

EJB and JavaServer Faces

Jaroslav Dytrych

Faculty of Information Technology Brno University of Technology
Božetěchova 1/2, 602 00 Brno - Královo Pole
dytrych@fit.vutbr.cz



14 October 2020

EJB (Enterprise Java Beans)

- EJB Introduction
- Session Beans
- Message Driven Beans (based on JMS)
- Transactions
- Deployment
- New features in EJB 3.1



- Enterprise JavaBeans (EJB)
 - Enterprise bean is a server-side component that encapsulates the business logic of an application.
 - EJB execute within an EJB container, which is running in the EJB server.



- EJB server is a part of an application server that hosts EJB containers
 - can be also standalone (Apache TomEE)
- EJBs do not interact directly with the EJB server.
- GlassFish (Oracle), WebSphere (IBM), WebLogic (Oracle), WildFly (Red Hat), WebObjects (Apple), ...
- EJB specification outlines eight services that must be provided by an EJB server:
 - Naming
 - Transaction
 - Security
 - Persistence
 - Concurrency
 - Life cycle
 - Messaging
 - Timer



- EJB Container is a runtime environment for EJB component beans.
- Containers are transparent to the client.
 - There is no client API to manipulate the container.
- Container provides EJB instance life cycle management and EJB instance identification.
- Container manages the connections to the enterprise information systems (EISs).
- Container provides services, such as transaction management, concurrency control, pooling (cache with beans) and security authorization.

- EJB components are server-side, modular, reusable, and containing specific units of functionality.
- They are similar to the Java classes as we create every day, but are subject to special restrictions and must provide specific interfaces for container and client use and access.
- We should consider using EJB components for applications that require scalability, transactional processing, or availability to multiple client types.

- Session Beans
 - models task or workflow
 - Façade for Entity beans
 - structural design pattern – simplified interface to a larger set of interfaces.
 - maintains conversational state with clients (one bean per client)
- Message Driven Beans
 - asynchronous communication with MOM (Message Oriented Middleware) – distributed over heterogeneous platforms
 - allows non-Java EE resources to access Session and Entity Beans via JCA (Java EE Connector Architecture – connects AS with EIS) Resource adapters
 - uses JMS
- (Entity Beans 2.1) JPA 2.0 in Java EE 6
 - implicitly persistent
 - transparent persistence with transaction support
 - typically stored in a relational database (Object/Relational Mapping)

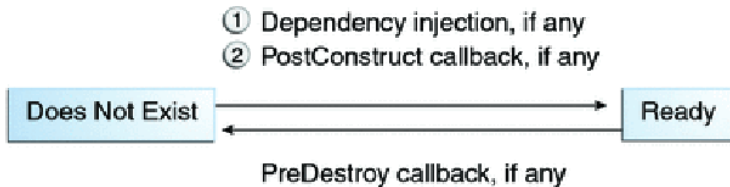




- `@Stateless` annotation
- State only for the duration of a client invocation of a single method.
- Pool of stateless beans is managed by the container.
- Any available stateless session bean may handle the client request.
- Lifecycle event callbacks supported for stateless session beans (optional)
 - `@PostConstruct` – occurs before the first business method invocation on the bean.
 - `@PreDestroy` – occurs at the time the bean instance is destroyed.

```
@Stateless
public class FooBean {
    @PostConstruct
    private void init() {...}

    @Remove
    public void remove() {} // removed from container
}
```



- Instance variables unique to the client/session.
- State is retained for the duration of the client/bean session.
- Client initiate creation and destruction (or timeout).
- Stateful Session Beans can not be pooled.
- Stateful Session Beans can be passivated.
- Supports following callbacks for the lifecycle events:
 - `@PostConstruct` – same as stateless bean, once for each session.
 - `@Init` – designates the initialization method of a bean.
 - `@PreDestroy` – same as stateless, once for each session.
 - `@PrePassivate` – container invokes this method right before it passivates a stateful session bean (clean up held resources, such as database connections or any resources that cannot be transparently passivated using object serialization).
 - `@PostActivate` – container invokes this method right after it reactivates the bean.
 - `@Remove` – called by container before it ends the life of the stateful session bean, first invoking the bean's `PreDestroy` method, if any.
 - `@AroundInvoke` – interceptor method that interposes on business methods (one per class).



```
@Stateful
public class FooBean {
    @PostConstruct
    private void init() {...}

    @Remove
    public void remove() {}
}
```

- 1 Create
- 2 Dependency injection, if any
- 3 PostConstruct callback, if any
- 4 Init method, or `ejbCreate<METHOD>`, if any



- 1 Remove
- 2 PreDestroy callback, if any

Example StatefullBean



- Singleton Session Bean is a POJOs marked with `@Singleton` annotation,
- has only a single instance per JVM,
- supports data sharing and concurrent access,
- allows to set order of initialization (`@DependsOn`),
- initialization can be eager (`@Startup`).

```
@Startup
@Singleton(name = "PrimaryBean")
@DependsOn("SecondaryBean")
public class PrimaryBean {...}
```

```
@Startup
@Singleton(name = "SecondaryBean")
public class SecondaryBean {...}
```

- Business interfaces
 - visible to the client
 - implemented inside the bean



- Local Interface (`@Local`)
 - invoking EJBs within the same JVM,
 - faster, but not so flexible, scalable and adaptable,
 - no network traffic,
 - parameters passed by reference.
- Remote Interface (`@Remote`)
 - invoking EJBs across JVMs,
 - anywhere on the network,
 - parameters passed by value (serialization/de-serialization).

```
@Stateless
public class CalculatorBean
    implements CalculatorRemote, CalculatorLocal {
    public int add(int x, int y) {
        return x + y;
    }
    public int subtract(int x, int y) {
        return x - y;
    }
}

public interface Calculator {
    int add(int x, int y);
    int subtract(int x, int y);
}

@Remote
public interface CalculatorRemote extends Calculator {}

@Local
public interface CalculatorLocal extends Calculator {}
```




- New in EJB 3.1
- EJB may not have business interface.
- Session Beans are simple POJO.
- Automatically exposes public methods.

```
@Stateless  
public class FooBean {  
    public void foo() {...}  
}
```

- It is not necessary to use `@local` or `@remote` (selected automatically).



- `@MessageDriven` annotation
- invoked asynchronously by messages from standard Session Beans,
- cannot be invoked with local or remote interfaces (from clients),
- stateless, transaction aware,
- JMS (Java Messaging Service) used as a transport layer,
- implements method `onMessage (...)`.
- Two types of messaging:
 - point-to-point (queues)
 - pub-sub (topics)



- Clients never use “new” operator on managed beans.
- Session Beans are accessed using DI or JNDI.
- Dependency Injection
 - only from clients in an Java EE environment
 - other EJB, other managed beans, servlet, ...
 - during deployment of the bean to the container
 - `@EJB Foo fooBean;`
- JNDI
 - programming interface to the directory services to locate any object in a network
 - even from non-EE clients
 - access remote business interface

```
Context ctx = new InitialContext();  
FooRemote example = (FooRemote)  
    ctx.lookup("java:global/myApp/FooRemote");
```

```
public class Main {
    @EJB
    private static BookEJBRemote bookEJB;
    public static void main(String[] args) {
        // Creates an instance of Book
        Book book = new Book();
        book.setTitle("The Hitchhiker's Guide to the Galaxy");
        book.setPrice(12.5F);
        book.setDescription("Science fiction by Douglas Adams.");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);
        book = bookEJB.createBook(book); // business layer request
        book.setTitle("H2G2");
        book = bookEJB.updateBook(book); // business layer request
        List<Book> books = bookEJB.findBooks(); // business layer
                                                // request
        System.out.println("List of books in DB:");
        for (Book b : books) {
            System.out.println(b);
        }
        bookEJB.deleteBook(book); // business layer request
    }
}
```

- `java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]`
- `java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]`
- `java:module/<bean-name>[!<fully-qualified-interface-name>]`
- Examples:
 - `java:global/fooEar/fooweb/FooBean`
 - `java:global/fooEar/fooweb/FooBean!com.acme.Foo`
 - `java:app/fooweb/FooBean`
 - `java:app/fooweb/FooBean!com.acme.Foo`
 - `java:module/FooBean`
 - `java:module/FooBean!com.acme.Foo`



- Annotation @Schedule with attributes:
 - year, month, dayOfMonth, dayOfWeek
 - hour, minute, second
 - timezone

```
@Singleton
public class ServiceBean {
    @Schedule(dayOfWeek = "Sun", hour = "2", minute = "30")
    public void cleanDatabase() {...}
}
```



- Calls are synchronous by default.
- Methods can be invoked also asynchronously.
- Return value is a `Future<V>` object of the `java.util.concurrent`

```
@Stateless
public class MathSessionBean {
    @Asynchronous
    public Future<Integer> compute(Integer x, Integer y) {
        Integer z = ...
        return new AsyncResult(z);
    }
}
```

```
Future<Integer> r = mathBean.compute(20, 11);
while (!r.isDone()) { ... }
Integer i = r.get();
```



- **Container-managed (default)**

- Container maintains persistence transparently using JDBC calls.

```
@TransactionAttribute(  
    TransactionAttributeType.REQUIRED)  
public void foo() {...}
```

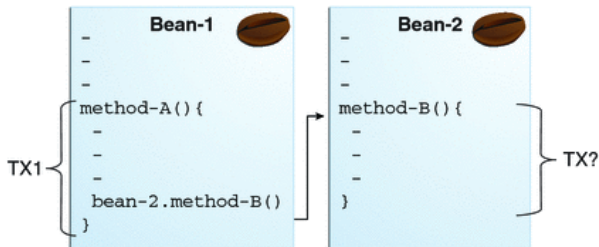
- **REQUIRED** – if the transaction is running, it will be used, else it will be created.
 - **REQUIRES_NEW** – new transaction required.
 - **MANDATORY** – transaction must be already running.
 - **NOT_SUPPORTED** – out of transaction.
 - **SUPPORTS** – can be inserted into transaction, but will not be created.
 - **NEVER** – can not be inserted into transaction.
- **Rollback**
 - Exception is thrown.
 - Transaction is marked for rollback and can never commit.

```
@Resource SessionContext scx;  
...  
scx.setRollbackOnly();
```

- Test if the transaction has been marked for rollback only:
getRollbackOnly()

- **Bean-managed**

- Programmer provides persistence logic.
- Used to connect to non-JDBC data sources like LDAP, mainframe etc.



- Bean-managed

```
@TransactionManagement(TransactionManagementType.BEAN)
class MyBean { ...

    @Resource
    UserTransaction userTransaction;

    ...
    userTransaction.begin();
    ...
    ...
    userTransaction.commit();
    userTransaction.rollback();
    ...
}
```



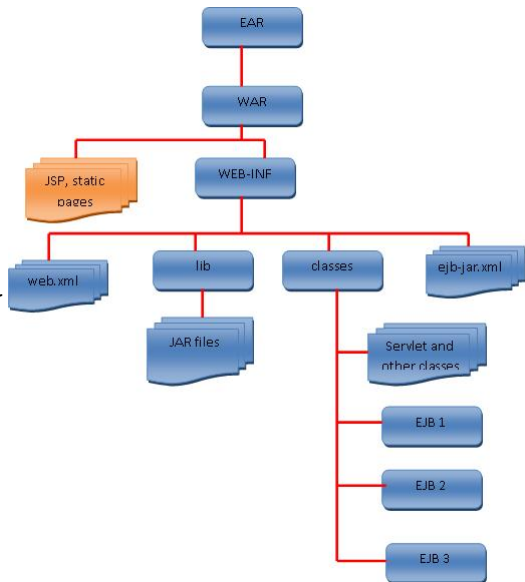
- EJB Lite is a subset of the full EJB.
- EJB Lite can be a part of .war file.
- A Java EE Web Profile certified container has to support the EJB Lite.
- A subset of EJB Full
 - no Message Driven Beans
 - no remote interfaces
 - no EJB timers and scheduling
 - no asynchronous invocation
 - no web services



- Client and EJB runs on the same JVM.
- Better support for testing.
- Batch processing.
- Usage of EJB programming model in desktop applications.

```
@Test
public void hello() throws Exception {
    EJBContainer ec = EJBContainer.createEJBContainer();
    Context c = ec.getContext();
    HelloSessionBean hello = (HelloSessionBean)
        c.lookup( "java:global/classes/HelloSessionBean" );
    String s = hello.sayHello( "Eva" );
    assertEquals( "Hello, Eva", s );
}
```

- Enterprise bean class
- Business interfaces
- Other classes
- Deployment descriptor (optional)



- Deployment Descriptors are included in the JARs, along with component-related resources.
- Deployment Descriptors are XML documents that describe configuration and other deployment settings.
- The statements in the deployment descriptor are declarative instructions to the Java EE container (transactional settings, ...).
- The deployment descriptor for an EJB component must be named `ejb-jar.xml`, and it resides in the META-INF directory inside the EJB JAR files.
- It is also possible to configure it using annotations.

- JSR-299 (CDI)
<http://docs.jboss.org/weld/reference/latest/en-US/html/>
- Java EE 6 Tutorial
<http://java.sun.com/javaee/6/docs/tutorial/doc>
- Oracle® Containers for J2EE Enterprise JavaBeans Developer's Guide
https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/toc.htm
- Programming WebLogic Enterprise JavaBeans
https://docs.oracle.com/cd/E13222_01/wls/docs81/ejb/session.html
- Simplify enterprise Java development with EJB 3.0, Part 1
<https://www.javaworld.com/article/2072037/java-web-development-simplify-enterprise-java-development-with-ejb-3-0-part-1.html>
- Distributed Multitiered Applications – Tiers
 - <https://docs.oracle.com/javaee/6/tutorial/doc/bnaay.html>

- The Open Tutorials - Java EE

<http://theopentutorials.com/content/tutorials/java-ee/>

- Stateful session beans (EJB 3)

<http://index-of.es/Java/EJB-3-IN-ACTION.pdf>

- Others

- <https://access.redhat.com/solutions/158153>

JavaServer Faces

- Introduction
- Creating pages and backing beans (dynamic binding with EL)
- Defining navigation
- JSF Lifecycle
- Data Conversion and Validation
- Events
- Other frameworks (PrimeFaces)
- Conclusions

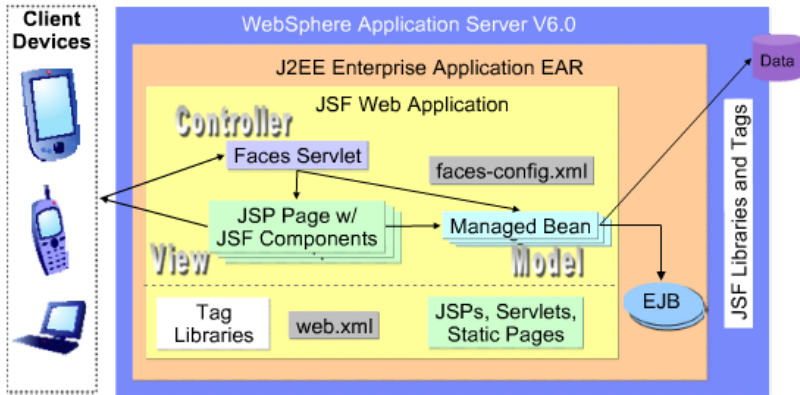


- JavaServer Faces (JSF)
 - application framework for creating Web-based user interfaces
 - component-based
 - provides standard set of components
 - transparently saves and restores component state
 - event handling, server side validation, data conversion
 - framework for implementing custom components
 - based on Model-View-Controller (MVC)
 - running in the standard web container (e.g. Tomcat or Jetty)



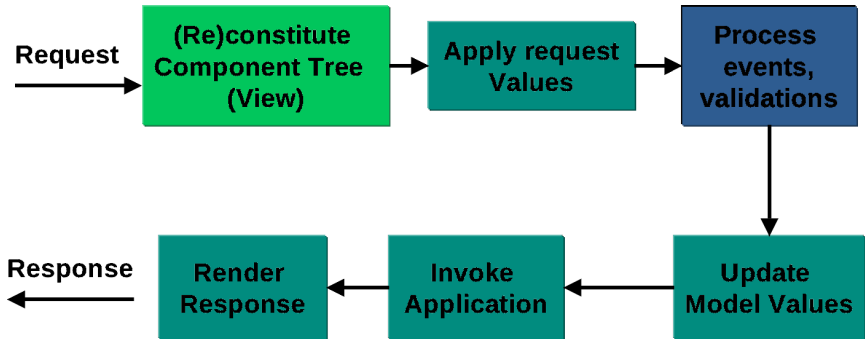
- JSF tags in a Java Server Pages (JSP) files (`.jsp`) or in facelets (`.xhtml`)
 - defines the layout of JSF components
- Backing beans (`.java`)
 - JavaBeans components, defines properties and functions for UI components on a page
- Configuration resource files (`web.xml`, `faces-config.xml`)
 - navigation, configuration of backing beans, custom components, deployment descriptor
- Custom objects (`.java`)
 - Custom UI components, validators, converters, listeners
- Custom tag library definition (`.xml`)

JSF runtime: The big picture





- JSF manages components' state





- ID of the view is extracted from the request (name of the page).
- Component tree for the current view is created.
- Two possibilities
 - initial request
 - build a view of the page (ui components), event handlers, validators, ...
 - postback
 - form was sent
 - restore view from state information (server or client)



- Every component retrieves current state.
- Calls the decode method on component renderer
 - set component values, queue events, messages
 - values from request (headers, cookies, form data)
- If any decode methods / event listeners called `renderResponse` on the current `FacesContext` instance, the JSF moves to the render response phase.
- Values are converted from strings and set into instances of UI component classes.



- Queued events processed after Apply Request Phase.
- Processes all validators registered on the components in the tree
 - validators may `setValid(false)`
 - error message added to `FacesContext`
 - if any validation error, skip to Render Response phase
 - otherwise, continue to the Update Model Values Phase.
- If the local value is invalid, the JSF adds an error message to the `FacesContext` instance, and the life cycle advances to the render response phase and display the same page again with the error message.
 - `FacesContext` contains objects on the page.



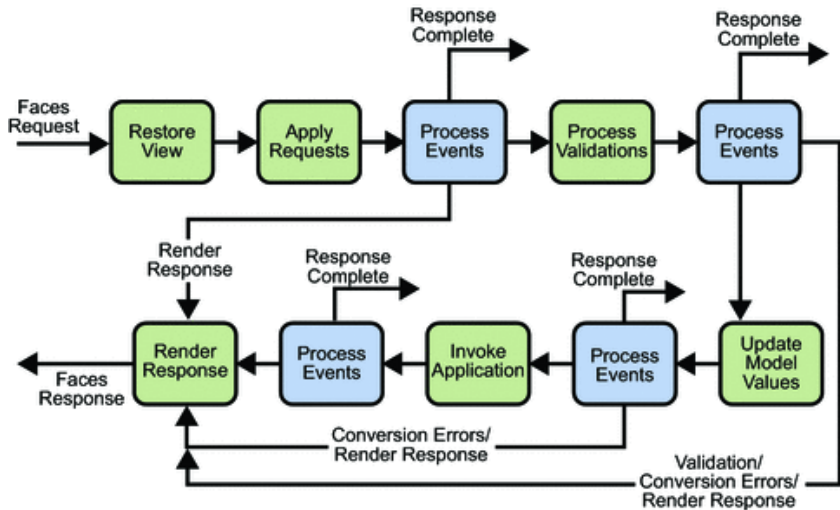
- Updates values of the model (properties of the managed beans).
- Conversion errors may happen.
- If any `updateModels` methods or any listeners called `renderResponse` on the current `FacesContext` instance, the JSF moves to the render response phase.



- Handles any application-level events, such as submitting of a form or linking to another page.
- Application may define a next view
 - action event (form submission).
 - default `ActionListener` handles the “action” result and passes to `NavigationHandler`.
 - Value of `action` in `UICommand` is compared with the rules in the `faces-config.xml`



- JSP or facelet container will render the page
 - JSF tag handlers will setup rendering of the components.
 - User can see the result.
- After the content of the view is rendered, the response state is saved so that subsequent requests can access it and it is available to the restore view phase.
 - Tags `<message/>` and `<messages/>`





JSF UI Components

- basic building blocks of a JSF application
- can represent simple to complex user interface components ranging from a button or input field to a complete page
- can be associated to Model data objects through Value Binding
 - usage of EL (`${...}` and `#{...}`)
- UI Components use helper objects: validators, converters, listeners/events



- JSF components have two parts
 - Component (Java Class)
 - Renderer (jQuery)
- Displaying of component
 - Direct implementation (component encodes/decodes itself)
 - Delegated implementation (encoding/decoding delegated to Renderer – used for extensions)
- `UIComponent`
 - form a tree, instance of `UIViewRoot` class is the root
 - hold state (interface `StateHolder`), registers listeners, ...
- UI Component classes
 - `UIViewRoot`, `UIForm`
 - `UIOutput`, `UIInput`
 - `UICommand`
 - `UISelectMany`, `UISelectOne`, `UISelectItem`
 - `UIData`, `UIColumn`
 - `UIPanel`



- **Renderer**
 - JSF tree to HTML (XUL, SVG, ...)
- **Renderkit**
 - set of renderers with same output format
 - multiple renderers for one component
 - `UICommand` – `commandLink`, `commandButton`
 - `UISelectOne` – `selectOneListbox`, `selectOneMenu`, `selectOneRadio`
 - defined in tag library
- **Component tags**
 - `outputText`, `inputText`, `inputTextarea`, `inputHidden`, `inputSecret`, ...
- **Component tag attributes**
 - `id`: unique id of the component
 - `immediate`: events, validation, conversion should happen in the “apply request phase” – use this attribute to skip validation in case of Cancel button is clicked.
 - `rendered`: do not render if set to false
 - `value`: binds the value to some backing bean property
 - `binding`: binds the instance to some backing bean property (whole component!)
 - `style`
 - `styleClass`

- JSP pages with JSF tag library
- View is a tree of JSF components
- Editable components inside a “form” component

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:view>
  <h:form id="signinForm">
    <h:inputText id="name" value="#SignInBean.name" required="true"/>
    <h:commandButton id="submit" action="#SignInBean.signin"
      value="Submit" />
  </h:form>
</f:view>
```



- Usually one managed bean per page.
- Backing beans defines properties and methods associated with UI components.
- Component value to bean property binding defined in JSF page using Unified Expression Language (EL)

```
class SigninBean {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String signin() {
        ...
        if (ok) { return "success"; } else { return null; }
    }
}
```



- JSF Managed Beans works as Model for UI component,
- can be accessed from JSF page,
- contains getters, setters (like backing bean),
 - Backing bean contains all the HTML form value, managed not.
- registered through annotations
 - `@ManagedBean(name="helloWorld", eager=true)`
 - `eager=true` – created before requested, otherwise lazy

- **Scopes**

`@RequestScoped, @NoneScoped, @ViewScoped,`
`@SessionScoped, @ApplicationScoped,`
`@CustomScoped`

- `@NoneScoped` created on every single EL expression referencing the bean.
- `@ManagedProperty` annotation
 - Bean can be injected to another managed bean.

- From JSF 2.2 it is highly recommended to use CDI (Contexts and Dependency Injection).
- `@ManagedBean` is deprecated
 - `@Named` (`javax.inject.Named`) can be used instead
 - `eager = true` needs extension
<https://www.javacodegeeks.com/2013/02/jsf-eager-cdi-beans.html>
- `javax.faces.bean.SessionScoped` is deprecated
 - `javax.enterprise.context.SessionScoped` can be used instead
 - similar for other scopes (e. g. `javax.enterprise.context.ApplicationScoped`)



- Page navigation is defined in `faces-config.xml` file
- can be defined in managed beans
- implicit navigation
- set of navigation rules
 - `from-view-id` (optional)
 - `navigation-case`
 - `from-outcome` – outcome as defined in JSF page or as returned by action method

```
<h:commandButton action="success" value="Submit" />
<h:commandButton action="#{SignInBean.signIn}"
                  value="Submit" />
```

- It is also possible to return page name directly, but it is not recommended.
- Use `<redirect/>` to switch view and redirect (change URL) instead of `forward` (internal forwarding – same URL).

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

Example JSFPageNavigation



- Value of a component can be bound (not same as binding) to a backing bean property.
- Conversion to some types is automatic
 - e.g. `UISelectBoolean` to `boolean` or `java.lang.Boolean`
- Converter Classes
 - `BigDecimalConverter`, `IntegerConverter`, `DoubleConverter`, ...
 - ```
<h:inputText value="#{bean.foo}"
 converter="javax.faces.convert.IntegerConverter" />
```
- `DateTimeConverter`
  - ```
<h:outputText value="#{bean.someDate}">
  <f:convertDateTime style="full" />
</h:outputText>
```
 - `style` full, short, medium, long
 - `pattern` „MM dd yyyy“
- Can be created also for custom objects.

- Custom converters

- interface Converter

- Object getObject(FacesContext, UIComponent, String)

- String getString(FacesContext, UIComponent, Object)

- f:converter

- registered converter

- <h:inputText>

- <f:converter converterId="MyConverter" />

- </h:inputText>

- binding attribute

- Converter registration in faces config

- <converter>

- <converter-for-class>com.example.MyClass

- </converter-for-class>

- <converter-class>com.example.MyConverter

- </converter-class>

- </converter>



- Validates components data, produces an error message.
- Standard validators
 - `validateDoubleRange`, `validateLength`, `validateLongRange`
 - ```
<h:inputSecret id="password">
 <f:validateLength minimum="8"/>
</h:inputSecret>
<h:message for="password"/>
```
- Custom validation
  - ```
<h:inputText validator="#{bean.validateFoo}"/>
```
 - ```
public void validateFoo(FacesContext context, UIComponent
 toValidate, Object value) {
 ...
 ((UIInput)toValidate).setValid(false);
}
```
  - or implement a `Validator` interface (method `validate()` throws `ValidatorException`) and register a validator in `faces-config` ...
    - ```
<f:validator validatorId="MyValidator"/>
```




- Action events

- clicking on buttons or links
- ```
<h:commandLink action="something">
 <f:actionListener binding="fooBean.bar"/>
</h:actionLink>
```

  - ```
public void bar(ActionEvent event);
```
- ```
<f:setPropertyActionListener
target="#{user.username}" value="some"/>
```

  - sets target property of backing bean (e.g. on click)

- Value change event

- user changes the value of a `UIInput` component
- `f:valueChangeListener` tag
  - `type` – name of the class that implements `ValueChangeListener` interface
  - `binding` – EL expression which evaluates to bean instance of class implementing `ValueChangeListener`

- Data model event (interface `DataModelListener`)

- new row of a `UIData` is selected

- `h:dataTable`
  - `value` - EL expression which evaluates to:
    - list/array of beans, `javax.faces.model.DataModel`, `java.sql.ResultSet`, `javax.sql.RowSet`, ...
  - `var` - name of the variable which iterates through the `DataModel`
  - `first`, `rows`, ...

```
<h:dataTable var="i" value="#{DataTableBean.items}">
 <h:column>
 <f:facet name="header"><h:outputText value="key"/></f:facet>
 <h:commandLink action="details" value="#{i.key}">
 <f:setPropertyActionListener
 target="#{ItemDetailBean.item}"
 value="#{i}"/>
 </h:commandLink>
 </h:column>
 <h:column>
 <f:facet name="header">
 <h:outputText value="value"/>
 </f:facet>
 <h:outputText value="#{i.value}"/>
 </h:column>
</h:dataTable>
```

- **UIComponent**
  - subclass `UIComponentBase`
  - behavioral interfaces `StateHolder`, `ValueHolder`, `NamingContainer`, `EditableValueHolder`, `ActionSource`
  - component family
- **Renderer**
  - `encodeBegin` (before descendants)
  - `encodeEnd` (after descendants)
  - `decode` (state from request)
- **Tag**
  - abstract class `UIComponentELTag` (values from EL API)
  - `UIComponentBodyELTag` (processing of tag body)
  - `getComponentType`, `getRendererType`
  - sets the component properties based on attributes



- Using Ajax technique, JavaScript code exchanges data with server and updates parts of web page without reloading of the whole page (partial rendering).
- JSF Tag

```
<f:ajax execute="input-component-name"
 render="output-component-name" />
```

- Tag attributes

- disabled
- event (what events invokes AJAX – blur, keypress, click, ...)
- execute (processed components – @all)
- immediate (True – action attribute evaluated in apply request values / False – action attribute evaluated in invoke application phase)
- listener – what should be called during AJAX request (on the server)
- oneerror – name of JavaScript function, when error in AJAX occurs
- onevent – JavaScript callback to handle UI event
- render – what should be rendered

- Apache MyFaces
  - Implementation of JSF with additional components
  - UI-Component Sets
    - Trinidad
    - Tobago
- ICEFaces (ICEsoft Technologies) and RichFaces (Red Hat)
  - AJAX components without writing JavaScript code
  - skinnable
  - Menus, Trees, Calendar, File Upload, Drag and Drop, Google Maps, ...
  - Rich text editor
    - TinyMCE in RichFaces
    - CKEditor in ICEFaces
- PrimeFaces (PrimeTek Informatics)
- Others

- JavaServer Faces is a standard EE component based framework.
- JSF by default uses JSP for rendering, provides basic HTML-based components (facelets are replacing JSPs).
- Managed beans as backing beans for pages defines properties and methods.
- View state is stored between requests.
- Rich frameworks on top of JSF.

- JSF
  - <http://www.tutorialspoint.com/jsf/>
- API
  - <http://docs.oracle.com/javaee/6/api/>
- Others
  - <https://www.mkyong.com/jsf2/jsf-page-forward-vs-page-redirect/>

- GlassFish AS have deployment descriptor `glassfish-web.xml` (see documentation)
- For correct reading of the form values with diacritic, it is necessary to set:

```
<parameter-encoding default-charset="UTF-8"/>
```



Thank you for your attention!