



Learn to Accelerate Your Web App Development with WebSphere Liberty Lab Instructions

Objective

In this lab, you learn:

- How to set up a Liberty runtime environment
- How to create a server to run in this environment
- How to deploy a simple web application to this server
- How to build a simple web application using JAX-RS, CDI and JPA

Prerequisite Knowledge

To get the most out of this lab, knowledge of the following areas is useful

- Basic knowledge of Java EE
- Basic familiarity with the Eclipse IDE

Registration Application

In this exercise you will set up the WebSphere Application Server Liberty Profile and create a web application to run on it. You will learn how to create an application that uses Java RESTful Services (JAX-RS), Context and Dependency Injection (CDI) and Java Persistence API (JPA) to implement a simple registration application. The registration application will be able to register the name and email of the attendee you would like to add to an event, and display a list of all registered attendees. We will also include additional functionality to remove attendees from the list.

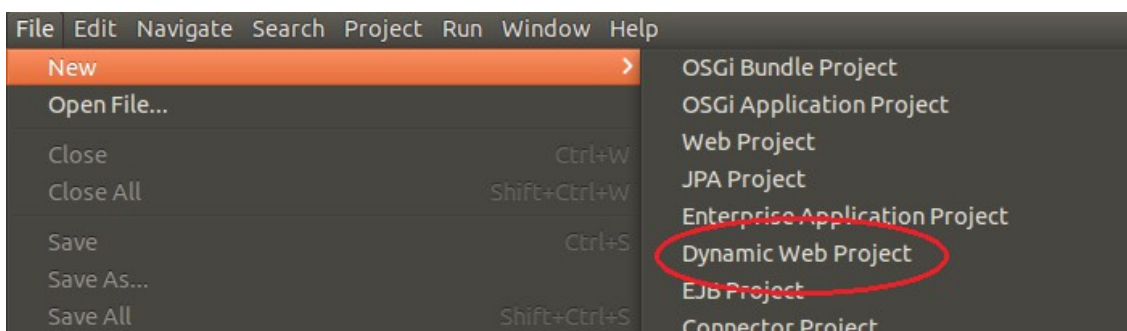
Getting Started

1. Download and install WebSphere Developer Tools for Eclipse. Instructions can be found on WASdev.net
2. Launch the Eclipse IDE. If you are prompted to provide a workspace location, provide a path to an empty folder that can be used to store your work with Eclipse and click OK.

Creating the server

We will start by setting up our Liberty runtime environment and creating a server that our application can run on. We will be using the WAS Liberty with Java EE7 Web Profile runtime and will set it up using the archive package.

1. Create a new Dynamic Web Project by clicking **File > New > Dynamic Web Project**.



2. Enter the name of the project as `RegistrationAppWeb`. Ensure that **Dynamic web module version** is set to 3.1, and ensure that **Add project to an EAR** is not checked.

3. Under Target runtime click **New Runtime**.

New Dynamic Web Project

Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: RegistrationAppWeb

Project location
 Use default location
Location: /home/lab/workspace/RegistrationAppWeb

Target runtime
<None> **New Runtime...**

Dynamic web module version
3.1

Configuration
<custom> **Modify...**

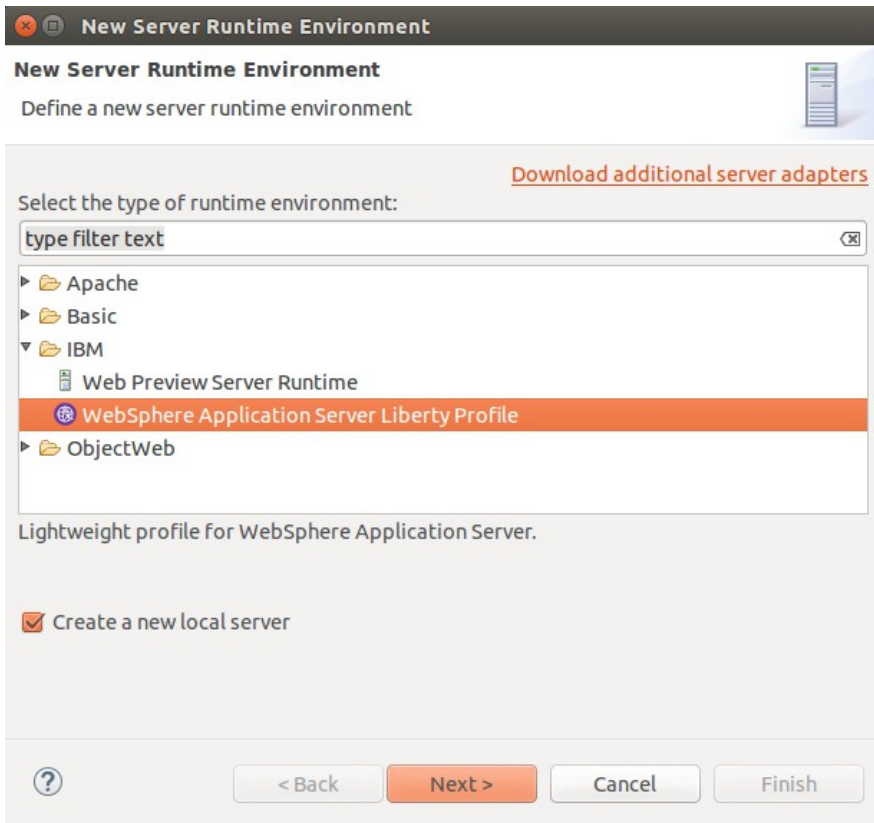
Hint: Get started quickly by selecting one of the pre-defined project configurations.

EAR membership
 Add project to an EAR
EAR project name: RegistrationAppWebEAR **New Project...**

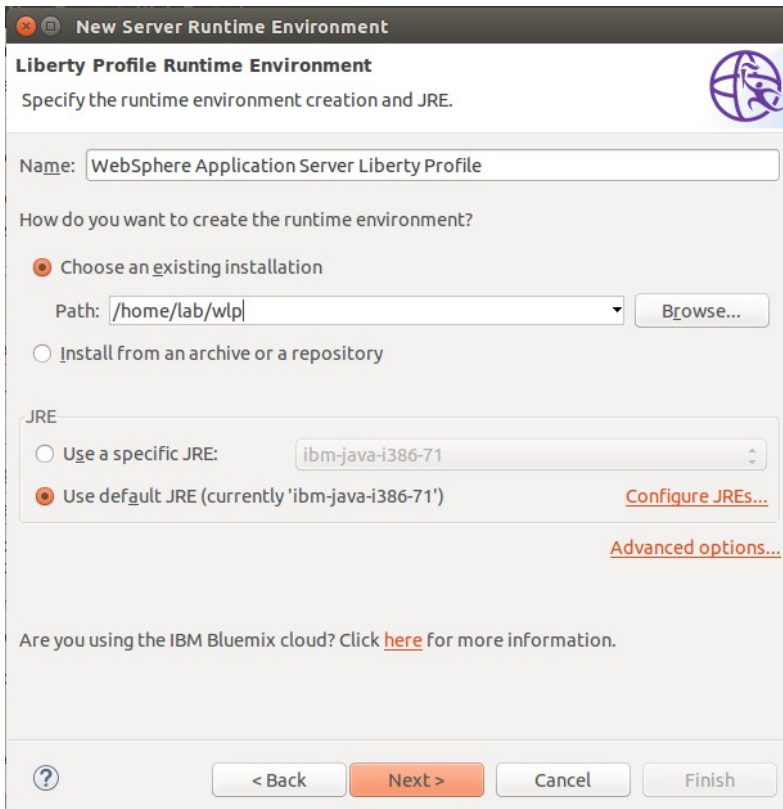
Working sets
 Add project to working sets
Working sets: **Select...**

< Back **Next >** **Cancel** **Finish**

- Expand **IBM**, and select **WebSphere Application Server Liberty Profile**. Check **Create a new local server**. Click **Next**.



- At this stage you can either point to an existing Liberty installation or download the Liberty runtime from the Liberty Repository.
- Click **Next**.



7. Leave the **Server name** as defaultServer and click **Finish**.

New Server Runtime Environment

New Liberty Profile Server

Specify the name of the new server.

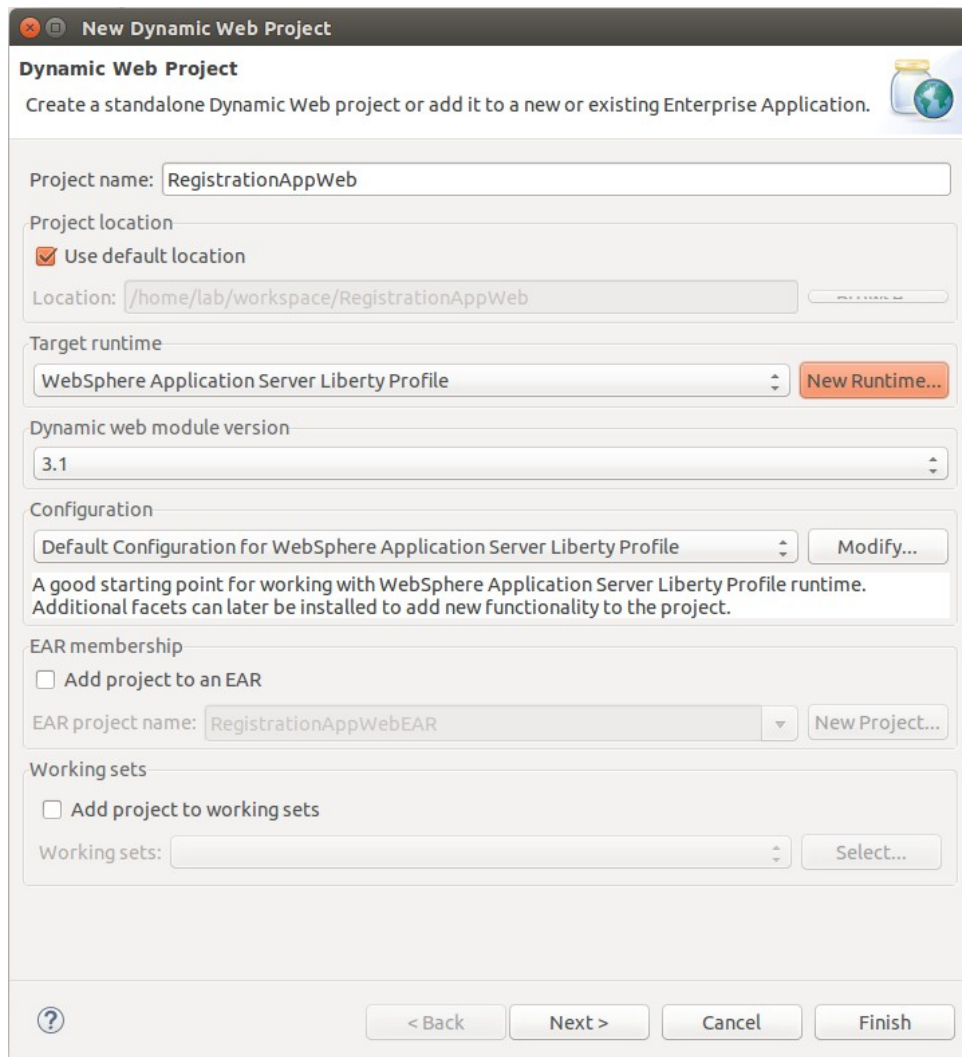
User directory: WebSphere Application Server Liberty Profile

Server name: defaultServer

Template: defaultServer

? < Back Next > Cancel Finish

8. On the **New Dynamic Web Project** window click **Finish**.

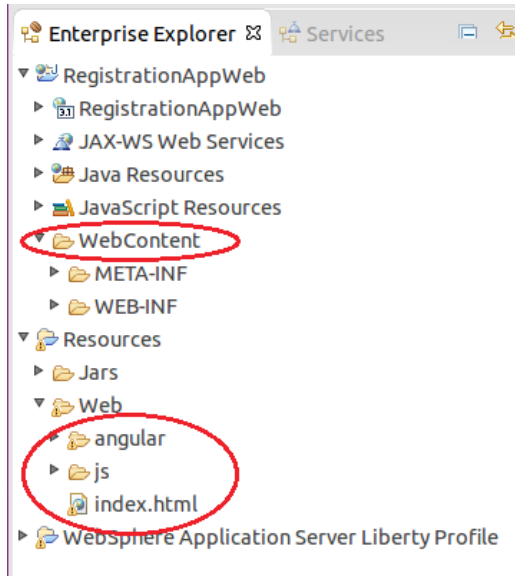


9. At this point Eclipse may suggest that you switch to the Web perspective, select no.

Creating the web page

For our web application we are going to use client-side technologies such as Angular and Javascript. The code for the web page has already been provided for you and can be found in the Resources folder which was provided alongside this document.

1. Copy the `Angular` folder, the `js` folder and the `index.html` file from the `Web` folder of the **Resources** project by dragging it into the `WebContent` directory of the **RegistrationAppWeb** project.

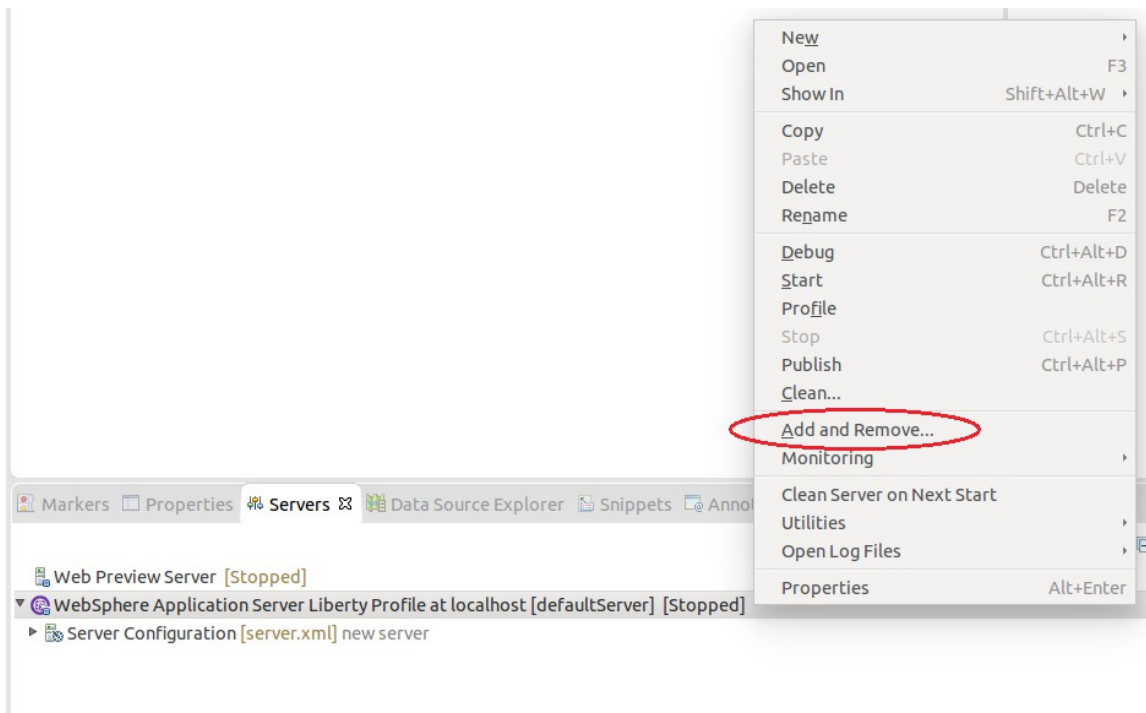


Running the application

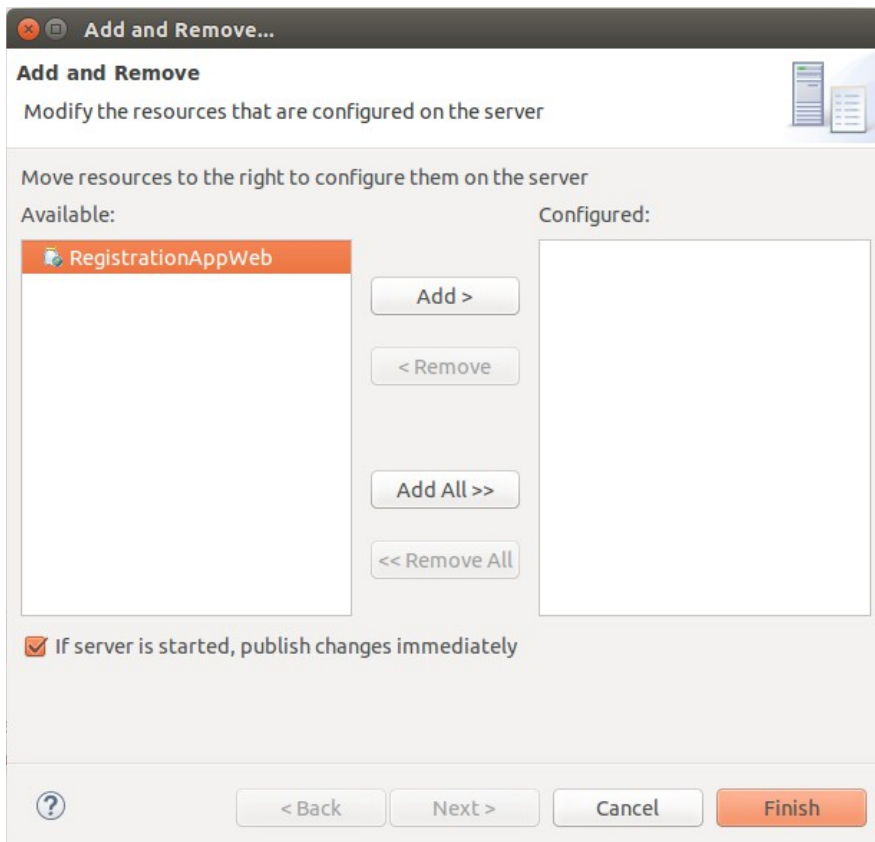
Now that we have set up a server and created the web page for our application, we can go ahead and start the server to test it out.

1. Go to the **Servers** view in Eclipse.

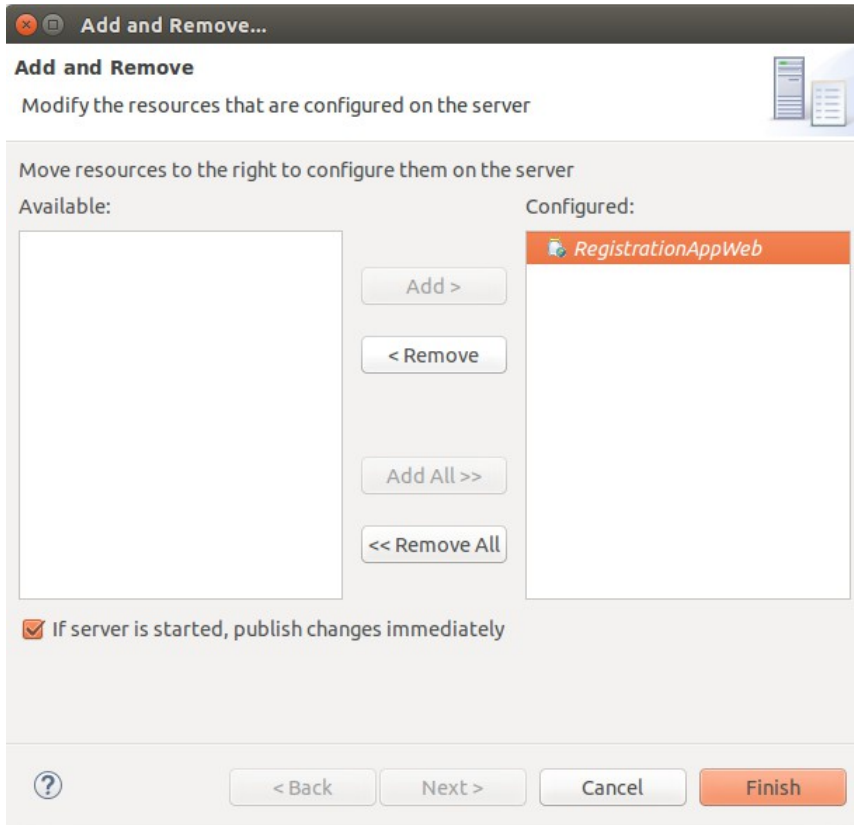
2. Right click on the server and select **Add and Remove...**



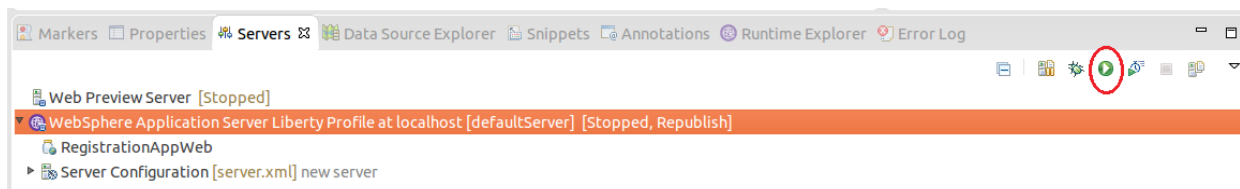
3. Select the **RegistrationWebApp** application and click **Add**.



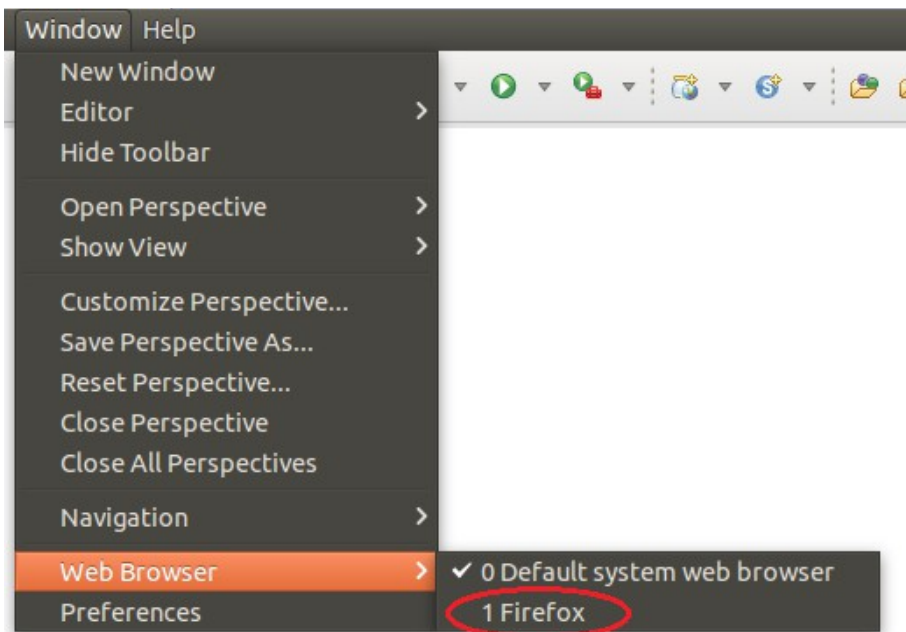
4. Click **Finish**.



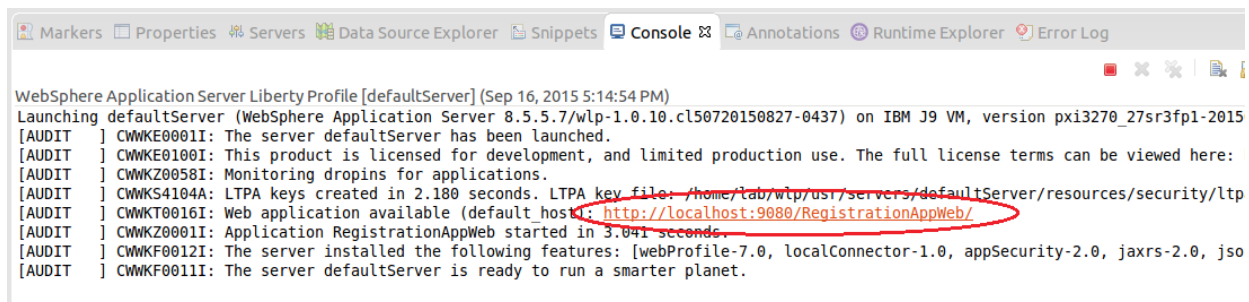
5. With the server selected, click the green start button in the **Servers** view to start the server.



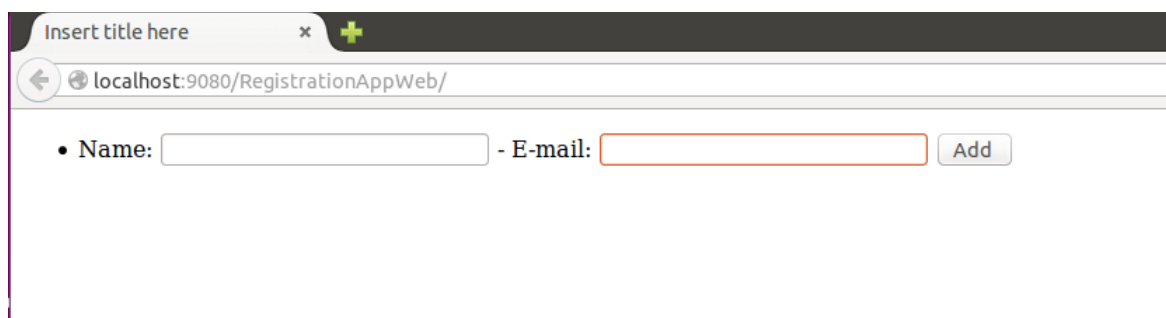
- Go to **Window > Web Browser > Firefox**. This will ensure that Eclipse will launch our web application in the Firefox web browser.



- From the **Console** view, click on the link to our web application to open the application in a browser.



- We now have the client available to us in the browser. Enter the details of an attendee and click the add button.



At the moment our application does not have the functionality to register an attendee. If you look in the **Console** view in Eclipse, you should see the following error:

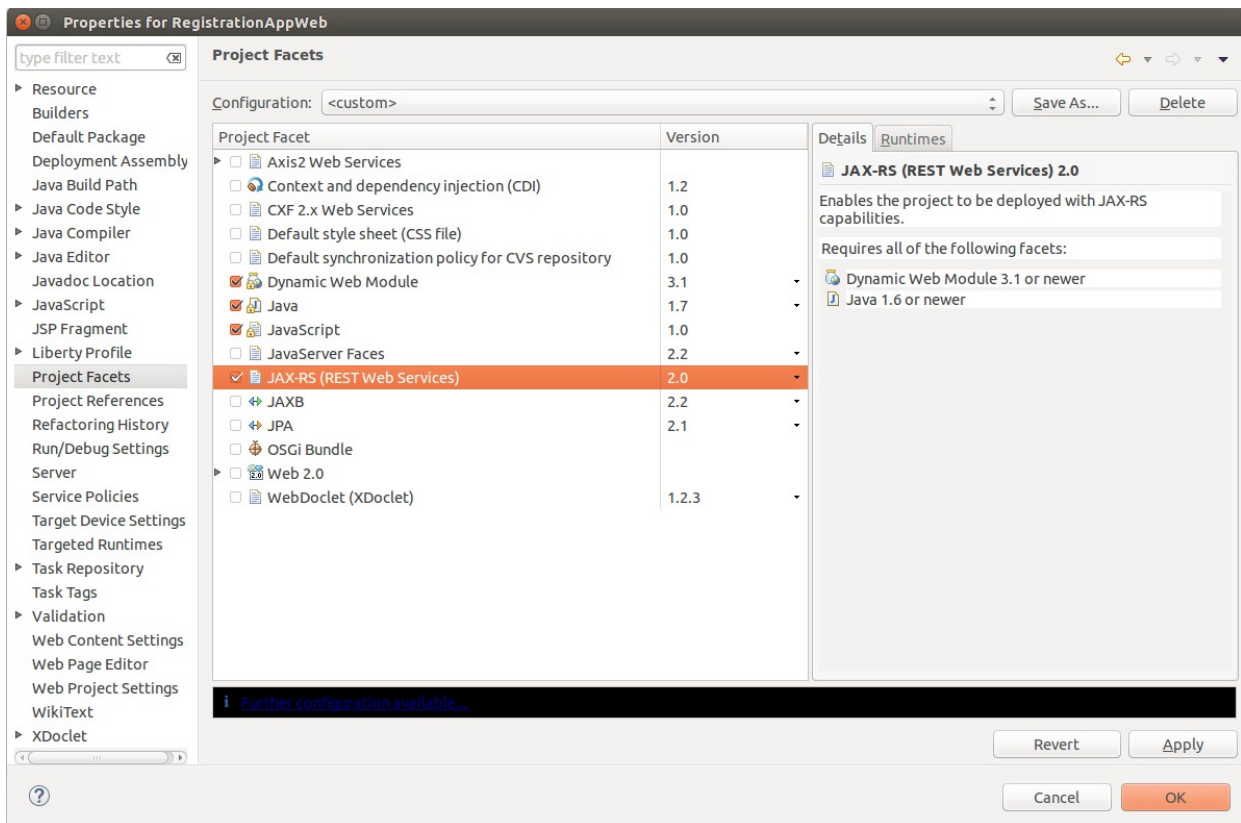
```
[WARNING] SRVE0190E: File not found: /api/attendees
```

The error occurs because the client is making a REST API call to try and register the attendee, but it cannot find the JAX-RS endpoint named **attendees** on the path **/api/attendees**. We are going to solve this problem by creating a JAX-RS interface that will handle the clients requests.

Adding RESTful services using JAX-RS

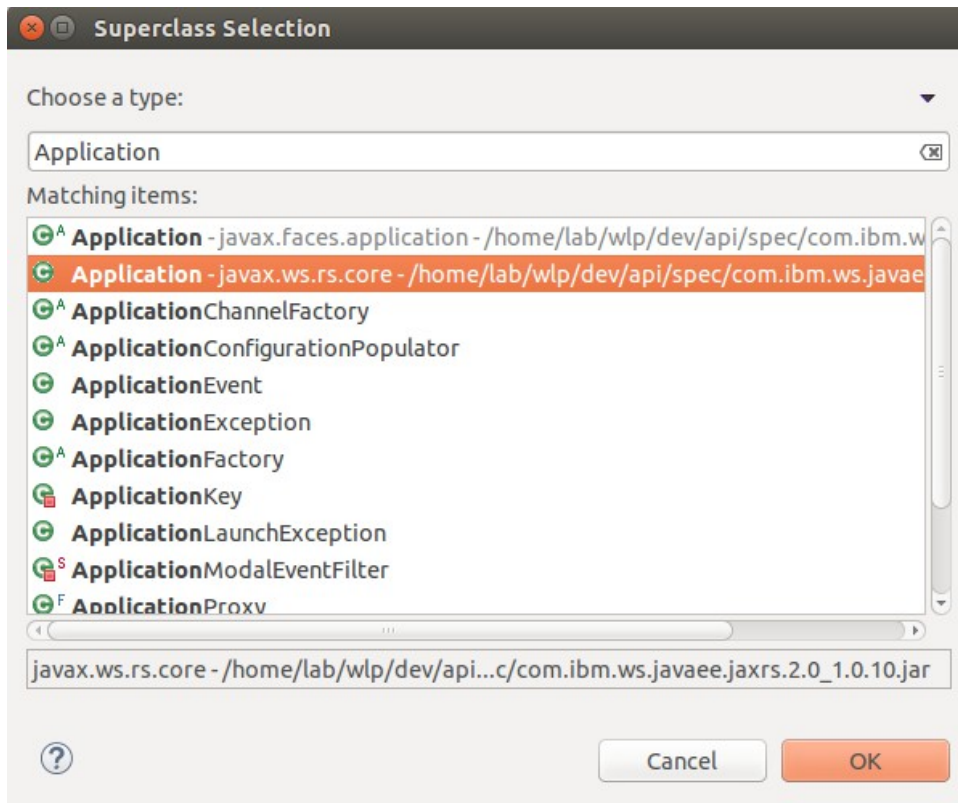
JAX-RS allows you to write a RESTful interface that uses Java objects to communicate with clients. Web requests are made to classes with annotations that identify them as handling different HTTP requests. The JAX-RS application consists of a class that handles making the application available and a class that performs the request handling. We will first create the class that makes our application available.

1. Right click on the **RegistrationAppWeb** project and select **Properties**.
2. On the Project Facets, check the checkbox for **JAX-RS (REST Web Services)** and set the version to **2.0**. Click **OK**.



3. We will now create the class that performs request handling. Right click on **RegistrationAppWeb**, select **New > Class**.
4. Set the package to `net.wasdev.reg` and the name to `RegApp`.

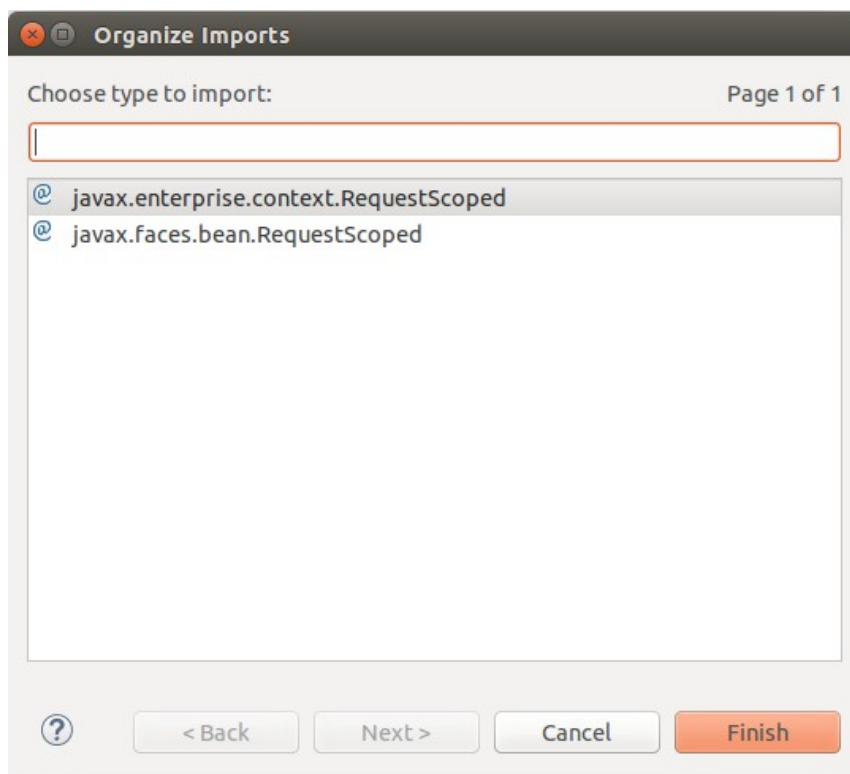
5. Click the **Browse** button next to the **Superclass** field. Using the filter box, type `Application`, and select the **Application** class from the `javax.ws.rs.core` package. Click **OK**.



6. Click **Finish** on the **New Java Class** window.
7. Add the following code above the class declaration in the **RegApp** class.
`@ApplicationPath("/api")`
8. To resolve the import errors, use the **control-shift-o** keyboard shortcut. Use **control-s** to save the class.
9. We have now created the class that makes our application available. It should resemble the image below.

```
RegApp.java
1 package net.wasdev.reg;
2
3 import javax.ws.rs.ApplicationPath;
4 import javax.ws.rs.core.Application;
5
6 @ApplicationPath("/api")
7 public class RegApp extends Application {
8
9 }
10
```

10. We will now create the class that performs the request handling. Right click on **RegistrationAppWeb**, select **New > Class**.
11. Set the package to `net.wasdev.reg` and the name to `Attendees`. Click **Finish**.
12. Add the following annotations to the class declaration.
`@Path("/attendees")`
`@RequestScoped`
13. To resolve the import errors, use the **control-shift-o** keyboard shortcut. This will cause a prompt for ambiguous imports.
14. For the **RequestScoped** annotation, select **javax.enterprise.context.RequestScoped**. Click **Finish**. Use **control-s** to save the class.



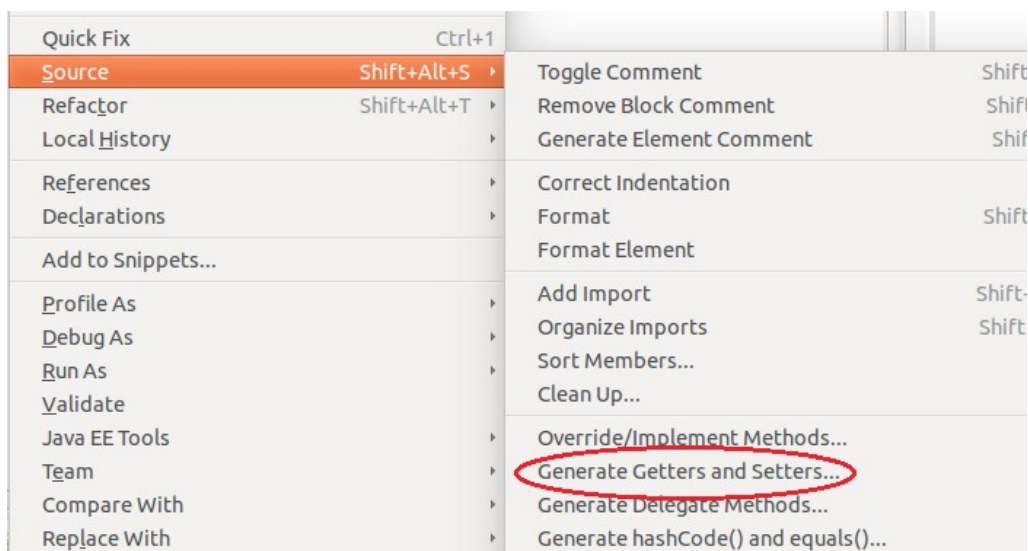
15. We have now created the class that performs the request handling. It should resemble the image below.

```
1 package net.wasdev.reg;
2
3
4 import javax.enterprise.context.RequestScoped;
5
6 import javax.ws.rs.Path;
7
8 @RequestScoped
9 @Path("/attendees")
10 public class Attendees {
11
12 }
13
```

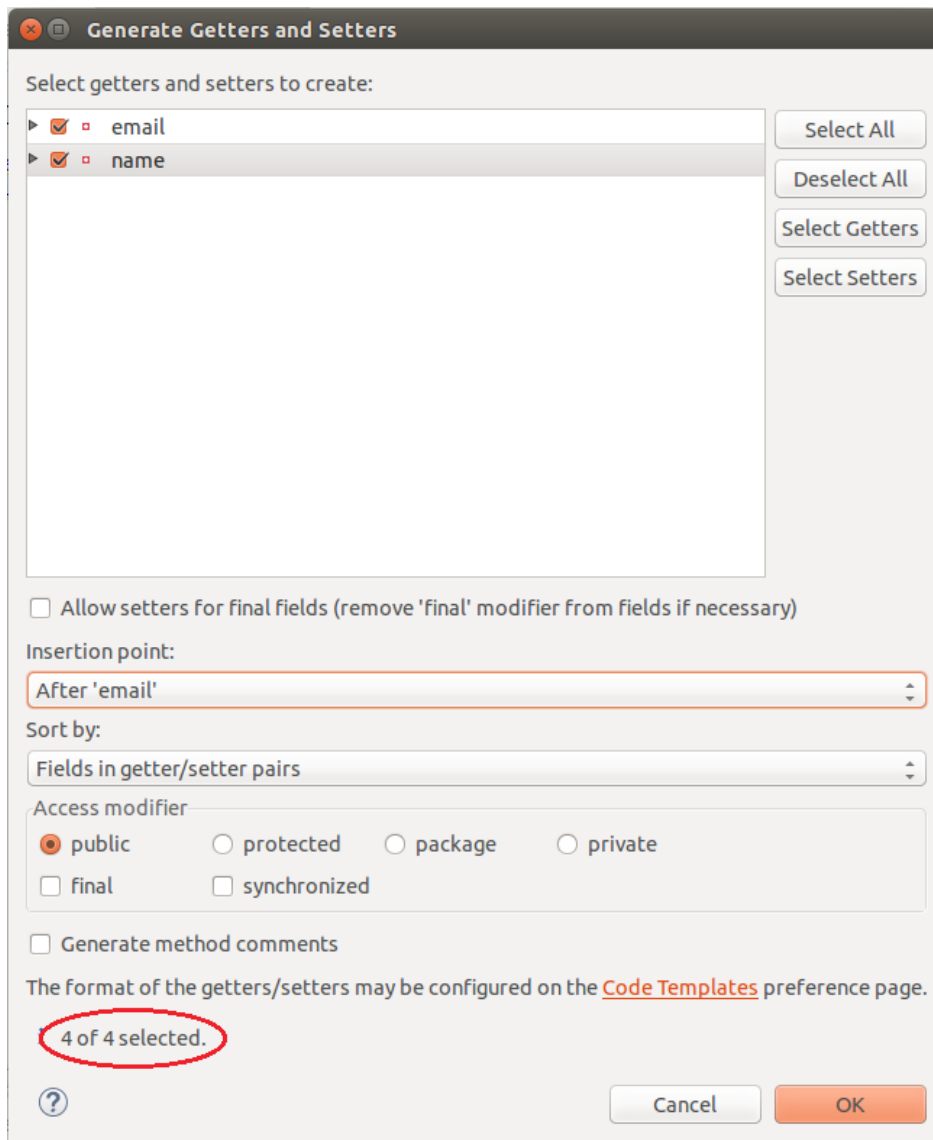
Creating a Java object to represent Attendee

Before we write the request handlers for our REST API, we will need a Java Object that will be used to store information about the attendee to pass to the client.

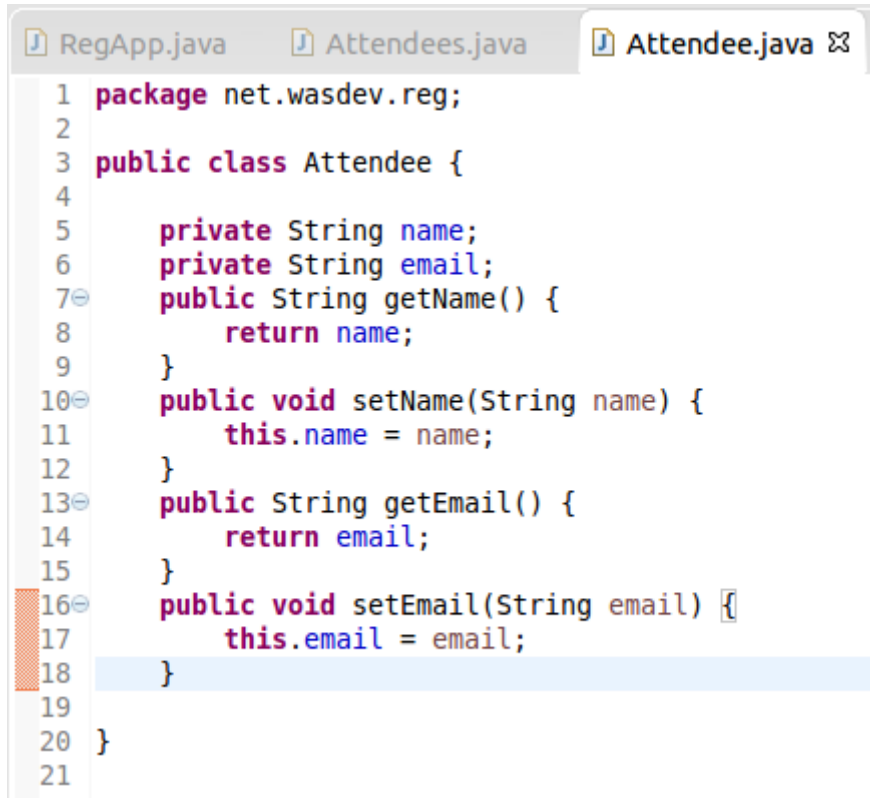
1. Right click on **RegistrationAppWeb**, select **New > Class**.
2. Set the package to `net.wasdev.reg` and the name to `Attendee`. Click **Finish**.
3. Add the following fields to the **Attendee** class.
`private String name;`
`private String email;`
4. Right click on the **Attendee.java** class file. Select **Source > Generate Getters and Setters**.



5. Check the boxes next to **name** and **email**. Set the Insertion point to **After 'email'**. Ensure that the message **4 of 4 selected** appears at the bottom of the window to indicate that you have chosen to generate getters and setters for both fields. Click **OK**.



6. Use **control-s** to save the class. It should now resemble the image below.



```
1 package net.wasdev.reg;
2
3 public class Attendee {
4
5     private String name;
6     private String email;
7     public String getName() {
8         return name;
9     }
10    public void setName(String name) {
11        this.name = name;
12    }
13    public String getEmail() {
14        return email;
15    }
16    public void setEmail(String email) {
17        this.email = email;
18    }
19
20 }
21
```

7. We have now set up our JAX-RS interface which the client can communicate with. The application will have automatically updated after the changes were saved. Enter another attendee using the web interface, you should see the following error message in the console.

```
[WARNING ] No resource methods have been found for resource class net.wasdev.reg.Attendees
[ERROR   ] No resource classes found
```

Creating a JAX-RS POST request handling method

The error shows that no resource methods have been found. The resources the client is looking for are the request handlers that will respond to different HTTP requests. The first one we need to add will handle POST requests. These will be used to register the details of an attendee. For now, we will store the details of our attendees in an ArrayList in the JAX-RS request handler class.

1. Go to the **Attendees** class and add the following field.
`private ArrayList<Attendee> attendees = new ArrayList<Attendee>();`
2. Add the following method into the **Attendees** class to add an attendee to the list:
`public void addAttendee(Attendee attendee) {
 attendees.add(attendee);
}`
3. To call the method when receiving a HTTP POST request, and accept an Attendee object as JSON, add the following annotations to the method:
`@POST
@Consumes(MediaType.APPLICATION_JSON)`

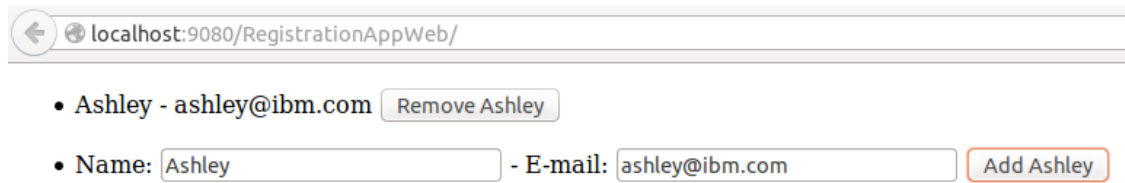
Our application will now add an attendee to the ArrayList every time it receives a post request, however we will not be able to see evidence of this in the browser. To change this we need our POST request handler to return the list of attendees to the client.

4. To return the updated list to the client add the following annotation to the method:
`@Produces(MediaType.APPLICATION_JSON)`
5. We also need to return the list of attendees from the method. Add the following code to the end of the method body:
`return attendees;`
6. Change the method header from `void` to `List<Attendee>`.
7. To resolve any import errors, use the **control-shift-o** keyboard shortcut.
8. For **MediaType**, select `javax.ws.rs.core.MediaType`. Click **Next**.
9. For **Produces**, select `javax.ws.rs.Produces`. Click **Next**.
10. For **List**, select `java.Util.List`. Click **Finish**. Use **control-s** to save the class.
11. The **Attendees** class should now resemble the following image.



```
1 package net.wasdev.reg;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import javax.enterprise.context.RequestScoped;
7 import javax.ws.rs.Consumes;
8 import javax.ws.rs.POST;
9 import javax.ws.rs.Path;
10 import javax.ws.rs.Produces;
11 import javax.ws.rs.core.MediaType;
12
13 @RequestScoped
14 @Path("/attendees")
15 public class Attendees {
16
17     private ArrayList<Attendee> attendees = new ArrayList<Attendee>();
18
19     @POST
20     @Consumes(MediaType.APPLICATION_JSON)
21     @Produces(MediaType.APPLICATION_JSON)
22     public List<Attendee> addAttendee(Attendee attendee){
23         attendees.add(attendee);
24         return attendees;
25     }
26
27 }
28
```

12. Register an attendee through the browser. We can now see that they get displayed.



If you now try and register another attendee you will notice that they replace the previous attendee, rather than being appended to the list. This is because the JAX-RS class works on a request scoped basis, meaning that every time a request is made a new resource is made, and we lose the data stored in the `ArrayList`. To solve this we will create a CDI bean which will work on an application scope and be a better place to store our data. It is also good practice to separate the business logic from our request handlers, and this can be done using a CDI bean.

Creating a CDI Bean

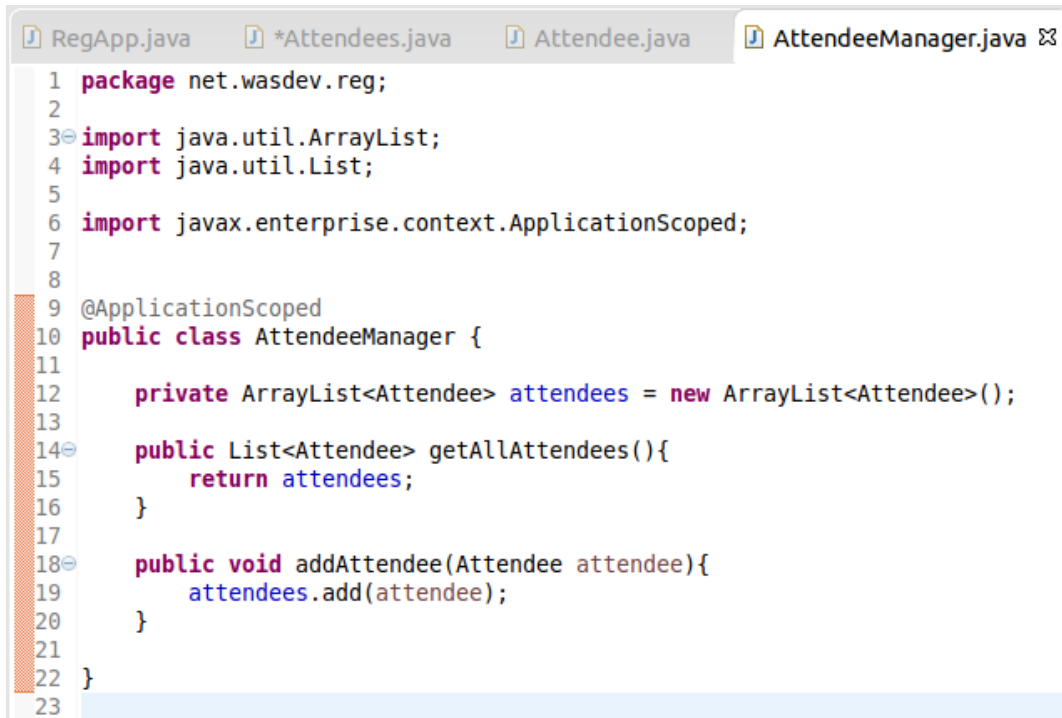
1. Right click on **RegistrationAppWeb**, select **New > Class**.
2. Set the package to `net.wasdev.reg` and the name to `AttendeeManager`.
3. Add the following annotation to the class declaration:
`@ApplicationScoped`
4. We will now store our list of attendees in the CDI bean and perform any operations on the list from within this bean. Add the following code inside the **AttendeeManager** class body:

```
private ArrayList<Attendee> attendees = new ArrayList<Attendee>();

public List<Attendee> getAllAttendees() {
    return attendees;
}

public void addAttendee(Attendee attendee) {
    attendees.add(attendee);
}
```
5. To resolve the import errors, use the **control-shift-o** keyboard shortcut.
6. For **ApplicationScoped**, select `javax.enterprise.context.ApplicationScoped`.
7. For **List**, select `java.Util.List`. Click **Finish**. Use **control-s** to save the class.

8. The **AttendeeManager** class should resemble the image below.



```
1 package net.wasdev.reg;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import javax.enterprise.context.ApplicationScoped;
7
8
9 @ApplicationScoped
10 public class AttendeeManager {
11
12     private ArrayList<Attendee> attendees = new ArrayList<Attendee>();
13
14     public List<Attendee> getAllAttendees(){
15         return attendees;
16     }
17
18     public void addAttendee(Attendee attendee){
19         attendees.add(attendee);
20     }
21
22 }
23
```

9. Now we need to make our JAX-RS class call our CDI bean to operate on the list of attendees rather than doing it within the JAX-RS class. Open the **Attendees** class. Remove the following code:

```
private ArrayList<Attendee> attendees = new ArrayList<Attendee>();
```

10. To make the CDI bean available to the JAX-RS class, add the following code to the top of the **AttendeeManager** class body:

```
@Inject
AttendeeManager attendeeManager;
```

11. Replace the contents of the **addAttendee** method body with the following code:

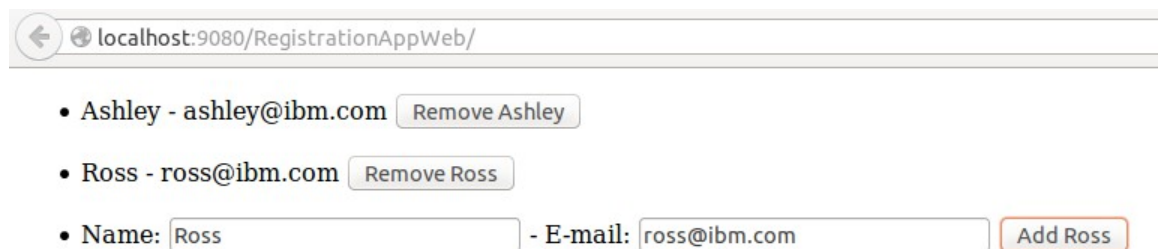
```
attendeeManager.addAttendee(attendee);
return attendeeManager.getAllAttendees();
```

12. To resolve the import errors, use the **control-shift-o** keyboard shortcut. Use **control-s** to save the class.

13. The **Attendees** class should resemble the image below.

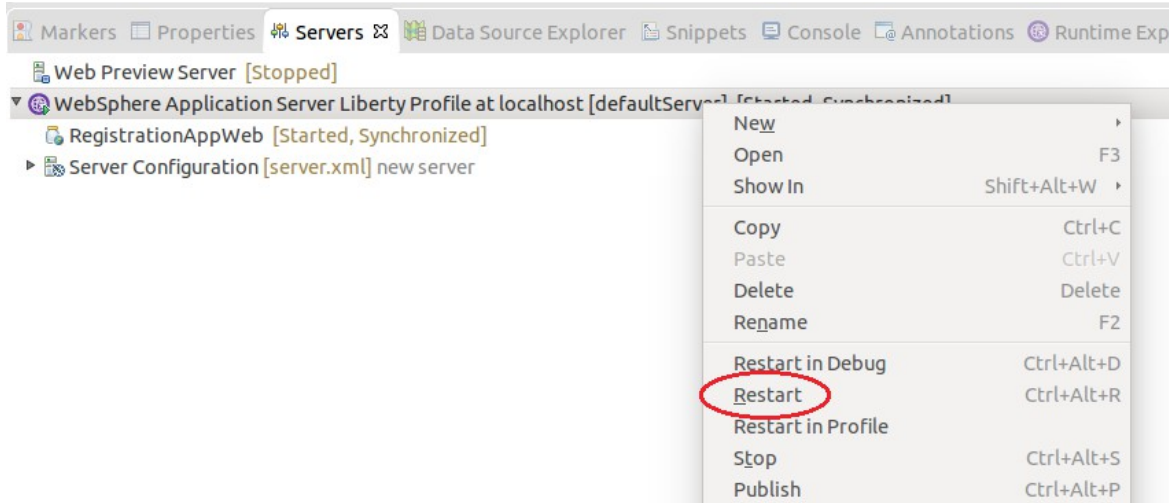
```
RegApp.java Attendees.java Attendee.java AttendeeMan
4
5 import javax.enterprise.context.RequestScoped;
6 import javax.inject.Inject;
7 import javax.ws.rs.Consumes;
8 import javax.ws.rs.POST;
9 import javax.ws.rs.Path;
10 import javax.ws.rs.Produces;
11 import javax.ws.rs.core.MediaType;
12
13 @RequestScoped
14 @Path("/attendees")
15 public class Attendees {
16
17     @Inject
18     AttendeeManager attendeeManager;
19
20     @POST
21     @Consumes(MediaType.APPLICATION_JSON)
22     @Produces(MediaType.APPLICATION_JSON)
23     public List<Attendee> addAttendee(Attendee attendee){
24         attendeeManager.addAttendee(attendee);
25         return attendeeManager.getAllAttendees();
26     }
27
28 }
```

14. Register two attendees through the browser. We can now see that they both get displayed and the information in the list is retained beyond each request.

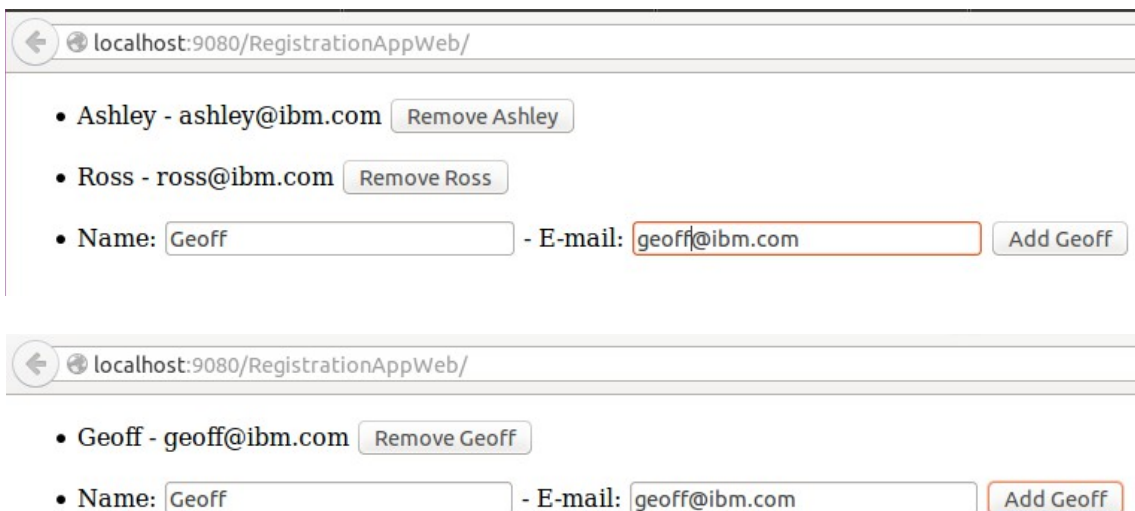


15. Go to the **Servers** view in Eclipse.

16. Right click on the server and select **Restart**.



17. Return to the browser and register another attendee.



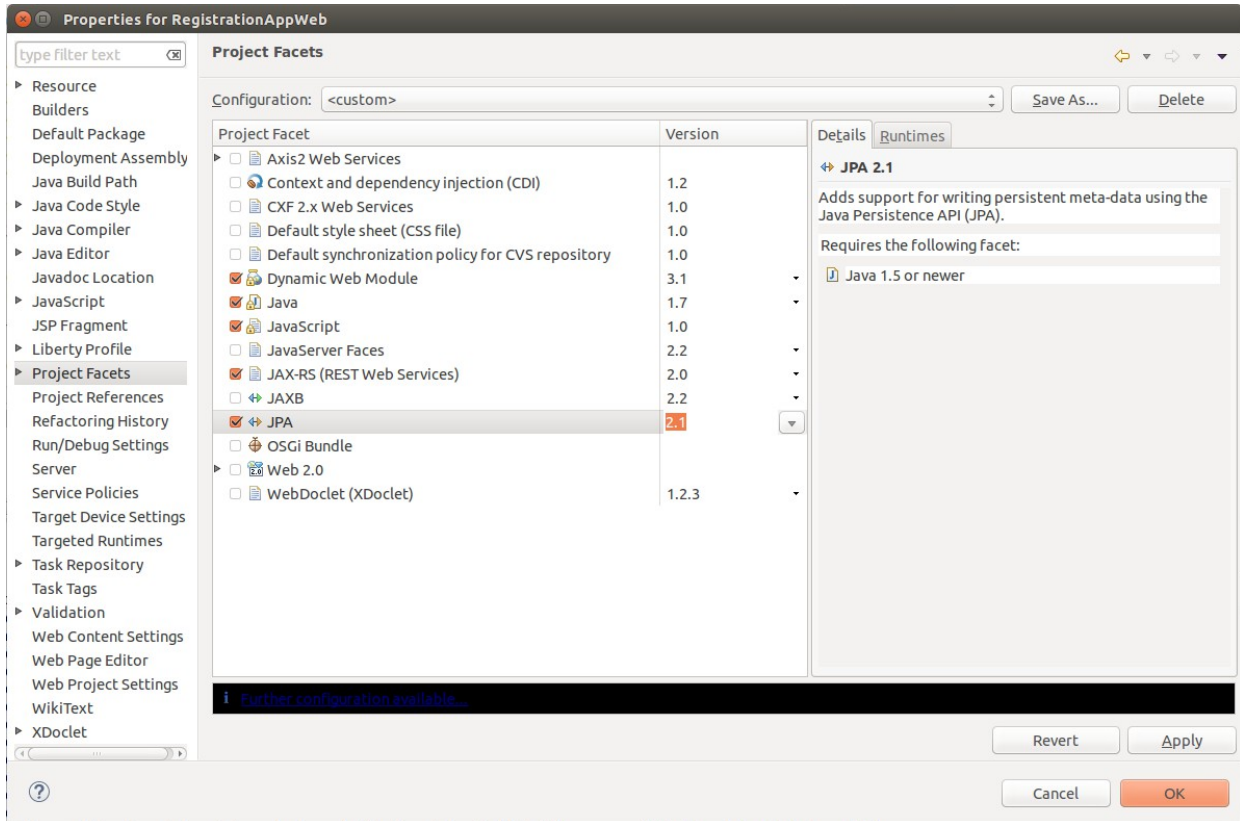
We can see that the two attendees previously registered have been replaced by the attendee we have just registered. This is because the data stored in the CDI bean will only persist as long as the application is running, once the server is restarted that data is lost. For more persistent storage we need our application to communicate with a database.

Creating a JPA Entity

For our application we will use JPA to persist our data and store it in a database. By designating our Attendee Java object as a JPA entity we can store the object's fields in a database.

1. Right click on the **RegistrationAppWeb** project and select **Properties**.

2. On the Project Facets, check the checkbox for **JPA** and set the version to **2.1**. Click **OK**.



3. Open the **Attendee** class.

4. To define the **Attendee** class as a JPA entity add the following annotation to the class declaration:

```
@Entity
```

5. We need to designate one of the fields in our **Attendee** class as the primary key for the database entry. To do this add the following annotation to the email field:

```
@Id
```

6. To resolve the import errors, use the **control-shift-o** keyboard shortcut. Use **control-s** to save the class.

7. The **Attendee** class should resemble the image below.

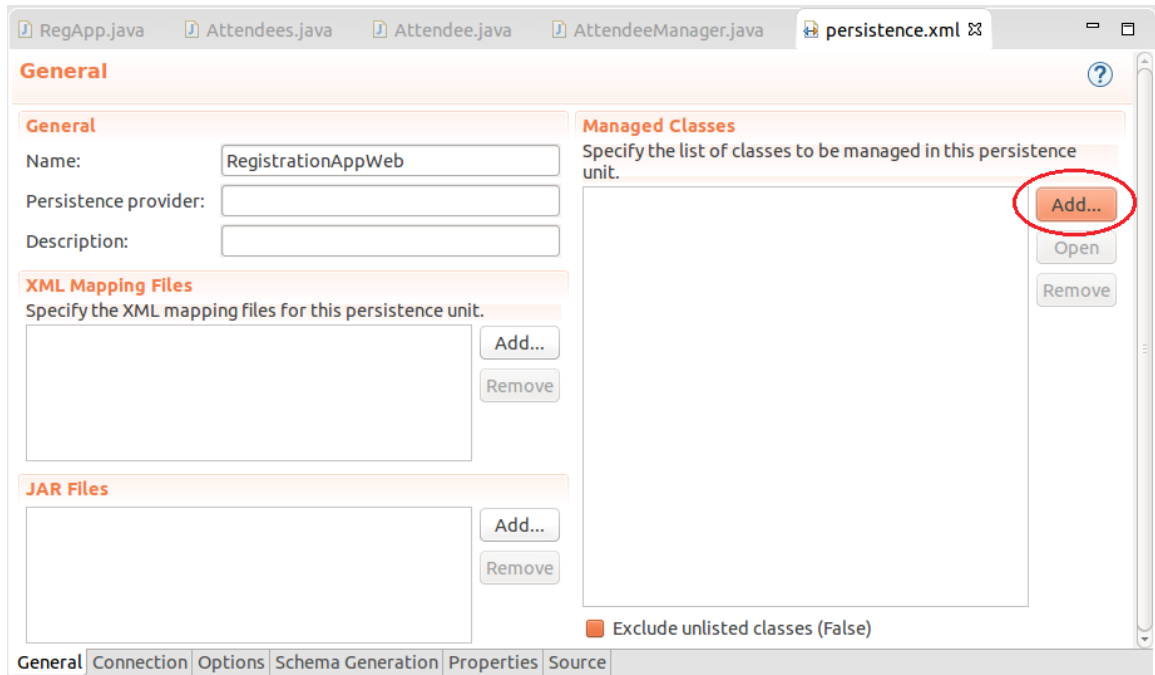
```
RegApp.java  Attendees.java  Attendee.java ✕
1  package net.wasdev.reg;
2
3  import javax.persistence.Entity;
4  import javax.persistence.Id;
5
6  @Entity
7  public class Attendee {
8
9      private String name;
10     @Id
11     private String email;
12     public String getName() {
13         return name;
14     }
15     public void setName(String name) {
16         this.name = name;
17     }
18     public String getEmail() {
19         return email;
20     }
21     public void setEmail(String email) {
22         this.email = email;
23     }
24 }
25 }
26
```

You may see an error in the console stating **Class "net.wasdev.reg.Attendee" is managed, but is not listed in the persistence.xml file.** This is because JPA uses Persistence Units to define the types of data required to be stored by your application and our class is not listed in the Persistence Unit. We will configure this in an XML file called `persistence.xml`, which was created for us when the JPA project Facet was added. This will resolve our error.

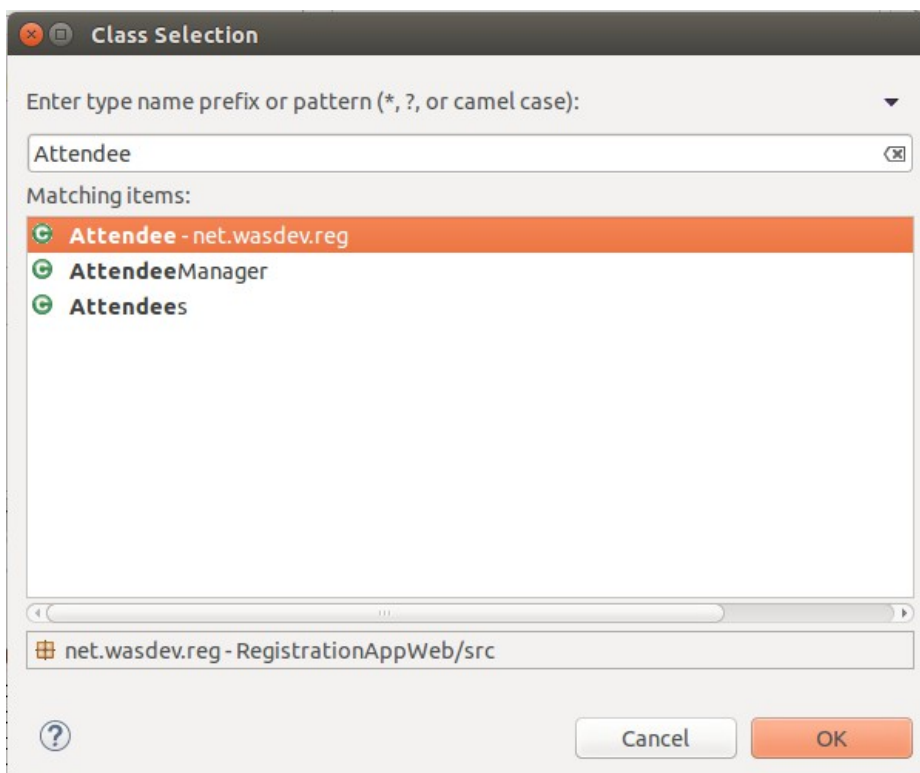
1. Expand the JPA Content section of the RegistrationAppWeb project, and double click on the persistence.xml file.



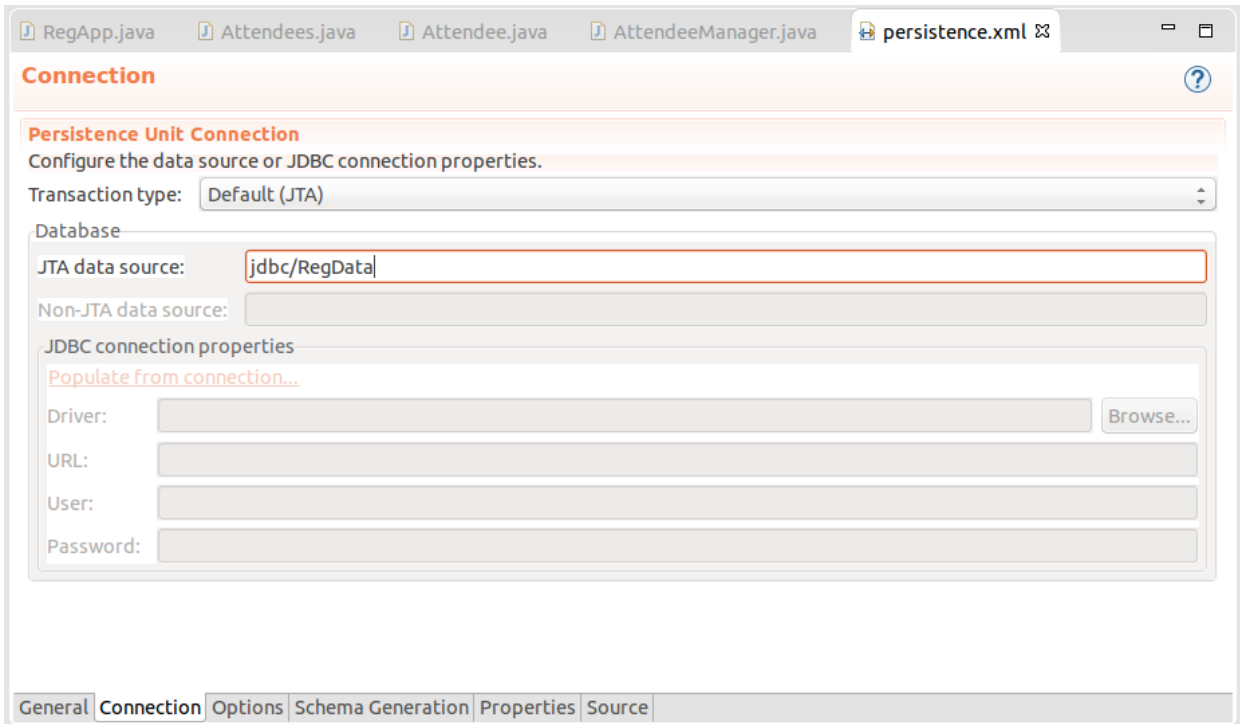
2. Under the Managed Classes section, click **Add**.



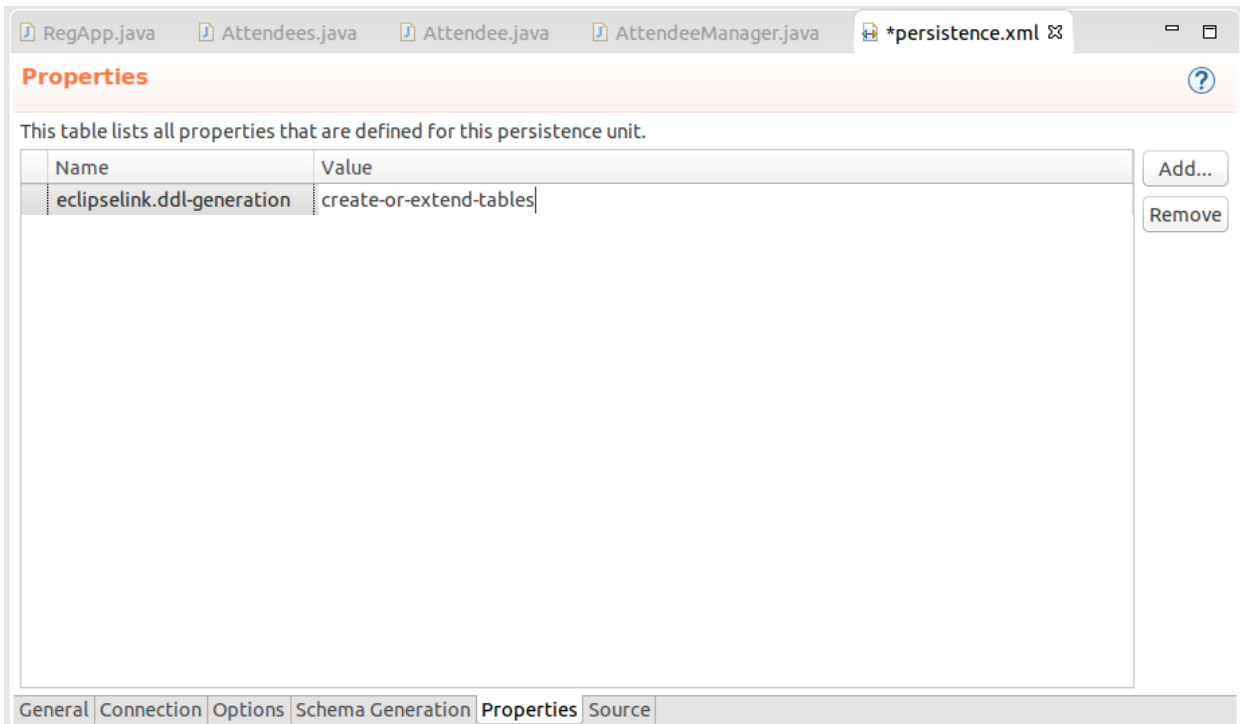
3. Type Attendee in the filter box and select the Attendee class. Click **OK**.



- Switch to the **Connection** tab.
- Set the **JTA data source** to jdbc/RegData



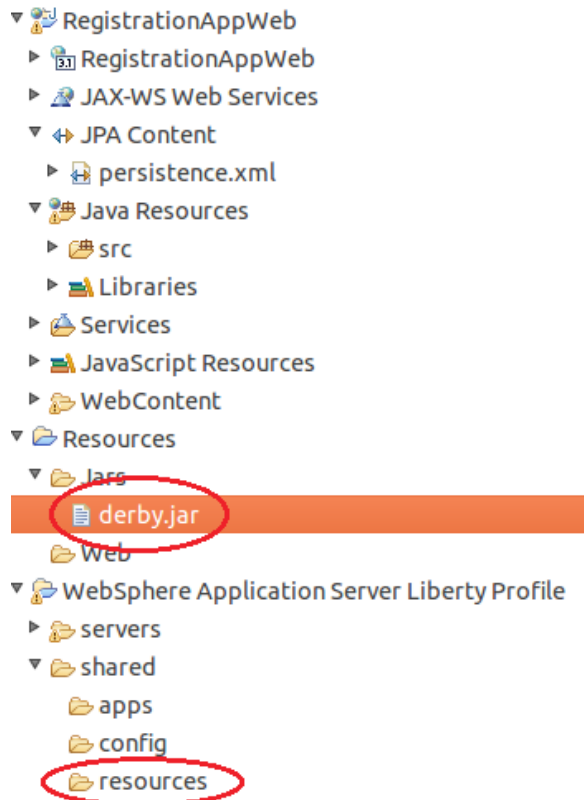
- Switch to the Properties tab, and click the **Add** button.
- Set the **Name** to *eclipselink.ddl-generation*, and the **Value** to *create-or-extend-tables*.



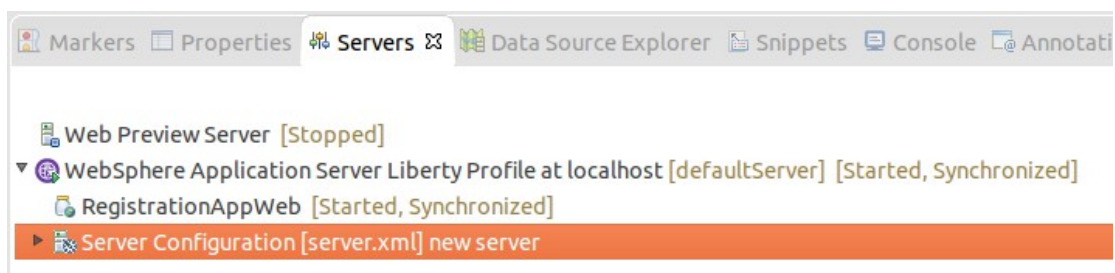
- Use the **control-S** shortcut to save the file.

Setting up the database

1. Copy the `derby.jar` file from the `Jars` folder of the **Resources** project by dragging it into the `shared/resources` directory of the **WebSphere Application Server Liberty Profile** project.



2. Go to the **Servers** view in Eclipse.
3. Double click on **Server Configuration** under our server.



4. Switch to the **Source** tab. Add the following code inside the server tags:

```
<dataSource jndiName="jdbc/RegData">
  <properties.derby.embedded databaseName="${shared.resource.dir}/regData"
createDatabase="create"/>
  <jdbcDriver>
    <library>
      <file name="${shared.resource.dir}/derby.jar" />
    </library>
  </jdbcDriver>
</dataSource>
```

5. Use the **control-S** shortcut to save the file.
6. The **server.xml** class should resemble the image below.

```

1 <server description="new server">
2
3     <!-- Enable features -->
4     <featureManager>
5         <feature>webProfile-7.0</feature>
6         <feature>localConnector-1.0</feature>
7     </featureManager>
8
9     <!-- To access this server from a remote client add a host attribute to the following element, e.g. host="*" -->
10    <httpEndpoint httpPort="9080" httpsPort="9443" id="defaultHttpEndpoint"/>
11
12
13    <applicationMonitor updateTrigger="mbean"/>
14
15    <webApplication id="RegistrationAppWeb" location="RegistrationAppWeb.war" name="RegistrationAppWeb"/>
16
17    <dataSource jndiName="jdbc/RegData">
18        <properties.derby.embedded databaseName="${shared.resource.dir}/regData" createDatabase="create"/>
19        <jdbcDriver>
20            <library>
21                <file name="${shared.resource.dir}/derby.jar" />
22            </library>
23        </jdbcDriver>
24    </dataSource>
25
26 </server>

```

We now need our CDI bean to store the Attendees in the database rather than the ArrayList.

1. Open the **AttendeeManager** class. Remove the following code:
`private ArrayList<Attendee> attendees = new ArrayList<Attendee>();`
2. Add the following code underneath the class declaration:
`@PersistenceContext(unitName = "RegistrationAppWeb")
EntityManager em;`
3. Replace the body of the **getAllAttendees** method with the following code:
`CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Attendee> cq = cb.createQuery(Attendee.class);
Root<Attendee> rootEntry = cq.from(Attendee.class);
CriteriaQuery<Attendee> all = cq.select(rootEntry);
TypedQuery<Attendee> allQuery = em.createQuery(all);
return allQuery.getResultList();`
4. Replace the body of **addAttendee** with the following code:
`em.persist(attendee);`
5. To resolve the import errors, use the **control-shift-o** keyboard shortcut.
6. For **Root**, select **javax.persistence.criteria.Root**. Click **Finish**. Use **control-s** to save the class.

7. The **AttendeeManager** class should resemble the image below.

```
RegApp.java Attendees.java Attendee.java AttendeeManager.java x
1 package net.wasdev.reg;
2
3 import java.util.List;
4
5 import javax.enterprise.context.ApplicationScoped;
6 import javax.persistence.EntityManager;
7 import javax.persistence.PersistenceContext;
8 import javax.persistence.TypedQuery;
9 import javax.persistence.criteria.CriteriaBuilder;
10 import javax.persistence.criteria.CriteriaQuery;
11 import javax.persistence.criteria.Root;
12
13
14 @ApplicationScoped
15 public class AttendeeManager {
16
17     @PersistenceContext(unitName = "Reg")
18     EntityManager em;
19
20     public List<Attendee> getAllAttendees() {
21         CriteriaBuilder cb = em.getCriteriaBuilder();
22         CriteriaQuery<Attendee> cq = cb.createQuery(Attendee.class);
23         Root<Attendee> rootEntry = cq.from(Attendee.class);
24         CriteriaQuery<Attendee> all = cq.select(rootEntry);
25         TypedQuery<Attendee> allQuery = em.createQuery(all);
26         return allQuery.getResultList();
27     }
28
29     public void addAttendee(Attendee attendee){
30         em.persist(attendee);
31     }
32
33 }
34
```

8. Enter another attendee using the web interface, you should see the following error message in the console.

```
[WARNING ] Application {http://reg.wasdev.net/}Attendees has thrown exception, unwinding now
No active transaction for PuId=RegistrationAppWeb#RegistrationAppWeb.war#RegistrationAppWeb
[WARNING ] Exception in handleFault on interceptor org.apache.cxf.jaxrs.interceptor.JAXRSDefaultFaultOutInterceptor@520b345c
No active transaction for PuId=RegistrationAppWeb#RegistrationAppWeb.war#RegistrationAppWeb
[ERROR ] Error occurred during error handling, give up!
No active transaction for PuId=RegistrationAppWeb#RegistrationAppWeb.war#RegistrationAppWeb
```

We see this error because the methods that are querying the database are expected to be transactional.

9. Add the following annotation to the **getAllAttendees** and **addAttendee** methods:
`@Transactional`

10. To resolve the import errors, use the **control-shift-o** keyboard shortcut. Use **control-s** to save the class.

11. The **AttendeeManager** class should resemble the image below.

```
RegApp.java Attendees.java Attendee.java AttendeeManager.java
1 package net.wasdev.reg;
2
3 import java.util.List;
4
5 import javax.enterprise.context.ApplicationScoped;
6 import javax.persistence.EntityManager;
7 import javax.persistence.PersistenceContext;
8 import javax.persistence.TypedQuery;
9 import javax.persistence.criteria.CriteriaBuilder;
10 import javax.persistence.criteria.CriteriaQuery;
11 import javax.persistence.criteria.Root;
12 import javax.transaction.Transactional;
13
14
15 @ApplicationScoped
16 public class AttendeeManager {
17
18     @PersistenceContext(unitName = "RegistrationAppWeb")
19     EntityManager em;
20
21     @Transactional
22     public List<Attendee> getAllAttendees() {
23         CriteriaBuilder cb = em.getCriteriaBuilder();
24         CriteriaQuery<Attendee> cq = cb.createQuery(Attendee.class);
25         Root<Attendee> rootEntry = cq.from(Attendee.class);
26         CriteriaQuery<Attendee> all = cq.select(rootEntry);
27         TypedQuery<Attendee> allQuery = em.createQuery(all);
28         return allQuery.getResultList();
29     }
30
31     @Transactional
32     public void addAttendee(Attendee attendee){
33         em.persist(attendee);
34     }
35
36 }
37
```

Now if we restart the server and register another attendee, we can see that attendees registered before the server restart have remained and the data has persisted beyond the scope of the application.

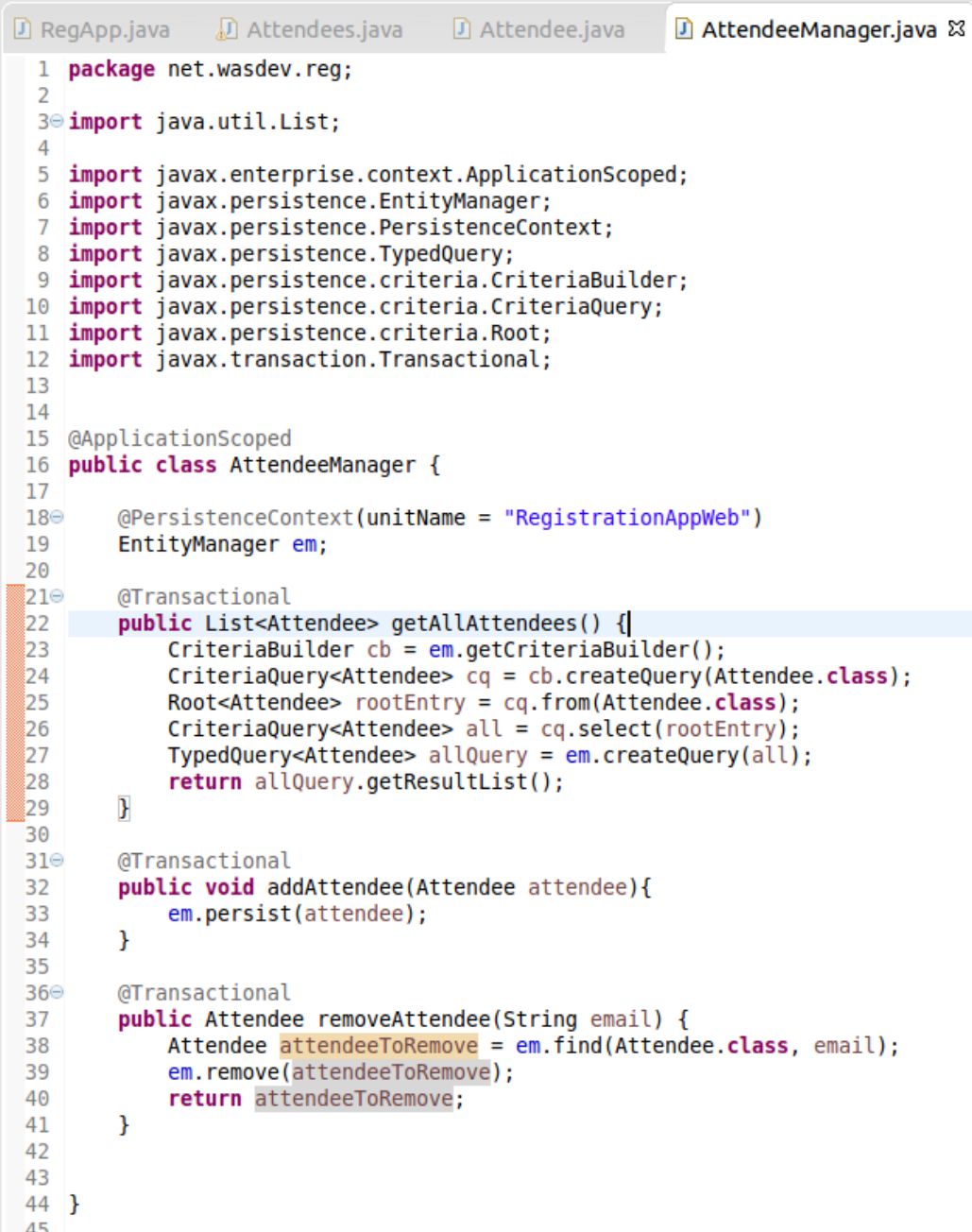
Adding more functionality

We have now created an application that can register an attendee. The client makes a REST API POST request which is handled by our JAX-RS classes, which then calls a CDI bean that handles the business logic, and finally JPA is used to store our attendee in a database. We are still missing some functionality in our application, we want the ability to see the attendees displayed without having to perform a POST request and the ability to remove an attendee. This functionality can be implemented by using GET and DELETE requests. The client that is implemented using the Angular code provided is expecting a GET request to return a list of attendees to display, and is expecting a DELETE request to remove a particular attendee from the list. We will now implement these GET and DELETE requests in our application.

First we will add a method to our CDI bean which handles removing a user from the database.

1. Open the **AttendeeManager** class.
2. Add the following code to the class body:

```
@Transactional
public Attendee removeAttendee(String email) {
    Attendee attendeeToRemove = em.find(Attendee.class, email);
    em.remove(attendeeToRemove);
    return attendeeToRemove;
}
```
3. Use **control-s** to save the class.
4. The **AttendeeManager** class should resemble the image below.



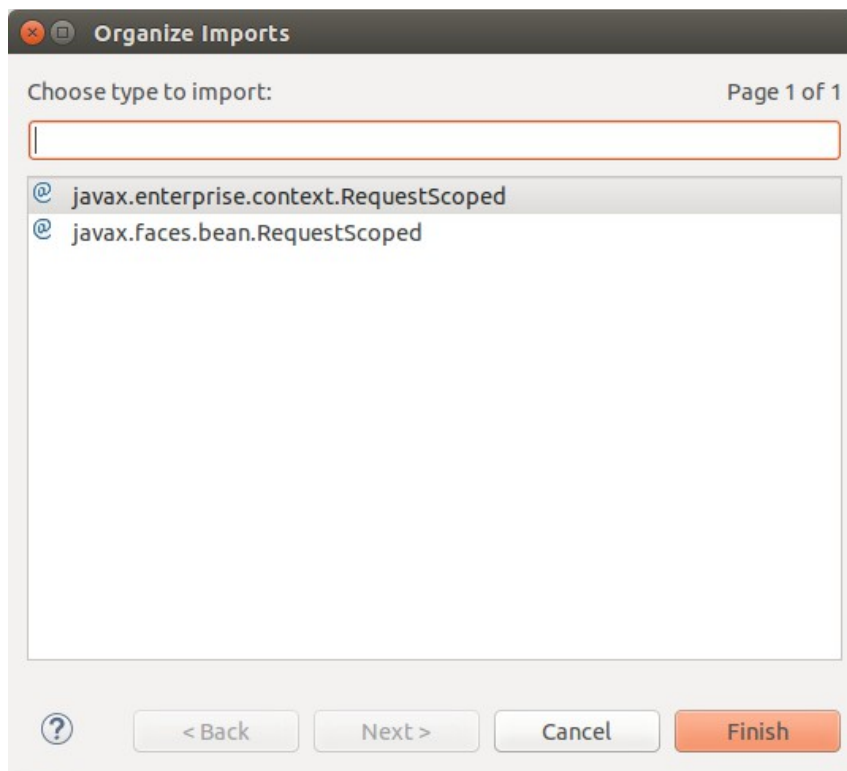
```
RegApp.java Attendees.java Attendee.java AttendeeManager.java ✖
1 package net.wasdev.reg;
2
3 import java.util.List;
4
5 import javax.enterprise.context.ApplicationScoped;
6 import javax.persistence.EntityManager;
7 import javax.persistence.PersistenceContext;
8 import javax.persistence.TypedQuery;
9 import javax.persistence.criteria.CriteriaBuilder;
10 import javax.persistence.criteria.CriteriaQuery;
11 import javax.persistence.criteria.Root;
12 import javax.transaction.Transactional;
13
14
15 @ApplicationScoped
16 public class AttendeeManager {
17
18     @PersistenceContext(unitName = "RegistrationAppWeb")
19     EntityManager em;
20
21     @Transactional
22     public List<Attendee> getAllAttendees() {
23         CriteriaBuilder cb = em.getCriteriaBuilder();
24         CriteriaQuery<Attendee> cq = cb.createQuery(Attendee.class);
25         Root<Attendee> rootEntry = cq.from(Attendee.class);
26         CriteriaQuery<Attendee> all = cq.select(rootEntry);
27         TypedQuery<Attendee> allQuery = em.createQuery(all);
28         return allQuery.getResultList();
29     }
30
31     @Transactional
32     public void addAttendee(Attendee attendee){
33         em.persist(attendee);
34     }
35
36     @Transactional
37     public Attendee removeAttendee(String email) {
38         Attendee attendeeToRemove = em.find(Attendee.class, email);
39         em.remove(attendeeToRemove);
40         return attendeeToRemove;
41     }
42
43
44 }
45
```

We will now add GET and DELETE request handlers to our JAX-RS resource class.

1. Open the **Attendees** class.
2. To make our JAX-RS class handle GET requests, add the following code to the class body:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Attendee> getAttendees() {
    return attendeeManager.getAllAttendees();
}
```
3. To make our JAX-RS class handle DELETE requests, add the following code to the class body:

```
@DELETE
@Produces(MediaType.APPLICATION_JSON)
@Path("/{email}")
public List<Attendee> removeAttendee(@PathParam("email") String email) {
    attendeeManager.removeAttendee(email);
    return attendeeManager.getAllAttendees();
}
```
4. To resolve the import errors, use the **control-shift-o** keyboard shortcut.
5. For **PathParam**, select **javax.ws.rs.PathParam**. Click **Finish**. Use **control-s** to save the class.



6. The **Attendees** class should resemble the image below.

```
RegApp.java Attendees.java Attendee.java AttendeeManager.java pe
1 package net.wasdev.reg;
2
3 import java.util.List;
4
5 import javax.enterprise.context.RequestScoped;
6 import javax.inject.Inject;
7 import javax.ws.rs.Consumes;
8 import javax.ws.rs.DELETE;
9 import javax.ws.rs.GET;
10 import javax.ws.rs.POST;
11 import javax.ws.rs.Path;
12 import javax.ws.rs.PathParam;
13 import javax.ws.rs.Produces;
14 import javax.ws.rs.core.MediaType;
15
16 @RequestScoped
17 @Path("/attendees")
18 public class Attendees {
19
20     @Inject
21     AttendeeManager attendeeManager;
22
23     @GET
24     @Produces(MediaType.APPLICATION_JSON)
25     public List<Attendee> getAttendees() {
26         return attendeeManager.getAllAttendees();
27     }
28
29     @DELETE
30     @Produces(MediaType.APPLICATION_JSON)
31     @Path("{email}")
32     public List<Attendee> removeAttendee(@PathParam("email") String email) {
33         attendeeManager.removeAttendee(email);
34         return attendeeManager.getAllAttendees();
35     }
36
37     @POST
38     @Consumes(MediaType.APPLICATION_JSON)
39     @Produces(MediaType.APPLICATION_JSON)
40     public List<Attendee> addAttendee(Attendee attendee){
41         attendeeManager.addAttendee(attendee);
42         return attendeeManager.getAllAttendees();
43     }
44
45 }
46
```

Now if you refresh the browser page, we can see a list of attendees displayed. If we click the remove button, that attendee will be removed from the list. Our application now has all of the required functionality.

Summary

In this lab you learned:

- How to set up a Liberty runtime environment
- How to create a server to run in this environment
- How to deploy a simple web application to this server
- How to build a simple web application using JAX-RS, CDI and JPA

If you are interested in learning more please visit <http://wasdev.net>. WASdev is the developer focussed community for WebSphere Application Server developers, providing:

- Useful articles on getting started
- Samples and tutorials of specific features
- Configuration snippets
- The latest releases of available Early Access Programs for Liberty and related products.
- Forums for finding further information from other developers, and getting answers to questions.