TDW-1148
Hands-on Lab: Building Mobile Applications Using JAXRS, Dojo, and Phone Gap

Bryce A. Curtis, Ph.D.
Mobile & Emerging Technologies,
Todd Kaplinger
Web2/FeP Release Architect

March 25, 2011

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal at IBM's sole discretion.Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole **discretion.**

Part I - Mysurance PhoneGap JAX-RS WebSphere Service

1.0 Introduction

The Mysurance application can be viewed as two components, server and client application components. For Part I of this demo lab, we will be concentrating on the server application component. The mysurance application provides three JAX-RS REST APIs for services relating to User Profiles, Vehicles and File Upload. For those not familiar with JAX-RS, JAX-RS or Java API for RESTful Web Services is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural style. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints. In this demo we will be leveraging RAD 8.01 and WebSphere V8.0 Beta to demonstrate how to create JAX-RS services as well as how to access these REST APIs via HTTP.

1.1 Prerequisites

Prior to getting started, there will be one prerequisite required for validating our JAX-RS services that we build in this demo. Using Using the VM image, obtain the Firefox plugin Poster. (https://addons.mozilla.org/en-us/firefox/addon/poster/). Poster will allow us to simulate the various JAX-RS REST API calls that will be performed and verify that each of these actions are working successfully prior to writing our Mysurance client APIs that will be used in the Android emulator in Part II.

1.1.1 Installing Poster

The first step in installing poster is to click on the button "Add to Firefox" to install Poster.

Illustration 1: Poster Add to Firefox dialog

At this point, you will receive a dialog that asks whether you want to continue to install. Select the Poster item and then click “Install now” which will continue the install process.

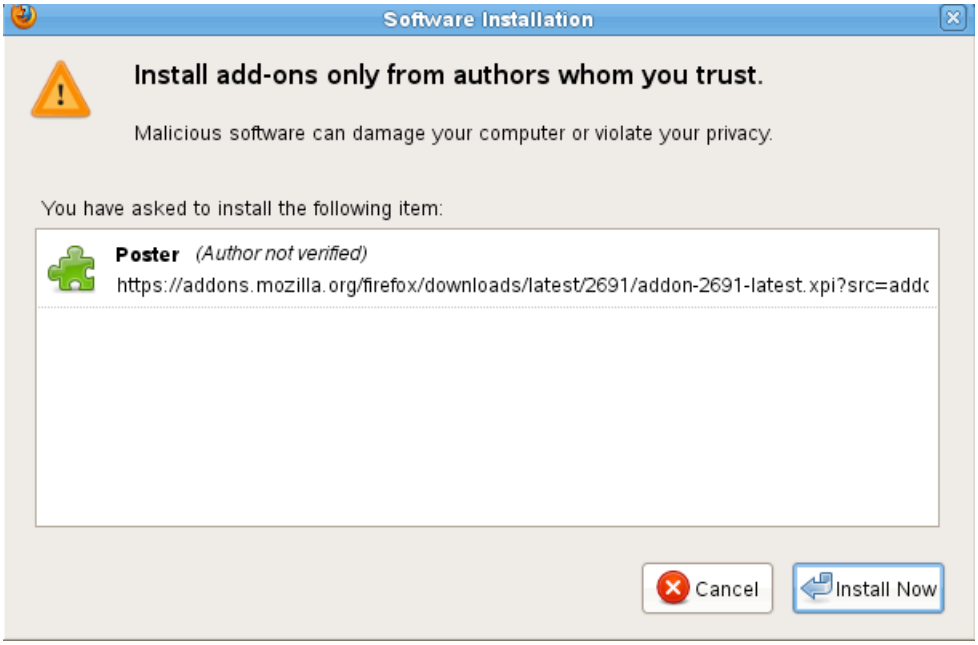

Illustration 2: Install verifcation screen for Poster

After the plugin installs, the browser will ask you to “Restart Firefox”. Click on this button and the plugin will complete the installation process and be available for use later in the demo.

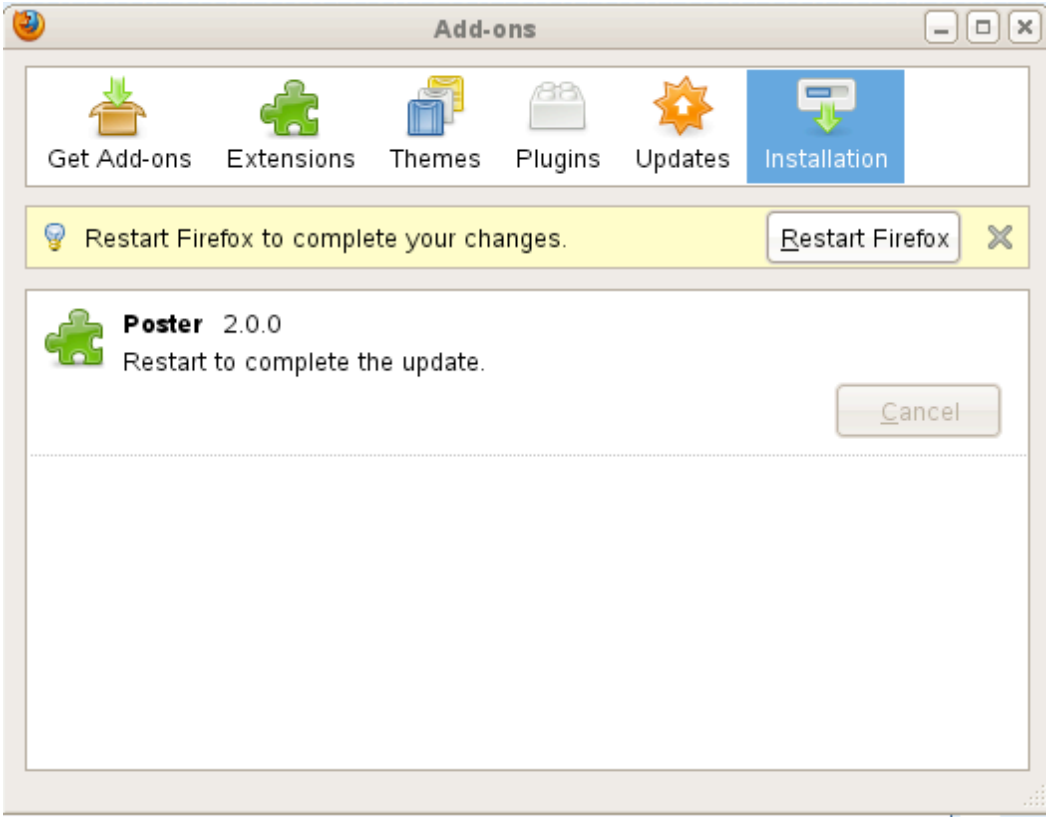

Illustration 3: Restart Firefox dialog after Poster plugin has been installed

1.2 Getting Started

To begin the demo, we will first download the latest version of mysurance app from git hub (http://brycecurtis.github.com/phonegap-mysurance/mysurance.war) to your desktop**.** The mysurance.app provides the initial application structure that we will use to demonstrate the JAX-RS services providing a default set of template files and configuration that we can build upon as we progress through the demonstration. Once the WAR file has been downloaded to you desktop, you should see an icon similar to the following illustration.

Illustration 4: mysurance war file downloaded to Desktop

1.2.1 Starting Rational Application Developer 8.01

Click on the **RAD 8** icon on the desktop. This instance of RAD is preloaded with a runtime for testing the JAX-RS services that we will accessed from the android emulator as well as via Poster for our initial testing. In this demonstration, we will be using WebSphere 8.0 Beta for hosting our Mysurance application and will demonstrate how to build an deploy this application to the WebSphere 8.0 Unit Test Environment (UTE).

1.2.2 Using the existing Impact2011Demo Web Application Archive *(Optional)*

For users that are already familiar with JAX-RS, there is already a prebuilt version of these JAX-RS services located in Impact2011Demo if you would prefer to skip this portion of the demo. However, before moving to the next section, there are two small changes that are required to be made to make the services compatible with the PhoneGap client code that will be interacting with these services.

```
com.ibm.impact.demo.services.UserProfileDataService.createUserProfile
com.ibm.impact.demo.services.VehicleDataService.createVehicle
```

change the @POST annotation to @PUT for these two methods. The original code incorrectly had these methods defined as POST (update) when it should have been PUT (create). At this point, you can now continue onto **Part II - Mysurance PhoneGap Client for Android** and start to build the PhoneGap aspect of the application.

1.3 Building the Mysurance Web Application

To aid in the development process, we are providing the mysurance.war with a set of templates and code fragments already packaged as part of the web application. The goal of these templates and code fragments are to make things easier to find and allow you to more quickly assemble the applications for testing.

1.3.1 Importing the Mysurance Web Application Archive

The next step is the development of the server side components is to import the mysurance.war into your RAD 8 workspace. To import the newly downloaded war file, select from the top level menu "File → Import". At this point in time you should see a dialog to import a WAR file.

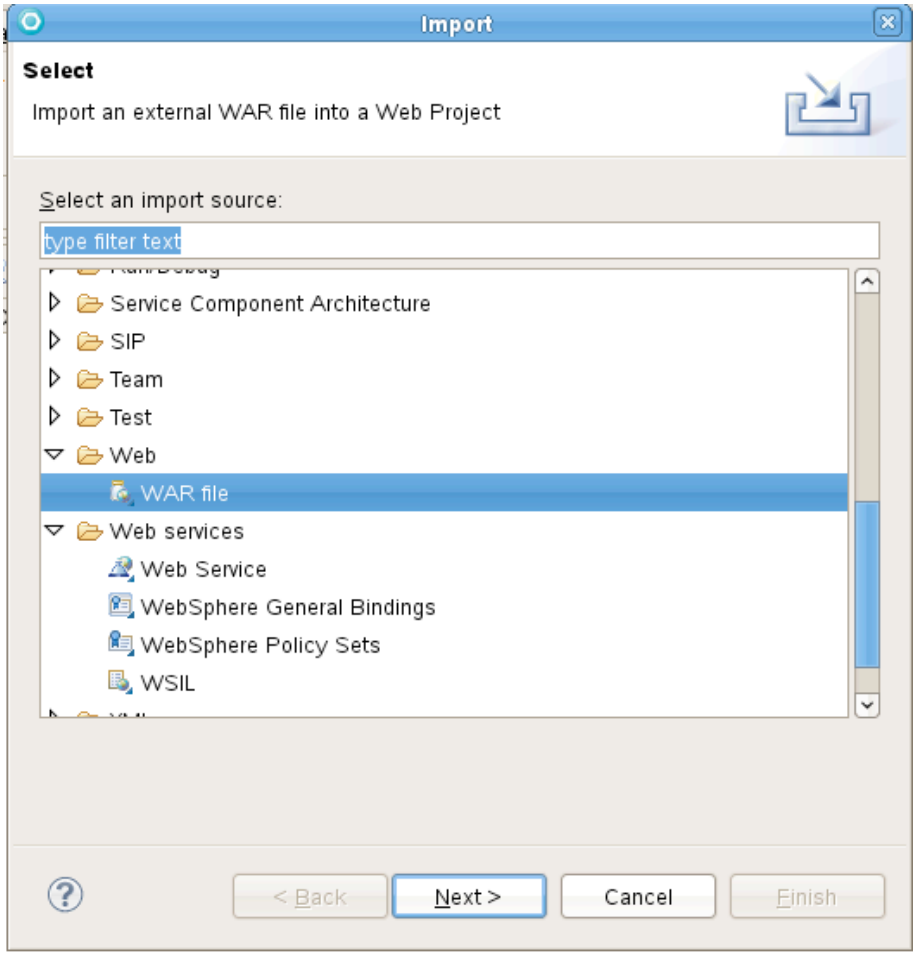

Illustration 5: Import Web resource wizard

Click on WAR file and then click "Next >".

At this point, it is time to select the WAR file you want to import. Based upon the prior step where you downloaded the mysurance.war file to the "Desktop", click on the "Desktop" item in the list of "Places" and you should see the mysurance.war file.

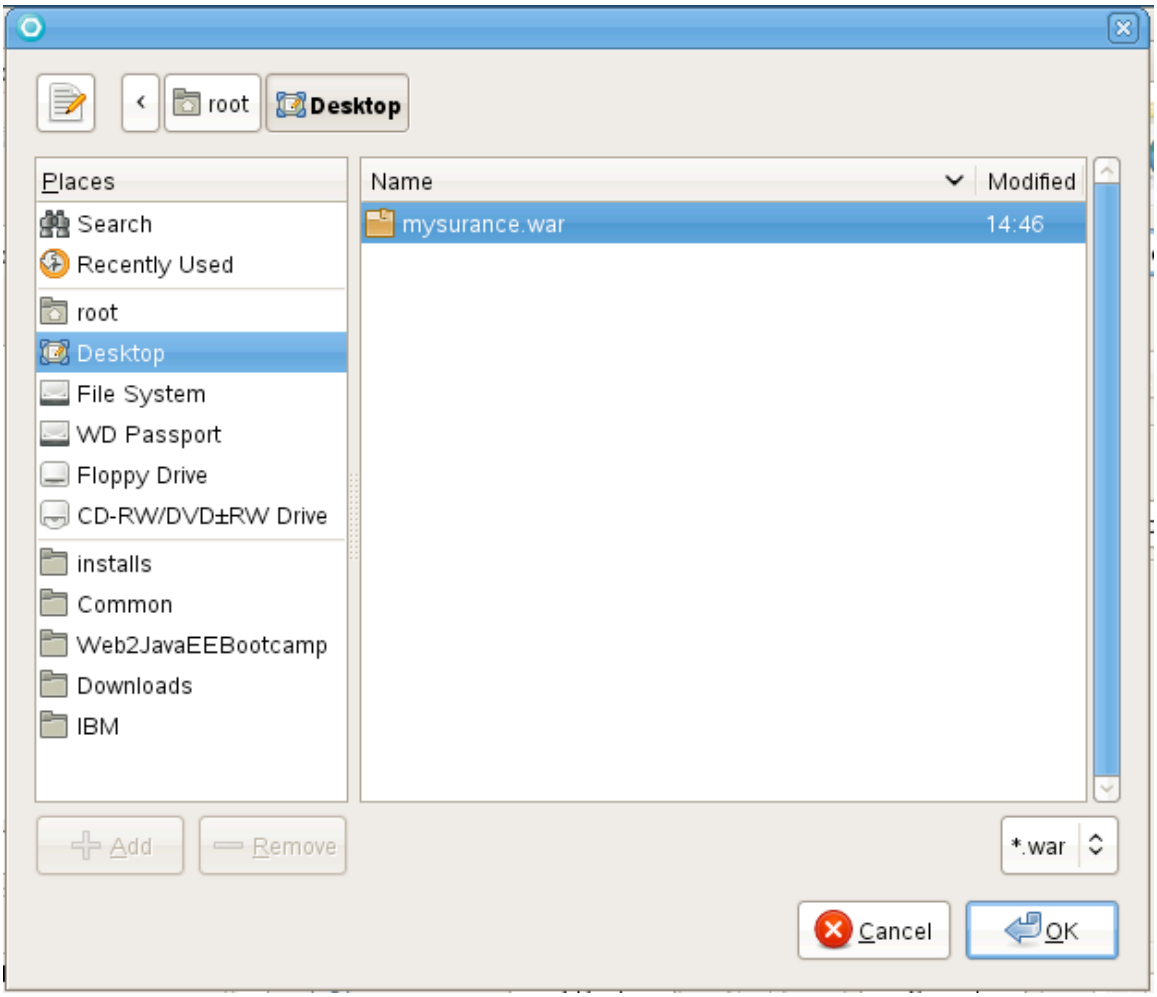

Illustration 6: Panel for locating mysurance in the filesytem (Desktop)

Select the mysurance.war file and click on "OK". At this point in time, the wizard dialog should bring you back to the panel where you can complete the process for importing the WAR file. For this example, you can just click "Finish" to complete the import process.

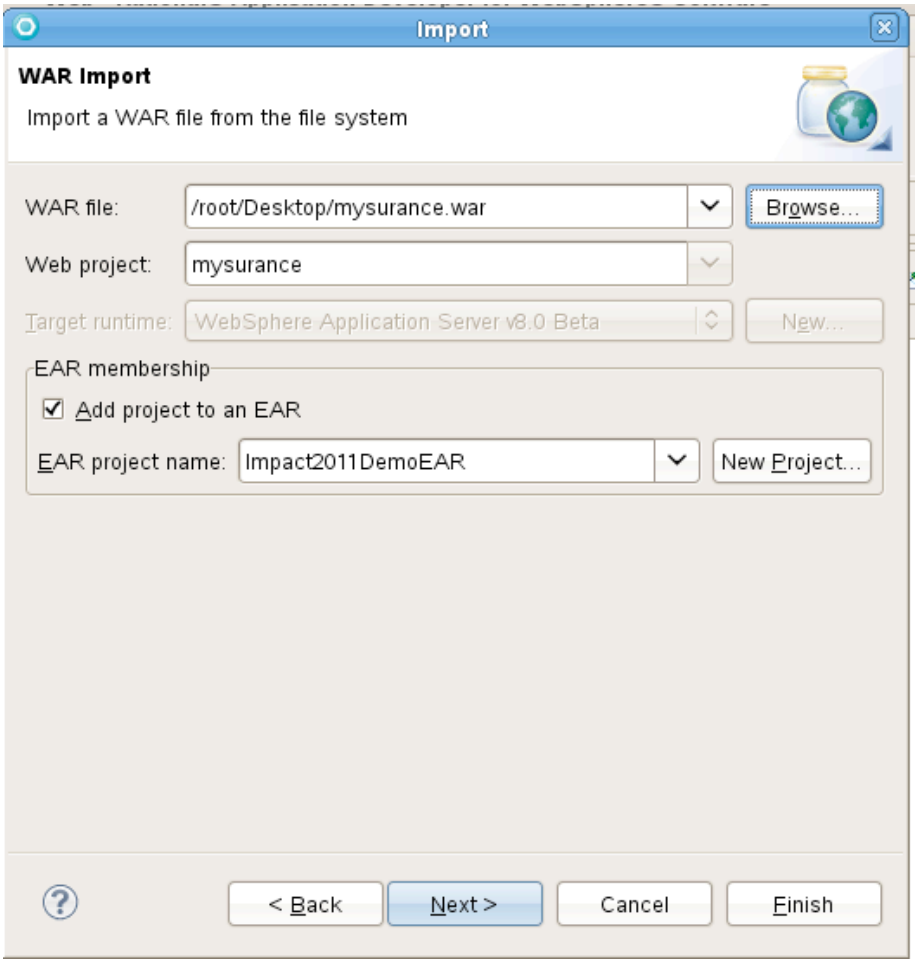

Illustration 7: Associating the mysurance.war with the Impact2011DemoEAR

At this point in time the mysurance.war has been imported into RAD and has been associated with the Impact2011DemoEAR project.

1.3.2 Confirm the importing of the Mysurance Web Application

Now that you have gone through the steps of importing the mysurance.war and associated the application with the Impact2011DemoEAR, the project workspace should look as follows.

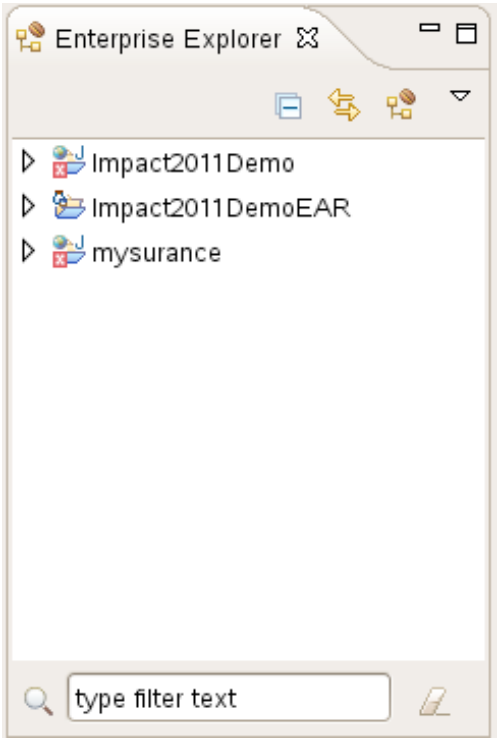

Illustration 8: Explorer view after importing the mysurance.war into the workspace

1.3.3 Copying the required libraries for application.

Inside of the Impact2011Demo project there are a set of JAX-RS core and support libraries that will be required as part of the mysurance application.

Expand the Impact2011Demo project WebContent → WEB-INF → lib directory

Copy *all* of the JAR files to mysurance → WebContent → WEB-INF → lib directory (1)

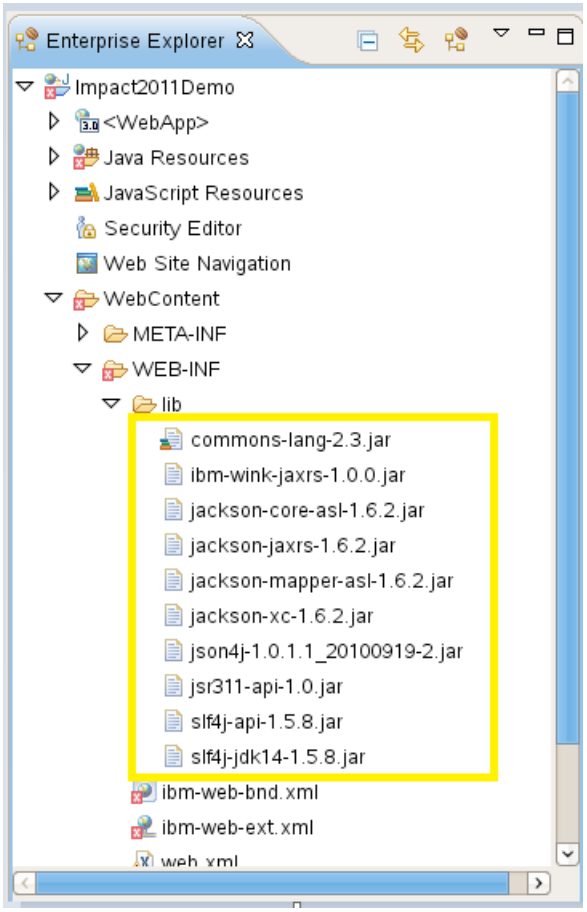

Illustration 9: List of JAR resources to copy from Impact2011Demo war

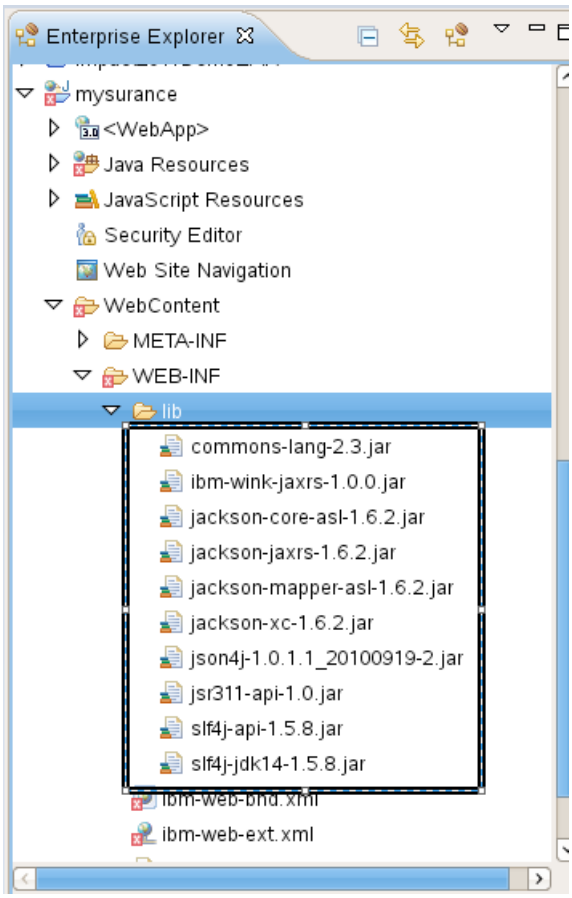

Illustration 10: Verification JAR resources have been copied to mysurance.war

1.3.4 Registering our JAX-RS Servlet and configuring our Application instance

In this step, we will configure the JAX-RS Servlet that will act as a controller servlet for all JAX-RS resources that are registered for the application. The servlet mapping for each of these JAX-RS resources will be prefixed with '/rest'.

In the com.ibm.impact.demo.DemoApplicationRegistry, the Application instance will be responsible for registering each of the JAX-RS services that we will want to expose for this demo as well as JAXB handler responsible for transforming our payloads between JSON and JAXB.

The web.xml that we will edit is located in mysurance → WebContent → WEB-INF. The default view for the web.xml is “Design” view and will look similar to what is presented below.

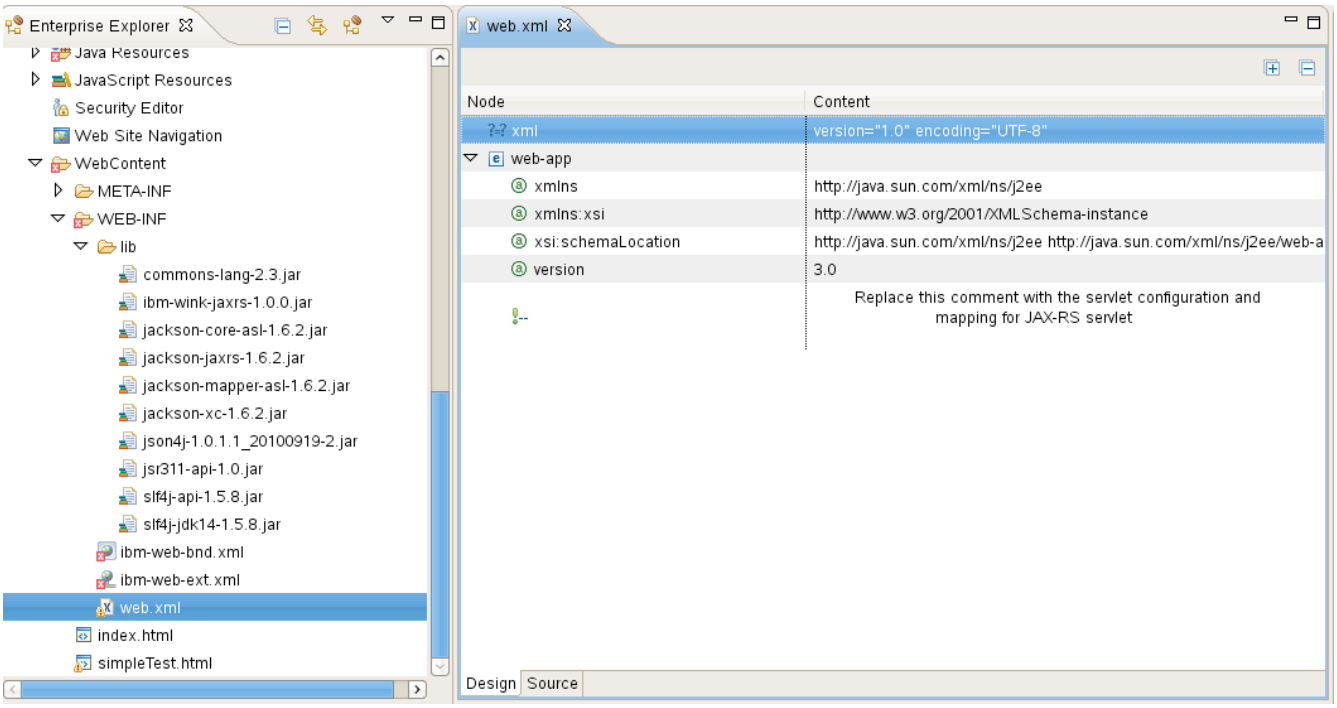

Illustration 11: Design view of mysurance.war's web.xml

Now that we have the web.xml in the "Design" view, click on the "Source" view option in the bottom part of the current editor window. At this time you will see the XML representation of the web.xml.

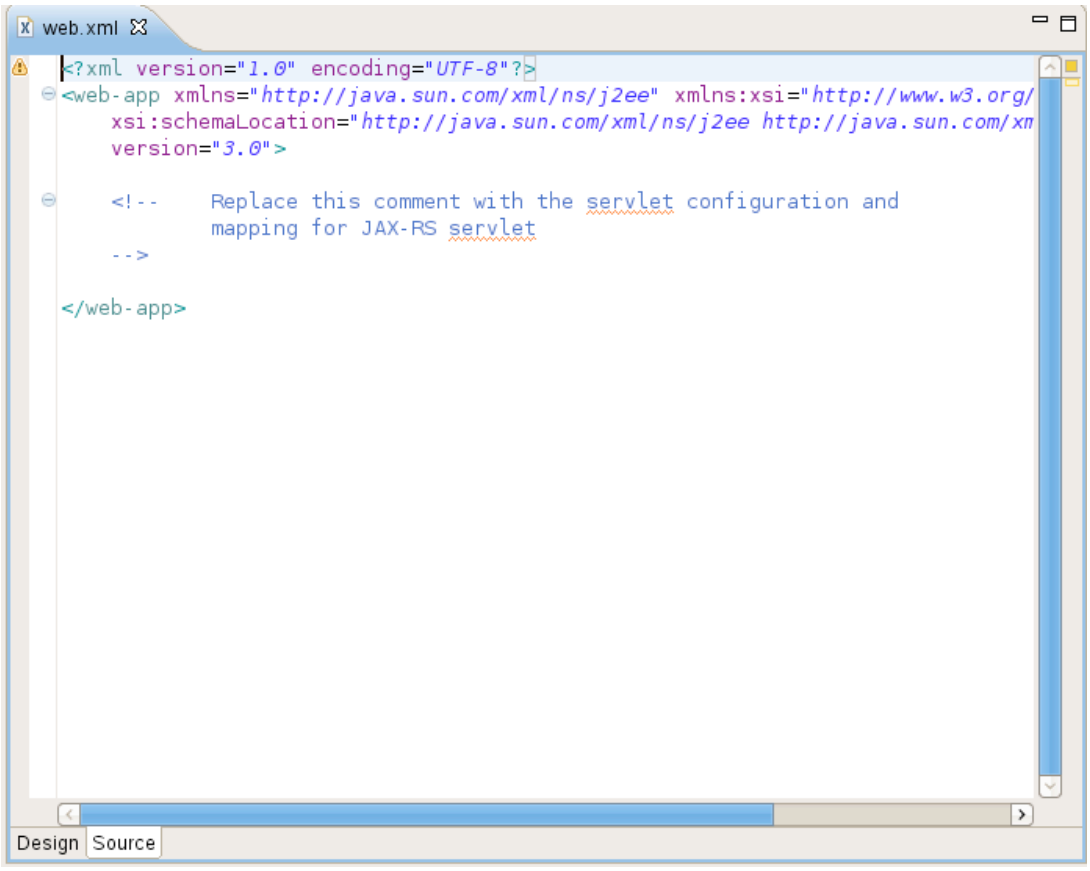

Illustration 12: Starting point source view of mysurance.war's web.xml

Per the instructions, you can now replace the placeholder XML comment with the actual JAX-RS servlet configuration defined here.

```
    <servlet>
        <servlet-name>JAXRSServlet</servlet-name>
        <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.ibm.impact.demo.DemoApplicationRegistry</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>JAXRSServlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
```

The web.xml should now look now have the following configuration.

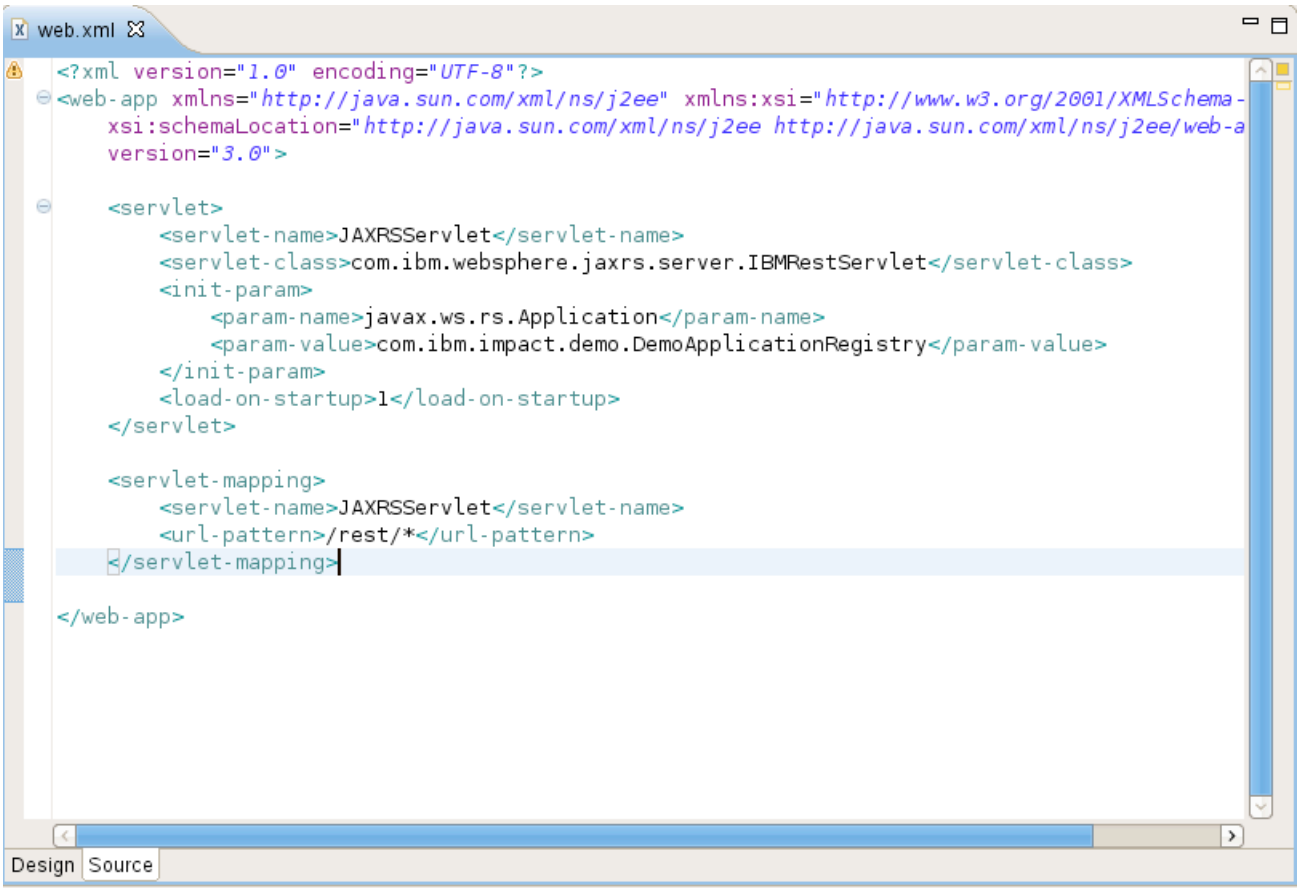

Illustration 13: Server configuration file.

1.3.5 Creation of the Application instance

Now that we have defined the javax.ws.rs.Application instance named com.ibm.impact.demo.DemoApplkciationRegistry, it is time to define this class . To start with, we will

create an empty implementation. As a placeholder, the mysurance application has an empty class named DemoApplicationRegistry that matches what has been defined in our web.xml for this application.

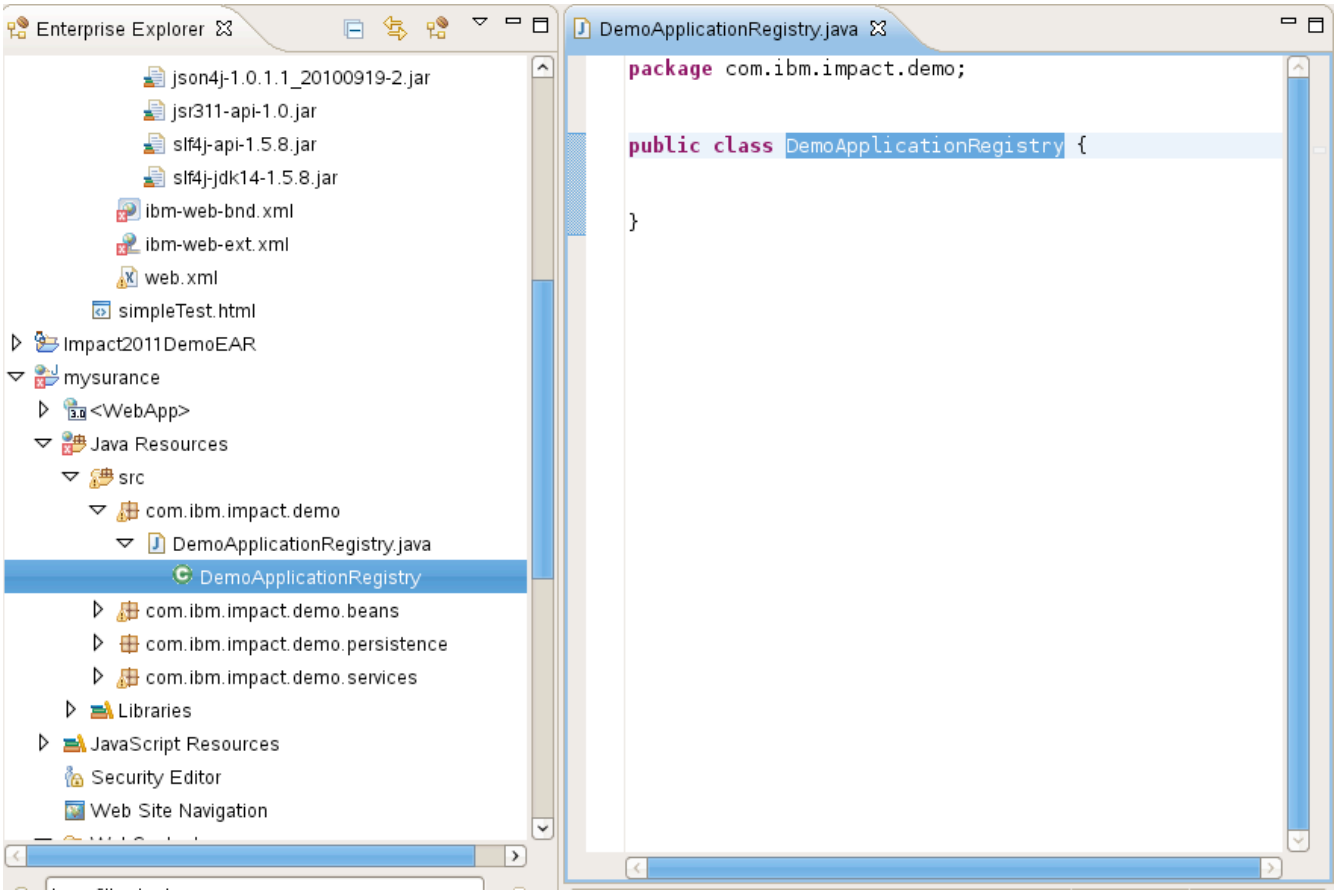

Illustration 14: Template for DemoApplicationRegistry

Now that we have configured via the web.xml the Application instance for the JAX-RS service, we need to define our Application instance. This class will be used later on to register the various JAX-RS resources that we will be demonstrating in this lab.

For now, simply add this empty implementation that overrides the getClasses method and returns an empty set of Class instances. This method will be augmented as we start to implement our custom JAX-RS resources for the mysurance application.

```
package com.ibm.impact.demo;
import java.util.HashSet;
import java.util.Set;
import javax.ws.rs.core.Application;

public class DemoApplicationRegistry extends Application {

	@Override
	public Set<Class<?>> getClasses() {
		Set<Class<?>> classes = new HashSet<Class<?>>();
		return classes;
	}
}
```

After we add this code snippet, the source code editor should look like this.

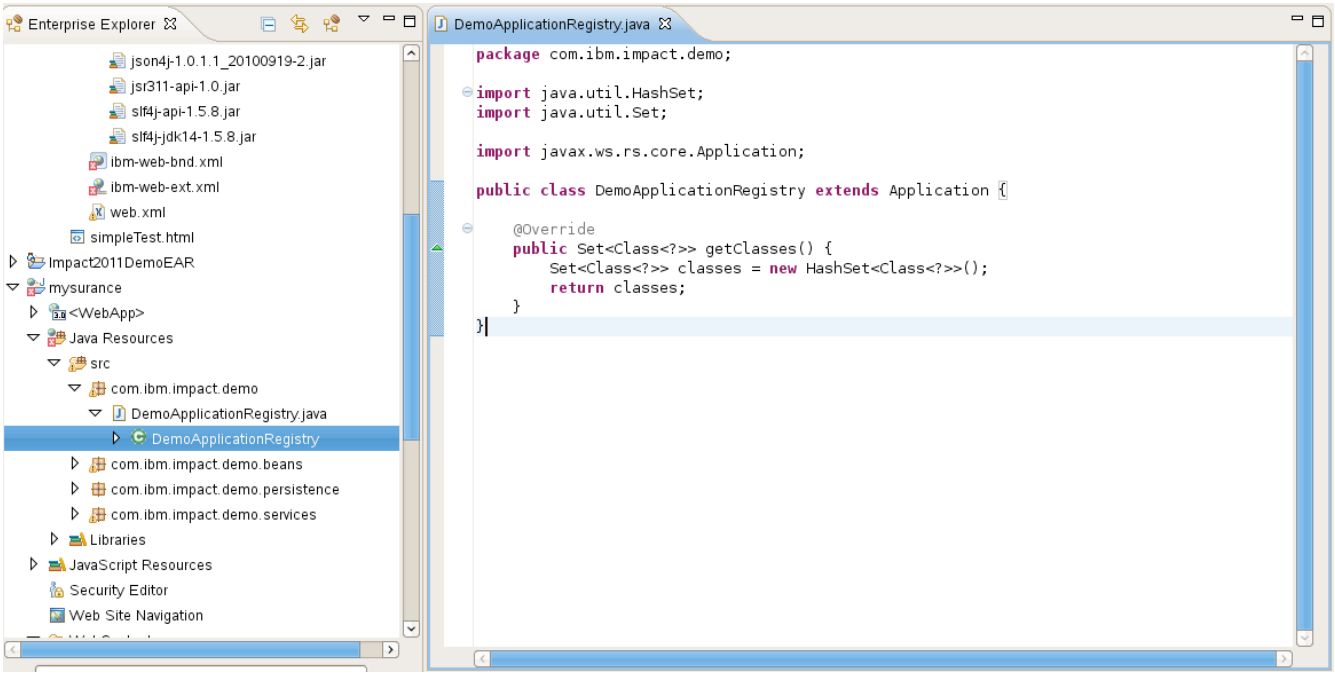

Illustration 15: Adding javax.ws.rs.core.Application support to DemoApplicationRegistry

At this point in time, we should now be ready to write our first JAX-RS service.

1.4 Overview of JAXB

Before we start to leverage JAXB, lets provide the definition and why it is useful.
Java Architecture for XML Binding (JAXB) allows Java developers to map Java classes to XML representations. JAXB provides two main features: the ability to marshal Java objects into XML and the inverse, i.e. to unmarshal XML back into Java objects. In other words, JAXB allows storing and retrieving data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure.

1.4.1 Why did we chose to demonstrate JAXB as data model?

We decided it would be simpler to deal with Plain Old Java Objects (Pojos) when interacting with our services versus having to deal with the wire protocol which is JSON.

1.5 Mysurance User Profile Service

One of the components of the Mysurance application is the notion of a User Profile where the application provides a form that users provider details about themselves such as first name, last name, address and so forth. To help demonstrate how this information could be obtained on a mobile device and then collected by a insurance provider, we implemented a JAX-RS service that is responsible for detailing with everything related to a user's profile.

1.5.1 Creation of a UserProfile Pojo for JAXB

In this section, we will demonstrate the use of JAXB in conjunction with JAX-RS when referencing the the JSON payloads that are passed via our service using JAXB. The JAXB class that we will be working on this in portion of the demo is located in com.ibm.impact.demo.beans.UserProfile. The UserProfile is responsible for maintaining all of the various attributes that we typically associate with a User.

1.5.2 Defining the UserProfile JAXB Root Element

The first step in exposing the Pojo via JAXB is the configuring the root element of the JAXB object and defining the accessor type for the Pojo when it is serialized into a JAXB object. In this first step we will denote the following annotations just above the definition of the UserProfile class that implements ImpactBeanType.

```
@XmlRootElement(name = "UserProfile")
// Every non static, non transient field in a JAXB-bound class
// will be automatically bound to XML, unless annotated by XmlTransient
@XmlAccessorType(XmlAccessType.FIELD)
public class UserProfile implements ImpactBeanType {
```

At this point the class should have the following content (ignore the errors for the moment)

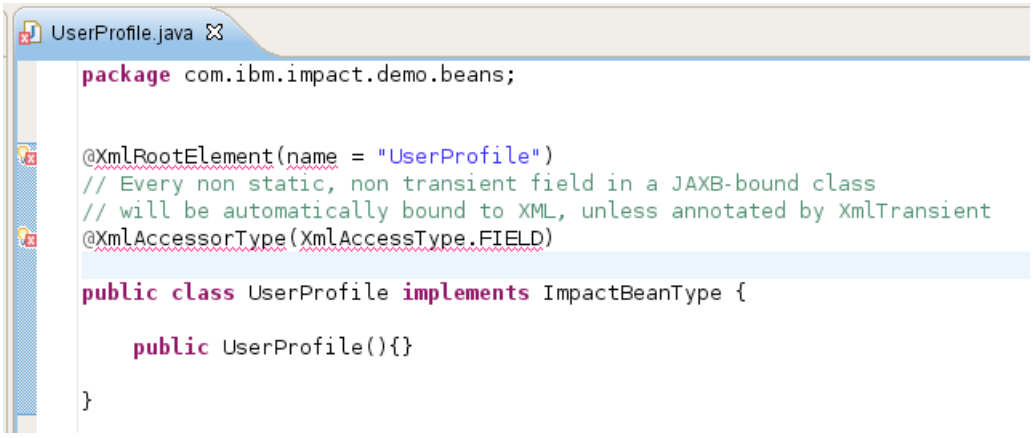

Illustration 16: Annotating UserProfile to be a JAXB type

1.5.3 Defining the UserProfile JAXB elements and attributes

The second step is to define the attributes (attribute for the xml object) and elements (child nodes for the xml object) that define what a UserProfile instance consists of. These attributes will map the JSON keys that are sent in the payload when performing CRUD operations on the UserProfile instance.

For this step, copy and paste the following instance variables and their respective annotations.

```
@XmlAttribute
private String uniqueId;
@XmlElement
private String fName;
@XmlElement
```

```
private String lName;
@XmlElement
 private String street;
@XmlElement
private String city;
@XmlElement
private String state;
@XmlElement
private String zip;
@XmlElement
private String phone;
@XmlElement
private String email;
@XmlElement
private String licenseNo;
```

```
package com.ibm.impact.demo.beans;

import javax.xml.bind.annotation.XmlAccessType;


@XmlRootElement(name = "UserProfile")
// Every non static, non transient field in a JAXB-bound class
// will be automatically bound to XML, unless annotated by XmlTransient
@XmlAccessorType(XmlAccessType.FIELD)

public class UserProfile implements ImpactBeanType {

    @XmlAttribute
    private String uniqueId;
    @XmlElement
    private String fName;
    @XmlElement
    private String lName;
    @XmlElement
    private String street;
    @XmlElement
    private String city;
    @XmlElement
    private String state;
    @XmlElement
    private String zip;
    @XmlElement
    private String phone;
    @XmlElement
    private String email;
    @XmlElement
    private String licenseNo;

    public UserProfile(){}
```

Illustration 17: UserProfile defining the xml attributes and elements associated with this JAXB object

1.5.4 Resolving errors in UserProfile

At this point in time, there should be a set of errors indicating the various types of these annotations cannot be resolved. To remove these errors, simply right mouse click in the editor click on Source → 'Organize Imports'. *As the imports are added, please ensure you select the javax.xml prefixed packages versus the com.sun prefixed packages.* Once this has been done the errors should now be resolved and you should see the following imports at the top of your class.

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

1.5.5 Exposing setting of uniqueId in UserProfile

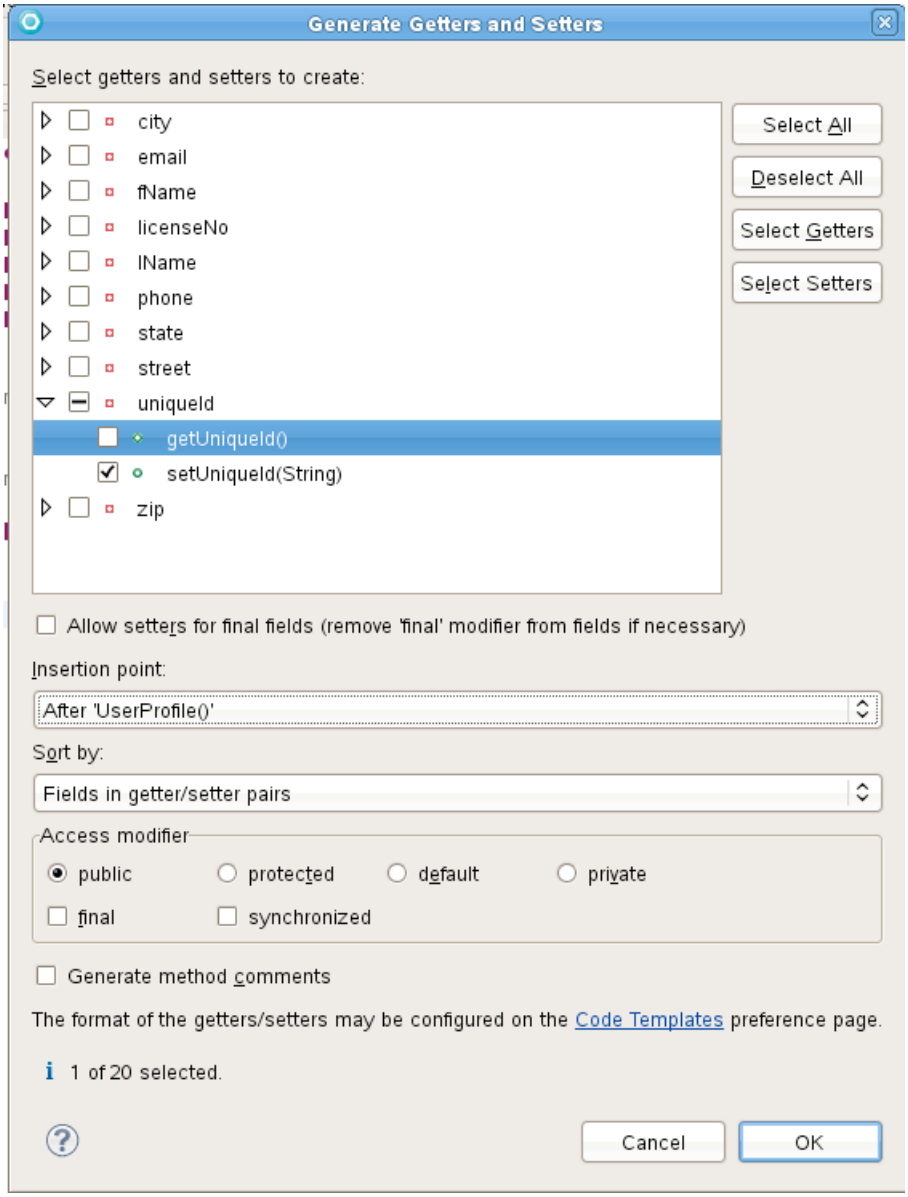

Illustration 18: Wizard step for adding the setter for uniqueId for UserProfile

The last step required for this JAXB instance is to generate the setter method for uniqueId instance variable. This will be used later on in the demo when both creating and updating the UserProfile object. Right mouse click on the uniqueId definition and select Source → 'Generate Getters and Setters...'. Only the setter is required for this demo but having both the getter and setter should not matter in this demonstration. If you want, you can simply uncheck the getter one and then press the ok button. For code clarity, we suggest selecting the 'Insertion Point' of "After 'UserProfile'" and click on

the “OK”.

1.5.6 Confirm actions completed in creating the UserProfile

At this point in time, we can now review the generated code to ensure that we have followed the steps outlined above. Here is what the UserProfile Pojo should look like after adding the class level annotations, added the JAXB attributes and the setter method for uniqueId.

UserProfile.java

```
package com.ibm.impact.demo.beans;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;


@XmlRootElement(name = "UserProfile")
// Every non static, non transient field in a JAXB-bound class
// will be automatically bound to XML, unless annotated by XmlTransient
@XmlAccessorType(XmlAccessType.FIELD)

public class UserProfile implements ImpactBeanType {

    @XmlAttribute
    private String uniqueId;
    @XmlElement
    private String fName;
    @XmlElement
    private String lName;
    @XmlElement
    private String street;
    @XmlElement
    private String city;
    @XmlElement
    private String state;
    @XmlElement
    private String zip;
    @XmlElement
    private String phone;
    @XmlElement
    private String email;
    @XmlElement
    private String licenseNo;

    public UserProfile(){}

    public void setUniqueId(String uniqueId) {
        this.uniqueId = uniqueId;
    }

}
```

Illustration 19: Final version of hte UserProfile JAXB implementation

1.5.7 Creation of a JAX-RS UserProfile service leveraging JAXB

Now that the JAXB Pojo has been created and it's various bean elements have been defined, the JAX-RS service can now reference and serialize and deserialize the JSON payloads into UserProfile instances. The JAX-RS class that we will be working on this in portion of the demo is located in com.ibm.impact.demo.services.UserProfileDataService. Following the model that we have done earlier in the demo, we provide a starting point class to be used to build this service.

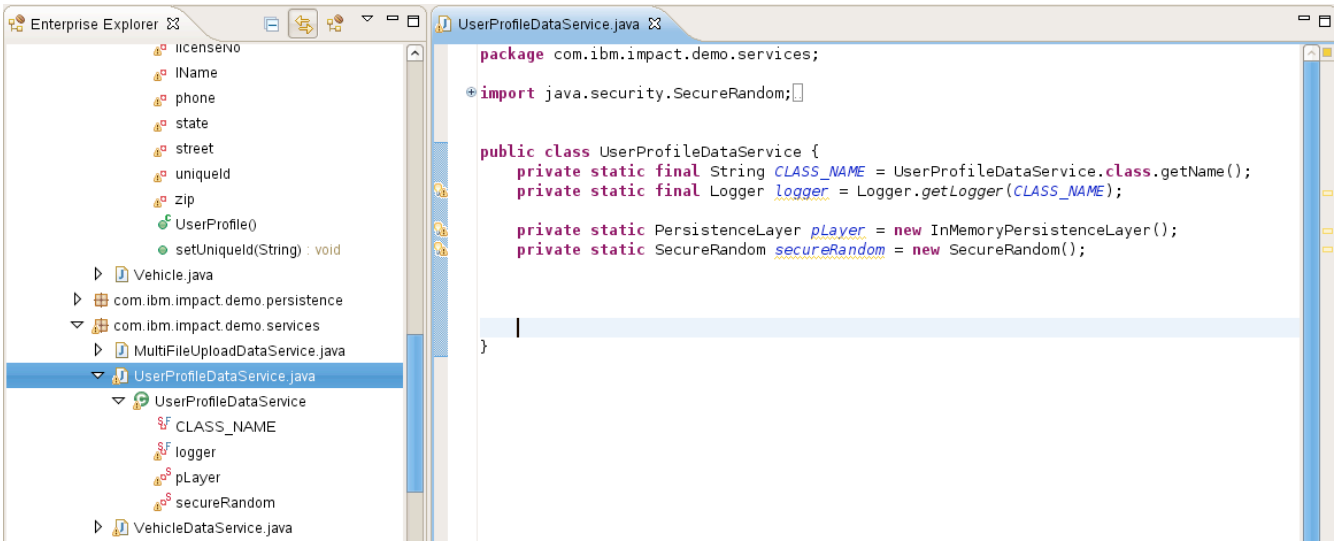

Illustration 20: Template for hte UserProfileDataService starting point

1.5.8 Defining the REST API for the UserProfileDataService

When creating a UserProfile service, the runtime needs to be aware of the REST API in which this JAX-RS will be registered. In this example, we provide an annotation that binds this REST API to the path of '/userProfile'.

To configure this path, the annotation would look like the following

```
@Path("userProfile")
public class UserProfileDataService {
```

This @Path annotation indicates to the JAX-RS runtime that this UserProfileDataService will be accessible via the path of /rest/userProfile. The '/rest' portion of the URI is inherited from the servlet mapping that was defined in the web.xml for the JAX-RS service earlier in this demo.

Illustration 21: Exposing the UserProfileDataService as a JAX-RS Service URI via annotations

1.5.9 Enabling creation of user profiles via JAX-RS

When creating JAX-RS services, developers need to decide which CRUD (create,read,update,delete) operations should be supported for this service. In this example, we will define the create, retrieve and update operations

The first method we will expose is the Create method.

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createUserProfile(final UserProfile userProfile){
	if(logger.isLoggable(Level.INFO)){
		logger.logp(Level.INFO, CLASS_NAME, "createUserProfile",
				"creating userProfile {0}", userProfile);
	}
	Integer uniqueIdInt = secureRandom.nextInt();
	String uniqueId = uniqueIdInt.toString();

	userProfile.setUniqueId(uniqueId);

	pLayer.createEntry(uniqueId, userProfile);
	if(logger.isLoggable(Level.INFO)){
		logger.logp(Level.INFO, CLASS_NAME, "createUserProfile",
			"created userProfile {0} location {1} ",
			new Object[] { userProfile, URI.create("/userProfile/"+ uniqueId) } );
	}

	return Response.created(URI.create("/userProfile/"+ uniqueId)).build();
}
```

The create method has a few interesting annotations that we will describe. The first is the @PUT annotation. The @PUT is the annotation that indicates the HTTP request method that will be used when making REST API calls to this annotation. In following the REST paradigm, PUT will correspond to the creation of a UserProfile action. The next two annotations indicate the payloads for the REST API. The two annotations @Consumes and @Produces correspond to the request

(@Consume) and response (@Produces) format that this REST API supports. Since we are dealing with a Web2.0 style browser client, we have defined JavaScript Object Notation(JSON) as the format that we will be using for communicating with the client. In addition to the annotations, please note the argument that is passed into this method. While JSON is the request payload format, the actual Object that is being passed in as an argument is the UserProfile object created earlier. This allows the developer to abstract away the fact that JSON was being passed as the payload and instead gain the benefits of dealing with a simple Pojo. When this CREATE method is called, the JAX-RS runtime will return a 201 created message to the client with an HTTP location header pointing to where this resource could be retrieved and/or updated.

To remove the errors that are displayed, we again right mouse click "Source" → "Organize Imports". The result will be as follows.

1.5.10 Enabling update of user profiles via JAX-RS

The second method that we will expose is the Update method.

```
@POST
@Path("{uniqueId}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public ImpactBeanType updateUserProfile(@PathParam("uniqueId") String uniqueId,
                final UserProfile userProfile){
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "updateUserProfile",
                                "updateUserProfile userProfile {0} {1}", new Object[]{uniqueId,
userProfile} );
        }
        userProfile.setUniqueId(uniqueId);
        ImpactBeanType entryUpdated= pLayer.updateEntry(uniqueId, userProfile);
                if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "updateUserProfile",
                                "returning updated userProfile uniqueId {0} userProfile {1}",
                                new Object [] { uniqueId, entryUpdated });
        }
         return entryUpdated;
}
```

The Update method contains the same annotations for @Consumes and @Produces as the Create but includes an @Path annotation. The @Path annotation can be used to define additional path elements that could be passed in the REST API. In this example, we defined a @Path element called 'uniqueid'.

The uniqueId is the portion of the URI that denotes that is what we will call the primary key for this UserProfile and is used to retrieve and update the UserProfile. In this example, we will receive both a uniqueId as well as the UserProfile instance that is being updated.

To remove the errors that are displayed, we again right mouse click "Source" → "Organize Imports".

1.5.11 Enabling retrieving of user profiles via JAX-RS

Now that we have added the ability to create and update user profiles, we would like to be able to retrieve these user profiles via REST to ensure that the data has been updated and is available for others

to retrieve. In this section we will define how to retrieve a specific user profile by id or retrieve all user profiles.

1.5.12 Enabling retrieving of user profiles via JAX-RS

Below we have two different methods defined and associated with the HTTP Method GET. The first GET method "getUserProfile" is responsible for retrieving a single user profile based upon the uniqueId that is associated with the user. The return type is ImpactBeanType which the UserProfile object implements. By leveraging JAXB in combination with the @Produces annotation, we can deal with Pojos when retrieving the user profile while still being able to return JSON in the response payload. In the second method "getUserProfiles", there is no uniqueId passed in the request so this API is picked instead of the getUserProfile. As a result, the service will return all user profiles that is aware of as a collection of ImpactBeanTypes. Since this API also denotes a @Produces annotation of JSON, it will transform the collection of JAXB objects into JSON and then render the result back to the client.

```
@GET
@Path("{uniqueId}")
@Produces(MediaType.APPLICATION_JSON)
public ImpactBeanType getUserProfile(@PathParam("uniqueId") String uniqueId){
	if(logger.isLoggable(Level.INFO)){
		logger.logp(Level.INFO, CLASS_NAME, "getUserProfile",
				"get userProfile uniqueId {0}", uniqueId);
	}
	ImpactBeanType userProfile = pLayer.getEntry(uniqueId);
	if(logger.isLoggable(Level.INFO)){
		logger.logp(Level.INFO, CLASS_NAME, "getUserProfile",
				"returning userProfile uniqueId {0} userProfile {1}",
				new Object [] { uniqueId, userProfile });
	}
	if(userProfile == null){
		// can't return a resource that doesn't exist!  Return a 404
		throw new WebApplicationException(Response.Status.NOT_FOUND);
	}
	return userProfile;
}

@GET
@Produces(MediaType.APPLICATION_JSON)
public Collection<ImpactBeanType> getUserProfiles(){
	if(logger.isLoggable(Level.INFO)){
		logger.logp(Level.INFO, CLASS_NAME,
				"getUserProfiles", "entering listing userProfiles");
	}
	Collection<ImpactBeanType> values = (Collection<ImpactBeanType>)
					pLayer.getEntries().values();
	if(logger.isLoggable(Level.INFO)){
		logger.logp(Level.INFO, CLASS_NAME, "getUserProfiles",
				"returning userProfiles {0}", new Object [] { values });
	}
	return values;
}
```

To remove the errors that are displayed, we again right mouse click "Source" → "Organize Imports".

Once these methods have been defined, we should now be ready to register this JAX-RS service

1.5.13 Registering the user profile service with the Application registry.

In this step, we will now register two classes via the Application instance we defined in our web.xml. At this point, navigate to com.ibm.impact.demo.DemoApplicationRegistry and open the file for editing.

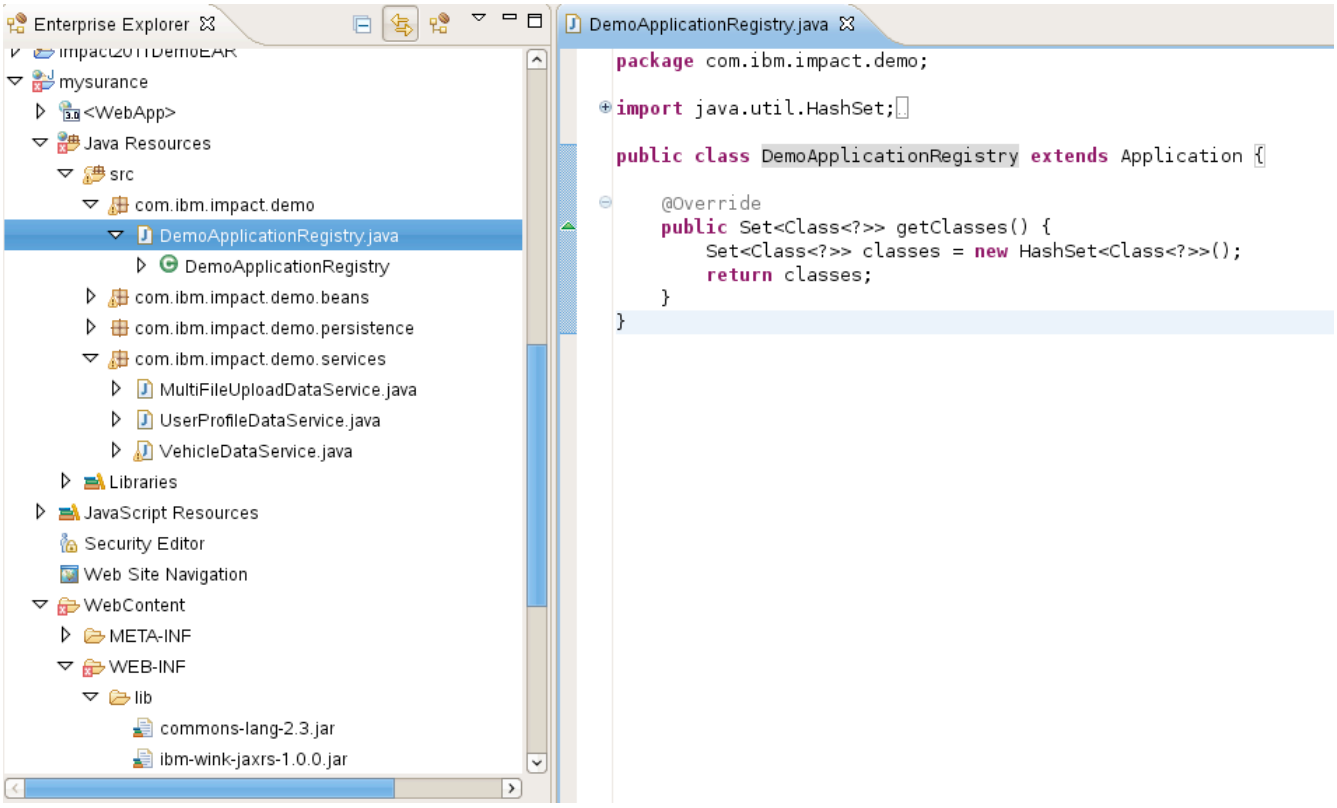

Illustration 22: Starting point for DemoApplicationRegistry

We can now add the following lines to the getClasses method of the this registry instance.

```
classes.add(org.codehaus.jackson.jaxrs.JacksonJaxbJsonProvider.class);
classes.add(UserProfileDataService.class);
```

The first will be our JacksonJaxbJsonProvider class and the second will be our UserProfileDataService that we defined previously. The JacksonJaxbJsonProvider$_2$ provides the support to transform between JSON and JAXB formats used in the UserProfileDataService as well as a service we will define later in the demo. The second class that is registered is the UserProfileDataService. This registration is used by the JAX-RS service to identify the JAX-RS service and makes it available for service via the REST API that is defined as a class level @Path annotation.

To remove the errors that are displayed, we again right mouse click "Source" → "Organize Imports".

We should now have have a Application instance that looks like the following.

DemoApplicationRegistry.java

```
package com.ibm.impact.demo;

import java.util.HashSet;

public class DemoApplicationRegistry extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(org.codehaus.jackson.jaxrs.JacksonJaxbJsonProvider.class);
        classes.add(UserProfileDataService.class);
        return classes;
    }
}
```

Illustration 23: DemoApplicationRegistry after we register JAXB support and UserProfileDataService

1.6 Starting the Mysurance application

To start the mysurance application, right click on the mysurance application, select "Run As" → "Run on Server".

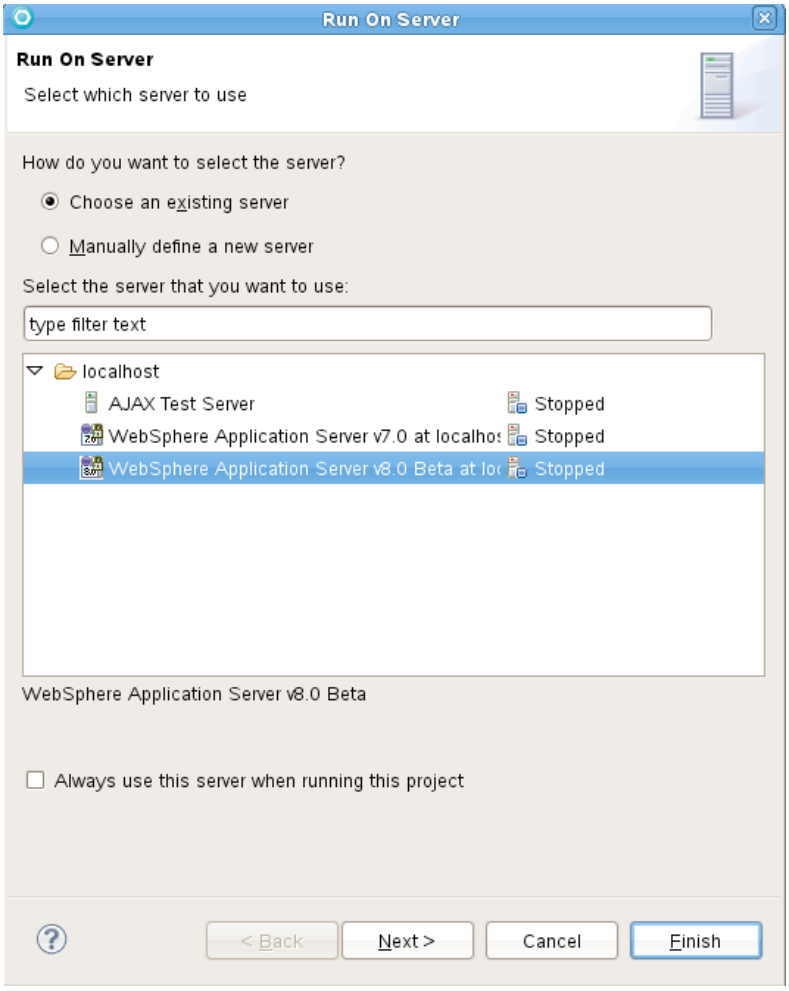

Illustration 24: WAS 8.0 Beta Test Environment

For this demo, we are going to test using the WebSphere Application Server V8.0 Beta test server as follows and then click on “Finish”.

Once the server has been completely initialized and started, the “Servers” will display the following.

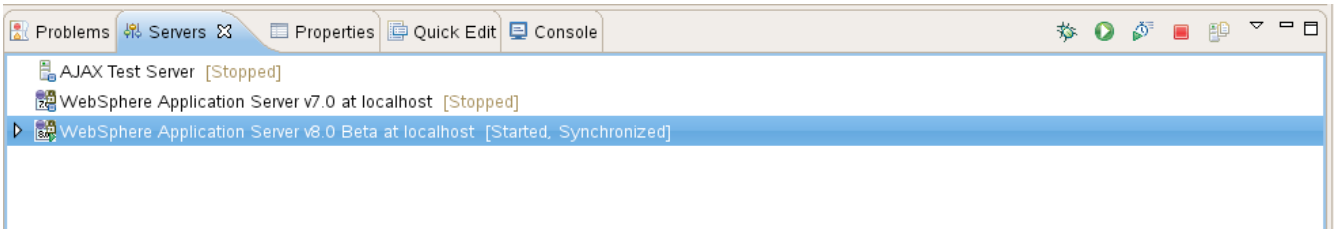

Illustration 25: Servers view showing WAS has started and is ready for requests

If the “Console” view is not currently visible next to “Servers”, select from the top set of menu options “Window” → “Show View” → 'Console”. We will use this view when verifying our REST APIs as we progress through the demo.

1.7 Verifying the UserProfile REST APIs

Now that we have created our various services, we should now test whether the APIs are working as designed. For the UserProfile we are going to cover the create, retrieve and upate scenarios leveraging Poster which we installed earlier in the demo. To begin testing, we are going to go back to Firefox and open “Tools” --> “Poster”.

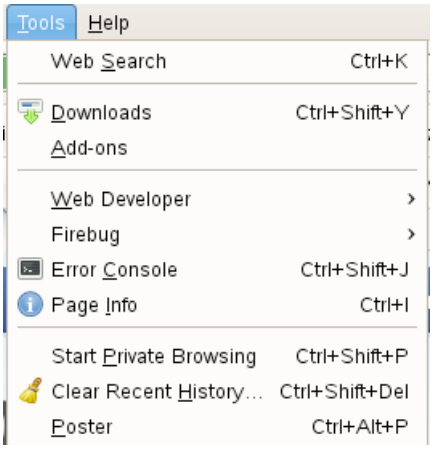

Illustration 26: Firefox menu to open Poster plugin

1.7.1 Demonstrate the create scenario

For the create scenario, we need to provide four sets of data.

```
URL : http://localhost:9081/mysurance/rest/userProfile
Content-type: application/json
Action: PUT (representing a CREATE)
Content to send:
{"fName" : "Todd", "lName" : "Kaplinger", "street": "555 Main Street", "city":
```

```
"Raleigh", "state": "NC", "zip": "27617","phone": "919-555-5555",
"email":"nospam@us.ibm.com","licenseNo": "555-555-5555" }
```

These keys map to the JAXB object defined in the UserProfile JAXB

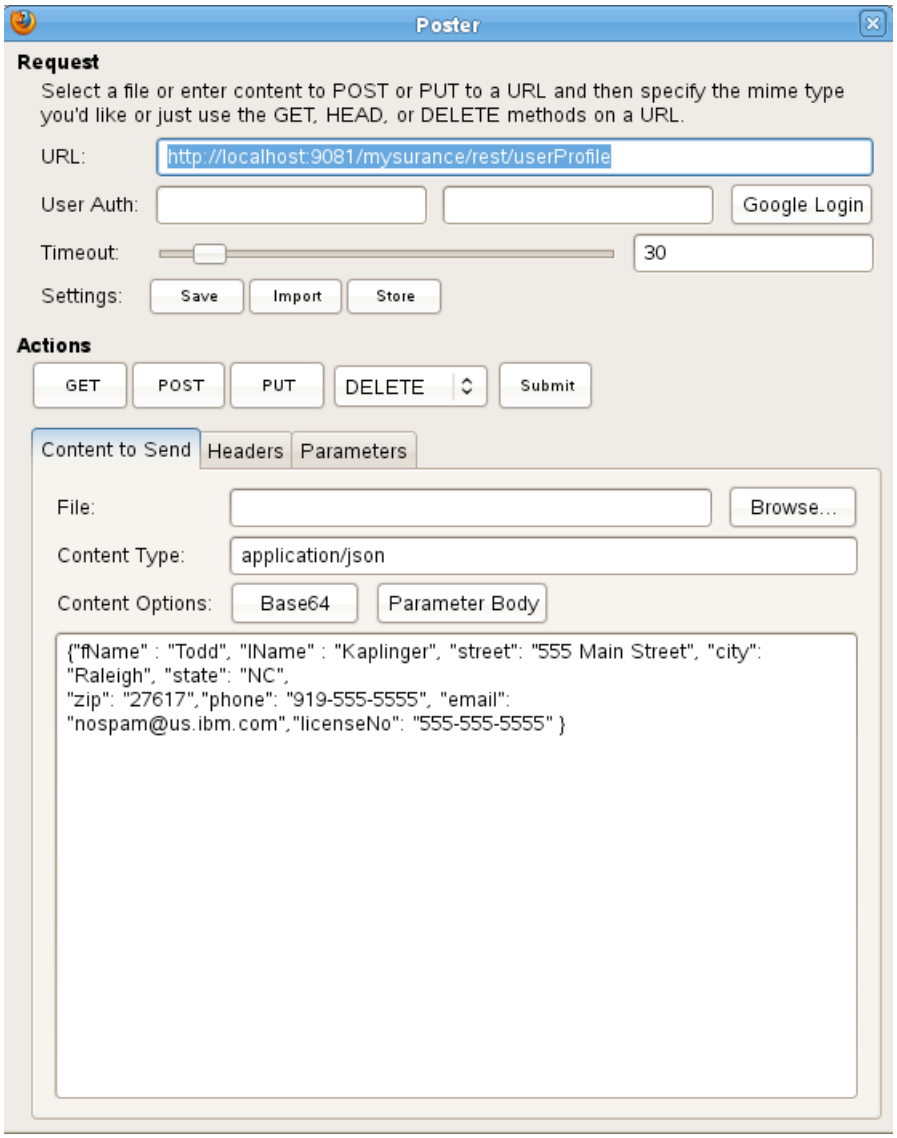

Illustration 27: User Profile CREATE request using Poster

Upon hitting the "Submit", the server will respond back with success or failure. In this instance, we receive the following.

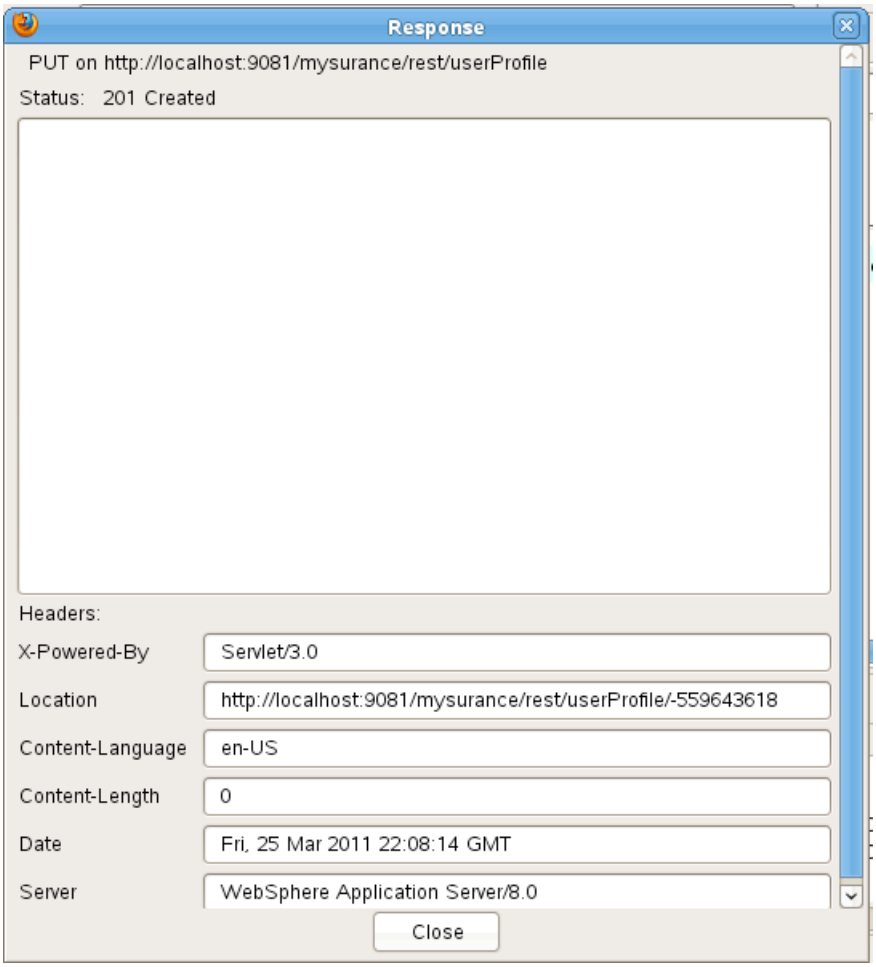

Illustration 28: User Profile CREATE response using Poster

The key item that we are looking for in the create case is the Location Header *(note this ID is autogenerated and will not be the same value when you run your demo scenario)* and the http status code of 201 (created). The location header will provide a reference for retrieving the item in the next section.

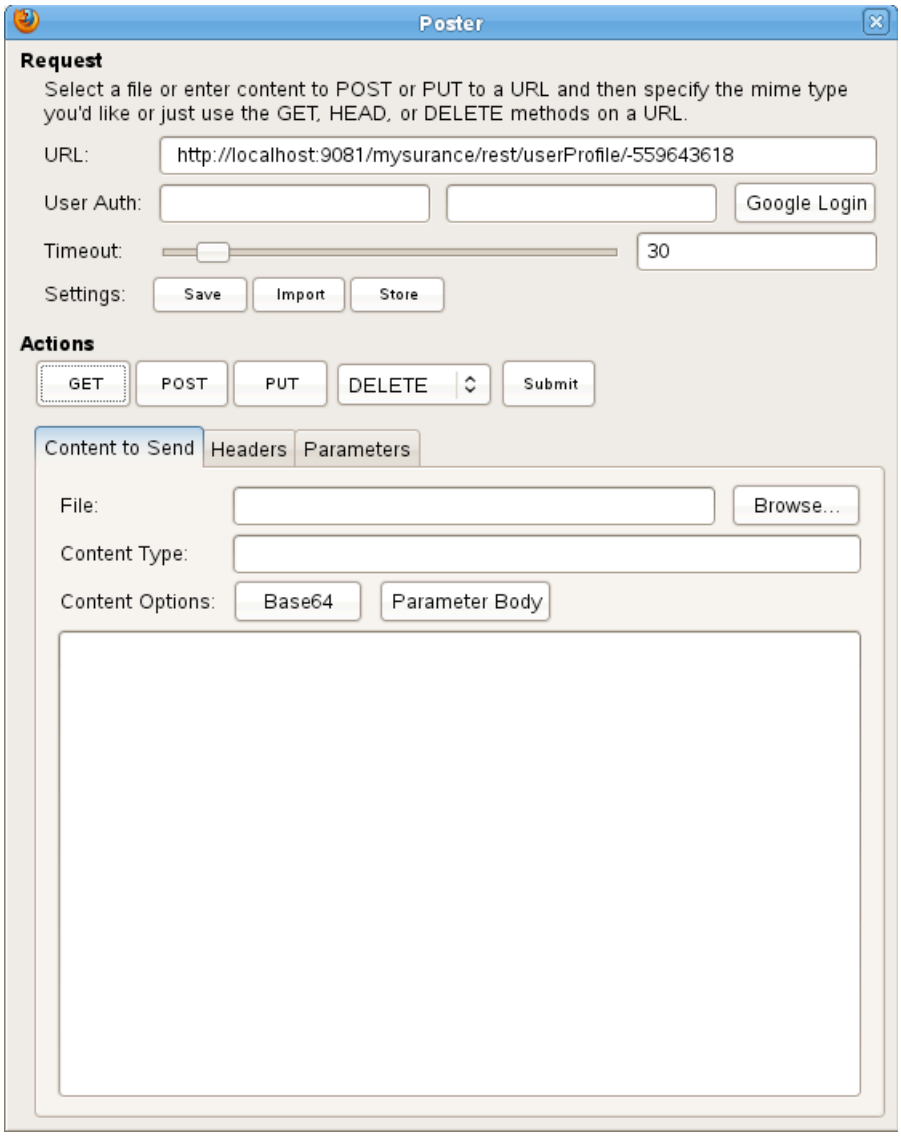

Illustration 29: User Profile RETRIEVE request using Poster

1.7.2 Demonstrate the retrieving scenario

Now that we have done a create, we can either retrieve a specific user profile or list them all. Since there is only one User Profile created so far, the resulting list will both only have one entry. Take the Location header that you received and use that value to retrieve. For this Poster request, we are only interested in two items. The URL to request and the request method of GET. Here is what the Poster request form looks like for this example.

And the associated result.

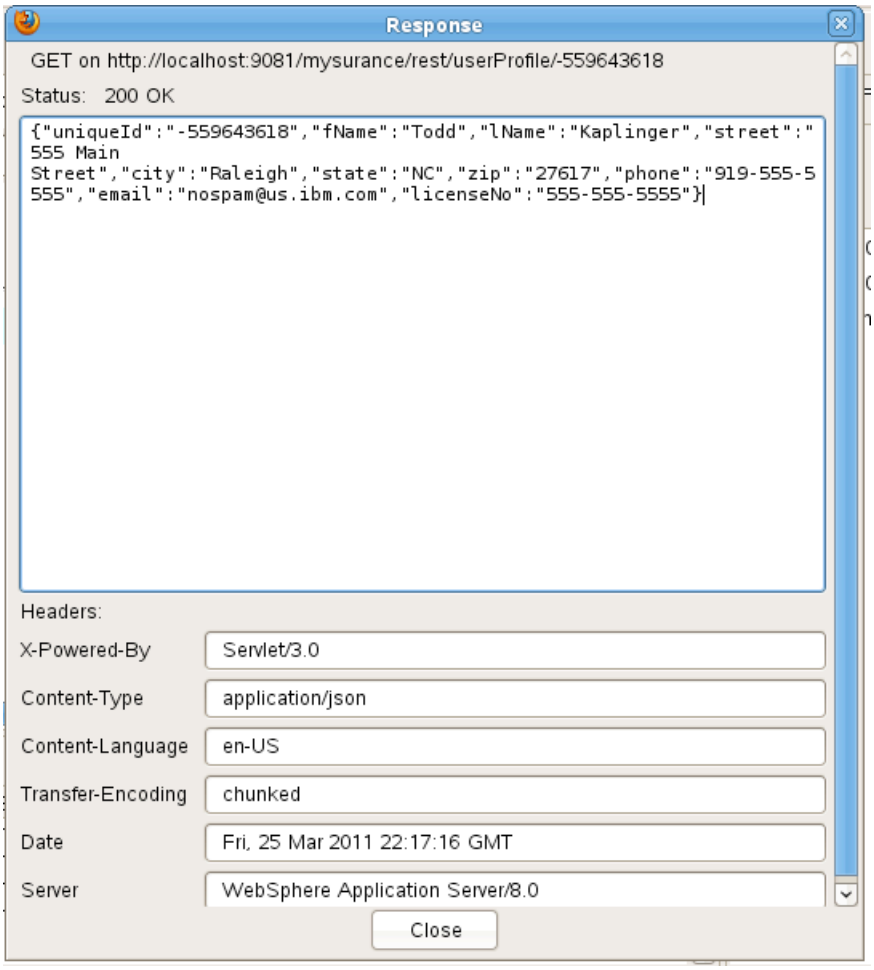

Illustration 30: User Profile RETRIEVE response using Poster

Note the response is a JSON data structure which should be the same data that was sent in the original create with an additional field called uniqueId that should be the same value that was used to retrieve the resource for this example.

In addition the server responded with a 200 status code to notify the client the request was successful. Now that we have shown how to create and retrieve, we are ready to demonstrate how to update an entry.

1.7.3 Demonstrate the update scenario

Using the same user profile that we created and retrieved in the prior two examples, we will now modify this user to change the street address information which this person lives in. For this example the interesting fields are

```
URL:  http://localhost:9081/mysurance/rest/userProfile/-559643618
Request Method:  POST (Update)
Content-type: application/json
```

```
Content to send:
{"uniqueId":"-559643618","fName":"Todd","lName":"Kaplinger","street":"123 Warren
Avenue","city":"Brockton","state":"MA","zip":"02401","phone":"919-555-
5555","email":"nospam@us.ibm.com","licenseNo":"555-555-5555"}
```

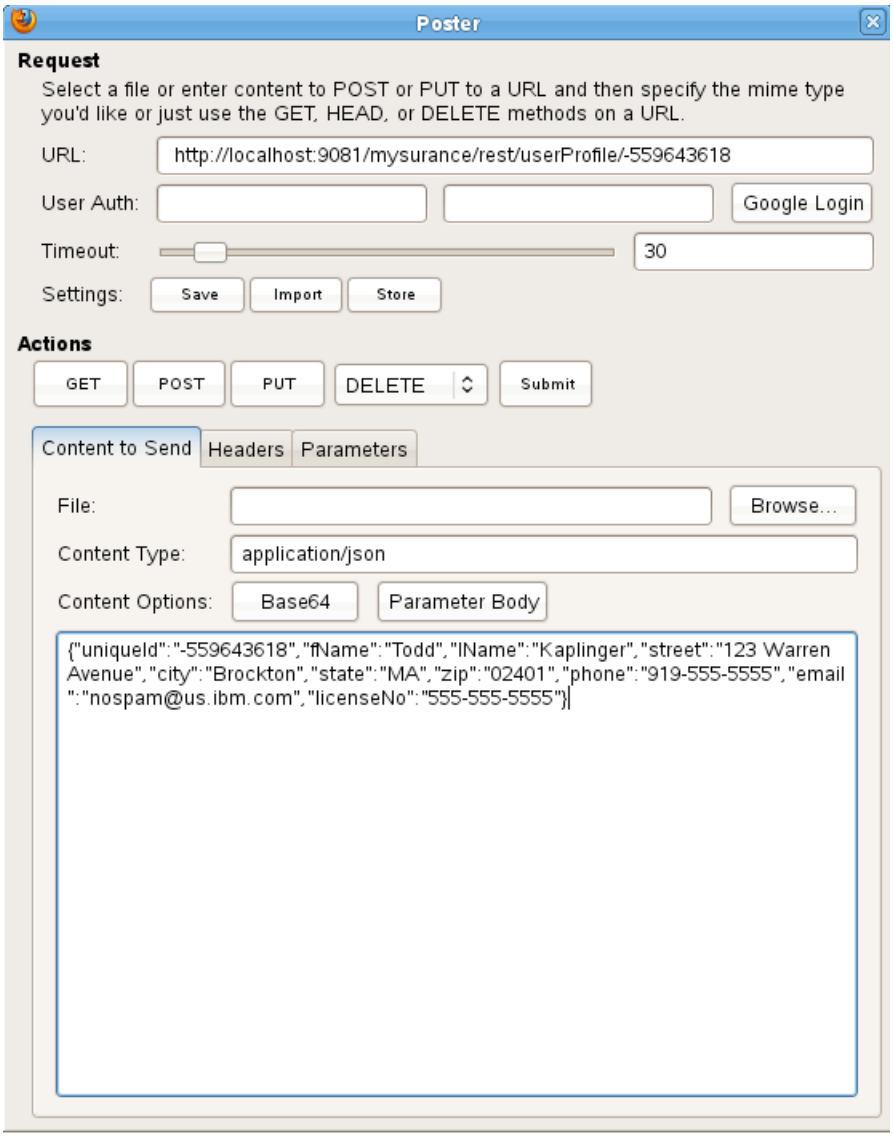

Illustration 31: User Profile UPDATE request using Poster

and the result of this update is

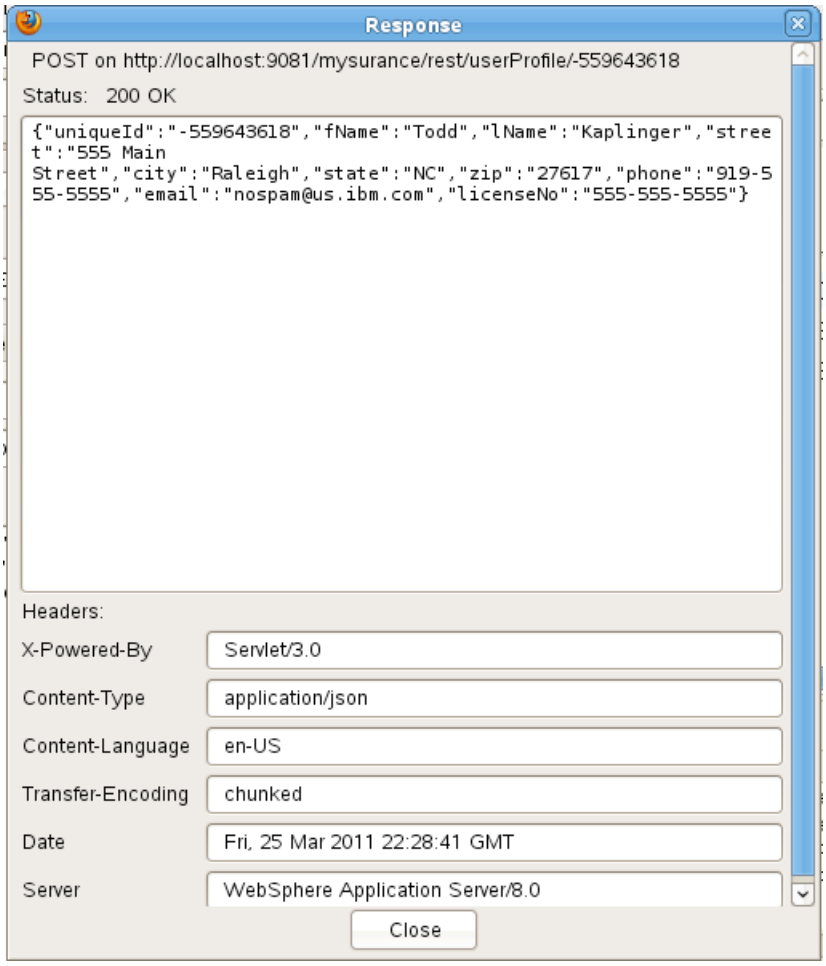

Illustration 32: User Profile UPDATE response using Poster

Note: This update can again be verified by retrieving the REST resource via the same id that was POSTed to this this example but since the UPDATE shows the newly updated content and also provided the client with a 200 status code indicating ok, we skipped this step in the demo steps.

1.8 Mysurance Vehicle Service

Now that we have covered the user profile scenario, we can now move onto another similar but slightly different JAX-RS service that handles Vehicle registration. In this service, we will collect the attributes typically associated with a vehicle registration. Similar to our UserProfile data service, we will again leverage JAXB since we prefer to deal with Pojos in our coding versus having to deal with JSON.

1.8.1 Creation of a Vehicle Pojo for JAXB

Leveraging the knowledge that we acquired in the first JAX-RS service, we will create a second JAX-RS service that leverages JAXB. The JAXB class that we will be working on this in portion of the demo is located in com.ibm.impact.demo.beans.Vehicle. The Vehicle Pojo is similar in function to what we are doing with the UserProfile but the information that is associated with this resource focuses on the actual Vehicle that the owner of this insurance policy.

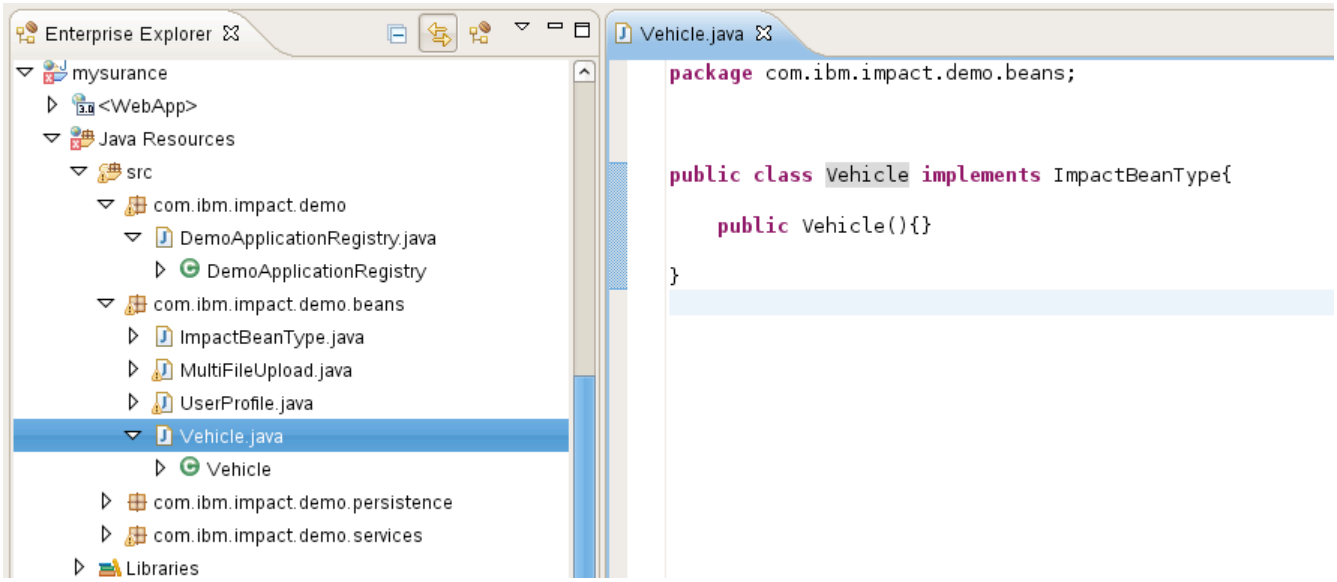

Illustration 33: Template for defining our Vehicle JAXB instance

1.8.2 Defining the UserProfile JAXB Root Element

The first step in exposing the Pojo via JAXB is the configuring the root element of the JAXB object and defining the accessor type for the Pojo when it is serialized into a JAXB object. In this first step we will denote the following annotations just above the definition of the Vehicle class that implements ImpactBeanType.

```
@XmlRootElement(name = "Vehicle")
// Every non static, non transient field in a JAXB-bound class
// will be automatically bound to XML, unless annotated by XmlTransient
@XmlAccessorType(XmlAccessType.FIELD)
public class Vehicle implements ImpactBeanType{
```

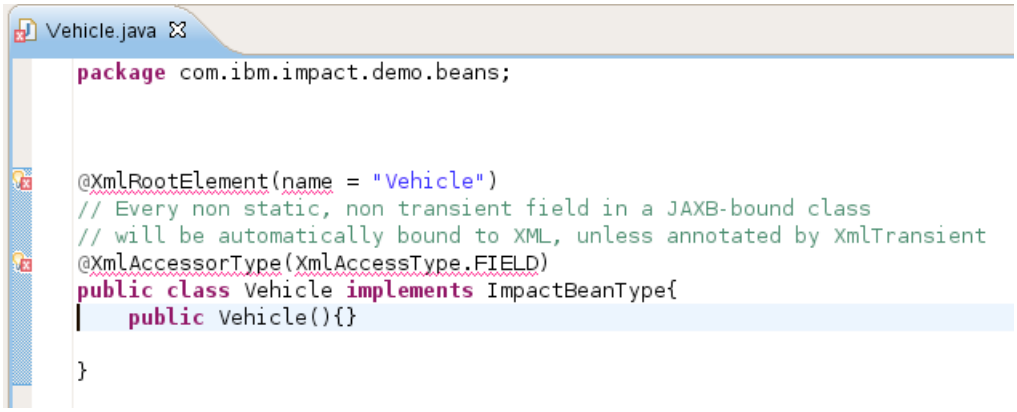

Illustration 34: Annotating our Vehicle Object to be of type JAXB

1.8.3 Defining the Vehicle JAXB elements and attributes

The second step is to define the attributes (attribute for the xml object) and elements (child nodes for the xml object) that define what a Vehicle instance consists of. For this step, copy and paste the following instance variables and their respective annotations.

```
@XmlAttribute
private String uniqueId;
@XmlElement
private String vName;
@XmlElement
private String vMake;
@XmlElement
private String vModel;
@XmlElement
private String vYear;
@XmlElement
private String vPlate;
@XmlElement
private String vNumber;
```

1.8.4 Resolving errors in Vehicle

At this point in time, there should be a set of errors indicating the various types of these annotations cannot be resolved. To remove these errors, simply right mouse click in the editor click on Source → 'Organize Imports'. Once this has been done the errors should now be resolved and you should see the following imports at the top of your class.

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

Vehicle.java

```
package com.ibm.impact.demo.beans;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;



@XmlRootElement(name = "Vehicle")
// Every non static, non transient field in a JAXB-bound class
// will be automatically bound to XML, unless annotated by XmlTransient
@XmlAccessorType(XmlAccessType.FIELD)
public class Vehicle implements ImpactBeanType{
    @XmlAttribute
    private String uniqueId;
    @XmlElement
    private String vName;
    @XmlElement
    private String vMake;
    @XmlElement
    private String vModel;
    @XmlElement
    private String vYear;
    @XmlElement
    private String vPlate;
    @XmlElement
    private String vNumber;

    public Vehicle(){}

}
```

Illustration 35: Defining the xml elements and attributes for our Vehicle JAXB instance

1.8.5 Exposing setting of uniqueId in Vehicle

The last step required for this JAXB instance is to generate the setter method for uniqueId instance variable. This will be used later on in the demo when both creating and updating the Vehicle object. Right mouse click on the uniqueId definition and select Source → 'Generate Getters and Setters...'. Only the setter is required for this demo but having both the getter and setter should not matter in this demonstration. If you want, you can simply uncheck the getter one and then press the ok button.

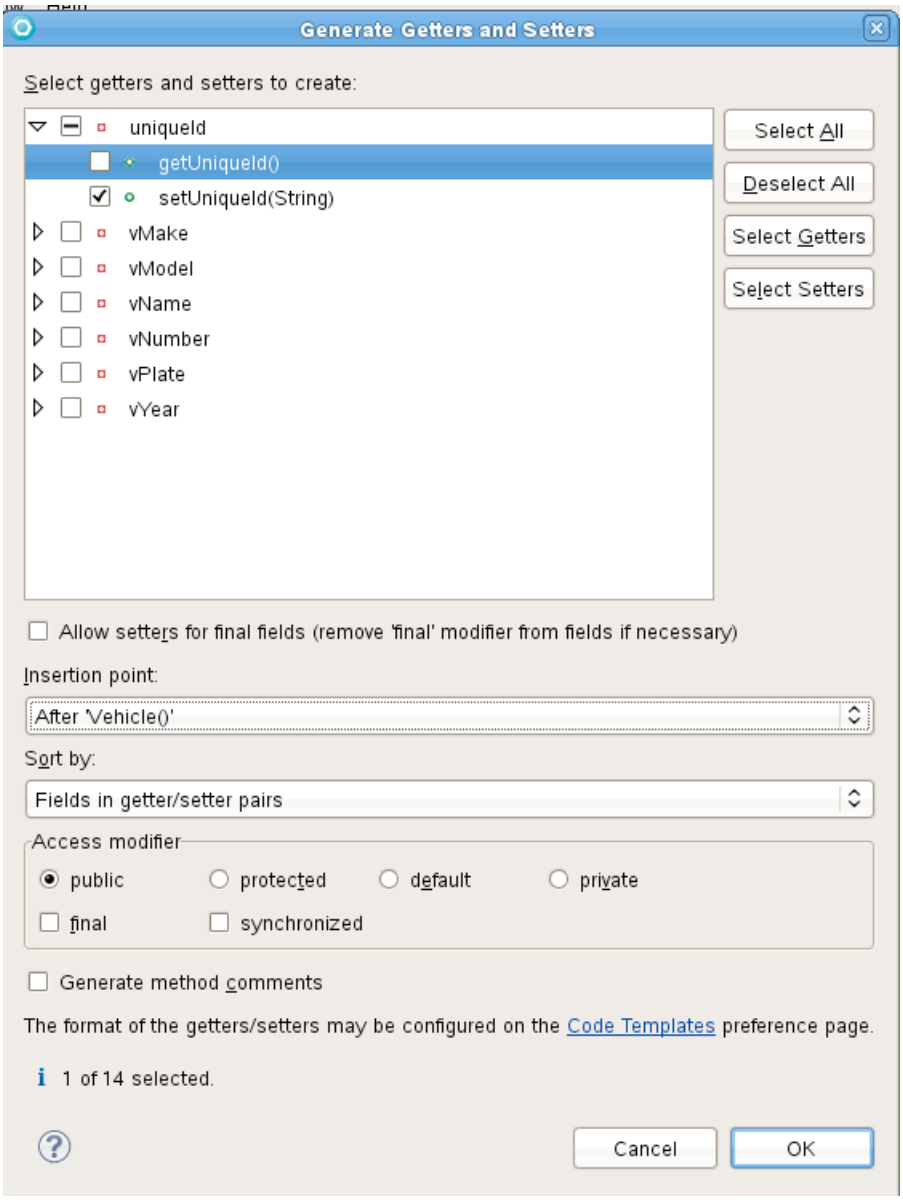

Illustration 36: Wizard for adding setter method for our uniqueId attribute in Vehicle

Vehicle.java

```java
package com.ibm.impact.demo.beans;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;



@XmlRootElement(name = "Vehicle")
// Every non static, non transient field in a JAXB-bound class
// will be automatically bound to XML, unless annotated by XmlTransient
@XmlAccessorType(XmlAccessType.FIELD)
public class Vehicle implements ImpactBeanType{
    @XmlAttribute
    private String uniqueId;
    @XmlElement
    private String vName;
    @XmlElement
    private String vMake;
    @XmlElement
    private String vModel;
    @XmlElement
    private String vYear;
    @XmlElement
    private String vPlate;
    @XmlElement
    private String vNumber;

    public Vehicle(){}

    public void setUniqueId(String uniqueId) {
        this.uniqueId = uniqueId;
    }

}
```

Illustration 37: Final version of our Vehicle JAXB instance

1.8.7 Creation of a JAX-RS Vehicle service leveraging JAXB

Now that the JAXB Pojo has been created and it's various bean elements have been defined, the JAX-RS service can now reference and serialize and deserialize the JSON payloads into Vehicle instances. The JAX-RS class that we will be working on this in portion of the demo is located in com.ibm.impact.demo.services.VehicleDataService.

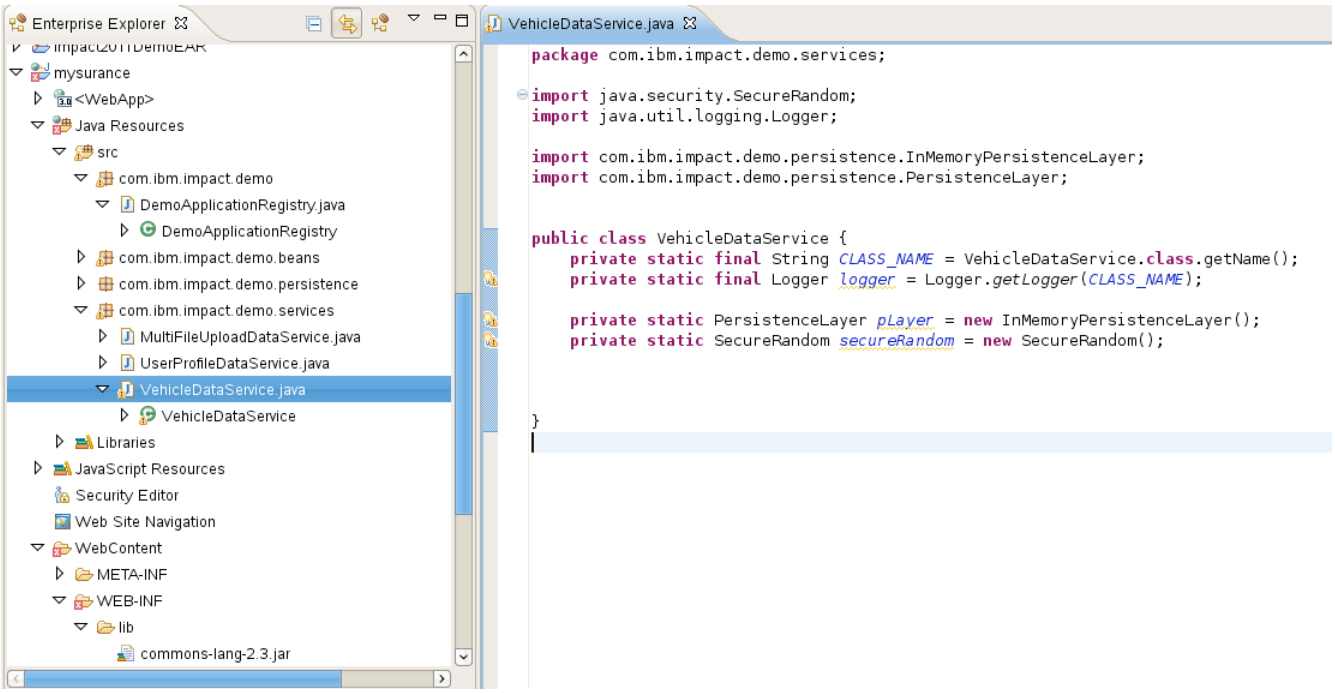

Illustration 38: Template for our VehicleDataService JAX-RS service

1.8.8 Defining the REST API for the UserProfileDataService

When creating a Vehicle service, the runtime needs to be aware of the REST API in which this JAX-RS will be registered. In this example, we provide an annotation that binds this REST API to the path of '/vehicle'. To configure this path, the annotation would look like the following

```
@Path("vehicle")
public class VehicleDataService {
```

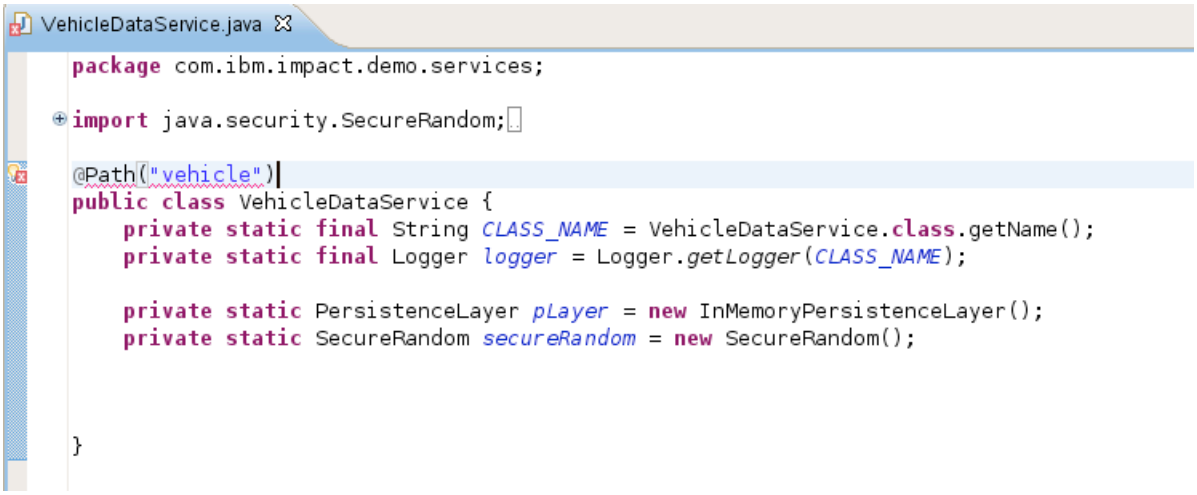

Illustration 39: Exposing our VehicleDataService as a JAX-RS service using annotations

This @Path annotation indicates to the JAX-RS runtime that this VehicleDataService will be accessible

via the path of /rest/userProfile. The '/rest' portion of the URI is inherited from the servlet mapping that was defined in the web.xml for the JAX-RS service earlier in this demo.

1.8.9 Enabling creation of vehicles via JAX-RS

When creating JAX-RS services, developers need to decide which CRUD (create,read,update,delete) operations should be supported for this service.

The first method we will expose is the Create method.

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createVehicle(final Vehicle vehicle){
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "createVehicle",
                                "creating vehicle {0}", vehicle);
        }
        Integer uniqueIdInt = secureRandom.nextInt();
        String uniqueId = uniqueIdInt.toString();
        vehicle.setUniqueId(uniqueId);

        pLayer.createEntry(uniqueId, vehicle);

        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "createVehicle",
                        "created vehicle {0} location {1} ",
                        new Object[] { vehicle, URI.create("/vehicle/"+ uniqueId) } );
        }

        return Response.created(URI.create("/vehicle/"+ uniqueId)).build();
}
```

The create method has a few interesting annotations that we will describe. The first is the @PUT annotation. The @PUT is the annotation that indicates the HTTP request method that will be used when making REST API calls to this annotation. In following the REST paradigm, PUT will correspond to the creation of a Vehicle action.

The next two annotations indicate the payloads for the REST API. The two annotations @Consumes and @Produces correspond to the request (@Consume) and response (@Produces) format that this REST API supports. Since we are dealing with a Web2.0 style browser client, we have defined JavaScript Object Notation(JSON) as the format that we will be using for communicating with the client.

In addition to the annotations, please note the argument that is passed into this method. While JSON is the request payload format, the actual Object that is being passed in as an argument is the Vehicle object created earlier. This allows the developer to abstract away the fact that JSON was being passed as the payload and instead gain the benefits of dealing with a simple Pojo. When this CREATE method is called, the JAX-RS runtime will return a 201 created message to the client with an HTTP location header pointing to where this resource could be retrieved and/or updated. The updated scenario will be the focus of the next step in the demo.

1.8.10 Enabling update of vehicles via JAX-RS

The second method that we will expose is the Update method.

```
@POST
@Path("{uniqueId}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public ImpactBeanType updateVehicle(@PathParam("uniqueId") String uniqueId,
                final Vehicle vehicle){
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "updateVehicle",
                                "updateVehicle vehicle {0} {1}", new Object[]{uniqueId, vehicle} );
        }
        vehicle.setUniqueId(uniqueId);
        ImpactBeanType entryUpdated= pLayer.updateEntry(uniqueId, vehicle);
                if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "updateVehicle",
                                "returning updated vehicle uniqueId {0} updateVehicle {1}",
                                new Object [] { uniqueId, entryUpdated });
        }
        return entryUpdated;
}
```

The Update method contains the same annotations for @Consumes and @Produces as the Create but includes an @Path annotation. The @Path annotation can be used to define additional path elements that could be passed in the REST API. In this example, we defined a @Path element called 'uniqueid'. The uniqueId is the portion of the URI that denotes that is what we will call the primary key for this Vehicle and is used to retrieve and update the Vehicle. In this example, we will receive both a uniqueId as well as the Vehicle instance that is being updated.

1.8.11 Enabling retrieving of vehicles via JAX-RS

Now that we have added the ability to create and update vehicles, we would like to be able to retrieve these vehicles via REST to ensure that the data has been updated and is available for others to retrieve. In this section we will define how to retrieve a specific vehicle by id or retrieve all user profiles.

1.8.12 Enabling retrieving of user profiles via JAX-RS

Below we have two different methods defined and associated with the HTTP Method GET. The first GET method “getVehicle” is responsible for retrieving a single user profile based upon the uniqueId that is associated with the user. The return type is ImpactBeanType which the UserProfile object implements. By leveraging JAXB in combination with the @Produces annotation, we can deal with Pojos when retrieving the user profile while still being able to return JSON in the response payload.

In the second method “getVehicles”, there is no uniqueId passed in the request so this API is picked instead of the getVehicle. As a result, the service will return all vehicles that is aware of as a collection of ImpactBeanType's. Since this API also denotes a @Produces annotation of JSON, it will transform the collection of JAXB objects into JSON and then render the result back to the client.

```
@GET
@Path("{uniqueId}")
@Produces(MediaType.APPLICATION_JSON)
```

```
public ImpactBeanType getVehicle(@PathParam("uniqueId") String uniqueId){
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME,
                                "getVehicle", "get vehicle uniqueId {0}", uniqueId);
        }
        ImpactBeanType vehicle = pLayer.getEntry(uniqueId);
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "getVehicle",
                                "returning uniqueId {0} vehicle {1}",
                                new Object [] {uniqueId, vehicle });
        }
        if(vehicle == null){
                // can't return a resource that doesn't exist!  Return a 404
                throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        return vehicle;
}

@GET
@Produces(MediaType.APPLICATION_JSON)
public Collection<ImpactBeanType> getVehicles(){
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME,
                                "getVehicles", "listing vehicles");
        }
        Collection<ImpactBeanType> values = pLayer.getEntries().values();
        if(logger.isLoggable(Level.INFO)){
                logger.logp(Level.INFO, CLASS_NAME, "getVehicles",
                                "returning vehicles {0}", new Object [] { values });
        }
        return values;
}
```

At this point in time, there should be a set of errors indicating the various types of these annotations cannot be resolved. To remove these errors, simply right mouse click in the editor click on Source → 'Organize Imports'. Once this has been done the errors should now be resolved .

Now that these methods have been defined, we should now be ready to register this JAX-RS service

1.8.13 Registering the vehicle service with the Application registry.

In this step, we will again register with the application service this newly updated JAX-RS service. Add the following line just after the previously registered classes

```
classes.add(VehicleDataService.class);
```

At this point in time, there should be a set of errors indicating the various types of these annotations cannot be resolved. To remove these errors, simply right mouse click in the editor click on Source → 'Organize Imports'.

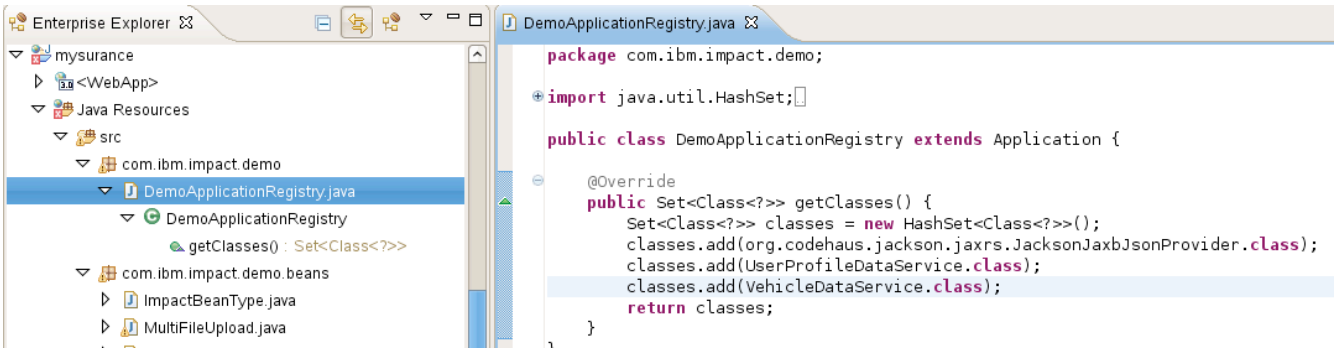

Illustration 40: DemoApplicationRegistry after registering VehicleDataService

1.9 Verifying the Vehicle REST API

Similar to what we did for User Profile in the previous section, we will again use the Firefox plugin Poster to verify our newly created REST APIs.

1.9.1 Demonstrate the create scenario

Following the same model that was done for the User Profile we will identify the key components for creating a Vehicle via the JAX-RS service.

```
URL : http://localhost:9081/mysurance/rest/vehicle
Content-type: application/json
Action: PUT (representing a CREATE)
Content to send:
{ "vName": "MySportsCar", "vMake": "BMW", "vModel": "3 Series", "vYear":
"2011","vPlate": "MyVanityPlate", "vNumber": "1234567890"}
```

These keys map to the JAXB object defined in the Vehicle JAXB

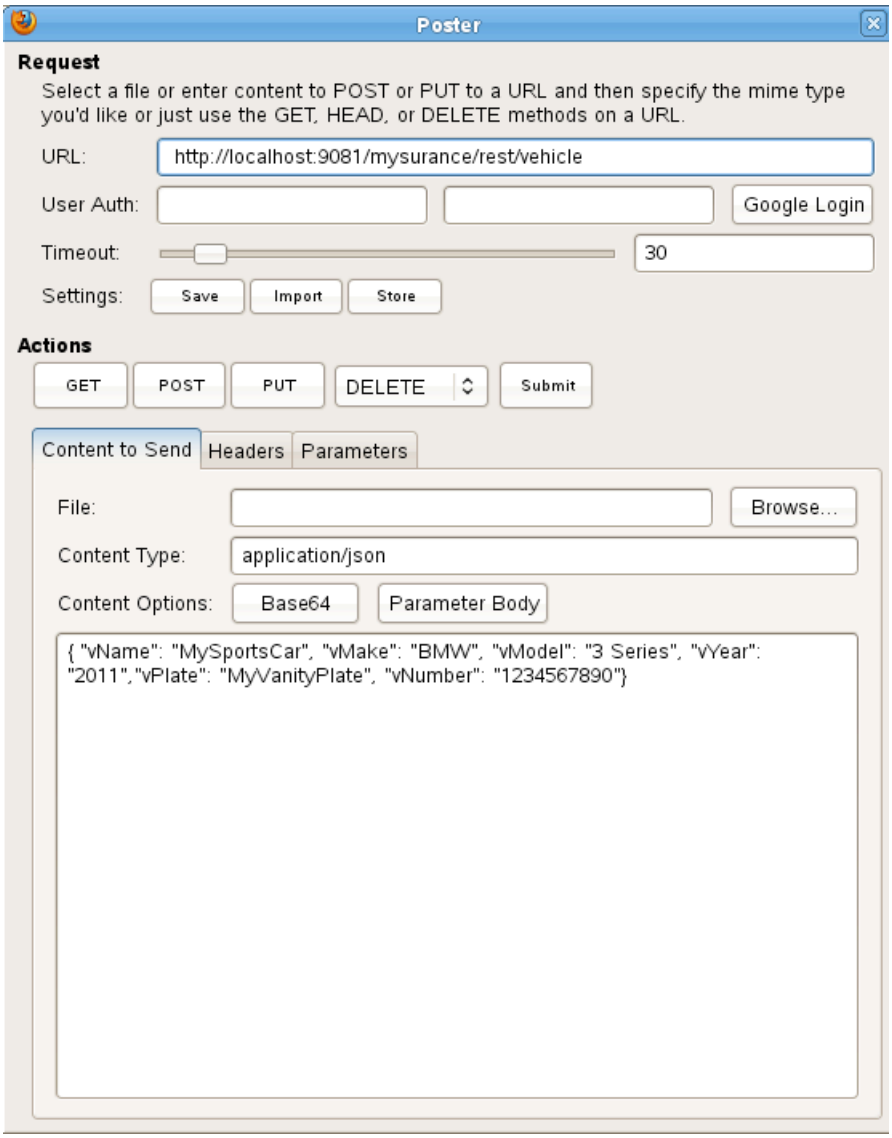

Illustration 41: Vehicle CREATE request using Poster

The result of this CREATE call is the following...

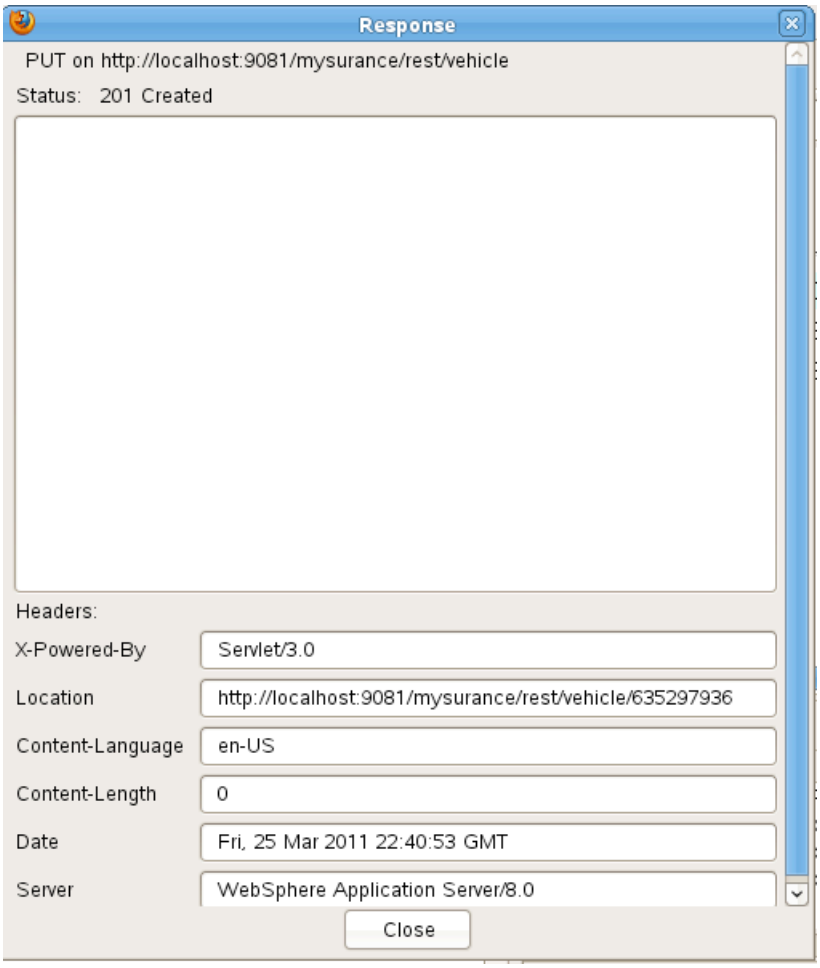

Illustration 42: Vehicle CREATE response using Poster

This response indicates the CREATE was successful (status code of 201) and the location where this vehicle can be referenced for retrieving.

1.9.2 Demonstrate retrieving of a vehicle

We will now make a GET request to the server to retrieve this newly created Vehicle. This will validate the create actually completed.

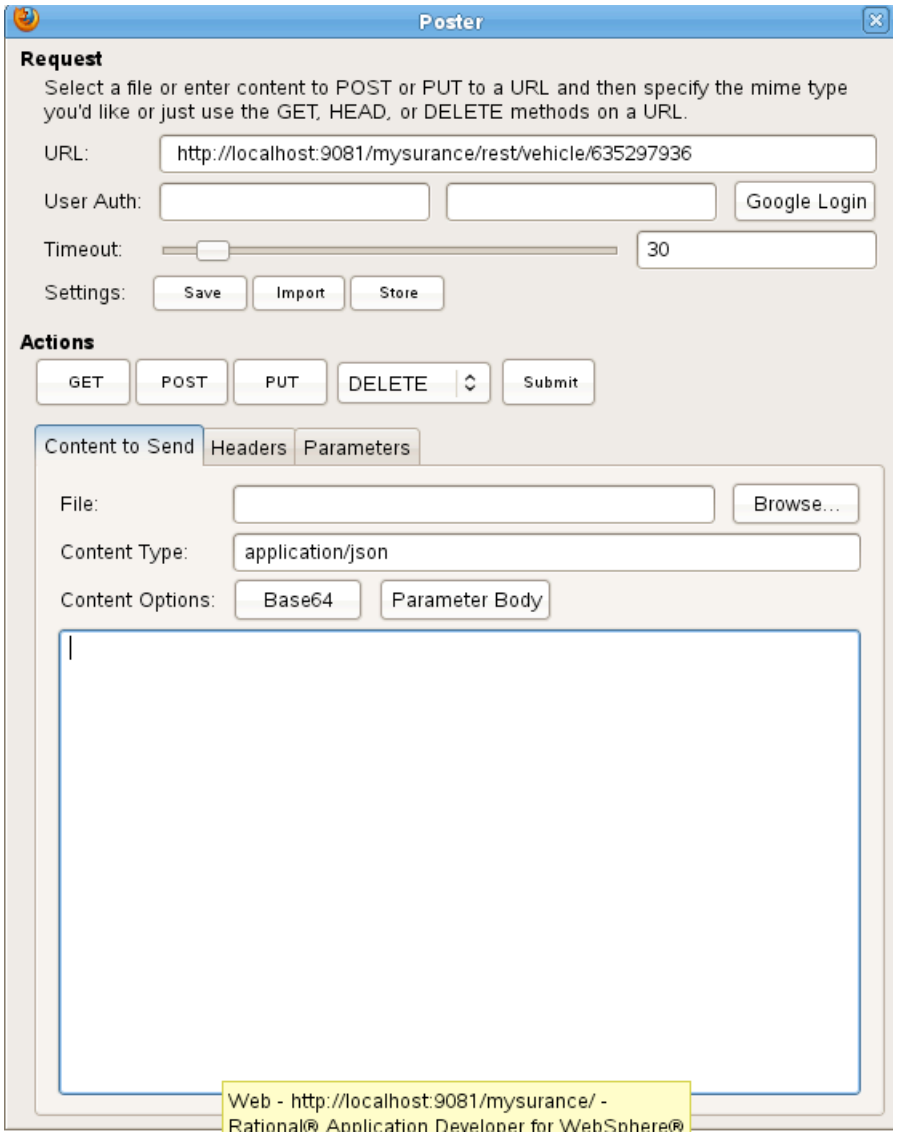

Illustration 43: Vehicle RETRIEVE request using Poster

After hitting submit for this GET request, we can now validate that the entry does indeed exist.

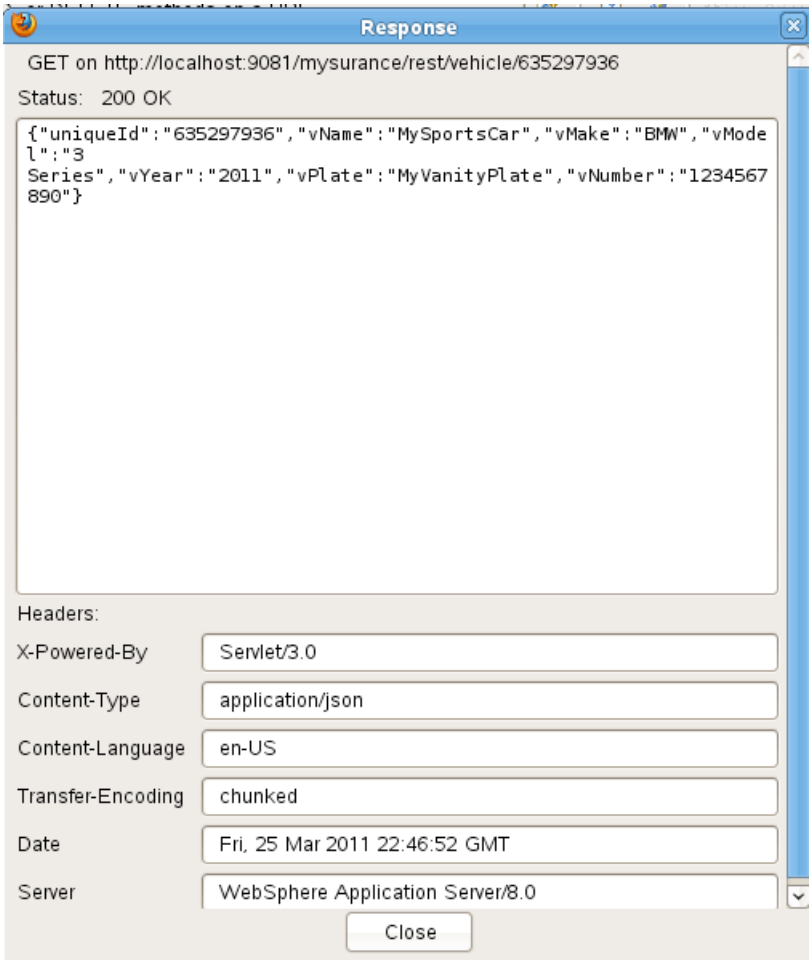

Illustration 44: Vehicle RETRIEVE response using Poster

The response from the server is what we expected. A 200 status code to indicate the resource was found and the appropriate Vehicle that we created earlier was returned.

At this point in time, you can also test the scenario where the resource does not exist. Instead of passing the uniqueId as the last parameter in the URL, add some bogus value instead. The server should return a 404.

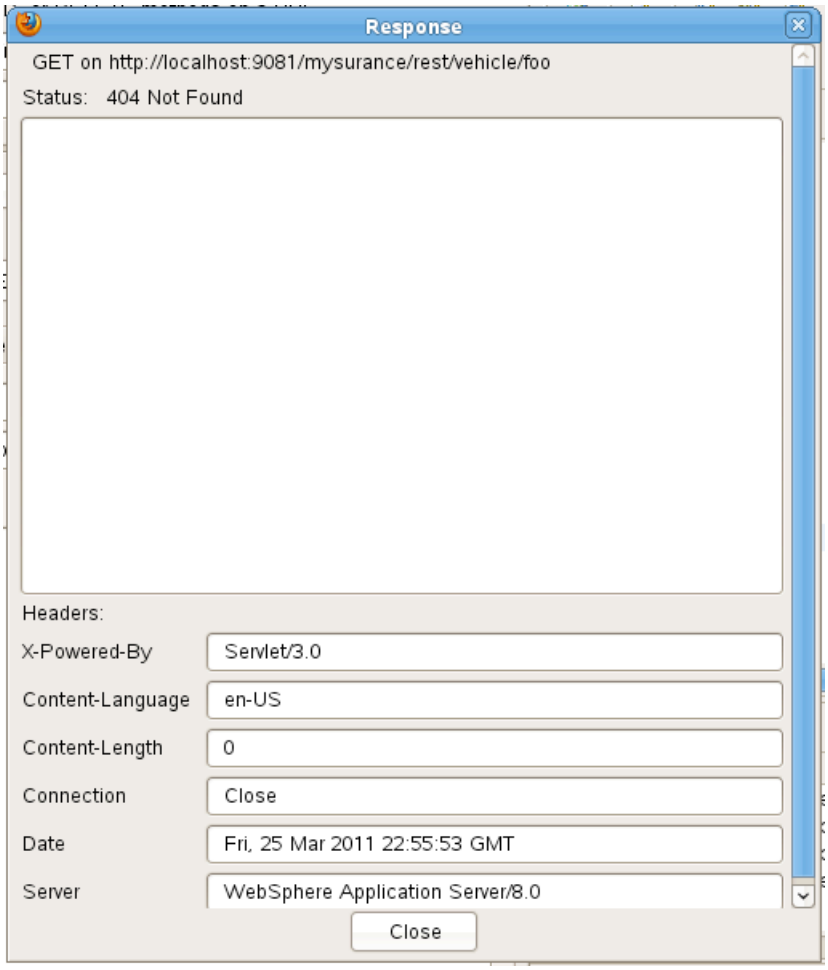

Illustration 45: Vehicle RETRIEVE response failure case using Poster

1.9.3 Demonstrate the update scenario

The final test for this Vehicle JAX-RS service is the update scenario. In this case, we are changing our license plate and want to let our insurance company know this is being done.

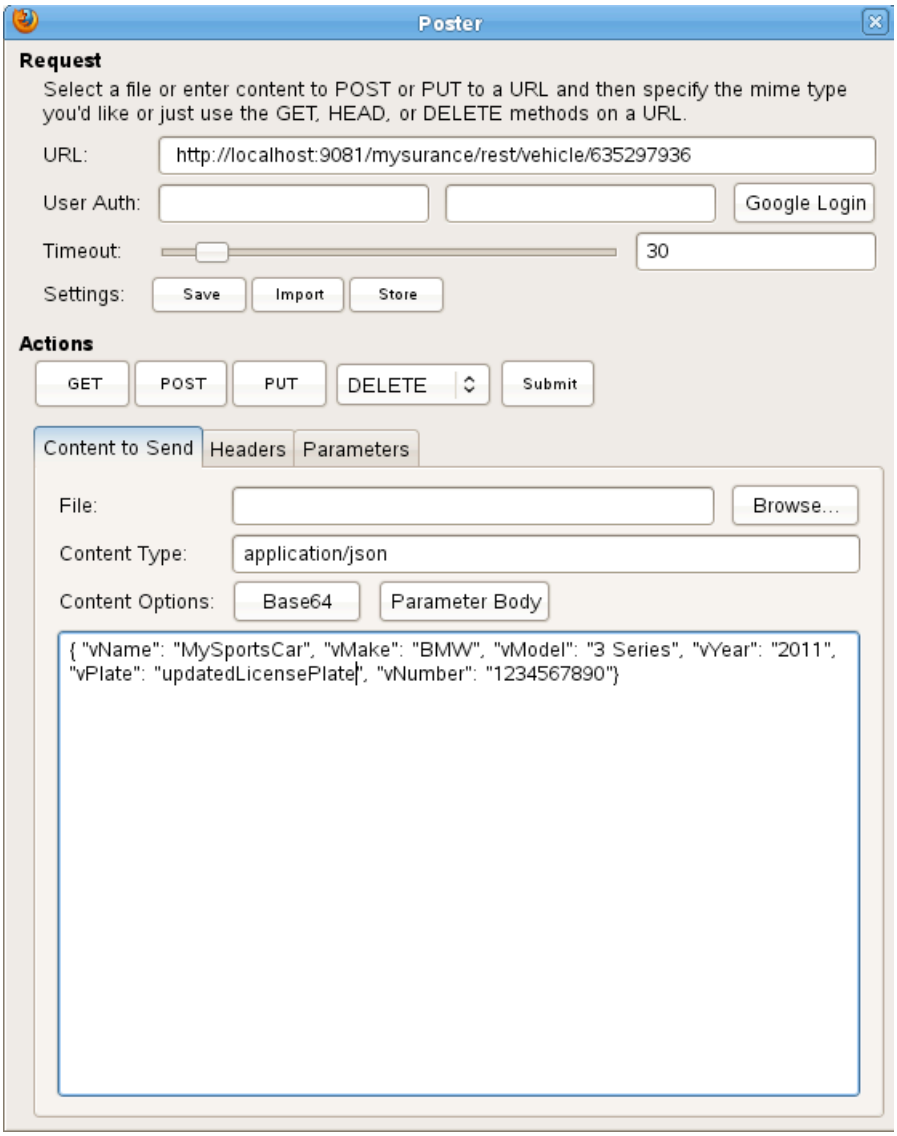

Illustration 46: Vehicle UPDATE request using Poster

So now we have changed the license place to “updatedLicensePlate” and sent a POST request to the server. The response should be 200 as shown below with the updated license plate information included in the response.

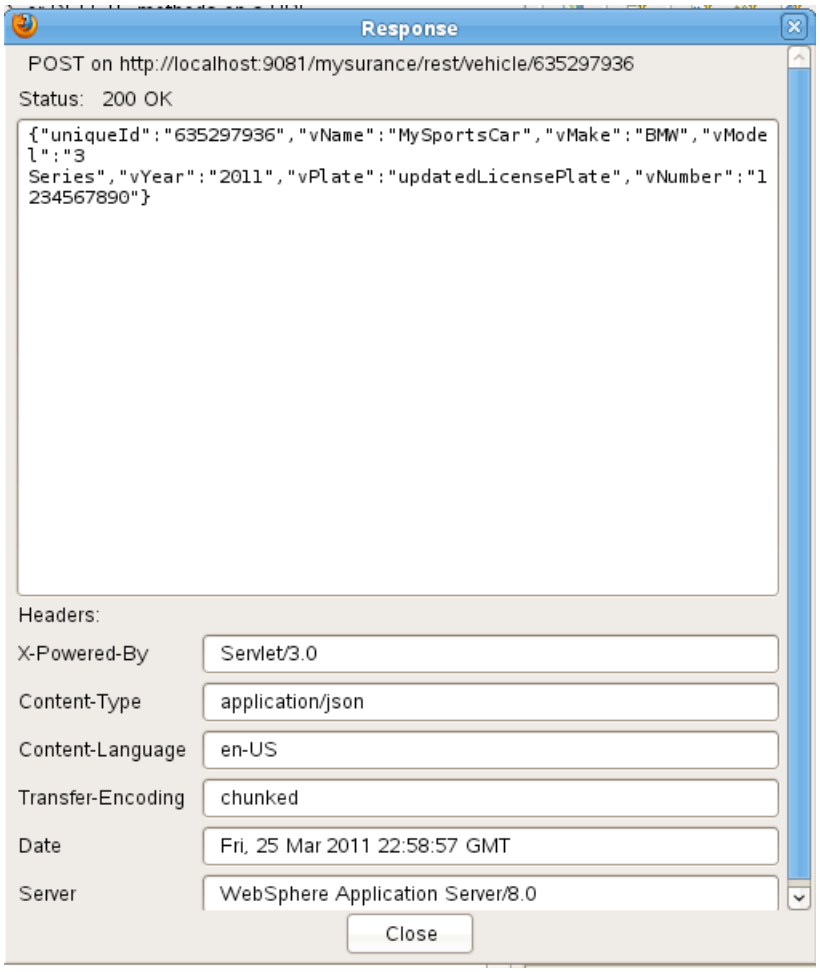

Illustration 47: Vehicle UPDATE response using Poster

1.10 Overview of the File Upload Service.

The file upload service is responsible for persisting files that have been uploaded to the server. In the demo, the client will upload photos taken with the mobile device camera. For the purpose of this demo, we will only be demonstrating the upload capabilities.

1.10.1 Registering the FileUpload Service with the Application registry.

In this step, we will again register with the application service this JAX-RS service. Add the following line just after the previously registered classes

```
classes.add(MultiFileUploadDataService.class);
```

At this point in time, there should be a set of errors indicating the various types of these annotations cannot be resolved. To remove these errors, simply right mouse click in the editor click on Source → 'Organize Imports'.

DemoApplicationRegistry.java

```
package com.ibm.impact.demo;

import java.util.HashSet;

public class DemoApplicationRegistry extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(org.codehaus.jackson.jaxrs.JacksonJaxbJsonProvider.class);
        classes.add(UserProfileDataService.class);
        classes.add(VehicleDataService.class);
        classes.add(MultiFileUploadDataService.class);
        return classes;
    }
}
```

Illustration 48: DemoApplicationRegistry after registering MultiFileUploadDataService

10.1.2 Uploading Files

In this portion of the demo, we will review the code that handles the file upload versus copy and paste as was done previously. In the class com.ibm.impact.demo.services.MultiFileUploadDataService we provide a REST Service for updating files via the @POST annotation. This service is slightly different than the previously written services since the POST payload is in the format of multipart/form-data. Leveraging support provided by Apache Wink's JAX-RS we are able to consume the POST body relatively easily without having to have too much knowledge of the way multi-part form data is constructed.

For the response, we are using the same type of data format (JSON) that we use in other demos. This format is consist with other multipart form clients that may access this resource such as Dojo's file uploader. We are again leveraging JAXB for the data format we use inside of the JAX-RS Service (MultiPartForm JAXB Pojo).

```
@POST
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)
public Collection<ImpactBeanType> upload(@Context ServletConfig servletConfig, @Context
HttpHeaders httpHeaders, @Context UriInfo uriInfo, BufferedInMultiPart bimp) throws IOException
{
```

10.1.3 Review what we learned and attempt to upload an image to the server.

Now that we have configured this JAX-RS Service, we should now be able to test whether the file-upload works.

10.1.4 Verifying the File-upload REST API

For testing of the file-upload service we are going to use a simple HTML form. The reason for this is that the format for sending file-upload is not JSON as was done in the previous JAX-RS services.

10.1.5 Simple form for testing file upload

The simple test form is located @ http://localhost:9081/mysurance/simpleTest.html
In this test form, we have 3 file upload fields to simulate multiple files being uploaded from the client.

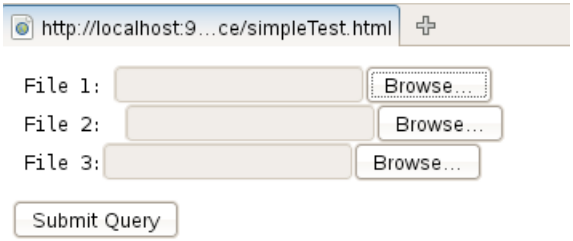

Illustration 49: Test file upload form

10.1.6 Verification of the file upload service

For the file upload test,m we are going to browser through our local file system and select a few files to upload. Once we have selected the files, we will submit the form. The result should be a JSON structure containing the HTTP URL indicating the location of each of the files that were uploaded.

Since the browser cannot by default display JSON, the browser will ask you whether you want to save the content. For the purpose of verifying the result select yes. You will then be given a chance to use an editor such as gedit to view the file. Once the file is readable in the text editor, you should see something such as

```
[{"name":"license.jpg","uri":"http://localhost:9081/mysurance/rest/upload/license.jpg"},
{"name":"tail_light.jpg","uri":"http://localhost:9081/mysurance/rest/upload/tail_light.jpg"},
{"name":"vpd.properties","uri":"http://localhost:9081/mysurance/rest/upload/vpd.properties"}]
```

At this point in time, we have completed the testing of our JAX-RS services and should be ready to embark on the PhoneGap portion of the demonstration.

Notes for Part I - Mysurance PhoneGap JAX-RS WebSphere Service

[1] The JAR files in step 2 are for the IBM version of Apache Wink that is supported by WebSphere Application Server 6.1 and later. Although it has not been tested as the time of this document being written, the version of Apache Wink on apache.org should also work but has not been verified. To use the Apache Wink version of JAX-RS, simply replace the servlet class defined in your web.xml with 'org.apache.wink.server.internal.servlet.RestServlet'.

[2] Jackson is available from http://jackson.codehaus.org/ if not using the WebSphere version of Apache Wink.

Part II - Mysurance PhoneGap Client for Android

2.0 Introduction

PhoneGap is a mobile development framework for developing cross-platform mobile applications using HTML, JavaScript and CSS. Native capabilities are available to the mobile application by calling the PhoneGap JavaScript APIs.

This section of the lab will step you through the development of a mobile PhoneGap application that calls the JAX-RS services written previously and running on the WebSphere server.

The application that we are writing entails a significant amount of HTML and JavaScript code. We will start off with a simple application and build upon it, until we have the complete Mysurance application. To save time and paper, not every line of code is listed in this document, but it is available for you to copy and paste from:

```
http://brycecurtis.github.com/phonegap-mysurance/
```

You can load this URL into your browser by clicking on the icon on the desktop.

There is also a pdf version of this document available in color at the same URL.

2.1 Open Eclipse IDE for Android Development

Eclipse provides an excellent development environment for developing Android applications. There is an Android plugin available that makes writing, compiling and testing applications easy to do.

Eclipse has been pre-installed on your lab computer and the Android development environment has been configured. To start, click on the **Android** icon on the desktop. This instance of Eclipse is preloaded with a simple Android application that will be used as a starting point for the lab.

2.2 Explore Android Project

Once Eclipse has initialized, the Mysurance sample application will appear in the **Package Explorer**.

Expand the **PhoneGap-Mysurance** tab and the **src**, **assets** and **libs** sub-tabs as shown to reveal several important files used by a typical Android application.

Java Code

Android applications are written in Java and are stored in the **src** directory. There is a main class that is launched when the icon is selected on the mobile device. For our example, this is the **com.ibm.mysurance.Mysurance** class. We will take a look at this class later, and will see that it is a very simple class that essentially loads our HTML file.

Web Content

PhoneGap applications are written using HTML/JS/CSS. These files are located under the **assets/www** directory. To keep code organized, there are two directories: **apps** and **libs**.

The **assets/www/apps** directory is where our Mysurance web code will be saved. We will start with a very simple application in **step1.html**. There is also a **storageDojo.js** file that has some JavaScript code used by our application to save data to an HTML5 database. (Mobile phones now have an SQL database on them.)

The **assets/www/libs** directory has **phonegap.0.9.4.js**, which is the PhoneGap JavaScript API library, and numerous other files which comprise the Mobile Dojo library (dojox.mobile). Mobile Dojo includes themes for both Android and iPhone that resemble native application UIs.

Native Libraries

The libs directory includes all jar files used by the Android application. The interesting library is **phonegap.0.9.4.jar**, which contains the native implementation of the various device capabilities. The **phonegap.0.9.4.js** code communicates directly with this jar file to enable calling of native capabilities from JavaScript.

Configuration

Android applications have an **AndroidManifest.xml** configuration file that describes the Java class to launch, device properties and permissions used by the application.

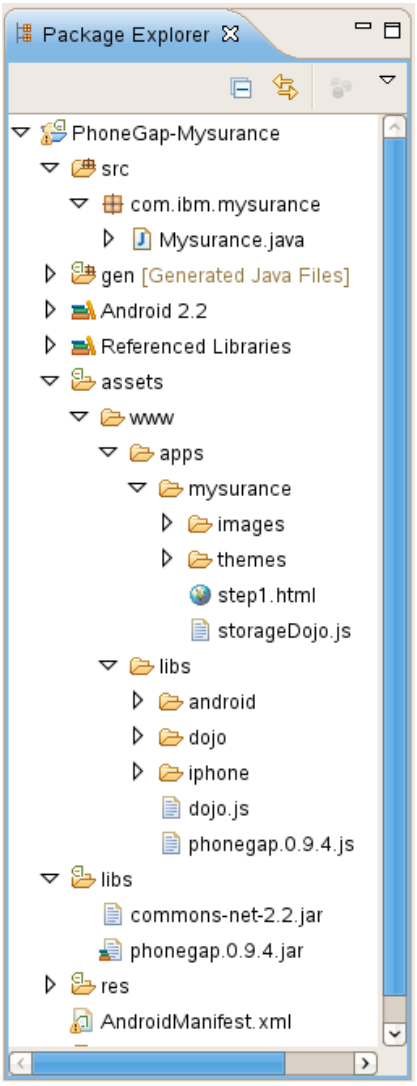

Illustration 50: Package Explorer for Android Project.

2.3 What is Mysurance.java?

The Android application is defined by **com.ibm.mysurance.Mysurance**. If it is not already shown, you can view this file by double-clicking on it in the **Package Explorer**.

```
package com.ibm.mysurance;

import com.phonegap.DroidGap;

import android.os.Bundle;

public class Mysurance extends DroidGap {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        super.loadUrl("file:///android_asset/www/apps/mysurance/step1.html");
    }
}
```

The primary function of this class is to load the HTML file containing our PhoneGap web content. The `file:///android_asset` path references files within the assets directory of our project.

Notice that **Mysurance** extends **DroidGap**, which is included in **phonegap.0.9.4.jar**. **DroidGap** is responsible for creating a browser container (WebView object) and loading the specified HTML file into it.

(You can close step2.html, step3.html and step9.html if they are open. The files don't exist, so don't worry about any errors.)

2.4 Our First Application

Our PhoneGap application is located in **step1.html**, which is the filename specified in the `loadUrl()` call above. To view its contents, right-click on it's name in the **Package Explorer** and select **Open With → Text Editor**.

```
<!DOCTYPE html >
<html lang="en">
<head>

<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
<meta name="viewport" content="width=device-width; initial-scale=1.0; maximum-scale=1.0; user-
scalable=no" />
<meta name="format-detection" content="telephone=yes" />

<title>Mysurance Demo</title>

<script type="text/javascript" charset="utf-8" src="../../libs/phonegap.0.9.4.js"></script>
<script type="text/javascript">

/**
 * Called when initial web page is loaded
 */
function onBodyLoad()
{
    console.log("onBodyLoad()");

    // Listen for PhoneGap framework to finish init
    document.addEventListener("deviceready",onDeviceReady,false);
}

/**
 * When this function is called, PhoneGap has been initialized and is ready to roll
 */
function onDeviceReady() {
    console.log("onDeviceReady()");
}
</script>
</head>
<body onload="onBodyLoad();">
This is our Mysurance demo app.
</body>
</html>
```

The <meta> tags are used to set mobile phone properties and how the HTML will be rendered within the browser container.

To use PhoneGap, the JavaScript library **phonegap.0.9.4.js** must be loaded. This connects our HTML

application to the underlying native PhoneGap code.

Our application should listen for the **onload** event and register for the **deviceready** event when the HTML page has been fully loaded. Once PhoneGap has been initialized, the **deviceready** event will be fired indicating that it is now safe to call PhoneGap JavaScript APIs.

If at any time during this lab you want to restore your code to the content above, you can go to the **step1.html** link on the website given in section 2.0 to copy and paste the contents.

2.5 Setting Up the Android Emulator

Before running our application, an Android emulator needs to be created. We have already created one for you, but it needs to be updated.

Select menu item **Window → Android SDK and AVD Manager**

This will display the **Android SDK and AVD Manager** that manages the emulators.

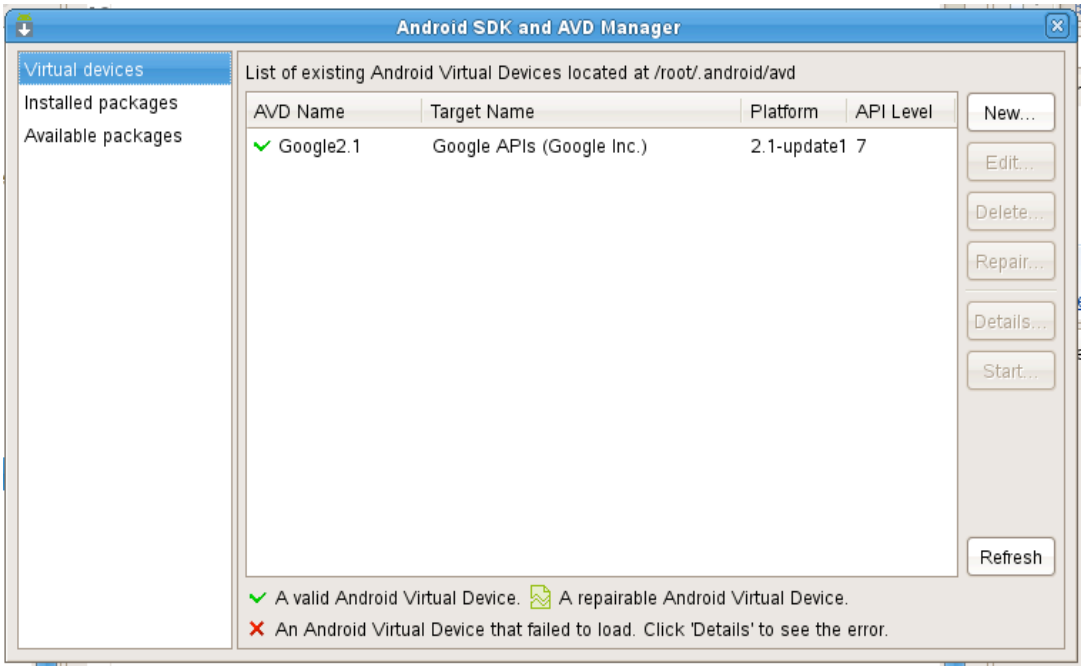

Illustration 51: Android SDK and AVD Manager.

Select the **Google2.1** AVD, and click **Edit...**

Change the **Skin: Built-in:** to *HVGA*

Click **Edit AVD** to save the emulator settings.

Close the **Android SDK and AVD Manager** by clicking on the X in the upper right corner.

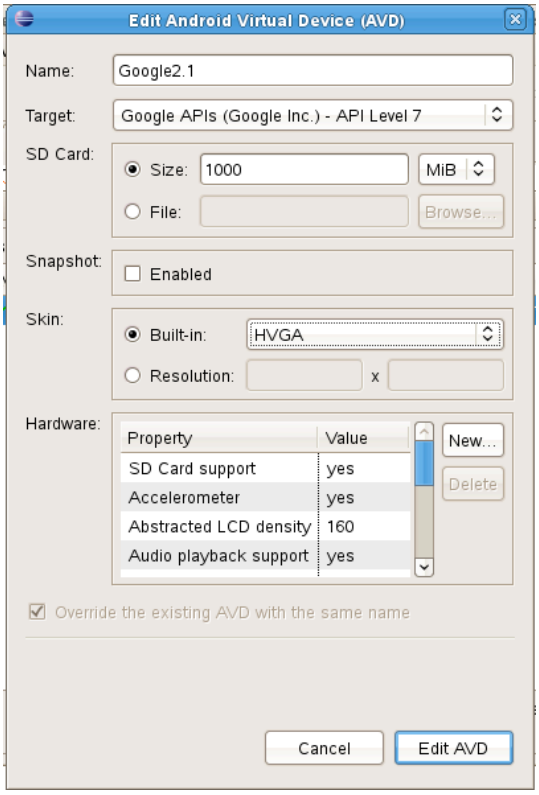

Illustration 52: Editing Android AVD for Version 2.1.

We now have an Android emulator created for version 2.1 with a screen resolution of HVGA or 320 x 480 pixels.

2.6 Add LogCat View to Eclipse

To see *console.log()* messages from the emulator, the **LogCat** view needs to be added to Eclipse.

Click menu item **Window → Show View → Other...**

Select **LogCat** from the list and press **OK**

You should now have a **LogCat** view.

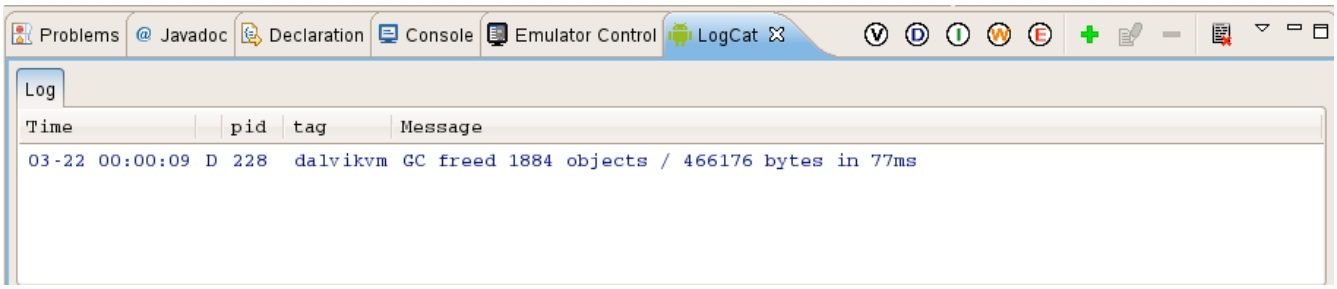

Illustration 53: Android LogCat view added to Eclipse.

2.7 Running Our Application

To run our application, we can either click the  run icon in the toolbar, or select the menu item **Run → Run**. This will automatically launch the Android emulator and load our application into it.

Note that it takes over a minute to start the emulator, so typical development workflow is to start the emulator once and keep it running throughout the development session.

Once our application has loaded, we will see the rendered HTML content.

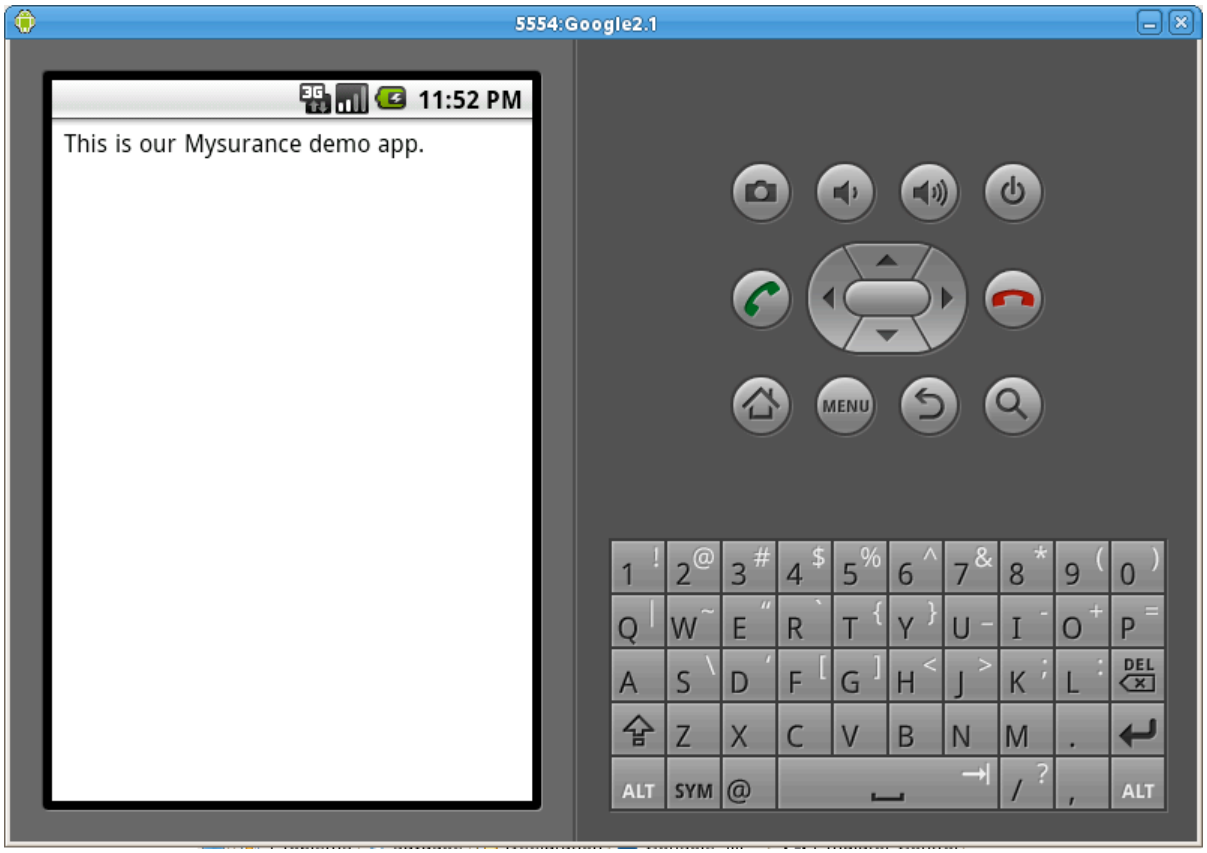

Illustration 54: Our First PhoneGap Application.

If you look in the **LogCat** view, you will see that our **onload** and **deviceready** handlers were called, indicating that PhoneGap has been initialized.

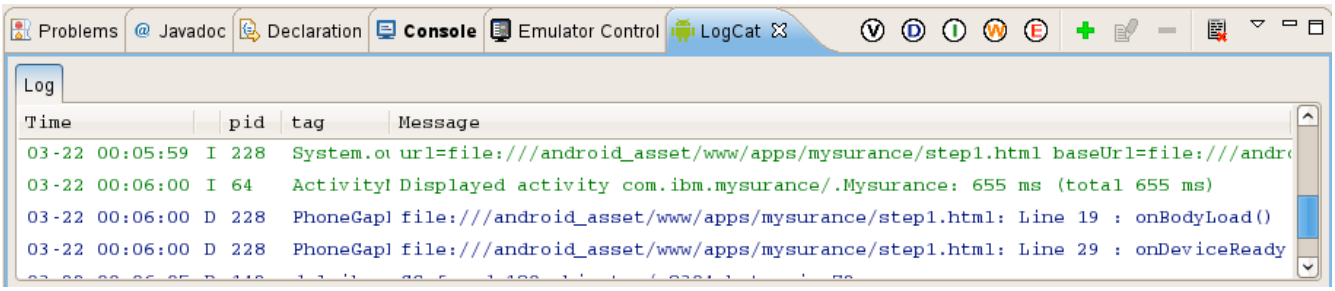

Illustration 55: PhoneGap Initialized and Ready.

2.8 Mysurance Application Main Screen

Now that we know our simple Android application runs, the next step is to start writing the Mysurance application. In this section we will build the starting screen.

The code marked with a gray background should be added to the existing code in **step1.html** that you opened into the Eclipse editor as described in Section 2.4. You will continue editing this same file throughout the lab.

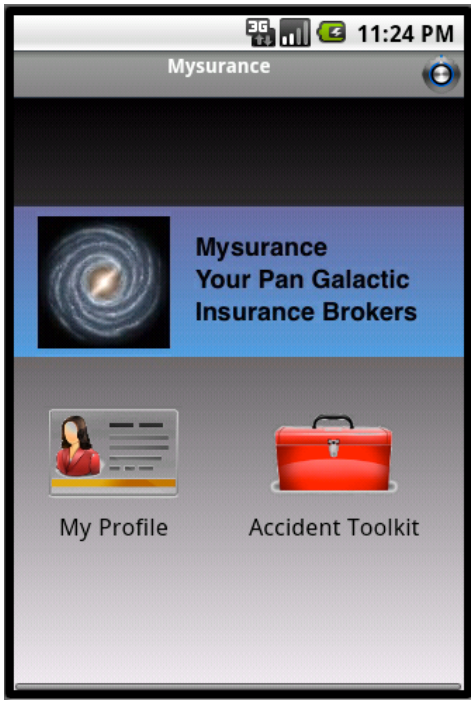

Illustration 56: Mysurance start screen.

To make it easier, the CSS files and images are already copied into the Mysurance project.

First, we need to add the **mysurance.css** and Dojo CSS files in the <head> tag.

The CSS files for the Mysurance plus the Dojo CSS themes for native looking widgets will be added. Locate the <title> tag and add the yellow highlighted code after it.

```
<title>Mysurance Demo</title>

<link rel="stylesheet" href="themes/android/mysurance.css" type="text/css" media="screen"
title="no title" charset="utf-8">
<link rel="stylesheet" href="../../libs/dojo/dojox/mobile/themes/android/android.css"
type="text/css" media="screen" title="no title" charset="utf-8">
```

Next, we need to load the PhoneGap and Dojo JavaScript libraries. We also load **storageDojo.js**, which has several methods that will help us persist our data to the phone's local database.

```
<script type="text/javascript" charset="utf-8" src="../../libs/phonegap.0.9.4.js"></script>
<script type="text/javascript" src="../../libs/dojo.js" djConfig="parseOnLoad:true"></script>
```

```
<script type="text/javascript" src="storageDojo.js"></script>

<script type="text/javascript">
dojo.require("dojox.mobile.parser");
dojo.require("dojox.mobile");
```

We will be using a database for our application. The *getDatabase()* function is defined in **storageDojo.js**, and hides all of the database details from us. It has functions for creating the tables, and storing and retrieving data from them.

```
/**
 * When this function is called, PhoneGap has been initialized and is ready to roll
 */
function onDeviceReady() {
    console.log("onDeviceReady()");
    // Open database for our app
    getDatabase();
}
</script>
```

The next step is to replace the <body> of our HTML file with the following markup needed to display the start screen.

```
<body onload="onBodyLoad();">

<!-- HOME PAGE -->
<div id="home" dojoType="dojox.mobile.View" selected="true">

    <h1 style="height: 32px; vertical-align: middle" dojoType="dojox.mobile.Heading">
        Mysurance
       <img style="float: right" onclick="serverClicked();" src="images/settings.png" />
    </h1>

    <img src="images/splash.jpg" style="position:absolute:top:0px;left:0px;" />
    <table width="100%" style="cell-padding:5px;position:absolute;top:250px;">
        <tr>
            <td valign="middle" align="center" onclick="myInfoClicked();">
               <img src="images/profile.png" width="100px"/></td>
            <td valign="middle" align="center" onclick="accHelpClicked();">
               <img src="images/toolbox.png" width="100px"/><td>
        </tr>
        <tr>
             <td align="center"><span style="color:black;">My Profile</span></td>
             <td align="center"><span style="color:black;">Accident Toolkit</span></td>
       </tr>
    </table>
</div>
<script type="text/javascript">

/**
 * Display profile panel
 */
function myInfoClicked() {
    loadProfile(true);
    dijit.byId("home").performTransition("myInfo", 1, 'slide');
}

/**
 * Display accident toolkit panel
 */
function accHelpClicked() {
```

View is the Dojo Container

Heading specifies the title bar

Backsplash image

Menu images

Menu labels

```
    dijit.byId("home").performTransition("accHelp", 1, 'slide');
}

/**
 * Display settings panel
 */
function serverClicked() {
    dijit.byId("home").performTransition("appSettings", 1, 'slide');

    // Set the input with the server address from local storage
    document.getElementById("serverAddr").value = serverAddress;
}
</script>
</body>
</html>
```

The HTML markup shown above builds the user interface by laying out the images and defining the functions to be called when the images are selected.

When the profile image is clicked, the *myInfoClicked()* function is called. This function loads the user's profile from the database and fills in the input elements in the **User Profile** screen that will be created in Section 2.10.

Similarly, when the toolbox image is clicked, the *accHelpClicked()* function is called. It displays the **Accident Toolkit** screen to be created in Section 2.11.

There is another image in the title bar that is used to configure the address of the WebSphere server that will be used later. When the settings image is clicked, the *serverClicked()* function is called. It displays the **Settings** screen, which will be created in the next section.

This completes the first part of our Mysurance application. Compare your HTML file to **step2.html** on the website to see if you need to make any corrections. You can also copy and paste the entire contents directly into your HTML file. You will continue editing the same **step1.html** file you opened in Section 2.4 throughout this lab. If you really, really want to create a new file for each step, you will have to change **Mysurance.java** to specify the correct HTML file to be loaded.

Run your application as before, by selecting menu item **Run → Run** or clicking on the **run** icon.

2.9 Adding Our Second Screen

This section will show how to add a second screen to our application.

To create another screen for our application, we create a new with dojoType="*dojox.mobile.View*".

The **Settings** screen will display the IP address and context root of the WebSphere server. When the small, settings icon on the right side in the title bar is clicked, the following screen will be shown.

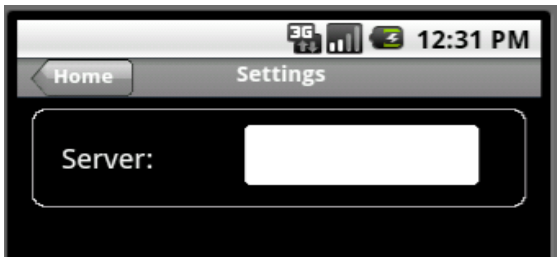

Illustration 57: The Settings Screen.

This HTML markup should be added at the bottom of the HTML file, just before the </body> tag.

```
<!-- APPLICATION SETTINGS PAGE -->
<div dojoType="dojox.mobile.View" id="appSettings">
    <h1 dojoType="dojox.mobile.Heading" back="Home" moveTo="home">Settings</h1>
    <ul dojoType="dojox.mobile.RoundRectList">
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="serverAddr" >Server:</label>
            <input class="itemInput" id="serverAddr" type="text"
                onChange="serverAddressChanged(this);"/>
        </li>
    </ul>
</div>
<script type="text/javascript">

// Global variable
var serverAddress = localStorage.getItem("serverAddress");

/**
 * The server address has been changed
 */
function serverAddressChanged(el) {
    serverAddress = el.value;
    localStorage.setItem("serverAddress", serverAddress);
}
</script>
</body>
</html>
```

The for the **Settings** screen has an **id="appSettings"**. This is the same id passed in to *dijit.performTransition()* in our *serverClicked()* function defined in Section 2.8. When the settings icon is clicked, Dojo will slide this new screen into view. Dojo also adds the **Home** button, so we can return to the previous screen.

The server IP address will be saved in the a global variable *serverAddress*. This variable is set when the value of the input element changes. We are also used HTML5 Local Storage to save the server address, so we don't have to reenter it every time we restart our application. HTML5 Local Storage saves data as name/value pairs and can be easily accessed using *localStorage.getItem(key)* and *localStorage.setItem(key, value)*.

After adding the HTML markup above (or from **step3.html** on the website), launch your application again. If you kept the Android emulator open, the modified application will be uploaded and restarted.

Click on the settings icon in the title bar to see this new screen.

Before entering the IP address of the server, it needs to be retrieved. To get the address,

- Open up a command prompt
- Type in **ifconfig**
- Look for the inet address for eth#.

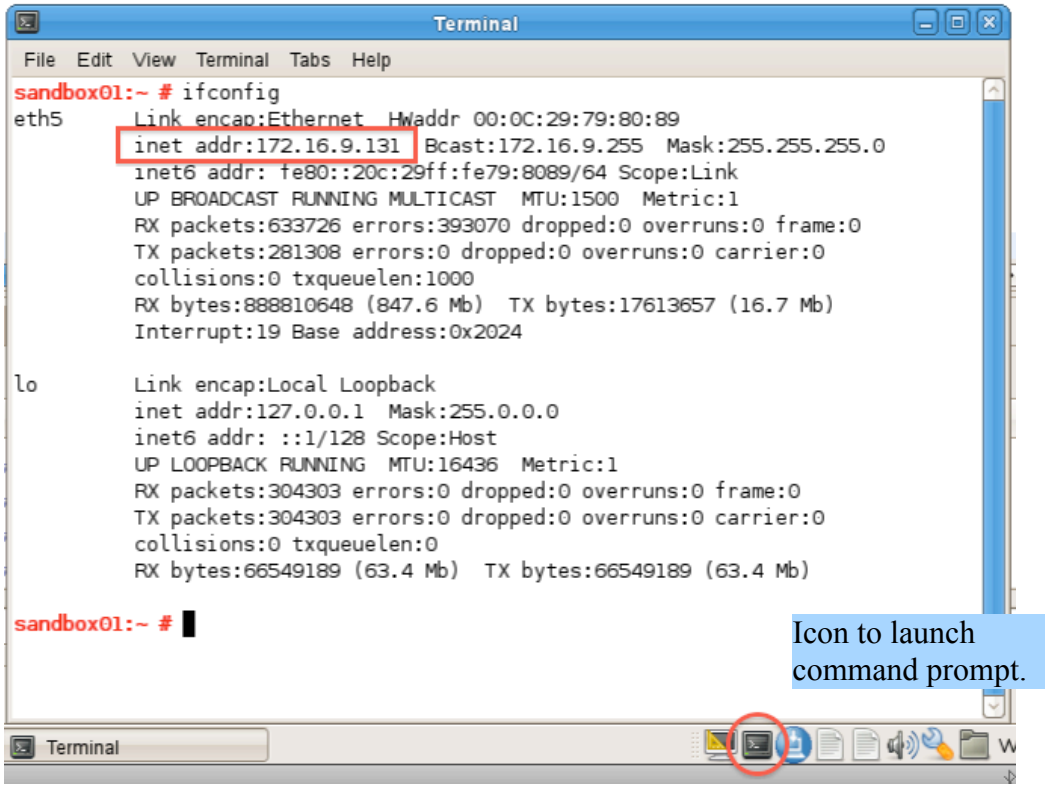

Illustration 58: How to find the server IP address.

Switch back to the Android emulator. For the server shown above,

- Enter *172.16.9.131:9081/mysurance* into the input field.
- Press the **Home** button to return back to the starting screen.

For our application, you should click the back button in the title bar of the screen. This will return you to the previous screen. There is a **Back** button on Android phones that is typically used to return to the previous screen, however, our screens are not true Android screens. The Android **Back** button can be overridden to behave the way we want, but it was not done for our example to keep the code simpler.

2.10 The User Profile Screen

The next screen to be created is the **My Info** screen. This screen is displayed when a user clicks on the profile image on the **Start** screen. To return back to the **Start** screen, you should press the **Home** button in the title bar.

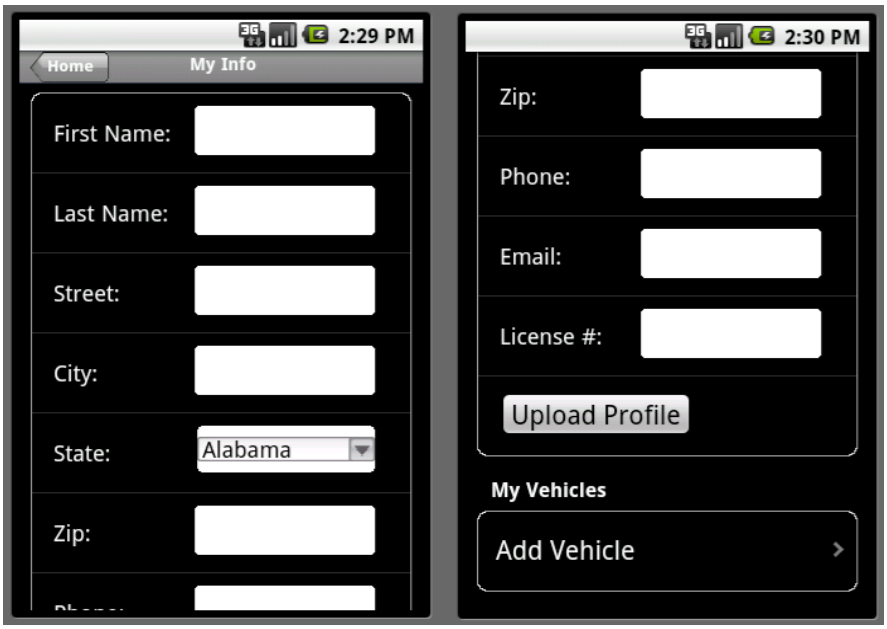

Illustration 59: The My Info screen.

As before, we create a new that defines the screen. The following HTML markup should be added at the end of the HTML file, just before the </body> tag. (See **step4.html** link on the website.)

Heading creates the "My Info" title bar with the Home button.

RoundRectList creates the rounded borders.

ListItem creates a row with label and input.

```
<!-- EDIT PROFILE PAGE -->
<div id="myInfo" dojoType="dojox.mobile.View">
    <h1 dojoType="dojox.mobile.Heading" back="Home" moveTo="home"
        onClick="storeProfile(true);">My Info</h1>
    <ul dojoType="dojox.mobile.RoundRectList">
        <li dojoType="dojox.mobile.ListItem">
           <label class="itemText" for="fName" >First Name:</label>
           <input class="itemInput" id="fName" type="text" onChange="userProfileChanged();"/>
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="lName" >Last Name:</label>
            <input class="itemInput" id="lName" type="text" onChange="userProfileChanged();">
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="street" >Street:</label>
            <input class="itemInput" id="street" type="text" onChange="userProfileChanged();">
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="city" >City:</label>
            <input class="itemInput" id="city" type="text" onChange="userProfileChanged();" />
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="state" >State:</label>
            <select class="itemInput" id="state" type="text" onChange="userProfileChanged();"/>
                <option value="AL">Alabama</option>
                   ...
                <option value="WY">Wyoming</option>
            </select>
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="zip" >Zip:</label>
            <input class="itemInput" id="zip" type="number" size="5"
               onChange="userProfileChanged();"/>
```

```
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="phone" >Phone:</label>
            <input class="itemInput" id="phone" type="tel" onChange="userProfileChanged();"/>
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="email" >Email:</label>
            <input class="itemInput" id="email" type="email" onChange="userProfileChanged();"/>
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <label class="itemText" for="licenseNo" >License #:</label>
            <input class="itemInput" id="licenseNo" type="number"
               onChange="userProfileChanged();"/>
        </li>
        <li dojoType="dojox.mobile.ListItem">
            <button id="profileBtn" onclick="uploadProfile();">Upload Profile</button>
        </li>
    </ul>
    <input type="hidden" id="profileId" />
    <input type="hidden" id="vCount" />
    <h2 dojoType="dojox.mobile.RoundRectCategory">My Vehicles</h2>
    <ul dojoType="dojox.mobile.RoundRectList" id="vList">
        <li dojoType="dojox.mobile.ListItem" id="addItem" moveTo="addVehicle"
            transition="slide" onclick="addVehicleClicked();">
            Add Vehicle
        </li>
    </ul>
</div>
```

Dojo allows us to annotate our HTML and specify screen formatting. First of all, each screen must be identified by it's own with `dojoType="dojox.mobile.View"`. The title bar is indicated by <h1> with `dojoType="dojox.mobile.Heading"`. The `back` attribute specifies the text for the back button, and the `moveTo` attribute specifies which to load when the back button is clicked.

To complement the HTML above, we need several JavaScript functions to handle change and click events for the input elements.

We will use *profileChanged* variable to keep track if any of the input elements have changed, so we don't needlessly save data. The *userProfileId* variable identifies the particular user in the database.

The function *storeProfile()* saves the profile data in the database if *profileChanged* variable is set. This helper function is located in the **storageDojo.js** file.

```
<script type="text/javascript">

//Global Variable
var profileChanged = false;
var userProfileId = null;

/**
 * Set flag indicating that profile has been modified
 */
function userProfileChanged() {
    console.log("userProfileChanged()");
    profileChanged = true;
}

/**
 * The Add Vehicle menu item was clicked
```

```
 */
function addVehicleClicked() {
    console.log("addVehicleClicked()");

    // Save changes to profile
    storeProfile(true);

    // Set the ownerId of the form to the profileId
    document.getElementById("ownerId").value = document.getElementById("profileId").value;
}

/**
 * The View Vehicle menu item was clicked
 *
 * @param el        The el of the vehicle item
 */
function viewVehicleClicked(el) {
    console.log("viewVehicleClicked("+el+")");
    var id = el.getAttribute("vid");
    console.log(" - id="+id);

    // Save changes to profile
    storeProfile(true);

    // Set the ownerId of the form to the profileId
    document.getElementById("ownerId").value = document.getElementById("profileId").value;
    document.getElementById("vehicleId").value = id;

    // Load vehicle profile
    loadVehicle(id);
}
</script>

</body>
</html>
```

The **Upload Profile** button will be used to send the profile data from this screen up to the server to be stored. It will be calling the JAX-RS **UserProfile** service created previously.

The service can be accessed at URI *http://server:9081/mysurance/rest/userProfile* using ajax. Since this is a REST service, a PUT request is used to create the user profile, POST is used to update it, GET is used to retrieve it, and DELETE is used to delete it.

For our application, we will only implement creating or updating the user's profile when the user clicks on the **Upload Profile** button.

Normally, synchronizing data with a server is handled automatically by the application. When a change is made locally the request is queued. If there is a data connection to the server, then it is uploaded. If no connection is available, then the change is held until a connection becomes available. This is referred to as one-way data synchronization.

Two-way synchronization is used if the data can be changed by another client. In that case the application should download and update the user profile data stored on the device.

The **UserProfile** service running on the server is expecting all messages to be sent as a JSON string. The *stringifyProfile()* function fills this role by retrieving the values from each of the input elements on the screen and assembling a JSON string from them. This string is then sent to the server using the

following code, which should be added before the ending </script></body> tag. (You can copy and paste it from **step4.html** on the website.)

```
/**
 * Return profile as JSON string
 */
function stringifyProfile() {
    var profile = '{';
    if (userProfileId != null) {
          profile = profile + '"uniqueId": ' + '"' + userProfileId + '", ';
    }
    profile = profile + '"fName": ' + '"' + document.getElementById("fName").value + '", ';
    profile = profile + '"lName": ' + '"' + document.getElementById("lName").value + '", ';
    profile = profile + '"street": ' + '"' + document.getElementById("street").value + '", ';
    profile = profile + '"city": ' + '"' + document.getElementById("city").value + '", ';
    profile = profile + '"state": ' + '"' + document.getElementById("state").value + '", ';
    profile = profile + '"zip": ' + '"' + document.getElementById("zip").value + '", ';
    profile = profile + '"phone": ' + '"' + document.getElementById("phone").value + '", ';
    profile = profile + '"email": ' + '"' + document.getElementById("email").value + '", ';
    profile = profile + '"licenseNo": ' + '"' + document.getElementById("licenseNo").value +
'"}';
    return profile;
}

/**
 * Upload profile to server
 */
function uploadProfile() {
    console.log("uploadProfile()");
    if (!serverAddress) {
        return;
    }
    // Make an Ajax request to profile service
    var request = new XMLHttpRequest();
    if (userProfileId == null) {
        request.open("PUT", "http://" + serverAddress + "/rest/userProfile", true);
    } else {
        request.open("POST", "http://" + serverAddress + "/rest/userProfile/" + userProfileId,
true);
    }
    request.setRequestHeader("Content-type", "application/json");

    //Call a function when the state changes
    request.onreadystatechange = function() {
        console.log("uploadProfile: state = " + request.readyState);
        if(request.readyState == 4) {
            console.log("uploadProfile: status = "+request.status);
            console.log("uploadProfile: response = "+request.responseText);
            if ((userProfileId == null) && request.responseText) {
                var location = request.getResponseHeader("Location");
                userProfileId = location.substring(location.lastIndexOf('/') + 1,
location.length);
                console.log(userProfileId);
            }
        }
    }
    // Send JSON object to server
    sendValue = stringifyProfile();
    console.log("Profile: " + sendValue);
    console.log("Sending user profile");
    request.send(sendValue);
}
```

```
</script>
</body>
</html>
```

If the *userProfileId* is not set, then this is a new profile that is sent to the server using PUT, otherwise it is POSTed to the server using ajax.

Let's test out the application and send the user profile to the server.

- Run the application and fill in some profile data.
- Press the **Upload Profile** button.

You should see log messages on the server console to indicate that the request was received.

```
com.ibm.impact.demo.services.UserProfileDataService createUserProfile creating
userProfile UserProfile [uniqueId=null, fName=Bryce, lName=Curtis, street=123 Main,
city=Austin, state=TX, zip=78758, phone=, email=, licenseNo=]
```

Text 1: WebSphere Console after receiving PUT request to User Profile Service.

You should also see log messages in the LogCat console showing the device/server interaction.

```
uploadProfile()
Profile: {"fName": "Bryce", "lName": "Curtis", "street": "123 Main", "city": "Austin",
"state": "TX", "zip": "78758", "phone": "", "email": "", "licenseNo": ""}
Sending user profile
state = 2
state = 4
uploadProfile: status = 201
uploadProfile: response =
```

Text 2: Android LogCat Console after Uploading Profile to Server.

The status code of 201 returned by the server is the typical response to a PUT and indicates that the profile was created.

The **Add Vehicle** screen is very similar to the **Profile** screen, so we won't go into the details. However, the HTML markup included in **step5.html** on the website should be copy and pasted into your HTML file to implement this screen. (As always, you can copy and paste the entire contents of the HTML file from the website into your local HTML file.)

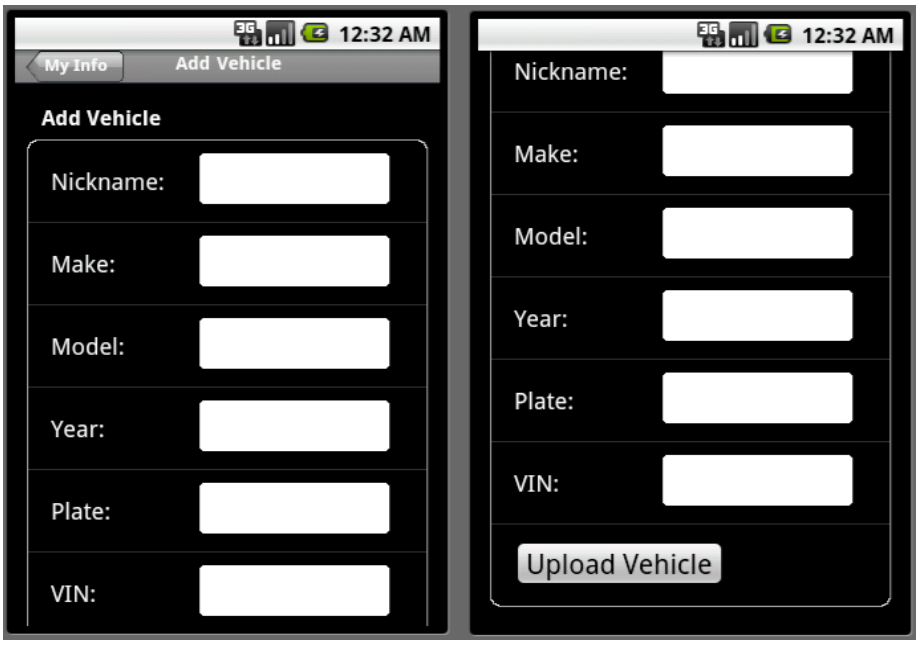

Illustration 60: Add Vehicle Screen.

The **Upload Vehicle** button calls the JAX-RS **Vehicle** service. You should see log messages in both the LogCat and server consoles indicating that the vehicle information was uploaded to the server.

2.11 The Accident Toolkit

In this section we will create the **Accident Toolkit** screen. This screen is a list of the several items that may be useful during an accident.

Illustration 61: The Accident Toolkit screen.

The first two items, **Call the Police** and **Call for a Tow Truck**, will display a map of the nearest Police station or Towing service, and their phone numbers based upon your current location. Most phones recognize a geolocation URI of the form:

```
geo:lat,lng?q=query
```

Loading this URI will display the phone's map application.

The last three items display additional screens in our application.

Below is the HTML markup that creates the **Accident Toolkit** screen. It should be added to the bottom of your HTML file, just before the </body> tag. You can also copy and paste it from **step6.html** on the website.

```
<!-- ACCIDENT TOOLKIT PAGE -->
<div dojoType="dojox.mobile.View" id="accHelp">
    <h1 dojoType="dojox.mobile.Heading" back="Home" moveTo="home">Accident</h1>
    <div class="text">If you are in an accident, you should first move to a safe location.
Below are some additional actions you can take:</div>
    <ul dojoType="dojox.mobile.RoundRectList">
        <li dojoType="dojox.mobile.ListItem" id="policeListItem"
onclick='mapQuery("police");'>Call the Police</li>
        <li dojoType="dojox.mobile.ListItem" onclick='mapQuery("towing");'>Call for a Tow
Truck</li>
        <li dojoType="dojox.mobile.ListItem" moveTo="accInfo" transition="slide"
onClick="loadAccidentInfo();">Exchange Driver Info</li>
        <li dojoType="dojox.mobile.ListItem" moveTo="accLocation" transition="slide"
onClick="loadLocation();">Record Accident Location</li>
        <li dojoType="dojox.mobile.ListItem" moveTo="accPhotos" transition="slide"
onClick="loadPhotos();">Take Photos of Accident</li>
    </ul>
</div>
<script type="text/javascript" charset="utf-8">

/**
 * Display locate query string
 */
function mapQuery(queryString) {
    console.log("mapQuery()");
    window.location = "geo:0,0?q="+queryString;
}
</script>
```

The Dojo markup should look familiar by now. It is used to create a rounded rectangle frame with items that can be selected.

The *mapQuery()* method loads the geo URI, displaying the phone's map application. The pair 0,0 used for lat,lng implies that our current location should be used. The query string is either *police* or *towing*.

If you run the application and select **Call for a Tow Truck**, you should see a map.

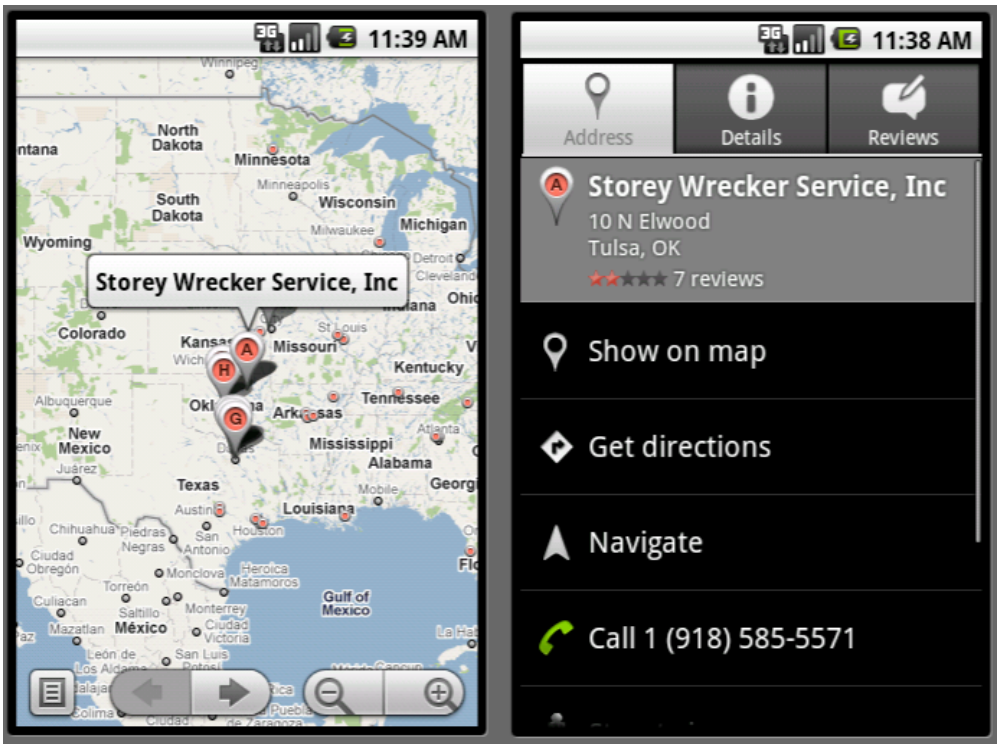

Illustration 62: Call for a Tow Truck displays the phone's mapping application.

Unfortunately, the geolocation function doesn't work with the Android emulator running in our lab environment. So, the current location defaults to the middle of the country.

If you click on one of the pins, another page will be displayed with information about the towing business. You can call the business or get directions to it.

To return back to your **Accident Toolkit** screen, press the  back button on the Android emulator. Each click of the back button will back up one screen in the map application. Once you get back to our **Accident Toolkit** screen, don't press it again or you will exit our application.

If you want to see the towing services available in the Las Vegas area, change the lat,lng pair from 0,0 to 36.18,-115.13 and rerun our application.

2.12 Exchanging Driver Info

The **Exchange Driver Info** screen is created in the same way that the **User Profile** screen was. This screen saves several pieces of information to the local database. The screen includes information about the other driver and their vehicle.

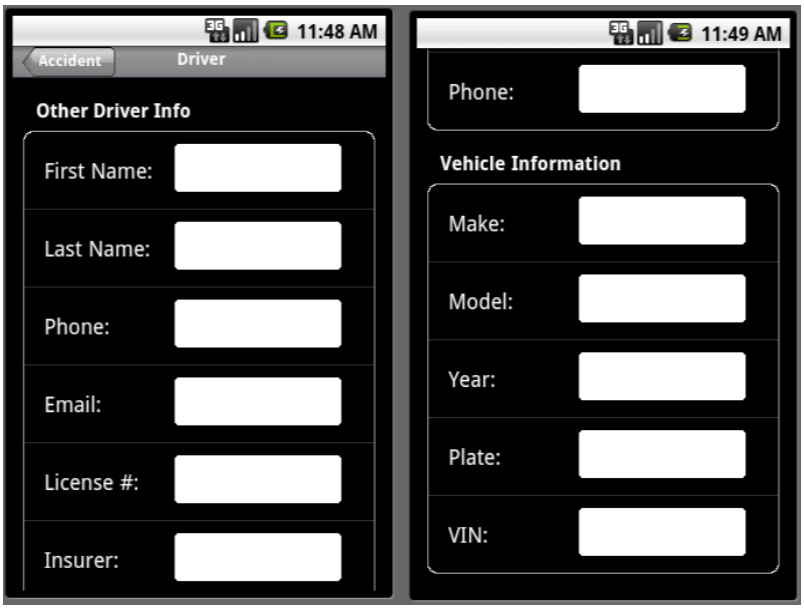

Illustration 63: Other Driver Info Screen.

The HTML markup for this screen is very similar to the other user input screens. Copy and paste this section from **step7.html** on the website to the end of your HTML file.

2.13 Accident Location

The **Accident Location** screen has inputs to record the location of the accident. There is also a **Use Current Location** button that can be clicked to retrieve the current address by mapping the latitude and longitude of the phone to an address using a reverse geocoding Google web service.

Illustration 64: Accident Location Screen.

Referring to the code below, the current location is retrieved by calling `navigator.geolocation.getCurrentPosition()`. This function is part of HTML5 and is built-in to most phones today. The function returns a location object to the specified callback function. The latitude and longitude from this location object are sent to the Google reverse geocoding service using an ajax request. The service returns a JSON object that contains the street address of the latlng parameter requested. We can populate our input elements with the result.

Unfortunately, the Android emulator used in the lab does not include support for geolocation, we will have to specify our own location.

```
/**
* Use current location button clicked
*/
function onButtonLocClicked(e) {
    console.log("onButtonLocClicked()");
    navigator.geolocation.getCurrentPosition(reverseGeocode,
            function() {
             console.log("Error retrieving current location.  Using hardcoded location for Las
Vegas.");
             reverseGeocode({coords: {latitude:36.175, longitude:-115.1363889}});
        });
}

/**
 * Find street address from lat,lng
 */
function reverseGeocode(loc) {
    console.log("Current location: ("+loc.coords.latitude+","+loc.coords.longitude+")");

    // Do reverse geocode
    var url = "http://maps.google.com/maps/api/geocode/json?
latlng="+loc.coords.latitude+","+loc.coords.longitude+"&sensor=true";
    var req = new XMLHttpRequest();
    console.log("Reverse geocode: requested url: " + url);
    req.open("GET", url, true);
    req.onreadystatechange = function(){
        if (req.readyState == 4) {
            console.log("status=" + req.status + " length=" + req.responseText.length);
            // only if "OK"
            if (req.status == 200 || req.status == 0) {
                try {
                    eval('var r='+req.responseText+';');
                    var streetNum = '';
                    var street = '';
                    var city = '';
                    var state = '';
                    var zip = '';
                    var a = r.results[0].address_components;
                    for (var i in a) {
                        var l = a[i];
                        if (l.types[0] == 'street_number') {
                            streetNum = l.long_name;
                        }
                        if (l.types[0] == 'route') {
                            street = l.long_name;
                        }
                        if (l.types[0] == 'locality') {
                            city = l.long_name;
                        }
                        if (l.types[0] == 'administrative_area_level_1') {
```

```
                            state = l.short_name;
                        }
                        if (l.types[0] == 'postal_code') {
                            zip = l.long_name;
                        }
                    }
                    console.log(" streetNum="+streetNum+" street="+street+" city="+city+"
state="+state+" zip="+zip);
                    dojo.byId("streetLoc").value = streetNum+" " + street;
                    dojo.byId("cityLoc").value = city;
                    dojo.byId("stateLoc").value = state;
                    dojo.byId("zipLoc").value = zip;
                    locationDataChanged();
                } catch (e) {
                    console.log("Error evaluating geocode json response: "+e);
                }
            }
            else {
                console.log("Error: Unexpected response: " + req.responseText);
            }
        }
    };
    req.send();
}
```

Copy the HTML markup from **step8.html** on the website into our HTML file and save it. Now, let's run our application with these changes.

- Navigate to the accident **Location** screen
- Press the **Use Current Location** button

You should see the inputs populated with an address in Las Vegas, Nevada.

2.14 Taking Pictures

The camera and file transfer capabilities provided by PhoneGap will be used in this section to take a picture and then upload it to the WebSphere server.

To take a picture, `navigator.camera.getPicture(successCallback, errorCallback, options)` is called. This function launches the phone's camera application. After a picture has been taken, the filename of the image is returned to the success callback. If there was an error, or the user cancelled the camera, the error callback is called. The quality of the captured image can be specified in the options.

Illustration 65: Accident Photos Screen.

```
// Global variable
var photoId = null;

/**
 * Take photo
 *
 * @param id        The photo id
```

```
 */
function takePhoto(id) {
    console.log("takePhoto("+id+")");
    photoId = id;
    navigator.camera.getPicture(photoSuccess, photoError, { quality: 20, destinationType:
Camera.DestinationType.FILE_URI });
};

/**
 * Callback for successful photo taken
 *
 * @param imageData
 */
function photoSuccess(imageData) {
    console.log("photoSuccess() for photo "+photoId);
    displayPhoto(photoId, imageData);
    try {
        storePhoto(photoId, imageData);
    }
    catch (e) {
        console.log("Error saving photo to database: " + e);
    }
    photoId = "";
};

/**
 * Display photo
 *
 * @param id
 * @param imageData
 */
function displayPhoto(id, imageData) {
    console.log("displayPhoto("+id+","+imageData.length+")");
    var el = document.getElementById("photo"+id+"-text");
    if (imageData) {
        el.style.visibility = 'hidden';
        el.style.display = 'none';
    }
    else {
        el.style.visibility = 'visible';
        el.style.display = 'block';
    }

    var img = document.getElementById("photo"+id);
    if (imageData) {
        img.style.height = "150px";
        img.style.visibility = 'visible';
        img.style.display = 'block';
        img.src = imageData;
    }
    else {
        img.style.height = "150px";
        img.style.visibility = 'hidden';
        img.style.display = 'none';
        img.src = "";
    }
};

/**
 * Callback for camera error
 *
 * @param error
 */
```

```
function photoError(error) {
    console.log("photoError() for photo "+photoId+": " + error);
};
```

For our application, once the picture is taken, it is displayed on the screen by setting the src attribute of an <img> tag and unhiding the image. The filename of the image is also saved in the database.

The image can be uploaded to the WebSphere server by calling the **File Upload** service. The service can be accessed at URI *http://server:9081/mysurance/rest/upload*.

```
/**
 * Upload photo
 *
 * @param id
 */
function uploadPhoto(id, imageURL) {
    console.log(imageURL);
    var options = new FileUploadOptions();
    options.fileKey="file";
    options.fileName="photo" + id + ".jpg";
    options.mimeType="image/jpeg";
    var ft = new FileTransfer();
    ft.upload(imageURL, "http://" + serverAddress + "/rest/upload", uploadSuccess, uploadError,
options);
}

/**
 * Callback for successful upload
 */
function uploadSuccess() {
    alert("You photo has been uploaded to the server");
}

/**
 * Callback for unsuccessful upload
 *
 * @param error
 */
function uploadError(error) {
    console.log("uploadError() for photo: " + error);
}
```

PhoneGap's **FileTransfer** service is used to upload the image to the WebSphere server. The upload function does a multi-form POST of the image data to a URI. There are callbacks for success and failure.

If you copy the HTML markup from **step9.html** on the website into your HTML file, you should be able to take a picture and upload it to the server. The emulator does not support a camera, but does provide an Android image instead.

When you click on one of the four picture areas, the camera application will be shown. Press the shutter icon to snap the picture, then press the Ok button to finish. Upon taking a picture, you should see the image added to the screen.

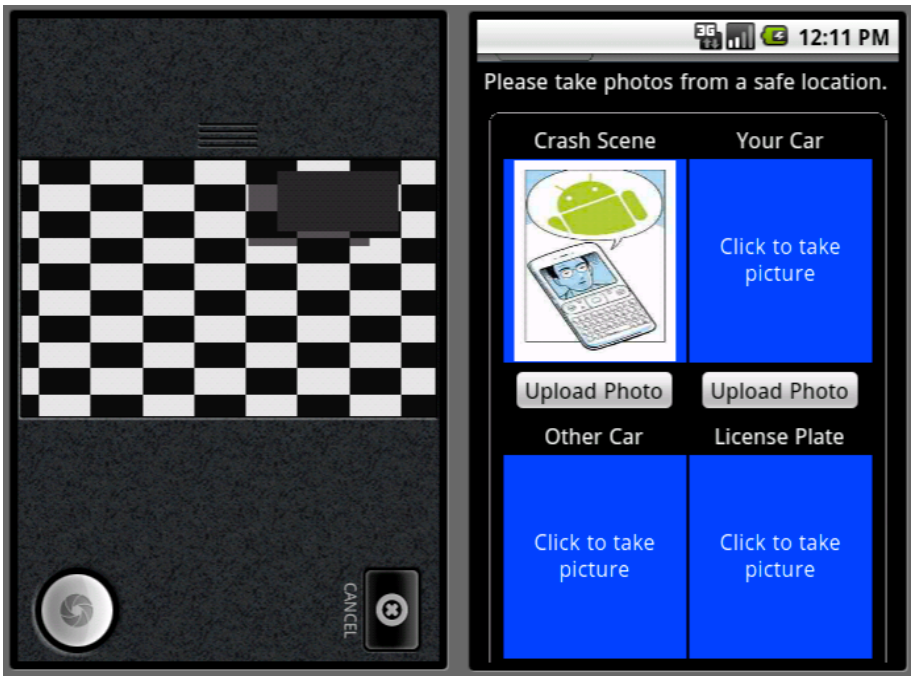

Illustration 66: The Camera Application and Photo Taken.

If you upload the image to the server using the **Upload Photo** button, the LogCat and WebSphere server consoles should indicate that the image was successfully uploaded.

```
getPhoto(0)
content://media/external/images/media/1
got response from server
[{"name":"photo0.jpg","uri":"http://172.16.9.131:9081/Impact2011Demo/rest/upload/photo0.jpg"}]
****** About to return a result from upload
```

Text 3: Android LogCat Console After Uploading an Image to WebSphere.

There should also be a dialog box displayed on the phone indicating success.

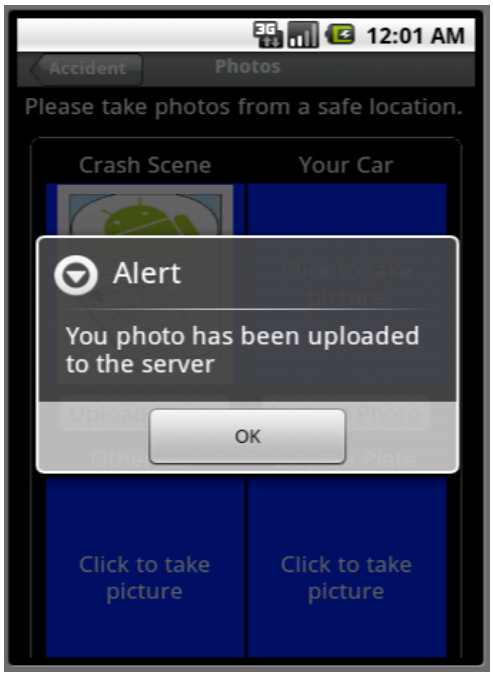

Illustration 67: Successful Upload of Image to Server.

2.15 Done

You have completed our Mysurance application. If you have time and want to modify the application, you can always restore your HTML file to the various steps by copying content from the website.

3.0 Links for Further Learning

PhoneGap

Main site: http://www.phonegap.com
Wiki: http://wiki.phonegap.com
Roadmap: http://wiki.phonegap.com/roadmap-planning
Forums:
    http://groups.google.com/group/phonegap
    http://groups.google.com/group/phonegap-dev
GIT source code: https://github.com/phonegap
Compile as a service: http://build.phonegap.com
Twitter: http://twitter.com/phonegap