

```
11     list.add(places[i]);
12 }
13 root.put("places", list);
14 res.getWriter().write(root.toString());
15 }
```

**Listing 3.4** Generating responses (JSON)

When JSON is requested, the servlet will produce something like this as the response (resulting screenshot shown in the bottom right corner of Fig. 3.6):

In reality, it would still be nice to have some support (a framework). Having a framework can help reduce boilerplate-type coding. In Java, the libraries that help RESTful service development are specified in the JAX-RS (<http://jcp.org/en/jsr/detail?id=311>) standard specification. There are many implementations of JAX-RS. Through the lab exercise, we will see an example of JAX-RS support using Apache CXF.

## Practical Exercise III

### 3.10 Lab Exercise 03: A Simple REST Service with Apache CXF (JAX-RS)

In this lab exercise, we are going to build a simple RESTful service. A RESTful Web service is designed and implemented using the HTTP methods and the principles of the REST architectural style. A typical API document for a REST-based service will provide the base URI for the service, the MIME types it uses to communicate (e.g., XML or JSON) and the set of operations (POST, GET, PUT, DELETE) supported.

Java community defines standard REST support via JAX-RS (the Java API for RESTful Web Services) in JSR 311. There are a few implementations of this standard available now (e.g., Apache CXF, RESTEasy by JBoss, Jersey by Sun). We will use the Apache CXF implementation.

#### *Exercise Files*

The exercise files involved in the lab can be downloaded from the Github site:

- <https://github.com/SOA-Book>

Follow the repository exercises to lab03.

### 3.10.1 Activity 1: Start with Hello World

We will start by getting a template from Apache CXF Maven Archetype. Of course, this is not the only way to get yourself started with REST and Apache CXF, but since we tried their template for SOAP (JAX-WS), let us go with their template for REST as well.

1. In a terminal window, navigate to your workspace.
2. Type in the following (mind the ‘:’ at the end):

```
$ mvn archetype:generate -Dfilter=org.apache.cxf.archetype:
```

3. You will be prompted to ‘Choose archetype:’, enter number 1 (i.e., `cxf-jaxrs-service` (Simple CXF JAX-RS webapp service using Spring configuration))
4. At the next prompt, ‘Choose a version:’, enter number 51.
5. Then, at ‘Define value for property groupId:’, enter the groupId value as `au.edu.unsw.soacourse`.
6. Then, at ‘Define value for property artifactId:’, enter the artifactId value as `HelloWorldCxfRest`.
7. Then, at ‘Define value for property version:’, The default value is `1.0-SNAPSHOT` (case sensitive). Leave the value as is, and hit Enter.
8. Next, at ‘Define value for property package:’, enter the package name as `au.edu.unsw.soacourse.hello`.
9. Now Maven will generate the template code. You should see a ‘Build Success’ message. You should also see a new directory called ‘HelloWorldCxfRest’ under your workspace.
10. The `pom.xml` file given from this template includes some plug-ins that we do not require. Replace it with the `pom.xml` provided for this exercise. We have removed unnecessary plug-ins and set `maven-compiler-plugin` to use JDK 1.7.

Let us import this into the Eclipse IDE as an *Existing Maven Project* and examine what has been generated.

1. Open Eclipse, and navigate to `File → Import → Maven → Existing Maven Project`. Navigate to `HelloWorldCxfRest` and complete the import process.
2. After importing, you should have a project structure similar to that in Fig. 3.7:
3. There are a few things to note. First, open `web.xml`. You will see a declaration of `CXFServlet`. The implementation of the servlet is provided by Apache CXF, as the name `org.apache.cxf.transport.servlet.CXFServlet` suggests. The servlet is mapped to a URL pattern `/*`. That is, all requests starting with `http://localhost:8080/HelloWorldCxfRest` are directed to this servlet.

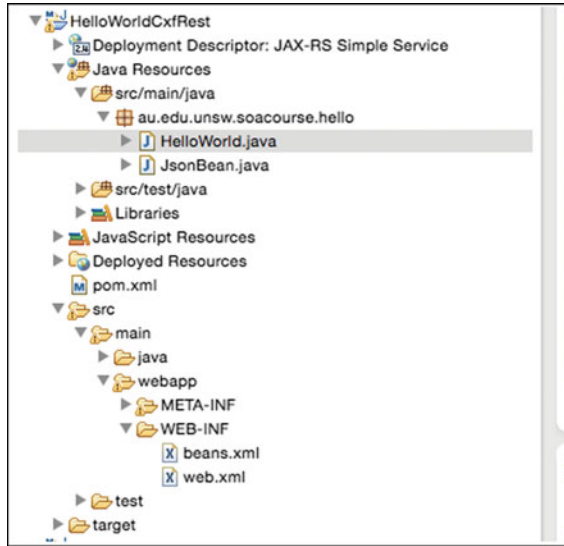


Fig. 3.7 JAX-RS HelloWorld project structure

4. The rest of the configuration details come from WEB-INF/beans.xml. Open beans.xml. Pay attention to the `jaxrs:server` declaration. This is where you will declare your REST implementation class. For example, in this current template, class `au.edu.unsw.soacourse.hello.HelloWorld` provides the implementation code. The `jaxrs:providers` part is for registering JSON support class `org.codehaus.jackson.jaxrs.JacksonJsonProvider` so you can consume/produce JSON in your service.
5. Now, open HelloWorld.java.
  - class HelloWorld is annotated with `@Path("/hello")`. `@Path` can be applied to resource classes or methods. When marked on a class, it means the class will handle the requests starting with `/hello/{...}`. With the current mapping in web.xml, the URL path will be:

```
http://localhost:8080/HelloWorldCxfRest/hello
```

- The `ping()` method is annotated with the following. Let us go through them one by one.
  - `@Path(/echo/{input})` indicates that this method will handle the request path matching:  

```
http://localhost:8080/HelloWorldCxfRest/hello/echo/{input}
```
  - `@GET` indicates that the method will handle the GET request method on the said request path.

- `@Produces("text/plain")` indicates that the response will contain plain text. `@Produces(MediaType.TEXT_PLAIN)` has the same meaning.
- `@PathParam("input")` annotation applies to the `String` input parameter of `ping()`. It will map the text value of the path parameter appearing after `/echo` to `String` input.
- The method itself simply returns the input as a `String`.
- The `modifyJson()` method is annotated with:
  - `@Path("/jsonBean")`, indicates that this method will handle the request path matching:  
`http://localhost:8080/HelloWorldCxfRest/hello/jsonBean`
  - `@POST`, indicates that the method will handle the POST request method on the said request path
  - `@Produces("application/json")`, indicates that the response will contain JSON.
  - `@Consumes("application/json")`, which applies to the input parameter `JsonBean` input. It will map the body content of the POST request (which will contain JSON) to `JsonBean` input.
  - The method itself simply sets the value of `Val2` to the value of `Val1` and returns the object as response.

The template is actually functional as is. Let's compile and deploy to Tomcat and see how it behaves.

1. Run Maven Install – and add the project to the Tomcat server to run.
2. Open a browser. Type in:

```
http://localhost:8080/HelloWorldCxfRest/hello/echo/HelloWorld
```

You should see a plain text 'HelloWorld' returned.

### 3.10.2 Activity 2: Testing the HelloWorld Service

To test the REST service properly, we need a REST service client tool. There are many options, but one we settle with in this lab is Chrome POSTMAN. You can download it from the Chrome Web Store at the following address:

<https://chrome.google.com/webstore/category/apps>

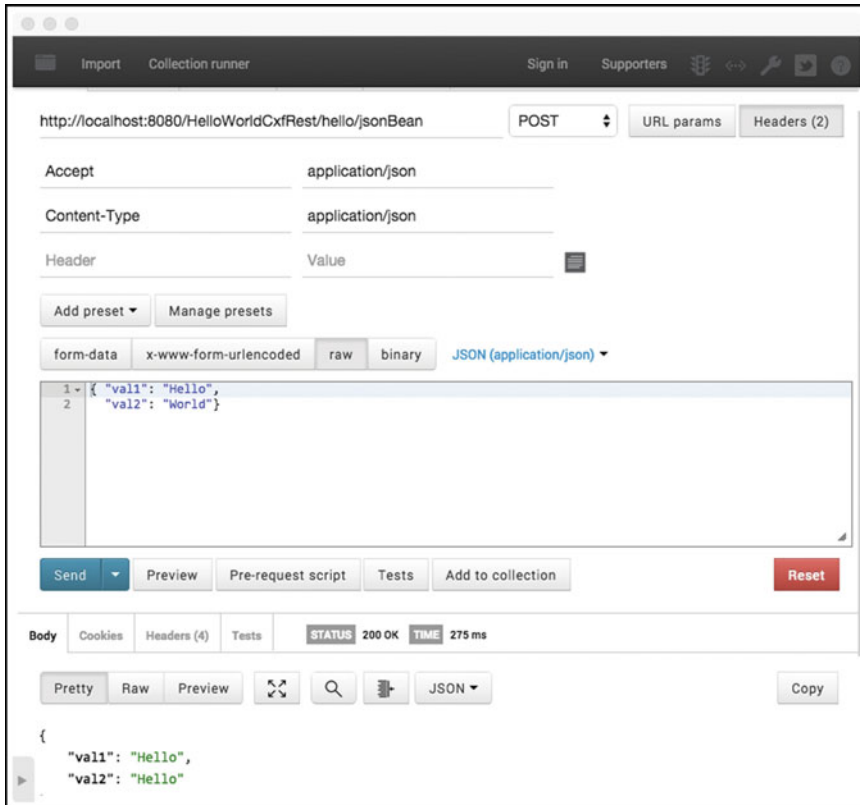
*NOTE: There are a plug-in version and an app version. We think the app version is easier to use. So download and install the app version of POSTMAN.*

Once you have installed POSTMAN, open the app, then try the following:

1. type in the URL you want to test,
2. choose the request method you want to perform on the URL,

3. choose the data format you want to send and type in necessary text,
4. click Send.

For example, to test the POST implementation in HelloWorld.java, you would do something similar to what is shown in Fig. 3.8.



**Fig. 3.8** Chrome POSTMAN

Play around with this tool to become familiar with what it does. It is a handy tool to learn for any REST service development task.

### 3.10.3 Activity 3: *CRUD Operations on Resources*

Now that we have tried the basic skeleton of Apache CXF on REST services, let us try and build a simple REST service from scratch.

Resources are at the centre of a RESTful Web service. The first thing in designing a RESTful Web service will be deciding what the resources you want to expose are – and coding some basic operations for them. As we have seen briefly in the HelloWorld template, In JAX-RS, resources are implemented by a POJO with an `@Path` annotation that associates an identifier (URI) to the resource.

A resource can also have sub-resources. In this case, such a resource is a resource collection while the sub-resources are member resources (e.g., class and students). For this exercise, let us use good old books as resources and develop a basic set of operations to create/read/update/delete them.

1. Use the HelloWorld project as a template and create a new Maven project out of that.
2. Edit `pom.xml` to reflect the following properties:
  - `groupId: au.edu.unsw.soacourse`
  - `artifactId: RestfulBookService`
  - `package: au.edu.unsw.soacourse.books`
3. Do Maven → Update Project to reflect the changes correctly.
4. Add a Data Model and a Data Access Model for the Resources. For this simple exercise, we are going to write a couple of Java classes for the data model and the data access model (not really useful in real applications, but will do for this lab exercise.)
  - **[A Book class (plain Java class)]** Using the `Book.java` file provided for the exercise, create the `Book` class with the given package detail. Note that this class also defines automatic mapping to XML (i.e., the class is automatically mapped to XML and back for serialisation). This is done via Java JAXB and the `@XmlElement` annotation.<sup>2</sup>
  - **[BookDao class]** Using the `BooksDao.java` file provided for the exercise, create the `BookDao` class with the given package details. Note that this class is just a `HashMap`-based in-memory data store, so to speak, which has no significance outside this lab guide. In a more realistic development, you will have a proper Data Access Object (DAO) which interacts with a real data store through queries. You may later turn the design into a proper DAO with full database access when needed.
5. Implement a REST-based service based on this model. We are going to write a class that exposes book resources in different ways (i.e., GET, PUT, DELETE, etc.).  
**[BooksResource Class]** Using the `BooksResource.java` file provided for the exercise, create the `BooksResource` class with the given package detail. Note that this class exposes the main REST interface for the book collection. It exposes GET (get the list of books, get the size of the list) and POST (for creating a book

---

<sup>2</sup><https://docs.oracle.com/javase/tutorial/jaxb/intro/>.

under the collection). Also, it has a path param mapping for handling single book resources.

6. Configure the service and compile, then deploy. Open `WEB-INF/beans.xml`. You should configure the `jaxrs : server` declaration so that the implementation bean correctly points to `au.edu.unsw.soacourse.books.BooksResource`. Now run ‘Maven install’ and deploy the service to Tomcat.
7. Using POSTMAN, try out each method in the service and see its behaviour (especially the input/output).

### 3.10.3.1 Issues with `BooksResource.java`

Although functional, there are a few issues with the way we have implemented the REST operations in our previous activity. Let us go through the issues here so that you can address them. We would like to implement the operations so that their behaviour reflects the correct interpretation of RESTful HTTP operations.

In `BooksResource.java`, there are a few things to improve. Especially, we should consider what each operation’s input and output should be.

Take the POST operation. Two things to note here:

- The input values are passed in as form params. In doing so, the ‘id’ of the resource is expected from the user as well. This is not correct behaviour of a POST operation, as POST should generate the id and let the user know what it is in response to.
- The POST operation returns nothing. We should generate a response that will contain links to the new resource (if created), and also to other resources that the client might be interested in.

Take the DELETE operation. Instead of returning a normal response with a status code (e.g., `NOT_FOUND`), it throws an error when the given id is not found in the data store. You always want to return something sensible to the user.

Take the PUT operation; the idea implemented in `putAndGetResponse()` seems to be that if the given key exists in the data store, it will update the value associated with the key. If it doesn’t, it creates it. What are the status codes used here? For example, what does `Response.created().build()` return? Are they appropriate given that the implementation logic here deals with two possible scenarios?

Think about how you might improve these operations and make them truly RESTful operations. We leave this as an exercise for you.

### 3.10.4 Activity 4: Developing a Client for RESTful Services

There are many choices for building REST clients. For this lab, we will show what Apache CXF provides you with in terms of accessing REST services from a client. There are two sources of documents you can refer to:

- Apache CXF Client API documentation,<sup>3</sup>
- Apache CXF WebClient class documentation.<sup>4</sup>

**[BookServiceClient Class]** The `BookServiceClient.java` file provided for this exercise contains most of the features of the client API. Using the file provided, create the `BookServiceClient` class with the given package details in the current working project. Note that this class (which has its own `main()`) is a demonstrator of the `WebClient` class in Apache CXF. Look through the code. Compile and run the code. The results should be in the Console view.

---

<sup>3</sup><http://cxf.apache.org/docs/jax-rs-client-api.html#JAX-RSClientAPI-CXFWebClientAPI>.

<sup>4</sup><https://cxf.apache.org/javadoc/latest/org/apache/cxf/jaxrs/client/WebClient.html>.

Web Service Implementation and Composition  
Techniques

Paik, H.H.-Y.; Lemos, A.L.; Barukh, M.C.; Benatallah, B.;  
Natarajan, A.

2017, XIII, 256 p. 102 illus., 80 illus. in color., Hardcover  
ISBN: 978-3-319-55540-9