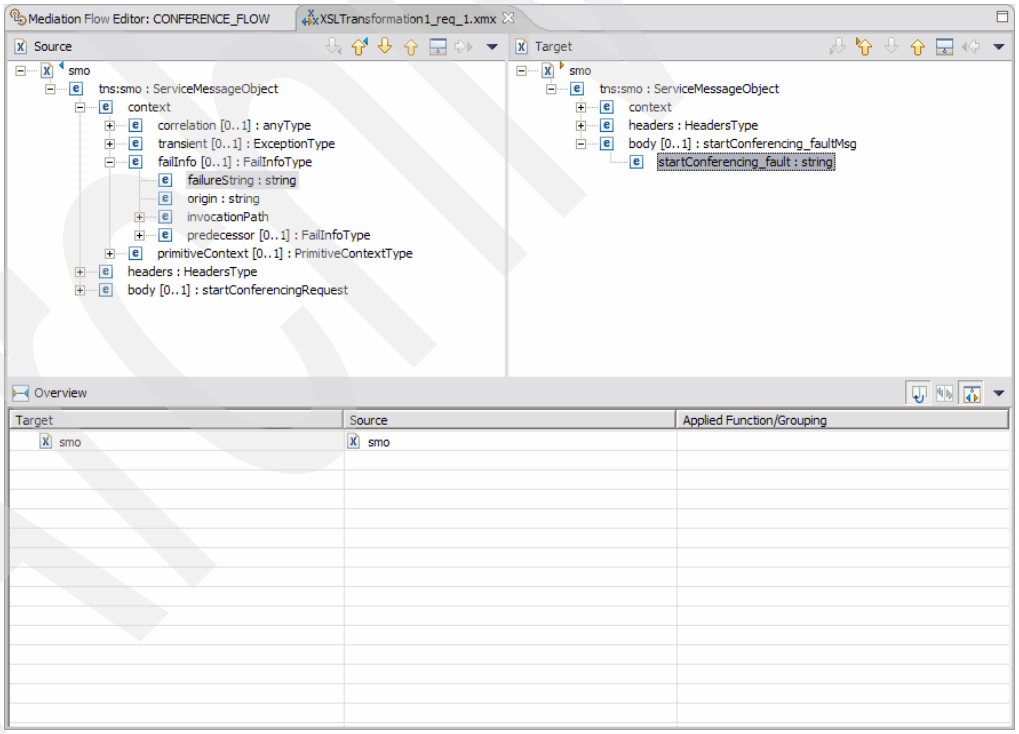


Figure 5-75 IXML to XML Mapping editor

f. In the Target pane, expand **smo** → **ServiceMessageObject** → **body** → **startConferencing_fault**.

g. Right-click **failureString** and click **Create Mapping** to map the fault description from the context to the body.

Archived

h. Because we selected / as root, we need to manually map the remaining fields. Click **smo** in the Source, then click **smo** in the Target. Right-click and select **Match Mapping**. All remaining fields should now be mapped.

3. To finish the fault flows, wire the fault terminals of the Transaction Identifier and the Policy Retrieval primitives to the input terminal of the XSL Transformation.

Build the response flow

To build your custom mediation response flow you need to execute the following steps:

1. Select the **Response** tab at the bottom of the mediation flow editor. The back-end Web service responses invoke the response flow.
2. Click the primitives you want in your flow and drag them to the Output terminal. Click the Output terminal and drag the wire to the Input terminal on another node.

For our example, we add a Network Statistics primitive and an Message Element Remover to both the response flow as well as the fault response flow. The final response flow is shown in Figure 5-76.

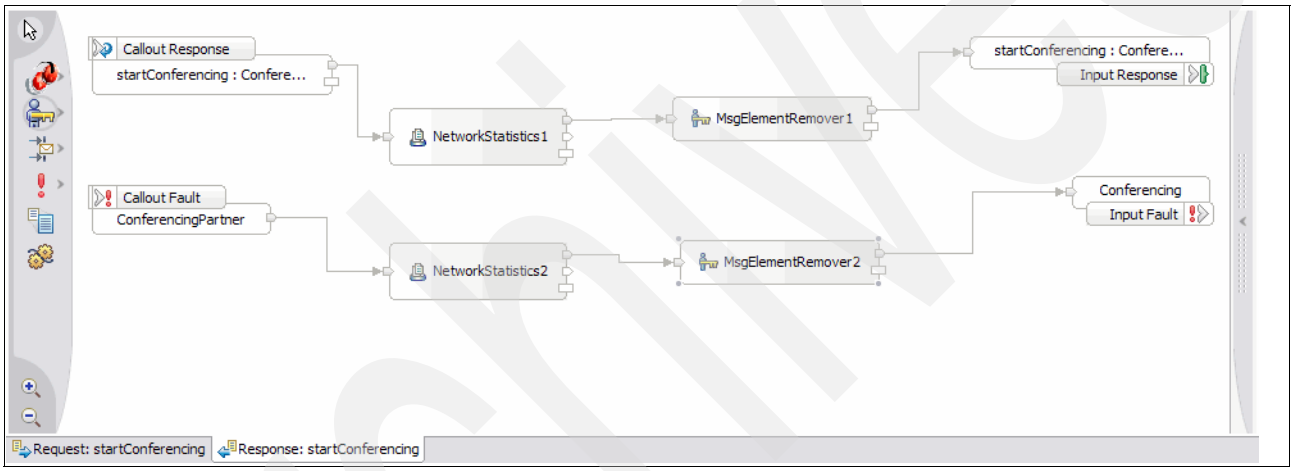


Figure 5-76 Response flow

5.4.9 Assemble the EAR

To create your deployable archive, follow these steps.

1. In WebSphere Integration Developer, select **File** → **Export.** In the Export selection window (Figure 5-77), select EAR and click **Next**.

Archived

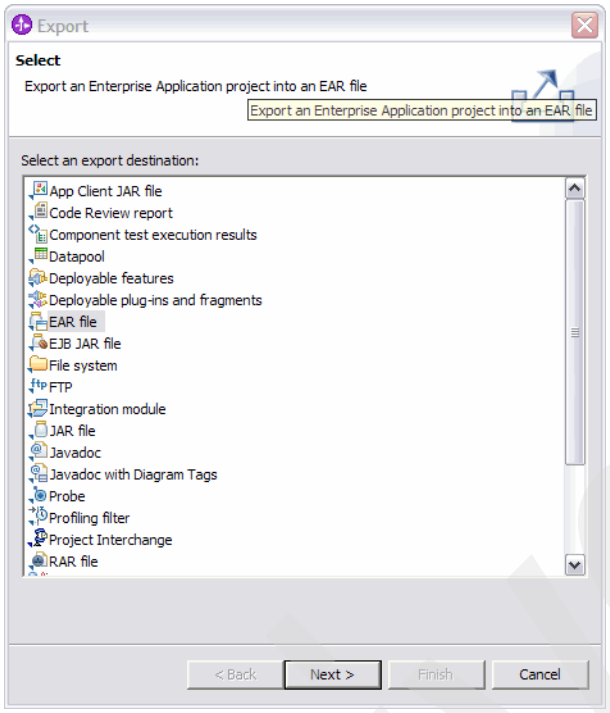


Figure 5-77 Export window

2. In the EAR export window, select the project you want to deploy and the destination directory for the exported EAR file (Figure 5-78 on page 201).

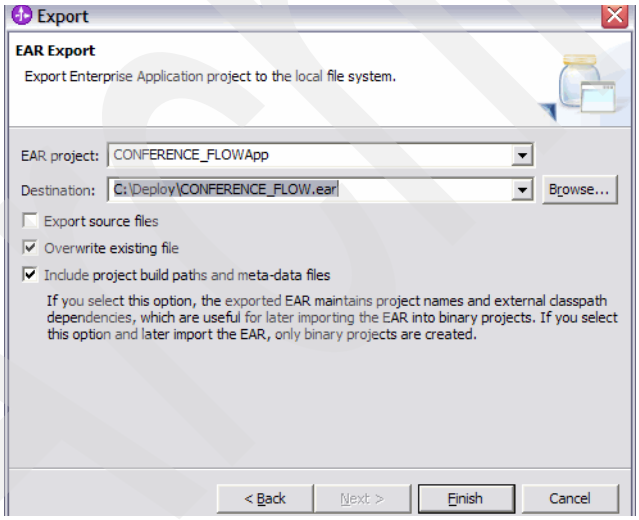


Figure 5-78 EAR export window

5.4.10 Deploy the EAR to the runtime environment

Follow the instructions in 4.6.1, “Deploying the default mediation flow” on page 105 to install the mediation flow EAR file on the server.

Archived

6

Common components

This chapter discusses the common components provided with the Service Platform for IBM WebSphere Telecommunications Web Services Server. The common components are unique to IBM WebSphere Telecommunications Web Services Server in that they provide *value and flexibility* for customizing service implementations through Web service implementation substitution *without requiring modifications to service implementation code*. Common components are shared within the application server by all deployed service implementations.

In this chapter, we describe these common components provided with the product. From a technical perspective, we describe:

- A recommended order in which to call them
- References to the WSDL documentation for each of the common components
- A discussion of the required input parameters and expected outputs for each of the components
- Guidance for how to invoke the common components
- Sample code snippets illustrating how to invoke the common components used in the sample scenario for this IBM Redbooks publication

Archived

© Copyright IBM Corp. 2008. All rights reserved.

6.1 Common components

All access to service platform common components is through Web Services. This provides flexibility for customizing service implementations through Web Services implementation substitution without requiring modifications to service implementation code. Common components are shared within the application server by all deployed service implementations. To maximize the performance of Web service invocations, all invocations to common components will be performed locally, using XML schema types that closely match native Java types. This allows WebSphere Application Server to optimize Web Services invocation and minimize impact to the system.

Figure 6-1 shows the IBM WebSphere Telecommunications Web Services Server Service platform artifacts.

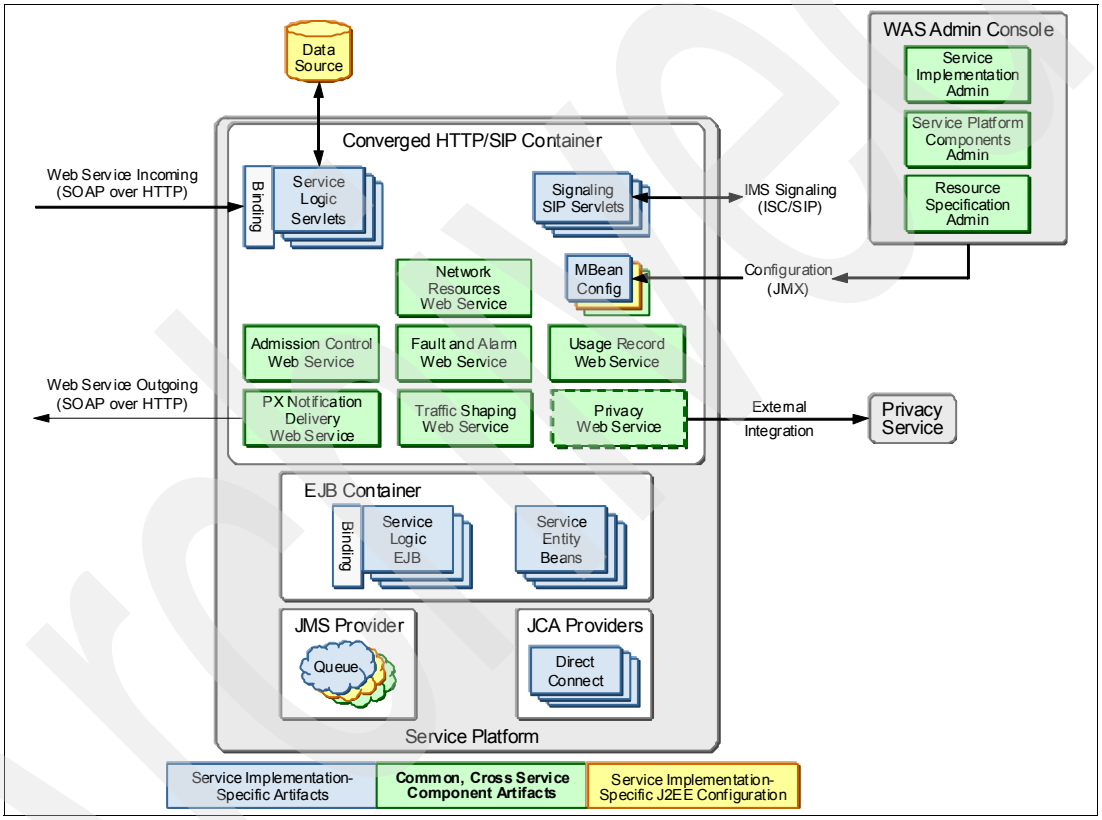


Figure 6-1 IBM WebSphere Telecommunications Web Services Server Service platform artifacts

6.2 Description of common components provided with IBM WebSphere Telecommunications Web Services Server

The following section discusses each of the common components and what their key function is.

6.2.1 Admission Control component Web service

The Admission Control component provides a function to ensure that the number of service implementation Web service requests admitted into the system (server and cluster level) do not exceed a configured rate for a given time interval. Immediately upon receipt of a Web service request, service implementation logic should invoke the Admission Control component to determine whether to accept or reject the Web service request. If the request is accepted, then processing continues as normal. If the request is rejected, then an error message is returned to the requester. This allows service providers to control how much traffic will be accepted by each service and ensure that servers do not exceed planned capacity in the advent of a flood of requests.

Admission Control traffic information is maintained locally in-memory for each application server instance. The request rate is measured in requests per second for each service. A hierarchical rate limiting bucket algorithm will be used to control the rate of admitted requests. Service and operation limits will be configurable through MBeans provided with the Admission Control component.

6.2.2 Traffic Shaping component Web service

Traffic Shaping controls the rate at which traffic initiated by Web Services can be directed towards an element in the network. The intent is to keep service implementations from saturating the network element with work. A service implementation may interact with one or more network elements. Each network element will be associated with a resource specification defined through the Network Resources common component. The network resource specification defines the rate of traffic that can be directed towards the network element. Traffic Shaping will employ a token bucket-based mechanism to allow for limiting traffic burst size and average rate. The implementation of this component takes a non-queuing approach to Traffic Shaping.

The Traffic Shaping component is intended to control traffic being generated by a Web service request outbound towards the network; it does not control the rate of inbound traffic from the network. In addition, the Traffic Shaping component is intended to control the rate of flow of traffic. It does not implement a reservation scheme (for example, around the handling of notifications) against network resource capacity.

Integration of Traffic Shaping into the service implementation will be specific to service requirements, specifically, determining which Network Resources are involved, the point in the processing logic at which Traffic Shaping occurs, and the calculation of the cost of generated traffic. For some services, there will be a direct correlation between the up front cost and the back-end interaction. For others (particularly SIP), there will be some decoupling as certain protocol actions commit the service implementation to completing a set of exchanges. Weight estimates can be used to compensate in such situations.

Traffic measurements for Traffic Shaping must be applied across the cluster. A single active coordinator will be used to maintain in-memory traffic information for shaping traffic across the cluster. Local Traffic Shaping session beans will communicate with the active coordinator through a reservation protocol designed to reduce intra-cluster communication. This requires local Traffic Shaping instances to estimate their traffic rate and reserve traffic for a given resource across the cluster for local traffic. This approach also offers the advantage that multiple service implementations that are deployed within the same cluster and that talk to the same resource can share Traffic Shaping information.

Archived

6.2.3 Network Resources component Web service

The resource name component provides a means of defining specifications (resource attributes) for network resources (elements) that the service platform will interact with. The information in these specifications is used to tailor the behavior of the service platform towards the element. For example, the message processing capacity of the network element can be used by the service platform to control the rate at which traffic is generated towards the element. Resource attributes can also be used for storing additional properties regarding protocol interactions with the element.

This component provides a Web service interface to access resource attributes from other elements. It is used by the Traffic Shaping common component to fetch network resource properties. Resource names are intended to be a flexible set of properties, and thus can be used by other components and services.

6.2.4 Notification Management component Web service

The Notification Management component captures information about activate notifications on the service platform for administrative purposes. Service implementations register and remove notifications as they are created or torn down. The Notification Management component also gathers some rudimentary statistics about notification deliveries and failures. This can provide some useful statistics for an administrator to gain insight into notification delivery in a running system. This component is only intended for capturing information about all notifications on the platform and for capturing statistics around notification delivery attempts; service implementations should still manage storage of notification data in the manner that is most efficient for the service. The Notification Management component includes an administrative interface for querying notification information and terminating active notifications.

Note: The administrative interface is a Web Services interface, and a user interface is not yet provided in the product.

The goal of this component is to provide simple information that can be used by an administrator or care representative to view information about active notifications in the system. The API also allows those individuals to reset outstanding notifications through termination; upon termination, the client-side application would need to re-establish the notification.

6.2.5 PX Notification Delivery Component Web service (Parlay X-specific)

Many Parlay X service implementations require sending outbound Web service notifications to Web service clients as an indicator of network events. A single network event might result in multiple outbound notifications. The PX notification delivery component provides facilities for the delivery of notifications to the destination endpoint through the Access Gateway. This component also provides an asynchronous model for invocation, where an application can send a notification, continue its processing without blocking, and then receive a confirmation of delivery. This component will be Parlay X-specific, as it will contain an interface design intended to deliver Parlay X notifications and its implementation will need to include Parlay X-specific stubs.

The PX notification delivery component decouples the service implementation from the mechanics of the notification delivery. This includes any necessary persistence of notifications and modification of outbound messages to go through the Access Gateway front end.

6.2.6 Faults and Alarms component Web service

An alarm is typically generated when administrative action is needed. When encountering error conditions, service implementations will need to output fault information and potentially emit an alarm for severe error conditions. Fault and alarm information will both be emitted in common base event (CBE) format using CEI and through JMX management notifications. The CBE format is an extensible XML schema for describing event information. The Faults and Alarms component will provide the extensions to the CBE format to describe implementation faults. CEI will be used as the infrastructure for notifying interested parties of events and will provide a persistent event repository that supports flexible event queries based on XPath. The advantage to this approach is that CEI decouples the service platform from interested parties, providing event persistence, and integration with external systems. JMX notifications can be used to allow third parties to hook into WebSphere Application Server and process notifications, such as by generating SNMP alerts. Because there is no listener for CEI from JMX notifications, this component will output CBE events in addition to JMX notifications.

While CEI can be used for the emission of any kind of business event, given the large volume of business events passing through the service platform, the use of CEI will be limited to the alarm and fault event generation encapsulated by this component. This will limit the number of application server instances that must be set aside for processing of CEI events. This limitation is reasonable, since the real-time events that are of most interest to security and monitoring tools are abnormal conditions. The CEI client API comes embedded with the base WebSphere Application Server and can be activated for free by customers that wish to use the CEI infrastructure. The Faults and Alarms component will make use of the CEI emitter client API only.

6.2.7 Usage Records component Web service

Each service implementation should generate a service usage record that describes how the service was used for accounting and billing purposes. Each service usage record will contain common usage record information and service data describing the service delivered. Service data is packed into a single data field for efficient storage. This field must be parsed and expanded during processing of the usage record. The codes identifying those attributes and the format of attribute values will be unique to each service implementation.

6.2.8 Privacy component Web service

The privacy interface provides a means for service implementations to determine whether the requester is allowed to view requested information. It checks whether the requester is allowed to perform an operation on a target (view or update or other operation). IBM WebSphere Telecommunications Web Services Server and all service implementations integrate with the privacy interface, however, there is not a standard privacy implementation that is provided in the IBM WebSphere Telecommunications Web Services Server product, since it generally requires service provider integration with the user terminal registry and security systems. For example, a location-based service should not allow everyone to openly view a subscriber's location. Another example is when a user wishes to block or restrict visibility to his or her information, such as who may view a user's buddy list.

The privacy interface defines an external Web service integration point following the common component model for communication with a Service provider's privacy system. An implementation of the Privacy component is not provided with the service platform. Instead, the client-side portion of the Web service will be included within the service implementations, allowing for invocation of the integration implementation.

The privacy interface essentially acts as a filter to determine whether a request is allowed to be executed against the back-end system. The request is evaluated in the context of the tuple (requester, service, operation, and target). For operations that may execute against multiple targets within a single invocation, the privacy session bean should be consulted for each target.

6.3 WSDL documentation available for the common components

WSDL is available for the Service Platform common components that are described below.

Note: In order to make sure that this information is kept current, we refer to the Information Center for details on the common component APIs.

WSDL documentation for all the common components can be found at this URL:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/wsdl_c.html

Links are provided below for direct links to each of the specific APIs.

6.3.1 Admission Control component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/admctl/index.html?noframes=true

6.3.2 Traffic Shaping component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/traffic/index.html?noframes=true

6.3.3 PX Notification component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter (PX Notification component Web service)

- Link to call direction WSDL documentation: CallDirection:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxcd/overview.html

- Link to call notification WSDL documentation: CallNotification:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxcn/overview.html

- Link to delivery confirmation callback WSDL documentation: DeliveryConfirmationCallbackInterface:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxdcc/overview.html

- Link to message notification WSDL documentation: MessageNotification:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxmm/overview.html

- Link to presence notification WSDL documentation: PresenceNotification:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxprs/overview.html

- Link to SMS notification WSDL documentation: SmsNotification:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxsms/overview.html

- Link to terminal location notification call direction WSDL documentation: TerminalLocationNotification:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxtl/overview.html

- Link to terminal status notification WSDL documentation: TerminalStatusNotification:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/pxnotify/pxts/overview.html

6.3.4 Notification Management component API: Notification Administration component Web service

- Link to notification administration service WSDL documentation: NotificationAdministrationService:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/notify/nmadm/overview.html

- Link to notification administration support WSDL documentation: NotificationAdministrationSupport:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/notify/nmadmsup/overview.html

- Link to notification registration WSDL documentation: NotificationRegistration:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/notify/nmreg/overview.html

- Link to notification statistic publishing WSDL documentation: NotificationStatisticPublishing:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/notify/nmreg/overview.html

Archived

6.3.5 Network Resources component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/netres/index.html?noframes=true

6.3.6 Fault and Alarms component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/fltalm/index.html?noframes=true

6.3.7 Usage Records component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/userec/index.html?noframes=true

6.3.8 Privacy component API

Refer the IBM WebSphere Telecommunications Web Services Server InfoCenter:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.javadoc.doc/privacy/index.html?noframes=true

6.4 Common component Web service client MBeans

The Web service client stubs that communicate with the common components use a common set of MBeans for configuring the invocation settings. These MBeans are reused in each application, appearing in the scope of the application. These configuration properties are:

- AdmissionControlClientMBean
- TrafficShapingClientMBean
- FaultAlarmClientMBean
- NetworkResourceClientMBean
- NotifyManagementClientMBean
- PrivacyClientMBean
- PxNotifyClientMBean
- UsageRecordClientMBean

Table 6-1 shows how to configure the invocation settings.

Archived

Table 6-1 Configuring the invocation settings

Name	Type	String
EndPointURI	String	The endpoint URI used for calling the common component. If set to blank, then calling the common component is disabled.
UserName	String	The user name to use for transport authentication.
Password	String	The password to use for transport authentication.
Timeout	int	Web service invocation timeout.

6.5 Configuration for the common components

The configuration parameters for server-side components are documented in the InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/wsdl_c.html

You can also refer to 6.3, “WSDL documentation available for the common components” on page 208. For specific configuration of the common components used in the sample service implementation used in this IBM Redbooks publication, refer to Chapter 8, “Developing the service implementation” on page 251.

6.6 Invoking IBM WebSphere Telecommunications Web Services Server common components

Telecom Web Services service implementations consist of several reusable common components deployed on top of the WebSphere Application Server platform that provides a common function for implementing Web services for a service provider network.

The common components are intended to be shared by all Parlay X Web service implementations and to facilitate rapid development of new services. Telecom Web Services service implementations are packaged as an individual J2EE EAR and is Web service callable.

Local Web service invocations are used (by virtue of the enableInProcessConnections configuration setting) to access Telecom Web Services service implementations. The components should be deployed on each instance of WebSphere Application Server within a clustered environment.

Parlay X Web service implementations provides functions in the following areas:

- Traffic management: Controls how much traffic is processed and output by the system to prevent overload conditions or excessive use of resources.
- Web service notification support: Facilitates the creation and delivery of Web service notifications to third-party applications.

- Information control: Controls access to sensitive information, that is, information privacy.
- Auditing: Records information about service usage and fault information.
- Notification Management: Enables viewing and management of notifications enabled within the system.
- Fault and Alarms: Alerts® an administrator if a serious error occurs that needs administrative action.

The goal of Telecom Web Services service implementations is to provide supporting function for the service implementations, but to never be limiting. Service implementations should always be constructed in the most efficient manner within the context of the service. The following provides a suggested template and ordering for the invocation of Telecom Web Services service implementations for each service:

1. Call Admission Control: This determines whether there is enough capacity to handle the service request and whether a request should be accepted into the system. This should be the first thing performed by each service.
2. Call privacy: This determines whether the requester has access to execute a service operation for a given target. Parlay X Web service implementations can iterate over all members of the group.
3. Notification Management: If a notification is being started or stopped, this enables the appropriate information to be shared with the Administration system.
4. Call Traffic Shaping: This determines whether the back end can handle the traffic being generated by this request. This call should include a calculation of the traffic being generated by this Web service request.
5. Call usage record: This creates an initial usage record describing the start time of the service.
6. The core logic of the service implementation is executed at this step, performing service processing, with potential calls to the other remaining Telecom Web Services service implementations. Typically, the core logic of the service is dependent on the service operation that is being invoked by the service client application. At this step, the incoming SOAP message is deciphered and an appropriate protocol-specific request is prepared utilizing the request context information that is made available through SOAP headers. The constructed request is then sent by the core logic to the back-end network element and responses are handled. Request processing can vary based on the protocol and the network element behavior. For more information about handling requests, refer to 8.5, “Implementing the core logic of the service” on page 287.
7. Call usage record: This records the stop time of a service once processing comes to an end.

The decision regarding when to output Usage Records is service implementation-specific. Usage Records provide crucial information used for billing; however, too many Usage Records can also greatly reduce the performance of the system. This should be balanced in the design of a custom service.

The ServicePlatform class provides a convenient factory for common component clients that utilize the configuration in the IBM WebSphere Telecommunications Web Services Server Administration Console. These clients also provide asserted identity handling for passing the necessary IMS security credentials. These clients also provide caching for Web service client stubs and local invocations so that the performance impact of making a Web service call is minimized.

Also of note, a convention has been adapted and used within Parlay X Web service implementations for the invocation of Telecom Web Services service implementations. When an endpoint is set to an empty value in IBM WebSphere Telecommunications Web Services Server Administration Console, then the functionality provided by the common component is considered disabled. In such cases, the service implementation logic should omit the function associated with the Telecom Web Services service implementations. Custom Parlay X Web service implementations should follow this convention in order to comply with IBM WebSphere Telecommunications Web Services Server architecture.

If the service implementation is using the PXNotification component to send an asynchronous notification, the service should also implement the DeliveryConfirmationCallback Web service interface so that it can receive confirmations that the notification was delivered.

If the service implementation is using the Notification Management component to manage notifications, the service NotificationAdministrationSupport Web service interface must also be implemented so that it can receive actions from the Notification Manager.

6.6.1 Service Platform package

IBM WebSphere Telecommunications Web Services Server provides the service platform package with the ability to efficiently invoke the common components and leverage the capabilities of IBM WebSphere Telecommunications Web Services Server.

ServicePlatform class

The ServicePlatform class provides public interfaces into the IBM WebSphere Telecommunications Web Services Server Service Platform so that new service implementations can be more easily developed.

ServicePlatformHandler class

The Service Platform Handler class is a handler that should be defined in the webservices.xml for each Web service implementation. Example 6-1 illustrates an example webservices.xml.

Example 6-1 webservice.xml

```
<handler>
   <description>ServicePlatformHandler</description>
   <handler-name>ServicePlatformHandler</handler-name>
   <handler-class>com.ibm.twss.platform.ServicePlatformHandler</handler-class>
</handler>
```

ServicePlatformLogger class

The Service Platform Logger object provides the API for logging and tracing.

Refer to the InfoCenter for APIs of ServicePlatform package for more information and for best practices in using the ServicePlatform package:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/soaruntimepublicjavadoc/com/ibm/twss/platform/package-summary.html

6.6.2 Invoking IBM WebSphere Telecommunications Web Services Server Admission Control common components

The Admission Control component Web service is used to control the admission rate of service operations into individual servers and across clusters. It provides a function to ensure that the number of requests from Web service implementations that are admitted into the system (server and cluster level) do not exceed a configured rate for a given time interval.

Immediately upon receipt of a Web service request, the service implementation logic invokes the Admission Control component Web service to determine whether to accept or reject the Web service request. If the request is accepted, processing continues as normal. If the request is rejected, this component returns an error message to the requester.

This component allows service providers to control how much traffic can be accepted by each service for each member of the cluster and across the cluster. Controlling traffic helps ensure that in the event of a prohibitively large number of requests, servers do not exceed planned capacity.

AdmissionControlInterface

The Admission Control component tracks service usage information and smooths the rate of accepted requests, both at the server and cluster level. Cluster level Admission Control limits should be enforced across the cluster and proportioned by the implementation scheme. When admission capacity is exceeded, either at the server level or the cluster level, then the request will be rejected with a limit exceeded fault.

Weightings for services and operations are calculated according the internal scheme of the implementation. Any weightings must be taken into account when interpreting the assigned rate limits.

- Operation name: verifyAdmittable()
- Description: Verifies whether to admit a service operation.
- Input: verifyAdmittableRequest

  Verifies admittable operation input parameters. Specifies the service operation for verification of admittance.
- Output: verifyAdmittableResponse

  Verifies admittable operation response result. This response is an empty message and indicates success. If a limit has been exceeded, a fault will be returned instead.
- Fault
  - ServiceAdmissionControlFault

    Service Admission Control fault. This indicates that the rate limit for the service has been exceeded. This fault applies to exceeding both local and cluster limits.
  - OperationAdmissionControlFault

    Operation Admission Control fault. This indicates that the rate limit for the operation has been exceeded. This fault applies to exceeding both local and cluster limits.
  - NoAdmissionControlDefinedFault

    No Admission Control defined fault. This indicates that no Admission Control limits have been defined for the supplied service and operation. This fault may be returned for missing local or cluster limits.

Example 6-2 shows a code snippet for Admission Control.

Example 6-2 Code snippet for Admission Control

```
try {
//  Create the variables
         ServicePlatform svcPlatform = new ServicePlatform();
         AdmissionControlInterface adm = svcPlatform.getAdmissionControlClient();
         verifyAdmittableRequest AdmReq = new VerifyAdmittableRequest();
//  Initialize the input parameters
         AdmReq.setGlobalTransactionID(svcPlatform.getGlobalTransactionID());
         AdmReq.setService(getService());
         AdmReq.setOperation(getOperation());
//  Call the verifyAdmittable() operation
         verifyAdmittableResponse AdmRes = adm.verifyAdmittable(AdmReq);
      } catch (Exception e) {
         // Log the exception
         // Rethrow the exception
      }
```

6.6.3 Invoking IBM WebSphere Telecommunications Web Services Server Traffic Shaping common components

The Traffic Shaping component helps control the flow of traffic towards network elements. Resource specifications that describe network element capacity are used as inputs to determine how much traffic the network element can handle. Traffic Shaping controls two kinds of flows of traffic towards the element: burst and rate traffic. The Traffic Shaping algorithm will use a token-bucket algorithm to shape traffic. The token-bucket algorithm works as follows: a bucket containing tokens (abstract units of work) with a fixed capacity is used to represent the capacity of the network element. When the bucket is below maximum capacity, it regenerates tokens at the rate of allowable traffic. This model allows for burst traffic by having burst traffic deplete existing tokens within the bucket; a full bucket allows for the maximum burst size. Once the capacity for burst traffic has been depleted, then traffic becomes constrained by the rate at which the tokens are regenerated. Traffic Shaping is always tracked across the cluster. This is because resource capacity definitions describe the capacity for all traffic directed towards the resource.

Each operation will have a weighting associated that represents how many tokens must be consumed by the bucket for the given operation. This mechanism provides an interface that indicates whether or not the request can currently be admitted towards the network element given the current state of the bucket. If the request is rejected, then the service implementation has the option of either queuing the request or rejecting it with an appropriate service level exception. Any queuing decision must take into account that the state of the bucket only reflects the local view of the cluster member; it is unknown what other allocations are given to other cluster members. The decision whether or not to queue a request depends on the quality of service criteria for the particular service implementation.

Requests for capacity can take into account service/operation weightings, as well as the number of targets for the operation. This information should be calculated by the service implementation, since it bests understands the context in which to calculate the weighting. The number of targets for the operation can be retrieved from the operationTargets SOAP header passed from the front end. This is preferable to counting operation targets.

TrafficShapingInterface

The Traffic Shaping common component helps control the flow of traffic towards network elements. Resource specifications that describe network element capacity are used as input by the implementation to determine how much traffic the network element can handle. Traffic Shaping controls two kinds of flow of traffic towards the element: burst and rate traffic. The burst size indicates the maximum number of messages (in abstract tokens) that can be handled by the network resource in an instantaneously short period. The rate indicates the rate of sustained traffic that can be handled by the network resource over long periods.

Each operation will have a weighting associated that represents how many tokens must be consumed against the resource for the given operation. The number of tokens associated with a given operation is determined in a service implementation specific way, and should take into account the number of targets against which the operation is being executed.

```
TrafficShapingInterface_verifyResourceCapacityResponse   verifyResourceCapacity
(verifyResourceCapacityRequest parameters) [faults:  TrafficExceededFault
NoSuchResourceFault  InvalidResourceSpecificationDefinedFault  ]
```

This verifies whether a network resource has sufficient processing capacity to accept the traffic generated by the specified service operation.

- Input: TrafficShapingInterface_verifyResourceCapacityRequest

  Verifies network resource capacity input parameters. Specifies the network resource logical name against which the operation is being executed.

- Output: TrafficShapingInterface_verifyResourceCapacityResponse

  Verifies the network resource capacity response result. It indicates that the network resource has sufficient capacity to accept the additional traffic.

- Fault: TrafficExceededFault

  Traffic exceeded fault. This indicates that the specified service operation request would generate traffic that exceeds the network resource specification's current available capacity.

- NoSuchResourceFault

  No such resource fault. This indicates that the specified resource is not known.

- InvalidResourceSpecificationDefinedFault

  Invalid resource specification defined fault. This indicates that the network resource specification for the logical resource name is invalid or is missing properties required by this component.

Example 6-3 shows a code snippet for Traffic Shaping.

Archive

Example 6-3 Code snippet for Traffic Shaping

```
try {
VerifyResourceCapacityRequest  req = new VerifyResourceCapacityRequest();
// The service associated with the operation.
req.setService(ServicePlatform.getCurrentApplicationName());
// The operation being invoked. The operation name corresponds to a WSDL operation
for the service.
   req.setOperation(req.getOperation());
// The global transaction ID for the Web service transaction.
   req.setGlobalTransactionID(ServicePlatform.getGlobalTransactionID());
// The network resource logical name.
   req.setResource("PresenceServer");
   // The requested capacity for this operation.
   req.setRequestCapacity(0);.VerifyResourceCapacityResponse capacityResponse =
trafficShaping.verifyResourceCapacity(req);
}
```

Administering the Traffic Shaping component Web service

The following section discusses how to configure the Traffic Shaping component Web service.

Configuration

Network Resources Traffic Shaping limits are configured within the IBM WebSphere Telecommunications Web Services Server Administration Console under the Network Resources section. The Network Resources component Web service must be installed in order for this function to be enabled. A network resource has a logical name that is associated with a resource specification, or a set of properties. Multiple service implementations may be configured to refer to the same network resource logical name. The Traffic Shaping component Web service expects the following network resource specification properties to be defined:

- Maximum Burst Size: The maximum amount of burst traffic that can be handled by the network resource. This is measured in number of tokens, where a single service implementation Traffic Shaping request may consume multiple tokens.
- Maximum Average Sustained Rate: The maximum average sustained rate of work, measured in tokens per second that can be handled by the resource. This corresponds to the rate of token regeneration in the token bucket algorithm.

Guidelines for choosing limits

The maximum burst size must be chosen such that when burst size is split evenly among all members of the cluster, each member's local token bucket has sufficient tokens to accommodate the largest request weight. Otherwise, there will never be sufficient tokens to satisfy such a request and this request will always be rejected. In addition, configuring some additional burst above the maximum request size will provide better adaptability of the algorithm to different traffic distributions of traffic being generated within the cluster. An example is a burst configured with a maximum burst of 10 requests, when the maximum request weighting is 1.

When running at capacity, the algorithm has a few limitations for request distributions that approach the traffic rates near the specified limit for the network resource. Reservation requests for rates are made on an as-needed basis when processing incoming requests. Each request may result in a reservation request for enough rate to replenish the tokens consumed by that request. Clusters are typically fronted by a round-robin load-balancer and thus may generate outbound traffic in a similar fashion. When running at capacity, it is possible that an individual node will get a chunk of rate allocated during a round robin spray

and the other remaining members be denied. This may not match expectations across the cluster. For example, consider a cluster with three members, with a rate limit for an operation of 30 tokens per second across the cluster and each cluster member allocated nine tokens per second. This leaves three tokens per second worth of rate to reserve. If an additional spray of requests comes on top of the 27 tokens per second rate across the cluster, then the first request in the spray will result in the first server getting allocated the three tokens per second and the remainder being denied additional capacity. This may result in less token regeneration across the cluster than expected. This behavior typically manifests itself as a ping-pong effect, where the last little bit of tokens get traded off between members of the cluster. This can be avoided by running a rate that is less than: (number of cluster members * maximum token per request size). The algorithm will tolerate running within this threshold, but some premature rejections may occur in rare traffic patterns.

6.6.4 Invoking IBM WebSphere Telecommunications Web Services Server Fault and Alarm common components

Exceptions, faults, and alarms can be generated by either the underlying Telecom Web Services service implementations or by a particular service implementation.

Exceptions are runtime Java exceptions that are raised in the course of service execution. An exception does not necessarily correspond to a fault (it might lead to a change in execution flow that results in successful execution).

Faults correspond to error conditions that arise during service execution. Faults typically correspond to application exceptions within the service implementation code structure. Faults are standard error conditions during execution flow and alarms are severe, infrequent events that interfere with service delivery.

Alarms correspond to severe error conditions that are unexpected execution flow conditions that the administrator should be informed about.

An event should be considered either a benign exception, a fault, or an alarm, but not a combination of these options.

The following procedures will be used for handling service implementation exceptions:

- The exception will be logged in an informational or warning message to the Parlay X Web service implementations log. The service implementation can choose to supply either contents of the exception (such as the stack trace) or simply an informational message describing the exception situation.
- The service request will recover and continue execution.

The following procedures will be used for handling faults:

- The exception or error condition that caused the fault should be logged as an error message to the Parlay X Web service implementations log. The log message should contain stack trace information.
- The service implementation should call the recordFault operation on the Fault and Alarm component. This will emit a common base event (CBE) through the common event infrastructure (CEI) and emit a JMX notification through the Fault and Alarm component MBean.

Archive

FaultAlarmInterface

The Fault and Alarm component performs appropriate logging and notification generation for service implementation Faults and Alarms. Services are identified through their service parameter, which should be uniquely named to avoid conflicts. Implementations of this component may implement alarm suppression, where similar alarms are suppressed to avoid emitting the same alarms repeatedly in an alarm condition. Configuration of such behavior is specific to the implementation of this component.

- Operation Name: recordFault ().
- Description: Records a fault.
- Input: recordFaultRequest.

  Records fault operation input parameters. Specifies the fault information to record.
- Output: recordFaultResponse.

  Records fault response result.

Example 6-4 shows a code snippet for the Fault and Alarm component.

Example 6-4 Code snippet for fault and alarm

```
Try {
   RecordFaultRequest faultReq = new RecordFaultRequest();
   faultReq.setGlobalTransactionID(globalTransId);
   faultReq.setService(serviceName);
   faultReq.setSource(sourceClass.getName());
   faultReq.setCode(messageId);
   faultReq.setDetail("Fault Occured. Presence Server not responding");
   faultReq.setMessage(msg);
   faultReq.setSeverity(SeverityEnumeration.Info);
   RecordFaultResponse faultRes = faultAlarmClient.recordFault(faultReq);
      } catch (Exception e) {
         // Log the exception
         // Rethrow the exception
      }
```

The service implementation should create a fault response, which corresponds to a Web service application-specific fault message whose fault element contains the exception that caused the fault. This fault response is returned back to the Web service requester.

The following procedures will be used for handling alarms:

- The exception or error condition that caused the fault should be logged as an error message to the Parlay X Web service implementations log. The log message should contain stack trace information.
- The service implementation should call the recordAlarm operation on the Fault and Alarm component only once for the duration of the alarm condition. This will emit a common base event (CBE) through the common event infrastructure (CEI) and emit a JMX notification.

Administering the Fault And Alarm component Web service

The Fault and Alarm component Web service can be deployed to capture Faults and Alarms that occur for each deployed Web service implementation, for the Admission Control component Web service, and for the usage record component Web service. It can write fault and alarm information to the common event infrastructure (CEI) repository.

Archived

If you plan to use the CEI repository to capture fault and alarm data, you must enable the CEI service and configure a data source for it.

The CEI service uses the database included with WebSphere Application Server, but it can be configured to use DB2, Oracle®, or another database supported by WebSphere Application Server.

When encountering error conditions, service implementations will need to emit fault information. For severe error conditions, they will need to emit alarms. Both Faults and Alarms are emitted in common base event (CBE) format using JMX management notifications, and, optionally, using CEI as well. The CBE format is an extensible XML schema for describing event information.

The Fault and Alarm component Web service provides extensions to the CBE format to describe implementation faults. CEI is used as the infrastructure for notifying interested parties of events and provides a persistent event repository that supports flexible event queries based on XPath. JMX notifications are useful for enabling third parties to access WebSphere Application Server and handle notifications, for example, by generating SNMP alerts. However, because JMX notifications do not include a listener for CEI, the Fault and Alarm component Web service produces CBE events in addition to JMX notifications.

Although CEI can be used to emit any kind of business event, CEI is limited in this implementation to generating fault and alarm events. This is done so that you do not need to set aside a large number of application server instances to process CEI events. The CEI client API comes embedded with the base WebSphere Application Server product and can be activated at no charge.

Configuration

Settings for Faults and Alarms are configured within the IBM WebSphere Telecommunications Web Services Server Administration Console. The Fault and Alarm component Web service expects the following value to be defined:

- Enable CEI: A Boolean value indicating whether CEI events are emitted when processing a fault or alarm. If set to false (the default), only JMX notifications are emitted. For details, refer to the topic “Enabling the Fault and Alarm component Web service to use CEI” at the IBM WebSphere Telecommunications Web Services Server InfoCenter at:

  `http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.config.doc/enable_cei_t.html`

6.6.5 Invoking IBM WebSphere Telecommunications Web Services Server Network Resource common components

The Network Resources component Web service provides a way to define specifications for network resources (elements) with which the service platform will interact. Specifications that you design using the Network Resources component Web service are used to tailor the way in which the service platform behaves toward each resource.

Note: Normally, Network Resources is not called directly by a service implementation unless it has a special need to do so. It is generally called by Traffic Shaping internally.

For example, the Service Platform can use a network element's message processing capacity to control the rate at which traffic is generated towards the element. Other specifications can be used for storing additional properties regarding protocol interactions with the element.

This component provides a Web service interface to access resource specification properties from other elements. It is used by the Traffic Shaping component Web service to fetch network resource properties. Resource specifications are intended to be a flexible set of properties, and thus they can be used by other components and services.

NetworkResourceInterface

The Network Resources component allows other common components and service implementations to access network resource names. Network resource names are a collection of properties and values associated with a network resource logical name. These properties should be definable within the implementation of the Network Resources common component. This interface makes no assumptions about the properties within the resource specification, allowing for extensibility of network resource properties.

- Operation Name: retrieveResourceProperties().
- Description: Retrieves the network resource properties response.
- Input: RetrieveResourcePropertiesRequest.

  Retrieves all network resource properties requests.
- Output: ResourceProperty[]

  Retrieve network resource properties response message. The response contains the supplied list of network resource properties and their values. The retrieveResourcePropertiesResponse element in XML instance can be substituted with other elements.
- Fault:
  - InvalidResourcePropertyFault:

    Invalid resource specification property fault. This indicates that a network resource specification property for the logical resource name does not exist.
  - NetworkResourceRetrievalFault:

    Retrieves a network resource property fault. This indicates that the supplied property name does not exist within the resource specification.
  - NoSuchResourceFault:

    No such resource fault. This indicates that the specified resource is not known.
- Operation Name: retrieveAllResourceProperties().
- Description: Retrieves all network resource properties responses.
- Input: RetrieveAllResourcePropertiesRequest.

  Retrieve all network resource properties request.
- Output: ResourceProperty[].
- Fault:
  - NetworkResourceRetrievalFault
  - NoSuchResourceFault
  - InvalidResourcePropertyFault

Archived

Example 6-5 shows a code snippet for the Networks Resource common component.

Example 6-5 Code snippet for Network Resource

```
Try {
//  Create the variables
        ServicePlatform svcPlatform = new ServicePlatform();
        NetworkResourceInterface net = svcPlatform.getNetworkResourceClient();
        RetrieveResourcePropertiesRequest netReq = new VerifyAdmittableRequest();
//  Initialize the input parameters
        netReq.setResourceName(getResourceName());
        netReq.setResourcePropertyNames(getResourcePropertyNames());
//  Call the retrieveResourceProperties() operation
        ResourceProperty[] Res = net.retrieveResourceProperties(netReq);
     } catch (Exception e) {
        // Log the exception
        // Rethrow the exception
     }
```

6.6.6 Invoking IBM WebSphere Telecommunications Web Services Server Usage Records common components

The IBM WebSphere Telecommunications Web Services Server Usage Record common component is a write-only component that records billing information.

Service Usage Records describe how a service was used for accounting and billing purposes. When integrating with a service provider environment, you should create a billing mediator application to extract Usage Records, process them to generate call detail records, and purge processed entries from the database. Usage Records are written to a table definition, which is a primary Usage Records table that contains the master record and service attributes (See also Appendix C, “Developing a Usage Record Billing Mediator common component” on page 385.)

Each IBM WebSphere Telecommunications Web Services Server Web service implementation generates a service usage record that describes how the service was used for accounting and billing purposes. Service Usage Records are stored in relational table format. Each service usage record contains common event data that can be used to uniquely identify the service record, and that references a properties table containing application-specific attributes. This provides a uniform infrastructure for creating and storing service Usage Records.

Service Usage Records consist of general service usage information that may be generated at multiple points during service execution. An event type field is used to differentiate the different recording points. Each service implementation defines the event types (generation points), status codes, and service data attributes that are generated for storage in the service usage table. The Parlay X Call Notification over SIP/IMS creates a service record by calling the Usage Record component Web service.

- Usage Records for the Parlay X Call Notification over SIP/IMS

  Parlay X Call Notification over SIP/IMS uses the service usage record to store network events, call party information, and called party information. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/notification_records_c.html

- Usage Records for Parlay X Presence over SIP/IMS

  The Presence service implementation uses the Usage Record component Web service to record Web service invocation details. Because they exist solely to either service or generate Parlay X requests, IMS-based network signaling events do not have their own Usage Records. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/presence_records_c.html

- Usage Records for Parlay X Terminal Status over SIP/IMS

  Parlay X Terminal Status over SIP/IMS uses the Usage Record component Web service to record Web service invocation details. Because they exist solely to either service or generate Parlay X requests, IMS-based network signaling events do not have their own Usage Records. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/termstat_records_c.html

- Usage Records for the Third-Party Call service implementation

  The Parlay X Third-Party Call over SIP/IMS uses the Usage Record component Web service to record events related to a service request. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/tpc_records_c.html

- Usage Records for Parlay X Payment over Usage Records/CEI

  The Parlay X Payment over Usage Records/CEI Web service implementation uses Usage Records to store accounting and billing information. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/payment_records_c.html

- Usage Records for Parlay X Address List Manager over XCAP

  The Parlay X Address List Manager over XCAP Web service uses the service usage record to record any event generated during create, read, update and delete group definition stored within XDMS. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/addresslist_records_c.html

- Usage Records for WAP Push over SMPP

  The WAP Push over SMPP Web service implementation uses the Usage Record component Web service to record exception details. For the synchronous part of the Web service request, an error condition maps to an exception, which is returned to the caller. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/wappush_records_c.html

- Usage Records for Parlay X SMS over SMPP

  Parlay X SMS over SMPP uses the Usage Record component Web service to record events related to a service request. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.services.doc/sms_smpp_records_c.html

Archive

- Usage Records for Parlay X Terminal Location over MLP

  The Parlay X Terminal Location over MLP uses the Usage Record component Web service to record events related to a service request. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.services.doc/termlocationmlp_records_c.html

- Usage Records for Parlay X Multimedia Messaging over MM7

  The Parlay X Multimedia Messaging over MM7 Web service implementation uses the Usage Record component Web service to get, send, and start notification messages. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.services.doc/mmsmm7_records_c.html

- Usage Records for Parlay X Terminal Status over Parlay

  Parlay X Terminal Status over Parlay uses the Usage Record component Web service to record Web service invocation details. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.services.doc/termstat_backend_records_c.html

- Usage Records for Parlay X Terminal Location over Parlay

  The Parlay X Terminal Location over Parlay Web service implementation, uses the Usage Record component Web service to record events related to a service request. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.services.doc/termloc_records_c.html

- Usage Records for Parlay X SMS over Parlay

  The Parlay X SMS over Parlay Web service implementation uses the Usage Record component Web service to record events related to a service request. You can find more information about this topic at:

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/topic/com.ibm.twss.services.doc/sms_parlay_records_c.html

UsageRecordInterface

Service Usage Records consist of general service usage information and a variable number of attributes for storing custom information about the service delivered. Service Usage Records may be generated at multiple points during service execution. An event type field differentiates the different record points. Each service implementation should define the event types (generation points), status codes, and service data attributes it intends on generating for storage in the service usage table.

- Operation Name: writeUsageRecord()
- Input: WriteUsageRecordRequest

  Writes usage record operation input parameters. Contains information about the usage record to be written.

- Output: WriteUsageRecordResponse

  Writes usage record operation response result. Contains the record ID of the written usage record. Also indicates that the record was successfully written.

- Fault: WriteUsageRecordFault

  Writes usage record fault. Indicates that an error occurred while writing the usage record.

Example 6-6 shows a code snippet for the Usage Records common component.

Example 6-6 Code Snippet for Usage Records

```
try {
//  Create the variables
        ServicePlatform svcPlatform = new ServicePlatform();
        UsageRecordInterface usageRecord = svcPlatform.getUsageRecordClient();
        WriteUsageRecordRequest usageRequest  = new WriteUsageRecordRequest();
//  Initialize the input parameters
        usageRequest.setService(getServiceName());

usageRequest.setGlobalTransactionID(svcPlatform.getGlobalTransactionID());
        usageRequest.setEventType(getEventType());
        usageRequest.setRecordTime(getRecordTime());
        usageRequest.setCode(getCode().intValue());
        usageRequest.setServiceAttributes(getServiceAttributes());
        usageRequest.setServiceValues(getServiceValues());
//  Call the writeUsageRecord() operation to record the usage record
        WriteUsageRecordResponse usageResponse =
usageRecord.writeUsageRecord(usageRequest);
    } catch (Exception e) {
      // Log the exception
      // Rethrow the exception}
```

Archived

Archived

7

Design considerations for the service implementation

In this chapter, we introduce you to the design aspects of service implementations on the Service Platform component of IBM WebSphere Telecommunications Web Services Server. These design principles may be used as a reference by architects and developers working on custom service implementations.

The purpose of these design guidelines are:

- Serving as a common means of understanding the IBM WebSphere Telecommunications Web Services Server architecture and Service Platform components for the developer community
- Adhering to these guidelines helps with faster and more efficient development of reliable services to be deployed on IBM WebSphere Telecommunications Web Services Server
- Following the packaging and naming guidelines allows multiple service implementations to coexist in the environment
- Conforming to Logging and Tracing guidelines helps debug problems and minimize the time spent on bug fixes
- Following these guidelines assists in adherence to IBM Development standards, which means easier integration and interoperability with out-of-box IBM WebSphere Telecommunications Web Services Server service implementations.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

7.1 Architecture overview

The IBM WebSphere Telecommunications Web Services Server Service Platform enables Telecom Service Providers to expose high-level Web services interfaces for underlying network services. These Web services can be made available to provision rich, value added services. Services such as these can form core infrastructure, which can be leveraged by both third-party service providers or external customers. The Web service interfaces provide technology-agnostic access to service capabilities. Each Web service interface can have multiple implementations (also referred to as instantiations or flavors) in a given service provider environment, providing access to IMS services through SIP, PSTN functionality through a Parlay/OSA gateway, Direct Connect access to network protocols, or custom integrated services. IBM WebSphere Telecommunications Web Services Server provides a middleware infrastructure for managing Web service access and provides an environment for hosting Web service API implementations.

7.2 Telecom Web Services

A Telecom Web Service implementation comprises a Web service implementation exposing standards based abstraction to underlying network services, for example, Parlay X Presence. Service implementation participates in the IBM WebSphere Telecommunications Web Services Server Service Platform environment, allowing a Service Provider to integrate and expose network interfaces and higher level IT interfaces that enable development of rich, value added services.

The subsequent sections discuss the design considerations specific to custom service implementations that could be either Parlay X service implementation or Direct Connect service implementations. IBM WebSphere Telecommunications Web Services Server facilitates deployment of service implementations, as described in the following sections.

7.2.1 Parlay X Web Services

IBM WebSphere Telecommunications Web Services Server provides an execution environment and a common set of components to facilitate Parlay X Web service implementations in addition to Parlay service implementations. The Parlay X Web service APIs are the first instantiation of services. Consequently, parts of the design can include functions that provide specific support for Parlay X Web Services, but the architecture is general enough to provide infrastructure services for any kind of Web service implementation. Figure 7-1 on page 229 provides an overview of the IBM WebSphere Telecommunications Web Services Server architecture.

Archive

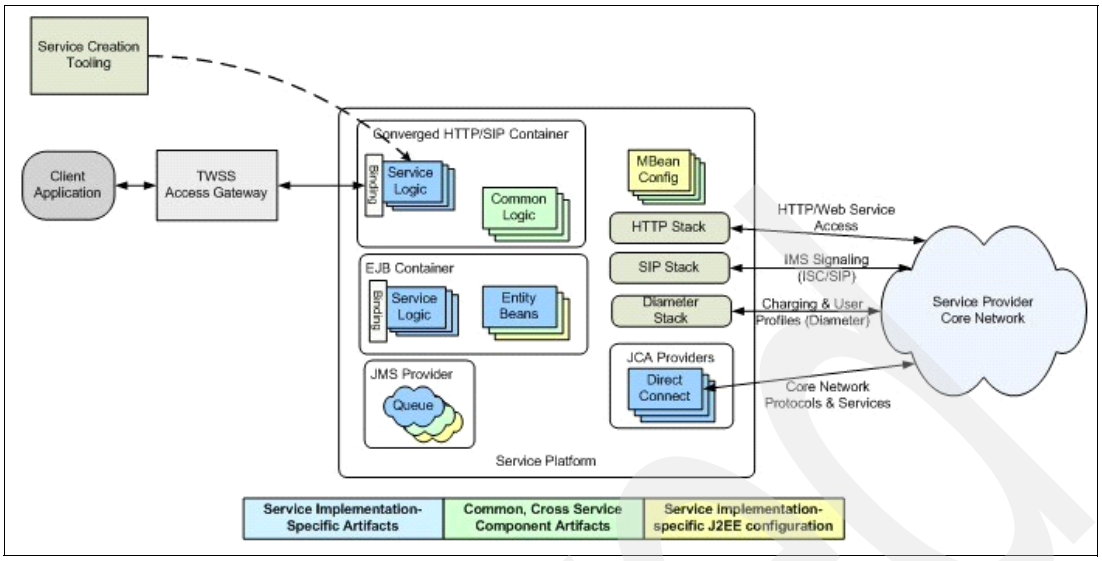


Figure 7-1 IBM WebSphere Telecommunications Web Services Server Service Platform architecture

All Web service requests and responses pass through and are inspected by Telecom Web Services Access Gateway before being passed on for processing. Telecom Web Services Access Gateway acts as an intermediary, consults the policy information from the Subscription Management system for all communication between client and service endpoints, and processes both incoming requests to Parlay X Web service implementations and outgoing requests and responses made by Parlay X Web service implementations. Parlay X Web service implementations provide a bridge between the Web service API and core network, typically initiating one or more network-level activities for each request. For Parlay X Web service implementations capable of issuing outbound requests, either provisioned through a notification service or through other means, these implementations must send outbound requests through Telecom Web Services Access Gateway to be processed in a similar fashion.

Communication between Telecom Web Services Access Gateway and Parlay X Web service implementations uses SOAP messages over HTTP. The HTTP transport should be used when tighter latency response times are required. Document literal is the preferred SOAP message encoding format. It allows maximum interoperability between Web service clients. Document literal encoding should be used as the encoding format when generating all Web service proxy stubs. The architectural design focuses primarily on the use of document literal encoding over HTTP. HTTPS communication is also assumed as an alternative transport to HTTP where HTTP is mentioned within this document, but requires additional deployment time configuration.

IBM WebSphere Telecommunications Web Services Server consists of a WebSphere Application Server environment with support for IMS protocols. The converged HTTP/SIP Servlet container is used for hosting Web and SIP application logic. SIP Servlets can interact with the IMS SIP signaling elements according to the 3GPP ISC interface. The IMS Connector stack provides a client API for accessing diameter services for integration of security, user profile, and accounting functionality. For direct access to network protocols for which a stack in base WebSphere Application Server does not exist, the Java Connector Architecture (JCA) framework provides a means of extending the application server environment with additional protocol stacks. Finally, the Java Messaging Service (JMS) provided with base WebSphere Application Server can be used for asynchronous, latency-insensitive, high throughput communication between application components.

Parlay X Web service implementations are based primarily on the Web container programming model for Web service implementation bindings and common service components. JAX-RPC servlets with plain Java beans are the recommended Web service implementation binding, although Parlay X Web service implementations are free to choose the method that is most efficient for a given implementation, such as Web services adhering to J2EE specification JSR 109.

7.3 Design considerations

In this section, you will find the design guidelines for service implementations. These guidelines are more applicable to Parlay X Web Services implementation than Parlay and Direct Connect implementations.

7.3.1 Conforming to IBM WebSphere Telecommunications Web Services Server conventions

IBM WebSphere Telecommunications Web Services Server software utilizes several significant types of named resources. These resources generally follow a naming convention. The use of naming conventions is an important aspect of design, as it uniquely identifies an object's or service's role in the system. There are other conventions that should be considered, such as configuration conventions for new service, policy naming conventions, trace message codes, and messages.

Service independence

Firstly, service implementations should be designed independently of each other as a single deployable enterprise archive EAR file. In case one or more service implementations are required to share common utility classes, the shared code can be packaged as a OSGi bundle or Java archive and the same can be included in the respective enterprise archives. Refer to the 8.3.2, "Service Platform Application Template" on page 256 for a detailed discussion on service logic development and application template.

Policy naming conventions

Policies are configuration information that are defined within the IBM WebSphere Telecommunications Web Services Server's Service Policy Manager (SPM) run time and console. Policies relate to requesters or user identities, service names, and service operation names of Web services. A policy name is typically scoped for better readability. It is independent from the position that the policy is defined within the Service Policy Manager. Here are a list of policy categories defined in SPM along with respective descriptions:

- message: Category for message processing service policies, typically related to Access Gateway processing.
- message.capacity: Policies for capacity measurements.
- message.intercept: Policies for message intercept.
- message.sla: Policies for SLA measurements.
- message.statistics: Policies for network statistics.
- message.privacy: Policies for privacy.
- requester: Category for requester policies, typically related to requester specific policy processing.
- requester.operation: Policies to apply to requester use of operations.

- requester.trace: Policies related to tracing requests by requester.
- service: Category of service policies, typically related to service specific policy processing.
- service.config: Policies related to service configuration.
- service.custom: Policies defined for service implementations.
- service.standard: Policies defined by Web services specifications.
- customer: Category for values that are customer unique and are not recorded in the Service usage record.

The Policy naming should follow the pattern <prefix>.<Policy_name>, as shown in Example 7-1.

Example 7-1 Sample policy names

```
message.statistics.RecordStatistics
service.config.PresenceServerURI
```

Note: Prefixes should be in lower case. Policy names should start with upper case and use upper case to delineate compound words.

Telecom Web service implementation naming conventions

A service implementation within the IBM WebSphere Telecommunications Web Services Server must have a unique identifier name that distinguishes its type of service, and the back-end network technology that it communicates to. The naming pattern for Parlay X Web service is PX<version>_<webservicetype>_<backendtype>.

A list of Parlay X Web Services types implemented in IBM WebSphere Telecommunications Web Services Server follows:

- TS: Terminal Status
- TL: Terminal Location
- TPC: Third Party Call Control
- CN: Call Notification
- CD: Call Direction
- CH: Call Handling
- PRS: Presence
- ALM: Address List Management
- PMT: Payment
- MMS: Multimedia Messaging Service
- SMS: Short Messaging Service

Note: If necessary, you could extend the naming convention as required. This could be the case if your organization had multiple implementations of the service with the same back-end type. For example, TL could interface with the Presence/SIP and a Location service with a SIP interface to get the presence information.

Archived

A list of Parlay X Web services back-end types that identify the network connectivity or protocol follows:

- IMS: IP Multimedia Subsystem (SIP - Session Initiation Protocol based)
- Parlay: Parlay / OSA based
- SMPP: Short Message Peer to Peer protocol based
- MM7: Multimedia Messaging application interface Version 7 based
- MLP: Mobile Location Protocol

J2EE resources naming conventions

J2EE resources refer to the named objects, such as DataSource Providers, DataSources, and JMS Queues, which are typically created in the WebSphere Application Server Administration Console. These resources are used by service implementations. The naming pattern to be followed for J2EE resource is <service_name>_<resource_type>. The service name indicates the service that accesses the resource. Some resources may be shared by multiple services. In such cases, the name of the resource should be generic. The list of resource types by type of resource follows:

- Provider: Data Source Provider
- Queue: JMS Queue
- Scheduler: WebSphere Application Server Scheduler
- AS: Activation Specification
- Dest: Destination
- Partition: WebSphere Partition
- QCF: Queue Connection Factory
- Alias: A Java Authentication and Authorization Service (JAAS) Alias
- DB: Database

J2EE resources are typically associated with a Java Naming Directory Interface (JNDI) name, which is the used by services and applications to do lookup using the Application Server Java Naming Directory. The JNDI names should follow certain conventions. The prefixes of JNDI names are mentioned below:

- jdbc: DataSource
- ejb: Enterprise Java Bean
- eis: Enterprise Information Service
- sched: Scheduler
- jms: Java Messaging Service

JNDI names are typically of the pattern <jndi_prefix>/<service_name>_<resource_type>.

Database naming conventions

The database instance is named after the system component that uses it. Typical database names used in IBM WebSphere Telecommunications Web Services Server environment are:

- TWSS63
- SPM62
- PARLAY62

Archived

Security role naming conventions

Each service implementation should have one or more roles for its interfaces. Since IBM WebSphere Telecommunications Web Services Server provides user-level authorization, role-based security is of limited value and might only have value in composite service scenarios. So, one or two roles for each service should be sufficient. The naming conventions for role names should follow the pattern <service_type>_<backend_type>_<role_name>. The role name should indicate the authorizing WebSphere Application Server role, for example:

- Administrator
- Accessor

In some cases, the service type or back-end type is so generic that it is necessary to use “TWSS” or similar text that adequately describes the scope of use.

Namespace naming conventions

Namespaces are defined in the WSDL of a Web service interface. Since custom Web services can be defined and implemented in BM WebSphere Telecommunications Web Services Server, it may be necessary to define a unique and context sensitive namespace. The namespaces are also used to define the Service Policy Manager service identifier. Namespaces typically appear like URLs. They should begin with a host name, then the type of namespace, followed by interface name. The interface name may have multiple naming components to indicate things like the administration or notification interfaces. A namespace declaration should end with a version number, followed by the object type. The object types can be for instance, local, service, fault, interface, and so on. Here is a sample namespace for reference:

```
http://www.ibm.com/wsdl/wappush/notification/v1_0/interface
```

Note: The namespace looks like a Web URL. The URL location does not mean a hosted site or location. Namespace is used to provide a scoped name for identifying a functionality.

Web root context naming conventions

The root context is the base part of the URL where Web applications of an enterprise application can be reached. This root context name is defined in the enterprise archive (EAR). These names should follow a naming convention so that they do not conflict with the contexts of other services deployed in IBM WebSphere Telecommunications Web Services Server. The root context definition can be located in the application.xml file of the service enterprise archive. Service implementations should maintain the pattern TWSS/<interface><version>/<service>/<interfaces>. The URL should also have a servlet mapping that defines the URL suffix. It typically defines the type of interface and interface names. For example, for a Web service interface, it should define ‘services’, and then the name of the interface, as shown here:

```
/services/ReserveAmountCharging
```

J2EE application naming conventions

The J2EE application, comprising of the service implementation logic, should be named in a way consistent with the service implementation. However, because the file name of enterprise archive is identified as the enterprise application name, the following pattern should be followed: PX<version>_<webservicetype>_<backendtype>.ear.

Package naming conventions

Package names should define the namespace in which the code artifacts exist. The package names in Java are similar to a reverse of a fully qualified host name; however, a package is independent of the host name, for example:

```
com.ibm.twss.<component>.<sub component>
```

Usage Record naming conventions

Service implementations are expected to generate usage records using the common component, the Usage Records, for each operation that is invoked, and for notification events. When creating a Usage Record, there are a set of property names that are defined. During the design phase, it is important to determine what information needs to be generated into a Usage Record. This is generally the information from the request context as well as the parameters of the request. The attributes to be recorded should be named in a way consistent as those defined for other services, and to the extent possible, use the same property names for the same data items. Usage Records, or alternatively referred to as Service Records, are typically named using upper case words with underscores to separate words.

Message identifiers

Service implementations should plan on utilizing the error message codes described below as part of the low-level design of custom service implementations. The message codes should eventually be captured in the WebSphere Application Server logs and traces. The format of the error message numbers are defined by IBM corporate standards for software products. Following the standards, consistency should be maintained when using log generation utilities in service implementations. The naming conventions should be of the pattern <messageprefix><messagenumber><messagetype>. The message prefixes should define a four letter prefix for an error message. IBM WebSphere Telecommunications Web Services Server defines the following message prefixes:

- SOAC: Message codes are used by the IBM WebSphere Telecommunications Web Services Server runtime components to print messages in WebSphere system log and trace files. These message codes are *not* to be used by custom service implementations.
- SOAS: Message codes should be used to log messages only if new common components are implemented.
- SOAX: Message codes are intended for use in service implementations. Custom service implementations should utilize the “Available” range of message codes to capture meaningful operational information in WebSphere Application Server system logs and trace files.

Note:

- For Custom Service implementations, we recommend consulting the IBM WebSphere Telecommunications Web Services Server InfoCenter Web site to ensure that the range of SOAX message codes intended to be used *do not* collide with the message codes of existing IBM WebSphere Telecommunications Web Services Server components.
- In case of enhancements to Service Implementations, *do not* re-use or adopt existing message codes. Instead, create new message codes within the “Available” range.

The message number should be at least a four digit message number. The message type can be one of the following:

- T: Trace
- I: Information
- W: Warning
- E: Fatal Error

The message should utilize the {0} or {1} syntax for specifying a substitution parameter, where the digit indicates the position of the substitution parameter. When translated, these message substitution parameters may appear in different orders. Messages should never be constructed by assuming an order. At times, it could be necessary to append debug information to an error message. This can be done adequately if the debug information is surrounded by a identifiable delimiter, thereby having it appear as supplemental information that is not actually part of the translated error message.

Message numbers should also be allocated within a particular numerical range. These ranges are reserved for particular IBM WebSphere Telecommunications Web Services Server components. “IMS” refers to SIP based service implementation or other IMS component. “Parlay” refers to the Parlay connector component of IBM WebSphere Telecommunications Web Services Server. “DC” refers to Direct Connect service implementations. Refer to Table 7-1 for the number ranges used by various components in BM WebSphere Telecommunications Web Services Server as well as the number ranges that are available for service implementations.

Table 7-1 Message code ranges of IBM WebSphere Telecommunications Web Services Server

Component	Message code range
TWSS Common	SOAC0000 to SOAC1999
Service Platform Common	SOAC2000 to SOAC3999
ESB Common	SOAC4000 to SOAC5999
Available	SOAC6000 to SOAC9999

Table 7-2 shows the message code ranges for each of the common components in Service Platform.

Table 7-2 Message code ranges of Common components

Service Platform common components	Message code range
Common	SOAS0000 to SOAS0499
Admission Control	SOAS0500 to SOAS0999
Traffic Shaping	SOAS1000 to SOAS1499
Notification Management	SOAS1500 to SOAS1999
PX Notification Delivery	SOAS2000 to SOAS2499
Faults and Alarms	SOAS2500 to SOAS2999
Usage Records	SOAS3000 to SOAS 3499
Privacy	SOAS3500 to SOAS3999
Available	SOAS4000 to SOAS9999

Table 7-3 through Table 7-16 on page 239 shows the message code ranges for each of the Service Implementation components.

Table 7-3 Message code range for Common and Third-Party Call service

Service logic components	Message code range
Common	SOAX0000 to SOAX0499
Third Party Call	SOAX0500 to SOAX0999
IMS	SOAX0500 to SOAX0649
DC	SOAX0650 to SOAX0799
Parlay	SOAX0800 to SOAX0949
Available	SOAX0950 to SOAX0999

Table 7-4 Message code range for Call Notification service

Component	Message code range
Call Notification	SOAX1000 to SOAX1499
IMS	SOAX1000 to SOAX1149
DC	SOAX1150 to SOAX1299
Parlay	SOAX1300 to SOAX1449
Available	SOAX1450 to SOAX1499

Table 7-5 Message code range for Short Messaging service

Component	Message code range
Short Messaging	SOAX1500 to SOAX1999
IMS	SOAX1500 to SOAX1649
DC	SOAX1650 to SOAX1799
Parlay	SOAX1800 to SOAX1949
Available	SOAX1950 to SOAX1999

Table 7-6 Message code range for Multimedia Messaging service

Component	Message code range
Multimedia Messaging	SOAX2000 to SOAX2499
IMS	SOAX2000 to SOAX2149
DC	SOAX2150 to SOAX2299
Parlay	SOAX2300 to SOAX2449
Available	SOAX2450 to SOAX2499

Archive

Table 7-7 Message code range for Payment service

Component	Message code range
Payment	SOAX2500 to SOAX2999
IMS	SOAX2500 to SOAX2649
DC	SOAX2650 to SOAX2799
Parlay	SOAX2800 to SOAX2949
Available	SOAX2950 to SOAX2999

Table 7-8 Message code range for Account Management service

Component	Message code range
Account Management	SOAX3000 to SOAX3499
IMS	SOAX3000 to SOAX3149
DC	SOAX3150 to SOAX3299
Parlay	SOAX3300 to SOAX3449
Available	SOAX3450 to SOAX3499

Table 7-9 Message code range for Terminal Status service

Component	Message code range
Terminal Status	SOAX3500 to SOAX3999
IMS	SOAX3500 to SOAX3649
DC	SOAX3650 to SOAX3799
Parlay	SOAX3800 to SOAX3949
Available	SOAX3950 to SOAX3999

Table 7-10 Message code range for Terminal Location service

Component	Message code range
Terminal Location	SOAX4000 to SOAX4499
IMS	SOAX4000 to SOAX4149
DC	SOAX4150 to SOAX4299
Parlay	SOAX4300 to SOAX4449
Available	SOAX4450 to SOAX4499

Archive

Table 7-11 Message code range for Call Handling service

Component	Message code range
Call Handling	SOAX4500 to SOAX4999
IMS	SOAX4500 to SOAX4649
DC	SOAX4700 to SOAX4799
Parlay	SOAX4800 to SOAX4949
Available	SOAX4950 to SOAX4999

Table 7-12 Message code range for Audio Call service

Component	Message code range
Audio Call	SOAX5000 to SOAX5499
IMS	SOAX5000 to SOAX5149
DC	SOAX5150 to SOAX5299
Parlay	SOAX5300 to SOAX5449
Available	SOAX5450 to SOAX5499

Table 7-13 Message code range for Multimedia Conference service

Component	Message code range
Multimedia Conference	SOAX5500 to SOAX5999
IMS	SOAX5500 to SOAX5649
DC	SOAX5650 to SOAX5799
Parlay	SOAX5800 to SOAX5949
Available	SOAX5950 to SOAX5999

Table 7-14 Message code range for Address List Management service

Component	Message code range
Address List Management	SOAX6000 to SOAX6499
IMS	SOAX6000 to SOAX6149
DC	SOAX6150 to SOAX6299
Parlay	SOAX6300 to SOAX6449
Available	SOAX6450 to SOAX6499

Archive

Table 7-15 Message code range for Presence service

Component	Message code range
Presence	SOAX6500 to SOAX6999
IMS	SOAX6500 to SOAX6649
DC	SOAX6650 to SOAX6799
Parlay	SOAX6800 to SOAX6949
Available	SOAX6950 to SOAX6999

Table 7-16 Miscellaneous and Available code ranges

Miscellaneous components	Message code range
SIP Servlets	SOAX7000 to SOAX7499
Available	SOAX7500 to SOAX9999

Configuration conventions

The configuration methods to be followed in IBM WebSphere Telecommunications Web Services Server can be categorized as follows:

- Policies: Creation of Policies is the recommended way to store configuration data that affects the behavior of service logic. Service logic uses the policy information while processing requests. Policies can be applied based on the requester, the service, and service operation that is being requested. Policies can be administered using the Service Policy Manager administration console. Refer to Chapter 3, "Working with service policies and the Service Policy Manager" on page 33 for more information about creating policies.
- IBM WebSphere Telecommunications Web Services Server Administration Console properties: IBM WebSphere Telecommunications Web Services Server Administration Console can be located at the WebSphere Application Server Integrated Services Console. The configuration properties offered by IBM WebSphere Telecommunications Web Services Server Administration console are applicable for deployment of services, interconnectivity of a service with common components, and connectivity to network resources.
- Network Resources: In IBM WebSphere Telecommunications Web Services Server, some of the network resources properties of network elements are captured in a generalized common component by the name Network Resources. These properties can be defined at the deployment time of a service that requires connectivity with a network resource. The Traffic Shaping component limits the request load sent to a specific network resource by accessing the properties configured in Network Resources. The administrator can define resource names and corresponding properties for each network resource. The network resource name should be specified in the configuration section of individual service implementations that require access to a specific network resource.
- Deployment Descriptors: Deployment descriptors are artifacts that allow you to define environment variables that can be set during the pre-deployment configuration of a service. This type of configuration is intended only to be static for a service implementation and these configurations do not surface in administrator console. These settings are not dynamic enough for 24/7 operations, since the application should be redeployed for the configuration settings to come into effect. Such configuration settings should be used rarely.

- JNDI String Bindings: JNDI String bindings can be used to set dynamic configuration values in the WebSphere Application Server Integrated Services Console. JNDI bindings are reasonably dynamic in nature. These bindings can be changed by an administrator frequently. These are sometimes necessary to configure software components that are not contained within any specific enterprise application archive (EAR).
- Java Process Custom properties: Java System properties can be used to set properties for customizing the operation of specific Java classes in cases when no other option is possible to affect the behavior of the operation. However, this option is not desirable, because such a setting requires a restart of the server environment, and at times might be unsafe for the server run time.
- Properties files: Java applications can store configuration information in *.properties* files. However, configuration settings using this approach are not user friendly and dynamic enough for 24/7 operations.

Internationalization conventions

IBM WebSphere Telecommunications Web Services Server service implementations frequently need to have certain configurations that rely on the locale in which either the server is hosted or the request is originated from. The most common use case are text strings that are typically messages and captions on the administration console that should be localized. However, there are other internationalization issues, such as date time formats and currency formats that should be considered.

Translated strings for messages and captions are externalized and stored in a .properties file. The IBM WebSphere Telecommunications Web Services Server conventions are to store such strings in US English format, which is the default master copy for all messages and captions. The .properties file typically has the format of a file name followed by a short code representing the language and country, for example, ServiceProperties_en_US.properties holds service properties in American English. During the component build phase, these files are copied to generic fallback .properties file, such as ServiceProperties.properties. Other locale specific strings can be constructed and added to the build as required.

The generic .properties file without a locale suffix can have any configuration data or other types of data in addition to messages and captions, which do not need to be localized, such as error messages.

The localized captions and similar text are displayed in various service configuration pages that are part of the IBM WebSphere Telecommunications Web Services Server administration console. Using the JMX MBean framework, captions and other messages can be made available to the IBM WebSphere Telecommunications Web Services Server administration console. Developers should use .properties, depending the localization support requirement, for the JMX MBean framework. In addition to this functionality, developers should provide a help page for each Form page or Configuration Page by providing adequate user guidance. The convention for storing localized .html pages is to have each localized Web page created in a locale-specific folder. For example, the folder en_US should have a help_page.html that is in American English. All translatable material should be in UTF-8 encoding format when stored in the build.

IBM WebSphere Telecommunications Web Services Server components must be translatable to Group 1 languages by default. The Group 1 languages are:

- English
- Japanese
- Traditional Chinese
- Simplified Chinese

- Korean
- French
- German
- Italian
- Spanish
- Brazilian Portuguese

Message bundles

Message bundles are an essential part of a service implementation that allow you to convey context specific information, typically in operational messages, debug diagnostics, and so on. By default, debug diagnostics are not translated, but if they are required, this behavior can be customized. Operational messages can be translated depending on the locale. Follow the naming conventions for messaging bundles described in “Policy naming conventions” on page 230.

Additionally, since the messages need to be documented in a user guide, you must also provide a description of the usage of a particular message, and the actions that are needed to be taken by the administrator, if any. It is convenient to include the usage and actions along with the message ID and description in the same message bundle using the aforementioned conventions.

As an example, each message might appear in this form in the message resource bundle, as shown in Example 7-2.

Example 7-2 Sample message entry

```
SOAX4999 = SOAX4999E:{0} {1} the sample error message with a substitution of {2}.
SOAX4999_detail = A detailed description of the message
SOAX4999_action = The action that is recommended for the Administrator
```

The message itself may have substitution parameters. Typically in IBM WebSphere Telecommunications Web Services Server, the {0} is replaced with the transaction identifier, and the {1} is replaced with the component or service identifier. Utility methods are provided in the IBM WebSphere Telecommunications Web Services Server components to extract the message information from the message resource bundle.

Trace logging and First Failure Data Capture

Tracing and message logging of operational messages to the WebSphere Application Server logs is a requirement for IBM WebSphere Telecommunications Web Services Server service implementations and it is also the primary way problems are debugged. It is important to plan the implementation of logging and tracing at significant points in the code. Utility methods are provided in the application template framework to manage logging tasks in a consistent way. Excessive tracing can inhibit performance, so you should use the isLoggingEnabled() method before constructing message strings with dynamic information for logging.

Additionally, First Failure Data Capture (FFDC) is a requirement for IBM WebSphere Telecommunications Web Services Server. Any service implementation code that runs in the IBM WebSphere Telecommunications Web Services Server framework should also provide for the FFDC feature. The utility methods for trace related to logging for an exception will also provide FFDC information automatically. FFDC is very useful when diagnosing a problem in the IBM WebSphere Telecommunications Web Services Server environment, in the event trace is turned off, and the size of the SystemOut.log files are overwritten due to limited file size.

Security

When designing a service implementation that communicates with another system that is external to IBM WebSphere Telecommunications Web Services Server, it is important to determine how the service implementation will relate to the security model, and how the security for the connection to the network elements will be handled. Some of the aspects that should be clarified are:

- Whether the network element being connected by the service implementation is inside the IMS trusted network and responds to asserted identity headers for authentication.
- If so, what security mode is the IMS component running in and whether is there a need to have the IMS Trusted Asserted Identity (TAI) installed in the environment.
- What is the mechanism required to provide appropriate asserted identities on the client side, and accept the asserted identities on the server side?
- Alternatively, does the service implementation need to provide any other configuration for security credentials to be passed on to the network element? In general, the network elements are provided by other companies, and they will define the security capabilities of those elements. The protocol specifications may or may not provide information about how security is accomplished.

High availability, scalability, failover, and reliability

High availability is a very important requirement for IBM WebSphere Telecommunications Web Services Server service implementations. High availability means that the functions of the service must be available to the application on a continuous operational basis. This is sometimes referred to as 24/7, which means 24 hours a day, 7 days week, and 365 days a year, which in theory leaves no scope for scheduled downtime for maintenance or failures that disrupt operations.

The clustering architecture of WebSphere Application Server enables multiple servers that process requests, so that there is always a server available to handle incoming requests. Clustering of application instances is intended to avoid single points of failure. In effect, clustering enables redundancy for system resources.

High availability also relates to scalability, which is the ability to grow the capacity of the system dynamically, ensuring that the system always has the capacity to handle incoming requests and avoid failures due to overload. As the system load increases, additional capacity should be added to the system so that if a node failure occurs, the remaining capacity is adequate to continue operations at an acceptable load level. Scalability of the system is generally handled through clustering, although there are other factors for scalability, such as performance bottlenecks in the architecture, where the number of requests that can be realistically processed at once are limited by some constraints.

In IBM WebSphere Telecommunications Web Services Server, there are two basic types of request flows: A basic Web service request, which in itself could be a request-reply or one-way request, and a notification event. Typically, a third-party application makes a Web service request to IBM WebSphere Telecommunications Web Services Server to perform some operation. This request propagates through the system and typically translates into another request to a network element. The Web service requests benefit from the WebSphere HTTP proxy and load balancing behavior to provide high availability and redundancy within a cluster. The notification event is more complicated in terms of high availability and depends greatly on the network protocol that is being used.

Notification events are typically asynchronous events sent back to the IBM WebSphere Telecommunications Web Services Server cluster by network elements. When a network element needs to send an asynchronous event to IBM WebSphere Telecommunications Web Services Server cluster, it needs to be able to deliver that event to any of the server instances

in the IBM WebSphere Telecommunications Web Services Server cluster. Most protocols are not easily routable, such as HTTP, and require point-to-point communication (such as Parlay over CORBA/IIOP or SMPP). During the design phase, it is necessary to determine how the callback notifications will be handled by the connector logic. Much of the logic depends on the network protocol and network element behavior. By leveraging WebSphere Application Server features for high availability and scalability, as well as the messaging resources and connector infrastructure, high availability and failover can be achieved. Appropriate design decisions should be taken to handle the events of failure, and an approach to determine how requests can be re-routed to a functioning server instance should be adopted.

If the protocol is HTTP based, you should send the event back through the HTTP proxy, which can proxy the request to a working application server. For other protocols, it is usually necessary for the service implementation to detect which other server instances have been stopped or failed and notification registrations targeted to the stopped server should be re-initialized so that they can be targeted to a working server instance. The Parlay Connector does this automatically for the Parlay service calls. Other protocols generally need to have designated primary and backup server instances configured so that redundant callback handling exists.

For Non-HTTP protocols, the failover, recovery, and overall high availability of services depends on the protocol and the network element. A detailed analysis of the possible failure scenarios and recovery procedures as well as the corresponding effects on the request processing should be done during the design phase. In general, additional coding that leverage WebSphere Application Server components is necessary to have the desired high availability.

Designing for performance

During the design phase, it is also important to determine what kind of performance the new service implementation is expected to have. You should determine what processing the service implementation needs to perform that may inhibit performance.

The typical performance bottleneck is access to a database. It is important to design a service in such a way as to minimize the number of creates or database table inserts, table reads, updates, or deletes on table from a database. If a service needs to do a lookup from the database, a database record index should be provided in the Database Definition Language (DDL) for every index, which is leveraged when doing a EJB Finder or EJB Select lookup. Primary key fields have implicit indexes created that allow for faster retrieval of data. The use of a JDBC™ Prepared Statement is the preferred method to access database records.

Database indexes are important for another reason. When doing a database SELECT, multiple rows may be locked during the processing of records, and more rows may be locked than the records that match the SELECT statement. This can easily cause a database deadlock in which each of the two threads are waiting for the other to complete. The indexes will ensure that only records that match the selection are locked, thereby significantly reducing the chance of a deadlock.

Local EJB implementations should be used for entity beans to improve performance. Local transactions can also be used for entity beans to improve the performance. When using entity beans, if the entity bean has more than one field that is being updated, it is desirable to include the updates in a stateless session bean that defines the transaction boundary, which will reduce database accesses.

Note: You can also use the Java Persistence API as an alternative to using entity beans, although we do not cover this approach in detail within this scope of this IBM Redbooks publication.

Access intents define how the application will access the data, and whether it is likely if multiple threads of execution will need to update the same row at the same time. Generally in service implementations, to avoid locking the database tables, and avoid delays in responding to calling applications, we recommend using wsOptimisticUpdate for all methods in entity beans.

Entity bean references, Data Source references, and other J2EE resource references are JNDI names that can be cached. Accessing JNDI names is particularly slow, so you should cache the JNDI references in the service implementation logic.

Performance Monitoring Infrastructure

Performance Monitoring Infrastructure (PMI) is another feature of WebSphere Application Server that provides operational information that can be captured by operational monitoring applications. The IBM WebSphere Telecommunications Web Services Server platform provides some basic utility methods for generating performance metrics.

The metrics should be defined during the design phase, and should follow these guidelines:

- Each component should provide a time metric that points to when the service was started or first used, which provides the ability to calculate service availability.
- Each request should provide a count of the total number of requests received, as well as the number of successful and failed requests.
- Latency times for access to database and calls to common components or network elements are also sometimes meaningful.
- Latency times for the time duration between receipt of the request and passing the request on to back-end network element are also sometimes meaningful.
- Current size counts for service data structure sizes that are in-memory structures like hashtable objects or session instance objects are also sometimes meaningful.

Usability and accessibility requirements

The IBM WebSphere Telecommunications Web Services Server components have usability requirements; however, these requirements are not very applicable to the service implementations themselves, since the actual user interface is rendered by the IBM WebSphere Telecommunications Web Services Server Administration console or SPM console. Therefore, from the perspective of service implementations, the goal is to provide consistency with existing software that relates to the service implementations, such as the IBM WebSphere Telecommunications Web Services Server Administration console, help pages for configuration pages, product InfoCenter, and other forms of documentation.

If a custom service implementation requires an administrative or other self-help user interfaces, particularly one with accessibility to telephone users, such a design should conform to IBM Usability design procedures. For more information, refer to the following link:

`http://www-935.ibm.com/services/us/index.wss/offerfamily/igs/a1023541`

User interfaces must also be made accessible for users with special needs. Here are a few guidelines to avoid problem areas:

- Ensure that all functions are available through the keyboard.
- Associate labels, names, and titles to the data entry fields that they relate to.
- Set the initial focus when the window is loaded.
- Ensure that all fields of the form window can be accessed with a keyboard.
- Avoid sound, multimedia, animation, colors, low-contrast, or blinking as the only means of conveying information.
- Avoid using timed responses.
- Use system colors and fonts.

Documentation requirements

Service implementations produce a variety of information that should be documented in a user reference. During the design phase, ensure that these items are fully documented and described:

- Supported usage scenarios
- Usage Records attributes
- Error messages
- Faults
- Alarms
- Policies used
- Administration console configuration parameters
- Protocol versions and any limitations
- Limitations to any service functions
- Database tables that the service provider is expected to interface with
- Interfaces to any integration points unique to the service
- Failover configuration issues

Test requirements

During the design phase, it is important to consider what testing will be required for the service implementation and what testing tools may be required. Service implementations are expected to perform as described in the appropriate standards and specifications. Aside from functional testing, there are other operational requirements, such as:

- Running in a cluster, and scaling the work load, to test work load distribution
- Stopping and starting server instances while maintaining load, to test failover conditions
- Failover testing for all system nodes
- Recovery from failures to maintain continuous service
- Interoperability tests with devices and network elements of vendors adhering to appropriate versions of specifications
- Utilizing simulations of network element functions to verify functionality
- Utilizing simulations of network elements to drive stress and soak tests to identify performance bottlenecks in service implementations.
- Verifying the affects of configuration changes on a running system

Archived

- National language support and globalization testing
- Accessibility testing

7.4 Sample Parlay X Web service scenario

In this section, we will take you through the design of a Parlay X Web service implementation, by using the example of a Parlay X Presence Supplier interface following the Parlay X V2.1 'ETSI ES 202 391-14 V1.2.1 (2006-12)' specification. This service is not fully functional and not intended for use in a production environment, and is primarily used as a reference for developing custom service implementations.

Note: For more information about Parlay X V2.1 specifications, go to:

http://portal.etsi.org/docbox/TISPAN/Open/OSA/ParlayX21.html

Note: The Presence Supplier code supplied with this book cannot be replicated, and is provided for reference purposes only.

7.4.1 Presence Supplier detailed design

In this section, we will discuss the design of Parlay X Presence Supplier design details and the dependencies on the IBM Presence server architecture. IBM Presence Server provides a robust presence aggregation environment that enables integration with SIP/SIMPLE standards-based applications, such as instant messaging and presence-enabled address management tools. IBM Presence Server leverages the capabilities of IBM XML Document Management Server (XDMS) to provide advanced features, such as access to Watcher information, Authorization rules, Partial publish, Partial notify, and so on.

Parlay X Presence Supplier service interface

A Presence Supplier application is typically responsible for publishing presence data on behalf of a *presentity* to a Presence Server. Each service operation expects that the request is associated with a presentity or user's identity. As per the Parlay X Presence Supplier specification, a presence supplier application can have following functionalities:

- publish() service operation: Called by the presentity to publish the presence data of a presentity.
- getOpenSubscription() service operation: Called by the presentity to access open or pending subscriptions and corresponding subscribed attributes from individual watcher identities interested in its presence information.
- updateSubscriptionAuthorization(): Called by the presentity to authorize open or pending subscription of a specific watcher. This operation also has the scope to allow or disallow a watcher's access to specific presence attributes. Depending on the service implementation, the presentity may not be required to call this operation. If authorization policies for the watcher already exist in the Presence Server environment, the Presence Server might accept the watcher's subscription request and give access to subscribed presence data.
- getMyWatchers(): Called by a presentity to access the list of watcher identities or subscribers who are interested in its presence information.

- getSubscribedAttributes(): Called by a presentity to access the list of presence attributes that an individual watcher is interested in accessing.
- blockSubscription(): Called by a presentity to block a watcher's subscription to its presence information. This results in the termination of a watcher's active or pending subscription.

Presence Supplier design dependencies and limitations

The Parlay X Presence Supplier specification assumes that a check for the identity of a user or presentity that is being requested by the client application or device is successfully affirmed before the Presence Supplier service instance services the request. IBM WebSphere Telecommunications Web Services Server run time leverages the WebSphere Application Server security infrastructure to authenticate and assert the identity of the user associated with an incoming request by using HTTP(S) based authentication.

Apart from ascertaining the identity of the user of an incoming request, the Privacy common component implementation, when deployed, performs an authorization check on the user identity. This check protects the IBM WebSphere Telecommunications Web Services Server environment from unauthorized access to services.

Certain functions from the aforementioned list of functionalities have dependencies on the IBM Presence Server capabilities and the system architecture of the IMS service plane. One of the significant limitations is the reactive authorization. The Presence subscription process of execution might require you to add a step to perform a reactive authorization mechanism before processing a presence subscribe request. Additionally, the most notable dependencies on IBM Presence Server are:

- The getOpenSubscriptions() operation is required to have access to the watcher information, such as identity, state of subscription, duration of subscription, and so on. In addition, the individual presence attributes that are being subscribed to by the watcher is also required. IBM Presence Server partially fulfills this requirement by presenting watcher information. The required level of granularity to allow a subscription for specific presence attributes, and the ancillary functionality of persistence, filtration, and retrieval of specific presence attributes by watcher or presentity, is not available in Presence Server V6.2.
- The updateSubscriptionAuthorization() operation is required to update a watcher's access to a presentity's presence data. This requirement is partially fulfilled by Presence Server. The presence rules functionality of underlying XDMS can be leveraged to update a watcher's subscription. An XCAP request can be constructed and submitted to the XDMS Presence rules server, in order to update a watcher's subscription information. However, updates related to specific presence attributes cannot be done. This refers back to the same point that Presence Server does not provide this functionality.
- The getSubscribedAttributes() operation also depends on Presence Server capabilities to serve Presence attributes that are being subscribed to by a watcher.

Presence Supplier service components

The Parlay X Presence Supplier service is realized as a Web service. The service logic is made up of the following components:

- The Web service implementation with Java Bean binding, along with the logic to invoke common components of IBM WebSphere Telecommunications Web Services Server
- SIP servlets that handle SIP signalling to and from the Presence server
- XCAP request and response handling to and from XDMS Presence rules server
- Java Management Extensions MBean components to manage the service in IBM WebSphere Telecommunications Web Services Server

- Finally, logging and trace messages at various stages of processing

From the system architecture perspective, the components in Presence Supplier service function in a converged container. The Web service logic should be able to share session information with SIP servlets as well as handle asynchronous responses, if any, from the SIP servlets. The System component view shown in Figure 7-2 gives a better understanding of the components that make up the service as well as the integration points with other salient components in the IBM WebSphere Telecommunications Web Services Server and IMS architecture.

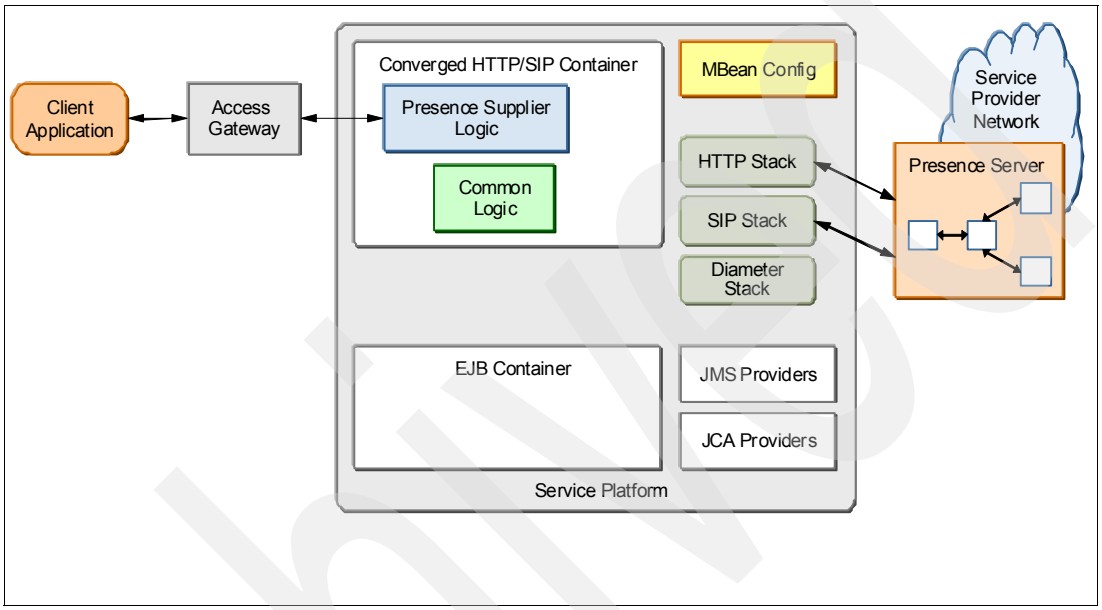


Figure 7-2 Presence Supplier Service in IBM WebSphere Telecommunications Web Services Server system architecture

Details of publish service operation

The publish service operation of Parlay X Presence Supplier service allows a client application to publish presence information for a presentity. The client application is a Web services client and it should carry authentication credentials for the presentity along with the Presence attributes as described in the specification. For more information about the Parlay X Presence specification, refer to Parlay X V2.1 'ETSI ES 202 391-14 V1.2.1 (2006-12). The sequence of interactions of Presence Supplier service operations with common components and SIP servlets that handle the communication issues with Presence are depicted in Figure 7-3 on page 249.

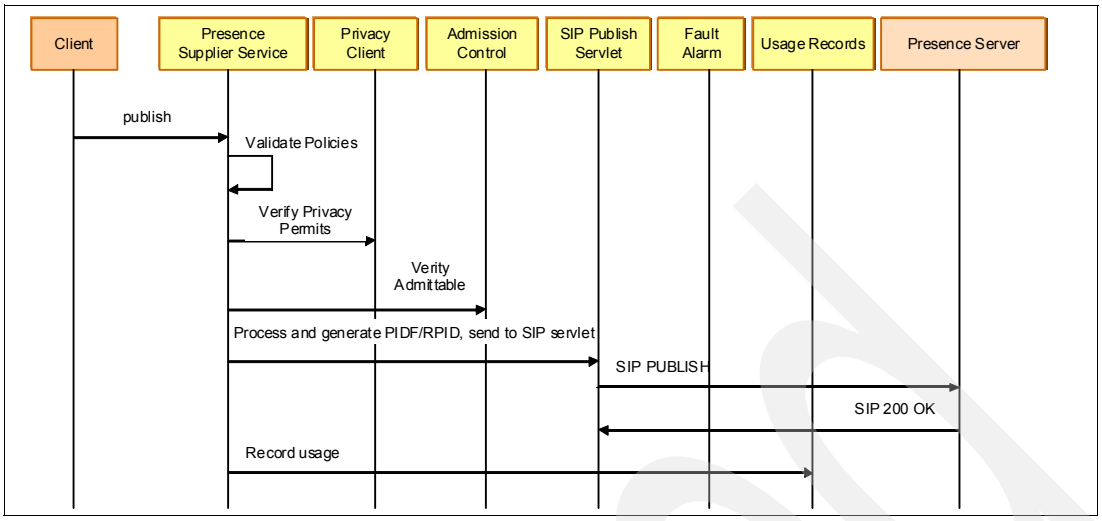


Figure 7-3 Object interaction diagram for publish method

In Figure 7-3, the sequence of interactions between the service logic and common components are as follows:

1. A publish request is sent by a Presence Supplier client application to the Presence Supplier Web service URL. The request comprises the Presence Attributes of a specific presentity. The request should carry a presentity's credentials as per the HTTP(S) conventions.
2. The incoming request is initially processed by the converged container, where an authentication check is performed on the credentials. The credentials are checked against an underlying user repository, such as an LDAP server. After a successful authentication, the request headers are extracted by the Service Platform API and the request is handed over to the Web service implementation. The Web service implementation code should perform validations on the incoming policies. Depending on the policies, certain policy values might require values from the service's JMX MBean attribute values for the construction of a sane request to a network element.
3. A privacy check should be the first common component invocation. This step applies authorization rules to the incoming request. Since Privacy Client is not implemented, this service invocation is just a way station to the next step.

   Note: Step 4 is optional only if the Privacy Client common component is configured for the Service. The relevant configuration can be done in the IBM WebSphere Telecommunications Web Services Server Administration console.

4. The next step in the interaction is the invocation of the Admission Control common component. The Admission Control component is responsible for protecting the IBM WebSphere Telecommunications Web Services Server run time from request overload conditions. In order for the Admission Control component to work, the necessary configurations should be done as part of installing the service. For more information about Admission Control specific configurations, refer to 8.6.1, "Admission Control configuration settings" on page 291.
5. Upon successful execution of the Admission Control service, the service logic should invoke the Traffic Shaping common component. The Traffic Shaping component consults the Network Resource component to determine the accumulated maximum number of requests that can be sent to a registered Network Resource. For more information about

the configuration specific to Network Resources, refer to 8.6.2, “Traffic Shaping configuration settings” on page 292.

6. On successful execution of Traffic Shaping component, the logic to compose a PIDF/RPID XML document is executed. The Presence attributes received in the request should be parsed and an RPID or PIDF document must be generated. Presence information should adhere to PIDF and RPID standards in order for Presence Server to process the information and persist it for later use. For more information about PIDF and RPID standards, refer to RFC 4480 and RPID: Rich Presence Extensions to the Presence Information Data Format (PIDF).
7. The logic to send a SIP message to the Presence Server is executed, which is a SIP Servlet in the case of the Presence Supplier service. The SIP servlet is responsible for sending a SIP PUBLISH message to the Presence Server and receiving a SIP 200 OK message from the Presence Server.
8. Because the publish operation is a one-way type service operation, the service implementation should invoke the writeUsageRecord operation on the Usage Records common component after a successful completion of the SIP message publish logic. This serves the purpose of recording the usage of each request that is being processed by IBM WebSphere Telecommunications Web Services Server.

Note: In the event that there is any exceptions during the invocation of the common components, a fault should be recorded in the system. Recording a fault is done by invoking the Fault and Alarm common component. The Fault and Alarm Web service relies on CEI resources, so in order to use the Fault and Alarm service, CEI should be enabled in the Service Platform environment. For more information about enabling CEI, refer to the Configuration section of the IBM WebSphere Telecommunications Web Services Server InfoCenter.

7.5 Conclusion

In this chapter, we discussed the IBM WebSphere Telecommunications Web Services Server design guidelines for developing custom service implementations. In addition, you learned what components a basic Parlay X Web service implementation is typically comprised of. In Chapter 8, “Developing the service implementation” on page 251, we will walk you through the development, building, and deployment of a custom Web service implementation.

Archived

8

Developing the service implementation

This chapter describes the approach and steps required to develop the custom service implementation from our common use case, namely the Presence publish operation. This chapter begins with a review of the development configuration requirements to begin creating the service implementation and then proceeds into the key steps for developing a Parlay X Web service. It discusses the core logic of the service implementation code, then discusses how to unit test and ultimately deploy and test the completed sample.

Note: The sample code for this service implementation is available for download. Refer to Appendix E, “Additional material” on page 399 for detailed instructions on how to download and work with this sample code.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

8.1 Introduction

In this chapter, we will discuss the step by step activities for developing a custom service implementation. In order to successfully develop and deploy a custom service implementation on IBM WebSphere Telecommunications Web Services Server, it is essential to understand the architecture and design of the IBM WebSphere Telecommunications Web Services Server system. Therefore, we strongly suggest you read Chapter 7, "Design considerations for the service implementation" on page 227 before continuing further.

This chapter focuses on developing implementation logic for the Parlay X Presence Supplier Web service interface. For more information about the Parlay X Presence specification, refer to Parlay X V2.1 'ETSI ES 202 391-14 V1.2.1 (2006-12).

Note: For more information about Parlay X V2.1 specifications, go to:

http://portal.etsi.org/docbox/TISPAN/Open/OSA/ParlayX21.html

8.2 Focus of this chapter within the context of the common use case

This chapter focuses on developing implementation logic for the Parlay X Presence Supplier Web service interface.

The key topics are as follows:

- "Development environment" on page 254
- "Developing a Sample Parlay X Web service" on page 264
- "Implementing the core logic of the service" on page 287
- "Configurations for a new service" on page 291
- "Management provisions for new service" on page 299
- "Deploy and test" on page 326

Figure 8-1 on page 253 illustrates the primary focus of this chapter within the context of the common use case.

Archived

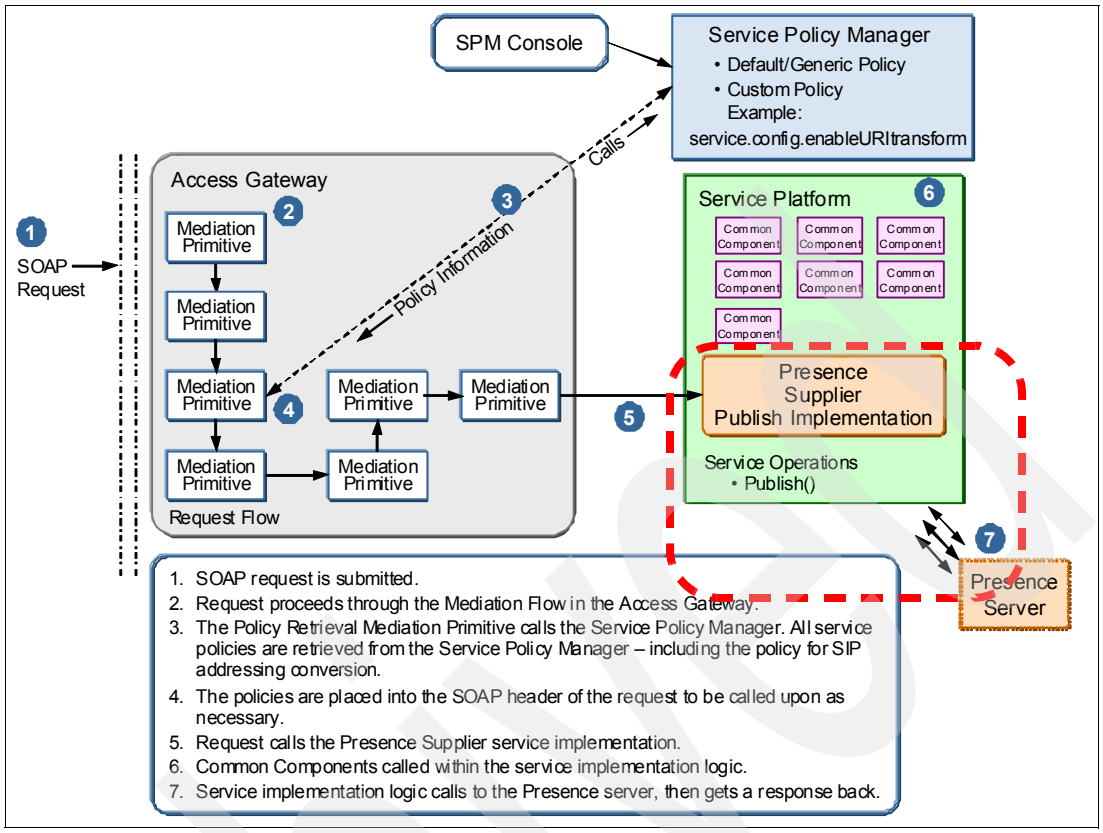


Figure 8-1 Focus for this chapter

8.2.1 Dependencies of the IBM WebSphere Telecommunications Web Services Server environment

IBM WebSphere Telecommunications Web Services Server comprises of three main components: Access Gateway, Service Policy Manager, and Service Platform. The out-of-box Parlay X services running in IBM WebSphere Telecommunications Web Services Server environment take advantage of the common components and common logic provided by the IBM WebSphere Telecommunications Web Services Server environment, such as:

- Policy-based access based on service level, operation level, and requester level policies.
- Avoiding overload conditions on inbound requests, specifically, imposing a limit on the number of requests that can be serviced on a per-service, per-operation basis by IBM WebSphere Telecommunications Web Services Server at any given time.
- Avoiding overload conditions on outbound requests, specifically, imposing a limit on the number of requests generated for a specific network resource by IBM WebSphere Telecommunications Web Services Server at any given time.
- Record usage based on the number of requests services by IBM WebSphere Telecommunications Web Services Server.

Similar to the out-of-box services, a custom Parlay X service should utilize the rich features of IBM WebSphere Telecommunications Web Services Server, and conform to the IBM WebSphere Telecommunications Web Services Server design guidelines. Depending on the requirements, a custom Parlay X is typically developed for two purposes:

1. To implement the functionality of a Parlay X interface that is not available out-of-box in IBM WebSphere Telecommunications Web Services Server.
2. To implement alternate or custom functionality to a Parlay X service that is available out-of-box in IBM WebSphere Telecommunications Web Services Server. This kind of service is often required in cases where the network elements have a proprietary means to fulfill the functionality.

As part of the IBM WebSphere Telecommunications Web Services Server services portfolio, the Parlay X Presence Service is available out-of-box. This service has implementations for Presence Consumer Interface, Presence Notification Manager, and Presence Supplier interfaces. However, the Presence Supplier implementation is not implemented.

As a sample scenario, in this chapter, we will discuss developing a custom Parlay X Presence Supplier service implementation. There are couple of implications that need to be dealt with when it comes to co-existence with an already available service. In this case, the out-of-box Parlay X Presence service has a Presence Supplier service, but the Presence Supplier service has not yet been implemented. We will deal with those problems in 8.6, “Configurations for a new service” on page 291 and 8.6.3, “Initial service policy settings” on page 293.

Important: As a general best practice, it is important to note that a new service implementation should be developed using the application template as a starting point (platform.ear) which is available beginning with IBM WebSphere Telecommunications Web Services Server V6.2.0.1. This application template enables a new service to take advantage of existing utilities and provide consistency with other IBM WebSphere Telecommunications Web Services Server services.

8.3 Development environment

The Rational Application Developer (RAD V7.0) is the recommended development environment. Rational Application Developer provides an integrated development environment (IDE) with user-friendly wizards and widgets for developing J2EE applications as well as Web Services based applications. Before we delve into the development steps, let us see the various development utilities and corresponding guidelines that are available to us in the IBM WebSphere Telecommunications Web Services Server installer.

8.3.1 Development utilities

IBM WebSphere Telecommunications Web Services Server provides a set of development utilities and guidelines in addition to the design guidelines described in Chapter 7, “Design considerations for the service implementation” on page 227. These utilities and guidelines will assist in developing custom service implementations. The utilities are part of the IBM WebSphere Telecommunications Web Services Server installer.

Service platform interfaces

When planning the development of a service implementation, it is important to understand the programming interfaces that are provided by the IBM WebSphere Telecommunications Web Services Server Service Platform. The Service Platform provides several useful interfaces, which are discussed in the following sections.

Parlay X Bindings

A IBM WebSphere Telecommunications Web Services Server service implementation will typically implement one of the Parlay X V2.1 Web services. Typically, this may involve several service interfaces, such as request, management, and notification interfaces. The parlayx21.jar, bundled as part of the Service Platform Application Template discussed in 8.3.2, "Service Platform Application Template" on page 256, has the pre-generated Java class files for all the supported Parlay X service interfaces and types. For a list of supported Parlay X services, refer to IBM WebSphere Telecommunications Web Services Server InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

The Java documentation of the classes for supported Parlay X services are available at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/parlayx21javadoc/overview-summary.html

Parlay X V2.1 WSDL files

The Parlay X V2.1 Web Service Definition Language (WSDL) files are also necessary to build a Web service implementation. These WSDL files are necessary for building the Web services deployment descriptors. Rational Application Developer provides wizards and widgets specifically for JSR 109 standards based Web service bindings code generation. During the generation of code, it should be noted that the default namespace is used to generate package names for interfaces and types; an example is shown in Example 8-1.

Example 8-1 Sample Parlay X Service namespace

```
http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface
```

The corresponding Java package generated by RAD V7.0 is shown in Example 8-2.

Example 8-2 Sample package name generated by RAD V7.0

```
org.csapi.www.wsdl.parlayx.presence.supplier.v2_3._interface
```

From these examples, there is a possibility of class names with package qualification exceeding RAD's file name length limit. We recommend defining custom package names adhering to IBM WebSphere Telecommunications Web Services Server design guidelines. Refer to 8.4.2, "Generating Web service bindings" on page 270 for more information.

Common component client interfaces

The service platform consists of a set of reusable common components that are embodied in Web services along with their respective Web Service Definition Language (WSDL) files. The Service Platform Application Template, which is delivered as part of the IBM WebSphere Telecommunications Web Services Server installation bundle, provides the admctl-client.jar, fltalm-client.jar, netres-client.jar, notifymgmt-client.jar, privacy-client.jar, pxnotify-client.jar, trafficsh-client.jar, and userec-client.jar files. These files comprise the pre-generated web service client Java classes.

Service Platform API

In order to ease the development effort for developing custom services, IBM WebSphere Telecommunications Web Services Server provides the new Service Platform API as part of the latest Fix Pack for IBM WebSphere Telecommunications Web Services Server. The Service Platform API provides utility classes to simplify the development of service implementations. These interfaces are documented in the IBM WebSphere Telecommunications Web Services Server Service Platform JavaDoc as part of the IBM WebSphere Telecommunications Web Services Server InfoCenter. For more information about this topic, go to:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/intro_c.html

The com.ibm.twss.platform.ServicePlatform class is the primary class that provides access to the following capabilities:

- Access to the IBM WebSphere Telecommunications Web Services Server Headers SOAP header information of the incoming request from Access Gateway.
- Access to the Web service client interfaces of all the common components. The underlying logic implicitly takes care of performance, security, and configuration integration in IBM WebSphere Telecommunications Web Services Server.
- PMI enablement.
- Trace and FFDC is provided.
- Policy and Configuration property encryption/decryption functionality is provided.
- Message bundle handling, message parameter substitution, and translation for globalization is provided.
- Administration and Topology information.

Encryption and decryption utilities

Some of the policies and configurations may contain sensitive data, such as security credentials or passwords. Both policies and configurations pertaining to a service are stored in a database table per the IBM WebSphere Telecommunications Web Services Server service design. The Service Policy Manager defines a special data type with the name “password”. The purpose of this datatype is to handle encrypted data while storing it in the database. The service implementation should decrypt such encrypted configuration settings and policies in order to process the request. For this purpose, the ServicePlatform class of ServicePlatform API provides utility methods to manage the encryption of data in a consistent way.

8.3.2 Service Platform Application Template

The Service Platform component of IBM WebSphere Telecommunications Web Services Server provides a service implementation application template with the name platform.ear, which is distributed in the IBM WebSphere Telecommunications Web Services Server installer. The platform.ear is a skeleton IBM WebSphere Telecommunications Web Services Server service implementation that is a J2EE application, which provides all the required classes and libraries that are required for a service implementation to function in the IBM WebSphere Telecommunications Web Services Server run time. The functionality provided by the libraries of platform.ear are broadly covered in “Service Platform API”. The Enterprise Application platform.ear consists of the following subcomponents:

- Common component client libraries, which have the pattern *-client.jar. These files are listed in “Common component client interfaces” on page 255.

- A sample Enterprise Application deployment descriptor, which can be located in the folder META-INF. It includes the application.xml, admin.xma, and other useful files.
- Libraries that required in the IBM WebSphere Telecommunications Web Services Server run time for the service implementation to function as desired.

The .jar files or libraries are typically used as the basis for developing additional IBM WebSphere Telecommunications Web Services Server Service implementations, and therefore are potentially embedded in the service platform application template.

Note: The Service Platform API will be available in IBM WebSphere Telecommunications Web Services Server installer beginning with IBM WebSphere Telecommunications Web Services Server V2.0.1.

Note:

- The Service Platform Application Template, which is the platform.ear, will undergo defect fix procedures just as any other component in IBM WebSphere Telecommunications Web Services Server would.
- Software updates to platform.ear with fixes will be made available as part of the IBM WebSphere Telecommunications Web Services Server iFix or Fix Pack deliverables.
- In the event an updated version of platform.ear is released as part of a IBM WebSphere Telecommunications Web Services Server iFix or Fix Pack, the custom service implementations that were developed using earlier versions of platform.ear should at the least import libraries that come with a new version of platform.ear. Refer to 8.3.4, “Update procedure for custom service implementations” on page 263 for more information about this topic.

Since custom service implementations are developed utilizing the service platform application template as a base, we recommend refreshing the implementation code base to incorporate the fixes to platform.ear as they are made available through the IBM WebSphere Telecommunications Web Services Server iFix or Fix Pack mechanism.

8.3.3 Creating a custom service project

Start the Rational Application Developer (RAD V7.0) and choose or create a new workspace. Import the platform.ear if it is a new service development. In the case of applying fixes to an existing service implementation, replace the older libraries with new ones and rebuild the projects.

Follow these steps to create a new service implementation project utilizing the Service Platform Application Template:

1. Import platform.ear into Rational Application Developer. Select **File** → **Import...**. The Import wizard is displayed, as shown in Figure 8-2.

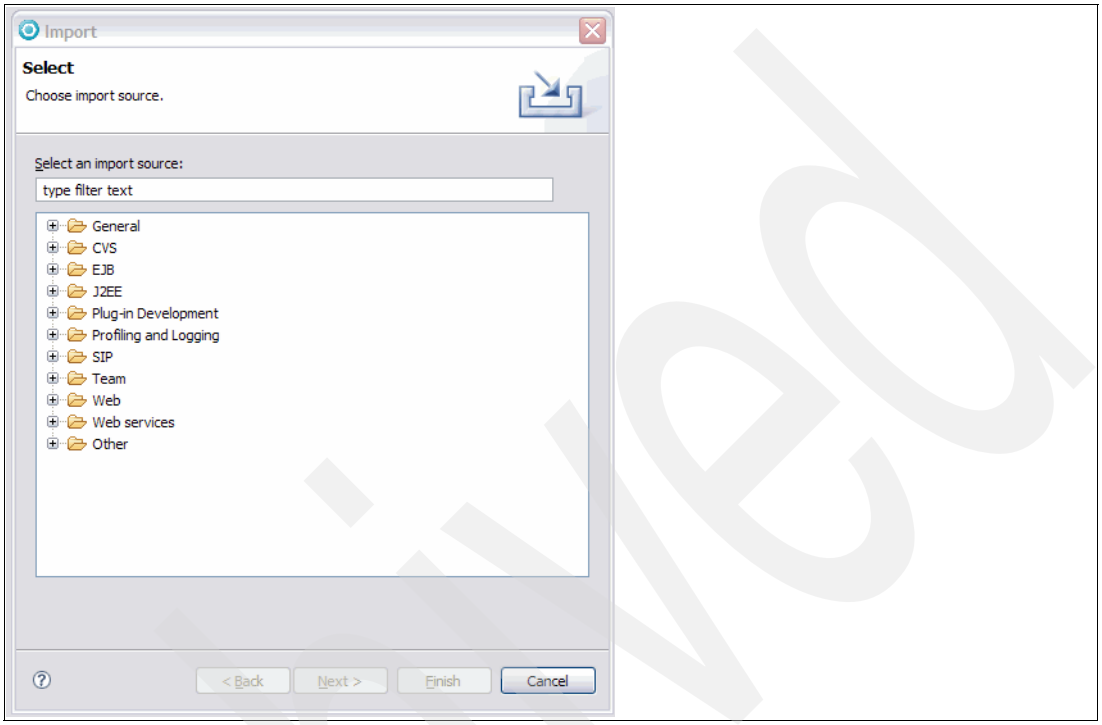


Figure 8-2 RAD V7.0 Import wizard

Archived

2. Expand the **J2EE** folder, select the **EAR file** option, and click the **Next** button, as shown in Figure 8-3.

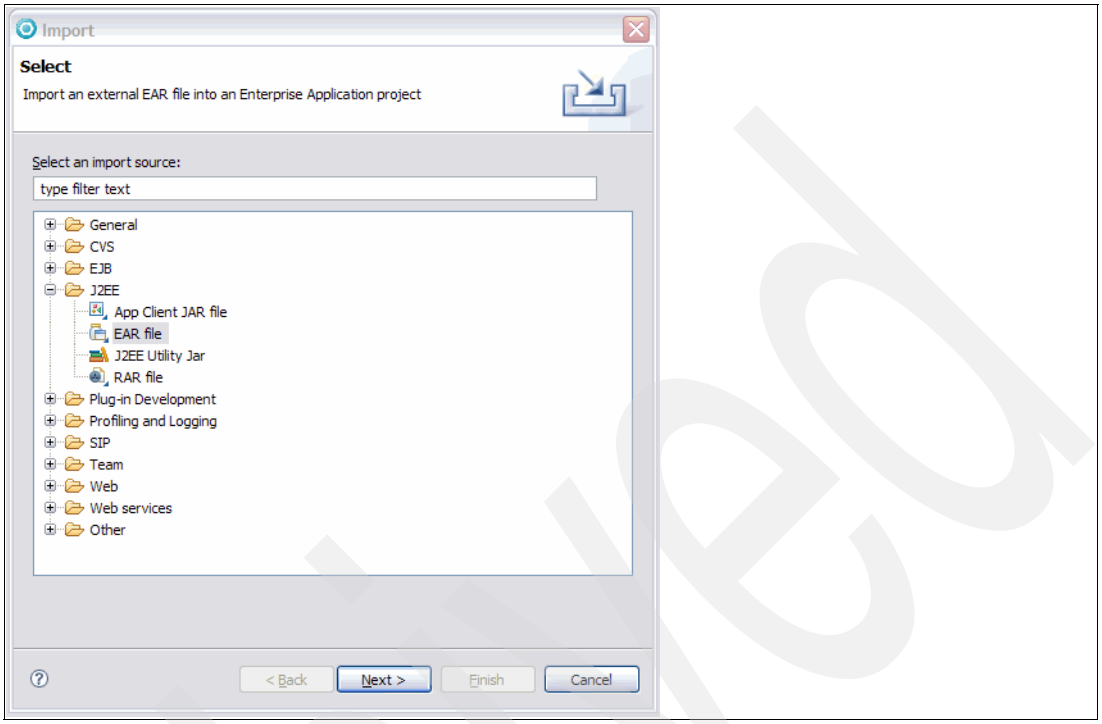


Figure 8-3 Import an EAR file

Archived

3. In the Enterprise Application Import window (Figure 8-4), select the platform.ear file location first. The EAR project combo box by default displays “platform” as the project name. This name should reflect the name of a new service implementation. Following the IBM WebSphere Telecommunications Web Services Server design guidelines for application naming described in “J2EE resources naming conventions” on page 232, rename the EAR project as PX21_PRS_SPLR_IMSApp. The name refers to Parlay X 21 Presence Supplier for IMS Application. The reason that you use the suffix App is to have a unique name for the enterprise application project. This will help in adding child projects, such as a SIP project to this enterprise application project without needing to rename or invent project names. Click the **Next** button.

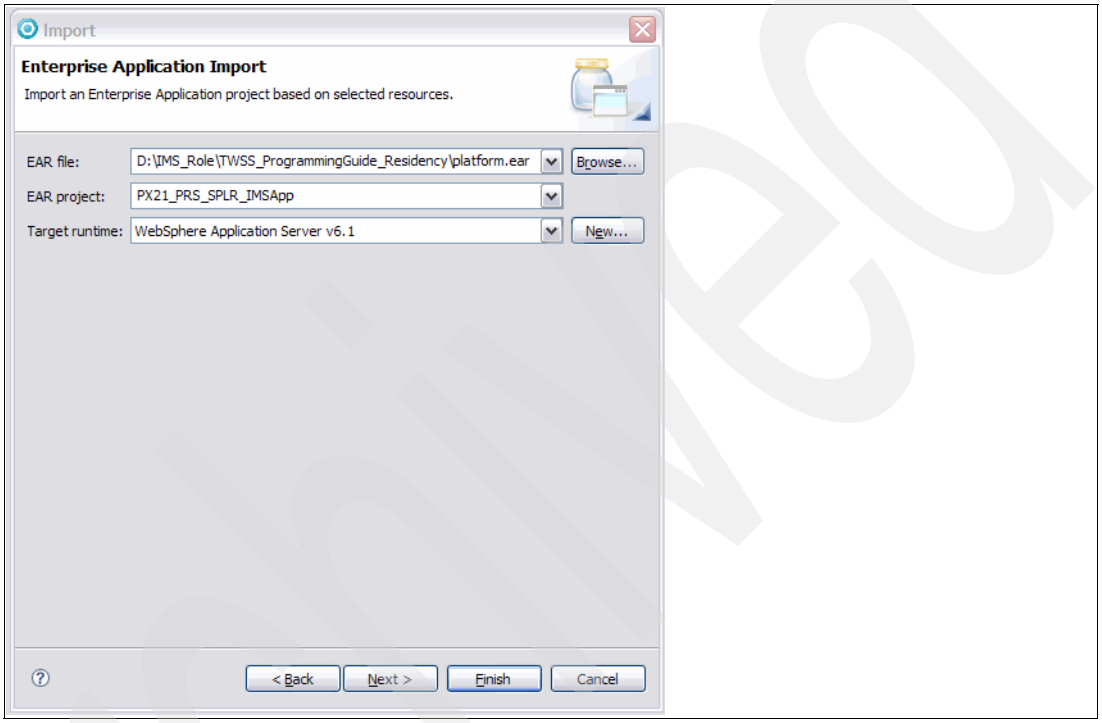


Figure 8-4 Name the New Service Project

4. The next window should display a list of libraries with check boxes for each library. The .jar files or libraries are a required part of the EAR project. In an enterprise archive (EAR), these libraries are stored as *utility JARs* if left unchecked. Select all the libraries, which will create utility projects under the enterprise application. Note that mustbeinear.jar should be unchecked so that it is imported as a JAR only, and not a utility project. The enterprise application at run time refers to the classes packaged in these utility JARs. The window should now look like Figure 8-5 on page 261.

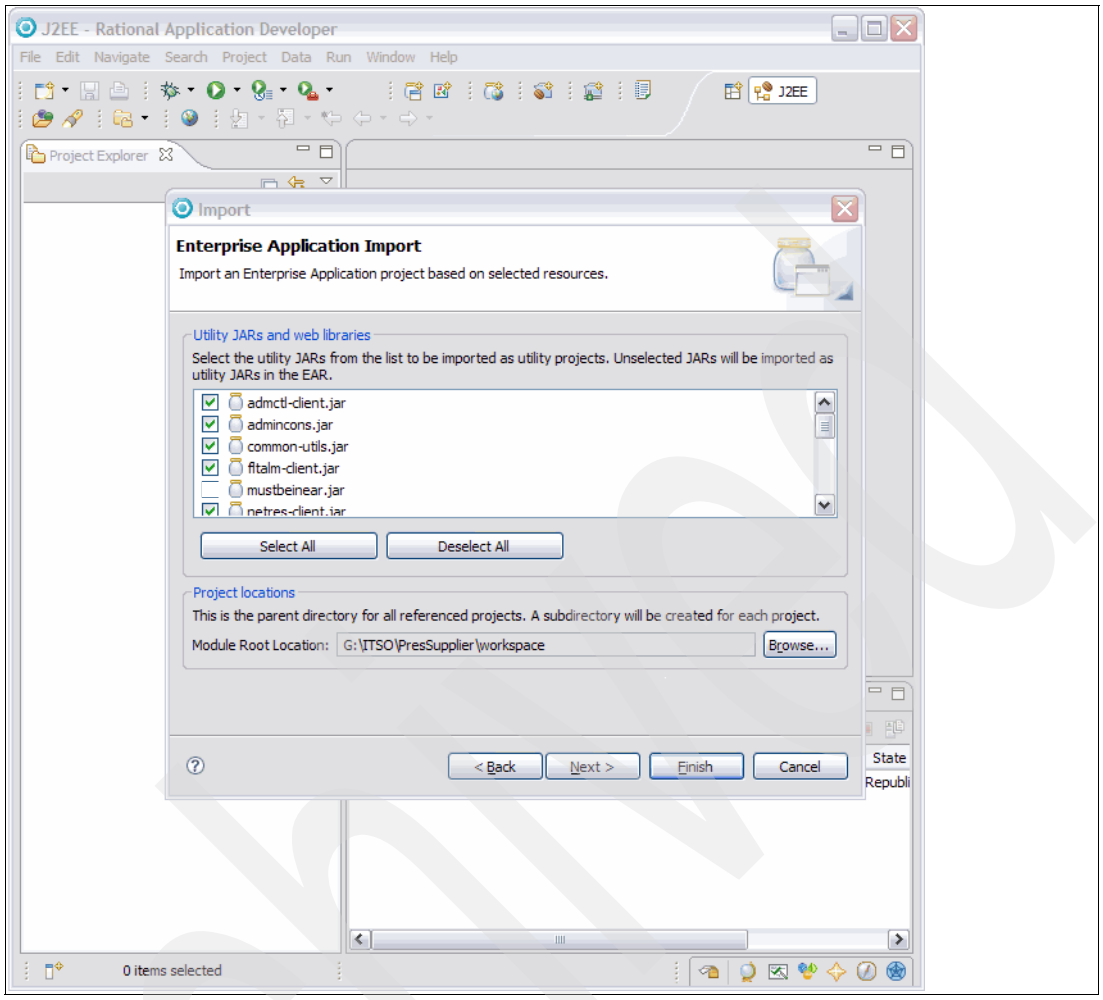


Figure 8-5 Select all Utility JARs

5. Click the **Next** button. In addition to the utility JARs, which are plain Java archives, admincons-ejb.jar and admincons-web.war are also packaged in platform.ear. The import wizard of RAD creates two child projects under the new enterprise application project, one for each of these archives. This is an optional step, so you can click the **Finish** button to complete the import process.

6. After successful completion of the above steps, the Project Explorer should appear as shown in Figure 8-6.

Note: In Step 4, if the checked boxes for the utility libraries are selected, then all the .jar files show up as Java projects in the project explorer. In addition, the utility JARs are added to the enterprise application project as utility JARs.

For your reference, the Deployment Descriptor with the name application.xml in the enterprise application project has a section called “Utility JARs”. All the imported JARs should be listed there.

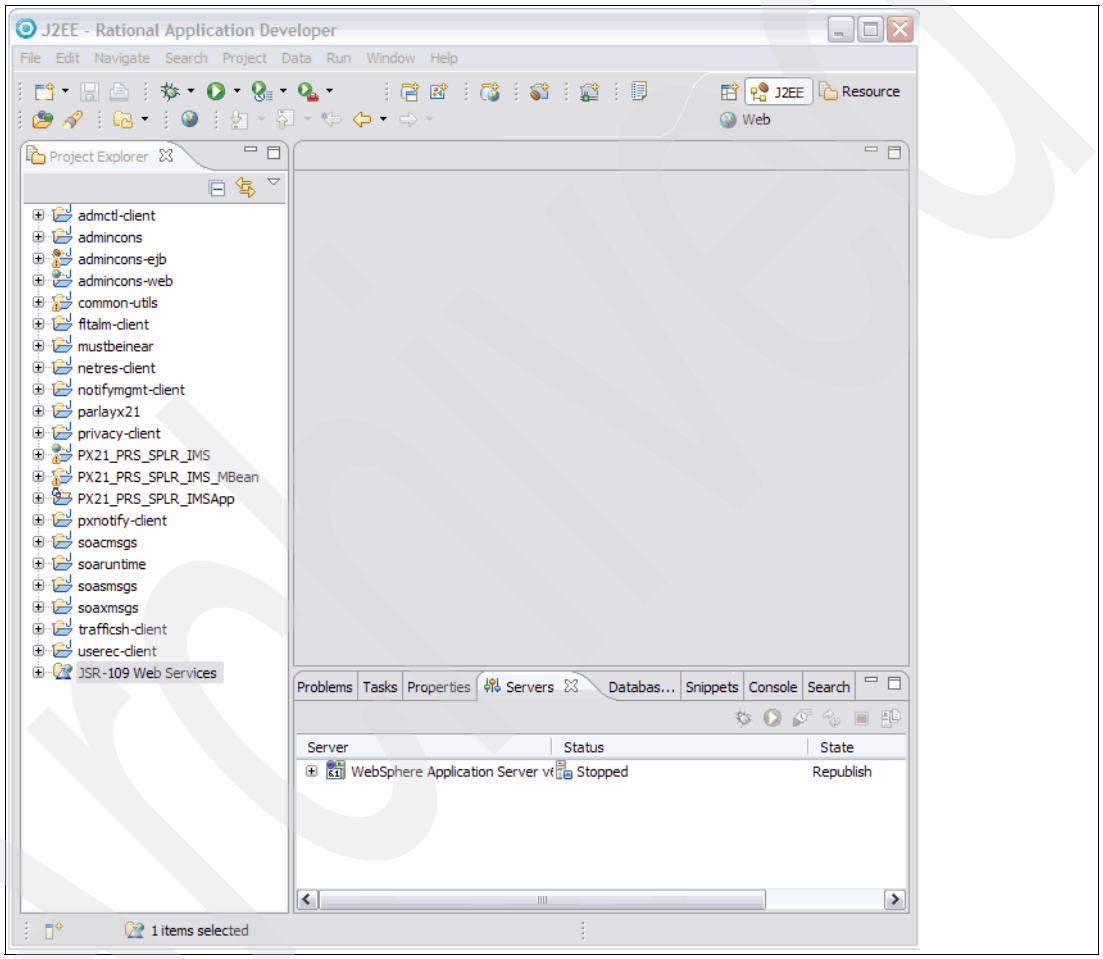


Figure 8-6 Initial project hierarchy of new IBM WebSphere Telecommunications Web Services Server service project

7. Expand the project in the Project Explorer. Locate the Deployment Descriptor: Platform file and open it. In the Display name, enter PX21_PRS_SPLR_IMS. Save the changes made by selecting **File** → **Save**. The window should now look like Figure 8-7 on page 263.

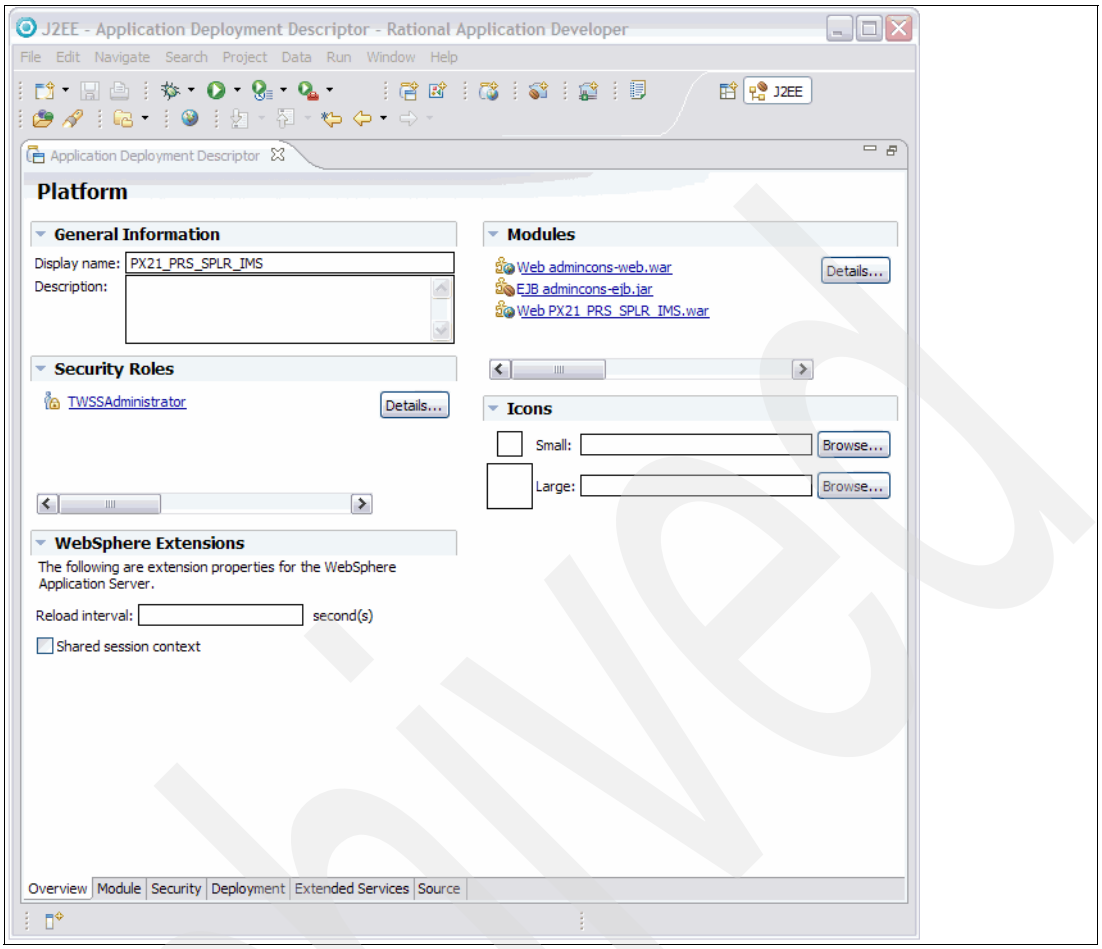


Figure 8-7 Naming the sample service enterprise application

8. Close the Deployment Descriptor by selecting **File** → **Close**. This concludes the creation of a enterprise application project for a custom service implementation by extending IBM WebSphere Telecommunications Web Services Server Service Platform Application Template.
9. The JNDI names used in the admincons-ejb.jar project should be changed so that they are unique. This is typically done by appending the service abbreviations at the end of the default JNDI name provided in the .jar file.

8.3.4 Update procedure for custom service implementations

As mentioned in 8.4, “Developing a Sample Parlay X Web service” on page 264, it is essential to apply the updated versions of platform.ear to custom service implementations. We recommend following the usage guidelines that come with the product

Typically, updates to platform.ear result in changes or additions to Java interfaces and methods. These changes might lead to updating the Java libraries that are part of platform.ear. In such cases, make a backup copy of the source of the existing custom service implementation project using the Rational Application Developer Project Interchange export mechanism. Import the new platform.ear following the steps detailed in 8.3.3, “Creating a custom service project” on page 257. Copy the service bindings logic from the backup copy into the new project. Make changes to the existing code in the Service bindings file as required to utilize the updated functionality provided by platform.ear. Generate and export the

the new Enterprise Application Archive (EAR). Re-deploy the service into the IBM WebSphere Telecommunications Web Services Server environment.

8.4 Developing a Sample Parlay X Web service

In 7.4, “Sample Parlay X Web service scenario” on page 246, we cover the introduction to the Parlay X Web service sample scenario. We now discuss the development activities needed to create a Parlay X Presence Supplier service implementation. It should be noted that this service implementation covers only the publish service operation and is provided for reference purposes only.

8.4.1 Service implementation prerequisites

The Parlay X Web service specification for Presence comes with a set of Web Service Definition Language (WSDL) files along with referenced schema definition (XSD) files. These WSDL files are provided by the IBM WebSphere Telecommunications Web Services Server installer to ease custom service development in the Java archive with name parlayx21wsdl.jar. The default namespace declarations of these Web services are typically long and result in longer package names for classes generated by Rational Application Developer. Because RAD has a file name limitation of 260 characters, the recommended approach to generate namespace to package mapping during code generation is to use the WSDLs provided in parlayx21wsdl.jar. The exact step by step instructions to generate namespace to package mapping is described in 8.4.2, “Generating Web service bindings” on page 270. Copy the parlayx21wsdl.jar to the development environment and extract the JAR file in a location accessible to Rational Application Developer.

Note: Typically, the development environment is a separate machine with a Windows® or Linux OS. The WSDL files packaged in parlayx21wsdl.jar should be accessible to the Rational Application Developer hosted on the development machine.

Note: During the installation of the IBM WebSphere Telecommunications Web Services Server Service Platform, the parlayx21wsdl.jar file is copied to the WebSphere Application Server install root, which is /opt/IBM/WebSphere/AppServer. Locate parlayx21wsdl.jar at {WebSphere Application Server install root}/installableApps/TWSS-Base/wsdl.

In general, Web service implementation code is typically generated and packaged in a Web project. In the case of the Parlay X Presence Supplier Web service, the service operations at some point should deal with SIP signalling to a Presence Server. As discussed in 7.4, “Sample Parlay X Web service scenario” on page 246, certain interactions between the service operations and Presence server, require the service implementation code to be able to receive SIP messages from the Presence Server. To cater to the requirements of the Parlay X Presence Supplier interface, one or more SIP servlets are required, and therefore a SIP project to contain the SIP servlets.

Follow these steps to associate a SIP project to the existing enterprise application:

1. Select the **PX21_PRS_SPLR_IMSApp** project in the Project Explorer and select **File** → **New** → **Project...**, as shown in Figure 8-8.

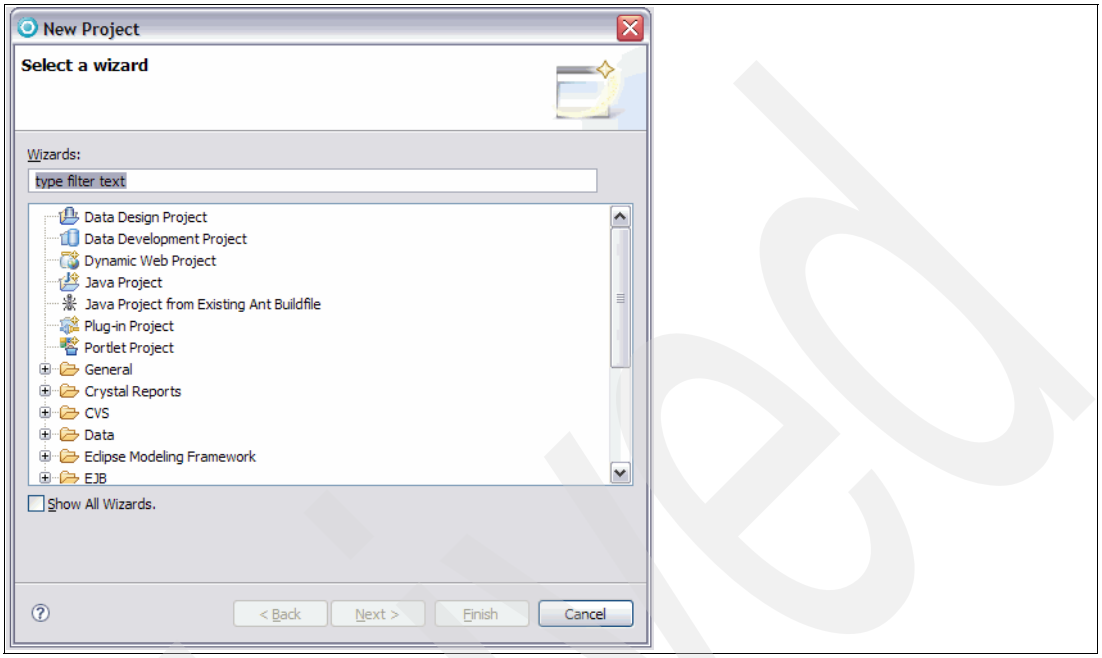


Figure 8-8 New Project wizard

Archived

2. In the New Project wizard, scroll down and expand the **SIP**, select **SIP Project**, and click **Next**, as shown in Figure 8-9.

> **Note:** We select **SIP Project** in this step because the Presence service is SIP related. Not all services will need a SIP project. However, other types of services may need some kind of notification project to receive an inbound notification from the back-end element. This will differ based on the back-end type. For example, for Parlay, an EJB project will be needed. For some HTTP based protocols, a servlet will be need. For other protocols that use JCA adapters, an EJB or MDB may be needed.

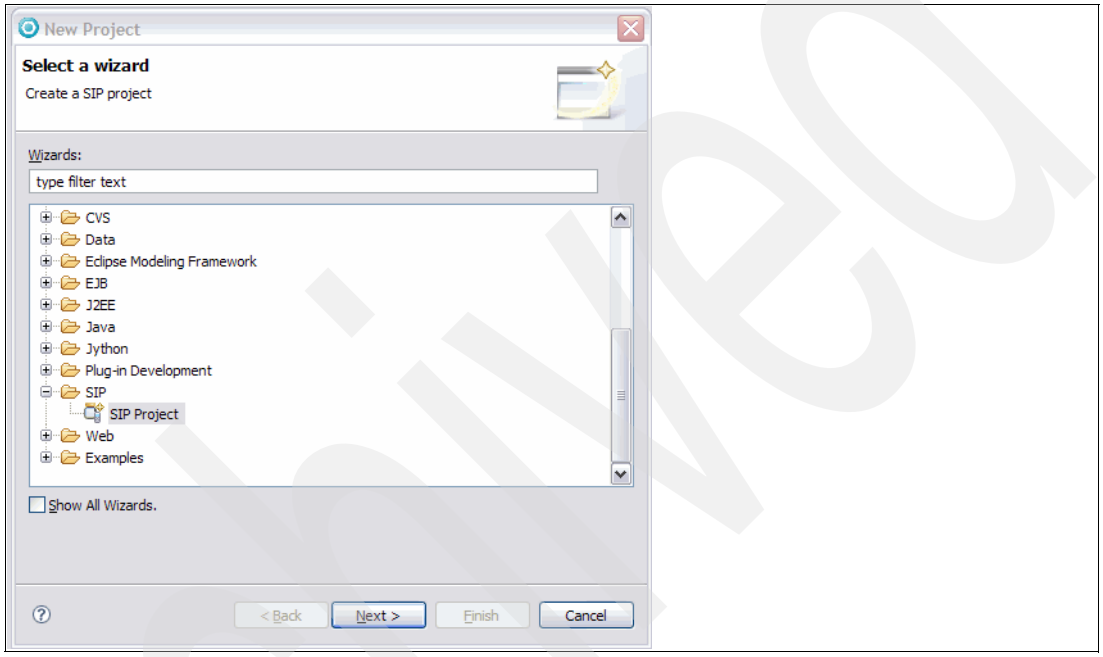


Figure 8-9 Select SIP project

3. In the New SIP/HTTP Project window (Figure 8-10 on page 267), enter PX21_PRS_SPLR_IMS as the name of the SIP project. Leave the other options as their defaults and click **Finish**. Note that the wizard allows you to select several Web features in the subsequent windows that are not required for this project.

Archived

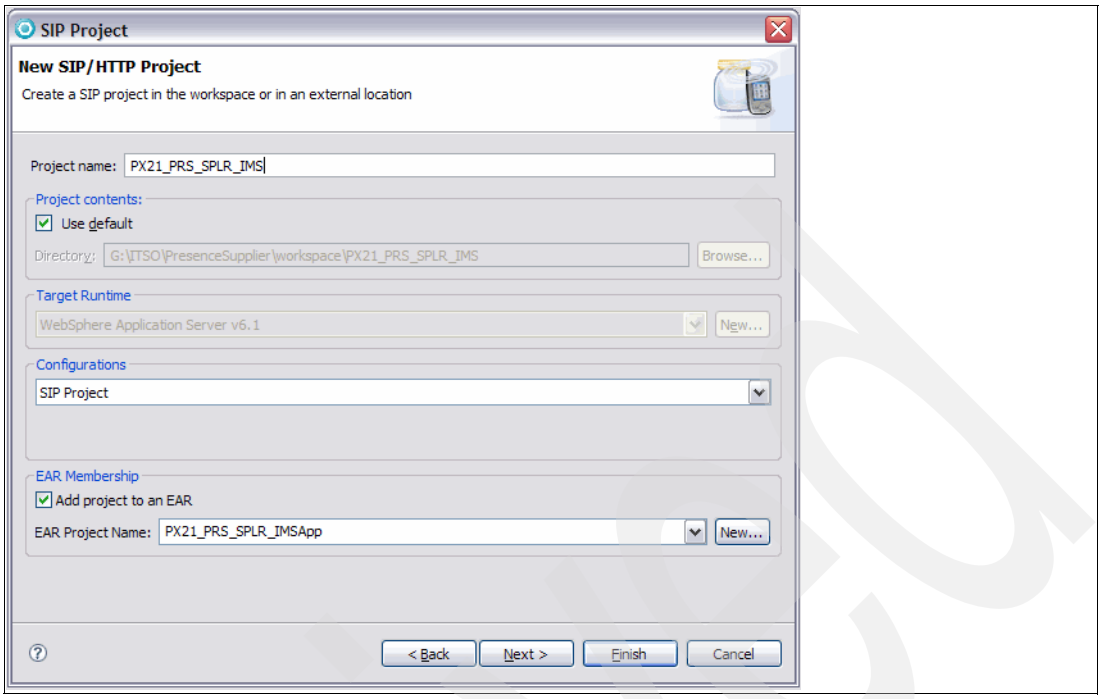


Figure 8-10 Naming the SIP project

4. Rational Application Developer creates a new SIP project and offers to open an associated Web perspective. Because this service implementation does not have to deal with a Web UI, click **No** in the Open Associated Perspective pop-up menu.

Note: By default, RAD creates a WebDiagram.gph file to enable developers to easily create Web page flows and so on. This service implementation does not need this feature, so close the file and continue.

Archive

5. This SIP project should have access to most of the libraries that were imported as part of the J2EE Application PX21_PRS_SPLR_IMSApp. The libraries can be found under the Utility JARs section of the enterprise application deployment descriptor. In order to point the SIP project to Utility JARs, select the **SIP Project**, right-click it, and select **Properties**. The SIP Project properties dialog is displayed. Select the **J2EE Module Dependencies** from the left pane of the properties dialog. In the right pane, all the Utility libraries have check boxes, as shown in Figure 8-11.

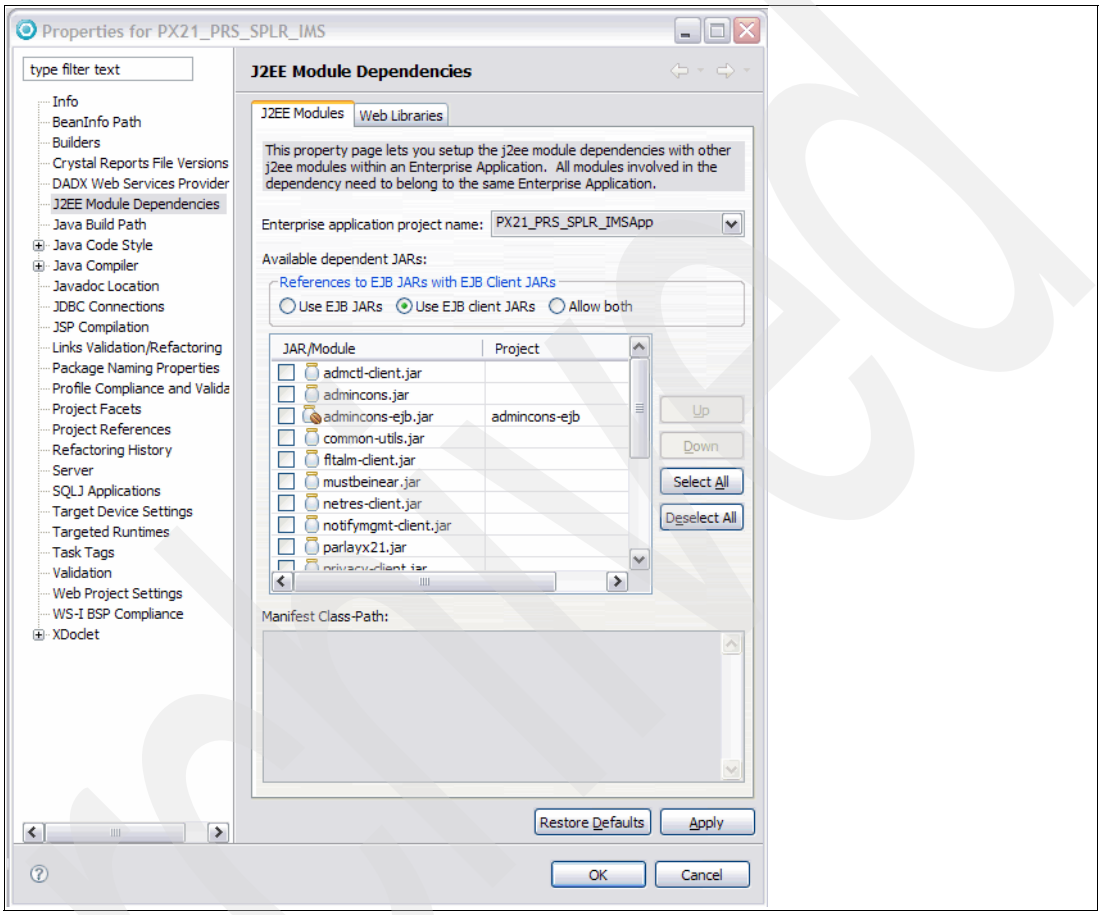


Figure 8-11 SIP Project properties dialog

6. Select the check boxes of all the JAR files in the list and click **OK**. This action will set the SIP project with all the essential IBM WebSphere Telecommunications Web Services Server libraries required for developing a service implementation.

Import the Parlay X WSDLs

After creating a SIP Project, we will now discuss how to import the required set of Parlay X Presence WSDL and Schema definition files from the IBM WebSphere Telecommunications Web Services Server Install location:

1. In the Project Explorer, expand the SIP project **PX21_PRS_SPLR_IMS**. Expand the **WebContent** folder. Select the **WEB-INF** folder, right-click it, and select **New** → **Folder**. The New Folder wizard is displayed (Figure 8-12 on page 269). Enter wsdl in the Folder name text box and click **Finish**.

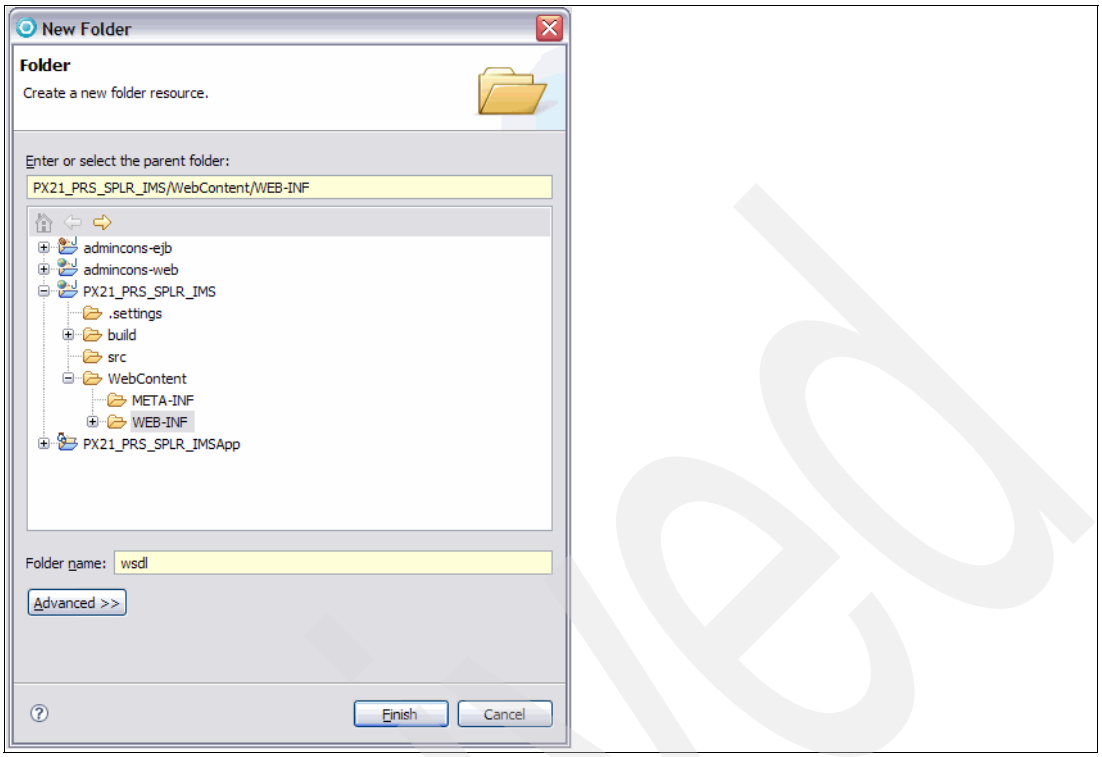


Figure 8-12 Create folder with the name wsdl

2. Select the **wsdl** folder in the Project Explorer, right-click it, and select **Import...**. The Import wizard is displayed (Figure 8-13). Expand **General** and select **File System**. Click **Next.**

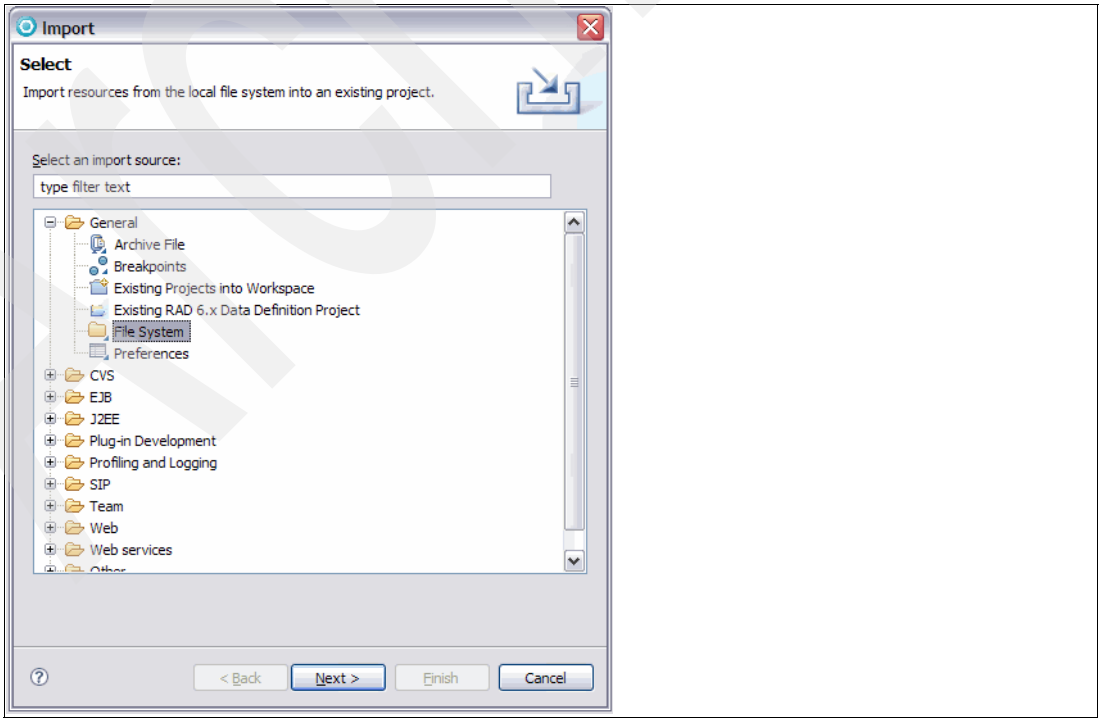


Figure 8-13 Select File System in the Import wizard

3. Select the directory in the File system window. The directory is the location where parlayx21wsdl.jar is extracted. Expand the directories, as shown in Figure 8-14.

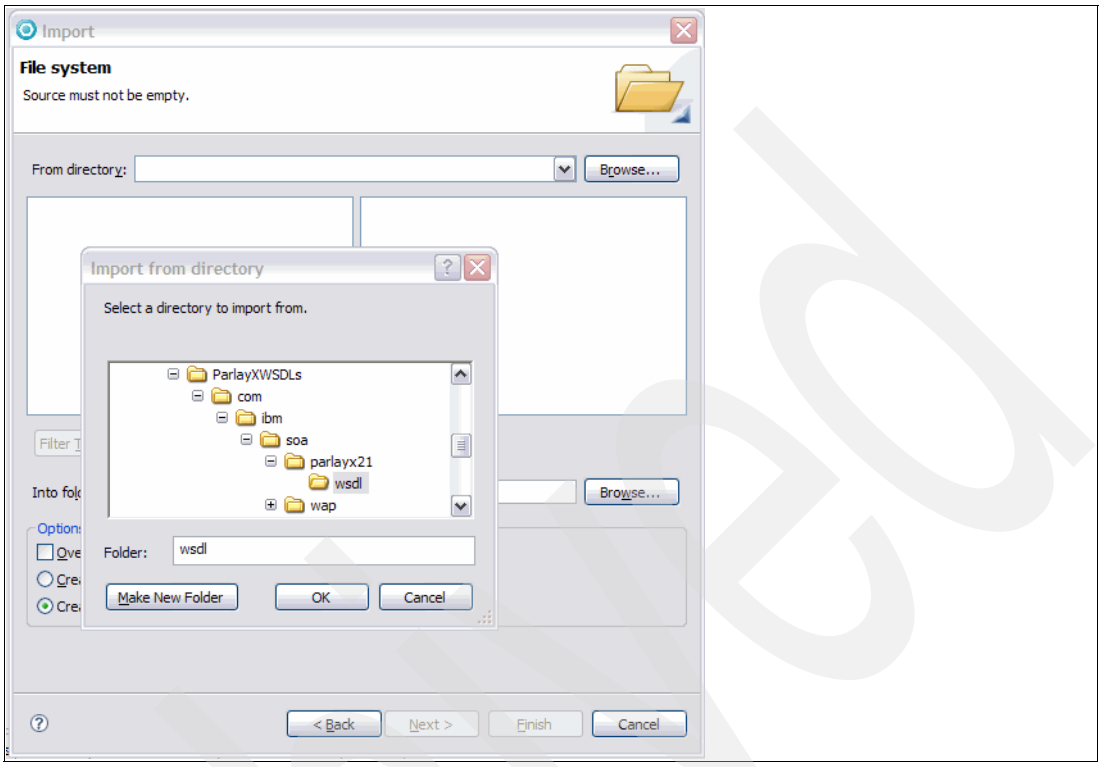


Figure 8-14 Traversing to the Parlay X WSDL location

4. Locate and select the following files from the directory and click **Finish**.
   - ParlayX_2_1.properties
   - px_cmn_f_2_0.wsdl
   - px_cmn_t_2_1.xsd
   - px_prs_si_2_3.wsdl
   - px_prs_ss_2_3.wsdl
   - px_prs_t_2_3.xsd

8.4.2 Generating Web service bindings

The Web service bindings are the entry points to service implementation logic. Using Rational Application Developer, two types of JAX-RPC bindings can be generated:

- Java Bindings
- EJB Bindings

The preferred binding for IBM WebSphere Telecommunications Web Services Server service implementations are JAX-RPC Java bean bindings. The subsequent development steps discuss the generation of the JAX-RPC Java bean bindings. The rationale behind choosing Java Bindings is:

- JAX-RPC Java beans are comparatively lighter in weight than EJB bindings. EJB bindings involve transaction manager and other managed processes, which add up to more processing.

- JAX-RPC Java beans have been easier to tune for high performance in practice because they use servlets.
- JAX-RPC Java beans simplify the construction of converged SIP applications, as the SIP programming model is also based on Servlets.
- Generated JAX-RPC Java beans can implement the javax.xml.rpc.server.ServiceLifecycle interface in order to access the ServletContext and gain access to servlet resources.

Note: Access to the ServletContext is not thread safe. Synchronization must be used when writing and reading data from multiple threads. Where possible, a HTTP session should be used to avoid the impact of synchronization.

EJB bindings do have some usage, for example, in services where each Web service call corresponds to a transaction or where the service is primarily dealing with an Entity bean and is essentially processing a one-way request.

Note: As a best practice, run the WSDL validation before attempting to generate wEb service bindings.

Having prepared the project environment with the prerequisite files, we will now see how to generate Web service bindings. The px_prs_ss_2_3.wsdl file is used to generate Web service binding, and it comprises service, port, and address definitions.

Archived

Follow these steps to generate service bindings:

1. Right-click **px_prs_ss_2_3.wsdl** and select **Web Services** → **Generate Java bean skeleton**. The Web Services wizard is displayed as shown in Figure 8-15.

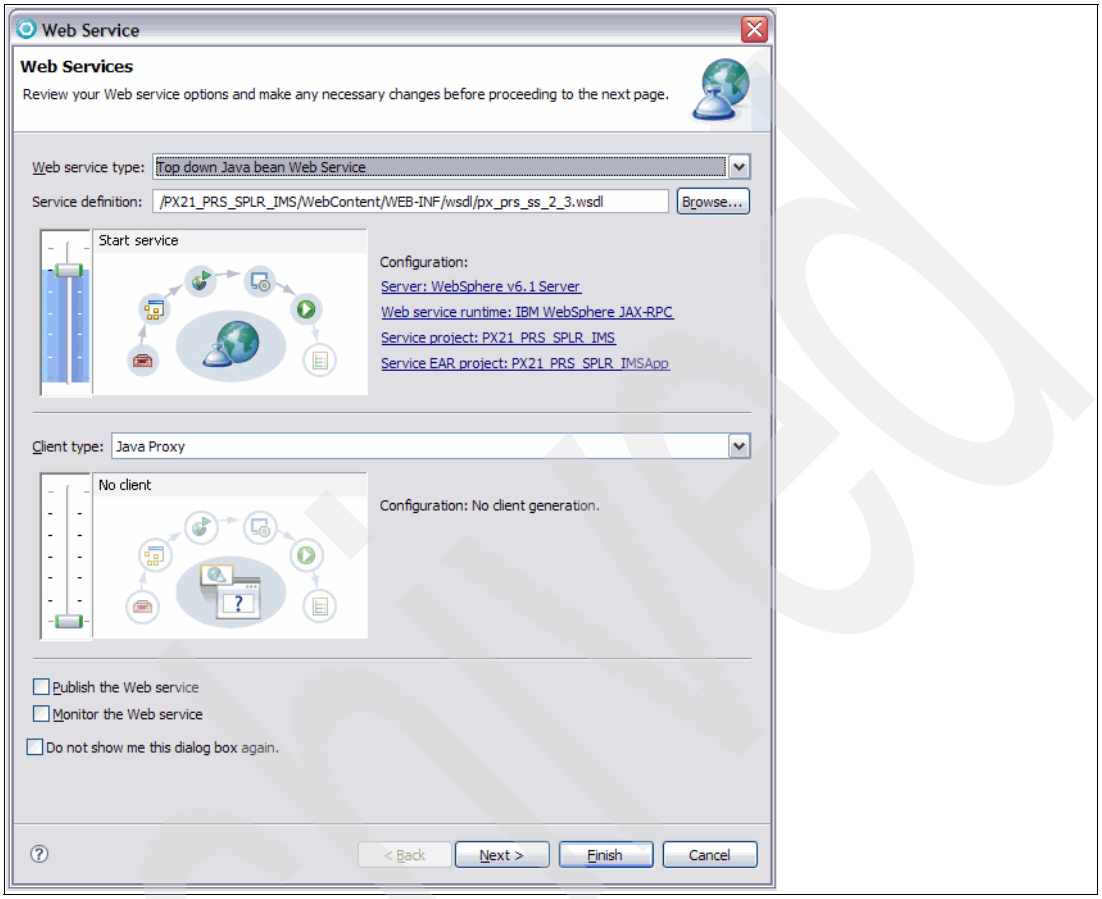


Figure 8-15 Web Services bindings options

Archive

2. In the Web Services window, leave the default settings as is and click **Next**. The Web Service Skeleton Java Bean Configuration window is displayed (Figure 8-16). Check the **Define custom mapping for namespace to package** check box and leave the other settings as is. Click **Next**.

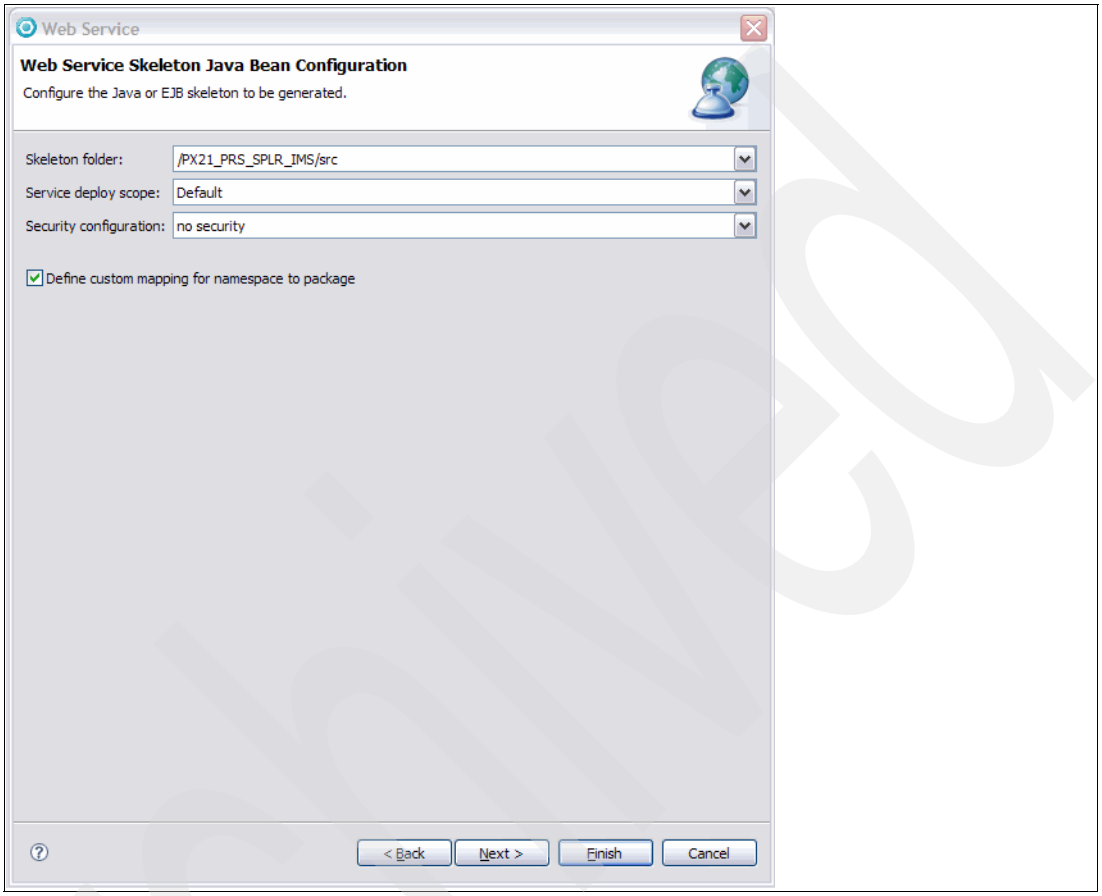


Figure 8-16 Web Service options window

Archive

3. The Web Service Skeleton namespace to package mapping window is displayed (Figure 8-17). In accordance with IBM WebSphere Telecommunications Web Services Server design and development guidelines, we will make a few changes to the package names here. Click **Import...**. The Browse Files dialog box is displayed. Select the **PX21_PRS_SPLR_IMS** SIP project. Expand the folders to the wsdl folder. Select the **ParlayX_2_1.properties** file. Click **OK**.

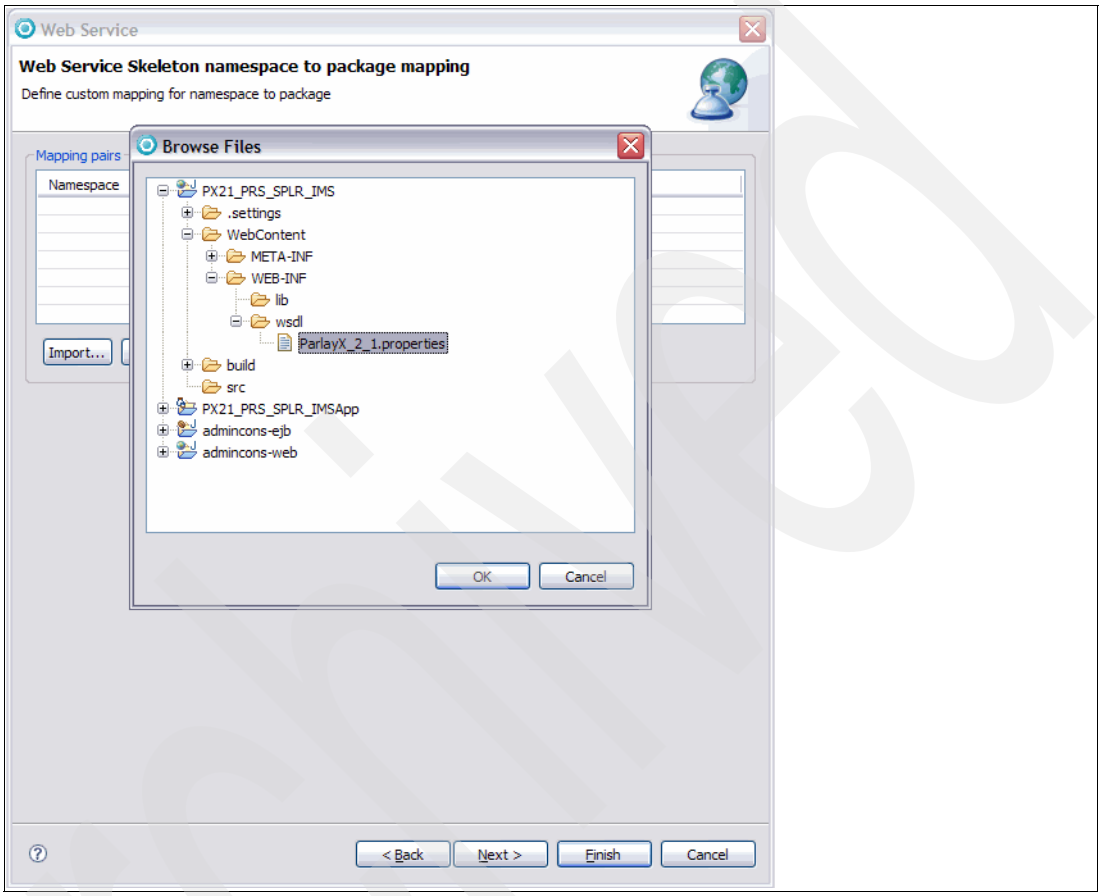


Figure 8-17 Web service Namespace to Package mapping selection

4. The Web service Skeleton namespace to package mapping window displays the package name mappings for all the Parlay X namespaces (Figure 8-18 on page 275). Click **Next**. At this point, RAD takes a few minutes to generate the Java files for the service and the schema definitions following the JSR 109 specification (also known as Web Services for J2EE 1.0). After generating the code, the wizard prompts you to start the embedded WebSphere Application Server V6.1 instance. Do not click the **Start server** button; click the **Cancel** button instead.

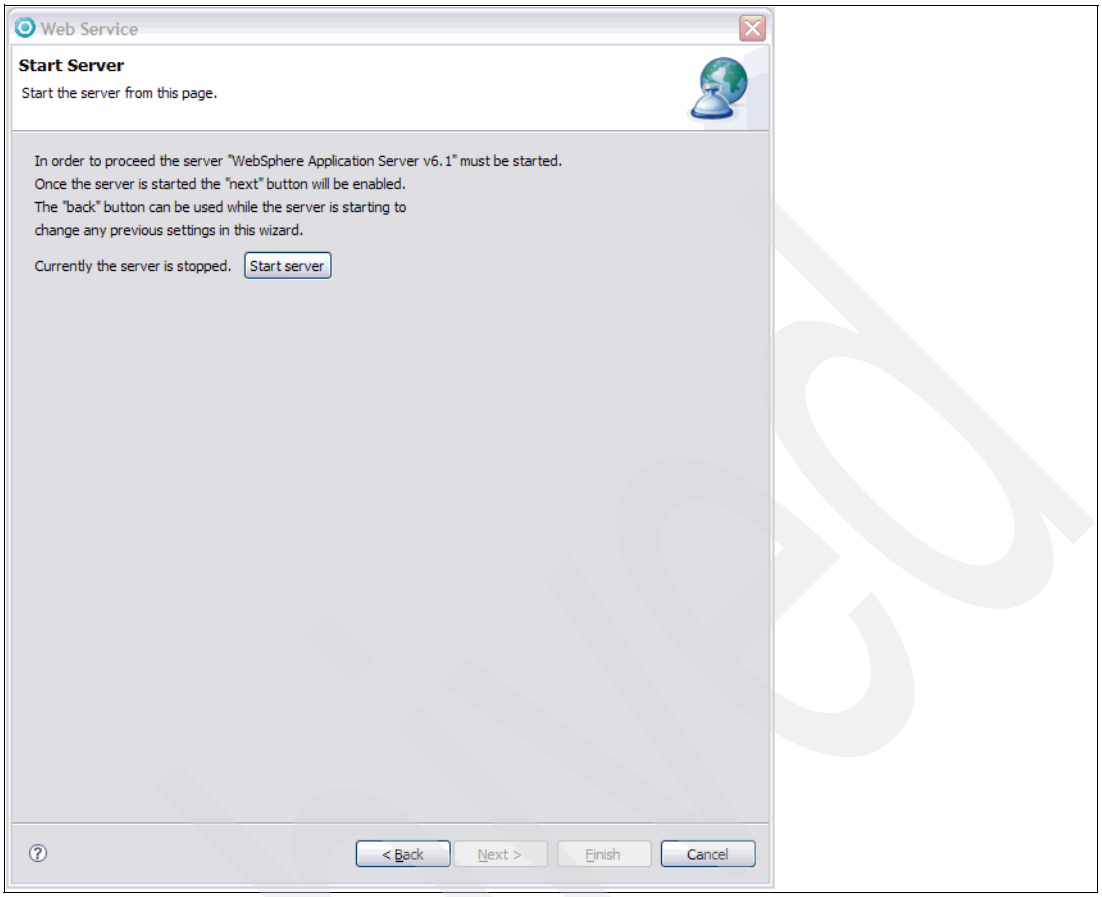


Figure 8-18 Web service generation wizard prompting you to start the server

Note: The code generators of Rational Application Developer fail to generate certain variable names in the Java classes for Schema definitions. As a result, you would find errors reported in the SIP project. These errors are because of the usage of enum as a variable name in certain Java classes. Since enum is a reserved word in JDK™ V1.5, the compiler reports errors for all the occurrences in the .java files.

For now, the above problem can be resolved by renaming the occurrences of enum to enumerator. Follow the steps below to rectify the problem.

5. Expand the SIP project, expand to **Java Resources: src**, and then expand the **com.ibm.soa.parlayx21.presence** package. The files ActivityValue.java, CommunicationMeansType.java, PlaceValue.java, PresenceAttributeType.java, PrivacyValue.java, and SphereValue.java report code errors (Figure 8-19).

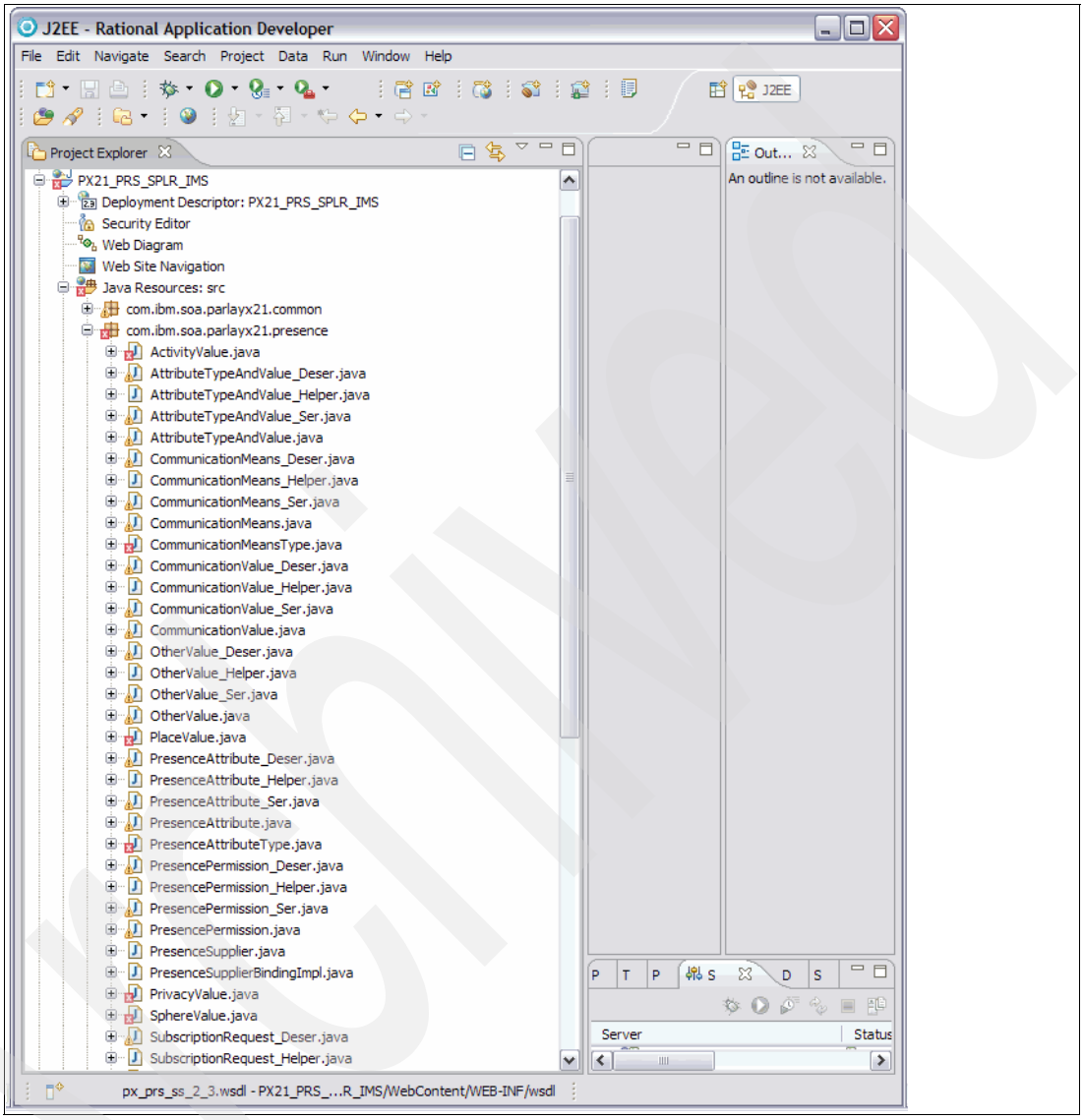


Figure 8-19 Code generation errors during the creation of Web Services bindings

6. As mentioned earlier, the reason for these errors is using enum as a variable name in some of the .java files. Open each .java file that reports these errors and replace the variable name enum with enumerator.

Note: Rational Application Developer has the auto compilation feature. Soon after you replace the occurrences of enum with enumerator in all the .java files, you should not see any errors in the IDE.

7. Check the bindings code generated by RAD. This should comprise all the service operation method signatures. Expand the package **com.ibm.soa.parlayx21.presence** and locate the file PresenceSupplierBindingImpl.java. Double-click the file to open it. Notice that all the methods have empty bodies.

This concludes the Web Services bindings generation process.

8.4.3 Guidelines for IBM WebSphere Telecommunications Web Services Server service implementations

In this section, we will discuss the IBM WebSphere Telecommunications Web Services Server coding guidelines that should be followed for a service implementation. Wherever possible, we will demonstrate the implementation by taking the Parlay X Presence Supplier implementation as a reference.

Developing the basic utilities

Before delving into the actual service implementation code for each service operation, let us discuss the basic utilities required by a service implementation. Following the IBM WebSphere Telecommunications Web Services Server design guidelines, the utilities, such as logging and tracing and message bundles, can be created first. Here are some of the salient coding practices for custom service implementation.

Logging and tracing utilities

- We recommend using java.util.logging.Logger as the basis for logging and tracing operational messages in the service implementations. The run time of IBM WebSphere Telecommunications Web Services Server uses utility classes that leverage this Java logging mechanism. We recommend that any service implementation that is intended to be deployed in the IBM WebSphere Telecommunications Web Services Server environment adheres to the Java logging, as this will help debug the code in a consistent manner.
- Typically, utility classes are created to handle logging and tracing to WebSphere Application Server First Failure Data Capture (FFDC). A sample of the class can be found in the sample code provided with this book. Refer to the com.ibm.twss.parlayx21.presence.supplier.utils.TraceLogger Java class (refer to Appendix E, “Additional material” on page 399 to discover how to access this file).
- Much of the content that is logged requires appropriate messages and message codes that provide contextual information in the logs and traces. Following the IBM WebSphere Telecommunications Web Services Server design guidelines for message codes and message bundle conventions, create .properties files for the default locale, which is en_US, as well as the base .properties file. The message codes should start and end in the Available range as per the design guidelines in “Message identifiers” on page 234.

Data access utilities

Many Parlay X Web service implementations require persistence of state as part of the regular service operation for failover purposes. A caveat is that Parlay X Web service implementations should always try to maintain as much state in-memory as possible in order to improve performance. Parlay X services can choose to use JDBC for SQL access to data sources or entity beans to access database records. To enhance performance, Parlay X Web Services implementations should always attempt to prepare statements prior to execution, thus allowing query caching for optimized data retrieval. If an application uses many different types of queries, increasing the WebSphere Application Server statement cache can yield a significant performance improvement.

For performance reasons, only local interface access to entity beans should be used. In addition, in most cases, the scope of the EJB transaction is only database write; by default, WebSphere Application Server assumes that a thread of execution participates in a global transaction, which might interfere with invocations to Telecom Web service implementations

that execute on the same thread. The solution is to modify the EJB deployment descriptors to use the local transaction feature of WebSphere Application Server.

For information about the exact configuration steps, refer to:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ae/ae/cjta_loctran.html

Customizing the Web service bindings

Customizing a default binding implementation generated by RAD is required in the case of the Parlay X Presence Supplier service. In this section, we will make couple of changes to the generated code.

- Move the Presence supplier service binding to a new package named com.ibm.twss.parlayx21.presence.supplier.impl. This is an optional modification.
- Since the service operations should send and receive SIP messages, the binding should be able to communicate and share session information with SIP servlets. To enable this functionality, the binding implementation generated by RAD needs to be modified. As described in the 8.4.2, “Generating Web service bindings” on page 270, the Binding should implement the javax.xml.rpc.server.ServiceLifecycle interface. This change allows the binding code to access the underlying ServletContext object at run time.

Figure 8-20 shows the source code of PresenceSupplierBindingImpl after making these changes. The two methods init and destroy should be overridden by the binding class. The init method gives access to the ServletContext object of this binding’s underlying servlet.

Figure 8-20 Web service binding implements ServiceLifecycle

Note: Refer to the PresenceSupplierBindingImpl.java code that has been made available along with this IBM Redbooks publication.

Accessing the ServicePlatform API

As mentioned in 8.3.2, “Service Platform Application Template” on page 256, we discuss how to use the ServicePlatform API. The underpinnings of the ServicePlatform API are a set of utility classes and a JAX-RPC handler that wrap object references to various useful components of IBM WebSphere Telecommunications Web Services Server. So in order to use the ServicePlatform API, the JAX-RPC handler with the name com.ibm.twss.platform.ServicePlatformHandler must be registered in the Web Services

deployment descriptor, which is named webservices.xml. The exact steps to make this configuration are given in Figure 8-21 on page 279.

1. Expand the SIP project. Then expand the **WebContent** → **WEB-INF** folder. Locate the webservices.xml. Double-click it to open it.

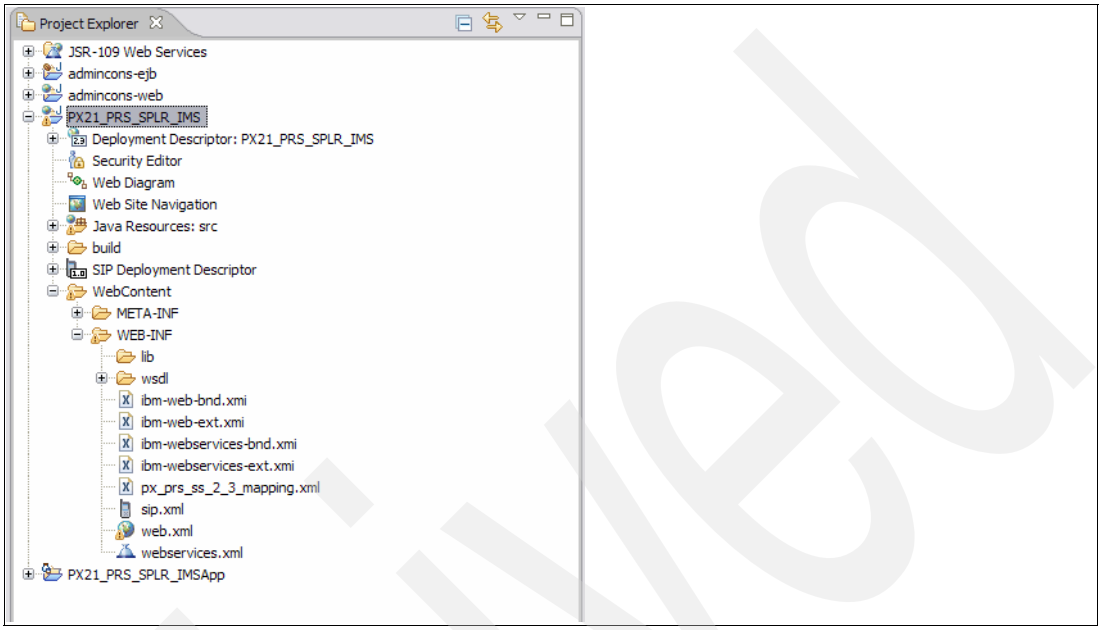


Figure 8-21 webservice.xml location in Project Explorer

Archived

2. In the Web Services Editor, locate and click the **Handlers** tab at the bottom of the Web services editor pane. Go to the Handlers section (Figure 8-22).

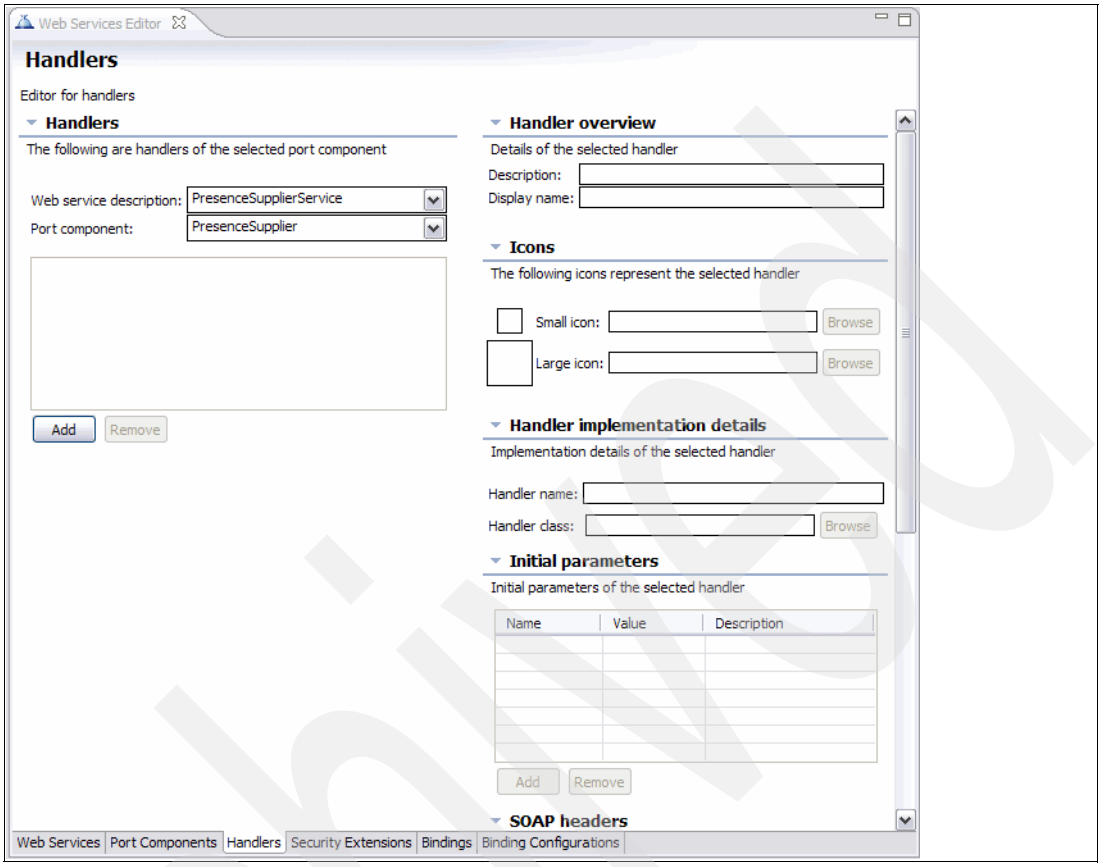


Figure 8-22 Web service handlers section in webservices.xml

Archive

3. In the Handlers section, click **Add**. This action opens up the Class Browser. Type the class name com.ibm.twss.platform.ServicePlatformHandler into the text box provided. If the CLASSPATH is configured properly, as per the directions in the J2EE Dependency Modules setup (illustrated in 8.3.3, “Creating a custom service project” on page 257), you should be able to find the class name and Java package in the Matching Types list. Select the **ServicePlatformHandler** from the list and click **OK**.

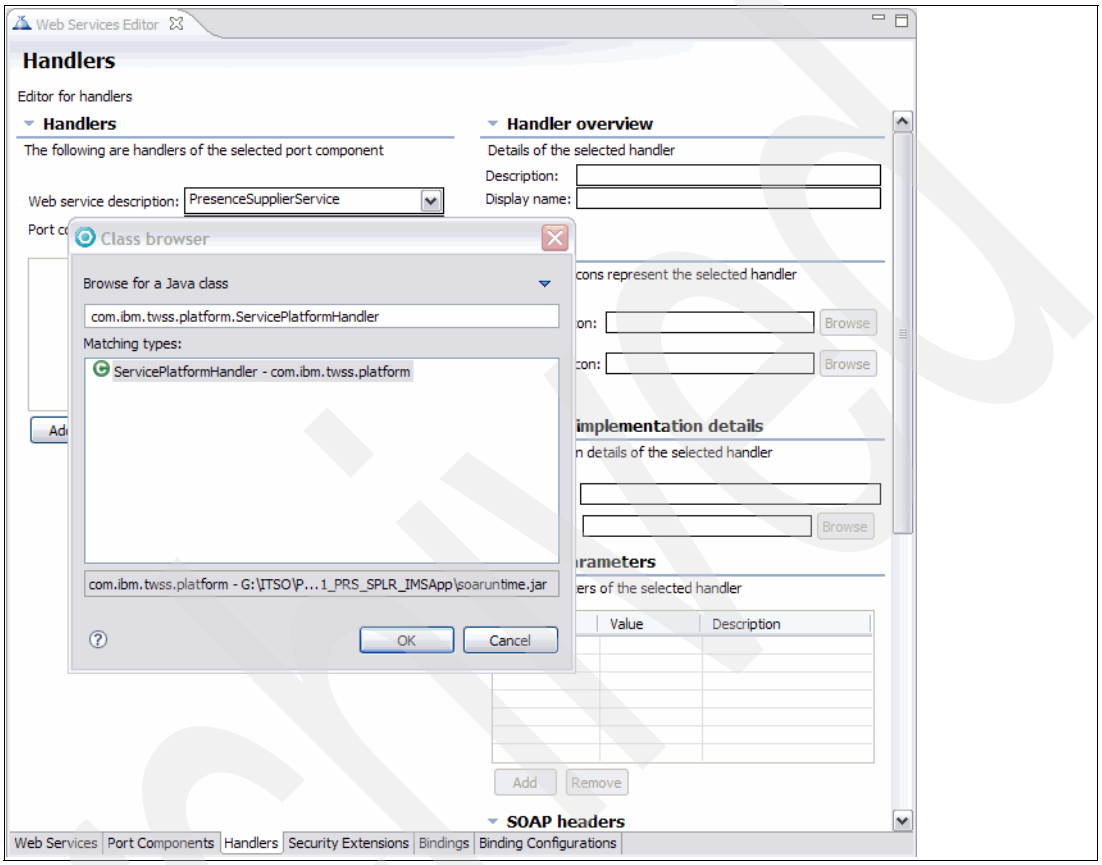


Figure 8-23 ServicePlatformHandler selection

4. Save the changes to Web Services Editor. Select **File** → **Save**. If the package and the Handler class name is not displayed in the Handlers section, then close the editor and re-open it.

Archive

5. Go to Handlers section (Figure 8-24). Select **com.ibm.twss.platform.ServicePlatformHandler**, and in the right side of the window, optionally enter a display name for the handler; we enter ServicePlatformHandler. Save the changes to Web Services Editor by selecting **File** → **Save**.

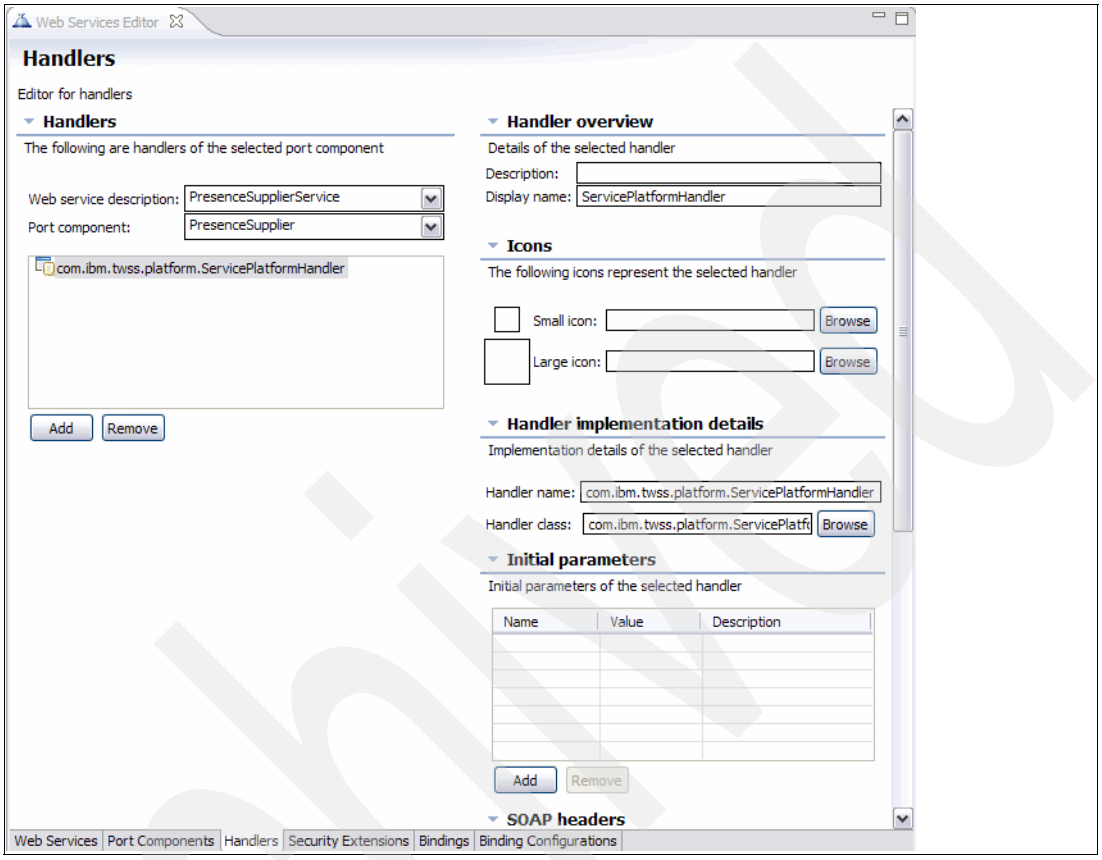


Figure 8-24 Configuration of ServicePlatformHandler

6. By configuring the ServicePlatformHandler, the service implementation can now have access to the following request context information:
   - Transaction Identifier
   - Requester Identifier
   - Service Identifier
   - Service Operation Identifier
   - Exceptions encountered during processing of ServicePlatformHandler
   - Policies that are attached to the request as a result of processing in the Access Gateway Flow and mediations
   - Requester's Locale information
   - Resolved Group members, if any
   - The request URL

   The Service Platform API methods for accessing the request context information can only access this information during the processing of an incoming request from Access Gateway. If an asynchronous thread is executing, either as a result of a message driven bean or due to a response received from a network element, the information from the original request is not available. If the service knows that this request information is

desired during later execution, it should retrieve the information while it has access to it, and store or cache it in the appropriate data structures as necessary for later access.

7. Utilizing the features of ServicePlatform API, we will see how to access the client interfaces of the IBM WebSphere Telecommunications Web Services Server common component services. Open the PresenceSupplierBindingImpl.java in the SIP project. Go the method “publish” and add the code to access the common component client interfaces, as shown in Figure 8-25.

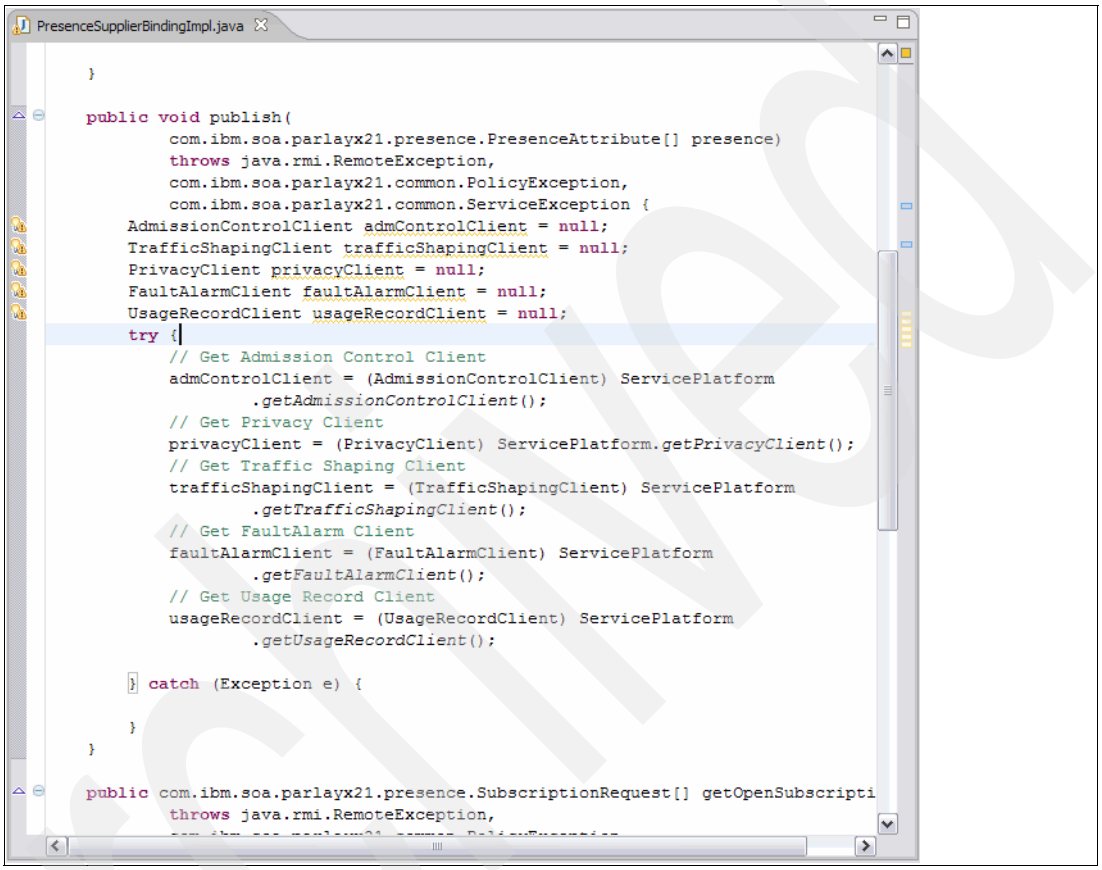


Figure 8-25 Using ServicePlatform API to access common component client interfaces

So far, we have seen the basic configuration required to set up the development environment and the utilities given by IBM WebSphere Telecommunications Web Services Server for custom service implementation development. In the next section, we will discuss the invocation sequence of common components from the service implementation code.

8.4.4 IBM WebSphere Telecommunications Web Services Server common component invocations

The ServicePlatform API provides access to the client interfaces of all common components of IBM WebSphere Telecommunications Web Services Server. The usage of each individual common component depends on the type of service being implemented. Certain common components are only used depending on the nature of request processing required in the service implementation, such as PXNotify and Notification Management common components. For more information about common components, refer to Chapter 6, “Common components” on page 203. In our sample Parlay X Presence Supplier service

implementation, we will implement the common component invocation sequence shown in the following sections.

Note: In order for the Service implementation to access and leverage the features of Common components such as Admission Control and Traffic Shaping, certain configurations are mandatory. For more information about the mandatory configuration settings for services, refer to 8.6, “Configurations for a new service” on page 291.

Admission Control Client

The Admission Control client will be the first common component to be called in the service operation logic of the service binding. Refer to 6.2.1, “Admission Control component Web service” on page 205 for more information about available admission control client operations, the request, the response, and faults. The code snippet shown in Example 6-2 on page 215 is provided for your reference.

Privacy Client

Typically, the next common component that is invoked in the sequence is the Privacy Client. Privacy integration is required to check requester’s authorization to access the information of a named party that is already registered or accessible in the network. In most cases, the identifier of a party in the IMS network is typically a SIP or TEL URI, such as sip:user1@example.com or sip:11122233344@example.com or tel:11122233344. Privacy service implementation is not provided out-of-box with IBM WebSphere Telecommunications Web Services Server. Therefore, Privacy Client invocation can be treated as optional functionality. However, if a privacy implementation is provided, we want the Parlay X Presence Supplier service to make a call to Privacy service and perform an authorization check. A code snippet of Privacy Client invocation is provided in Example 8-3.

Example 8-3 Privacy Client code sample

```
            // Create Request object
            VerifyPrivacyPermitsRequest privacyRequest = new
  VerifyPrivacyPermitsRequest();
            // Set Current Application Name
            privacyRequest.setService(publishReq.getApplicationName());
            // Set Current Operation Name
            privacyRequest.setOperation(publishReq.getOperation());
            // Set Transaction identifier
            privacyRequest.setGlobalTransactionID(publishReq
                  .getGlobalTransactionId());
            // Set Requester identifier
            privacyRequest.setRequester(publishReq.getRequesterId());
            // Set Role identifier
            privacyRequest.setActingOnBehalfOf(publishReq.getRequesterId());
            // Get Target Parties from Request
            String[] requestTargets = publishReq.getTargets();
            // Set the targets / party addresses
            privacyRequest.setTargets(requestTargets);
            // Privacy service call
            boolean privacyResults[] = privacyClient
                  .verifyPrivacyPermits(privacyRequest);
```

Traffic Shaping Client

After Admission Control Client invocation, the Traffic Shaping Client should be invoked. The Traffic Shaping service depends on another common component, the Network Resources.

Typically, the service implementations eventually communicate with at least one network element. The Network Resources common component allows you to define a network resource mapping or alias for a specific network element and the set of attributes that affect the network element's processing capabilities. In the case of Parlay X Presence Supplier, the Traffic Shaping service essentially should control the outflow of requests to a particular network resource, so that the total number of requests do not exceed a predefined limit at any given point of time. The limits designated for a specific network resource can be configured in the IBM WebSphere Telecommunications Web Services Server Administration Console. Refer to 8.6.1, "Admission Control configuration settings" on page 291 for more details.

The Traffic Shaping service operation that should be invoked is verifyResourceCapacity. In Example 8-4, you will find the corresponding Request and Response objects. The two significant method calls on VerifyResourceCapacityRequest are:

- setResource(): This method takes the Network Resource Name that is intended to be used by this service. Typically, the network resource name or the alias for a network element with which the service interacts is defined as part of the Service's Java Management Extensions (JMX) MBean attribute. This technique allows for the configuration of a network resource name from the IBM WebSphere Telecommunications Web Services Server Administration Console.
- setRequestCapacity(): This method takes an int type as its argument. The total number of atomic or indivisible interactions between a service and a network element for a given service operation should be specified here. In the case of the Parlay X Presence Supplier service, for the publish operation, the interaction with presence server is a single SIP PUBLISH. Since the publish is a one-way service operation, the requested capacity for a single service operation invocation is 1.

For your reference, a code snippet of the Traffic Shaping client is provided in Example 8-4.

Example 8-4 Traffic Shaping client code sample

```
        PublishPresenceServiceRequest publishReq =
(PublishPresenceServiceRequest) req;
        // Create Request Object
        VerifyResourceCapacityRequest capacityRequest = new
VerifyResourceCapacityRequest();
        // Set Current Application name
        capacityRequest.setService(publishReq.getApplicationName());
        // Set Current Operation name
        capacityRequest.setOperation(publishReq.getOperation());
        // Set Transaction identifier

capacityRequest.setGlobalTransactionID(publishReq.getGlobalTransactionId());
          // Get the Presence Resource Name from MBean Attributes
          capacityRequest

.setResource(PresenceSupplierAttributes.PRESENCE_SERVER_RESOURCE_NAME
                      .getName());// Only 1 SIP PUBLISH is sent to Presence
Server
          capacityRequest.setRequestCapacity(1);
          com.ibm.soa.sp.trafficsh.VerifyResourceCapacityResponse
capacityResponse = trafficShapingClient
                .verifyResourceCapacity(capacityRequest);
```

Note: Typically, in any service implementation, the core logic of the service operation is executed after invoking the Traffic Shaping client for each service operation. Refer to the core logic implementation for Parlay X Presence Supplier sample in 8.5, “Implementing the core logic of the service” on page 287.

Note: The Java Management Extensions (JMX) MBean is an essential component of a service implementation. The design guidelines of IBM WebSphere Telecommunications Web Services Server recommends the usage of JMX for managing the service implementation configurations. For more information, refer to 8.7.1, “Developing the JMX MBean for service” on page 300.

Usage Record Client

The Usage Record common component stores the usage information for all the IBM WebSphere Telecommunications Web Services Server services that are available for external applications to invoke. The usage information is recorded specifically for each invocation of the service operation by a client application. A usage record is primarily recorded after successful execution of the core service logic. Also, your design may require recording usage in the event of failures as well, since the service and infrastructure spent time and resources in servicing the request.

Therefore, during the development phase, it is important to invoke the Usage Records client at significant points in the code based on the design decisions of the service. The *status code* of a usage record identifies one of the three possible outcomes that can be recorded after the execution of the core service logic. The status codes are:

- 0: The core service logic execution is successful.
- 1: A general failure is encountered anywhere in the service operation logic. These failures might include failures from common component invocations also.
- 2: A failure is encountered during the privacy component invocation.

As a reference, a code snippet of the Usage Record client is provided in Example 8-5.

Example 8-5 Usage Record client code sample

```
            // Create Request Object
            WriteUsageRecordRequest usageRecordReq = new
  WriteUsageRecordRequest();
            // Set Current Application name
            usageRecordReq.setService(publishReq.getApplicationName());
            // Set Current Operation name
            usageRecordReq.setEventType(publishReq.getOperation());
            // Set Transaction identifier
            usageRecordReq.setGlobalTransactionID(publishReq
                  .getGlobalTransactionId());
            // Set Current time
            usageRecordReq.setRecordTime(System.currentTimeMillis());
            // Set Status code
            usageRecordReq.setCode(statusCode);
            // Set Policy Names
            usageRecordReq.setServiceAttributes(namesArray);
            // Set Policy Values
            usageRecordReq.setServiceValues(valuesArray);
            WriteUsageRecordResponse usageRecordRes = usageRecordClient
```

```
.writeUsageRecord(usageRecordReq);
```

Fault and Alarm Client

As part of the service logic, the Fault Alarm common component must invoked to record erroneous situations that might occur during the execution of service logic. The Fault and Alarm component emits a JMX event by default. Additionally, the Fault and Alarm component can be enabled to emit a Common Base Event (CBE) by utilizing the Common Event Infrastructure (CEI). For more information about design related guidelines as well as coding guidelines, refer to 6.6.4, “Invoking IBM WebSphere Telecommunications Web Services Server Fault and Alarm common components” on page 218.

In this section, we discuss the common component client invocations that are required for Parlay X Presence Supplier sample service. In the next section, we discuss the core service logic that communicates with Presence Server to accomplish the publish service operation functionality.

8.5 Implementing the core logic of the service

The core of service logic has a dependency on the network protocol and infrastructure that the service interacts with. Typically, service logic comprises at least four activities:

- Construction of a request in a protocol that is specific to a back-end network element.
- Transmission of the request to the network element.
- Reception of responses, if any, from the network element.
- Maintaining the appropriate state data in a database (if necessary) for the service to support all operations. Not all services have state data that needs to be shared on a cluster, or between threads.

Response handling

As with any Web service implementations, most of the service operations in Parlay X service interfaces follow four patterns.

Request - reply pattern

The most common pattern followed by most of the operations of Parlay X Service Interfaces. The service operation execution is synchronous in nature. Typically, for every request sent by the service operation logic, the network element responds within a finite amount of time. If there is an exception, depending on the protocol, the appropriate error information is sent by the network element.

Request - only pattern

Few Parlay X service operations demonstrate this pattern. The service operations that service requests do not guarantee a successful execution completion. A request is sent by the service operation logic to a network element on a best effort basis. From a client application perspective, the service invocation does not yield for any response, for example, the Parlay X Presence Supplier publish operation. Some operations return “void”, but they return exceptions that are a valid response.

Archived

Response - only pattern

Few Parlay X service operations exhibit this pattern. Typically, in this kind of scenario, the required request context information to query a network element is already available to the service run time, or could be constructed through other standard mechanisms, such as, for example, the identity assertion mechanism, by which a party's credentials are obtained. The request context information thus established is used as the basis for sending a request message to network element. From a client perspective, the service invocation results in a response.

Request with asynchronous responses pattern

Some of the Parlay X service interfaces are comprised of service operations that follow this pattern. Asynchronous responses are delivered in Parlay X as callback requests that are synchronous requests. The service operation is responsible for constructing the request and transmitting to a network element. As part of the operation, the network element may send one or more responses that stretch over a period of time. The core service logic should basically be able to handle asynchronous events as well as handle synchronization of threads in some cases. "Synchronizing threads and handling asynchronous events" discusses the design aspects needed to meet this requirement. Finally, note that the semantics of a Parlay X API may different from the request semantics of the back-end network protocol, and in some cases some adaptation of protocols call sequences are necessary.

Synchronizing threads and handling asynchronous events

From the above patterns, we understood that few Parlay X service operations are defined to be synchronous. This means that the service operations execute the logic and return a result object. Few services are asynchronous and return only an identifier that allows a subsequent request to check on the status of the request submitted earlier or expect a callback when the task is completed.

There are cases where the service operation is synchronous in nature, but the interaction between the service core logic and the back-end network element may be asynchronous. This behavior is typical with services that involve protocols such as Parlay and SMPP. In these circumstances, the service should map a synchronous method operation onto an asynchronous thread, which is complicated within an application server environment like WebSphere Application Server.

This coordination of synchronous and asynchronous activities will require the synchronization of multiple threads of execution, and in order to avoid scalability problems or deadlock problems, it will be necessary to limit the time that one thread waits on the other incoming thread to a minimal duration. Additionally, sometimes the response from the network element might arrive even before the requester is ready to receive it, or perhaps even the response did not return at all. These boundary conditions must be considered when designing the service logic.

To coordinate the request and asynchronous responses, a unique request identifier will be required. The call flow for coordination between threads would require the operation to have logic similar to this chain:

1. Thread A will identify the appropriate request identifier, such as an assignmentId. Sometimes this identifier may not be available until the response is received with a unique correlation identifier.
2. Thread A will look up or create an instance of a result object indexed by the request identifier as soon as possible.
3. Thread A will make the asynchronous request to the back-end network element using the request identifier or somehow associate the request identifier to the request that is being sent to backend.

Note: The request identifier that is embedded in the request to the back-end network element is typically returned in the response. This serves as a correlation identifier for the service logic to continue with response processing.

4. Thread A will wait for a finite period of time for the asynchronous response to arrive and retrieve the result data and release the result object.
5. If a timeout occurs, Thread A will return the appropriate non-response error to the client application.
6. If a result is returned, Thread A will return the appropriate results to the application, and remove the result object.
7. Thread B will be used to process any incoming responses to the asynchronous request sent earlier.
8. Thread B will retrieve the request identifier from the response data.
9. Thread B will look up or create the result object indexed by the request identifier.
10. If the result object is created, either Thread A has not yet created the result object, or has already timed out and consequently the result object is removed by Thread A. The result object created by Thread B should also be removed or be scheduled to be removed in such case.
11. If the result object is found, Thread B will set the result and notify the waiting Thread A with the accumulated result.
12. Thread B returns. Essentially, the client application gets callback notification from IBM WebSphere Telecommunications Web Services Server run time.
13. Thread C will periodically check for older result objects and remove them.

Note:

- Although this approach works well for handling asynchronous events, planning for the appropriate capacity and sizing of the infrastructure to handle asynchronous events while handling synchronous requests is a must.
- Since the Web container request thread that is waiting for the asynchronous results negatively impact the scalability of the application server run time, for example, in cases where the number of requests that require asynchronous responses reach the number of Web container threads, it is necessary to increase the Web container thread pool size to a larger number to account for the scalability issue.

Presence Supplier publish operation

The core service logic of the publish service operation in the Parlay X Presence Supplier interface should send a SIP PUBLISH request to a Presence Server. As the publish is a one-way operation, a successful submission of SIP PUBLISH to the Presence server is considered the completion of the operation's execution. In 8.4.4, "IBM WebSphere Telecommunications Web Services Server common component invocations" on page 283, the service binding code for the publish method invokes the prerequisite common components prior to executing the SIP code.

The code snippet in Example 8-6 shows how the SIP servlet is being called to create a SIP PUBLISH request.

Example 8-6 Sample code to call a SIP Servlet from Service Binding

```
        // Invoke the core logic of service operation
        // call SIP Servlet
        PublishServlet pubServlet = PublishServlet
              .getInstance(endpointContext.getServletContext());
        PresencePublish presencePublish = new PresencePublish(req,
              sipFactory);
        SipServletRequest sipRequest = pubServlet.createInitialPublish(
              presencePublish, getSipApplicationSession());
        sipRequest.send();
```

The publish method in Service bindings code uses the Servlet Context object reference to access a specific SIP servlet that handles SIP PUBLISH logic. We recommend creating the required data structures to capture the request context that is made available through the Service Platform API and use the data structures for downstream processing. For reference, follow the sample in Example 8-7.

Example 8-7 Sample code for the SIP PUBLISH request

```
        // Create From URI & To URI
        SipURI fromURI = validateSipURI((SipURI) publishable.getPresentityURI(),
publishable);
        SipURI toURI = validateSipURI((SipURI) publishable.getPresentityURI(),
publishable);
        // Create SIP PUBLISH request
        sipRequest = sipFactory.createRequest(sas, "PUBLISH", fromURI,
              toURI);
        // Request URI is the Presence Server SIP URI
        SipURI reqURI = (SipURI) sipFactory.createURI(presenceServerURI);
        reqURI.setTransportParam("TCP");
        sipRequest.setRequestURI(reqURI);
        // Add 'presence' Event Header
        sipRequest = setEventHeader(publishable, sipRequest);
        // Add 'P-Asserted-Identity'
        sipRequest = addPAssertedIdentity(publishable, sipRequest);
        // Set SIP PUBLISH request Expiration
        sipRequest.setExpires(publishable.getPresenceTimeout());
        // Add content to SIP Request
        String content = publishable.getContent();
        xLogger.info("createInitialPublish", "PIDF Content: \n"+content);
        if (content != null) {
           int contentLength = content.length();
           sipRequest.setContentLength(contentLength);
           sipRequest.setContent(content, "application/pidf+xml");
        }
```

The publish method should invoke the Usage Records common component client after executing the SIP request handling logic. Typically, the Usage Records client is invoked in the case of a successful execution of the service logic as well as in some failure cases. However, the usage information recorded should be qualified by the appropriate status codes. Refer to “Usage Record Client” on page 286 for more information.

In this section, we discussed the service logic specific details as well as the invocation of various common components. In 8.6, “Configurations for a new service” on page 291, we will discuss the configuration requirements for the scenario sample, Parlay X Presence Supplier service.

8.6 Configurations for a new service

In IBM WebSphere Telecommunications Web Services Server, configuration settings play a significant role for any service that is deployed in order to realize a fully functional service. The functionality of common components, namely Admission Control, Traffic Shaping, and Network Resources, depend on the configuration settings created for a specific service. The following sections discuss these configurations in detail.

8.6.1 Admission Control configuration settings

Admission Control configuration specific to a service should have four properties defined at the service level as well as at the operation level. The properties are the Description, Service Weight, ServiceLocalRateLimit, and ServiceClusterRateLimit. The Admission Control component decides about allowing or disallowing a service operation call made by the client application based on the values configured for these four properties. For more information about the algorithm implemented by the Admission Control component, refer to the section “Administering the Admission Control Component Web service” in the IBM WebSphere Telecommunications Web Services Server InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

The configuration settings are persisted in a IBM WebSphere Telecommunications Web Services Server database in a table with the name CONFIGPROPERTIES. For the out-of-box services, IBM WebSphere Telecommunications Web Services Server provides Database Definition Language (DDL) files, which should be executed during the deployment process of each individual service. Following the same procedure, the DDL shown in Example 8-8 is created for Parlay X Presence Supplier service.

Example 8-8 Admission Control component configuration settings for Parlay X Presence Supplier service

```
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.Description','IMS-based
Parlay X Presence Supplier Web service');
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.ServiceWeight','1');
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.ServiceLocalRateLimit','1
000');
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.ServiceClusterRateLimit',
'5000');
```

The settings shown in Example 8-9 are for the publish service operation of the Parlay X Presence Supplier service.

Example 8-9 Admission Control component configuration settings for publish operation

```
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.Operations.publish.Descri
ption','Publish a presentity''s rich presence information');
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.Operations.publish.Operat
ionWeight','1');
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.Operations.publish.Operat
ionLocalRateLimit','1000');
INSERT INTO CONFIGPROPERTIES
VALUES('template.AdmissionControlMBean.PX21_PRS_SPLR_IMS.Operations.publish.Operat
ionClusterRateLimit','5000');
```

For the sample scenario, the functionality implemented is limited to publish service operation only. Depending on the number of operations that will be supported in a service, a similar set of configuration settings should be created for each service operation starting with the service-specific settings.

The configuration values set using the DDL files can be modified using the IBM WebSphere Telecommunications Web Services Server Administration console. After implementing the JMX MBeans for the Parlay X Presence Supplier service, we will discuss the configuration modifications in detail. Refer to 8.7, “Management provisions for new service” on page 299.

8.6.2 Traffic Shaping configuration settings

Traffic Shaping configuration settings are dependent on Network Resources common component configuration settings. The required set of properties for the Network Resources common component are Name, MaxAverageSustainedRate, and MaxBurstSize. Prior to the execution of a service operation call, the Traffic Shaping common component invokes the Network Resources Web service and employs an algorithm to decide if the calls to the network element are allowed or not. For more information about the processing and algorithm employed by Traffic Shaping common component, refer to the section “Administering the Traffic Shaping component Web service” in the IBM WebSphere Telecommunications Web Services Server InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

The configuration settings for the Traffic Shaping component specific to a service are persisted in a Database table with the name CONFIGPROPERTIES. The Database Definition Language (DDL) file for creating the configuration settings is shown in Example 8-10.

Example 8-10 Traffic Shaping component configuration settings for Parlay X Presence Supplier service

```
INSERT INTO CONFIGPROPERTIES VALUES
('template.NetworkResource.PX21_PRS_SPLR_IMS.Name','A SIP SIMPLE presence
server.');
INSERT INTO CONFIGPROPERTIES VALUES
('template.NetworkResource.PX21_PRS_SPLR_IMS.MaxAverageSustainedRate','1000');
INSERT INTO CONFIGPROPERTIES VALUES
('template.NetworkResource.PX21_PRS_SPLR_IMS.MaxBurstSize','1000');
```

The configuration values set using the DDL files can be modified using IBM WebSphere Telecommunications Web Services Server Administration console.

8.6.3 Initial service policy settings

Similar to the configuration settings for a IBM WebSphere Telecommunications Web Services Server service, the initial service policy settings enable the utilization of a service. Hence, creation of the initial service policies is an important activity during the deployment phase of a IBM WebSphere Telecommunications Web Services Server service. References about the initial service policy creation can be found in the sections “Initializing Policies” and “Reference: default Policies for the Access Gateway and Web service implementations” in the IBM WebSphere Telecommunications Web Services Server InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

Service policies and IBM WebSphere Telecommunications Web Services Server admin console

The Policy attributes of a IBM WebSphere Telecommunications Web Services Server service are persisted in the IBM WebSphere Telecommunications Web Services Server subscription management database tables. Similar to the out-of-box services, the Parlay X Presence Supplier service should have an associated initial set of policies. The standard way to create these initial policies are to compose Jython scripts and execute the script using the WebSphere Application Server command-line administration tool wsadmin.sh. In the following sections, we will walk you through the required set Jython scripts with relevant explanations.

Creating the initial policies for Parlay X Services

The base policies of IBM WebSphere Telecommunications Web Services Server are similar to meta-data, based on which service implementations are registered during the deployment phase, accessed by the subscription management at run time during request processing, and accessed by the IBM WebSphere Telecommunications Web Services Server administration console to enable management of services. These policies are defined for the service interface name, and the set of operations that belong to each service.

Note: For more information about Parlay X V2.1 specifications, go to:

http://portal.etsi.org/docbox/TISPAN/Open/OSA/ParlayX21.html

The base set of policies typically define the following entities pertaining to a service:

- The Type entity denotes the kind of service supported by IBM WebSphere Telecommunications Web Services Server. Typically it is Parlay X only.
- The Service Identifier entity is a short descriptive string to identify a particular service interface.
- The Parent entity basically allows a self referential parent-child relationship with the Service Identifier.
- The Enabled entity is a flag that denotes whether a service is active or inactive.

As a reference, let us look at the Parlay X Presence Consumer service base policies. The Jython script for creating these policies is provided in Example 8-11.

Example 8-11 Initial policies of Parlay X Presence Consumer service definition

```
import sys.argv
import getopt

from pytwss.mbean import spm_mbean
from pytwss.utils import _, log
from pytwss.mbean.spm_mbean import SERVICE, SERVICE_GROUP

def main(**kwargs):
   svc_admin = spm_mbean.ServiceAdministration()

   # Initialize Presence
   spm_mbean.createService(svc_admin, 'Presence',
                'Parlay X', 1, 'Parlay X Presence', SERVICE_GROUP)
   spm_mbean.createService(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/consumer/v2_3/interface',
                'Presence', 1, 'Parlay X Presence Consumer Service', SERVICE)
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/consumer/v2_3/interface',
                'subscribePresence', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/consumer/v2_3/interface',
                'getUserPresence', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/consumer/v2_3/interface',
                'startPresenceNotification', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/consumer/v2_3/interface',
                'endPresenceNotification', 1, '')
   spm_mbean.createService(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'Presence', 1, 'Parlay X Presence Supplier Service', SERVICE)
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'publish', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'getOpenSubscriptions', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'updateSubscriptionAuthorization', 1, '')
   spm_mbean.createOperation(svc_admin,
```

Archived

```
'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'getMyWatchers', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'getSubscribedAttributes', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface',
                'blockSubscription', 1, '')
   spm_mbean.createService(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface',
                'Presence', 1, 'Parlay X Presence Notification Service', SERVICE)
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface',
                'statusChanged', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface',
                'statusEnd', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface',
                'notifySubscription', 1, '')
   spm_mbean.createOperation(svc_admin,

'http://www.csapi.org/wsdl/parlayx/presence/notification/v2_3/interface',
                'subscriptionEnded', 1, '')
```

The following points can be noted:

- The first statement in the script registers a Service Group type, which is identified by the name Presence. A service group is a collection of service interfaces.
- The service group identifier Presence is of type Parlay X.
- The Presence service group is flagged as enabled or active.
- Along with the above information, a meaningful description of the service group is also provided.
- The parent entity in this case is the word SERVICE_GROUP.

Let us look at the statements that create policies for a service:

- The second statement in the script registers a Service type, which is identified by the name Presence. A service is distinguished by the service XML namespace.
- The http://www.csapi.org/wsdl/parlayx/presence/consumer/v2_3/interface object is the namespace for the Parlay X Presence consumer service interface.

Archived

Note: It is possible to have multiple Service Implementations for a single Service Interface. At run time, the Web service requests from client applications comprise the service namespace. This namespace declaration in the request is used by IBM WebSphere Telecommunications Web Services Server to route the request to a specific service implementation. A detailed discussion about this topic is provided in “Creating policies for service implementation”.

- The Presence service is flagged as enabled or active.
- The parent entity in this case is the namespace mentioned above.

After creating policies for a service, let us look at the creation of policies for service operations:

- The third statement in the script registers a service operation with the name subscribePresence.
- The operation is flagged as enabled or active.
- There is no description provided in this case.

The rest of the statements in the script are for creating policies for the Presence Supplier and Presence Notifications service interfaces and their respective service operations.

Creating policies for service implementation

In IBM WebSphere Telecommunications Web Services Server, policies are created for out-of-box Parlay X service implementations during the deployment phase of their respective services. In the case of a custom Parlay X service implementation, a set of initial policies associated with it should also be created. The standard way to create service policies is to compose a Jython script and execute the scrip using the wsadmin.sh command line administration tool. As a reference, let us look at the sample script shown in Example 8-12.

Example 8-12 Sample script for creating service policies

```
import sys.argv
import getopt
from com.ibm.twss.spm.admin.common import ScopedPolicy
from com.ibm.twss.spm.admin.common import PolicyValue

from pytwss.mbean import spm_mbean, mbean_utils
from pytwss.utils import _, log

SI_SERVICE = 'http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface'
SVC_GROUP = 'PresenceSupplier'
SERVICE_IMPL = 'PX21_PRS_SPLR_IMS'
SI_IMPL = 'PX21_PRS_SPLR_IMS_SI'

def main(**kwargs):
    req_scope = 'ALL'
    if kwargs.has_key('req_scope'):
        req_scope = kwargs['req_scope']
    #svc_scope = SVC_GROUP
    svc_scope = SI_IMPL
    if kwargs.has_key('svc_scope'):
        svc_scope = kwargs['svc_scope']
        op_scope = 'ALL'
    if kwargs.has_key('op_scope'):
```

```
        op_scope = kwargs['op_scope']

    log(_("SOAC7550I") %{ 'impl' : SERVICE_IMPL })

    svc_admin = spm_mbean.ServiceAdministration()
    policy_admin = spm_mbean.PolicyAdministration()

    spm_mbean.registerServiceImplementation(
        svc_admin, SI_SERVICE, SI_IMPL, 1,
        'Parlay X V2.1 Presence Supplier IMS Service')

    policies = [
        ScopedPolicy(
            requesterIdentifier=req_scope,
            serviceIdentifier=svc_scope,
            operation=op_scope,
            name='service.config.ServiceImplementationName',
            value=PolicyValue(
                value='PX21_PRS_SPLR_IMS',
                type='String'
            )
        ),
        ScopedPolicy(
            requesterIdentifier=req_scope,
            serviceIdentifier=svc_scope,
            operation=op_scope,
            name='service.config.presence.supplier.PresenceServerURI',
            value=PolicyValue(
                value='',
                type='String'
            )
        ),
        ScopedPolicy(
            requesterIdentifier=req_scope,
            serviceIdentifier=svc_scope,
            operation=op_scope,
            name='service.Endpoint',
            value=PolicyValue(

value='http://9.3.4.188:9080/PX21_PRS_SPLR_IMS/services/PresenceSupplier',
                type='String'
            )
        ),
        ScopedPolicy(
            requesterIdentifier=req_scope,
            serviceIdentifier=svc_scope,
            operation=op_scope,
            name='service.config.presence.supplier.PresenceServerResourceName',
            value=PolicyValue(
                value='PX21_PRS_SPLR_IMS',
                type='String'
            )
        ),
        ScopedPolicy(
            requesterIdentifier=req_scope,
```

Archived

```
            serviceIdentifier=svc_scope,
            operation=op_scope,
            name='service.config.presence.supplier.PublishPresenceTimeout',
            value=PolicyValue(
                value='',
                type='String'
            )
        ),
        ScopedPolicy(
            requesterIdentifier=req_scope,
            serviceIdentifier=svc_scope,
            operation=op_scope,
            name='service.config.presence.supplier.SubscribePresenceWinfoTimeout',
            value=PolicyValue(
                value='',
                type='String'
            )
        )
]
    spm_mbean.createPolicies(policy_admin, policies)
    log(_("SOAC7551I") %{ 'impl' : SERVICE_IMPL })
```

From the above script, the following points can be noted:

- After the variable definitions, the applicability scopes are defined. The applicability of these policies are for a service implementation with the name PX21_PRS_SPLR_IMS. This identifier is the name of Enterprise Application comprised of the Parlay X Presence Supplier service implementation. The scopes defined here enable the subscription management function of IBM WebSphere Telecommunications Web Services Server to route an incoming request to a specific service implementation. The scopes are:
  - Requester
  - Service Identifier
  - Operation
- In the next significant statement, the JMX MBean references of Service Administration and Policy Administration, which are an integral part of Service Policy Manager, are obtained.
- The method call to register the service implementation creates a subscription entry in the Service Policy Manager database table.
- After registering the service implementation name, a set of service policies are created with the scope Requester as ALL, Service Identifier as PX21_PRS_SPLR_IMS_SI, and the Operation scope as ALL.

Note: The Service Policy Manager console, which is an essential tool provided in IBM WebSphere Telecommunications Web Services Server, is used to manage policies. It should be possible to create the above mentioned policies using the SPM console. Due to a limitation in the SPM console in IBM WebSphere Telecommunications Web Services Server V6.2, creation of policies specific to a service implementation should be done at the service definition level, not at the service implementation level. However, in a future release, this functionality will be made available in the Service Policy Manager Console.

Archived

In this section, we discussed the creation of the initial service settings and service policies for Parlay X Presence Supplier service. In the next section, we will walk you through the steps to create JMX MBeans and related artifacts to allow management of the service.

Note: The sample scripts discussed in this section are available for download. Refer to Appendix E, “Additional material” on page 399 for detailed instructions on how to download and work with this sample code.

8.7 Management provisions for new service

The IBM WebSphere Telecommunications Web Services Server administration console is integrated with the WebSphere Application Server administration console. The IBM WebSphere Telecommunications Web Services Server administration console allows administrators to manage the JMX MBean attributes of all services available in IBM WebSphere Telecommunications Web Services Server. A custom service implementation should also provide adequate support to manage the service and its attributes. By design, IBM WebSphere Telecommunications Web Services Server requires MBeans to be created specific to a service. During the processing of a request, if the policies are not found as SOAP headers in the request, the service implementation logic is obligated to use MBean attribute values of the service and process the request.

IBM WebSphere Telecommunications Web Services Server provides several libraries and a suggested packaging structure for registration and automatic rendering of the administration Web pages in the IBM WebSphere Telecommunications Web Services Server administration console. The IBM WebSphere Telecommunications Web Services Server administration console plug-in automatically handles navigation and rendering of Web content. The guidelines below describe how to construct and register a service implementation MBean. By registering the MBean of a service implementation, the MBean service specific attributes appear in the appropriate locations on the IBM WebSphere Telecommunications Web Services Server administration pages within the administration console.

In order to provide management features to a new service implementation, the following integration activities should be performed:

- An implementation of the com.ibm.soa.common.mbean.ServiceComponent interface class should be coded. The service specific implementation of the above interface will coordinate with MBean instances of the IBM WebSphere Telecommunications Web Services Server administration console and the service implementation MBean instance.
- An MBean implementation specific to the service implementation. IBM WebSphere Telecommunications Web Services Server Administration console provides three types of interfaces that extend the JMX javax.management.DynamicMBean interface. One of these provided interfaces should be extended for each MBean implementation of a specific service implementation. While it is it is true that it must inherit from DynamicMBean, it generally needs to conform to one of the IBM WebSphere Telecommunications Web Services Server MBean interfaces as well.

Archived

8.7.1 Developing the JMX MBean for service

The JMX MBeans and any supporting classes should be bundled along with the enterprise application that contains the service implementation. Refer to 8.3.2, “Service Platform Application Template” on page 256 for details about creating a custom service application based on a Service Platform Application Template. The platform.ear comprises several libraries and a deployment descriptor to enable integration of service specific MBean implementations with the IBM WebSphere Telecommunications Web Services Server administration console. These libraries contain components that initialize and register service specific MBean implementations during the startup process of the service enterprise application.

Note: The startup process of an application on WebSphere Application Server happens because of:

- Post deployment of an application to the application server, if the application is required to be active and service requests. The administrator can start the application using WebSphere Administration Console.
- During the startup process of an application server instance, all the deployed applications are also started.

Prerequisites for Service Management provisions

The com.ibm.soa.common.mbean.ServiceComponent interface provides the entry point for the IBM WebSphere Telecommunications Web Services Server administration console sub-system to initialize and register service implementation specific MBeans. The implementing class is registered with the IBM WebSphere Telecommunications Web Services Server administration console sub-system.

A Java archive (JAR) file comprising the Service component implementation described above, the MBean implementation, and any supporting helper classes should be created. This JAR file name should be specified in an additional deployment descriptor with the name admin.xma. The admin.xma should be bundled in the META-INF folder of the enterprise application archive (EAR) that comprises a service implementation. The admin.xma uses a simple comma-delimited format to initialize a service implementation specific service component class. The service component class then initializes and registers MBean implementations of the service implementation.

Note: The admin.xma deployment descriptor uses the following format:

<Category Name>,<Order>,<Component Name>,<Service Component class>,<J2EE Utility JAR name>

In the admin.xma, multiple lines can be used to register multiple service components. The meanings of these fields are described below:

- Category Name: The possible category names are:
  - Service: This category is intended for all service implementations. Based on this category, the service specific MBean attributes are rendered in a separate Web page in the IBM WebSphere Telecommunications Web Services Server administration console.
  - SupportService: This category is intended for support service implementations such as the Service Platform common components or other application components that provide utilities commonly consumed by all services in the Service Platform.

  - NetworkResources: The Network Resources category is intended for configuring information related to network element resources that are shared between services.
  - Global: The purpose of this category is to render other configuration settings that might be shared between service implementations. These settings can define the HTML table tag heading text. On the IBM WebSphere Telecommunications Web Services Server administration console, the first page displays the name of the service implementation as well as a description in a HTML table for each service.

- Order: The order in which this component will appear in the category when rendered on the IBM WebSphere Telecommunications Web Services Server administration console.
- Component Name: A unique component name that identifies the MBean of the service implementation. This name is passed to the ServiceComponent implementation to initialize the MBean implementation.
- J2EE Utility JAR name: The utility Java archive (JAR) that is part of the enterprise application archive (EAR). The ServiceComponent class, the MBean implementation, and any supporting classes should be packaged as a JAR file. This JAR file should be added as a utility module in the enterprise application archive that comprises the service implementation.

In addition to the deployment descriptor admin.xma, the service component must have Java resource bundles associated with it that contain translatable text for the IBM WebSphere Telecommunications Web Services Server administration console to render in a locale-specific manner. The label elements on the IBM WebSphere Telecommunications Web Services Server administration console Web pages for each service and service attribute should be captured in the resource bundle as key names. During the render phase, the IBM WebSphere Telecommunications Web Services Server libraries read the resource bundle key names and their respective values to dynamically generate the labels on the Web pages. A sample set of resource bundle entries are shown in Example 8-13.

Example 8-13 Sample entries in a resource bundle supporting a MBean implementation

```
soa.MyMBean.title=My MBean title
soa.MyMBean.help=This page allows for changing My MBean settings
soa.MyMBean.breadcrumb=MBean Settings
soa.MyMBean.tab=Runtime
soa.MyMBean.attribute.name.MyAttribute1=My Custom Attribute #1
soa.MyMBean.attribute.description.MyAttribute1=Description of My Custom Attribute
#1
```

From the above sample, the key names from this Java resource bundle are fetched by the IBM WebSphere Telecommunications Web Services Server administration console sub-system for rendering Web content in the Web pages. As the HTML elements, such as labels and text, are generated dynamically during navigation, the IBM WebSphere Telecommunications Web Services Server administration console sub-system will call the ServiceComponent class's method `getString(String componentName, String key, Locale locale)` or the MBean implementation's method `getString(String key, Locale locale)` to resolve the translatable key value text.

If a special non-translatable key must be defined in order to classify service components within the IBM WebSphere Telecommunications Web Services Server administration console, then define the entries shown in Example 8-14 in the resource bundle.

Example 8-14 Entries to be defined in the resource bundle

```
# Non-translatable classification such as for Web service implementations
soa.SOAConsoleSettings.type_name=SOAConsoleWebServices
```

The cases where non-translatable key names are applicable are definitions of EAR names that appear in the IBM WebSphere Telecommunications Web Services Server administration console under the Services, Support Services, or Network Resources listed in the left navigation panel of the IBM WebSphere Telecommunications Web Services Server administration console.

Note: The possible values for the above resource bundle key are:

- SOAConsoleWebServices
- SOAConsoleServicePlatform
- SOAConsoleNetworkResources

Implementing an MBean for service implementations

In order to implement MBean for a service implementation, three MBean interfaces are provided with the IBM WebSphere Telecommunications Web Services Server administration console. The IBM WebSphere Telecommunications Web Services Server InfoCenter provides Java Documentation with descriptions at the following link:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

The interfaces extend the JMX DynamicMBean interface and provide additional methods for rendering and interacting with MBean from the IBM WebSphere Telecommunications Web Services Server administration console. These interfaces are:

- com.ibm.soa.common.mbean.GeneralProperties: This interface provides a Web page comprising MBean attributes that are a simple list of attribute/value pairs.
- com.ibm.soa.common.mbean.ComponentProperties: This interface provides a simple list of attribute/value pairs and a set of additional properties that link to other MBeans. This allows for the organization of information in the IBM WebSphere Telecommunications Web Services Server administration console in a meaningful way.
- com.ibm.soa.common.mbean.CustomProperties: This interface provides a list of attributes/value pairs that can be defined and modified by the administrator, enabling the administrator to Create, Retrieve, Update, and Delete the values of these properties.
- com.ibm.soa.common.mbean.ActionList: A list of objects that can have an action applied to them.

In order for the IBM WebSphere Telecommunications Web Services Server administration console to correctly interact with the MBean, the object name of the MBean must include the following attribute properties:

- Cell: The name of the WebSphere Application Server cell
- Node: The name of the WebSphere Application Server node
- Process: The name of the WebSphere Application Server instance
- Name: The name of the MBean
- Type: The type name of the MBean

- Interface: The name of the Administrator console interface used (that is, GeneralProperties, ComponentProperties, CustomProperties, or ActionList). These names are provided with package qualification (without a package name).
- ApplicationName: The name of the Enterprise application that is running the resource.

8.7.2 MBean implementation for sample scenario

In this section, we will walk you through the sample scenario, that is, the Parlay X Presence Supplier service implementation specific to MBean implementation and the Service Component class. Along with the required helper classes, we will also look at a sample deployment descriptor and packaging the MBean implementation code in the service enterprise application.

The MBean implementation class for Parlay X Presence Supplier service is com.ibm.twss.parlayx21.presence.supplier.mbean.PresenceSupplierMBean.

The Service Component implementation class is com.ibm.twss.parlayx21.presence.supplier.mbean.PresenceSupplierServiceComponent.

The MBean class for our sample scenario must encapsulate a set of configurable attribute value pairs. These attribute value pairs are displayed on the IBM WebSphere Telecommunications Web Services Server administration console. An administrator can edit the value of each attribute depending on the attribute type. Since the MBean attributes are rendered by the IBM WebSphere Telecommunications Web Services Server administration console sub-system on to Web pages, the MBean attributes for Parlay X Presence Supplier should have certain characteristics:

- Each MBean attribute should have a 'name' attribute and corresponding accessor method.
- Each MBean attribute should have a 'type' attribute and corresponding accessor method. The MBean attributes should be either READONLY or allow for the Create, Retrieve, Update, and Delete operations. For simplicity, these two types can be represented as constants by the names READONLY and STRING respectively.
- The value of each MBean attribute is a java.lang.String type. In order to format the String to represent a number or date, the appropriate formatting logic can be implemented.
- Each MBean attribute should have a default value defined.
- To provide an extension mechanism in cases where an MBean attribute is required to have one out of a finite set of values, you can optionally make a provision for an allowed set of values. As this feature requires additional functionality to render the HTML options tag, this feature can be treated as optional.
- Each MBean attribute should have the appropriate validation mechanism depending on the type of the attribute value.

The sample code shown in Example 8-15 demonstrates the behavior expected out of a MBean helper class. This helper class holds the state of an MBean attribute.

Example 8-15 Sample code snippet from MBeanAttributeHelper.java; constructor and accessor methods

```
public MBeanAttributeHelper(String name, Class clazz,
      AttrType attributeType, Object defaultValue,
      String possibleValues[], MBeanAttributeValidator validator,
      String resourceBundleName, String resourceBundleKey) {
   if (name == null)
      throw new NullPointerException("name == null");
```

```
        if ("".equals(name.trim()))
            throw new IllegalArgumentException(
                  "name must contain non-whitespace");
        this.name = name;
        if (clazz == null)
            throw new NullPointerException("clazz == null");
        this.clazz = clazz;
        if (attributeType == null)
            throw new NullPointerException("attributeType == null");
        this.attributeType = attributeType;
        if (defaultValue == null)
            throw new NullPointerException("defaultValue == null");
        this.defaultValue = defaultValue;
        if (resourceBundleName == null)
            throw new NullPointerException("resourceBundleName == null");
        if (resourceBundleKey == null)
            throw new NullPointerException("resourceBundleKey == null");
        String resourceBundleMessage = null;
        try {
            ResourceBundle resourceBundle = ResourceBundle.getBundle(
                  resourceBundleName, Locale.getDefault());
            resourceBundleMessage = resourceBundle.getString(resourceBundleKey);
        } catch (RuntimeException re) {
            resourceBundleMessage = null;
            if (LogHelper.isExceptionEnabled(logger)) {
                LogHelper.caughtTrace(logger, "<init>", re);
            }

        }
        if (resourceBundleMessage == null
              || resourceBundleMessage.trim().length() == 0)
            description = "";
        else
            description = resourceBundleMessage;
        if (possibleValues == null)
            throw new NullPointerException("possibleValues == null");
        this.possibleValues = possibleValues;
        if (validator == null)
            throw new NullPointerException("validator == null");
        this.validator = validator;
        info = new MBeanAttributeInfo(this.name, this.clazz.getName(),
              description, true, !this.attributeType
                    .equals(AttrType.READONLY), false);
    }

    public String getName() {
        String retval = name;
        return retval;
    }

    public String getAttributeType() {
        String retval = String.valueOf(attributeType);
        return retval;
    }
```

```
public Object getDefaultValue() {
   Object retval = defaultValue;
   return retval;
}

public String[] getAttributePossibleValues() {
   String retval[] = possibleValues.length != 0 ? possibleValues : null;
   return retval;
}

public MBeanAttributeValidator getValidator() {
   MBeanAttributeValidator retval = validator;
   return retval;
}

public MBeanAttributeInfo getMBeanAttributeInfo() {
   MBeanAttributeInfo retval = info;
   return retval;
}
```

Now that we know how to create MBean attributes, let us look at the specifics of creating a MBean for a Parlay X Presence Supplier implementation. One of the vital features that an MBean implementation should provide is persistence of the MBean attribute state. By design, we recommend that custom service implementations should provide persistence of the MBean state. For the sample scenario, we have implemented a simple data access object that performs the following functions:

- Retrieve a persisted Attribute Value for an Attribute Name. The method that implements this logic is public String readFromStore(String key).
- Retrieve all persisted Attribute Name Value pairs. The method that implements this logic is public Hashtable readAllFromStore().
- Insert an Attribute Name Value pair.
- Update the Attribute Value for an Attribute Name.

Note: The sample code for the Parlay X Presence Supplier MBean implementation and the supporting classes are available for download. Refer to Appendix E, “Additional material” on page 399 for detailed instructions on how to download and work with this sample code

The following methods in javax.management.DynamicMBean are overridden by the MBean implementation of the Parlay X Presence Supplier service. The MBean implementation invokes various methods of the data access object. The MBean methods and corresponding method calls on the data access object are listed here:

- The public Object getAttribute(String name) method of an MBean implementation invokes the public String readFromStore(String key) method of a data access object.
- The public AttributeList getAttributes(String[] attributes) method of an MBean implementation invokes the public Hashtable readAllFromStore() method of a data access object.
- The public void setAttribute(Attribute attribute) method of an MBean implementation invokes the public void writeToStore(String key, String value) method of a data access object.

- The public AttributeList setAttributes(AttributeList attributes) method of MBean implementation invokes the public void writeToStore(String key, String value) method of data access object in a loop for all the attributes.

Apart from the methods that get access in order to persist and retrieve attribute name value pairs, the other significant method calls are described here:

- The public void registerMBean() method comprises logic that obtains a reference to the com.ibm.websphere.management.AdminService class and registers the PresenceSupplierMBean instance in the JMX server of WebSphere Application Server.
- The public void unregisterMBean() method comprises logic that obtains a reference to the com.ibm.websphere.management.AdminService class and un-registers the PresenceSupplierMBean instance in the JMX server.
- The public Object invoke(String actionName, Object[] params, String[] signatures) method comprises logic that uses the Java reflection mechanism and calls on various methods of the PresenceSupplierMBean instance. This method is used extensively by the IBM WebSphere Telecommunications Web Services Server administration console sub-system for accessing the MBean state.

The complete listing of the MyMBean code is provided in Example 8-16.

Note: We recommend designing an abstract class to implement the com.ibm.soa.common.mbean.GeneralProperties and javax.management.DynamicMBean interfaces provided in Example 8-16.

Example 8-16 Code from MyMBean.java

```
//IBM Sample Source Materials
//
//Sample source materials are supplied As-Is.
//No warranty is expressed or implied.
//
//Product(s): 5724-005
//
//(C)Copyright IBM Corp. 2000, 2006
//
//The source code for this program supplied under the terms of the
//End User License Agreement (EULA) that accompanied this product.
//***
package com.ibm.sample.mbean;
import java.lang.reflect.*;
import java.util.*;
import java.io.*;
import javax.management.*;
import com.ibm.websphere.management.*;
import com.ibm.soa.common.mbean.GeneralProperties;
/**
* The class for administered settings on MyService.
*/
public class MyMBean implements GeneralProperties, DynamicMBean {
/**
* instance.
*/
private static MyMBean instance = null;
```

```
/**
*  The  MBean  ObjectName.
*/
private  ObjectName  objectName  =  null;

/**
*  The  MBeanInfo  for  this  MBean.
*/
private  MBeanInfo  info  =  null;

/**
*  The  display  Name.
*/
private  String  displayName  =  null;

/**
*  The  constructor
*/
public  MyMBean()  {

   instance  =  this;

}

/**
*  Get  the  settings  instance
*  @return  the  bean
*/
public  static  MyMBean  getInstance()  {
      if  (instance  ==  null)  {
         new  MyMBean();
      }
      return  instance;
}

/**
*  Register  this  mbean
*/
public  void  registerMBean()  {
      //  Register  MBean  with  a  properly  formated ObjectName with the
      //  AdminService

      //  The  applicationName  is  the  name  of  the  EAR  file  this  code is
running
      //  in.
      String  applicationName  =  ServicePlatform.getCurrentApplicationName();
      String  beanName  =  "MyMBean";

      try  {
//  Get  the  Administration  Service  and  attributes
AdminService  as  =  AdminServiceFactory.getAdminService();
ObjectName  server  =  as.getLocalServer();
String  serverName  =  server.getKeyProperty("name");
String  cell  =  server.getKeyProperty("cell");
String  node  =  server.getKeyProperty("node");
```

Archived

```
String process = server.getKeyProperty("process");

Properties props = new Properties();
props.put("type", beanName);
props.put("name", beanName);
props.put("cell", cell);
props.put("node", node);
props.put("process", process);
props.put("interface", "GeneralProperties");
props.put("ApplicationName", applicationName);
props.put("resource", "soa");
objectName = new ObjectName("WebSphere", props);

// If not already registered then register it.
if (!as.isRegistered(objectName)) {
   as.registerMBean(this, objectName);
} else {
   unregisterMBean(objectName);
   as.registerMBean(this, objectName);
}
} catch (Exception e) {
   e.printStackTrace();
}
}

/**
* Unregister the MBean
*/
public void unregisterMBean() {
      // Un register this MBean instance with the AdminService
      try {
         // Get the Administration Service and attributes
         AdminService as = AdminServiceFactory.getAdminService();

Class types[] = new Class[1];
types[0] = javax.management.ObjectName.class;
Object parms[] = new Object[1];
parms[0] = objectName;
Class cls = as.getClass();
Method m = cls.getMethod("unregisterMBean", types);
m.invoke(as, parms);

objectName = null;
      } catch (Exception e) {
         e.printStackTrace();
      }
}

/**
* Unregister the MBean.
* @param name
* the ObjectName
*/
public static void unregisterMBean(ObjectName name) {
   try {
```

Archived

```
// Get the Administration Service and attributes
AdminService as = AdminServiceFactory.getAdminService();

Class types[] = new Class[1];
types[0] = javax.management.ObjectName.class;
Object parms[] = new Object[1];
parms[0] = name;
Class cls = as.getClass();
Method m = cls.getMethod("unregisterMBean", types);
m.invoke(as, parms);

} catch (Exception e) {
   e.printStackTrace();
}
}

/* ******************************************************** */
/* **** General Properties Implementation ***** */
/* ******************************************************** */
/**
* Get the type of named attribute.
*
* @param name the name of the attribute
* @return "CHOICE", "STRING", "READONLY", "TRANSCHOICE"
*/
public String getAttributeType(String name) {
   if (name.equals("ConnectionTimeout")) {
      return "STRING";
   }
   else if (name.equals("EndPoint")) {
      return "STRING";
   }
   return "READONLY";
}

/**
* Get the list of possible values for the attribute that is of type
* "CHOICE" or TRANSCHOICE. A choice defiles a choice of possible values.
   * A TRANSCHOICE defines a set of keys to translated values.
*
* @param name the name of the attribute
* @return the array of possible values, return null if the name is a
text
* field item
*/
public String[] getAttributePossibleValues(String name) {
     return null;
}

/**
* Get the ObjectName String for the named attribute's link.
* @param name the name of an attribute
* @return the ObjectName string that links to another MBean
*/
public String getAttributeProperties(String name) {
```

Archived

```
   return  null;
}

/**
*  Get  the  current  attribute  context. This is a string that is used to scope
the data when storing
   * in a data store.
*  @param  name  the  attribute  name
*  @return  the  context  string
*/
public  String  getAttributeContext(String  name)  {
   return  "";
}

/**
*  Gets  the  navigation  context  for  an  MBean.  The  navigation  context
*  is  a  name  qualified  with  dots  (for  example,  'attrib1.attrib2')  that
*  references  the  linking  context  between  bean  objects.  A  sub-class
*  can  use  this  method  to
*  determine  the  navigation  context  when  manipulating  attribute  keys.
*  @return  A  qualified  context  string.  May be  an empty string.
*/
public  String  getContext()  {
   return  "";
}

/**
*  Sets  the  navigation  context  for  an  MBean.  The  navigation  context
*  is  a  name  qualified  with  dots  (for  example,  'attrib1.attrib2')  that
references the
*  linking  context  between  bean  objects.  This  method  is  called  by  the
JSP  as
*  it  propagates  the  context  during  navigation.
*  @param  context  A  qualified  context  string.  May be an empty string.
*/
public  void  setContext(String  context)  {
     //  not  used  for  simple  general  properties
}

/**
*  Get  the  Application  Name  (EAR  Name).
*  @return  the  application  name
*/
public  String  getApplicationName()  {
     return  objectName.getKeyProperty("ApplicationName");
}

/**
*  Get  the  Object  name  of  this  MBean.
*  @return  the  ObjectName  String
*/
public  String  getObjectName()  {
     return  objectName.toString();
}
```

Archived

```
/**
*  Get  the  raw  ObjectName  of  this  MBean.
*  @return  the  raw  Objectname
*/
public  ObjectName  getRawObjectName()  {
      return  objectName;
}

/**
*  Get  the  typename  that  this  bean  WebSphere  ApplicationServerregistered
under.
*  @return  the  type  name
*/
public  String  getTypeName()  {
      return  objectName.getKeyProperty("type");
}

/**
*  Get  the  MBean  name.
*  @return  the  name
*/
public  String  getBeanName()  {
      return  objectName.getKeyProperty("name");
}

/**
*  Get  the  interfaces  names  that  this  MBean  supports  forJSP  rendering.
*  @return  the
   */
public  String  getInterfaceName()  {
      return objectName.getKeyProperty("interface");
}

/**
*  If  the  Title  or  Breadcrumb  for  an  MBean  (defined  in  the  resource
bundle)
*  contains  the  placeholder  string  ${name}  then  the  JSP  will  call  this
*  method  on  the  MBean  for  the  display  name  to  substitute  for  the
*  placeholder.
*  @return  "",  or  the  string  to  substitue  for  the  placeholder.
*/
public  String  getDisplayName()  {
      return  displayName;
}

/**
*  If  the  Title  or  Breadcrumb  for  an  MBean  (defined  in  the  resource
bundle)
*  contains  the  placeholder  string  ${name}  then  the  JSP  will  call  this
*  method  on  the  MBean  for  the  display  name  to  substitute  for  the
*  placeholder.
*  @param  locale  the  locale
*  @return  "",  or  the  string  to  substitue  for  the  placeholder.
*/
public  String  getDisplayName(Locale  locale)  {
```

Archived

```
        return displayName;
    }

    /**
    * Set the display Name.
    * @param name the name
    */
    public void setDisplayName(String name) {
        this.displayName = name;
    }

    /**
    * Get an attribute
    *  @param name the name of the attribute
    * @return the attribute value
    * @throws AttributeNotFoundException
    * @throws MBeanException
    * @throws ReflectionException
    */
    public Object getAttribute(String name) throws AttributeNotFoundException,
       MBeanException, ReflectionException {
          String keyBase = getBeanName();
          String attr = name;
          if (attr.equals("ConnectionTimeouit") || attr.equals("EndPoint")) {
             String value = getProperty(keyBase + "." + attr);
             return value;
          }
          return null;
    }

    /**
    * Set the attribute
    *
    * @param attribute the attribute to set
    * @throws AttributeNotFoundException
    * @throws InvalidAttributeValueException
    * @throws MBeanException
    * @throws ReflectionException
    */
    public void setAttribute(Attribute attribute)
    throws AttributeNotFoundException, InvalidAttributeValueException,
    MBeanException, ReflectionException {

          String keyBase = getBeanName();
          String attr = attribute.getName();
          String value = (String)attribute.getValue();

          if (attr.equals("ConnectionTimeout")) {
             setProperty(keyBase + "." + attribute.getName(), value);
             return;
          }
          else if (attr.equals("EndPoint")) {
             setProperty(keyBase + "." + attribute.getName(), value);
             return;
          }
```

```
}

/**
*  Set  a  set  of  attributes.
*  @param  attributelist  the  list  of  attributes  to  set
*  @return  the  result  attribute
*/
public  AttributeList  setAttributes(AttributeList  attributelist)  {
      //  No  attributes  are  supported
      if  (attributelist  ==  null)  {
         return  null;
      }

      String[]  attributes  =  new  String[attributelist.size()];
      for  (int  i  =  0;  i  <  attributelist.size();  i++)  {
         try  {
            Attribute  attribute  =  (Attribute)attributelist.get(i);
            attributes[i]  =  attribute.getName();
            setAttribute(attribute);
         }
         catch  (ReflectionException  re)  {
            re.printStackTrace();
         }
         catch  (AttributeNotFoundException  anfe)  {
            anfe.printStackTrace();
         }
         catch  (InvalidAttributeValueException  iave){
            iave.printStackTrace();
         }
         catch  (MBeanException  mbe)  {
            mbe.printStackTrace();
         }
      }
      return  getAttributes(attributes);
}

/**
*  Validate  the  attribute
*
*  @param  attribute  the  attribute  to  validate
*  @throws  AttributeNotFoundException
*        on  invalid  attribute
*  @throws  InvalidAttributeValueException
*        on  an  validation  error
*  @throws  MBeanException
*        on  bean  error
*  @throws  ReflectionException
*        on  dynamic  invocation  error
*/
public  void  validateAttribute(Attribute  attribute)
      throws  AttributeNotFoundException,  InvalidAttributeValueException,
MBeanException,  ReflectionException  {

      String  name  =  attribute.getName();
      String  value  =  (String)attribute.getValue();
```

Archived

```
      //  validate  individual  attribute  values

      if  (name.equals("ConnectionTimeout"))  {
         if  (!IsValidInteger(value))  {
            throw  new  InvalidAttributeValueException("Invalid  value  "
                  +  name);
         }
      }
      else  if  (name.equals("EndPoint"))  {
         if  (!IsValidEndPoint(value))  {
            throw  new  InvalidAttributeValueException("Invalid  value  "
                  +  name);
         }
      }
}

/**
*  Get  the  attribute  names.
*  @param  as  the  array  of  attribute  names
*  @return  the  result  list
*/
public  AttributeList  getAttributes(String  as[])  {
      AttributeList  list  =  new  AttributeList();
      //  no  attributes  are  supported  by  the  base  class
      if  (as  !=  null)  {
         //  Helper  to  get  a  specific  list  of  attributes
         for  (int  i  =  0;  i  <  as.length;  i++)  {
            try  {
               String  name  =  as[i];
               Object  value  =  getAttribute(name);
               list.add(new  Attribute(name,  value));
            }
            catch  (ReflectionException  re)  {
               //  ignore
            }
            catch  (AttributeNotFoundException  anfe)  {
               //  ignore
            }
            catch  (MBeanException  mbe)  {
               //  ignore
            }
         }
      }
      else  {
         //  get  the  list  from  the  attribute  info
         MBeanAttributeInfo  attrs[]  =  makeMBeanAttributeInfo();
         as  =  new  String[attrs.length];
         for  (int  i  =  0;  i  <  attrs.length;  i++)  {
            as[i]  =  attrs[i].getName();
         }
      }
      //  if  the  parameter  is  null,  then  build  the  AttributeList,and
return,
      //  so  subclasses  can
```

Archived

```
      //  add  in  their  names.
      return  list;
}

/**
*  Get  the  MBeanInfo  which  describes  the  operations,  attributes  and
*  notifications  of  the  MBean.
*  @return  the  MBeanInfo
*/
public  MBeanInfo  getMBeanInfo()  {
      if  (info  ==  null)  {
         //  Set  the  MBeanInfo
         info  =  makeMBeanInfo();
      }
      return  info;
}

/**
*  Make  the  MBeanInfo  object.
*  @return  the  MBeanInfo
*/
private  MBeanInfo  makeMBeanInfo()  {
      MBeanOperationInfo  ops[]  =  new  MBeanOperationInfo[0];
      MBeanAttributeInfo  attrs[]  =  makeMBeanAttributeInfo();
      MBeanConstructorInfo  cons[]  =  new  MBeanConstructorInfo[0];
      MBeanNotificationInfo  nots[]  =  new  MBeanNotificationInfo[0];

      MBeanInfo  info  =  new  MBeanInfo(this.getClass().getName(),
            getBeanName(),  attrs,  //  MBeanAttributesInfo[]
            cons,  //  MBeanContructorInfo[]
            ops,  //  MBeanOperationInfo[]
            nots  //  MBeanNotificationInfo[]
      );

      return  info;

}

/**
*  Make  the  attribute  info
*
*  @return  the  info  array
*/
protected  MBeanAttributeInfo[]  makeMBeanAttributeInfo()  {

      MBeanAttributeInfo  attrs[]  =  new  MBeanAttributeInfo[]  {
            new  MBeanAttributeInfo("ConnectionTimeout",  "java.lang.String",
                  "Connection  timeout  in  milliseconds",  true,  true, false),
            new  MBeanAttributeInfo("EndPoint",  "java.lang.String",
                  "Network  Element  Endpoint",  true,  true,  false)  };

      return  attrs;
}

/**
```

Archived

```
* Get a string from a resource if possible, otherwise return null, and the
console
   * will continue looking for the resource and possibly provide a default value.
*
* @param key the key
* @param locale the locale
* @return the result, or null if not found.
*/
public String getString(String key, Locale locale) {
      String bundleClass = "com.ibm.sample.mbean.resources.MyMBean";
      String value = null;
      try {
         ResourceBundle bundle = ResourceBundle.getBundle(bundleClass, locale);
         value = bundle.getString(key);
         return value;
      } catch (Exception e) {
      }
      return null;
}

/**
* Get the help page.
*
* @param name the name of the file
* @param locale the locale
* @return the page content
*/
public String getHelpPage(String name, Locale locale) {
      String bundleClass = "com/ibm/sample/mbean/resources/" +
locale.toString() + "/" + name;
      ClassLoader ldr = this.getClass().getClassLoader();
      String result = getResourceAsString(ldr, bundleClass);
      return result;
}

/**
* Get the resource as a String.
* @param ldrClass the class loader.
* @param resourceName the resourceName.
* @return the string, or returns the name of the resourcethatismissing.
*/
protected String getResourceAsString(ClassLoader ldrClass,
      String resourceName) {
      try {
         final ClassLoader ldr = ldrClass;
         final String res = resourceName;

         InputStream is = (InputStream)java.security.AccessController
         .doPrivileged(new java.security.PrivilegedAction() {
            public Object run() {
            // We're assuming that the requested locale is of
            // the specific form (XX_XX)
            // If it is not, we'll try what they give us, then
            // just fail....
            int iIndexOfBar = res.lastIndexOf("_");
```

```
        String resourceName_specific = res;
        InputStream is2 = null;
        // Attempt the provided locale (which is generally a
        // specific locale (ie zh_TW) but can be general
        // (ie zh)
        is2 = ldr
              .getResourceAsStream(resourceName_specific);
        if (is2 != null || iIndexOfBar < 0) {
           return is2;
        }

        // If we can't find it, look for a more general
        // locale.
        String resourceName_general = res;
        int i = resourceName_general.lastIndexOf("_");
        if (i >= 0 && i < resourceName_general.length() - 3) {
           // remove the _XX from the string, and try again
           resourceName_general = resourceName_general
                 .substring(0, i)
                 + resourceName_general.substring(i + 3);
        }

        // Attempt the general locale (ie zh)
        is2 = ldr.getResourceAsStream(resourceName_general);
        if (is2 != null || iIndexOfBar < 0) {
           return is2;
        }

        // Didn't find a specific or general locale, so try
        // en_US.
        String resourceName_default = res;
        i = resourceName_default.lastIndexOf("_");
        if (i >= 0 && i >= 3
              && i < resourceName_default.length() - 3) {
           resourceName_default = resourceName_default
                 .substring(0, i - 2)
                 + "en_US"
                 + resourceName_default.substring(i + 3);
        }

        // Attempt the default ("en_US")
        is2 = ldr.getResourceAsStream(resourceName_default);
        return is2;
     }
  });
     Stringvalue =null;

     int size = is.available();
     byte buf[] = new byte[size];
     int len = is.read(buf, 0, size);
     is.close();
     value = new String(buf, 0, len);
     return value;
  } catch (Exception e) {
```

```
        //e.printStackTrace();
        return "name: " + resourceName;
      }
}

/**
* Refresh the dynamic content.
*/
public void refreshDynamicContent() {
     // update any dynamic information
}

/**
* Invoke an operation on this mbean.
    * @param s the operation name
* @param aobj the parameters
* @param as the parameter types
* @return the result
* @throws MBeanException
*    on mbean error
* @throws ReflectionException
*    on dynamic invocation error
*/
public Object invoke(String s, Object aobj[], String as[])
        throws MBeanException, ReflectionException {
     // Now do the action
     try {
        Class cls = this.getClass();
        Class types[] = new Class[(as != null ? as.length : 0)];
        for (int j = 0; j < types.length; j++) {
           types[j] = Class.forName(as[j], true,
this.getClass().getClassLoader());
        }
        Method m = cls.getMethod(s, types);
        Object result = (Object)m.invoke(this, aobj);
        return result;
     }
     catch (java.lang.reflect.InvocationTargetException ite) {
        Throwable targetException = ite.getTargetException();
        if (targetException instanceof MBeanException) {
           throw (MBeanException)targetException;
        }
        else {
           throw new MBeanException((Exception)targetException,
                 targetException.getMessage());
        }
     }
     catch (Exception e) {
        throw new ReflectionException(e, "Error accessing the method("+s +
")");
     }
}

/**
* Get a property from the data store * @param key the key
```

```
*  @return  the  value
*  @throws  MBeanException
*           on  error
*/
protected  String  getProperty(String  key)  throws  MBeanException  {
      //  read  from  a  persistent  location
      return "a  value";
}

/**
*  Set  a  property  in  the  data  store *  @param  key  the  key
*  @param  value  the  value
*  @throws  InvalidAttributeValueException
*           on  length  error
*  @throws  MBeanException
*           on  error
*/
protected  void  setProperty(String  key,  String  value)
         throws  InvalidAttributeValueException,  MBeanException {
      //  write  to  a  persistent  location
}

/**
*  Is  this  string  a  valid  integer.
*  @param  s  the  string
*  @return  true  for  a  valid  integer
*/
private  boolean  IsValidInteger(String  s)  {
      return  true;
}

/**
*  Is  this  string  a  valid  endpoint
*  @param  s  the  string
*  @return  true  for  a  valid  endpoint
*/
    private  boolean  IsValidEndPoint(String  s)  {
      return  true;
    }
}
```

The Service Component implementation class coordinates with the IBM WebSphere Telecommunications Web Services Server administration console sub-system and executes various methods on the instance of the MBean implementation.

Archived

The listing of the all the methods in the com.ibm.twss.parlayx21.presence.supplier.mbean.PresenceSupplierServiceComponent class are shown in Example 8-17.

Example 8-17 Sample code snippet from ServiceComponent implementation class

```
public String getBeanName(String string) throws RemoteException {
   if (LogHelper.isEntryEnabled(logger)) {
      LogHelper.entry(logger, "getBeanName", new Object[] { string });
   }
   return PresenceSupplierMBean.getInstance().getBeanName();
}

public String getObjectName(String string) throws RemoteException {
   if (LogHelper.isEntryEnabled(logger)) {
      LogHelper.entry(logger, "getObjectName", new Object[] { string });
   }
   return PresenceSupplierMBean.getInstance().getObjectName();
}

public String getString(String string, String key, Locale locale)
      throws RemoteException {
   if (LogHelper.isEntryEnabled(logger)) {
      LogHelper.entry(logger, "getString", new Object[] { string });
   }
   return PresenceSupplierMBean.getInstance().getString(key, locale);
}

public void initialize(String string1, String string2)
      throws RemoteException {
   if (LogHelper.isEntryEnabled(logger)) {
      LogHelper.entry(logger, "initialize", new Object[] { string1,
            string2 });
   }
   PresenceSupplierMBean.getInstance().initialize();
}

public void registerMBean(String string) throws RemoteException {
   PresenceSupplierMBean.getInstance().registerMBean();
}

public void unregisterMBean(String string) throws RemoteException {
   PresenceSupplierMBean.getInstance().unregisterMBean();
}
```

The MBean implementation classes, along with the supporting classes and libraries (if any) pertaining to a custom service, should be packaged in a Java archive (JAR) as described in “Prerequisites for Service Management provisions” on page 300. Let us look at the following steps for packaging the MBean implementation code of the Parlay X Presence Supplier service:

1. The creation and packaging of MBean implementation and supporting classes should start with the creation of a *Java project*. In Rational Application Developer (RAD), the MBean Java project should be made part of the Parlay X Presence Supplier service enterprise application project. Figure 8-26 on page 321 illustrates the creation of a MBean Java project. In the Project Explorer of Rational Application Developer, right-click the

Enterprise Application project **PX21_PRS_SPLR_IMSApp**, select **New**, and then select **Project...**. In the New Project dialog, expand **Java** and select **Java Project**.

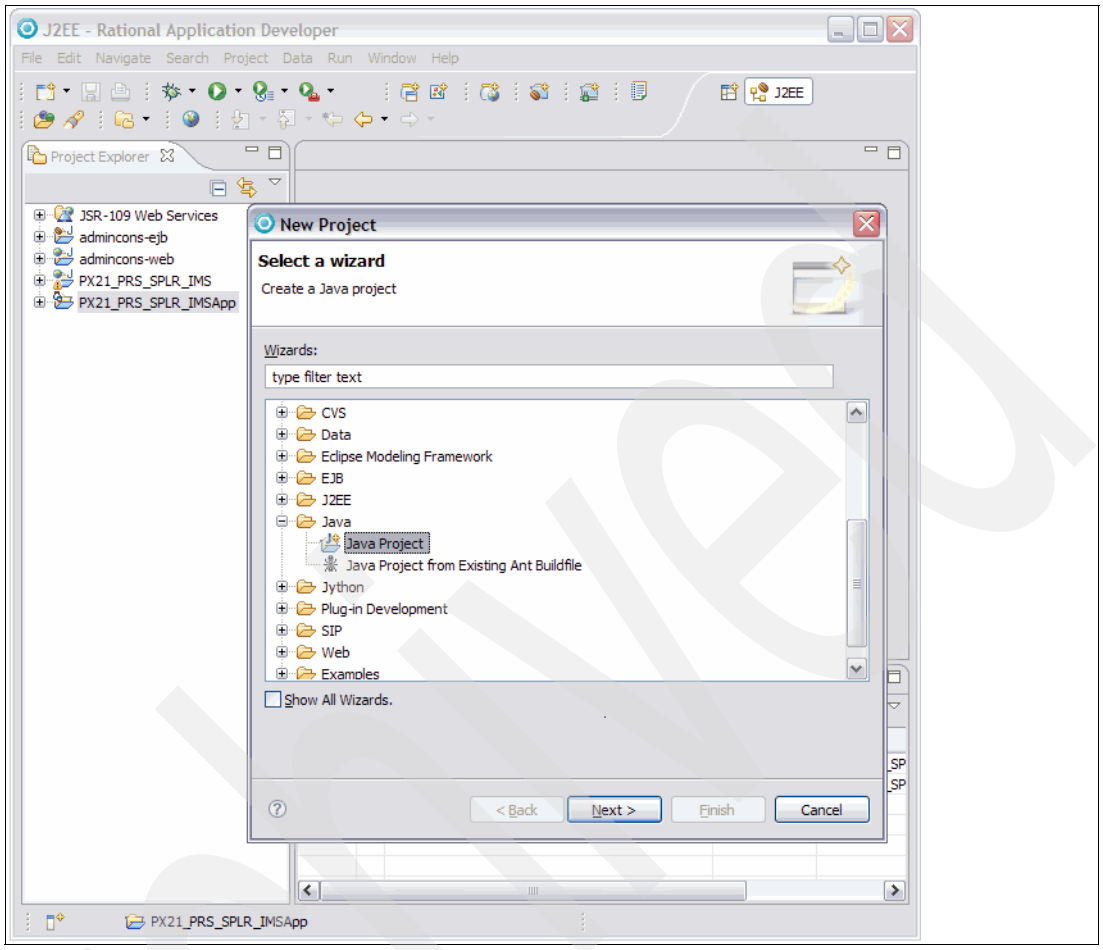


Figure 8-26 Creation of a Java project for MBean implementation

2. Click the **Next** button. In the Project name field, enter PX21_PRS_SPLR_IMSMBean. Click the **Next** button at the bottom, leaving the default settings. In the Java Settings window (Figure 8-27 on page 322), select the **Libraries** tab and add the following JAR files:
   - 'com.ibm.ws.runtime_6.1.0.jar' is located in the folder WAS_ROOT/plugins.
   - admincons.jar can be added by selecting it from the PX21_PRS_SPLR_IMSApp project. Use the **Add JARs...** button to select the JAR file.
   - soaruntime.jar can be added by selecting it from the PX21_PRS_SPLR_IMSApp project. Use the **Add JARs...** button to select the JAR file.

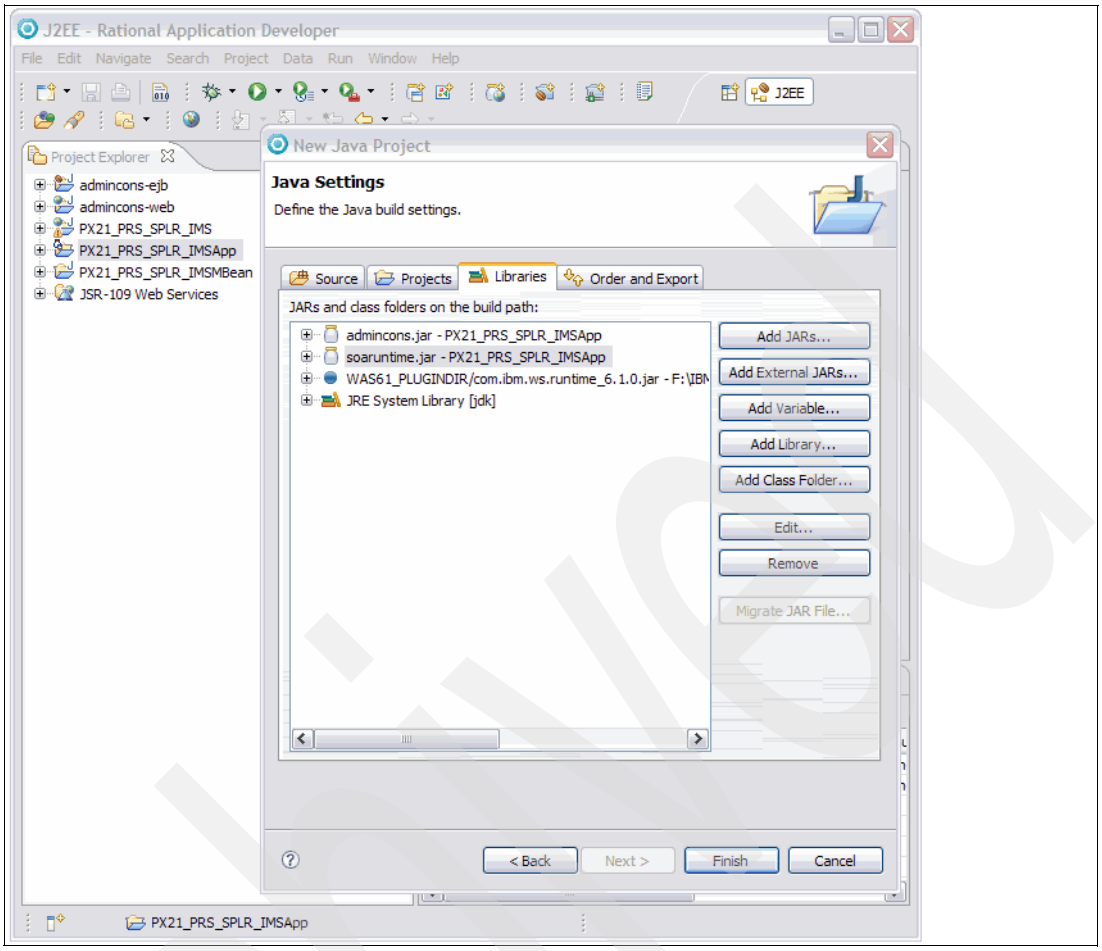


Figure 8-27 Setting up the required Java libraries in the MBean Java project

3. Click the **Finish** button. This action will create the Java project. Create the required set of classes by following the instructions detailed in 8.7.2, “MBean implementation for sample scenario” on page 303.

Note: The sample code for MBean implementations is available for downloads. Refer to Appendix E, “Additional material” on page 399 for guidance on downloading the sample code.

4. After completing the development of MBean implementation, add the MBean Java project as a Utility JAR in the PX21_PRS_SPLR_IMSApp enterprise application project. In the Project Explorer, expand the **PX21_PRS_SPLR_IMSApp** project. Open the **Deployment Descriptor: PX21_PRS_SPLR_IMS** file. In the deployment descriptor window, select the **Module** tab. The window should look like Figure 8-28.

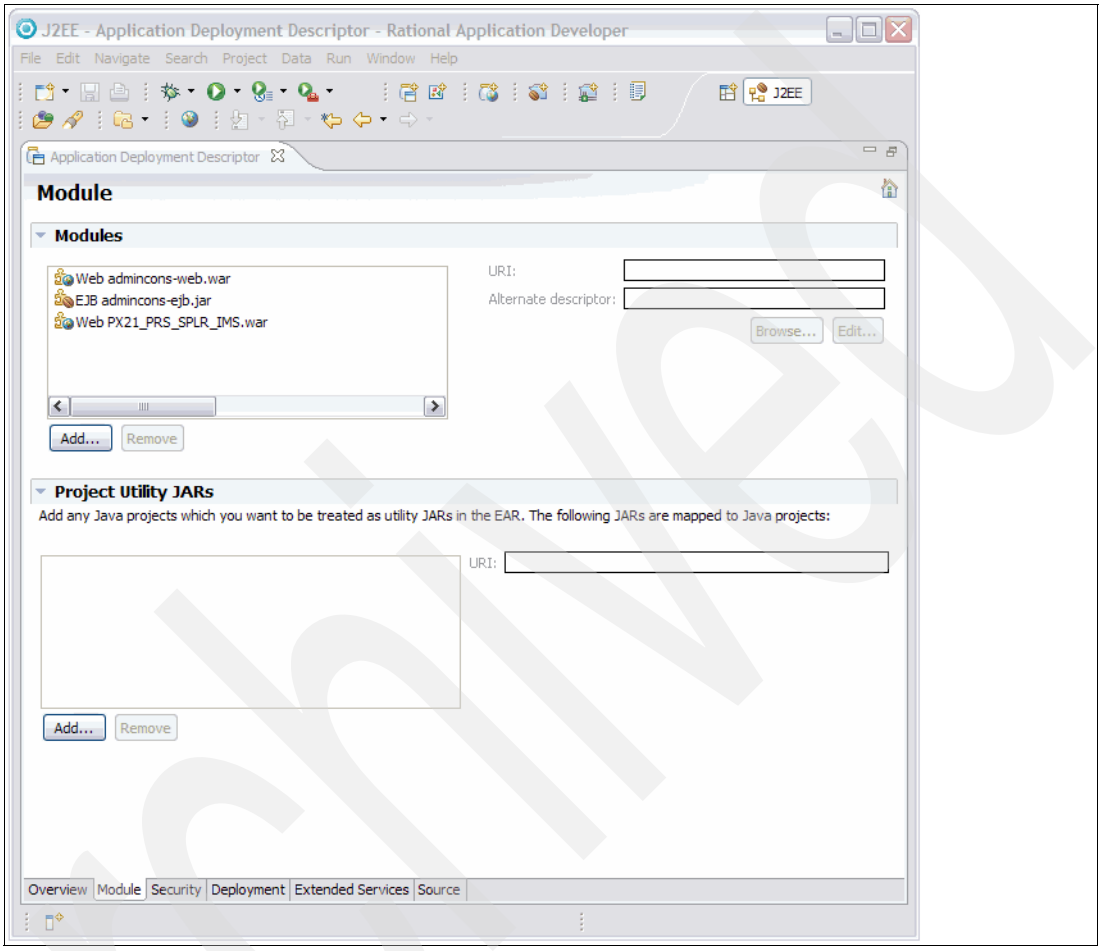


Figure 8-28 Module configuration in Enterprise Application Deployment Descriptor

Archive

5. In the Project Utility JARs section, click the **Add...** button. The Add Utility JAR dialog is displayed. Select the project entry **PX21_PRS_SPLR_IMSMBean** in the dialog and click **Finish**. Save the changes to the deployment descriptor. The window should look like Figure 8-29.

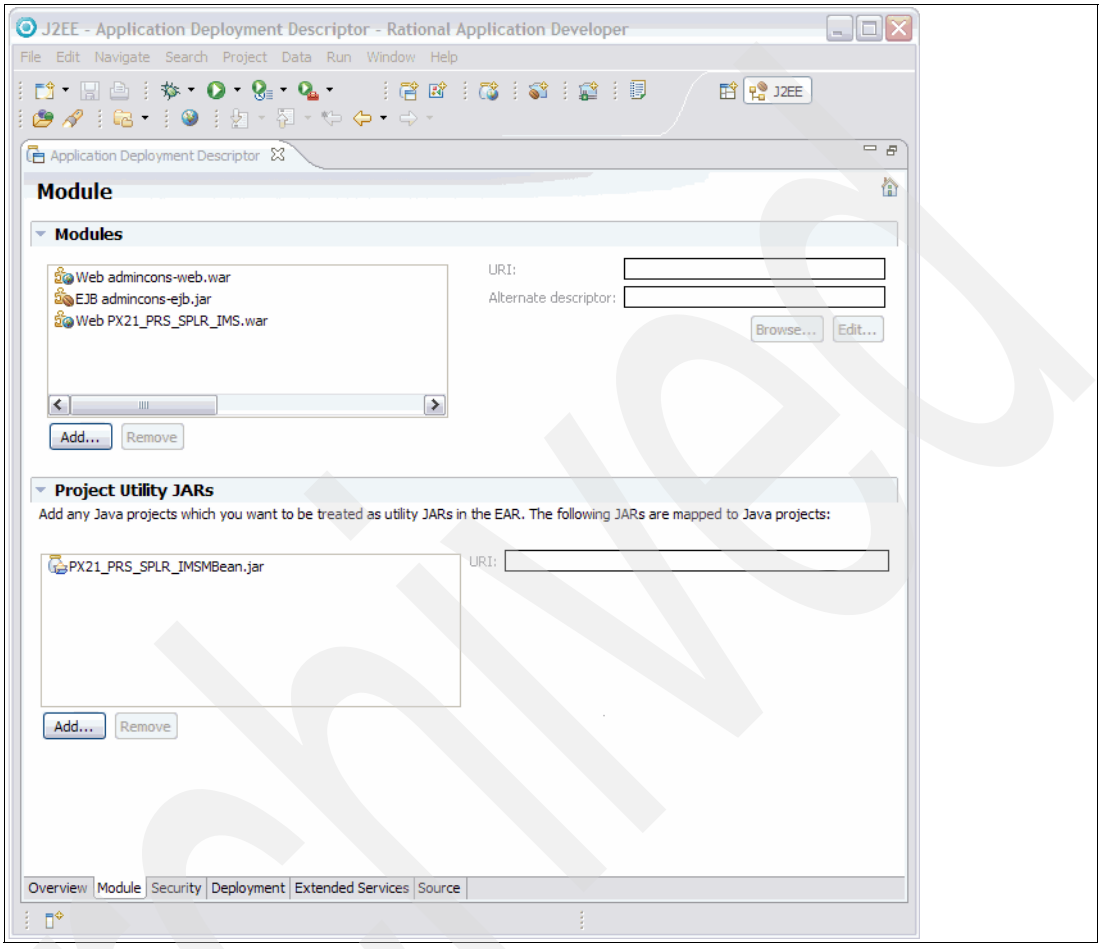


Figure 8-29 MBean Java project added to Project Utility JARs of service enterprise application

6. In the Project Explorer, expand the **META-INF** folder in the **PX21_PRS_SPLR_IMSApp** project. Select **admin.xma** and open it. This is a special deployment descriptor that is used by the IBM WebSphere Telecommunications Web Services Server administration console sub-system to render service MBeans. For more information, refer to 8.7.1, “Developing the JMX MBean for service” on page 300. The admin.xma should look like Figure 8-30 on page 325.

Archive

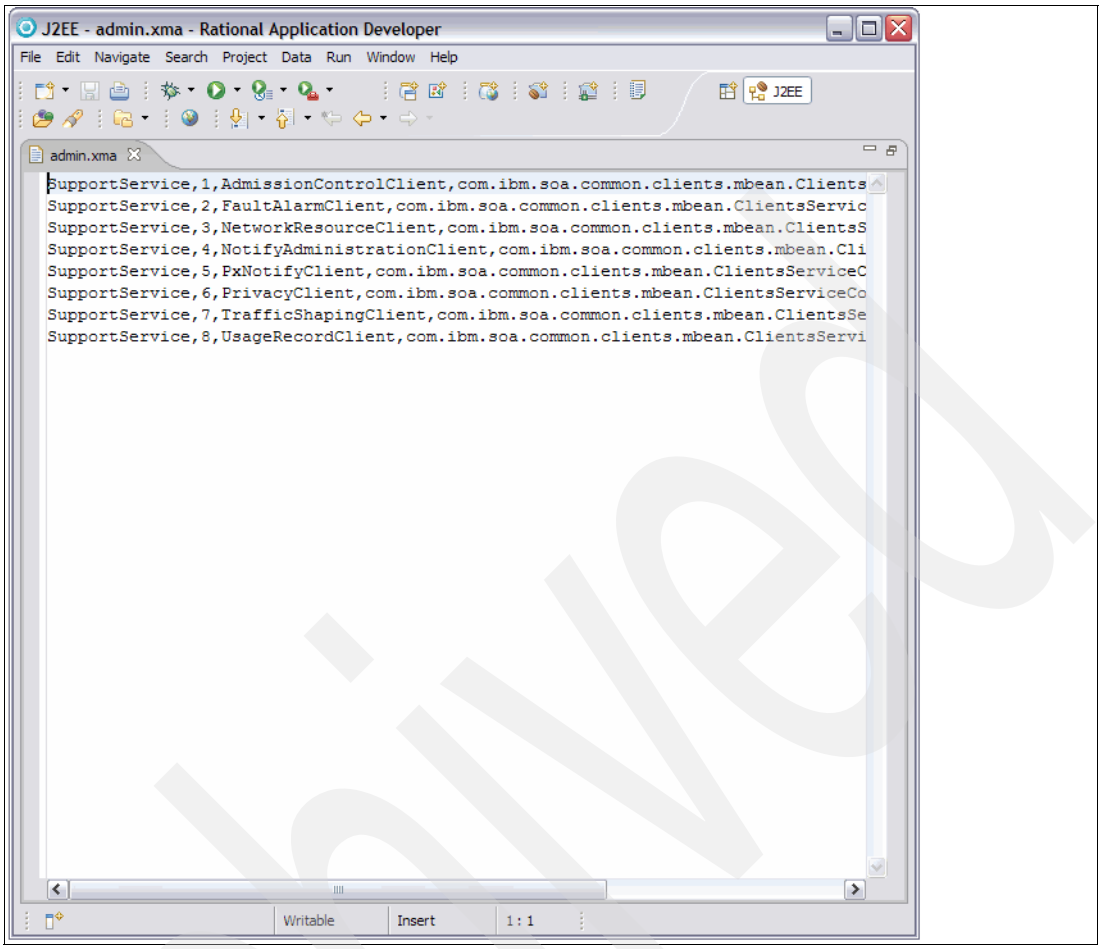


Figure 8-30 Deployment Descriptor for MBean implementation

7. At this point, the details of the new MBean ServiceComponent implementation created for the Parlay X Presence Supplier service should be entered in admin.xma along with the Service Category, Service name, and JAR file name of the MBean implementation. Additionally, the admin.xma file should be edited to reflect all the referenced common component service specific ServiceComponent implementations as the SupportService category. The entry should look like the text below:

   Service,7,PresenceSupplierIms,com.ibm.twss.parlayx21.presence.supplier.mbean.PresenceSupplierServiceComponent,PX21_PRS_SPLR_IMSMBean.jar

8. The ‘admin.xma’ file should look like Figure 8-31 on page 326.

Note: Note that the entries specific to common components, that is, NotifyAdministrationClient and PxNotifyClient, are removed from admin.xma. As the implementation of the Parlay X Presence Supplier service has limited functionality, the common components are not required. We recommend following this approach for any custom IBM WebSphere Telecommunications Web Services Server service implementation.

Entries of unutilized or unreferenced common components should not be included in admin.xma.

Archive

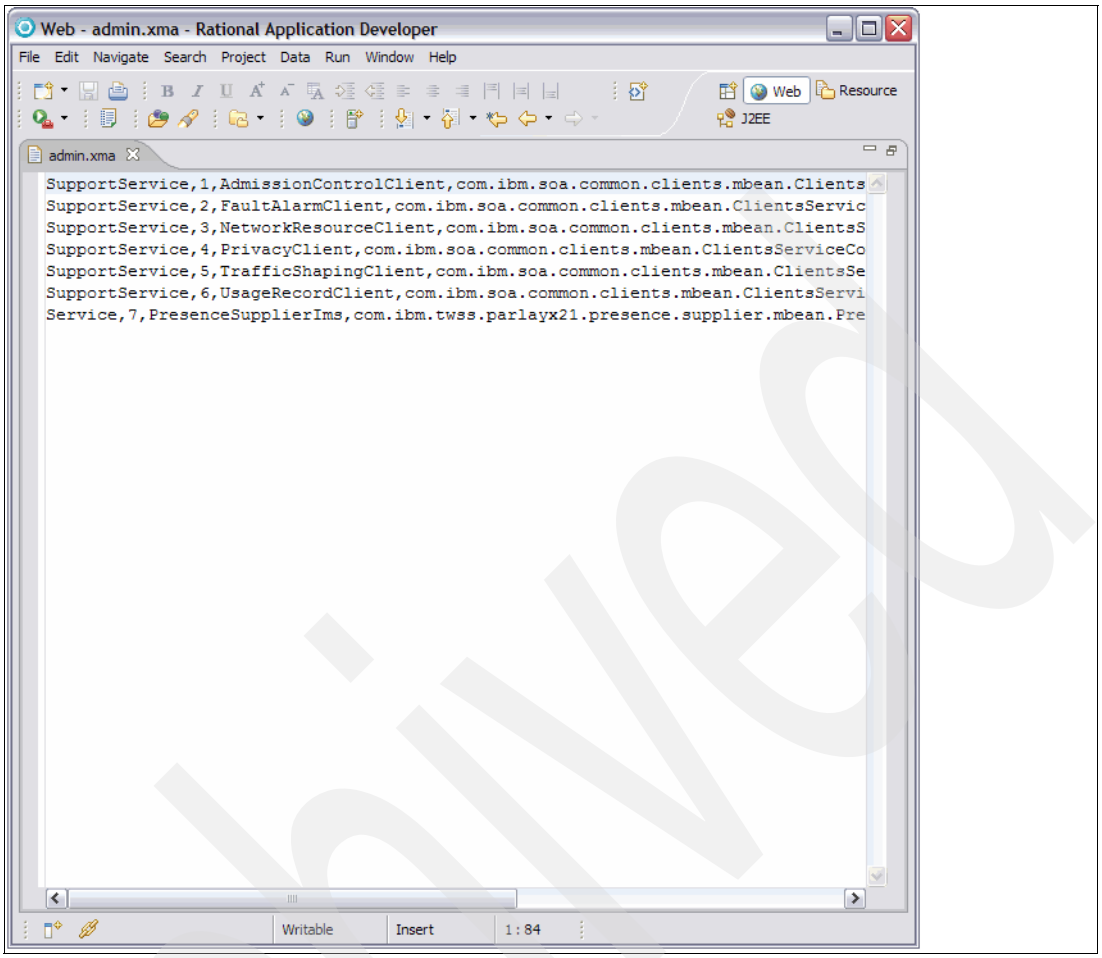


Figure 8-31 ServiceComponent class of Parlay X Presence Supplier service added to admin.xma

This concludes the development steps for creating and bundling MBean and ServiceComponent implementations and the other required classes for the Parlay X Presence Supplier service.

8.8 Deploy and test

In the previous sections, we discussed the procedure to create a service implementation and MBean implementation for a service. In this section, we discuss the deployment procedure of a custom service implementation.

8.8.1 Deployment procedure

A custom service implementation is packaged as an enterprise application archive (EAR). The deployment of a service project is typically done using the WebSphere Application Server administration console. Follow these steps to deploy a service implementation to the IBM WebSphere Telecommunications Web Services Server run time:

1. Export the Enterprise Application project that we created for the Parlay X Presence Supplier service implementation. Select the **PX21_PRS_SPLR_IMSApp** project. Right-click and select the **Export'** menu item in the pop-up menu. Select the **EAR file** menu item. This action opens the Export dialog (Figure 8-32 on page 327). Enter a valid

folder followed by the file name. The file name should be the same as the application name and the file extension should be .ear.

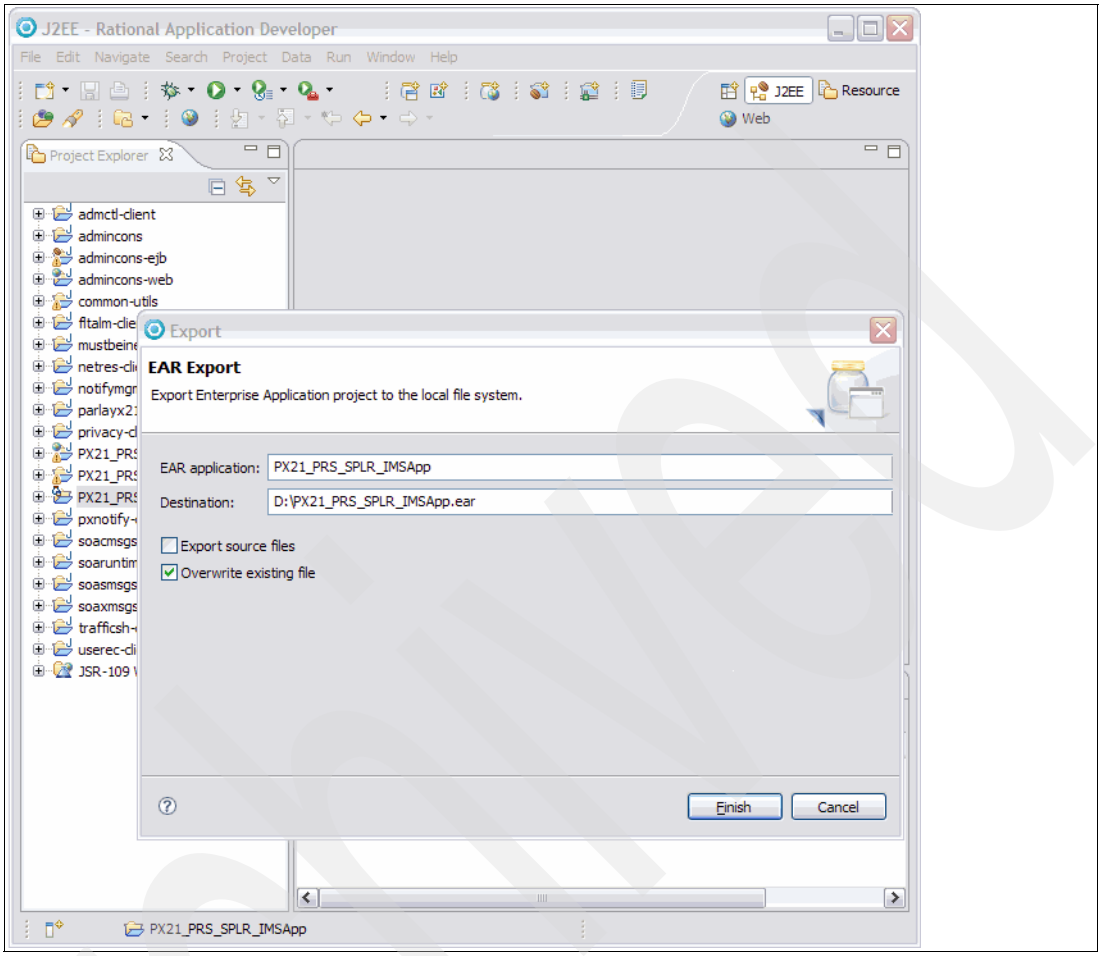


Figure 8-32 Export the service implementation enterprise application archive (EAR)

2. The first step to deploy a service implementation enterprise application archive (EAR) is to access the Web administration console of WebSphere Application Server, commonly known as the Integrated Solutions Console. Log in to the administration console of the WebSphere Application Server instance or deployment manager (in the case of a clustered environment), where the Service Platform components are installed. Expand the **Applications** menu on the left panel and select **Install New Application** (Figure 8-33).

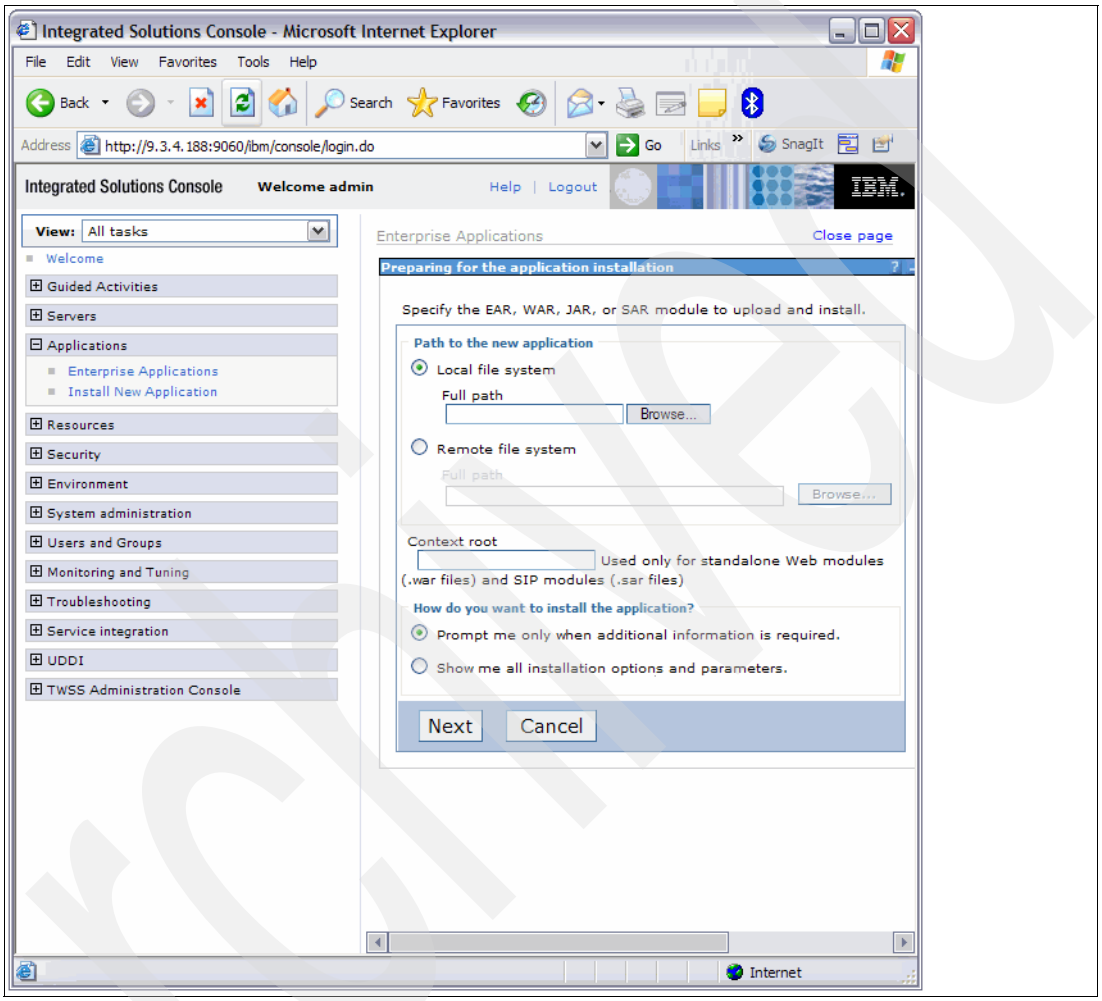


Figure 8-33 Accessing the Installation wizard of Integrated Solutions Console

3. Click the **Browse** button in the right panel of the administration console. This action opens a file selection dialog. Select the **PX21_PRS_SPLR_IMSApp.ear** file from the file system. Click the **Next** button. The Installation Options Page is displayed. Leave the default settings and click the **Next** button (Figure 8-34).

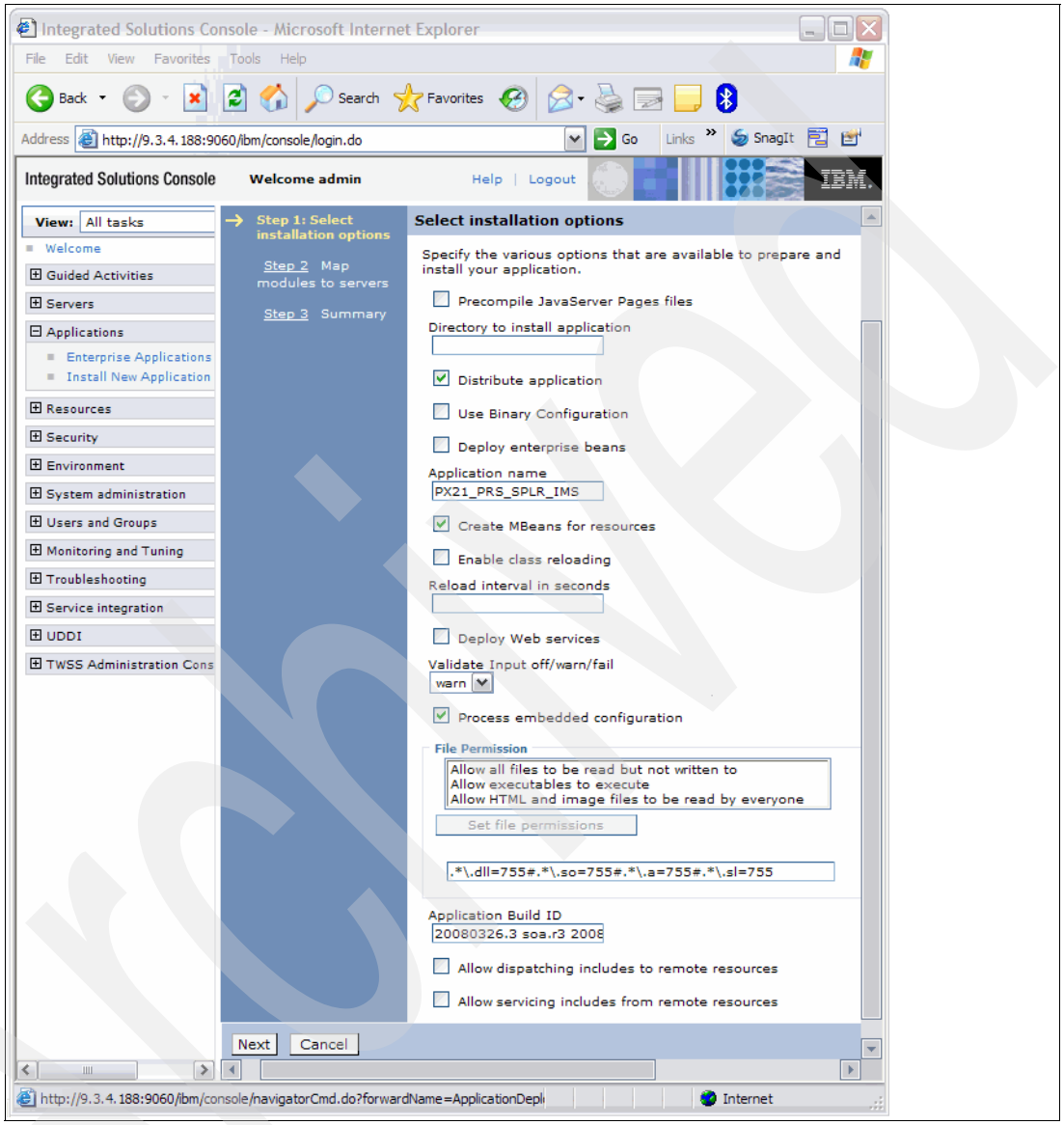


Figure 8-34 Installation Options page for service enterprise application

Archive

4. Click the **Next** button in the Map Modules to servers page. Click the **Next** button. In the Summary page, click the **Finish** button. The installation confirmation page is displayed (Figure 8-35). Click the **Save** link on this page. The installation of the enterprise application is completed.

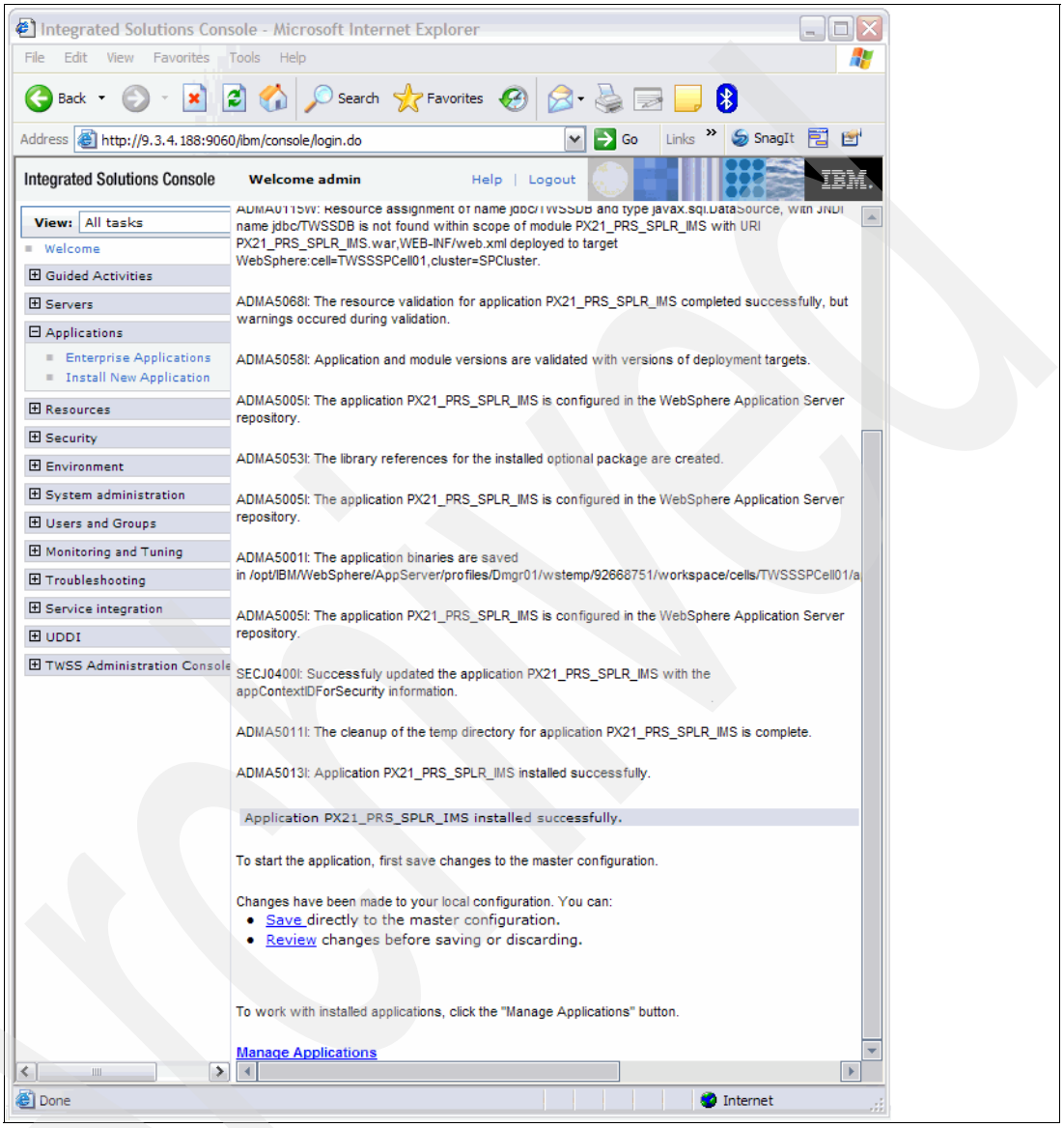


Figure 8-35 Installation confirmation page for service enterprise application

5. After installation, the service enterprise application should be started. Expand **Applications** and then click the **Enterprise Applications** menu on the left panel of the administration console. On the right panel, the enterprise applications hosted by the application server instance are displayed with check boxes for each application entry. Locate the PX21_PRS_SPLR_IMS application. Select the check box and click the **Start** button to start the application.

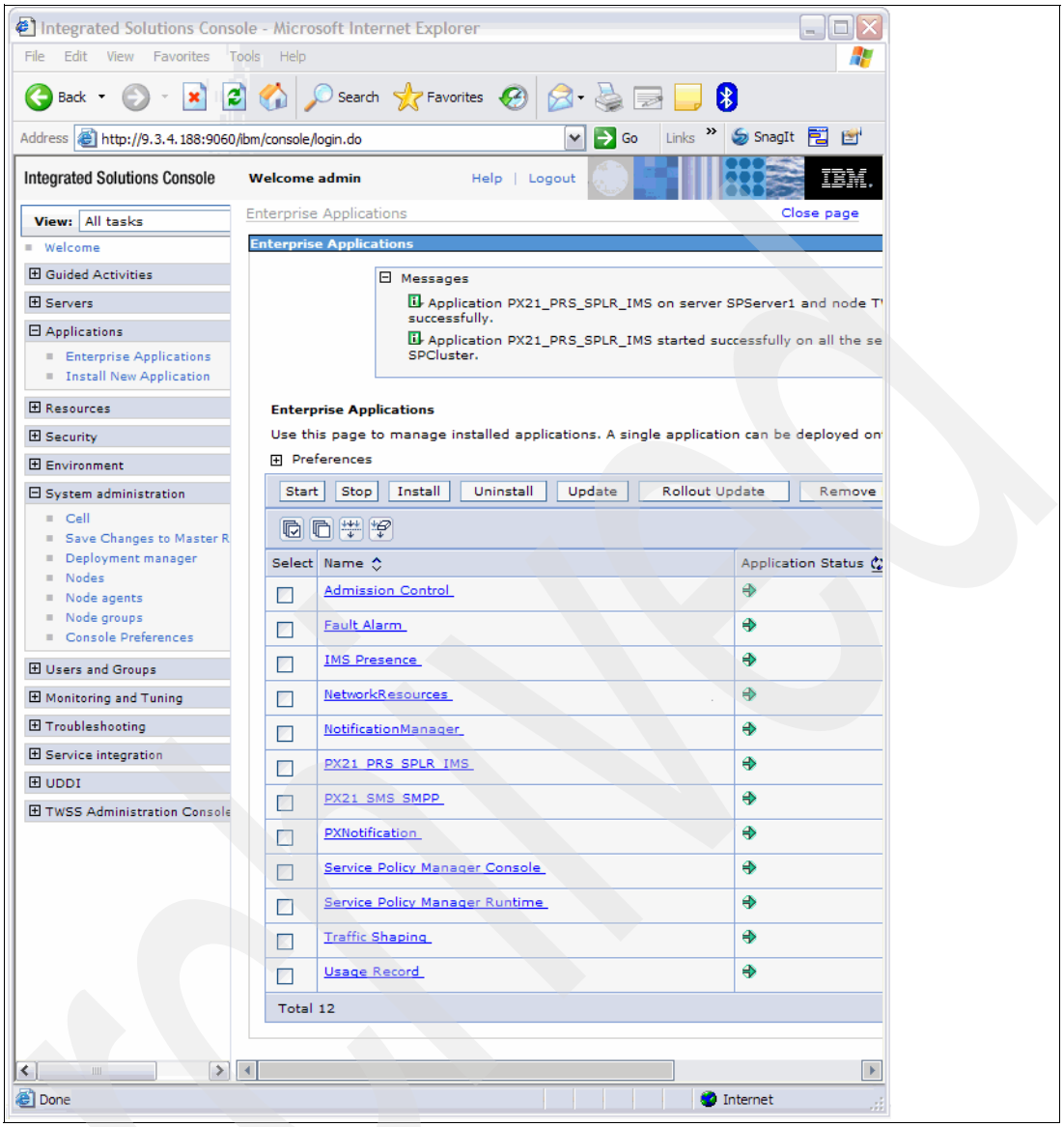


Figure 8-36 Parlay X Presence Supplier service application started

At this time, the service enterprise application should start without any errors.

8.8.2 Service administration

In the previous section, we discussed the deployment procedure. After a successful deployment, we will now see the administrative procedures pertaining to IBM WebSphere Telecommunications Web Services Server services. The relevant guidelines are listed in the sections below.

Configuring the common components

1. The first administrative step is to configure the endpoints of all common component services for the Parlay X Presence Supplier service. On the WebSphere Administration Console, locate the TWSS Administration Console menu item on the left panel and expand it. Click the **Web Services** link. The right side panel displays a list of IBM WebSphere Telecommunications Web Services Server services (Figure 8-37).

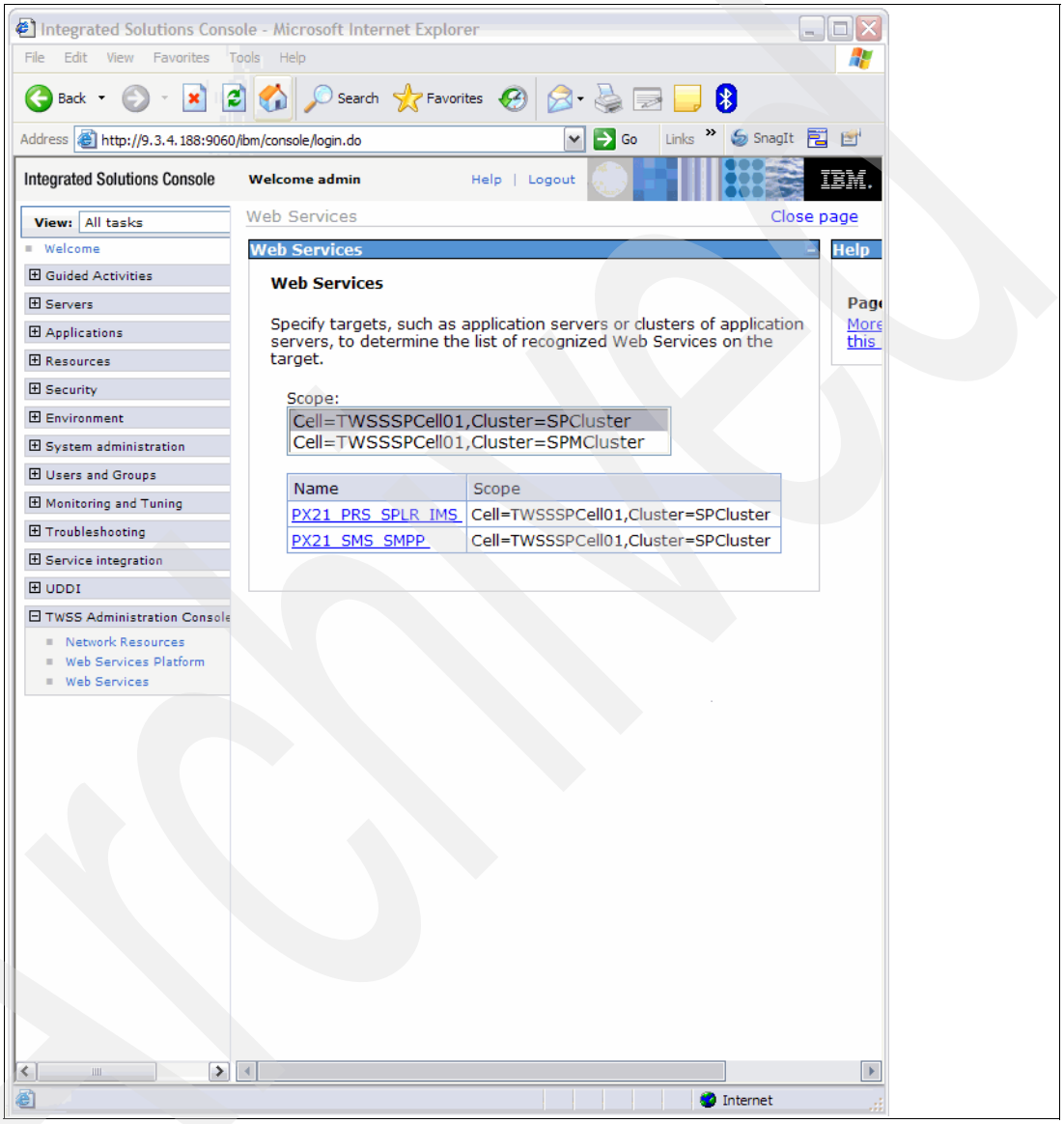


Figure 8-37 Available list of IBM WebSphere Telecommunications Web Services Server services

Archived

2. Locate the PX21_PRS_SPLR_IMS link and click it. The details page comprising the main service, the IMS Presence Supplier Web service link, and a list of supported services or the common component services are displayed in the page (Figure 8-38).

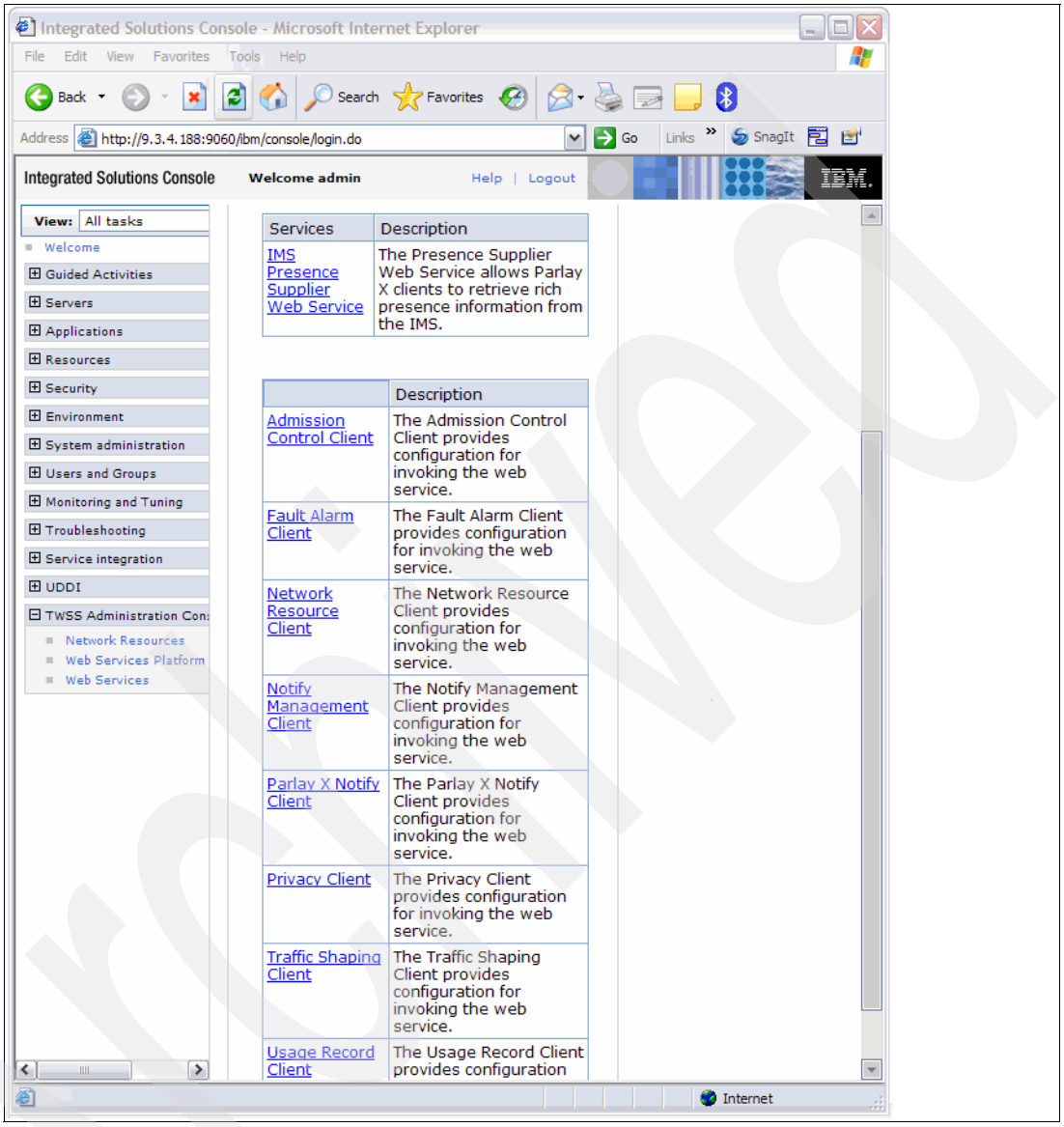


Figure 8-38 Page with Presence Supplier service and common component service links

Archive

3. Click the **Admission Control Client** link. This action should display the **General Properties** section that comprise admission control component attributes (Figure 8-39).

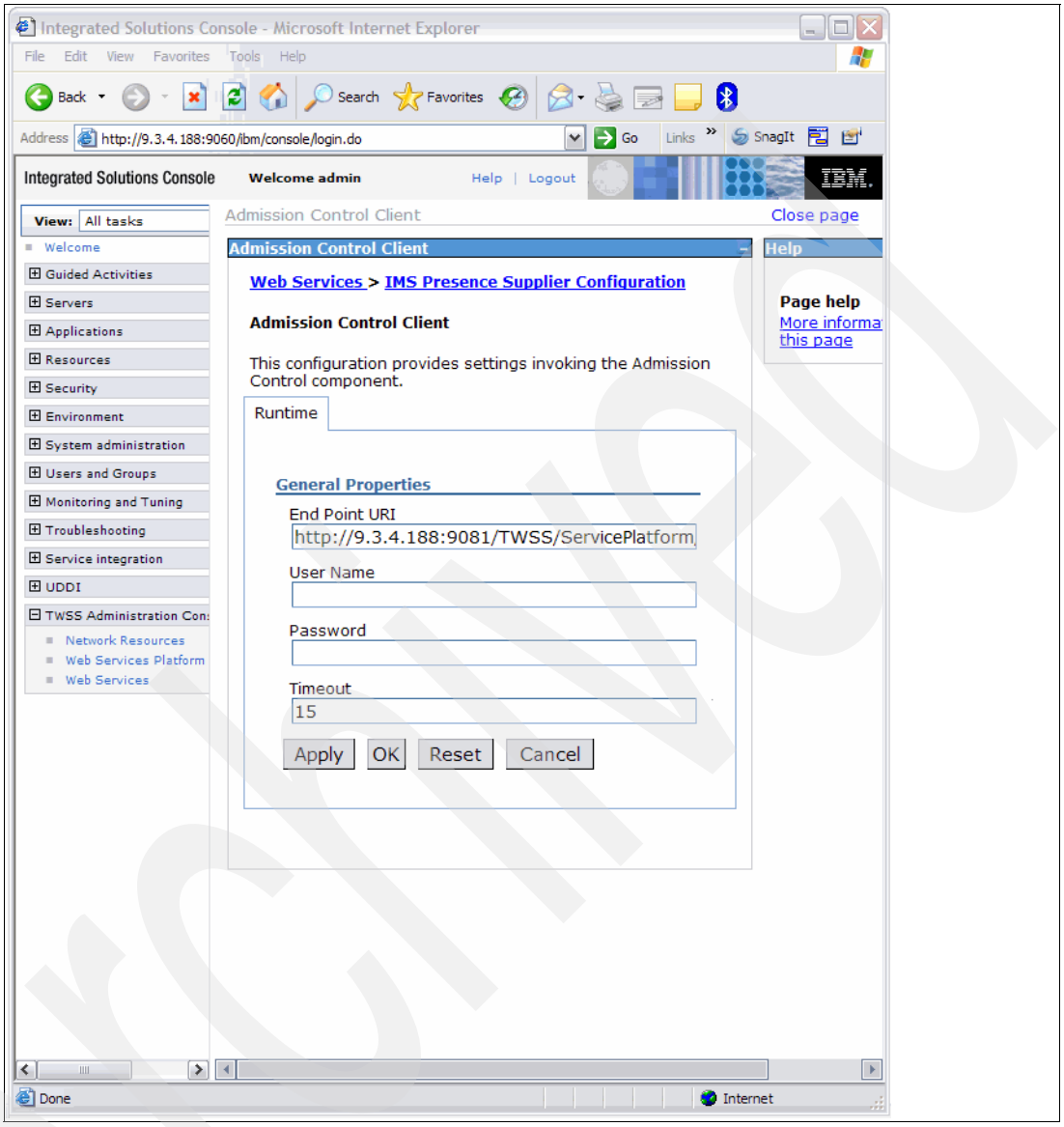


Figure 8-39 Admission Control Client properties setting for Parlay X Presence Supplier service

4. Locate the ‘End Point URI field value and validate that the Web service URL is valid. The easiest way to validate a Web service URL is to direct a Web browser to the URL. If the Web service URL is valid, you should see the window shown in Figure 8-40 on page 335.

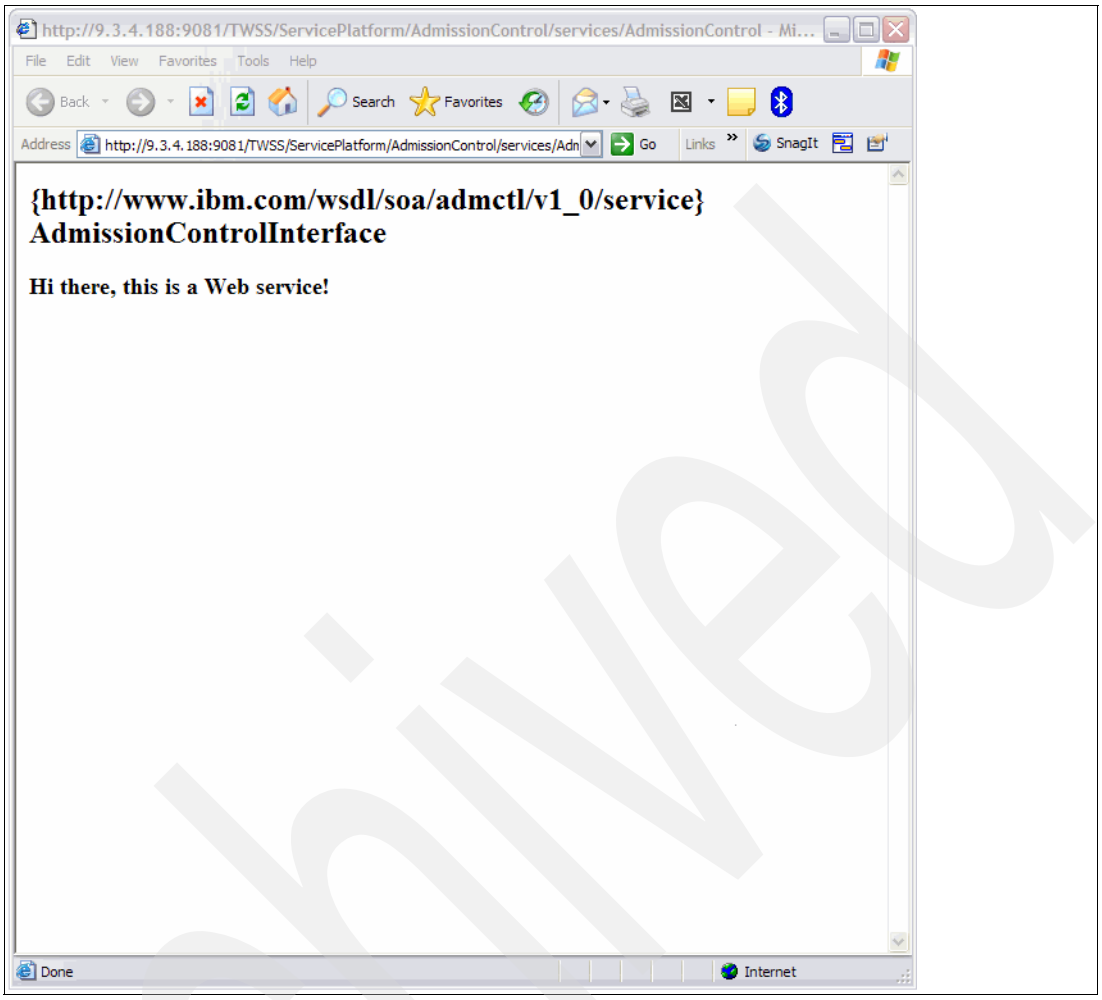


Figure 8-40 Simple step to validate a Web service URL

5. If the URL is not valid, typically the virtual host and port number have to be verified. Fix the the virtual host or host name and port values in the configuration settings.
6. Follow steps 3 through step 5 for each of the common component service URLs and make sure that the individual web service URLs are valid.

Note: If a specific common component invocation is *not* required, you must leave the End Point URI field value of the specific common component *empty*.

Archived

Scenario MBean implementation

In this section, we will walk through the configuration settings specific to our sample scenario, the Parlay X Presence Supplier service:

1. Authenticate to the WebSphere Application Server administration console where the Service Platform components are installed. On the left panel, locate and click the **TWSS Administration Console** link. The window should resemble Figure 8-41.

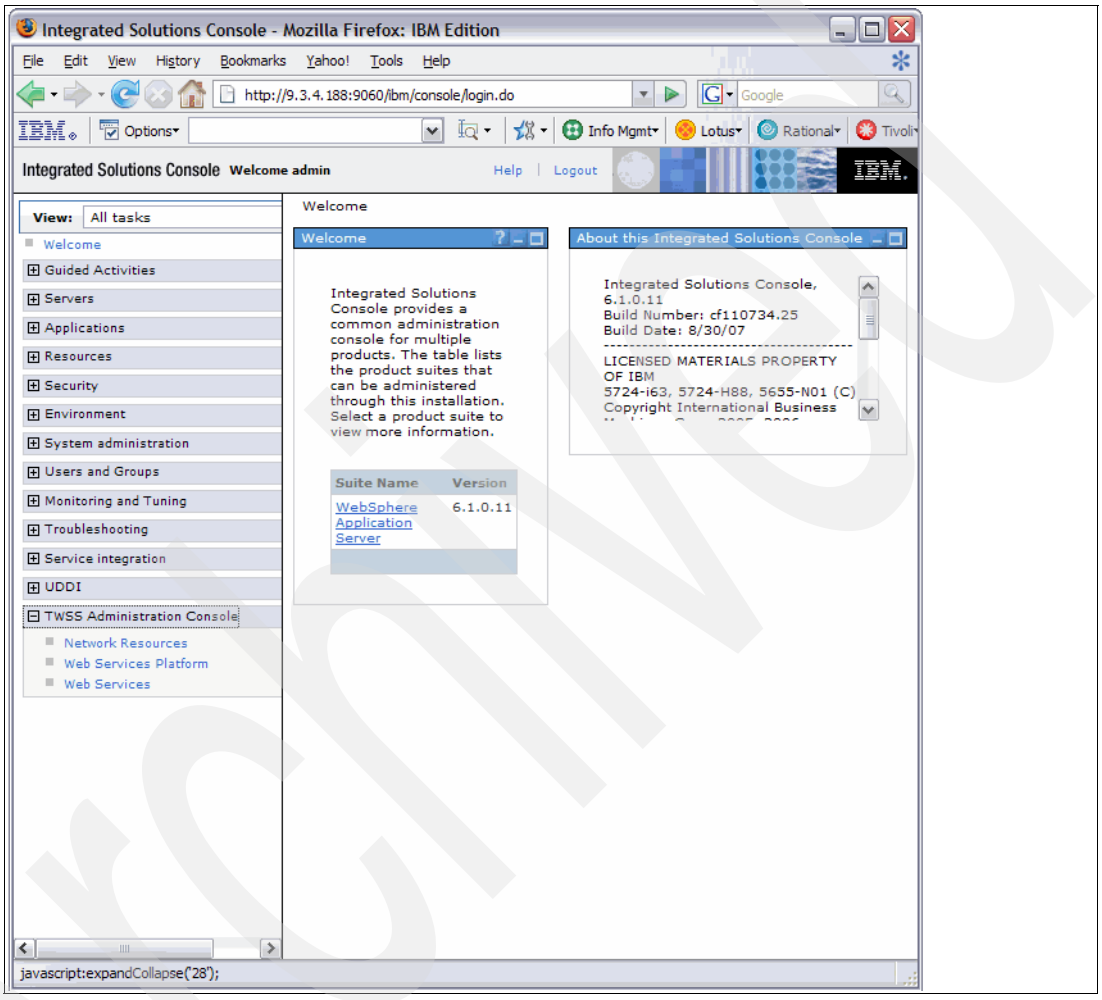


Figure 8-41 TWSS Administration Console link on WebSphere Application Server ISC

Archive

2. Click the **Web Services** link. In the right panel, a list of links representing the installed services are displayed. Locate and click the link **PX21_PRS_SPLR_IMS**. This action opens the main configuration Web page for the sample scenario, the Parlay X Presence Supplier service (Figure 8-42).

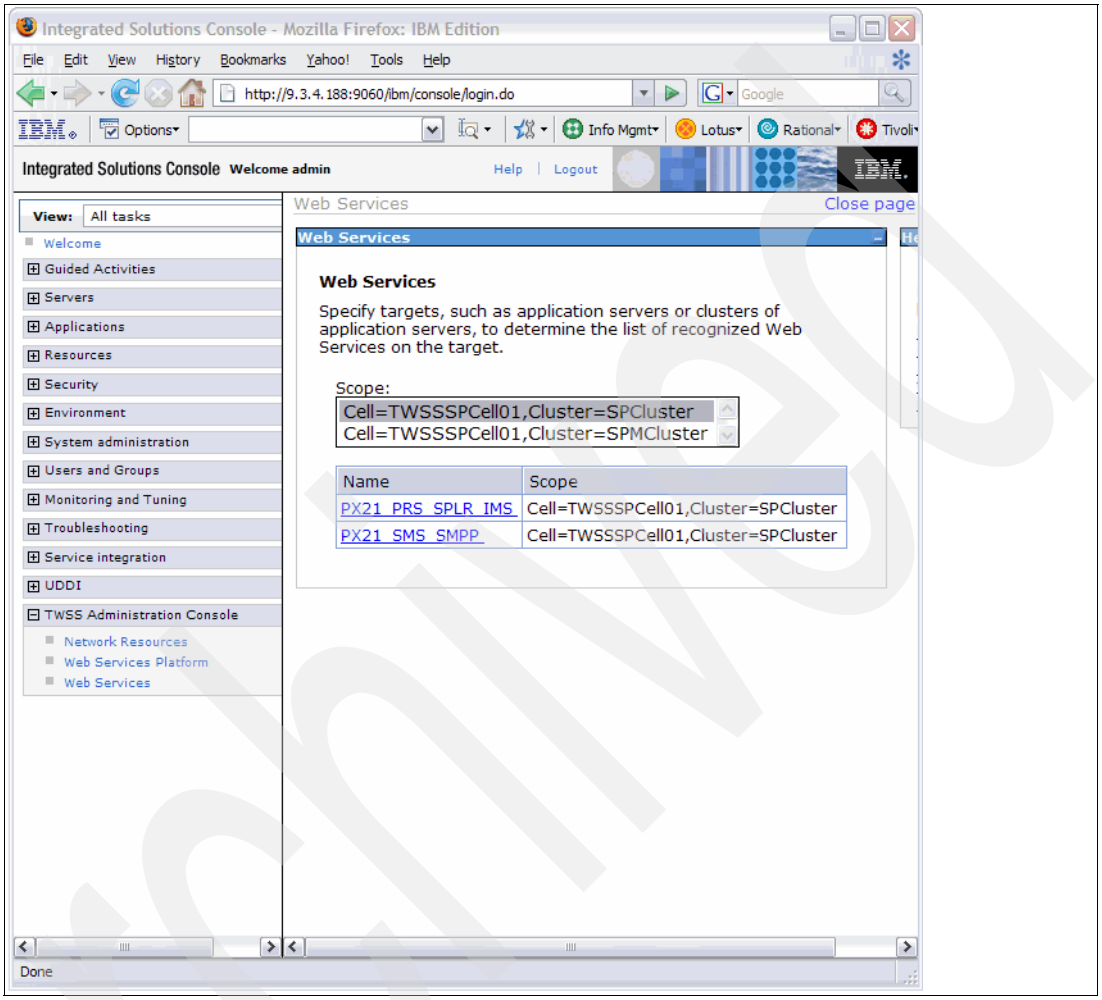


Figure 8-42 The services page in IBM WebSphere Telecommunications Web Services Server administration console

Archive

3. In the main page, locate the **IMS Presence Supplier Web service** link and click it. This action opens the configuration page for our sample scenario (Figure 8-43).

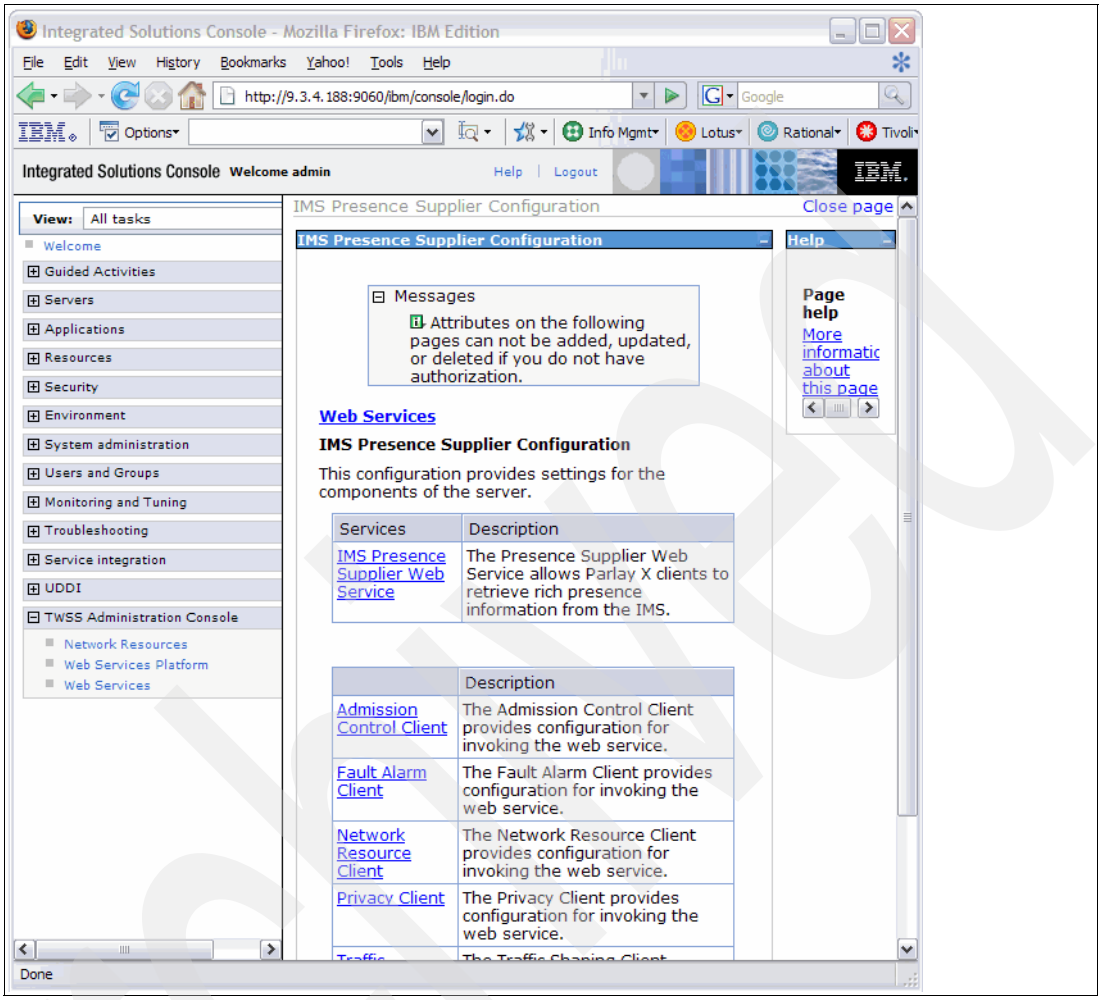


Figure 8-43 The Presence Supplier service administration page

4. Notice the text fields displayed in this configuration page. The text fields correspond with the MBean attributes created during the MBean implementation. The user-friendly labels are retrieved from the ConsoleResources.properties resource bundle. The window should look like Figure 8-44 on page 339.

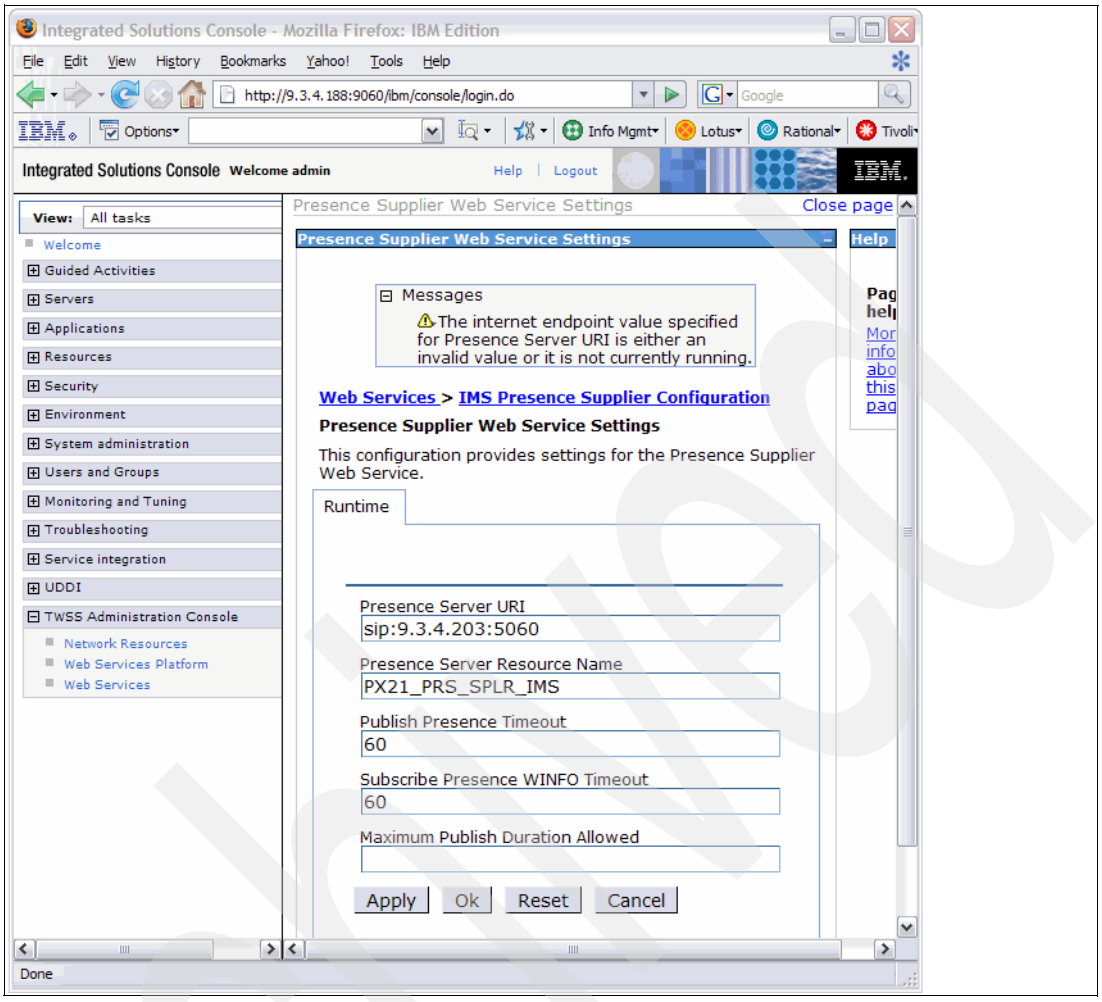


Figure 8-44 The Presence Supplier service configuration settings page

5. Optionally, change the default configuration values depending on your requirement. Click the **OK** or **Apply** button to save the changes. This action invokes the IBM WebSphere Telecommunications Web Services Server Administration console sub-system to call the service MBean implementation code and eventually persist the modifications to the data store.

This concludes the deployment and configuration settings for the Parlay X Presence Supplier service implementation. In the next section, we will discuss the test client sample application and the test execution details.

8.8.3 Test client

In this section, we will discuss how to code a client application for our sample scenario. The development environment used is Rational Application Developer V7.0. The sample client application comprises a Web UI based on the Java Server Faces (JSF) framework and the Web service client based on the JAX-RPC proxy implementation. The following sections discuss in detail the essential programming steps required to create a Web service client and test it in a Rational Application Developer environment.

Note: The following sections only discuss the Web service client generation. A few screen captures of a Web based application are provided in order to make understanding the test execution easier. The Web UI code is available as part of the sample code (refer to Appendix E, “Additional material” on page 399).

Note that the client applications accessing IBM WebSphere Telecommunications Web Services Server need not be Web based and need not necessarily follow the JSF programming model.

Test environment setup

Open the Rational Application Developer IDE with a new workspace. For the sample Web service client, we create a Web project. In our sample, we will start with the creation of a Enterprise Application Project. Follow these steps:

1. Select **File** → **New** → **Project...**. The New Project wizard is displayed. In the wizard, locate and expand **J2EE**. Select **Enterprise Application Project**.

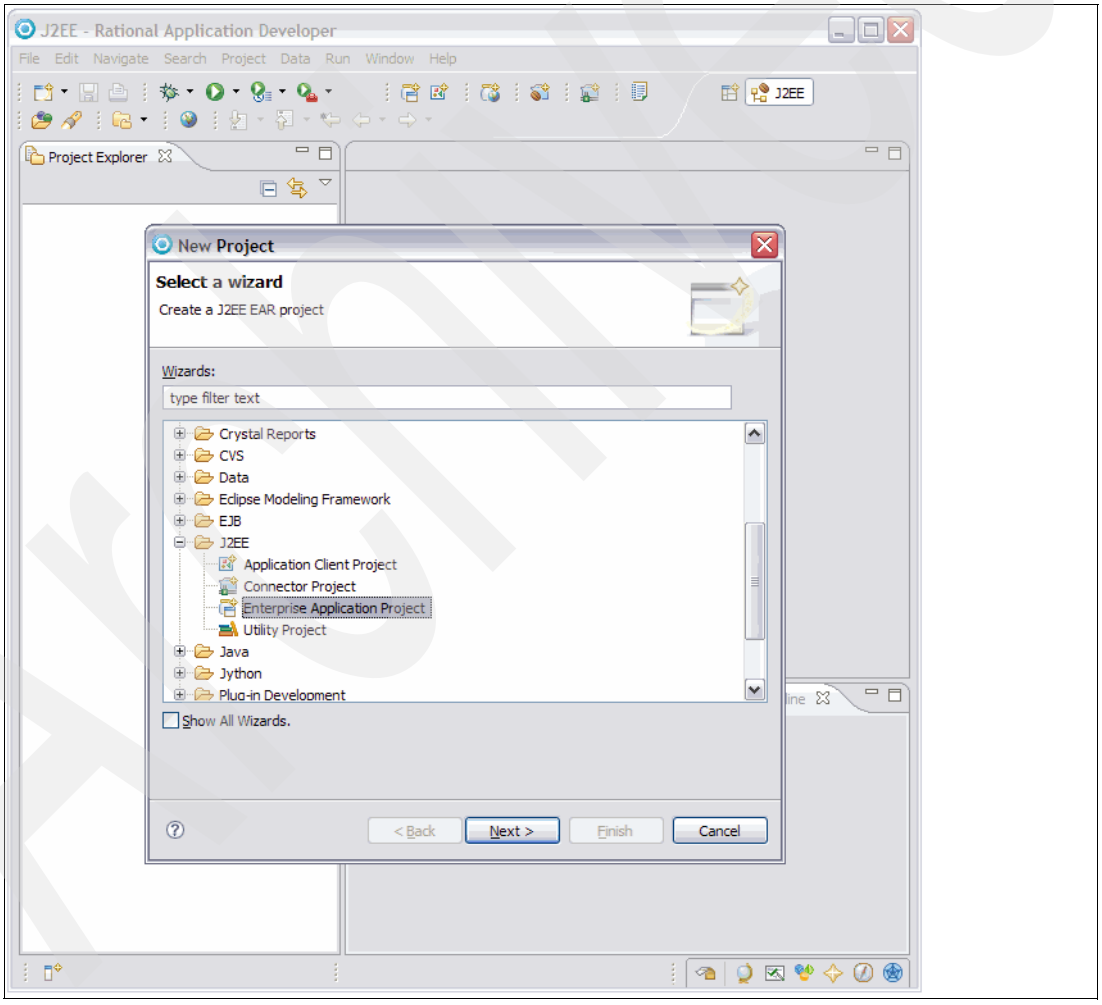


Figure 8-45 New Enterprise Application Project for sample client

2. Click the **Next** button. In the EAR Application Project window, type PresenceSupplierClientApp for the Project name field. Click the **Finish** button. At this stage, we do not require any other wizard options. The Project Explorer should resemble Figure 8-46.

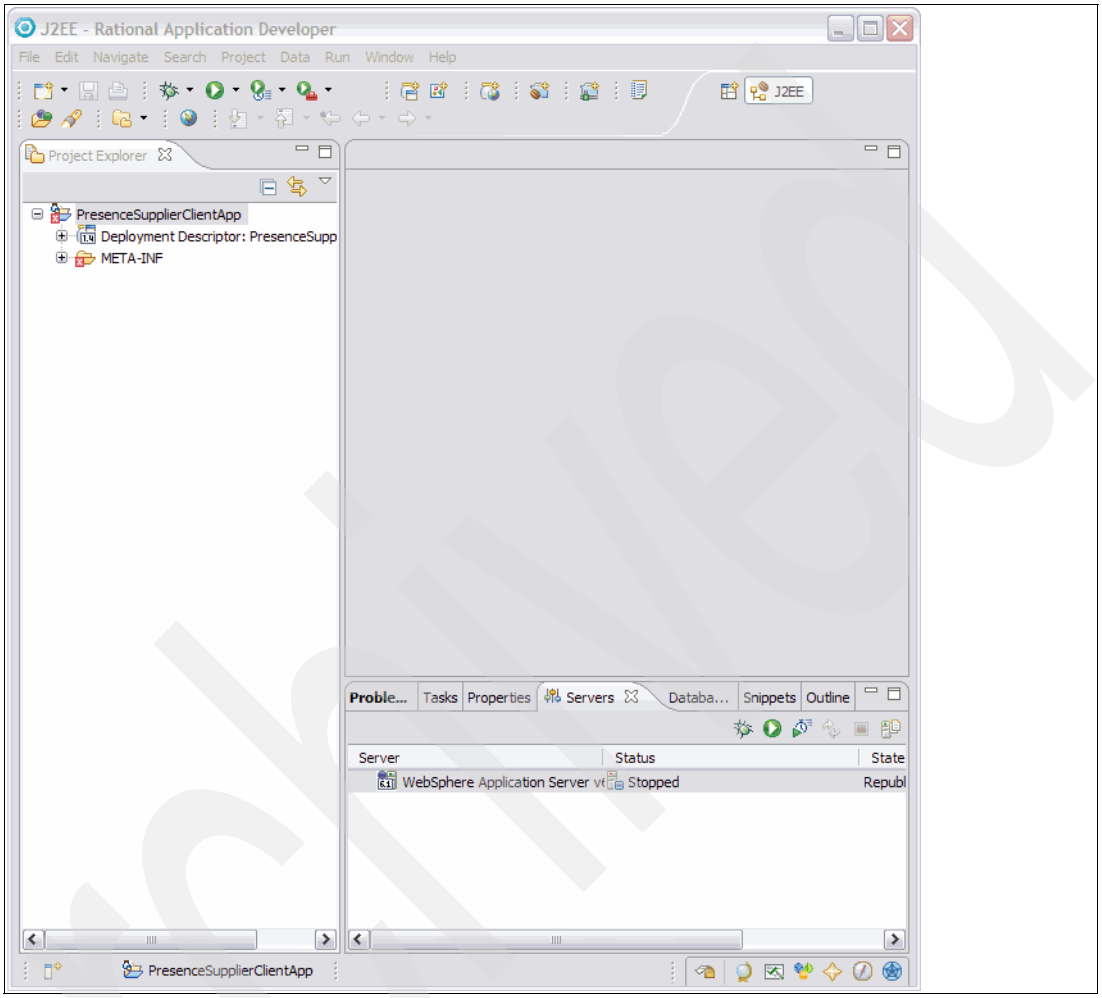


Figure 8-46 Initial view of the client Enterprise application project

3. Now we will create a Web Project that will house the Web service client application code. Right-click the **PresenceSupplierClientApp** project, select **New** from the pop-up menu, and select the **Dynamic Web Project** menu item. In the Dynamic Web Project wizard, enter PresenceSupplierClient for the field Project name. Leave the rest of the values at their defaults and click the **Next** button, as shown in Figure 8-47.

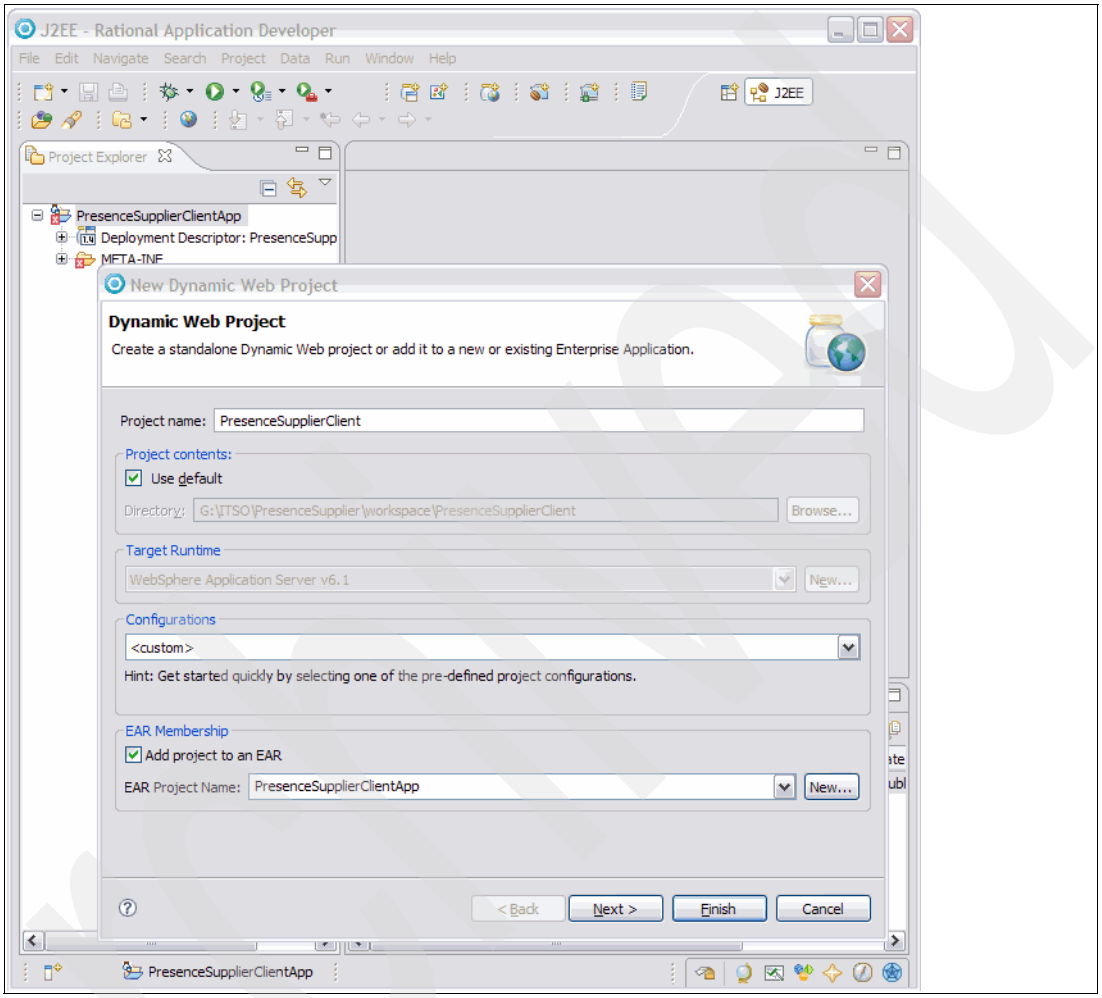


Figure 8-47 New Web Project for sample client application

4. In the Project Facets window, various features are enabled for this Web project. The sample client application comprises a JSF based Web UI. To enable the JSF features, the **Base Faces Support** and **JSTL 1.1** project facets should be selected. The window should resemble Figure 8-48

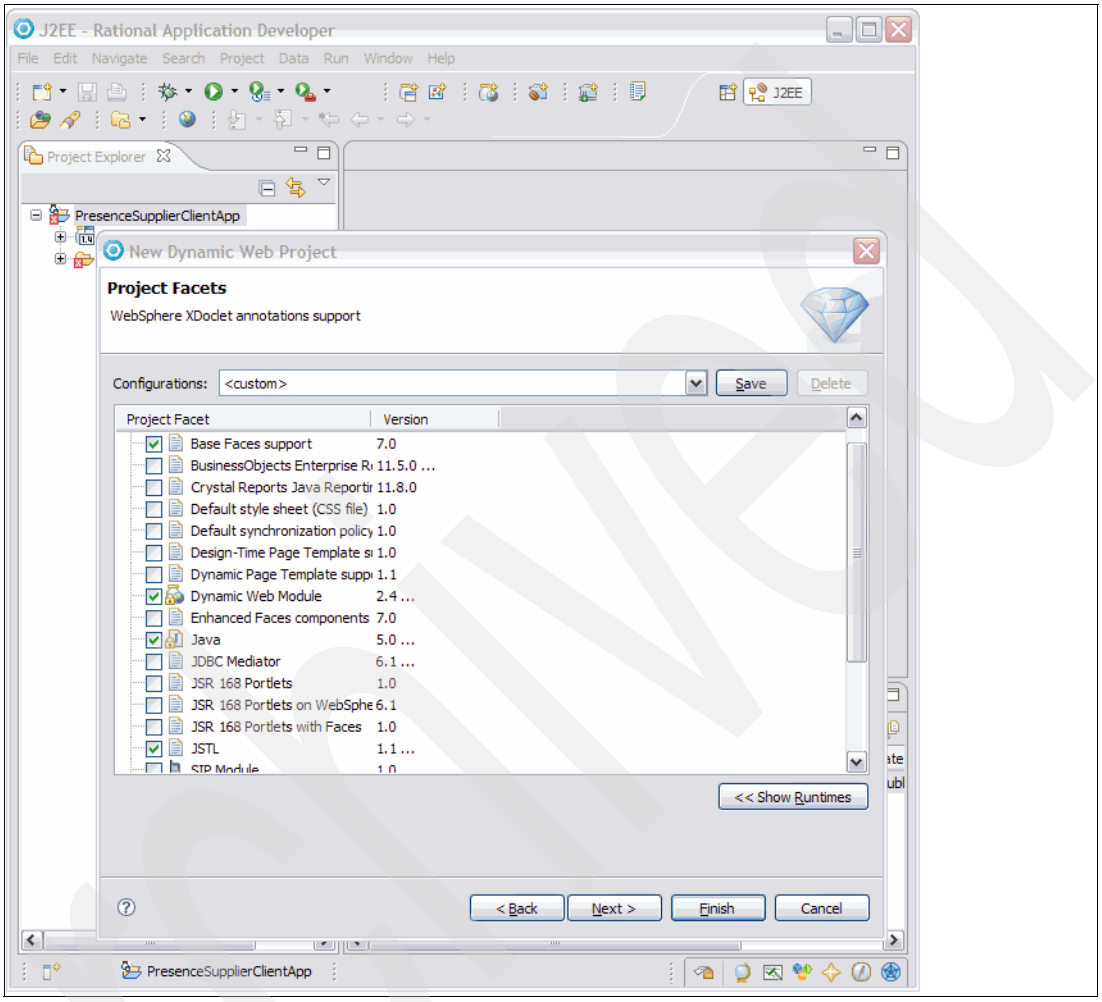


Figure 8-48 Web Project Facets selection for sample client application

5. Click the **Finish** button. Click the **No** button on the Open Web perspective pop-up dialog. Close the WebDiagram.gph as well. The Project Explorer should resemble Figure 8-49.

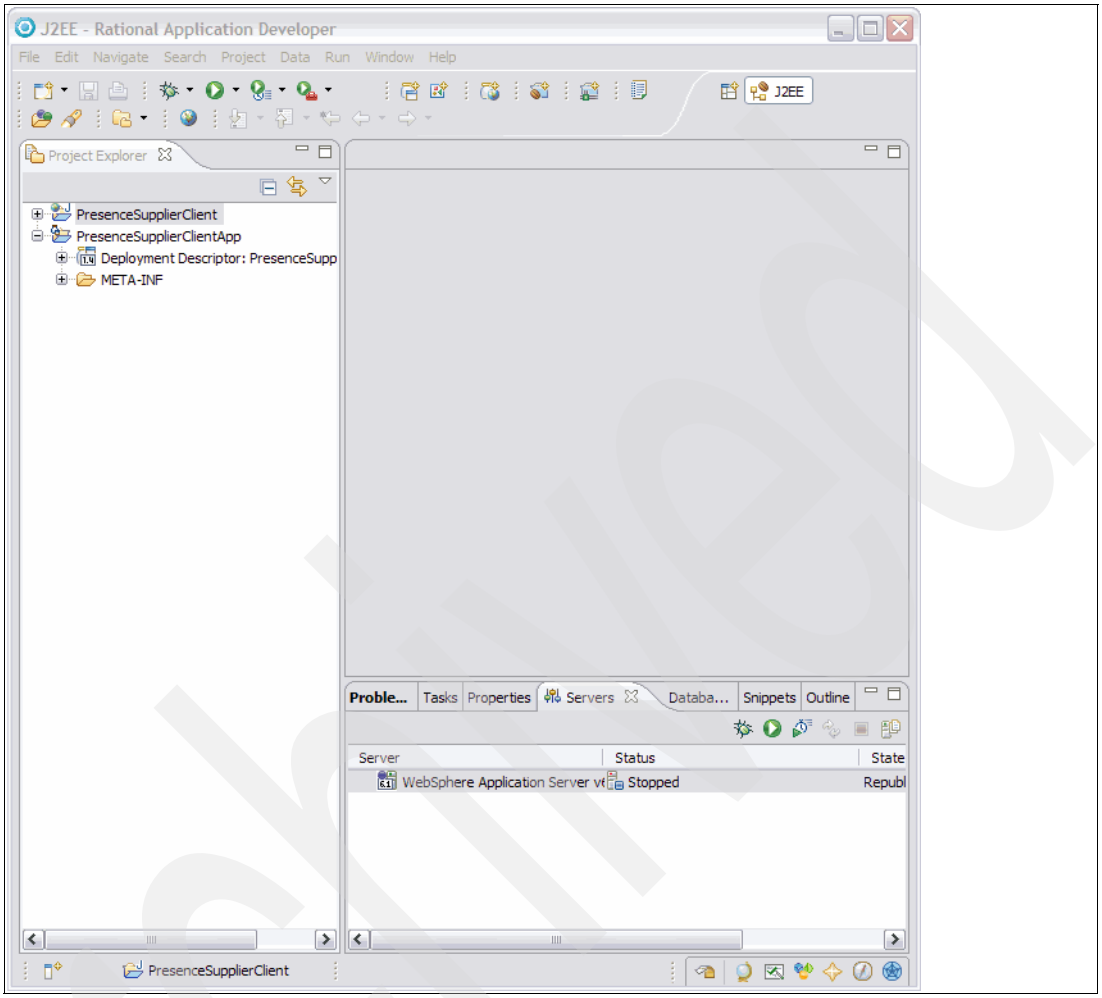


Figure 8-49 Sample client Web project

6. In the Project Explorer, expand the Web project **PresenceSupplierClient**, locate the Web Content folder, and expand it. Locate the WEB-INF folder, right-click it and select **New** in the pop-up menu, and then select the **Folder** menu item. The **New Folder** wizard is displayed. Enter wsdl for the Folder name field. The window should look like Figure 8-50 on page 345.

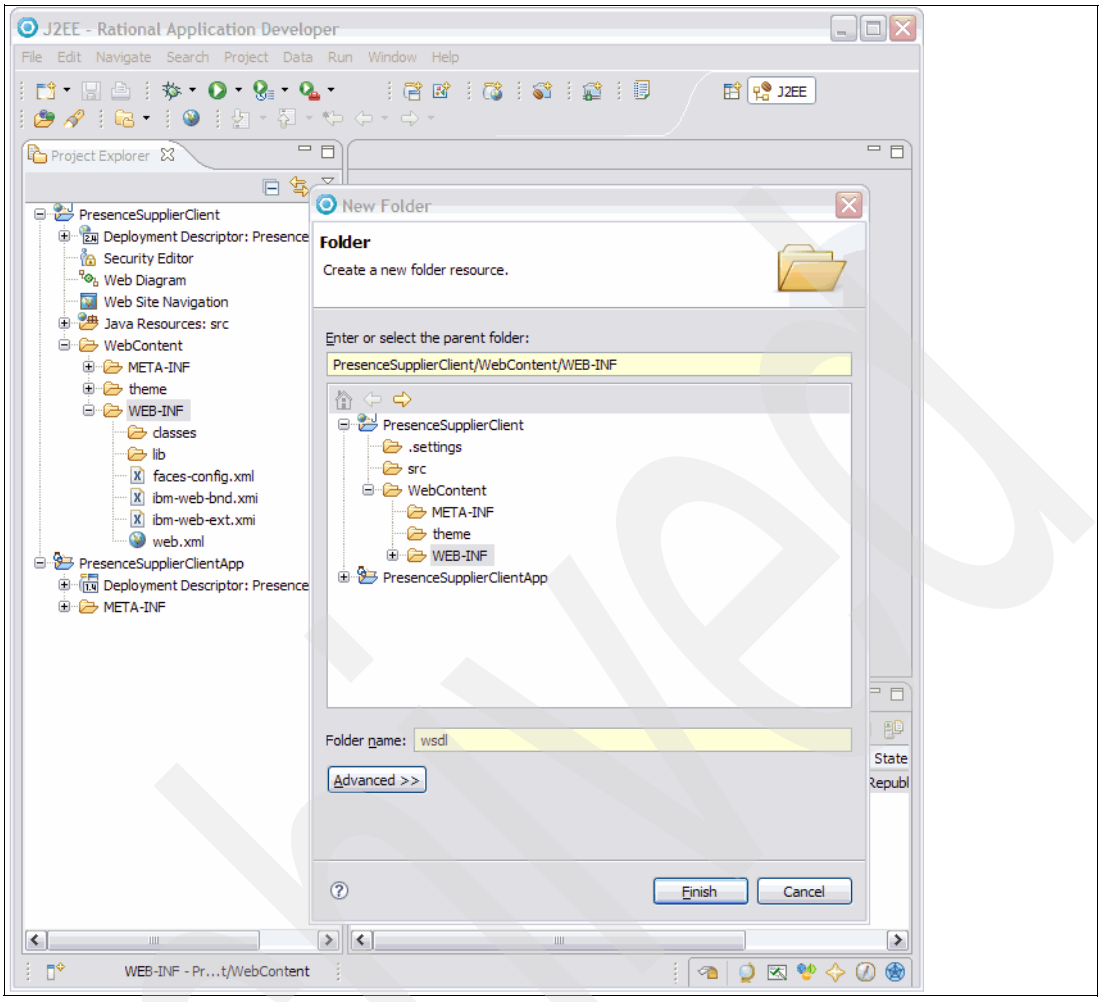


Figure 8-50 Creation of 'wsdl' folder in sample client Web project

7. In the Project Explorer, right-click the newly created **wsdl** folder and select **Import...**. Follow the steps described in 8.4, "Developing a Sample Parlay X Web service" on page 264 section to import the WSDL files and dependent XSD files.

Note: The sample client application should have at least the following WSDL and XSD files in the wsdl folder:

- px_cmn_f_2_0.wsdl
- px_cmn_t_2_1.xsd
- px_prs_si_2_3.wsdl
- px_prs_ss_2_3.wsdl
- px_prs_t_2_3.xsd

The Project Explorer should look like Figure 8-51.

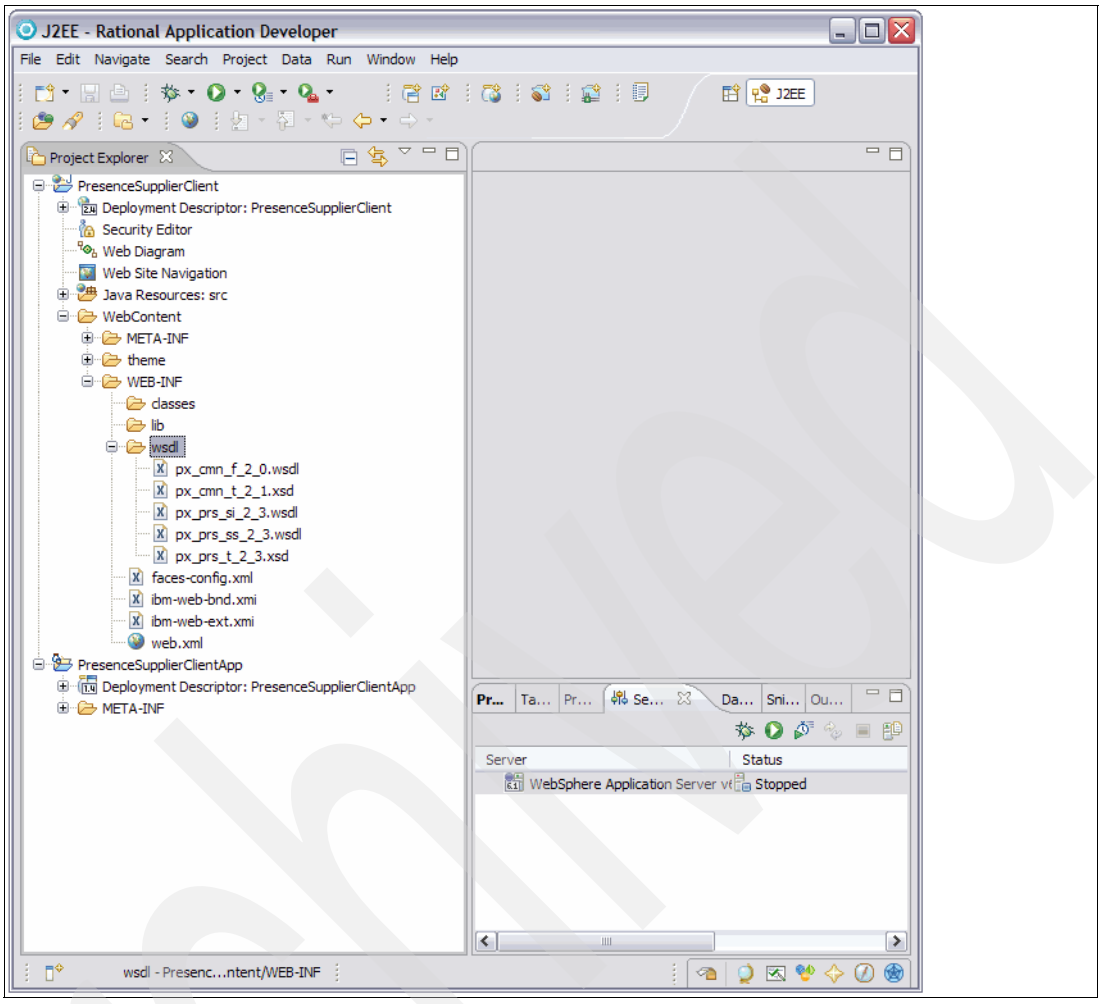


Figure 8-51 Parlay X Presence Supplier interface: WSDL and XSD files

8. Using the px_prs_ss_2_3.wsdl file, we will be able to create the Web Services Client code. Right-click **px_prs_ss_2_3.wsdl**, select **Web Services**, and click **Generate Client**. The Web Services Client wizard is displayed. Leave the default values in the first window. Click the **Next** button. Optionally, select **Define custom mapping for namespace to package**.

Note: If the package structure must conform to IBM WebSphere Telecommunications Web Services Server package conventions, select the **Define custom mapping for namespace to package** check box. This is considered an optional step for the client application.

Depending on the length of workspace folder structure, decide whether to use the IBM WebSphere Telecommunications Web Services Server package structure. For more information, refer to “Package naming conventions” on page 234. Follow the directions in 8.4.2, “Generating Web service bindings” on page 270.

Click the **Finish** button on the wizard. This action will start the client code generation in RAD. After completing the code generation, the expanded Java Resources folder in the Project Explorer should look like Figure 8-52.

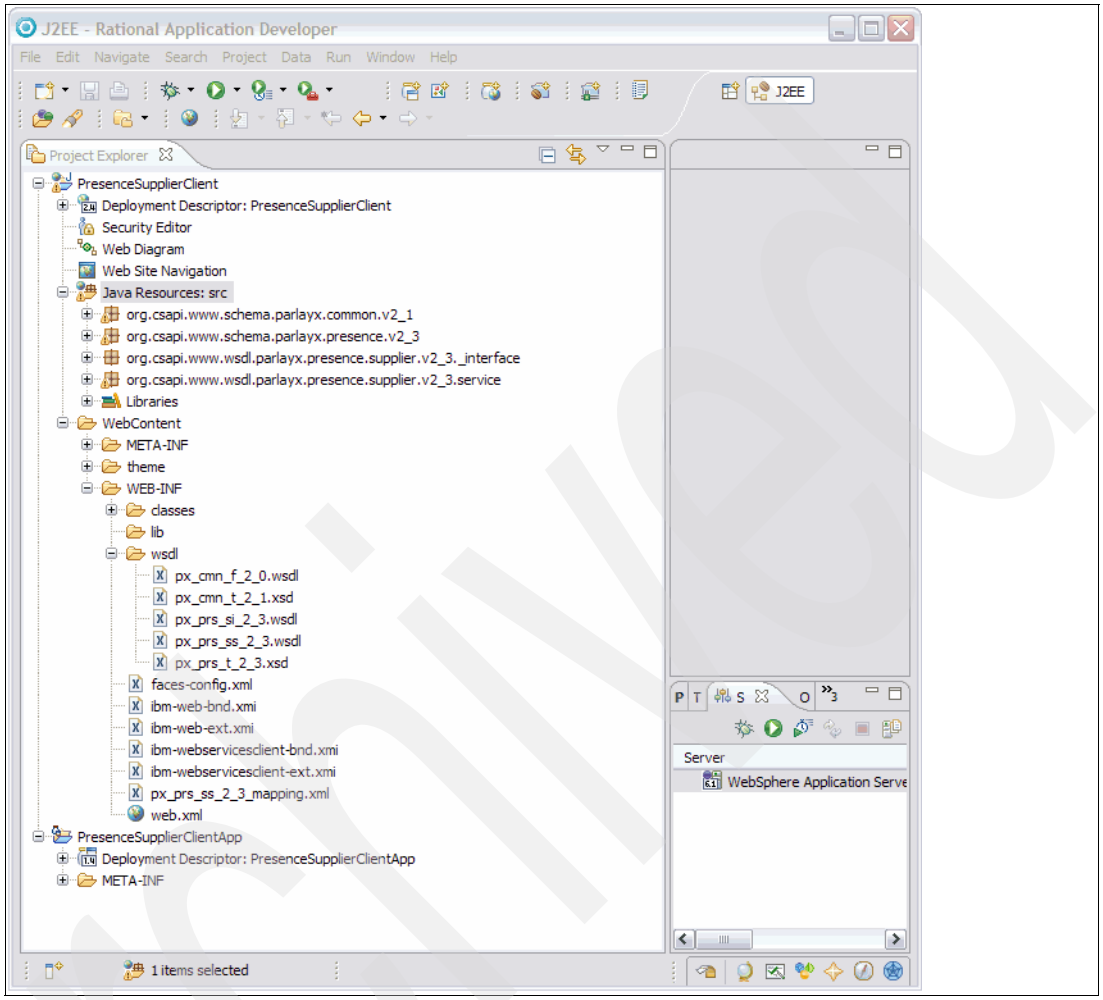


Figure 8-52 Web service client code package structure

Archive

9. Expand the package **org.csapi.www.wsdl.parlayx.presence.supplier.v2_3._interface**. Check if the class PresenceSupplierProxy.java exists. This class should have methods similar to that of the service operations in the Parlay X Presence Supplier interface. This class can be instantiated in the client logic to invoke the Parlay X Presence Supplier service operations. Select the class **PresenceSupplierProxy.java**, right-click it, and click **Open**. In the editor, scroll down and locate all the methods of the Presence Supplier Interface. The class should look like Figure 8-53.

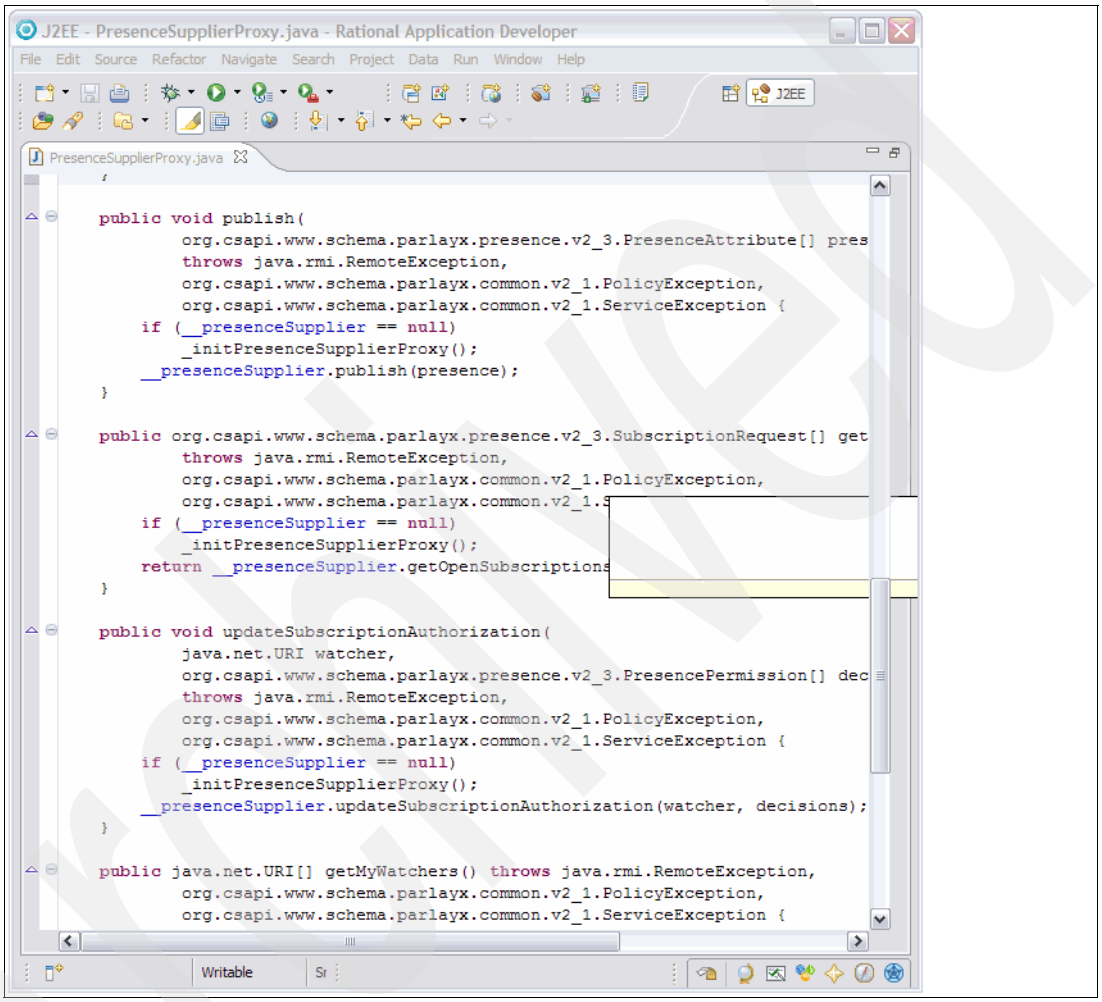


Figure 8-53 The auto generated PresenceSupplierProxy class

Code customization

In the previous section, we saw the client code generated by Rational Application Developer. Typically, the client logic invokes the publish method of PresenceSupplierProxy along with the required Java object parameters. In order to successfully execute the operation, the code should be customized to include at least the following functionality:

The Service Endpoint should be set before the method invocation. Typically, the Service URL or Endpoint specified WSDL files usually refer to localhost as the host name. In the sample client application, the Service Endpoint is stored as a context parameter in the Web deployment descriptor. The code to set the Service Endpoint on the PresenceSupplierProxy class is shown in Example 8-18 on page 349.

Example 8-18 Sample showing code snippet to set the Service Endpoint

```
        // Instantiate the PresenceSupplierProxy
        PresenceSupplierProxy proxy = new PresenceSupplierProxy();
        // Override service endpoint
        String svcEndpoint = (String)
getFacesContext().getExternalContext().getInitParameter("TWSS_SERVICE_ENDPOINT");
        proxy.setEndpoint(svcEndpoint);
```

Follow these steps to configure the context parameter TWSS_SERVICE_ENDPOINT in the Web Deployment descriptor:

1. Open the Web deployment descriptor. In the Overview tab, scroll down and locate the Context Parameters section, as shown in Figure 8-54.

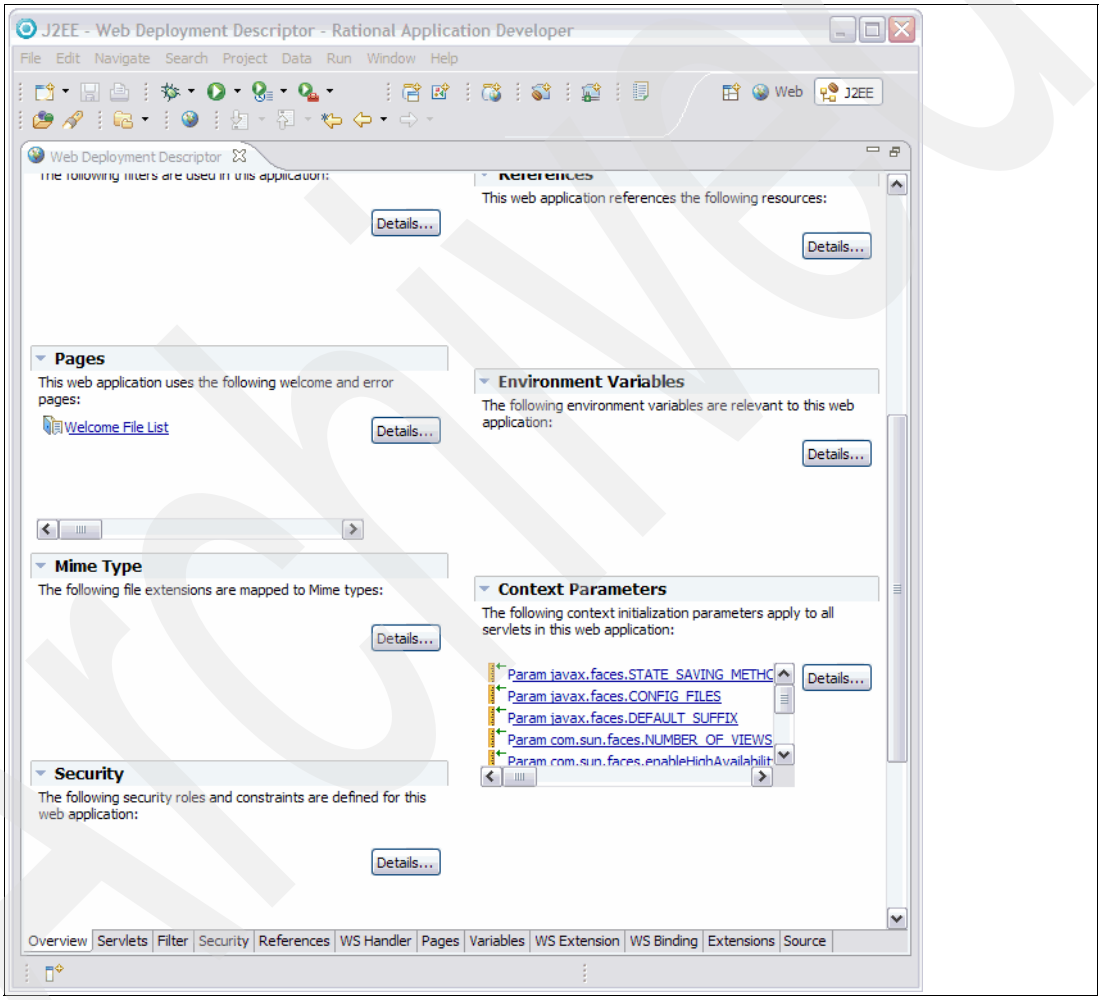


Figure 8-54 Context Parameters section in Overview tab of web.xml

Archive

2. Click the **Details** button on the right. The Variables tab is displayed. Click the **Add...** button. The **Add Parameter** dialog is displayed, as shown in Figure 8-55.

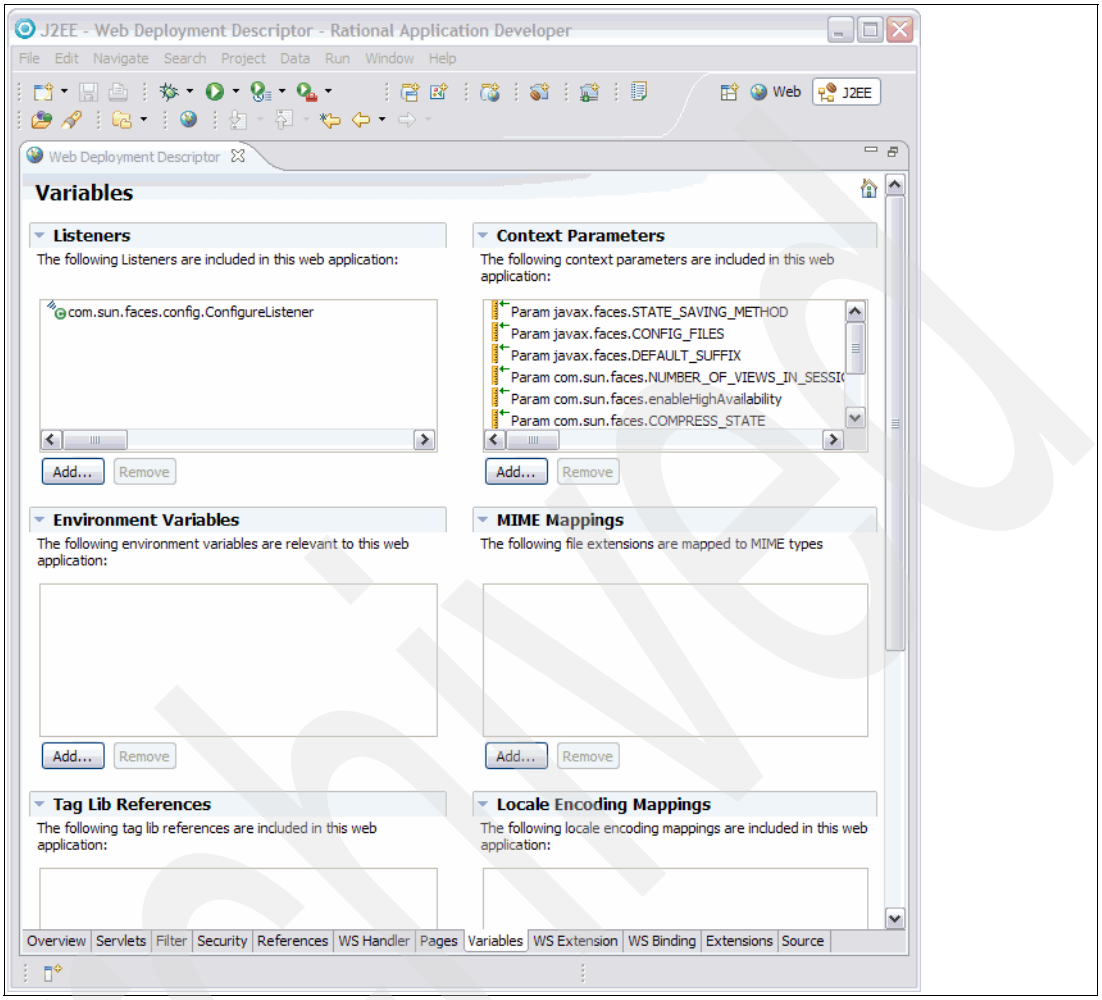


Figure 8-55 Context Parameter details in the Variables tab of web.xml

Archive

3. Enter TWSS_SERVICE_ENDPOINT in the Parameter name and http://127.0.0.1:80/PresenceService/services/PresenceSupplier in the Parameter value field. Click the **Finish** button (Figure 8-56).

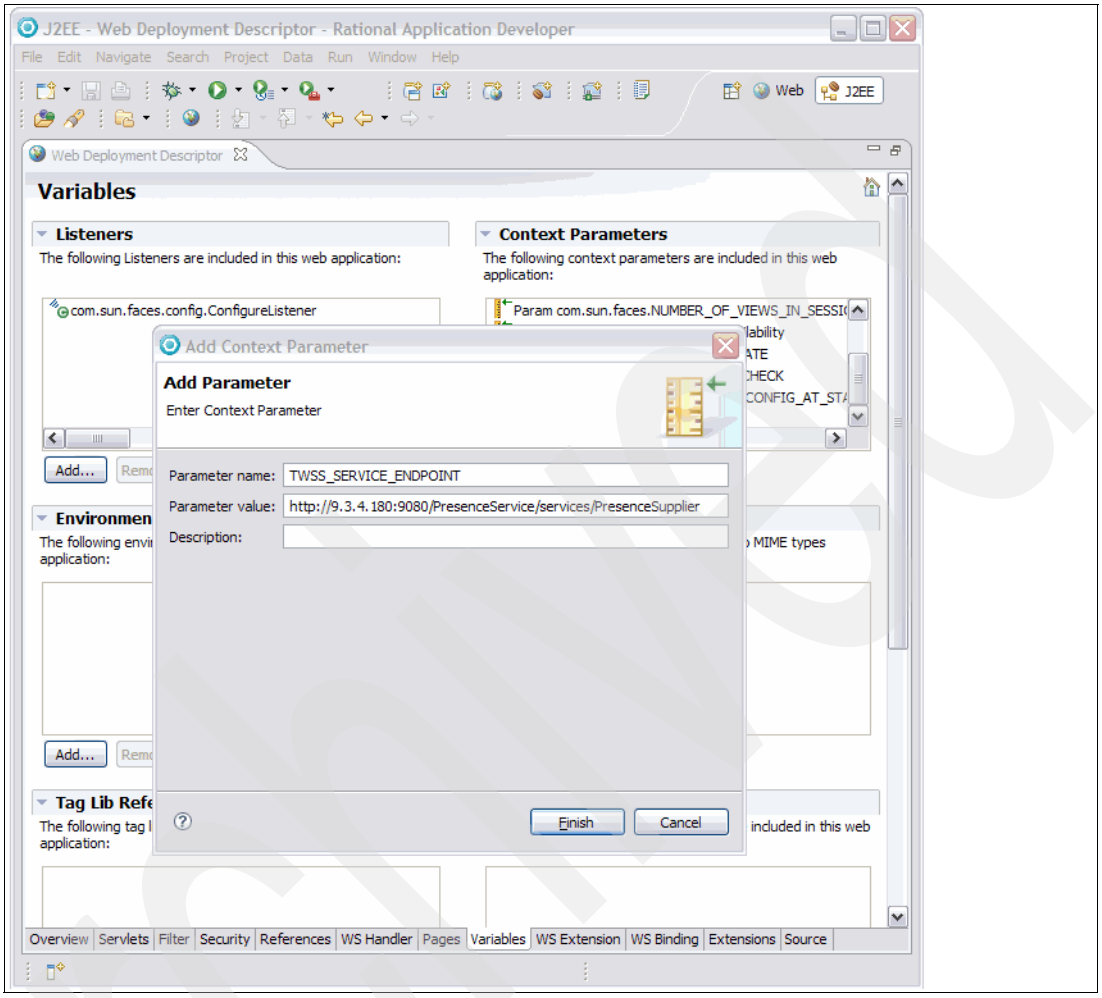


Figure 8-56 Add the TWSS_SERVICE_ENDPOINT context parameter

The context parameter can be found in the Web Deployment Descriptor or web.xml. Figure 8-57 shows the web.xml where TWSS_SERVICE_ENDPOINT is defined.

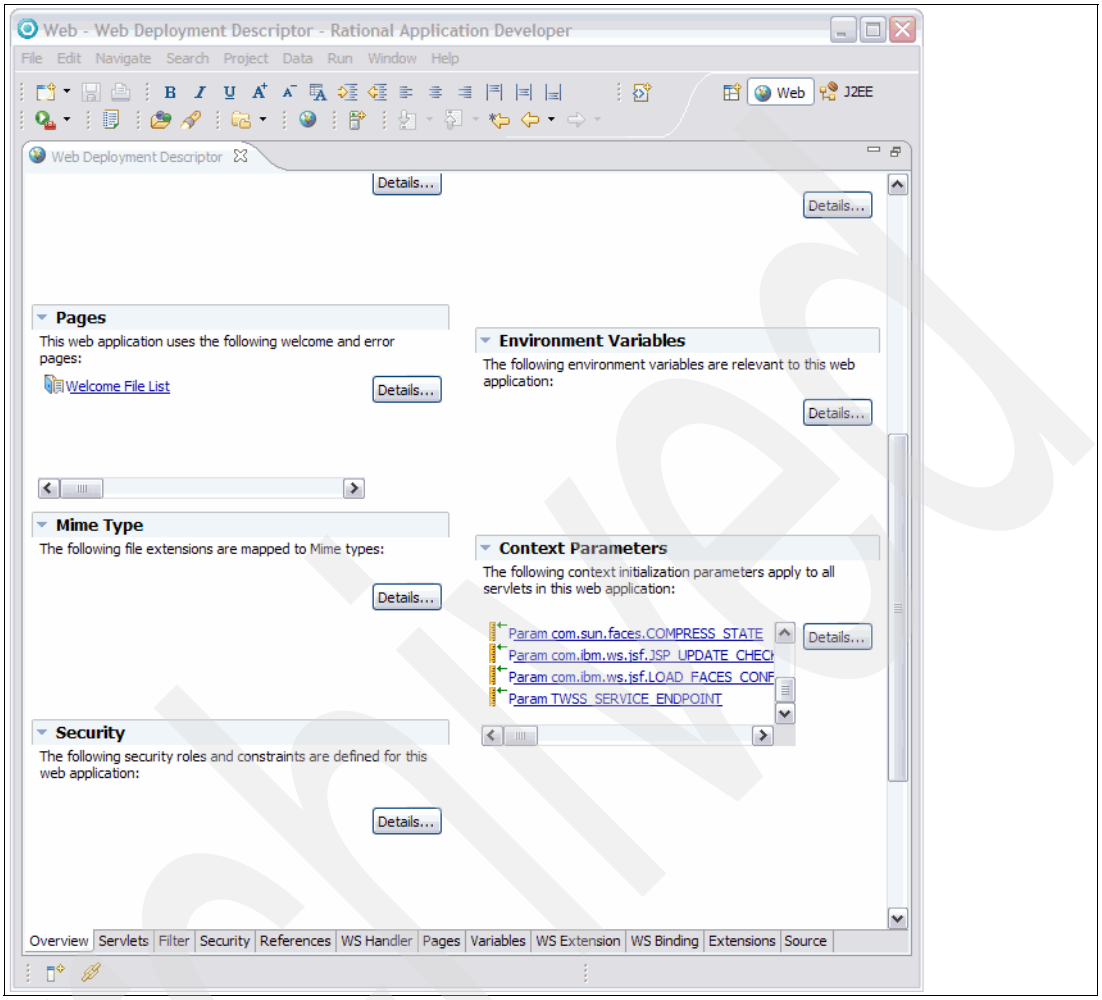


Figure 8-57 Context parameters in the Client application web.xml

- The value of the **TWSS_SERVICE_ENDPOINT** context parameter should be a valid Web Service Endpoint of the Parlay X Presence Supplier service implementation. The value in our case is `http://9.3.4.180:9080/PresenceService/services/PresenceSupplier`.

> **Note:** In 8.8.5, "Test execution" on page 362, we discuss the TCP/IP monitor configuration. As part of the monitor configuration, we discuss how to modify the TWSS _SERVICE_ENDPOINT context parameter value.

- In our sample scenario, user credentials must be passed along with a SOAP request for the publish service operation. Since the Access Gateway instance is secured, service invocation is required to have valid user credentials. To facilitate this requirement, we will customize the PresenceSupplierProxy class. Example 8-19 on page 353 shows the modifications to include the username and password member variables and the corresponding getter and setter methods.

Archive

Example 8-19 Sample code snippet showing the addition of username and password member variables

```
public class PresenceSupplierProxy
      implements

org.csapi.www.wsdl.parlayx.presence.supplier.v2_3._interface.PresenceSupplier {
   private boolean _useJNDI = true;

   private String _endpoint = null;
   // User Name
   private String _userName = null;
   // Password
   private String _password = null;
   public String get_password() {
      return _password;
   }

   public void set_password(String _password) {
      this._password = _password;
   }

   public String get_userName() {
      return _userName;
   }

   public void set_userName(String name) {
      _userName = name;
   }
```

In addition to the above mentioned modifications to PresenceSupplierProxy, we require another method, _myInitPresenceSupplierProxy(). This method is called by all the service methods before making the remote call to a Web service. Example 8-20 shows the method in detail.

Example 8-20 Sample code snippet to set the username and password in the SOAP request

```
   /**
    * _myInitPresenceSupplierProxy
    *
    * This method sets the 'username' and 'password' values to
    * javax.xml.rpc.Stub.USERNAME_PROPERTY and
    * javax.xml.rpc.Stub.PASSWORD_PROPERTY respectively.
    */
   private void _myInitPresenceSupplierProxy() {
      if (__presenceSupplier != null) {
         if (_endpoint != null)
            ((javax.xml.rpc.Stub) __presenceSupplier)._setProperty(
                  "javax.xml.rpc.service.endpoint.address", _endpoint);
         else
            _endpoint = (String) ((javax.xml.rpc.Stub) __presenceSupplier)
                  ._getProperty("javax.xml.rpc.service.endpoint.address");
         if (_userName != null) {
            System.out.println("User Name: " + _userName);
            ((javax.xml.rpc.Stub) __presenceSupplier)._setProperty(
```

Archived

```
                javax.xml.rpc.Stub.USERNAME_PROPERTY, _userName);
        }
        if (_password != null) {
           System.out.println("Password: " + _password);
           ((javax.xml.rpc.Stub) __presenceSupplier)._setProperty(
                javax.xml.rpc.Stub.PASSWORD_PROPERTY, _password);
        }
     }
  }
```

This concludes the generation and customization of Web Service Client code for the sample scenario Parlay X Presence Supplier service. Additionally, the Web Service Client code has the logic to generate the Parlay X Presence PresenceAttribute type and its sub-types, and the logic to render the Web pages.

Note: The complete sample code for this service client is available for download. Refer to Appendix E, “Additional material” on page 399 for detailed instructions on how to download and work with this sample code.

The sample code also comprises the PresenceSupplierClient project interchange zip, which can be imported into a RAD workspace.

In the next section, we discuss the test execution and results.

8.8.4 Test environment configuration

In this section, we demonstrate the test execution as well as the usage of TCP/IP monitor to monitor the SOAP requests and responses. The TCP monitor tool is available in Rational Application Developer.

Archive

Do the following steps:

1. Switch to the J2EE perspective in RAD. In the tabbed panel at the bottom of the IDE, locate the Servers tab. Typically, WebSphere Application Server V6.1 is configured in RAD V7.0 by default. Select the configured server, right-click it, and select **Start**. The IDE should look like Figure 8-58 upon successful start of the server.

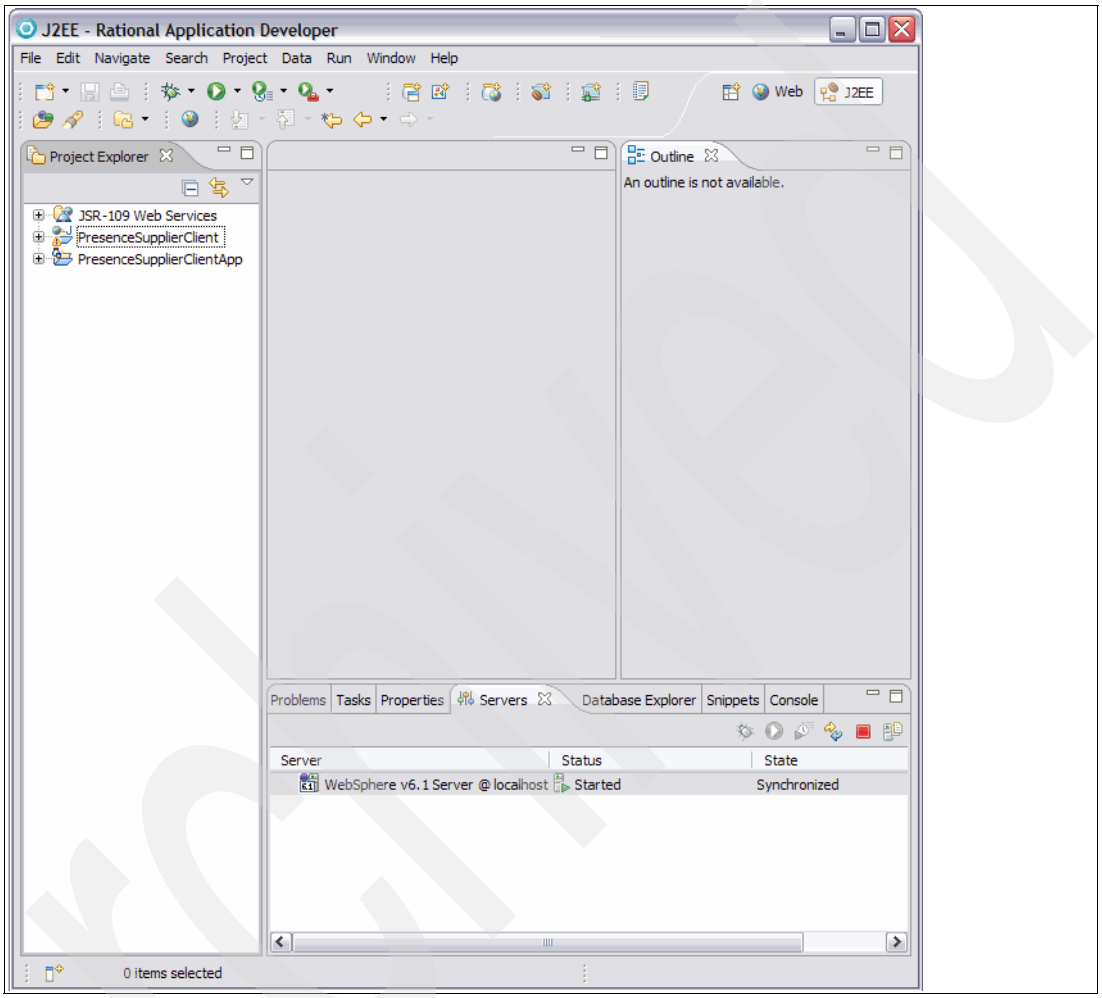


Figure 8-58 Test environment server being started

2. Once the server is started, we can deploy the client application to this server instance. Right-click the configured server and select **Add and Remove Projects...** in the pop-up menu. The Add and Remove Projects wizard is displayed. From the Available Projects list on the left, select **PresenceSupplierClientApp** and click **Add** in the center of the window. Click the **Finish** button (Figure 8-59).

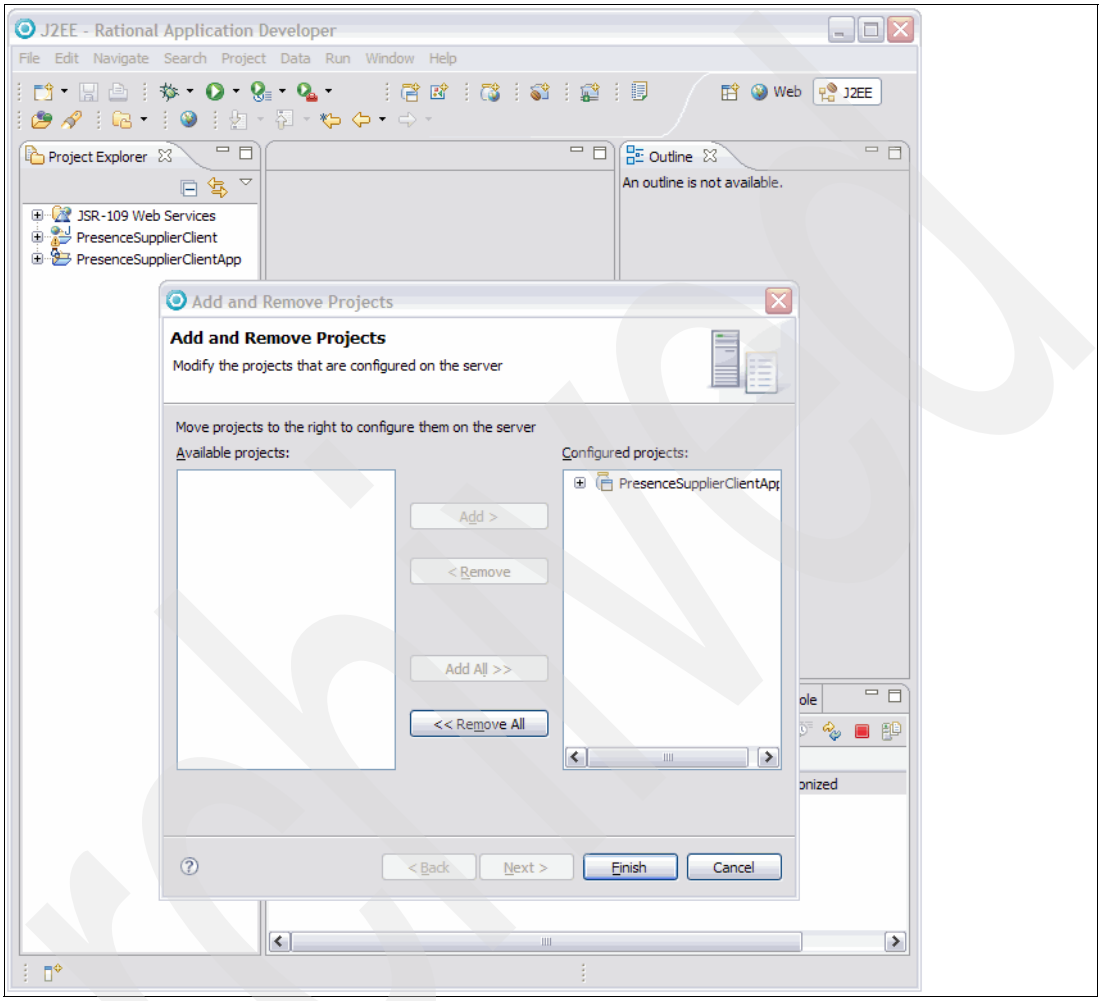


Figure 8-59 Deployment of sample client application

3. Upon successful deployment, the PresenceSupplierClientApp application should be displayed under the configured server in the Servers tab. The IDE should look as shown in Figure 8-60 on page 357.

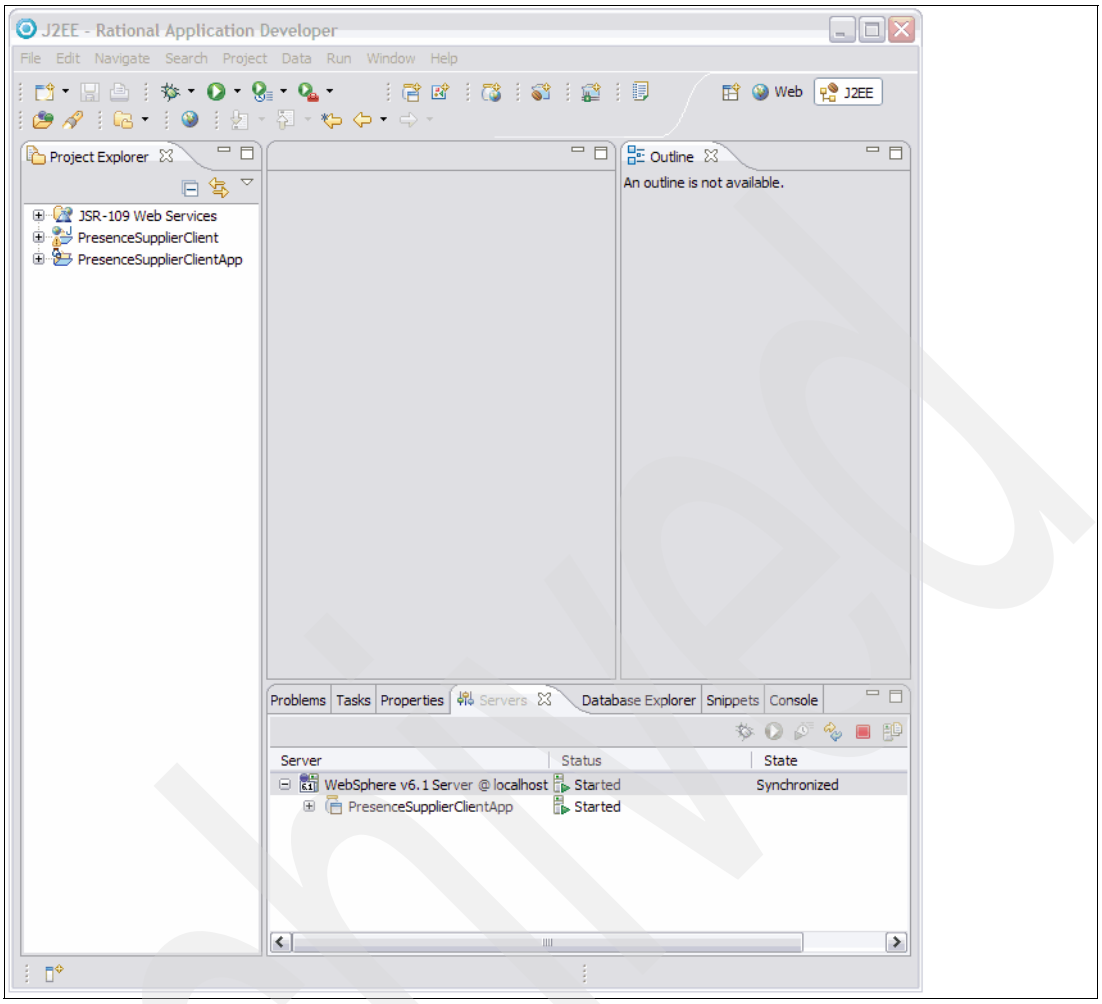


Figure 8-60 Sample client application deployed

This concludes the deployment of the sample client enterprise application.

Archive

Enabling SOAP message monitoring

The SOAP messages sent by the client application to Access Gateway and the responses received can be monitored using TCP/IP Monitor tool in Rational Application Developer. To enable this tool, follow these steps:

1. Select the **Window** menu, select **Show View**, and select **Other...**. The Show View wizard is displayed. Locate the **Debug** element in the tree and expand it. Select **TCP/IP Monitor** and click the **OK** button. The TCP/IP Monitor tab is added to the tabbed pane at the bottom of the IDE (Figure 8-61).

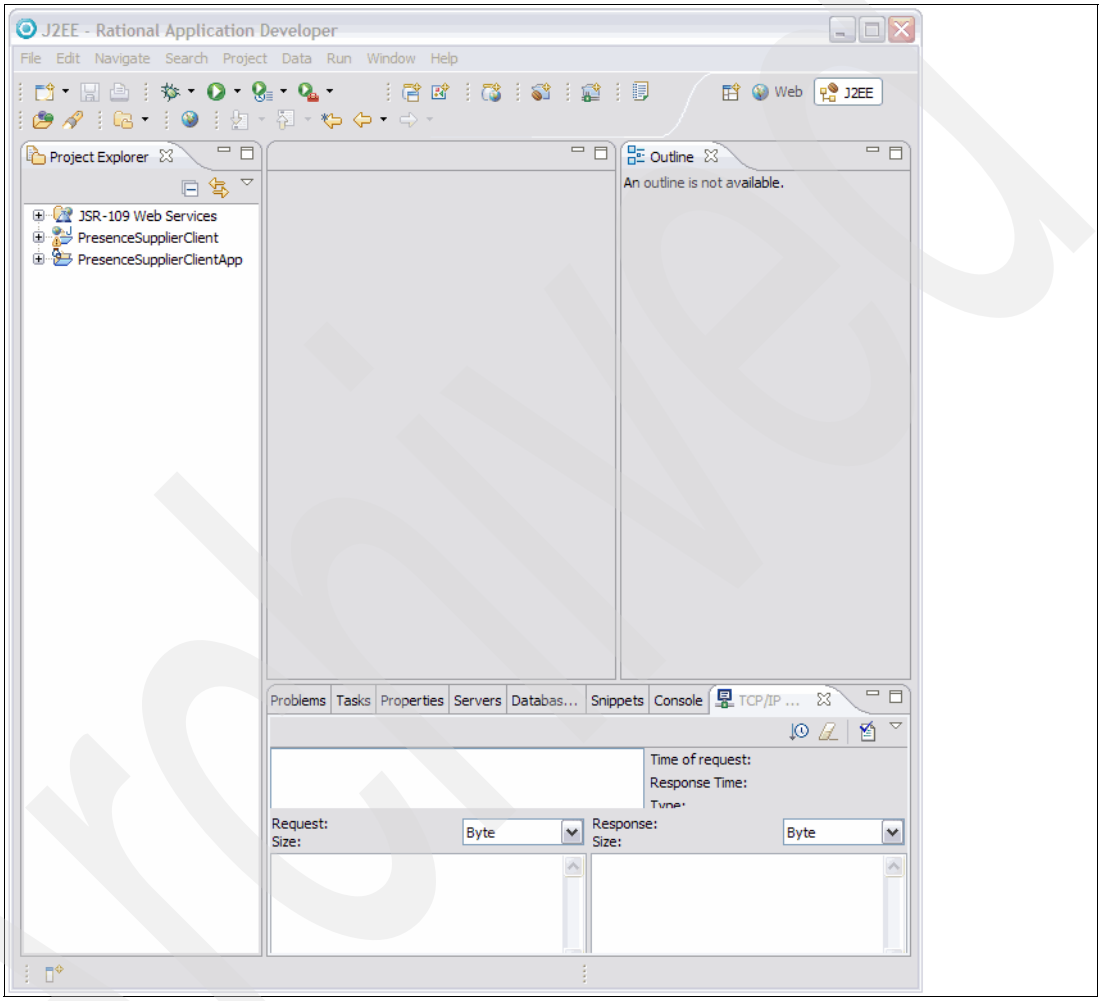


Figure 8-61 The TCP/IP Monitor view in RAD

2. To configure the TCP/IP monitor, right-click the text area in the TCP/IP Monitor tab. Select **Properties**. This action opens the Preferences wizard for the TCP/IP monitor. Check the **Show the TCP/IP Monitor view when there is activity** check box. Click the **Add** button on the right side of the wizard. This action opens up the New Monitor dialog box (Figure 8-62).

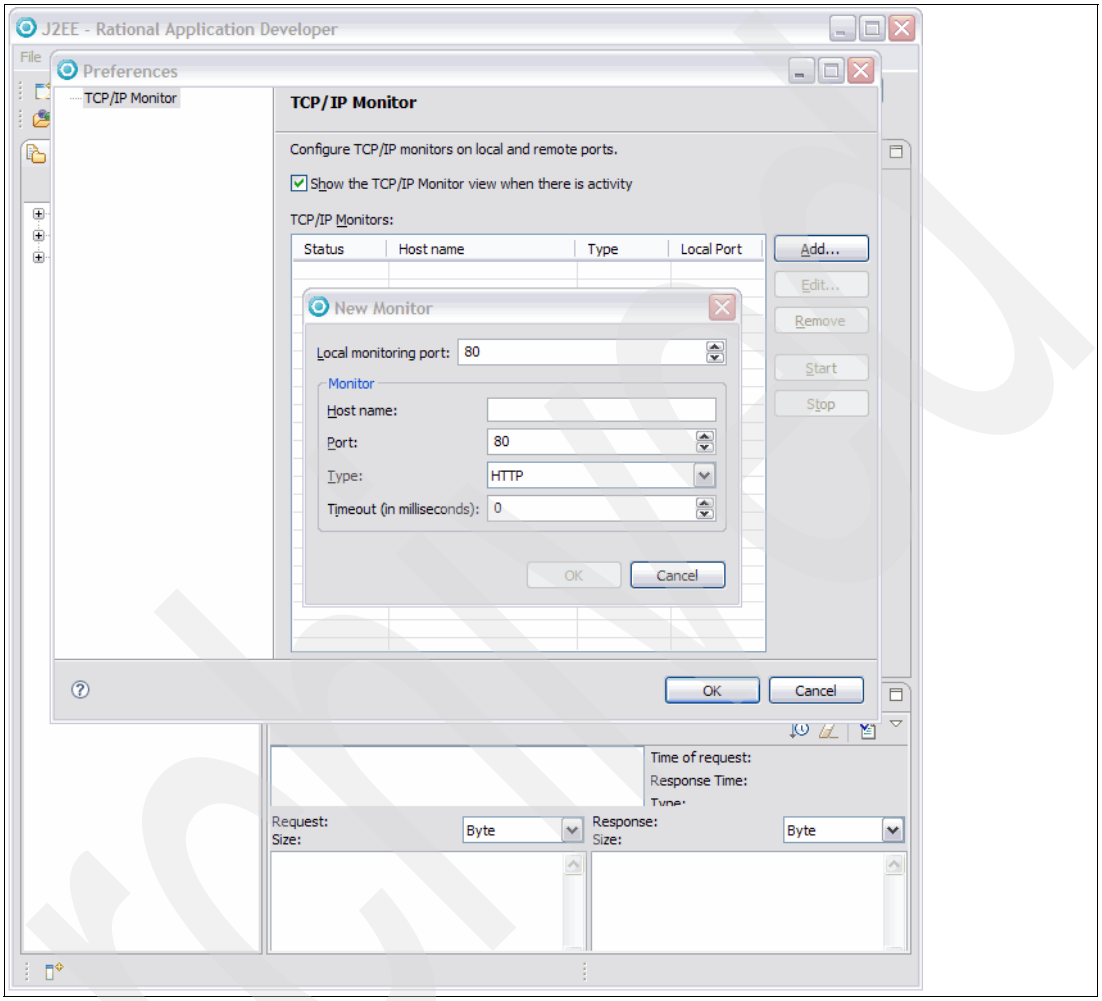


Figure 8-62 Configure new TCP/IP monitor

3. If port 80 is not in use on the system, leave the Local monitoring port as 80; otherwise, enter an unused port. In the Monitor section, enter the host name or IP address of Access Gateway in the Host name field. In the Port field, enter the HTTP port where the Web service is accessible.

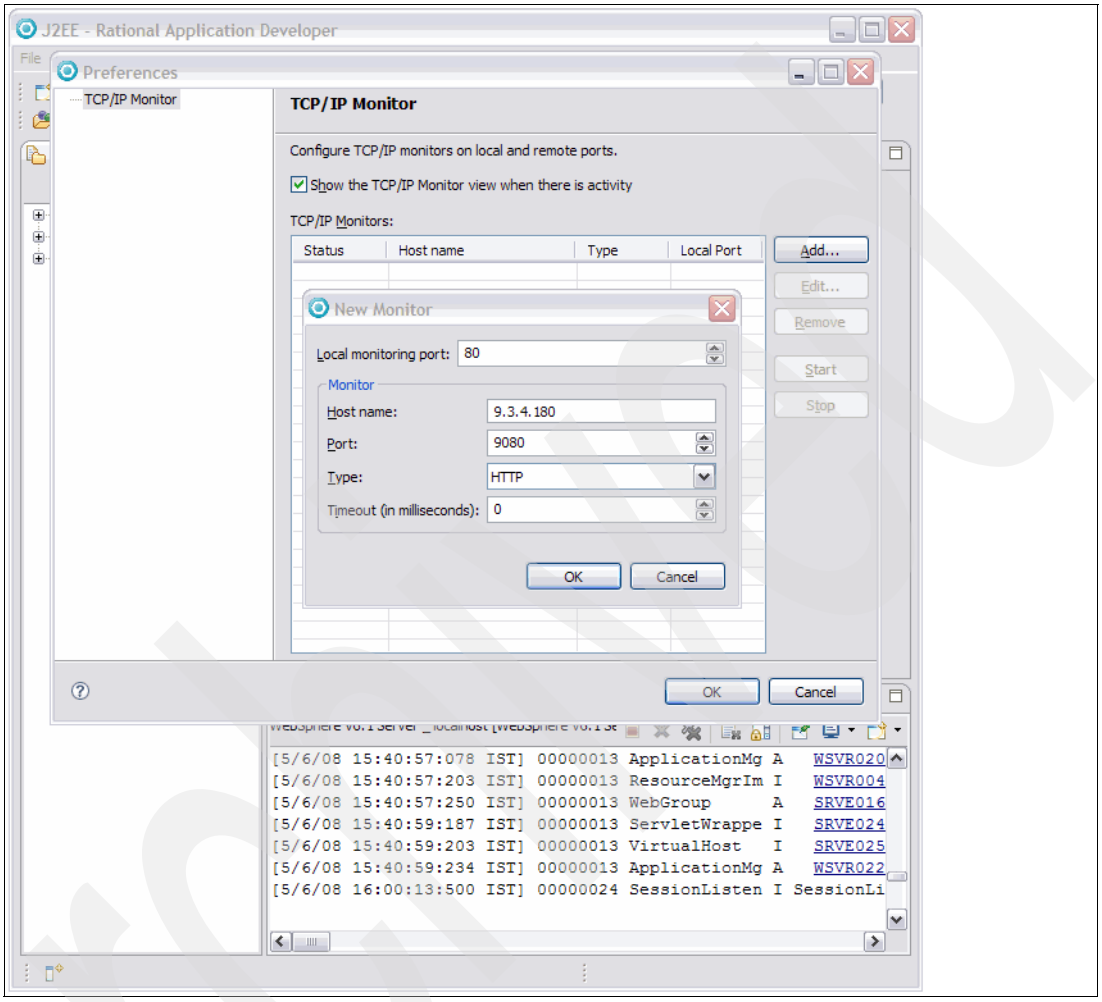


Figure 8-63 Configuration of Access Gateway host and port in the TCP/IP Monitor

Note: The above configuration primarily enables the TCP/IP monitor to:

- Inspect the TCP/IP messages that are either sent or received on localhost and port 80.
- The inspected messages are sent to access gateway 9.3.4.180 and port 9080.

Additionally, the above configuration requires us to change the TWSS_SERVICE_ENDPOINT context parameter value to http://127.0.0.1:80/PresenceService/services/PresenceSupplier.

4. Open the Web deployment descriptor. In the Overview tab, scroll down to the Context Parameters section. Click the **Details** button. The Variables tab is displayed. In the list of context parameters, select **TWSS_SERVICE_ENDPOINT** and click the **Remove** button.

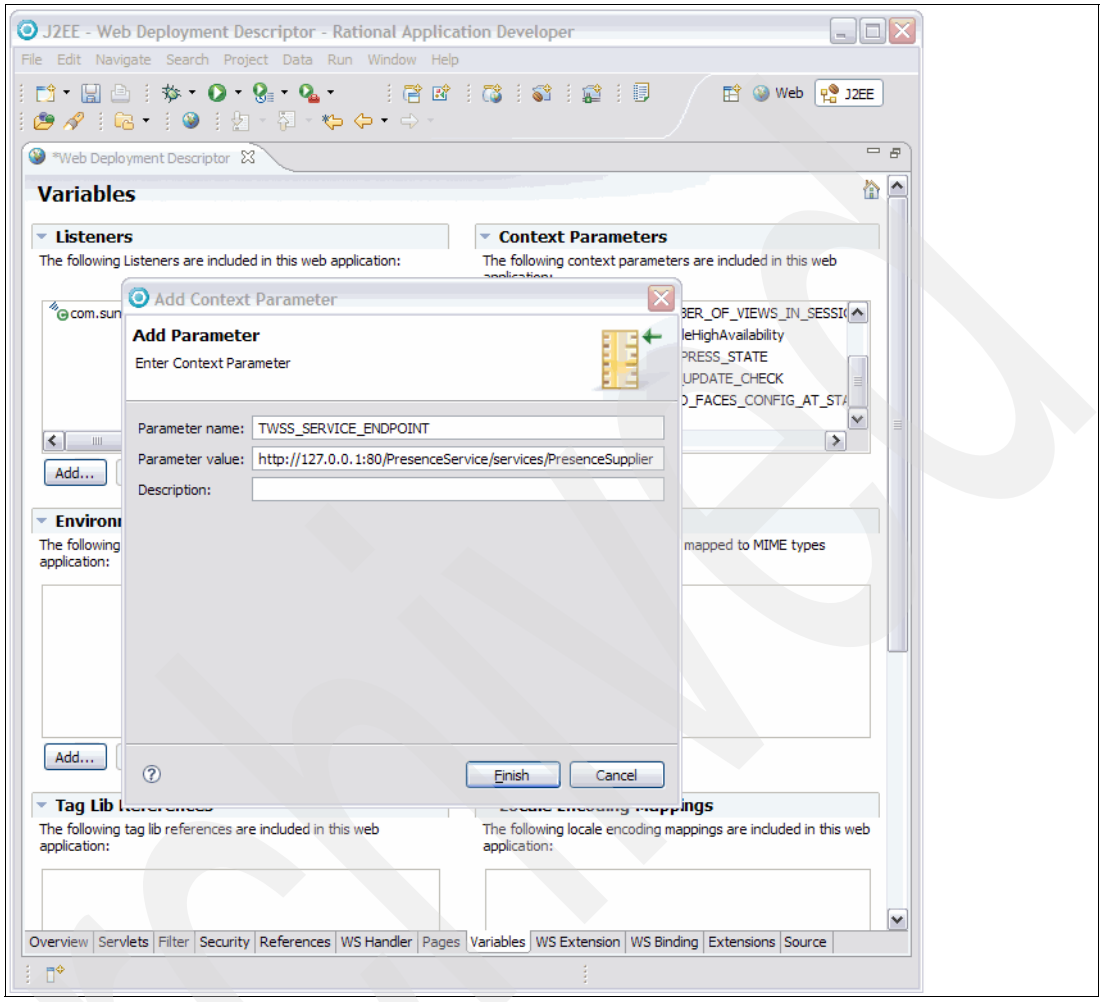


Figure 8-64 Change the host and port of the TWSS_SERVICE_ENDPOINT context parameter

5. Start the monitor and have it listen on port 80 of localhost for any TCP/IP traffic. Right-click the text area in the top portion of the TCP/IP Monitor tab and select **Properties**. In the TCP/IP Monitors table, select the monitor entry specific to the Access Gateway host and port created earlier. Click the **Start** button on the right side. The monitor status should resemble Figure 8-65.

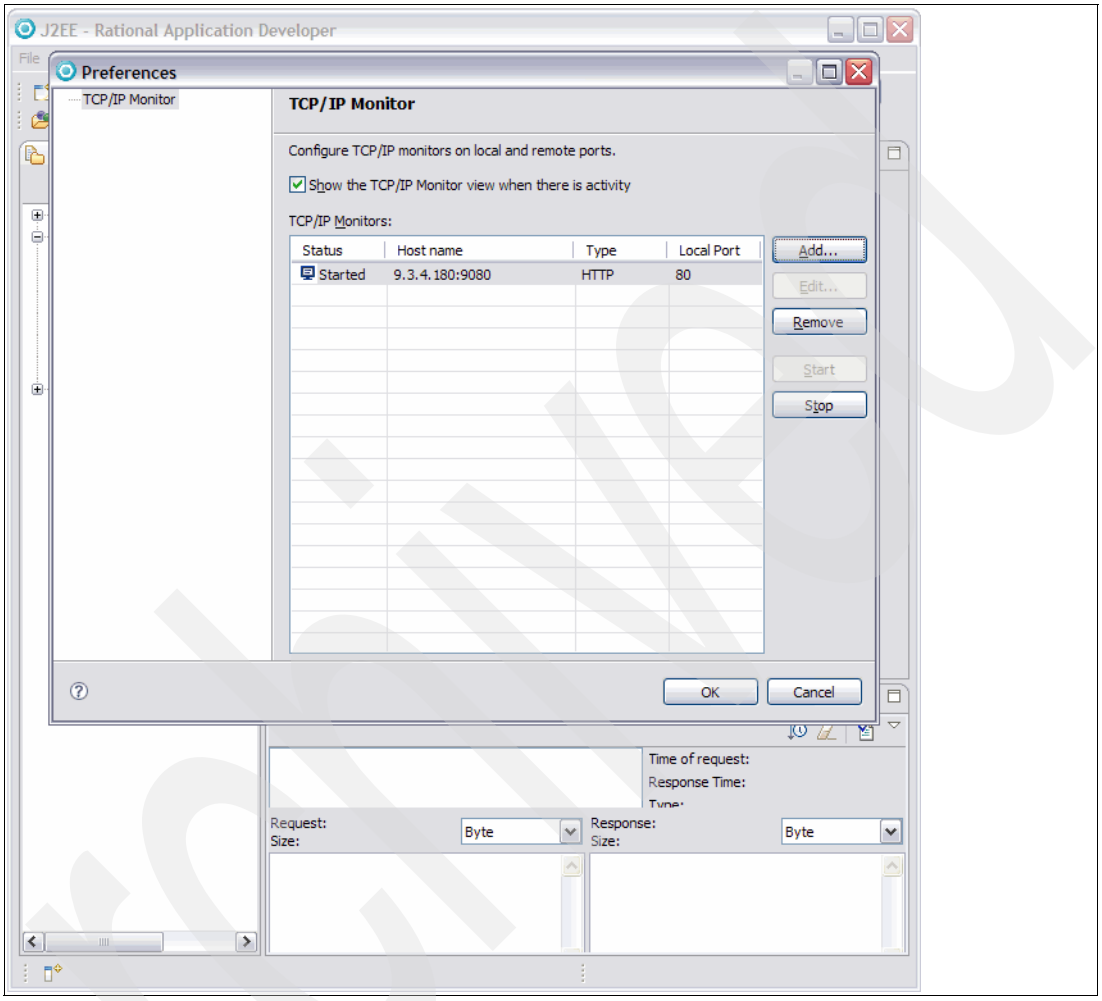


Figure 8-65 TCP/IP Monitor proxying AG Host and Port started

This concludes the configuration settings for TCP/IP monitor as well as the client application deployment.

8.8.5 Test execution

In this section, we will walk through the test execution for the sample scenario that comprises the invocation of the publish service operation. The client application is a Web Service Client and Web UI. The client application makes a Web service call to the Parlay X Presence Supplier Service interface hosted on Access Gateway. The Access Gateway executes the mediation flow associated with the service interface. The mediation flow is responsible for integrating with the service policy manager and enforce policies on the in-flight request. As a result, the request is enriched and finally handed over to the service platform. The service implementation logic is invoked, which eventually initiates SIP signaling with the back-end SIP Presence Server.

We will demonstrate this end-to-end execution of the scenario using screen captures and log entries of the trace.log files of Access Gateway and Service Platform instances. The Presence Monitor Web UI will be used to demonstrate that the presentity's presence information is in fact published to the Presence Server.

Do the following steps:

1. Prepare the SOAP request message. Figure 8-66 shows a sample publish SOAP Request.

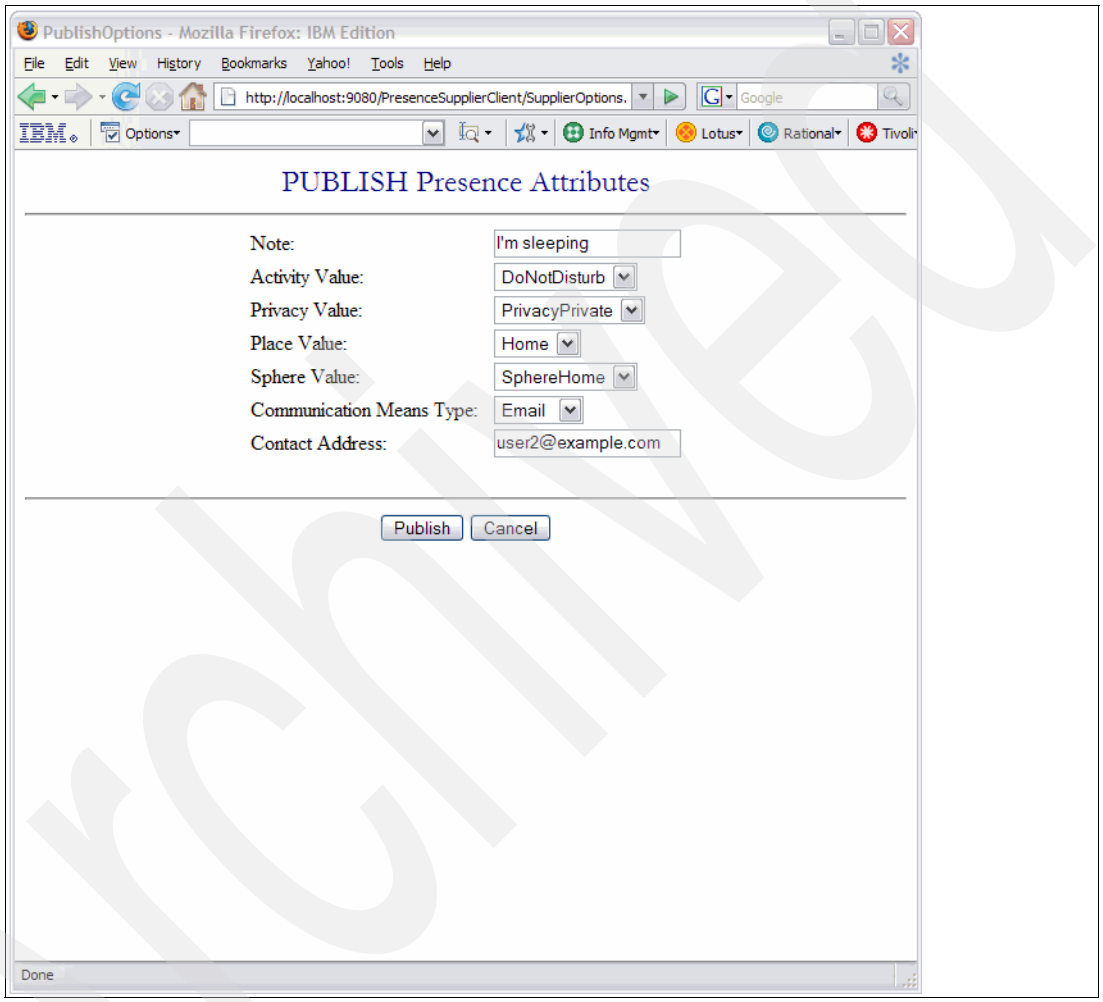


Figure 8-66 Prepare Presence Attributes for PUBLISH request

Archive

2. The request XML created by the information in Figure 8-66 on page 363 is shown in Example 8-21.

Example 8-21 The generated request XML that will form the payload in the publish SOAP request

```
<?xml version="1.0" encoding="UTF-8"?>
<loc:publish
xmlns:loc="http://www.csapi.org/schema/parlayx/presence/supplier/v2_3/local">
    <presence>
        <lastChange>2008-05-06T17:02:57:593+0530</lastChange>
        <note>I'm sleeping</note>
        <typeAndValue>
            <UnionElement/>
            <Activity>
                <DoNotDisturb/>
            </Activity>
            <Place>
                <Home/>
            </Place>
            <Privacy>
                <PrivacyPrivate/>
            </Privacy>
            <Sphere>
                <SphereHome/>
            </Sphere>
            <Communication>
                <priority>0.5</priority>
                <contact>user2@example.com</contact>
                <type>
                    <Email/>
                </type>
            </Communication>
            <Other>
                <name/>
                <value/>
            </Other>
        </typeAndValue>
    </presence>
</loc:publish>
```

After generating the request XML, the Web service call is made by the client. At this stage, the TCP/IP monitor configured earlier in “Enabling SOAP message monitoring” on page 358 can be used to check the SOAP message. Figure 8-67 on page 365 shows the SOAP message.

Archived

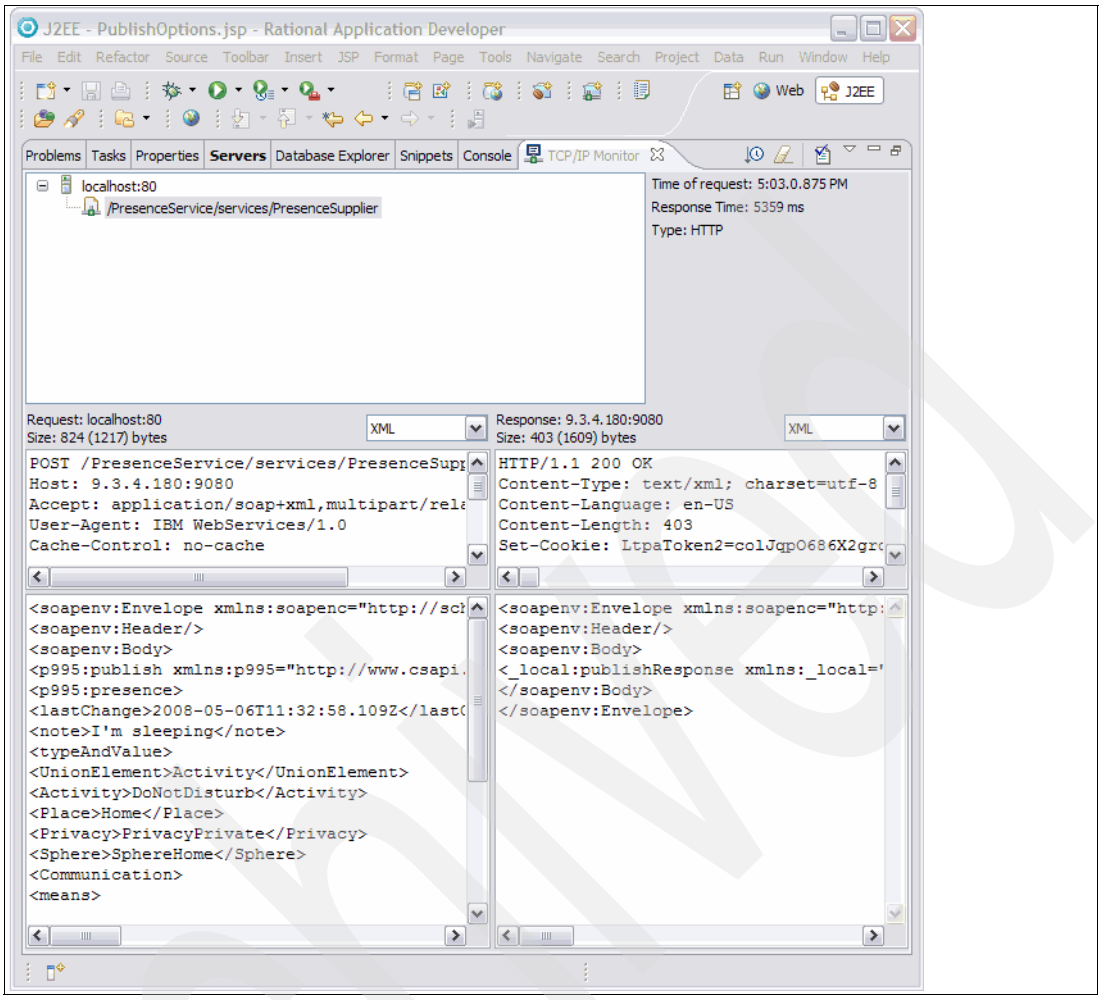


Figure 8-67 The publish SOAP Message as inspected by the TCP/IP monitor

3. The above step confirms that the SOAP message has been received by localhost and port 80. The monitor behaves like a proxy to Access Gateway and passes on the SOAP message to the Access Gateway host and port. In this step, we will check the trace entries in the Access Gateway trace.log file, specifically the processing of ServiceInvocationMediation mediation primitive (Example 8-22). This step confirms that the mediation flow for Parlay X Presence Supplier in the Access Gateway instance successfully executed all the mediation primitives and is ready to hand over the SOAP request to the service implementation hosted in Service Platform.

Example 8-22 Extract from Access Gateway trace.log showing the SMO entering and exiting ServiceInvocationMediation mediation primitive

```
[4/21/08 5:45:16:505 EDT] 00000073 ServiceInvoca 3
com.ibm.twss.ag.mediation.logic.ServiceInvocationMediation mediate Entered method
[4/21/08 5:45:16:507 EDT] 00000073 ServiceInvoca 3
com.ibm.twss.ag.mediation.logic.ServiceInvocationMediation mediate Input message:
<?xml version="1.0" encoding="UTF-8"?>
<ServiceMessageObject:smo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ServiceMessageObject="http://www.ibm.com/websphere/sibx/smo/v6.0.1"
xmlns:_local="http://www.csapi.org/schema/parlayx/presence/supplier/v2_3/local"
xmlns:_pX2="http://PX21_PRS_FLOW" xmlns:_v10="http://www.ibm.com/schema/twss/v1_0"
```

```
xmlns:interface="http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interfac
e">
  <context>
    <transient xsi:type="_pX2:ExceptionType"/>
  </context>
  <headers>
    <SMOHeader>
      <MessageUUID>705F7C8E-0119-4000-E000-6202C0A80166</MessageUUID>
      <Version>
        <Version>6</Version>
        <Release>0</Release>
        <Modification>2</Modification>
      </Version>
      <MessageType>Request</MessageType>
    </SMOHeader>
    <SOAPHeader>
      <nameSpace>http://www.ibm.com/schema/twss/v1_0</nameSpace>
      <name>twssHeaders</name>
      <prefix>twss</prefix>
      <value xsi:type="_v10:twssHeaders">

<globalTransactionID>705F7C8E-0119-4000-E000-6202C0A80166</globalTransactionID>
        <requesterID>sip:user2@example.com</requesterID>
        <policies>
          <policy attribute="service.config.ServiceImplementationName"
value="PX21_PRS_SPLR_IMS"/>
          <policy attribute="message.sla.LocalRequesterRate" value="0"/>
          <policy attribute="message.sla.ClusterOperationRate" value="0"/>
          <policy
attribute="service.config.presence.supplier.PresenceServerResourceName"
value="PX21_PRS_SPLR_IMS"/>
          <policy attribute="message.sla.LocalServiceRate" value="0"/>
          <policy attribute="message.sla.LocalWeight" value="0"/>
          <policy attribute="message.transaction.RecordTransaction"
value="false"/>
          <policy attribute="message.groups.MaxGroupSize" value="100"/>
          <policy attribute="service.config.presence.supplier.PresenceServerURI"
value=""/>
          <policy attribute="message.sla.LocalOperationRate" value="0"/>
          <policy attribute="service.standard.MaximumNotificationFrequency"
value="1"/>
          <policy attribute="requester.anonymousAccessAllowed" value="true"/>
          <policy attribute="service.standard.MaximumNotificationDuration"
value="92233720368547758O7"/>
          <policy attribute="service.config.AdditionalGroupURISchemes"
value="glmgroup"/>
          <policy attribute="service.standard.UnlimitedCountAllowed"
value="true"/>
          <policy
attribute="service.config.service.config.presence.PresenceFetchTimeoutInMillis"
value="1000"/>
          <policy attribute="service.config.enableURItransformation"
value="true"/>
          <policy attribute="message.sla.ClusterEnabled" value="false"/>
          <policy attribute="message.sla.ClusterRequesterRate" value="0"/>
```

```
          <policy
attribute="service.config.service.config.presence.SubscribePresenceTimeout"
value="86400000"/>
          <policy
attribute="service.config.service.config.presence.SubscribePresenceWinfoTimeout"
value="86400000"/>
          <policy attribute="service.Endpoint"
value="http://9.3.4.188:9082/PX21_PRS_SPLR_IMS/services/PresenceSupplier"/>
          <policy attribute="service.config.DefaultNotificationDuration"
value="86400000"/>
          <policy attribute="message.transaction.RecordStatistics" value="false"/>
          <policy attribute="message.LoggingEnabled" value="true"/>
          <policy attribute="message.sla.ClusterServiceRate" value="0"/>
          <policy attribute="service.standard.MaximumCount" value="2147483647"/>
          <policy attribute="message.sla.ClusterWeight" value="0"/>
<policy attribute="message.sla.LocalEnabled" value="false"/>
          <policy attribute="service.config.requestIDorig" value="user2"/>
        </policies>

<serviceID>http://www.csapi.org/wsdl/parlayx/presence/supplier/v2_3/interface</ser
viceID>
      </value>
    </SOAPHeader>
  </headers>
  <body xsi:type="interface:PresenceSupplier_publishRequest">
    <publish>
      <_local:presence>
        <lastChange>2008-05-06T11:32:58.109Z</lastChange>
        <note>I'm sleeping</note>
        <typeAndValue>
          <UnionElement>Activity</UnionElement>
          <Activity>DoNotDisturb</Activity>
          <Place>Home</Place>
          <Privacy>PrivacyPrivate</Privacy>
          <Sphere>SphereHome</Sphere>
          <Communication>
            <means>
              <priority>0.5</priority>
              <contact>mailto:user2@example.com</contact>
              <type>Email</type>
            </means>
          </Communication>
        </typeAndValue>
      </_local:presence>
    </publish>
  </body>
</ServiceMessageObject:smo>
[4/21/08 5:45:16:510 EDT] 00000073 ServiceInvoca 3
com.ibm.twss.ag.mediation.logic.ServiceInvocationMediation mediate Exiting method
```

Archived

4. In this step, we will check the trace.log of the Service Platform instance. The logic to handle the SOAP request and initiate SIP signaling with Presence Server confirms that the request is processed by the Parlay X Presence Supplier service implementation.

Example 8-23 Extract from Service Platform trace.log showing the generation of RPID content from SOAP request payload as well as submission of a request to Presence Server

```
[4/22/08 19:58:03:256 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.sip.PresencePublish <init> Presentity
URI: sip:user2@example.com
[4/22/08 19:58:03:259 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.sip.PresencePublish <init> Presence
Server URI: sip:9.3.4.203:5060
[4/22/08 19:58:03:262 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.sip.PresencePublish <init> Presence
Timeout: 60
[4/22/08 19:58:03:265 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.sip.PresencePublish <init> Event Package
Name: presence
[4/22/08 19:58:03:278 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.sip.PublishServlet createInitialPublish
Presence Server URI: sip:9.3.4.203:5060
[4/22/08 19:58:03:284 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.sip.PublishServlet createInitialPublish
PIDF Content:
<?xml version="1.0" encoding="UTF-16"?><presence
xmlns="urn:ietf:params:xml:ns:pidf" entity="sip:user2@example.com"
xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
xmlns:lt="urn:ietf:params:xml:ns:location-type"
xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"><tuple
id="610764352"><status><basic>open</basic></status></tuple><dm:person
id="372730457"><rpid:activities><DoNotDisturb/></rpid:activities><rpid:privacy><Pr
ivacyPrivate/></rpid:privacy><rpid:place><Home/></rpid:place><rpid:sphere><SphereH
ome/></rpid:sphere></dm:person><dm:device
id="1495965739">urn:device:000</dm:device><rpid:class>Email</rpid:class><rpid:cont
act priority="0.5">mailto:user2@example.com</rpid:contact></presence>
[4/22/08 19:58:03:297 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.impl.PresenceSupplierBindingImpl publish
SIP PUBLISH with PRESENCE Information sent.
[4/22/08 19:58:03:384 CDT] 0000003d                I
com.ibm.twss.parlayx21.presence.supplier.impl.PresenceSupplierBindingImpl
invokeUsageRecords Usage Record Response: 78C986ED-0119-4000-E000-355DC0A80165
[4/22/08 19:58:03:415 CDT] 00000045 SystemOut     0   SIP-Tag:
1210076798_WPS.server.1A.1207679587255.0_43_41_1
[4/22/08 19:58:03:416 CDT] 00000045 SystemOut     0   Expiry duration: 60
```

5. In the final step, we will check the Presence Server Monitor Web UI. The Presence Monitor application helps visualize the Presence Information stored in the Presence Server and XDMS servers. Figure 8-68 on page 369 shows the Published Presence Information that is the end result of the Web service call in step 1 on page 363.

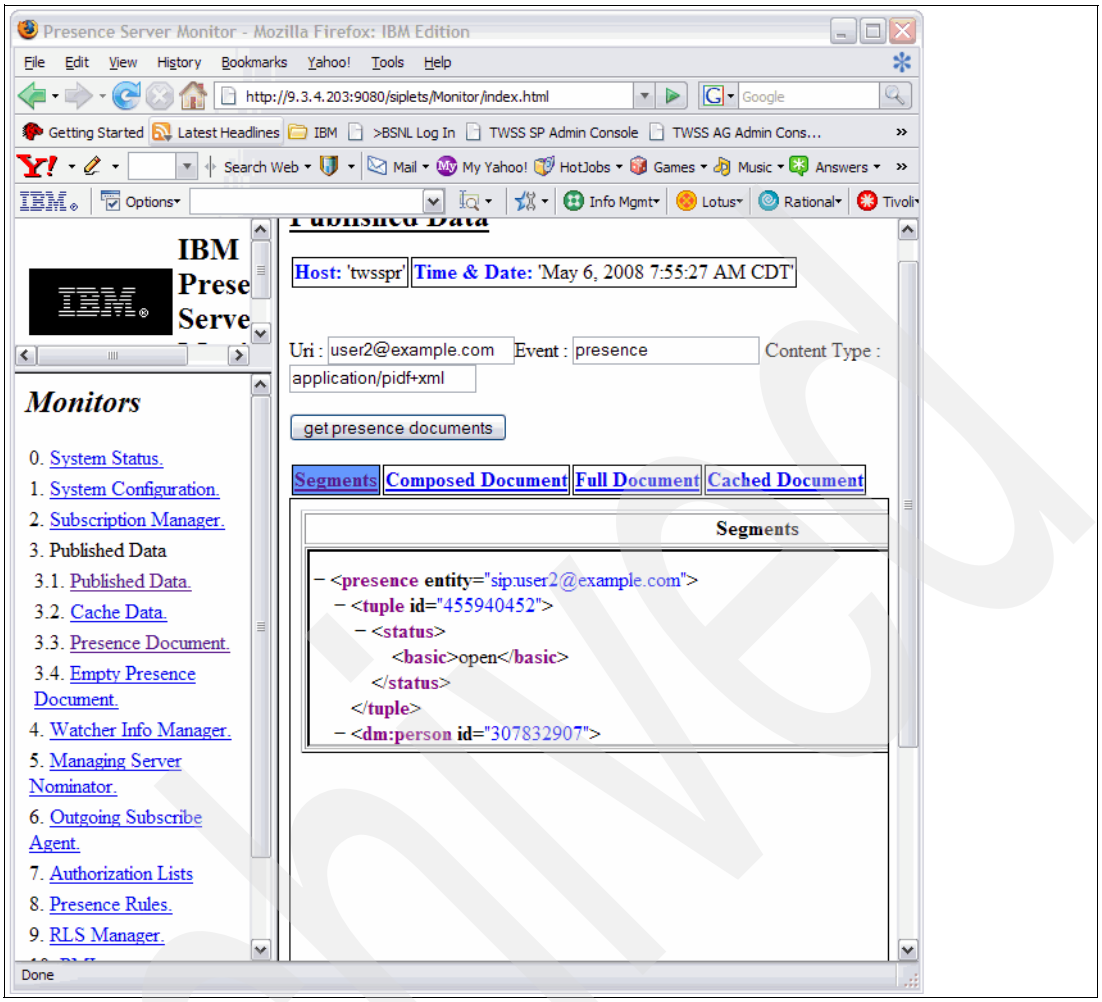


Figure 8-68 User2's Presence Information as displayed in the Presence Monitor Web UI

This concludes the end to end testing of the sample scenario.

Archive

Archived

A

Developing custom common components

You can develop your own common components for the IBM WebSphere Telecommunications Web Services Server. The common components are defined by a Web service WSDL interface. This appendix provides a brief example of how to do this task.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

Creating a IBM WebSphere Telecommunications Web Services Server service implementation project

You can substitute IBM WebSphere Telecommunications Web Services Server with custom implementations to meet a service provider's specific needs.

The process of implementing a IBM WebSphere Telecommunications Web Services Server implementation is similar to creating a service implementation, with a few differences:

- IBM WebSphere Telecommunications Web Services Servers service implementations do not have access to policy information or SOAP headers. IBM WebSphere Telecommunications Web Services Server service implementations only see data described within the IBM WebSphere Telecommunications Web Services Server service implementations Web Services Description Language (WSDL) interface.
- Multiple Parlay X Web service implementations can make use of IBM WebSphere Telecommunications Web Services Server service implementations.

You can deploy custom IBM WebSphere Telecommunications Web Services Server service implementations alongside IBM WebSphere Telecommunications Web Services Server service implementations. You can configure Parlay X Web service implementations within the IBM WebSphere Telecommunications Web Services Server service implementations to use the appropriate component implementation.

To create a IBM WebSphere Telecommunications Web Services Server service implementations project, follow the same procedure as for service implementation projects described in creating a custom binding implementation.

Integrating with IBM WebSphere Telecommunications Web Services Server administration console

The following change must be made in order to correctly classify the custom common component within IBM WebSphere Telecommunications Web Services Server administration console.

IBM WebSphere Telecommunications Web Services Server service implementations can follow the same procedures as for the integration of custom service implementations. However, for a common component implementation, these should be classified as Support Services when displayed in the administration console. This is defined in the admin.xma file, and provides better usability for the Administrator.

Additionally, the soa.SOAConsoleSettings.type_name=SOAConsoleServicePlatform property should be set, so the settings are displayed in the appropriate category in the administration console.

Other resources for learning more about custom common components

Refer to the IBM WebSphere Telecommunications Web Services Server Information Center, specifically to the WSDL documentation for WebSphere IBM WebSphere Telecommunications Web Services Server, for more information about WSDL or where it can

be located when installing the IBM WebSphere Telecommunications Web Services Server Base installer. This InfoCenter can be found at:

http://publib.boulder.ibm.com/infocenter/wtelecom/v6r1/index.jsp?topic=/com.ibm.twss.javadoc.doc/wsdl_c.html

The WSDL for the IBM WebSphere Telecommunications Web Services Server common components is provided. Custom implementations can be developed by providing an alternate implementation to the same interface, or by creating a new interface from which you call custom Web service implementations.

Archived

Archived

B

Developing service provider integrations

This appendix describes how to address the issue of integrating group resolution within IBM WebSphere Telecommunications Web Services Server.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

Integrating group resolution

The group resolution mediation primitive provided with IBM WebSphere Telecommunications Web Services Server Access Gateway acts as a Parlay X V2.1 Address List Management Web service client to resolve group URIs into member URIs.

Group resolution is performed only for specific Parlay X operations that reference groups URIs within an invocation. The group resolution mediation primitive makes use of the Group interface's queryMembers() operation to resolve groups. This operation is always called with the resolution of sub-groups set to true; Parlay X Web service implementations only operate on flattened lists of group members.

The IBM WebSphere Telecommunications Web Services Server Address List Management service provides an implementation for this interface. The IBM WebSphere XML Document Management Server Component provides an out-of-the-box implementation compatible with the IBM WebSphere Telecommunications Web Services Server Address List Management service. The endpoint to invoke the group resolution component can be configuring during the deployment of IBM WebSphere Telecommunications Web Services Server Access Gateway.

Custom integrations of a Service Provider's group management system can be done by providing an implementation of the queryMembers() method, and configuring the Access Gateway to invoke that implementation.

This interface is reproduced here for convenience:

- Operation: queryMembers.
- Get the list of members contained within a group.

If nested groups are supported, then the member list can contain group URIs as members. Therefore, two manners are supported for retrieving the list of members: with members resolved and without.

- If ResolveGroups is 'true', then the exclusive union of all the members contained within the group, and any nested subgroups, is the result (exclusive union means that after retrieving all members, duplicate members are removed).
- If ResolveGroup is 'false', then the group members are returned including group URIs as members of the group. If members within nested groups are required, subsequent calls to this operation with those groups can be used to retrieve those members.
- If nested groups are not supported, the value of ResolveGroups is ignored.

Table B-1 shows the input message for queryMembersRequest.

Table B-1 Input message: queryMembersRequest

Part name	Part type	Optional	Description
Group	xsd:anyURI	No	URI of group
ResolveGroups	xsd:Boolean	No	If true, return a set of members after resolving groups (including subgroups). If false, return members including group references.
Members	xsd:anyURI[0..unbounded]	Yes	Members of group.

Referenced faults

ServiceException from 3GPP TS 29.199-1 [6]:

- SVC0001: Service error
- SVC0002: Invalid input value
- POL0001: Policy error

Integrating Service Policy Management

There are several ways to integrate with the Service Policy Manager.

Replacing the implementation

The policy/subscription mediation primitive acts as a Web service client requester to a policy access interface defined by Service Policy Manager.

This interface defines a getServicePolicies() operation that allows for retrieval of policies for a given (requester, service, or operation) tuple. This operation returns a list of policy attribute/value pairs used as a runtime configuration for service delivery.

Service Policy Manager provides an out-of-the-box implementation of the getServicePolicies() operation, as well as additional interfaces for administering requester subscriptions and service policies. The endpoint to invoke the SPM component can be configured during deployment of the access gateway. This interface needs only to be substituted when integrating it with an earlier policy management system.

The IBM WebSphere Telecommunications Web Services Server Information Center contains the Web Services Description Language (WSDL) ServicePolicyAccessService interface that defines the getServicePolicies() operation.

Finally, in addition to the WSDL interfaces for the service policy manager, there are MBeans that are hosted there, and interfaces used during the install and configuration process to configure the initial system. If a completely alternate implementation is used, some architecture/design will be needed to migrate the default policies from the policy tables initialized by IBM WebSphere Telecommunications Web Services Server into the alternate repository.

Accessing from the SPM Administrative MBeans and Web Services

The Service Policy Manager also now provides MBean interfaces as well as Web service interfaces to the administrative interface of the Service Policy Manager. This allows opportunities for integration with the Service Provider.

In a solution deployment, the service provider may need to coordinate some policy information between IBM WebSphere Telecommunications Web Services Server and other components in the solution. The Administrative Interface, by using the MBeans or Web Services, can be used to push data into the Service Policy Manager.

The InfoCenter describes the WSDL Interface to the Web Services and the Java interface of data objects used when accessing the MBeans. The WebSphere WSADMIN AdminControl functions can be used from a script or Java software to communicate with these MBeans.

Archived

Integrating privacy management

Parlay X Web service implementations provided with IBM WebSphere Telecommunications Web Services Server support integration with a privacy management system by acting as a Web service client to a privacy management interface defined by IBM WebSphere Telecommunications Web Services Server.

IBM WebSphere Telecommunications Web Services Server does not include a privacy implementation; this interface is intended for integration with an existing privacy management system in the service provider environment. By default, Parlay X Web service implementations contain an invalid endpoint for calling a privacy implementation. If no integration will be provided, then the privacy function can be disabled by setting the privacy common component endpoint to a blank endpoint.

The privacy interface allows for verification of the right to execute a service operation against a specific target. The result of a privacy check is to either allow or deny the execution of the operation against the target. A privacy check will be made for each target within an operation; this can be made in a single call or in multiple calls to the privacy interface.

The IBM WebSphere Telecommunications Web Services Server Information Center contains the Web Services Description Language (WSDL) PrivacyInterfaceService interface that defines the verifyPrivacyPermits() operation.

Integrating with database tables

The IBM WebSphere Telecommunications Web Services Server Information Center defines a number of database tables that contain audit information for billing and operations. The service provider is expected to capture the data from these database tables, and convert it to a form appropriate for their billing or operational support systems. This process is called *mediation*.

The tables that fit this category are:

- TRANSACTIONS
- NETWORKSTATISTICS
- CEI
- MSGLOG
- USAGERECORDS

After the transient auditing data has been captured, these tables need to be pruned. The IBM WebSphere Telecommunications Web Services Server software does not provide any automatic cleanup for these tables, and these tables will grow until cleared by the service provider.

Additionally, when doing capacity planning for a deployment, the service provider will need to determine the growth rate and the cleanup period for these tables based on the deployed services and the expected transaction rates, in order to determine the required database size.

Archive

Integrating with the IBM WebSphere Telecommunications Web Services Server Administration MBeans

The Administration MBeans described above are used to integrate with the IBM WebSphere Telecommunications Web Services Server Administration Console and provide a visualization of the configuration parameters; however, these same MBeans can be interacted with through WSADMIN Jython scripting to provide configuration information programmatically.

A convenient way to identify which MBean instance you are interested in is to navigate to that location in the IBM WebSphere Telecommunications Web Services Server Administration Console with a browser, and by viewing the properties of the page, you can see the real URL that references that page. Embedded in that URL is the ObjectName of the MBean instance that is manipulated by the page.

The ObjectName information, particularly the type= or name= parameter, can be used with the AdminService queryNames() JMX method or $AdminControl Jython object to resolve the MBean:

```
$ ./wsadmin.sh -lang jython

   mbean_list = AdminControl.queryMBeans("type=MyMBean,*")
length = mbean_list.size()
if length > 0:
    inst = mbean_list[0] # Just use the first one returned
    obj_name = str(inst.getObjectName())
    # Print out all attributes
    print AdminControl.getAttributes(obj_name, None)
```

Developing a JMX Event Listener

The IBM WebSphere Telecommunications Web Services Server Service Platform FaultAlarm component and the Access Gateway JMXEventMediator emit JMX notifications for alarm events. This information can be captured by a service provider's system software and appropriately handled.

Example B-1 shows an example of how this can be done. The service provider can provide a separate EAR that contains a servlet that initializes upon server start and listens for notifications from the NotificationService.

Example: B-1 JMX Event Listener code

```
import javax.management.Notification;
import javax.management.NotificationFilterSupport;
import javax.management.NotificationListener;
import javax.management.ObjectName;
import javax.servlet.http.HttpServlet;

import com.ibm.websphere.management.AdminService;
import com.ibm.websphere.management.AdminServiceFactory;
import com.ibm.websphere.management.NotificationConstants;
/**
 * Servlet that listens for JMX events for a server and prints them out.
 */
public class JMXListenerServlet extends HttpServlet {
```

Archived

```
    /**
     * Object name.
     */
    static ObjectName srvObjectName = null;

    /**
     * A listener.
     */
    static NotificationListener srvListener = null;

    /**
     * ObjectName of the mbean.
     */
    static ObjectName loggingObjectName = null;

    /**
     * A listener.
     */
    static NotificationListener loggingListener = null;

    /**
     * Called by the servlet container.  Indicates to a servlet that the
     * servlet is being placed into service.
     *
     * @throws javax.servlet.ServletException if an exception occurs that
     *     interrupts the servlet's normal operation
     */
    public void init() throws javax.servlet.ServletException {

        // Attach a listener to the server
        try {
            AdminService as = AdminServiceFactory.getAdminService();
            srvObjectName = as.getLocalServer();

            srvListener = new LoggingListener("Server");
            loggingListener = new LoggingListener("Logging");

            as.addNotificationListener(srvObjectName, srvListener, null, this);

            loggingObjectName = getLoggingServiceObjectName();
            if (loggingObjectName != null) {

                NotificationFilterSupport filter = new
NotificationFilterSupport();
                filter.enableType(NotificationConstants.TYPE_RAS_FATAL);
                filter.enableType(NotificationConstants.TYPE_RAS_ERROR);
                filter.enableType(NotificationConstants.TYPE_RAS_WARNING);
                filter.enableType(NotificationConstants.TYPE_RAS_INFO);
                filter.enableType(NotificationConstants.TYPE_RAS_AUDIT);
                filter.enableType(NotificationConstants.TYPE_RAS_SERVICE);

                as.addNotificationListener (loggingObjectName, loggingListener,
null, this);
            }
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    /**
     * Get ObjectName of the logging service.
     *
http://www-128.ibm.com/developerworks/websphere/techjournal/0304_lauzon/lauzon.htm
l
     *
     * @return the ObjectName
     */
    public static ObjectName getLoggingServiceObjectName() {
        ObjectName appMBean = null;

        try {

            AdminService as = AdminServiceFactory.getAdminService();

            // Get the current server name
            ObjectName server = as.getLocalServer();

            // Get WAS key properties
            String cell = server.getKeyProperty("cell");
            String node = server.getKeyProperty("node");
            String process = server.getKeyProperty("process");
            String serverName = server.getKeyProperty("name");

            // Find the notification mbean
            ObjectName queryMBean =
            new ObjectName(
                        "WebSphere:type=NotificationService"
                        + ",process="
                        + process
                        + ",*");
            Iterator appMBeanList =
            (Iterator) as.queryNames(queryMBean, null).iterator();
            if (appMBeanList != null && appMBeanList.hasNext()) {
                appMBean = (ObjectName) appMBeanList.next();
                loggingObjectName = appMBean;
            } else {
                loggingObjectName = null;
            }

        } catch (Exception e) {
            e.printStackTrace();
        }

        return appMBean;
    }
```

Archived

```
    /**
     * The server console listener waits for a stop request on the application
server.
     * @version %I%
     */
    class LoggingListener implements NotificationListener {

        private String type;

        /**
         * The constructor of this listener.
         *
         * @param type type
         */
        public LoggingListener(String type) {
            this.type = type;
        }

        /**
         * Handle the notification.
         *
         * @param not the server notification
         * @param obj the instance object passed in
         */
        public void handleNotification(Notification not, Object obj) {

           //System.err.println("JMXListenerServlet " + not.getType() + " " + not
+ " " + " obj: " + obj);
            if (not.getType().equals("j2ee.state.stopping")) {
                try {
                    // remove notifications
                    AdminService as = AdminServiceFactory.getAdminService();
                    if (srvListener != null) {

as.removeNotificationListener(as.getLocalServer(),srvListener);
                        srvListener = null;
                    }

                    if (loggingListener != null) {
                        as.removeNotificationListener(loggingObjectName,
loggingListener);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } else {

                // Event from the RasLoggingService
                try {

                    Object srcObject = not.getSource();
                    String srcName = srcObject.toString();
                    if (srcObject instanceof ObjectName) {
                        ObjectName srcObjectName = (ObjectName) srcObject;
                        srcName = srcObjectName.getKeyProperty("name");
```

```
                    }

                    int i = not.getType().lastIndexOf(".");
                    String eventType = not.getType().substring(i+1);

                    long sequence = not.getSequenceNumber();

                   String result = "ALARM: " + eventType + " " + sequence + " " +
srcName;

                    if (not.getTimeStamp() != 0) {
                        Date dt = new Date(not.getTimeStamp());
                        result += " [" + dt.toString() + "]";
                    }

                    // If there is a message provided, format it
                    if (not.getMessage() != null) {
                        result += " " + not.getMessage();
                    }

                    // If there is more user info, format it into the message
                   if (not.getUserData() instanceof com.ibm.ejs.ras.MessageEvent)
{
                        result += " " +
                                  ((com.ibm.ejs.ras.MessageEvent)
not.getUserData()).
                                  getLocalizedMessage(Locale.getDefault());
                    }

                    System.err.println(result);

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Developing a CEI Event Listener

WebSphere includes the Common Event Infrastructure components. IBM WebSphere Telecommunications Web Services Server FaultAlarm and Access Gateway CEIEmitter mediation primitives generate CEI events. Service Provider applications can be developed to subscribe or listen for these events. For developer guidance on how to subscribe to a CEI event, refer to the following Web link:

http://download.boulder.ibm.com/ibmdl/pub/software/dw/autonomic/books/cbepractice/index.pdf

Simple Network Management Protocol (SNMP) is frequently mentioned when integrating IBM WebSphere Telecommunications Web Services Server or WebSphere with other systems in the Service Provider's IT network environment. IBM WebSphere Telecommunications Web Services Server does not specifically provide support for SNMP, but there are a number of

Archived

other vendors and products that can integrate Java Management Extensions (JMX) with SNMP, particularly for SNMP traps that are events that are analogous to alarms.

HP Openview, IBM Tivoli NetView®, and AdventNet are popular platforms for monitoring JMX and SNMP events, and may be already in use in the Service Provider environment:

http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=solutions/adventnet&S_TACT=&S_CMP=campaign
http://www.adventnet.com/products/snmpadaptor/index.html
http://www-304.ibm.com/jct03001c/software/tivoli/products/netview/

IMS Client Toolkit

The WebSphere IP Multimedia Subsystem (IMS) Client Toolkit provides some additional information about how to integrate with IMS components in addition to IBM WebSphere Telecommunications Web Services Server.

The IMS components in IMS V6.2 are the Presence Server, the XML Document Management Server, the IMS Connector, and the IBM WebSphere Telecommunications Web Services Server.

The Presence Server provides status information and notifications about user presence. The XML Document Management Server provides a data store for service provider, user, and group operational information. The IMS Connector provide authentication, authorization, and auditing support for an IMS deployment. The IBM WebSphere Telecommunications Web Services Server provides a manageable platform for hosting Web Services that exposes service provider functions to third-party applications.

More information about the IMS Client Toolkit can be found at:

https://review.boulder.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-wsteltkt

Archived

C

Developing a Usage Record Billing Mediator common component

The IBM WebSphere Telecommunications Web Services Server Usage Record common component is a write-only component that records billing information. When integrating with a service provider environment, you should create a billing mediator application to extract Usage Records, process them to generate call detail records, and purge processed entries from the database. Usage Records are written to a table definition, a primary Usage Records table that contains the master record and service attributes.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

Table definitions for the Usage Records Billing Mediator

Table C-1 describes the table definitions for the Usage Records Billing Mediator.

Table C-1 USAGERECORDS table definitions for the Usage Records Billing Mediator

Field	Key	Data Type	Contents
RECORDID	PK	CHAR(36)	Unique record identifier.
SEGMENT	PK	SMALLINT	Segment number.
GLOBALID		VARCHAR(127)	Unique transaction identifier.
SERVICE		VARCHAR(127)	Service implementation name.
HOST		VARCHAR(255)	The host name of the machine emitting the record.
EVENTTYPE		VARCHAR(127)	Type of service implementation service usage record.
RECORDTIME		TIMESTAMP	Usage record recording time.
STATUSCODE		INTEGER	Status code regarding the service implementation state. Service specific.
SERVICEDATA		VARCHAR(7000)	Data contents for the usage record packed into a single string value. Packing is used to store data in a single field in order to reduce database space impact. The decision to pack all fields into a single data column was made after determining that the ratio of database and foreign key impact required to separate usage properties into a separate table was too great. Contents will be packed into the data field using the following scheme: key=value;key=value

Archive

A usage record may be split into segments, where each segment has a sequential segment number.

Other tables in IBM WebSphere Telecommunications Web Services Server also require similar mediation by the service provider:

- TRANSACTIONS
- NETWORKSTATISTICS
- MSGLOG

Note: The table definitions for each of these other tables can be looked up in the .DDL used to create them, and may differ slightly between types of databases.

Archived

Archived

D

Sample Usage Record cleanup program

This sample program demonstrates a method of removing Usage Records and other transient data records from a running system.

Archived

© Copyright IBM Corp. 2008. All rights reserved.

Sample code

Example D-1 illustrates the sample code for removing Usage Records and other transient data records from a running system.

Example: D-1 Sample code for removing Usage Records

```
//IBM Sample Source Materials
//
//Sample source materials are supplied As-Is.
//No warranty is expressed or implied.
//
//Product(s): 5724-005
//
//(C)Copyright IBM Corp. 2000, 2006 , 2007
//
//The source code for this program supplied under the terms of the
//End User License Agreement (EULA) that accompanied this product.
//*************************************************************


import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.text.NumberFormat;
import java.util.Date;
import java.util.Properties;

/**
* The IBM WebSphere Telecommunications Web Services Server Usage Records common component is a
write-only common which produces Usage Records for
* each IBM WebSphere Telecommunications Web Services Server service used. These Usage Records
are used for recording billing information.
* Once a usage record has been processed, the processed entry should be deleted from the
database.
*
* This Sample Usage Record pruning class removes usages records from a database based on the
records
* age. A SQL DELETE query is made of all the Usage Records older than the user provided input
date and time.
* This query limits each delete request to 100 deletes, to provide user feedback, and also
improve performance by
* issuing frequent database commits.
*
* Running the sample class CleanUpDbTable:
*
* Program assumptions:
*
*         - Run program on a IBM WebSphere Telecommunications Web Services Server server running
environment.
*         - Run program inside a database command-line environment
*         - Suggested JVM version java 1.41
```

```
*
* Running the Sample:
*
* set CLASSPATH=.;%CLASSPATH%
* java  CleanUpDbTable  DB2 TWSSDB62 localhost 50000 userid password 2007-09-14 11:00:00
TableName TimeFieldName PrimaryKeyName
*
* or
*
* java   -cp .;%CLASSPATH% CleanUpDbTable.class   DB2 TWSSDB62 localhost 50000 userid password
2007-09-14 11:00:00 TableName FieldName PrimaryKeyName
*
*
* Typical runtime problems:
*
*         Error:  ClassNotFoundException
*         Action: Ensure WAS Server and database is running or Program can't find JDBC driver in
classpath.
*
*         Error:  Incorrect Userid or Password
*         Action: Ensure correct information , recheck commmand line positions.
*
*         Error:  SQLException errors
*         Action: Check proper date or time command line input.
*
*/
public class CleanUpDbTable {

   /**
    * Error messages
    */
   static final String ILLEGAL_ACCESS = "IllegalAccessException during driver resolution: " ;
   static final String INSTANTIATION_EXCEPTION = "InstantiationException during driver
resolution: ";
   static final String CLASS_NOT_FOUND = "ClassNotFoundException during driver resolution:  ";
   static final String SQL_DRIVER = "SQLException during driver getConnection:  ";
   static final String SQL_COUNT = "SQLException during UseageRecord count query: ";
   static final String SQL_EXECUTE = "SQLException during query executeUpdate: ";
   static final String SQL_DELETE =  "SQLException during query delete processing: " ;
   static final String SQL_CLOSE = "Error closing connection: " ;

   /**
    * Informational messages
    */
   static final String USAGE1 = "Usage: database-alias server-hostname portNumber userId
password date(ex:2007-08-01) time(ex:11:00:00) TableName FieldName PrimaryKeyName\n"+
    "Example parameters: [DB2|ORACLE] WPS_AL dbserverhostname 50000 username <password>
2007-09-01 11:00:00 USAGERECORDS RECORDTIME RECORDID";
   static final String MSG1  = "Starting query count of Usage Records to delete...";
   static final String MSG2  = "SQL Query = \n";
   static final String MSG3  = "Usage Records = " ;
   static final String MSG4  = "Committed delete UsageRecords row(s): ";
   static final String MSG5  = "Current elasped time is: ";
   static final String MSG6  = "Averaging delete time: " ;
   static final String MSG7  = "Total UsageRecord cleanup time:  ";
```

```
static final String MSG8  = "UsageRecord total delete count query time: " ;
static final String MSG9  = "UsageRecord count query matching row(s) count: ";
static final String MSG10 = "Starting Delete usage query...";
static final String MSG11 = "Preparing jdbc URL: ";
static final String MSG12 = "Preparing jdbc driver class: ";
static final String MSG13 = "No UsageRecords found matching query criteria. ";

/**
 * Remember database type
 */
static boolean isDB2 = false;
static boolean isOracle = false;


/**
 * URL property file inputs
 */
static final String USERID = "user";
static final String PW = "password";

/**
 * CommandLine parameter inputs
 */
static final int _DB = 0 ;
static final int _ALIAS = 1 ;
static final int _SERVER = 2 ;
static final int _PORTNUM = 3 ;
static final int _USERID = 4 ;
static final int _PW = 5 ;
static final int _TIME = 6 ;
static final int _DATE = 7 ;
static final int _TABLE = 8 ;
static final int _TIMEFIELD = 9 ;
static final int _PRIMARYKEY = 10 ;
static final int MAX_PARM = 11;

/**
 * Java Main Application entry point
 *
 * @param args
 */
public static void main(String[] args) {

   CleanUpDbTable example = new CleanUpDbTable();

   // check for parameters
   if (args.length > MAX_PARM || args.length < MAX_PARM) {
      example.myexit (-1 , USAGE1 );
   }

   example.processCmd(args);

   String alias = args[_ALIAS] ;
   String server = args[_SERVER];
   int portNumber = Integer.valueOf(args[_PORTNUM]).intValue();
```

Archived

```
String userId = args[_USERID];
String password = args[_PW];
String tableName = args[_TABLE];
String timeFieldName = args[_TIMEFIELD];
String primaryKeyName = args[_PRIMARYKEY];

// for collecting some simple satistics
Date startTime = new Date();
int count = 0;
int records = 0;
Connection connection = null;
String url = null;
String driver = null;
if (isDB2) {
   url = "jdbc:db2://" + server + ":" + portNumber + "/" + alias;
   driver = "com.ibm.db2.jcc.DB2Driver";
} else {
   url = "jdbc:oracle:thin:@" + server + ":" + portNumber + ":" + alias;
   driver = "oracle.jdbc.driver.OracleDriver";
}

// MSG11 = "Preparing jdbc URL: ";
// MSG12 = "Preparing jdbc driver class: ";
example.tracemsg (MSG11 + url);
example.tracemsg (MSG12 + driver);


try {
   Class.forName(driver).newInstance();
} catch (IllegalAccessException e) {
       e.printStackTrace();
   example.myexit(-2,ILLEGAL_ACCESS + e);
} catch (InstantiationException e) {
         e.printStackTrace();
   example.myexit(-2,INSTANTIATION_EXCEPTION + e);
} catch (ClassNotFoundException e) {
          e.printStackTrace();
   example.myexit(-2,CLASS_NOT_FOUND + e);
}

Properties props = new Properties();
props.setProperty(USERID, userId);
props.setProperty(PW, password);

try {
   connection = DriverManager.getConnection(url, props);
   connection.setAutoCommit(false);
} catch (SQLException e) {
          e.printStackTrace();
   example.myexit(-3,SQL_DRIVER + e);
}

if (connection != null) {

   try {
```

Archived

```
            // msg1: "Starting count of Usage Records to delete...";
            example.tracemsg (MSG1);
            //-----------------------------------------------------
            // Query the DB for a total all records older
            // than the time/date provided
            //-----------------------------------------------------
            Statement statement = connection.createStatement();

            String query = "SELECT  COUNT( " + timeFieldName + " ) AS TOTAL  FROM " + tableName +
" WHERE (" + timeFieldName + " < TIMESTAMP('"
               + args[_TIME] + " "
               + args[_DATE].trim() + "'))";
            // msg2:"Using query: ";
            example.tracemsg (MSG2+ query);


            //--------------------------------
            // Perform Usage Record Count
            //--------------------------------
            ResultSet rs = statement.executeQuery(query);
            rs.next();
            records = new Integer(rs.getInt("TOTAL")).intValue()  ;

            // records = Integer.parseInt(recordCountString);
            // msg3:"Usage Records = " + records
            example.tracemsg (MSG3+ records);
            rs.close();
            statement.close();

         } catch (SQLException e) {
               e.printStackTrace();
            example.myexit(-4,SQL_COUNT + e);
         }

         Date finshTime = new Date();
         // MSG8 = "UsageRecord total delete count query time: " ;
         // MSG9 = "UsageRecord count query matching row(s) count:";
         example.tracemsg (MSG8+  example.timeInSeconds(finshTime.getTime() -
startTime.getTime()));
         example.tracemsg (MSG9+records)  ;


         // Check if any delete matching criteria
         if (records != 0) {

            // Prepare to delete and commit 100 Usage Records per loop
            count = 0;
            startTime = new Date();

            //msg10 = "Starting Delete usage query...";
            example.tracemsg (MSG10)  ;

             String query = "DELETE FROM " + tableName + " WHERE " + primaryKeyName + " in
(SELECT " + primaryKeyName + " FROM " + tableName + " WHERE " + timeFieldName + " < TIMESTAMP('"
             + args[_TIME] + " "
```

```
             + args[_DATE].trim() + "') ORDER BY " + primaryKeyName + " FETCH FIRST 100 ROWS
ONLY)";

            example.tracemsg (MSG2+ query);

            try {
               PreparedStatement statement = connection.prepareStatement(query);
               Date runningTime = new Date();

               int delcnt = 0 ;
               while (records > 0) {

                  try {

                     delcnt = 0;
                     //--------------------------------
                     // Perform Usage Record Delete
                     //--------------------------------
                     delcnt= statement.executeUpdate();

                  } catch (SQLException e) {
                        e.printStackTrace();
                     example.myexit(-4,SQL_EXECUTE + e);
                  }
                  count+= delcnt;
                  records -= delcnt;

                  //-------------------------------------------
                  // To prevent accessive locks commit often and
                  // improves preformance
                  //-------------------------------------------
                  connection.commit();

                  finshTime = new Date();
                  long intermediateTime = finshTime.getTime() - runningTime.getTime();


                  // msg4: "Committed delete UsageRecords row(s): " +delcnt
                  // msg5: "Current elasped time is: "+
example.timeInSeconds((finshTime.getTime() - startTime.getTime()))
                  // msg6: "Averaging delete time in seconds: " +
example.timeInSeconds(intermediateTime / 100));
                  example.tracemsg (MSG4+ delcnt );
                  example.tracemsg (MSG5+  example.timeInSeconds( (finshTime.getTime() -
startTime.getTime()) ) );
                  example.tracemsg (MSG6+  example.timeInSeconds(intermediateTime / 100));

                  runningTime = new Date();
               }

               statement.close();

            } catch (SQLException e) {
                e.printStackTrace();
               example.myexit(-4,SQL_DELETE + e);
```

```
            }

            finshTime = new Date();
            // msg7  "Total UsageRecord cleanup time:  ";
            example.tracemsg (MSG7+ example.timeInSeconds(finshTime.getTime() -
startTime.getTime())) ;

         } else { // if count
            // MSG13 = "No UsageRecords found matching query criteria. ";
            example.tracemsg (MSG13) ;

         }
      }  // if connection
      if (connection != null) {
         // Disconnecting ...
         try {
            connection.commit();
            connection.close();
         } catch (SQLException e) {
               e.printStackTrace();
            example.myexit(-4,SQL_CLOSE + e);

         }
      }

      example.myexit(0,"");

   }

   /** A number formatter*/
   private NumberFormat nf = NumberFormat.getInstance();

   /**
    * Convert time in milliseconds to a String in seconds
    * @param time
    * @return
    */
   private  String timeInSeconds(float time) {
      return nf.format(time/1000F) + " seconds";
   }


   /**
    * processCmd:  Check input parms.
    * @param args  Program input parms.
    */

   private void processCmd (String[] args ){

      // Did the user request help?
      if ( (args[_DB].indexOf("?")> -1) || (args[_DB].trim().toLowerCase().indexOf("help")> -1)
) {
         myexit (-1 , USAGE1 );

         // Check for DB2
```

Archived

```
      }  else if (  (args[_DB].trim().toUpperCase().indexOf("DB2") > -1) ) {
         isDB2 = true;
         // Check for Oracle
      }  else if (  (args[_DB].trim().toUpperCase().indexOf("ORACLE") > -1) ) {

      } else {
         myexit (-1 , "" );

      }

   }


   /**
    * tracemsg: Output messages
    * @param message display message
    */

   private void tracemsg (String message ){

      System.out.println(String.valueOf(message));
   }

   /**
    * Exit: System exit
    * @param retc    return error code
    * @param errmsg  display message
    */

   private void myexit(int retc , String errmsg  ){

      tracemsg(String.valueOf(errmsg));
      System.exit(retc);

   }
}
```

Archived

Archived

E

Additional material

This book refers to additional material that can be downloaded from the Internet, as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG247589

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247589.

Using the Web material

The additional Web material that accompanies this book includes the files listed in Table E-1.

Table E-1 Files available for download

File name	Description
CH52 Sample.zip	Project interchange file. Contains the PresenceSupplier publish Mediation Module, including the custom primitive, as described in 5.2, “Customize a default mediation flow” on page 128.

© Copyright IBM Corp. 2008. All rights reserved.

File name	Description
CH53 Sample plugin.zip	Project interchange file. Contains both projects that are required to create the WebSphere Integration Developer plug-in, as described in 5.3, “Extending the WebSphere Integration Developer Tooling Environment” on page 153.
CH53 Sample flow.zip	Project interchange file. Contains the PresenceSupplier publish Mediation Module, including the custom mediation primitive, as described in 5.3.6, “Using the plug-in to customize a flow” on page 168.
CH53 Sample java.zip	ZIP archive. Contains the source code of the AdressTransformation.java class, which is required in 5.3.3, “Develop the mediation business logic” on page 162.
CH54 Samplw.zip	Project interchange file. Contains the CONFERENCING Mediation Module, as described in 5.4, “Develop a custom mediation flow” on page 173.
PX21_PRS_SPLR_IMSApp.ear	Sample service implementation of the Presence supplier service.
ParlayXPresenceSupplier_JavaDoc.zip	Java Documentation.
PresenceSupplierClient.zip	Simple Testing Client for Presence Supplier.
PresenceSupplierClientApp.ear	Simple Testing Client for Presence Supplier.

Archive

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

For information about ordering these publications, see “How to get Redbooks” on page 401. Note that some of the documents referenced here may be available in softcopy only.

- *Developing SIP and IP Multimedia Subsystem (IMS) Applications*, SG24-7255, found at:

  http://www.redbooks.ibm.com/abstracts/sg247255.html?Open

Online resources

These Web sites are also relevant as further information sources:

- IBM WebSphere Telecommunications Web Services Server Information Center

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp

- Javadoc for IBM WebSphere Telecommunications Web Services Server

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/intro_c.html

- WSDL documentation for common components

  http://publib.boulder.ibm.com/infocenter/wtelecom/v6r2m0/index.jsp?topic=/com.ibm.twss.javadoc.doc/wsdl_c.html

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

© Copyright IBM Corp. 2008. All rights reserved.

Archived

Index

© Copyright IBM Corp. 2008. All rights reserved.

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

Archived

S

T

U

V

W

X

Archived

IBM WebSphere Telecommunications Web Services Server Programming Guide

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

Redbooks

Archived

Archived

IBM WebSphere Telecommunications Web Services Server Programming Guide

Architecture and component overview

Design considerations and best practices

Creating a custom service implementation

This IBM Redbooks publication is a programming guide that provides developers with the information they need to create new Web service implementations for IBM WebSphere Telecommunications Web Services Server.

IBM WebSphere Telecommunications Web Services Server allows you to expose high-level Web service interfaces to network services for third parties.

Third parties are typically external service providers, customers, or organizational divisions that want to develop new services that integrate with the service provider network infrastructure. Web service interfaces provide access to service capabilities in a programming language and technology independent way. Each Web service interface can have multiple back-end implementations for connecting with a service provider's environment. For example, a Web service interface may connect to a service provider's network through the Session Initiation Protocol (SIP), using a Parlay Connector through a Parlay Gateway, through native service provider protocols, or using custom integrated services.

In this IBM Redbooks publication, we provide specific references, best practices, guidance, and implementation examples for programming IBM WebSphere Telecommunications Web Services Server components and customize it for your organization's particular needs. More specifically, we discuss the following items within the context of a common example scenario:

- Working with the Service Policy Manager and creating a custom policy
- Working with the Access Gateway to make modifications to a default mediation flow, create a custom mediation primitive, or create a completely new mediation flow from scratch
- Creating a custom Parlay X Service Implementation, which is based on creating the Publish ( ) operation within a Presence Supplier interface.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7589-00 ISBN 0738431427