FH Bielefeld
University of
Applied Sciences

Campus Minden

Webbasierte Anwendungen

SS 2018

Java Server Faces

Dozent: B. Sc. Florian Fehring
mailto: florian.fehring@fh-bielefeld.de

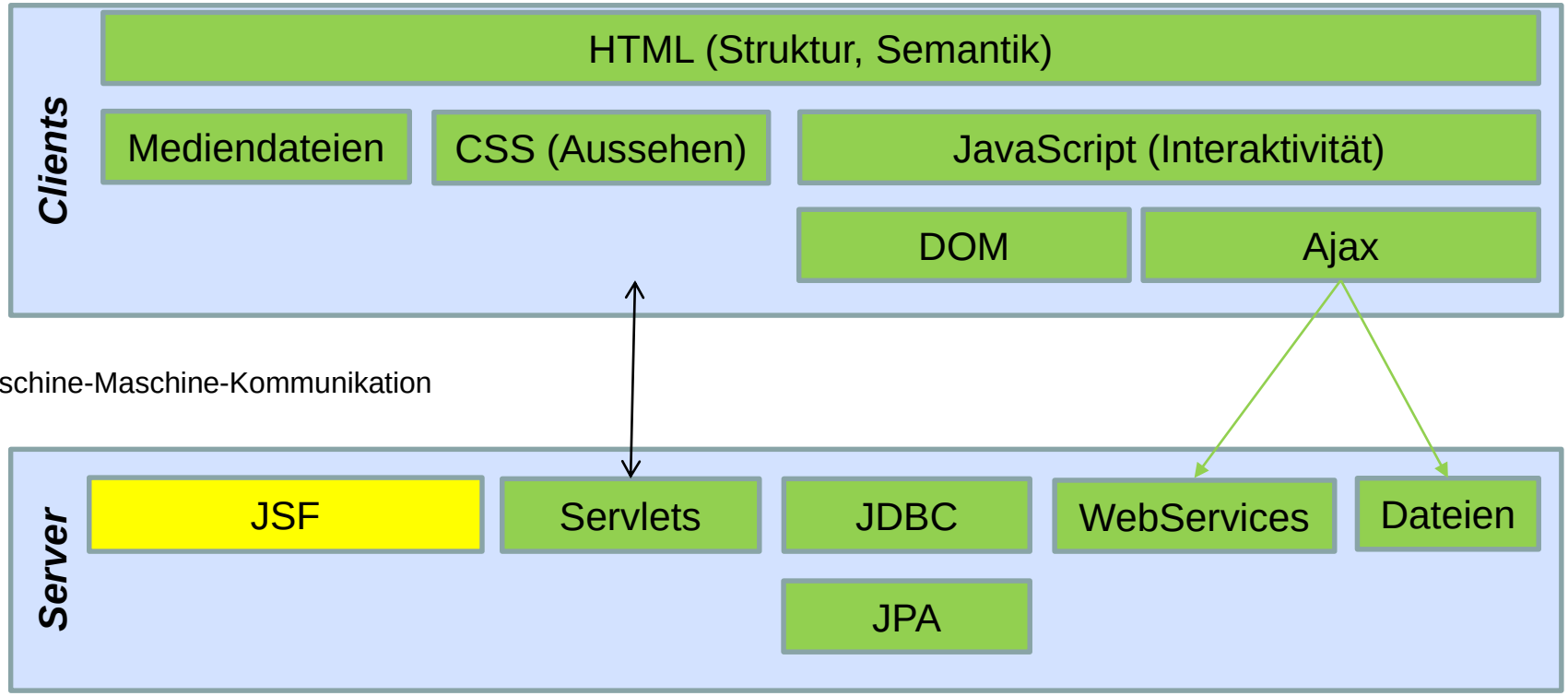
Studiengang Informatik

Java Server Faces (JSF)

1. **Kontext und Motivation**
2. Bibliotheken und Frameworks
3. jQuery
4. Bootstrap
5. Knockout.js
6. Angular
7. Darüber hinaus
8. Projekt

Problemfelder

Mensch-Maschine-Kommunikation



Problemstellung

Problemstellung: Sowohl auf Frontend-, als auch auf Backend-Seite können verschiedene Bibliotheken und Frameworks zum Einsatz kommen.

Probleme:

- Überdeklaration
  - Bibliotheken und Frameworks müssen zusammen passen und dürfen nicht unterschiedliche Funktionen unter dem selben Namen bereitstellen.
    - Kann durch Namespaces gelöst werden.
- Schnittstelle
  - Die Kommunikation zwischen Backend und Frontend muss implementiert werden, da meist unterschiedliche Technologien eingesetzt werden.

Lösung:

- Full-Stack-Framework

Java Server Faces

1. Kontext und Motivation
2. **Eigenschaften und Verwendung**
3. Bearbeitungsmodell
4. ManagedBeans
5. JSF-EL
6. Validierung und Konvertierung
7. Darüber hinaus

Eigenschaften

Definition: **JavaServerFaces ist ein Full-Stack-Framework**

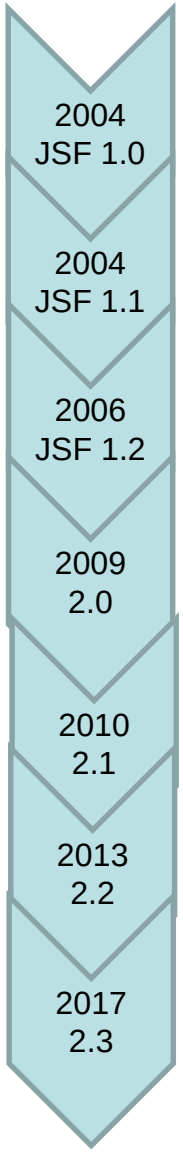
Eigenschaften:

- realisiert mit Servlet-API
  - basiert auf dem Request-Response-Modell des HTTP-Protokolls
- Vollständige Umsetzung des MVC-Konzepts
  - Zustände spielen eine wichtige Rolle
  - JSF implementiert die Zustandsverwaltung
  - Implementiert ein Event-Modell
- Requests werden in verschiedene Schritte aufgeteilt
  - Definiert durch ein Bearbeitungsmodell
- Verwendet deklarative Implementierung
  - Annotationen für die Festlegung wichtiger Eigenschaften
- Verwendet eine eigene Expression-Language
  - Basierend auf HTML
- Verwendet Bean-Konzepte
- Liefert fertig verwendbare Komponenten
- Um Komponentenbibliotheken (und eigene) erweiterbar

Eigenschaften und Einbindung III

Geschichte von JSF:

1.0	Spezifikation in JSR 127
1.1	Veröffentlichung der Referenzimplementierung Mojarra
1.2	Zusammenführung mit JavaEE 5, API Verbesserung
2.0	Vereinfachte Benutzung (Annotationen) JavaEE 6 kompatibel, Erweiterte Funktionen und bessere Performance
2.1	Kleinere Änderungen und Fehlerbehebungen
2.2	Neue Konzepte: stateless views, page flow,…
2.3	Push-Kommunikation, Suchausdrücke,…



Java Server Faces

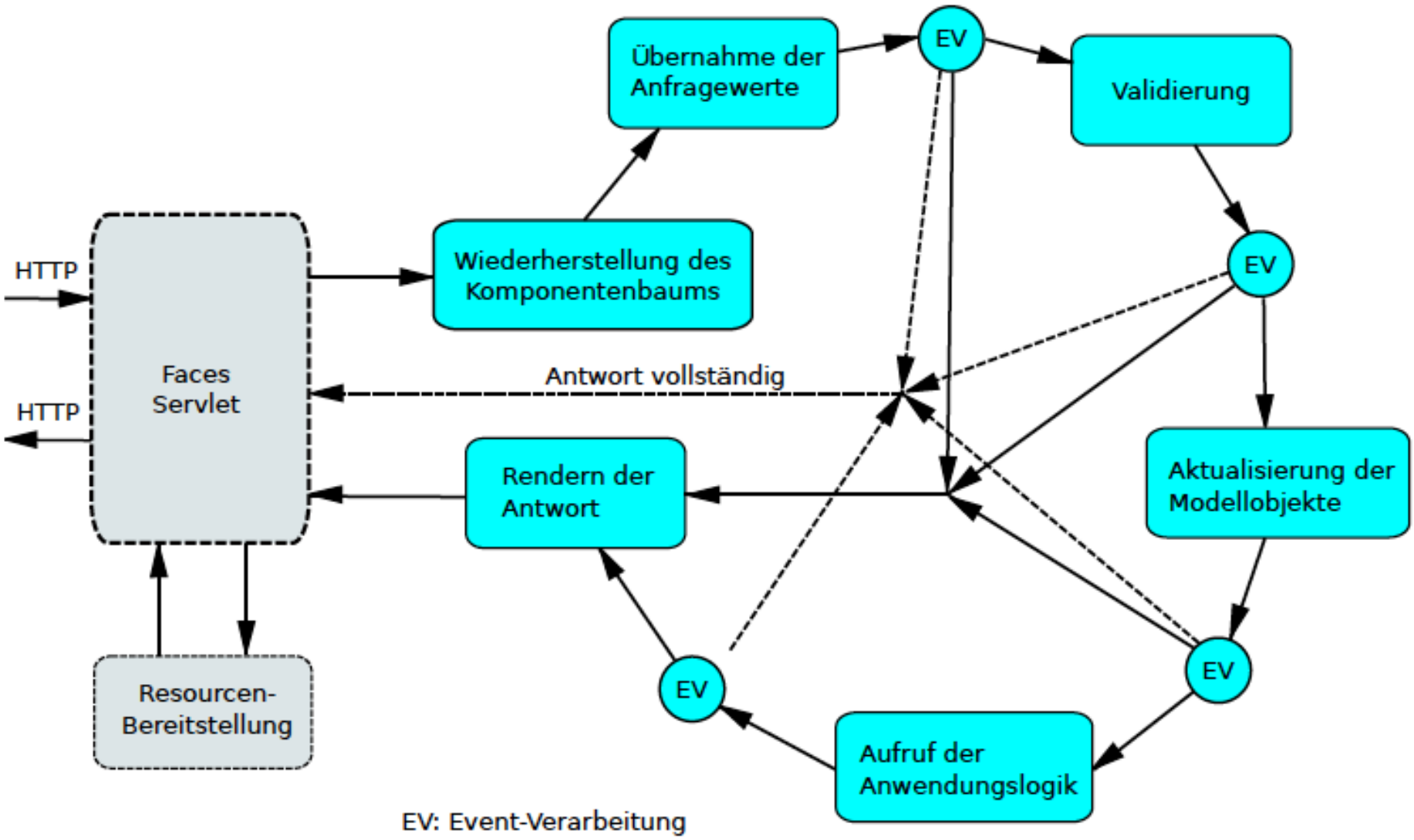
1. Kontext und Motivation
2. Eigenschaften und Verwendung
3. **Bearbeitungsmodell**
4. ManagedBeans
5. JSF-EL
6. Validierung und Konvertierung
7. Darüber hinaus

Bearbeitungsmodell

Definition: **Das Bearbeitungsmodell legt den Ablauf der Bearbeitung einer Anfrage an eine JSF-Komponente fest, welche eine JSF-Antwort generiert.**

JSF-Komponenten können auch nicht-JSF Antworten liefern (z.B. PDFs). Solche Aufrufe werden nicht durch das Bearbeitungsmodell abgedeckt.

Das Bearbeitungsmodell einer Anfrage



Wiederherstellung des Komponentenbaums

- ***die View*** ist eine nicht sichtbare Komponente und ***Wurzel des Komponentenbaumes einer Seite***
- sie enthält damit alle Komponenten der Seite
- Komponenten werden zwischen der Antwort und einer erneuten Anfrage gespeichert:
  - Wiederholter Besuch (Alternative 1): Wiederherstellung des gespeicherten Komponentenbaums
  - Erstaufruf (Alternative 2): Erstellung eines neuen Komponentenbaums
- der View wird eine View-Id zugeordnet, die aus der Anfrage-URI besteht

  Zum Beispiel:
  - view: **`/comedians`**
  - view-id: **`/comedian.jsf`**
- View-Id wird in Session gespeichert

- Komponentenbaum wird dann im ***FacesContext*** gespeichert (http://java.sun.com/javaee/javaserverfaces/1.1_01/docs/api/javax/faces/context/FacesContext.html)

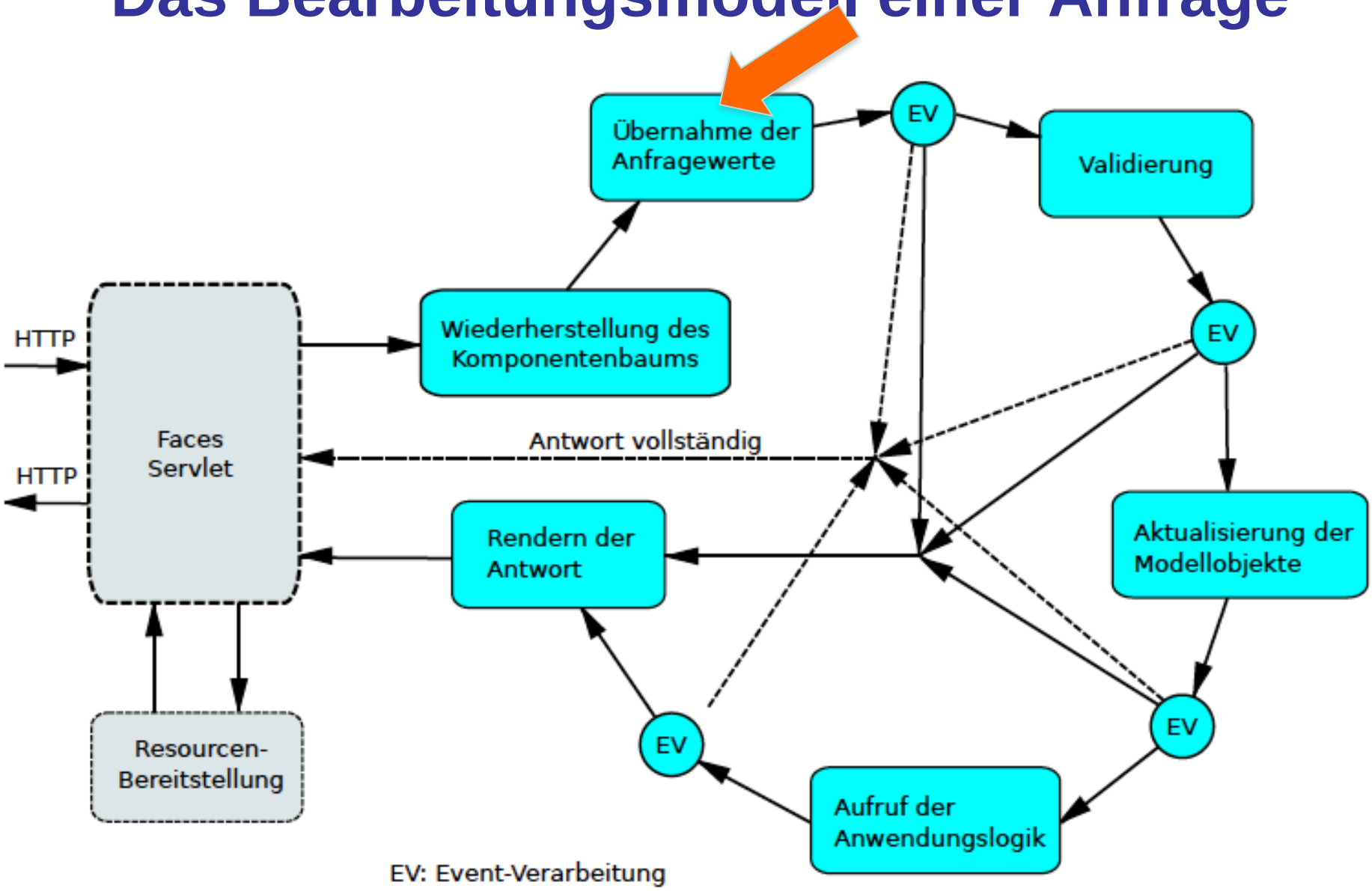
Wiederherstellung des Komponetenbaums II

- ***FacesContext*** ist eine Klasse und ***enthält alle Informationen im Zusammenhang der Bearbeitung einer JSF-Anfrage.***

- Wiederherstellung umfasst auch Wiederherstellen aller verbundenen Event-Listener, Validierer, Konvertierer und Managed Beans

- Bei Erstaufruf wird nach phase 1 „Wiederherstellung des Komponentenbaumes“ gleich zum letzten Schritt „Rendern“ gesprungen.
- Ansonsten z.B. weiter mit Schritt 2 „Übernahme der Anfragewerte“

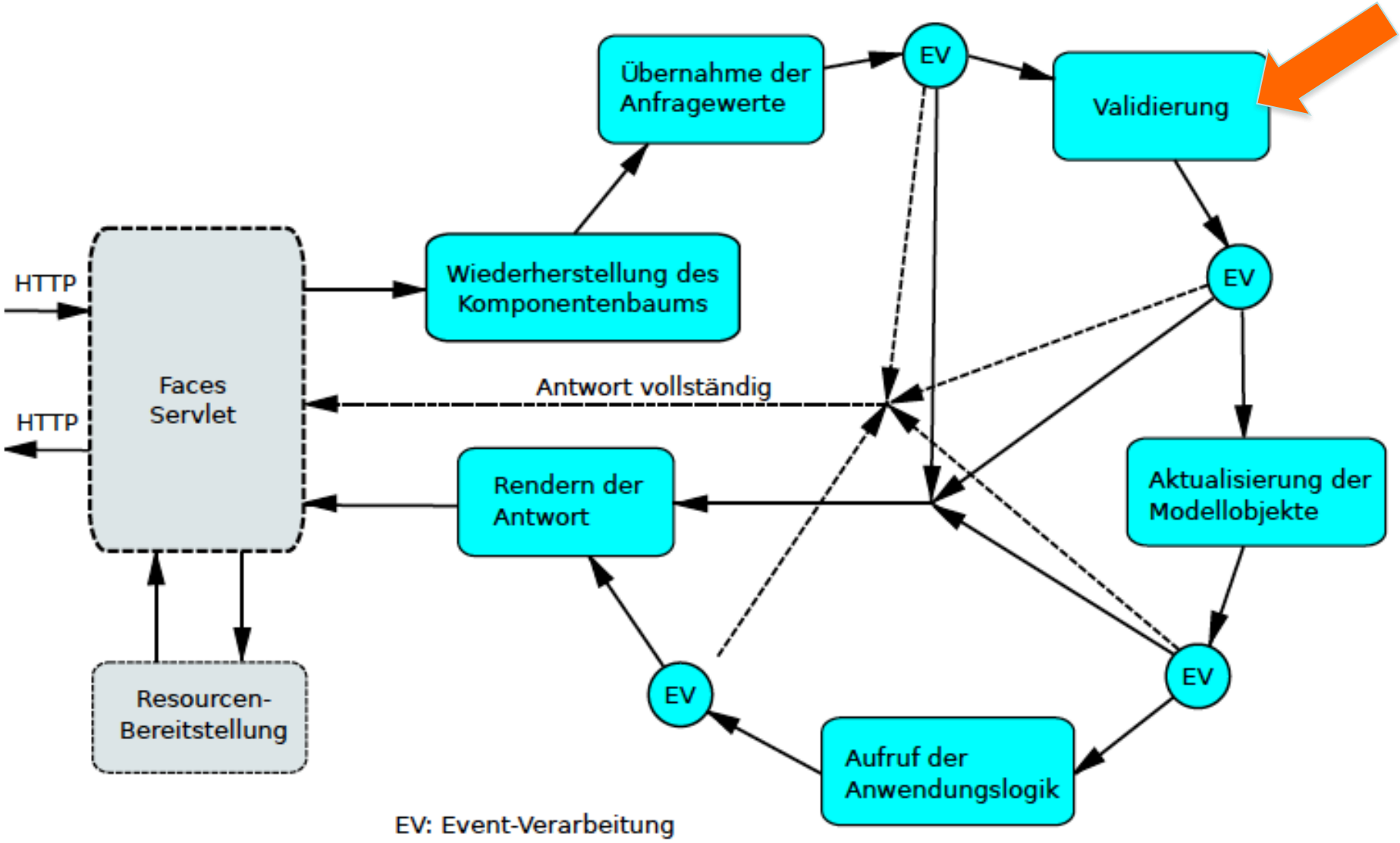
Das Bearbeitungsmodell einer Anfrage



Übernahme der Anfragewerte

- einige UI-Komponenten lassen Benutzereingaben zu
- das zugrunde liegende HTML-Formular schickt diese als POST-Request per HTTP an den Server
- dieser POST-String muss geparst werden und die Parameter mit ihren jeweiligen Werten müssen herausgefiltert werden
- diese ***Werte werden*** dann ***vorläufig den Komponenten zugewiesen***
- vorläufig, weil ***Konvertierung und Validierung noch erfolgen*** und ggf. auf einen Fehler laufen können

Konvertierung und Validierung



Konvertierung und Validierung

- Ausgangspunkt: alle Anfrageparameter für UI-Komponenten verfügbar…

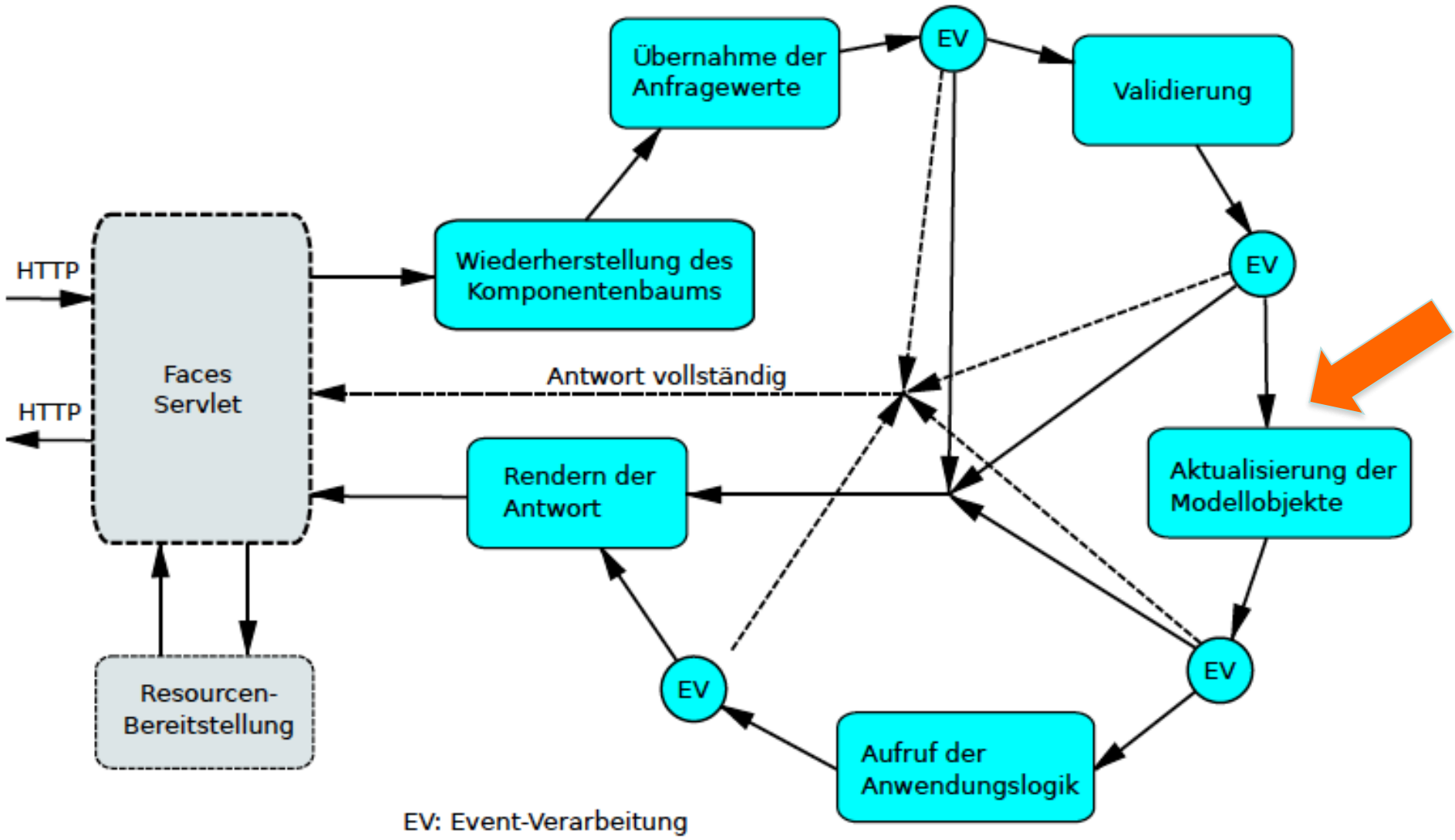
1) **Zuerst Konvertierung:**
   - alle Post-Parameter sind Strings, daher in entsprechenden Typ konvertieren (automatisch für **`Character, Boolean, Byte, Integer,Short, Long, Float, Double`** und deren primitive Versionen sowie **`BigDecimal und BigInteger`**)
2) **Validierer überprüfen dann Eingaben**
   - eingebaut: **`required="true"`** Zahlen von bis, String-Länge,..
3) **Alle Konvertierungen und Validierungen erfolgreich? -> dann der Komponente Wert zuweisen**
   - Wertänderung seit letztem Request?-> **`ValueChangeEvent`** werfen und an registrierte Listener weitergeben
   - Listener können: Rendering-Phase einleiten, Antwort erzeugen oder Aktualisierung der Modellobjekte vornehmen.

Validierungen in einem Beispiel:

```
<h:inputText id="vorname"  required="true" value="#
        {comedianHandler.aktuellerComedian.vorname}"/>
<f:validateLength minimum="3" maximum="25"/>
```

`requiered = "true"` //Überprüfung, ob das Eingabefeld einen Wert enthält
`minimum="3" maximum="25"` // Überprüfung, ob der Wert zwischen 3 und 25 liegt

Aktualisierung der Modellobjekte



Aktualisierung der Modellobjekte

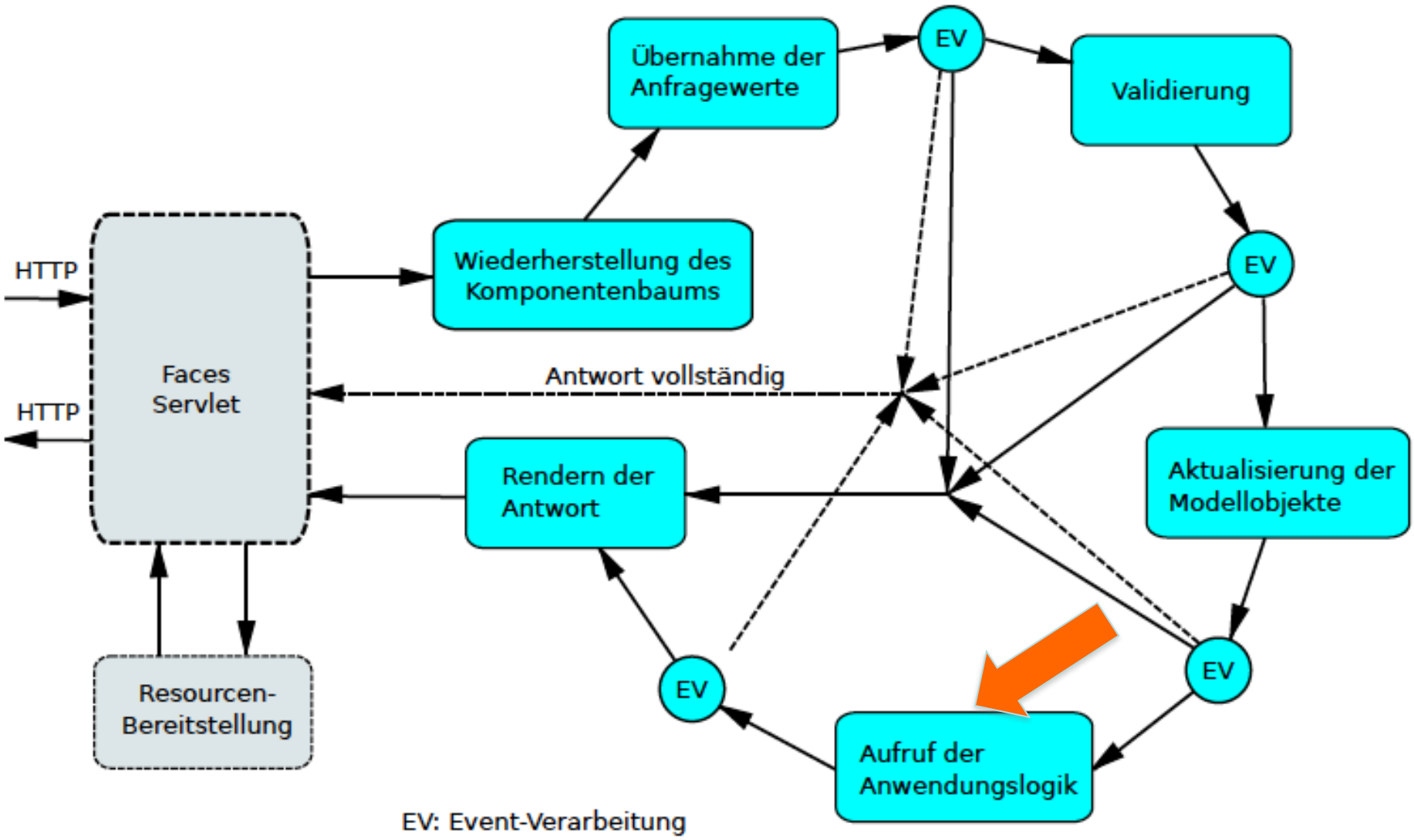
- Zu Beginn der Phase ***alle Vorgänge in den UI-Komponenten abgeschlossen*** -> Übergang zum Modell
- ***Daten valide und vom richtigen Typ? ->*** werden den Modellobjekten zugewiesen
- Gleicher Phasenabschluß:
  - Events werfen,
  - Listener informieren
  - Ggf. Bearbeitung beenden

Aktualisierung der Modellobjekte in unserem Beispiel:

```
<h:inputText id="vorname"  required="true"
value="#{comedianHandler.aktuellerComedian.vorname}">
```

- JSF-Implementierung sucht nach einer Managed Bean mit dem Schlüssel `comedianHandler`
- Das Property vorname des Ergebnisobjekts wird auf Wert aus der UI-Komponente gesetzt (z.B. setter –Aufruf mit Wert :“Mario“).

Aufruf der Anwendungslogik



Aufruf der Anwendungslogik I

- bis jetzt alles automatisch von JSF-Implementierung ausgeführt, ohne Anwendungslogik:
  - decodieren
  - konvertieren
  - validieren
  - Wertzuweisung (Setter-Aufruf)

- Anwendungslogik wird durch Listener aufgerufen, die auf Action-Events (Schaltflächen für ActionEvents oder Hyperlinks) registriert sind.

Aufruf der Anwendungslogik II

Zwei Arten von Listenern:

1.„**Richtige“ Listener** –

- werden über das Attribut `actionListener` registriert

```
z.B. <h:commandButton id="feld-0" image="#{tttHandler.image[0]}
actionListener="#{tttHandler.zug}"
```

// 9 Schaltflächen bei TicTacToe

```
public void zug(ActionEvent ae)
```

// Listenermethode `zug` muss `ActionEvent` als Parameter haben.

2.Default-Action-Listener

–automatisch für Steuerkomponenten registrierte action-listener
–Action Attribut in der Komponente
–Parameterlose Action-methode im Handlerobjekt liefert String oder Object zurück

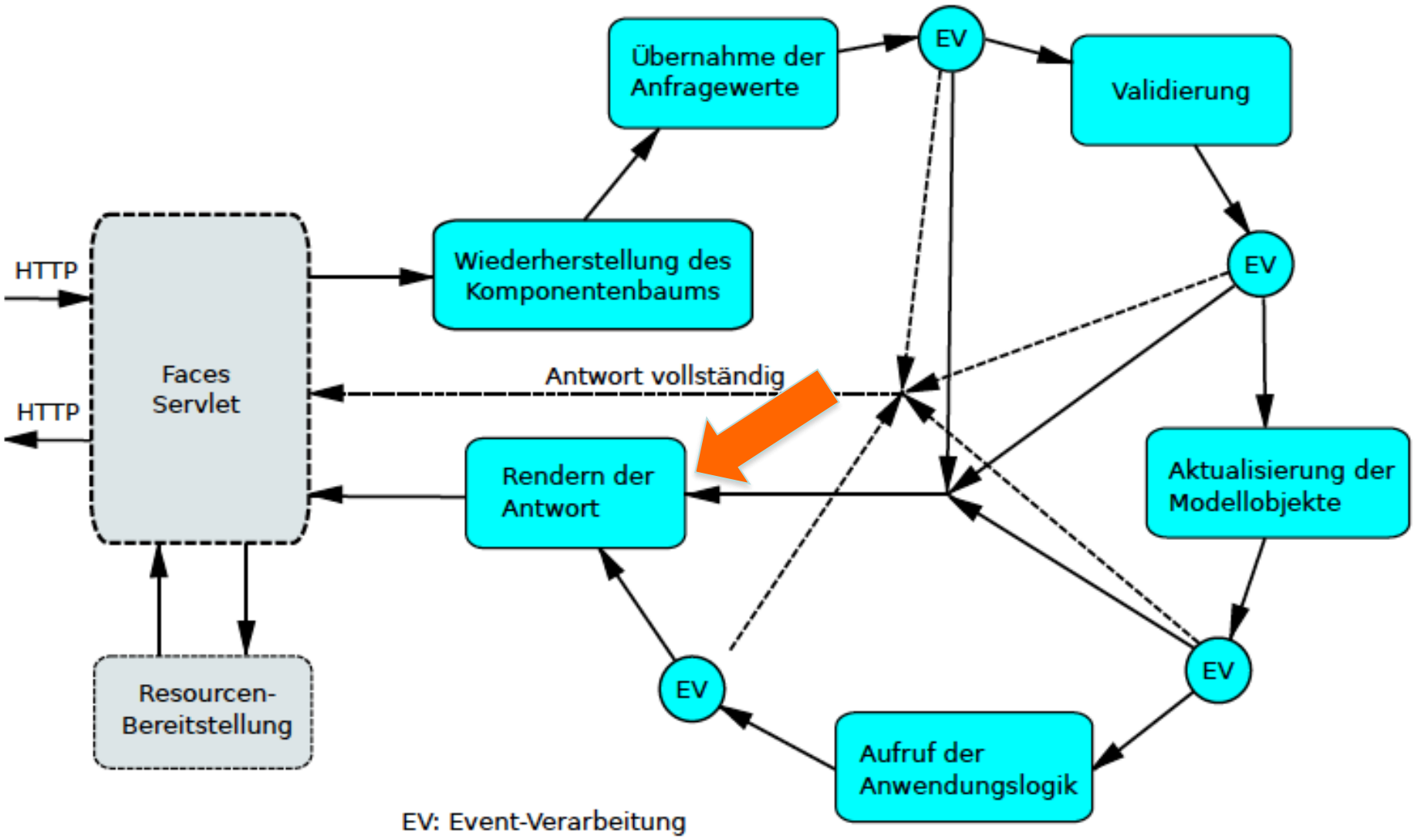
```
z.B. <h:commandButton action =
"#{comedianHandler.speichern}"    value="Speichern"/>
```

hierbei Action-Methode login:

```
public String login() {
```

//parameterlos mit String oder Object als Rückgabe

Rendern der Antwort



Rendern der Antwort

- ***Zielsprache zum Rendern*** durch Spezifikation nicht festgelegt:
  - jede JSF-konforme Implementierung muss mindestens JSP/HTML unterstützen
  - seit JSF2.0 Faclets/XHTML

- zuletzt ***Abspeichern des Komponentenbaumes***
  - Seit JSF 2.0 „Partial State Saving“ möglich
  - Festlegung dazu in faces-config.xml

Java Server Faces

1. Kontext und Motivation
2. Eigenschaften und Verwendung
3. Bearbeitungsmodell
4. **ManagedBeans**
5. JSF-EL
6. Validierung und Konvertierung
7. Darüber hinaus

Managed Beans deklarieren

- Durch die Annotation `@ManagedBean` wird eine Java-Bean-Klasse zu einer ***Managed Bean***
- als Name wird der Klassenname verwendet.
  - im Beispiel für Klasse `ComedianHandler` ist der Bean-Name: `comedianHandler`

- die Annotation `@SessionScoped` deklariert als Scope der Managed Bean die Session

Managed Beans

Aufgaben:

•Haupteinsatzgebiet ist Reaktion auf Benutzereingaben

•sammeln Daten von UI-Komponenten

•implementieren Event-Listener

•können Referenzen auf UI-Komponenten halten

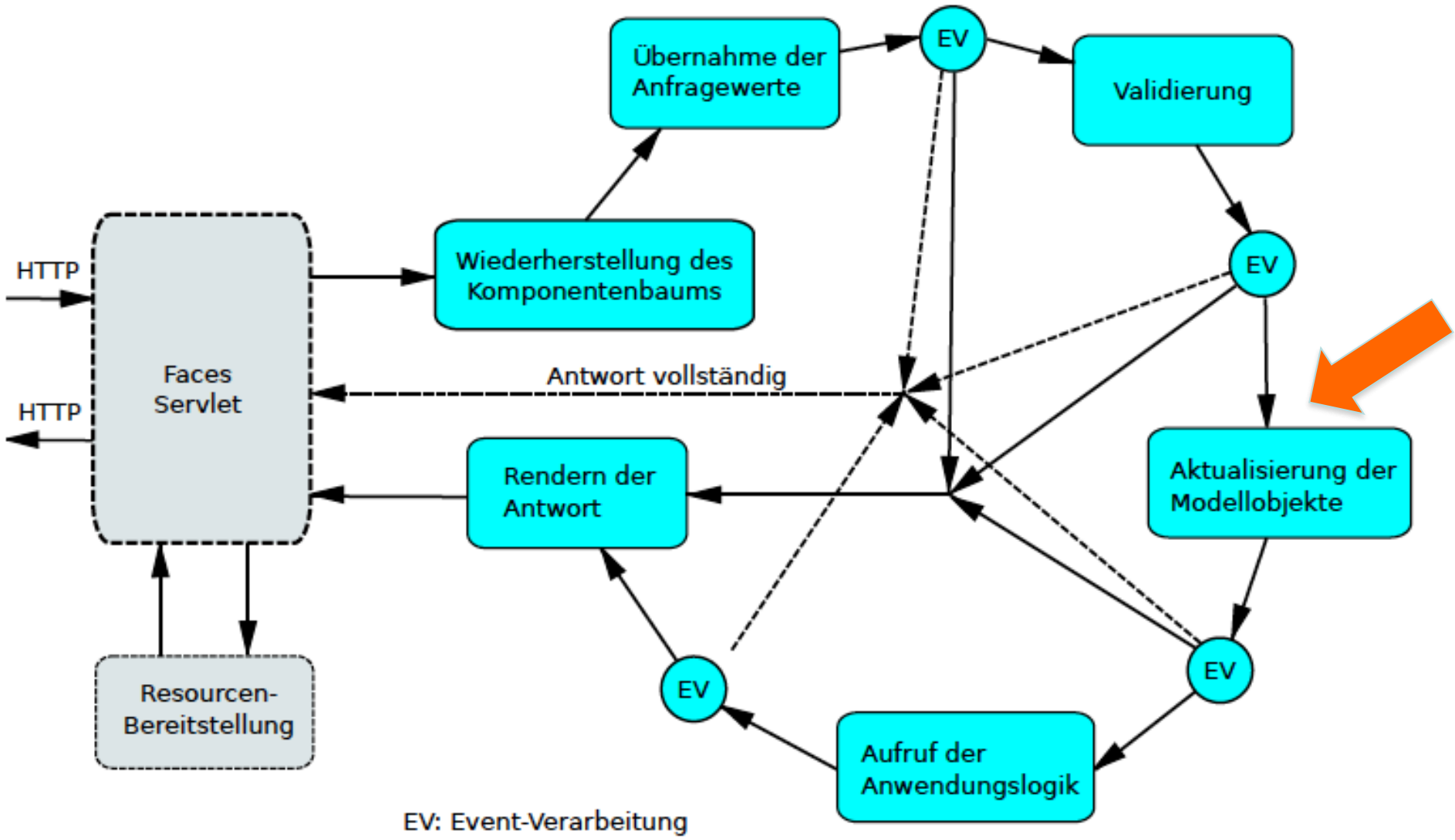
•Verwendung in der EL (Zugriff auf Anwendungsdaten für Seitenentwickler ohne Java-Kenntnisse, saubere Architektur)

•bereits in Comedian-Anwendung verwendet:
ComedianHandler

Automatische Verwaltung durch JSF-Container

- daher auch Managed-Bean genannt
- Erzeugung bei Bedarf
- festgelegte Lebensdauer
- Konfiguration möglich mit:
  - Konfigurationsdatei <managed-bean>-Element
  - Annotation

Aktualisierung der Modellobjekte



Managed Beans: Annotationen

- Seit JSF 2.0 Annotationen generell in JSF spezifiziert
- Alternative zur Beschreibung innerhalb der xml-Konfigurationsdatei **`faces-config.xml`**
- Bei Widersprüchen zwischen Konfigurationsdatei und Annotation gilt Konfigurationsdatei
- Nachfolgend Annotationen für Manged Beans
  - MB‘s deklarieren
  - Properties injizieren
  - Scope definieren

Managed Beans: Annotationen I/II

```
@ManagedBean(name = "mbAnnotationHandler")
@RequestScoped
//@ViewScoped
//@SessionScoped
//@ApplicationScoped
public class MBAnnotationHandler {

        @ManagedProperty(value = "Annotierte Managed-Beans")
        private String title;

        @ManagedProperty(value = "3.1415")
        private Double pi;
```

MBAnnotationHandler.java

- Klasse MKAnnotationHandler ist annotiert mit `@ManagedBean` und `@RequestScoped`
- Klassenname wird ***nach Annotation mit* @ManagedBean *automatisch als Bean-Name*** übernommen: Klassenname mit Kleinschreibung des ersten Buchstaben (wäre gewesen: **`mBAnnotationHandle`**r)
- MB-Name kann auch über Attribut Name festgelegt werden (siehe oberes Bsp.)

Managed Beans: Annotationen II/II

```
<h:panelGrid columns="1" rowClasses="odd,even">
    <f:facet name="header">#{mbAnnotationHandler.title}</f:facet>
    <h:panelGroup rendered=
        "#{not empty  requestScope.mbAnnotationHandler}">
        Bean ist im Request-Scope
    </h:panelGroup>
    <h:panelGroup rendered=
        "#{not empty viewScope.mbAnnotationHandler}">
        Bean ist im View-Scope
    </h:panelGroup>
    <h:panelGroup rendered=
        "#{not empty sessionScope.mbAnnotationHandler}">
        Bean ist im Session-Scope
    </h:panelGroup>
    <h:panelGroup rendered=
        "#{not empty applicationScope.mbAnnotationHandler}">
        Bean ist im Application-Scope
    </h:panelGroup>
</h:panelGrid>
```

mb-annotation.xml

Java Server Faces

1. Kontext und Motivation
2. Eigenschaften und Verwendung
3. Bearbeitungsmodell
4. ManagedBeans
5. **JSF-EL**
6. Validierung und Konvertierung
7. Darüber hinaus

JSF – Expression Language (kurz: EL)

- Stammt ursprünglich von JSTL und JSP
  - Einfacher Zugriff auf Anwendungsdaten für JSP-Entwickler ohne Java-Zugriff od. –Kennnisse
- In JSF2.0 gibt es eine ***Unified Expression Language***
  - Für JSF 1.2 als Teildokument von JSP-Spezifikation 2.1 definiert
  - In JSF 2.0 erweitert

JSF – Expression Language (kurz: EL)

- EL-Ausdrücke sind Strings
- werden zweimal ausgewertet :
  - Wertebindungen lesend und schreibend ausgeführt

- ***Punkt-Notation*** über Objekt-Properties ähnlich wie in JavaScript

- EL-Ausdrücke können sein:
  - Wertebindungen
  - Methodenbindungen
  - Arithmetische und logische Ausdrücke
  - Seit JSF 2.0 auch Methodenparameter (später)

Syntax: `#{ EL-Ausdruck }`

JSF – Expression Language (kurz: EL)

- ***Wertebindungen*** (engl. value binding) an Bean-Properties:
  - Wert einer UI-Komponente an Property
  - UI-Komponente an eine Bean-Property
  - Initialisierung der UI-Komponente einer Property

- ***Methodenbindungen*** (engl. method binding)
  - Bindet Wert einer UI-Komponente an eine Bean-Methode (z.B. Verwendung bei Event-Handler und Validierungsmethoden)

Beispiele für Wertbindungen

- **Lesen** der Wertebindung (getter) in der Phase ***Rendern der Antwort***
- **Schreiben** der Wertebindung (setter) in der Phase ***Aktualisierung der Modellkomponente***

- Alles Ausgabekomponenten im Beispiel mit lesbaren Wertebindungen

- z.B. für `#{elHandler.getName}` wird zum getter **`getName()`** evaluiert also: `name = “Übungen mit der Expression-Language“`

Beispiele für Wertbindungen

Codeausschnitt aus jsf - Seitenbeispiel:

```
1 <h:outputText value="Zugriff auf Bean-Properties:"
2 style="font-weight: bold;" />
3 <h:outputText value="#{elHandler.name}" />
4 <h:outputText value="#{elHandler['name']}" />
5 <h:outputText value="Dies sind tolle #{elHandler.name}" />
6 <h:outputText value="#{elHandler.array[0]}" />
7 <h:outputText value="#{elHandler.map['zwei']}" />
8 <h:outputText value="#{elHandler.map[elHandler.array[2]]}" />
```

Codeauschnitt aus el.xhtml aus Eclipse-Projekt jsf-im-detail auf ILIAS unter Codeausschnitte

Die zugehörige Managed Bean **`elHandler`** ist eine Instanz der Klasse **`ELHandler`** mit Codeausschnitt auf der nächsten Seite…

Beispiele für Wertbindungen

Codeausschnitt aus ELHandler.java für bean aus Klasse ELHandler:

```
public class ELHandler {

private String name = "Übungen mit der Expression-Language";
private Integer jahr = new Integer(new java.text.SimpleDateFormat
                       ("yyyy").format(new java.util.Date()));
private String[] array = new String[]{ "eins", "zwei", "drei" };
private Map<String, String> map = new HashMap<String, String>();

public ELHandler() {
super();
map.put("eins", "Erster Map-Eintrag");
map.put("zwei", "Zweiter Map-Eintrag");
map.put("drei", "Dritter Map-Eintrag");
}
// ab hier nur einfache Getter und Setter
```

Codeauschnitt aus ELHandler.java

Beispiele für Wertbindungen

```
1 <h:outputText value="Zugriff auf Bean-Properties:"
2 style="font-weight: bold;" />
3 <h:outputText value="#{elHandler.name}" />
4 <h:outputText value="#{elHandler['name']}" />
5 <h:outputText value="Dies sind tolle #{elHandler.name}" />
6 <h:outputText value="#{elHandler.array[0]}" />
7 <h:outputText value="#{elHandler.map['zwei']}" />
8 <h:outputText value="#{elHandler.map[elHandler.array[2]]}" />
```

Zugriff auf Bean-Properties
Übungen mit der Expression-Language
Übungen mit der Expression-Language
Dies sind tolle Übungen mit der Expression-Language
eins
Zweiter Map-Eintrag
Dritter Map-Eintrag

Implizite Objekte I/III

JSF ***definiert über Variablen implizite Objekte*** zur Verwendung in der Expression Language. Sie entstammen der zugrunde liegenden *Servlet- und JSF-Implementierung.*

Vordefinierte Variablen

Variablenname	Beschreibung
header	Eine `Map` von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist *ein* String.
headerValues	Eine `Map` von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist ein Array von Strings.
cookie	Eine `Map` von Cookies (`javax.servlet.http.Cookie`). Schlüssel ist der Cookie-Name.
initParam	Eine `Map` von Initialisierungsparametern der Anwendung. Diese werden im Deployment-Deskriptor definiert.

Implizite Objekte II/III

Variablenname	Beschreibung
param	Eine `Map` von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist *ein* String.
paramValues	Eine `Map` von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist ein Array von Strings.
facesContext	Die `FacesContext`-Instanz der aktuellen Anfrage.
component	Im Augenblick bearbeitete Komponente.
cc	Im Augenblick bearbeitete zusammengesetzte Komponente.
resource	Map von Ressourcen.
flash	Map von temporären Objekten für nächste View.

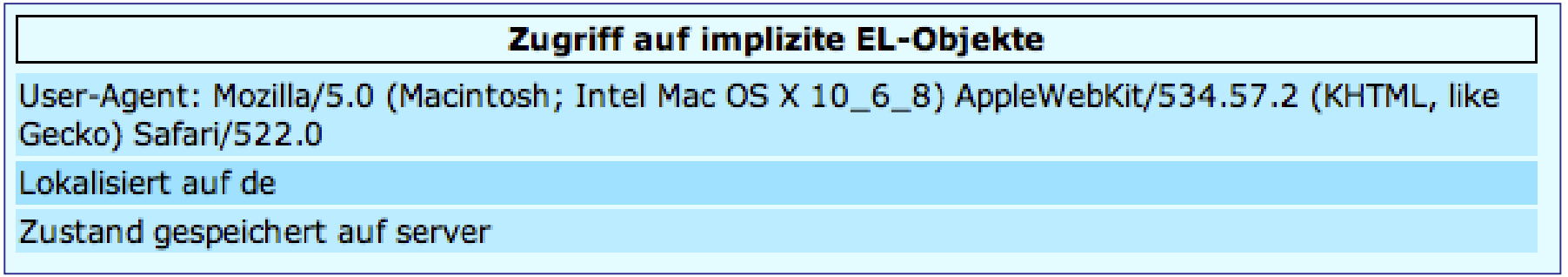
Implizite Objekte III/III

Variablenname	Beschreibung
view	Die aktuelle View.
viewScope	Eine `Map` von Variablen mit View-Scope.
request	Das Request-Objekt.
requestScope	Eine `Map` von Variablen mit Request-Scope.
session	Das Session-Objekt.
sessionScope	Eine `Map` von Variablen mit Session-Scope.
application	Das Application-Objekt.
applicationScope	Eine `Map` von Variablen mit Application-Scope.

Implizite Objekte der EL - Zugriffsbeispiele

```
<h:panelGrid columns="1" rowClasses="odd,even">
    <f:facet name="header">
        Zugriff auf implizite EL-Objekte
    </f:facet>
    <h:outputText value="User-Agent: #{header['User-Agent']}" />
    <h:outputText value="Lokalisiert auf #{view.locale}" />
    <h:outputText value="Zustand gespeichert auf #{initParam
                    ['javax.faces.STATE_SAVING_METHOD']}" />
</h:panelGrid>
```

Codeausschnitt aus el.xhtml aus Eclipse-Projekt jsf-in-detail



EL: Vergleiche, arithmetische und logische Ausdrücke

JSF-EL umfasst ***vollständige Arithmetik und Logik***, so dass in der Regel ***kein Rückgriff auf Java*** notwendig ist.

Operatoren, Teil 1:

Op	Alt.	Beschreibung
.		Zugriff auf eine Property, Methode oder einen Map-Eintrag
[]		Zugriff auf ein Array- oder Listen-Element oder einen Map-Eintrag
()		Klammerung für Teilausdrücke
?:		Bedingter Ausdruck: <expr> ? <true-value> : <false-value>
+		Addition
-		Subtraktion oder negative Zahl
*		Multiplikation

EL: Vergleiche, arithmetische und logische Ausdrücke

Operatoren, Teil 2:

/	div	Division
%	mod	Modulo
==	eq	gleich (equals()-Methode)
!=	ne	ungleich
<	lt	kleiner
>	gt	größer
<=	le	kleiner-gleich
>=	ge	größer-gleich
&&	and	logisches UND
||	or	logisches ODER
!	not	logische Negation
empty		Test auf null, einen leeren String, oder Test auf Array, Map oder Collection ohne Elemente

EL: arithmetische Ausdrücke

Codeausschnitt aus jsf - Seitenbeispiel:

```
<f:facet name= "header">
    Arithmetische Ausdrücke
</f:facet>
<h:outputText value="#{17 + 4}" />
<h:outputText value="Das übernächste Jahr ist #{elHandler.jahr +2}"/>
<h:outputText value="#{elHandler.jahr} ist
 #{((elHandler.jahr % 4) == 0 ? 'ein' : 'kein')} Schaltjahr" />
```

Codeauschnitt aus el.xhtml aus Eclipse-Projket jsf-in-detail

Arithmetische Ausdrücke
21
Das übernächste Jahr ist 2014
2012 ist ein Schaltjahr

EL: Vergleiche (akt. Jahr 2012)

```
<f:facet name="header">
    Vergleichsausdrücke und logische Ausdrücke
</f:facet>
<h:outputText value="#{'eins' == elHandler.array[0]}" />
<h:outputText value="#{2009 == elHandler.jahr}" />
<h:outputText value="#{'2009' == elHandler.jahr}" />
<h:outputText value="#{2008 == elHandler.jahr}" />
<h:outputText value="#{elHandler.jahr > 2000}" />
<h:outputText value="
     #{elHandler.jahr > 2000 &amp;&amp; (elHandler.jahr %4 != 0)}" />
```

Codeauschnitt aus el.xhtml aus Eclipse-Projket jsf-in-detail

Vergleichsausdrücke und logische Ausdrücke
true
false
false
false
true
false

EL: Methodenaufrufe und –parameter I/II

Ausschnitt aus ELHandler.java

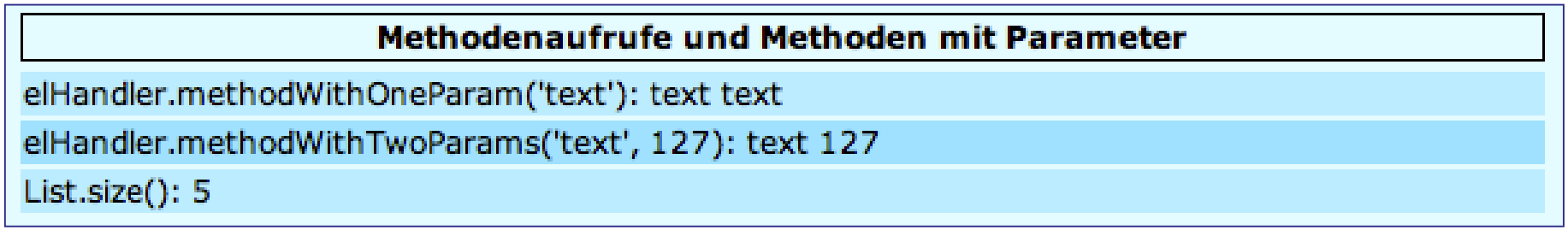
// Ab JSP 2.1 (JSR 245) Maintenance Review 2 sind Methodenparameter erlaubt

```
public String methodWithOneParam(String param) {
                return param + " " + param;
        }

public String methodWithTwoParams(String param1, int param2) {
                return param1 + " " + param2;
        }
public List<Integer> getList() {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1); list.add(2); list.add(3); list.add(4); list.add(5);
    return list;
        }
```

EL: Methodenaufrufe und –parameter II/II

Ausschnitt aus el.xhtml

```
<f:facet name="header">
    Methodenaufrufe und Methoden mit Parameter
</f:facet>
<h:outputText value="elHandler.methodWithOneParam('text'):
                #{elHandler.methodWithOneParam('text')}" />
<h:outputText value="elHandler.methodWithTwoParams('text', 127):
                #{elHandler.methodWithTwoParams('text', 127)}" />
<h:outputText value="List.size(): #{elHandler.list.size()}" />
```



Analog zu Werteausdrücken können auch Methodenausdrücke parametrisiert werden.
z.B. diese Syntax :

```
<h:commandButton action="#{bean.doAction(arg1,arg2)}" />
```

Verwendung der EL in Java I/II

- Erzeugung eines Werteausdrucks in Java auf Grundlage des :
  - **`FacesContext-`** und des **`Application-`**Objects
  - Methode **`getExpressionFactory()`** erzeugt **`ExpressionFactory-`** Instanz vom **`Application`**-Object
  - In Java kann dann bezüglich des aktuellen Kontextes ein Werteausdruck erzeugt und ausgewertet werden.

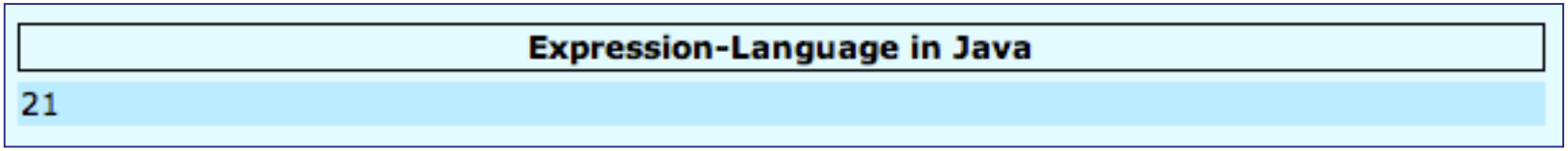
Ausschnitt aus ELHandler.java:

```
public Integer getTestAusdruck() {
    FacesContext faces = FacesContext.getCurrentInstance();
    Application app = faces.getApplication();
    ExpressionFactory expressionFactory =
                                        app.getExpressionFactory();
    ELContext el = faces.getELContext();
    Integer val = (Integer) expressionFactory.
    createValueExpression(el, "#{17 + 4}",
Integer.class).getValue(el);
```

Verwendung der EL in Java II/II

Ausschnitt aus el.xhtml:

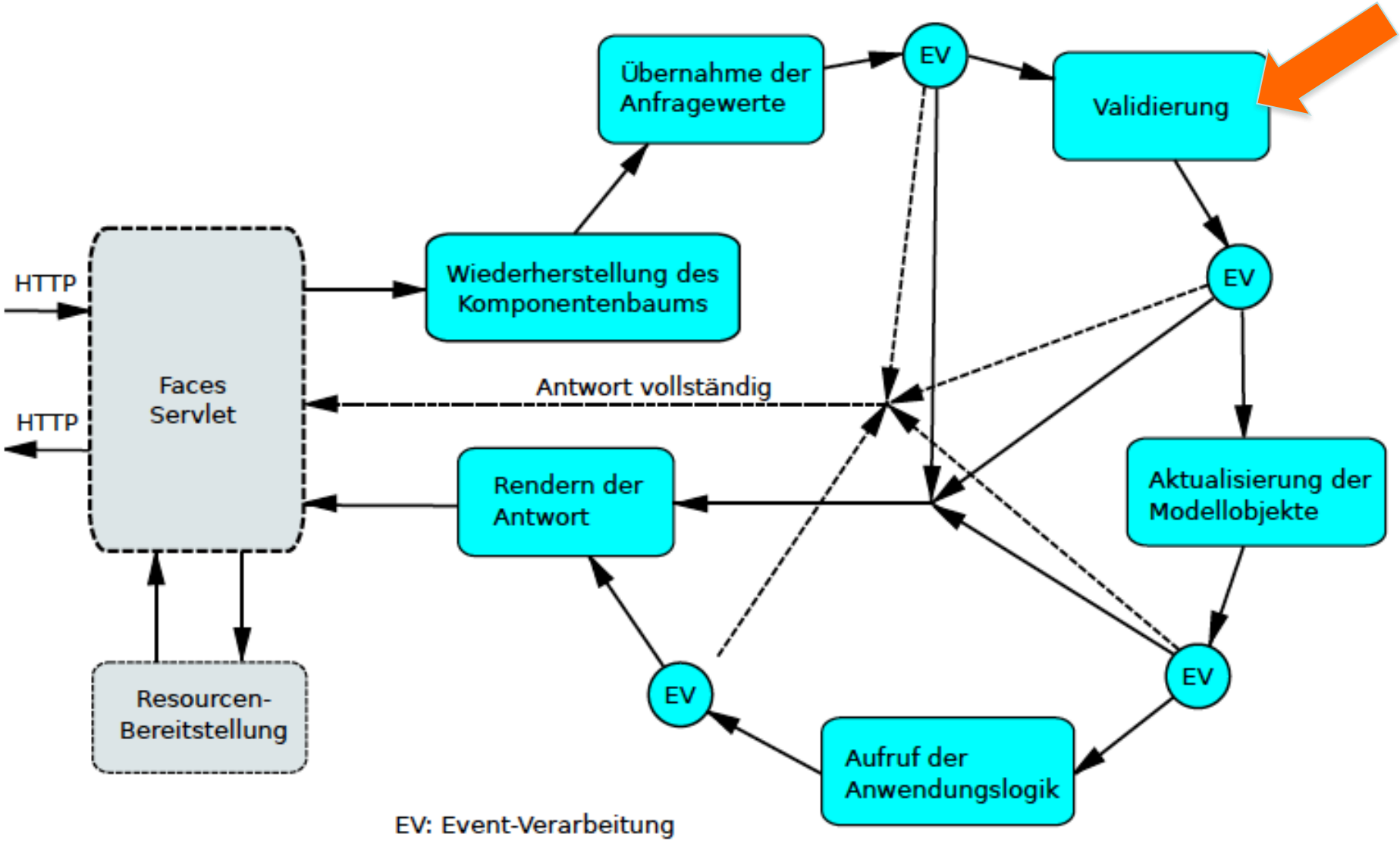
```
<f:facet name="header">
    Expression-Language in Java
</f:facet>
<h:outputText value="#{elHandler.testAusdruck}" />
```



Java Server Faces

1. Kontext und Motivation
2. Eigenschaften und Verwendung
3. Bearbeitungsmodell
4. ManagedBeans
5. JSF-EL
6. **Validierung und Konvertierung**
7. Darüber hinaus

Konvertierung und Validierung



Validierung und Konvertierung

Validierung:

- ***Syntaktische Validierung:***
  - ob überhaupt eine Eingabe vorhanden ist
  - Formatüberprüfungen z.B. bei Datum TT.MM.JJJJ
- ***Semantische Validierung:***
  - Überprüfungen ob Zuordnungen der Daten stimmen (z.B. Kennwort zur Kundennummer, oder ob ein Datum in der Zukunft liegt)

- ***JSF-Komponenten*** können viele syntaktische Überprüfungen selbst vornehmen
- Semantische Überprüfungen können durch ***vordefinierte Validierer*** erzielt werden, die an die Komponenten geknüpft werden können
- ***Eigene Validierer*** können erstellt und verwendet werden

Konvertierung:

- String-basierte HTTP-Eingaben werden durch ***Standardkonvertierer*** in Typen der zugehörigen Bean-Properties umgesetzt
- ***Eigene Konvertierer*** können definiert werden

Standardkonvertierer

• jeder Konvertierer muss Interface **`javax.faces.convert.Converter`** implementieren mit den Methoden:

```
public java.lang.Object getAsObject(
        javax.faces.context.FacesContext context ,
        javax.faces.component.UIComponent component ,
        java.lang.String value )
        //String wird in Objekt umgewandelt

public java.lang.String getAsString (
        javax.faces.context.FacesContext context ,
        javax.faces.component.UIComponent component ,
        java.lang.Object value )
        //Objekt wird in String umgewandelt
```

•JSF enthält Standardkonvertierer (im Package javax.faces.convert) für:

•Character, Boolean, Byte, Integer, Short, Long, Float, Double, char, boolean, byte, int, short, long, float, double, BigDecimal, BigInteger

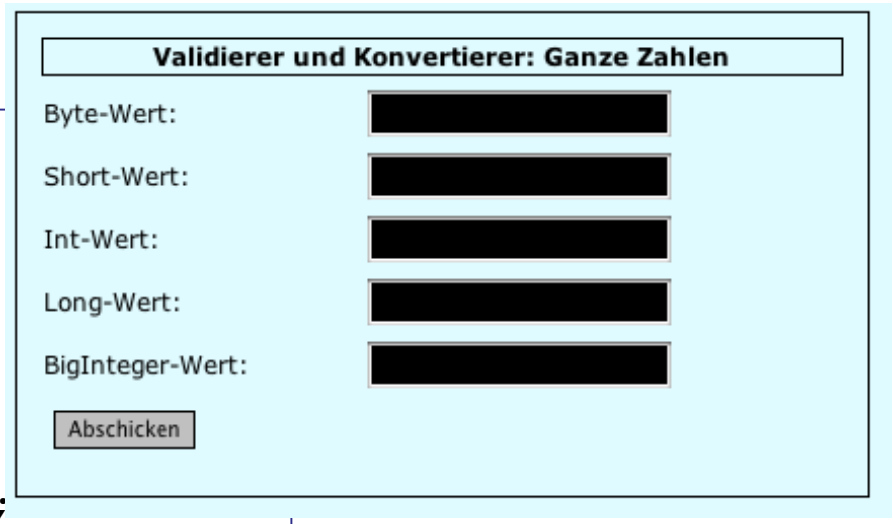
•diese ***Konvertierer arbeiten immer automatisch***, wenn ***Wertebindung mit Property*** von entsprechendem Typ erfolgt

Konvertierung ganzer Zahlen (JSF)

```
<h:panelGrid columns="2">
    <f:facet name="header">Validierer und Konvertierer: Ganze Zahlen
    </f:facet>
    <h:outputLabel for="byteValue" value="Byte-Wert:" />
    <h:inputText id="byteValue"
            value="#{ganzeZahlenHandler.byteValue}" />
    <h:outputLabel for="shortValue" value="Short-Wert:" />
    <h:inputText id="shortValue"
            value="#{ganzeZahlenHandler.shortValue}" />
    <h:outputLabel for="intValue" value="Int-Wert:" />
    <h:inputText id="intValue"
            value="#{ganzeZahlenHandler.intValue}" />
    <h:outputLabel for="longValue" value="Long-Wert:" />
    <h:inputText id="longValue"
            value="#{ganzeZahlenHandler.longValue}" />
    <h:outputLabel for="bigIntValue" value="BigInteger-Wert:" />
    <h:inputText id="bigIntValue"
            value="#{ganzeZahlenHandler.bigIntValue}" />
</h:panelGrid>
```

vc.xhtml

Konvertierung ganzer Zahlen (Java)



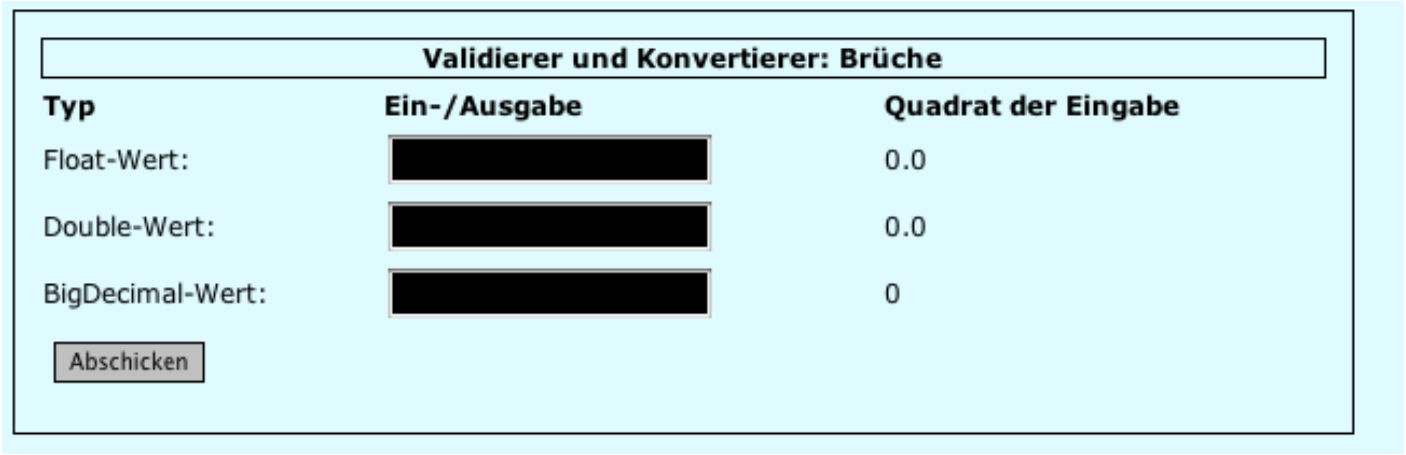

```
import java.math.BigInteger;

public class GanzeZahlenHandler {
    private Byte byteValue;
    private Short shortValue;
    private Integer intValue;
    private Long longValue;
    private BigInteger bigIntValue;

// ab hier nur Getter und Setter
public Byte getByteValue() {
    return byteValue;
    }

public void setByteValue(Byte byteValue) {
    this.byteValue = byteValue;
    }
...
}
```

GanzeZahlenHandler.java

Konvertierung gebr. Zahlen (JSF)

```
<h:panelGrid columns="3">
    <f:facet name="header">Validierer und Konvertierer: Brüche</f:facet>
    <h:outputText value="Typ" style="font-weight: bold;"/>
    <h:outputText value="Ein-/Ausgabe" style="font-weight: bold;"/>
    <h:outputText value="Quadrat der Eingabe" style="font-weight: bold;"/>
    <h:outputLabel for="floatValue" value="Float-Wert:" />
    <h:inputText id="floatValue" value="#{bruecheHandler.floatValue}" />
    <h:outputText value="#{bruecheHandler.floatValueQuadrat}" />
    <h:outputLabel for="doubleValue" value="Double-Wert:" />
    <h:inputText id="doubleValue" value="#{bruecheHandler.doubleValue}" />
    <h:outputText value="#{bruecheHandler.doubleValueQuadrat}" />
    <h:outputLabel for="bigDecimalValue" value="BigDecimal-Wert:" />
    <h:inputText id="bigDecimalValue" value=„{bruecheHandler.bigDecimalValue}“/>
    <h:outputText value="#{bruecheHandler.bigDecimalValueQuadrat}" />
</h:panelGrid>
```

brueche.xml

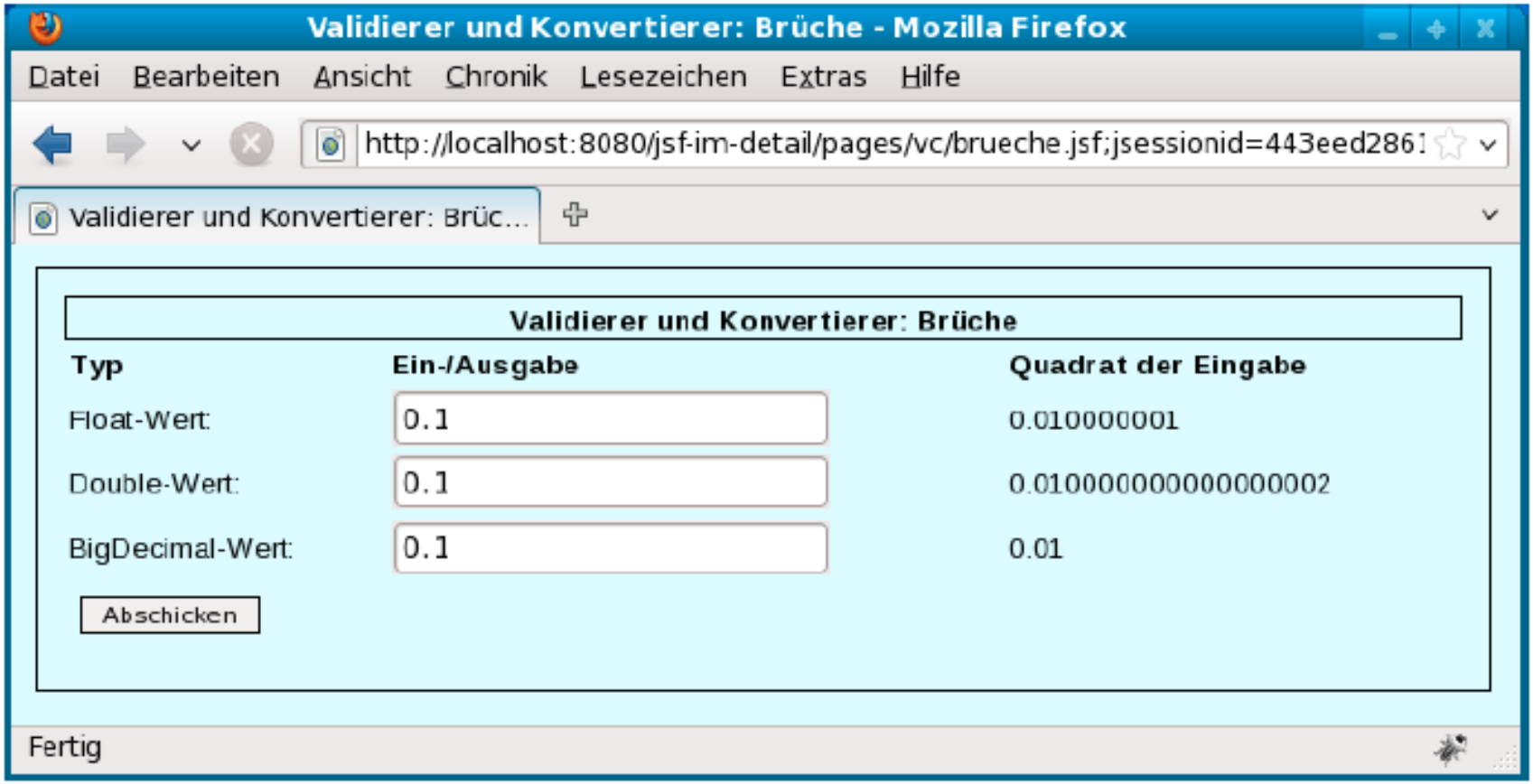


Konvertierung gebr. Zahlen (Java)

```
import java.math.BigDecimal;
public class BruecheHandler {
    private Float floatValue;
    private Double doubleValue;
    private BigDecimal bigDecimalValue;

    public String abschicken() {
        return null;}

    public Float getFloatValueQuadrat() {
        if (floatValue == null)
        return (float) 0.0;
        return floatValue * floatValue;}

    public Double getDoubleValueQuadrat() {
        if (doubleValue == null)
        return 0.0;
        return doubleValue * doubleValue;}

    public BigDecimal getBigDecimalValueQuadrat() {
        if (bigDecimalValue == null)
        return new BigDecimal(0);
        return bigDecimalValue.multiply(bigDecimalValue);}
…
        // ab hier nur Getter und Setter
```

BruecheHandler.java

Konvertierung gebr. Zahlen (Java)



Rechne nie mit Float und Double, denn das gibt nur Trouble.

import java.math.BigDecimal;

Kalenderdaten und Zahlen

Zusätzliche Standardkonvertierer von JSF:

Datumskonvertierer unter Beachtung der Lokalisierung:

- **`<f:convertDateTime>`**
  - Attribut **`type`** mit den Werten **`date, time`** und **`both`**
  - Attribut **`timezone, locale ,`**
  - Alternative zu **`type : pattern`**

Zahlenkonvertierer unter Beachtung der Lokalisierung:

- **`<f:convertNumber>`**
  - Attribute wie **`type`** mit den Werten **`number, currency, percent`**
  - Attribute: **`locale, currencyCode, CurrencySymbol`**
  - Alternative zu **`type : pattern`**

Konvertierung von Kalenderdaten (JSF)

Beispiel für Nutzung des DateTimeConverter (date-time-de.xhtml):

```
<f:view locale="de">      //deutsche Lokalisierung für view
<h:form>
    <h:panelGrid columns="2" rowClasses="odd,even">
    <f:facet name="header">Der DateTimeConverter</f:facet>

    <h:outputText value="ohne Attribute" />
    <h:outputText value="#{dtHandler.date}" />

    <h:outputText value="type=&quot;date&quot;" />
    <h:outputText value="#{dtHandler.date}">
        <f:convertDateTime type="date" />
    </h:outputText>

    <h:outputText value="type=&quot;time&quot;" />
    <h:outputText value="#{dtHandler.date}">
        <f:convertDateTime type="time" />
    </h:outputText>

    <h:outputText value="type=&quot;both&quot;" />
    <h:outputText value="#{dtHandler.date}">
        <f:convertDateTime type="both" />
    </h:outputText>
```

Sun Oct 28 19:48:16 CET 2012

28.10.2012

18:48:16

28.10.2012 18:48:16

...

Konvertierung von Kalenderdaten (java)

Beispiel für Nutzung des DateTimeConverter (DateTimeHandler.java):

```
package de.jsfpraxis.detail.vc;

import java.util.Date;



public class DateTimeHandler {

		public Date getDate() {
			return new Date();
		}
}
```

Konvertierung von Kalenderdaten

Der DateTimeConverter	
Attribute	**Darstellung**
ohne Attribute	Sun Oct 28 19:48:16 CET 2012
type="date"	28.10.2012
type="time"	18:48:16
type="both"	28.10.2012 18:48:16
type="both" timeZone="America/Los_Angeles"	28.10.2012 11:48:16
type="both" timeZone="Europe/Berlin"	28.10.2012 19:48:16
type="date" dateStyle="short"	28.10.12
type="date" dateStyle="medium"	28.10.2012
type="date" dateStyle="long"	28. Oktober 2012
type="date" dateStyle="full"	Sonntag, 28. Oktober 2012
type="time" dateStyle="short"	18:48:16
type="time" dateStyle="medium"	18:48:16
type="time" dateStyle="long"	18:48:16
type="time" dateStyle="full"	18:48:16
type="date" pattern="dd.MM.yyyy"	28.10.2012
type="date" pattern="dd. MMM yyyy"	28. Okt 2012
type="date" pattern="'Heute,' EEEE 'der' dd. MMMM yyyy"	Heute, Sonntag der 28. Oktober 2012
type="date" pattern="dd.MM.yyyy G, HH:mm:ss:SSS"	28.10.2012 n. Chr., 18:48:16:895

Konvertierung von Kalenderdaten: Attribute des `<f:convertDateTime>` Tabelle 4.4

Attribut	Werte und Beschreibung
`type`	`date` (Default), `time` oder `both`. Anzeige von Datum, Zeit, oder Datum und Zeit.
`dateStyle`	`short`, `medium` (Default), `long` und `full`. Formatangabe für den Datumteil, falls `type` gesetzt.
`timeStyle`	`short`, `medium` (Default), `long` und `full`. Formatangabe für den Zeitteil, falls `type` gesetzt.
`timeZone`	Angabe der Zeitzone. Falls nicht gesetzt, ist der Default Greenwich-Mean-Time (GMT).
`locale`	Lokalisierung, entweder als Instanz von `java.util.Locale` oder als String.
`pattern`	Angabe eines Patterns zur Formatierung. Alternative zu `type`. Details siehe Tabelle 4.5.

Konvertierung von Kalenderdaten: Zeichen zur Verwendung des Attributs `pattern` Tab. 4.6

Zeichen	Bedeutung	Beispiel	Darstellung
G	Epoche	G	n. Chr.
y	Jahr	yyyy	2006
M	Monat des Jahres	MM MMM	06 Juni
w	Woche des Jahres	w	14
W	Woche im Monat	W	3
d	Tag im Monat	dd	05
D	Tag im Jahr	D	21
E	Tag in der Woche	EE EEEE	Mo Montag
h	Stunde (0–12, am/pm)	h	8
H	Stunde (0–23)	HH	08
m	Minuten	mm	26
s	Sekunden	ss	59
S	Millisekunden	SSS	123
’	Fluchtzeichen für Text	’Heute’	Heute
’’	Apostroph	’’	’

Konvertierung von Zahlen (JSF)

Beispiel für Nutzung des NumberConverters (number-converter.xhtml) :

```
<h:outputText value="ohne Attribute" />
<h:inputText value="#{numberHandler.bigDecimalValue}" />

<h:outputText value="type=&quot;number&quot;" />
<h:inputText value="#{numberHandler.bigDecimalValue}">
<f:convertNumber type="number" />
</h:inputText>

<h:outputText value="locale=&quot;en_US&quot;" />
<h:inputText value="#{numberHandler.bigDecimalValue}">
<f:convertNumber locale="en_US" />
</h:inputText>

<h:outputText value="type=&quot;percent&quot;" />
<h:inputText value="#{numberHandler.bigDecimalValue}">
<f:convertNumber type="percent" />
</h:inputText>
```

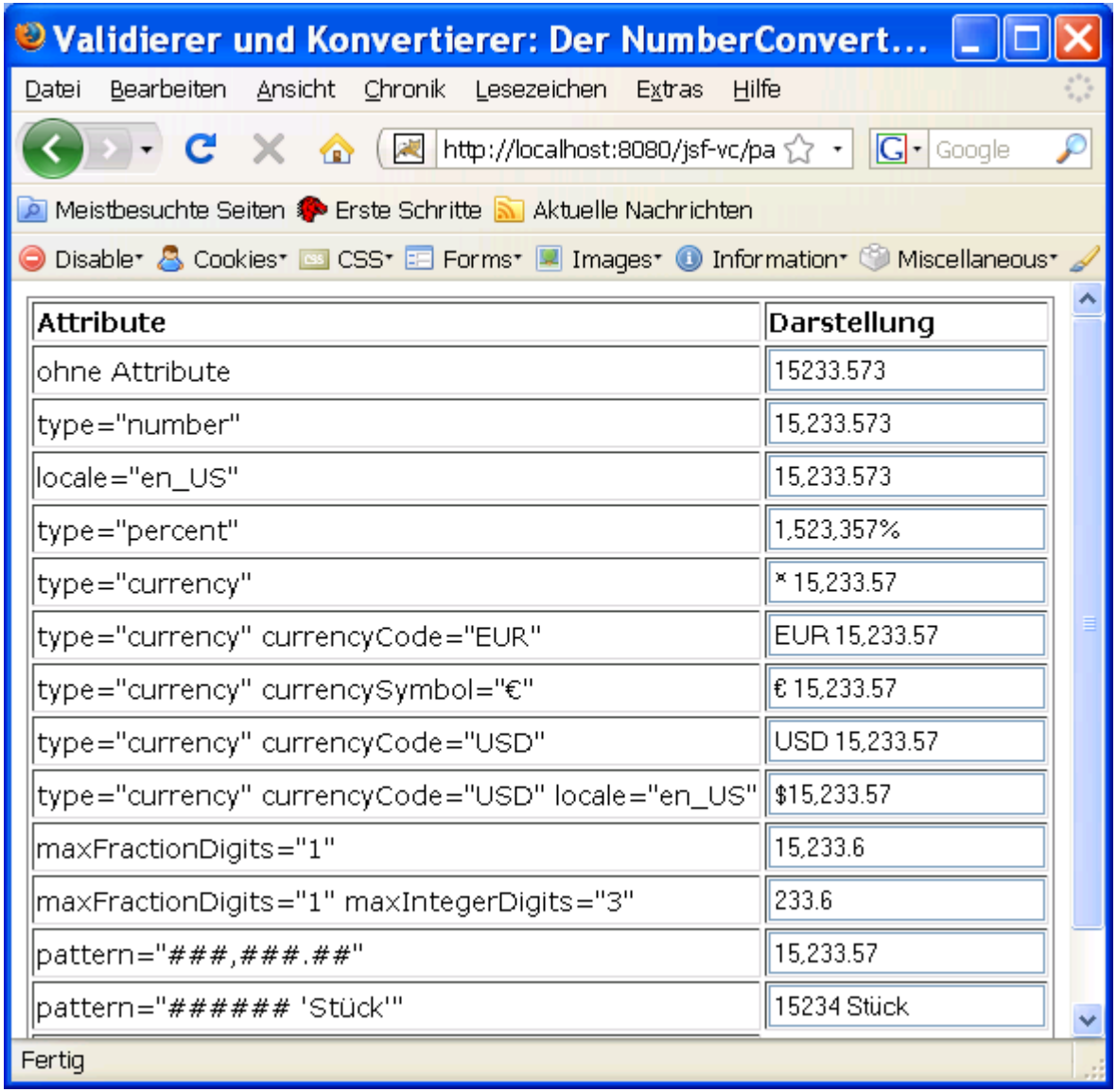
...

Konvertierung von Zahlen (Java)

Beispiel für Nutzung des NumberConverters (NumberHandler.Java) :

```
package de.jsfpraxis.detail.vc;

import java.math.BigDecimal;

public class NumberHandler {

	private BigDecimal bigDecimalValue = new
BigDecimal("15233.573");

	public String abschicken() {
		return null;
	}

	public BigDecimal getBigDecimalValue() {
		return bigDecimalValue;
	}

	public void setBigDecimalValue(BigDecimal bigDecimalValue) {
		this.bigDecimalValue = bigDecimalValue;
	}
}
```

Konvertierung von Zahlen



Attribute	Darstellung
ohne Attribute	15233.573
type="number"	15,233.573
locale="en_US"	15,233.573
type="percent"	1,523,357%
type="currency"	¤ 15,233.57
type="currency" currencyCode="EUR"	EUR 15,233.57
type="currency" currencySymbol="€"	€ 15,233.57
type="currency" currencyCode="USD"	USD 15,233.57
type="currency" currencyCode="USD" locale="en_US"	$15,233.57
maxFractionDigits="1"	15,233.6
maxFractionDigits="1" maxIntegerDigits="3"	233.6
pattern="###,###.##"	15,233.57
pattern="###### 'Stück'"	15234 Stück

Standardvalidierer I

- Validierung Hauptaufgabe eines GUIs
- JSF enthält einige Validierer,
- weitere eigene kann man hinzuzufügen
- In den Klassen aus Package **`javax.faces.validator`** enthalten
- Es gibt Validierer:
  - **`LengthValidator –`** überprüft die Länge der Eingabe
  - **`LongRangeValidator`** – überprüft Bereich der Eingabe
  - **`DoubleRangeValidator`** – überprüft den Wert der Eingabe bei Begrenzung durch **`maximum`** und **`minimum`**
  - **`RegexValidator (2.0)–`** überprüft regulären Ausdruck
  - **`RequieredValidator (2.0)`** – überprüft Existenz

Standardvalidierer II

Verwendung - in Eingabekomponenten enthalten:

- **<f:validateLength>**
- **<f:validateLongRange>**
- **<f:validateDoubleRange>**
- mit Attributen **minimum** und **maximum**

Beispiel:

```
<h:inputText value ="#{eingabeHandler.longValue}" required ="true">
<f:validateLongRange minimum ="100" maximum ="500" />
</h:inputText >
```

Standardvalidierer III

Verwendung :

- **`<f:validateRegex>`** – mit regulärem Ausdruck im Attribut **`pattern`**

Beispiel :

```
<h:inputText id="email1" value="#{eingabeHandler.email1}">
   <f:validateRequired/>
   <f:validateRegex pattern=".+@.+\..+" />
</h:inputText>
```

Standardvalidierer IIII

Verwendung :

- **<f:validateRequired>** – kann in Eingabekomponente enthalten sein oder auch mehrere Eingaben umfassen

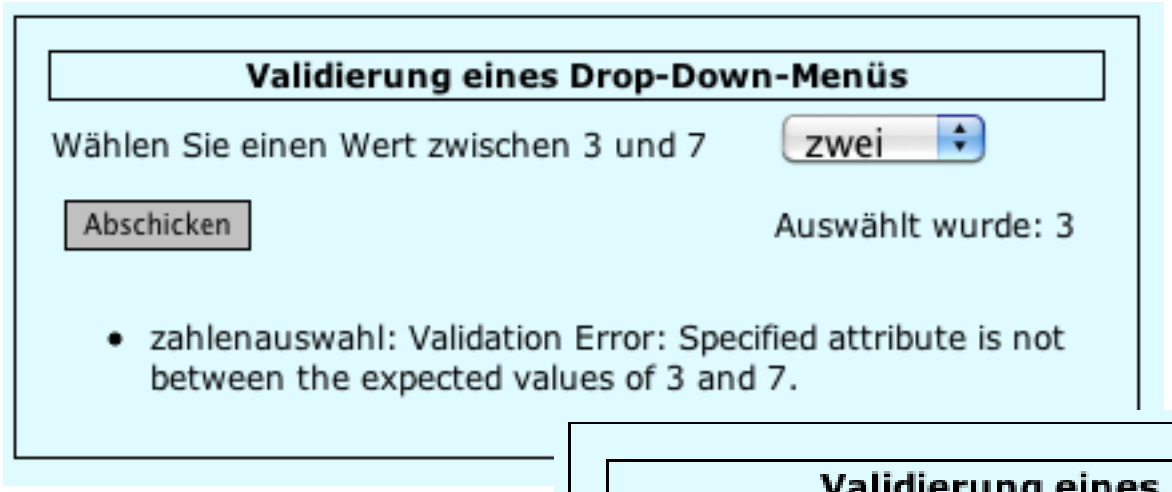
Beispiel :

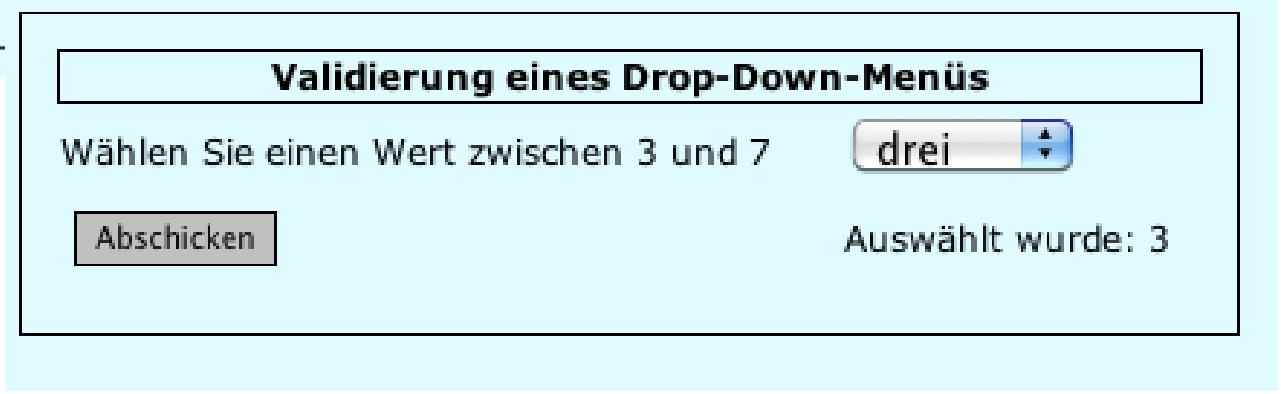
```
<f:validateRequired>
...
<h:inputText id="email1" value="#{eingabeHandler.email1}">
   <f:validateRegex pattern=".+@.+\..+" />
</h:inputText>
…
<h:inputText id="email2" value="#{eingabeHandler.email1}">
   <f:validateRegex pattern=".+@.+\..+" />
</h:inputText>
…
<f:validateRequired>
```

Standardvalidierer IIII

Verwendung :

- von Standardvalidierern ist prinzipiell nicht auf einfache Texteingaben beschränkt, sondern kann in allen Eingabekomponenten vorkommen:
  - Drop-Down-Menü **`<h:selectedOneMenu>`**,
  - Check-Box **`<h:selectedManyCheckbox>`**
  - Radio-Button **`<h:selectedOneRadio>`**





Standardvalidierer IIII

Beispiel: validiere-menu.xhtml

```
<h:panelGrid columns="2">
   <f:facet name="header">Validierung eines Drop-Down-Menüs</f:facet>
   <h:outputText value="Wählen Sie einen Wert zwischen #{eingabeHandler.min} und #
                        {eingabeHandler.max}" />
   <h:selectOneMenu id="zahlenauswahl" required="true"
                                    value="#{eingabeHandler.menueauswahl}">
     <f:selectItem itemValue="" itemLabel="" />
     <f:selectItem itemValue="1" itemLabel="eins" />
     <f:selectItem itemValue="2" itemLabel="zwei" />
                                    ...       //alle zwischen 3 und 8
     <f:selectItem itemValue="9" itemLabel="neun" />
     <f:validateLongRange minimum="#{eingabeHandler.min}"
                                 maximum="#{eingabeHandler.max}" />
   </h:selectOneMenu>
   <h:commandButton action="#{eingabeHandler.abschicken}“ value="Abschicken" />
   <h:outputText value="Auswählt wurde: #{eingabeHandler.menueauswahl} " />
</h:panelGrid>
```

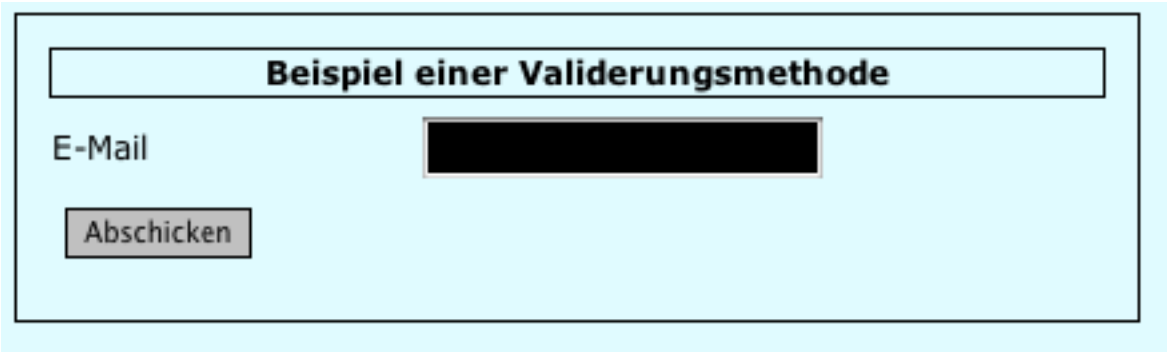
Validierungsmethoden

- Standardvalidierer relativ eingeschränkt
- mögliche Anwendungswünsche in der Beispielapplikation:
  - Überweisungsbetrag gedeckt ?
  - E-Mail-Adresse korrekt ?
  - passt PLZ zu Ort ?
  - . . .
- möglich mit:
  - eigener Validierer, als Komponente wiederverwendbar (hier nicht Gegenstand der Lehrveranstaltung, aber sehr komfortabel in der Nutzung)
  - eigene Validierungsmethode: jede Eingabekomponente kennt Attribut validator (Methodenbindung für Validierungsmethode)

Beispiel (validierungsmethode.xhtml):

```
<h:inputText id="eingabe"
    validator ="#{eingabeHandler.validateEmail }"
    value ="#{eingabeHandler.textValue}"
    required ="true" />
```

//Methode `validateEmail` wird
//an Property `textValue` gebunden



Validierungsmethoden

- Methode **`validateEmail()`** in der Managed-Bean EingabeHandler.java:

```
public void validateEmail ( FacesContext context,
                            UIComponent component,
                            Object value )
   throws ValidatorException {
   if (!(( String ) value ). matches (".*@.*")) {
                throw new ValidatorException (
                        new FacesMessage (" Klammeraffe fehlt "));
        }
   }
```

- Parameter:
  - Faces-Context des aktuellen Requests // **`FacesContext context`**
  - Komponente, deren Wert zu validieren ist // **`UIComponent component,`**
  - der Wert selbst **`value`** **`//Object value`**

- Werfen einer **`ValidatorException`** bei negativem Validierungsergebnis

Fehlermeldungen

- ***nicht geglückte*** Konvertierungen, Validierungen und andere Informationen müssen angezeigt werden

- JSF definiert dafür ein Verfahren:
  - interne Darstellung: Resource-Bundle **`javax.faces.Messages`**
  - Benutzeranzeige: <h:message>, <h:messages>
  - Lokalisiert als Ressource-Dateien (Englisch, Deutsch, Französisch,…)
  - Ressource-Dateien sind Properties-Dateien (java.util.Properties)
  - und damit Schlüssel/Wert-Paare mit Gleichheitszeichen als Trenner
  - JSF-Spec gibt Schlüssel vor, Werte nicht
  - Jeder Schlüssel tritt auch mit Suffix **`_detail`** auf (für detaillierte Meldung)
  - Fehlermeldungen für Standardkonvertierer und Standardvalidierer schon in den Sprachdateien enthalten z.B. nächste Seite…

Beispiel: Standardfehlermeldungen aus `javax.faces.Messages_de.properties`

```
javax.faces.component.UIInput.CONVERSION = \Konvertierungsfehler

javax.faces.component.UIInput.CONVERSION_detail = \"{0}": Fehler beim
                                                  Model-Update

javax.faces.component.UIInput.REQUIRED = \Validierungsfehler

javax.faces.component.UIInput.REQUIRED_detail = \"{0}": Eingabe
                                                erforderlich

javax.faces.component.UISelectOne.INVALID = \Validierungsfehler

javax.faces.component.UISelectOne.INVALID_detail = \"{0}": Wert ist
                                                   keine gueltige Auswahl .

javax.faces.validator.NOT_IN_RANGE = \Validierungsfehler
```

Fehlermeldungen generieren

- Parametrisierung der Fehlermeldungen mit {zahl} nach java.text.MessageFormat -> Positionsparameter

- Unterscheidung zwischen Meldungen :
  - ***für eine Seite*** **`<h:messages>`**
  - ***für eine Komponente*** **`<h:message for=“ “>`**

    - **`<h:messages>`** erlaubt boolesches Attribut **`globalOnly`**
    - **`true:`** nur Meldungen, die keiner Komponente zugeordnet sind
    - **`False:`** alle Meldungen einer Seite

Tipp zu Fehlermeldungen generieren

Achtung:

Während der Entwicklung empfiehlt es sich, ein `<h:messages>` in jeder Seite zu haben. Damit ist sicher gestellt, dass Sie auch alle Fehler angezeigt bekommen.

Alternativ/zusätzlich mit JSF2

Zentrale Konfiguration für Kontextparameter `PROJECT_STAGE` mit Wert `Development:`

```
<context-param>
<param-name>javax.faces.PROJECT_STAGE </param-name>
<param-value>Development</param-value>
<context-param>
```

Alle Werte erlaubt aus Enum:

`javax.faces.application.ProjectStage:`

`(Production, Development, UnitTest, SystemTest, Extension)`

Tipp zu Fehlermeldungen generieren

Achtung:

Während der Entwicklung empfiehlt es sich, ein `<h:messages>` in jeder Seite zu haben. Damit ist sicher gestellt, dass Sie auch alle Fehler angezeigt bekommen.

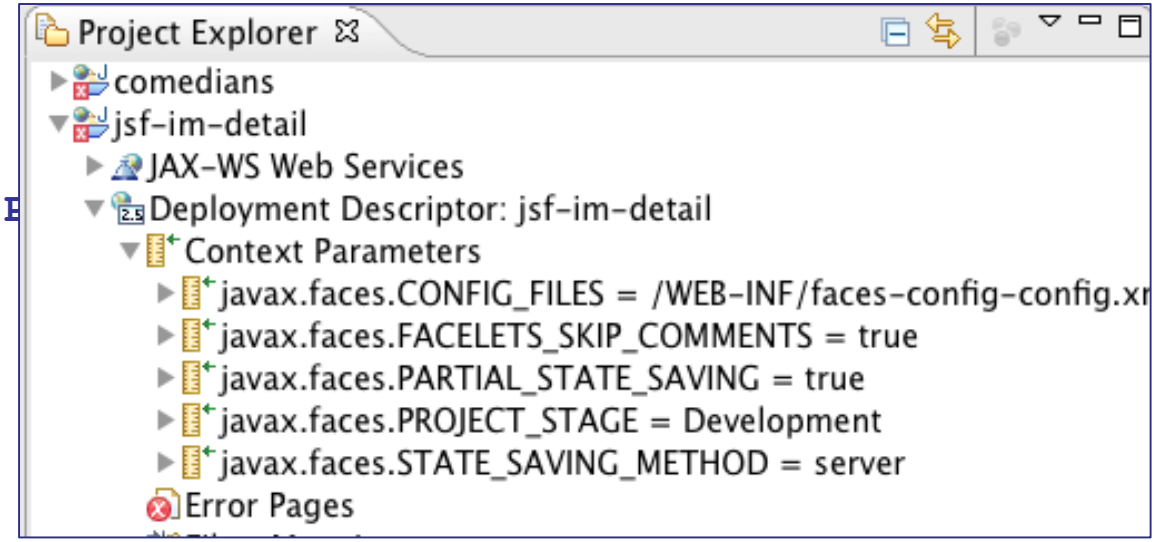
Alternativ/zusätzlich mit JSF2

Zentrale Konfiguration für Kontextparameter `PROJECT_STAGE` mit Wert `Development:`

```
<context-param>
<param-name>javax.faces.PROJECT_STAGE </param-name>
<param-value>Development</param-value>
<context-param>
```

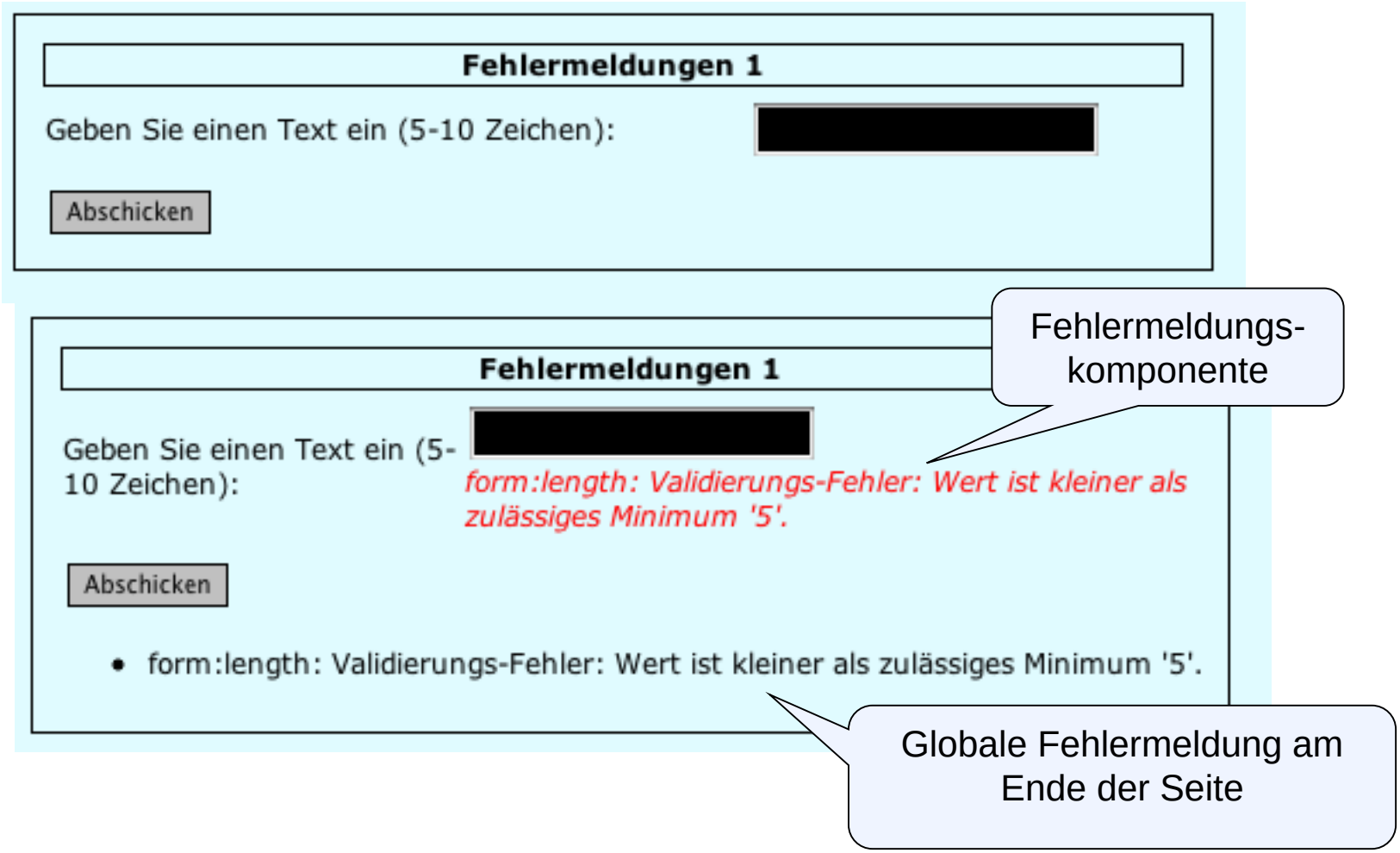
Alle Werte erlaubt aus Enum:

`javax.faces.application.P`



Fehlermeldungen generieren: Beispiel

Jsf-im-detail/vc/fehlermeldungen-1.xhtml



Fehlermeldungen generieren: Beispiel

Jsf-im-detail/vc/fehlermeldungen-1.xhtml

```
<h:form id="form">
   <h:panelGrid columns="2">
      <f:facet name="header">Fehlermeldungen 1</f:facet>
      <h:outputLabel for="length" value="Geben Sie einen Text ein (5-10 Zeichen):" />
      <h:panelGroup>
         <h:inputText id="length" value="#{meldungenHandler.text}" required="true">
                                   <f:validateLength minimum="5" maximum="10" />
         </h:inputText>
         <h:message for="length" styleClass="meldung" />
      </h:panelGroup>
   </h:panelGrid>
   <h:commandButton action="sucess" value="Abschicken" />
   <h:messages showDetail="true" showSummary="false" />
</h:form>
```

Längenvalidierer

Fehlermeldungskomponente

Globale Fehlermeldung mit Attributen für detaillierte Ausgabe

Css-Klasse zu Meldungen im stylesheet:

```
.meldung {
   font-style : italic;
   font-size : larger;
   color : red;
}
```

Eigene Fehlermeldungen definieren

1) eigenes Message-Bundle in faces-config.xml definieren

```
<application>
<message-bundle>de.jsfpraxis.detail.vc.meine-meldungen</message-bundle>
</application>
```

2) Die Datei **meine-meldungen.properties** anlegen in **de.jsfpraxis.detail.vc:**

```
<javax.faces.validator.LengthValidator.MAXIMUM = Fehler
javax.faces.validator.LengthValidator.MAXIMUM_detail = Der eingegebene
   Wert ist länger als die maximal zulässige Anzahl von {0} Zeichen.
javax.faces.validator.LengthValidator.MINIMUM = Fehler
javax.faces.validator.LengthValidator.MINIMUM_detail = Der eingegebene
   Wert ist kürzer als die minimal zulässige Anzahl von {0} Zeichen.
```

Fehlermeldungen mit Validierungsmethode

Jsf-im-detail/vc/fehlermeldungen-2.xhtml

```
<h:form id="form">
   <h:panelGrid columns="2">
      <f:facet name="header">Fehlermeldungen 2</f:facet>
      <h:outputLabel for="length" value="Geben Sie einen Text ein (3-7 Zeichen):" />
      <h:panelGroup>
         <h:inputText id="length" value="#{meldungenHandler.text}"
                              required="true“ validator="#{meldungenHandler.validateText}">
                                    <f:validateLength minimum=“3" maximum=“7" />
         </h:inputText>
         <h:message for="length“ errorClass="meldung“
                                    infoClass=“info“ />
      </h:panelGroup>
   </h:panelGrid>
   <h:commandButton action="sucess" value="Abschicken" />
   <h:messages showDetail="true" showSummary="false" />
</h:form>
```

Validierungsmethode **validateText**

Css-Klasse **info**
Wird der Meldung zugewiesen.

Css-Klasse info im stylesheet:

```
.info {
   font-style : italic;
   color : blue;
}
```

Validierungsmethode `validateText()`

Codeausschnitt:
~src/de.jsfpraxis.vc/MeldungenHandler.java

```
public void validateText(FacesContext context, UIComponent component,Object value) throws
ValidatorException {
int min = 0, max = 0;
int length = ((String) value).length();

Validator[] validator = ((UIInput) component).getValidators();
String mb = context.getApplication().getMessageBundle();
ResourceBundle rb = ResourceBundle.getBundle(mb);

for (int i = 0; i < validator.length; i++) {
    if (validator[i] instanceof LengthValidator) {
          LengthValidator lv = (LengthValidator) validator[i];
          min = lv.getMinimum();
          max = lv.getMaximum();
              if (length == min || length == max) {
                    String message = rb.getString("de.jsfpraxis.MINMAX");
                    String messageDetail = rb.getString("de.jsfpraxis.MINMAX_detail");
                    context.addMessage(component.getClientId(context),
                    new FacesMessage(FacesMessage.SEVERITY_INFO,message, messageDetail));
}}}}
```

Lokale Fehlermeldungen ab JSF2.0

Jsf-im-detail/vc/fehlermeldungen-3.xhtml

```
<h:form id="form">
   <h:panelGrid columns="2">
      <f:facet name="header">Fehlermeldungen 3 mit label-Attribut von JSF2.0</f:facet>

      <h:outputLabel for="length" value="Geben Sie einen Text ein (5-10 Zeichen):" />
      <h:panelGroup>
         <h:inputText id="length“ label=“Text“ value="#{meldungenHandler.text}“
required="true“
                      requiredMessage=“Der Text muss eingegeben werden.“
                       validatorMessage=“Die Textlänge liegt außerhalb des zulässigen
Bereichs“>
                    <f:validateLength minimum=“5“ maximum=“10“ />
         </h:inputText>
         <h:message for="length“ styleClass="meldung“/>
      </h:panelGroup>
   </h:panelGrid>
   <h:commandButton action="sucess" value="Abschicken" />
   <h:messages showDetail="true" showSummary="false" />
</h:form>
```

Attribut **label=“Text“** kennzeichnet zu validierende Komponente

Attribute **requieredMessage, validatorMessage** sind nur für diese eine Eingabekomponente gültig

Java Server Faces

1. Kontext und Motivation
2. Eigenschaften und Verwendung
3. Bearbeitungsmodell
4. ManagedBeans
5. JSF-EL
6. Validierung und Konvertierung
7. **Darüber hinaus**

Navigation

Aufgabe der Navigationskomponente (Navigation-Handler) ist die Unterstützung des anwendungsbezogenen Workflows

- Frühe statische Web-Anwendungen mit in HTML hat verlinkten
- JSF für Unternehmensanwendungen dynamisch generierten Verlinkungen der Views
- ***Ziel ist eine Navigation auf Basis anwendungsbezogener Workflows***
- Dafür besitzt JSF eigenen ***Navigation-Handler:***
  - ***Action-Events*** , bzw. deren Behandlung geben einen Wert zurück, der für die Navigation verwendet wird.
  - ***Navigationsregeln in XML-*** hinterlegen deklarative Navigationspfade
    - Besser bei größeren Projekten
    - Überschreiben implizite Navigationsregeln
  - Seit JSF2.0 auch ***implizite Navigation,*** ohne Regeln
    - String-Literale werden als view-Id‘s interpretiert
    - einfach verständlich für Anfänger und
    - gut geeignet für kleine Anwendungen

Implizite Navigation

Unter impliziter Navigation versteht man die direkte Angabe der View-Id, zu der navigiert werden soll, entweder als String-Lateral im JSF-Tag oder als Rückgabewert der Action-Methode.

- Direkte Angabe der View-Id
  - In JSF-Seite
  - Return der Action

- In JSF-Seite:
  - Attribut `action` bei `<h:commandButton>, <h:commandLink>`
  - Attribut `outcome` bei `<h:button>, <h:link>`

`Anmerkung:` Implizite Navigationsregeln werden von deklarativen Navigationsregeln in der JSF-Konfigurationsdatei überschrieben.

Implizite Navigation : Beispiel

Direkt in JSF-Seite:

```
<h:commandButton value ="... " action ="ziel.xhtml"/>
```

Oder als Action-Methode:

```
<h:commandButton value ="... "
                          action ="#{handler.action}"/>
```

mit Rückgabewert in Java-Action-Methode:

```
public String action () {
...
return "ziel.xhtml";
}
```

Anmerkung: Angabe der Dateinamenserweiterung .xhtml ist optional, es kann als Kurzform auch nur der Dateiname `ziel` ohne Erweiterung verwendet werden.

View-to-View-Regeln

- ***View:*** alle Komponenten, die eine UI-Seite in JSF ausmachen
- jede View hat eine eindeutige Bezeichnung: die ***View-Id***
- ***View-Id*** ist kontextrelativer Pfad einer JSF-Seite
- ***View-Id*** kann als Quelle und Ziel einer Navigationsregel verwendet werden

- ***Navigationsregel*** durch **`<navigation-rule>`** definiert
- **`<from-view-id>`**: die Ursprungsseite dieser Regel
- mehrere **`<navigation-case>`** als Sprungverteiler:
  - **`<to-view-id>`**: obligatorisch und Ziel dieser Navigationsregel
  - **`<from-outcome>:`** Ergebnis der Action-Methode (Konstante)
  - **`<from-action>:`** von welcher Action-Methode sinnvoll, falls mehrere Schaltflächen auf einer Seite

Struktur eines Ausschnittes aus faces-config.xml:

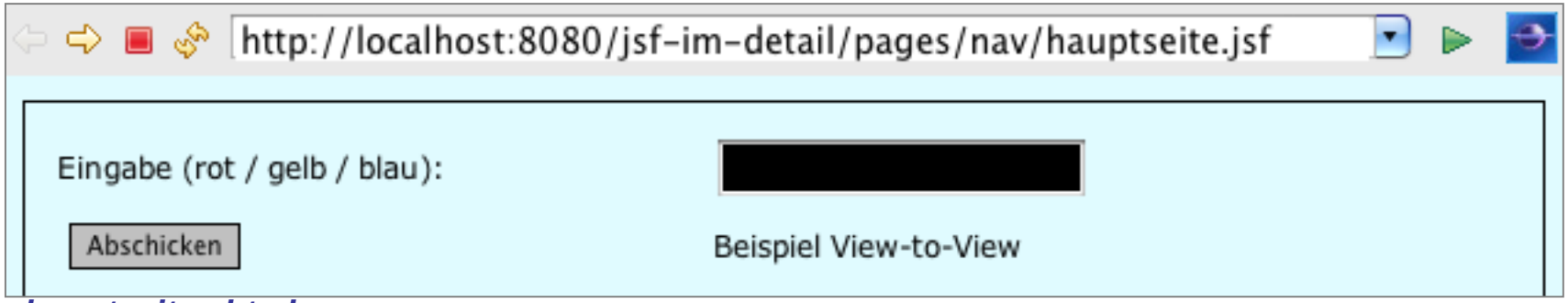
```
<navigation-rule>                 //definiert eine Navigationsregel
   <from-view-id>                 //oblig. Ursprungsseite
        <navigation-case>*        //unter welchen Bed., welche Zielseite ansteuern?
        <from-action>?            // opt. Welche Action-Methode
        <from-outcome>?           //opt. Ergebnis einer Actionmethode verwenden
        <to-view-id>              //obl. Zielseite
        <redirect>?
```

Beispiel: View-to-View-Regeln

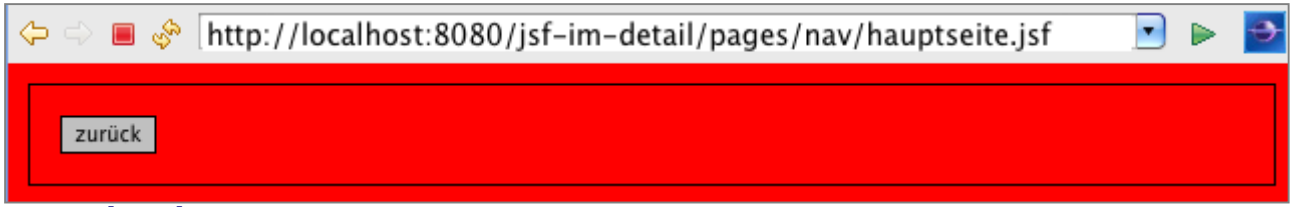
```
<navigation-rule>
   <description>
   Beispiele für alle relevanten View-to-View-Regeln
   </description>
   <from-view-id>/pages/hauptseite.xhtml</from-view-id>
        <navigation-case>
        <from-outcome>rot</from-outcome>
        <to-view-id>/pages/rot.xhtml</to-view-id>
   </navigation-case>
   <navigation-case>
        <from-outcome>gelb</from-outcome>
        <to-view-id>/pages/gelb.xhtml</to-view-id>
   </navigation-case>
   <navigation-case>
        <from-outcome>blau</from-outcome>
        <to-view-id>/pages/blau.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

faces-config-nav.xml

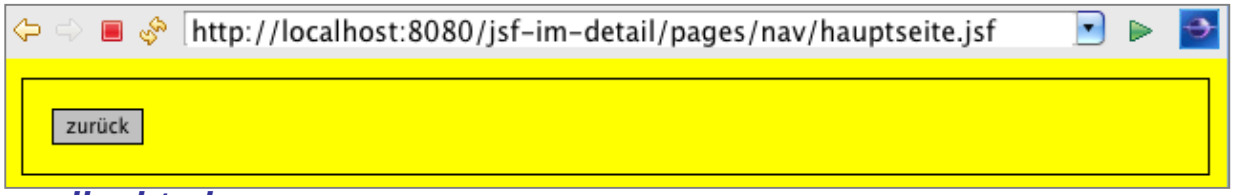
Beispiel: View-to-View-Regeln



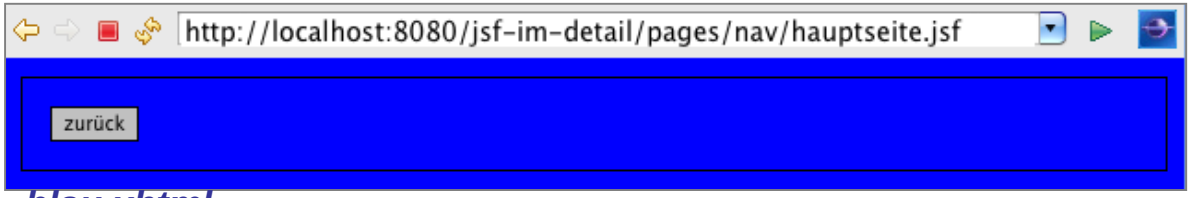
hauptseite.xhtml



rot.xhtml



gelb.xhtml



blau.xhtml

Beispiel: View-to-View-Regeln

Quellcode aus jsf-Seite, die Regel verwendet:

```
<h:outputLabel for="eingabe" value="Eingabe (rot/gelb/blau):" />
<h:inputText id="eingabe" value="#{naviHandler.eingabe}" />
<h:commandButton action="#{naviHandler.verteiler}"
                                    value="Abschicken" />
```

- Actionmethode `verteiler()` wird an comand-button gebunden
- Methode entscheidet anwendungslogisch über weitere Navigation
- nutzt dabei property `eingabe`:

```
public class NaviHandler {

private String eingabe;

public String verteiler() {
    if (eingabe.equals("rot"))
        return "rot";
    else if (eingabe.equals("gelb"))
        return "gelb";
    else if (eingabe.equals("blau"))
        return "blau";
    else
        eingabe = "bitte nochmal";
return "error";
}
```

/jsf-detail-navigation/web/webinf/faces-config.xml

Navigationsregel für mehrere Seiten

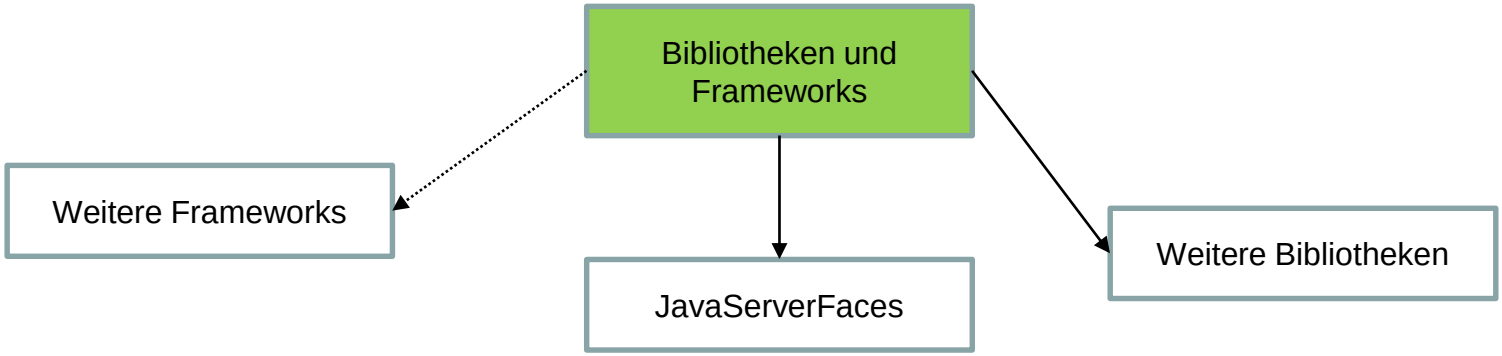
Quellcode aus faces-config.xml:

```
<navigation-rule>
   <description>
        Beispiel für Wildcard
   </description>
   <from-view-id>/pages/*</from-view-id>
   <navigation-case>
        <from-outcome>zurück</from-outcome>
        <to-view-id>/pages/nav/hauptseite.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

- durch “*“ werden mehrere gültige jsf-Seiten der Reihe nach angesprochen.

/jsf-detail-navigation/web/webinf/faces-config.xml

Darüber hinaus



Vertiefung im Fach WebEngineering

Links:

Frameworks und Bibliotheken - https://hosting.1und1.de/digitalguide/websites/web-entwicklung/beliebte-javascript-frame

-