

Developing RESTful Web Services with JAX-RS (Jersey)

Jersey is the reference library implementation of the JAX-RS standard. In this tutorial you will set up a sample Jersey project and implement some simple RESTful resources.

Disclaimer: this guide was created for **JDK8**, **eclipse-jee-2021-03-R-win32-x86_64** and **Tomcat 10.0.5 x64**. This guide may not work properly for other JDK, Eclipse EE or Tomcat versions.

If you have problems running the code samples, you should first check that you have installed the correct versions of each piece of software.

1. Download and unzip Eclipse IDE for Enterprise Java and Web Developers 2021-03 R to a convenient location (e.g. C:)

Pick the correct version of Eclipse EE for your machine from:

<https://www.eclipse.org/downloads/packages/release/2021-03/r/eclipse-ide-enterprise-java-and-web-developers>

2. Download and unzip Tomcat 10.0.5 x64 to a convenient location (e.g. C:)

Windows_x64: <https://ftp.heanet.ie/mirrors/www.apache.org/dist/tomcat/tomcat-10/v10.0.5/bin/apache-tomcat-10.0.5-windows-x64.zip>

or pick the correct version for your machine from: <https://tomcat.apache.org/download-10.cgi>

3. Setup an Eclipse EE Workspace

Open Eclipse EE (not standard edition!) and create a fresh workspace.

4. Add Tomcat server to your Eclipse EE Project

Go to the “Servers” tab in Eclipse EE and click the link to create a new server.

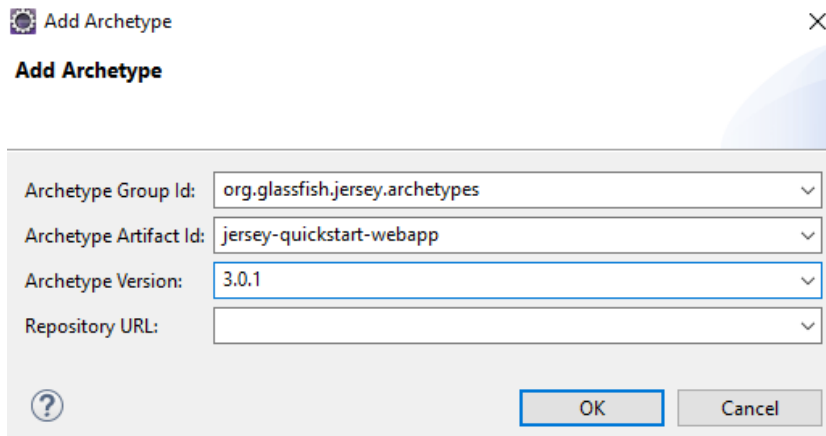
Select “Tomcat v10.0 Server” and click Next.

Browse to the Tomcat directory from Step 1 and then click Finish – “Tomcat v10.0 Server at localhost” should be displayed on the “Servers” tab.

5. Setup a Jersey sample project

Right click in the Project Explorer in Eclipse, select “New > Project...>Maven Project” and click Next and Next again.

In the Select an Archetype pane, click “Add Archetype...” and enter the following values:

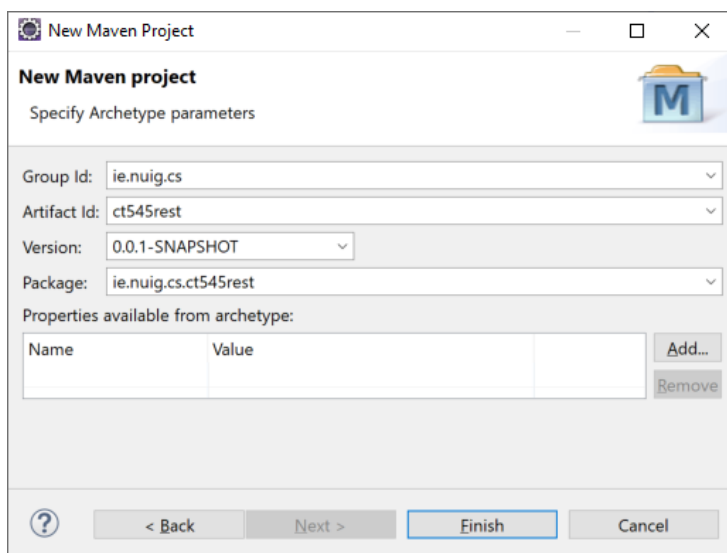


Click OK, then select the archetype that was just created and click Next.

Enter **ie.nuig.cs** as the Group Id (package name)

Enter **ct545rest** as the Artifact Id

N.B. it is **essential** that you enter the Group Id and Artifact Id exactly as shown. These are case sensitive, and the URLs later in this .pdf will not work if the Artifact Id is not **ct545rest**



Click Finish, and the project ct545rest should be created in your Project Explorer. This will import and set up all required libraries, configuration etc.

(To remove the HttpServlet library not found error, right click on the ct545rest project, select "Properties>Project Facets>Runtimes" and make sure that Apache Tomcat v10.0 is checked)

6. Run the Jersey sample project

Right click on the ct545rest project in the Project Explorer, and select “Run As > Run on Server”. Click Next and then Finish.

You should see the landing page at <http://localhost:8080/ct545rest/> (generated by index.jsp) for the sample Jersey RESTful Web Application.

Click on the “Jersey resource” link to see the text “Got it!” displayed – this is the output of the getIt() method in the class MyResource. Note that this resource is accessed at the following URL: <http://localhost:8080/ct545rest/webapi/myresource>

7. Examine the structure of the sample project

Open and inspect the following files:

src>main>webapp>WEB-INF>web.xml

pom.xml

src>main>webapp>index.jsp

ie.nuig.cs.ct545rest>MyResource.java

8. Add a HelloWorld Resource which uses the @PathParam annotation

Copy the file HelloName.java from the CT545_2021_REST_tutorial folder into the ie.nuig.cs.ct545rest package in your Eclipse EE project, and then stop and restart Tomcat to deploy the resource.

Note that this class has two methods with the @GET annotation. One may be accessed using the URL: <http://localhost:8080/ct545rest/webapi/hello>. The output of this method is the string “Hello World!”.

The second method uses the @PathParam annotation to accept a name as an input parameter, and may be accessed at the URL: <http://localhost:8080/ct545rest/webapi/hello/name/Patrick>. The output of the method is the string “Hello Patrick!”. Note that “Patrick” is the input parameter which is passed via the URL. You can edit the path parameter portion of the URL to display a different name.

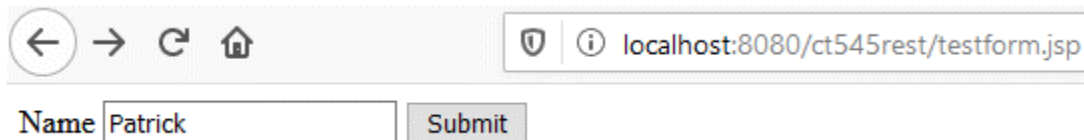
9. Handling HTML form values using the @FormParam annotation

Copy the file testform.jsp from the CT545_2021_REST_tutorial into the ct545rest/src/main/webapp directory of your project.

Note: the extension .jsp stands for JavaServer Pages – these files have a structure similar to HTML documents, and they can also have dynamic elements that are generated using scriptlets (sections of embedded Java source code within the HTML). JSP pages are essentially a very high-level abstraction of servlets. As JSP is an older technology we will not spend a lot of time discussing it in class.

Copy the file HandleForm.java from the CT545_2021_REST_tutorial into the ie.nuig.cs.ct545rest package in your project.

Navigate to <http://localhost:8080/ct545rest/testform.jsp> in your browser - you should see a simple HTML form like the one below.



Enter a value into the form field and click the Submit button – this will POST the form and its values to HandleForm.java at the URL <http://localhost:8080/ct545rest/webapi/handleform>



Note that the value Patrick is posted as a form parameter called "name" by the form on the .jsp page. The annotation `@FormParam("name")` on the `respondToForm()` method allows this form parameter to be picked up and used in a similar manner to the path parameter in part 7 above. Your RESTful resource methods can handle as many form parameters as you wish, once the correct annotations are present.

10. Add a HelloWorld Resource which is a singleton

Copy the file `HelloSingleton.java` from the `CT545_2021_REST_tutorial` into the `ie.nuig.cs.ct545rest` package in your Eclipse EE project. By default, Jersey creates a new instance of the class which handles a given `@Path` for each new request which is received. Note that the class `HelloSingleton` makes use of the `@Singleton` annotation in order to implement a singleton design pattern (i.e. there is only ever a single instance of `HelloSingleton` created). The `HelloSingleton` Resource may be accessed at the URL: <http://localhost:8080/ct545rest/webapi/hellosingleton>. Refresh the browser window, and you should see that the counter variable is incremented with each new request which is handled by the resource.

Comment out the `@Singleton` annotation and refresh the page. You should see that the counter now only ever displays 1 – the default instantiation behaviour for Jersey has been restored. You may need to stop and restart Tomcat for the change to take effect.

11. Exercise: Use PuTTY to retrieve different responses based on the client's Accept: type

One of the most flexible features of a RESTful architecture is the ability for servers to service requests in the client's preferred format. This exercise will demonstrate how this may be achieved using Jersey with PuTTY as the client.

Copy the file `HelloAcceptTypes.java` from `CT545_2021_REST_tutorial` into the `ie.nuig.cs.ct545rest` package in your Eclipse EE project. Note that in this example there are two methods which have the `@GET` annotation, and which can handle requests to the `@Path("/helloaccepttypes")`, but that they have different `@Produces()` annotations. Adding different `@Produces` annotations for the same `@Path()` allows different implementations based on the `Accept: type` (e.g. `Accept: text/html`) which is specified by the client.

Open the following URL in a web browser to check that the example is working:
<http://localhost:8080/ct545rest/webapi/helloaccepttypes>

Download the PuTTY binary for Windows x64 from:
<https://the.earth.li/~sgtatham/putty/latest/w64/putty.exe>

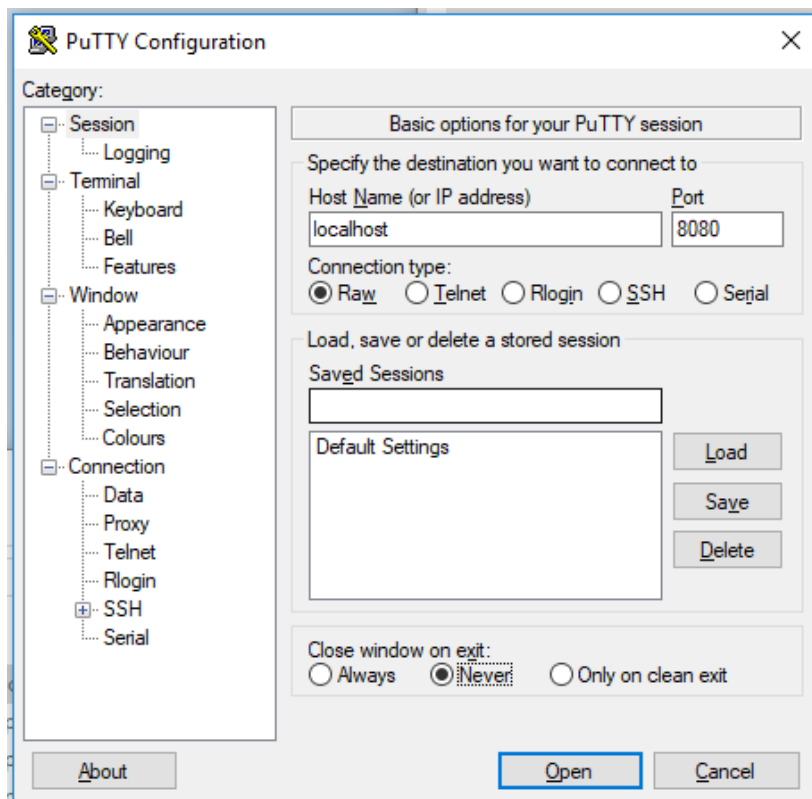
Open PuTTY and input the following settings, then click Open:

Host: localhost

Port: 8080

Connection type: Raw

Close window on exit: Never



Copy the contents of the Putty_Request.txt file into the PuTTY console, and hit enter. You should see a response like the following:

```
DESKTOP-RDKL74G - PuTTY
GET /ct545rest/webapi/helloaccepttypes HTTP/1.1
Host: localhost:8080
Accept: text/plain

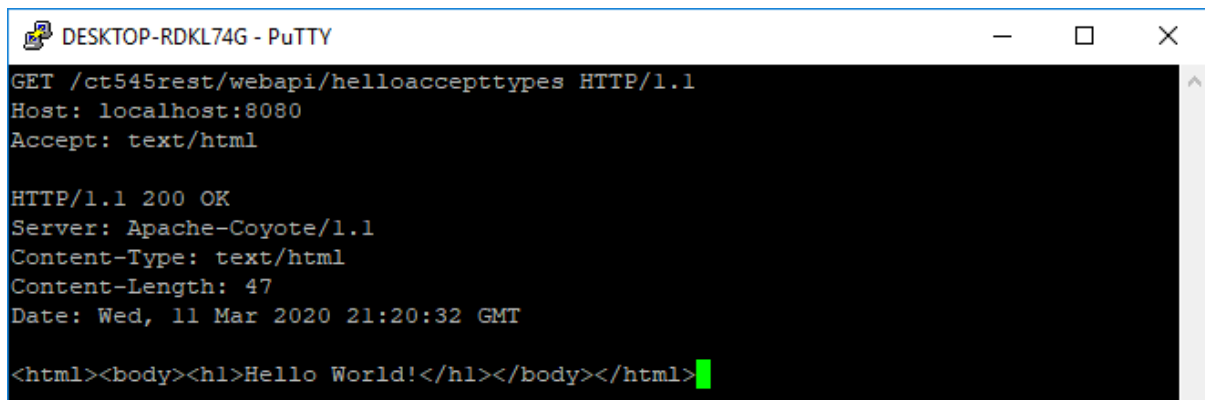
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/plain
Content-Length: 12
Date: Wed, 11 Mar 2020 21:18:24 GMT

Hello World!
```

```
jiarongli@Jiarongs-MacBook-Pro ~ % curl -v http://localhost:8080/ct545rest/webapi/helloaccepttypes
* Trying ::1:8080...
* Connected to localhost (::1) port 8080 (#0)
> GET /ct545rest/webapi/helloaccepttypes HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.77.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Content-Type: text/plain
< Content-Length: 12
< Date: Fri, 14 Jan 2022 17:56:24 GMT
<
* Connection #0 to host localhost left intact
Hello World!%
```

```
jiarongli@Jiarongs-MacBook-Pro ~ % curl -H 'Accept:text/html' POST http://localhost:8080/ct545rest/webapi/helloaccepttyp
es
curl: (6) Could not resolve host: POST
<html><body><h1>Hello World!</h1></body></html>%
```

Change the Accept: type to "Accept: text/html" (without quotes), then copy the modified GET request back into PuTTY. You should see a response like the following:



```

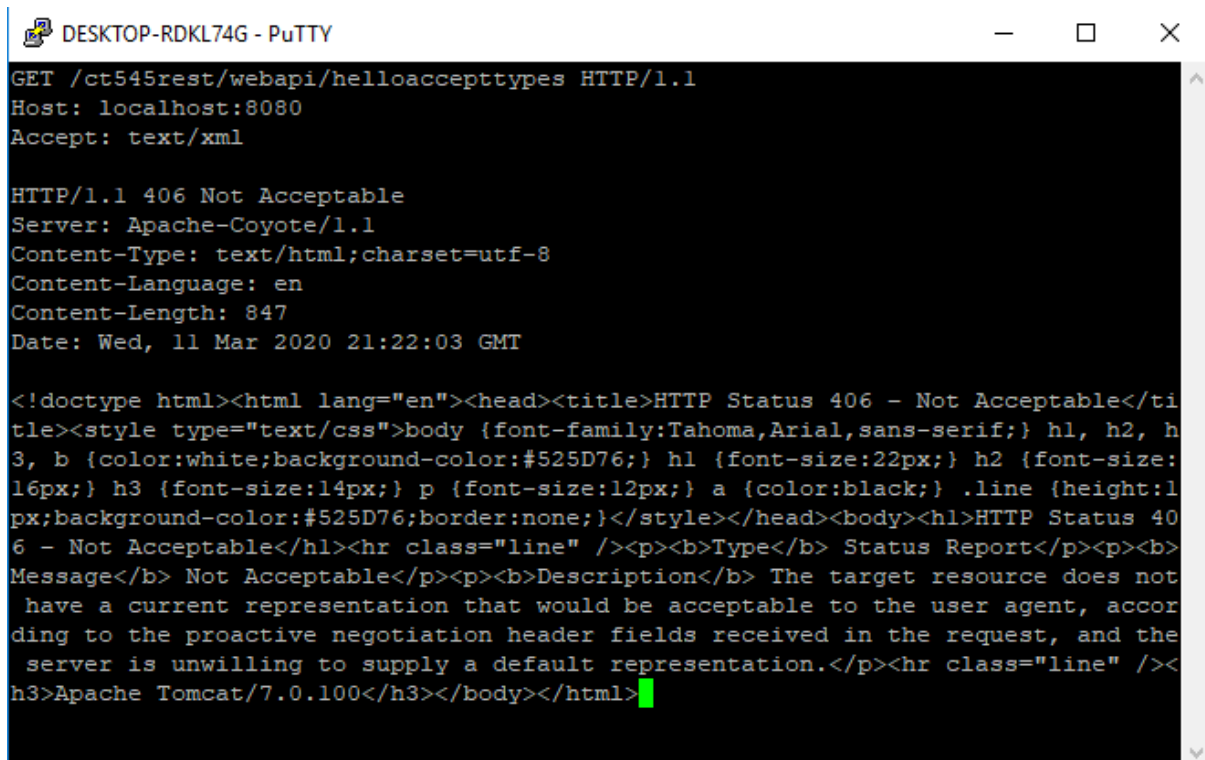
DESKTOP-RDKL74G - PuTTY
GET /ct545rest/webapi/helloaccepttypes HTTP/1.1
Host: localhost:8080
Accept: text/html

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html
Content-Length: 47
Date: Wed, 11 Mar 2020 21:20:32 GMT

<html><body><h1>Hello World!</h1></body></html>

```

Change the Accept: type to "Accept: text/xml" (or any other Accept: type which does not have an @Produces() annotation to handle it), then copy the modified GET request back into PuTTY. You should see a response like the following:



```

DESKTOP-RDKL74G - PuTTY
GET /ct545rest/webapi/helloaccepttypes HTTP/1.1
Host: localhost:8080
Accept: text/xml

HTTP/1.1 406 Not Acceptable
Server: Apache-Coyote/1.1
Content-Type: text/html; charset=utf-8
Content-Language: en
Content-Length: 847
Date: Wed, 11 Mar 2020 21:22:03 GMT

<!doctype html><html lang="en"><head><title>HTTP Status 406 - Not Acceptable</ti
tle><style type="text/css">body {font-family:Tahoma,Arial,sans-serif;} h1, h2, h
3, b {color:white;background-color:#525D76;} h1 {font-size:22px;} h2 {font-size:
16px;} h3 {font-size:14px;} p {font-size:12px;} a {color:black;} .line {height:1
px;background-color:#525D76;border:none;}</style></head><body><h1>HTTP Status 40
6 - Not Acceptable</h1><hr class="line" /><p><b>Type</b> Status Report</p><p><b>
Message</b> Not Acceptable</p><p><b>Description</b> The target resource does not
have a current representation that would be acceptable to the user agent, accord
ing to the proactive negotiation header fields received in the request, and the
server is unwilling to supply a default representation.</p><hr class="line" /><
h3>Apache Tomcat/7.0.100</h3></body></html>

```

12. Add a Calculate Resource

Copy the file CalculateResource.java from the CT545_2021_REST_tutorial folder into the ie.nuig.cs.ct545rest package in your Eclipse EE project. Stop and restart Tomcat.

Note that this example makes use of the javax.ws.rs.core.Response class to build the HTTP response to the caller. You can use the Response class to generate response codes other than 200 OK.

More on javax.ws.rs.core.Response:

<https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Response.html>

Test out the methods which are made available by the Calculate resource by building the appropriate URLs, e.g.

<http://localhost:8080/ct545rest/webapi/calculate/add/1/1>

<http://localhost:8080/ct545rest/webapi/calculate/subtract/1/1>

<http://localhost:8080/ct545rest/webapi/calculate/squareroot/1>

13. Exercise: Create an interface-based version of the Calculate Resource

Create two new .java files in the ie.nuig.cs.ct545rest package: CalculateInt.java and CalculateImpl.java

In CalculateInt.java, create an interface which has the same method signatures and annotations as the CalculateResource.java file above. Do not include the outer @Path() annotation on the interface, but do include the @Path() annotation on each method declaration.

In CalculateImpl.java, create the implementation for the methods specified in CalculateInt.java by copying them from CalculateResource.java

Apply the @Path("/calculateimpl") annotation to the CalculateImpl class, NOT to the CalculateInt interface (otherwise Jersey will not be able to find your implementation class and will return a 404).

Note that by using an interface for (almost) all of the JAX-RS/Jersey annotations, it is possible to separate out business logic (the implementation) from the annotations, resulting in cleaner code.

14. JSON Externalisation with JAX-RS/Jersey

Open pom.xml and remove the opening and closing xml comments (i.e. delete lines 51 and 56). Lines 52-55 should remain as is. Maven should download the appropriate libraries to enable JSON data binding in the jersey-quickstart-webapp.

```
51      <!-- uncomment this to get JSON support
52      <dependency>
53          <groupId>org.glassfish.jersey.media</groupId>
54          <artifactId>jersey-media-json-binding</artifactId>
55      </dependency>
56      -->
```

Copy the files Employee.java, Employees.java and EmployeeService.java from the CT545_2021_REST_tutorial folder into the ie.nuig.cs.ct545rest package in your Eclipse EE project. Stop and restart Tomcat.

Employee.java is our data model class, instances of which can be marshalled/unmarshalled to/from JSON format during RESTful CRUD operations (i.e. when the HTTP verbs POST, GET, PUT and DELETE are used).

Employees.java is a wrapper class for an ArrayList that holds instances of the class Employee.

EmployeeService.java is the JAX-RS resource class that will handle all incoming HTTP requests. Note that this class contains three different Java methods, that are available at different URLs:

<http://localhost:8080/ct545rest/webapi/employees> - When a GET request is received at this URL, a JSON representation of the entire list of employees is returned in the body of the HTTP response.

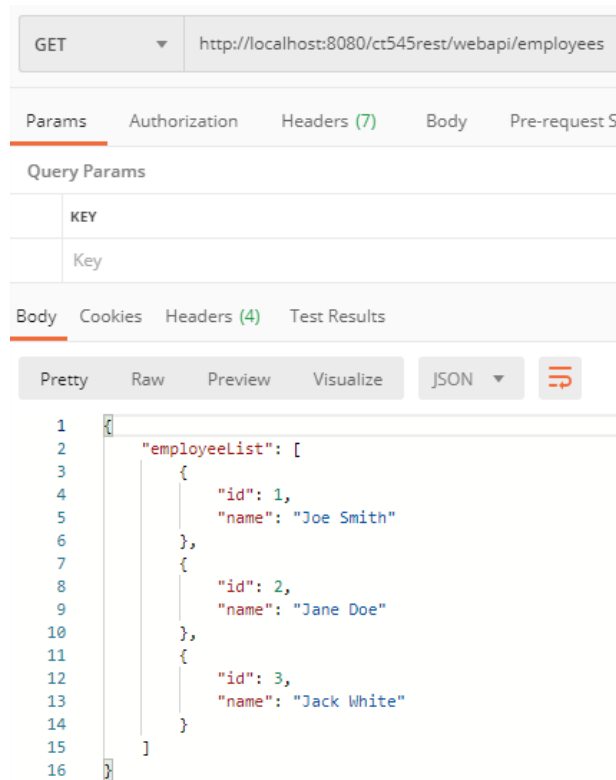
<http://localhost:8080/ct545rest/webapi/employees/2> - When a GET request is received at this URL, a JSON representation of the employee with the given id (in this case id=2) is returned in the body of the HTTP response. A 404 response is returned if an invalid id is specified.

<http://localhost:8080/ct545rest/webapi/employees> - When a POST request with a JSON representation of an Employee is received at this URL, the JSON is unmarshalled into a Java Employee object. This new object is then added to the ArrayList of Employees, and the JSON representation of the object that was just created is returned in the HTTP response.

We will use Postman to test out the RESTful API that is made available by EmployeeService.java. Download the Postman app from <https://www.postman.com/downloads/>

Try testing out the functionality of each of the Java methods above using Postman.

First retrieve the JSON representation of all Employees:



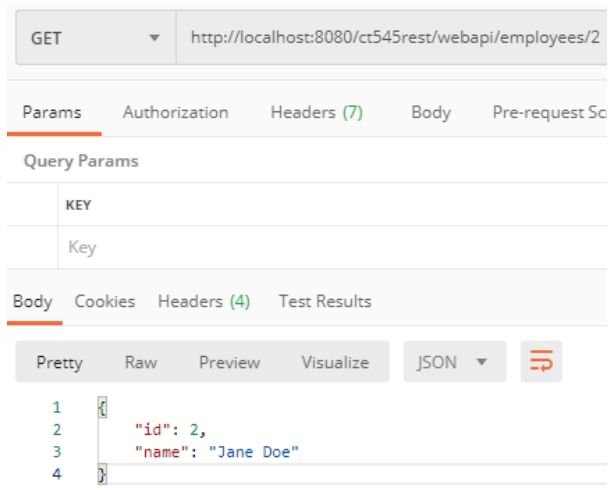
Next, retrieve the JSON representation of an employee with a specific id:

retrieve the JSON representation of all Employees:

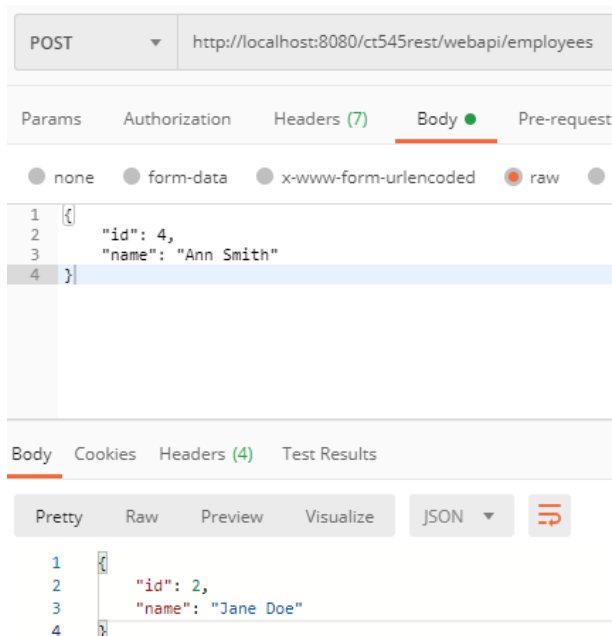
```
jiarongli@Jiarongs-MacBook-Pro ~ % curl http://localhost:8080/ct545rest/webapi/employees
{"employeeList":[{"id":1,"name":"Joe Smith"},{"id":2,"name":"Jane Doe"},{"id":3,"name":"Jack White"},{"id":4,"name":"ab"
}]]}%
jiarongli@Jiarongs-MacBook-Pro ~ % █
```

retrieve the JSON representation of an employee with a specific id:

```
jiarongli@Jiarongs-MacBook-Pro ~ % curl http://localhost:8080/ct545rest/webapi/employees/1
{"id":1,"name":"Joe Smith"}%
jiarongli@Jiarongs-MacBook-Pro ~ % curl http://localhost:8080/ct545rest/webapi/employees/4
{"id":4,"name":"ab"}%
jiarongli@Jiarongs-MacBook-Pro ~ % █
```



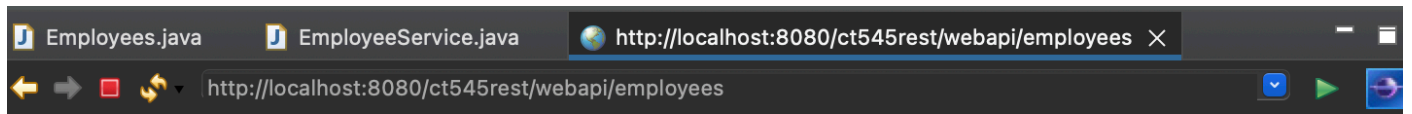
Next, create a POST request with a JSON representation of a new Employee in the request body. Be sure to set the media type in the request to JSON before sending the POST request.



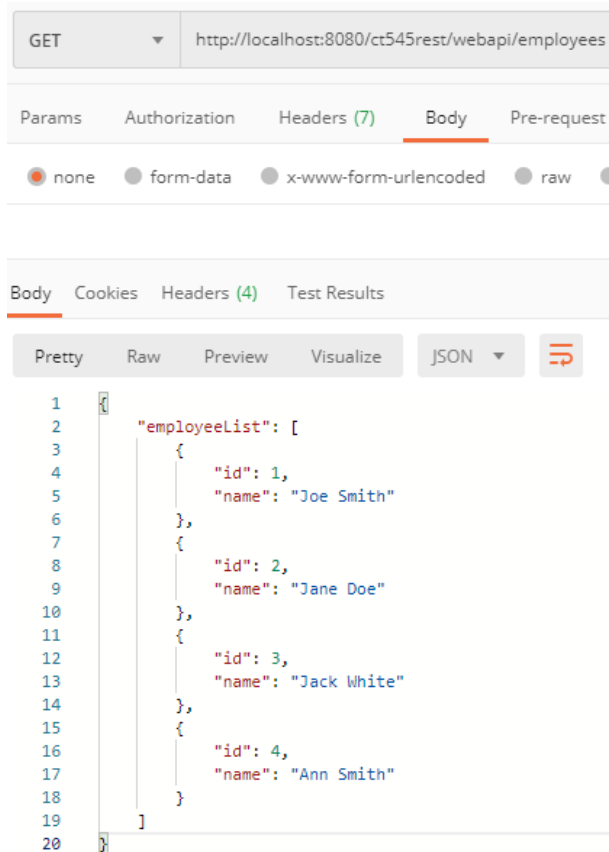
Finally, run the first Postman query to retrieve the list of all employees in JSON format again. Note how the new Employee now appears in the list of all Employees on the server.

create a POST request with a JSON representation of a new Employee in the request body.

```
jiarongli@Jiarongs-MacBook-Pro ~ % curl -X POST http://localhost:8080/ct545rest/webapi/employees -H "Content-Type:application/json" -d '{"id":4,"name":"ab"}'
```



```
{ "employeeList": [ { "id": 1, "name": "Joe Smith" }, { "id": 2, "name": "Jane Doe" },  
{ "id": 3, "name": "Jack White" }, { "id": 4, "name": "ab" } ] }
```



15. Documenting JAX-RS/Jersey RESTful APIs using Web Application Description Language

Jersey provides a simple and automatic way to document the functionality of a RESTful API using the HTTP OPTIONS method. A Web Application Description Language (WADL) document is automatically generated describing the resource endpoints (URLs), available HTTP methods, input and output parameters etc. WADL is based on XML – note the very similar structure.

Once you have created your RESTful web service, the WADL description could be used by other programmers who wish to know how to use your service. WADL essentially serves the same purpose in RESTful web services as interfaces do in Java; WADL and Java interfaces describe how methods may be used, without the need to know or specify any implementation details.

Try using Postman to send a HTTP request with the OPTIONS method to some of the RESTful resource URLs from earlier in this document and see what happens. The example below is for the `EmployeeService.java` resource from part 12 above.

OPTIONS ▼ http://localhost:8080/ct545rest/webapi/employees

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL

This request

Body Cookies Headers (6) Test Results

Pretty Raw Preview Visualize

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.30.1 2020-02-21 08:10:47"/>
  <grammars/>
  <resources base="http://localhost:8080/ct545rest/webapi/">
    <resource path="employees">
      <method id="addEmployee" name="POST">
        <request>
          <representation mediaType="application/json"/>
        </request>
        <response>
          <representation mediaType="application/json"/>
        </response>
      </method>
      <method id="getAllEmployees" name="GET">
        <response>
          <representation mediaType="application/json"/>
        </response>
      </method>
      <resource path="{id}">
        <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id" style="template" type="xs:int"/>
        <method id="getEmployee" name="GET">
          <response>
            <representation mediaType="application/json"/>
          </response>
        </method>
      </resource>
    </resource>
  </resources>
</application>
```

Appendix: useful links

<https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Response.html>

<https://www.oracle.com/technical-resources/articles/java/jaxrs20.html>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

Oracle tutorial on JSP: <https://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html>

Tutorial on sending HTML form parameters to a JAX-RS resource:

<https://www.logicbig.com/tutorials/java-ee-tutorial/jax-rs/form-param-post.html>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

