Oracle® Fusion Middleware

Developing Oracle Infrastructure Web Services

14*c* (14.1.2.0.0)
G12135-01
December 2024

Oracle Fusion Middleware Developing Oracle Infrastructure Web Services, 14*c* (14.1.2.0.0)

G12135-01

Primary Author: Panendra Puttachar

Contents

List of Figures

List of Tables

Preface

This preface describes the intended audience, document accessibility features, and conventions used in this guide.

Audience

This document is intended for programmers that are developing Oracle Infrastructure web services, including SOA, Application Development Framework (ADF) services.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Fusion Middleware 12*c* documentation set:

Table 1 Related Documents

Document	Description
Understanding Web Services	Provides an introduction to web services for Oracle Fusion Middleware 12c.
Understanding WebLogic Web Services for Oracle WebLogic Server	Provides an introduction to WebLogic Web Services (Java EE).
Understanding Oracle Web Services Manager	Introduces Oracle Web Services Manager (OWSM) used for web services policy attachment and management.
Administering Web Services	Describes how to secure and administer web services.
Securing Web Services and Managing Policies with Oracle Web Services Manager	Describes how to secure web services using OWSM policies. This document also describes how to create and manage OWSM policies.
Use Cases for Securing Web Services Using Oracle Web Services Manager	Provides use cases that demonstrate how to secure web services using OWSM.

Table 1 (Cont.) Related Documents

Document	Description
Extensibility Guide for Oracle Web Services Manager	Describes how to build custom policy assertions for OWSM.
Interoperability Solutions Guide for Oracle Web Services Manager	Describes how to implement the most common OWSM interoperability scenarios.
Developing SOA Applications with Oracle SOA Suite	Describes how to develop SOA composite services.
Developing Fusion Web Applications with Oracle Application Development Framework	Describes how to develop ADF components.
"Developing and Securing Web Services" in *Developing Applications with Oracle JDeveloper*	Describes how to develop web services and attach policies using Oracle JDeveloper.

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.

What's New in This Guide

The following topics introduce the new and changed features of Oracle Infrastructure web services and other significant changes that are described in this guide, and provides pointers to additional information.

New and Changed Features for 14*c* (14.1.2.0.0)

This revision contains no new features. Minor updates were made throughout the guide.

1

Introduction to Oracle Infrastructure Web Services

An introduction of Oracle Infrastructure web services, description of concepts for developing Oracle Infrastructure web services, and an overview of supported standards and related documentation is described in this chapter.

- Overview of Oracle Infrastructure Web Services
- Supported Standards for Developing Oracle Infrastructure Web Services
- Related documents for developing Oracle Infrastructure Web Services

For definitions of unfamiliar terms found in this and other books, see the Glossary.

1.1 Overview of Oracle Infrastructure Web Services

In Oracle Fusion Middleware 12*c*, 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, Application Development Framework (ADF), Oracle Service Bus, and Oracle Enterprise Scheduler services
- Java EE web services—SOAP (JAX-WS) and RESTful (JAX-RS) web services

For more information about the web service and client types, see "Overview of Web Services in Oracle Fusion Middleware 12c" in *Understanding Web Services*.

For more information about Java EE web services, see Overview of WebLogic Web Services *Understanding WebLogic Web Services for Oracle WebLogic Server*.

1.2 Supported Standards for Developing Oracle Infrastructure Web Services

Oracle considers interoperability of web services platforms to be more important than providing support for all possible edge cases of the web services specifications.

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

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 *Understanding Oracle Web Services Manager*.

Table 1-1 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**—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
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 1-1 (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 Services Security Standards" in *Understanding Oracle Web Services Manager*. • **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 • **WS-SecureConversation**—This specification defines extensions that are build on WS-Security to allow security context establishment and sharing, and session key derivation for multiple message exchanges. For more information, see the following URLs: – http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html – http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/ws-secureconversation.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 1-1 (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
Optimizing XML transmission	• **Fast Infoset**—Compressed binary encoding format that provides a more efficient serialization than the text-based XML format. Fast Infoset optimizes both document size and processing performance.The Fast Infoset specification, *ITU-T Rec. X.891 and ISO/IEC 24824-1 (Fast Infoset)* is defined by both the ITU-T and ISO standards bodies. The specification can be downloaded from the ITU Web site: http://www.itu.int/rec/T-REC-X.891-200505-I/en • **Message Transmission Optimization Mechanism** (MTOM)—Defines a method for optimizing the transmission of XML data of type `xs:base64Binary` or `xs:hexBinary` in SOAP messages.
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.

1.3 Related documents for developing Oracle Infrastructure Web Services

Oracle Infrastructure web services development is one of the two categories of web service supported by Oracle Fusion Middleware 12c.

Refer the following section for a summary of documents related to Oracle Infrastructure web services development, security, and administration: Related Documents

2

Understanding How Policies Attach to Oracle Infrastructure Web Services

Certain policies and policy sets are attached to Oracle Infrastructure web services to manage and secure web services consistently across your organization.
This chapter describes policies and policy sets, provides information about the OWSM predefined policies and templates, and explains how policies attach to Oracle Infrastructure web services. It included the following sections:

2.1 What Are Policies and Policy Sets?

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.

For more information, see "Understanding Polices" in *Understanding Oracle Web Services Manager*.

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 sets provide the ability to specify a runtime constraint that determines the context in which the policy set is relevant. For example, you can specify that a service use message protection when communicating with external clients only since the message may be transmitted over insecure public networks. However, when communicating with internal clients on a trusted network, message protection may not be required. For more information about policy sets, see "Global Policy Attachments Using Policy Sets" in *Understanding Oracle Web Services Manager*.

2.2 Understanding OWSM Predefined Policies and Assertion Templates

Oracle Web Services Manager (OWSM) provides a policy framework to manage and secure web services consistently across your organization.

OWSM can be used by both developers, at design time, and system administrators in production environments. For more information about the OWSM policy framework, see "Understanding the OWSM Policy Framework" in *Understanding Oracle Web Services Manager*.

There is a set of predefined OWSM 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 OWSM policies and assertion templates, see the following sections in *Securing Web Services and Managing Policies with Oracle Web Services Manager*:

- "Predefined Policies"
- "Predefined Assertion Templates"

2.3 Overview of How Policies Attach to Web Services

Security policies provide a framework to manage and secure web services consistently across your organization.

Security policies can be attached directly to web services endpoints:

- Programmatically, at design time, using annotations. When developing an application using JDeveloper, you can take advantage of the wizards available to attach policies to web services and clients.
- Post-deployment using Oracle Fusion Middleware and WLST.

In addition, policy sets provide a means to attach policies globally to a range of endpoints of the same type.

For complete details, see "Attaching Policies" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

3

Introduction to Securing Oracle Infrastructure Web Services

You have to secure Oracle Infrastructure web services.

This chapter describes how to secure Oracle Infrastructure web services. It includes the following sections:

3.1 Overview of Web Services Security

Oracle Web Services Manager (WSM) is designed to define and implement web services security.

Web services security includes several aspects, as described below:

- **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:
  - Something one has, for example, credentials issued by a trusted authority such as a passport (real world) or a smart card (IT world).
  - Something one knows, for example, a shared secret such as a password.
  - 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 *Understanding Oracle Web Services Manager*.

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, OWSM 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 OWSM, see "Understanding OWSM Policy Framework" in *Understanding Oracle Web Services Manager*.

3.2 About OWSM Predefined Security Policies and Assertion Templates

OWSM provides a set of predefined policies and assertion templates that are automatically available when you install Oracle Fusion Middleware.

For more information, seeUnderstanding How Policies Attach to Oracle Infrastructure Web Services .

OWSM 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 in this pre-defined set:

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

For more information about the predefined OWSM policies and assertion templates, see the following sections in *Securing Web Services and Managing Policies with Oracle Web Services Manager*:

- "Predefined Policies"
- "Predefined Assertion Templates"

For assistance in determining which security policies to use, see "Determining Which Security Policies to Use" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

3.3 About Security Policies Attachment

You can attach security policies to Oracle Infrastructure web services and clients at design time using Oracle JDeveloper, or at runtime using the Fusion Middleware Control.

For more information see Understanding How Policies Attach to Oracle Infrastructure Web Services .

3.4 About Security Policies Configuration

You must configure the security policies before you can use them in your environment.

The steps to configure security policies are described in "Securing Web Services" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

4

Introduction to Developing Asynchronous Web Services

An introduction of asynchronous web service concepts and a description of how to develop and configure asynchronous web services is detailed in this chapter.

It includes the following topics:

- Understanding Asynchronous Web Services
- About Using JDeveloper to Develop and Deploy Asynchronous Web Services
- Annotation to Develop an Asynchronous Web Service
- Creating the Request and Response Queues
- Annotation to Configure the Callback Service
- Configuring SSL for Asynchronous Web Services
- Defining Asynchronous Web Service Clients
- Attaching Policies to Asynchronous Web Services and Clients

4.1 Understanding Asynchronous Web Services

A client can continue its processing, without interruption by calling a web service asynchronously.

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:

- Understanding the Flow of an Asynchronous Web Service Using a Single Request Queue
- Understanding the Flow of an Asynchronous Web Service Using a Request and a Response Queue
- Understanding the Client Perspective of an Asynchronous Web Service Call
- Understanding How Asynchronous Messages Are Correlated

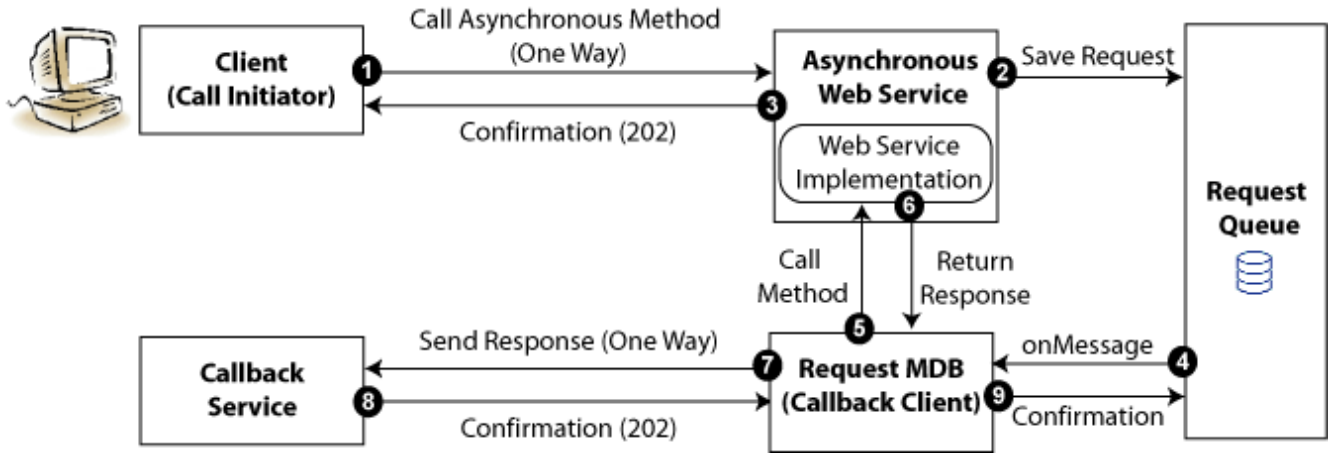
4.1.1 Understanding the Flow of an Asynchronous Web Service Using a Single Request Queue

In an asynchronous web service using a single request queue there is a single message-driven bean (MDB) associated with the request queue that handles both the request and response processing.

> **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 Understanding the Flow of an Asynchronous Web Service Using a Request and a Response Queue, and not the single request queue scenario.

The following diagram illustrates the flow of an asynchronous web service using a single request queue. In this scenario, there is a single message-driven bean (MDB) associated with the request queue that handles both the request and response processing.

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



The following steps describe 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.

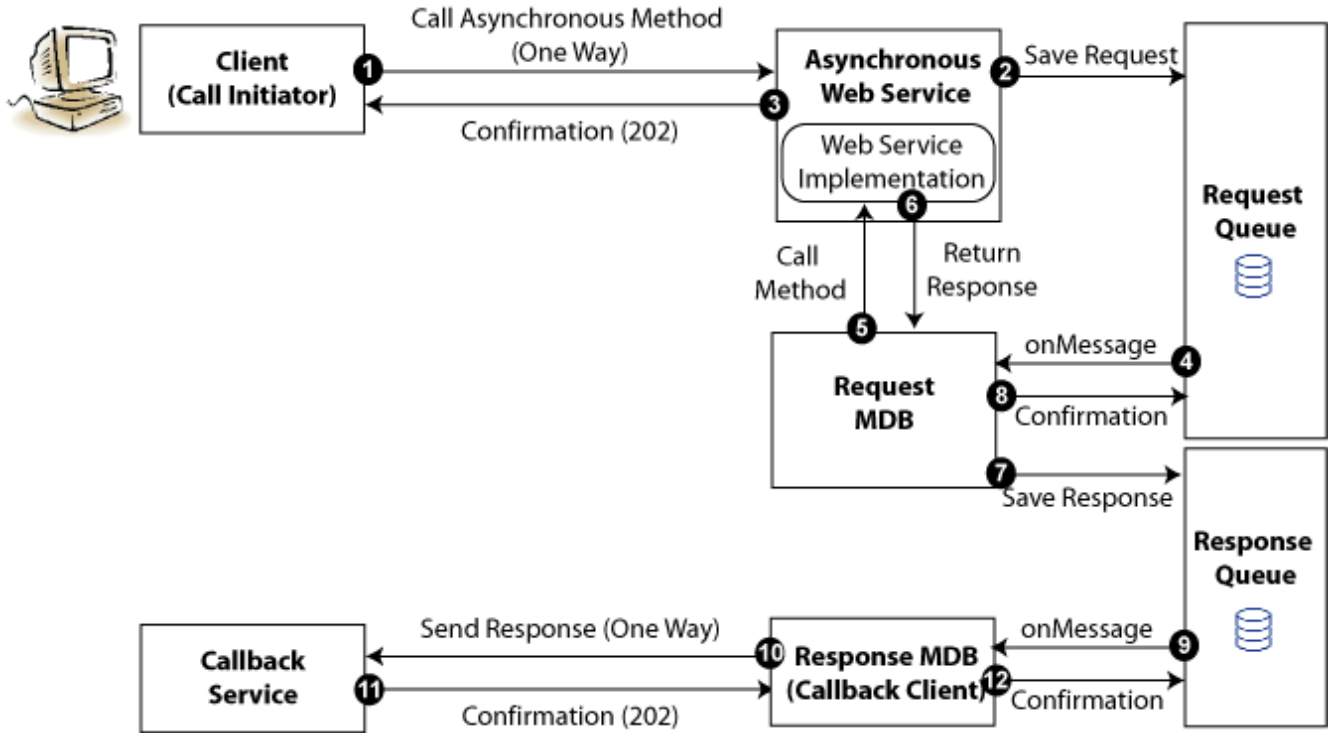
In this scenario, if there is a problem connecting to the callback service (in Step 7), then the response will not be sent. If the request is retried later, the flow resumes from Step 4 and the web service implementation will be called again (in Step 5). This may not be desirable depending on your application 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 Understanding the Flow of an Asynchronous Web Service Using a Request and a Response Queue

Use of two MDBs provide improved error recovery over the single queue model.

The following diagram illustrates the flow of an asynchronous method call using a single request 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 Understanding the Flow of an Asynchronous Web Service Using a Single Request Queue.

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



The following steps describe 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.

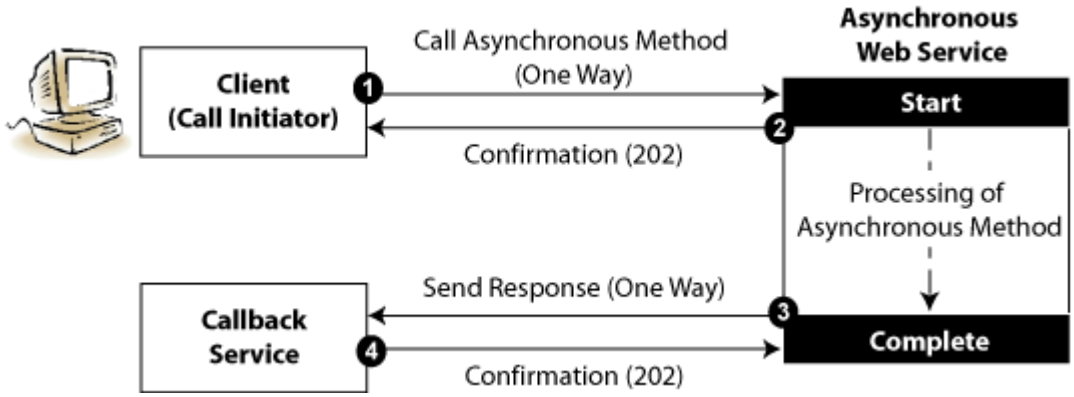
3. The asynchronous web service sends a receipt confirmation to the client.
4. The MDB listener on the request queue receives the message and initiates processing of the request.
5. The request MDB calls the required method in the web service implementation.
6. The web service implementation returns 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 Understanding the Client Perspective of an Asynchronous Web Service Call

Before initiating the asynchronous call, the client must deploy a callback service to listen for the response from the asynchronous web service.

The following diagram illustrates the flow, from the client perspective, of the asynchronous method call consisting of two one-way message exchanges.

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 following steps describe the message flow shown in the previous figure:

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 Understanding How Asynchronous Messages Are Correlated

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 runtime.

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

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 About 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 by using JDeveloper.

For complete details of development and deployment of asynchronous web service methods, 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 Annotation to Develop an Asynchronous Web Service

You can use annotations to develop 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')
```

By default, all operations associated with the web service are asynchronous. To mark a specific method as synchronous, use the `@CalbackMethod` annotation, as described in Annotation to Configure the Callback Service. 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 more information about that `@AsyncWebService` and `@PortableWebService` annotation, see Java API Reference for Oracle Infrastructure Web Services.

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

4.4 Creating the Request and Response Queues

You must create and secure the queues used to store the request and response before you deploy your asynchronous web services.

The process of creating and securing the queues is described in the following sections:

- Using the Default WebLogic JMS Queues
- Creating Custom Request and Response Queues
- About Custom Request and Response Queues.

- Best Practices for Creating the Custom Request and Response Queues.
- Modify Request and Response Queues at Runtime
- Securing the Request and Response Queues
- Confirming the Request and Response Queue Configuration

4.4.1 Using the Default WebLogic JMS Queues

The process for using the default WebLogic JMS queues varies, based on whether you are using a clustered or non-clustered domain.

This section includes the following topics:

- Default WebLogic JMS Queues in Non-clustered Domains
- Tuning the Default JMS Delivery Failure Parameters

4.4.1.1 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.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 a WebLogic Domain in the Graphical Mode*Creating WebLogic 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.

4.4.1.2 Tuning the Default JMS Delivery Failure Parameters

The following default values are configured 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 Main Steps for Configuring Basic JMS System Resources in *Administering JMS Resources 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

You have to customize the request and response queues, if the default WebLogic JMS queues do not meet your requirements. Follow these steps to customize your request and response queues:

1. Create the request and response queues manually, as described in Create a JMS Queue or Topic.
2. Add them to your application code using the following annotations:
   - **Request queue**: `oracle.webservices.annotations.async.AsyncWebServiceQueue`

     For example:

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

     For more information about the `@AsyncWebServiceQueue`, see Java API Reference for Oracle Infrastructure Web Services.
   - **Response queue**:
     `oracle.webservices.annotations.async.AsycnWebServiceResponseQueue`

     For example:

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

     For more information about `@AsyncWebServiceResponseQueue`, see Java API Reference for Oracle Infrastructure Web Services.

4.4.3 About Custom Request and Response Queues

You have to customize the request and response queues, if the default WebLogic JMS queues do not meet your requirements.

Asynchronous requests are initially saved in the request JMS queue for asynchronous execution. A Message-driven Bean (MDB) accesses the requests from the queue and executes the business logic.

To process many requests simultaneously, application servers create a pool of MDB instances, which can be configured based on the requirements of the application. The more MDB instances in a pool, the better the throughput. However, more resources, such as threads and database connections, will be used. The exact resource requirements are dependent on the type of persistence store used for saving the asynchronous request messages.

By default WebLogic Server initializes the pool with only one MDB instance and continues to increase the pool size, up to 16, as more requests are queued for execution. Once the maximum pool size is reached, no additional MDB instances are added, and the server waits for the existing MDB instances to complete the currently executing requests.

Users can configure the initial pool size and maximum pool size based on the application requirements. It is recommended that you use the default values, and adjust them based on the resource load observed. If the pool consumes too many resources, the maximum pool size can be reduced. If there are ample resources in the system and too many requests that are waiting in the queue, the maximum pool size can be increased.

4.4.4 Best Practices for Creating the Custom Request and Response Queues

A Message-driven Bean (MDB) accesses the requests from JMS queue for the execution of Asynchronous requests.

The following list provides the best practices for creating the custom request and response queues (Step 1 in "Creating Custom Request and Response Queues"):

- 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.

4.4.5 Modify Request and Response Queues at Runtime

You can modify the request and response queues at runtime using Fusion Middleware Control.

For complete details, see "Configuring Asynchronous Web Services" in *Administering Web Services*.

4.4.6 Securing the Request and Response Queues

Oracle recommends that you secure the JMS request and response queues with a user- or role-based security policy to secure access to these resources.

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

The steps to secure the JMS request and response queues include:

1. Optionally, configure the JMS System User, as described in About Configuring a Custom 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 About the WLST Script for Securing the Request and Response Queues.

4.4.6.1 About Configuring a Custom 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 Java API Reference for Oracle Infrastructure Web Services.

After your application has been deployed, you can change the JMS System User in Fusion Middleware Control and in the WebLogic Server Administration Console as described in "Changing the JMS System User for Asynchronous Web Services" in *Administering Web Services*.

4.4.6.2 About the WLST Script for Securing 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>
```

4.4.7 Confirming the Request and Response Queue Configuration

You have to configure the requests and response queues to meet the requirements.

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

1. Invoke the Administration Console, as described in Invoking the Administration Console in *Understanding WebLogic 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 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:
   a. Click **Deployments**.
   b. Expand the application in the Deployments table.
   c. 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`.
   d. Verify that each MDB displayed is connected to the JMS destination.
   e. Select the EJB and click the **Monitoring** tab and then the **Running** tab.
   f. Verify that the Connection Status field is Connected.
   g. If the queues were not created correctly, perform one of the following:

      Create and configure the required JMS queues manually. For more information, see "Messaging" in *Information Roadmap for Oracle WebLogic Server*.

      or

      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 Using the Default WebLogic JMS Queues.

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

4.5 Annotation to Configure 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. For example: `@CallbackMethod(exclude=true)`

`@CallbackProperties`	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")`
	`}`
	`)`

`@Property`	Specify a property that is required in the message context when calling the callback service.
	For example:
	`@CallbackProperties(`
	`{`
	`@Property(`
	`key = SecurityConstants.ClientConstants.WSS_CSF_KEY,`
	`value = "basic.credentials")`
	`}`
	`)`

`@ResponseWebService`	Customize the response web service port information.
`@Retry`	Specify whether the asynchronous method is idempotent, or retriable, in the event that its execution is terminated abnormally (for example, due to system failure). 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`.

4.6 Configuring SSL for Asynchronous Web Services

There are several tasks that you must perform to configure SSL for asynchronous web services.

The tasks required to be performed are as follows:

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

   See "Obtaining Private Keys, Digital Certificates, and Trusted Certificate Authorities" in *Administering Security for 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 Online Help*.

3. Configure the password for the identity and trust keystores in the OWSM Credential store provider using Fusion Middleware Control.

   See "Configuring the Credential Store" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

   Ensure that the map with the name `oracle.ws.async.ssl.security` exists. If it does not exist, you must create it.

   Then, create the three key entries as listed below:

   - `trust-keystore-password` with user name `async` and password `<trust_store_password>`
   - `identity-keystore-password` with user name `async` and password `<identity_store_password>`
   - `key-password` with user name `async` and password `<key_password>`

4.7 Defining Asynchronous Web Service Clients

Two types of clients which can call asynchronous web services are SOA/BPEL clients and WebLogic Java EE JAX-WS Clients.

- 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 *Developing SOA Applications with 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 Clients" in the *Developing Applications with Oracle JDeveloper*.

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

- Asynchronous Client Code
- 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 Asynchronous Client Code

You can use a client code to support asynchronous web services.

The following example 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.

```
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;

// !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 (12.x.x.x.x, build 090303.0200.48673)

public class HelloServicePortClient
{
  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 Callback Service Code

You can use callback service as a web service by using 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 (12.x.x.x.x, 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_PROPERT
Y);
```

```
    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

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.

> **Note:**
>
> Web services reliable messaging (WS-ReliableMessaging) is not supported for Oracle Infrastructure asynchronous web services. That is, you cannot attach a reliable messaging policy at design time, using the `@ReliabilityPolicy` annotation, or runtime, using Fusion Middleware Control or WLST, to the asynchronous web service or the callback client.

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

- Using annotations at design time.
- Using Fusion Middleware Control or WLST 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, 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" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

You can attach policies to the following asynchronous components:

- About Attaching Policies to Asynchronous Web Service Clients
- Policies to Attach for Asynchronous Callback Services
- About Attaching Policies to Callback Clients
- Policies to Attach for Asynchronous Callback Services

4.8.1 About Attaching Policies to Asynchronous Web Service Clients

Various policies are attached 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 *Developing SOA Applications with 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 Clients" in the *Developing Applications with Oracle JDeveloper*.

At runtime, you can use the Fusion Middleware Control to attach policies to each type of client, as described in "Attaching Policies to Web Services and Clients Using Fusion Middleware Control" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

4.8.2 Policies to Attach for Asynchronous Callback Services

You can use certain annotations to attach policies while sending the asynchronous response to the client's callback service.

At design time, to attach policies while sending the asynchronous response to the client's callback service, you can use one of the annotations defined in Table 4-2.

Table 4-2 Annotations for Attaching Policies to Web Services and Callback Services

Annotation	Description
`@CallbackManagementPolicy`	The `oracle.webservices.annotations.async.CallbackManagementPolicy` annotation attaches a management policy when sending the asynchronous response to the client callback service. For more information, see "@CallbackManagementPolicy" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*. **Note**: This annotation has been deprecated. Oracle recommends that you use the `oracle.wsm.metadata.annotation.CallbackPolicySet` annotation, as described in "@CallbackPolicySet" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
`@CallbackMtomPolicy`	The `oracle.webservices.annotations.async.CallbackMtomPolicy` annotation attaches an MTOM policy when sending the asynchronous response to the client callback service. For more information, see "@CallbackMtomPolicy" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*. **Note**: This annotation has been deprecated. Oracle recommends that you use the `oracle.wsm.metadata.annotation.CallbackPolicySet` annotation, as described in "@CallbackPolicySet" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
`@CallbackPolicySet`	The `oracle.wsm.metadata.annotation.CallbackPolicySet` annotation attaches one or more policy sets to the callback client of the asynchronous web service that will connect to the callback service. By default, no policies are attached. For more information, see "@CallbackPolicySet" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
`@CallbackSecurityPolicy`	The `oracle.webservices.annotations.async.CallbackSecurityPolicy` annotation attaches a security policy to the callback web service. For more information, see "@CallbackSecurityPolicy" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*. **Note**: This annotation has been deprecated. Oracle recommends that you use the `oracle.wsm.metadata.annotation.CallbackPolicySet` annotation, as described in "@CallbackPolicySet" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

Table 4-2 (Cont.) Annotations for Attaching Policies to Web Services and Callback Services

Annotation	Description
`@FastInfosetCallbackClient`	The `com.oracle.webservices.api.FastInfosetCallbackClient` annotation enables and configures Fast Infoset on the callback client of the asynchronous web service that will connect to the callback service. For more information, see "@FastInfosetCallbackClient" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

At runtime, you can use the Fusion Middleware Control to attach policies to asynchronous callback services, as described in "Attaching Policies" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

Use the following guidelines when attaching message protection policies to asynchronous 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 About Attaching Policies to Callback Clients

You can use Fusion Middleware Control at runtime, to attach policies to asynchronous callback clients.

For more information, see "Attaching Policies to Asynchronous Web Service Callback Clients" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

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

5

Introduction to Using Web Services Reliable Messaging

An introduction and description of the usage of Web Services Reliable Messaging (WS-ReliableMessaging) with Oracle Infrastructure web services, to exchange messages reliably is detailed in this chapter.
It includes the following sections:

5.1 Web Services Reliable Messaging

Message exchanges between web services can be disrupted by various errors, including network, system, and software component errors or anomalies. Oracle Infrastructure web services support the messaging protocol defined by the Web Services Reliable Messaging specification to resolve these disruptions.

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/v1.2/wsrm.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 in Oracle Infrastructure Web Services

A set of predefined policies are automatically available when you install oracle fusion middleware and reliable messaging policy is one such type of predefined policy.

Reliable messaging is driven by policies and the Policy framework. As described in Understanding How Policies Attach to Oracle Infrastructure Web Services , OWSM 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 in this pre-defined set:

Table 5-2 Predefined Reliable Messaging Policies

Reliable Messaging Policy	Description
`oracle/no_reliable_messaging_policy`	When directly attached to an endpoint or globally attached at a lower scope, effectively disables a globally attached Web Services Reliable Messaging policy at a higher scope.
`oracle/reliable_messaging_policy`	Configures web services reliable messaging on the web service and client.

For more information about the reliable messaging predefined policies, see "Reliable Messaging Policies" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

5.3 About Attachment of Reliable Messaging Policies to Oracle Infrastructure Web Services

You can attach reliable messaging policies to Oracle Infrastructure web services or clients at design time using Oracle JDeveloper, or at runtime using the Fusion Middleware Control.

For more information, see Understanding How Policies Attach to Oracle Infrastructure Web Services .

5.4 Reliable Messaging Policies Configuration

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

The steps to configure reliable messaging policies are described in "Reliable Messaging Policies" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

6

Introduction to Using Web Services Atomic Transactions

An introduction of usage of the web services atomic transactions to enable interoperability with other external transaction processing systems is detailed in this chapter.
It includes the following sections:

6.1 Overview of Web Services Atomic Transactions Framework

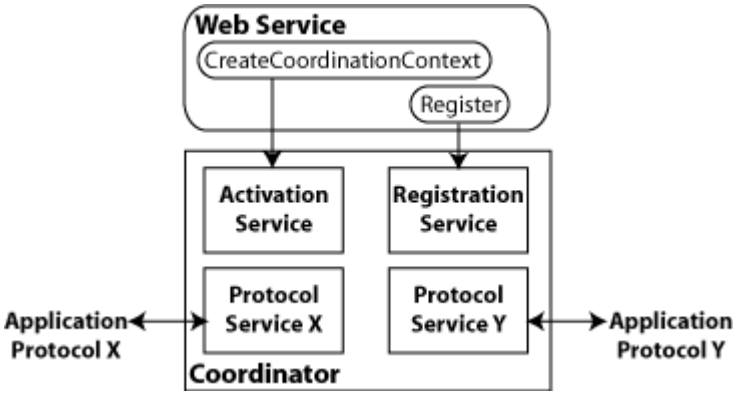
WebLogic web services enable interoperability with other external transaction processing systems, such as Websphere, Microsoft .NET, and so on, through the support of certain specifications.

These specifications are as follows:

- 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

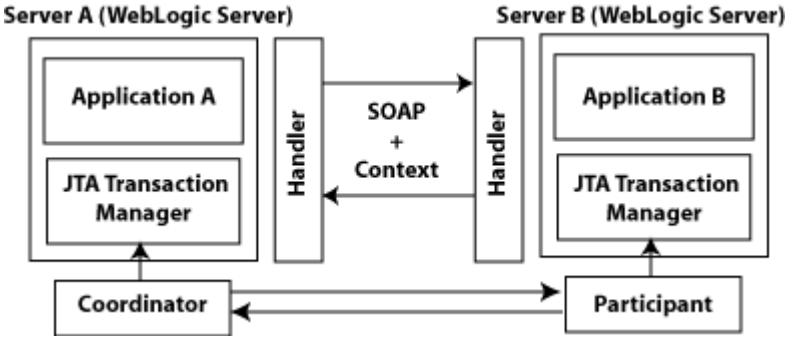


6.2 Overview of Web Services Atomic Transactions in WebLogic Server Environment

WebLogic Server interacts within the context of a web services atomic transaction.

Figure 6-2 shows two instances of such interaction. 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 Java Transaction API and Oracle WebLogic Extensions in *Developing JTA Applications for Oracle WebLogic Server*.

The following steps describe 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.

6.3 Components of Web Services Atomic Transactions

Web services atomic transactions consists of a coordinator, activation service, registration service and application protocol X, Y.

Table 6-1 describes these components in detail.

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.

6.4 How Web Services Atomic Transactions are Enabled on a Web Service (Inbound)

You enable and configure web services atomic transactions on a web service at design time using Oracle JDeveloper when creating a web service, or at deployment time using the Fusion Middleware Control.

For more information, refer to the following sections:

- **Design time**: WS-AtomicTransaction Support in *Developing SOA Applications with Oracle SOA Suite*.
- **Deployment time**: "Configuring Atomic Transactions Using Fusion Middleware Control" in *Administering Web Services*.
- "Configuring Atomic Transactions Using WLST" in *Administering Web Services*.

For information about configuration options, see Web Services Atomic Transaction Configuration.

6.5 How Web Services Atomic Transactions are Enabled on a Web Service Client (Outbound)

You enable and configure web services atomic transactions on a web service client at deployment time using the Fusion Middleware Control.

You configure the version and flow type, as defined in Table 6-3.

For more information, see "Configuring Atomic Transactions Using Fusion Middleware Control" in *Administering Web Services*.

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 Fusion Middleware Control. For more information, refer to the following sections:

- **Design time**: WS-AtomicTransaction Support in *Developing SOA Applications with Oracle SOA Suite*.
- **Deployment time**: "Configuring Atomic Transactions Using Fusion Middleware Control" in *Administering Web Services*.

For information about configuration options, see Web Services Atomic Transaction Configuration.

6.6 Web Services Atomic Transaction Configuration

You can set certain configurations to enable web services atomic transactions.

Table 6-2 summarizes the configuration options that you can set when enabling web services atomic transactions.

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 web service clients, 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 web services is driven by the inbound request and can be any of the values. The `DEFAULT` value for web service clients 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.

> **Note:**
>
> For information about enabling web service atomic transactions, see the following:
>
> - How Web Services Atomic Transactions are Enabled on a Web Service (Inbound)
> - How Web Services Atomic Transactions are Enabled on a Web Service Client (Outbound).

Table 6-3 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 Transaction 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.

6.7 Properties Configured for Messages Exchanged Between the Coordinator and Participant

You can secure messages exchanged between the coordinator and participant using the WebLogic Server Administration Console.

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 Online 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 Online 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

Introduction to Optimizing XML Transmission Using Fast Infoset

An introduction and description of, how to enable and use Fast Infoset to optimize XML transmission for Oracle Infrastructure web services is detailed in this chapter.
It includes the following sections:

7.1 Overview of Fast Infoset

Fast Infoset is a compressed binary encoding format that provides a more efficient serialization than the text-based XML format. Fast Infoset optimizes both document size and processing performance.

When enabled, Fast Infoset converts the XML Information Set in the SOAP envelope into a compressed binary format before transmitting the data. Fast Infoset optimizes encrypted and signed messages, MTOM-enabled messages, and SOAP attachments, and supports both HTTP and JMS transports.

The Fast Infoset capability is enabled on all web services, by default. For web service clients, Fast Infoset is enabled if it is enabled on the web service and advertised in the WSDL.

You can explicitly enable and configure Fast Infoset on a web service or client, as described in the following sections.

7.2 Enabling Fast Infoset on Web Services

The Fast Infoset capability is enabled on a web service and advertised in the WSDL, by default. Follow one of these methods to enable Fast Infoset explicitly on a web service:

- **At design time**, using `com.oracle.webservices.api.FastInfosetService` annotation. For example, following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetService` annotation to enable and configure Fast Infoset on a web service at design time.

```
package examples.webservices.fastinfoset;
import com.oracle.webservcies.api.FastInfosetService;
import oracle.webservices.annotations.PortableWebService;
import javax.jws.WebMethod;

@PortableWebService
@FastInfosetService(enabled = true)
public class HelloImplFastInfosetEnabled {
   @WebMethod
   public String hello(String name) {
      return "Hello, " + name + "! (from FI Enabled Service)";
```

```
    }
}
```

For more information about the annotation, see "@FastInfosetService" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

For more information about the `@PortableWebService` annotation, see *Java API Reference for Oracle Infrastructure Web Services*.

- **Post-deployment**, by attaching the `oracle/fast_infoset_service_policy` to the web service.

  For more information, see the following sections:

  – "Attaching Policies to Web Services and Clients Using Fusion Middleware Control" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

  – "Configuring Fast Infoset Using WLST" in *Administering Web Services*.

7.3 About Enabling and Configuring Fast Infoset on Web Services Clients

The Fast Infoset capability is enabled on a web service and advertised in the WSDL, by default. Follow one of these methods to explicitly enable and configure Fast Infoset on a web service client:

- At design time, using the `com.oracle.webservices.api.FastInfosetClientFeature` feature class, as shown in Using FastInfosetClientFeature Feature Class at Design Time
- Post-deployment, by attaching `oracle/fast_infoset_client_policy` to the web service.

  For more information, see the following sections:

  – "Attaching Policies to Web Services and Clients Using Fusion Middleware Control" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

  – "Configuring Fast Infoset Using WLST" in *Administering Web Services*.

To enable and configure Fast Infoset on the client and on a web service at design time, see the following sections:

- Content Negotiation Strategy
- Using FastInfosetClientFeature Feature Class at Design Time

7.3.1 Content Negotiation Strategy

You can configure the content negotiation policy when enabling Fast Infoset on the client.

Table 7-1 summarizes the valid content negotiation strategies defined by `com.oracle.webservices.api.FastInfosetContentNegotiationType`.

Table 7-1 Content Negotiation Strategy

Value	Description
`OPTIMISTIC`	Assumes that Fast Infoset is enabled on the service. All requests will be sent using Fast Infoset.

Table 7-1 (Cont.) Content Negotiation Strategy

Value	Description
PESSIMISTIC	Initial request from client is sent without Fast Infoset enabled, but with an HTTP Accept header that indicates that the client supports the Fast Infoset capability. If the service response is in Fast Infoset format, confirming that Fast Infoset is enabled on the service, then subsequent requests from the client will be sent in Fast Infoset format.
NONE	Client requests will not use Fast Infoset.

Please note:

- If the content negotiation strategy is configured explicitly on the client:
  - It takes precedence over the negotiation strategy advertised in the WSDL.
  - If the configured content negotiation strategy conflicts with the capabilities advertised by the service (for example, if the client configures `OPTIMISTIC` and the service has Fast Infoset disabled), then an exception is generated.
- If the content negotiation strategy is not configured explicitly by the client:
  - If Fast Infoset is enabled and advertised on the service, the `OPTIMISTIC` content negotiation strategy is used.
  - If Fast Infoset is disabled and not advertised on the service, the `NONE` content negotiation strategy is used.

7.3.2 Using FastInfosetClientFeature Feature Class at Design Time

You can enable and configure Fast Infoset on a web service at design time using the `com.oracle.webservices.api.FastInfosetClientFeature` feature class.

The following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetClientFeature` feature class.

```
package examples.webservices.fastinfoset;
import com.oracle.webservices.api.FastInfosetClientFeature;
import com.oracle.webservices.api.FastInfosetContentNegotiationType;
...
public class HelloServicePortClient {

    private static HelloServiceService helloServiceService;

    public static void main(String [] args)
    {
        FastInfosetContentNegotiationType clientNeg =
            FastInfosetContentNegotiationType.PESSIMISTIC;
        FastInfosetClientFeature feature =
FastInfosetClientFeature.builder().fastInfosetContentNegotiation(clientNeg).enabled(true)
.build();
        helloServiceService = new HelloServiceService();
        HelloService helloService =
            helloServiceService.getHelloServicePort(feature);
...
}
```

7.4 Disabling Fast Infoset on Web Services and Clients

You can explicitly disable fast infoset on web services and clients at the design time.

To disable Fast Infoset explicitly:

- On a web service, set the `enabled` flag to `false` on the annotation, as described in Enabling Fast Infoset on Web Services
- On a web service client, set the `enabled` flag to `false` or set the content negotiation strategy to `NONE` on the annotation or Feature class, as described in About Enabling and Configuring Fast Infoset on Web Services Clients

The following code excerpt provides an example of using the `com.oracle.webservices.api.FastInfosetService` annotation to disable Fast Infoset on a web service at design time.

```
package examples.webservices.fastinfoset;
import com.oracle.webservices.api.FastInfosetService;
...
@PortableWebService
@FastInfosetService(enabled = false)
public class HelloImplFastInfosetDisabled{
   @WebMethod
   public String hello(String name) {
      return "Hello, " + name + "! (from FI Disabled Service)";
   }
}
```

At post-deployment time, to disable Fast Infoset:

- Detach the `oracle/fast_infoset_service_policy` or `oracle/fast_infoset_client_policy` policy from the web service or client, respectively.

  For complete details, see the following sections:

  - "Attaching Policies to Web Services and Clients Using Fusion Middleware Control" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
  - "Configuring Fast Infoset Using WLST" in *Administering Web Services*.
- To disable Fast Infoset globally, at a higher scope on a web service or client, define a policy set that includes `oracle/no_fast_infoset_service_policy` or `oracle/no_fast_infoset_client_policy` policy, respectively.

  For complete details, see the following sections:

  - "Attaching Policies Globally Using Policy Sets" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
  - "Attaching Policies Globally Using Policy Sets Using WLST" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

8

Introduction to Using MTOM Encoded Message Attachments

Using MTOM to pass binary content improves web services performance. OWSM enables you to attach MTOM policies to your web services.
This chapter includes the following sections:

8.1 Overview of Message Transmission Optimization Mechanism

You can use Message Transmission Optimization Mechanism (MTOM), to pass Binary content such as an image in JPEG format between the client and the web service, which reduces the transmission size on the wire.

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/

8.2 About Predefined MTOM Attachment Policies

MTOM attachment policies are part of the pre-defined policy set of OWSM.

As described in Understanding How Policies Attach to Oracle Infrastructure Web Services , OWSM provides a set of predefined policies and assertion templates that are automatically available when you install Oracle Fusion Middleware. The **MTOM attachment** policy listed in Table 8-1 is available in this pre-defined set:

Table 8-1 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 *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

8.3 About MTOM Policies Attachment

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 Understanding How Policies Attach to Oracle Infrastructure Web Services .

8.4 About MTOM Policies Configuration

No configuration steps are required.

9

Introduction to Developing RESTful Web Services

An introduction of Representational State Transfer (RESTful) web service concepts and a description of how to develop and configure RESTful web services using Java API for RESTful Web Services (JAX-RS) is detailed in this chapter.
It included the following sections:

- Overview of RESTful Web Services
- How RESTful Web Services Requests Are Formed and Processed
- Understanding the Limitations of RESTful Web Service Support

9.1 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).

9.2 How RESTful Web Services Requests Are Formed and Processed

A description of how RESTful web service requests are formed on the client side and how they are processed on the server side is detailed in these sections:

- HTTP Get Requests
- HTTP Post Requests
- RESTful Responses

9.2.1 HTTP Get Requests

RESTful web services support HTTP Get Requests. You can invoke RESTful requests by using a GET with a specific URL.

This section describes how to build HTTP Get Requests. It includes the following sections:

- About HTTP Get Requests
- Building HTTP Get Requests

9.2.1.1 About 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.

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

9.2.1.2 Building HTTP Get Requests

You can invoke RESTful requests 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.

```
<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.

```
<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.

```
<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 addNumbers operation example.

9.2.2 HTTP Post Requests

RESTful web services support HTTP POST Requests that are simple XML messages and not SOAP envelopes.

This section describes how to build HTTP POST Requests. It includes the following sections:

- About the HTTP Post Requests
- Building HTTP Post Requests

9.2.2.1 About the HTTP Post Requests

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.

9.2.2.2 Building HTTP Post Requests

You can process RESTful requests 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, this RESTful request will be wrapped in the SOAP envelope illustrated the following 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.

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

9.2.3 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.

```
<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.

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

```
</ns0:result>
```

9.3 Understanding the Limitations of RESTful Web Service Support

Oracle Web Services support for RESTful web services is constrained by certain limitations.

The following list describes the limitations.

- 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.

10

Invoking a Web Service from a Standalone Client

A description of how to invoke an Oracle Infrastructure web service from a standalone client is detailed in this chapter.
It includes the following sections:

- Using a Standalone Client Jar to Invoke a Web Service
- Supporting Basic Authentication
- Supporting SSL Policies

10.1 Using a Standalone Client Jar to Invoke a Web Service

When invoking an Oracle Infrastructure web service from an environment that does not have Oracle Fusion Middleware installed locally, with the entire set of Oracle Fusion Middleware classes in the CLASSPATH, you can use the standalone client JAR file to invoke the web service.

The standalone client JAR supports basic Oracle Infrastructure web service client-side functionality and OWSM security policies.

To use the standalone client JAR file with your client application, perform the following steps:

1. Create a Java SE client using your favorite IDE, such as Oracle JDeveloper.
2. Copy the file *ORACLE_HOME*`/oracle_common/modules/clients/com.oracle.webservices.fmw.client_12.1.3.jar` from the computer hosting Oracle Fusion Middleware to the client computer, where *ORACLE_HOME* is the directory you specified as Oracle Home when you installed Oracle Fusion Middleware.

   For example, you might copy the file into the directory that contains other classes used by your client application.
3. Add the JAR file to your CLASSPATH.
4. Configure your environment for Oracle Web Services Manager (OWSM) policies. This step is optional, required only if you are attaching OWSM security policies to the web service client.

   The configuration steps required vary based on the type of policy being attached. See the following examples:

   - Supporting Basic Authentication
   - Supporting SSL Policies

> **Note:**
>
> For additional configuration requirements, see Configuring Java SE Applications to Use OPSS in *Securing Applications with Oracle Platform Security Services*.

10.2 Supporting Basic Authentication

You have to perform certain steps to support basic authentication when you invoke an Oracle Infrastructure web service from a standalone client.

To support basic authentication using the `oracle/wss_http_token_client_policy` security policy, perform the following steps:

1. Copy the `jps-config-jse.xml` and `audit-store.xml` files from the *`domain_home`*`/config/fmwconfig` directory, where *`domain_home`* is the name and location of the domain, to a location that is accessible to the web service client.
2. Create a wallet (`cwallet.sso`) in the same location that you copied the files in step a that defines a map called `oracle.wsm.security` and the credential key name that the client application will use (for example, `weblogic-csf-key`).

   The location of the file `cwallet.sso` is specified in the configuration file `jps-config-jse.xml` with the element `<serviceInstance>`.
3. On the Java command line, pass the following property defining the JPS configuration file copied in step 1:

   ```
   -Doracle.security.jps.config=<pathToConfigFile>
   ```

10.3 Supporting SSL Policies

You have to perform certain steps to support SSL policies when you invoke an Oracle Infrastructure web service from a standalone client.

To support SSL policies, perform the following steps:

1. Copy the `jps-config-jse.xml` and `audit-store.xml` files from the *`domain_home`*`/config/fmwconfig` directory, where *`domain_home`* is the name and location of the domain, to a location that is accessible to the web service client.
2. On the Java command line, pass the following properties, defining the JPS configuration file copied in step 1:

   Define the JPS configuration file copied in step 1:

   ```
   -Doracle.security.jps.config=<pathToConfigFile>
   ```

   Define the trust store containing the trusted certificates:

   ```
   -Djavax.net.ssl.trustStore=<trustStore>
   ```

   Define the trust store password:

   ```
   -Djavax.net.ssl.trustStorePassword=<password>
   ```

11

About Testing Web Services

You can use the Web Services Test Client or the Fusion Middleware Control Test Web Service page to test basic and advanced features of your Oracle Infrastructure web services, such as authentication, authorization, quality of service (QoS), HTTP header options, and so on. You can also perform stress testing of the security features.

For information about testing web services using the Web Services Test Client or Fusion Middleware Control Test Web Service page, see "Testing Web Services" in *Administering Web Services*.

12

Interoperability Guidelines

An introduction of the guidelines for ensuring interoperability with Oracle Infrastructure web services is detailed in this chapter.
It includes the following sections:

- Introduction to Web Service Interoperability
- Web Service Interoperability Organizations
- Recommended Guidelines for Creating Interoperable Web Services

12.1 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.

12.2 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.

12.2.1 About the 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://java.sun.com/webservices/interop/soapbuilders/index.html`

12.2.2 About the WS-Interoperability Organization

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.

12.3 Recommended Guidelines for Creating Interoperable Web Services

You have to follow certain guidelines to create interoperable web services. The first general guideline is to create web services that 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.

For this reason, Oracle recommends that you follow these general guidelines when creating interoperable web services.

12.3.1 Why Design Web Services Using a Top Down Approach?

Using a 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.

12.3.2 About Designing 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.

12.3.3 Keeping Data Types Simple

You should avoid unnecessarily complex schema data types such as `xsd:choice`.

Simple data types provide the best interoperability and have the added benefit of improved performance.

For additional information, see the following:

12.3.3.1 Why 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.

12.3.3.2 Why Differentiate Between Empty Arrays and Null References to Arrays?

To prevent issues, be sure to differentiate between empty arrays and null references to arrays.

For example, if you have an array with attributes `minoccurs=0` and `xsd:nillable=true` 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.

12.3.3.3 Why Avoid Using 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.

12.3.3.4 Why 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.

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

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

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

12.3.3.5 Why Map Any Unsupported xsd:types to a 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.

12.3.4 About Using Null Values With Care

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.

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?

12.3.5 About Using 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/`

12.3.6 Why Consider 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.

12.3.7 Why 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.

12.3.8 Understanding How to 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.

12.3.9 Why 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.

12.3.10 Why 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.