RESTful Web Services Developer's Guide

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

Part No: 820–4867–05
May 2008

080502@19860

Contents

◆ ◆ ◆ 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?

Representational State Transfer (REST) is a software application architecture modeled after the way data is represented, accessed, and modified on the web. In the REST architecture, 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 architecture is fundamentally a client-server architecture, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourages REST applications to be simple, lightweight, and have high performance.

RESTful web services are web applications built upon the REST architecture. They:

- Expose resources (data and functionality) through web URIs.
- Use the four main HTTP methods to create, retrieve, update, and delete resources.

RESTful web services typically map the four main HTTP methods to the so-called CRUD actions: create, retrieve, update, and delete. The following table shows a mapping of HTTP methods to these CRUD actions.

TABLE 1–1 HTTP Methods and their Corresponding CRUD Action

HTTP Method	CRUD Action
GET	Retrieve a resource.

TABLE 1–1 HTTP Methods and their Corresponding CRUD Action *(Continued)*

HTTP Method	CRUD Action
POST	Create a resource.
PUT	Update a resource.
DELETE	Delete a resource.

How Does Jersey Fit In?

Jersey is the open source reference implementation for Java API for RESTful Web Services (JAX-RS, JSR 311). 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 Jersey 0.7 API's can be viewed at
`https://jsr311.dev.java.net/nonav/releases/0.7/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.

- *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

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 sections provides details for installing Jersey on GlassFish.

▼ Downloading and Installing Jersey on GlassFish

This task describes how to download and install Jersey onto the GlassFish container. This section assumes that GlassFish and Ant are already installed on your system, and that the `ANT_HOME` environment variable has been defined.

1. **Open your web browser and browse to** `http://jersey.dev.java.net`**.**
2. **Click Download.**
3. **Expand the** `stable` **folder.**
4. **Click on** `jersey-0.7–ea.zip` **to download this file.**
5. **Unzip the Jersey files and change to the** `jersey-0.7–ea` **directory.**
6. **To install Jersey on GlassFish, run the following command in a terminal window:** `ant -Dgf.home=`*gf.home* `-f jersey-on-glassfish.xml install`**, where** *gf.home* **is the fully-qualified path to your GlassFish installation directory. For example, this might be** `/home/yourname/glassfish`**.**

   This step copies the Jersey JAR files, the Jersey API documentation files, and the Jersey example applications to the *gf.home*`/jersey` directory.

Installing Jersey in NetBeans

The RESTful Web Services plugin comes bundled with NetBeans IDE 6.1. No additional steps are needed to configure and use the Jersey APIs with NetBeans.

◆ ◆ ◆ CHAPTER 3

Creating a RESTful Resource Class

A *resource class* is a Java class that uses JAX-RS annotations to implement a corresponding Web resource. Resource classes are POJOs (Plain Old Java Objects) that have at least one method annotated with `@Path` or a request method designator (`@GET`, `@PUT`, `@POST`, `@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 REST-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 server.

The Jersey .7 API's can be viewed at
`https://jsr311.dev.java.net/nonav/releases/0.7/index.html`.

Specifying the URI Path Template

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 URI path template:

```
http://example.com/users/{username}
```

A Jersey web service configured to respond to requests to this URI path template will respond to all of the following URIs:

```
http://example.com/users/jgatsby
http://example.com/users/ncarraway
http://example.com/users/dbuchanan
```

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`.

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.

Examples of URI Path Template Variables

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`: `East%20Egg`
- `question`: `why`

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

TABLE 3–1 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/`

TABLE 3–1 Examples of URI path templates *(Continued)*

URI Path Template	URI After Substitution
`http://example.com/maps/{location}`	`http://example.com/maps/East%20Egg`
`http://example.com/{name3}/home/`	`http://example.com//home/`

The `Path` Annotation

The `javax.ws.rs.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.

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>
```

Responding to HTTP Requests

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

The Request Method Designator Annotations

A *request method designator* is a runtime annotation. Within a resource class file, HTTP methods are mapped to Java programming language methods using the request method designator annotations. 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.

For example, the following resource defines a single method that responds to HTTP `GET` requests:

```
@Path("/hello")
public class Hello {
    ...
    @GET
    public String sayHello() {
        ...
    }
}
```

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 Method Parameters From URIs" on page 12. 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.

This tutorial will only discuss the four main HTTP methods: `GET`, `POST`, `PUT`, and `DELETE`.

Extracting Method Parameters From URIs

There are five types of parameters you can extract for use in your resource class: query parameters, URI path 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.

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.

Cookie parameters (indicated by decorating the parameter with `javax.ws.rs.CookieParam`) bind HTTP cookies to method parameters, class fields, or bean properties. Header parameters (indicated by decorating the parameter with `javax.ws.rs.HeaderParam`) bind HTTP headers to method parameters, class fields, or bean properties. Matrix parameters (indicated by

decorating the parameter with `javax.ws.rs.MatrixParam`) bind URI matrix parameters to method parameters, class fields, or bean properties. These parameters are beyond the scope of this tutorial.

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

- all primitive types except `char`
- all wrapper classes of primitive types except `Character`
- `String`
- any class with the static method `valueOf(String)`
- any class with a constructor that takes a single `String` as a parameter
- `List<T>`, where *T* matches the already listed criteria

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.

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 `@ConsumeMime` and `@Provider` annotations:

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

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

```
@ProduceMime("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();
    }
```

Customizing Requests and Responses

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

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

The `@ConsumeMime` Annotation

The `@ConsumeMime` annotation is used to specify which MIME types a resource class, method, or `MessageBodyReader` can accept. If `@ConsumeMime` is applied at the class level, all the response methods accept the specified MIME types by default. If `@ConsumeMime` is applied at the method level, it overrides any `@ConsumeMime` 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 `@ConsumeMime` is a comma separated list of acceptable MIME types. For example:

```
@ConsumeMime("text/plain,text/html")
```

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

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

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

The `doPost` method defaults to the MIME type of the `@ConsumeMime` annotation at the class level. The `doPost2` method overrides the class level `@ConsumeMime` 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 `@ProduceMime` Annotation

Similar to the `@ConsumeMime` annotation, the `@ProduceMime` annotation is used to specify the MIME types a resource or `MessageBodyWriter` can produce and send back to the client. If `@ProduceMime` 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 `@ProduceMime` 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 `@ProduceMime` is a comma separated list of MIME types. For example:

```
@ProduceMime("image/jpeg,image/png")
```

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

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

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

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

◆ ◆ ◆ CHAPTER 4

Deploying and Running Jersey Applications

Once you've used the Jersey annotations to add RESTful functionality to your web service, you will need to deploy and test the application. The following sections discuss deploying and testing your Jersey application from the command line and from NetBeans.

Several example applications are shipped with the Jersey download. These example applications are found in the *<JERSEY_INSTALL>*`/examples/` directory. There is a `README.HTML` file for each example that describes the example and describes how to deploy and test the example.

Deploying and Running a Jersey Application with GlassFish

The following sections describes how to deploy and run a Jersey application with GlassFish.

▼ Deploying and Testing a Jersey Application with GlassFish

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 "Downloading and Installing Jersey on GlassFish" on page 7.

1. **Bundle all of the JAR files in the Jersey distribution `lib` directory into the application's WAR file as compile time and run time dependencies of your project. This has already been done for the example projects.**

2. **Create the Ant tasks to compile and deploy the application. The Ant tasks for processing Jersey-annotated applications are the same as for non-Jersey-annotated applications. For examples of Ant tasks, look at the `build.xml` file in the *<JERSEY_INSTALL>*`/examples/`*<example>* directory.**

3 **Open a web browser, and enter the URL to which the application was deployed. Refer to the** `README.HTML` **file in each of the example applications for the correct URL for each example.**

Deploying and Running a Jersey Application with NetBeans

▼ Deploying and Running a Jersey Application from NetBeans

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 8.

1 **Right-click the project node. Select Properties, then select Run.**

2 **Type the URI path in the Relative URL field and click OK. You can find this information in the** `README.HTML` **file for the example applications.**

3 **Right-click the project node and choose Run. The first time GlassFish is started, you will be prompted for the admin password.**

The IDE starts the web container, builds the application, and displays the application in your browser. You have now successfully deployed a Jersey-enabled web service.

◆ ◆ ◆ CHAPTER 5

Using Jersey: Examples

This chapter discusses some examples that demonstrate how to create and use the Jersey annotations in your application. These examples are installed with Jersey into the *glassfish.home*/`jersey/examples` directory.

The Jersey Examples

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

- `HelloWorldWebApp` is a simple "Hello, world" example that responds to HTTP `GET` requests.
- `StorageService` demonstrates a simple, in-memory, web storage service.
- `Bookstore` demonstrates how to connect JSP pages to resources.

Configuring Your Environment

To run the examples, you must have:

- installed Jersey onto GlassFish v3 TP2
- optionally installed NetBeans IDE 6.1, which contains the RESTful Web Services plugin

The `HelloWorldWebApp` Application

This section discusses the `HelloWorldWebApp` application that ships with Jersey 0.7. The `HelloWorldWebApp` 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 responds to HTTP `GET` requests with a greeting that is sent back as plain text.

The following code is the contents of the `HelloWorldResource` class:

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

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

When you run the `HelloWorldWebApp` application, the annotations are processed during runtime.

The `web.xml` deployment descriptor for `SimpleServlet.war` 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.ws.rest.spi.container.servlet.ServletContainer</servlet-class>
</servlet>
<servlet-mapping>
     <servlet-name>Jersey Application</servlet-name>
         <url-pattern>/*</url-pattern>
</servlet-mapping>
```

The `com.sun.ws.rest.spi.container.servlet.ServletContainer` servlet is part of the JAX-RS API runtime, and works with the generated `HellWorldResource` 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 `HelloWorldWebApp` example application, the `@Path` is set to `/helloworld`, and the context root specified in `sun-web.xml` is `/HelloWorldWebApp`so our resource will respond to requests of the form:

```
http://<server>:<server port>/HelloWorldWebApp/helloworld
```

▼ Building and Running the `HelloWorldWebApp` Application in NetBeans IDE 6.1

1. **Select File→Open Project in NetBeans IDE 6.1.**
2. **Navigate to `<jersey.home>/examples`, select `HelloWorldWebApp`, and click OK.**

3. **Right click on the `HelloWorldWebApp` application in the Projects pane and select Run Project.**

   This will generate the helper classes and artifacts for your resource, compile the classes, package the files into a WAR file, deploy the WAR to your GlassFish v3 TP2 Application Server instance, and open a web browser to the following URL:

   ```
   http://<server>:<server port>/HelloWorldWebApp/helloworld
   ```

   Note – You will see some warning messages in your Output pane. These warnings can safely be ignored.

   You will see the following output in your web browser:

   ```
   Hello World
   ```

▼ Building and Running the `HelloWorldWebApp` Application with Ant

1. **Open a terminal prompt and navigate to *glassfish.home*`/jersey/examples/HelloWorldWebApp`.**

2. **Enter `ant` and press Enter.**

   This will build and package the `SimpleServlet.war` web application.

3. **Start GlassFish from the command line if it is not already started by entering this command: `$AS_HOME/bin/asadmin start-domain`, where $AS_HOME is a either a defined environment variable, or the path to GlassFish is used in place of the environment variable, or the GlassFish directory is in your path.**

4. **Enter `$AS_HOME/bin/asadmin deploy dist/SimpleServlet.war` and press Enter.**

   This will deploy `SimpleServlet.war` to GlassFish v3 TP2 Application Server.

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

   ```
   http://<server>:<server port>//HelloWorldWebApp/helloworld
   ```

   You will see the following output in your web browser:

   ```
   Hello World
   ```

The `StorageService` Application

The `StorageService` example application demonstrates a simple, in-memory, web storage service. 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. The key for the item is specified in the request URI. There are three web resources that are shown below.

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

The first resource, `com.sun.ws.rest.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 the contents of `ContainersResource.java`:

```
@Path("/containers")
@ProduceMime("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();
    }
}
```

The next resource, `com.sun.ws.rest.samples.storageservice.resources.ContainerResource`, enables 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 using the `getItemResource` method that is annotated with `@Path`. The following code is the contents of `ContainerResource.java`:

```
@ProduceMime("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(value="{item}", limited=false)
    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++) {
            int j = 0;
            for (; j < search.length; j++) {
                if (b[i + j] != search[j])
                    continue OUTER;
            }

            return true;
        }

        return false;
    }
}
```

The next resource, `com.sun.ws.rest.samples.storageservice.resources.ItemResource`, enables reading, creating, updating, and deleting of an item. The following code is the contents of `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 StorageService 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 `StorageService` Application

1 **Open a terminal prompt and navigate to** *glassfish.home*`/jersey/examples/StorageService`**.**

2 **Enter** `ant run-server` **and press Enter.**

This will build, package, and deploy the web storage service to the Lightweight HTTP Server that ships with Java SE 6.0. 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.

3. **To view the WADL description, open a web browser and navigate to:**

   ```
   http://127.0.0.1:9998/storage/application.wadl
   ```

4. **To get the containers, enter the following at the terminal prompt:**

   ```
   java -jar dist/StorageService.jar GET http://127.0.0.1:9998/storage/containers
   ```

   The response will indicate that no containers are present.

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

   ```
   java -jar dist/StorageService.jar PUT
   http://127.0.0.1:9998/storage/containers/quotes
   ```

   This step creates a container call `quotes`. If you run the GET command from the previous step again, this time it will return information about the `quotes` container.

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

```
echo "Something is rotten in the state of Denmark" | java -jar dist/StorageService.jar PUT http://127.0.0.1:9998/
echo "I could be bounded in a nutshell" | java -jar dist/StorageService.jar PUT http://127.0.0.1:9998/storage/con
echo "catch the conscience of the king" | java -jar dist/StorageService.jar PUT http://127.0.0.1:9998/storage/con
echo "Get thee to a nunnery" | java -jar dist/StorageService.jar PUT http://127.0.0.1:9998/storage/containers/quo
```

   If you run the GET command again with `/quotes` at the end, it will show that the `quotes` container has 4 items associated with keys 1, 2, 3, and 4.

7. **You can search the contents of the `quotes` container. For example, the following command would search for the `String` `king`, which return an XML document containing item 3.**

```
java -jar dist/StorageService.jar GET http://127.0.0.1:9998/storage/containers/quotes?search=king
```

8. **To get the contents of item 3, use the following command:**

   ```
   java -jar dist/StorageService.jar GET 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`.

9. **More examples demonstrating the web storage container's capabilities are available in the *glassfish.home*`/jersey/examples/StorageService/README.html` file.**

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 example consists of four web resources, described below.

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`.

```
@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;
    }

    public long getSystemTime() {
        return System.currentTimeMillis();
    }

    public Map<String, Item> getItems() {
        return items;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Both the `Book` and the `CD` resource inherit from the `Item` class. This allows the resources to be managed polymorphically. The contents of `Item.java` are shown below:

```
package com.sun.ws.rest.samples.bookstore.resources;

public class Item {

    private String title;
    private String author;


    public Item(final String title, final String author) {
        this.title = title;
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }
}
```

The `Book` resource includes a title and an author.

```
package com.sun.ws.rest.samples.bookstore.resources;

public class Book extends Item {

    public Book(final String title, final String author) {
        super(title, author);
    }
}
```

The `CD` resource contains a title, an author, and a list of tracks. The 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];
    }
}
```

The `Track` resource includes a name and a length of the track.

```
public class Track {

    private String name;
    private int length;

    /** Creates a new instance of Track */
    public Track(String name, int length) {
        this.name = name;
        this.length = length;
    }

    public String getName() {
        return name;
    }

    public int getLength() {
        return length;
    }
}
```

▼ Building and Running the `Bookstore` Application

1. **Open a terminal prompt and navigate to** *glassfish.home*`/jersey/examples/Bookstore`**.**
2. **Enter** `ant run-on-glassfish` **and press Enter.**
   This will build, package, and deploy the `Bookstore.war` web application.
3. **In a web browser navigate to:**
   `http://<server>:<server port>/Bookstore/`
4. **For further description of how this application works, read** *glassfish.home*`/jersey/examples/Bookstore/README.html`**.**

Other Jersey Examples

For more examples that demonstrate the Jersey technology, look in the *glassfish.home*/jersey/examples/ directory. The following list provides a brief description of each example.

- *Bookmark* demonstrates how to use JPA in the backend.
- *Bookstore* demonstrates how to connect JSP pages to resources.
- *EntityProvider* demonstrates how to add support for custom Java types as resource class method parameters.
- *HelloWorld* demonstrates how to develop a RESTful web service with the JVM-embedded HTTP container.
- *HelloWorldWebApp* demonstrates how to develop a RESTful web service with a Servlet 2.5 container.
- *jMakiBackEnd* demonstrates providing jMaki widget JSON data models as Jersey resources.
- *JsonFromJaxb* demonstrates how to use JSON representation of JAXB—based resources.
- *Mandel* demonstrates a service, implemented in Scala, that calculates the Mandelbrot set over a specified area (of the complex plane) and returns an image of that area that represents the set.
- *OptimisticConcurrency* demonstrates how to apply optimistic concurrency to a web resource.
- *SimpleAtomServer* demonstrates a simple Atom server that partially conforms to the *Atom Publishing Format and Protocol*.
- *SimpleConsole* demonstrates how to develop RESTful web services with the Lightweight HTTP Server included in Java SE 6.
- *SimpleJAXWSEndpoint* demonstrates how to develop RESTful web services with a JAX-WS Endpoint.
- *SimpleServlet* demonstrates how to develop RESTful web services with a Servlet 2.5 container.
- *StorageService* demonstrates a simple, in-memory, web storage service.

Index