**PaaS - Development of RESTful Java Web Services with AWS Elastic Beanstalk**

**Objective:** Get practical experience in RESTful Java Web services development and deployment on Cloud PaaS with AWS Elastic Beanstalk as well as in creating and running web service clients.

**Tasks:**

1. Development of a RESTful Java Web Service with the Jersey framework
2. Testing a RESTful Java Web Service using SoapUI tool
3. Development of a Web Service Java client
4. Deployment of a RESTful Java Web Service on AWS Elastic Beanstalk PaaS

**Lab environment**:

* Java 8 (JDK 1.8.0)
* Eclipse IDE for Enterprise Java developers 2020-06
  + AWS Toolkit for Eclipse 2.0
  + Maven integration for Eclipse WTP 1.2.0
* Jersey framework 2.25.1
* Apache Tomcat web server 8.5
* SoapUI tool

*Contents*

[Step 1: Developing your own RESTful Web Service 3](#_heading=h.3znysh7)

[Step 2: Developing your own Client of a RESTful Web Service 11](#_heading=h.2et92p0)

[Step 3: Deploying your RESTful Web Service on AWS Elastic Beanstalk 12](#_heading=h.tyjcwt)

[Step 4: Testing your RESTful Web Service 12](#_heading=h.3dy6vkm)

[Step 5: Removing AWS Elastic Beanstalk Resources 12](#_heading=h.1t3h5sf)

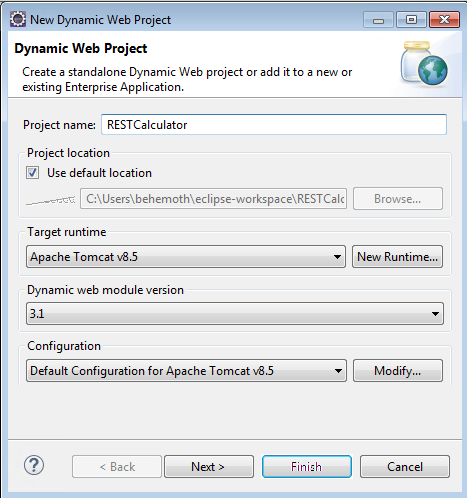
# Step 1: Developing your own RESTful Web Service

Using the example considered in the previous lab, develop a simple RESTful Java Web Service and a Client on your own. It can be an advanced calculator performing an extended set of arithmetic operations, currency exchange rate converter, a converter between metric and imperial units, etc. Do not forget to delete AWS resources after use.

***1. Create a new Dynamic Web Project in Eclipse***

Open Eclipse IDE.

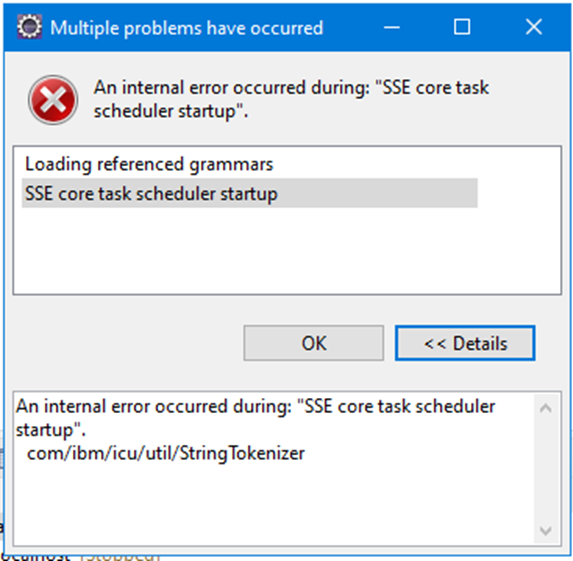
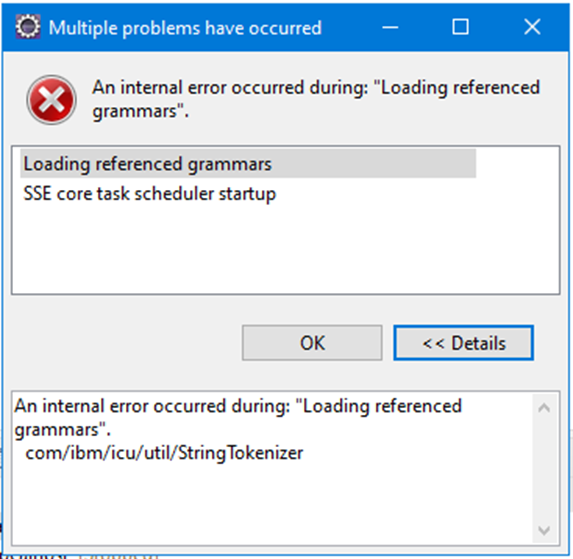
Select Dynamic Web Project in Eclipse -> File -> New. Specify project name (e.g. RESTCalculator), target runtime (Apache Tomcat v8.5) and dynamic web module version (3.1). Click on Finish to create the project.



***2. Create a Deployment Descriptor***

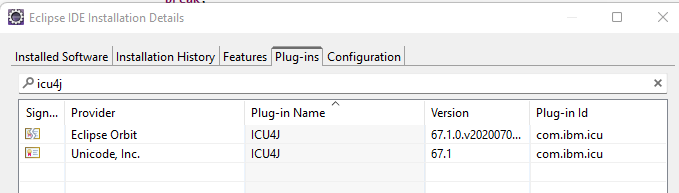
Right click on the project. Select Java EE Tools and click on Generate Deployment Descriptor Stub. This will create web.xml file under src\main\webapp\WEB-INF\ folder.

**Note:** At this point, or later, when editing web.xml, you may (or may not - depends on the state of Eclipse IDE) encounter one or two error messages:

This is because some of Eclipse plugins (namely, ICU4J by ‘Eclipse Orbit’ and ‘Unicode, Inc’; this plugin is used for XML parsing) updates have not been propagated yet to all lab PCs (Eclipse is updated in the background; this is why some students may experience this issue, while others may not - it depends on Eclipse update status). This error leads to exceptions arising during service deployment. To update plugins forcefully, follow the instructions below.

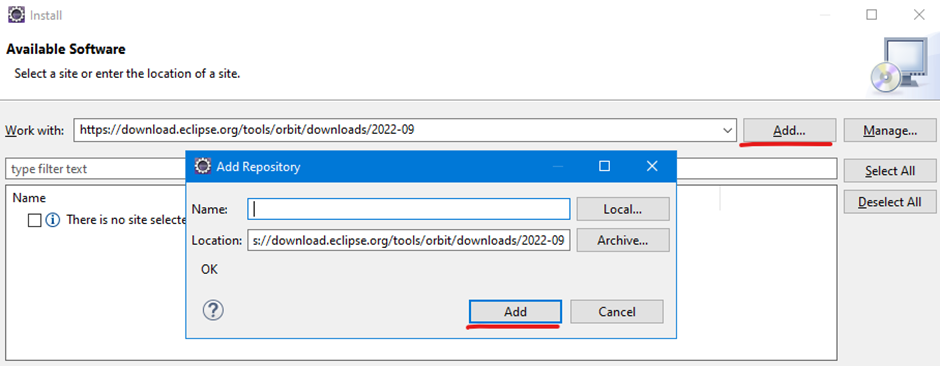
***Check ICU4J plugin version***. You can verify your ICU4J plugin version via menu: Help -> About Eclipse IDE -> Installation Details -> plugins (filter results for ‘icu4j’). This is how it should looks like when versions are compatible to each other.:



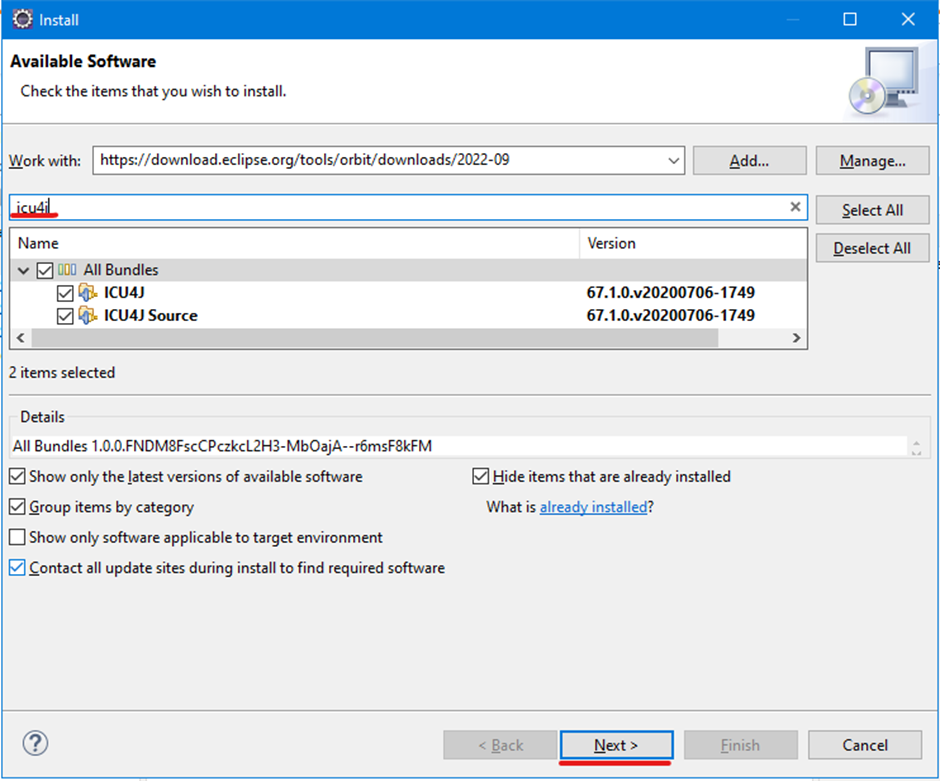
If ICU4J by ‘Eclipse Orbit’ plugin version is older than ICU4J ‘Unicode, Inc’ version, then you need to update ‘Eclipse Orbit’.

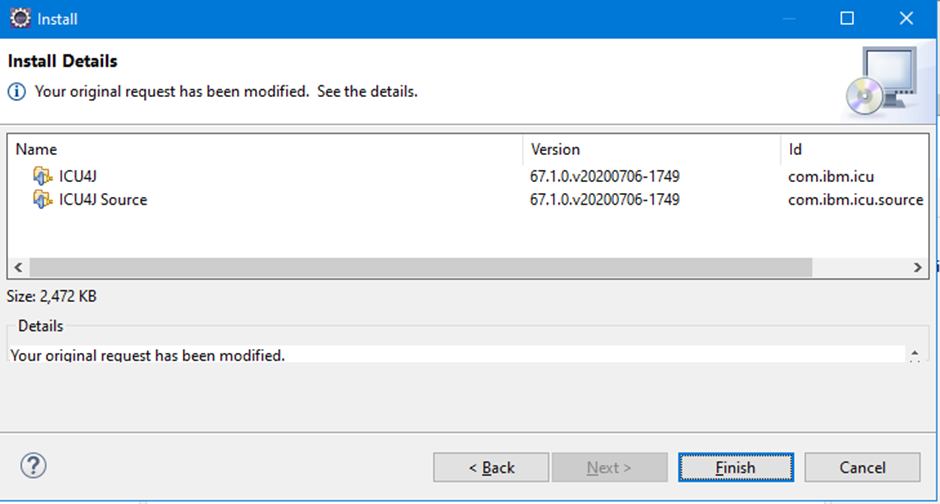


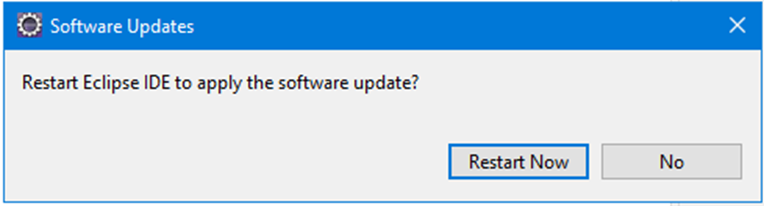
***Update ICU4J by ‘Eclipse Orbit’*** ***plugin***. Select: Help -> Install New Software…. Insert the following repository URL <https://download.eclipse.org/tools/orbit/downloads/2022-09> into ‘Work with:’ field and press Add… -> Add.



Filter for ‘icu4j’, select ‘All Bundles’, click on ‘Next’ and ‘Finish’ and restart Eclipse after installation finishes:







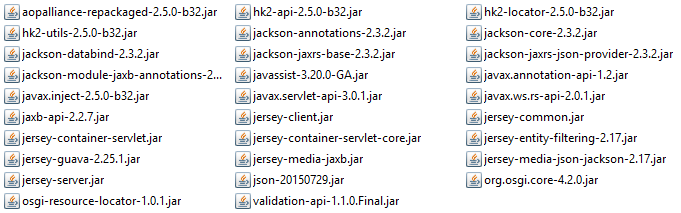
***3. Convert the project to Maven project (optional)***

At this step you can convert your project to Maven project or proceed to the next step if you have all required .jar files (Jersey 2.25.1 and all its dependencies, e.g. Jackson 2.3.2, etc.). You can find all such files in the WEB-INF\lib folder of the RESTfulCalculator project you worked with last week.

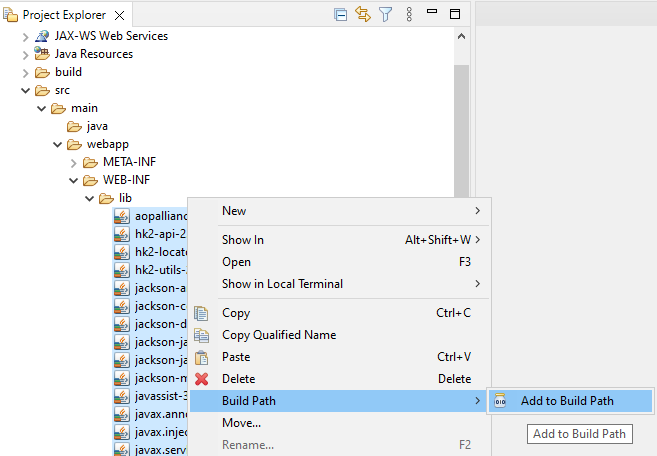
***Maven*** is a build automation tool used for Java projects. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache (thus, you don't have to manually search, download and add the required .jar files to the project). Pom.xml file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins.

***4. Add Jersey REST framework and its dependencies to the project***

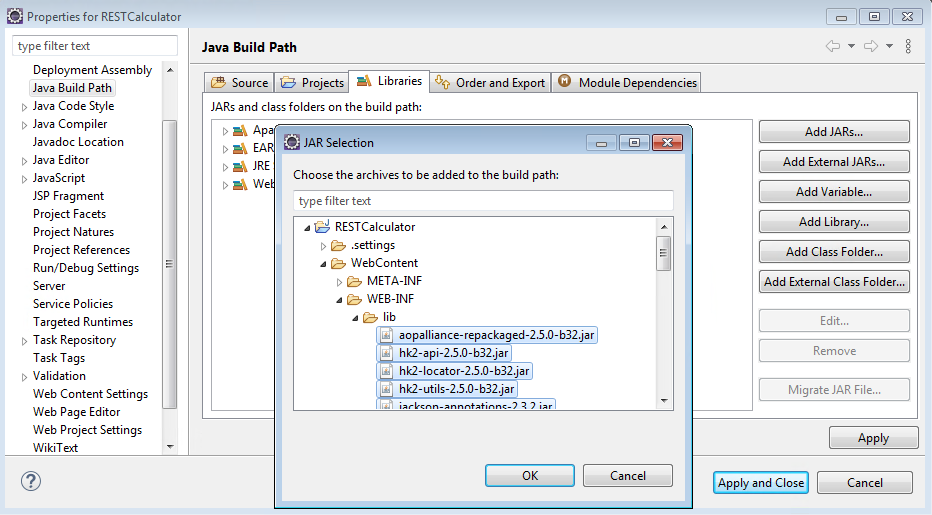
If you skipped Step-3, you need to add the following .jar files into the WebContent\WEB-INF\lib folder of your project and add them into the Java build path.



You can copy/paste these .jar files from the RESTfulCalculator project saved in the eclipse-workspace. Once you copied all necessary .jar files you need to add them to the Build Path by selecting all .jar files and adding them to the build path by right clicking (Build Path -> Add to Build Path)



or by right clicking on the project name -> Build Path -> Configure Build Path… -> Libraries -> Add JARs).



If on Step 3 you decided to opt for Maven project, insert the following dependencies into the pom.xml file

...

<dependencies>

<dependency>

<groupId>org.glassfish.jersey.containers</groupId>

<artifactId>jersey-container-servlet-core</artifactId>

<version>2.25.1</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-jaxb</artifactId>

<version>2.25.1</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-json-jackson</artifactId>

<version>2.25.1</version>

</dependency>

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20150729</version>

</dependency>

<dependency>

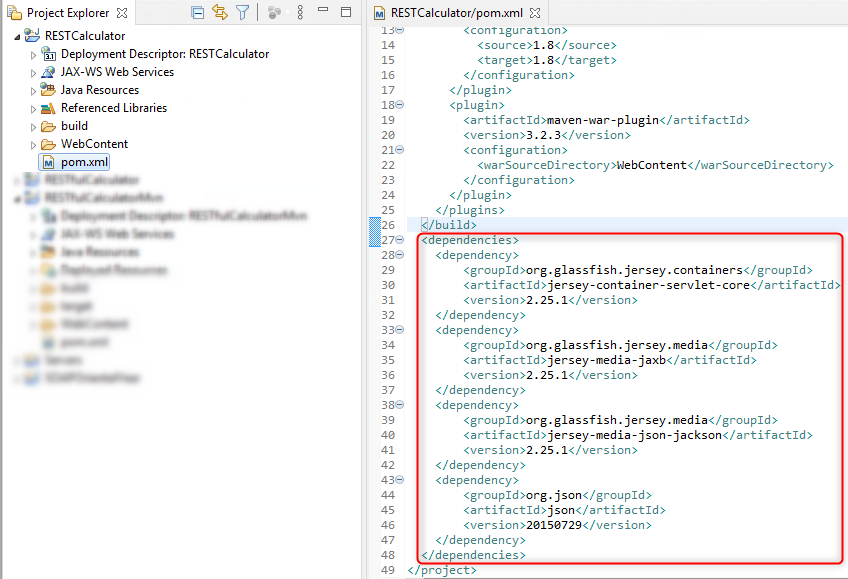
<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.2.7</version>

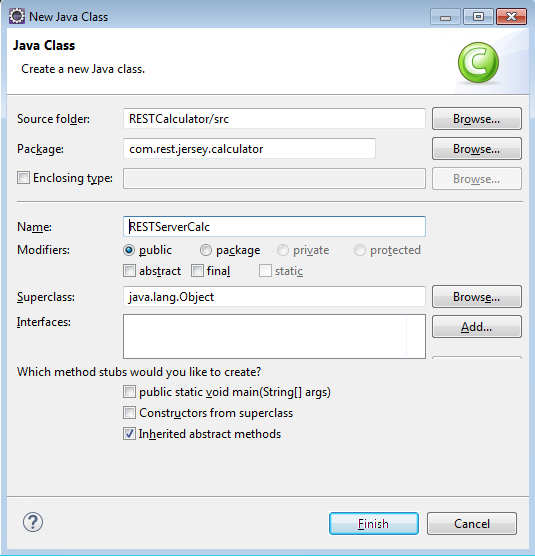
</dependency>

</dependencies>



***5. Add a source code of the RESTful Web Service***

In the Project Explorer go to Java Resources and right click on src -> New -> Class.   
Specify Class name (e.g. RESTServerCalc) and Package (e.g. com.rest.jersey.calculator).



Insert your code, e.g. (notice the imported classes):

**Note:** Do not copy-paste the program code from the labsheet opened in Google Docs (some characters will be automatically filtered by Google; hence, your code will be compiled with errors)!!! Instead, download the labsheet on your PC as .docx and open it in Microsoft Word, from where you can safely copy-paste the code as it is.

**package** com.rest.jersey.calculator;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.PathParam;

**import** javax.ws.rs.QueryParam;

**import** javax.ws.rs.core.Response;

**import** org.json.JSONException;

**import** org.json.JSONObject;

@Path("/calc") //The root path

@Consumes(MediaType.***APPLICATION\_JSON***) //Input data type

**public** **class** RESTServerCalc {

@GET //Invocation method

@Produces(MediaType.***APPLICATION\_JSON***) //Output data type

//Invocation: http://localhost:8080/RESTCalculator/calc?a=6&b=7&op=add

**public** Response Calculator(@QueryParam("a") **double** a, @QueryParam("b") **double** b, @QueryParam("op") String op) **throws** JSONException {

JSONObject json = **new** JSONObject();

**double** c;

**switch**(op) {

**case** "+":

**case** "add":

c = a + b;

**break**;

**case** "-":

**case** "sub":

c = a - b;

**break**;

**case** "\*":

**case** "mul":

c = a \* b;

**break**;

**case** "/":

**case** "div":

**if** (b==0) {

**return** Response.*status*(400).entity("400 Arithmetic exception: division by zero error").build();

}

**else** {c = a / b;};

**break**;

**default**:

**return** Response.*status*(400).entity("400 Invalid Operator").build();

}

json.put("a", a);

json.put("b", b);

json.put("op", op);

json.put("c", c);

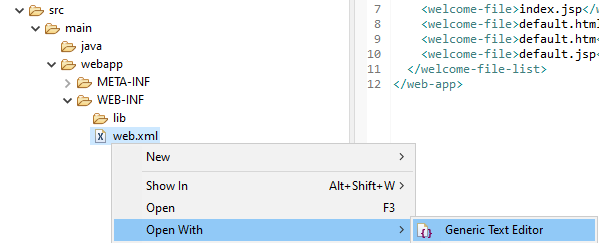
String result = ""+json;

**return** Response.*status*(200).entity(result).build();

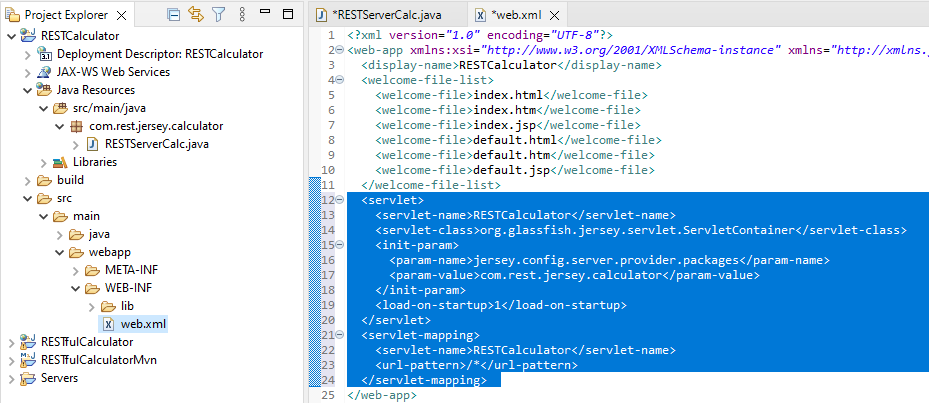
}

}

***6. Double click on the web.xml file and switch to Source view: ***

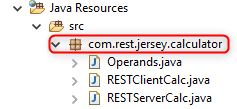
******

***Update the web.xml file by adding the servlet description as follows.***



Specify your <display-name> and <servlet-name>.

Make sure you use a correct package name in the <param-value>



<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* id=*"WebApp\_ID"* version=*"3.0"*>

<display-name>RESTCalculator</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>RESTCalculator</servlet-name>

<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>

<init-param>

<param-name>jersey.config.server.provider.packages</param-name>

<param-value>com.rest.jersey.calculator</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>RESTCalculator</servlet-name>

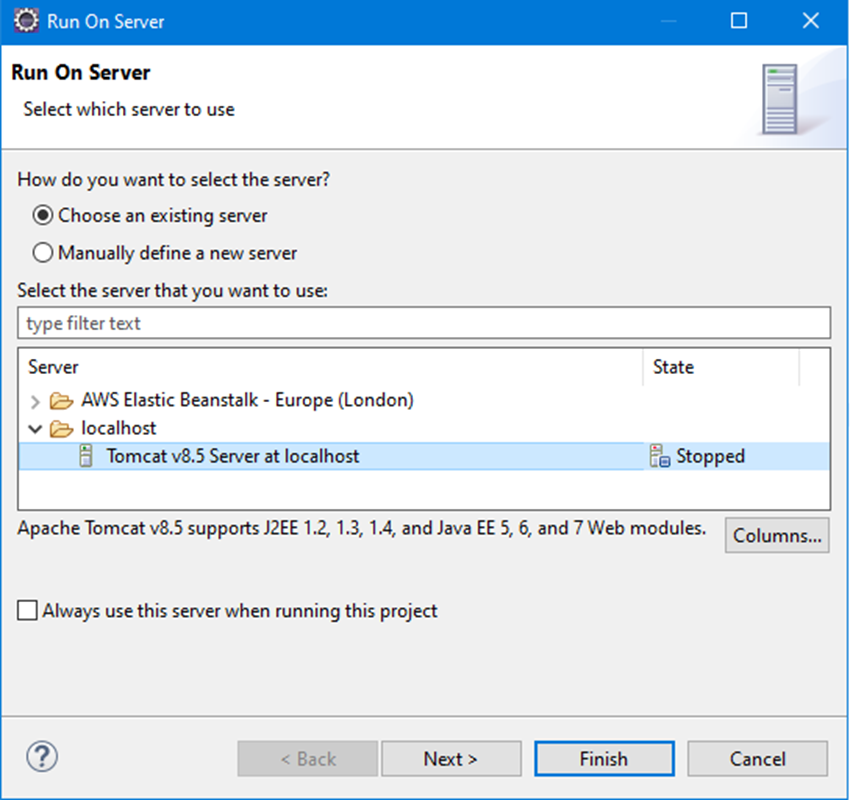
<url-pattern>/\*</url-pattern>

</servlet-mapping>

</web-app>

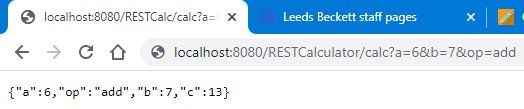
***7. Run your RESTful Web Service***

If you did not convert your project to Maven project, run your Web service by right clicking on the project name -> Run As -> Run on Server.



After the project is deployed, open Google Chrome (IE does not display JSON) and invoke some functions, i.e. http://localhost:8080/RESTCalculator/calc?a=6&b=7&op=add

If everything is right, you will see the response from the service:



—----------------------------------------------------------------------------------

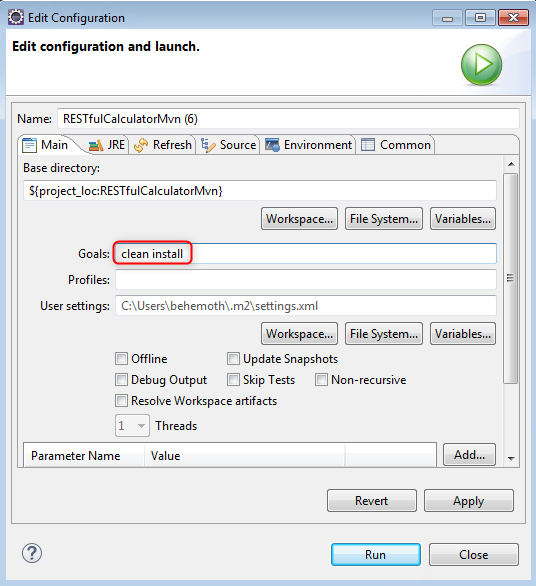
If you converted your project to **Maven project**, perform the following operations:

1. Click on your project in the Project Explorer

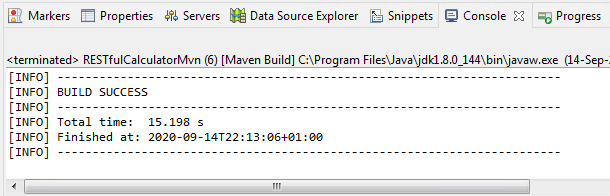
2. Choose Clean in the Project menu

2. Right click on the project name -> Maven -> Update Project

3. Right click on the project name -> Run As -> Maven Build -> Add "clean install" as Goals -> Run



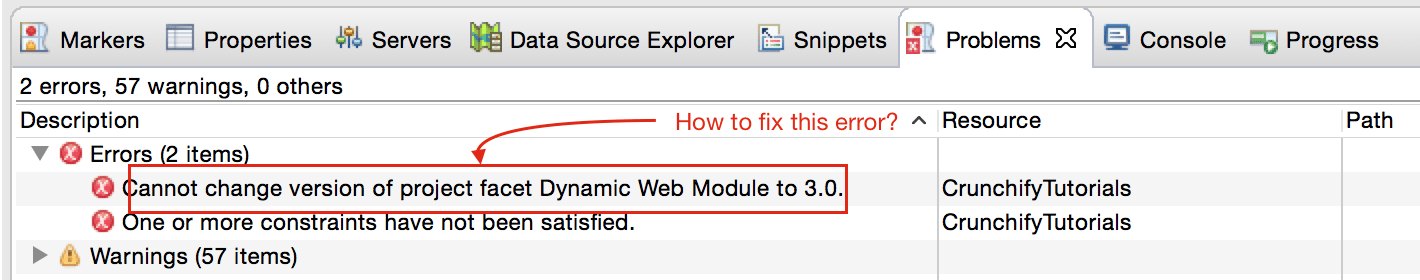
You should see BUILD SUCCESS message



After that you can run your Web service by right clicking on the project name -> Run As -> Run on Server.

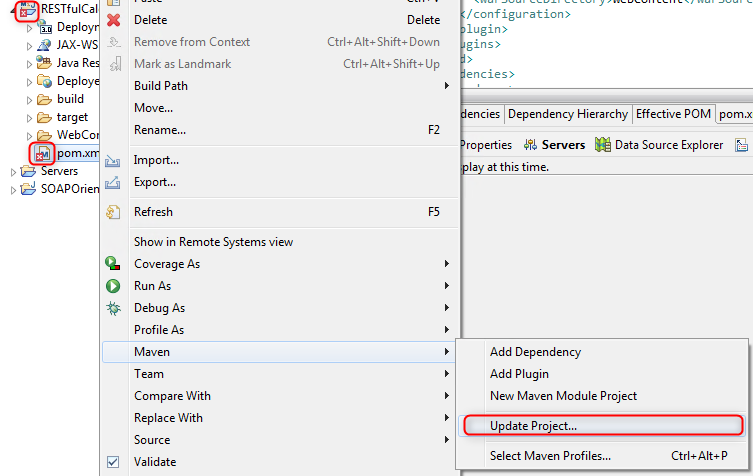
**Possible problems with Maven project**:

1. An error message “Cannot change version of project facet Dynamic Web Module to 3.0” is displayed:



***Solution:*** read here <https://crunchify.com/how-to-fix-cannot-change-version-of-project-facet-dynamic-web-module-to-3-0-error-in-eclipse/>

1. Pom.xml highlights errors



***Solution:*** right click on pom.xml ->Maven -> Update Project

# 

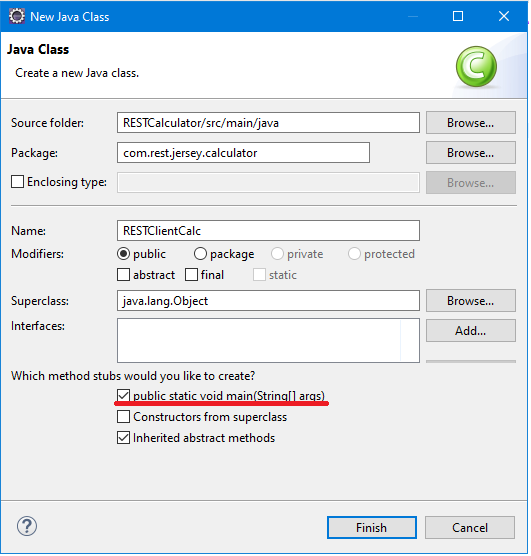
# 

# 

# Step 2: Developing your own Client of a RESTful Web Service

***1. Add a source code of a Client to invoke a RESTful Web Service***

In the Project Explorer go to Java Resources of your RESTful web service and right click on src/main/java -> New -> Class. Specify the Class name for your client (e.g. RESTClientCalc) and the Package (e.g. com.rest.jersey.calculator) and tick a checkbox to create a stub for the main method.



Insert your code, e.g. (notice the imported classes):

**package** com.rest.jersey.calculator;

**import** java.net.URI;

**import** javax.ws.rs.client.Client;

**import** javax.ws.rs.client.ClientBuilder;

**import** javax.ws.rs.client.WebTarget;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.core.UriBuilder;

**import** javax.ws.rs.client.Entity;

**import** org.glassfish.jersey.client.ClientConfig;

**public** **class** RESTClientCalc {

**private** **static** **final** String ***webServiceURI*** = "http://localhost:8080/RESTCalculator";

**public** **static** **void** main(String[] args) {

ClientConfig clientConfig = **new** ClientConfig();

Client client = ClientBuilder.*newClient*(clientConfig);

URI serviceURI = UriBuilder.*fromUri*(***webServiceURI***).build();

WebTarget webTarget = client.target(serviceURI);

System.***out***.println("\nUsing GET QUERY parameters to invoke a RESTful service returning JSON output:");

System.***out***.println(webTarget.path("calc")

.queryParam("a", "74").queryParam("b", "12")

.queryParam("op", "+").request()

.accept(MediaType.***APPLICATION\_JSON***).get(String.**class**));

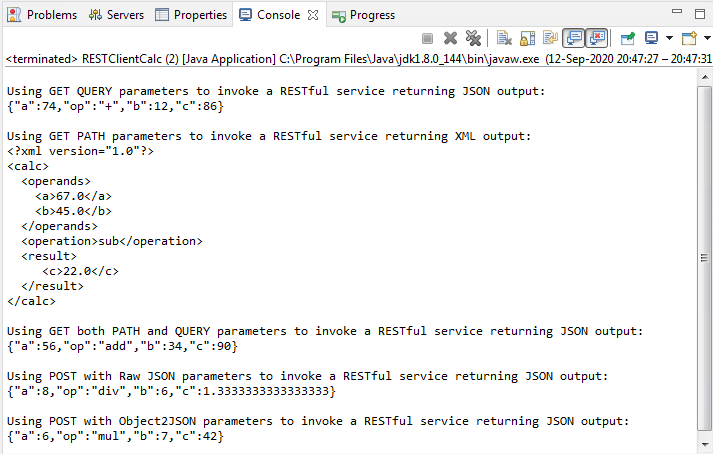
}

}

***2. Execute your Client to test a RESTful Web Service***

Run your Client as Java application (Right click on the RESTClientCalc .java -> Run As -> java Application).

Check the console output. It should look like this:



# 

# Step 3: Deploying your RESTful Web Service on AWS Elastic Beanstalk

Deploy your RESTful Web Service on AWS Elastic Beanstalk following an instruction from the previous lab.

# Step 4: Testing your RESTful Web Service deployed on AWS Elastic Beanstalk

Test your RESTful Web Service deployed on AWS Elastic Beanstalk using SoapUI and a desktop client by following an instruction from the previous lab.

**Note:** Do not forget to provide a correct URL pointing to your RESTful Java Web Service deployed at AWS Elastic Beanstalk instead of ‘localhost:8080’ used to test its local deployment



# Step 5: Removing AWS Elastic Beanstalk Resources

Remove AWS Elastic Beanstalk resources after use.