

Hands-on! JavaServer Faces using Apache MyFaces

Gerald Müllan (IRIAN.at)

Matthias Weißendorf (Oracle Germany)

Agenda

- Introduction to JSF
 - theory and practice
- Introduction to Apache MyFaces
 - Using Custom Components
 - Portal-Integration
- Some enhanced stuff (if time is there 😊)

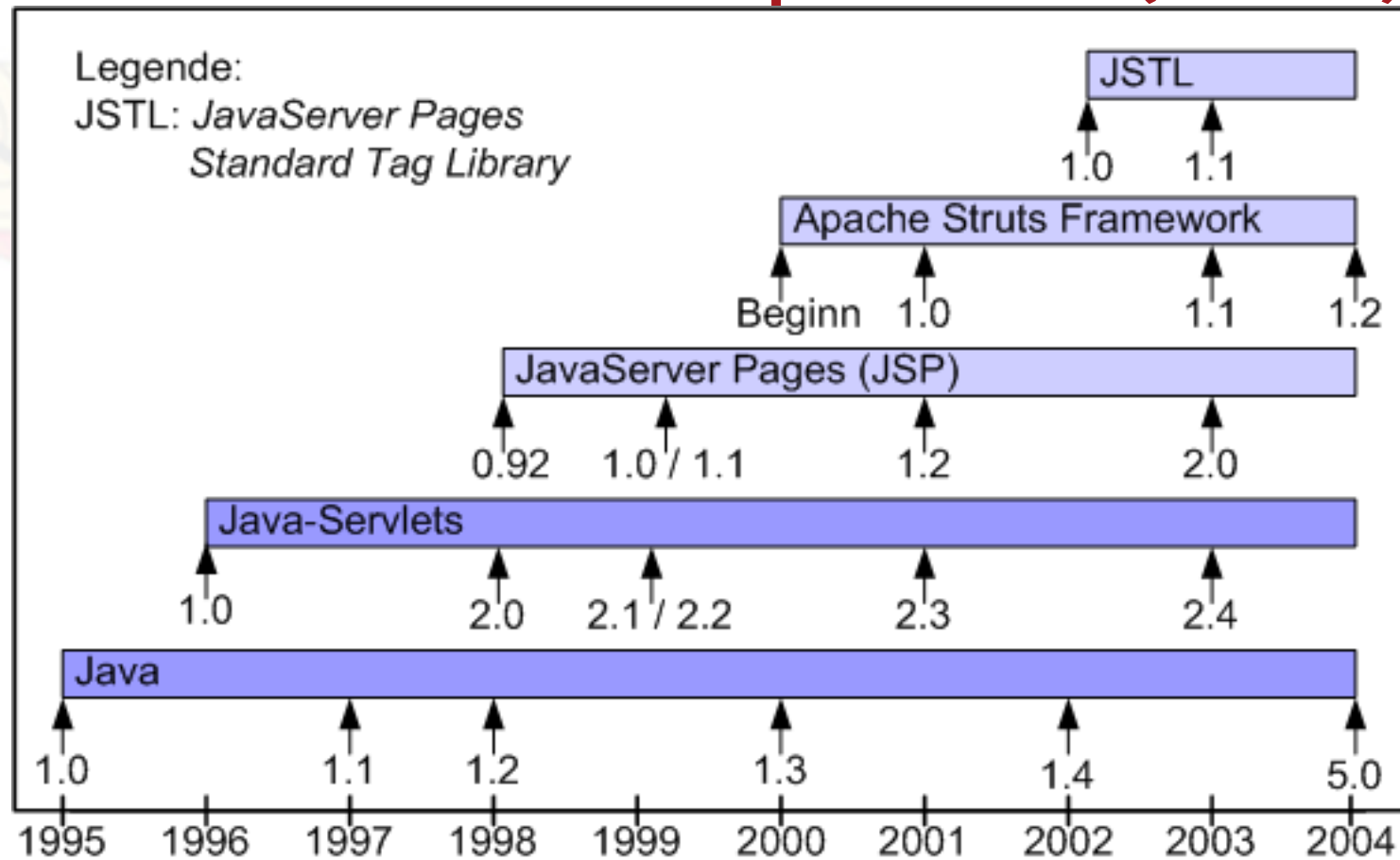
JavaServer Faces

- It's easy to use JSF – however, it's easy to abuse it as well

Web-Development (generally)

- Web apss become more and more important
- More and more complexity
 - Ajax, validation (server vs. client), ...
- Higher customer requirement over the years
 - Rich user experience (easy to use)
 - Ergonomics vs. functionality
- There is always the time ...

Web development (Java)



Servlets

-

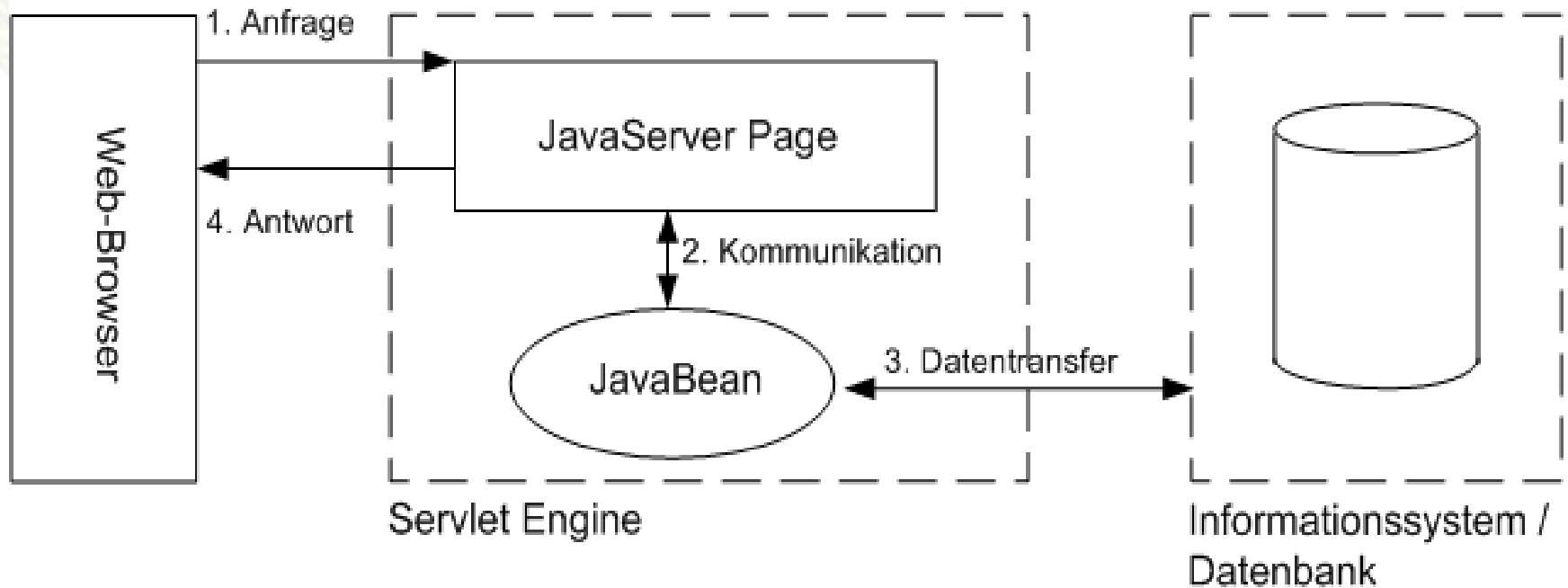
...

```
Collection kunden = datenbank.alleKunden();
PrintWriter writer = response.getWriter();
writer.println("<table border=\"1\">");
Iterator it = kunden.iterator();
while(it.hasNext()) {
writer.println("<tr>"); writer.println("<td>");
writer.println(((Kunde)kunden.next()).getKundenNumm
er());
writer.println("</td>"); writer.println("</tr>");
}
writer.println("</table>");
...
```

JavaServer Pages

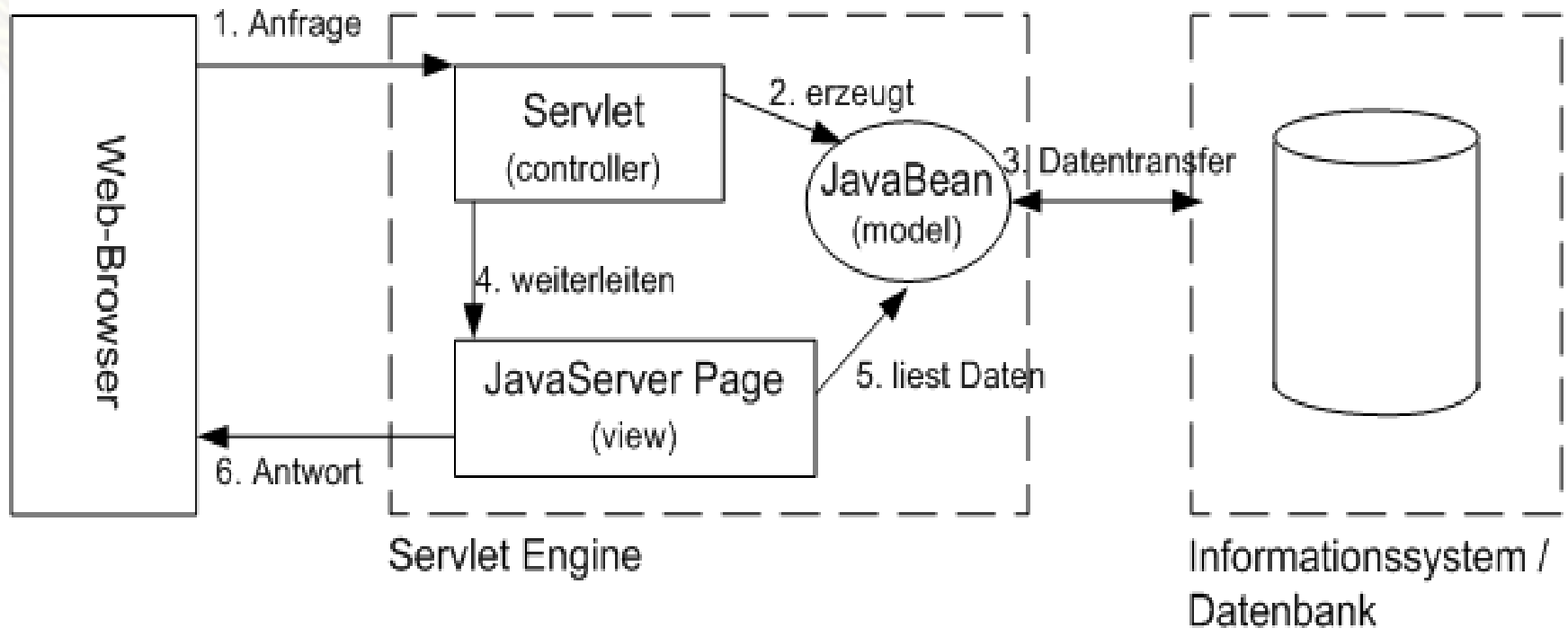
- ```
<%
 Collection kunden = datenBank.alleKunden();
 Iterator it = kunden.iterator();
%>
<table border="1">
<% while(it.hasNext()) { %>
<tr>
<td> <%= ((Kunde) kunden.next()).getKundenNummer() %>
</td>
</tr>
<% } %>
</table>
```

# Model 1





# Model 2



# Java-Web-Frameworks

- Lot's of Model-2 based frameworks out there (too much)
- Apache Struts
- WebWork (soon Struts' Action2 Framework)
- Stripes
- Cocoon
- and many many more ...
- still some „homegrown“

# JSTL (standard)

- ```
<table border="1">
<c:forEach items="$ {kunden}" var="kunde">
  <tr><td>
    <c:out value="$ {kunde.kundenNummer}" />
  </td></tr>
</c:forEach>
</table>
```

JSTL (custom)

- ```
<apacheCon:table items="\${kunden}">
 <apacheCon:column value="\${kundenNummer}" />
</apacheCon:table>
```

# Problems (I)

- Java delivers not enough for webapps.
- It is hard to integrate several frameworks (some times not able)
  - Every framework has its special idea to solve the problem
    - Examples:
      - Struts vs. Cocoon
      - Struts vs. Tapestry
      - Struts vs. Stripes
      - Struts vs. ... (what's your first choice?)

# Problems (II)

- JSTL is not abstract...
  - describes only the rendering
  - Problem: „Tags“ for formular data like a date-picker
    - A particular “tag” is tied to the used framework
    - Not that easy to port it to another, if requirements change...

# What's up ... ?

- Standard is missing!
  - for a web framework
  - for an unified API to build Java Web Components
- SOLUTION:
  - JavaServer Faces ! :-)

# JSF in a nutshell

- JSF is a ...
  - ... Java-Technology for Web-Apps
  - ... component-based framework
  - ... event-driven framework
  - ... RAD
  - ... industrial standard



# Technology for Web-apps

- JSF supports:
  - the Web designer during creating simple templates for his application
  - the Java developer during writing backend code, which is simply independent from the Web server
  - Tool-vendors through its standardized platform

# component driven framework

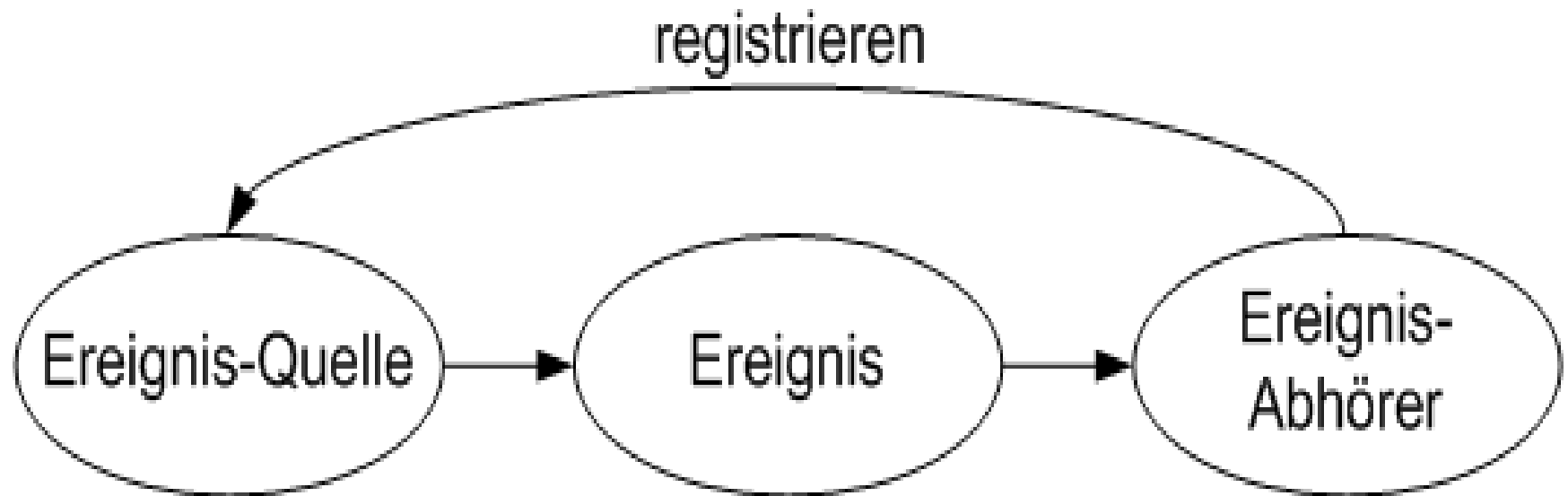
- JSF has builtin components
- define these components inside a JSP file, for instance
- the ,rendering' transforms these components to markup like HTML 4.0.1

`<h:inputText id="x" />`  `<input type="text" id="form:x"/>`

# event driven framework

- Events in JSF:
  - components generate events
  - enforces a method call  
("action" and "event" handler)
  - the state of the web app changes due to that caused event

# event driven framework



# Rapid Application Development

- 4 layers:
    - basal component architecture
    - set of von standard components
    - application infrastructure
    - the RAD tool itself
- ⇒ JSF standardizes the first three point and allows the creation of RAD tools

# Why JavaServer Faces?

- industrial standard (backed by JCP)
  - JSR 127 (JSF 1.0 and JSF 1.1) 2004
  - JSR 252 (JSF 1.2) - 2006 (Java EE 5.0)
    - JSF 1.2 - better interaction with JSP 2.1 and bugfixes
  - JSF 2.0 (architecture, AJAX, more UI components, ..)

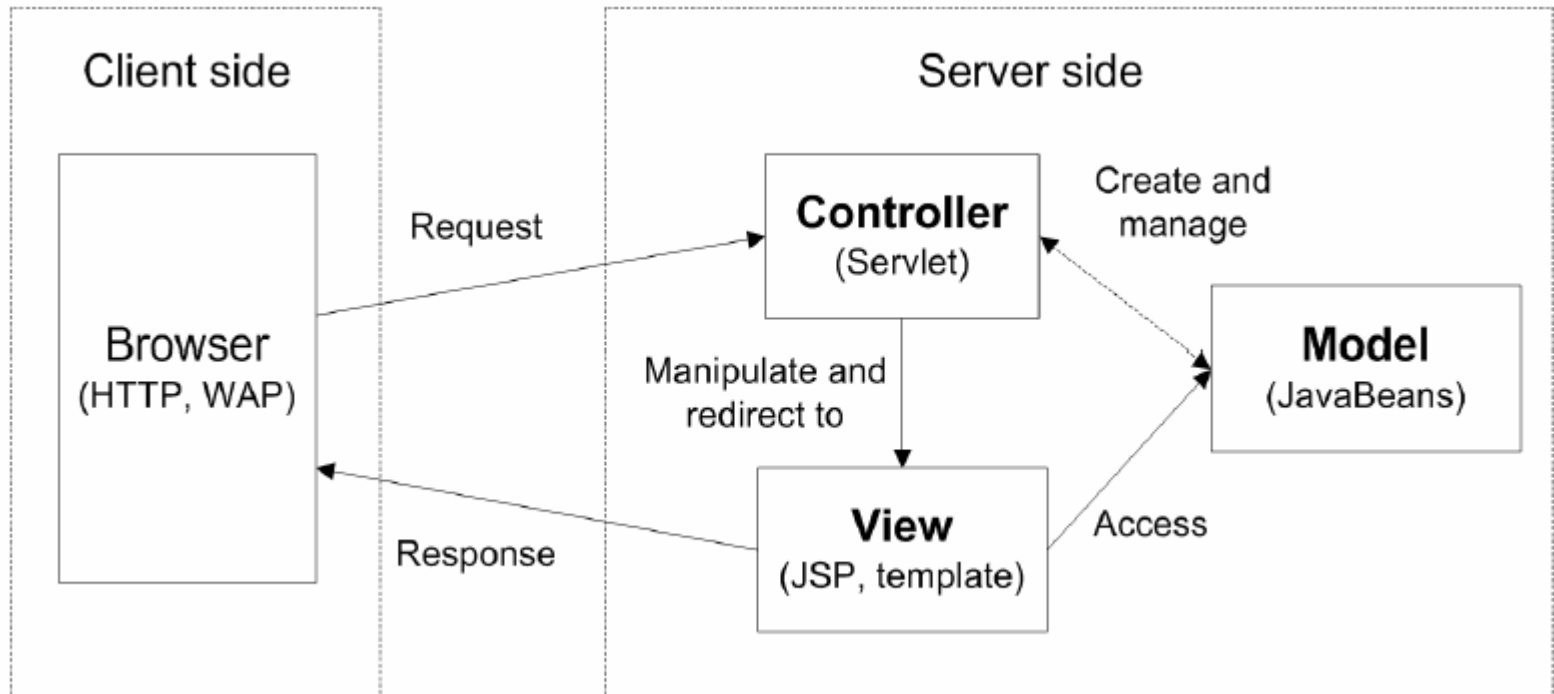
Question is ... when ... 2006, 2007 ?

- Java EE 5.0
- BIG support
  - IDEs (Sun, Eclipse, Oracle, ...)
  - 3rd party UI-components (Oracle, Apache MyFaces)

# Implementation: Apache MyFaces

- First free open source implementation ☺
- Founders:
  - Manfred Geiler (OEKB)
  - Thomas Spiegl (Irian Austria)
- Biggest JSF user community

# JSF - MVC Framework (1)





# JSF - MVC Framework (2)

- **Model:** objects and properties of application (business tier bindings)
- **View:** Renderers take care of the view. That might be HTML (or XUL, or WML)
- **Controller:** FacesServlet / JSF - infrastructure defines the flow of the application

# Reusability

- JSF allows the reuse of ...
    - ... components
      - Reuse of widgets, once created
    - ... views
      - possible to build a layout based on subviews
    - ... your design
      - components support the design
      - Creation of a “Corporate Design”
- ✂ → Reuse for your next project

# Integration (1)

- JSF is flexible; enhance able and can be adopted
  - Fit's into several standards
  - Based upon JSPs and Servlets
  - Frameworks ontop of JSF...
    - Seam, Facelets, Shale, ...
- Part of a big sepc. Java Enterprise Edition 5.0
  - Java EE 5 enforces app servers to ship a JSF implementation.
  - Today it is already shipped by JBoss and SUN

# Integration (2)

- Integration with web portals (JSR 168) possible
  - Page contains several subapplications (portlets)
  - JSR-168 bridges (RI, MyFaces, Apache Portals)
- Supported by other web frameworks
  - Struts classic (Struts 1.2 and Struts 1.3)
    - Struts Integration Library (Craig McClanahan)
  - SAF2 (Struts Actions2 Framework)
    - special FacesInterceptors
    - Blog entry by Don Brown available
  - Cocoon has JSF support

# Tools

- run time:
  - every servlet container
  - Every Java EE 5.0 compliant Application Server has JSF „out of the box“.
- design time:
  - Sun One Studio Creator
  - Eclipse and MyEclipse, Exadel Studio
  - Oracle JDeveloper

# Development process

- with the “right” tools:
  - Drag&Drop:
    - Drag your components from a pallet to the page
  - wire the components to “backing beans”
  - creation of a persistence layer

# References

- Companies using Apache MyFaces

[http://wiki.apache.org/myfaces/Companies\\_Using\\_MyFaces](http://wiki.apache.org/myfaces/Companies_Using_MyFaces)

- Austria (for instance):
  - OeKB: Roncalli, ADAS, QMS, Gruppenkalender, Zeiterfassung
  - Prisma Kreditversicherungen: Prismanet, PrismaCIS
  - IRIAN GesmbH: <http://www.irian.at>

# Example

Wake up again ...



# JSF - Hello World (JSP file)

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<f:view>
<html><body>
<h:form id="form">
 <h:panelGrid columns="2">
 <h:outputLabel for="input1">
 <h:outputText id="input1Label" value="first name"/>
 </h:outputLabel>
 <h:inputText id="input1" required="true"
 value="#{customer.firstname}"/>

 <h:commandButton value="send it!"
 action="#{customer.send}"/>

 <h:messages style="color:red" layout="table"/>
 </h:panelGrid>
</h:form>
</body></html>
</f:view>
```

# JSF and JSP

- JSF Spec describes the support of JSP
  - Alternates possible (Facelets)
- JSP-Support via Taglibs
  - Core (the frameworks core)
    - like validation, conversion
  - HTML (renders “simple” markup (HTML 4.0.1))
    - <table/>, <input/>, ...

# JSF - Hello World (JavaBean)

```
public class Customer {

 private String firstname = null;
 //getter and setter

 public String send(){
 //back-end access (e.g. BusinessDelegate)
 return ("ok");
 }
}
```

# JSF - XML Config (1)

```
<managed-bean>
 <managed-bean-name>customer</managed-bean-name>
 <managed-bean-class>
 foo.Customer
 </managed-bean-class>
 <managed-bean-scope>request</managed-bean-scope>

 <managed-property>
 <property-name>firstname</property-name>
 <value>Gerald</value>
 </managed-property>

</managed-bean>
```

# JSF - XML Config (2)

```
<navigation-rule>
 <from-view-id>/form.jsp</from-view-id>
 <navigation-case>
 <from-action>#{customer.send}</from-action>
 <from-outcome>ok</from-outcome>
 <to-view-id>/output.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

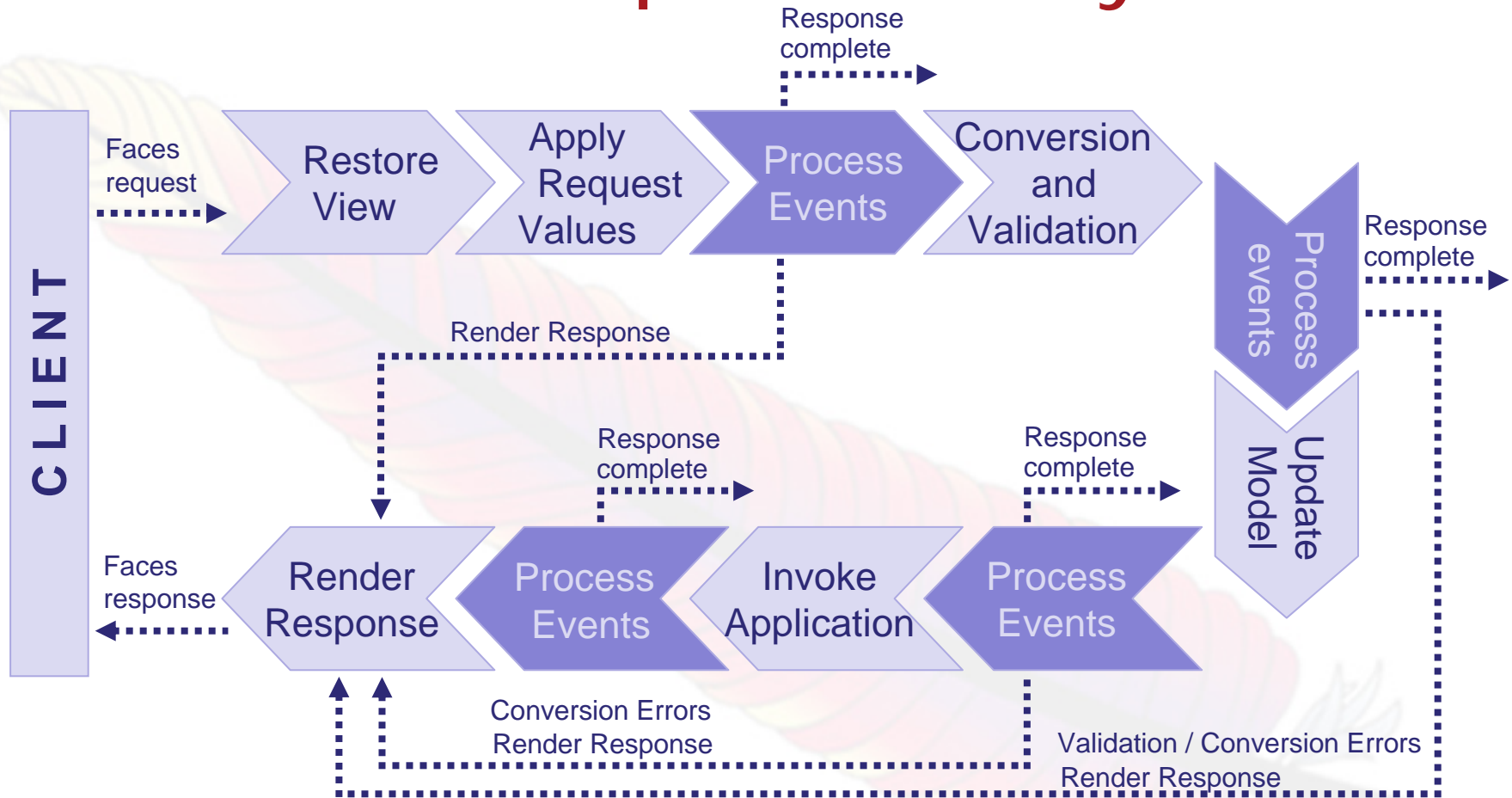
# Practice - Hello World

- Write a simple Hello World on your own.
- If you need help, SCREAM!

# JSF Request Life-Cycle

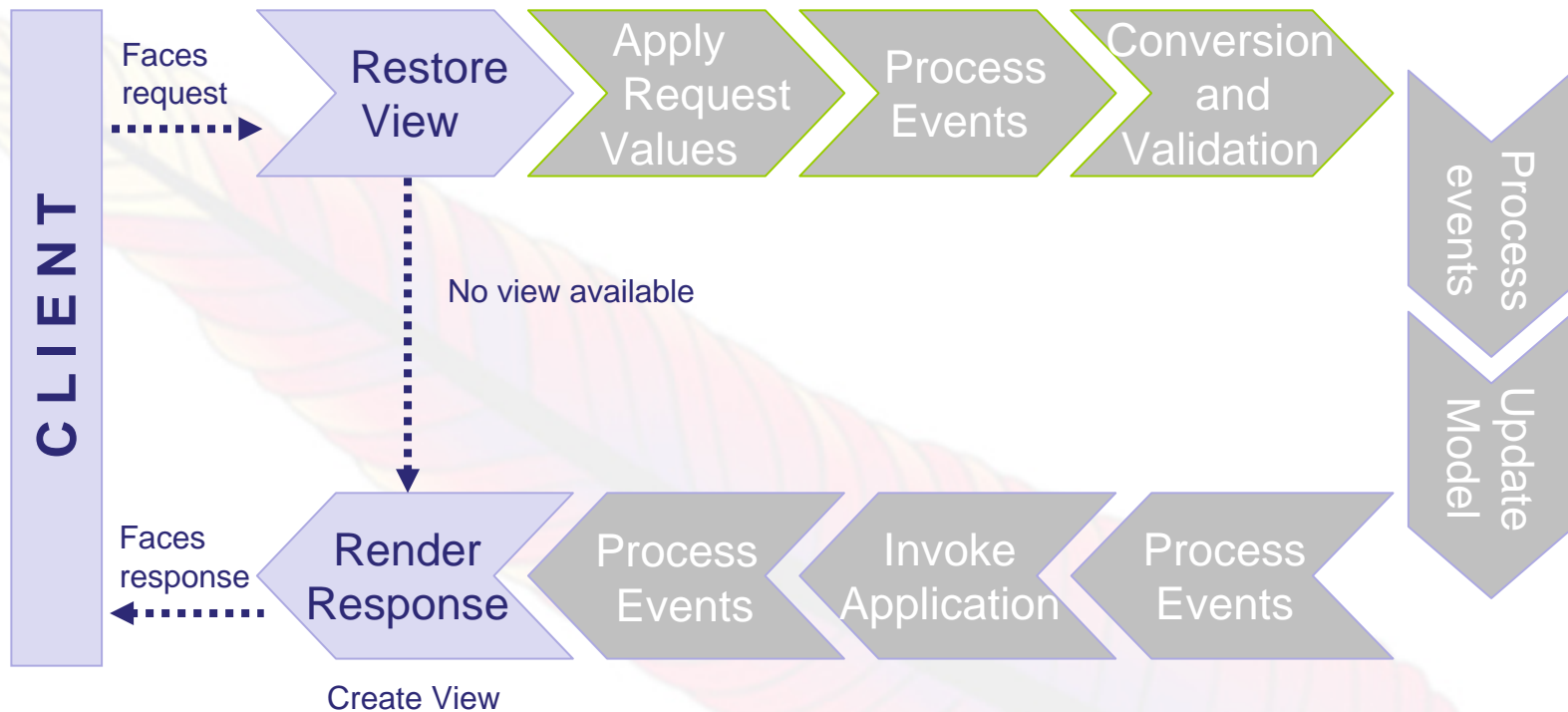
What's under the hood of a JSF request?

# JSF Request Lifecycle

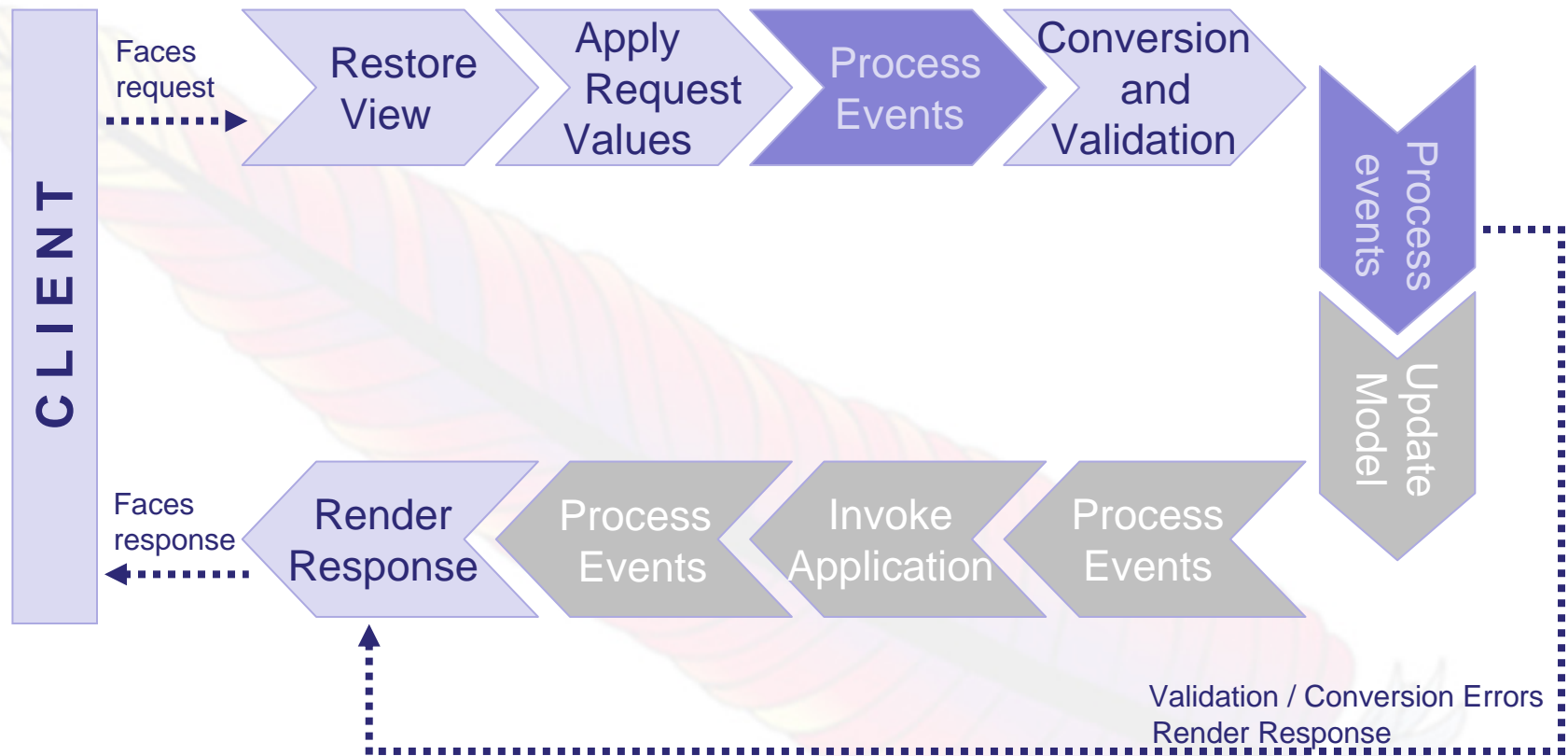




# JSF Lifecycle - first request



# JSF Lifecycle - Validation fails



# Restore View - Phase 1

- building the component tree
- first request (non-postback):
  - go to „Render Response“-Phase (Phase 6)
  - Use the template; create the tree during parsing the template
  - save the tree in the „state“
- Postback:
  - Create the tree from the „state“
  - Execute the lifecycle

# Apply Request Values - Phase 2

- decoding:
  - **processDecodes ( )** called recursive on each component in the tree (Starting at UIViewRoot)
  - every component takes care of it's value (reading HTTP-parameters, Cookies, Headers, etc.)
  - Saves the submitted value using **setSubmittedValue ( )**

# Process Validations – Phase 3

- Calls listener for the **ValueChangeEvent**
- conversion (!) and validation
  - **processValidators()**; called recursive, starts with `UIViewRoot`
  - **getSubmittedValue()**; like `"21.11.2005"`
  - converts to a object of class **java.util.Date**
  - enforce validation by calling the registered validators
  - save the correct value (**„local-value“**) by calling **setValue()**
- Error occurs on conversion or validation:  
GO TO **Render-Response**-Phase

# Update Model Values - Phase 4

- **processUpdates( ) ;** (again starting on UIViewRoot)
- component value is converted and valid; so it should be pushed to the model
- using the corresponding **backing bean**
  - **#{bean.property}**
- Calling the setter
  - **setProperty( )**

# Invoke Application - Phase 5

- **processApplication()** (UIViewRoot...)
- event handling for:
  - **action** / **actionListener**
  - executed
- sequence:
  - first **actionListener(s)**
  - calling **action** method

# Render Response - Phase 6

- Navigation: **NavigationHandler** determinates the next „view“ (a JSP page for instance)
- **ViewHandler** takes over - in case of JSP the (JSP)ViewHandler enforces a forward
- JSP page gets parsed by the JSP container. Performing a lookup for each component's
- **Renderer**. Calling several methods are called:
  - `encodeBegin()`; `encodeChildren()`; `encodeEnd()`;
    - Starting at `UIViewRoot`



# Changing the lifecycle (1)

- **immediate** property
- UICommand components:
  - action is called immediately. No validation or model update.
  - Use it for a cancel button
- UIInput components:
  - components value will be validated and converted in Apply Request Values
  - a ValueChangeEvent is generated and it's listener is called after "Apply Request Values"
  - calling facesContext.renderResponse() inside a ValueChangeListener  
→ go to "RenderResponse"
  - No conversion and validation of other (non immediate) components!

# Changing the lifecycle (2)

- Optional Validation Framework
  - for each request
  - optional switching validation on/off
  - „**required**“-attribute → own validator
  - many additional features
- Project: JSF-Comp at [sourceforge.net](http://sourceforge.net)

# Changing the lifecycle (3)

- No usage of JSF validation facility
- Do it yourself inside the **action** method
- WARNING: converter is still needed
- Maybe: special converter, which doesn't generate a error message

# Changing the lifecycle (4)

- Go to “Render-Response” by calling:  
**public void renderResponse();**
- Stopping the JSF lifecycle by calling:  
**public void responseComplete();**

# PhaseListener - configuration

- JSF provides a special Listener for the lifecycle
- PhaseListener executed at the beginning and at the end of a phase.
- register in faces-config.xml:

```
<lifecycle>
 <phase-listener>
 org.apache.confs.eu.PhaseListener
 </phase-listener>
</lifecycle>
```

# PhaseListener - Sample

```
public class DebugPhaseListener
 implements PhaseListener
{
 public void beforePhase(PhaseEvent
event){}

 public void afterPhase(PhaseEvent
event){
 System.out.println("afterPhase");
 }
 public PhaseId getPhaseId(){
 return PhaseId.ANY_PHASE;
 // return PhaseId.INVOKE_APPLICATION;
 }
}
```

# Exercise - Phaselistener

- Create your own PhaseListener class.
  - Use it to debug your JSF web app.
  - Register it to your faces-config.
  - Play with the application and look what's going on!
- 
- HEIP? Ask!

# terms ...

... from the JSF world



# terms (1)

- Component
- Renderer
- Converter
- Validator
- Event /  
EventListener
- Message /  
FacesMessage
- Action Method
- Model Objects
- View
- Navigation System /  
NavigationHandler
- Backing Bean /  
Managed Bean
- Value Binding

# Components

- interaction with the user
- server side (compared to Swing or SWT)
- support for IDEs b/c of JavaBean standard
- std. components: renderer independent
- know their state (StateHolder interface)
- stored in a tree structure (parent-client)
- unique id

# Renderer

- called by the component
- renders a special markup (HTML or WML)
- all Renderers belong to a RenderKit
- Renderer takes care of:
  - Encoding and Decoding

# digression: rendering

- Direct implementation model
  - Component -> encodeEnd -> HTML
- Delegated implementation model
  - Component -> encodeEnd-> Renderer->encodeEnd -> for instance HTML

# Converters

- data type in HTTP, HTML is “String”
- JSF backing beans: all type are possible
- by that fact, converting mechanism needed
- used for i18n and l10n
- converter choice based on data type
- custom converters

# connecting a converter

- as child element:

```
<h:outputText value="#{user.dateOfBirth}">
 <f:convertDateTime type="both"
 dateStyle="full" />
</h:outputText>
```

- build-in custom converter:

```
<h:outputText value="#{user.dateOfBirth}"
 converter="#{converterProvider.dateConverter}" />
```

# standard DateTimeConverter

- for date / time values:

```
<f:convertDateTime
 type=„date“ /*time, both*/
 dateStyle=„default“
 /*short, medium, long, full */
 timeStyle=„default“ /*-“-*/
 pattern=„dd.MM.yyyy HH:mm“
 timeZone=„GMT“
 locale=„en_US“ /*oder Locale*/ />
```

# standard NumberConverter

- Konvertieren von Zahlen:

```
<f:convertNumber
 type=„number“ /*currency, percentage*/
 currencyCode=„EUR“
 currencySymbol=„€“
 groupingUsed=„true“
 locale=„en_US“ /* oder Locale */
 minFractionDigits=„3“
 maxFractionDigits=„3“
 minIntegerDigits=„2“
 maxIntegerDigits=„2“
 pattern=„#.###,##“/>
```



# Example - converters

- Using the standard converters
  - converting a date
    - your birthdate
      - convert the input and the output
  - converting a number
    - your salary 😊
      - use the NumberConverter
- Need help? Bug us!!

# custom converter (1)

- implementiert `javax.faces.convert.Converter`
- optional (has arguments to save) `javax.faces.component.StateHolder`
- implements the methods:
  - `getAsString()`;
  - `getAsObject()`;
- on error:
  - `throw new ConverterException(FacesMessage msg)`

# custom converter (2)

- JSP-Tag possible (`extends ConverterTag`)
  - not needed for Facelets...
- JavaBean constructor that calls inside `setConverterId()`
- only setter for its properties
- overwrite `createConverter()`
- register in `faces-config.xml` / `tag.tld`
- register in `tag.tld` (only JSP)

# Overwriting a converter

- all converter are describe in `faces-config.xml`
- replace a standard convert with your custom:

```
<converter>
 <converter-for-class>
 java.util.Date
 </converter-for-class>
 <converter-class>
 org.apache.confs.eu.MyDateTimeConverter
 </converter-class>
</converter>
```

# Exercise - custom converter

- Write your first custom converter
- Implement it inside your backing bean
- Converting a `TelephoneNumber.java` object
  - `String countryCode`
  - `String areaCode`
  - `String number`
- Help needed ? ☺

# validation

- Checks the converted value against a special rule
- standard: done on the server (client side is possible...)
- per component approach
- validation error generates a FacesMessage
  - displayed with `<h:message/>` or `<h:messages/>`
- custom validators possible

# connecting a validator

- validation for mandatory values:

```
<h:inputText required="true"/>
```

- validation against a special scope (here a range):

```
<h:inputText>
```

```
 <f:validateLength minimum="2"
 maximum="10" />
```

```
</h:inputText>
```

# standard validators

- length:  
`<f:validateLength minimum=„3“  
maximum=„7“/>`
- range (long):  
`<f:validateLongRange minimum=„0“  
maximum=„1000“/>`
- range (double):  
`<f:validateDoubleRange minimum=„0.0“  
maximum =„0.5“/>`



# combining validator

- it's possible to combine validators
- sample:

```
<h:inputText
 value="#{backingBean.wert}">
 <f:validateLength
 minimum="1" maximum="4" />
 <f:validateLongRange
 minimum="0" maximum="1000" />
</h:inputText>
```

# Example - validator

- using the standards
  - declare a field you like as mandatory
    - required
  - check the length of your zip code
    - validateLength
- We'd like to help you!
  - or we drink a coffee ☺

# custom validation (1)

- pretty easy to define a custom validator

- implement a method like

```
public void validate(FacesContext,
UIComponent, Object) throws
ValidatorException
```

- wire the validation to the component:

```
<h:inputText
 value="#{backingBean.wert}"
 validator="#{backingBean.validate}"/>
```

# custom validation (2)

```
public void validate(
 FacesContext context,
 UIComponent component,
 Object value) throws ValidatorException
{
 if(value instanceof String)
 {
 String strValue = (String) value;

 if(!(strValue.equals("yes") &&
 !(strValue.equals("no")))
 {
 throw new ValidatorException(
 new FacesMessage(messageText, null));
 }
 }
}
```

## custom validation(3)

- implement the interface  
`javax.faces.validator.validator`
- Optional (arguments?)  
`javax.faces.component.StateHolder`
  - overwrite method:
    - `validate();`
  - On error:
    - `throw new ValidatorException(FacesMessage msg)`

# custom validation (4)

- JSP-Tag possible (`extends ValidatorTag`)
  - not needed when using Facelets
- JavaBean constructor
- setter for the properties
- overwrite `createValidator()`  
and call `setValidatorId()`
- register in `faces-config.xml`
- register in `tag.tld` (no need when using Facelets)

# Hands-on: custom validation

- create a simple custom validator inside of your backing bean
- check if the submitted value is a email address
  - We don't want you to use that RegExpr. stuff, so simple check if „@“ is inside the submitted String.
- Questions ?

# Events / EventListener

- fired due an event
- JSF defines four standard events:
  - FacesEvent (abstract)
  - ValueChangeEvent
  - ActionEvent
  - PhaseEvent
  - DataModelEvent (not a FacesEvent)



# ValueChangeListener (1)

- UIInput's **"valueChangeListener"** attribut
- Usage:

```
<h:inputText
 valueChangeListener="#{myForm.processValueChange}"/>
```

- the backing bean:

```
public void processValueChange(ValueChangeEvent event)
{
 HtmlInputText sender =
 (HtmlInputText)event.getComponent();
 sender.setReadOnly(true);
 changePanel.setRendered(true);
}
```

# ValueChangeListener (2)

- Using the JSP-Tag:

```
<h:inputText>
 <f:valueChangeListener type=„example.TestListener“/>
</h:inputText>
```

- The „example.TestListener“:

```
public class TestListener implements ValueChangeListener
{
 public void processValueChange(ValueChangeEvent event)
 throws AbortProcessingException {
 HtmlInputText sender = (HtmlInputText)event.getComponent();
 sender.setReadOnly(true);
 changePanel.setRendered(true);
 }
}
```

# Example - ValueChangeEvent

- Let's check some submitted values
- use the valueChangeListener attribute
- Print old and new value by using `System.out.println();`
- HELP ?!?

# ActionEvent

- UICommand's **"action"** attribut
- usage:
  - hard coded String (like "success")
  - use JSF' MethodBinding  
**"#{actionBean.newDetail}"**
- backing bean needs:  
**public String newDetail()**

# ActionListener

- UICommand's "**actionListener**" attribut
- usage:
  - MethodBinding.  
**"#{actionBean.newDetailListener}"**
- the method:  
**public void  
newDetailListener(ActionEvent e)**

# Example - Action Events

- add the „actionListener“ attribute to your button/link component
- create the method and do some `System.out.println()`;
- create the Method for the „action“ attribute
- add some `System.out.println()`; to your action method too
- What's happening?
- *We guide you, if needed.*

# return value from action method

- return value:
  - used to define the next view
  - described in faces-config.xml as
- a “Navigation-Rules”
  - from-view-id
  - or global
  - action source (method and outcome)
  - to-view-id

# Messages

- created when conversion/validation fails
- standard messages defined:  
**`javax.faces.Messages.properties`**
- JSP-Tag used for displaying them:  
**`<h:messages showSummary=„false“  
showDetail=„true“/>  
<h:message for=„componentId“/>  
!!!`**



# Messages

- standard messages don't fit to every use-case
  - validation message:
    - "{0}": Input required.
- overwrite them simple in your ResourceBundle
  - `javax.faces.component.UIInput.REQUIRED_detail = {0} please enter a value`

# Messages

- providing custom messagen (due to login error)
- **FacesContext.getCurrentInstance()**  
    **.addMessage(clientId, FacesMessage)**
- clientId = component's id (or null for global)
- new FacesMessage(FacesMessage.Severity  
    summary, detail)
  - **WARN, INFO, FATAL, ERROR**

# backing beans / managed beans

- POJO – Plain Old Java Objects
- Java Beans
  - „**public**“ constructor with no args
  - „**private**“ properties
  - „**getter**“/“**setter**“
- declare them in **faces-config.xml**

# backing beans / managed beans

- possible scopes
  - application (one instance per application)
  - session (one instance per session/user)
  - request (one instance per request)
  - none (bean created due to a reference)

# ValueBinding / ValueExpression

- ValueBinding
  - Wire attribute to a backing bean

- usage

```
<h:outputText
 value="#{user.surName}" />
```

- property "surName" from the bean "user"

# JavaServer Faces standard components

components, components, components

# standard components - Text

- outputText

```
<h:outputText
```

```
value="#{user.userNameDescr}" />
```

- inputText

```
<h:inputText
```

```
value="#{user.userName}" />
```

Benutzer:\*

Kennwort:\*

# standard components - UICommand

- commandLink

```
<h:commandLink
action="#{actionBean.test}"/>
```

- commandButton

```
<h:commandButton
action="#{actionBean.test2}"/>
```

Benutzer:\*

Kennwort:\*

Anmelden



# standard components -

## OutputLink

- `HtmlCommandLink` used for postbacks.
- Linking other websites:  
`<h:outputLink value=„url“  
target=„_blank“/>`
- Caution!: state get's lost, since this is not a postback
- HTTP parameters:  
`<h:outputLink value=„url“>  
  <f:param name=„allowCache“  
  value=„true“/>  
</h:outputLink>`

# standard components - UIForm

- ```
<h:form title="Form">  
  <h:outputText value=  
    "Enter a value." />  
  <h:inputText />  
  <h:commandButton value="Submit" />  
</h:form>
```
- JSF: every submit is a POST
→ Caution: **commandLink** needs a **form**

standard components - UIPanel

- Doing Layout with JSF
- renders a HTML span element:
 - `<h:panelGroup>...</h:panelGroup>`
- renders a HTML table:
 - `<h:panelGrid
columns=„2“>...</h:panelGrid>`
 - amount of components must be a multiple of
„columns“
 - If not use empty `<h:panelGroup/>`

standard components - UIData

- best for presenting structured data (like `java.util.List`)
- **horizontal**: every column is defined by a **UIColumn** component
- **vertical**: each row represents one item of the structured data
- Facets (**<f:facet />**) allow defining header and footer

standard components - UIData

- Example:

```
<h:dataTable value="#{table.items}" var="item">
  <h:column>
    <h:outputText value="#{item.column1}"/>
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Header in column 2"/>
    </f:facet>
    <h:outputText value="#{item.column2}"/>
  </h:column>
</h:dataTable>
```

standard components - Image

- usage:

```
<h:graphicImage id="Grafik" url="/images/Grafik.jpg"  
  alt="#{bundle.chooseLocale}" title="Grafikanzeige"  
  width="149" height="160"/>
```

- No component for an “ImageMaps” defined inside the JSF Spec.

standard components - UIInput

- text input
 - `<h:inputText/>`
- password input
 - `<h:inputSecret/>`
- hidden field:
 - `<h:inputHidden/>`
- textareas
 - `<h:inputTextarea/>`

standard components - Label

- Label for a component

```
<h:outputLabel for=„myId“  
    value=„#{bean.labelText}“/>  
<h:selectOneRadio id=„myId“ value=„something“/>
```

- Apache MyFaces: label text can be used in FacesMessage

standard components - Format

- parameterised output:

```
<h:outputFormat value="Hello {0}. \
    second value is {1}. \
    Have a good one, {0}.">
    <f:param value="{user.firstName}"/>
    <f:param value="hartcodiert"/>
</h:outputFormat>
```

- Important for i18n and l10n

standard components - UISelectBoolean

- input field for **boolean/Boolean** values
- like:

```
<h:selectBooleanCheckbox  
  title=„yesOrNo“  
  value=„#{bean.yesOrNo}“ />
```

standard components -

UISelectMany

- Choose more than one input value
- JSP tags:
 - `<h:selectManyCheckbox>`
 - `<h:selectManyListbox/>`
 - `<h:selectManyMenu/>`
- rendered as
 - list of checkboxes,
 - html listbox,
 - or as a menu (not that good one...)

standard components - UISelectOne

- Choose one value
- components:
 - `<h:selectOneRadio>`
 - `<h:selectOneListbox/>`
 - `<h:selectOneMenu/>`
- rendered:
 - list of radio fields,
 - listbox,
 - or as a combobox

standard components - UISelectItem(s)

- use them in `<h:selectManyXXX/>` or `<h:selectOneXX/>`
- like:
 - `<f:selectItem itemValue=„...” itemLabel=„...”/>`
 - `<f:selectItems value=„#{bean.itemsList}”/>`
 - Array, Collection mit SelectItem
 - `Map.put(String, SelectItem);`

standard components - UISelectItem(s)

- combine the `<f:selectItem(s) />`
- use `<f:selectItem/>` for an empty entry
- pick the real choices from a `java.util.List`
- `<h:selectOneMenu
id="betreuerWahl"
value="#{bean.auswah}">

 <f:selectItem />
 <f:selectItems value="#{bean.list}" />

</h:selectOneMenu>`

Creating JSF views

- All JSP-Tags of JSF must be inside the root:

<f:view/>

(UIViewRoot).

- If you need **<jsp:include/>** or **<c:import/>** wrap them with:

<f:subview/> (UINamingContainer)

- Needed for Tiles integration too!

MessageBundles

- i18n:

```
<f:loadBundle  
  basename=„org.apache.conf.eu.messages“  
  var=„messages“ />
```
- Usage
 - messages.properties: `test1=hallo`
 - your.jsp:

```
<h:outputText  
  value=„#{messages[‘test1’]}“ />
```
- Caution:
 - values get only set when parsing the JSP
(since this is a JSP tag, not a component)
 - problems with Facelets

JSF 1.1 workarounds - verbatim

- the “JSF and JSP” combination has problems, when using plain HTML inside your page
- embedded HTML output is rendered directly; JSF goes to a buffer...
- work around:
 - wrap all plain HTML inside a
 - **<f:verbatim>...</f:verbatim>**
 - simply adds a HtmlOutputText component
 - fixed in JSF 1.2 spec

Unified Expression Language

Value- and Method-Expressions

Unified EL

- JSP EL
- JSF EL
 - JSF 1.2 Unified EL
- JSF EL Syntax refers to JSP EL
- but: JSF EL – expressions are evaluated deferred, JSP – EL immediate

Samples for the UL (1)

- `value="#{user.username}„`
- `value="#{person.address.street}"`
- `rendered="#{user.username != null}„`
- `value="#{bill.sum * 13.7603}"`
- `style="#{grid.displayed ?
 'display:inline;' :
 'display:none;'}"`
- `value="Hallo Benutzer
 #{user.username}"`

Samples for the UL (2)

- `action="#{user.storeUser}"`
- `actionListener="#{dtBean.deleteRow}"`
- `value="#{mapBean['index']}"`
- `value="#{mapBean[user.username]}"`
- `value="#{listBean[5]}"`

configuration

JSF configured the right way

configuration (1)

- required: copy JSF/MyFaces jar-files to WEB-INF/lib
- register FacesServlet inside web.xml
- edit your faces-config.xml file for further JSF configurations like
 - backing beans
 - components ...

faces-config.xml - managed beans

- managed beans:

```
<managed-bean>
```

```
  <description>The one and only  
    HelloBean.</description>
```

```
  <managed-bean-name>helloBean  
</managed-bean-name>
```

```
  <managed-bean-class>  
    org.apache.hello.HelloBean  
</managed-bean-class>
```

```
  <managed-bean-scope> request  
</managed-bean-scope>
```

```
</managed-bean>
```

- Scope: application, session, request, none

faces-config.xml - navigation rules

- the navigation rules:

```
<navigation-rule>
  <from-view-id>
    /limit_list.jsp <!-- * ... global -->
  </from-view-id>
  <navigation-case>
    <from-outcome>show_item</from-outcome>
    <to-view-id>/limit_detail.jsp
  </to-view-id>
  </navigation-case>
</navigation-rule>
```

faces-config.xml - enhanced

- JSF is customisable
- inside **<application/>** element
- providing of custom **ActionListener, ViewHandler, NavigationHandler, ELResolver, StateManager** possible
- setting of l10n
- this is a central point!

web.xml - what is needed?

- FacesServlet:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    <!--MyFaces:
    org.apache.myfaces.webapp.MyFacesServlet-->
    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/*
    <!-- *.faces -->
</url-pattern>
</servlet-mapping>
```

web.xml - JSF config

```
<context-param>  
  <param-name>  
    javax.faces.CONFIG_FILES</param-name>  
    <param-value>  
      /WEB-INF/examples-config.xml  
    </param-value>  
    <description>  
      Comma separated list of URIs of  
      (additional) faces config files.  
      (e.g. /WEB-INF/my-config.xml)  
      See JSF 1.0 PRD2, 10.3.2  
    </description>  
</context-param>
```

web.xml - state saving

```
<context-param>  
  <param-name>  
    javax.faces.STATE_SAVING_METHOD  
  </param-name>  
  <param-value>client</param-value>  
  <description>  
    State saving method: "client" or  
    "server" (= default) See JSF  
    Specification 2.5.2  
  </description>  
</context-param>
```

Apache MyFaces

First Free Open Source JSF Implementation

JSF Implementations

- Sun (RI)
 - IBM
 - Apache MyFaces
 - Simplicia (based on Apache MyFaces)
- additionally, there are several 3rd party UI components that *should* run with *any* implementation.

Apache MyFaces

- Founded in 2002 by Manfred Geiler and Thomas Spiegl, CEO of IRIAN.at
 - sourceforge and LGPL based
- In July 2004: move to Apache Software Foundation (Incubator)
- Since February 2005 TLP (myfaces.apache.org)
- 25 developers
- currently 1.1.1

MyFaces provides:

- Implementation of JSF-API
 - `javax.faces.**` Classes
- Implementation of JSF Spec
 - `org.apache.myfaces.**` Classes
- Custom Components
 - Scrollable Table, Validator, Tree components ...
- Custom extensions
 - Built-in Tiles-Support, RenderKit for WML/WAP
- Support for Portlet Spec (JSR 168)
 - MyFaces apps runs in Pluto, JBoss Portal and some others.

JAR files of Apache MyFaces

- myfaces-impl.jar
- myfaces-jsf-api.jar
- tomahawk.jar
- sandbox.jar
- myfaces-all.jar (all in one jar file - except sandbox)

MyFaces compatibility (tested)

- Java 1.4 and Java5
- Tomcat (4.1.x, 5.0.x and 5.5.x)
- JBoss (3.2.x and 4.0.x)
- JRun4
- Bea Weblogic 8.1
- Jonas 3.3.6 w/ Tomcat
- Resin 2.1.x
- Jetty 4.2
- Websphere 5.1.2
- OC4J

MyFaces Internals I

- ExtensionsFilter
 - used during upload (parses Multipart requests)
 - adds resources (images, js,...) that are needed by components (easier to reuse components)
 - good performance

MyFaces Internals II

- special Servlet Context parameter
 - ALLOW_JAVASCRIPT
 - DETECT_JAVASCRIPT
 - AUTO_SCROLL
 - PRETTY_HTML
- dummy form for commandLinks

MyFaces in Action

- several custom components
- custom validator components
- custom extensions

Custom calendar component

- Renders as a form:

```
<x:inputCalendar ...  
  value="#{travel.arrival}" />
```

- Renders as a popup:

```
<x:inputCalendar ...  
  renderAsPopup="true"  
  value="#{travel.depature}" />
```

- Sample

Custom Upload Component

- Upload is not part of JSF spec (currently)
- uses Servlet Filter (MyFaces' Extension Filter)
- special interface
(`org.apache.myfaces.custom.fileupload.UploadedFile`)

```
<h:form enctype="multipart/form-data">  
  <x:inputFileUpload  
    value="#{backing.file}"  
    required="true" />  
  ...  
</h:form>
```

- Sample

Tree Component (Tree2)

- MyFaces provides two tree components
- define your data inside a backing bean
 - TreeNode (Interface)
 - TreeNodeBase (Implementation class)
- define your layout in a JSF page via facets
- Navigation via CommandLink component
- client and server toggle

Tree Component Java code

```
private TreeNode tree;  
tree = new  
    TreeNodeBase(„folder“, „navi“, true) ;  
  
tree.getChildren().add(  
new TreeNodeBase(„doc“, „entry“, false)  
)
```

Tree Component JSP

```
<x:tree2 value=„#{bean.tree}" clientSideToggle=„true" var=„node"
    varNodeToggle=„t" ...>
<f:facet name=„doc">

<h:panelGroup>
    <h:commandLink styleClass="document" action=„nav">
    <h:graphicImage value="images/document.png"
        border="0" />

    <h:outputText value="#{node.description}" />
    <f:param name=„reqVal" value="#{node.identifier}" />
    </h:commandLink>
</h:panelGroup>

</f:facet>
...
</x:tree2>
```

Sample

Tabbed Pane

- Tab control as known from classic GUIs
- Contains two custom JSF tags
 - `<x:panelTabbedPane />`
 - `<x:panelTab />`
- reuses standard UI components
 - for instance `<h:inputText />`
- click on a tab ends up in a request, but tab saves the state of the nested input fields

Tabbed Pane JSP code

```
<x:panelTabbedPane bgcolor=„#FFFFCC“>

  <x:panelTab id=„tab1“ label=„Main Menu“>
    <h:outputText .../>
    <h:inputText value=„#{bean.property}“/>
    ...
  </x:panelTab>
  <x:panelTab id=„tab2“ label=„second Menu“>
    ...
  </x:panelTab>
  <h:commandButton value=„Submit it!“ />
</x:panelTabbedPane>
```

- Sample

custom Table component

- MyFaces contains a custom table component
- extends UIData (standard component)
 - preserveDataModel
 - preserveRowState
 - sortColumn
 - sortAscending
 - preserveSort
 - renderedIfEmpty
 - rowIndexVar

scrollable Table component

```
<x:dataTable id="data" ...>
```

```
...
```

```
</x:dataTable>
```

```
<x:dataScroller id="scroll_1" for="data" fastStep="10"  
  pageCountVar="pageCount" pageIndexVar="pageIndex"  
  styleClass="scroller" paginator="true"  
  paginatorMaxPages="9" paginatorTableClass="paginator"  
  paginatorActiveColumnStyle="font-weight:bold;">
```

```
<f:facet name="first" >
```

```
<h:graphicImage url="images/arrow-first.gif" border="1" />
```

```
</f:facet>
```

```
...
```

```
</x:dataScroller>
```

Sample

sortable Table component

- needs MyFaces `<x:dataTable/>` attributes:
 - `sortColumn="#{sorter.sort}"`
 - `sortAscending="#{sorter.asc}"`
 - `preserveSort="true"`
- uses MyFaces `<x:dataTable/>` BackingBean needs method `(sort())` that contains a `Comparator` impl.
- call `sort()` before return the data model.
 - here: call inside of `getWorkers()`;
- Sample

Using *Legacy* JavaScript

- JSF Components using IDs:

```
<h:form id=„foo“>  
<h:inputText id=„bar“ ... >  
</h:form>
```

generates foo:bar

- document.getElementById();
- special forceId Attribute (JSF 1.2 contains a similar concept):

```
<h:form id=„foo“>  
<x:inputText id=„bar“ forceId=„true“... >  
</h:form>
```

generates bar

Custom Validators

- nest them inside Input Components

```
<h:inputText value=„...>  
  <x:validateEmail/>  
</h:inputText>
```

- ISBN (<x:validateISBN/>)
- CreditCard (<x:validateCreditCard/>)
- Regular Expression
 <x:validateRegExpr pattern=„\d{5}\"/>

- Equal

```
<h:inputText id=„password1“ ...>  
<h:inputText id=„password2“ ...>  
  <x:validateEqual for=„password1\"/>  
</h:inputText>
```

JSF - composing pages

- Standard provides a plain subview component
 - `<jsp:include />` or `<c:import />`
- realizes the Composite View Pattern
- bound to file names (e.g. `footer.jsp`)
- good framework for composing pages
 - Tiles (used in Struts, Velocity or plain JSP)

MyFaces Tiles integration

- custom ViewHandler for Tiles
 - must be registred in `faces-config.xml`
 - needs tiles configuration location as ContextParameter (`web.xml`)
 - looks up `*.tiles` mappings in tiles definition file
 - page definitions are described in `tiles.xml`
 - known issues
 - none-tiles pages must be wrapped inside of tiles config

MyFaces/Tiles - definitions

```
<tiles-definitions>
<definition name="layout.example"
  path="/template/template.jsp" >
  <put name="header" value="/common/header.jsp" />
  <put name="menu" value="/common/navigation.jsp" />
</definition>

<definition name="/page1.tiles"
  extends="layout.example" >
  <put name="body" value="/page1.jsp" />
</definition>

<!-- workaround for non-tiles JSF pages-->
<definition name="non.tiles1" path="/non-tile.jsp"/>
```

MyFaces/Tiles - master template

```
<table>
<tr><td>
<b:f:subview id="menu">
  <tiles:insert attribute="menu" flush="false"/>
</f:subview>
</td>

<td>
<f:subview id="body">
  <tiles:insert attribute="body" flush="false"/>
</f:subview>
</td>
</tr>
</table>
```

MyFaces' WML RenderKit

- supports basic JSF components to render in WAP devices
- supports WML and not XHTML MP (WAP2.0)
- add WML RenderKit to `faces-config.xml`
- uses XDoclet to generate components, tag classes and tld file
- contribution from Jiri Zaloudek

WML RenderKit - code

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">

<%@ page contentType="text/vnd.wap.wml" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://myfaces.apache.org/wap" prefix="wap" %>

<wml> <card id="helloId" title="Hello WML World">
<p> <f:view>

<wap:form id="form">
<wap:outputText id="label" value="Your name"/>
<wap:inputText id="name" value="#{hello.yourname}" />
<wap:commandButton id="submit" action="#{hello.send}"
    value="submit it" />
</wap:form>
</f:view>

...
SAMPLE
```


MyFaces - Portlet support

- Built-in-support for JSR 168
- contribution by Stan Silvert (JBoss Group)
- what must a user do?
 - Make sure your JSF MyFaces application runs as a stand-alone servlet.
 - Remove any redirects from your faces-config.xml. Portlets can not handle these.
 - Create a Portlet WAR as per the instructions for your Portlet container. Make sure it contains everything that was included in step 1.
 - Update your portlet.xml

MyFaces - portlet.xml

```
<portletclass>  
org.apache.myfaces.portlet.MyFacesGenericPortlet  
</portlet-class>  
  
<init-param>  
  <name>default-view</name>  
  <value>/some_view_id_from_faces-config</value>  
</init-param>  
  
<init-param>  
  <name>default-view-selector</name>  
  <value>com.foo.MyViewSelector</value>  
</init-param>
```

Long term visions for MyFaces

- Big TLP for JSF in general
 - Apache Faces
- MyFaces provides only API and impl
 - MyFaces should be a subproject of Apache Faces
 - more a dream, currently :-)
- JSF 1.2 compliant implementation
- WML RenderKit integration ... :-)

writing custom components

when the standard doesn't solve your
problems ...

preparatory work

- What is your super class ?
`UIOutput`, `UIInput`, `UISelectOne`,
`UISelectMany`, `UICommand`, `UIPanel`
- classification: component family,
component type and renderer type to be
defined

Examples:

- `org.apache.myfaces.HtmlInputText`
 - component type:
`org.apache.myfaces.HtmlInputText`
 - component family: `javax.faces.Input`
 - renderer type: `org.apache.myfaces.Text`
- `javax.faces.component.html.HtmlInputText`
 - component type: `javax.faces.HtmlInputText`
 - component family: `javax.faces.Input`
 - renderer type: `javax.faces.Text`

Examples

- `org.apache.myfaces.custom.tabbedpane.HtmlPanelTabbedPane`
 - component type:
`org.apache.myfaces.HtmlPanelTabbedPane`
 - component family: `javax.faces.Panel`
 - Renderertyp: `org.apache.myfaces.TabbedPane`
- `org.apache.myfaces.custom.navmenu.UINavigationMenuItem`
 - Komponententyp:
`org.apache.myfaces.NavigationMenuItem`
 - Komponentenfamilie: `javax.faces.SelectItem`
 - Renderertyp: `null`

classification by:

- component class:
 - `UIComponent.getComponentType()`;
 - `UIComponent.getComponentFamily()`;
 - in constructor:
`super.setDefaultRendererType(
 DEFAULT_RENDERER_TYPE);`
- JSP-Tag class
 - `UIComponentTag.getComponentType()`;
 - `UIComponentTag.getRendererType()`;

Tag-Library-Definition TLD

- standard JSP taglib file:

```
<!-- commandButton -->
<tag>
  <name>commandButton</name>
  <tag-class>
    org.apache.myfaces.taglib.html.ext.HtmlCommandButtonTag
  </tag-class>
  <body-content>JSP</body-content>
  <description>
    Extended standard commandButton
  </description>
  <attribute>
    <name>action</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <type>java.lang.String</type>
  </attribute>
  ...
</tag>
```

A JSP-Tag class for JSF components

- setXXX() for each property
- release() method:
set every property back to “null”
- implement the setProperties(); method

A JSF/JSP-Tag class

```
protected void setProperties(UIComponent component)
{
    super.setProperties(component);
    setStringProperty(component,
        HTML.TABINDEX_ATTR, _tabindex);
    setStringProperty(component,
        HTML.TYPE_ATTR, _type);
    setActionProperty(component, _action);
    setActionListenerProperty(component,
        _actionListener);
    setBooleanProperty(component,
        JSFAttr.IMMEDIATE_ATTR, _immediate);
    setStringProperty(component, JSFAttr.IMAGE_ATTR,
        _image);
}
```

component class

- JavaBean std. (getter/setter for property)
 - Caution! Take care of JSF's ValueBinding
- Overwrite `restoreState()` and `saveState()` to be able to save the component state
- if needed methods for `EventListener` (like JavaBean std.)

the getter / setter

```
public void setValue(Object value)
{
    _value = value;
}

public Object getValue()
{
    if (_value != null) return _value;
    ValueBinding vb = getValueBinding("value");
    return vb != null ?
        (Object)vb.getValue(getFacesContext()) : null;
}
```

StateHolder's saveState

```
public Object saveState(FacesContext context)
{
    Object values[] = new Object[6];
    values[0] = super.saveState(context);
    values[1] = saveAttachedState(context,
        methodBindingActionListener);
    values[2] = saveAttachedState(context,
        actionExpression);
    values[3] = immediate ? Boolean.TRUE :
        Boolean.FALSE;
    values[4] = immediateSet ? Boolean.TRUE :
        Boolean.FALSE;
    values[5] = value;

    return (values);
}
```

StateHolder's restoreState

```
public void restoreState(FacesContext context,
    Object state)
{
    //Die Variable "state" speichert den Zustand
    //der Komponente als Feld von Objekten
    Object values[] = (Object[]) state;

    //Rücksichern des vererbten Status
    super.restoreState(context, values[0]);

    //Rücksichern der Attribute der Komponente
    methodBindingActionListener = (MethodBinding)
        restoreAttachedState(context, values[1]);
    actionExpression =
        (MethodExpression) restoreAttachedState(context, values[2]);
    immediate = ((Boolean) values[3]).booleanValue();
    immediateSet = ((Boolean) values[4]).booleanValue();
    value = values[5];
}
```

Renderer

```
public abstract class Renderer {  
  
    public void decode(FacesContext context,  
        UIComponent component){  
  
    public void encodeBegin(FacesContext context,  
        UIComponent component)  
        throws IOException {}  
  
    public void encodeChildren(FacesContext context,  
        UIComponent component)  
        throws IOException {}  
  
    public void encodeEnd(FacesContext context,  
        UIComponent component)  
        throws IOException {}  
}
```


Renderer

```
public String convertClientId(FacesContext
    context, String clientId) {}

public boolean getRendersChildren() {}

public Object getConvertedValue(FacesContext
    context, UIComponent component,
    Object submittedValue)
    throws ConverterException {}
}
```

Renderer - encodeEnd

```
RendererUtils.checkParamValidity(facesContext,  
    uiComponent, UICommand.class);  
  
String clientId = uiComponent.getClientId(facesContext);  
  
ResponseWriter writer = facesContext.getResponseWriter();  
  
writer.startElement(HTML.INPUT_ELEM, uiComponent);  
  
writer.writeAttribute(HTML.ID_ATTR, clientId,  
    JSFAttr.ID_ATTR);  
writer.writeAttribute(HTML.NAME_ATTR, clientId,  
    JSFAttr.ID_ATTR);  
  
...
```

alternatives

instead of a custom component

substitute the renderer

- a renderer implements
 - encoding (encodeXXX())
 - decoding (decode())
 - converting process
- You can substitute a renderer. Often this helps!

substitute the renderer

- this is done global
 - for each objects of a component
 - the new renderer will be used every time!
- with the used RenderKit
 - a RenderKit contains all used renderers.
 - Only one RenderKit per JSF app
 - possible to change ...

substitute the renderer

- faces-config.xml:

```
<render-kit>
  <render-kit-id>HTML_BASIC</render-kit-id>
  <renderer>
    <component-family>
      javax.faces.Output</component-family>
    <renderer-type>
      javax.faces.Label</renderer-type>
    <renderer-class>
      mypackage.RequiredLabelRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

the renderer class

```
public class RequiredLabelRenderer extends HtmlLabelRenderer {
    protected void encodeBeforeEnd( FacesContext facesContext,
        ResponseWriter writer, UIComponent uiComponent) throws
        IOException {
        String forAttr = getFor(uiComponent);
        if(forAttr!=null) {
            UIComponent forComponent =
                uiComponent.findComponent(forAttr);

            if(forComponent instanceof UIInput &&
                ((UIInput) forComponent).isRequired()) {
                writer.startElement(HTML.SPAN_ELEM, null);
                writer.writeAttribute(HTML.ID_ATTR,
                    uiComponent.getClientId(facesContext)+
                    "RequiredLabel",null);
                writer.writeAttribute(HTML.CLASS_ATTR,
                    "requiredLabel",null);
                writer.writeText("*",null);
                writer.endElement(HTML.SPAN_ELEM);
            }
        }
    }
}
```

provide a JSP-Tag

- without a new Tag every <h:outputLabel uses the new renderer
- maybe confusing to the users
- change the renderer type

```
public String getComponentType() {  
    return ("javax.faces.HtmlOutputLabel");  
}  
  
public String getRendererType() {  
    return ("de.jax.RequiredLabel");  
}
```


substitute component class

- component contains properties
- encoding, decoding and conversion is also included into a component!
- validation customisable
- You can replace a component globally, means for all JSP-Tags (like the renderer).

substitute component class

- faces-config.xml:

```
<component>  
  <component-type>  
    javax.faces.HtmlInputText</component-type>  
  <component-class>  
    mypackage.SpecialHtmlInputText  
  </component-class>  
</component>
```

the component class

```
public class SpecialHtmlInputText extends
    HtmlInputText {
    public SpecialHtmlInputText()
    {
        super();

        setConverter(ConverterFactory.
            getSpecialConverter());
    }
}
```

component binding

- ValueBinding != component binding
- uses JSF EL:
 - „binding=„#{bean.myComponent}“
- return of special / own components, which fit the desired type, is possible

component binding

- JSP:

```
<h:outputText  
  value="#{limitDetail.limitView.comment}","  
  binding="#{componentBean.  
    outputWithBreaks}" />
```

- backing bean:

```
UIComponent getOutputWithBreaks()  
{  
    return new OutputTextWithBreaks();  
}
```

component binding

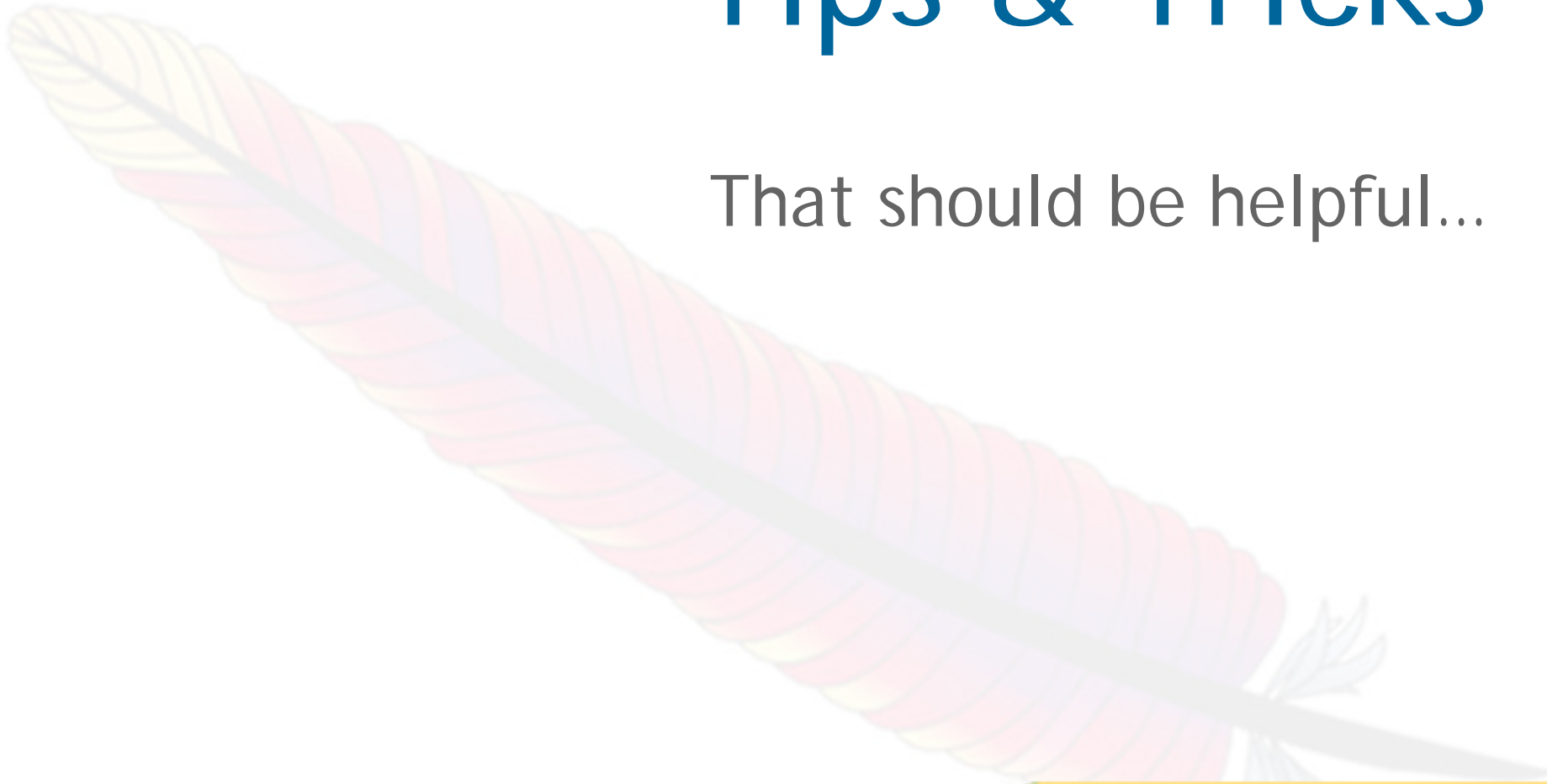
```
public static final class OutputTextWithBreaks extends HtmlOutputText
{
    public OutputTextWithBreaks()
    {
        super();
    }

    public void encodeEnd(FacesContext context) throws
                        IOException
    {
        String text = RendererUtils.getStringValue(
            context, this);
        text = HTMLEncoder.encode(text, true, true);

        //Erstellen aller Zeilenumbrüche
        text = text.replaceAll("\r", "<br/>");
        renderOutputText(context, this, text, false);
    }
}
```

Tips & Tricks

That should be helpful...



dynamic relies not on a JSP

adding components to the component tree:

```
public void addControls(ActionEvent actionEvent)
{
    Application application =
    FacesContext.getCurrentInstance().getApplication();
    List children = controlPanel.getChildren();
    children.clear();
    for (int count = 0; count < numControls; count++)
    {
        HtmlOutputText output = (HtmlOutputText)application.
        createComponent(HtmlOutputText.COMPONENT_TYPE);
        output.setValue(" " + count + " ");
        output.setStyle("color: blue");
        children.add(output);
    }
}
```


ActionListener for Navigation

- inside the ActionListener:

```
FacesContext.getCurrentInstance().  
    getApplication().getNavigationHandler().  
        handleNavigation(  
            FacesContext.getCurrentInstance(),  
            null, outcome);
```

- Needs:
 - global **navigation-rule** for the String **outcome**

Using HTML inside OutputText

- the tag:
 - `<h:outputText value=„#{bean.htmlText}\"/>`
- Problem: HTML will be „escaped“
- like: `
` → `
`
- work around:
 - `<h:outputText
value=„#{bean.htmlText}\"
escape=„false\"/>`

passing arguments with the EL

- EL expressions are powerful, but ...
 - ... don't take arguments
- work around:
 - backing bean implements Map interface
 - On a **Map.get („key“)** call, the method get's called and a argument is passed through (**„key“**)
 - usage: **# {mapBean[,key`] }**

Master Detail (1)

- Liste:

```
<h:dataTable var="bean" value=...>
```

```
...
```

```
<h:commandLink  
    actionListener=„#{bean.editItem}“ />
```

```
...
```

```
</h:dataTable>
```

- using commandLink for editing the details
- actionListener instead of action

Master Detail (2)

- backing bean:

```
public void editItem(ActionEvent ev)
{
    UIData datatable =
        findParentHtmlDataTable(
            ev.getComponent());
    Item item = (Item)
        datatable.getRowData()
    //edit the item...
}
```

Master Detail (3)

- helper method:

```
private HtmlDataTable findParentHtmlDataTable(
    UIComponent component)
{
    if (component == null)
    {
        return null;
    }
    if (component instanceof HtmlDataTable)
    {
        return (HtmlDataTable) component;
    }
    return findParentHtmlDataTable(
        component.getParent());
}
```

Master Detail (4)

- other possibilities:
 - `<f:param ... />` (well, ok...)
 - Apache MyFaces:
`<t:updateActionListener />`
- `<t:updateActionListener />`
 - When an action is called the “value” is set to a backing bean’s property
 - `<t:updateActionListener
property="#{countryForm.id}"
value="#{country.id}" />`

showing/ hiding components

- „**rendered**“ attribute:
 - should a component be rendered ?
 - JSP: **<h:inputText
rendered=„#{bean.showMe}“ />**
- replacement for „**c:if**“ or ugly Java code (scriptlets)
- Warning:
 - **rendered** evaluated during each phase
 - also on a postback (no decoding for not rendered components)

Links and books

- MyFaces AJAX examples
 - http://www.irian.at/open_source.jsf
(sandbox components)
- AJAX web resources
 - <http://www.adaptivepath.com>
 - <http://www.ajaxinfo.com/>
 - http://www.ajaxpatterns.org/Ajax_Frameworks
 - <http://www.ajaxdeveloper.org>

Literature

- Mann, Kito D. (2005): Java Server Faces in Action. Manning, Greenwich
- Hall, Marty (2001): JSF. A quick introduction to JavaServer Faces. <http://www.coreservlets.com/JSF-Tutorial/>
- Bergsten, Hans (2004): JavaServer Faces. O'Reilly.
- Dudney, Bill et. al (2004): Mastering JavaServer Faces. Wiley
- Ed Burns et.al (2004): JavaServer Faces (Version 1.1.) Specification.