Assignment : Implement a RESTful web service

Step I : Tomcat and Web Applications

Go to the Apache Tomcat web site (https://tomcat.apache.org/download-70.cgi) and download the version of Tomcat that is most appropriate for your operating system.
In a nutshell, Tomcat is an example of a class of applications called "web containers" (others here: http://en.wikipedia.org/wiki/Web_container) which, when run, listen to, intercept and process http-based requests.

[Background :
Any computer connected to (an) internet, a network of computers that uses the HTTP protocols, has an "IP" address that identifies it, like 98.174.196.243 or 127.0.0.1[1]. IP addresses are usually mapped (using a service called "DNS") to logical names which are easier to handle, like "www.google.com" or "localhost". When a computer is on the internet, it can communicate sending and receiving messages to processes listening on a "port". Port 80 is usually dedicated to web servers, that is, servers which handle www-related requests.
So, when your browser connects e.g. to "www.asu.edu", it sends a GET request to a web server, active on a machine at the address (as I'm writing) 129.219.10.58[2], listening on the port 80, to retrieve the content at the "root" address. That content will be an HTML document: the browser will receive a copy and render it on the screen.

Our goal is to create our own (web) server, so that it can handle requests on behalf of our client applications – browsers, general purpose tools like SoapUI or our own programs alike].
Tomcat is a web container – it creates a process that intercepts requests directed to the machine it is installed on. It is called a "container" because it "contains" the actual applications that will handle the requests. Tomcat itself will act as a broker between the client and the actual application. We will see how in a moment.]

Install tomcat and launch it using the startup.bat/.sh script in the "bin" directory. Depending on the OS, there may be other ways to launch it. To check that tomcat is active, open a web browser and connect to "http://localhost:8080". Localhost is an IP address, it's an alias for "this machine". Of course this is not a "public" address. The ":" separates the address from the port. Even if "80" is the default[3] port for web servers, tomcat runs by default on "8080" until reconfigured.
A GET for the "/" is handled by tomcat itself, which returns a web page that presents the current state of the server. You should see a web page titled "Apache Tomcat". Consider that this page comes from **your** machine. Other URLs, like http://localhost:8080/fhir/Patient or http:localhost:8080/cts2/codesystems, will have to be mapped to and handled by our applications
Ensure that tomcat is up and running before moving to the next step.

1 You can check what is the IP address of your computer e.g. here https://www.whatismyip.com/
2 Try it here : http://tracert.com/resolver
3 If you don't specify a port, 80 is taken for granted. That is, www.google.com is the same as www.google.com:80

Step II : Creating and deploying a Web application

To create the shell of our application, we will use a Maven archetype. A Maven archetype is a parametric "proto-project" that can be instantiated to create actual maven projects. There are dozens of archetypes – you can even create your own – but here will use the "tomcat webapp" archtetype (http://tomcat.apache.org/maven-plugin-trunk/archetype.html).
Go to your usual project folder and run:

mvn **archetype:generate** -DarchetypeGroupId=**org.apache.tomcat.maven**
-DarchetypeArtifactId=**tomcat-maven-archetype** -DarchetypeVersion=**2.2**

the command "archetype:generate" does exactly what it shows: it creates a maven project from an archetype. The archetype's GAV are passed as parameters. During the creation process, you will be asked to enter values for the archetype's parameters. In this case, you will just have to specify the GAV of your project and the main package for your classes. After a last confirmation ("Y"), you will have a fully working maven modular project that can be imported in Eclipse / IntelliJ / NetBeans or any other IDE.

The modules are as follows (replace "Test" with the name you have chosen for your project)

Test-api : this module will contain the service interfaces
Test-impl : this module will contain the actual implementation
Test-webapp : this modules packages the api & impl into a "WAR" file, a packge that can be deployed in tomcat. A "WAR" is the web- counterpart of a "JAR".
Test-webapp-exec : this module contains a standalone instance of tomcat that can be used to run the application (in case you don't have tomcat installed and you don't need other web applications)
Test-webapp-it : the test module.

We will focus mainly on the first three, since we will deploy the WAR in the main tomcat installation.

1) Run a full "`mvn clean install`" from the root folder, to check that everything compiles, installs and runs successfully. You should find a ".war" file in the Test-webapp/target folder. This is the packaged web application

2) Now we'll make maven deploy the application in the main tomcat installation. Open the Test-webapp/pom.xml, you should find something like:

```
<build>
  <pluginManagement>
    <plugins>
      <!-- original before change -->
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat6-maven-plugin</artifactId>
        <configuration>
          <port>8080</port>
          <path>/</path>
        </configuration>
      </plugin>
      <!-- edited -->
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <configuration>
          <port>8080</port>
          <path>/[PATH]</path>
          <username>[user]</username>
          <password>[pass]</password>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

There are actually two plugins: one for tomcat v6 and one for tomcat v7. Before you make any changes, both point to the "embedded" version of tomcat that would be started by the self-contained Test-webapp-exec configuration. Assuming that you have downloaded Tomcat 7.x, we will change the pom file similar to the code snippet above.

- The <port> has to be the same port Tomcat is listening to. By default, for a newly downloaded installation, it's 8080
- The <username> and <password> need to reflect the tomcat credentials.

  To create a user in tomcat:
  - Open the file [TOMCAT_INSTALLATION_DIR]/conf/tomcat-users.xml
  - Ensure the following content is present

    ```
    <role rolename="admin-gui"/>
    <role rolename="manager-gui"/>
    <role rolename="manager-script"/>
     <role rolename="manager-jmx"/>
    <role rolename="manager-status"/>
    <role rolename="tomcat"/>
    <role rolename="admin"/>
    <user username="[user]" password="[pass]"
    roles="admin,tomcat,admin-gui,manager-gui,manager-script,manager-jmx,manager-status"/>
    ```
  - Restart tomcat
- You can choose whatever you want as the "PATH" of the application.

Now, from inside the Test-webapp folder, you can run "`mvn tomcat7:deploy`", it will take the WAR package and deploy it in Tomcat, provided that tomcat is running and the credentials are configured correctly. You will use "`mvn tomcat7:redeploy`" if the application was already deployed and needs to be updated.

Step III : Exploring a Web Application

1) Thc HTML content

As a test, open the browser and try connecting to your application PATH within the scope of Tomcat. For example, if you had picked "myFirstTestApp" as the PATH, you would connect to:

http://localhost:8080/myFirstTestApp

Tomcat will realize that /myFirstTestApp is a registered PATH and will delegate the handling of the request to our application. By default, a web application behaves as a... web application, so a GET request to access the root "/" of the application will, by default, look for an "index.html" - a web page that acts a the gateway of our application (remember that this technology was created for multimedia hyper-texts!). In fact, the Test-webapp module contains a src/main/webapp folder which, not surprisingly, contains an index.html. That resource is included in the WAR and retrieved by the web server. If you had sub-folders and more HTML resources, e.g. a src/main/webapp/myFolder/someExtraContent.html, a call to http://localhost:8080/myFirstTestApp/myFolder/someExtraContent.html would be associated to that file.

The default "main" page of our application actually contains a way to interact with a "hello world" REST service that is also part of the application. Try it yourself, for example with your name, and see the result.

2) The Service

Now we will try to access the service directly, without the help of the HTML controls.
Try to connect to (GET) the address:

http://localhost:8080/myFirstTestApp/restServices/TestServices/HelloService/sayHello/abc

As all URLs, it is composed by several parts, in detail:

1. http://localhost:8080 connects to Tomcat on our machine
2. /myFirstTestApp routes the request to our application (Tomcat may have more than one!)
3. /restServices our application's REST services are located under this sub-branch.
   Tomcat will know that this is not a web page, but a REST call
4. /TestServices "Test" is taken from the project name. It is a logical grouper
5. /HelloService corresponds to the name of an interface (collection of operations)
6. /sayHello is the operation we are trying to invoke.
   It corresponds to an operation:
   ```
   String sayHello( String in ) { return "Hello " + in; }
   ```
7. /abc is the value of the input argument passed to the operation

The browser will take the output of the operation (a String "hello abc"). Since the result is raw and is not wrapped in HTML (unlike the home page), most browsers will just display it as plain text.

Now that we understand the functionality from a client perspective, we need to find where and how this behavior is defined.

3) The interface
Go the Test-api module and look for a class HelloService inside the src/main/java folder.
You will see this code:

```java
@Path( "HelloService" )
public interface HelloService
{
   @Path( "sayHello/{who}" )
   @GET
   @Produces( { MediaType.TEXT_PLAIN } )
   String sayHello( @PathParam( "who" ) String who );
}
```

Java @nnotations are used to bind the Java interface to the REST interface.
The @Path(“HelloService”) corresponds to part (5) in the URL described in 2). It has the same name as the Java interface, but is not strictly necessary.

Each method defined in this interface is linked to its URL fragment counterpart. The @GET + @Path and @PathParam on the operation and its argument serve this purpose (see part (6) and (7) of the URL). When a GET is issued to …/HelloService/sayHello/..., this is the interface that will be resolved.

A quick summary of the admissible annotations can be found, for example, here
http://www.techferry.com/articles/RESTful-web-services-JAX-RS-annotations.html
A more comprehensive description is here[4]:
https://wikis.oracle.com/display/Jersey/Overview+of+JAX-RS+1.0+Features

4) The implementation
The actual implementation of the service can be found in a class DefaultHelloService in the module Test-api-impl.

```java
@Service( "helloService#default" )
public class DefaultHelloService
   implements HelloService
{
   public String sayHello( String who )
   {
      return "Hello " + who;
   }
}
```

Notice that it implements the (Java) interface. The implementation is not associated directly to the REST service, but does so through the interface. It is, however, annotated with @Service (which we'll see in a moment). Its main focus is to provide the implementation for the methods. The implementation can be done by any means, shape and form.

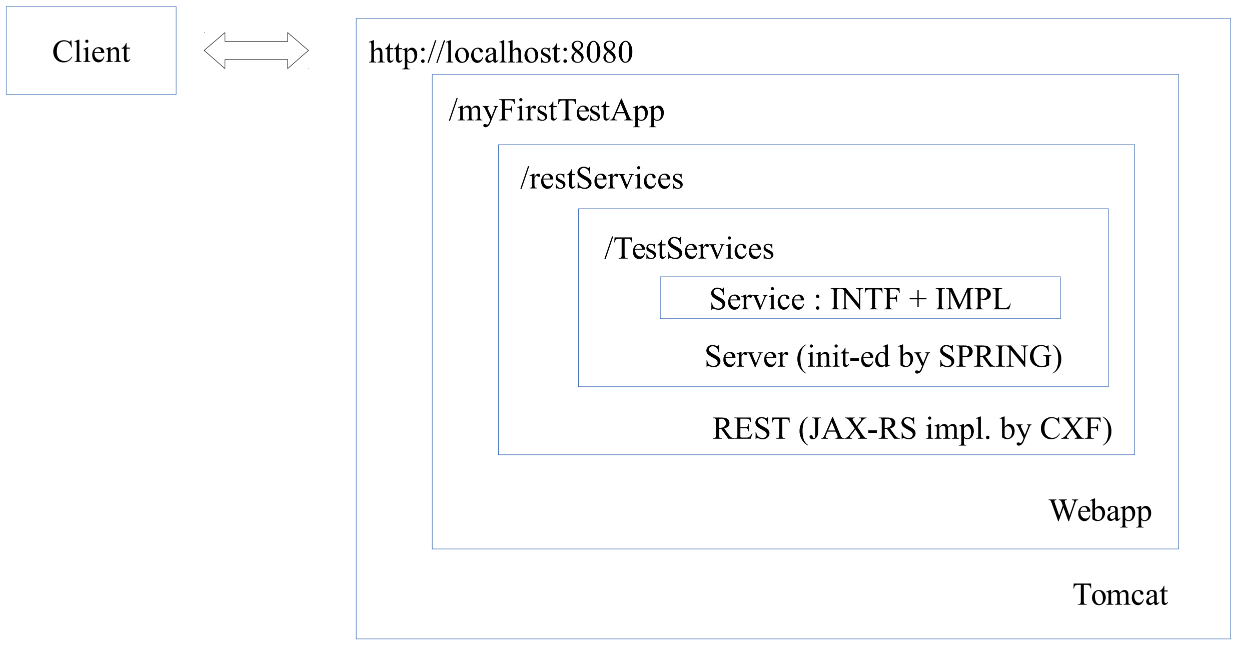
4 This style of creating web services is based on “JAX-RS”, the Java API for web services. Other approaches exist, in Java and (of course) in other platforms.

5) The framework
The last bit to understand is how the interface (and its implementation) eventually get resolved when the GET request is received.

First, see Test-webapp/src/main/resources/webapp/WEB-INF/web.xml
This file is consumed by the container (Tomcat in our case) to configure our application. We see, for example, the file “index.html” included in the <welcome-file-list>, making it the resource that is displayed when a GET for the root “/” of the application's context is received.
More importantly, we see the <servlet-mapping>, which instructs tomcat to consider an URL scoped by /restServices (see item (3)) as a REST service call. Such calls are mapped (and handled) by CXF, an implementation of JAX-RS.

CXF will resolve the REST URLs and invoke the methods on the appropriate Java interfaces and implementing classes. Another actor, SPRING, is responsible for finding, instantiating and initializing the Java objects that CXF will invoke.
This implementation information is in the module Test-api-impl, under src/main/resources/META-INF/spring-context.xml
Inside the file, you will see the definition of a server (Testservices), which corresponds to item (4) in the URL. The capabilities of that server are defined in terms of the “beans” identified with the @Service key which, in turn, link to the interface APIs.

So, in summary:



Step III: Implementing a simple FHIR API

Now we will mimic the very same steps and create a FHIR rest API to read a fhir:Patient's information.

1) Create a FHIRInterface.java interface in your API module. Notice the path, compliant with the FHIR specification. Notice also that this method returns a fhir:Patient in XML format, rather than a plain String.

Notice also that you will have to add a maven dependency to the module. You should try using **your own** implementation of FHIR. CXF relies on JaxB to (de)serialize objects. Otherwise, you will have to provide a custom mechanism for the (de)serialization.

```
@Path( "FHIRService" )
public interface FHIRService
{
   @Path( "Patient/{pid}" )
   @GET
   @Produces( { "application/xml" } )
   Patient getPatient( @PathParam( "pid" ) String pid );
}
```

2) Create the implementation, a FHIRService.java in your IMPL module.
The first mock implementation will simply instantiate a Patient and return it.

```
@Service( "fhirService#default" )
public class DefaultFHIRService
   implements FHIRService
{
   public Patient getPatient( String pid ) {
      Patient pat = new Patient();
      // assign id and some other fields

      return pat;
   }
}
```

BONUS : Retrieve the patient from a DB / File system / repository

3) Register the service

Add the bean @Service ID to the spring-context file:

```
<jaxrs:serviceBeans>
  <ref bean="helloService#default"/>
  <ref bean="fhirService#default" />
</jaxrs:serviceBeans>
```

Step IV : Try the service

After reinstalling (mvn clean install) and redeploying (mvn tomcat7:redeploy) the application, now we are ready to test it.

1) With a browser:
Open a browser window and go to:

http://localhost:8080/myFirstTestApp/restServices/testServices/FHIRService/Patient/p123

Adjust the URL according to your configuration. You should see the XML that corresponds to the object that you created/retrieved in the implementation of the service.

2) With SOAP-UI
Open Soap-UI (see the lecture slides for the download link) and create a new REST project.
This time we will import the WADL that describes all the available services. The WADL is available at

http://localhost:8080/myFirstTestApp/restServices/testServices?_wadl

You will see two possible requests, one for the original "hello world", one for the FHIR call we have just implemented. Try either with different parameters.

3) From Java code
Look at the unit test in Test-api-impl/src/test/java/.../TestDefaultHelloService.java
Here we can see two interesting aspects: how to launch tomcat on the fly and, more importantly, how to construct a WS client using ONLY the API module.

As follows. We need the base URL where the service can be found, which we have seen is composite.
Then we need the (Java) interface, which we know carries all the local @Path information.
Finally, we need the JAXRSClientFactory, which returns a REST client ready to be used.

```java
String endpoint = "http://localhost:8080";
String path = "/hello";
String rest = "/restServices";
String server = "/testServices/";

@Test
public void testFHIR()
{
   FHIRService service =
      JAXRSClientFactory.create(
         endpoint + path + rest + server, FHIRService.class );
   String pid = "foo";
   Patient p = service.getPatient( pid );
     ...
}
```