RPC entre votre application Android et votre serveur

(from http://avilyne.com/?p=105)

This part will describe simple client-server communications between an Android app and a web service using REST techniques, particularly GET and POST. For this demo, we will be sending and receiving data for a simple Java class called DimmySwitch, which includes a name, an address and a dimmy value. The data stored in an instance of this class will be transmitted from our web service in JSON format.

An important requirement is that your Android device needs to be on the same network as your Tomcat server. The simplest way to do this would be to make sure you are running your emulator on the same computer that is running Tomcat.

About REST

REST – "Representational State Transfer" is a technique that makes use of standard web protocols for the implementation of a web service. A RESTful web service uses the standard HTTP GET PUT DELETE and POST actions to submit, retrieve or modify server-side data.

Commonly, in a REST web service, the standard HTTP actions are used as follows:

- GET – retrieves or queries for data, usually using passed criteria
- PUT – creates a new entry, record
- DELETE – removes a resource
- POST- updates a resource or creates a resource

The data that is transmitted uses standard MIME types, including images, video, text, html, XML and JSON.

A key feature of a REST service is its use of URI paths as query parameters. For example, a fictional web service for a book library might return a list of Civil War history books with this URI:
http://librarywebservice.com/books/history/CivilWar
Or, for magazines about tennis:
http://librarywebservice.com/magazines/sports/tennis
These are the absolute basics. REST has become a very popular replacement to SOAP for the development of web services.

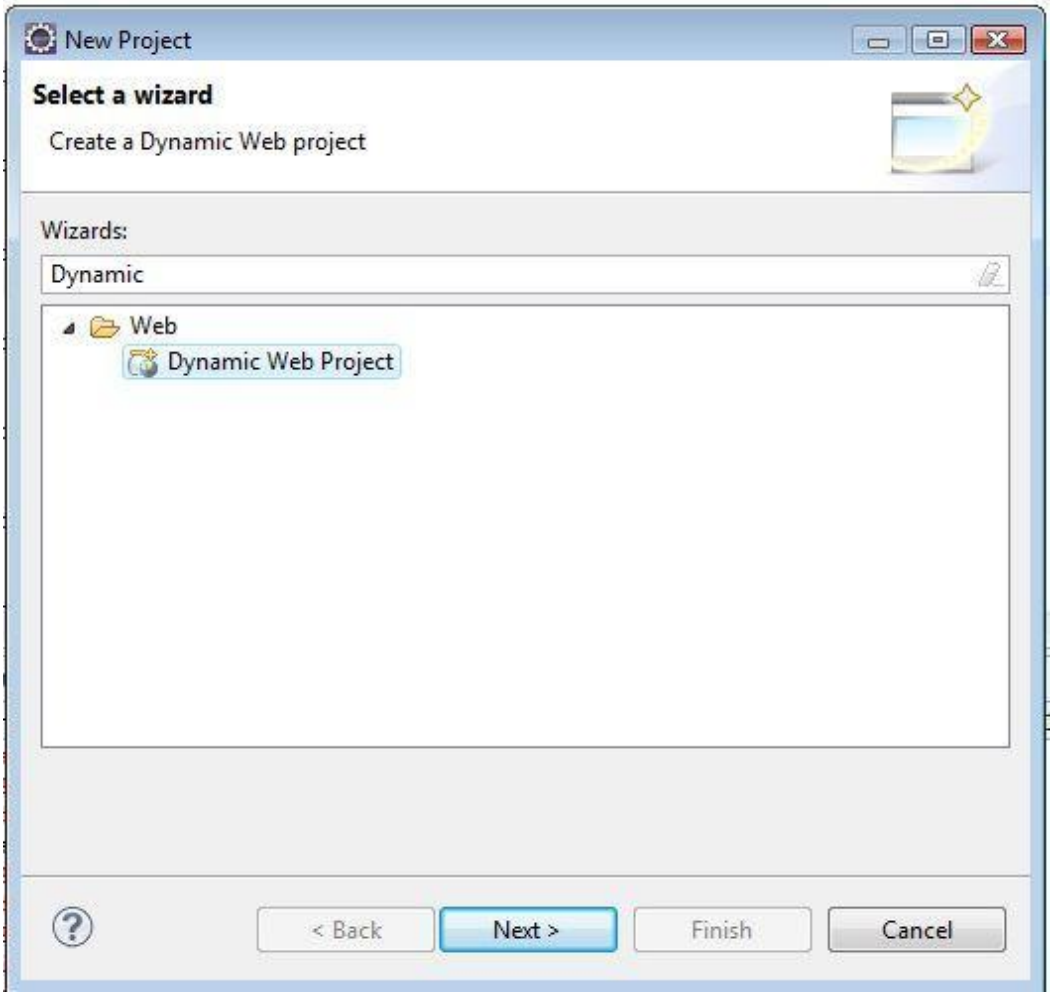
Download the Jersey Jars

For the web server, this demo will use Tomcat, and will make use of a reference implementation of the JSR 311 Java library, otherwise known as “jersey”. The Jersey API can significantly streamline the development of a RESTful web service, and much of its ease comes from the use of annotations.

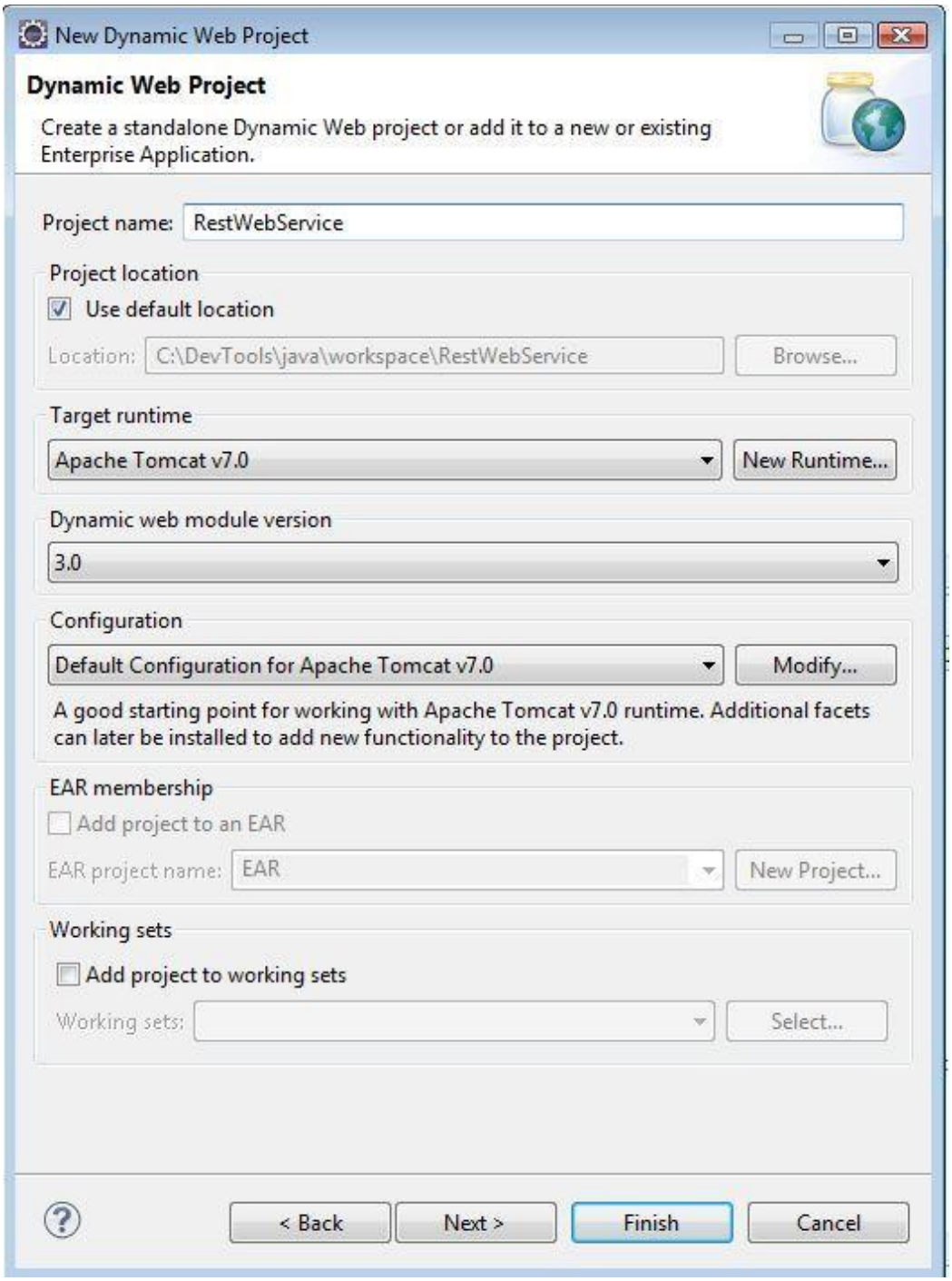
The home page, and the place to download the required jars for jersey can be found here. As of the writing of this part, the jars will be found in a zip called jersey-archive-1.16.zip . Please download this zip and expand it to a folder where you can find those jars.

Create a “Dynamic Web Project” in Eclipse

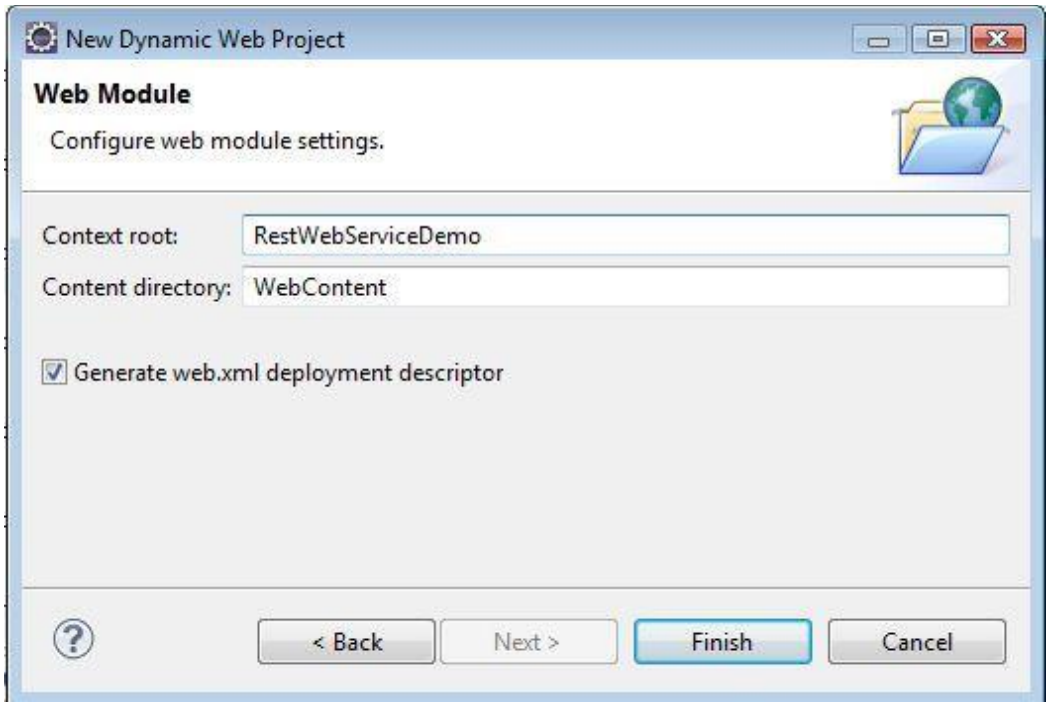
In Eclipse, select “File”->”New…”->”Project” and use the filter to find “Dynamic Web Project”



I have Apache Tomcat 7 running on my laptop, and have previously configured a connection between Tomcat and Eclipse. The Dynamic web module version I am using is 3.0, but this tutorial can work with version 2.5.

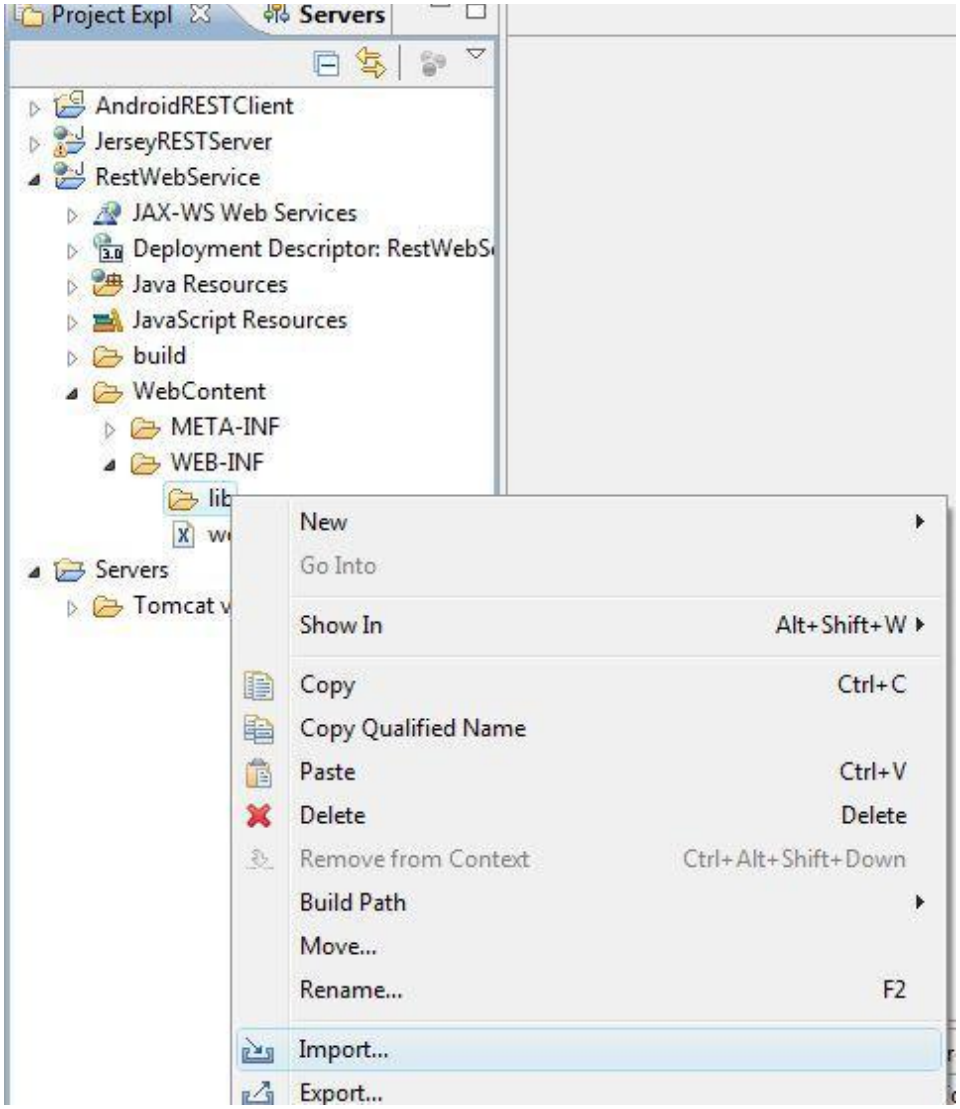


Click “Next” and define your context root. That is the starting-point URL for your web service. With the context root shown in the screenshot below, the resulting URL on my laptop will be http://localhost:8080/RestWebServiceDemo . Also, have Eclipse generate a web.xml deployment descriptor.

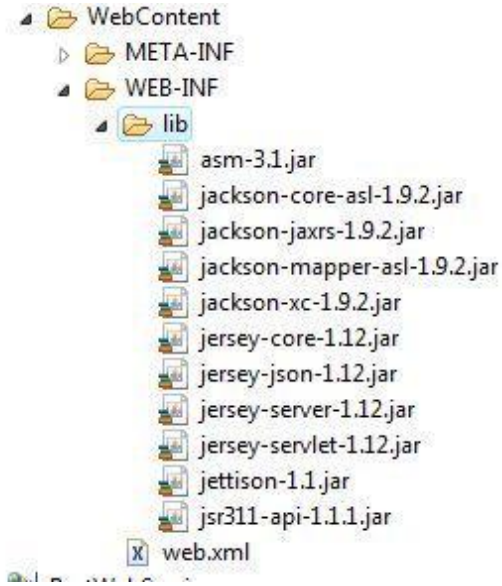


Add Jersey Jars to Project

You will need to import the jars that you've downloaded from http://jersey.java.net into the WEB-INF/lib folder. Right click that folder (WebContent/WEB-INF/lib) and select Import…

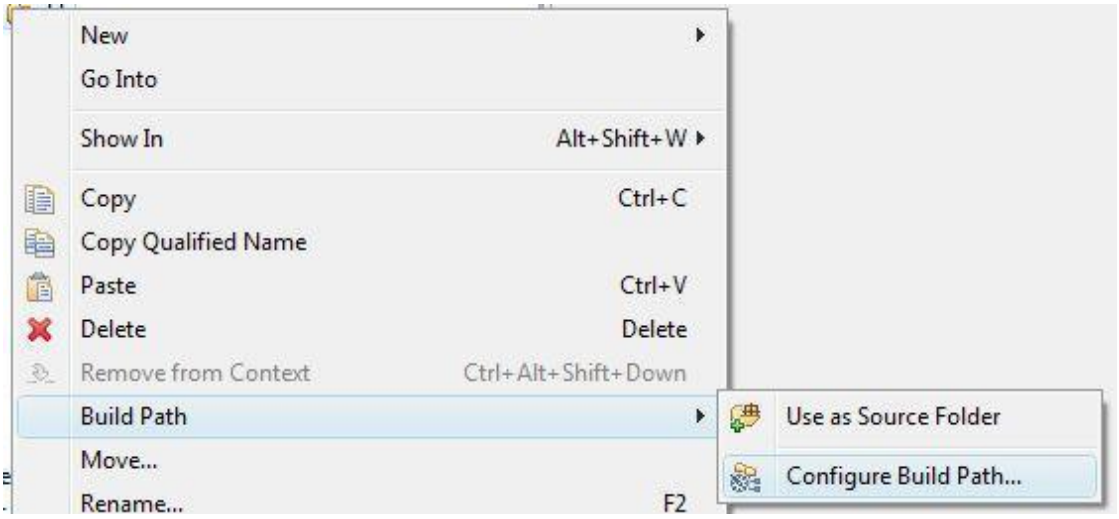


I've simply thrown in all the jars into that folder for the purposes of this tutorial. Crude, but effective. The smart thing would be to only use the jars you need.

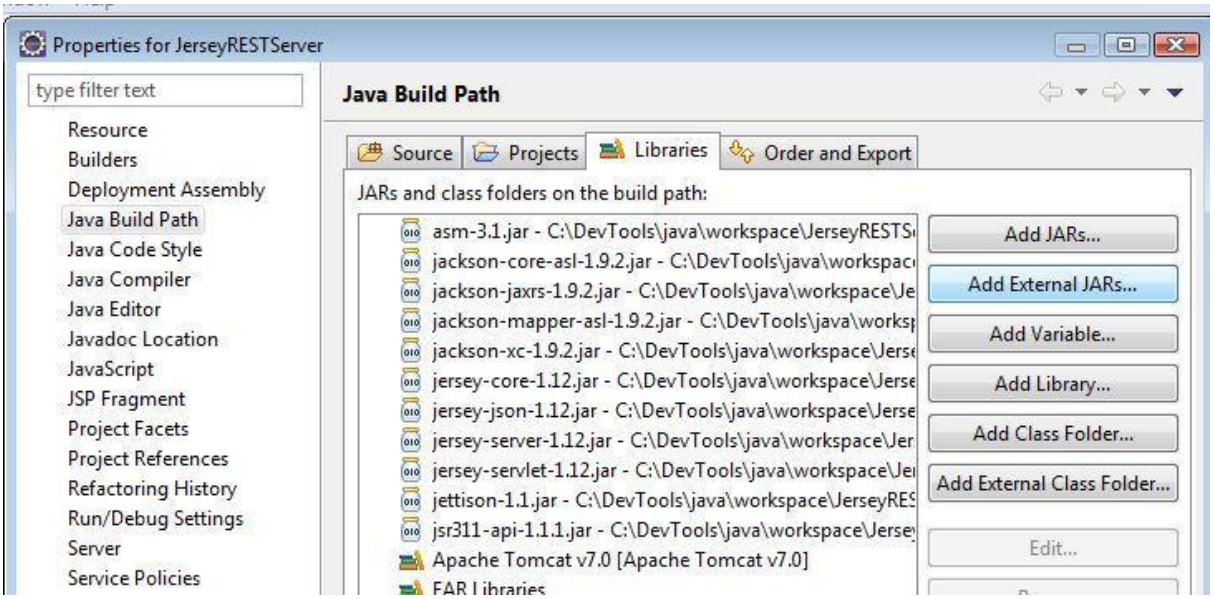


Update Project Build Path to Include Jersey Jars

Now that the jars are in the WEB-INF/lib folder, you will need to configure the project to include these jars in its build path. From within your project in the Package Explorer on the left of your Eclipse screen, right click to select “Build Path”->”Configure Build Path…”



On the “Libraries” tab, click on “Add External JARS…” and select the jars that are now in your project’s WEB-INF/lib path.



Create the POJO DimmableSwitch class

Within the \src folder, create a package called **fr.esir2.ticb.rest.model**, and in that package, create a Java class called **DimmableSwitch**. Note, in the code shown below, the**@XmlRootElement** annotation. This tells Jersey that this would be the root object of any generated XML (or JSON) representation of this class. This might not be very useful for this class, but if you had a compound class, you’d be able to control what the XML or JSON output would look like by the addition of annotations like this.

```
package fr.esir2.ticb;

@XmlRootElement
```

```
public class DimmableSwitch {

	String name;
	String address;
	int value;
	public DimmableSwitch() {
	}
	public DimmableSwitch(String name, String address, int value) {
		super();
		this.name = name;
		this.address = address;
		this.value = value;
	}
	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;
	}
	public int getValue() {
		return value;
	}
	public void setValue(int value) {
		this.value = value;
	}
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((address == null) ? 0 :
address.hashCode());
		result = prime * result + ((name == null) ? 0 : name.hashCode());
		result = prime * result + value;
		return result;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		DimmableSwitch other = (DimmableSwitch) obj;
		if (address == null) {
```

```
                if (other.address != null)
                        return false;
        } else if (!address.equals(other.address))
                return false;
        if (name == null) {
                if (other.name != null)
                        return false;
        } else if (!name.equals(other.name))
                return false;
        if (value != other.value)
                return false;
        return true;
    }


}
```

Create the DimmableSwitchResource class

Again, in the \src folder, create a new package called **com.avilyne.rest.resource**, and in that package create a class called **DimmableSwitchResource**. (We are separating our data model from our controller).
The **DimmableSwitchResource** class will be the interface between a **DimmableSwitch** object and the web. When a client requests a DimmableSwitch object, the PersonResource will handle the request and return the appropriate object.
Note, in the code below, the annotations. The **@Path("/switch")** annotation allows us to define the URI that will have access to this resource. It will be something along the lines of http://RestServerDemo/rest/switch, but this will be explained in more detail as we map this code in our web.xml file.
The @**Context** annotation allows us to inject, in this example, UriInfo and Request objects into our DimmableSwitchResource object. Our code can inspect those injected objects for any desired Context information.

```
package fr.esir2.ticb;

import javax.ws.rs.core.Context;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;

public class DimmableSwitchResource {


    // The @Context annotation allows us to have certain contextual objects
    // injected into this class.
    // UriInfo object allows us to get URI information (no kidding).
    @Context
    UriInfo uriInfo;

    // Another "injected" object. This allows us to use the information that's
```

```
        // part of any incoming request.
        // We could, for example, get header information, or the requestor's address.
        @Context
        Request request;
}
```

The @**GET** annotation lets us specify what method will be called when a client issues a GET to the webservice. Similar logic applies to the @**POST** annotation. Note that the actual method name will not be a part of the client's URI.

Also note that more than one method has a @GET annotation. The first @GET is simply there so that, when we start our service, we can use a browser to retrieve some sort of response that indicates the service is running. Technically, instead of producing TEXT_PLAIN, the first GET could have produced a TEXT_HTML page.

```
        // Basic "is the service running" test
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        public String respondAsReady() {
         return "Demo service is ready!";
        }
```

The @**Path** annotation lets us append a parameter onto our URI. Using the example from earlier, the hypothetical URI would be http://RestServerDemo/rest/person/sample .

```
 DimmableSwitch sw = new DimmableSwitch("test", "test 1", 56);
```

```
        @GET
        @Path("sample")
        @Produces(MediaType.APPLICATION_JSON)
        public DimmableSwitch getSampleDimmableSwitch() {

         System.out.println("Returning sample Dimmable Switch: " + sw.getName() + " "
+ sw.getAddress() + " " + sw.getValue());

         return sw;
        }
```

Note that paths do NOT need to be literal. One can have a path parameter as a variable, e.g. **@Path("{id}")** would refer to a switch whose id value matched the given URI value. http://RestServerDemo/rest/switch/1 should return a switch whose id is 1.
The @**Produces** annotation allows us to define how the output from our resource should be transmitted to our client. Note that, for the getSampleSwitch() method, we return a switch

object, and the annotation lets us tell Jersey to format and transmit that switch as a JSON object.
The method with the @POST annotation also includes a **@Consumes** annotation. As you can guess, this method is called in response to a client's POST request. The data for the "switch" object being transmitted from the client is not a JSON object, but is a collection of Name-Value pairs. The @Consumes annotation allows us to specify that the data passed from the client is an array of these pairs, and we can pull out the values we want from that array.

```
        // Use data from the client source to create a new Person object, returned in JSON format.
        @POST
        @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
        @Produces(MediaType.APPLICATION_JSON)
public DimmableSwitch postDimmableSwitch(
        MultivaluedMap<String, String> switchParams
        ) {

        String name = switchParams.getFirst(NAME);
        String address = switchParams.getFirst(ADDRESS);
        String value = switchParams.getFirst(VALUE);

        System.out.println("Storing posted " + name + " " + address + " " + value);

        sw.setName(name);
        sw.setAddress(address);
        sw.setValue(Integer.parseInt(value));

        System.out.println("Switch info: " + sw.getName() + " " + sw.getAddress() + " " + sw.getValue());

        return sw;

}
```

For each of the methods in this DimmableSwitchResource, I've added a System.out.println() as a crude way of letting us see when a request is being processed. There are probably more elegant ways of doing this (Logging, for one), and one would almost never include a System.out.println in a production service. This is just a demo.

```
package fr.esir2.ticb;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
```

```
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;

@Path("/switch")
public class DimmableSwitchResource {


        private final static String NAME = "name";
private final static String ADDRESS = "address";
private final static String VALUE = "value";

        // The @Context annotation allows us to have certain contextual objects
        // injected into this class.
        // UriInfo object allows us to get URI information (no kidding).
        @Context
        UriInfo uriInfo;

        // Another "injected" object. This allows us to use the information that's
        // part of any incoming request.
        // We could, for example, get header information, or the requestor's address.
        @Context
        Request request;

        // Basic "is the service running" test
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        public String respondAsReady() {
         return "Demo service is ready!";
        }

        DimmableSwitch sw = new DimmableSwitch("test", "test 1", 56);


        @GET
        @Path("sample")
        @Produces(MediaType.APPLICATION_JSON)
        public DimmableSwitch getSampleDimmableSwitch() {

         System.out.println("Returning sample Dimmable Switch: " + sw.getName() + " "
+ sw.getAddress() + " " + sw.getValue());

         return sw;
        }

        // Use data from the client source to create a new Person object, returned in JSON
format.
```

```
        @POST
        @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
        @Produces(MediaType.APPLICATION_JSON)
        public DimmableSwitch postDimmableSwitch(
         MultivaluedMap<String, String> switchParams
         ) {

         String name = switchParams.getFirst(NAME);
         String address = switchParams.getFirst(ADDRESS);
         String value = switchParams.getFirst(VALUE);

         System.out.println("Storing posted " + name + " " + address + " " + value);

         sw.setName(name);
         sw.setAddress(address);
         sw.setValue(Integer.parseInt(value));

         System.out.println("Switch info: " + sw.getName() + " " + sw.getAddress() + " "
+ sw.getValue());

         return sw;

        }

}
```

Create or Edit the WebContent\WEB-INF\web.xml file

In our web.xml file, we want to direct all requests to a servlet called Jersey REST Service. We will also tell this service where to find the resources that we want to make available to our client app.

Update the web.xml file (or create it, in WebContent\WEB-INF\web.xml, if it does not exist) to match the XML shown below.

The XML defines the Jersey REST Service from the com.sun.jersey.spi.container.ServletContainer class. That class, as you might guess, is in one of the Jersey Jars. The init-param section allows us to tell that servletcontainer to use the classes found in the com.avilyne.rest.resource package for the mapping of URIs to java code.

In the servlet-mapping section, we create a global url-pattern, which essentially says that any request that goes to **/rest/** will attempt to be mapped to the appropriate methods.

```
<?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"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID"
```

```
version="3.0">
        <display-name>JerseyRESTServer</display-name>
        <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-class> org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>fr.esir2.ticb</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        </servlet>
<servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
        <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

Run the WebService

Run the completed project as a service on your Tomcat servlet container. Export you project as a war file. (Right clic export as war) The actual URL for the service will vary slightly, depending partly on the name of your war file you export. I called my war file **RestWebServiceDemo.war** and here is the URL for the web service on my computer:

http://localhost:8080/RestWebServiceDemo/rest/switch

RestWebServiceDemo is the context root that we defined when we first started this project. I'll mention that I built a few versions of this project before writing this post, so unfortunately some of the images show a URL from an earlier version of the project.
If I bring up this on a browser, it returns with this:

If I add the "sample" path to the URL, like this:
http://localhost:8080/RestWebServiceDemo/rest/switch/mylamp
The web service returns a JSON object:

```
{"name":"mySwitch","address":"bla","id":"1","value":"100"}
```

I would recommend that you make sure you can retrieve this sample person before you create the Android client app.

Edit an Android Client App

Edit your previously created Android project. I used API level 10 for this project, but one can probably use a lower level API, if required. Android HttpClient requests have been around since the earliest days of the OS.

Edit the \res\layout\main.xml File

The interface will be very simple, having three labels and three edit controls for name, address and value. It will also have three buttons – one to GET, one to POST, and one to clear the controls.

Edit the AndroidRESTClientActivity.java file

Discuss the WebServiceTask, and the handleResponse() method.

The full source code for the Android client is listed below, but I want to highlight a few items in the code that you should be aware of. On my home network, the computer running the service on Tomcat is at IP address 192.168.1.9. On your network, the address will most definitely be different. If you are running the Android client and the Tomcat service on the same computer. Use the IP address of the computer on your network that is running the Tomcat service.)

For this tutorial, the most important code is the internal "WebServiceTask" class, which is extended from an "AsyncTask" class. An AsyncTask class descendant allows a process to run in a separate thread. If our communication with our service were on the Android app's main thread, the user interface would lock up as the process was waiting for results from the server.

One can define the types of parameters that are passed to one's instance of the AsyncTask. (more on that later). The communication with the web service occurs in the WebServiceTask's "doInBackground()" code. This code uses Android's HttpClient object, and for the GET method, uses HttpGet, and for the POST method, uses HttpPost.

The AsyncTask class includes two other methods that one has the option to overwrite. One is onPreExecute(), which one can use to prepare for the background process, and the other is onPostExecute(), which one can use to do any required clean-up after the background process is complete. This code overrides those methods to display and remove a progress dialog.

The background task also includes two timeout options. One is a timeout period for the actual connection to the service, and the other is a timeout period for the wait for the service's response.

```
package com.example.testrestandroid;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

import org.apache.http.HttpResponse;
import org.apache.http.NameValuePair;
import org.apache.http.client.HttpClient;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpConnectionParams;
import org.apache.http.params.HttpParams;
```

```
import org.json.JSONObject;

import android.app.Activity;
import android.app.ProgressDialog;
import android.content.Context;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.inputmethod.InputMethodManager;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity {
private static final String SERVICE_URL =
"http://192.168.1.9:8080/RestWebServiceDemo/rest/person";
private static final String TAG = "AndroidRESTClientActivity";
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
}
public void retrieveSampleData(View vw) {
String sampleURL = SERVICE_URL + "/sample";
WebServiceTask wst = new WebServiceTask(WebServiceTask.GET_TASK, this, "GETting
data...");
wst.execute(new String[] { sampleURL });
}
public void clearControls(View vw) {
EditText edFirstName = (EditText) findViewById(R.id.name);
EditText edLastName = (EditText) findViewById(R.id.address);
EditText edvalue = (EditText) findViewById(R.id.value);
edFirstName.setText("");
edLastName.setText("");
edvalue.setText("");
}
public void postData(View vw) {
EditText edFirstName = (EditText) findViewById(R.id.name);
EditText edLastName = (EditText) findViewById(R.id.address);
EditText edvalue = (EditText) findViewById(R.id.value);
String firstName = edFirstName.getText().toString();
String lastName = edLastName.getText().toString();
String value = edvalue.getText().toString();
if (firstName.equals("") || lastName.equals("") || value.equals("")) {
Toast.makeText(this, "Please enter in all required fields.",
Toast.LENGTH_LONG).show();
return;
}
WebServiceTask wst = new WebServiceTask(WebServiceTask.POST_TASK, this,
"Posting data...");
```

```
wst.addNameValuePair("firstName", firstName);
wst.addNameValuePair("lastName", lastName);
wst.addNameValuePair("value", value);
// the passed String is the URL we will POST to
wst.execute(new String[] { SERVICE_URL });
}
public void handleResponse(String response) {
EditText edFirstName = (EditText) findViewById(R.id.name);
EditText edLastName = (EditText) findViewById(R.id.address);
EditText edvalue = (EditText) findViewById(R.id.value);
edFirstName.setText("");
edLastName.setText("");
edvalue.setText("");
try {
JSONObject jso = new JSONObject(response);
String firstName = jso.getString("firstName");
String lastName = jso.getString("lastName");
String value = jso.getString("value");
edFirstName.setText(firstName);
edLastName.setText(lastName);
edvalue.setText(value);
} catch (Exception e) {
Log.e(TAG, e.getLocalizedMessage(), e);
}
}
private void hideKeyboard() {
InputMethodManager inputManager = (InputMethodManager) MainActivity.this
.getSystemService(Context.INPUT_METHOD_SERVICE);
inputManager.hideSoftInputFromWindow(
MainActivity.this.getCurrentFocus()
.getWindowToken(), InputMethodManager.HIDE_NOT_ALWAYS);
}
private class WebServiceTask extends AsyncTask<String, Integer, String> {
public static final int POST_TASK = 1;
public static final int GET_TASK = 2;
private static final String TAG = "WebServiceTask";
// connection timeout, in milliseconds (waiting to connect)
private static final int CONN_TIMEOUT = 3000;
// socket timeout, in milliseconds (waiting for data)
private static final int SOCKET_TIMEOUT = 5000;
private int taskType = GET_TASK;
private Context mContext = null;
private String processMessage = "Processing...";
private ArrayList<NameValuePair> params = new ArrayList<NameValuePair>();
private ProgressDialog pDlg = null;
public WebServiceTask(int taskType, Context mContext, String processMessage) {
this.taskType = taskType;
this.mContext = mContext;
this.processMessage = processMessage;
}
public void addNameValuePair(String name, String value) {
```

```
params.add(new BasicNameValuePair(name, value));
}
private void showProgressDialog() {
pDlg = new ProgressDialog(mContext);
pDlg.setMessage(processMessage);
pDlg.setProgressDrawable(mContext.getWallpaper());
pDlg.setProgressStyle(ProgressDialog.STYLE_SPINNER);
pDlg.setCancelable(false);
pDlg.show();
}
@Override
protected void onPreExecute() {
hideKeyboard();
showProgressDialog();
}
protected String doInBackground(String... urls) {
String url = urls[0];
String result = "";
HttpResponse response = doResponse(url);
if (response == null) {
return result;
} else {
try {
result = inputStreamToString(response.getEntity().getContent());
} catch (IllegalStateException e) {
Log.e(TAG, e.getLocalizedMessage(), e);
} catch (IOException e) {
Log.e(TAG, e.getLocalizedMessage(), e);
}
}
return result;
}
@Override
protected void onPostExecute(String response) {
handleResponse(response);
pDlg.dismiss();
}
// Establish connection and socket (data retrieval) timeouts
private HttpParams getHttpParams() {
HttpParams htpp = new BasicHttpParams();
HttpConnectionParams.setConnectionTimeout(htpp, CONN_TIMEOUT);
HttpConnectionParams.setSoTimeout(htpp, SOCKET_TIMEOUT);
return htpp;
}
private HttpResponse doResponse(String url) {
// Use our connection and data timeouts as parameters for our
// DefaultHttpClient
HttpClient httpclient = new DefaultHttpClient(getHttpParams());
HttpResponse response = null;
try {
switch (taskType) {
```

```
case POST_TASK:
HttpPost httppost = new HttpPost(url);
// Add parameters
httppost.setEntity(new UrlEncodedFormEntity(params));
response = httpclient.execute(httppost);
break;
case GET_TASK:
HttpGet httpget = new HttpGet(url);
response = httpclient.execute(httpget);
break;
}
} catch (Exception e) {
Log.e(TAG, e.getLocalizedMessage(), e);
}
return response;
}
private String inputStreamToString(InputStream is) {
String line = "";
StringBuilder total = new StringBuilder();
// Wrap a BufferedReader around the InputStream
BufferedReader rd = new BufferedReader(new InputStreamReader(is));
try {
// Read response until the end
while ((line = rd.readLine()) != null) {
total.append(line);
}
} catch (IOException e) {
Log.e(TAG, e.getLocalizedMessage(), e);
}
// Return full string
return total.toString();
}
}
}
```

Update the AndroidManifest.xml

Since this app needs to communicate via the internet, one must give the app the appropriate permissions.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.avilyne.android"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:icon="@drawable/ic_launcher"
```

```
        android:label="@string/app_name" >
        <activity
            android:name=".AndroidRESTClientActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category   android:name="android.intent.category.LAUNCHER"
/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Test your application ;)

It should work.