RESTful Web Services Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–4867–05
October 2008

081028@21288

Contents

1 Introduction to RESTful Web Services and Jersey 5
What Are RESTful Web Services? 5
How Does Jersey Fit In? 6
Learning More About RESTful Web Services 6

2 Installing Jersey and the Jersey Sample Applications 9
Installing Jersey on GlassFish 9
▼ Adding Jersey to GlassFish 9
Installing Jersey in NetBeans 10
Installing and Running the Jersey Sample Applications 10
Installing the Jersey Sample Applications 10
Running the Jersey Examples 10
▼ Running the Examples from the Command Line 10
▼ Running the Jersey Examples from NetBeans 11

3 Creating a RESTful Resource Class 13
Developing RESTful Web Services with Jersey 13
Overview of a Jersey-Annotated Application 13
What Are Some of the Annotations Defined by JAX-RS? 15
The @Path Annotation and URI Path Templates 16
More on URI Path Template Variables 17
Responding to HTTP Resources 18
The Resource Method Designator Annotations 18
Using Entity Providers to Map HTTP Response and Request Entity Bodies 19
Using @Consumes and @Produces to Customize Requests and Responses 21
The `@Produces` Annotation 21
The `@Consumes` Annotation 22

Extracting Request Parameters 23
Overview of JAX-RS and Jersey: Further Information 27

4 Creating, Deploying, and Running Jersey Applications 29
Using NetBeans to Create Jersey Applications 29
▼ Creating a Jersey-Annotated Web Application using NetBeans IDE 29
▼ Deploying and Testing a Jersey Application using NetBeans IDE 31
Deploying and Testing a Jersey Application Without NetBeans 31
▼ Deploying and Testing a Jersey Application without NetBeans 31
Deploying a RESTful Web Service 32

5 Jersey Sample Applications 35
The Jersey Sample Applications 35
Configuring Your Environment 35
The HelloWorld-WebApp Application 36
Annotating the Resource Class 36
Configuring the Resource with the Runtime 37
▼ Building and Running the HelloWorld-WebApp Application in NetBeans IDE 6.5 37
▼ Building and Running the HelloWorld-WebApp Application with Maven 38
The Storage-Service Application 38
Understanding the Web Resource Classes for the Storage-Service Example 39
▼ Building and Running the Storage-Service Application from the Command Line 44
▼ Exploring the Storage-Service Example 44
Extending the Storage-Service Example 46
Example: The Storage-Service WADL 47
The Bookstore Application 48
Web Resources for Bookstore Application 48
Mapping the URI Path in the Bookstore Example 50
Mapping the URI Paths and JSP Pages 50
▼ Building and Running the Bookstore Application from a Terminal Window 51
▼ Building and Running the Bookstore Application from NetBeans IDE 51
Other Jersey Examples 52

Index 55

◆ ◆ ◆ CHAPTER 1

Introduction to RESTful Web Services and Jersey

This chapter describes the REST architecture, RESTful web services, and Sun's reference implementation for JAX-RS (Java™ API for RESTful Web Services, JSR-311), which is referred to as *Jersey*.

What Are RESTful Web Services?

RESTful web services are services that are built to work best on the web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client-server architecture, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourages RESTful applications to be simple, lightweight, and have high performance.

RESTful web services typically map the four main HTTP methods to the operations they perform : create, retrieve, update, and delete. The following table shows a mapping of HTTP methods to the operations they perform.

TABLE 1–1 Mapping HTTP Methods to Operations Performed

HTTP Method	Operations Performed
GET	Get a resource
POST	Create a resource and other operations, as it has no defined semantics

TABLE 1–1 Mapping HTTP Methods to Operations Performed *(Continued)*

HTTP Method	Operations Performed
PUT	Create or update a resource
DELETE	Delete a resource

How Does Jersey Fit In?

Jersey is Sun's production quality reference implementation for JSR 311: JAX-RS: The Java API for RESTful Web Services. Jersey implements support for the annotations defined in JSR-311, making it easy for developers to build RESTful web services with Java and the Java JVM. Jersey also adds additional features not specified by the JSR.

The latest version of the JAX—RS API's can be viewed at
`https://jsr311.dev.java.net/nonav/javadoc/index.html`

Learning More About RESTful Web Services

The information in this guide focuses on learning about Jersey. If you are interested in learning more about RESTful Web Services in general, here are a few links to get you started.

- The Community Wiki for Project Jersey has loads of information on all things RESTful. You'll find it at `http://wikis.sun.com/display/Jersey/Main`.
- *Fielding Dissertation: Chapter 5: Representational State Transfer (REST)*, at `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.
- *Representational State Transfer*, from Wikipedia, `http://en.wikipedia.org/wiki/Representational_State_Transfer`.
- *RESTful Web Services*, by Leonard Richardson and Sam Ruby. Available from O'Reilly Media at `http://www.oreilly.com/catalog/9780596529260/`.

Some of the Jersey team members discuss topics out of the scope of this tutorial on their blogs. A few are listed below:

- Earthly Powers, by Paul Sandoz, at `http://blogs.sun.com/sandoz/category/REST`.
- Marc Hadley's Blog, at `http://weblogs.java.net/blog/mhadley/`
- Japod's Blog, by Jakub Podlesak, at `http://blogs.sun.com/japod/category/REST`.

You can always get the latest technology and information by visiting the Java Developer's Network. The links are listed below:

- Get the latest on JSR-311, the Java API's for RESTful Web Services (JAX-RS), at `https://jsr311.dev.java.net/`.

- Get the latest on Jersey, the open source JAX-RS reference implementation, at `https://jersey.dev.java.net/`.

◆ ◆ ◆ CHAPTER 2

Installing Jersey and the Jersey Sample Applications

The chapter describes how to download and install Jersey onto the GlassFish™ container. It also describes how to download the Jersey plugin for NetBeans™.

Installing Jersey on GlassFish

The following section provides details for installing Jersey on Sun GlassFishEnterprise Server v3 Prelude (hereafter referred to as GlassFish v3 Prelude, or simply GlassFish). These steps assume that GlassFish is installed on your system. Jersey is installed as an add-on to GlassFish using the Update Tool that ships with GlassFish. The sample applications that ship with Jersey are also installed during this step.

▼ Adding Jersey to GlassFish

This task describes how to download and add Jersey technology to the GlassFish container.

1 **Start the Update Tool.**

There are several ways to start the Update Tool. Here a few of the options:

- From the Windows Start menu, select GlassFish v3 Prelude, then select Start Update Tool.
- From a Windows file browser or command prompt, or from a Unix terminal prompt, change to the directory where GlassFish was installed, then the `bin` directory, and run `updatetool.exe` (Windows) or `updatetool` (Unix).
- From a web browser, open the Admin Console by going to `http://localhost:4848`, then select Update Tool from the left pane.

2 **Click Available Add-ons.**

3 **Select Jersey RESTful Web Services for GlassFish.**

4. **Click Install.**

5. **Accept the license.**

Installing Jersey in NetBeans

The RESTful Web Services plugin comes bundled with NetBeans IDE 6.5 when you select a NetBeans Pack that includes Java Web and EE. No additional steps are needed to configure and use the Jersey APIs with one of these NetBeans packs. If you've installed a pack that doesn't include Jersey, use the update center to install it.

Installing and Running the Jersey Sample Applications

Jersey includes a very thorough collection of sample applications intended to help you become acquainted with using the RESTful APIs. Several of the sample applications are referenced for sample code throughout this tutorial, and several of the sample applications are discussed in some detail in Chapter 5, "Jersey Sample Applications"

Installing the Jersey Sample Applications

If you have installed the Jersey add-on to GlassFish, you will find the sample applications in the directory *glassfish-install*`/jersey/samples`.

If you installed Jersey as part of the NetBeans IDE, you will have to download the sample applications from the repository. To download the version of the samples that are shipping with Jersey FCS 1.0, click here.

Running the Jersey Examples

The Jersey sample applications are built, run, and deployed using Maven. Maven is a software project management tool, similar to Ant. Ant is a build tool for Java programs. Maven is also a build tool, and can in fact run Ant targets, but Maven adds an organization and structure layer to make the build process easier, to provide a uniform build environment, and to generate quality project information.

▼ Running the Examples from the Command Line

After you have downloaded the Jersey samples, follow these steps to run the samples from the command line.

1. **Download and install Maven 2.0.9 or higher from the Apache Maven Project web site at `http://maven.apache.org/download.html`. Make sure to follow the instructions in the Maven `README.html` file which include adding the `maven/bin` directory to your path statement.**

2. **In a terminal window or command prompt, change to the directory for the sample application you'd like to run. For example, to run the HelloWorld sample, change to *jersey*`/samples/helloworld`.**

3. **Read the `README.html` file for the sample, and follow the steps listed for running the sample. For example, to run the HelloWorld sample, run `mvn compile exec:java`, and follow the instructions on the screen and in the `README.html` file.**

Running the Jersey Examples from NetBeans

To run the Jersey samples from NetBeans, follow these steps.

1. **If you didn't do this in the previous task, download and install Maven 2.0.9 or higher from the Apache Maven Project web site at `http://maven.apache.org/download.html`. Make sure to follow the instructions in the Maven `README.html` file which include adding the `maven/bin` directory to your path statement.**

2. **From the NetBeans IDE, install the Maven plugin. To do this, select Tools→Plugins, select Maven, click Install, and follow the prompts.**

3. **Configure Maven in NetBeans IDE. To do this, select Tools→Options, select Miscellaneous from the top panel, then select the Maven tab.**

4. **For the External Maven Home field, browse to your Maven installation.**

5. **If the option is available, check Always use external Maven for building projects. Close the dialog.**

6. **From the NetBeans IDE, select File→Open Project, and then browse to the location of the project you'd like to open, for example, `jersey/samples/helloworld`.**

7. **Check Open as Main Project, then click Open Project.**

8. **Right-click the project and select Run.**

9. **Follow the instructions from the prompt. For example, if you're running HelloWorld, enter `http://localhost:9998/helloworld` in a web browser.**

◆ ◆ ◆ CHAPTER 3

Creating a RESTful Resource Class

Root resource classes are POJOs (Plain Old Java Objects) that are either annotated with`@Path` or have at least one method annotated with `@Path` or a request method designator such as `@GET`, `@PUT`, `@POST`, or `@DELETE`. *Resource methods* are methods of a resource class annotated with a request method designator. This section describes how to use Jersey to annotate Java objects to create RESTful web services.

Developing RESTful Web Services with Jersey

The JAX-RS API for developing RESTful web services is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with HTTP-specific annotations to define resources and the actions that can be performed on those resources. Jersey annotations are runtime annotations, therefore, runtime reflection will generate the helper classes and artifacts for the resource, and then the collection of classes and artifacts will be built into a web application archive (WAR). The resources are exposed to clients by deploying the WAR to a Java EE or web server.

Overview of a Jersey-Annotated Application

The following code sample is a very simple example of a root resource class using JAX-RS annotations. The sample shown here is from the samples that ship with Jersey, and which can be found in the following directory of that installation:
`jersey/samples/helloworld/src/main/java/com/sun/jersey/samples/helloworld/resources/H`

```
package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;
```

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

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

The following annotations are used in this example:

- The `@Path` annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path `/helloworld`. This is an extremely simple use of the `@Path` annotation. What makes JAX-RS so useful is that you can embed variables in the URIs. *URI path templates* are URIs with variables embedded within the URI syntax.
- The `@GET` annotation is a request method designator, along with`@POST`, `@PUT`, `@DELETE`, and `@HEAD`, that is defined by JAX-RS, and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "`text/plain`".
- The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client. The above example could be modified to set the cliched message as shown here:

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

What Are Some of the Annotations Defined by JAX-RS?

Here is a listing of some of the Java programming annotations that are defined by JAX-RS, with a brief description of how each is used. Further information on the JAX-RS API's can be viewed at `https://jsr311.dev.java.net/nonav/javadoc/index.html`.

TABLE 3–1 Summary of Jersey Annotations

Annotation	Description
@Path	The `@Path` annotation's value is a relative URI path indicating where the Java class will be hosted, for example, `/helloworld`. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user, and pass it to the application as a variable in the URI, like this, `/helloworld/{username}`.
@GET	The `@GET` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@POST	The `@POST` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PUT	The `@PUT` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@DELETE	The `@DELETE` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@HEAD	The `@HEAD` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PathParam	The `@PathParam` annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.
@QueryParam	The `@QueryParam` annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
@Consumes	The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.

TABLE 3–1 Summary of Jersey Annotations *(Continued)*

Annotation	Description
@Produces	The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client, for example, `"text/plain"`.
@Provider	The `@Provider` annotation is used for anything that is of interest to the JAX-RS runtime, such as `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity, and which can be built using `Response.ResponseBuilder`.

The @Path Annotation and URI Path Templates

The `@Path` annotation identifies the URI path template to which the resource responds, and is specified at the class level of a resource. The `@Path` annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the WAR, and the URL pattern to which the Jersey helper servlet responds.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following `@Path` annotation:

```
@Path("/users/{username}")
```

In this type of example, a user will be prompted to enter their name, and then a Jersey web service configured to respond to requests to this URI path template will respond. For example, if the user entered their user name as Galileo, the web service will respond to the following URL:

```
http://example.com/users/Galileo
```

To obtain the value of the username variable, the `@PathParam`annotation may be used on the method parameter of a request method, as shown in the following code example.

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

If it is required that a user name must only consist of lower and upper case numeric characters, it is possible to declare a particular regular expression that will override the default regular expression, `"[^/]+?"`. The following example shows how this could be used with the `@Path` annotation.

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

In this type of example the username variable will only match user names that begin with one upper or lower case letter and zero or more alpha numeric characters and the underscore character. If a user name does not match that template, then a 404 (Not Found) response will occur.

An `@Path` value may or may not begin with a forward slash (/), it makes no difference. Likewise, by default, an `@Path` value may or may not end in a forward lash (/), it makes no difference, and thus request URLs that end or do not end with a forward slash will both be matched. However, Jersey has a redirection mechanism, which, if enabled, automatically performs redirection to a request URL ending in a `/` if a request URL does not end in a `/` and the matching `@Path` does end in a `/`.

More on URI Path Template Variables

A URI path template has one or more variables, with each variable name surrounded by curly braces, `{` to begin the variable name and `}` to end it. In the example above, `username` is the variable name. At runtime, a resource configured to respond to the above URI path template will attempt to process the URI data that corresponds to the location of `{username}` in the URI as the variable data for `username`.

For example, if you want to deploy a resource that responds to the URI path template `http://example.com/myContextRoot/jerseybeans/{name1}/{name2}/`, you must deploy the WAR to a Java EE server that responds to requests to the `http://example.com/myContextRoot` URI, and then decorate your resource with the following `@Path` annotation:

```
@Path("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

In this example, the URL pattern for the Jersey helper servlet, specified in `web.xml`, is the default:

```
<servlet-mapping>
      <servlet-name>My Jersey Bean Resource</servlet-name>
      <url-pattern>/jerseybeans/*</url-pattern>
</servlet-mapping>
```

A variable name can be used more than once in the URI path template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with `%20`.

Be careful when defining URI path templates that the resulting URI after substitution is valid.

The following table lists some examples of URI path template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

- `name1:jay`
- `name2: gatsby`
- `name3:`
- `location: Main%20Street`
- `question: why`

Note – The value of the `name3` variable is an empty string.

TABLE 3–2 Examples of URI path templates

URI Path Template	URI After Substitution
`http://example.com/{name1}/{name2}/`	`http://example.com/jay/gatsby/`
`http://example.com/{question}/{question}/{question}/`	`http://example.com/why/why/why/`
`http://example.com/maps/{location}`	`http://example.com/maps/Main%20Street`
`http://example.com/{name3}/home/`	`http://example.com//home/`

Responding to HTTP Resources

The behavior of a resource is determined by the HTTP methods (typically, `GET`, `POST`, `PUT`, `DELETE`) to which the resource is responding.

The Resource Method Designator Annotations

A *request method designator*annotations are runtime annotations, defined by JAX-RS, and which correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods using the request method designator annotations. The behavior of a resource is determined by which of the HTTP methods the resource is responding to. Jersey defines a set of request method designators for the common HTTP methods: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@HEAD`, but you can create your own custom request method designators. Creating custom request method designators is outside the scope of this document.

The following example is an extract from the storage service sample that shows the use of the `PUT`method to create or update a storage container.

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri =  uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

By default the JAX-RS runtime will automatically support the methods `HEAD` and `OPTIONS` if not explicitly implemented. For `HEAD`, the runtime will invoke the implemented `GET` method (if present) and ignore the response entity (if set). For `OPTIONS`, the `Allow` response header will be set to the set of HTTP methods support by the resource. In addition Jersey will return a WADL document describing the resource.

Methods decorated with request method designators must return `void`, a Java programming language type, or a `javax.ws.rs.core.Response` object. Multiple parameters may be extracted from the URI using the `PathParam` or `QueryParam` annotations as described in "Extracting Request Parameters" on page 23. Conversion between Java types and an entity body is the responsibility of an entity provider, such as `MessageBodyReader` or `MessageBodyWriter`. Methods that need to provide additional metadata with a response should return an instance of `Response`. The `ResponseBuilder` class provides a convenient way to create a `Response` instance using a builder pattern. The HTTP `PUT` and `POST` methods expect an HTTP request body, so you should use a `MessageBodyReader` for methods that respond to `PUT` and `POST` requests.

Using Entity Providers to Map HTTP Response and Request Entity Bodies

Entity providers supply mapping services between representations and their associated Java types. There are two types of entity providers: `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body using a `MessageBodyWriter`. If the application needs to supply additional metadata, such

as HTTP headers or a different status code, a method can return a `Response` that wraps the entity, and which can be built using `Response.ResponseBuilder`.

The following list contains the standard types that are supported automatically for entities. You only need to write an entity provider if you are not choosing one of the following, standard types.

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

The following example shows how to use `MessageBodyReader` with the `@Consumes` and `@Provider` annotations:

```
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> {
```

The following example shows how to use `MessageBodyWriter` with the `@Produces` and `@Provider` annotations:

```
@Produces("text/html")
@Provider
public class FormWriter implements MessageBodyWriter<Hashtable<String, String>> {
```

The following example shows how to use `ResponseBuilder`:

```
    @GET
    public Response getItem() {
        System.out.println("GET ITEM " + container + " " + item);

        Item i = MemoryStore.MS.getItem(container, item);
        if (i == null)
            throw new NotFoundException("Item not found");
```

```
        Date lastModified = i.getLastModified().getTime();
        EntityTag et = new EntityTag(i.getDigest());
        ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);
        if (rb != null)
            return rb.build();

        byte[] b = MemoryStore.MS.getItemData(container, item);
        return Response.ok(b, i.getMimeType()).
                lastModified(lastModified).tag(et).build();
    }
```

Using @Consumes and @Produces to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the `javax.ws.rs.Consumes` and `javax.ws.rs.Produces` annotations.

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

The `@Produces` Annotation

The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If `@Produces` is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If it is applied at the method level, it overrides any `@Produces` annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the Jersey runtime sends back an HTTP "406 Not Acceptable" error.

The value of `@Produces` is an array of `String` of MIME types. For example:

```
@Produces({"image/jpeg,image/png"})
```

The following example shows how to apply `@Produces` at both the class and method levels:

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

```
    }

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

The `doGetAsPlainText` method defaults to the MIME media type of the `@Produces` annotation at the class level. The `doGetAsHtml` method's `@Produces` annotation overrides the class-level `@Produces` setting, and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more that one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically, the `Accept` header of the HTTP request declared what is most acceptable. For example if the `Accept` header is `Accept: text/plain`, the `doGetAsPlainText` method will be invoked. Alternatively if the `Accept` header is `Accept: text/plain;q=0.9, text/html`, which declares that the client can accept media types of `text/plain` and `text/html`, but prefers the latter, then the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same `@Produces` declaration. The following code example shows how this is done.

```
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}
```

The `doGetAsXmlOrJson` method will get invoked if either of the media types `application/xml` and `application/json` are acceptable. If both are equally acceptable, then the former will be chosen because it occurs first. The examples above refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors. For more information, see the constant field values of MediaType.

The `@Consumes` Annotation

The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If `@Consumes` is applied at the class level, all the response methods accept the specified MIME types by default. If `@Consumes` is applied at the method level, it overrides any `@Consumes` annotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the Jersey runtime sends back an HTTP “415 Unsupported Media Type” error.

The value of `@Consumes` is an array of `String` of acceptable MIME types. For example:

```
@Consumes({"text/plain,text/html"})
```

The following example shows how to apply `@Consumes` at both the class and method levels:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

The `doPost` method defaults to the MIME media type of the `@Consumes` annotation at the class level. The `doPost2` method overrides the class level `@Consumes` annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 error (Unsupported Media Type) is returned to the client.

The HelloWorld example discussed previously in this section can be modified to set the cliched message using `@Consumes`, as shown in the following code example.

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

In this example, the Java method will consume representations identified by the MIME media type `text/plain`. Notice that the resource method returns void. This means no representation is returned and response with a status code of HTTP 204 (No Content) will be returned.

Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the `@PathParam` parameter to extract a path parameter from the path component of the request URL that matched the path declared in `@Path`. There are six types of parameters you can extract for use in your resource class: query parameters, URI path parameters, form parameters, cookie parameters, header parameters, and matrix parameters.

Query parameters are extracted from the request URI query parameters, and are specified by using the `javax.ws.rs.QueryParam` annotation in the method parameter arguments. The following example (from the sparklines sample application) demonstrates using `@QueryParam` to extract query parameters from the `Query` component of the request URL.

```
@Path("smooth")
@GET
public Response smooth(
        @DefaultValue("2") @QueryParam("step") int step,
        @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
        @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
        @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
        @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
        @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
        @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
        ) { ... }
```

If a query parameter "step" exists in the query component of the request URI, then the "step" value will be extracted and parsed as a 32–bit signed integer and assigned to the step method parameter. If "step" does not exist, then a default value of 2, as declared in the `@DefaultValue` annotation, will be assigned to the step method parameter. If the "step" value cannot be parsed as a 32–bit signed integer, then an HTTP 400 (Client Error) response is returned.

User-defined Java types such as `ColorParam` may be used. The following code example shows how to implement this.

```
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        } else {
            try {
                Field f = Color.class.getField(s);
                return ((Color)f.get(null)).getRGB();
            } catch (Exception e) {
                throw new WebApplicationException(400);
            }
        }
```

```
    }
}
```

`@QueryParam` and `@PathParam` can only be used on the following Java types:

- All primitive types except `char`
- All wrapper classes of primitive types except `Character`
- Have a constructor that accepts a single `String` argument
- Any class with the static method named `valueOf(String)` that accepts a single String argument
- Any class with a constructor that takes a single `String` as a parameter
- `List<T>`, `Set<T>`, or `SortedSet<T>`, where *T* matches the already listed criteria. Sometimes parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values.

If `@DefaultValue` is not used in conjunction with `@QueryParam`, and the query parameter is not present in the request, then value will be an empty collection for `List`, `Set`, or `SortedSet`; null for other object types; and the Java-defined default for primitive types.

URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation. URI parameters are specified using the `javax.ws.rs.PathParam` annotation in the method parameter arguments. The following example shows how to use `@Path` variables and the `@PathParam` annotation in a method:

```
@Path("/{userName}")
public class MyResourceBean {
    ...
    @GET
    public String printUserName(@PathParam("userName") String userId) {
        ...
    }
}
```

In the above snippet, the URI path template variable name `userName` is specified as a parameter to the `printUserName` method. The `@PathParam` annotation is set to the variable name `userName`. At runtime, before `printUserName` is called, the value of `userName` is extracted from the URI and cast to a `String`. The resulting `String` is then available to the method as the `userId` variable.

If the URI path template variable cannot be cast to the specified type, the Jersey runtime returns an HTTP `400 Bad Request` error to the client. If the `@PathParam` annotation cannot be cast to the specified type, the Jersey runtime returns an HTTP `404 Not Found` error to the client.

The `@PathParam` parameter and the other parameter-based annotations, `@MatrixParam`, `@HeaderParam`, `@CookieParam`, and `@FormParam` obey the same rules as `@QueryParam`.

Cookie parameters (indicated by decorating the parameter with `javax.ws.rs.CookieParam`) extract information from the cookies declared in cookie-related HTTP headers. *Header parameters* (indicated by decorating the parameter with `javax.ws.rs.HeaderParam`) extracts information from the HTTP headers. *Matrix parameters* (indicated by decorating the parameter with `javax.ws.rs.MatrixParam`) extracts information from URL path segments. These parameters are beyond the scope of this tutorial.

Form parameters (indicated by decorating the parameter with `javax.ws.rs.FormParam`) extract information from a request representation that is of the MIME media type `application/x-www-form-urlencoded` and conforms to the encoding specified by HTML forms, as described here. This parameter is very useful for extracting information that is POSTed by HTML forms. The following example extracts the form parameter named "name" from the POSTed form data.

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}
```

If it is necessary to obtain a general map of parameter names to values, use code such as that shown in the following example , for query and path parameters.

```
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

Or code such as the following for header and cookie parameters:

```
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = ui.getRequestHeaders();
    Map<String, Cookie> pathParams = ui.getCookies();
}
```

In general `@Context` can be used to obtain contextual Java types related to the request or response.

For form parameters it is possible to do the following:

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(MultivaluedMap<String, String> formParams) {
    // Store the message
}
```

Overview of JAX-RS and Jersey: Further Information

The following documents contain information that you might find useful when creating applications using Jersey and JAX-RS.

- Overview of JAX-RS 1.0 Features

  This document contains some of the information from this tutorial, as well as additional topics such as *Representations and Java types*, *Building Responses*, *Sub-resources*, *Building URIs*, *WebApplicationException and mapping Exceptions to Responses*, *Conditional GETs and Returning 304 (Not Modified) Responses*, *Life-cycle of root resource classes*, *Security*, `Rules of Injection`, *Use of @Context*, and *APIs defined by JAX-RS*.

- Overview of Jersey 1.0 Features

  This document contains the following topics: Deployment, Web-Deployment Using Servlet, Embedded-Web-Deployment Using GlassFish, Embedded-Deployment Using Grizzly, Embedded-Web-Deployment Using Grizzly, Client-Side API, Client-Side Filters, Integration with Spring, JSON, JAXB, Module View Controller with JSPs, Resource Class Life-Cycle, Resource Class Instantiation, Web Application Description Language (WADL) Support, Pluggable Templates for Model View Controller, Server-Side Filters URI utilities, Web Application Reloading, Pluggable Injection, Pluggable Life-Cycle, Pluggable HTTP containers, and Pluggable IoC Integration.

◆ ◆ ◆ CHAPTER 4

Creating, Deploying, and Running Jersey Applications

This chapter provides an introduction to creating, deploying, and running your own Jersey applications. This section demonstrates the steps that you would take to create, build, deploy, and run a very simple web application that is annotated with Jersey.

Another way that you could learn more about deploying and running Jersey applications is to review the many sample applications that ship with Jersey. When you install the Jersey add-on to GlassFish, these samples are installed into the *glassfish.home*`/jersey/samples` directory. If you have installed Jersey in another way, read the section "Installing and Running the Jersey Sample Applications" on page 10 for information on how to download the sample applications. There is a `README.html` file for each sample that describes the sample and describes how to deploy and test the sample, and there is a Project Object Model file, `pom.xml`, that is used by Maven to build the project.

Note – For GlassFish v3 Prelude release only, Jersey-based applications that use the JSP Views feature need to bundle the Jersey JAR files with the application WAR file. An example of how this is done can be found in the `bookstore` sample application.

Using NetBeans to Create Jersey Applications

This section gives a simple introduction to using Jersey in NetBeans.

▼ Creating a Jersey-Annotated Web Application using NetBeans IDE

This section describes, using a very simple example, how to create a Jersey-annotated web application.

Before You Begin Before you can deploy a Jersey application using NetBeans, you must have installed the RESTful Web Services plugin, as described in "Installing Jersey in NetBeans" on page 10.

1. **In NetBeans IDE, create a simple web application. For this example, we will work with a very simple "Hello, World" web application.**
   a. **Open NetBeans IDE.**
   b. **Select File→New Project.**
   c. **From Categories, select Java Web. From Projects, select Web Application. Click Next.**
   d. **Enter a project name, `HelloWorldApp`, click Next.**
   e. **Make sure the Server is GlassFish v3 Prelude.**
   f. **Click Finish.**
2. **The project will be created. The file `index.jsp` will display in the Source pane.**
3. **Right-click the project and select New, then select RESTful Web Services from Patterns.**
   a. **Select Singleton to use as a design pattern. Click Next.**
   b. **Enter a Resource Package name, like `HelloWorldResource`. For MIME Type select text/html.**
   c. **Enter `/helloworld` in the Path field. Enter `HelloWorld` in the Resource Name field.**
   d. **Click Finish.**
      A new resource, `HelloWorldResource.java`, is added to the project and displays in the Source pane.
4. **In `HelloWorldResource.java`, modify or add code to resemble the following example.**

```
/**
     * Retrieves representation of an instance of helloworld.HellowWorldResource
     * @return an instance of java.lang.String
     */
    @GET
    @Produces("text/html")
    public String getXml() {
        return "<html><body><h1>Hello World!</body></h1></html>";
    }

    /**
     * PUT method for updating or creating an instance of HelloWorldResource
```

```
     * @param content representation for the resource
     * @return an HTTP response with content of the updated or created resource.
     */
    @PUT
    @Consumes("application/xml")
    public void putXml(String content) {
    }
```

▼ Deploying and Testing a Jersey Application using NetBeans IDE

This section describes building, deploying, and testing a Jersey-annotated application using NetBeans IDE.

1. **Right-click the project node and click Test RESTful Web Services.**
   This step will deploy the application and bring up a test client in the browser.
2. **When the test client displays, navigate to the helloworld resource and click the Test button.**
   The words `Hello World!` will display in the Response window.

See Also For other sample applications that demonstrate deploying and running Jersey applications using NetBeans, read "Building and Running the HelloWorld-WebApp Application in NetBeans IDE 6.5" on page 37, "Building and Running the Bookstore Application from NetBeans IDE" on page 51, or look at the tutorials on the NetBeans tutorial site, such as the one titled Getting Started with RESTful Web Services: NetBeans 6.5. This tutorial includes a section on creating a CRUD application from a database.

Deploying and Testing a Jersey Application Without NetBeans

The following sections describes how to deploy and run a Jersey application *without* using the NetBeans IDE. This example describes copying and editing an existing Jersey sample application. An example that starts from scratch can be found here.

▼ Deploying and Testing a Jersey Application without NetBeans

The easiest way to create and run an application without NetBeans is to copy and edit one of the Jersey sample applications. We'll use the simplest sample application, `HelloWorld`, to demonstrate one way you could go about creating your own application without NetBeans IDE.

Before You Begin Before you can deploy an application to GlassFish from the command line, you must have downloaded and installed Jersey onto GlassFish, as described in "Adding Jersey to GlassFish" on page 9.

1. **Copy the `HelloWorld` application to a new directory named `helloworld2`.**
2. **Do a search for all *directories* named `helloworld` and rename them to `helloworld2`.**
3. **Search again for all *files* containing the text `helloworld` and edit these files to replace this text with `helloworld2`.**
4. **Using a text editor, open the file**
   `jersey/samples/helloworld2/src/main/java/com/sun/jersey/samples/helloworld/resources/Hello`
5. **Modify the text that is returned by the resource to `Hello World 2`. Save and close the file.**
6. **Use Maven to compile and deploy the application. For this sample application, it is deployed onto Grizzly. Enter the following command from the command line to compile and deploy the application: `mvn compile exec:java`.**
7. **Open a web browser, and enter the URL to which the application was deployed, which in this examples is `http://localhost:9998/helloworld2`. `Hello World 2` will display in the browser.**

See Also You can learn more about deploying and running Jersey applications by reviewing the many sample applications that ship with Jersey. There is a `README.html` file for each sample that describes the sample and describes how to deploy and test the sample, and there is a Project Object Model file, `pom.xml`, that is used by Maven to build the project. Find a project that is similar to one you are hoping to create and use it as a template to get you started.

An example that starts from scratch can be found here.

For questions regarding Jersey sample applications, visit the Jersey Community Wiki page, or send an email to the users mailing list, `users@jersey.dev.java.net`.

Deploying a RESTful Web Service

This section is taken from the document titled Overview of JAX-RS 1.0 Features.

JAX-RS provides the deployment-agnostic abstract class Application for declaring root resource classes and root resource singleton instances. A Web service may extend this class to declare root resource classes, as shown in the following code example.

```
public class MyApplicaton extends Application {
    public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
```

```
            s.add(HelloWorldResource.class);
            return s;
        }
    }
```

Alternatively, it is possible to reuse a Jersey implementation that scans for root resource classes given a classpath or a set of package names. Such classes are automatically added to the set of classes that are returned by the `getClasses` method. For example, the following code example scans for root resource classes in packages `org.foo.rest`, `org.bar.rest`, and in any their sub-packages.

```
public class MyApplication extends PackagesResourceConfig {
    public MyApplication() {
        super("org.foo.rest;org.bar.rest");
    }
}
```

For servlet deployments, JAX-RS specifies that a class that implements `Application` may be declared instead of a servlet class in the *<server-class>* element of the application deployment descriptor, `web.xml`. As of this writing, this is not currently supported for Jersey. Instead it is necessary to declare the Jersey-specific servlet and the `Application` class, as shown in the following code example.

```
<web-app>
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>MyApplication</param-value>
        </init-param>
    </servlet>
    ....
```

An even simpler approach is to let Jersey choose the `PackagesResourceConfig` implementation automatically by declaring the packages as shown in the following code.

```
<web-app>
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>com.sun.jersey.config.property.packages</param-name>
            <param-value>org.foo.rest;org.bar.rest</param-value>
        </init-param>
    </servlet>
    ....
```

JAX-RS also provides the ability to obtain a container-specific artifact from an `Application` instance. In the following code example, Jersey supports using Grizzly.

```
SelectorThread st =
    RuntimeDelegate.createEndpoint(new MyApplication(),
    SelectorThread.class);
```

Jersey also provides Grizzly helper classes to deploy the `ServletThread` instance at a base URL for in-process deployment. The Jersey samples provide many examples of servlet-based and Grizzly-in-process-based deployments.

◆ ◆ ◆ CHAPTER 5

Jersey Sample Applications

This chapter discusses some sample applications that demonstrate how to create and use the Jersey annotations in your application. When you install the Jersey add-on to GlassFish, these samples are installed into the *glassfish.home*`/jersey/samples` directory. If you have installed Jersey in another way, read the section "Installing and Running the Jersey Sample Applications" on page 10 for information on how to download the sample applications.

The Jersey Sample Applications

All of the Jersey sample applications can be executed from the command line or from NetBeans IDE 6.5 using Maven version 2.0.9 or greater. Maven makes it easy to distribute the samples without redistributing third-party dependencies, which can be very large.

There are three samples included in the tutorial that demonstrate how to create and use resources. They are:

- `HelloWorld-WebApp` is a simple "Hello, world" sample that responds to HTTP `GET` requests.
- `Storage-Service` demonstrates a simple, in-memory, web storage service.
- `Bookstore` demonstrates how to connect JSP pages to resources.

Configuring Your Environment

To run the sample applications, you must install Jersey onto either GlassFish v3 Prelude or NetBeans IDE 6.5, and you need to install and configure Maven. Here are the links to more information on these topics.

- Installing Jersey onto GlassFish v3 Prelude. This task is described in "Adding Jersey to GlassFish" on page 9.

- Installing Jersey as part of the NetBeans Web and Java EE pack, or by installing Jersey as a separate plugin to NetBeans IDE 6.5. This task is described in "Installing Jersey in NetBeans" on page 10.
- Installing Maven to run the sample applications. This task is described in "Running the Jersey Examples" on page 10.

The HelloWorld-WebApp Application

This section discusses the `HelloWorld-WebApp` application that ships with Jersey. The `HelloWorld-WebApp` application is a "Hello, world" application that demonstrates the basics of developing a resource. There is a single class, `HelloWorldResource` that contains one method, `getClichedMessage` that produces a textual response to an HTTP `GET` request with a greeting that is sent back as plain text.

Annotating the Resource Class

The following code is the contents of the `com.sun.jersey.samples.helloworld.resources.HelloWorldResource` class:

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

@Path("/helloworld")
public class HelloWorldResource {

        @GET
        @Produces("text/plain")
        public String getClichedMessage() {
            return "Hello World";
    }
}
```

In this example, the following annotations are processed at runtime:

- The `@Path` annotation specifies that the Java class will be hosted at the URI path `/helloworld`.
- The `@GET` annotation specifies that the Java method will process HTTP GET requests.
- The `@Produces` annotation specifies that the Java method will produce content identified by the MIME media type `text/plain`.

Configuring the Resource with the Runtime

The `helloworld-webapp/src/main/webapp/WEB-INF/web.xml` deployment descriptor for `HelloWorld-Webapp`contains the settings for configuring your resource with the JAX-RS API runtime:

```
<servlet>
     <servlet-name>Jersey Web Application</servlet-name>
     <servlet-class>
         com.sun.jersey.spi.container.servlet.ServletContainer
     </servlet-class>
            <init-param>
                    <param-name>
                            com.sun.jersey.config.property.packages
                    </param-name>
                    <param-value>
                            com.sun.jersey.samples.helloworld.resources
                    </param-value>
            </init-param>
            <load-on-startup>1</load-on-startup>
    </servlet>
<servlet-mapping>
     <servlet-name>Jersey Web Application</servlet-name>
         <url-pattern>/*</url-pattern>
</servlet-mapping>
```

The `com.sun.jersey.spi.container.servlet.ServletContainer` servlet is part of the JAX-RS API runtime, and works with the generated `HelloWorldResource` class to get the resources provided by your application. The `<servlet-mapping>` elements specify which URLs your application responds to relative to the context root of your WAR. Both the context root and the specified URL pattern prefix the URI Template specified in the `@Path` annotation in the resource class file. In the case of the `HelloWorld-WebApp` sample application, `@Path` is set to `/helloworld`, the URL pattern is set to the wild card character, and the context root specified in `sun-web.xml` is `/helloworld-webapp`, so the resource will respond to requests of the form:

```
http://<server>:<server port>/helloworld-webapp/helloworld
```

▼ Building and Running the HelloWorld-WebApp Application in NetBeans IDE 6.5

1. **Make sure that Maven is installed and configured, as described in "Running the Jersey Examples" on page 10.**

2. **Select File→Open Project in NetBeans IDE 6.5.**

3. **Navigate to `jersey/samples`, select `HelloWorld-WebApp`, and click OK.**

4. **Right click the `HelloWorld-WebApp` project in the Projects pane and select Run.**

   This will generate the helper classes and artifacts for your resource, compile the classes, package the files into a WAR file, and deploy the WAR to your GlassFish v3 Prelude instance.

5. **If a web browser doesn't open automatically to display the output, you may need to open a web browser and enter the URL for the application.**

   ```
   http://localhost:8080/helloworld-webapp/helloworld
   ```

   You will see the following output in your web browser:

   ```
   Hello World
   ```

▼ Building and Running the HelloWorld-WebApp Application with Maven

1. **Make sure that Maven is installed and configured, as described in "Running the Jersey Examples" on page 10.**

2. **Open a terminal prompt and navigate to *jersey.home*`/samples/HelloWorld-WebApp`.**

3. **Enter `mvn glassfish:run` and press Enter.**

   This will build, package, deploy, and run the web application. It will also start GlassFish if it is not running.

4. **In a web browser navigate to:**

   ```
   http://localhost:8080/helloworld-webapp/helloworld
   ```

   You will see the following output in your web browser:

   ```
   Hello World
   ```

The Storage-Service Application

The `Storage-Service` sample application demonstrates a simple, in-memory, web storage service and test code using the Jersey client API. The web storage service enables clients to create and delete containers. Containers are used to create, read, update, and delete items of arbitrary content, and to search for items containing certain content. A container can be thought of as a hash map of items. The key for the item is specified in the request URI. There are three web resources that are shown below.

The web storage service ensures that content can be cached (by browsers and proxies) by supporting the HTTP features `Last-Modified` and `ETag`. The sample consists of three web resources implemented by the classes described in the following paragraphs.

Note – Two of the resource classes have similar names. `ContainersResource` (plural) and `ContainerResource` (singular).

Understanding the Web Resource Classes for the Storage-Service Example

The first resource class, `com.sun.jersey.samples.storageservice.resources.ContainersResource`, provides metadata information on the containers. This resource references the `ContainerResource` resource using the `@Path` annotation declared on the `ContainersResource.getContainerResource` method. The following code is extracted from `ContainersResource.java`:

```
@Path("/containers")
@Produces("application/xml")
public class ContainersResource {
    @Context UriInfo uriInfo;
    @Context Request request;

    @Path("{container}")
    public ContainerResource getContainerResource(@PathParam("container")
        String container) {
        return new ContainerResource(uriInfo, request, container);
    }

    @GET
    public Containers getContainers() {
        System.out.println("GET CONTAINERS");

        return MemoryStore.MS.getContainers();
    }
}
```

Another resource class , `com.sun.jersey.samples.storageservice.resources.ContainerResource`, provides for reading, creating, and deleting of containers. You can search for items in the container using a URI query parameter. The resource dynamically references the `ItemResource` resource class using the `getItemResource` method that is annotated with `@Path`. The following code is extracted from `ContainerResource.java`:

```
@Produces("application/xml")
public class ContainerResource {
    @Context UriInfo uriInfo;
    @Context Request request;
    String container;

    ContainerResource(UriInfo uriInfo, Request request, String container) {
        this.uriInfo = uriInfo;
        this.request = request;
        this.container = container;
    }

    @GET
    public Container getContainer(@QueryParam("search") String search) {
        System.out.println("GET CONTAINER " + container + ", search = " + search);

        Container c = MemoryStore.MS.getContainer(container);
        if (c == null)
            throw new NotFoundException("Container not found");


        if (search != null) {
            c = c.clone();
            Iterator<Item> i = c.getItem().iterator();
            byte[] searchBytes = search.getBytes();
            while (i.hasNext()) {
                if (!match(searchBytes, container, i.next().getName()))
                    i.remove();
            }
        }

        return c;
    }

    @PUT
    public Response putContainer() {
        System.out.println("PUT CONTAINER " + container);

        URI uri =  uriInfo.getAbsolutePath();
        Container c = new Container(container, uri.toString());

        Response r;
        if (!MemoryStore.MS.hasContainer(c)) {
            r = Response.created(uri).build();
        } else {
            r = Response.noContent().build();
        }
```

```
        MemoryStore.MS.createContainer(c);
        return r;
    }

    @DELETE
    public void deleteContainer() {
        System.out.println("DELETE CONTAINER " + container);

        Container c = MemoryStore.MS.deleteContainer(container);
        if (c == null)
            throw new NotFoundException("Container not found");
    }


    @Path("{item: .+}")
    public ItemResource getItemResource(@PathParam("item") String item) {
        return new ItemResource(uriInfo, request, container, item);
    }

    private boolean match(byte[] search, String container, String item) {
        byte[] b = MemoryStore.MS.getItemData(container, item);

        OUTER: for (int i = 0; i < b.length - search.length; i++) {
             for (int j = 0; j < search.length; j++) {
                if (b[i + j] != search[j])
                    continue OUTER;
            }

            return true;
        }

        return false;
    }
}
```

The last resource class discussed in this sample, `com.sun.jersey.samples.storageservice.resources.ItemResource`, provides for reading, creating, updating, and deleting of an item. The last modified time and entity tag are supported in that it conditionally returns content only if the browser or proxy has an older version of the content. The following code is extracted from `ItemResource.java`:

```
public class ItemResource {
    UriInfo uriInfo;
    Request request;
    String container;
    String item;

    public ItemResource(UriInfo uriInfo, Request request,
```

```
        String container, String item) {
    this.uriInfo = uriInfo;
    this.request = request;
    this.container = container;
    this.item = item;
}

@GET
public Response getItem() {
    System.out.println("GET ITEM " + container + " " + item);

    Item i = MemoryStore.MS.getItem(container, item);
    if (i == null)
        throw new NotFoundException("Item not found");
    Date lastModified = i.getLastModified().getTime();
    EntityTag et = new EntityTag(i.getDigest());
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);
    if (rb != null)
        return rb.build();

    byte[] b = MemoryStore.MS.getItemData(container, item);
    return Response.ok(b, i.getMimeType()).
            lastModified(lastModified).tag(et).build();
}

@PUT
public Response putItem(
        @Context HttpHeaders headers,
        byte[] data) {
    System.out.println("PUT ITEM " + container + " " + item);

    URI uri = uriInfo.getAbsolutePath();
    MediaType mimeType = headers.getMediaType();
    GregorianCalendar gc = new GregorianCalendar();
    gc.set(GregorianCalendar.MILLISECOND, 0);
    Item i = new Item(item, uri.toString(), mimeType.toString(), gc);
    String digest = computeDigest(data);
    i.setDigest(digest);

    Response r;
    if (!MemoryStore.MS.hasItem(container, item)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    Item ii = MemoryStore.MS.createOrUpdateItem(container, i, data);
    if (ii == null) {
```

```
            // Create the container if one has not been created
            URI containerUri = uriInfo.getAbsolutePathBuilder().path("..").
                    build().normalize();
            Container c = new Container(container, containerUri.toString());
            MemoryStore.MS.createContainer(c);
            i = MemoryStore.MS.createOrUpdateItem(container, i, data);
            if (i == null)
                throw new NotFoundException("Container not found");
        }

        return r;
    }

    @DELETE
    public void deleteItem() {
        System.out.println("DELETE ITEM " + container + " " + item);

        Item i = MemoryStore.MS.deleteItem(container, item);
        if (i == null) {
            throw new NotFoundException("Item not found");
        }
    }


    private String computeDigest(byte[] content) {
        try {
            MessageDigest md = MessageDigest.getInstance("SHA");
            byte[] digest = md.digest(content);
            BigInteger bi = new BigInteger(digest);
            return bi.toString(16);
        } catch (Exception e) {
            return "";
        }
    }
}
```

The mapping of the URI path space is shown in the following table.

TABLE 5–1 URI Path Space for Storage-Service Example

URI Path	Resource Class	HTTP Methods
/containers	`ContainersResource`	GET
/containers/{container}	`ContainerResource`	GET, PUT, DELETE
/containers/{container}/{item}	`ItemResource`	GET, PUT, DELETE

Building and Running the Storage-Service Application from the Command Line

1. **Make sure that Maven is installed and configured, as described in "Running the Jersey Examples" on page 10.**

2. **Open a terminal prompt and navigate to** *jersey.home*`/samples/storage-service`**.**

3. **To run the sample, enter** `mvn clean compile exec:java` **and press Enter.**

   This will build, package, and deploy the web storage service using Grizzly, an HTTP web server.

   Tip – To run the application on GlassFish, copy the classes from the example into sources of the web application. Then, create a `web.xml` file that uses the Jersey servlet. The Java classes are not dependent on a particular container.

   ---

4. **To view the Web Application Description Language file (the** *WADL***), open a web browser and navigate to:**

   ```
   http://localhost:9998/storage/application.wadl
   ```

   The WADL description in also shown in this document at "Example: The Storage-Service WADL" on page 47

5. **Leave the terminal window open, and open a new terminal window to continue with exploring this sample.**

Exploring the Storage-Service Example

Once the service is deployed, it is a good idea to see how it works by looking at the contents of the containers, creating a container, adding content to the container, looking at the date and time stamp for the items in the container, searching for content in the containers, modifying content of an item in the containers, deleting an item from the container, and deleting the container. To accomplish these tasks, use the cURL command line tool, as shown in the following steps. cURL is a command line tool for transferring files with URL syntax.

1. **To show which containers currently exist, open a new terminal prompt and enter the following command. This command will henceforth be referred to as the GET command.**

   ```
   curl http://127.0.0.1:9998/storage/containers
   ```

   The response is in XML format. Because there are no containers present, the response is an empty containers element.

2 **To create a container, enter the following at the terminal prompt:**

```
curl -X PUT http://127.0.0.1:9998/storage/containers/quotes
```

This step creates a container called `quotes`. If you run the GET command from the previous step again, it will return information in XML format showing that a container named `quotes` exists at the URI `http://127.0.0.1:9998/storage/containers/quotes`.

3 **Create some content in the quotes container. The following example shows how to do this from the terminal prompt:**

```
curl -X PUT -HContent-type:text/plain --data
            "Something is rotten in the state of Denmark"
            http://127.0.0.1:9998/storage/containers/quotes/1
curl -X PUT -HContent-type:text/plain --data
            "I could be bounded in a nutshell"
            http://127.0.0.1:9998/storage/containers/quotes/2
curl -X PUT -HContent-type:text/plain --data
            "catch the conscience of the king"
            http://127.0.0.1:9998/storage/containers/quotes/3
curl -X PUT -HContent-type:text/plain --data
            "Get thee to a nunnery"
            http://127.0.0.1:9998/storage/containers/quotes/4
```

If you run the GET command again with `/quotes` at the end (`curl http://127.0.0.1:9998/storage/containers/quotes`), it will return information in XML format that show that the `quotes` container contains 4 items. For each item, the following information is displayed: digest, date last modified, MIME type, item name (also referred to as its key), and URI address. For example, the listing for the first item looks like this:

```
<item>
    <digest>7a54c57975de11bffcda5bc6bd92a0460d17ad03</digest>
    <lastModified>2008-10-10T15:011:48+01:00</lastModified>
    <mimeType>text/plain</mimeType>
    <name>1</name>
    <uri>http://127.0.0.1:9998/storage/containers/quotes/1</uri>
</item>
```

4 **You can search for information within the contents of the `quotes` container. For example, the following command would search for the `String` `king`.**

```
curl "http://127.0.0.1:9998/storage/containers/quotes?search=king"
```

This returns the information for the item that contains the text, similar to that shown in the previous step, but not the text itself. The next step shows how to see the contents of the item containing the text `king`.

5 **To get the contents of the item containing the text you found using the search, use the following command:**

```
curl  http://127.0.0.1:9998/storage/containers/quotes/3
```

This step returns the contents of item 3, which is the quote `catch the conscience of the king`.

6. **To delete an item from the container, use the following command:**

```
curl -X DELETE http://127.0.0.1:9998/storage/containers/quotes/3
```

7. **To delete an item from the container, use the following command:**

```
curl -X DELETE http://127.0.0.1:9998/storage/containers/quotes/3
```

You can use the GET command to verify that item 3 has been removed.

8. **To delete the entire `quotes` container, use the following command:**

```
curl -X DELETE http://127.0.0.1:9998/storage/containers/quotes
```

You can use the GET command to verify that the container has been removed.

9. **For a discussion of the caching of HTTP requests and responses, look at the *glassfish.home*`/jersey/samples/Storage-Service/README.html` file.**

If you go back to the terminal window that is running the web storage service, you will see the history of HTTP requests and responses, which will look something like this:

```
GET CONTAINERS
PUT CONTAINER quotes
GET CONTAINERS
PUT ITEM quotes 1
PUT ITEM quotes 2
PUT ITEM quotes 3
PUT ITEM quotes 4
GET CONTAINER quotes, search = null
PUT ITEM quotes 4
PUT ITEM quotes 4
GET CONTAINER quotes, search = king
DELETE ITEM quotes 3
GET CONTAINER quotes, search = null
DELETE CONTAINER quotes
GET CONTAINER quotes, search = null
GET CONTAINERS
```

Extending the Storage-Service Example

This example demonstrates storing and returning plain text strings. This example can easily be modified to store and return arbitrary binary data, for example, images. You can easily store any piece of data with any media type. All you have to do is, in the examples in the previous section, change the `text/plain` parameter to `image/jpeg`, or some other value, and point to a JPEG file. For example, to create some content that is a JPEG file, you could use the following curl command.

```
curl -X PUT -HContent-type:image/jpeg --data
            /home/jersey_logo.jpg
            http://127.0.0.1:9998/storage/containers/images/1
```

To retrieve the contents of the item containing the image, use the following command:

```
curl  http://127.0.0.1:9998/storage/containers/images/1
```

Example: The Storage-Service WADL

This is the Web Application Description Language file (the WADL) that is generated for the `storage-service` sample application:

```
<?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: 1.0-ea-SNAPSHOT 10/02/2008 12:17 PM"/>
    <resources base="http://localhost:9998/storage/">
        <resource path="/containers">
            <method name="GET" id="getContainers">
                <response>
                    <representation mediaType="application/xml"/>
                </response>
            </method>
            <resource path="{container}">
                <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                                type="xs:string" style="template" name="container"/>
                <method name="PUT" id="putContainer">
                    <response>
                        <representation mediaType="application/xml"/>
                    </response>
                </method>
                <method name="DELETE" id="deleteContainer"/>
                <method name="GET" id="getContainer">
                    <request>
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                                                type="xs:string" style="query" name="search"/>
                    </request>
                    <response>
                        <representation mediaType="application/xml"/>
                    </response>
                </method>
                <resource path="{item: .+}">
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                                        type="xs:string" style="template" name="item"/>
                    <method name="PUT" id="putItem">
                        <request>
```

```
                        <representation mediaType="*/*"/>
                    </request>
                    <response>
                        <representation mediaType="*/*"/>
                    </response>
                </method>
                <method name="DELETE" id="deleteItem"/>
                <method name="GET" id="getItem">
                    <response>
                        <representation mediaType="*/*"/>
                    </response>
                </method>
            </resource>
        </resource>
    </resource>
</resources>
</application>
```

The Bookstore Application

The `Bookstore` web application shows how to connect JSP pages to resources. The `Bookstore` web application presents books, CDs, and tracks from CDs. The sample application consists of five web resources, described and outlined in the following section.

This sample demonstrates how to support polymorphism of resources and JSP pages, which means that it allows values of different data types to be handled using a uniform interface. The use of polymorphism makes it possible to add another resource, such as a DVD resource with associated JSP pages, which would extend `Item` without having to change the logic of `Bookstore` or any of the existing JSP pages.

Web Resources for Bookstore Application

The `Bookstore` resource returns a list of items, either CDs or books. The resource dynamically references a `Book` or `CD` resource using the `getItem` method that is annotated with `@Path`. Both the `Book` and `CD` resources inherit from the `Item` class, therefore, the item can be managed polymorphically.

The following snippet of code from `com.sun.jersey.samples.bookstore.resources.Bookstore.java` adds some items to the bookstore application and, using a URI path template to pass in the item ID, provides a method for retrieving the item.

```
@Path("/")
@Singleton
public class Bookstore {
```

```
    private final Map<String, Item> items = new TreeMap<String, Item>();

    private String name;

    public Bookstore() {
        setName("Czech Bookstore");
        getItems().put("1", new Book("Svejk", "Jaroslav Hasek"));
        getItems().put("2", new Book("Krakatit", "Karel Capek"));
        getItems().put("3", new CD("Ma Vlast 1", "Bedrich Smetana", new Track[]{
            new Track("Vysehrad",180),
            new Track("Vltava",172),
            new Track("Sarka",32)}));
    }

    @Path("items/{itemid}/")
    public Item getItem(@PathParam("itemid") String itemid) {
        Item i = getItems().get(itemid);
        if (i == null)
            throw new NotFoundException("Item, " + itemid + ", is not found");

        return i;
    }
   . . .
  }
```

Both the `Book` and the `CD` resources inherit from the `Item` class. This allows the resources to be managed polymorphically. The `Book` resource has a title and an author. The `CD` resource has a title, author, and list of tracks. The `Track` resource has a name and a track length.

The following code snippet from the `CD` resource dynamically references the `Track` resource using the `getTrack` method that is annotated with `@Path`.

```
public class CD extends Item {

    private final Track[] tracks;

    public CD(final String title, final String author, final Track[] tracks) {
        super(title, author);
        this.tracks = tracks;
    }

    public Track[] getTracks() {
        return tracks;
    }

    @Path("tracks/{num}/")
    public Track getTrack(@PathParam("num") int num) {
        if (num >= tracks.length)
```

```
            throw new NotFoundException("Track, " + num + ",
            of CD, " + getTitle() + ", is not found");
        return tracks[num];
    }
}
```

Mapping the URI Path in the Bookstore Example

JSP pages are associated with resource classes. These JSP pages are resolved by converting the fully-qualified class name of the resource class into a path and appending the last path segment of the request URI path to that path. For example, when a GET is performed on the URI path "/", the path to the JSP page is `/com/sun/jersey/samples/bookstore/resources/Bookstore/`. For this example, since the last path segment is empty, `index.jsp` is appended to the path. The request then gets forwarded to the JSP page at that path. Similarly, when a GET is performed on the URI path `count`, the path to the JSP page is `/com/sun/jersey/samples/bookstore/resources/Bookstore/count.jsp`.

The JSP variable `it` is automatically set to the instance of `Bookstore` so that the `index.jsp`, or `count.jsp`, has access to the `Bookstore` instance as a Java bean.

If a resource class inherits from another resource class, it will automatically inherit the JSP pages from the super class.

A JSP page may also include JSP pages using the inheritance mechanism. For example, the `index.jsp` page associated with the `Book` resource class includes a `footer.jsp` page whose location is specified by the super class, `Item`.

The mapping of the URI path space is shown in the following table.

URI Path	Resource Class	HTTP method
/	Bookstore	GET
/count	Bookstore	GET
/time	Bookstore	GET
/items/{itemid}	Book, CD	GET
/items/{itemid}/tracks/{num}	Track	GET

Mapping the URI Paths and JSP Pages

The mapping of the URI paths and JSP pages is shown in the following table.

URI Path	JSP Page
/	/com/sun/jersey/samples/bookstore/resources/Bookstore/index.jsp
/count	/com/sun/jersey/samples/bookstore/resources/Bookstore/count.jsp
/time	/com/sun/jersey/samples/bookstore/resources/Bookstore/time.jsp
/items/{itemid}	/com/sun/jersey/samples/bookstore/resources/Book/index.jsp
/items/{itemid}	/com/sun/jersey/samples/bookstore/resources/CD/index.jsp
/items/{itemid}/tracks/{num}	/com/sun/jersey/samples/bookstore/resources/Track/index.jsp

▼ Building and Running the Bookstore Application from a Terminal Window

1. **Open a terminal prompt and navigate to** *glassfish.home*`/jersey/examples/bookstore`**.**
2. **Enter** `mvn glassfish:run` **and press Enter.**
   This will build, package, and deploy the project onto GlassFish using Maven.
3. **In a web browser navigate to:**
   `http://localhost:8080/Bookstore/`

▼ Building and Running the Bookstore Application from NetBeans IDE

1. **Select File→Open Project, and browse to the Jersey Bookstore sample application directory to open the project.**
2. **Right-click the project and select Properties. On the Actions tab, make sure that Use external Maven for build execution is checked. Also on the Actions tab, select Run Project under Actions, then change the Execute Goals field to `package glassfish:run`, and change the Set Properties field to `netbeans.deploy=true`. On the Run tab, make sure that the server is set to GlassFish v3 Prelude.**
3. **Right-click the project and select Run.**
   This will build, package, and deploy the project onto GlassFish using Maven.
4. **In a web browser navigate to:**
   `http://localhost:8080/Bookstore/`

When the sample is running, it looks like the following example:

```
URL: http://locahlhost:8080/Bookstore/
Czech Bookstore

Item List

    * Svejk
    * Krakatit
    * Ma Vlast 1

Others

count inventory
get the system time
regular resources
```

Other Jersey Examples

For more sample applications that demonstrate the Jersey technology, look in the *glassfish.home*/`jersey/samples/` directory. The following list provides a brief description of each sample application.

- *HelloWorld*: This is how everybody starts, using Grizzly as the process HTTP server.
- *HelloWorld-WebApp*: This is how everybody starts using a Web application. This application is described in this book. See section "The HelloWorld-WebApp Application" on page 36.
- *Bookstore* web application: Demonstrates how to use polymorphism with resources and views that are JSP pages. This application is described in this book. See section "The Bookstore Application" on page 48.
- *Entity-Provider*: Demonstrates pluggable entity providers.
- *Generate-WADL*: Demonstrates how to customize generation of the WADL file.
- *Jaxb*: Demonstrates the use of Java Architecture for XML Binding (JAXB)-based resources.
- *JMaki-backend* web application: Provides JavaScript Object Notation (JSON) to be consumed by jMaki widgets.
- *Json-From-Jaxb*: Demonstrates how to use JSON representation of JAXB-based resources.
- *Mandel*: A Mandelbrot service written in the Scala programming language using Scala's actors to scale-up the calculation.
- *Optimistic-Concurrency*: Demonstrates the application of optimistic concurrency to a web resource.

- *Simple-Atom-Server*: Simple Atom server that partially conforms to the Atom Publishing Protocol.
- *Simple-Console*: Demonstrates a simple service using Grizzly.
- *Simple-Servlet*: Demonstrates how to use a Servlet container.
- *Sparklines*: A Sparklines application inspired by Joe Gregorio's python application.
- *Spring-annotations*: An example leveraging Jersey's Spring-based annotation support.
- *Storage-Service*: Demonstrates a basic in-memory web storage service.

This list contains sample applications that are not installed with Jersey, but which also demonstrate the Jersey technology.

- *RESTful Web Services and Comet*: Demonstrates interacting remotely with a Comet web application. Learn how to build one with Dojo, Java EE technologies, and GlassFish.`http://developers.sun.com/appserver/reference/techart/cometslideshow.html`

Index

Numbers and Symbols

@Consumes, 21
@DELETE, 13, 18
@GET, 13, 18
@Path, 13, 16
@PathParam, 23
@POST, 13, 18
@Produces, 21
@PUT, 13, 18
@QueryParam, 23

A

annotations
  Jersey, 13
  overview, 13-14
applications
  creating, 29-34
  deploying, 29-34
  running, 29-34
  sample, 35

C

client-side, 27
cookie parameters, 23
creating, with NetBeans, 29-31
creating applications, 29-34

D

deploying, 31, 32-34
  with NetBeans, 31
  without NetBeans, 31-32
deploying applications, 29-34
deployment descriptor, example, 37

E

entity providers, 19

F

form parameters, 23

H

header parameters, 23
HTTP methods, 18

I

installing, 9
  on GlassFish, 9-10

J

JAX-RS, 5

JAX-RS *(Continued)*
APIs, 6
Jersey, 5
APIs, 6
in NetBeans, 10
installing, 9
on GlassFish, 9-10
other info sources, 6-7
samples
installing, 9, 10-11
running, 10
JSR-311, 5

M
matrix parameters, 23
Maven, running samples, 44
MessageBodyReader, 19
MessageBodyWriter, 19

O
overview, further topics, 27

P
parameters, extracting, 23
path, templates, 16
path parameters, 23

Q
query parameters, 23

R
request method designator, 13, 18
resource, configuring with runtime, 37
resource class, 13
resource method, 13
ResponseBuilder, 19
RESTful web services, 5
running applications, 29-34

S
sample applications, 35
Jersey distribution, 52
sample code, 52
samples
installing, 10-11
running, 10
in NetBeans, 37-38
with Maven, 38
security, 27

T
testing
with NetBeans, 31
without NetBeans, 31-32

U
URI path templates, 16

W
WADL, 44
web.xml, example, 37