Oracle® Fusion Middleware

Developing and Securing RESTful Web Services for Oracle WebLogic Server

14c (14.1.1.0.0)
F18314-03
February 2023

Oracle Fusion Middleware Developing and Securing RESTful Web Services for Oracle WebLogic Server, 14c (14.1.1.0.0)

F18314-03

Contents

Preface

This documentation describes how to develop Java EE web services for Oracle WebLogic Server 14c.

Audience

This documentation is written for software developers who want develop Java EE web services for Oracle WebLogic Server 14c that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

Documentation Accessibility

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

Access to Oracle Support

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

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

New and Changed Features for This Release

For a comprehensive listing of the new and changed WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

Conventions

The following text conventions are used in this document:

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

1

Introduction to RESTful Web Services

RESTful web services are Java EE web services that you develop to conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

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.

Table 1-1 (Cont.) Constraints of the REST Architectural Style

Constraint	Description
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.

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 retrieve, create, update, and delete resources, respectively.

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 8 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 the following JAX-RS API and Reference Implementation (RI):

- JAX-RS 2.1
- Jersey 2.29

> **Note:**
>
> Jersey 2.x (JAX-RS 2.1 RI) support is provided by default in this release of WebLogic Server. Registration as a shared library is no longer required.
>
> The Jersey 1.x server-side APIs are no longer supported. You should use the corresponding standard JAX-RS 2.1 or Jersey 2.x APIs instead. The Jersey 1.x client API is deprecated. It is recommended that you update your RESTful client applications to use the JAX-RS 2.1 client APIs at your earliest convenience.

The Jersey 2.x (JAX-RS 2.1 RI) includes the following functionality:

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

Table 1-2 lists key features delivered with Jersey 2.x (JAX-RS 2.1 RI).

Table 1-2 Key Features in Jersey 2.x (JAX-RS 2.1 RI)

Key Feature	Description
Client API	Communicate with RESTful web services in a standard way. The Client API facilitates the consumption of a web service exposed via HTTP protocol and enables developers to concisely and efficiently implement portable client-side solutions that leverage existing and well established client-side HTTP connector implementations. For complete details, see: • Client API in *Jersey 2.29 User Guide* • Accessing REST Resources with the JAX-RS Client API in *The Java EE 8 Tutorial*
Asynchronous communication	Invoke and process requests asynchronously. For complete details, see: • Asynchronous Services and Clients in the *Jersey 2.29 User Guide* • Advanced Features of the Client API in *The Java EE 8 Tutorial*
Filters and interceptors	Using filters, modify inbound and outbound requests and responses., such as header information. Using interceptors, modify entity input and output streams. Filters and interceptors can be used on both the client and server side. For complete details, see Filters and Interceptors in the *Jersey 2.29 User Guide*.

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

Roadmap for Implementing RESTful Web Services

Review a roadmap of common tasks for developing, packaging and deploying, securing, and monitoring RESTful web services and clients. These tasks are listed in Table 1-3.

Table 1-3 Roadmap for Implementing RESTful Web Services and Clients

Task	More Information
Develop RESTful web services.	Developing RESTful Web Services
Develop clients to invoke the RESTful web services.	Summary of Tasks to Develop RESTful Web Service Clients
Package and deploy RESTful web services.	• Packaging With an Application Subclass • Packaging With a Servlet • Packaging as a Default Resource
Secure RESTful web services.	• Securing RESTful Web Services Using web.xml • Securing RESTful Web Services Using SecurityContext • Securing RESTful Web Services Using Java Security Annotations
Test RESTful web services.	Testing RESTful Web Services
Monitor RESTful web services.	Monitoring RESTful Web Services and Clients
(Optional) Migrate existing applications from Jersey 1.*x* to 2.*x*.	Migration Guide in *Jersey 2.29 User Guide*

Learn More About RESTful Web Services

Additional information about RESTful web services is available from resources such as the Community Wiki for Project Jersey, jcp.org, the JSR-370 JAX-RS 2.1 Specification, and more. These resources are listed in Table 1-4.

Table 1-4 Resources for More Information

Resource	Link
Jersey User Guide	*Jersey 2.29 User Guide*
Jersey API Javadoc	*Jersey 2.29 API Documentation*
Community Wiki for Project Jersey	`https://jersey.github.io/`
JSR-370 JAX-RS 2.1 Specification	`https://jcp.org/en/jsr/detail?id=370`
JAX-RS API Javadoc	`https://github.com/jax-rs`
JAX-RS Project	`https://github.com/jax-rs`
RESTful Web Services (JAX-RS) sample	Sample Application and Code Examples in *Understanding Oracle WebLogic Server*.
The Java EE 8 Tutorial—Building RESTful Web Services With JAX-RS	`https://javaee.github.io/tutorial/jaxrs.html#GIEPU`
"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

To develop Java EE web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS), you perform tasks such as defining the root resource class, mapping incoming HTTP requests to Java methods, customizing media types for requests and responses, and more.

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 Building, Packaging, and Deploying RESTful Web Service Applications , 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:

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

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 More Advanced RESTful Web Service Tasks.

> **Note:**
>
> In addition to the development tasks described in Table 2-1, you may wish to take advantage of features available with Jersey 2.x (JAX-RS 2.0 RI) when developing your RESTful web services. For a list of key features, see Table 1-2.

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

Task	More Information
Define the root resource class.	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.	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.	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.	Customizing Media Types for the Request and Response Messages
Extract information from the request.	Extracting Information From the Request Message
Build custom response messages to customize response codes or include additional metadata.	Building Custom Response Messages
Access information about the application deployment context or the context of individual requests.	Accessing the Application Context
Build new or extend existing resource URIs.	Building URIs
Evaluate one or more preconditions before processing a GET request, potentially reducing bandwidth and improving server performance.	Using Conditional GETs
Access the WADL.	Accessing the WADL
Optionally, 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.	Packaging With an Application Subclass
Secure your RESTful web services.	Securing RESTful Web Services and Clients

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`. See Defining the Relative URI of the Root Resource and Subresources.

- The `sayHello` method supports the HTTP GET method. See Mapping Incoming HTTP Requests to Java Methods.
- The `sayHello` method produces content of the MIME media type `text/plain`. See Customizing Media Types for the Request and Response Messages.

Additional examples are listed in 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!";
   }
}
```

Defining the Root Resource Class

A *root resource class* is a Plain Old Java Object (POJO) that meets specific annotation requirements. The root resource class must satisfy one or both of the following statements:

- Is annotated with `@Path`. See 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. See Mapping Incoming HTTP Requests to Java Methods.

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 as 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:

- How to Define the Relative URI of the Resource Class (@Path)
- How to Define the Relative URI of Subresources (@Path)
- What Happens at Runtime: How the Base URI is Constructed

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 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.

See the `@Path` annotation in the *Java EE 8 API Documentation*.

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 2-4, 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() {
      ...
   }
}
```

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. 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`. See Building, Packaging, and Deploying RESTful Web Service Applications .

- *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`.

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 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 How to Create or Update the Representation of the Resource (@PUT).	Yes
`@DELETE`	Deletes the representation of the resource identified by the URI. See 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 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. See the `@HEAD` annotation in the *Java EE 8 API Documentation*. 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. See the `@OPTIONS` annotation in the *Java EE 8 API Documentation*. 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. See the `@HttpMethod` annotation in the *Java EE 8 API Documentation*.	N/A

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

- About the Jersey Bookmark Sample
- How to Transmit a Representation of the Resource (@GET)
- How to Create or Update the Representation of the Resource (@PUT)
- How to Delete a Representation of the Resource (@DELETE)
- How to Create, Update, or Perform an Action on a Representation of the Resource (@POST)

About the Jersey Bookmark Sample

The examples referenced in the following sections are excerpted from the **bookmark sample** that is delivered with Jersey 2.x (JAX-RS 2.1 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:

- Accessing the bookmark sample at `https://repo1.maven.org/maven2/org/glassfish/jersey/examples/bookmark/`
- Browsing the bookmark sample source code on GitHub: `https://github.com/jersey/jersey/tree/master/examples/bookmark`
- Browsing the Maven repositories for all Jersey examples, including a WebLogic Server-specific example bundle for each version, at: `https://repo1.maven.org/maven2/org/glassfish/jersey/bundles/jersey-examples/`

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. See the `@GET` annotation in the *Java EE 8 Specification APIs*.

In Example 2-5, the annotated Java method, `getBookmarkAsJsonArray`, from the `BookmarksResource` class in the Jersey bookmark sample, will process HTTP GET requests. See 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(MediaType.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. See 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(MediaType.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;
```

```
        }
    }
}
```

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. See the `@PUT` annotation in the *Java EE 8 Specification APIs*.

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. See 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(MediaType.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);
        }});
    }

}
```

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. See the `@DELETE` annotation in the *Java EE 8 Specification APIs*.

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. See 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);
        }});
    }
}
```

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. See the `@POST` annotation in the *Java EE 8 Specification APIs*.

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. See 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(MediaType.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();
    }
}
```

Customizing Media Types for the Request and Response Messages

To customize the media types for request and response messages, add the `javax.ws.rs.Consumes` or `javax.ws.rs.Produces` annotation at the class level of the resource. This task is described in the following sections:

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

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.

See the `@Consumes` annotation in the *Java EE 8 Specification APIs*.

In Example 2-10, 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
   }
}
```

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.

See the `@Produces` annotation in the *Java EE 8 Specification APIs*.

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() { ... }
}
```

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.

Extracting Information From the Request Message

The `javax.ws.rs` package defines a set of annotations that enable you extract information from the request message to inject into parameters of your Java method. These annotations are listed and described in Table 2-4.

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

Annotation	Description
`@BeanParam`	Inject aggregated request parameters into a single bean. See the `@BeanParam` annotation in the *Java EE 8 API Documentation*. For additional usage information, see Parameter Annotations (@*Param) in the *Jersey 2.29 User Guide*.
`@CookieParam`	Extract information from the HTTP cookie-related headers to initialize the value of a method parameter. See the `@CookieParam` annotation in the *Java EE 8 API Documentation*.
`@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`. See How to Define the DefaultValue (@DefaultValue).
`@Encoded`	Enable encoding of a parameter value that is bound using one of the following annotations: `@FormParam`, `@MatrixParam`, `@PathParam`, or `@QueryParam`. See Enabling the Encoding Parameter Values (@Encoded).
`@FormParam`	Extract information from an HTML form of the type `application/x-www-form-urlencoded`. See the `@FormParam` annotation in the *Java EE 8 API Documentation*.
`@HeaderParam`	Extract information from the HTTP headers to initialize the value of a method parameter. See the `@HeaderParam` annotation in the *Java EE 8 API Documentation*.
`@MatrixParam`	Extract information from the URI path segments to initialize the value of a method parameter. See the `@MatrixParam` annotation in the *Java EE 8 API Documentation*.
`@PathParam`	Define the relative URI as a variable value (referred to as "URI path template"). See 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. See How to Extract Request Parameters (@QueryParam).

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 How to Define the DefaultValue (@DefaultValue).

In Example 2-12, 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) {
       ...
   }
}
```

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 How to Define the DefaultValue (@DefaultValue).

See the `@QueryParam` annotation in the *Java EE 8 Specification APIs*.

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)
    { ... }
}
```

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`. See the `@DefaultValue` annotation in the *Java EE 8 Specification APIs*.

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)
    { ... }
}
```

Enabling the Encoding Parameter Values (@Encoded)

Add the `javax.ws.rs.Encoded` 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. See the `@Encoded` annotation in the *Java EE 8 Specification APIs*.

In Example 2-15, the `@Encoded` 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.Encoded;


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

    @Encoded

    public String getUser(@PathParam("username") String userName) {
      ...
```

```
    }
}
```

Building Custom Response Messages

Instead of the default response codes, you can customize the response codes returned or include additional metadata information in the response.

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.

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 to create a javax.ws.rs.core.Response.ResponseBuilder instance and build the Response instance. The methods you can use are defined in Table 2-5. For more information, see the `Response` methods in the *Java EE 8 Specification APIs*.

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. See the `Response.ResponseBuilder` methods in the *Java EE 8 Specification APIs*.

Table 2-6 ResponseBuilder Methods for Building a Custom Response

Method	Description
`allow()`	Sets the list of allowed methods for the resource.
`build()`	Creates the Response instance from the current `ResponseBuilder` instance.
`cacheControl()`	Sets the cache control.

Table 2-6 (Cont.) ResponseBuilder Methods for Building a Custom Response

Method	Description
`clone()`	Create a copy of the `ResponseBuilder` to preserve its state.
`contentLocation()`	Sets the content location.
`cookie()`	Add cookies to the response.
`encoding()`	Sets the message entity content encoding.
`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.
`link()`	Adds a link header.
`links()`	Adds one or more link headers.
`location()`	Sets the location.
`newInstance()`	Creates a new `ResponseBuilder` instance.
`replaceAll()`	Replaces all existing headers with the newly supplied headers.
`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. See the `GenericEntity` methods in the *Java EE 8 Specification APIs*.

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();
   }
}
```

Mapping HTTP Request and Response Entity Bodies Using Entity Providers

HTTP request and response entity bodies automatically support a set of Java types that can be utilized by your RESTful web service. These Java types are listed in Table 2-7.

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`)

Table 2-7 (Cont.) Java Types Supported for HTTP Request and Response Entity Bodies

Java Type	Supported Media Types
`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.
	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.
	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 extension modules. See JSON in the *Jersey 2.29 User Guide*.

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() { }
}
```

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 `@Context` annotation in the *Java EE 8 Specification APIs*.

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. See Using Conditional GETs.
`SecurityContext`	Access the security context and secure the RESTful web service. See Securing RESTful Web Services Using SecurityContext.
`UriInfo`	Access application and request URI information. See Building URIs.

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 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;
    }
}
```

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;
    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();
        }
    }
}
```

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>
```

More Advanced RESTful Web Service Tasks

The *Jersey 2.29 User Guide* provides information about more advanced RESTful web service development tasks. See this guide, available at https://eclipse-ee4j.github.io/jersey.github.io/documentation/2.29/index.html, for the following topics:

- Context and Dependency Injection (CDI)
- Enterprise Java Beans (EJB)
- JSON
- XML

3

Developing RESTful Web Service Clients

You can develop Java EE web service clients that conform to the Representational State Transfer (REST) architectural style using the Jersey 2.x Java API for RESTful Web Services (JAX-RS) 2.1 reference implementation (RI).

> **Note:**
>
> Support for the Jersey 1.18 (JAX-RS 1.1RI) client APIs are deprecated in this release of WebLogic Server but are maintained for backward compatibility. See Develop RESTful Web Service Clients Using Jersey 1.18 (JAX-RS 1.1 RI)
>
> Oracle recommends that you update your RESTful client applications to use the Jersey 2.x (JAX-RS 2.1 RI) client APIs as described in this chapter at your earliest convenience.

This chapter includes the following sections:

Summary of Tasks to Develop RESTful Web Service Clients

Some of the tasks required to develop a RESTful web service client include creating the client class, targeting a web resource, identifying resources on the target, and more. The following table summarizes a subset of the tasks that are required to develop RESTful web service clients using Jersey 2.x (JAX-RS 2.0 RI).

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

Task	More Information
Create and configure an instance of the `javax.ws.rs.client.Client` class.	Creating and configuring a Client instance in *Jersey 2.29 User Guide*
Target the Web resource.	Targeting a web resource in *Jersey 2.29 User Guide*
Identify resources on WebTarget.	Identifying resource on WebTarget in *Jersey 2.29 User Guide*
Invoke an HTTP request.	Invoking a HTTP request in *Jersey 2.29 User Guide*

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

Example of a RESTful Web Service Client

You can learn more about how to create a RESTful web service client by viewing an example.The following is a simple example that shows how a client can be used to call the RESTful web service defined in Example 2-1. In this example:

- The `Client` instance is created and a `WebTarget` defined.
- The resource path is defined to access the Web resource.
- The `Invocation.Builder` is used to send a `get` request to the resource.
- The response is returned as a String value.

Example 3-1 Simple RESTful Web Service Client Using Jersey 2.x (JAX-RS 2.0 RI)

```
package samples.helloworld.client;
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class helloWorldClient{
    public static void main(String[] args) {
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:7101/restservice");
        WebTarget resourceWebTarget;
        resourceWebTarget = target.path("resources/helloworld");
        Invocation.Builder invocationBuilder;
        invocationBuilder = resourceWebTarget.request(
          MediaType.TEXT_PLAIN_TYPE);
        Response response = invocationBuilder.get();
        System.out.println(response.getStatus());
        System.out.println(response.readEntity(String.class));
...
    }
...
}
```

For complete details, see Client API in *Jersey 2.29 User Guide*.

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, and without 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 RESTful web service client JAR supports basic JAX-RS client-side functionality and OWSM security policies.

To use the standalone RESTful web service 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. See Developing and Securing Web Services in *Developing Applications with Oracle JDeveloper*.
2. Copy the file *ORACLE_HOME*/oracle_common/modules/clients/com.oracle.jersey.fmw.client.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.

Using the Reactive JAX-RS Client API

Reactive Client API is part of the JAX-RS 2.1 specification.

All invocations in the client API are set in synchronous mode by default. In synchronous processing, each request is processed in a single HTTP thread. After the processing is finished, the thread is returned back to the pool. This approach can result in taking more time to complete and unnecessary blocking of the resources.

Asynchronous programming in JAX-RS enables client to unblock certain threads by pushing the work to background threads which can be monitored and joined at a later time. The resources are used optimally to achieve quick response time.

In JAX-RS 2.1, you can achieve asynchronous programming by providing an instance of `InvocationCallback`, which also enables a more reactive programming style in which the user-provided code reacts only when a certain event has occurred. Callback works well for simple cases but the coding becomes complex when multiple events come into play. To make the asynchronous programming more readable, a new interface `CompletionStage` is introduced for managing large number of methods dedicated for asynchronous computations.

See Usage and Extension Modules in *Jersey 2.29 User Guide* for more information about the different types of invokers based on CompletionStage.

See Reactive JAX-RS Client API in *Jersey 2.29 User Guide* for more detailed information.

> **Note:**
>
> In WebLogic Server, the following reactive libraries are not supported:
>
> - `RxJava` (Observable)
> - `RxJava` (Flowable)
> - `Guava` (ListenableFuture)

4

Building, Packaging, and Deploying RESTful Web Service Applications

Oracle WebLogic Server provides the components and utilities you need to package and deploy Java EE web services that conform to the Representational State Transfer (REST) architectural style using the Jersey 2.x Java API for RESTful Web Services (JAX-RS) 2.1 reference implementation (RI).

Building RESTful Web Service Applications

You can build your RESTful web service and client applications using the compilation tools, such as Apache Ant, Maven, or your favorite IDE, such as Oracle JDeveloper.See Overview of WebLogic Server Application Development in *Developing Applications for Oracle WebLogic Server*. For more information about JDeveloper, see Building Java Projects in *Developing Applications with Oracle JDeveloper*.

Packaging RESTful Web Service Applications

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.

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. See 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. See 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`. See Packaging as a Default Resource.

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. See `javax.ws.rs.core.Application` in the *Java EE 8 Specification APIs*.

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 define 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 What Happens at Runtime: How the Base URI is Constructed. See the `@ApplicationPath` annotation in the *Java EE 8 Specification APIs*.

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 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;
    }
}
```

Alternatively, use the following API to scan for root resource and provider classes for a specified classpath or a set of package names:

- `org.glassfish.jersey.server.ResourceConfig`, as described in JAX-RS Application Model in *Jersey 2.29 User Guide*.

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.

- How to Package the RESTful Web Service Application with Servlet 3.0
- 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*.

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`.

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

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>`	Not required.

`<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`
	`...`
	See Packaging With an Application Subclass.
	• The value `resources`. This is the default base URI pattern for RESTful web service applications. See 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 What Happens at Runtime: How the Base URI is Constructed.

The following example demonstrates 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>
        <servlet-name>org.foo.rest.MyApplication</servlet-name>
    </servlet>
    ...
    <servlet-mapping>
        <servlet-name>org.foo.rest.MyApplication</servlet-name>
        <url-pattern>/resources</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

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.
`<servlet-class>`	Set this element to `org.glassfish.jersey.servlet.ServletContainer` to delegate all Web requests to the Jersey servlet.
`<init-param>`	Not required.
`<servlet-mapping>`	Set as the base URI pattern that gets mapped to the servlet. If not specified, this value defaults to `resources`. See Packaging as a Default Resource. For more information about how this information is used in the base URI of the resource, see What Happens at Runtime: How the Base URI is Constructed.

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

> **Note:**
>
> The JAX-RS Specification requires the RESTful Web Service application using the `web.xml` without the Application subclass for Servlet 3.0 to set the `servlet-name` to `javax.ws.rs.Application` as described in the *Jersey 2.29 User Guide*. The packaging method defined in this section is not supported by the JAX-RS Specification.

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

```
<web-app>
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey Web Application</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

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 `org.glassfish.jersey.servlet.ServletContainer` to delegate all Web requests to the Jersey servlet.

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 specify the packages to be scanned for resources and providers, as follows:
	
	`<init-param>`
	`  <param-name>`
	`  jersey.config.server.provider.packages`
	`  </param-name>`
	`  <param-value>`
	`  *project1*`
	`  </param-value>`
	`</init-param>`
	`<init-param>`
	`  <param-name>`
	`  jersey.config.server.provider.scanning.recursive`
	`  </param-name>`
	`  <param-value>`
	`  false`
	`  </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`
	`...`
	
	See Packaging With an Application Subclass.
	• The value `resources`. This is the default base URI pattern for RESTful web service applications. See 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 What Happens at Runtime: How the Base URI is Constructed.

The following example demonstrates how to update the `web.xml` file if a 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>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
           <param-name>jersey.config.server.provider.packages</param-name>
           <param-value>org.foo.myresources,org.bar.otherresources</param-value>
        </init-param>
        <init-param>
           <param-name>jersey.config.server.provider.scanning.recursive</param-name>
           <param-value>false</param-value>
        </init-param>
        ...
    </servlet>
    ...
</web-app>
```

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 Packaging With a Servlet.
- The `@ApplicationPath` annotation is not defined in the `javax.ws.rs.core.Application` subclass, as described in 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`

Deploying RESTful Web Service Applications

Application deployment refers to the process of making an application or module available for processing client requests in a WebLogic domain. For information about deploying a web application, see Understanding WebLogic Server Deployment in *Deploying Applications to Oracle WebLogic Server*.

5

Securing RESTful Web Services and Clients

Oracle WebLogic Server fully supports the means to secure Java EE web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS) reference implementation (RI).

About RESTful Web Service Security

You can secure your RESTful web services so that they can support authentication, authorization, or encryption. You can use one of the following methods:

- Updating the `web.xml` deployment descriptor to access information about the authenticated users. See Securing RESTful Web Services Using web.xml.
- Using the `javax.ws.rs.core.SecurityContext` interface to access security-related information for a request. See Securing RESTful Web Services Using SecurityContext.
- Applying annotations to your JAX-RS classes. See Securing RESTful Web Services Using Java Security Annotations.

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*.

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 8 Tutorial*.

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. 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`. See user-data-constraint in *Developing Applications with the WebLogic Security Service*.

Example 5-1 Securing RESTful Web Services Using Basic Authentication

The following example demonstrates how to secure a Jersey 2.x (JAX-RS 2.0) RESTful web service using basic authentication.

```
<web-app>
    <servlet>
        <servlet-name>RestServlet</servlet-name>

        <servlet-class>org.glassfish.jersey.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>
```

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 following topics in the *Java EE 8 Specification APIs*:

- `SecurityContext` interface
- `@Context` annotation

Example 5-2 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 5-2 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.");
        }
```

Securing RESTful Web Services Using Java Security Annotations

The `javax.annotation.security` package provides annotations that you can use to secure your RESTful web services.These annotations are defined in Table 5-1.

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.

Before you can use the annotations defined in Table 5-1, you must register the roles-allowed feature, as described in Securing JAX-RS resources with standard javax.annotation.security annotations in the *Jersey 2.29 User Guide*.

Example 5-3 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 5-3 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!";
   }
}
```

See also:

- Specifying Authorized Users by Declaring Security Roles in *The Java EE 8 Tutorial*
- `javax.annotation.security` Javadoc

6
Testing RESTful Web Services

After you have deployed a Web application that contains a RESTful web service to Oracle WebLogic Server, you can test your application. This chapter describes how to test Java EE web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).
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
WebLogic Server Administration Console	Navigate to the Testing tab for your application deployment in the WebLogic Server Administration Console to validate the application deployment and view the WADL file. See Test RESTful Web Services in *Oracle WebLogic Server Administration Console Online Help*.

7

Monitoring RESTful Web Services and Clients

Oracle WebLogic Server supports a number of ways to monitor Java EE web services that conform to the Representational State Transfer (REST) architectural style using Java API for RESTful Web Services (JAX-RS).

About Monitoring RESTful Web Services

WebLogic Server provides several runtime MBeans that capture runtime information and let you monitor runtime statistics for your RESTful web service applications.Application monitoring is useful when you need to identify the performance hotspots in your JAX-RS application, observe execution statistics of particular resources, or listen to application or request lifecycle events.
You can use the methods defined in Table 7-1 to monitor your RESTful web service applications.

Table 7-1 Methods for Monitoring RESTful Web Services

Method	Description
WebLogic Server Administration Console	Access runtime information and monitor runtime statistics, as described in Monitoring RESTful Web Services Using the Administration Console.
WebLogic Scripting Tool (WLST)	Access runtime information and monitor runtime statistics, as described in 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 Enabling the Tracing Feature.

In addition to the monitoring methods described in Table 7-1, Jersey 2.x (JAX-RS 2.1 RI) provides additional monitoring features, including support for event listeners and statistics monitoring. See Monitoring Jersey Applications in the *Jersey 2.29 User Guide*.

> **Note:**
>
> RESTful web service monitoring is enabled by default. In some cases, this may result in increased memory consumption. You can disable the monitoring feature at the domain level, and at the application level. See Disabling RESTful Web Service Application Monitoring.

Monitoring RESTful Web Services Using the Administration Console

Using the WebLogic Server Administration Console, you can monitor enhanced runtime statistics for your RESTful applications and resources, including detailed deployment and configuration data, global execution statistics, and resource and resource method execution statistics.

To monitor your deployed RESTful web services using the WebLogic Server Administration Console, follow these steps:

1. Invoke the WebLogic Server 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*.

Monitoring RESTful Web Services Using WLST

You can use WLST to monitor the runtime MBeans that capture runtime information and runtime statistics for your RESTful web service applications.These MBeans are listed and described in Table 7-2.

Table 7-2 Runtime MBeans for Monitoring RESTful Web Services

Runtime MBean	Description
`ExceptionMapperStatistics`	Displays monitoring information about the RESTful web service application exception mapper executions. See JaxRsExceptionMapperStatisticsRuntimeMBean in *MBean Reference for Oracle WebLogic Server*.
`JaxRsApplication`	Displays monitoring information for the RESTful web service application. See JaxRsApplicationRuntimeBean in *MBean Reference for Oracle WebLogic Server*.

Table 7-2 (Cont.) Runtime MBeans for Monitoring RESTful Web Services

Runtime MBean	Description
`RequestStatistics`	Displays monitoring information about requests executed by the RESTful web service application. The statistics apply to all requests handled by the application and are not bound to any specific resource or resource method. See JaxRsExecutionStatisticsRuntimeMBean in *MBean Reference for Oracle WebLogic Server*.
`ResourceConfig`	Displays monitoring information about the RESTful web service application resource configuration. See JaxRsResourceConfigTypeRuntimeBean in*MBean Reference for Oracle WebLogic Server*. **Note:** The `JaxRsResourceConfigTypeRuntimeBean` is deprecated in this release of WebLogic Server. You should use the `Properties` and `ApplicationClass` attributes of the `JaxRsApplicationRuntimeMBean` instead. See JaxRsApplicationRuntimeBean in*MBean Reference for Oracle WebLogic Server*.
`ResponseStatistics`	Displays monitoring information about responses created by the RESTful web service application. The statistics apply to all responses created by the application and are not bound to any specific resource or resource method. See JaxRsResponseStatisticsRuntimeMBean in *MBean Reference for Oracle WebLogic Server*.
`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. See JaxRsResourceRuntimeMBeanin *MBean Reference for Oracle WebLogic Server*. **Note:** This MBean is deprecated in this release of WebLogic Server. You should use RootResourcesByClass instead.
`RootResourcesByClass`	Displays monitoring information for each resource class that is deployed in the RESTful web service application. One resource class can serve requests matched to different URIs. The array contains resource classes that are registered in the resource model plus resource classes of sub resources returned from sub resource locators. See JaxRsResourceRuntimeMBean in*MBean Reference for Oracle WebLogic Server*.
`RootResourcesbyURI`	Displays monitoring information for each URI that is exposed in the RESTful web service application. See JaxRsUriRuntimeMBean in *MBean Reference for Oracle WebLogic Server*.
`Servlet`	Displays monitoring information for the servlet that hosts the RESTful web service application. See ServletRuntimeMBean in *MBean Reference for Oracle WebLogic Server*.

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 JAX-RS 2.0 Asynchronous Processing sample delivered with the WebLogic Server Samples Server, described at Sample Applications 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:\Oracle\oracle_common\common\bin> 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 runtime MBean, as described in serverRuntime in *WLST Command Reference for WebLogic Server*.

   For example:

   ```
   wls:/samples/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:/samples/serverRuntime>
   ```

4. Navigate to the Web application component runtime MBean.

   For example, to navigate to runtime MBean for the application named `jaxrs-async`:

   ```
   wls:/samples/serverRuntime> cd('ApplicationRuntimes/jaxrs-async')
   wls:/samples/serverRuntime/ApplicationRuntimes/jaxrs-async>
   cd('ComponentRuntimes')
   wls:/samples/serverRuntime/ApplicationRuntimes/jaxrs/ComponentRuntimes> cd
   ('AdminServer_/jaxrs-async')
   ```

5. Navigate to the application runtime MBean for the RESTful web service request statistics.

   For example:

   ```
   wls:/samples/serverRuntime/ApplicationRuntimes/jaxrs-async/
   ComponentRuntimes> cd ('AdminServer_/jaxrs-async'
   wls:/samples_domain/serverRuntime/ApplicationRuntimes/jaxrs-async/
   ComponentRuntimes/AdminServer_/jaxrs-async>
   cd ('JaxRsApplications/
   examples.javaee7.jaxrs.async.MessageApplication/RequestStatistics/
   examples.javaee7.jaxrs.async.MessageApplication_RequestStatistics')
   ```

6. Review the monitoring information displayed for the RESTful web service application. See JaxRsApplicationRuntimeBean in *MBean Reference for Oracle WebLogic Server*.

   For example:

   ```
   wls:/samples/serverRuntime/ApplicationRuntimes/jaxrs-async/ComponentRuntimes/
   AdminServer_/jaxrs-async
   /JaxRsApplications/examples.javaee7.jaxrs.async.MessageApplication/
   RequestStatistics
   /examples.javaee7.jaxrs.async.MessageApplication_RequestStatistics>
   ls()
   -r--   AvgTimeLast15m                              -1
   -r--   AvgTimeLast15s                              -1
   ```

```
-r--   AvgTimeLast1h                              -1
-r--   AvgTimeLast1m                              -1
-r--   AvgTimeLast1s                              -1
-r--   AvgTimeTotal                               0
-r--   MaxTimeLast15m                             -1
-r--   MaxTimeLast15s                             -1
-r--   MaxTimeLast1h                              -1
-r--   MaxTimeLast1m                              -1
-r--   MaxTimeLast1s                              -1
-r--   MaxTimeTotal                               0
-r--   MinTimeLast15m                             -1
-r--   MinTimeLast15s                             -1
-r--   MinTimeLast1h                              -1
-r--   MinTimeLast1m                              -1
-r--   MinTimeLast1s                              -1
-r--   MinTimeTotal                               0
-r--   Name
examples.javaee7.jaxrs.async.MessageApplication_RequestStatisti
cs
-r--   RequestCountLast15m                        0
-r--   RequestCountLast15s                        0
-r--   RequestCountLast1h                         0
-r--   RequestCountLast1m                         0
-r--   RequestCountLast1s                         0
-r--   RequestCountTotal                          0
-r--   RequestRateLast15m                         0.0
-r--   RequestRateLast15s                         0.0
-r--   RequestRateLast1h                          0.0
-r--   RequestRateLast1m                          0.0
-r--   RequestRateLast1s                          0.0
-r--   RequestRateTotal                           0.0
-r--   Type                                       JaxRsExecutionStatisticsRuntime

wls:/samples/serverRuntime/ApplicationRuntimes/jaxrs-async/ComponentRuntimes/
AdminServer_/jaxrs-async
/JaxRsApplications/examples.javaee7.jaxrs.async.MessageApplication/
RequestStatistics
/examples.javaee7.jaxrs.async.MessageApplication_RequestStatistics>
```

7. Navigate to any of the other runtime MBeans described in Table 7-2 to view additional monitoring information.
8. Exit WLST, as described in Exiting WLST in *Understanding the WebLogic Scripting Tool*.

   For example:

```
wls:/samples/serverRuntime/ApplicationRuntimes/jaxrs-async/ComponentRuntimes/
AdminServer_/jaxrs-async
/JaxRsApplications/examples.javaee7.jaxrs.async.MessageApplication/
RequestStatistics
/examples.javaee7.jaxrs.async.MessageApplication_RequestStatistics>exit()
Exiting WebLogic Scripting Tool.
c:\>
```

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.

When enabled, the Jersey 2.x tracing facility collects useful information for individual requests from all components of the JAX-RS server-side request processing pipeline. The information collected may provide vital details for troubleshooting your Jersey or JAX-RS application.

The tracing information for a single request is returned to the requesting client in the HTTP headers of the response. In addition, the information is logged on the server-side using a dedicated Java Logger instance.

For more information about enabling the Jersey 2.x tracing facility, see Tracing Support in *Jersey 2.29 User Guide*.

Disabling RESTful Web Service Application Monitoring

You can disable monitoring for an individual Jersey 2.x Java API for RESTful Web Services (JAX-RS) application, or globally for an entire WebLogic domain.

For example, you can disable monitoring in the following ways:

- At the application level, you can set a WebLogic Server-specific Jersey 2.x application property, `jersey.config.wls.server.monitoring.enabled`. See Disabling Monitoring for a RESTful Web Service Application Using Jersey Property.
- At both the application level and at the domain level, you can disable monitoring using a WebLogic Configuration MBean, `WebAppComponentMBean.JaxRsMonitoringDefaultBehavior`. See Disabling Monitoring for a RESTful Web Service Application Using WebLogic Configuration MBean and Disabling RESTful Web Service Application Monitoring for a WebLogic Domain.

WebLogic Server uses the following algorithm to determine whether monitoring should be enabled or disabled for each application.

1. WebLogic Server checks the JAX-RS application property `jersey.config.wls.server.monitoring.enabled`.

   If it is set for the application, then WebLogic Server uses this value to determine if monitoring should be enabled or disabled for the application. If this value is not set, it proceeds to the next step.

2. WebLogic Server checks the configuration MBean `WebAppComponentMBean.JaxRsMonitoringDefaultBehavior` property for the individual application.

   If it is set for the application, then WebLogic Server uses this value to determine if monitoring should be enabled or disabled for the application. If this value is not set, it proceeds to the next step.

3. WebLogic Server checks the configuration MBean `WebAppContainerMBean.JaxRsMonitoringDefaultBehavior` property setting for the domain.

   If it is set for the domain, then WebLogic Server uses this value to determine if monitoring should be enabled or disabled for the application. If this value is not set, it proceeds to the next step.

4. WebLogic Server uses the default setting, which is to enable JAX-RS monitoring for the application if none of the configuration properties in the previous steps have been set.

Disabling Monitoring for a RESTful Web Service Application Using Jersey Property

Jersey 2.x supports the following WebLogic Server-specific property that you can use to disable application monitoring for an individual RESTful web service application:

`jersey.config.wls.server.monitoring.enabled`

Setting this property to `false` disables monitoring in the application. You can set this property programmatically in the JAX-RS application subclass code, or declaratively via Servlet init parameters specified in the `web.xml` as shown in the following examples.

For convenience, the property name is stored in the `weblogic.jaxrs.server.WeblogicServerProperties.MONITORING_ENABLED` constant field.

Example 7-1 provides an example of how you can disable monitoring programmatically in a RESTful web service application by extending the JAX-RS Application class.

Example 7-1 Disable Application Monitoring Programmatically by Extending the JAX-RS Application Class

```
ApplicationPath("/")
public class MyApplication extends Application {

  public Map<String, Object> getProperties() {
    final Map<String, Object> properties = new HashMap<>();
    // Disable JAX-RS Application monitoring (and WLS console monitoring) for this
internal application.
    properties.put(weblogic.jaxrs.server.WeblogicServerProperties.MONITORING_ENABLED,
false);

    return properties;
  }
}
```

Example 7-2 provides an example of how you can disable monitoring programmatically in a RESTful web service application by extending the JAX-RS Jersey `ResourceConfig` class.

Example 7-2 Disable Application Monitoring Programmatically by Extending the Jersey ResourceConfig Class

```
@ApplicationPath("/")
public class MyApplication extends ResourceConfig {

  public MyApplication() {
    // ...

    // Disable JAX-RS Application monitoring (and WLS console monitoring) for this
internal application.
    property(weblogic.jaxrs.server.WeblogicServerProperties.MONITORING_ENABLED, false);
  }

  // ...
}
```

Example 7-3 provides an example of how you can disable monitoring declaratively using Servlet init parameters specified in the `web.xml`.

Example 7-3 Disable Application Monitoring Declaratively Using Servlet Init Parameters in web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.5"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

  <servlet>
    <servlet-name>com.examples.MyApplication</servlet-name>
    ...
    <init-param>
      <param-name>jersey.config.wls.server.monitoring.enabled</param-name>
      <param-value>false</param-value>
    </init-param>
    ...
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>com.examples.MyApplication</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Disabling Monitoring for a RESTful Web Service Application Using WebLogic Configuration MBean

After you have deployed a RESTful web service application on WebLogic Server, you can disable monitoring of the application by using WLST, for example, to set the `JaxRsMonitoringDefaultBehavior` property to `false` on its `WebAppComponentMBean`:

```
webAppComponentMBean.setJaxRsMonitoringDefaultBehavior("false")
```

This is a per-application property that is internally used by Jersey/WebLogic integration code to determine the state of the default monitoring behavior in the JAX-RS application:

- If set to `true`, monitoring for the JAX-RS application is enabled.
- If set to `false`, monitoring for the JAX-RS application is disabled.
- If the property is not set, then the domain-level Web Application Container property `WebAppContainerMBean.isJaxRsMonitoringDefaultBehavior())` is used as a fall-back.

> **Note:**
>
> The value of this application-specific property (if set) overrides the value of domain-level configuration property.
>
> By default the value is not explicitly set.

Disabling RESTful Web Service Application Monitoring for a WebLogic Domain

Application monitoring is enabled by default for all RESTful web service applications deployed to a WebLogic domain. It is possible to reverse this default behavior in a WebLogic domain and disable JAX-RS monitoring for all RESTful web service applications deployed in the domain (unless overridden by an application-specific configuration) by setting the `JaxRsMonitoringDefaultBehavior` property on `WebAppContainerMBean` to `false`:

```
WebAppContainerMBean.setJaxRsMonitoringDefaultBehavior("false")
```

This Web Application Container property is a domain-level property used by Jersey/WebLogic integration code to determine the behavior of monitoring in JAX-RS applications at the domain level:

- If set to `true` (or not set), then JAX-RS monitoring is enabled (if not overridden by properties set directly in an application). By default this property is not set explicitly and monitoring is enabled.
- If set to `false`, then monitoring for all JAX-RS applications is disabled by default for the given domain.

> **Note:**
>
> You can override this domain-level setting in each JAX-RS application by setting similar properties, `WebAppComponentMBean#isJaxRsMonitoringDefaultBehavior())`, at the application level. See Disabling Monitoring for a RESTful Web Service Application Using WebLogic Configuration MBean.

You can update the `WebAppContainerMBean.JaxRsMonitoringDefaultBehavior` property for the domain using WLST commands before starting the domain, or before deploying any applications, as shown in Example 7-4.

Example 7-4 provides a sample WLST script that disables JAX-RS monitoring for the entire domain by default.

Example 7-4 Sample WLST Script for Disabling JAX-RS Monitoring at Domain Level

```
connect(<user>, <password>)
edit()
startEdit()
cd("WebAppContainer/<domain_name>/")
cmo.setJaxRsMonitoringDefaultBehavior(false)
activate()
```

> **Note:**
>
> You must restart the domain after you disable monitoring to ensure that all previously deployed applications are redeployed with the new setting.

[Example 7-5 shows a section of the resulting domain configuration document at *DOMAIN_NAME*/config/config.xml after you have changed the jax-rs-monitoring-default-behavior setting to false.

Example 7-5 config.xml file with JAX-RS Monitoring Disabled at the Domain Level

```
<?xml version='1.0' encoding='UTF-8'?>
<domain ...>
  <name>mydomain</name>
  ...
  <web-app-container>
    <jax-rs-monitoring-default-behavior>false
</jax-rs-monitoring-default-behavior>
  </web-app-container>
  ...
</domain>
```

> **Note:**
>
> Although it is possible to do so, Oracle does not recommend editing the config.xml file directly. See Domain Configuration Files in *Understanding Domain Configuration for Oracle WebLogic Server*.

Enable Monitoring of Synthetic Jersey Resources in a RESTful Web Service Application

When a RESTful web service application is deployed on WebLogic Server, the Jersey runtime (to satisfy JAX-RS specification requirements) introspects all the application resources and eventually extends the resource model of the application with additional synthetic resources and/or resource methods. For example, synthetic resources and resource methods are added to support:

- Resources exposing the WADL for the entire JAX-RS application, as well as a partial WADL for any deployed resource.
- OPTIONS method handlers for each resource or resource method of the JAX-RS application.
- HEAD method handlers for each resource or resource method of the JAX-RS application.

Depending on the application, it is possible that quite a lot of additional synthetic resources may get added to a deployed application. For performance reasons, WebLogic Server, by default, does not expose runtime MBeans for these extended synthetic resources and resource methods.

You can, however, display information about these additional synthetic resources in the WebLogic Server Administration Console by setting the following Jersey 2.x/JAX-RS application property to true:

```
jersey.config.wls.server.monitoring.extended.enabled
```

You can set this property programmatically in the JAX-RS application subclass code, or declaratively via Servlet init parameters specified in the `web.xml` as shown in the following examples.

For convenience, the property name is stored in the `weblogic.jaxrs.server.WeblogicServerProperties.MONITORING_EXTENDED_ENABLED` constant field.

Example 7-6 provides an example of how you can enable monitoring for synthetic resources programmatically in a JAX-RS application by extending the JAX-RS Application class.

Example 7-6 Enable Synthetic Monitoring Programmatically by Extending the JAX-RS Application Class

```
@ApplicationPath("/")
public class MyApplication extends Application {

  public Map<String, Object> getProperties() {
    final Map<String, Object> properties = new HashMap<>();
    // Expose MBeans for extended JAX-RS resources and resource methods

properties.put(weblogic.jaxrs.server.WeblogicServerProperties.MONITORING_EXTENDED_ENABL
ED, true);

    return properties;
  }
}
```

Example 7-7 provides an example of how you can enable monitoring of synthetic resources programmatically in a JAX-RS/Jersey application by extending the JAX-RS Jersey `ResourceConfig` class.

Example 7-7 Enable Synthetic Monitoring Programmatically by Extending the Jersey ResourceConfig Class

```
@ApplicationPath("/")
public class MyApplication extends ResourceConfig {

  public MyApplication() {
    // ...

    // Expose MBeans for extended JAX-RS resources and resource methods

property(weblogic.jaxrs.server.WeblogicServerProperties.MONITORING_EXTENDED_ENABLED,
true);
  }

  // ...
}
```

Example 7-8 provides an example of how you can enable monitoring of synthetic resources declaratively using Servlet init parameters specified in the `web.xml`.

Example 7-8 Enable Synthetic Monitoring Declaratively Using Servlet Init Parameters in web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app version="2.5"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

   <servlet>
    <servlet-name>com.examples.MyApplication</servlet-name>
    ...
    <init-param>
      <param-name>jersey.config.wls.server.monitoring.extended.enabled</param-
name>
      <param-value>true</param-value>
    </init-param>
    ...
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>com.examples.MyApplication</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

8

Using Server-Sent Events in WebLogic Server

Oracle WebLogic Server supports server-sent events through the integration of the Jersey 2.x library.The Jersey 2.x library provides the Reference Implementation (RI) of JSR 370 (JAX-RS 2.1: Java API for RESTful Web Services).

Overview of Server-Sent Events (SSE)

Server-sent events enable servers to push data to web pages over standard HTTP or HTTPS through a unidirectional client-server connection. In the server-sent events communication model, the browser client establishes the initial connection, and the server provides the data and sends it to the client. For general information about server-sent events, see the Server-Sent Events W3C Candidate Recommendation.
Server-sent events are part of the HTML 5 specification, which also includes WebSocket technology. Both communication models enable servers to send data to clients unsolicited. However, server-sent events establish one-way communication from server to clients, while a WebSocket connection provides a bidirectional, full-duplex communication channel between servers and clients, promoting user interaction through two-way communication. The following key differences exist between WebSocket and server-sent events technologies:

- Server-sent events can only push data to the client, while WebSocket technology can both send and receive data from a client.
- The simpler server-sent events communication model is better suited for server-only updates, while WebSocket technology requires additional programming for server-only updates.
- Server-sent events are sent over standard HTTP and therefore do not require any special protocol or server implementation to work. WebSocket technology requires the server to understand the WebSocket protocol to successfully upgrade an HTTP connection to a WebSocket connection.

For more information about WebSocket technology, see Using the WebSocket Protocol in WebLogic Server in *Developing Applications for Oracle WebLogic Server*.

Using Server-Sent Events

Server-sent events are used to push notifications asynchronously to the client over standard HTTP protocol.

From JAX-RS 2.1, the server-sent event APIs are defined in the `javax.ws.rs.sse` package. This package includes the interfaces `Sse`, `SseEventSink`, `SseEvent`, `SseBroadcaster`, and `SseEventSource` for server-sent events.

The server-sent events server API is used to accept connections and send events to one or more clients. From the server side, an instance that implements the interface `SseEventSink` corresponds to a single client HTTP connection.

You can also configure the applications to send events to multiple clients simultaneously using the `SseBroadcaster` interface. The interface enables to send events to all registered event outputs.

From the client side, the interface `SseEventSource` is used to open a connection to the `Web Target` that is configured with a resource location. The clients must request the opening of a server-sent event connection using the media type `text/event-stream` in the Accept header. The established connection is persistent and can be re-used to send multiple events from the server.

Understanding the WebLogic Server-Sent Events API

WebLogic Server supports server-sent events through the integration of the Jersey 2.x. The use of server-sent events through Jersey 2.x is supported only in JAX-RS resources.
From JAX-RS 2.1, the server-sent event APIs are defined in the `javax.ws.rs.sse` package. This package includes the interfaces `Sse`, `SseEventSink`, `SseEvent`, `SseBroadcaster`, and `SseEventSource` for server-sent events.

For more information about server-sent events in Jersey 2.x, see Server-Sent Events (SSE) Support in the *Jersey 2.29 User Guide*.

The WebLogic Server Server-Sent Events API is in the package `org.glassfish.jersey.media.sse`. For information about the interfaces and classes included in this package, see the API documentation for `javax.ws.rs.sse` in the *Java EE 8 API Documentation*.

Sample Applications for Server-Sent Events

Sample applications for server-sent events are available through the Jersey project. Refer to the following locations:

- https://github.com/eclipse-ee4j/jersey/tree/master/examples/server-sent-events-jaxrs
- https://github.com/eclipse-ee4j/jersey/tree/master/examples/sse-item-store-jaxrs-webapp
- https://github.com/eclipse-ee4j/jersey/tree/master/examples/sse-twitter-aggregator

A

Compatibility with Earlier Jersey/JAX-RS Releases

Some Jersey 1.x (JAX-RS 1.1 RI) features have been deprecated or are no longer supported in Oracle WebLogic Server, but have been maintained for backward compatibility.

- Develop RESTful Web Service Clients Using Jersey 1.18 (JAX-RS 1.1 RI)
- Support for Jersey 1.18 (JAX-RS 1.1 RI) Deployments Packaged with Pre-3.0 Servlets

Develop RESTful Web Service Clients Using Jersey 1.18 (JAX-RS 1.1 RI)

Support for several client packages, including the `com.sun.jersey` package, its nested packages, and the `weblogic.jaxrs.api.client` package, is deprecated in this release of Oracle WebLogic Server.

> **Note:**
>
> Oracle recommends that you update your RESTful client applications to use the JAX-RS 2.0 client APIs at your earliest convenience. See Summary of Tasks to Develop RESTful Web Service Clients.
>
> The Jersey 1.x server-side APIs are no longer supported. You should use the corresponding standard JAX-RS 2.0 or Jersey 2.x server APIs instead.

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 More Advanced RESTful Web Service Client Tasks

Table A-1 Summary of Tasks to Develop RESTful Web Service Clients

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

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

Example of a RESTful Web Service Client

The following simple example demonstrates how a RESTful web service client can be used to call the RESTful web service defined in Example 2-1. In this example:

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

Additional examples are listed in Learn More About RESTful Web Services.

Example A-1 Simple RESTful Web Service Client Using Jersey 1.18 (JAX-RS 1.1 RI)

```
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/
jersey/helloworld");
        String response = resource.get(String.class);
        System.out.println(response);
    }
}
```

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.

Optionally, you can pass client configuration properties, defined in Table A-2, when creating the client instance by defining a

`com.sun.jersey.api.client.config.ClientConfig` and passing the information to the `create` method. See the `ClientConfig` interface in the *jersey-bundle 1.18 API*.

Table A-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 A-2 provides an example of how to create a client instance.

Example A-2 Creating a Client Instance

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

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

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

Example A-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", "password"));
      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);
...
```

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 the following in the *jersey-bundle 1.18 API*:

- `WebResource`
- `AsyncWebResource`

> **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 A-6 provides an example of how to create an instance to a Web resource hosted at `http://example.com/helloworld`.

Example A-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 A-7 provides an example of how to create an instance to an asynchronous Web resource hosted at `http://example.com/helloworld`.

Example A-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.asyncResource("http://example.com/helloWorld");
...
```

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:

- How to Build Requests
- How to Send HTTP Requests
- How to Configure the Accept Header
- How to Pass Query Parameters

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 A-3, followed by the HTTP request method, as described in How to Send HTTP Requests. Examples of how to build a request are provided in the sections that follow.

See the `RequestBuilder` methods in the *jersey 1.18 bundle API*.

Table A-3 Building a Request

Method	Description
`accept()`	Defines the acceptable media types. See How to Configure the Accept Header.

Table A-3 (Cont.) Building a Request

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

How to Send HTTP Requests

Table A-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. See the `Future<V>` interface methods in the *Java Platform, Standard Edition 8 API Specification*.

Table A-4 WebResource Methods to Send HTTP Requests

Method	Description
`get()`	Invoke the HTTP GET method to get a representation of the resource.
`head()`	Invoke the HTTP HEAD method to get the meta-information of the resource.
`options()`	Invoke the HTTP OPTIONS method to get the HTTP methods that the JAX-RS service supports.
`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 A-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 A-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 A-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 A-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. See the `GenericType` class in *jersey-bundle 1.18 API*.

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

Example A-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>>() {});
...
```

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, see the `MultivaluedMap` interface in *Java EE 8 API Documentation*.

Example A-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 A-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);
...
```

How to Configure the Accept Header

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

Example A-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 A-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);
...
```

How to Add a Custom Header

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

Example A-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 A-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);
...
```

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 A-14 provides an example of how to configure a request entity and type.

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

Example A-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);
...
```

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 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`. See `ClientResponse` class in *jersey-bundle 1.18 API*.

- How to Access the Status of Request
- How to Get the Response Entity

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 `ClientResponse.Status` in *jersey-bundle 1.18 API*.

Example A-16 provides an example of how to access the status code of the response.

Example A-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();
...
```

How to Get the Response Entity

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

Example A-17 provides an example of how to get the response entity.

Example A-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);
...
```

More Advanced RESTful Web Service Client Tasks

For more information about advanced RESTful web service client tasks, including those listed below, see the *Jersey 1.18 User Guide*.

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

Support for Jersey 1.18 (JAX-RS 1.1 RI) Deployments Packaged with Pre-3.0 Servlets

For backwards compatibility, deployments that reference a subset of servlet classes are supported in this release of Oracle WebLogic Server.Refer to Table A-5, which lists these servlet classes and describes the corresponding elements to update in the `web.xml` deployment descriptor to package the RESTful web service application with a pre-3.0 servlet.

Table A-5 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`

`<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>`

Table A-5 (Cont.) Packaging the RESTful Web Service Application with Pre-3.0 Servlets

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`
	`...`
	See Packaging With an Application Subclass.
	• The value `resources`. This is the default base URI pattern for RESTful web service applications. See 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 What Happens at Runtime: How the Base URI is Constructed.

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

Example A-18 Updating web.xml for Pre-3.0 Servlets

```
<web-app>
    <servlet>
        <servlet-name>Jersey Web 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>org.foo.rest;org.bar.rest</param-value>
        </init-param>
        ...
    </servlet>
    ...
</web-app>
```