Oracle® Fusion Middleware

Using the Jersey JAX-RS Reference Implementation

11*g* Release 1 (10.3.6)

E41958-02

April 2015

Documentation for software developers that describes how to use the Jersey JAX-RS Reference Implementation (RI) with Oracle Fusion Middleware 11*g*.

ORACLE

Oracle Fusion Middleware Using the Jersey JAX-RS Reference Implementation, 11*g* Release 1 (10.3.6)

E41958-02

Contents

Preface

This preface describes the document accessibility features and conventions used in this guide—*Using the Jersey JAX-RS Reference Implementation*.

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.

Conventions

The following text conventions are used in this document:

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

1

Using the Jersey JAX-RS Reference Implementation

This document describes how to use the Jersey JAX-RS Reference Implementation (RI) with Oracle Fusion Middleware 11*g*. Sections include:

- Introduction to the REST Architectural Style
- What are RESTful Web Services?
- Developing RESTful Web Service on WebLogic Server
- Summary of the Jersey JAX-RS RI Shared Libraries
- Steps to Use the Jersey JAX-RS RI Shared Libraries
- Registering the Jersey JAX-RS RI Shared Libraries With Your WebLogic Server Instances
- Configuring the Web Application to Use the Jersey JAX-RS RI
- Creating RESTful Web Services and Clients
- Securing the Jersey Servlet Application
- Securing RESTful Web Service Clients
- Registering a More Recent Version of the Jersey JAX-RS RI

1.1 Introduction to the REST Architectural Style

REST describes any simple interface that transmits data over a standardized interface (such as HTTP) without an additional messaging layer, such as Simple Object Access Protocol (SOAP). REST is an *architectural style*—not a toolkit—that provides a set of design rules for creating stateless services that are viewed as *resources*, or sources of specific information (data and functionality). Each resource can be identified by its unique Uniform Resource Identifiers (URIs).

A client accesses a resource using the URI and a standardized fixed set of methods, and a *representation* of the resource is returned. A representation of a resource is typically a document that captures the current or intended state of a resource. The client is said to *transfer* state with each new resource representation.

Table 1–1 defines a set of constraints defined by the REST architectural style that must be adhered to in order for an application to be considered "RESTful."

Table 1–1 Constraints of the REST Architectural Style

Constraint	Description
Addressability	Identifies all resources using a uniform resource identifier (URI). In the English language, URIs would be the equivalent of a *noun*.
Uniform interface	Enables the access of a resource using a uniform interface, such as HTTP methods (GET, POST, PUT, and DELETE). Applying the English language analogy, these methods would be considered *verbs*, describing the actions that are applicable to the named resource.
Client-server architecture	Separates clients and servers into interface requirements and data storage requirements. This architecture improves portability of the user interface across multiple platforms and scalability by simplifying server components.
Stateless interaction	Uses a stateless communication protocol, typically Hypertext Transport Protocol (HTTP). All requests must contain all of the information required for a particular request. Session state is stored on the client only. This interactive style improves: ■ Visibility—Single request provides the full details of the request. ■ Reliability—Eases recovery from partial failures. ■ Scalability—Not having to store state enables the server to free resources quickly.
Cacheable	Enables the caching of client responses. Responses must be identified as cacheable or non-cacheable. Caching eliminates some interactions, improving efficiency, scalability, and perceived performance.
Layered system	Enables client to connect to an intermediary server rather than directly to the end server (without the client's knowledge). Use of intermediary servers improve system scalability by offering load balancing and shared caching.

1.2 What are RESTful Web Services?

RESTful web services are services that are built according to REST principles and, as such, are designed to work well on the Web.

RESTful web services conform to the architectural style constraints defined in Table 1–1. Typically, RESTful web services are built on the HTTP protocol and implement operations that map to the common HTTP methods, such as GET, POST, PUT, and DELETE to create, retrieve, update, and delete resources, respectively.

1.3 Developing RESTful Web Service on WebLogic Server

WebLogic Server ships with a set of pre-built shared libraries, packaged as Web applications, that are required to run applications that are based on the Jersey JAX-RS RI. The following versions are supported:

- Jersey JAX-RS RI Version 1.9
- Jersey JAX-RS RI Version 1.1.5.1

The following sections summarize the Jersey JAX-RS RI shared libraries and the steps to use them, and how to register a more recent version of the Jersey JAX-RS RI.

1.4 Summary of the Jersey JAX-RS RI Shared Libraries

The shared libraries are located in the following directory: *WL_HOME*`/common/deployable-libraries`.

Table 1–2 lists the pre-built shared library that supports Jersey JAX-RS RI Version 1.9 Web services.

Table 1–2 Shared Library for Jersey JAX-RS RI 1.9

Functionality	Description
■ Jersey ■ JSON processing and streaming ■ ATOM processing	■ Shared Library Name: jersey-bundle ■ JAR Filename: jersey-bundle-1.9.jar ■ WAR Filename: jersey-bundle-1.9.war ■ Version: 1.9 ■ License: SUN CDDL+GPL
JAX-RS API	■ Shared Library Name: jsr311 ■ JAR Filename: jsr311-api-1.1.1.jar ■ WAR Filename: jsr311-api-1.1.1.war ■ Version: 1.1.1 ■ License: JSR311 license

Table 1–3 summarizes the pre-built shared libraries that support Jersey JAX-RS RI Version 1.1.5.1 Web services, organized by the functionality that they support. The table also indicates whether the shared library is required or optional.

Table 1–3 Shared Libraries for Jersey JAX-RS RI 1.1.5.1

Functionality	Description	Required?
Jersey	■ Shared Library Name: jersey-bundle ■ JAR Filename: jersey-bundle-1.1.5.1.jar ■ WAR Filename: jersey-bundle-1.1.5.1.war ■ Version: 1.1.5.1 ■ License: SUN CDDL+GPL	Required
JAX-RS API	■ Shared Library Name: jsr311 ■ JAR Filename: jsr311-api-1.1.1.jar ■ WAR Filename: jsr311-api-1.1.1.war ■ Version: 1.1.1 ■ License: JSR311 license	Required
JSON processing	■ Shared Library Name: jackson-core-asl ■ JAR Filename: jackson-core-asl-1.1.1.jar ■ WAR Filename: jackson-core-asl-1.1.1.war ■ Version: 1.1.1 ■ License: Apache 2.0	Optional
JSON processing	■ Shared Library Name: jackson-jaxrs ■ JAR Filename: jackson-jaxrs-1.1.1.jar ■ WAR Filename: jackson-jaxrs-1.1.1.war ■ Version: 1.1.1 ■ License: Apache 2.0	Optional

Table 1–3 (Cont.) Shared Libraries for Jersey JAX-RS RI 1.1.5.1

Functionality	Description	Required?
JSON processing	■ Shared Library Name: jackson-mapper-asl ■ JAR Filename: jackson-mapper-asl-1.1.1.jar ■ WAR Filename: jackson-mapper-asl-1.1.1.war ■ Version: 1.1.1 ■ License: Apache 2.0	Optional
JSON streaming	■ Shared Library Name: jettison ■ JAR Filename: jettison-1.1.jar ■ WAR Filename: jettison-1.1.war ■ Version: 1.1 ■ License: Apache 2.0	Optional
ATOM processing	■ Shared Library Name: rome ■ JAR Filename: rome-1.0.jar ■ WAR Filename: rome-1.0.war ■ Version: 1.0 ■ License: Apache 2.0	Optional

In addition, the following table lists the dependent JARs that are available on WebLogic Server, and not required to be registered as shared libraries.

Table 1–4 Dependent JARs (Available on WebLogic Server)

Functionality	JAR Filename
jdom Version 1.0 API for ATOM processing	com.bea.core.jdom_1.0.0.0_1-0.jar
JAXB Version 2.1.1 API	javax.xml.bind_2.1.1.jar
Servlet Version 2.5 API	Javax.servlet_1.0.0.0_2-5.jar

1.5 Steps to Use the Jersey JAX-RS RI Shared Libraries

To use the Jersey JAX-RS RI, perform the following steps:

1. Register the Jersey JAX-RS RI shared libraries with one or more WebLogic Server instances. See "Registering the Jersey JAX-RS RI Shared Libraries With Your WebLogic Server Instances" on page 5.
2. Configure the Web application that contains the RESTful Web service to use the Jersey JAX-RS RI shared libraries. See "Configuring the Web Application to Use the Jersey JAX-RS RI" on page 6.
3. Create the JAX-RS Web services and clients. See "Creating RESTful Web Services and Clients" on page 8.
4. Optionally, secure the Jersey servlet application or RESTful Web service client using Oracle Web Services Manager (Oracle WSM) policies. For more information, see:
   - "Securing the Jersey Servlet Application" on page 9
   - "Securing RESTful Web Service Clients" on page 11

As required, you can build and deploy a more recent version of the Jersey JAX-RS RI shared libraries. See "Registering a More Recent Version of the Jersey JAX-RS RI" on page 16.

For more information about the Jersey JAX-RS RI and examples of developing RESTful Web services, see `https://jersey.java.net`.

1.6 Registering the Jersey JAX-RS RI Shared Libraries With Your WebLogic Server Instances

Shared Java EE libraries are registered with one or more WebLogic Server instances by deploying them to the target servers and indicating that the deployments are to be shared. Shared Java EE libraries must be targeted to the same WebLogic Server instances you want to deploy applications that reference the libraries.

When a referencing application is deployed, WebLogic Server merges the shared library files with the application. If you try to deploy a referencing application to a server instance that has not registered a required library, deployment of the referencing application fails.

Based on the functionality required by your application and the version of the Jersey JAX-RS RI that you want to use, you can register one or more of the Jersey JAX-RS shared libraries defined in "Summary of the Jersey JAX-RS RI Shared Libraries" on page 2, as follows:

1. Choose whether you want to use Version 1.9 or 1.1.5.1 of the Jersey JAX-RS RI. Based on the version you choose, refer to Table 1–2 or Table 1–3, respectively, to determine the shared libraries that are required by your application.
2. Determine the WebLogic Server targets to which you will register the shared libraries. Shared libraries must be registered to the same WebLogic Server instances on which you plan to deploy referencing applications. (You may consider deploying libraries to all servers in a domain, so that you can later deploy referencing applications as needed.)
3. Register a shared library by deploying the shared library files to the target servers identified in Step 2, and identifying the deployment as a library using the `-library` option.

   The following shows an example of how to deploy the shared libraries that provide support for Jersey JAX-RS RI Version 1.9 functionality and JAX-RS API.

   ```
   weblogic.Deployer -verbose -noexit -source C:\myinstall\wlserver_
   10.3\common\deployable-libraries\jersey-bundle-1.9.war -targets myserver
   -adminurl t3://localhost:7001 -user system -password ******** -deploy -library

   weblogic.Deployer -verbose -noexit -source C:\myinstall\wlserver_
   10.3\common\deployable-libraries\jsr311-api-1.1.1.war -targets myserver
   -adminurl t3://localhost:7001 -user system -password ******** -deploy -library
   ```

   If you wish to use Jersey JAX-RS RI Version 1.1.5.1, the following shows an example of how to deploy the shared libraries that provide support for the basic Jersey JAX-RS RI functionality and JAX-RS API.

   ```
   weblogic.Deployer -verbose -noexit -source C:\myinstall\wlserver_
   10.3\common\deployable-libraries\jersey-bundle-1.1.5.1.war -targets myserver
   -adminurl t3://localhost:7001 -user system -password ******** -deploy -library

   weblogic.Deployer -verbose -noexit -source C:\myinstall\wlserver_
   10.3\common\deployable-libraries\jsr311-api-1.1.1.war -targets myserver
   ```

```
-adminurl t3://localhost:7001 -user system -password ******** -deploy -library
```

For more information about the weblogic.Deployer, see "weblogic.Deployer Command-Line Reference" in *Deploying Applications to Oracle WebLogic Server*.

1.7 Configuring the Web Application to Use the Jersey JAX-RS RI

You need to configure the Web application that contains the RESTful Web services to use the Jersey shared libraries. Specifically, you need to update the following two deployment descriptor files that are associated with your application:

- `web.xml`—Update to delegate Web requests to the Jersey servlet. See "Updating web.xml to Delegate Web Requests to the Jersey Servlet" on page 6.
- `weblogic.xml`—Update to reference the shared libraries from Table 1–3 that are required by your application. See "Updating weblogic.xml to Reference the Shared Libraries" on page 7.

1.7.1 Updating web.xml to Delegate Web Requests to the Jersey Servlet

Update the `web.xml` file to delegate all Web requests to the Jersey Servlet, `com.sun.jersey.spi.container.servlet.ServletContainer`. The `web.xml` file is located in the `WEB-INF` directory in the root directory of your application archive.

The following provides an example of how to update the `web.xml` file:

```
<web-app>
    <servlet>
        <display-name>My Jersey Application</display-name>
        <servlet-name>MyJerseyApp</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>myPackage.myJerseyApplication</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyJerseyApp</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

As shown in the previous example, you need to define the following elements:

- `<servlet-class>` element defines the servlet that is the entry point into the Jersey JAX-RS RI. This value should always be set to `com.sun.jersey.spi.container.servlet.ServletContainer`.
- `<init-param>` element defines the class that extends the `javax.ws.rs.core.Application`, as describe in "An Application Subclass" on page 9.
- `<servlet-mapping>` element defines the base URL pattern that gets mapped to the `MyJerseyApp` servlet. The portion of the URL after the `http://<host>:<port>+<webAppName>` is compared to the `<url-pattern>` by WebLogic Server. If the patterns match, the servlet mapped in this element will be called.

For more information about the `web.xml` deployment descriptor, see "web.xml Deployment Descriptor Elements" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

1.7.2 Updating weblogic.xml to Reference the Shared Libraries

Update the `weblogic.xml` file to reference the shared libraries that are required by your application. The `weblogic.xml` file is located in the `WEB-INF` directory in the root directory of your application archive.

The `<exact-match>` directive enables you to control whether the latest version of the shared libraries that are deployed will be used. If set to true, then the version specified in the `weblogic.xml` will be used, regardless of whether a newer version has been deployed to WebLogic Server. If set to false, then the latest version deployed to WebLogic Server will be used, regardless of what is specified in the weblogic.xml file.

For example, if you set the `<exact-match>` directive to false and register as a shared library a more recent version of the Jersey software, as described in "Registering a More Recent Version of the Jersey JAX-RS RI" on page 16, then the more recent version of the shared library will be used by your application automatically; you do not have to edit the `weblogic.xml` file in this case to pick up the latest version.

The following example shows how to update the `weblogic.xml` file to use the Jersey JAX-RS RI Version 1.9.

```
<library-ref>
    <library-name>jax-rs</library-name>
    <specification-version>1.1</specification-version>
    <implementation-version>1.9</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
```

The following example shows how to update the `weblogic.xml` file to use the Jersey JAX-RS RI Version 1.1.5.1. Not all shared library references will be required for every Web application; the jersey-bundle and jsr311 shared libraries are both required to use the Jersey JAX-RS RI. In this example, `<exact-match>` is set to false specifying that the latest version of the shared library deployed to WebLogic Server should be used.

```
<library-ref>
    <library-name>jersey-bundle</library-name>
    <specification-version>1.1.1</specification-version>
    <implementation-version>1.1.5.1</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
<library-ref>
    <library-name>jsr311</library-name>
    <specification-version>1.1.1</specification-version>
    <implementation-version>1.1.1</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
<library-ref>
    <library-name>jackson-core-asl</library-name>
    <specification-version>1.0</specification-version>
    <implementation-version>1.1.1</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
<library-ref>
    <library-name>jettison</library-name>
    <specification-version>1.1</specification-version>
    <implementation-version>1.1</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
<library-ref>
    <library-name>rome</library-name>
    <specification-version>1.0</specification-version>
```

```
    <implementation-version>1.0</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
```

For more information about the `weblogic.xml` deployment descriptor, see "weblogic.xml Deployment Descriptor Elements" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

1.8 Creating RESTful Web Services and Clients

After you have registered the Jersey JAX-RS RI and configured your Web application, you can start creating RESTful Web services and clients, and an Application subclass, as required by your deployment. The following sections show a simple Web service and client.

1.8.1 A Simple RESTful Web Service

The following provides a very simple example of a RESTful Web service:

```
package samples.helloworld;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

// Specifies the path to the RESTful service
@Path("/helloworld")
public class helloWorld {

   // Specifies that the method processes HTTP GET requests
   @GET
   @Path("sayHello")
   @Produces("text/plain")
   public String sayHello() {
      return "Hello World!";
   }
}
```

1.8.2 A Simple RESTful Client

The following provides a simple RESTful client that calls the RESTful Web service defined previously. This sample uses classes that are provided by the Jersey JAX-RS RI specifically; they are not part of the JAX-RS standard.

```
package samples.helloworld.client;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;

public class helloWorldClient {
   public helloWorldClient() {
      super();
   }

   public static void main(String[] args) {
      Client c = Client.create();
      WebResource resource =
c.resource("http://localhost:7101/RESTfulService-Project1-context-root/jersey/hell
```

```
oWorld");
      String response = resource.get(String.class);
   }
}
```

1.8.2.1 An Application Subclass

The following example shows how to create a class that extends `javax.ws.rs.core.Application` to define the components of a RESTful Web service application deployment and provides additional metadata. For more information about `javax.ws.rs.core.Application`, see the Javadoc at https://jersey.java.net/apidocs/1.9/jersey/javax/ws/rs/core/Application.html.

Within the `Application` subclass, override the `getClasses()` and `getSingletons()` methods, as required, to return the list of RESTful Web service resources. A resource is bound to the `Application` subclass that returns it.

Note that an error is returned if both methods return the same resource.

Use the `javax.ws.rs.ApplicationPath` annotation to defined the base URI pattern that gets mapped to the servlet. For more information about the `@ApplicationPath` annotation, see the Javadoc at: https://jersey.java.net/apidocs/1.9/jersey/javax/ws/rs/core/Application.html.

The following provides an example of a class that extends `javax.ws.rs.core.Application` and uses the `@ApplicationPath` annotation to define the base URI of the resource.

```
import javax.ws.rs.core.Application;
javax.ws.rs.ApplicationPath;
...
@ApplicationPath("resources")
public class MyApplication extends Application {
    public Set<java.lang.Class<?>> getClasses() {
        Set<java.lang.Class<?>> s = new HashSet<Class<?>>();
        s.add(HelloWorldResource.class);
        return s;
    }
}
```

1.9 Securing the Jersey Servlet Application

To secure the Jersey servlet application, you can attach one or more of the following Oracle WSM policies. For more information about these policies and how to manually configure them, see "Predefined Policies" in *Security and Administrator's Guide for Web Services*.

Authentication Policies

- `oracle/wss_http_token_service_policy`
- `oracle/http_basic_auth_over_ssl_service_policy`
- `oracle/http_jwt_token_service_policy`
- `oracle/http_jwt_token_over_ssl_service_policy`
- `oracle/http_oam_token_service_policy`

- oracle/http_saml20_token_bearer_service_policy
- oracle/http_saml20_token_bearer_over_ssl_service_policy
- oracle/multi_token_rest_service_policy (exactly-one policy)
- oracle/multi_token_over_ssl_rest_service_policy (exactly-one policy)

> **Note:** You can also attach a SPNEGO token policy that you create using the oracle/http_spnego_token_service_template assertion template. For more information, see "Configuring Kerberos With SPNEGO Negotiation" in *Security and Administrator's Guide for Web Services*.

Authorization Policies

- oracle/binding_authorization_denyall_policy
- oracle/binding_authorization_permitall_policy
- oracle/binding_permission_authorization_policy

To secure the Jersey servlet application using Oracle WSM policies, update the web.xml file to secure the Jersey servlet application using the procedures described in "Attaching Policies to Servlet Applications" in *Security and Administrator's Guide for Web Services*.

The following provides an example of how to update the web.xml file:

Example 1–1 Example of web.xml File to Attach Policies to the Jersey Servlet Application

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5" xmlns="http://java.sun.com/xml/ns/javaee">
   <filter>
      <filter-name>OWSM Security Filter</filter-name>
      <filter-class>oracle.wsm.agent.handler.servlet.SecurityFilter</filter-class>
      <init-param>
         <param-name>servlet-name</param-name>
         <param-value>MyJerseyApp</param-value>
      </init-param>
      <init-param>
         <param-name>oracle.wsm.metadata.policySet</param-name>
         <param-value><![CDATA[<sca11:policySet name="policySet"
               appliesTo="REST-Resource()"
               attachTo="Service('*')"
               xmlns:sca11="http://docs.oasis-open.org/ns/opencsa/sca/200903"
               xmlns:orawsp="http://schemas.oracle.com/ws/2006/01/policy"
               xmlns:wsp15="http://www.w3.org/ns/ws-policy">
               <wsp15:PolicyReference
                  URI="oracle/multi_token_rest_service_policy"
                  orawsp:category="security" orawsp:status="enabled">
               </wsp15:PolicyReference>
               <wsp15:PolicyReference
                  URI="oracle/binding_authorization_permitall_policy"
                  orawsp:category="security" orawsp:status="enabled">
               </wsp15:PolicyReference>
            </sca11:policySet>]]>
         </param-value>
```

```
      </init-param>
   </filter>
   <filter-mapping>
      <filter-name>OWSM Security Filter</filter-name>
      <servlet-name>MyJerseyApp</servlet-name>
   </filter-mapping>
    <servlet>
        <display-name>My Jersey Application</display-name>
        <servlet-name>MyJerseyApp</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>myPackage.myJerseyApplication</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyJerseyApp</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

1.10 Securing RESTful Web Service Clients

To secure RESTful Web service clients, you can attach one or more of the following Oracle WSM policies. For more information about these policies and how to manually configure them, see "Predefined Policies" in *Security and Administrator's Guide for Web Services*.

Authentication Policies

- `oracle/wss_http_token_client_policy`
- `oracle/http_basic_auth_over_ssl_client_policy`
- `oracle/http_jwt_token_client_policy`
- `oracle/http_jwt_token_over_ssl_client_policy`
- `oracle/http_saml20_token_bearer_client_policy`
- `oracle/http_saml20_token_bearer_over_ssl_client_policy`

> **Note:** You can also attach a SPNEGO token policy that you create using the `oracle/http_spnego_token_client_template` assertion template. For more information, see "Configuring Kerberos With SPNEGO Negotiation" in *Security and Administrator's Guide for Web Services*.

To secure RESTful Web services using Oracle WSM policies:

1. Register the shared libraries required by the Oracle WSM RESTful client filter with your WebLogic Server instances, as described in "Registering the Shared Libraries Required by the Oracle WSM RESTful Client Filter With Your WebLogic Server Instances" on page 12.
2. Update the `weblogic.xml` deployment descriptor to reference the shared libraries required by the Oracle WSM RESTful client filter, as described in "Configuring the Web Application to Use the Oracle WSM RESTful Client Filter" on page 13.

3. Attach Oracle WSM policies to your RESTful Web service clients using one of the following methods:

   - Directly, using Feature classes, as described in "Attaching Policies to RESTful Web Service Clients Using Feature Classes" on page 14.
   - Globally using the `rest-client` resource type, as described in "Creating and Managing Policy Sets" in *Security and Administrator's Guide for Web Services*. For example:

```
C:\Oracle\Middleware\oracle_common\common\bin> wlst.cmd
...
wls:/offline>
connect("weblogic","password","t3://myAdminServer.example.com:7001")
Connecting to t3://myAdminServer.example.com:7001" with userid weblogic
...
Successfully connected to Admin Server "AdminServer" that belongs to
domain "my_domain".

Warning: An insecure protocol was used to connect to the
server. To ensure on-the-wire security, the SSL port or
Admin port should be used instead.

wls:/my_domain/serverConfig> beginRepositorySession()

Session started for modification.

wls:/my_domain/serverConfig> createPolicySet('myPolicySet','rest-client',
'Domain("*")')

Description defaulted to "Global policy attachments for REST clients."
The policy set was created successfully in the session.

wls:/my_domain/serverConfig> attachPolicySetPolicy('oracle/wss_http_token_
client_policy')

Policy reference "oracle/wss_http_token_service_policy" added.

wls:/my_domain/serverConfig> commitRepositorySession()

The policy set myPolicySet is valid.
Creating policy set myPolicySet in repository.

Session committed successfully.
```

In the event you experience problems using the Oracle WSM RESTful client filter, see "Diagnosing Problems with the Oracle WSM RESTful Client Filter" in *Security and Administrator's Guide for Web Services*.

1.10.1 Registering the Shared Libraries Required by the Oracle WSM RESTful Client Filter With Your WebLogic Server Instances

Note: You must register the Jersey JAX-RS RI shared libraries, as well, as described in "Registering the Jersey JAX-RS RI Shared Libraries With Your WebLogic Server Instances" on page 1-5.

The shared libraries required by the Oracle WSM RESTful client filter (in addition to the Jersey JAX-RS RI shared libraries) include:

- *MW_HOME*/oracle_common/modules/oracle.wsm.common_11.1.1/wsm-rest-lib.war
- *MW_HOME*/oracle_common/modules/oracle.webservices_11.1.1/wls-rest-client.war

For Java EE RESTful clients, to register the shared libraries required by Oracle WSM RESTful client filter with your WebLogic Server instances:

1. Determine the WebLogic Server targets to which you will register the shared libraries. A shared library must be registered to the same WebLogic Server instances on which you plan to deploy referencing applications. (You may consider deploying libraries to all servers in a domain, so that you can later deploy referencing applications as needed.)
2. Register the shared libraries by deploying the shared library files to the target servers identified in Step 1, and identifying the deployment as a library using the `-library` option.

   For example:

```
weblogic.Deployer -verbose -noexit -source C:\myinstall\oracle_
common\modules\oracle.wsm.common_11.1.1\wsm-rest-lib.war -targets myserver
-adminurl t3://localhost:7001 -user system -password ******** -deploy -library

weblogic.Deployer -verbose -noexit -source C:\myinstall\oracle_
common\modules\oracle.webservices_11.1.1\wls-rest-client.war -targets myserver
-adminurl t3://localhost:7001 -user system -password ******** -deploy -library
```

For Java SE RESTful clients, to register the shared libraries required by your Oracle WSM RESTful client filter with your WebLogic Server instances, you must add the shared libraries to the classpath.

1.10.2 Configuring the Web Application to Use the Oracle WSM RESTful Client Filter

> **Note:** You must configure the Web application to use the Jersey JAX-RS RI, as well, as described in "Configuring the Web Application to Use the Jersey JAX-RS RI" on page 1-6.

Update the `weblogic.xml` file to reference the shared libraries required by the Oracle WSM RESTful client filter. The `weblogic.xml` file is located in the `WEB-INF` directory in the root directory of your application archive.

The following example shows how to update the `weblogic.xml` file to use the Oracle WSM RESTful client filter.

> **Note:** In the following example, the Web application is configured to use the Jersey JAX-RS RI Version 1.9 shared library. For more information about configuring the Jersey JAX-RS RI version, see "Configuring the Web Application to Use the Jersey JAX-RS RI" on page 1-6.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<weblogic-web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
 xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-web-app
http://www.bea.com/ns/weblogic/weblogic-web-app/1.0/weblogic-web-app.xsd"
xmlns="http://www.bea.com/ns/weblogic/weblogic-web-app">
...
<library-ref>
    <library-name>jax-rs</library-name>
    <specification-version>1.1</specification-version>
    <implementation-version>1.9</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
<library-ref>
    <library-name>wsm-rest-lib</library-name>
    <exact-match>false</exact-match>
</library-ref>
<library-ref>
    <library-name>wls-rest-client</library-name>
    <specification-version>1.1</specification-version>
    <implementation-version>1.1.0.0</implementation-version>
    <exact-match>false</exact-match>
</library-ref>
...
</weblogic-web-app>
```

For more information about the `weblogic.xml` deployment descriptor, see "weblogic.xml Deployment Descriptor Elements" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

1.10.3 Attaching Policies to RESTful Web Service Clients Using Feature Classes

You can programmatically attach Oracle WSM security policies to RESTful web service clients using the Feature classes defined in Table 1–5. The classes are provided in the `oracle.wsm.metadata.feature` package.

Table 1–5 Feature Classes Used for Attaching Policies to RESTful Clients

Feature Class	Description
`AbstractPolicyFeature`	Base abstract class for policy subject feature classes.
`PolicySetFeature`	Set of policy references and configuration override properties to attach to the policy subject.
`PolicyReferenceFeature`	Single policy reference to attach to the policy subject.
`PropertyFeature`	Optional property that can be used to override the configuration of one or more policies.

When you create a RESTful client instance, optionally you can pass client configuration properties by defining a `com.sun.jersey.api.client.config.ClientConfig` and passing the information to the `create` method of the `com.sun.jersey.api.client.Client` class.

Using the `ClientConfig`, you can attach Oracle WSM policies and override configuration properties, as shown in the following example.

The following code attaches OAuth 2.0 policies to the RESTful client and overrides configuration properties using Feature classes.

Example 1–2 Attaching Policies to RESTful Web Service Clients Using Feature Classes

```
package sample.restclient;
import java.io.IOException;
```

```
import java.io.PrintWriter;

import weblogic.jaxrs.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.ClientConfig;
import com.sun.jersey.api.client.config.DefaultClientConfig;

import javax.servlet.*;
import javax.servlet.http.*;
import org.json.simple.JSONObject;
import org.json.simple.JSONValue;

import oracle.wsm.metadata.feature.AbstractPolicyFeature;
import oracle.wsm.metadata.feature.PolicyReferenceFeature;
import oracle.wsm.metadata.feature.PolicySetFeature;
import oracle.wsm.metadata.feature.PropertyFeature;

public class BankingServlet extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=UTF-8";
    private static final String OAUTH_TOKEN_ENDPOINT =
"http://example.com:1234/ms_oauth/oauth2/endpoints/oauthservice/tokens";
    private static final String RESOURCE_SERVER_ADDRESS =
"http://example.com:1234";
    private static final String ACCOUNT_RESOURCE = RESOURCE_SERVER_ADDRESS +
"/banking_owsm/account/balance";
    private static final String CLIENT_CSF_KEY = "bankingweb.csf.key";
    private static final String OAUTH_RESOURCE_SCOPE =
"AccountResource.balance.read";

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }
    public void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        response.setContentType(CONTENT_TYPE);

        String userName = request.getRemoteUser();

        out.println("<html>");
        out.println("<head><title>The OAuth Bank</title></head>");
        out.println("<body>");
        out.println("<p>Hello " + userName + ", welcome to the OAuth Bank</p>");

        ClientConfig cc = new DefaultClientConfig();

// Setting the token.uri. This is used by the Oracle WSM client agent
// when submitting requests for the authorization code.
        PropertyFeature oAuthConfigPropFeature = new
PropertyFeature("token.uri",OAUTH_TOKEN_ENDPOINT);
        PropertyFeature[] oAuthClientPropFeatures = new PropertyFeature[2];

// Setting oauth2.client.csf.key and scope.
// The key pointed to by oauth2.client.csf.key must be available in the
// OPSS credential store as a password credential, holding the
// client id/client secret pair identifying the client.
// scope is the resource scope for which the requested access token is valid.
        oAuthClientPropFeatures[0] = new PropertyFeature("oauth2.client.csf.key",
```

```
CLIENT_CSF_KEY);
        oAuthClientPropFeatures[1] = new PropertyFeature("scope",OAUTH_RESOURCE_
SCOPE);

        PolicyReferenceFeature[] policyFeatures = new PolicyReferenceFeature[2];
        policyFeatures[0] = new PolicyReferenceFeature("oracle/oauth2_config_
client_policy", oAuthConfigPropFeature);
        policyFeatures[1] = new PolicyReferenceFeature("oracle/http_oauth2_token_
client_policy", oAuthClientPropFeatures);

        // Attaching policies.
        cc.getProperties().put(AbstractPolicyFeature.ABSTRACT_POLICY_FEATURE, new
PolicySetFeature(policyFeatures));

        Client client = Client.create(cc);
        WebResource webResource = client.resource(ACCOUNT_RESOURCE);
        ClientResponse result =
webResource.accept("application/json").get(ClientResponse.class);

        if (result.getStatus() == 200) {

            String body = result.getEntity(String.class);

            final JSONObject obj = (JSONObject)JSONValue.parse(body);


            if (obj != null && obj.get("balance") != null) {
                String balance = obj.get("balance").toString();
                out.println ("<p>Account balance: " + balance);
            }
            else {
                out.println("<p>Error while retrieving account balance.");
            }
        }

        else {
            out.println("<p>Error retrieving account balance: HTTP response
status: " + result.getStatus());
        }

        out.println("</body></html>");

        out.close();
    }
}
```

1.11 Registering a More Recent Version of the Jersey JAX-RS RI

If you wish to use a more recent version of the Jersey JAX-RS RI shared libraries than the version that is provided with WebLogic Server, you need to perform the following steps:

1. Download the required version of the relevant Jersey JAR file from the Jersey Web site at: http://jersey.java.net.
2. Expand the JAR file downloaded in Step 1 and create a new shared library following the steps described in "Creating Shared Java EE Libraries" in *Developing Applications for Oracle WebLogic Server*.

3. Register the shared library by deploying the shared library files to the target servers identified in Step 2, and identifying the deployment as a library using the `-library` option. You must do the following:

   - Set the `-name` argument to match the standard Jersey JAX-RS RI shared library name, defined in Table 1–3. For example, `jersey-bundle`.
   - Set the `-libSpecVer` and `-libImplVer` arguments to distinguish between the different shared library versions.

   The following shows an example of how to deploy the latest versions of the Jersey JAX-RS RI functionality. For more information about the weblogic.Deployer, see "weblogic.Deployer Command-Line Reference" in *Deploying Applications to Oracle WebLogic Server*.

   ```
   weblogic.Deployer -verbose -noexit -name jersey-bundle -source
   C:\myinstall\wlserver_10.3\common\deployable-libraries\jersey-bundle-1.2.war
   -targets myserver -adminurl t3://localhost:7001 -user system -password
   ******** -deploy -library -libspecver 1.2 -libimplver 1.2
   ```

4. Determine if you need to reconfigure your Web application.

   If you set the `<exact-match>` directive to false in the `weblogic.xml` file when configuring your Web application, as described in "Configuring the Web Application to Use the Jersey JAX-RS RI" on page 6, then the shared library with the most recent specification version will be used and you do not have to update your Web application configuration.

   If you set the `<exact-match>` directive to true or if you want to use a version of the Jersey JAX-RS RI that is not the most recent version, then you will have to update the `weblogic.xml` to reference the desired shared library. For more information, see "Configuring the Web Application to Use the Jersey JAX-RS RI" on page 6.

5. Redeploy any applications that needs to use the newly registered version of the Jersey JAX-RS shared library.