HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

Sommersemester 2014

**Hochschule für angewandte Wissenschaft München**
Studiengang: Wirtschaftsinformatik IB-4C

# Seminararbeit im FWP-Modul

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

# Benutzeroberflächen mit JSF 2.2


vorgelegt von: Nikola Topalovic
Matrikelnummer: 04898112
Telefon: 08141 / 223812
Email: topalovic.nikola@gmx.de


eingereicht bei: Herr Michael Theis

Abgabetermin: 09.05.2014

# Inhaltsverzeichnis

1 Einleitung....2
2 Geschichte der Webentwicklung ....3
3 Grundlagen JSF ....4
3.1 Softwareumgebung ....5
3.2 JSF Konfiguration....5
4 Das Model-View-Controller-Prinzip....6
4.1 Model 1....7
4.2 Model 2....7
5 Seitendeklarationssprachen ....8
5.1 Facelets....8
5.2 Templates....9
6 Konzepte von JSF ....11
6.1 Komponenten ....11
6.2 Managed-Beans ....13
6.3 Unified Expression Language....15
6.3.1 Value-Expressions ....15
6.3.2 Method-Expressions....16
6.4 Konvertierung ....17
6.4.1 Standardkonverter....17
6.4.2 Benutzerdefinierte Konverter....19
6.5 Validierung ....20
6.5.1 Bean Validation nach JSR-303 ....20
6.5.2 Standardvalidatoren ....22
6.6 Navigation ....22
6.6.1 Implizite Navigation ....23
6.6.2 Regelbasierende Navigation ....23
6.7 Lebenszyklus....24
6.8 Ereignisse....26
7 JSF Standard-Komponenten ....27
7.1 Core-Tag-Library ....28
7.2 HTML-Custom-Tag-Library....29
7.2.1 Befehlskomponenten....29
7.2.2 Ausgabekomponenten ....30
7.2.3 Ausblick auf weitere Komponenten ....31
8 PrimeFaces ....32
8.1 Grundlegendes....33
8.2 p:menubar ....33
8.3 p:rating ....35
8.4 Themes ....36
9 JSF und CDI....37
9.1 Konversationen ....38
10 Fazit....39
11 Literaturverzeichnis....41
12 Abbildungsverzeichnis....41
13 Tabellenverzeichnis....41
14 Listingverzeichnis ....42
Erklärung ....43

# 1 Einleitung

Die Popularität des Internets wächst immer weiter und die Anwendungsvielfalt der Programme steigt ins Unermessliche. Immer mehr Firmen präsentieren sich im World Wide Web oder greifen auf diverse Webapplikationen und Services zu.

Die steigende Beliebtheit bringt jedoch auch Probleme mit sich, denn Anwendungsentwickler müssen sich mit immer komplexer werdenden Programmier-Codes beschäftigen.
Erschwert wird die Webentwicklung durch die stetig steigenden Qualitätsanforderungen, damit die Anwendungen auch nachhaltig an neue Anforderungen angepasst werden können.
Umfangreiche Entwicklungen erfordern neben sauberen Dokumentationen auch klar voneinander abgekapselte und strukturierte Module.

Viele aktuelle Webentwicklungen leiden unter dem Problem, dass der Programmier-Code nicht klar strukturiert und somit nicht nur schwer zu verstehen ist, sondern auch Wartungsarbeiten sowie Software-Erweiterungen sehr schwer umsetzbar werden. Ein recht typisches Problem ist die Vermischung von Programmier-Codes der Darstellungs- und Geschäftslogikschicht. Hier sollten Programmierer nicht komplett auf sich alleine gestellt werden, sondern durch ein unterstützendes Entwicklungs-Framework geleitet werden.

Bei der Entwicklung von Webapplikation gibt es bereits viele Technologien von diversen Herstellern. Dabei gibt es neben kostenpflichtigen Technologien auch diverse Open Source Produkte.

Speziell Java-Entwicklern steht seit über 15 Jahren die JavaServer Pages-Technologie (JSP) zum Erstellen von dynamischen Webseiten zur Verfügung. Mit JavaServer Faces (JSF) steht seit 2004 den Java-Entwicklern noch eine weitere Entwicklungstechnologie zur Verfügung, die besonders seit dem Update auf die Version 2.0 für viel Aufsehen sorgt.

In den folgenden Kapiteln soll es um die Frage gehen was JSF ist, welche Konzepte hinter JSF stecken, wie JSF funktioniert und ob JSF wirklich den Anforderungen eines modernen Webframework gerecht wird.

Für das Verständnis dieser Arbeit sind Grundkenntnisse der HTML- und Java-Programmierung erforderlich.

Ziel der Arbeit ist nicht nur die Pflichterfüllung im FWP-Modul „Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen" an der Hochschule München, sondern auch allen interessierten Webentwicklern einen umfassenden Überblick über das erstellen dynamischer Benutzeroberflächen mit JSF zu bieten. Die vielen individuellen Beispiele und Listings sollen zum Verständnis in dem jeweiligen Themengebiet über die Sachverhalte beitragen. Bei umfangreichen Teilgebieten werden oft Ausblicke gewährt, die ein Selbststudium anregen und erleichtern sollen.

# 2 Geschichte der Webentwicklung

Ein Rückblick in die Geschichte verdeutlicht die Problematik bei der Programmierung von Webapplikationen. Es geht los bei der Übertragung der ersten Hypertext Markup Language (HTML) über das Hypertext Transfer Protocol (HTTP) im Jahre 1991. Dieser Erfolg hat das Zeitalter der Webentwicklung ins Leben gerufen. Während sich HTML in den Folgejahren immer mehr zur Layoutsprache erweiterte, haben sich sowohl browserseitige als auch serverseitige Skriptsprachen entwickelt, um den dynamischen Anforderungen der Webentwicklung nachzukommen. (vgl. Kurz et al. 2014, S. 1f)

```java
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class BeispielServlet extends HttpServlet
{
	public void doGet(HttpServletRequest req, HttpServletResponse res)
		throws ServletException, IOException
	{
		String name = req.getParameter("name");
		String text = req.getParameter("text");

		res.setContentType("text/html");

		ServletOutputStream out = res.getOutputStream();
		out.println("<html><head><title>");
		out.println(name);
		out.println("</title></head><body>");
		out.println(text);
		out.println("</body></html>");
		out.flush();
	}
}
```

***Listing 1:** Einfaches Servlet - Behandlung der GET-Methode*

Seit dem Jahr 1997 gab es auch für Java-Entwickler mit der Servlet-Technologie eine Möglichkeit HTML-Seiten aus Befehlen im Java-Code zu erzeugen. Prinzipiell beruht diese Technologie darauf

die HTML-Codes mittels Druckanweisungen („System.out.print"-Befehle) zu erzeugen und über einem „OutputStream" zu übertragen. (ebd. 2014, S. 2)

Listing 1 zeigt ein Muster eines alten Servlets und soll verdeutlichen, wie unübersichtlich die Methode war. Servlets haben sich jedoch im Laufe der Zeit weiterentwickelt und etwas vereinfacht. Eine Vereinfachung war beispielsweise das Auslagern von Hilfsklassen, welche das Schreiben von HTML-Tags erleichterten. Doch bei umfangreicheren und komplexeren Seiten wird es auch mit den Hilfsklassen schnell unübersichtlich. (ebd. 2014, S. 2f)

Abhilfe konnte erst mit JSP geschaffen werden. JSP setzt im Gegensatz zu den vorangegangenen Methoden HTML als treibende Kraft ein. In HTML-Tags ist es möglich über sogenannte Scriplets, den Java-Code aufzurufen. Komplexe Entwicklungen konnten mit JSP deutlich vereinfacht werden. Das Problem war jedoch, dass die Entwickler immer mehr Codes in die JSP Seiten mit aufgenommen haben, sodass erneut Mischungen aus HTML-Tags und Java-Codes entstand. (ebd. 2014, S. 3)

```
<html>
      <body>
            Heute ist der <%= new java.util.Date() %>.
      </body>
</html>
```

***Listing 2:** Ein einfaches JSP-Beispiel*

Wie die Historie zeigt, war es mit den bis dato verfügbaren Technologien zwar möglich umfangreiche und komplexe Webapplikationen zu realisieren, welche aber im Aspekt Nachhaltigkeit nicht die wesentlichen Anforderungen erfüllen. JSP-Applikationen gelten heutzutage als schwer zu warten, da sie zu unübersichtlich sind.

# 3 Grundlagen JSF

Wie die Erkenntnisse zeigen, war es an der Zeit für ein modernes Tool, welches die Webentwicklung erleichtert und nachhaltige Projekte fördert.
Aufgrund dessen versuchte man mit JavaServer Faces auf Basis der Java Plattform Enterprise Edition ein modernes Webframework zu schaffen, dass die Entwicklung von grafischen Benutzeroberflächen ermöglicht und dabei kompatibel zur bisherigen Technologie JSP ist.

JSF wurde im Jahr 2004 als Standard veröffentlicht und galt als zu umständlich und unflexibel. Die JSF-Versionen 1.1 und 1.2 in den zwei darauf folgenden Jahren, verbesserten die Probleme nur unwesentlich. Erst im Jahre 2009 mit der Version 2.0, als Teil der Java EE 6, gelang es JSF sich auf dem Markt zu etablieren. Das Update 2.0 brachte nicht nur viele Vereinfachungen mit sich,

sondern auch viele neue Features, von denen einige im Rahmen dieser Arbeit noch vorgestellt werden. Die aktuelle Version 2.2 von JSF wurde im Mai 2013 veröffentlicht und gilt als Basis für diese Arbeit.

## 3.1 Softwareumgebung

Als Grundlage für eine JSF-Applikation ist eine Installation des Java Development Kits (JDK) in der Version 6 oder 7 erforderlich. Für die Entwicklung sollte eine moderne Entwicklungsumgebung wie Eclipse, NetBeans oder IntelliJ IDEA gewählt werden. Wobei prinzipiell auch eine Entwicklung mit einem simplen Editor möglich wäre. Bei Webentwicklungen wird zudem noch ein Anwendungsserver, wie beispielsweise GlassFish 4, benötigt.

Anstatt eines Anwendungsservers kann auch ein Servlet-Container ab der Versionsnummer 3.0 hergenommen werden. Tomcat 7 und Jetty 8 wären hierfür mögliche Varianten.
Es sollte jedoch berücksichtigt werden, dass Servlet-Container wesentlich eingeschränktere Funktionen haben und vieles manuell integriert werden muss. Weiterhin sollte die Verwendung von Appache Maven in Erwägung gezogen werden, da Maven ein äußerst hilfreiches Tool zum Verwalten von Projekten ist.

## 3.2 JSF Konfiguration

Die Konfiguration einer JSF-Anwendung erfolgt über zwei Dateien. Besonders wichtig ist die Deployment-Descriptor. Diese sollte als web.xml im Verzeichnis /WEB-INF abgelegt werden. Da JSF im Prinzip ein Aufsatz auf die Servlet-API ist, muss in der web.xml ein Servlet konfiguriert werden. Der Eintrag <servlet-mapping> erzeugt, dass alle Dateien mit der Endung .jsf von einem JSF-Servlet bearbeitet werden. Der Eintrag <welcome-file-list> definiert die Startseite der Webapplikation. Über <context-param> kann der Zustand einer Webapplikation bestimmt werden. In der Entwicklungsphase ist es üblich den Parameter „Development“ zu übergeben, da in diesem Zustand, JSF wesentlich detailliertere Fehlermeldungen erzeugt. Listing 3 zeigt eine web.xml-Datei.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
	http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
	version="3.0">

	<description>MeinProjekt</description>
	<servlet>
		<servlet-name>Faces Servlet</servlet-name>
		<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>Faces Servlet</servlet-name>
		<url-pattern>*.jsf</url-pattern>
	</servlet-mapping>
	<welcome-file-list>
		<welcome-file>index.html</welcome-file>
	</welcome-file-list>
	<context-param>
		<param-name>javax.faces.PROJECT_STAGE</param-name>
		<param-value>Development</param-value>
	</context-param>
</web-app>
```

***Listing 3:*** *web.xml*

Die Datei faces-config.xml ist für den Start einer Anwendung in JSF 2.2 nicht relevant und kann somit leer- beziehungsweise weggelassen werden. An bestimmten Stellen dieser Arbeit wird sie dennoch benötigt und entsprechend vermerkt.

Wenn Projekte mit Maven erstellt werden, ist noch die Datei pom.xml von Bedeutung, da hier alle Projektabhängigkeiten eingetragen werden, die von Maven verwaltet werden.

# 4 Das Model-View-Controller-Prinzip

Umfangreiche Webentwicklungen werden oft in verschiedene Schichten aufgeteilt, wobei jede Schicht unterschiedliche Aufgaben übernimmt. Ein typisches Schichtenmodell ist die 3-Schichten-Architektur. Dieses sieht die Spaltung einer Anwendung in drei Schichten vor: Daten-, Geschäftslogik- und Präsentationsschicht. JSF-Entwicklungen laufen in der Regel in der Präsentationsschicht ab, wobei eine Überlappung zur Geschäftslogikschicht gegeben ist.
Für die Präsentationsschicht konnten sich im Laufe der Jahre spezialisierte Formen des Musters des Model-View-Controller-Prinzips (MVC) etablieren.
Dieses Muster sieht die Strukturierung der Präsentationsschicht in die drei Einheiten Modell, Präsentation und Steuerung vor, die für jeweils unterschiedliche Aufgaben zuständig sind. Model 1 und Model 2 sind abgeänderte Varianten des MVC-Prinzips für den speziellen Einsatz im Web.

## 4.1 Model 1

Model 1 findet Einsatz in JSP und sieht keinerlei Trennung von View und Controller vor. Benutzer greifen also direkt auf eine View zu, die sowohl die Aufgabe der Darstellung übernimmt, als auch die Ablaufsteuerung, sowie die Instanziierung der Geschäftslogik. Im Konzept von Model 1 hat somit die View nicht nur die eigenen Aufgaben zu erledigen, sondern auch die Aufgaben eines Controllers. Das könnte bei kleinen Webanwendungen durchaus sinnvoll sein. Man sollte jedoch ab einer gewissen Größe der Anwendung den Aspekt der Code-Trennung nicht zu wenig Wert widmen, da besonders die Wartungsarbeiten dadurch erschwert werden.

## 4.2 Model 2

Eine striktere Trennung der Aufgaben verfolgt der Ansatz von Model 2, auf das auch das JSF-Framework basiert. Der Benutzer greift in diesem Muster nicht direkt auf eine View zu, sondern auf einen Controller. Auf Basis der vorgenommen Eingaben entscheidet der Controller dann, welche Aktionen ausgeführt werden sollen und welche Folgeseite aufgerufen wird. Das Model beinhaltet die darzustellenden Daten und wird in JSF durch Managed-Beans repräsentiert. Die View übernimmt die Aufgabe der Darstellung, für die es in JSF mehrere Deklarationsmöglichkeiten gibt. In Kapitel 5 werden Deklarationssprachen genauer betrachtet. Abbildung 1 zeigt die Einheiten des Model2-Prinzips im Zusammenspiel.

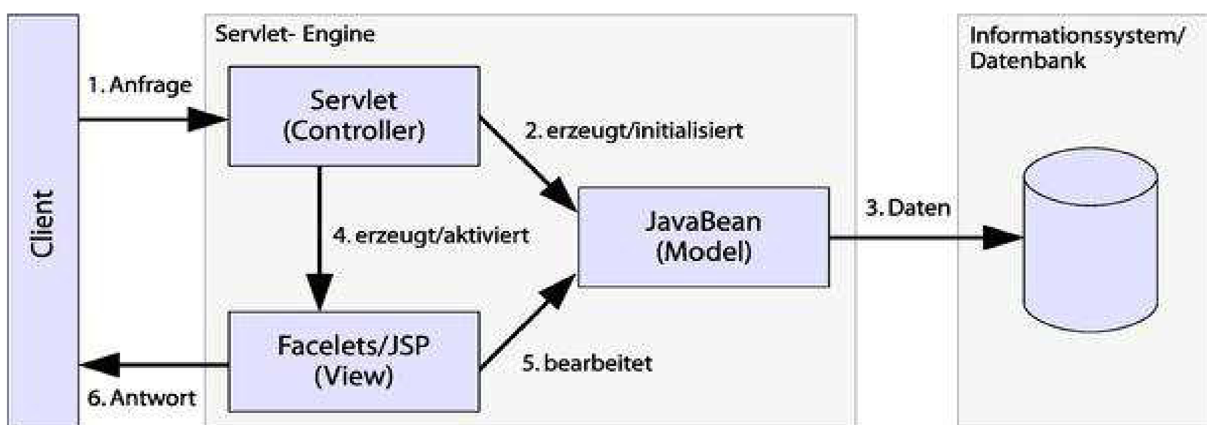


***Abbildung 1:*** *Model2-Prinzip*

# 5 Seitendeklarationssprachen

Bei Seitendeklarationssprachen (View Declaration Language; kurz: VDL) handelt es sich um Technologien, die zur Definition von Views verwendet werden.
Bis zur Veröffentlichung von JSF 2.0 war JSP die primäre VDL. Mit JSF 2.0 wurden die extra für JSF entwickelten Facelets zur primären VDL. Aus Kompatibilitätsgründen wird JSP aber immer noch unterstützt. Wie Kurz und Marinschek ausführen, sei JSF in Kombination mit JSP problematisch, da JSP-Anfragen einen wesentlich kürzeren und einfacheren Lebenszyklus hätten. (ebd. 2014, S. 73f).

Im nächsten Abschnitt werden Facelets genauer betrachtet und der wesentliche Vorteil gegenüber der JSP-Technologie wird erläutert.

## 5.1 Facelets

Eine Facelet ist im Wesentlichen ein XHTML-Dokument mit dem Doctype XHTML Transitional. Die Tag-Bibliotheken werden über die Namensräume im Wurzelelement des Dokuments eingebunden. Der restliche Teil des Dokuments besteht nur mehr aus Komponenten-Tags und reinem HTML, siehe Listing 4 (vgl. Kurz et al. 2014, S....)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
      <h:head>
            <title>Facelets</title>
      </h:head>
      <h:body>
            <h1><h:outputText value="meinFacelet"/></h1>
      </h:body>
</html>
```

***Listing 4:*** *Beispiel eines Facelets*

Praktischerweise verarbeiten Facelets sowohl die klassischen HTML-Tags als auch die Tags aus der HTML-Custom-Tag-Library. Die klassischen HTML-Tags sollten jedoch generell vermieden werden, da sie bei einigen Komponenten die Funktionalitäten einschränken können. Beispielsweise funktionieren eingebundene PrimeFaces-Komponenten nur, wenn sie innerhalb eines h:body-Elementes definiert werden. Bei Verwendung eines einfachen body-Elementes wäre

die Funktionalität von PrimeFaces-Komponenten nicht gegeben. Das Verhalten von JSF, wie es Seitendeklarationen interpretieren soll, kann über Kontextparameter in der web.xml gesteuert werden.

## 5.2 Templates

Viele Webapplikation bestehen aus einem einheitlichen Layout und Design, dass sich in den meisten Views wiederholt. Beispielsweise ist bei einem typischen 2-Spalten-Layout mit Kopf- und Fußzeile, in den meisten Fällen, sowohl der Kopfbereich als auch der Fußbereich für jede einzelne View gleich. Hier zeigen Facelets mit dem Template-Konzept den wohl größten Vorteil gegenüber der JSP-Technologie. Templates ermöglichen die zentrale Erfassung von Layout und Design. Die Kopf- und Fußzeile aus dem genannten Beispiel könnten somit zentral definiert werden und wären von jeder beliebigen View aus verwendbar. Eine zentrale Erfassung senkt nicht nur Redundanzen, sondern erhöht auch die Bedienbarkeit und Flexibilität einer Software. Änderungen am Design wären im Idealfall nur noch an einer Stelle im Programm auszuführen und in allen benutzenden Views kämen die Änderungen gleichzeitig zum Vorschein.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<head>
      <title>MeinLayout</title>
      <link rel="stylesheet" type="text/css" href="style.css"/>
</head>
<body>
      <div id="header">
            <ui:insert name="header">
                  <h:graphicImage value="/images/Header.png"/>
            </ui:insert>
      </div>
      <div id="content" style="margin: 50px">
            <ui:insert name="content"/>
      </div>
      <div id="footer">
            <ui:insert name="footer">
                  <h:graphicImage value="/images/Footer.png"/>
            </ui:insert>
      </div>
</body>
</html>
```

***Listing 5:*** *Ein einfaches Template*

Prinzipiell sind Templates gewöhnliche XHTML-Dokumente, in dem die grundlegende Seitenstruktur festgelegt wird. Der Unterschied sind die ersetzbaren Bereiche, die mittels ui:insert-

Tags über die Facelets-Tag-Library (namespace ui) realisiert werden können. Listing 5 zeigt ein einfaches Beispiel eines Templates mit drei ersetzbaren Bereichen names header, content und footer. Während header und footer jeweils ein Bild integrieren, ist der content-Bereich bis auf eine kleine Abstandsformatierung leer gelassen.

Sogenannte Template-Cients können nun auf dieses Template zugreifen und alle ersetzbaren Bereiche überschreiben. Das bedeutet, dass der Template-Client in dem Beispiel alle drei Bereiche überschreiben kann. Findet keine Überschreibung statt, ladet der Template-Client automatischen den Wert der innerhalb des ui:insert-Tags angegeben wurde.

Die Einbindung des Template-Clients erfolgt über die Tags ui:composition und ui:define. Während ui:composition über das Attribut „template“ Verbindung zum Template herstellt, überschreibt ui:define über das Attribut „name“ den jeweiligen Inhalt, siehe Listing 6.

```
<ui:composition template="template.xhtml"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
      <ui:define name="content">
            <h1><h:outputText value="mein Inhalt"/></h1>
      </ui:define>
</ui:composition>
```

***Listing 6:*** *Ein einfacher Template-Client*

Die in diesem Kapitel vorgestellte Template-Funktion gilt lediglich als Basisfunktion. Templates können noch viel mehr, wie beispielsweise über mehrstufiges Templating eine Hierarchie von Templates aufzubauen oder auch die Möglichkeit mehrere Templates gleichzeitig von einem Template-Client einzubinden. Aufgrund des Umfangs werden beide Möglichkeiten jedoch nicht näher vorgestellt. Abschließend präsentiert Abbildung 2 das Ergebnis der vorherigen beiden Listings.

***Abbildung 2:*** *Benutzung eines Templates*

# 6 Konzepte von JSF

Hinter der JSF-Techologie steht eine Reihe von Konzepten. Die Verbindung und das Zusammenspiel all dieser Konzepte sind maßgeblich für eine funktionierende Webapplikation. In den folgenden Abschnitten werden die wichtigsten Bestandteile von JSF und deren Funktionsweisen vorgestellt.

## 6.1 Komponenten

Das JSF-Framework ist komponentenorientiert. Das bedeutet, dass die Seiten in JSF-Anwendungen aus eigenständigen und wiederverwendbaren Bausteinen aufgebaut sind. JSF bietet eine umfangreiche Auswahl an vordefinierten Komponenten, die in Kapitel 7 erörtert werden.

Alle Komponenten zusammen werden als Ansicht bezeichnet. Die Komponenten sind dabei in Form eines Komponentenbaums miteinander verknüpft. Die Wurzel dieses Baumes ist die unsichtbare UIViewRoot-Komponente. Alle Komponenten hängen als Kinder unter diesem Element. Genauer gesagt werden die Tags der XHTML-Seiten von Facelets in Komponenten umgesetzt und im Komponentenbaum angeordnet. (vgl. Kurz et al. 2014, S. 24ff)

Listing 7 zeigt einen Code-Ausschnitt, der zum Aufbau des folgend aufgeführten Komponentenbaums führt.

```
<h:form>
      <h:panelGrid id="grid" columns="2">
            <h:outputLabel for="inputName" value="Name:"/>
            <h:inputText id="inputName" value="#{meineBean.meinName}"/>
            <h:commandButton id="btnAccept" value="Übernehmen"
                        action="/nächsteSeite.xhtml"/>
      </h:panelGrid>
</h:form>
```

***Listing 7:** Beispiel Komponentenbaum (Ausschnitt)*

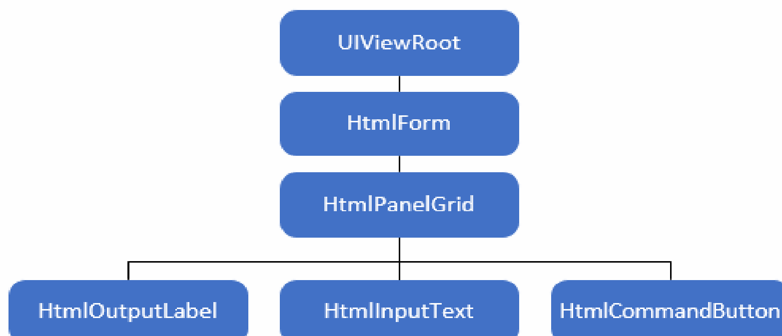


***Abbildung 3:** Beispiel Komponentenbaum*

Benutzer die nun den Inhalt dieser Seite sehen wollen, müssen den Namen dieser Seite mit der Endung .jsf in die Adressleiste eintippen. Das klingt zunächst etwas ungewöhnlich, da bisher immer nur von Seiten mit der Endung .xhtml die Rede war. Das liegt daran, dass nach außen hin nur die JSF-Ansicht sichtbar ist, die intern auf einer Seitendeklaration aufgebaut wird, wie sie bereits in Kapitel 5 besprochen wurde. JSP-Dateien können natürlich auch interpretiert werden. Hierzu muss nur ein zur JSP-Datei entsprechender View-Identifier in der web.xml gesetzt werden.

Bevor die Ausgabe einer Ansicht erfolgt, wird der fertig aufgebaute Komponentenbaum von einem Renderer in eine Ausgabesprache umgesetzt. In den meisten Fällen wird es sich dabei um eine HTML-Ausgabe handeln. Renderer können jedoch auch ausgetauscht werden, womit beispielsweise auch PDF-Ausgaben möglich wären. Die Renderer übernehmen also die in der Komponenteninstanz gespeicherten Daten und geben den entsprechenden HTML-Code für die jeweiligen Komponenten aus. (ebd. 2014, S. 28)

Abbildung 4 zeigt abschließend die HTML-Ausgabe zu Listing 7 und dem abgebildeten Komponentenbaum.

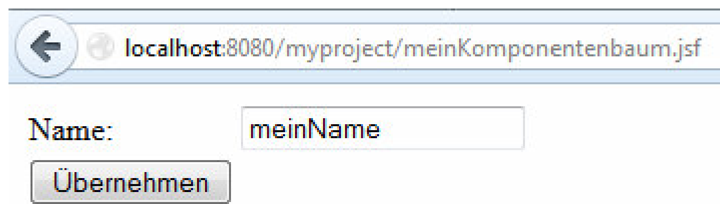


***Abbildung 4:** Komponentenbaum - Browseranzeige*

## 6.2 Managed-Beans

Managed-Beans sind zentraler Bestandteil in JSF-Anwendungen und bilden das Model beziehungsweise die Verbindung zum Modell und der Geschäftslogik.
Managed-Bean sind Java-Klassen, die prinzipiell nur einen Konstruktor ohne Parameter benötigen. Um die Werte repräsentieren zu können, werden in den Beans Datenfelder mit entsprechenden Getter- und Setter-Methoden definiert. Funktionalitäten erhalten die Beans durch Methoden. Listing 8 zeigt eine einfache Bean, die nur die Eigenschaft „`meinName`“ enthält und eine Methode „`willkommen()`“, die den Namen mit dem Vorwort „Hallo“ ausgibt. Da in der Bean „`MeineJSFBean`“ kein Konstruktor definiert wurde, wird der default-Konstruktor verwendet. Es gäbe auch die Möglichkeit nur eine Getter-Methode für „`meinName`“ zu definieren. In diesem Fall wäre „`meinName`“ nur eine lesbare Eigenschaft. (vgl. Kurz et al. 2014, S. 29ff)

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class MeineJSFBean {

	private String meinName;

	public String getMeinName() {
		return meinName;
	}

	public void setMeinName(String meinName) {
		this.meinName = meinName;
	}

	public String willkommen() {
		return "Hallo " + meinName;
	}
}
```

***Listing 8:** Eine einfache Java-Bean*

Etwas ungewöhnlich erscheinen noch die zwei import-Anweisungen, sowie die zwei Annotationen vor dem Beginn der Klasse. Diese sind jedoch notwendig, um die Java-Bean aus einer View heraus verwenden zu können. Man spricht auch von Registrierung oder Deklarierung einer Java-

Bean. Seit JSF 2.0 gibt es hierfür zwei Möglichkeiten. Listing 8 zeigt die Deklarierung der Klasse mit dem Namen „MeineJSFBean“ über Annotationen. Die andere Variante wäre über Konfiguration der Java-Bean in der Datei faces-config.xml.
Die in diesem Beispiel verwendete Annotation „`@ManagedBean`“ bewirkt, dass die Managed-Bean den Klassennamen erhält, jedoch beginnend mit einem kleinen Buchstaben. Also „meineJSFBean“.

Durch „`@SessionScoped`“ wird der Gültigkeitsbereich der Bean definiert. Tabelle 1 zeigt eine Auflistung dieser sogenannten Scopes und ihrer individuellen Lebensdauer.

| Name | Annotation | Eigenschaft |
|---|---|---|
| None-Scope | `@NoneScoped` | Die Managed-Bean wird bei jedem Aufruf neu erstellt. |
| Request-Scope | `@RequestScoped` | Die Managed-Bean lebt für die Zeitdauer einer HTTP-Anfrage. |
| View-Scope | `@ViewScoped` | Die Lebensdauer der Managed-Bean ist an die Ansicht geknüpft, in der sie verwendet wird. |
| Session-Scope | `@SessionScoped` | Die Managed-Bean lebt für die Dauer einer Sitzung, in der der Benutzer mit der Anwendung verbunden ist. |
| Application-Scope | `@ApplicationScoped` | Für die gesamte Lebensdauer der Anwendung ist nur eine für alle Benutzer gleiche Instanz dieser Managed-Bean vorhanden. |

***Tabelle 1:*** *Gültigkeitsbereiche*

Die Managed Bean Creation Facility ist für die eben vorgestellten Beans die zentrale Behandlungsstelle. Die Managed Bean Creation Facility hat folgende Aufgaben:

- Deklaration sämtlicher Managed-Beans
- Festlegung der Lebensdauer der Managed-Beans
- Automatische Erzeugung, Initialisierung, Verwendung und Löschung der Manged-Bean-Instanzen
- Bereitstellung der Managed-Beans über die Unified Expression Language

Beim ersten Zugriff auf eine Bean, erfolgt eine automatische Instanziierung über die Managed Bean Creation Facility. Ist eine Instanz der Bean bereits vorhanden, erfolgt keine Instanziierung, sondern die existierende Bean wird wieder zurückgegeben. Nach der Bean-Erzeugung werden alle Managed-Properties initialisiert und im Anschluss wird die Managed-Bean unter der spezifizierten Lebensdauer gespeichert. (ebd. 2014, S. 31ff)

Wie schon bei der Konfiguration der Bean selbst, gibt es auch für die Initialisierung der Managed-Properties (Eigenschaften) zwei Umsetzungsmöglichkeiten. Um eine einheitliche Struktur beizubehalten wird hier nur die Variante über Annotationen betrachtet.
Über „@ManagedProperty“ können Datenfelder annotiert und mit einem Initialwert belegt werden.

```
@ManagedProperty(value = “3“)
private int loginVersuche;
@ManagedProperty(value = “#{andereBean.attribut}“)
private AndereBean andereBean;
```

***Listing 9:** Beispiele Managed-Prperties*

Listing 9 zeigt ein Beispiel der Zuweisung von Eigenschaften. In diesem Beispiel wird dem Datenfeld „loginVersuche“ über das Attribut „value“ direkt ein Wert von drei zugewiesen, während das Datenfeld „andereBean“ verdeutlichen soll, auch problemlos andere Beans in Datenfeldern referenziert werden können.

Die Managed Bean Creation Facility ist die von JSF zu Verfügung gestellte Möglichkeit zur Verwaltung von Beans. In Kapitel 9 erfolgt noch ein Ausblick auf die Verbindung von JSF und CDI. CDI ist eine weitaus mächtigere und komfortablere Technologie zur Verwaltung von Manged-Beans.

## 6.3 Unified Expression Language

Die Unified Expression Language ist das dynamische Verbindungsglied zu den Daten der Managed-Beans. Durch die Unified EL ist es möglich, Daten aus den Managed-Beans zu lesen oder auch Daten ins Modell zurückzuschreiben. Besonders gebräuchlich in JSF Applikationen sind Value-Expressions und die Method-Expressions, die im Folgenden genauer betrachtet werden.

### 6.3.1 Value-Expressions

ValueExpressions binden das Komponentenattribut an eine bestimmte Managed-Bean. Beim Rendern einer Seite wird der Wert des Attributes über die getter-Methode gelesen und in der Aktualisierungsphase wird über die setter-Methode der Wert des Attributes in die Bean zurückgeschrieben.

Syntax: `value = “#{klasse.datenfeldname}“`

Neben der Benutzung von Datenfeldern lassen sich in der Unified EL auch arithmetische Operatoren und Vergleichsoperatoren einbinden, womit äußerst nützliche Befehle entstehen können. Tabelle 2 zeigt diverse Beispiele der Value-Expressions in Verbindung mit Operatoren.

| Befehl | Beschreibung |
|---|---|
| `value="Hallo + #{meineBean.meinName}"` | Eine Kombination von Zeichenketten. |
| `value="33 * #{meineBean.meineZahl}"` | Eine Multiplikation einer Eigenschaft mit einem Wert. |
| `style="#{grid.displayed ? 'display: inline' : 'display: none';}"` | Eine Variante um Komponenten für gewisse Attribute ein- und auszublenden. |
| `rendered="#{meineBean.meinName != null}"` | Die Komponente wird nicht gerendert, wenn sie den Wert null hat. |

***Tabelle** 2: Beispiele Value-Expressions*

## 6.3.2 Method-Expressions

Mithilfe von Method-Expressions werden Komponenten an Methoden einer Managed-Bean verknüpft. Seit Java EE 6 ist es auch möglich die Methoden mit Parametern aufzurufen. Dies war zuvor nicht, beziehungsweise nur eingeschränkt über Umwege mit statischen Methoden möglich. (vgl. Kurz et al. 2014, S. 38 - 41)

Syntax: `value = "#{klasse.methodenname}"`

Tabelle 3 zeigt noch einige nützliche Method-Expressions.

| Befehl | Beschreibung |
|---|---|
| `value="#{meineBean.meineListe. size()}"` | Auslesen der Anzahl an Elementen des Attributes „meineListe". |
| `value="#{meineBean.getMeinName()}"` | Aufrufen der getter-Methode. Im Gegensatz zu value="{meineBean.meinName}" kann der Wert mittels der getter-Methode nur gelesen werden. |
| `value="#{meineBean.meinString. Replace('.', ',')}"` | Alle Punkte im Attribut „meinString" werden durch Kommas ersetzt. |

***Tabelle** 3: Beispiele Method-Expressions*

## 6.4 Konvertierung

Dem Konverter in JSF steht eine wichtige Rolle zu, da er für die Umwandlung der Werte zwischen dem Webclient und dem Server zuständig ist. Dem Webclient werden die Werte in Form von Strings angezeigt, während diese serverseitig als Java-Typen dargestellt werden. Konverter wandeln also beim Absenden eines Formulars, Strings in Java-Typen um und beim Rendern wieder den Java-Datentyp (z.B. Integer) in einen String.

Dabei unterscheidet man in JSF zwischen Standard- und benutzerdefinierten Konvertern. Sofern kein benutzerdefinierter Konverter erstellt wurde, benutzt JSF automatisch den Standardkonverter (= implizite Konvertierung), den es für viele einfache Java-Datentypen in beidseitiger Konvertierungsrichtung gibt. Listing 10 zeigt das Interface, dass jeder Konverter implementieren muss (vgl. Kurz et al. 2014, S. 77f).

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.ConverterException;

public interface Converter
{
      Object getAsObject(FacesContext context, UIComponent component,
            String value) throws ConverterException;
      String getAsString(FacesContext context, UIComponent component,
            Object value) throws ConverterException;
}
```

***Listing 10:*** *Das Converter-Interface*

### 6.4.1 Standardkonverter

Die wichtigsten Standardkonverter in der Praxis sind die Tags f:convertDateTime und `<f:convertNumber>`, da Datumsangaben und Zahlenformate länderspezifisch sind und somit nicht von JSF vollständig standardisiert werden können.
Aufgrund des Umfangs wird folgend nur das Tag `<f:convertNumber>` genauer betrachtet.

| Attribut | Beschreibung |
|---|---|
| type | Wahl des Formattyps [number (=default), currency, percentage] |
| currencyCode | Einstellung der Währung über den ISO 4217 Code [z.B. Euro = EUR] |
| currencySymbol | Einstellung der Währung des Symbols [z.B. Euro = €] |
| groupingUsed | Verwendung von Seperatoren [true (=default), false] |
| locale | Angabe der Lokalisierung [z.B. de_DE] |
| integerOnly | Nur Integer-Datentypen werden geparst [true, false] |
| maxFractionDigits | Maximale Anzahl an Nachkommastellen |
| minFractionDigits | Minimale Anzahl an Nachkommastellen |
| maxIntegerDigits | Maximale Anzahl an Vorkommastellen |
| minIntegerDigits | Minimale Anzahl an Vorkommastellen |
| pattern | Festlegung eines Musters [z.B. #.###,## (= deutsches Zahlenformat] |

***Tabelle 4:*** *Attribute von f:convertNumber*

Der folgende Code in Listing 11 zeigt eine kleine Beispielanwendung des convertNumber-Tags in Form einer expliziten Konvertierung.

Eine vorherige Eingabe der Nummer „99.9051“ würde mit entsprechend existierendem Datenfeld „number“ in der angegebenen JavaBean und den unten aufgeführten Konvertierungseigenschaften zu folgender Ausgabe führen: „99,91 €“

```
<h:outputText value="#{IrgendeineBean.number}" >
      <f:convertNumber groupingUsed="true"
type="currency"/>
</h:outputText>
```

***Listing 11:*** *Beispiel von f:convertNumber*

### 6.4.2 Benutzerdefinierte Konverter

Neben Standardkonvertern bietet JSF auch die Möglichkeit diese Standardkonverter zu verändern oder sogar neue Konverter für andere Java-Datentypen zu entwickeln.

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;

@FacesConverter("MeinKonverter")
public class MeinKonverter implements Converter
{
	public Object getAsObject(FacesContext context, UIComponent component,
		String value) throws ConverterException
	{
		if(value == null) return null;
		return value.replaceAll("\\s+", "_");
	}
	public String getAsString(FacesContext context, UIComponent component,
		Object value) throws ConverterException
	{
		if(value == null) return null;
		return value.toString();
	}
}
```

***Listing** 12: Ein benutzerdefinierter Konverter*

Listing 12 zeigt eine konkrete Klasse des benutzerdefinierten Konverters, der das bereits vorgestellte Converter-Interface mit den beiden Methoden getAsObject() und getAsString() implementiert. Eine kleine Modifizierung findet in der Methode getAsObject() statt. Es werden alle vom Benutzer eingegebenen Leerzeichen durch Unterstriche ersetzt.

Seit JSF 2.0 gibt es für die Registrierung und Einbindung eines Konverters mehrere Möglichkeiten. Vor JSF 2.0 war die Registrierung nur über die faces-config.xml-Datei möglich. In der neuen Version kann die Registrierung über die Annotation „`@FacesConverter`“ erfolgen. Verwendet wird der Konverter über das Tag `<f:converter>` mit Angabe der vorgegebenen converterId, siehe Listing 13.

```
<h:outputLabel for="inputName" value="Name:"/>
<h:inputText id="inputName" value="#{meineBean.meinString}">
	<f:converter converterId="meinKonverter" />
</h:inputText>
```

***Listing** 13: Verwendung des benutzerdefinierten Konverters*

Es gäbe auch noch die Möglichkeit den Konverter innerhalb der h:inputText-Komponente mittels dem converter-Attribut und entsprechender Method-Expression einzubinden. Doch diese Variante wird nicht näher erläutert.

## 6.5 Validierung

Jede Webanwendung die Benutzereingaben entgegen nimmt, sollte defensiv programmiert werden. Das bedeutet, dass fehlerhafte Benutzereingaben rechtzeitig abgefangen und verwaltet werden sollten. Dadurch beugt man nicht nur Systemabstürze vor, sondern gewährleistet auch, dass die entgegengenommen Benutzerdaten nicht fehlerhaft sind.

JSF bietet für die Validierung von Daten mehrere Varianten an. Seit JSF 2.0 werden zusätzlich sogenannte Bean-Validatoren zur Verfügung gestellt, die sich besonderer Beliebtheit erfreuen und auch Inhalt des folgenden Abschnitts sind.

### 6.5.1 Bean Validation nach JSR-303

Die Bean-Validation sei ein großer Schritt vorwärts, führen die Autoren Kurz und Marinschek aus, da bisherige Validierungsmöglichkeiten zu Redundanzen, höheren Programmieraufwand und erhöhter Fehleranfälligkeit geführt hätten.

Listing 14 zeigt die Integration einer einfachen Bean-Validation.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;
import javax.validation.constraints.Size;

@ManagedBean
@SessionScoped
public class MeinValidator {

	@NotNull @Size(min = 1, max = 10)
	private String meinName;
	@NotNull @Min(value = 0) @Max(value = 120)
	private Integer meinAlter;
	…
	…
```

***Listing 14:*** *Ausschnitt einer Bean-Validation (Getter- und Setter-Methoden ausgelassen)*

Diese Bean sorgt dafür, dass das String-Attribut „meinName“ mindestens einen und höchstens 10 Buchstaben enthält, sowie das Integer-Attribut „meinAlter“ keinen Wert unter 0 oder über 120 annehmen kann. Möglich wird diese Validierung durch die Benutzung von Annotationen, die über den Datenfeldern definiert wurden.

Damit die Annotationen vom Compiler erkannt und eingesetzt werden können, müssen entsprechende Klassen aus dem Paket javax.validation.costraints importiert werden (siehe Listing 14).

```
<context-param>
      <param-name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
      </param-name>
      <param-value>true</param-value>
</context-param>
```

***Listing 15:*** *Konfiguration der web.xml für die Annotation @NotNull*

Standardmäßig ignoriert JSF fehlende Benutzereingaben in den Eingabekomponenten. Damit JSF Nullwerte interpretieren kann, müssen die in Listing 15 gezeigten Kontextparameter in der web.xml definiert werden.

JSF bietet auch die Möglichkeit eigene Validatoren und Constraints zu erstellen, die dann wie im oben aufgeführtem Beispiel, über die Annotations einsetzbar wären. Dieses Thema soll aber nicht mehr Inhalt dieser Arbeit sein.

Tabelle 5 soll das Thema Bean-Validierung nach JSR-303 abschließen und einen Ausblick auf weitere vordefinierte Bean-Validations, die in dem Beispiel nicht verwendet wurden, bieten.

| Annotations | Eigenschaft |
|---|---|
| @AssertFalse / @AssertTrue | Das Element muss false/true sein. |
| @DecimalMax / @DecimalMin | Der value darf nicht über/unter dem Wert sein. |
| @Digits / @Fraction | Die erlaubten Stellen vor/nach dem Komma. |
| @Future / @Past | Das Element muss ein Datum in der Zukunft/Vergangenheit sein. |
| @Pattern | Das Element muss dem angegebenen Ausdruck entsprechen. |

***Tabelle 5:*** *Beispiele Bean-Validations*

### 6.5.2 Standardvalidatoren

Neben der Bean-Validation gibt es noch die klassische JSF-Validierung. Bei dieser Validierungsart erfolgt die Werteprüfung mittels den Kind-Tags aus der Core-Tag-Library. Das folgende Listing 16 zeigt eine äquivalente Validierungsmöglichkeit zum vorherigen Kapitel, aber diesmal in Form einer JSF-Validierung.

```
<h:outputLabel for="inputName" value="Name:"/>
<h:inputText id="inputName" value="#{meineBean.meinName}">
      <f:validateRequired />
      <f:validateLength minimum="1" maximum="10"/>
</h:inputText>

<h:outputText value="Alter:"/>
<h:inputText value="#{meineBean.meinAlter}">
      <f:validateRequired />
      <f:validateLongRange minimum="0" maximum="120" />
</h:inputText>
```

***Listing 16:*** *JSF-Validierung*

Selbstverständlich bietet auch hier JSF wieder die Möglichkeit benutzerdefinierte Validatoren zu generieren. Dies ist besonders dann nützlich, wenn beispielsweise die Überprüfung der Standard-Validatoren nicht ausreicht und individuell angepasst werden muss. Um Validatoren selber zu erzeugen, muss eine entsprechende Bean geschrieben werden, deren Methoden die Kontrollen durchführen und ValidatorExceptions schmeißen, sobald die Kontrolle fehlgeschlagen ist. Eingebunden werden die benutzerdefinierten Validatoren in eine View über das Attribut „validator“ mittels Method-Expressions zu entsprechender Bean.

## 6.6 Navigation

Die Navigation zwischen den einzelnen Ansichten spielt bei Webanwendungen eine fundamentale Rolle. In der Regel navigieren Benutzer über Befehlskomponenten, wie beispielsweise Links oder Buttons. Für die Navigation werden also Komponenten benötigt, die das Absenden der aktuellen Seite an den Server veranlassen und somit die Abarbeitung des Lebenszyklus anstoßen, siehe Kapitel 6.7 Lebenszyklus. (vgl. Kurz et al. 2014, S. 51f)

Seit JSF 2.0 gibt es für die Navigation zwei gängige Varianten:

### 6.6.1 Implizite Navigation

Die implizite Navigation steht seit JSF 2.0 zur Verfügung und stellt eine äußerst einfache Möglichkeit dar, zwischen Ansichten zu navigieren. Über das action-Attribut kann durch Angabe der View-Id eine direkte Navigation erfolgen.

Das action-Attribut kann aber auch über eine Method-Expression zu einer Bean referenziert werden, die die aufzurufende View-Id als String zurückliefert. Beide Varianten sind im Kapitel 7 unter Befehlskomponenten zu finden.

Da bei der Navigation, zu einer neuen Seite gewechselt wird und diese dann direkt als HTTP-Response zurückgegeben wird, erscheint zwar beim Clienten die neue gewünschte Ansicht, jedoch verändert sich nicht der Pfad in der Adressleiste. Um dieses Problem zu umgehen, können redirects verwendet werden, durch die der Browser des Clienten aufgefordert wird, einen separaten HTTP-Request für die neue Seite abzusenden, wodurch sich die Adressleiste aktualisiert. Redirects können verwendet werden, wenn davon auszugehen ist, dass der Benutzer einen Bookmark auf diese Ansicht setzen könnte. Jedoch sollte bedacht werden, dass der zusätzliche Request die Anwendung verlangsamt (vgl. Urbanek, 2010, S. 99).

```
<h:commandButton id="btnAccept" value="Übernehmen"
      action="/zweiteSeite.xhtml?faces-redirect=true"/>
```

***Listing** 17: Befehlskomponente mit redirect-Funktion*

### 6.6.2 Regelbasierende Navigation

Vor JSF 2.0 konnten lediglich Navigationsregeln verwendet werden, die in der Datei faces-config.xml konfiguriert wurden. Listing 18 zeigt ein simples Beispiel der regelbasierenden Navigation.

Mit dem Element „`navigation-rule`“ kann eine beliebige Anzahl an Navigationsregeln festgelegt werden. Das Element „`from-view-id`“ definiert von welcher Seite aus das folgende „`navigation-case`“ gültig ist. Die Zeichenkette „`ok`“ im Filterelement „`from-outcome`“ definiert den erwarteten Rückgabewert einer Action-Methode. Das Element „`to-view-id`“ definiert abschließend die Zielseite. (vgl. Kurz et al. 2014, S. 51f)

```
<navigation-rule>
      <from-view-id>/ersteSeite.xhtml</from-view-id>
      <navigation-case>
            <from-outcome>ok</from-outcome>
            <to-view-id>/zweiteSeite.xhtml</to-view-id>
      </navigation-case>
</navigation-rule>
```

***Listing 18:** Regelbasierende Navigation*

Die hier im Beispiel definierte Navigation bezieht sich explizit nur auf die View `erstSeite.xhtml`. Es lassen sich jedoch noch globale Navigationsregeln erstellen, die für alle Seiten gelten. Seit JSF 2.0 lassen sich auch „`if`“-Tags einbauen, um eine bedingte Abfrage in der Navigation einzubauen. (ebd. 2014, S. 53ff)

## 6.7 Lebenszyklus

Der Lebenszyklus einer Webseite wird immer dann in Gang gesetzt, wenn der Benutzer die aktuelle Seite absendet oder das System selbst ein Event auslöst. Der Lebenszyklus gliedert sich in sechs Phasen, von der jede unterschiedliche Aufgaben verfolgt. Das Besondere an dem Lebenszyklus in JSF ist, dass den Programmierern viele Möglichkeiten gegeben werden, diese zu beeinflussen und zu verändern.

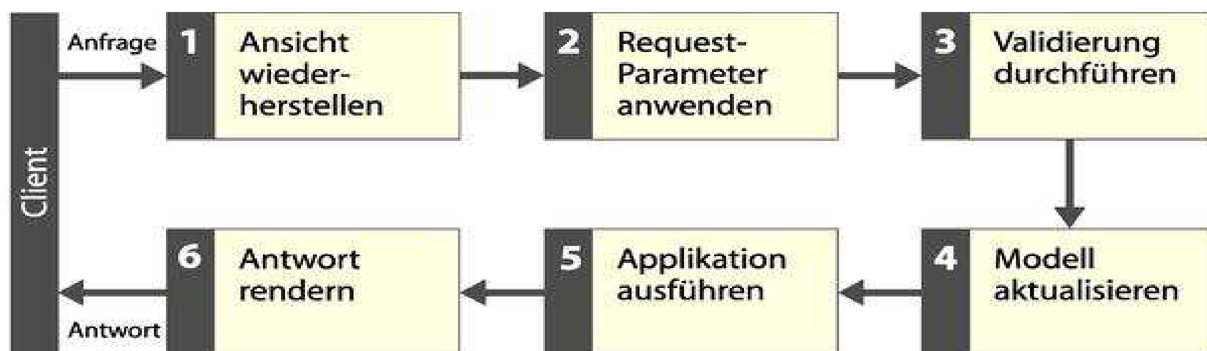


***Abbildung 5:** JSF - Lebenszyklus*

**Phase 1 (Ansicht wiederherstellen; Englisch: Restore-View):**

Diese Phase beinhaltet den Aufbau des Komponentenbaums (=View) und allen seinen Komponenten. Beim ersten Aufruf einer Seite wird ein View-Objekt erzeugt und im FacesKontext reingehängt. Da es sich beim ersten Aufruf einer Seite um einen leeren Komponentenbaum handelt, sind keine Werte zu verarbeiten, Validierungen durchzuführen und Konverter zu verknüpfen. Es kann somit von Phase 1 direkt in Phase 6 gesprungen werden. Wird die Seite zum wiederholten Male aufgerufen, so wird der bereits existierende Komponentenbaum in der

FacesKontext geladen und mit allen Validatoren, Konvertern und Listenern verknüpft, was zur Folge hat, dass der Lebenszyklus die Phasen normal durchläuft.

**Phase 2 (Request-Parameter anwenden; Englisch: Apply Request Values):**

In dieser Phase wird der Komponentenbaum mit allen Komponenten aktualisiert. Das bedeutet, dass alle vom Benutzer eingetragenen Werte den einzelnen Komponenten zugewiesen werden und Aktionskomponenten, wie beispielsweise Buttons, überprüfen, ob sie ausgelöst wurden.
Von besonderer Bedeutung in Phase 2 ist das Setzen des immediate-Attributes in einer Komponente. Dadurch wird nämlich bewirkt, dass die Konvertierung und Validierung aus der nachfolgenden Phase vorgezogen und somit bereits in Phase 2 ausgeführt werden.

**Phase 3 (Validierung durchführen; Englisch: Process Validations):**

Phase 3 ist besonders für die Eingabekomponenten von Bedeutung, da die vom Benutzer eingegebenen Werte nun konvertiert und validiert werden. Sowohl die Konvertierung als auch die Validierung wurden bereits in vorherigen Kapiteln genauer beschrieben.
Hervorzuheben in dieser Phase ist noch die Situation einer fehlgeschlagenen Validierung. In diesem Fall werden entsprechende Fehlermeldungen generiert und die aktuelle Seite als Antwort gerendert. Was bedeutet, dass von Phase 3 direkt auf Phase 6 gesprungen wird.

**Phase 4 (Modell aktualisieren; Englisch: Update Model Values):**

Hier werden die überprüften Werte (Submitted-Values) in das Modell übernommen.

**Phase 5 (Applikationen ausführen; Englisch: Invoke Application):**

Nun werden alle Ereignisse bearbeitet und es wird ermittelt welche Ansicht als nächstes aufgerufen wird. Der Komponentenbaum dieser Ansicht wird erstellt und in der FacesKontext abgelegt.

**Phase 6 (Antwort rendern; Englisch: Render Response):**

In der letzten Phase wird der Komponentenbaum gerendert und dem Benutzer als Antwort geschickt. JSF speichert diesen Komponentenbaum ab, für den Fall, dass weitere Anfragen auf diese Ansicht eintreffen.

## 6.8 Ereignisse

Ereignisse sind wesentliche Bestandteile einer Webapplikation. Jeder Benutzer erwartet nämlich, dass bestimmte Aktionen ausgeführt werden, wenn er beispielsweise einen Button betätigt. In JSF werden solche Ereignisse auch Events genannt und werden serverseitig von Event-Listenern behandelt. Ab der Phase 2 des Lebenszyklus können solche Events generiert und in eine Event-Queue eingefügt werden. Nach Abarbeitung der jeweiligen Phase, werden für die Events in der Queue, die registrierten Events aufgerufen. (vgl. Kurz et al. S. 55f)

In JSF gibt es mehrere Arten von Events mit entsprechenden Event-Listenern, die durch Benutzeraktionen oder vom System ausgelöst werden. Folgend werden alle Arten aufgelistet und grob beschrieben.

**Value-Change-Events**

Diese Events werden von Eingabekomponenten ausgelöst, sofern der Benutzer den Inhalt geändert hat und das Formular absendet. Bleibt der Wert in der Eingabekomponente unverändert, wird kein Event ausgelöst. Event-Listener für Value-Change-Events können auf zwei Arten registriert werden. Entweder kann die Registrierung in der Komponente über Method-Expressions im Attribut „`valueChangeListener`“ oder über das Kindelement aus der Core-Tag-Library „`f:valueChangeListener`“ erfolgen. Bei nicht gesetztem immediate-Attribut werden Value-Change-Events in der Regel in Phase 5 ausgeführt. (ebd. S. 57ff)

**Action-Events**

Action-Events werden von Befehlskomponenten ausgelöst. Das Auslösen einer Befehlskomponente bewirkt das Absenden der aktuell angezeigten Seite und somit den Anstoß des Lebenszyklus.

Wie Kurz und Marinschek ausführen, sei der bei Abarbeitung von Action-Events, zwischen Actions und Action-Listenern zu unterscheiden. Beide nutze man prinzipiell um Action-Events zu bearbeiten. Aber Action-Listener werden immer kurz vor Actions aufgerufen.

Während der Einsatz von Action-Listener eher für die UI-zentrierte Logik gedacht sei, wären für die Aufrufe der Geschäftslogik Actions ideal (ebd. S. 59)

Die Einbindung von Action-Events kann ähnlich der Value-Events entweder über das Komponentenattribut „`actionListener`“ oder über das Kindelement „`f:actionListener`“ erfolgen.

**System-Events**

Diese Art von Events stehen erst seit JSF 2.0 zur Verfügung und können zu einem bestimmten Zeitpunkt vom System ausgelöst werden. Grundsätzlich unterscheidet man zwischen System-Events, die beim Ausführen des Lebenszyklus ausgelöst werden und den System-Events, die beim Ausführen der Anwendung unabhängig von einer Komponente ausgelöst werden. Die Einbindung für die komponentenabhängigen System-Events kann über das Kindelement „f:event“ erfolgen. Über das Attribut „type“ muss der Name der Event-Klasse angegeben werden und im Attribut „listener“ eine Method-Expression für die Listener-Methode. Komponentenunabhängige System-Events lassen sich nicht über „f:event“ registrieren, sondern müssen als eigene Klassen umgesetzt werden. (ebd. S. 63ff)

**Phase-Events**

Diese Events werden routinemäßig bei der Ausführung des Lebenszyklus vor und nach jeder Phase ausgelöst. Die Registrierung von Listenern für Phase-Events kann über Konfiguration der Datei `faces-config.xml` durchgeführt werden oder über das Kindelement `<f:phaseListener>`.

# 7 JSF Standard-Komponenten

Für die Erstellung von Benutzeroberflächen bietet JSF zwei umfangreiche Standard-Bibliotheken an. Das grundlegende Verhalten einer Komponente wird in der Klasse javax.faces.component.UIComponentBase definiert, die von allen JSF-UI-Komponentenklassen erweitert wird.

HTML-Custom-Tag-Library und Core-Tag-Library heißen die zwei Standard-Tag-Bibliotheken. Um die Bibliotheken verwenden zu können, müssen sie über ihre Namensräume im Wurzelelement einer View eingebunden werden und sind dann unter dem jeweiligen Präfix verfügbar.
Wie Kurz und Marinschek in ihrem Buch ausführen, sind mit der JSF-Version 2.2 die Namensräume dieser Bibliotheken geändert worden. (siehe Tabelle 6)

| Name | Namensraum | Präfix |
|---|---|---|
| HTML-Custom-Tag-Library | http://xmlns.jcp.org/jsf/html (ab Version 2.2) | h |
| | http://java.sun.com/jsf/html (vor Version 2.2) | h |
| Core-Tag-Library | http://xmlns.jcp.org/jsf/core (ab Version 2.2) | f |
| | http://java.sun.com/jsf/core (vor Version 2.2) | f |

***Tabelle** 6: Namensräume der Standard-Tag-Bibliotheken in JSF 2.2*

Weiterhin empfehlen die beiden Autoren für eine optimale Zukunftssicherheit nur die aktuellen Namensräume zu verwenden, obwohl auch noch die alten Namensräume unterstützt werden. (Kurz et al. 2014, S. 109)

## 7.1 Core-Tag-Library

Die Tags dieser Bibliothek sind für die Basisfunktionalitäten verantwortlich. Einige von diesen, wie beispielsweise „`f:convertNumber`“, kamen bereits im Rahmen dieser Arbeit vor. Da mit den Tags der Core-Tag-Library nur Funktionalität geliefert wird, werden diese oft in Kombinationen beziehungsweise in Tags der HTMLCostum-Tag-Library verschachtelt angewendet. Tabelle 7 zeigt einen Ausblick auf bisher bekannte und neue JSF-Core-Tags.

| Tag | Beschreibung |
|---|---|
| `f:actionListener` | Hinzufügen eines Action-Listener zu einer Komponente. |
| `f:ajax` | Hinzufügen von Ajaxfunktionalitäten. |
| `f:attribute`, `f:attributes` | Hinzufügen eines Komponentenattributes. |
| `f:convertDateTime` | Hinzufügen eines DateTime-Konverters. |
| `f:converter` | Hinzufügen eines frei wählbaren Konverters. |
| `f:convertNumber` | Hinzufügen eines Nummern-Konverters. |
| `f:event` | Hinzufügen eines System-Events. |
| `f:facet` | Hinzufügen von speziellen Eigenschaften eines Kindelements. |
| `f:validator` | Hinzufügen eines benutzerdefinierten Validators. |
| `f:valueChangeListener` | Hinzufügen eines Value-ChangeListener zu einer Komp. |

***Tabelle** 7: Beispiele Core-Tag-Library*

Die Tabelle zeigt nur einen kleinen Teil von Tags der Core-Tag-Library.
Mit JSF 2.2 kam das Tag `<f:resetValues>` hinzu, welches ein bereits länger bekanntes Problem elegant löst.

Die Autoren Kurz und Marinschek beschreiben, dass es in einigen Fällen nicht möglich gewesen wäre, Eingabefelder zu aktualisieren, wenn der Wert zuvor nicht zurückgesetzt worden wäre. Das Tag `<f:resetValues>` behebt dieses Problem, in dem es Action-Listener zu Befehlskomponenten hinzufügt, die alle Eingabekomponenten zurücksetzen, deren Client-IDs im Attribut „`render`“ angegeben sind. (vgl. Kurz et al. 2014, S. 113)

## 7.2 HTML-Custom-Tag-Library

Die HTML-Custom-Tag-Library bietet die Tags für die Standard-JSF-Komponenten und ihre Darstellung als HTML-Ausgabe. Unter dem Präfix h sind diese Tags verfügbar.

### 7.2.1 Befehlskomponenten

Befehlskomponenten werden zum Auslösen von Aktionen verwendet. Typische Benutzeraktionen sind beispielsweise das Betätigen eines Buttons oder eines Links. Die dahinterliegenden Mechanismen werden von den Komponenten h:commandButton und h:commandLink zur Verfügung gestellt und sind in Listing 19 beispielhaft aufgeführt.

```
<h:commandButton id="button" value="Alle Filme ab 18"
action="/filmeAb18.xhtml"/>

<h:commandLink id="link" rendered="#{meineBean.Alter >= 18}"
      action="#{meineBean.zeigeFilmeAb18}" >
      <h:outputText value="Alle Filme ab 18"/>
</h:commandLink>
```

***Listing 19:*** *Befehlskomponenten*

Im ersten Beispiel wird lediglich ein beschrifteter Button erstellt, der einen bei Betätigung zu einer anderen Seite weiterleitet.
Etwas komplexer ist das Beispiel für den Link. Durch eine Abfrage im rendered-Attribut wird dafür gesorgt, dass der Link nur den Benutzern angezeigt wird, die mindestens 18 Jahre alt sind. Bei Betätigung des Links wird über eine Method-Expression eine Referenz zu angegebener Bean geschaffen und entsprechend weitergeleitet.

### 7.2.2 Ausgabekomponenten

Ausgabekomponenten verwalten die Ausgabe von Text und Bildern. Listing 20 in Verbindung mit Abbildung 6 zeigen vier verschiedene Typen von Ausgabekomponenten.

```
<body>
      <h1><h:outputText value="Beispiele Standardkomponenten"/></h1>
      <h2><h:outputText value="Ausgabekomponenten"/></h2>
      <h:form id="form">
      <h:panelGrid id="gridOutput" columns="1">

            <h:outputText value="Ich bin ein &lt;em>h:outputText&lt;/em>"
                  escape="false"/>

            <h:outputLabel value="Ich bin ein h:outputLabel"
                  style="font-weight:bold"/>

            <h:outputLink value="http://google.de">
                  <h:graphicImage id="graphic" url="/images/Beispielbild.png"/>
            </h:outputLink>

            <h:outputFormat value="{0} bin ein {1}">
                  <f:param value="Ich"/>
                  <f:param value="h:outputFormat"/>
            </h:outputFormat>

      </h:panelGrid>
      </h:form>
</body>
```

***Listing 20:*** *Ausgabekomponenten*



***Abbildung 6:*** *Ausgabekomponenten*

**<h:outputText>**
Diese Komponente ist zum Ausgeben von Texten in HTML-Form gedacht. Über das Attribut „`value`“ wird der auszugebende Text eingebunden.
Kurz und Marinschek zeigen mittels der Setzung des escape-Attributes auf false (siehe Listing 20) noch eine Möglichkeit direkte Textformatierungen durchzuführen. Ohne dieses Attribut käme es zu folgender Ausgabe: Ich bin ein<em>h:outputText</em>
(Kurz et al. 2014, S120f)

**<h:outputLabel>**
Die beiden Autoren beschreiben, dass mit dieser Komponente Bezeichnungen mit Eingabefeldern verbunden werden können. Durch Setzen eines for-Attributes, welches auf eine ID eines Eingabefeldes zeigt, wäre dies möglich. (ebd. 2014, S. 121)
Listing 20 zeigt eine Möglichkeit, wie über das value-Attribut, die „h:outputLabel“-Komponenten ähnlich wie „h:outputText“-Komponenten verwendet werden können. Zusätzlich wird hier noch das style-Attribut verwendet, um die Schrift in fett zu formatieren.

**<h:outputLink>**
Ferner zeigt Listing 20 eine Verschachtelung der Ausgabekomponenten <h:graphicImage> und `<h:outputLink>`. Über das value-Attribut in der Link-Komponente wird das Ziel der Weiterleitung definiert. Über das url-Attribut in der Bildkomponente kann der Pfad des Bildes angeben werden. Kurz und Marinschek beschreiben in ihrem Werk noch eine Möglichkeit der Bildereinbindung über die JSF-Resourcenverwaltung. Auf diese wird aber nicht näher eingegangen.

**<h:outputFormat>**
Mit dieser Komponente ist die Einbindung von Text möglich. Die Einbindung kann entweder statisch vorgegeben (siehe Listing 20) oder dynamisch über Unified-EL-Ausdrücke eingebunden werden. Mittels Platzhaltern (die Zahlen in den geschwungenen Klammern) ist es möglich sich auf einen Index eines „f:param“-Kindelements zu verweisen und dessen Wert zu verwenden. (ebd. 2014, S. 123)

### 7.2.3 Ausblick auf weitere Komponenten

Die in den vorangegangenen Kapiteln dargestellten Komponenten, zeigen nur einen Bruchteil, von dem was die Standardbibliotheken zu bieten haben. Aufgrund des Umfangs können nicht alle Komponenten vorgestellt werden.

Tabelle 8 soll jedoch einen kleinen Ausblick auf weitere Komponenten geben, die mindestens genauso wichtig sind, wie die Befehls- und Ausgabekomponenten.

| Bezeichnung | Typ | Aufgabe |
|---|---|---|
| h:form | Formularkomponente | Unsichtbare Komponente, die alle Eingabe- und Befehlskomponenten umschließen muss. |
| h:dataTable | Datenkomponente | HTML-Ausgabe und Iteration über Elemente in einer Sammlung. |
| h:inputText | Eingabekomponente | Erzeugung eines Texteingabefeldes. |
| h:inputSecret | Eingabekomponente | Erzeugung eines Passworteingabefeldes. |
| h:inputTextarea | Eingabekomponente | Erzeugung eines mehrzeiligen Texteingabefeldes. |
| h:inputFile | Dateiuploadkomponente | Upload einer Datei (erst seit JSF 2.2) |
| h:selectBooleanCheckbox | Auswahlkomponente | Erzeugung einer booleschen Checkbox. |
| h:selectOneRadio | Auswahlkomponente | Darstellung von Objekten als Optionsfeld. |
| h:selectOneListbox | Auswahlkomponente | Darstellung von Objekten als Listenfeld. |
| h:selectOneMenu | Auswahlkomponente | Darstellung von Objekten im Klappmenü. |
| h:selectBooleanCheckbox | Auswahlkomponente | Erzeugung einer Gruppe von Checkboxen. |
| h:message | Nachrichtenkomponente | Darstellung von einer Nachricht. |
| h:messages | Nachrichtenkomponente | Darstellung von mehreren Nachrichten. |
| h:outputScript | Ressourcenkomponente | Ausgabe einer Skript-Ressource. |
| h:outputStylesheet | Ressourcenkomponente | Ausgabe einer Stylesheet-Ressource |

***Tabelle** 8: Standard-Komponenten*

# 8 PrimeFaces

In den vorherigen Kapiteln wurde bereits verdeutlicht welch immense Bedeutung die Komponentenorientierung in JSF hat. In Kapitel 7 wurden bereits zwei Standardbibliotheken von JSF vorgestellt.

Nun soll eine weitere Open-Source-Komponentenbibliothek namens PrimeFaces präsentiert werden, die seit der Veröffentlichung im Jahre 2010 eine besonders große Bedeutung in JSF erlangte.

## 8.1 Grundlegendes

PrimeFaces ist eine Komponentenbibliothek mit etwa 100 aufeinander abgestimmten Komponenten, die viele Funktionalitäten weit über den JSF-Standard hinaus anbietet, wie beispielsweise Komponenten für Diagramme oder Google-Maps. Das Erscheinungsbild der Komponenten kann darüber hinaus noch mit Themes angepasst werden.
(vgl. Kurz et al. 2014, S. 309)

Ziel der Entwickler von PrimeFaces war es eine leichtgewichtige und performante Bibliothek zu schaffen, um die Entwicklung von Webapplikationen zu vereinfachen.

Die Integration von PrimeFaces geschieht über eine Jar-Datei, die in das Projekt eingebunden werden muss. Bei der Verwendung von Maven müssen die in Listing 21 angegebenen Abhängigkeiten in die pom.xml eingetragen werden.

```
<repositories>
        <repository>
            <id>prime-repo</id>
            <name>PrimeFaces Maven Repository</name>

<url>http://repository.primefaces.org</url>
            <layout>default</layout>
        </repository>
    </repositories>

<dependencies>
      ...
      ...
      <dependency>
            <groupId>org.primefaces</groupId>
            <artifactId>primefaces</artifactId>
            <version>4.0</version>
        </dependency>
      ...
      ...
<dependencies>
```

***Listing 21:*** *Einbindung von PrimeFaces in ein Maven-Projekt*

## 8.2 p:menubar

Weil es kaum noch Webseiten ohne ein Menü gibt, handelt dieser Abschnitt von der Einrichtung eines Menüs über PrimeFaces-Komponenten. PrimeFaces stellt hierfür mehrere Menü-Varianten zur Verfügung. Folgend wird nur das Tag `<p:menubar>` mit allen notwendigen Kind-Elementen genauer betrachtet.

***Abbildung** 7: PrimeFaces - menubar*

```
<p:menubar>
	<p:submenu label="Datei" icon="ui-icon-document">
		<p:submenu label="Neu" icon="ui-icon-folder-open">
			<p:menuitem value="Dokument X" url="#" />
			<p:menuitem value="Dokument Y" url="#" />
			<p:menuitem value="Dokument Z" url="#" />
		</p:submenu>
		<p:separator />
		<p:menuitem value="Schließen" url="#" icon="ui-icon-close" />
	</p:submenu>
	<p:submenu label="Bearbeiten" icon="ui-icon-pencil">
		<p:menuitem value="Rückgängig" url="#"
			icon="ui-icon-arrowreturnthick-1-w" />
		<p:menuitem value="Wiederholen" url="#"
			icon="ui-icon-arrowreturnthick-1-e" />
	</p:submenu>
	<p:submenu label="Hilfe" icon="ui-icon-help">
		<p:menuitem value="Problem melden" url="#" />
		<p:menuitem value="Suche" icon="ui-icon-search" url="#"/>
	</p:submenu>
</p:menubar>
```

***Listing** 22: PrimeFaces – menubar*

Bei p:menubar handelt es sich um ein horizontal angeordnetes Menü, dass hierarchisch aufgebaut werden kann. Um diese Struktur zu erreichen, werden die Komponenten `<p:submenu>` verschachtelt zusammengesetzt. Die Beschriftung der Submenüs erfolgt im Attribut „`label`“. Die Komponenten `<p:menuitem>` repräsentieren die Menüeinträge, die über das value-Attribut beschriftet werden. Menüeinträge können über das Tag `<p:seperator>` in einem Submenü nochmals visuell durch einen Unterstrich voneinander abgegrenzt werden. In Listing 22 stellen die Menüeinträge nur Platzhalter dar (url="#").

Sehr benutzerfreundlich gestaltet sich die Verwendung der kleinen Menübilder. Diese müssen nicht erst in das JSF-Projekt eingebunden werden, sondern stehen direkt mit der Integration von PrimeFaces zur Verfügung. Leider nur sind die Bezeichnungen der Bilder nicht immer eindeutig. Doch im Internet findet man recht schnell viele bebilderte Auflistungen.

Das in Abbildung 7 und Listing 22 vorgestellte Menü blendet die Untermenüs erst ein, wenn mit der Maus darüber gefahren wird (mouseover). Bei der Umstellung auf ein Mausklick-Menü zeigt sich PrimeFaces besonders flexibel und komfortabel. Lediglich das Attribut „autoDisplay“ der Komponente `<p:menubar>` muss auf „false“ gestellt werden und schon erscheinen die Untermenüs erst nach einem Mausklick.

## 8.3 p:rating

Heutzutage werden Bewertungssysteme auf den Webseiten vieler großer Anbieter wie Facebook, Amazon oder dem Google-Store genutzt. Besonders populär sind Bewertungssysteme mit anklickbaren Sternen geworden. Diese können nicht nur zum Abgeben, sondern auch zum Anzeigen einer Bewertung verwendet werden.
PrimeFaces liefert mit `<p:rating>` eine sehr komfortable Schnittstelle für dieses Anwendungsgebiet.

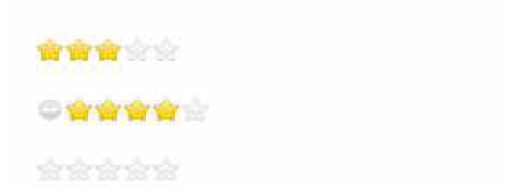


***Abbildung 8:*** *PrimeFaces - rating*

```
<p:rating value="#{meineBean.meinWert}" cancel="false" stars="5"/>
<p:rating value="#{meineBean.meinWert}" stars="5"/>
<p:rating value="#{meineBean.meinWert}" readonly="true"
stars="5"/>
```

***Listing 23:*** *PrimeFaces – rating*

Listing 23 zeigt drei verschiedene Einsatzmöglichkeiten des Bewertungssystems. Alle drei Varianten referenzieren ihre Werte über eine Bean und besitzen insgesamt fünf Sterne. Der ersten Version fehlt im Vergleich zur zweiten Version lediglich der cancel-Button. Die dritte Version zeigt eine Möglichkeit das Bewertungssystem nur als reine Ausgabekomponente darzustellen.

## 8.4 Themes

Eine weitere große Stärke von PrimeFaces ist die einfache Verwaltung der Themes. Mit Themes bezeichnet mein das Erscheinungsbild von Komponenten und Zeichen.

PrimeFaces bietet momentan über 30 verschiedene Erscheinungsmuster an, die über http://www.primefaces.org/themes.html erhältlich sind. (vgl. Kurz et al, 2014. S. 321f)

Die Integration der Themes erfolgt entweder über Einbindung der jeweiligen Jar-Datei oder bei Verwendung von Maven über die Abhängigkeit in der pom.xml.

Listing 24 zeigt eine praktische Möglichkeit mit der gleich alle verfügbaren Themes über Maven integriert werden.

```
<dependency>
	<groupId>org.primefaces.themes</groupId>
	<artifactId>all-themes</artifactId>
	<version>1.0.10</version>
</dependency>
```

***Listing 24:*** *Einbindung der Themes*

Das aktuell verwendete Theme kann in der web.xml konfiguriert werden (siehe Listing 25).

```
<context-param>
	<param-name>primefaces.THEME</param-name>
	<param-value>trontastic</param-value>
</context-param>
```

***Listing 25:*** *Theme-Einstellung in der web.xml*

***Abbildung 9:*** *PrimeFaces - Theme trontonic*

Abbildung 9 zeigt die erstmals in Kapitel 7.2 vorgestellte Menüleiste mit der Einbindung des trontastic-Themes. Bei fehlender Angabe eines Themes verwendet PrimeFaces standardmäßig das aristo-Theme.

Mit den Themes von PrimeFaces lassen sich aber noch viel mehr Sachen machen als die hier vorgestellten. Aufgrund des breiten Umfangs, werden diese Features aber nicht detailliert vorgestellt, sondern sollen lediglich erwähnt werden, um zu verdeutlichen, wie praktisch die Anwendungsmöglichkeiten der Themes von PrimeFaces sind.
Die Autoren Kurz und Marinschek zeigen dazu wie mittels der Unified-EL und entsprechend geschriebener Java-Bean die Einbindung der Themes in die Webapplikation auf dynamische Weise erfolgen kann. Des Weiteren stellen sie dar, wie über das Online-Tool jQuery ThemeRoller, eigene Themes erstellt und in das Projekt eingebunden werden können. (ebd. 2014, S. 322f)

# 9 JSF und CDI

In Kapitel 4.1 wurde bereits präsentiert, wie mittels der Managed Bean Creation Facilitiy intern Beans von JSF verwaltet werden können. Doch mit Java EE 6 wurde mit Contexts and Dependency Injection for Java (CDI) ein neuer Standard eingeführt. Dieser ist weitaus mächtiger als die interne JSF-Variante.

Die Hauptaufgabe von CDI ist die Verwaltung von Beans und deren Lebenszyklus, sowie die Verknüpfung der Beans über die typensichere Dependency-Injection (vgl. Kurz et al. 2014, S. 289).

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;

@Named
@RequestScoped
public class MeineCDIBean {

	private String meinName;

	...(Getter- und Setter-Methoden ausgelassen)
}
```

***Listing 26:** CDI-Bean*

Listing 26 zeigt den Aufbau einer CDI-Bean, die sich im Wesentlichen nicht sonderlich anders als die bereits bekannten JSF-Beans implementieren lässt. Lediglich die Annotationen `@Named` und `@RequestScope` sind anders, wodurch auch andere Pakte importiert werden müssen. Die Benutzung der CDI-Bean erfolgt außerdem wie eine gewöhnliche JSF-Bean über Unified EL.

Wird JSF mit einem Applikationsserver wie Glassfish 4 oder JBoss AS7 verwendet, ist CDI bereits integriert und sofort einsatzbereit.
Läuft die Anwendung hingegen auf einem Servlet-Container wie Tomcat oder Jetty, muss eine CDI-Implementierung manuell integriert werden. (vgl. Kurz et al. 2014, S. 295)

Die Vorteile durch die Nutzung von CDI ergeben sich nicht nur durch die einfache Implementierung und Benutzung. CDI bietet mit sogenannten producer-Methoden eine weitere sehr flexible Variante, um Bean-Instanzen zu erzeugen. Des Weiteren gibt es mit dem Projekt Apache MyFaces Extensions CDI (kurz CODI) eine Erweiterung für CDI die viele nützliche Funktionen mit sich bringt.

## 9.1 Konversationen

Als Abschluss für diese Arbeit soll noch ein Problem angesprochen werden, dass über CODI auf einfache Art gelöst werden kann.

Oftmals beinhalten Webanwendung diverse Abläufe, die sich über mehrere Seiten erstrecken, aber aus Sichtweise des Benutzers, zu einer einzigen Einheit zusammengehören. Aus Sicht des Programms sind es meistens mehrere Anweisungen auf mehrere Seiten. Ein klassisches Beispiel dazu wäre der Warenkorb eines Webshops. Oft werden Produkte aus diversen Seiten zusammengesucht und zum Warenkorb gesendet, der dann zum Schluss die Basis für die Bestellung darstellt. Ein großes Problem liegt darin, dass die eine Seite nichts von einer anderen Seite weiß. Das hierfür der Einsatz von Managed-Beans gedacht ist, wurde bereits im Rahmen dieser Arbeit angesprochen und bearbeitet.

Ein weiteres Problem jedoch ist, welcher Gültigkeitsbereich gewählt werden soll. Die in Kapitel 6.2 vorgestellten Scopes sind nicht optimal für einen solchen Einsatz. Request- und View-Scopes wären zu kurzlebig, da nur Beans mit der Lebensdauer einer einzigen Anfrage beziehungsweise einer einzigen Ansicht erzeugt werden.

Der Gültigkeitsbereich der Session-Scope ist dagegen zu langlebig, da hier die Beans für die Dauer der ganzen Sitzung erzeugt werden. Bei Verwendung von CDI-Beans würde sich das Conversation-Scope anbieten, wovon allerdings die Autoren Kurz und Marinschek abraten, da es zu unpraktisch sei. Zu empfehlen seien jedoch die Scopes von CODI, da über CODI viele flexible Gültigkeitsbereiche definiert werden können (vgl. Kurz et al. 2014, S. 290).

Tabelle 9 zeigt abschließend, die von CODI angebotenen Gültigkeitsbereiche.

| Name | Annotation | Eigenschaft |
|---|---|---|
| Conversation-Scope | `@ConversationScoped` | Konversation mit manueller Lebensdauer. |
| View-Acsess-Scope | `@ViewAccessScoped` | Konversation mit automatischer Lebensdauer. |
| Window-Scope | `@WindowScoped` | Definition einer Art von Session pro Browserfenster/-tab. |

***Tabelle 9:*** *Gültigkeitsbereiche von CODI*

# 10 Fazit

So hoch die Euphorie noch vor Beginn dieser Hausarbeit war, endlich mal wieder mit Java zu programmieren, umso höher war die Frustration währenddessen. JSF ist wirklich ein sehr umfangreiches Framework und gehört nicht in die Hände von Anfängern. Besonders große Schwierigkeiten bereitete mir das Zusammensuchen und Integrieren aller notwendigen Programme, Server, Bibliotheken und wichtigen Tools, um meine JSF-Anwendung zum Laufen zu bringen. Da gibt es doch wesentlich freundlichere Einstiegsprogramme.

Doch nach diesem holprigen Einstieg und der recht flachen Lernkurve hat das Arbeiten mit JSF angefangen, mir immer mehr und mehr Freude zu bereiten. Wurden erst einmal die generellen Konzepte und Arbeitsweisen von JSF verstanden, war es wirklich ein Leichtes, seine eigenen Ideen auszuprobieren und umzusetzen.

JSF strotzt nur so vor Funktionen, Features und Konzepten, die man auf unterschiedlichste Art und Weise verwenden kann. Im Rahmen dieser Arbeit konnten zwar die meisten JSF-Konzepte vorgestellt werden, jedoch gehen diese jeweils noch deutlich tiefer ins Detail. Und das ist gerade eines der ganz besonderen Dinge in JSF, dass nämlich so ziemlich alles beeinflussbar ist und individuell angepasst werden kann. Braucht man spezifische Konverter oder Validatoren, dann erstellt man einfach selber welche. Braucht man auf seine Anwendung angepasste Komponenten, dann erstellt man einfach selber welche. Man kann sogar seine eigenen Annotationen erzeugen. Das Schönste daran ist, dass man aber eigentlich gar nichts selber erzeugen muss, da JSF standardmäßig schon so vieles anbietet und noch viel mehr auf einfache Art integriert werden kann. Besonders gefallen hat mit dir Komponentenbibliothek PrimeFaces. Diese Bibliothek ist optimal auf JSF angepasst und lässt sehr leicht verwenden. Außerdem bietet sie eine sehr viel schönere Darstellung von Standard-Komponenten und noch viele weitere nützliche Komponenten, die es standardmäßig nicht gibt. PrimeFaces nehmen dem Programmierer extrem viel Arbeit ab und sind somit von mir nur weiterzuempfehlen.

Mit JSF kann man aber auch problemlos andere Technologien wie beispielsweise Spring oder Ajax einbinden. Gerade in Verbindung mit Ajax gilt JSF momentan als eines der beliebteren Webframeworks.

Ein weiterer Pluspunkt von JSF ist die Beliebtheit im Unternehmensumfeld und die damit verbundene Zukunftssicherheit. Man spürt wie die Entwickler mit Aspekten, wie der gegebenen Kompatibilität zu JSP, einen Umstieg zu JSF fördern. Alle diese Dinge führen dann letzten Endes dazu, dass sich JSF immer weiter verbreitet, über die Jahre immer mehr Support erhält und dadurch noch weiter wächst.

Die im Rahmen dieser Arbeit verwendete JSF Version 2.2 hat verglichen mit der Version 2.1, meiner Meinung nach, keinen besonderen Schub erhalten. Sowohl die Neuerungen für die Version 2.2, als auch die schon wesentlich bedeutenderen Neuerungen von 1.2 auf 2.0, wurden in den vorangegangenen Kapiteln präsentiert und entsprechend vermerkt.

Zum Schluss bleibt mir nur noch übrig zu sagen, dass sich mit JSF wirklich sehr umfangreiche und komplexe Benutzeroberflächen erstellen lassen. Die Konzepte und die ganze JSF-Struktur erlauben es erst gar nicht, grobe Code-Vermischungen von Java und HTML zu produzieren, sodass mit JSF ein wirklich sehr altes langlebiges Problem endlich gelöst werden konnte.

# 11 Literaturverzeichnis

Michael Kurz, Martin Marinschek (2014). JavaServer Faces 2.2 – Grundlagen und erweiterte Konzepte, 3. Auflage. Heidelberg: dpunkt.verlag GmbH

Marcel Urbanek (2010). JavaServer Faces - *JSF verstehen und praktisch einsetzen*. Witten: W3L-Verlag. Herdecke.

Christen Ullenboom (2012). Java 7 – Mehr als eine Insel – Das Handbuch zu den Java SE-Bibliotheken (o.O.) Galileo Computing

# 12 Abbildungsverzeichnis

**Abbildung 1:** Model2-Prinzip........ 1
**Abbildung 2:** Benutzung eines Templates........ 1
**Abbildung 3:** Beispiel Komponentenbaum........ 1
**Abbildung 4:** Komponentenbaum - Browseranzeige........ 1
**Abbildung 5:** JSF - Lebenszyklus........ 1
**Abbildung 6:** Ausgabekomponenten........ 1
**Abbildung 7:** PrimeFaces - menubar........ 1
**Abbildung 8:** PrimeFaces - rating........ 1
**Abbildung 9:** PrimeFaces - Theme trontonic........ 1

Abb1. nach Michael Kurz, Martin Marinscheck (2014) JavaServer Faces 2.2 – Grundlagen und erweiterte Konzepte. http://jsfatwork.irian.at/grafiken/mvc-model2.jpg [Zugriff: 18.04.2014]

Abb2. nach Michael Kurz, Martin Marinscheck (2014) JavaServer Faces 2.2 – Grundlagen und erweiterte Konzepte. http://jsfatwork.irian.at/grafiken/lifecycle-complete.jpg [Zugriff: 20.04.2014]

# 13 Tabellenverzeichnis

**Tabelle 1:** Gültigkeitsbereiche........ 14
**Tabelle 2:** Beispiele Value-Expressions........ 16
**Tabelle 3:** Beispiele Method-Expressions........ 16
**Tabelle 4:** Attribute von f:convertNumber........ 18
**Tabelle 5:** Beispiele Bean-Validations........ 21
**Tabelle 6:** Namensräume der Standard-Tag-Bibliotheken in JSF 2.2........ 28
**Tabelle 7:** Beispiele Core-Tag-Library........ 28
**Tabelle 8:** Standard-Komponenten........ 32
**Tabelle 9:** Gültigkeitsbereiche von CODI........ 39

# 14 Listingverzeichnis

**Listing 1:** Einfaches Servlet - Behandlung der GET-Methode ........ 1
**Listing 2:** Ein einfaches JSP-Beispiel ........ 1
**Listing 3:** web.xml ........ 1
**Listing 4:** Beispiel eines Facelets ........ 1
**Listing 5:** Ein einfaches Template ........ 1
**Listing 6:** Ein einfacher Template-Client ........ 1
**Listing 7:** Beispiel Komponentenbaum (Ausschnitt) ........ 1
**Listing 8:** Eine einfache Java-Bean ........ 1
**Listing 9:** Beispiele Managed-Prperties ........ 1
**Listing 10:** Das Converter-Interface ........ 1
**Listing 11:** Beispiel von f:convertNumber ........ 1
**Listing 12:** Ein benutzerdefinierter Konverter ........ 1
**Listing 13:** Verwendung des benutzerdefinierten Konverters ........ 1
**Listing 14:** Ausschnitt einer Bean-Validation (Getter- und Setter-Methoden ausgelassen). 1
**Listing 15:** Konfiguration der web.xml für die Annotation @NotNull ........ 1
**Listing 16:** JSF-Validierung ........ 1
**Listing 17:** Befehlskomponente mit redirect-Funktion ........ 1
**Listing 18:** Regelbasierende Navigation ........ 1
**Listing 19:** Befehlskomponenten ........ 1
**Listing 20:** Ausgabekomponenten ........ 1
**Listing 21:** Einbindung von PrimeFaces in ein Maven-Projekt ........ 1
**Listing 22:** PrimeFaces – menubar ........ 1
**Listing 23:** PrimeFaces – rating ........ 1
**Listing 24:** Einbindung der Themes ........ 1
**Listing 25:** Theme-Einstellung in der web.xml ........ 1
**Listing 26:** CDI-Bean ........ 1

## Erklärung

„Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angegeben Hilfsmittel benutzt und die verwendete Literatur vollständig aufgeführt sowie Zitate kenntlich gemacht habe. Ich versichere ferner, dass die Arbeit noch nicht zu anderen Prüfungen vorgelegt wurde.“

______________________________________________

Ort, Datum und Unterschrift