Learn to Accelerate Your Web App Development with the Liberty Profile Lab Instructions

Authors:
Alasdair Nottingham, Senior Software Engineer
Tim deBoer, Senior Technical Staff Member
Ross Pavitt, Software Engineer

Table of Contents

Objective

In this lab, you learn:

- How to install the IBM WebSphere Application Server V8.5.5 Liberty Profile.
- How to create and deploy a simple web application using the IBM WebSphere Application Server Developer Tools for Eclipse V8.5.5.
- How to create and deploy a simple registration web application that uses Servlets, JPA, EJBs, Context and Dependency Injection, and JAX-RS.
- How to secure applications to use SSL.
- How to generate a customised Liberty profile image, and use that image to deploy your application outside of a development environment.

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

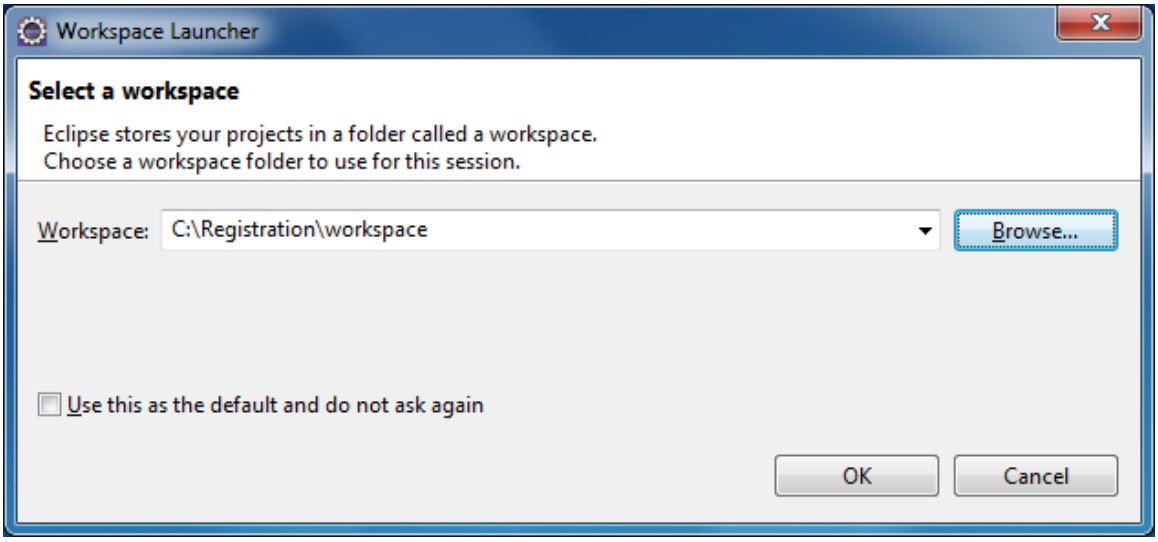
Setting up Eclipse

To run this lab you need to start Eclipse, configure a workspace and import the required resources.

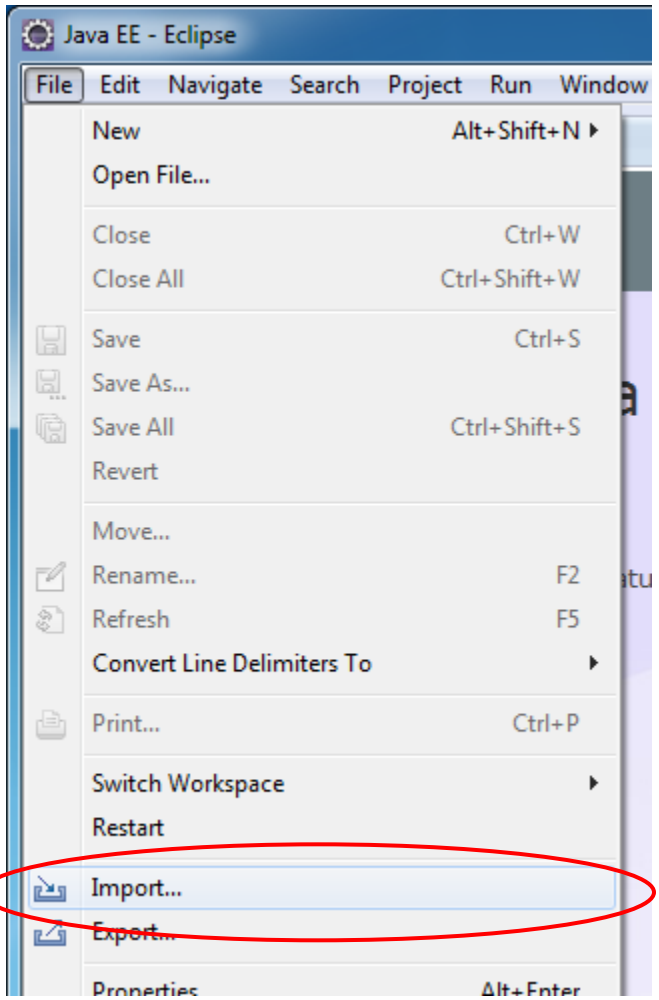
Instructions

1. Go to https://www.ibmdw.net/wasdev/websphere-application-server-developer-tools-v8-5-5/, and follow instructions to install WebSphere Application Server Developer Tool for Eclipse V8.5.5.

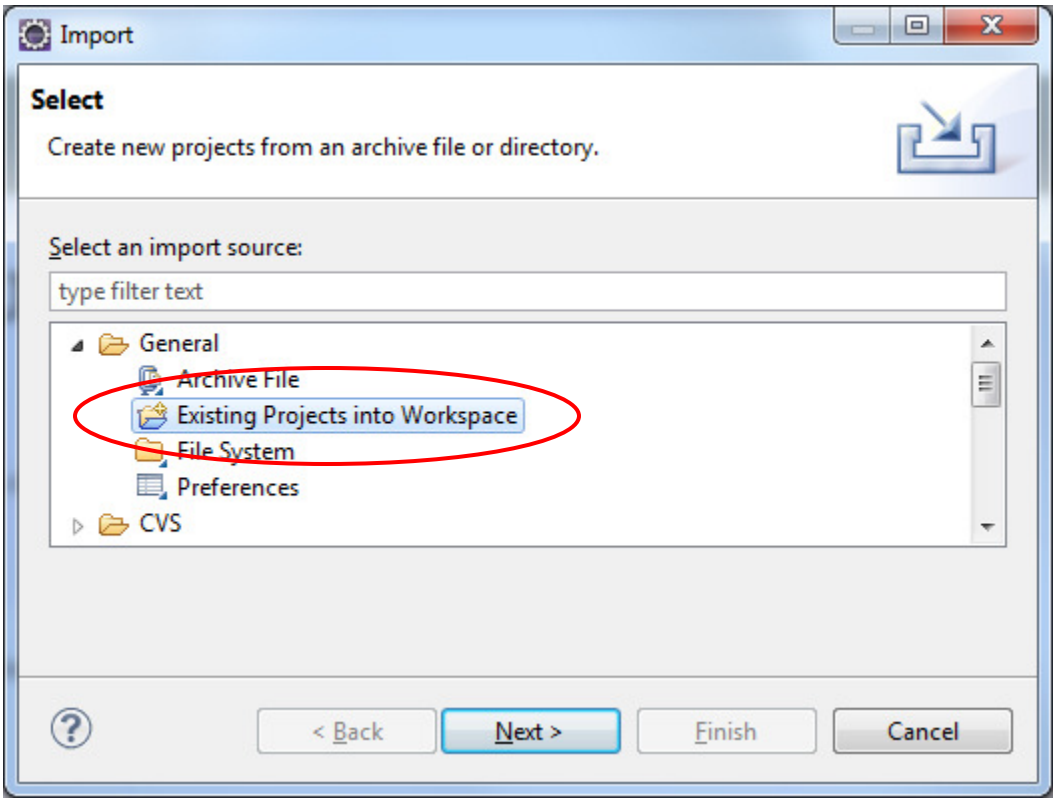
2. Launch the Eclipse IDE. If you are prompted to provide a workspace location, provide a path to an empty folder that can be used to store your work with Eclipse and click **OK**.



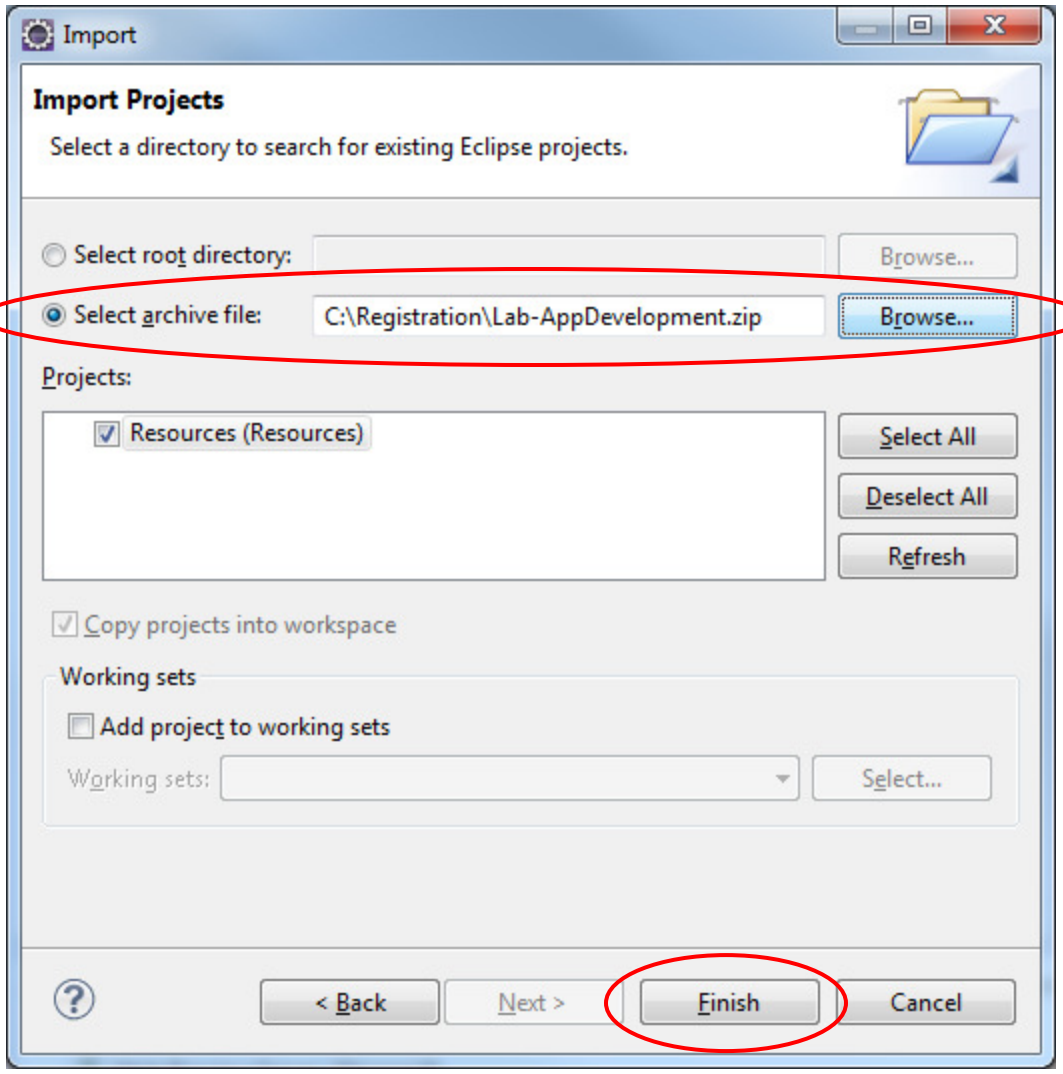
3. Select **File > Import…**



4. In the **Import** dialog, expand the **General** section, and select **Existing Projects into Workspace**, then click **Next**.



5. Select the option for **Select archive file**, and use the **Browse** button to navigate to the copy of `Lab-AppDevelopment.zip` that came with this set of instructions. Once this is done, click **Finish**.

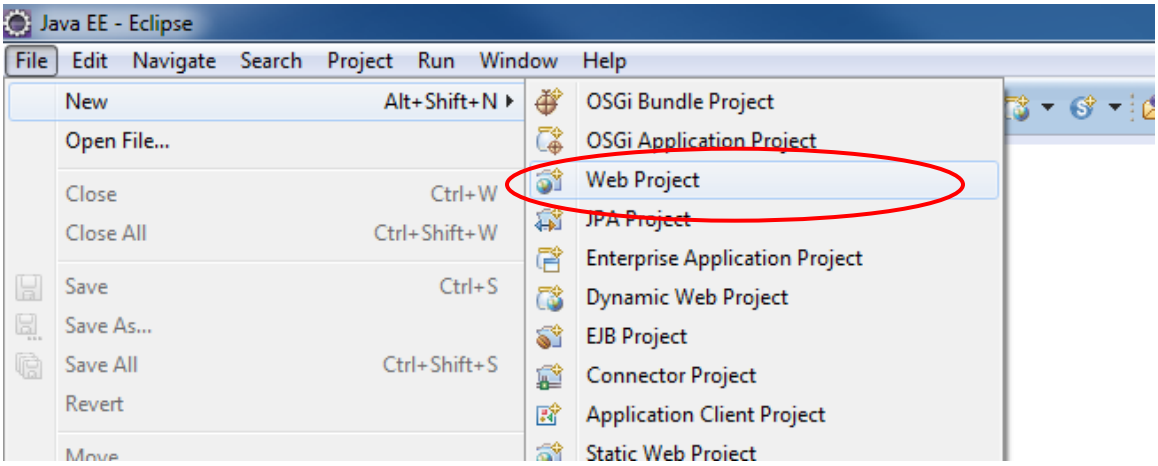


The Hello World Application

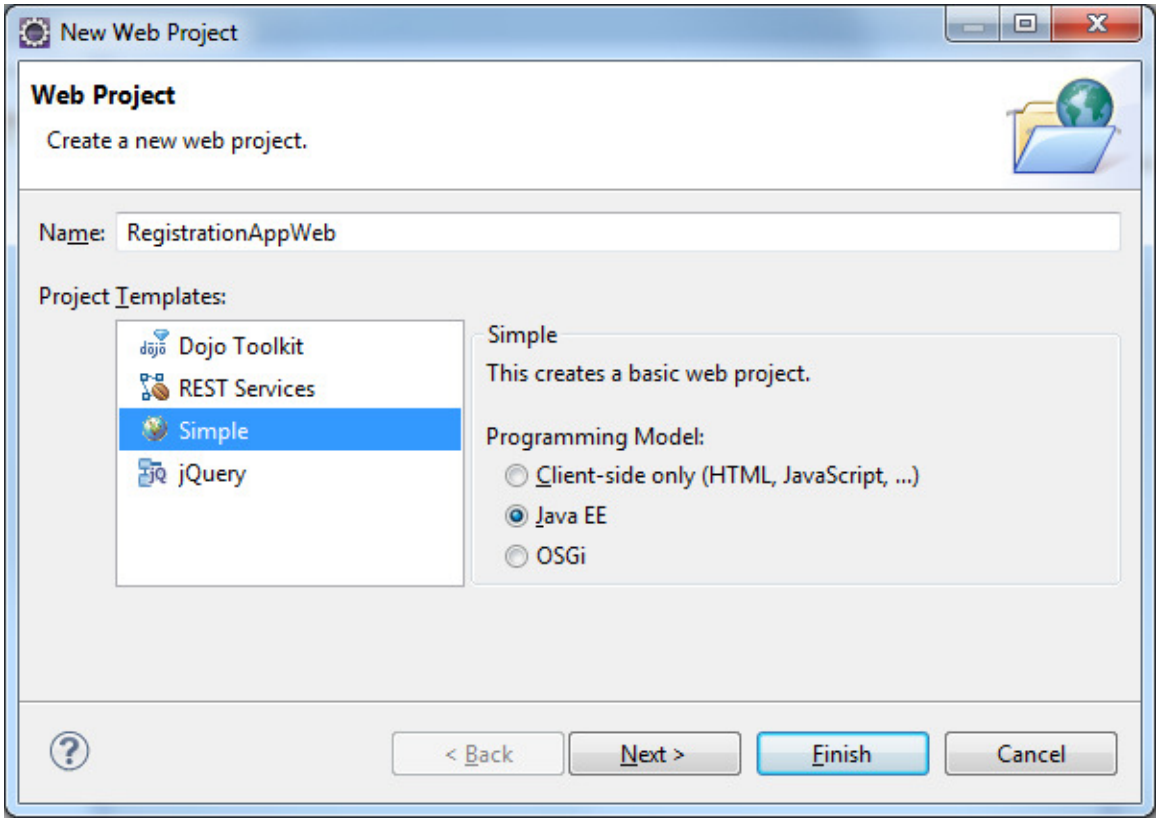
In this exercise you will install the Liberty profile, create a simple web project, write an application consisting of a single HTML page, and deploy that application to a Liberty profile server. This application forms the basis for later parts of the lab.

Instructions

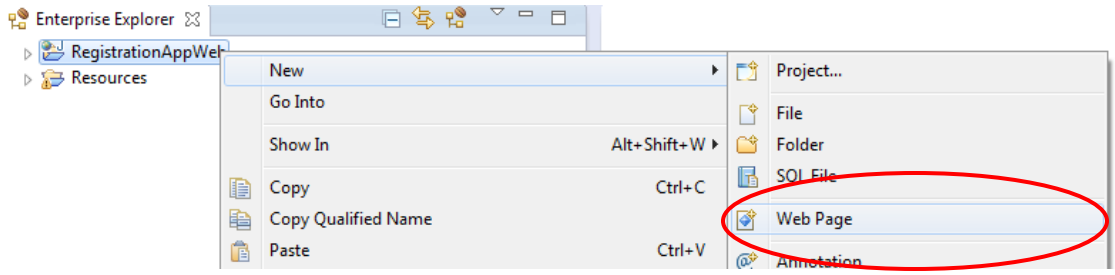
1. Create a new Web Project by clicking **File > New > Web Project**.



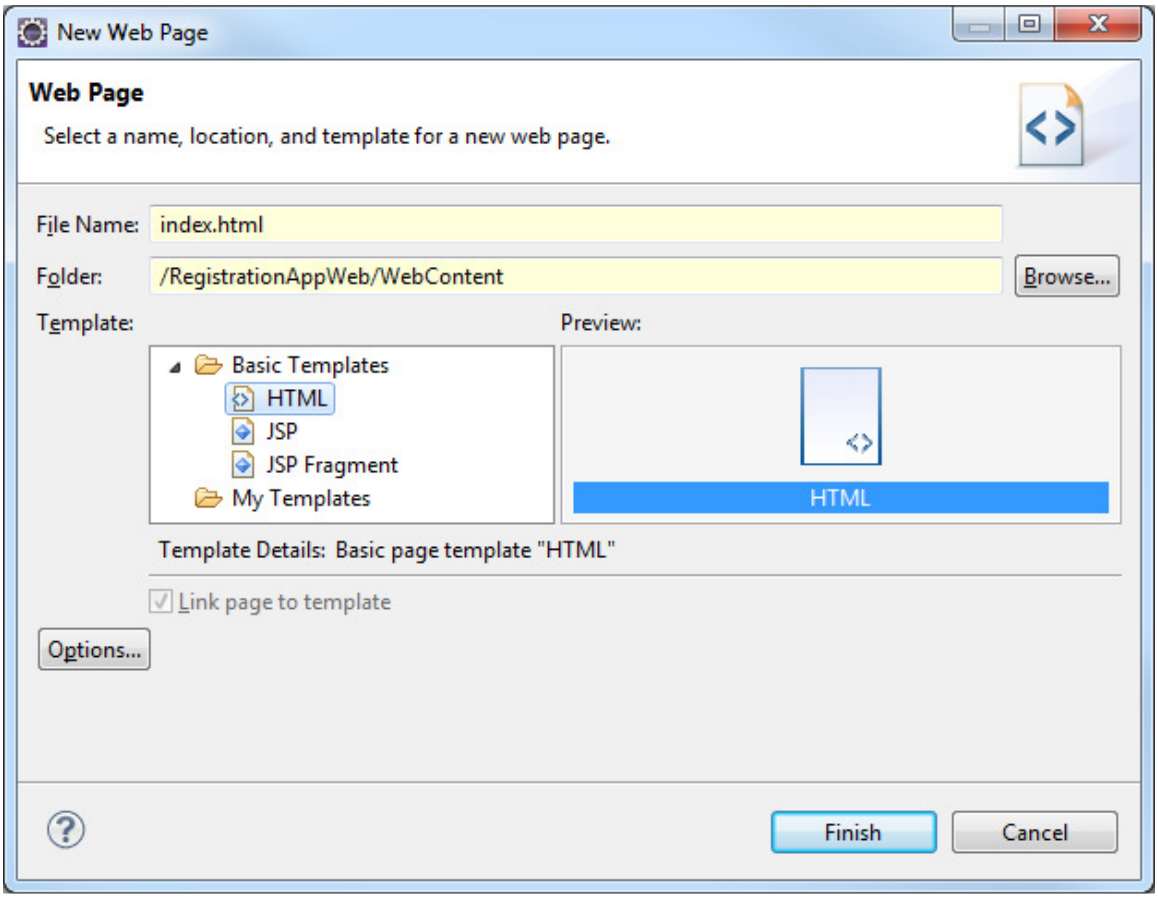
2. Enter the name of the project as **RegistrationAppWeb**. Select the *Simple* option from the **Project Templates**, and check the **Programming model** is set to *Java EE*. Click **Finish**. If prompted to open the Web perspective, click **Yes**.



3. To create the web page right click the **RegistrationAppWeb** project and select **New > Web Page**.



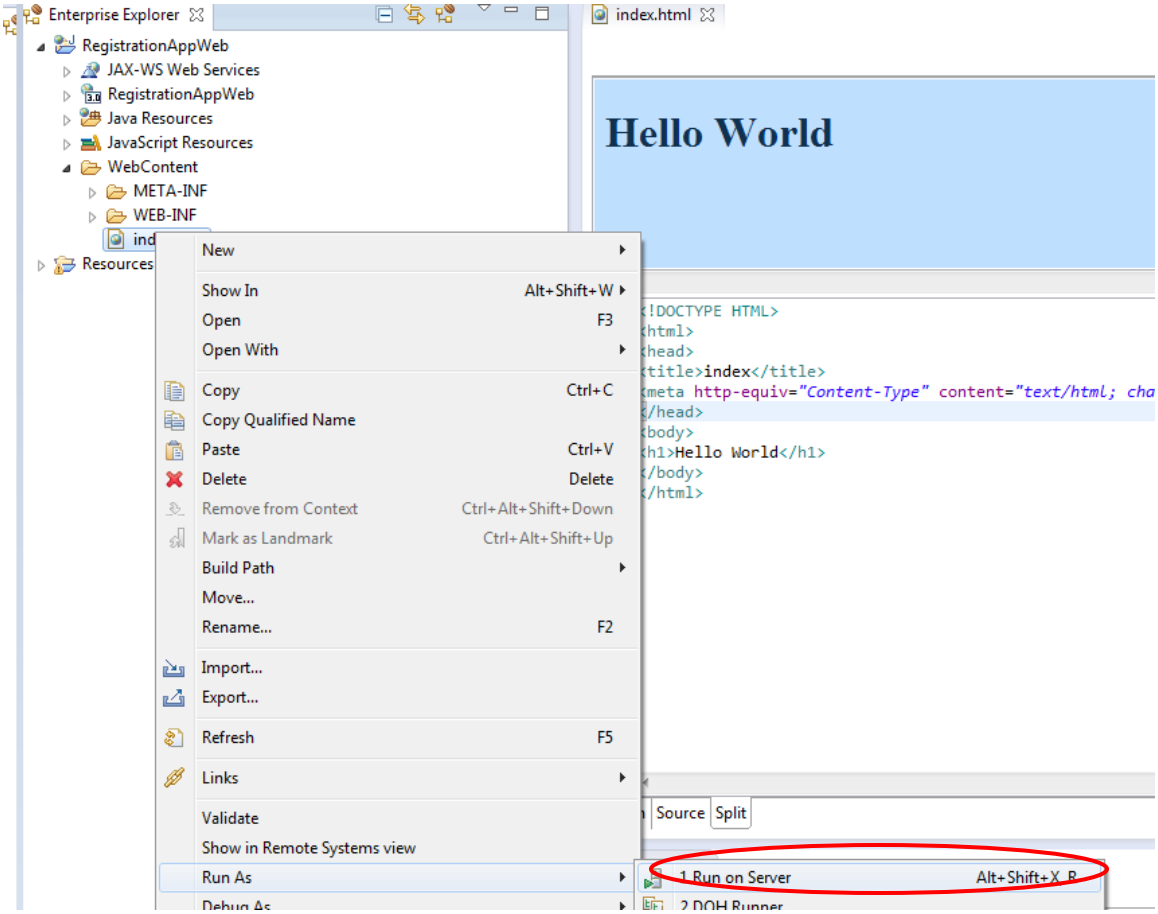
4. Set the **File Name** to *index.html*, and set the **Template** to *HTML* from the **Basic Templates**, then click **Finish**.



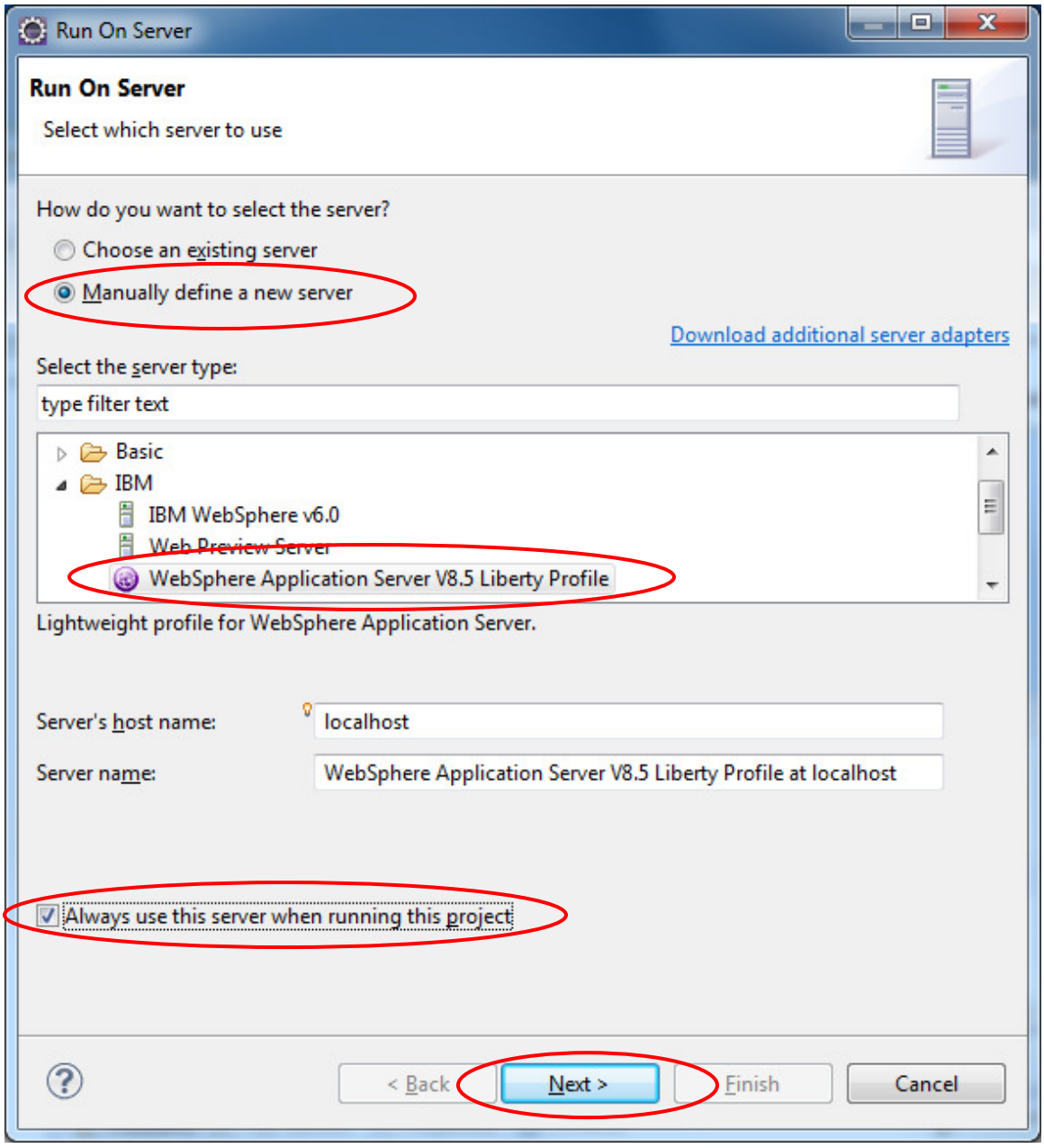
5. In the web page, add `<h1>Hello World</h1>` between the opening and closing tags of the `body` element.

6. Save the file using the **control-S** shortcut.

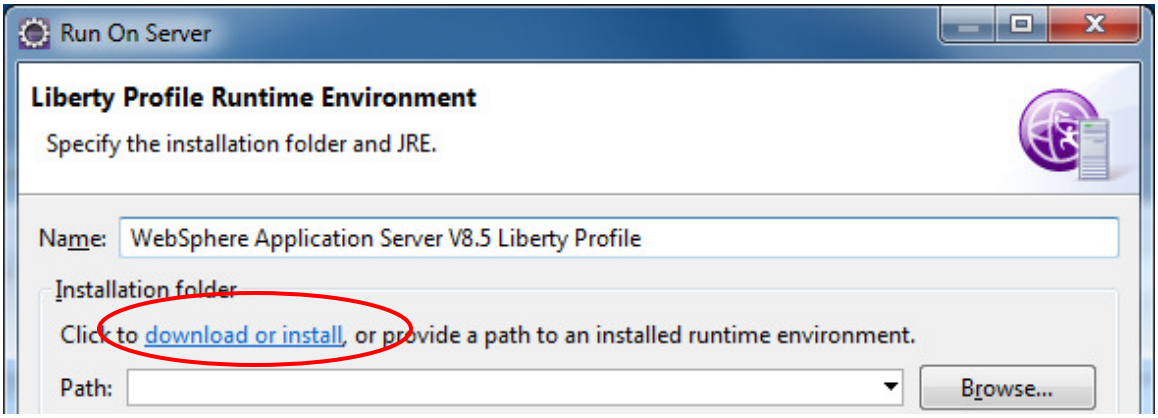
7. Now the basic application is complete. To deploy, right click on the *index.html* file and select **Run As > Run on Server**.



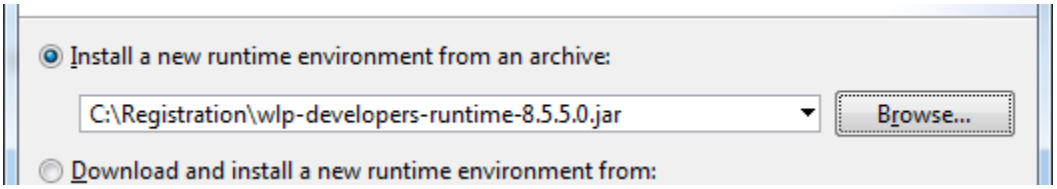
8. Select **Manually define a server**. Expand **IBM**, and select **WebSphere Application Server V8.5 Liberty Profile**. Check **Always use this server when running this project**. Click **Next**.



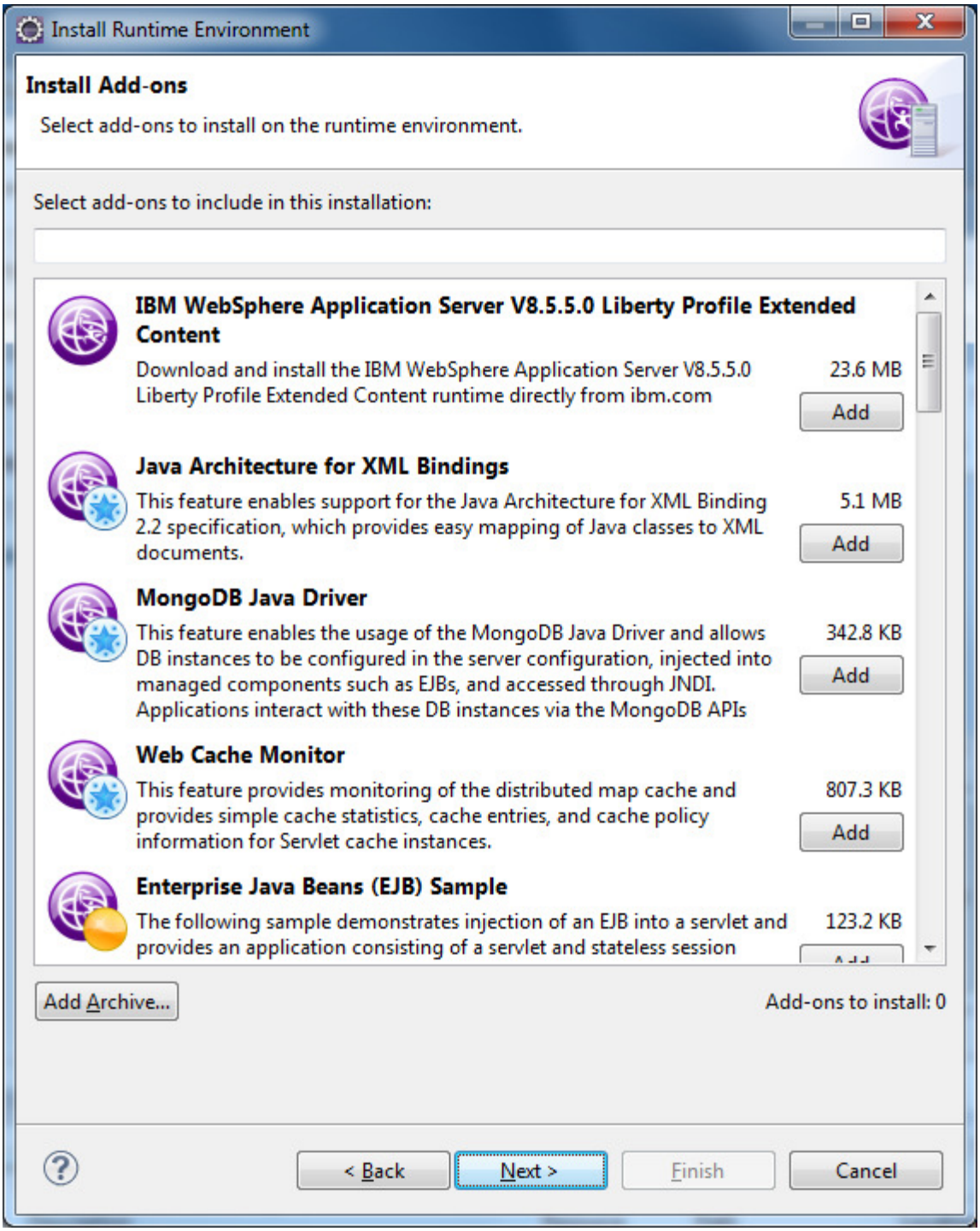
9. Click the **download or install** link.



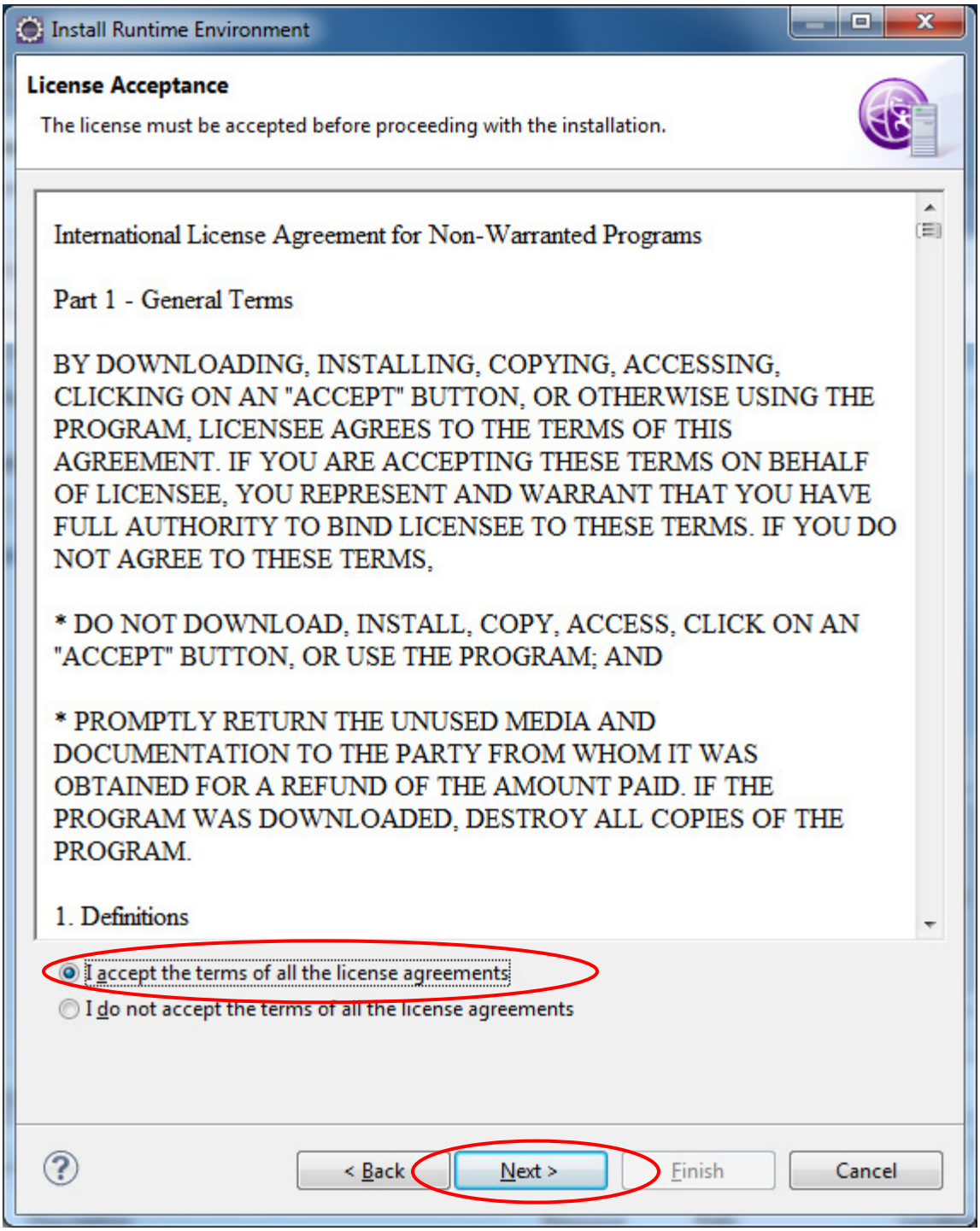
10. Select the option to **Install a new runtime from an archive**, and use the **Browse** button to select the `wlp-developers-runtime-8.5.5.0.jar` and click **Next**.



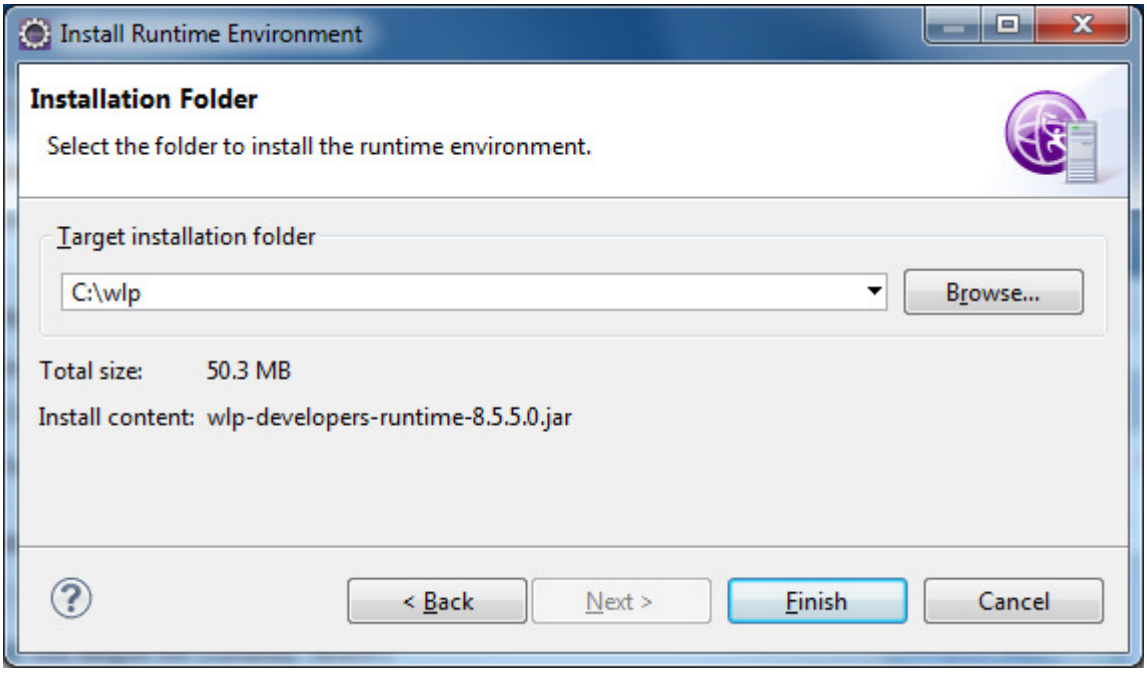
11. Click **Next** on the **Install Add-ons** screen.



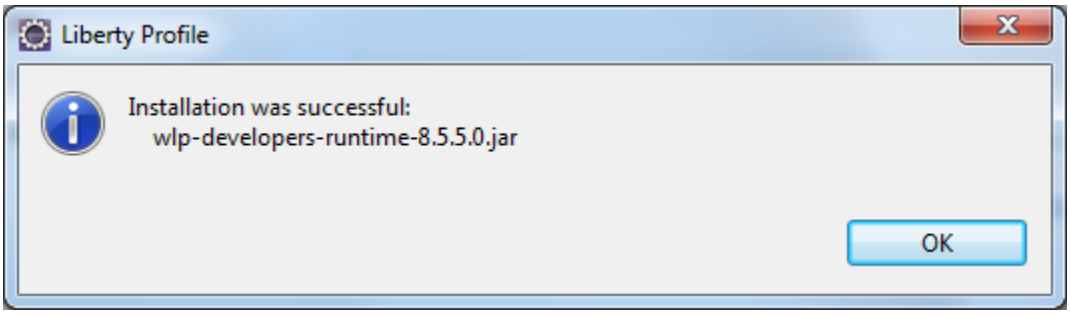
12. Select the radio button for *I accept the terms of the license agreement* and click **Next**.



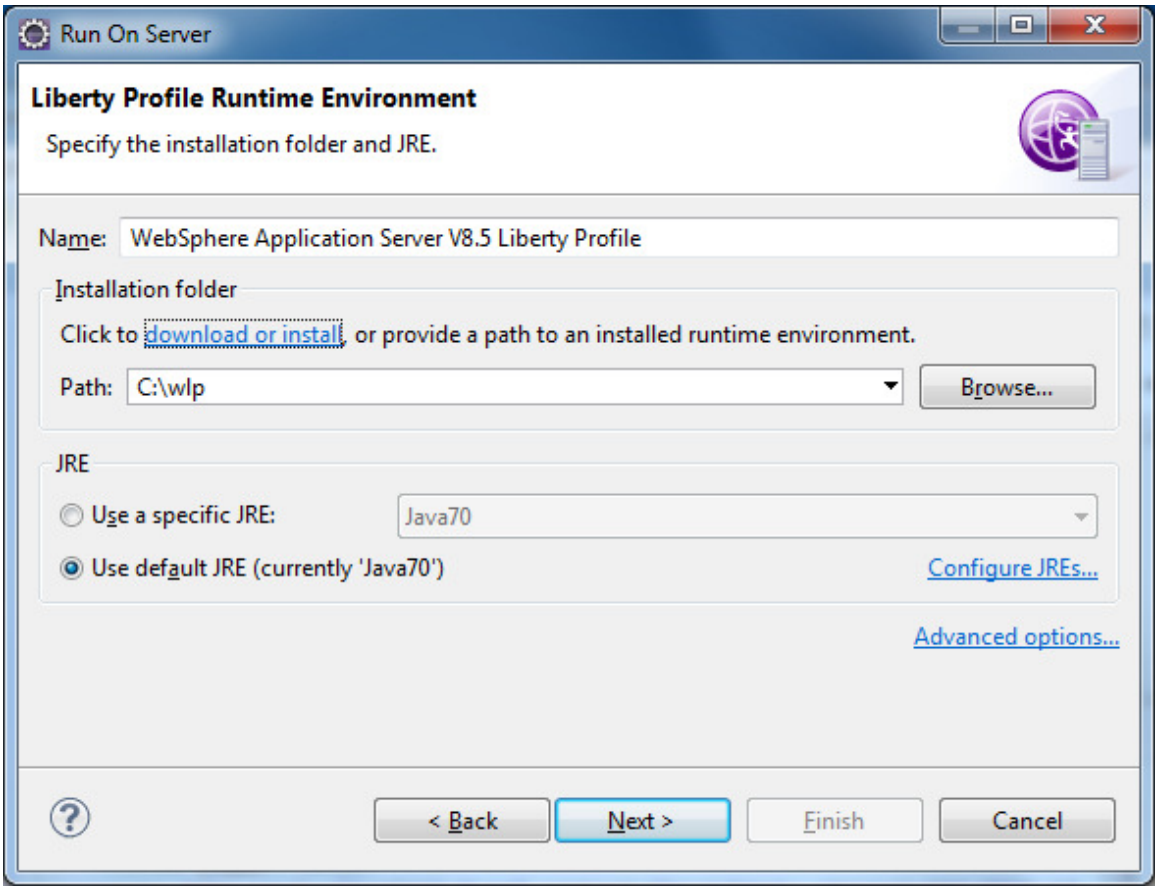
13. Set the **Target installation folder** to *C:\wlp*, and then click **Finish**.



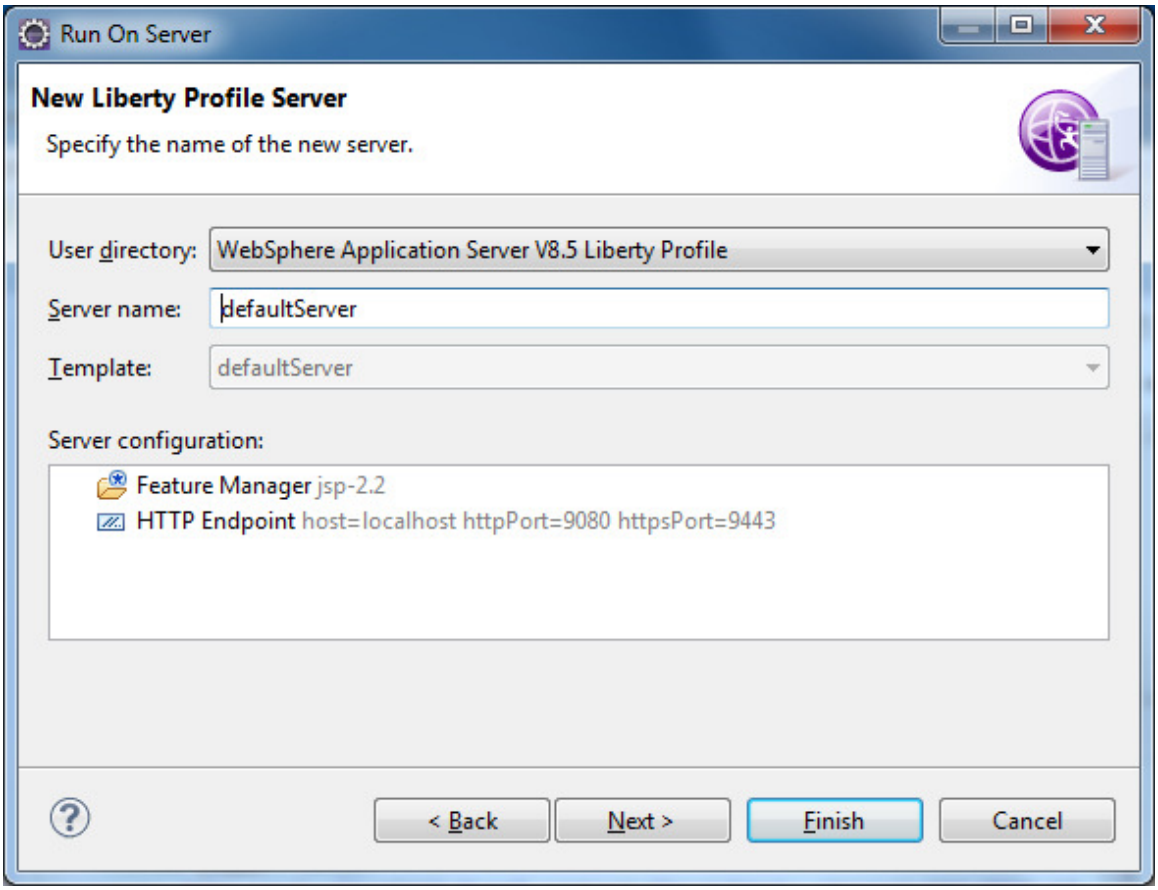
14. Click **OK** when presented with a message box stating the installation was successful.



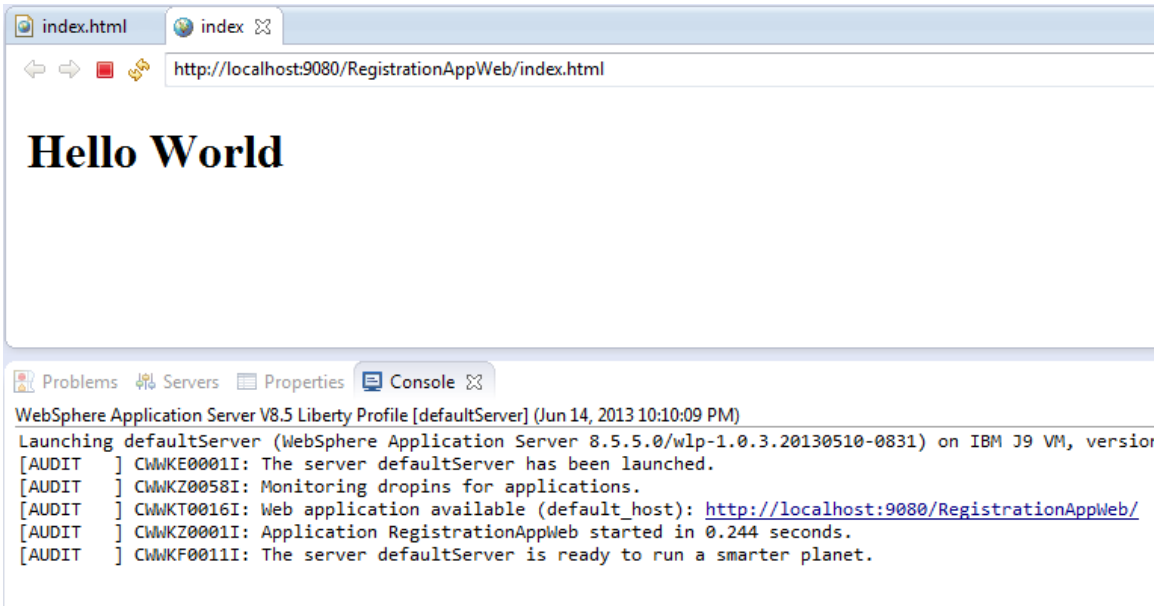
15. The next step is to define a new server. To do this click **Next**.



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



17. The server is started and *index.html* file is opened in a browser.



Summary

In this exercise you have learned how to

- Create a simple web application
- Install the Liberty profile
- Deploy an application to a server

This example illustrates how easy it is to get up and running with the Liberty profile and the WebSphere Application Server Developer Tools for Eclipse.

Registration Application

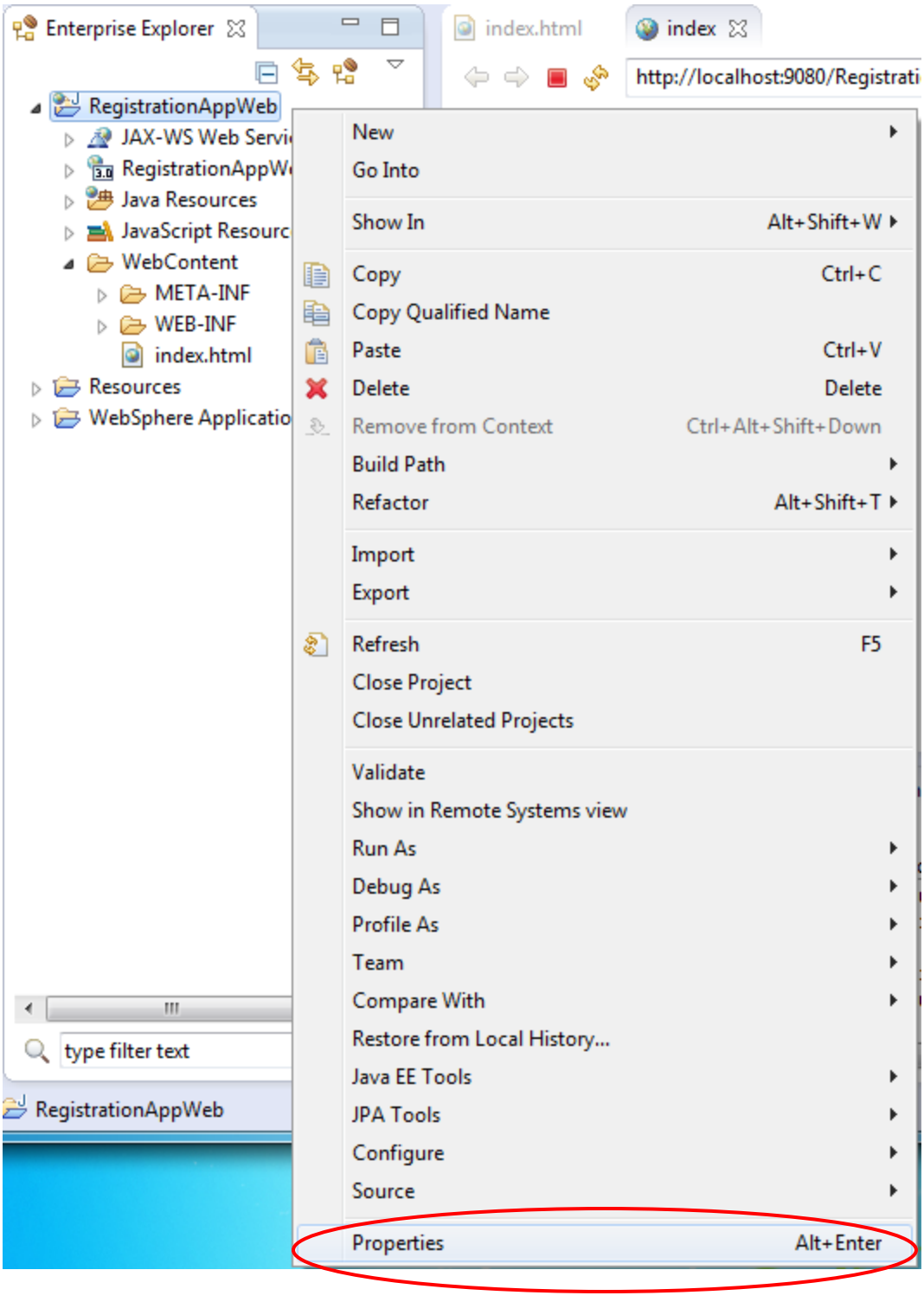
In this next exercise you create a more complex application. You will learn how to create an application that uses Enterprise Java Beans (EJBs), Context and Dependency Injection (CDI), Java Server Pages (JSP), Java Persistence API (JPA), and Servlets to implement a simple registration application. Enterprise Java Beans provide a mechanism to separate business logic from presentation logic. The first task is to create the JPA entity that will be required by the EJB, and then to create the business logic of the EJB.

Defining the JPA entity

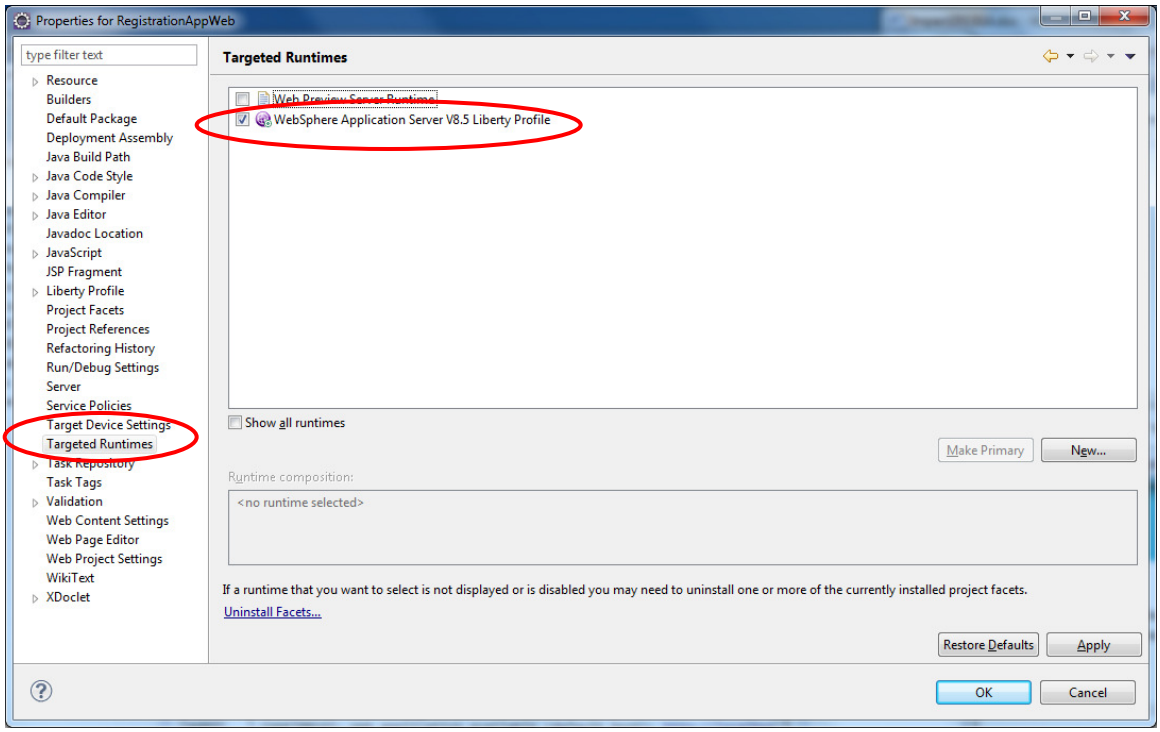
As we are using WebSphere Application Server Developer Tools for Eclipse we benefit from tools to aid the creation of JPA entities. To do this we will include the JPA project Facet in Eclipse, and then define a new JPA entity called `Person`.

.

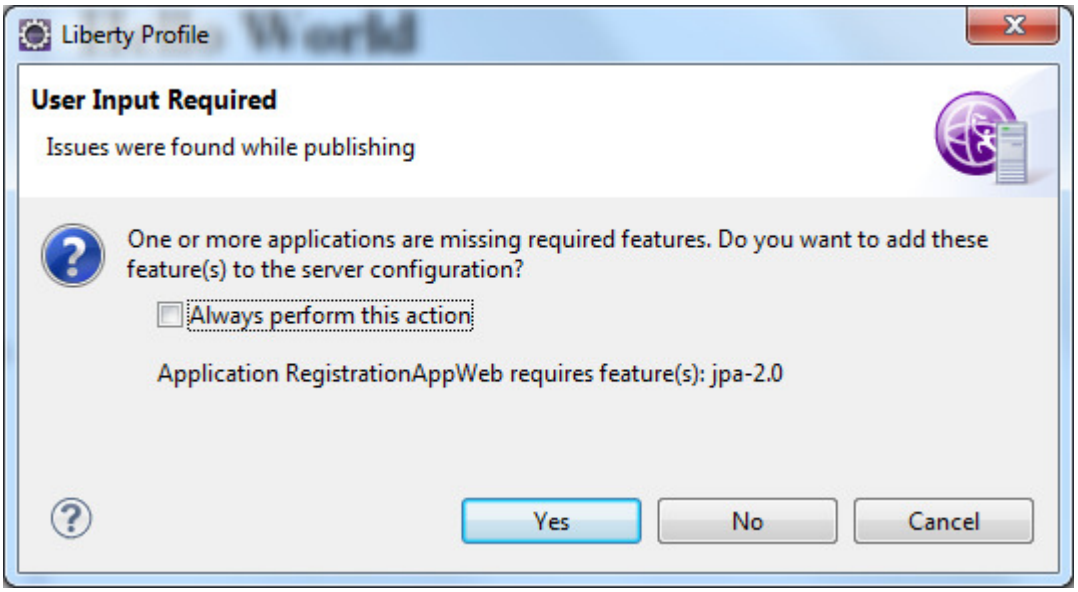
1. Right click on the RegistrationAppWeb project in Eclipse and select **Properties.**



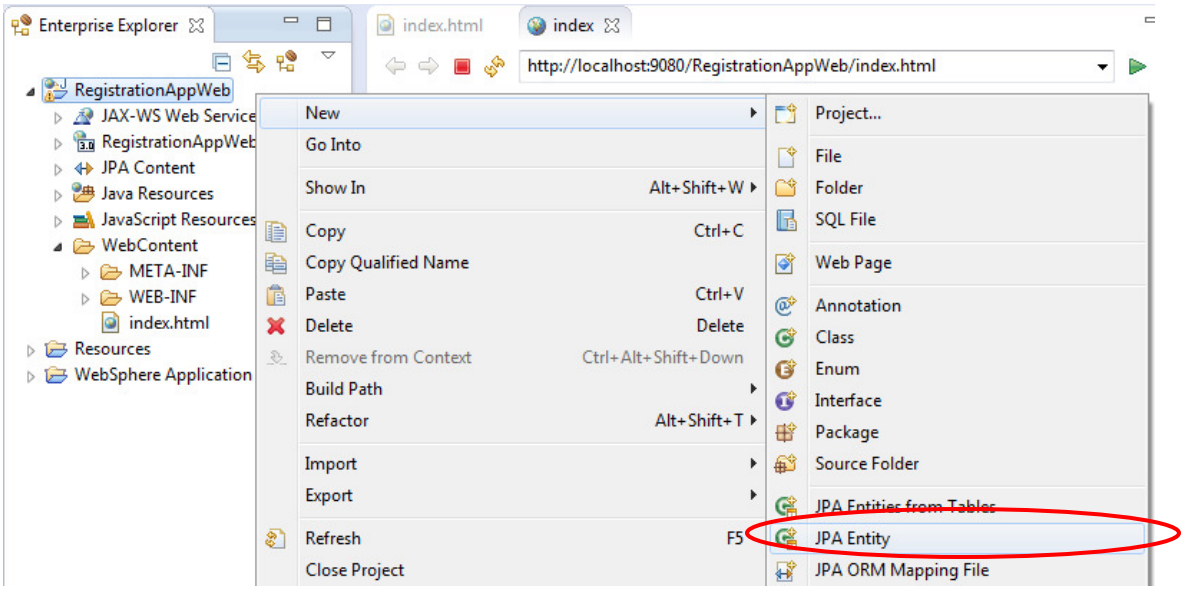
2. Navigate to the **Targeted Runtimes** section, and ensure the checkbox is selected next to **WebSphere Application Server V8.5 Liberty Profile**.



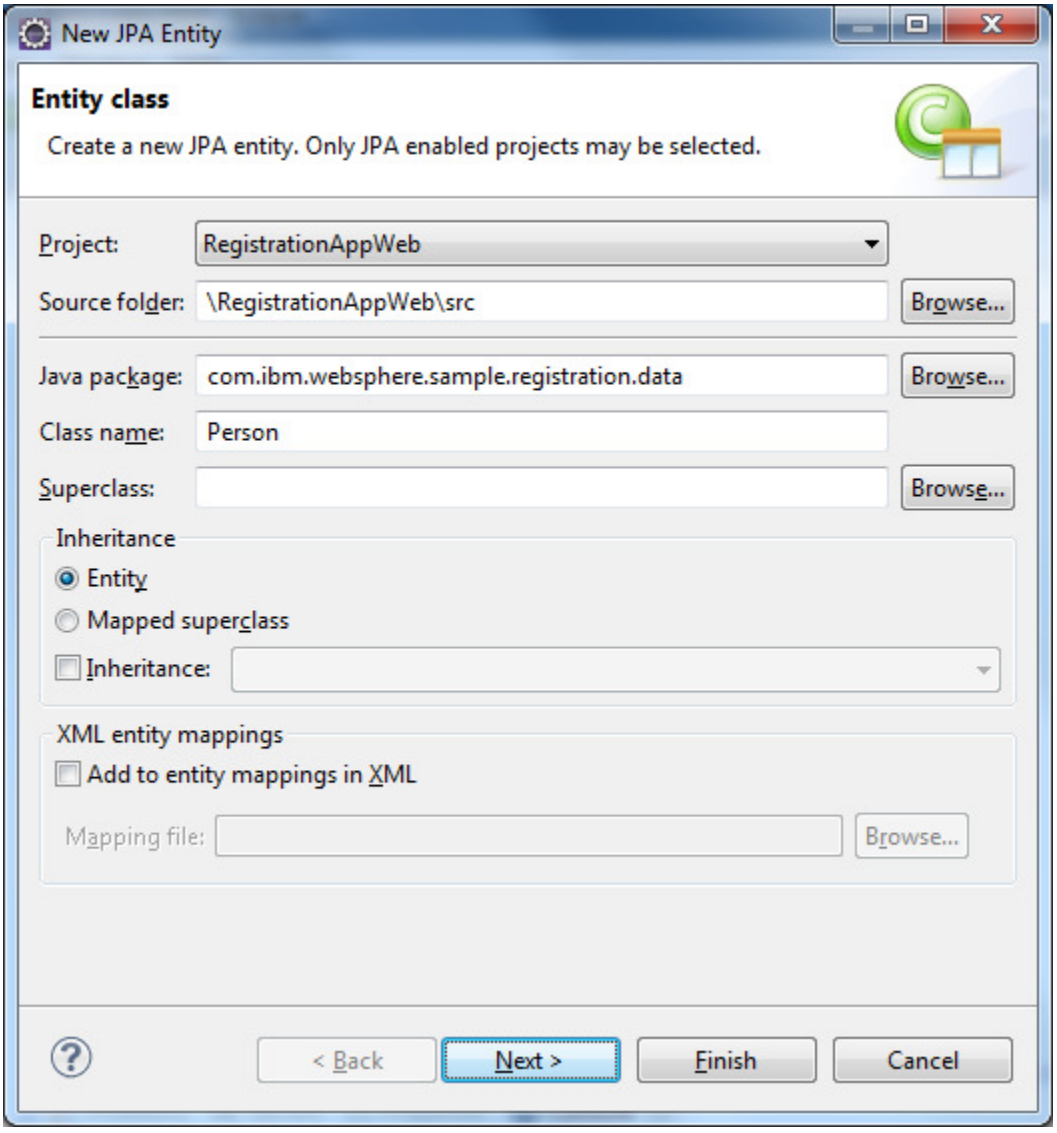
3. Next, navigate to the **Project Facets** section. Check the checkbox for **JPA**, and click **OK**. A prompt will appear asking to add the jpa-2.0 feature to the Liberty profile, select **Yes**.



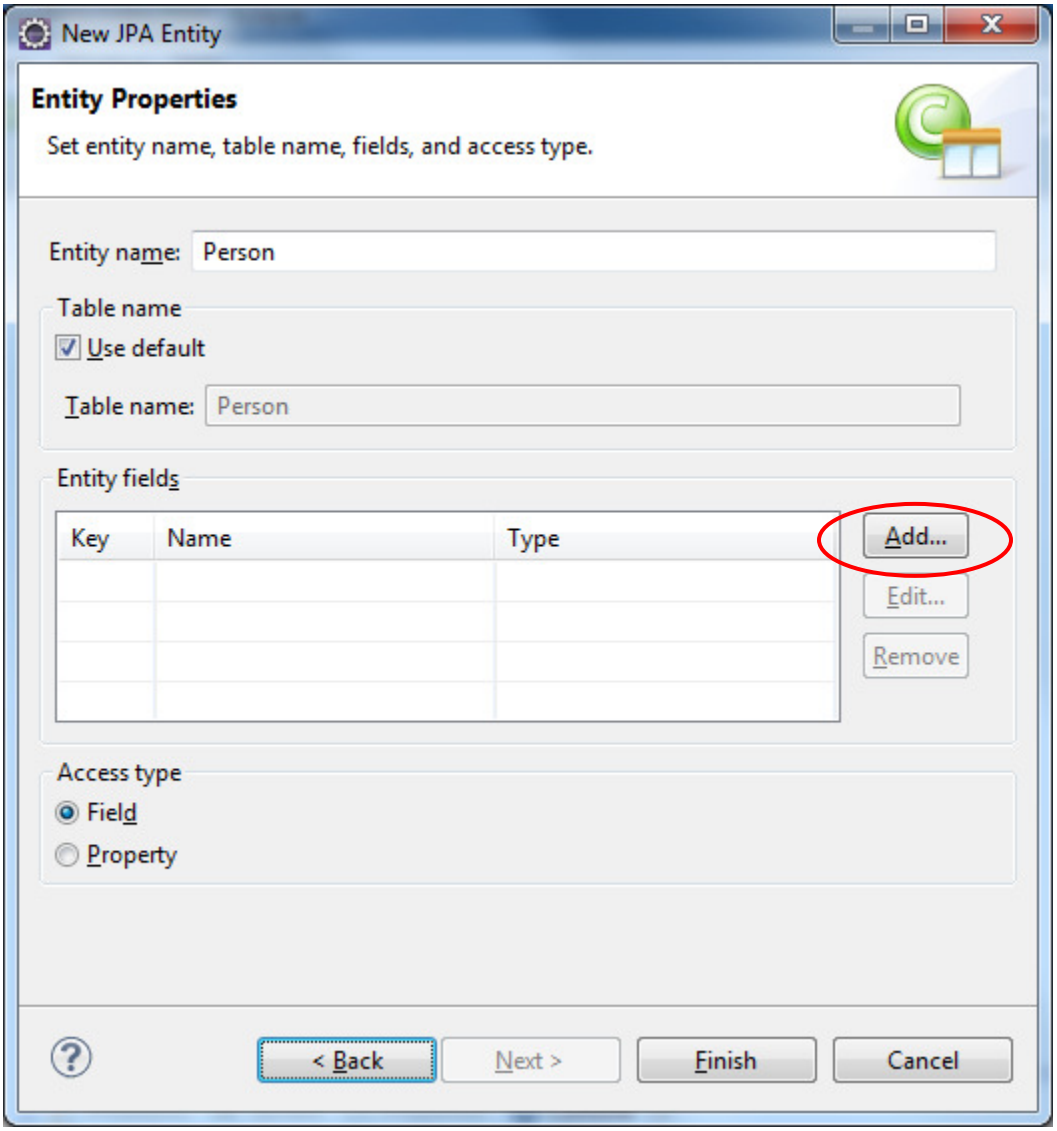
4. Right click on the **RegistrationAppWeb** project and select **New > JPA Entity**.



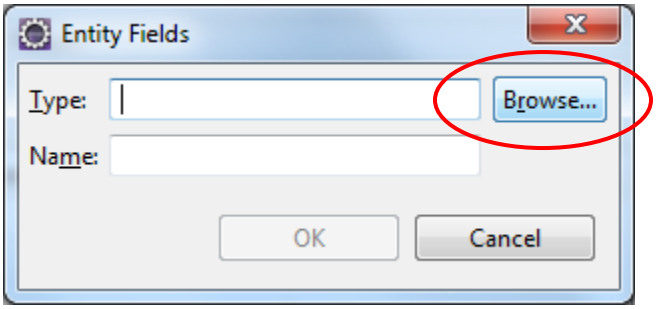
5. Set the **Java package** to *com.ibm.websphere.sample.registration.data*, and the **Class name** to *Person*, then click **Next**.



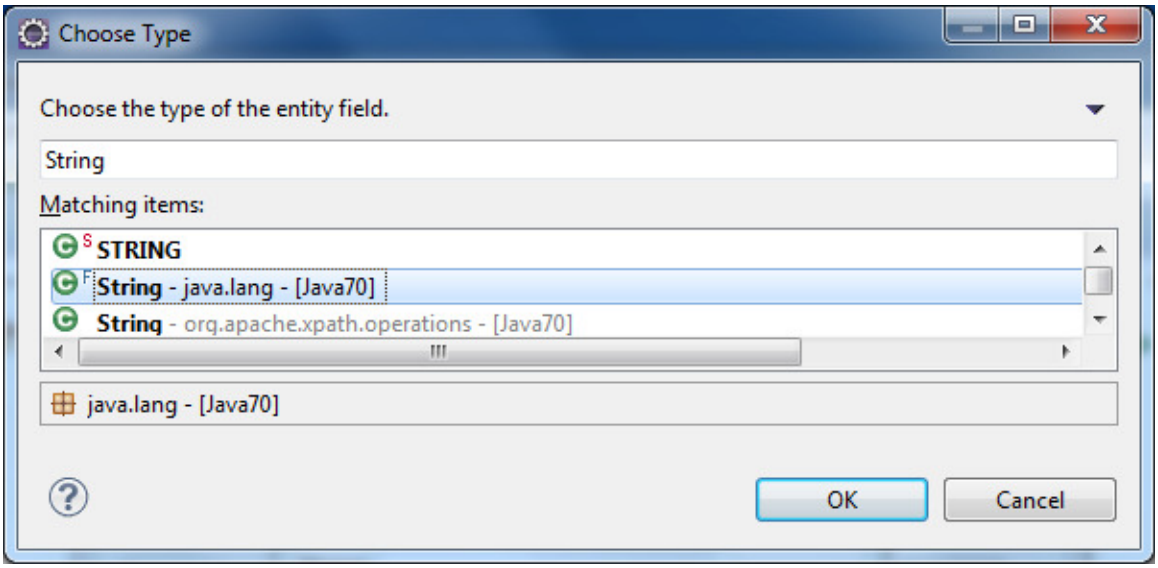
6. Click the **Add** button to add a new field to the class.



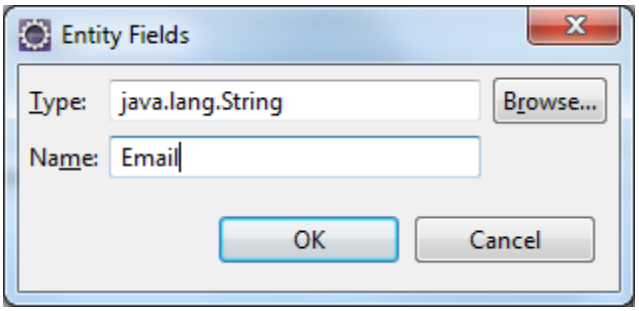
7. Click the Browse button to select a Java class type.
8.



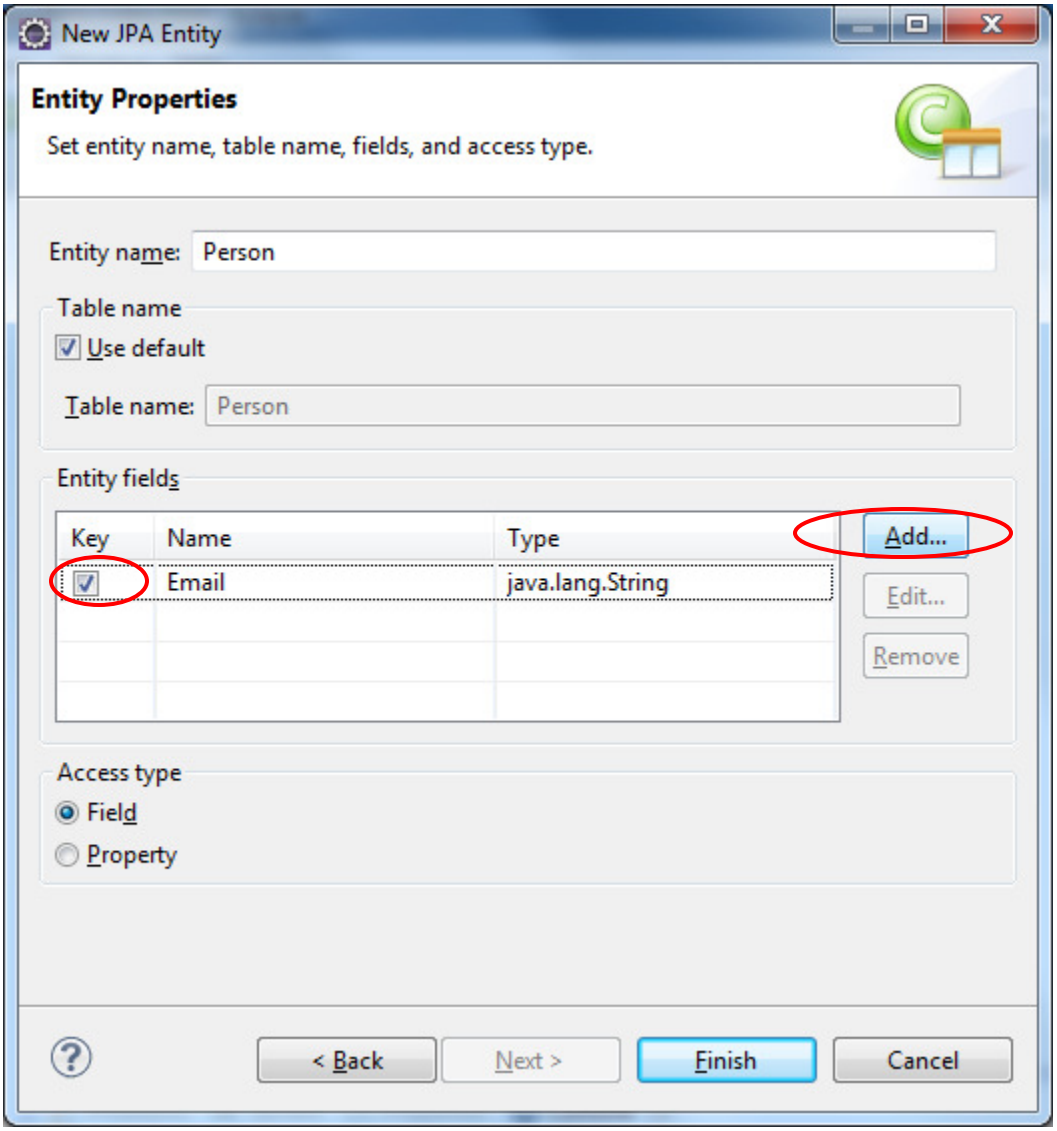
9. In the filter box enter *String*, and select the `String` from the `java.lang` package, then click **OK**.



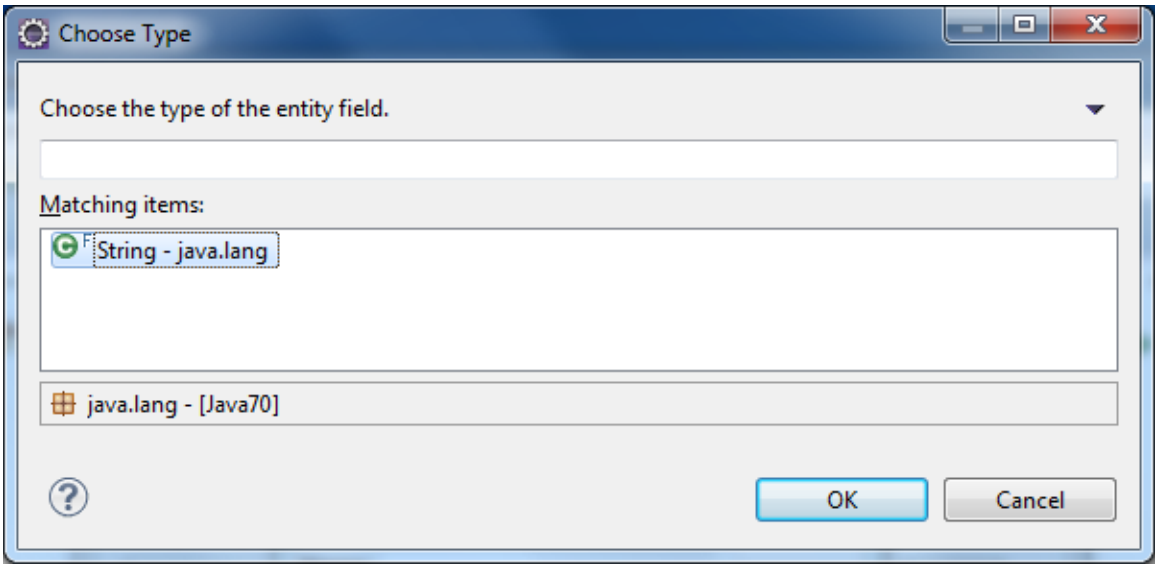
10. Set the **Name** to Email and click **OK**.



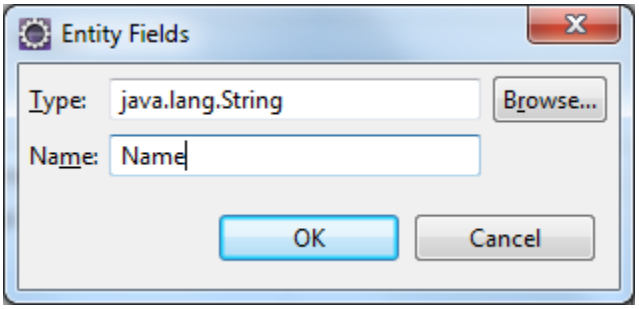
11. To set this field to a Primary Key (ensuring this field must be unique across all entities), check the checkbox under the **Key** header. Then click the **Add…** button to add a new field.



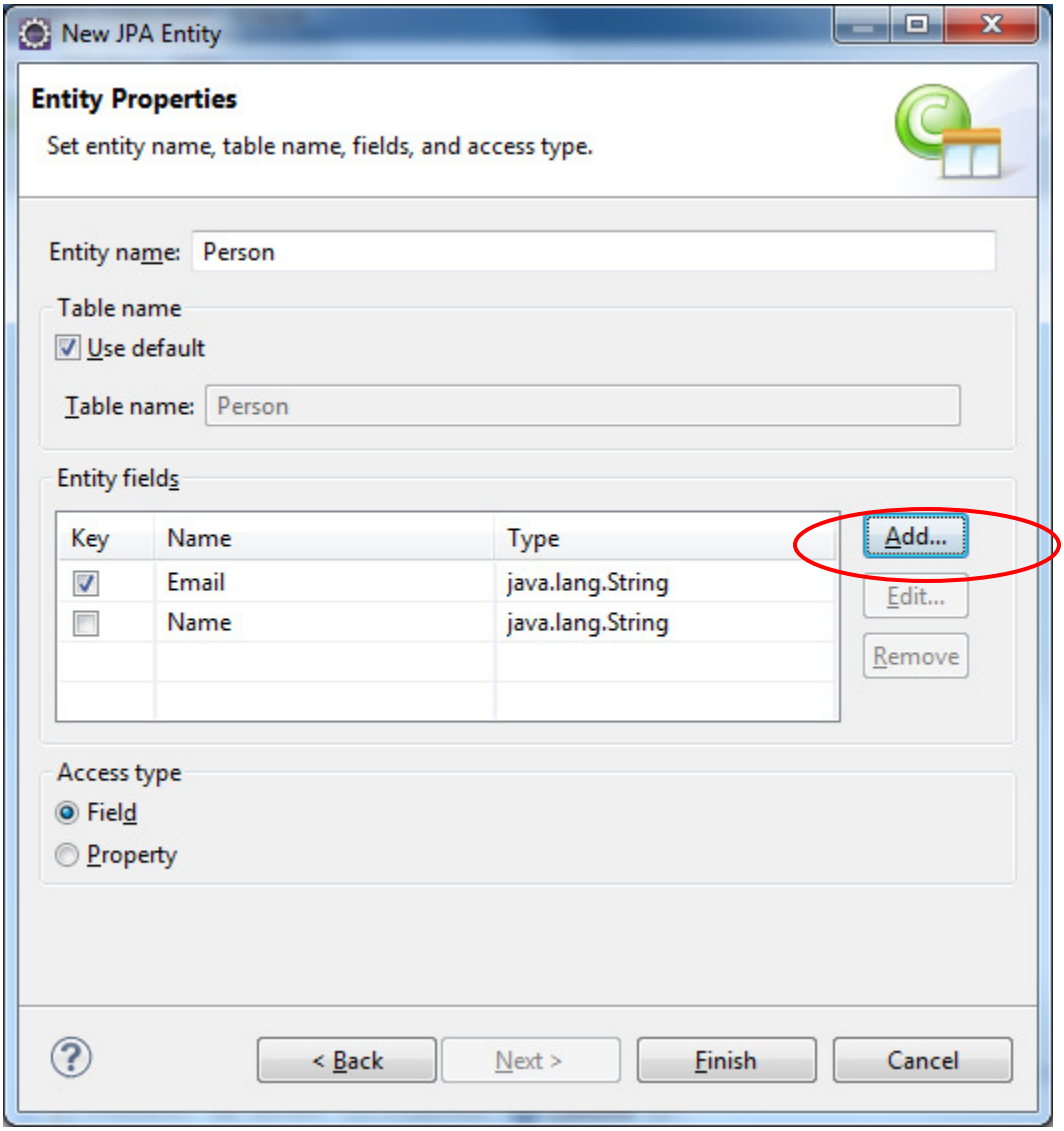
12. Click the **Browse** button again, and select the `String` class from the `java.lang` package again, which should be stored in your history of **Matching Items**.



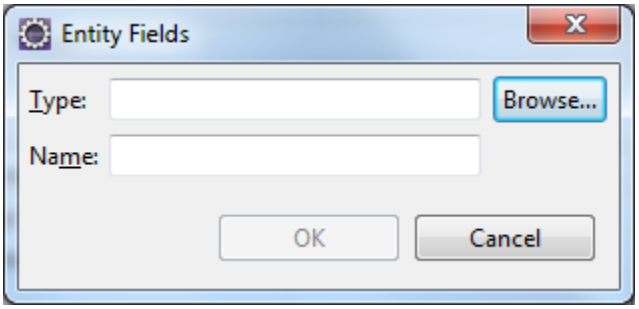
13. Enter a **Name** of *Name*, then click **OK**.



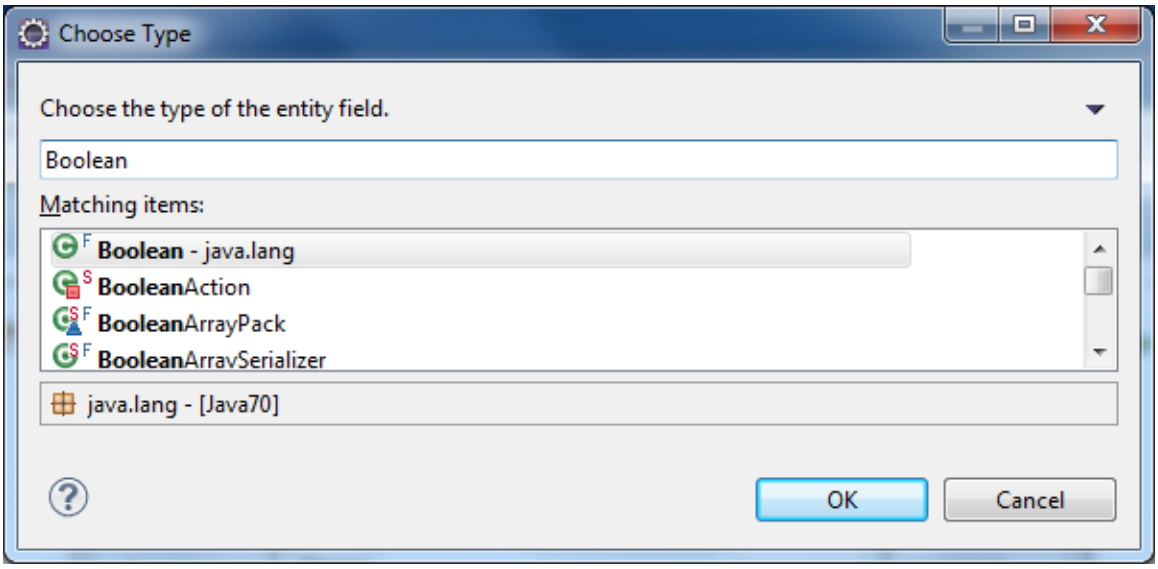
14. Click the Add… button again to add the last field to this entity.



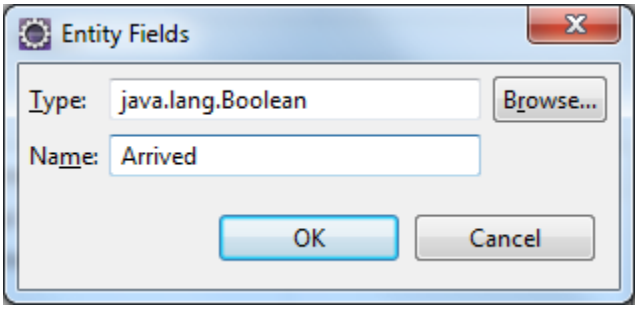
15. Select the **Browse** button again.



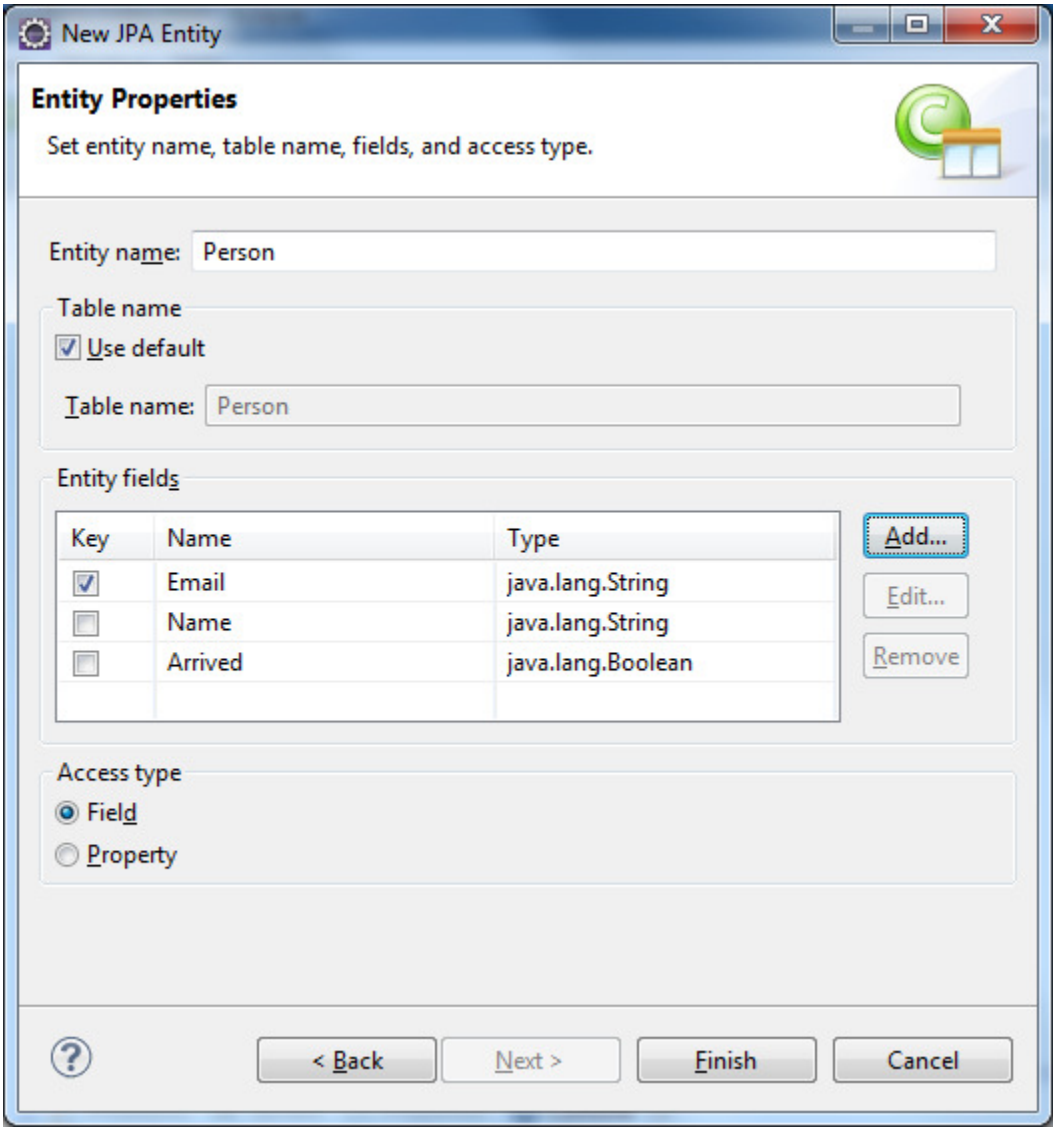
16. Type *Boolean* into the filter box, and select the `Boolean` class from the `java.lang` package. Then click **OK**.



17. Set a **Name** of *Arrived*, then click **OK**.



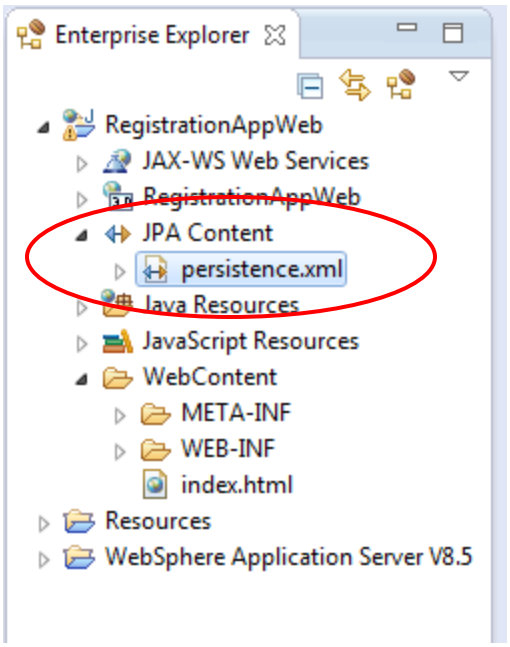
18. The set up of the entity should look as below. Click **Finish** to create the entity.



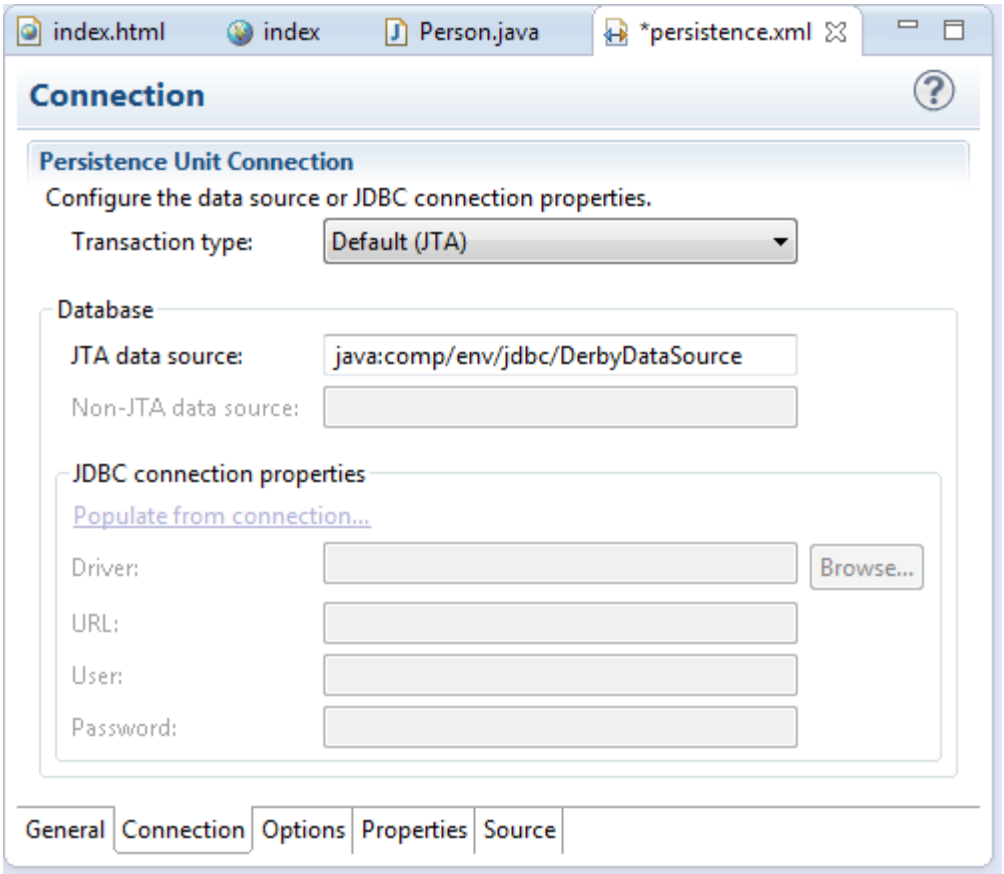
Create the Persistence Unit

JPA uses Persistence Units to define the types of data required to be stored by your application. We will configure this in an XML file called `persistence.xml`, which was created for us when the JPA project Facet was added.

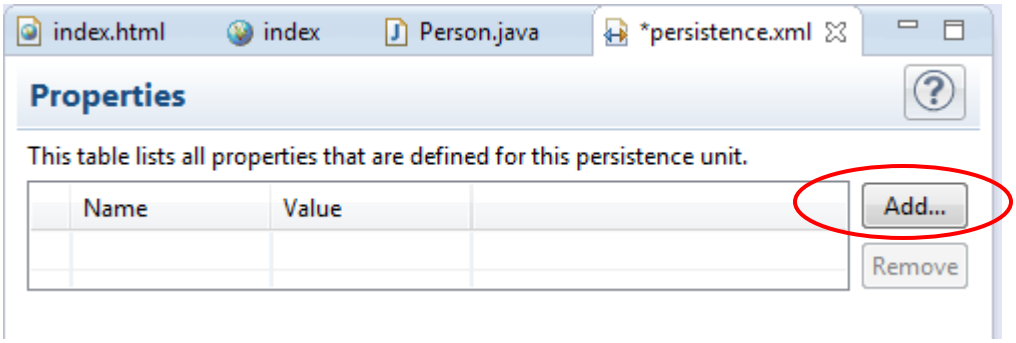
1. Expand the JPA Content section of the RegistrationAppWeb project, and double click on the persistence.xml file.



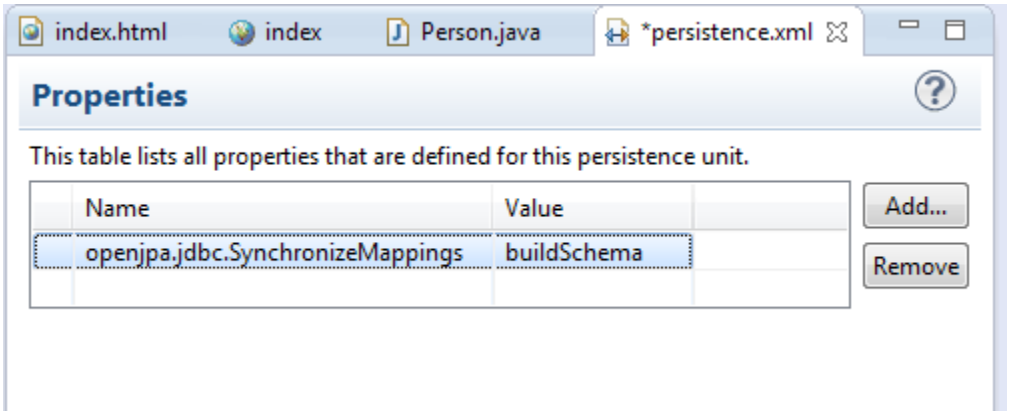
2. Switch to the **Connection** tab, and set the **JTA data source** to *java:comp/env/jdbc/DerbyDataSource*.



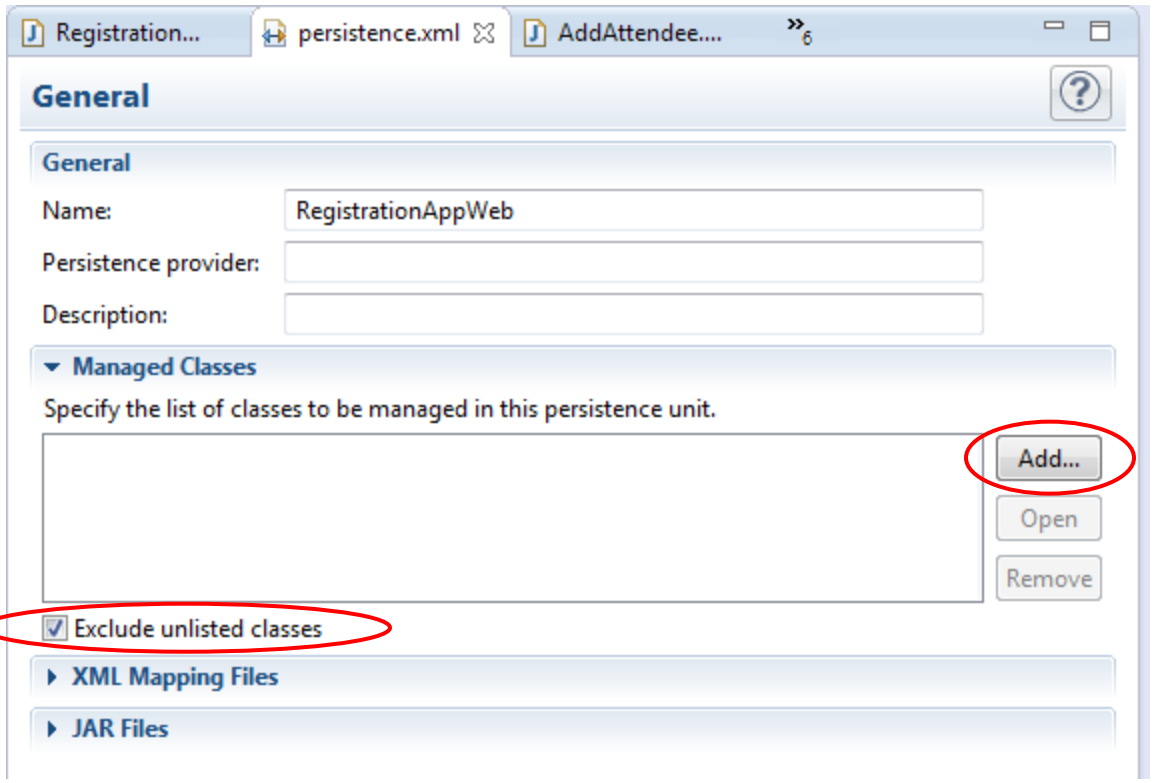
3. Switch to the Properties tab, and click the Add… button.



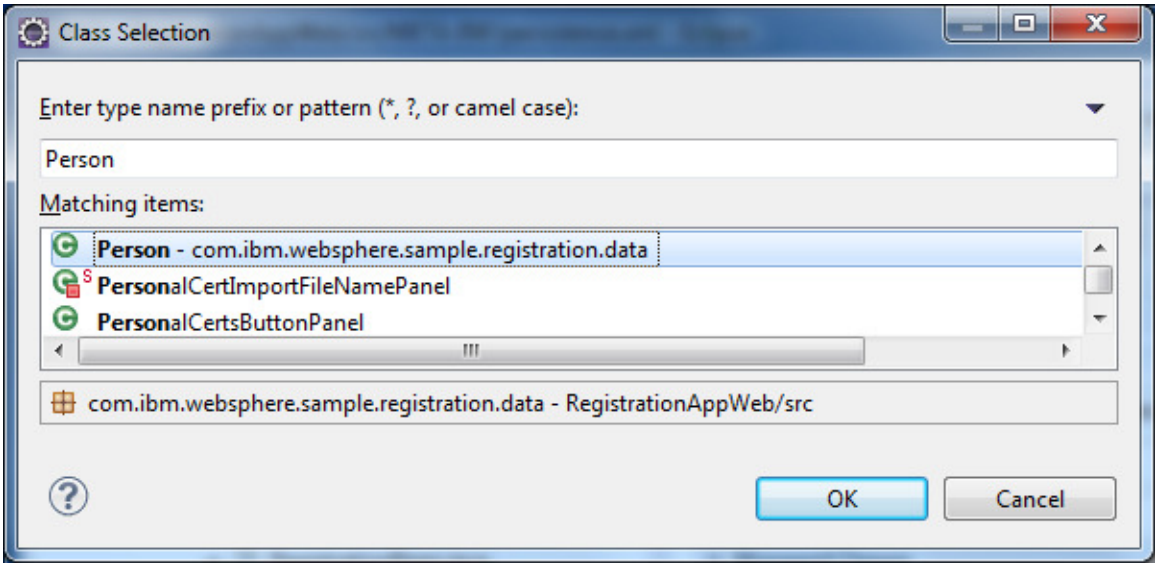
4. Set the **Name** to *openjpa.jdbc.SynchronizeMappings*, and the **Value** to *buildSchema*.



5. Switch to the **General** tab. Expand the **Managed Classes** section. Check the checkbox for **Exclude unlisted classes**, then click the **Add…** button.



6. In the filter box, type *Person*. Select the `Person` class from the package `com.ibm.websphere.sample.registration.data`, and then click **OK**.

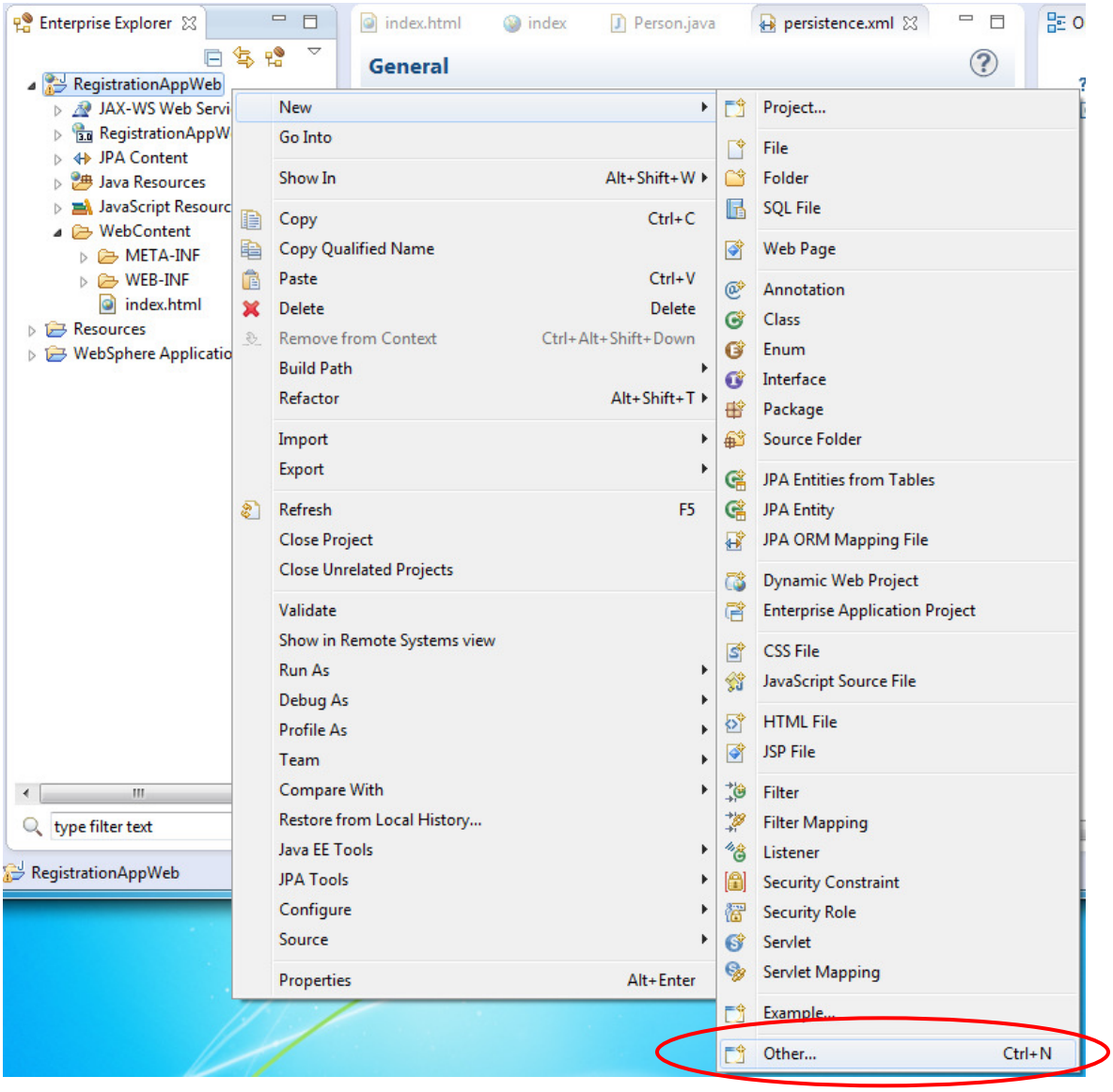


7. Use the **control-S** shortcut to save the file.

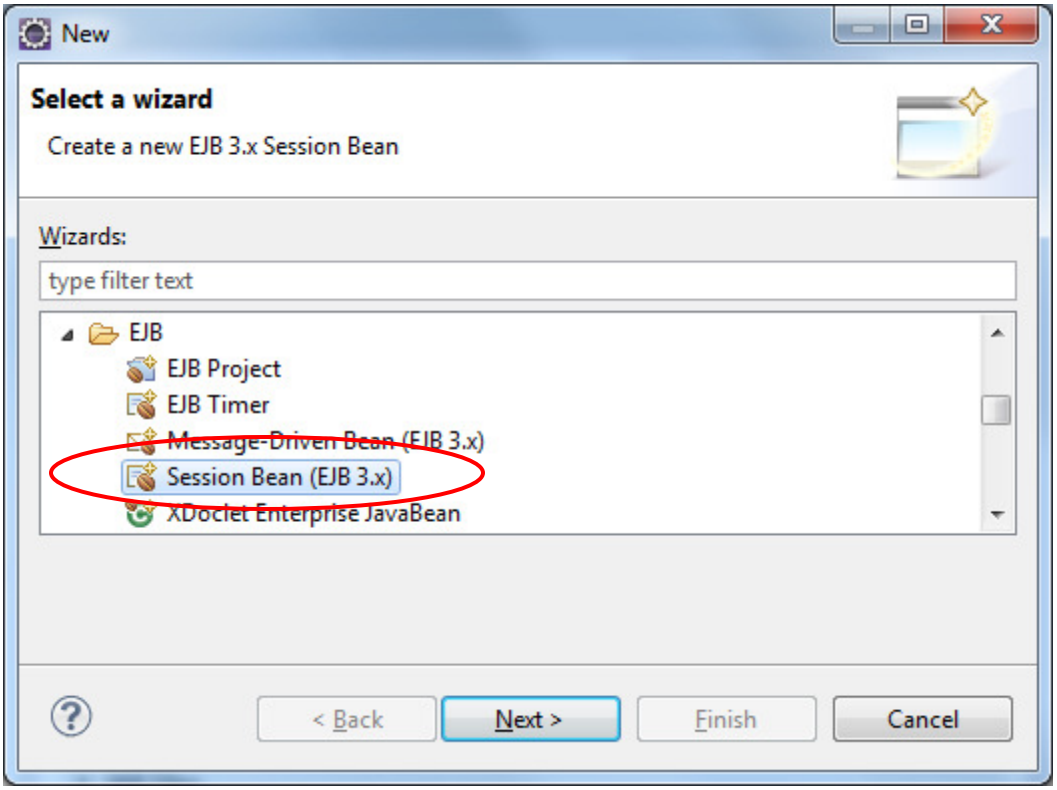
Building the EJB

Enterprise Java Beans (EJBs) are a useful programming model to assist separating Business logic from presentation logic. This application uses an EJB to separate to process of managing an attendee from the presenting of information about attendees. Creating this EJB is the next task.

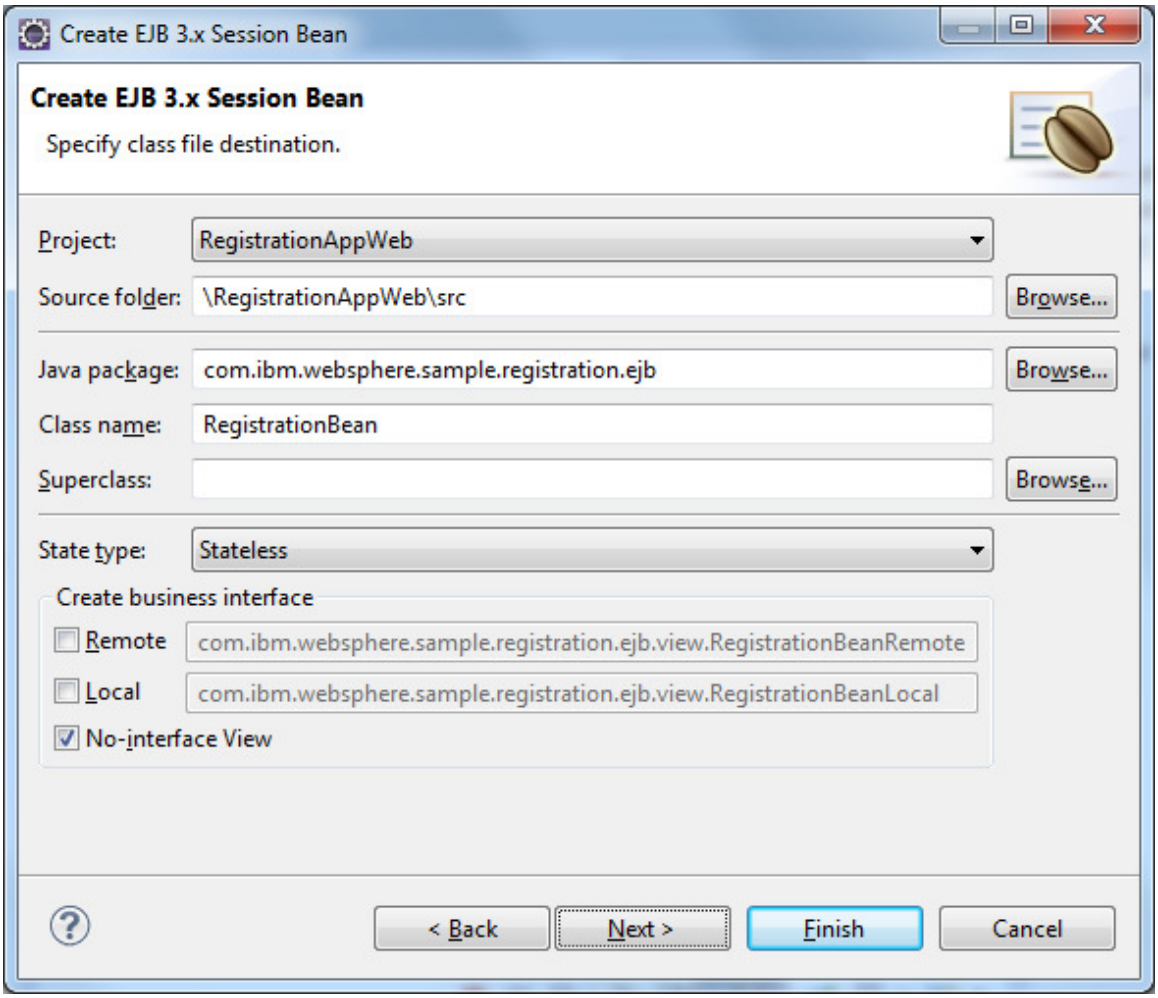
1. Right click on the **RegistrationAppWeb** and select **New > Other…**.



2. In the **New Wizard** dialog, expand the section for **EJB** and select the **Session Bean (EJB 3.x)**. Then click **Next**.



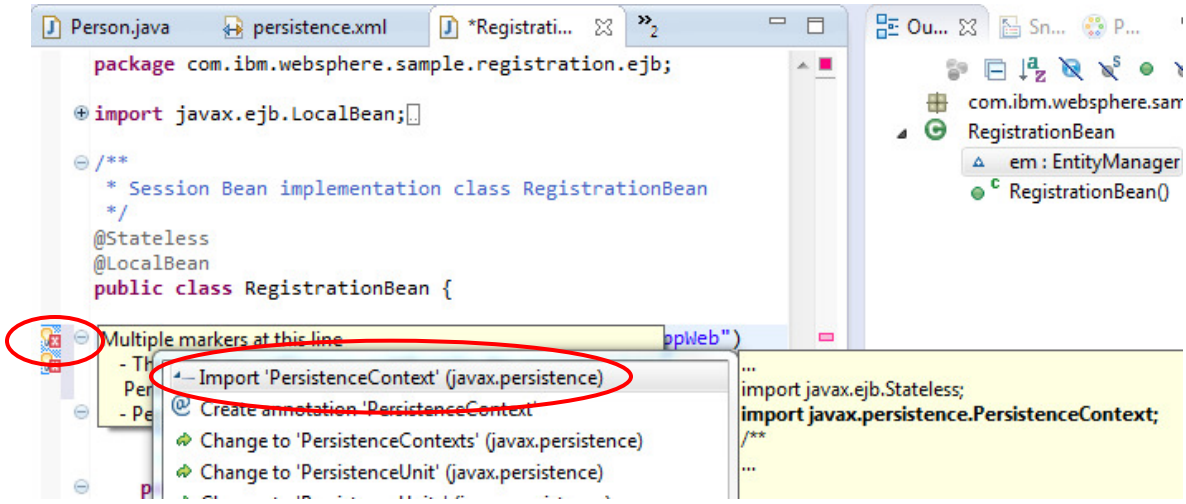
3. Set the **Java package** to *com.ibm.websphere.sample.registration.ejb*, and set the **Class name** to *RegistrationBean*. Click **Finish**.



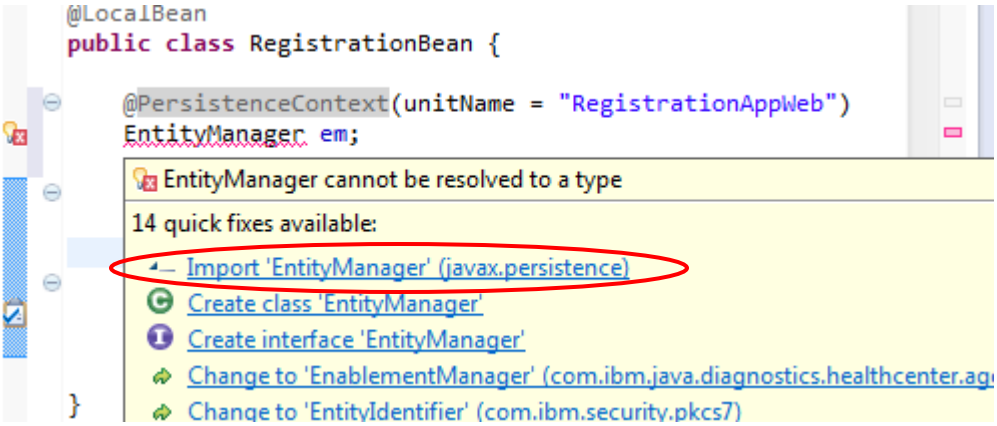
4. This will open the `RegistrationBean`. This class requires access to an `EntityManager` to provide JPA function, so add the following lines just below the class definition:

```
@PersistenceContext(unitName = "RegistrationAppWeb")
EntityManager em;
```

5. This will cause errors to appear, as the classes have not been imported. To fix this, begin by clicking on the red X to the left of the `@PersistenceContext` annotation, and double clicking **Import 'PersistenceContext' (javax.persitence)** from the menu.



6. Next hover over the `EntityManager` and select **Import 'EntityManager' (javax.persistence)** from the pop-up menu.



7. Now we will add the business methods. First, add the method to register users by placing the following code below the RegistrationBean constructor:

```
public void register(String name, String email) {
  Person p = new Person();
  p.setEmail(email);
  p.setName(name);
  em.persist(p);
}
```

8. Next, add a method to remove users by adding the code below just below the previous one:

```
public void unregister(String email) {
  Person p = em.find(Person.class, email);
  em.remove(p);
}
```

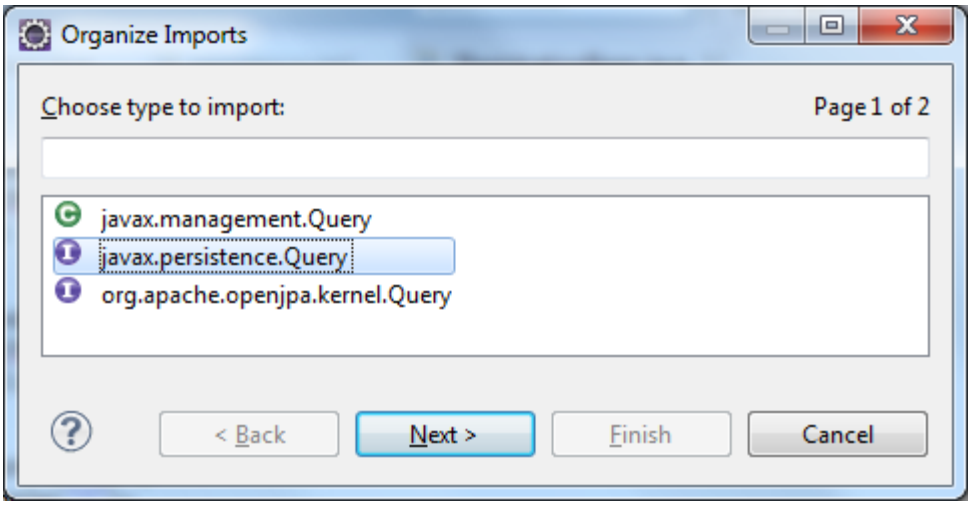
9. Next, add the functions to mark whether a person has attended or not by adding the following code below the previous set:

```
public void markAttended(String email) {
  Person p = em.find(Person.class, email);
  p.setArrived(true);
}

public void markUnattended(String email) {
  Person p = em.find(Person.class, email);
  p.setArrived(false);
}
```

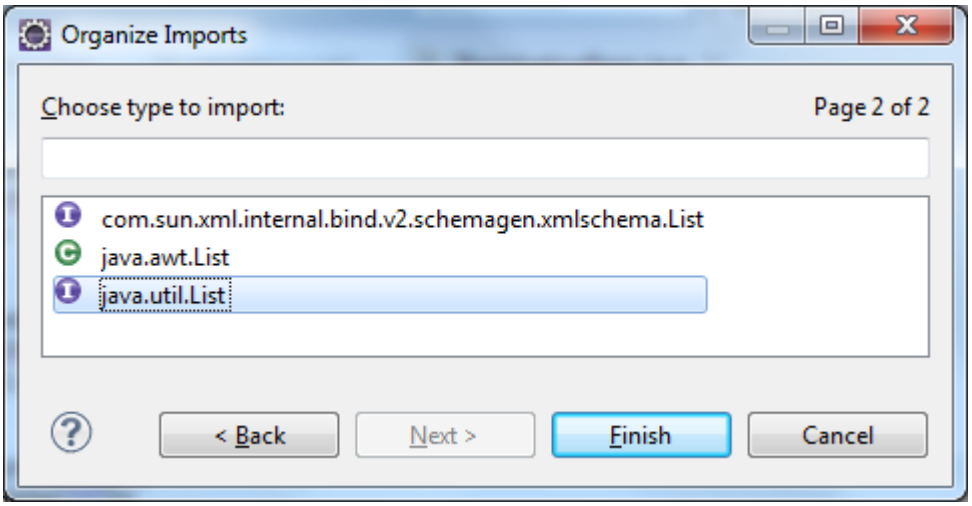
10. Add the following code below the previous set, in order to allow searches and retrieval of specific attendees.

```
public List<Person> getPeople() {
  return searchForPeople("*");
}

public Person getPerson(String email) {
  return em.find(Person.class, email);
}

public List<Person> searchForPeople(String searchTerm)
{
  String queryString = "SELECT p FROM Person p WHERE
p.Name LIKE :searchTerm OR p.Email LIKE :searchTerm";
  Query query = em.createQuery(queryString);
  String cleanedSearchTerm = "%";

  if (searchTerm != null) {
    cleanedSearchTerm = "%" +
searchTerm.replaceAll("\\*", "%").toLowerCase() + "%";
  }

  query.setParameter("searchTerm", cleanedSearchTerm);

  @SuppressWarnings("unchecked")
  List<Person> people = query.getResultList();

  return people;
}
```

11. Use the **control-shift-o** shortcut to automatically set up imports. This will present a dialog asking for clarification of which packages to use for specific classes. To begin, select **javax.persistence.Query**, then click **Next**.



12. On the next dialog, select **java.util.List**, then click **Finish**.



13. Use the **control-S** shortcut to save the file. The file should resemble the screenshot below.

index.html | index | Person.java | persistence.xml | RegistrationBean.java

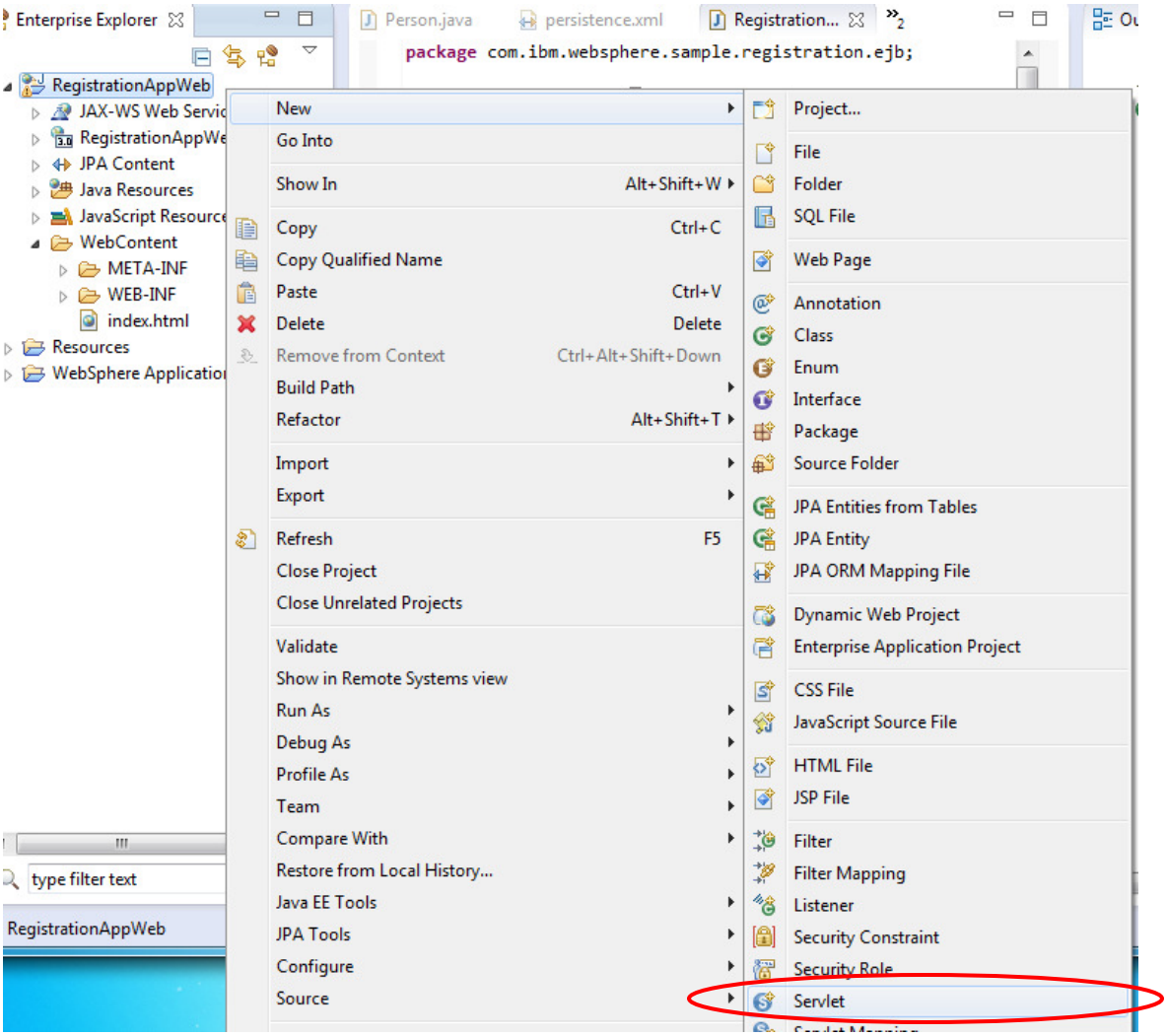
```
public class RegistrationBean {

    @PersistenceContext(unitName = "RegistrationAppWeb")
    EntityManager em;

    /**
     * Default constructor.
     */
    public RegistrationBean() {
        // TODO Auto-generated constructor stub
    }

    public void register(String name, String email) {
        Person p = new Person();
        p.setEmail(email);
        p.setName(name);
        em.persist(p);
    }

    public void unregister(String email) {
        Person p = em.find(Person.class, email);
        em.remove(p);
    }

    public void markAttended(String email) {
        Person p = em.find(Person.class, email);
        p.setArrived(true);
    }

    public void markUnattended(String email) {
        Person p = em.find(Person.class, email);
        p.setArrived(false);
    }

    public List<Person> getPeople() {
        return searchForPeople("*");
    }

    public Person getPerson(String email) {
        return em.find(Person.class, email);
    }

    public List<Person> searchForPeople(String searchTerm) {
        String queryString = "SELECT p FROM Person p WHERE p.Name LIKE :searchTerm OR p.Email LIKE :searchTerm";
        Query query = em.createQuery(queryString);
        String cleanedSearchTerm = "%";

        if (searchTerm != null) {
            cleanedSearchTerm = "%" + searchTerm.replaceAll("\\*", "%").toLowerCase() + "%";
        }

        query.setParameter("searchTerm", cleanedSearchTerm);

        @SuppressWarnings("unchecked")
        List<Person> people = query.getResultList();

        return people;
    }
}
```

Adding Servlets

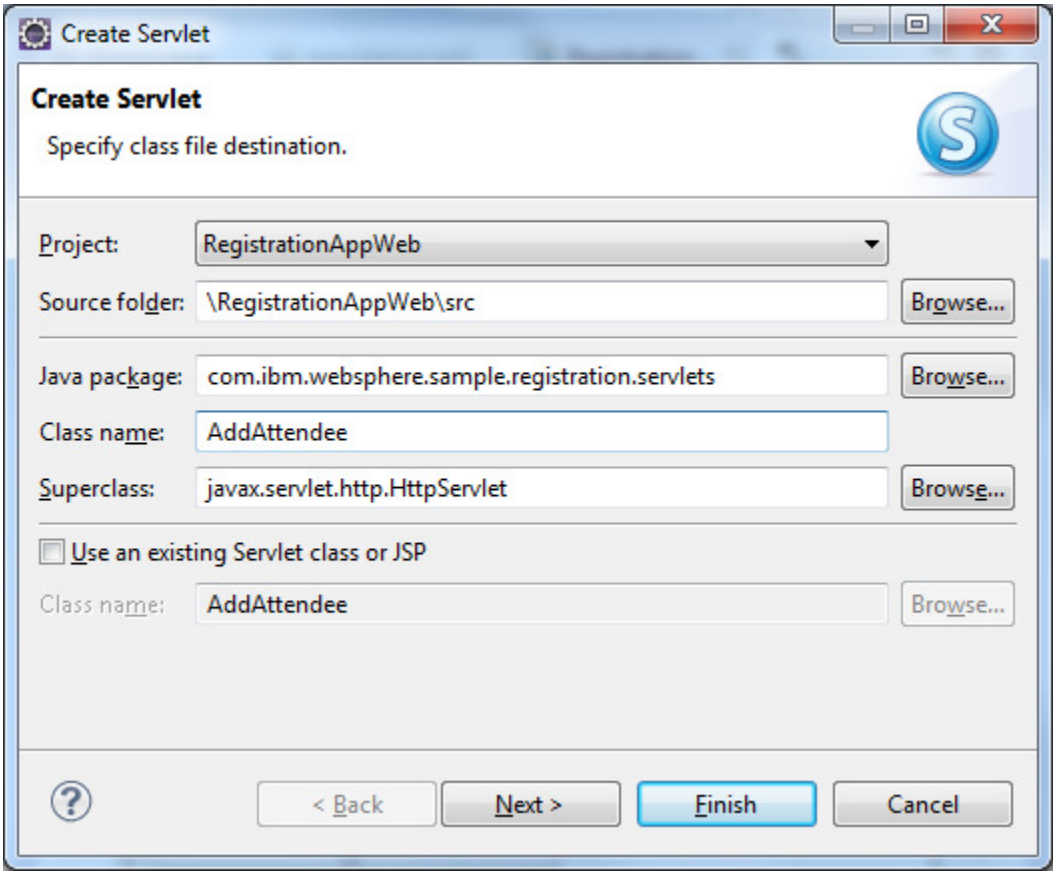
To register and edit users, you will use servlets to drive methods of the `RegistrationBean`. Note that this could also be achieved via JAX-RS, and we will use JAX-RS later in the lab.

AddAttendee servlet

1. Right click on the **RegistrationAppWeb** project, and select **New >Servlet**.



2. Provide a **Java package** of *com.ibm.websphere.sample.registration.servlets*, and a **Class name** of *AddAttendee*. Click **Finish**.



3. We need to define the data resources used by JPA. These can be defined in multiple locations, in our case we are using an annotation. Add this just below the `WebServlet` annotation in the `AddAttendee` class:

```
@Resources({
    @Resource(name = "jdbc/DerbyDataSource", type =
javax.sql.DataSource.class),
    @Resource(name = "jdbc/NonTxDerbyDataSource", type
= javax.sql.DataSource.class) })
```

4. Use the **control-shift-o** shortcut to import the required packages and classes.

5. This servlet will be used to register the attendees, so you need to inject the `RegistrationBean`. To do this, add the following line to the class, just above the constructor:

```
@Inject RegistrationBean rb;
```

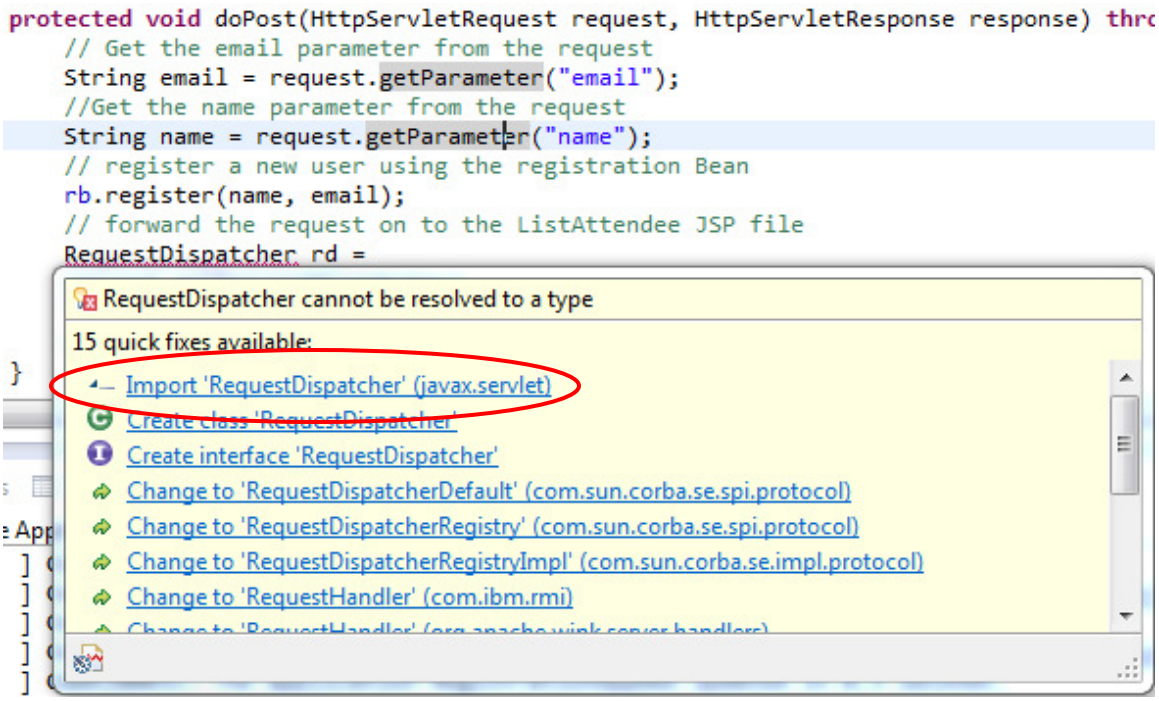
6. Hover over the `@Inject` annotation and select **Import 'Inject' (javax.inject)** from the dialog.

7. Next, hover over the `RegistrationBean` class and select **Import 'RegistrationBean' (com.ibm.websphere.sample.registration.ejb)** from the pop-up menu.

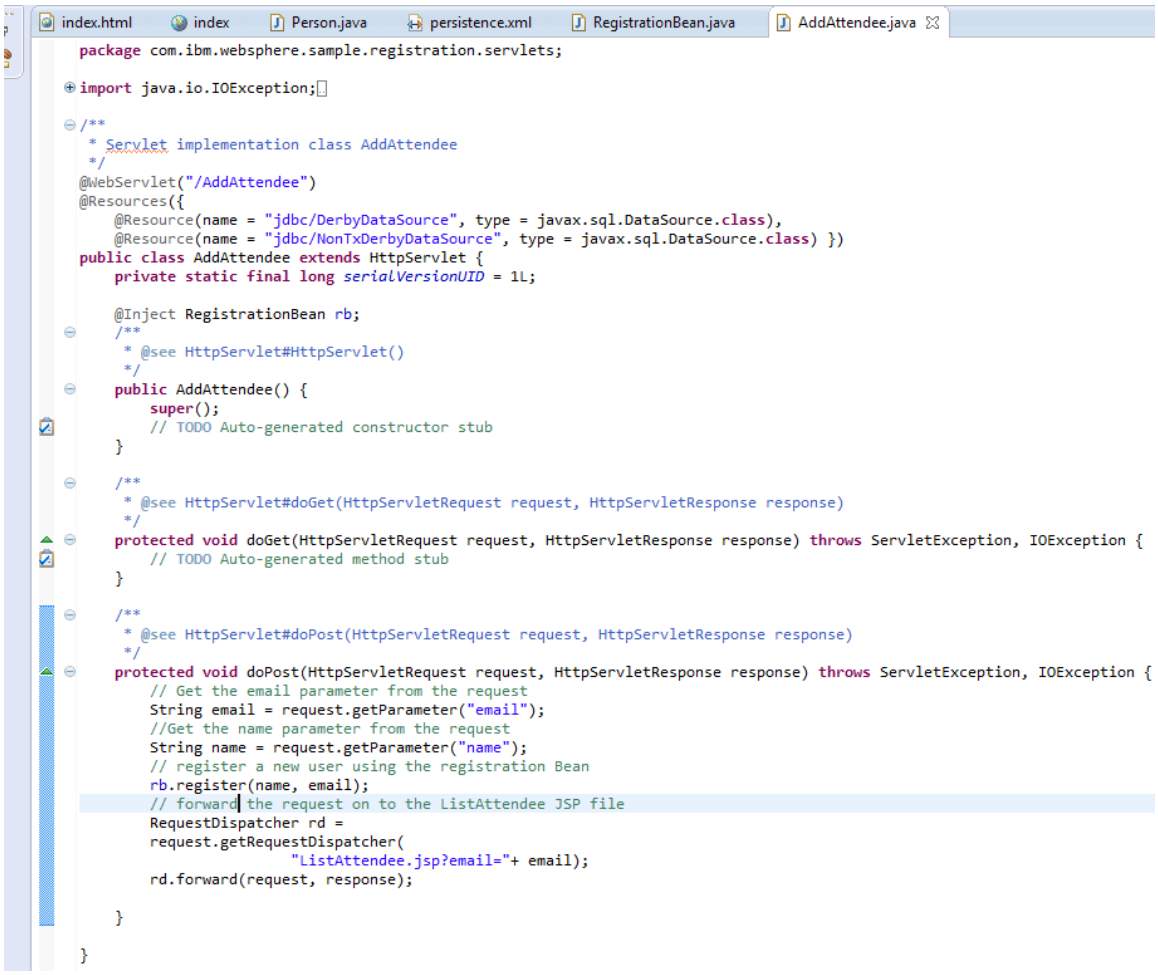
8. Next, add the following lines to the `doPost()` method of the `AddAttendee` class:

```
// Get the email parameter from the request
String email = request.getParameter("email");
//Get the name parameter from the request
String name = request.getParameter("name");
// register a new user using the registration Bean
rb.register(name, email);
// forward the request on to the ListAttendee JSP file
RequestDispatcher rd =
     request.getRequestDispatcher(
                    "ListAttendee.jsp?email="+ email);
rd.forward(request, response);
```

9. Hover over `RequestDispatcher` and select **Import 'RequestDispatcher' (javax.servlet)** from the dialog.



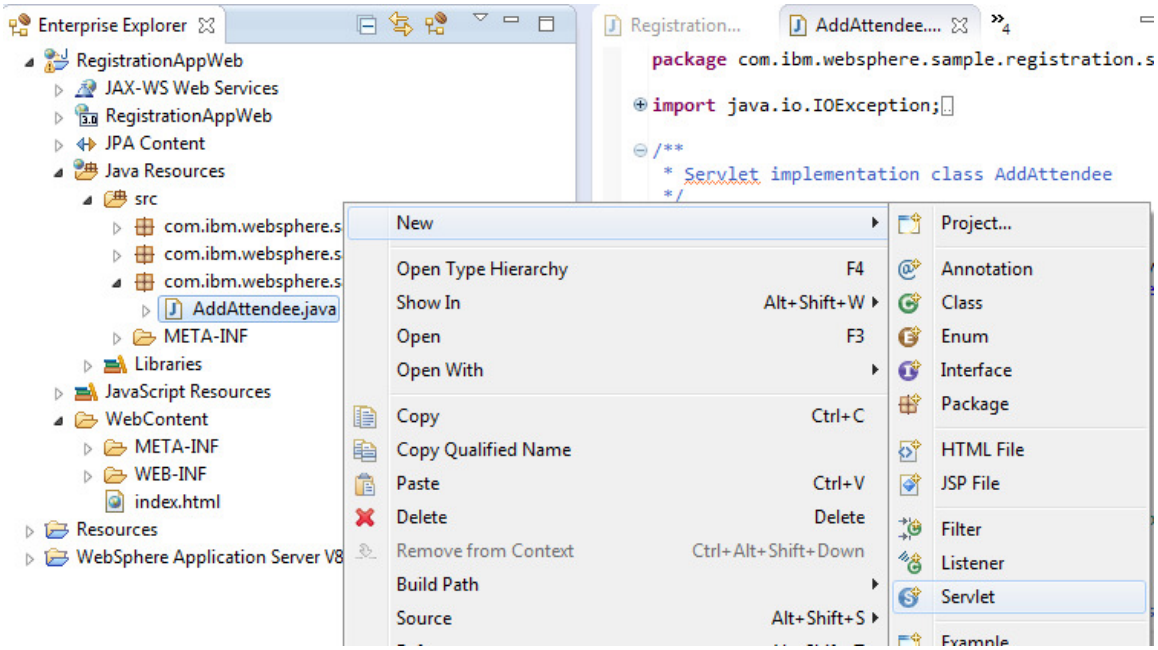
10. Save the file using the **control-S** shortcut. The completed file should match the image below.



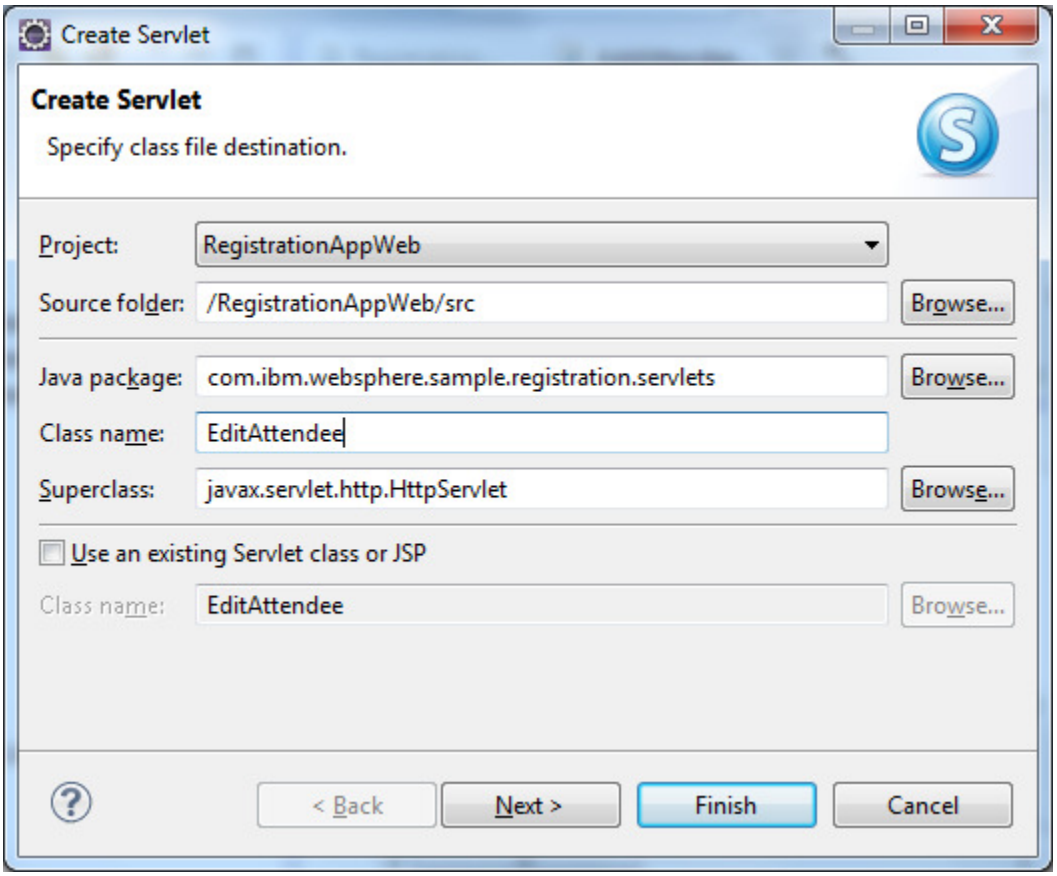
EditAttendee servlet

As well as adding an attendee, we also need to be able to edit attendees. To do this, we will create the `EditAttendee` servlet.

1. Expand the **Java Resources** section of the **RegistrationAppWeb** project, then expand the **src** directory inside. Next, expand the **com.ibm.websphere.sample.registration.servlets** package. Right click on the `AddAttendee` servlet and select **New > Servlet**.



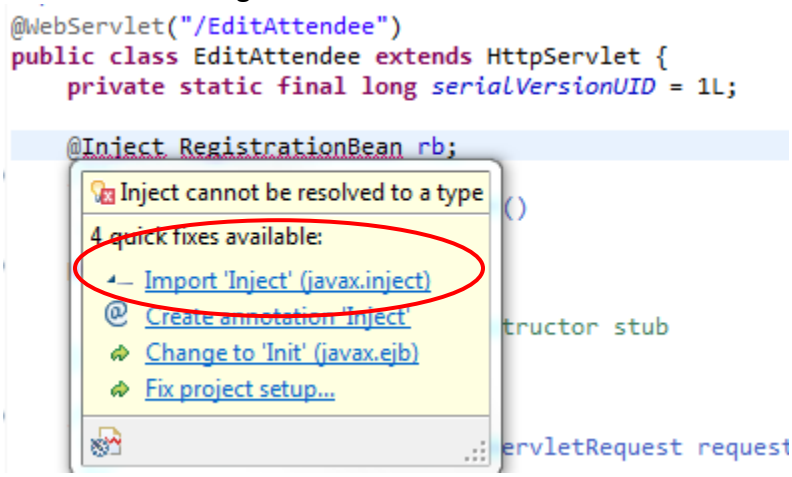
2. Clear the checkbox for **Use an existing Servlet class or JSP**, and provide a **Class name** of *EditAttendee*. Click **Finish**.



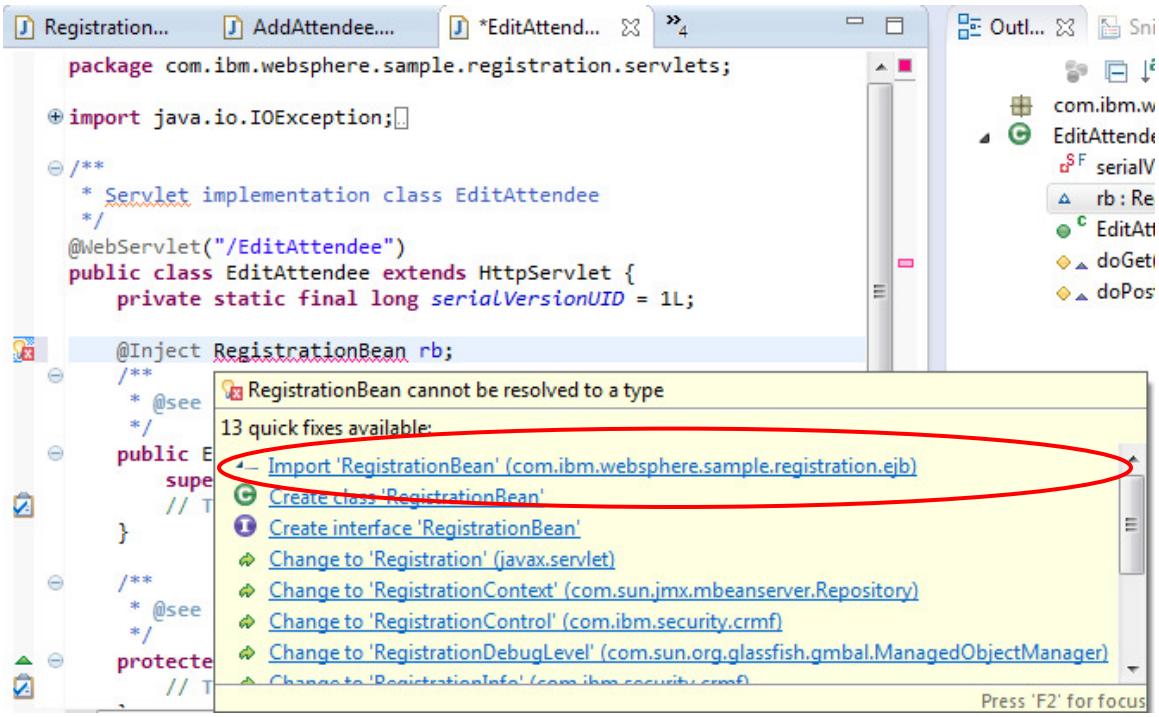
3. Add the following line just above the constructor of `EditAttendee.java`:

```
@Inject RegistrationBean rb;
```

4. Hover over the `@Inject` annotation and select **Import 'Inject' (javax.inject)** from the dialog.



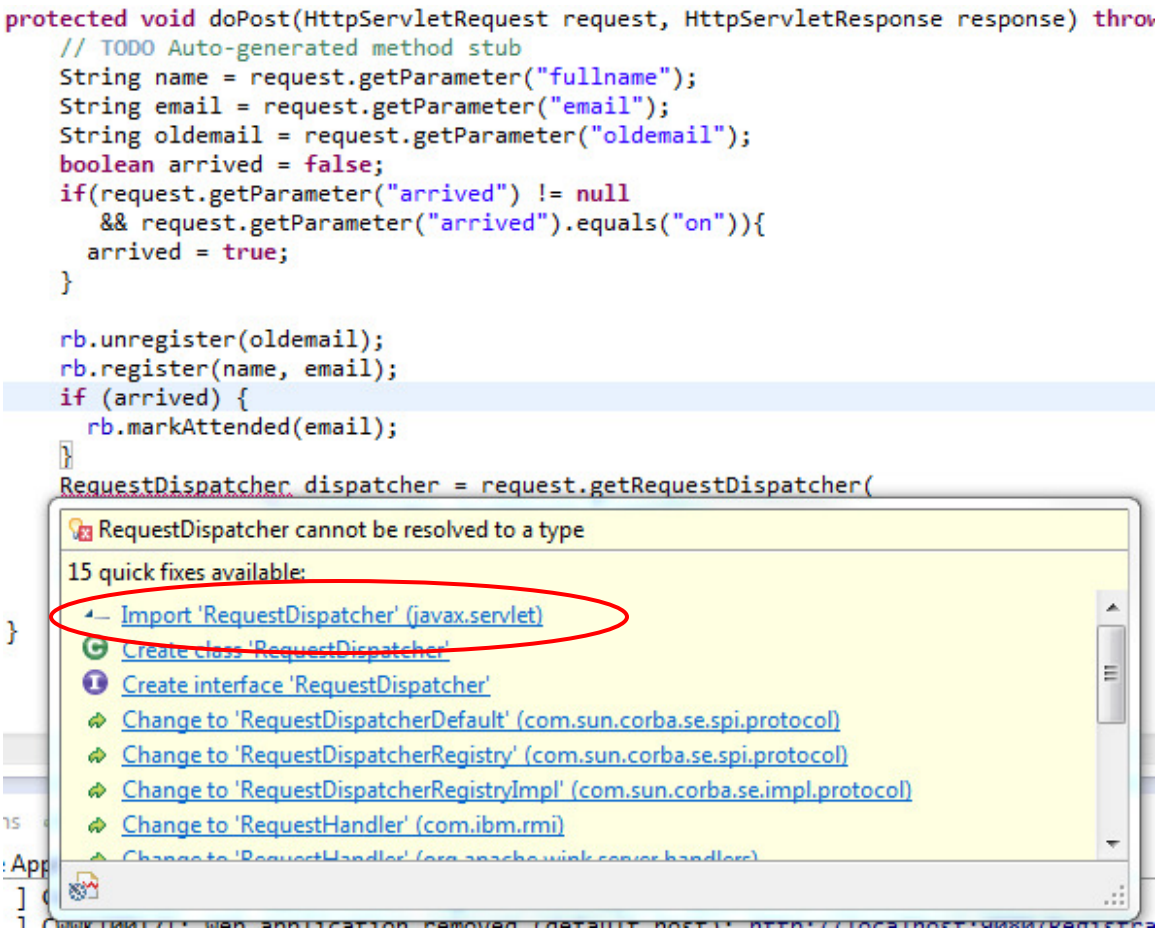
5. Hover over the `RegistrationBean` class and select **Import 'RegistrationBean' (com.ibm.websphere.sample.registration.ejb)** from the dialog.



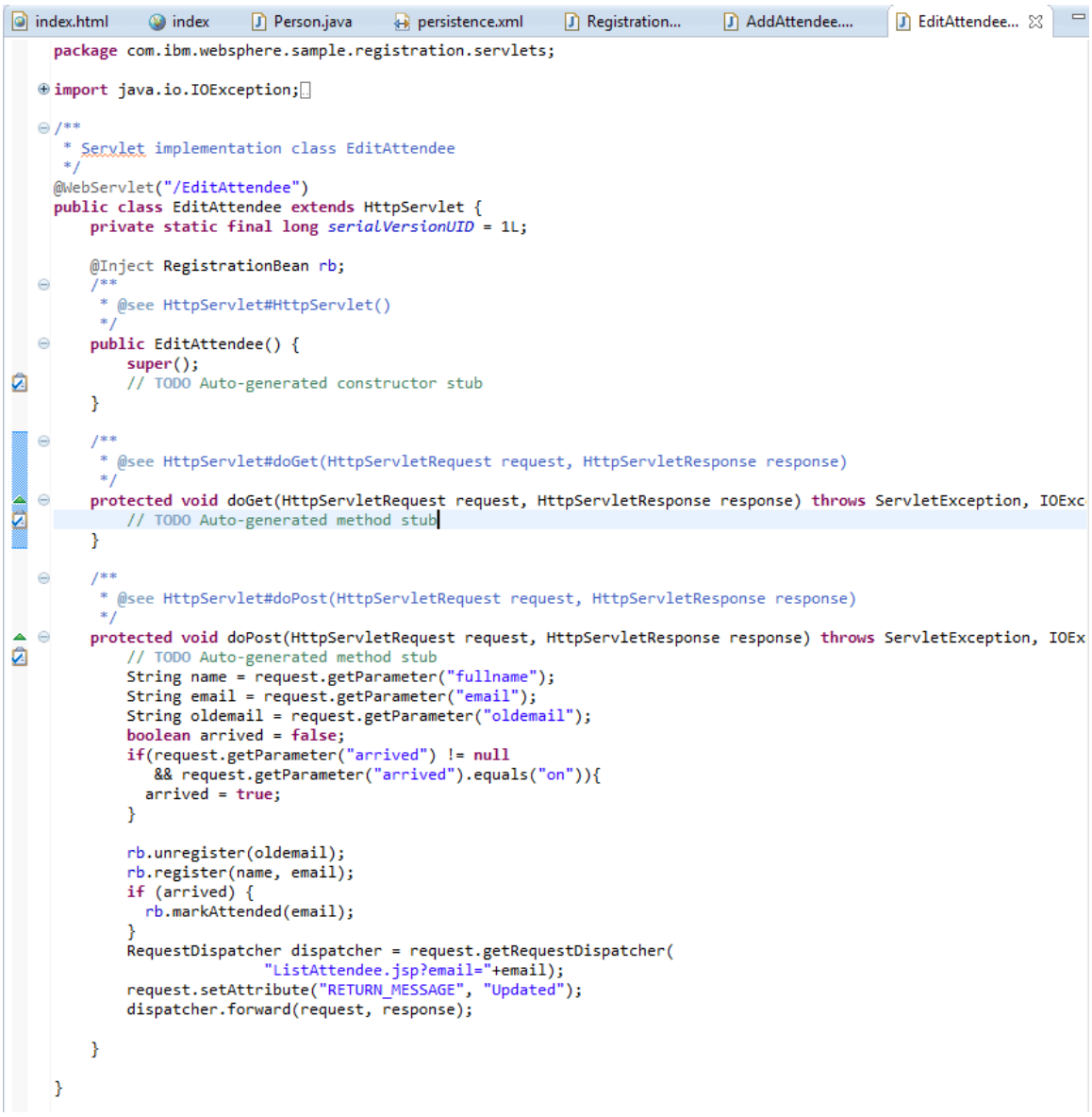
6. In the `doPost()` method, add the following code:

```
String name = request.getParameter("fullname");
String email = request.getParameter("email");
String oldemail = request.getParameter("oldemail");
boolean arrived = false;
if(request.getParameter("arrived") != null
   && request.getParameter("arrived").equals("on")){
  arrived = true;
}

rb.unregister(oldemail);
rb.register(name, email);
if (arrived) {
  rb.markAttended(email);
}
RequestDispatcher dispatcher =
request.getRequestDispatcher(
               "ListAttendee.jsp?email="+email);
request.setAttribute("RETURN_MESSAGE", "Updated");
dispatcher.forward(request, response);
```

7. Hover over `RequestDispatcher` and select **Import 'RequestDispatcher ' (javax.servlet)** from the dialog.



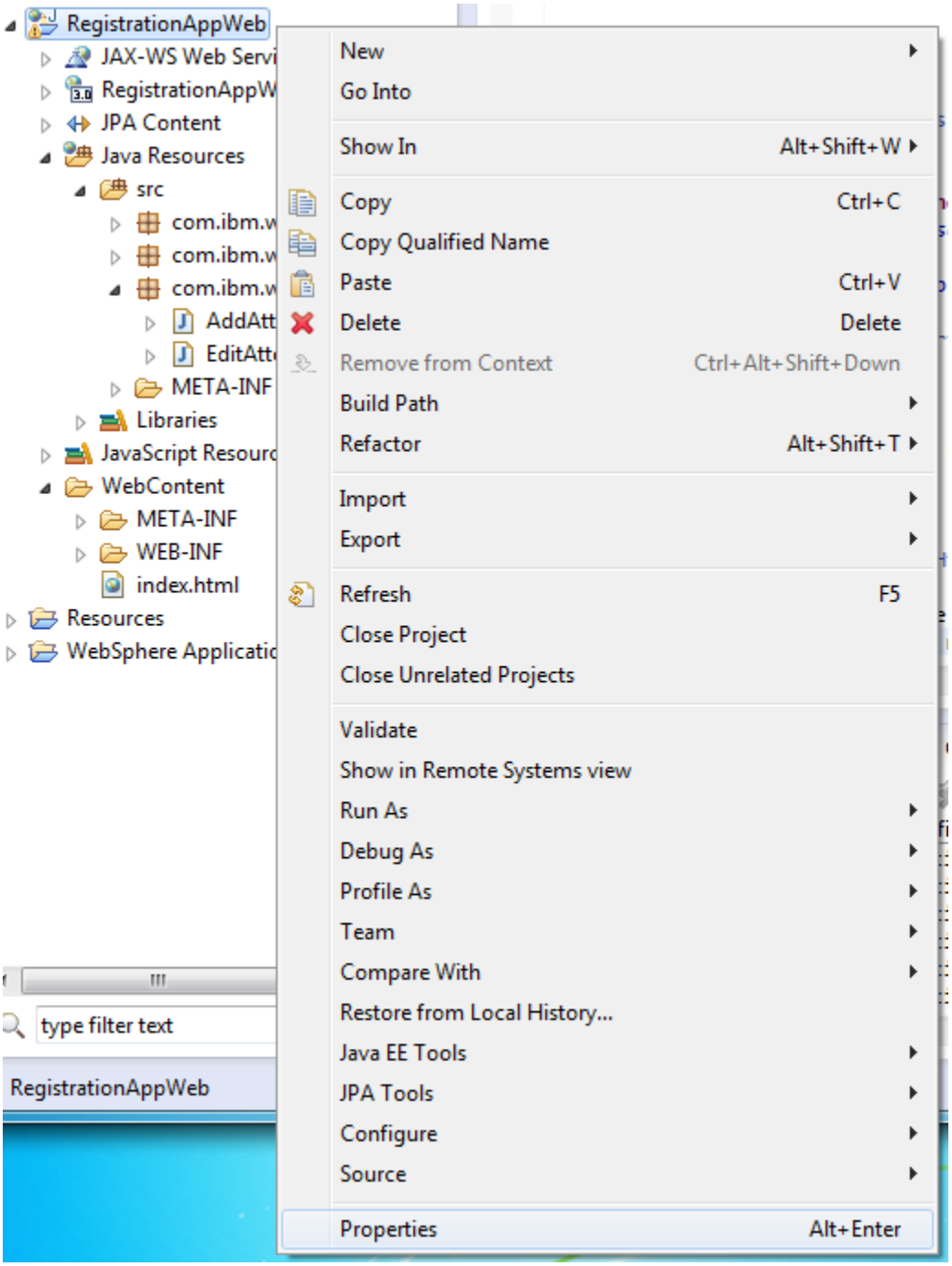
8. Save the file using the **control-S** shortcut. The completed file should match the image below.



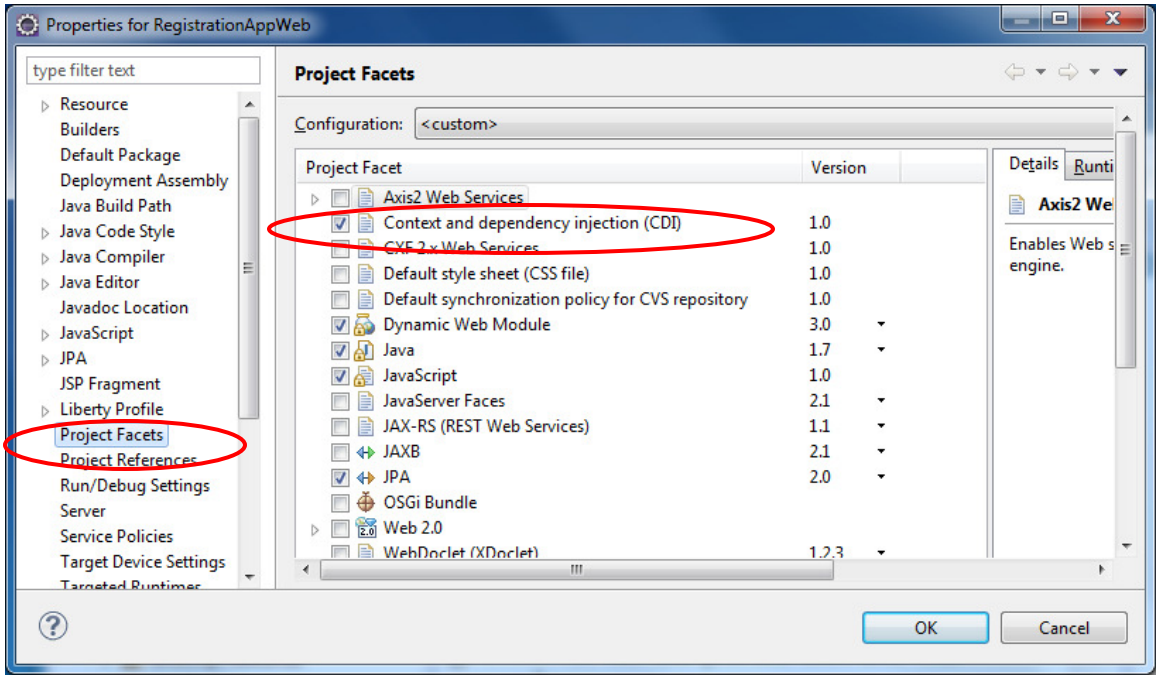
Adding CDI to the Application

To use CDI we need ensure the `beans.xml` file is added. One way to add the `beans.xml` is to add the CDI project facet to the Eclipse project. This ensures the `beans.xml` file is added and the tooling is configured correctly.

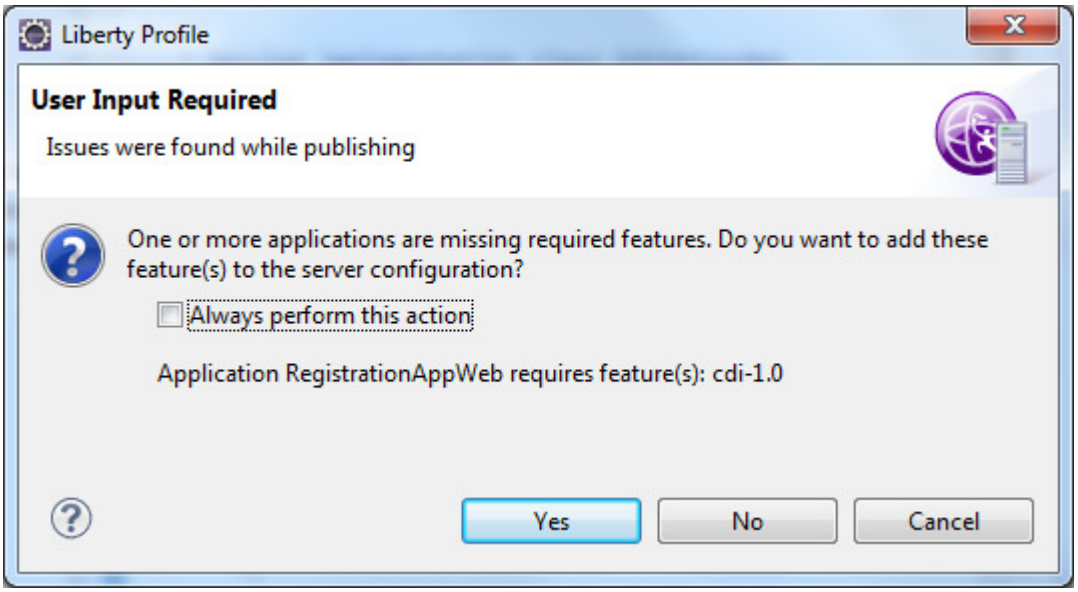
1. Right click on the **RegistrationAppWeb** project, and select **Properties**.



2. Navigate to the **Project Facets**, and check the checkbox for **Context and dependency injection (CDI)**, then click **OK**.



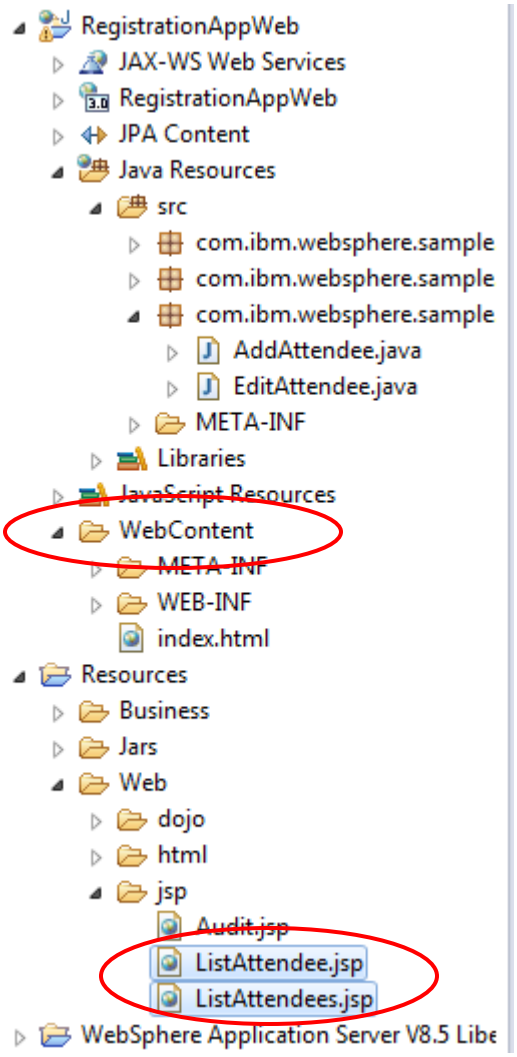
3. If a prompt appears to install the `cdi-1.0` feature, click **Yes**.



Importing JSP resources

All required JSP files are provided in the **Resources** project.

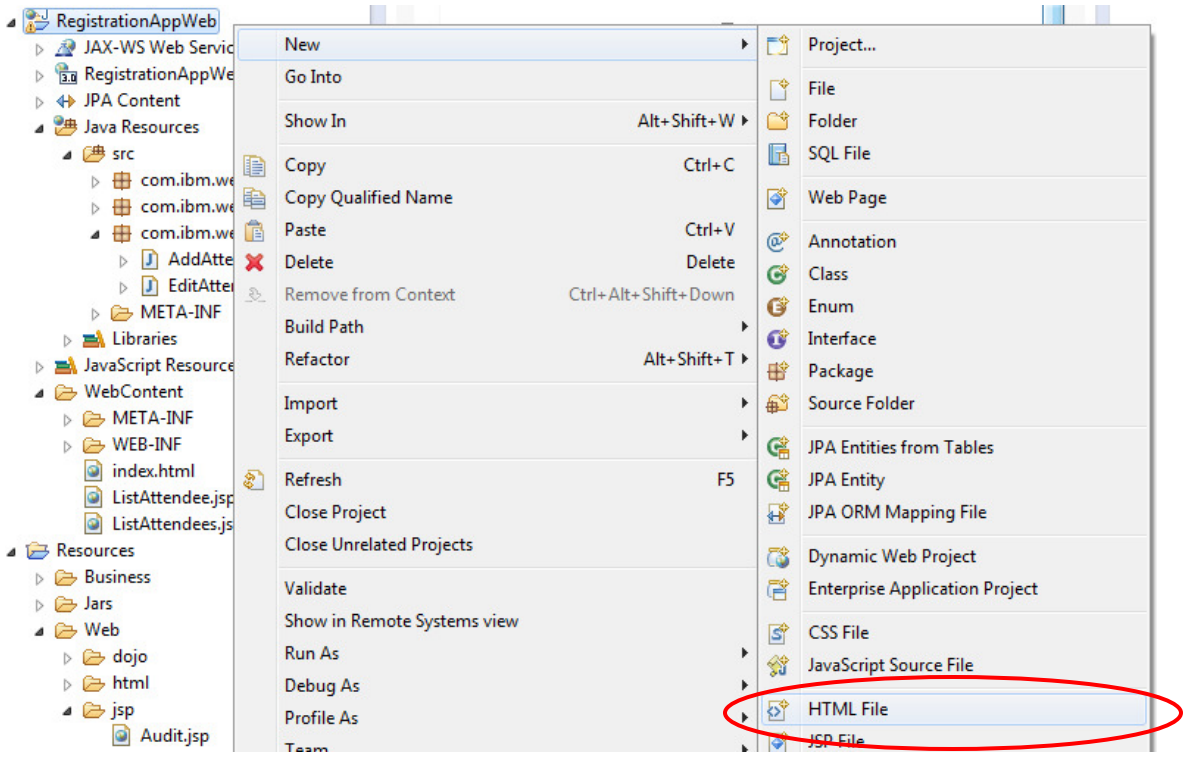
1. Copy `ListAttendee.jsp and ListAttendees.jsp` files by dragging and dropping the files from the `Web/jsp` directory in the **Resources** project to the `WebContent` directory of the **RegistrationAppWeb** project.



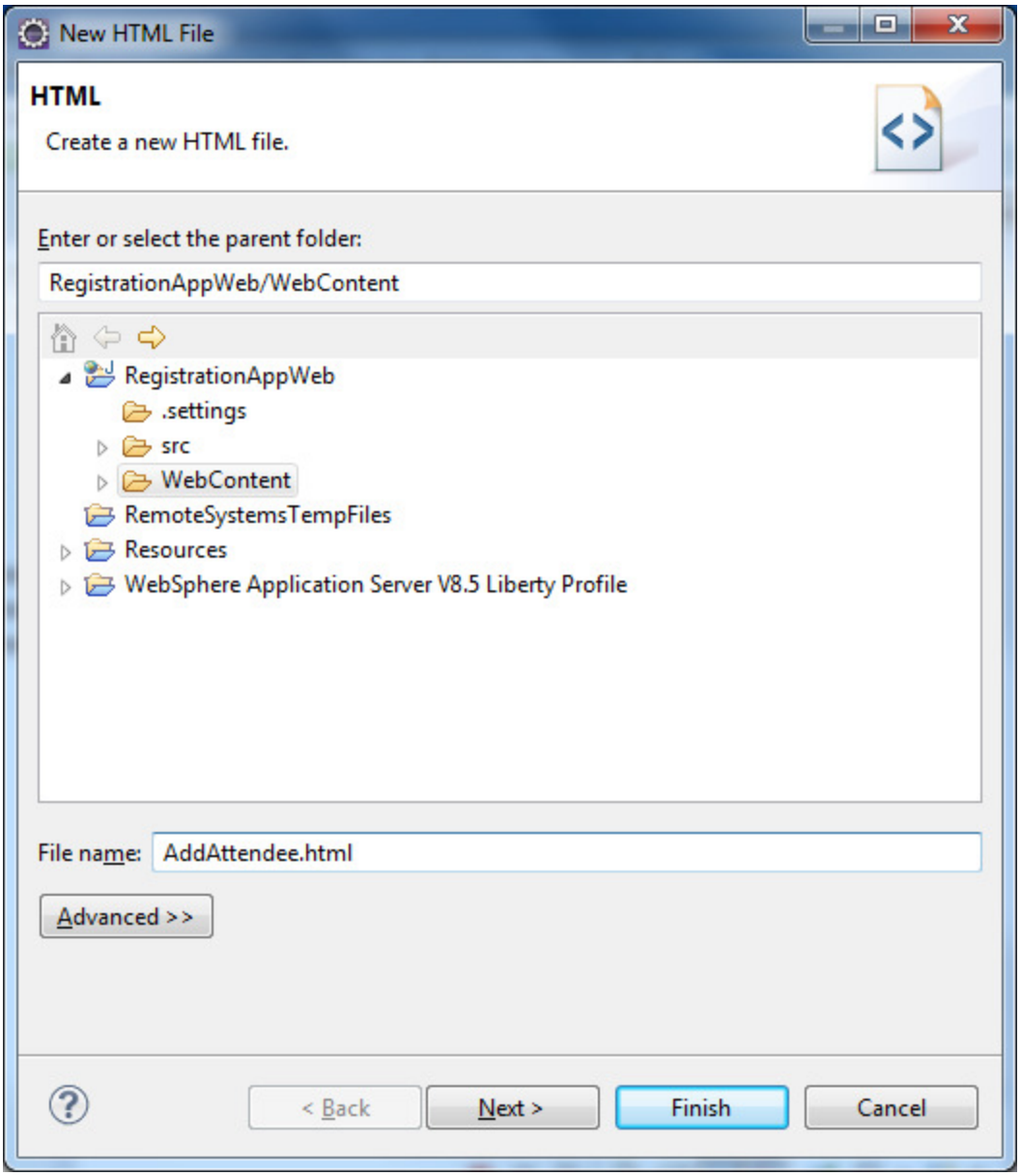
Create the Add Attendee page

To drive the `AddAttendee` servlet, we need to create a new HTML page that contains a form. To create this page, we will use the Web Page editor built into Eclipse.

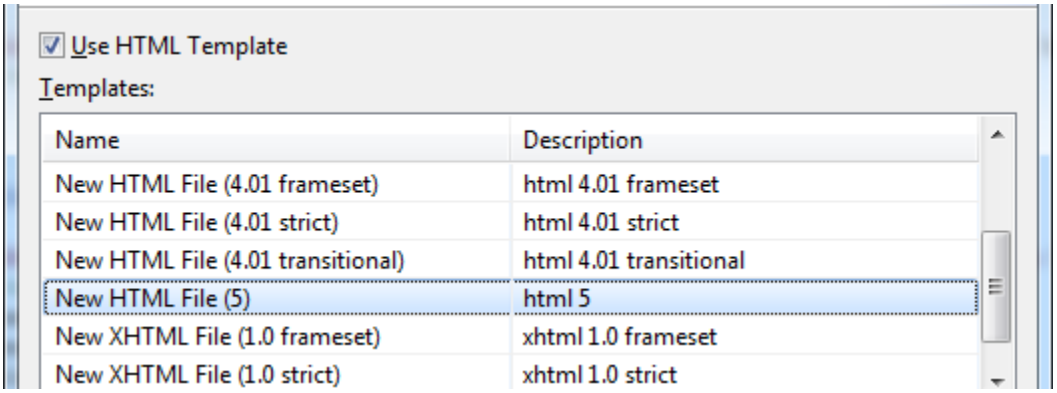
1. Right click on the **RegistrationAppWeb** project, and select **New > HTML File**.



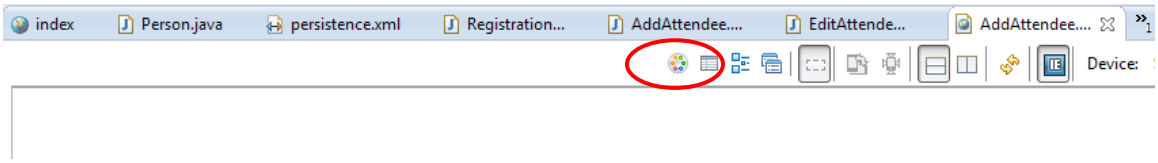
2. Give the file a name of `AddAttendee.html`. Click **Next**.



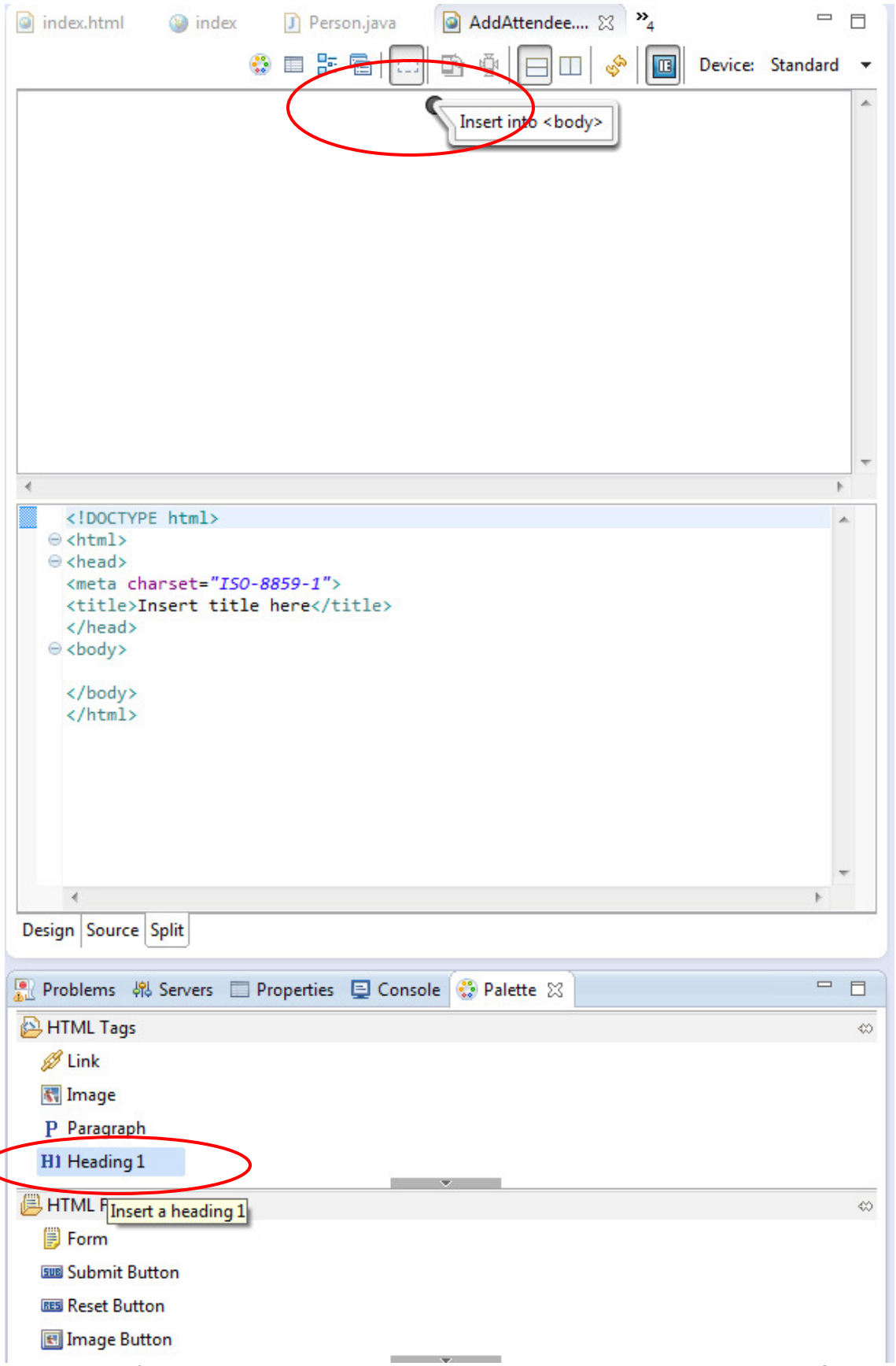
3. Ensure the HTML 5 template is selected. Click **Finish**.



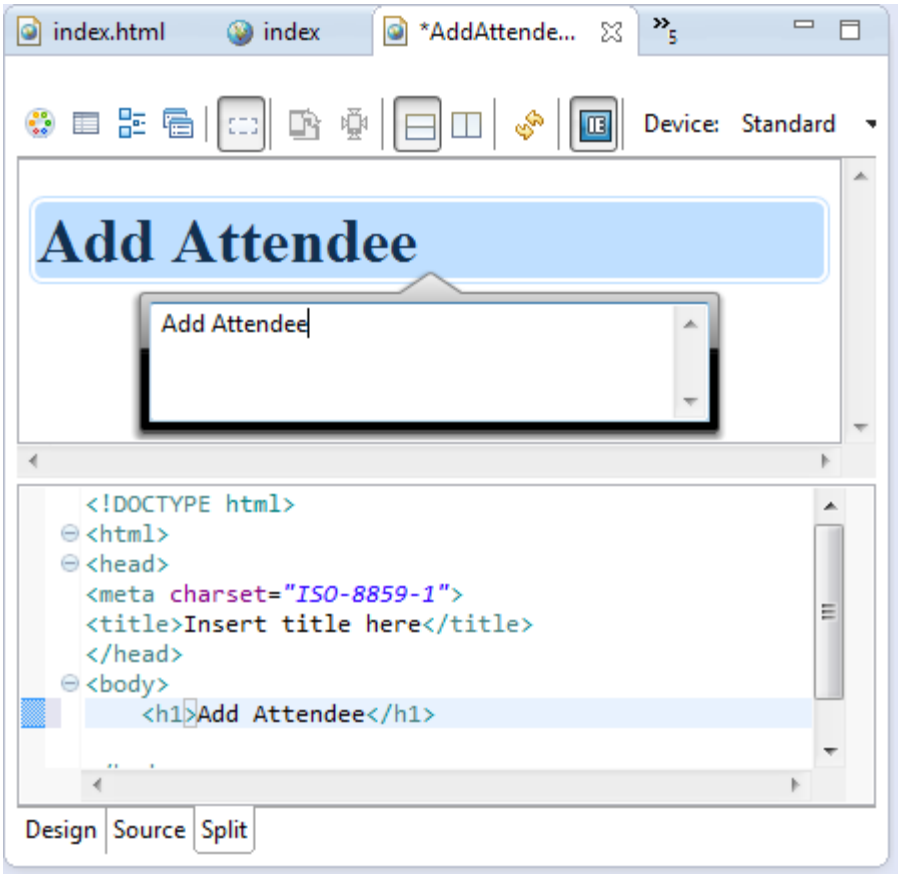
4. In the HTML editor, click the *Palette* icon at the top to open the **Palette** view.



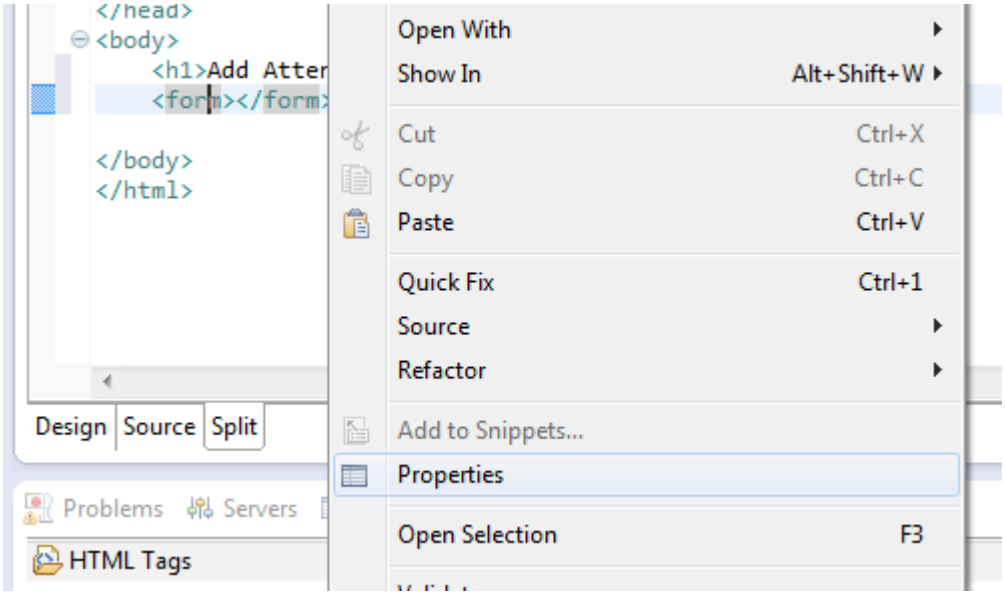
5. From the **Palette**, select the **Heading 1** from the **HTML Tags** section, and drag and drop it into the **Design** section.



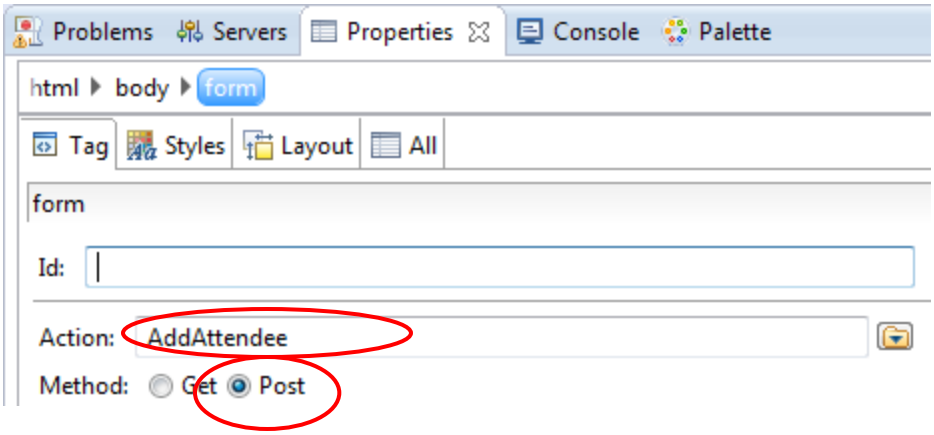
6. Double click the box it creates, and type in “Add Attendee”.



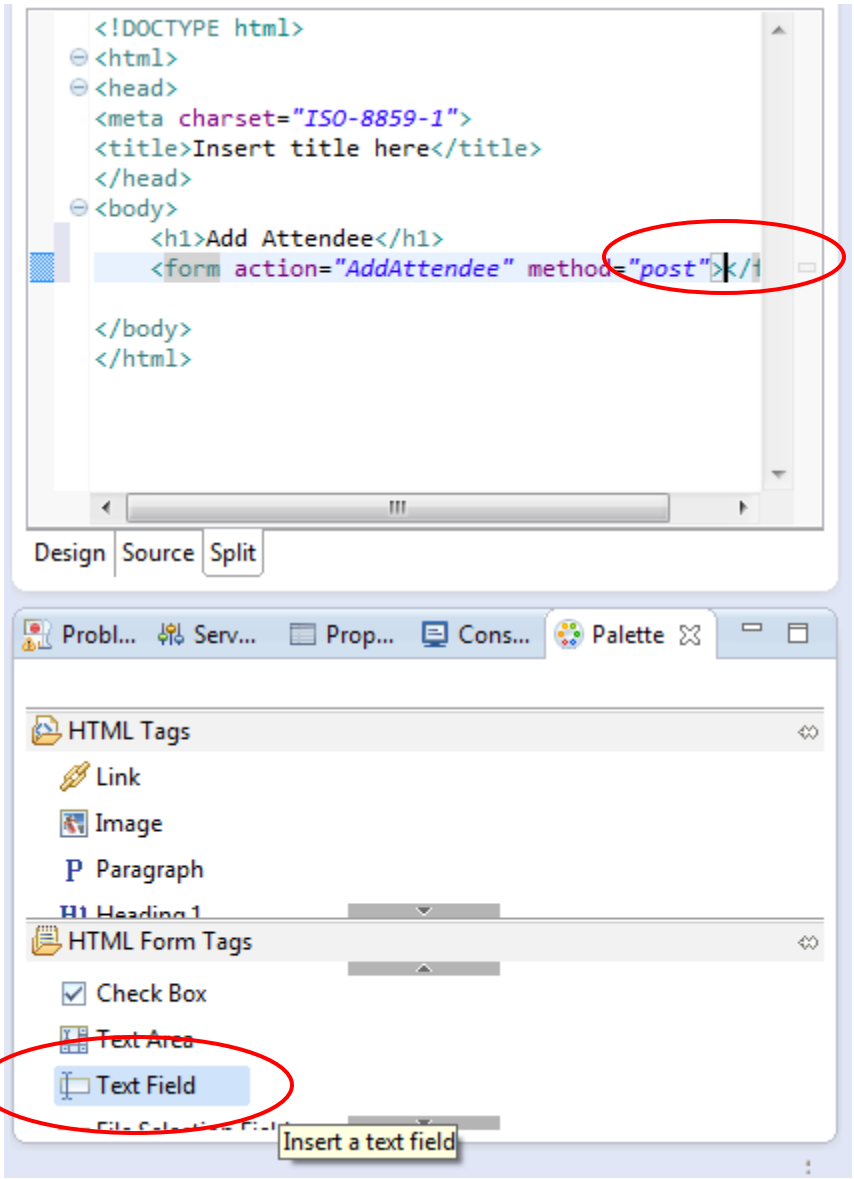
7. Drag the *form* element into the main area just below the heading. Right click on the form tag in the **Source** section, and then select **Properties**.



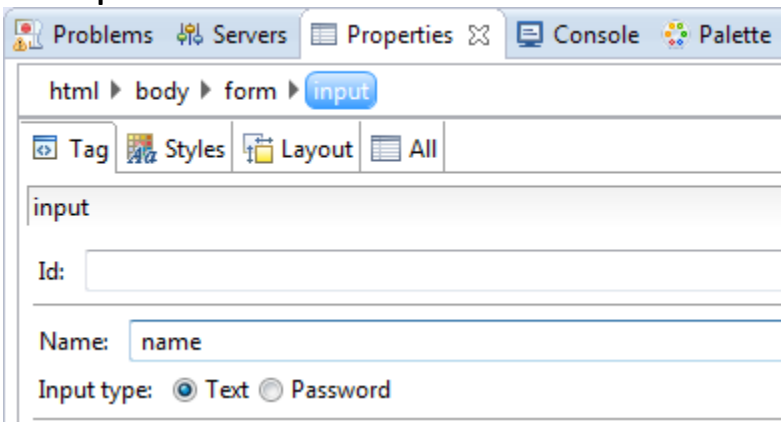
8. Set the **Action** to *AddAttendee*, and set the **Method** to *Post*.



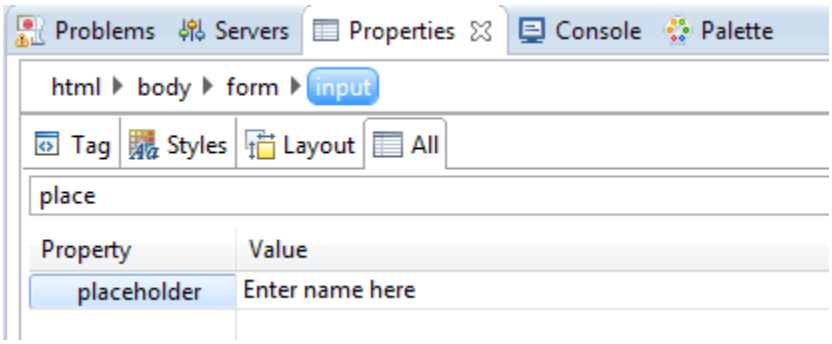
9. Switch back to the **Palette** view, and drag a *Text field* element into the **Source** view, between the *form* tags.



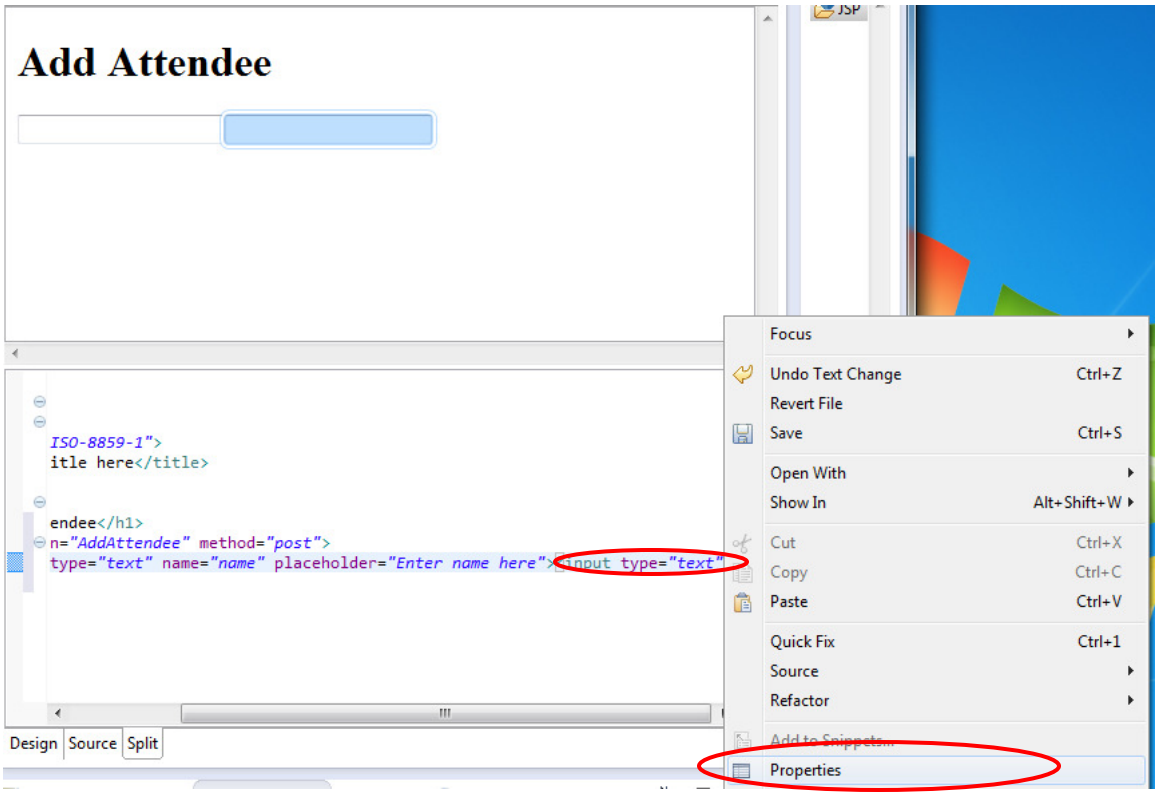
10. Right click on the input tag you just inserted in the **Source** view and select **Properties**. Set the *name* to "name"



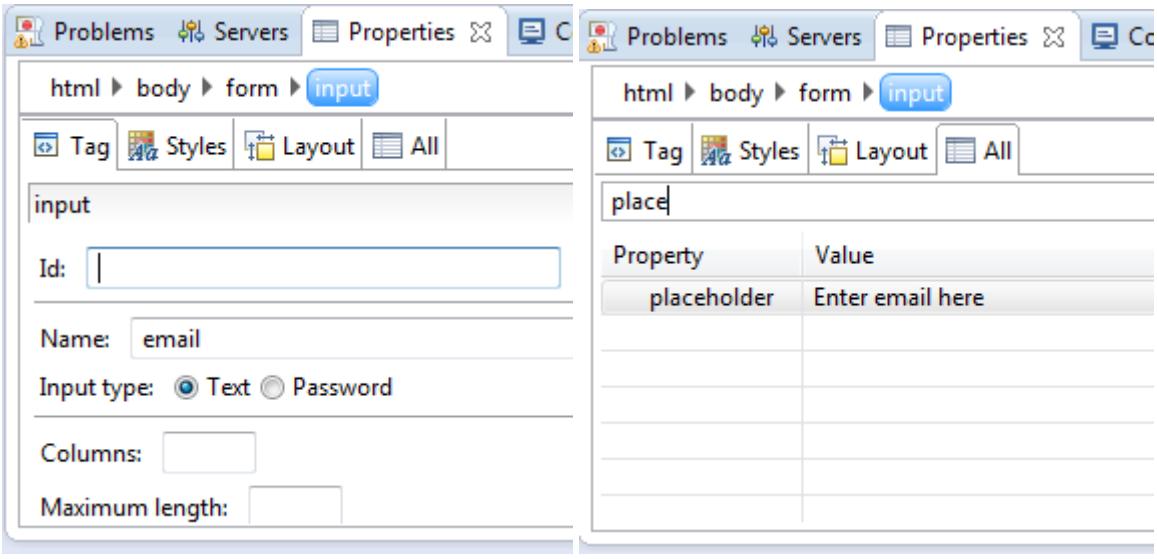
11. Under the **All** tab set the **placeholder** property to “Enter name here”.



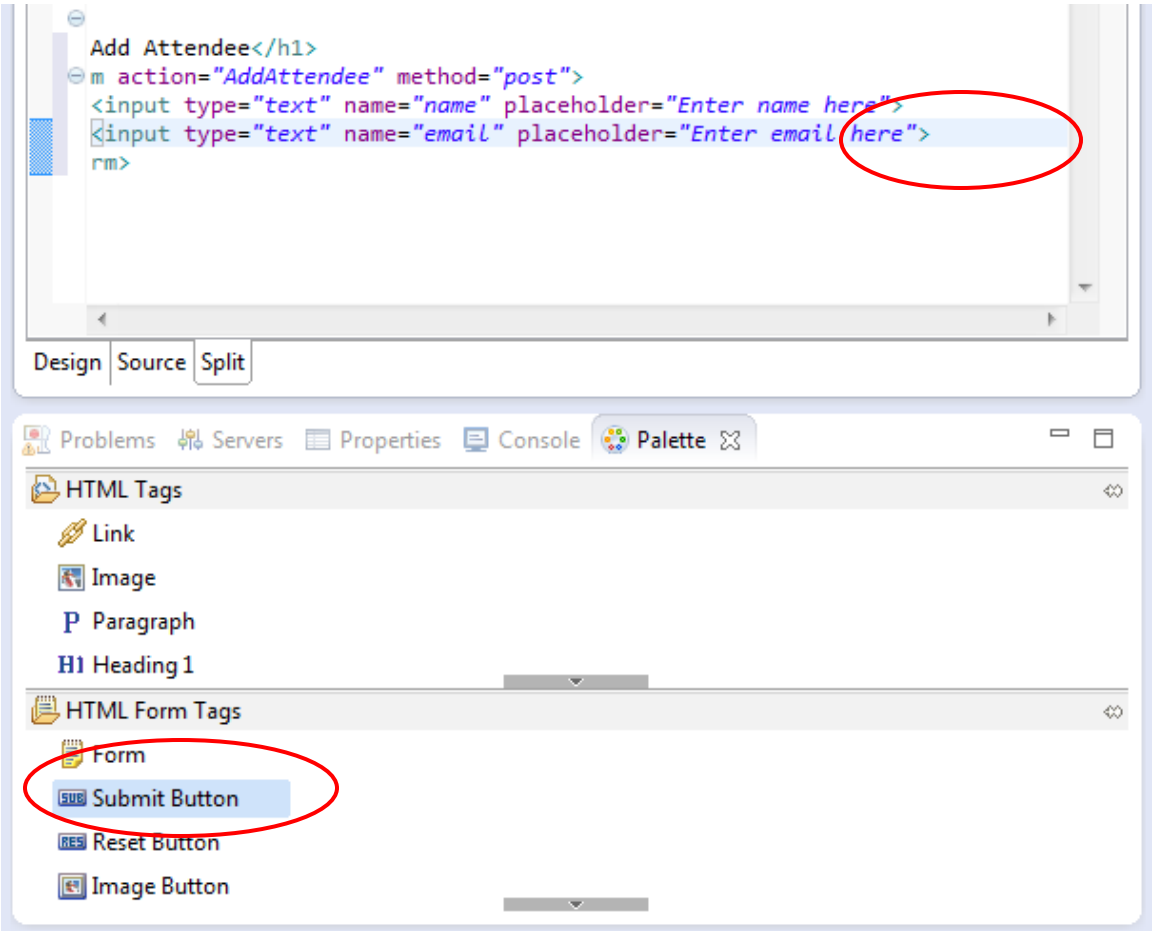
12. Drag another *Text field* into the *form*. Right click on the new input tag and select **Properties**.



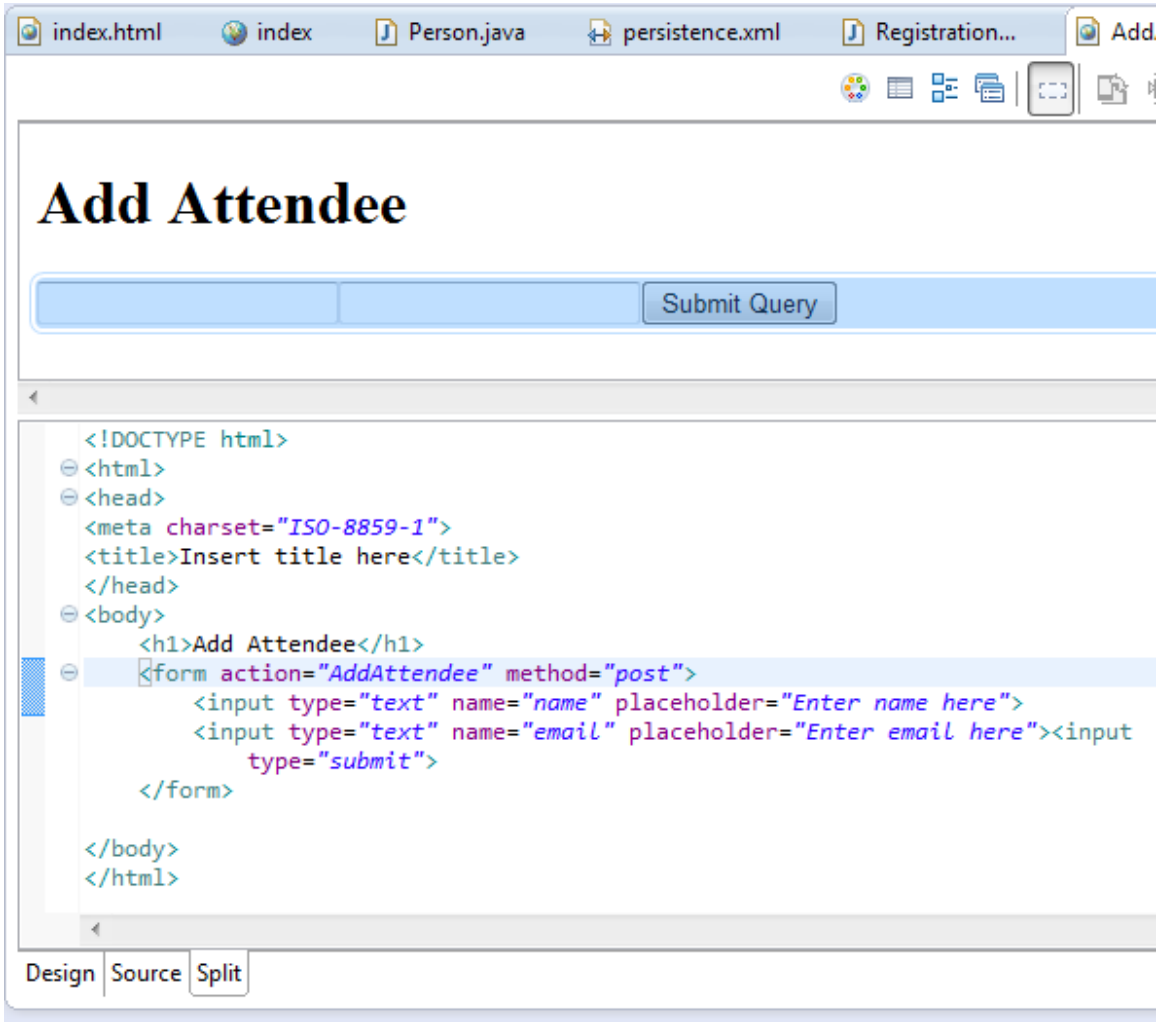
13. In the Properties view, set the **name** to “email” in the **Tag** tab, and the **placeholder** to “Enter email here”.



14. Finally, drag a *Submit button* into the *form*.



15. Use the **control-S** shortcut to save the page. The completed for should match the image below.



Update the landing page

In a previous step we created the landing page `index.html`. We now need to update this so that the pages we need to return to are easily available.

1. Expand the `WebContent` folder in the **RegistrationAppWeb** project, and double click the `index.html` file to open it.

2. Update the contents of the `<body>` tag to be

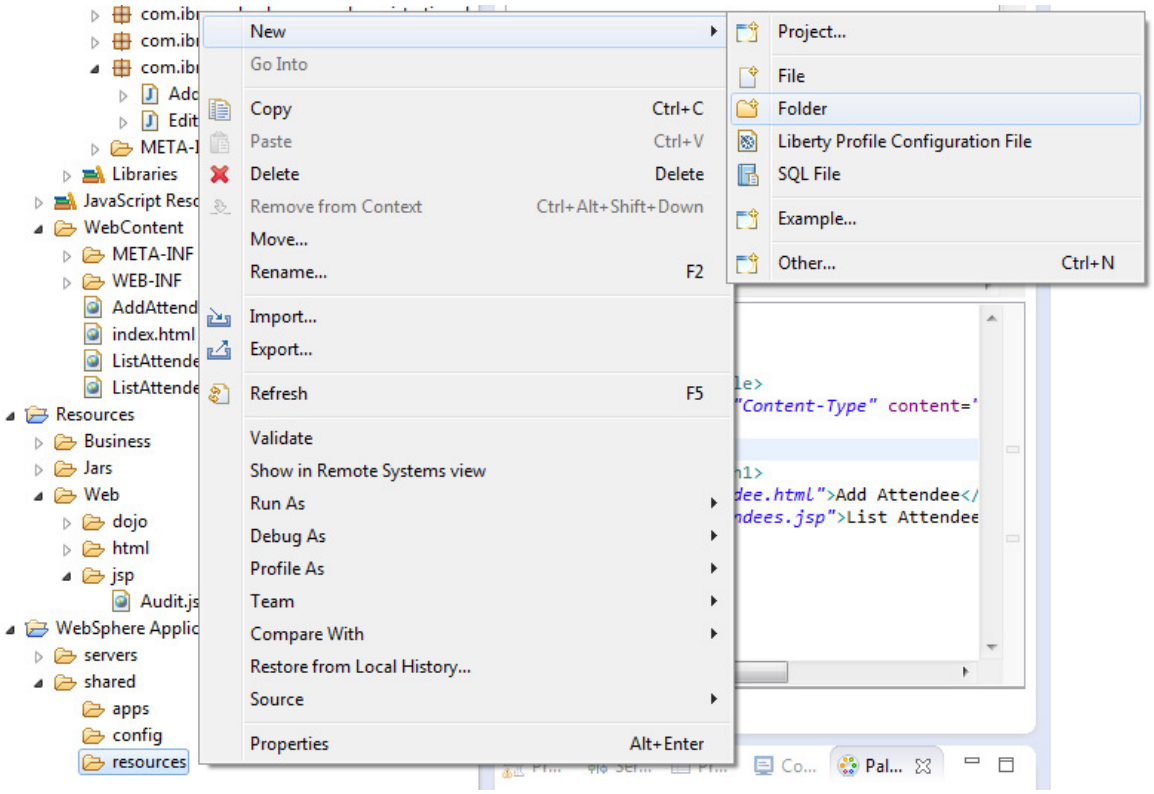
```
<h1>Hello World</h1>
<a href="AddAttendee.html">Add Attendee</a><br>
<a href="ListAttendees.jsp">List Attendees</a>
```

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

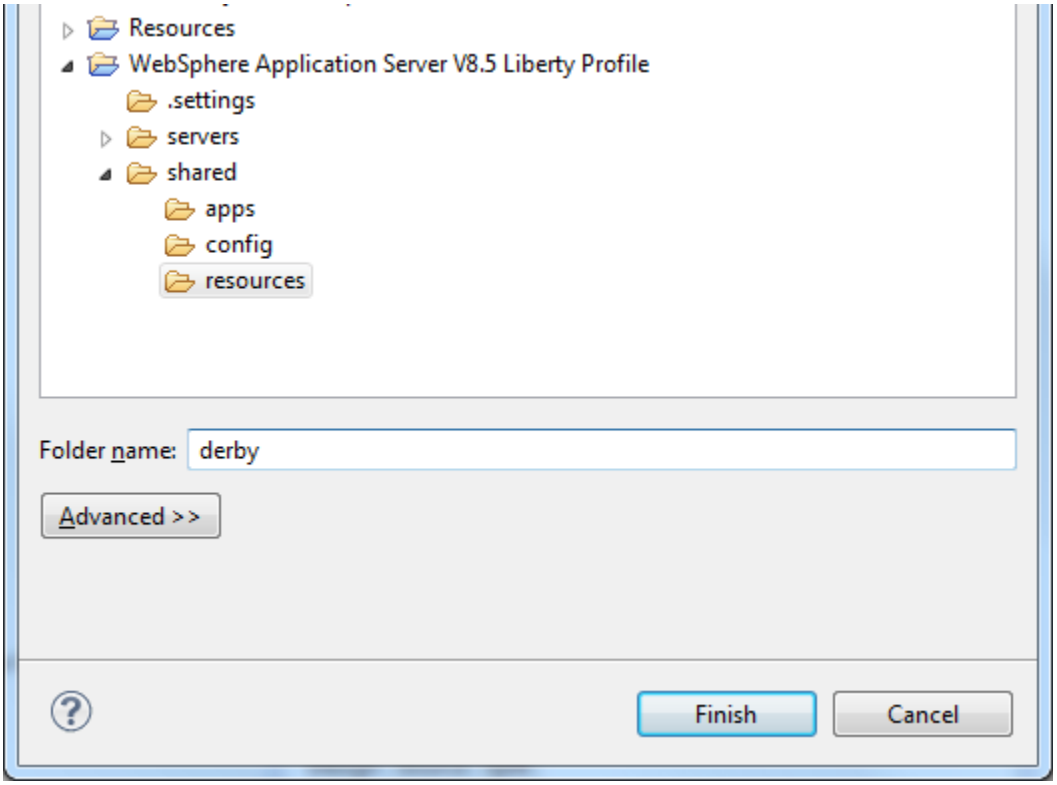
Configure the server

Now all the resources are in place to add and edit attendees. We will now configure the server. This involves adding the `derby.jar` to allow JDBC function, and configuring the server's configuration file, `server.xml`.

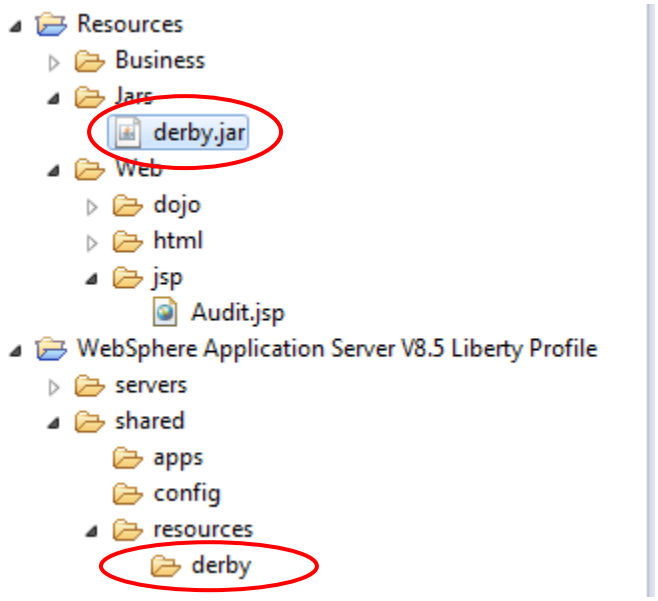
1. Open the **WebSphere Application Server V8.5 Liberty Profile** project, and expand the `shared` folder. Right click on the `resources` directory, and select **New > Folder**.



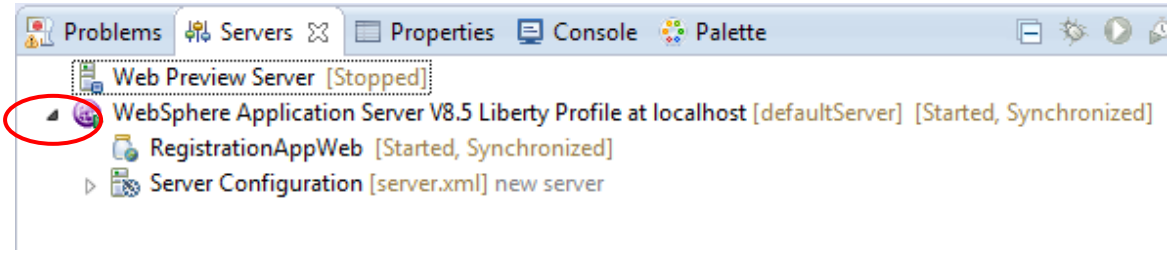
2. Call the folder `derby`. Click **Finish**.



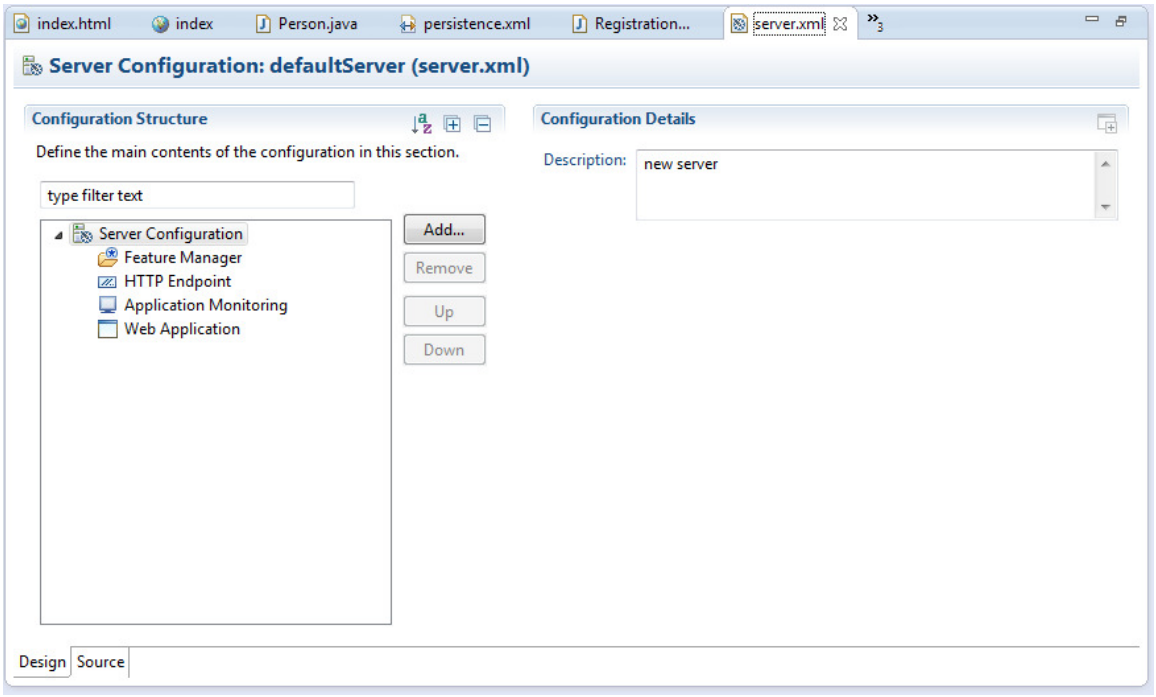
3. Open the **Resources** project and expand the `Jars` folder. Drag and drop the `derby.jar` from this folder into the `shared/resources/derby` folder you just created in the **WebSphere Application Server V8.5 Liberty Profile** project.



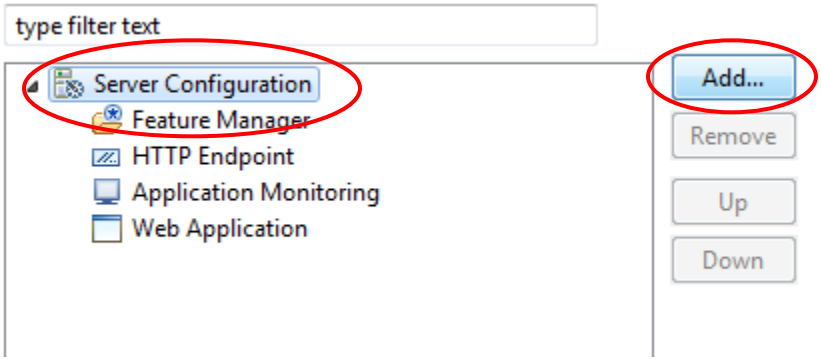
4. Open the **Servers** view and expand the **WebSphere Application Server V8.5 Liberty Profile at localhost** server definition.



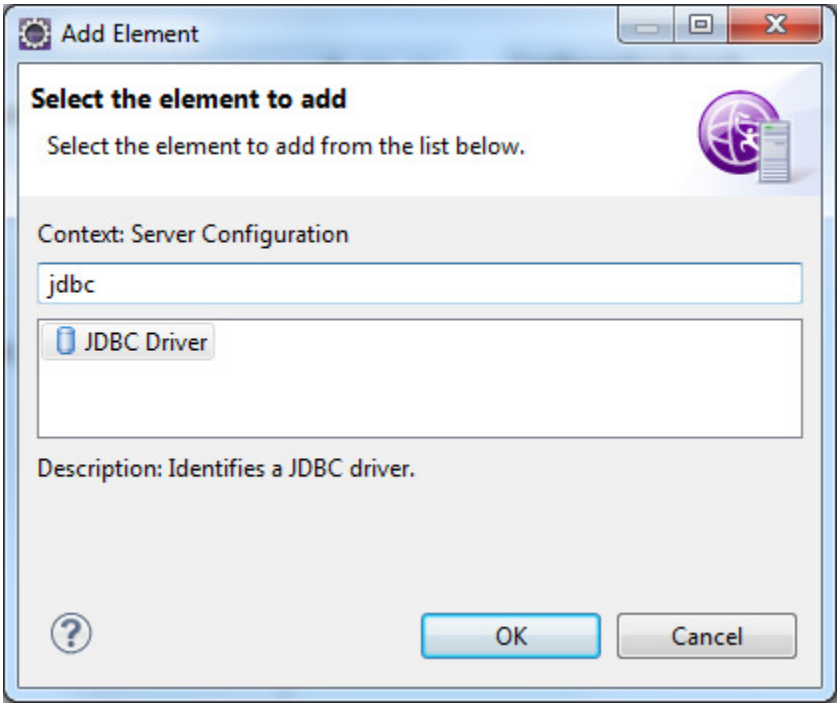
5. Double click on the **Server Configuration**. This opens the `server.xml` configuration tool.



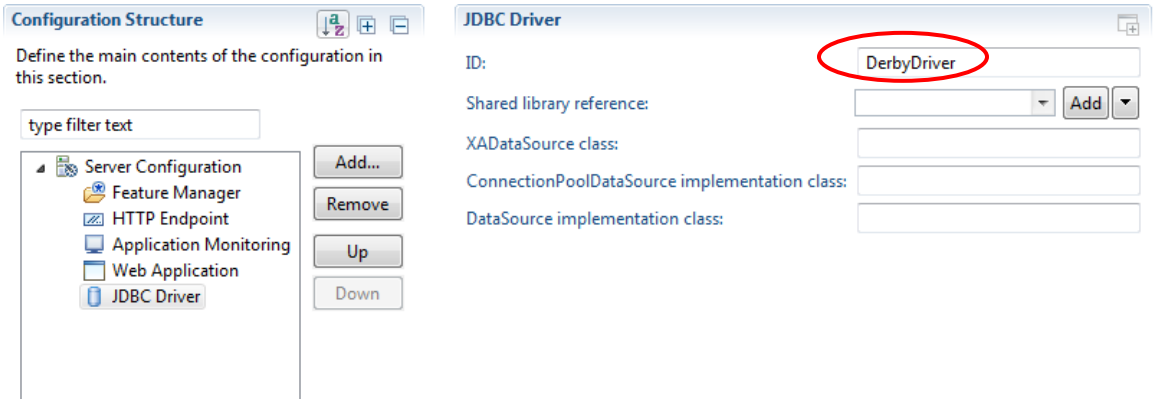
6. Click on the **Server Configuration** element, then click **Add**.



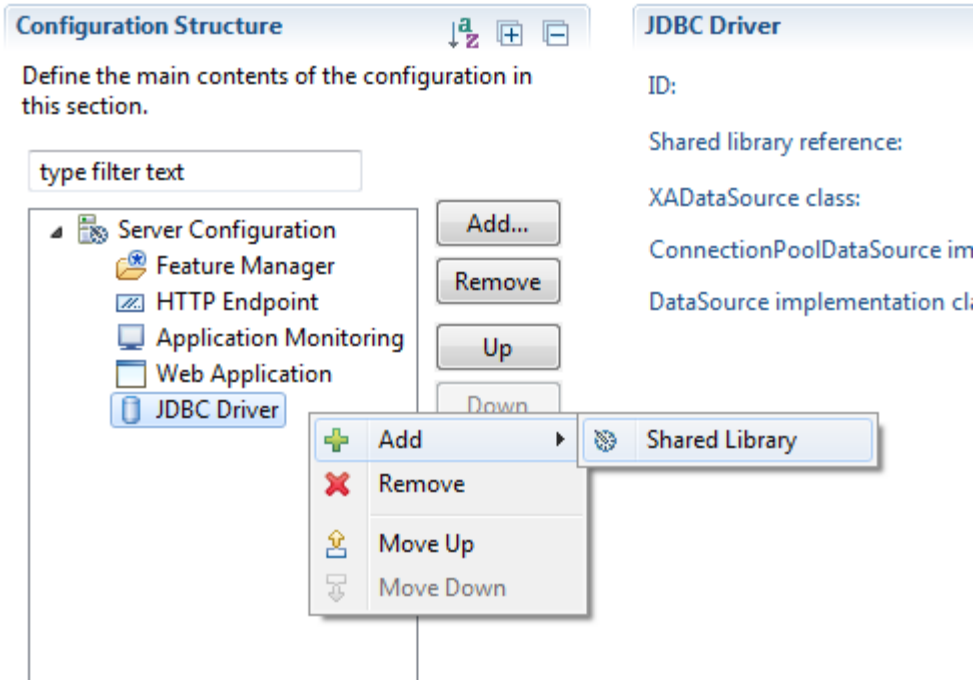
7. Using the filter box, type *JDBC*, then select the **JDBC Driver** element and click **OK**.



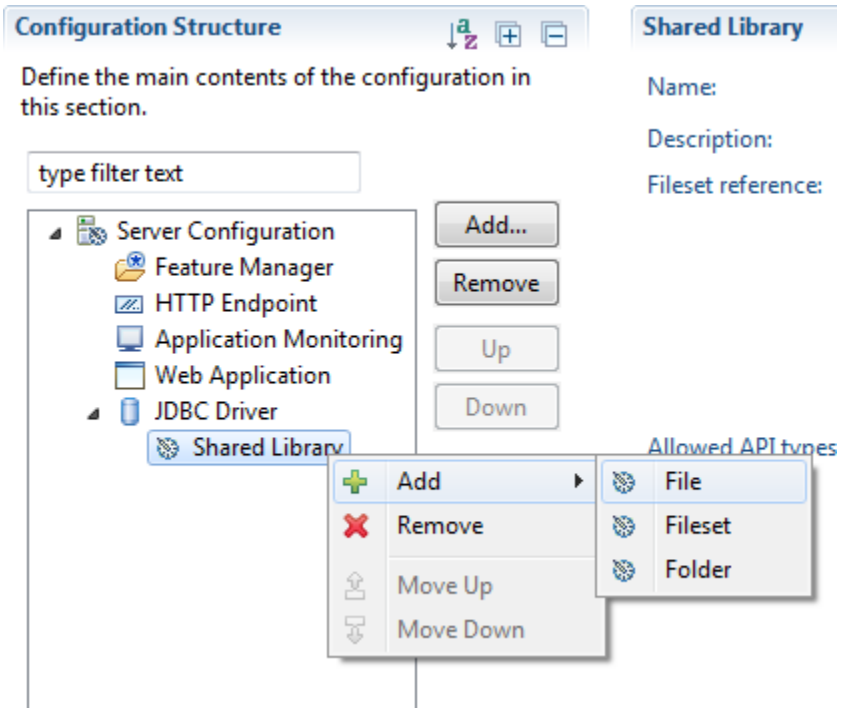
8. Set the **Id** to be *DerbyDriver*.



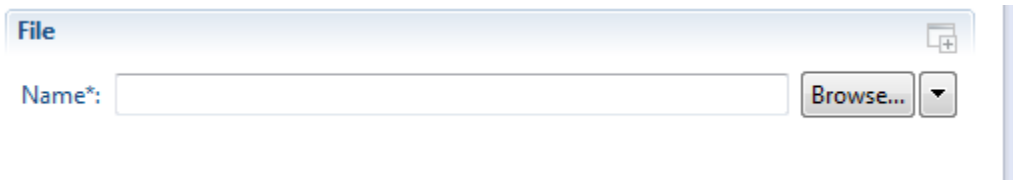
9. Right click on the **JDBC Driver** and select **Add > Shared Library**.



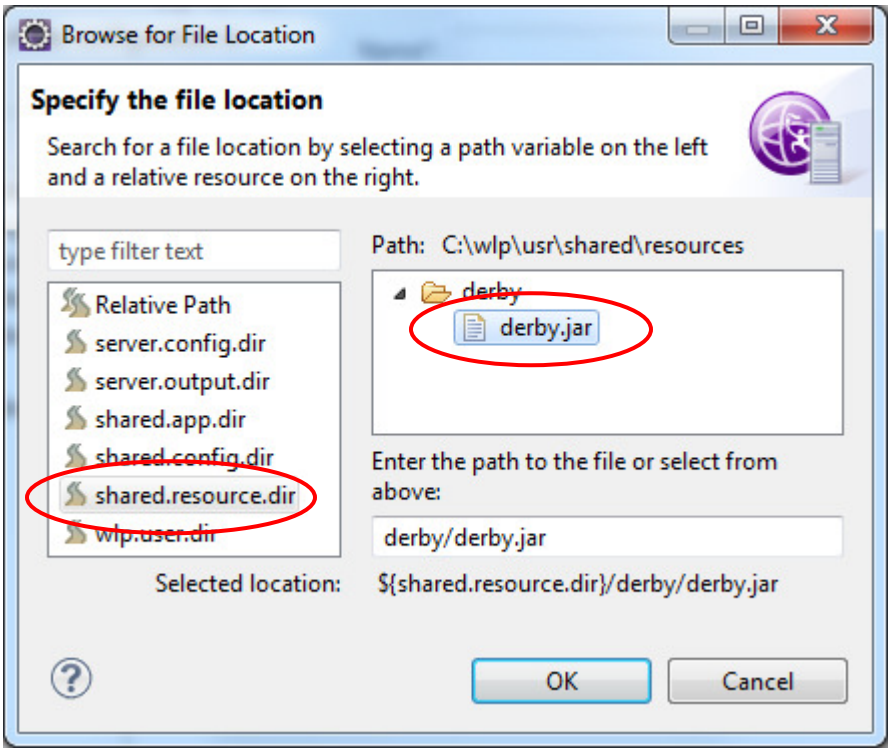
10. Right click on the **Shared Library** and select **Add > File**.



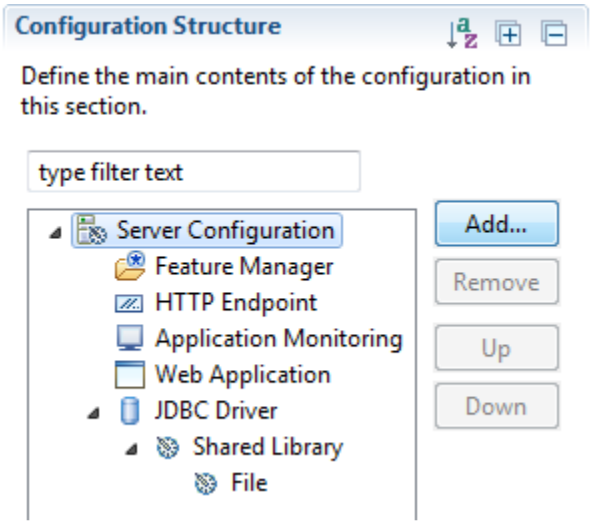
11. Click on the **File** element, and select the **Browse** button under **File**.



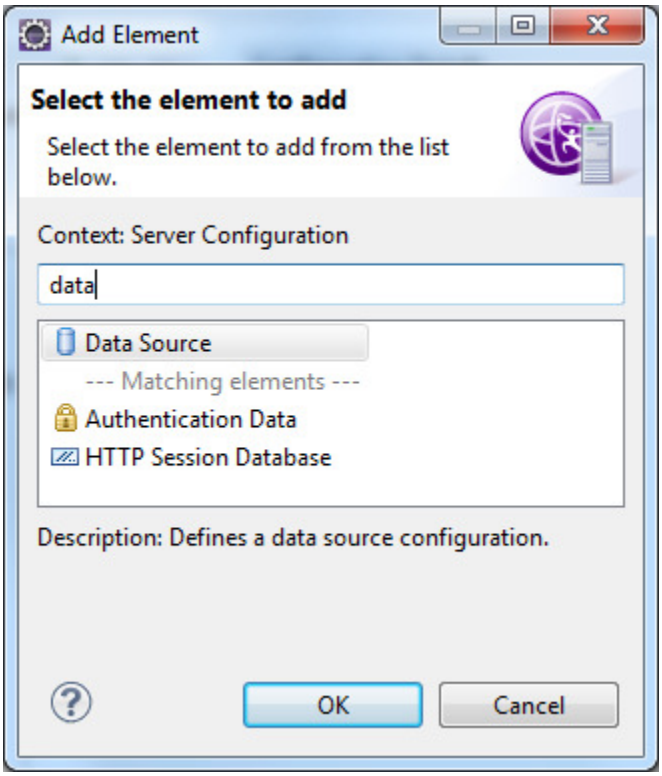
12. Select the *shared.resource.dir* from the left hand navigation pane. Expand the `derby` folder, click on the `derby.jar` and then click **OK**.



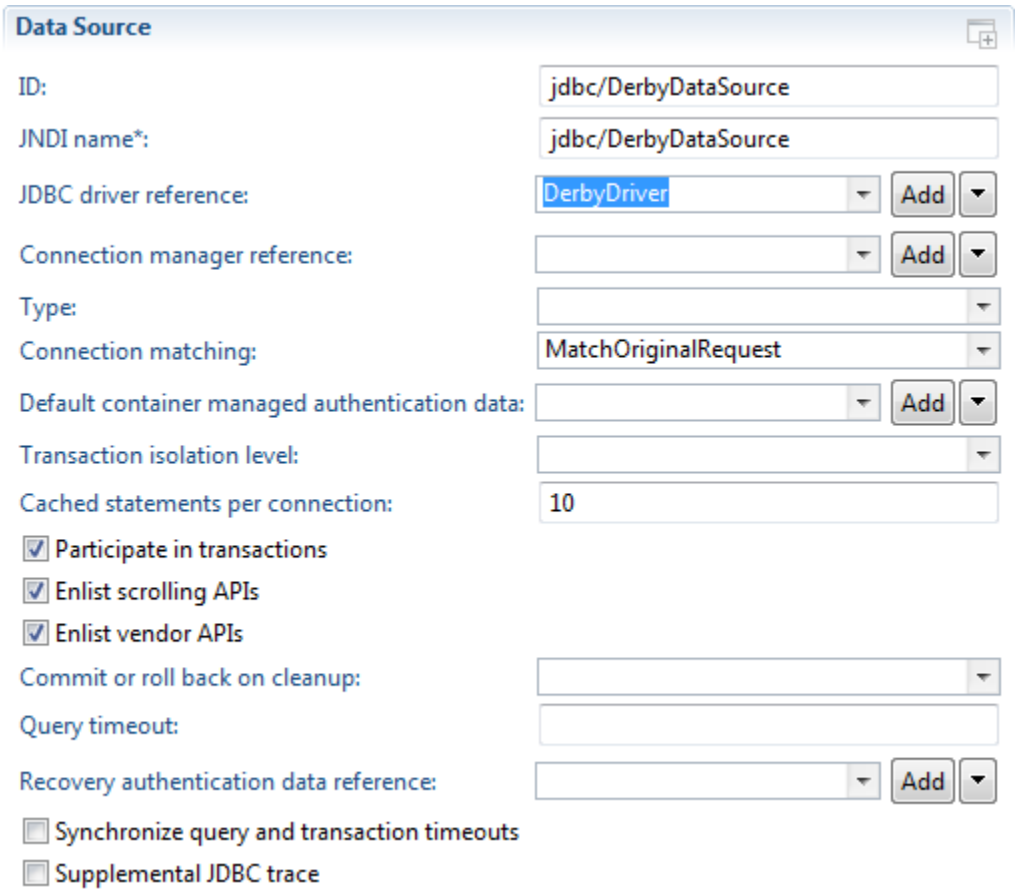
13. Click on the **Server Configuration** and click the **Add** button.



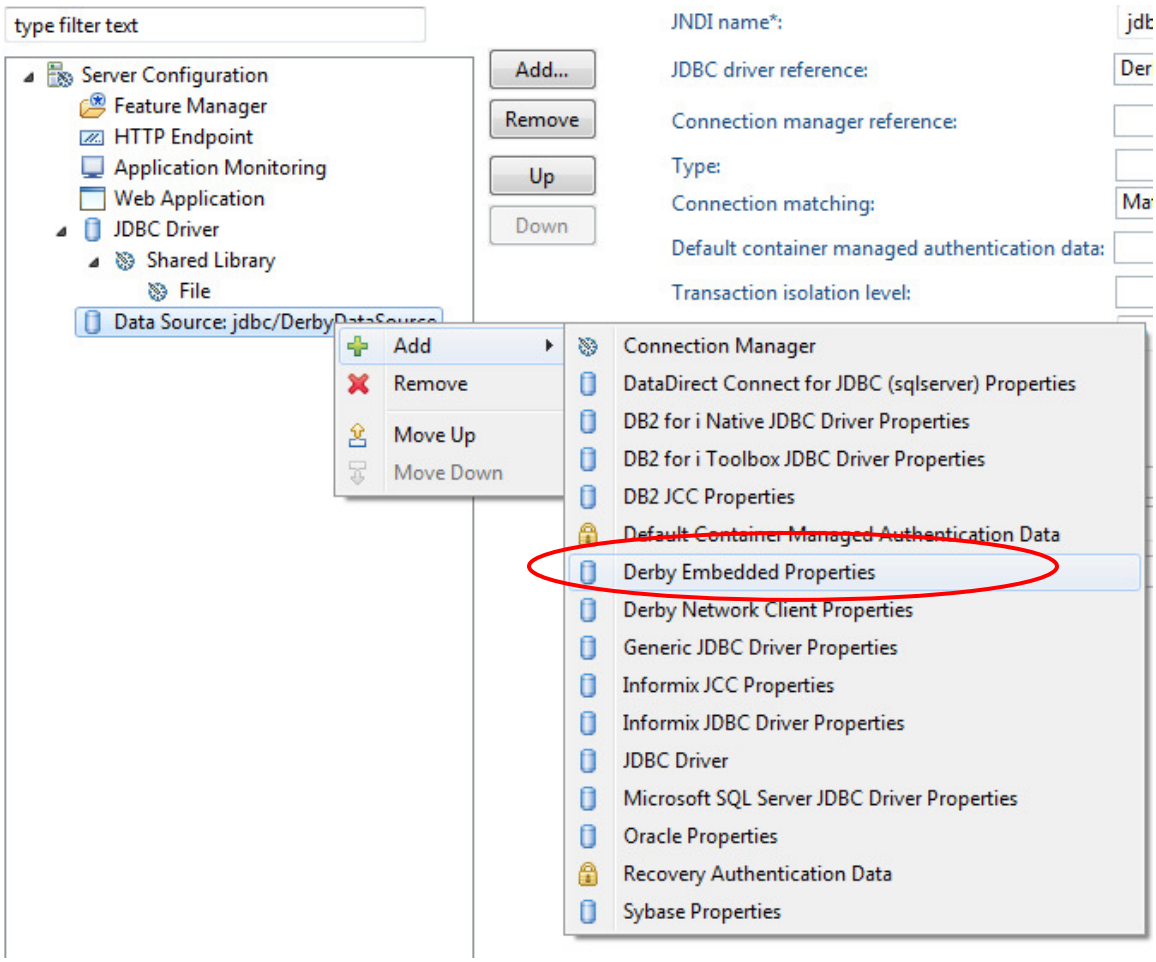
14. Using the filter, type *data* and then select the **Data Source** element and click **OK**.



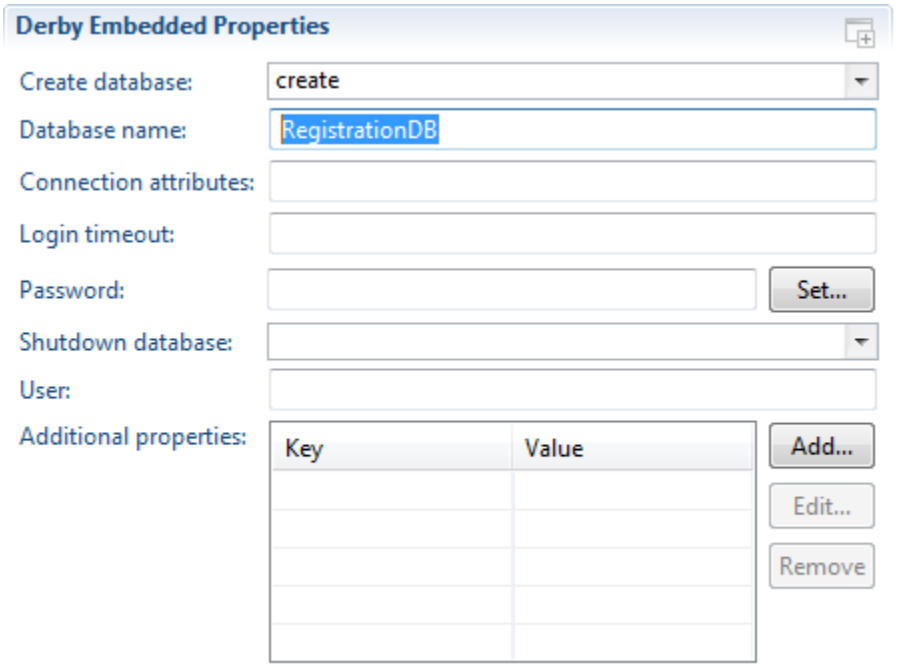
15. Set the **Id** to *jdbc/DerbyDataSource*, and the **JNDI Name** to *jdbc/DerbyDataSource*. Set the **JDBC driver** to *DerbyDriver* using the dropdown.



16. Right click on the data source you have just created and select **Add > Derby Embedded Properties**.



17. Set the **Database name** to *RegistrationDB*, and the **Create database** dropdown to *create*.

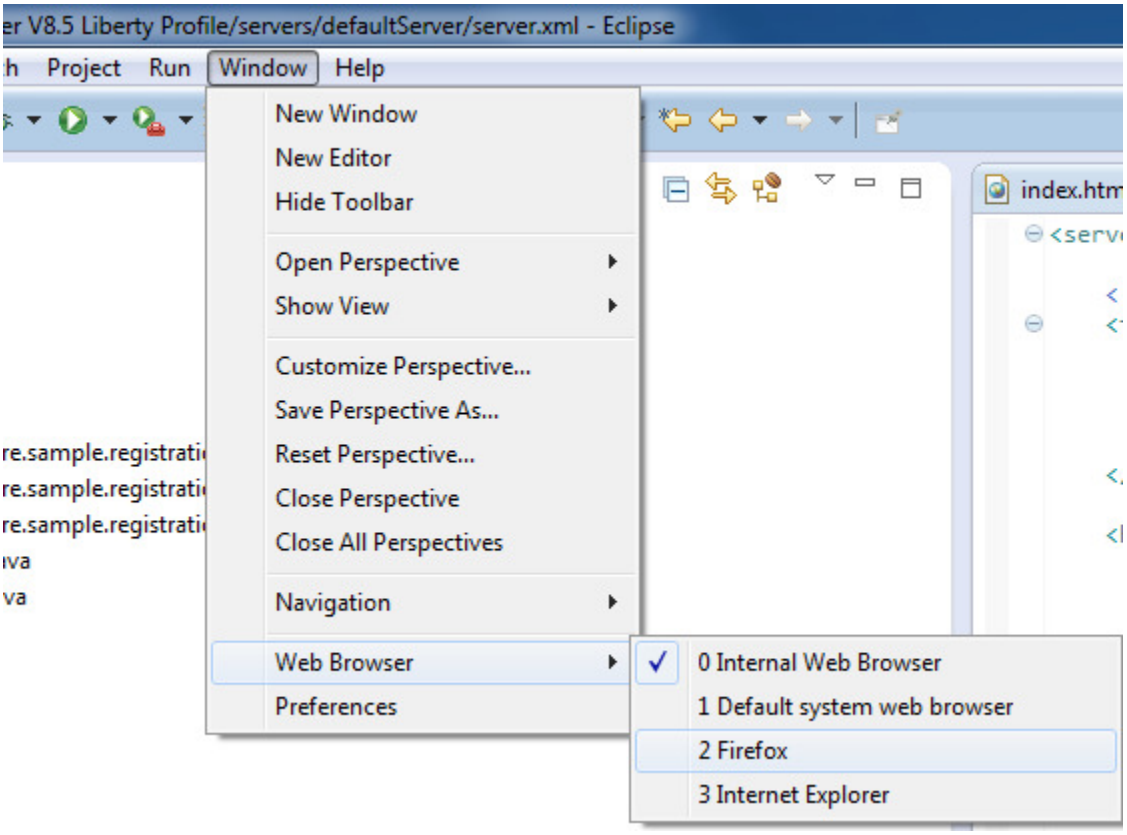


18. Save the file using the **control-S** shortcut. The configuration should resemble the image below.

Running the application

Now the application is ready, we can run it in the server and test it.

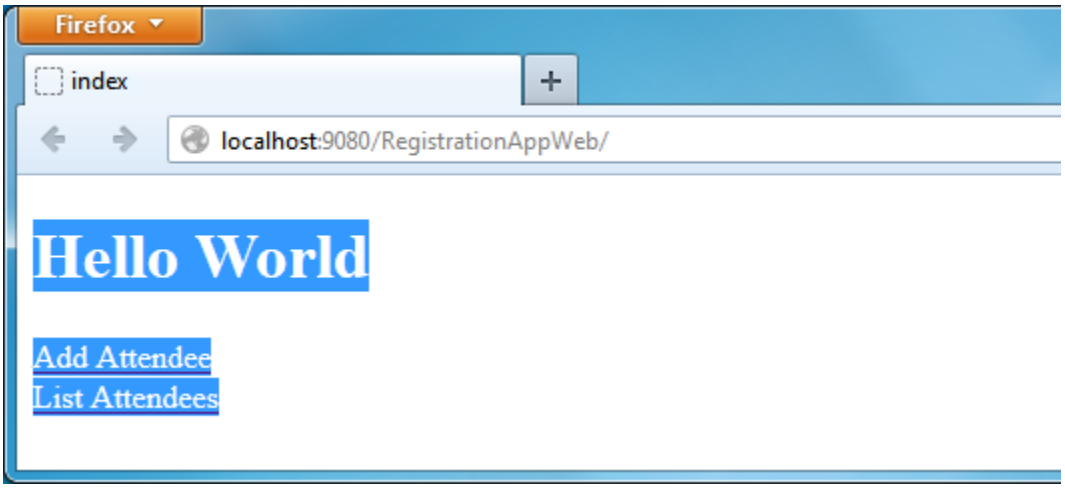
8. Go to **Window > Web browser**, and select *Firefox* from the list of options. This redirects any web pages opened in Eclipse to the Firefox web browser instead, which is required for some of the technologies being used in this lab.



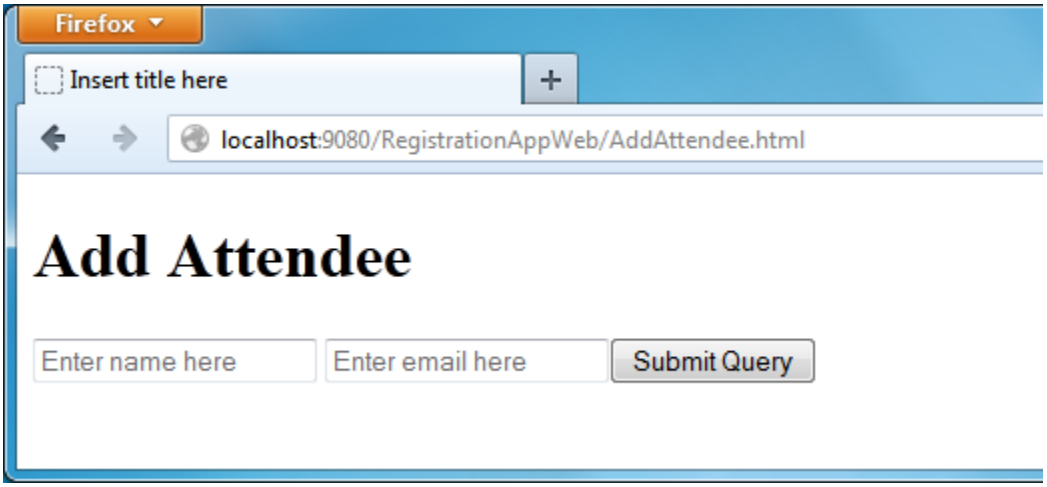
9. Go to the **Console** view. The application will be started, and you should have a line like this:

   [AUDIT] CWWKT0016I: Web application available (default_host): http://localhost:9080/RegistrationAppWeb/

10. Click the hyperlink in the **Console** view to open the landing page of the application.

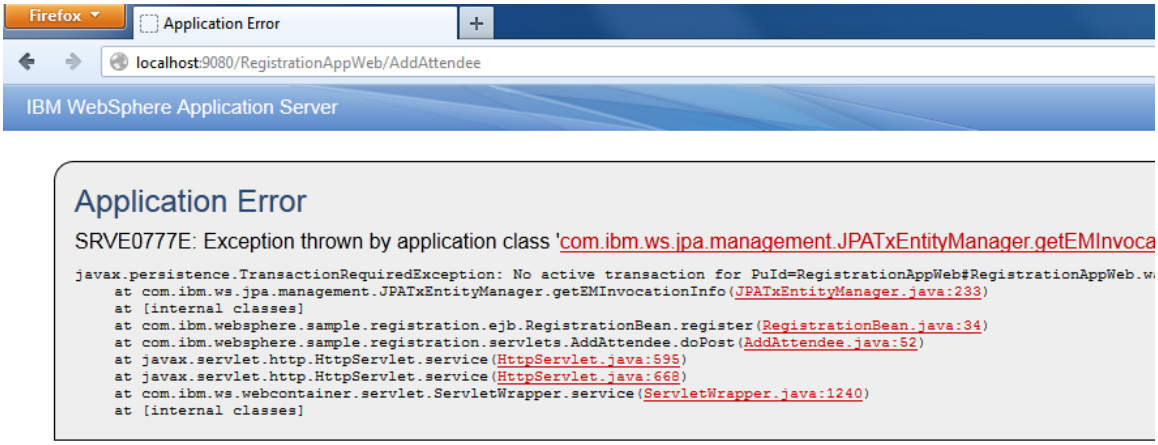


11. To add an attendee, click the **AddAttendee** link on the landing page.



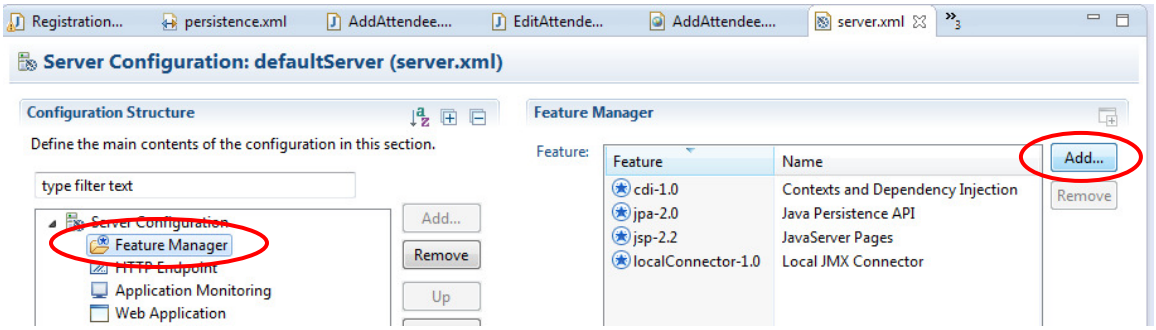
Note that the placeholder text may not appear, depending on the browser used.

12. Provide an email address and name for the user, then click **submit**. This will take you to the `ListAttendee.jsp` page, but display an error message.

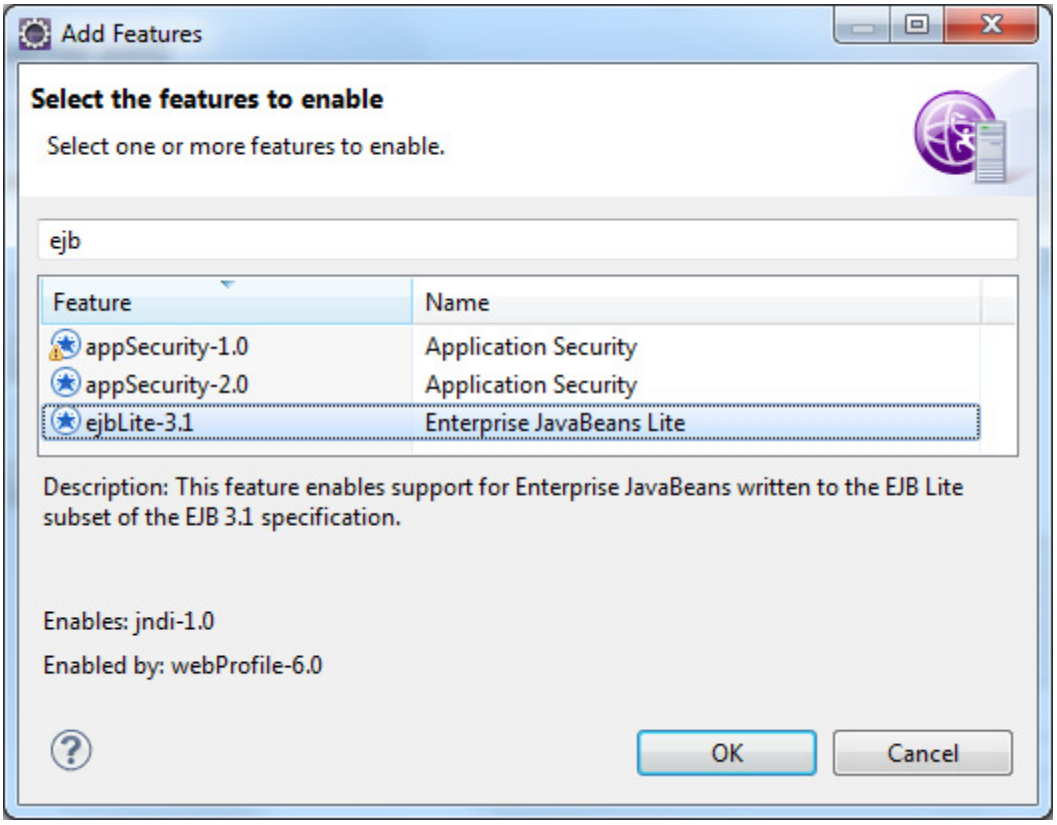


13. This is because the `EJBLite` feature is not enabled, meaning your EJB is not being run correctly. To fix this we need to enable the missing feature in `server.xml`. Open `server.xml`.

14. Click on the Feature Manager section under the server Configuration, then click the Add… button under the detailed Feature Manager panel.



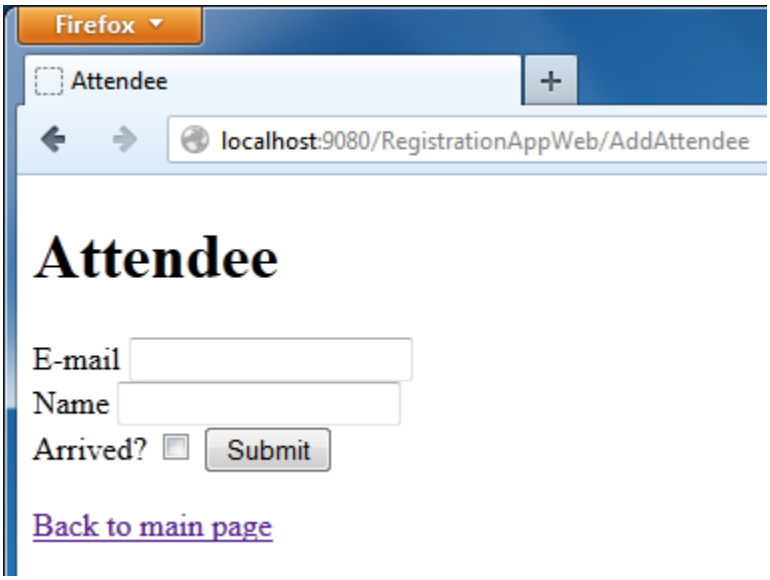
15. From the dialog, type *ejb* into the filter box. Then, select the **ejbLite-3.1** feature, and then click **OK**.



16. Save the file using the **control-S** shortcut.

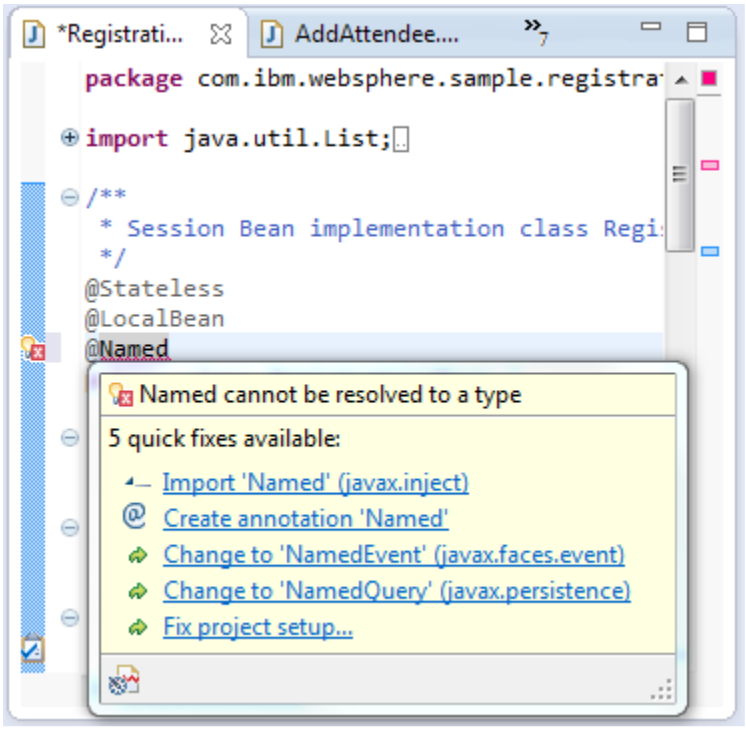
17. Reopen the homepage by clicking on the link in the console.

18. Try adding a new Attendee again. This time it works, but the page you are taken to shows empty fields.



19. This is because the `RegistrationBean` is not marked as `@Named`, and so is not being managed correctly. Open the `RegistrationBean` class.

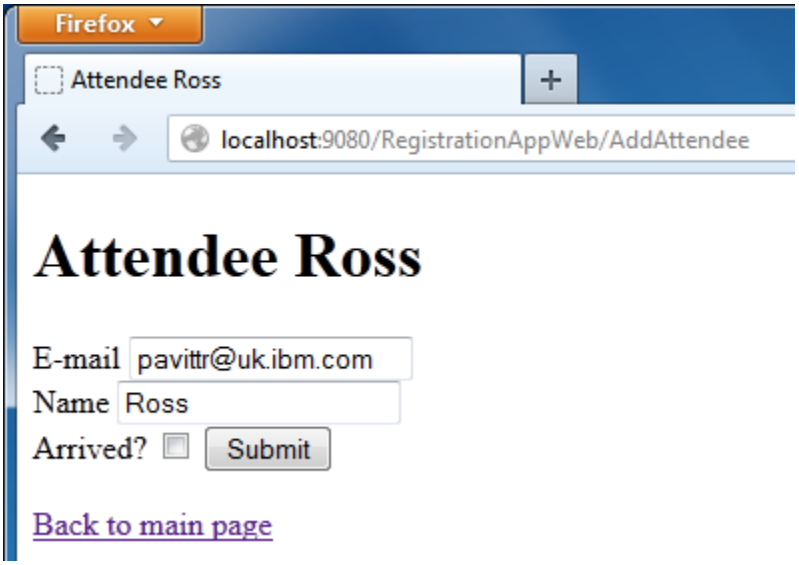
20. Between the **@LocalBean** annotation and the class definition line, add the `@Named` annotation. To import this annotation, hover over it and select **Import 'Named' (javax.inject)** from the dialog.



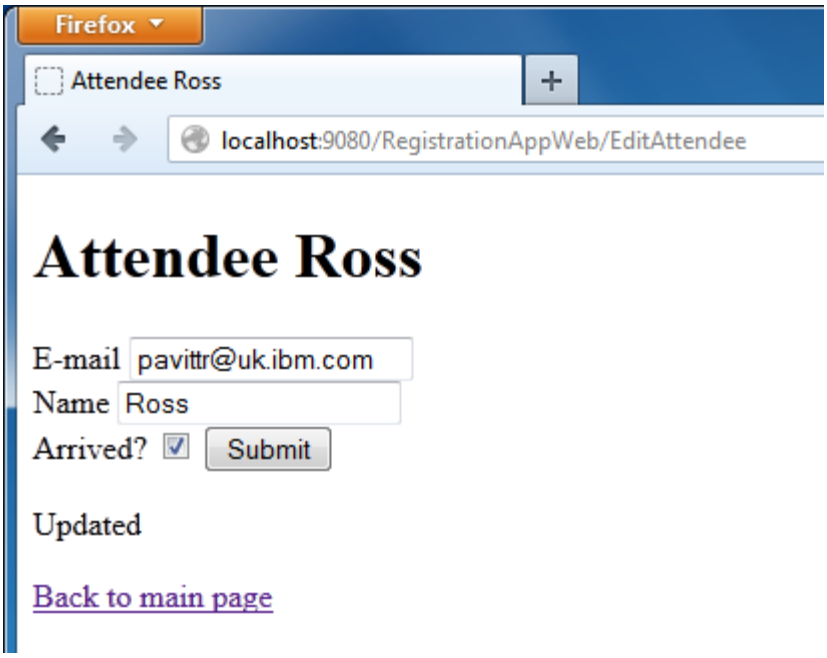
21. Save the file using the **control-S** shortcut.

22. Return to the landing page, and click the **Add Attendee** link again.

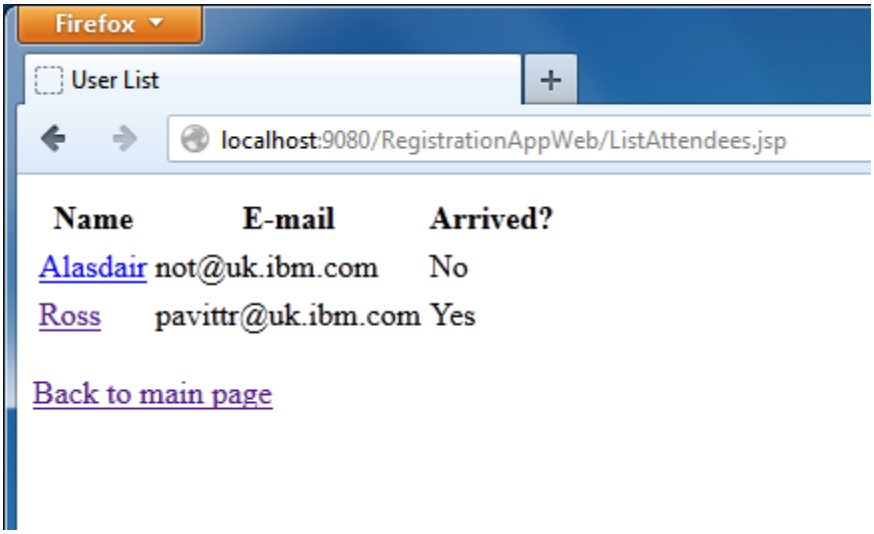
23. Fill out the details of a **different** Attendee, and click **Submit**. This will take you to the **List Attendee** page, where you can update details.



24. Mark the attendee as arrived, and click **Submit**. You will get a message that says "Updated", and the checkbox will remained checked.



25. Return to the landing page and click the **List Attendees** link. This shows a list of users you have added and their current state of attendance.



Summary

In this section you learned:

- How to create and deploy an EJB
- How to enable CDI
- How to configure the Liberty profile server for JPA

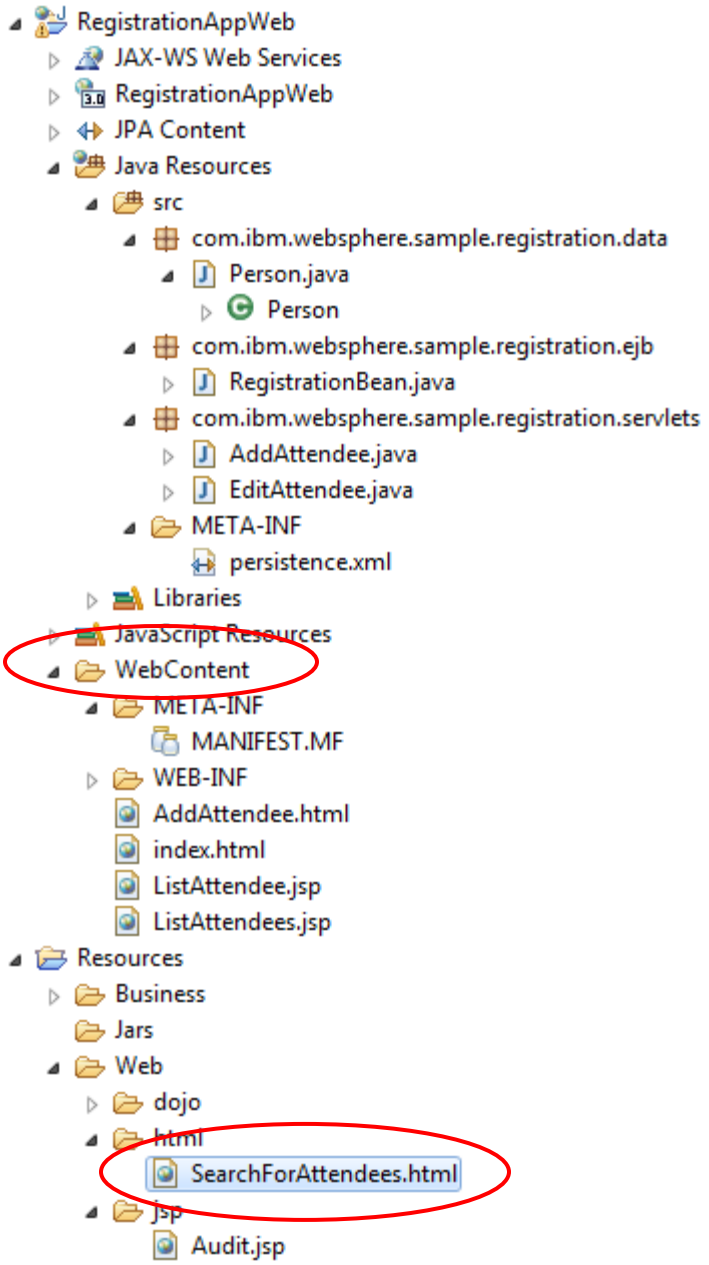
Adding RESTful services using JAX-RS

JAX-RS allows you to write a RESTful interface that uses POJOs to communicate to clients. Web requests are made to classes with annotations that identify them as handling different HTTP requests. We will use JAX-RS here to enable a search facility.

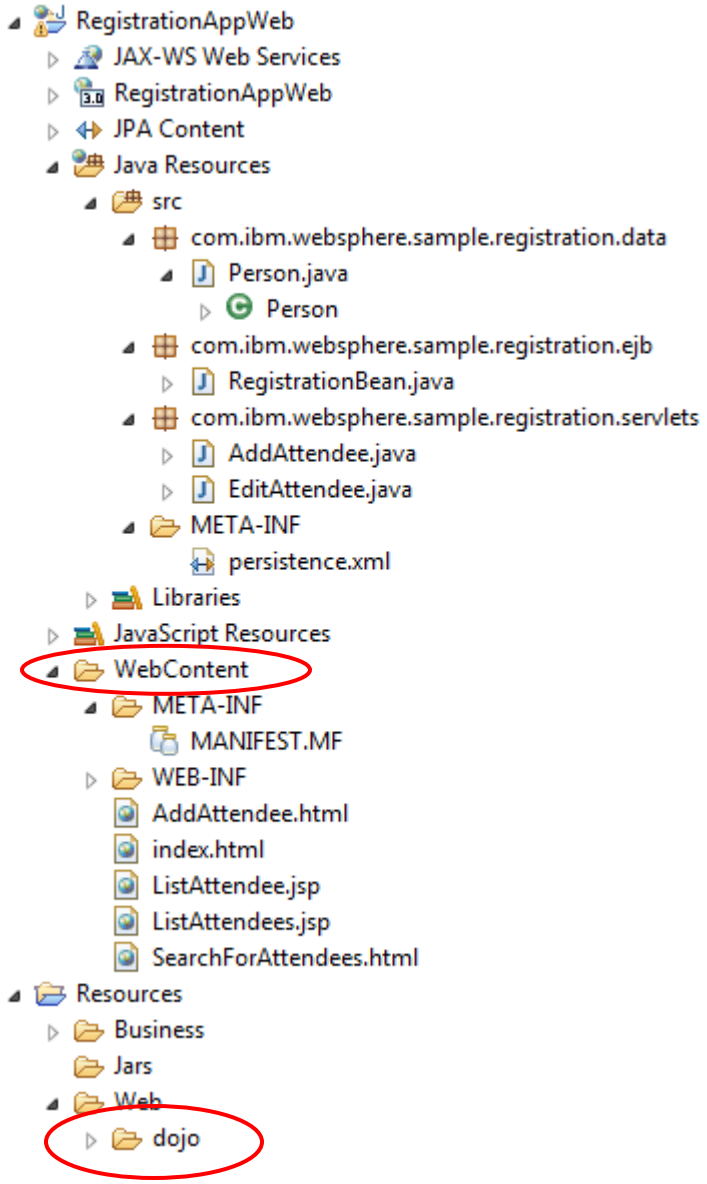
Adding the client-side capability

To enable searching we are going to use client-side technologies such as Dojo. The **Resources** project contains Dojo modules, plus some modules in the registration folder that are used to send requests to a web server and receive responses. These resources will be added to the Web project so they form part of our application.

1. Copy the `SearchForAttendees.html` file from the `Web/html` folder of the **Resources** project into the `WebContent` folder of the **RegistrationAppWeb** project.



2. Copy the `dojo` folder from the `Web` folder of the **Resources** project by dragging it into the `WebContent` directory of the **RegistrationAppWeb** project.



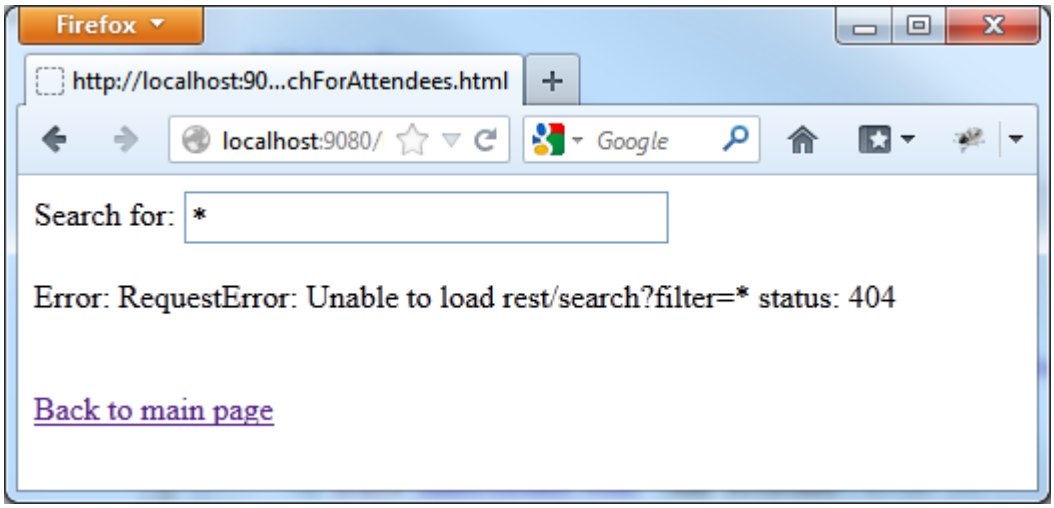
3. Open the `index.html` file in the **RegistrationAppWeb** project, and update the `<body>` tag to be:

```
<h1>Hello World</h1>
<a href="AddAttendee.html">Add Attendee</a><br />
<a href="ListAttendees.jsp">List Attendees</a><br />
<a href="SearchForAttendees.html">Search For
Attendees</a><br />
```

4. Save the file using the **control-S** shortcut.

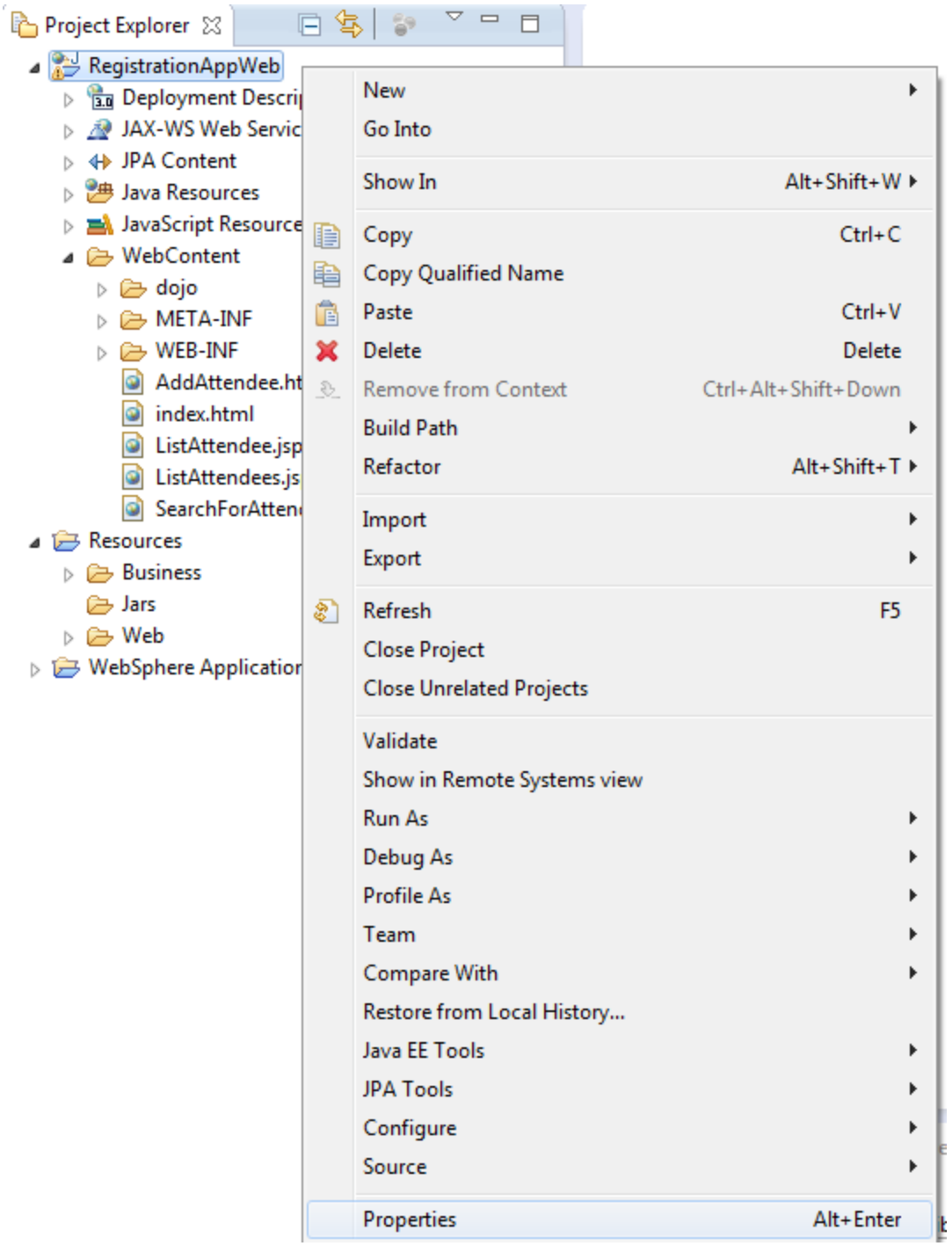
Adding JAX-RS to the Web Project

The `SearchForAttendees.html` page is now available. However, when you attempt to search, you will receive an error like:

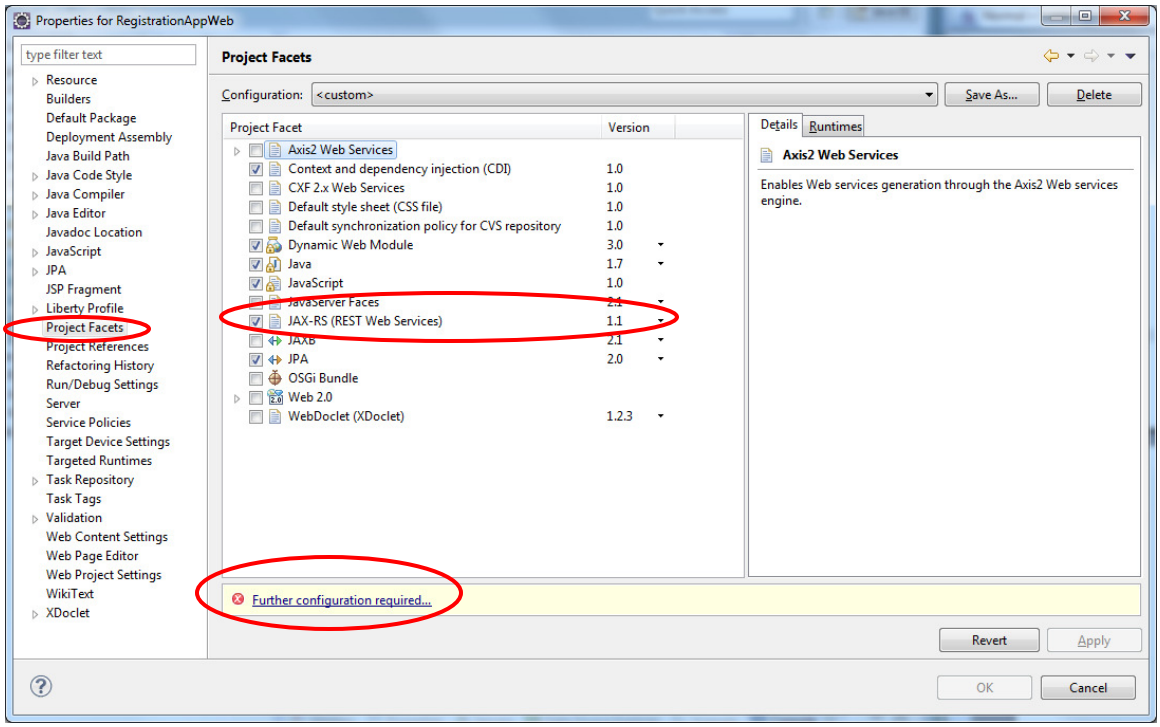


We are going to solve this by providing a JAX-RS class that will answer these requests.

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

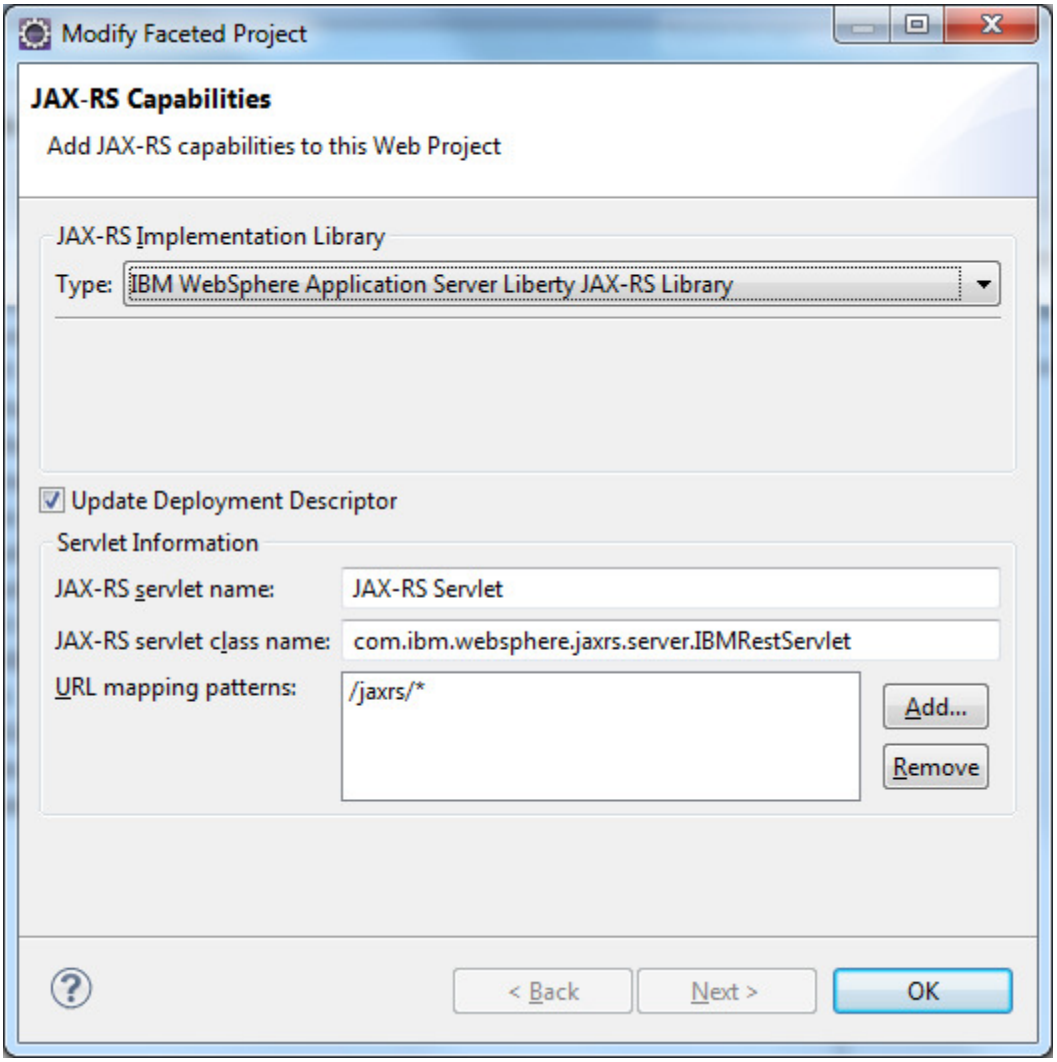


2. On the **Project Facets**, check the checkbox for **JAX-RS (REST Web Services)**.

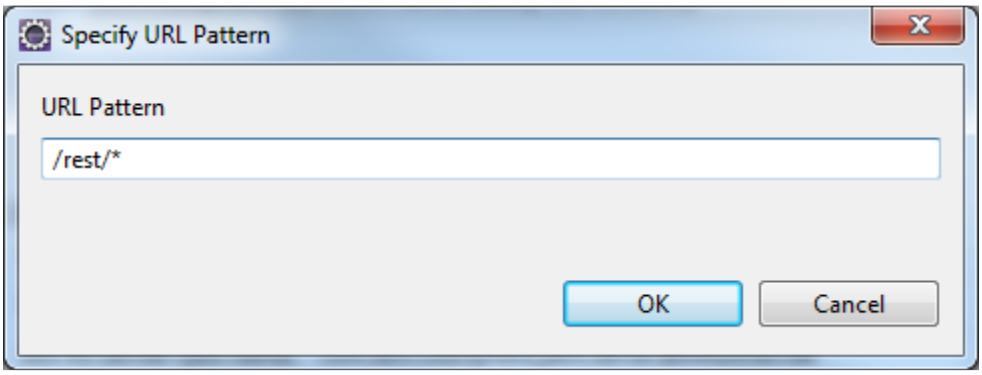


3. Click the link at the bottom labelled **Further configuration required…**.

4. Click **Add**.

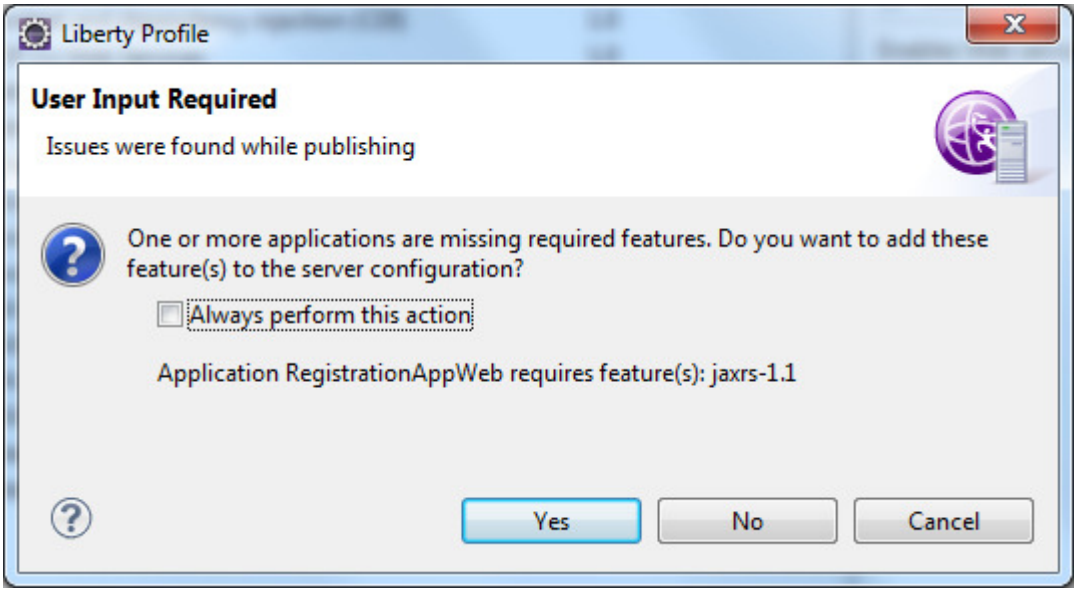


5. Specify the **URL Pattern** as /rest/* and click **OK**.



6. Click **OK** to exit the **Modify Faceted Project** dialog.

7. Click **Apply**, you are prompted to add the `jaxrs-1.1` feature. Click **Yes**.

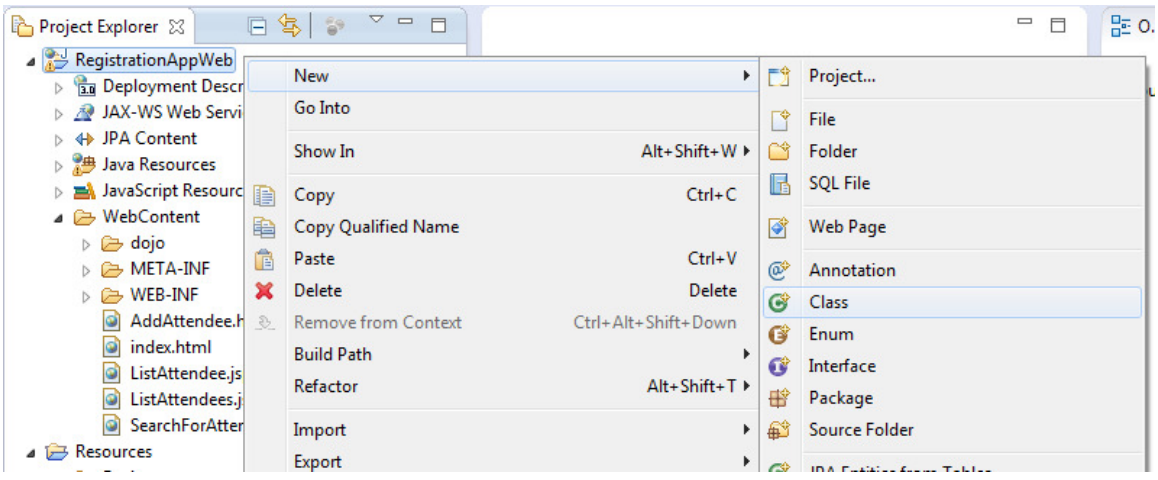


8. Click **OK**.

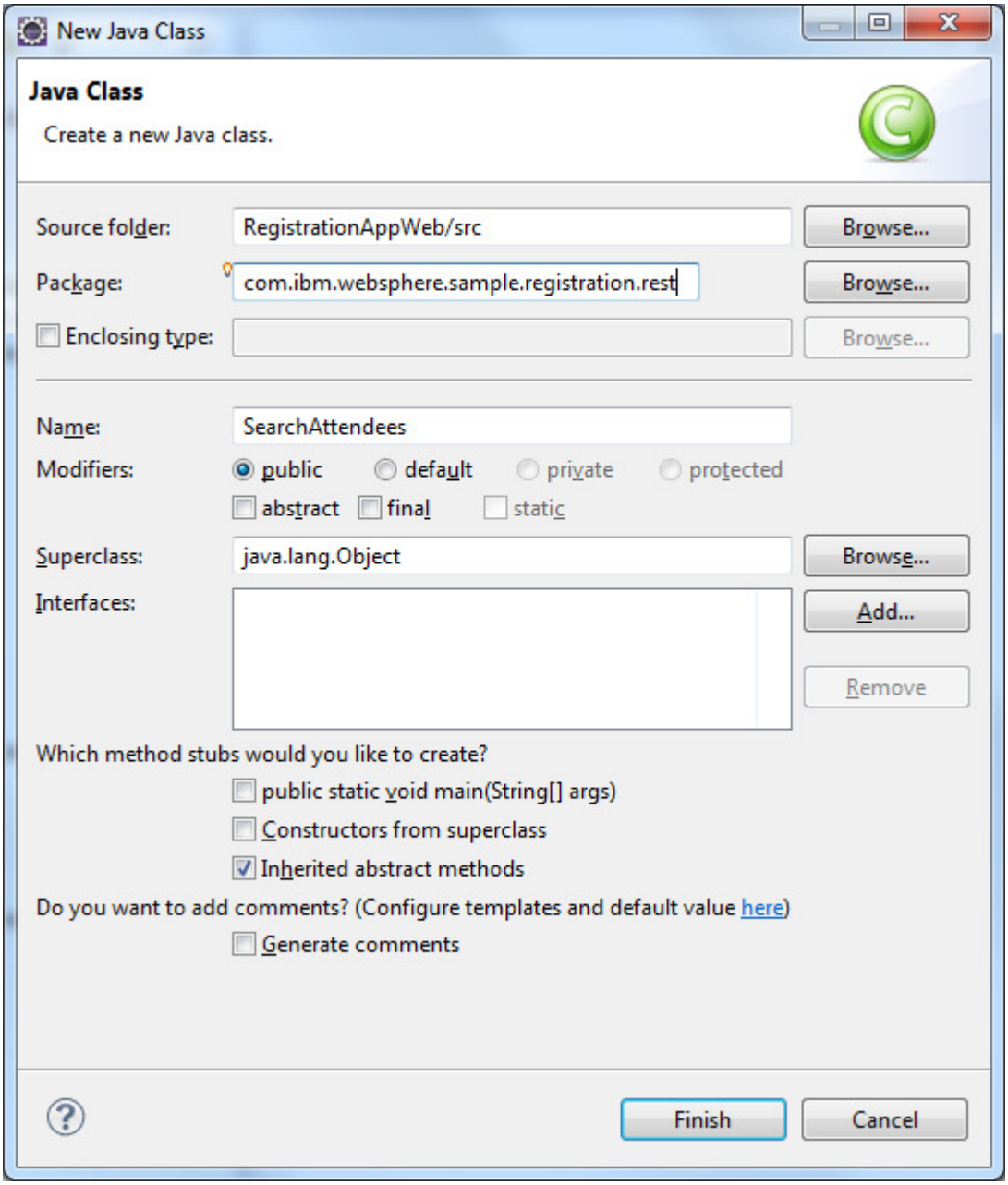
Creating the JAX-RS application

The JAX-RS application consists of a class that handles making the application available, a class that performs the request handling, and some configuration in the `web.xml` file. Adding the JAX-RS project facet provides most of the `web.xml` configuration, however we still need to complete that and also add the other classes.

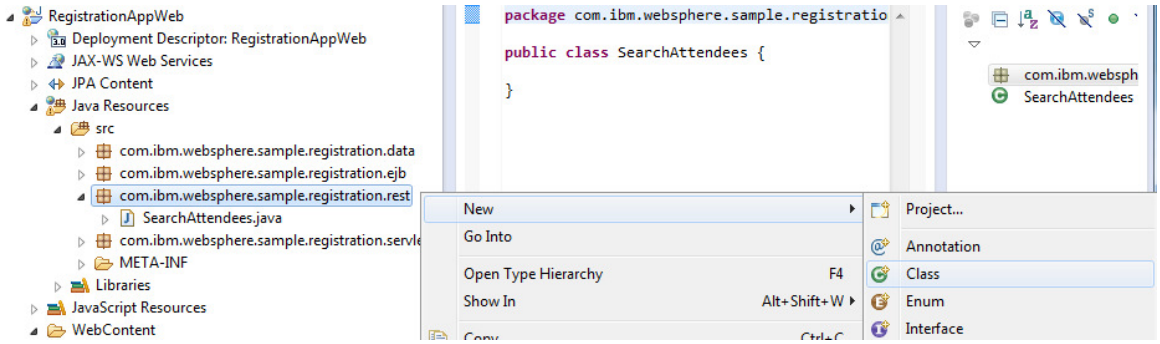
1. Right click on the **RegistrationAppWeb** project and select **New > Class**.



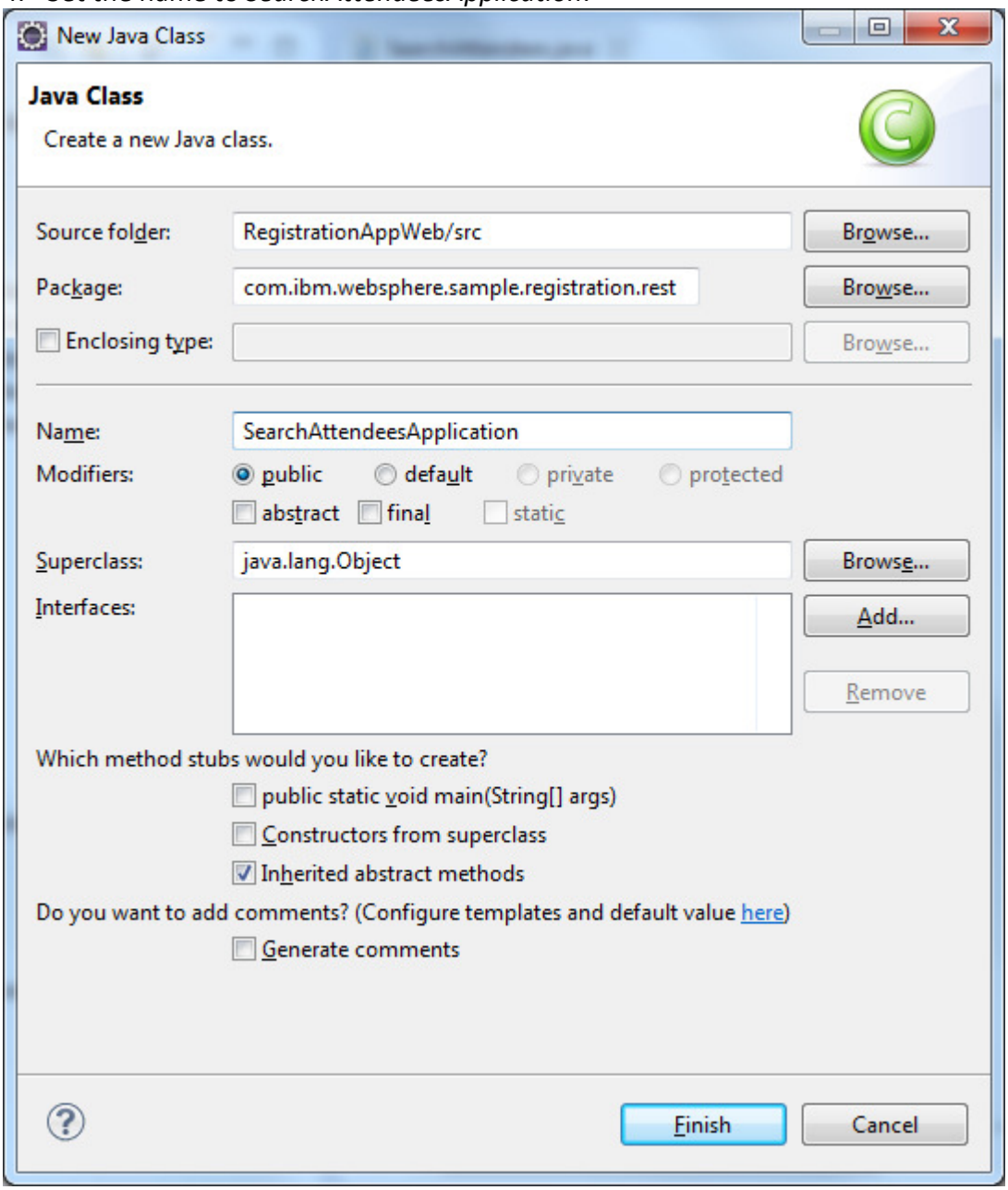
2. Set the package name to *com.ibm.websphere.sample.registration.rest*, and the **Class name** to *SearchAttendees*. Click *Finish*.



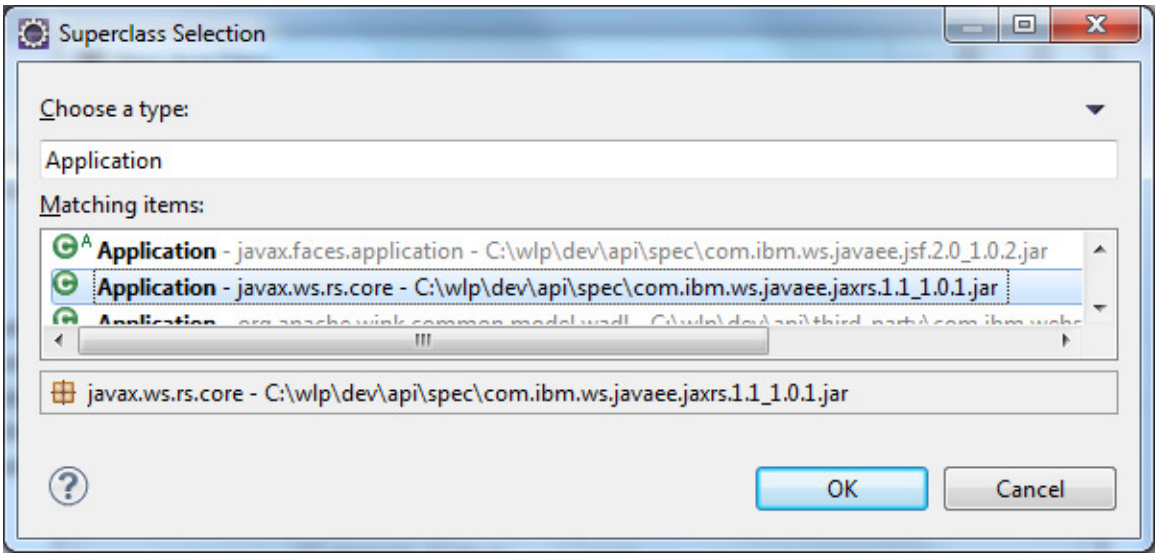
3. Right click on the com.*ibm.webshere.sample.registration.rest* package and select **New > Class**.



4. Set the name to *SearchAttendeesApplication*.



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**.

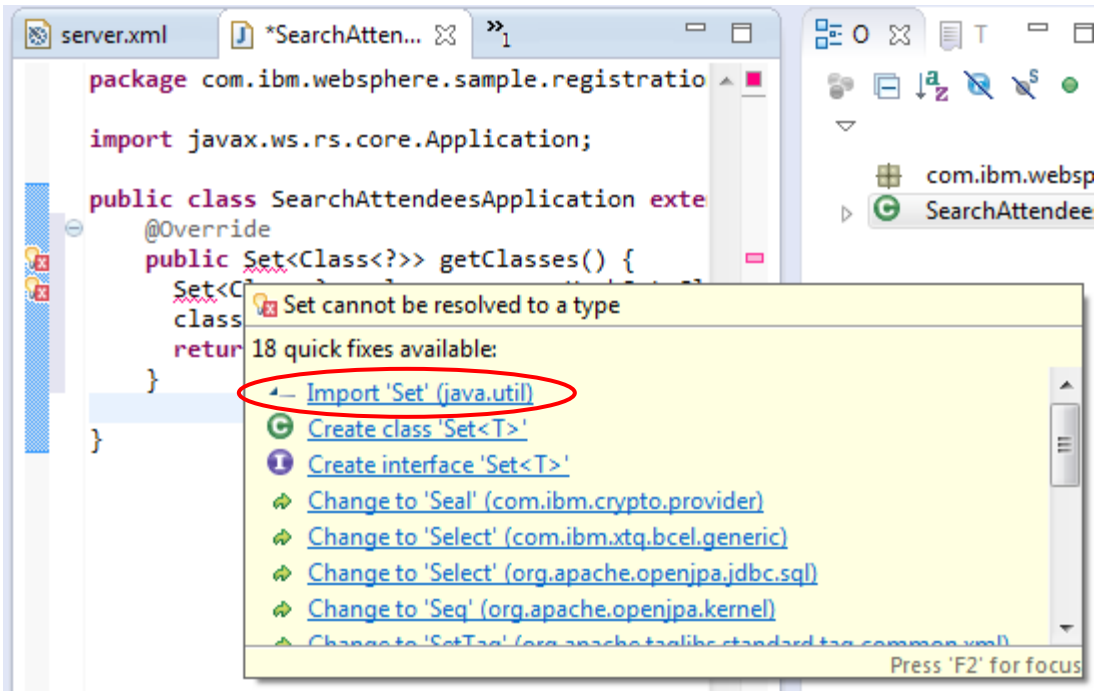


6. Click **Finish**.

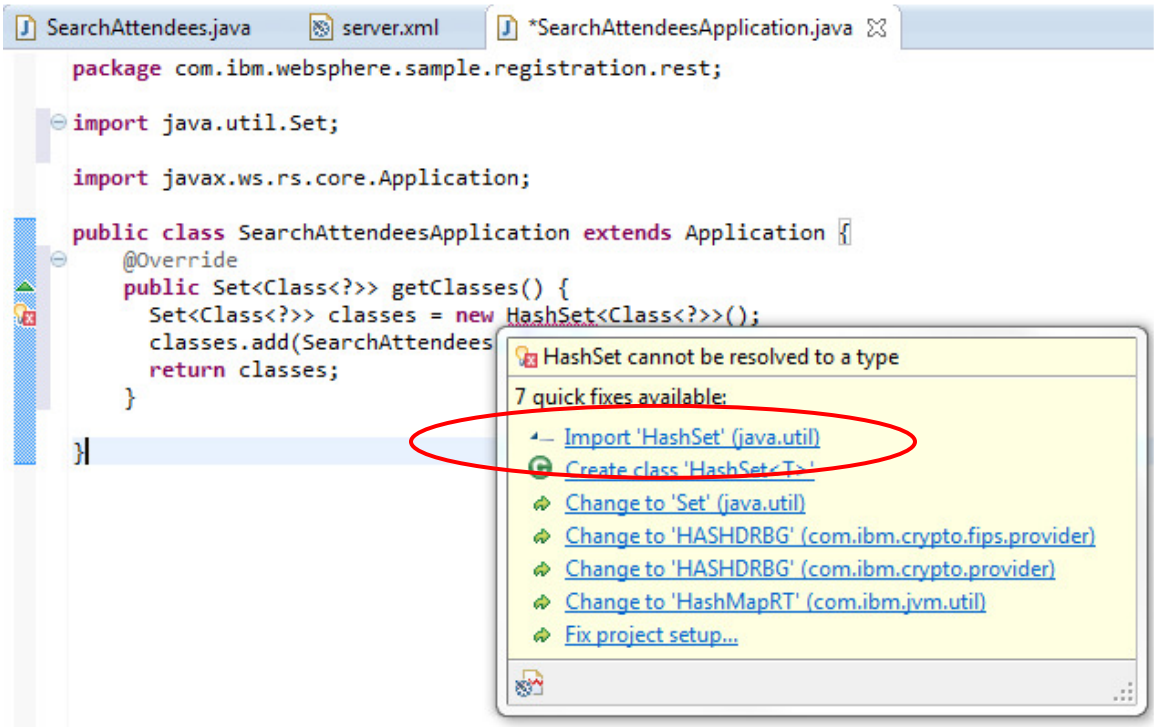
7. Add the following method to the `SearchAttendeesApplication` class:

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

8. Hover over the `Set` class and select **Import 'Set' (java.util)** from the dialog.



9. Hover over `HashSet` and select **Import 'HashSet' (java.util)** from the dialog.

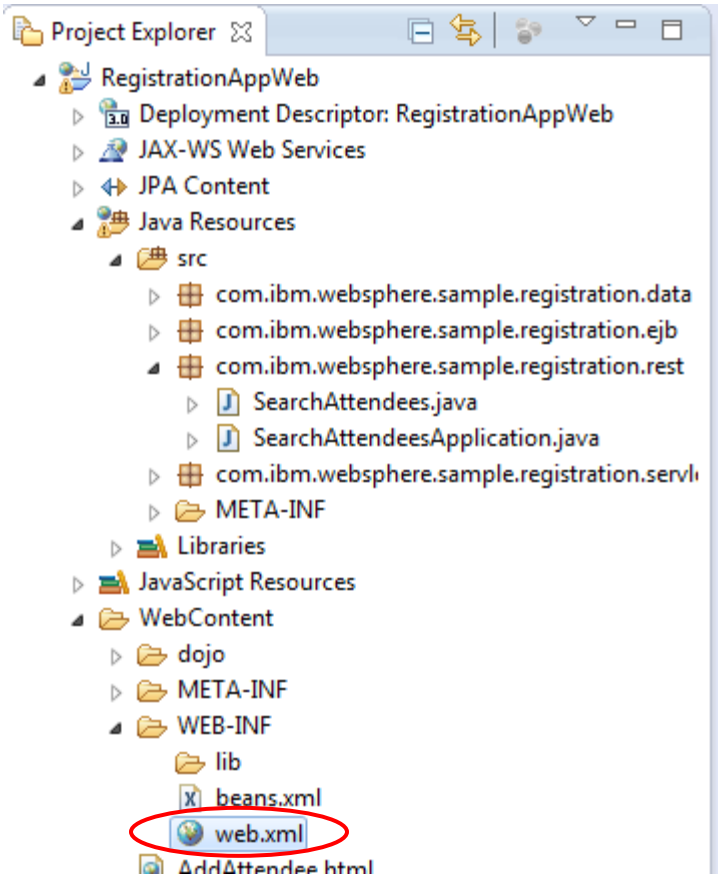


10. Save the file using the **control-S** shortcut. The completed class should match the following image.

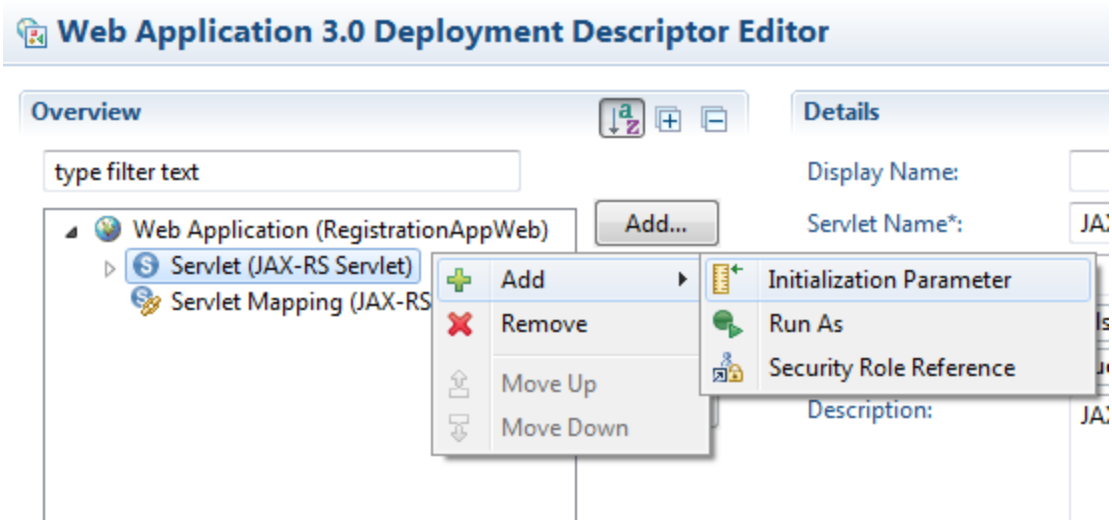
SearchAttendees.java | server.xml | SearchAttendeesApplication....

```
package com.ibm.websphere.sample.registration.rest;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.core.Application;

public class SearchAttendeesApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
      Set<Class<?>> classes = new HashSet<Class<?>>();
      classes.add(SearchAttendees.class);
      return classes;
    }

}
```

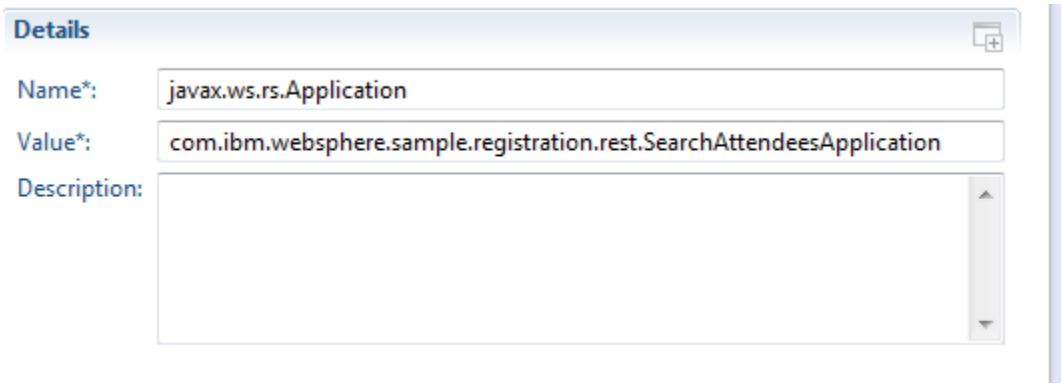
11. Open the `web.xml` file in the `WebContent/WEB-INF` folder of the **RegistrationAppWeb** project.



12. Right click on the **Servlet** element on the left pane and select **Add > Initialization Parameter**.



13. In the **Details** panel on the right, set the **Name** to *javax.ws.rs.Application*. In the **Value**, enter *com.ibm.websphere.sample.registration.rest.SearchAttendeesApplication*.

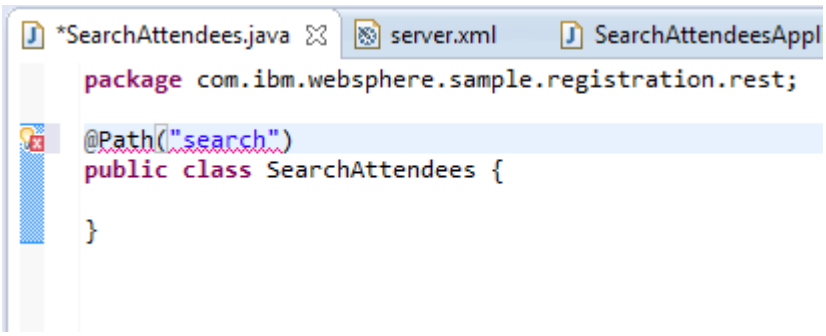


14. Save the file using the **control-S** shortcut.

15. Open the `SearchAttendees.java` class.

16. Add the following annotation to the `SearchAttendees` class declaration:

```
@Path("search")
```



17. Save the file using the **control-S** shortcut. Note that the file contains an import error. We will resolve this during the next few steps.

Creating a JAX-RS GET request handling method

The JAX-RS application is now in place. However, it does not contain any methods that are configured to handle requests. The first one we need to add is to handle GET requests. These will come from searches.

1. Add the following method into the `SearchAttendees` class to enable the search request to be processed:

```
public Collection<Person> searchForAttendees
                                    (String searchTerm) {
  return null;
}
```

2. To receive requests via HTTP GET requests, and return a collection of `Person` objects as JSON, add the following annotations to the method:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Collection<Person> searchForAttendees
                                      (String searchTerm) {
```

3. Inject the `RegistrationBean` EJB so that we can look up a list of users based on the filter by adding this line to the `SearchAttendees` class:

```
@Inject RegistrationBean rb;
```

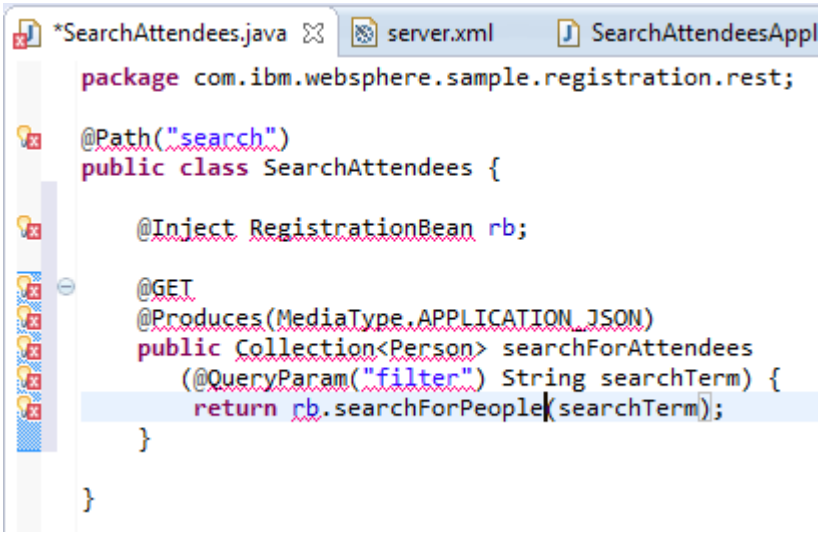
4. Change the return statement of the `searchForAttendees()` method to call the `RegistrationBean`.

```
return rb.searchForPeople(searchTerm);
```

5. Lastly, update the `filter` parameter with the @QueryParam annotation:

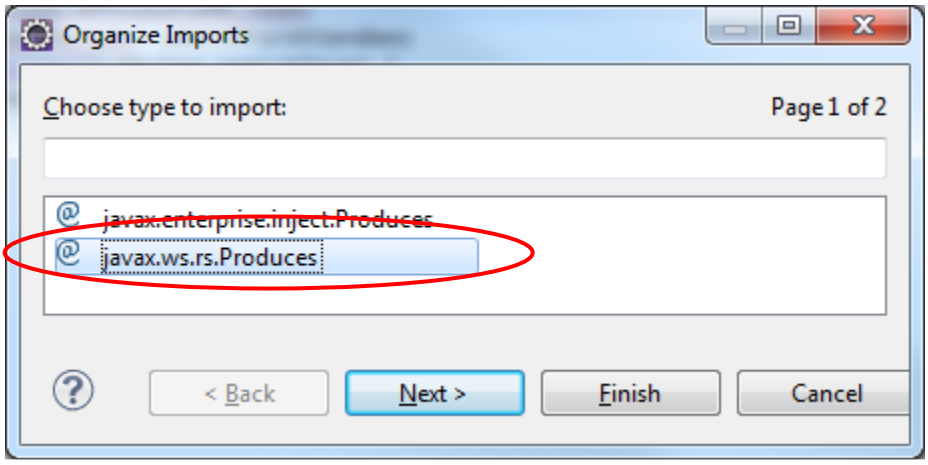
```
public Collection<Person>
          searchForAttendees(@QueryParam("filter")
                                    String searchTerm) {
```

6. The file at this stage should resemble the image below.

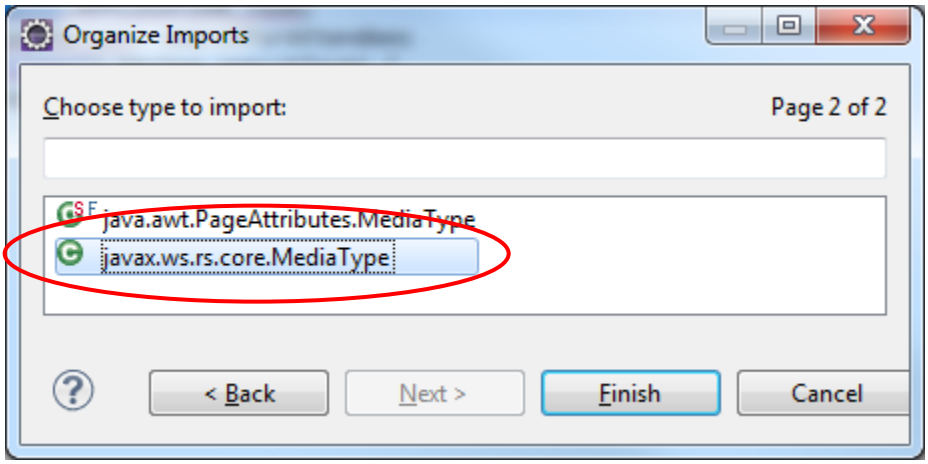


7. To resolve the import errors, use the **control-shift-o** keyboard shortcut. This will cause a prompt for ambiguous imports.

8. For the `Produces` annotation, select `javax.ws.rs.Produces`. Click **Next**.



9. For `MediaType` class, select `javax.ws.rs.core.MediaType`. Click **Finish**.



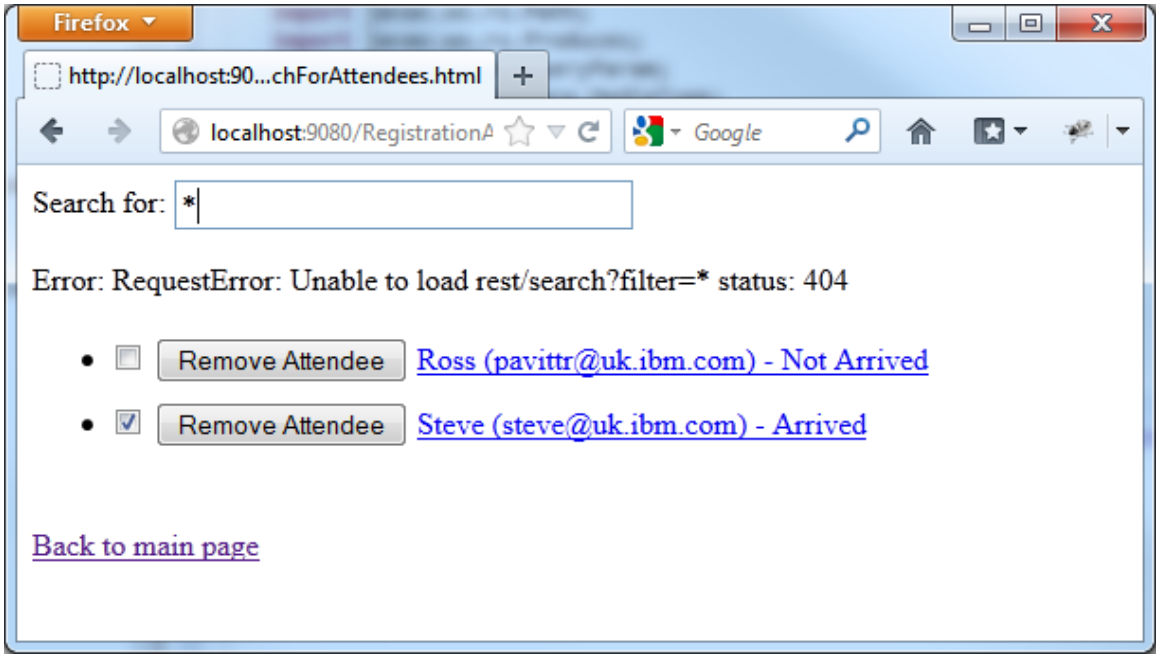
10. Save the file using the **control-S** shortcut. The file should resemble the image below. Ensure the import statements match.

SearchAttendees.java | server.xml | SearchAttendeesApplication.java

```
package com.ibm.websphere.sample.registration.rest;

import java.util.Collection;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

import com.ibm.websphere.sample.registration.data.Person;
import com.ibm.websphere.sample.registration.ejb.RegistrationBean;

@Path("search")
public class SearchAttendees {

    @Inject RegistrationBean rb;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Collection<Person> searchForAttendees
       (@QueryParam("filter") String searchTerm) {
        return rb.searchForPeople(searchTerm);
    }

}
```

Creating JAX-RS POST requests

Now that the search handler is in place, the search function should work (you can try this out by opening the Search For Attendees page in Firefox and setting the search term to "*").



Each search result displayed on the page includes a checkbox to mark whether the attendee is attending or not, and a button to allow the attendee to be unregistered. However clicking these currently returns a 404 error. This is because no handlers have been defined for these requests.

1. Open the `SearchAttendees` java class and copy the method below into the class to allow processing of POST requests with a URL of */search/update*. We still return a string of JSON text, but this time the return type is Person, not Collection<Person>, so the returned JSON string only contains the details of one person object (the one we have updated).

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Path("/update")
public Person changeAttendance(
           @QueryParam("email") String emailAddress,
           @QueryParam("state") boolean attending)
                                        throws Exception {
  String email =
          emailAddress.substring("checkbox".length());
  if (attending)
    rb.markAttended(email);
  else
    rb.markUnattended(email);
  return rb.getPerson(email);
```

```
}
```

2. Add the following method to the `SearchAttendees` class to allow the delete request to be processed, and to return a representation of the person who was unregistered:

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Path("/delete")
public Person changeAttendance(
           @QueryParam("email") String emailAddress)
                                        throws Exception {
  String email =
            emailAddress.substring("button".length());
  Person p = rb.getPerson(email);
  rb.unregister(email);
  return p;
}
```

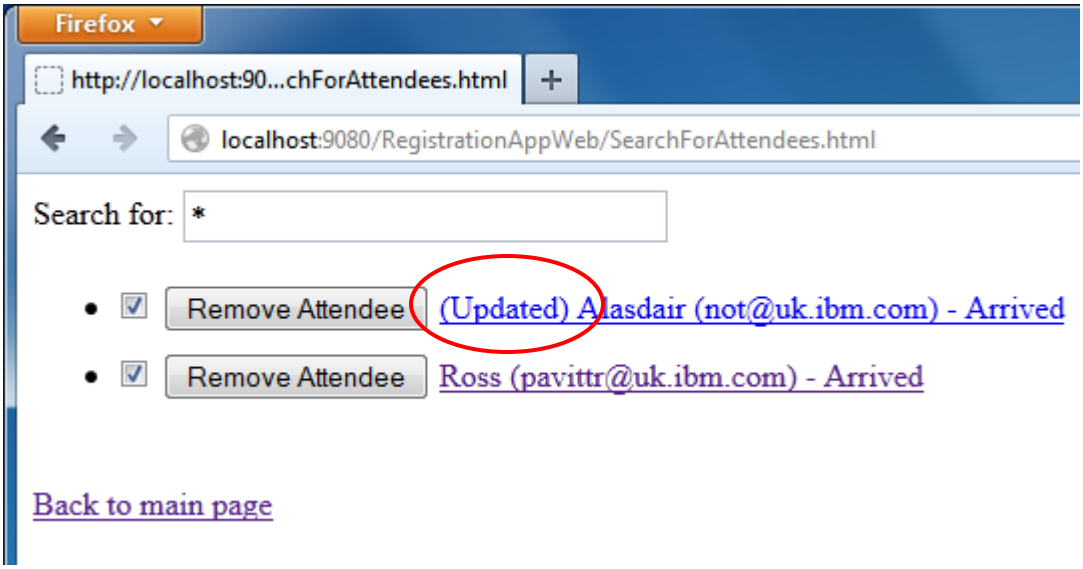
3. Resolve any import issues by using the **control-shift-o** shortcut again.

4. Save the file using the **control-S** shortcut. The class should resemble the image below.

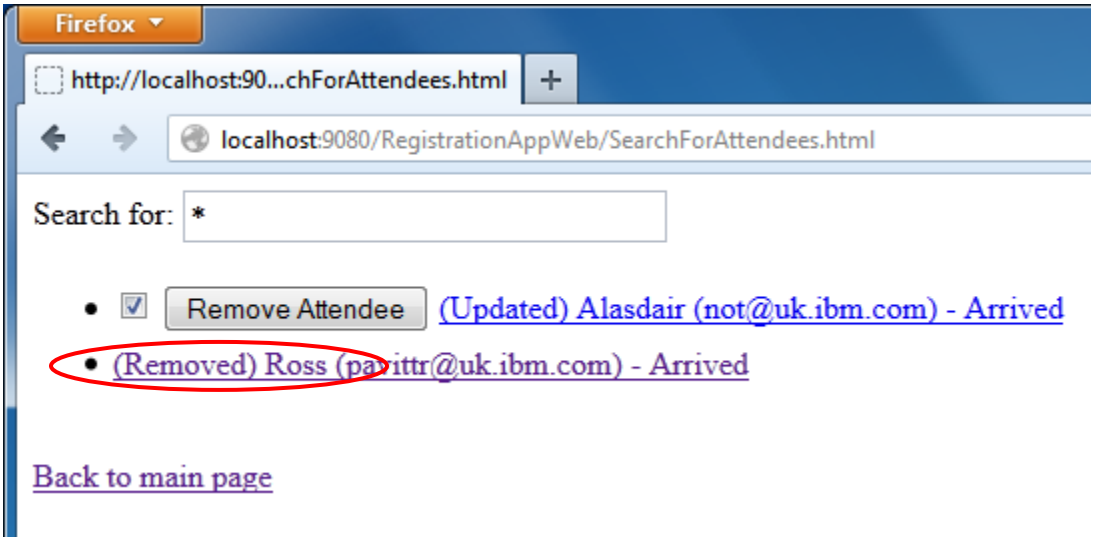
SearchAttendees.java | server.xml | SearchAttendeesApplication.java

```
package com.ibm.websphere.sample.registration.rest;

import java.util.Collection;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

import com.ibm.websphere.sample.registration.data.Person;
import com.ibm.websphere.sample.registration.ejb.RegistrationBean;

@Path("search")
public class SearchAttendees {

    @Inject RegistrationBean rb;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Collection<Person> searchForAttendees
       (@QueryParam("filter") String searchTerm) {
        return rb.searchForPeople(searchTerm);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/update")
    public Person changeAttendance(
              @QueryParam("email") String emailAddress,
              @QueryParam("state") boolean attending)
                                        throws Exception {
      String email =
             emailAddress.substring("checkbox".length());
      if (attending)
        rb.markAttended(email);
      else
        rb.markUnattended(email);
      return rb.getPerson(email);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/delete")
    public Person changeAttendance(
              @QueryParam("email") String emailAddress)
                                        throws Exception {
      String email =
               emailAddress.substring("button".length());
      Person p = rb.getPerson(email);
      rb.unregister(email);
      return p;
    }
}
```

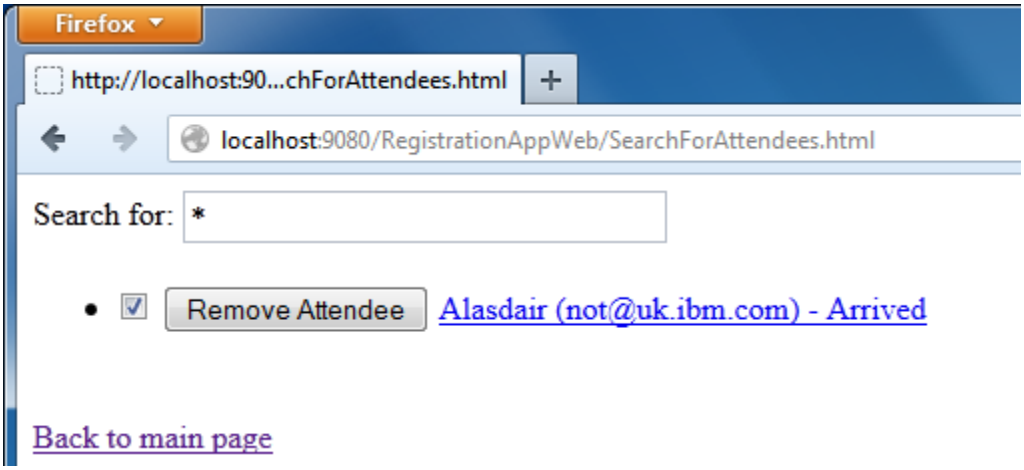
5. Return to the landing page in your browser, and select the **Search Attendees** link. Perform the search again. Now, attempt to toggle the checkbox for the attendee. This causes the attendee to be updated as shown below.



6. Remove one of the attendees by clicking the **Remove Attendee** button. This removes the button and marks the attendee as removed.



7. Performing a new search shows the attendee has gone.



Summary

In this section you learned:

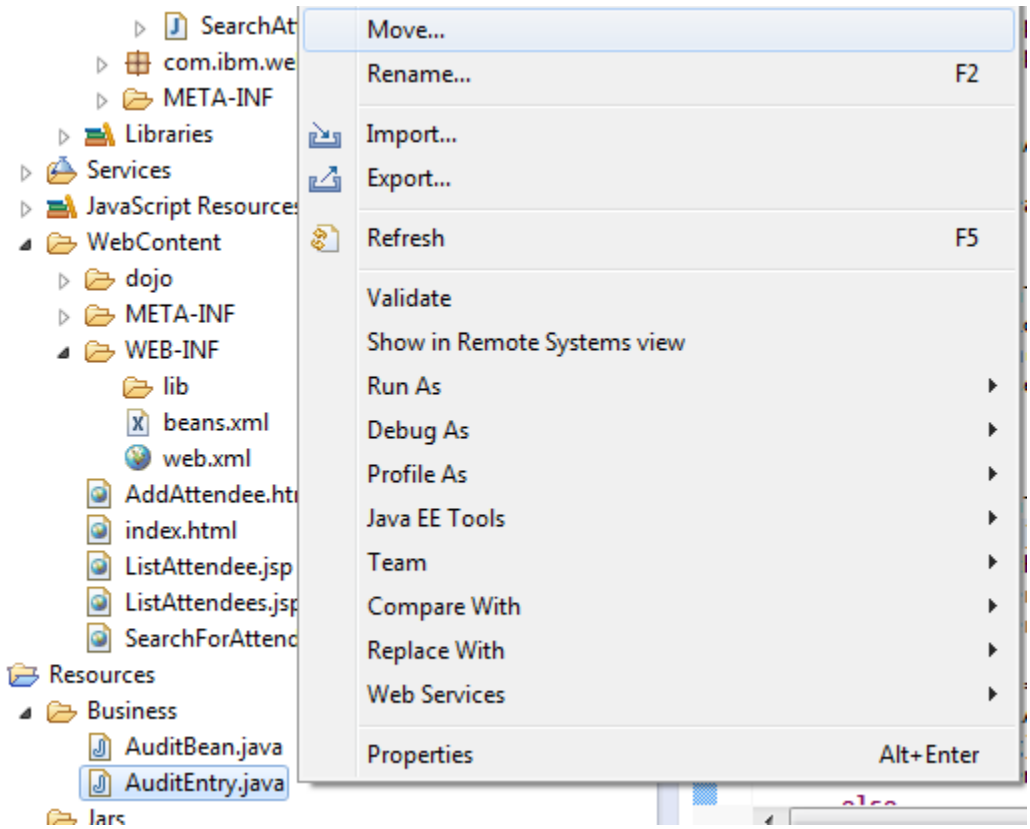
- How to enable JAX-RS support for an application
- How to interact with client side technologies using AJAX techniques

Adding another EJB

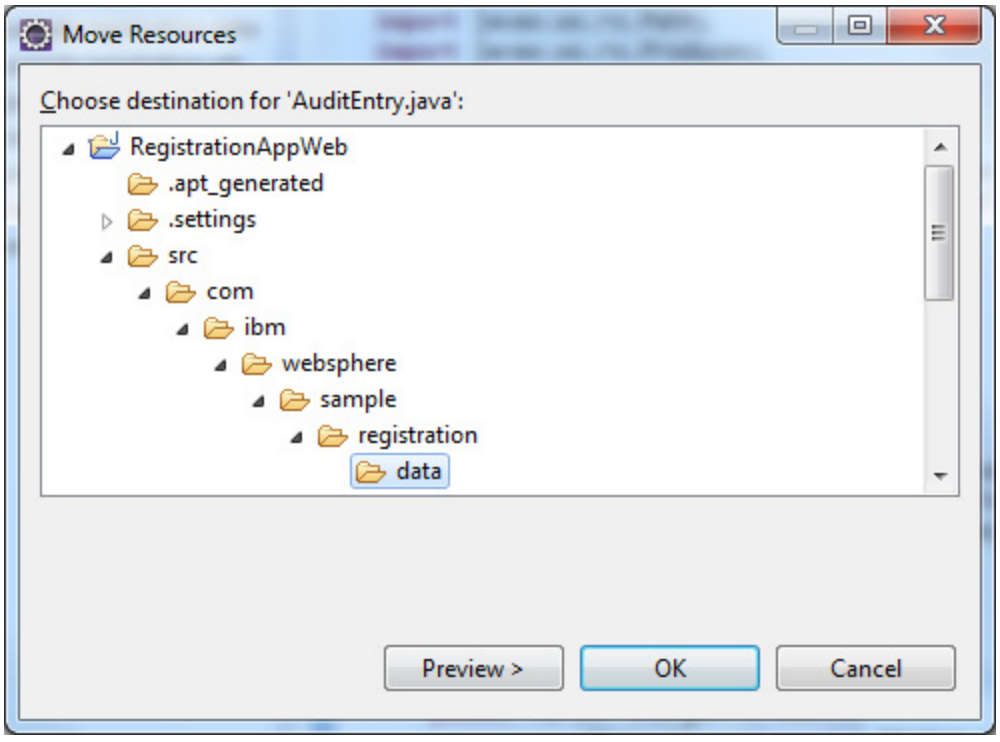
Currently our application uses one EJB to handle the business logic of the application.

In this section, we will add another function, which will log use of certain business logic methods on the `RegistrationBean`. As we want to keep our logging separate, we will use a separate EJB to manage that data access.

1. Open the `Business/view` folder of the **Resources** project, right click on the `AuditEntry` class and click on **Move**.

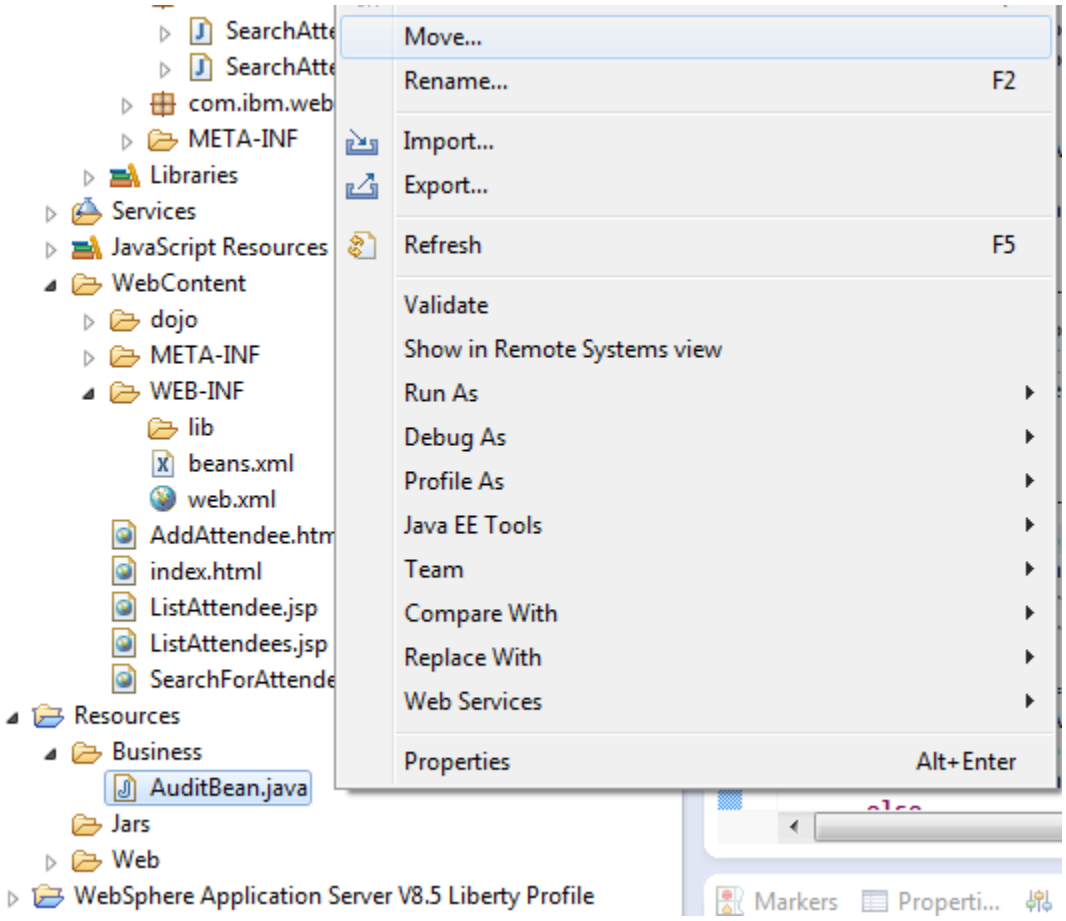


2. Expand the **RegistrationAppWeb** project down to `src/com/ibm/websphere/sample/registration/data`.

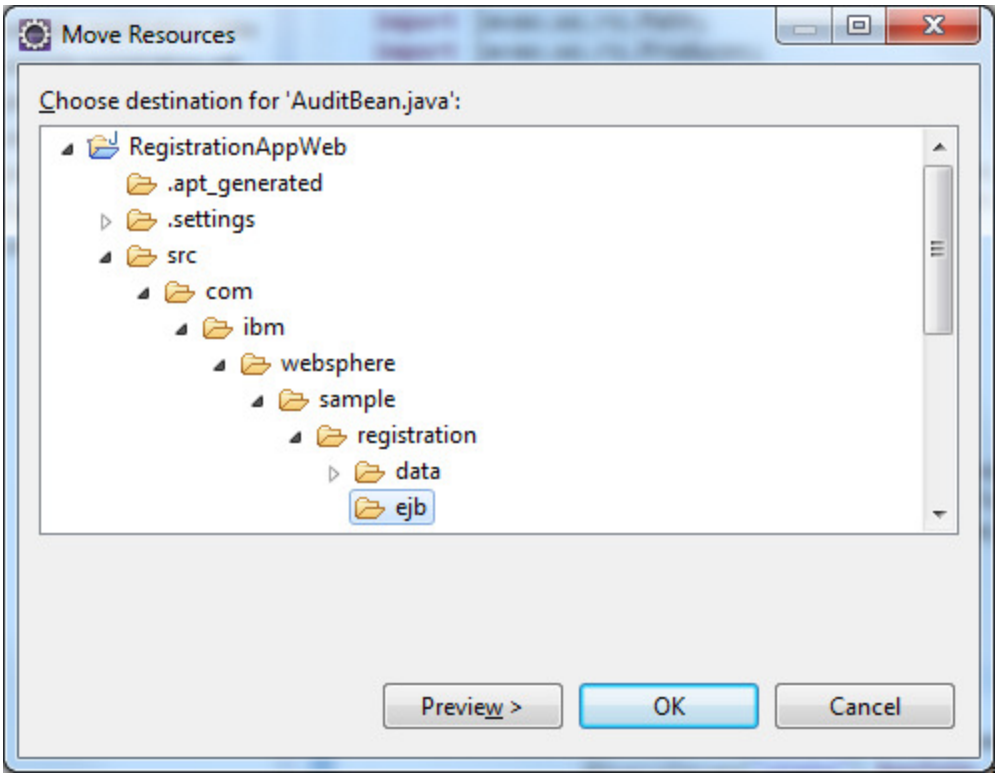


3. Click on the `data` folder and click **OK**. This is the same folder that contains the `Person.java` class.

4. Open the `Business` folder of the **Resources** project, click on the `AuditBean` class. Next, right click and select **Move**.



5. Expand the **RegistrationAppWeb** project down to `src/com/ibm/websphere/sample/registration/ejb`.

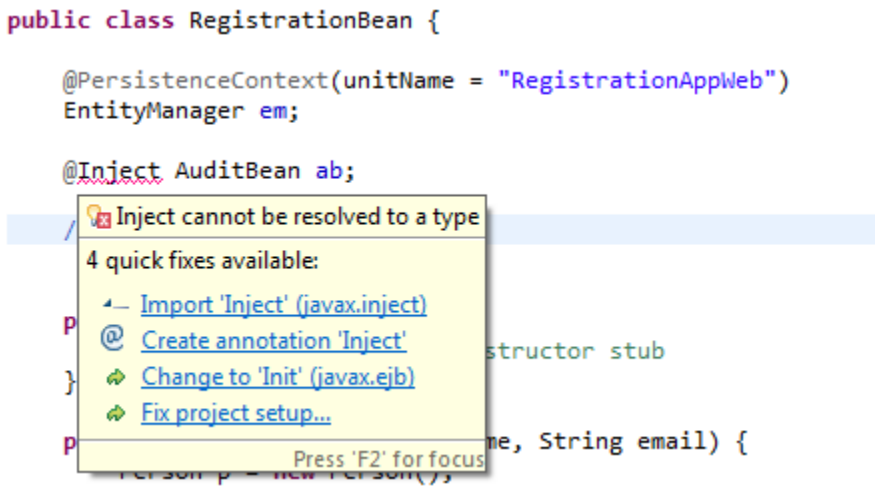


6. Click on the `ejb` folder and click **OK**.

7. Open the `RegistrationBean` class and add the injection for the `AuditBean`:

   ```
   @Inject AuditBean ab;
   ```

8. Hover over the `@Inject` annotation and select **Import 'Inject' (javax.inject)** from the dialog.



9. Next, add audit lines to the register and unregister methods:

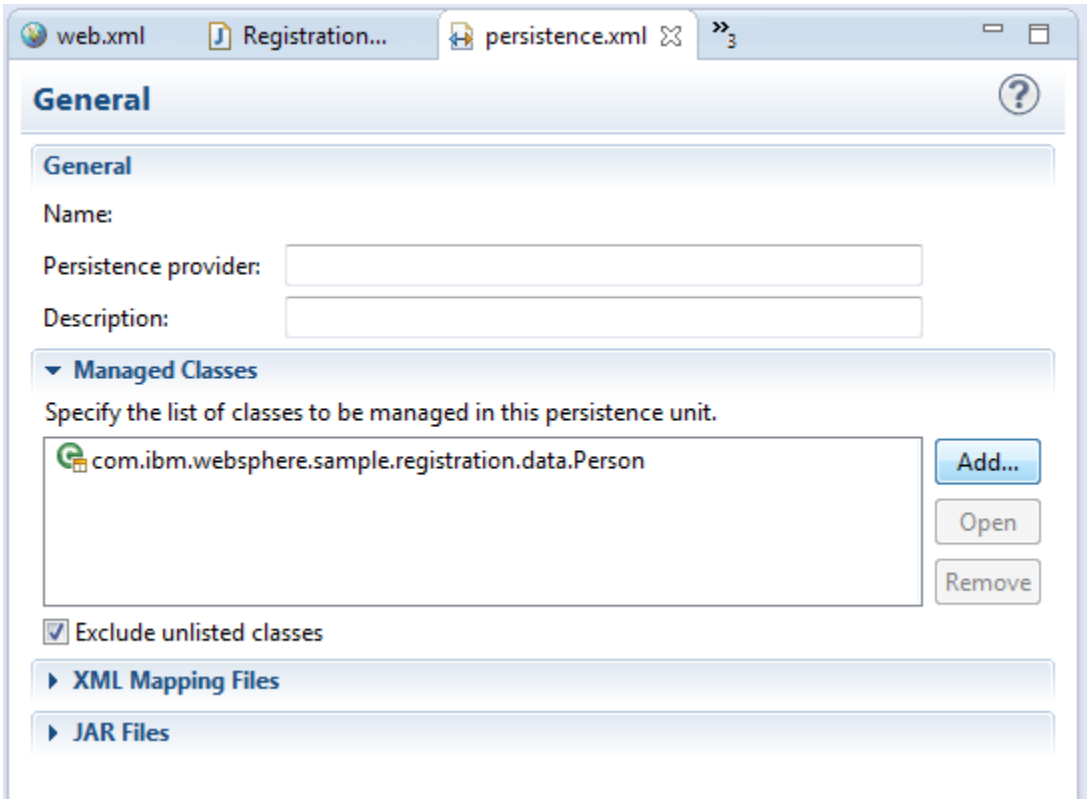
```
  public void register(String name, String email) {
    ab.logUpdate("Registering attendee "
                    + name + " with email " + email);

  public void unregister(String email) {
    ab.logUpdate("Unregistering attendee with email "
                                                + email);
```

```
public void register(String name, String email) {
    ab.logUpdate("Registering attendee "
            + name + " with email " + email);

    Person p = new Person();
    p.setEmail(email);
    p.setName(name);
    em.persist(p);
}

public void unregister(String email) {
    ab.logUpdate("Unregistering attendee with email "
            + email);

    Person p = em.find(Person.class, email);
    em.remove(p);
}
```

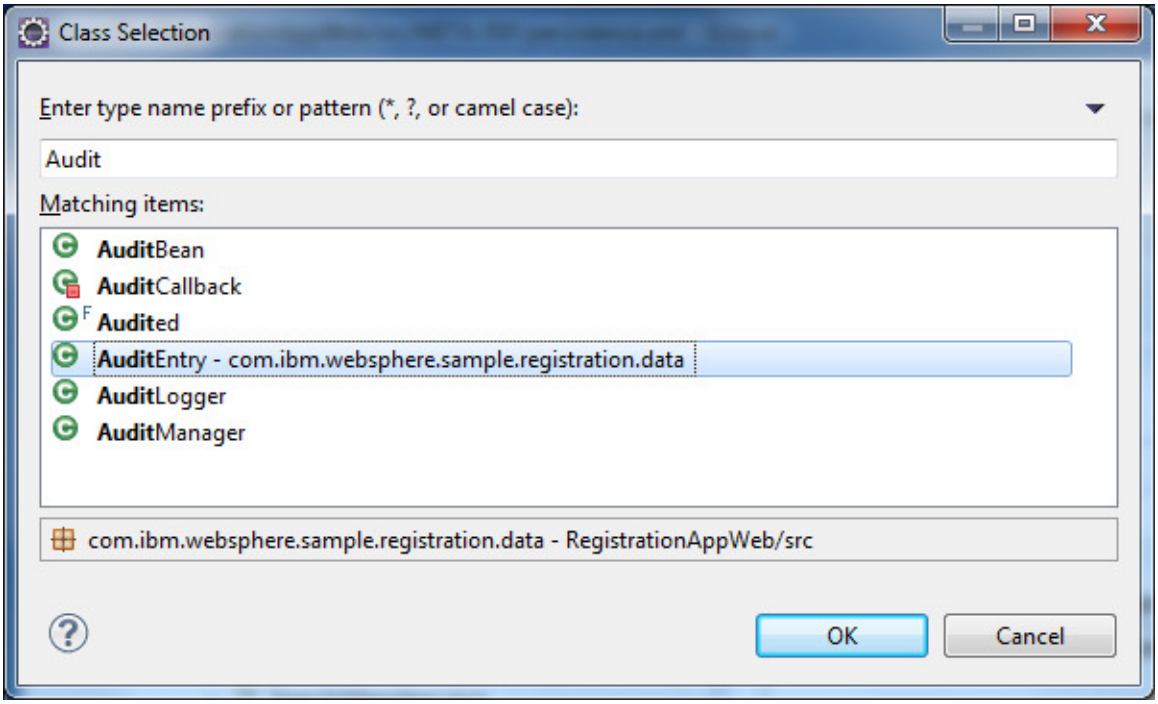
10. Save the file using the **control-S** shortcut.

11. Open the `persistence.xml` file.

12. Switch to the **General** tab, and click the **Add…** button.



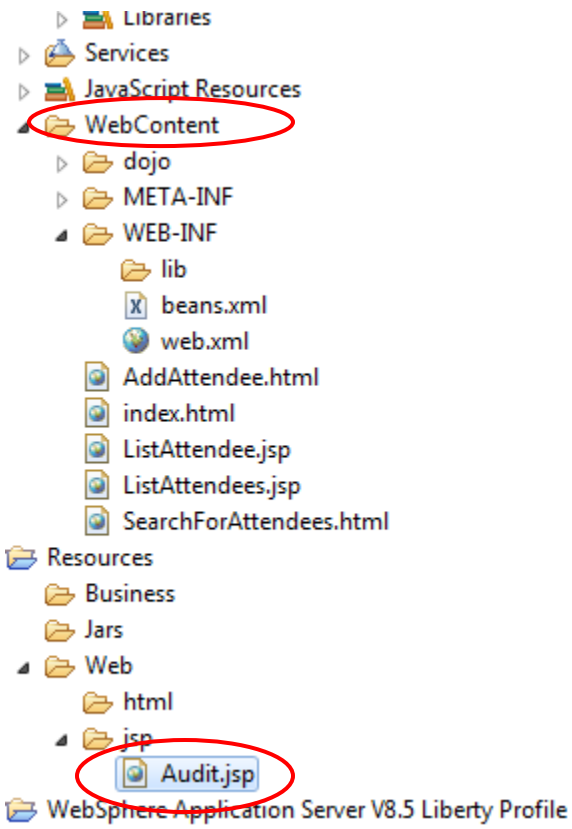
13. Type *Audit* in the filter box, and select `AuditEntry` from the `com.ibm.websphere.sample.registration.data` package.



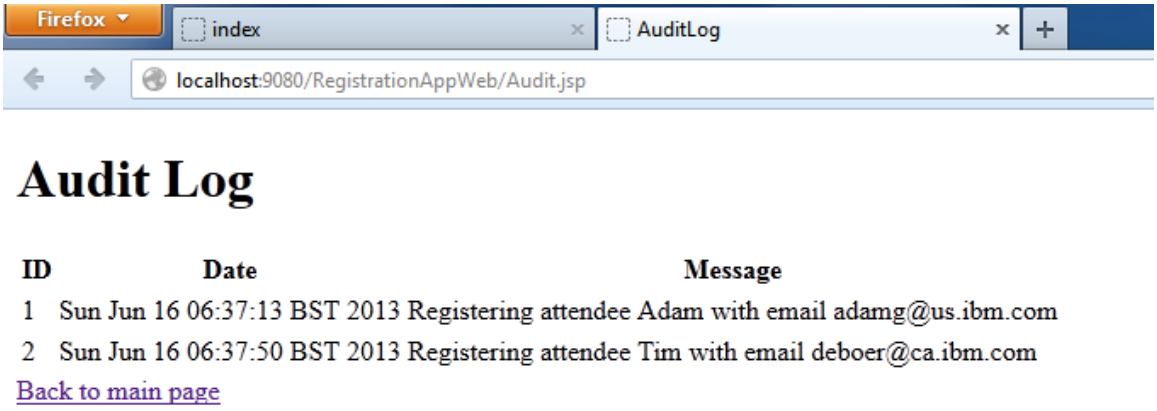
14. Save the file using the **control-S** shortcut.

15. Open the landing page, click on the **Add Attendees**, and add a couple of attendees.

16. Copy the `Audit.jsp` file into the application by dragging it from the `Web/jsp` folder of the **Resources** project, and dropping it into the `WebContent` folder of the **RegistrationAppWeb** project.



17. Navigate to the page using the URL http://localhost:9080/RegistrationAppWeb/Audit.jsp . This contains a line for the register of an attendee that you just performed. It will also show any unregisters you perform, and any edits to an attendee which will show the attendee being removed and then re-registered.

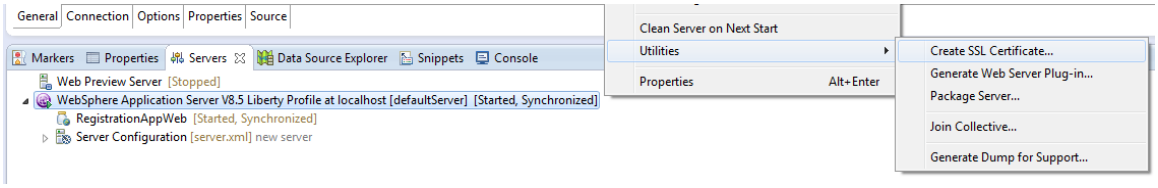


Securing the application

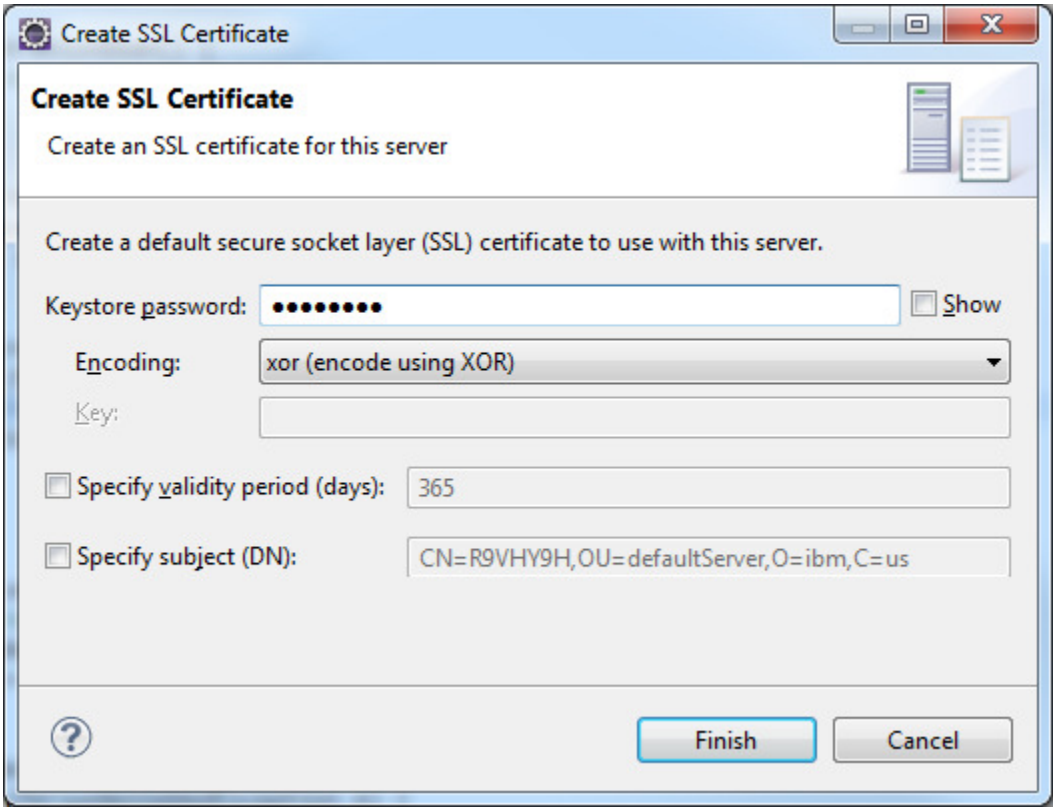
Applications may handle sensitive data such as personal information, and may require the use of SSL to ensure secure transportation of data over a network. The following section will add SSL to the application, and ensure it is enforced in all communication between the application and the client.

Add an SSL certificate

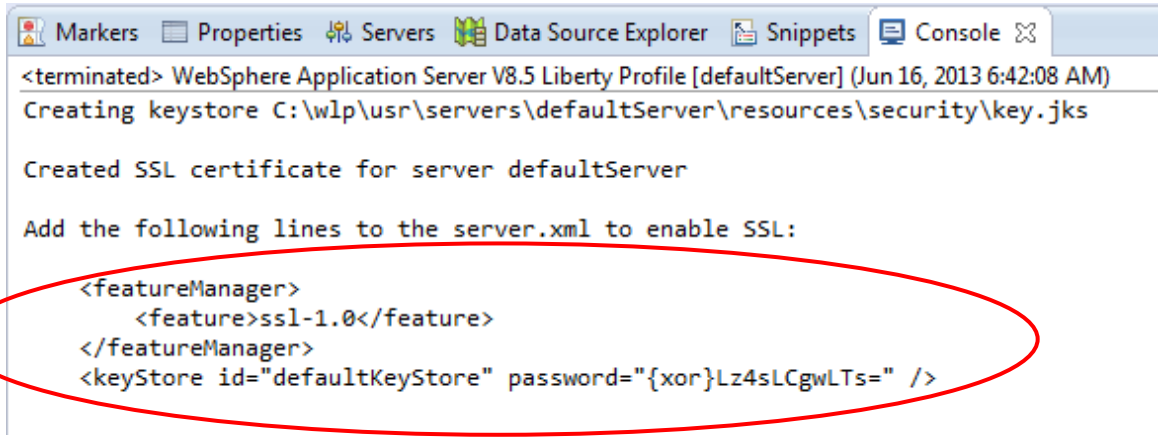
1. Open the **Servers** view, right click on the server and select **Utilities > Create SSL Certificate**.



2. Provide a password for your keystore, then click **Finish**.



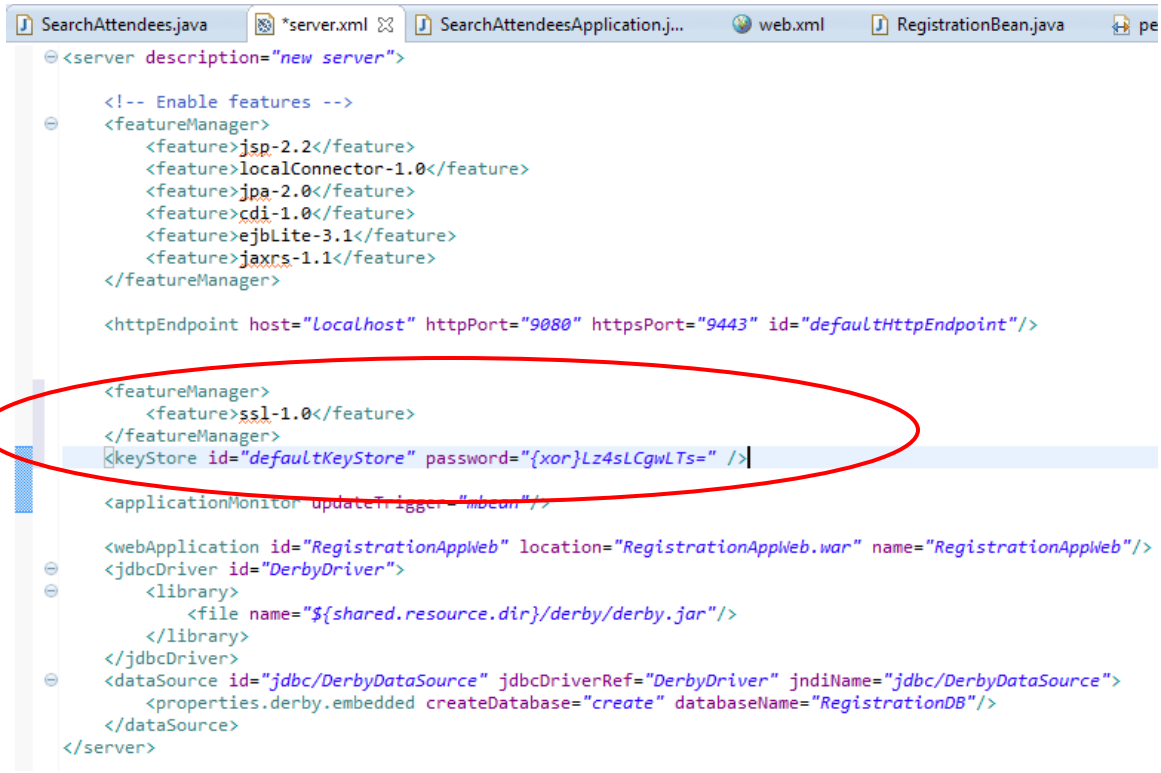
3. A **Console** window will appear, containing the configuration for SSL. Highlight the configuration section in the console, and copy it into memory using the control-C shortcut.



4. In the **Servers** view, expand the server definition to show the **Server Configuration** element.

5. Right click on the **Server Configuration** element and select **Open**.

6. Switch to the **Source** tab, and paste the contents of the clipboard using the **control-V** shortcut.



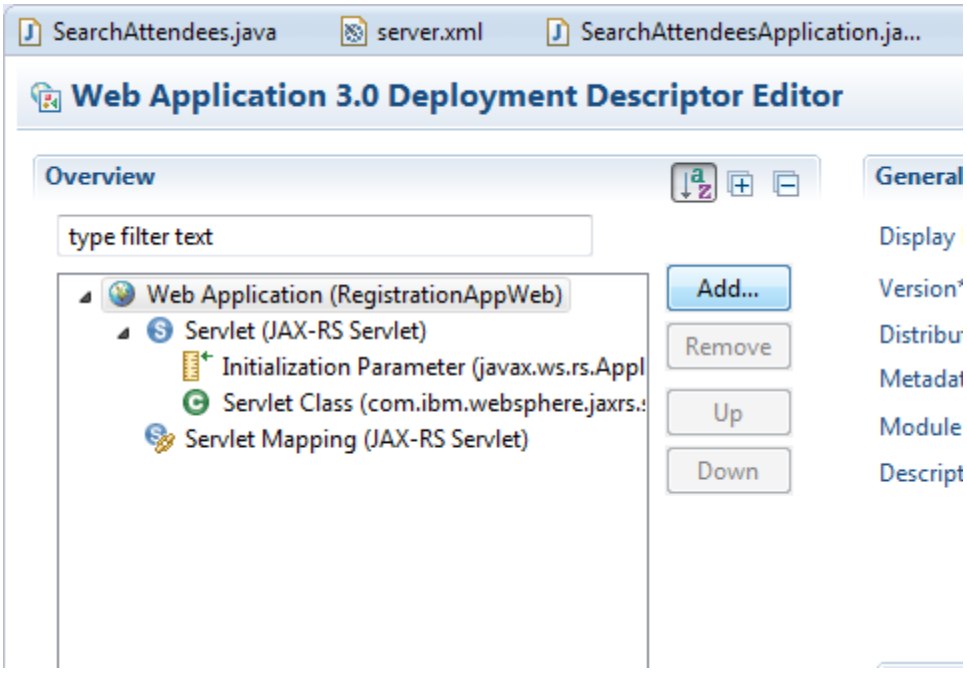
7. Save the file using the **control-S** shortcut.

Configuring the application to use transport guarantees

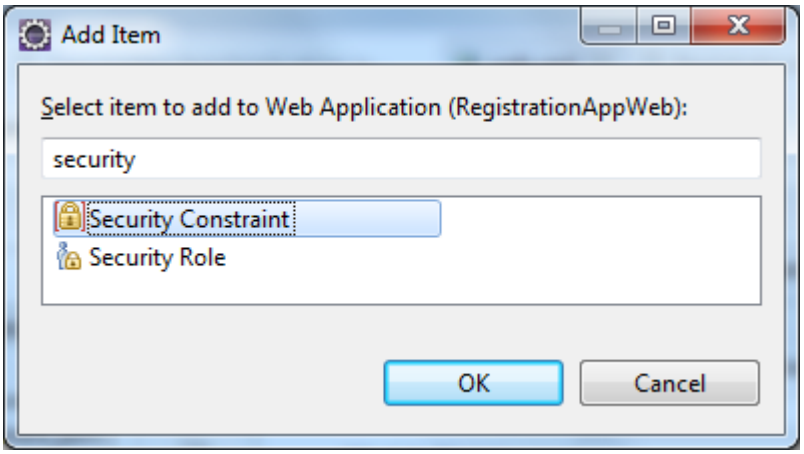
The configuration above will enable the use of SSL, but it does not force the application to only work over SSL, meaning it can be accessed via HTTP. However, we can configure this in the application to insist on using the CONFIDENTIAL transport guarantee. Be aware that there is one more configuration step required after this to ensure the Liberty profile server also enforces SSL only.

1. Open the `web.xml` file of the `WebContent/WEB-INF` folder of the **RegistrationAppWeb** project.

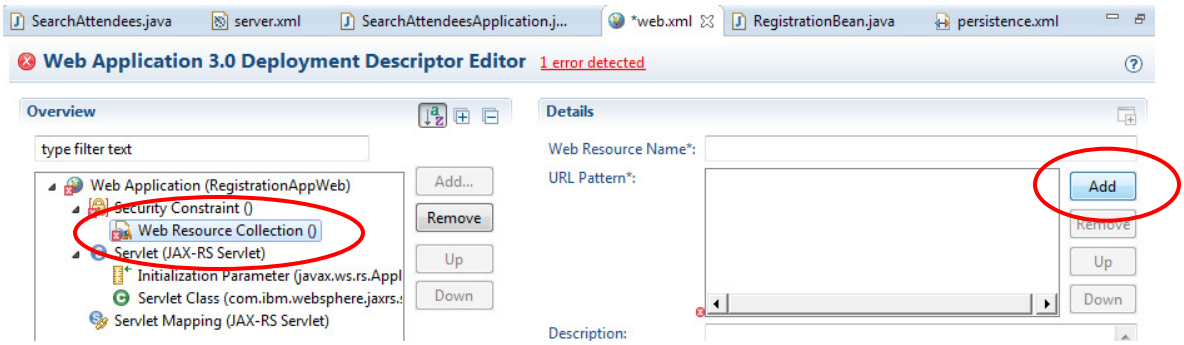
2. Click on **Web Application (RegistrationAppWeb)**, and click the **Add** button.



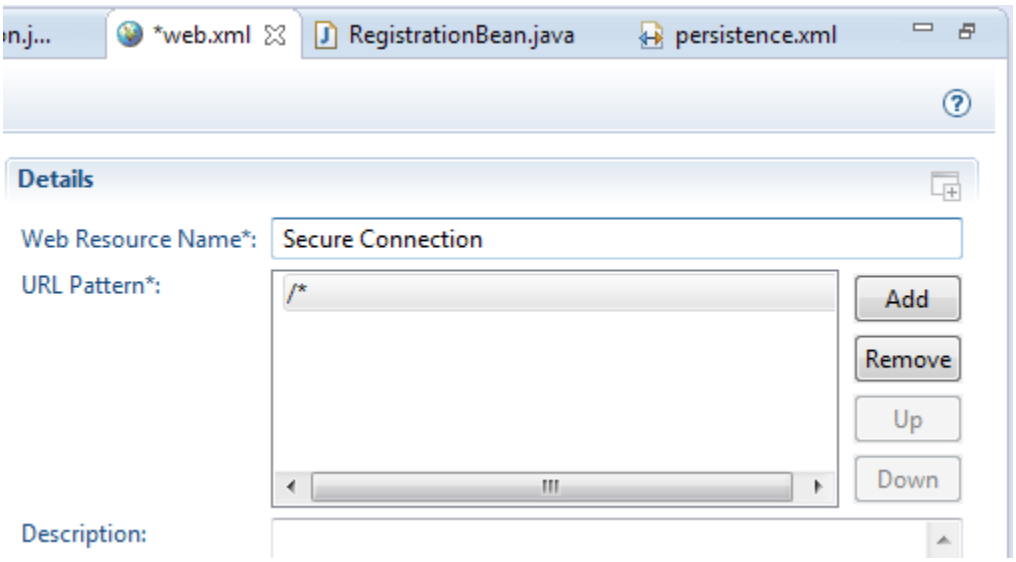
3. Using the filter, type *Security*, and select the **Security Constraint** element. Click **OK**.



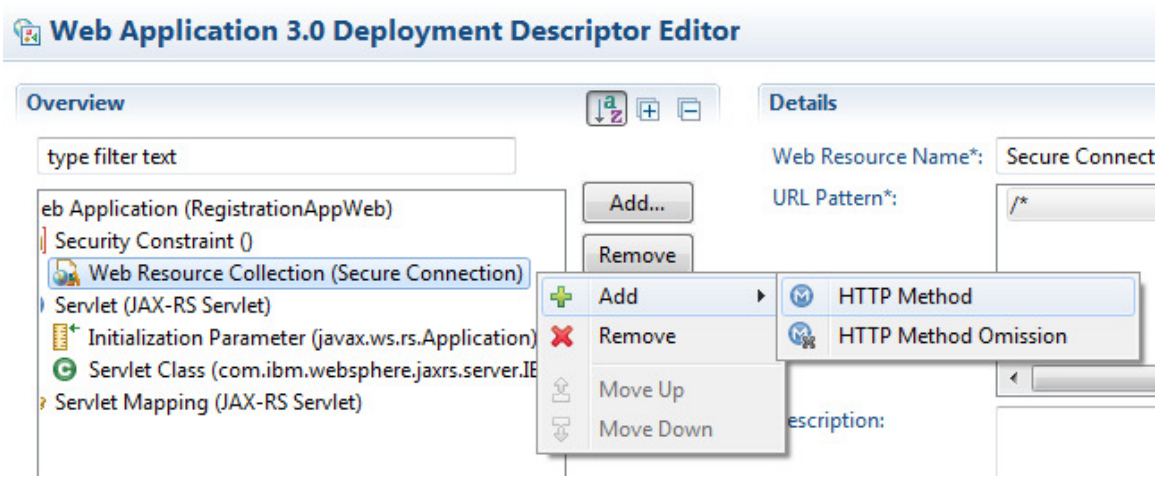
4. Click the newly created **Web Resource Collection** element, and click **Add** next to the **URL Pattern.**



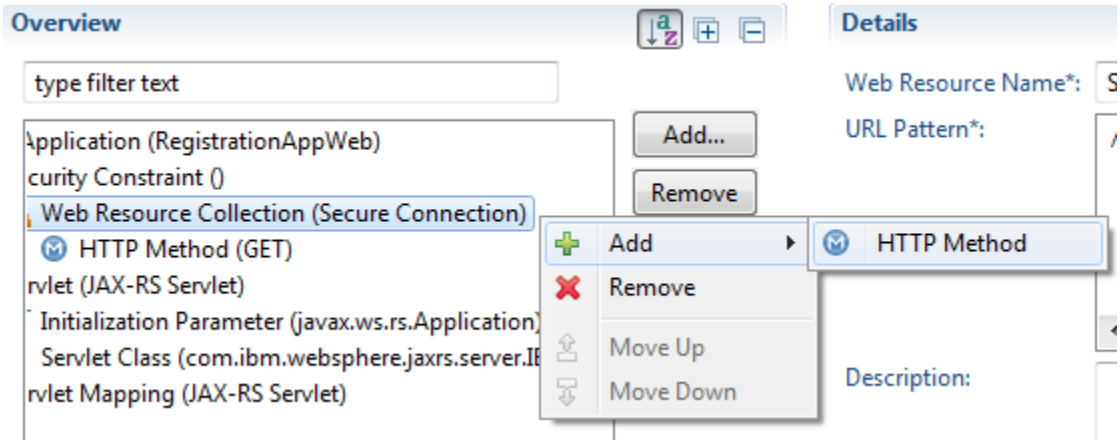
5. Type the pattern as "/*". Provide a **Web Resource Name** of *Secure Connection*.



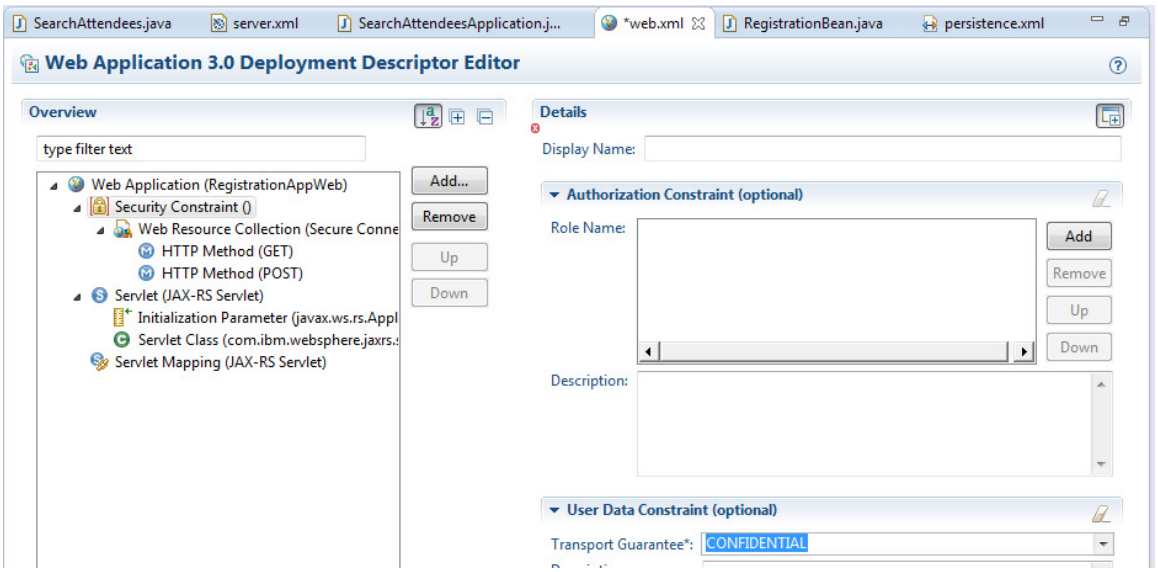
6. Right click on the **Web Resource Collection** in the left pane, and add a **HTTP Method**. In the **Details** section set the **HTTP Method** to *GET*.



7. Right click on the **Web Resource Collection** in the left pane again, add another **HTTP Method**. In the **Details** section set the **HTTP Method** to *POST*.



8. Click on the **Security Constraint**. Scroll down to the **User Data Constraint (optional)** section, and change the **Transport Guarantee** dropdown to *CONFIDENTIAL*.



9. Save the file using the **control-S** shortcut.

Configure security on the Liberty profile

The final stage to enable security is to ensure the security feature is enabled on the Liberty profile.

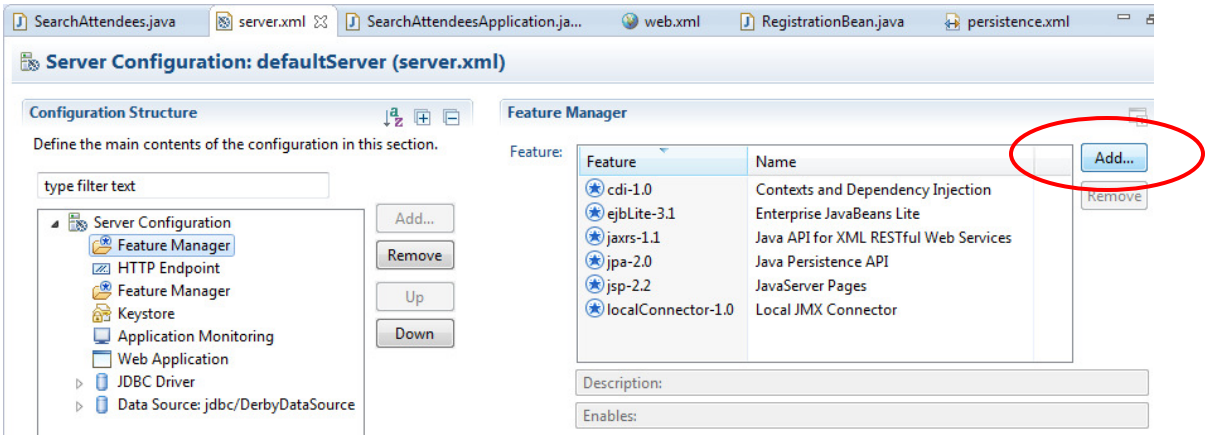
1. In the **Servers** view, expand the server definition to show the **Server Configuration** element.

2. Right click on the **Server Configuration** element and select **Open**.

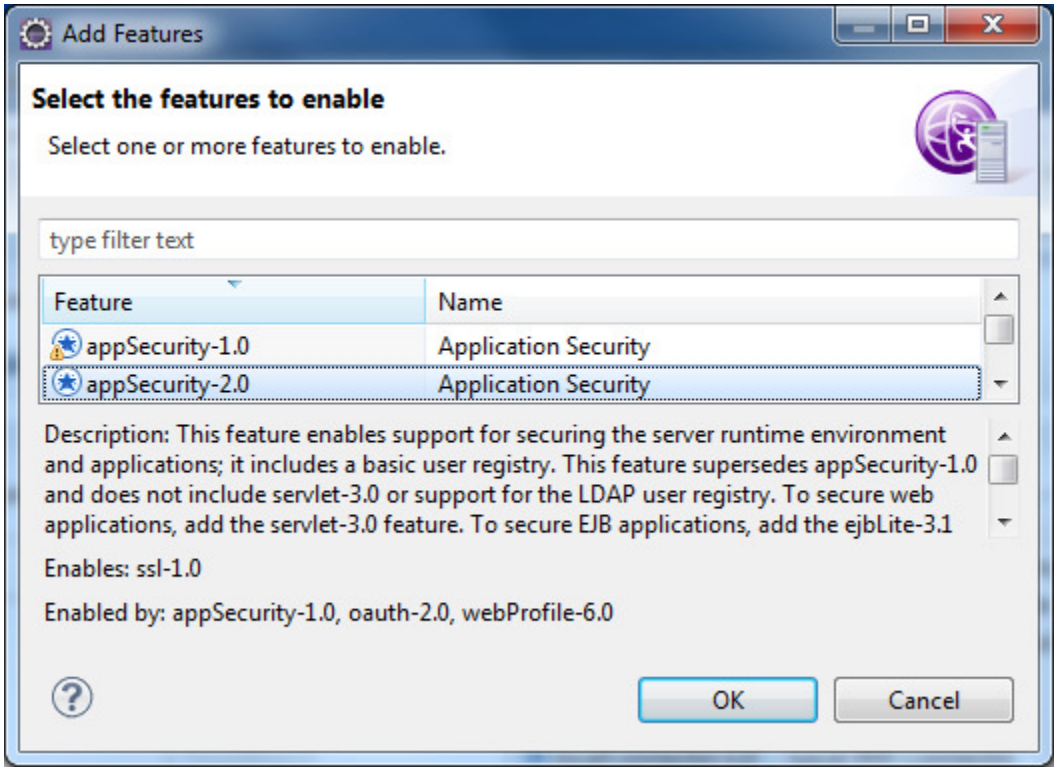
3. Switch to the **Design** tab.

4. Expand **Server Configuration** and click on any **Feature Manager** element (there may be more than one).

5. In the right pane, click the **Add** button.



6. From the dialog, select `appSecurity-2.0`. Click **OK**.



7. Save the file using the **control-S** shortcut.

Testing security

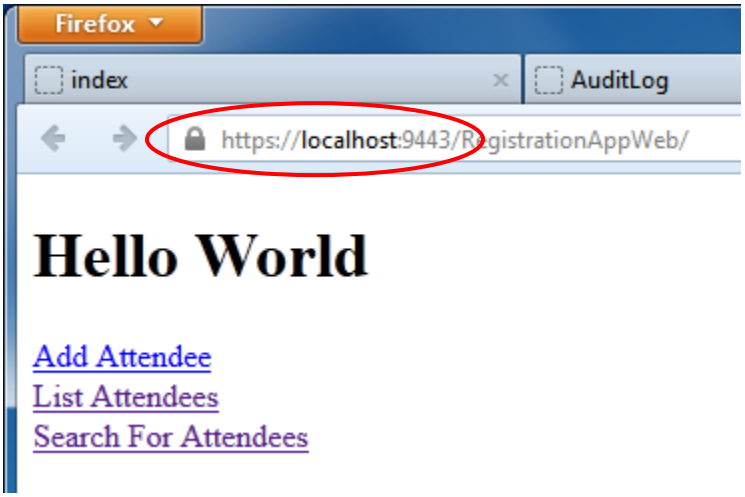
The final stage is to attempt to connect to the server via an unsecured port, and ensure your traffic is redirect to the SSL port.

1. In Firefox, open a connection to http://localhost:9080/RegistrationAppWeb.

2. If this succeeds, you will be redirected to the SSL port (port 9443).

3. Accept the security exception caused by the unknown certificate. This is we are using the certificate you generated in a previous step.

4. Ensure the page displays as expected, the protocol is set to HTTPS, and the port is the secured 9443 HTTPS port.



Summary

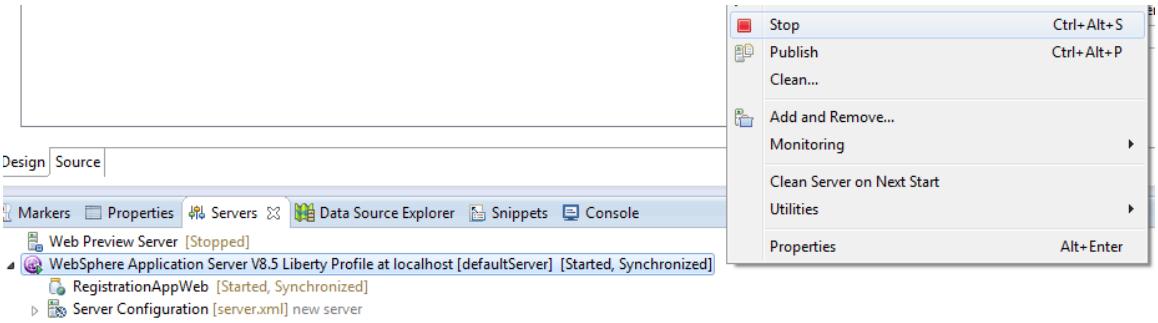
In this section you learned:

- How to secure the Liberty profile server for secure communication using SSL
- How to enforce SSL only traffic on applications

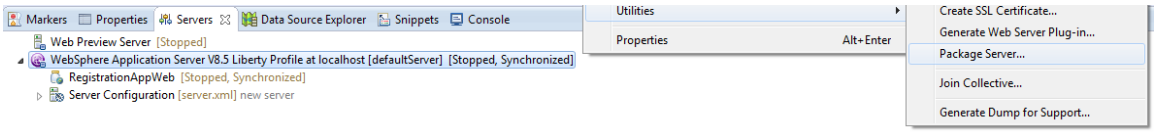
Packaging the server for deployment

The Liberty profile contains the ***Minify*** function. This allows you to build a customised Liberty package that contains only the features you require to run the applications you have installed.

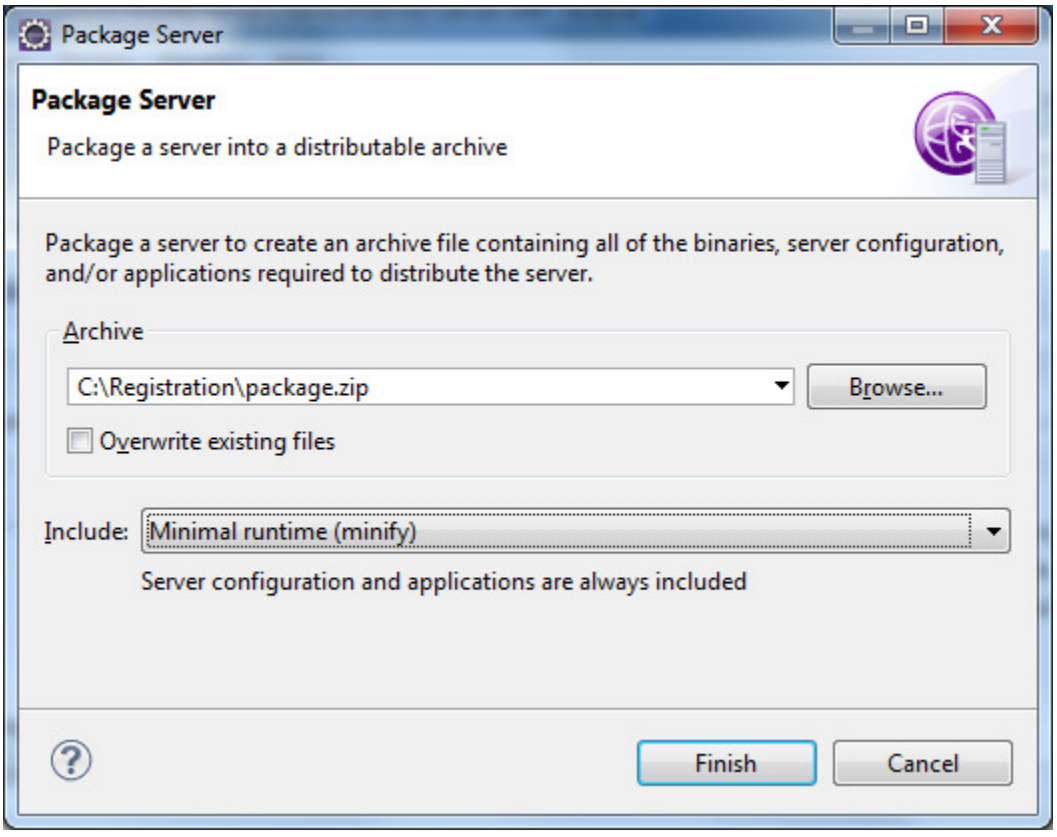
1. In the **Servers** view, shutdown the server by right clicking on the server and selecting **Stop**.



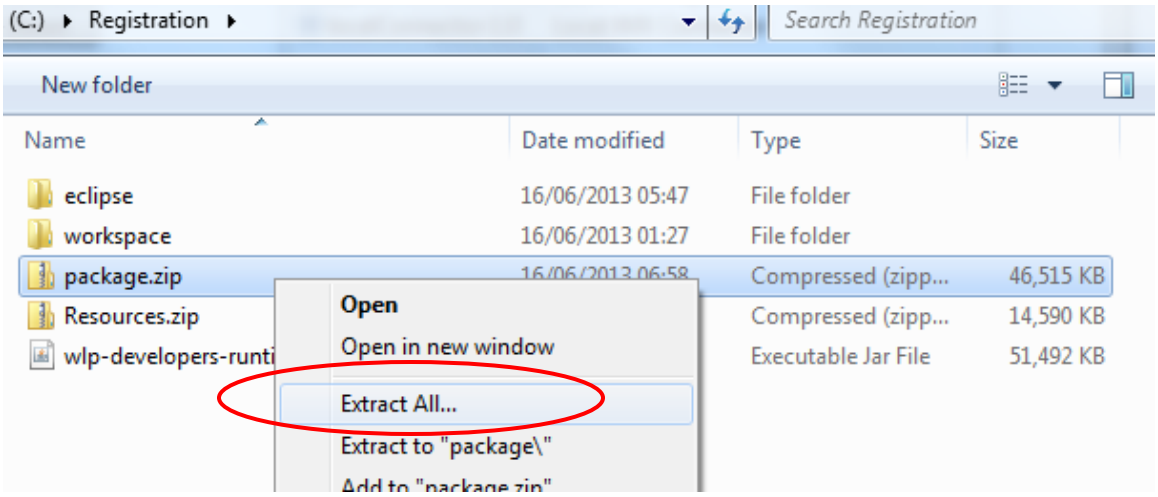
2. Right click on the server in the **Server** view, and select **Utilities > Package Server**.



3. Click **Browse** next to the **Archive name**. Choose a folder, and set the name to `package.zip`. Under the **Include** section, change the dropdown to **Minimal Runtime (minify)**, and click **Finish**.

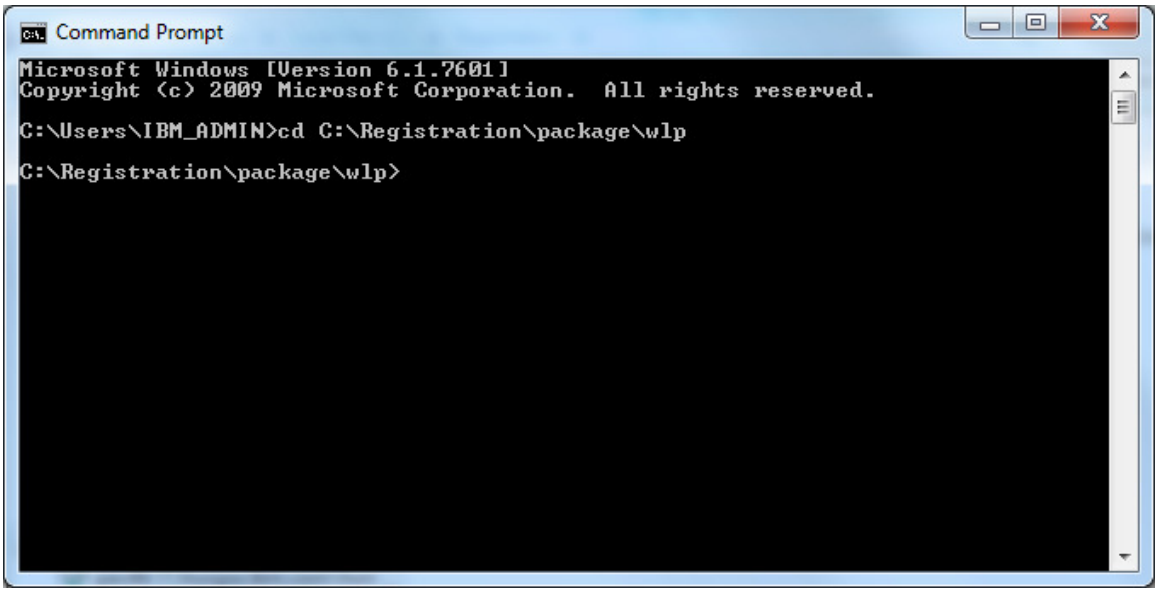


4. Once the packaging process is complete and the `package.zip` file is created, use Windows Explorer to navigate to the folder `package.zip` was created. Right click on the file and select **Extract All**.

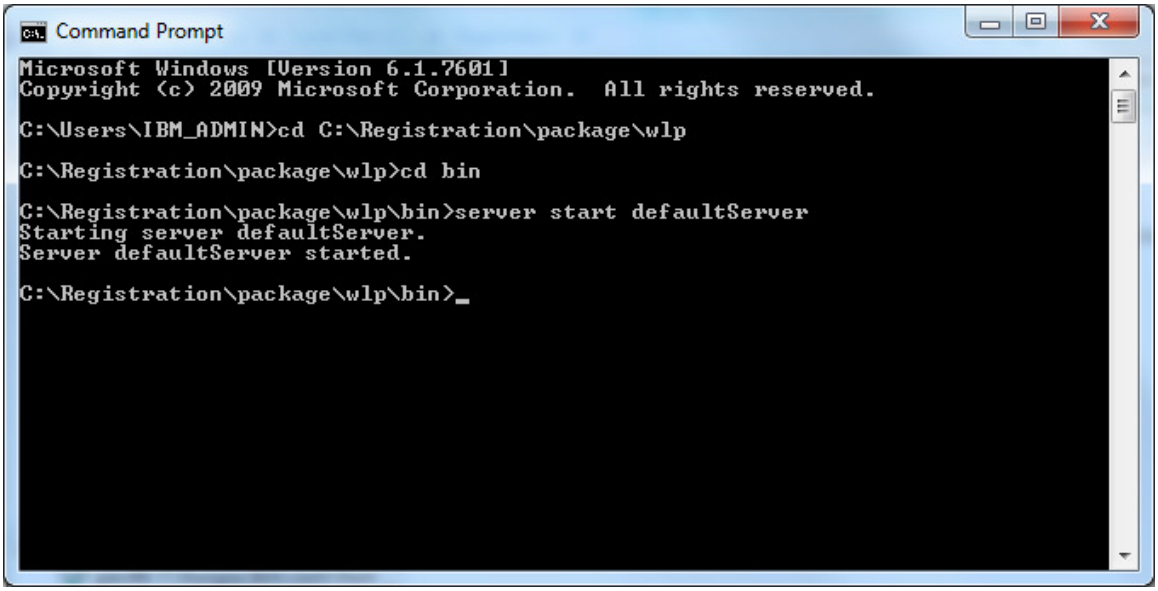


5. Follow the instructions to extract the zip file.

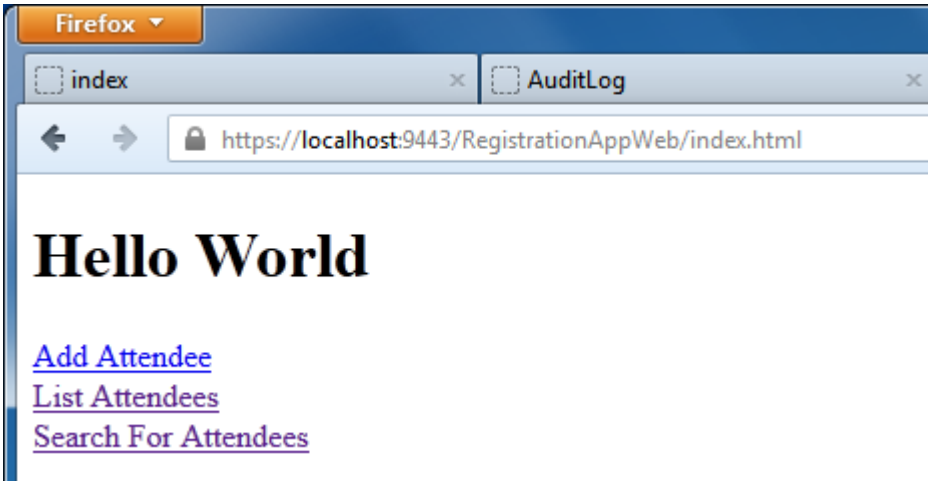
6. Open a command prompt, from the start menu and navigate to the location where the `package.zip` file was extracted to.



7. Navigate to the `bin` directory inside the extract location, and then run the command **server start defaultServer**.



8. Navigate to http://localhost:9080/RegistrationAppWeb. This will redirect to the secure port. You are now accessing your application from the packaged minified server.



Summary

In this lab you learned:

- How to install the IBM WebSphere Application Server V8.5.5 Liberty profile
- How to create and deploy a simple web application using the IBM WebSphere Application Server Developer Tools for Eclipse V8.5.5.
- How to create and deploy a simple registration web application that uses Servlets, JPA, EJBs, Context and Dependency Injection, and JAX-RS.
- How to secure applications to use SSL.

- How to generate a customised Liberty profile image, and use that image to deploy your application outside of a development environment.

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.