Oracle® Fusion Middleware

Developing and Securing RESTful Web Services for Oracle WebLogic Server



26 (26.1.0.0.0)
G56941-01
November 2026



Oracle Fusion Middleware Developing and Securing RESTful Web Services for Oracle WebLogic Server, 26 (26.1.0.0.0)

G56941-01

Copyright © 2007, 2026, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation i
Conventions i

1 Introduction to RESTful Web Services

Introduction to the REST Architectural Style 1
What are RESTful Web Services? 1
Standards Supported for RESTful Web Service Development on WebLogic Server 2
Roadmap for Implementing RESTful Web Services 3
Learn More About RESTful Web Services 3

2 Developing RESTful Web Services

About RESTful Web Service Development 1
- Summary of Tasks to Develop RESTful Web Services 1
- Example of a RESTful Web Service 2

Defining the Root Resource Class 3
Defining the Relative URI of the Root Resource and Subresources 3
- How to Define the Relative URI of the Resource Class (@Path) 3
- How to Define the Relative URI of Subresources (@Path) 4
- What Happens at Runtime: How the Base URI is Constructed 4

Mapping Incoming HTTP Requests to Java Methods 5
- About the Jersey Bookmark Sample 6
- How to Transmit a Representation of the Resource (@GET) 7
- How to Create or Update the Representation of the Resource (@PUT) 8
- How to Delete a Representation of the Resource (@DELETE) 9
- How to Create, Update, or Perform an Action on a Representation of the Resource (@POST) 9

Customizing Media Types for the Request and Response Messages 10
- How To Customize Media Types for the Request Message (@Consumes) 10

Copyright © 2007, 2026, Oracle and/or its affiliates.

How To Customize Media Types for the Response Message (@Produces) 11
What Happens At Runtime: How the Resource Method Is Selected for Response Messages 11
Extracting Information From the Request Message 11
How to Extract Variable Information from the Request URI (@PathParam) 12
How to Extract Request Parameters (@QueryParam) 13
How to Define the DefaultValue (@DefaultValue) 13
How to Preserve Encoded Request Parameter Values (@Encoded) 14
Building Custom Response Messages 14
Mapping HTTP Request and Response Entity Bodies Using Entity Providers 17
Accessing the Application Context 18
Building URIs 19
Using Conditional GETs 20
Accessing the WADL 21
More Advanced RESTful Web Service Tasks 21

3 Developing RESTful Web Service Clients

Summary of Tasks to Develop RESTful Web Service Clients 1
Example of a RESTful Web Service Client 1
Invoking a RESTful Web Service from a Standalone Client 2
Using the Reactive JAX-RS Client API 3

4 Building, Packaging, and Deploying RESTful Web Service Applications

Building RESTful Web Service Applications 1
Packaging RESTful Web Service Applications 1
Packaging With an Application Subclass 1
Packaging With a Servlet 2
How to Package the RESTful Web Service Application with Servlet 3.0 2
How to Package the RESTful Web Service Application with Pre-3.0 Servlet Descriptors 4
Packaging as a Default Resource 6
Deploying RESTful Web Service Applications 6

5 Securing RESTful Web Services and Clients

About RESTful Web Service Security 1
Securing RESTful Web Services Using web.xml 1
Securing RESTful Web Services Using SecurityContext 2
Securing RESTful Web Services Using Java Security Annotations 3

Copyright © 2007, 2026, Oracle and/or its affiliates.

6 Testing RESTful Web Services

7 Monitoring RESTful Web Services and Clients

About Monitoring RESTful Web Services 1
Monitoring RESTful Web Services Using WLST 1
Enabling the Tracing Feature 4
Disabling RESTful Web Service Application Monitoring 5
Disabling Monitoring for a RESTful Web Service Application Using Jersey Property 5
Disabling Monitoring for a RESTful Web Service Application Using WebLogic Configuration MBean 7
Disabling RESTful Web Service Application Monitoring for a WebLogic Domain 7
Enable Monitoring of Synthetic Jersey Resources in a RESTful Web Service Application 9

8 Using Server-Sent Events in WebLogic Server

Overview of Server-Sent Events (SSE) 1
Using Server-Sent Events 1
Understanding the WebLogic Server-Sent Events API 2
Sample Applications for Server-Sent Events 2

Copyright © 2007, 2026, Oracle and/or its affiliates.

Preface

This documentation describes how to develop Jakarta Enterprise web services for Oracle WebLogic Server 26.

Audience

This documentation is written for software developers who want to develop Jakarta EE web services for Oracle WebLogic Server 26 that conform to the Representational State Transfer (REST) architectural style using Jakarta 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.

Related Documentation

New and Changed WebLogic Server Features

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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

1

Introduction to RESTful Web Services

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

Introduction to the REST Architectural Style

REST is an architectural style for distributed hypermedia systems. For HTTP-based services, resources are identified by URIs and accessed through a uniform interface. Clients and servers exchange representations and metadata about resources; an HTTP response need not contain a representation.

A client sends a request that identifies a resource and states the requested operation. When a message contains a representation, it can convey the resource's current state or a proposed new state. REST also constrains how clients, servers, and intermediaries interact, as summarized in Table 1-1.

Table 1-1 summarizes the REST architectural constraints. Code on demand is optional; the other listed constraints define the style.

Table 1-1 Constraints of the REST Architectural Style

Constraint	Description
Client-server	Separate client concerns from server concerns so each can evolve independently.
Stateless	Each client request contains the information needed to process it; the server does not rely on stored client session context between requests.
Cache	Responses indicate whether their data can be reused for equivalent later requests.
Uniform interface	Identify resources, manipulate them through representations, use self-descriptive messages, and guide application state through hypermedia. HTTP methods are one way an HTTP-based service expresses this interface.
Layered system	A component interacts through its adjacent layer without needing to know whether an intermediary or origin server handles the request.
Code on demand (optional)	A server may extend client functionality by transferring executable code, such as scripts; REST does not require this feature.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Standards Supported for RESTful Web Service Development on WebLogic Server

The Jakarta RESTful Web Services specification provides support for creating web services according to REST architectural style. Jakarta RESTful Web Services use 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.

Jakarta RESTful Web Services 4.0 is a required component of both the Jakarta EE 11 Web Profile and the full Jakarta EE 11 Platform. It integrates with Jakarta Contexts and Dependency Injection (CDI), Jakarta Enterprise Beans (EJB), and Jakarta Servlet.

WebLogic Server 26.1.0.0.0 supports the following Jakarta RESTful Web Services specification:

- Jakarta RESTful Web Services 4.0
- Jersey 4.0.0

> **Note**
>
> Jersey support is provided by default in this release of WebLogic Server. Registration as a shared library is no longer required.

Jersey includes the following functionality:

- Jersey
- Jakarta RESTful Web Services
- JSON processing and streaming

Table 1-2 lists key features delivered with Jersey.

Table 1-2 Key Features in Jersey

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 User Guide* • Accessing REST Resources with the JAX-RS Client API in *The Jakarta EE Tutorial*
Asynchronous communication	Invoke and process requests asynchronously. For complete details, see: • Asynchronous Services and Clients in the *Jersey User Guide* • Advanced Features of the Client API in *The Jakarta EE Tutorial*

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 1-2 (Cont.) Key Features in Jersey

Key Feature	Description
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 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 earlier Jersey versions.	Migration Guide in *Jersey User Guide*

Learn More About RESTful Web Services

For additional information about RESTful web services, review the resources in Table 1-4.

Table 1-4 Resources for More Information

Resource	Link
Jersey User Guide	*Jersey User Guide*
Jersey API Javadoc	*Jersey API Documentation*
Community Wiki for Project Jersey	`https://jersey.github.io/`
Jakarta RESTful Web Services Specification	`https://jakarta.ee/specifications/restful-ws/`
JAX-RS API Javadoc	`https://jakartaee.github.io/rest/apidocs/`

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 1-4 (Cont.) Resources for More Information

Resource	Link
JAX-RS Project	https://github.com/jakartaee/rest
RESTful Web Services (JAX-RS) sample	Sample Application and Code Examples in *Understanding Oracle WebLogic Server*.
The Jakarta EE Tutorial: Building RESTful Web Services With JAX-RS	https://jakarta.ee/learn/docs/jakartaee-tutorial/9.1/websvcs/rest/rest.html
"Representational State Transfer (REST)" in *Architectural Styles and the Design of Network-based Software Architectures* (Dissertation by Roy Fielding)	http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Copyright © 2007, 2026, Oracle and/or its affiliates.

2

Developing RESTful Web Services

To develop Jakarta Enterprise web services that conform to the Representational State Transfer (REST) architectural style using Jakarta 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 Jakarta 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.
For information about developing RESTful web services using Oracle JDeveloper, see Creating RESTful Web Services and Clients in *Developing Applications with Oracle JDeveloper*.

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

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 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-1 (Cont.) Summary of Tasks to Develop RESTful Web Services

Task	More Information
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 `jakarta.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`. For example, if the application is deployed at the `/rest` context root and the REST servlet is mapped to `/resources/*`, the full resource URL on the `http://examples.com` origin is `http://examples.com/rest/resources/helloworld`. The context root is `/rest`, not the origin URL. 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 jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.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")
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
   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 `jakarta.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)

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 jakarta.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 `jakarta.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).

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 jakarta.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 user name does not match the requirements, a `404 (Not Found)` response will be sent to the client.

See the `@Path` annotation in the *Jakarta EE 11 API Documentation*.

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

Add the `jakarta.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 jakarta.ws.rs.GET;
import jakarta.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`

Copyright © 2007, 2026, Oracle and/or its affiliates.

- *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 `jakarta.ws.rs.ApplicationPath` annotation to the class that extends `jakarta.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 Jakarta 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 jakarta.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

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-2 (Cont.) jakarta.ws.rs Annotations for Mapping HTTP Requests to Java Methods

Annotation	Description	Idempotent
`@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 *Jakarta EE 11 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. See the `@OPTIONS` annotation in the *Jakarta EE 11 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.	Yes
`@HttpMethod`	Indicates that the annotated method should be used to handle HTTP requests. See the `@HttpMethod` annotation in the *Jakarta EE 11 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

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

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

Table 2-3 About the Jersey Bookmark Sample

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

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

- Accessing the bookmark sample at `https://repo1.maven.org/maven2/org/glassfish/jersey/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/`

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

The `jakarta.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 *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.GET;
import jakarta.ws.rs.Produces;
import jakarta.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 jakarta.ws.rs.GET;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.Path;
...
public class BookmarkResource {
...
    @GET

    @Produces(MediaType.APPLICATION_JSON)

    public JSONObject getBookmark() {
        return asJson();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    }
...
    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 `jakarta.ws.rs.PUT` annotation creates or updates the representation of the specified resource identified by the URI. See the `@PUT` annotation in the *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.PUT;
import jakarta.ws.rs.Produces;
import jakarta.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);
        }});
    }

}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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

The `jakarta.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 *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.DELETE;
import jakarta.ws.rs.Produces;
import jakarta.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 `jakarta.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 *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.POST;
import jakarta.ws.rs.Produces;
...
public class BookmarksResource {
...
    @POST

    @Consumes(MediaType.APPLICATION_JSON)

    public Response postForm(JSONObject bookmark) throws JSONException {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
        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 `jakarta.ws.rs.Consumes` or `jakarta.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)

The `jakarta.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 *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.Consumes;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
...
@Path("/helloworld")
public class helloWorld {
...
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    @POST
    @Consumes("text/plain")
    public void postMessage(String message) {
       // Store the message
   }
}
```

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

The `jakarta.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 *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.Produces;
import jakarta.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

For a request with `Accept: text/html`, the runtime selects `doGetAsHtml`. For `Accept: text/plain`, it selects `doGetAsPlainText`. If both are acceptable, such as `Accept: text/html;q=1.0, text/plain;q=0.5`, the client quality values favor the HTML method, subject to the normal Jakarta REST selection rules.

Extracting Information From the Request Message

The `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 2-4 jakarta.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 *Jakarta EE 11 API Documentation*. For additional usage information, see Parameter Annotations (@*Param) in the *Jersey 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 *Jakarta EE 11 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 How to Preserve Encoded Request Parameter Values (@Encoded).
`@FormParam`	Extract information from an HTML form of the type `application/x-www-form-urlencoded`. See the `@FormParam` annotation in the *Jakarta EE 11 API Documentation*.
`@HeaderParam`	Extract information from the HTTP headers to initialize the value of a method parameter. See the `@HeaderParam` annotation in the *Jakarta EE 11 API Documentation*.
`@MatrixParam`	Extract information from the URI path segments to initialize the value of a method parameter. See the `@MatrixParam` annotation in the *Jakarta EE 11 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 `jakarta.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 jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.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) {
      ...
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    }
}
```

How to Extract Request Parameters (@QueryParam)

Add the `jakarta.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 *Jakarta EE 11 API Documentation*.

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 an optional query parameter is absent and no `@DefaultValue` is specified, a primitive `int` parameter receives `0`. Use `Integer` when `null` must distinguish absence from zero. If conversion of a present query value to int fails, Jakarta REST maps the error to HTTP 404 (`NotFoundException`). An application can validate and map input differently when it explicitly implements that behavior.

Example 2-13 Extracting Request Parameters (@QueryParam)

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

How to Define the DefaultValue (@DefaultValue)

Add the `jakarta.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 *Jakarta EE 11 API Documentation*.

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

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

How to Preserve Encoded Request Parameter Values (@Encoded)

By default, Jakarta REST decodes applicable path, query, form, and matrix parameter values before injection. Add `@Encoded` to preserve the encoded form of a parameter value instead. The annotation does not encode an unencoded value.

In Example 2-15, `@Encoded` preserves the encoded form of the `@PathParam("username")` value.

Example 2-15 Preserving Encoded Request Parameter Values Using @Encoded.

```
package samples.helloworld;

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

import jakarta.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.

For a PUT resource method that returns `void`, Jakarta REST uses HTTP 204 No Content by default. A non-null returned entity uses 200 OK by default. When a PUT creates a resource, the application can explicitly return `Response.created(uri).build()` to send 201 Created with the resource URI.

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 `jakarta.ws.rs.core.Response` class.

Copyright © 2007, 2026, Oracle and/or its affiliates.

An application can extend the `Response` class directly or use one of the static Response methods to create a jakarta.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 *Jakarta EE 11 API Documentation*.

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 an 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 *Jakarta EE 11 API Documentation*.

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.
`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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Method	Description
`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 jakarta.ws.rs.Path;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.PathParam;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.core.ResponseBuilder;
...
@Path("/content")
public class getDocs {
   @GET
   @Path("{id}")
   public Response getHTMLDoc(@PathParam("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 `jakarta.ws.rs.core.GenericEntity` object to preserve the generic type. See the `GenericEntity` methods in the *Jakarta EE 11 API Documentation*.

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 jakarta.ws.rs.Path;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.PathParam;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.core.ResponseBuilder;
import jakarta.ws.rs.core.GenericEntity;
...
@Path("/content")
public class getDocs {
   @GET
   @Path("{id}")
   public Response getHTMLDoc(@PathParam("id") int docId)
   {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
      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 (*/*)
`jakarta.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`)
`jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

Entity Provider	Description

`jakarta.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> { ... }`

`jakarta.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 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 `jakarta.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 *Jakarta EE 11 API Documentation*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

    @Context UriInfo uriInfo;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    ...


    @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 `jakarta.ws.rs.core.Request` contextual interface enabling you to perform conditional GETs. You call the `evaluatePreconditions()` method and pass a `jakarta.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-Modified-Since` headers, respectively, if these headers are sent with the request.

Call `request.evaluatePreconditions` with the current entity tag or last-modified date. If it returns a `ResponseBuilder`, build and return that precondition response. If it returns `null`, return the current representation normally.

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

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.EntityTag;
import jakarta.ws.rs.core.Request;
import jakarta.ws.rs.core.Response;

@Path("/employee/123")
public class EmployeeResource {
    @GET
    @Produces("text/plain")
    public Response get(@Context Request request) {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
            String body = "Employee 123";
            EntityTag tag = new EntityTag("employee-123-v1");
            Response.ResponseBuilder precondition =
    request.evaluatePreconditions(tag);
            if (precondition != null) {
                return precondition.build();
            }
            return Response.ok(body).tag(tag).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 User Guide* provides information about more advanced RESTful web service development tasks:

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

3

Developing RESTful Web Service Clients

You can develop Jakarta EE web service clients that conform to the Representational State Transfer (REST) architectural style using the Jakarta RESTful Web Services specification and the Jersey implementation.

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.

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

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

```
package samples.helloworld.client;

import jakarta.ws.rs.client.Client;
import jakarta.ws.rs.client.ClientBuilder;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
import jakarta.ws.rs.client.WebTarget;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

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

For complete details, see Client API in *Jersey 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Using the Reactive JAX-RS Client API

The Reactive Client API is part of the Jakarta RESTful Web Services 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, 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, use the `CompletionStage` interface to manage large number of methods dedicated for asynchronous computations.

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

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

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

Copyright © 2007, 2026, Oracle and/or its affiliates.

4

Building, Packaging, and Deploying RESTful Web Service Applications

Oracle WebLogic Server provides the components and utilities you need to package and deploy Jakarta EE web services that conform to the Representational State Transfer (REST) architectural style using the Jakarta RESTful Web Services specification and the Jersey implementation.

This chapter includes the following sections:

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 `jakarta.ws.rs.core.Application` to define the components of a RESTful web service application deployment and provide additional metadata. You can add a `jakarta.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 `jakarta.ws.rs.core.Application` to define the components of a RESTful web service application deployment and provides

Copyright © 2007, 2026, Oracle and/or its affiliates.

additional metadata. See `jakarta.ws.rs.core.Application` in the *Jakarta EE 11 API Documentation*.

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 `jakarta.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 *Jakarta EE 11 API Documentation*.

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 `jakarta.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 jakarta.ws.rs.core.Application

```
import jakarta.ws.rs.core.Application;
import jakarta.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 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.

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 `jakarta.ws.rs.core.Application`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `jakarta.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 `jakarta.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 `jakarta.ws.rs.core.Application` subclass. For example:
	`package test;`
	`@ApplicationPath("res")`
	`public class MyJaxRsApplication extends jakarta.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 `jakarta.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 `jakarta.ws.rs.core.Application` is **not** packaged with `web.xml`, then define the elements as described in Table 4-3.

Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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 `jakarta.ws.rs.core.Application` is **not** packaged with `web.xml`.

> **Note**
>
> The Jakarta RESTful Web Services 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 `jakarta.ws.rs.Application` as described in the *Jersey User Guide*. The packaging method defined in this section is not supported by the Jakarta RESTful Web Services 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 Servlet Descriptors

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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Table 4-4 Packaging the RESTful Web Service Application with Pre-3.0 Servlet Descriptors

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>`	Set this element to define the class that extends the `jakarta.ws.rs.core.Application`:
	`<init-param>`
	`  <param-name>`
	`  jakarta.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 `jakarta.ws.rs.core.Application` subclass. For example:
	`package test;`
	`@ApplicationPath("res")`
	`public class MyJaxRsApplication extends jakarta.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 `jakarta.ws.rs.core.Application` is **not** packaged with `web.xml`.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `jakarta.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*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

5

Securing RESTful Web Services and Clients

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

This chapter includes the following sections:

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 `jakarta.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 Jakarta EE Web applications. For complete details, see:

- Developing Secure Web Applications in *Developing Applications with the WebLogic Security Service*.
- Securing Web Applications in *The Jakarta EE 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. In `web.xml`, use `<login-config>` to select the authentication method, such as `BASIC`. Its `<realm-name>` supplies the HTTP Basic challenge realm label; it does not select or change the WebLogic domain security realm that authenticates users.
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*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Example 5-1 Securing RESTful Web Services Using Basic Authentication

The following example demonstrates how to secure a Jersey Jakarta 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>
      <url-pattern>/orders/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-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>
```

An HTTPS listener and appropriate TLS configuration must be available for `CONFIDENTIAL` transport.

Securing RESTful Web Services Using SecurityContext

The `jakarta.ws.rs.core.SecurityContext` interface provides access to security-related information for a request. The `SecurityContext` provides functionality similar to `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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

For more information, see the following topics in the *Jakarta EE 11 API Documentation*:

- `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 jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.ForbiddenException;
import jakarta.ws.rs.core.SecurityContext;
import jakarta.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 ForbiddenException("User is unauthorized.");
        }
```

Securing RESTful Web Services Using Java Security Annotations

The `jakarta.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 jakarta.annotation.security annotations in the *Jersey 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

Copyright © 2007, 2026, Oracle and/or its affiliates.

`@RolesAllowed` 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 jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.annotation.security.RolesAllowed;


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

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

See also:

- Specifying Authorized Users by Declaring Security Roles in *The Jakarta EE Tutorial*
- `jakarta.annotation.security` Javadoc

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Jakarta 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 Remote Console	To validate the application deployment and construct the URL to view the WADL file, see Test Application Deployment in the *Oracle WebLogic Remote Console Online Help*.
Send an HTTP request	For the unsecured Example 2-1 deployed at context root `/rest` and servlet path `/resources/*`, run `curl -i http://localhost:7001/rest/resources/helloworld` (replace host and port with the running server). Confirm HTTP 200 and `Hello World!` in the response body. For the secured Example 5-1, use HTTPS and authorized credentials. Do not send Basic credentials over plain HTTP.

Copyright © 2007, 2026, Oracle and/or its affiliates.

7

Monitoring RESTful Web Services and Clients

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

This chapter includes the following sections:

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 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 Jakarta RESTful Web Services components, as described in Enabling the Tracing Feature.

In addition to the monitoring methods described in Table 7-1, Jersey (Jakarta RESTful Web Services) provides additional monitoring features, including support for event listeners and statistics monitoring. See Monitoring Jersey Applications in the *Jersey 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 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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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*.
`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 JaxRsResourceRuntimeMBean in *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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Oracle 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 Oracle 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 and then to the application runtime MBean for the RESTful web service request statistics.

   ```
   cd('ApplicationRuntimes/jaxrs-async')
   cd('ComponentRuntimes')
   cd('AdminServer_/jaxrs-async')
   cd('JaxRsApplications/examples.javaee7.jaxrs.async.MessageApplication/
   RequestStatistics/
   examples.javaee7.jaxrs.async.MessageApplication_RequestStatistics')
   ls()
   ```

5. 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
   ```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
-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>
```

6. Navigate to any of the other runtime MBeans described in Table 7-2 to view additional monitoring information.
7. 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 Jakarta RESTful Web Services 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 tracing facility collects useful information for individual requests from all components of the Jakarta RESTful Web Services 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 tracing facility, see Tracing Support in *Jersey User Guide*.

Copyright © 2007, 2026, Oracle and/or its affiliates.

Disabling RESTful Web Service Application Monitoring

You can disable monitoring for an individual Jersey Jakarta RESTful Web Services 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 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 Jakarta RESTful Web Services 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 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 Jakarta RESTful Web Services application subclass code, or

Copyright © 2007, 2026, Oracle and/or its affiliates.

declaratively using 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 xmlns="http://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://jakarta.ee/xml/ns/jakartaee https://
jakarta.ee/xml/ns/jakartaee/web-app_6_1.xsd"
         version="6.1">
  <servlet>
    <servlet-name>com.examples.MyApplication</servlet-name>
    ...
    <init-param>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
      <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:

Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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 set this property programmatically in the JAX-RS application subclass code, or declaratively using 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_ENABLED
, true);

    return properties;
  }
}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 xmlns="http://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://jakarta.ee/xml/ns/jakartaee https://
jakarta.ee/xml/ns/jakartaee/web-app_6_1.xsd"
         version="6.1">


   <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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

8

Using Server-Sent Events in WebLogic Server

WebLogic Server supports server-sent events through the integration of the Eclipse Jersey library. The Jersey library provides the Reference Implementation (RI) of Jakarta RESTful Web Services.

This chapter includes the following sections:

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 `jakarta.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.

Copyright © 2007, 2026, Oracle and/or its affiliates.

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 Eclipse Jersey. The use of server-sent events through Jersey is supported only in Jakarta RESTful Web Services resources.
From JAX-RS 2.1, the server-sent event APIs are defined in the `jakarta.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, see Server-Sent Events (SSE) Support in the *Jersey User Guide*.

For new SSE applications, use the standard Jakarta REST 4 API in `jakarta.ws.rs.sse`, documented by the Jakarta EE API reference at `jakarta.ws.rs.sse`. Jersey also retains the older Jersey-specific `org.glassfish.jersey.media.sse` package for applications that use that API. Consult the Jersey 4 SSE guide for its compatibility guidance.

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`

Copyright © 2007, 2026, Oracle and/or its affiliates.