Oracle® Fusion Middleware

Developing and Securing RESTful Web Services for Oracle WebLogic Server

12*c* (12.1.2)

E28162-03

February 2014

Documentation for software developers that describes how to develop WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

ORACLE

Oracle Fusion Middleware Developing and Securing RESTful Web Services for Oracle WebLogic Server, 12*c* (12.1.2)

E28162-03

Contents

3 Developing RESTful Web Service Clients

4 Packaging and Deploying RESTful Web Services

5 Securing RESTful Web Services and Clients

6 Testing RESTful Web Services

7 Monitoring RESTful Web Services and Clients

A Updating the Version of Jersey JAX-RS RI

Preface

This preface describes the document accessibility features and conventions used in this guide—*Developing and Securing RESTful Web Services for Oracle WebLogic Server*.

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 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.

What's New in This Guide

The following topics introduce the new and changed features of RESTful Web services and provides pointers to additional information. This document was released initially in Oracle Fusion Middleware 12*c* Release 1 (12.1.1).

New and Changed Features for 12*c* (12.1.2)

Oracle Fusion Middleware 12*c* (12.1.2) includes the following new and changed features for this document.

- Secure RESTful Web services using Oracle Web Services Manager (OWSM) policies. For more information, see "Securing RESTful Web Services and Clients Using OWSM Policies" on page 5-1.
- New standalone Web service client JAR files that support basic RESTful Web service client-side functionality and Oracle Web Services Manager (OWSM) security policy support. See "Invoking a RESTful Web Service from a Standalone Client" on page 3-10.

1 Introduction to RESTful Web Services

This chapter provides an overview of developing WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

This chapter includes the following sections:

- Section 1.1, "Introduction to the REST Architectural Style"
- Section 1.2, "What are RESTful Web Services?"
- Section 1.3, "Standards Supported for RESTful Web Service Development on WebLogic Server"
- Section 1.4, "Roadmap for Implementing RESTful Web Services"
- Section 1.6, "Learn More About RESTful Web Services"

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 Standards Supported for RESTful Web Service Development on WebLogic Server

The JAX-RS provides support for creating Web services according to REST architectural style. JAX-RS uses annotations to simplify the development of RESTful Web services. By simply adding annotations to your Web service, you can define the resources and the actions that can be performed on those resources. JAX-RS is part of the Java EE 6 full profile, and is integrated with Contexts and Dependency Injection (CDI) for the Java EE Platform (CDI), Enterprise JavaBeans (EJB) technology, and Java Servlet technology.

WebLogic Server supports Jersey 1.13 JAX-RS Reference Implementation (RI), which is a production quality implementation of the JSR-311 JAX-RS 1.1 specification, defined at: `https://jcp.org/en/jsr/summary?id=311`. As required, you can use a more recent version of the Jersey JAX-RS RI, as described in Appendix A, "Updating the Version of Jersey JAX-RS RI."

The Jersey 1.13 JAX-RS RI bundle includes the following functionality:

- Jersey
- JAX-RS API
- JSON processing and streaming

For more information about JAX-RS and samples, see Section 1.6, "Learn More About RESTful Web Services."

1.4 Roadmap for Implementing RESTful Web Services

The following table provides a roadmap of common tasks for developing, packaging and deploying, invoking, and monitoring RESTful Web services and clients using WebLogic Server.

Table 1–2 Roadmap for Implementing RESTful Web Services

This chapter . . .	Describes how to . . .
Chapter 2, "Developing RESTful Web Services"	Develop RESTful Web services.
Chapter 3, "Developing RESTful Web Service Clients"	Develop clients to invoke the RESTful Web service using the Jersey client API.
Chapter 4, "Packaging and Deploying RESTful Web Services"	Package and deploy RESTful Web services.
Chapter 5, "Securing RESTful Web Services and Clients"	Secure RESTful Web services.
Chapter 6, "Testing RESTful Web Services"	Test RESTful Web services.
Chapter 7, "Monitoring RESTful Web Services and Clients"	Monitor RESTful Web services
Appendix A, "Updating the Version of Jersey JAX-RS RI"	Update the version of the Jersey JAX-RS Reference Implementation (RI) used by your RESTful Web service applications.

1.5 Upgrading RESTful Web Services from Oracle WebLogic Server 10.3.*x* to 12.1.*x*

You can upgrade a RESTful Web service that was built in the Oracle WebLogic Server 11*g* Release 1 (10.3.*x*) environment to run in the Oracle WebLogic Server 12*c* Release 1 (12.1.*x*) environment. For detailed steps, see "Upgrading a 10.3.*x* RESTful Web Service (JAX-RS) to 12.1.*x*" in *Upgrading Oracle WebLogic Server*.

1.6 Learn More About RESTful Web Services

Table 1.6 provides a list of resources for more information about RESTful Web services.

Table 1–3 Resources for More Information

Resource	Link
Jersey 1.13 User Guide	`https://jersey.java.net/nonav/documentation/1.13/user-guide.html`
RESTful Web Services (JAX-RS) sample	"Sample Application and Code Examples" in *Understanding Oracle WebLogic Server*
The Java EE 6 Tutorial—Building RESTful Web Services With JAX-RS	`http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html`
Jersey site	`https://jersey.java.net`

Table 1–3 (Cont.) Resources for More Information

Resource	Link
Community Wiki for Project Jersey	https://wikis.oracle.com/display/Jersey/Main
Jersey 1.13 API Javadoc	https://jersey.java.net/nonav/apidocs/1.13/jersey/overview-summary.html
JSR-311 JAX-RS Specification	https://jcp.org/en/jsr/summary?id=311
JSR-311 JAX-RS Project	https://jsr311.java.net/
JSR-311 JAX-RS API Javadoc	https://jsr311.java.net/nonav/javadoc/index.html
"Representational State Transfer (REST)" in *Architectural Styles and the Design of Network-based Software Architectures* (Dissertation by Roy Fielding)	http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

2

Developing RESTful Web Services

This chapter describes how to develop WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

This chapter includes the following sections:

For information about developing RESTful Web service clients using Oracle JDeveloper, see "Creating RESTful Web Services and Clients" in *Developing Applications with Oracle JDeveloper*.

2.1 About RESTful Web Service Development

JAX-RS is a Java programming language API that uses annotations to simplify the development of RESTful Web services. JAX-RS annotations are runtime annotations. When you deploy the Java EE application archive containing JAX-RS resource classes to WebLogic Server, as described in Chapter 4, "Packaging and Deploying RESTful Web Services," the runtime configures the resources, generates the helper classes and artifacts, and exposes the resource to clients.

The following sections provide more information about RESTful Web service development:

2.1.1 Summary of Tasks to Develop RESTful Web Services

Table 2–1 summarizes a subset of the tasks that are required to develop RESTful Web service using JAX-RS annotations. For more information about advanced tasks, see Section 2.13, "More Advanced RESTful Web Service Tasks."

In addition to the development tasks described below, you may wish 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, see Section 4.2, "Packaging With an Application Subclass".

Table 2–1 Summary of Tasks to Develop RESTful Web Services

Task	More Information
Define the root resource class.	Section 2.2, "Defining the Root Resource Class"
Define the relative URI of the root resource class and its methods using the `@Path` annotation. If you define the `@Path` annotation using a variable, you can assign a value to it using the `@PathParam` annotation.	Section 2.3, "Defining the Relative URI of the Root Resource and Subresources"
Map incoming HTTP requests to your Java methods using `@GET`, `@POST`, `@PUT`, or `@DELETE`, to get, create, update, or delete representations of the resource, respectively.	Section 2.4, "Mapping Incoming HTTP Requests to Java Methods"
Customize the request and response messages, as required, to specify the MIME media types of representations a resource can produce and consume.	Section 2.5, "Customizing Media Types for the Request and Response Messages"
Extract information from the request.	Section 2.6, "Extracting Information From the Request Message"
Build custom response messages to customize response codes or include additional metadata.	Section 2.7, "Building Custom Response Messages"
Access information about the application deployment context or the context of individual requests.	Section 2.9, "Accessing the Application Context"
Build new or extend existing resource URIs.	Section 2.10, "Building URIs"
Evaluate one or more preconditions before processing a GET request, potentially reducing bandwidth and improving server performance.	Section 2.11, "Using Conditional GETs"
Access the WADL.	Section 2.12, "Accessing the WADL"
Secure your RESTful Web services.	Chapter 5, "Securing RESTful Web Services and Clients"

2.1.2 Example of a RESTful Web Service

Example 2–1 provides a simple example of a RESTful Web service. In this example:

- The `helloWorld` class is a resource with a relative URI path defined as `/helloworld`. At runtime, if the context root for the WAR file is defined as `http://examples.com`, the full URI to access the resource is

`http://examples.com/helloworld`. For more information, see Section 2.3, "Defining the Relative URI of the Root Resource and Subresources."

- The `sayHello` method supports the HTTP GET method. For more information, see Section 2.4, "Mapping Incoming HTTP Requests to Java Methods."
- The `sayHello` method produces content of the MIME media type `text/plain`. For more information, see Section 2.5, "Customizing Media Types for the Request and Response Messages."

Additional examples are listed in Section 1.6, "Learn More About RESTful Web Services."

Example 2–1 Simple 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
   @Produces("text/plain")
   public String sayHello() {
      return "Hello World!";
   }
}
```

2.2 Defining the Root Resource Class

A *root resource class* is a Plain Old Java Object (POJO) for which one or both of the following statements are true:

- Is annotated with `@Path`. For more information, see Section 2.3, "Defining the Relative URI of the Root Resource and Subresources."
- Has at least one method annotated with `@Path` or with a *request method designator*, such as `@GET`, `@POST`, `@PUT`, or `@DELETE`. A *resource method* is a method in the resource class that is annotated using a request method designator. For more information, see Section 2.4, "Mapping Incoming HTTP Requests to Java Methods."

2.3 Defining the Relative URI of the Root Resource and Subresources

Add the `javax.ws.rs.Path` annotation at the class level of the resource to define the relative URI of the RESTful Web service. Such classes are referred to a root resource classes. You can add `@Path` on methods of the root resource class as well, to define subresources to group specific functionality.

The following sections describe how to define the relative URI of the root resource and subresources:

- Section 2.3.1, "How to Define the Relative URI of the Resource Class (@Path)"

- Section 2.3.2, "How to Define the Relative URI of Subresources (@Path)"
- Section 2.3.3, "What Happens at Runtime: How the Base URI is Constructed"

2.3.1 How to Define the Relative URI of the Resource Class (@Path)

The `@Path` annotation defines the relative URI path for the resource, and can be defined as a constant or variable value (referred to as "URI path template"). You can add the `@Path` annotation at the class or method level.

To define the URI as a constant value, pass a constant value to the `@Path` annotation. Preceding and ending slashes (`/`) are optional.

In Example 2–2, the relative URI for the resource class is defined as the constant value, `/helloworld`.

Example 2–2 Defining the Relative URI as a Constant Value

```
package samples.helloworld;
import javax.ws.rs.Path;
...
// Specifies the path to the RESTful service
@Path("/helloworld")
public class helloWorld {. . .}
```

To define the URI as a URI path template, pass one or more variable values enclosed in braces in the `@Path` annotation. Then, you can use the `javax.ws.rs.PathParam` annotation to extract variable information from the request URI, defined by the `@Path` annotation, and initialize the value of the method parameter, as described in Section 2.6.1, "How to Extract Variable Information from the Request URI (@PathParam)."

In Example 2–3, the relative URI for the resource class is defined using a variable, enclosed in braces, for example, `/users/{username}`.

Example 2–3 Defining the Relative URI as a Variable Value

```
package samples.helloworld;

import javax.ws.rs.Path;
...
// Specifies the path to the RESTful service
@Path("/users/{username}")
public class helloWorld {. . .}
}
```

To further customize the variable, you can override the default regular expression of "[^/]+?" by specifying the expected regular expression as part of the variable definition. For example:

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]}")
```

In this example, the `username` variable will match only user names that begin with one uppercase or lowercase letter followed by zero or more alphanumeric characters or the underscore character. If the username does not match the requirements, a `404 (Not Found)` response will be sent to the client.

For more information about the `@Path` annotation, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/Path.html`.

2.3.2 How to Define the Relative URI of Subresources (@Path)

Add the `javax.ws.rs.Path` annotation to the method of a resource to define a subresource. Subresources enable users to group specific functionality for a resource.

In Example 2–3, if the request path of the URI is `users/list`, then the `getUserList` subresource method is matched and a list of users is returned.

Example 2–4 Defining a Subresource

```
package samples.helloworld;

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

// Specifies the path to the RESTful service
@Path("/users")
public class UserResource {
. . .
    @GET
   @Path("/list")
    public String getUserList() {
      ...
   }
}
```

2.3.3 What Happens at Runtime: How the Base URI is Constructed

The base URI is constructed as follows:

`http://`*`myHostName`*`/`*`contextPath`*`/`*`servletURI`*`/`*`resourceURI`*

- *`myHostName`*—DNS name mapped to the Web Server. You can replace this with *`host:port`* which specifies the name of the machine running WebLogic Server and the port used to listen for requests.
- *`contextPath`*—Name of the standalone Web application. The Web application name is specified in the `META-INF/application.xml` deployment descriptor in an EAR file or the `weblogic.xml` deployment descriptor in a WAR file. If not specified, it defaults to the name of the WAR file minus the `.war` extension. For more information, see "context-root" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.
- *`servletURI`*—Base URI for the servlet context path. This path is configured as part of the packaging options defined in Table 4–1. Specifically, you can define the servlet context path by:
  - Updating the `web.xml` deployment descriptor to define the servlet mapping.
  - Adding a `javax.ws.rs.ApplicationPath` annotation to the class that extends `javax.ws.rs.core.Application`, if defined.

  If the servlet context path is configured using both options above, then the servlet mapping takes precedence. If you do not configure the servlet context path in your configuration using either of the options specified above, the WebLogic Server provides a default RESTful Web service application context path, `resources`. For more information, see Section 4.1, "About RESTful Web Service Packaging and Deployment."
- *`resourceURI`*—`@Path` value specified for the resource or subresource. This path may be constructed from multiple resources and subresources `@Path` values.

In Example 2–2, at runtime, if the context path for the WAR file is defined as `rest` and the default URI for the servlet (`resources`) is in effect, the base URI to access the resource is `http://myServer:7001/rest/resources/helloworld`.

In Example 2–3, at runtime, the base URI will be constructed based on the value specified for the variable. For example, if the user entered `johnsmith` as the username, the base URI to access the resource is `http://myServer:7001/rest/resources/users/johnsmith`.

2.4 Mapping Incoming HTTP Requests to Java Methods

JAX-RS uses Java annotations to map an incoming HTTP request to a Java method. Table 2–2 lists the annotations available, which map to the similarly named HTTP methods.

Table 2–2 javax.ws.rs Annotations for Mapping HTTP Requests to Java Methods

Annotation	Description	Idempotent
`@GET`	Transmits a representation of the resource identified by the URI to the client. The format might be HTML, plain text, JPEG, and so on. See Section 2.4.2, "How to Transmit a Representation of the Resource (@GET)."	Yes
`@PUT`	Creates or updates the representation of the specified resource identified by the URI. See Section 2.4.3, "How to Create or Update the Representation of the Resource (@PUT)."	Yes
`@DELETE`	Deletes the representation of the resource identified by the URI. See Section 2.4.4, "How to Delete a Representation of the Resource (@DELETE)."	Yes
`@POST`	Creates, updates, or performs an action on the representation of the specified resource identified by the URI. See Section 2.4.5, "How to Create, Update, or Perform an Action on a Representation of the Resource (@POST)."	No
`@HEAD`	Returns the response headers only, and not the actual resource (that is, no message body). This is useful to save bandwidth to check characteristics of a resource without actually downloading it. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?j7avax/ws/rs/HEAD.html`. The `HEAD` method is implemented automatically if not implemented explicitly. In this case, the runtime invokes the implemented `GET` method, if present, and ignores the response entity, if set.	Yes
`@OPTIONS`	Returns the communication options that are available on the request/response chain for the specified resource identified by the URI. The `Allow` response header will be set to the set of HTTP methods supported by the resource and the WADL file is returned. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/OPTIONS.html`. The `OPTIONS` method is implemented automatically if not implemented explicitly. In this case, the `Allow` response header is set to the set of HTTP methods supported by the resource and the WADL describing the resource is returned.	Yes
`@HttpMethod`	Indicates that the annotated method should be used to handle HTTP requests. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/HttpMethod.html`.	N/A

The following sections provide more information about the JAX-RS annotations used for mapping HTTP requests to Java methods.

- Section 2.4.1, "About the Jersey Bookmark Sample"

- Section 2.4.2, "How to Transmit a Representation of the Resource (@GET)"
- Section 2.4.3, "How to Create or Update the Representation of the Resource (@PUT)"
- Section 2.4.4, "How to Delete a Representation of the Resource (@DELETE)"
- Section 2.4.5, "How to Create, Update, or Perform an Action on a Representation of the Resource (@POST)"

2.4.1 About the Jersey Bookmark Sample

The examples referenced in the following sections are excerpted from the **bookmark sample** that is delivered with Jersey 1.13 JAX-RS RI. The bookmark sample provides a Web application that maintains users and the browser bookmarks that they set.

The following table summarizes the resource classes in the sample, their associated URI path, and the HTTP methods demonstrated by each class.

Table 2–3 About the Jersey Bookmark Sample

Resource Class	URI Path	HTTP Methods Demonstrated
`UsersResource`	`/users/`	GET
`UserResource`	`/users/{userid}`	GET, PUT, DELETE
`BookmarksResource`	`/users/{userid}/bookmarks`	GET, POST
`BookmarkResource`	`/users/{userid}/bookmarks/{bmid}`	GET. PUT, DELETE

The bookmark sample, and other Jersey samples, can be accessed in one of the following ways:

- Downloading the Jersey 1.13 software at `https://jersey.java.net`
- Browsing the Maven repositories at: `https://maven.java.net/content/repositories/releases/com/sun/jersey/samples`

2.4.2 How to Transmit a Representation of the Resource (@GET)

The `javax.ws.rs.GET` annotation transmits a representation of the resource identified by the URI to the client. The format or the representation returned in the response entity-body might be HTML, plain text, JPEG, and so on. For more information about the `@GET` annotation, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/GET.html`.

In Example 2–5, the annotated Java method, `getBookmarkAsJsonArray`, from the `BookmarksResource` class in the Jersey bookmark sample, will process HTTP GET requests. For more information about the Jersey bookmark sample, see Section 2.4.1, "About the Jersey Bookmark Sample."

Example 2–5 Mapping the HTTP GET Request to a Java Method (BookmarksResource Class)

```
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
...
public class BookmarksResource {
...
    @Path("{bmid: .+}")
```

```
    public BookmarkResource getBookmark(@PathParam("bmid") String bmid) {
        return new BookmarkResource(uriInfo, em,
                userResource.getUserEntity(), bmid);
    }
    @GET
    @Produces("application/json")
    public JSONArray getBookmarksAsJsonArray() {
        JSONArray uriArray = new JSONArray();
        for (BookmarkEntity bookmarkEntity : getBookmarks()) {
            UriBuilder ub = uriInfo.getAbsolutePathBuilder();
            URI bookmarkUri = ub.
                    path(bookmarkEntity.getBookmarkEntityPK().getBmid()).
                    build();
            uriArray.put(bookmarkUri.toASCIIString());
        }
        return uriArray;
    }
...
}
```

In Example 2–6, the annotated Java method, `getBookmark`, from the `BookmarkResource` class in the Jersey bookmark sample, will process HTTP GET requests. This example shows how to process the JSON object that is returned. For more information about the Jersey bookmark sample, see Section 2.4.1, "About the Jersey Bookmark Sample."

Example 2–6 Mapping the HTTP GET Request to a Java Method (BookmarkResource Class)

```
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
...
public class BookmarkResource {
...
    @GET
    @Produces("application/json")
    public JSONObject getBookmark() {
        return asJson();
    }
...
    public JSONObject asJson() {
        try {
            return new JSONObject()
            .put("userid", bookmarkEntity.getBookmarkEntityPK().getUserid())
            .put("sdesc", bookmarkEntity.getSdesc())
            .put("ldesc", bookmarkEntity.getLdesc())
            .put("uri", bookmarkEntity.getUri());
        } catch (JSONException je){
            return null;
        }
    }
}
```

2.4.3 How to Create or Update the Representation of the Resource (@PUT)

The `javax.ws.rs.PUT` annotation creates or updates the representation of the specified resource identified by the URI. For more information about the `@PUT` annotation, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/PUT.html.

In Example 2–7, the annotated Java method, `putBookmark`, from the `BookmarkResource` class in the Jersey bookmark sample, will process HTTP PUT requests and update the specified bookmark. For more information about the Jersey bookmark sample, see Section 2.4.1, "About the Jersey Bookmark Sample."

Example 2–7 Mapping the HTTP PUT Request to a Java Method

```
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
...
public class BookmarkResource {
...
    @PUT
    @Consumes("application/json")
    public void putBookmark(JSONObject jsonEntity) throws JSONException {
        bookmarkEntity.setLdesc(jsonEntity.getString("ldesc"));
        bookmarkEntity.setSdesc(jsonEntity.getString("sdesc"));
        bookmarkEntity.setUpdated(new Date());

        TransactionManager.manage(new Transactional(em) { public void transact() {
            em.merge(bookmarkEntity);
        }});
    }

}
```

2.4.4 How to Delete a Representation of the Resource (@DELETE)

The `javax.ws.rs.DELETE` annotation deletes the representation of the specified resource identified by the URI. The response entity-body may return a status message or may be empty. For more information about the `@DELETE` annotation, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/DELETE.html`.

In Example 2–8, the annotated Java method, `deleteBookmark`, from the `BookmarkResource` class in the Jersey bookmark sample, will process HTTP DELETE requests, and delete the specified bookmark. For more information about the Jersey bookmark sample, see Section 2.4.1, "About the Jersey Bookmark Sample."

Example 2–8 Mapping the HTTP DELETE Request to a Java Method

```
import javax.ws.rs.DELETE;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
...
public class BookmarkResource {
...
    @DELETE
    public void deleteBookmark() {
        TransactionManager.manage(new Transactional(em) { public void transact() {
            UserEntity userEntity = bookmarkEntity.getUserEntity();
            userEntity.getBookmarkEntityCollection().remove(bookmarkEntity);
            em.merge(userEntity);
            em.remove(bookmarkEntity);
        }});
    }
}
```

2.4.5 How to Create, Update, or Perform an Action on a Representation of the Resource (@POST)

The `javax.ws.rs.POST` annotation creates, updates, or performs an action on the representation of the specified resource identified by the URI. For more information about the `@POST` annotation, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/POST.html.

In Example 2–9, the annotated Java method, `postForm`, from the `BookmarksResource` class in the Jersey bookmark sample, will process HTTP POST requests, and update the specified information. For more information about the Jersey bookmark sample, see Section 2.4.1, "About the Jersey Bookmark Sample."

Example 2–9 Mapping the HTTP POST Request to a Java Method

```
import javax.ws.rs.POST;
import javax.ws.rs.Produces;
...
public class BookmarksResource {
...
    @POST
    @Consumes("application/json")
    public Response postForm(JSONObject bookmark) throws JSONException {
        final BookmarkEntity bookmarkEntity = new BookmarkEntity(
                getBookmarkId(bookmark.getString("uri")),
                userResource.getUserEntity().getUserid());

        bookmarkEntity.setUri(bookmark.getString("uri"));
        bookmarkEntity.setUpdated(new Date());
        bookmarkEntity.setSdesc(bookmark.getString("sdesc"));
        bookmarkEntity.setLdesc(bookmark.getString("ldesc"));
        userResource.getUserEntity().getBookmarkEntityCollection().add(bookmarkEntity);

        TransactionManager.manage(new Transactional(em) { public void transact() {
            em.merge(userResource.getUserEntity());
        }});

        URI bookmarkUri = uriInfo.getAbsolutePathBuilder().
                path(bookmarkEntity.getBookmarkEntityPK().getBmid()).
                build();
        return Response.created(bookmarkUri).build();
    }
}
```

2.5 Customizing Media Types for the Request and Response Messages

Add the `javax.ws.rs.Consumes` or `javax.ws.rs.Produces` annotation at the class level of the resource to customize the request and response media types, as described in the following sections:

- Section 2.5.1, "How To Customize Media Types for the Request Message (@Consumes)"
- Section 2.5.2, "How To Customize Media Types for the Response Message (@Produces)"
- Section 2.5.3, "What Happens At Runtime: How the Resource Method Is Selected for Response Messages"

2.5.1 How To Customize Media Types for the Request Message (@Consumes)

The `javax.ws.rs.Consumes` annotation enables you to specify the MIME media types of representations a resource can consume that were sent from the client. The `@Consumes` annotation can be specified at both the class and method levels and more than one media type can be declared in the same `@Consumes` declaration.

If there are no methods in a resource that can consume the specified MIME media types, the runtime returns an HTTP `415 Unsupported Media Type` error.

For more information about the `@Consumes` annotation, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/Consumes.html.

In Example 2–11, the `@Consumes` annotation defined for the Java class, `helloWorld`, specifies that the class produces messages using the `text/plain` MIME media type.

Example 2–10 Customizing the Media Types for the Request Message Using @Consumes

```
package samples.consumes;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...
@Path("/helloworld")
public class helloWorld {
...
    @POST
    @Consumes("text/plain")
    public void postMessage(String message) {
       // Store the message
   }
}
```

2.5.2 How To Customize Media Types for the Response Message (@Produces)

The `javax.ws.rs.Produces` annotation enables you to specify the MIME media types of representations a resource can produce and send back to the client. The `@Produces` annotation can be specified at both the class and method levels and more than one media type can be declared in the same `@Produces` declaration.

If there are no methods in a resource that can produce the specified MIME media types, the runtime returns an HTTP `406 Not Acceptable` error.

For more information about the `@Produces` annotation, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/Produces.html.

In Example 2–11, the `@Produces` annotation specified for the Java class, `SomeResource`, specifies that the class produces messages using the `text/plain` MIME media type. The `doGetAsPlainText` method defaults to the MIME media type specified at the class level. The `doGetAsHtml` method overrides the class-level setting and specifies that the method produces HTML rather than plain text.

Example 2–11 Customizing the Media Types for the Response Using @Produces

```
package samples.produces;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

@Path("/myResource")
```

```
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() { ... }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() { ... }
}
```

2.5.3 What Happens At Runtime: How the Resource Method Is Selected for Response Messages

If a resource class is capable of producing more that one MIME media type, then the resource method that is selected corresponds to the acceptable media type declared in the `Accept` header of the HTTP request. In Example 2–11, if the `Accept` header is `Accept: text/html`, then the `doGetAsPlainText` method is invoked.

If multiple MIME media types are included in the `@Produces` annotation and both are acceptable to the client, the first media type specified is used. In Example 2–11, if the `Accept` header is `Accept: application/html, application/text`, then the `doGetAsHtml` method is invoked and the `application/html` MIME media type is used as it is listed first in the list.

2.6 Extracting Information From the Request Message

The `javax.ws.rs` package defines a set of annotations, shown in Table 2–2, that enable you extract information from the request message to inject into parameters of your Java method.

Table 2–4 javax.ws.rs Annotations for Extracting Information From the Request Message

Annotation	Description
`@CookieParam`	Extract information from the HTTP cookie-related headers to initialize the value of a method parameter. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/CookieParam.html`.
`@DefaultValue`	Define the default value of the request metadata that is bound using one of the following annotations: `@CookieParam`, `@FormParam`, `@HeaderParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`. For more information, see Section 2.6.3, "How to Define the DefaultValue (@DefaultValue)."
`@Encode`	Enable encoding of a parameter value that is bound using one of the following annotations: `@FormParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`. For more information, see Section 2.6.4, "Enabling the Encoding Parameter Values (@Encode)."
`@FormParam`	Extract information from an HTML form of the type `application/x-www-form-urlencoded`. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/FormParam.html`.
`@HeaderParam`	Extract information from the HTTP headers to initialize the value of a method parameter. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/HeaderParam.html`.

Table 2–4 (Cont.) javax.ws.rs Annotations for Extracting Information From the Request Message

Annotation	Description
`@MatrixParam`	Extract information from the URI path segments to initialize the value of a method parameter. For more information, see `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/MatrixParam.html`.
`@PathParam`	Define the relative URI as a variable value (referred to as "URI path template"). For more information, see Section 2.6.1, "How to Extract Variable Information from the Request URI (@PathParam)."
`@QueryParam`	Extract information from the query portion of the request URI to initialize the value of a method parameter. For more information, see Section 2.6.2, "How to Extract Request Parameters (@QueryParam)."

2.6.1 How to Extract Variable Information from the Request URI (@PathParam)

Add the `javax.ws.rs.PathParam` annotation to the method parameter of a resource to extract the variable information from the request URI and initialize the value of the method parameter. You can define a default value for the variable value using the `@DefaultValue` annotation, as described in Section 2.6.3, "How to Define the DefaultValue (@DefaultValue)."

In Example 2–3, the `@PathParam` annotation assigns the value of the `username` variable that is defined as part of the URI path by the `@Path` annotation to the `userName` method parameter.

Example 2–12 Extracting Variable Information From the Request URI

```
package samples.helloworld;

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

// Specifies the path to the RESTful service
@Path("/users")
public class helloWorld {
. . .
    @GET
    @Path("/{username}")
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
      ...
   }
}
```

2.6.2 How to Extract Request Parameters (@QueryParam)

Add the `javax.ws.rs.QueryParam` annotation to the method parameter of a resource to extract information from the query portion of the request URI and initialize the value of the method parameter.

The type of the annotated method parameter can be any of the following:

- Primitive type (`int`, `char`, `byte`, and so on)
- User-defined type
- Constructor that accepts a single String argument

- Static method named `valueOf` or `fromString` that accepts a single String argument (for example, `integer.valueOf(String)`)
- `List<T>`, `Set<T>`, or `SortedSet<T>`

If the `@QueryParam` annotation is specified but the associated query parameter is not present in the request, then the parameter value will set as an empty collection for `List`, `Set` or `SortedSet`, the Java-defined default for primitive types, and NULL for all other object types. Alternatively, you can define a default value for the parameter using the `@DefaultValue` annotation, as described in Section 2.6.3, "How to Define the DefaultValue (@DefaultValue)."

For more information about the `@QueryParam` annotation, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/QueryParam.html.

In Example 2–13, if the `step` query parameter exists in the query component of the request URI, the value will be assigned to the `step` method parameter as an integer value. If the value cannot be parsed as an integer value, then a `400 (Client Error)` response is returned. If the `step` query parameter does not exist in the query component of the request URI, then the value is set to NULL.

Example 2–13 Extracting Request Parameters (@QueryParam)

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.QueryParam;
...
    @Path("smooth")
    @GET
    public Response smooth(@QueryParam("step") int step)
    { ... }
}
```

2.6.3 How to Define the DefaultValue (@DefaultValue)

Add the `javax.ws.rs.DefaultValue` annotation to define the default value of the request metadata that is bound using one of the following annotations: `@CookieParam`, `@FormParam`, `@HeaderParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`. For more information about the `@DefaultValue` annotation, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/DefaultValue.html.

In Example 2–14, if the `step` query parameter does not exist in the query component of the request URI, the default value of 2 will be assigned to the `step` parameter.

Example 2–14 Defining the Default Value (@DefaultValue)

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.QueryParam;
...
    @Path("smooth")
    @GET
    public Response smooth(@DefaultValue("2") @QueryParam("step") int step)
    { ... }
}
```

2.6.4 Enabling the Encoding Parameter Values (@Encode)

Add the `javax.ws.rs.Encode` annotation at the class or method level to enable the encoding of a parameter value that is bound using one of the following annotations: `@FormParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`. If specified at the class level, parameters for all methods in the class will be encoded. For more information, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/Encoded.html.

In Example 2–15, the `@Encode` annotation enables the encoding of parameter values bound using the `@PathParam` annotation.

Example 2–15 Encoding Parameter Values

```
package samples.helloworld;

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

// Specifies the path to the RESTful service
@Path("/users")
public class helloWorld {
. . .
    @GET
    @Path("/{username}")
    @Produces("text/xml")
    @Encode
    public String getUser(@PathParam("username") String userName) {
       ...
   }
}
```

2.7 Building Custom Response Messages

By default, JAX-RS responds to HTTP requests using the default response codes defined in the HTTP specification, such as `200 OK` for a successful GET request and `201 CREATED` for a successful PUT request.

In some cases, you may want to customize the response codes returned or include additional metadata information in the response. For example, you might want to include the `Location` header to specify the URI to the newly created resource. You can modify the response message returned using the `javax.ws.rs.core.Response` class.

An application can extend the `Response` class directly or use one of the static `Response` methods, defined in Table 2–5, to create a `javax.ws.rs.core.Response.ResponseBuilder` instance and build the `Response` instance. For more information about the `Response` methods, see the Javadoc at http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/Response.html.

Table 2–5 Creating a Response Instance Using the ResponseBuilder Class

Method	Description
`created()`	Creates a new `ResponseBuilder` instance and sets the `Location` header to the specified value.
`fromResponse()`	Creates a new `ResponseBuilder` instance and copies an existing response.
`noContent()`	Creates a new `ResponseBuilder` instance and defines an empty response.
`notAcceptable()`	Creates a new `ResponseBuilder` instance and defines a unacceptable response.
`notModified()`	Creates a new `ResponseBuilder` instance and returns a not-modified status.
`ok()`	Creates a new `ResponseBuilder` instance and returns an OK status.
`seeOther()`	Creates a new `ResponseBuilder` instance for a redirection.
`serverError()`	Creates a new `ResponseBuilder` instance and returns a server error status.
`status()`	Creates a new `ResponseBuilder` instance and returns the specified status.
`temporaryRedirect()`	Creates a new `ResponseBuilder` instance for a temporary redirection.

Once you create a `ResponseBuilder` instance, you can call the methods defined in Table 2–6 to build a custom response. Then, call the `build()` method to create the final `Response` instance. For more information about the `ResponseBuilder` class and its methods, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/Response.ResponseBuilder.html.

Table 2–6 ResponseBuilder Methods for Building a Custom Response

Method	Description
`build()`	Creates the Response instance from the current `ResponseBuilder` instance.
`cacheControl()`	Sets the cache control.
`clone()`	Create a copy of the `ResponseBuilder` to preserve its state.
`contentLocation()`	Sets the content location.
`cookie()`	Add cookies to the response.
`entity()`	Defines the entity.
`expires()`	Sets the expiration date.
`header()`	Adds a header to the response.
`language()`	Sets the language.
`lastModified()`	Set the last modified date.
`location()`	Sets the location.
`newInstance()`	Creates a new `ResponseBuilder` instance.
`status()`	Sets the status.
`tag()`	Sets an entity tag.
`type()`	Sets the response media type.
`variant()`	Set representation metadata.
`variants()`	Add a `Vary` header that lists the available variants.

Example 2–16 shows how to build a `Response` instance using `ResponseBuilder`. In this example, the standard status code of `200 OK` is returned and the media type of the response is set to `text/html`. A call to the `build()` method creates the final `Response` instance.

Example 2–16 Building a Custom Response

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.ResponseBuilder;
...
@Path("/content")
public class getDocs {
   @GET
   @Path("{id}")
   public Response getHTMLDoc(@PathParm("id") int docId)
   {
      Document document = ...;
      ResponseBuilder response = Response.ok(document);
      response.type("text/html");
      return response.build();
   }
}
```

If you wish to build an HTTP response using a generic type, to avoid type erasure at runtime you need to create a `javax.ws.rs.core.GenericEntity` object to preserve the generic type. For more information, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/GenericEntity.html.

Example 2–17 provides an example of how to build an HTTP response using `GenericEntity` to preserve the generic type.

Example 2–17 Building a Custom Response Using a Generic Type

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.ResponseBuilder;
javax.ws.rs.core.GenericEntity;
...
@Path("/content")
public class getDocs {
   @GET
   @Path("{id}")
   public Response getHTMLDoc(@PathParm("id") int docId)
   {
      Document document = ...;
      List<String> list = new ArrayList<String>();
      GenericEntity<List<String>> entity = new GenericEntity<List<String>>(list) {};
      ...
      ResponseBuilder response = Response.ok(document);
      response.entity(entity);
      return response.build();
   }
}
```

2.8 Mapping HTTP Request and Response Entity Bodies Using Entity Providers

Table 2–7 lists the Java types that are supported automatically by HTTP request and response entity bodies.

Table 2–7 Java Types Supported for HTTP Request and Response Entity Bodies

Java Type	Supported Media Types
`byte[]`	All media types (`*/*`)
`java.lang.String`	All media types (`*/*`)
`java.io.InputStream`	All media types (`*/*`)
`java.io.Reader`	All media types (`*/*`)
`java.io.File`	All media types (`*/*`)
`javax.activation.DataSource`	All media types (`*/*`)
`javax.xml.transform.Source`	XML media types (`text/xml`, `application/xml`, and `application/*+xml`) and JSON media types (`application/json`, `application/*+json`)
`javax.xml.bind.JAXBElement` and application-supplied JAXB classes	XML media types (`text/xml`, `application/xml`, and `application/*+xml`)
`MultivaluedMap<String,String>`	Form content (`application/x-www-form-urlencoded`)
`StreamingOutput`	All media types (`*/*`), `MessageBodyWriter` only

If your RESTful Web service utilizes a type that is not listed in Table 2–7, you must define an entity provider, by implementing one of the interfaces defined in Table 2–8, to map HTTP request and response entity bodies to method parameters and return types.

Table 2–8 Entity Providers for Mapping HTTP Request and Response Entity Bodies to Method Parameters and Return Types

Entity Provider	Description

`javax.ws.rs.ext.MessageBodyReader`	Maps an HTTP request entity body to a method parameter for an HTTP request. Optionally, you can use the `@Consumes` annotation to specify the MIME media types supported for the entity provider, as described in "Customizing Media Types for the Request and Response Messages" on page 2-10.
	For example:
	`@Consumes("application/x-www-form-urlencoded")`
	`@Provider`
	`public class FormReader implements MessageBodyReader<NameValuePair> { ... }`

`javax.ws.rs.ext.MessageBodyWriter`	Maps the return value to an HTTP response entity body for an HTTP response. Optionally, you can use the `@Produces` annotation to specify the MIME media types supported for the entity provider, as described in "Customizing Media Types for the Request and Response Messages" on page 2-10.
	For example:
	`@Produces("text/html")`
	`@Provider`
	`public class FormWriter implements`
	`MessageBodyWriter<Hashtable<String, String>> { ... }`

Note: Jersey JSON provides a set of JAX-RS `MessageBodyReader` and `MessageBodyWriter` providers distributed with the jersey-json module. For more information, see "JSON Support" in the `Jersey User Guide` at: https://jersey.java.net/documentation/1.13/user-guide.html#json.

The following code excerpt provides an example of a class that contains a method (`getClass`) that returns a custom type, and that requires you to write an entity provider.

```
public class Class1
{
  public String hello() { return "Hello"; }
  public Class2 getClass(String name) { return new Class2(); };
}

public class Class2
{
  public Class2() { }
}
```

2.9 Accessing the Application Context

The `javax.ws.rs.core.Context` annotation enables you to access information about the application deployment context and the context of individual requests. Table 2–9 summarizes the context types that you can access using the `@Context` annotation. For more information, see the Javadoc at http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/Context.html.

Table 2–9 Context Types

Use this context type . . .	To . . .
`HttpHeaders`	Access HTTP header information.
`Providers`	Lookup Provider instances based on a set of search criteria.
`Request`	Determine the best matching representation variant and to evaluate whether the current state of the resource matches any preconditions defined. For more information, see Section 2.11, "Using Conditional GETs."
`SecurityContext`	Access the security context and secure the RESTful Web service. For more information, see Section 5.4, "Securing RESTful Web Services Using SecurityContext."
`UriInfo`	Access application and request URI information. For more information, see Section 2.10, "Building URIs."

2.10 Building URIs

You can use `javax.ws.rs.core.UriInfo` to access application and request URI information. Specifically, `UriInfo` can be used to return the following information:

- Deployed application's base URI
- Request URI relative to the base URI
- Absolute path URI (with or without the query parameters)

Using `UriInfo` you can return a URI or `javax.ws.rs.core.UriBuilder` instance. `UriBuilder` simplifies the process of building URIs, and can be used to build new or extend existing URIs.

The `UriBuilder` methods perform contextual encoding of characters not permitted in the corresponding URI component based on the following rules:

- `application/x-www-form-urlencoded` media type for query parameters, as defined in "Forms" in the HTML specification at the following URL: `http://www.w3.org/TR/html4/interact/forms.html#h-17.13.4.1`
- RFC 3986 for all other components, as defined at the following URL: `http://www.ietf.org/rfc/rfc3986.txt`

Example 2–18 shows how to obtain an instance of `UriInfo` using `@Context` and use it to return an absolute path of the request URI as a `UriBuilder` instance. Then, using `UriBuilder` build a URI for a specific user resource by adding the user ID as a path segment and store it in an array. In this example, the `UriInfo` instance is injected into a class field. This example is excerpted from the bookmark sample, as described in Section 2.4.1, "About the Jersey Bookmark Sample."

Example 2–18 Building URIs

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.core.UriBuilder;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.core.Context;
...
@Path("/users/")
public class UsersResource {

    @Context UriInfo uriInfo;

    ...

    @GET
    @Produces("application/json")
    public JSONArray getUsersAsJsonArray() {
        JSONArray uriArray = new JSONArray();
        for (UserEntity userEntity : getUsers()) {
            UriBuilder ub = uriInfo.getAbsolutePathBuilder();
            URI userUri = ub.path(userEntity.getUserid()).build();
            uriArray.put(userUri.toASCIIString());
        }
        return uriArray;
    }
}
```

2.11 Using Conditional GETs

A *conditional GET* enables you to evaluate one or more preconditions before processing a GET request. If the preconditions are met, a `Not Modified (304)` response can be returned rather than the normal response, potentially reducing bandwidth and improving server performance.

JAX-RS provides the `javax.ws.rs.core.Request` contextual interface enabling you to perform conditional GETs. You call the `evaluatePreconditions()` method and pass a

javax.ws.rs.core.EntityTag, the last modified timestamp (as a java.util.Date object), or both. The values are compared to the If-None-Match or If-Not-Modified headers, respectively, if these headers are sent with the request.

If headers are included with the request and the precondition values match the header values, then the evaluatePreconditions() methods returns a predefined ResponseBuilder response with a status code of Not Modified (304). If the precondition values do no match, the evaluatePreconditions() method returns null and the normal response is returned, with 200, OK status.

Example 2–19 shows how to pass the EntityTag to the evaluatePreconditions() method and build the response based on whether the preconditions are met.

Example 2–19 Using Conditional GETs

```
...
@Path("/employee/{joiningdate}")
public class Employee {

    Date joiningdate;
    @GET
    @Produces("application/xml")
    public Employee(@PathParam("joiningdate") Date joiningdate, @Context Request req,
            @Context UriInfo ui) {

        this.joiningdate = joiningdate;
        ...
        this.tag = computeEntityTag(ui.getRequestUri());
        if (req.getMethod().equals("GET")) {
            Response.ResponseBuilder rb = req.evaluatePreconditions(tag);
            // Preconditions met
            if (rb != null) {
                return rb.build();
            }
            // Preconditions not met
            rb = Response.ok();
            rb.tag(tag);
            return rb.build();
        }
    }
}
```

2.12 Accessing the WADL

The Web Application Description Language (WADL) is an XML-based file format that describes your RESTful Web services application. By default, a basic WADL is generated at runtime and can be accessed from your RESTful Web service by issuing a GET on the /application.wadl resource at the base URI of your RESTful application. For example:

```
GET http://<path_to_REST_app>/application.wadl
```

Alternatively, you can use the OPTIONS method to return the WADL for particular resource.

Example 2–20 shows an example of a WADL for the simple RESTful Web service shown in Example 2–1.

Example 2–20 Example of a WADL

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
   <doc xmlns:jersey="http://jersey.dev.java.net/"
             jersey:generatedBy="Jersey: 0.10-ea-SNAPSHOT 08/27/2008 08:24 PM"/>
   <resources base="http://localhost:9998/">
      <resource path="/helloworld">
         <method name="GET" id="sayHello">
            <response>
               <representation mediaType="text/plain"/>
            </response>
         </method>
      </resource>
   </resources>
</application>
```

2.13 More Advanced RESTful Web Service Tasks

For more information about advanced RESTful Web service development tasks, including those listed below, see the *Jersey 1.13 User Guide* at http://jersey.java.net/nonav/documentation/1.13/user-guide.html.

- Integrating JAX-RS with EJB technology and Contexts and Dependency Injection (CDI)
- Using JAXB and JSON

3 Developing RESTful Web Service Clients

This chapter describes how to develop WebLogic Web service clients that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

This chapter includes the following sections:

For information about developing RESTful Web service clients using Oracle JDeveloper, see "Creating RESTful Web Services and Clients" in *Developing Applications with Oracle JDeveloper*.

3.1 About RESTful Web Service Client Development

The Jersey JAX-RS RI provides a client API for developing RESTful Web services clients. To access the client API, you create an instance of the `weblogic.jaxrs.api.client.Client` class and then use that instance to access the Web resource and send HTTP requests.

> **Note:** A standard client API will be supported as part of the JSR-311 JAX-RS 2.0 specification.

The following sections provide more information about RESTful Web service client development:

3.1.1 Summary of Tasks to Develop RESTful Web Service Clients

The following table summarizes a subset of the tasks that are required to develop RESTful Web service clients. For more information about advanced tasks, see Section 3.6, "More Advanced RESTful Web Service Client Tasks."

Table 3–1 Summary of Tasks to Develop RESTful Web Service Clients

Task	More Information
Create an instance of the `weblogic.jaxrs.api.client.Client` class.	Section 3.2, "Creating and Configuring a Client Instance"
Create an instance of the Web resource.	Section 3.3, "Creating a Web Resource Instance"
Send requests to the resource. For example, HTTP requests to GET, PUT, POST, and DELETE resource information.	Section 3.4, "Sending Requests to the Resource"
Receive responses from the resource.	Section 3.5, "Receiving a Response from a Resource"

3.1.2 Example of a RESTful Web Service Client

The following provides a simple example of a RESTful Web service client that can be used to call the RESTful Web service defined in Example 2–1, "Simple RESTful Web Service". In this example:

- The `Client` instance is created to access the client API. For more information, see Section 3.2, "Creating and Configuring a Client Instance."
- The `WebResource` instance is created to access the Web resource. For more information, see Section 3.3, "Creating a Web Resource Instance."
- A `get` request is sent to the resource. For more information, see Section 3.4, "Sending Requests to the Resource."
- The response is returned as a String value. For more information about receiving the response, see Section 3.5, "Receiving a Response from a Resource."

Additional examples are listed in Section 1.6, "Learn More About RESTful Web Services."

Example 3–1 Simple RESTful Web Service Client

```
package samples.helloworld.client;

import weblogic.jaxrs.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);
    }
}
```

3.2 Creating and Configuring a Client Instance

To access the Jersey JAX-RS RI client API, create an instance of the `weblogic.jaxrs.api.client.Client` class.

Note: Alternatively, you can create an instance of the `com.sun.jersey.api.client.Client` class. However, you will not be able to take advantage of the Oracle extensions, such as securing the RESTful client using Oracle Web Services Manager (OWSM) policies, as described in Section 5.2, "Securing RESTful Web Services and Clients Using OWSM Policies."

Optionally, you can pass client configuration properties, defined in Table 3–2, when creating the client instance by defining a `com.sun.jersey.api.client.config.ClientConfig` and passing the information to the `create` method. For more information, see `https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/config/ClientConfig.html`.

Table 3–2 RESTful Web Service Client Configuration Properties

Property	Description
`PROPERTY_BUFFER_RESPONSE_ENTITY_ON_EXCEPTION`	Boolean value that specifies whether the client should buffer the response entity, if any, and close resources when a `UniformInterfaceException` is thrown. This property defaults to `true`.
`PROPERTY_CHUNKED_ENCODING_SIZE`	Integer value that specifies the chunked encoding size. A value equal to or less than 0 specifies that the default chunk size should be used. If not set, then chunking will not be used.
`PROPERTY_CONNECT_TIMEOUT`	Integer value that specifies the connect timeout interval in milliseconds. If the property is 0 or not set, then the interval is set to infinity.
`PROPERTY_FOLLOW_REDIRECTS`	Boolean value that specifies whether the URL will redirect automatically to the URI declared in `3xx` responses. This property defaults to `true`.
`PROPERTY_READ_TIMEOUT`	Integer value that specifies the read timeout interval in milliseconds. If the property is 0 or not set, then the interval is set to infinity.

Example 3–2 provides an example of how to create a client instance.

Example 3–2 Creating a Client Instance

```
import weblogic.jaxrs.api.client.Client;
...
    public static void main(String[] args) {
        Client c = Client.create();
...
```

Example 3–3 provides an example of how to create a client instance and pass configuration properties to the `create` method.

Example 3–3 Creating and Configuring a Client Instance

```
import com.sun.jersey.api.client.*;
import weblogic.jaxrs.api.client.Client;
...
    public static void main(String[] args) {
        ClientConfig cc = new DefaultClientConfig();
        cc.getProperties().put(ClientConfig.PROPERTY_FOLLOW_REDIRECTS, true);
```

```
        Client c = Client.create(cc);
...
```

Alternatively, you can configure a client instance after the client has been created, by setting properties on the map returned from the `getProperties` method or calling a specific setter method.

Example 3–4 provides an example of how to configure a client after it has been created. In this example:

- `PROPERTY_FOLLOW_REDIRECTS` is configured by setting the property on the map returned from the `getProperties` method.
- `PROPERTY_CONNECT_TIMEOUT` is configured using the setter method.

Example 3–4 Configuring a Client Instance After It Has Been Created

```
import com.sun.jersey.api.client.*;
import weblogic.jaxrs.api.client.Client;
...
    public static void main(String[] args) {
        Client c = Client.create();
        c.getProperties().put(ClientConfig.PROPERTY_FOLLOW_REDIRECTS, true);
        c.setConnectTimeout(3000);
...
```

Example 3–5 provides an example of how to configure a client instance to use basic authentication.

Example 3–5 Configuring a Client Instance to Use Basic Authentication

```
import javax.ws.rs.core.MediaType;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.filter.HTTPBasicAuthFilter;
...
      Client c = Client.create();
      c.addFilter(new HTTPBasicAuthFilter("weblogic", "weblogic1"));
      WebResource resource =
c.resource("http://localhost:7001/management/tenant-monitoring/datasources/JDBC%20Data%20Source-0")
;
      String response = resource.accept("application/json").get(String.class); //application/xml
      // resource.accept(MediaType.APPLICATION_JSON_TYPE).get(String.class);
      System.out.println(response);
...
```

3.3 Creating a Web Resource Instance

Before you can issue requests to a RESTful Web service, you must create an instance of `com.sun.jersey.api.client.WebResource` or `com.sun.jersey.api.client.AsyncWebResource` to access the resource specified by the URI. The `WebResource` or `AsyncWebResource` instance inherits the configuration defined for the client instance. For more information, see:

- `WebResource`: https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/WebResource.html

- AsyncWebResource: https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/AsyncWebResource.html

> **Note:** Because clients instances are expensive resources, if you are creating multiple Web resources, it is recommended that you re-use a single client instance whenever possible.

Example 3–6 provides an example of how to create an instance to a Web resource hosted at `http://example.com/helloworld`.

Example 3–6 Creating a Web Resource Instance

```
import com.sun.jersey.api.client.*;
import weblogic.jaxrs.api.client.Client;
...
    public static void main(String[] args) {\
...
        Client c = Client.create();
        WebResource resource = c.resource("http://example.com/helloWorld");
...
```

Example 3–6 provides an example of how to create an instance to an asynchronous Web resource hosted at `http://example.com/helloworld`.

Example 3–7 Creating an Asynchronous Web Resource Instance

```
import com.sun.jersey.api.client.*;
import weblogic.jaxrs.api.client.Client;
...
    public static void main(String[] args) {\
...
        Client c = Client.create();
         AsyncWebResource asyncResource = c.resource("http://example.com/helloWorld");
...
```

3.4 Sending Requests to the Resource

Use the `WebResource` or `AsyncWebResource` instance to build requests to the associated Web resource, as described in the following sections:

- Section 3.4.1, "How to Build Requests"
- Section 3.4.2, "How to Send HTTP Requests"
- Section 3.4.4, "How to Configure the Accept Header"
- Section 3.4.3, "How to Pass Query Parameters"

3.4.1 How to Build Requests

Requests to a Web resource are structured using the builder pattern, as defined by the `com.sun.jersey.api.client.RequestBuilder` interface. The `RequestBuilder` interface is implemented by `com.sun.jersey.api.client.WebResource`, `com.sun.jersey.api.client.AsyncWebResource`, and other resource classes.

You can build a request using the methods defined in Table 3–3, followed by the HTTP request method, as described in Section 3.4.2, "How to Send HTTP Requests." Examples of how to build a request are provided in the sections that follow.

For more information about `RequestBuilder` and its methods, see https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/RequestBuilder.html.

Table 3–3 Building a Request

Method	Description
`accept()`	Defines the acceptable media types. See Section 3.4.4, "How to Configure the Accept Header."
`acceptLanguage()`	Defines the acceptable languages using the `acceptLanguage` method.
`cookie()`	Adds a cookie to be set.
`entity()`	Configures the request entity. See Section 3.4.6, "How to Configure the Request Entity."
`header()`	Adds an HTTP header and value. See Section 3.4.4, "How to Configure the Accept Header."
`type()`	Configures the media type. See Section 3.4.6, "How to Configure the Request Entity."

3.4.2 How to Send HTTP Requests

Table 3–4 list the `WebResource` and `AsyncWebResource` methods that can be used to send HTTP requests.

In the case of `AsyncWebResource`, a `java.util.concurrent.Future<V>` object is returned, which can be used to access the result of the computation later, without blocking execution. For more information about Future<V>, see http://docs.oracle.com/javase/7/docs/api/index.html?java/util/concurrent/Future.html.

Table 3–4 WebResource Methods to Send HTTP Requests

Method	Description
`get()`	Invoke the HTTP GET method to get a representation of the resource.
`post()`	Invoke the HTTP POST method to create or update the representation of the specified resource.
`put()`	Invoke the HTTP PUT method to update the representation of the resource.
`delete()`	Invoke the HTTP DELETE method to delete the representation of the resource.

If the response has an entity (or representation), then the Java type of the instance required is declared in the HTTP method.

Example 3–8 provides an example of how to send an HTTP GET request. In this example, the response entity is requested to be an instance of `String`. The response entity will be de-serialized to a `String` instance.

Example 3–8 Sending an HTTP GET Request

```
import com.sun.jersey.api.client.WebResource;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
```

```
        String response = resource.get(String.class);
...
```

Example 3–9 provides an example of how to send an HTTP PUT request and put the entity `foo:bar` into the Web resource. In this example, the response entity is requested to be an instance of `com.sun.jersey.api.client.ClientResponse`.

Example 3–9 Sending an HTTP PUT Request

```
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.ClientResponse;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
        ClientResponse response = resource.put(ClientResponse.class, "foo:bar");
...
```

If you wish to send an HTTP request using a generic type, to avoid type erasure at runtime, you need to create a `com.sun.jersey.api.client.GenericType` object to preserve the generic type. For more information, see https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/GenericType.html.

Example 3–10 provides an example of how to send an HTTP request using a generic type using `GenericType` to preserve the generic type.

Example 3–10 Sending an HTTP GET Request Using a Generic Type

```
import com.sun.jersey.api.client.WebResource;
...
    public static void main(String[] args) {
...
       WebResource resource = c.resource("http://example.com/helloWorld");
       List<String> list = resource.get(new GenericType<List<String>>() {});
...
```

3.4.3 How to Pass Query Parameters

You can pass query parameters in the GET request by defining a `javax.ws.rs.core.MultivaluedMap` and using the `queryParams` method on the Web resource to pass the map as part of the HTTP request.

For more information about `MultivaluedMap`, see http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/MultivaluedMap.html.

Example 3–11 provides an example of how to pass parameters in a GET request to a Web resource hosted at `http://example.com/helloworld`, resulting in the following request URI: `http://example.com/base?param1=val1&param2=val2`

Example 3–11 Passing Query Parameters

```
import com.sun.jersey.api.client.WebResource;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.MultivaluedMapImpl;
...
    public static void main(String[] args) {
...
```

```
        WebResource resource = c.resource("http://example.com/helloWorld");
        MultivaluedMap queryParams = new MultivaluedMapImpl();
        queryParams.add("param1", "val1");
        queryParams.add("param2", "val2");
        String response = resource.queryParams(queryParams).get(String.class);
...
```

3.4.4 How to Configure the Accept Header

Configure the `Accept` header for the request using the `accept` method on the Web resource.

Example 3–12 provides an example of how to specify `text/plain` as the acceptable MIME media type in a GET request to a Web resource hosted at `http://example.com/helloworld`.

Example 3–12 Configuring the Accept Header

```
import com.sun.jersey.api.client.WebResource;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
        String response = resource.accept("text/plain").get(String.class);
...
```

3.4.5 How to Add a Custom Header

Add a custom header to the request using the `header` method on the Web resource.

Example 3–13 provides an example of how to add a custom header `FOO` with the value `BAR` in a GET request to a Web resource hosted at `http://example.com/helloworld`.

Example 3–13 Adding a Custom Header

```
import com.sun.jersey.api.client.WebResource;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
        String response = resource.header("FOO", "BAR").get(String.class);
...
```

3.4.6 How to Configure the Request Entity

Configure the request entity and type using the `entity` method on the Web resource. Alternatively, you can configure the request entity type only using the `type` method on the Web resource.

Example 3–14 provides an example of how to configure a request entity and type.

Example 3–14 Configuring the Request Entity

```
import com.sun.jersey.api.client.WebResource;
...
    public static void main(String[] args) {
...
```

```
        WebResource resource = c.resource("http://example.com/helloWorld");
        String response = resource.entity(request, MediaType.TEXT_PLAIN_TYPE).get(String.class);
...
```

Example 3–15 provides an example of how to configure the request entity media type only.

Example 3–15 Configuring the Request Entity Media Type Only

```
import com.sun.jersey.api.client.WebResource;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
        String response = resource.type(MediaType.TEXT_PLAIN_TYPE).get(String.class);
...
```

3.5 Receiving a Response from a Resource

You define the Java type of the entity (or representation) in the response when you call the HTTP method, as described in Section 3.4.2, "How to Send HTTP Requests."

If response metadata is required, declare the Java type `com.sun.jersey.api.client.ClientResponse` as the response type. the `ClientResponse` type enables you to access status, headers, and entity information.

The following sections describes the response metadata that you can access using the `ClientResponse`. For more information about `ClientResponse`, see `https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/ClientResponse.html`.

- Section 3.5.1, "How to Access the Status of Request"
- Section 3.5.2, "How to Get the Response Entity"

3.5.1 How to Access the Status of Request

Access the status of a client response using the `getStatus` method on the `ClientResponse` object. For a list of valid status codes, see `https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/client/ClientResponse.Status.html`.

Example 3–12 provides an example of how to access the status code of the response.

Example 3–16 Accessing the Status of the Request

```
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.ClientResponse;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
        ClientResponse response = resource.get(ClientResponse.class);
        int status = response.getStatus();
...
```

3.5.2 How to Get the Response Entity

Get the response entity using the `getEntity` method on the `ClientResponse` object.

Example 3–12 provides an example of how to get the response entity.

Example 3–17 Getting the Response Entity

```
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.ClientResponse;
...
    public static void main(String[] args) {
...
        WebResource resource = c.resource("http://example.com/helloWorld");
        ClientResponse response = resource.get(ClientResponse.class);
        String entity = response.getEntity(String.class);
...
```

3.6 More Advanced RESTful Web Service Client Tasks

For more information about advanced RESTful Web service client tasks, including those listed below, see the *Jersey 1.13 User Guide* at https://jersey.java.net/nonav/documentation/1.13/user-guide.html.

- Adding new representation types
- Using filters
- Enabling security with HTTP(s) URLConnection

3.7 Invoking a RESTful Web Service from a Standalone Client

When invoking a RESTful Web service from an environment that does not have Oracle Fusion Middleware or WebLogic Server installed locally, with the entire set of Oracle Fusion Middleware or WebLogic Server classes in the CLASSPATH, you can use the standalone client JAR file when invoking the Web service.

The standalone client JAR supports basic JAX-RS 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. For more information, see "Developing and Securing Web Services and Clients" in *Developing Applications with Oracle JDeveloper*.
2. Copy the file *ORACLE_HOME*`/oracle_common/modules/clients/com.oracle.jersey.fmw.client_12.1.2.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.

> **Note:** Ensure that your CLASSPATH includes the JAR file that contains the Ant classes (`ant.jar`) as a subset are used by the standalone client JAR files. This JAR file is typically located in the `lib` directory of the Ant distribution.

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 RESTful client.

   The configuration steps required vary based on the type of policy being attached. Examples are provided below. For additional configuration requirements, see "Configuring Java SE Applications to Use OPSS" in *Securing Applications with Oracle Platform Security Services*.

   > **Note:** To access the Jersey JAX-RS RI client API, ensure that you create an instance of the `weblogic.jaxrs.api.client.Client` class.
   >
   > Alternatively, you can create an instance of the `com.sun.jersey.api.client.Client` class. However, you will not be able to take advantage of the Oracle extensions, such as securing the RESTful client using Oracle Web Services Manager (OWSM) policies, as described in Section 5.2, "Securing RESTful Web Services and Clients Using OWSM Policies."

   Example: Basic Authentication

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

   a. 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 RESTful client.

   b. Create a wallet (`cwallet.sso`) in the same location that you copied the files in step 2 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>`.For more information, see "Using a Wallet-based Credential Store" in *Securing Applications with Oracle Platform Security Services*.

   c. On the Java command line, pass the following property defining the JPS configuration file copied in step 1:

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

      For more information, see "Scenario 3: Securing a Java SE Application" in *Securing Applications with Oracle Platform Security Services*.

   Example: SSL

   For example, to support SSL policies, perform the following steps:

   a. 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 RESTful client.

b. 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>
```

For more information, see "Scenario 3: Securing a Java SE Application" in *Securing Applications with Oracle Platform Security Services*.

Define the trust store containing the trusted certificates:

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

For more information, see "Setting Up the WebLogic Server in Case of a Java SE Application" in "Setting Up a One-Way SSL Connection to the LDAP" in *Securing Applications with Oracle Platform Security Services*.

Define the trust store password:

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

4

Packaging and Deploying RESTful Web Services

This chapter describes how to package and deploy WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

This chapter includes the following topics:

- Section 4.1, "About RESTful Web Service Packaging and Deployment"
- Section 4.2, "Packaging With an Application Subclass"
- Section 4.3, "Packaging With a Servlet"
- Section 4.4, "Packaging as a Default Resource"

4.1 About RESTful Web Service Packaging and Deployment

All RESTful Web service applications must be packaged as part of a Web application. If your Web service is implemented as an EJB, it must be packaged and deployed within a WAR. For more information about deploying a Web application, see "Understanding WebLogic Server Deployment" in *Deploying Applications to Oracle WebLogic Server*.

Table 4–1 summarizes the specific packaging options available for RESTful Web service applications.

Table 4–1 Packaging Options for RESTful Web Service Applications

Packaging Option	Description
Application subclass	Define a class that extends `javax.ws.rs.core.Application` to define the components of a RESTful Web service application deployment and provide additional metadata. You can add a `javax.ws.rs.ApplicationPath` annotation to the subclass to configure the servlet context path. For more information, see Section 4.2, "Packaging With an Application Subclass."
Servlet	Update the `web.xml` deployment descriptor to configure the servlet and mappings. The method used depends on whether your Web application is using Servlet 3.0 or earlier. For more information, see Section 4.3, "Packaging With a Servlet."
Default resource	If you do not configure the servlet context path in your configuration using either of the options specified above, the WebLogic Server provides a default RESTful Web service application servlet context path, `resources`. For more information, see Section 4.4, "Packaging as a Default Resource."

4.2 Packaging With an Application Subclass

In this packaging scenario, you 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 `http://docs.oracle.com/javaee/6/api/index.html?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 how this information is used in the base URI of the resource, see Section 2.3.3, "What Happens at Runtime: How the Base URI is Constructed." For more information about the `@ApplicationPath` annotation, see the Javadoc at: `http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/ApplicationPath.html`.

For simple deployments, no `web.xml` deployment descriptor is required. For more complex deployments, for example to secure the Web service or specify initialization parameters, you can package a `web.xml` deployment descriptor with your application, as described in Section 4.3, "Packaging With a Servlet."

Example 4–1 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.

Example 4–1 Example of a Class that Extends javax.ws.rs.core.Application

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

4.3 Packaging With a Servlet

The following sections describe how to package the RESTful Web service application with a servlet using the `web.xml` deployment descriptor, based on whether your Web application is using Servlet 3.0 or earlier.

- Section 4.3.1, "How to Package the RESTful Web Service Application with Servlet 3.0"
- Section 4.3.2, "How to Package the RESTful Web Service Application with Pre-3.0 Servlets"

The `web.xml` file is located in the `WEB-INF` directory in the root directory of your application archive. 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*.

> **Note:** If you are running a Web application that uses pre-3.0 Servlets, see Section 4.3, "Packaging With a Servlet." This section applies only to Web applications running Servlets 3.0.

4.3.1 How to Package the RESTful Web Service Application with Servlet 3.0

To package the RESTful Web Service application with Servlet 3.0, update the `web.xml` deployment descriptor to define the elements defined in the following sections. The elements vary depending on whether you include in the package a class that extends `javax.ws.rs.core.Application`.

- Section 4.3.1.1, "Packaging the RESTful Web Service Application Using web.xml With Application Subclass"
- Section 4.3.1.2, "Packaging the RESTful Web Service Application Using web.xml Without Application Subclass"

For more information about any of the elements, see "servlet" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

4.3.1.1 Packaging the RESTful Web Service Application Using web.xml With Application Subclass

If a class that extends `javax.ws.rs.core.Application` is packaged with `web.xml`, then define the elements as described in Table 4–2. For an example, see Example 4–2.

Table 4–2 Packaging the RESTful Web Service Application Using web.xml With Application Subclass

Element	Description
`<servlet-name>`	Set this element to the fully qualified name of the class that extends `javax.ws.rs.core.Application`. You can specify multiple servlet entries to define multiple `Application` subclass names.
`<servlet-class>`	Not required.

`<init-param>`	Set this element to define the class that extends the `javax.ws.rs.core.Application`:
	`<init-param>`
	`  <param-name>`
	`  javax.ws.rs.Application`
	`  </param-name>`
	`  <param-value>`
	`  *ApplicationSubclassName*`
	`  </param-value>`
	`</init-param>`

Table 4–2 (Cont.) Packaging the RESTful Web Service Application Using web.xml With Application

Element	Description

<servlet-mapping	Set as the base URI pattern that gets mapped to the servlet.
	If not specified, one of the following values are used, in order of precedence:
	■ @ApplicationPath annotation value defined in the javax.ws.rs.core.Application subclass. For example:
	`package test;`
	`@ApplicationPath("res")`
	`public class MyJaxRsApplication extends java.ws.rs.core.Application`
	`...`
	For more information, see Section 4.2, "Packaging With an Application Subclass."
	■ The value resources. This is the default base URI pattern for RESTful Web service applications. For more information, see Section 4.4, "Packaging as a Default Resource."
	If both the <servlet-mapping> and @ApplicationPath are specified, the <servlet-mapping> takes precedence.
	For more information about how this information is used in the base URI of the resource, see Section 2.3.3, "What Happens at Runtime: How the Base URI is Constructed."

The following provides an example of how to update the web.xml file if a class that extends javax.ws.rs.core.Application is packaged with web.xml.

Example 4–2 Updating web.xml for Servlet 3.0 If Application Subclass is in Package

```
<web-app>
    <servlet>
        <display-name>My JAX-RS Servlet</display-name>
        <servlet-name>jersey.samples.MyApplication</servlet-name>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>myPackage.myJaxRsApplication</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyJaxRsApp</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

4.3.1.2 Packaging the RESTful Web Service Application Using web.xml Without Application Subclass

If a class that extends javax.ws.rs.core.Application is **not** packaged with web.xml, then define the elements as described in Table 4–3.

> **Note:** In this scenario, you cannot support multiple RESTful Web service applications.

Table 4–3 Packaging the RESTful Web Service Application Using web.xml Without Application Subclass

Element	Description
<servlet-name>	Set this element to the desired servlet name.

Table 4–3 (Cont.) Packaging the RESTful Web Service Application Using web.xml Without Application

Element	Description
`<servlet-class>`	Set this element to one of the following classes to delegate all Web requests to the Jersey servlet: ■ `weblogic.jaxrs.server.portable.servlet.ServletContainer` ■ `com.sun.jersey.spi.container.servlet.ServletContainer`

`<init-param>`	Set this element to define the class that extends the `javax.ws.rs.Application`:
	`<init-param>`
	`  <param-name>`
	`  javax.ws.rs.Application`
	`  </param-name>`
	`  <param-value>`
	`  *ApplicationSubclassName*`
	`  </param-value>`
	`</init-param>`

`<servlet-mapping`	Set as the base URI pattern that gets mapped to the servlet. If not specified, this value defaults to `resources`. For more information, see Section 4.4, "Packaging as a Default Resource." For more information about how this information is used in the base URI of the resource, see Section 2.3.3, "What Happens at Runtime: How the Base URI is Constructed."

The following provides an example of how to update the `web.xml` file if an class that extends `javax.ws.rs.core.Application` is **not** packaged with `web.xml`.

Example 4–3 Updating web.xml for Servlet 3.0 If Application Subclass is Not in Package

```
<web-app>
    <servlet>
        <servlet-name>RestServlet</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>RestServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

4.3.2 How to Package the RESTful Web Service Application with Pre-3.0 Servlets

Table 4–4 describes the elements to update in the `web.xml` deployment descriptor to package the RESTful Web service application with a pre-3.0 servlet.

Table 4–4 Packaging the RESTful Web Service Application with Pre-3.0 Servlets

Element	Description
`<servlet-name>`	Set this element to the desired servlet name.
`<servlet-class>`	Set this element to one of the following classes to delegate all Web requests to the Jersey servlet: ■ `weblogic.jaxrs.server.portable.servlet.ServletContainer` ■ `com.sun.jersey.spi.container.servlet.ServletContainer`

Table 4–4 (Cont.) Packaging the RESTful Web Service Application with Pre-3.0 Servlets

Element	Description

`<init-param>`	Set this element to define the class that extends the `javax.ws.rs.core.Application`:
	`<init-param>`
	`  <param-name>`
	`  javax.ws.rs.Application`
	`  </param-name>`
	`  <param-value>`
	`  *ApplicationSubclassName*`
	`  </param-value>`
	`</init-param>`
	
	Alternatively, you can declare the packages in your application, as follows:
	`<init-param>`
	`  <param-name>`
	`  com.sun.jersey.config.property.packages`
	`  </param-name>`
	`  <param-value>`
	`  ***project1***`
	`  </param-value>`
	`</init-param>`

`<servlet-mapping`	Set as the base URI pattern that gets mapped to the servlet.
	If not specified, one of the following values are used, in order of precedence:
	■ `@ApplicationPath` annotation value defined in the `javax.ws.rs.core.Application` subclass. For example:
	`package test;`
	`@ApplicationPath("res")`
	`public class MyJaxRsApplication extends java.ws.rs.core.Application`
	`...`
	
	For more information, see Section 4.2, "Packaging With an Application Subclass."
	■ The value `resources`. This is the default base URI pattern for RESTful Web service applications. For more information, see Section 4.4, "Packaging as a Default Resource."
	If both the `<servlet-mapping>` and `@ApplicationPath` are specified, the `<servlet-mapping>` takes precedence.
	For more information about how this information is used in the base URI of the resource, see Section 2.3.3, "What Happens at Runtime: How the Base URI is Constructed."

The following provides an example of how to update the `web.xml` file if an class that extends `javax.ws.rs.core.Application` is **not** packaged with `web.xml`.

Example 4–4 Updating web.xml for Pre-3.0 Servlets

```
<web-app>
    <servlet>
        <display-name>My JAX-RS Servlet</display-name>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <servlet-class>weblogic.jaxrs.server.portable.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.resourceConfigClass</param-name>
            <param-value>com.sun.jersey.api.core.PackagesResourceConfig</param-value>
        </init-param>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param-name>
```

```
            <param-value>project1</param-value>
        </init-param>
    </servlet>
</web-app>
```

4.4 Packaging as a Default Resource

By default, WebLogic Server defines a default RESTful Web service application context path, `resources`. The default RESTful Web service application context path is used if the following are true:

- You did not update the `web.xml` deployment descriptor to include a Servlet mapping, as described in Section 4.3, "Packaging With a Servlet."
- The `@ApplicationPath` annotation is not defined in the `javax.ws.rs.core.Application` subclass, as described in Section 4.2, "Packaging With an Application Subclass."

> **Note:** If a servlet is already registered at the default context path, then a warning is issued.

For example, if the relative URI of the root resource class for the RESTful Web service application is defined as `@Path('/helloworld')` and the default RESTful Web service application context path is used, then the RESTful Web service application resource will be available at:

`http://<host>:<port>/contextPath/resources/helloworld`

5

Securing RESTful Web Services and Clients

This chapter describes how to secure WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

This chapter includes the following sections:

For information about developing RESTful Web service clients using Oracle JDeveloper, see "How to Attach Policies to RESTful Web Services and Clients" in *Developing Applications with Oracle JDeveloper*.

5.1 About RESTful Web Service Security

You can secure your RESTful Web services using one of the following methods to support authentication, authorization, or encryption:

- Attaching Oracle Web Services Manager (OWSM) policies. See Section 5.2, "Securing RESTful Web Services and Clients Using OWSM Policies".
- Updating the `web.xml` deployment descriptor to access information about the authenticated users. See Section 5.3, "Securing RESTful Web Services Using web.xml".
- Using the `javax.ws.rs.core.SecurityContext` interface to access security-related information for a request. See Section 5.4, "Securing RESTful Web Services Using SecurityContext".
- Applying annotations to your JAX-RS classes. See Section 5.5, "Securing RESTful Web Services Using Java Security Annotations".

5.2 Securing RESTful Web Services and Clients Using OWSM Policies

Only a subset of OWSM security policies are supported for RESTful Web services, as described in "Which OWSM Policies Are Supported for RESTful Web Services" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

You can attach OWSM security policies to RESTful Web services using one of the following methods:

- Programmatically, at design time, as described in "Attaching Policies to RESTful Web Services and Clients at Design Time" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
- Post-deployment, both directly and globally, using:
  - Fusion Middleware Control, as described in "Attaching Policies Using Fusion Middleware Control" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
  - WLST, as described in "Attaching Policies Using WLST" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

Example 5–1 provides an example of using WLST to attach the `oracle/http_basic_auth_over_ssl_service_policy` policy to a RESTful service. For more information, see "Attaching Policies Directly Using WLST" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.

Example 5–1 Securing RESTful Web Services Using OWSM Policies With WLST

```
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> beginWSMSession()
Location changed to domainRuntime tree. This is a read-only tree with DomainMBean as the root.
For more help, use help('domainRuntime')

Session started for modification.
wls:/my_domain/serverConfig> selectWSMPolicySubject('weblogic/my_domain/jaxrs_pack', '#jaxrs_
pack.war', 'REST-Resource(Jersey)')

The policy subject is selected for modification.

wls:/my_domain/serverConfig> attachWSMPolicy('oracle/http_basic_auth_over_ssl_service_policy')

Policy reference "oracle/http_basic_auth_over_ssl_service_policy" added.

wls:/my_domain/serverConfig> commitWSMSession()

The policy set for subject "/weblogic/my_domain/jaxrs_pack|#jaxrs_pack.war|REST-Resource(Jersey)"
was saved successfully.
```

5.3 Securing RESTful Web Services Using web.xml

You secure RESTful Web services using the `web.xml` deployment descriptor as you would for other Java EE Web applications. For complete details, see:

- "Developing Secure Web Applications" in *Developing Applications with the WebLogic Security Service*.
- "Securing Web Applications" in *The Java EE 6 Tutorial* at: http://docs.oracle.com/javaee/6/tutorial/doc/gkbaa.html

For example, to secure your RESTful Web service using basic authentication, perform the following steps:

1. Define a `<security-constraint>` for each set of RESTful resources (URIs) that you plan to protect.
2. Use the `<login-config>` element to define the type of authentication you want to use and the security realm to which the security constraints will be applied.
3. Define one or more security roles using the `<security-role>` tag and map them to the security constraints defined in step 1. For more information, see "security-role" in *Developing Applications with the WebLogic Security Service*.
4. To enable encryption, add the `<user-data-constraint>` element and set the `<transport-guarantee>` subelement to `CONFIDENTIAL`. For more information, see "user-data-constraint" in *Developing Applications with the WebLogic Security Service*.

The following shows an example of how to secure a RESTful Web service using basic authentication.

Example 5–2 Securing RESTful Web Services Using Basic Authentication

```
<web-app>
    <servlet>
        <servlet-name>RestServlet</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>RestServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <security-constraint>
         <web-resource-collection>
             <web-resource-name>Orders</web-resource-name>
             <url-pattern>/orders</url-pattern>
             <http-method>GET</http-method>
             <http-method>POST</http-method>
         </web-resource-collection>
         <auth-constraint>
             <role-name>admin</role-name>
         </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>default</realm-name>
    </login-config>
    <security-role>
        <role-name>admin</role-name>
    </security-role>
</web-app>
```

5.4 Securing RESTful Web Services Using SecurityContext

The `javax.ws.rs.core.SecurityContext` interface provides access to security-related information for a request. The `SecurityContext` provides functionality similar to `javax.servlet.http.HttpServletRequest`, enabling you to access the following security-related information:

- `java.security.Principal` object containing the name of the user making the request.

- Authentication type used to secure the resource, such as BASIC_AUTH, FORM_AUTH, and CLIENT_CERT_AUTH.
- Whether the authenticated user is included in a particular role.
- Whether the request was made using a secure channel, such as HTTPS.

You access the `SecurityContext` by injecting an instance into a class field, setter method, or method parameter using the `javax.ws.rs.core.Context` annotation.

For more information, see the Javadoc at:

- `SecurityContext` interface: http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/SecurityContext.html
- `@Context` annotation: http://docs.oracle.com/javaee/6/api/index.html?javax/ws/rs/core/Context.html

Example 5–3 shows how to inject an instance of `SecurityContext` into the `sc` method parameter using the `@Context` annotation, and check whether the authorized user is included in the `admin` role before returning the response.

Example 5–3 Securing RESTful Web Service Using SecurityContext

```
package samples.helloworld;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.Context;

...

@Path("/stateless")
@Stateless(name = "JaxRSStatelessEJB")
public class StlsEJBApp {
...
        @GET
        @Produces("text/plain;charset=UTF-8")
        @Path("/hello")
        public String sayHello(@Context SecurityContext sc) {
                if (sc.isUserInRole("admin"))  return "Hello World!";
                throw new SecurityException("User is unauthorized.");
        }
```

5.5 Securing RESTful Web Services Using Java Security Annotations

The `javax.annotation.security` package provides annotations, defined in Table 5–1, that you can use to secure your RESTful Web services. For more information, see:

- "Specifying Authorized Users by Declaring Security Roles" in *The Java EE Tutorial*.
- `javax.annotation.security` Javadoc

Table 5–1 Annotations for Securing RESTful Web Services

Annotation	Description
`DenyAll`	Specifies that no security roles are allowed to invoke the specified methods.
`PermitAll`	Specifies that all security roles are allowed to invoke the specified methods.
`RolesAllowed`	Specifies the list of security roles that are allowed to invoke the methods in the application.

Example 5–4 shows how to define the security roles that are allowed, by default, to access the methods defined in the `helloWorld` class. The sayHello method is annotated with the `@RolesAllows` annotation to override the default and only allow users that belong to the `ADMIN` security role.

Example 5–4 Securing RESTful Web Service Using Java Security Annotations

```
package samples.helloworld;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.annotation.Security.RolesAllowed;


@Path("/helloworld")
@RolesAllowed({"ADMIN", "ORG1"})
public class helloWorld {

   @GET
   @Path("sayHello")
   @Produces("text/plain")
   @RolesAllows("ADMIN")
   public String sayHello() {
      return "Hello World!";
   }
}
```

6

Testing RESTful Web Services

This chapter describes how to test WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

After you have deployed a Web application that contains a RESTful Web service to WebLogic Server, you can test your application.Table 6–1 lists the methods that can be employed to test your RESTful Web service.

Table 6–1 Methods for Testing RESTful Web Services

Method	Description
Enterprise Manager Fusion Middleware Control	Use the test interface provided with Enterprise Manager Fusion Middleware Control to test the RESTful Web service resource methods. For more information, see "Testing a RESTful Web Service" in *Securing Web Services and Managing Policies with Oracle Web Services Manager*.
Administration Console	Navigate to the Testing tab for your application deployment in the Administration Console to validate the application deployment and view the WADL file. For more information, see "Test RESTful Web services" in *Oracle WebLogic Server Administration Console Online Help*.

7

Monitoring RESTful Web Services and Clients

This chapter describes how to monitor WebLogic Web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

This chapter includes the following sections:

7.1 About Monitoring RESTful Web Services

To monitor your RESTful Web services, you can use one of the methods defined in Table 7–1.

Table 7–1 Methods for Monitoring RESTful Web Services

Method	Description
Fusion Middleware Control Enterprise Manager	Access run-time information and monitor run-time statistics, as described in Section 7.2, "Monitoring RESTful Web Services Using Enterprise Manager Fusion Middleware Control."
WebLogic Server Administration Console	Access run-time information and monitor run-time statistics, as described in Section 7.3, "Monitoring RESTful Web Services Using the Administration Console."
WebLogic Scripting Tool (WLST)	Access run-time information and monitor run-time statistics, as described in Section 7.4, "Monitoring RESTful Web Services Using WLST."
Logging filter	Monitor how a request is processed and dispatched to Jersey JAX-RS RI components, as described in Section 7.5, "Enabling the Tracing Feature."

7.2 Monitoring RESTful Web Services Using Enterprise Manager Fusion Middleware Control

Using Enterprise Manager Fusion Middleware Control, you can monitor statistics for your RESTful Applications and resources, such as error and invocation counts, execution times, and so on. For complete information, see "Monitoring Web Services" in *Administering Web Services*.

7.3 Monitoring RESTful Web Services Using the Administration Console

Using the Administration Console, you can monitor statistics for your RESTful Applications and resources, such as error and invocation counts, execution times, and so on.

To monitor your deployed RESTful Web services using the Administration Console, follow these steps:

1. Invoke the Administration Console in your browser using the following URL:

   `http://[host]:[port]/console`

   where:

   - `host` refers to the computer on which WebLogic Server is running.
   - `port` refers to the port number on which WebLogic Server is listening (default value is `7001`).

2. Follow the procedure described in "Monitor RESTful Web services" in *Oracle WebLogic Server Administration Console Online Help*.

7.4 Monitoring RESTful Web Services Using WLST

WebLogic Server provides several run-time MBeans, including those defined in Table 7–2, that capture run-time information and let you monitor run-time statistics for your RESTful Web service applications.

Table 7–2 Run-time MBeans for Monitoring RESTful Web Services

Run-time MBean	Description
`JaxRsApplication`	Displays monitoring information for the RESTful Web service application. For more information, see "JaxRsApplicationRuntimeBean" in the *WebLogic Server MBean Reference*.
`ResourceConfig`	Displays monitoring information about the RESTful Web service application resource configuration. For more information, see "JaxRsResourceConfigTypeRuntimeBean" in the *WebLogic Server MBean Reference*.
`RootResources`	Displays monitoring information about the RESTful Web service resource. Any object that is managed by a container (such as EJB) will have application scope. All other resources by default will have request scope. For more information, see "JaxRsResourceRuntimeBean" in the *WebLogic Server MBean Reference*.
`Servlet`	Displays monitoring information for the servlet that hosts the RESTful Web service application. For more information, see "ServletRuntimeMBean" in the *WebLogic Server MBean Reference*.

To monitor RESTful Web services using WLST, perform the steps provided in the following procedure.

In this procedure, the example steps provided demonstrate how to monitor the RESTful Web Services sample delivered with the WebLogic Server Samples Server, described at "Sample Application and Code Examples" in *Understanding Oracle WebLogic Server*

1. Invoke WLST, as described in "Invoking WLST" in *Understanding the WebLogic Scripting Tool*.

   For example:

   ```
   c:\> java weblogic.WLST
   ```

2. Connect to the Administration Server instance, as described in "connect" in *WLST Command Reference for WebLogic Server*.

   For example:

   ```
   wls:/offline> connect('weblogic','password','t3://localhost:8001')
   ```

3. Navigate to the server run-time MBean, as described in "serverRuntime" in *WLST Command Reference for WebLogic Server*.

   For example:

   ```
   wls:/wl_server/serverConfig> serverRuntime()
   Location changed to serverRuntime tree. This is a read-only tree
   with ServerRuntimeMBean as the root.
   For more help, use help('serverRuntime')
   wls:/wl_server/serverRuntime>
   ```

4. Navigate to the Web application component run-time MBean.

   For example, to navigate to run-time MBean for the application named `jaxrs`:

   ```
   wls:/wl_server/serverRuntime> cd('ApplicationRuntimes/jaxrs')
   wls:/wl_server/serverRuntime/ApplicationRuntimes/jaxrs> cd('ComponentRuntimes')
   wls:/wl_server/serverRuntime/ApplicationRuntimes/jaxrs/ComponentRuntimes>
   cd('examplesServer_/jaxrs')
   ```

5. Navigate to the application run-time MBean for the RESTful Web service servlet.

   For example:

   ```
   wls:/wl_
   server/serverRuntime/ApplicationRuntimes/jaxrs/ComponentRuntimes/example
   sServer_/jaxrs> cd('JaxRsApplications/RestServlet')
   ```

6. Review the monitoring information displayed for the RESTful Web service application. For more information, see "JaxRsApplicationRuntimeBean" in the *WebLogic Server MBean Reference*.

   For example:

   ```
   wls:/wl_
   server/serverRuntime/ApplicationRuntimes/jaxrs/ComponentRuntimes/example
   sServer_/jaxrs/JaxRsApplications/RestServlet> ls()
   dr--   ResourceConfig
   dr--   RootResources
   dr--   Servlet

   -r--   ErrorCount                                0
   -r--   ExecutionTimeAverage                      0
   ```

```
-r--   ExecutionTimeHigh                       0
-r--   ExecutionTimeLow                        0
-r--   ExecutionTimeTotal                      0
-r--   HttpMethodCounts                        {}
-r--   InvocationCount                         0
-r--   LastErrorDetails                        null
-r--   LastErrorMapper                         null
-r--   LastErrorTime                           0
-r--   LastHttpMethod                          null
-r--   LastInvocationTime                      0
-r--   LastResponseCode                        -1
-r--   Name                                    RestServlet
-r--   ResponseCodeCounts                      {}
-r--   StartTime                               1321907929703
-r--   Type                                    JaxRsApplicationRuntime

-r-x   preDeregister                           Void :

wls:/wl_
server/serverRuntime/ApplicationRuntimes/jaxrs/ComponentRuntimes/example
sServer_/jaxrs/JaxRsApplications/RestServlet>
```

7. Navigate to any of the following run-time MBeans to view additional monitoring information. For more information about the MBeans, see Table 7–2 or the *WebLogic Server MBean Reference*.
   - `ResourceConfig`
   - `RootResources`
   - `Servlet`
8. Exit WLST, as described in "Exiting WLST" in *Understanding the WebLogic Scripting Tool*.

   For example:

```
wls:/wl_
server/serverRuntime/ApplicationRuntimes/jaxrs/ComponentRuntimes/example
sServer_/jaxrs/JaxRsApplications/RestServlet> exit()
Exiting WebLogic Scripting Tool ...
c:\>
```

7.5 Enabling the Tracing Feature

The Jersey tracing feature provides useful information that describes how a request is processed and dispatched to Jersey JAX-RS RI components. Trace messages are output in the same order as they occur, so the numbering is useful to reconstruct the tracing order.

The following provides an example of a trace message. The message shows the request path and the initial set of regular expressions that will be matched, in order from left to right.

Example 7–1 Example of a Trace Message

```
Trace 001:
X-Jersey-Trace-001  match path "/items/3/tracks/2/" -> "/application\\.wadl(/.\*)?",
"/happy(/.\*)?", "(/.\*)?"
```

You can enable this feature server-wide or on a per-request basis, as described in the following sections:

- Section 7.5.1, "How to Enable Server-wide Tracing"
- Section 7.5.2, "How to Enable Per-request Tracing"

7.5.1 How to Enable Server-wide Tracing

You can enable server-wide tracing for RESTful Web service applications that are packaged with servlets, as described in Section 4.3, "Packaging With a Servlet," and the following elements are present in the `web.xml` file:

- Servlet class, defined as follows:

```
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
```

- One of the following `<init-param>`s, set to `True`:
  - `com.sun.jersey.config.feature.TracePerRequest`
  - `com.sun.jersey.config.feature.Trace`

To enable server-wide tracing:

- Update the `web.xml` deployment descriptor that is packaged with your application to add the following `<init-param>` element:

```
<init-param>
    <param-name>com.sun.jersey.config.feature.Trace</param-name>
    <param-value>true</param-value>
</init-param>
```

  For more information about the `<init-param>` element, see "servlet" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

- Enable the `com.sun.jersey.api.core.ResourceConfig.FEATURE_TRACE` property. For more information, see https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/core/ResourceConfig.html#FEATURE_TRACE.

7.5.2 How to Enable Per-request Tracing

You can enable per-request tracing for RESTful Web service applications that are packaged with servlets, as described in Section 4.3, "Packaging With a Servlet," and the following elements are present in the `web.xml` file:

- Servlet class, defined as follows:

```
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
```

- One of the following `<init-param>`s, set to `True`:
  - `com.sun.jersey.config.feature.TracePerRequest`
  - `com.sun.jersey.config.feature.Trace`

With per-request tracing, trace messages are output if a request header is present with the header name of `X-Jersey-Trace-Accept`.

Trace messages are output primarily as response headers with a header name of the form `X-Jersey-Trace-`*`XXX`*, where *`XXX`* is a decimal value corresponding to the trace

message number, and a header value that is the trace message. In certain cases, trace messages will be logged to the server if they cannot be included in the response headers, for example if the messages are too long.

To enable tracing on a per-request basis:

- Update the `web.xml` deployment descriptor that is packaged with your application to add the following `<init-param>` element:

  ```
  <init-param>
      <param-name>com.sun.jersey.config.feature.TracePerRequest</param-name>
      <param-value>true</param-value>
  </init-param>
  ```

  For more information about the `<init-param>` element, see "servlet" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

- Enable the `com.sun.jersey.api.core.ResourceConfig.FEATURE_TRACE_PER_REQUEST` property. For more information, see https://jersey.java.net/apidocs/1.13/jersey/index.html?com/sun/jersey/api/core/ResourceConfig.html#FEATURE_TRACE.

A

Updating the Version of Jersey JAX-RS RI

This appendix describes how to update the version of the Jersey JAX-RS Reference Implementation (RI) used by your RESTful Web service applications.

This appendix includes the following sections:

- Section A.1, "About Updating the Version of Jersey JAX-RS RI"
- Section A.2, "Updating the Version of Jersey JAX-RS RI at the Application Level"

A.1 About Updating the Version of Jersey JAX-RS RI

WebLogic Server supports Jersey 1.13 JAX-RS Reference Implementation (RI), which is a production quality implementation of the JSR-311 JAX-RS 1.1 specification, defined at: `https://jcp.org/en/jsr/summary?id=311`. As required, you can use a more recent version of the Jersey JAX-RS RI at the application level.

In WebLogic Server, any JAR file present in the system classpath is loaded by the WebLogic Server system classloader. All applications running within a server instance are loaded in application classloaders which are children of the system classloader. If a third-party JAR exists in the system classloader, applications must use a *filtering classloader* to load and use a different version.

The filtering classloader provides a mechanism for you to configure deployment descriptors to explicitly specify that certain packages should always be loaded from the application, rather than being loaded by the system classloader. For more information about using a filtering classloader, see "Using a Filtering Classloader" in *Developing Applications for Oracle WebLogic Server*.

A.2 Updating the Version of Jersey JAX-RS RI at the Application Level

To update the version of Jersey JAX-RS RI at the application level, use one of the following procedures based on whether you are updating an EAR or WAR file:

- Section A.2.1, "How to Update the Version of Jersey JAX-RS RI in an EAR File"
- Section A.2.2, "How to Update the Version of Jersey JAX-RS RI in a WAR File"

A.2.1 How to Update the Version of Jersey JAX-RS RI in an EAR File

To update the version of Jersey JAX-RS RI in an EAR file:

1. Include the preferred Jersey JAX-RS RI packages with the deployed Web application.
2. Edit the `weblogic-application.xml` deployment descriptor packaged with your application EAR file to include a `<prefer-application-packages>` element with

the preferred library packages listed. For more information about adding the `<prefer-application-packages>` element to `weblogic-application.xml`, see "weblogic-application.xml Deployment Descriptor Elements" in *Developing Applications for Oracle WebLogic Server*.

Example A–1 provides an example of how to update the `weblogic-application.xml` file.

Example A–1 Updating the Version of Jersey JAX-RS RI in a weblogic-application.xml File

```
<weblogic-web-app
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://www.oracle.com/technology/weblogic/weblogic-web-app/1.1/weblogic-web-app.xsd"
xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app">
    <prefer-application-packages>
        <!-- jersey-bundle-*.jar -->
        <package-name>com.sun.jersey.*</package-name>
        <package-name>com.sun.research.ws.wadl.*</package-name>
        <package-name>com.sun.ws.rs.ext.*</package-name>

        <!-- Jackson-*.jar -->
        <package-name>org.codehaus.jackson.*</package-name>

        <!-- jettison-*.jar -->
        <package-name>org.codehaus.jettison.*</package-name>

        <!-- jsr311*.jar -->
        <package-name>javax.ws.rs.*</package-name>

        <!-- asm.jar -->
        <package-name>org.objectweb.asm.*</package-name>
    </prefer-application-packages>
...
</weblogic-web-app>
```

A.2.2 How to Update the Version of Jersey JAX-RS RI in a WAR File

To update the version of Jersey JAX-RS RI in an WAR file (packaged standalone or within an EAR file):

1. Include the preferred Jersey JAX-RS RI packages with the deployed Web application.
2. Edit the `weblogic.xml` deployment descriptor packaged with your application WAR file to include a `<prefer-application-packages>` element with the preferred library packages listed. For more information about adding the `<prefer-application-packages>` element to `weblogic.xml`, see "prefer-application-packages" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

   Example A–2 provides an example of how to update the `weblogic.xml` file.J

Example A–2 Updating the Version of Jersey JAX-RS RI in a weblogic.xml File

```
<?xml version="1.0" encoding="UTF-8"?>

<wls:weblogic-web-app
xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-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/ejb-jar_3_0.xsd
http://xmlns.oracle.com/weblogic/weblogic-web-app
http://xmlns.oracle.com/weblogic/weblogic-web-app/1.5/weblogic-web-app.xsd">

     ...
    <wls:container-descriptor>
        <wls:prefer-application-packages>
            <!-- jersey-bundle-*.jar -->
            <wls:package-name>com.sun.jersey.*</wls:package-name>
            <wls:package-name>com.sun.research.ws.wadl.*</wls:package-name>
            <wls:package-name>com.sun.ws.rs.ext.*</wls:package-name>

            <!-- Jackson-*.jar -->
            <wls:package-name>org.codehaus.jackson.*</wls:package-name>

            <!-- jettison-*.jar -->
            <wls:package-name>org.codehaus.jettison.*</wls:package-name>

            <!-- jsr311*.jar -->
            <wls:package-name>javax.ws.rs.*</wls:package-name>

            <!-- asm.jar -->
            <wls:package-name>org.objectweb.asm.*</wls:package-name>
        </wls:prefer-application-packages>
    </wls:container-descriptor>
</wls:weblogic-web-app>
```