Anno Accademico 2007-2008

Corso di Tecnologie Web

Web Application: Java Server Faces

http://www-lia.deis.unibo.it/Courses/TecnologieWeb0708/

Java Server Faces

> Java Server Faces definisce :

- un'architettura a componenti
- un set standard di componenti di *User Interface* (widget)
- un'infrastruttura applicativa per i diversi componenti

> I componenti sono *event oriented*: JSF consente di gestire eventi generati sul client che necessitano di una elaborazione server side (senza la manipolazione diretta di *HttpRequest* e *HttpResponse*)

> La *Java Community Process* (JCP) con l'incarico di definire lo standard per JSF (inizialmente definito da *Java Specification Request* 127) ha stabilito un insieme di API per creare GUI mediante Web Application Java seguendo le direttive sopra citate. La prima release delle specifiche è stata rilasciata nel 2003; la versione attuale è del 2006 (JSF 1.2 JSR-252).

JSF: tecnologie precedenti

> **Servlet**: sono il fondamento delle web application nell'architettura J2EE. L'approccio a basso livello per la produzione del *markup* (lo sviluppo di una pagina web richiede, sostanzialmente, di codificare un programma che produca come *output* i tag che il browser riceve e interpreta) rappresenta il limite principale all'adozione di tale tecnologia per la generazione massiva di GUI web. Le servlet API sono la base di ogni web application J2EE : ciò implica che, essendo un'applicazione JSF una web application, le servlet sono la tecnologia fondante di JSF. Le servlet API forniscono un insieme di funzionalità di base (session tracking, security, logging, filtering, lifecycle events, ecc) di cui si giova il framework JSF

> **JSP**: rappresentano il meccanismo di *template* standard definito nell'architettura J2EE. Le pagine JSP sono orientate ai *tag*: utilizzano, cioè, oltre ai consueti marcatori HTML i cosiddetti *custom tag* per introdurre comportamento dinamico nella pagina. Di più, consentono di inserire nella pagina codice Java *embedded* alternato ai tag descritti.

JSF: tecnologie precedenti

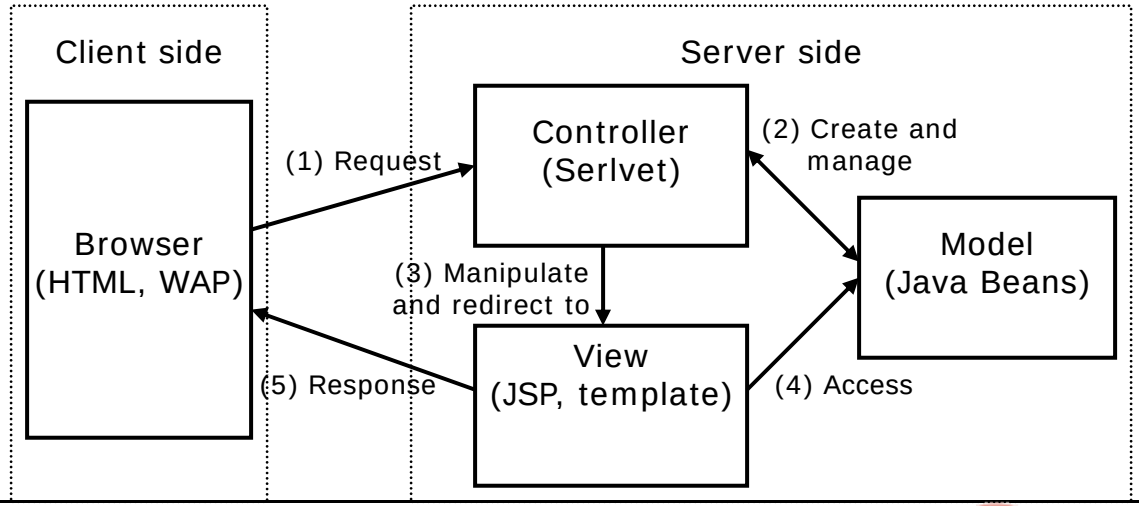
> **XML/XSLT**: un altro approccio per la produzione di pagine web è rappresentato da Extensible Markup Language/ Extensible Stylesheet Language Transformation. Queste tecnologie consentono di separare i dati di presentazione (contenuti nel documento XML) dalla struttura della pagina e dallo stile di presentazione (definito dal foglio di stile XSL). Tale meccanismo può essere implementato avvalendosi direttamente delle API XML incluse nella distribuzione J2EE, oppure adottando framework quali Cocoon (ASF Cocoon) o, utilizzando un approccio più strettamente *template based,* Velocity (ASF Cocoon) e WebMacro.

Model View Controller design pattern

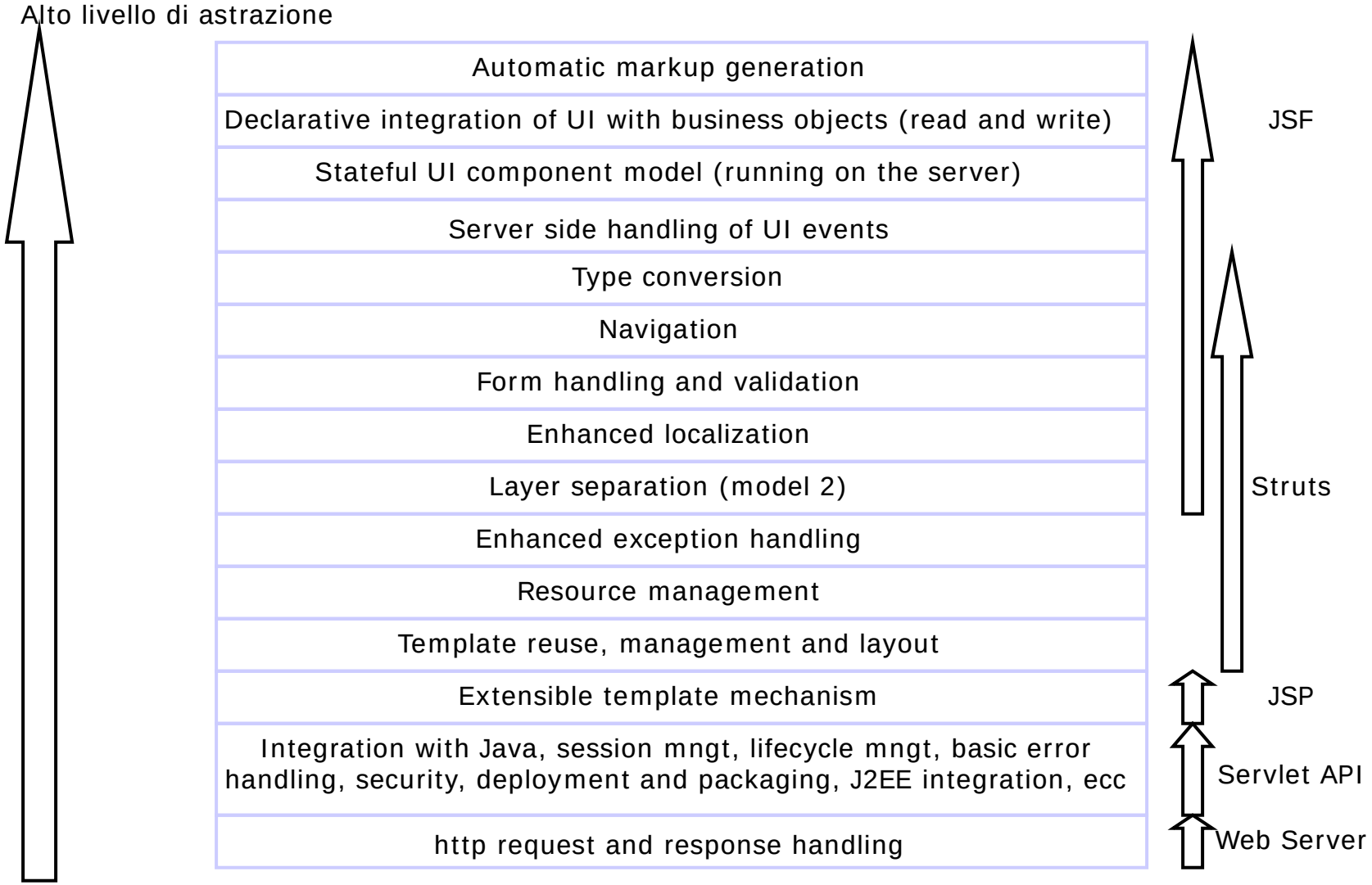
> Model View Controller è un design pattern largamente utilizzato nella realizzazione di User Interface. La variante per le web application Java è detta ***model 2***

> Secondo questo paradigma architetturale:

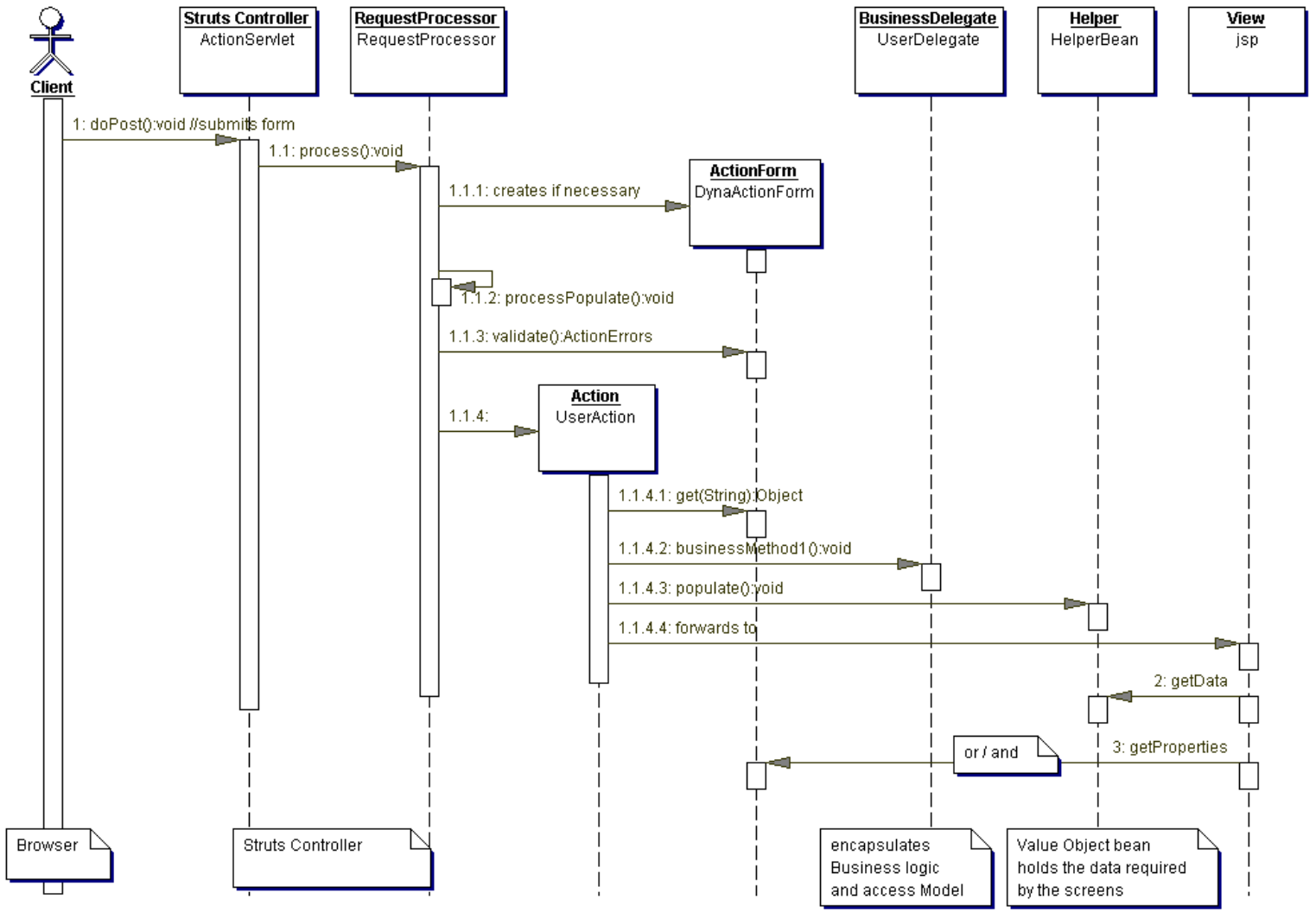
- la vista esprime il tier di presentazione che garantisce ed è responsabile dell'interazione con l'utente
- il modello rappresenta i dati e la logica di business del dominio applicativo
- il controller è il componente che risponde agli eventi generati dall'utente e integra modello e vista



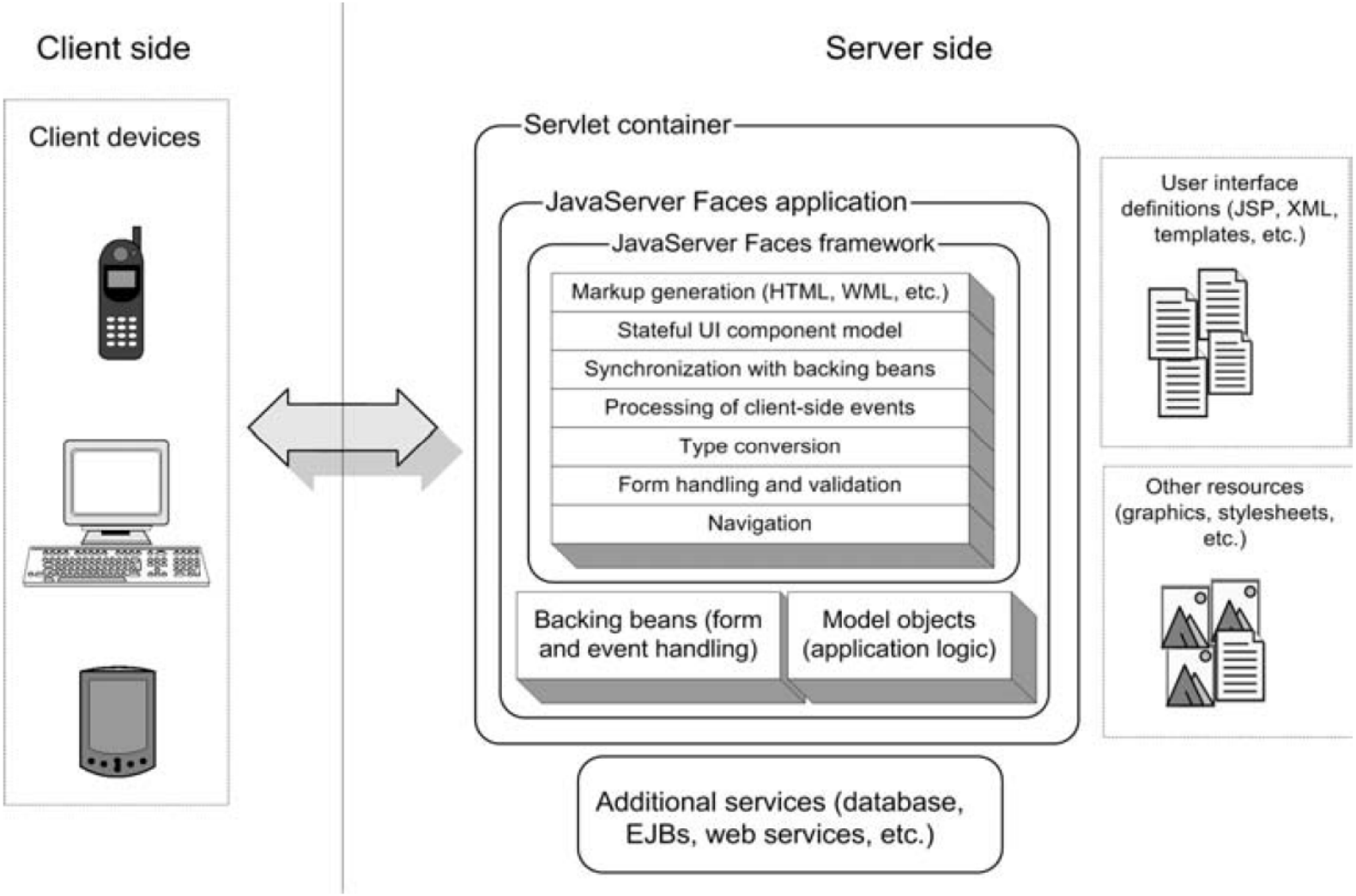
Web application Java: l'evoluzione



MVC nelle web application: model 2 e STRUTS



JSF: visione ad alto livello dell'architettura JSF



JSF: un semplice esempio

```
 1. <html>
 2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
 4.
 5.    <f:view>
 6.       <head>
 7.          <title>A Simple JavaServer Faces Application</title>
 8.       </head>
 9.       <body>
10.         <h:form>
11.            <h3>Please enter your name and password</h3>
12.            <table>
13.               <tr>
14.                  <td>Name:</td>
15.                  <td>
16.                     <h:inputText value="#{user.name}"/>
17.                  </td>
18.               </tr>
19.               <tr>
20.                  <td>Password:</td>
21.                  <td>
22.                     <h:inputSecret value="#{user.password}"/>
23.                  </td>
24.               </tr>
25.            </table>
26.            <p>
27.               <h:commandButton value="Login" action="login"/>
28.            </p>
29.         </h:form>
30.      </body>
31.   </f:view>
32. </html>
```

JSF: un semplice esempio

JSF: un semplice esempio

> Nell'esempio, un certo numero di tag è rappresentato da standard HTML tag: **`body, table, ecc.`**

> Alcuni tag hanno prefissi, come **`f:view`** e **`h:inputText.`** Si tratta di JSF tag. Le due direttive **`taglib`** dichiarano le tag libraries JSF.

> I tag **`h:inputText`**, **`h:inputSecret`** e **`h:commandButton`** corrispondono rispettivamente a un campo di testo, un campo password e un bottone di submit

> I campi di input sono collegati a proprietà di oggetti. Per esempio, l'attributo **`value="#{user.name}"`** consente, avvalendosi del framework JSF, di collegare il campo di testo con la proprietà **`name`** dell'oggetto **`user`**

> Tutti i tag JSF sono inclusi in **`f:view.`** In luogo del tag **`form`** HTML si utilizza **`h:form`** e in esso si includono tutti i componenti JSF

> In luogo dei consueti **`input`** tag HTML si utilizzano **`h:inputText, h:inputSecret, h:commandButton`**

JSF: analisi del codice

> Gli ingredienti di una applicazione JSF (e, in particolare, di quella d'esempio) sono:

- pagine che definiscono le varie schermate. Nell'esempio queste pagine sono **`index.jsp`** e **`welcome.jsp`**
- uno o più bean che gestiscono i dati utente (nel nostro caso nome e password). Il bean è una classe Java che espone properties seguendo una semplice *naming convention* per i metodi *getter* e *setter* (*accessor* e *mutator*)
- un file di configurazione per l'applicazione che configura i bean utilizzati e esprime le regole di navigazione. Di default questo file è chiamato: **`faces-config.xml`**
- normali file per la gestione della web application: web.xml ed eventuali altre risorse di configurazione

> Anche applicazioni più avanzate hanno la stessa struttura, ma possono contenere classi Java addizionali come *event handler*, *validatori* e componenti custom

JSF: bean

> Le properties del bean sono *name* e *password*. E' lecito che qualcuna di queste properties possa essere solo leggibile o, viceversa, solo scrivibile. Tale risultato si ottiene omettendo i corrispondenti metodi di accesso e/o di modificazione (getter e setter).

```
 1. package it.unibo.deis;
 2.
 3. public class UserBean {
 4.    private String name;
 5.    private String password;
 6.
 7.    // PROPERTY: name
 8.    public String getName() { return name; }
 9.    public void setName(String newValue) { name = newValue; }
10.
11.   // PROPERTY: password
12.   public String getPassword() { return password; }
13.   public void setPassword(String newValue) { password = newValue; }
14. }
```

JSF: la seconda pagina JSP

```
 1. <html>
 2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
 4.
 5.    <f:view>
 6.       <head>
 7.          <title>A Simple JavaServer Faces Application</title>
 8.       </head>
 9.       <body>
10.          <h3>
11.             Welcome to JavaServer Faces,
12.             <h:outputText value="#{user.name}"/>!
13.          </h3>
14.       </body>
15.    </f:view>
16. </html>
```

JSF: la seconda pagina JSP

JSF: le pagine JSP usando XML style

```
<?xml version="1.0" ?>
<jsp:root version="2.0"
   xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html">
   <f:view>
      <f:verbatim><![CDATA[<!DOCTYPE html
         PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
          transitional.dtd">]]>
      </f:verbatim>
      <html xmlns="http://www.w3.org/1999/xhtml">
         <head>
            <title>A Simple Java Server Faces Application</title>
         </head>
         <body>
            <h:form>
            . . .
            </h:form>
         </body>
      </html>
   </f:view>
</jsp:root>
```

JSF: la navigazione

> JSF, come STRUTS, consente di specificare dichiarativamente le regole di navigazione della web application in un opportuno config file:

```
<navigation-rule>
   <from-view-id>/index.jsp</from-view-id>
   <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/welcome.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

> Il contenuto dell'elemento **`from-outcome`** coincide col valore dell'attributo **`action`** nel tag: **`<h:commandButton value="Login" action="login"/>`**

> Oltre alle regole di navigazione, nel file di configurazione è presente anche la definizione del bean. Il nome del bean e delle sue proprietà coincidono con i nomi utilizzati nei componenti UI. I delimitatori **`#{…}`** definiscono una "value binding expression":

```
<h:inputText value="#{user.name}"/>
```

JSF: il file `faces-config.xml`

> Il file **`faces-config.xml`** contiene le regole di navigazione e le definizioni dei bean:

```
 1. <?xml version="1.0"?>
 2.
 3. <!DOCTYPE faces-config PUBLIC
 4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
 5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
 6.
 7. <faces-config>
 8.    <navigation-rule>
 9.       <from-view-id>/index.jsp</from-view-id>
10.       <navigation-case>
11.          <from-outcome>login</from-outcome>
12.          <to-view-id>/welcome.jsp</to-view-id>
13.       </navigation-case>
14.    </navigation-rule>
15.
16.    <managed-bean>
17.       <managed-bean-name>user</managed-bean-name>
18.       <managed-bean-class>it.unibo.deis.UserBean</managed-bean-class>
19.       <managed-bean-scope>session</managed-bean-scope>
20.    </managed-bean>
21. </faces-config>
```

JSF: servlet configuration

> Essendo l'applicazione JSF una web application J2EE, è necessario configurare l'applicazione mediante **`web.xml`** :

```
 1. <?xml version="1.0"?>
 2.
 3. <!DOCTYPE web-app PUBLIC
 4.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 5.    "http://java.sun.com/dtd/web-app_2_3.dtd">
 6.
 7. <web-app>
 8.    <servlet>
 9.       <servlet-name>Faces Servlet</servlet-name>
10.       <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
11.       <load-on-startup>1</load-on-startup>
12.    </servlet>
13.
14.    <servlet-mapping>
15.       <servlet-name>Faces Servlet</servlet-name>
16.       <url-pattern>*.faces</url-pattern>
17.    </servlet-mapping>
18.
19.    <welcome-file-list>
20.       <welcome-file>index.html</welcome-file>
21.    </welcome-file-list>
22. </web-app>
```

JSF: servlet configuration

> Sì può anche definire un *prefix mapping* in alternativa al mapping sull'estensione delle URL (*.faces).* In questo caso si utilizza, per la mappatura delle URL sul front controller JSF, il seguente frammento XML nel file **`web.xml`**:

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

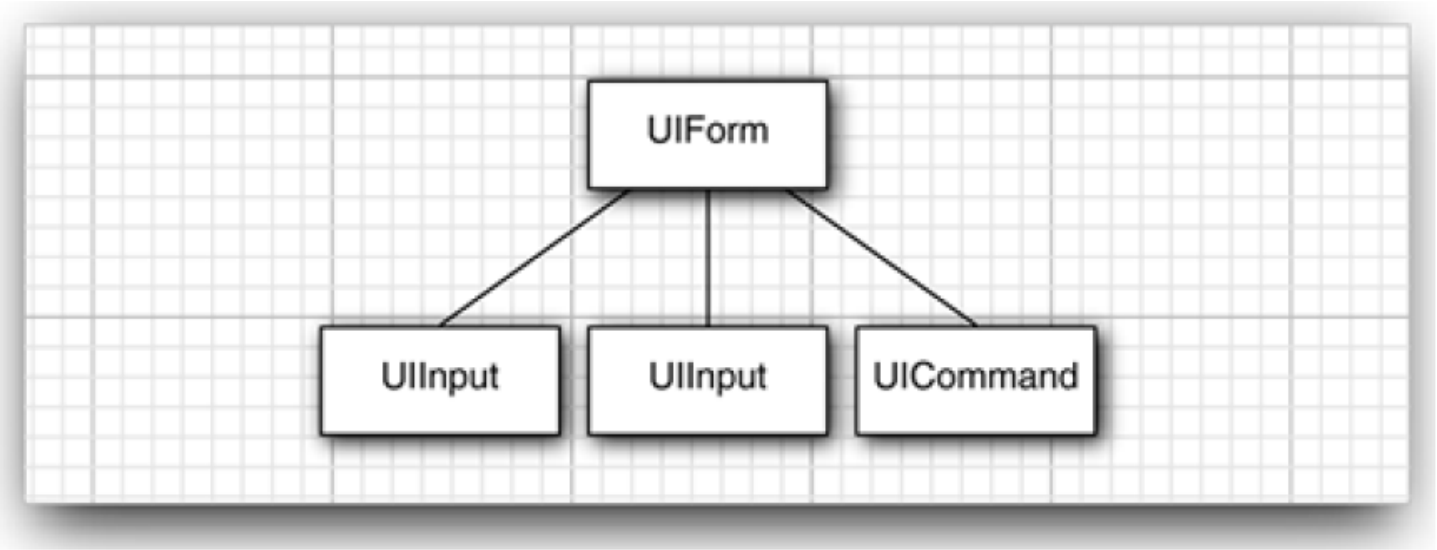
> Nel caso in cui si voglia utilizzare l'estensione *.jsf* per i file delle pagine, è necessario adottare qualche accorgimento sul file **`web.xml`**:

```
<servlet-mapping>
   <servlet-name>jsp</servlet-name>
   <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

```
<context-param>
   <param-name>javax.faces.DEFAULT_SUFFIX</param -name>
   <param-value>*.jsf</param-value>
</context-param>
```

JSF: dietro le quinte (component tree)

> Quando la pagina jsp (nel nostro caso index.jsp) viene invocata, i tag handler associati ai tag presenti nella pagina vengono eseguiti. I vari tag handler collaborano per realizzare il *component tree*:



> Il *component tree* è una struttura dati che contiene oggetti Java per tutti gli elementi UI di una pagina JSF.

JSF: dietro le quinte (rendering pages)

> Successivamente la pagina HTML viene *renderizzata*. Tutto ciò che non è un tag JSF viene scritto sull'*outputstream* o sull'apposito *writer*: si tratta, in ultima istanza, di una pagina JSP il cui ciclo di vita viene gestito dal *JSP container*. I tag riconosciuti (perché associati ad un definito *namespace*) vengono invece processati producendo omologhi tag HTML.

> Questo processo viene detto **encoding**. Il **renderer** associato all'oggetto **`UIInput`** invoca il framework JSF per il *look up* di un identificativo univoco (nel caso in cui venga esplicitamente indicato un *component identifier* dal programmatore tale passaggio non viene eseguito) ed ottenere il valore corrente dell'espressione **`user.name`** (con cui aggiorna il componente **`inputText`** per mantenere la sincronizzazione con il *backing bean* **`user`**):

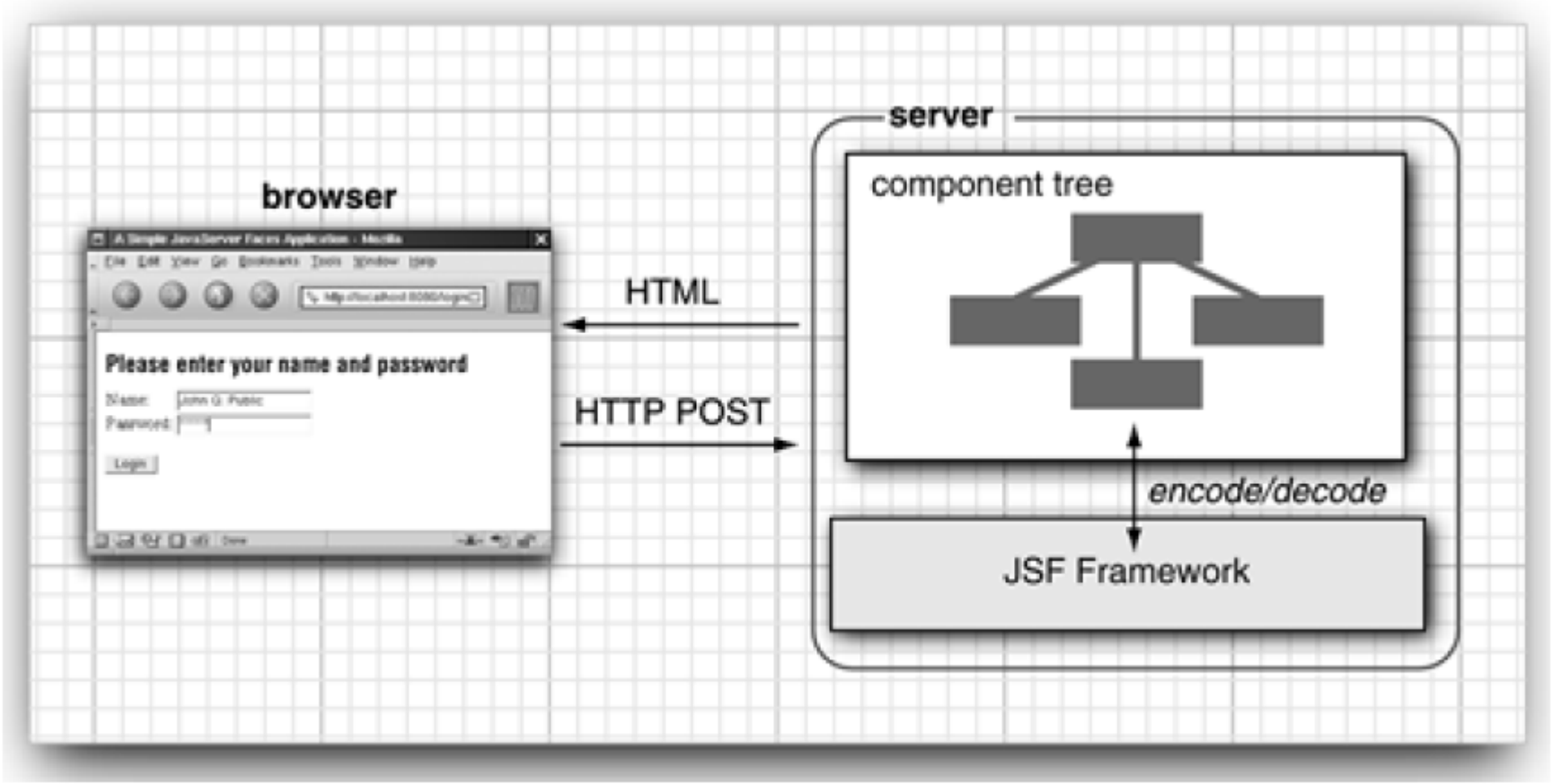
JSP: **`<h:inputText value="#{user.name}"/>`**

Output HTML risultante:

```
<input type="text" name="unique ID" value="current value"/>
```

JSF: dietro le quinte (rendering pages)

> La figura sottostante illustra il processo descritto nella pagina precedente



JSF: dietro le quinte (decoding request)

> La URL associata alla richiesta POST è la stessa della richiesta che ha generato il form *(postback)*. La navigazione verso una nuova pagina viene definita dopo che il form è stato processato

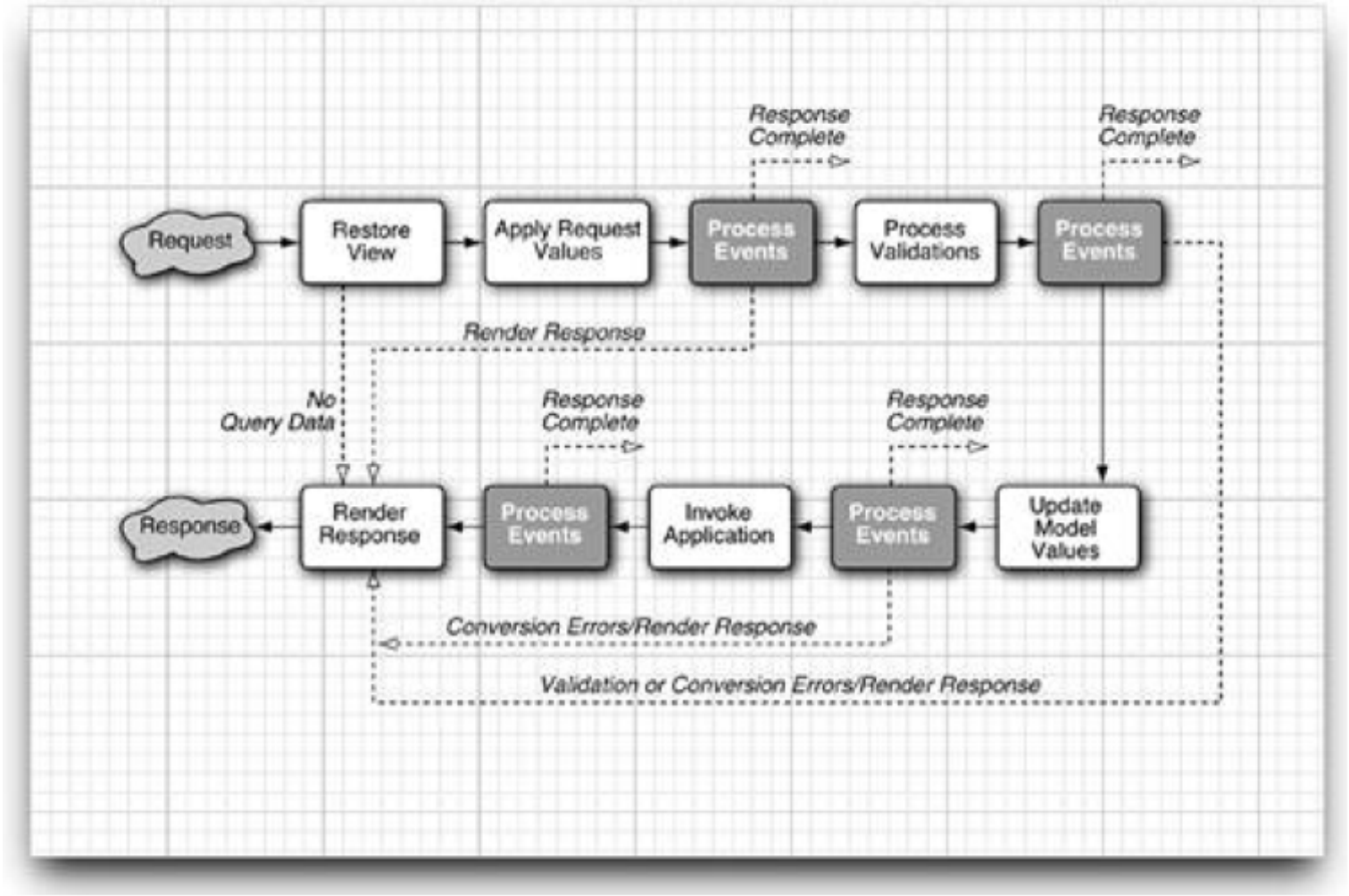
> Il controller processa i dati del form che vengono posti in una hash table a cui i vari componenti possono accedere. Successivamente il framework consente ai vari componenti di recuperare i dati di propria pertinenza.

> Nell'esempio visto, sono stati presentati 3 componenti:

- i componenti UIInput aggiornano le proprietà del bean referenziate nei rispettivi attributi **`value`**: vengono invocati i metodi *setter* utilizzando come parametri di input i dati forniti dall'utente (convertiti ed, eventualmente, validati)
- il componente UICommand controlla se il bottone è stato premuto. Se sì, genera un evento che sarà consumato dall'*action listener* associato al componente di tipo command.
- la stringa di *outcome* **`login`** (staticamente definita) referenziata dall'attributo **`action`** del comando porta il navigation handler a recuperare la successiva pagina welcome.jsp

JSF: dietro le quinte (request processing lifecycle)

> La figura illustra il percorso principale e i flussi alternativi



JSF: dietro le quinte (request processing lifecycle)

> La fase di *Restore View* ricerca il *component tree* della pagina richiesta. Se si tratta della prima richiesta (*initial request*) il *component tree* viene creato. Se la request non presenta dati in POST, l'implementazione JSF porta alla fase *Render Response* (accade quando la pagina viene richiesta la prima volta)

> Nella fase di *Apply Request Values* JSF itera sui componenti affinché ciascuno di essi memorizzi i dati di pertinenza (*submitted values*). Successivamente il dato viene convertito coerentemente col tipo della proprietà del bean indicato nella *value binding expression* associata

> Nella fase di *Process Validation* -se sono definiti validatori su uno o più componenti- viene attivato il processo di validazione che, se concluso senza errori, porta all'aggiornamento dei "*local value*" dei componenti e all'invocazione della fase successiva. Nel caso in cui, al contrario, vengano riscontrati errori di conversione o validazione, l'implementazione JSF invoca direttamente la fase *Render Response* la quale porta al ri-display della pagina per garantire all'utente di correggere i dati di input

JSF: dietro le quinte (request processing lifecycle)

> Se la fase di validazione si conclude senza errori si aggiorna il modello dei dati. In questo modo si è certi di operare con un modello che non presenta stati inconsistenti. Durante la fase *Update Model* i “local values” sono utilizzati per aggiornare le proprietà dei bean associati ai componenti

> Nella fase *Invoke Application* il metodo indicato nell’attributo **`action`** del componente *button* o *link* viene eseguito. Tecnicamente è presente un unico *action listener* di default associato a ciascun componente che delega l’*action method* per la gestione dell’azione. Questo metodo tipicamente implementa la logica di business associata all’azione dell’utente o, in architetture più complesse, delega l’esecuzione degli algoritmi di business ad altri oggetti (business delegate). Il metodo ritorna un *outcome string* che viene passata al navigation handler il quale provvede al look up della pagina successiva.

> Infine, la fase *Render Response* effettua l’*encoding* della risposta e la spedisce al browser. Quando l’utente invia un nuovo submit o effettua il click su un link (genera, in altre parole, una nuova richiesta) il ciclo ricomincia

JSF: Managed Beans

> I Managed bean vengono configurati nel file **faces-config.xml**:

```
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>it.unibo.deis.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

> Il frammento di codice XML riportato consente al framework JSF di costruire un oggetto della classe **it.unibo.deis.UserBean**, attribuirgli il nome **user** e memorizzarlo in sessione

> Dopo che il bean è stato definito esso può essere acceduto dai vari componenti JSF. Lettura e aggiornamento della property **password** del bean **user** si effettua nel modo visto (la notazione **#{…}** viene detta JSF Expression Language: è basata su JSP 2.0 EL)

```
<h:inputSecret value="#{user.password}"/>
```

JSF: EL

> JSF Expression Language, di cui abbiamo visto qualche esempio nelle JSP, consente di referenziare o aggiornare proprietà di bean, o valutare semplici statement, senza scrivere codice Java *full-fledged*.

> JSF EL viene valutato a *runtime (*tipicamente quando avviene il processo di *rendering*) e non in fase di compilazione

> JSF EL è *two-way*, nel senso che consente non solo di leggere properties ma anche aggiornarle

> Le espressioni JSF EL possono essere valutate usando codice Java tradizionale, e quindi non è legato a JSP

> Come marcatore utilizza **#** anziché **$** (utilizzato da JSP 2.0 EL da cui deriva)

> Nelle tabelle successive sono illustrati alcuni esempi di utilizzo di JSF EL e di cosiddetti *implicit object* referenziabili via EL

> L'oggetto di JSF che si occupa di risolvere EL è **`VariableResolver`**

JSF: EL

Example	Description
`#{myBean.value}`	Returns the `value` property of the object stored under the key `myBean`, or the element stored under the key `value` if `myBean` is a `Map`.
`#{myBean['value']}`	Same as `"{#myBean.value}"`.
`#{myArrayList[5]}`	Returns the fifth element of a `List` stored under the key `myArrayList`.
`#{myMap['foo']}`	Returns the object stored under the key `foo` from the `Map` stored under the key `myMap`.
`#{myMap[foo.bar]}`	Returns the object stored under the key that equals the value of the expression `foo.bar` from the `Map` stored under the key `myMap`.
`#{myMap['foo'].value}`	Returns the `value` property of the object stored under the key `foo` from the `Map` stored under the key `myMap`.
`#{myMap['foo'].value[5]}`	Returns the fifth element of the `List` or array stored under the key `foo` from the `Map` stored under the key `myMap`.
`#{myString}`	Returns the `String` object stored under the key `myString`.
`#{myInteger}`	Returns the `Integer` object stored under the key `myInteger`.
`#{user.role == 'normal'}`	Returns `true` if the `role` property of the object stored under the key `user` equals normal. Returns `false` otherwise.
`#{(user.balance - 200) == 0}`	If the value of the `balance` property of the object stored under the key user minus 200 equals zero, returns `true`. Returns `false` otherwise.
`Hello #{user.name}!`	Returns the string "Hello" followed by the `name` property of the object stored under the key `user`. So if the user's name is Sean, this would return "Hello Sean!"
`You are #{(user.balance > 100) ? 'loaded' : 'not loaded'}`	Returns the string "You are loaded" if the `balance` property of the object stored under the key `user` is greater than 100; returns "You are not loaded" otherwise.
`#{myBean.methodName}`	Returns the method called `method` of the object stored under the key `myBean`.
`#{20 + 3}`	Returns 23.

JSF: EL

Implicit variable	Description	Example	Supported in JSP 2.0 EL?
applicationScope	A `Map` of application-scoped variables, keyed by name.	`#{application-Scope.myVariable}`	Yes
cookie	A `Map` of cookie values for the current requested, keyed by cookie name.	`#{cookie.myCookie}`	Yes
facesContext	The `FacesContext` instance for the current request.	`#{facesContext}`	No
header	A `Map` of the HTTP header values for the current request, keyed by header name. If there are multiple values for the given header name, only the first is returned.	`#{header['User-Agent']}`	Yes

JSF: EL

Implicit variable	Description	Example	Supported in JSP 2.0 EL?
headerValues	A `Map` of the HTTP header values for the current request, keyed by header name. For each key, an array of `Strings` is returned (so that all values can be accessed).	`#{headerValues['Accept-Encoding'][3]}`	Yes
initParam	A `Map` of the application initialization parameters, keyed by parameter name. (These are also known as servlet context initialization parameters, and are set in the deployment descriptor).	`#{initParam.adminEmail}`	Yes
param	A `Map` of the request parameters, keyed by header name. If there are multiple values for the given parameter name, only the first is returned.	`#{param.address}`	Yes
paramValues	A `Map` of the request parameters, keyed by header name. For each key, an array of `Strings` is returned (so that all values can be accessed).	`#{param.address[2]}`	Yes
requestScope	A `Map` of request scoped variables, keyed by name.	`#{requestScope.userPreferences}`	Yes
sessionScope	A `Map` of session scoped variables, keyed by name.	`#{sessionScope['user']}`	Yes
view	The current view.	`#{view.locale}`	No

JSF: Message Bundles

> Tipicamente nello sviluppo di una web application si centralizza la gestione dei messaggi utilizzati nelle interfacce utente utilizzando file di properties:

```
currentScore=Your current score is:
guessNext=Guess the next number in the sequence!
```

> Il processo di lettura delle properties presenti in tali file è estremamente semplificato in JSF. Il file di properties viene tipicamente inserito nella directory **`WEB-INF/classes`** della web application. E' mandatorio l'estensione **`.properties.`** Nella pagina JSF viene agganciato il *bundle* con il seguente core tag:

```
<f:loadBundle basename="messages" var="msgs"/>
```

> E' possibile ora usare JSF EL per accedere ai messaggi:

```
<h:otputText value="#{msgs.guessNext}"/>
```

JSF: Message Bundles

> Quando si localizza un bundle file è necessario aggiungere un suffisso (che esprime il *locale)* al nome del file: un *underscore* seguito da 2 lettere in minuscolo che rappresentano il codice della lingua standardizzato da ISO-639 (http://www.loc.gov/standards/iso639-2/):

`messages_it.properties`

`messages_en.properties`

> Il supporto all'internazionalizzazione presente in Java (i18n) consente di caricare il bundle che soddisfa il *locale* corrente (java.util.Locale; java.util.ResourceBundle)

> Quando si prepara la traduzione si tenga in considerazione che i file di bundle non sono codificati in UTF-8! I caratteri UNICODE oltre il 127 devono essere codificati con sequenze di escape \uxxxx. La *utilility* **`native2ascii`** presente nel JDK semplifica la creazione di questi file

JSF: Message Bundles

> Una volta preparati i bundle vi sono 3 opzioni per settare il *locale*:

- utilizzando l'attribito **`locale`** per il tag **`f:view`**: **`<f:view locale="es">`**
- si possono settare i vari locale (quello di default e quelli supportati) in **`faces-config.xml`**:

```
<faces-config>
    <application>
        <locale-config>
            <default-locale>en</default-locale>
            <supported-locale>de</supported-locale>
        </locale-config>
    </application>
</faces-config>
```

quando il browser si connette ad una applicazione tipicamente include **`Accept-Language`** come header HTTP che indica la lingue accettate

- Applicativamente:

```
UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
viewRoot.setLocale(new Locale("de"));
```

JSF: Navigazione

> Posssiamo avere una navigazione *statica* oppure *dinamica*

> Nella Navigazione statica il *navigation handler* porta ad una pagina JSF predefinita a fronte di un'azione da parte dell'utente (click di un componente *button* o *link*)

> Nella maggior parte delle web application la navigazione è dinamica:

```
<h:commandButton label="Login" action="#{loginController.verifyUser}"/>
```

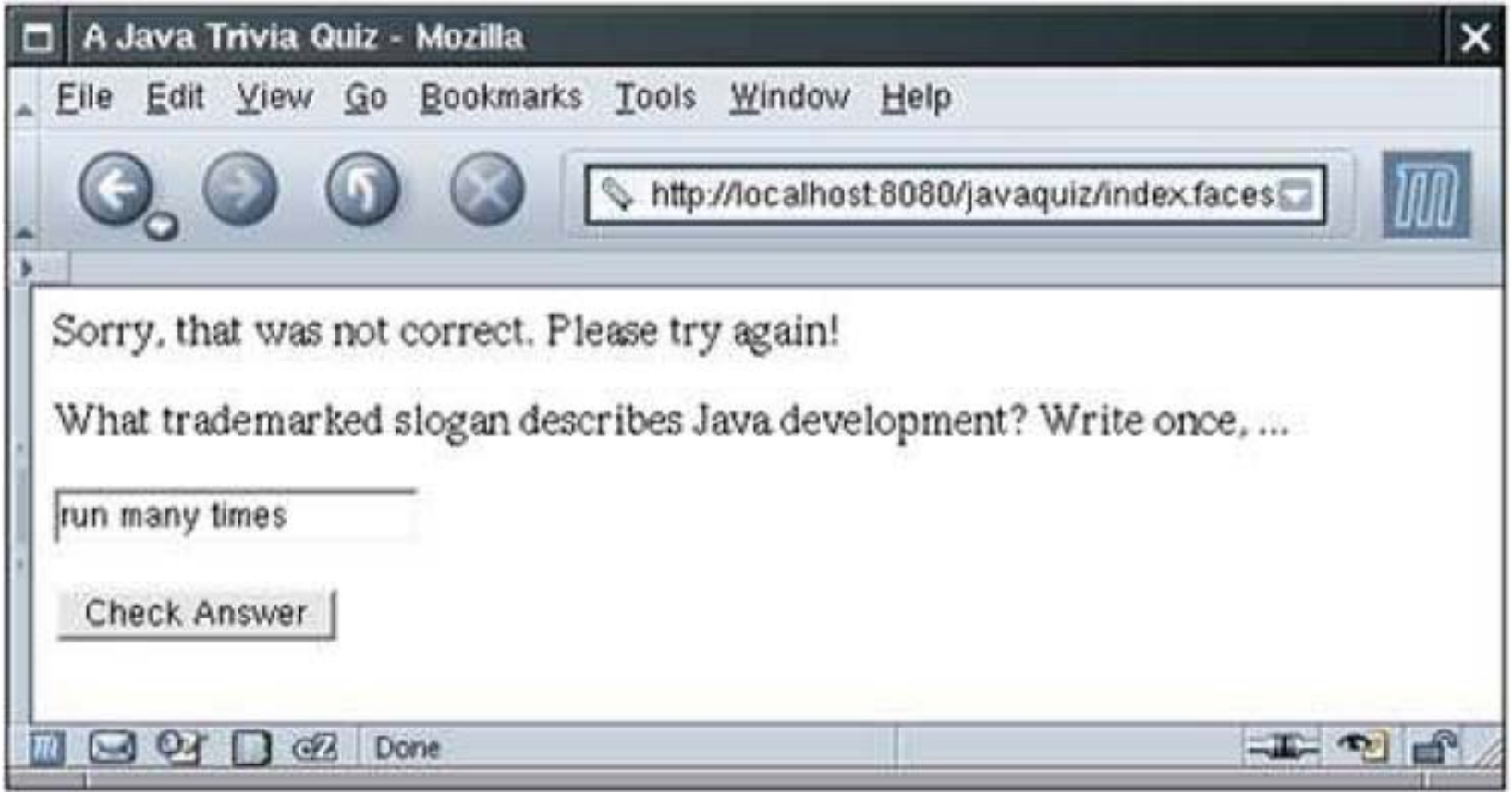
> Nell'esempio riportato **`loginController`** referenzia un bean di una qualche classe che implementa il metodo **`verifyUser`**

```
String verifyUser() {
      if (...)
            return "success";
      else return "failure";
}
```

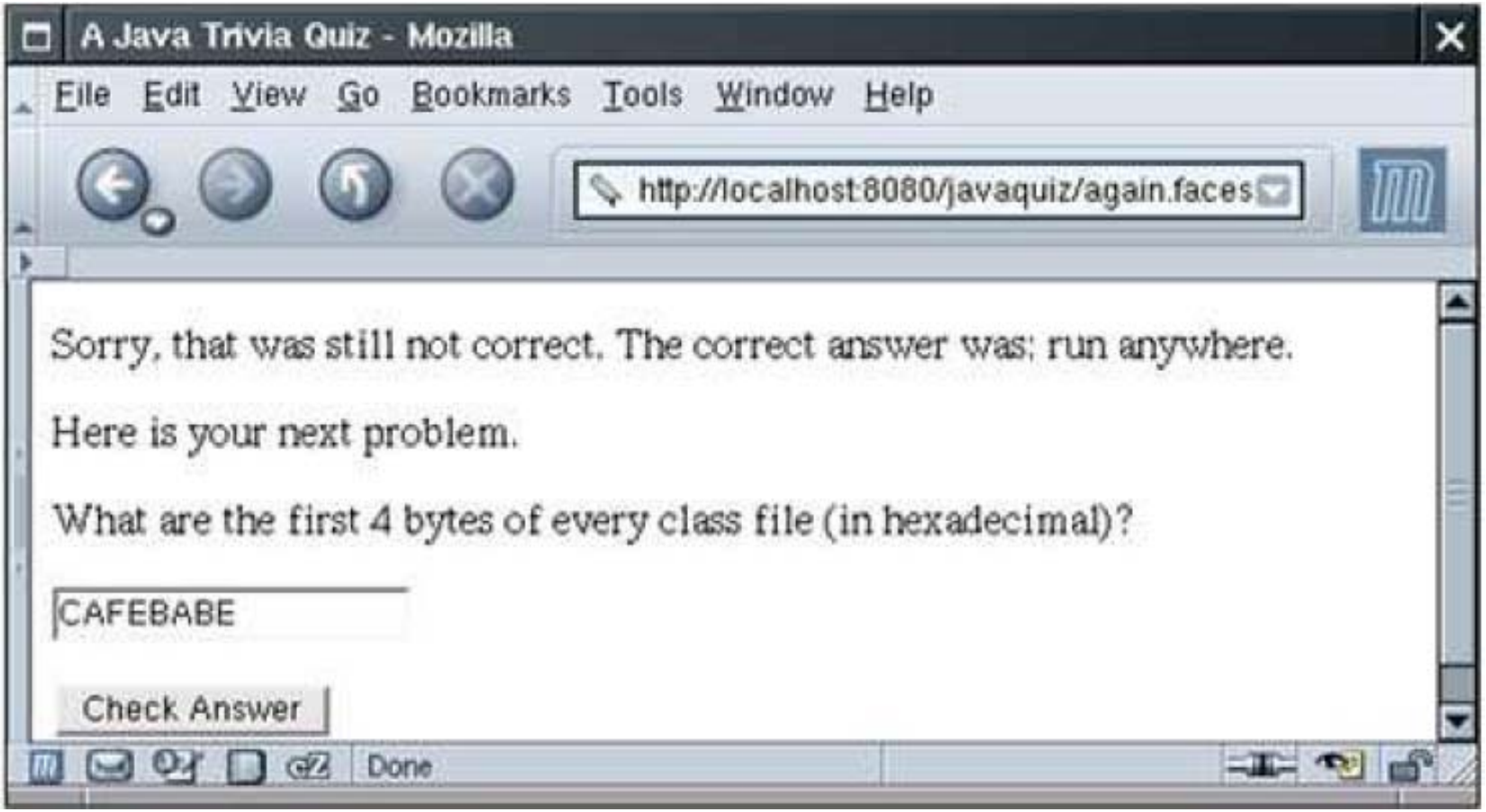
JSF: esempio di navigazione

> L'applicazione presenta una sequenza di quiz. Vi sono al massimo 2 possibilità per ogni domanda. Dopo la seconda risposta viene posta la domanda successiva. Ad ogni risposta corretta viene aggiornato lo score

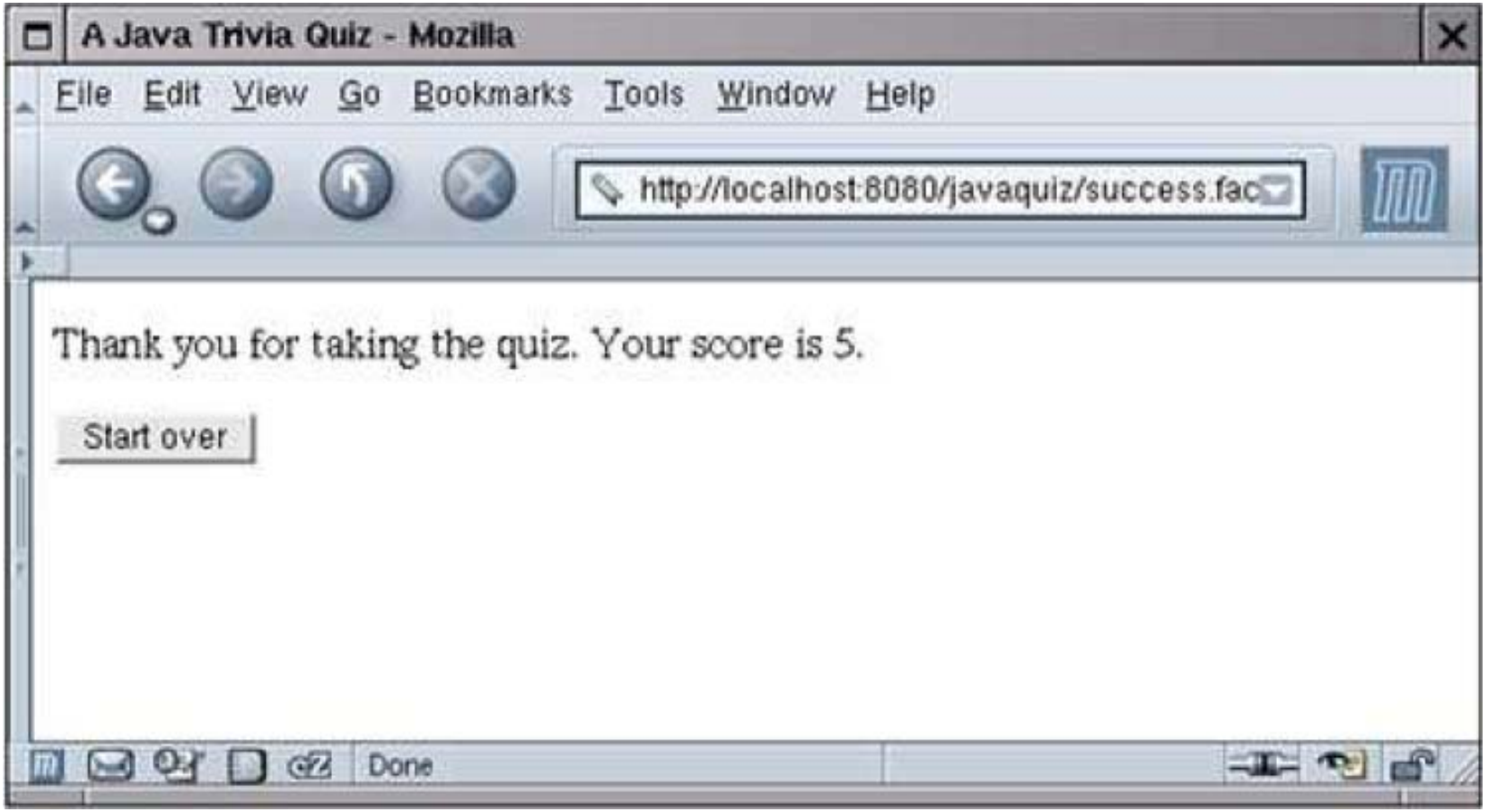
JSF: esempio di navigazione



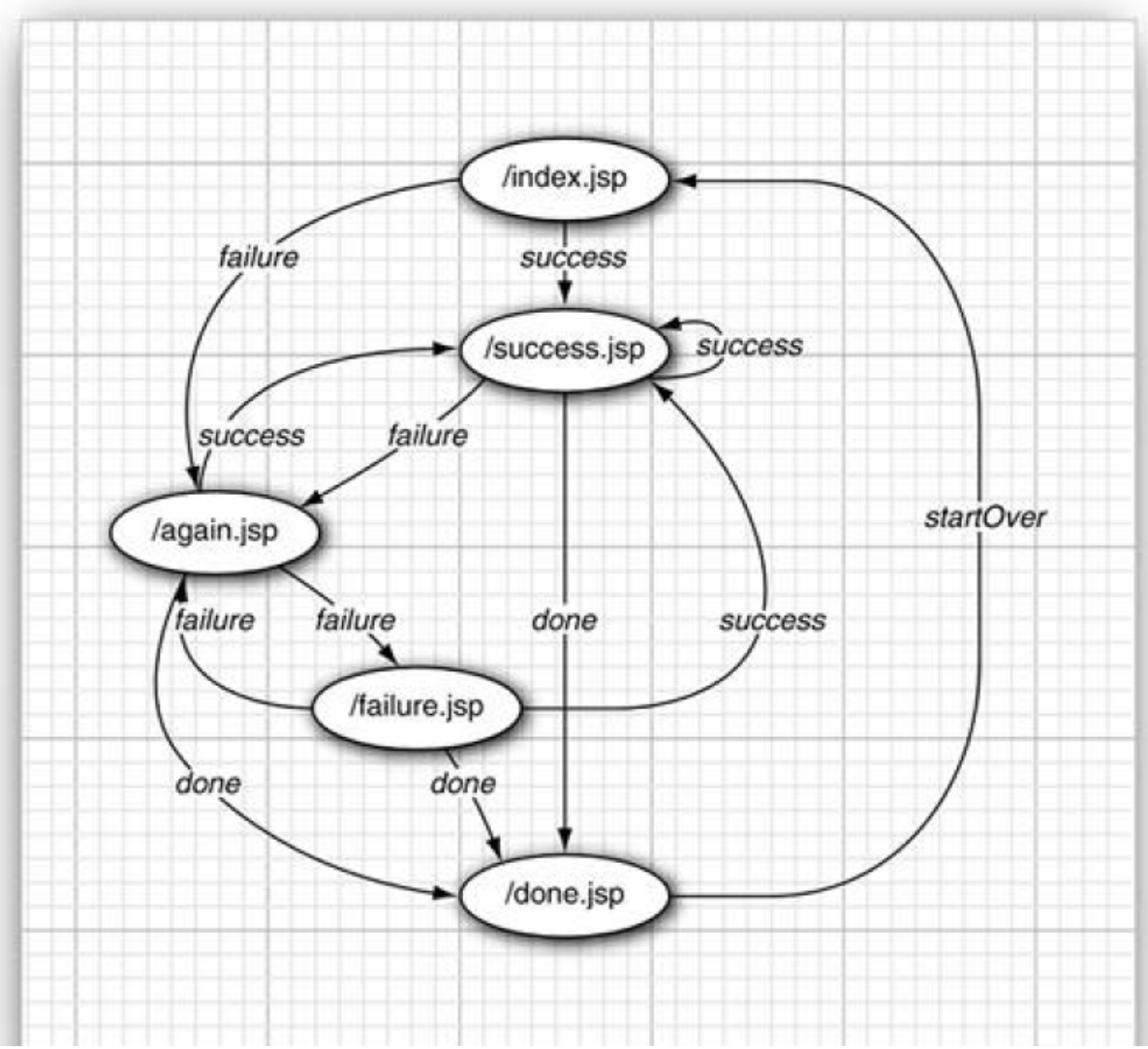
JSF: esempio di navigazione



JSF: esempio di navigazione



JSF: esempio di navigazione



JSF: esempio di navigazione

```
 1. <?xml version="1.0"?>
 2.
 3. <!DOCTYPE faces-config PUBLIC
 4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
 5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
 6.
 7. <faces-config>
 8.    <navigation-rule>
 9.       <navigation-case>
10.          <from-outcome>success</from-outcome>
11.          <to-view-id>/success.jsp</to-view-id>
12.          <redirect/>
13.       </navigation-case>
14.       <navigation-case>
15.          <from-outcome>again</from-outcome>
16.          <to-view-id>/again.jsp</to-view-id>
17.       </navigation-case>
18.       <navigation-case>
19.          <from-outcome>failure</from-outcome>
20.          <to-view-id>/failure.jsp</to-view-id>
21.       </navigation-case>
22.       <navigation-case>
23.          <from-outcome>done</from-outcome>
24.          <to-view-id>/done.jsp</to-view-id>
25.       </navigation-case>
26.       <navigation-case>
27.          <from-outcome>startOver</from-outcome>
28.          <to-view-id>/index.jsp</to-view-id>
29.       </navigation-case>
30.    </navigation-rule>
31.
32.    <managed-bean>
33.       <managed-bean-name>quiz</managed-bean-name>
34.       <managed-bean-class>it.unibo.deis.QuizBean</managed-bean-class>
35.       <managed-bean-scope>session</managed-bean-scope>
36.    </managed-bean>
37.
38. </faces-config>
```

JSF: esempio di navigazione (index.jsp)

```
 1. <html>
 2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
 4.
 5.    <f:view>
 6.       <f:loadBundle basename="messages" var="msgs"/>
 7.       <head>
 8.          <title><h:outputText value="#{msgs.title}"/></title>
 9.       </head>
10.       <body>
11.          <h:form>
12.             <p>
13.                <h:outputText value="#{quiz.question}"/>
14.             </p>
15.             <p>
16.                <h:inputText value="#{quiz.response}"/>
17.             </p>
18.             <p>
19.                <h:commandButton value="#{msgs.answerButton}"
20.                   action="#{quiz.answerAction}"/>
21.             </p>
22.          </h:form>
23.       </body>
24.    </f:view>
25. </html>
```

JSF: esempio di navigazione (done.jsp)

```
 1. <html>
 2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
 3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
 4.    <f:view>
 5.       <f:loadBundle basename="messages" var="msgs"/>
 6.       <head>
 7.          <title><h:outputText value="#{msgs.title}"/></title>
 8.       </head>
 9.       <body>
10.          <h:form>
11.             <p>
12.                <h:outputText value="#{msgs.thankYou}"/>
13.                <h:outputText value="#{msgs.score}"/>
14.                <h:outputText value="#{quiz.score}"/>.
15.             </p>
16.             <p>
17.                <h:commandButton value="#{msgs.startOverButton}"
18.                   action="#{quiz.startOverAction}"/>
19.             </p>
20.          </h:form>
21.       </body>
22.    </f:view>
23. </html>
```

JSF: esempio di navigazione (Problem.java)

```
 1. package it.unibo.deis;
 2.
 3. public class Problem {
 4.    private String question;
 5.    private String answer;
 6.
 7.   public Problem(String question, String answer) {
 8.      this.question = question;
 9.      this.answer = answer;
10.    }
11.
12   public String getQuestion() { return question; }
13.
14   public String getAnswer() { return answer; }
15.
16   // da sovrascrivere per implementazioni di controllo più complesse
17   public boolean isCorrect(String response) {
18      return response.trim().equalsIgnoreCase(answer);
19.   }
20. }
```

JSF: esempio di navigazione (QuizBean.java)

```
 1. package it.unibo.deis;
 2.
 3. public class QuizBean {
 4.    private int currentProblem;
 5.    private int tries;
 6.    private int score;
 7.    private String response;
 8.    private String correctAnswer;
 9.
10.    // In un'applicazione reale
11.    // i dati vengono caricati dal database
12.    private Problem[] problems = {
13.       new Problem(
14.          "What trademarked slogan describes Java development? Write once, ...",
15.          "run anywhere"),
16.       new Problem(
17.          "What are the first 4 bytes of every class file (in hexadecimal)?",
18.          "CAFEBABE"),
19.       new Problem(
20.          "What does this statement print? System.out.println(1+\"2\");",
21.          "12"),
22.       new Problem(
23.          "Which Java keyword is used to define a subclass?",
24.          "extends"),
25.       new Problem(
26.          "What was the original name of the Java programming language?",
27.          "Oak"),
28.       new Problem(
29.          "Which java.util class describes a point in time?",
30.          "Date")
31.    };
32.
33.   public QuizBean() { startOver(); }
34.
```

JSF: esempio di navigazione (QuizBean.java)

```
35.    // PROPERTY: question
36.    public String getQuestion() {
37.       return problems[currentProblem].getQuestion();
38.    }
39.
40.    // PROPERTY: answer
41.    public String getAnswer() { return correctAnswer; }
42.
43.    // PROPERTY: score
44.    public int getScore() { return score; }
45.
46.    // PROPERTY: response
47.    public String getResponse() { return response; }
48.    public void setResponse(String newValue) { response = newValue; }
49.
50.    public String answerAction() {
51.       tries++;
52.       if (problems[currentProblem].isCorrect(response)) {
53.          score++;
54.          nextProblem();
55.          if (currentProblem == problems.length) return "done";
56.          else return "success";
57.       }
58.       else if (tries == 1) {
59.          return "again";
60.       }
61.       else {
62.          nextProblem();
63.          if (currentProblem == problems.length) return "done";
64.          else return "failure";
65.       }
66.    }
67.
```

JSF: esempio di navigazione (QuizBean.java)

```
68.    public String startOverAction() {
69.       startOver();
70.       return "startOver";
71.    }
72.
73.    private void startOver() {
74.       currentProblem = 0;
75.       score = 0;
76.       tries = 0;
77.       response = "";
78.    }
79.
80.    private void nextProblem() {
81.       correctAnswer = problems[currentProblem].getAnswer();
82.       currentProblem++;
83.       tries = 0;
84.       response = "";
85.    }
86. }
```

JSF: standard Core Tags

Tag	Description
`view`	Creates the top-level view
`subview`	Creates a subview of a view
`facet`	Adds a facet to a component
`attribute`	Adds an attribute (key/value) to a component
`param`	Adds a parameter to a component
`actionListener`	Adds an action listener to a component
`valueChangeListener`	Adds a valuechange listener to a component
`converter`	Adds an arbitrary converter to a component
`convertDateTime`	Adds a datetime converter to a component
`convertNumber`	Adds a number converter to a component
`validator`	Adds a validator to a component
`validateDoubleRange`	Validates a `double` range for a component's value
`validateLength`	Validates the length of a component's value
`validateLongRange`	Validates a `long` range for a component's value
`loadBundle`	Loads a resource bundle, stores properties as a Map
`selectitems`	Specifies items for a select one or select many component
`selectitem`	Specifies an item for a select one or select many component
`verbatim`	Adds markup to a JSF page

JSF: standard HTML Tags

Tag	Description
`form`	HTML form
`inputText`	Single-line text input control
`inputTextarea`	Multiline text input control
`inputSecret`	Password input control
`inputHidden`	Hidden field
`outputLabel`	Label for another component for accessibility
`outputLink`	HTML anchor
`outputFormat`	Like `outputText`, but formats compound messages
`outputText`	Single-line text output
`commandButton`	Button: submit, reset, or pushbutton
`commandLink`	Link that acts like a pushbutton
`message`	Displays the most recent message for a component
`messages`	Displays all messages
`graphicImage`	Displays an image
`selectOneListbox`	Single-select listbox
`selectOneMenu`	Single-select menu

JSF: standard HTML Tags

Tag	Description
`selectOneRadio`	Set of radio buttons
`selectBooleanCheckbox`	Checkbox
`selectManyCheckbox`	Set of checkboxes
`selectManyListbox`	Multiselect listbox
`selectManyMenu`	Multiselect menu
`panelGrid`	HTML table
`panelGroup`	Two or more components that are laid out as one
`dataTable`	A feature-rich table control
`column`	Column in a `dataTable`

> Si tenga presente che i tag JSF rappresentano un componente e delegano la generazione del markup (HTML in questo caso) al renderer. Questo fatto garantisce circa la possibilità di adattare JSF a tecnologie di display dei dati alternative (ciò accade, per esempio, per le wireless JSF application)

JSF: attributi comuni ai tag

> Gli attributi condivisi tra vari HTML tag possono essere suddivisi in 3 tipi:

- basic
- HTML 4.0
- DHTML events

JSF: attributi Basic

Attribute	Component Types	Description
`id`	A (25)	Identifier for a component
`binding`	A (25)	Reference to the component that can be used in a backing bean
`rendered`	A (25)	A boolean; `false` suppresses rendering
`styleClass`	A (23)	Cascading stylesheet (CSS) class name
`value`	I, O, C (19)	A component's value, typically a value binding
`valueChangeListener`	I (11)	A method binding to a method that responds to value changes
`converter`	I,O (15)	Converter class name
`validator`	I (11)	Class name of a validator that's created and attached to a component
`required`	I (11)	A boolean; if `TRue`, requires a value to be entered in the associated field

[a] A = all, I = input, O = output, C = commands, (n) = number of tags with attribute

JSF: attributi Basic

> L'attributo **`id`** consente di:

- Accedere a componenti JSF da altri tag

```
<h:inputText id="name".../> <h:message for="name"/>
```

- Ottenere riferimenti a componenti da codice Java

```
UIComponent component =
event.getComponent().findComponent("name");
```

- Accedere a componenti HTML via script

> L'attributo **`binding`** consente di associare un componente alla proprietà di una classe (backing bean):

```
<h:outputText binding="#{form.statePrompt}".../>
```

Ciò permette di manipolare il componente "agganciato" direttamente da codice Java, potendone ottenere il riferimento con una semplice invocazione al metodo getter della proprietà su cui è stato fatto il *binding*

JSF: attributi HTML

> I tag JSF hanno appropriati attributi HTML 4.0 definiti *pass-through*. Questi attributi vengono passati direttamente ai corrispondenti elementi HTML generati. Ogni attributo previsto per i diversi elementi HTML ha un omologo server side associato al tag JSF. Per esempio:

```
<h:inputText value="#{form.name.last}" size="25".../>
```

genera l'elemento HTML seguente:

```
<input type="text" size="25".../>
```

JSF: attributi DHTML

Attribute	Description
`onblur (14)`	Element loses focus
`onchange (11)`	Element's value changes
`onclick (17)`	Mouse button is clicked over the element
`ondblclick (18)`	Mouse button is double-clicked over the element
`onfocus (14)`	Element receives focus
`onkeydown (18)`	Key is pressed
`onkeypress (18)`	Key is pressed and subsequently released
`onkeyup (18)`	Key is released
`onmousedown (18)`	Mouse button is pressed over the element
`onmousemove (18)`	Mouse moves over the element
`onmouseout (18)`	Mouse leaves the element's area
`onmouseover (18)`	Mouse moves onto an element
`onmouseup (18)`	Mouse button is released
`onreset (1)`	Form is reset
`onselect (11)`	Text is selected in an input field
`onsubmit (1)`	Form is submitted

JSF: form

> Il tag JSF form non prevede un attributo action, a differenza dell'omologo tag HTML form. Se si utilizza il tag **`h:form`** senza alcun attributo in una pagina denominata –ad esempio- **`/index.jsp,`** il renderer associato al componente Form genera il seguente HTML:

```
<form id="_id0" method="post" action="/forms/faces/index.jsp"
enctype="application/x-www-form-urlencoded">
```

Attributes	Description
`binding, id, rendered, styleClass`	Basic attributes[a]
`accept, acceptcharset, dir, enctype, lang, style, target, title`	HTML 4.0[b] attributes
`onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onreset, onsubmit`	DHTML events[c]

JSF: form e javascript

Form JSF

```
<h:form id="registerForm">
    <h:inputText id="password".../>
    <h:inputText id="passwordConfirm".../>
    ...
    <h:commandButton type="button“ onclick ="checkPassword(this.form)"/>

</h:form>
```

JS

```
function checkPassword(form) {
    var password = form["registerForm:password"].value;
    var passwordConfirm =    form["registerForm:passwordConfirm"].value;

    if (password == passwordConfirm)
        form.submit();
    else
        alert("Password and password confirm    fields don't    match");
}
```

HTML generato

```
<form id="registerForm" method="post"
     action="/javascript/faces/index.jsp"
    enctype="application/x-www-form-urlencoded">
    ...
    <input id="registerForm:password"
         type="text" name="registerForm:password"/>
    ...
    <input type="button" name="registerForm:_id5"
        value="Submit Form" onclick="checkPassword(this.form)"/>
    ...
</form>
```

JSF: text field e text area

Attributes	Description
`cols`	For `h:inputTextarea` only—number of columns
`immediate`	Process validation early in the life cycle
`redisplay`	For `h:inputSecret` only—when true, the input field's value is redisplayed when the web page is reloaded
`required`	Require input in the component when the form is submitted
`rows`	For `h:inputTextarea` only—number of rows
`valueChangeListener`	A specified listener that's notified of value changes
`binding, converter, id, rendered, required, styleClass, value, validator`	Basic attributes[a]
`accesskey, alt, dir, disabled, lang, maxlength, readonly, size, style, tabindex, title`	HTML 4.0 pass-through attributes[b]—`alt`, `maxlength`, and `size` do not apply to `h:inputTextarea`
`onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onselect`	DHTML events[c]

JSF: text field e text area

Example	Result
`<h:inputText value="#{form.testString}" readonly="true"/>`	12345678901234567890
`<h:inputSecret value="#{form.passwd}" redisplay="true"/>`	********** (shown after an unsuccessful form submit)
`<h:inputSecret value="#{form.passwd}" redisplay="false"/>`	(shown after an unsuccessful form submit)
`<h:inputText value="inputText" style="color: Yellow; background: Teal;"/>`	inputText
`<h:inputText value="1234567" size="5"/>`	123456
`<h:inputText value="1234567890" maxlength="6" size="10"/>`	123456

JSF: display di testo e immagini

> JSF fornisce 3 modalità per il display di testo e immagini

- **`h:outputText`**
- **`h:outputFormat`**
- **`h:graphicImage`**

> Attributi per **`h:outputText`** e **`h:outputFormat`**

Attributes	Description
`escape`	If set to `true`, escapes <, >, and & characters. Default value is `false`.
`binding, converter, id, rendered, styleClass, value`	Basic attributes[a]
`style, title`	HTML 4.0[b]

JSF: display di testo e immagini

> Attributi per **`h:graphicImage`**

Attributes	Description
`binding, id, rendered, styleClass, value`	Basic attributes[a]
`alt, dir, height, ismap, lang, longdesc, style, title, url, usemap, width`	HTML 4.0[b]
`onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup`	DHTML events[c]

JSF: display di testo e immagini

> Alcuni esempi:

```
<h:outputFormat value="{0} is {1} years old">
    <f:param value="Bill"/>
    <f:param value="38"/>
</h:outputFormat>
```

Example	Result
`<h:outputText value="#{form.testString}"/>`	12345678901234567890
`<h:outputText value="Number #{form.number}"/>`	Number 1000
`<h:outputText value="<input type='text' value='hello'/>"/>`	hello
`<h:outputText escape="true" value="<input type='text' value='hello'/>"/>`	`<input type="text" value="hello">`
`<h:graphicImage value="/tjefferson.jpg"/>`	

JSF: campi hidden

> Il tag JSF che genera un elemento HTML hidden è: **`h:inputHidden`**

Attributes	Description
binding, converter, id, immediate, required, validator, value, valueChangeListener	Basic attributes

JSF: bottoni e link

> JSF fornisce i seguenti tag per la gestione di bottoni e link

- **`h:commandButton`**
- **`h:commandLink`**
- **`h:otputLink`**

> I tag **`h:commandButton`** e **`h:commandLink`** rappresentano componenti JSF di tipo *command* –il framework JSF lancia eventi e invoca azioni quando il bottone o il link è attivato.

> Il tag **`h:outputLink`**