Developing REST Web Services Tutorial

Table of Contents

1. Introduction

This document will outline the process of developing a REST web service, deploying it to the internal MyEclipse Tomcat server and testing it with the REST Web Services Explorer. The REST features in MyEclipse are based on Jersey, which is the reference implementation for JAX-RS, the Java API for RESTful Web Services. We will be creating a simple web service which we will use to maintain a list of customers.

MyEclipse also supports developing SOAP web services using JAX-WS; for folks needing to develop and deploy WebSphere JAX-RPC or

WebSphere JAX-WS web services, please take a look at MyEclipse Blue Edition.

Additional resources covering web service creation using JAX-WS and JAX-RPC are included in the Resources section of this document.

top

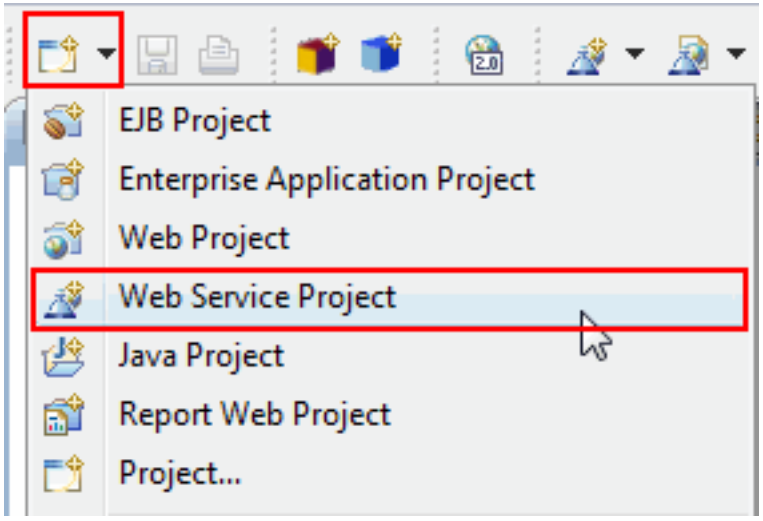
2. System Requirements

This tutorial was created with MyEclipse. However, if you notice portions of this tutorial looking different than the screens you are seeing, please let us know and we will make sure to resolve any inconsistencies.

top

3. Creating the REST Web Service Project

To get started we will create a simple Web Service Project by selecting **Web Service Project** from the new toolbar menu:



Alternatively, invoke the wizard using **File > New > Other > MyEclipse > Java Enterprise Projects > Web Service Project**.

Name the project *restdemo* and select **REST (JAX-RS)** from the list of frameworks.

New Web Service Project

New Web Service Project

Web service project creation details

Web Project Details

Project Name: restdemo

Location: ☑ Use default location

Directory: D:\gtech\runtime-myeclipse-7.0\restdemo Browse...

Source folder: src

Web root folder: WebRoot

Context root URL: /restdemo

Web Service & J2EE Details

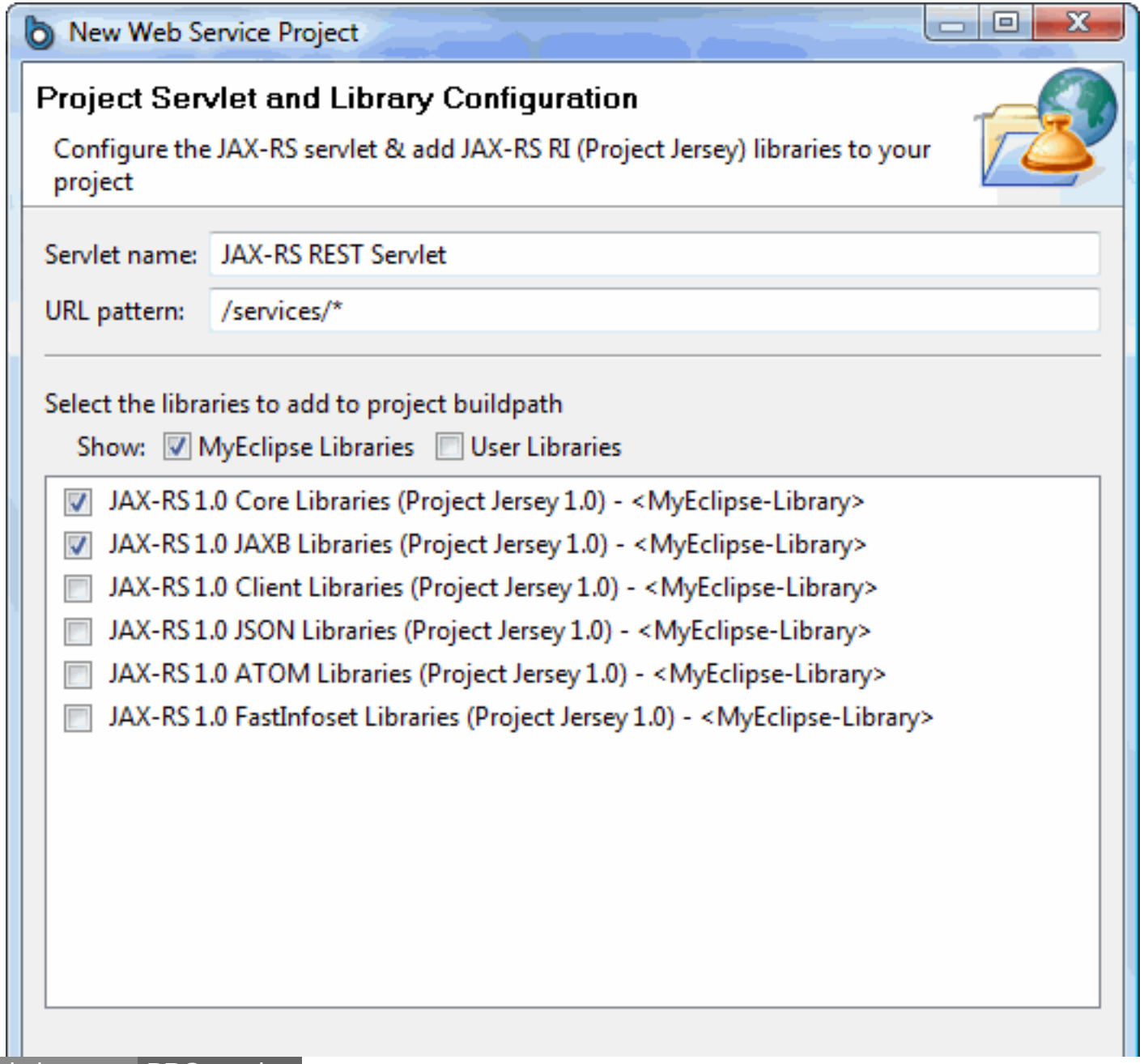
Framework:
- ○ JAX-WS
- ○ JAX-WS (WebSphere)
- ○ JAX-RPC (WebSphere)
- ◉ REST (JAX-RS)
- ○ XFire (deprecated)

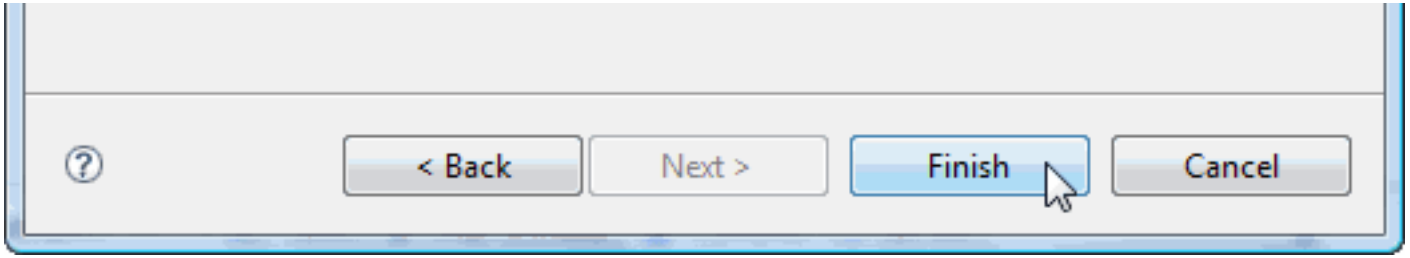
J2EE specification: ◉ Java EE 5.0 ○ J2EE 1.4 ○ J2EE 1.3

Target server: <unspecified>

< Back Next > Finish Cancel

Click **Next** to move to page 2 of the wizard. On this page you can specify the path at which the services will be available, the name of the corresponding JAX-RS servlet and libraries which you wish to add to your project. For this project the defaults are fine, so click **Finish** to create the project.






```
<servlet>
  <display-name>JAX-RS REST Servlet</display-name>
  <servlet-name>JAX-RS REST Servlet</servlet-name>
  <servlet-class>
      com.sun.jersey.spi.container.servlet.ServletContainer
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>JAX-RS REST Servlet</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

JAX-RS servlet generated in web.xml

Note: Instead of creating a new project, you may also add REST capabilities to any existing Java EE 5 Web project. From the project's context menu, select **MyEclipse > Add REST Capabilities...**

top

4. Creating the REST Web Service

4.1 Creating the Customer entity

To start, create a simple Customer class with id, name and address fields; this class represents the Customer entity we will be managing with our web service. Use the **File > New > Class** wizard, put *Customer* in the **Name** field, *com.myeclipseide.ws* in the **Package** field and **Finish** the wizard. Replace the contents of the generated class with the following code:

```java
package com.myeclipseide.ws;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Customer {

  private int id;
  private String name;
  private String address;

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

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

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }

}
```

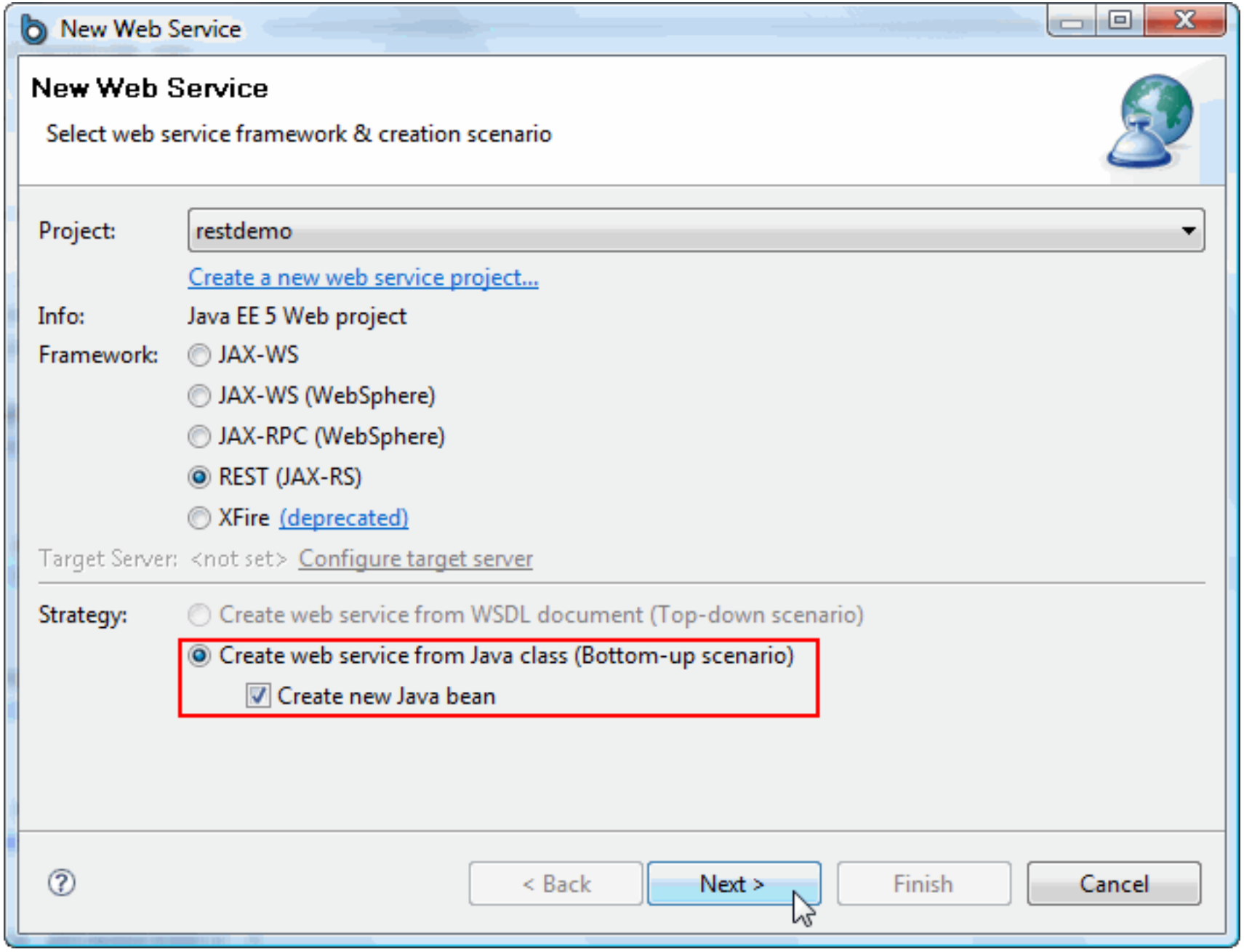
In this tutorial, we will be using XML as the serialization format, i.e. we will send and receive Customer entities from the web service using XML.

The @XMLRootElement annotation on the Customer class is a JAXB annotation which allows JAXB to convert this entity from Java to XML and back. It is possible to annotate the fields and methods within the class to customize the serialization, but for our tutorial the JAXB defaults are fine.

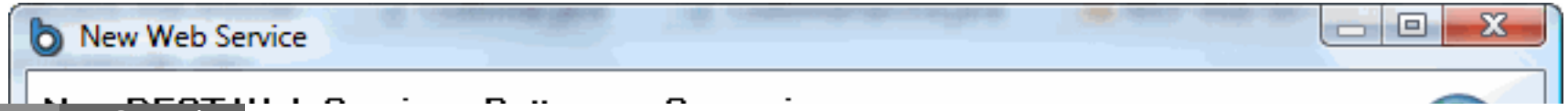
4.2 Creating the CustomersResource class, the core of our web service

1. Select the *restdemo* project and invoke the new web service wizard from the toolbar:

Make sure the Project combo displays the *restdemo* project and the **REST (JAX-RS)** framework is selected. Select **Create new Java bean** and click **Next**.



2. Fill out this page as shown in the following screenshot:



New REST Web Service - Bottom-up Scenario

Class contains no sub-resource methods

Project: restdemo

URL path: customers

Lifecycle: Per-request (JAX-RS default) Singleton

Consumes: <unspecified>

Produces: application/xml

Java source folder: src New...

Java package: com.myeclipseide.ws Browse... New...

Java class: CustomersResource

Java methods:

Add

Edit

Remove

Up

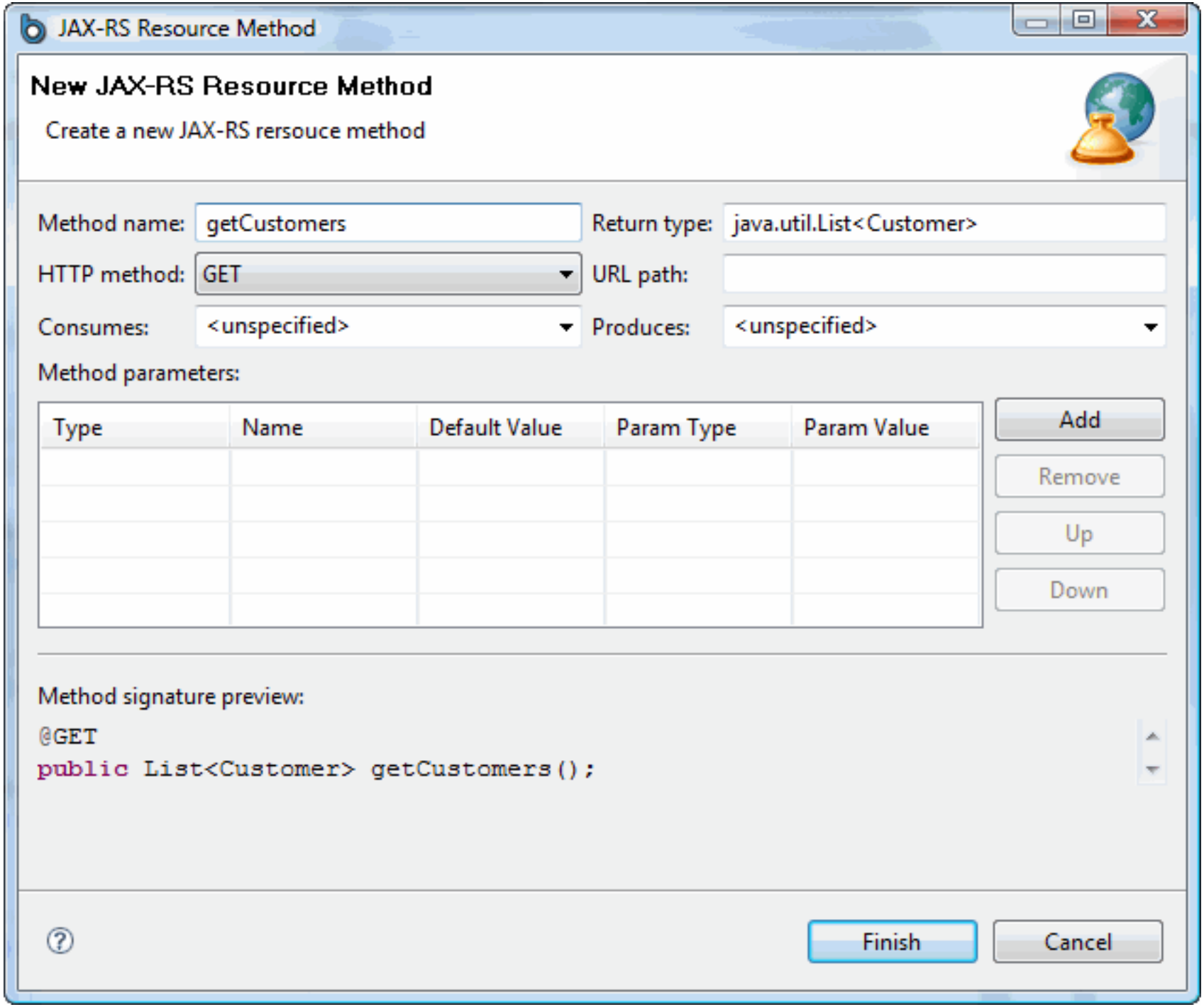
Down

Click Add

< Back Next > Finish Cancel

- The **URL path** field indicates the path at which this resource can be reached, for this tutorial, we will use *customers* as this resource manages our customer list. The resource will thus be hosted at "/customers".
- **Singleton** lifecycle ensures that only one instance of this class will created by Jersey per web-application.
- The **Consumes** and **Produces** combos can be used to specify the default mime type(s) of data which this resource can accept and generate. These values can be overridden by individual methods in the class. As mentioned above, we will be serializing to XML, so we use the **application/xml** mime type.

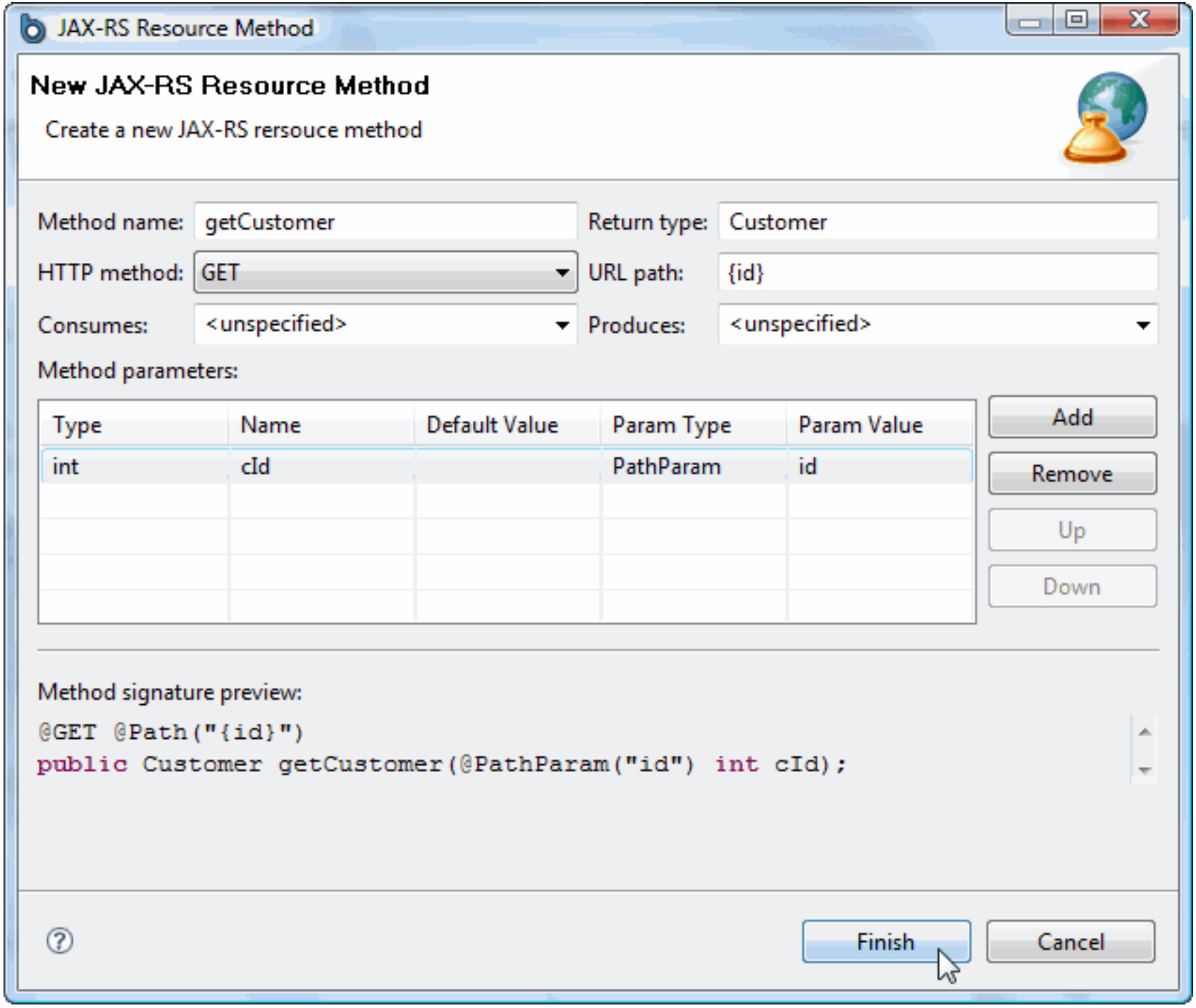
3. Click the **Add** button in the above dialog to add the method that will fetch a list of customers. Fill out the wizard that pops up like so and click **Finish**.



- The **HTTP method** combo can be used to specify what type of HTTP request this method will respond to; in this case, we

wish to respond to an HTTP GET request.
    - The **Method Signature preview** will be updated as you make changes to the page, giving you an idea of what your method will look like once generated.

4. Click the **Add** button again to add a method that will return details of a specific customer.



- The **URL path** field specifies the path at which this method can be reached, relative to the containing resource.

In this case we specify *{id}*, which means this resource method can be reached at /customers/{id}. The curly braces denote a URI variable. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI.
    - Click **Add** to add a method parameter which can be directly edited in the table. Since we need the value of the *id* variable, we use the **PathParam** annotation to map it to the *cId* parameter.

5. Finally, add a method which allows us to add a new customer to our list.

JAX-RS Resource Method

New JAX-RS Resource Method

Create a new JAX-RS rersouce method

Method name: addCustomer

Return type: java.lang.String

HTTP method: POST

URL path: add

Consumes: application/xml

Produces: text/html

Method parameters:

Type	Name	Default Value	Param Type	Param Value
Customer	customer		Entity	

Add

Remove

Up

Down

Method signature preview:

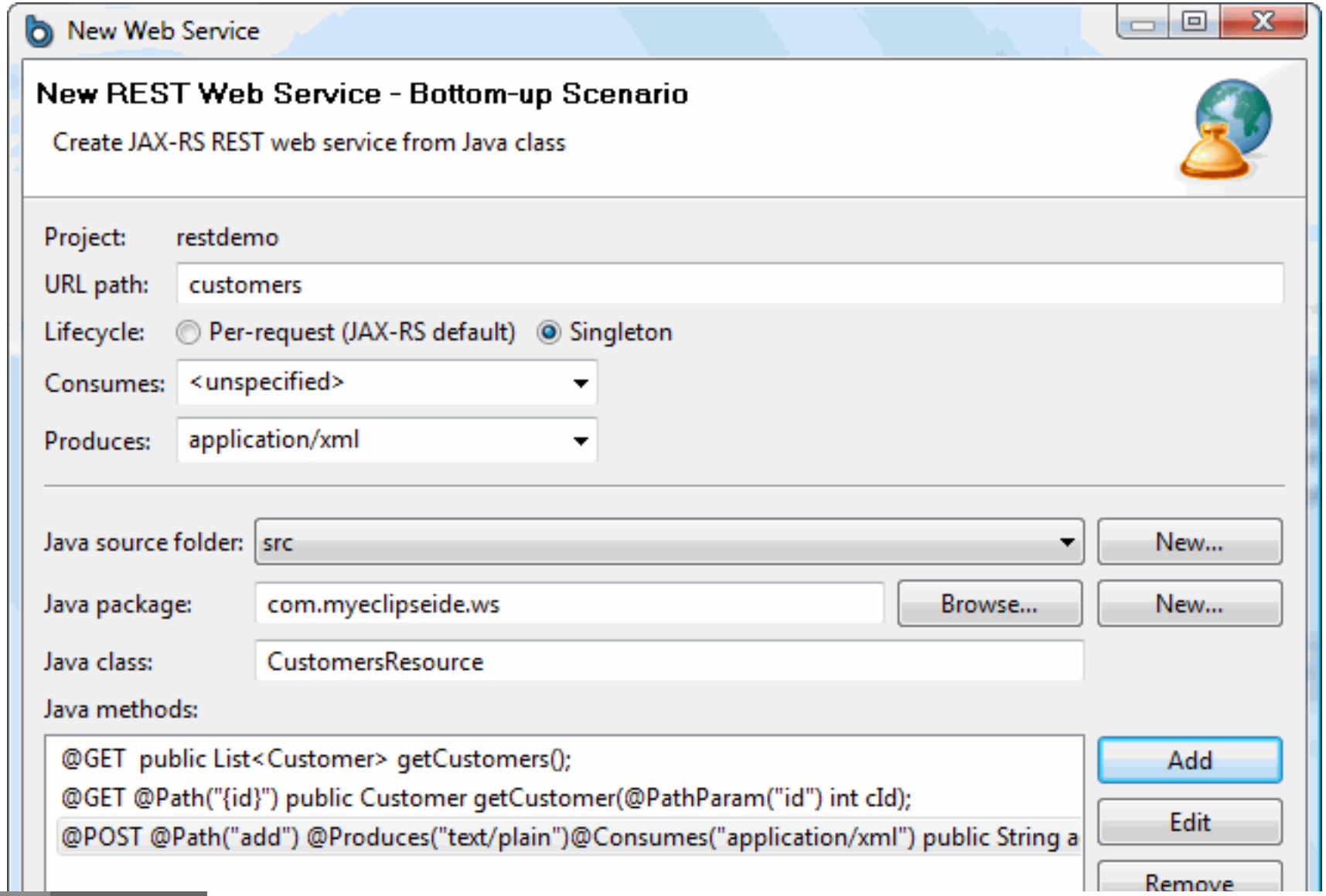
```
@POST @Path("add")
@Produces("text/html")@Consumes("application/xml")
public String addCustomer(Customer customer);
```

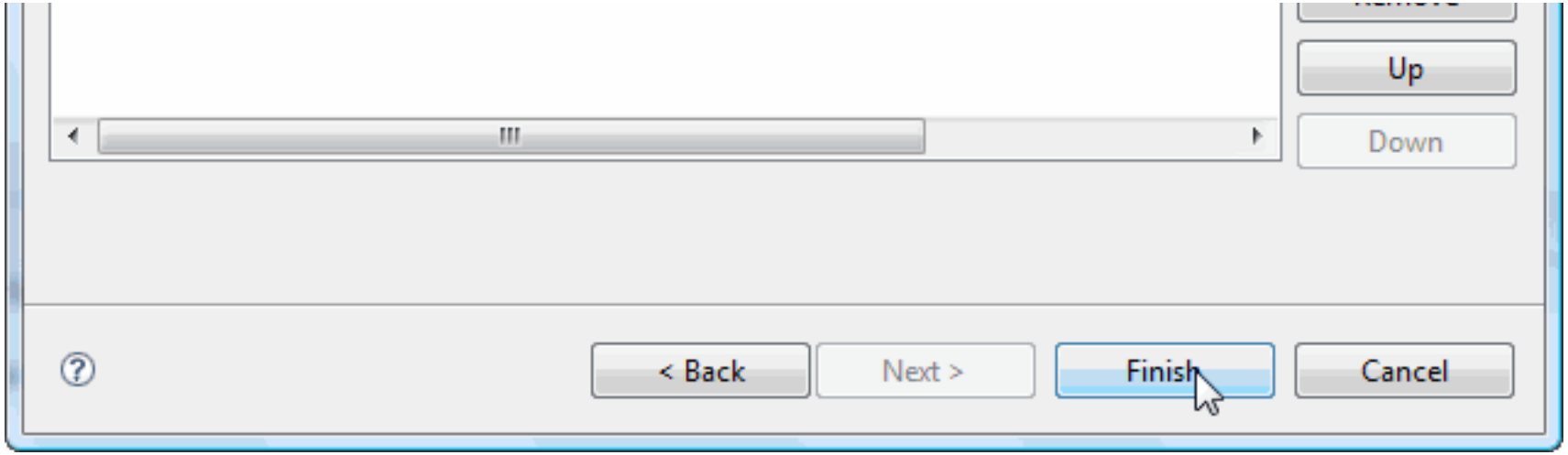
Finish

Cancel

- In this case, we're responding to a POST request and expect application/xml input which would be deserialized into the *customer* parameter.
- The *customer* parameter is an **Entity** parameter (unannotated) and is mapped directly from the message body of the incoming request.
- We also override the default application/xml output specified by the *CustomersResource* class and specify **text/html** instead.

6. After adding those 3 methods, your wizard should now look like this:





Click **Finish** to generate the *CustomersResource* class, you will see a class with stubbed out resource methods as shown below:

```
package com.myeclipseide.ws;

import java.util.List;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

import com.sun.jersey.spi.resource.Singleton;

@Produces("application/xml")
@Path("customers")
@Singleton
public class CustomersResource {

  @GET
  public List<Customer> getCustomers() {
    throw new UnsupportedOperationException("Not yet implemented.");
  }
```

```
  @GET
  @Path("{id}")
  public Customer getCustomer(@PathParam("id") int cId) {
    throw new UnsupportedOperationException("Not yet implemented.");
  }

  @POST
  @Path("add")
  @Produces("text/plain")
  @Consumes("application/xml")
  public String addCustomer(Customer customer) {
    throw new UnsupportedOperationException("Not yet implemented.");
  }
}
```

7. We must now provide implementations for the methods created by the above wizard. In a real application, at this point we would probably wire in a database using JPA or Hibernate to help manage our customer list, but a simple in-memory map is sufficient for this tutorial.

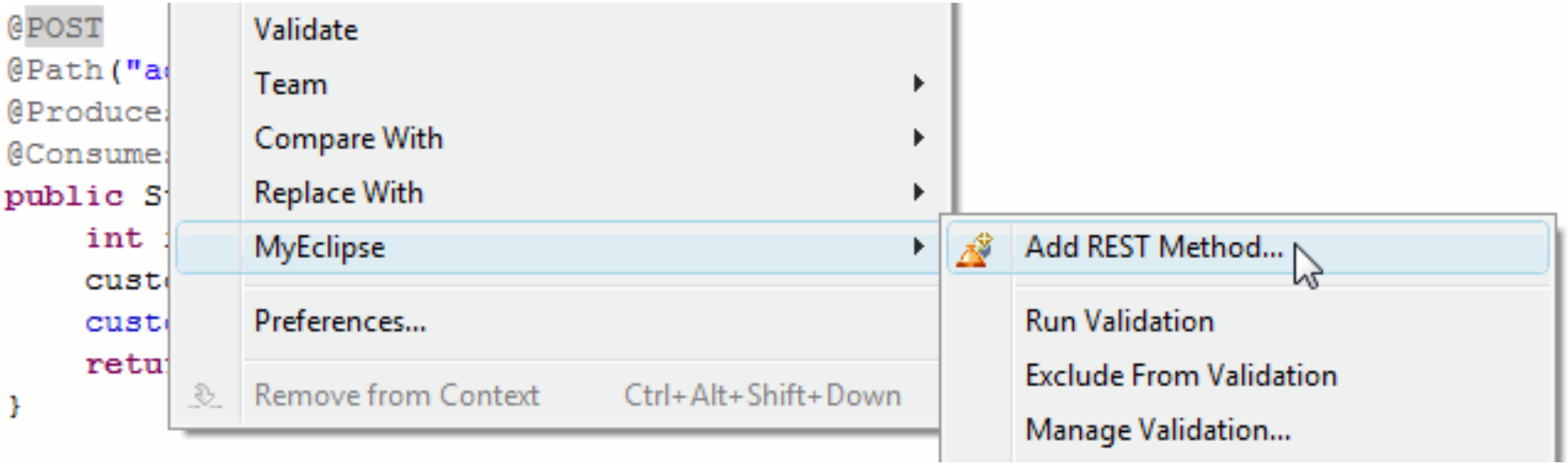
   Our implementation is simple; when a customer is received by our service, we give the entity a counter based id and add it to our map. Retrieving a customer from this map by id and providing a list of customers is straightforward as you can see below. You may copy this implementation into your class; observe that the class and method signatures have not changed, all we're doing is fleshing out the generated stubs with an implementation of our service. We also add a single customer to the list for demonstration purposes.

```
package com.myeclipseide.ws;

import java.util.ArrayList;
import java.util.List;
import java.util.TreeMap;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

import com.sun.jersey.spi.resource.Singleton;
```

```java
@Produces("application/xml")
@Path("customers")
@Singleton
public class CustomersResource {

  private TreeMap<Integer, Customer> customerMap = new TreeMap<Integer, Customer>();

  public CustomersResource() {
    // hardcode a single customer into the database for demonstration
    // purposes
    Customer customer = new Customer();
    customer.setName("Harold Abernathy");
    customer.setAddress("Sheffield, UK");
    addCustomer(customer);
  }

  @GET
  public List<Customer> getCustomers() {
    List<Customer> customers = new ArrayList<Customer>();
    customers.addAll(customerMap.values());
    return customers;
  }

  @GET
  @Path("{id}")
  public Customer getCustomer(@PathParam("id") int cId) {
    return customerMap.get(cId);
  }

  @POST
  @Path("add")
  @Produces("text/plain")
  @Consumes("application/xml")
  public String addCustomer(Customer customer) {
    int id = customerMap.size();
    customer.setId(id);
    customerMap.put(id, customer);
    return "Customer " + customer.getName() + " added with Id " + id;
  }
```

```
}
```

Note: You can invoke the the JAX-RS Method wizard directly for any class in a REST web service project by using the **Add REST Method...** context menu action. Right-click in the Java editor to bring up the context menu and select **Add REST Method...** from the **MyEclipse** submenu.

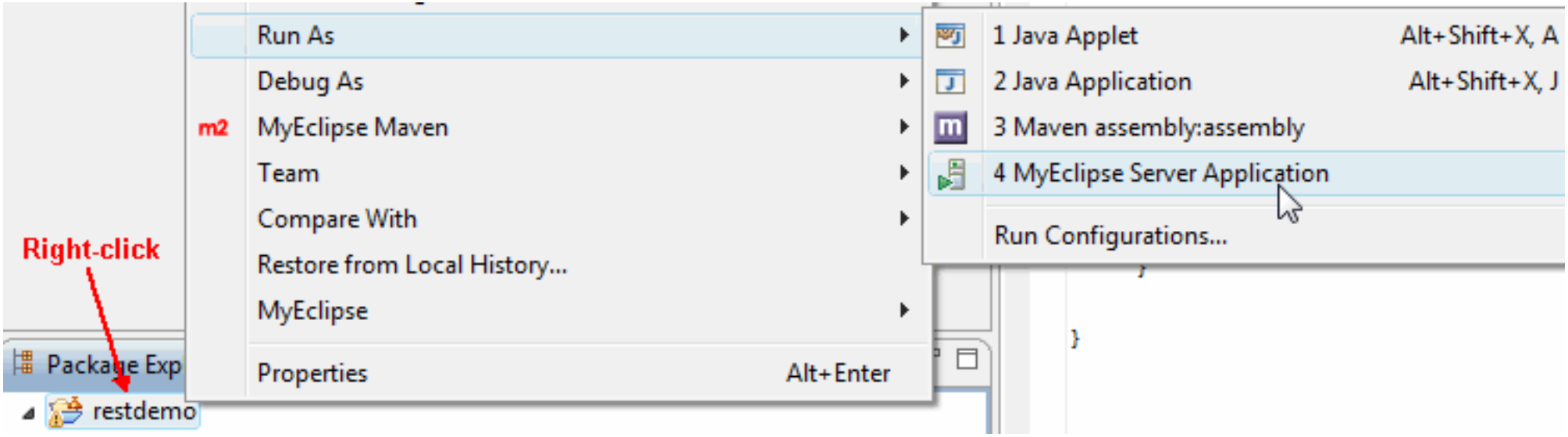


top

5. Deploying & Testing the Web Service

5.1 Deploying & Running the *restdemo* Project

The fastest way to deploy our web service is to deploy our web project using the *Run As* or *Debug As* action of **MyEclipse Server Application**. We can do that by right-clicking on our project, going down to *Debug As* (or *Run As*) and selecting **MyEclipse Server Application**:



If you have multiple server connectors configured, MyEclipse will ask you which one you want to use, for the purpose of this tutorial select **MyEclipse Tomcat**. If you don't have any connectors configured, MyEclipse Tomcat will be used automatically for you to deploy your project to and then run.

Now MyEclipse will perform the following steps for you automatically:

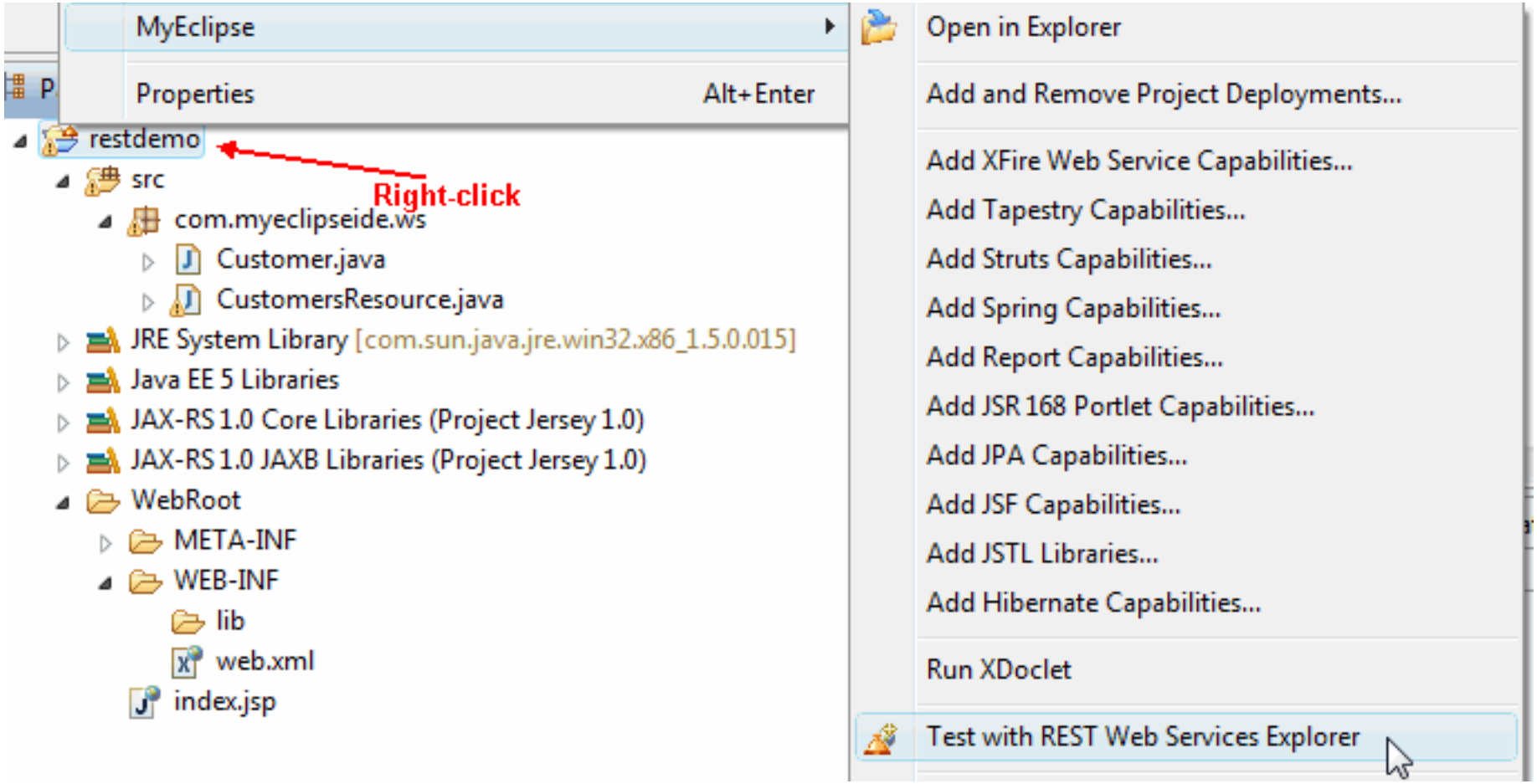
1. Package our web project, and deploy it in *Exploded* format to the application server
2. Start the application server for us, loading our web project

The MyEclipse Web Browser will popup and show you the default *index.jsp* page of our web app, we don't actually need this because we aren't testing a web page, so you can close this view.

5.2 Testing the Web Service with the REST Web Services Explorer (PRO Only)

The easiest way to test our web service is to use the **REST Web Services explorer**. Since the explorer is available only to MyEclipse Professional subscribers, if you are MyEclipse Standard subscriber, please follow the instructions in section 5.3.

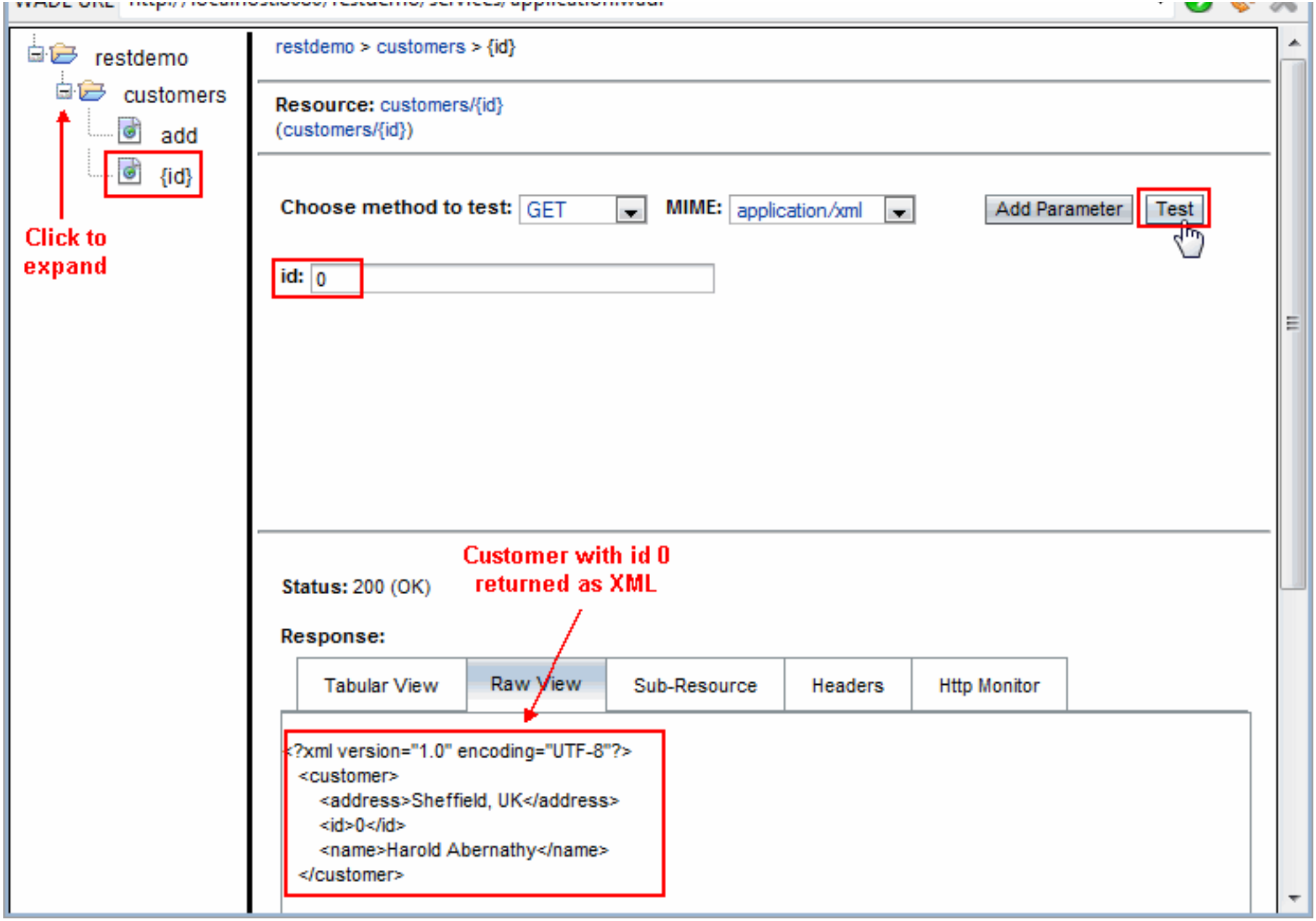
1. Right-click the *restdemo* project and select **Test with REST Web Services Explorer** from the **MyEclipse** submenu as shown below:



Note: If you deployed *restdemo* to an application server other than MyEclipse Tomcat, the WADL URL used in the explorer may contain an incorrect port, preventing the explorer from loading your WADL file. Correct the port and click the go button to proceed.

You may also open the REST Web Services Explorer using the drop down from the main eclipse toolbar. In this case, you need to manually enter the path of a WADL file in the address bar before proceeding.

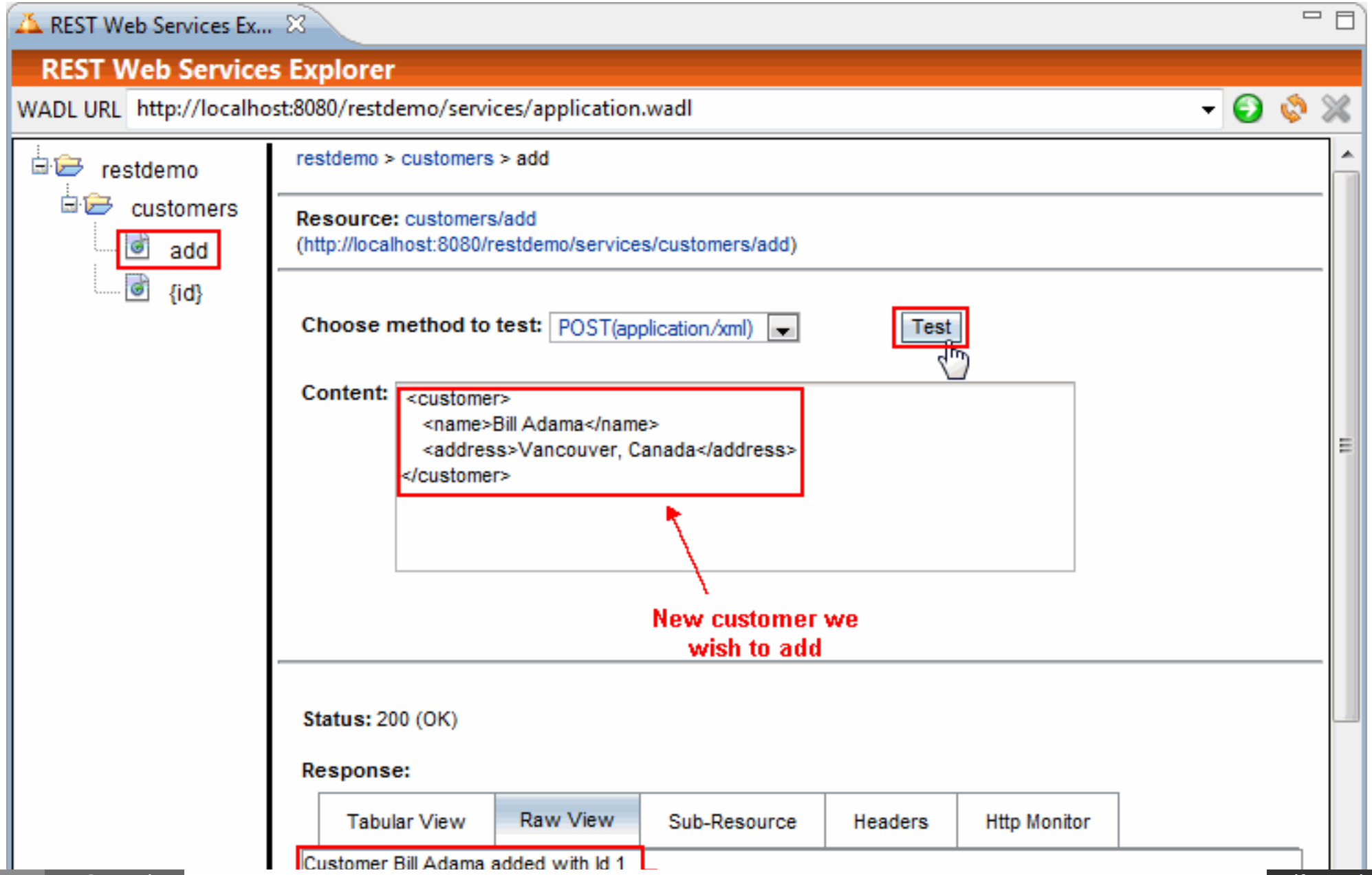
2. Expand the *customers* node and select *{id}*. In the *id* field on the right, enter *0* and click Test. In the **Raw View** tab, observe the lone customer we have in our map being returned in XML.



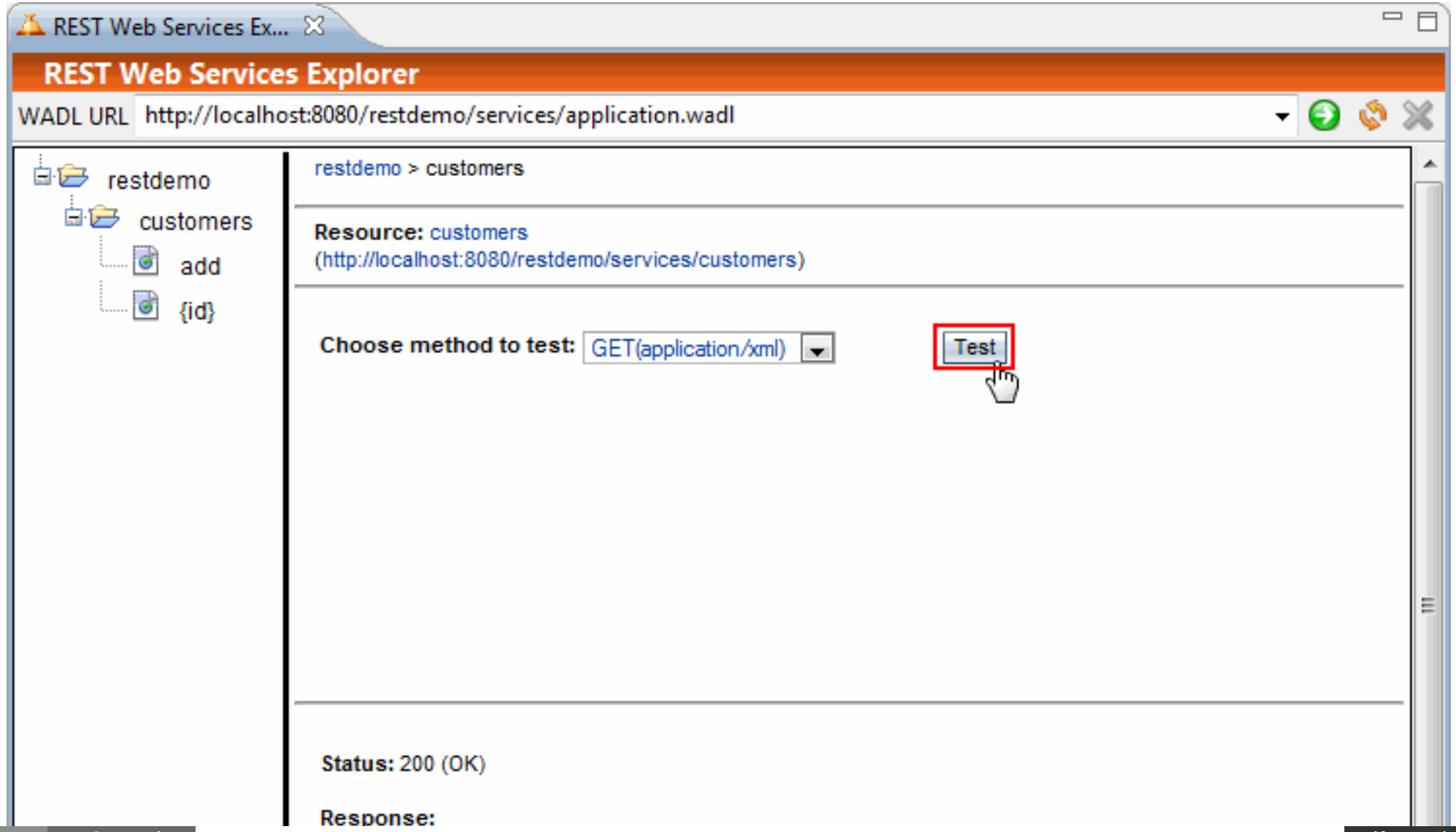
3. Let's add a customer to our list. Expand select *add* under the *customers* node. In the content area on the right, paste the following and click **Test**:

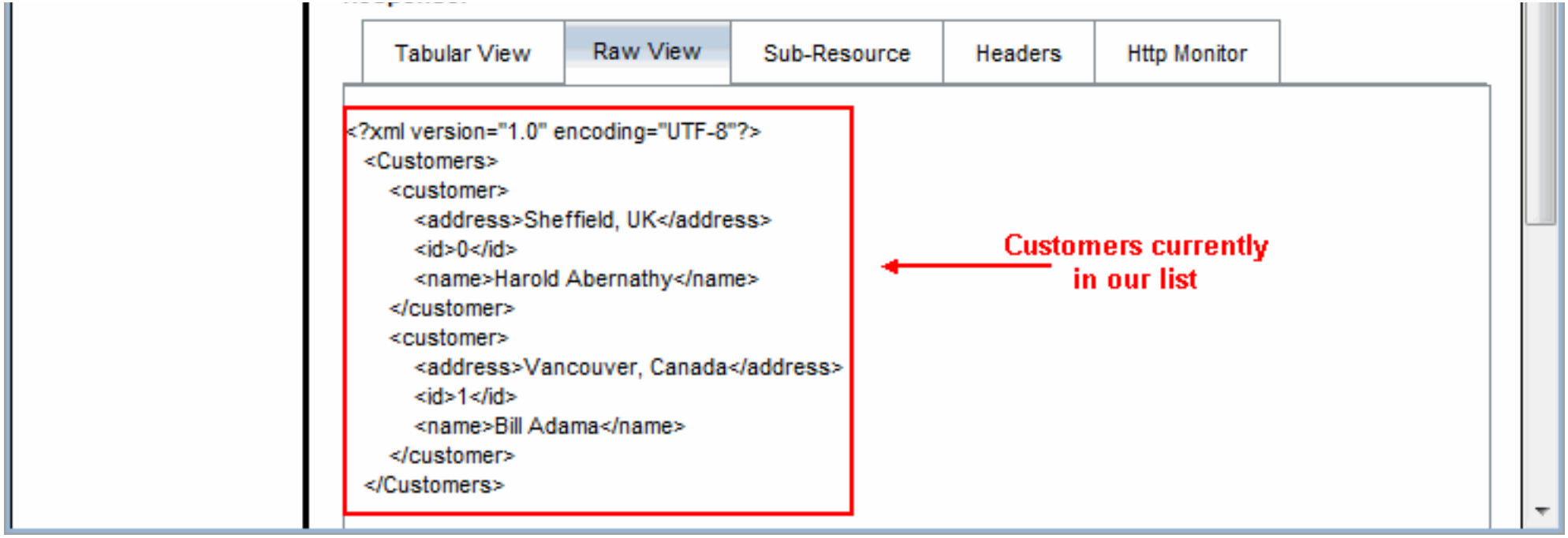
```
<customer>
    <name>Bill Adama</name>
    <address>Vancouver, Canada</address>
</customer>
```

The response should now say: *Customer Bill Adama added with Id 1*



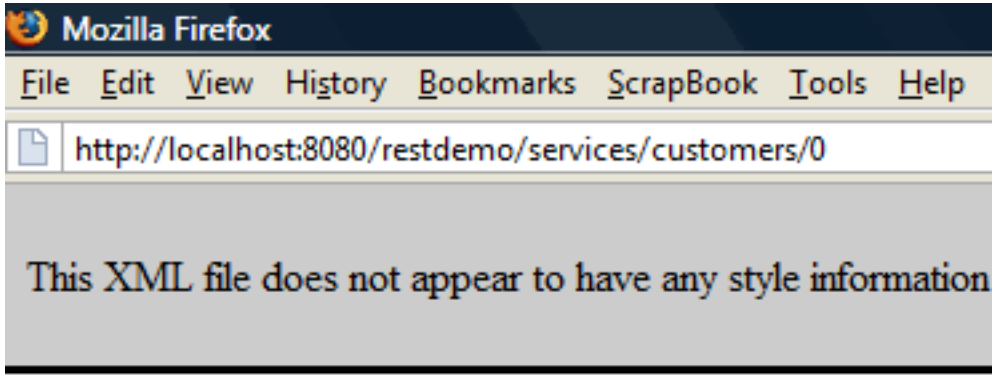
4. Select the *customers* node and click **Test**. The two customers in our list are returned by the service in XML.





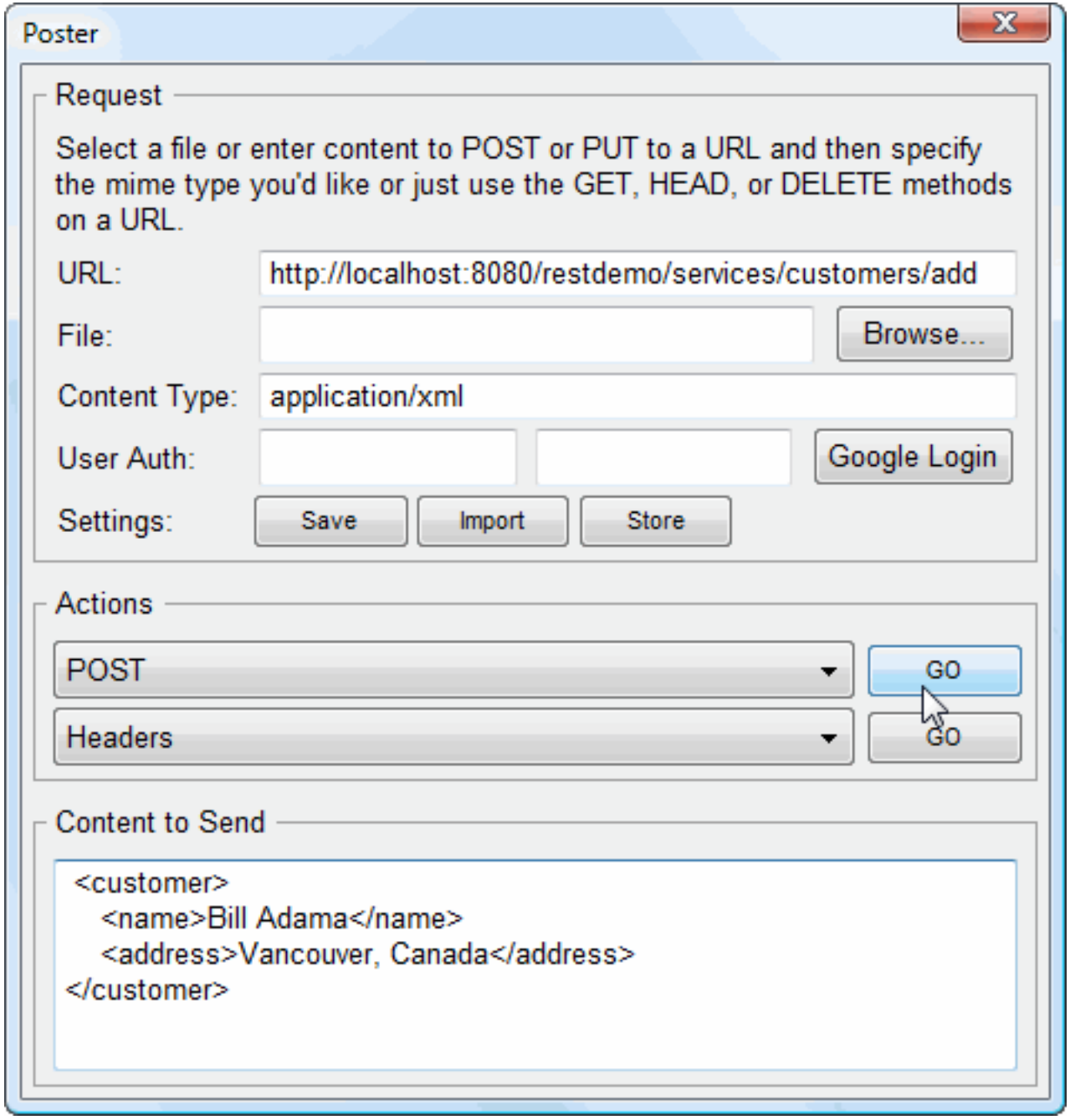
5.3 Testing the Web Service Using a Standard Browser

1. Open http://localhost:8080/restdemo/services/customers/0 in your browser to see the first customer in XML.

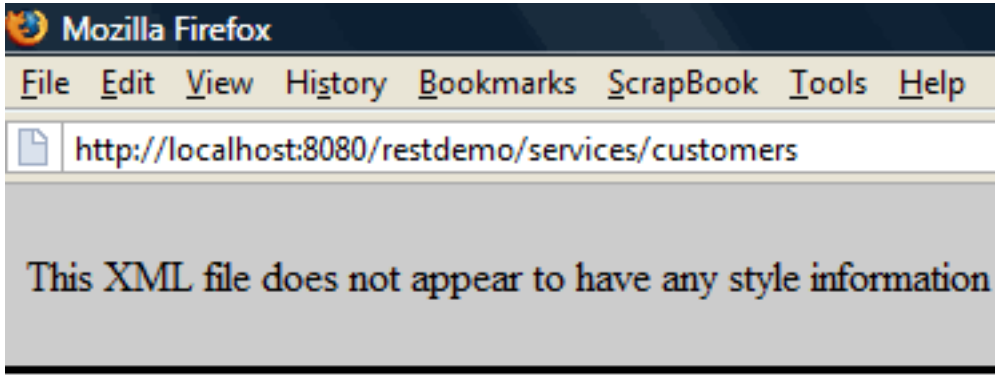



```
−<customer>
    <address>Sheffield, UK</address>
    <id>0</id>
    <name>Harold Abernathy</name>
  </customer>
```

2. To add a customer to our list, we need to send customer data to the service via an HTTP POST request. You could use a Firefox extension like Poster to make the request as shown below.



3. Open http://localhost:8080/restdemo/services/customers in your browser to get a list of all customers in XML.




```
- <Customers>
  - <customer>
      <address>Sheffield, UK</address>
      <id>0</id>
      <name>Harold Abernathy</name>
    </customer>
  - <customer>
      <address>Vancouver, Canada</address>
      <id>1</id>
      <name>Bill Adama</name>
    </customer>
  </Customers>
```

Note: If you deployed *restdemo* to an application server other than MyEclipse Tomcat, you may need to correct the port in the above links depending on your application server.

top

6. Resources

In this section we want to provide you with additional links to resources that supplement the topics covered in this tutorial. While

this is not an exhaustive list, we do make an effort to point to the more popular links that should provide you with diverse, high-quality information.

- REST Resources
  - restdemo.zip contains the *restdemo* project we created in this tutorial.
  - MyEclipse REST Web Services Overview
  - JSR 311- JAX-RS: Java API for RESTful Web Services.
  - Project Jersey is the JAX-RS reference implementation MyEclipse uses.
  - RESTful Web Services Developer's Guide
- MyEclipse SOAP Web Service Tutorials
  - Developing JAX-WS Web Services & Clients
  - Developing JAX-WS Web Services for WebSphere (MyEclipse Blue Edition)
  - Developing JAX-RPC Web Services for WebSphere (MyEclipse Blue Edition)

top

7. Feedback

We would like to hear from you! If you liked this tutorial, have some suggestions or even some corrections for us, please let us know. We track all user feedback about our learning material in our Documentation Forum. Please be sure to let us know which piece of MyEclipse material you are commenting on so we can quickly pinpoint any issues that arise.

top