EJB3.1/Java EE 6

A short primer

Preface: Dependency Injection

Dependency Injections

- A good design principle:
  - **Code for interfaces, not for classes**
- Problem: building object is difficult
- We want objects to be independant one from the other (loose coupling; testability; easy to understand)
- if object A creates object B, then A depends on B.

Dependency Injection

- Example: a small logging system

```
public class ForumFacade {
  private MyLog log= new MyLog();
  public void saveMessage(Message m) {
    log.log("saving "+ m.getTitle());
    .....
  }
}
```

ForumFacade depends on MyLog : bad !

Dependency Injection

Code for interfaces, not classes

```
public interface LogIF {
   void log(String s);
}
```

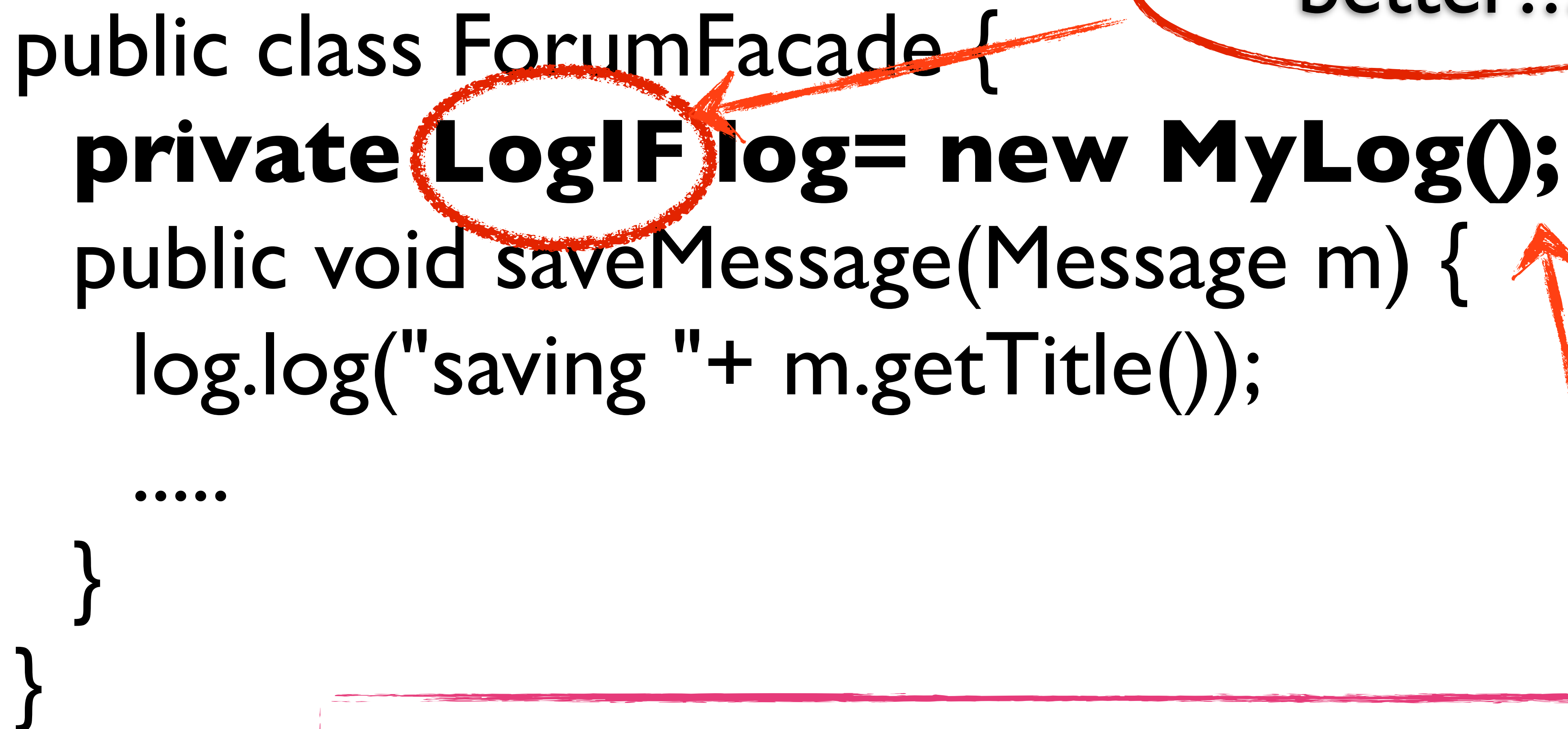
```
public class MyLog implements LogIF {
   public void log(String s) {
      System.err.println(s);
   }
}
```

```
public class NullLog implements LogIF {
   public void log(String s) {
      // do nothing
   }
}
```

```
public class DbLog implements LogIF {
   private MyJdbcConnection c;
   public void log(String s) {
```

Dependency Injection Coding for interface

- coding for interface:




```
public class ForumFacade {
  private LogIF log= new MyLog();
  public void saveMessage(Message m) {
    log.log("saving "+ m.getTitle());
    .....
  }
}
```

Dependency Injection

- Solution: don't do the «new» in ForumFacade

```
public class ForumFacade {
  private LogIF log;
  // WHO SETS LogIF ???
  public void saveMessage(Message m) {
    log.log("saving "+ m.getTitle());
    .....
  }
}
```

First solution: Repository

```
public class ForumFacade {
  private LogIF log= Repository.getLog();
  public void saveMessage(Message m) {
    log.log("saving "+ m.getTitle());
    .....
  }
}
```

Solution used by JDNI, etc...
a bit complex to configure,
still invasive

Second solution: Dependency Injection

```
public class ForumFacade {
  private LogIF log;
  public void setLog(LogIF log) {
    this.log= log;
  }
  public void saveMessage(Message m) {
    log.log("saving "+ m.getTitle());
    .....
  }
}
```

ok, but who calls this method??

ForumFacade doesn't depends on MyLog anymore

Dependency Injection

- Who calls the setter:
  - in general, the application
  - in EJB, the **container**
- Configuration:
  - usually through Java annotations
  - (other solutions include XML configuration files)

Dependency Injection

```
public class ForumFacade {
  private LogIF log;
  @Inject
  public void setLog(LogIF log) {
    this.log= log;
  }
  public void saveMessage(Message m) {
    log.log("saving "+ m.getTitle());
    .....
  }
}
```

Dependency injection

- Beware, only works for objects managed by the application server (if you create it with new, it's not the case)

Back to EJBs...

Application Server vs Servlet Container

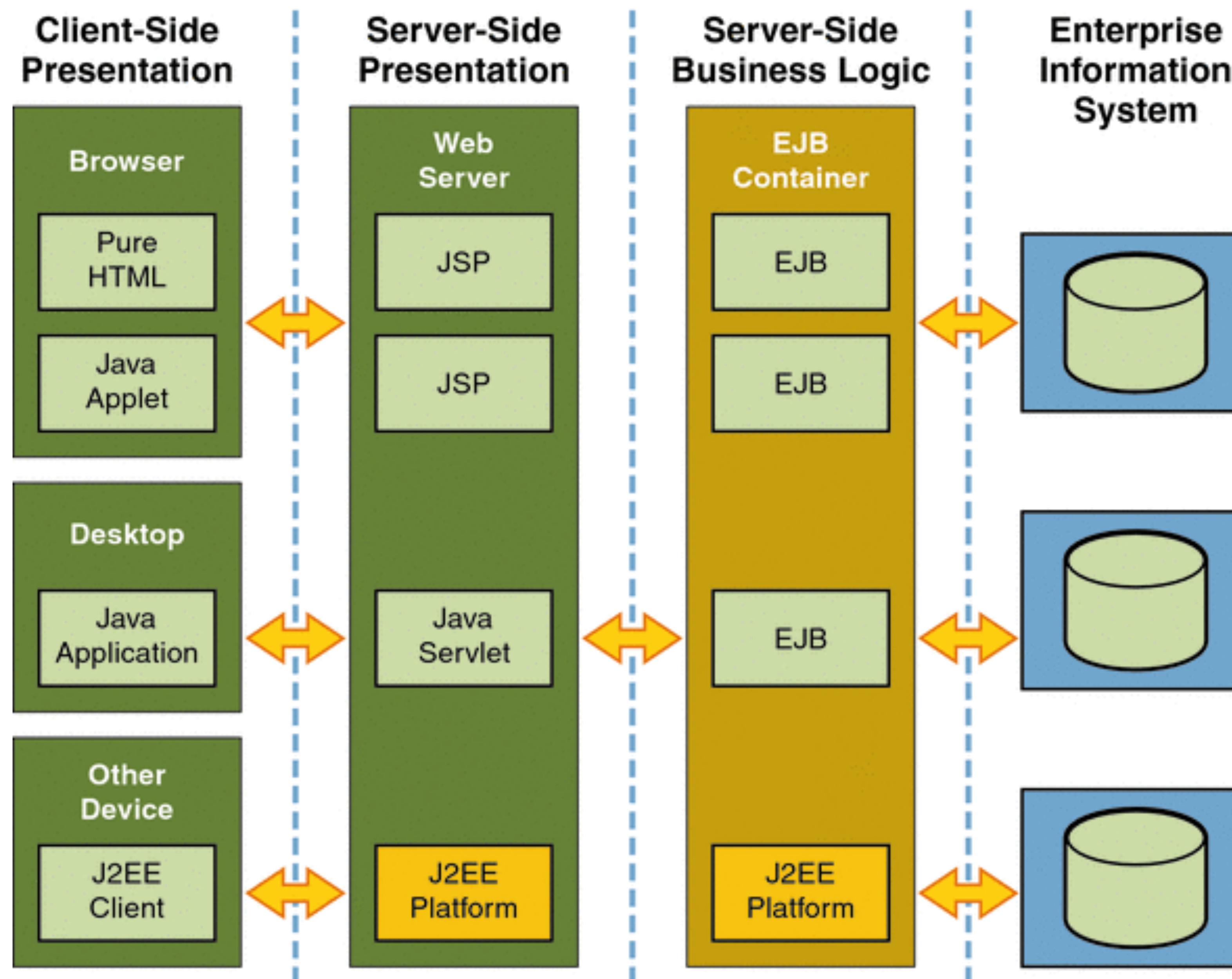
- **Servlet container**: Web server able to serve java Servlets and Jsps.
  - e.g. Tomcat, Jetty
- **Application server**: servlet container implementing the java EE stack, managing the life cycle of beans objects, and providing a number of facilities, like dependency injection, transaction management, JPA integration, messaging and JNDI repository...

A bit of history

- 1999: EJB 1.1/J2EE 1.2
  - «heavy» component container
  - lots of functionnalities
  - distributed programming
  - lots of boilerplate code
  - invasive: code written specifically for J2EE (e.g. Entity beans were not POJOs)
  - hard to test

A bit of history

- 2004: Spring framework, light, uses POJO and XML files, most functionnalities of EJB
- 2009: Java EE6/EJB 3.1: lots of ideas from Spring, Hibernate, etc.
  - application container comes with everything included



(from http://docs.oracle.com/cd/E19879-01/820-4343/abeat/index.html)

Enterprise Java Beans

- Objects managed by the application server
  - created, shared, etc... according to the choices of the application server
  - accessible by JNDI and Dependency injection
  - life cycle depends on the kind of bean
  - **You never call new to create an EJB!!!**

Stateless beans

- Objects
- Represents business methods
- Stateless : no information is kept between invocations
- Can be injected in Servlet (because they are stateless)

Injection of persistence context

- Database access is usually done through injection of persistence context in a Stateless bean.

```
@Local
public interface ForumIF {......}

@Stateless
public class ForumBean implements ForumIF {

    @PersistenceContext
    private EntityManager em;

    public Long addMessage(String titre, String contenu,
                     ForumUser author) {
        ... no need to open transaction here...
    }
   ....
```

```
@WebServlet(name = "Do", urlPatterns = {"/do"})
public class DoServlet extends HttpServlet {
  @EJB
  ForumIF forumBean;
  ...
}
```

Transaction management (real short primer, more later)

- The methods of Stateless and Statefull beans are, by default, Transactionnal: if they fail, the database manipulations are cancelled !

Let's work

- Let's work on the forum application once more
- Create a database, make the Message an entity, and change the ForumFacade into a stateless bean.
- Implement the code to list, display and add messages

Second Course

more on beans, and JSF

Stateful beans

- A stateful bean is meant to keep and store information in memory (like sessions or application beans in servlets)
- A Stateful bean is annotated with the @Stateful annotation
- A Stateful bean can have @Local and @Remote interfaces
- Stateful beans can be *passivated* (written to disk), and *their instance variable* should be serializable.
- Examples of stateful beans: basket, user information...

Singleton bean

- Annotated with @Singleton
- Exists in only one instance for each application.

JNDI
Java Naming and Directory Interface

- Provides unified access to various directory-like services (DNS, LDAP...)
- Directory: associates data (object) with names
- http://docs.oracle.com/javase/jndi/tutorial/
- JNDI access is provided by the application server

Beans access through JNDI

- Stateless beans and singletons can be injected in servlets with @EJB annotation
- In servlets, Stateful beans must be retrieved through JNDI, and «manually» stored in the session as session beans

Getting a beans with JNDI

```
import javax.naming.*;
....
....
Context c= new InitialContext();
String beanName= "java:app/MyApp/BasketBean";
BasketBean b= (BasketBean) c.lookup(beanName);
```

- If bean is a stateful bean, this creates a new bean
- If bean is a stateless bean, it *might* reuse an existing bean

Bean names (1)

- For beans in other web application, use global name:
  - java:global/**APP/MODULE/BEAN**
- For beans in other *modules* of an application (if the application is made of independent elements), use application name:
  - java:app/**MODULE/BEAN**
- For beans in the same module, you can use the module name:
  - java:module/**BEAN**
- where **APP=** application name; **MODULE** = module name; **Bean=** bean name (defaults to class name)
- you can add ‘ !**INTERFACE**’ after the module name, if more than one interface is possible.
- e.g. : java:app/SessionInViewTest/BasketBean!myshop.business.BasketIF"
- See http://glassfish.java.net/javaee5/ejb/EJB_FAQ.html#What_is_the_syntax_for_portable_global_ for details.

Summary

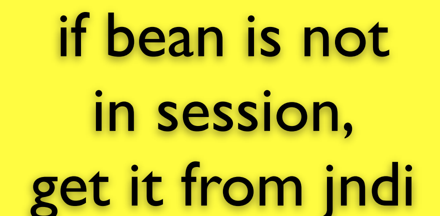
- Basket session bean in a servlet :

```
@Stateful
public class Basket {
    // Product is serializable !
    private List<Product> content=
            new ArrayList<Product> ();

    public void add(Product p) {
        content.add(p);
    }
   ....
```

Summary
Bean access in servlet

```
@WebServlet(name = "AddToBasketServlet", urlPatterns = {"/addToBasket"})
public class AddToBasketServlet extends HttpServlet {
        ...
    private Basket lookupBasketBean(HttpServletRequest request) {
        HttpSession session = request.getSession();
        if (session.getAttribute("basket") == null) {
            try {
                Context c = new InitialContext();
                Basket b = (Basket) c.lookup("java:global/SessionInViewTest/Basket");
                session.setAttribute("basket", b);
            } catch (NamingException ne) {
                Logger.getLogger(getClass().getName()).log(Level.SEVERE, "exception", ne);
                throw new RuntimeException(ne);
            }
        }
        return (Basket) session.getAttribute("basket");
    }
}
```



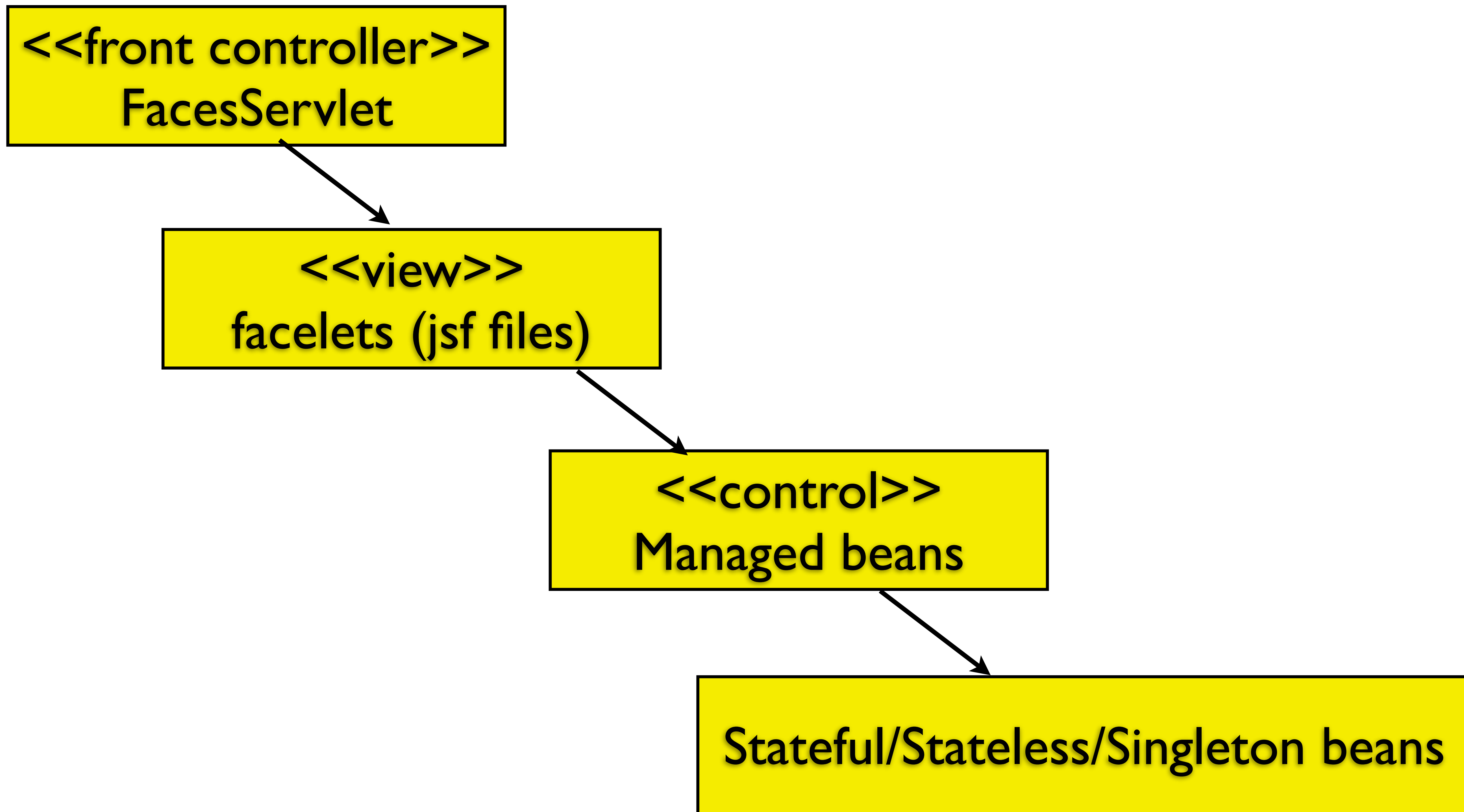
Exercice

- Add a user class to the forum. Users have a login, a password, and boolean tag telling if they are administrators or not.
- Implement user login/logout facilities in the forum
- Change the «message» class, so that messages have an author. Write the «add message servlet».
- add editing facilities : one can edit his own messages, and editors can edit everything.

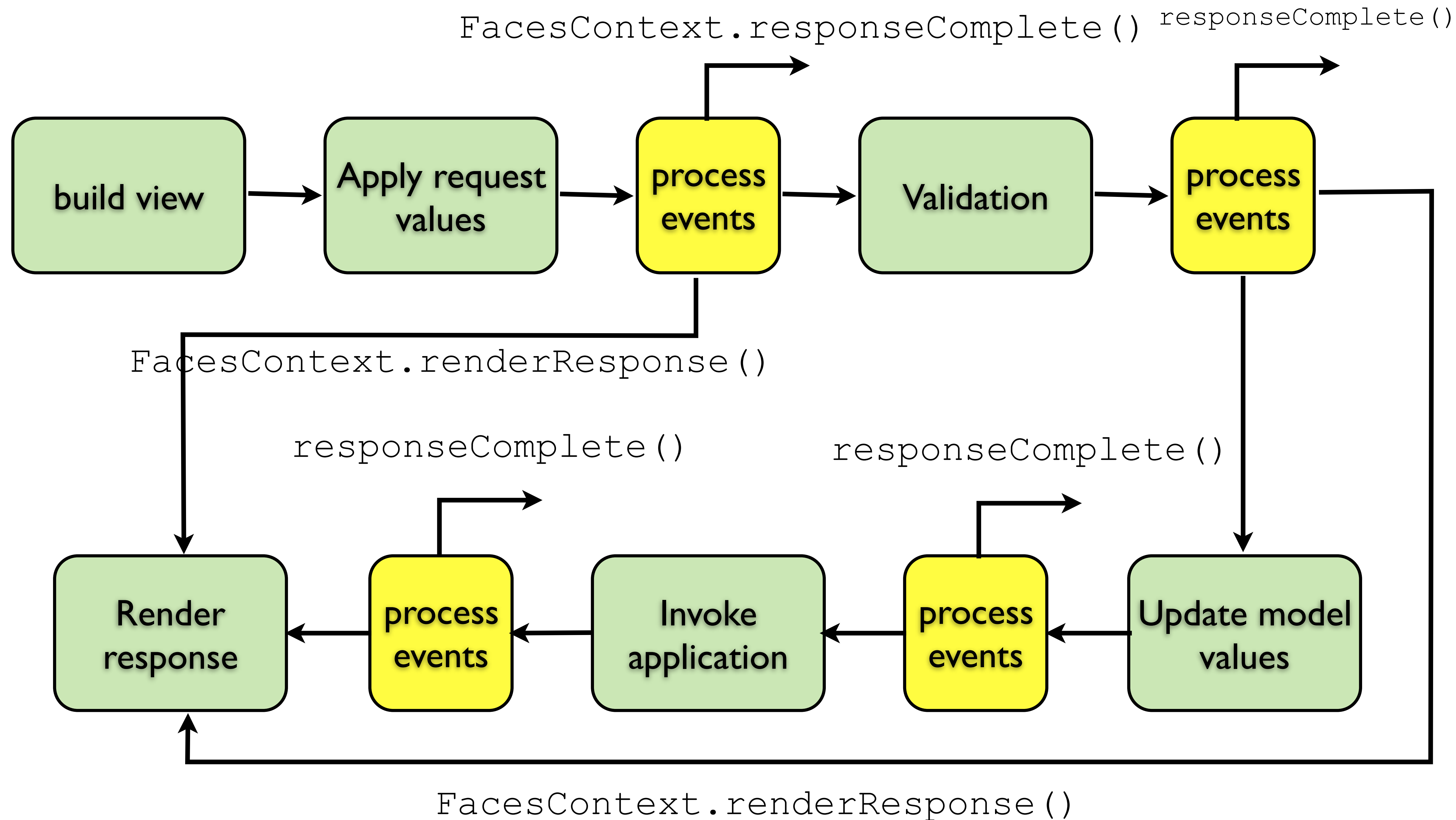
JSF

- Based on the servlet architecture
- A front controller is provided
- component-based: a JSF page produces a set of java component objects, which are then used to render the page
- easy access to beans through annotations
- manages navigation, data validation, ajax...

JSF architecture



Request processing (from oracle doc.)



Setting up JSF

- 1) Declare the front controller servlet : FacesServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <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>
  web-app>
```

Setting up JSF

- 2)(optionnal) create the faces-config.xml file in WEB-INF

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.1">
</faces-config>
```

Managed beans

- **Control** (part of the UI), not EJB
- Annotated with @ManagedBean
- Scope can be @SessionScoped, @RequestScoped, @ApplicationScoped, @ViewScoped
- Can access EJB with injection

facelets

- **View**
- look like JSP, but:
  - the XML file is parsed, and then an **object representation** of the document is built
  - the final representation (e.g. HTML content) is generated from both this data and managed beans content
  - (in JSP, there is no concept of intermediary object representation)
  - use ‘#’ instead of ‘$’ for beans access

Example

Managed bean

```
import javax.faces.bean.*;
@ManagedBean @SessionScoped
public class CounterControl {
    private int count=0;
    public int getCount() {
        return count++;
    }
}
```

Facelet

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">
    <h:head></h:head>
    <h:body>
    Count value: <h:outputText value="#{counterControl.count}"/>
    </h:body>
</html>
```

Forms in JSF

- embedded in the <h:form> tag
- usually backed up by a managed bean
- use JSF tags for form input
- action is performed by the managed bean.

Forms (simple example)

```
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">
   <h:head></h:head>
   <h:body>
        <h:form>
          <h:panelGrid columns="2">
              a <h:inputText value="#{addControl.first}"/>
              b <h:inputText value="#{addControl.second}"/>
              result <h:inputText value="#{addControl.result}"/>
              <h:commandButton value="add"
                                        action="#{addControl.add()}"/>
          </h:panelGrid>
       </h:form>
   </h:body>
</html>
```

links the control to a bean property.

method called when the button is clicked.

Form : backing bean

```
import javax.faces.bean.*;
@ManagedBean
@RequestScoped
public class AddControl {
    private int first, second, result;

    public void add() {
        result= first+ second;
    }
    .... accessors for properties ....

}
```

Forms

- data from the form is injected in the backing bean
- automated data validation ; error messages if the data is not correct
- redisplay of values if the form is not correct

Forms and error messages

```
<h:inputText label="a" id="a" value="#{addControl.first}"/>
<h:message for="a"/>
```

- id: used to identify the field
- label : name used in the error message
- for : refers to the id of a field.

JSF navigation

- the actions can return a String, which is used to choose the next view
- it's also possible to use a view name instead of an action
- the string might be the name of a view, or an abstract result (like «success» or «error») which is used to navigate in faces-config.xml.

Very simple example

(on page index.xhtml)

```
<h:link value="first example" outcome="add"/>
```

- creates a link with text «first example»
- will go to the «add» view, if it exists
- or : will search for a rule for outcome «add»

Example with navigation rule

in add2.xhtml

```
<h:form>
a <h:inputText label="a" id="a" value="#{addControl.first}"/>
b <h:inputText label="b" id="b"  value="#{addControl.second}"/>
 <h:commandButton value="add" action="#{addControl.add2()}"/>
 </h:form>
```

in addResult.xhtml

```
<h:body>
 Addition result :
  <h:outputText value="#{addControl.result}"/>
  <h:link value="back to index" outcome="index"/>
 </h:body>
```

managed bean:

```
@ManagedBean
@RequestScoped
public class AddControl {
    private int first, second, result;
     ....
     public String add2() {
        result= first+ second;
        return "success";
    }
}
```

Example with navigation rule

```
<navigation-rule>
    <from-view-id>/add2.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/addResult.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

- when an action from add2.xhtml returns «success», display view addResult.xhtml.
- separates navigation from direct action result.

Navigation rules

- flow can be displayed by netbeans

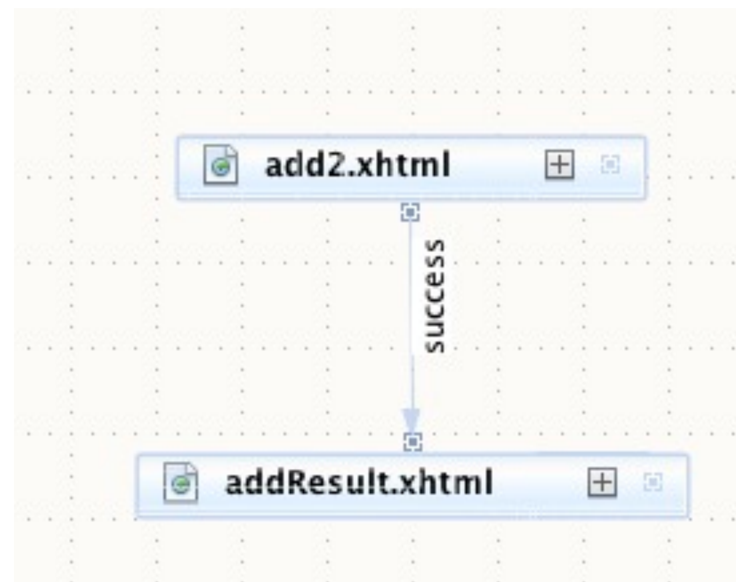


Table display in JSF

```
<h:dataTable var="p" value="#{personControl.personList}">
      <h:column>
        <f:facet name="header">
          <h:outputText value="Surname"/>
        </f:facet>
        <h:outputText value="#{p.surname}"/>
      </h:column>
      <h:column>
        <f:facet name="header">
          <h:outputText value="Name"/>
        </f:facet>
        <h:outputText value="#{p.name}"/>
      </h:column>
</h:dataTable>
```

Conditionnal display in JSF

- easy: JSF tags have a «rendered» attribute, which is a boolean.

```
<h:outputText rendered="#{empty personControl.personList}"
                    value="The person list is empty"/>
<h:panelGroup
        rendered="#{not empty personControl.personList}">
    The person list contains
   <h:outputText value="#{personControl.personList.size()}"/>
   persons.
</h:panelGroup>
```

JSF and ajax

- Ajax is quite simple with JSF. Use jsf:ajax.

```
<h:form>
  a <h:inputText label="a" id="a" value="#{addControl.first}"/>
  b <h:inputText label="b" id="b"  value="#{addControl.second}"/>
  result <h:outputText id="res" value="#{addControl.result}"/>
  <h:commandButton value="add" action="#{addControl.add()}">
    <f:ajax execute="@form" render="res"/>
  </h:commandButton>
</h:form>
```

Ajax

- f:ajax associates ajax behaviour to some javascript events (submit for form,
- the f:ajax tag is placed inside a component, or around a number of components.
- attributes
  - event : depends on the component.
  - render: what element is to be re-rendered ? (an id, or @form, @this, @all...)
  - execute : which elements values are to be transmitted to the beans ? @this, @all, @none, @form..., ids (defaults to @this)

Injecting request parameters in a bean

- For request beans only
- annotate the property in the bean with
  - @ManagedProperty(value="#{param.*NAME*}")
  - where *NAME* is the name of the property

GET methods in JSF

- to force GET method use, use h:button or h:link.

FacesContext

- Allows access to Request objet, session, etc.
- Get with

  FacesContext ctx=
  FacesContext.getCurrentInstance();

- ctx.getExternalContext() provides access to the session, request, etc.

Templates

- A template system allows you to create modular views.

External JSF component libraries

- Example: http://www.primefaces.org/
- ICEFaces
- RichFaces