

CT5106

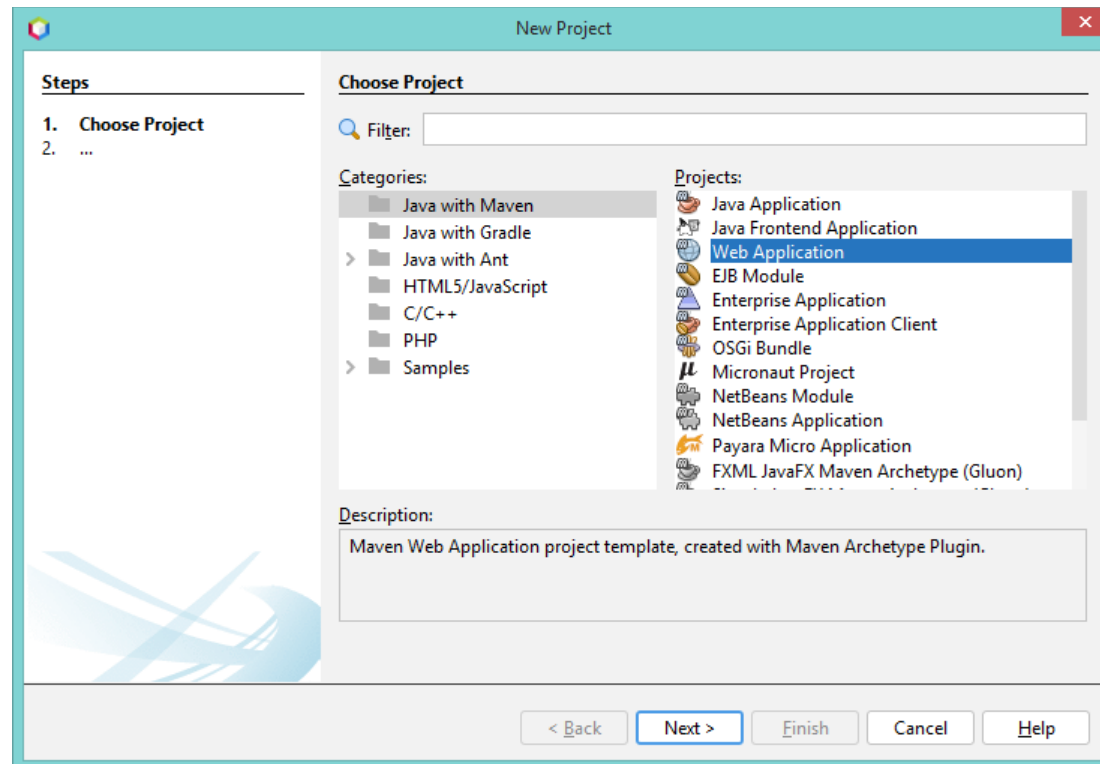
Java Server Faces (JSF)

JSF Overview

- JSF is an MVC framework for web applications
- Focused around connecting UI components / widgets with data sources and (server-side) event handlers
- JSF renders the UI components properly for the client type (in our case the web browser)

Setting up JSF on Apache NetBeans

- File / New Project / Java with Maven / Web Application




- Enter project name, and accept other defaults

The screenshot shows the 'New Web Application' dialog box with the 'Name and Location' tab selected. The 'Steps' panel on the left lists three steps: 1. Choose Project, 2. Name and Location (which is bolded), and 3. Settings. The 'Name and Location' tab contains several input fields: 'Project Name' with the value 'JSF1', 'Project Location' with the value 'C:\Users\o_molloy\Documents\NetBeansProjects' and a 'Browse...' button, 'Project Folder' with the value 'C:\Users\o_molloy\Documents\NetBeansProjects\JSF1', 'Artifact Id' with the value 'JSF1', 'Group Id' with the value 'com.mycompany', 'Version' with the value '1.0-SNAPSHOT', and 'Package' with the value 'com.mycompany.jsf1'. The 'Package' field is marked as '(Optional)'. At the bottom of the dialog, there are five buttons: '< Back', 'Next >' (which is highlighted with a blue border), 'Finish', 'Cancel', and 'Help'.

Steps	Name and Location
1. Choose Project	Project Name: JSF1
2. Name and Location	Project Location: C:\Users\o_molloy\Documents\NetBeansProjects <input type="button" value="Browse..."/>
3. Settings	Project Folder: C:\Users\o_molloy\Documents\NetBeansProjects\JSF1
	Artifact Id: JSF1
	Group Id: com.mycompany
	Version: 1.0-SNAPSHOT
	Package: com.mycompany.jsf1 (Optional)

< Back **Next >** Finish Cancel Help

New Web Application✕


Steps

1. Choose Project


2. Name and Location


3. **Settings**

Settings

Server: Payara Server 

Add...

Java EE Version: Java EE 8 Web 



< Back

Next >

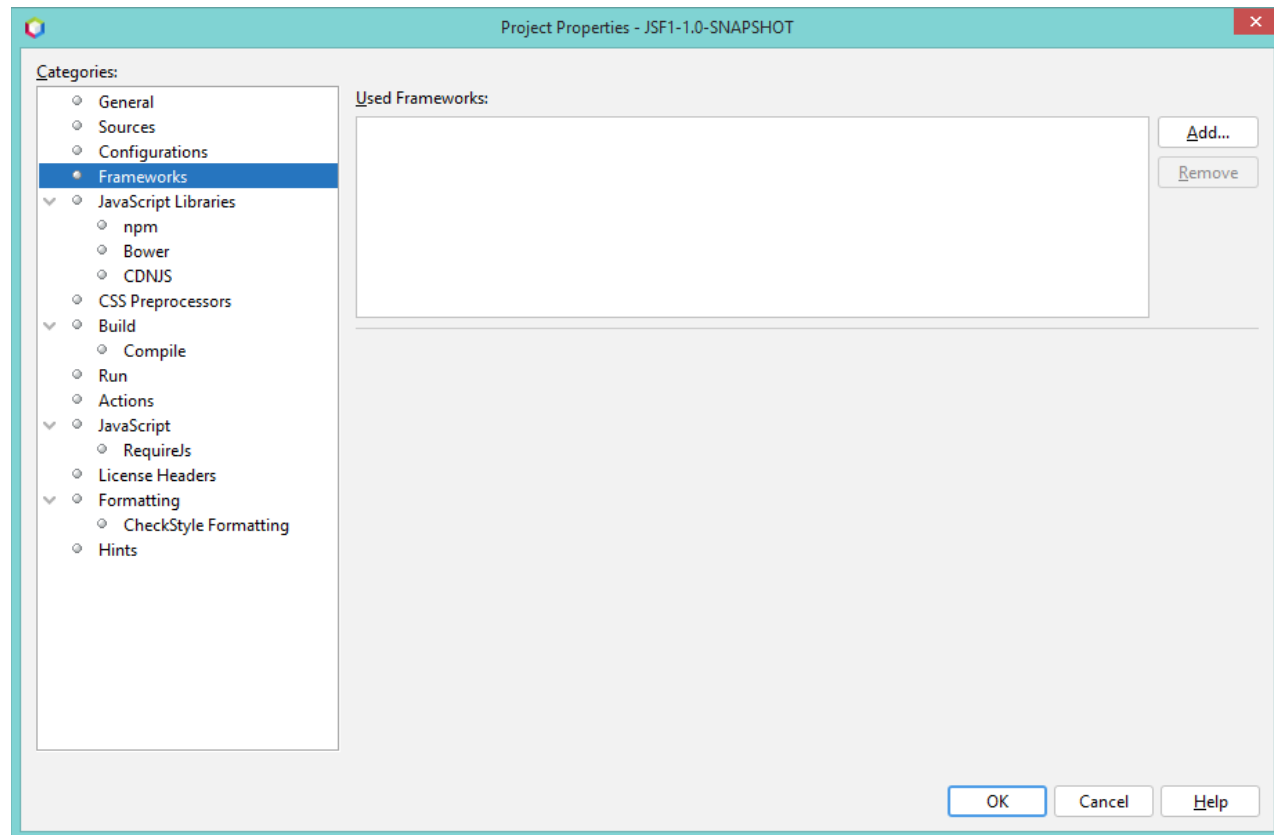
Finish

Cancel

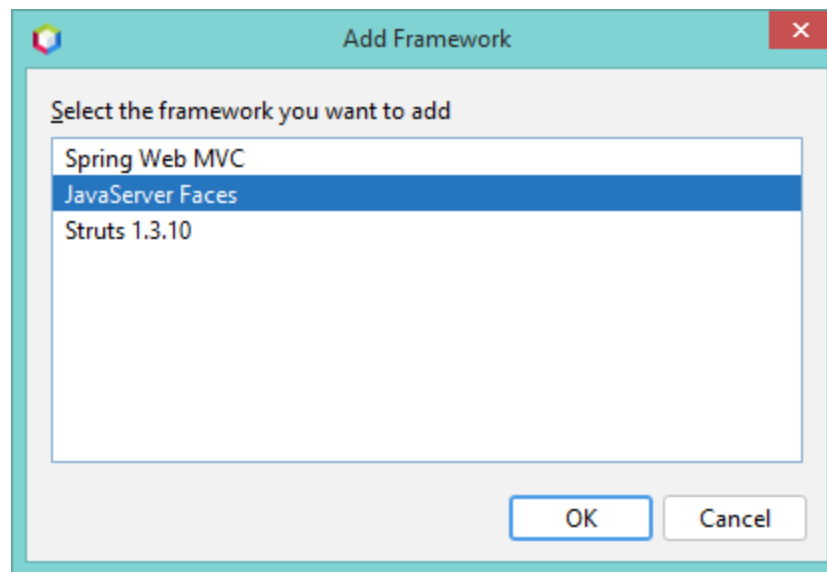
Help

Project Properties

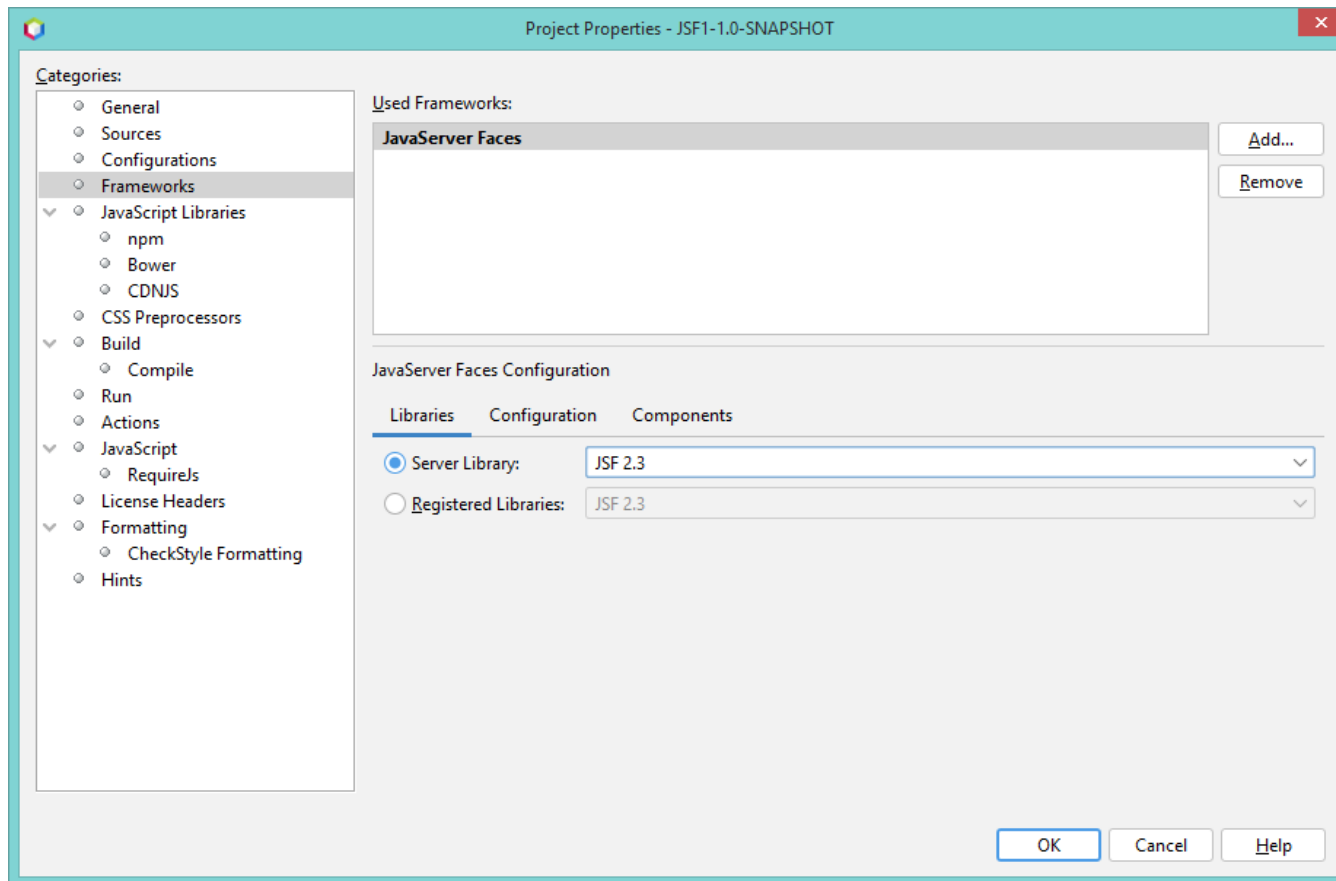
- ❑ Right-click on project
- ❑ Select 'Frameworks' and click on 'Add...' button



- Select JSF framework to add



□ OK



When you run you might get a serious error

□ Which contains the message

Failed to execute goal org.apache.maven.plugins:maven-war-plugin:2.3:war (default-war) on project JSF1: Execution default-war of goal org.apache.maven.plugins:maven-war-plugin:2.3:war failed: Unable to load the mojo 'war' in the plugin 'org.apache.maven.plugins:maven-war-plugin:2.3' due to an API incompatibility: org.codehaus.plexus.component.repository.exception.ComponentLookupException: null

Update the maven plugin

- In the pom.xml file, edit the following and change the plugin version to 3.3.2

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>3.3.2</version>
  <configuration>
    <failOnMissingWebXml>>false</failOnMissingWebXml>
  </configuration>
</plugin>
```

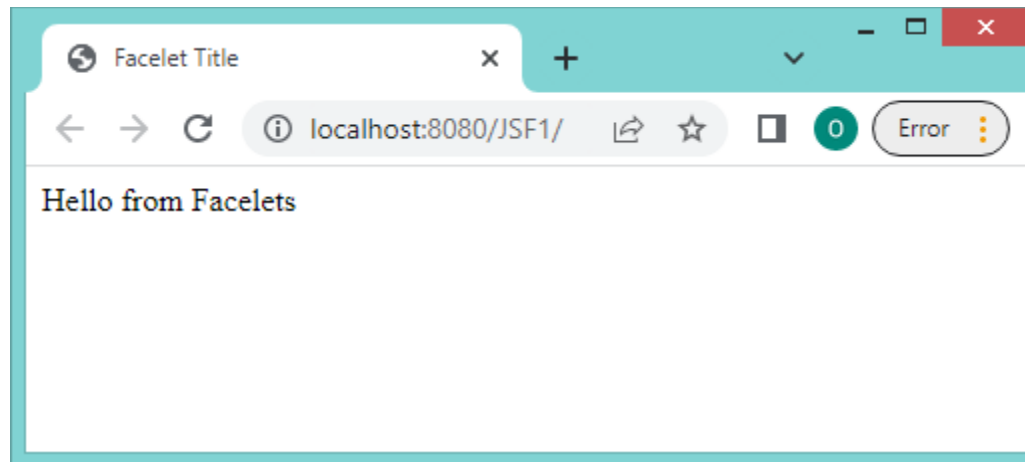
Check on config RE faces servlet

- To run incoming JSF requests, the web.xml file needs to contain instructions to
 - ▣ Load the Faces Servlet on startup
 - ▣ Route incoming requests with the pattern `/faces/*` to the Faces Servlet
- So your web.xml should have these lines in it

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

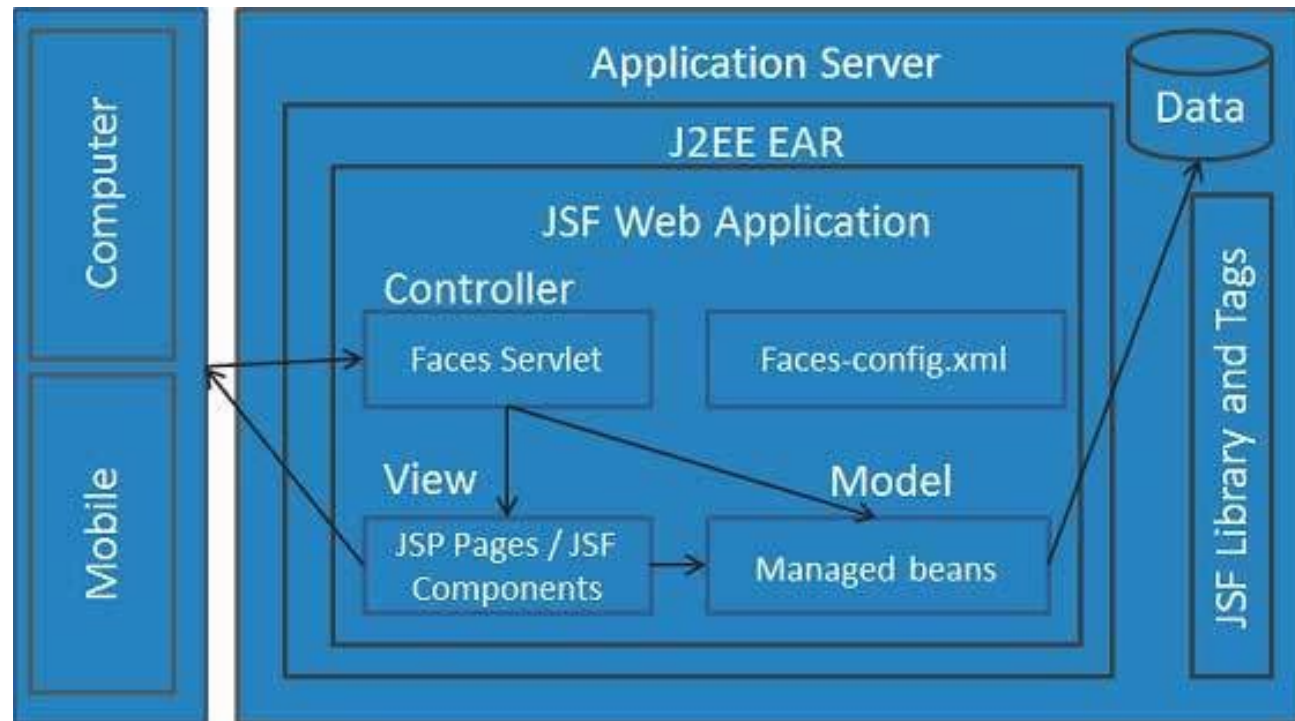
Then build again

- Right-click on project and select 'Build with Dependencies'
- If you see 'BUILD SUCCESS' then it is OK
- Then run the project – should see this in browser:



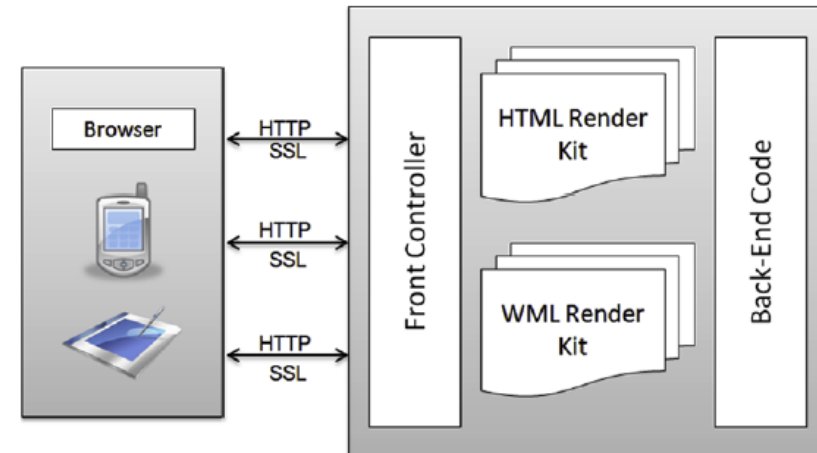
JSF Architecture

- Again, it is an MVC architecture, where the model is a *managed bean*, which in turn may interact with DAO classes or EJB's (e.g. session beans acting as facades for entity classes) which are the real underlying model in our application, but in JSF the managed beans are also considered part of the model



FacesServlet

- JSF has a front controller servlet called FacesServlet
- FacesServlet performs the role of brokering the incoming requests from clients to the right place
As mentioned earlier, JSF comes with reusable Web components that can be used to develop user interfaces
- These UI components can be associated with objects called *managed beans*
- *These managed beans handle the logic for the application and interact with back-end systems or components like EJBs*
- Each UI component in JSF can be associated with a different render kit that can generate different markup, such as HTML or WML (Wireless Markup Language), onto different types of devices



Finishing last week's example

- The project called 'intro-to-jsf'

inputComponents.xhtml

- This page demonstrates how to pull data from a managed bean into user interface components such as radio buttons, list boxes,...
- Note the use of the value property of the input components to tie the selected item to a property of the backing bean

The screenshot shows a web browser window with the title 'Facelet Title'. The address bar shows 'localhost:8080...'. The page content includes:

- A label 'Please enter a password' followed by a password input field with masked characters '.....'.
- A label 'This is a text area' followed by a text area containing the text 'sdfsd sdfsd'.
- A label 'This is a selectOneRadio' followed by a group of radio buttons with labels: 'Kheer', 'Ice Cream' (selected), 'Fudge Sundae', 'Apple Pie', 'Kheer', and 'Candied Apple'.
- A label 'This is a selectOneList' followed by a list box showing a list of items: 'Kheer', 'Ice Cream', 'Fudge Sundae' (highlighted), 'Apple Pie', 'Kheer', and 'Candied Apple'.

JSF *selectOneRadio* component

- When an item is selected, it's *itemValue* is placed in the variable *selectOneRadioSelection* in the backing bean *componentListing* (this is the java file *InputComponentPageBean.java*)
- The items in the list are populated from the list of *Dessert* objects (*desserts*) *@ApplicationScoped* bean in *DessertLoaderDAO.java*
- Each item takes the *dessertName* and value (*id*) from an item in the list

```
<h:outputLabel value="This is a selectOneRadio"/>
    <h:selectOneRadio value="#{componentListing.selectOneRadioSelection}">
        <f:selectItems value="#{dessertDao.desserts}" var="dessert"
            itemLabel="#{dessert.dessertName}"
            itemValue="#{dessert.dessertId}"/>
    </h:selectOneRadio>
```

- The functionality here is the same, apart from it being a list box instead of a radio box, so using a different JSF component (*selectOneListBox*)

```
<h:outputLabel value="This is a selectOneList"/>
    <h:selectOneListbox value="#{componentListing.selectListBoxSelection}">
        <f:selectItems value="#{dessertDao.desserts}" var="dessert"
            itemLabel="#{dessert.dessertName}"
            itemValue="#{dessert.dessertId}"/>
    </h:selectOneListbox>
```

```
@ApplicationScoped
@Named("dessertDao")
public class DessertLoaderDAO {

    private List<Dessert> desserts;

    @PostConstruct
    public void loadDesserts() {
        Dessert iceCream = new Dessert("Ice Cream", 10001);
        Dessert fudgeSundae = new Dessert("Fudge Sundae", 10002);
        Dessert kheer = new Dessert("Kheer", 10003);
        Dessert applePie = new Dessert("Apple Pie", 10004);
        Dessert candiedApple = new Dessert("Candied Apple", 10006);

        desserts = new LinkedList();

        desserts.add(kheer);
        desserts.add(iceCream);
        desserts.add(fudgeSundae);
        desserts.add(applePie);
        desserts.add(kheer);
        desserts.add(candiedApple);

        System.out.println("Loaded desserts");
    }
}
```

- So the values (id's) of the selected items in the radio and list box are stored in the highlighted (in red) variables in this bean

```
@RequestScoped
@Named("componentListing")
public class InputComponentPageBean {

    private String password1;

    private long selectOneRadioSelection;

    private long selectListBoxSelection;
```

New example this week

- This is an example that brings together JPA persistence, Java Server Faces and the Session Bean facades for entity classes – a design pattern which we saw a few weeks ago
- This example also uses a few UI components from the PrimeFaces library, which is probably the most used commercial JSF library
 - ▣ <https://www.primefaces.org/>
 - ▣ This example is fairly simple on the PrimeFaces side of things
 - ▣ There is a more elaborate one you can look at here
 - <https://www.oracle.com/technical-resources/articles/java/java-primefaces.html>

Create new JSF application

- ❑ Start with the standard new 'Java with Maven' / 'Web Application' project
- ❑ Then right-click on the project
- ❑ Select 'Frameworks'
- ❑ If JSF not already in the 'Used Frameworks' window, then click on 'Add' button and select 'Java Server Faces'
- ❑ Then click on OK


Adding PrimeFaces dependency

- There are a couple of ways to do this (just do one!):
 1. Add the following code to your pom.xml (maven build file) in the `<dependencies>` group:

```
<dependency>
  <groupId>org.primefaces</groupId>
  <artifactId>primefaces</artifactId>
  <version>7.0</version>
</dependency>
```

2. Add a project dependency via the GUI
 1. Right-click on 'Dependencies'
 2. Select 'Add Dependency'
 3. Query for 'primefaces' as show on the following slide
 4. Expand `or.primefaces : primefaces`, select the jar file and 'Add'

Select jar and add

 Add Dependency ✕

Group ID:

Artifact ID:

Version: Scope: compile ▼

Type: Classifier:

Search

Open Projects


Dependency Management

Query:
(coordinate, class name, project name...)

Search Results:

▼ a

org.primefaces : primefaces

 7.0 [jar] - local

Add

Cancel

web.xml

- When you look at web.xml you should see that the Faces Servlet will be created and loaded on startup of the application:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- You will also see that all requests coming from pages with the following url pattern will be directed to the Faces Servlet:

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Now the purpose of the app

- This is a simple app – it manages a list of books in a relational database table
- So we can start with the entity (Book) class
- You will find it in the **data** package
- Note that we are using the database to generate the entity primary key (id)
- I find that doing this is fine, but to ensure the new entity you create using the entity manager actually has an id, you have to call the entity manager flush() method (same as committing the DB changes), as you can see in the entity façade parent class, AbstractFacade:

```
public void create(T entity) {  
    getEntityManager().persist(entity);  
    getEntityManager().flush();  
    //System.out.println("flushed create in abstract facade");  
}
```

Back to the entity class

- As you will see, the id is a *Long*, and is generated automatically by the database (whichever method it chooses)
- So the id is not created and set in the new Book entity until the entity manager persists it in the database

```
@Id  
@GeneratedValue(strategy = GenerationType.AUTO)  
private Long id;
```

Entity facade

- By using the façade pattern, we have very little work to do with the entity manager and JPA directly, although we could add our own business methods to the BookFacade class if we wished
- To generate the façade we simple use the NetBeans commands File / New File / Enterprise JavaBeans / Session Beans for Entity Classes

Retrieving the list of Books from the database

- As the books are being created / edited / deleted by one or more users, I use a `@RequestScoped` backing bean to retrieve the current list of books from the database
- It has to be request scoped rather than application scoped, so that the list is constantly refreshed from the DB

```
@Named("bookListBean")  
@RequestScoped  
public class BookList  
{
```

- This BookList class uses **dependency injection** to get access to a working object of class BookFacade to retrieve the list of books
- These are stored in a property booksList
- This is done as soon as the bean is created, using the @PostConstruct method

```
@EJB
private BookFacade bookFacade;

private List<Book> booksList;

@PostConstruct
public void postConstruct()
{
    booksList = bookFacade.findAll()
}

public List<Book> getBooksList()
{
    return booksList;
}
```

- This means that any view which references this backing bean, when it is rendered (displayed) will cause the `postConstruct` method to be called, which will make the books list immediately available to that view
- For example, the view *books.xhtml* is an example, so we can look at this next

books.xhtml

- This is the main page of the application, where the book list is displayed, along with options to
- Create new books
- Edit and delete books in the list
- Note at the top, we include the primefaces UI library
- - it's components are referred to using the p: prefix in the code

```
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:f="http://java.sun.com/jsf/core"  
      xmlns:h="http://java.sun.com/jsf/html"  
      xmlns:p="http://primefaces.org/ui">
```


□ The first such component is the p:dataList

- It takes the list of books from the backing bean we have just discussed and iterates over it using the command

```
<p:dataList value="#{bookListBean.booksList}" var="book" type="ordered">
```

- So every item in the list is called 'book' inside this loop as it iterates over the items in the list

- The loop / list is terminated by

```
</p:dataList>
```

So what else is happening in books.xhtml?

- This component creates a named component inside a container component (the dataList)

```
<f:facet name="header">  
    Book List  
</f:facet>
```

- Each 'book' item in the list has its id output in 2 links – one to 'Edit' and one to 'Delete'
 - ▣ The p:link component has an **outcome**, such as "edit" – this means that when this link is clicked, "edit.xhtml" is called
 - ▣ Notice also the 'bookId' request parameter, whose value is 'book.id' is passed with the request

```
<p:link value="Edit" outcome="edit">
    <f:param name="bookId" value="#{book.id}" />
</p:link>
```

```
<p:link value="Delete" outcome="delete">
    <f:param name="bookId" value="#{book.id}" />
</p:link>
```

edit.xhtml

- When you click on 'Edit' in books.xhtml, the request param 'bookId' is created and the request sent to "edit.xhtml" as explained in the previous slide
- In edit.xhtml this param is included in the form as a hidden param – it is retrieved from the request using {param.bookId}
- When the user edits the title and clicks on 'Update', the update method of the bookBean managed bean is called

```
<h:form>
  <p:panel header="Update Book">
    <p:panelGrid columns="1" layout="grid">
      <p:outputLabel for="book-title" value="Enter new book title"/>
      <p:inputText id="book-title" value="#{bookBean.title}"/>
      <p:commandButton value="Update" action="#{bookBean.update}"/>
    </p:panelGrid>
    <input type="hidden" name="bookId" value="#{param.bookId}"/>
  </p:panel>
</h:form>
```

bookBean

- This bean is created when the edit.xhtml page is rendered and it has a `@postConstruct` method
- This method uses the Book façade session bean to retrieve the Book object using the 'bookId' hidden param we saw in the previous slide

```
@Named("bookBean")
@RequestScoped
public class BookBean {

    @EJB
    private BookFacade bookFacade;

    private String title;
    private long id;
    private Book book;

    @PostConstruct
    public void postConstruct() {
        String bookIdParam = FacesContext.getCurrentInstance().getExternalContext().getRequestParameterMap().get("bookId");
        if (bookIdParam != null) {
            id = Integer.parseInt(bookIdParam);
            book = bookFacade.find(id);
            title = book.getTitle();
        }
    }
}
```

bookBean update method

- This method simply uses the edit (i.e. update) method of the façade object to change the title of the book object

```
public String update () {  
    book.setTitle(title);  
    bookFacade.edit(book);  
    return "success";  
}
```

- Note that it returns the string “success” – we now need to see what that does

faces-config.xml

- This file contains the navigation rules for the app
- It simply defines for a specific view where the request should be routed to based on a particular outcome
- So for *edit.xhtml*, when it returns “success”, we are routed to *books.html*, where the list of books is displayed again
- This mechanism could be used, for example, to route users to a failure page if they log in incorrectly

```
43 <navigation-rule>
44   <from-view-id>/edit.xhtml</from-view-id>
45   <navigation-case>
46     <from-outcome>success</from-outcome>
47     <to-view-id>/books.xhtml</to-view-id>
48   </navigation-case>
49 </navigation-rule>
50
51 <navigation-rule>
52   <from-view-id>/delete.xhtml</from-view-id>
53   <navigation-case>
54     <from-outcome>success</from-outcome>
55     <to-view-id>/books.xhtml</to-view-id>
56   </navigation-case>
57 </navigation-rule>
```

- A similar mechanism is used in books.xhtml if you look at the “Create Book” link

```
</h:head> </h:head>
<h:body>
    <h:link outcome="book" value="Create Book"/>
    <n:dataList value="#{bookListBean.bookList}"
```

- The outcome “book” means that we go directly to “*book.xhtml*” when the link is clicked

book.xhtml

- In this view, we simply get the inputs for a new book (the title, as the id is autogenerated), and call the `add()` method of the `bookBean` (`BookBean.java`)

```
public String add() {  
    Book newBook = new Book();  
    newBook.setTitle(title);  
    bookFacade.create(newBook);  
    System.out.println("in add title = " + title);  
    return "success";  
}
```

- As we can see, the `add()` method uses the `bookFacade create()` method to create the book

AbstractFacade *create* method

- When we call the create method of the BookFacade, you will notice that after it uses the Entity Manager to persist the new book (entity), we call the Entity Manager's *flush()* method
- This is because we need the changes in the EM cache to be flushed to the database (committed) – this forces the database to generate the ID for the new entity which is passed back to the EM, which updates the entity in its cache

```
public void create(T entity) {  
    getEntityManager().persist(entity);  
    getEntityManager().flush();  
    //System.out.println("flushed create in abstract facade");  
}
```