Oracle® Fusion Middleware

Concepts Guide for Oracle Infrastructure Web Services

11*g* Release 1 (11.1.1)

E15184-03

January 2011

This document introduces you to Oracle Infrastructure Web services.

ORACLE®

Oracle Fusion Middleware Concepts Guide for Oracle Infrastructure Web Services, 11*g* Release 1 (11.1.1)

E15184-03

Primary Author: Oracle Corporation

Contents

9 Interoperability Guidelines

A Annotation Reference

Preface

This preface describes the intended audience, document accessibility features, and conventions used in this guide—*Concepts Guide for Oracle Infrastructure Web Services*.

Audience

This document is intended for programmers that are developing Oracle Infrastructure Web services, including SOA, ADF, and WebCenter services.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/support/contact.html` or visit `http://www.oracle.com/accessibility/support.html` if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Fusion Middleware 11*g* Release 1 (11.1.1) documentation set:

- *Oracle Fusion Middleware Introducing Web Services*
- *Security and Administrator's Guide for Web Services*
- *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*
- *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Developer's Guide for Oracle WebCenter*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introducing Oracle Infrastructure Web Services

This chapter introduces Oracle Infrastructure Web services and describes the standards supported.

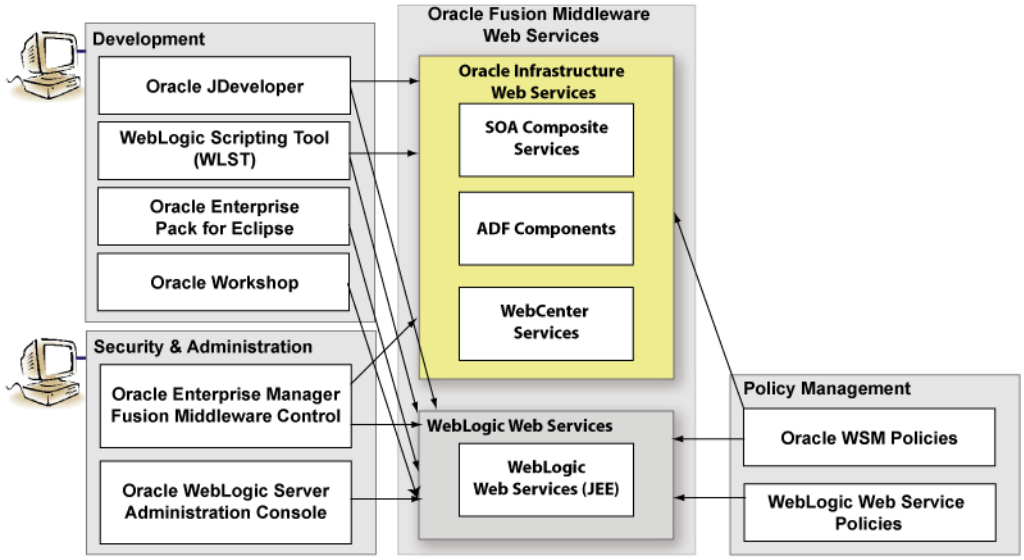
1.1 Overview of Oracle Infrastructure Web Services

In Oracle Fusion Middleware 11*g*, there are two categories of Web services to support the development, security, and administration of the following types of Web services:

- Oracle Infrastructure Web services—SOA, ADF, and Web Center services
- WebLogic Web services (Java EE Web services)

The following figure illustrates the two Web services categories. Oracle Infrastructure Web services are highlighted (in yellow) in the figure.

Figure 1–1 Web Services in Oracle Fusion Middleware 11g



Note: For more information about the full set of components shown in the figure, including the development; security and administration; and policy management tools, see *Introducing Web Services*.

This chapter describes concepts for developing Oracle Infrastructure Web services. For more information about WebLogic Web services, see *Introducing WebLogic Web Services for Oracle WebLogic Server*.

1.2 Types of Oracle Infrastructure Web Services and Clients

Table 1–1 summarizes the types of Oracle Infrastructure Web services supported in Oracle Fusion Middleware 11*g*.

Table 1–1 Oracle Infrastructure Web Services

Web Service	Description
SOA service components	SOA composite applications include SOA service components. SOA service components are the basic building blocks of SOA applications, implementing a part of the overall business logic functionality. The following SOA service components can be managed using Oracle WSM: ■ BPEL Process—Provides process orchestration and storage of synchronous and asynchronous processes. ■ Oracle Mediator—Routes events (messages) between different components. ■ Human Workflow—Enables you to model a workflow that describes the tasks for users or groups to perform as part of an end-to-end business process flow. ■ Business Rules—Design a business decision based on rules. For more information about developing SOA service components, see *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*. You can deploy SOA service components to the Oracle Fusion Middleware environment.
SOA service binding components	SOA Service binding components provide the outside world with an entry point to the SOA composite application. The WSDL file of the service advertises its capabilities to external applications. These capabilities are used for contacting the SOA composite application components. For more information, see *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

Table 1–1 (Cont.) Oracle Infrastructure Web Services

Web Service	Description
ADF Business Components	ADF Business Components simplify the development, delivery, and customization of business applications for the Java EE platform by providing a library of reusable components and supporting design time facilities in Oracle JDeveloper. Using ADF Business Components, developers are not required to write the application infrastructure code required by the typical Java EE application to perform the following tasks: ■ Connect to the database. ■ Retrieve data. ■ Lock database records. ■ Manage transactions. Additionally, Oracle JDeveloper facilities expose ADF Business Component application modules that encapsulate built-in data manipulation operations and custom methods as Web services so that a service-enabled application module can be consumed across modules of the deploy Fusion Web application. For more information, see "Integrating Service-Enabled Application Modules" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
WebCenter services	WebCenter services expose Web 2.0 technologies for social networking and personal productivity, such as Wiki, RSS, and blogs. WebCenter provides a set of features and services (for example, portlets, customization, and content integration) that you can selectively add to your application. For more information about developing WebCenter services, see *Oracle Fusion Middleware Developer's Guide for Oracle WebCenter*.

Table 1–2 summarizes the types of Oracle Infrastructure Web service clients supported in Oracle Fusion Middleware 11*g*.

Table 1–2 Oracle Infrastructure Web Service Clients

Web Service Client	Description
SOA reference binding components	SOA reference binding components connect the SOA composite application to external partners. For more information about developing SOA reference binding components, see *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
ADF Web applications	ADF Web applications can invoke a service, such as a WebLogic Web service, a SOA composite application, or a service-enabled ADF application module. For more information, see *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*. In addition, ADF Web applications can work with Web services in the user interface using a Web service data control. For more information about generating service-enabled application modules, calling a Web service from an ADF application module, or creating Web service data controls, see *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.
WebCenter portlets	WebCenter portlets enable you to surface WebCenter services. For more information about developing WebCenter portlets, see *Oracle Fusion Middleware Developer's Guide for Oracle WebCenter*.

1.3 Supported Standards

The following table summarizes the Oracle Infrastructure Web service specifications that are part of the Oracle implementation, organized by high-level feature.

Oracle considers interoperability of Web services platforms to be more important than providing support for all possible edge cases of the Web services specifications. Oracle complies with the following specifications from the Web Services Interoperability Organization and considers them to be the baseline for Web services interoperability:

- *Basic Profile 1.1 and 1.0*: http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html
- *Basic Security Profile 1.0*: http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html
- *WS-I Attachments Profile 1.0*: http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html

> **Note:** For more information about Oracle Infrastructure Web service security standards, see "Web Services Security Standards" in *Security and Administrator's Guide for Web Services*.

Table 1–3 Specifications Supported by Oracle Infrastructure Web Services

Feature	Specification
Programming model (based on metadata annotations) and runtime architecture	**Web Services Metadata Exchange (WS-MetadataExchange) 1.1**—Part of the WS-Federation roadmap which allows retrieval of metadata about a Web service endpoint. For more information, see *Web Services Metadata Exchange (WS-MetadataExchange)* specification at http://xml.coverpages.org/WS-MetadataExchange.pdf.
Web service description	■ **Web Services Description Language (WSDL) 1.1 and 2.0**—XML-based specification that describes a Web service. For more information, see *Web Services Description Language (WSDL)* at http://www.w3.org/TR/wsdl. ■ **Web Services Policy Framework (WS-Policy) 1.5 and 1.2**—General purpose model and corresponding syntax to describe and communicate the policies of a Web service. For more information, see: *WS-Policy 1.5 Framework* (Recommendation): http://www.w3.org/TR/ws-policy/ *WS-Policy 1.2 Framework* (Member Submission): http://www.w3.org/Submission/WS-Policy ■ **Web Services Policy Attachment (WS-PolicyAttachment) 1.5 and 1.2**—Abstract model and an XML-based expression grammar for policies. For more information, see: *WS-Policy Attachment 1.5* (Recommendation): http://www.w3.org/TR/ws-policy-attach/ *WS-PolicyAttachment 1.2* (Member Submission): http://www.w3.org/Submission/WS-PolicyAttachment

Table 1–3 (Cont.) Specifications Supported by Oracle Infrastructure Web Services

Feature	Specification
Data exchange between Web service and requesting client	■ **Simple Object Access Protocol (SOAP) 1.1 and 1.2**—Lightweight XML-based protocol used to exchange information in a decentralized, distributed environment. For more information, see *Simple Object Access Protocol (SOAP)* at http://www.w3.org/TR/SOAP. ■ **SOAP with Attachments API for Java (SAAJ) 1.3**—Implementation that developers can use to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments notes. For more information, see the *SOAP with Attachments API for Java (SAAJ)* specification at https://saaj.dev.java.net. ■ **Message Transmission Optimization Mechanism (MTOM)** you can specify that a Web service use a streaming API when reading inbound SOAP messages that include attachments, rather than the default behavior in which the service reads the entire message into memory. For more information, see *SOAP Message Transmission Optimization Mechanism* specification at http://www.w3.org/TR/soap12-mtom/.

Table 1–3 (Cont.) Specifications Supported by Oracle Infrastructure Web Services

Feature	Specification
Security	■ **Web Services Security (WS-Security) 1.0 and 1.1**—Standard set of SOAP [SOAP11, SOAP12] extensions that can be used when building secure Web services to implement message content integrity and confidentiality. For more information, see *OASIS Web Service Security Web* page at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss. Web services security supports the following security tokens: - Username—defines how a Web service consumer can supply a username as a credential for authentication). - X.509 certificate—a signed data structure designed to send a public key to a receiving party. - Kerberos ticket—a binary authentication and session token. - Security Assertion Markup Language (SAML) assertion—shares security information over the Internet through XML documents For more information, see "Web Service Security Standards" in *Security and Administrator's Guide for Web Services*. ■ **Web Services Security Policy (WS-SecurityPolicy) 1.3, 1.2, and 1.1**—Set of security policy assertions for use with the WS-Policy framework. For more information, see *Web Services Security Policy (WS-SecurityPolicy) 1.3* specification at http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200802 *Web Services Security Policy (WS-SecurityPolicy) 1.2* specification at http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html *Web Services Security Policy (WS-SecurityPolicy) 1.1* specification at http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf ■ **Security Assertion Markup Language (SAML) 2.0**—XML standard for exchanging authentication and authorization data between security domains. For more information, see the *Security Assertion Markup Language (SAML)* specification at http://docs.oasis-open.org/security/saml/v2.0/. ■ **Security Assertion Markup Language (SAML) Token Profile 1.1**—Set of SOAP extensions that implement SOAP message authentication and encryption. For more information, see the *Security Assertion Markup Language (SAML) Token Profile 1.1* specification at http://www.oasis-open.org/committees/download.php/16768/wss-v1.1-spec-os-SAMLTokenProfile.pdf. ■ **WS-Trust**—Defines extensions to WS-Security that provide a framework for requesting and issuing security tokens, and to broker trust relationships. For more information, see http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html ■ **XML Signature**—Defines an XML syntax for digital signatures. For more information, see *XML Signature Syntax and Processing* at http://www.w3.org/TR/xmldsig-core/. ■ **XML Encryption**—Defines how to encrypt the contents of an XML element. For more information, see *XML Encryption Syntax and Processing* at http://www.w3.org/TR/xmlenc-core/.

Table 1–3 (Cont.) Specifications Supported by Oracle Infrastructure Web Services

Feature	Specification
Reliable communication	▪ **Web Services Addressing (WS-Addressing) 1.0**—Transport-neutral mechanisms to address Web services and messages. For more information, see Web Services Addressing (WS-Addressing) specification at http://www.w3.org/TR/ws-addr-core. ▪ **Web Services Reliable Messaging (WS-ReliableMessaging) 1.0 and 1.1**—Implementation that enables two Web services running on different WebLogic Server instances to communicate reliably in the presence of failures in software components, systems, or networks. For more information, see: *Web Services Reliable Messaging (WS-ReliableMessaging) 1.1* specification at http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf. *Web Services Reliable Messaging (WS-ReliableMessaging) 1.0* specification at http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf. ▪ **Web Services Reliable Messaging Policy (WS-ReliableMessaging Policy) 1.1**—Domain-specific policy assertion for reliable messaging for use with WS-Policy and WS-ReliableMessaging. For more information, see *Web Services Reliable Messaging Policy (WS-ReliableMessaging Policy)* specification at http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.pdf
Atomic transactions	**Web Services Atomic Transaction**—Defines the Atomic Transaction coordination type that is to be used with the extensible coordination framework described in the Web Services Coordination specification. The WS-AtomicTransaction and WS-Coordination specifications define an extensible framework for coordinating distributed activities among a set of participants. For more information, see: ▪ WS-AtomicTransaction: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-cs-01/wstx-wsat-1.2-spec-cs-01.html ▪ WS-Coordination: http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-cs-01/wstx-wscoor-1.2-spec-cs-01.html,
Advertisement (registration and discovery)	▪ **Universal Description, Discovery, and Integration (UDDI) 2.0**—Standard for describing a Web service; registering a Web service in a well-known registry; and discovering other registered Web services. For more information, see the *Universal Description, Discovery, and Integration (UDDI)* specification at http://uddi.xml.org. ▪ **Web Services Inspection Language 1.0**—Provides an XML format for assisting in the inspection of a site for available services. For more information, see Web Services Inspection Language (WS-Inspection) 1.0 specification at http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-wsilspec/ws-wsilspec.pdf.

1.4 Related Documentation

The following table summarizes the documentation that is related to Oracle Infrastructure Web services development, security, and administration.

Table 1–4 Related Documentation

Document	Description
Oracle Fusion Middleware Introducing Web Services	This document. Provides an introduction to Web services for Oracle Fusion Middleware 11g.
Security and Administrator's Guide for Web Services	Describes how to secure and administer Web services.

Table 1–4 (Cont.) Related Documentation

Document	Description
Extensibility Guide for Oracle Web Services Manager	Describes how to build custom assertions for Oracle Web Services Manager (Oracle WSM).
Oracle Fusion Middleware Interoperability Guide for Oracle Web Services Manager	Describes how to implement the most common Oracle WSM interoperability scenarios.
Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite	Describes how to develop SOA composite services.
Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework	Describes how to develop ADF components.
Oracle Fusion Middleware Developer's Guide for Oracle WebCenter	Describes how to develop WebCenter services.
"Developing with Web Services" in the "Designing and Developing Applications" section of the Oracle JDeveloper online help	Describes how to develop Web services and attach policies using Oracle JDeveloper.
Oracle Workshop for WebLogic Server	Explains how to use Workshop to write and manage source code and design with sophisticated visual tools and Java frameworks.

2

Attaching Policies to Oracle Infrastructure Web Services

This chapter describes how to attach policies to Oracle Infrastructure Web services.

2.1 What Are Policies?

Policies describe the capabilities and requirements of a Web service such as whether and how a message must be secured, whether and how a message must be delivered reliably, and so on.

Oracle Fusion Middleware 11*g* Release 1 (11.1.1) supports the types of policies defined in Table 2–1.

Table 2–1 Types of Policies

Policy	Description
WS-ReliableMessaging	Reliable messaging policies that implement the WS-ReliableMessaging standard describes a wire-level protocol that allows guaranteed delivery of SOAP messages, and can maintain the order of sequence in which a set of messages are delivered. The technology can be used to ensure that messages are delivered in the correct order. If a message is delivered out of order, the receiving system can be configured to guarantee that the messages will be processed in the correct order. The system can also be configured to deliver messages at least once, not more than once, or exactly once. If a message is lost, the sending system re-transmits the message until the receiving system acknowledges it receipt.
Management	Management policies that log request, response, and fault messages to a message log. Management policies may include custom policies.

Table 2–1 (Cont.) Types of Policies

Policy	Description
WS-Addressing	WS-Addressing policies that verify that SOAP messages include WS-Addressing headers in conformance with the WS-Addressing specification. Transport-level data is included in the XML message rather than relying on the network-level transport to convey this information.
Security	Security policies that implement the WS-Security 1.0 and 1.1 standards. They enforce message protection (message integrity and message confidentiality), and authentication and authorization of Web service requesters and providers. The following token profiles are supported: username token, X.509 certificate, Kerberos ticket, and Security Assertion Markup Language (SAML) assertion. For more information about Web service security concepts and standards, see *Security and Administrator's Guide for Web Services*.
Message Transmission Optimization Mechanism (MTOM)	Binary content, such as an image in JPEG format, can be passed between the client and the Web service. In order to be passed, the binary content is typically inserted into an XML document as an `xsd:base64Binary` string. Transmitting the binary content in this format greatly increase the size of the message sent over the wire and is expensive in terms of the required processing space and time. Using Message Transmission Optimization Mechanism (MTOM), binary content can be sent as a MIME attachment, which reduces the transmission size on the wire. The binary content is semantically part of the XML document. Attaching an MTOM policy ensures that the message is converted to a MIME attachment before it is sent to the Web service or client.

2.2 What are Policy Sets?

A policy set, which can contain multiple policy references, is an abstract representation that provides a means to attach policies globally to a range of subjects of the same type. Attaching policies globally using policy sets provides a mechanism for the administrator to ensure that all subjects are secured in situations where the developer, assembler, or deployer did not explicitly specify the policies to be attached. Policies that are attached using a policy set are considered externally attached.

Policy subjects to which policy sets can be attached include SOA components, SOA service endpoints, SOA references, Web services endpoints, Web service clients, Web service connections, and asynchronous callback clients. Policy sets can be attached at the following scopes:

- Domain — all services in a domain
- Server instance—all services in a server instance
- Application—all services in an application
- SOA composite—all services in a SOA composite
- Application Module—all services in an application module

For details about using policy sets to globally attach policies, see "Attaching Policies Globally Using Policy Sets" in *Security and Administrator's Guide for Web Services*.

2.3 Oracle WSM Predefined Policies and Assertion Templates

Oracle Web Services Manager (WSM) provides a policy framework to manage and secure Web services consistently across your organization. Oracle WSM can be used by both developers, at design time, and system administrators in production environments. For more information about the Oracle WSM policy framework, see "Understanding Oracle WSM Policy Framework" in *Security and Administrator's Guide for Web Services*.

There is a set of predefined Oracle WSM policies and assertion templates that are automatically available when you install Oracle Fusion Middleware. The predefined policies are based on common best practice policy patterns used in customer deployments.

You can immediately begin attaching these predefined policies to your Web services or clients. You can configure the predefined policies or create a new policy by making a copy of one of the predefined policies.

Predefined policies are constructed using assertions based on predefined assertion templates. You can create new assertion templates, as required.

For more information about the predefined Oracle WSM policies and assertion templates, see the following sections in *Security and Administrator's Guide for Web Services*:

- "Predefined Policies"
- "Predefined Assertion Templates"

2.4 Attaching Policies to Web Services Using Annotations

You can use annotations defined in Table 2–2 to attach policies to Web services. The annotations are included in the oracle.webservices.annotations and oracle.webservices.annotations.async packages.

For more information about the annotations available, see *Oracle Fusion Middleware Java API Reference for Oracle Web Services*. For more information about the predefined policies, see "Predefined Polices" in *Security and Administrator's Guide for Web Services*.

Table 2–2 Annotations for Attaching Policies to Web Services

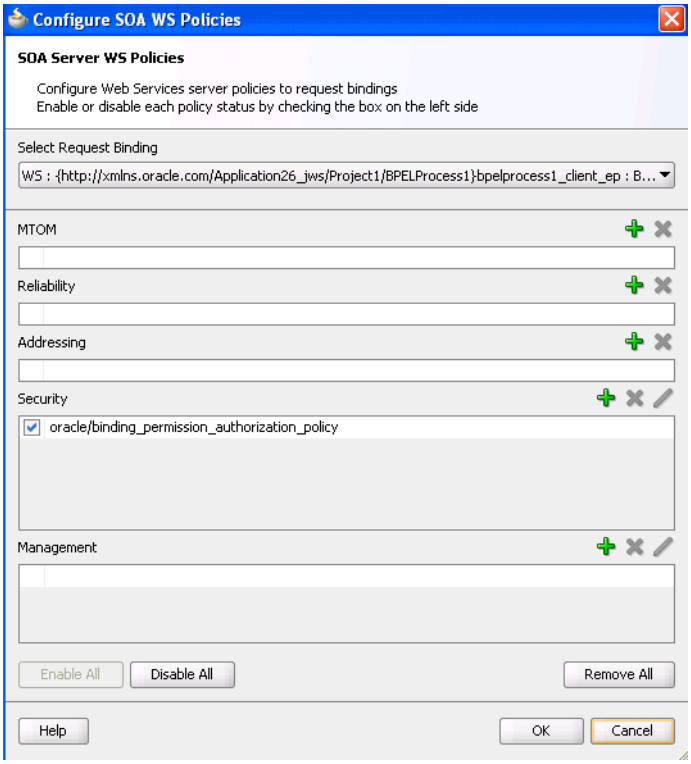
Annotation	Description
@AddressingPolicy	Attaches a WS-Addressing policy to the Web service. For more information, see Section A.2, "@AddressingPolicy Annotation".
@CallbackManagementPolicy	Attaches a management policy to the callback client of the asynchronous Web service that will connect to the callback service. For more information, see Section A.7, "@CallbackManagementPolicy Annotation".
@CallbackMtomPolicy	Attaches an MTOM policy to the callback client of the asynchronous Web service that will connect to the callback service. For more information, see Section A.9, "@CallbackMtomPolicy Annotation".
@CallbackSecurityPolicy	Attaches one or more security polices to the callback client of the asynchronous Web service that will connect to the callback service. By default, no security policies are attached. For more information, see Section A.11, "@CallbackSecurityPolicy Annotation".
@ManagementPolicy	Attaches a management policy to the Web service. For more information, see Section A.12, "@ManagementPolicy Annotation".
@MtomPolicy	Attaches an MTOM policy to the Web service. For more information, see Section A.13, "@MtomPolicy Annotation".
@SecurityPolicies	Specifies an array of @SecurityPolicy annotations. Use this annotation if you want to attach more than one WS-Policy files to a class. For more information, see Section A.19, "@SecurityPolicies Annotation".
@SecurityPolicy	Attaches a security policy to the Web service. For more information, see Section A.20, "@SecurityPolicy Annotation".

2.5 Attaching Policies Using Oracle JDeveloper

When creating an application using JDeveloper, you can take advantage of the wizards available to attach policies to Web services and clients.

For example, the following figure shows the Configure SOA WS Policies wizard that you can use to attach policies to SOA service or reference binding components quickly and easily.

Figure 2–1 Configure SOA WS Policies Wizard



For more information, see:

- "Managing Policies" and "Attaching Policies to Binding Components and Service Components" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
- "Securing Web Service Data Controls" in *Oracle Application Development Framework Developer's Guide*
- "Creating Web Service Proxies" in the *JDeveloper Online Help*.

2.6 Attaching Policies Using Oracle Enterprise Manager

After a Web service or client is deployed, you can attach policies directly to endpoints using the Oracle Enterprise Manager Fusion Middleware Control. You can also attach policies globally to a set of endpoints using policy sets.

For example, Figure 2–2 shows the OWSM Policies tab on the Web Service Endpoint page from which you can attach policies to a Web service endpoint.

Figure 2–2 Attaching Policies Using Oracle Enterprise Manager

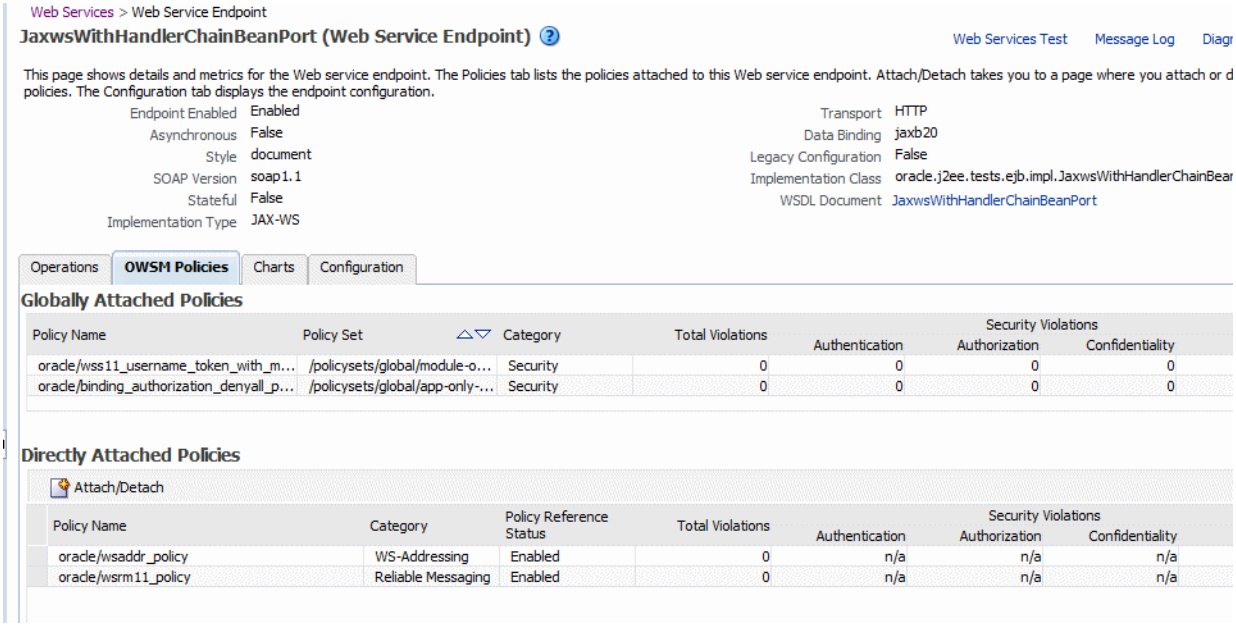
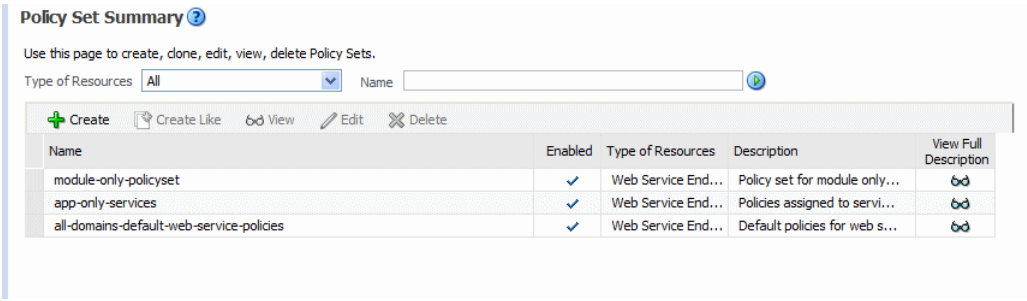


Figure 2–3 shows the Policy Set Summary page from which you can create policy sets to attach policies globally to a range of endpoints.

Figure 2–3 Creating Policy Sets Using Oracle Enterprise Manager



Complete details are provided in the following sections of *Security and Administrator's Guide for Web Services*:

- "Attaching a Policy to a Single Subject"
- "Attaching a Policy to Multiple Subjects (Bulk Attachment)"
- "Attaching Policies to Web Service Clients"
- "Attaching Policies Globally Using Policy Sets"
- "Creating and Managing Policy Sets"

2.7 Attaching Policies Using WebLogic Scripting Tool (WLST)

The Web services WLST policy management commands perform many of the same management functions that you can complete using Oracle Enterprise Manager Fusion Middleware Control.

After a Web service or client is deployed, you can attach policies directly to Oracle Infrastructure Web Service endpoints using WLST. You can also use WLST to create policy sets to attach policies globally to a range of endpoints.

Procedural information for using these commands is provided in the following sections of *Security and Administrator's Guide for Web Services*:

- "Attaching a Policy to a Web Service Using WLST"
- "Using WLST" in "Attaching Policies to Web Service Clients"
- "Attaching Policies Globally Using Policy Sets"
- "Creating and Managing Policy Sets"

Reference information for these commands is provided in "Web Services Custom WLST Commands" in *WebLogic Scripting Tool Command Reference*.

3

Securing Oracle Infrastructure Web Services

This chapter describes how to secure Oracle Infrastructure Web services.

3.1 Overview of Web Services Security

Web services security includes several aspects:

- **Authentication**—Verifying that the user is who she claims to be. A user's identity is verified based on the credentials presented by that user, such as:

  1. Something one has, for example, credentials issued by a trusted authority such as a passport (real world) or a smart card (IT world).
  2. Something one knows, for example, a shared secret such as a password.
  3. Something one is, for example, biometric information.

  Using a combination of several types of credentials is referred to as "strong" authentication, for example using an ATM card (something one has) with a PIN or password (something one knows).

- **Authorization (or Access Control)**—Granting access to specific resources based on an authenticated user's entitlements. Entitlements are defined by one or several attributes. An attribute is the property or characteristic of a user, for example, if "Marc" is the user, "conference speaker" is the attribute.

- **Confidentiality, privacy**—Keeping information secret. Accesses a message, for example a Web service request or an email, as well as the identity of the sending and receiving parties in a confidential manner. Confidentiality and privacy can be achieved by encrypting the content of a message and obfuscating the sending and receiving parties' identities.

- **Integrity, non repudiation**—Making sure that a message remains unaltered during transit by having the sender digitally sign the message. A digital signature is used to validate the signature and provides non-repudiation. The timestamp in the signature prevents anyone from replaying this message after the expiration.

For more information about these Web services security concepts, see "Understanding Web Services Security Concepts" in *Security and Administrator's Guide for Web Services*.

Oracle Web Services Manager (WSM) is designed to define and implement Web services security in heterogeneous environments, including authentication, authorization, message encryption and decryption, signature generation and validation, and identity propagation across multiple Web services used to complete a single transaction. In addition, Oracle WSM provides tools to manage Web services based on service-level agreements. For example, the user (a security architect or a systems administrator) can define the availability of a Web service, its response time, and other information that may be used for billing purposes. For more information about Oracle WSM, see "Understanding Oracle WSM Policy Framework" in *Security and Administrator's Guide for Web Services*.

3.2 Oracle WSM Predefined Security Policies and Assertion Templates

As described in Chapter 2, "Attaching Policies to Oracle Infrastructure Web Services,", Oracle WSM provides a set of predefined policies and assertion templates that are automatically available when you install Oracle Fusion Middleware.

The following categories of **security** policies and assertion templates are available out-of-the-box:

- Authentication Only Policies
- Message Protection Only Policies
- Message Protection and Authentication Policies
- Authorization Only Policies

For complete details about the predefined security policies and assertion template, see the following sections in *Security and Administrator's Guide for Web Services*:

- "Security Policies"
- "Security Assertion Templates"

For assistance in determining which security policies to use, see "Determining Which Security Policies to Use" in *Security and Administrator's Guide for Web Services*.

3.3 Attaching Security Policies

You can attach security policies to Oracle Infrastructure Web services and clients at design time using Oracle JDeveloper, or runtime using the Oracle Enterprise Manager. For more information, see Chapter 2, "Attaching Policies to Oracle Infrastructure Web Services."

3.4 Configuring Security Policies

You must configure the security policies before you can use them in your environment. The steps to configure security policies are described in "Configuring Policies" in *Security and Administrator's Guide for Web Services*.

The following table provides references to the configuration steps for each policy category.

Table 3–1 Configuring Security Policies

Policy Category	Configuration Steps in *Security and Administrator's Guide for Web Services*
Authentication Only Policies	"Authentication-Only Policies and Configuration Steps"
Message Protection Only Policies	"Message Protection-Only Policies and Configuration Steps"
Message Protection and Authentication Policies	"Message Protection and Authentication Policies and Configuration Steps"
Authorization Policies	"Authorization Policies"

4

Developing Asynchronous Web Services

The JAX-WS specification provides an asynchronous client API that enables you to call synchronous methods in an asynchronous way. This chapter introduces asynchronous Web service concepts and describes how to develop and configure asynchronous Web services.

4.1 Overview of Asynchronous Web Services

When you invoke a Web service synchronously, the invoking client application waits for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the Web service might be adequate. However, because request processing can be delayed, it is often useful for the client application to continue its work and handle the response later on. By calling a Web service asynchronously, the client can continue its processing, without interrupt, and will be notified when the asynchronous response is returned.

The following sections step through several asynchronous message flow diagrams, provide the perspective from the client-side, and explain how asynchronous messages are correlated:

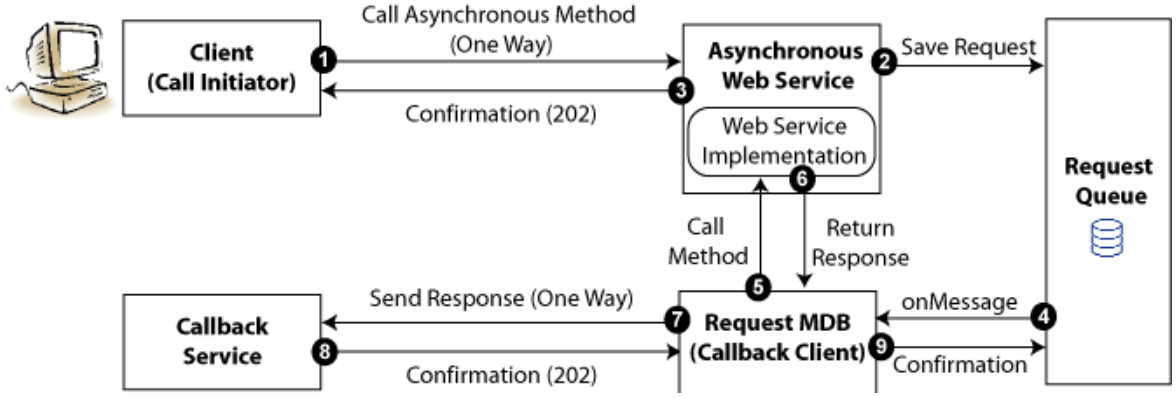
4.1.1 Asynchronous Web Service Using a Single Request Queue

> **Note:** Although the single request queue scenario may provide better performance, it is less reliable than the request and response queue scenario. Oracle recommends that you use the request and response queue scenario, described in Section 4.1.2, "Asynchronous Web Service Using a Request and a Response Queue", and not the single request queue scenario.

In this scenario, there is a single message-driven bean (MDB) associated with the request queue that handles both the request and response processing.

The following figure shows the flow of an asynchronous method call using a single request queue.

Figure 4–1 Asynchronous Web Service Using a Single Request Queue



The following describes the flow shown in the previous figure:

1. The client calls an asynchronous method.
2. The asynchronous Web services receives the request and stores it in the request queue.
3. The asynchronous Web service sends a receipt confirmation to the client.
4. The MDB listener on the request queue receives the message and initiates processing of the request.
5. The request MDB calls the required method in the Web service implementation.
6. The Web service implementation returns the response.
7. The request MDB, acting as a callback client, returns the response to the callback service.
8. The callback service returns a receipt confirmation message.
9. The request MDB returns a confirmation message to the request queue to terminate the process.

In this scenario, if there is a problem connecting to the callback service (in Step 7), then the response will not be sent. If the request is retried later, the flow resumes from Step 4 and the Web service implementation will be called again (in Step 5). This may not be desirable depending on your application logic or transactional control processing. In the next scenario, the response is stored in a separate response queue, eliminating the

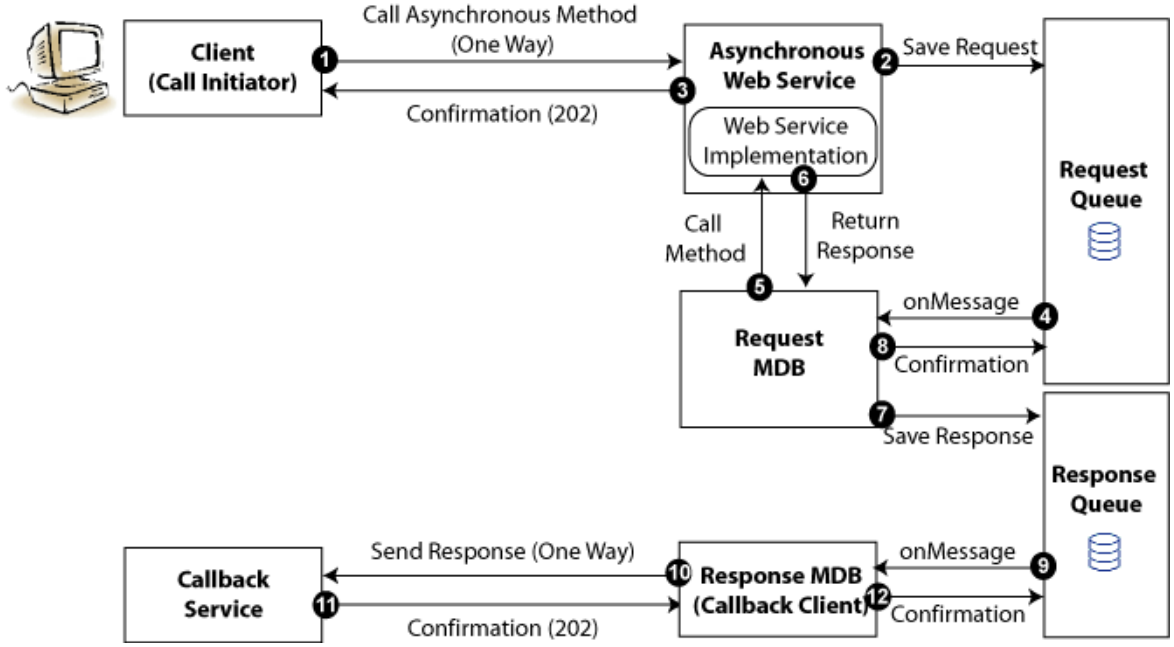
need to recall the Web service implementation in the event that the callback service is not available initially.

4.1.2 Asynchronous Web Service Using a Request and a Response Queue

In this scenario, there are two MDBs, one to handle the request processing and one to handle the response processing. By separating the execution of business logic from the response return, this scenario provides improved error recovery over the single queue model described in Section 4.1.1, "Asynchronous Web Service Using a Single Request Queue".

The following figure shows the flow of an asynchronous method call using a single request queue.

Figure 4–2 Asynchronous Web Service Using a Request and Response Queue



The following describes the flow shown in the previous figure:

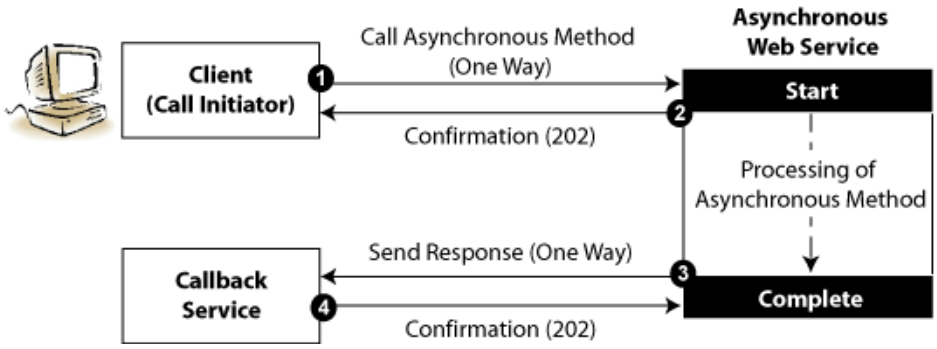
1. The client calls an asynchronous method.
2. The asynchronous Web services receives the request and stores it in the request queue.
3. The asynchronous Web service sends a receipt confirmation to the client.
4. The MDB listener on the request queue receives the message and initiates processing of the request.
5. The request MDB calls the required method in the Web service implementation.
6. The Web service implementation returns the response.
7. The request MDB saves the response to the response queue.
8. The request MDB sends a confirmation to the request queue to terminate the process.
9. The onMessage listener on the response queue initiates processing of the response.

10. The response MDB, acting as the callback client, returns the response to the callback service.
11. The callback service returns a receipt confirmation message.
12. The response MDB returns a confirmation message to the response queue to terminate the sequence.

4.1.3 Client Perspective of Asynchronous Web Service Call

From the client perspective, the asynchronous method call consists of two one-way message exchanges, as shown in the following figure.

Figure 4–3 Asynchronous Web Service Client Flow



As shown in the previous figure, before initiating the asynchronous call, the client must deploy a callback service to listen for the response from the asynchronous Web service.

The message flow is as follows:

1. The client calls an asynchronous method.
2. The asynchronous Web services receives the request, sends a confirmation message to the initiating client, and starts process the request.
3. Once processing of the request is complete, the asynchronous Web service acts as a client to send the response back to the callback service.
4. The callback service sends a confirmation message to the asynchronous Web service.

4.1.4 How Asynchronous Messages Are Correlated

Note: Message correlation is handled automatically by the SOA runtime. This section is for informational purposes only.

When the callback service receives a response, it needs a way to correlate the response back to the original request. This is achieved using WS-Addressing and is handled automatically by the SOA runtime.

The client sets the following two fields in the WS-Addressing part of the SOAP header:

- ReplyTo address—Address of the callback service.
- MessageID—Unique ID that identifies the request. For example, a UUID.

The callback client sends the MessageId corresponding to the initial request in the relatesToId field in the WS-Addressing header. If additional data is required by the callback service to process the response, clients can perform one of the following tasks:

- Clients can send the data as a reference parameter in the `ReplyTo` field. Asynchronous Web services return all reference parameters with the response, so the callback service will be able to access the information.
- If sending the data as part of the asynchronous request message is not practical, then the client can save the MessageID and data required to the local data store.

4.2 Using JDeveloper to Develop and Deploy Asynchronous Web Services

You can develop and deploy asynchronous Web service methods for an ADF Business Component quickly and easily. For complete details, see "How to Generate Asynchronous Web Service Methods" in *Developer's Guide for Oracle Application Development Framework*.

The following sections describe in more detail how an asynchronous Web service is implemented. In some cases, the information is provided for informational purposes only.

4.3 Developing an Asynchronous Web Service

A JAX-WS Web service can be declared an asynchronous Web service using the following annotation: oracle.webservices.annotations.async.AsyncWebService.

The following provides a very simple POJO example of an asynchronous Web service:

```
import oracle.webservices.annotations.PortableWebService
import oracle.webservices.annotations.async.AsyncWebService

@PortableWebService
@AsyncWebService
public class HelloService {
    public String hello(String name) {
        return "Hi " + name;
    }
}
```

The generated WSDL for the asynchronous Web service contains two one-way operations defined as two portTypes: one for the asynchronous operation and one for the callback operation.

For example:

```
<wsdl:portType name="HelloService">
    <wsdl:operation name="hello">
        <wsdl:input message="tns:helloInput"
               xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
               ns1:Action=""/>
    </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="HelloServiceResponse">
    <wsdl:operation name="helloResponse">
        <wsdl:input message="tns:helloOutput"
               xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
               ns1:Action=""/>
    </wsdl:operation>
```

```
</wsdl:portType>
```

Optionally, you can define a system user to secure access to the asynchronous Web service using the `systemUser` argument. If not specified, this value defaults to OracleSystemUser.

For example:

```
@AsyncWebService(systemUser="ABCIncSystemUser')
```

For more information about securing the request and response queues used by asynchronous Web services, Section 4.4.4, "Securing the Request and Response Queues".

4.4 Creating the Request and Response Queues

Before you deploy your asynchronous Web services, you must create and secure the queues used to store the request and response, as described in the following sections:

- Section 4.4.1, "Using the Default WebLogic JMS Queues"
- Section 4.4.2, "Creating Custom Request and Response Queues"
- Section 4.4.3, "Administering Request and Response Queues at Runtime"
- Section 4.4.4, "Securing the Request and Response Queues"
- Section 4.4.5, "Confirming the Request and Response Queue Configuration"

4.4.1 Using the Default WebLogic JMS Queues

The process varies based on whether you are using a clustered or non-clustered domain, as described in the following sections.

- Section 4.4.1.1, "Using the Default WebLogic JMS Queues in Clustered Domains"
- Section 4.4.1.2, "Using the Default WebLogic JMS Queues in Non-clustered Domains"
- Section 4.4.1.3, "Tuning the Default JMS Delivery Failure Parameters"

4.4.1.1 Using the Default WebLogic JMS Queues in Clustered Domains

For clustered domains, you must create two JMS Uniform Distributed Destinations (UDDs) per cluster. An offline WLST script is provided to assist you in adding the default queues to your clustered environment.

The script is available at the following location:

```
<MW_HOME>/oracle_common/webservices/bin/jrfws-async-createUDDs.py
```

The following provides an example of how you might execute this script:

```
java -classpath <some_path>/weblogic.jar weblogic.WLST ./jrfws-async-createUDDs.py
--domain_home <domain> --cluster <cluster>
```

4.4.1.2 Using the Default WebLogic JMS Queues in Non-clustered Domains

For non-clustered domains, a pair of default WebLogic JMS queues are provided as part of the following WebLogic domain extension template: oracle.jrf.ws.async_template_11.1.1.jar. The default JMS queues included in the extension template include:

- Request queue: oracle.j2ee.ws.server.async.DefaultRequestQueue
- Response queue: oracle.j2ee.ws.server.async.DefaultResponseQueue

The default JMS connection factory, weblogic.jms.XAConnectionFactory, provided as part of the base domain, is used by default.

To create the required default queues, when creating or extending your domain using the Fusion Middleware Configuration Wizard, select **Oracle JRF Web Services Asynchronous services**. For more information, see *Creating Domains Using the Configuration Wizard*.

Note: When using this domain extension template, ensure that you explicitly target the JMS module (JRFWSAsyncJmsModule) to non-clustered one or more servers in your domain, as required.

4.4.1.3 Tuning the Default JMS Delivery Failure Parameters

The following default values are configured by default for the JMS delivery failure parameters. It is recommended that you review the settings and modify them, as appropriate, for your environment. Configuration related to JMS delivery failure parameters can be modified using the WebLogic Server Administration Console, as described in *Configuring and Managing JMS for Oracle WebLogic Server*.

- Set the **Redelivery Delay Override** value to **900000** milliseconds. That is, wait 15 minutes before rolled back or recovered messages are redelivered, regardless of the redelivery delay specified by the message consumer or the connection factory.
- Set the **Redelivery Limit** value to **100**. This value specifies the number of redelivery attempts allowed before a message is moved to the Error Destination defined for the queues.
- Set the **Expiration Policy** value to **Redirect**. This moves expired messages from their current location to the Error Destination defined for the queues. If enabled, verify that the corresponding Error Destination has been configured.

4.4.2 Creating Custom Request and Response Queues

If the default WebLogic JMS queues do not meet your requirements, you can perform the following steps:

1. Create the request and response queues manually, as described in *Programming JMS for Oracle WebLogic Server*.
2. Add them to your application code using the following annotations:
   - **Request queue**: oracle.webservices.annotations.async.AsyncWebServiceQueue

     For more information, see Section A.4, "@AsyncWebServiceQueue Annotation".
   - **Response queue**: oracle.webservices.annotations.async.ResponseQueue

     For more information, see Section A.5, "@AsyncWebServiceResponseQueue Annotation".

The following provides a list of **best practices** for creating the custom request and response queues (in Step 1):

- Configure queues in a cluster for high availability and failover.

- Configure a single JMS Server and WebLogic persistence store for each WebLogic Server and target them to the server's default migratable target.
- For the JMS Server, configure quotas, as required.

  To prevent out-of-memory errors, it is recommended that you configure the maximum messages quota for each JMS Server. As a guide, each message header consumes approximately 512 bytes. Therefore, a maximum quota of 500,000 message will require approximately 250MB of memory. Consider the memory resources available in your environment and set this quota accordingly.
- Configure a single JMS system module and target it to a cluster.
- For the JMS system module, configure the following:
  - Single subdeployment and populate it with each JMS Server.
  - Required uniform distributed destination(s) and define targets using advanced subdeployment targeting (defined above).
  - Custom connection factory. If transactions are enabled, the connection factory must support transactions, if enabled. By default, WebLogic JMS provides the `weblogic.jms.XAConnectionFactory` connection factory to support transactions.

4.4.3 Administering Request and Response Queues at Runtime

Using Oracle Enterprise Manager, you can modify the request and response queues at runtime. You will need to stop and restart the server in order for the updates to take effect. For complete details, see "Configuring Asynchronous Web Services" in *Security and Administrator's Guide for Web Services*.

4.4.4 Securing the Request and Response Queues

> **Note:** This section applies to ADF Web services only. It does not apply to SOA Web services.

It is recommended that you secure the JMS request and response queues with a user- or role-based security policy to secure access to these resources. The steps to secure the JMS request and response queues include:

1. Optionally, configure the JMS System User, as described in Section 4.4.4.1, "Configuring a JMS System User (Optional)".

   By default, the JMS System User that is authorized to access the JMS queues is set as OracleSystemUser. In most cases, the default user is sufficient.
2. Run the WLST script to secure the request and response queues, as described in Section 4.4.4.2, "Running the WLST Script to Secure the Request and Response Queues".

4.4.4.1 Configuring a JMS System User (Optional)

By default, the JMS System User that is authorized to access the JMS queues is set as OracleSystemUser. In most cases, this default value is sufficient. However, if you need to change this value to a custom user in your security realm, you can specify a custom system user using the systemUser attribute of the @AsyncWebService annotation. For example:

```
@AsyncWebService(systemUser = "ABCIncSystemUser")
```

In order for this change to take effect, you need to regenerate the application EAR file using JDeveloper or the `ojdeploy` command line utility. For more information about that @AsyncWebService annotation, see Section A.3, "@AsyncWebService Annotation".

After your application has been deployed, you can change the JMS System User in Oracle Enterprise Manager Fusion Middleware Control and in the WebLogic Server Administration Console as described in "Changing the JMS System User for Asynchronous Web Services" in *Security and Administrator's Guide for Web Services*.

4.4.4.2 Running the WLST Script to Secure the Request and Response Queues

An online WLST script is provided to assist you in securing the request and response queues. You pass the JMS system module name that you want to secure and the security role to be assigned, in addition to the Administration Server connection details (URL, username, and password).

The script is available at the following location:

```
<MW_HOME>/oracle_common/webservices/bin/secure_jms_system_resource.py
```

The following provides an example of how you might execute this script:

```
java -classpath <some_path>/weblogic.jar weblogic.WLST ./secure_jms_system_
resource.py
 --username <AdminUserName> --password <AdminPassword> --url <AdminServer_t3_url>
 --jmsSystemResource <JMSSystemResourceName> --role <SecurityRoleToUse>
```

4.4.5 Confirming the Request and Response Queue Configuration

To confirm that the request and response queues have been configure as required, perform one of the following tasks:

1. Invoke the Administration Console, as described in "Invoking the Administration Console" in *Getting Started With JAX-WS Web Services for Oracle WebLogic Server*.
2. Verify that the following JMS resources are defined and available under **Services > Messaging**:
   - JMS server named `JRFWSAsyncJmsServer`.

     Ensure that the JMS module is targeted to one or more servers, as required, for a non-clustered domain (standard queues) or to the cluster for clustered domains (UDDs). The JMS Server is targeted appropriately when configuring the domain using the Configuration Wizard or WLST. See also Section 4.4.1, "Using the Default WebLogic JMS Queues."
   - JMS module named `JRFWSAsyncJmsModule`.
   - Request queue with JDNI name `oracle.j2ee.ws.server.async.DefaultRequestQueue` and a corresponding error queue.
   - Response queue with JDNI name `oracle.j2ee.ws.server.async.DefaultResponseQueue` and a corresponding error queue.
3. Ensure that the asynchronous Web service is deployed and verify that the system message-driven beans (MDBs) are connected to the JMS destination(s), as required, as follows:

1. Click **Deployments**.
2. Expand the application in the Deployments table.
3. Under EJBs, verify that there are two system MDBs per Web service to support the asynchronous Web service runtime. For example, `<asyncwebservicename>_AsyncRequestProcessorMDB` and `<asyncwebservicename>_AsyncResponseProcessorMDB`.
4. Verify that each MDB displayed is connected to the JMS destination.
5. Select the EJB and click the **Monitoring** tab and then the **Running** tab.
6. Verify that the Connection Status field is Connected.
7. If the queues were not created correctly, perform one of the following steps:
   - Create and configure the required JMS queues manually. For more information, see "Messaging" in *Information Roadmap for Oracle WebLogic Server* .
   - Remove the JMS server, JMS module, and the default queues that were created for asynchronous Web service and recreate them using the steps provided for clustered and non-clustered domains, as described in Section 4.4.1, "Using the Default WebLogic JMS Queues."

Note: JMS resources are stored in the `config.xml` file for the domain.

4.5 Configuring the Callback Service

You can use the annotations defined in the following table to customize characteristics for the callback service (portType). The annotations are included in the `oracle.webservices.annotations.async` package.

Table 4–1 Annotations Used to Configure the Callback Service‘

Annotation	Description
`@CallbackMethod`	Customize the names of the WSDL entities for the corresponding operation in the callback portType and set whether a method is synchronous or asynchronous.
`@CallbackProperties`	Specify a set of properties that are required in the message context when calling the callback service. For more information, see Section A.10, "@CallbackProperties Annotation".
`@ResponseWebService`	Customize the response Web service port information. For more information, see Section A.17, "@ResponseWebService Annotation".
`@Retry`	Specifies whether the asynchronous method is idempotent, or retriable, in the event that its execution is terminated abnormally (for example, due to system failure). For more information, see Section A.18, "@Retry Annotation".

4.6 Configuring SSL for Asynchronous Web Services

To configure SSL for asynchronous Web services, you must perform the following tasks:

1. Create the identity and trust keystores using the `keytool`.

   See "Obtaining Private Keys, Digital Certificates, and Trusted Certificate Authorities" in *Securing Oracle WebLogic Server*.

2. Configure the keystore, keystore type, and password for the identity and trust keystores using Oracle WebLogic Server Administration Console.

   See "Configure Keystores" in *Oracle WebLogic Server Administration Console Help*.

3. Configure the password for the identity and trust keystores in the Oracle WSM Credential store provider using Fusion Middleware Control Enterprise Manager.

   See "Configuring the Credential Store Provider" in *Security and Administrator's Guide for Web Services*.

   Ensure that the map with the name `oracle.ws.async.ssl.security` exists. If it does not exist, you must create it, as described in "Configuring the Credential Store Provider" in *Security and Administrator's Guide for Web Services*.

   Then, create the three key entries defined in the following table.

Table 4–2 Configure Identity and Trust Keystores

Key	User Name	Password
trust-keystore-password	async	`<trust_store_password>`
identity-keystore-password	async	`<identity_store_password>`
key-password	async	`<key_password>`

4.7 Defining Asynchronous Web Service Clients

The following two types of clients can call asynchronous Web services:

- SOA/BPEL clients—The process is identical to that of calling other asynchronous BPEL clients. For more information, see "Invoking an Asynchronous Web Service form a BPEL Process" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
- WebLogic Java EE JAX-WS Clients—Using the Create Web Service Proxy wizard in JDeveloper, select the **Generate As Async** option to generate an asynchronous proxy. For more information about creating Web service clients using the wizard, see "Creating Web Service Proxies" in the *JDeveloper Online Help*.

The following sections step through an example an asynchronous client and callback service that is generated and describe the updates that need to be implemented.

- Section 4.7.1, "Updating the Asynchronous Client Code"
- Section 4.7.2, "Updating the Callback Service Code"

> **Note:** The steps described in the following sections are applicable for WebLogic Java EE JAX-WS clients, and not SOA/BPEL clients.

4.7.1 Updating the Asynchronous Client Code

The following provides a sample of the client code that is generated to support asynchronous Web services. As shown in bold, the client code must set two fields in the outbound header to correlate the asynchronous messages:

- ReplyTo address—Address of the callback service.
- MessageID—Unique ID that identifies the request. For example, a UUID.

The client code that is generated provides a UUID for the message ID that is sufficient in most cases, though you may wish to update it. You need to update the ReplyTo address to point to the callback service.

Example 4–1 Updating the Asynchronous Client Code

```
package async.jrf;

import com.sun.xml.ws.api.addressing.AddressingVersion;
import com.sun.xml.ws.api.addressing.WSEndpointReference;
import com.sun.xml.ws.developer.WSBindingProvider;
import com.sun.xml.ws.message.StringHeader;
import java.util.UUID;
import javax.xml.ws.WebServiceRef;

// !THE CHANGES MADE TO THIS FILE WILL BE DESTROYED IF REGENERATED!
// This source file is generated by Oracle tools
// Contents may be subject to change

// For reporting problems, use the following
// Version = Oracle WebServices (11.1.1.0.0, build 090303.0200.48673)

public class HelloServicePortClient
{
  @WebServiceRef
  private static HelloServiceService helloServiceService;
  private static final AddressingVersion WS_ADDR_VER = AddressingVersion.W3C;

  public static void main(String [] args)
  {
    helloServiceService = new HelloServiceService();
    HelloService helloService = helloServiceService.getHelloServicePort();
    // Get the request context to set the outgoing addressing properties
    WSBindingProvider wsbp = (WSBindingProvider)helloService;
    WSEndpointReference repl
         new WSEndpointReference(
             "http://<replace with the URL of the callback service>",
               WS_ADDR_VER);
    String uuid = "uuid:" + UUID.randomUUID();
   wsbp.setOutboundHeaders(
               new StringHeader(WS_ADDR_VER.messageIDTag, uuid),
               replyTo.createHeader(WS_ADDR_VER.replyToTag));
    // Add your code to call the desired methods.
  }
}
```

4.7.2 Updating the Callback Service Code

The following provides a sample of the callback service code. The code shown in bold illustrates how to extract the relatesToID from the message header, which is sent by the client as the MessageID.

You need to implement the code to process the response and deploy the callback service as a Web service. Once deployed, add the URL of the callback service to the client code as the replyTo field.

```
package async.jrf;

import com.sun.xml.ws.api.addressing.AddressingVersion;
```

```
import com.sun.xml.ws.api.message.Header;
import com.sun.xml.ws.api.message.HeaderList;
import com.sun.xml.ws.developer.JAXWSProperties;
import javax.annotation.Resource;
import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.Action;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.soap.Addressing;
// !THE CHANGES MADE TO THIS FILE WILL BE DESTROYED IF REGENERATED!
// This source file is generated by Oracle tools
// Contents may be subject to change
// For reporting problems, use the following
// Version = Oracle WebServices (11.1.1.0.0, build 090303.0200.48673)

@WebService(targetNamespace="http://jrf.async/", name="HelloServiceResponse")
@XmlSeeAlso(
  { async.jrf.ObjectFactory.class })
@SOAPBinding(style=Style.DOCUMENT)
@Addressing(enabled=true, required=true)
public class HelloServiceResponseImpl
{
  @Resource
  private WebServiceContext wsContext;
  private static final AddressingVersion WS_ADDR_VER = AddressingVersion.W3C;
  @WebMethod
  @Action(input="")
  @RequestWrapper(localName="helloResponse",targetNamespace="http://jrf.async/",
    className="async.jrf.HelloResponse")
  @Oneway
  public void helloResponse(@WebParam(targetNamespace="", name="return")
    String _return)
  {
    // Use the sample code to extract the relatesTo id for correlation and then
add your rest of the logic
    System.out.println("Received the asynchronous reply");
    // get the messageId to correlate this reply with the original request
    HeaderList headerList =
(HeaderList)wsContext.getMessageContext().get(JAXWSProperties.INBOUND_HEADER_LIST_
PROPERTY);
    Header realtesToheader = headerList.get(WS_ADDR_VER.relatesToTag, true);
    String relatesToMessageId = realtesToheader.getStringContent();
    System.out.println("RelatesTo message id: " + relatesToMessageId);
    // Add your implementation here.
  }
}
```

4.8 Attaching Policies to Asynchronous Web Services and Clients

You can attach policies to the following asynchronous components:

- Client calling asynchronous Web service—See "Attaching Policies to Asynchronous Web Service Clients" on page 4-14.

- Asynchronous Web service—See "Attaching Policies to Asynchronous Web Services and Callback Services" on page 4-14.
- Asynchronous callback client—See "Attaching Policies to Callback Clients" on page 4-15.
- Asynchronous callback service—See "Attaching Policies to Asynchronous Web Services and Callback Services" on page 4-14.

The asynchronous Web service and client policies must comply with one another. Similarly, the asynchronous callback client and callback service policies must comply with one another.

You can use one of the following methods to attach policies to the components:

- Using annotations at design time.
- Using Enterprise Manager at runtime.

Each method is described in detail in the following sections.

Note: You do not need to attach the `oracle/wsaddr_policy` again to your asynchronous Web service or client. By default, the `oracle/wsaddr_policy` policy is attached to all asynchronous Web services and clients and advertised in the WSDL, as it is required by asynchronous Web service processing. However, Enterprise Manager Fusion Middleware Control does not reflect that the policy is attached and it is available for selection in the Available Policies list when attaching polices, as described in "Attaching Policies to Web Services" in *Security and Administrator's Guide for Web Services.*

4.8.1 Attaching Policies to Asynchronous Web Service Clients

At design time, you can use the following methods to attach policies:

- For SOA/BPEL clients, you can use the SOA Composite Editor to attach policies, as described in "Managing Policies" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
- For WebLogic Java EE JAX-WS Clients, you can use the Create Web Service Proxy wizard in JDeveloper to attach policies. For more information about creating Web service clients using the wizard, see "Creating Web Service Proxies" in the *JDeveloper Online Help*.

At runtime, you can use the Enterprise Manager to attach policies to each type of client, as described in "Attaching Policies to Web Service Clients" in *Security and Administrator's Guide for Web Services*.

4.8.2 Attaching Policies to Asynchronous Web Services and Callback Services

At design time, to attach policies to an asynchronous Web services and callback services, you can use one of the annotations defined in Table 4–3. The annotations are included in the oracle.webservices.annotations package.

Table 4–3 Annotations for Attaching Policies to Web Services

Annotation	Description
@AddressingPolicy	Attaches a WS-Addressing policy to the Web service. For more information, see Section A.2, "@AddressingPolicy Annotation".
@ManagementPolicy	Attaches a management policy to the Web service. For more information, see Section A.12, "@ManagementPolicy Annotation".
@MtomPolicy	Attaches an MTOM policy to the Web service. For more information, see Section A.13, "@MtomPolicy Annotation".
@SecurityPolicies	Specifies an array of @SecurityPolicy annotations. Use this annotation if you want to attach more than one WS-Policy files to a class. For more information, see Section A.19, "@SecurityPolicies Annotation".
@SecurityPolicy	Attaches a security policy to the Web service. For more information, see Section A.20, "@SecurityPolicy Annotation".

At runtime, you can use the Enterprise Manager to attach policies to asynchronous Web services and callback services, as described in "Attaching Policies to Web Services" in *Security and Administrator's Guide for Web Services*.

Use the following guidelines when attaching message protection policies to asynchronous Web services and callback services:

- If you want to enforce a message protection policy on the callback service, you must also enforce a message protection policy on the asynchronous request. Otherwise, an error message will be returned at runtime indicating that the asynchronous client public encryption certificate is not found.

  You can enforce a message protection policy on the asynchronous request without enforcing that same policy on the callback service.

- If you enforce a message protection policy on the asynchronous Web service, then you must configure the client public encryption certificate in the client keystore.

4.8.3 Attaching Policies to Callback Clients

Note: The policies that you attach to the callback client are advertised in the asynchronous Web service WSDL.

At design time, to attach policies to an asynchronous callback client, you can use one of the annotations defined in Table 4–4. The annotations are included in the oracle.webservices.annotations.async package.

Table 4–4 Annotations for Attaching Policies to Callback Clients

Annotation	Description
@CallbackManagementPolicy	Attaches a Management policy to the callback client of the asynchronous Web service that will connect to the callback service. By default, no Management policy is attached. For more information, see Section A.7, "@CallbackManagementPolicy Annotation".
@CallbackMtomPolicy	Attaches an MTOM policy to the callback client of the asynchronous Web service that will connect to the callback service. By default, no MTOM policy is attached. For more information, see Section A.9, "@CallbackMtomPolicy Annotation".
@CallbackSecurityPolicy	Attaches one or more security policies to the callback client of the asynchronous Web service that will connect to the callback service. By default, no security policies are attached. For more information, see Section A.11, "@CallbackSecurityPolicy Annotation".

At runtime, you can use the Enterprise Manager to attach policies to asynchronous callback clients, as described in"Attaching Policies to Asynchronous Web Service Callback Clients" in *Security and Administrator's Guide for Web Services*.

5

Using Web Services Reliable Messaging

This section describes how to use Web Services Reliable Messaging (WS-ReliableMessaging) to exchange message reliably.

5.1 Overview of Web Services Reliable Messaging

Message exchanges between Web services can be disrupted by various errors, including network, system, and software component errors or anomalies. Once a Web service or client sends a message, there is no immediate way, except by consulting network-level exceptions or SOAP fault messages, to determine whether the message was delivered successfully, if delivery failed and requires a retransmission, or if a sequence of messages arrived in the correct order.

To resolve these issues, Oracle Infrastructure Web services support the messaging protocol defined by the Web Services Reliable Messaging (WS-ReliableMessaging) specification at `http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf`. This specification describes a protocol that makes message exchanges reliable. *Reliable* is defined as the ability to guarantee message delivery between the two Web Services. It ensures that messages are delivered reliably between distributed applications regardless of software component, system, or network failures. Ordered delivery is assured and automatic retransmission of failed messages does not have to be coded by each client application.

A reliable Web service provides the following delivery assurances.

Table 5–1 Delivery Assurances for Reliable Messaging

Delivery Assurance	Description
At Most Once	Messages are delivered at most once, without duplication. It is possible that some messages may not be delivered at all.
At Least Once	Every message is delivered at least once. It is possible that some messages are delivered more than once.
Exactly Once	Every message is delivered exactly once, without duplication.
In Order	Messages are delivered in the order that they were sent. This delivery assurance can be combined with one of the preceding three assurances.

Consider using reliable messaging if your Web service is experiencing the following problems:

- Network failures or dropped connections.
- Messages are lost in transit.
- Messages are arriving at their destination out of order.

5.2 Predefined Reliable Messaging Policies

Reliable messaging is driven by policies and the Policy framework. As described in Chapter 2, "Attaching Policies to Oracle Infrastructure Web Services,", Oracle WSM provides a set of predefined policies that are automatically available when you install Oracle Fusion Middleware.

The reliable messaging policies listed in Table 5–2 are available out-of-the-box.

Table 5–2 Predefined Reliable Messaging Policies

Reliable Messaging Policy	Description
oracle/wsrm11_policy	Provides support for version 1.1 of the Web Services Reliable Messaging protocol.
oracle/wsrm10_policy	Provides support for version 1.0 of the Web Services Reliable Messaging protocol.

For more information about the reliable messaging predefined policies, see "Reliable Messaging Policies" in *Security and Administrator's Guide for Web Services*.

5.3 Attaching Reliable Messaging Policies

You can attach reliable messaging policies to Oracle Infrastructure Web services or clients at design time using Oracle JDeveloper, or at runtime using the Oracle Enterprise Manager. For more information, see Chapter 2, "Attaching Policies to Oracle Infrastructure Web Services."

5.4 Configuring Reliable Messaging Policies

You must configure the reliable messaging policies before you can use them in your environment. You need to configure the following:

- Delivery assurance (see Table 5–1)
- Message store information, such as type and name
- Retransmission interval—Interval, in milliseconds, that the source endpoint waits after transmitting a message and before it retransmits the message if it receives no acknowledgment for that message.
- Inactivity timeout—Amount of time, in milliseconds, that can elapse between message exchanges associated with a particular WS-RM sequence before the sequence is terminated.

The steps to configure reliable messaging policies are described in "Reliable Messaging Policies and Configuration Steps" in *Security and Administrator's Guide for Web Services*.

6 Using Web Services Atomic Transactions

This section describes how to use Web services atomic transactions to enable interoperability with other external transaction processing systems. This section applies to SOA Web services and references only.

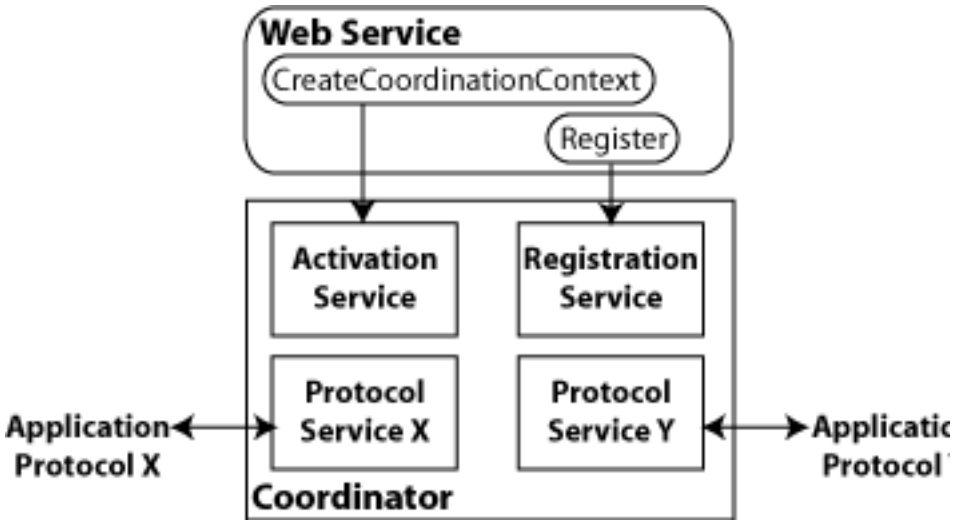
6.1 Overview of Web Services Atomic Transactions

WebLogic Web services enable interoperability with other external transaction processing systems, such as Websphere, JBoss, Microsoft .NET, and so on, through the support of the following specifications:

- WS-AtomicTransaction Version (WS-AT) 1.0, 1.1, and 1.2: `http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-cs-01/wstx-wsat-1.2-spec-cs-01.html`
- WS-Coordination Version 1.0, 1.1, and 1.2: `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-cs-01/wstx-wscoor-1.2-spec-cs-01.html`

These specifications define an extensible framework for coordinating distributed activities among a set of participants. The **coordinator**, shown in the following figure, is the central component, managing the transactional state (coordination context) and enabling Web services and clients to register as participants.

Figure 6–1 Web Services Atomic Transactions Framework



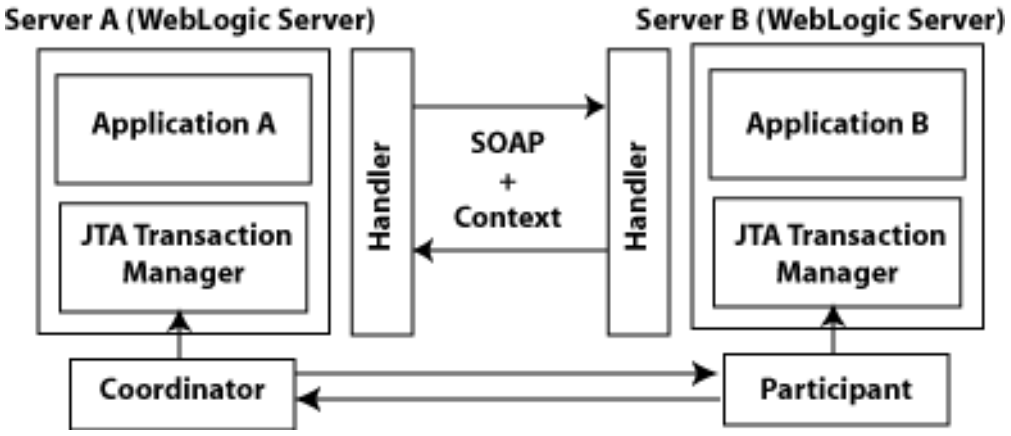
The following table describes the components of Web services atomic transactions, shown in the previous figure.

Table 6–1 Components of Web Services Atomic Transactions

Component	Description
Coordinator	Manages the transactional state (coordination context) and enables Web services and clients to register as participants.
Activation Service	Enables the application to activate a transaction and create a coordination context for an activity. Once created, the coordination context is passed with the transaction flow.
Registration Service	Enables an application to register as a participant.
Application Protocol X, Y	Supported coordination protocols, such as WS-AtomicTransaction.

The following figure shows two instances of WebLogic Server interacting within the context of a Web services atomic transaction. For simplicity, two WebLogic Web service applications are shown.

Figure 6–2 Web Services Atomic Transactions in WebLogic Server Environment



Please note the following:

- Using the local JTA transaction manager, a transaction can be imported to or exported from the local JTA environment as a *subordinate transaction*, all within the context of a Web service request.
- Creation and management of the coordination context is handled by the local JTA transaction manager.
- All transaction integrity management and recovery processing is done by the local JTA transaction manager.

For more information about JTA, see *Programming JTA for Oracle WebLogic Server*.

The following describes a sample end-to-end Web services atomic transaction interaction, illustrated in Figure 6–2:

1. Application A begins a transaction on the current thread of control using the JTA transaction manager on Server A.
2. Application A calls a Web service method in Application B on Server B.
3. Server A updates its transaction information and creates a SOAP header that contains the coordination context, and identifies the transaction and local coordinator.
4. Server B receives the request for Application B, detects that the header contains a transaction coordination context and determines whether it has already registered as a participant in this transaction. If it has, that transaction is resumed and if not, a new transaction is started.

   Application B executes within the context of the imported transaction. All transactional resources with which the application interacts are enlisted with this imported transaction.
5. Server B enlists itself as a participant in the WS-AT transaction by registering with the registration service indicated in the transaction coordination context.
6. Server A resumes the transaction.
7. Application A resumes processing and commits the transaction.

6.2 Enabling Web Services Atomic Transactions on an Oracle SOA Suite Web Service (Inbound)

On an Oracle SOA Suite Web service, you enable and configure Web services atomic transactions at design time using Oracle JDeveloper when creating a Web service, or at deployment time using the Oracle Enterprise Manager. For more information, refer to the following sections:

- **Design time**: "WS-Atomic Transaction Support" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
- **Deployment time**: "Configuring Web Services Atomic Transactions" in *Security and Administrator's Guide for Web Services*.

For information about configuration options, see Section 6.4, "Configuring Web Services Atomic Transactions".

6.3 Enabling Web Services Atomic Transactions on an Oracle SOA Suite Reference (Outbound)

On an Oracle SOA Suite reference, you enable and configure Web services atomic transactions at deployment time using the Oracle Enterprise Manager. You configure the version and flow type, as defined in Table 6–3. For more information, see "Configuring Web Services Atomic Transactions" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

You enable and configure Web services atomic transactions at design time using Oracle JDeveloper when creating a Web service, or at deployment time using the Oracle Enterprise Manager. For more information, refer to the following sections:

- **Design time**: "WS-Atomic Transaction Support" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
- **Deployment time**: "Configuring the Web Service Client" in *Security and Administrator's Guide for Web Services*.

For information about configuration options, see Section 6.4, "Configuring Web Services Atomic Transactions".

6.4 Configuring Web Services Atomic Transactions

The following tables summarizes the configuration options that you can set when enabling Web services atomic transactions on a SOA Web service. For information about enabling Web service atomic transactions, see Section 6.2, "Enabling Web Services Atomic Transactions on an Oracle SOA Suite Web Service (Inbound)" and Section 6.3, "Enabling Web Services Atomic Transactions on an Oracle SOA Suite Reference (Outbound)".

Table 6–2 Web Services Atomic Transactions Configuration Options

Attribute	Description
Version	Version of the Web services atomic transaction coordination context that is supported for the SOA service or reference. For SOA references, it specifies the version used for outbound messages only. The value specified must be consistent across the entire transaction. Valid values include `WSAT10`, `WSAT11`, `WSAT12`, and `DEFAULT`. The `DEFAULT` value for SOA services is all three versions (driven by the inbound request). The `DEFAULT` value for SOA references is as follows: ▪ If the flow option is `WSDLDRIVEN`, then the version advertised in the WSDL is used. ▪ If the flow option is any setting other than `WSDLDRIVEN`, then `WSAT10` is used.
Flow type	Whether the Web services atomic transaction coordination context is passed with the transaction flow. For valid values, see Table 6–3.

The following table summarizes the valid values for flow type and their meaning on the Web service and client. The table also summarizes the valid value combinations when configuring web services atomic transactions for an EJB-style web service that uses the `@TransacationAttribute` annotation.

Table 6–3 Flow Ttransaction coordination contextypes Values

Value	Web Service Client	Web Service	Valid EJB `@TransactionAttribute` Values
`NEVER` (Default for SOA service)	**JTA transaction**: Do not export transaction coordination context. **No JTA transaction**: Do not export transaction coordination context.	**Transaction flow exists**: Do not import transaction coordination context. If the CoordinationContext header contains `mustunderstand="true"`, a SOAP fault is thrown. **No transaction flow**: Do not import transaction coordination context.	`NEVER`, `NOT_SUPPORTED`, `REQUIRED`, `REQUIRES_NEW`, `SUPPORTS`
`SUPPORTS`	**JTA transaction**: Export transaction coordination context. **No JTA transaction**: Do not export transaction coordination context.	**Transaction flow exists**: Import transaction context. **No transaction flow**: Do not import transaction coordination context.	`SUPPORTS`, `REQUIRED`
`MANDATORY`	**JTA transaction**: Export transaction coordination context. **No JTA transaction**: An exception is thrown.	**Transaction flow exists**: Import transaction context. **No transaction flow**: Service-side exception is thrown.	`MANDATORY`, `REQUIRED`, `SUPPORTS`
`WSDLDRIVEN` (SOA references only, and the default)	Behaves according to the value that is advertised in the Web service WSDL.	N/A	Depends on advertised value.

6.5 Securing the Messages Exchanged Between the Coordinator and Participant

To secure messages exchanged between the coordinator and participant, you can configure the properties defined in the following table using the WebLogic Server Administration Console. These properties are configured at the domain level. For detailed steps, see "Configure Web services atomic transactions" in the *Oracle WebLogic Server Administration Console Help*.

Table 6–4 Securing Web Services Atomic Transactions

Property	Description
Web Services Transactions Transport Security Mode	Specifies whether two-way SSL is used for the message exchange between the coordinator and participant. This property can be set to one of the following values: ■ SSL Not Required—All Web service transaction protocol messages are exchanged over the HTTP channel. ■ SSL Required—All Web service transaction protocol messages are exchanged over the HTTPS channel. This flag must be enabled when invoking Microsoft .NET Web services that have atomic transactions enabled. ■ Client Certificate Required—All Web service transaction protocol messages are exchanged over HTTPS and a client certificate is required. For more information, see "Configure two-way SSL" in the *Oracle WebLogic Server Administration Console Help*.
Web Service Transactions Issued Token Enabled	Flag the specifies whether to use `IssuedToken` to enable authentication between the Web services atomic transaction coordinator and participant. The `IssuedToken` is issued by the coordinator and consists of a security context token (SCT) and a session key used for signing. The participant sends the signature, signed using the shared session key, in its registration message. The coordinator authenticates the participant by verifying the signature using the session key.

7

Using MTOM Encoded Message Attachments

This chapter describes how Oracle Infrastructure Web services process messages that are encoded in Message Transmission Optimization Mechanism (MTOM) format.

7.1 Overview of Message Transmission Optimization Mechanism

Binary content, such as an image in JPEG format, can be passed between the client and the Web service. In order to be passed, the binary content is typically inserted into an XML document as an `xsd:base64Binary` string. Transmitting the binary content in this format greatly increases the size of the message sent over the wire and is expensive in terms of the required processing space and time.

Using MTOM, binary content can be sent as a MIME attachment, which reduces the transmission size on the wire. The binary content is semantically part of the XML document. This is an advantage over SWA (SOAP Messages with Attachments), in that it enables you to apply operations such as WS-Security signature on the message. For more information, refer to the following specifications:

- SOAP 1.2: `http://www.w3.org/TR/2005/REC-soap12-mtom-20050125`
- SOAP 1.1: `http://www.w3.org/Submission/2006/SUBM-soap11mtom10-20060405`

Using MTOM to pass binary content as an attachment improves the performance of the Web services stack. Performance is not affected if an MTOM-encoded message does not contain binary content.

MTOM provides an optimized transmission mechanism for SOAP 1.2 messages with an envelope that contains elements of XML schema type `xs:base64` binary. MTOM makes use of data handling mechanisms described in the following specifications:

- XOP (XML-binary Optimized Packaging)—Provides a mechanism to more efficiently serialize XML Infosets that have content of type `xs:base64`Binary. The XOP specification is available at the following Web site: `http://www.w3.org/TR/xop10/`
- DMCBDX (Describing Media Content of Binary Data in XML)—Provides content type information for the binary data in the XML instance and schema. This

information can be used to optimize the processing of binary data. The DMCBDX specification is available at the following Web site:
http://www.w3.org/TR/2005/NOTE-xml-media-types-20050504/

- RRSHB (Resource Representation SOAP Header Block)—allows a SOAP message to carry a representation of a Web resource, such as a JPEG image, in a SOAP header block. When combined with MTOM and XOP, the Web resource contained in a RRSHB SOAP header block can be transmitted as a raw binary MIME attachment instead of an xs:base64Binary string in the SOAP header. The RRSHB specification is available at the following Web site:
http://www.w3.org/TR/2005/REC-soap12-rep-20050125/

These specifications fulfill the requirements outlined in OSUCR (SOAP Optimized Serialization Use Cases and Requirements), as described at:
http://www.w3.org/TR/2004/WD-soap12-os-ucr-20040608/

7.2 Predefined MTOM Attachment Policies

As described in Chapter 2, "Attaching Policies to Oracle Infrastructure Web Services,", Oracle WSM provides a set of predefined policies and assertion templates that are automatically available when you install Oracle Fusion Middleware.

The MTOM attachment policies listed in Table 7–1 are available out-of-the-box.

Table 7–1 Predefined MTOM Attachment Policies

Reliable Messaging Policy	Description
oracle/wsmtom_policy	Rejects inbound messages that are not in MTOM format and verifies that outbound messages are in MTOM format.

For more information about the MTOM attachment predefined policies, see "MTOM Attachment Policies" in *Security and Administrator's Guide for Web Services*.

7.3 Attaching MTOM Policies

You can attach MTOM policies to Oracle Infrastructure Web services and clients at design time using Oracle JDeveloper, or at runtime using the Oracle Enterprise Manager. For more information, see Chapter 2, "Attaching Policies to Oracle Infrastructure Web Services."

7.4 Configuring MTOM Policies

No configuration steps are required.

8 Developing RESTful Web Services

This chapter introduces RESTful Web service concepts and describes how to develop and configure RESTful Web services.

8.1 Overview of RESTful Web Services

Representational State Transfer (REST) describes any simple interface that transmits data over a standardized interface (such as HTTP) without an additional messaging layer, such as SOAP. REST provides a set of design rules for creating stateless services that are viewed as *resources*, or sources of specific information, and can be identified by their unique URIs. A client accesses the resource using the URI, a standardized fixed set of methods, and a *representation* of the resource is returned. The client is said to *transfer* state with each new resource representation.

When using the HTTP protocol to access RESTful resources, the resource identifier is the URL of the resource and the standard operation to be performed on that resource is one of the HTTP methods: GET, PUT, DELETE, POST, or HEAD.

You build RESTful endpoints using the `invoke()` method of the `javax.xml.ws.Provider<T>` interface (see `http://java.sun.com/javaee/5/docs/api/javax/xml/ws/Provider.html`). The `Provider` interface provides a dynamic alternative to building an service endpoint interface (SEI).

8.2 How RESTful Web Services Requests Are Formed and Processed

The following sections describe how RESTful Web service requests are formed on the client side and how they are processed on the server side.

8.2.1 Building HTTP Get Requests

If a SOAP endpoint that is REST enabled is deployed at the following URL:

```
http://example.com/my-app/my-service
```

Then HTTP GET requests will be accepted at the following URL:

```
http://example.com/my-app/my-service/{operationName}?{param1}={value1}&{param2}={value2}
```

In the example above, {*operationName*} specifies one of the operation names in the WSDL for the service. For RPC-literal operations, {*param1*}, {*param2*}, and so on, are the part names defined in the operation's input wsdl:message. Note that these must be simpleTypes (`xsd:int`, and so on).

> **Note:** Some browsers limit the size of the HTTP GET URL (typically less than 2000 characters). Try to keep the size of the URL small by using a limited number of parameters and short parameter names and values.

For document-literal operations, messages have only a single parameter. To simulate multiple parameters, the WSDL specifies a single parameter that is defined in the schema as a sequence. Each member of the sequence is considered a parameter. In this case, {*param1*}, {*param2*}, and so on, will be the members of the sequence type, instead of message parts. As with RPC-literal, these must be simpleTypes.

The following example illustrates the request message defined for an operation named addNumbers.

Example 8–1 GET Request on an Operation

```
<wsdl:message name="AddNumbersRequest">
      <wsdl:part name="a" type="xsd:int" />
      <wsdl:part name="b" type="xsd:int" />
</wsdl:Message>
```

This request can be invoked by using a GET with the following URL:

```
http://{yourhost}/{context-path}/{service-url}/addNumbers?a=23&b=24
```

The following example illustrates the SOAP envelope that the Oracle Web Services platform will create on the server side from the GET request. This message will be processed like any other incoming SOAP request.

Example 8–2 SOAP Envelope Created from a GET Request

```
<soap:Envelope>
    <soap:Body>
        <ns:addNumbers>
            <ns:a>23</ns:a>
            <ns:b>24</ns:b>
        </ns:addNumbers>
    <soap:Body>
<soap:Envelope>
```

The following example illustrates the request message sent for the addNumbers service when it is defined as a document-literal operation.

Example 8–3 Sample GET Request on a Document-Literal Wrapped Operation

```
<wsdl:message name="AddNumbersRequest">
     <wsdl:part name="params" type="tns:AddNumbersRequstObject" />
```

```
</wsdl:Message>
```

The following example illustrates the definition of the AddNumbersRequestObject as it would be defined in the schema.

Example 8–4 XML Definition of a Document-Literal Wrapped Operation

```
<xsd:complexType name="AddNumbersRequestObject">
    <xsd:complexContent>
        <xsd:sequence>
            <xsd:element name="a" type="xsd:int"/>
            <xsd:element name="b" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexContent>
</xsd:complexType>
```

This operation can be invoked by a GET request with the following URL.

```
http://{yourhost}/{context-path}/{service-url}/addNumbers?a=23&b=24
```

Note: This is the same URL that is used for the RPC-literal request in Example 8–1.

8.2.2 Build HTTP Post Request

RESTful Web services support HTTP POST requests that are simple XML messages—not SOAP envelopes. RESTful requests do not support messages with attachments. Since the service also supports SOAP requests, the implementation must determine if a given request is meant to be SOAP or RESTful request.

When a SOAP service receives a POST request, it looks for a SOAP action header. If it exists, the implementation will assume that it is a SOAP request. If it does not, it will find the QName of the root element of the request. If it is the SOAP envelope QName, it will process the message as a SOAP request. Otherwise, it will process it as a RESTful request.

RESTful requests will be processed by wrapping the request document in a SOAP envelope. The HTTP headers will be passed through as received, except for the Content-Type header in a SOAP 1.2 request. This Content-Type header will be changed to the proper content type for SOAP 1.2, application/soap+xml.

For example, the following RESTful request will be wrapped in the SOAP envelope illustrated in Example 8–6.

Example 8–5 RESTful Request

```
<ns:addNumbers>
    <ns:a>23</ns:a>
    <ns:b>24</ns:b>
</ns:addNumbers>
```

The following request will be processed as a normal SOAP request.

Example 8–6 SOAP Envelope Wrapping the RESTful Request

```
<soap:Envelope>
       <soap:Body>
              <ns:addNumbers>
```

```
                        <ns:a>23</ns:a>
                        <ns:b>24</ns:b>
                    </ns:addNumbers>
                </soap:Body>
</soap:Envelope>
```

8.2.3 Building RESTful Responses

For any request (either GET or POST) that was processed as a RESTful request, the response must also be in RESTful style. The server will transform the SOAP response on the server into a RESTful response before sending it to the client. The RESTful response will be an XML document whose root element is the first child element of the SOAP body. For example, assume that the SOAP envelope illustrated in the following example exists on the server.

Example 8–7 SOAP Response

```
<soap:Envelope>
    <soap:Body>
        <ns0:result xmlns:nso="…">
            <ns:title>How to Win at Poker</ns:title>
            <ns:author>John Doe</ns:author>
        </ns0:result>
    </soap:Body>
</soap:Envelope>
```

The following example illustrates the response sent back to the client. Note that the `Content-Type` will always be `text/xml`. Any SOAP headers or attachments will not be sent back to the client.

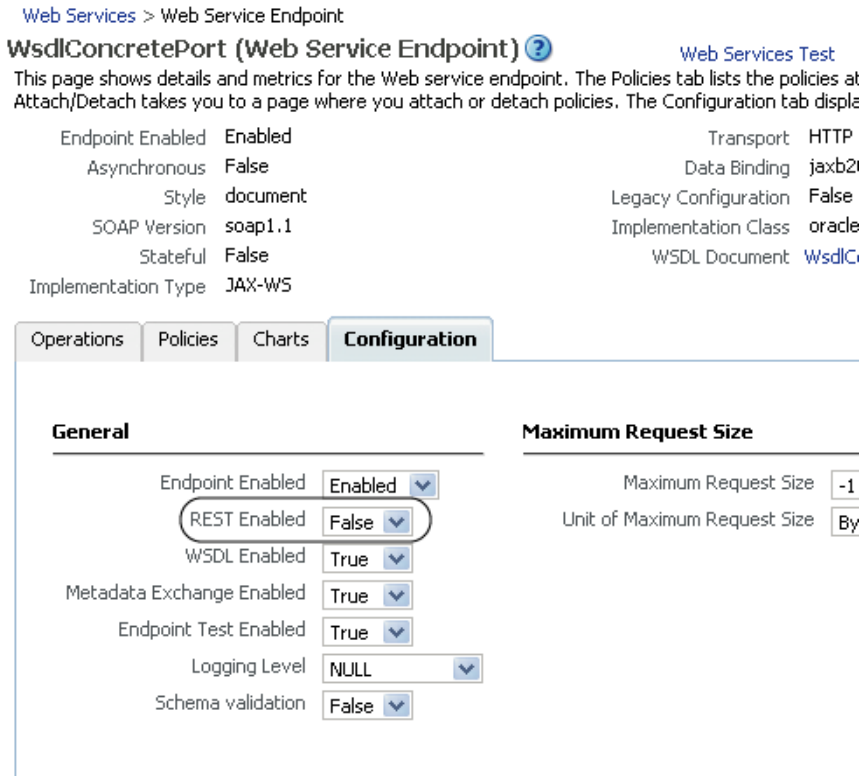
Example 8–8 RESTful Response

```
<ns0:result xmlns:ns0="…">
       <ns:title>How to Win at Poker</ns:title>
       <ns:author>John Doe</ns:author>
</ns0:result>
```

8.3 Enabling RESTful Web Services

When administering a Web service using the Oracle Enterprise Manager, you can enable it as a RESTful Web service by setting **REST Enabled** to **true** on the Configuration tab of the Web service endpoint page, as shown in the following figure. For more information, see "Enabling or Disabling RESTful Web Services" in *Security and Administrator's Guide for Web Services*.

Figure 8–1 Enabling RESTful Web Services



8.4 Limitations of RESTful Web Service Support

The following list describes the limitations in Oracle Web Services support for RESTful Web services.

- RESTful Web service support is available only for Web service applications with literal operations (both request and response should be literal).
- HTTP GET is supported only for Web service operations without (required) complex parameters.
- Some browsers limit the size of the HTTP GET URL, typically to 2000 characters or less. Try to keep the size of the URL small by using a limited number of parameters and short parameter values and names.
- RESTful Web services send only simple XML messages. You cannot send messages with attachments.
- Many management features, such as security and reliability, are not available with RESTful Web services. This is because SOAP headers, which are typically used to carry this information, cannot be used with RESTful invocations of services.
- RESTful invocations cannot be made from generated Stubs or DII clients. Invocations from those clients will be made in SOAP.
- There is no REST support for the Provider framework.
- Operation names in RESTful Web services cannot contain multibyte characters.

9 Interoperability Guidelines

This chapter provides guidelines for ensuring interoperability.

9.1 Introduction to Web Service Interoperability

The goal of the Web service architecture is to allow heterogeneous business applications to smoothly work together. The architecture is loosely coupled and based on XML standards. Web services are designed to work with each other by defining Web Service Description Language (WSDL) files as service contracts, regardless of which operating system and development technology are behind them. However, because of the complexity involved in service contracts, standards like WSDL and SOAP leave room for ambiguous interpretations. In addition, vendor-specific enhancements and extensions work against universal interoperability.

Business applications must invoke each other's services. These services are often implemented with different technologies. Interoperability failures tend to increase as Web service complexity increases. If you host a publicly available Web service, you will want to ensure that your clients from all over the world, using vastly different tool kits, can successfully invoke it. Likewise, your business application might need to integrate or interact with another vendor's Web service that was built on top of an existing legacy system and has an unusual interface design.

Interoperability issues can originate from any layer of the protocol stack. On the transport level, both parties involved in exchanging messages must agree on a specific physical transport mechanism. For example, you cannot expect to use JMS transport from a non-Java platform. This is why using basic HTTP protocol increases your chance of interoperability. On the message level, because SOAP allows virtually any type of data encoding to be used, interoperability can become difficult. For example, a standard ArrayList on a Java platform will not be automatically translated into a System.collections.ArrayList on the .NET platform. Also, interoperability issues arise at the basic WSDL and SOAP level—advanced Web service developers will find many more new challenges when they start implementing quality of service (QOS) features such as security, reliability, and transaction services.

Difficulties in interoperability do exist. However, with a few good guidelines, your Oracle Infrastructure Web service should work seamlessly with other Java EE vendor platforms or non-Java platforms like the Microsoft .NET platform.

9.2 Web Service Interoperability Organizations

As interoperability gains more and more importance in the Web service community, a number of organizations have been established to achieve this goal.

9.2.1 SOAPBuilders Community

SOAPBuilders is a loosely organized forum of developers working towards a set of tests for interoperability between SOAP implementations. Interoperability is demonstrated by implementing a canonical set of tests that are collectively defined by the participants in the forum.

The tests developed by the SOAPBuilder community are, by and large, based on vendor practices. However, practices shift over time. Clean and well-defined rules organized in a formal manner are needed for Web service vendors, Web service developers, and Web service consumers. See the following Web site for more information on SOAPBuilder tests: http://www.whitemesa.net/

9.2.2 WS-Interoperability

The Web Services Interoperability organization (WS-I) is an open industry organization that creates, promotes, and supports generic protocols for the interoperable exchange of messages between Web services. WS-I profiles are guidelines and recommendations for how the standards should be used. These profiles aim to remove ambiguities by adding constraints to the underlying specifications.

WS-I deliverables are profiles, common or best practices, scenarios, testing software, and testing materials. You should design your Web service so that it adheres to WS-I basic profiles. WS-I compliant services agree to clear contracts and have a greater chance of interoperability.

For example, a WS-I basic profile-compliant Web service should use the following features.

- Use HTTP or HTTPS as the transport binding. HTTP 1.1 is preferred over HTTP 1.0.
- Use literal style encoding. Do not use SOAP encoding.
- Use stricter fault message syntax. When a MESSAGE contains a soap:Fault element, its element children must be unqualified.
- Use XML version 1.0.
- The service should not declare array wrapper elements using the convention ArrayOfXXX.

The WS-I Web site provides more information about WS-I profiles, and the rules defined within profiles: http://www.ws-i.org/

Oracle is a member of the WS-I organization and is fully committed to helping our customers achieve interoperability. The Oracle Infrastructure Web Services platform allows a high degree of flexibility to help you create interoperable Web services.

Oracle JDeveloper supports integrated testing of WSDL files and running Web services for WS-I Basic Profile conformance. It delivers an enhanced HTTP Analyzer for monitoring and logging, and provides a built-in analysis and reporting tool to better diagnose interoperability issues. For more information, see the Oracle JDeveloper online help.

9.3 General Guidelines for Creating Interoperable Web Services

The first general guideline is to create Web services which are WS-I compliant, if possible. The WS-I profiles, however, do not solve all interoperability problems. Many Web services were implemented before WS-I profiles existed. Also, the legacy systems that you are enabling as a Web service might have placed restrictions on your designs. Thus, good practice in designing Web services should always be adopted from the beginning of the development process, whenever possible.

The following sections provide general guidelines for creating interoperable Web services.

9.3.1 Design Your Web Service Top Down

The top-down from WSDL approach enables you to design your Web service from service contracts that are not tied to any platform or language-specific characteristics. Contract-level interoperability can be ensured even before your Web service is implemented. Other Web service platform tools will be able to process your WSDL file and it is less likely that the service will be affected by existing legacy APIs.

9.3.2 Design Your Data Types Using XSD First

If possible, use an XSD schema editor to design your data types with schema types. Resist using platform-specific data types such as the .NET DataSet data type, Java collections, and so on.

9.3.3 Keep Data Types Simple

Avoid unnecessarily complex schema data types such as `xsd:choice`. Simple types provide the best interoperability and have the added benefit of improved performance.

9.3.3.1 Use Single-dimensional Arrays

Use single dimensional arrays, if possible. Use arrays of arrays instead of multi-dimensional arrays.

Multi-dimensional arrays (applicable in RPC-encoded formats only) are not supported on the .NET platform. Also, in the case of multi-dimensional arrays, the length of the inner arrays must be the same. Arrays of arrays provides flexibility in such a way that the length of contained arrays can be different.

Note: While XSD supports the definition of multi-dimensional arrays in the WSDL, programming languages such as Java map them to arrays of arrays and express the payload in a multi-dimensional format. While converting the payload to multi-dimensional format, the Java VM must ensure that the length of each inner array is the same, as well as perform other checks.

9.3.3.2 Differentiate Between Empty Arrays and Null References to Arrays

If you have an array with attributes `minoccurs=0` and `xsd:nillable=true`, for example, and the service implementation attempts to return a null reference to this array, then the representation of the payload becomes problematic. Some implementations can completely ignore this element in the payload as `minoccurs=0`,

while other implementations can specify this in the payload with the attribute `xsi:nil=true`.

The same question arises when you attempt to deserialize the array. You can deserialize either to a null reference or to an array that contains no element. In this case, the guideline is to check for null always before checking for length.

9.3.3.3 Avoid Sparse, Variable-sized, or Multi-dimensional Arrays

Although sparse, variable-sized, and multi-dimensional arrays are supported by XSD, they may not be supported by your target platform. If you are creating your Web service top down from WSDL, try to avoid these array types and use regular arrays.

9.3.3.4 Avoid Using xsd:anyType

Typically, `xsd:anyType` is mapped to `java.lang.Object` in the Java platform; this allows you to pass any run-time that will require a separate serializer and deserializers to be registered. The guideline is to find all the possible types that can be used in the runtime and define them in the WSDL.

The following example illustrates a class `MyAnyType` and two classes, `MyPossibleType1` and `MyPossibleType2`, that extend it. In this case, use `MyAnyType` instead of `xsd:anyType`, in the Web method.

Example 9–1 Classes with Member Variables

```
public class MyAnyType {
    //No member variable
}

public class MyPossibleType1 extends MyAnyType {
    //member variable.
}

public class MypossibleType2 extends MyAnyType {
    //member variable
}
..
```

9.3.3.5 Map Any Unsupported xsd:types to SOAPElement

It is possible for the presence of only one unsupported `xsd:type` in the WSDL to affect interoperability. The easy work around is to map the unsupported XSD type to SOAPElement by using a mapping mechanism, such as a JAX-RPC mapping file. Even if you have a type that is supported but fails during runtime, you can still attempt to map it to SOAPElement, then parse the SOAPElement inside the client or server.

9.3.4 Use Null Values With Care

Decide what you want to do with null values. For example, should an array be allowed to be null? Should you use a null string or an empty string? If you are sending a null value across platforms, will it cause exceptions on the receiver side?

Avoid sending null values if possible. If you must use null values in your application, design your schema types to clearly indicate that a null value is allowed.

9.3.5 Use a Compliance Testing Tool to Validate the WSDL

If your Web service is designed to be WS-I compliant, use the WS-I monitor tool to log messages and the analyzer tool to validate for conformance. You can obtain free downloads of WS-I tools from the following Web site: http://www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools/

9.3.6 Consider the Differences Between Platform Native Types

Some schema types, such as `xsd:unsignedshort` and `xsd:unsignedint`, do not always have a direct native type mapping. For example, there are no Java platform equivalent unsigned types. Schema numeric types such as `xsd:double`, `xsd:float`, and `xsd:decimal` might have different precisions once mapped to their platform native types.

There are also limitations on the `xsd:string` type. The strings must not contain illegal XML characters and the \r (carriage return) character will typically be mapped to the \n (line feed) character.

Use `byte[]` instead of `xsd:string` when you do not know the character set that the source data uses. Binary content is more interoperable than text content.

If you are creating the Web service bottom up from Java classes, you must ensure that you use the closest possible data type. Thus, use Java data types closer to the `xsd:type`. For example, if you want to use `xsd:dateTime` to represent a date and time, then use the `javax.xml.datatype.XMLGregorianCalendar` data type instead of `java.lang.Data` or `java.util.Calendar`. The `XMLGregorianCalendar` data type can return a more precise time because it can store fractional seconds.

If you are creating the Web service top down from WSDL, use mapping files to map the `xsd:dateTime` to `XMLGregorianCalendar` if time accuracy is very important.

Note: All of the pre-Java EE platforms, such as J2EE 1.*, used `Calendar`, but now Java EE-compliant platforms use `XMLGregorianCalendar`. This provides better .Net interoperability because it can handle long fractions of seconds.

9.3.7 Avoid Using RPC-Encoded Message Format

By itself, the RPC-encoded message format does not imply that you will not be able to interoperate with other platforms and clients. In many cases, there are RPC-encoded Web services which are in use today. The reason to move away from RPC-encoded message formats is to avoid some of the edge cases where different interpretations of the underlying specification and implementation choices break interoperability. Some examples include the treatment of sparse arrays, multi-dimensional arrays, custom fault code QNames, un-typed payloads, and so on.

9.3.8 Avoid Name Collisions

If you are creating Web services bottom up from Java classes, you can avoid name collisions by using explicit package names in your classes. If you are creating Web services top down from WSDL, you can avoid name collisions by using unique namespace URIs.

Note: By default, when most Web service assembly tools use the top down approach, they will try to derive a package name for the generated classes from the namespace URI. When they use the bottom up approach, they will try to use the Java package name to derive a namespace URI.

Unfortunately, because of valid package name limitations, this derivation is not 1-1. So, specifying namespace URI in such a way that it does not produce a conflicting package name that another namespace URI has yielded, is very important.

9.3.9 Use Message Handlers, Custom Serializers, or Interceptors

You can use message handlers, custom serializers, or interceptors to easily fix some interoperability issues. This idea is to fix the issue on the payload at a very granular level, by intercepting and changing the message either before it encounters the deserializers, or after the payload is generated by the serializers and before it is put on the wire.

9.3.10 Apply WS-* Specifications Judiciously

Many WS-* specifications are in early adoption phase, and could potentially reveal interoperability issues with different stacks. A suggestion would be to make sure that a WS-* feature is absolutely required before applying it. Another suggestion would be to apply features that are most commonly used in the Web services space. For example, if you are in a situation where there are several possible options, such as choosing either Basic Authentication, X.509, or Kerberos for security, then choose the option which is most commonly used in the Web services space.

A

Annotation Reference

This appendix describes the asynchronous Web service and policy annotations that are used by Oracle Infrastructure Web services.

A.1 Overview of Annotations

The WebLogic Web Services programming model uses the JDK 5.0 metadata annotations at `http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#annotations` feature (specified by JSR-175 at `http://www.jcp.org/en/jsr/detail?id=175`). In this programming model,

you create an annotated Java file to specify the shape and characteristics of the Web service.

Table A–1 summarizes the asynchronous Web service and policy annotations described in this appendix.

For more information about the annotations available, see *Oracle Fusion Middleware Java API Reference for Oracle Web Services*. For more information about the predefined policies, see "Predefined Polices" in *Security and Administrator's Guide for Web Services*.

Table A–1 Oracle Infrastructure Web Service Annotations

Annotation	Description
Section A.2, "@AddressingPolicy Annotation"	Attaches a WS-Addressing policy to the Web service.
Section A.3, "@AsyncWebService Annotation"	Declares a Web service to be an asynchronous Web service.
Section A.4, "@AsyncWebServiceQueue Annotation"	Specifies the details of a queue used for saving the request for later processing.
Section A.5, "@AsyncWebServiceResponseQueue Annotation"	Defines the queue used to save the response for later processing
Section A.6, "@CallbackAddressingPolicy Annotation"	Not used in this release.
Section A.7, "@CallbackManagementPolicy Annotation"	Attaches a management policy to the callback client of the asynchronous Web service that will connect to the callback service.
Section A.8, "@CallbackMethod Annotation"	Enables you to customize the names of the WSDL entities for the corresponding operation in the callback portType, set whether a method is synchronous or asynchronous, and disable the automatic sending of the response.
Section A.9, "@CallbackMtomPolicy Annotation"	Attaches an MTOM policy to the callback client of the asynchronous Web service that will connect to the callback service.
Section A.10, "@CallbackProperties Annotation"	Enables you to specify properties that are required in the message context when calling the callback service.
Section A.11, "@CallbackSecurityPolicy Annotation"	Attaches one or more security polices to the callback client of the asynchronous Web service that will connect to the callback service.
Section A.12, "@ManagementPolicy Annotation"	Attaches a management policy to the Web service.
Section A.13, "@MtomPolicy Annotation"	Attaches an MTOM policy to the Web service.
Section A.14, "@PortableWebService Annotation"	Specifies, at the class level, that the JWS file implements an Oracle Infrastructure Web service.
Section A.16, "@Property Annotation"	Enables you to specify a single property that is required in the message context when calling the callback service.

Table A–1 (Cont.) Oracle Infrastructure Web Service Annotations

Annotation	Description
Section A.17, "@ResponseWebService Annotation"	Customizes the response Web service port information.
Section A.19, "@SecurityPolicies Annotation"	Attaches a list of security policies to the Web service.
Section A.20, "@SecurityPolicy Annotation"	Attaches a security policy to the Web service.

A.2 @AddressingPolicy Annotation

The `oracle.webservices.annotations.AddressingPolicy` annotation attaches a WS-Addressing policy to the Web service.

For example:

```
@AddressingPolicy(
       value="oracle/wsaddr_policy",
        enabled = "true")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.AddressingPolicy` annotation.

Table A–2 Attributes for oracle.webservices.annotations.AddressingPolicy Annotation

Attribute	Description	Default
`value`	Location from which to retrieve the WS-Policy file. Use the `http:` prefix to specify the URL of a WS-Policy file on the Web. Use the `policy:` prefix to specify that the WS-Policy file is packaged in the policy repository.	""
`enabled`	Boolean value that specifies whether or not the policy is enabled.	true

A.3 @AsyncWebService Annotation

The `oracle.webservices.annotations.async.AsyncWebService` annotation declares a Web service to be an asynchronous Web service.

By default, all operations associated with the Web service are asynchronous. To mark a specific method as synchronous, see Section A.8, "@CallbackMethod Annotation". If you want to be able to call a method both synchronously and asynchronously, you will have to create two methods and annotate them accordingly.

For example:

```
@PortableWebService
@AsyncWebService
public class HelloService {
   public String hello(String name) {
      return "Hi " + name;
   }
}
```

A.4 @AsyncWebServiceQueue Annotation

The `oracle.webservices.annotations.async.AsyncWebServiceQueue` annotation defines the queue used to save the request for later processing. For more information, see Section 4.4, "Creating the Request and Response Queues."

For example:

```
@AsyncWebServiceQueue (
    connectionFactory = "weblogic.jms.XAConnectionFactory",
    queue = "oracle.j2ee.ws.server.async.NonDefaultRequestQueue",
    enableTransaction = true
    transactionTimeout=3600
)
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.AsyncWebServiceQueue` annotation.

Table A–3 Attributes for oracle.webservices.annotations.async.AsyncWebService Queue Annotation

Attribute	Description	Default
`connectionFactory`	Name of the JMS queue connection factory for the request queue.	weblogic.jmx.XAConnection Factory
`queue`	Name of the JMS queue used to store asynchronous requests.	oracle.j2ee.ws.server.async. DefaultRequestQueue
`enableTransaction`	Flag that specifies whether the MDB should process the asynchronous request as a transaction. **Note**: A user transaction is maintained separately from the MDB transaction so that they can be committed or rolled back independently.	false
`transactionTimeout`	Transaction timeout in seconds for processing a queued message. This attribute is meaningful only if the transaction is enabled or the method is non-idempotent. The timeout is applied to the MDB that processes the request messages.	0—Server-level configured value is used by default.

A.5 @AsyncWebServiceResponseQueue Annotation

The `oracle.webservices.annotations.async.AsyncWebServiceResponseQueue` annotation defines the queue used to save the response for later processing.

For example:

```
@AsyncWebServiceQueue (
    connectionFactory = "weblogic.jms.XAConnectionFactory",
    queue = "oracle.j2ee.ws.server.async.NonDefaultResponseQueue",
    enableTransaction = true
)
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.AsyncWebServiceResponseQueue` annotation.

Table A–4 Attributes for oracle.webservices.annotations.async.AsyncWebService ResponseQueue Annotation

Attribute	Description	Default
`enabled`	Flag that specifies whether the response queue is enabled. If enabled, the request MDB processes the request and stores the response in the response queue. If disabled, the response processing is performed by the request MDB.	true
`connectionFactory`	Name of the JMS queue connection factory for the response queue.	weblogic.jmx.XAConnection Factory
`queue`	Name of the JMS queue used to asynchronous response. The response MDB retrieves messages from the response queue and forwards them to the callback service.	oracle.j2ee.ws.server.async. DefaultResponseQueue
`enableTransaction`	Flag that specifies whether the MDB should process the asynchronous response as a transaction. **Note**: A user transaction is maintained separately from the MDB transaction so that they can be committed or rolled back independently.	false

A.6 @CallbackAddressingPolicy Annotation

The `oracle.webservices.annotations.async.CallbackAddressingPolicy` annotation is not used in this release as WS-Addressing is used by default to correlate the response message with the callback service. In the future, it may be used to advertise the policy to the callback exchange.

A.7 @CallbackManagementPolicy Annotation

The `oracle.webservices.annotations.async.CallbackManagementPolicy` annotation attaches a management policy to the callback client of the asynchronous Web service that will connect to the callback service. By default, no management policy is attached.

For example:

```
@CallbackManagementPolicy("oracle/log_policy")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.CallbackManagementPolicy` annotation.

Table A–5 Attributes for oracle.webservices.annotations.async.CallbackManagementPolicy Annotation

Attribute	Description	Default
`value`	URI of a management policy.	""
`enabled`	Flag that specifies whether the specified security policies are enabled for the callback client.	true
`sendAutoResponse`	Flag that specifies whether the response is sent automatically upon completion of the asynchronous operation. In some cases, you may wish to disable automatic sending of the response, and access it manually, when the application is ready.	true

A.8 @CallbackMethod Annotation

The `oracle.webservices.annotations.async.CallbackMethod` annotation enables you to customize the names of the WSDL entities for the corresponding operation in the callback portType and set whether a method is synchronous or asynchronous. This annotation is similar to the javax.jws.WebMethod annotation.

For example:

```
@CallbackMethod(exclude=true)
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.CallbackMethod` annotation.

Table A–6 Attributes for oracle.webservices.annotations.async.CallbackMethod Annotation

Attribute	Description	Default
`name`	Name of the callback method in the callback portType that corresponds to the annotated method. This annotation is ignored at runtime; it is only used when generating a callback interface from a POJO implementation class.	"onResult" + <name of the annotated method>
`operationName`	Name of the wsdl:operation for this method in the callback portType.	"onResult" + <name of the annotated method>
`action`	Name of the action for this method in the callback portType. For SOAP bindings, this value determines the value of the SOAP action.	Name of the operation
`serviceRefName`	Name used for the service reference used to send the response message. This value must be unique for the deployed archive and is used to define: ■ `<service-ref-name>` and `<display-name>` elements under the `<service-ref>` element in the standard deployment descriptor. ■ Name attribute in the `<service-ref-mapping>` element in the proprietary deployment descriptor	<name of the annotated class> + "ResponseWebService.SERVICE_REF_NAME_SUFFIX"
`exclude`	Flag that specifies whether the method is asynchronous. Setting this value to true specifies that the method is synchronous; in this case, all other attribute settings are ignored.	false

A.9 @CallbackMtomPolicy Annotation

The `oracle.webservices.annotations.async.CallbackMtomPolicy` annotation attaches an MTOM policy to the callback client of the asynchronous Web service that will connect to the callback service. By default, no MTOM policy is attached.

For example:

```
@CallbackMtomPolicy("oracle/wsmtom_policy")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.CallbackMtomPolicy` annotation.

Table A–7 Attributes for oracle.webservices.annotations.async.CallbackMtomPolicy Annotation

Attribute	Description	Default
`value`	URI of an MTOM policy.	""
`enabled`	Flag that specifies whether the specified security policies are enabled for the callback client.	true

A.10 @CallbackProperties Annotation

The `oracle.webservices.annotations.async.CallbackProperties` annotation enables you to specify a set of properties that are required in the message context when calling the callback service.

For example:

```
@CallbackProperties(
   {
   @Property(
       key = SecurityConstants.ClientConstants.WSS_CSF_KEY,
        value = "basic.credentials")
   }
)
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.CallbackProperties` annotation.

Table A–8 Attributes for oracle.webservices.annotations.async.CallbackProperties Annotation

Attribute	Description	Default
`value`	Array of `oracle.webservices.annotations.Property` type values. Each property specifies a key-value pair.	""

A.11 @CallbackSecurityPolicy Annotation

The `oracle.webservices.annotations.async.CallbackSecurityPolicy` annotation attaches one or more security polices to the callback client of the asynchronous Web service that will connect to the callback service. By default, no security policies are attached.

For example:

```
@CallbackSecurityPolicy("oracle/wss10_saml_token_with_message_protection_client_
policy")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.async.CallbackSecurityPolicy` annotation.

Table A–9 Attributes for oracle.webservices.annotations.async.CallbackSecurityPolicy Annotation

Attribute	Description	Default
`value`	Array of String type values that specifies the URL of a client-side security policy.	""
`enabled`	Flag that specifies whether the specified security policies are enabled for the callback client.	true

A.12 @ManagementPolicy Annotation

The `oracle.webservices.annotations.ManagementPolicy` annotation attaches a management policy to the Web service.

For example:

```
@ManagementPolicy(
       value="oracle/log_policy",
        enabled = "true")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.ManagementPolicy` annotation.

Table A–10 Attributes for oracle.webservices.annotations.ManagementPolicy Annotation

Attribute	Description	Default
`value`	URI of a management policy.	""
`enabled`	Boolean value that specifies whether or not the policy is enabled.	""

A.13 @MtomPolicy Annotation

The `oracle.webservices.annotations.MtomPolicy` annotation attaches an MTOM policy to the Web service.

For example:

```
@MtomPolicy(
       value="oracle/wsmtom_policy",
        enabled = "true")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.MtomPolicy` annotation.

Table A–11 Attributes for oracle.webservices.annotations.MtomPolicy Annotation

Attribute	Description	Default
`value`	URI of an MTOM policy.	""
`enabled`	Boolean value that specifies whether or not the policy is enabled.	""

A.14 @PortableWebService Annotation

The `oracle.webservices.annotations.PortableWebService` annotation specifies, at the class level, that the JWS file implements an Oracle Infrastructure Web service.

For example:

```
@PortableWebService
@AsyncWebService
public class HelloService {
   public String hello(String name) {
      return "Hi " + name;
   }
}
```

The following table defines the attributes that are passed to the `oracle.webservices.annotations.PortableWebService` annotation.

Table A–12 Attributes for oracle.webservices.annotations.PortableWebServiceAnnotation

Attribute	Description	Default
`endpointInterface`	Name of the service endpoint interface defining the service's abstract Web service contract. This annotation allows you to separate the interface contract from the implementation. If this annotation is present, the service endpoint interface is used to determine the abstract WSDL contract (portType and bindings). The service endpoint interface can include JSR-181 annotations to customize the mapping from Java to WSDL. The service implementation may implement the service endpoint interface, but this is not required. If this attribute is not specified, the Web service contract is generated from the annotations defined for the service implementation. If a service endpoint interface is required by the target environment, it will be generated into an implementation-defined package with an implementation-defined name. **Note**: This attribute is invalid when annotating an endpoint interface.	""
`name`	Name of the Web service. This name is used for the wsdl:portType when mapped to WSDL 1.1.	""
`portName`	Port name of the Web service. This name is used for the wsdl:port when mapped to WSDL 1.1. **Note**: This attribute is invalid when annotating an endpoint interface.	""

Table A–12 (Cont.) Attributes for oracle.webservices.annotations.PortableWebServiceAnnotation

Attribute	Description	Default
`serviceName`	Service name of the Web service. This name is used for the wsdl:service when mapped to WSDL 1.1. **Note**: This attribute is invalid when annotating an endpoint interface.	""
`targetNamespace`	Target namespace. If annotating a service endpoint interface, the target namespace used for the wsdl:portType and all associated elements. If annotating a service implementation that does not reference a service endpoint interface (through the endpointInterface attribute), the target namespace is used for the wsdl:portType, wsdl:service, and the associated XML elements. If annotating a service implementation that references a service endpoint interface (through the endpointInterface attribute), the target namespace is used for the wsdl:portType, wsdl:service, and the associated XML elements.	""
`wsdlLocation`	Location of the predefined WSDL describing the Web service. The value is a URL (relative or absolute) that refers to a pre-existing WSDL file. This attribute indicates that the service implementation is implementing a predefined WSDL contract. If there is an inconsistency between the service implementation and the portType and bindings declared in the WSDL, an error message is returned. Note that a single WSDL file may contain multiple portTypes and multiple bindings. The annotations on the service implementation determine the specific portType and bindings that correspond to the Web service.	""

A.15 @PortableWebServiceProvider Annotation

The `oralce.webservices.annotations.PortableWebServiceProvider` annotation marks a Java class as implementing an Oracle Infrastructure Web Services Provider.

For example:

```
@PortableWebServiceProvider
@AsyncWebService
public class HelloService {
   public String hello(String name) {
      return "Hi " + name;
   }
}
```

The following table defines the attributes that are passed to the `oracle.webservices.annotations.PortableWebServiceProvider` annotation.

Table A–13 Attributes for oracle.webservices.annotations.PortableWebServiceProvider Annotation

Attribute	Description	Default
`portName`	Name of the Web service port.	""
`serviceName`	Name of the Web service.	""
`targetNamespace`	Target namespace.	""
`wsdlLocation`	Location of the predefined WSDL describing the Web service. The value is a URL (relative or absolute) that refers to a pre-existing WSDL file. This attribute indicates that the service implementation is implementing a predefined WSDL contract. If there is an inconsistency between the service implementation and the portType and bindings declared in the WSDL, an error message is returned. Note that a single WSDL file may contain multiple portTypes and multiple bindings. The annotations on the service implementation determine the specific portType and bindings that correspond to the Web service.	""

A.16 @Property Annotation

The `oracle.webservices.annotations.Property` annotation enables you to specify a single property that is required in the message context when calling the callback service.

For example:

```
@Property(
       key = SecurityConstants.ClientConstants.WSS_CSF_KEY,
        value = "basic.credentials")
```

The following table defines the attributes that can be passed to the `oracle.webservices.annotations.CallbackProperty` annotation.

Table A–14 Attributes for oracle.webservices.annotations.async.CallbackProperty Annotation

Attribute	Description	Default
`key`	String type value that specifies the message context property name.	""
`value`	String type value that specifies the message context property value.	""

A.17 @ResponseWebService Annotation

The `oracle.webservices.annotations.async.ResponseWebService` annotation enables you to customize the response Web service port information.

The following table defines the attributes that are passed to the `oracle.webservices.annotations.async.ResponseWebService` annotation.

Table A–15 Attributes for oracle.webservices.annotations.async.ResponseWebService Annotation

Attribute	Description	Default
`name`	Name of the response portType. Used as the name of the wsdl:portType when mapped to WSDL 1.1.	<name of the annotated class> + "Response"
`targetNamespace`	Target namespace used for the wsdl:portType, wsdl:service, and all associated elements.	Same as the asynchronous Web service target namespace
`partnerLinkRole`	Role name of the callback portType used in the partner link element of the WSDL.	N/A
`serviceRefName`	Service reference name used for the service reference that is used to send the response message. This value must be unique for the deployed archive and is used to define: ■ <service-ref-name> and <display-name> elements under the <service-ref> element in the standard deployment descriptor. ■ Name attribute in the <service-ref-mapping> element in the proprietary deployment descriptor	<name of the annotated class> + "ResponseWebService.SERVICE_REF_NAME_SUFFIX"
`serviceName`	Not used in this release.	N/A
`portName`	Not used in this release.	N/A

A.18 @Retry Annotation

The `oracle.webservices.annotations.async.Retry` annotation specifies whether the asynchronous method is idempotent, or retriable, in the event that its execution is terminated abnormally (for example, due to system failure). This annotation can be applied at the class or method level; the method-level setting overrides the class-level setting.

By default, all asynchronous methods are idempotent; this implies that there are no side effects of calling the asynchronous method more than once. If an asynchronous method is not idempotent, you should explicitly set this annotation with the `enable` attribute set to false.

The following table defines the attributes that are passed to the `oracle.webservices.annotations.Retry` annotation.

Table A–16 Attributes for oracle.webservices.annotations.Retry Annotation

Attribute	Description	Default
`enable`	Flag that specifies whether the method is idempotent and can be retried safely in the event that execution is abnormally terminated.	true
`supportsGlobalTransaction`	Flag that specifies whether the method can participate in the global transaction. This field is used only if there are no other annotations specifying the global transaction behavior, such as `javax.ejb.TransactionAttribute` or WS-AT annotation. For example, if the transaction type is set to "Required", "Mandatory" and "Supports" then the method is assumed to participate in the global transaction. **Note**: If this field is specified in addition to a global transaction specific annotation, then their values should be consistent. If the method can participate in a global transaction, set the `supportsGlobalTransaction` attribute to true. In this case, the system can achieve the non-idempotent behavior in the most reliable way. The execution of the method and the posting of the response are accomplished within a single global transaction. If any of the processing steps result in an abnormal termination, the transaction will be rolled back and the method will be executed again in a new transaction. In this scenario, it is guaranteed that the method is executed only once, as part of a completed transaction, and the result of the execution is posted only once to the response queue. If the method cannot participate in a global transaction, set the `supportsGlobalTransaction` to false. In this case, the system guarantees that the non-idempotent method is executed "at most once." A fault response message is returned to the callback service in the event either of the following occurs: ■ If execution of the asynchronous method does not complete due to abnormal termination. ■ If there is an abnormal termination after completely executing the method, but before the response has been posted to the queue.	false

A.19 @SecurityPolicies Annotation

The `oracle.webservices.annotations.SecurityPolicies` annotation specifies an array of `@SecurityPolicy` annotations. Use this annotation if you want to attach more than one WS-Policy files to a class.

For example:

```
@SecurityPolicies({
    @SecurityPolicy(value=
    "policy:oracle/wss10_username_token_with_message_protection_server_policy"),
    @SecurityPolicy(value=
    "policy:oracle/authorization_policy")})
```

A.20 @SecurityPolicy Annotation

The `oracle.webservices.annotations.SecurityPolicy` annotation attaches a security policy to the request or response SOAP message. This annotation can be used on its own to apply a single WS-Policy file to a class. If you want to apply more than one WS-Policy file to a class, use the `@SecurityPolicies` annotation to group them together.

For example:

```
@SecurityPolicy(value=
    "policy:oracle/wss10_username_token_with_message_protection_server_policy"),
```

The following table summarizes the attributes that you can pass to the `oracle.webservices.annotations.SecurityPolicy` annotation.

Table A–17 Attributes for oracle.webservices.annotations.SecurityPolicy Annotation

Attribute	Description	Default
`value`	Location from which to retrieve the WS-Policy file. Use the `http:` prefix to specify the URL of a WS-Policy file on the Web. Use the `policy:` prefix to specify that the WS-Policy file is packaged in the policy repository.	""
`enabled`	Optional. Boolean value that specifies whether the policy is enabled.	true
`Properties`	Optional. Array of property value-name pairs.	""