Exercise 1

Create and consume a REST Service using Apache CXF and JAX-RS

Prior Knowledge
Basic understanding HTTP verbs, REST architecture
Some Java coding skill

Objectives
Understand what it takes to create REST services. Understand the REST model.
Interact with a REST service using simple web clients in Chrome, on the
command line, and in Java.
See how Maven and Tomcat can be used.

Software Requirements
(see separate document for installation of these)

- Java Development Kit 7
- Apache Maven 3.0.4 or later
- Eclipse Juno (4.2 SR1) or later – Java Development IDE
- Tomcat 7.0.30 or later
- curl
- Google Chrome plus Chrome Advanced REST extension

Step 1. **Create a new project using Maven**

Maven is a very powerful (and somewhat arcane) build tool. We are going to use Maven to create and build our project.

Maven has the ability to create new projects using "archetypes".

a. First create a directory structure to store the exercise projects in.
(e.g. ~/oxsoa/, c:\oxsoa\, etc)

From now on this will be referred to as $oxsoa

b. Open a command line and change to that directory

c. Test that you have maven properly installed. Execute
```
mvn -v
```

You should see something similar to this (dependent on your machine, JVM, etc)
```
Apache Maven 3.0.4 (r1232337; 2012-01-17 08:44:56+0000)
Maven home: /Users/paul/Apps/apache-maven-3.0.4
Java version: 1.6.0_35, vendor: Apple Inc.
Java home:
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x", version: "10.8.1", arch: "x86_64", family: "mac"
```

d. Use Maven to create a sample project:
Execute

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

This will prompt you for some choices:
```
Choose archetype:
1: remote -> org.apache.cxf.archetype:cxf-jaxrs-service
(Simple CXF JAX-RS webapp service using Spring
configuration)
2: remote -> org.apache.cxf.archetype:cxf-jaxws-javafirst
(Creates a project for developing a Web service starting
from Java code)
Choose a number or apply filter (format:
[groupId:]artifactId, case sensitive contains): :
```

Select 1 (*type '1' and hit enter!*)

Now it will ask which CXF **version** to use. The default is the latest (at the time of writing 2.7.0). *Just hit enter.*

Now it asks for a default **groupid**.

```
Define value for property 'groupId': :
```

This is a namespace. When creating this lab I chose *me.freo.rest,* so that is what you will see in screenshots, etc. Please stick to this, because the code I'm providing you with uses that.

```
Define value for property 'artifactId': :
```

This will be the name of the WAR and the overall maven artifact created. We are going to create an Order service, so type *OrderService***.**

```
Define value for property 'version':  1.0-SNAPSHOT: :
```

Change this to 1.0 and hit Enter.

```
Define value for property 'package':
com.mycom.restservice: :
```

This will default to the same namespace you chose for the groupid. That should be fine, so hit *Enter* to accept.

It will then ask you to confirm these settings. Hit *Enter* and it will go and generate the code.

You should see plenty of output explaining what is happening, and also a line showing where the resulting code was placed, e.g.:

```
[INFO] project created from Archetype in dir:
/Users/paul/oxsoa/OrderService
```

This will have created a set of code and a tree structure for you. If you are on Linux you can use the nice **tree** command to show this:

```
$ tree
.
└── OrderService
    ├── pom.xml
    └── src
        ├── main
        │   ├── java
        │   │   └── me
        │   │       └── freo
        │   │           └── rest
        │   │               ├── HelloWorld.java
        │   │               └── JsonBean.java
        │   └── webapp
        │       └── WEB-INF
        │           ├── beans.xml
        │           └── web.xml
        └── test
            └── java
                └── me
                    └── freo
                        └── rest
                            └── HelloWorldIT.java

14 directories, 6 files
```

e. You can now build this code:

```
cd ~/oxsoa/OrderService [on Linux/Mac]
cd ~\oxsoa\OrderService [on Windows]
mvn clean install
```

The first time this is run this will download a lot of stuff from the central maven repositories on the web. Depending how fast the network is, maybe a coffee is in order. You will need an active internet connection for this to work.

This will build **and test** the sample code. Its pretty cool. It actually starts an embedded Tomcat to run the service and call unit tests against it.

f. Assuming your build worked just fine, you now have a WAR file (Web Application aRchive) that you can deploy in Tomcat. Check that there is a file:

~/oxsoa/OrderService/target/OrderService-1.0.war

g. Install Tomcat

Just unzip the Tomcat zipfile into your home directory and you should get a new directory ~/apache-tomcat-7.0.33.

h. Install your webapp

cp ~/oxsoa/OrderService/target/OrderService-1.0.war ~/apache-tomcat-7.0.33/webapps

i. Start Tomcat so it runs on the command line (so you can see the logs)

From the tomcat directory:
```
bin/catalina.sh run
```

j. Try your REST service

Browse
http://localhost:8080/OrderService-1.0/hello/echo/paul

You should see "paul".

You can also try this as a command line:

```
curl http://localhost:8080/OrderService-1.0/hello/echo/paul -v
```

You should see something like:
```
* About to connect() to localhost port 8080 (#0)
*   Trying 127.0.0.1...
* connected
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /OrderService-1.0/hello/echo/paul HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0)
libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Date: Fri, 07 Dec 2012 14:41:20 GMT
< Content-Type: text/plain
< Content-Length: 4
<
* Connection #0 to host localhost left intact
paul* Closing connection #0
```

f. You can also build the Eclipse project for this too:
```
mvn eclipse:eclipse
```

This creates a project file that you can import into Eclipse with the right classpath, settings, etc.

g. Before you import the project, you do need to let Eclipse know where your Maven is installed.

You can do this manually in Eclipse by adding the M2_REPO variable, but there is also a command line tool for this:

```
mvn -Declipse.workspace={path to eclipse workspace}
    eclipse:add-maven-repo
```

On my Mac, the path to my eclipse workspace is
`/Users/paul/Documents/workspace`

h. Once you have done this start (or restart) Eclipse, and then you can import the new project.

To do this, in Eclipse:

File -> Import -> General/Existing Projects Into Workspace->[Choose the directory where OrderService is]->Finish

Now you should have the project installed in your Eclipse and be able to edit and build it. **Take a look at the sample hello service.**

Now we are ready to build our own Rest Service.

Step 2. Creating the OrderService

Rather than spending a lot of time writing Java, which is not the main point of this exercise, you should focus on the REST and HTTP aspects of this. I have ready written a set of Java classes that:
  1) Implement a simple "Order Scenario"
  2) Serialize and Deserialize as JSON

I have also written a test case that validates a service interface.
Your aim is to create a Java Service that utilizes the existing code, and is correctly annotated using JAX-RS annotations so that it meets the test case.

Here is a rough set of documentation that explains the service interface.

Method	URI template	Description	Supported encoding
GET	./orders	Get a list of href links to available orders If no orders are on the system, return an empty list.	Produces application/json
GET	/orders/{id}	Get back a representation of order with identifier id. If no such order is yet in the system, returns HTTP Not Found If the order previously existed but has been deleted, returns HTTP Gone	Produces application/json
POST	/orders	Passes a representation of the order and create a new entry in the order database. On success returns HTTP 201 Created and an HTTP Location header containing the URI of the resulting order	Consumes application/json
PUT	/orders/{id}	Updates an existing order On success return HTTP 200 OK If no such order is yet in the system, returns HTTP 404 Not Found If the order previously existed but has been deleted, returns HTTP 410 Gone	Consumes application/json
DELETE	/orders/{id}	Marks an order as deleted Returns HTTP 200 OK on success If no such order is yet in the system, returns HTTP Not Found If the order previously existed but has been deleted, returns HTTP Gone	No body content

a) In the directory ~/oxsoa/OrderService, unzip the file jaxrs-service-code.zip

You can find the file here: https://raw.github.com/pzfreo/ox-soa/master/lab-exercises/source/jaxrs-service-code.zip

Try another `mvn clean install`. You will have a compile error because you don't have the correct dependencies. Edit the pom.xml to include

```
<dependency>
     <groupId>org.json</groupId>
     <artifactId>json</artifactId>
     <version>20090211</version>
</dependency>
```

as an XML sub-element of
`/project/dependencies`

b) Now `mvn clean install` again. This time it should build, but you will have an Integration Test failure.

c) Now add a new Java class to your codebase:
`me.freo.rest.OrderService`

You can start with this snippet of code which will give you an Order object to work with:
```
package me.freo.rest;
public class OrderService {

  Order orderSingleton = null;
  public OrderService() {
    orderSingleton = Order.getInstance()
  }
}
```

d) You also need to modify the `src/main/webapp/WEB-INF/beans.xml` file to define the OrderService file to CXF as a JAX-RS service.

Under the line:
```
<bean class="me.freo.rest.HelloWorld" />
```

Add a new entry:
```
<bean class="me.freo.rest.OrderService" />
```

e) Now you can incrementally add the correct methods and annotations for get/post/put/delete until the test case is met.

f) ONCE YOU GET YOUR BUILD WORKING:

Re-install the webapp into Tomcat. (It should hot deploy if you've left Tomcat running)

g) Now you can test it in Chrome, using the Advanced REST Test Client. Start Chrome and get up a blank/empty page (Command-N or Ctrl-N)

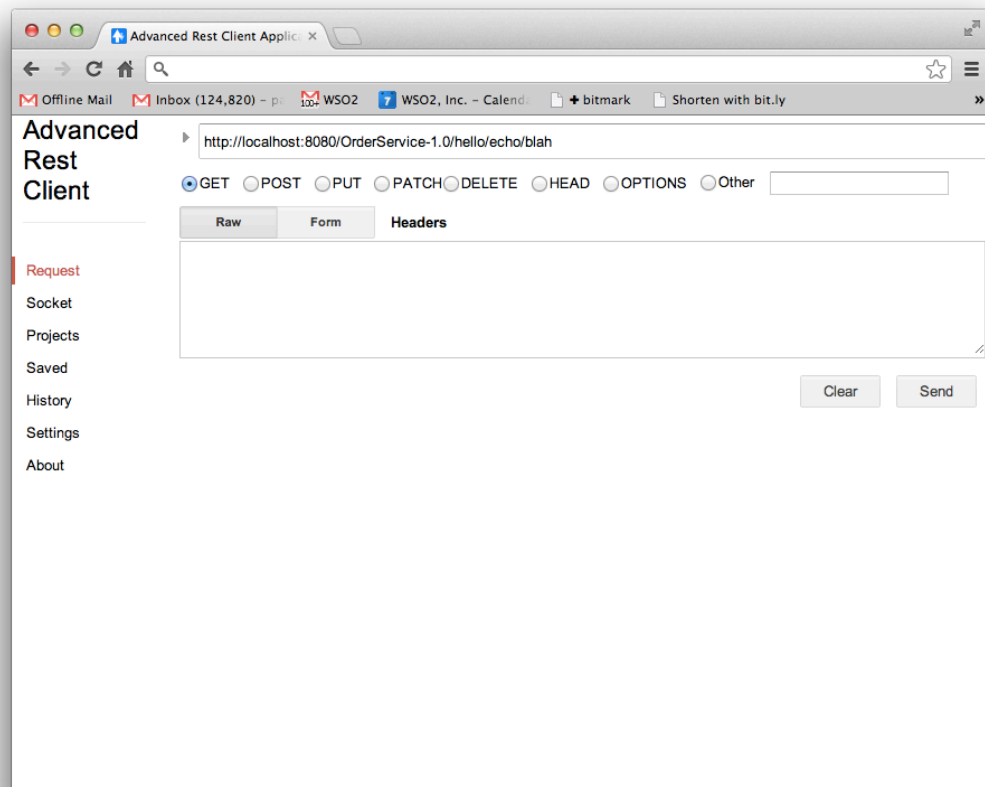h) Click on the Apps link at the bottom of the page:
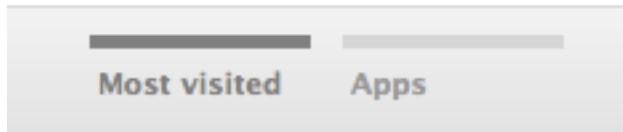
You should now see the following icon:

Advanced REST client

i) Click on that. You should see a screen like this:

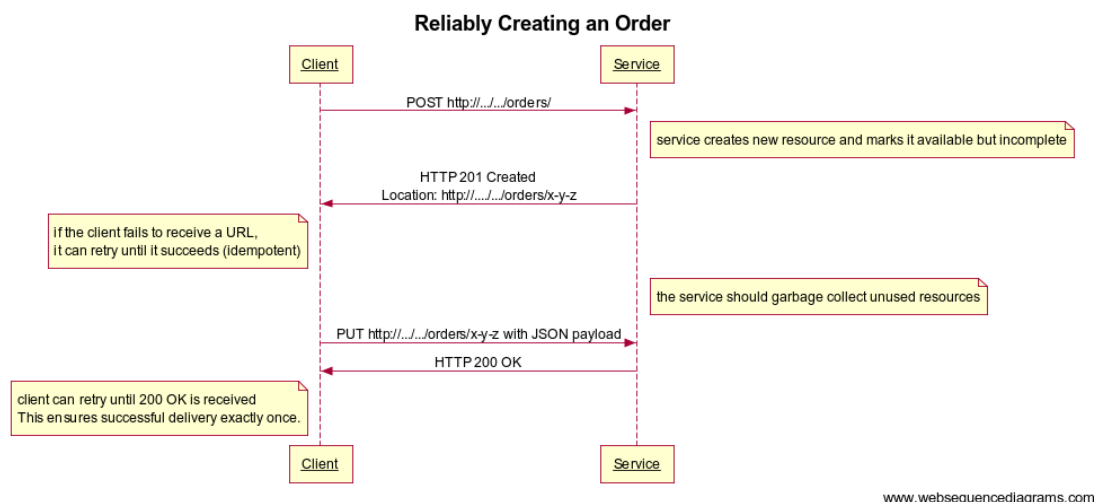By now you should understand the REST patterns and URLs well enough to be able to test our your app. It should be available at: http://localhost:8080/OrderService-1.0/orders/

j) Also try interacting with it using curl commands. See if you can automate posting an order.

EXTENSION

The model we have built does not support reliability. It is impossible to know if you have successfully created a new order. To fix this, we need to implement a different pattern. In this pattern, the POST is empty, and returns a location, and only when the POST has successfully returned a Location and the client has the location, do we PUT an order into the existing location.

Here is a sequence diagram showing this model.

Reliably Creating an Order

Client   Service

POST http://.../.../orders/

service creates new resource and marks it available but incomplete

HTTP 201 Created
Location: http://.../.../orders/x-y-z

if the client fails to receive a URL,
it can retry until it succeeds (idempotent)

the service should garbage collect unused resources

PUT http://.../.../orders/x-y-z with JSON payload
HTTP 200 OK

client can retry until 200 OK is received
This ensures successful delivery exactly once.

Client   Service

www.websequencediagrams.com

Implement this model by changing the code.