Java Server Faces

PROGRAMMER'S CHOICE

Die Wahl für professionelle Programmierer und Softwareentwickler. Anerkannte Experten wie z. B. Bjarne Stroustrup, der Erfinder von C++, liefern umfassendes Fachwissen zu allen wichtigen Programmiersprachen und den neuesten Technologien, aber auch Tipps aus der Praxis.
Die Reihe von Profis für Profis!

Hier eine Auswahl:

Die C++ Programmiersprache

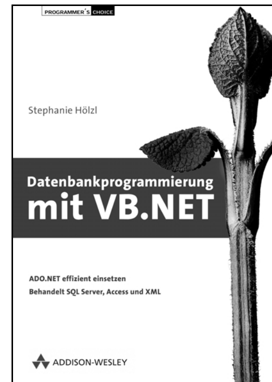
Bjarne Stroustrup
1084 Seiten
€ 49,95 (D), € 51,40 (A)
ISBN 3-8273-1660-X

Das Buch, geschrieben vom Erfinder der Sprache, ist das umfassendste Werk zu C++. Es basiert auf dem ANSI/ISO-C++-Standard und vermittelt aktuelle und verständliche Informationen zur Sprache, zur Standard Library und zu Design-Techniken. Die 4. Auflage des Bestsellers hat zwei neue Anhänge über Locales und Exception Safety.

Die C# Programmiersprache

Anders Hejlsberg, Scott Wiltamuth, Peter Golde
696 Seiten
€ 49,95 (D), € 51,40 (A)

C# ist eine moderne, objektorientierte und typsichere Programmiersprache, die die Produktivität eines RAD-Tool mit der Leistungsfähigkeit von Sprachen wie C oder C++ verbindet. Mit diesem Buch erhalten Sie die komplette Sprachreferenz. geschrieben vom Erfinder selbst und Mitgliedern des Design-Teams.



Datenbankprogrammierung mit VB.NET

Stephanie Hölzl
700 Seiten
€ 49,95 (D), € 51,40 (A)
ISBN 3-8273-2155-7

Mit dem .NET Framework kam auch eine neue Art des Datenbankzugriffs, die von den Programmierern Umdenken erfordert. Dieses Buch zeigt sowohl die Unterschiede zwischen dem klassischen ADO und ADO.NET als auch die Möglichkeiten, die die neue Technologie mit sich bringt. Die Funktionalität und der Umgang mit der Technologie wird anhand zahlreicher Beispiele erläutert.

PEARSON Education

ADDISON-WESLEY

Andy Bosch

Java Server Faces

Das Standard-Framework zum Aufbau webbasierter Anwendungen

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen
eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Abbildungen und Texten wurde mit größter
Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben
und deren Folgen weder eine juristische Verantwortung noch
irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der
Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden,
sind gleichzeitig eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.

5 4 3 2 1
06 05 04

ISBN 3-8273-2127-1

© 2004 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Marco Lindenbeck, webwo GmbH (mlindenbeck@webwo.de)
Titelbild: © Karl Blossfeldt Archiv; Ann und Jürgen Wilde, Zülpich/VG Bild-Kunst Bonn, 2004
Korrektur: Simone Meißner, Fürstenfeldbruck
Lektorat: Frank Eller, feller@pearson.de
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Inhalt

	Vorwort	**9**
1	**Einführung**	**11**
1.1	Zielsetzung dieses Buches	11
1.2	Voraussetzungen des Lesers	12
1.3	Systemvoraussetzungen	12
1.4	Aufbau des Buches	14
2	**Grundlagenwissen**	**17**
2.1	Servlets	17
2.2	JavaServer Pages	23
2.3	JSP-Architektur-Modelle	30
2.4	Taglibs und die JSTL	32
2.4.1	Funktionsweise von Taglibs	33
2.4.2	Standardisierte Taglibs – die JSTL	37
2.5	Das Model-View-Controller-Entwurfsmuster	45
2.6	Fazit	48
3	**Installation der JavaServer Faces-Demoanwendungen**	**49**
3.1	Vorbereitungen	49
3.2	Start der Beispiele	50
3.3	Fazit	52
4	**Einführung in JavaServer Faces**	**53**
4.1	Was ist JavaServer Faces?	53
4.2	Begriffsbestimmung – Webanwendung, Framework und Portale	54
4.3	JSF und andere Frameworks	57
4.4	Warum JSF?	59
4.5	Was ist eine JSF-Anwendung?	60
4.6	Zielgruppe von JSF	61
4.7	Wer steckt hinter JSF?	62
4.8	Spezifikation, Implementation und Open Source	62
4.9	JSF und andere Technologien	63
4.10	Toolunterstützung	64
4.11	JSF und EJBs	65
4.12	JSF und JavaScript	65
4.13	Rollenkonzept	66
4.14	Lebenszyklus einer JavaServer Faces-Seite	67
4.15	Fazit	72

5 Die erste eigene JSF-Applikation 73
5.1 Vorgehensweise 73
5.2 Einrichten der Entwicklungsumgebung 75
5.3 Aufsetzen eines neuen Projektes 76
5.4 Konfigurationsdateien 78
5.5 Benötigte jar-Dateien hinzufügen 81
5.6 JavaBeans und Modellobjekte 82
5.7 Bean Management 85
5.8 Erzeugen der JSF-Seiten 86
5.9 Anlegen einer index-Datei 88
5.10 Festlegung der Navigation 89
5.11 Start der Anwendung 90
5.12 Ausblick 91
5.13 Fazit 92

6 JSF im Detail 93
6.1 Konfigurationsdateien 94
6.2 Die JSF-Tag-Bibliotheken 97
6.2.1 Core-Tags 98
6.2.2 HTML-Tags 100
6.3 Das UI-Komponentenmodell 102
6.3.1 Die Basisklasse UIComponent 104
6.3.2 Die UI-Komponentenklassen 106
6.3.3 Das Rendering-Modell 107
6.4 Bean Management 109
6.4.1 Managed-Bean-Konzept 109
6.4.2 Backing Beans 114
6.4.3 Initialisierungsparameter 117
6.4.4 ValueBinding und MethodBinding 124
6.5 Validierung 130
6.6 Meldungen und Fehlermeldungen 136
6.7 Datenkonvertierung 142
6.8 FacesContext 149
6.9 Navigationskonzept 151
6.9.1 Statische versus dynamische Navigation 158
6.10 Eventhandling 161
6.10.1 Eventverarbeitung 162
6.10.2 Action-Events 163
6.10.3 Value-Change-Events 166
6.10.4 Phase Events 169
6.11 Zustandsspeicherung 171
6.12 Application-Objekt 174
6.13 Internationalisierung 175
6.14 Fazit 177

7 UI-Komponenten und ihre Darstellung 179
7.1 Grundaufbau von UI-Komponenten 180
7.2 Komponentenbaum 181
7.3 Renderer 182
7.4 HTML-Komponenten 183
7.5 Views und Subviews 184

7.6	UIForm-Komponente	185
7.7	UICommand-Komponente	187
7.7.1	HTMLCommandButton-Komponente	187
7.7.2	HTMLCommandLink-Komponente	191
7.8	UIInput-Komponente	194
7.8.1	HtmlInputText-Komponente	196
7.8.2	HtmlInputSecret-Komponente	199
7.8.3	HtmlInputHidden-Komponente	200
7.8.4	HtmlInputTextArea-Komponente	200
7.9	UISelectBoolean-Komponente	201
7.10	UISelectOne-Komponente	203
7.10.1	HTMLSelectOneRadio-Komponente	206
7.10.2	HTMLSelectOneListbox-Komponente	208
7.10.3	HTMLSelectOneMenu-Komponente	209
7.11	UISelectMany-Komponente	210
7.11.1	HTMLSelectManyCheckbox-Komponente	213
7.11.2	HTMLSelectManyMenu-Komponente	214
7.11.3	HTMLSelectManyListbox-Komponente	216
7.12	UIOutput-Komponente	217
7.12.1	HtmlOutputText-Komponente	217
7.12.2	HtmlOutputLabel-Komponente	219
7.12.3	HtmlOutputFormat-Komponente	220
7.12.4	HtmlOutputLink-Komponente	221
7.13	UIMessage- und UIMessages-Komponente	222
7.14	UIGraphic-Komponente	224
7.15	UIPanel-Komponente	226
7.15.1	HtmlPanelGrid-Komponente	226
7.15.2	HtmlPanelGroup-Komponente	233
7.16	UIData-Komponente	234
7.17	Fazit	242
8	**Die Beispielapplikation JSF-WebLog**	**243**
8.1	Einführung in die Beispielapplikation	243
8.2	Vorgehensweise	245
8.3	Entwickeln der Use-Cases	245
8.4	Screenflow	249
8.5	Datenmodellierung	253
8.6	Datenbankanbindung mit Torque	254
8.6.1	Installation von Torque	255
8.6.2	Anpassen der Konfigurationsdateien	255
8.6.3	Erstellen des Datenbankschemas	257
8.6.4	Erzeugen der Modell- und Peerklassen	260
8.6.5	Erstellen der Datenbank	261
8.6.6	Einbindung in die Anwendung	261
8.7	Projekt einrichten	262
8.8	Managed-Beans	263
8.9	Entwurf der Webseiten	269
8.10	Aufbau der Navigation	270
8.11	Implementieren der Funktionalität	272
8.12	Erweiterungen des WebLogs	274

8.12.1 Benutzerfreundliche Fehlermeldungen 274
8.12.2 RSS-Newsfeed 284
8.13 Fazit 306

9 JSF erweitern und anpassen 307
9.1 Erstellen eigener UI-Komponenten 307
9.1.1 Die Prozentanzeige-Komponente 309
9.1.2 Eine Eingabekomponente für Kreditkartendaten 324
9.1.3 Die Kreditkartenkomponente II 342
9.2 Benutzerspezifische Validatoren 351
9.2.1 Überprüfung von E-Mail-Adressen 352
9.2.2 Überprüfung von Kreditkartenangaben 357
9.3 Benutzerspezifische Konverter 366
9.4 Fazit 370

10 Beispiele aus der Praxis – Little Best Practices 371
10.1 Bereitstellen von Managed-Beans 372
10.2 JSF und JSTL SQL – Eine Alternative? 375
10.3 Session oder Datenbank? 381
10.4 Gültigkeitsbereich von Modellobjekten 383
10.5 Ressourcendateien 384
10.6 Anlegen einer index.html 386
10.7 Steuerung der Komponentensichtbarkeit 387
10.8 Wohin mit der Business-Logik? 394
10.9 Aktions-Methode oder ActionEvent? 397
10.10 Verwendung von Stylesheets 398
10.11 Seitenverlinkung 401
10.12 JSF und EJBs 402
10.13 Cancel-Buttons 418
10.14 Fazit 418

11 JSF und Struts 421
11.1 Was ist Struts? 421
11.2 Weitere Entwicklung von Struts 422
11.3 Zusammenspiel von JSF und Struts 422
11.4 Von der Theorie in die Praxis 423
11.5 Einblick in die Struts-Anwendung 426
11.5.1 Struts-Komponenten 426
11.5.2 Struts-Seiten 427
11.5.3 Formular-Beans 429
11.5.4 Controller 431
11.5.5 Struts-Konfigurationsdatei 432
11.6 Verwendung der Integrationsbibliothek 434
11.7 Fazit 439

Anhang 441
A.1 Inhalt der CD 441
A.2 Konfigurationsdatei faces-config.xml 441
A.3 Webadressen und -ressourcen 451
A.4 Faces Console 451

Index 453

Vorwort

Ein Buch über eine neue Technologie zu schreiben ist immer etwas Besonderes. Früher galt es noch als Abenteuer, mit dem Schiff ferne Länder zu erforschen und neue Erkenntnisse in die Heimat zu bringen. Ähnlich erging es mir beim Erstellen dieses Buches. Es war bekannt, dass eine neue Technologie für Webanwendungen – JavaServer Faces oder kurz JSF genannt – erscheinen sollte. Bald gab es auch eine erste Early Access-Version, der noch drei weitere folgen sollten. Mit jeder neuen Version des Frameworks kamen neue Dinge dazu, die zunächst ausprobiert und häufig nur durch *Trial and Error* erlernt werden konnten, war doch die in dieser Phase zur Verfügung stehende Dokumentation noch so gut wie überhaupt nicht vorhanden. Aber es war spannend, die Entwicklung Stück für Stück mitzuverfolgen.

Eine große Hilfe beim Erkunden und auch beim Erlernen der neuen Technologie war dabei die JSF-Community, die sich sehr schnell (unterstützt durch einige wenige Foren im Internet) bildete und einen regen Informationsaustausch betrieb. Oftmals fanden gerade in der Anfangszeit von JSF hier sehr interessante und lehrreiche Diskussionen statt. Daher geht mein erster Dank an diese Community. Dies ist wieder einmal ein Beweis, wie das Internet Menschen der ganzen Welt zusammenbringen und somit einen aktiven Informationsaustausch unterstützen kann.

Mein Dank gilt auch Timo Buhmann von TEQneers, der mir beim Design der Beispielanwendung JSF-Weblog eine große Hilfe war. Da ich selbst zwar erkennen kann, ob eine Webseite ansprechend ist oder nicht, aber dies nicht selbst umsetzen kann, hat Timo mir hinsichtlich der graphischen Fragen unterstützend zur Seite gestanden.

Viele Anregungen zum Aufbau und zum Inhalt dieses Buches bekam ich auch von meinem Lektor Frank Eller, der sich gerade auch in der Entscheidungsphase, ein Buch über JSF zu publizieren, sehr stark für dieses Thema eingesetzt hat. Auch hierfür danke ich ihm sehr.

Last but not least danke ich auch meiner Lebensgefährtin Klaudia für die Geduld und auch für das Verständnis, dass ich gerade in den letzten Wochen vor Fertigstellung dieses Buches so gut wie überhaupt nicht mehr zu Hause zu sehen war.

1 Einführung

1.1 Zielsetzung dieses Buches

Webanwendungen haben sich in den letzten Jahren von ersten kleinen Anwendungen im Zuge des Internet-Hypes hin zu nutzenorientierten und etablierten Anwendungen im Internet gewandelt. Basierend auf der J2EE-Spezifikation von Sun, die einen einheitlichen Standard für die Entwicklung von Geschäftsanwendungen im Intra- und Internet definiert, existieren heutzutage eine nicht mehr überschaubare Anzahl an Webanwendungen. Seien es kleine Zinsrechner oder Abwicklungen von komplexen Geschäftsprozessen, Webanwendungen finden sich in nahezu jedem Einsatzgebiet wieder.

Für die Entwicklung und Programmierung von Webanwendungen gibt es jedoch mindestens genauso viele Frameworks und Technologien, die einen Entwicklungsprozess vereinfachen und beschleunigen wollen, wie Anwendungen selbst. Einen Überblick hierüber zu bekommen ist fast unmöglich. Zudem kommen ständig neue Technologien und kommerzielle wie freie (Open-Source) Frameworks auf den Markt.

Daher ist es sicherlich vorteilhaft, auf offizielle Standards der »Großen« im Markt zu setzen. Mit den *JavaServer Faces (JSF)* von Sun existiert ein offizielles Framework, das die Entwicklung von Webanwendungen einerseits definiert und spezifiziert, andererseits dem Entwickler einen mächtigen Werkzeugkasten an die Hand gibt.

Ziel dieses Buch ist es, Ihnen einen Einblick sowie eine detaillierte Einführung in die neue Technologie der JavaServer Faces zu geben und Ihnen das Framework einerseits näher zu bringen, andererseits Ihnen aber auch die Vorteile und Stärken zu demonstrieren. Ziel soll es sicherlich nicht sein, andere Frameworks schlecht zu reden und Ihnen blindlings einreden zu wollen, sich in jedem Fall für JavaServer Faces zu entscheiden. Vielmehr soll Ihnen das Buch auch eine Entscheidungsgrundlage liefern, auf deren Basis Sie das für *Ihr* Projekt und *Ihr* Unternehmen passende Framework auswählen können.

JavaServer Faces ist mit Sicherheit bereits in dieser ersten Version sehr umfangreich und leistungsfähig. Aufgrund einer sehr erfahrenen Gruppe, die das Framework konzipiert und beratend entworfen hat, wurden wichtige und gute Designentscheidungen

getroffen, die das Ergebnis maßgeblich beeinflusst haben und so JavaServer Faces zu dem machten, was es heute bereits ist: ein leistungsfähiges und erweiterbares Framework zur Erstellung von User-Interfaces für Webanwendungen.

Dennoch ist JavaServer Faces ein Framework, das noch recht jung in diesem Markt ist. Dieses Buch begleitet Sie daher Schritt für Schritt beim Kennenlernen dieser neuen Technologie. Es werden viele Beispiele gezeigt und ausführlich erläutert. Dieses Buch soll Ihnen helfen, möglichst schnell die Konzepte von JavaServer Faces zu lernen, um Sie darauf aufbauend in einem konkreten Projekt zielgerichtet einsetzen zu können.

1.2 Voraussetzungen des Lesers

In diesem Buch erhalten Sie eine komplette Einführung in die Technologie von JavaServer Faces (JSF). JSF stellt ein Rahmenwerk für die Entwicklung von webbasierten Anwendungen dar. Nach Aussagen von Sun selbst soll mittels JSF auch ein nicht allzu erfahrener Entwickler schnell und einfach eigene Webanwendungen entwickeln können. Aufgrund dieser Aussage ist somit ein Basiswissen im Bereich Webanwendungen und Java durchaus ausreichend, wobei Grundlagenkenntnisse im Bereich der Anwendungsentwicklung von Webanwendungen sicherlich von Vorteil sind.

JSF selbst baut auf den Basistechnologien J2EE, JSP, Servlets und Taglibs auf. Im Kapitel 2 wird daher kurz auf das Basiswissen im Bereich JSP- und Servletprogrammierung sowie auf die Verwendung der JSTL und Taglibs allgemein eingegangen. Dies kann jedoch eine umfassende Einführung in diese Technologien nicht ersetzen.

Für einen reinen Einsatz von JSF reichen minimale Basiskenntnisse in der JSP- und Servletprogrammierung vollkommen aus, da die genaue Verwendung von JSF in diesem Buch ausführlich auch für Neueinsteiger erklärt wird. Für denjenigen Leser, der sich intensiver mit JSF auseinander setzen möchte und gegebenenfalls eigene Erweiterungen und Anpassungen in JSF vornehmen möchte, ist ein tiefer gehendes Verständnis der JSTL sowie von Tag-Bibliotheken erforderlich. Eine gute Einführung in diese Technologien leistet beispielsweise das Tutorial von Sun, das direkt über deren Webseite aufgerufen werden kann.

1.3 Systemvoraussetzungen

JavaServer Faces ist kein Open-Source-Produkt, jedoch kann sowohl die Spezifikation als auch die Referenzimplementierung kostenfrei aus dem Internet heruntergeladen werden.

Für die Entwicklung einer Webanwendung wird eine entsprechende Entwicklungsumgebung benötigt. Grundsätzlich kann hierfür jede Java-Entwicklungsumgebung ver-

wendet werden, in diesem Buch wird allerdings die Entwicklungsumgebung *Eclipse* verwendet. Eclipse ist eine freie Entwicklungsumgebung und kann kostenfrei unter *http://www.eclipse.org* heruntergeladen werden.

Da mittels JSF serverseitige Webanwendungen entwickelt werden, wird ferner als Laufzeitumgebung für die Applikation ein *Application Server* für das Testen und Ausführen der Beispielprogramme benötigt. Sämtliche Beispiele in diesem Buch wurden unter Verwendung des Jakarta-Tomcat-Servers aufgebaut. Für die Integration des Tomcat-Servers in die Entwicklungsumgebung Eclipse wird das Plug-In von *Sysdeo* verwendet, das über die Website *http://www.sysdeo.com/eclipse/tomcatPlugin.html* heruntergeladen werden kann. Mehr zur Installation und Konfiguration der Entwicklungsumgebung finden Sie im Kapitel 5.2.

Sollten Sie eine andere Entwicklungsumgebung bzw. einen anderen Server verwenden, müssen Sie eventuell in der dazugehörigen Dokumentation nachlesen, wie der Server für Testläufe in die Entwicklungsumgebung integriert werden kann. Zwar ist auch ein so genannter Stand-alone-Betrieb möglich, um jedoch eine Anwendung auch debuggen zu können ist eine Integration des Servers in die Entwicklungsumgebung sehr ratsam.

Da Webanwendungen häufig eine Datenbank für die Speicherung von Benutzer- und Programmdaten verwenden, wird in diesem Buch die Datenbank *MySQL* eingesetzt. Auch MySQL kann über das Internet unter *http://www.mysql.com/downloads* heruntergeladen und kostenfrei installiert werden. Auch hier gilt wieder, dass Sie natürlich auch eine andere Datenbank verwenden können. Da nur an wenigen Stellen auf eine Datenbankanbindung zurückgegriffen wird, kann bei diesen Beispielen auch sehr einfach der Transfer auf ein anderes Datenbanksystem erfolgen.

Im Rahmen der Beispielanwendung *JSF-Weblog*, die im Laufe des Buches aufgebaut und erläutert wird, kommt zudem das O/R (Objekt-Relational)-Mapping-Tool *Torque* zum Einsatz. Torque ist ein Teilprojekt des Jakarta-Teams der Apache-Gruppe und kann ebenfalls kostenfrei als Open-Source über die Website *http://www.apache.org* heruntergeladen werden. Damit soll die Anbindung einer Datenbank an eine Java-Anwendung vereinfacht werden, da durch den Einsatz von Torque kein Sql direkt mehr verwendet werden muss und die notwendigen Zugriffsklassen automatisch generiert werden. Mehr zu Apache Torque ist in Kapitel 8.6 zu finden.

Insgesamt sei erwähnt, dass viele der genannten Programme und Bibliotheken auf der beigefügten CD enthalten sind. Ebenso sind natürlich alle Kurzbeispiele sowie die Beispielanwendung als Quelldateien ebenfalls der CD beigefügt.

1.4 Aufbau des Buches

Dieses Buch ist so aufgebaut, dass Sie schrittweise in die Technologie von JavaServer Faces eingeführt werden. Nach einigen einleitenden ersten Kapiteln wird in Kapitel 2 zunächst einmal auf die Basistechnologien eingegangen, die JSF zu Grunde liegen. Dieses Kapitel ist nicht zwingende Voraussetzung, um JSF zu lernen. Es erleichtert jedoch den Umgang mit dieser neuen Technologie erheblich, wenn ein gewisses Grundverständnis für die Themen Servlets, JSP oder Taglibs vorhanden ist. Das Kapitel 2.5 dagegen ist schon fast ein Muss, um die Architekturgrundsätze von JSF besser verstehen zu können. Als erfahrener Java-Entwickler im J2EE-Bereich können Sie das Kapitel 2 natürlich auch überspringen.

Im Kapitel 3 wird die Installation der Beispielanwendungen erklärt, so dass Sie bereits nach kurzer Zeit eine erste JSF-Anwendung am Laufen haben. Es handelt sich dabei um die Beispielanwendungen, die bei der Referenzimplementierung mitgeliefert werden.

Im Kapitel 4 wird zunächst noch einmal auf einige grundlegende Fragen zu JSF eingegangen. Es wird dargestellt, wie genau sich JSF positioniert und was es von anderen Frameworks unterscheidet.

Danach geht es gleich »in medias res«, also mitten hinein. In Kapitel 5 wird die erste eigene Anwendung mit JSF realisiert. Damit erhalten Sie bereits sehr schnell einen ersten Eindruck, wie einfach und komfortabel Anwendungen mit JSF erstellt werden können. Natürlich wird dabei noch nicht auf jede Feinheit eingegangen, es soll nur einmal ein Gefühl vermitteln, was bei einer Anwendungsentwicklung mit JSF notwendig ist. Wichtig ist allerdings das Einrichten der Entwicklungsumgebung. Da gerade das Einrichten der Entwicklungsumgebung oftmals eine zeitraubende Angelegenheit ist, wird zunächst auf die möglichen Fallstricke und Besonderheit eingegangen. Es wird dabei die Entwicklungsumgebung für eine erste *Hallo-Welt*-Anwendung aufgebaut und konfiguriert. Auf dieses Kapitel können Sie natürlich immer wieder zurückgreifen, sollten Sie Schwierigkeiten bei der Konfiguration Ihrer Entwicklungsumgebung (Workspace) haben.

Das Kapitel 6 ist ein sehr wichtiges Kapitel hinsichtlich des Gesamtverständnisses für JSF. Es werden die wichtigsten Bestandteile von JSF im Einzelnen vorgestellt und erläutert. Dieses Kapitel beinhaltet alles wichtige Basiswissen, deshalb sollten Sie dieses Kapitel wenn möglich auch komplett durcharbeiten, bevor Sie tiefer in die Technologie einsteigen. Dieses Kapitel kann natürlich auch jederzeit als Nachschlagehilfe verwendet werden, wenn Sie zu bestimmten Themenbereichen nochmals etwas genauer wissen möchten.

Das Kapitel 7 ist dagegen weniger für ein komplettes Durchlesen bestimmt, sondern eher als Nachschlagewerk zu verstehen. Im Laufe Ihrer Arbeit mit JSF werden Sie ver-

schiedene UI-Komponenten in Ihre eigene Anwendung integrieren müssen. Für die genaue Verwendung können Sie das Kapitel 7 daher als Nachschlagewerk verwenden. Für ein erstes Verständnis, wie Oberflächenkomponenten in JSF verwendet werden, reicht ein »Überfliegen« dieses Kapitels aus.

In Kapitel 8 wird das gesamte Wissen, das Sie in den vorherigen Kapiteln erworben haben, anhand einer umfassenden Beispielanwendung nochmals zusammengefasst und anhand eines konkreten Beispiels erläutert. Zudem erhalten Sie mit der Beispielanwendung *JSF-Weblog* eine Anwendung, die Sie selbst verwenden und auch ins Internet stellen können.

Natürlich darf auch ein Kapitel über die Anpassungen und Erweiterungen in JSF nicht fehlen. Im Kapitel 9 wird erklärt, an welchen Stellen Sie angreifen müssen, um eigene Komponenten oder Validatoren sowie Renderer zu erzeugen und dem JSF-Framework hinzuzufügen. Diese Kapitel ist sicherlich kein Muss für den Umgang mit JSF. Setzen Sie JSF in der Praxis in einem konkreten Projekt einmal ein, werden Sie sicherlich bald feststellen, dass Sie um die Erzeugung von eigenen Komponenten oftmals nicht herum kommen. Zumal ist JSF speziell dafür geschaffen, eigene Erweiterungen und Anpassungen in das Framework miteinzubringen.

In Kapitel 10 finden Sie einige Anregungen und Tipps aus der Praxis, wie Sie mit Hilfe von JSF bestmögliche Ergebnisse erzielen können. Es werden Architekturansätze besprochen sowie so genannte *Best Practices*, also bewährte Lösungen aus der Praxis, vorgestellt.

Speziell für die Anhänger und Anwendung des Frameworks Struts wird im Kapitel 11 auf eine Integration bzw. Migration von Struts in JSF eingegangen. Struts ist ein sehr beliebtes Framework, das in der Zwischenzeit eine riesige Fangemeinde aufgebaut hat. Um Vorteile beider Frameworks nutzen zu können, wird in diesem separaten Kapitel auf die Integrationsbibliotheken Struts-Faces näher eingegangen.

Das gesamte Buch ist dabei so ausgelegt, dass wenn irgend möglich, Kurzbeispiele verwendet werden. Durch Kurzbeispiele ist recht schnell die konkrete Verwendung zu erkennen. Zusätzlich wird im Buch eine umfangreiche Beispielanwendung *JSF Weblog* aufgebaut. Die Kurzbeispiele sollen Ihnen somit eine schnelle Hilfe sein, um die Verwendung von JSF zu erklären, ohne ein aufwändiges Rahmenwerk dabei zu haben, das am Anfang nur verwirrend ist. Die etwas umfangreichere Beispielanwendung dagegen soll Ihnen den Einsatz von JSF in einer umfassenderen Anwendung demonstrieren.

2 Grundlagenwissen

Kapitelziel

Um JavaServer Faces anzuwenden und in Projekten erfolgreich einsetzen zu können, ist es nicht zwingend notwendig, ein erfahrener JSP- und Servletentwickler zu sein. Allerdings hilft dieses Wissen, die Funktionsweise von JavaServer Faces besser verstehen und einsetzen zu können.

Bevor direkt in die Arbeit mit JavaServer Faces eingestiegen wird, werden daher zunächst in diesem Kapitel die Technologien kurz besprochen, die für die Arbeit mit JavaServer Faces quasi die Basis bilden. Sicherlich kann man auch Erfolg mit JavaServer Faces haben, ohne sich in der Servlet- oder JSP-Programmierung auszukennen. Für ein tiefer gehendes Verständnis sollten jedoch mindestens die Grundzusammenhänge und Vorgehensweisen der Servlet- und JSP-Programmierung vorhanden sein. Auch im Hinblick auf die Zusammenarbeit der JSTL mit JSF ist es von Vorteil, die Arbeitsweise mit den Tags der JSTL zu kennen.

Um den grundlegenden Aufbau von JSF besser einordnen zu können, werden in diesem Kapitel wichtige und bekannte Architekturmodelle vorgestellt. Auch ohne den Anspruch, als Softwarearchitekt Anwendungen planen zu wollen, hilft ein Verständnis der Architektur von JavaServer Faces sehr viel.

Wie in vielen Frameworks, die sich mit dem Thema Oberfläche (GUI) beschäftigen, ist das *Model-View-Controller-Entwurfsmuster* (MVC) auch in JavaServer Faces umgesetzt. Daher wird zunächst das Prinzip und der Nutzen des MVC-Musters vorgestellt und näher erläutert.

2.1 Servlets

Mit Servlets wurde vor einigen Jahren erstmals die Möglichkeit geschaffen, Java auch serverseitig einzusetzen. Bis zu diesem Zeitpunkt wurde Java hauptsächlich dafür verwendet, clientseitige Anwendungen zu realisieren bzw. Java-Applets in Webseiten zu integrieren. Mit der Bereitstellung einer Servlet-API hatte ein Entwickler erstmals die Möglichkeit, schnell und komfortabel serverseitige Anwendungen zu realisieren. Die Anwendung selbst wird dabei in einem so genannten *Servlet-Container* im Webserver

ausgeführt. Das Ergebnis der Verarbeitung bzw. des Ablaufs eines Servlets wird im Normalfall als HTML-Seite an den Browser zurückgesendet und dort angezeigt. Folgende Darstellung verdeutlicht die Einordnung eines Servlets:

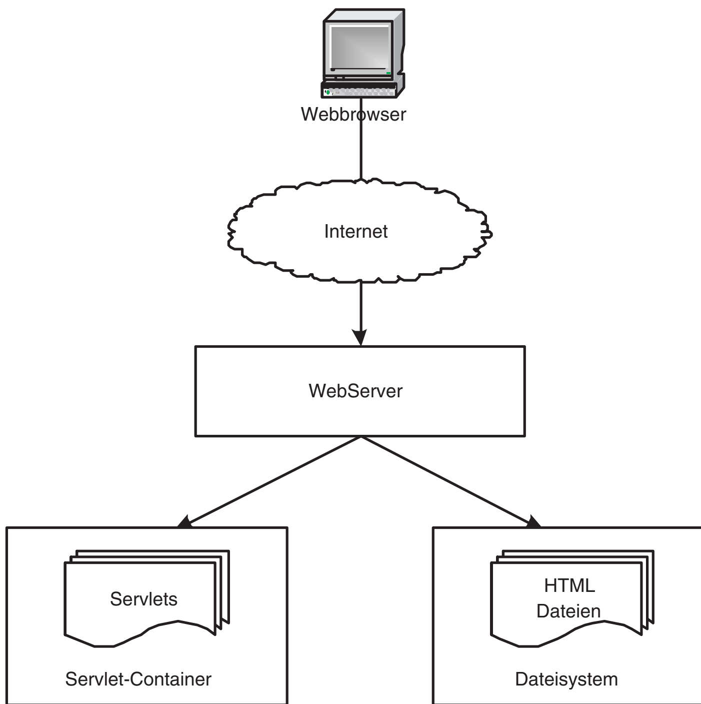


Abbildung 2.1: Servlet-Container im Webserver

Abbildung 2.1 verdeutlich das Zusammenspiel von Webserver, Servlet-Container und statischen Inhalten. Nachdem durch einen Benutzer eine Anfrage (ein Request) gestellt wurde, wird diese zunächst einmal am Webserver entgegengenommen. Dieser entscheidet, ob er selbst die Anfrage bearbeiten kann, und schickt gegebenenfalls statische Inhalte wie HTML-Seiten oder Bilder zum Browser zurück. Im Falle, dass ein Servlet angefragt wurde, wird die Anfrage an den Servlet-Container weitergeleitet, der daraufhin die Verarbeitung der Anfrage durchführt und das Ergebnis wiederum an den Browser zurücksendet. Für einen Benutzer ist es somit nicht direkt ersichtlich, ob eine statische Seite oder das Ergebnis eines Servlets zurückgeliefert wurde.

Ein Servlet-Container ist dabei eine Erweiterung eines Webservers in der Hinsicht, dass Java in einem separaten Umfeld, dem so genannten Servlet-Container, ausgeführt werden kann. Servlets sind dabei zunächst einmal Java-Klassen, die eine entspre-

chende Schnittstelle implementieren und somit die benötigte Funktionalität zur Verarbeitung eines Requests bereitstellen.

Aufbau eines Servlets

Ein Servlet ist zunächst einmal eine normale Java-Klasse, die (im Normalfall) von der Klasse `javax.servlet.http.HttpServlet` abgeleitet ist. Diese abstrakte Klasse implementiert bereits diejenigen Schnittstellen (Interfaces), die eine Servletklasse mindestens implementieren muss. Der Entwickler hat bei Verwendung der Klasse `HttpServlet` deshalb nur noch diejenigen Methoden zu überschreiben, die er für sein spezielles Servlet benötigt. Ein einfaches Servlet, das den Text *Hallo Welt* im Browser ausgibt, ist in Listing 2.1 abgebildet.

```
package com.edu.jsf.bsp.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Ein einfaches Hallo-Welt-Servlet
 */
public class HelloWorld extends HttpServlet {

  private String displayString;

  /**
   * Initialisierung des Servlets
   */
  public void init() throws ServletException {
    displayString = "Hallo Welt!!!";
  }

  /**
   * Implementierung des GET-Verfahrens
   */
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse res)
      throws ServletException, IOException {

    res.setContentType("text/HTML");

    PrintWriter pw = res.getWriter();
    pw.println("<HTML>");
    pw.println("<head><titel>");
```

```
    pw.println("Hallo Welt Servlet");
    pw.println("</titel></head>");
    pw.println("<body>");
    pw.println("<h3>" + displayString + "</h3>");
    pw.println("</body>");
    pw.println("</HTML>");
  }

  /**
   * Entladen des Servlets
   */
  public void destroy() {
    super.destroy();
    displayString = null;
  }
}
```

Listing 2.1: Ein einfaches Hallo-Welt-Servlet

Das Servlet kann in jedem Servlet-Container ausgeführt werden. Natürlich ist es ebenso möglich, das Servlet in einer Entwicklungsumgebung im Debug-Modus ausführen zu können. Damit kann eine Verarbeitung einer Anfrage Schritt für Schritt im Debugger nachvollzogen werden. Auf die Einrichtung einer funktionsfähigen Entwicklungsumgebung wird in Kapitel 5.2 näher eingegangen, sie wird deshalb in den folgenden Beispielen nicht weiter erläutert.

Wie in Abbildung 2.2 zu erkennen ist, wurde das Servlet über die Url *http://localhost:8080/JSFTraining/HelloWorld* aufgerufen. Diese Adressangabe besagt, dass der Applikationsserver auf der lokalen Maschine läuft (was für Testzwecke ratsam ist) und auf dem Port 8080 auf Anfragen hört. `JSFTraining` ist der so genannte Kontextpfad, der den Kontext bezeichnet, in dem das Servlet abläuft. `HelloWorld` wiederum ist der Servletname, über den das eigentliche Servlet angesprochen wird. Eine Zuordnung zwischen dem Servletnamen und der dazugehörigen Servletklasse wird im *Deployment-Deskriptor web.xml* hinterlegt. Der Deployment Deskriptor ist eine Konfigurationsdatei, in der pro Webanwendung verschiedene Einstellungen hinterlegt sind.

```
<servlet>
  <servlet-name>HelloWorld-Servlet</servlet-name>
  <servlet-class>
    com.edu.jsf.bsp.servlet.HelloWorld
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>HelloWorld-Servlet</servlet-name>
```

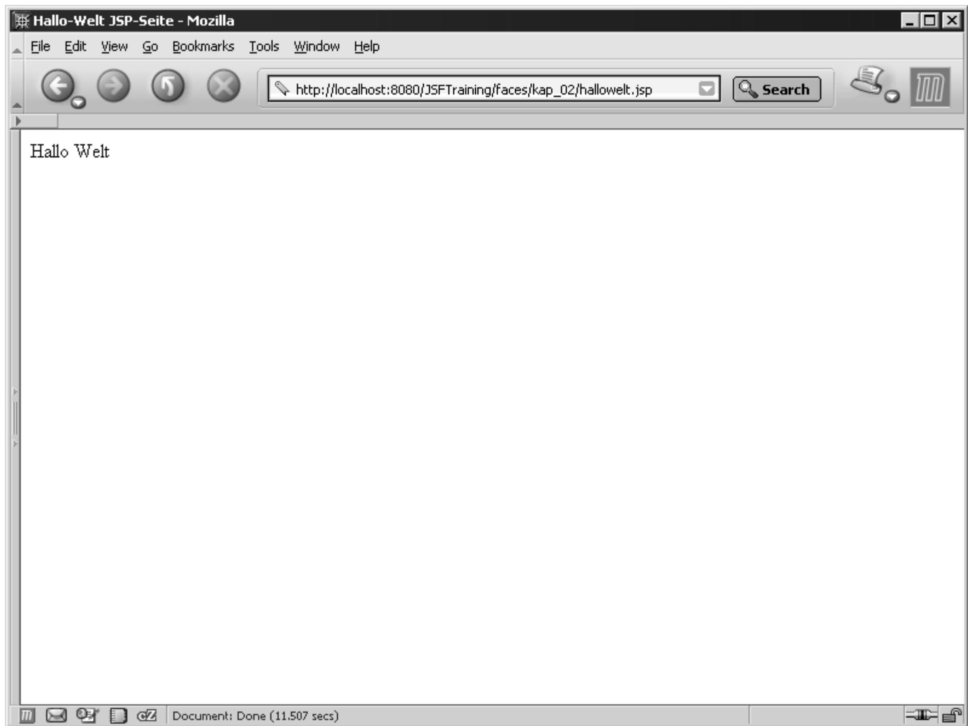


Abbildung 2.2: Hallo-Welt-Servlet

```
    <url-pattern>/HelloWorld</url-pattern>
  </servlet-mapping>
```

Listing 2.2: Auszug aus dem Deployment Deskriptor web.xml

In Listing 2.2 ist ein Auszug aus der Datei *web.xml* zu sehen. Es wird dort ein Servlet mit dem Namen *HelloWorld-Servlet* definiert, das die dazugehörige Klasse `com.edu.jsf.bsp.servlet.HelloWorld` besitzt. Des Weiteren kann dieses Servlet über die Url */HelloWorld* aufgerufen werden. Diese Zuordnung wird im Bereich `servlet-mapping` vorgenommen. Ein Verständnis des Deployment Deskriptors ist wichtig, da auch für JavaServer Faces später an dieser Stelle Eintragungen vorgenommen werden müssen.

Mit dem erfolgreichen Aufruf des Servlets werden eine Reihe von Aktionen in Gang gesetzt. So wird ein Servlet zunächst einmal initialisiert. Dies geschieht entweder bereits beim Start des Applikationsservers oder aber beim ersten Aufruf des Servlets. Danach kann das Servlet einkommende Requests verarbeiten, bis es schließlich irgendwann wieder entladen wird.

Charakteristisch für ein Servlet sind folgende Methoden:

- `init()`: Die `init`-Methode wird nur ein einziges Mal aufgerufen, und zwar genau zu dem Zeitpunkt, an dem das Servlet instanziert wird. Das Servlet bleibt so lange im Server (bzw. im Servlet-Container) geladen, bis es explizit, beispielsweise beim Herunterfahren des Servers, wieder aus dem Speicher entfernt wird.

- `doGet()` bzw. `doPost()`: Bei jedem Aufruf des Servlets wird ein eigener Thread gestartet, der eine `doGet`- bzw. `doPost`-Methode abarbeitet. In diesen Methoden findet die eigentliche Requestverarbeitung, also auch der gesamte Programmablauf, statt. Ein Servlet kann natürlich auch weitere Servlets aufrufen (Servlet-Chaining).

- `destroy()`: Analog zur `init`-Methode wird die `destroy`-Methode auch nur ein einziges Mal aufgerufen. Dies erfolgt im Normalfall beim Entladen des Servlets, wenn beispielsweise der Applikationsserver heruntergefahren wird.

Auf diese Weise ist es möglich, schnell und einfach servletbasierende Anwendungen zu entwickeln. Dadurch, dass Servlets kompiliert werden und somit nicht wie andere Skriptsprachen bei jedem Aufruf interpretiert werden, sind Servlets sehr performant. Auch die Möglichkeiten, die durch die Verwendung einer mächtigen Programmiersprache wie Java gegeben sind, haben Servlets eine große Beliebtheit gebracht.

Nachteile einer reinen Servlet-Architektur

Nach der ersten Euphorie, Java serverseitig einzusetzen, stellte man bald fest, dass eine Entwicklung von Webanwendungen basierend auf einer reinen Servlet-Technologie einige nachteilige Konsequenzen hatte:

- Aufgrund der Tatsache, dass die gesamte View in Java-Klassen codiert war, waren Designänderungen im Nachhinein nur sehr aufwändig zu realisieren.

- Designänderungen konnten nur von Entwicklern durchgeführt werden. Webdesigner ohne Programmierkenntnisse hatten keine Möglichkeit, eine fertige Anwendung zu verändern.

- Durch die Vermischung von HTML-Befehlen und der Java-Programmlogik war selbst die Schicht für Geschäftsprozesse (Business-Tier) oftmals nicht konsequent strukturiert. Ergebnis war daher häufig ein Wust an Klassen, die alle irgendwie miteinander verbunden waren, ohne dass jedoch eine saubere Schichteneinteilung zu erkennen war.

Für einfache und überschaubare Projekte konnten Servlets problemlos eingesetzt werden, bei umfangreicheren Projekten musste jedoch eine verbesserte Technologie zur Ergänzung eingebracht werden. Sun reagierte hierauf mit den JavaServer Pages (JSP).

2.2 JavaServer Pages

Aufgrund einiger Nachteile in einer reinen Servlet-Architektur veröffentlichte Sun die JavaServer Pages-Spezifikation (JSP). JSP ist eine Technologie, die Elemente aus HTML mit Elementen der Java-Programmiersprache vereint. Oftmals wird JSP auch als »Erweiterung von Java in Webseiten« bezeichnet. Letztenendes wird jedoch dem Entwickler durch die JSP-Technologie die Möglichkeit gegeben, Java-Code in eine bestehende HTML-Seite zu integrieren, ohne eine HTML-Seite zuerst komplett als Servlet umwandeln zu müssen. Wobei gerade letzte Aussage mit Vorsicht zu genießen ist. Auch eine JSP-Seite ist letzten Endes nichts weiter als ein Servlet, nur dass es für den Entwickler nicht direkt ersichtlich ist. Vielmehr wird beim Kompilieren einer JSP-Seite automatisch ein Servlet erzeugt, das entsprechend ausgeführt wird und somit die gleiche Funktionalität vorweist wie ein normales Servlet. Nur eben mit dem Unterschied, dass der Entwickler nicht mehr explizit ein Servlet erstellen muss, sondern lediglich einzelne Befehle in eine HTML-Seite einbaut.

```
<HTML>
<head>
  <title>Hallo-Welt JSP-Seite</title>
</head>
<body>
  <%="Hallo Welt" %>
</body>
</HTML>
```

Listing 2.3: Eine einfache JSP-Seite

In Listing 2.3 ist ein einfaches Beispiel für eine JSP-Seite gegeben. Es handelt sich dabei um eine JSP-Seite, in der eine einfache Anweisung eingebunden ist, die den Text *Hallo Welt* ausgibt. Um aus einer einfachen statischen HTML-Seite eine JSP-Seite zu machen, sind im Normalfall folgende Schritte notwendig:

1. Umbenennen der Datei von *.html*/*.htm* in *.jsp*, damit diese Datei als gültige JSP-Datei identifiziert werden kann.

2. Eventuell muss die JSP-Datei aus dem Verzeichnis für statischen Inhalt in ein Verzeichnis für dynamischen Inhalt verschoben werden. Während sich ein Webserver (z.B. Apache) um statische Inhalte wie HTML-Seiten oder Bilder kümmert, übernimmt ein Applikationsserver dynamische Inhalte wie Servlets oder JSP-Seiten (z.B. Apache Tomcat).

3. Einbinden von JSP-Elementen, um die Seite mit entsprechender Funktionalität zu versehen.

Natürlich hätte das geforderte Ergebnis auch mit einem reinen Servlet realisiert werden können. Es wäre dazu lediglich eine Servletklasse zu schreiben gewesen, die die gewünschte HTML-Ausgabe in eine out.println-Anweisung geschrieben hätte. Es ist jedoch mit Sicherheit sehr schnell zu erkennen, dass JSP-Seiten große Vorteile haben, speziell wenn es darum geht, HTML-Seiten mit Funktionalität zu versehen.

JSP-Elemente

In der JSP-Syntax existieren insgesamt sechs Typen von Elementen, die in eine JSP-Seite eingebaut werden können:

Elementtyp	**Tag Syntax**	**Beschreibung**
Direktive	<%@ Direktive %>	Über Direktiven werden spezielle Angaben zur JSP-Seite gemacht, die an den JSP-Compiler übermittelt werden. Dies kann z. B. eine Import-Anweisung oder auch ein Include sein.
Deklaration	<%! Deklaration %>	Über Deklarationen werden Variablen auf der Seite deklariert. Eine Deklaration selbst erzeugt keine Ausgabe, sondern bestimmt nur Variablen, die im weiteren Verlauf der Seite z. B. in Skriplets verwendet werden können.
Skriplet	<% Java-Code %>	Innerhalb eines Skriplets kann beliebiger Java-Code stehen. Ein Skriplet-Block kann dazu verwendet werden, um Programmlogik darin unterzubringen.
Ausdruck	<%=Ausdruck %>	Über einen Ausdruck kann eine direkte Ausgabe eines Wertes oder Ausdruckes direkt in die Seite erfolgen.
Aktion	<jsp:Aktionsname %>	Über Aktionen können bestimmte vordefinierte Instruktionen zur Laufzeit eingebunden werden, wie z. B. das Weiterleiten auf ein andere Seite oder das Befüllen eines Beans.
Kommentar	<%-- Kommentar --%>	JSP-Kommentare sind in der Ausgabe als HTML-Seite nicht mehr enthalten und sind daher nur in der eigentlichen JSP-Seite zu finden.

Tabelle 2.1: Übersicht über die verschiedenen JSP-Elementtypen

Implizite Variablen

Um einem Entwickler die Arbeit in einer JSP-Seite weiter zu vereinfachen, existieren zusätzlich neun implizite Variablen. Dies bedeutet, dass diese Variablen jederzeit in einer JSP-Seite benutzt werden können, ohne im Vorfeld explizit deklariert worden zu sein.

Variable	Klasse bzw. Interface	Beschreibung
`application`	`javax.servlet.ServletContext`	Kontext der Webanwendung. Über ihn können beispielsweise Angaben zur Umgebung abgefragt werden (z. B. der vollständige Installationspfad der Anwendung)
`session`	`javax.servlet.http.HttpSession`	Die Benutzersession der Webanwendung. Dieses Objekt ist nur gültig, wenn für die aktuelle Seite die Sessionverwaltung aktiv ist.
`request`	`javax.servlet.http.HttpServletRequest`	der originale Request an den Server
`response`	`javax.servlet.http.HttpServletResponse`	das `Response`-Objekt
`page`	`java.lang.Object`	das Servlet, das aus dieser Seite erzeugt wurde
`pageContext`	`javax.servlet.jsp.PageContext`	Im Gegensatz zum ServletContext existiert noch ein Seitenkontext, welcher Daten der aktuellen Seite speichert.
`out`	`javax.servlet.jsp.JspWriter`	der Outputstream für diese Seite
`config`	`javax.servlet.ServletConfig`	die Servletkonfiguration
`exception`	`java.lang.Throwable`	Ist nur auf Fehlerseiten gültig und speichert in solchen Fällen die Exception.

Tabelle 2.2: Implizite Variablen in einer JSP-Seite

So ist es z. B. möglich, direkt in einer JSP-Seite mittels

```
request.getParameter("id")
```

auf einen Parameter mit dem Namen `id` zuzugreifen, ohne im Vorfeld die Variable `request` deklariert zu haben. Die Erklärung ist darin zu suchen, dass aus einer JSP-Seite intern wiederum ein Servlet generiert wird. Beim Generieren der Servlet-Klasse werden dabei die in Tabelle 2.2 gezeigten Variablen automatisch miteingebunden.

Verwendung von JavaBeans

Ein Bean ist per Definition eine selbstständige, unabhängige Softwarekomponente, die Eigenschaften und Funktionen kapselt und selbst wiederum in andere Komponenten oder Programme integriert werden kann. Durch JavaBeans wird die Möglichkeit gegeben, Software modular und vor allen Dingen auch so zu entwerfen, dass einzelne Komponenten wiederverwendbar sind. JavaBeans können Teile der Programmlogik in sich kapseln und vereinfachen im Zusammenhang mit JSP die Übersichtlichkeit der einzelnen Seiten. Die Eigenschaften eines Beans sind im Normalfall `private` und vor direkten Zugriffen von außen geschützt. Für den Zugriff auf Eigenschaften existieren öffentliche Zugriffsmethoden (so genannte getter- und setter-Methoden).

```
package com.edu.jsf.bsp.bean;

/**
 * Diese Klasse stellt ein einfaches JavaBean dar.
 */
public class SimpleBean {

  private String firstname = "";
  private String lastname  = "";

  /**
   * liefert den Vornamen zurück
   */
  public String getFirstname() {
    return firstname;
  }

  /**
   * liefert den Nachnamen zurück
   */
  public String getLastname() {
    return lastname;
  }

  /**
   * setzt einen neuen Vornamen in das Bean
   */
  public void setFirstname(String newfirstname) {
    firstname = newfirstname;
  }

  /**
   * setzt einen neuen Nachnamen in das Bean
   */
  public void setLastname(String newlastname) {
    lastname = newlastname;
  }
}
```

Listing 2.4: Ein einfaches JavaBean

In Listing 2.4 ist ein einfaches JavaBean dargestellt. Es beinhaltet die Eigenschaften `firstname` und `lastname`, also Vor- und Nachname. Über öffentliche Zugriffsmethoden können die Eigenschaften abgefragt bzw. gesetzt werden. Wichtig ist dabei, dass nie direkt auf die Eigenschaften zugegriffen werden kann. Dadurch kann gewährleistet werden, dass sich ein Bean immer in einem konsistenten Zustand befindet. Um dies zu verdeutlichen, kann als Gedankenspiel wiederum das Bean für eine Person verwendet werden, das zusätzlich noch eine Eigenschaft für das Geburtsdatum aufweist. Wäre es möglich, auf die Variable, in der das Geburtsdatum gespeichert ist, direkt zuzugreifen,

könnte jedes beliebige Datum eingegeben werden. Es könnte somit auch ein Datum eingegeben werden, das erst in der Zukunft liegt. Werden jedoch die Setter-Methoden verwendet, findet darin meist eine erste Validierung statt, so dass z.B. das Geburtsdatum in der Vergangenheit liegen muss und das (berechnete) Alter nicht größer als 100 sein sollte.

Die JSP-Technologie unterstützt die JavaBean-Architektur in der Form, dass eine sehr einfache Verwendung der Beans in JSP-Seiten möglich ist. Dazu existieren hauptsächlich drei Aktionen, mit denen auf Beanfunktionalität zugegriffen werden kann:

- `<jsp:useBean>`: Um dem JSP-Compiler mitzuteilen, dass auf einer Seite ein Bean verwendet wird, muss auf jeder Seite eine useBean-Anweisung stehen.
- `<jsp:setProperty>`: Mit dieser Aktion können neue Werte in das Bean gesetzt werden.
- `<jsp:getProperty>`: das entsprechende Gegenstück zu setProperty

Mit diesen drei Aktionen kann ein Webentwickler bereits Beans in JSP-Seiten integrieren und damit einen Großteil der Programmlogik aus den JSP-Seiten heraus in eine separate Beanklasse transferieren. Das Beispiel in Listing 2.5 und Listing 2.6 zeigt die Verwendung der Syntax nochmals anhand einer kurzen Anwendung.

```
<jsp:useBean id="Person"
    class="com.edu.jsf.bsp.bean.SimpleBean" scope="session" />
<HTML>
<head>
  <title>JSF-Beispiel: Benutzeranmeldung</title>
</head>

<body>
  <h3>Jsp und Beans</h3>
  <i>Dieses Beispiel demonstriert die
     Verwendung von Beans in Jsp-Seiten</i>
  <br>
  <form name="testForm" method="post" action="usebean_answer.jsp">
    Ihr Vorname:
    <input type="text" name="firstname"
        value="<%=Person.getFirstname() %>">
    <br>
    Ihr Nachname:
    <input type="text" name="lastname"
        value="<%=Person.getLastname() %>">
    <br><br>
    <input type="submit" value="Abschicken">
  </form>
</body>
</HTML>
```

Listing 2.5: usebean_question.jsp: Dateneingabe

In Listing 2.5 ist ein einfaches Formular definiert, in dem zwei Eingabefelder für den Vor- und Nachnamen enthalten sind. Vor dem Anzeigen der Seite wird nachgeschaut, ob im aktuellen Objekt `Person` bereits Vorbelegungen für die Werte enthalten sind. Gegebenenfalls werden diese Werte dann gleich angezeigt. Das Formular kann mittels des *Submit*-Buttons an die Seite *usebean_answer.jsp* geschickt werden.

```
<jsp:useBean id="Person"
    class="com.edu.jsf.bsp.bean.SimpleBean" scope="session" />
<HTML>
<head>
  <title>JSF-Beispiel: Benutzeranmeldung</title>
</head>

<body>
  <h3>Jsp und Beans</h3>
  <i>Dieses Beispiel demonstriert die Verwendung
     von Beans in Jsp-Seiten</i>
  <br>

  <jsp:setProperty name="Person" property="firstname" />
  <jsp:setProperty name="Person" property="lastname" />
  <br>
  <u>Ihre Eingabe:</u><br>
  Vorname: <jsp:getProperty name="Person" property="firstname" />
  <br>
  Nachname: <jsp:getProperty name="Person" property="lastname" />
  <br>
  <form name="testForm" method="post"
      action="usebean_question.jsp">
    <input type="submit" value="Zurück">
  </form>
</body>
</HTML>
```

Listing 2.6: usebean_answer.jsp: Anzeige von Beanwerten

In diesem Beispiel, das aus zwei JSP-Seiten besteht, wird auf der ersten Seite, der Eingabeseite, nach dem Vor- und Nachnamen gefragt. Diese Angaben werden beim Abschicken des Formulars auf die zweite Seite, die Ausgabeseite, mit übergeben. Dabei müssen die Werte auf der zweiten Seite aus dem Request heraus in ein Bean gesetzt werden. Von dort aus werden sie wieder zur Anzeige gebracht.

Über die Anweisung `<jsp:setProperty>` kann auf den Request zugegriffen werden. Damit werden Angaben aus dem Request in ein entsprechendes Bean gesetzt. Die Angabe der Attribute `name` und `property` bestimmen, welches Bean und welche Eigenschaft befüllt werden soll. In dieser Syntax wird davon ausgegangen, dass der Wert für die Beaneigenschaft `firstname` im Request ebenfalls unter dem Schlüssel `firstname` abgelegt ist. Wurde hier ein anderer Schlüssel verwendet, muss zusätzlich ein Attribut `param` im `<jsp:setProperty>`-Tag mitangegeben werden.

Da von der Antwortseite die Möglichkeit gegeben ist, durch einen *Zurück*-Knopf nochmals auf die Eingabeseite zu wechseln, werden die aktuellen Eigenschaften des Beans ebenfalls in der Seite *usebean_question.jsp* angezeigt. Hierbei wird – wie in Listing 2.5 gezeigt – auf den Wert mittels eines Ausdrucks (*Expression*) zugegriffen.

Beim Setzen von Werten aus einem Request in ein Bean existiert jedoch noch eine weitere Möglichkeit, die eine Arbeit mit Beans in JSP-Seiten stark vereinfacht.

```
<jsp:useBean id="Person"
    class="com.edu.jsf.bsp.bean.SimpleBean" scope="session" />
<HTML>
<head>
  <title>JSF-Beispiel: Benutzeranmeldung</title>
</head>

<body>
  <h3>Jsp und Beans</h3>
  <i>Dieses Beispiel demonstriert die Verwendung
     von Beans in Jsp-Seiten</i>
  <br>

  <jsp:setProperty name="Person" property="*" />
  <br>
  <u>Ihre Eingabe:</u><br>
  Vorname: <jsp:getProperty name="Person" property="firstname" />
  <br>
  Nachname: <jsp:getProperty name="Person" property="lastname" />
  <br>
  <form name="testForm" method="post"
        action="usebean_question_2.jsp">
    <input type="submit" value="Zurück">
  </form>
</body>
</HTML>
```

Listing 2.7: Setzen der Eigenschaften mittels "*"

Ist sichergestellt, dass die Eigenschaften des Beans analog zu den Werten des Request benannt sind (bzw. umgekehrt), können sämtliche Eigenschaften durch eine einzige `<jsp:setProperty>`-Anweisung gesetzt werden. Im Attribut wird anstelle einer konkreten Eigenschaft der Platzhalter `*` angegeben. Es werden damit automatisch alle passenden Requestparameter durchsucht und bei Entsprechung in das Bean gespeichert. Diese Möglichkeit vereinfacht das Erstellen von formularbasierten Anwendungen in erheblichem Maße. In Kombination der JSP-Technologie mit JavaBeans können damit bereits sehr umfangreiche Projekte sehr elegant realisiert werden.

2.3 JSP-Architektur-Modelle

In den beiden letzten Abschnitten haben Sie gesehen, dass es an sich keine großen Unterschiede zwischen einem Servlet und einer JSP-Seite gibt. Grundsätzlich kann alles, was mit einem Servlet realisiert werden kann, auch in einer JSP-Seite abgebildet werden und umgekehrt. Es stellt sich daher die Frage, welche Technologie denn jetzt verwendet werden soll.

Im Regelfall besteht eine Webanwendung weder nur aus Servlets, noch ausschließlich aus JSP-Seiten. Das Zauberwort heißt hier *Kombination*. Denn jede Technik hat ihre Stärken und Vorteile, die auszunützen sind:

- JSP-Seiten eignen sich hervorragend zur Darstellung von Oberflächen. Dadurch, dass Befehle nur in eine bestehende HTML-Seite eingebaut werden müssen, können sehr schnell dynamische Oberflächen entwickelt werden. Auch haben Webdesigner die Möglichkeit, Änderungen in JSP-Seiten durchzuführen, wo hingegen dies in einer reinen Servlet-Architektur sicherlich nicht möglich wäre.
- Servlets eignen sich vor allem zur Entwicklung der Anwendungslogik. Dabei können gesonderte Module wie eine Benutzerauthentifizierung, Datenbankzugriffe oder auch Zugriffe auf Enterprise Java Beans mit integriert werden.

Aufgrund dieser Erkenntnis, dass moderne Webanwendungen aus einer Kombination aus Servlets und JSP-Seiten bestehen können, es jedoch unterschiedliche Architekturen geben kann, hat Sun zwei Architekturmodelle definiert. Nach diesen Vorgaben kann (oder sollte) sich ein Entwickler bei der Erstellung von Webanwendungen richten.

Architekturmodell I

Im Architekturmodell 1 existieren lediglich JSP-Seiten sowie JavaBeans. Jeder Request, der von einer Seite ausgelöst wird, wird auch immer auf eine JSP-Seite weitergeleitet. Es gibt somit keine zentrale Komponente, die eine Steuerungsfunktion übernimmt, vielmehr ist jede einzelne Seite für die korrekte Navigation zuständig. Die gesamte Anwendungslogik ist entweder in den JavaBeans oder in der JSP-Seite selbst hinterlegt. Dieser Ansatz eignet sich primär für kleinere, überschaubare Anwendungen.

Beim Ablauf eines Requests gelangt dieser zunächst an eine JSP-Seite, die bei Bedarf JavaBeans instanzieren und verwenden kann. Die JavaBeans wiederum greifen gegebenenfalls auf eine Datenbank zu und liefern die Werte über die Beaneigenschaften zur JSP-Seite zurück. Diese wird abgearbeitet und das Ergebnis im letzten Schritt an den Browser zurückgeliefert.

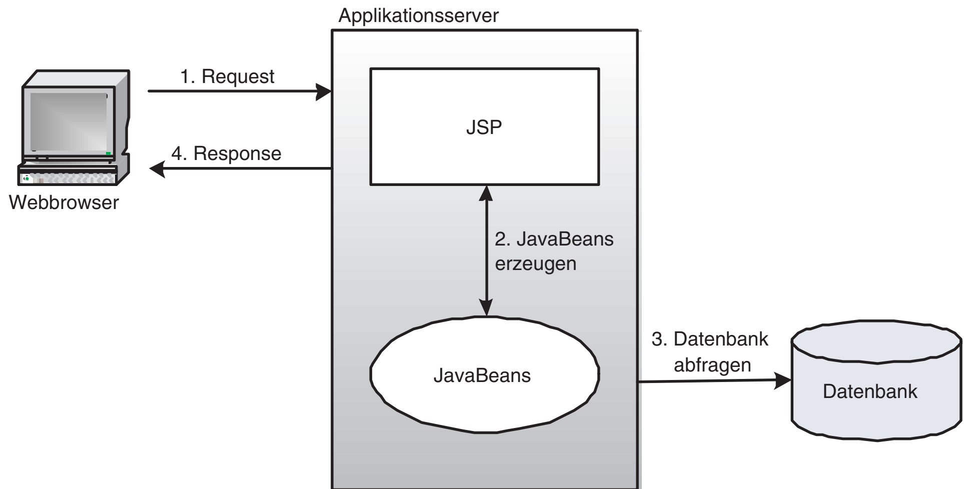


Abbildung 2.3: Modell-1-Architektur

Die Modell-1-Architektur lässt sich einerseits sehr schnell in einer Webanwendung umsetzen, da so gut wie keine zusätzlichen Verwaltungskomponenten entwickelt werden müssen. Jedoch weist dieses Modell einige gravierende Nachteile auf:

- Es gibt keine zentrale Stelle für die Navigation. Die Navigation ist verteilt auf alle JSP-Seiten. Dabei ist auf jeder JSP-Seite im `form`-Tag angegeben, wohin der Request erfolgen soll. Soll der Ablauf verändert werden, muss unter Umständen jede einzelne Seite angefasst werden.
- Extrem viel Programmlogik ist in der JSP-Seite selbst enthalten. Dies macht die Seite selbst sehr unübersichtlich. Was jedoch noch nachteiliger ist, ist die Tatsache, dass der Quellcode in der JSP-Seite nicht wiederverwendbar ist.
- Durch zu viel Programmlogik in der JSP-Seite wird es für Webdesigner extrem schwierig, selbst Änderungen an der Seite vornehmen zu können.

Viele Prototpyen oder kleinere Webanwendungen basieren auf der Modell-1-Architektur. Sie weist zwar einige Mängel auf, im Einzelfall kann es dennoch Sinn machen, Anwendungen gemäß Modell 1 aufzubauen. Wichtig ist es jedenfalls, die Einschränkungen, die mit dieser Architektur einhergehen, im Vorfeld genau zu kennen und bewusst in Kauf zu nehmen, wenn diese Architektur zum Einsatz kommt.

Architekturmodell 2

Das Architekturmodell 2 eliminiert viele Nachteile, die das Modell 1 aufweist. Die Architektur entspricht dem bekannten *Model-View-Controller (MVC)* Designpattern (vgl. dazu Abschnitt 2.5). Als zentrales Element in dieser Architektur dient ein Servlet,

das als zentrale Instanz bei jedem Request angesprochen wird. Es fungiert somit in der Rolle eines *Controllers*. Im Controller-Servlet erfolgt auch die Steuerung der Navigation, sprich die Weiterleitung auf die einzelnen JSP-Seiten. Das Servlet ist auch dafür zuständig, dass die JavaBeans, die in der JSP-Seite benötigt werden, korrekt und vor allem rechtzeitig befüllt werden. Gegebenenfalls finden Datenbankzugriffe oder entfernte Aufrufe statt. Die JSP-Seite selbst muss sich darum nicht kümmern, sie greift einfach nur auf die Daten der Beans zu.

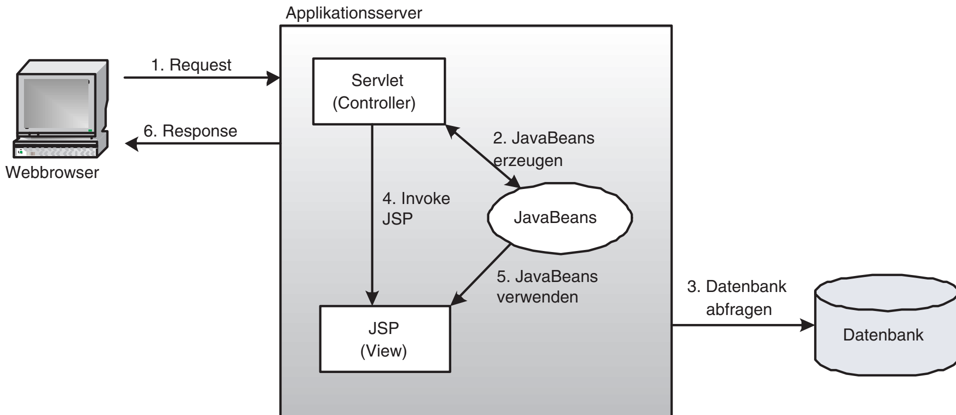


Abbildung 2.4: Modell-2-Architektur

Der Vorteil dieser Architektur liegt hauptsächlich in einer weiteren Modularisierung und Aufteilung in Zuständigkeiten (*separation of concerns*). Dadurch wird eine leichtere Wartbarkeit sowie eine erhöhte Wiederverwendbarkeit einzelner Komponenten ermöglicht. Viele Frameworks im Java-Umfeld (auch JavaServer Faces) beruhen auf genau diesem Ansatz.

2.4 Taglibs und die JSTL

JSPs sind eine hervorragende Möglichkeit, um statische HTML-Seiten mit Funktionalität und Dynamik zu versehen. Durch die Möglichkeit, Java dabei einzusetzen, hat der Entwickler ein großes Potenzial, mit dem er Funktionalitäten in eine Webseite einbauen kann. Oftmals sind es jedoch häufig wiederkehrende Aufgaben, die ein Entwickler von JSP-Seiten zu realisieren hat. So wäre es wünschenswert, hier eine Möglichkeit zu schaffen, dass bei gewissen Aufgabenstellungen keine Programmierarbeit mehr notwendig ist. Im Idealfall soll sogar ein Webdesigner selbst Funktionsblöcke in eine Seite integrieren können. Basierend auf diesen Vorgaben ermöglichte es Sun mit den so genannten Tag-Bibliotheken (Taglibs), dass Programmteile über benutzerdefinierte Tags in eine Seite eingebunden werden konnten.

2.4.1 Funktionsweise von Taglibs

Als Beispiel dient eine Anwendung, bei der an beliebiger Stelle innerhalb jeder Seite das aktuelle Datum angezeigt werden soll. Konkret kann diese Funktion z.B. in einer Webcommunity eingesetzt werden, in der aus Bedienerfreundlichkeit auf jeder Seite, die ein Benutzer besuchen kann, an einer fest definierten Stelle die Datumsanzeige erscheint. Mit konventionellen JSP-Funktionalitäten müsste jedes Mal ein Skriplet eingebaut werden, das mit jedem Ausführen das aktuelle Datum ermittelt, formatiert und entsprechend ausgibt.

```
<%@page import="java.util.Date" %>
<%@page import="java.text.DateFormat" %>

<HTML>
<head>
  <title>JSP-Beispiel: Aktuelles Datum</title>
</head>

<body>
  <h3>Datumsanzeige in einer Jsp-Seite</h3>
  <i>Dieses Beispiel demonstriert, wie das aktuelle Datum
     ohne Taglibs integriert werden kann:</i>
  <br>
  <br>
  Wir haben heute den
  <%
     Date curDate  = new Date();
     DateFormat df = DateFormat.getDateInstance();
     String formattedDate = df.format( curDate );
  %>
  <%=formattedDate %>

</body>
</HTML>
```

Listing 2.8: Beispiel einer Datumsausgabe ohne Taglibs

In Listing 2.8 erfolgt die Ausgabe des aktuellen Datums mit Hilfe eines Skriplets, in dem der String aufbereitet wird, zusammen mit einem Ausdruck, der den eigentlichen Textstring in der JSP-Seite ausgibt. Sicherlich könnte dies noch weiter vereinfacht werden, indem z.B. lediglich der Aufruf einer statischen Methode eingebaut wird, anstatt jedes Mal die komplette Funktion in der JSP-Seite zu haben. Dennoch ist diese Vorgehensweise nicht optimal. Ein Webdesigner muss sich damit einerseits mit Java-Quellcode beschäftigen, andererseits ist eine Änderung der Formatierung (wenn z.B. das Jahr nur noch 2-stellig angezeigt werden sollte) sehr aufwändig, da sämtliche Seiten geändert werden müssen, in denen diese Logik implementiert ist.

Mit Hilfe einer Taglib und eines entsprechenden benutzerdefinierten Tags ist daraufhin einerseits die Möglichkeit geschaffen worden, dass ein Webdesigner einfache Tags in seine Seite integrieren kann, ohne sich mit Java-Programmierung beschäftigen zu müssen. Andererseits wurde durch Einführung der Taglibs eine zusätzliche Möglichkeit geschaffen, Webanwendungen weiter zu modularisieren und damit die Wiederverwendbarkeit nochmals deutlich zu erhöhen.

```
<%@ taglib uri="/WEB-INF/customlib.tld" prefix="cl" %>

<HTML>
<head>
  <title>JSF-Beispiel: Aktuelles Datum</title>
</head>

<body>
  <h3>Datumsanzeige in einer Jsp-Seite</h3>
  <i>Dieses Beispiel demonstriert, wie das aktuelle Datum
     mit Taglibs integriert werden kann:</i>
  <br><br>
  Wir haben heute den <cl:currentdate />
</body>
</HTML>
```

Listing 2.9: Datumsanzeige mit Hilfe einer Taglib

In Listing 2.9 ist wiederum das Beispiel einer Datumsanzeige zu sehen, allerdings diesmal unter Zuhilfenahme einer Taglib. Es ist deutlich zu erkennen, dass die Verwendung von Taglibs die Übersichtlichkeit stark erhöht und auch einem Nicht-Entwickler die Möglichkeit bietet, sich nach dem Baukastenprinzip eine Seite zusammenzubauen. Für die Datumsanzeige wird jetzt lediglich ein benutzerspezifisches Tag `<cl:currentdate />` verwendet. Die eigentliche Funktionalität ist nicht mehr in der JSP-Seite vorhanden, sondern in eine externe Java-Klasse ausgelagert. Somit kann auch ein in der Programmierung unerfahrener Seitenautor ein Tag für die Datumsanzeige ohne Probleme einbinden. Auf die Programmierung des Tags selbst wird in den folgenden Abschnitten näher eingegangen.

Basierend auf dieser Logik können projektspezifisch weitere Tag-Befehle entwickelt werden. Bei Bedarf ist es auch möglich, Attribute zu übergeben, die im Tag verarbeitet werden können. Damit wird auch wieder eine strikte Rollenverteilung geschaffen: Der Tag-Entwickler, der mittels Programmierung eine gewünschte Funktionalität bereitstellt, und der Webdesigner, der eine vorhandene Funktionalität in eine HTML-Seite ohne Programmieraufwand integrieren kann.

Der Tag-Library-Deskriptor

Benutzerspezifische Tags werden in einer gesonderten Datei definiert. In einem *Tag-Library-Deskriptor (TLD)* wird festgelegt, über welchen Namen ein benutzerspezifisches Tag in einer JSP-Seite angesprochen werden kann sowie welche Klasse für die Verarbeitung verwendet wird. Zudem können optional Attribute angegeben werden, mit denen ein Tag verwendet wird.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>JavaServer Faces - Customer Taglib</short-name>

  <tag>
    <name>currentdate</name>
    <tag-class>com.edu.jsf.bsp.tag.DateTag</tag-class>
    <body-content>empty</body-content>
  </tag>

  <tag>
    <name>greetings</name>
    <tag-class>com.edu.jsf.bsp.tag.GreetingsTag</tag-class>
    <body-content>empty</body-content>
      <attribute>
        <name>name</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
      </attribute>
  </tag>

</taglib>
```

Listing 2.10: Beispiel eines Tag-Library-Descriptors

In Listing 2.10 ist ein Beispiel für ein Tag-Library-Deskriptor (TLD) dargestellt. Darin werden zwei Tags definiert, von denen das erste keine Attribute hat, das zweite dagegen ein Attribut verwendet. Dieses ist zugleich zwingend vorgeschrieben (`required` hat den Wert `true`).

Nachdem die Details der benutzerspezifischen Tags im Tag-Library-Deskriptor festgelegt wurden, kann dieser in eine JSP-Seite eingebunden werden:

```
<%@ taglib uri="/WEB-INF/customlib.tld" prefix="cl" %>
```

Bei der Einbindung einer Tag-Bibliothek müssen zwingend die Attribute `uri` und `prefix` angegeben werden. Im Attribut `uri` wird geregelt, an welcher Stelle die Tag-Bibliothek zu finden ist. Hier kann wahlweise eine relative oder eine absolute Adresse angegeben werden. Die Angabe eines Präfix verhindert, dass es unter Umständen zu Namenskonflikten kommen kann. Dies kann beispielsweise dann der Fall sein, wenn zwei Tag-Bibliotheken das gleiche Tag beinhalten. Durch das Präfix, also durch Angabe des Namensraumes, wird die Angabe eines Tags wiederum eindeutig.

```
<cl:currentdate />
```

Listing 2.11: Einbindung eines Tags in einer JSP-Seite

In Listing 2.11 wird das Tag `currentdate` in eine JSP-Seite eingebunden. Durch das Präfix `cl` ist klar geregelt, dass das Tag in der Bibliothek *customlib.tld* zu finden ist.

Implementierung der Tag-Klasse

Nachdem neue Tags im Tag-Library-Deskriptor hinterlegt wurden und auch bereits das Tag in eine JSP-Seite integriert wurde, wird im Folgenden exemplarisch die Implementierung selbst genauer vorgestellt. Im Normalfall wird natürlich zunächst das Tag ausprogrammiert, bevor es verwendet wird. Zur Darstellung der Funktionsweise wurde jedoch dies in diesem Fall umgedreht.

Eine Tag-Klasse ist grundlegend nichts anderes als eine normale Java-Klasse, die eines der drei Interfaces `Tag`, `IterationTag` oder `Body-Tag` implementiert. Zusätzlich existieren bereits so genannte *Adapter-Klassen*, die diese Interfaces bereits implementieren und von denen gegebenenfalls abgeleitet werden kann. Die Klasse, die für das `currentdate`-Tag in Listing 2.9 zuständig ist, ist von der Adapterklasse `TagSupport` abgeleitet und überschreibt lediglich eine Methode, in der das aktuelle Datum in die Seite ausgegeben wird.

```
package com.edu.jsf.bsp.tag;

import java.text.DateFormat;
import java.util.Date;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

/**
 * Tag-Klasse für eine Datumsanzeige
 */
public class DateTag extends TagSupport {

  public int doStartTag() throws JspException {
    try {
```

```
      Date curDate  = new Date();
      DateFormat df = DateFormat.getDateInstance();
      String formattedDate = df.format( curDate );
      JspWriter out = pageContext.getOut();
      out.print( formattedDate );
    } catch ( Exception exc ) {
      throw new JspException("Fehler in DateTag.doStartTag()");
    } // catch
    return SKIP_BODY;
  }
}
```

Listing 2.12: Tag-Klasse für das currentdate-Tag

Je nach Art des Tags existieren unterschiedliche Adapterklassen, von denen bei Bedarf mehrere Methoden überschrieben werden können.

Natürlich können die gezeigten Beispiele keine umfassende Einführung in das Konzept der Taglibs sein. Vielmehr soll der Zusammenhang zwischen den Themen Servlets, JSPs und Taglibs deutlich werden. Für den in diesen Technologien noch etwas unerfahrenen Entwickler soll zudem ein Gefühl dafür geschaffen werden, dass für die Konzeption einer Webanwendung einiges an Know-how notwendig ist. Durch Einsatz von JavaServer Faces wird jedoch die Möglichkeit gegeben, dass auch in Webtechnologien unerfahrene Entwickler schnell Anwendungen realisieren können, ohne die Basistechnologien im Detail kennen zu müssen.

2.4.2 Standardisierte Taglibs – die JSTL

Mit Taglibs hatten Entwickler erstmals die Möglichkeit, Tags zu definieren, die für das jeweilige eigene Projekt nützlich waren. Doch wurden auch zunehmend Tags entwickelt, die Routineaufgaben erledigten, die fast in jedem Projekt zu finden waren. Als Beispiel seien eine Iteration über einen Vector oder ein Zugriff auf eine relationale Datenbank genannt. Diese Anforderungen waren zwar in vielen Projekten ähnlich, doch jeder Entwickler hatte häufig seine eigenen Tag-Bibliotheken entwickelt und diese eingesetzt. Um hier wieder zu einem gemeinsamen Standard zu gelangen, wurde die *JSTL*, die *Java Standard Tag Library*, definiert. Die JSTL selbst wiederum beinhaltet unterschiedliche Bereiche, die sich jeweils einem gewissen Thema widmen:

Bereich	Funktionen / Funktionsbereiche	Präfix
Core	Unterstützung von Variablenabfragen und Deklarationen sowie bei Exceptions Funktionen für den Steuerfluss. Es werden einfache Bedingungen (c:if) oder auch Auswahlbedingungen (c:when) zur Verfügung gestellt.	c

Tabelle 2.3: Funktionsbereiche der JSTL

Bereich	Funktionen / Funktionsbereiche	Präfix
	Url-Funktionen, mit denen beispielsweise auf eine andere Adresse weitergeleitet werden kann	
	Verschiedenes	
Xml	Core	x
	Funktionen für den Steuerfluss, die sich auf eine Verarbeitung einer Xml-Datei beziehen	
	Verschiedenes	
I18n	Unterstützung von Locales	fmt
	Formatierung von Meldungen	
	Zahlen- und Datumsformate	
Datenbank	SQL-Zugriffe können damit über Tag-Befehle ausgeführt werden.	sql
Funktionen	Weitere Funktionen z. B. für die Stringmanipulation	fn

Tabelle 2.3: Funktionsbereiche der JSTL (Fortsetzung)

Bei der Entwicklung von JSF-Seiten (Achtung, F, nicht P) wird oftmals auf Funktionen der JSTL zurückgegriffen. Daher stellt JSF keine erweiterte JSTL zur Verfügung, sondern integriert Funktionen daraus in das eigene Framework. Ein tiefer gehendes Verständnis der JSTL ist daher bei der Entwicklung von JSF-Anwendungen von Vorteil. Im Folgenden wird daher auf einige wichtige Tags der JSTL (auch im Hinblick auf JSF) eingegangen und deren Anwendung exemplarisch dargestellt.

Core-Tags

Bevor Tags aus der JSTL verwendet werden können, muss zunächst einmal die entsprechende Tag-Bibliothek in eine JSP-Seite eingebunden werden.

```
<%@ taglib uri="http://java.sun.com/jsp/JSTL/core" prefix="c" %>
```

Listing 2.13: Einbindung der Core-Tag-Bibliothek

Es ist üblich, die Core-Bibliothek mit dem Präfix `c` einzubinden. Damit können alle Core-Tags in dieser JSP-Seite aufgerufen werden. Natürlich ist es auch möglich, verschiedene Tags miteinander zu kombinieren. Dabei muss jedoch darauf geachtet werden, die entsprechenden Tag-Bibliotheken einzubinden.

Die JSTL Core-Tags regeln grundlegende Funktionen für eine Variablenverarbeitung sowie zur Kontrolle eines Steuerflusses. In Java existieren dazu beispielsweise if-Statements oder auch das Exception Handling, das ebenfalls den Ablauf in einem Programm behandelt. Genau diese Elemente gibt es auch in der JSTL in Form von Tags.

Im Zusammenspiel der Core-Tags mit der *JSP Expression Language (EL)* können Variablen innerhalb einer JSP-Seite definiert, angezeigt und in Ausdrücken verwendet werden, ohne dass hierfür JSP-Skriplets oder JSP-Ausdrücke verwendet werden müssen.

```
<c:set var="dummyVar" scope="session" value="Hallo Welt"/>
Ausgabe: ${dummyVar}
```

Listing 2.14: Variablenzugriff via JSTL

In Listing 2.14 wird über das <c:set>-Tag eine Variable dummyVar definiert, die über die Expression Language ausgegeben wird. Die JSP Expression Language ist eine Erweiterung in der JSP-Spezifikation seit Version 2.0, die es ermöglicht, auf einfache Weise auf JavaBeans in JSP-Seiten zuzugreifen sowie einfache Ausdrücke und Abfragen zu erstellen. Der Ausdruck ${dummyVar} ist ein EL-Ausdruck, der auf die Variable dummyVar zugreift.

Basierend auf einer einfachen Variablendeklaration kann damit auch ohne eine umständliche Skriplet-Lösung eine elegante Abfrage realisiert werden:

```
<c:set var="warning" scope="session" value="true" />
<c:if test="${warning == true}">
  Warnhinweis
</c:if>
```

Listing 2.15: Bedingung mit der JSTL

Auch in diesem Beispiel ist wieder ein Zusammenspiel der JSTL und der EL gegeben. Natürlich muss eine Variable, auf die in einer Bedingung zugegriffen wird, nicht zwangsläufig über eine <c:set>-Syntax in der Seite deklariert werden. Vielmehr kann über dieselbe Syntax auch auf JavaBeans, die in der Seite gültig sind, zugegriffen werden.

Im Gegensatz zu Java existiert beim <c:if>-Tag kein else-Zweig, um eine Entweder-oder-Abfrage realisieren zu können. Hierfür kann ein <c:choose>-Tag verwendet werden.

Die meisten Operatoren im <c:if>-Tag entsprechen denen der Java-Syntax. In Listing 2.15 wurde der Vergleichsoperator == verwendet. Weitere Operatoren sind der Ungleich-Operator !=, der Kleiner- und Größer-Operator (< und >) sowie der Kleiner-Gleich und der Größer-Gleich-Operator (<= und >=).

Es existiert zusätzlich im if-Tag eine weitere Variante, um das Ergebnis der Bedingung direkt einer Variablen zuzuweisen. Diese Form kommt ohne einen Body-Block in der eigentlichen if-Anweisung aus.

```
<c:set var="counter" scope="session" value="8" />
<c:if test="${counter>7}" var="result" />
Das Ergebnis der Counter-Abfrage lautet:
<c:out value="${result}" />
```

Listing 2.16: if-Anweisung

Wie in Listing 2.16 abgebildet, wird in der `if`-Bedingung das Ergebnis der Abfrage direkt der Variable `result` zugewiesen. Es erfolgt keine direkte Abarbeitung innerhalb des `if`-Blocks. Vielmehr wird an anderer Stelle auf die in der `if`-Bedingung gesetzte Variable zugegriffen.

Das `if`-Tag eignet sich zur Abfrage einer einzelnen Bedingung. Werden jedoch mehrere Alternativen abgefragt, empfiehlt sich die Verwendung des `choose`-Tags. Dieses kann am ehesten mit einer `switch/case`-Anweisung verglichen werden. Innerhalb eines `choose`-Tags können mehrere Optionen mittels des `when`-Tags abgefragt werden. Wurde eine Bedingung erfüllt, erfolgt ein Rücksprung aus dem `choose`-Block heraus.

```
<c:set var="city" scope="session" value="stuttgart" />
<c:choose>
  <c:when test="${city=='hamburg'}">
    Ausgewählte Stadt ist Hamburg.
  </c:when>
  <c:when test="${city=='stuttgart'}">
    Ausgewählte Stadt ist Stuttgart.
  </c:when>
  <c:when test="${city=='frankfurt'}">
    Ausgewählte Stadt ist Frankfurt.
  </c:when>
  <c:otherwise>
    Ausgewählte Stadt unbekannt.
  </c:otherwise>
</c:choose>
```

Listing 2.17: Verwendung des choose-Tags

In Listing 2.17 ist das `choose`-Tag dargestellt. Wurde kein passender `when`-Zweig gefunden, wird das `otherwise`-Tag aktiv, ansonsten erfolgt der Rücksprung aus dem `choose`-Tag heraus.

Sehr hilfreich ist auch oftmals die Möglichkeit, Iterationen anstatt in Skriplets über entsprechende Tags durchzuführen. Angenommen, es soll eine Mitgliederliste als Aufzählung in einer JSP-Seite ausgegeben werden. Die Namen sind in einem `Vector` gespeichert. Mittels Skriplets würde eine Seite wie in Listing 2.18: realisiert werden:

```
<h3>Iteration Variante 1</h3>
<ul>
<%
```

```
      Vector nameList = new Vector();
      nameList.addElement( "Mustermann" );
      nameList.addElement( "Müller" );
      nameList.addElement( "Meier" );

      Iterator i = nameList.iterator();
      String name = "";
      while ( i.hasNext() ) {
         name = i.next().toString();
%>
   <li><%=name %>
<%
   } // while
%>
</ul>
```

Listing 2.18: Beispiel für eine Iteration mittels Skriplets

Es fällt in Listing 2.18 auf, dass wieder sehr viel Java-Code in der JSP-Seite vorhanden ist. Dies führt wieder zu einer starken Vermischung von Java und HTML, was einer Lesbarkeit und damit einer Wartbarkeit der JSP-Seiten eher hinderlich ist. Im Gegensatz dazu ist in Listing 2.19 die Lösung der Aufgabe mittels Tags dargestellt.

```
<h3>Iteration Variante 2</h3>
<c:set var="nameList" scope="session"
       value="Mustermann, Müller, Meier" />
<ul>
  <c:forEach var="item" items="${nameList}">
    <li>${item}
  </c:forEach>
</ul>
```

Listing 2.19: Eine Iteration mit Tags

Es ist deutlich zu erkennen, dass eine Lösung der Iteration mittels der JSTL nicht nur wesentlich weniger Programmierarbeit ist, sondern zudem eine JSP-Seite übersichtlicher und kompakter darstellt.

Verwendung der Expression Language

Wie bereits im Abschnitt über die JSTL Core-Tags angedeutet, existiert seit der JSP-Spezifikation 2.0 die Möglichkeit, eine spezielle Skriptsprache innerhalb der JSTL zu verwenden. Damit haben Applikationsentwickler ein weiteres wichtiges Werkzeug zur Hand, JSP-Seiten schnell und ohne großen Programmieraufwand realisieren zu können.

Mit der Expression Language ist es auch möglich, auf Eigenschaften sowie auf Parameter zuzugreifen.

```
Aufrufmethode dieser Seite:
  <c:out value="${pageContext.request.method}" />
Aufrufparameter 'category' der Seite:
  <c:out value="${param.category}" />
```

Listing 2.20: Verwendung der EL

In Listing 2.20 wird einerseits auf die Aufrufmethode der Seite zugegriffen (im Normalfall POST oder GET) sowie auf einen Parameter des Aufrufs. Der Bezeichner `param` ist dabei fest vordefiniert, der Wert nach dem Punkt kann frei gewählt werden und muss in der Schreibweise dem in einer Url mitgegebenen Parameter entsprechen (bei GET). Wird somit eine Seite mit dem in Listing 2.20 gezeigten Codefragment mittels

```
seite.jsp?category=products
```

aufgerufen, enthält der Ausdruck `param.category` den Wert `products`.

Formatierungs-Tags

Formatierungen sind ein leidiges Thema in jeder Programmiersprache. Datumsangaben müssen auf verschiedene Weisen dargestellt sowie aus einer Benutzereingabe ausgelesen werden. Auch Ganzzahlen sowie Dezimalzahlen müssen auf unterschiedlichste Art und Weise formatiert werden. Hierbei bietet die JSTL ebenfalls Unterstützung an.

Vor Verwendung von Formatierungs-Tags muss hierfür ebenso wieder die notwendige Tag-Bibliothek miteingebunden werden.

```
<%@ taglib uri="http://java.sun.com/jsp/JSTL/fmt" prefix="fmt" %>
```

Der übliche Namensraum für Formatierungs-Tags ist `fmt`, der nach Möglichkeit auch beibehalten werden sollte.

Für die Formatierung von Zahlen, Prozentwerten und Währungen gibt es das `formatNumber`-Tag. Zur genauen Beschreibung des gewünschten Formats existieren eine Reihe von Attributen, mit denen das Ausgabebild beeinflusst werden kann.

```
<fmt:formatNumber type="number" groupingUsed="true"
 minFractionDigits="2" value="10234.5" />
```

Listing 2.21: Formatierung von Zahlen

Wird die Anweisung in Listing 2.21 ausgeführt, wird als Ausgabe Folgendes geliefert:

```
10.234,50
```

Der Tausenderpunkt erfolgt aufgrund der Anweisung `groupingUsed="true"`, die Anzahl der Nachkommastellen wird über `minFractionDigits` gesteuert.

Attribut	**Beschreibung**
`type`	Ausgabetyp der Formatierung (number, currency, percent)
`pattern`	Für die Formatierung von Zahlen kann ein benutzerspezifisches Muster angegeben werden.
`currencyCode`	Währungsbezeichner
`currencySymbol`	Währungssymbol für die Formatierung von Währungen
`groupingUsed`	Verwendung eines Tausenderpunktes
`maxIntegerDigits`	Maximale Anzahl von Zahlen vor dem Komma
`minIntegerDigits`	Minimale Anzahl von Zahlen vor dem Komma (gegebenenfalls wird mit 0 aufgefüllt)
`maxFractionDigits`	Maximale Nachkommastellen
`minFractionDigits`	Minimale Nachkommastellen (gegebenenfalls wird mit 0 aufgefüllt)
`var`	Das Ergebnis der Formatierung kann einer Variablen zugewiesen werden.
`scope`	Der Gültigkeitsbereich der Variable

Tabelle 2.4: Attribute für das formatNumber-Tag

Gerade bei Zahlen wird das `pattern`-Attribut häufig angewandt, da hiermit jegliche Ausgabeform definiert werden kann. Bei Währungen oder Prozentzahlen jedoch ist das Attribut ohne eine Auswirkung.

Die Formatierung von Datumswerten ist ähnlich komplex. Hierfür steht das Tag `formatDate` bereit. Analog zur Formatierung von Zahlen gibt es auch für die Formatierung von Datumswerten im Tag selbst eine Anzahl an Attributen, um das gewünschte Ergebnis erzielen zu können.

```
<fmt:formatDate dateStyle="short"
     value="${<%=new java.util.Date() %>" />
```

Listing 2.22: Formatierung von Datumswerten

Als Ausgabe in Listing 2.22 erhält man

```
09.04.03
```

bzw. das aktuelle Datum in obiger Schreibweise.

Attribut	Beschreibung
type	Regelt, ob die Ausgabe ein Datum, eine Uhrzeit oder eine Kombination aus beidem darstellt (time, date, both).
dateStyle	Ein Datumswert kann in verschiedenen vordefinierten Stilen ausgegeben werden (short, medium, long, full).
timeStyle	Bei der Ausgabe eines Wertes als Uhrzeit kann auch wieder ein vordefinierter Stil verwendet werden (short, medium, long, full).
pattern	Für Sonderformate kann ein benutzerspezifisches Muster angegeben werden.
timeZone	Zeitzone
var	Das Ergebnis der Formatierung kann einer Variablen zugewiesen werden.
scope	der Gültigkeitsbereich der Variablen

Tabelle 2.5: Mögliche Attribute beim formatDate-Tag

Datenbank-Tags

Im Bereich der Datenbankanbindung stellt die JSTL hierfür ebenfalls leistungsfähige Tags zur Verfügung. Gerade für kleinere Projekte oder Prototypen bietet sich hiermit die Möglichkeit, einen einfachen Zugriff auf eine SQL-Datenbank via Taglibs zu ermöglichen. Der Vorteil ist, dass keine spezielle Datenbank-Engine oder separate Datenbank-Schicht mehr entwickelt werden muss. Vielmehr kann eine Select-Abfrage einfach in eine JSP-Seite integriert werden, ohne dass hierfür ein großer Programmieraufwand notwendig ist.

```
<sql:setDataSource
   var="expDb"
   driver="org.gjt.mm.mysql.Driver"
   url="jdbc:mysql://localhost/expdb"
   user="master"
   password="" />

<sql:transaction dataSource="${expDb}">
  <sql:query var="myresult">
    SELECT * FROM members
  </sql:query>
</sql:transaction>

<ul>
  <c:forEach items="${myresult.rows}" var="row">
    <li>${row.name}>
  </c:forEach>
<ul>
```

Listing 2.23: Verwendung der Sql-Befehle

In Listing 2.23 ist ein einfacher Datenbankzugriff dargestellt. Über die `setDataSource`-Anweisung wird einmalig eine Datenquelle definiert. In diesem Beispiel handelt es sich um eine MySQL-Datenbank mit dem Namen `expDb`. Über das Tag `transaction` wird eine Transaktion eingeleitet, in der über eine Select-Anweisung eine Abfrage der MEMBERS-Tabelle ausgeführt wird. Anschließend wird im `forEach`-Block über die Ergebniszeilen der Abfrage iteriert und das Ergebnis in Form einer Liste ausgegeben. Zur Verwendung der JSTL SQL-Befehle im Zusammenspiel mit JavaServer Faces existiert in 10.2 ein separater Abschnitt. Darin wird auch näher darauf eingegangen, ob die Verwendung der SQL-Befehle sinnvoll ist oder lieber eine andere Form der Datenbankanbindung verwendet werden soll.

2.5 Das Model-View-Controller-Entwurfsmuster

Entwurfsmuster gibt es schon, seit es Software gibt. Zwar wurde der Name *Design Pattern*, zu Deutsch *Entwurfsmuster*, erst in den letzten Jahren stark geprägt, dennoch gab es schon immer erfolgreiche Muster bei der Entwicklung von objektorientierten und prozeduralen Programmiersprachen, die innerhalb der Entwicklergemeinde erfolgreich weitergegeben und weiterentwickelt wurden. Entwurfsmuster sollen häufig auftretende Probleme und Fragestellungen beim Entwurf objektorientierter Software beschreiben und eine programmiersprachenneutrale Vorgehensweise zur Lösung dieses Problems anbieten. Entwurfsmuster beschreiben dabei eine Architektur zur Lösung einer Aufgabenstellung.

Das *Model-View-Controller-Entwurfsmuster (MVC)* im Speziellen soll eine Entwicklung und eine eventuelle spätere Erweiterung oder Anpassung einer graphischen Benutzeroberfläche (GUI) erleichtern. Des Weiteren soll eine Trennung der darzustellenden Daten von der eigentlichen Präsentation erfolgen. Es soll möglich sein, das Verhalten sowie die Daten zu verändern, ohne Eingriffe in die Darstellung selbst vornehmen zu müssen.

Gerade bei webbasierten JSP-Applikationen gab und gibt es immer noch Anwendungen, in denen die gesamte Logik in der View (also in der JSP-Seite) enthalten ist. Resultat solcher Entwicklungen sind oftmals Anwendungen, die u. U. zunächst erst einmal funktionieren, aber aufgrund ihrer Unübersichtlichkeit nur schwer wartbar sowie auch stark fehleranfällig sind. Gerade Entwickler, die sich neu in ein Projekt einarbeiten müssen, sehen dann oftmals nur sehr große und umfangreiche JSP-Seiten, in denen die gesamte Programmlogik mit den graphischen HTML-Elementen vermischt ist.

Sollten Design-Änderungen durch einen Webdesigner vorgenommen werden, war daher oftmals eine komplette Neuentwicklung der Anwendung notwendig. Ein Webdesigner konnte die komplexe Logik der JSP-Befehle nicht verstehen und gemäß einer neuen Layoutänderung umsetzen. Hier musste somit wieder ein erfahrener JSP-Experte die bisherige Logik in das neue Design-Konzept von neuem umsetzen.

Um genau solche Fehlentwicklungen zu vermeiden, stellt das MVC-Entwurfsmuster einen erprobten und lange bewährten Weg dar, wie eine saubere Trennung von Design, Datenhaltung und Programmlogik erfolgen kann. Die Komponenten, die beim MVC-Entwurfsmuster dabei verwendet werden, sind folgende:

- *Modell*: Das Modell ist für die Datenhaltung verantwortlich. Es beinhaltet noch keine große Programmintelligenz, sondern dient hauptsächlich der Speicherung von Daten, die in einer Webanwendung benötigt werden. Meist ist das Modell ein einfaches JavaBean.
- *View*: Die View ist die eigentliche Darstellung der Anwendung. Eine View ist in einer klassischen JSP-/Servlet-Entwicklung die JSP-Seite selbst. Wichtig ist jedoch, dass die View keine Logik enthält, sondern ausschließlich Darstellungsfunktionen übernimmt.
- *Controller*: Der Controller ist letzten Endes für die Steuerung der Anwendung verantwortlich. Er regelt den Navigationsfluss innerhalb der Anwendung und enthält die Logik, die für die Programmausführung selbst notwendig ist. Des Weiteren versendet der Controller Benachrichtigungen, sobald sich im Modell etwas geändert hat.

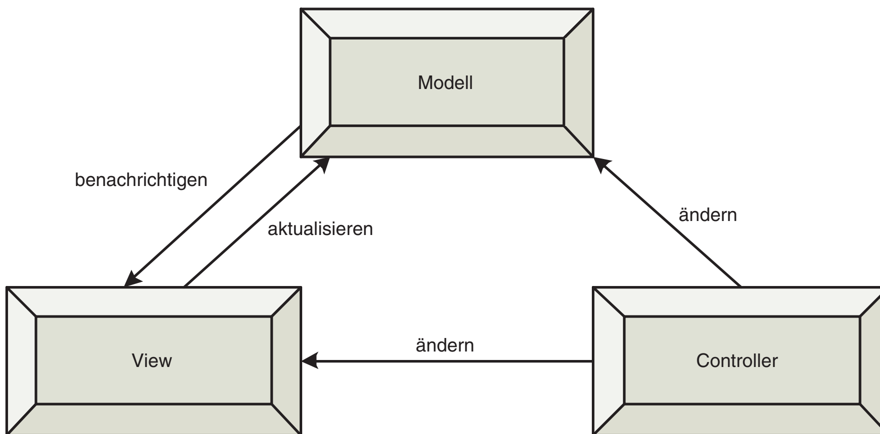


Abbildung 2.5: Zusammenspiel von Modell, View und Controller

In Abbildung 2.5 ist das Zusammenspiel der Komponenten *Modell*, *View* und *Controller* zu sehen. Das Modell stellt dabei ausschließlich die Daten bereit, die in der View verwendet werden können. Es ist jedoch vollkommen losgelöst von einer Darstellung und kann daher auch in einem anderen Kontext ohne Anpassung verwendet werden. Einzig bei einer Änderung der Daten verschickt das Modell eine Benachrichtigung, worauf sich eine graphische Oberfläche ihre aktualisierten Daten abholen kann. Wichtig ist, dass nicht das Modell die aktualisierten Daten direkt in eine Oberfläche stellt, da

das Modell nichts darüber weiß. Vielmehr holt sich die Oberfläche die neuen Werte selbstständig ab. Der Controller vermittelt quasi zwischen beiden Komponenten. Wird z.B. in der Oberfläche eine Aktion ausgelöst, weiß der Controller, wie er damit umzugehen hat. Gegebenenfalls werden daraufhin die Daten im Modell geändert, worauf wieder eine Benachrichtigung vom Modell an die Oberfläche versandt wird.

Die Vorteile dieser Modularisierung liegen in einer expliziten Aufgabenverteilung der einzelnen Komponenten. Das Modell kapselt dabei die Benutzerdaten unabhängig von der Darstellung selbst. Sollte somit z.B. statt einer JSP-Anwendung eine klassische Swing-Anwendung verwendet werden, können die Modell-Klassen ohne jegliche Änderung weiterverwendet werden. Denkbar wäre auch eine Erweiterung, in der auf einem Mobiltelefon eine Anwendung funktionieren soll. Dabei ist die Programmlogik dieselbe, lediglich die View wird nicht als HTML, sondern als WML ausgegeben. Gerade in solchen Beispielen wird der Vorteil des MVC-Musters deutlich.

Die View wiederum bezieht ihre Daten über das Modell. Dadurch, dass die Programmlogik und die Datenhaltung »ausgelagert« wurden, wird die View selbst wieder übersichtlich und wartbar. Bei einer konsequenten Ausrichtung der Anwendung auf das MVC-Pattern, wie es auch im JSF-Framework beschrieben ist, können Webdesigner auch ohne Programmierkenntnisse Erweiterungen und Änderungen in einer View vornehmen.

Vorteile eines Einsatzes von MVC:

- Es können mehrere Ansichten/Views desselben Modells existieren. Somit kann z.B. sowohl eine Swing- als auch eine JSP- oder WML-Oberfläche auf dasselbe Modell zugreifen.
- Durch den Controller ist jederzeit sichergestellt, dass alle Views die gleichen Daten anzeigen und somit eine synchronisierte Darstellung möglich ist.
- Durch die Trennung in Modell, View und Controller kann bei Bedarf eine View relativ schnell ausgetauscht werden, ohne dass die Anwendung komplett neu entwickelt werden muss.
- Das MVC-Pattern ist ein sehr bekanntes Muster. Neue Entwickler in einem Projekt können sich daher auch sehr schnell in vorhandenen Quellcode einarbeiten.

Nachteile eines Einsatzes von MVC

Dass ein Einsatz eines Entwurfsmusters auch gewisse Nachteile mit sich bringt, ist richtig. Im Folgenden sind kurz die häufig genannten Gründe aufgeführt, die eventuell gegen einen Einsatz von MVC sprechen können.

- Erhöhung der Komplexität durch eine Erhöhung der Klassen. Zwangsläufig muss eine gewisse Anzahl von neuen Klassen entwickelt werden, um die einzelnen Komponenten abbilden zu können.

- Eventuelle Performanceengpässe: Da ein Controller alle Views bei Änderungen im Modell benachrichtigt, kann es bei einer großen Anzahl von Views unter Umständen zu Performanceengpässen kommen. Sollte diese Gefahr bestehen, können jedoch im Anwendungsdesign im Vorfeld bereits Maßnahmen dagegen getroffen werden.
- Es ist ein erhöhter Planungs- und Programmieraufwand notwendig.

Zusammenfassend

Das MVC-Entwurfsmuster ist sehr wichtig, wenn nicht sogar das Wichtigste überhaupt im Bereich einer modernen Oberflächenentwicklung. Ein tiefer gehendes Wissen über MVC hilft nicht nur, die Architektur sowie die Konzepte von JavaServer Faces zu verstehen, sondern ermöglicht es einem Entwickler auch, die Überlegungen und Konzepte von MVC in eigene ähnliche Problemstellungen zu übertragen.

In den letzten Abschnitten wurden sowohl Vorteile als auch einige Nachteile von MVC gegenübergestellt. Zusammengenommen überwiegen jedoch die Vorteile des MVC-Patterns bei weitem. Da die Vorteile doch beachtlich sind, basiert das JSF-Framework (wie viele andere übrigens auch) auf dem MVC-Pattern.

2.6 Fazit

In diesem Kapitel wurde die Funktionsweise der Servlet- und JSP-Programmierung vorgestellt und Schwerpunkte wurden näher erläutert. Für die Entwicklung von Webanwendungen mit JavaServer Faces ist eine tiefer gehende Kenntnis der Servlet- und JSP-Programmierung zwar nicht unbedingt notwendig, es hilft jedoch die Funktionsweise in JSF zu verstehen.

Wichtig dagegen ist das Wissen über die JSTL. Die JSTL vereint wichtige Tags der JSP-Programmierung, die häufig in Webanwendungen auftauchen und daher standardisiert in der JSTL einem Entwickler zur Verfügung gestellt werden. Gerade auch die Kombination der JSTL mit JFS macht eine Anwendungsentwicklung einfacher und auch effizienter.

Weiterhin wurden die Architekturmodelle 1 und 2 der JSP-Spezifikation vorgestellt, wobei JavaServer Faces sich an das Modell 2 anlehnt und sogar in Teilbereichen noch darüber hinausgeht.

Da das Model-View-Controller-Entwurfsmuster auch in JavaServer Faces eine wichtige Rolle spielt, wurden das Prinzip und die Vorteile dieses Musters näher erläutert. Es wurde deutlich gemacht, welche Vorteile es mit sich bringt, eine klare Trennung von Modell, View und Controller zu haben.

3 Installation der JavaServer Faces-Demoanwendungen

Kapitelziel

Um eine erste JavaServer Faces-Anwendung »live« sehen und erleben zu können, sind in der Referenzimplementierung bereits einige Beispielanwendungen enthalten. Diese können ohne großen Installationsaufwand gestartet werden. Dieses Kapitel gibt daher eine erste kurze Einführung, wie die Beispielanwendungen der Referenzimplementierung installiert und ausgeführt werden können.

3.1 Vorbereitungen

Um eine JavaServer Faces-Anwendung ausführen zu können, wird natürlich grundlegend ein aktuelles *Java Development Kit* (JDK) von Sun benötigt. Alle im Buch vorhandenen Beispiele beruhen dabei auf der Version 1.4.2, damit sind auch die Beispielanwendungen der Referenzimplementierung lauffähig. Eine ältere JDK-Version kann nicht verwendet werden, aktuellere Versionen können natürlich jederzeit eingesetzt werden.

Die Version 1.0 der JavaServer Faces-Spezifikation sowie der Referenzimplementierung kann über die Webseite von Sun unter *http://java.sun.com/j2ee/javaserverfaces* heruntergeladen werden. JavaServer Faces benötigt zusätzlich das *Java Web Services Developer Pack*, aktuell in der Version 1.3. Künftig wird JavaServer Faces ebenfalls in diesem Paket mit ausgeliefert. Zum Zeitpunkt der Drucklegung dieses Buches besteht jedoch die Schwierigkeit, dass in der Version 1.3 des Web Services Pakets bereits eine frühe Testversion (ein so genanntes Early Access) enthalten ist. Daher müssen an dieser Stelle Ersetzungen mit den aktuellen Komponenten vorgenommen werden. Eine Anleitung, wie dies vorzunehmen ist, ist in der JavaServer Faces-Referenzimplementierung zu finden. Kurz zusammengefasst verläuft die Ersetzung der Early-Access-Version durch die aktuelle JavaServer Faces-Version in folgenden drei Schritten:

1. Installation des Web Services Pakets gemäß der Installationsanleitung (unter Windows steht ein Installationsprogramm zur Verfügung, das einen Benutzer durch die Installation begleitet)

2. Löschen des Verzeichnisses *JWSDP_HOME/jsf*
3. Auspacken der JavaServer Faces 1.0 Zip-Datei und Umbenennen des Verzeichnisses in *jsf*

Das Java Web Services Developer Pack ist ebenfalls über die Webseite von Sun unter

http://java.sun.com/webservices/downloads/webservicespack.html

zu beziehen bzw. ist ebenfalls der CD beigefügt ((TODO klären, ob das überhaupt erlaubt ist, ansonsten Satz umformulieren)). Mit enthalten ist bereits ein Apache Tomcat Server, der durch das Installationsprogramm mitinstalliert wird. Zum einfachen Starten und Testen der mitgelieferten Beispielanwendungen ist es ratsam, den Tomcat-Server aus dem Web Services Paket zu verwenden. Vorteil ist, dass die benötigten jar-Bibliotheken bereits in den vorgesehenen Verzeichnissen liegen sowie auch alle weiteren Komponenten in der passenden Version vorhanden sind. Es ist natürlich auch möglich, separat Tomcat in einer anderen Version zu installieren, dieses Verfahren wird in Kapitel 5.2 Einrichten der Entwicklungsumgebung erläutert.

Nachdem die Basisstruktur geschaffen wurde, können die Beispiele aus dem *JWSDP_HOME/jsf/samples*-Verzeichnis in das *JWSDP_HOME/webapps* übernommen werden. Dazu genügt es, eine entsprechende war-Datei zu kopieren. Tomcat sorgt selbst dafür, dass die Archiv-Datei ausgepackt und installiert wird.

3.2 Start der Beispiele

Nachdem das Java Development Kit, das Web Services Developer Pack und JavaServer Faces selbst installiert wurden sowie die gewünschten war-Dateien in das *webapps*-Verzeichnis kopiert wurden, kann der Tomcat-Server im Verzeichnis *JWSDP_HOME/bin* gestartet werden. Dazu steht für Windows die Datei *startup.bat* und unter Unix/Linux die Datei *startup.sh* zur Verfügung. Nachdem der Server gestartet wurde, kann bereits die Anwendung über einen Browser aufgerufen werden.

In Abbildung 3.1 wurde die Datei *jsf-cardemo.war* installiert und gestartet. Nach wenigen Sekunden, in denen die JSF-Seite kompiliert wird, erscheint die gezeigte Startseite und es kann mit der Anwendung gearbeitet werden.

Über dieselbe Vorgehensweise können natürlich auch die anderen Beispiele gestartet werden. Interessant auch, wenn Sie später nachschlagen wollen, wie benutzerspezifische Komponenten entwickelt werden können, ist die Demoanwendung *jsf-components*. Es sind darin Beispiele enthalten, wie z.B. mittels JavaServer Faces eine Baumkomponente oder auch eine Listenausgabe umgesetzt werden kann.

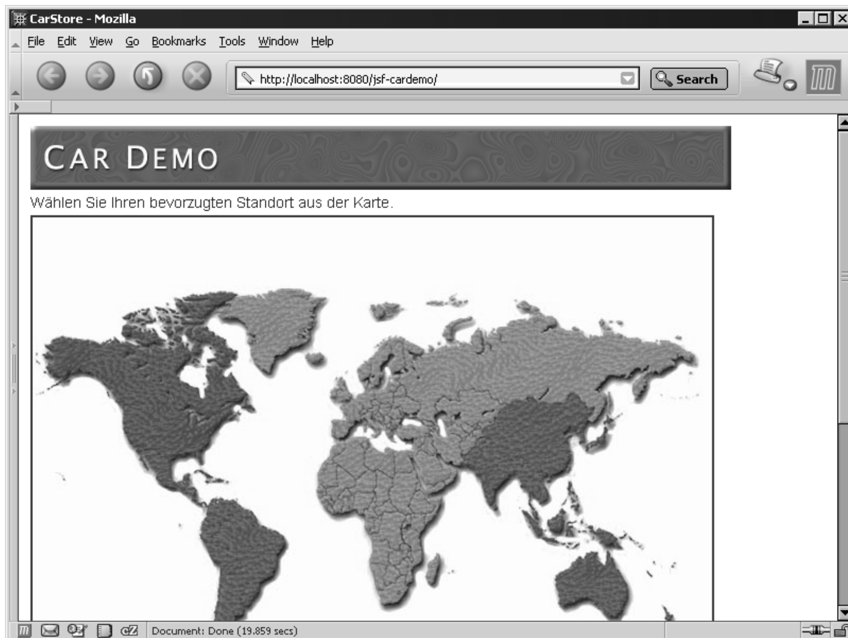


Abbildung 3.1: Demoanwendung Car-Demo

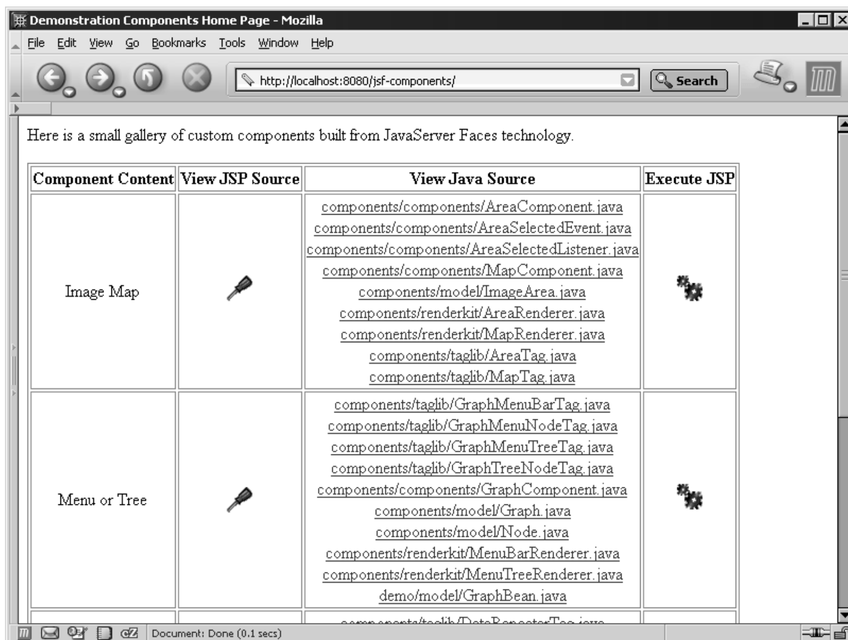


Abbildung 3.2: JavaServer Faces-Komponenten

Zu den Komponenten, die im Beispiel *jsf-components* zu sehen sind, muss angemerkt werden, dass diese nicht zum JSF-Standard gehören. Diese Komponenten gehören zu den benutzerdefinierten Komponenten, die im Gegensatz zu den Standardkomponenten durch jeden Entwickler nach Belieben erzeugt werden können.

3.3 Fazit

Dieses Kapitel hat aufgezeigt, wie die Beispielanwendungen, die mit der Referenzimplementierung von JavaServer Faces ausgeliefert werden, installiert und aufgerufen werden können. Es wurde die Installation des Web Services Developer Packs erklärt und auf die Problematik hingewiesen, dass das in der Version 1.3 enthaltene JSF-Verzeichnis durch das aktuelle Verzeichnis der JavaServer Faces-Referenzimplementierung in der Version 1.0 zu ersetzen ist.

Ferner wurde aufgezeigt, wie mit Hilfe der war-Archive die Demoprogramme installiert und aufgerufen werden können. In Kapitel 5 werden sie lernen, wie Sie eine erste eigene JavaServer Faces-Anwendung erstellen. Dazu sind im Vorfeld natürlich nochmals einige Installationsschritte nötig (u.a. das Installieren der Entwicklungsumgebung), die jedoch auch wieder schrittweise erläutert werden.

4 Einführung in JavaServer Faces

Kapitelziel

Ziel dieses Kapitels ist es, grundsätzliche Fragen zu JavaServer Faces zu beantworten und JavaServer Faces als Technologie einschätzen zu können. Wichtig ist auch die Begriffsbestimmung, um zu wissen, in welchem Bereich JavaServer Faces einzuordnen ist.

Webanwendungen, Portale und Frameworks sind häufig auftauchende Begriffe, die keine klare Abgrenzung zueinander haben. Daher wird in diesem Kapitel versucht, hier eine etwas konkretere Definition zu erarbeiten.

Des Weiteren wird in diesem Kapitel bereits das Rollenkonzept von JavaServer Faces vorgestellt. Ein Vorteil von JavaServer Faces ist es, dass mit verteilten Rollen an einer Webanwendung gearbeitet werden kann. Auf die verschiedenen Rollen wird daher genauer eingegangen.

Um die Verarbeitung eines Requests in JavaServer Faces näher zu beleuchten, wird der Lebenszyklus einer JavaServer Faces-Seite näher beleuchtet sowie die verschiedenen Phasen detailliert beschrieben.

4.1 Was ist JavaServer Faces?

JavaServer Faces ist (oder sind, darüber ist man sich noch nicht ganz einig) in erster Linie ein Framework zur Erstellung von graphischen Benutzeroberflächen für Webanwendungen. Dabei ist der Zusatz *Benutzeroberflächen* ein wichtiger Hinweis, in welche Richtung JSF zielt: JavaServer Faces ist *kein* Framework, das die Erstellung und den Ablauf von kompletten Webanwendungen beschreibt und spezifiziert, vielmehr wird der Ablauf auf Seiten der Benutzeroberfläche unterstützt und standardisiert. Webentwickler sollen mit Hilfe von JavaServer Faces schneller und effektiver Webanwendungen erstellen können. Dazu liefert JSF bereits eine große Anzahl von UI-Komponenten (User-Interface Komponenten) mit, die direkt in eigene Anwendungen eingebaut und verwendet werden können. Auch bereits etablierte Konzepte und Entwurfsmuster – wie z. B. das Eventhandling oder auch das Model-View-Controller-Prinzip – sollen einem Webentwickler helfen, ohne große Einarbeitungszeit moderne und stabile Webanwendungen entwickeln zu können.

4.2 Begriffsbestimmung – Webanwendung, Framework und Portale

Im Zusammenhang mit JavaServer Faces, aber auch in der allgemeinen Presse tauchen immer wieder die Begriffe Webanwendung (oder Webapplikation), Framework und Portale auf. Auch nach dem großen Internet-Hype der vergangenen Jahre sind es immer häufiger Marketingschlagworte, die eine neue Technologie in den Vordergrund rücken. Es gibt jedoch zwischen den vielen Begriffen entscheidende Unterschiede, die auch im Hinlick auf die Einordnung von JavaServer Faces relevant sind.

Webanwendungen

Eine Webanwendung ist zunächst einmal nichts anderes als eine Anwendung, die im Web lauffähig ist. Der Begriff entstammt der Zeit, in der das Internet für die Abwicklung von komplexen Anwendungen noch nicht verwendet wurde. Klassische Client-/ Server- Anwendungen liefen ausschließlich auf einem Client-PC sowie auf einem oder mehreren zentralen Servern. Im Normalfall musste der Client speziell auf einem PC installiert werden, um im Gesamtnetz einer Anwendung arbeiten zu können. Als dann erste Anwendungen im Internet bereitgestellt wurden, sprich mit einer Html-Oberfläche ohne Installation auf einem PC von überall aus im Internet zugreifbar waren, wurde der Begriff der Webanwendung geprägt. Somit bezeichnet eine Webanwendung einen Typ von Anwendung, die über eine Web-(meist Html-)Oberfläche über einen Internetbrowser bedienbar ist.

In der Praxis kommt es zumeist zu einem Zusammenspiel von klassischen Anwendungen und Webanwendungen. So wird eine typische Client-/Server-Anwendung im lokalen Netz wie gewohnt betrieben, bietet häufig jedoch auch eine Webschnittstelle. Diese kann derart aufgebaut sein, dass die komplette Anwendungsfunktionalität auch über das so genannte Webfrontend zur Verfügung steht oder auch nur ein gewisser Teilausschnitt. Letzterer Ansatz hat den Vorteil, dass nicht eine komplette Anwendung auf eine Webtechnologie umgestellt, sondern lediglich eine Anbindung geschaffen werden muss. Zumal ist es nicht immer sinnvoll, jegliche Form von Anwendung auf eine Internettechnologie umzustellen.

Ein typisches Beispiel einer frühen Webanwendung, die auf eine (Großrechner-)Struktur zurückgegriffen hat, ist das Homebanking via Internet. Die Anwendung selbst sammelte Daten über eine Webanwendung und gab diese an den eigentlichen Bankrechner (meist ein Großrechnersystem) weiter. Die Rückmeldung davon wurde wiederum in der Webanwendung (im Webfrontend) dargestellt.

Frameworks

Nachdem geklärt ist, *was* eine Webanwendung genau ist, kann die Frage gestellt werden, *wie* eine Webanwendung entsteht. Reine Präsentationsauftritte im World Wide Web (WWW) sind im Normalfall eine Sammlung statischer Html-Seiten ohne jegliche Funktionalität. Um jedoch eine Webanwendung zu realisieren, müssen Seiten dynamisch generiert sowie häufig auch eine Datenbank mitangebunden werden. Oftmals sind auch Themen wie eine Benutzerverwaltung oder ein Session-Management relevant.

Im Beispiel eines Online-Shops kann ein Besucher z. B. Bücher nach seinen persönlichen Kriterien suchen. Die Ergebnisliste ist eine zur Laufzeit erzeugte Seite (realisiert z. B. mit Servlets oder JSPs). Damit die Einstellungen eines Benutzers während seines gesamten Besuchs im Online-Shop gespeichert bleiben, kann dies mit Hilfe des Session-Managements realisiert werden. Dabei wird im Browser des Besuchers ein so genannter Cookie gespeichert, über den ein Besucher künftig identifizierbar ist. Auf Seiten des Servers des Online-Shops können nun Daten, die zu dieser Kennung des Cookies gehören, im Hauptspeicher, also in der Session, gespeichert werden.

Gerade für die Entwicklung von Webanwendungen hat sich in den letzten Jahren extrem viel getan. So waren erste Webanwendungen technisch betrachtet nichts anderes als einfache CGI-Programme, die kleinere Funktionalitäten abbildeten und das Ergebnis als dynamisch generierte Seite zurücklieferten. Auch Sun hat mit Java in diesem Bereich viele Entwicklungsschritte durchgeführt.

Doch obwohl mittlerweile verschiedenste Technologien wie z. B. JSP, ASP oder PHP, um nur wenige zu nennen, auf dem Markt existieren, hat sich vielfach die Erkenntnis durchgesetzt, dass mit einer Technologie allein noch keine erfolgreiche Webanwendung erstellt werden kann. Zu aufwändig ist es heutzutage oftmals, mit der zur Verfügung stehenden (Basis-) Technologie umfangreiche Webanwendungen zu realisieren. Daher wird immer häufiger auf bestehende *Frameworks* zurückgegriffen.

Frameworks sind ein anwendungsneutrales Rahmenwerk, das bereits Grundelemente sowie eine Architektur bereitstellt. Somit fallen für die eigentliche Anwendungsentwicklung weniger Programmier- und Planungsaufwendungen an. Zumal ist die Architektur, die von Frameworks häufig vorgegeben, bereits vielfach getestet und von vielen Experten entwickelt worden. Gerade auch im Open-Source-Bereich existieren eine Vielzahl sehr guter und ausgereifter Frameworks. Häufig taucht in der Literatur auch der Begriff der *Best Practices* auf. Damit ist gemeint, dass aufgrund zahlreicher Projekte und Lösungsansätze sich einige Techniken und Herangehensweisen an eine Problemstellung als »beste« Alternative im Laufe der Zeit herauskristallisiert haben. Diese gesammelten Erfahrungen sind häufig in Frameworks enthalten und geben einem Entwickler somit ein mächtiges und umfangreiches Basispaket an die Hand.

Doch nicht nur aus technologischen Gründen macht es Sinn, auf bestehende Frameworks aufzusetzen. Vor allem auch der Zeitfaktor spielt eine große Rolle. Da bei einem Einsatz von Frameworks die Hauptentwicklungszeit bei der eigentlichen Geschäftslogik liegt, wird für eine Basisentwicklung viel Zeit eingespart, was wiederum in kürzeren Projektlaufzeiten und damit auch in geringeren Projektkosten sichtbar wird.

Portale

Viele Anwendungen im Internet in den vergangenen Jahren waren erstaunlicherweise immer Portale. Damit wurde der Portalbegriff leider zum Teil auch wieder zu einem Marketing-Unwort. Doch wie unterscheidet sich (wenn es überhaupt einen Unterschied gibt) ein Portal von einer Webanwendung?

Möchte man die Vielzahl der Webanwendungen und Webangebote im Internet unterscheiden, tauchen schnell Begriffe wie Shop, Online-Auktionen, Community, Marketplace oder Portal auf. Technisch betrachtet unterscheiden sich diese Anwendungstypen nicht wesentlich. Die erzeugten und einem Besucher präsentierten Seiten werden dynamisch aufbereitet, und fast immer ist eine Datenbank für die Speicherung und Bereitstellung von Daten im Einsatz.

Genauer betrachtet existieren von der Anwendungsseite her allerdings grundlegende Unterschiede: Während ein *Shop* primär zum Einkauf von Waren über das Internet bestimmt ist, existiert in einer *Community* eine virtuelle Gemeinschaft mit bestimmten Gemeinsamkeiten. So existieren Communities für Singles, die auf Partnersuche sind, ebenso wie es auch Communities zum Thema JavaServer Faces gibt, die sich ausschließlich mit Wissens- und Erfahrungsaustausch zum Thema JSF beschäftigen. Ein *Marketplace* ist allgemeiner gehalten als ein Shop, in einem Marketplace können ähnlich wie auf einem reellen Markt verschiedene Firmen Waren anbieten und potenzielle Käufer verschiedenste Produkte suchen und erwerben. Im Gegensatz dazu ist bei *Online-Auktionen* nicht ein direkter Kauf möglich, sondern es wird ein Produkt oder eine Ware versteigert. Den Zuschlag erhält derjenige mit dem höchsten Gebot.

Den Portalbegriff einzugrenzen ist schwieriger. So ist ein *Portal* zunächst einmal eine Art Zugangstor. Oftmals wird der Portalbegriff bei großen Online-Diensten wie T-Online oder AOL verwendet, deren Homepage eine Art Zugangstor zu einem riesigen Angebot an Diensten und Servicen darstellt.

In der Praxis der vergangenen Jahre vermischen sich jedoch die Begriffe häufig. So weist ein Portal auch immer Community-Eigenschaften auf, wie auch die Möglichkeit, bestimmte Produkte online zu kaufen. Auch in Shops können vielfach Diskussionen stattfinden, genauso wie auch Foren und Chats häufig darin zu finden sind.

Was Sie genau mit Hilfe von JavaServer Faces machen, liegt bei Ihnen. Mit JSF haben Sie ein mächtiges Framework zur Hand, mit dem Sie (fast) jede Art von Webanwendung realisieren können – vom einfachen Newsticker bis hin zu einem komplexen und viel besuchten Portal im World Wide Web.

4.3 JSF und andere Frameworks

Im letzten Abschnitt wurde bereits eine Begriffsbestimmung für Frameworks angesprochen. Um die Frage beantworten zu können, wie sich JSF mit anderen Frameworks vergleichen lässt, erfolgt zunächst eine detailliertere Betrachtung, was ein Framework überhaupt auszeichnet. Nimmt man als ersten Anlaufpunkt das Deutsch-Englisch-Lexikon (oder für alle Onlinejunkies *http://dict.leo.org*) zu Hilfe, erhält man für den Begriff *Framework* folgende Übersetzungen angeboten (auszugsweise):

- Gerippe
- Gerüst
- Rahmen
- System

Oftmals finden Sie in der Literatur auch die Beschreibungen, ein Framework sei ein Programmiermodell oder auch ein Software-Architekturmodell. Sowohl die Übersetzungen als auch die häufig verwendeten Bezeichnungen treffen alle genau auf diese Eigenschaften, die ein gutes Framework tatsächlich ausmachen, zu.

Ein Framework ist kein fertiges Softwaresystem, sondern stellt einem Entwickler ein Rahmenwerk zur Verfügung, mit dessen Hilfe er schnell und zielgerichtet eine Softwareaufgabe lösen kann. Oftmals liefert ein Framework bereits Module und Bausteine mit, die ein Entwickler optional verwenden oder erweitern kann.

Dies ist sicherlich eine recht breit gefasste Erklärung des Begriffs Framework, und Sie finden im Internet eine fast nicht mehr überschaubare Anzahl an Frameworks. Wobei natürlich nicht jedes Framework für ein Projekt geeignet ist. Folgende Fragen sollten dabei im Vorfeld geklärt werden, um ein passendes Framework zu finden:

- Was für eine Art von Anwendung soll entwickelt werden (Webanwendung, Client/Server-Anwendung, Swing-Anwendung, ...)?
- In welcher Programmier-/Skriptsprache wird die Anwendung entwickelt (Java, C#, PHP, Perl ...)?
- Wie groß ist das Entwicklerteam im eigenen Projekt (Ein-Mann-Projekt oder Großprojekt mit mehreren hundert Entwicklern)?
- Welche Funktionalitäten sollten nach Möglichkeit durch ein Framework abgedeckt wer-den (Oberflächenkonzept, Datenbankanbindung, Benutzer- und Sessionverwaltung, ...)?
- Welche Unterstützung seitens des Herstellers oder seitens einer Community wird benötigt?

- Wie wichtig ist eine ständige und langfristige Weiterentwicklung des Frameworks (auch hinsichtlich der Investitionssicherheit)?
- Wie verbreitet ist das Wissen über das Framework (kann davon ausgegangen werden, dass neue Entwickler sich bereits mit diesem Framework befasst haben)?

Diese Liste ließe sich natürlich noch stark erweitern. Ziel soll es nicht sein, an dieser Stelle eine Entscheidungsmatrix zur Auswahl eines geeigneten Frameworks zu entwerfen. Vielmehr soll aufgezeigt werden, dass Vergleiche zwischen Frameworks oftmals subjektiv beeinflusst sind. Was für *ein* Projekt vielleicht als Nachteil gilt, ist in einem gänzlich *anderen* Projekt unter Umständen sogar ein Vorteil.

Versucht man jedoch, in dem Bereich, in den sich JavaServer Faces eingliedert, ähnliche Frameworks zu finden, stößt man schnell auf inzwischen recht bekannte Namen wie Struts, WebWork oder Tapestry, wobei dies natürlich keine repräsentative Auswahl darstellt. Jedes dieser Frameworks hat seine speziellen Stärken und Vorteile. Während beispielsweise Struts als *Webapplication Framework* bezeichnet wird, ist JavaServer Faces vielmehr ein Framework für Benutzeroberflächen in Webapplikationen. Das muss nicht heißen, dass sich ein Entwickler unbedingt für ein Framework entscheiden muss, vielmehr kann auch eine Kombination von mehreren Frameworks durchaus Sinn ergeben. So stellt gerade die Beziehung zwischen JSF und Struts ein gutes Beispiel dar, wie sich zwei hervorragende Frameworks durchaus noch ergänzen können. Speziell zu diesem Thema existiert auch ein separates Kapitel in diesem Buch, das sich einer Integration von Struts und JavaServer Faces widmet.

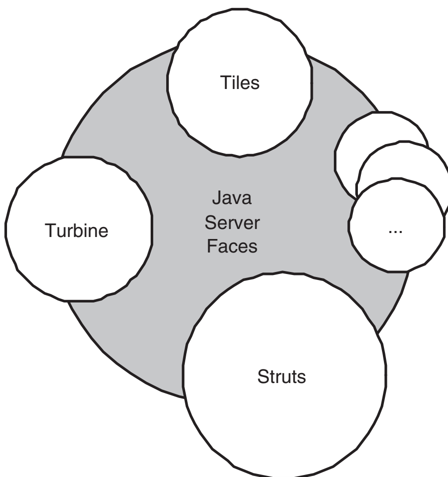


Abbildung 4.1: JSF und andere Frameworks

Die obige Darstellung wird sicherlich von Verfechtern anderer Frameworks komplett anders aussehen (was z. B. die Größe der Kreise anbelangt). Es ist jedoch so, dass viele Ideen und Ansätze anderer Frameworks in JSF übernommen und zum Großteil auch weiterentwickelt wurden. Natürlich hat jedes Framework zudem noch einige spezifische Eigenschaften, die JSF nicht direkt aufweisen kann. So existiert im Turbine-Framework zugleich ein Mechanismus für ein komfortables O/R-Mapping durch das in der Zwischenzeit entkoppelte Werkzeug *Torque*, auf das in Kapitel 8 näher eingegangen wird.

Auch Struts weist große Ähnlichkeiten mit JSF auf, wobei auch hier keine Konkurrenzsituation gegeben ist, sondern eher die Frage ist, wie am besten mit den erfolgreichen Ansätzen beider Frameworks eine Gesamtarchitektur realisiert werden kann. Die vielen Gemeinsamkeiten von JSF und Struts sind jedoch nicht ganz zufällig, ist doch Craig McClanahan als Kernentwickler und Visionär von Struts ebenfalls als Mitglied des so genannten *Specificaton Lead* des *Java Specification Requests 127* hinsichtlich der Entwicklung der JSF-Spezifikation somit auch maßgeblich an der Entwicklung von JSF beteiligt.

So ließe sich die Liste von Gemeinsamkeiten und Unterschieden von bestehenden Frameworks sicherlich endlos fortsetzen. Natürlich darf auch nicht vergessen werden, dass viele subjektive Faktoren oftmals auch ausschlaggebend für den Einsatz des einen oder anderen Frameworks sind.

4.4 Warum JSF?

JavaServer Faces ist – wenn es nach Sun geht – *der* neue Standard im Bereich Webanwendungen. Einen großen Vorteil von JavaServer Faces stellt sicherlich die Möglichkeit dar, dass Sie mittels JSF klar und strikt Anwendungslogik und Darstellung trennen können. Das Ziel einer Trennung von Anwendungslogik und Darstellung ist an sich nichts Neues, gibt es doch im Open-Source-Bereich bereits zahlreiche Projekte, die das gleiche Ziel haben (u. a. Struts). Mit JavaServer Faces gibt es jedoch einen ersten *offiziellen Standard* von Sun für die Erstellung von Webanwendungen. Zudem ist die Technologie, die Anwendungslogik von der Darstellung zu trennen, sehr konsequent und detailliert in JSF implementiert. Diese strikte Trennung im Programmiermodel erlaubt es Entwicklungsteams, mit unterschiedlichen Rollen an einem Webprojekt parallel zu arbeiten, wobei jedes Teammitglied sich um seinen Bereich kümmern kann:

So kann der Webdesigner mittels vordefinierten UI-(User Interface-)Komponenten eine Anwendung entwerfen, die wiederum vom UI-Komponenten-Entwickler mit Funktionalität versehen wird. Gerade diese Rollenteilung ist ein wichtiges Kriterium von JavaServer Faces.

Zusätzlich zum Programmiermodell bietet JavaServer Faces eine Vielzahl an vordefinierten Routinen und Funktionen, die eine Anwendungsentwicklung stark vereinfachen und beschleunigen. Als Beispiel seien Datenvalidierung, Event Handling oder Internationalisierung genannt. Auch dies ist sicherlich ein großer Pluspunkt, der für JavaServer Faces spricht.

Durch die Verwendung von Taglibs werden zudem die UI-Komponenten, die in der Webanwendung an verschiedenen Stellen verwendet werden, vereinheitlicht und deren Gebrauch vereinfacht. Ein Webdesigner muss sich somit nicht auf jeder Seite überlegen, mit welchen Attributen (graphisch) ein Button dargestellt werden soll. Er verwendet vielmehr die UI-Komponente unter Verwendung der entsprechenden Taglib und hat einen passenden Button zur Verfügung. Sollten UI-Komponenten angepasst bzw. auch neue Komponenten entwickelt werden, so kann dies an zentraler Stelle von einem Komponenten-Entwickler geschehen, der wiederum die Einbindung der Komponente dem Webdesigner mitteilt. Dieser kann daraufhin mittels einfachen Tags eine neue Komponente in seinem Seitenaufbau verwenden.

JSF hat somit sicherlich einige große Vorteile, wenn auch andere Frameworks diese Funktionalität in gleicher oder ähnlicher Weise ebenfalls anbieten. Daher kann mit Sicherheit keine generelle Aussage getroffen werden, wann JSF am besten eingesetzt werden sollte und wann nicht. Es kommt wie so häufig auf die projektspezifischen Gegebenheiten an. Ein Vorteil, der klar für JSF spricht, ist, dass JSF ein offizieller Standard ist. Sind jedoch in einem Projektteam alle Teammitglieder spezialisiert auf Struts und ist zudem das Projekt zeitlich gesehen in einem engen Rahmen gefasst, spielen natürlich andere Entscheidungskriterien eine Rolle. Daher soll dieses Buch Ihnen auch eine Entscheidungshilfe geben, damit Sie in Ihren speziellen Projekten entscheiden können, ob ein Einsatz von JavaServer Faces sinnvoll und nutzenbringend ist.

4.5 Was ist eine JSF-Anwendung?

Webanwendungen existieren bereits seit einigen Jahren. Eine große Zahl von Java-basierten Webanwendungen beruhen auf Servlet- oder JSP-Programmierung oder aus einer Kombination von beidem. Eine JSF-Anwendung ist prinzipiell erst einmal auch eine JSP-/Servlet-Anwendung. Auch eine JSF-Anwendung beinhaltet Komponenten wie:

- JavaBeans
- Serverseitige Helper-Klassen (z.B. für eine Kommunikation via RMI oder Corba, für Datenbankzugriffe etc.)
- JSP-Seiten für die Darstellung im Client

JavaServer Faces bietet jedoch zusätzlich noch weitere Funktionalitäten zum Bau einer Webanwendung. So sind u.a. folgende Funktionalitäten in JSF-Anwendungen bereits standardmäßig integriert:

- Eine Tag-Bibliothek für die Darstellung von UI-Komponetenten. Diese kann projektspezifisch den Erfordernissen angepasst oder bei Bedarf auch erweitert werden.
- Eine weitere Tag-Bibliothek für die Abbildung von Eventhandlern, Feldvalidierungen und weiteren Aktionen.
- Vordefinierte (aber auch erweiterbare und anpassbare) UI-Komponenten. Diese haben zudem die Möglichkeit, ihren kompletten Zustand server- oder clientseitig abzuspeichern.
- Ein Set an Handlern für Eventbearbeitung, Navigation und Feldprüfung

UI-Komponenten werden nicht mehr in jeder JSP-Seite separat erzeugt und dargestellt, sondern vielmehr mittels Tags beschrieben und mit Hilfe einer zentralen Tag-Bibliothek einheitlich für die Darstellung aufbereitet.

Auch die Verarbeitung von Events erfolgt nicht mehr separat in jeder einzelnen JSP-Seite, sondern wird an zentraler Stelle von Event-Handlern übernommen, die mittels einer Tag-Bibliothek eingebunden werden können.

Ein wichtiges Merkmal von JavaServer Faces stellt auch die Tatsache dar, dass sämtliche UI-Komponenten als *stateful object* auf dem Server oder Client gespeichert werden können. Dies bedeutet, dass mit der reinen Darstellung einer UI-Komponente die Aufgabe des Framework noch lange nicht erledigt ist, sondern vielmehr der komplette Zustand gespeichert wird.

4.6 Zielgruppe von JSF

JSF richtet sich sicherlich an die Entwicklergemeinde im Allgemeinen. Doch nicht nur der klassische Java-Entwickler soll JSF einsetzen können, auch Webdesigner und Webautoren sollen die Vorteile von JSF nutzen und einsetzen können. So soll eine JSF-Seite nicht ausschließlich nur durch einen Entwickler umgesetzt werden können, sondern nach einem Baukastenprinzip mit vorgefertigten und ausprogrammierten Komponenten soll auch ein in der Programmierung unbefleckter Webdesigner JSF-Seiten zusammensetzen können. Dies wird sicherlich erst dann richtig zum Tragen kommen, wenn entsprechende Toolhersteller die dazu notwendigen Programme bereitgestellt haben, aufgrund des großen Interesses seitens der möglichen Herstellerfirmen existieren bereits jetzt erste Programme dafür.

Zielgruppe ist jedoch auch der Anwendungsentwickler, der vorhandere Komponenten durch die notwendige Geschäftslogik ergänzt und somit komplexe Webanwendungen in kurzer Zeit realisieren kann. Der Basisentwickler soll durch die Entwicklung von

speziellen Komponenten ebenfalls durch den modularen Aufbau von JSF in seiner Arbeit profitieren. Somit adressiert JSF einen großen möglichen Kunden- und Nutzerkreis, der von der Technologie der JavaServer Faces profitieren kann.

4.7 Wer steckt hinter JSF?

JSF wird im Rahmen des *Java Community Process* von Sun entwickelt. Grundlage für die Entwicklung der JavaServer Faces war der *Java Specification Request (JSR)* Nummer 127. Über diese JSRs werden künftige Spezifikationen an Sun herangetragen und innerhalb eines Expertengremiums entsprechend ausgearbeitet. Ziel des JSR 127 ist:

This specification defines an architecture and APIs which simplify the creation and maintenance of Java Server application GUIs.

Sinngemäß definiert diese Spezifikation sowohl eine Architektur als auch eine API, die die Entwicklung und Wartung von Oberflächen von Webanwendungen vereinfacht.

Die Leitung des Gremiums unterliegt zum einen Ed Burns von Sun Microsystems, zum anderen Craig R. McClanahan. Gerade letzterer Name dürfte vielen bereits bekannt vorkommen. Craig R. McClanahan ist neben vielen anderen Engagements in der Apache Community maßgeblich an der Entwicklung von Struts beteiligt. Er hatte ursprünglich die Idee, Struts zu entwickeln, und ist auch heute noch Chef-Architekt und Visionär. Daher wundert es auch nicht, dass eine enge Verzahnung von JSF und Struts existiert und somit eine Koexistenz beider Frameworks möglich und auch gewollt ist.

In der Expertengruppe selbst reiht sich zudem vieles ein, was Rang und Namen hat. So sind Firmen wie BEA Systems, IBM, IONA Technologies, Novell, Borland, Hewlett-Packard, Oracle oder Siemens zu finden. Auch dies ist ein Indiz, mit welch großen Erwartungen JSF verbunden ist.

4.8 Spezifikation, Implementation und Open Source

Häufig werden bei einer Diskussion viele Begriffe durcheinander gewürfelt, wodurch am Ende meist keiner genau weiß, was alles im Detail zu bedeuten hat. Zunächst einmal ist der Begriff der *Spezifikation* deutlich von dem Begriff der *Implementation* zu unterscheiden.

Java-Spezifikationen werden im Rahmen des *Java Community Processes (JCP)* entwickelt und in einer *Specification Group* diskutiert und letzten Endes definiert. Auf den Webseiten des JCP unter *http://www.jcp.org* hat jeder einen öffentlichen Einblick in den Stand

des jeweiligen so genannten *JSR (Java Specification Requests)*. Für JSF existiert der JSR-127. Wichtig ist, dass eine Spezifikation noch keinerlei konkretes Produkt ist, sondern zunächst einmal eine systematische Sammlung von Anforderungen und Systembeschreibungen. Als Beweis sozusagen, dass die Spezifikation in sich schlüssig und anwendbar ist, wird zusammen mit der Spezifikation eine *Referenzimplementierung* publiziert, die auf Grundlage der Spezifikation eine konkrete Anwendung darstellt.

Daher kann es vorkommen (was ein durchaus gewollter Prozess ist), dass auf der Basis einer Spezifikation unterschiedliche Implementierungen existieren. Auch im Bereich JSF existiert neben der Referenzimplementierung von Sun bereits eine Open-Source-Variante *MyFaces*, die durch die Open-Source-Gemeinde entwickelt wurde.

Welche konkrete Implementierung verwendet wird, ist zunächst einmal eher eine politische Entscheidung als eine technische. Da durch eine Spezifikation geregelt ist, wie sich die Implementierung zu verhalten hat, ist bei einer korrekten Umsetzung eine Anwendung mit jeder Implementierung lauffähig.

Der Charme, den eine Open-Source-Implementierung wie z.B. *MyFaces* ausstrahlt, liegt mit Sicherheit auch darin begründet, dass sämtlicher Quellcode frei zugänglich ist. Auch war es in der Vergangenheit häufig der Fall, dass eventuelle Fehler in der Referenzimplementierung in der Open-Source-Implementierung nicht vorhanden waren oder aber schneller behoben wurden. Interessierte können sich über die Website von MyFaces unter

```
http://myfaces.sourceforge.net
```

eine aktuelle Version herunterladen und damit eigene Erfahrungen sammeln.

4.9 JSF und andere Technologien

JavaServer Faces baut auf einer Reihe von existierenden Technologien auf. So ist die Basis sicherlich der J2EE-Standard, in dessen Wirkungskreis Themen wie Servlets, JSP, JavaBeans oder Tag-Bibliotheken eine große Rolle spielen. JSF definiert jetzt nicht eine komplett neuartige Technologie, sondern ist eher als konsequente Weiterentwicklung bestehender und bereits etablierter Technologien zu verstehen. Es mussten zuerst notwendige Basistechnologien vorhanden sein, mit denen ein höherwertiges Framework entwickelt werden konnte. Dabei ist das Wort höherwertig nicht dahingehend fehlzuinterpretieren, dass JSF etwas Besseres als Servlets oder Tag-Bibliotheken sei. Vielmehr nutzt JSF die vorhandenen technischen Möglichkeiten, um auf dieser Basis ein umfangreiches und zum Teil auch komplexes Rahmenwerk bereitzustellen.

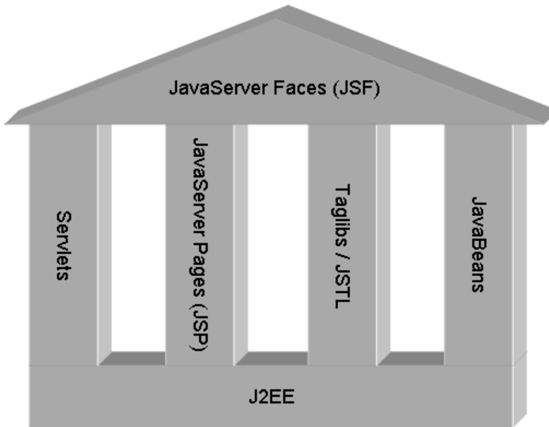


Abbildung 4.2: JSF und andere Technologien

Aufgrund der Tatsache, dass JSF nicht eine komplett neue Programmiersprache oder eine komplett neue Entwicklungsumgebung ist, sondern ein Framework basierend auf etablierten und umfangreichen Basistechnologien darstellt, ist im Kapitel 2 eine Einführung zu den Themen Servlets, JSPs und Tag-Bibliotheken (Taglibs) gegeben.

4.10 Toolunterstützung

Eine Technologie kann noch so gut konzeptioniert und durchdacht sein, ohne eine entsprechende Unterstützung seitens der Toolhersteller wird sie es schwer haben, sich im Markt zu etablieren. Ein Erfolg einer neuen Technologie beruht zum großen Teil auch darauf, wie diese von vorhandenen Editoren, Entwicklungsumgebungen und sonstigen Werkzeugen unterstützt wird. JavaServer Faces existiert zurzeit in einer ersten Version. Viele namhafte Firmen haben bereits eine Unterstützung von JSF zugesagt, können aber nicht von heute auf morgen fertige Toolunterstützung bieten. So sind die ersten JSF-Anwendungen, die entwickelt werden, sicherlich noch viel Handarbeit. Gerade im Bereich der Navigationsdefinition oder bei Umsetzung der einzelnen graphischen Webseiten wird es künftig komfortable Werkzeuge geben, um diesen Schritt wesentlich zu vereinfachen. Somit ist es für »Pioniere« in JSF-Anwendungen zum Teil noch etwas aufwändig und auch mühsam, in naher Zukunft jedoch wird es mit Sicherheit reichlich Unterstützung von fast allen namhaften Herstellern in diesem Bereich geben. Bereits kurz nach Erscheinen der Final-Version der Referenzimplementierung und der Spezifikation gab es jedoch von vielen bekannten Toolherstellern erste Beta-Versionen ihrer Software, die schon eine JSF-Integration boten. Somit ist abzusehen, dass eine allgemeine Toolunterstützung sehr schnell möglich sein wird.

4.11 JSF und EJBs

JSF ist in erster Linie ein Framework für das User Interface. *Enterprise JavaBeans* (EJB) dagegen sind Bausteine, die in einem Application-Server (umfangreiche) Funktionalitäten bereitstellen und in diesem Container ebenfalls ausgeführt werden. Beide Technologien stellen somit überhaupt keinen Gegensatz, sondern vielmehr eine gewollte und sinnvolle gegenseitige Ergänzung dar. Während JSF für die Navigation und die Darstellung in der Anwendung selbst verantwortlich ist, kann innerhalb dieses Ablaufs an einer geeigneten Stelle ein EJB-Aufruf z. B. für einen Datenbankzugriff oder für die Berechnung eines komplexen Algorithmus erfolgen. Bei einer entsprechend großen und umfangreichen Webanwendung sollten daher immer beide Technologien zu finden sein.

Gerade zum Thema JSF und EJB ist in Kapitel 10 ein Beispiel zu finden, wie aus einer JSF-Anwendung ein EJB-Aufruf realisiert werden kann.

4.12 JSF und JavaScript

Es ist vielleicht etwas vermessen, das Thema *JSF und JavaScript* direkt unterhalb des Abschnitts *JSF und EJBs* zu setzen. Soll doch nicht der Eindruck entstehen, EJBs und JavaScript sind eine ähnlich mächtige Technologie ;-) Dennoch soll an dieser Stelle kurz auf die Bedeutung von JavaScript innerhalb des JSF-Frameworks eingegangen werden.

Um JSF einsetzen zu können, sind erst einmal keinerlei Kenntnisse in JavaScript notwendig. JSF bietet vielmehr bereits eine mächtige Taglib inklusive Standardrenderern, die die Darstellung von UI-Komponenten beinhalten. Mit einem Renderer ist in diesem Zusammenhang eine Java-Klasse gemeint, die eine Ausgabe in der gewünschten Form (Html, Wml, …) erzeugt. Der Entwickler muss sich somit nicht mit Html- bzw. JavaScript-Kommandos befassen, nur um z. B. ein *MouseDown-Event* per JavaScript korrekt abfangen zu können.

Es ist jedoch wichtig zu wissen, dass JavaScript als Ergebnis eines Rendering-Vorgangs einer Komponente zum Einsatz kommen kann. Dies bedeutet demnach, dass die Browser, auf denen die Anwendung zum Einsatz kommt, ebenfalls JavaScript unterstützen müssen. Ist es ein Designziel einer Webapplikation, eine Anwendung vollkommen ohne JavaScript-Befehle zu realisieren, sind eigene Komponenten bzw. Renderer zu entwickeln, wie dies in Kapitel 9. JSF erweitern und anpassen gezeigt wird.

```
<a href="#" onmousedown="document.forms[1].actionfield.value='cmdList';
document.forms[1].submit()">Beispiellink</a><input type="hidden" name="cmdList"/>
```

Listing 4.1: Ausgabe eines mittels JSF erzeugten Hyperlinks

In Listing 4.1 ist ein Beispiel zu sehen, wie ein Hyperlink mit Hilfe der Standard-Tagbibliothek von JSF dargestellt wird. Es wird beim Klicken ein Wert in ein *Hidden-Field* gesetzt, das natürlich Bestandteil einer umgebenden *Form* sein muss. Diese Form wird nach Setzen des Wertes abgeschickt.

Der gesamte Ablauf ist ohne JavaScript so natürlich nicht möglich. Dazu müsste ein komplett neues Konzept samt Renderer erarbeitet und codiert werden. Es ist daher sicherlich empfehlenswert, auf die Funktionen von JSF, die JavaScript nutzen, zurückzugreifen, zumal heutzutage fast alle Browser die aktuelle Html- und JavaScript-Spezifikation unterstützen.

4.13 Rollenkonzept

Ein wichtiges Designziel von JavaServer Faces ist die strikte Trennung der einzelnen Arbeitsschritte und Arbeitsinhalte in einem Webprojekt. Ein häufiges Problem bei der Erstellung von Webanwendungen in der Vergangenheit war oftmals, dass Webdesigner JSP-Tags zwar nicht zwangsläufig programmieren, aber mindestens verstehen mussten, wollten sie eine funktionsfähige Seite abändern oder erweitern. Zudem mussten Programmierer wiederum allzu häufig im Seitenquelltext, also in den Html-Dateien direkt Änderungen vornehmen, was eigentlich Aufgabe der Webdesigner ist.

Mittels JavaServer Faces ist eine genaue Trennung nach Aufgabenbereich vorgesehen. So werden folgende Rollen beschrieben:

- Seitenautoren / Webdesigner
- Applikationsentwickler
- Komponentenentwickler
- Tool-Hersteller

Seitenautoren sind dabei für den graphischen Entwurf der Seiten verantwortlich und bauen – basierend auf den Tag-Bibliotheken – die einzelnen Webseiten auf. Sie verwenden dabei die Markupsprache Html. Zusätzlich verwenden sie Tags aus der JSF-Tagbibliothek sowie eventuell vorhandenen benutzerdefinierten Tagbibliotheken, die durch Komponentenentwickler bereitgestellt wurden.

Applikationsentwickler zeichnen sich für die Programmierung der Anwendungslogik aus. Ihnen obliegt es, die Daten, die beispielweise von einem Formular an einen Server geschickt wurden, korrekt zu verarbeiten, gegebenenfalls Datenbankzugriffe durchzuführen oder Ergebnisse zu berechnen.

Komponentenentwickler wiederum liefern sowohl den Seitenautoren als auch den Applikationsentwicklern Bausteine für UI-Komponenten, die von den Seitenautoren eingebaut und deren Events von den Applikationsentwicklern verarbeitet werden.

Mit Tool-Herstellern sind beispielsweise Softwarefirmen bezeichnet, die Werkzeuge für den generellen Einsatz von JavaServer Faces bereitstellen. Dies können Erweiterungen in Entwickungsumgebungen, Editoren oder sonstige Programme sein, die eine Arbeit mit JSF erleichtern oder verbessern.

In den meisten Projekten werden demnach hauptsächlich Seitenautoren und Applikationsentwickler zu finden sein. Unter Umständen kann auch der Bereich der Komponentenentwicklung involviert sein.

4.14 Lebenszyklus einer JavaServer Faces-Seite

Eine JavaServer Faces-Seite ist zunächst einmal eine JSP-Seite, die in den Kontext einer JSF-Anwendung eingebunden ist. Daher durchläuft eine JSF-Seite die gleichen sieben Phasen wie eine JSP-Seite:

Phase	**Beschreibung**
Seite übersetzen	Aus der JSP-Seite wird ein Servlet erzeugt.
Seite kompilieren	Das automatisch erzeugte Servlet wird kompiliert.
Seite laden	Die als Servlet kompilierte Seite wird durch den Classloader geladen.
Instanz erzeugen	Es wird für die JSP-Seite eine Instanz angelegt.
Seite initialisieren	Die in der Servlet-Spezifikation hinterlegte init-Routine des Servlets/ der JSP-Seite wird aufgerufen.
Service-Routine	Pro Aufruf der Seite wird ein Task gestartet, der die Service-Methode des Servlets ausführt.
Servlet entfernen	Wird das Servlet/die JSP-Seite nicht mehr benötigt bzw. wird z. B. der Servletcontainer heruntergefahren, wird das Servlet korrekt entladen.

Tabelle 4.1: Die sieben Phasen einer JSP-Seite

JavaServer Faces bieten jedoch zusätzlich zum »klassischen« Ablauf eine Vielzahl von Funktionen, die den Ablauf einer JSF-Seite beeinflussen und steuern. Vielmehr durchläuft der Request auf Seiten des Servers einen umfangreichen Lebenszyklus. Die Phasen, die innerhalb eines Request durchgeführt werden, unterscheiden sich jedoch, ob es sich um einen so genannten *Faces-Request* oder einen *Non-Faces-Request* handelt.

Request- und Response-Arten

In JavaServer Faces werden zwei Arten von Requests sowie zwei Arten von Responses unterschieden. Requests, also Anfragen an den Server, werden unterschieden nach:

- Faces Requests
- Non-Faces Requests

Bei Faces Requests entstammt die Anfrage selbst aus einer JSF-Seite, d.h. es wurde z.B. ein Formular einer JSF-Seite abgeschickt oder ein Hyperlink auf einer JSF-Seite angeklickt. Wichtig ist, dass der Request dabei *immer* auf das Faces-Servlet geleitet wird, da es ansonsten kein Request an eine JavaServer Faces-Anwendung darstellt. Ein Non-Faces Request ist eine Anfrage an den Server nach einer anderen als einer JSF-Ressource, beispielsweise eine normale Anfrage nach einer JSP-Seite oder einem sonstigen Servlet. Genauso ist das einfache Abrufen einer statischen Html-Seite ein Non-Faces Request.

Da nach einer Anfrage an einen Server auch immer eine Antwort (Response) erzeugt wird, werden auch hierbei zwei Arten unterschieden:

- Faces Response
- Non-Faces Response

Bei einer Faces Response wird gemäß dem Lebenszyklus einer JSF-Seite eine neue oder bestehende JSF-Seite gerendert und an den Browser zurückgeliefert. Bei einer Non-Faces Response wird dagegen keine JSF-Funktionalität benötigt, wie z.B. beim Abarbeiten einer normalen JSP-Seite oder eines Servlets. Ebenso stellt das Abrufen einer statischen Html-Seite eine Non-Faces-Response dar.

Basierend auf den unterschiedlichen Request- und Response-Arten sind natürlich verschiedene Szenarien und Kombinationen vorstellbar. So kann ein Non-Faces Request eine Faces Response auslösen, indem z.B. auf einer Html-Seite ein Link angeklickt wird, der auf die Adresse einer JSF-Anwendung verweist. Dies ist meist dann der Fall, wenn eine JSF-Anwendung aus einem statischen Bereich einer Webseite gestartet wird. Umgekehrt ist es natürlich genauso möglich, dass aus einer JSF-Seite heraus auf eine statische Html-Seite oder eine vollkommen anderweitig gelagerte Anwendung basierend auf reiner JSP-Technologie verwiesen wird. Das wohl häufigste und übliche Szenario ist jedoch, dass ein Faces Request eine Faces Response auslöst, indem innerhalb einer JSF-Anwendung durch Hyperlinks und Commandbuttons navigiert wird. Dieser so genannte *Standard Lebenszyklus* soll daher im Folgenden näher beleuchtet werden.

Der Standard Lebenszyklus eines Requests

In der folgenden Erklärung wird davon ausgegangen, dass der Request von einer JSF-Anwendung erzeugt wurde und das Ergebnis wiederum eine JSF-Seite darstellt. Andere Szenarios sind durchaus möglich und sind als Sonderfälle des Standard Lebenszyklus einzuordnen.

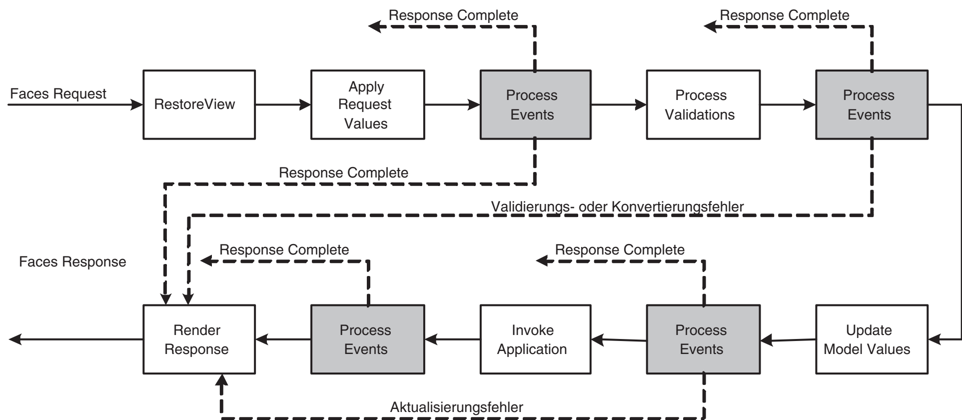


Abbildung 4.3: Standard Lebenszyklus

Der Weg, den ein Request auf seinem Weg durch das Framework durchschreitet, ist keinesfalls fix vorgegeben. Vielmehr kann aufgrund eines Ereignisses (z.B. beim Scheitern der Datenvalidierung) ein anderer Ablauf eingeschlagen werden. Daher ist der Standard-Weg mit einer durchgezogenen Linie eingezeichnet. Mögliche Alternativabläufe sind dagegen mit einer gestrichelten Linie dargestellt. Bei einem Fehler (z.B. in der Datenvalidierung) wird erneut dieselbe Seite angezeigt. Ist auf dieser Seite ein entsprechendes *Error-Tag* vorgesehen, wird an dieser Stelle eine Fehlermeldung ausgegeben.

Für das Verständnis der Verarbeitung innerhalb des Zyklus ist es wichtig zu wissen, dass innerhalb einer JSF-Anwendung jede Seite (*View*) als Baumstruktur dargestellt wird. Diese Baumstruktur wird als *View-Objekt* im so genannten *FacesContext* abgelegt. Erfolgt basierend auf einer Seite, deren View-Objekt bereits im FacesContext abgelegt ist, wiederum ein Request, so wird auf das gespeicherte View-Objekt zurückgegriffen. JSF-Anwendungen haben somit ein Dialoggedächtnis. Mehr zum Komponentenbaum ist in Kapitel 6.3 zu finden, Details zum Dialoggedächtnis folgen in Kapitel 6.11.

Phase Restore View

Die Phase *Restore View* ist die erste Phase innerhalb des Lebenszyklus eines Requests. Im Falle eines Non-Faces Requests wird in dieser Phase zunächst ein leeres View-Objekt erzeugt und im FacesContext abgelegt. Dieses View-Objekt wird im weiteren Verlauf mit den dazugehörigen Komponenten befüllt.

Im Falle, dass ein View-Objekt bereits vorhanden ist (Faces Request), wird dieses aus dem FacesContext geladen und mit den dazugehörigen Validatoren, Listenern und Konvertern verbunden. Dabei wird das Konzept der *Zustandsspeicherung* (*State Saving*) verwendet, bei dem der Zustand einer View entweder serverseitig oder clientseitig gespeichert wird.

Phase Apply Request Values

Im Anschluss an den Aufbau des Komponentenbaums holt sich jede einzelne Komponente den neuen Wert aus dem Request. Dieser neue Wert wird zunächst lokal in der Komponente selbst abgespeichert. Sollte es bereits in dieser Phase zu Fehlern kommen, wird eine entsprechende Fehlermeldung generiert und ebenfalls im FacesContext gespeichert.

Wichtig ist es zu wissen, dass die Werte des Requests zunächst nur lokal, also in der Komponente selbst, gespeichert werden. Es erfolgt an dieser Stelle noch kein Aktualisieren des zugrunde liegenden Modells.

In dieser Stufe finden ebenfalls erste Datenkonvertierungen statt. So wird ein Wert aus dem Request heraus in den entsprechenden Datentyp umgewandelt, so wie dieser im Modellobjekt hinterlegt ist.

Sind an einzelnen Komponenten entsprechende Eventlistener registriert, werden diese in der aktuellen Phase bereits benachrichtigt. Am Ende dieser Phase haben sämtliche Komponenten (aber nur die Komponenten, nicht das Modell!) die neuen Werte des Requests gesetzt.

Phase Process Validation

In der Phase der Validierung werden sämtliche Überprüfungen durchgeführt, die in Form so genannter Validatoren an eine Komponente gekoppelt sind. So ist es in JavaServer Faces möglich, z. B. einen *Required*-Validator oder einen *Range*-Validator zu hinterlegen, mit deren Hilfe die Eingaben eines Benutzers zusätzlich überprüft werden können.

Sollte es während dieser Phase zu Fehlern kommen, wird eine entsprechende Fehlermeldung generiert und im FacesContext als so genannte *FacesMessage* gespeichert. Unter Umständen existieren im Kontext bereits Meldungen aus der Phase zuvor; diese

werden nicht überschrieben, sondern zusammen mit den neu hinzugekommenen Meldungen dieser Phase dargestellt. Dazu wird der Verarbeitungsablauf unterbrochen und auf die gleiche Seite zurückverwiesen. Zusätzlich erscheinen die aufgetretenen Fehlermeldungen, falls auf dieser Seite ein Bereich durch die Angabe des Tags `<h:messages />` dafür vorgesehen ist.

Phase Update Model Values

Erst in dieser Phase wird das Modellobjekt mit den neuen Werten des Requests aktualisiert. Sollten bis zu dieser Phase noch keinerlei Fehler aufgetreten sein, werden die Werte, die zunächst lokal in den Komponenten gespeichert werden, in das Modellobjekt übertragen. Dazu wird der Komponentenbaum durchlaufen und sämtliche Werte in die passenden Eigenschaften des Modellobjektes abgelegt.

Auch in dieser Phase können eventuell registrierte Eventlistener benachrichtigt werden, z. B. so genannte *ValueChangeListener*, die bei einer Änderung eines Eingabewertes benachrichtigt werden.

Phase Invoke Application

In dieser Phase werden sämtliche Events auf Anwendungsebene abgearbeitet. Dies ist meist die eigentliche Weiterleitung zu einer Folgeseite aufgrund eines Rückgabewertes eines UICommands. Wird beispielsweise zu einem Hyperlink das Attribut `action` mit dem Wert `success` hinterlegt, wird in dieser Phase die entsprechende Seite zu diesem Rückgabewert ermittelt und dorthin weitergeleitet. Dazu wird wiederum der Komponentenbaum für die neue Seite aufgebaut und die Steuerung an die nächste Phase übergeben, die für das Rendern der neuen Seite verantwortlich ist.

Phase Render Response

Der Komponentenbaum, der von der vorherigen Phase im Kontext abgelegt wurde, wird mittels der entsprechenden `encode`-Methode jeder Komponente gerendert. Somit wird durch jede einzelne Komponente ein Teil der Gesamtseite erzeugt. Sollten Fehler in vorherigen Phasen aufgetreten sein, wird die ursprüngliche Seite nochmals dargestellt und zusätzlich der Bereich `<h:messages />` mit den entsprechenden Fehlermeldungen versehen. Je nachdem, ob das Rendering in der Komponente selbst stattfindet oder dafür ein separater Renderer benöigt wird, werden die notwendigen Funktionen aufgerufen.

4.15 Fazit

Ziel dieses Kapitels war es, grundlegende Fragen zu JavaServer Faces zu beantworten. So wurde zunächst geklärt, dass JavaServer Faces ein Framework, also ein Grundgerüst, darstellt, mit dem Webanwendungen allgemeiner Art sowie Portale, Content-Management-Systeme oder Online-Communities aufgebaut werden können.

Dass JavaServer Faces zwar eine offene Spezifikation besitzt, die Referenzimplementierung jedoch nicht Open Source ist, war ebenfalls Teil dieses Kapitels. In diesem Zusammenhang wurde auch auf die Open-Source-Implementierung MyFaces hingewiesen, die als Open-Source-Initiative die JSF-Spezifikation genauso erfüllt wie die Referenzimplementierung von Sun.

Der Lebenszyklus eines Faces-Requests wurde ebenfalls ausführlich dargestellt und erläutert. Abhängig davon, ob ein Faces-Request oder ein Nicht-Faces-Request erzeugt wurde, verhält sich der Ablauf unterschiedlich. Ebenso wird zwischen einer Faces-Response und einer Nicht-Faces-Response unterschieden.

5 Die erste eigene JSF-Applikation

Kapitelziel

Auch in dieser Einführung darf das kategorische *Hallo-Welt*-Beispiel natürlich nicht fehlen. Eine so genannte *Hallo-Welt*-Anwendung wird oftmals in Einführungen zu neuen Technologien verwendet, um zu zeigen, wie eine neue Technologie prinzipiell funktioniert. Dabei passiert in einer klassischen *Hallo-Welt*-Anwendung nichts anderes, als dass mit der zur Verfügung stehenden Technologie angestrebt wird, den Wortlaut »Hallo Welt« auszugeben. Daher entstammt auch der Name *Hallo-Welt*-Anwendung. Da eine einfache Ausgabe des Strings *Hallo Welt* vielleicht eine etwas zu kleine Anwendung ist, wird im folgenden Beispiel eine Anwendung demonstriert, die zumindest auch Benutzereingaben entgegennimmt sowie einen kleinen Geschäftsprozess abbildet.

Anhand der ersten Anwendung werden die Funktionsweise und die Arbeit mit dem Framework JavaServer Faces schrittweise erläutert und die grundlegenden Zusammenhänge aufgezeigt. So wird bei der Einrichtung der Entwicklungsumgebung begonnen. Bevor nämlich die Arbeit mit JavaServer Faces aufgenommen werden kann, ist die notwendige Infrastruktur bereitzustellen. Gerade bei diesem Thema sind einige Fallstricke zu beachten.

In einem weiteren Abschnitt wird auf die in JavaServer Faces verwendeten Konfigurationsdateien eingegangen und deren Einsatz erklärt. Des Weiteren werden wichtige Begriffe wie *Managed Bean*, *Backing Bean* oder *Modellobjekt* definiert und erläutert.

Natürlich wird auch auf die konkrete Programmierung einer Faces-Seite eingegangen sowie auf das Navigationskonzept, das in JavaServer Faces in einer separaten Konfigurationsdatei vorgehalten wird. Sind alle Arbeitsschritte erfolgreich durchlaufen, kann am Ende dieses Kapitels die erste Anwendung basierend auf JavaServer Faces gestartet werden.

5.1 Vorgehensweise

Im Folgenden wird eine kleine Webanwendung erzeugt, anhand derer ein erster Einblick in die Konzepte und Ideen hinter JSF gegeben werden soll. Die Beispielanwendung besteht aus zwei Seiten, die über ein Formular-Post miteinander in Verbindung

stehen. Dabei werden auf der ersten Seite der Vorname und der Nachname eines Besuchers sowie dessen Geburtsdatum in das Formular eingetragen, auf der zweiten Seite soll daraufhin nach Abschicken des Formulars das aktuelle Alter berechnet und ausgegeben werden. Die Anwendung berechnet somit auf Basis des Geburtsdatums das Alter in Jahren und gibt dieses letztendlich zusammen mit dem persönlichen Glückwunsch sowie dem Vor- und Nachnamen der ersten Seite auf der Ausgabeseite aus. Die Berechnung des Alters ist dabei schon ein erster Geschäftsprozess, der in dieser Anwendung durchgeführt wird (ok, Geschäftsprozess ist vielleicht ein wenig hoch gegriffen, aber prinzipell stimmt die Aussage schon).

Eine typische JSF-Entwicklung verläuft in mehreren Arbeitsgängen, in denen die einzelnen Bausteine entwickelt werden müssen. Wie in Kapitel 4.13 bereits aufgezeigt wurde, liegt ein großer Vorteil bei der Erstellung von JSF-Anwendungen darin, dass mit verteilten Rollen parallel an einer Anwendung gearbeitet werden kann. Dadurch, dass nicht ein oder mehrere Entwickler sequentiell bestimmte Arbeiten durchführen müssen, ist eine schnelle Anwendungsentwicklung auch mit verteiltem Know-how möglich.

Für die beschriebene Beispielanwendung werden die folgenden Schritte jedoch zur Demonstration der Reihe nach abgearbeitet:

- Einrichten der Entwicklungsumgebung
- Aufsetzen eines neuen Projekts sowie Anlegen der Konfigurationsdateien
- Aufbau eines Modellobjektes, das für die Datenhaltung zuständig ist
- Das Modellobjekt per Konfigurationsdatei dem Framework bekannt machen, damit eine automatische Verwaltung möglich ist
- Unter Verwendung der Tagbibliotheken die einzelnen JSF-Seiten erzeugen
- Die Navigationsstruktur festlegen
- Testen und Starten der Anwendung

Diese Arbeitsschritte werden meist von Personen mit unterschiedlichem Aufgabenschwerpunkt in einem Projektteam realisiert. Dies bedeutet, dass einerseits eine parallele Entwicklung möglich ist, jedoch eine Kommunikation untereinander unbedingt notwendig ist. So kann beispielsweise ein Anwendungsentwickler parallel zu einem Webdesigner arbeiten, sie sollten sich beide jedoch unbedingt über die Schnittstellen im Vorfeld geeinigt haben.

Um eine erste lauffähige JSF-Anwendung aufzubauen, könnte in einer Beispiel-Anwendung sicherlich der ein oder andere Arbeitsgang noch ausgelassen und gewisse Themen verkürzt dargestellt werden. Um jedoch die elementaren Komponenten und deren Verwendung aufzeigen zu können, werden für diese erste JSF-Anwendung bewusst viele wesentliche Basiskomponenten verwendet und angesprochen.

5.2 Einrichten der Entwicklungsumgebung

Das Einrichten der Entwicklungsumgebung ist oftmals eine etwas aufreibende Tätigkeit. Es müssen alle benötigten Komponenten aus dem Netz heruntergeladen und installiert werden. Oftmals merkt man aber erst während der Installation, dass gewisse Versionen einer Software nicht zu anderen Programmversionen passen. Aus diesem Grund sind die in diesem Buch verwendeten Programme und Bibliotheken komplett auf beigelegter CD enthalten. Diese sind auch im Zusammenspiel erprobt und können daher problemlos für die ersten Schritte mit JSF eingesetzt werden.

In Kapitel 3 wurden bereits die Demoanwendungen installiert, die der aktuellen Referenzimplementierung beigelegt sind. Dabei wurde der Apache Tomcat-Server verwendet, der im Web Services Developer Pack integriert ist. Für das folgende Beispiel wird im Gegensatz zu der in Kapitel 3 gezeigten Vorgehensweise ein separater Tomcat-Server installiert und mit der Entwicklungsumgebung verbunden.

Alle Beispiele werden mit der Entwicklungsumgebung *Eclipse* realisiert. Eclipse ist eine freie Entwicklungsumgebung, die unter *http://www.eclipse.org* heruntergeladen werden kann. In der Version 2.1.1 beträgt die Download-Größe etwa 18 MB. Auf beigefügter CD sind Versionen für Windows, Linux und Mac vorhanden, für weitere Zielplattformen gibt es auf der Webseite entsprechende Pakete.

Des Weiteren wird natürlich ein aktuelles *Java Development Kit* (JDK) von Sun benötigt. Alle im Buch vorhandenen Beispiele beruhen dabei auf der Version 1.4.2. Eine ältere Jdk-Version kann nicht verwendet werden, aktuellere Versionen können natürlich jederzeit eingesetzt werden.

Die Version 1.0 der JavaServer Faces-Spezifikation sowie der Referenzimplementierung kann über die Webseite von Sun unter *http://java.sun.com/j2ee/javaserverfaces* heruntergeladen werden.

Zusätzlich wird das *Java Web Services Developer Pack* benötigt, das zum Zeitpunkt der Drucklegung des Buches in der Version 1.3 vorliegt. Künftig wird JavaServer Faces ebenfalls in diesem Paket mit ausgeliefert. In der Version 1.3 ist zwar bereits eine JavaServer Faces Referenzimplementierung enthalten, allerdings handelt es sich dabei um eine Testversion, die nicht mehr verwendet werden sollte.

Das Java Web Services Developer Pack ist über die Webseite von Sun unter

http://java.sun.com/webservices/downloads/webservicespack.html

zu beziehen bzw. ist ebenfalls der CD beigefügt. Da wie bereits geschildert nicht der Tomcat Server des Web Services Developer Packs verwendet werden soll, sondern ein separater aktueller Tomcat-Server, kann dieser unter *http://jakarta.apache.org/* heruntergeladen und installiert werden. Auf der CD ist die Version 5.0.19 enthalten.

Um den Tomcat-Server aus der Entwicklungsumgebung heraus sowohl im Debug- wie auch im Run-Modus starten zu können, wird ein PlugIn der Firma *Sysdeo* verwendet, das aus Eclipse heraus eine Verbindung zu Tomcat herstellt. Somit kann komfortabel über die Entwicklungsumgebung der Tomcat-Server gestartet und beendet werden. Das PlugIn ist ebenfalls kostenfrei über die Webseite

http://www.sysdeo.com/eclipse/tomcatPlugin.html

zu beziehen. Eine kurze Installationsanleitung ist ebenfalls dabei. Im Regelfall genügt es, alle Dateien in das Eclipse-Plug-In-Verzeichnis zu extrahieren und über die Einstellungen in Eclipse das PlugIn zu aktivieren. Die Einstellungen befinden sich unter WINDOW / PREFERENCES / TOMCAT. An dieser Stelle ist unter *Tomcat-Home* das Verzeichnis des Tomcat-Servers anzugeben, in das dieser installiert wurde. Ebenfalls ist die Tomcat-Version *5.x* zu markieren.

Als letzten Schritt ist nochmals zu kontrollieren, ob Eclipse die korrekte Jdk-Version anzieht. Sollten Sie mehrere Jdk-Versionen auf Ihrem System installiert haben, kann es sein, dass unter Umständen die falsche Version angezogen wird. Unter WINDOW / PREFERENCES JAVA / INSTALLED JREs sollte das *Jdk 1.4.2* zu sehen und ebenfalls aktiviert sein. Ist dies nicht der Fall, muss es über ADD hinzugefügt werden.

5.3 Aufsetzen eines neuen Projektes

Zur Erstellung einer JavaServer Faces-Anwendung muss in der Entwicklungsumgebung ein J2EE-Projekt (bzw. ein Tomcat-Projekt unter Eclipse) angelegt werden. Damit wird eine grundsätzliche Verzeichnisstruktur bereits vordefiniert und eine Verknüpfung zum Tomcat-Server geschaffen. Wichtig hierfür ist, dass das Tomcat-PlugIn erfolgreich im Vorfeld installiert wurde, da ansonsten kein korrektes Tomcat-Projekt in Eclipse angelegt werden kann.

Über FILE / NEW / PROJECT wird ein neues *Tomcat-Projekt* angelegt. Als Projektname wird *JSFTraining* vergeben und entweder in das Default-Verzeichnis oder ein anderes frei wählbares Verzeichnis abgespeichert. Das Projekt *JSFTraining*, das in diesem Arbeitsschritt angelegt wird, kann später auch für die Beispiele der einzelnen Kapitel verwendet werden, daher ist der Name auch etwas allgemeiner gehalten. Sämtliche im Buch gezeigten Beispiele verwenden den Projektnamen *JSFTraining* (bis auf die Beispielanwendung JSF-Weblog), daher ist es empfehlenswert, für die eigenen Beispiele diesen ebenfalls zu verwenden. Das Zielverzeichnis, in dem die Anwendung aufgebaut werden soll, kann natürlich frei gewählt werden. Es wird durch den Assistenten ein Standardverzeichnis vorgeschlagen, das jedoch mit einer eigenen Angabe überschrieben werden kann.

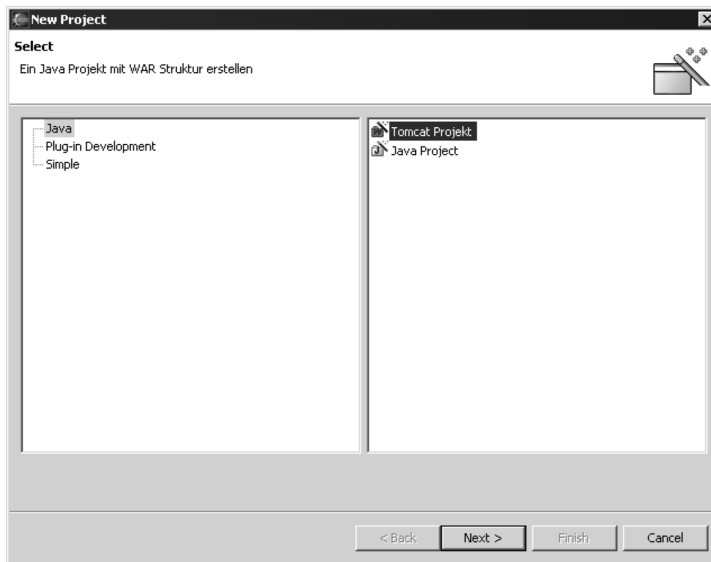


Abbildung 5.1: Anlegen eines Tomcat-Projektes

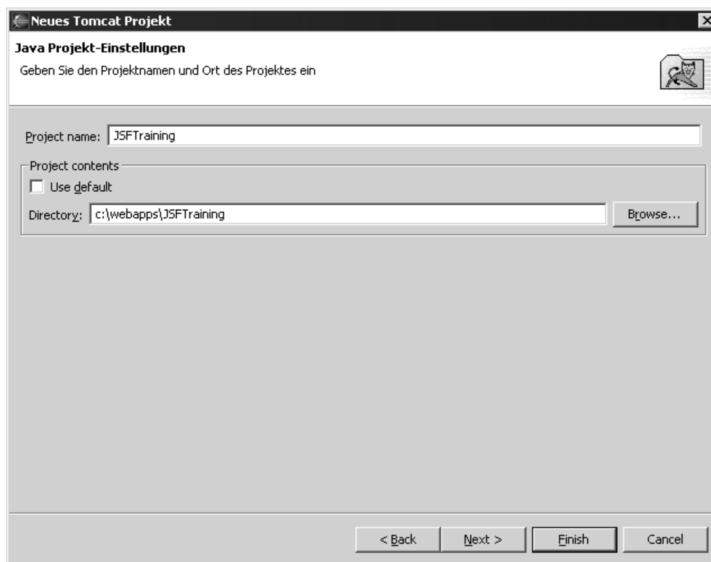


Abbildung 5.2: Festlegen des Projektnamens

Optional kann in einem dritten Schritt die *Anwendungs-URI* verändert werden. Die Anwendungs-URI definiert den Namen, über den die Webanwendung im Browser aufgerufen werden kann. Für das Beispiel werden die Vorgaben verwendet und durch Bestätigen mit FINISH das Anlegen eines neuen Projektes im Workspace angestoßen.

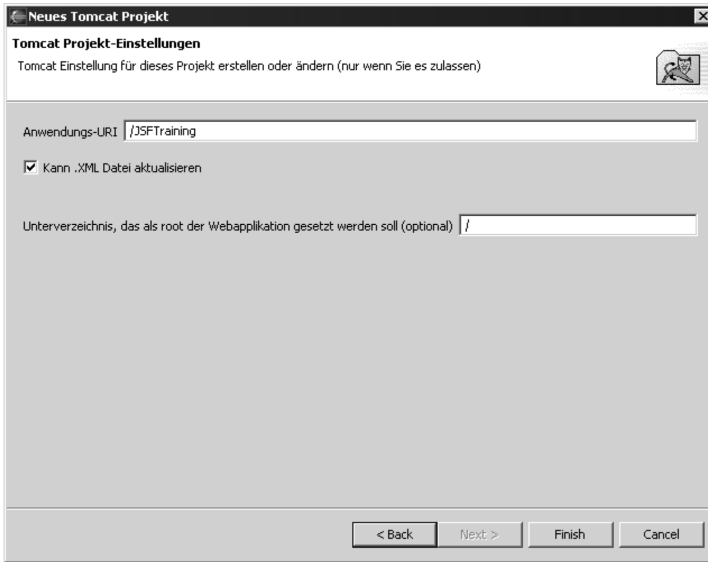


Abbildung 5.3: Angabe einer Anwendungs-URI

Nach dem Anlegen eines neuen Projektes durch den Projektassistenten ist bereits eine Grundstruktur vorgegeben, in die die Anwendung integriert werden kann. Im nächsten Abschnitt werden zunächst die notwendigen Konfigurationsdateien angelegt, bevor es an die eigentliche Arbeit mit JavaServer Faces geht.

5.4 Konfigurationsdateien

Eine JSF-Anwendung benötigt mindestens zwei Konfigurationsdateien. Zum einen den *Deployment Deskriptor*, den alle J2EE-konformen Webanwendungen aufweisen müssen, sowie die JSF-spezifische Anwendungskonfigurationsdatei *faces-config.xml*. Der Speicherort des Deployment-Deskriptors *web.xml* ist in der J2EE-Spezifikation fest vorgeschrieben. Der Deskriptor *web.xml* muss dabei im Unterverzeichnis *WEB-INF* der Webanwendung zu finden sein. Die Anwendungskonfigurationsdatei *faces-config.xml* kann an einer beliebigen Stelle abspeichert werden, es empfiehlt sich jedoch, diese ebenfalls im *WEB-INF*-Verzeichnis der Webanwendung abzulegen.

Der Deployment-Deskriptor

Der Deployment-Deskriptor *web.xml* beinhaltet grundlegende Einstellungen der Webanwendung. Es werden darin verwendete Servlets definiert, eventuelle Mappings hinterlegt sowie Initialisierungsparameter festgeschrieben. Für die Verwendung von JavaServer Faces müssen folgende Eintragungen vorgenommen werden:

- Es muss ein Servlet festgelegt werden, das für die Verarbeitung von JavaServer Faces Requests zuständig ist.
- Es muss ein dazu entsprechendes Servlet-Mapping konfiguriert werden.

Folgende XML-Datei zeigt den exemplarischen Aufbau einer *web.xml*-Datei für den Einsatz der *Hallo-Welt*-Applikation:

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
    version="2.4">

  <!-- Faces Servlet -->
  <servlet>
    <servlet-name>HelloWorld-Faces Servlet</servlet-name>
    <servlet-class>
      javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>HelloWorld-Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing 5.1: Beispiel eines Deployment-Deskriptors

Eine JSF-Anwendung ist so ausgelegt, dass *jeder* Faces-Request über ein Servlet geschickt wird und dort eine weitere Verarbeitung und eine Weiterleitung auf die eigentliche JSF-Seite vorgenommen wird. Das Faces-Servlet ist somit die zentrale Anlaufstelle eines jeden Requests. Kenner von Entwurfsmustern (*Design Patterns*) haben hier bereits das sogenannte *FrontController-Pattern* erkannt. Dieses besagt, dass sämtliche Anfragen an einen zentralen Controller geschickt werden. Dieser ist für erforderliche Dienste wie Security, Anstoßen von Geschäftsprozessen oder das Anzeigen einer gültigen View verantwortlich.

Damit das Servlet bei jedem Request angesprochen wird, ist ein Mapping angelegt, das sämtliche Anfragen unterhalb des Verzeichnisses *faces* an das Servlet weiterreicht. Genau hier liegt oftmals ein Fehler, den viele Neueinsteiger beim Testen der ersten JSF-Anwendung machen. Sobald später erste JSF-Seiten erstellt wurden, dürfen diese nicht direkt aufgerufen werden, sondern ausschließlich über das Faces-Servlet. Doch dazu später mehr.

Die Angabe, dass sämtliche Requests mit der Url */faces* beginnen müssen, kann natürlich auch dahingegend angepasst werden, dass eine andere Kennzeichnung oder ein Muster mit Hilfe einer Dateiendung verwendet wird. Z.B. könnten alle Requests mit dem Muster **.faces* an das Faces-Servlet weitergeleitet werden. Mehr dazu lernen Sie in Kapitel 6.1 Konfigurationsdateien.

Wichtig ist jedenfalls, dass die Angabe des Servlet-Names im Bereich Mappings genau gleich benannt wird wie im Bereich des Servlets. Eventuelle Groß- und Kleinschreibungen sollten beachtet werden.

Die Anwendungskonfigurationsdatei

Jede Faces-Anwendung hat zusätzlich zum Deployment-Deskriptor noch ihre eigene Konfigurationsdatei *faces-config.xml*. In späteren Kapiteln werden Sie sehr detailliert sehen, wofür eine separate Konfigurationsdatei nützlich sein kann. Als Ausblick kann bereits gesagt werden, dass hierüber u.a. die Navigation der Anwendung festgelegt wird sowie das zentrale Beanmanagement erfolgt. Als ersten Schritt genügt es jedoch vorläufig, eine leere Hülle wie in Listing 5.2 anzulegen und im *WEB-INF*-Verzeichnis der Anwendung zu speichern.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>


</faces-config>
```

Listing 5.2: Eine noch leere Anwendungskonfigurationsdatei

Nachdem beide notwendigen Konfigurationsdateien angelegt wurden, sollte ihre Verzeichnisstruktur wie in Abbildung 5.4 aussehen.

In Abbildung 5.4 ist die Verzeichnisstruktur der *Hallo-Welt*-Anwendung zu sehen. Diese Verzeichnisstruktur ist typisch für J2EE-Projekte. Die J2EE-Spezifikation definiert unter anderem, an welcher Stelle in einem Verzeichnisbaum die einzelnen Klassen, Bibliotheken sowie Deskriptoren zu finden sind. Dabei ist festgeschrieben, dass es im Anwendungsverzeichnis immer ein Verzeichnis *WEB-INF* (Großschreibung beachten) geben muss. Darin befindet sich der so genannte Deployment Deskriptor *web.xml*, der grundlegende Angaben zu Servlets sowie Initialisierungsparameter enthält. Unterhalb des *WEB-INF*-Verzeichnisses existiert ein *lib*-Verzeichnis, in dem alle benötigten jar-Bibliotheken zu finden sind. Klassen selbst sind im *classes*-Verzeichnis abgelegt. Nicht definiert dagegen ist die Ablage der Anwendungskonfigurationsdatei *faces-con-*

fig.xml, die Angaben für JavaServer Faces-Anwendungen bereithält. Es empfiehlt sich jedoch, diese auf gleicher Ebene zur Datei *web.xml* im *WEB-INF*-Verzeichnis abzulegen.

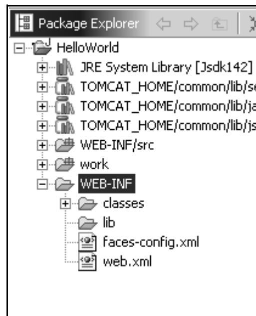


Abbildung 5.4: Verzeichnisstruktur

5.5 Benötigte jar-Dateien hinzufügen

Bis zum jetzigen Zeitpunkt existiert lediglich ein J2EE-konformer Workspace, in dem zwei Konfigurationsdateien vorhanden sind. Damit in dieser Umgebung eine JavaServer Faces- Anwendung laufen kann, sind im Vorfeld noch folgende Bibliotheken zu installieren:

- JavaServer Faces: Alle Dateien im Unterverzeichnis lib des JavaServer Faces- Installationsverzeichnisses (jar-Bibliotheken und dtd-Dateien) müssen in das WEB-INF/ lib-Verzeichnis der Anwendung kopiert werden.
- Web Services Pack: Ebenso sind aus dem jstl/lib-Verzeichnis des Web Services Developer Packs die zwei Bibliotheken jstl.jar und standard.jar in das WEB-INF/ lib-Verzeichnis der Anwendung zu übernehmen.
- Zur Unterstützung der aktuellen JSP- und Servletspezifikation werden die jar-Bibliotheken jsp.jar und servlet.jar benötigt. Diese werden jedoch bei einem Tomcat-Projekt unter Eclipse automatisch eingebunden. Bei einer anderen Entwicklungsumgebung sind diese Bibliotheken gegebenenfalls separat einzubinden.

Damit sind alle notwendigen Bibliotheken vorhanden, um ein JavaServer Faces-Projekt mit einem Standalone-Tomcatserver zu entwickeln und auszuführen. Die installierten jar-Bibliotheken müssen unter der Entwicklungsumgebung Eclipse zusätzlich noch in den so genannten *Build-Pfad* aufgenommen werden, damit der Compiler die Bibliotheken bei einem Kompilierungslauf mit berücksichtigt. Der Build-Pfad wird eingestellt, indem mittels der rechten Maustaste auf dem Projekt *JSFTraining* im *Package Explorer* über *Properties* ein Einstellungsfenster geöffnet wird, in dem es dann einen Abschnitt *Java Build Path* gibt.

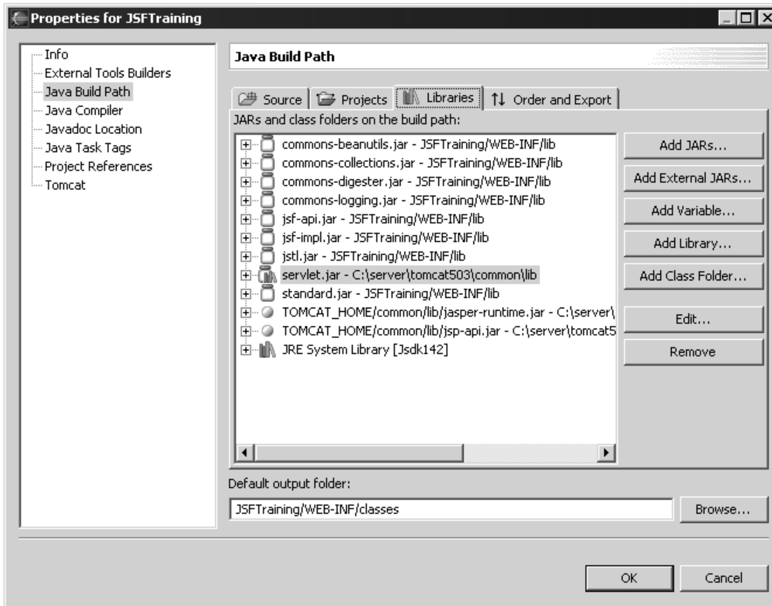


Abbildung 5.5: Einstellen des Build-Pfades

In Eclipse wird zwischen einem Klassenpfad zur Laufzeit und einem Klassenpfad zur Kompilierung von Projekten unterschieden. Während es für das Ausführen von Anwendungen genügt, dass die entsprechenden jar-Bibliotheken im *WEB-INF/lib* bzw. *WEB-INF/classes*-Verzeichnis vorhanden sind, muss für einen Kompilierungslauf explizit angegeben werden, welche Bibliotheken dabei mit dazugezogen werden sollen.

5.6 JavaBeans und Modellobjekte

JavaBeans, Modellobjekt, Managed-Bean; diese Begriffe werden bei ihrer Arbeit an und mit JSF regelmäßig auftauchen. Einerseits ist es irgendwie alles das Gleiche, andererseits gibt es doch einige entscheidende Unterschiede. Im Detail wird hierauf in Kapitel 6.4 näher eingegangen. Zunächst sollte es erst einmal genügen, wenn Sie davon ausgehen, dass ein Modellobjekt vom Aufbau her einem JavaBean entspricht und in JSF zur Datenspeicherung verwendet wird. Bezugnehmend auf das MVC-Pattern stellt das Modellobjekt damit das Modell dar, das für die Datenhaltung der in der View benötigten Daten verantwortlich ist.

Als ersten Schritt auf dem Weg zur eigenen JSF-Anwendung wird somit ein JavaBean geschrieben, das die benötigten Daten bereitstellt und diese über setter- und getter-Methoden manipulieren lässt.

```
package com.edu.jsf.bsp.bean;

import java.util.Date;

/**
 * ein einfaches Besucher-Bean
 */
public class VisitorBean {

  private String firstname;
  private String lastname;
  private Date birth;
  private int age;

  /**
   * liefert den Vornamen zurück
   */
  public String getFirstname() {
    return firstname;
  }

  /**
   * liefert den Nachnamen zurück
   */
  public String getLastname() {
    return lastname;
  }

  /**
   * liefert das Alter zurück
   * Dieses wird als "Geschäftsprozess"
   * ausnahmsweise direkt im Bean berechnet.
   */
  public int getAge() {
    long curDate = System.currentTimeMillis();
    long visitorDate = birth.getTime();
    long lAge = curDate - visitorDate;

    long lAgeYear = lAge / 1000; // Alter in Sekunden
    lAgeYear = lAgeYear / 60; // Alter in Minuten
    lAgeYear = lAgeYear / 60; // Alter in Stunden
    lAgeYear = lAgeYear / 24; // Alter in Tagen
    lAgeYear = lAgeYear / 365; // Alter in Jahren

    age = (int)lAgeYear;
    return age;
  }

  /**
   * liefert das Geburtsdatum zurück
   */
```

```
    public Date getBirth() {
      return birth;
    }

    /**
     * setzt einen neuen Vornamen
     */
    public void setFirstname(String string) {
      firstname = string;
    }

    /**
     * setzt einen neuen Nachnamen
     */
    public void setLastname(String string) {
      lastname = string;
    }

    /**
     * setzt ein neues Alter
     */
    public void setAge(int i) {
      age = i;
    }

    /**
     * setzt ein neues Geburtsdatum
     */
    public void setBirth(Date date) {
      birth = date;
    }
  }
```

Listing 5.3: Ein einfaches Besucher-Bean

In diesem Beispiel wurde die Klasse `VisitorBean` im Package `com.edu.jsf.bsp.bean` angelegt. Die »Anwendungslogik«, sprich die Berechnung des Alters, erfolgt in der getter-Methode des Alters selbst. Ein solches Design ist natürlich in »richtigen« Anwendungen nicht zu empfehlen. Vielmehr erfolgt bei einem Zugriff auf Geschäftslogik ein Aufruf an eine entsprechende Service-Klasse oder sogar ein EJB-Aufruf. Um das Beispiel jedoch möglichst einfach zu halten, ist in diesem Fall die Berechnung des Altes im Bean selbst realisiert.

Die Klasse `VisitorBean` stellt ein gewöhnliches JavaBean dar. Es besitzt private Variablen für den Vor- und Nachname sowie für das Geburtsdatum und das Alter, die mit öffentlichen setter- und getter-Methoden abgefragt und gesetzt werden können. Besonderheiten, die JavaServer Faces betreffen, sind hierbei (noch) nicht enthalten.

5.7 Bean Management

Wie Sie im Laufe Ihrer Arbeit mit JavaServer Faces noch sehr detailliert erfahren werden, ist das Bean Management in JavaServer Faces ein großer Vorteil und eine erhebliche Arbeitserleichterung für einen Anwendungsentwickler. Mittels des Bean Managements ist es möglich, benötigte Beans deklaratorisch in der Anwendungskonfigurationsdatei zu hinterlegen. Danach kann in JSF-Seiten jederzeit auf diese Beans zugegriffen werden, ohne sich Gedanken um die rechtzeitige und korrekte Erzeugung machen zu müssen. Konkret geschieht dies, indem in der Anwendungskonfigurationsdatei für JavaServer Faces-Anwendungen deklariert wird, welches Bean mit welchem Namen im Laufe der Anwendung bereitgestellt werden muss. Die Konfigurationsdatei (Application Configuration File) liegt standardmäßig im *WEB-INF* Verzeichnis des Projektes und trägt die Bezeichnung *faces-config.xml*. Im Gegensatz zu herkömmlicher JSP-Programmierung ist durch die Verwendung von so genannten *Managed-Beans* ein großer Vorteil darin zu suchen, dass durch das deklarative Einbinden der Beans eine separate `<jsp:useBean>`-Angabe wie bei konventioneller JSP-Programmierung nicht mehr notwendig ist. Sobald ein Bean in der Anwendungskonfigurationsdatei einmal hinterlegt wurde, steht es automatisch auf jeder JSF-Seite zur Verfügung. JavaServer Faces kümmert sich um die rechtzeitige Bereitstellung der Objekte (daher der Begriff *Managed Bean)*.

Eine einfache *faces-config.xml*-Datei, die eine Deklaration für die Klasse `VisitorBean` beinhaltet, ist in Listing 5.4 abgebildet:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

  <managed-bean>
    <managed-bean-name>Visitor</managed-bean-name>
    <managed-bean-class>
       com.edu.jsf.bsp.bean.VisitorBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

</faces-config>
```

Listing 5.4: faces-config.xml mit einer Managed-Bean-Angabe

Die Anwendungskonfigurationsdatei wird beim Start der Anwendung eingelesen. Änderungen an der Konfigurationsdatei bedürfen daher immer eines Neustarts der Webanwendung. Im gezeigten Beispiel wird ein Visitor-Bean im Gültigkeitsbereich *Session* abgelegt. Dies bedeutet, dass während der gesamten Lebensdauer einer Session

eine Instanz der Klasse `VisitorBean` für Zugriffe zur Verfügung steht. Nachdem eine Session ungültig wurde, greifen Zugriffe auf das Bean jedoch nicht ins Leere, sondern es wird bei Bedarf eine neue Instanz der Klasse angelegt. Das Bean ist über den Bezeichner `Visitor` in Faces-Seiten zugreifbar.

5.8 Erzeugen der JSF-Seiten

Im Unterschied zur klassischen JSP-Entwicklung, in der direkt mittels Expressions und Skriplets Programmcode in die Seite eingebaut wurde, arbeitet JavaServer Faces komplett basierend auf Taglibs. Sämtliche UI-Komponenten sowie deren Verhalten sind in den Tag-Bibliotheken gekapselt. Ein Webautor muss sich somit an die Verwendung der Taglibs gewöhnen und diese konsequent bei der Seitenerstellung einsetzen. Dies ist jedoch nicht so dramatisch, wie es sich im ersten Moment anhört. Künftig wird es genügend Toolhersteller auf dem Markt geben, die mit geeigneten Oberflächen eine Seitenbearbeitung zulassen, ohne dass ein Benutzer sich direkt mit den einzelnen Tags befassen muss.

Um einen ersten Eindruck der Verwendung der JSF-Taglibs zu bekommen, hier der Quellcode der ersten Seite, die den Vor- und Nachnamen sowie das Geburtsdatum in einem Formular abfragt.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
  <head>
    <title>Hallo Welt</title>
  </head>

  <body>
  <h3>Hallo-Welt-Beispiel</h3>
  <i>Dieses Beispiel zeigt ein erstes kurzes JSF-Beispiel</i>
  <br><br>
  Bitte geben Sie hier Ihre Daten ein:
  <f:view>
    <h:form>
      Vorname: <h:inputText value="#{Visitor.firstname}" /><br>
      Nachname: <h:inputText value="#{Visitor.lastname}" /><br>
      Geburtsdatum:
        <h:inputText value="#{Visitor.birth}">
          <f:convertDateTime dateStyle="short" type="date" />
        </h:inputText>
      <br><br>
      <h:commandButton action="success" value="Submit" />
    </h:form>
  </f:view>
</body>
</html>
```

Listing 5.5: Eine erste JSF-Seite

Sollten Sie bisher noch nie mit Taglibs gearbeitet haben, ist diese Darstellung sicherlich ein wenig gewöhnungsbedürftig. Mit der Zeit jedoch werden sie alle wichtigen Befehle von JavaServer Faces kennen und anwenden lernen. Hintergrund der Verwendung von Taglibs ist unter anderem der, dass dadurch das Rendering, also die eigentliche Darstellung, ausgelagert werden kann. Damit können Sie z. B. sehr einfach ein vorhandenes Render-Kit durch ein anderes ersetzen, ohne damit die komplette Anwendung neu entwickeln zu müssen. Mit Hilfe der Taglibs beschreiben Sie quasi Ihre Seite.

In Listing 5.5 ist zu sehen, dass z. B. ein Tag `<h:inputText>` verwendet wird. Damit wird ein Eingabefeld definiert, dessen Werte aus einem Modellobjekt zu beziehen sind. Aus welchem Modellobjekt, ist durch das Attribut `value` bestimmt. JSF weiß somit, welche Werte angezeigt werden sollen bzw. wie die eingegebenen Werte beim Abschicken des Formulars in das jeweilige Modellobjekt zurückgeschrieben werden sollen.

Bei der Eingabe des Geburtsdatums fällt auf, dass ein verschachteltes Tag verwendet wird. Dabei kommt ein so genannter *Konverter* zum Einsatz. Dieser definiert, dass das eingegebene Datum im Stil `short` eingegeben wird (also in der Form Tag.Monat.Jahr). Damit weiß auch das Framework, wie die textuelle Eingabe eines Benutzers zu einem Datum umgewandelt werden kann.

Wichtig ist an dieser Stelle schon einmal der Hinweis, dass sämtliche JSF-Tags stets durch `ein <f:view>`-Tag umgeben sind. Die Eingaben der ersten Seite werden per Post-Request an den Server geleitet und auf der folgenden zweiten Seite wieder zur Anzeige gebracht:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
  <head>
    <title>Hallo Welt</title>
  </head>

  <body>
  <h3>Hallo-Welt-Beispiel</h3>
  <i>Dieses Beispiel zeigt ein erstes kurzes JSF-Beispiel</i>
  <br><br>
  Ergebnis Ihrer Eingabe:
  <f:view>
    <br><br>
    Herzlichen Glückwunsch,
    <h:outputText value="#{Visitor.firstname}" />
    <h:outputText value="#{Visitor.lastname}" />,
    die erste JavaServer Faces-Anwendung funktioniert.
    <br><br>
    Ihr berechnetes Alter: <h:outputText value="#{Visitor.age}" />
  </f:view>
</body>
</html>
```

Listing 5.6: Die Ausgabeseite

Auch auf der Ausgabeseite, die in Listing 5.6 abgebildet ist, kommen wiederum Taglibs zum Einsatz. Diesmal wird z.B. eine reine Ausgabekomponente `<outputText>` verwendet. Damit ist an dieser Stelle keine Eingabe mehr möglich, es erfolgt lediglich eine textuelle Ausgabe der angegebenen Werte.

In beiden gezeigten Beispielen fällt auf, dass, sobald auf Werte von zugrunde liegenden Beans zugegriffen wird, die Syntax `#{...}` verwendet wird. Damit wird angezeigt, dass es sich hierbei nicht um einen textuellen Wert handelt, der einfach ausgeben werden soll, sondern um einen Ausdruck, der entsprechend interpretiert werden muss. Auch fällt in den Beispielen auf, dass an keiner Stelle explizit angegeben wurde, ein Objekt der Klasse `VisitorBean` zu erzeugen. Dies erfolgt automatisch, da in der Anwendungskonfigurationsdatei hinterlegt wurde, dass ein solches Objekt bereitgestellt werden soll. Dieses kann dann wie gezeigt über den Bezeichner `Visitor` angesprochen werden.

5.9 Anlegen einer index-Datei

JSF-Seiten müssen immer über das Faces-Servlet geleitet werden. Ein direkter Aufruf von JSF-Seiten kann zu Laufzeitfehlern führen. Es muss daher immer beachtet werden, dass das im Deployment Deskriptor hinterlegte Mapping zum Tragen kommt. In der Beispielanwendung bedeutet dies, dass die Anfrage nach der ersten Seite ein */faces* in der URL benötigt. Da dies jedoch gerne vergessen wird, empfiehlt es sich, im Startverzeichnis der Anwendung eine index-Datei zu hinterlegen, die auf die eigentliche JSF-Startseite weiterleitet, jedoch mit dem korrekten URL-Aufruf.

```
<html>
  <head>
    <meta http-equiv="Refresh"
     content="0;URL=/JSFTraining/faces/kap_05/eingabe.jsp">
  </head>
  <body>
  </body>
</html>
```

Listing 5.7: Indexdatei

Sollte somit eine Anfrage über die URL *http://localhost:8080/JSFTraining/kap_05/* erfolgen, wird diese Anfrage automatisch über das Faces-Servlet zur Eingabeseite weitergeleitet. Dies ist für einen Neueinsteiger meist recht ungewöhnlich, da es in der Webanwendung kein physikalisches Verzeichnis */faces* gibt. Diese Angabe dient jedoch lediglich dazu, die Anfrage an das Faces-Servlet weiterzuleiten. Von dort erfolgt dann die Weiterleitung zur eigentlichen JSF-Seite.

5.10 Festlegung der Navigation

Bisher wurden zwei Faces-Seiten erzeugt, die jedoch noch in keinem Verhältnis zueinander stehen. Betrachtet man das `commandButton`-Tag auf der Seite *eingabe.jsp* genauer, erfolgt beim Klicken auf den Button nur eine Aktion mit der Bezeichnung `success`. Wie das Framework beim Eintreten einer solchen Aktion zu reagieren hat, wird in einer Beschreibung der Navigation hinterlegt. Die Navigation selbst ist auch wieder in der Anwendungskonfigurationsdatei *faces-config.xml* zu finden. Für die *Hallo Welt*-Anwendung existiert genau ein Navigationspfad, nämlich wenn der Benutzer den Knopf drückt und auf die folgende Seite geleitet werden soll.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>

  <navigation-rule>
    <from-view-id>/kap_05/eingabe.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/kap_05/ausgabe.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>Visitor</managed-bean-name>
    <managed-bean-class>
       com.edu.jsf.bsp.bean.VisitorBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

</faces-config>
```

Listing 5.8: Faces-config.xml mit Navigationsregeln

In der Anwendungskonfigurationsdatei in Listing 5.8 wurde eine Regel definiert, dass in dem Fall, wenn auf der Seite *eingabe.jsp* (im entsprechenden Unterverzeichnis) ein Rückgabewert `success` erzeugt wird, auf die Seite `ausgabe.jsp` weitergeleitet wird.

Vorteil dieser Vorgehensweise ist, dass Änderungen an der Navigation nur Änderungen in einer Konfigurationsdatei nach sich ziehen, Änderungen am Quellcode sind nur in Ausnahmefällen notwendig.

Um nochmals explizit darauf hinzuweisen: In der eigentlichen JSF-Seite wurde keine direkte Angabe getroffen, auf welche Seite weitergeleitet werden soll, nachdem der Button gedrückt wurde. Es wurde lediglich festgelegt, dass mit Hilfe des Bezeichners `success` in der Konfigurationsdatei nach der passenden Folgeseite gesucht werden soll.

5.11 Start der Anwendung

Sind obige Schritte komplett durchlaufen, kann der Tomcat-Server für einen ersten Test der Anwendung gestartet werden. Werden beim Starten der Anwendung keine Fehler angezeigt (Ausgabe in der Serverkonsole), kann die Anwendung wie folgt über einen Browser aufgerufen werden:

```
http://localhost:8080/JSFTraining/faces/kap_5/eingabe.jsp
```

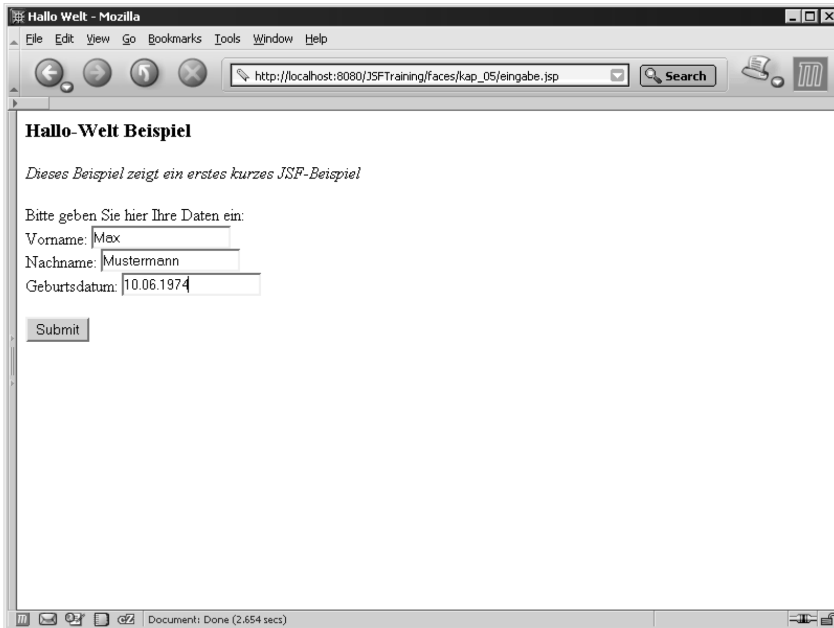


Abbildung 5.6: Eingabemaske der Anwendung

Beachten Sie bei der Eingabe der Url unbedingt, dass die JSF-Seite nicht direkt angesprochen wird, sondern über das Faces-Servlet geleitet wird (bzw. ein Aufruf über die Index-Seite erfolgt). Ein direktes Aufrufen einer JSF-Seite (in diesem Fall über die Adresse *http://localhost:8080/JSFTraining/kap_05/eingabe.jsp* führt zu einem Fehler, da das Faces-Framework noch nicht initialisiert wurde. Dieser Fehler passiert oftmals Neueinsteigern in JSF, da man als JSP-Entwickler es immer gewohnt war, Seiten direkt anzusprechen. Durch die Index-Datei, die ebenfalls angelegt wurde, kann die Anwendung natürlich auch über die Adresse *http://localhost:8080/JSFTraining/kap_05/* gestartet werden.

Nach Eingabe des Vor- und Nachnamens sowie des Geburtsdatums kann das Formular über den `Submit`-Button abgeschickt werden. Danach sollte folgende Ausgabeseite erscheinen:

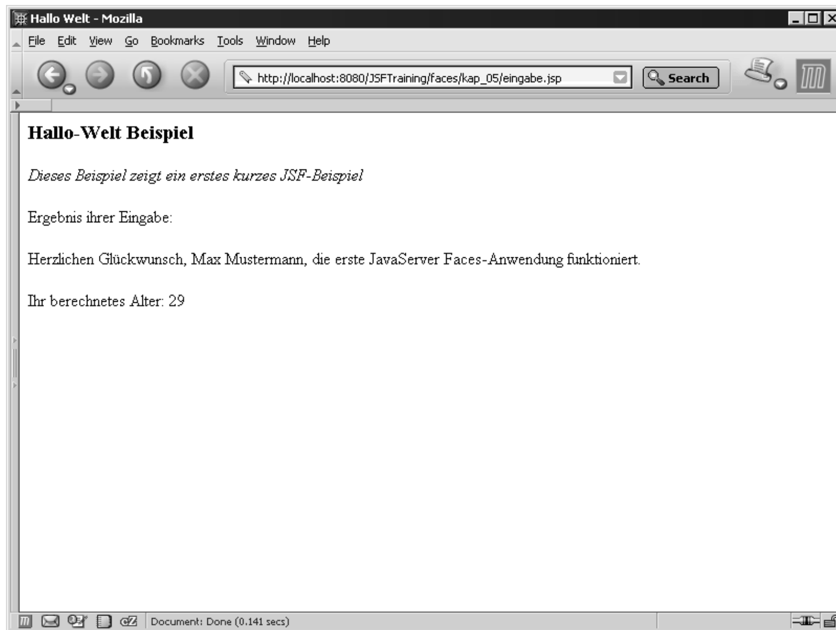


Abbildung 5.7: Ausgabeseite der Beispielanwendung

An dieser Stelle können Sie sich zunächst einmal beglückwünschen. Sie haben soeben Ihre erste JSF-Anwendung erfolgreich realisiert!

Damit haben Sie in den letzten Abschnitten einen ersten Eindruck und ein erstes Gefühl dafür bekommen, was es heißt, JSF-Anwendungen zu entwickeln. Sie haben einen ersten Einblick in die Konzepte und in die Architektur von JSF bekommen. Es gibt mit Sicherheit noch viel mehr, das auch in einem ersten Schritt erwähnenswert wäre. In den nächsten Kapiteln wird daher Schritt für Schritt jede Facette von JavaServer Faces genau erläutert und an Beispielen demonstriert. Die folgenden Beispiele können Sie entweder in einem neuen Workspace durchführen oder aber in Ihren bestehenden Workspace mit einbauen. Alle Kurzbeispiele sind so gewählt, dass Sie auch zur gleichen Zeit im gleichen Workspace aktiv sein können, ohne sich gegenseitig zu beeinflussen. Sämtliche Beispiele sind natürlich ebenfalls auf der beigelegten CD enthalten.

5.12 Ausblick

Eine erste JSF-Anwendung haben Sie soeben erfolgreich realisiert. Sie können natürlich jetzt selbst noch Änderungen an der Anwendung vornehmen, um so per »Trial-and-Error« erste Erfahrungen mit der neuen Technologie zu sammeln. Sie werden in den folgenden Kapitel weitere nützliche Funktionen von JSF kennen und anwenden ler-

nen. Im gezeigten Beispiel wäre unter anderem denkbar, eine Feldprüfung auf der Eingabeseite zu hinterlegen. Eine Berechnung des Alters macht natürlich nur Sinn, wenn ein gültiges Datum hinterlegt wurde. Hierfür stellt JSF bereits eine entsprechende Funktionalität bereit (über so genannte *Validatoren*). Aber auch die Ausgabedarstellung kann mittels JSF komfortabel angepasst werden. So kann die Ausgabe des Alters auch mit Dezimalstellen erfolgen, wenn statt eines `int`-Datentypes ein `double`-Datentyp verwendet wird. Die Anzahl der Nachkommastellen kann hierbei dann über ein entsprechendes Format angegeben werden.

Wenn Feldprüfungen durchgeführt werden, kann es natürlich dazu kommen, dass auch ein Fehler auftritt. Dieser muss dem Benutzer auch mitgeteilt werden. D.h. es muss in der Webseite ein Bereich für Fehlerausgaben vorgesehen werden. Auf das Verhalten im Fehlerfall wird in den folgenden Kapiteln natürlich ebenfalls eingegangen.

5.13 Fazit

Ziel dieses Kapitels war es, eine erste eigene Webanwendung basierend auf JavaServer Faces zu entwickeln. Dabei wurden alle relevanten Arbeitsschritte, die bei einer Arbeit mit JavaServer Faces auftauchen, kurz besprochen und aufgezeigt.

Es wurde gezeigt, dass bei der Einrichtung der Entwicklungsumgebung einige Dinge zu beachten sind. Ebenso wurde die zentrale Konfigurationsdatei *faces-config.xml* erläutert und deren Verwendung vorgestellt.

Das Managed-Bean-Konzept ist ein wichtiger Bestandteil in JavaServer Faces. Es regelt, dass notwendige Modellobjekte jederzeit zur Verfügung stehen bzw. bei Bedarf automatisch erzeugt werden. Da die Verwaltung und die Erzeugung dieser Beans automatisch durch das Framework geschieht, werden diese als »Managed« bezeichnet.

Die direkte Arbeit in einer Faces-Seite ist mit Sicherheit gewöhnungsbedürftig, falls im Vorfeld noch nie mit Tagbibliotheken wie der JSTL gearbeitet wurde. Mit Aufkommen von visuellen Entwicklungswerkzeugen müssen jedoch künftig die Arbeiten nicht mehr unbedingt auf Quellcodeebene erfolgen.

6 JSF im Detail

Kapitelziel

In diesem Kapitel wird ein detailierter Einblick in JavaServer Faces gegeben. Begonnen wird damit, dass die Konfigurationsdateien, die für eine Arbeit mit JavaServer Faces benötigt werden, näher beleuchtet werden. Es wird die Anwendungskonfigurationsdatei *faces-confg.xml* vorgestellt sowie der Deployment Deskriptor, der grundsätzlich für jede Webanwendung basierend auf dem J2EE-Standard benötigt wird.

Des Weiteren wird das Faces-eigene Komponentenmodell erläutert, wobei auch das Rendering-Modell näher beleuchtet wird. Durch die Trennung von Funktionalität und Darstellung existieren so genannte *Renderer* neben den eigentlichen Komponentenklassen, die zunächst einmal von der späteren Darstellung vollkommen losgelöst behandelt werden.

Ein weiteres großes und wichtiges Thema in JavaServer Faces ist das *Bean-Management*, das einen großen Fortschritt gegenüber herkömmlicher JSP-Programmierung darstellt. Damit wird es dem Anwendungsentwickler einfacher gemacht, in einer Webanwendung auf benötigte Modellobjekte zuzugreifen, ohne sich um deren rechtzeitige Erzeugung und Bereitstellung kümmern zu müssen.

Ebenfalls werden das *Navigationskonzept* sowie das *Eventhandling* besprochen. Analog zu Anwendungen basierend auf Java Swing können auch in JavaServer Faces so genannte *Listener* an Komponenten angehängt werden. Diese werden bei Eintreten bestimmter Ereignisse (z. B. bei Betätigen eines Buttons oder aber auch nach einer Werteänderung) aufgerufen.

Da zunehmend moderne Webanwendungen im internationalen Umfeld zum Einsatz kommen, ist der Thematik der Internationalisierung ein eigener Abschnitt gewidmet.

Für die folgenden Beispiele können Sie den in Kapitel 5 angelegten Workspace verwenden und darin die demonstrierten Beispiele nachprogrammieren. Natürlich können Sie auch einen separaten Workspace einrichten. Die Vorgehensweise entspricht der in Kapitel 5 gezeigten Weise.

6.1 Konfigurationsdateien

In JavaServer Faces existiert primär eine Konfigurationsdatei, in der grundlegende Angaben zur Faces-Anwendung enthalten sind. Des Weiteren muss wie bei allen J2EE-konformen Webanwendungen ein so genannter *Deployment Deskriptor* für das Projekt eingerichtet werden. Mit Anpassen bzw. Erstellen dieser zwei Konfigurationsdateien steht einer lauffähigen JSF-Anwendung nichts mehr im Wege. Dabei sind die notwendigen Einstellungen in den Konfigurationsdateien sehr gering, einige wenige Angaben genügen, um bereits erste Webanwendungen ausführen zu können. Im Extremfall würde sogar eine leere Anwendungskonfigurationsdatei genügen, in der keine weiteren Angaben vorhanden sind, sondern lediglich die Tags vorzufinden sind, die zwingend in der DTD vorgeschrieben sind.

Der Deployment Deskriptor

Der Deployment Deskriptor *web.xml* liegt gemäß der J2EE-Spezifikation im *WEB-INF*-Unterverzeichnis des Anwendungsverzeichnisses. In ihm sind u.a. grundlegende Angaben zu Servlets sowie zu Initialisierungsparametern hinterlegt. Das Prinzip von JavaServer Faces ist so ausgelegt, dass sämtliche Anfragen vom Browser immer über ein zentrales Servlet geleitet werden. Von dort aus werden alle weiteren Aufrufe gestartet und letzten Endes auf die nächste JSF-Seite weitergeleitet. Damit alle Requests über das zentrale *Faces-Servlet* geleitet werden, müssen sowohl das Servlet selbst wie auch ein *Servlet-Mapping* im Deployment Deskriptor angelegt werden. Das Muster, bei dem sämtliche Anfragen über eine zentrale Instanz geleitet werden, ist ein bekanntes Entwurfsmuster, das so genannte *Frontcontroller*-Muster.

```
<servlet>
  <servlet-name>JSF-Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>JSF-Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Listing 6.1: Auszug aus der web.xml

In Listing 6.1 ist ein Auszug aus einem Deployment Deskriptor zu sehen. Dem Faces-Servlet mit der Klasse `javax.faces.webapp.FacesServlet` wird ein Servletname `JSF-Servlet` zugewiesen. Dazu wird ein Mapping angelegt, das sämtliche Aufrufe, die ein `/faces/` in der Url angegeben haben, an das JSF-Servlet weiterleitet. Diese Angabe ist zwingend notwendig, direkte Aufrufe von JSF-Seiten sind nicht möglich. Alle Aufrufe

von JSF-Seiten müssen immer über das Faces-Servlet geleitet werden, das wiederum die Aufrufe an die entsprechende JSF-Seite weiterleitet. Dies ist gerade für Neueinsteiger eine häufige Fehlerquelle, wenn eine Faces-Anwendung nicht wie gewollt funktioniert. Es wird dabei meist vergessen, dass alle Angaben über das Faces-Servlet geleitet werden müssen und somit ein direkter Aufruf der Faces-Seite nicht funktionieren kann. In solchen Fällen erscheint im Browser meist ein Stacktrace, der besagt, dass der FacesContext nicht initialisiert sei.

Das in Listing 6.1 gezeigte Beispiel verwendet ein Präfix `/faces`, um Faces-spezifische Requests kenntlich zu machen. Daher wird diese Art des Mappings mit *Prefix-Mapping* bezeichnet. Genauso ist es auch möglich, mit einer speziellen Dateiendung dem Container mitzuteilen, dass es sich um eine Faces-Datei handelt.

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

Listing 6.2: Mapping über die Dateiendung

In Listing 6.2 wird hierbei die Endung **.faces* verwendet. Dies wird als *Extension Mapping* bezeichnet. Natürlich kann das Mapping auch in einer anderen Form deklariert werden, indem z.B. eine Dateiendung **.jsf* zur Kennzeichnung verwendet wird. Ab der JSP-Spezifikation 2.0 ist die Dateiendung **.jsf* speziell für Faces-Seiten reserviert.

Die Anwendungskonfigurationsdatei

Die Anwendungskonfigurationsdatei *faces-config.xml* ist eine Faces-spezifische Konfigurationsdatei. In ihr werden Angaben zur Navigation, zu den in JSF verwendeten Modellobjekten sowie zu benutzerdefinierten Komponenten und Renderern hinterlegt. Auf die einzelnen Abschnitte der Konfigurationsdatei wird in den folgenden Kapiteln Stück für Stück eingegangen. Es empfiehlt sich, die Anwendungskonfigurationsdatei *faces-config.xml* im Verzeichnis *WEB-INF* abzulegen, worin auch der Deployment Deskriptor *web.xml* zu finden ist. Dennoch kann es unter Umständen auch einmal notwendig sein, die Anwendungskonfigurationsdatei an einem anderen Ort abzulegen. Dabei sucht das Framework in folgender Reihenfolge nach einer passenden Datei:

- Die Bibliotheken im Verzeichnis *WEB-INF/lib* werden nach einer Ressource */META-INF/faces-config.xml* durchsucht. Wird in einer beliebigen jar-Datei eine solche Ressource gefunden, wird diese entsprechend geladen.
- Über einen Initialisierungsparameter im Deployment Deskriptor kann ein Pfad (bzw. mehrere Pfade, die durch Kommata getrennt sind) angegeben werden, in dem nach der Anwendungskonfigurationsdatei gesucht wird (siehe Listing 6.3).

- Es wird direkt nach einer Datei *faces-config.xml* im Verzeichnis *WEB-INF* gesucht.

```
<context-param>
  <param-name>javax.faces.application.CONFIG_FILES</param-name>
  <param-value>/configfiles/faces-config.xml</param-value>
</context-param>
```

Listing 6.3: Auszug aus einer web.xml

In Listing 6.3 wurde die Anwendungskonfigurationsdatei *faces-config.xml* in einem Unterverzeichnis */configfiles* der Anwendung abgelegt. Durch Angabe eines Kontextparameters weiß das Framework damit, wo nach dieser Datei gesucht werden muss. Aus dieser Angabe ist des Weiteren zu erkennen, dass der komplette Pfad inklusive des Dateinamens angegeben ist. Daher ist es auch möglich, durch diese Angabe im Deployment Deskriptor einen anderen Namen für die Faces-Konfigurationsdatei zu verwenden.

Die Anwendungskonfigurationsdatei ist eine normale Xml-Datei, was bedeutet, dass diese mit einem normales Text-Editor oder auch einem speziellen Xml-Editor bearbeitet werden kann. Zusätzlich bieten verschiedene Hersteller bereits die Möglichkeit an, mittels einer graphischen Oberfläche diese Datei zu bearbeiten. Mit der *Faces-Konsole* steht zudem eine freie Oberfläche zur Verfügung, mit der bequem die Konfiguration bearbeitet werden kann. Näheres dazu ist auch im Anhang zu finden.

Die Konfigurationsdateien werden meist während des Startvorganges geladen. Je nach Einstellung kann auch die Faces-Konfigurationsdatei bei Start der Webanwendung oder beim ersten Request an die Anwendung selbst eingelesen werden. Sind die Anwendungsdateien fehlerhaft (z.B. ein Parsing-Fehler), hat dies zur Folge, dass die gesamte Webanwendung nicht zur Verfügung steht.

Mehrere Anwendungskonfigurationsdateien

Natürlich ist es auch möglich, mehrere Anwendungskonfigurationsdateien in einer einzelnen Webanwendung zu haben. Die Spezifikation legt zwar eine Reihenfolge fest, in der diese gesucht und eingelesen werden, sie besagt aber nicht, dass es lediglich eine Datei davon geben darf. Dies hat den enormen Vorteil, dass z.B. Bibliotheken mit benutzerdefinierten Komponenten als jar-Datei verwendet werden können, die selbst wiederum bereits eine Anwendungskonfigurationsdatei enthalten. In dieser sind die notwendigen Angaben bereits hinterlegt. Somit kann durch diesen Ansatz ein modularer Aufbau einer Webanwendung erreicht werden, in dem verschiedene Komponenten evtl. samt Renderern lediglich durch Einbindung einer Bibliothek hinzugefügt werden können.

6.2 Die JSF-Tag-Bibliotheken

JavaServer Faces verwendet zwei *Tag-Bibliotheken* (auch *Taglibrary* oder kurz *Taglib* genannt): die *html_basic* Taglib und die *jsf_core* Taglib.

Die html_basic Taglib stellt Tags für die Darstellung von HTML-UI-Komponenten bereit. UI-Komponenten (User Interface Komponenten) sind graphische Bestandteile der Benutzeroberfläche (User Interface). So wird beispielsweise in der html_basic Taglib geregelt, welche Attribute und welche Ausprägungen ein Eingabefeld hat und welche Angaben bei der Verwendung dieser UI-Komponenten zwingend erforderlich sind. Die html_basic Taglib hat somit immer etwas mit der graphischen Darstellung von Komponenten zu tun.

Explizit sei jedoch erwähnt, dass in der *html_basic* Taglib nur die Verwendung der UI-Komponenten festgelegt ist. Es ist definiert, welche Attribute in welcher Ausprägung z. B. zur Beschreibung eines Eingabefeldes erforderlich sind und welche Attribute überhaupt zulässig sind. Z. B. existiert für ein Eingabefeld ein Attribut *maxlength*, das die maximale Eingabelänge festlegt. Bei einem Auswahlfeld würde diese Angabe keinen Sinn machen.

Die eigentliche Darstellung, wie genau eine so beschriebene Komponente im Browser letztendlich dargestellt wird, geschieht über so genannte *Renderer*, auf die im Kapitel 7.3 Renderer genauer eingegangen wird.

Die *jsf_core* Taglib dagegen stellt die Basisfunktionalität für das Eventhandling und die Datenvalidierung bereit sowie weitere Grundfunktionalitäten für eine JSF-Applikation. Es ist damit möglich, beispielsweise an Eingabefelder bestimmte *Validatoren* zu knüpfen, die eine Dateneingabe auf gewisse Kriterien überprüfen. Sprich es ist möglich, eine Regel zu hinterlegen, nach der Benutzereingaben automatisch durch das Framework überprüft werden. Genauso können an spezielle UI-Komponenten *Eventlistener* angehängt werden, die bei Eintreten eines Ereignisses einen entsprechenden Event auslösen. Diese Funktionalitäten können über die *jsf_core* Taglib angesprochen werden. Sämtliche Angaben der *jsf_core* Taglib sind unabhängig davon, wie die zugrunde liegenden Komponenten später dargestellt werden. Da eine Darstellung von Komponenten über so genannte Renderer erfolgt, ist die Funktionalität der *jsf_core* Taglib vollkommen losgelöst und unabhängig von einer späteren Darstellung.

Bei Bedarf werden auf einer JSF-Seite weitere Taglibs eingebunden, z. B. bei Verwendung einer Datenbank die *JSTL-Bibliothek* für die Datenbankunterstützung. Diese Bibliotheken sind aber nicht direkter Bestandteil der JavaServer Faces-Technologie, werden jedoch häufig in Zusammenspiel mit JSF-Anwendungen verwendet.

Einbinden der Tag-Bibliotheken in eine JSF-Seite

Um auf die Funktionalitäten der Tag-Bibliotheken zugreifen zu können, genügt es, beide Bibliotheken am Anfang einer JSF-Seite wie übliche Tag-Bibliotheken einzubinden:

```
<%@ taglib uri="http://java.sun.com/jsf/html/" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core/" prefix="f" %>
```

Listing 6.4: Einbinden der JSF-Tag-Bibliotheken

Das Attribut `uri` ist für eine eindeutige Identifikation der Taglib erforderlich. Über das Präfix `h` bzw. `f` wird geregelt, wie die Tags der deklarierten Bibliotheken angesprochen werden, es wird der so genannte Namensraum festgelegt. Das Präfix dient zur genauen Kennzeichnung der Tags, damit der Servlet-Container weiß, welche Tags aus welcher Bibliothek verwendet werden sollen.

Bei Verwendung obiger Präfixe wird z. B. ein Ausgabefeld wie folgt verwendet:

```
<h:inputText id="firstname" value="#{Person.firstname}" />
```

Listing 6.5: Einbindung eines Eingabefeldes

Es ist technisch zwar möglich und erlaubt, eigene Präfixe für die JSF-Bibliotheken zu verwenden, es ist jedoch ratsam, die Kürzel `h` und `f` zu verwenden. Zum einen haben Neueinsteiger in ein JSF-Projekt damit schneller einen Überblick, da man sich schnell an die Verwendung bestimmter Präfixe gewöhnt, zum anderen kann auch schneller aus Beispielen in der Literatur der Transfer auf das eigene Projekt erfolgen.

6.2.1 Core-Tags

Die Core-Tags stellen Funktionalitäten für grundlegende JSF-Abläufe sowie für das Eventhandling und die Datenvalidierung bereit. Die Verwendung der Core-Tags wird an verschiedenen Stellen in diesem Buch erläutert.

Art der Tags	Tagname	Beschreibung
Container Tag	`view`	Root-Tag für alle JSP-Seiten, in denen JSF-Tags verwendet werden. Das Tag `<f:view>` muss dabei sämtliche JSF-Tags umschließen.
	`subview`	`<f:subview>` schließt alle Aktionen ein, die dynamisch aus einer anderen Seite eingebunden werden, z. B. mittels `<c:import>`.

Tabelle 6.1: Übersicht über die Core-Tags

Art der Tags	Tagname	Beschreibung
Eventhandling-Tag	`actionListener`	registriert einen Action-Listener an einer Komponente. Dies kann z. B. ein Button oder ein Hyperlink sein. Beim Auslösen wird ein entsprechendes Event erzeugt.
	`valueChangeListener`	registriert einen Listener an einer Komponente, der auf Werteänderungen reagiert. Komponenten sind alle UI-Komponenten, die von `UIInput` ableiten. Sobald ein neuer Wert für die Komponente eingegeben wurde, wird ein entsprechendes Event erzeugt.
Attribut-Beschreibungs-Tag	`attribute`	fügt Attribute zu einer Komponente hinzu. Dies kann z. B. ein Zahlenformat bei einem Eingabefeld sein, bei dem der eingegebene Wert als Prozentwert zu interpretieren ist.
Facet Tag	`facet`	wird im Zusammenhang mit der panelGrid-Komponente verwendet und regelt z.B., wie Komponenten des Komponentenbaumes aktualisiert werden. Des Weiteren wird durch ein Faces-Tag z. B. eine Kopf- und Fußzeile in einer Tabelle definiert.
Parameter-Tag	`param`	Parameter-Tags können bestimmten Komponenten mitgegeben werden, um zusätzliche Informationen in der HTML-Ausgabe zu erzeugen.
Validierungs-Tags	`validateDoubleRange`	Ein Validierungs-Tag kann zur Überprüfung einer Benutzereingabe an ein Eingabefeld angehängt werden. Das Tag `validateDoubleRange` überprüft dabei, ob der Wert innerhalb eines Bereiches liegt.
	`validateLongRange`	wie `validateDoubleRange`, nur dass in diesem Fall der Bereich innerhalb des Datentyps *long* überprüft wird
	`validateLength`	Die Länge kann bei einem String-Attribut mit Hilfe von `validateLength` auf ihre Gültigkeit hin überprüft werden.
	`validator`	Zusätzlich zu den in JSF bereits enthaltenen Validatoren können eigene Validatoren mit Hilfe des `validator`-Tags eingebunden werden.
Konverter-Tags	`convertDateTime`	registriert einen Datetime-Konverter an einer UI-Komponente.
	`convertNumber`	registriert einen Number-Konverter an einer UI-Komponente.
	`converter`	registriert einen Konverter mittels eines Bezeichners an einer Komponente.

Tabelle 6.1: Übersicht über die Core-Tags (Fortsetzung)

Art der Tags	Tagname	Beschreibung
Ausgabe-Tag	verbatim	dient zur Steuerung von Ausgabetexten. Es kann festgelegt werden, ob Ausgabestrings als Markup oder normaler Text ausgegeben werden.
Tags für Listeneinträge	selectitem	fügt eine UISelectItem-Komponente einer umgebenden Komponente hinzu.
	selectitems	fügt eine UISelectItems-Komponente einer umgebenden Komponente hinzu.

Tabelle 6.1: Übersicht über die Core-Tags (Fortsetzung)

6.2.2 HTML-Tags

Die HTML-Tags definieren in einer JSF-Seite das Aussehen und die Funktionalität der UI-Komponenten. Es wird somit z. B. definiert, wie groß (bzw. lang) ein Eingabefeld in einem Formular ist oder welche Listeneinträge eine Listauswahl aufweist. Es wird aber ebenso festgelegt, wie auf bestimmte Ereignisse (z. B. ein *onmouseover*-Ereignis bei einer Grafik) reagiert werden soll.

Es sei aber nochmals explizit erwähnt, dass die Darstellung der UI-Komponenten *nicht* in der HTML-Tag-Bibliothek selbst geschieht, sondern über so genannte *Renderer* bestimmt wird. Die Tags der HTML-Tag-Bibliothek definieren lediglich, mit welchen Attributen die einzelnen Komponenten ausgestattet sind und welche Tagklasse zum Einsatz kommt, wogegen die entsprechenden Renderer für die letztendliche Darstellung verantwortlich sind. Die Zuordnung des dazugehörigen Renderers kann entweder in der Tagklasse geschehen oder das Rendering findet in der Komponente selbst statt. Details dazu folgen in Kapitel 9. JSF erweitern und anpassen.

Die Tabelle 6.2 zeigt, welche Komponenten von der HTML-Tagbibliothek bereitgestellt werden.

Tag	HTML-Entsprechung	Beschreibung
form	<form>	definiert ein Formular, in dem wiederum weitere Ein- und Ausgabefelder sowie Commandbuttons enthalten sein können.
column	<td>	definiert eine Spalte in einer tabellarischen Auflistung.
commandButton	<input type="wert"> *Wert* kann dabei die Ausprägungen *submit*, *reset* oder *image* aufweisen	definiert einen Button, der beispielsweise Eingaben eines Formulars abschickt oder auch alle Eingaben auf ihren Standardwert zurücksetzt.

Tabelle 6.2: UI-Komponenten in der html_basic Taglib

Tag	HTML-Entsprechung	Beschreibung
commandLink	<a href="Ziel">	zeigt einen Hyperlink an, mit dem entweder eine Aktion ausgelöst wird oder aber auf eine bestimmte Url verwiesen wird.
dataTable	<table>	iteriert über eine Collection, um z. B. mehrere Zeilen innerhalb einer Tabelle darzustellen.
graphicImage	<img …>	definiert ein Bild.
inputText	<input type="text">	definiert ein Texteingabefeld.
inputHidden	<input type="hidden">	Ein verstecktes Eingabefeld dient z. B. der Hinterlegung von Variablen, die in der Oberfläche nicht angezeigt werden.
inputSecret	<input type="password">	Bei Eingabe von Passwörtern erscheinen bei der Eingabe nicht die Buchstaben und Zahlen, sondern lediglich ein »*«-Zeichen.
inputTextarea	<textarea>	definiert ein mehrzeiliges und mehrspaltiges Eingabefeld, in dem ein Benutzer längeren Text eingeben kann.
message	Normaler Text	gibt eine entsprechende Meldung für eine Komponente oder für die komplette Seite aus.
messages	Normaler Text	gibt u. U. mehrere Meldungen nacheinander aus.
outputText	Normaler Text	Ausgabe eines normalen Textes. Der Text kann als Wert direkt hinterlegt werden oder aber sich auf ein *Managed-Bean*beziehen.
outputLabel	<label>	Es wird ein Bezeichner ausgegeben, der einen direkten Bezug auf eine Komponente, z. B. ein Eingabefeld, hat.
outputFormat	Normaler Text	Gemäß der Locale, die aktuell gesetzt ist, wird eine Meldung ausgegeben.
outputLink	<a href="Ziel">	gibt einen externen Link aus, d. h. einen Link aus der JSF-Anwendung heraus.
panelGrid	<table>	Ausgabe einer Tabelle, in der die einzelnen Werte in Zeilen (<tr>) und Spalten (<td>) angeordnet sind.
panelGroup	Keine direkte Entsprechung, aber vergleichbar mit dem colspan-Attribut.	Hiermit können bestimmte Komponenten gruppiert, d.h. zusammengefasst werden. Bei Tabellen bewirkt diese Angabe z.B., dass mehrere Ausgaben innerhalb einer Zelle dargestellt werden.

Tabelle 6.2: UI-Komponenten in der html_basic Taglib (Fortsetzung)

Tag	HTML-Entsprechung	Beschreibung
selectBooleanCheck-box	<input type="check-box">	definiert eine Checkbox. Eine Checkbox kann unabhängig von anderen Checkboxen selektiert und deselektiert werden.
selectManyCheckbox	mehrere Angaben von <input type="check-box">	definiert eine Sammlung von Checkboxen.
selectManyListbox	<select size="x">, wobei x ein Wert größer 0 ist	definiert eine Listbox mit mehreren Auswahlmöglichkeiten. Dabei werden alle Auswahlmöglichkeiten auf einmal angezeigt.
selectManyMenu	<select size="1">	definiert wie selectManyListbox auch wieder eine Liste mit mehreren Auswahlmöglichkeiten, allerdings ist das Ganze als Combobox hinterlegt.
selectOneListbox	<select size="x">, wobei x ein Wert größer 0 ist	definiert wie selectManyListbox eine Listbox, allerdings kann in diesem Fall maximal ein Element ausgewählt werden.
selectOneMenu	<select size="1">	Definiert ähnlich wie selectManyMenu eine Combobox, aus der genau ein Element ausgewählt werden kann.
selectOneRadio	<input type="radio">	Definiert eine Auswahlmöglichkeit, die als Radio-Button dargestellt wird.

Tabelle 6.2: UI-Komponenten in der html_basic Taglib (Fortsetzung)

Zusätzlich zu den bereits festgelegten Tags können weitere Komponenten entworfen und für die Verwendung in JSF-Seiten eingebaut werden. Mehr dazu finden Sie im Kapitel 9. Mittlerweile existieren im Internet bereits eine Vielzahl von Open-Source-Initiativen, die eigene Komponenten bereitstellen. Diese können dann über entsprechende Tags ebenfalls in eine eigene Anwendung integriert werden.

6.3 Das UI-Komponentenmodell

Die Komponentenarchitektur von JavaServer Faces wurde so konzipiert, dass eine Trennung von Modell, Darstellung und Verarbeitung gegeben ist. Damit ist es u.a. möglich, Ausgaben nicht nur in Html, sondern beispielsweise auch in Xml oder Wml (Markup-Sprache, die in Wap-Handys verwendet wird) ohne große technische Umstellungen zu erzeugen. Es muss dazu lediglich ein entsprechendes Render-Kit geschrieben werden, die Basisfunktionalität der Komponenten bleibt im Normalfall unverändert erhalten.

Das Zusammenspiel von Taglibs, Komponenten, Renderern und Modellobjekten verdeutlicht die Abbildung 6.1:

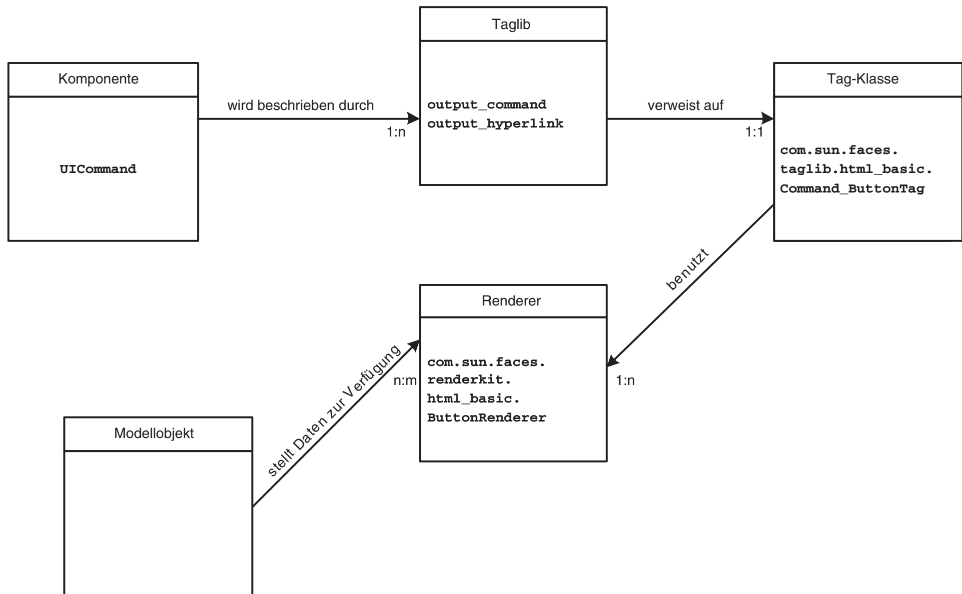


Abbildung 6.1: Zusammenspiel der Komponenten

UI-Komponenten sind in JavaServer Faces durch eigene Komponentenklassen beschrieben. So existieren eine Vielzahl von Standardkomponenten, die durch eigene benutzerdefinierte Komponenten erweitert werden können. Diese Komponenten wiederum können auf verschiedene Weisen dargestellt werden. So kann eine Komponente `UICommand` beispielsweise als Commandbutton oder auch als Hyperlink dargestellt werden. Die einzelnen Tags, die ein JSF-Entwickler dabei verwenden kann, sind in der *html_basic*-Taglib hinterlegt. Darin ist beschrieben, welche Attribute einer Komponente übergeben werden können und auch, welcher Renderer zur Darstellung verwendet wird.

Jedes Tag der Tagbibliothek hat natürlich auch seine eigene Tagklasse. Im Renderer selbst erfolgt die eigentlich Darstellung einer Komponente, wobei bei Bedarf auf ein oder mehrere Modellobjekte zugegriffen werden kann. In diesen Ablauf kann bei Bedarf jederzeit eingegriffen werden, wenn z.B. benutzerspezifische Komponenten erzeugt werden sollen oder ein spezieller Renderer für eine Komponente entwickelt werden soll. Details zur Erweiterung und Anpassung von Komponenten und Renderern finden Sie in Kapitel 9.

6.3.1 Die Basisklasse UIComponent

Die abstrakte Basisklasse für sämtliche User-Interface-Komponenten ist `javax.faces.component.UIComponent`. In dieser Basisklasse ist das gesamte Verhalten einer Komponente definiert, die der JSF-Spezifikation genügt. Gerade auch im Hinblick auf eigens erstellte Komponenten ist die Kenntnis dieser Klasse sehr wichtig. Aufgrund der Tatsache, dass sämtliche Methoden in dieser Klasse als `abstract` gekennzeichnet sind, müssten Komponentenklassen, die davon ableiten, zunächst einmal alle Methoden implementieren. Um dies zu vereinfachen, wird eine so genannte *Convenience-Klasse* `javax.faces.component.UIComponentBase` bereitgestellt. Diese implementiert bereits sämtliche Methoden von `UIComponent`, so dass Klassen, die von `UIComponentBase` ableiten, lediglich die relevanten Methoden überschreiben bzw. implementieren müssen. Daher stammt auch der Begriff *Convenience*, der besagt, dass in der Klasse selbst keine Funtionalität zu finden ist, sondern lediglich ein Komfort bereitgestellt wird, der anderen Entwicklern die Arbeit erleichtert. Im Folgenden wird auf die wichtigsten Methoden dieser Komponentenklasse eingegangen.

Bezeichner von Komponenten

Jede Komponente trägt einen eindeutigen Bezeichner. Innerhalb eines Namensraumes muss dabei jeder Bezeichner eindeutig gewählt werden. Bei der Wahl eines Bezeichners muss ebenfalls darauf geachtet werden, dass

- dieser mit einem Buchstaben oder einem Unterstrich beginnt
- folgende Stellen entweder aus Buchstaben, Zahlen, dem Bindestrich oder dem Unterstrich bestehen.

```
public String getId();
public void setId(String componentId);
```

Die Methode `setId` muss im Normalfall nicht direkt aufgerufen werden. Innerhalb einer Anwendungsentwicklung ist es jedoch oftmals notwendig, den Bezeichner einer Komponente herauszufinden. Dazu kann über `getId` der Bezeichner abgefragt werden.

Wie später in Kapitel 7 noch näher erläutert wird, kann jeder Komponente durch ein zugehöriges Tag eine explizite Id mitgegeben werden. Erfolgt dies nicht manuell, erzeugt JSF eine automatische Id.

Bezeichner auf Clientseite

Der clientseitige Bezeichner wird bei der Codierung und Decodierung direkt in den erzeugten Komponenten verwendet. So entspricht der Bezeichner z. B. in einem Textfeld dem `name`-Attribut, über das z. B. per JavaScript zugegriffen werden kann.

Wird über die Tags kein expliziter Bezeichner angegeben, wird durch das Framework automatisch ein Bezeichner vergeben. Sollten eigene Bezeichner hinterlegt werden, ist unbedingt darauf zu achten, dass diese nicht mehrfach verwendet werden.

Der Bezeichner auf Clientseite resultiert aus dem eigentlichen Komponentenbezeichner sowie aus dem Bezeichner der nächsten Komponente, die einen so genannten `NamingContainer` darstellt. Über

```
public String getClientId( FacesContext context )
```

wird der für die gesamte View eindeutige Bezeichner einer Komponente zurückgeliefert. Dieser setzt sich aus dem eigentlichen Bezeichner der Komponente sowie den übergeordneten Komponenten zusammen. Daher können die Methoden `getId` und `getClientId` den gleichen Rückgabewert liefern, meist unterscheiden sich jedoch beide Bezeichner.

Ist beispielsweise ein Eingabefeld mit dem Bezeichner *firstname* in einem Formular eingebunden, das selbst wiederum den Namen *inputForm* trägt, so lautet die ClientId des Eingabefeldes *inputForm:firstname*. Damit ist die Eindeutigkeit einer Komponente gewährleistet. Bei JavaScript-Routinen, die eventuell in eine Webanwendungen integriert werden, muss daher die Namensvergabe mit berücksichtigt werden.

ValueBinding

Komponenten können ein so genanntes *ValueBinding* zugeordnet haben. Dies bedeutet, dass anstatt der Angabe eines expliziten Werts auf eine Eigenschaft eines so genannten *Backing Beans* zugegriffen wird, von dem der entsprechende Wert bezogen wird. So kann z.B. bei einer Eingabekomponente der Wert als Text oder Zahl explizit hinterlegt werden oder aber über ein ValueBinding an ein Modellobjekt gekoppelt werden. Über das ValueBinding kann dann mittels eines entsprechenden Ausdrucks auf das Objekt selbst zugegriffen werden. Backing Beans werden in der Anwendungskonfigurationsdatei definiert und werden dann durch das Framework automatisch verwaltet und bei Bedarf bereitgestellt.

```
public ValueBinding getValueBinding(String name);
public void setValueBinding(String name, ValueBinding binding);
```

Bei Abfrage eines ValueBindings wird eine Instanz des Klasse `javax.faces.el.ValueBinding` zurückgeliefert. Über den Aufruf von `getValue` kann der konkrete Wert bezogen werden. Beispiele hierzu werden in den folgenden Abschnitten ausführlich demonstriert.

Zugriff auf den Komponentenbaum

Jede erzeugte Seite wird innerhalb des Frameworks in einer Baumstruktur dargestellt. Daher sind alle in einer Seite enthaltenen Komponenten hierarchisch angeordnet. Über

```
public UIComponent getParent();
```

kann auf die übergeordnete Komponente zugegriffen werden. Ebenso kann über

```
public List getChildren();
```

eine Liste aller untergeordneten Komponenten bezogen werden. Diese Funktionen sind gerade im Bezug auf benutzerdefinierte Komponenten und Renderer von großer Bedeutung. Nur wenn die Komponentenhierarchie korrekt interpretiert wird, kann eine erwünschte Ausgabe durch eigene Renderer richtig erzeugt werden.

6.3.2 Die UI-Komponentenklassen

JavaServer Faces definiert ein umfangreiches und erweiterungsfähiges Komponentenmodell. Darin sind die elementaren Komponenten zum Aufbau einer graphischen Oberfläche hinterlegt und spezifiziert. Dabei existieren für jede Komponente Angaben zum aktuellen Status, eine Referenz zum aktuellen Modellobjekt sowie eine Implementierung eines Event-Modells. Die Komponenten können wiederum auf verschiedene Arten dargestellt werden, wofür entsprechende Renderer zuständig sind. So existiert eine 1:N-Beziehung zwischen den Komponenten und den in der `html_basic` Taglib definierten Tags. Allen Komponentenklassen ist gemeinsam, dass sie von der Klasse `javax.faces.component.UIComponentBase` ableiten. Sollten einmal benutzerspezifische Komponenten entwickelt werden, empfiehlt es sich jedoch meist nur in Sonderfällen, eine neue Komponente von `UIComponentBase` erben zu lassen. Meist ist es sinnvoll, bereits von einer vorhandenen Komponente abzuleiten und damit bereits auf einer funktionsfähigen Komponente aufbauen zu können.

In Tabelle 6.3 ist eine Übersicht über die UI-Komponentenklassen dargestellt.

Komponente	Beschreibung
`UIForm`	Eine UIForm-Komponente stellt einen Rahmen für weitere Komponenten bereit. Ein Formular beinhaltet wiederum mehrere Komponenten wie z. B. Eingabe- und Ausgabekomponenten (`UIInput` und `UIOutput`).
`UIColumn`	stellt eine Spalte innerhalb einer Tabellenstruktur dar.
`UICommand`	Ein Kommando löst bei Aktivierung eine Aktion aus.
`UIInput`	Eingabekomponenten dienen der Eingabe von Daten durch den Benutzer.

Tabelle 6.3: Übersicht über alle UI-Komponentenklassen

Komponente	**Beschreibung**
`UIOutput`	Über Ausgabekomponenten können (formatierte) Daten ausgegeben werden.
`UIPanel`	Ein Panel ist eine Komponente für die Darstellung von Daten in Tabellenform.
`UIParameter`	Komponente für zusätzliche Parameter.
`UIGraphic`	Anzeigekomponente für eine Graphik
`UISelectItem`	definiert eine Selektionsmöglichkeit.
`UISelectItems`	definiert eine Sammlung von Auswahlmöglichkeiten.
`UISelectBoolean`	definiert eine Komponente, die genau zwei Zustände (true/false bzw. selektiert/nicht selektiert) darstellen kann.
`UISelectMany`	Komponente zur Auswahl mehrerer Elemente.
`UISelectOne`	Komponente, bei der genau eine Auswahl getroffen werden kann.
`UIMessage`	definiert eine Meldung, die in einer Seite dem Benutzer ausgegeben wird.
`UIMessages`	definiert eine Sammlung mehrerer Meldungen.
`UINamingContainer`	Komponente mit eigenem Namensraum
`UIData`	Komponente für eine Ansammlung mehrerer Daten in Form einer Collection

Tabelle 6.3: Übersicht über alle UI-Komponentenklassen (Fortsetzung)

6.3.3 Das Rendering-Modell

Die aufgezeigte Komponentenarchitektur hat den Vorteil, dass eine Komponente unabhängig von ihrer Darstellung beschrieben werden kann. Es kann Funktionalität unabhängig von einer späteren Darstellung entwickelt werden. Die konkrete Umsetzung z.B. in Html übernehmen so genannte *Renderer*. JavaServer Faces liefert im Standardumfang bereits ein Render-Kit für die Ausgabe in Html aus, weitere Render-Kits können jedoch jederzeit entwickelt werden.

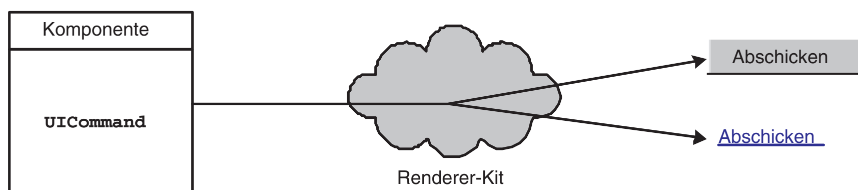


Abbildung 6.2: Das Rendering-Modell

Ein *Render-Kit* beinhaltet mehrere Render-Objekte, von denen jeder Renderer eine Komponente darstellt. Eine Komponente selbst wiederum kann durch verschiedene Renderer dargestellt werden. In Abbildung 6.2 ist die `UICommand`-Komponente einmal

als Commandbutton und einmal als Hyperlink dargestellt. Jede Darstellung beruht auf einem eigenen Renderer, beide Darstellungen verwenden jedoch die gleiche Komponente `UICommand`.

Grundsätzlich werden in JSF zwei Programmiermodelle für das Rendering unterschieden. Bei der direkten Implementierung (*direct implementation*) ist das Rendering in der Komponente selbst bereits enthalten. Bei der verteilten Implementierung (*delegated implementation*) wird für das Rendering eine spezielle Rendererinstanz angezogen. Dieses Modell ist auch das bevorzugte, da damit der Vorteil erreicht wird, dass das Verhalten einer Komponente vollkommen losgelöst von deren späterer Darstellung implementiert ist.

Render-Kits können über die Anwendungskonfigurationsdatei hinterlegt werden. Dabei kann ein entsprechender Bezeichner angegeben werden, der genau ein Render-Kit identifiziert. Ist kein Bezeichner angegeben, wird das Standard-Render-Kit verwendet.

```
<render-kit>
  <render-kit-id>myRenderKit</render-kit-id>
    <renderer>
      <renderer-type>UsageBar</renderer-type>
      <renderer-class>
        com.edu.jsf.bsp.tag.UsageBarRenderer
      </renderer-class>
    </renderer>
    <renderer>
      <renderer-type>CreditCard</renderer-type>
      <renderer-class>
        com.edu.jsf.bsp.tag.CreditCardRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

Listing 6.6: Angabe eines Render-Kit-Bezeichners

Listing 6.6 zeigt einen Ausschnitt aus der Anwendungskonfigurationsdatei, in der zwei Renderer zu einem Render-Kit mit dem Bezeichner *myRenderKit* zusammengefasst sind. In diesem Fall handelt es sich um zwei benutzerdefinierte Renderer. Auf die genaue Bedeutung der Eintragungen und wie benutzerdefinierte Renderer und Komponenten überhaupt erzeugt werden, wird in Kapitel 9 eingegangen.

Die Abfrage, welche Render-Kit-Id aktuell gesetzt ist, kann im Programmablauf über

```
FacesContext context = FacesContext.getCurrentInstance();
context.getViewRoot().getRenderKitId();
```

erfolgen. Ebenso kann auch zur Laufzeit ein neues Render-Kit mittels

```
FacesContext context = FacesContext.getCurrentInstance();
context.getViewRoot().setRenderKitId("myRenderKit");
```

gesetzt werden.

6.4 Bean Management

Ein zentraler und wichtiger Bestandteil einer jeder Webanwendung ist die Verwaltung von applikationsspezifischen Beans. Dabei dienen JavaBeans meist zur Speicherung von Programm- und Benutzerdaten. Daten aus Eingabeformularen werden dabei üblicherweise in JavaBeans übertragen, die wiederum zur weiteren Verarbeitung z.B. an entfernte Servicemethoden weitergereicht werden. Bezugnehmend auf das *Model-View-Controller* Designpattern (vgl. dazu Kapitel 2.5 Model-View-Controller-Pattern) wird mit Hilfe von JavaBeans das *Modell* abgebildet. Daher spricht man in diesem Zusammenhang auch von *Modellobjekten*.

Ein Modellobjekt dient auch in JSF-Anwendungen der Speicherung der Programm- und Benutzerdaten. Da das Modellobjekt als JavaBean den bekannten Bean-Spezifikationen unterliegt, stellt das Erzeugen eines Modellobjektes keine allzu große technische Neuerung dar. Ein Beispiel für ein einfaches Modellobjekt wurde bereits in Kapitel 5 gezeigt.

JavaBeans

Ein JavaBean ist zunächst einmal ein Objekt, das eine eigenständige und wiederverwendbare Komponente darstellt. Es kapselt dabei Eigenschaften (»Variablen«), die über öffentliche Zugriffsmethoden (die so genannten *getter*- und *setter*-Methoden) gesetzt und abgefragt werden können. Oftmals ist zu lesen, dass JavaBeans in einer visuellen Entwicklungsumgebung bearbeitet werden können. Dies trifft jedoch nur auf Beans zu, die eine Oberflächenkomponente darstellen. Fasst man den Begriff des JavaBeans etwas weiter, sind damit auch Komponenten gemeint, die nicht unmittelbar für graphische Darstellung verwendet werden, sondern einfach gewisse Daten in ihrem Inneren kapseln und über definierte Zugriffe eine Bearbeitung zulassen.

6.4.1 Managed-Bean-Konzept

Ein Unterschied zur klassischen Verwendung von JavaBeans zu den Beans in JSF ist zunächst einmal in der Einbindung in eine Webseite zu finden. Klassisch werden Beans mittels der `<jsp:useBean>`-Syntax in jeder einzelnen JSP-Seite deklariert und eingebunden. Dieses Verfahren birgt jedoch gewisse Fehleranfälligkeiten in sich. So kann es

unter Umständen passieren, dass ein Benutzer über einen anderen Navigationspfad auf eine bestimmte Seite gelangt, wie vom Entwickler anfänglich vorgesehen. Dort wird dann auf ein Bean zugegriffen, das jedoch in diesem Falle noch nicht erzeugt wurde. Es kommt in diesem Fall zu einem Programmfehler. Sicherlich kann man durch eine entsprechende Programmierung und auch durch einen umfangreichen Programmtest solche Fehler vermeiden, die Vorgehensweise an sich ist jedoch immer etwas gefährlich. Jedoch ist dieses Verfahren aufwändig, da in jeder Seite die entsprechende Anweisung enthalten sein muss.

In JSF wird deswegen das Prinzip der so genannten *Managed-Beans* verwendet. Dies besagt, dass die Erzeugung der Modellobjekte (also der Beans) auf deklarativer Ebene erfolgt. Die Modell-Objekte, die in einer JSF-Applikation verwendet werden, werden somit in der Anwendungskonfigurationsdatei deklariert und damit der Kontrolle des JSF-Frameworks übergeben. Wie der Begriff *managed* schon andeutet, wird es durch das Framework somit gesteuert, dass die Beans zur rechten Zeit vorhanden sind. Ein Entwickler muss sich somit nicht mehr explizit vor jeder Verwendung eines Beans damit beschäftigen, dass dieses auf der aktuellen Seite auch vorhanden ist.

Managed-Bean versus Modellobjekt versus JavaBean

In den Erläuterungen wird immer wieder der Begriff *Managed-Bean*, *Modellobjekt* bzw. *JavaBean* verwendet. Ein JavaBean ist prinzipiell erst einmal ein Objekt gemäß den Bean-Spezifikationen von Sun. JavaBeans werden in unterschiedlichsten Anwendungsbereichen eingesetzt. Der Begriff Modellobjekt ist zunächst einmal auch JSF-unabhängig. Im Umfeld von JSF jedoch stellen Modellobjekte Daten aus dem »Business Model« dar. Modellobjekte können als Managed Beans automatisch erzeugt werden und somit für JSF-Komponenten jederzeit zugreifbar gemacht werden. Des Weiteren kann ein Modellobjekt bereits Funktionalität beinhalten, um gewisse Aktionen auszuführen. So können in einem Modellobjekt bereits Validierungsmethoden vorhanden sein sowie Aktionsmethoden, die durch bestimmte Aktionen von UI-Komponenten ausgelöst werden.

Die Anwendungskonfigurationsdatei wird beim Laden der Webanwendung eingelesen. Dies ist meist der Fall, wenn der Applikationsserver neu gestartet wird. Nach dem Einlesen der Anwendungskonfigurationsdatei stehen die darin hinterlegten Managed-Beans *automatisch* auf jeder JSF-Seite zur Verfügung, ohne dass sie explizit deklariert oder eingebunden werden müssen. Eine Einbindung dieser Beans durch `ein <jsp:useBean>`-Tag oder Ähnliches entfällt damit.

Verwendung von Managed-Beans

Bei Programmstart, d.h. im Normalfall beim Start des Applikationsservers, wird die Anwendungskonfigurationsdatei *faces-config.xml* eingelesen und ausgewertet. Um eine einfache Bean-Klasse als Managed-Bean der Anwendung bereitzustellen, muss eine

Bean-Klasse in der Anwendungskonfigurationsdatei hinterlegt werden. Nach herkömmlicher JSP-Programmierung wurde ein `<jsp:useBean>`-Tag in jede Seite eingebaut, von der aus auf das Bean zugegriffen wurde.

```
<jsp:useBean id="Person" class="com.edu.jsf.bsp.bean.PersonBean"
    scope="session" />
```

Listing 6.7: Konventionelle Einbindung eines Beans in eine JSP-Seite

Eine häufig auftauchende Fehlerquelle bei klassischer JSP-Programmierung ist die Initialisierung des *Scope*, also des Gültigkeitsbereichs von Beans, auf unterschiedlichen Seiten mit unterschiedlichen Werten. Auch aus diesem Grund ist die deklarative Einbindung der Beans über die Anwendungskonfigurationsdatei ein großer Vorteil.

```
...
<managed-bean>
  <managed-bean-name>Person</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.PersonBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>city</property-name>
    <value>Stuttgart</value>
  </managed-property>
</managed-bean>
...
```

Listing 6.8: Einbindung eines Beans in der Anwendungskonfigurationsdatei in JSF

Dieses Beispiel deklariert ein Managed-Bean `Person`, das im Gültigkeitsbereich `Session` von der Anwendung jederzeit angesprochen werden kann. Folgende Tags werden für eine Deklaration verwendet:

- `<managed-bean-name>`: Der Name, über den das Bean in der Anwendung jederzeit angesprochen werden kann. Der Name muss eindeutig sein und darf keine Leerzeichen oder den Bindestrich enthalten.
- `<managed-bean-class>`: Der Klassenname des Beans. Dabei muss der Klassenname vollqualifizierend angegeben werden. D.h. wenn die Klasse `PersonBean` in einem Package `com.firma.beans` liegt, muss bei `<managed-bean-class>` der vollständige Packagename samt Klassenname angegeben werden.
- `<managed-bean-scope>`: Hierüber wird analog zur reinen JSP-Programmierung der Gültigkeitsbereich des Beans festgelegt. Folgende Werte sind dabei zugelassen:

- ▶ `application`: Das Bean ist während der gesamten Laufzeit der Anwendung gültig und wird erst mit dem Stoppen des Applikationservers bzw. mit dem Entladen der Webanwendung ungültig. Dieses Verhalten bedingt natürlich, dass Beans dieses Gültigkeitsbereichs globalen Charakter haben und nicht mehr benutzer- oder sessionspezifisch unterschiedlich gehandhabt werden.
- ▶ `session`: Dies ist wohl die häufigste Variante, wie Beans in die Anwendung eingebaut werden. Während der Arbeitssitzung, also während der Gültigkeitsdauer einer Session, bleibt das Bean verfügbar und gültig. Erst mit dem Löschen der Session verliert auch das Bean seine Gültigkeit. Dies kann beispielsweise dann passieren, wenn explizit ein Logout-Mechanismus aufgerufen wird und damit die Session auf »invalide« gesetzt wird. Es kommt aber auch dann vor, wenn der Browser einfach geschlossen wird und somit die Sessionverwaltung im Applikationsserver nach dem Timeout die Session auf invalide setzt.
- ▶ `request`: Der Gültigkeitsbereich *request* ist der kleinstmögliche Bereich, in dem Beans gespeichert werden können. Sie sind in diesem Falle nur während der Requestverarbeitung im Server vorhanden und werden danach sofort wieder entfernt.
- ▶ `none`: Natürlich ist es ebenfalls möglich, Beans überhaupt nicht dauerhaft zu speichern, sondern nur für die Berechnung eines Ergebnisses kurzfristig zu verwenden und danach sofort wieder aus dem Speicher zu entfernen.

▶ `<managed-bean-property>`: Ein `managed-bean`-Element kann mehrere `managed-bean-property`-Elemente beinhalten. Diese können Eigenschaften des Beans gleich beim Erzeugen einer Instanz festlegen. Mittels `property-name` wird der Name der Eigenschaft angesprochen und danach mittels `value` ein entsprechender String übergeben. Es liegt natürlich in der Verantwortung des Entwicklers, hier nur Bean-Eigenschaften zu verwenden, die in der Bean-Klasse auch mit entsprechenden getter- und setter-Methoden vorhanden sind. Auf das Initialisieren selbst wird in den folgenden Abschnitten näher eingegangen, da hierbei einige Besonderheiten zu beachten sind.

Vorteile des Managed-Bean-Konzepts

Neben dem bereits erwähnten Vorteil einer deklarativen Einbindung von Beans in die Anwendung bietet das Verfahren weitere Vorteile gegenüber der klassischen Einbindung mittels `<jsp:useBean>`-Tags:

▶ Deklaration und Erzeugung der Beans an einer zentralen Stelle in der Applikation. Objekte werden nicht mehr (wahllos) an verteilten Stellen im Programmablauf erzeugt, sondern entsprechend ihres Gültigkeitsbereichs automatisiert verwaltet. Dies erhöht einerseits die Übersichtlichkeit über die zu erzeugenden Beans, andererseits reduziert dies zudem die Fehleranfälligkeit im Programmablauf.

- Die Bean-Eigenschaften können direkt über die Konfigurationsdatei beeinflusst und verändert werden, ohne dass ein Eingriff in die Programmsourcen notwendig ist.
- Mittels des `value`-Elements können auch Abhängigkeiten mehrerer Beans realisiert werden. So kann der Wert eines Bean-Attributs das Ergebnis einer Rechnung innerhalb eines anderen Beans sein.

Einschränkungen bei Verwendung von Managed-Beans

Neben den Vorteilen, die eine Verwendung des Managed-Bean-Konzepts mit sich bringt, existieren jedoch auch einige Nachteile bzw. Einschränkungen, auf die an dieser Stelle hingewiesen werden soll:

Managed-Beans werden nicht auf Vorrat erzeugt, sondern bei der ersten Verwendung in einer JSF-Seite kreiert (diese Vorgehensweise wird *lazy loading* genannt). Dies hat zur Folge, dass es zu Fehlern kommen kann, wenn über eine andere Zugriffsmöglichkeit als über JSF auf ein Managed-Bean zugegriffen werden soll. So bietet die JSTL verschiedene Möglichkeiten, um beispielsweise Daten auszugeben, über eine Liste zu iterieren oder Variablen zu setzen.

Wird aber über die JSTL auf ein Managed-Bean zugegriffen, wird die automatische Erzeugung von Beans quasi umgangen, das Bean ist nicht erzeugt und ist daher im Kontext auch nicht zu finden. Erst nachdem ein Managed-Bean mindestens einmal über eine JSF-Funktion aufgerufen wurde, kann man sicher sein, dass es korrekt erzeugt wurde und vorhanden ist. Es gibt zwar eine Möglichkeit, die Bean-Erzeugung »von Hand« zu starten, so dass im Nachhinein über die JSTL darauf zugegriffen werden kann, dies ist jedoch nur bedingt zu empfehlen.

```
ApplicationFactory factory =
  (ApplicationFactory) FactoryFinder.getFactory(
FactoryFinder.APPLICATION_FACTORY);
Application application = factory.getApplication();
application.createValueBinding("MeinBean");
```

Listing 6.9: Zugriff auf ein Managed Bean über die Factory

Mit dem Aufruf aus Listing 6.9 wird über eine `ApplicationFactory` ein Bean angefordert. Sollte es noch nicht existent sein, wird es automatisch angelegt. Danach kann über andere Zugriffswege beispielsweise über die JSTL auf das Bean zugegriffen werden.

Lazy loading

Das Konzept des *lazy loadings* wird häufig dazu verwendet, um Systemressourcen beispielsweise beim Programmstart zu sparen. Hinsichtlich JSF werden Managed-Beans in der Anwendungskonfigurationsdatei hinterlegt. Dies kann bei umfangreicheren Anwendungen durchaus eine sehr große Anzahl an Objekten sein, die erzeugt und gegebenenfalls in der Session abgespeichert werden müssen. Um hier einerseits die Startzeit nicht über die Maßen zu erhöhen, andererseits auch um Ressourcen zu schonen, werden die Objekte erst bei ihrer allerersten Verwendung erzeugt, anstatt sämtliche Objekte gleich beim Start des Applikationsservers zu instanzieren und in der Session abzulegen.

6.4.2 Backing Beans

Backing Beans halten in Faces-Anwendungen eine Verbindung zu Oberflächenkomponenten. So tauchen in einer typischen Faces-Anwendung meist mehrere Backing Beans auf, die als serverseitige Objekte Eigenschaften von UI-Komponenten vorhalten. Oft beinhalten Backing Beans noch weitere Funktionalitäten wie Validierungsmethoden oder Aktionsmethoden. Backing Beans werden durch das Konzept der Managed-Beans bereitgestellt und erzeugt.

Es ist somit falsch, Backing Beans im Gegensatz zu Managed-Beans zu sehen. Managed Beans ist vielmehr ein Verfahren als eine Klassifikation. Etwas kontrovers wird häufig auch diskutiert, wie es sich mit der Wiederverwendbarkeit von Backing Beans verhält. Da diese wie bereits erwähnt bestimmte Aktionsmethoden beinhalten können, die aufgrund eines Oberflächenereignisses ausgelöst werden, sind die Beans meistens sehr anwendungs- bzw. auch JSF-spezifisch. Damit kann ein Backing Bean nicht mehr ohne weiteres in einem komplett andersartigen Kontext wiederverwendet werden. Diese Konsequenz geht man jedoch bewusst ein. Aus Architektursicht sollte daher darauf geachtet werden, eine saubere Trennung zwischen den Backing Beans und Methoden bzw. Klassen der Geschäftslogik-Schicht zu erhalten. Somit kann zumindest die Schicht für die Geschäftslogik auch in einem anderen Kontext wiederverwendet werden.

Verwendung von Backing Beans

Als Beispiel dient folgendes Modell-Objekt, das auch in den folgenden Abschnitten schrittweise erweitert und als Grundlage für die Kurzbeispiele verwendet wird:

```
package com.edu.jsf.bsp.bean;

import java.util.ArrayList;
import javax.faces.component.SelectItem;
```

```
/**
 * Klasse für ein PersonBean
 */
public class PersonBean {

  private String firstname;
  private String lastname;
  private String street;
  private String city;

  /**
   * Konstruktor
   */
  public PersonBean() {
  }

  /**
   * Getter-Methoden
   */
  public String getFirstname() {
    return firstname;
  }

  public String getLastname() {
    return lastname;
  }

  public String getStreet() {
    return street;
  }

  public String getCity() {
    return city;
  }

  /**
   * Setter-Methoden
   */
  public void setFirstname(String string) {
    firstname = string;
  }

  public void setLastname(String string) {
    lastname = string;
  }

  public void setStreet(String string) {
    street = string;
  }
```

```
    public void setCity(String string) {
      city = string;
    }
  }
```

Listing 6.10: Ein einfaches Modellobjekt

Um dieses Modellobjekt in der Anwendungskonfigurationsdatei zu hinterlegen, sind folgende Eintragungen in der *faces-config.xml* notwendig:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

  ...
  <managed-bean>
    <managed-bean-name>Person</managed-bean-name>
    <managed-bean-class>
        com.edu.jsf.bsp.bean.PersonBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
  ...

</faces-config>
```

Listing 6.11: Auszug aus der faces-config.xml

Aus diesem Auszug der *faces-config.xml* ist zu ersehen, dass ein Modell-Objekt mit Namen `Person` im Gültigkeitsbereich `Session` als *Managed Bean* erzeugt wird. Das Bean entstammt der Klasse `com.edu.jsf.bsp.bean.PersonBean`. Allein durch diese Eintragung kann in den einzelnen Seiten auf das Objekt zugegriffen werden. Eine Ausgabe des Vornamens an einer beliebigen Stelle in einer JSF-Seite kann wie folgt geschehen:

```
<h:outputText value="#{Person.firstname}" />
```

Dadurch, dass in der Anwendungskonfigurationsdatei der Bezeichner `Person` hinterlegt wurde, weiß das Framework, dass auf ein Objekt der Klasse `PersonBean` zugegriffen werden soll. Das Objekt muss in diesem Fall im Session-Kontext zu finden sein. Ist es dort nicht vorhanden, wird es automatisch erzeugt und dort abgelegt.

6.4.3 Initialisierungsparameter

Oftmals ist es erforderlich, Modellobjekte bereits mit einem Standardwert vorzubelegen. Dies erhöht die Anwenderfreundlichkeit einer Anwendung und vereinfacht einem Benutzer das Ausfüllen von Formularen. So ist es z.B. denkbar, dass bei einem Anmeldeformular für eine örtliche Veranstaltung die Angabe der Stadt bereits vorbelegt ist, da fast ausschließlich Besucher aus einer bestimmten Stadt sich hierüber anmelden. Natürlich ist die Angabe eines Standardwerts auch in dieser Form realisierbar, indem eine Eigenschaft eines Beans im Konstruktor bereits einen Wert zugewiesen bekommt. Diese Vorgehensweise ist jedoch nicht sehr empfehlenswert, da Änderungen an den Vorbelegungen immer mit Änderungen im Quellcode einer Anwendung verbunden sind. Vorteilhafter ist es, Initialisierungsparameter auf deklarativer Ebene zu hinterlegen. In JSF kann dies über die Anwendungskonfigurationsdatei *faces-config.xml* erfolgen.

Initialisierungsparameter vom Typ String

Initialisierungsparameter vom Datentyp String sind sicherlich die einfachsten Parameter, da hierbei keine Datenkonvertierung vorgenommen werden muss. Die komplette Zeichenkette, die zwischen den beiden `value`-Tags steht, wird komplett als String-Parameter an das Modellobjekt übergeben (durch den entsprechenden Aufruf der setter-Methode). Innerhalb eines `managed-bean`-Tags kann es mehrere `managed-bean-property`-Tags geben. Dies verdeutlicht auch der Auszug aus der dazugehörigen DTD:

```
<!ELEMENT managed-bean (description*, display-name*, icon*, managed-bean-name,
managed-bean-class, managed-bean-scope, managed-property*)>
```

Listing 6.12: Auszug aus der web-facesconfig_1_0.dtd

Für das Modellobjekt aus Listing 6.10 soll exemplarisch die Angabe der Stadt mit dem Wert `München` vorbelegt werden.

```
<managed-bean>
  <managed-bean-name>Person</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.PersonBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>city</property-name>
    <value>München</value>
  </managed-property>
</managed-bean>
```

Listing 6.13: Initialisierungsparameter für einen String-Wert

Es ist bei der Angabe einer Eigenschaft zu beachten, dass die Schreibweise exakt mit der Variablen übereinstimmen muss (inklusive der Groß- und Kleinschreibung).

Zahlen als Initialisierungsparameter

Werden Zahlen in der Anwendungskonfigurationsdatei im Bereich `<managed-property>` angegeben, so muss bei der Zuweisung eine entsprechende Typumwandlung (eine Umwandlung des String in einen Zahlendatentyp) erfolgen. Diese Typumwandlung wird automatisch vorgenommen und muss nicht explizit erfolgen.

Als Beispiel wird das `PersonBean` um das Attribut *Postleitzahl* erweitert, das ein `int`-Datentyp ist.

```
package com.edu.jsf.bsp.bean;

/*
 * Klasse für ein PersonBean
 */
public class PersonBean {

  ...
  private int zip;
  ...

  /*
   * liefert die Postleitzahl zurück
   */
  public int getZip() {
    return zip;
  }

  /*
   * setzt eine neue Postleitzahl
   */
  public void setZip(int i) {
    zip = i;
  }

  ...
}
```

Listing 6.14: Erweitertes Personen-Bean

Für das erweiterte `PersonBean` aus Listing 6.14 wird die Postleitzahl mit `80000` vorbelegt. Dazu genügt es, analog zur Vorbelegung der Stadt den Zahlenwert `80000` in der Anwendungskonfigurationsdatei zu hinterlegen. Die Datenkonvertierung geschieht dabei automatisch.

```
<managed-bean>
  <managed-bean-name>Person</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.PersonBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>

  ...

  <managed-property>
    <property-name>zip</property-name>
    <value>80000</value>
  </managed-property>
</managed-bean>
```

Listing 6.15: Ein Initialisierungsparameter für den int-Datentyp

Wie aus Listing 6.15 zu ersehen ist, muss für einen `int`-Datentyp keine spezielle Konvertierung angegeben werden, um den Wert *80000* als Initialisierungsparameter dem Modellobjekt mitzugeben.

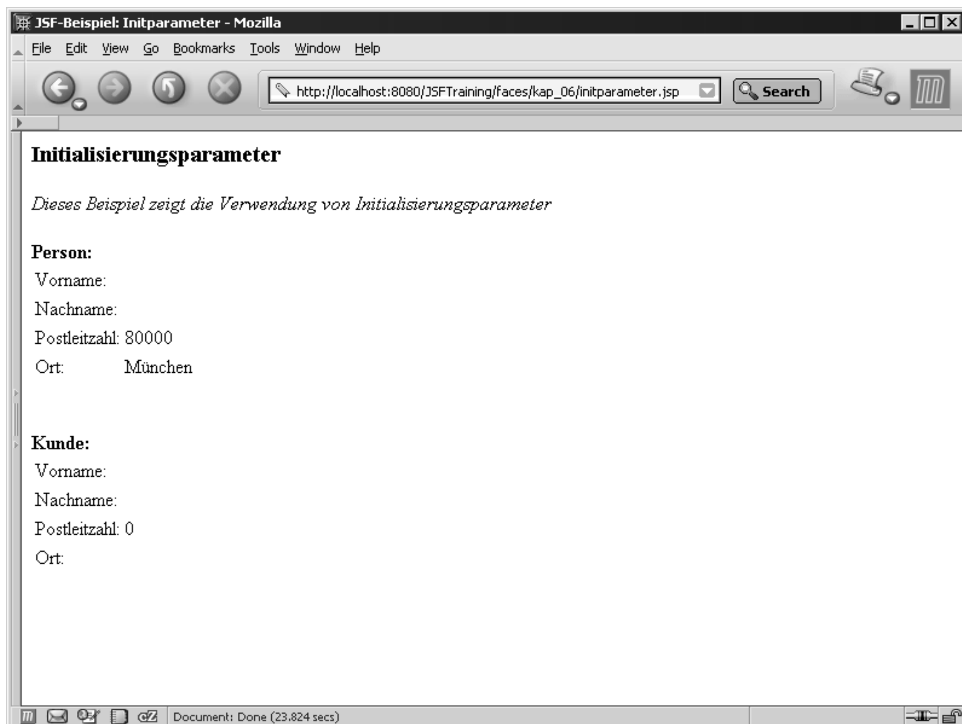


Abbildung 6.3: Ausgabe der Postleitzahl als int-Datentyp

In einem anderen Beispiel wäre es eventuell gewollt gewesen, eine Zahl in einer formatierten Darstellung auszugeben, beispielsweise als Währung oder mit Tausenderpunkten. Dazu kann im `outputText`-Tag ein spezielles Zahlenformat angegeben werden. Auf diese Thematik wird in Abschnitt 6.7 näher eingegangen.

Initialisierungsparameter vom Typ Boolean

Soll per Anwendungskonfigurationsdatei eine Eigenschaft eines Beans (bzw. eines Modellobjekts) mit einem Wahrheitswert (Boolean-Typ) vorbelegt werden, so kann dies erfolgen, indem die Werte `true` oder `false` direkt als Wort eingesetzt werden. Um dies anhand eines Beispiels zu verdeutlichen, wird dazu nochmals die Klasse `PersonBean` um die Eigenschaft `male erweitert`, mit der abgefragt wird, ob eine Person männlichen Geschlechts ist oder nicht (und damit folgerichtig weiblich ist):

```
private boolean male;
```

Natürlich gibt es auch dazu wieder die entsprechenden Zugriffsmethoden:

```
public boolean isMale() {
  return isMale;
}

public void setMale(boolean b) {
  isMale = b;
}
```

In der Anwendungskonfigurationsdatei wird im Falle einer Vorbelegung für `male=true` folgende Eintragung vorgenommen:

```
<managed-bean>
  ...
  <managed-property>
    <property-name>male</property-name>
    <value>true</value>
  </managed-property>
</managed-bean>
```

Listing 6.16: Beispiel für eine Initialisierung mit einem Boolean-Datentyp

Die Groß- und Kleinschreibung für die Werte `true` oder `false` ist dabei unerheblich. Alternativ können auch die Zahlen 0 (für *false*) sowie 1 (für *true*) verwendet werden.

Initialisierungsparameter vom Typ Map

Die Initialisierungsroutine über die Anwendungskonfigurationsdatei ist nicht nur auf einfache Datentypen beschränkt. Es können auch komplexe Datentypen wie eine `Map` (`java.util.Map`) per Datei initialisiert werden. Statt jedoch ein einfaches `value`-Tag zu verwenden, ist der Aufbau der Wertematrix in der DTD wie folgt vorgegeben:

```
<!ELEMENT map-entries (key-class?, value-class?, map-entry*) >
```

Die Attribute `key-class` sowie `value-class` sind dabei optional, können jedoch maximal einmal auftreten, Map-Entrys können überhaupt nicht oder aber mehrfach vorhanden sein.

Die Attribute key-class und value-class

Standardmäßig werden sowohl die Schlüssel-Werte als auch die Werte selbst als String in der Map abgelegt. Ist dieses Verhalten nicht gewollt, kann mittels `key-class` das Standardverhalten für den Schlüsselwert überschrieben werden bzw. mittels `value-class` das Standardverhalten für die Werteeinträge. Es wird damit ein Datentyp angegeben, in den die Werte konvertiert und in der Map abgelegt werden.

Achtung:

Da in einer Map nur Objekte abgespeichert werden können, kann eine *key-class* niemals ein primitiver Datentyp wie `int`, `float`, `double` etc. sein. Als key-class-Werte sind nur Klassen erlaubt mit ihrem vollqualifizierenden Namen. Z.B. `java.math.BigDecimal` oder `java.math.Integer`.

Map-Entry

Das Map-Entry-Element ist in der DTD wie folgt beschrieben:

```
<!ELEMENT map-entry (key, (null-value|value>
```

Beide Attribute sind Pflicht, es muss für jeden Eintrag sowohl der Schlüssel (*key*) als auch der dazugehörige Wert angegeben werden. Der Wert selbst kann dabei ein Null-Wert sein oder ein konkreter Wert (*value*). Innerhalb des `value`-Attributes ist es natürlich auch möglich, mit einem `ValueBinding`-Ausdruck zu arbeiten. Damit kann innerhalb der Anwendungskonfigurationsdatei auf andere Managed-Beans referenziert werden.

```
<managed-bean>
  <managed-property>
    <property-name>umrechnungskurse</property-name>
    <map-entries>
      <value-class>java.math.BigDecimal</value-class>
      <map-entry>
        <key>dollar</key>
        <value>1.12</value>
      </map-entry>
      <map-entry>
        <key>Yen</key>
        <value>#{Kurse.yen}</value>
```

```
      </map-entry>
    </map-entries>
  </managed-property>
</managed-bean>
```

Listing 6.17: Beispiel einer Initialisierung einer Map

Im Listing 6.17 wird eine Map mit dem Namen *umrechnungskurse* mit zwei Werten befüllt. Ein Wert – der Dollar-Kurs – wird direkt mittels des `value`-Tags eingegeben. Es findet eine Datenkonvertierung in einen `BigDecimal` statt, da durch Angabe des `value-class`-Tags die Werte der Map als `BigDecimal` abgespeichert werden. Das zweite Element der Map wird nicht direkt mit einem Wert befüllt, sondern es ist eine Referenz auf ein Bean `Kurse` und dort auf die Eigenschaft `yen` angegeben. Existiert ein auf diese Weise referenziertes Bean zu diesem Zeitpunkt noch nicht, wird es automatisch angelegt.

Initialisierungsparameter vom Typ Array oder List

Im Gegensatz zum Map-Datentyp wird bei einem Array oder einer List ausschließlich der Wert gespeichert, eine Key-Zuordnung existiert hier nicht. Für die Initialisierung zunächst auch hier wieder der Auszug aus der DTD:

```
<!ELEMENT list-entries (value-class?, (null-value|value)*)>
```

Im Gegensatz zur Initialisierung eines einzelnen Elements, das mit Hilfe des Tags `value` funktioniert, kommt bei einer List oder einem Array das Element `list-entries` zum Einsatz. Optional kann hier ein Datentyp angegeben werden, in den ein einzelner Wert konvertiert wird, bevor er einem Array oder einer List hinzugefügt wird (Element `value-class`). Erfolgt keine explizite Angabe einer Klasse, wird auch hier wieder der Datentyp `String` als Standard verwendet.

```
<managed-bean>
  ...
  <managed-property>
    <property-name>farben</property-name>
    <list-entries>
      <value>rot</value>
      <value>grün</value>
      <value>blau</value>
      <value>gelb</value>
      <null-value/>
    </list-entries>
  </managed-property>
</managed-bean>
```

Auch an dieser Stellen kann im `value`-Attribut wiederum mittels eines `ValueBinding`-Ausdrucks auf eine Eigenschaft eines anderen Beans verwiesen werden.

Initialisierungsparameter über den Deployment Deskriptor

Anstatt Werte für ein Managed-Bean direkt in der Anwendungskonfigurationsdatei *faces-config.xml* zu hinterlegen, bietet JSF auch die Möglichkeit, auf Eintragungen im Deployment-Deskriptor (*web.xml*) der Anwendung selbst zu referenzieren. Damit ist eine weitere komfortable Möglichkeit gegeben, an zentraler Stelle Initialisierungsparameter der Anwendung mit zu übergeben.

```
<managed-bean>
  <managed-bean-name>Person</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.PersonBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>city</property-name>
    <value>Stuttgart</value>
    </managed-property>
  <managed-property>
    <property-name>zip</property-name>
    <value>#{initParam.defaultZip}</value>
  </managed-property>
</managed-bean>
```

Listing 6.18: Auszug aus der faces-config.xm

In Listing 6.18 wird bei der Angabe der Postleitzahl (`zip`) auf einen Initialisierungsparameter `defaultZip` referenziert, der im Deployment-Deskriptor hinterlegt wurde. Ein Auszug aus der entsprechenden *web.xml* verdeutlicht dies:

```
<context-param>
  <param-name>defaultZip</param-name>
  <param-value>70563</param-value>
</context-param>
```

Listing 6.19: Auszug aus der web.xml

Der Parameter `defaultZip` wurde – wie in Listing 6.19 dargestellt – als Kontextparameter hinterlegt. Über diesen kann wiederum mittels der `value`-Angabe in der *faces-config.xml* zugegriffen werden. Der Bezeichner `initParam` gehört zu den impliziten Objekten. Diese sind automatisch in JSF-Anwendungen vorhanden und können bei Bedarf jederzeit angesprochen werden.

Initialisierungsparameter über Referenzen

Wenn es möglich ist, auf Kontextparameter des Deployment-Deskriptors zuzugreifen, ist es natürlich ebenfalls möglich, in einer Managed-Bean-Deklaration auf Initialisierungsparameter anderer Managed-Beans zu referenzieren. Damit ist es möglich, in

einer Art Baumstruktur verschiedene Initialisierungsparameter über mehrere Managed-Beans zu verwenden. Wichtig ist jedoch, dass Zirkelbezüge nicht möglich sind und zu einem Fehler führen.

```
<managed-bean>
  <managed-bean-name>Customer</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.CustomerBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>city</property-name>
    <value>#{Person.city}</value>
  </managed-property>
</managed-bean>
```

Listing 6.20: Auszug aus der faces-config.xml

In Listing 6.20 wird ein Managed-Bean `Customer` deklariert. Die Angabe der Stadt (`city`) wird dabei mit einem Standardwert vorbelegt. Dieser wird nicht direkt mitgegeben, sondern wird über ein anderes Managed-Bean (`Person`) referenziert.

6.4.4 ValueBinding und MethodBinding

Der Vorteil von Backing Beans liegt wie bereits geschildert in der sehr einfachen Verwendung innerhalb einzelner JSF-Seiten. Für Ein- oder Ausgabekomponenten kann dadurch über einen so genannten *ValueBinding*-Ausdruck auf die Werte zugegriffen werden.

```
Vorname: <h:outputText value="#{Person.firstname}" />
Nachname: <h:outputText value="#{Person.lastname}" />
```

Listing 6.21: Zugriff auf Werte eines Backing Beans

Damit JSF erkennen kann, dass es sich innerhalb des `value`-Attributes nicht um eine Textausgabe, sondern um einen ValueBinding-Ausdruck handelt, wird die Syntax `#{Ausdruck}` verwendet. Dieser Ausdruck der *JavaServer Faces Expression Language (JSF EL)* bewirkt, dass der Wert einer Komponente an eine Eigenschaft eines Backing Beans gebunden ist. Weitere Beispiele für gültige Ausdrücke sind auch:

- #{Person.partner.nachname}
- #{Person.kinder[2].nachname}
- #[Personen[1].nachname}

Im Beispiel `#{Person.partner.nachname}` wird davon ausgegangen, dass das zugrunde liegende Bean eine Methode `getPartner` hat, die ein Objekt zurückliefert, das wiederum die Methode `getNachname` aufweist.

Bei der Syntax `#{Person.kinder[2].nachname}` wird davon ausgegangen, dass bei einer Person eine Liste von Kindern zurückgeliefert wird, wovon explizit auf das dritte Element zugegriffen werden soll (die Zählung beginnt bei 0).

Zugriff aus Methoden heraus

Während der Zugriff auf Backing Beans in JSF-Seiten sehr komfortabel erfolgt, ist für den Zugriff z.B. in `Listener`-Klassen einiges Wissen über den `FacesContext` erforderlich. Im Vorgriff auf eine detaillierte Beschreibung der Verwendung von Backing Beans in UI-Komponenten ist in Listing 6.22 zu sehen, wie Werte aus einem Backing Bean `Person` in einer JSF-Seite ausgegeben werden.

```
<h:form id="myForm">
  Vorname: <h:inputText value="#{Person.firstname}" /><br>
  Nachname: <h:inputText value="#{Person.lastname}" /><br>
  Straße: <h:inputText value="#{Person.street}" /><br>
  Plz / Ort: <h:inputText value="#{Person.zip}" />
  <h:inputText value="#{Person.city}" /><br>

  <h:commandButton value="Vorbelegen">
    <f:actionListener
        type="com.edu.jsf.bsp.listener.PreselectListener" />
  </h:commandButton>
</h:form>
```

Listing 6.22: Zugriff auf ein Modellobjekt

Im Beispiel in Listing 6.22 ist eine Eingabemaske für Personendaten aufgebaut. Durch Betätigen eines Buttons sollen bestimmte Werte vorbelegt werden. Dazu wurde hier exemplarisch ein *Listener* an den Button angehängt, wobei das Prinzip der Listener nochmals in einem späteren Kapitel ausführlich erläutert wird.

```
package com.edu.jsf.bsp.listener;

import javax.faces.FactoryFinder;
import javax.faces.application.Application;
import javax.faces.application.ApplicationFactory;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;
import javax.faces.event.PhaseId;
```

```
/**
 * Implementierung eines ActionListeners
 */
public class PreselectListener implements ActionListener {

  /**
   * wird bei Auslösen eines Events aufgerufen
   */
  public void processAction( ActionEvent event )
      throws AbortProcessingException {
    FacesContext context = FacesContext.getCurrentInstance();
    ApplicationFactory factory =
      (ApplicationFactory) FactoryFinder.getFactory(
        FactoryFinder.APPLICATION_FACTORY);
    Application application = factory.getApplication();

    ValueBinding binding =
          application.createValueBinding( "#{Person.city}" );
    binding.setValue( context, "Stuttgart" );
  }

}
```

Listing 6.23: Zugriff auf Modellobjekte in Java-Klassen

Listing 6.23 zeigt die Implementierung der Listenerklasse, die bei Auslösen des Commandbuttons aus Listing 6.22 verwendet wird. Zunächst wird eine Instanz des `FacesContext` angezogen. Dies erfolgt durch Aufruf der statischen Methode

```
FacesContext.getCurrentInstance()
```

Des Weiteren wird die so genannte `ApplicationFactory` ebenfalls wieder über einen statischen Aufruf angezogen. Die `ApplicationFactory` ist eine Factory-Klasse (nach den üblichen Bezeichnungen der Design Patterns) und liefert als Ergebnis ein Objekt der Klasse `Application` zurück. Darin können JSF-basierende Webanwendungen anwendungsweite *Singletons* ablegen, die JSP-relevante Funktionen bereitstellen. Eine wichtige Funktion ist die Methode `createValueBindung`, die einen Zugriff auf die entsprechenden Modellobjekte verschafft. Über die entsprechenden Methoden `setValue` und `getValue` der ValueBinding-Klasse kann damit auf Werte der Modellobjekte zugegriffen und diese auch mit neuen Werten überschrieben werden.

Anstatt jedoch über die Angabe eines Listeners ein Event auszulösen, das in einer entsprechenden Klasse verarbeitet wird, kann in einer Command-Komponente auch direkt mittels des *MethodBindings* eine Aktionsmethode ausgeführt werden.

```
<h:commandButton action="#{Person.doLogin}" value="Login" />
```

Durch den Ausdruck im `action`-Attribut wird JSF signalisiert, hier nicht in der Anwendungskonfigurationsdatei nach einem passenden Eintrag bezüglich der Navigation zu suchen, sondern die Methode `doLogin` auf einem Objekt des Klasse `PersonBean` auszuführen. Das Backing Bean trägt dabei den Bezeichner `Person`. Der Rückgabewert dieser Aktionsmethode regelt dann das weitere Navigationsverhalten (das dann natürlich den Rückgabewerte wieder gegen die Einträge aus der Konfigurationsdatei überprüft). Die Aktionsmethode, die über das `MethodBinding` angegeben wird, darf keine Parameter erwarten und muss als Rückgabewert einen String zurückliefern. Natürlich muss die Methode selbst auch `public` deklariert sein.

Component Binding

Backing Beans dienen der Unterstützung von Abläufen innerhalb einer Faces-Seite. Mit Backing Beans können sowohl Werte als auch Methoden direkt über die JSF-Tags angesprochen werden. Daneben gibt es jedoch noch eine weitere Variante, Beans mit den entsprechenden Komponenten in Verbindung zu bringen. Über das so genannte *Component Binding* können Beans konkrete Komponenten als Eigenschaften aufweisen. Um dies zu demonstrieren, ein kurzer Auszug aus einer Beanklasse:

```
private UISelectBoolean boolComp;

...
  public void testComponentBinding( ActionEvent event ) {
    FacesContext context = FacesContext.getCurrentInstance();
    UIComponent component = event.getComponent();

    FacesMessage infoMsg = null;
    if ( boolComp.isSelected() ) {
      infoMsg = new FacesMessage( "Checkbox ist selektiert.",
                    "Checkbox ist selektiert." );
    } else {
      infoMsg = new FacesMessage( "Checkbox ist NICHT selektiert.",
                    "Checkbox ist NICHT selektiert." );
    } // else
    context.addMessage(component.getClientId(context), infoMsg);
  }

  public UISelectBoolean getBoolComp() {
    return boolComp;
  }

  public void setBoolComp(UISelectBoolean bool) {
    boolComp = bool;
  }
```

Listing 6.24: Erweitertes CustomerBean

In Listing 6.24 wurde die Klasse `CustomerBean` um eine Eigenschaft `boolComp` erweitert. Diese weist keine Eigenschaft in Form eines einfachen Typs wie `String` oder `int` auf, sondern eine vollständige Komponente. In einer Faces-Seite kann über einen Binding-Ausdruck dann direkt darauf referenziert werden:

```
<h:selectBooleanCheckbox title="Component Binding"
   binding="#{Customer.boolComp}" >
</h:selectBooleanCheckbox>
```

Listing 6.25: Zugriff über Backing Beans

Wie in Listing 6.25 zu sehen, ist die Eingabekomponente direkt mit der Eigenschaft `boolComp` des Backing Beans `Customer` verbunden. In entsprechenden Aktionsmethoden kann dann direkt auf die Komponente zugegriffen und der notwendige Wert abgefragt werden. Um dies zu verdeutlichen, wird auf der Seite eine Aktionsmethode aufgerufen. Diese wird durch das Betätigen des Buttons aufgerufen.

```
<h:commandButton id="submit" value="Submit"
   actionListener="#{Customer.testComponentBinding}" />
```

In der Aktionsmethode wird dann lediglich eine Meldung im FacesContext hinterlegt, ob die Checkbox selektiert ist oder nicht.

```
public void testComponentBinding( ActionEvent event ) {
  FacesContext context = FacesContext.getCurrentInstance();
  UIComponent component = event.getComponent();

  FacesMessage infoMsg = null;
  if ( boolComp.isSelected() ) {
    infoMsg = new FacesMessage( "Checkbox ist selektiert.",
      "Checkbox ist selektiert." );
  } else {
    infoMsg = new FacesMessage( "Checkbox ist NICHT selektiert.",
      "Checkbox ist NICHT selektiert." );
  } // else
  context.addMessage(component.getClientId(context), infoMsg);
}
```

Listing 6.26: Aktionsmethode

Ob diese Vorgehensweise, eine Komponente direkt anzusprechen, besser oder vorteilhafter ist, anstatt mit direkten Werten zu arbeiten, ist eine subjektive Frage. Beide Vorgehensweisen haben ihre Vorteile. Letzten Endes hängt es vom Entwickler ab, welche Variante zum Einsatz kommt.

Allgemeine Regeln

Bei der Anwendung des Managed-Bean-Konzepts gelten einige wichtige Regeln und Konventionen, die bei einem Einsatz beachtet werden sollten:

- Es ist ein Fehler, eine Bean-Klasse anzugeben, die nicht existiert bzw. die keinen öffentlichen Konstruktor ohne Parameter hat.
- Beim Initialisierungsparameter werden die einzelnen Werte entsprechend der Reihenfolge in der Anwendungskonfigurationsdatei gesetzt.
- Wenn ein Bean Eigenschaften besitzt, die nicht explizit über Initialisierungsparameter gesetzt werden, so wird die entsprechende setter-Methode auch nicht aufgerufen.
- Der Programmablauf erzeugt einen Fehler, wenn der Wert, der innerhalb eines value-Attributes nicht in den benötigten Wert der Bean-Eigenschaft konvertiert werden kann.
- Es dürfen keine zyklischen Referenzen auftreten, sprich es darf innerhalb eines value-Attributes nicht auf ein anderes Modellobjekt verwiesen werden, das in dem Initialisierungsparameter wiederum auf einen Parameter des ersten Beans verweist.

Diese Regeln sind an sich einleuchtend und auch nicht weiter erklärungsbedürftig. Wichtig ist jedoch noch folgende Einschränkung, auf die nochmals explizit eingegangen werden soll:

Innerhalb eines Beans darf nicht auf einen Wert eines anderen Beans verwiesen werden, das eventuell eine kürzere Gültigkeitsdauer besitzt. Besitzt z. B. ein Bean den Gültigkeitsbereich `application`, darf es nicht auf Eigenschaften von Beans mit dem Gültigkeitsbereich `session` oder `request` verweisen. Folgende Kombinationen sind daher nur möglich:

Gültigkeitsbereich des verweisenden Beans	**erlaubte Gültigkeitsbereiche des Verweis empfangenden Beans**
none	none
application	none, application
session	none, application, session
request	none, application, session, request

Tabelle 6.4: Erlaubte Kombinationen von Gültigkeitsbereichen

6.5 Validierung

Der Bereich der Datenvalidierung ist in fast jeder Webanwendung ein zentraler (und meist auch ungeliebter) Bestandteil der Benutzereingabeverarbeitung. Um einen Geschäftsprozess korrekt und fehlerfrei anstoßen zu können, müssen die Eingabedaten vollständig und fehlerfrei vorliegen. Damit die benötigten Daten aus einem Formular korrekt zur Weiterverarbeitung weitergeleitet werden können, findet sowohl auf Clientseite wie auch auf Serverseite eine Datenvalidierung statt.

Technisch könnten einfache Überprüfungen, z. B. ob ein Feld überhaupt ausgefüllt ist oder ein Eingabefeld einen numerischen Charakter aufweist, auf Clientseite mittels JavaScript stattfinden. Komplexere Prüfungen, ob z. B. ein gewählter Loginname bei einer Community-Anwendung bereits vergeben ist, kann nur auf Serverseite erfolgen, da hierbei meist ein Datenbankzugriff notwendig ist.

In JSF finden alle Prüfungen auf dem Server statt. Dies ist in den Standard-Renderern so umgesetzt. Sollte dieses Verhalten jedoch nicht erwünscht sein und stattdessen eine erste Validierung z. B. mittels JavaScript auf Clientseite erfolgen, können hierfür eigene Renderer geschrieben werden.

Bezugnehmend auf den Lebenszyklus einer JSF-Seite finden die Prüfungen **vor** dem Aktualisieren des Modellobjekts statt. Für die Validierung bietet das Framework bereits einige Standard-Validatoren an:

Validator-Tag	Beschreibung
`validateLength`	Bei einem Datentyp String kann die Länge der Eingabe überprüft werden.
`validateLongRange`	Überprüfung hinsichtlich eines Wertebereiches für den Datentyp long
`validateDoubleRange`	Überprüfung hinsichtlich eines Wertebereiches für den Datenyp double

Tabelle 6.5: Übersicht über die Standard-Validatoren

Die Standard-Validatoren sind bereits vollständig implementiert und bedürfen bei einer Einbindung in die eigene Applikation keinerlei Programmierkenntnisse. Es genügt, das entsprechende Validator-Tag mit einzubinden.

```
<h:inputText id="loginname" value="">
  <f:validateLength minimum="3"/>
</h:inputText>
```

Listing 6.27: Verwendung eines Validators

Wie einfach ein Validator auch von nicht programmiertechnisch versierten Webdesignern einzubinden ist, zeigt Listing 6.27. Ein einfaches verschachteltes Tag genügt, um wie in diesem Beispiel abzuprüfen, dass ein Wert für einen Loginnamen eine minimale Länge von drei Zeichen aufweist.

Zusätzlich zu den Standard-Validators können benutzerspezifische Validatoren geschrieben werden. Mit benutzerspezifischen Validatoren ist es bei Bedarf auch möglich, auf sämtliche JavaScript-Funktionalität zu verzichten sowie weitere Prüfungen einzubauen. Ebenfalls kann auch eine Validierung ausschließlich mit JavaScript realisiert werden. Mehr über benutzerspezifische Validatoren ist im Kapitel 9 zu finden.

Darstellung von Fehlermeldungen

Eine Datenvalidierung macht natürlich nur dann Sinn, wenn eventuelle Fehler entsprechend abgefangen und vor allem dem Benutzer auch angezeigt werden können. Ein Stack-Trace, wie er teilweise in Anwendungen auftritt, die einen Fehler nicht sauber abfangen, ist sicherlich nicht der geeignete Weg.

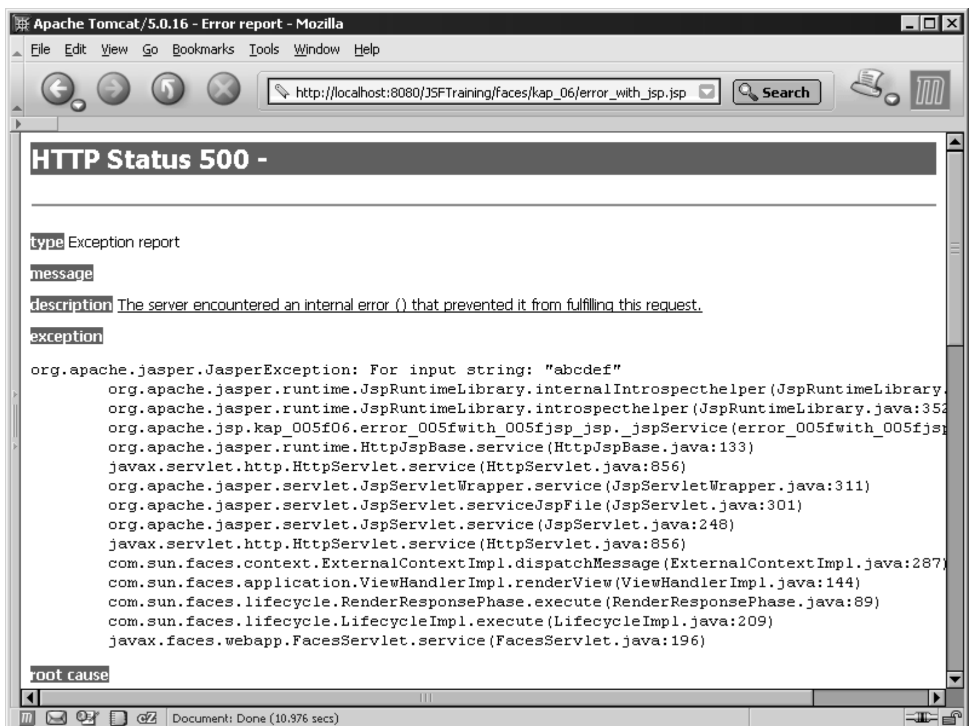


Abbildung 6.4: Unschöne Fehlermeldung als Stack-Trace

Um Fehlermeldungen darzustellen, existiert in JSF das Tag `<h:message />` bzw. `<h:messages />`. Damit wird ein Bereich definiert, in dem Fehlermeldungen dargestellt werden. Vom Ablauf her wurde bereits erwähnt, dass im Fehlerfalle auf dieselbe Seite zurückgeleitet wird, von der aus eine falsche Eingabe an den Server geschickt wurde, nur dass dann im Bereich `message` eine entsprechende Fehlermeldung angezeigt wird.

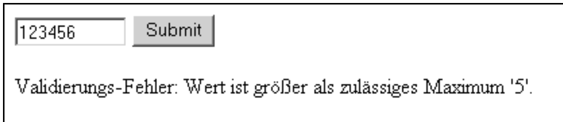


Abbildung 6.5: Darstellung eines Fehlers

In Abbildung 6.5 ist zu sehen, wie eine Fehlermeldung in einer Webseite dargestellt wird. In diesem Beispiel ist eine Angabe eines Wertes auf maximal fünf Zeichen beschränkt. Wird das Formular mit einer Eingabelänge von mehr als fünf Zeichen abgeschickt, wird eine entsprechende Fehlermeldung erzeugt und an der dafür vorgesehenen Stelle (bei `<h:messages />`) ausgegeben.

Optional können dem `<h:messages>`-Tag Stylesheet-Angaben mitgegeben werden, so dass eine Ausgabe anders formatiert werden kann, beispielsweise in einer anderen Farbe oder Schriftart.

```
<h:messages style="color: red;
    font-family: arial;
    text-decoration: underline"/>
```

Listing 6.28: Formatierte Ausgabe einer Fehlermeldung

Sollen an einer speziellen Stelle nur Fehlermeldungen einer bestimmten Komponente erscheinen, kann auch dies dem Tag mitgegeben werden.

```
<h:message for="firstname" />
```

Durch das `for`-Attribute wird geregelt, dass an dieser Stelle lediglich Fehlermeldungen erscheinen, die sich auf die Eingabekomponente mit der Bezeichnung `firstname` beziehen. Es sollte jedoch beachtet werden, dass das `for`-Attribut lediglich im `message`-Tag, nicht aber im `messages`-Tag zur Verfügung steht.

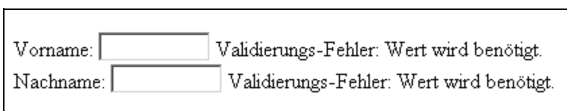


Abbildung 6.6: Fehlerausgabe pro Komponente

In Abbildung 6.6 wurde neben jedem Eingabefeld eine Ausgabe der Fehler definiert. So kann pro Eingabefeld ein Bereich definiert werden, in dem die genau für dieses Eingabefeld vorliegenden Fehlermeldungen visualisiert werden. Dies ist vor allem bei umfangreichen Formularen sehr hilfreich. Denn wenn in einem komplexen Formular unter Umständen mehrere Fehlermeldungen gesammelt an einer Stelle stehen, muss sich ein Anwender erst orientieren, in welchem Feld eine falsche Eingabe überhaupt vorliegt.

Achtung:

Es existieren zwei Tags zur Darstellung von Fehlern: Zum einen das `messages`-Tag, das sämtliche Fehler anzeigt, sowie das `message`-Tag, das lediglich eine einzige Fehlermeldung darstellen kann. Sollten Fehlermeldungen nur für eine Komponente ausgegeben werden, kann das `message`-Tag mit dem zusätzlichen Attribut `for` versehen werden. Das `messages`-Tag dagegen hat dieses Attribut nicht.

Für die folgenden Beispiele der Validatoren existieren auf beiliegender CD entsprechende Beispieldateien, die für jeden Standard-Validator deren genaue Verwendung aufzeigen.

Der Required-Validator

Für eine Überprüfung, ob überhaupt ein Wert eingegeben wurde oder das Feld leer gelassen wurde, erfolgt eine indirekte Verwendung eines Validators. Es existiert kein spezielles allein stehendes Tag für einen Required-Validator, sondern für jede Eingabekomponente wird ein Attribut `required` abgefragt.

```
<h:inputText value="" required="true" size="10" />
```

Listing 6.29: Verwendung des required-Attributes

Das `required`-Attribut prüft auf das Vorhandensein einer Eingabe, unabhängig von deren Datentyp. Somit kann auch eine Kombination des `required`-Attributes mit anderen Validatoren erfolgen.

Der Längen-Validator

Der Längen-Validator legt die alphanumerische Eingabelänge in einem Eingabefeld fest. Dabei kann dem Validator eine minimale und eine maximale Länge mitgegeben werden. Wichtig ist, dass auch bei einer Eingabe von einer Minimallänge von 1 kein Fehler erzeugt wird, wenn keinerlei Eingabe erfolgt. Soll sichergestellt werden, dass überhaupt ein Wert eingegeben wird, so muss das Attribut `required` im Eingabetag dazu verwendet werden. Erst wenn überhaupt ein Wert eingegeben wurde, greift der Längenvalidator, um die angegebenen Minimal- und Maximalgrenzen zu überprüfen.

```
<h:form id="myform">
  <h:inputText id="zip" value="" size="10">
    <f:validateLength minimum="1" maximum="5" />
  </h:inputText>

  <h:commandButton id="submit" value="Submit" />
  <p>
  <h:messages />
</h:form>
```

Listing 6.30: Verwendung des Längen-Validators

In Listing 6.30 ist gut zu erkennen, wie einfach ein Validator für eine Längenüberprüfung miteingebunden werden kann. Klar ist, dass komplexe Überprüfungen wie zuvor schon erwähnt, nach wie vor »von Hand« entwickelt werden müssen. Jedoch erspart man sich bei Verwendung der Standard-Validatoren eine Menge Programmierarbeit.

Ein weiterer Vorteil liegt darin begründet, dass die Validatoren auch von Webdesignern angegeben werden können, die im Regelfall nicht in der Lage sind, selbst solche Prüfungen in Java zu codieren (was auch nicht deren Aufgabe ist).

Eine Längenbegrenzung muss jedoch nicht ausschließlich mittels des Längen-Validators erfolgen. Das `inputText`-Tag akzeptiert hierbei das Attribut `maxlength`. Damit wird im Tag selbst bereits eine maximale Eingabelänge definiert. Der Vorteil dieses Ansatzes liegt darin, dass durch diese Angabe das Eingabefeld so gerendert wird, dass es dem Benutzer überhaupt nicht möglich ist, einen längeren Text einzugeben.

```
<h:form id="myform">
  <h:inputText id="zip" value="" maxlength="5" size="10">
    <f:validateLength minimum="1" />
  </h:inputText>

  <h:commandButton id="submit" value="Submit" />
  <p>
  <h:messages />
</h:form>
```

Listing 6.31: Definition der maximalen Eingabelänge mit Hilfe des Attributs `maxlength`

In Listing 6.31 wird ein Eingabefeld erzeugt, in das ein Benutzer maximal 10 Zeichen eingeben kann. Mehr Zeichen können nicht eingegeben werden, da das Eingabefeld eine weitere Eingabe nach 10 Zeichen überhaupt nicht zulässt. Mit dem Validator wird zusätzlich noch die Minimallänge festgelegt. Bei Bedarf könnte als dritter Validator noch das `required`-Attribut hinzugenommen werden, das überprüft, ob überhaupt eine Eingabe vorhanden ist.

Der LongRange-Validator

Mit Hilfe des LongRange-Validators kann ein numerischer Wertebereich angegeben werden, in dem sich ein Eingabewert befinden muss. So kann beispielsweise zur Prüfung von Postleitzahlen ein Wertebereich bis maximal 99999 angegeben werden. Es kann jedoch ein beliebiger Wertebereich gewählt werden, wobei es allerdings nicht möglich ist, mehrere nicht zusammengehörige Wertebereiche anzugeben (eine Überprüfung, dass ein Wert zwischen 0 und 10 oder 30 und 40 liegt, ist mit den Standardvalidatoren nicht möglich).

```
<h:form id="myform">

Bitte geben Sie genau einen long-Wert ein:
  <h:inputText id="middleInitial" size="10" value="" >
    <f:validateLongRange minimum="-10" maximum="20" />
  </h:inputText>

  <h:commandButton id="submit" value="Submit" />
  <p>
  <h:messages />
</h:form>
```

Listing 6.32: Verwendung des LongRange-Validators

Die Angabe der Attribute `minimum` und `maximum` sind optional, so wird bei Fehlen eines Attributes lediglich auf das vorhandene Attribut geprüft. Folgende Syntaxmöglichkeiten sind daher erlaubt:

```
<f:validateLongRange />
<f:validateLongRange minimum="-10" />
<f:validateLongRange maximum="20" />
<f:validateLongRange minimum="-10" maximum="20" />
```

Tipp: Typüberprüfung

Bei Verwendung der Range-Validatoren für den Datentyp Long und Double können beide Attribute `minimum` und `maximum` weggelassen werden. Es wird dann lediglich eine Typprüfung durchgeführt, ob der eingegebene Wert ein gültiger Long- bzw. Double-Datentyp ist.

Der LongRange-Validator kann ausschließlich im Zusammenhang mit einem Eingabefeld verwendet werden. An anderen Komponenten ist die Verwendung nicht zugelassen bzw. auch überhaupt nicht sinnvoll. Sind beide Attribute `minimum` und `maximum` angegeben, muss das `maximum`-Attribut größer sein als das `minimum`-Attribut.

Der DoubleRange-Validator

Der DoubleRange-Validator unterscheidet sich nicht wesentlich vom LongRange-Validator. Auch hierbei kann wiederum ein Wertebereich angegeben werden, in dem sich die Benutzereingaben befinden müssen.

```
<h:form id="myform">

  Bitte geben Sie einen double-Wert ein:
  <h:inputText size="10" value="" >
    <f:validateDoubleRange minimum="-5.0" maximum="5" />
  </h:inputText>

  <h:commandButton id="submit" value="Submit" />
  <p>
  <h:messages />
</h:form>
```

Listing 6.33: Verwendung des DoubleRange-Validators

Auch hierbei gilt wieder, dass der Wert des `maximum`-Attributs größer sein muss als der Wert des `minimum`-Attributs.

6.6 Meldungen und Fehlermeldungen

Zur Darstellung von Fehlermeldungen sowie zur Anzeige sonstiger Meldungen steht in JSF die Klasse `FacesMessage` bereit. Objekte vom Typ `FacesMessage` sind zunächst einmal Nachrichten, die einer Komponente zugeordnet sein können, aber nicht müssen. In den Nachrichten können Fehlermeldungen aufgrund einer fehlgeschlagenen Datenvalidierung einer Benutzereingabe enthalten sein oder Fehlermeldungen, die aus einem Programmablauf entstanden sind. Natürlich muss nicht jede Nachricht automatisch einen Fehler bedeuten, es ist auch möglich, Warnhinweise oder rein informative Nachrichten darin zu verpacken.

Wie im letzten Abschnitt ausführlich behandelt, liefert JSF bereits einige Validatoren standardmäßig mit, mit denen eine Bereichs- und Längenprüfung stattfinden kann. Zugleich werden auch Standardfehlermeldungen für die eventuell auftretenden Fehler mitgeliefert. Diese sind als Ressourcendatei in den Bibliotheken enthalten. Im Umfang ist auch bereits eine deutsche Ressourcendatei enthalten, jedoch sind die Übersetzungen nicht sonderlich glücklich gewählt. Allerdings ist es möglich (und auch sehr empfehlenswert), bei Verwendung der Standard-Validatoren die entsprechenden Fehlermeldungen selbst zu definieren.

Doch nicht nur bei den Validatoren können Fehler auftreten, die das Anzeigen von Meldungen zur Folge haben. Oftmals kann es auch im Programmablauf selbst zu Situationen kommen, in denen dem Benutzer eine Meldung angezeigt werden soll.

Hier muss im Programmcode eine Meldung vom Typ `FacesMessage` erzeugt und in den Kontext gestellt werden. Von dort wird die Meldung automatisch angezogen und dem Benutzer zur Anzeige gebracht.

Fehlermeldungen bei Validatoren

In JSF ist für die Standardvalidatoren bereits eine Ressourcendatei enthalten, die Texte für Fehlermeldungen beinhaltet. Als Beispiel wird nochmals der Längenvalidator verwendet:

```
<h:form id="frmInput">
  <h:inputText id="loginname" value="" size="10">
    <f:validateLength minimum="5" maximum="10" />
  </h:inputText>
  <h:commandButton id="submit" value="Submit" />
  <p>
  <h:messages />
</h:form>
```

Listing 6.34: Längenvalidator und Fehlerausgabe

Im Beispiel ist ein Eingabefeld vorgegeben, in dem eine Eingabe von mindestens fünf Stellen, höchstens jedoch zehn Stellen erwartet wird. Dies ist z.B. dann der Fall, wenn ein Loginnamen für eine Anwendung gewählt werden soll oder auch ein Passwort. Hier ist eine Mindestlänge vorgeschrieben sowie eine maximale Länge, die innerhalb einer Anwendung verwendet werden kann.

Bei einer zu kurzen Eingabe erscheint folgende Ausgabe:

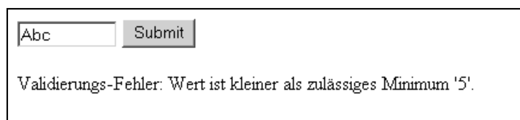


Abbildung 6.7: Standardausgabe des Längenvalidators

Die Aussage, dass ein zulässiges Minimum von fünf unterschritten wurde, ist prinzipiell zwar korrekt, aber der eigentliche Text ist sicherlich verbesserungsfähig.

Eine Faces-Anwendung hat jedoch die Möglichkeit, die vordefinierten Einträge zu überschreiben, indem mit Hilfe eines `message-bundle`-Tags in der Anwendungskonfigurationsdatei eine eigene Ressourcendatei hinterlegt wird. In dieser können dann eigene Meldungen angegeben werden.

```
<application>
  <message-bundle>mymessages</message-bundle>
</application>
```

Listing 6.35: Hinterlegung einer Ressourcendatei

Listing 6.35 zeigt einen Auszug aus der Konfigurationsdatei, in der eine Ressource mit dem Namen *mymessages* hinterlegt wurde. Wichtig ist, dass sich die Ressourcendatei mit dem Namen *mymessages.properties* im Klassenpfad befindet. Die Ressourcendateien sind jedoch nichts Faces-Spezifisches, sondern entsprechen dem Konzept der Ressourcen allgemein in Java. So kann für jedes Land bzw. jede Lokale eine eigene Ressourcendatei angelegt werden, die automatisch von der Anwendung angezogen wird. Mehr dazu folgt in Abschnitt 6.13 Internationalisierung.

Um die vorhandenen Fehlermeldungen zu überschreiben, muss der Schlüssel exakt gleich benannt werden. Auch ist bei der Benennung der Texte auf eventuelle Parameter, die mit `{...}` definiert sind, zu achten.

Schlüssel	**Standardtext**
`javax.faces.validator.NOT_IN_RANGE`	Validierungs-Fehler: Spezifiziertes liegt nicht zwischen {0} und {1}.
`javax.faces.validator.DoubleRangeValidator.LIMIT`	Validierungs-Fehler: Spezifiziertes Attribut kann nicht in einen Double-Typ umgewandelt werden.
`javax.faces.validator.DoubleRangeValidator.MAXIMUM`	Validierungs-Fehler: Wert ist größer als zulässiges Maximum "{0}".
`javax.faces.validator.DoubleRangeValidator.MINIMUM`	Validierungs-Fehler: Wert ist kleiner als zulässiges Minimum "{0}".
`javax.faces.validator.DoubleRangeValidator.TYPE`	Validierungs-Fehler: Wert ist nicht vom richtigen Datentyp.
`javax.faces.validator.LengthValidator.LIMIT`	Validierungs-Fehler: Spezifiziertes Attribut kann nicht in korrekten Typ umgewandelt werden.
`javax.faces.validator.LengthValidator.MAXIMUM`	Validierungs-Fehler: Wert ist größer als zulässiges Maximum "{0}".
`javax.faces.validator.LengthValidator.MINIMUM`	Validierungs-Fehler: Wert ist kleiner als zulässiges Minimum "{0}".
`javax.faces.validator.LongRangeValidator.LIMIT`	Validierungs-Fehler: Spezifiziertes Attribut kann nicht in korrekten Typ umgewandelt werden.

Tabelle 6.6: Standardfehlermeldungen

Schlüssel	Standardtext
`javax.faces.validator.LongRangeValidator.MAXIMUM`	Validierungs-Fehler: Wert ist größer als zulässiges Maximum "{0}".
`javax.faces.validator.LongRangeValidator.MINIMUM`	Validierungs-Fehler: Wert ist kleiner als zulässiges Minimum "{0}".
`javax.faces.validator.LongRangeValidator.TYPE`	Validierungs-Fehler: Wert ist nicht vom richtigen Datentyp.
`javax.faces.component.UIInput.REQUIRED`	Validierungs-Fehler: Wert wird benötigt.

Tabelle 6.6: Standardfehlermeldungen (Fortsetzung)

In Tabelle 6.6 sind die Schlüssel aufgezählt, die die Standardfehlermeldungen bei Validatoren betreffen. Diese können entweder komplett oder auch nur selektiv überschrieben werden. Listing 6.36 zeigt einen Ausschnitt aus der selbst erzeugten Datei *mymessages_de.properties*, die im Klassenpfad hinterlegt wurde.

```
javax.faces.validator.LengthValidator.MAXIMUM=
   Die Eingabe ist zu lang. Sie übersteigt das gesetzte
   Maximum von ''{0}''.
javax.faces.validator.LengthValidator.MINIMUM=
   Der eingegebene Wert ist zu klein. Die Eingabe muss
   eine Mindestlänge von ''{0}'' haben.
```

Listing 6.36: Auszug aus einer eigenen Ressourcendatei

Mit diesen Eintragungen wird das Standardverhalten überschrieben und man erhält seine eigenen definierten Fehlermeldungen bei Bedarf angezeigt.

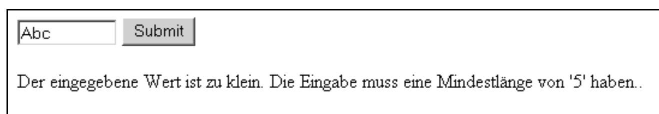


Abbildung 6.8: Fehlermeldung mit selbst definiertem Inhalt

Erzeugen einer FacesMessage

Fehlermeldungen werden jedoch nicht ausschließlich von JSF selbst erzeugt, sondern können durchaus auch im eigenen Quellcode auftreten. Statt einfach eine Exception zu werfen, die eine Ausgabe in einer Protokolldatei schreibt, ist es natürlich eleganter, eine entsprechende Meldung zu erzeugen, die dem Benutzer an der Oberfläche angezeigt wird.

Für das Erzeugen eigener Meldungen kann die Klasse `FacesMessage` genutzt werden. Eine einfache Möglichkeit ist es, diese zu instanzieren und die entsprechenden Fehlermeldungen mitanzugeben.

```
FacesContext context = FacesContext.getCurrentInstance();
UIComponent component = evtent.getComponent();

FacesMessage errMsg = new FacesMessage(
  FacesMessage.SEVERITY_ERROR,
  "Loginname bereits vorhanden, bitte einen anderen Namen wählen.",
  "Der Loginname ist bereits vorhanden. Bitte wählen Sie
   einen anderen Loginnamen aus." );

context.addMessagecomponent.getClientId(context, errMsg);
```

Listing 6.37: Erzeugen von Meldungen

Das Codefragment aus Listing 6.37 wurde einem Listener übernommen, daher wird die Komponente über den Aufruf

```
UIComponent component = event.getComponent();
```

ermittelt. Die Komponente wird deshalb benötigt, um die Fehlermeldung explizit an die Komponente anzuhängen. Die erzeugte Fehlermeldung wird anschließend dem FacesContext hinzugefügt und wird daraufhin automatisch in der JSF-Seite angezeigt. Dazu ist es natürlich wieder erforderlich, dass ein entsprechendes `messages`-Tag in der betreffenden Seite vorhanden ist.

Wird dagegen der Methode `addMessage` kein Bezeichner einer Komponente mitgegeben, sondern `null`, erscheint zwar die Meldung genauso auf der Seite, ist jedoch nicht explizit einer Komponente zugeordnet. Es ist erst dann relevant, wenn im Tag `message` das `for`-Attribut verwendet wird. Durch das `for`-Attribut kann zusätzlich angegeben werden, dass nur Fehlermeldungen einer bestimmten Komponente angezeigt werden.

```
<h:message for="loginfield" />
```

zeigt beispielsweise lediglich Fehlermeldungen an, die die Komponente mit dem Bezeichner `loginfield` betreffen.

Als Argument bei einer `FacesMessage` wird zudem ein Grad der Schwere mitangegeben, von denen es in JSF insgesamt vier Stück gibt:

- `FacesMessage.SEVERITY_INFO`
- `FacesMessage.SEVERITY_WARN`
- `FacesMessage.SEVERITY_ERROR`
- `FacesMessage.SEVERITY_FATAL`

Diese Liste ist in aufsteigender Reihenfolge angeordnet, so dass `FacesMessage.SECURITY_FATAL` den schlimmsten Fehlergrad darstellt. Wird dieses Argument nicht explizit mitangegeben, wird automatisch `SEVERITY_INFO` unterstellt.

Internationalisierung

Die Variante, direkt eine Instanz der Klasse `FacesMessage` zu erzeugen, der zwei Texte mitgegeben werden, hat einen gravierenden Nachteil: Es können keine mehrsprachigen Texte verwendet werden. Ist dies in der Anwendung jedoch gefordert, ist die Verwendung von so genannten *ResourceBundles* notwendig. Eine Anwendung kann mehrere solcher *ResourceBundles* aufweisen, von denen jede durch einen eindeutigen Bezeichner gekennzeichnet ist. Ein weiterer Vorteil liegt darin, dass die Texte jederzeit relativ problemlos geändert werden können, ohne dass ein Eingriff in den Quellcode notwendig ist.

Durch Verwendung unterschiedlicher *ResourceBundles* können Gruppen von Meldungen erstellt werden, um damit eine logische Unterteilung der Texte zu erreichen.

```
btnSubmit=Abschicken
btnOk=Ok
btnCancel=Abbrechen
btnSave=Speichern
txt_1=Testausgabe
txt_2=Überschrift
msgErr=Es ist ein Fehler aufgetreten.
msgTimeout=Es ist ein Timeout aufgetreten. Bitte neu anmelden.
```

Listing 6.38: Auszug aus einer Ressourcendatei

Eine Ressourcendatei ist eine einfache Textdatei, in der pro Zeile ein Schlüsselwort und ein entsprechender Text zugeordnet sind. Diese Ressourcendateien können für jede Sprache bzw. für jede Lokale separat gepflegt werden. Aus der Anwendungsentwicklung heraus kann wie folgt darauf zugegriffen werden:

```
ResourceBundle bundle = null;
try {
  bundle = ResourceBundle.getBundle( "commandBundle",
              context.getViewRoot().getLocale() );
} catch (Exception exc) {
  // .. error handling
}
String errTxt = bundle.getString( "key" );
FacesMessage errMsg = new FacesMessage( errTxt, errTxt );
context.addMessage(component.getClientId(context), errMsg);
```

Listing 6.39: Hinzufügen einer Fehlermeldung

Zunächst wird das `ResourceBundle` über den Aufruf `getBundle` geladen. Als Parameter wird einmal der Name der Ressource (ohne die Endung *.properties*) mitangegeben sowie die Lokale, die aus der aktuellen View ermittelt wird. Im Anschluss daran kann über `getString` und dem Schlüssel des Eintrages der eigentliche Wert ausgelesen werden, aus dem daraufhin ein Objekt der Klasse `FacesMessage` erzeugt wird. Dieses wird dann dem Kontext hinzugefügt und kann an der Oberfläche entsprechend angezeigt werden.

6.7 Datenkonvertierung

UI-Komponenten, sprich Komponenten für die graphische Benutzeroberfläche, werden in JSF durch so genannte *Renderer* standardmäßig in Html gewandelt und im Browser dargestellt. Die Komponenten beziehen dabei ihre Werte aus Modellobjekten, in die sie die Werte nach erfolgreicher Validierung auch wieder zurückschreiben. Html kennt jedoch keine Datentypen, sondern nur einfachen Text. Sämtliche Benutzereingaben in Eingabefeldern, Listboxen oder sonstigen Eingabekomponenten liegen zuerst einmal in Textform vor. Im Objekt selbst jedoch existieren die Werte in ihrem »ursprünglichen« Datentyp. Damit liegen die Daten einer Komponente in zweifacher Form vor: Zum einen in der *Modellsicht*, in der die korrekten Daten in Form von Membervariablen mit dazugehörigem Datentyp gespeichert sind, und zum anderen in der *Präsentationssicht*, in der die Daten vom Benutzer verändert werden können. Als Beispiel kann eine Integer-Zahl angezogen werden; intern, also im Datenmodell, ist diese als `int` oder `java.lang.Integer` abgespeichert. Sobald der Wert jedoch in einer HTML-Seite angezeigt wird, ist dieser als Textstring in einem Eingabe- oder Ausgabefeld zu finden.

Um die Daten aus der Präsentationsschicht (die wie bereits erwähnt sämtliche Daten als Strings beinhaltet) in die Modellsicht zu überführen, findet eine *Datenkonvertierung* statt. Im Normalfall übernimmt diese Datenkonvertierung der Renderer der Komponente selbst, so dass sich der Anwendungsentwickler darum nicht kümmern muss. Es kann jedoch vorkommen, dass die vordefinierten Konvertierungen nicht ausreichen und eigene spezifische Konverter geschrieben werden müssen (vgl. dazu Kapitel 9).

JavaServer Faces stellt bereits die gebräuchlichsten Konverter bereit, die sich alle im Paket `javax.faces.convert` befinden:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`

- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`
- `NumberConverter`
- `ShortConverter`

Alle Standardkonverter implementieren ein gemeinsames Interface `Convert` im Paket `javax.faces.convert`, das genau zwei Methoden aufweist:

- `getAsObject`
- `getAsString`

Diese beiden Methoden sind dafür verantwortlich, dass die Daten aus einer Darstellungsform in die andere überführt werden können. Um die Standardkonverter aus JSF verwenden zu können, gibt es verschiedene Möglichkeiten. Zum einen kann im `converter`-Attribut einer Eingabekomponente der Konverter explizit angegeben werden oder es kann ein eigenes Konverter-Tag hinzugefügt werden, in dem mittels einer Converter-Id der Konverter benannt wird.

Für die folgenden Beispiele wird das in Listing 6.40 abgebildete Modellobjekt verwendet:

```
package com.edu.jsf.bsp.bean;

 import java.util.Date;

/**
 * Bean, das einen Kunden abbildet
 */
public class CustomerBean {

  private String firstname;
  private String lastname;
  private int zip;
  private String city;
  private double turnover;
  private Date birth;

  /**
```

```
   * Getter-Methoden
   */
  public String getFirstname() {
    return firstname;
  }

  public String getLastname() {
    return lastname;
  }

  public int getZip() {
    return zip;
  }

  public String getCity() {
    return city;
  }

  public Date getBirth() {
    return birth;
  }

  public double getTurnover() {
    return turnover;
  }

  /**
   * Setter-Methoden
   */
  public void setFirstname(String string) {
    firstname = string;
  }

  public void setLastname(String string) {
    lastname = string;
  }

  public void setZip(int i) {
    zip = i;
  }

  public void setCity(String string) {
    city = string;
  }

  public void setBirth(Date date) {
    birth = date;
  }

  public void setTurnover(double db) {
```

```
        turnover = db;
      }
    }
```

Listing 6.40: Ein einfaches KundenBean

Für eine Eingabe der Kundendaten wird eine einfache Faces-Seite verwendet, die in Abbildung 6.9 zu sehen ist.

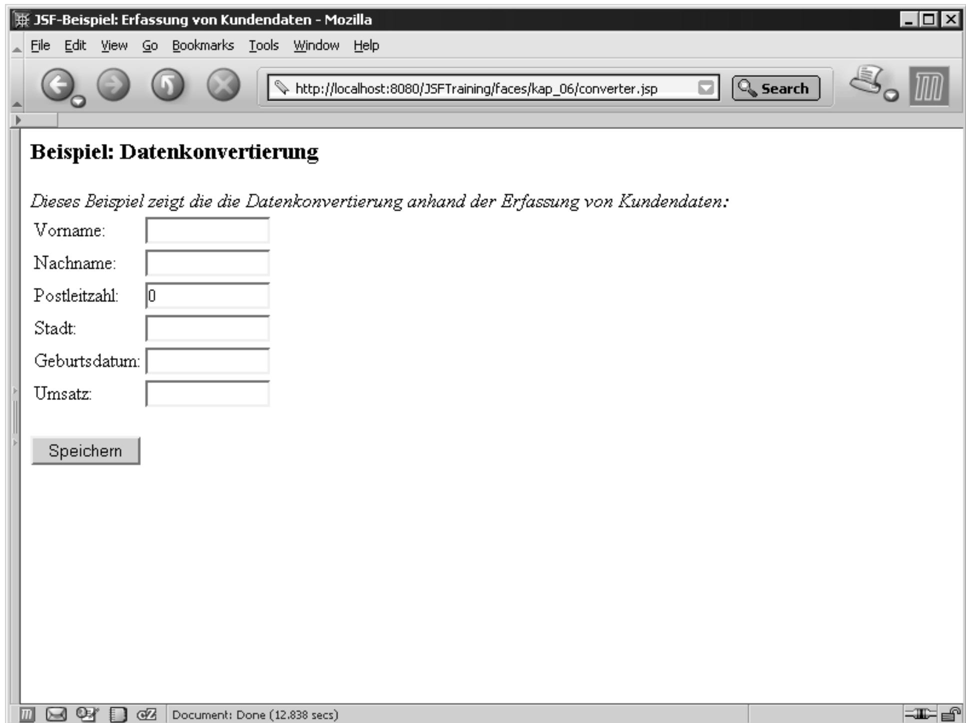


Abbildung 6.9: Erfassung von Kundendaten

Diese Minimalanwendung besteht lediglich aus einer Seite. Sobald der Benutzer den Speichern-Button drückt, wird wieder auf dieselbe Seite verwiesen (da für die ausgelöste Aktion keine Entsprechung in den Navigationsregeln vorhanden ist). Allerdings sind die Werte dann bereits im Modellobjekt gespeichert (der Lebenszyklus wurde komplett durchlaufen). Dabei muss bei den Feldern *Vorname*, *Nachname* und *Stadt* keine Konvertierung vorgenommen werden, da diese Felder auch als String im Modellobjekt gespeichert werden. Bei *Postleitzahl*, *Geburtstag* und *Umsatz* dagegen muss eine Konvertierung vorgenommen werden.

```
...
Vorname:
<h:inputText value="#{Customer.firstname}" size="15" />
...
Nachname:
<h:inputText value="#{Customer.lastname}" size="15" />
...
Postleitzahl:
<h:inputText value="#{Customer.zip}"
   converter="javax.faces.Integer" size="15">
</h:inputText>
...
Stadt:
<h:inputText value="#{Customer.city}" size="15" />
...
Geburtsdatum:
<h:inputText value="#{Customer.birth}" size="15">
  <f:convertDateTime dateStyle="short" type="date" />
</h:inputText>
...
Umsatz:
<h:inputText value="#{Customer.turnover}" size="15">
  <f:convertNumber maxIntegerDigits="2" groupingUsed="true"
     minFractionDigits="2" />
</h:inputText>
...
```

Listing 6.41: Nutzung der Standardkonverter

Listing 6.41 zeigt einen Auszug aus der Faces-Seite, in der die Eingabekomponenten für die Kundendaten hinterlegt sind. Es fällt auf, dass für die numerische Eingabe bei Postleitzahl ein Konverter-Attribut angegeben wurde, beim Umsatz jedoch ein eigenes Konverter-Tag verwendet wird. Beim Feld Umsatz allerdings könnte die Angabe eines Konverters entfallen, da in diesem Fall JSF automatisch versucht, den passenden Standardkonverter zu verwenden. Wird jedoch die Konverter-Angabe bei der Angabe des Geburtsdatums weggelassen, läuft dies zwangsläufig auf einen Fehler hinaus. Denn im Gegensatz zu einem `double`-Wert gibt es bei einem Datumswert eine Vielzahl von Varianten, in denen der Wert dargestellt werden kann. So kann der Monat ausgeschrieben sein, an erster oder zweiter Stelle stehen usw. Aus diesem Grund muss einem Datum explizit ein Konverter angegeben werden, wohingegen beim Feld Umsatz dies nicht zwingend benötigt wird.

Des Weiteren fällt auf, dass bei der Eingabe der Postleitzahl ein Attribut `converter` im `inputText`-Tag verwendet wurde, bei Eingabe des Geburtsdatums jedoch ein eigenes Tag. Dies hat die Bewandtnis, dass im Falle des `Integer-Konverters` keine weiteren Attribute mehr mitgegeben werden müssen, im Falle der Datumskonvertierung jedoch weitere Angaben wie `dateStyle` oder `type` notwendig sind. Daher ist in letzterem Fall ein eigenes Tag zu verwenden.

Konvertierungsattribute

Bei der automatischen Konvertierung von Ganzzahlen wie in Abbildung 6.9, in der eine Postleitzahl in einen entsprechenden int-Datentyp konvertiert wurde, ist es klar, wie die Konvertierung selbst vonstatten gehen muss: Sämtliche Zahlen des Textstrings werden komplett in eine int-Zahl umgewandelt. Bei anderen Datentypen, wie z.B. bei einem Datum, ist das Unterfangen schon etwas schwieriger. Der Konverter muss wissen, in welcher Form das Datum eingegeben wird, um eine erfolgreiche Konvertierung durchführen zu können. Das Datum kann dabei z.B. in einer Kurzform (»12.08.03«) oder einer ausführlichen Form (»Sonntag, 10. Januar 2004«) hinterlegt sein. Um diese zusätzlichen Attribute angeben zu können, können bei den Konverterangaben zusätzliche Informationen angegeben werden.

Für die Eingabe der Postleitzahl im CustomerBean-Beispiel kann die Eingabe der Postleitzahl mit Hilfe des Integer-Konverters realisiert werden:

```
<h:inputText value="#{Customer.zip}"
   converter="javax.faces.Integer" size="15">
</h:inputText>
```

Listing 6.42: Eingabe einer Zahl mit Hilfe eines Konverters

Für die Eingabe des Umsatzes könnte das `converter`-Attribut weggelassen werden, was ebenfalls problemlos funktioniert. Angenommen jedoch, der Umsatz soll als Währung oder sogar mit einem Tausendertrenner formatiert werden, kann nicht mehr nur ein `converter`-Attribut verwendet werden. Stattdessen ist ein eigenes Konverter-Tag zu verwenden, dem in den Attributen zusätzliche Parameter mit übergeben werden können.

```
<h:inputText value="#{Customer.turnover}" size="15">
  <f:convertNumber maxIntegerDigits="2" groupingUsed="true"
     minFractionDigits="2" />
</h:inputText>
```

Listing 6.43: Verwendung des Standardkonverters für ein Datumsfeld

In Listing 6.43 wurde das Umsatzfeld so aufbereitet, dass ein Tausendertrenner und eine Mindestnachkommastellenzahl von zwei verwendet wird.

In Listing 6.41 ist die Datumseingabe näher spezifiziert durch das Attribut `dateStyle`. Eine einfache Angabe des Konverters `Date` reicht in diesem Falle nicht mehr aus, daher muss das Format der Eingabe mit Hilfe des Attribute-Tags näher definiert werden. Auch der Datumskonverter wird in einem separaten Abschnitt nochmals näher erläutert.

Verwendung des NumberConverters

Wie bereits in Listing 6.41 gesehen, existieren für die Zahlenkonvertierung mit Hilfe des NumberConverters eine Reihe von Optionen, die dem `convertNumber`-Tag mitgegeben werden können.

Attribut	Typ	Beschreibung
currencyCode	String	ISO 4217-Ländercode, wird jedoch nur bei der Formatierung von Währungen angezogen
currencySymbol	String	Währungssymbol bei der Formatierung von Währungen
groupingUsed	boolean	gibt an, ob Tausendertrenner verwendet werden.
integerOnly	boolean	Es wird lediglich der integer-Anteil einer Eingabe geparst.
maxFractionDigits	int	maximale Anzahl an Stellen, die im Nachkommastellenbereich formatiert werden
maxIntegerDigits	int	maximale Anzahl an Stellen, die im Vorkommastellenbereich formatiert werden
minFractionDigits	int	minimale Anzahl an Stellen, die im Nachkommastellenbereich formatiert werden
minIntegerDigits	int	minimale Anzahl an Stellen, die im Vorkommastellenbereich formatiert werden
parseLocale	String oder Locale	Eine explizite Lokale kann angegeben werden, um den Wert einzulesen und zu formatieren.
pattern	String	Es kann auch ein benutzerspezifisches Format angegeben werden, wie der Wert eingelesen und formatiert werden soll.
type	String	Es kann angegeben werden, ob der Wert als `number`, `currency` oder `percentage` eingelesen und formatiert wird. Bei keiner Angabe wird `number` verwendet.

Tabelle 6.7: Attribute bei convert Number

Wie in Tabelle 6.7 zu erkennen ist, bietet das `convertNumber`-Tag eine große Anzahl an Möglichkeiten, Nummernwerte einzulesen und entsprechend formatiert auszugeben. Sollte es mit den vorhandenen Attributen nicht ausreichen, eine gewünschte Formatierung zu bekommen, kann durch das `pattern`-Attribut zusätzlich noch ein komplett eigenes Format angegeben werden.

Verwendung des DateTimeConverters

Neben der Zahlenformatierung und -konvertierung ist die Behandlung von Datums- und Zeitwerten ebenfalls ein häufig auftretendes Szenario. Analog zum `convertNumber`-Tag stellt auch das `convertDateTime`-Tag eine Reihe an Varianten bereit, Ein- und Ausgaben zu formatieren.

Attribut	Typ	Beschreibung
dateStyle	String	Legt das Datumsformat fest. Mögliche Werte sind default, short, medium und long. Bei keiner Angabe wird default verwendet.
parseLocale	String oder Locale	Standardmäßig wird die Locale verwendet, die im FacesContext hinterlegt ist. Bei Bedarf kann diese Einstellung überschrieben werden.
pattern	String	Für eine benutzerspezifische Formatierung kann ein eigenes Datums- bzw. Zeitformat angegeben werden. Ist dies der Fall, werden die Attribute dateStyle, timeStyle und type nicht verwendet.
timeStyle	String	Legt das Zeitformat fest. Mögliche Werte sind default, short, medium, long und full. Bei keiner Angabe wird default verwendet.
timeZone	String oder Timezone	Legt die Zeitzone fest, wie die Zeitangaben zu interpretieren sind.
type	String	Zeigt an, ob der Wert als date, time oder eine Kombination aus beidem zu interpretieren ist. Bei keiner Angabe wird date unterstellt.

Tabelle 6.8: Attribute bei convert DateTime

6.8 FacesContext

Der FacesContext nimmt in JavaServer Faces-Anwendungen eine zentrale und bedeutende Stellung ein. Über den FacesContext kann beispielsweise der Zustand eines Requests ausgelesen werden. Damit besteht die Möglichkeit, tief in die Verarbeitung eines Requests eingreifen zu können.

FacesContext selbst ist eine abstrakte Klasse. Um an eine konkrete Instanz des FacesContext zu gelangen, existiert die statische Methode.

```
FacesContext context = FacesContext.getCurrentInstance()
```

Über den FacesContext selbst hat man damit Zugriff auf eine Vielzahl weiterer Methoden.

Ebenfalls im FacesContext werden Nachrichten abgelegt, die beispielsweise während der Validierung einer Benutzereingabe erzeugt werden. Auch innerhalb einer selbst entwickelten Routine können Fehler auftreten und im FacesContext abgelegt werden, die dann als Fehler auf der Seite angezeigt werden.

```
public void addMessage( String clientId, FacesMessage message)
```

Über `addMessage` wird somit eine Nachricht, die speziell für eine Komponente bestimmt ist, im FacesContext abgelegt. Über

```
public Iterator getMessages( String clientId )
```

können natürlich auch wieder alle Nachrichten ausgelesen werden, die speziell für eine Komponente hinterlegt sind. Sollen alle Nachrichten ausgelesen werden, egal welcher Komponente sie zugeordnet sind, steht die Methode

```
public Iterator getMessages()
```

zur Verfügung. Interessant ist in diesem Zusammenhang die Methode

```
public Iterator getClientIdsWithMessages()
```

über die ein Iterator zurückgeliefert wird, der die Bezeichner aller Komponenten enthält, an denen Meldungen anstehen.

Sollte es einmal notwendig sein, auf direkte Funktionalitäten der Servlet-API zugreifen zu müssen, kann dies über den `ExternalContext` erfolgen.

```
public ExternalContext getExternalContext()
```

Damit steht gewissermaßen eine Hintertüre zur Verfügung, um auf Objekte wie z.B. `HttpServletRequest` oder `HttpServletResponse` zugreifen zu können. Im Normalfall ist solch ein Zugriff nicht notwendig. Es kann jedoch in solchen Fällen benötigt werden, in denen z.B. eine Integration mit Nicht-JSF-Anwendungen entwickelt werden soll oder aber auf Sonderfunktionalitäten zugegriffen werden soll, die JSF nicht zur Verfügung stellt.

Somit kann beispielsweise auf Parameter eines Requests mit

```
Map requestMap =
   facescontext.getExternalContext().getRequestParameterMap()
```

zugegriffen werden. In der zurückgelieferten `Map` sind die Parameter einer Anfrage enthalten und können über

```
requestMap.get( parameterName )
```

ausgelesen werden. Diese Vorgehensweise ist im Normalfall jedoch nicht notwendig. Durch das Managed-Bean-Konzept ist kein direkter Zugriff auf die Requestparameter notwendig. Es sollte sich somit auf absolute Ausnahmefälle beschränken, dass ein solcher Zugriff realisiert wird.

Wie in den folgenden Kapiteln noch näher erläutert wird, wird jede JSF-Seite intern in einer Baumstruktur abgebildet. Über den FacesContext hat man auch wieder die Möglichkeit, sich diese Baumstruktur zu holen.

```
public UIViewRoot getViewRoot
```

liefert ein Objekt der Klasse `UIViewRoot` zurück. Damit ist es unter anderem möglich, einzelne Komponenten aus dem Komponentenbaum zu entfernen oder weitere Komponenten hinzuzufügen.

6.9 Navigationskonzept

Die Navigation ist ein zentraler Bestandteil einer jeden Webanwendung. Die Navigation steuert, welche Seite nach einer Aktion – wie beispielsweise dem Klicken eines Buttons – angezeigt wird bzw. wie sich die Anwendung nach einer Aktion überhaupt verhält. Das Navigationskonzept sollte nach Möglichkeit einfach wartbar sowie schnell erweiterbar sein. In den Anfängen der JSP-Programmierung sah man häufig die Vorgehensweise, dass die Folgeseite als fester String in eine JSP-Seite codiert war. Dies funktionierte zwar, hatte jedoch den Nachteil, dass alle JSP-Seiten durchsucht werden mussten, falls an einer Stelle eine weitere Seite in einen Ablauf eingefügt werden sollte. Gerade bei komplexen Anwendungen war diese Vorgehensweise sehr fehleranfällig und aufwändig in der Wartung. JavaServer Faces verwendet für die Navigation ein Konzept, in dem fast die komplette Navigationslogik in einer Xml-Datei ausgelagert ist. »Fast« bedeutet, dass es einige Ausnahmefälle gibt, in denen Teile des Navigationskonzepts nach wie vor im Java-Quellcode vorhanden sind. Der Hauptteil der Navigation lässt sich in JSF jedoch komfortabel über eine Xml-Datei steuern. Dieses Konzept, die Navigation in eine Konfigurationsdatei auszulagern, beinhaltet mehrere Vorteile:

- erhöhte Übersichtlichkeit: Bei der Definition der Navigation in einer getrennten Datei ist keine Anwendungslogik enthalten. Damit wird die Datei überschaubarer und lesbarer.
- leichtere Wartbarkeit: Um die Navigation zu verändern, muss (im Normalfall) keine Änderung in Java-Klassen vorgenommen werden. Eine Änderung in der Konfigurationsdatei genügt, um das Ablaufverhalten der Anwendung zu verändern.
- Toolunterstützung: Bereits heute gibt es schon Werkzeuge, mit denen graphisch die Navigation erstellt und dargestellt werden kann. Ein Beispiel, die Faces-Konsole, wird im Anhang vorgestellt.

Die Navigationseinträge sind Teil der Anwendungskonfigurationsdatei *faces-config.xml*. Neben den Einträgen für Managed-Beans finden sich somit auch die Einträge für die Navigation in dieser zentralen Konfigurationsdatei wieder.

Ablauf beim Auslösen eines Kommandos

Für Kommandos, die in einer JSF-Anwendung ausgelöst werden können, existiert mit `UICommand` eine eigene Komponente. Diese kann in Form eines Commandbuttons oder eines Hyperlinks dargestellt werden. Nachdem ein Benutzer ein Kommando via Commandbutton oder Hyperlink ausgelöst hat, wird natürlich im ersten Schritt eine Datenvalidierung durchgeführt. Tritt an dieser Stelle bereits ein Fehler auf, wird wiederum auf die gleiche Seite zurückverwiesen. War die Datenvalidierung erfolgreich, gibt es zwei Ablaufmöglichkeiten: Falls keine Action-Methode an das Kommando angehängt war, wird über das `action`-Attribut der Komponente in der Navigationssteuerung nachgeschaut, welche Seite dieser Aktion zugeordnet ist. Ist dagegen eine Action-Methode für die Komponente vorhanden, wird diese entsprechend abgearbeitet. Diese Action-Methode liefert einen Rückgabewert zurück, der einen anderen Ablauf bewirken kann als der, der in der Navigationssteuerung standardmäßig vorgesehen ist.

Konzept der Navigation

Das Konzept der Navigation in JSF basiert auf so genannten Aktionen. Aktionen werden entweder durch einen Commandbutton oder einen Hyperlink ausgelöst. Für die definitive Bestimmung der Folgeseite gibt es in JSF drei Faktoren, die für das Navigationsverhalten entscheidend sind:

- die aktuelle Seite, von der aus ein Kommando ausgeführt wird,
- die Aktion selbst, die durch die Komponente aufgerufen wurde,
- und der Rückgabewert, der von einer Action-Methode zurückgeliefert wird.

Mit diesem Konzept ist es möglich, über die Anwendungskonfigurationsdatei die Navigation einer Anwendung festzulegen. Ein Ausschnitt aus der *faces-config.xml* verdeutlicht, wie der Aufbau der Definition zu erfolgen hat:

```
<navigation-rule>
  <from-view-id>/userlogin.jsp</from-view-id>
  <navigation-case>
    <from-action>#{LogonForm.logon}</from-action-ref>
    <from-outcome>success</from-outcome>
    <to-view-id>/welcome.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-action>#{LogonForm.logon}</from-action-ref>
    <from-outcome>failed</from-outcome>
    <to-view-id>/failure.jsp</to-view-id>
```

```
    </navigation-case>
  </navigation-rule>
```

Listing 6.44: Festlegung des Navigation in der faces-config.xml

In Listing 6.44 wird ein Anmeldevorgang skizziert. Es wird der Fall betrachtet, dass sich ein Benutzer über eine *userlogin.jsp*-Seite in der Anwendung anmelden möchte. Die Anmeldung besteht aus einem Formular mit zwei Eingabefeldern – für den Loginnamen und das Passwort. Nach Auslösen des Anmeldebuttons wird eine Aktion aufgerufen, die als Ergebnis *success* oder *failed* zurückliefert, je nachdem, ob die Anmeldung korrekt war oder nicht. Abhängig vom Rückgabewert wird danach auf die *welcome.jsp*-Seite verwiesen, in der der Benutzer willkommen geheißen wird, oder auf eine *failure.jsp*-Seite, die eine Meldung bzgl. der Falschanmeldung ausgibt.

Dieser Ausschnitt ist jedoch lediglich eine Konfigurationsmöglichkeit. So können Varianten definiert werden, dass unabhängig von der aktuellen Seite aufgrund eines bestimmten Rückgabewerts immer auf eine fest definierte Seite verwiesen wird. Auch ist es möglich, die Aktion, die ausgeführt wurde, zu ignorieren und die Navigation nur aufgrund von Rückgabewerten festzulegen. Auf die verschiedenen Möglichkeiten, die Navigation zu definieren, wird im Folgenden eingegangen.

Wichtig ist zu erwähnen, dass im Falle, dass keine passende Navigationsregel gefunden wurde, auf die ursprüngliche Seite wieder verwiesen wird und somit kein Wechsel auf eine neue Seite stattfindet. Der Request-Lebenszyklus wird aber auch in diesem Fall komplett durchlaufen.

Navigationsregeln

In einer Anwendungskonfigurationsdatei kann es beliebig viele Navigationsregeln (so genannte *Navigation Rules*) geben, wie in Listing 6.44 zu sehen ist. Je nach Komplexität einer Anwendung reichen manchmal einige wenige Navigationsregeln aus, es ist jedoch auch möglich, mehrere hundert Regeln festzulegen. Hierbei ist die Arbeit in einem Editor von Hand sicherlich sehr aufwändig und teilweise auch unübersichtlich. Hierzu existieren bereits Werkzeuge, mit denen die Navigation graphisch aufbereitet wird, wodurch eine leichtere Festlegung der Navigationsregeln möglich ist (vgl. Faces Konsole im Anhang). Eine Navigationsregel ist aus folgenden Tags aufgebaut:

- `from-view-id`: Es ist möglich, dass eine Aktion (z.B. eine Aktion »Hilfe anzeigen«) auf verschiedene Seiten verweist, je nachdem, von welcher Seite diese Aktion aufgerufen wurde. Im Tag from-view-id wird deshalb festgelegt, für welche Seite die folgenden Navigationselemente gelten.
- `navigation-case`: Innerhalb einer Navigationsregel kann es mehrere so genannter *Navigationsfälle* geben. In Listing 6.44 gibt es für die Login-Seite zwei Fälle, je nachdem, ob die Anmeldung erfolgreich war oder nicht.

Da es innerhalb einer *Navigationsregel* mehrere *Navigationsfälle* geben kann, wird der Aufbau eines Navigationsfalles einmal genauer betrachtet:

- `from-action`: Ein Rückgabewert eines Kommandos kann entweder direkt in der UI-Komponente integriert sein oder aber als Ergebnis einer Aktion zurückgeliefert werden. Durch die Angabe von from-action kann ein Navigationsfall speziell für eine Aktion hinterlegt werden.
- `from-outcome`: Dies ist der Rückgabewert, der entweder bereits in der Komponente hinterlegt ist oder durch eine Aktion zurückgeliefert wird.
- `to-view-id`: Hier wird festgelegt, wohin die Anwendung weitergeleitet werden soll, wenn dieser Navigationsfall eintritt.

Zusammenfassend kann gesagt werden, dass eine Navigationsregel das Verhalten steuert, wie von einer JSF-Seite auf eine andere gelangt werden kann, wobei es innerhalb einer Navigationsregel mehrere Navigationsfälle geben kann.

Rückgabewerte

Die Rückgabewerte, die von einem `UICommand` oder einer Aktion zurückgeliefert werden, steuern das weitere Verhalten der Navigation und damit der Anwendung. Die Bezeichnungen sind dabei einfache Strings und können nach Belieben definiert werden. Es empfiehlt sich jedoch, sich an einige Quasi-Standard-Bezeichnungen zu halten, da sie die Lesbarkeit und Wartbarkeit von Anwendungen vereinfachen.

Bezeichnung	**Kontext**
`success`	wird als Ergebnis einer erfolgreich durchgeführten Aktion zurückgeliefert.
`error`	Es kam zu einem Fehler. Häufig wird daraufhin an eine Fehlerseite weitergeleitet.
`logon`	Falls in einer Webanwendung ein Logon erforderlich ist, verweist diese Bezeichnung auf die Login-Seite.
`no_results`	Falls eine eindeutige Zuordnung zu einer Navigationsregel nicht gefunden wurde, gehe zu einer Suchen- oder Indexseite.

Tabelle 6.9: Empfohlene Bezeichner

Wildcards

In den bisherigen Beispielen war eine Navigationsregel immer so aufgebaut, dass von einer eindeutigen Seite je nach Rückgabewert auf eine Folgeseite verwiesen wurde. Häufig macht es jedoch Sinn, gewisse Rückgabewerte anwendungsübergreifend zu definieren. So kann es z. B. möglich sein, von jedem Punkt der Anwendung auf eine zentrale Hilfeseite springen zu können. Hierbei für jede einzelne JSF-Seite einen eige-

nen Navigationsfall aufzubauen wäre dafür extrem viel Aufwand. Aus diesem Grund ist es auch bei der Definition von Navigationsregeln möglich, Platzhalter (*Wildcards*) zu verwenden.

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>help</from-outcome>
    <to-view-id>/help.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 6.45: Navigationsregel mit einer Wildcard

In Listing 6.45 wird ein Rückgabewert `help` für sämtliche JSF-Seiten festgelegt. Somit kann von jeder Stelle im Anwendungsablauf auf die Hilfeseite verwiesen werden. Das `from-view-id`-Tag wurde dazu einfach mit einem »*« ausgefüllt. Es ist auch möglich, ein Verzeichnis in Zusammenhang mit einem Platzhalter zu verwenden:

```
<navigation-rule>
  <from-view-id>/loginprocess/*</from-view-id>
  <navigation-case>
    <from-outcome>help</from-outcome>
    <to-view-id>/loginhelp.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/application/*</from-view-id>
  <navigation-case>
    <from-outcome>help</from-outcome>
    <to-view-id>/applicationhelp.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 6.46: Navigationskonzept mit Wildcards

In Listing 6.46 sind zwei Navigationsregeln festgehalten, in denen in beiden Fällen der Rückgabewerte `help` verwendet wird. Diese haben ihren jeweiligen Geltungsbereich, wenn der Rückgabewert von einer Seite unterhalb von */loginprocess* bzw. unterhalb von */application* zurückgeliefert wurde.

Vorgabe eines Standardverhaltens

Die Navigationsregeln überlassen einem Anwendungsentwickler einen großen Spielraum, die Navigation sehr detailliert zu hinterlegen. So gibt es neben den Wildcards, die mittels eines Musters einzelne Navigationsfälle behandeln, auch die Möglichkeit, ein so genanntes Standardverhalten anzugeben.

```
<navigation-rule>
  <from-view-id>eingabe.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>ausgabe.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <to-view-id>information.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 6.47: Festlegen eines Standardverhaltens

Listing 6.47 zeigt ein Beispiel, in dem basierend auf einer Eingabeseite ein Rückgabewert `success` abgefragt und daraufhin auf eine Ausgabeseite verwiesen wird. Im zweiten Navigationsfall wird kein `from-view-id`-Tag angegeben. Dies bedeutet, dass bei jedem Rückgabewert mit Ausnahme von `success` auf eine Seite *information.jsp* navigiert wird.

Während die Angabe eines `from-view-id`-Tags optional ist, ist bei der Verwendung dieses Tags ein leerer Inhalt nicht erlaubt. Folgendes Beispiel läuft daher auf einen Fehler hinaus:

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

Listing 6.48: Fehlerhafte Navigationsregel

Funktionsweise der Navigation

Für ein tiefer gehendes Verständnis des Navigationskonzepts ist es wichtig zu verstehen, wie genau die Bestimmung der Navigation verläuft. In JSF ist eine Seite nicht einfach eine flache Datei, sondern wird als Komponentenbaum intern dargestellt (der so genannte *Component Tree*). Sämtliche Komponenten, die auf einer JSF-Seite vorhanden sind, sind in hierarchischer Form im Komponentenbaum angeordnet. Ein Komponentenbaum ist somit die serverseitige Darstellung einer JSF-Seite. Erfolgt ausgehend von einer Faces-Seite ein Request an den Server, wird in der Phase *Restore View* zunächst der Komponentenbaum wiederhergestellt.

Auf einer Seite selbst sind alle UICommand-Komponenten automatisch an einen Default-ActionListener gebunden. Sobald eine Aktion ausgelöst wird (durch Drücken eines Commandbuttons oder Anklicken eines Hyperlinks) wird ein Event ausgelöst.

Dieses Event wird in der Phase *Invoke Application* des Lebenszyklus einer Seite abgearbeitet. Im ActionListener selbst wird der Rückgabewerte einer Komponente mitgegeben. Dieser ist entweder direkt über das `action`-Attribut in der Komponente selbst hinterlegt oder wird als Rückgabewert eines Aktionsaufrufes zurückgeliefert. Im letzteren Fall wurde bei der Komponente eine Aktion angegeben, die bei Auslösen der Komponente eine entsprechende Methode mit einen String-Rückgabewert ausführt. Diese Methode liefert dann wiederum einen Rückgabewert, der ebenfalls wieder im Default-ActionListener mit übergeben wird. Im ActionListener wiederum wird der Rückgabewert an den so genannten *NavigationHandler* weitergereicht, der aufgrund des Rückgabewerts, der aufgerufenen Aktion und der aktuellen Seite mit Hilfe der Eintragungen in der Anwendungskonfigurationsdatei die Folgeseite bestimmt.

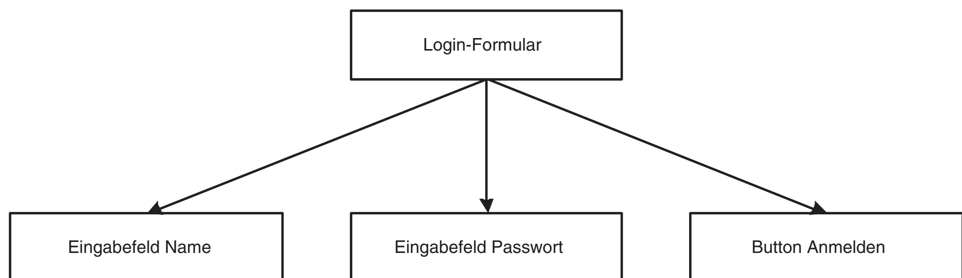


Abbildung 6.10: Komponentenbaum

Reihenfolge der Abarbeitung

Im vorherigen Abschnitt wurde bereits erläutert, dass es innerhalb einer Navigationsregel mehrere Navigationsfälle geben kann. Diese bestimmen je nach Aktion und Rückgabewert, auf welche Seite im weiteren Programmablauf weitergeleitet wird. Aufgrund der Tatsache, dass sowohl das `from-outcome`-Tag als auch das `from-action`-Tag optional angegeben werden können, kann es vorkommen, dass sich einzelne Navigationsfälle teilweise auch einmal überlappen. JSF arbeitet in diesen Fällen die einzelnen Einträge in folgender Reihenfolge ab:

- Zuerst wird ein passender Navigationsfall gesucht, bei dem sowohl die Aktion als auch der Rückgabewert übereinstimmen.
- Danach wird nur der Rückgabewert überprüft, ob es hierfür einen Anwendungsfall gibt, der die weitere Navigation regelt.
- Als Letztes wird das `from-action`-Tag überprüft, ob es eventuell hierfür eine Entsprechung gibt.

Wurde überhaupt keine Entsprechung gefunden, wird wiederum auf die gleiche Seite verwiesen, von der aus ein Kommando aufgerufen wurde. Ein expliziter Fehler wird vom Framework nicht geworfen. Wurde jedoch eine Entsprechung gefunden, wird daraufhin der Komponentenbaum für diese Seite aufgebaut und gerendert.

6.9.1 Statische versus dynamische Navigation

Im letzten Abschnitt wurde das Konzept der Navigation ausführlich vorgestellt und beschrieben. Im Folgenden wird nochmals detailliert betrachtet, wie eine Navigation in der Anwendung konkret stattfindet, d.h. welche Möglichkeiten es gibt, einen Navigationswechsel aufzurufen. Für das Auslösen einer Aktion stehen in JSF zwei Möglichkeiten zur Verfügung. Zum einen kann eine Aktion direkt durch Angabe des `action`-Attributs einer UICommand-Komponente angegeben werden, zum anderen kann auch in der UICommand-Komponente ein `action`-Attribut verwendet werden, das eine Methode aufruft, deren Ergebnis wiederum die Navigation steuert.

Wie im Beispiel bereits gesehen, kann beispielsweise in einem `commandButton`-Tag ein `action`-Attribut mitgegeben werden. Anhand dieses Eintrags wird in der Navigationsdefinition nachgeschaut, auf welche Seite weitergeleitet werden soll. Da in diesem Fall genau festgelegt ist, dass bei Auslösen eines Kommandos auf eine fest definierte Seite weitergeleitet wird, spricht man von *statischer Navigation*.

Oftmals ist es jedoch nicht möglich, die Folgeseite bereits im Voraus zu kennen. Dies ist z.B. bei einer Anmeldung in einer Anwendung der Fall. Abhängig davon, ob der Loginname und das Passwort korrekt waren, wird auf eine *Fehlerseite* oder eine *Willkommensseite* verwiesen. Für diesen Fall kann natürlich kein festes `action`-Attribut angegeben werden. JSF bietet hier die Möglichkeit, über einen Methodenaufruf, der aufgrund einer Programmlogik einen entsprechenden Rückgabewert liefert, das weitere Verhalten der Navigation zu bestimmen. Da in diesem Fall die Navigation nicht fest hinterlegt ist, sondern durch den Programmablauf bestimmt wird, spricht man hier von *dynamischer Navigation*.

Das action-Attribut

Die UICommand-Komponente kann auf zwei unterschiedliche Arten dargestellt werden: zum einen als Hyperlink, dessen Anklicken eine bestimmte Aktion anstößt, zum anderen als Commandbutton, der ebenfalls durch Anklicken ausgelöst werden kann. Die genaue Verwendung der UI-Komponenten wird im Kapitel 7 ausführlich erläutert, an dieser Stelle greifen wir jedoch ein klein wenig vor, um die Navigation näher zu erläutern.

```
<h:commandButton value="Hilfe" action="help">
```

Listing 6.49: Verwendung eines Commandbuttons

In Listing 6.49 wird ein Commandbutton definiert, der die Bezeichnung *Hilfe* trägt. Sobald dieser Button ausgelöst wird, wird in der Anwendungskonfigurationsdatei gesucht, ob für die aktuelle Seite und den Rückgabewert `help` ein Navigationsfall hinterlegt ist. Wird ein passender Navigationsfall gefunden, so wird auf die angegebene Seite weitergeleitet. Das Attribut `action` bezieht sich somit direkt auf das `from-outcome`-Tag in der Navigation.

```
<h:commandLink action="help">
  <h:outputText value="Hilfe" />
</h:commandLink>
```

Listing 6.50: Verwendung eines Hyperlinks

Bei Verwendung der Darstellungsform eines UICommands als Hyperlink gilt das gleiche Prinzip wie bei einem Commandbutton. Bei Aktivierung des Hyperlinks wird das `action`-Attribut ausgewertet und ein passender Navigationsfall dafür gesucht.

Methodenaufrufe beim action-Attribut

Anstatt durch das `action`-Attribut direkt auf eine Folgeseite zu verweisen, besteht durch Angabe eines Methodennamens die Möglichkeit, eine Methode innerhalb einer Beanklasse direkt aufzurufen und das Ergebnis der Methode für die weitere Navigation zu verwenden. Diese Möglichkeit kommt meist dann zum Einsatz, wenn eine gewisse Programmlogik notwendig ist, um nach Auslösen eines Kommandos die Folgeseite zu bestimmen. Als Beispiel kann hier eine Anmeldung in einer Web-Community angesehen werden; nach dem Betätigen des Login-Buttons muss zuerst geprüft werden, ob die Anmeldung korrekt war oder nicht. Danach wird entweder auf eine Willkommensseite oder aber auf eine Fehlerseite verwiesen.

```
public String doLogin {
  ...
}
```

Listing 6.51: Die Action-Methode

In Listing 6.51 ist ein Beispiel für eine `Action`-Methode abgebildet.

Der Rückgabewert der Action-Methode bezieht sich wiederum auf die `from-outcome`-Angabe in der Anwendungskonfigurationsdatei. Damit eine Methode aus einer Command-Komponente heraus aufgerufen werden kann, muss sie

- öffentlich sein (`public`)
- einen String als Rückgabewert liefern
- und keine Parameter erwarten.

Der Rückgabewert dieser Methode wird dann wiederum gegen das `from-output`-Tag in den Navigationsregeln geprüft, worauf die Folgeseite dann bestimmt werden kann.

Achtung!
Oftmals wird der Fehler gemacht, dass versucht wird, beliebige Methoden einer beliebigen Klasse aufzurufen. Dies ist jedoch nicht möglich. Es können nur Methoden aufgerufen werden, die obige Kriterien erfüllen. Zudem können auch nur Methoden von *Backing Beans* aufgerufen werden, die in der Anwendungskonfigurationsdatei hinterlegt sind, also nicht Methoden in irgendwelchen beliebigen Klassen.

```
package com.edu.jsf.bsp.bean;

 /**
 * Bean, das für ein korrektes Login verantwortlich ist
 */
public class LoginBean {

  private String loginname;
  private String password;

  /**
   * Getter-Methoden
   */
  public String getLoginname() {
    return loginname;
  }

  public String getPassword() {
    return password;
  }

  /**
   * Setter-Methoden
   */
  public void setLoginname(String string) {
    loginname = string;
  }

  public void setPassword(String string) {
    password = string;
  }

  /**
   * Überprüft einen Login und liefert das entsprechende
   * Ergebnis zurück.
   */
  public String doLogin() {
    if ( loginname==null || password==null )
      return "failed";
```

```
        if ( loginname.equals("master")
             && password.equals("secret") ) {
          return "success";
        } else {
          return "failed";
        } // else
      }

    }
```

Listing 6.52: Ein LoginBean mit Action-Klasse

In Listing 6.52 ist eine Aktion `doLogin` vorhanden, die nach Betätigen des Anmeldeknopfes ausgeführt werden soll. Darin wird überprüft, ob die Anmeldeinformationen korrekt waren. Wenn ja, wird ein entsprechender Rückgabewert zurückgeliefert. Die Rückgabewerte der Methode sind in der Anwendungskonfigurationsdatei hinterlegt (siehe Listing 6.53).

```
<navigation-rule>
  <from-view-id>/kap_6/userlogin.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/kap_6/userlogin_ok.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failed</from-outcome>
    <to-view-id>/kap_6/userlogin_failed.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 6.53: Navigationsfälle für einen Login

Auf der Anmeldeseite selbst wird die Action-Methode durch einen Ausdruck der JSF EL (*Expression Language*) angegeben:

```
<h:commandButton action="#{Demologin.doLogin}" value="Anmelden" />
```

Wichtig ist auch hier wieder die korrekte Syntax. Wird die Umklammerung durch `#{` und `}` weggelassen, wird der Wert als Literal interpretiert und unter diesem Schlüssel direkt nach einer Navigationsregel gesucht. Da dies hier jedoch nicht gewollt ist, sondern vielmehr die Angabe so zu interpretieren ist, dass eine Methode ausgeführt werden soll, ist der Aufruf als Ausdruck der Expression Language anzugeben.

6.10 Eventhandling

JavaServer Faces wird oftmals auch als »Swing auf dem Server« bezeichnet. Damit ist gemeint, dass das Verhalten sowie viele Konzepte, die Swing beinhaltet, in JSF wiederzufinden sind. Auch beim Thema Eventhandling trifft dies zu. Es ist in JSF möglich, an

Komponenten so genannte *Eventlistener* zu registrieren, die bei Eintreten definierter Events aufgerufen werden. Die Ähnlichkeit der Konzepte hinsichtlich des Eventhandling macht es gerade für Einsteiger in JavaServer Faces einfach, sich in dieser neuen Technologie zurechtzufinden.

Dieses Vorgehen von Sun macht durchaus Sinn; erprobte und für gut befundene Konzepte sollen weiterentwickelt und regelmäßig eingesetzt werden. Gerade in Zeiten, in denen das Thema »Design Patterns« in aller Munde ist, werden auch in JavaServer Faces bekannte und erprobte Konzepte wiederverwendet.

6.10.1 Eventverarbeitung

In JSF existieren Event- und Listener-Klassen. Ein Eventobjekt beinhaltet dabei zum einen die Komponente selbst, die das Event erzeugt hat, sowie weitere zusätzliche Informationen zum Event. Ein Applikationsentwickler hat mit Hilfe der Listener-Klassen die Möglichkeit, sich an eine Komponente anzuhängen. Sobald dann ein Event erzeugt wird, werden alle registrierten Listener-Objekte benachrichtigt.

Eventtypen

JSF kennt drei Arten von Events: Action-Events, ValueChange-Events und Data-Model-Events. Action-Events werden bei Eintreten von bestimmten Aktionen erzeugt, beispielsweise beim Auslösen eines Commandbuttons oder bei Aktivierung eines Hyperlinks. ValueChange-Events werden erzeugt, sobald sich ein Wert einer Komponente ändert. Die ist z.B. dann der Fall, wenn ein Benutzer in einem Textfeld einen neuen Wert eingibt und die Werte an den Server abschickt. Data-Model-Events sind weniger häufig, sie werden bei einer Auswahl einer neuen Reihe in einer UIData-Komponente ausgelöst.

Damit eine Anwendung auf einen Event reagieren kann, sind folgende Schritte im Vorfeld notwendig:

- Erzeugen einer Listenerklasse, die das entsprechende Interface implementiert. In dieser Klasse findet dann die Abarbeitung des Events statt.
- Registrierung des Listeners an der Komponente. Damit wird bei Eintreten eines Events der entsprechende Listener aufgerufen.

Dieses Vorgehen trifft prinzipiell erst einmal für die Standardkomponenten zu, die in JSF bereits enthalten sind. Bei benutzerdefinierten Komponenten ist das Vorgehen ein wenig anders. Mehr dazu finden Sie im Kapitel 9. JSF erweitern und anpassen.

Zusätzlich gibt es auch noch die Möglichkeit, anstatt einer eigenen Listener-Klasse über die Möglichkeit des *Method Bindings* eine Methode einer Klasse direkt auszuführen. Dieser Weg wird ebenfalls im Folgenden beschrieben.

Event- und Listenerklassen

Action-Events werden durch Auslösen einer UICommand-Komponente ausgelöst. Durch Aktivierung eines Hyperlinks oder Klicken eines Commandbuttons wird ein Objekt der Klasse `javax.faces.event.ActionEvent` erzeugt. Die Listener, die auf ein Action-Event hören, implementieren das Interface `javax.faces.event.ActionListener`, das wiederum von `javax.faces.event.FacesListener` abgeleitet ist.

Ein Value-Change-Event tritt in Zusammenhang mit Komponenten auf, die Subklassen der UIInput-Komponente sind sowie zur Klasse `UIInput` selbst gehören – beispielsweise `UISelectBoolean` oder `UISelectOne`. Graphisch werden diese Komponenten als Texteingabefelder, Checkboxen oder Listboxen dargestellt. Grundsätzlich kann an jede Eingabekomponente ein Value-Change-Listener angehängt werden. Die Events gehören dabei zur Klasse `javax.faces.event.ValueChangeEvent`, die Listener implementieren das Interface `javax.faces.event.ValueChangeListener`, das auch wiederum von `javax.faces.event.FacesListener` abgeleitet ist.

6.10.2 Action-Events

Action-Events werden von UICommand-Komponenten ausgelöst. Dies kann somit ein Commandbutton oder auch ein Hyperlink sein. Sobald das Kommando durch Drücken eines Commandbuttons oder durch Anklicken eines Hyperlinks ausgelöst wird, wird ein entsprechendes Action-Event erzeugt und an registrierte Listener übergeben.

Implementieren eines ActionListeners

Eine ActionListener-Klasse implementiert das Interface `javax.faces.event.ActionListener`, das wiederum vom Interface `javax.faces.event.FacesListener` abgeleitet ist. Das Interface `ActionListener` defniert genau eine Methode, die zu implementieren ist:

```
public void processAction( ActionEvent event )
```

Diese wird aufgerufen, sobald ein Listener registriert ist und ein Event erzeugt wurde.

```
package com.edu.jsf.bsp.listener;

import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

/**
 * Implementierung eines ActionListeners.
 */
public class DemoListener implements ActionListener {

  /**
```

```
     * wird bei Auslösen eines Events aufgerufen
     */
    public void processAction( ActionEvent event )
        throws AbortProcessingException {
      System.out.println("In processAction");
    }

}
```

Listing 6.54: Ein ActionListener

In Listing 6.54 ist ein Beispiel für einen ActionListener zu sehen. In Abbildung 6.11 ist eine dazu passende Beispielseite abgebildet. Dabei wird ein ActionEvent sowohl über einen Button als auch über einen Hyperlink ausgelöst. Beim Auslösen wird in der `processAction`-Methode lediglich eine Meldung in der Konsole ausgegeben, damit der Benutzer weiß, dass die Methode auch tatsächlich aufgerufen wurde.

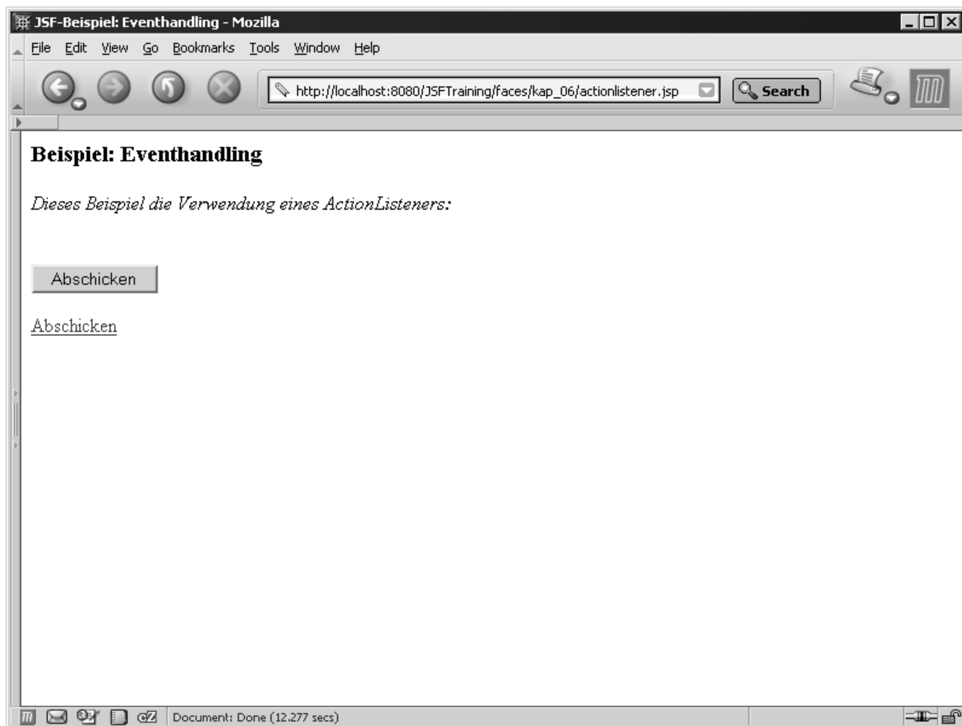


Abbildung 6.11: Beispiel zur Verwendung eines ActionListeners

Registrieren des ActionListeners an einer Komponente

Das Registrieren eines ActionListeners an einer Komponente erfolgt lediglich durch ein weiteres Tag, das innerhalb eines `commandButton`-Tags oder innerhalb eines Hyperlink-Tags als verschachteltes Element eingesetzt wird.

```
<h:commandButton action="success" value="Abschicken">
  <f:actionListener
     type="com.edu.jsf.bsp.listener.DemoListener" />
</h:commandButton>
```

Listing 6.55: Registrieren eines ActionListeners an einem Commandbutton

Wichtig ist bei der Angabe des Attributes `type`, dass der vollständige Klassenname angegeben wird.

```
<h:commandLink action="success">
  <h:outputZext value="Abschicken" />
  <f:actionListener
     type="com.edu.jsf.bsp.listener.DemoListener" />
</h:commandLink>
```

Listing 6.56: Registrieren eines ActionListeners an einem Hyperlink

Bei einem Hyperlink wie in Listing 6.56 ist die Vorgehensweise analog zu der bei einem Commandbutton.

Architekturempfehlung

Häufig wird in Foren, die sich mit JavaServer Faces beschäftigen, die Frage diskutiert, an welcher Stelle Programmlogik zu stehen hat, die für die Navigation verantwortlich ist. Als Beispiel sei eine Benutzeranmeldung verwendet, die nach Eingabe eines Namens und eines Passwortes nach Auslösen des Anmeldebuttons die Anmeldedaten überprüft. Je nach Ergebnis der Prüfung wird dann auf eine Willkommensseite oder auf eine Fehlerseite verwiesen.

Prinzipiell existieren zwei Stellen, an denen diese Navigationslogik untergebracht werden kann. Zum einen kann ein ActionListener an den Anmeldebutton angehängt werden. Nach Auslösen können die Eingaben überprüft und die weitere Navigation festgelegt werden. Dieses Verfahren ist jedoch nicht zu empfehlen. Zum einen ist es nicht so einfach, die Navigation auf diese Weise zu beeinflussen (es müssen dafür schon ein paar Zeilen Code geschrieben werden), zum anderen sollte in Listenern auch keine Navigationslogik enthalten sein. Es wird daher empfohlen, mittels des `action`-Attributes beim Anmeldebutton eine entsprechende Action-Methode aufzurufen, die die entsprechenden Überprüfungen übernimmt und einen passenden Rückgabewert zurückliefert.

ActionListener bei Backing Beans

Statt eine eigene separate Listenerklasse zu implementieren, bietet JSF die Möglichkeit, direkt aus einer Komponente heraus einen Methodenaufruf durchzuführen. Dabei wird mit Hilfe der JSF EL (*Expression Language*) der Methodenname als Attribut z.B. in einem `commandButton`-Tag mit angegeben. Die Methode, die dabei aufgerufen wird, muss

- öffentlich (`public`) sein
- einen Rückgabewert `void` haben
- als Argument ein `ActionEvent` entgegennehmen.

```
public void changeColor( ActionEvent event ) {
   ...
}
```

Listing 6.57: Beispiel einer Actionmethode

Listing 6.57 zeigt einen Aufruf einer Methode innerhalb eines Backing Beans. In einer Faces-Seite sieht das dann wie folgt aus:

```
<h:commandLink action="settings"
   actionListener="#{Settings.changeColor}">
```

Das `actionListener`-Attribut kann jedoch nur bei UICommand-Komponenten verwendet werden, die das Interface `javax.faces.component.ActionSource` implementieren. Im Beispiel wird im Backing Beans `Settings` nach der Methode `changeColor` gesucht und diese ausgeführt. Das weitere Navigationsverhalten wird dabei nicht beeinflusst, dieses ist durch die feste Angabe des `action`-Attributs bereits festgelegt.

6.10.3 Value-Change-Events

Value-Change-Events werden von allen Unterklassen von UIInput erzeugt. Der Standardfall ist der, dass ein Benutzer in einem Textfeld einen neuen Wert einträgt, worauf ein entsprechendes Event ausgelöst wird. Da jedoch die komplette Verarbeitung auf dem Server erfolgt, wird ein solches Event natürlich auch erst nach dem Abschicken eines Formulars an den Server erzeugt. Hier ist verständlicherweise die Verarbeitung bei einer Weboberfläche ein wenig anders als die Eventverarbeitung z.B. bei einer Swing-Applikation. Während bei letzterer direkt nach dem Verlassen eines Textfeldes ein Event erzeugt werden kann, ist dies bei JSF in dieser Form nicht vorgesehen. Technisch könnte dies zwar mittels JavaScript-Funktionen auch realisiert werden, dies ist standardmäßig jedoch nicht umgesetzt. Vielmehr wird nach dem Absenden eines Formulars an den Server im Framework überprüft, ob sich ein Wert geändert hat und ein entsprechender Listener darauf registriert ist.

Implementieren eines ValueChangeListeners

Analog zur Implementierung eines ActionListeners wird bei einem ValueChangeListener das Interface `javax.faces.event.ValueChangeListener` implementiert, das ebenfalls wieder von `javax.faces.event.FacesListener` abgeleitet ist. Auch im `ValueChangeListener` muss eine Methode `processValueChange` implementiert werden.

```
package com.edu.jsf.bsp.listener;

import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;

/*
 * Implementierung eines ValueChangeListeners
 */
public class DemoValueChangeListener implements
       ValueChangeListener {

  /*
   * Wird bei Auslösen eines Events aufgerufen.
   */
  public void processValueChange( ValueChangeEvent vEvent) {
    System.out.println("In processValueChange");
    System.out.println("Old value: " + vEvent.getOldValue() );
    System.out.println("New value: " + vEvent.getNewValue() );
  }
}
```

Listing 6.58: Beispiel eines ValueChangeListeners

Wichtig ist bei einer Wertveränderung natürlich, was sich genau verändert hat. Dazu kann über das `ValueChangeEvent` abgefragt werden, wie der ursprüngliche und wie der neue Wert ist. Durch die Methode

```
event.getOldValue()
```

wird ein Objekt der Klasse `java.lang.Object` zurückgeliefert, das gegebenenfalls in den eigentlichen Datentyp gecastet werden kann. Ebenso wird über

```
event.getNewValue()
```

das neue Objekt zurückgeliefert, das ebenso in den gewünschten Typ gecastet werden kann.

Registrieren des ValueChangeListeners an der Komponente

Auch bei einem `ValueChangeListener` genügt es, durch ein entsprechendes Tag den Listener an einer Komponente zu registrieren. Es können natürlich nur `UIInput`-Komponenten und deren Unterklassen einen `ValueChangeEvent` auslösen.

```
<h:inputText value="#{Customer.lastname}">
  <f:valueChangeListener
     type="com.edu.jsf.bsp.listener.DemoValueChangeListener" /
</h:inputText>
```

Listing 6.59: Registrieren eines ValueChangeListeners

In Listing 6.59 wird für ein Eingabefeld ein `ValueChangeListener` registriert. Wird durch einen Benutzer ein neuer Wert in das Eingabefeld eingetragen und das Formular abgeschickt, wird ein entsprechendes Event erzeugt und der Listener aufgerufen.

Abfrage weiterer Eventinformationen

Gerade bei ValueChangeEvents ist es oftmals wichtig, beide Werte, also den ursprünglichen und den neu eingegebenen, zu kennen. Nur dann kann entschieden werden, ob eventuell ein Fehler geworfen werden soll oder eine Verarbeitung angestoßen werden kann. Die Klasse `ValueChangeEvent` stellt daher speziell zwei Methoden zu diesem Zweck zur Verfügung.

- `getNewValue()`: Liefert den neuen Wert der Komponente zurück.
- `getOldValue()`: Liefert den ursprünglichen Wert der Komponente zurück.

Die Rückgabewerte sind in beiden Fällen vom Typ `java.lang.Object`. Bei Bedarf müssen die Rückgabewerte somit in die passende Klasse gecastet (umgewandelt) werden. Hierbei muss jedoch darauf geachtet werden, dass der Rückgabewert der Methoden auch `null` sein kann.

ValueChangeListener bei Backing-Beans

Auch bei `ValueChange`-Events ist es möglich, statt einer separaten Listenerklasse mit Hilfe der Expression Language einen direkten Methodenaufruf per Attribut durchzuführen.

```
<h:inputText value="#{Customer.lastname}"
   valueChangeListener="#{Customer.lastnameChange}" />
```

Listing 6.60: ValueChangeListner und Backing-Beans

In Listing 6.60 wird mit Hilfe des Attributs `valueChangeListener` direkt die Methode `lastnameChange` bei einer Änderung des Eingabewertes aufgerufen. Dabei muss in der entsprechenden Klasse die Methode `lastnameChange` vorliegen, die auch wieder öffentlich (`public`) ist, einen Rückgabewert `void` aufweist und als Argument ein `ValueChange Event` entgegennimmt.

6.10.4 Phase Events

Phase Events werden am Beginn und am Ende der Verarbeitung eines Requests geworfen. Ein einkommender Request wird gemäß des Request-Lebenszyklus (vgl. Kapitel 4.14 Lebenszyklus einer JavaServer Faces Seite) abgearbeitet. Dabei wird sowohl am Anfang als auch am Ende jeder Phase ein Event erzeugt, das mittels entsprechender Listener abgefangen werden kann. Das *Phase Event* selbst entstammt der Klasse `javax.faces.event.PhaseEvent` und weist zwei Methoden auf, mit denen Informationen über das Event abgefragt werden können.

```
getFacesContext
```

liefert ein Objekt vom Typ `FacesContext` zurück, das den Kontext beinhaltet, der im zugrunde liegenden Request verwendet wurde. Über

```
getPhaseId
```

erhält man die Phase, zu der ein bestimmtes *Phase Event* erzeugt wurde. Mögliche Rückgabewerte sind:

- `PhaseId.RESTORE_VIEW`
- `PhaseId.APPLY_REQUEST_VALUES`
- `PhaseId.PROCESS_VALIDATIONS`
- `PhaseId.UPDATE_MODEL_VALUES`
- `PhaseId.INVOKE_APPLICATION`
- `PhaseId.RENDER_RESPONSE`

Anhand dieser Rückgabewerte lässt sich ermitteln, in welcher Phase des Request-Lebenszyklus das Event geworfen wurde.

In der entsprechenden Listenerklasse `javax.faces.event.PhaseListener` liegen zwei Methoden vor, die entweder vor Beginn der Phase oder nach Durchlaufen der Phase aufgerufen werden. Des Weiteren kann in der Listenerklasse festgelegt werden, in welcher Phase der Listener benachrichtigt werden soll.

```
package com.edu.jsf.bsp.listener;

import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

/*
 * Listenerklasse, die PhaseEvents verarbeitet
 */
```

```
public class MyLifecycleListener implements PhaseListener {

  /*
   * wird vor Beginn einer Phase aufgerufen
   */
  public void beforePhase(PhaseEvent event) {
    System.out.println("********* beforePhase  ");
  }

  /*
   * wird nach Beginn einer Phase aufgerufen
   */
  public void afterPhase(PhaseEvent event) {
    System.out.println("********* afterPhase  ");
  }

  /*
   * zeigt an, für welche Phase der Listener registriert ist.
   */
  public PhaseId getPhaseId() {
    return javax.faces.event.PhaseId.ANY_PHASE;
  }
}
```

Listing 6.61: PhaseListener-Klasse

Listing 6.61 zeigt eine entsprechende Listenerklasse. In der Methode `getPhaseId` wird angegeben, für welche Phase der Listener registriert ist. Die Angabe `PhaseId.ANY_PHASE` bewirkt, dass vor Beginn und nach Durchlaufen jeder Phase der Listener benachrichtigt wird. Dabei werden die Methoden `beforePhase` und `afterPhase` entsprechend aufgerufen.

Um den Listener zu registrieren, kann in der Anwendungskonfigurationsdatei ein Eintrag vorgenommen werden. Damit wird der Listener bei allen einkommenden Requests entsprechend benachrichtigt.

```
<lifecycle>
  <phase-listener>
    com.edu.jsf.bsp.listener.MyLifecycleListener
  </phase-listener>
</lifecycle>
```

Listing 6.62: Registrieren eines PhaseListeners

Sinnvoll ist ein Einsatz eines PhaseListeners z.B. dann, wenn vor Verarbeitung einer Faces-Seite Initialisierungen durchgeführt werden müssen. So kann es vorkommen, dass eine Faces-Seite, die direkt von einem Anwender angesprochen wird, noch nicht mit den notwendigen Beans initialisiert wurde und daher zunächst eine entsprechende Routine aufgerufen werden muss.

6.11 Zustandsspeicherung

Zwischen einem Webbrowser und einem Applikationsserver besteht keine permanente Verbindung. Das zugrunde liegende HTTP-Protokoll ist so aufgebaut, dass für die Phase einer Anfrage (eines Requests) Daten zwischen Server und Browser ausgetauscht werden und danach die Verbindung wieder getrennt wird. Die Kommunikation besteht daher aus einer Vielzahl von Browseranfragen (*Request*) und Serverantworten (*Response*).

Aufgrund der Tatsache, dass zwischen einzelnen Anfragen Zustände von Komponenten gespeichert werden müssen, ist das Thema der *Zustandsspeicherung* ein sehr elementares Thema im Umfeld von Webframeworks.

Eine Zustandsspeicherung (oft auch besser bekannt unter der englischen Bezeichnung *State Saving*) ist dafür verantwortlich, sämtliche relevanten Informationen über eine View zwischen einzelnen Anfragen zu speichern. Speziell bei JavaServer Faces wird durch den Mechanismus des State Saving jede Einstellung einer View und aller darin enthaltenen Komponenten gespeichert. Einstellungen können dabei nicht nur visuelle Einstellungen sein, z. B. ob eine Komponente aktuell sichtbar ist oder nicht, sondern auch angehängte Listener z. B. bei Kommandokomponenten oder Konverter und Validatoren.

JavaServer Faces unterstützt zwei Möglichkeiten, den Zustand einer View zu speichern: *Client Site Saving* und *Server Site Saving*.

Bei *Client Site Saving* werden sämtliche Zustandsinformationen clientseitig versteckt. Dabei kommen so genannte versteckte Felder (*Hidden Fields*) zum Einsatz. Diese sind für den Benutzer nicht sichtbar, sind jedoch in einer im Browser dargestellten Seite enthalten. Sobald ein Request an den Server geschickt wird, sind damit automatisch alle notwendigen Informationen im Request mit enthalten.

Bei *Server State Saving* werden sämtliche Informationen auf Serverseite vorgehalten. Dies geschieht unter Verwendung von Sessions. Diese werden im Speicher des Servers bereitgehalten und stehen jedem Benutzer zur Verfügung. Erfolgt eine Anfrage eines Benutzers bzw. eines Browsers an den Server, wird dort die zugewiesene Session gesucht und auf die darin gespeicherten Informationen zurückgegriffen. Die Zuordnung zwischen der Session im Server und einem Browser bzw. einem Benutzer erfolgt dabei meist über so genannte Cookies. Dies sind kleine Kennungen bzw. textuelle Informationen, die im Browser des Benutzers gespeichert werden und somit bei einer (erneuten) Anfrage an den Server den Benutzer identifizieren.

Die Einstellung, ob Zustandsinformationen serverseitig oder clientseitig abgespeichert werden sollen, wird über den Deployment Deskriptor vorgenommen.

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

Listing 6.63: Client site saving

Entsprechend dem Listing 6.63 kann als Wert `client` oder `server` hinterlegt werden. Standardmäßig wird eine serverseitige Speicherung vorgenommen.

Die Standardkomponenten in JavaServer Faces stellen bereits die notwendige Funktionalität bereit, um ihren Zustand abzuspeichern und auch wiederherzustellen. Dazu existiert das Interface `javax.faces.component.StateHolder`, das von Komponentenklassen implementiert wird, die ihren Zustand abspeichern.

Um Werte abzuspeichern, wird eine Methode `saveState` verwendet, die durch das Interface vorgegeben ist:

```
public Object saveState( FacesContext context ) {
  removeDefaultActionListener(context)
  Object values[] = new Object[2];
  values[0] = super.saveState(context);
  values[1] = visible ? Boolean.TRUE : Boolean.FALSE;
  addDefaultActionListener(context);
  return (values);
}
```

Wichtig dabei ist, dass zunächst der DefaultActionListener entfernt wird, damit dieser nicht mit abgespeichert wird. Danach wird ein Array definiert, in dem zunächst die Superklasse aufgerufen und danach die lokale Eigenschaft `visible` abgefragt wird.

Im Anschluss kann der DefaultActionListener wieder angehängt und das Werte-Array zurückgeliefert werden.

Zum Wiederherstellen des Zustandes wird eine Methode `restoreState` verwendet, die ebenfalls im Interface `StateHolder` definiert ist.

```
public void restoreState(FacesContext context, Object state) {
  removeDefaultActionListener(context);
  Object values[] = (Object[]) state;
  super.restoreState(context, values[0]);
  visible = Boolean) values[1]).booleanValue();
  addDefaultActionListener(context);
}
```

Auch hierbei wird zunächst der *DefaultActionListener* entfernt und danach auf die gespeicherten Werte zugegriffen.

Die Wahl der passenden Methode der Zustandsspeicherung ist oftmals ein entscheidendes Kriterium, ob eine Anwendung überhaupt fehlerfrei lauffähig ist. Um die Wichtigkeit einer Zustandsspeicherung sowie die Konsequenzen einer serverseitigen und clientseitigen Speicherung darzustellen, wird folgendes kurzes Beispiel verwendet:

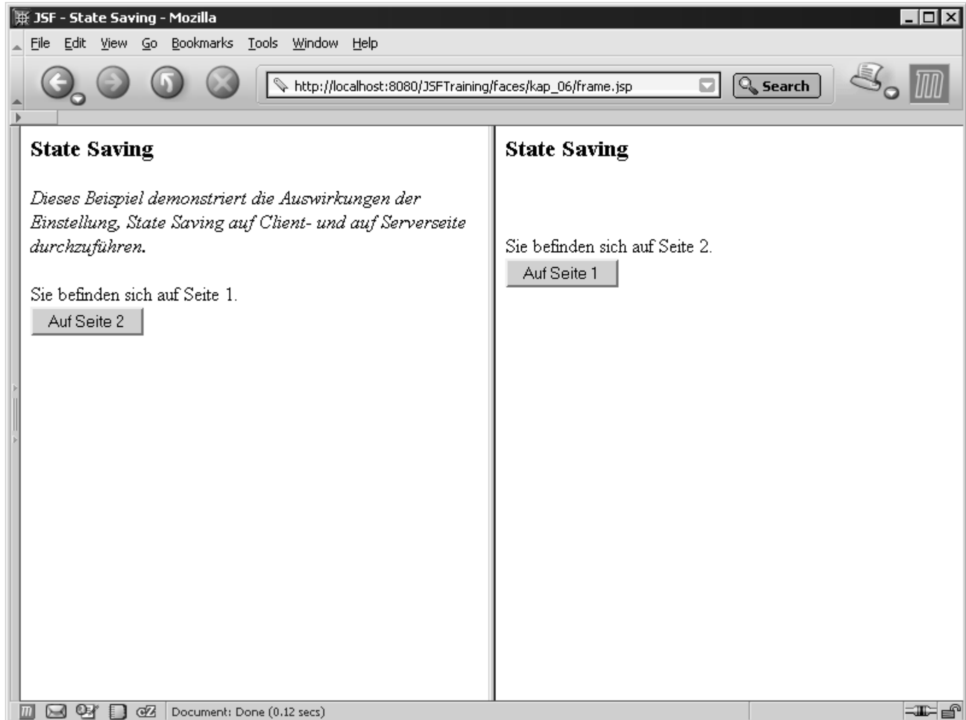


Abbildung 6.12: State Saving im Frameset

Innerhalb eines Framesets (ein Frame unterteilt eine Seite eines Browsers in zwei oder mehrere Teilbereiche) wird jeweils die *gleiche* Faces-Seite angezeigt. Auf dieser Seite befindet sich lediglich ein Button, mit dem auf eine Folgeseite navigiert werden kann. Von der Folgeseite kann ebenfalls wieder zurücknavigiert werden. Ist eine serverseitige Zustandsspeicherung hinterlegt (dies ist das Standardverhalten), so wird der Zustand der Seite (der View) in der Session auf dem Server gespeichert. Da jedoch beide Frames auf die gleiche Session zugreifen, liegt immer nur die View des Frames in der Session, der den letzten Requests durchgeführt hat. Erfolgt dann von dem anderen Frame ein Request, ist in der Session nicht die zu diesem Frame passende View hinterlegt. Das weiß das Framework jedoch nicht und rekonstruiert die View aus der Session. Dies führt dann natürlich zu einem nicht korrekten Verhalten. Im Beispiel passiert daher zunächst einmal nichts, obwohl ein Button gedrückt wurde und ein Seitenwech-

sel stattfinden sollte. Das hat den Hintergrund, dass in der View noch der Zustand des anderen Frames hinterlegt ist, der zur Anfrage des neuen Frames natürlich nicht passt. Erst nach einem erneuten Betätigen des Buttons liegt vom vorhergehenden Request die korrekte View in der Session und es erfolgt wieder die korrekte Navigation.

Wird dagegen eine clientseitige Zustandsspeicherung wie in Listing 6.63 verwendet, erfolgt die Speicherung der View im Client, d.h. in so genannten versteckten Feldern (hidden fields). In diesen Feldern, die durch JSF automatisch der Seite hinzugefügt werden, ist die Session in serialisierter Form abgelegt. Da diese Information in jedem Frame separat hinterlegt ist, kann bei jedem eingehenden Request die passende View aus den Sessioninformationen der versteckten Felder rekonstruiert werden. Es erfolgt dann auch die gewünschte Navigation.

6.12 Application-Objekt

Ein `Application`-Objekt steht als Singleton mit einer einzigen zentralen Instanz einer Webanwendung zur Verfügung. Im Gegensatz zu session- oder benutzerabhängigen Objekten existiert vom `Application`-Objekt lediglich eine Instanz pro Webanwendung. Somit kann Application als globale Ressource bezeichnet werden. Zudem stellt die Klasse selbst einige Factory-Methoden für das Erzeugen von UI-Komponenten, Konvertern oder Validatoren bereit. Zugriff erhält man über den statischen Aufruf

```
ApplicationFactory.getApplication()
```

Eine häufig verwendete Methode des `Application`-Objekts ist der Zugriff auf ein so genanntes `ValueBinding`. Ein ValueBinding repräsentiert einen bestimmten Wert oder aber einen Ausdruck einer Action-Referenz. Somit kann über das ValueBinding auf Managed-Beans zugegriffen werden.

```
Application application = factory.getApplication();
ValueBinding binding = application.createValueBinding(
                                "PersonBean");
PersonBean person = (PersonBean) binding.getValue( context );
```

Wie in obigem Beispiel ersichtlich, wird hierbei über das ValueBinding auf ein Objekt der Klasse `PersonBean` zugegriffen. Die Klasse `ValueBinding` bietet für einen Zugriff mehrere Methoden zur Auswahl an:

- `getType(FacesContext context)`: Liefert den Typ (die Klasse) des beinhalteten Wertes zurück.
- `getValue(FacesContext context)`: Liefert den eigentlichen Wert zurück.
- `isReadOnly(FacesContext context)`: Liefert zurück, ob der Wert schreibgeschützt ist.
- `setValue(FacesContext context, Object value)`: Setzt einen neuen Wert.

Im Regelfall wird mit den Methoden `getValue` und `setValue` gearbeitet. So ist es damit möglich, bestimmte Werte aufgrund eines Programmverlaufs zu setzen bzw. deren aktuelle Werte auszulesen.

6.13 Internationalisierung

Webanwendungen sind heutzutage nur noch in seltenen Fällen ausschließlich für den deutschen Markt und den deutschen Anwender bestimmt. Vielmehr bieten ja gerade Webanwendungen den Vorteil, von überall auf der Welt bedienbar zu sein. Daher ist es vor allem für Anwendungen im Web wichtig, verschiedene Sprachen und Datenformate zu unterstützen. Mittels den Konzepten der Internationalisierung wird eine Anwendung darauf vorbereitet, in mehreren Länder- bzw. Sprachversionen eingesetzt zu werden. JavaServer Faces bieten basierend auf den Konzepten von Java selbst Lösungsvorschläge für eine Internationalisierung der Anwendung an.

Statische Ausgaben

Sämtliche JSF-Komponenten, die Textausgaben erzeugen, bieten die Möglichkeit, anstatt eines festen Bezeichners einen Platzhalter zu verwenden, über den in einer entsprechenden Ressourcendatei der länderspezifische Text angezogen wird. Dieses Konzept der `ResourceBundles` ist nichts JSF-Spezifisches, sondern wird von Java schon seit langem bereitgestellt. Um auf Resourcebundles zugreifen zu können, ist zuerst einmal eine entsprechende Textdatei mit der Endung *.properties* zu erzeugen, in der in einer einfachen Schlüssel-Wert-Darstellung die notwendigen Texte hinterlegt werden.

```
btnSubmit=Abschicken
btnOk=Ok
btnCancel=Abbrechen
btnSave=Speichern

txt_1=Testausgabe
txt_2=Überschrift

msgErr=Es ist ein Fehler aufgetreten.
msgTimeout=Es ist ein Timeout aufgetreten. Bitte neu anmelden.
```

Listing 6.64: Ausschnitt aus einer Ressourcendatei

Basierend auf einer Basis-Ressourcendatei, in der sämtlichen erforderlichen Bezeichner und Texte hinterlegt sind, können anschließend länderspezifische Versionen erstellt werden. Diese unterscheiden sich natürlich zum einen in den Inhalten, wobei die Schlüssel immer genau dieselben sein müssen. Zum anderen haben die länderspezifischen Dateien entsprechende Dateiendungen. Nimmt man die Bezeichnung *commandBundle.properties* als Basisdatei, sind z. B. folgende Sprachversionen denkbar:

- *commandBundle_de_DE.properties* für Deutschland
- *commandBundle_fr_FR.properties* für Frankreich
- …

Um eine Ressourcendatei in eine JSF-Seite einzubinden, stellt JSF ein entsprechendes Tag zur Verfügung:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head>
<title>JSF-Beispiel</title>
</head>

<f:loadBundle basename="commandBundle" var="bundle"/>
...
```

Listing 6.65: Verwendung einer Ressourcendatei

In Listing 6.65 wird eine Ressourcendatei *commandBundle* eingebunden. Dazu wird über das Tag `<f:loadBundle>` die Ressource unter dem Namen *bundle* eingebunden. Wichtig zu wissen ist, dass die Ressourcendateien im Klassenpfad gesucht werden, daher ist entsprechend der vollqualifizierende Name mitanzugeben, abhängig vom Speicherort der Dateien.

Nach diesen Vorarbeiten kann in den entsprechenden JSF-Komponenten mittels eines Ausdruckes auf die Ressourcendatei zugegriffen werden.

```
<h:outputText value="#{bundle.txt_1}" />
<h:commandButton action="success" value="#{bundle.btnSubmit}" />
```

Listing 6.66: Button und Textfeld mit Bezeichnung aus einer Ressourcendatei

In Listing 6.66 wird ein Commandbutton sowie ein Text ausgegeben, dessen Bezeichnungen über den Schlüssel *bundle* aus der Ressourcendatei angezogen werden. Nach diesem Prinzip verläuft die Verwendung von Einträgen aus Ressourcendateien in jeder Komponente. Es muss jedoch die Ressourcendatei auf jeder Seite explizit geladen werden.

Dynamische Ausgaben

Analog zur Ausgabe von Texten in UI-Komponenten mit Hilfe einer Ressourcendatei können auch dynamische Meldungen nach demselben Schema erzeugt werden. Auch hierbei wird über das `loadBunde`-Tag die Ressourcendatei zunächst geladen, um im Anschluss daran im Tag verwendet zu werden. Der Unterschied zum `outputText`-Tag liegt darin, dass zusätzliche Parameter mitgegeben werden können.

```
<h:outputFormat value="#{bundle.greetings}">
  <f:param value="#{session.firstname}"/>
  <f:param value="#{session.lastname}"/>
</h:outputFormat>
```

Listing 6.67: Ausgabe mit Parameter

In Listing 6.67 wird eine dynamische Meldung ausgegeben, die als Parameter den Vor- und Nachnamen übergeben bekommt, die wiederum beide im aktuellen Sessionkontext zu finden sind. Näheres zu diesem Tag sowie zu weiteren UI-Komponenten ist ausführlich im Kapitel 7 erläutert.

6.14 Fazit

JavaServer Faces ist ein sehr mächtiges und umfangreiches Framework – dies ist mit Sicherheit in diesem Kapitel zum Ausdruck gekommen. Dennoch sind die Konzepte sehr einleuchtend und nachvollziehbar.

Um eine Webanwendung mittels JavaServer Faces umzusetzen, ist primär eine Konfigurationsdatei notwendig. In ihr werden Einträge für das Bean-Management vorgenommen sowie das Navigationsverhalten festgelegt.

Basierend auf einer strikten Trennung zwischen Funktionalität und Darstellung existieren in JavaServer Faces Komponentenklassen, die darstellungsunabhängig Funktionalität für eine Komponente bereitstellen. Erst so genannte Renderer sorgen dafür, dass eine Komponente z.B. in Html dargestellt wird. Eine Art Zwitterfunktion haben die *HTML-Komponenten* inne. Sie repräsentieren Komponenten, die als Ausgabe ausschießlich in HTML dargestellt werden.

Für eine Überprüfung der Eingabedaten können an die meisten Eingabekomponenten Validatoren angehängt werden. So kann z.B. mit einem LongRange-Validator ein Bereich festgelegt werden, in dem eine Benutzereingabe vorliegen muss. Andernfalls wird ein entsprechender Fehler erzeugt.

Ebenfalls sehr ausführlich wurde das Thema Eventhandling erläutert. So ist es möglich, an Kommandokomponenten wie z.B. einen Commandbutton oder einen Commandlink, Listener zu registrieren, die bei Auslösen der Komponente aufgerufen werden. Neben diesen ActionEvents existieren zudem noch so genannte ValueChangeEvents, die bei einer Werteänderung aktiviert werden. Im Gegensatz zur klassischen Swing-Programmierung werden ValueChangeEvents jedoch erst nach Abschicken des Eingabeformulars an den Server erzeugt, da nur auf Serverseite eine Veränderung der Eingabewerte festgestellt werden kann.

Um Anwendungen international zu machen, existiert in JavaServer Faces das Konzept der Ressourcendateien. In ihnen können länderspezifische Texte und Ausgaben hinterlegt werden. Auch Fehlermeldungen können in verschiedenen Sprachen hinterlegt werden und werden durch das Framework jeweils passend angezogen.

7 UI-Komponenten und ihre Darstellung

Kapitelziel

JavaServer Faces ist mehr als ein reines Framework zur Verwaltung und Steuerung von Programmabläufen. Mittels der *HTML_basic-Taglib* steht dem Entwickler zugleich eine umfangreiche Tag-Bibliothek (Taglib) zur Verfügung, mit deren Hilfe schnell und einfach umfangreiche und moderne Oberflächen erstellt werden können.

Ziel dieses Kapitels ist es daher, die Verwendung der einzelnen Tags sowie der dazugehörigen Komponenten zu demonstrieren und die umfangreichen Möglichkeiten darzustellen.

Sicherlich sind in JavaServer Faces nicht UI-Komponenten für jeden erdenklichen Einsatzzweck standardmäßig enthalten. Jedoch steht eine umfangreiche Basisbibliothek zur Verfügung, die bei Bedarf mit überschaubarem Aufwand schnell erweitert und an die eigenen Bedürfnisse angepasst werden kann.

Wie bereits im Kapitel 4 beschrieben, werden einige der in diesem Kapitel beschriebenen Komponenten teilweise mittels JavaScript gerendert. An Stellen, an denen dieses Verhalten nicht gewünscht ist, muss gegebenenfalls eine eigene Komponente bzw. ein eigener Renderer geschrieben werden. Wenn JavaScript zur Darstellung einer Komponente verwendet wird, wird darauf speziell hingewiesen.

Da eine Kommunikation einer Weboberfläche mit einem Benutzer hauptsächlich über Formulare geschieht, beinhalten die Tags der *HTML_basic-Taglib* zum Großteil Steuerkomponenten für die Formularverarbeitung sowie ein paar wenige zusätzliche weitere Komponenten.

In den folgenden Abschnitten werden alle wichtigen Komponenten der *HTML_basic-Taglib* hinsichtlich ihrer Verwendung erläutert und es wird an Beispielen gezeigt, wie diese in eine Applikation integriert werden können. Wichtigste Voraussetzung ist, dass sämtliche Tags *immer* in einem `view`-Tag eingeschlossen sein müssen. Dieses Tag darf nur einmal pro Seite verwendet werden und muss sämtliche JSF-Tags umgeben.

Aufgrund der Tatsache, dass für jede Komponente eine Vielzahl an Attributen vorgesehen sind, kann natürlich nicht auf jedes Attribut im Detail eingegangen werden. Es wird vielmehr der Schwerpunkt auf die häufigsten und in Anwendungen am ehesten

verwendeten Attribute gelegt. Sollten Sie ein eventuelles Attribut vermissen bzw. wollen Sie wissen, welche Attribute es überhaupt für eine Komponente gibt, kann in der *HTML_basic-Taglib* direkt nachgelesen werden bzw. es wird eine ausführliche Taglib-Dokumentation mit JSF direkt mitausgeliefert.

7.1 Grundaufbau von UI-Komponenten

Für das Verständnis von JavaServer Faces ist es wichtig, die Unterscheidung zwischen Komponente und Darstellung zu kennen. Eine Komponente ist zunächst einmal ein Baustein innerhalb einer Webanwendung, der gewisse Funktionalitäten bereitstellt. Eine Komponente ist jedoch unabhängig von einer konkreten Darstellung. Meist wird die Darstellung sicherlich in HTML erfolgen, doch auch andere Darstellungen beispielsweise in Wml oder sogar rein in Xml sind denkbar und auch gewollt. Im Standard liefert JavaServer Faces Renderer für die Ausgabe im HTML-Format mit. In den folgenden Kapiteln werden daher die Komponenten vorgestellt samt ihrer Verwendung bei Nutzung der Standardrenderer und der mitgelieferten Taglibs.

Für einen Entwickler ist es jedoch freigestellt, basierend auf den vorhandenen Komponenten eigene Renderer zu schreiben oder auch eine eigene Taglib mit weiteren Attributen bereitzustellen. Natürlich können auch jederzeit neue UI-Komponenten entworfen werden. Mehr dazu ist im Kapitel 9 zu finden.

Allen UI-Komponenten ist jedoch gemeinsam, dass sie einen eindeutigen Bezeichner haben. Dieser kann über die Tags der jeweiligen Komponenten gezielt gesetzt werden. Wird kein spezieller Bezeichner angegeben, wird durch das Framework automatisch eine eindeutige Id vergeben.

Daneben gibt es eine Anzahl an Attributen, die bei den meisten Komponenten vorzufinden sind, abhängig von deren genauer Art:

- `value`: Ein- und Ausgabekomponenten haben meist einen Wert, den sie anzeigen bzw. deren Wert durch den Benutzer eingegeben werden kann. Der Wert kann dabei als fixer String angegeben werden oder aber aus einem Modellobjekt bezogen werden. Auch ist es möglich, Werte aus einer Ressourcendatei anzugeben.
- `action`: Beim Auslösen eines Hyperlinks oder eines Buttons soll oftmals nicht nur auf eine nächste Seite geleitet werden, sondern ebenfalls eine Aktion ausgelöst werden. Mittels action kann daher auf eine Aktionsmethode verwiesen werden, die beim Auslösen einer Komponente aufgerufen wird. Oftmals wird jedoch an dieser Stelle lediglich ein Bezeichner angegeben, der über die Anwendungskonfigurationsdatei das weitere Verhalten steuert.

7.2 Komponentenbaum

Komponenten in JSF werden in einer hierarchischen Baumstruktur im Kontext (FacesContext) vorgehalten. Dies hat den großen Vorteil, dass ein direkter Zugriff über den Komponentenbaum möglich ist und somit Eingriffe in die Darstellung jederzeit möglich sind. So können aufgrund des Programmablaufs Komponenten aus dem Komponentenbaum gelöscht oder verändert werden. Über den Aufruf

```
public UIViewRoot getViewRoot()
```

erhält man ein Objekt der Klasse `javax.faces.component.UIViewRoot`. Dies ist die Komponente, die die Basis für den gesamten Komponentenbaum einer Seite ist. Sämtliche weiteren Komponenten einer Faces-Seite sind Kindelemente dieser Komponente. `UIViewRoot` stellt somit die erste und oberste Komponente in der Komponentenhierarchie dar. `UIViewRoot` selbst hat keine Renderer, die eine Darstellung der Komponente ermöglichen. Vielmehr ist `UIViewRoot` lediglich als zentrale Stelle im Komponentenbaum zu verstehen. Die Klasse stellt u.a. folgende Methoden zur Verfügung:

- `public String getRenderKitId()`
- `public void setRenderKitId( String renderKitId )`
- `public String getViewId()`
- `public void setViewId( String viewId )`
- `public String createUniqueId()`

Über den Aufruf `getRenderKitId` wird ein eindeutiger Bezeichner zurückgeliefert, über den ein so genanntes Render-Kit bezeichnet wird. Für einfache HTML-basierte Anwendungen genügt im Normalfall das Standard-Render-Kit. Sollte jedoch beispielsweise auch die Möglichkeit gegeben sein, die Anwendung auf mobilen Geräten wie z.B. einem Wap-Handy darzustellen, kann ein anderes Render-Kit gesetzt werden. Natürlich muss auch jede View einen eindeutigen Bezeichner haben. Dieser kann mit `getViewId` abgefragt und mit `setViewId` explizit gesetzt werden. Diese Aufgaben werden jedoch in der Regel durch das Framework erledigt. Für den Fall, dass eine eindeutige Id erzeugt werden muss, kann durch den Aufruf `createUniqueId` eine innerhalb einer View eindeutige Id erzeugt werden.

`UIViewRoot` ist von der Klasse `UIComponentBase` abgeleitet, die wiederum verschiedene Möglichkeiten zum Zugriff auf den Komponentenbaum bereitstellt. So kann über den Aufruf

```
public java.util.List getChildren()
```

eine Liste aller untergeordneten Komponenten abgefragt werden. Ist eine Seite sehr umfangreich, kann auch mittels

```
public int getChildCount()
```

zunächst nur die Anzahl der Kindelemente bestimmt werden. Natürlich kann dann wiederum von jedem Kindelement auch ein Iterator über dessen Unterelemente angefordert werden. `UIComponentBase` stellt für die Navigation innerhalb der Komponentenhierarchie eine Vielzahl nützlicher Methoden zur Verfügung, mit denen sehr einfach zur gewünschten Komponente navigiert werden kann.

Methode	Beschreibung
`public UIComponent findComponent( String expr )`	Sucht gemäß des Ausdruckes nach einer Komponente
`public int getChildCount()`	Liefert die Anzahl der Kindelemente zurück.
`public java.util.List getChildren()`	Liefert eine Liste aller Kindelemente zurück.
`public void setParent( UIComponent parent)`	Setzt das Elternelement für eine Komponente.

Tabelle 7.1: Einige Methoden von UIComponentBase

7.3 Renderer

Komponenten in JSF sind zunächst einmal losgelöst von ihrer späteren Darstellung zu betrachten. Während Komponenten die eigentliche Funktionalität beinhalten, werden Renderer für die Darstellung der Komponente verwendet. So existieren Renderer für Buttons, Eingabe- und Ausgabefelder, Listen und vieles mehr. Hauptsächlich werden JSF-Anwendungen sicherlich im Webumfeld, sprich im HTML-Umfeld, eingesetzt werden. Aus diesem Grund existieren bereits Standardrenderer für die Ausgabe in HTML. Doch auch für Ausgaben auf Wap-Handys oder mobilen Geräten ist JSF sehr geeignet. Dies wird erreicht, indem durch benutzerdefinierte Renderer eine für das Ausgabegerät definierte Ausgabe erzeugt wird. Das Standard-Render-Kit von JSF liefert als Ausgabe HTML, das jeder Internetbrowser unterstützt. Da das Framework jedoch so ausgelegt ist, dass Anwendungs- bzw. Komponentenentwickler jederzeit neue Komponenten und Renderer entwickeln können, kann für nahezu jede Ausgabeform ein spezieller Renderer erstellt werden.

Ein *Render-Kit* ist dabei eine logische Zusammenfassung verschiedener einzelner Renderer (z.B. eines RendererS für einen Button, eines Renderers für ein Ausgabefeld etc.) für ein Ausgabeformat. Es ist somit möglich, mehrere Render-Kits anzulegen und diese bei Bedarf einzusetzen. Das ganze könnte auch während eines Programmablaufs geschehen.

Für das Verständnis von JSF ist es daher sehr wichtig, den Unterschied zwischen einer Komponente und der Darstellung zu verstehen. In JSF gibt es häufig eine 1:1-Zuordnung von Komponente und Renderer, doch es gibt auch Ausnahmen. Die UICommand-Komponente kann z.B. als Button oder aber als Hyperlink gerendert, also dargestellt, werden.

In einem späteren Kapitel werden sie lernen, eigene Renderer zu erzeugen, beispielsweise für die Ausgabe von WML- oder XML-Daten. Eigene Renderer werden häufig im Zusammenhang mit eigenen neuen Komponenten verwendet, aber auch um vorhandene Komponenten um neue Renderer zu erweitern ist das Wissen um Render-Kits hilfreich.

7.4 HTML-Komponenten

UI-Komponenten sind wie bereits gesehen primär losgelöst von deren späterer Darstellung in einer Faces-Seite. Dies ist auch ein großer Vorteil von JSF. In den UI-Komponenten steckt bereits sehr viel an Funktionalität. Dabei ist es noch vollkommen unabhängig davon zu sehen, wie die Komponenten dargestellt werden – sei es in HTML, Wml oder in einer reinen Xml-Darstellung. Weitere Darstellungsformen sind natürlich jederzeit möglich. Erst durch Renderer wird den Komponenten eine konkrete Gestalt verliehen.

So existieren auf der einen Seite Komponenten, die noch losgelöst von deren späterer Darstellung sind. Auf der anderen Seite gibt es Renderer, die eine Komponente z.B. in HTML-Form darstellen. Zusätzlich zu den Komponenten, die direkt von `UIComponent` erben, gibt es die so genannten *HTML-Komponenten*. Für jede mögliche Kombination aus Standard-Renderer und UI-Komponente existiert eine HTML-spezifische Komponente. Alle HTML-Komponenten sind dabei im Paket `javax.faces.component.html` zu finden. Jede darin enthaltene Klasse ist eine Unterklasse einer Standardkomponente, erweitert um render-spezifische Eigenschaften.

Als Beispiel nehmen wir die Komponente `UIInput`. Diese dient der Darstellung von Benutzereingaben und -ausgaben. Die Komponente besitzt einen Wert, der angezeigt oder auch neu gesetzt werden kann. Bezugnehmend auf eine Darstellung in HTML existieren verschiedene Möglichkeiten, eine Benutzereingabe abzubilden. So kann ein einfaches Eingabefeld verwendet werden, ein Textbereich oder im Falle einer Eingabe eines Passworts auch ein Eingabefeld, das mit Sternchen geschützt ist. Daher existieren HTML-spezifische Unterklassen der `UIInput`-Komponente, z.B. `HtmlInputHidden`, `HtmlInputText` oder `HtmlInputTextArea`. Diese haben natürlich wiederum ihre speziellen Renderer, z.B. den Renderer `Textarea` für die Komponente `HtmlInputTextArea`.

Die Komponentenklasse `HtmlInputTextArea` selbst weist entsprechende Eigenschaften auf, die auf die Darstellung der Komponente in HTML passen. So existieren u.a. die Methoden `getCols` oder `getRows`, die die Größe des Eingabefeldes im Browser zurückliefern.

In den folgenden Beschreibungen werden daher sowohl die UI-Komponenten selbst wie auch die HTML-Komponenten erläutert. Da das Standard-Render-Kit, das in der Referenzimplementierung enthalten ist, HTML als Ausgabe erzeugt, sind die Beispiele auch im Hinblick auf eine HTML-Darstellung im Browser ausgerichtet. Eine Ausgabe

für ein anderes Ausgabeformat (XML-Format) ist in der Beispielanwendung in Kapitel 9 erläutert. Um die Zusammenhänge zwischen Komponentenklasse, eventueller HTML-Komponentenklasse, Renderer und Tag deutlich zu machen, ist in den folgenden Abschnitten zu Beginn jeweils eine Übersichtstabelle gegeben. Auf die einzelnen Inhalte wird in den darauf folgenden Erläuterungen im Detail eingegangen.

7.5 Views und Subviews

Übersicht

Komponentenklasse	javax.faces.component.UIViewRoot
HTML-Komponente	-
Renderer	
Tag	<f:view>, <f:subview>

Erläuterung

Eine Faces-Seite ist immer in einer Baumstruktur aufgebaut. Basiskomponente ist die UIViewRoot-Komponente. Das Tag <f:view> repräsentiert diese Komponente innerhalb einer Faces-Seite. Es bildet somit den Rahmen für alle weiteren Tags in einer Seite. Eine explizite Ausgabe wird durch das Tag nicht erzeugt, in der späteren HTML-Ausgabe ist nichts zu erkennen. Vielmehr hat das Tag lediglich organisatorische Aufgaben. Optional kann dem Tag als Attribut eine Locale mitgegeben werden. Wird diese nicht explizit gesetzt, wählt JSF automatisch die passende Locale aus.

In JavaServer Pages (kein Schreibfehler, muss Pages heißen und nicht Faces) gibt es die Möglichkeit, innerhalb einer Seite dynamisch andere JSP-Seiten einzubinden. Dazu existiert das Tag

```
<jsp:include page="seite.jsp"/>
```

Auch in einer Faces-Seite kann eine weitere Seite auf diese Weise eingebunden werden. Allerdings muss diese Seite dann innerhalb eines subview-Tags stehen.

```
<f:subview>
  <jsp:include page="seite.jsp"/>
</f:subview>
```

Listing 7.1: Einbinden einer weiteren Seite

Dabei kann das subview-Tag innerhalb der Hauptseite stehen oder erst in der eingebundenen Seite selbst. Innerhalb einer Faces-Seite können so verschiedene weitere Seiten mit einem jeweils umgebenden subview-Tag eingebunden werden. Wie auch im Falle

des `view`-Tags erzeugt das `subview`-Tag keinerlei Ausgaben in der resultierenden HTML-Seite, sondern steht lediglich zur logischen Aufteilung einer Faces-Seite zur Verfügung.

7.6 UIForm-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UIForm`
HTML-Komponente	`javax.faces.component.html.HtmlForm`
Renderer	`javax.faces.Form`
Tag	`<h:form>`

Erläuterung

Vergleichbar zum HTML-Standard ist auch in JSF der Bereich von Benutzerein- und -ausgaben immer durch ein `form`-Tag umgeben. Natürlich ist es möglich, auf einer einzigen JSF-Seite mehrere Formulare zu hinterlegen. Beim Auslösen eines Kommandos wird dann genau der Inhalt des Formulars an den Server übertragen, zu dessen Formular das Kommando gehört. Wichtig ist auch an dieser Stelle, dass sämtliche Komponenten, die für die Datenein- sowie für die Datenausgabe zuständig sind, innerhalb des `form`-Tags stehen. Dasselbe gilt natürlich ebenso für die *Command-Komponenten*, die eine entsprechende Aktion bei Aktivierung auslösen.

Die `UIForm`-Klasse ist wie alle Komponenten im Paket `javax.faces.component` zu finden. `UIForm` ist von `UIComponentBase` abgeleitet und bietet daher dieselben Möglichkeiten wie alle UI-Komponenten auch. Repräsentiert wird die UIForm-Komponente durch `das form`-Tag.

```
<h:form id="inputForm"
   ... an dieser Stelle stehen weitere Eingabe- und
   ... Ausgabekomponenten ...
</h:form>
```

Listing 7.2: Verwendung des form-Tags

Die Angabe eines Bezeichners (*Id*) für ein Formular ist optional wie auch alle weiteren Attribute. Folgende weitere Attribute werden häufig noch verwendet:

- `target`: Bei Framesets kann durch Angabe des Attributs `target` bestimmt werden, in welchen Frame hinein der Request erfolgen soll. Ebenfalls kann durch Angabe von `_blank` eine Anzeige der folgenden Seite in einem neuen Browser erreicht werden.

- `styleClass`: Bei Verwendung von Cascading Stylesheets kann speziell für ein Formular eine Klasse angegeben werden, nach deren Angaben im Stylesheet das Formular gerendert wird.

Eine häufige Fehlerquelle ist oft, dass Eingabe- und Ausgabefelder im Browser nicht dargestellt werden, da das umgebende `form`-Tag vergessen wurde. Während manche Browser diesen Fehler teilweise verzeihen und die Formularelemente dennoch anzeigen, wird bei Verwendung von JSF jedoch eine Verwendung des `form`-Tags bei Nutzung von Eingabe- oder Ausgabekomponenten zwingend benötigt.

```
<%@ taglib uri="http://java.sun.com/jsf/HTML" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<HTML>
<head>
  <title>UI-Komponenten</title>
  <link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
<f:view>
  <h3>Die UIForm-Komponente</h3>
  <h:form id="frmTest" styleClass="mainForm">
    ... an dieser Stelle stehen die Eingabe-
        und Ausgabekomponenten ...
  </h:form>
</f:view>
</body>
</HTML>
```

Listing 7.3: Beispiel für den Rahmen eines JSF-Formulares

In Listing 7.3 wird ein einfaches Formular erzeugt, das der Formklasse `mainForm` angehört. Damit wird auf ein Stylesheet verwiesen, das im Kopf-Bereich mittels des `link`-Tags in die JSF-Seite integriert wurde.

HTML und JSF

Bei der Umsetzung von HTML-Seiten stellt sich oftmals die Frage, ob sämtlicher Inhalt von HTML-Tags in JSF-Tags umgewandelt werden soll. Grundsätzlich ist diese Frage positiv zu beantworten. Da durch die Renderer in JSF das Ausgabeformat definiert wird, sollte in der eigentlichen Seite kein Tag eines speziellen Ausgabeformats auftauchen. In einer korrekten JSF-Seite sind somit lediglich JSF-Tags zu finden, für den HTML-Output sorgen einzig die Renderer. Damit wird es möglich, durch eine einfache Ersetzung des Renderer-Kits eine andersartige Ausgabe zu bekommen, ohne etwas an den Programmdateien verändern zu müssen. Um jedoch die Kurzbeispiele möglichst einfach zu halten, werden in den Beispielen HTML und JSF-Tags vermischt. Dies ist zwar nicht 100%ig korrekt, erleichtert aber gerade für den Neueinsteiger die Lesbarkeit der Kurzbeispiele.

7.7 UICommand-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UICommand`
HTML-Komponente	`javax.faces.component.html.HtmlCommandButton`
	`javax.faces.component.html.HtmlCommandLink`
Renderer	`javax.faces.Button`
	`javax.faces.Link`
Tag	`<h:commandButton>, <h:commandLink>`

Erläuterung

UICommand-Komponenten lösen aufgrund einer Benutzeraktion (Drücken der Return-Taste, Klicken eines Submit-Buttons etc.) eine Aktion aus. Im Regelfall wird das Formular, in dessen Kontext sich die Command-Komponente befindet, an den Server zur Verarbeitung übertragen. `UICommand`-Komponenten werden meist durch folgende Darstellungen repräsentiert:

- Command-Buttons
- Hyperlinks

Es ist aber auch möglich, eine `UICommand`-Komponente als Menüeintrag darzustellen. Dies macht dann Sinn, wenn z.B. in einem Formular eine Listbox mit bestimmten Wahlmöglichkeiten angeboten wird. Jetzt kann ein Benutzer entweder einen Wert in einer Listbox auswählen und danach einen Button betätigen, oder aber das Auswählen eines Listeintrages löst selbst eine Aktion aus. In diesem Fall wird die `UICommand`-Komponente als Listeintrag repräsentiert.

7.7.1 HTMLCommandButton-Komponente

Die `HtmlCommandButton`-Komponente wird in der HTML-Darstellung als Command-Button dargestellt. Command-Buttons werden meist dazu verwendet, Formularinhalte abzuschicken, damit serverseitig Aktionen durchgeführt werden können. Eine einfache Einbindung eines Command-Buttons zeigt folgendes Listing:

```
<h:form id="frmTest">
  <h:commandButton id="testCmd" value="Submit" action="success" />
</h:form>
```

Listing 7.4: Verwendung des commandButton-Tags

Folgende Attribute kommen in Listing 7.4 zum Einsatz:

- `value`: Mittels des `value`-Attributes wird ein fester String angegeben, der als Beschriftung innerhalb des Buttons in der Anwendung erscheint. Durch Angabe eines Ausdruckes der JSF EL (Expression Language) könnte auch auf einen Wert eines Modellobjekts oder auch auf eine Ressourcendatei verwiesen werden.
- `id`: Eine eindeutige Bezeichnung ist zur Identifizierung einer Komponente wichtig. Die Id muss nicht explizit angegeben werden, sondern kann auch durch JSF automatisch vergeben werden.
- `action`: Durch eine Aktion wird das Verhalten der Anwendung gesteuert. Im Regelfall wird nach Auslösen des Kommandos auf die Seite weitergeleitet, die in der Anwendungskonfigurationsdatei *faces-config.xml* unter dem Schlüssel des action-Attributes hinterlegt ist. Es ist jedoch auch möglich, an dieser Stelle einen Methodenaufruf zu hinterlegen, dessen Rückgabewert das weitere Verhalten steuert.

Wie bei fast allen Komponenten kann auch einem Commandbutton ein Attribut `styleClass` mitgegeben werden. Dieses bezieht sich auf eine Klasse, die über ein Stylesheet definiert wird. Durch Stylesheets können weitere graphische Faktoren einer Komponente beeinflusst werden.

Wie in einem späteren Kapitel noch genauer dargestellt wird, kann an einen Button auch jederzeit ein *ActionListener* angehängt werden, der ausgelöst wird, sobald der Button betätigt wird. Ein ActionListener kann an jede UICommand-Komponente angehängt werden.

Stylesheets

Stylesheets sind eine Ergänzung zu HTML. Mit Stylesheets können Sie keine Funktionen programmieren. Vielmehr können HTML-Elemente mittels Stylesheets näher definiert werden. So kann z. B. bestimmt werden, dass ein Commandbutton immer in einer bestimmten Farbe erscheint oder z. B. auch Überschriften immer mit der Schriftart *Arial* dargestellt werden. Der Vorteil bei Verwendung von Stylesheets liegt darin, dass bei kleineren graphischen Änderungen (z. B. aufgrund einer Änderung der Schriftart im Corporate Design) eine einzige Änderung im Stylesheet genügt, es muss nicht mehr jede einzelne HTML-Seite angefasst werden. Zumal können in Stylesheets auch Angaben zu HTML-Komponenten getroffen werden, die über eine reine HTML-Syntax nicht möglich wären.

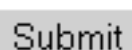


Abbildung 7.1: Darstellung einer UICommand-Komponente als Commandbutton

Interessant ist es auch, sich einmal den HTML-Quellcode genauer zu betrachten, der bei Ausführung der JSF-Seite im Browser vom Server übermittelt wird:

```
<form id="frmTest" method="post"
      action="/JSFTraining/faces/kap_7/commandbutton.jsp">
  <input type="submit" name="frmTest:testCmd" value="Submit" />
  <input type="hidden" name="frmTest" value="frmTest" />
</form>
```

Listing 7.5: HTML-Ansicht im Browser

In Listing 7.5 ist deutlich zu erkennen, dass in dem Falle, dass ein einfacher Command-Button in ein Formular integriert ist, keinerlei JavaScript zur Darstellung und Verarbeitung im Browser verwendet wird. Ebenso ist es sofort ersichtlich, dass das Framework für den Submit-Button den Namen für den Button automatisch vergeben hat, basierend auf den Vorgaben im Tag selbst. Der Name setzt sich zusammen aus dem Bezeichner des Formulares sowie dem Bezeichner des Buttons, der explizit im Tag angegeben wurde. Diese Syntax wird generell verwendet, dass sich der resultierende Name in HTML-Komponenten zusammensetzt aus der hierarchischen Struktur, in der sich die Komponente befindet.

Nachteilhaft an der Verwendung des `commandButton`-Tags aus Beispiel Listing 7.4 ist, dass der Wert des Buttons, also die Bezeichnung *Submit*, fix in der JSF-Seite verdrahtet ist. Hier empfiehlt sich die Verwendung einer Ressourcendatei, um die Bezeichner dynamisch aus einer Datei einzulesen und so bereits eine mögliche Internationalisierung der Anwendung vorbereitet zu haben.

```
<%@ taglib uri="http://java.sun.com/jsf/HTML" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<HTML>
<head>
  <title>UI-Komponenten</title>
  <link rel="stylesheet" type="text/css" href="styles.css" />
</head>

<f:loadBundle basename="commandBundle" var="bundle"/>

<body>
<f:view>
  <h3>Die UICommand-Komponente</h3>
    <i>Dieses Beispiel demonstriert die Verwendung
       der Komponente UICommand unter Zuhilfenahme
       einer Ressourcendatei.</i>
    <h:form id="frmTest">
      <h:commandButton id="testCmd" value="#{bundle.btnSubmit}"
         action="success" />
```

```
    </h:form>
</f:view>
</body>
</HTML>
```

Listing 7.6: Verwendung von Resourcebundles

In Listing 7.6 wird ein `Commandbutton` erzeugt, dessen Bezeichnung über den Schlüssel `btnSubmit` aus der Ressourcendatei `commandBundle` angezogen wird. Wichtig bei Verwendung von Ressourcendateien ist, dass

- das entsprechende Tag für die Verwendung von Ressourcendateien angegeben ist
- die Syntax für die Bezeichnung der Ressource korrekt ist
- und die Ressourcendatei im Klassenpfad zu finden ist.

Mehr zum Thema Verwendung von Ressourcendateien ist in Kapitel 6.13 Internationalisierung zu finden.

Verwendung von Aktionen

Alternativ zur Angabe einer konkreten Aktion, sprich eines Textbezeichners, der in der Anwendungskonfigurationsdatei hinterlegt ist, kann über das `action`-Attribut auch eine Methode aufgerufen werden, deren Rückgabewert das Navigationsverhalten steuert. Die Methode muss jedoch gewisse Bedingungen erfüllen. So muss diese

- öffentlich sein (public)
- einen String als Rückgabewert liefern
- und keine Parameter erwarten.

Der Rückgabewert wird dann wieder auf die hinterlegten Navigationsregeln überprüft und das weitere Navigationsverhalten bestimmt. Auf diese Weise ist es möglich, dynamisch aufgrund des Programmverlaufs die Navigation zu beeinflussen und die Seitenfolge zu steuern.

```
public String doSomeAction() {
  // do something
  return "success";
}
```

Listing 7.7: Eine Aktions-Methode

Listing 7.7 zeigt eine mögliche Action-Methode. In der Faces-Seite kann diese Methode wie folgt aufgerufen werden, vorausgesetzt, diese ist in einer Klasse implementiert, die als Modellobjekt mit dem Namen `UserBean` angesprochen wird.

```
<h:form>
  <h:commandButton value="Submit"
     action="#{UserBean.doSomeAction}" />
</h:form>
```

Listing 7.8: Verwendung des action-Attributes

In Listing 7.8 wird nach Auslösen des Buttons im Bean `UserBean` eine Methode `doSomeAction` aufgerufen, die als Rückgabewerte einen String zurückliefert, der wiederum das weitere Navigationsverhalten steuert.

Dies ist auch die einzige Ausnahme, in der Teile der Navigation im Programmcode hinterlegt sind. Es wurde in den einführenden Abschnitten zum Navigationskonzept bereits angesprochen, dass das Auslagern der Navigation in eine separate Datei ein großer Vorteil ist. In den Aktionsmethoden wird dieses Prinzip jedoch ein wenig aufgeweicht, indem Navigationslogik im Programmcode selbst hinterlegt ist. Da jedoch ausschließlich Bezeichner zurückgeliefert werden (und keine harten Dateinamen mit Pfadangaben), ist dieser Ansatz jedoch durchaus vertretbar.

7.7.2 HTMLCommandLink-Komponente

Die Repräsentation der `UICommand`-Komponente als Hyperlink entspricht dem klassischen Hyperlink aus HTML. Wichtig ist in diesem Fall die Betonung auf *Command*, da die Ausgabe des HTML-Codes immer auf eine Aktion führt. Es ist zwar auch möglich, einen Link durch JSF ausgeben zu lassen, der wie ein klassischer Link lediglich zu einer neuen Adresse führt. Im Falle der Verwendung der Darstellung als Hyperlink der `HtmlCommandLink`-Komponente wird jedoch immer eine Aktion ausgeführt.

Sollten Sie somit lediglich einen Link beispielsweise zu einer externen Url ausgeben wollen, kommt dafür die `HtmlOutputLink`-Komponente in Frage, die in einem späteren Abschnitt näher beschrieben wird. In diesem Fall würde keine Aktion ausgelöst werden, sondern lediglich zu einer angegebenen Adresse verwiesen.

Ein typisches Einsatzgebiet des Hyperlink in Form eines Kommandos ist somit im Zusammenhang mit Formularen zu sehen. Statt der Anzeige eines Buttons kann auch ein Link zum Versenden von Formularinhalten verwendet werden.

```
<h:form id="frmTest">
  <h:commandLink action="success">
    <h:outputText value="Submit" />
  </h:commandLink>
</h:form>
```

Listing 7.9: Verwendung des commandLink-Tags

Die Ausgabe des Textes des Links erfolgt durch ein geschachteltes `outputText`-Tag, in dem der String als `value`-Attribut direkt angegeben werden kann. Natürlich ist es auch hier wieder möglich, auf eine Ressourcendatei zuzugreifen.

Das Listing 7.9 führt zu folgender Ausgabe:

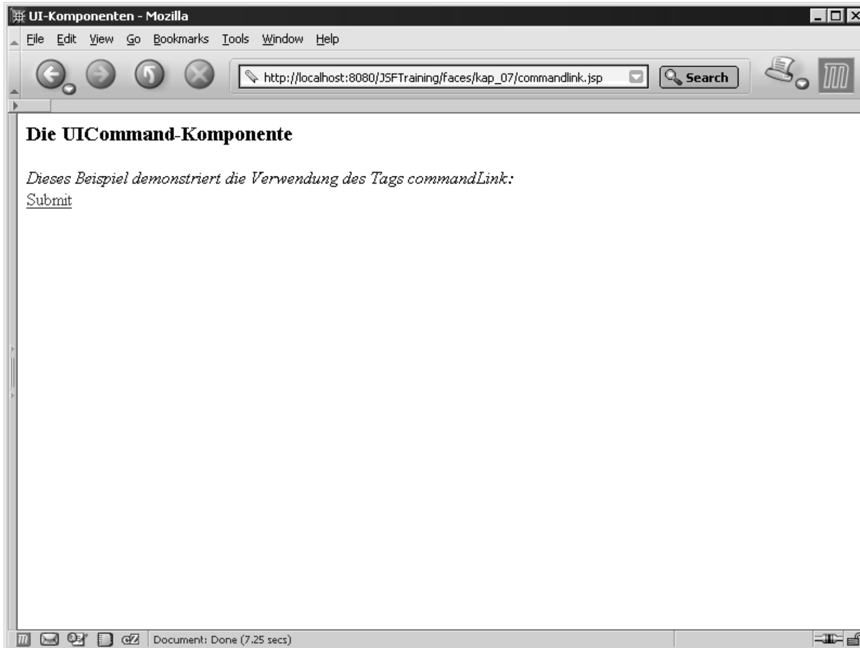


Abbildung 7.2: Darstellung als Hyperlink

Zu beachten bei Verwendung des Tags `commandLink` ist, dass hierbei zum Rendern Java-Script verwendet wird. Der HTML-Quellcode verdeutlicht, in welcher Form dies geschieht:

```
<form id="frmTest" method="post"
      action="/JSFTraining/faces/kap_7/command_link.jsp">
  <a href="#"
     onmousedown="document.forms[0]['frmTest:_id0'].value=
     'frmTest:_id0'; document.forms[0].submit()">
    Submit
  </a>
  <input type="hidden" name="frmTest:_id0" />
  <input type="hidden" name="frmTest" value="frmTest" />
</form>
```

Listing 7.10: Ausgabe im Browser bei Verwendung des commandLink-Tags

Es werden vom Framework beim Rendern automatisch versteckte Felder miteingebaut (so genannte *hidden fields*), in denen Werte des Kommandos eingetragen werden und auf diesem Wege an den Server übermittelt werden können. Das Befüllen der Hidden Fields sowie das eigentliche Abschicken des Formulars geschieht dabei mittels JavaScript.

Ebenso wie bei der Darstellung als Button kann auch bei der Darstellung als Link auf zwei Arten das Verhalten beim Auslösen angegeben werden:

- Direkter Wert beim `action`-Attribut: Über das `action`-Attribut kann direkt angegeben werden, auf welche Folgeseite weitergeleitet werden soll. Der angegebene Wert bezieht sich dabei wieder auf die Navigationsregeln in der Anwendungskonfigurationsdatei.
- Aufruf einer `Action`-Methode: Durch die Angabe einer `Action`-Methode wird diese bei Auslösen des Links aufgerufen. Der Rückgabewert der Methode steuert dann das weitere Navigationsverhalten.

Angabe zusätzlicher Parameter

Gerade bei Verwendung von Formularen kommt es häufig vor, dass zusätzlich zu den sichtbaren Oberflächenkomponenten weitere Informationen in Form von *Hidden Fields* mitgegeben werden müssen. Dies sind meist Steuer- und Statusinformationen, die für eine Anzeige im Browser nicht bestimmt sind, für den weiteren Programmablauf jedoch notwendige Angaben sind. Es kann aber auch der Fall sein, dass durch bestimmte JavaScript-Routinen Hidden Fields verwendet werden müssen. Auch bei Einsatz des `commandLink`-Tags ist es möglich, weitere Parameter anzugeben, die beim Auslösen des Hyperlinks mit an den Server übertragen werden.

```
<h:form id="frmTest">
  <h:commandLink action="success">
    <h:outputText value="Submit" />
    <f:param name="userid" value="27" />
  </h:commandLink>
</h:form>
```

Listing 7.11: Angabe weiterer Parameter

Zu beachten ist lediglich, dass das Parameter-Tag innerhalb des `commandLink`-Tags stehen muss, es ist somit ein gültiges Start- und Endtag notwendig. Das Listing 7.11 wird durch die Renderer wie folgt im Browser dargestellt:

```
<form id="frmTest" method="post"
  action="/JSFTraining/faces/kap_7/command_link_param.jsp">
    <a href="#"
       onmousedown="document.forms[0]['frmTest:_id0'].value=
      'frmTest:_id0';document.forms[0]['name'].value='27';
      document.forms[0].submit()">Submit</a>
```

```
    <input type="hidden" name="frmTest:_id0" />
    <input type="hidden" name="name" value="27" />
    <input type="hidden" name="frmTest" value="frmTest" />
  </form>
```

Listing 7.12: Ergebnis von Parametern in der HTML-Ansicht

Es wird somit bei Angabe des Parameter-Tags ein zusätzliches *Hidden Field* erzeugt, das zusammen mit den anderen Angaben mit Auslösen des Hyperlinks im Formular an den Server übermittelt wird.

Natürlich kann auch in einem Parameter-Tag auf Managed-Beans zurückgegriffen werden. Dies geschieht wiederum durch Angabe eines `value`-Attributes.

```
<h:form id="frmTest">
  <h:commandLink action="success">
    <h:outputText value="Submit" />
    <f:param name="userid" value="#{Person.userId}" />
  </h:commandLink>
</h:form>
```

Listing 7.13: Verwendung des value-Attributes

7.8 UIInput-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UIInput`

HTML-Komponente	`javax.faces.component.html.HtmlInputHidden`
	`javax.faces.component.html.HtmlInputSecret`
	`javax.faces.component.html.HtmlInputText`
	`javax.faces.component.html.HtmlInputTextarea`

Renderer	`javax.faces.Hidden`
	`javax.faces.Secret`
	`javax.faces.Text`
	`javax.faces.Textarea`

Tag	`<h:inputHidden>`
	`<h:inputText>`
	`<h:inputSecret>`
	`<h:inputTextarea>`

Erläuterung

Die UIInput-Komponenten dienen dazu, Benutzereingaben entgegenzunehmen und an ein Modellobjekt zu übertragen. Dargestellt im Browser werden UIInput-Komponenten als Eingabefelder, Checkboxen, Listboxen oder Menüeinträge. Im folgenden »engeren« Sinne werden in diesem Abschnitt zunächst diejenigen Eingabekomponenten besprochen, die als Eingabefeld dargestellt werden. UIInput-Komponenten, die als Checkbox, Listbox etc. dargestellt werden, folgen in einem späteren Abschnitt. Während im klassischen HTML ein Eingabefeld lediglich alphanumerischen Inhalt entgegennimmt, existiert in JSF die Möglichkeit, das Eingabeformat bzw. den Eingabetyp genauer zu spezifizieren. In der aktuellen Referenzimplementierung erfolgt die Überprüfung der Eingaben auf Serverseite; das bedeutet, dass der gesamte Formularinhalt zuerst an den Server übermittelt wird und das Ergebnis nach Validierung an den Browser zurückübermittelt wird. Eine Überprüfung mittels JavaScript, was einer Validierung auf dem Client entsprechen würde, ist nicht vorgesehen und müsste gegebenenfalls mittels eigenen Renderern hinzugefügt werden. Die Klasse UIInput hat eine Reihe von abgeleiteten Klassen, die für unterschiedliche Arten von Eingabekomponenten verwendet werden können:

- HtmlInputText
- HtmlInputTextArea
- UISelectBoolean
- HtmlInputHidden
- HtmlInputSecret
- UISelectOne
- UISelectMany

Passend zu einzelnen Komponenten existieren folgende Renderer zur Darstellung der einzelnen Komponenten im Browser:

- Text-Renderer zur Darstellung nicht formatierter, alphanumerischer Eingaben.
- Textarea-Renderer: zur Eingabe eines Textblocks, der sich über mehrere Zeilen und Spalten erstrecken kann
- Hidden-Renderer: für die Angabe so genannter Hidden Fields
- Secret-Renderer: Für die Eingabe von Passwörtern oder Daten, die nicht an der Oberfläche angezeigt werden sollen. Bei der Eingabe wird pro getipptem Buchstaben oder Zahl ein »*« -Zeichen ausgegeben.
- CheckBox-Renderer zur Darstellung von Boolean-Werten.
- Menu-Renderer zur Darstellung der Komponente UISelectOne
- Listbox-Renderer für die Komponente UISelectMany

Aufgrund der Eigenheit von HTML, dass sämtliche Formulardaten, die in einem Eingabefeld beim Abschicken an den Server übermittelt sind, zuerst einmal als String übermittelt werden, findet im Framework eine entsprechende Datenkonvertierung statt. Mehr dazu ist in Kapitel 6.7 Datenkonvertierung nachzulesen.

7.8.1 HtmlInputText-Komponente

Die `HtmlInputText`-Komponente dient zur Eingabe von normalem, d.h. nicht formatiertem Text. Sie wird mit Hilfe des TextRenderers unter Verwendung des Tags `inputText` dargestellt. Das `inputText`-Tag ist in der Verwendung sicherlich das einfachste aller Eingabekomponenten. Es wird im Browser ein einfaches Eingabefeld angezeigt, in das der Benutzer einen einzeiligen alphanumerischen Text eingeben kann.

```
<h:form>
  <h:inputText value="- bitte Text eingeben -" />
  <h:commandButton value="Submit" action="success" />
</h:form>
```

Listing 7.14: Verwendung des inputText-Tags

Im Listing 7.14 wird ein einfaches Eingabefeld ausgegeben, das den Wert »– bitte Text eingeben –« als Standardbelegung bereits vorgefüllt hat. Als Ausgabe erscheint im Browser ein einfaches Eingabefeld:

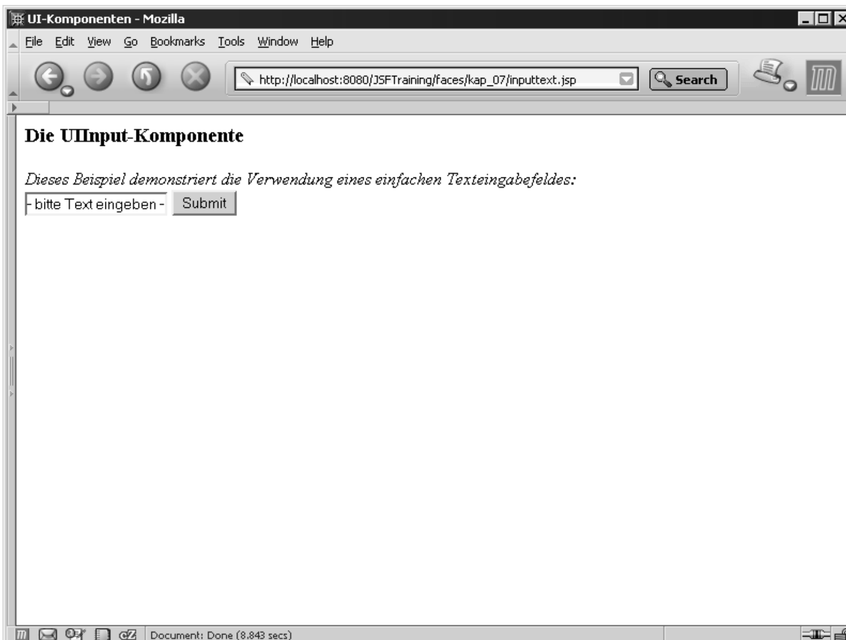


Abbildung 7.3: Ein einfaches Eingabefeld

Bei der Verwendung des `inputText`-Tags können u. a. zusätzlich folgende Attribute verwendet werden:

- `id`: Soll in einer Funktion auf genau diese Eingabekomponente selbst (also nicht auf den Wert) Bezug genommen werden, ist die Angabe einer Id notwendig. Wird dieses Attribut weggelassen, so wird eine automatische Id vergeben.
- `value`: Durch Angabe des `value`-Attributes kann direkt ein Wert für die Komponente angegeben werden bzw. es kann auch wieder durch die Expression Language auf eine Ressourcendatei oder ein Backing Bean verwiesen werden.
- `converter`: Bei Angabe eines Konverters wird dieser dazu benutzt, die Benutzereingabe entsprechend zu verarbeiten und die Daten in geeigneter Form in das Modellobjekt zu übernehmen. Die genaue Verwendung des Konverters ist in Kapitel 6.7 Datenkonvertierung erläutert.
- `size`: Das `size`-Attribut regelt die Größe des Eingabefeldes. Die Größe bezieht sich jedoch ausschließlich auf die Anzeigegröße, nicht auf die Anzahl der möglichen Eingabestellen.
- `maxlength`: Durch das `maxlength`-Attribut wird die maximale Anzahl an Eingabestellen begrenzt, die das Eingabefeld aufnehmen kann. Vorteil ist, dass bereits auf Clientseite eine Fehleingabe durch Begrenzung der Eingabelänge vermieden werden kann (z. B. wird ein Datum nur in Ausnahmefällen mehr als 10 Stellen haben).

Diese Auflistung ist natürlich in keinster Weise vollständig. Einen exakten Überblick über alle möglichen Attribute dieses Tags ist in der Taglib selbst oder in der mit der Referenzimplementierung mitgelieferten Taglib-Dokumentation zu finden.

Eingabe von Zahlen

Da in einem Eingabefeld sämtliche Inhalte zunächst als Text vorliegen, muss eine entsprechende Datenkonventierung stattfinden, wenn der Inhalt eines Feldes als Zahlenwert in ein Modellobjekt übernommen werden soll. Dazu kann als Attribut im `inputText`-Tag ein so genannter Konverter mitangegeben werden, der für eine korrekte Umsetzung des Textstrings in einen Zahlentyp verantwortlich ist.

```
<h:form>
  <h:inputText converter="javax.faces.Integer"
     value="#{Customer.zip}" />
  <h:commandButton value="Submit" action="continue" />
  <br><br>
  <h:messages />
</h:form>
```

Listing 7.15: Verwendung eines Konverters

Im Beispiel in Listing 7.15 wird dem Eingabefeld durch die Angabe von `converter="javax.faces.Integer"` mitgeteilt, dass die Benutzereingabe mit Hilfe eines Integer-Konverters umgewandelt und im Modellobjekt als Zahl gespeichert werden soll. Fehlt diese Angabe, versucht JSF selbst, den passenden Konverter zu finden und die Eingabe entsprechend umzuwandeln. In diesem Beispiel würde dies auch funktionieren, bei Datumsfeldern oder Zeitangaben ist die Angabe eines Konverters jedoch immer notwendig.

Eingabe von Datumswerten

Bei Datumswerten ist das Verfahren der Datenkonvertierung ein wenig aufwändiger, wenn man nur die verschiedenen Möglichkeiten bedenkt, wie ein Datum durch einen Benutzer eingegeben werden kann:

- 7.4.2004
- 07.04.2004
- 04/07/2004
- 7. April 2004
- ...

Diese Liste zeigt nur eine kleine Auswahl an Möglichkeiten, die für eine Datumseingabe vorhanden sind. Natürlich muss durch die Webanwendung vorgegeben werden, wie ein Benutzer das Datum einzugeben hat (z. B. in einem Hilfetext oder einem Erläuterungstext direkt neben dem entsprechenden Eingabefeld). Jedoch muss JSF mitgeteilt werden, in welcher Form die Informationen vorliegen, damit die Werte in ein gültiges `Date`-Objekt umgewandelt werden können.

```
<h:form>
  <h:inputText value="#{Customer.birth}">
    <f:convertDateTime dateStyle="short" />
  </h:inputText>
  <h:commandButton value="Submit" action="continue" />
  <h:messages />
</h:form>
```

Listing 7.16: Eingabe eines Datumswertes

In Listing 7.16 fällt auf, dass der Datumskonverter nicht als Attribut im `inputText`-Tag angegeben ist, sondern als eigenständiges Tag innerhalb des `inputText`-Tags verwendet wird. Dies hat den Hintergrund, dass zusätzlich zur Angabe des Konverters weitere Optionen zur Konvertierung angegeben werden müssen, im vorliegenden Beispiel das Attribut `dateStyle`. In diesen Fällen, in denen zusätzliche Angaben dem Konverter mitgeteilt werden müssen, wird dieser nicht mehr als Attribut innerhalb des Eingabetags angegeben, sondern als eigenständiges Tag verschachtelt im Eingabetag.

Weitere Optionen zum Datumskonverter sind in 6.7 Datenkonvertierung zu finden.

7.8.2 HtmlInputSecret-Komponente

Bei Eingabe von Passwörtern ist es oftmals nicht erwünscht, dass diese in Klarschrift direkt auf dem Bildschirm angezeigt werden. Vielmehr soll an Stelle der eingegebenen Zeichen ein *-Zeichen erscheinen. Dazu kann das `inputSecret-Tag` verwendet werden, das auf die Komponente `HtmlInputSecret` zurückgreift.

```
<h:inputSecret value="" />
```

Beim Rendern in HTML wird daraus ein `<input type="password">`-Element erzeugt. Damit kann ein Benutzer sein Passwort eingeben, ohne darauf achten zu müssen, ob ihm nicht jemand über die Schulter schaut und das Passwort vom Bildschirm ablesen kann.

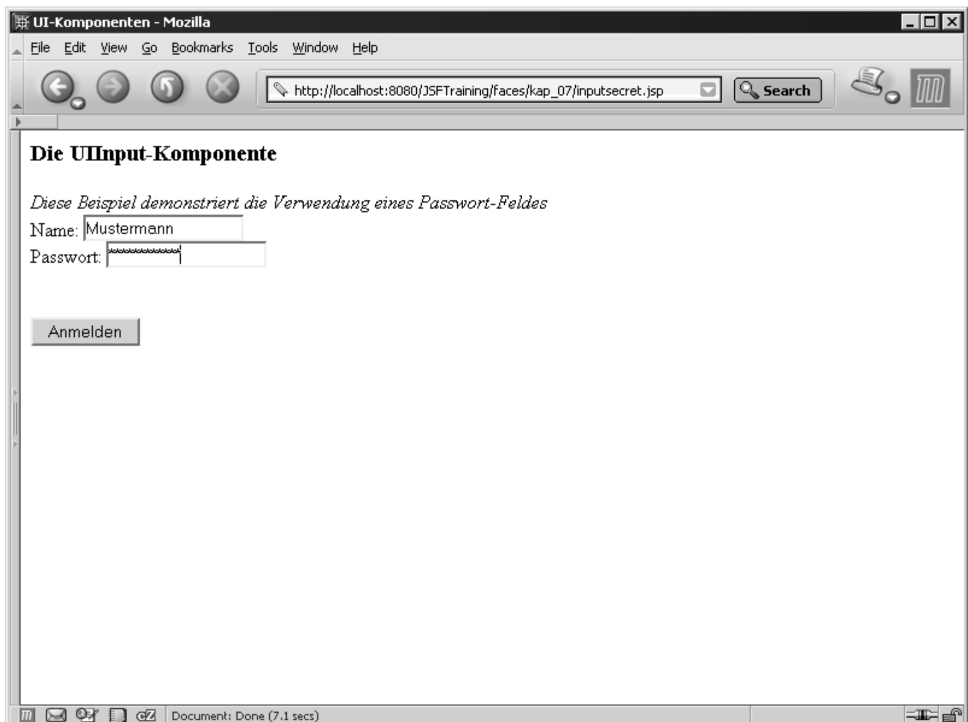


Abbildung 7.4: Das inputSecret-Tag

7.8.3 HtmlInputHidden-Komponente

In vielen Webanwendungen werden Variablen, die für einen Ablauf in der Anwendung wichtig sind, direkt in einem Formular im Browser abgespeichert. Damit der Benutzer diese Daten weder sehen noch verändern kann, kommen hierbei versteckte Eingabefelder zum Einsatz. Diese ähneln vom Verhalten einem normalen Eingabefeld: Sie enthalten Werte, die beim Abschicken des Fomulares im Request an den Server übertragen werden. Der Hauptunterschied liegt jedoch darin, dass versteckte Eingabefelder im Browser nicht sichtbar sind. Allerdings können etwas versiertere Anwender die Felder über die Quelltextansicht des Browsers dennoch ersehen. Versteckte Eingabefelder werden oftmals auch von JavaScript-Routinen verwendet, um Kontrollvariablen oder Kontrollwerte darin abzulegen.

Für die Verwendung steht das Tag `inputHidden` bereit, das auch von der sonstigen Parametrisierung der eines normalen Eingabefeldes entspricht.

7.8.4 HtmlInputTextArea-Komponente

Für die Eingabe größerer Textpassagen ist ein einfaches Eingabefeld nicht geeignet. HTML kennt hierfür das Tag `textarea`, mit dem ein mehrzeiliges und mehrspaltiges Eingabefeld festgelegt werden kann. In JSF wird dieses Element durch das Tag `inputTextarea` dargestellt. Hierbei kann über die Attribute `cols` und `rows` die Anzahl der Spalten und Zeilen mitangegeben werden.

```
<h:form>
  <h:inputTextarea value="" cols="40" rows="8" />
  <h:commandButton value="Submit" action="save" />
</h:form>
```

Listing 7.17: Verwendung des Tags inputTextarea

Mit Hilfe des Attributs `cols` wird angegeben, wie breit das gesamte Textfeld dargestellt werden soll, mit Angabe von `rows` werden die Zeilen, sprich die Höhe des Textfeldes angegeben. Ein eventueller Scrollbalken auf der rechten Seite wird bei Bedarf automatisch ergänzt, wobei dies keine JSF-Funktionalität darstellt, sondern bereits durch HTML bzw. den Browser erfolgt.

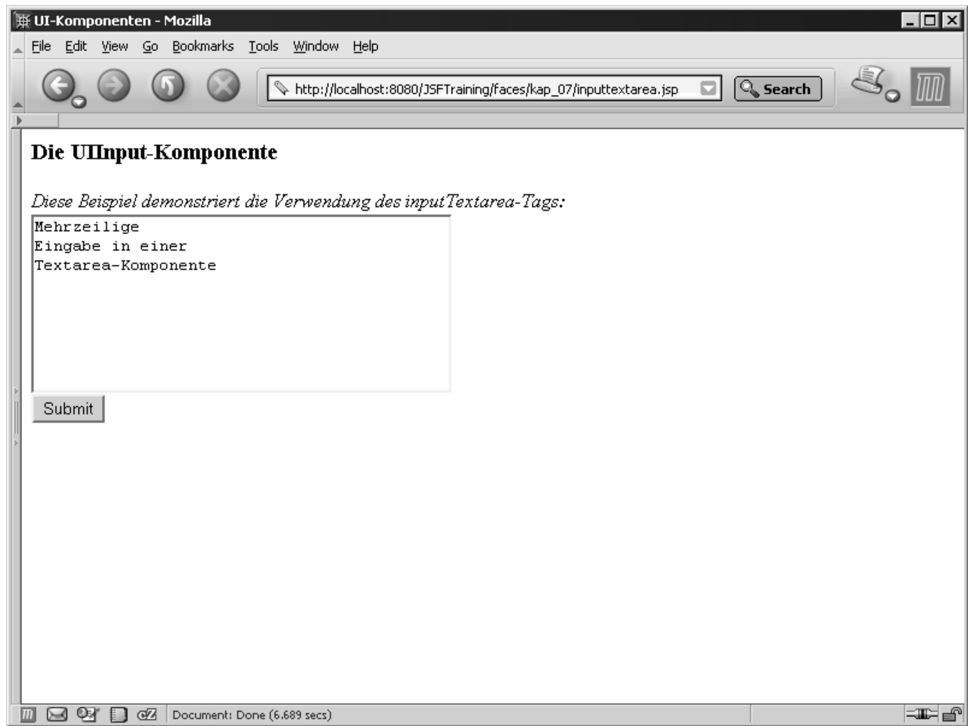


Abbildung 7.5: Das inputTextarea-Tag

7.9 UISelectBoolean-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UISelectBoolean`
HTML-Komponente	`javax.faces.component.html.HtmlSelectBooleanCheckbox`
Renderer	`javax.faces.Checkbox`
Tag	`<h:selectBooleanCheckbox>`

Erläuterung

Die `UISelectBoolean`-Komponente dient zur Darstellung von Wahrheitswerten (Boolean-Datentyp) in Formularen. `UISelectBoolean`-Komponenten werden in HTML daher als Auswahlboxen (Checkboxen) gerendert. Dazu existiert wieder die HTML-spezifische Komponente `HtmlSelectBooleanCheckbox`. Ist die Checkbox ausgewählt, entspricht dies dem Wert `true`, nicht selektiert dem Wert `false`.

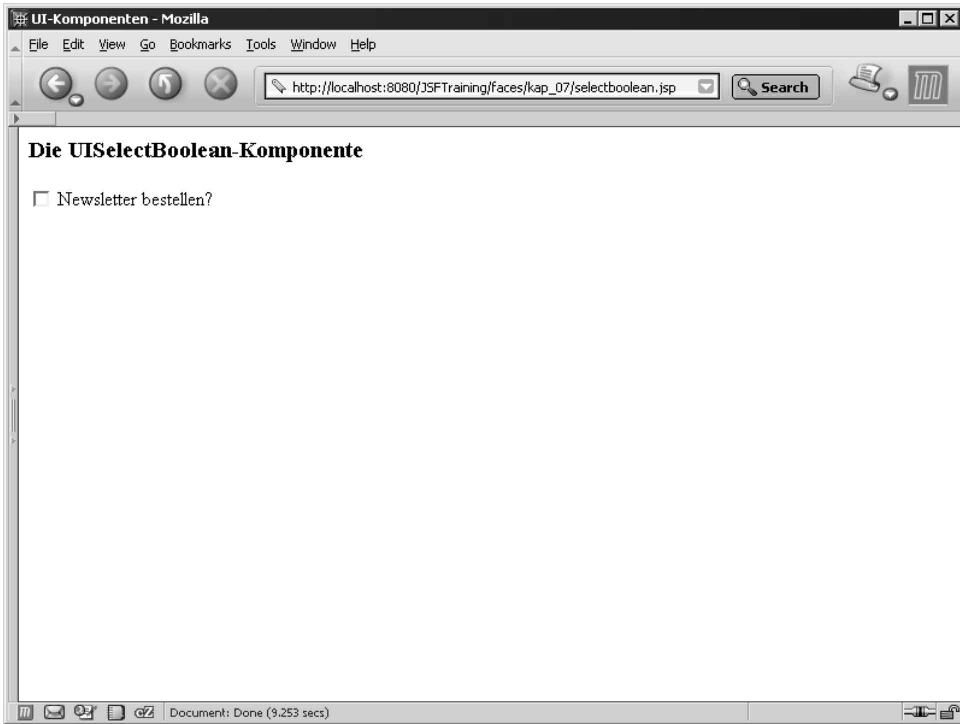


Abbildung 7.6: Einsatz der UISelectBoolean-Komponente

Die `UISelectBoolean`-Komponente ist auch eine Eingabekomponente und erbt daher von `javax.faces.component.UIInput`.

> Gerade an dieser Komponente sieht man eindeutig wieder den Vorteil des JSF-Frameworks hinsichtlich der Datenkonvertierung und des Setzens der korrekten Werte in das Modellobjekt: In der klassischen JSP- oder Servletprogrammierung konnte man Parameter, die durch einen Request an den Server übermittelt wurden, mit der Methode `request.getParameter( Parametername )` abfragen. Dies funktionierte bei Textparametern aus normalen Eingabefeldern meist auch problemlos. Bei Checkboxen gibt es die Besonderheit, dass falls die Checkbox nicht ausgewählt war, die Abfrage im Request nicht `false` ergab, sondern null zurückgeliefert wurde. Dies führte oftmals zu Programmfehlern, wenn der Boolean-Datentyp nicht jedes Mal gesondert behandelt wurde. Durch das JSF-Framework wird einem diese Arbeit abgenommen und automatisch der passende Wert in das Modellobjekt übernommen.

```
<h:form>
  <h:selectBooleanCheckbox id="selectNewsletter"
      value="#{Order.isNewsletter}" />
  <h:outputLabel for="selectNewsletter">
```

```
      <h:outputText value="Newsletter bestellen?"/>
    </h:outputLabel>
  </h:form>
```

Listing 7.18: Verwendung der HTMLISelectBoolean-Komponente

Im Listing 7.18 wird der Boolean-Wert `Order.isNewsletter` des Modellobjekts mit Hilfe einer Checkbox dargestellt. Ein Attribut, das die Bezeichnung der Checkbox angibt, ist in diesem Tag nicht vorgesehen. Um dennoch einen Text neben der Komponente anzuzeigen, kann ein einfaches `outputText`-Tag verwendet werden. Im Beispiel wird dabei auch nochmals die Verwendung des Tags `outputLabel` deutlich. Es zeigt an, dass die Bezeichnung *Newsletter bestellen* in direktem Zusammenhang zu der Checkbox mit der Id *selectNewsletter* zu sehen ist. Für die Darstellung im Browser hat diese Angabe zwar keine direkte Auswirkung, dennoch sieht die HTML-Spezifikation 4.0 eine genauere Beschreibung der Elemente vor. Es ist daher zurzeit noch als rein informative Angabe zu sehen, die in den HTML-Quellcode integriert wird, jedoch noch keine unmittelbare Wirkung erzielt. Ein kleiner Vorteil ist es sicherlich, dass der generierte HTML-Quellcode durch diese Angabe etwas deutlicher lesbar wird, da genau nachvollzogen werden kann, welches Textfeld zu welcher Checkbox gehört. Eventuell können künftig Programme oder Browser solche Angaben auch interpretieren und dementsprechend wiederum graphisch umsetzen.

7.10 UISelectOne-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UISelectOne`
HTML-Komponente	`javax.faces.component.hmtl.HtmlSelectOneListbox`
	`javax.faces.component.html. HtmlSelectOneMenu`
	`javax.faces.component.html. HtmlSelectOneRadio`
Renderer	`javax.faces.Listbox`
	`javax.faces.Menu`
	`javax.faces.Radio`
Tag	`<h:selectOneRadio>`
	`<h:selectOneListbox>`
	`<h:selectOneMenu>`

Erläuterung

Mit der `UISelectOne`-Komponente wird dem Benutzer eine Auswahl von Möglichkeiten vorgeschlagen, von denen *genau eine* selektiert werden kann. Häufig werden zur Darstellung einer Auswahlmöglichkeit, bei der genau ein Element selektiert werden kann, so genannte Radiobuttons verwendet. Es ist jedoch genauso gut möglich, eine Darstellung als Listbox oder Drop-Down-Menü zu wählen. Dies regeln die einzelnen Renderer bzw. die HTML-Komponentenklassen. Die `UISelectOne`-Komponente gehört auch zu den Eingabekomponenten und erbt daher von `javax.faces.component.UIInput`.

Als Beispiel dient eine Personenbeschreibung, die in einem Formular von einem Benutzer ausgefüllt werden muss. Die Eingabe eines Landes ist dabei auf die Länder Österreich, Schweiz und Deutschland beschränkt. Von diesen drei Ländern muss der Benutzer genau ein Herkunftsland angeben.

```
package com.edu.jsf.bsp.bean;

import java.util.ArrayList;

import javax.faces.component.SelectItem;

/**
 * Klasse für ein einfaches Personen-Bean
 */
public class PersonBean {

  private String firstname;
  private String lastname;
  private String street;
  private int zip;
  private String city;
  private String country;

  private ArrayList possibleCountries;

  /**
   * Konstruktor
   */
  public PersonBean() {
    possibleCountries = new ArrayList(3);
    possibleCountries.add(
      new SelectItem("DE", "Deutschland",
                     "Eintrag für Deutschland") );
    possibleCountries.add(
      new SelectItem("AT", "Österreich",
                     "Eintrag für Österreich") );
    possibleCountries.add(
      new SelectItem("CH", "Schweiz", "Eintrag für Schweiz") );
  }
```

```
/**
 * Getter-Methoden
 */
public String getCity() {
  return city;
}

public String getFirstname() {
  return firstname;
}

public String getLastname() {
  return lastname;
}

public String getStreet() {
  return street;
}

public int getZip() {
  return zip;
}

public String getCountry() {
  return country;
}

/**
 * liefert alle möglichen Länder, die zur
 * Auswahl stehen, zurück
 */
public ArrayList getPossibleCountries() {
  return possibleCountries;
}

/**
 * Setter-Methoden
 */
public void setFirstname(String string) {
  firstname = string;
}

public void setLastname(String string) {
  lastname = string;
}

public void setStreet(String string) {
  street = string;
}

public void setZip(int i) {
```

```
    zip = i;
  }

  public void setCity(String string) {
    city = string;
  }

  public void setCountry(String string) {
    country = string;
  }

  /**
   * setzt die Möglichkeiten für die Länderauswahl
   */
  public void setPossibleCountries(ArrayList list) {
    possibleCountries = list;
  }
}
```

Listing 7.19: Ein einfaches Bean für eine Person

Der ausgewählte Wert für das Herkunftsland ist in `country` hinterlegt, die Wahlmöglichkeiten selbst sind in der ArrayList `possibleCountries` gespeichert. Diese wird im Konstruktor des Beans initialisiert und befüllt. Das Befüllen der ArrayList im Konstruktor ist natürlich nur für dieses Kurzbeispiel akzeptabel. Üblicherweise wird solch eine Liste über eine Datenbank gesteuert, aus der die möglichen Wahlmöglichkeiten dynamisch ausgelesen werden.

Das abgebildete `PersonBean` dient für die folgenden Abschnitte als Basis, um die verschiedenen Darstellungsformen zu veranschaulichen.

7.10.1 HTMLSelectOneRadio-Komponente

Zunächst wird die Länderauswahl aus Listing 7.19 mit Hilfe eines Radiofeldes dargestellt. Die drei Länder Deutschland, Österreich und Schweiz werden je einzeln als Radiofeld dargestellt, von denen der Benutzer genau eine Auswahl treffen kann. Dabei kommt die `HtmlSelectOneRadio`-Komponente zum Einsatz, die als Standardrenderer den Renderer `Radio` verwendet.

```
<h:form>
  ...
  <h:selectOneRadio id="selCountry" value="#{Person.country}">
    <f:selectItems value="#{Person.possibleCountries}" />
  </h:selectOneRadio>
  ...
</h:form>
```

Im `selectOneRadio`-Tag wird durch Angabe des Attributs `value` angegeben, an welcher Stelle im Modellobjekt der ausgewählte Wert abgespeichert werden soll. Die eigentlichen Werte, die zur Auswahl stehen, werden durch das Tag `selectItems` (Achtung, Mehrzahl!) angegeben. Die `value`-Angabe hierbei greift auf eine Membervariable zu, die die einzelnen Wahlmöglichkeiten in einer `ArrayList` gespeichert hat. Wichtig ist, dass die Objekte, die in der `ArrayList` vorhanden sind, Objekte der Klasse `selectItem` sind. Details zur Verwendung der `selectItem`-Klassen folgen in den nächsten Abschnitten.

Zu beachten ist ebenfalls, dass das Tag `selectItems` nicht in der `HTML_basic` Taglib enthalten ist, sondern in der `jsf_core` Taglib. Daher ist dem Tag auch der Namensraum `f` voranzustellen.

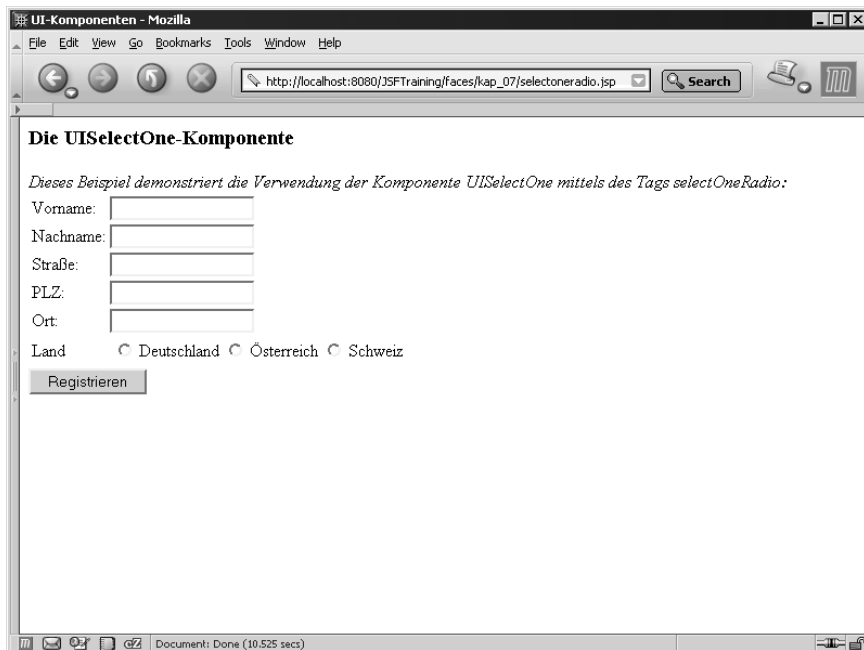


Abbildung 7.7: Verwendung des Tags selectOneRadio

Statt der Angabe der Auswahlmöglichkeiten über eine `ArrayList` können die einzelnen Wahlmöglichkeiten auch direkt in der JSF-Seite angegeben werden.

```
<h:selectOneRadio id="selCountry" value="#{Person.country}">
  <f:selectItem itemValue="DE" itemLabel="Deutschland" />
  <f:selectItem itemValue="AT" itemLabel="Österreich" />
  <f:selectItem itemValue="CH" itemLabel="Schweiz" />
</h:selectOneRadio>
```

Listing 7.20: Direkte Angabe der einzelnen Optionen

Welche Möglichkeit verwendet wird, um Auswahlwerte einer `UISelectOne`-Komponente hinzuzufügen, sollte immer im Einzelfall entschieden werden. Bei Verwendung des `selectItems`-Attributs ist sicherlich von Vorteil, dass dynamisch ohne Probleme weitere Auswahlmöglichkeiten hinzugefügt werden können. So kann die Anzahl durch eine Funktion im Programmablauf festgelegt werden.

Ist die Anzahl der Wahlmöglichkeiten recht statisch wie im Listing 7.20, in dem eben nur eine Auswahl von drei Ländern angeboten werden, hat die `selectItem`-Angabe den Vorteil, dass hierbei keinerlei Java-Code geschrieben werden muss, sondern alle notwendigen Angaben direkt im Tag in der JSF-Seite vorgenommen werden können. Dabei kann z.B. auch ein Webdesigner die Auswahlmöglichkeiten bestimmen, ohne Programmierkenntnisse in JSF zu haben.

7.10.2 HTMLSelectOneListbox-Komponente

Wie der Name bereits vermuten lässt, wird bei der `HtmlSelectOneListbox`-Komponente eine Listbox mit den möglichen Auswahlelementen angezeigt. Der Benutzer kann aus der dargestellten Liste wiederum nur ein Element selektieren. Die Größe der Liste richtet sich dabei nach der Anzahl der vorhandenen Auswahlelemente. Es werden immer *alle* Auswahlelemente angezeigt.

```
<h:selectOneListbox id="selCountry" value="#{Person.country}">
  <f:selectItem itemValue="DE" itemLabel="Deutschland" />
  <f:selectItem itemValue="AT" itemLabel="Österreich" />
  <f:selectItem itemValue="CH" itemLabel="Schweiz" />
</h:selectOneListbox>
```

Listing 7.21: Das Tag selectOneListbox

> Bei fast allen UI-Komponenten gibt es das Attribut `disabled`. Damit können UI-Komponenten im Browser als deaktiviert dargestellt werden. Man kann den Zustand auch als *read-only* bezeichnen. Im Falle der Listbox aus Abbildung 7.8 würde zwar die komplette Liste der Wahlmöglichkeiten angezeigt, der Benutzer hätte jedoch nicht die Möglichkeit, einen Wert zu selektieren. Die Verwendung des Attributes `disabled` wird häufig dann benutzt, wenn Informationen angezeigt und damit gelesen werden dürfen, eine Berechtigung für das Verändern von Daten über die Oberfläche jedoch nicht gewährt ist.

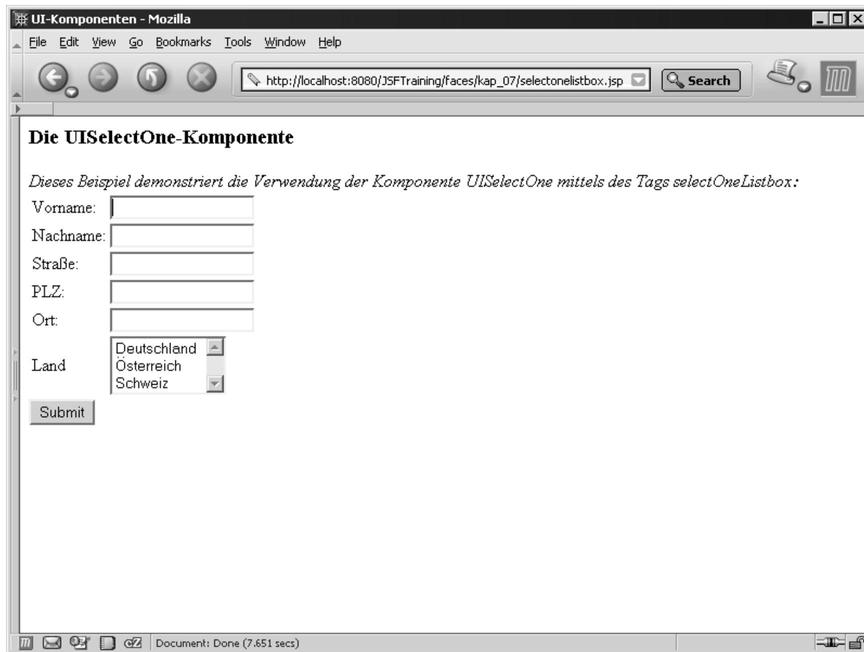


Abbildung 7.8: Das selectOneListbox-Tag

7.10.3 HTMLSelectOneMenu-Komponente

Bei der `HTMLSelectOneMenu`-Komponente werden die Einträge in einer Drop-Down-Box angezeigt. Dabei kommt das Tag `selectOneMenu` zum Einsatz. Der Hauptunterschied zur Darstellung mittels `selectOneListbox` besteht lediglich darin, dass nicht alle Auswahlmöglichkeiten immer angezeigt werden, sondern erst nach Anklicken ein entsprechendes Menü geöffnet wird. Aus der geöffneten Menüliste kann dann ein Element gewählt werden.

```
<h:selectOneMenu value="#{Person.country"}>
  <f:selectItems value="#{Person.possibleCountries}" />
</h:selectOneMenu>
```

Alle drei gezeigten Darstellungsformen einer `UISelectOne`-Komponente – Darstellung als Radiogruppe, als Listbox oder als Drop-Down-Box – haben den gleichen Informationsgehalt und die gleiche Programmlogik. Es ist deshalb allein vom Oberflächenkonzept abhängig, welche Darstellungsform für das jeweilige Formular am besten geeignet ist. Bei einer Radiogruppe ist von Vorteil, dass alle Wahlmöglichkeiten auf einmal zu sehen sind, bei einer Drop-Down-Box dagegen nicht. Dafür hat man allerdings bei letzterer Darstellungsform den Vorteil der Platzersparnis, was bei komplexen Oberflächen, bei denen sehr viel Informationen enthalten sind, durchaus wiederum von Vorteil sein kann.

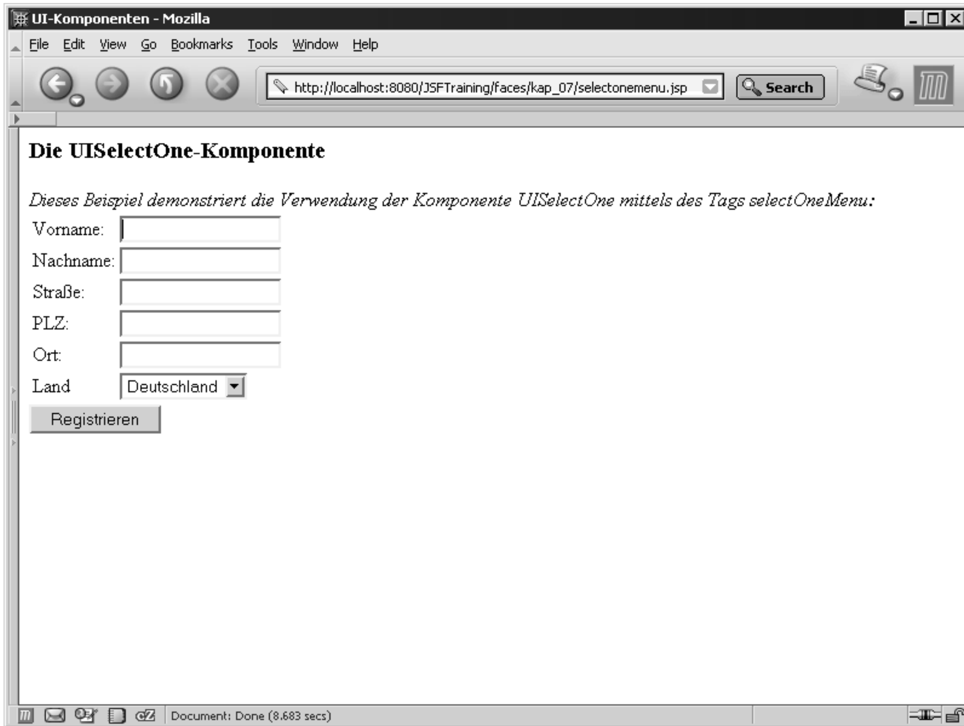


Abbildung 7.9: Einsatz des selectOneMenu-Tags

7.11 UISelectMany-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UISelectMany`
HTML-Komponente	`javax.faces.component.html.HtmlSelectManyCheckbox`
	`javax.faces.component. html.HtmlSelectManyListbox`
	`javax.faces.component. html.HtmlSelectManyMenu`
Renderer	`javax.faces.Checkbox`
	`javax.faces.Listbox`
	`javax.faces.Menu`
Tag	`<h:selectManyCheckbox>`
	`<h:selectManyListbox>`
	`<h:selectManyMenu>`

Erläuterung

Oftmals ist es notwendig, dass von einer Anzahl von Wahlmöglickeiten mehrere durch einen Benutzer zu selektieren sind. Dazu ist die Komponente `UISelectMany` vorgesehen. Die Verwendung der drei möglichen Darstellungsformen – als Listbox, als DropDown-Box oder als Checkboxliste – erfolgt analog zur `UISelectOne`-Komponente. Der offensichtlichste Unterschied liegt allein darin, dass bei der `UISelectMany`-Komponente keine Radiogruppe, sondern eine Checkboxliste als dritte Darstellungsform angeboten wird. Dies ist jedoch keine JSF-spezifische Einschränkung. Per Definition sind Einfachauswahlen nur in einer Radiogruppe möglich, wogegen Mehrfachselektionen über Checkboxen realisiert werden.

Zur Verdeutlichung dient eine Umfragekomponente. Benutzer geben auf einer Webseite ein paar statistische Daten an. Zu den Vor- und Nachnamen können aus einer Reihe von Interessen eigene Interessen ausgewählt werden. Entscheidend hierbei ist, dass ein Befragter mehrere Interessen angeben kann, sich somit nicht nur für eine Möglichkeit entscheiden muss.

```
package com.edu.jsf.bsp.bean;

import java.util.ArrayList;

import javax.faces.component.SelectItem;

/**
 * ein Bean zur Erfassung einer Umfrage
 */
public class SurveyBean {

  private String firstname;
  private String lastname;

  private ArrayList interests;
  private ArrayList allInterests;

  /**
   * Konstruktor
   */
  public SurveyBean() {
    allInterests = new ArrayList(4);
    allInterests.add( new SelectItem("1", "Tauchen",
                     "Eintrag für Tauchen") );
    allInterests.add( new SelectItem("2", "Angeln",
                     "Eintrag für Angeln") );
    allInterests.add( new SelectItem("3", "Segeln",
                     "Eintrag für Segeln") );
    allInterests.add( new SelectItem("4", "Schwimmen",
                     "Eintrag für Schwimmen") );
```

```
    }

    /**
     * Getter-Methoden
     */
    public String getFirstname() {
      return firstname;
    }

    public String getLastname() {
      return lastname;
    }

    public ArrayList getInterests() {
      return interests;
    }

    public ArrayList getAllInterests() {
      return allInterests;
    }

    /**
     * Setter-Methoden
     */
    public void setFirstname(String string) {
      firstname = string;
    }

    public void setLastname(String string) {
      lastname = string;
    }

    public void setAllInterests(ArrayList arraylist) {
      allInterests = arraylist;
    }

    public void setInterests(ArrayList arraylist) {
      interests = arraylist;
    }
  }
```

Listing 7.22: Ein Umfrage-Bean

Die möglichen Interessen, die ausgewählt werden können, werden wieder in einer `ArrayList` vorgehalten, die im Konstruktor befüllt wird. Auch hier gilt wieder, dass das Befüllen der `ArrayList` im Konstruktor für dieses Kurzbeispiel eine akzeptable Vorgehensweise ist. In einer realen Anwendung kämen diese Angaben zumeist aus einer Datenbank.

Wichtig ist ebenfalls der Hinweis auf die Eigenschaft `interests`, in der die gewählten Interessen gespeichert werden. Da ein Benutzer mehrere Angaben treffen kann, muss die Eigenschaft ebenfalls vom Typ `ArrayList` sein.

Diese `Bean`-Klasse dient den nachfolgenden Beispielen als Basis, um die verschiedenen Darstellungsformen zu veranschaulichen.

7.11.1 HTMLSelectManyCheckbox-Komponente

Die `HTMLSelectManyCheckbox` dient zur Darstellung einer Mehrfachauswahl in Form einer Checkboxliste. Auch bei der Klasse `SurveyBean` liegen im Bean wiederum zwei Eigenschaften vom Typ `ArrayList` vor. Die ArrayList `allInterests` beinhaltet die Anzahl aller möglichen Interessen, die vom Benutzer ausgewählt werden können. Die ArrayList wird in diesem Beispiel im Konstruktor mit vier Werten befüllt. Genauso gut kann eine Initialisierungsroutine geschrieben werden, die alle möglichen Werte z.B. aus einer Datenbank ausliest. In der ArrayList `interests` sind dagegen diejenigen Werte gespeichert, die vom Benutzer über die Oberfläche ausgewählt wurden.

```
<h:selectManyCheckbox value="#{Survey.interests}">
  <f:selectItems value="#{Survey.allInterests}" />
</h:selectManyCheckbox>
```

Listing 7.23: Einbindung des Tags selectManyCheckbox

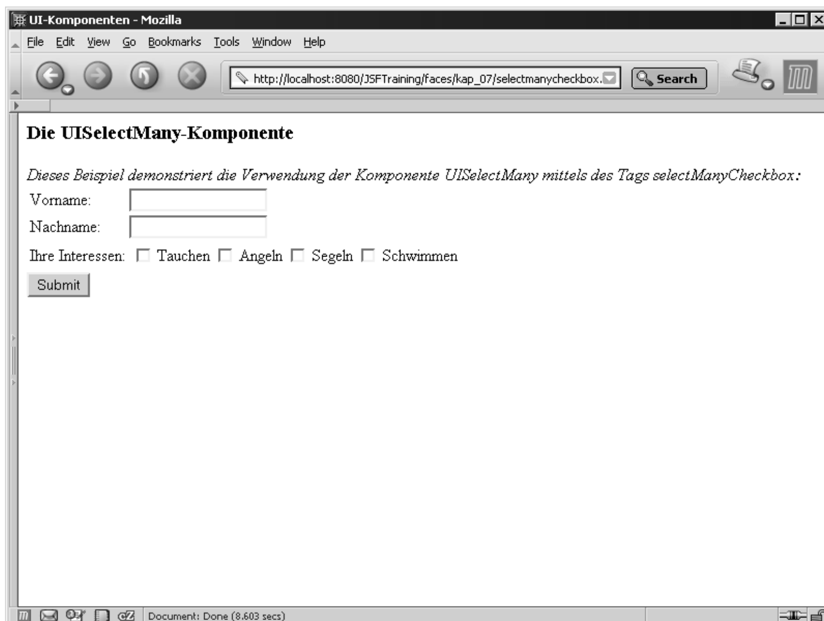


Abbildung 7.10: Mehrfachauswahl

In Listing 7.23 wird ebenfalls wieder das Tag `selectItems` verwendet. Natürlich ist es ebenso möglich, die einzelnen Werte anstatt über die `ArrayList`, die sich hinter `selectItems` befindet, über die direkte Angabe in der JSF-Seite anzugeben.

```
<h:selectManyCheckbox value="#{Survey.interests}">
  <f:selectItem itemLabel="Tauchen" itemValue="1" />
  <f:selectItem itemLabel="Angeln" itemValue="2" />
  <f:selectItem itemLabel="Segeln" itemValue="3" />
  <f:selectItem itemLabel="Schwimmen" itemValue="4" />
</h:selectManyCheckbox>
```

Entgegen einer Auswahl in Form von Radiobuttons können durch die Anzeige mehrerer Checkboxen verschiedene Optionen gleichzeitig ausgewählt werden. Somit ist von Fall zu Fall zu unterscheiden, ob ein Benutzer nur eine Option oder mehrere Optionen wählen kann. Danach kann entweder das Tag `selectOneRadio` oder `selectManyCheckbox` verwendet werden.

7.11.2 HTMLSelectManyMenu-Komponente

Bei der `HTMLSelectManyMenu`-Komponente kommt zur Darstellung eine Liste der Größe 1 zum Einsatz. In der Darstellung wird dabei das `selectManyMenu`-Tag verwendet. Zur Veranschaulichung der Darstellungsform wird auch in diesem Beispiel das Umfrage-Bean verwendet. Die Interessen, die vom Benutzer ausgewählt werden können, sind diesmal jedoch in Form einer Liste dargestellt.

```
<h:selectManyMenu value="#{Survey.interests">
  <f:selectItems value="#{Survey.allInterests}" />
</h:selectManyMenu>
```

Listing 7.24: Das selectManyMenu-Tag

Im Listing 7.24 ist die Größe der Liste automatisch auf 1 ausgelegt. Der Benutzer kann unter Verwendung der Taste `Strg` mehrere Einträge selektieren. Ist die Anzahl der möglichen Elemente größer als die Anzahl der Elemente, die auf einmal in der Liste zu sehen sind (was eigentlich der Normalfall ist), erscheint zusätzlich ein Scrollbalken.

Eine klassische Drop-Down-Box lässt sich mit der `selectManyMenu`-Komponente nicht realisieren. Aufgrund der Tatsache, dass bei einer Drop-Down-Box immer nur ein Element ausgewählt sein kann (das, das gerade angezeigt wird), kann dies bei einer Wahlmöglichkeit von mehreren Elementen nicht funktionieren.

Verwendung der Klasse SelectItem

In den vorhergehenden Beispielen wurden für die `UISelectOne`- und `UISelectMany`-Komponenten häufig die Tags `selectItem` und `selectItems` verwendet. Ersteres findet sich auch in Form der Klasse `SelectItem` wieder. Eine Instanz der Klasse `SelectItem` stellt

eine Wahlmöglichkeit bei Auswahlfeldern dar. Ein `SelectItem` wird durch folgende Eigenschaften beschrieben:

- `Value`: ein Wert, der bei Auswahl in einer Komponente im Modellobjekt gespeichert wird
- `Label`: ein Bezeichnungstext, der in der Oberfläche angezeigt wird
- `Description`: ein kurzer beschreibender Text

Werden mehrere Instanzen von `SelectItem` in einem Objekt vom Typ `Array`, `Map`, `List` oder `Collection` zusammengefasst, können diese über das `selectItems`-Tag wiederum abgefragt werden.

Anmerkung:

`Array`, `Map`, `List` und `Collection` sind Interfaces und Teil des *Java Collections Frameworks*. Klassen, die z.B. das `Collection`-Interface implementieren, sind `List` oder `Set`. Beide können wiederum verwendet werden, um mehrere Instanzen von `selectItem` zusammenzufassen.

Im Bean selbst wiederum können einzelne Instanzen von der Klasse `SelectItem` erzeugt und abgespeichert werden.

```
allInterests = new ArrayList(4);
allInterests.add( new SelectItem("1", "Tauchen",
                  "Eintrag für Tauchen") );
allInterests.add( new SelectItem("2", "Angeln",
                  "Eintrag für Angeln") );
allInterests.add( new SelectItem("3", "Segeln",
                  "Eintrag für Segeln") );
allInterests.add( new SelectItem("4", "Schwimmen",
                  "Eintrag für Schwimmen") );
```

Listing 7.25: Liste für die Interessenauswahl

Listing 7.25 zeigt, dass in diesem Falle eine `ArrayList` mit einer Initialgröße von vier erzeugt wird. Daraufhin werden vier Elemente hinzugefügt. Wichtig ist, dass nur Objekte der Klasse `SelectItem` hinzugefügt werden. Wird beispielsweise ein String direkt einer `ArrayList` zugewiesen, entsteht zur Laufzeit zwar kein Fehler, jedoch erfolgt auch keine Darstellung der einzelnen Komponenten. Da leider keine `ClassCastException` oder Ähnliches geworfen wird, ist eine Fehlersuche meist aufwändig und mühsam (der Autor weiß dies aus eigener schmerzlicher Erfahrung).

7.11.3 HTMLSelectManyListbox-Komponente

Die `HTMLSelectManyListbox` wird in ihrer Präsentation auf HTML-Seiten ebenfalls wieder als Liste dargestellt – analog zur `HTMLSelectManyMenu`-Komponente. Es kommt jedoch der Renderer `Listbox` zum Einsatz. Dieser bewirkt, dass *mehrere* Elemente in der Listbox gleichzeitig sichtbar sein können. Durch Angabe eines zusätzlichen Attributs `size` kann die Größe der Liste angegeben werden. Übersteigt die Anzahl der Elemente die Listgröße, wird automatisch ein Scrollbalken hinzugefügt.

Abbildung 7.11: Verwendung des Tags selectManyListbox

Die `HtmlSelectManyListbox`-Komponente kommt meist dann zum Einsatz, wenn der Benutzer mehrere Auswahlelemente zugleich sehen soll. Die Auswahl mehrerer Elemente erfolgt ebenfalls wieder, indem die `Strg`-Taste gedrückt bleibt, während die einzelnen Elemente ausgewählt werden.

```
<h:selectManyListbox value="#{Survey.interests}">
  <f:selectItems value="#{Survey.allInterests}" />
</h:selectManyListbox>
```

Listing 7.26: Verwendung des selectManyListbox-Tags

7.12 UIOutput-Komponente

Übersicht

Komponentenklasse	javax.faces.component.UIOutput
HTML-Komponente	javax.faces.component.html.HtmlOutputText
	javax.faces.component.html. HtmlOutputFormat
	javax.faces.component.html. HtmlOutputLabel
	javax.faces.component.html. HtmlOutputLink
Renderer	javax.faces.Text
	javax.faces.Format
	javax.faces.Label
	javax.faces.Link
Tag	<h:outputText>
	<h:outputFormat>
	<h:outputLabel>
	<h:outputLink>

Erläuterung

Entgegen den Eingabekomponenten dienen Ausgabekomponenten lediglich der Darstellung von Werten. Varianten gibt es bei den Ausgabekomponenten nicht so viele wie bei den Eingabekomponenten, dennoch weist die Klasse `UIOutput` immerhin vier wichtige Unterklassen auf.

- `HtmlOutputText`
- `HtmlOutputLabel`
- `HtmlOutputMessage`
- `HtmlOutputLink`

Es sei zudem noch erwähnt, dass die Klasse `UIInput` ebenfalls von `UIOutput` ableitet. Eine Eingabekomponente ist letzten Endes auch nichts anderes als eine Komponente, die zur Ausgabe zusätzlich noch eine Eingabe entgegennimmt.

7.12.1 HtmlOutputText-Komponente

Die Komponente `HtmlOutputText` ist wohl die gebräuchlichste aller Ausgabekomponenten. Standardmäßig wird bei Verwendung der `outputText`-Tags der Textrenderer verwendet, der übergebene Werte als normalen Text ausgibt.

```
<h:form>
  <h:outputText value="Hallo Welt !!!" />
  <h:commandButton value="Submit" action="continue" />
</h:form>
```

Listing 7.27: Verwendung des Tags outputText

Das Listing 7.27 zeigt eine einfache Verwendung des `outputText`-Tags. Die Verwendung von Attributen erfolgt analog zur `UIInput`-Komponente und deren `inputText`-Tag. So kann bei Bedarf eine Id angegeben werden. Der Wert selbst kann wie in Listing 7.27 mit einem fixen Wert im `value`-Attribut angegeben werden oder aus dem Modellobjekt bezogen werden. Um Texte auszulagern, kann bei der Komponente ebenso wieder mit einer Ressourcendatei gearbeitet werden (vgl. Kapitel 6.13).

Bei der Ausgabe kann ebenfalls analog zu den Eingabekomponenten ein Konverter mit angegeben werden, der für eine passende Datenkonvertierung verantwortlich ist.

```
<h:form>
  Eingabe:<br>
  <h:inputText value="#{Customer.birth}">
    <f:convertDateTime dateStyle="short" />
  </h:inputText><br><br>
  Ausgabe:<br>
  <h:outputText value="#{Customer.birth}" /> <br><br>
  <h:commandButton value="Submit" action="continue" />
</h:form>
```

Listing 7.28: Ausgabe eines Datums

Das Listing 7.28 definiert auf derselben Seite zwei Felder, ein Eingabefeld und ein Ausgabefeld für das Geburtsdatum. Sobald eine Eingabe erfolgt ist und das Formular abgeschickt wurde, wird das Datum in das entsprechende Feld im Modellobjekt weggeschrieben. Da die Aktion `continue` in der Anwendungssteuerung nicht hinterlegt ist, wird nach dem Abschicken wieder auf dieselbe Seite verwiesen, auf der dann das eingegebene Datum in der Ausgabekomponente dargestellt wird.

Da bei der Ausgabekomponente kein Konverter angegeben ist, wird die Standardkonvertierung verwendet, die folgendes Ergebnis liefert:

Die Ausgabe in Abbildung 7.12 ist so natürlich nicht gewollt. Erst durch Angabe des Konverters erfolgt das gewünschte Ausgabeformat:

```
<h:outputText value="#{Customer.birth}">
  <f:convertDateTime dateStyle="short" />
</h:outputText>
```

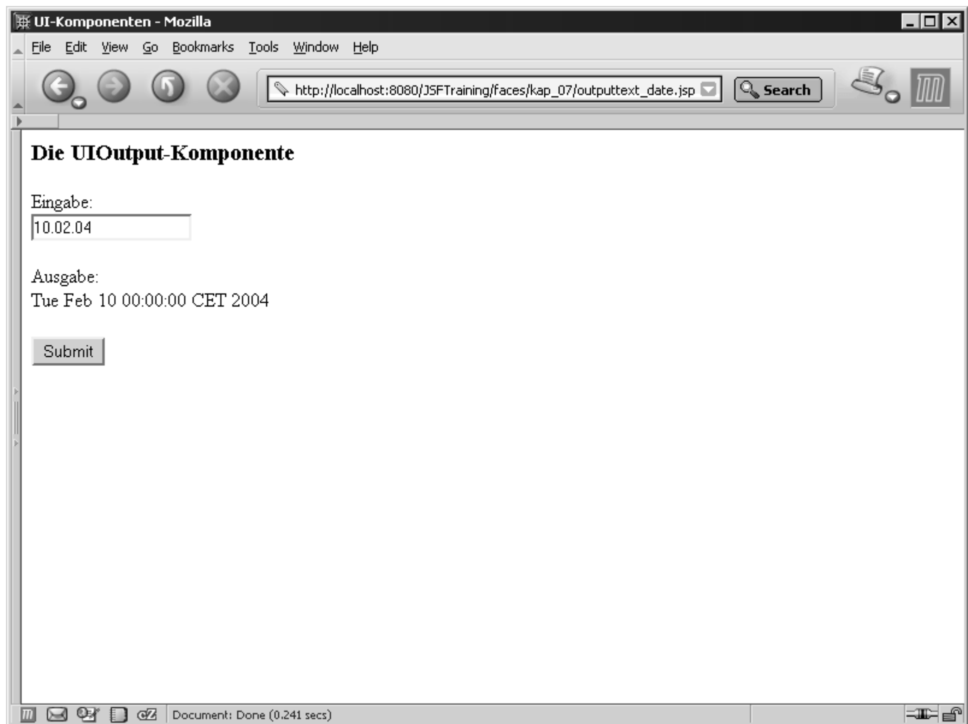


Abbildung 7.12: Standardausgabe eines Datums

Das Tag `convertDateTime` ist dasselbe Tag, das auch für die Eingabekomponenten verwendet wird. Es gelten somit die gleichen Attribute und Optionen, mit denen die Ausgabe eines Datums (und ebenso einer Zeitangabe) angepasst werden kann.

7.12.2 HtmlOutputLabel-Komponente

Mittels der `HtmlOutputLabel`-Komponente wird eine Verbindung zwischen einem Bezeichnerfeld sowie einem Ausgabefeld geschaffen. So kann es in Eingabeformularen auftauchen, dass zu einem speziellen Eingabefeld (z.B. einer Eingabe des Namens) ein Bezeichnerfeld zu finden ist, damit ein Benutzer auch weiß, was er in das leer stehende Eingabefeld einzutragen hat. Da somit ein inhaltlicher Zusammenhang zwischen dem Eingabefeld sowie dessen beschreibendem Text besteht, kann dies durch die HtmlOutputLabel-Komponente bzw. das `outputLabel`-Tag zum Ausdruck gebracht werden.

Als Resultat sollte das Bezeichnerfeld in der Nähe des eigentlichen Eingabefeldes angezeigt werden.

```
<h:outputLabel for="fullName">
  <h:outputText value="Ihr vollständiger Name:"/>
</h:outputLabel>
<h:inputText id="fullName" />
```

Listing 7.29: Verwendung des outputLabel-Tags

Das `outputLabel`-Tag verwendet ein Attribut `for`, das anzeigt, zu welcher Eingabekomponente das Label gehört. Der Wert, der im `for`-Attribut angegeben wird, muss dem Attribut `id` der eigentlichen Eingabekomponente entsprechen.

7.12.3 HtmlOutputFormat-Komponente

Zur Ausgabe von Meldungen allgemeiner Art stehen die Komponente `HtmlOutputFormat` sowie das dazugehörige Tag `outputFormat` zur Verfügung. Mit Hilfe des Tags `outputFormat` können Meldungen gemäß dem Message-Format-Schema ausgegeben werden. Dies bedeutet, dass eine Meldung mit Platzhaltern versehen werden kann und dadurch parametrisierbar ist. Die Parameter selbst können entweder als Wert in der Anwendung fest hinterlegt werden oder aber zur Laufzeit aus einem Modellobjekt bezogen werden.

Die `HtmlOutputFormat`-Komponente ist der `HtmlOutputText`-Komponente zwar sehr ähnlich, hat jedoch einen grundsätzlich anderen Verwendungsbereich. Während die `Html OutputText`-Komponente zur Ausgabe allgemeiner Texte und Werte verwendet wird, werden mit der `HtmlOutputFormat`-Komponente komplette Meldungstexte ausgegeben. Zudem sind diese Ausgabemeldungen parametrisierbar, was mit der `HtmlOutputText`-Komponente nicht möglich ist.

```
<h:form>
  <h:outputFormat
       value="Guten Tag, {0} {1}. " >
    <f:param id="prmFirstname" value="Peter"/>
    <f:param id="prmLastname" value="Mustermann" />
  </h:outputFormat>
</h:form>
```

Listing 7.30: Verwendung des Tags outputMessage

In Listing 7.30 wird eine Meldung ausgegeben, der zwei Parameter übergeben werden. Die Syntax ist dahingehend aufgebaut, dass die Parameternummer mit einer geschwungenen Klammer umgeben wird. Die Parameter werden als verschachteltes Element mittels des `param`-Tags übergeben. Beim Rendering der Seite wird als Ausgabe dann der komplette String

```
Guten Tag, Peter Mustermann
```

zusammengesetzt. Natürlich ist es auch hier wieder möglich, die Werte für die Parameter aus einer Ressourcendatei zu beziehen oder aus einem Modellobjekt, das beispielsweise in der Session abgelegt ist.

```
<h:form>
  <h:outputFormat value="Guten Tag, {0} {1}. " >
    <f:param value="#{Customer.firstname}"/>
    <f:param value="#{Customer.lastname}" />
  </h:outputFormat>
</h:form>
```

Listing 7.31: Zugriff auf Modellobjekte bei Nutzung des param-Tags

7.12.4 HtmlOutputLink-Komponente

Bei der Verwendung der `UICommand`-Komponenten wurde bereits eine Komponente erläutert, deren Repräsentation einen Link darstellt. Die `HtmlOutputLink`-Komponente sowie das zugehörige Tag `outputLink` sind jedoch nicht dazu gedacht, Aktionen innerhalb einer Anwendung auszulösen. Die `HtmlOutputLink`-Komponente stellt vielmehr einen gewöhnlichen Link zu einer beliebigen Adresse dar.

```
<h:form>
  <h:outputLink value="http://www.sun.com">
    <h:outputText value="Link zu Sun" />
  </h:outputLink>
</h:form>
```

Listing 7.32: Verlinkung zu einer externen Url

Das Ergebnis dieser Angabe ist ein einfacher Hyperlink. Die eigentliche Adresse, zu der ein Link hinführen soll, wird durch das `value`-Attribut übergeben, der Text, der für den Link angezeigt wird, ist als verschachteltes Element anzugeben. Im Falle des Listing 7.32 wird ein `outputText`-Tag dazu verwendet.

Die Angabe zusätzlicher Parameter ist auch bei dieser Form von Verlinkung möglich. Allerdings wird in diesem Falle ein so genannter Query-String aufgebaut, d.h. die einzelnen Parameter werden in der Url weitergegeben.

```
<h:form>
  <h:outputLink value="http://www.irgendeine-url.de">
    <h:outputText value="Link" />
    <f:param id="userid" name="name" value="4711" />
    <f:param id="language" name="name" value="de" />
  </h:outputLink>
</h:form>
```

Listing 7.33: Verwendung von Parametern

In Listing 7.33 wird als Ausgabe ein Link mit der Url

http://www.irgendeine-url.de/?name=4711&name=de

erzeugt. Die im Parameter-Tag übergebenen Werte werden mittels des `?`- und `&`-Zeichens an die eigentliche Adresse angehängt und können somit weitere Informationen an eine externe Webanwendung übergeben.

7.13 UIMessage- und UIMessages-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UIMessage` `javax.faces.component.UIMessages`
HTML-Komponente	`javax.faces.component.html.HtmlMessage` `javax.faces.component.html.HtmlMessages`
Renderer	`javax.faces.Message` `javax.faces.Messages`
Tag	`<h:message>` `<h:messages>`

Erläuterung

Eine wichtige Rolle spielen die `UIMessage`- und die `UIMessages`-Komponente. Beide haben die Aufgabe, eventuelle Fehlermeldungen, die in einer Anwendung auftreten können, zur Anzeige zu bringen. Ist beispielsweise für ein Eingabefeld ein Wertebereich von 0 bis 100 vorgesehen, wird bei Eingabe einer Zahl größer 100 automatisch eine entsprechende Fehlermeldung erzeugt. Dies passiert automatisch bei Verwendung von Standardvalidatoren. Es wird im Fehlerfalle auf der Seite verblieben und nicht auf eine eventuelle Folgeseite weitergeleitet. Dafür wird in der Seite zusätzlich eine Fehlermeldung an der Stelle ausgegeben, an der ein entsprechendes Tag gesetzt ist.

Auch im Falle der `UIMessage`-Komponente existiert wieder eine HTML-spezifische Komponente: `HtmlMessage`- bzw. `HtmlMessages`-Komponente.

Wie bereits im Kapitel 6.6 beschrieben, können mehrere `message`-Tags in einer Seite vorhanden sein, wobei die Tags jeweils für spezielle Komponenten zuständig sind (durch Angabe des `for`-Attributs). Fehlt eine passende Fehlerausgabekomponente auf einer Seite, erfolgt zwar auch keine Weiterleitung auf eine Folgeseite, der Benutzer bekommt aber keinerlei Hinweise, was der Grund dafür sein könnte. Daher sollte immer darauf geachtet werden, an Stellen, an denen ein Benutzer Eingaben vornehmen kann, auch entsprechende Bereiche für die Fehlerausgabe vorzusehen.

Natürlich werden durch die `UIMessage`-Komponente nicht lediglich Fehler dargestellt, die durch Validatoren erzeugt werden, sondern sämtliche Fehlermeldungen, die im Kontext abgelegt sind. So ist es z.B. möglich, dass innerhalb einer Programmlogik ein Fehler auftritt. Daraufhin kann eine entsprechende Fehlermeldung im Kontext abgelegt werden, die dann auf Seiten des Benutzers zur Anzeige kommt.

```
<h:form>
  <h:inputText converter="javax.faces.Integer"
     value="#{Customer.zip}" />
  <h:commandButton value="Submit" action="continue" />
  <br><br>
  <h:messages />
</h:form>
```

Listing 7.34: Verwendung des messages-Tag

In Listing 7.34 wird in einem Eingabefeld eine numerische Eingabe für die Postleitzahl erwartet. Wird anstelle eines numerischen Wertes ein willkürlicher Wert eingegeben, erscheint die Standardfehlermeldung an der Stelle, an der das `messages`-Tag hinterlegt ist.

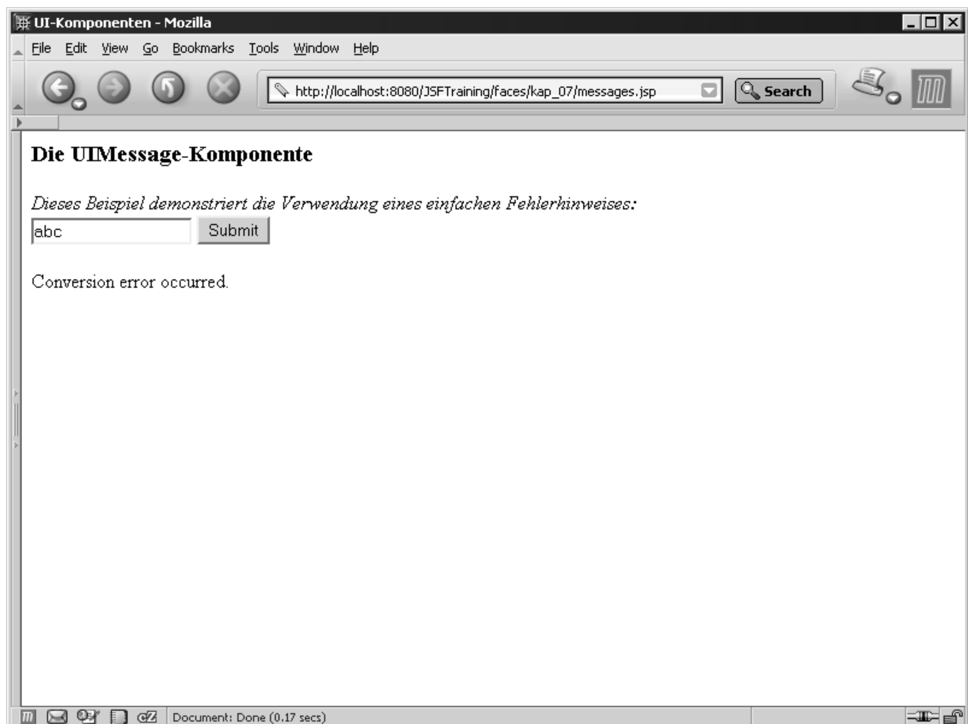


Abbildung 7.13: Darstellung von Fehlermeldungen

Die Standardfehlermeldungen können natürlich jederzeit überschrieben werden, um aussagekräftigere Texte darzustellen. Mehr dazu ist im Kapitel 6.6 zu finden.

Dem `message`-Tag können zur Darstellung verschiedene Attribute mitgegeben werden, die abhängig von der Fehlerklasse unterschiedliche Ausgabeformatierungen ermöglichen. In JSF werden die Fehlerkategorien

- `FacesMessage.SEVERITY_INFO`
- `FacesMessage.SEVERITY_WARN`
- `FacesMessage.SEVERITY_ERROR`
- `FacesMessage.SEVERITY_FATAL`

unterschieden. Dazu passend existieren im `message`-Tag die Attribute

- `infoStyle`
- `warnStyle`
- `errorStyle`
- `fatalStyle`

Somit kann abhängig von der Fehlerkategorie die Ausgabe mit Hilfe von Stylesheet-Angaben weiter beeinflusst werden.

7.14 UIGraphic-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UIGraphic`
HTML-Komponente	`javax.faces.component.html.HTMLGraphicImage`
Renderer	`javax.faces.Image`
Tag	`<h:graphicImage>`

Erläuterung

Bei der `UIGraphic`-Komponente gibt es wenig Überraschendes. Sie dient zur Anzeige von Bildern. Dabei werden die üblichen Formate *gif*, *jpg*, und *png* natürlich unterstützt. Wobei gerade letztere Aussage einzuschränken ist. JSF führt selbstverständlich keine Überprüfung durch, ob die Grafik tatsächlich angezeigt werden kann. Vielmehr wird die Angabe des Bildes direkt in HTML umgesetzt und die Darstellung dem Browser überlassen.

```
<h:graphicImage url="/java_logo.gif" />
```

Listing 7.35: Verwendung des graphicImage-Tags

Als Attribut kann eine Url mitangegeben werden, von wo aus das Bild, das angezeigt werden soll, geladen wird. Bei der `UIGraphic`-Komponente existiert ebenfalls eine HTML-spezifische Komponente: `HtmlGraphicImage`, die als Renderer standardmäßig den Typ `image` beinhaltet.

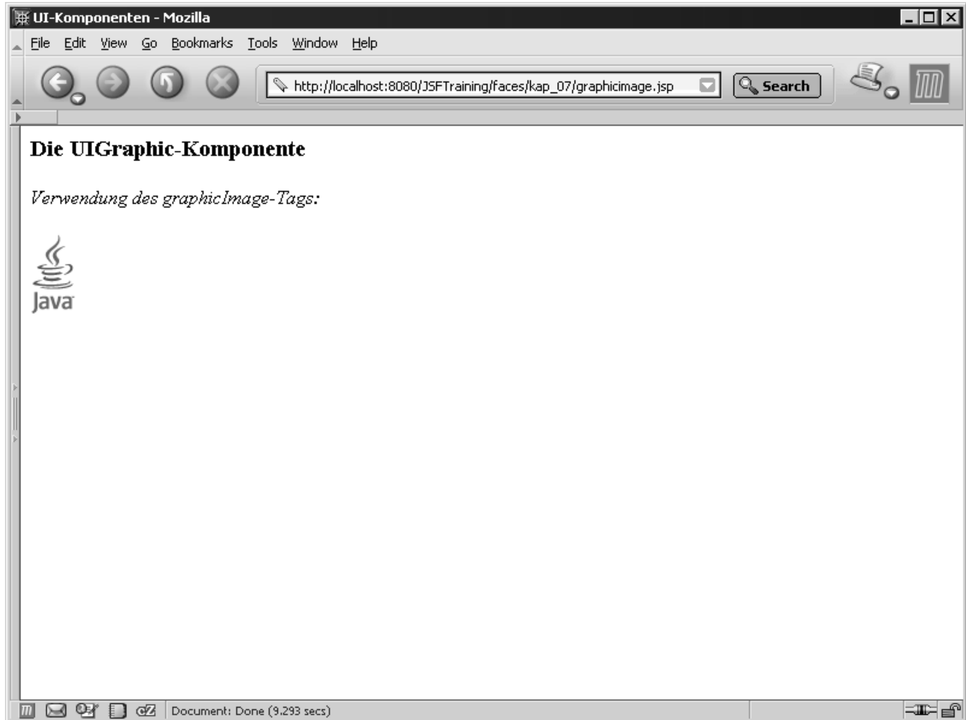


Abbildung 7.14: Tag graphicImage in Aktion

In Webanwendungen ist es üblich, zusätzlich zum eigentlichen Bild einen so genannten Alt-Text (*alternate text*) anzugeben. Dieser Text wird angezeigt, wenn das Bild aus welchem Grund auch immer nicht angezeigt werden kann. Dies kann z.B. dann der Fall sein, wenn ein Browser grundsätzlich keine Bilder darstellen kann (solche Browser gibt es tätsächlich!), oder aber aus Gründen der Ladezeit die Anzeige von Bildern deaktiviert wurde. Der Vorteil bei Verwendung der JSF-Tags gegenüber reinem HTML liegt auch darin, bei den Alt-Texten z.B. auf eine Ressourcendatei zugreifen zu können.

```
<h:graphicImage url="/img/country.jpg" alt="#{bundle.country}" />
```

Hierbei wird der Alt-Text aus einem Ressourcenbundle angezogen. Daher können die Alt-Texte je nach gesetzter Locale variieren, was bei einer festen Codierung mittels HTML nicht möglich wäre.

7.15 UIPanel-Komponente

Übersicht

Komponentenklasse	javax.faces.component.UIPanel
HTML-Komponente	javax.faces.component.html.HtmlPanelGrid
	javax.faces.component.html.HtmlPanelGroup
Renderer	javax.faces.Grid
	javax.faces.Group
Tag	<h:panelGrid>
	<h:panelGroup>

Erläuterung

Die `UIPanel`-Komponente ist sicherlich eine der etwas komplexeren Komponenten in JSF. Mit Hilfe der `UIPanel`-Komponente ist es möglich, Komponenten in strukturierter und gegliederter Form darzustellen. Wichtig bei der `UIPanel`-Komponente ist, dass die Anzahl der Zeilen bereits im Vorfeld feststehen muss und sich nicht aus dem Programmablauf ergeben kann. So ist es z.B. mit der `UIPanel`-Komponente nicht möglich, eine dynamische Anzahl an Produkten eines Warenkorbes auszugeben. Für solche Zwecke steht die `UIData`-Komponente zur Verfügung.

Eine `UIPanel`-Komponente wird durch die Renderer standardmäßig als HTML-Tabelle dargestellt. Es existieren zwei Unterklassen, `HtmlPanelGrid` und `HtmPanelGroup`, die mit den jeweils dazugehörigen Tags eine unterschiedliche Ausgabe bewirken. Darauf wird im Folgenden näher eingegangen.

Tag	**Beschreibung**
`panelGrid`	zur Darstellung einer Tabelle mit vorgegebener Zeilen- und Spaltenzahl
`panelGroup`	gruppiert eine bestimmte Anzahl von Komponenten

Tabelle 7.2: Tags bei Verwendung der UIPanel-Komponente

7.15.1 HtmlPanelGrid-Komponente

Mit Hilfe der `HtmlPanelGrid`-Komponente kann ein so genanntes *Grid*, also ein Gitternetz, logisch definiert werden. Damit können UI-Elemente wie Textfelder, Ein- und Ausgabekomponenten oder auch Bilder in tabellarischer Form dargestellt werden. HTML kennt für die Darstellung solcher Informationen das `table`-Element, das wiederum einzelne Zeilen und Spalten enthalten kann. Analog zum *GridLayout* bei Java/AWT wird durch Angabe der Spalten ein Gitternetz erzeugt, das durch Hinzufügen

einzelnen Komponenten zuerst waagerecht (also von links nach rechts) und danach senkrecht aufgefüllt wird. Ein direktes Setzen von Komponenten auf eine spezielle Koordinate ist nicht möglich. Zellen werden im Normalfall durch einzelne Komponenten festgelegt, dies können Ein- oder Ausgabekomponenten sein. Dargestellt wird die `HtmlPanelGrid`-Komponente mit Hilfe des `panelGrid`-Tags. Dabei besteht analog zum `colspan`-Attribut in HTML auch beim Tag `panelGrid` die Möglichkeit, einzelne Komponenten in einer Ausgabezelle zusammenzufassen.

Wichtig ist, dass durch ein `panelGrid`-Tag lediglich eine Struktur festgelegt wird, die durch enthaltene Komponenten befüllt wird. Die Komponenten sind fix, Wiederholelemente sind hierbei nicht möglich.

```
<h:panelGrid border="1" columns="2">

  <h:outputText value="Loginname" />
  <h:inputText value="#{Demologin.loginname}" size="15" />

  <h:outputText value="Passwort" />
  <h:inputText value="#{Demologin.password}" size="15" />

  <h:commandButton value="Anmelden" action="submit" />

  <h:messages />
</h:panelGrid>
```

Listing 7.36: Verwendung des panelGrid-Tags

In Listing 7.36 ist ein einfaches Beispiel für die Verwendung des `panelGrid`-Tags gegeben. Es wird ein Grid definiert, das aus zwei Spalten besteht. Die Anzahl der Zeilen ergibt sich anhand der Komponenten automatisch. Damit die Struktur des Grids deutlich wird, wird die HTML-Tabelle mit einer Rahmenbreite von 1 gerendert. Charakteristisch bei Anwendung des `panelGrid`-Tags ist es, dass einzelne Komponenten in der angegebenen Reihenfolge in das Grid eingestellt werden. Einen Zeilenumbruch oder ein exaktes Positionieren in eine Zelle ist nicht möglich.

In Abbildung 7.15 ist die Ausgabe von Listing 7.36 zu sehen. Prinzipiell ist die Darstellung zwar korrekt, doch existieren in der momentanen Fassung einige Einschränkungen. Dies wird deutlich, wenn an erster Stelle ein Ausgabetext definiert wird, der den Titel *Anmeldung* trägt.

```
<h:panelGrid border="1" columns="2">

  <h:outputText value="ANMELDUNG" />

  <h:outputText value="Loginname" />
  <h:inputText value="#{Demologin.loginname}" size="15" />
```

```
    <h:outputText value="Passwort" />
    <h:inputText value="#{Demologin.password}" size="15" />

    <h:commandButton action="submit" value="Anmelden"  />
    <h:messages />

  </h:panelGrid>
```

Listing 7.37: Erweitertes Grid

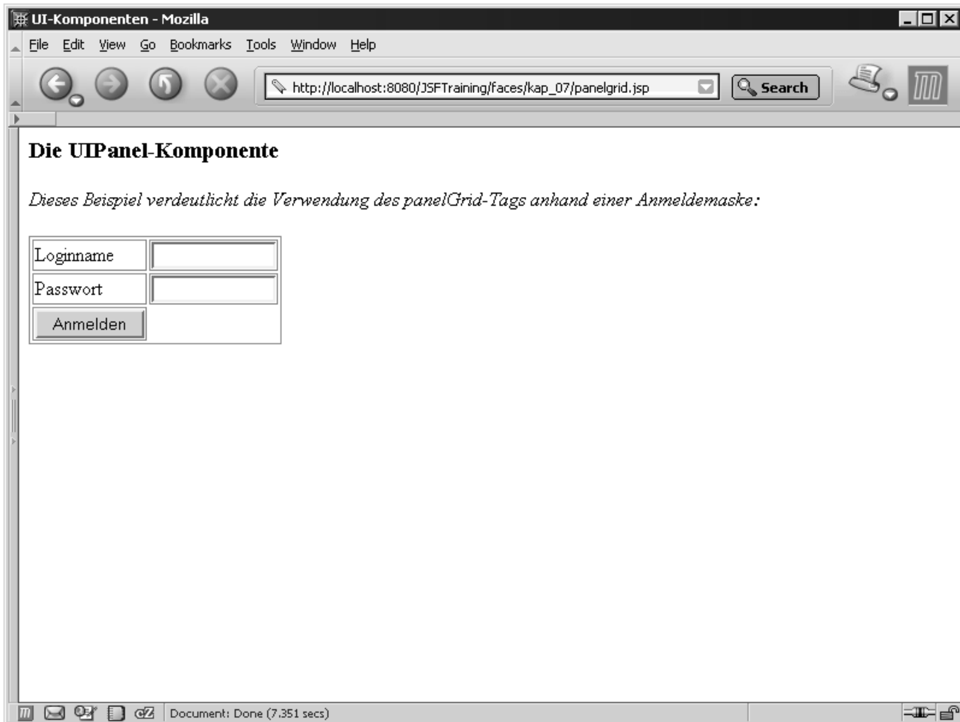


Abbildung 7.15: Anwendung des panelGrid-Tags

Für den Entwickler ist es natürlich klar, dass die Ausgabe des Texts *Anmeldung* oberhalb der sonstigen Angaben zu platzieren ist. Da der Renderer jedoch stupide Komponente für Komponente in eine Zelle setzt, erscheint als Ergebnis die in Abbildung 7.16 gezeigte Darstellung.

Die in Abbildung 7.16 gezeigte Darstellung ist natürlich nicht akzeptabel. Es muss somit dem Renderer mitgeteilt werden, dass sowohl die Ausgabezeile mit dem Text *Anmeldung* sowie auch der Anmeldebutton als Kopf- bzw. Fußzeile jeweils in eine eigene Zeile gehören. In JSF wird hierbei das `facet`-Tag verwendet, das grundlegend

einmal definiert, dass der darin enthaltene Wert unabhängig von dem umgebenden Element der Tabelle ist. Sowohl Kopf- als auch Fußzeile haben einen statischen Charakter und aktualisieren somit auch keine Modellobjekte, ein Update der Kopf- und Fußzeile bei sich ändernden Inhalten der eigentlichen Tabelle ist somit nicht notwendig. Dies wird durch ein `facet`-Tag festgelegt.

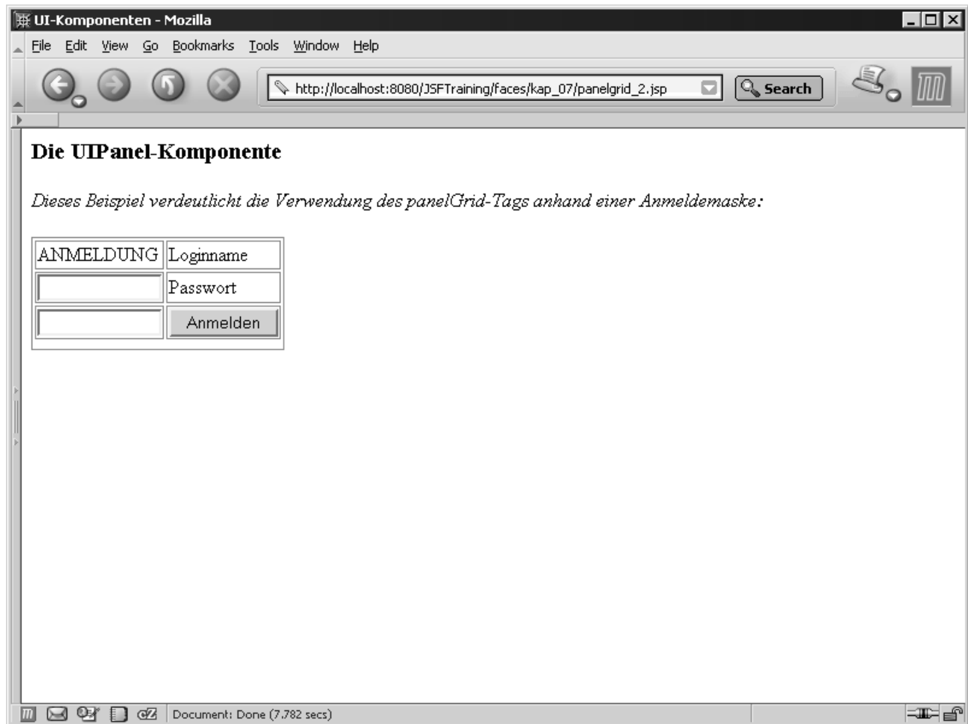


Abbildung 7.16: Fehlerhafte Darstellung

```
<h:panelGrid border="1" columns="2">

  <f:facet name="header">
    <h:outputText value="ANMELDUNG" />
  </f:facet>

  <h:outputText value="Loginname" />
  <h:inputText value="#{Demologin.loginname}" size="15" />

  <h:outputText value="Passwort" />
  <h:inputText value="#{Demologin.password}" size="15" />
```

```
    <f:facet name="footer">
      <h:commandButton value="Anmelden" action="login" />
    </f:facet>

  </h:panelGrid>
```

Listing 7.38: Verwendung des facet-Tags

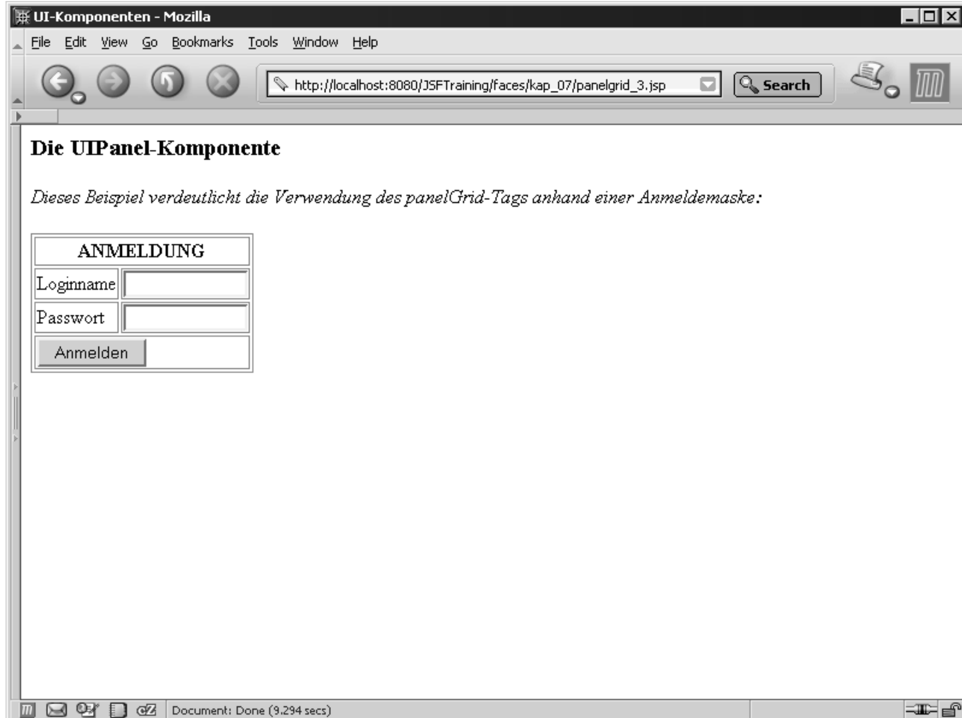


Abbildung 7.17: Facet-Tag zur Darstellung von Kopf- und Fußzeilen

Mit Hilfe des `facet`-Tags können Kopf- und Fußzeilen definiert werden, die unabhängig von der Spaltenzahl eine Zeile überspannen. Es kann jedoch auch die umgekehrte Forderung vorliegen. Mehrere Komponenten sollen innerhalb einer Zelle zusammengehörig dargestellt werden. Im Beispiel aus Abbildung 7.17 wäre es denkbar, dass mehrere Buttons in der Fußzeile angeordnet werden sollen. Oder bei einer Benutzeranmeldemaske soll ein Eingabefeld für die Postleitzahl in der gleichen Zelle stehen wie für die Eingabe des Ortes. Beispiele sind hier viele vorstellbar. Genaueres folgt in den nächsten Abschnitten.

Verwendung von Stylesheets

Ein Großteil des Erscheinungsbildes von UI-Komponenten wird durch Verwendung von Stylesheets beeinflusst. So sind nicht nur die Renderer-Klassen für eine Darstellung von Komponenten verantwortlich, sondern auch die entsprechenden Angaben in den Stylesheets, die über HTML-Befehle in eine Seite eingebunden werden. Das panelGrid-Tag unterstützt dabei eine Vielzahl von Möglichkeiten, das Aussehen der Tabelle zu beeinflussen.

Attribut	Resultat in HTML
styleClass	<table class="tablestyle">
headerClass	<th class="headerstyle">
footerClass	<th class="footerstyle">
rowClasses	<tr class="rowstyle">
columnClasses	<td class="columnstyle">

Tabelle 7.3: Stylesheet-Angaben beim panelGrid-Tag

Mit Hilfe der in Tabelle 7.3 aufgeführten Styleangaben kann das Aussehen einer Tabelle an unternehmensspezifische Corporate-Identity-Angaben angepasst werden, ohne dass dafür ein spezieller Renderer entwickelt werden muss.

Abbildung 7.18 zeigt eine HTML-Tabelle, die mit Hilfe des panelGrid-Tags sowie zusätzlichen Stylesheet-Angaben erzeugt wurde.

```
<h:panelGrid columns="2"
   headerClass="tableheader" footerClass="tablefooter"
   styleClass="smarttable">
```

Listing 7.39: Stylesheet-Angaben im panelGrid-Tag

In Listing 7.39 sind die Stylesheet-Angaben zu sehen, deren Ergebnis in Abbildung 7.18 gezeigt ist. Im Gegensatz zu HTML, wo die Stylesheet-Angaben in verschiedenen Tags getroffen werden, werden im panelGrid-Tag alle Angaben in diesem einen Tag eingetragen. Bei der Einbindung eines Stylesheets in einer JSF-Seite ist wie bei einer normalen HTML-Seite eine entsprechende Anweisung in den Header-Angaben vorzunehmen.

```
<HTML>
<head>
  <title>UI-Komponenten</title>
  <link rel="stylesheet" type="text/css" href="styles.css" />
</head>
```

Listing 7.40: Einbindung eines Stylesheets

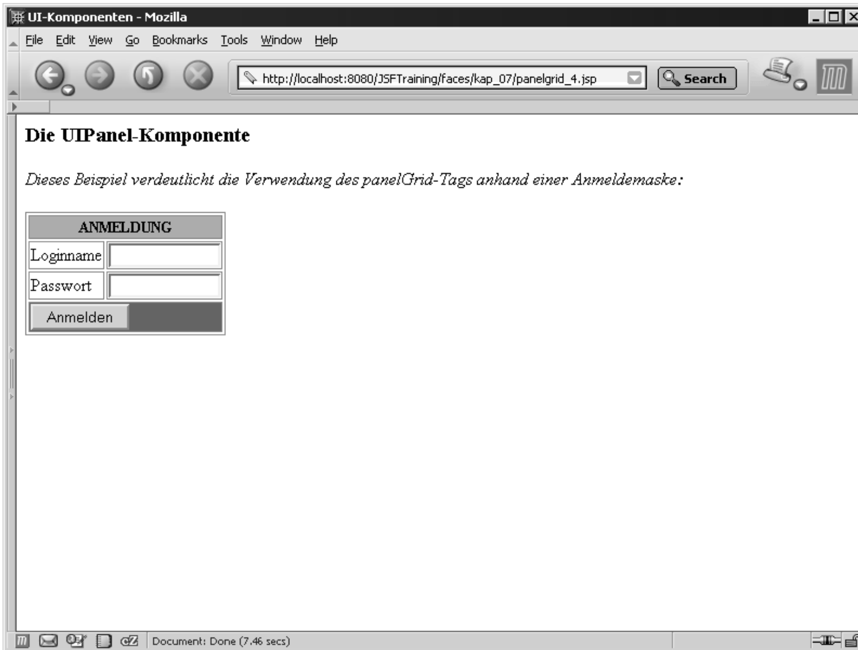


Abbildung 7.18: Einbindung eines Stylesheets

In Listing 7.40 wird das Stylesheet *styles.css* in die Seite mit eingebunden. Die Einbindung erfolgt relativ, d.h. es ist kein absoluter Pfad angegeben. Es wird somit die Datei im selben Verzeichnis wie die JSF-Seite gesucht. Dieses Vorgehen ist zwar für dieses Beispiel anwendbar, in der Praxis liegen jedoch Stylesheets im statischen Bereich direkt im Verzeichnis des Webservers, wohingegen JSF-Seiten im Bereich der dynamischen Seiten eines Applikationsservers abgelegt sind. Daher ist hierbei meist eine absolute Pfadangabe notwendig. In Listing 7.41 ist das in den Beispielen verwendete Stylesheet abgebildet.

```
.smarttable {
  border:medium solid black;
  border-spacing: 4px;
  background-color:#DCDCDC;
}
.tableheader {
  background-color:#DDDD00;
  font-size: 14px;
}
.tablefooter {
  background-color:#BD5203;
  font-size: 14px;
}
```

Listing 7.41: Stylesheet

7.15.2 HtmlPanelGroup-Komponente

Die `HtmlPanelGroup`-Komponente dient dazu, einzelne Komponenten zu gruppieren und zusammenzufassen. Gruppierte Komponenten werden daher wie eine einzige Komponente behandelt. So werden Komponenten, die über das dabei verwendete `panelGroup`-Tag zusammengefasst sind, in einer tabellarischen Darstellung in einer Zelle angeordnet. Daher wird die `HtmlPanelGroup`-Komponente meist im Zusammenhang mit der `HtmlPanelGrid`-Komponente verwendet.

Exemplarisch wird jetzt die Anmeldemaske aus den vorangegangenen Beispielen um einen Reset-Button erweitert, der die Eingabewerte der Anmeldemaske zurücksetzt. Aufgrund der Tatsache, dass das `facet`-Tag nur ein Kindelement besitzen darf, wird das `panelGroup`-Tag verwendet.

```
...
<f:facet name="footer">
  <h:panelGroup>
    <h:commandButton value="Anmelden" action="submit" />
    <h:commandButton value="Registrieren" action="reg" />
  </h:panelGroup>
</f:facet>
...
```

Listing 7.42: Verwendung des panelGroup-Tags

Das Listing 7.42 bezieht sich auf das in der Beschreibung über das HtmlPanelGrid-Komponente verwendete Beispiel einer Anmeldemaske. In der Fußzeile der tabellarischen Darstellung sollen zwei Buttons dargestellt werden. Aufgrund der Tatsache, dass das `facet`-Tag lediglich ein Kindelement akzeptiert, wird das `panelGroup`-Tag verwendet, das die darin enthaltenen Komponenten zu einer Komponente zusammenfasst.

Wie in Abbildung 7.19 abgebildet, erscheinen mit Verwendung des `panelGroup`-Tags beide Buttons wie gewünscht in der Fußzeile der Anwendung. Natürlich können über das `panelGroup`-Tag beliebig viele Elemente zusammengefasst werden und erscheinen nach außen wie eine einzige Komponente.

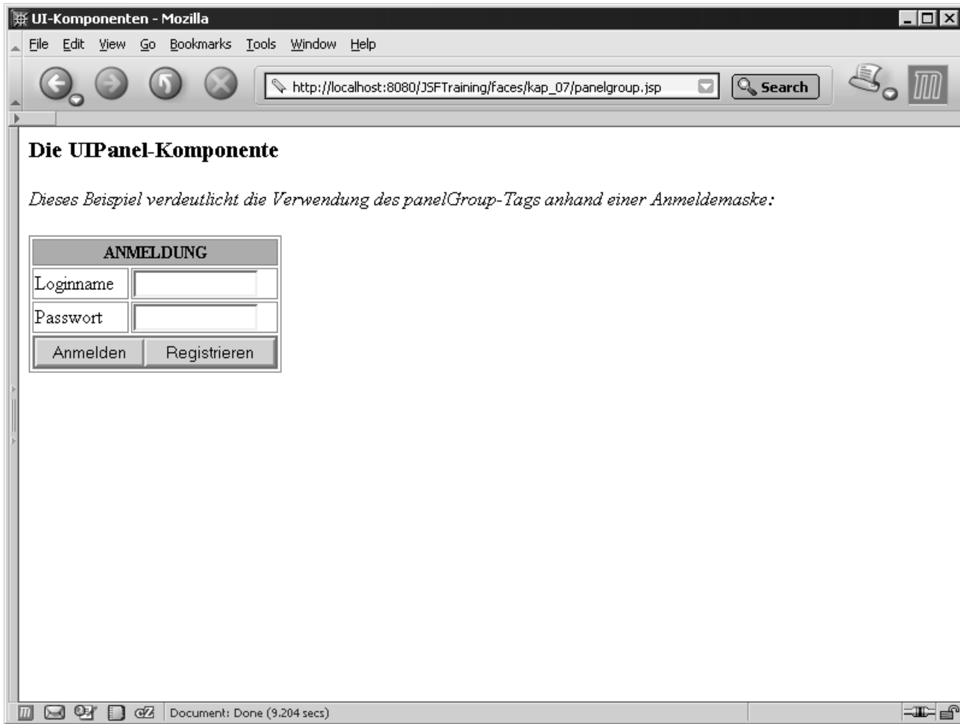


Abbildung 7.19: Verwendung des panelGroup-Tags

7.16 UIData-Komponente

Übersicht

Komponentenklasse	`javax.faces.component.UIData`
HTML-Komponente	`javax.faces.component.html.HtmlDataTable`
Renderer	`javax.faces.Table`
Tag	`<h:dataTable>`

Erläuterung

Die `UIData`-Komponente ist der `UIPanel`-Komponente einerseits sehr ähnlich, andererseits existieren doch entscheidende Unterschiede. Beiden gleich ist zum einen, dass diese Komponenten standardmäßig als Tabelle gerendert werden. Die `UIData`-Komponente hat als Unterklasse die `HtmlDataTable`-Komponente, die zur Darstellung in HTML verwendet wird. Zur Darstellung wird für die `HtmlDataTable`-Komponente das Tag `dataTable` verwendet.

Im Unterschied zum panelGrid-Tag werden jedoch beim dataTable-Tag keine Spaltenangaben explizit mit angegeben. Die Anzahl der Spalten ergibt sich vielmehr aus den einzelnen Ein- und Ausgabekomponenten. Die Anzahl der Zeilen wiederum ist auch nicht festgelegt, sondern wird dynamisch ermittelt. Dabei wird über eine Collection, ein Array, einen Iterator oder eine Map iteriert. Die Wiederholelemente werden somit zeilenweise angezeigt.

Der Hauptunterschied der Komponenten UIPanel und UIData liegt somit darin, dass die Ausgabemenge bei UIPanel bekannt sein muss, während sie bei UIData dynamisch variieren kann.

Tag	Funktion
dataTable	definiert eine dynamische Tabelle
column	definiert genau eine Spalte einer Tabelle

Tabelle 7.4: Tags bei der Verwendung der UIData-Komponente

Ein typischer Anwendungsfall für die UIData-Komponente ist die Ausgabe eines Warenkorbs in einer Shopanwendung. Der Kunde kann jederzeit Waren aus dem Shop in seinen Warenkorb hineinlegen oder auch Waren wieder herausnehmen. Dabei hat der Kunde zu jeder Zeit die Möglichkeit, sich in der Webanwendung den Inhalt seines Warenkorbs anzeigen zu lassen. Die Ausgabe des Warenkorbinhaltes könnte mit einer UIData-Komponente realisiert werden.

Im folgenden Beispiel soll eine Preisliste ausgegeben werden. Der Inhalt der Preisliste kann in einer reellen Anwendung aus einer Datenbank geladen werden. Für das Beispiel wird die Preisliste in einer Methode einfach befüllt. Die Preisliste soll dann mittels der UIData-Komponente im Browser ausgegeben werden.

```
package com.edu.jsf.bsp.bean;

import java.util.ArrayList;
import java.util.Date;

/**
 * beinhaltet eine Preisliste, die wiederum aus
 * mehreren Objekten der Klasse ProductBean besteht
 */
public class PricelistBean {

  private String name;
  private Date validfrom;
  private Date validuntil;
  private ArrayList productlist;

  /**
```

```
 * Konstruktor
 */
public PricelistBean() {
  loadDefaults();
}

/**
 * Getter-Methoden
 */
public String getName() {
  return name;
}

public Date getValidfrom() {
  return validfrom;
}

public Date getValiduntil() {
  return validuntil;
}

public ArrayList getProductlist() {
  return productlist;
}

/**
 * Setter-Methoden
 */
public void setName(String string) {
  name = string;
}

public void setProductlist(ArrayList list) {
  productlist = list;
}

public void setValidfrom(Date date) {
  validfrom = date;
}

public void setValiduntil(Date date) {
  validuntil = date;
}

/**
 * initialisiert die Produktliste mit einigen
 * Beispielwerten und initialisiert die Preisliste
 * selbst mit Demowerten
 */
private void loadDefaults() {
  name = "PC-Zubehörpreisliste";
```

```
      validfrom = new Date();
      validuntil =
        new Date( System.currentTimeMillis() + (1000*60*60*24*28;

      ProductBean product = null;
      productlist = new ArrayList();

      product = new ProductBean();
      product.setName("Tastatur");
      product.setPrice( 24.50 );
      productlist.add( product );

      product = new ProductBean();
      product.setName("PS/2 Maus");
      product.setPrice( 12.50 );
      productlist.add( product );

      product = new ProductBean();
      product.setName("Mausepad");
      product.setPrice( 2.50 );
      productlist.add( product );
    }
  }
```

Listing 7.43: Ein Preislisten-Bean

```
package com.edu.jsf.bsp.bean;

/**
 * Klasse für ein einfaches Produkt-Bean
 */
public class ProductBean {

  private String name;
  private double price;

  /**
   * Getter- und Setter-Methoden
   */
  public String getName() {
    return name;
  }

  public double getPrice() {
    return price;
  }

  public void setName(String string) {
    name = string;
  }
```

```
    public void setPrice(double d) {
      price = d;
    }
  }
```

Listing 7.44: Das Produkt-Bean

In Listing 7.43 und Listing 7.44 ist zum einen ein Preislisten-Bean zu sehen, das als eine Eigenschaft eine Anzahl von Produkten in einer *ArrayList* bereithält. Da zurzeit der Entwicklung einer JSF-Seite die Anzahl der in der Preisliste enthaltenen Elemente unbestimmt ist, eignet sich hierfür die Verwendung des `dataTable`-Tags ideal. Es wird über eine unbestimmte Anzahl von Elementen eines Modellobjektes iteriert, wobei für jedes Element eine neue Zeile in einer Tabelle verwendet wird.

```
<h:dataTable var="list" value="#{Pricelist.productlist}">

  <f:facet name="header">
    <h:outputText value="#{Pricelist.name}" />
  </f:facet>

  <h:column>
    <h:outputText value="#{list.name}" /><br>
  </h:column>
  <h:column>
    <h:outputText value="#{list.price}" />
  </h:column>

  <f:facet name="footer">
    <h:outputText value="#{Pricelist.validuntil}">
      <f:convertDateTime dateStyle="short" />
    </h:outputText>
  </f:facet>

</h:dataTable>
```

Listing 7.45: Verwendung des dataTable-Tags

In Listing 7.45 ist die Verwendung des `dataTable`-Tags demonstriert. Durch das `dataTable`-Tag wird festgelegt, dass über die Eigenschaft `Pricelist.productlist` iteriert werden soll. Dies funktioniert, da `productlist` vom Datentyp `java.util.Array` ist. Jedes Objekt, über das iteriert wird, kann innerhalb der Iteration über die Variable `list` angesprochen werden. Über die Angabe von `<h:outputText value="#{list.price}" />` wird somit der Preis des aktuellen Objekts der Iteration ausgegeben.

In Abbildung 7.20 ist das Ergebnis des `dataTable`-Tags zu sehen. Je nach Inhalt der ArrayList werden dynamisch weitere Zeilen angezeigt. Wichtig bei Verwendung des Tags ist es zudem, die einzelnen Spalten durch Angabe des `column`-Tags zu kennzeichnen.

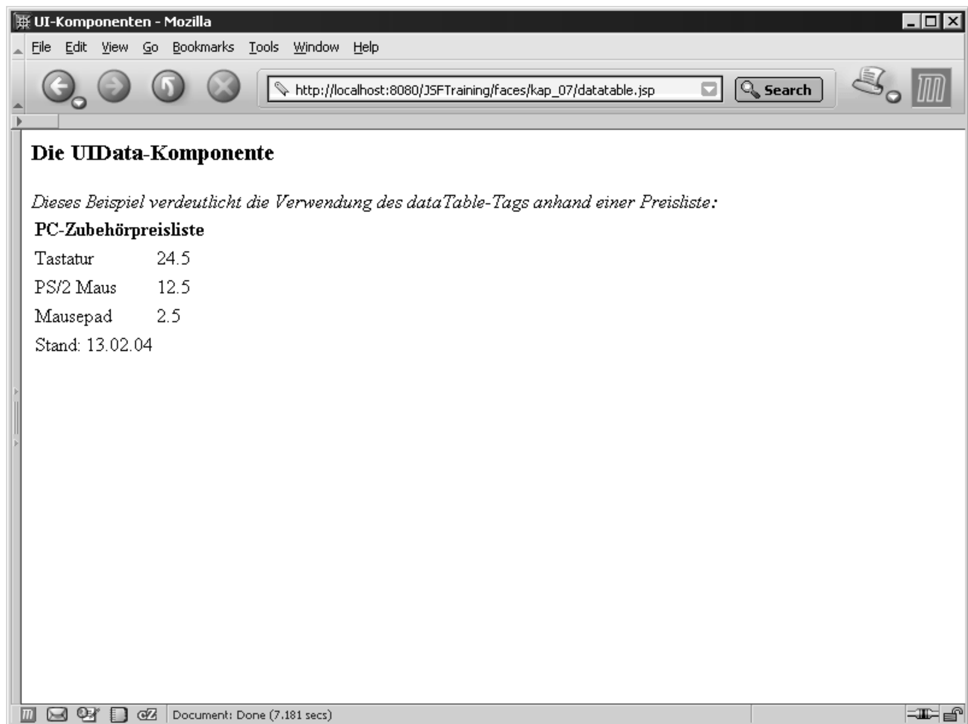


Abbildung 7.20: Das Tag dataTable in Aktion

Damit über eine unbestimmte Anzahl an Elementen iteriert werden kann, muss die Eigenschaft einem der folgenden Datentypen entsprechen:

- `List`
- `Array`
- `javax.faces.model.DataModel`
- `java.sql.ResultSet`
- `javax.servlet.jsp.jstl.sql.ResultSet`
- `javax.sql.RowSet`
- oder auch einfach ein einzelnes Bean

Zur Anpassung des Aussehens der gerenderten Tabelle existieren wie bei fast allen Ausgabetags auch für das `dataTable`-Tag entsprechende Stylesheet-Angaben. Diese entsprechen weitestgehend den Angaben des `panelGrid`-Tags.

Eingrenzung der Ausgabemenge

Eine sehr hilfreiche Funktion besteht darin, die durch das `dataTable`-Tag auszugebenden Datenzeilen einzugrenzen. Angenommen, es soll eine Preisliste mit mehreren Hundert Datensätzen angezeigt werden. Diese in ihrer Gesamtheit anzuzeigen wäre nicht ratsam, besser ist es, eine Funktion zu implementieren, mit der schrittweise eine Seite (also beispielsweise 20 Datensätze) vor- und zurückgeblättert werden kann.

Um dies zu realisieren, kann die Ausgabemenge des `dataTable`-Tags so beeinflusst werden, dass das erste und das letzte anzuzeigende Element angegeben werden kann. Es kann somit hinterlegt werden, dass nur die Datensätze 10-20 oder 25-35 angezeigt werden.

```
<h:form>
  <h:dataTable first="#{Pricelist.counter}"
     rows="8" var="list" value="#{Pricelist.productlist}">

    <f:facet name="header">
      <h:outputText value="#{Pricelist.name}" />
    </f:facet>

    <h:column>
      <h:outputText value="#{list.name}" /><br>
    </h:column>
    <h:column>
      <h:outputText value="#{list.price}" />
    </h:column>

    <f:facet name="footer">
      <h:panelGroup>
        <h:commandButton value="Zurück" action="backward"
           actionListener="#{Pricelist.backward}" />
        <h:commandButton value="Vor" action="forward"
           actionListener="#{Pricelist.forward}" />
      </h:panelGroup>
    </f:facet>

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

Listing 7.46: Navigation in einer Liste

Das Listing 7.46 wurde so erweitert, dass im Attribut `first` auf eine Modellreferenz verwiesen wird, in der der aktuelle Zählerstand gespeichert ist, ab der die Datensätze angezeigt werden sollen. Durch das Attribut `rows` wird gesteuert, dass maximal acht Datensätze auf einmal angezeigt werden. Damit der Zähler bei Betätigen der Vor- und Zurückbuttons gesetzt wird, wird im `commandButton`-Tag eine Aktion aufgerufen, in der der jeweilige Zählerwert erhöht bzw. erniedrigt wird.

Wichtig ist, dass beide Buttons im Fußbereich durch ein `panelGroup`-Element umklammert sind, da das `facet`-Tag lediglich ein Kindelement verarbeitet. Durch das `panelGroup`-Element werden die beiden Buttons wie ein Element behandelt.

Da durch Einbau der Buttons aus der einfachen Ausgabeliste jetzt ein Formular wurde, das eben durch die beiden Buttons abgeschickt werden kann, muss zudem ein `form`-Tag mit eingebaut werden.

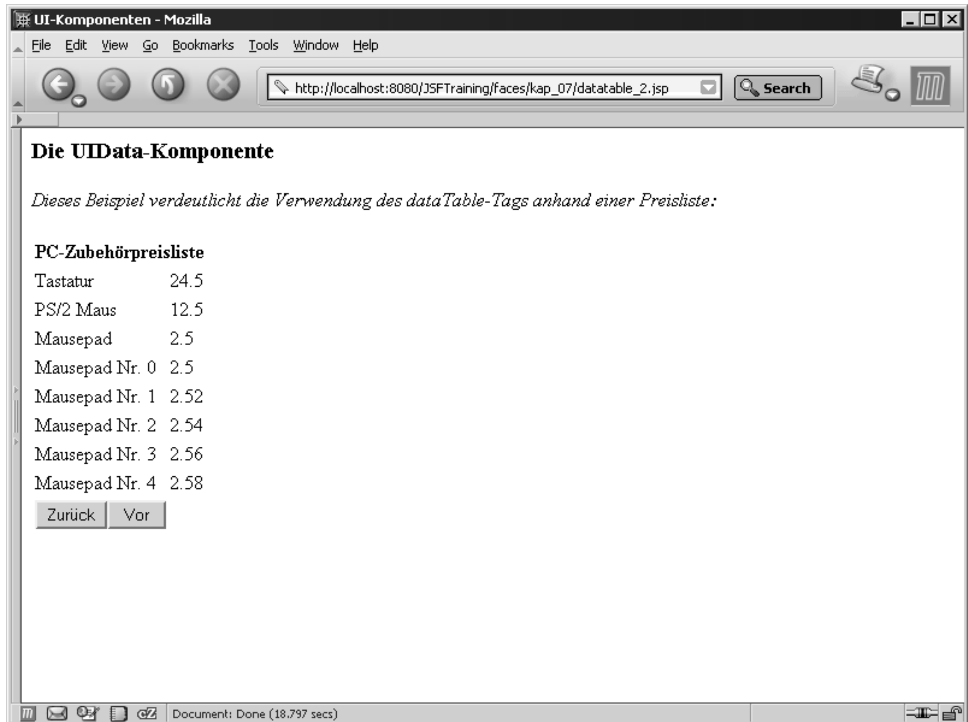


Abbildung 7.21: Vorwärts- und Rückwärtsnavigation

Abbildung 7.21 zeigt die Ausgabe der Liste mittels der Einschränkung der Ausgabeelemente. Durch die Buttons *Zurück* und *Vor* kann die Preisliste entsprechend durchsucht werden. An dieser Stelle würden in einer reellen Anwendung noch weitere Prüfungen dazukommen, z.B. dass nicht über das Ende der gesamten Produkte hinaus geblättert werden kann. Für die hier abgebildete Beispielanwendung wurde die Klasse `PricelistBean` lediglich wie folgt ergänzt:

```
...
public void backward( ActionEvent event ) {
  counter -= 8;
  if ( counter < 0 ) counter = 0;
```

```
  }

  public void forward( ActionEvent event ) {
    counter += 8;
  }
  ...
```

Listing 7.47: Die Action-Methoden in der Klasse PricelistBean

Des Weiteren wurden in der Klasse eine Eigenschaft `counter` vom Typ `int` ergänzt sowie die dazugehörigen setter- und getter-Methoden.

7.17 Fazit

In diesem Kapitel wurde die Verwendung der Tagbibliotheken ausführlich demonstriert. Sie haben gelernt, die Standard-UI-Komponenten von JavaServer Faces einzusetzen und damit bereits erste kleinere Beispiele aufzubauen.

Wichtig ist die Unterscheidung zwischen einer Komponente und einem Renderer. Während Komponenten grundsätzlich unabhängig von der späteren Darstellung sind, regeln Renderer das letztendliche Erscheinungsbild der Komponente. Eine Sonderfunktion nehmen die HTML-Komponenten ein, die zunächst auch von der Darstellung losgelöst sind, jedoch bereits viele Eigenschaften von HTML-spezifischen Komponenten vorhalten.

Es wurde zudem gezeigt, dass jede Seite durch ein `view`-Tag umgeben wird, erweitert um eventuelle `subview`-Tags, falls weitere JSP-/JSF-Seiten über eine `include`-Anweisung eingebunden werden.

Mittels der Standardkomponenten und der Standardrenderer können im Normalfall sicherlich die meisten Anforderungen an eine moderne Webanwendung abgedeckt werden. An den Stellen, an denen Spezialfunktionen benötigt werden, können benutzerdefinierte Komponenten zum Einsatz kommen. Dies wird in Kapitel 9 erläutert.

8 Die Beispielapplikation JSF-WebLog

Kapitelziel

Dieses Kapitel zeigt exemplarisch die Entwicklung einer kompletten Webanwendung unter Verwendung von JavaServer Faces auf. Sämtliche Vorgehensweisen, Techniken und Einsatzmöglichkeiten, die in den vorhergehenden Kapiteln erläutert wurden, sind in diesem Kapitel in einem umfassenderen Beispiel eingebettet.

Des Weiteren werden Fragestellungen besprochen, wie sie mit Sicherheit auch in der Praxis auftauchen können. So wird u.a. vorgestellt, wie eine Initialisierung von Beans vorgenommen werden kann, bevor eine Aktion in JavaServer Faces überhaupt stattgefunden hat.

Des Weiteren wird eine Möglichkeit präsentiert, wie eine Webanwendung sehr einfach an eine Datenbank angebunden werden kann und das Zusammenspiel zwischen Datenbank und JavaServer Faces technologisch sehr gut passen kann.

8.1 Einführung in die Beispielapplikation

Eine neue Technologie lernt man am schnellsten, indem man deren konkreten Einsatz an einem Beispiel betrachten kann. Aus diesem Grund wird in den folgenden Abschnitten exemplarisch eine komplette Anwendung auf Basis von JavaServer Faces konzipiert und entwickelt. Bei der Webanwendung handelt es sich um eine Blogging-Engine. Blogging-Anwendungen sind eine zurzeit sehr verbreitete Form von Anwendung, bei der Benutzer ihre Tagebücher (»WebLogs«) online führen und Dritten zum Lesen zugänglich machen.

Um an einer Blogging-Community teilzunehmen, genügt es im Normalfall, sich mit einigen persönlichen Angaben in einer Community zu registrieren. Danach erhält man ein Passwort und kann künftig online sein Tagebuch führen und dort viel Persönliches, Amüsantes oder Nachdenkliches der breiten Öffentlichkeit zur Verfügung stellen. Oftmals bieten einige Blogs die Möglichkeit, einen Steckbrief des Benutzers – teilweise auch mit Foto – zu betrachten. So bekommen die Besucher und Leser ein noch besseres Bild des Verfassers eines Online-Tagebuchs. Teilweise findet man auch Blogs, in denen

Besucher die Einträge eines Verfassers zusätzlich kommentieren können. Es ensteht damit eine eigene Community, in der offen über Meinungen und Erfahrungen kommuniziert und häufig auch diskutiert wird.

Blogs können auch als eine Art Mischung aus persönlicher Nachrichtenseite und Gästebuch verstanden werden. Einerseits setzt der Autor in regelmäßigen Abständen bestimmte Informationen ins Web, andererseits können sich Leser aktiv daran beteiligen, indem einzelne Einträge auch kommentiert werden können. So müssen Blogs nicht ausschließlich reine Tagebuchaufgaben wahrnehmen, in denen lediglich der Tagesablauf chronologisch aufgeschrieben wird. In Gegenteil, es können auch bestimmte Erlebnisse und Meinungen dargelegt und diskutiert werden. Blogs sind somit eine weitere Möglichkeit der Kommunikation und des (Gedanken-)Austausches im Internet.

Die im Folgenden beschriebene Beispielapplikation *JSF-WebLog* hat genau das Ziel, eine solche Plattform für WebLogs aufzubauen. Neue Benutzer sollen die Möglichkeit haben, sich auf der Plattform zu registrieren. Bereits angemeldete Benutzer können einen Steckbrief über sich selbst verwalten sowie natürlich ihr Online-Tagebuch führen. Eine Kommentierung der Einträge wird nicht behandelt, aber natürlich steht es Ihnen frei, aufbauend auf diesem Beispiel diese Erweiterung in Eigenregie zu übernehmen. Zusätzlich bekommt jeder Benutzer zu seinem WebLog ein Gästebuch, in das andere Benutzer Kommentare und Nachrichten schreiben können.

Diese Beispielapplikation soll natürlich nicht den Anspruch erheben, eine bereits bis ins letzte ausgearbeitete Blogging-Engine zu sein. Es ist zwar möglich, diese Anwendung so auch online zu schalten, doch soll die Anwendung sie eher dazu ermuntern, eigene Erweiterungen und Funktionalitäten miteinzubauen. Denn erst im konkreten Umgang mit einer Anwendung erlernt sich eine neue Technologie am effektivsten. Nutzen Sie also die Gelegenheit, um basierend auf dem Rahmenwerk eigene Entwicklungen mit JSF zu versuchen und somit auch in die Feinheiten des neuen Frameworks vorzudringen!

Die folgenden Abschnitte beschreiben Schritt für Schritt, wie eine Anwendung von der Idee bis zur Online-Schaltung mit Hilfe von JavaServer Faces konzipiert und vor allen Dingen auch realisiert wird. Die Beispielanwendung soll Ihnen daher nicht nur zeigen, wie Sie mittels JSF eine Anwendung realisieren können, sondern auch, mit welchen Verfahren und Techniken moderne Webanwendungen entwickelt werden können. Auch wird Ihnen im Laufe des Kapitels ein Open-Source-Produkt (Apache Torque) vorgestellt, das sehr gut für ein Objekt-Relationales-Mapping geeignet ist. Damit kann eine Datenbank sehr komfortabel an eine Webanwendung angebunden werden. Der Vorteil dieses Werkzeugs liegt darin, dass Sie sich um SQL-Statements, Connection-Pooling usw. keine Gedanken machen müssen, dies wird alles durch Torque bereits einsatzfähig zur Verfügung gestellt. Sollten Sie dieses Tool bisher noch nicht gekannt haben, bietet Ihnen die kurze Einführung in die Verwendung des Tools sicherlich auch einige interessante Ansätze.

8.2 Vorgehensweise

Bei der Entwicklung der Beispielanwendung wird systematisch vorgegangen, indem zunächst mittels Use-Cases die Anforderungen genau aufgenommen werden. Aufbauend auf diesen Anwendungsfällen wird mit Hilfe so genannter Screenflows das Verhalten und die Navigation in der Anwendung festgelegt.

Erst danach geht es an die Datenmodellierung, bei der die benötigten Entitäten herausgearbeitet und die Grundlagen für eine Datenbankanbindung festgelegt werden. Die eigentliche Arbeit mit JSF beginnt im Anschluss an die Datenmodellierung, indem zunächst die Entwicklungsumgebung eingerichtet und danach mit dem Thema Managed-Beans begonnen wird. Nach erfolgreichem Aufbau der JSF-Seiten und der Navigation erfolgt in einem weiteren Abschnitt der Ausblick auf einen weiteren Ausbau der Anwendung.

8.3 Entwickeln der Use-Cases

Uses-Cases, die sich auf Deutsch am ehesten mit der Bezeichnung *Anwendungsfall* übersetzen lassen, sind Teil der *Unified Modeling Language (UML)*. Die UML ist ein Standard für die Notation für die Objektorientierte Modellierung. Die UML ist eine Notation, die ein objektorientiertes System auf verschiedenste Weisen beschreibt. Wichtig ist jedoch der Hinweis, dass die Uml lediglich ein System beschreibt, nicht, wie das Vorgehensmodell zur Realisierung des Systems auszusehen hat. Dazu gibt es andere Verfahren, wie beispielsweise den *Rational Unified Process (RUP)* oder auch *eXtreme Programming (XP)*, ein Vertreter eines leichtgewichtigen (agilen) Entwicklungsprozesses.

Für die Beschreibung eines objektorientierten Systems existieren in der Uml eine Reihe so genannter *Sichten*, die durch unterschiedliche Diagramme abgebildet werden. Das *Use-Case-Diagramm* ist ein bekannter Vertreter hieraus. Mit Hilfe von Use-Cases werden diejenigen Anwendungsfälle entwickelt, die für die Anwendung später benötigt werden. Use-Cases werden meist in Zusammenarbeit von Entwicklern und Kunden erarbeitet, oft auch in einem speziellen Workshop. Da die Notation der Use-Cases sehr anschaulich und wenig techniklastig ist, können auch in der UML-Notation unerfahrene Anwender sehr schnell damit umzugehen lernen. Use-Cases sind somit ein ideales Werkzeug, um eine erste Vorstellung über die Anforderungen an eine Anwendung zu erhalten und diese zugleich zu dokumentieren. Zusätzlich zu den graphisch in Kurzform dargestellten Use-Cases kann eine textuelle Beschreibung der einzelnen Anwendungsfälle erstellt werden, in denen jeder einzelne Use-Case nochmals näher beschrieben werden kann.

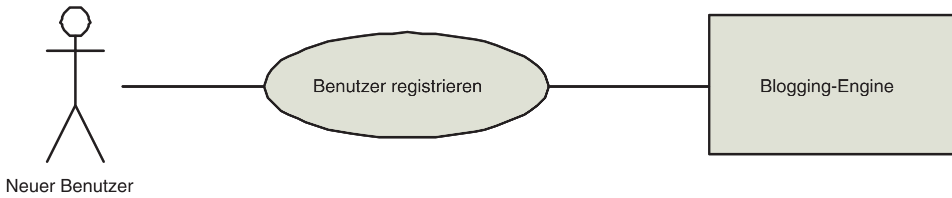


Abbildung 8.1: Use-Case »Anmelden am System«

Use-Case-Name	**Anmelden am System**
Vorbedingung	Der Benutzer ist nicht im System angemeldet.
Nachbedingung Erfolg	Der Benutzer ist erfolgreich mit Name und Passwort auf der Plattform angemeldet.
Nachbedingung Misserfolg	Der Benutzer ist nicht auf der Plattform angemeldet und erhält einen Hinweis, dass die Anmeldung fehlgeschlagen ist.
Akteure	Benutzer
Beschreibung	Der Benutzer meldet sich am System durch Eingabe von Name und Kennwort an und hat danach die Möglichkeit, die volle Funktionalität der Anwendung zu nutzen. Bei falscher Anmeldung wird ein entsprechender Hinweis gegeben.

Tabelle 8.1: Beschreibung des Use-Cases »Anmelden am System«

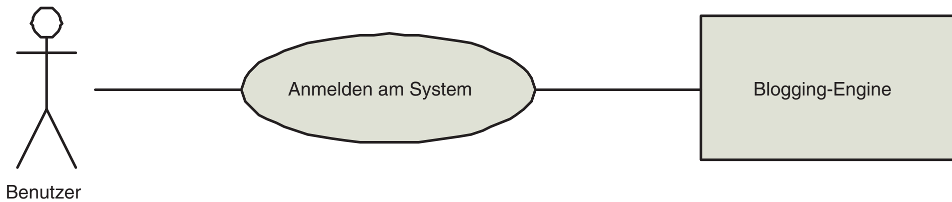


Abbildung 8.2: Use-Case »Benutzer registrieren«

Use-Case-Name	**Benutzer registrieren**
Vorbedingung	-
Nachbedingung Erfolg	Der Benutzer hat sich mit seinen persönlichen Daten auf der Plattform registriert. Ein entsprechender Benutzereintrag wurde vom System durchgeführt.
Nachbedingung Misserfolg	Der Benutzer ist nicht registriert und erhält einen Hinweis, welche Felder nicht vollständig ausgefüllt sind bzw. welcher Fehler vorliegt.
Akteure	Benutzer

Tabelle 8.2: Beschreibung des Use-Cases »Benutzer registrieren«

Use-Case-Name	**Benutzer registrieren**
Beschreibung	Um als neuer Benutzer auf der Plattform agieren zu können, ist eine Registrierung notwendig. Dabei werden u.a. der Loginname sowie eine Mailadresse und ein Passwort hinterlegt. Die Daten werden in der Datenbank gespeichert, so dass sich künftig der Benutzer über die Plattform anmelden kann.

Tabelle 8.2: Beschreibung des Use-Cases »Benutzer registrieren« (Fortsetzung)

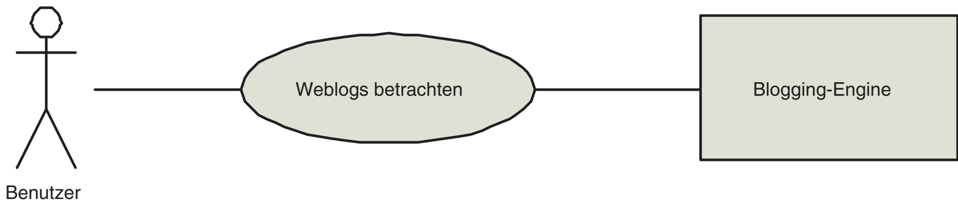


Abbildung 8.3: Use-Case »WebLogs betrachten«

Use-Case-Name	**WebLogs betrachten**
Vorbedingung	-
Nachbedingung Erfolg	Der Benutzer kann Einträge anderer Benutzer lesen sowie deren Steckbrief einsehen. Des Weiteren kann er das Gästebuch einsehen und auch eigene Einträge hinzufügen.
Nachbedingung Misserfolg	-
Akteure	Benutzer
Beschreibung	Sowohl als angemeldeter wie auch als nicht angemeldeter Benutzer hat man die Möglichkeit, Einträge sowie den Steckbrief und das Gästebuch von anderen WebLog-Benutzern zu betrachten.

Tabelle 8.3: Beschreibung des Use-Cases »WebLogs betrachten«

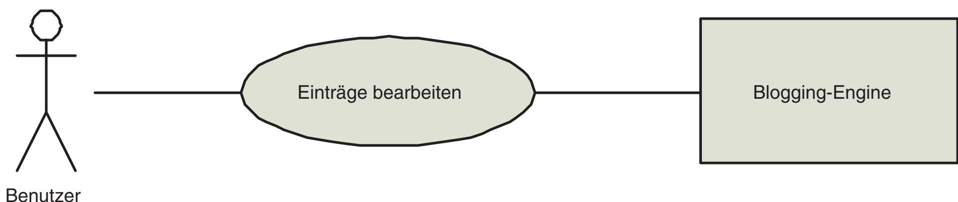


Abbildung 8.4: Use-Case »Einträge bearbeiten«

Use-Case-Name	Einträge bearbeiten
Vorbedingung	Der Benutzer hat sich erfolgreich angemeldet.
Nachbedingung Erfolg	Der Benutzer hat seinen Steckbrief geändert oder Eintragungen im WebLog vorgenommen.
Nachbedingung Misserfolg	Die angestrebten Änderungen werden nicht durchgeführt und der Benutzer erhält einen Fehlerhinweis.
Akteure	Benutzer
Beschreibung	Nach erfolgreicher Anmeldung auf der Plattform kann ein Benutzer seinen eigenen Steckbrief bearbeiten sowie weitere Eintragungen im WebLog vornehmen.

Tabelle 8.4: Beschreibung des Use-Cases »Einträge bearbeiten«

Für die Anwendung JSF-WebLog wurden somit insgesamt vier Anwendungsfälle identifiziert. Sicherlich wären weitere Anwendungsfälle denkbar bzw. könnten einzelne Anwendungsfälle auch in mehrere kleinere Fälle unterteilt werden. Wichtig ist jedoch, dass mit Hilfe der Use-Cases das Gesamtsystem beschrieben wird und alle Anforderungen seitens des Kunden darin widergespiegelt werden.

Bei größeren Anwendungen können an dieser Stelle weitere Uml-Diagramme erstellt werden. Meist folgt im Anschluss an die Use-Case-Erstellung die Erarbeitung eines Klassenschemas, da durch die Use-Cases auf Seiten der Fachanwendung die benötigten Klassen ersichtlich sind. Im Falle der Beispielanwendung wird jedoch an dieser Stelle nicht weiter in die Uml eingestiegen, sondern aufbauend auf den (fachlichen) Anforderungen bereits ein erster Navigationsentwurf erarbeitet.

Für die Erstellung der Use-Case-Diagramme wurde das Open-Source-Produkt *ArgoUml* verwendet. Mit ArgoUml existiert in der Open-Source-Gemeinde seit einigen Jahren ein sehr beliebtes und leistungsstarkes Uml-Modellierungs-Werkzeug. Zurzeit in der Version 0.12 (stable version) weist es bereits ein umfangreiches Spektrum an Diagrammtypen und Funktionen für eine Unterstützung eines toolbasierten Entwicklungsprozesses auf. Im Vergleich zu kommerziellen Systemen wie Rational Rose oder Together überzeugt es vor allem durch seine einfache Bedienung sowie auch durch die Tatsache, dass es als Open-Source-Produkt lizenzkostenfrei erhältlich ist. Es kann über die Webseite *http://www.sourceforge.net* heruntergeladen werden.

Die Screenshots im Buch wurden mithilfe von Visio umgesetzt.

8.4 Screenflow

Nachdem mit Hilfe der Use-Cases die einzelnen Anwendungsfälle festgelegt und dokumentiert wurden, entsteht sowohl auf Entwickler- wie auch auf Kundenseite mit Sicherheit bereits ein konkreteres Bild der Anwendung. Während sich ein Entwickler jedoch meist von Berufs wegen auf einem sehr abstrakten Level bewegen kann, kann sich ein Kunde oftmals noch keine bildliche Vorstellung von der zu schaffenden Anwendung machen. Um hier frühzeitig eine gemeinsame Vorstellung zu ermöglichen, werden oftmals so genannte *Screenflow-Diagramme* eingesetzt. Screenflow-Diagramme sind *kein* offizieller Bestandteil der Uml, eignen sich jedoch hervorragend, um zusammen mit dem Kunden einen ersten Eindruck der zu entwickelnden Anwendung zu bekommen.

Screenflow-Diagramme deuten ein ungefähres Aussehen der einzelnen Bildschirmseiten (Screens) und deren Ablauf an. Sie werden meist mit Hilfe von Grafikprogrammen oder auch in MS Word oder MS Visio erstellt. Sie sollen keine graphischen Richtlinien für Designer enthalten, sondern lediglich den ungefähren Informationsgehalt sowie den Programmablauf der Anwendung aufzeigen. Screenflow-Diagramme sind ein sehr geeignetes Instrument, um in Zusammenarbeit mit dem Kunden eine bessere Vorstellung der zu erstellenden Anwendung zu erarbeiten.

Im Prinzip sind Screenflow-Diagramme vergleichbar mit Prototypen, die schnell entwickelt werden, um dem Kunden ein ungefähres Gefühl für die spätere Anwendung zu geben. Bei Screenflow-Diagrammen wird jedoch noch keine Zeile Quellcode geschrieben, sondern lediglich die Anwendung bzw. der Anwendungsablauf mit Hilfe von Masken (auf Papier) dargestellt.

In Abbildung 8.5 ist der Ablauf der späteren JSF-Seiten dargestellt. Von einer zentralen Indexseite hat der Benutzer die Möglichkeiten, sich entweder zunächst auf der Plattform zu registrieren oder als bereits registrierter Benutzer seine eigenen Einträge zu bearbeiten. Des Weiteren kann sowohl ein nicht angemeldeter als auch ein angemeldeter Benutzer Einträge anderer Benutzer ansehen.

Im Anschluss an die Erarbeitung der Use-Cases und den ersten Entwurf der Screenflows werden bereits die Webdesigner aktiv, die aufbauend auf diesen Informationen erste Designvorschläge für die künftige Anwendung entwerfen. Da der objektorientierte Entwicklungsprozess ein iterativer Prozess ist (oder besser sein sollte), kann es durchaus vorkommen, dass noch weitere Use-Cases hinzukommen sowie Änderungen am Screenflow vorgenommen werden. Die Abbildungen 8.6, 8.7 und 8.8 zeigen den statischen Entwurf der JSF-WebLog-Anwendung. Alle Seiten sind noch ohne jegliche Funktionalität als statische HTML-Seiten abgespeichert. Diese Seiten sind die graphische Vorlage, mit der ein (JSF-)Anwendungsentwickler seine Arbeit beginnen kann.

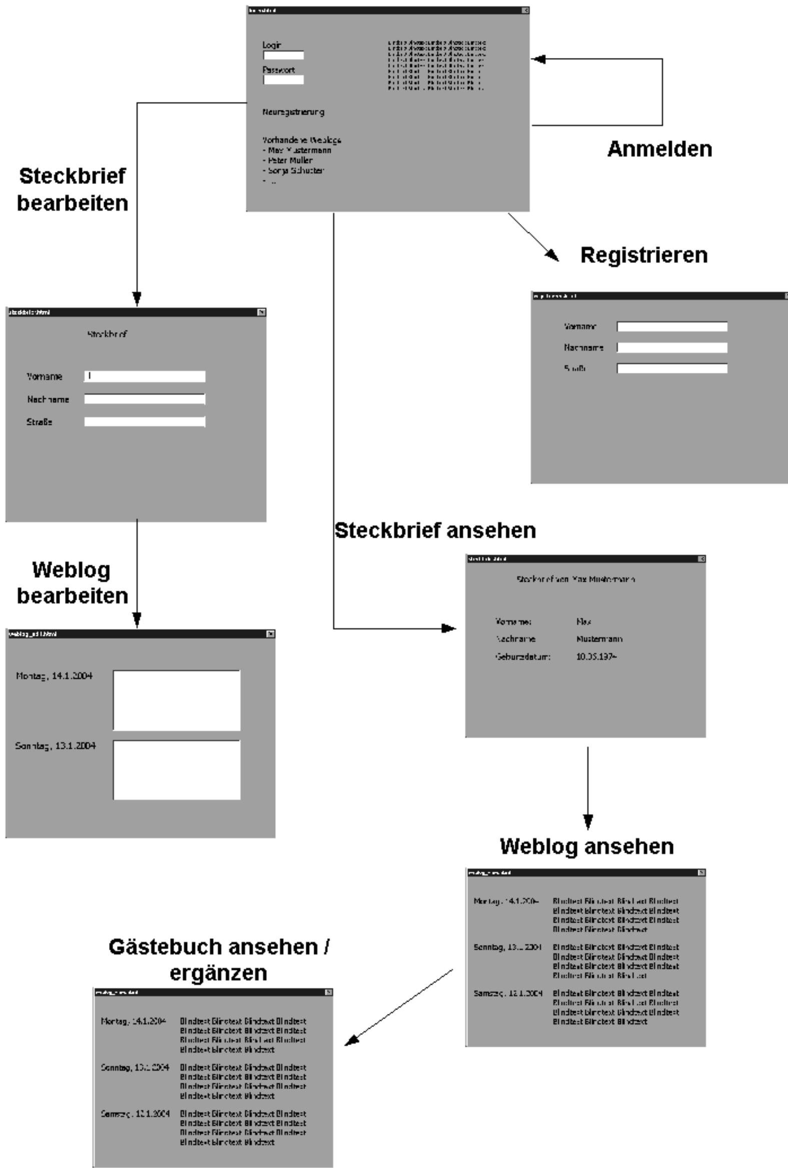


Abbildung 8.5: Screenflow-Diagramm

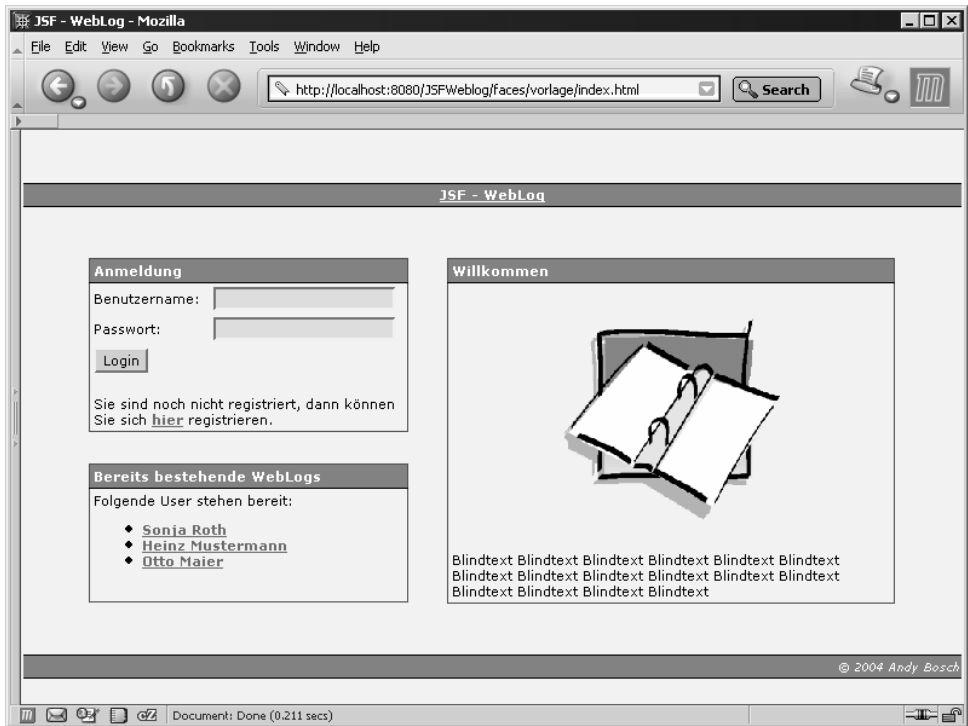


Abbildung 8.6: Vorlage für die Startseite der Anwendung

Die zentrale Startseite ist so aufgebaut, dass ein Benutzer sich direkt anmelden kann oder sich als neuer Besucher zunächst registrieren kann. Ebenfalls ist eine Liste von vorhandenen Blogs zu sehen, die jederzeit von jedem betrachtet werden können.

Die Registrierung neuer Benutzer erfolgt auf einer separaten Seite. Interessant ist die Vorlage des Designers, wie Fehler visualisiert werden sollen. Anstatt lediglich einen Fehlertext auszugeben, soll zusätzlich das Eingabefeld farblich hinterlegt werden. Dies ist sicherlich eine interessante Anforderung für die Komponentenentwickler.

Das WebLog selbst ist in der aktuellen Fassung noch recht unspektakulär. Ein Benutzer kann zwischen der Blog-Sicht und der Steckbrief-Sicht umschalten, auch ein Zurücknavigieren zur Startseite soll jederzeit möglich sein. Im Bereich der Gästebucheintragungen soll er vorhandene Eintragungen ansehen sowie einen eigenen Eintrag vornehmen können.

Mit diesen mittlerweile schon sehr konkreten Abbildungen der künftigen Anwendung können mit dem Kunden nochmals eventuelle Änderungen besprochen und der Ablauf bei Bedarf nochmals modifiziert werden. Es sei nochmals erwähnt, dass bis zum jetzigen Zeitpunkt noch keine Zeile Quellcode geschrieben wurde, sondern die künftige Anwendung aus konzeptioneller Sicht zunächst beschrieben wurde.

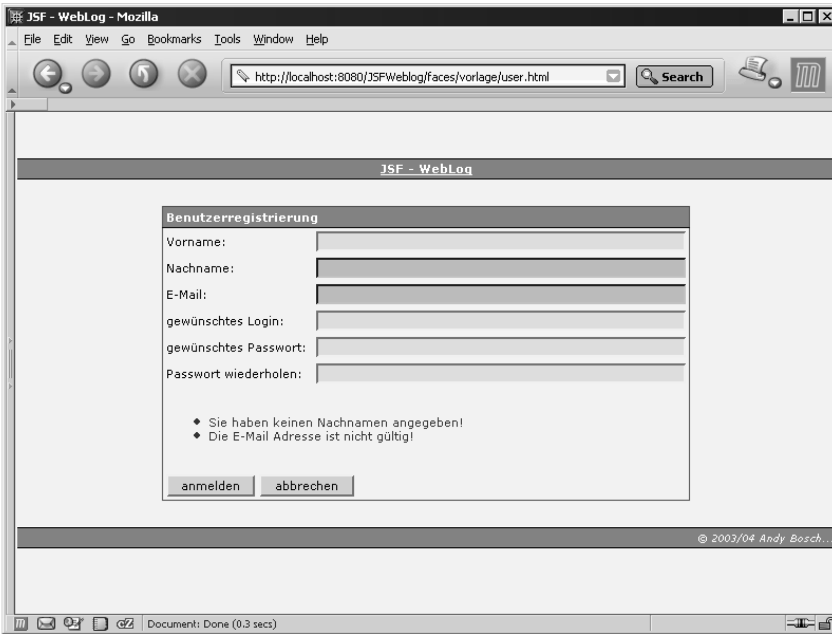


Abbildung 8.7: Benutzerregistrierung

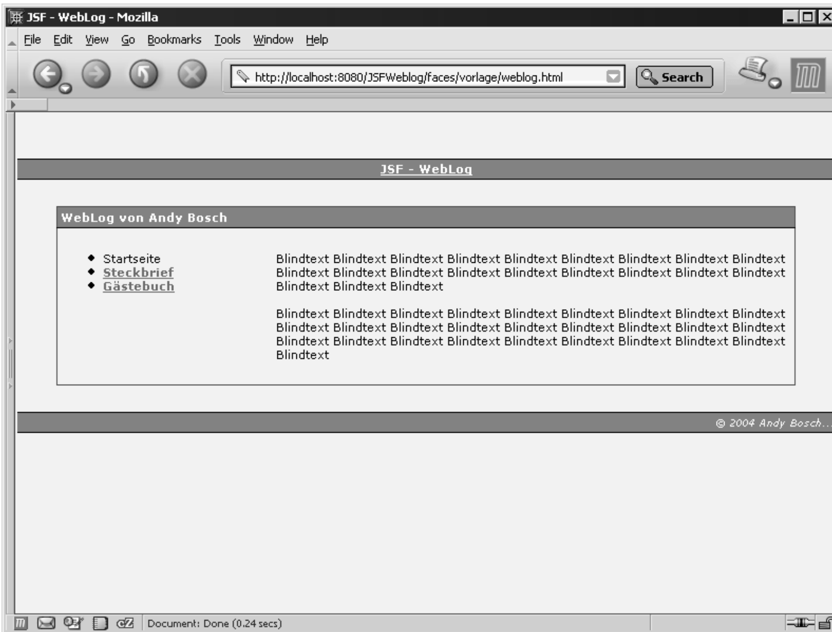


Abbildung 8.8: Darstellung des WebLogs

8.5 Datenmodellierung

Die zu entwickelnde Anwendung weist eine Reihe von Daten auf, die persistent gespeichert werden müssen. So existieren einerseits Benutzerdaten, mit denen sich ein Benutzer überhaupt auf der Plattform anmelden kann. Dazu werden die Daten des Steckbriefs sowie die einzelnen WebLog-Einträge separat für jeden Benutzer mitabgespeichert. Auch die einzelnen Gästebucheinträge sollen in einer Datenbank abgelegt werden können. Für die Speicherung der Daten lassen sich somit folgende unterschiedliche Objekte identifizieren:

- Benutzer: Ein Benutzer kann sich auf der Plattform anmelden und weist Attribute wie Name, Passwort und eventuell eine E-Mail-Adresse auf.
- WebLog: Ein WebLog ist das Tagebuch selbst, das zum einen den Introtext für Besucher speichert sowie die Verbindung zum eigentlichen Besitzer.
- WebLog-Einträge: Ein WebLog hat wiederum beliebig viele Einträge. Ein Eintrag weist den Text für einen Tag auf.
- Steckbrief: Zwischen einem Benutzer und dem Steckbrief existiert eine 1:1-Beziehung, da jeder Benutzer genau einen Steckbrief mit entsprechenden Optionen hinterlegen kann. Daher wäre es auch möglich, sowohl die Benutzerdaten als auch die Daten des Steckbriefs später in einer Tabelle zu speichern. Aus Gründen der Übersichtlichkeit werden jedoch zwei Objekte definiert, die später auch in getrennten Tabellen gespeichert werden.
- Gästebucheinträge: Die eingegebenen Kommentare von Anwendern im Bereich des Gästebuchs sind ebenfalls in der Datenbank zu speichern.

Basierend auf diesen identifizierten Objekten kann ein Entity-Relationship-Diagramm (ERD) entworfen werden, das die zu erstellenden Tabellenstrukturen samt ihren Beziehungen untereinander beinhaltet:

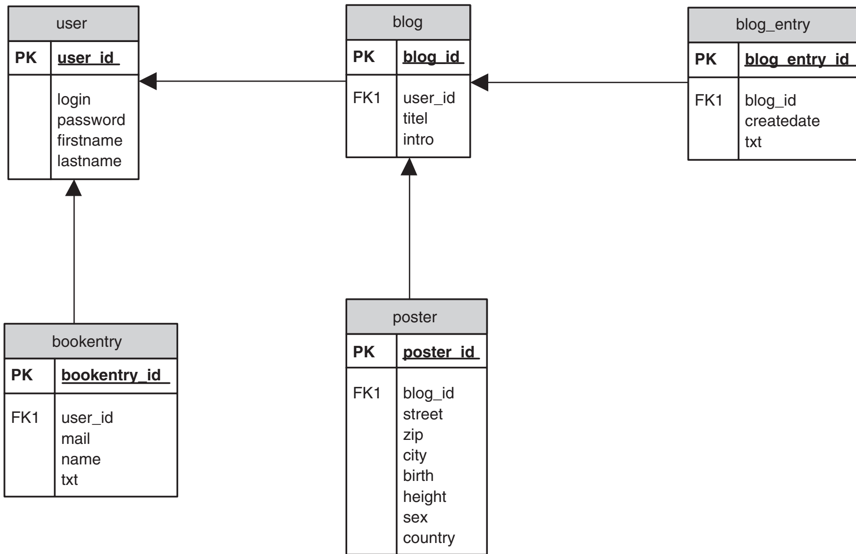


Abbildung 8.9: Entity-Relationship-Diagramm

8.6 Datenbankanbindung mit Torque

Ziel dieses Buches ist es sicherlich, Sie in die Arbeit mit JSF einzuführen und Ihnen die Anwendung dieser neuen Technologie an vielen Beispielen anschaulich zu erläutern. Sie sollen aber auch für Ihre Arbeit in JSF-Projekten Anregungen über weitere Werkzeuge erhalten, die – falls Sie diese bisher noch nicht eingesetzt haben – Ihnen einiges an Arbeit abnehmen können.

Daher wird für die Datenbankanbindung in dieser Beispielanwendung das O/R-Werkzeug *Torque* verwendet. Torque ist ein O/R-Mapping Tool, das vom Jakarta-Team der Apache-Gruppe als Open-Source-Produkt entwickelt und gepflegt wird. Es ist jedoch nicht neu, sondern wurde bereits im Rahmen des *Turbine-Frameworks* entwickelt und eingesetzt. Mit der Version 2.2 des Turbine-Frameworks wurde Torque ausgegliedert und steht seitdem unter »eigener Flagge«. Mit Hilfe von Torque kann eine Datenbank bzw. ihre Inhalte objektorientiert über Java-Objekte verwaltet werden. Eine unschöne Mischung von Java und SQL ist nicht mehr notwendig. Der große Vorteil von Torque liegt jedoch hauptsächlich darin, dass lediglich die verwendeten Entitäten beschrieben werden müssen, ansonsten ist so gut wie keine Programmierarbeit mehr notwendig, um eine Datenbank über Java-Objekte ansprechen zu können.

Eine Alternative, beispielsweise die Datenbankanbindung selbst via JDBC zu realisieren, wäre ein ungemein höherer Aufwand. Zudem bietet Torque den Vorteil, dass die Beschreibung der Daten in einem herstellerneutralen Format vorliegt und daher ein Wechsel auf eine andere Datenbank ohne Probleme jederzeit möglich ist.

Torque kann über die Website der Apache Group *http///db.apache.org* heruntergeladen werden. Zudem ist Torque in der Version 3.1 auf der beiliegenden CD enthalten.

8.6.1 Installation von Torque

Die Installation von Torque besteht aus zwei Teilen, der Installation des Generators sowie der Installation der Laufzeitumgebung. Die notwendigen Installationspakete sind getrennt über die Apache-Webseite zu beziehen bzw. ebenfalls auf der beigelegten CD enthalten. Die Pakete müssen zur Installation lediglich in ein separates Verzeichnis extrahiert werden, weitere Installationsschritte sind nicht notwendig. Natürlich sollte auf dem System bereits eine *MySQL-Datenbank* vorhanden und lauffähig sein. Des Weiteren ist es empfehlenswert, gleich zu Beginn die von Torque erforderlichen Bibliotheken aus dem *lib*-Verzeichnis in das projekteigene *WEB-INF/lib*-Verzeichnis zu kopieren. Damit sind die benötigten Bibliotheken im Klassenpfad der Anwendung vorhanden.

8.6.2 Anpassen der Konfigurationsdateien

Um sich mittels Torque sowohl das notwendige Datenbankschema als auch die notwendigen Java-Klassen generieren zu lassen, müssen insgesamt drei (eventuell auch vier) Dateien bearbeitet werden: eine Build-Datei, in der generelle Eigenschaften der Datenbank hinterlegt werden, eine Datei, die das Datenbankschema selbst beschreibt, sowie eine Datei für die Laufzeit-Parameter. Wird der *IDBroker-Service* verwendet, auf den später noch näher eingegangen wird, muss hierfür ebenfalls eine Datei angepasst werden.

Als erste Datei wird die Build-Datei direkt im Torque-Hauptverzeichnis angepasst. In ihr werden hauptsächlich die Verbindungseinstellungen zur Datenbank vorgehalten.

```
application.root = .

torque.project = WebLog
torque.targetPackage = com.edu.jsf.WebLog.dbbean

torque.database = MySQL
torque.database.createUrl = jdbc:MySQL://localhost/WebLog
torque.database.buildUrl = jdbc:MySQL://localhost/WebLog
torque.database.url = jdbc:MySQL://localhost/WebLog
torque.database.driver = org.gjt.mm.MySQL.Driver
torque.database.user =
torque.database.password =
torque.database.host = 127.0.0.1
```

Listing 8.1: Ausschnitt aus der Datei build.properties

In Listing 8.1 ist ein Ausschnitt der Datei *build.properties* zu sehen. Sämtliche Parameter sind in der Vorlage, die mit Torque mitausgeliefert wird, näher erläutert. Es wird hinterlegt, wie die Datenbank benannt werden soll, sowie der Name und das Passwort für den Zugriff auf die Datenbank. Da im Beispiel eine MySQL-Datenbank verwendet wird, ist der Typ *MySQL* ebenso wie der erforderliche Datenbanktreiber anzugeben. Der Datenbanktreiber ist meist in Form einer jar-Bibliothek vorhanden und muss ebenfalls in das *lib*-Verzeichnis von Torque sowie das *WEB-INF/lib*-Verzeichnis der Anwendung kopiert werden. Weiterhin wichtig ist die Angabe des `targetPackage`. Hiermit wird geregelt, mit welcher Package-Angabe die Klassen erzeugt werden sollen.

Neben der Konfigurationsdatei *build.properties* ist noch die Datei für die Laufzeiteigenschaften anzupassen. Diese Datei wird später von der Anwendung eingelesen und ausgewertet. Während die *build.properties*-Datei lediglich einmal für die Erzeugung des Datenbankschemas und der Java-Klassen benötigt wird, ist die Datei für die Laufzeiteigenschaften für das Ausführen der späteren Webanwendung zwingend notwendig.

```
torque.applicationRoot = .

log4j.category.org.apache.torque = ALL, org.apache.torque
log4j.appender.org.apache.torque = org.apache.log4j.FileAppender
log4j.appender.org.apache.torque.file =
   ${torque.applicationRoot}/logs/torque.log
log4j.appender.org.apache.torque.layout = org.apache.log4j.PatternLayout
  log4j.appender.org.apache.torque.append = false

torque.database.default=WebLog
torque.database.WebLog.adapter=MySQL

torque.dsfactory.WebLog.factory=
   org.apache.torque.dsfactory.SharedPoolDataSourceFactory
torque.dsfactory.WebLog.pool.defaultMaxActive=10
torque.dsfactory.WebLog.pool.testOnBorrow=true
torque.dsfactory.WebLog.pool.validationQuery=SELECT 1
torque.dsfactory.WebLog.connection.driver =
   org.gjt.mm.MySQL.Driver
torque.dsfactory.WebLog.connection.url =
   jdbc:MySQL://localhost/WebLog
torque.dsfactory.WebLog.connection.user =
torque.dsfactory.WebLog.connection.password =

torque.idbroker.cleverquantity=true
torque.manager.useCache = true
```

Listing 8.2: Ausschnitt aus der Datei WebLog.properties

Die Datei für die Laufzeiteigenschaften *WebLog.properties* ähnelt vom Aufbau her der Build-Datei. Zusätzlich werden jedoch u.a. Angaben zum Logging von Torque gemacht.

Am Anfang ist das Anpassen aller Konfigurationsdateien sicherlich ein wenig aufwändig, dennoch sind die Vorteile eines Einsatzes von Torque enorm. Bei einem Wechsel des Datenbanksystems z.B. sind nur wenige Eingriffe notwendig. Auch eine Erweiterung des Datenbankschemas stellt kein größeres Problem dar, da einzig die Schemadatei erweitert werden muss.

8.6.3 Erstellen des Datenbankschemas

Für die Definition der Entitäten wird die Datei *WebLog-schema.XML* verwendet. Darin ist in XML-Syntax die gesamte Datenstruktur beschrieben. Es empfiehlt sich, die Datei im *schema*-Unterverzeichnis gemäß dem Namensschema *xxx-schema.XML* in *WebLog-schema.XML* umzubenennen. Der Aufbau der XML-Datei selbst ist weitestgehend selbst erklärend. Es werden sowohl die Tabellennamen als auch die Namen und Datentypen der einzelnen Spalten hinterlegt. Welche Ausprägungen für die einzelnen Angaben möglich sind, schlägt man am besten in den entsprechenden Dokumentation zu Torque nach, die ebenfalls über die Webseite zu beziehen sind. Erwähnenswert ist noch die Angabe `defaultIdMethod` im `database`-Tag. Hiermit wird gesteuert, wie eine Vergabe von Ids erfolgt. Torque unterstützt hierbei verschiedene Varianten:

Methode	Beschreibung
native	Torque verwendet in diesem Fall den (proprietären) Mechanismus der zugrunde liegenden Datenbank, um eindeutige Primärschlüssel zu erzeugen.
idbroker	Torque stellt über den IDBroker-Service einen eigenen Service zur Erzeugung von Primärschlüsseln bereit. Dieser Service ist nicht an ein spezielles Datenbanksystem gebunden.
none	Bei Angabe von ‚none' werden keine Primärschlüssel erzeugt. Dies ist z.B. dann sinnvoll, wenn sich in einer n:m-Zuordnungstabelle der Primärschlüssel aus mehreren Fremdschlüsseln ergibt und somit kein eigener Primärschlüssel erzeugt werden muss.

Tabelle 8.5: Methoden für die Id-Vergabe

Da eine MySQL-Datenbank über die *Autoincrement*-Eigenschaft selbst eindeutige Ids generieren kann, wäre im konkreten Beispiel die Methode `native` möglich. Soll jedoch zu einem späteren Zeitpunkt die Datenbank ausgetauscht werden, kann dies eventuell zu Problemen führen. Es ist deshalb meist ratsam, den *IDBrokerService* von Torque zu verwenden. Dazu wird im Datenbanksystem automatisch von Torque eine weitere

Tabelle angelegt, die die gesammte Id-Verwaltung beinhaltet. So wird hierbei gespeichert, welche Id in welcher Tabelle zuletzt vergeben wurde. Auch die Schrittweite zwischen den einzelnen Ids kann darüber beeinflusst werden.

```
<?XML version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE database SYSTEM "http://jakarta.apache.org/turbine/dtd/database.dtd">

<database name="WebLog" defaultIdMethod="idbroker">

<table name="user" idMethod="idbroker">
  <column name="user_id" required="true"
   primaryKey="true" type="INTEGER"/>
  <column name="login" required="true" size="30" type="VARCHAR"/>
  <column name="password" required="true" size="50"
   type="VARCHAR"/>
  <column name="firstname" size="30" type="VARCHAR"/>
  <column name="lastname" size="30" type="VARCHAR"/>
</table>

<table name="blog" idMethod="idbroker">
  <column name="blog_id" required="true"
   primaryKey="true" type="INTEGER"/>
  <column name="titel" required="true" size="30" type="VARCHAR"/>
  <column name="intro" required="false" type="LONGVARCHAR"/>
  <column name="user_id" required="true" type="INTEGER"/>

  <foreign-key foreignTable="user">
    <reference local="user_id" foreign="user_id"/>
  </foreign-key>
</table>

<table name="blog_entry" idMethod="idbroker">
  <column name="blog_entry_id" required="true"
   primaryKey="true" type="INTEGER"/>
  <column name="createdate" required="true" size="30"
   type="BIGINT"/>
  <column name="txt" type="LONGVARCHAR"/>
  <column name="blog_id" required="true" type="INTEGER"/>

  <foreign-key foreignTable="blog">
    <reference local="blog_id" foreign="blog_id"/>
  </foreign-key>
</table>

<table name="poster" idMethod="idbroker">
  <column name="poster_id" required="true"
   primaryKey="true" type="INTEGER"/>
  <column name="street" size="30" type="VARCHAR"/>
  <column name="zip" size="6" type="INTEGER"/>
  <column name="city" size="30" type="VARCHAR"/>
```

```
    <column name="birth" type="DATE"/>
    <column name="height" type="DECIMAL"/>
    <column name="sex" size="1" type="VARCHAR"/>
    <column name="country" size="30" type="VARCHAR"/>
    <column name="blog_id" required="true" type="INTEGER"/>

    <foreign-key foreignTable="blog">
      <reference local="blog_id" foreign="blog_id"/>
    </foreign-key>
  </table>

  <table name="bookentry">
    <column name="bookentry_id" required="true" primaryKey="true"
      type="INTEGER"/>
    <column name="user_id" required="true" type="INTEGER"/>
    <column name="name" required="true" size="50" type="VARCHAR"/>
    <column name="mail" required="true" size="50" type="VARCHAR"/>
    <column name="txt" required="false" type="LONGVARCHAR"/>
    <column name="createdate" type="DATE"/>

    <foreign-key foreignTable="user">
      <reference local="user_id" foreign="user_id"/>
    </foreign-key>
  </table>
  </database>
```

Listing 8.3: Definition des Schemas in der Datei WebLog-schema.XML

Kommt der IDBroker-Service zum Einsatz, ist die Datei *id-table-schema.XML* im */schema*-Unterverzeichnis zusätzlich anzupassen. Es genügt hierbei jedoch, den Namen der Zieldatenbank mitanzugeben.

```
  <?XML version="1.0" encoding="ISO-8859-1" standalone="no"?>
  <!DOCTYPE database SYSTEM "http://db.apache.org/torque/dtd/database_3_1.dtd">
  <database name="WebLog">
    <table name="ID_TABLE" idMethod="idbroker">
      <column name="ID_TABLE_ID" required="true"
        primaryKey="true" type="INTEGER"/>
      <column name="TABLE_NAME" required="true"
        size="255" type="VARCHAR"/>
      <column name="NEXT_ID" type="INTEGER"/>
      <column name="QUANTITY" type="INTEGER"/>

      <unique>
        <unique-column name="TABLE_NAME"/>
      </unique>

    </table>
  </database>
```

Listing 8.4: Die Datei id-table-schema.XML

8.6.4 Erzeugen der Modell- und Peerklassen

Nachdem alle benötigten Dateien für die Anwendung angepasst wurden, kann damit begonnen werden, die Java-Klassen sowie das Datenbankschema selbst anzulegen. Dazu wird das Build-Tool *ant* verwendet, das ebenfalls über die Webseite von Apache (*http://ant.apache.org*) zu beziehen ist. Zudem ist *ant* ebenfalls in der Version 1.6.1 auf der CD enthalten.

Ant

Ant ist Java-basiertes Build-Werkzeug. Mit Ant ist es möglich, so genannte Build-Dateien in Form einer XML-Datei zu erstellen. Ant wird häufig beim Deployment von Anwendungen eingesetzt. In einer Build-Datei wird u.a. hinterlegt, welche Klassen kompiliert werden müssen und welche Dateien in welcher Form (z.B. als jar-Bibliothek gepackt) in welches Verzeichnis gestellt werden müssen. Mit Ant können somit Abläufe automatisiert werden. In der Unix-Welt existiert bereits seit vielen Jahren ein Verfahren mittels make-Dateien. Ant Build-Dateien haben jedoch den Vorteil, dass die XML-Syntax sehr einfach zu verstehen ist und plattformübergreifend eingesetzt werden kann.

Im ersten Schritt werden die Java-Klassen generiert, die in die eigene Anwendung eingebunden werden können.

```
ant -f build-torque.XML
```

Nach einigen Sekunden sollte die Meldung `BUILD SUCCESSFUL` erscheinen. Danach können die Dateien, die im Unterzeichnis *src/java* gespeichert werden, in den Workspace der eigenen Anwendung kopiert werden. Die erzeugten Klassen bieten die Möglichkeit, über Methodenaufrufe auf Datensätze der Datenbank zuzugreifen und mit diesen zu arbeiten. Für jede in der Schemadatei hinterlegte Entität werden im Build-Lauf insgesamt vier Dateien erzeugt. Am Beispiel der Entität *Blog* sind dies: *BaseBlog*, *BaseBlogPeer*, *Blog* und *BlogPeer*.

Die Klassen `Blog` sowie `BlogPeer` sind dabei Subklassen von `BaseBlog` und `BaseBlogPeer`. Die Basisklassen beinhalten die Logik, die von Torque für den Datenbankzugriff automatisch generiert wird. Diese Klassen werden auch bei jedem neuen Generierungslauf überschrieben. Alle Erweiterungen, die zusätzlich eingebaut werden, sollten somit in den Klassen `Blog` und `BlogPeer` erfolgen.

Die generierten Klassen selbst werden im Falle eines erfolgreichen Generierungslaufes im Unterverzeichnis *src/java* abgelegt und können von dort direkt in das eigene Projekt übernommen werden.

8.6.5 Erstellen der Datenbank

Nach dem Erzeugen der Modell- und Peerklassen kann die Datenbank angelegt werden. Torque hat bereits mit dem Erstellen der Java-Klassen zugleich Dateien zum Erstellen der Datenbank (*create-db.sql*) sowie zum Anlegen der Tabellen (*project-schema.sql*) erzeugt. Diese können ebenfalls mit Hilfe des Ant-Tools ausgeführt werden.

```
ant -f build-torque.XML create-db
ant -f build-torque.XML insert-sql
```

Da in den Tabellendefinitionen angegeben wurde, dass der Torque-eigene Mechanismus zur Erstellung von Ids verwendet wird, muss dieser Service ebenfalls angestoßen werden.

```
ant -f build-torque.XML id-table-init-sql
```

Damit ist die Arbeit bezüglich der Datenbankanbindung im ersten Schritt erledigt. Eine Speicherung und Abfrage der Daten aus der Datenbank erfolgt über einfache Java-Methoden und ist in der Anwendung selbst genau nachzuverfolgen.

8.6.6 Einbindung in die Anwendung

Torque selbst generiert bereits sämtliche Methoden, die für das Lesen, das Aktualisieren und das Speichern von Daten notwendig sind. Damit Torque beim Start der Anwendung zunächst initialisiert wird, ist die `init`-Methode in der eigenen Anwendung aufzurufen.

```
{
  if (initialized == false) {
    // Torque-DB initialisieren
    try {
      Torque.init( applicationPath + "WebLog.properties" );
      initialized = true;
    } catch (Exception e) {
      System.err.println("Error in Torque initialization " + e);
    } // catch
  } // if
}
```

Listing 8.5: Initialisieren von Torque

In der Anwendung wird eine statische Variable `initialized` verwendet, in der gespeichert wird, ob Torque bereits initialisiert wurde. Damit kann sichergestellt werden, dass die Initialisierungsroutine auch nur einmal aufgerufen wird. Als Argument wird der `init`-Methode der Pfad zur Konfigurationsdatei mitgegeben, damit daraus die not-

wendigen Einstellungen gelesen werden können. Nach erfolgreicher Initalisierung ist Torque bereit für den Einsatz. Eventuell müssen Sie an dieser Stelle Anpassungen an Ihre Umgebung vornehmen.

```
crit = new Criteria();
crit.add(BlogEntryPeer.BLOG_ID, aBlog.getBlogId());
crit.addDescendingOrderByColumn(BlogEntryPeer.CREATEDATE);
List l = BlogEntryPeer.doSelect(crit);
```

Listing 8.6: Ausführen eines Selects

Listing 8.6 zeigt das Ausführen eines einfachen *Selects*. Dabei kann der `doSelect`-Methode ein `Criteria`-Objekt mitgegeben werden, das quasi das *where*-Statement beinhaltet. Es kann hiermit die Anzahl der Datensätze eingegrenzt oder aber auch eine Sortierung mit angegeben werden. Wichtig ist, dass für die gesamte Aktion keinerlei SQL verwendet wird, sondern lediglich Methodenaufrufe zum gewünschten Ergebnis führen.

```
blog.setBlogId(WebLog_id);
blog.setUserId(user_id);
blog.setTitel(titel);
blog.setIntro(intro);
blog.save();
```

Listing 8.7: Speichern eines Objektes

Das Speichern eines Datensatzes funktioniert ähnlich elegant wie das Abfragen. Es werden mit Hilfe der setter-Methoden die erforderlichen Daten in ein Objekt geschrieben und anschließend mit der `save`-Methode der komplette Satz gespeichert. Dabei weiß Torque von selbst, ob es sich bei dem Datensatz um eine Neuanlage handelt (und somit ein *Insert* ausgeführt werden muss) oder um eine Aktualisierung eines bestehenden Datensatzes (und somit ein *Update* durchgeführt wird.).

In der Anwendung sind weitere Möglichkeiten zu finden, wie Datenbankoperationen mit Torque sehr einfach und effizient durchgeführt werden können.

8.7 Projekt einrichten

In einem guten Projekt startet die eigentliche Kodierungsphase relativ spät. Nicht aufgrund der Tatsache, dass die Programmierer vorher keine Lust dazu hatten, sondern aufgrund der Tatsache, dass vor der ersten Zeile Code viel Planungs- und Kommunikationsaufwand notwendig ist. Es gilt, das Projektziel mit dem Kunden genau zu definieren und somit frühzeitig mögliche Missverständnisse aus dem Weg zu räumen. Denn eine Änderung des Programmablaufs in der Konzeptionsphase ist leichter zu

realisieren als dann, wenn der fertige Programmcode bereits entwickelt ist. Auch in der beschriebenen Beispielanwendung wurde der Programmablauf mit Hilfe der Screenflow-Diagramme konzipiert und somit für den Kunden visuell dargestellt. Ebenfalls wurde die notwendige Datenstruktur modelliert und eine Technologie für die Datenbankanbindung (Torque) ausgewählt.

Mit diesen Vorarbeiten kann nun das eigentliche Projekt in der Entwicklungsumgebung eingerichtet werden. Bei Verwendung von Eclipse ist hierbei wieder ein Tomcat-Projekt zu erstellen. Als Projektname kann *JSFWebLog* verwendet werden. Dies ist zugleich die URI für die spätere Webanwendung.

Entsprechend den in den bisherigen Kapiteln durchgeführten Arbeitsschritten sind grundlegend zwei Konfigurationsdateien anzulegen: der Deployment-Deskriptor *web.XML* sowie die Anwendungskonfigurationsdatei *faces-config.XML*. Sollte Ihnen der Aufbau der Dateien nicht mehr geläufig sein, können Sie hierzu nochmals im Kapitel 6.1 Konfigurationsdateien nachschlagen. Für den Anfang genügt es zunächst einmal, eine leere Anwendungskonfigurationsdatei bereitzustellen. Diese wird im Laufe der Entwicklung des Projekts schrittweise vervollständigt.

Des Weiteren sind die für das Projekt notwendigen Bibliotheken in das *WEB-INF/lib*-Verzeichnis zu kopieren. Dies sind zum einen die Bibliotheken für JSF selbst sowie die Dateien für Torque. Für Torque ist ebenfalls die Laufzeit-Datei *WebLog.properties* bereitzustellen. Des Weiteren ist der MySQL-Jdbc-Treiber miteinzubinden, dies ist eine einzelne *.jar*-Datei. Genaueres zum Einrichten des Workspaces können Sie in 5.2 Einrichten der Entwicklungsumgebung ebenfalls nochmals im Detail nachlesen.

8.8 Managed-Beans

Basierend auf den Screenflow-Diagrammen und den Use-Cases der Anwendung können bereits erste JavaBeans definiert werden. Diese können ebenfalls als Managed-Beans in die Anwendungskonfigurationsdatei eingetragen werden, so dass diese in den JSF-Seiten zur Verfügung stehen. Im ersten Schritt werden folgende Beans benötigt:

- `UserBean`: Ein User stellt einen vollständig registrierten Benutzer der Plattform dar. Neben dem vollständigen Namen, einer E-Mail-Adresse und dem Loginnamen werden zusätzlich weitere Adressdaten mit einem User verwaltet. Da der User in der Datenbank abgespeichert wird, wird hierzu die Userklasse verwendet, die von Torque bereits generiert wurde.
- `LoginBean`: Ein registrierter Besucher der Community muss sich auf der Plattform anmelden können. Dazu wird ein Loginname sowie ein Passwort entgegengenommen und verifiziert. War die Anmeldung erfolgreich, wird ein Flag hinterlegt, das auf eine korrekte Anmeldung hinweist. Das LoginBean muss eine Gültigkeit wäh-

rend der gesamten Session haben, da nur so sichergestellt ist, dass jederzeit der Anmeldestatus eines Benutzers abgefragt werden kann. Eine dauerhafte Speicherung des Logins in der Datenbank ist nicht notwendig, da ein angemeldeter User bereits über die Usertabelle in der Datenbank vorhanden ist.

- `RegisterBean`: Zur Registrierung in der Community müssen ebenfalls Daten über die Oberfläche gesammelt und bei Bedarf in eine Datenbank gespeichert werden. Die Daten der Registrierung werden daher temporär in einem separaten Bean gesammelt.
- `BlogBean`: Ein Blog ist der Sammelbegriff für ein WebLog, sprich für ein Tagebuch. Ein Blog kann wiederum mehrere Einträge (BlogEntries) vorweisen. Da auch das Blog in der Datenbank gespeichert wird, wurde die Klasse Blog bereits durch Torque angelegt.
- `BlogEntry`: Ein Blog besteht aus mehreren BlogEntries, sprich aus mehreren Eintragungen. Dabei wird im Normalfall jeder Eintrag eines Tages als ein `BlogEntry` abgespeichert. Auch hierzu wird die von Torque generierte Klasse verwendet.
- `BlogEntryList`: Um sämtliche BlogEntries, also Tagebucheintragungen, in einer Listform oder einer Tabellenform ausgeben zu können, müssen alle Eintragungen gebündelt in einem `Vector` oder einer anderen `Collection` gesammelt werden. Hierzu wird ein Bean `BlogEntrylist` verwendet, das nicht persistent ist, sondern lediglich während der Anwendung alle aus der Datenbank gelesenen Eintragungen gesammelt vorhält.
- `BlogList`: Das, was eine `BlogEntryList` für die einzelnen Eintragungen darstellt, ist die `BlogList` für die WebLogs. Da auf der Startseite der Anwendung eine Auswahl an WebLogs abgebildet sind, müssen auch die WebLogs selbst für JSF in einer Collection vorgehalten werden. Die `BlogList` ist daher nur während der Session oder auch während des Requests vorhanden und wird nicht in der Datenbank abgespeichert bzw. lässt sich indirekt aus den einzelnen WebLog-Einträgen ermitteln.
- `PosterBean`: Ein `PosterBean` soll die zusätzlichen Userangaben speichern. Es ist somit der Steckbrief eines Benutzers. Da zwischen einem User und einem `PosterBean` eine 1:1-Beziehung vorliegt, hätte beides auch in einer Entität realisiert werden können. Aus Gründen der Übersichtlichkeit wurden jedoch zwei Entitäten entwickelt. Die Posterklasse selbst ist auch wieder eine von Torque generierte Klasse, da auch diese Informationen in der Datenbank gespeichert werden müssen.
- `BookEntryBean`: Für die einzelnen Gästebucheintragungen wird ebenfalls wieder die von Torque bereitgestellte Klasse verwendet.

- `GuestbookEntryList`: Während ein `BookEntryBean` einen einzelnen Gästebucheintrag vorhält, stellt die `GuestbookEntrylist` eine Liste aller für einen bestimmten Benutzer vorhandenen Eintragungen dar.

Da bereits durch den Generator von Torque viele Klassen bereitgestellt wurden, müssen nur noch wenige Klassen zusätzlich entwickelt werden. Es sind ausschließlich Klassen für Beans, die nicht persistent gehalten werden und nur für Anzeigefunktionen bereitgestellt werden.

```
package com.edu.jsf.WebLog.bean;

import java.util.ArrayList;
import com.edu.jsf.WebLog.dbbean.Blog;

/**
 * Bean, das eine Liste von verschiedenen
 * WebLogs vorhält
 */
public class Bloglist {

  private ArrayList WebLogs;

  /**
   * Konstruktor
   */
  public Bloglist() {
  }

  /**
   * Setter- und Getter-Methoden
   */
  public ArrayList getWebLogs() {
    return WebLogs;
  }

  public void setWebLogs(ArrayList list) {
    WebLogs = list;
  }
}
```

Listing 8.8: Die Klasse Bloglist

Listing 8.8 zeigt die `Bloglist`-Klasse. Diese ist sehr einfach aufgebaut, hält sie doch als einzige Eigenschaft lediglich eine `ArrayList` für einzelne WebLogs. Es wird deshalb ein Bean in dieser Form benötigt, um in den JSF-Seiten via Managed-Beans komfortabel darauf zugreifen zu können. Analog zu dieser Klasse sind auch die Klassen `Blogentrylist` und `Guestbookentrylist` aufgebaut.

```
package com.edu.jsf.WebLog.bean;

/**
 * Bean für das Login in die Community
 */
public class LoginBean {

  private String loginname;
  private String password;
  private boolean confirmed;

  /**
   * Getter-Methoden
   */
  public String getLoginname() {
    return loginname;
  }

  public String getPassword() {
    return password;
  }

  public boolean isConfirmed() {
    return confirmed;
  }

  /**
   * Setter-Methoden
   */
  public void setLoginname(String string) {
    loginname = string;
  }

  public void setPassword(String string) {
    password = string;
  }

  public void setConfirmed(boolean b) {
    confirmed = b;
  }
}
```

Listing 8.9: Das LoginBean

Auch in der `LoginBean`-Klasse, die in Listing 8.9 abgebildet ist, ist zunächst einmal nichts Spektakuläres zu finden. Das Flag `confirmed` beinhaltet, ob ein Besucher sich bereits erfolgreich auf der Plattform angemeldet hat. Im Laufe der Anwendungsentwicklung werden jedoch noch weitere Methoden in die Login-Klasse aufgenommen, die einen Teil der Funktionalität der Anwendung übernehmen.

Nachdem die notwendigen Bean-Klassen codiert sind, können die benötigten Managed-Bean Eintragungen in der Anwendungskonfigurationsdatei vorgenommen werden.

```
<managed-bean>
  <managed-bean-name>Login</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.LoginBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Blog</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.dbbean.Blog
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Poster</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.dbbean.Poster
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Register</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.RegisterBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Bloglist</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.Bloglist
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Blogentry</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.dbbean.BlogEntry
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

```
<managed-bean>
  <managed-bean-name>Blogentrylist</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.Blogentrylist
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>User</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.dbbean.User
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>GuestbookList</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.Guestbookentrylist
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>GuestbookEntry</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.dbbean.Bookentry</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Listing 8.10: Auszug aus der faces-config.XML

Wie in Listing 8.10 zu sehen ist, liegen fast sämtliche Beans in der Session. Dies ist jedoch mit Vorsicht zu genießen, da eine Session nicht zu groß werden sollte. Es ist nicht sehr elegant, einfach jegliche Daten in die Session zu legen. In obigem Beispiel wäre es auch möglich, die Gästebucheintragungen (`GuestbookList`) im Gültigkeitsbereich `request` abzulegen. Dies hätte jedoch die Konsequenz, dass bei jedem Anzeigen der Gästebucheinträge ein Datenbankzugriff erfolgen müsste, was zwar die Sessiongröße senken, den Kommunikationsaufwand mit der Datenbank aber erhöhen würde.

Die Bloglist hat den Gültigkeitsbereich `application`. Dies hat den Hintergrund, dass eine Liste von WebLogs sich nicht für jeden Besucher unterscheidet, sondern anwendungsweit gleich vorgehalten wird. Es muss nur sichergestellt werden, dass beim Anlegen eines neuen WebLogs die Änderungen in der Bloglist nachvollzogen werden.

8.9 Entwurf der Webseiten

Vom Webdesigner wurden die statischen HTML-Seiten geliefert. Diese vermitteln bereits einen ersten guten Eindruck von der Anwendung. Es empfiehlt sich, diese statischen Seiten mit dem Kunden bereits durchzusprechen, denn oftmals ergeben sich bereits hierbei Änderungsanforderungen. Da die Seiten noch nicht in JSF umgewandelt wurden, beschränkt sich der Änderungsaufwand hierbei auf ein Minimum. Denn je später Änderungen eingebracht werden, desto (zeit-)aufwändiger gestaltet sich der Prozess.

Sind die graphischen Vorlagen soweit in Ordnung, kann mit der Umsetzung der HTML-Seiten in JSF-Seiten begonnen werden. Dabei gehen Sie am besten schrittweise vor. Zunächst werden die Seiten von der Endung von *.HTML* in *.jsp* (oder auch *.jsf*) umbenannt. Danach können grundlegende Elemente (z.B. das `<f:view>`-Tag) integriert werden. Mit Fertigstellung der Beanklassen und nach Eintrag der Managed-Bean-Deklarationen können bereits erste Eingabe- und Ausgabeformulare umgesetzt werden. Aktionen, die z.B. hinter Links oder Commandbuttons stehen, können in einem späteren Schritt ergänzt werden.

HTML und JSF

Bei der Umsetzung von HTML-Seiten stellt sich oftmals die Frage, ob sämtlicher Inhalt von HTML-Tags in JSF-Tags umgewandelt werden soll. Grundsätzlich ist die Antwort auf die Frage ganz klar Ja! Da durch die Renderer in JSF das Ausgabeformat definiert wird, sollte in der eigentlichen Seite kein Tag eines speziellen Ausgabeformats auftauchen. In einer korrekten JSF-Seite sind somit lediglich JSF-Tags zu finden, für den HTML-Output sorgen einzig die Renderer. Damit wird es möglich, durch eine einfache Ersetzung des Renderer-Kits eine andersartige Ausgabe zu bekommen, ohne etwas an den Programmdateien verändern zu müssen.

Faktisch ist es jedoch so, dass dies ein enormer Aufwand ist. Es müssen eine Vielzahl von benutzerdefinierten Komponenten und Renderern entwickelt werden, da u.a. nicht einmal im Standard ein Tag vorgesehen ist, das die HTML-Basisangaben wie `<HTML>`, `<head>`, `<title>`, ... vornimmt. Daher ist es sicherlich legitim, bei kleineren Anwendungen HTML-Tags direkt mit einzubinden. Inoffiziell gibt es hierfür die Empfehlung, reine HTML-Ausgaben in ein `<f:verbatim>`-Tag einzubetten, womit wiederum eine »reine« JSF-Seite erzeugt wird. Wichtig ist jedoch, dass sich jeder Entwickler der Konsequenzen bewusst ist, die eine Vermischung von HTML- und JSF-Tags mit sich bringt und sich eventuell bewusst dafür oder dagegen entscheidet.

Um die Beispielapplikation nicht zu umfangreich werden zu lassen, wurden in den Beispielen ebenfalls bewusst HTML- und JSF-Tags in Kombination verwendet.

In der ersten Version der WebLog-Anwendung werden im Bereich der Benutzerregistrierung noch Standard-Eingabefelder verwendet. In einem späteren Kapitel werden dann der Aufbau und der Einbau von benutzerdefinierten Komponenten beschrieben. Bevor Sie an diesen Abschnitt gehen, sollten Sie jedoch das Kapitel 9 durcharbeiten. Denn um das gewünschte Verhalten zu erreichen, fehlerhafte Eingabefelder rot hinterlegen zu können, müssen benutzerdefinierte Komponenten (bzw. Renderer) zum Einsatz kommen.

8.10 Aufbau der Navigation

Die Navigation ist ein zentrales Element jeder Anwendung. Daher sollten auch bereits in der Konzeptionsphase einige Vorüberlegungen angestellt werden. Die Screenflow-Diagrammme geben bereits einen ersten Einblick, wie die Navigation ungefähr aussehen wird. Zusätzlich kann anhand der statischen Webseiten der Designer bestimmen, welche einzelnen Aktionen auf jeder Seite ausgeführt werden können, und gegebenenfalls auf weitere Seiten verweisen.

Wichtig ist dabei, zwischen der reinen Navigation und den möglichen Aktionen zu unterscheiden. In den Elementen, die später einen eventuellen Seitenwechsel auslösen (Hyperlink oder Button), wird bei Bedarf zusätzlich eine Aktion mitaufgerufen, wenn neben einem Seitenwechsel eine konkrete Aktion stattfinden soll.

Von Seite	**Mögliche Bezeichner**	**Beschreibung**
index.jsp	`show_register`	wechselt zur Registrierungsseite, bei der sich ein neuer Benutzer auf der Plattform anmelden kann
* (von allen Seiten)	`home`	verzweigt immer auf die zentrale Startseite
	`edit_WebLog`	Nach erfolgreicher Anmeldung kann der Benutzer sein eigenes WebLog bearbeiten.
	`edit_intro`	wechselt zur Bearbeitungsseite für die Introtexte.
	`edit_poster`	Bearbeitungsseite für die Steckbrief-Angaben
	`show_WebLog`	Anzeigeseite für WebLog-Einträge sowie den dazugehörigen Intro-Texten
	`show_poster`	Seite zur Anzeige der Steckbrief-Angaben
	`show_guestbook`	zeigt die vorhandenen Gästebucheinträge eines Benutzers. Ebenfalls können dort neue Eintragungen vorgenommen werden.

Tabelle 8.6: Navigationsmöglichkeiten (Auszug)

Tabelle 8.6 gibt einen ersten Überblick über die Navigationsmöglichkeiten der Anwendung. Natürlich muss ein Navigationsfall nicht explizit für jede einzelne Seite definiert werden, es ist natürlich auch möglich, seitenübergreifende Definitionen vorzunehmen. So ist in obiger Tabelle ein Bezeichner `home` eingeführt, der, egal auf welcher Seite man sich befindet, immer auf die Startseite leitet.

In der konkreten Entwicklung werden zusätzlich zu den Navigationsangaben bedarfsweise Aktionen ausgeführt. So führt das Registrieren nach einem Speichern-Knopf nicht wieder auf die zentrale Startseite zurück, sondern es wird konkret eine Aktion ausgeführt. In dieser Aktion wird ein neuer Datensatz in der Datenbank angelegt.

```
<h:commandButton action="#{Register.register}" value="Save" />
```

Im Attribut `action` wurde kein fester String als Konstante angegeben, sondern es wird eine Methode aufgerufen, deren Rückgabewert das weitere (Navigations-)Verhalten steuert.

```
public String register() {

  FacesContext context = FacesContext.getCurrentInstance();
  if (password == null
      || password2 == null
      || !password.equals(password2) {
    UIComponent comp =
           context.getViewRoot().findComponent("password");
    context.addMessage( comp.getClientId(context),
      new FacesMessage() {
        public String getDetail() {
          return "Die beiden Passwörter stimmen nicht überein.";
        }

        public Severity getSeverity() {
          return FacesMessage.SEVERITY_ERROR;
        }

        public String getSummary() {
          return "Die beiden Passwörter stimmen nicht überein.";
        }
      });
    } else {

      UserMgr aUserMgr = new UserMgr();
      aUserMgr.registerUser( loginname, firstname, lastname,
        street, zip, city, birth, password);
    } // else

return NavigationConst.HOME;
```

Listing 8.11: Aktionsmethode zur Registrierung

Zunächst wird in der Aktionsmethode aus Listing 8.11 überprüft, ob überhaupt ein Passwort eingegeben wurde und dieses mit der zweiten Kontrolleingabe auch übereinstimmt. Ist dies nicht der Fall, wird für die Komponente eine Fehlermeldung erzeugt und an die Komponente selbst angehängt. In diesem Fall wird die Fehlermeldung direkt durch die Klasse `FacesMessage` erzeugt. Wenig elegant ist die Tatsache, dass der Meldungstext direkt im Programmquellcode hinterlegt ist. Als Verbesserung kann an dieser Stelle auf eine Ressourcendatei zugegriffen werden (vgl. dazu Kapitel 6.13 Internationalisierung).

War die Überprüfung des Passworts positiv, wird über eine Klasse `UserMgr` der Benutzer in der Datenbank angelegt. Wenn Sie jetzt meinen, dass lediglich die Überprüfung des Passworts zu wenig sei, sei an dieser Stellt gleich erwähnt, dass in der JSF-Seite selbst mittels Validatoren eine erforderliche Eingabe bei den anderen Feldern abgefangen wurde.

Als Rückgabewert liefert die Methode die Konstante `HOME` zurück, die im Erfolgsfall auf die zentrale Startseite verweist. Liegen jedoch im Kontext Fehlermeldungen vor, wird automatisch auf derselben Seite verblieben und gegebenenfalls die Fehlermeldungen zur Anzeige gebracht. Eventuell könnten an dieser Stelle weitere Sonderfälle abgefangen werden, bei denen z. B. nach einer Anmeldung zunächst auf eine allgemeine Begrüßungsseite weitergeleitet wird. An dieser Stelle ist somit genügend Raum für eigene Kreativität gegeben.

8.11 Implementieren der Funktionalität

Nachdem das Grundgerüst erzeugt und die einzelnen JSF-Seiten im Grundbau erstellt wurden, kann die eigentliche Funktionalität implementiert werden. In den einzelnen Formularen können die Buttons, die eine Aktion auslösen sollen, mit einer Aktionsmethode verbunden werden. In der Beispielanwendung wird die Funktionalität in der Form implementiert, dass es einen zentralen WebLog-Manager in Form der Klasse `WebLogMgr` gibt, in dem sämtliche Funktionalitäten für das Einfügen, Löschen und Selektieren von Daten enthalten sind.

Mit Hilfe der Managed-Beans werden die Fomulare für die Ein- und Ausgabe vervollständigt und schrittweise deren korrektes Verhalten geprüft.

Eine Besonderheit wurde jedoch der Anwendung noch hinzugefügt, die zu Beginn der Entwicklungsarbeiten noch nicht ganz klar gewesen ist. Da auf der Startseite bereits eine Liste mit vorhandenen WebLogs von Benutzern angezeigt werden soll, ist es notwendig, dass das Bean, das diese Daten bereithält, vor dem ersten Seitenaufruf bereits befüllt werden müsste. Dies wird in der Beispielanwendung in der Form realisiert, dass es eine neue Startseite gibt, auf der lediglich ein Logo sowie ein kurzer Begrüßungstext vorhanden ist. Dies ist bereits die erste JSF-Seite. Klickt ein Benutzer das

Logo oder den Schriftzug an, kann mittels der Aktionsmethoden eine Routine aufgerufen werden, die die notwendigen Daten für die WebLog-Liste bereitstellt. Danach wird auf die eigentliche Startseite verwiesen, auf der dann wiederum die Daten der WebLog-Liste bereitstehen und angezeigt werden können.

Das Thema des Bereitstellens von Beans vor dem ersten Zugriff ist ein sehr wichtiges und in der Praxis sehr häufig auftretendes Problem. Zur Lösung existieren mehrere Varianten. Eine davon ist das Vorschalten einer anderen Faces-Seite, wie es in diesem Beispiel demonstriert wird. Eine weitere Variante ist der Einsatz eines *FrontController-Servlets*. Diese Vorgehensweise wird im weiteren Verlauf der Entwicklung dieser Beispielanwendung ebenfalls erklärt und eingesetzt. Die dritte Möglichkeit ist das Registrieren eines *ServletContextListeners*. Darauf wird in Kapitel 10.1 Bereitstellen von Managed-Beans näher eingegangen.

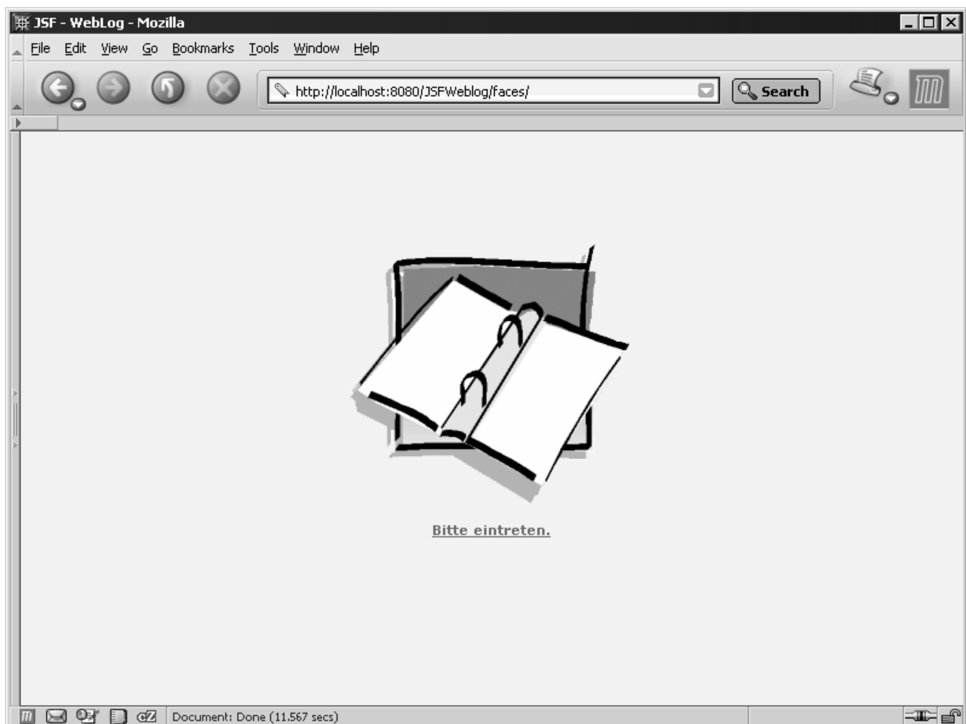


Abbildung 8.10: Neue Startseite für die WebLog-Anwendung

Damit ist die WebLog-Anwendung in einer ersten Ausbaustufe fertig gestellt. Die besondere Darstellung der Fehlermeldungen wird im folgenden Abschnitt nochmals aufgeriffen und näher behandelt. Sämtliche erforderlichen Quelldateien und JSF-Seiten sind natürlich auf der beigelegten CD enthalten.

8.12 Erweiterungen des WebLogs

Die WebLog-Anwendung wurde in den letzten Abschnitten zu einer vollständigen und lauffähigen Anwendung entwickelt. Sie ist in ihrer jetzigen Fassung durchaus einsetzbar. Da es jedoch auch eine alte »Softwareweisheit« ist, dass eine Software eigentlich nie fertig ist, werden auch in den folgenden zwei Abschnitten einige Erweiterungen eingebaut. Dies ist zum einen eine sehr benutzerfreundliche Fehlerdarstellung, zum anderen ein zusätzliches Feature, das gerade im WebLog-Umfeld häufig anzutreffen ist: ein *RSS-Newsfeed*.

Beide Erweiterungen werden mit Hilfe von benutzerspezifischen Komponenten realisiert, da diese speziellen Erweiterungen mit den Standardkomponenten nicht realisiert werden können. Es empfiehlt sich daher, das Kapitel 9 über die Erstellung benutzerspezifischer Komponenten im Vorfeld zu lesen, um das notwendige Grundlagenwissen im Bezug auf eigene Komponenten zu haben.

8.12.1 Benutzerfreundliche Fehlermeldungen

Eine weniger komfortable Stelle in der bisherigen Anwendung ist sicherlich die Benutzerregistrierung, bzw. deren Fehlermeldungen. Werden bei der Anmeldung einzelne Felder nicht angegeben, erscheinen die Fehlertexte gesammelt in einer Zeile am Ende des Formulars. Da der Bezug zum ursprünglichen Feld fehlt, sorgt dies oftmals für Verwirrung, wenn mehrere Eingabefelder frei gelassen wurden oder mehrere Fehler aufgetreten sind.

Vom Webdesigner der Seiten wurde bereits zu Beginn der Vorschlag eingereicht, die Eingabefelder farblich zu hinterlegen, bei denen Fehler aufgetreten sind (vergleiche Abbildung 8.7). So sieht der Benutzer gleich, an welchen Stellen weitere Eingaben zu tätigen sind. Des Weiteren sollen die Fehlermeldungen direkt am Eingabefeld und nicht mehr gesammelt in einer Zeile erscheinen. Unschön ist auch die allgemeine Fehleraussage, dass ein Wert benötigt wird. Hilfreicher wäre eine etwas ausführlichere Fehlermeldung, z. B. für welches Feld ein Wert benötigt wird.

In den Vorschlägen aus Abbildung 8.7 ist zu erkennen, dass die farbliche Unterlegung mit Hilfe von Stylesheets realisiert wurde. So wird die Farbe – Rot für Fehler und Transparent für kein Fehler – im Stylesheet bestimmt. Die Aufgabe des Entwicklers ist es jetzt, dynamisch aufgrund der Validierung die passenden Stylesheet-Angaben zu verwenden.

Die Standardkomponenten von JSF bieten jedoch nicht die Möglichkeit, dynamisch eine Stylesheet-Angabe mit einzubinden. Daher ist es erforderlich, hier eine benutzerspezifische Komponente zu verwenden.

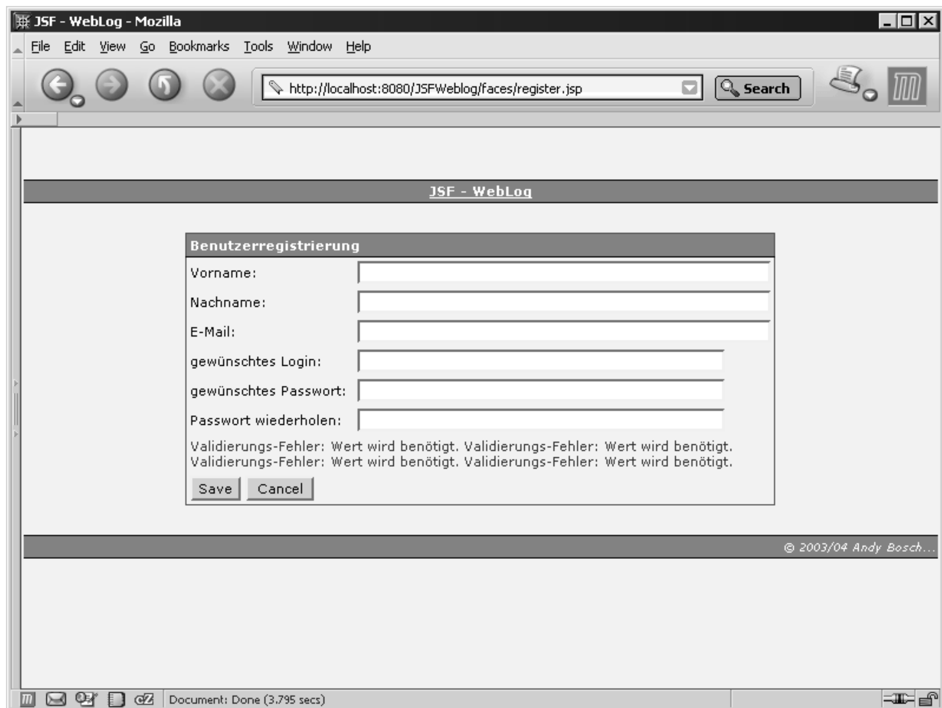


Abbildung 8.11: Fehlerdarstellung, die für Verwirrung sorgt

Vorgehensweise

Zunächst wird eine benutzerspezifische Komponente entwickelt. Da die meiste Funktionalität aus der eines bereits vorhandenen Eingabefelds `UIInput` verwendet werden kann, kann hier mittels Vererbung auf viele Klassen zurückgegriffen werden. Im Anschluss daran können dann die Validatoren eingebaut werden, die für die Prüfung der Mindesteingabelänge verantwortlich sind. Ebenfalls kann der Fehlertext angepasst werden. Als letzten Schritt kann daraufhin die bestehende JSF-Seite umgebaut werden. In diesem Zusammenhang wird auch die bereits vorhandene Klasse `RegisterBean` etwas modifiziert, in dem bei ungleichen Passwörtern die Fehlermeldung nicht mehr direkt aus dem Quellcode angezogen wird, sondern aus einer separaten Konfigurationsdatei. Dabei wird auf das Konzept der Fehlertexte nochmals eingegangen.

Die Tagklasse sowie der TLD

Die neu zu schaffende Komponente wird mit `UIInputError bezeichnet`, um deutlich zu machen, dass die Fehlerdarstellung etwas anders als mit den üblichen JSF-Mitteln erfolgt. Da ein Großteil der Funktionalität aus der Standard- `UIInput`-Komponente verwendet werden kann, wird die neue Komponente von `UIInput` erben. Um dynamisch

die Style-Angabe setzen zu können, müssen in der Tag-Beschreibung für die Komponente des Weiteren zwei Attribute vorgesehen werden: ein Attribut für die normale Styleangabe, eines für den Fehlerfall.

Zunächst gilt es daher, eine Tagklasse zu entwickeln. Auch an dieser Stelle wird auf die bereits vorhandene Tagklasse des `inputText`-Tags zurückgegriffen. Aus der *Html_basic.tld* ist zu ersehen, dass die dazugehörige Tagklasse `com.sun.faces.taglib.html_basic.InputTextTag` ist. Diese Klasse wird in einer spezifischen Tagklasse für die benutzerspezifische Komponente erweitert.

```
package com.edu.jsf.WebLog.tag;

import javax.faces.component.UIComponent;

import com.sun.faces.taglib.Html_basic.InputTextTag;

/*
 * Tag-Klasse für die InputError-Komponente.
 * Diese zeigt bei einem Fehler das Feld
 * mit rotem Hintergrund an.
 */
public class InputErrorTag extends InputTextTag {

  private String normalStyle;
  private String errorStyle;

  /*
   * liefert den Renderer zurück
   */
  public String getRendererType() {
    return "InputErrorText";
  }

  /*
   * liefert den Komponententyp zurück.
   * Eine entsprechende Eintragung dazu ist in
   * der faces-config.XML zu finden.
   */
  public String getComponentType() {
    return "InputErrorText";
  }

  /*
   * setzt die Properties in die Komponente
   */
  protected void setProperties(UIComponent component) {
    super.setProperties(component);
    UIInputError inputcomponent = (UIInputError)component;
    if (normalStyle != null
      && inputcomponent.getNormalStyle() == null)
```

```
      inputcomponent.setNormalStyle( normalStyle );
    if (errorStyle != null
      && inputcomponent.getErrorStyle() == null)
       inputcomponent.setErrorStyle( errorStyle );
  }

  /*
   * Getter-Methoden
   */
  public String getErrorStyle() {
    return errorStyle;
  }

  public String getNormalStyle() {
    return normalStyle;
  }

  /*
   * Setter-Methoden
   */
  public void setErrorStyle(String string) {
    errorStyle = string;
  }

  public void setNormalStyle(String string) {
    normalStyle = string;
  }
}
```

Listing 8.12: Die Tagklasse für die benutzerspezifische Fehler-Komponente

Die Tagklasse in Listing 8.12 ist nicht sonderlich spektakulär. Sie weist zwei Eigenschaften auf, die die jeweilige Stylesheet-Angabe für die Normaldarstellung und die Fehlerdarstellung beinhalten. Dazu passend existieren die erforderlichen setter- und getter-Methoden. Des Weiteren werden diese Werte in der Methode `setProperties` in die Komponente gesetzt. Sehr wichtig ist in dieser Methode der Aufruf von `super.setProperties(), da nur so sichergestellt ist, dass alle weiteren Attribute (z.B. size) korrekt in die Komponente gesetzt werden.`

`Die Methode getRendererType` liefert zugleich einen Bezeichner zurück, mit dem in einem folgenden Schritt der dazugehörige Renderer für die Komponente entwickelt wird. Der Bezeichner wird in der Anwendungskonfigurationsdatei hinterlegt, daher ist die genaue Schreibweise zu beachten. In `getComponentType` wird wiederum ein Bezeichner zurückgeliefert, der die eigentliche Komponentenklasse identifiziert. Auch dieser Bezeichner muss später in der Anwendungskonfigurationsdatei hinterlegt werden.

Damit die neue Tagklasse auch angezogen und verwendet werden kann, muss in einem passenden TLD (Taglibrary Deskriptor) ein entsprechendes Tag definiert werden.

```
<?XML version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">

<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>Custom Tag Library for JSF</short-name>
  <uri>http://java.sun.com/jsf/HTML</uri>
  <description>
    Further tags for JSF
  </description>

  <tag>
    <name>inputErrorText</name>
    <tag-class>com.edu.jsf.WebLog.tag.InputErrorTag</tag-class>
    <body-content>none</body-content>
    <attribute>
      <name>normalStyle</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>errorStyle</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>value</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>size</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>id</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Listing 8.13: Der Taglibrary-Deskriptor WebLog.tld

Der TLD aus Listing 8.13 wird unter dem Namen *WebLog.tld* im *WEB-INF*-Verzeichnis der Anwendung abgespeichert. Im Gegensatz zu den Tagklassen können die Beschreibungen im TLD nicht vererbt werden, daher müssen auch alle Attribute, die im neuen Tag verwendet werden, explizit angeben werden, da ansonsten durch den Servletcontainer ein Fehler beim Kompilieren der Seite geworfen wird. Der Einfachheit halber können natürlich auch alle definierten Attribute aus der `Html_basic.tld` kopiert werden. Wichtig für die Anwendung sind hauptsächlich die beiden neuen Attribute `normalStyle` und `errorStyle`. Diese sind durch die Angabe des `required`-Attributes zwingend erforderlich. Natürlich wäre es auch möglich, diese beiden neuen Attribute auf nicht zwingend erforderlich zu setzen. In der Rendererklasse müsste dies berücksichtigt werden, da in diesem Falle ein Standardwert angezogen werden muss. Dadurch jedoch, dass diese Angaben für erforderlich deklariert werden, können Sie sich in der Rendererklasse auf das Vorhandensein der Werte verlassen und es muss kein eventueller Fehler- oder Sonderfall berücksichtigt werden.

Die UIInputError-Komponente

Nachdem die Tagklasse angelegt wurde, kann die eigentliche Komponentenklasse erstellt werden. Diese Klasse wird ebenfalls auf Funktionalitäten der in JSF bereits vorhandenen `UIInput`-Klasse zurückgreifen. Da das Rendering nicht in der Komponente selbst, sondern in einem speziellen Renderer vorgenommen wird, ist die Komponentenklasse relativ spartanisch aufgebaut.

```
package com.edu.jsf.WebLog.tag;

import javax.faces.component.UIInput;

/**
 * Komponentenklasse für TextInputFeld
 */
public class UIInputError extends UIInput {

  private String normalStyle;
  private String errorStyle;

  /**
   * Getter-Methoden
   */
  public String getErrorStyle() {
    return errorStyle;
  }

  public String getNormalStyle() {
    return normalStyle;
  }
```

```
  /**
   * Setter-Methoden
   */
  public void setErrorStyle(String string) {
    errorStyle = string;
  }

  public void setNormalStyle(String string) {
    normalStyle = string;
  }
}
```

Listing 8.14: Die Komponentenklasse UIInputError

Damit die Komponente letztlich auch in JSF-Seiten verwendet werden kann, muss diese in der Anwendungskonfigurationsdatei eingetragen werden. Dazu wird ebenfalls wieder der Bezeichner `InputErrorText` verwendet, der bereits in der Tagklasse zu finden ist. An dieser Stelle ist zwingend darauf zu achten, dass der Bezeichner korrekt hinterlegt wird.

```
<component>
  <component-type>InputErrorText</component-type>
  <component-class>
    com.edu.jsf.WebLog.tag.UIInputError
  </component-class>
</component>
```

Listing 8.15: Bekanntmachen der Komponente in faces-config.XML

Das Rendering

Wie bereits in den anderen entwickelten Klassen wird auch in der Renderer-Klasse auf bestehende Funktionalitäten der `UIInput`-Komponente zurückgegriffen. Der Renderer, der für das `inputText`-Tag angezogen wird, ist die Klasse `TextRenderer`. Diese wird somit für die Renderer-Klasse als Basisklasse verwendet.

```
package com.edu.jsf.WebLog.tag;

import java.io.IOException;
import java.util.Iterator;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;

import com.sun.faces.renderkit.Html_basic.TextRenderer;
import com.sun.faces.util.Util;

/*
```

```
 * Rendererklasse für die Usagebar-Komponente
 */
public class InputErrorRenderer extends TextRenderer {

  public void encodeBegin(FacesContext context,
      UIComponent component) throws IOException {
    if (context == null || component == null)
      throw new NullPointerException(
        Util.getExceptionMessage(
        "com.sun.faces.NULL_PARAMETERS_ERROR"));

    UIInputError inputcomponent = (UIInputError)component;

    Iterator itr =
      context.getMessages( component.getClientId(context) );
    int size = 0;
    while ( itr.hasNext() ) {
      size++;
      itr.next();
    } // while

    String cls = "";
    if ( size>0 )
      cls = inputcomponent.getErrorStyle();
    else
      cls = inputcomponent.getNormalStyle();

    component.getAttributes().put("styleClass", cls );
  }
}
```

Listing 8.16: Renderer-Klasse

Das Erben von bereits vorhandenen Komponenten ist ein sehr sinnvoller und häufig auftretender Fall bei der Entwicklung eigener Komponenten. Selten kann es sicherlich vorkommen, dass eine komplett neuartige Komponente realisiert werden soll. In den meisten Fällen werden jedoch spezielle Eigenschaften oder Verhaltensweisen von vorhandenen Komponenten benötigt, so dass auf die Grundfunktionalitäten der Komponenten durch das Prinzip der Vererbung zugegriffen werden kann. Eine Entwicklung eigener Komponenten wird somit wesentlich beschleunigt. Wie in Listing 8.16 gut zu erkennen ist, wurde für die Renderer-Klasse lediglich die Methode `encodeBegin` überschrieben. Ob ein Fehlerfall für eine Komponente vorliegt, ist daran zu erkennen, dass Nachrichten an dieser Komponente angefügt sind. Über den Aufruf

```
context.getMessages(component.getClientId(context) )
```

werden Nachrichten zurückgeliefert, die genau für diese Komponente vorliegen. Wird dieselbe Methode ohne Parameter aufgerufen, werden alle vorliegenden Nachrichten zurückgegeben.

Für den zurückgelieferten Iterator wird in einer `while`-Schleife die Größe ermittelt. Ist diese größer als 0, liegen Meldungen vor und der Fehlerfall tritt ein. Daraufhin wird das Attribut `styleClass` auf die Fehler-Styleangabe gesetzt.

Hinzuweisen ist zudem noch auf den Cast

```
UIInputError inputcomponent = (UIInputError)component;
```

Als Parameter der Methode `encodeBegin` wird immer eine Komponente der Klasse `UIComponent mit übergeben`, die die Basisklasse für alle UI-Komponenten darstellt. Da im Beispiel davon ausgegangen werden kann, dass der Renderer nur für `UIInputError`-Komponenten aufgerufen wird, kann der Cast auf die Klasse `UIInputError` ohne Bedenken durchgeführt werden.

Validierung

Bevor die Komponente getestet werden kann, ist im Vorfeld noch ein passender Validator an die Komponente anzuhängen, mit dem überhaupt erst Fehlermeldungen an die Komponente angehängt werden können. Für die Beispielanwendung werden zwei Validatoren verwendet: zum einen der *Required*-Validator, der zunächst einmal überprüft, ob überhaupt ein Wert eingegeben wurde. Damit auch die korrekte Mindestlänge eingehalten wurde, wird der *ValidateLength*-Validator eingesetzt.

Wie bereits in Kapitel 6.6 Meldungen und Fehlermeldungen gesehen, sind die Standardfehlermeldungen nicht immer sehr glücklich gewählt. Es ist also empfehlenswert, an dieser Stelle eigene Fehlertexte zu hinterlegen. Dazu muss zunächst eine entsprechende Datei in der Anwendungskonfigurationsdatei hinterlegt werden.

```
<application>
  <message-bundle>WebLogmessages</message-bundle>
</application>
```

Diese Datei muss im Klassenpfad der Anwendung zu finden sein. In der Datei selbst können dann die Fehlertexte mittels der Schlüssel der Standardfehlermeldungen überschrieben werden (vgl. dazu Kapitel 6.6).

```
javax.faces.validator.LengthValidator.LIMIT=
  Die Eingabe ist zu kurz. Bitte einen längeren
  Bezeichner eingeben.
javax.faces.component.UIInput.REQUIRED=
  Eine Eingabe ist erforderlich.
javax.faces.validator.LengthValidator.MINIMUM=
  Die Eingabe ist zu kurz.
```

Listing 8.17: Auszug aus WebLogmessages.properties

Listing 8.17 zeigt einen Ausschnitt aus der Fehlerdatei, in der die Standardfehlermeldungen durch eigene Texte überschrieben wurden.

Anpassen der JSF-Seiten

Der Umbau der JSF-Seiten gestaltet sich relativ einfach. Da das Stylesheet bereits in der Seite verlinkt ist, muss an dieser Stelle keine Änderung vorgenommen werden. Es muss jedoch für die Verwendung der neuen Komponenten bzw. Tags die entsprechende Tag-Bibliothek eingebunden werden.

```
<%@ taglib uri="/WEB-INF/WebLog.tld" prefix="cst" %>
```

Hierbei wird unterstellt, dass die entsprechende Bibliothek direkt im *WEB-INF*-Verzeichnis der Anwendung zu finden ist. Als Namensraum wurde *cst* für *custom* gewählt.

Die Komponente wird in die entsprechenden Bereiche über die Syntax

```
<cst:inputErrorText normalStyle="userInput" id="firstname"
     errorStyle="userInputError" value="#{Register.firstname}"
     size="50" required="true">
  <f:validateLength minimum="2" />
</cst:inputErrorText>
<h:message for="firstname" />
```

eingebaut. Wichtig ist, dass innerhalb des `inputErrorText`-Tags der Validator gesetzt wird. Wichtig ist es auch, dass der Eingabekomponente eine eindeutige Id explizit mitgegeben wird, da sich nur so die Fehlerausgabe direkt darauf beziehen kann.

Des Weiteren wird für jede Eingabekomponente ein separates `message`-Tag verwendet, so dass die Fehlermeldungen passend zum Eingabefeld direkt unterhalb dargestellt werden können.

Fazit

Durch die beschriebenen Erweiterungen in den letzten Abschnitten konnten hinsichtlich der Benutzerfreundlichkeit der Anwendung einige Verbesserungen erreicht werden. Als mögliche Erweiterung würde sich jetzt noch anbieten, die Passworteingabefelder ebenfalls mit eigenen Komponenten zu ersetzen, damit auch diese einen farblichen Hintergrund im Falle einer Fehlermeldung erhalten können. Bei Verwendung von Datumseingabefeldern müssten gegebenenfalls auch diese durch benutzerspezifische Komponenten bzw. Renderer abgelöst werden, um an dieser Stelle ein einheitliches Verhalten erreichen zu können.

8.12.2 RSS-Newsfeed

Als Krönung der WebLog-Anwendung soll jetzt noch mit dem Bereitstellen eines so genannten News-Feeds ein sehr attraktives Feature entwickelt werden. Damit können Nutzer so genannter RSS-News-Reader aktuelle Einträge in einzelnen Blogs komfortabel lesen und sind immer über Aktualisierungen auf dem Laufenden. Technisch interessant in diesem Zusammenhang ist, dass entgegen aller bisherigen Beispiele jetzt als Ausgabeformat nicht mehr HTML, sonderm XML benötigt wird. Es wird somit in diesem Beispiel demonstriert, wie mit Hilfe von JSF auch andere Ausgabeformate erzeugt werden können.

Was ist RSS?

RSS ist eine Methode zum Austausch von Daten. Die Daten liegen dabei in einem genau spezifizierten XML-Format vor. Wofür RSS genau steht, ist umstritten. So kursieren Meinungen, RSS stehe für *Rich Site Summary* oder auch für *RDF Site Summary.* Aber auch der Begriff *Really Simple Syndication* taucht regelmäßig auf. Entscheidend bei RSS ist jedoch, dass damit ein Format geschaffen wurde, mit dem auf einfache Weise Nachrichten jeglicher Art ausgetauscht werden können. So kann beispielsweise ein Nachrichtenportal die einzelnen Meldungen nicht nur mittels HTML-Seiten im Web anbieten, sondern auch im RSS-Format. Damit haben andere Anbieter die Möglichkeit, auf diesen so genannten *RSS-Channel* zuzugreifen und die Nachrichten in die eigene Webseite zu integrieren. Ohne RSS wäre dies prinzipiell zwar auch möglich, es müsste jedoch ein Programm zunächst die HTML-Seite untersuchen und den reinen Nachrichtengehalt aus der Menge aller Graphiken, HTML-Tags und Stylesheet-Angaben heraussuchen. Programmgestützt ist dies nur bedingt realisierbar. RSS dagegen hat durch seine standardisierte Struktur den Vorteil, dass jeder, der sich mit RSS befasst, genau weiß, an welcher Stelle in der XML-Datei welche Information abgelegt wird.

Doch nicht nur für Contentprovider (Anbieter von Content), sondern auch für Endnutzer weist RSS einige wesentliche Vorteile auf. So können mittels RSS-Reader Nachrichten verschiedener Portale und Nachrichtenanbieter automatisiert nach Neuigkeiten abgesucht werden und diese in einheitlicher Form im Reader-Programm dargestellt werden. Ein Benutzer muss somit nicht mehr auf alle für ihn relevanten Webseiten einzeln gehen, nur um dann festzustellen, dass keine Neuigkeiten vorliegen.

Diese in Abbildung 8.12 aufgelisteten Nachrichten könnten über einen RSS-Feed in einem XML-Format versendet und beispielsweise auch in andere Portale eingebaut werden. In der WebLog-Beispielanwendung ist es vorstellbar, dass Einträge des persönlichen WebLogs neben der Anzeige im WebLog-Portal, zeitgleich auch auf einer persönlichen Homepage erscheinen können.

Abbildung 8.12: Aktuelle Nachrichten auf yahoo.de

RSS-Versionen

Der Grundgedanke von RSS ist sicherlich sehr lobenswert, nämlich ein einfaches und standardisiertes Format für den Datenaustausch zu etablieren. In der Praxis jedoch herrschen aktuell eine Menge an RSS-Spezifikationen vor, die untereinander nicht kompatibel sind und auch nicht von allen Reader-Programmen gelesen werden können.

Begonnen hatte alles im Juni 2000, als Netscape das Format RSS 0.90 bzw. 0.91 herausgebracht hatte. Damals als Teil der Strategie des *My Newscape*-Portals angedacht, fand es sehr schnell eine weite Verbreitung bei den Portalbetreibern. In der ursprünglichen Version war es allerdings nur möglich, Überschriften zu hinterlegen, die dann wiederum auf eine komplette Meldung per Link verwiesen. Erst mit der leicht überarbeiteten Version 0.91 wurde es möglich, zu einer Überschrift einen weiteren kurzen Beschreibungstext zu hinterlegen.

Dieses von Netscape definierte Format fand daraufhin einen großen Interessenten- und Nutzerkreis, die Anzahl der bereitgestellten so genannten Channels wuchs steil an. So wurden in CMS- (Content-Management-System) und Portalsystemen bereits Möglichkeiten geschaffen, fremde RSS-Channels zu integrieren oder Nachrichten im

RSS-Format bereitzustellen. Die Weiterentwicklung des Formats wurde jedoch nicht nur von Netscape, sondern auch von weiteren Firmen vorangetrieben. So stellte die Firma Userland die Versionen 0.92, 0.93 und 0.94 bereit, die zunächst noch auf dem Netscape-Format aufbauten. Die Firma Userland ist ein professioneller (kommerzieller) Anbieter von WebLogs im Internet. Im Laufe der Entwicklung wurde dann auch das Format 2.0 entwickelt, das wesentliche Erweiterungen und Neuerungen gegenüber den 0.9x-Versionen beinhaltete. Da jedoch in der Zwischenzeit das Format 1.0 vom W3C-Konsortium standardisiert wurde, herrschten auf einmal zwei ähnlich weit entwickelte Standards im RSS-Markt vor. Somit ist das Format 2.0 nicht die Weiterentwicklung von 1.0, sondern ein eigener Entwicklungsast. In den hier gezeigten Beispielen wird das Format 0.91 verwendet, da es Stand heute von den allermeisten Programmen verarbeitet werden kann.

Die Parallelentwicklung von 1.0 und 2.0 ist sicherlich keine sehr ruhmreiche Entwicklung eines Standards, jedoch muss auch gesagt werden, dass heutige RSS-Reader-Programme zumeist mehrere Standards verstehen und somit für den Nutzer der Versionskonflikt nicht direkt zum Tragen kommt.

Verwendung von RSS

Um RSS verwenden zu können, sind zwei Dinge notwendig: Zum einen muss ein so genannter Contentprovider Nachrichten oder Meldungen in Form eines RSS-Stromes bereitstellen. Zumeist erkennt man dieses an einem kleinen XML-Bild auf einer Webseite.

Abbildung 8.13: Hinweis auf einen RSS-Feed

Für das Empfangen und Lesen von Nachrichten eines RSS-Channels existieren mittlerweile eine Vielzahl von freien und kommerziellen Programmen. Auch nur eine Auswahl der Programme würde den Rahmen dieses Kapitels weit sprengen. Für die Beispiele dieses Buches wird der *HotSheet* verwendet. HotSheet kann unter der Adresse

http://www.johnmunsch.com/projects/HotSheet/

heruntergeladen werden. Auf den Webseiten sind ebenfalls Dokumentationen zum Programm zu finden. HotSheet zeichnet sich vor allem dadurch aus, dass es eine reine Java-Anwendung und daher auf verschiedenen Plattformen (Windows, Linux, Macintosh, ...) lauffähig ist. Es gibt jedoch noch viele weitere Programme, die über eine einfache Internetrecherche sehr schnell gefunden werden können.

Nach der Installation des RSS-Readers genügt es im Normalfall, die Channels (also die Adressen der Nachrichtenanbieter) im Programm zu hinterlegen, alles Weitere wird durch das Programm selbst erledigt. Oftmals kann auch hinterlegt werden, in welchen Zeitabständen nach neuen Meldungen gesucht und ob bei neuen Meldungen ein akkustischer Hinweis abgespielt werden soll.

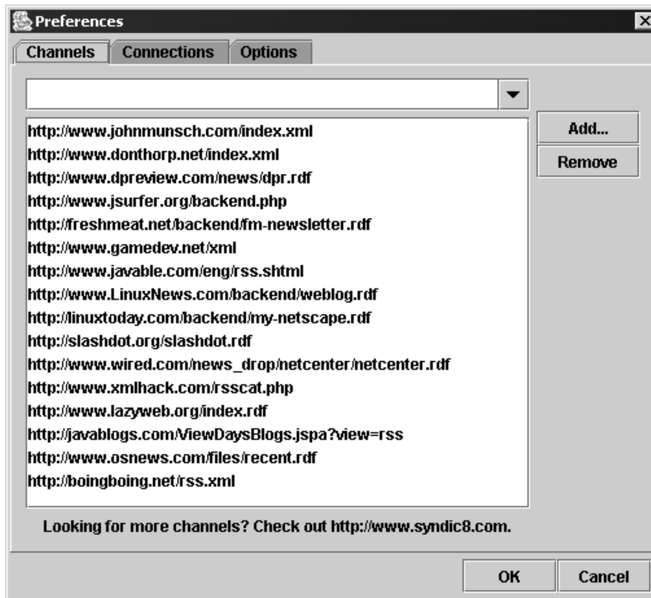


Abbildung 8.14: Standard-News-Liste in HotSheet

Wie in Abbildung 8.14 zu sehen, werden die meisten Reader-Programme gleich mit einer Liste von Channels mitgeliefert. Diese können jederzeit entfernt oder aber um neue ergänzt werden. Sobald die Beispiel-WebLoganwendung einen eigenen Channel bereitstellt, wird dieser in dieser Liste mit hinterlegt.

Dynamisches Generieren einer RSS-Datei

Für die Beispielanwendung soll ebenfalls ein RSS-Channel bereitgestellt werden, über den die letzten Einträge eines WebLog-Benutzers abrufbar sind. Da ein RSS-Channel letzten Endes immer eine XML-Datei ist, muss diese Datei dynamisch generiert werden. Als Parameter ist die User-Id mitzugeben, damit eine Unterscheidung auf Nutzerebene vorgenommen werden kann. Es sollen im Newsfeed die letzten fünf Eintragungen eines Nutzers angezeigt werden.

Ein alternativer Ansatz wäre es, die XML-Datei nicht beim Abruf zu generieren, sondern durch ein separates Programm vollständig zu erzeugen und diese als »richtige« XML-Datei im Dateisystem bereitzustellen. Im gezeigten Beispiel wird jedoch der Weg gewählt, dass die XML-Datei eine JSF- bzw. JSP-Datei ist, die erst zur Laufzeit Ausgaben in Form des XML-Formats erzeugt. Der Vorteil hierbei ist, dass kein nächtlicher oder stündlicher Generierungslauf notwendig ist, sondern die Datei jeweils aktuell beim Abruf erzeugt wird. Auch kann durch die Parameterübergabe die Ausgabe beeinflusst werden.

Da eine RSS-Datei im Normalfall die Dateiendung `rds` aufweist, gibt es zwei Möglichkeiten, diese Datei dynamisch generieren zu lassen: Zum einen kann der Application Server so konfiguriert werden, dass auch Dateien mit der Endung *.RSS* als JSP- bzw. JSF-Dateien interpretiert werden, zum anderen kann die RSS-Datei auch die Endung *.jsp* tragen, Reader-Programme haben damit im Normalfall keine Probleme.

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

Listing 8.18: Ausschnitt aus der web.XML

In Listing 8.18 wird in der Standard *web.XML* im */conf*-Verzeichnis von Tomcat ein Servletmapping angelegt, das Dateien mit der Endung *.rdf* auf das JSP-Servlet abbildet und somit eine Kompilierung und eine Ausführung der Datei anstößt.

Aufbau einer RSS-Datei

Das zentrale Element jeder RSS-Datei sind immer die so genannten *items*, sprich die einzelnen Meldungen. Jede RSS-Datei beinhaltet immer mindestens eine Meldung. Dabei besteht eine Meldung meist aus den Elementen

- Title
- Description
- Link

Die im Folgenden getroffenen Angaben beziehen sich wie bereits erwähnt zunächst auf die Version 0.91. Die anderen Versionen unterscheiden sich jedoch nicht gravierend in deren Aufbau, meist sind lediglich die Elemente anders angeordnet oder um neue ergänzt worden.

Der *title* ist, wie der Name bereits vermuten lässt, eine kurze Überschrift der Meldung. In Newsreadern werden meist die Titel in einer größeren Schrift und farblich abgehoben dargestellt. *Description* ist ein kurzer Erläuterungstext zur Meldung selbst. In Falle von Nachrichten ist dies meist ein so genannter Teaser-Text, der zur eigentlichen Nach-

richt hinleitet. Über einen *Link* wird auf die eigentliche Webseite verwiesen, auf der die gesamte Meldung in ausführlicher Länge zu finden ist. Der Titel sowie die Beschreibung müssen nicht zwangsläufig mit den Angaben in der eigentlichen Webseite zusammenhängen, so kann die Beschreibung in der RSS-Datei von eventuellen Beschreibungstexten in einer Indexseite im Web durchaus variieren.

```
<item>
  <title>Montag, 5, Januar 2004: Neues Tutorial zu JSF</title>
  <description>
    Auf den Seiten des JSF-Forums ist ein neues Tutorial
    zu JSF zu finden.
  </description>
  <link>http://www.jsf-forum.de</link>
</item>
```

Listing 8.19: Ausschnitt einer RSS-Datei

Neben den Angaben zu einzelnen *Item*-Elementen wie in Listing 8.19 abgebildet sind in einer RSS-Datei noch zusätzlich Angaben zum verwendeten RSS-Format sowie allgemeine Angaben zum Channel selbst zu finden.

```
<?XML version="1.0" encoding="ISO-8859-1" ?>
<RSS version="0.91">
<channel>

<title>JSF-Nachrichten</title>
<link>http://www.jsf-forum.de</link>
<description>
  Die aktuellsten Nachrichten zum Thema JSF
</description>
<language>de-de</language>

<item>
  <title>Final Release</title>
  <description>
    Nach langer Zeit des Wartens liegt jetzt die Final-Version
    der Spezifikation und der Referenzimplementierung vor.
  </description>
  <link>http://www.javasoft.com</link>
</item>

<item>
  <title>
    JavaServer Faces Proposed Final Draft and Beta 1.0 RI Shipped
  </title>
  <description>
    Es liegt eine neue Version der Faces-Spezifikation
    sowie eine Beta der Referenzimplementierung vor.
  </description>
```

```
    <link>http://www.javasoft.com</link>
  </item>

  </channel>

  </RSS>
```

Listing 8.20: Eine vollständige RSS-Datei

Wichtig bei einer RSS-Datei ist die Angabe des Formats. In Listing 8.20 wird dies durch die Angabe von

```
<RSS version="0.91">
```

explizit gesetzt. Des Weiteren kann auch im Channel ein Bild hinterlegt werden, das zusammen mit den jeweiligen Meldungen angezeigt wird. Dies ist meist ein charakterisierendes Bild des Unternehmens oder des Contentanbieters, damit ein gewisser Wiedererkennungseffekt gegeben ist, wenn ein Nutzer eine Liste von Meldungen von diversen Anbietern in seinem Newsreader zu sehen bekommt.

RSS 2.0 weist zudem noch viele weitere zum Teil auch sehr nützliche Tags auf, die jedoch in diesem Rahmen nicht weiter besprochen werden. Für ein einfaches RSS-Newsfeed genügen die in diesem Kapitel behandelten Elemente vollkommen.

Benötigte JSF-Komponenten

Um nach dieser Einführung in RSS mittels JSF einen Newsfeed bereitstellen zu können, müssen spezielle Komponenten bzw. Renderer entwickelt werden, die die Datenbankinhalte einzelner Nutzer nicht mehr in HTML-Form, sondern in XML-Form ausgeben.

Eine einfache Möglichkeit, die gewünschte XML-Ausgabe zu erzielen, ist, zwei benutzerspezifische JSF-Komponenten zu entwickeln. Eine Komponente ist dabei für die Ausgabe eines *items* zuständig, d.h. es soll einer Komponente sowie deren Renderer ein Bean übergeben werden, die daraufhin als Ausgabe einen XML-Strom erzeugen, der genau einen `item`-Tag mit den dazu notwendigen Kindelementen enthält. Dazu ist es im Vorfeld natürlich notwendig, ein entsprechendes Item-Bean zu entwickeln, das die geforderten Eigenschaften aufweist. Dieses kann dann wiederum als Attribut des zu entwickelnden Tags übergeben werden, worauf durch den Renderer daraufhin die geforderte Ausgabe erzeugt wird.

```
<item>
  <title>Montag, 10, Januar 2004: ...</title>
  <description>Blindtext ...</description>
  <link>http://localhost:8080/...</link>
</item>
```

Listing 8.21: Gewünschter XML-Output

Die zweite zu entwickelnde Komponente ist eine Erweiterung der `UIPanel`-Komponente. Diese Komponente kann für eine Ausgabe einer *Collection* verwendet werden, indem über die enthaltenden Werte iteriert wird. Der Renderer jedoch, der von JSF für Listen mitgeliefert wird, erzeugt eine HTML-Tabelle. Da in diesem Fall jedoch überhaupt keine Ausgabe benötigt wird, sondern lediglich die einzelnen `item`-Elemente aneinander gereiht werden müssen, wird an dieser Stelle die Komponente erweitert und ein eigener Renderer implementiert. Das Ergebnis soll somit eine Liste von item-Elementen sein, die ohne zusätzliche Ausgabe nacheinander erscheinen.

Für die Channel-Angaben im RDS-Strom genügt es, wenn ein entsprechendes Managed-Bean bereitgestellt wird, das über `outputText`-Tags die notwendigen Inhalte liefert.

Knackpunkt: Bereitstellen von Inhalten vor dem ersten JSF-Request

Die Schwierigkeit bei der Bereitstellung eines RSS-Channels liegt mit Sicherheit darin, dass die für die Erzeugung der RSS-Datei notwendigen Managed-Beans bereitstehen müssen, *bevor* überhaupt eine JSF-Aktion gestartet wurde. Sprich es wird eine JSF-Seite aufgerufen, ohne dass aufgrund einer Aktion im Vorfeld über einen Datenbankzugriff Beans bereitgestellt werden konnten. Da die Angabe des Benutzers, dessen Einträge angezeigt werden sollen, dynamisch über einen Parameter übergeben werden, können auch nicht standardmäßig die Beans bereitgestellt werden. Es muss somit ein Mechanismus geschaffen werden, der die benötigten Beans vor dem Aufruf der JSF-Seite befüllt.

Mit Hilfe des *FrontController-Designpatterns* wird daher ein Servlet erzeugt, das zunächst einmal die Anfrage nach einer RSS-Datei entgegennimmt. In der Service-Methode können daraufhin basierend auf den Request-Parametern die notwendigen Datenbankzugriffe angestoßen werden. Ebenso kann das Ergebnis der Datenbankabfrage daraufhin in Beans für JSF angelegt werden. Im Anschluss daran leitet das Servlet den Request an die eigentliche JSF-Seite, sprich an die RSS-Datei, weiter.

Mit diesem Lösungsansatz kann erreicht werden, dass Aktionen stattfinden, **bevor** der erste JSF-Request zustande kommt. Das `FrontController`-Entwurfsmuster ist hierfür eine sehr gute Möglichkeit, diese Anforderungen abzubilden. JSF selbst nutzt natürlich ebenfalls das `FrontController`-Entwurfsmuster, indem zunächst einmal alle Requests an das Faces-Servlet geleitet werden, wonach daraufhin die gemäß den Navigationsregeln hinterlegten JSF-Seiten aufgerufen werden.

Vorgehensweise

Zunächst werden die benötigten Modellobjekte zusammengestellt und diese als Java-Klassen implementiert. Danach werden die benötigten Komponenten, Renderer und Tagklassen erstellt sowie der TLD (Tag Library Deskriptor) angepasst. Darauf aufbauend kann dann die eigentliche Faces-Seite in Form einer RSS-Datei aufgebaut und anschließend getestet werden.

Modellobjekte

Für die Darstellung einer RSS-Datei werden folgende Beans benötigt, die zum Teil in Form von Managed-Beans in die Anwendung eingebaut werden:

- `ItemBean`: Ein ItemBean beinhaltet die Informationen für genau eine Meldung. Dazu gehören ein Titel einer Meldung, ein kurzer Erklärungstext (so genannter Teaser) sowie ein Link auf die eigentliche Seite mit dem ausführlichen Text.
- `ItemList`: Da in einer RSS-Datei eine bestimmte Anzahl einzelner ItemBeans vorhanden sein können, werden die jeweiligen Meldungen für eine RSS-Datei gesammelt in einer ArrayList vorgehalten.
- `ChannelBean`: Neben den einzelnen Meldungen weist eine RSS-Datei ebenfalls Informationen genereller Art auf, wie beispielsweise einen allgemeinen Text über den RSS-Channel. Diese Informationen werden über ein ChannelBean bereitgestellt.

Aufgrund der Tatsache, dass in der späteren JSF-Datei ausschließlich auf die Beans `ItemList` und `ChannelBean` zugegriffen wird, muss das `ItemBean` nicht in der Anwendungskonfigurationsdatei als Managed-Bean hinterlegt werden. Es wird lediglich für die Verarbeitung intern benutzt.

```
<managed-bean>
  <managed-bean-name>ItemList</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.ItemList
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Channel</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.WebLog.bean.ChannelBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Listing 8.22: Managed-Beans in der Anwendungskonfigurationsdatei

FrontController Servlet

Wie bereits im Abschnitt zur Problemstellung geschildert, müssen die in der JSF-Seite (bzw. RSS-Seite) verwendeten Managed-Beans im Vorfeld des Seitenaufrufs bereitgestellt werden. Hierzu wird ein Servlet entwickelt, das eine entsprechende Anfrage entgegennimmt, die Requestparameter auswertet und daraufhin das Initialisieren der Managed-Beans anstößt. Im Anschluss daran wird dann auf die eigentliche JSF-Seite verwiesen.

```
package com.edu.jsf.WebLog.srv;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;

import javax.faces.FactoryFinder;
import javax.faces.application.Application;
import javax.faces.application.ApplicationFactory;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.edu.jsf.WebLog.bean.ChannelBean;
import com.edu.jsf.WebLog.bean.ItemBean;
import com.edu.jsf.WebLog.dbbean.BlogEntry;

/**
 * FrontController-Servlet
 */
public class RSSFrontController extends HttpServlet {

  private UserMgr aUserMgr;
  private WebLogMgr aWebLogMgr;
  private ApplicationFactory factory;
  private Application application;

  /**
   * Initialisierung des Servlets
   */
  public void init() throws ServletException {
    aUserMgr = new UserMgr();
    aWebLogMgr = new WebLogMgr();

    factory = (ApplicationFactory) FactoryFinder.getFactory(
      FactoryFinder.APPLICATION_FACTORY);
    application = factory.getApplication();
  }

  /**
   * Implementierung des GET-Verfahrens
   */
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse res)
      throws ServletException, IOException {

    // Abfrage auf einen User
```

```
    Object obj = req.getParameter("user");
    String user = (obj!=null) ? obj.toString() : "";

    // Abfrage auf eine Aktion
    obj = req.getParameter("action");
    String action = (obj!=null) ? obj.toString() : "";

    if ( action.equals("blog") ) {
      bindUserBlog( req, res, user );
    } else if ( !user.equals("") ) {
      bindLatestBlogs( req, res, user );
    } else {
      // ... auf eine Fehlerseite weiterleiten
    } // else

  }

  private void bindLatestBlogs(HttpServletRequest req,
               HttpServletResponse res, String user )
      throws ServletException, IOException {
    String sNewUrl = req.getRequestURL().toString();
    sNewUrl = sNewUrl.substring( 0, sNewUrl.lastIndexOf('/') );
    int userId = Integer.parseInt( user );
    FacesContext context = FacesContext.getCurrentInstance();

    ArrayList alRSSList = new ArrayList();

    // Liste der letzten WebLogeinträge eines Users holen
    ArrayList items = aWebLogMgr.getLatestWebLogsForUser( userId );
    Iterator itr = items.iterator();
    BlogEntry aEntry;
    ItemBean aItem;
    // WebLog-Beans in ItemBean umsetzen
    while ( itr.hasNext() ) {
      aEntry = (BlogEntry)itr.next();
      aItem = new ItemBean();
      aItem.setTitle( aEntry.getCreatedate().toString() );
      if ( aEntry.getTxt().length()>50 )
        aItem.setDescription( aEntry.getTxt().substring(0, 50));
      else
        aItem.setDescription( aEntry.getTxt() );
      aItem.setLink( sNewUrl + "/RSS?action=blog?user=" + user );

      alRSSList.add( aItem );
    } // while

    // Liste der WebLogeinträge im FacesContext ablegen
    ValueBinding binding = null;
    binding = application.createValueBinding("#{ItemList.items}");
    binding.setValue( context, alRSSList );
```

```
      ChannelBean aChannel = new ChannelBean();
      aChannel.setTitle("WebLog-News");
      aChannel.setDescription(
        "Die letzten WebLog-Eintragungen von "
        + aWebLogMgr.getUserName( user ) );
      aChannel.setLink( "http://www.addison-wesley.de" );

      binding = application.createValueBinding( "#{Channel}" );
      binding.setValue( context, aChannel );

      // auf die eigentliche Seite weiterleiten
      RequestDispatcher rd =
        getServletContext().getRequestDispatcher(
        "/news.rdf" );
      rd.forward( req, res );
    }

    private void bindUserBlog( HttpServletRequest req,
        HttpServletResponse res, String user_id )
        throws ServletException, IOException {
      String sNewUrl = req.getRequestURL().toString();
      sNewUrl = sNewUrl.substring( 0, sNewUrl.lastIndexOf('/') );

      FacesContext context = FacesContext.getCurrentInstance();
        aWebLogMgr.initializeWebLog( context,
      aWebLogMgr.getBlogIdForUser(Integer.parseInt(user_id)) );
      res.sendRedirect( sNewUrl + "/show_WebLog.jsp" );
    }
  }
```

Listing 8.23: Das RSS-Frontcontroller-Servlet

In der `init`-Routine des in Listing 8.23 abgebildeten `FrontController`-Servlets werden zunächst nur die notwendigen Initialisierungen vorgenommen. Da das Servlet ausschließlich über Get-Aufrufe angesprochen wird, ist die gesamte Logik in der `doGet`-Routine enthalten. Das Servlet wertet zwei Parameter aus, die durch den Aufruf

```
req.getParamter( Name des Parameters )
```

aus dem Request herausgelesen werden. Dass zwei Parameter ausgelesen werden, ist für den weiteren Programmablauf wichtig, da das FrontController-Servlet für zwei Programmverzweigungen verwendet wird:

Zum einen werden die letzten WebLog-Eintragungen als Liste im Newsreader angezeigt. Die Steuerung, auf welchen Nutzer sich die Liste bezieht, wird durch den Parameter `user` gesteuert. Wird somit in der URL des Aufrufes der Parameter `user` mitgegeben, erhält man als Ergebnis die aufbereitete RSS-Datei für genau einen Nutzer.

Möchte man jedoch über den Link in den einzelnen Meldungen in der RSS-Datei auf einen konkreten WebLog-Eintrag springen, erfolgt der Ablauf hierbei auch über das FrontController-Servlet. Im Gegensatz zur Liste mit den letzten WebLog-Eintragungen im RSS-Format, verweist der Ablauf beim Ansprechen genau eines Blogs auf eine HTML-Seite.

Im Falle, dass die RSS-Datei angefragt wurde, wird in der Methode `bindLatestBlogs` zunächst eine `ArrayList` mit den einzelnen Meldungen durch den Aufruf von

```
ArrayList items = aWebLogMgr.getLatestWebLogsForUser( userId );
```

erzeugt. In der entsprechenden Methode des `WebLogMgr` erfolgt mit Hilfe der Zugriffsklassen von Torque der Datenbankzugriff. Das Ergebnis sind eine Anzahl von Blog-Objekten, die als `ArrayList` zurückgeliefert werden. In der folgenden `while`-Schleife wird aus den `Blog`-Objekten ein `Item`-Objekt erzeugt, das die notwendigen Informationen für die Darstellung eines Eintrages in der RSS-Datei bereithält.

Um die ebenfalls als `ArrayList` vorliegenden *item*-Objekte für JSF verfügbar zu machen, wird diese `ArrayList` über den Mechanismus des *ValueBinding* eingesetzt.

```
ValueBinding binding = null;
binding = application.createValueBinding( "#{ItemList.items}" );
binding.setValue( context, alRSSList );
```

Wichtig ist die genaue Schreibweise `ItemList.items`, da ansonsten das Setzen des Wertes fehlschlägt. Danach stehen die gefüllten Modellobjekte für die weitere Verarbeitung in JSF bereit.

Nachdem alle Vorbereitungen getroffen wurden, wird über

```
RequestDispatcher rd =
  getServletContext().
  getRequestDispatcher( "/news.rdf" );
rd.forward( req, res );
```

auf die eigentliche JSF-Seite, die RSS-Datei, weitergeleitet und dort werden die Inhalte der Managed-Beans, die im Vorfeld bereitgestellt wurden, angezeigt.

Damit das RSS-Frontcontroller-Servlet angesprochen werden kann, ist das Servlet im Deployment Deskriptor *web.XML* bekannt zu machen.

```
...
<servlet>
  <servlet-name>RSS Servlet</servlet-name>
  <servlet-class>
    com.edu.jsf.WebLog.srv.RSSFrontController
  </servlet-class>
  <load-on-startup>0</load-on-startup>
```

```
</servlet>
...
<servlet-mapping>
  <servlet-name>RSS Servlet</servlet-name>
  <url-pattern>/RSS</url-pattern>
</servlet-mapping>
...
```

Listing 8.24: Auszug aus web.XML

Zunächst wird das Servlet mit dem Namen *RSS Servlet* und der Servletklasse angegeben. Das *Servlet Mapping* regelt, dass alle Aufrufe mit dem URL-Muster */RSS* an dieses Servlet und damit an die angegebene Servletklasse weitergeleitet werden.

Als Ergänzung sei noch erwähnt, dass die Initalisierung anstelle eines FrontController-Servlets auch mit Hilfe so genannter *PhaseListener* möglich gewesen wäre. Dabei wäre vor jedem Request eine registrierte Listenerklasse benachrichtigt worden, die auch die notwendigen Initialisierungen vornehmen hätte können. Ein kurzes Beispiel eines PhaseListeners ist in Kapitel 6.10.4 zu finden.

Zwischenstand

Zum jetzigen Entwicklungsstand existieren die benötigten Beans sowie die Geschäftslogik für das Auslesen der Inhalte aus der Datenbank. Ebenso ist durch das FrontController-Servlet eine Möglichkeit geschaffen worden, requestgesteuerte JSF-Seiten direkt aufrufen zu können. Sollte die Ausgabe mittels üblicher HTML-Syntax erfolgen, könnte im nächsten Schritt direkt an die Umsetzung der JSF-Seiten gegangen werden. Da jedoch für die RSS-Datei ein XML-Format verlangt wird, muss zunächst noch mit Hilfe neuer Renderer die XML-Ausgabe vorbereitet werden.

Die UIOutputItem-Komponente

Die Überschrift sorgt vielleicht schon für erste Verwirrung: Es ist doch geplant, aus den in JSF vorhandenen Komponenten nur durch eigene Renderer die XML-Ausgabe zu realisieren, weshalb taucht in der Überschrift dann das Wort Komponente auf?

Der Grund liegt darin, dass für die XML-Ausgabe einer Meldung (eines Items) ein Attribut `item` verwendet wird, über das der Zugriff auf ein `ItemBean` realisiert wird. Die Basis der neuen Komponente ist somit nach wie vor die `UIOuput`-Komponente, erweitert um das Attribut `item`.

```
package com.edu.jsf.WebLog.tag;

import javax.faces.FactoryFinder;
import javax.faces.application.Application;
import javax.faces.application.ApplicationFactory;
import javax.faces.component.UIOutput;
```

```
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.el.ValueBinding;

import com.edu.jsf.WebLog.bean.ItemBean;

/**
 * Komponentenklasse für die
 * Ausgabe eines Items im XML-Format
 */
public class UIOutputItem extends UIOutput {

  private String item = null;

  /**
   * Codierung
   */
  public void encodeEnd(FacesContext context)
      throws java.io.IOException {

    ApplicationFactory factory =
      (ApplicationFactory) (FactoryFinder)
      .getFactory("javax.faces.application.ApplicationFactory")));
    Application application = factory.getApplication();
    ValueBinding binding = application.createValueBinding( item );
    ItemBean item = (ItemBean)binding.getValue( context );

    // Rendering
    ResponseWriter writer = context.getResponseWriter();
    writer.write("<item>\n");
    writer.write("<title>" + item.getTitle() + "</title>\n");
    writer.write("<description>\n");
    writer.write( item.getDescription() +"\n" );
    writer.write("</description>\n");
    writer.write("<link>" + item.getLink() + "</link>\n");
    writer.write("</item>\n");
  }

  /**
   * Setter- und Getter-Methoden
   */
  public String getItem() {
    return item;
  }

  public void setItem(String string) {
    item = string;
  }
}
```

Listing 8.25: Die Komponentenklasse UIOutputItem

Die Komponente `UIOuputItem` erbt sämtliche Funktionalität von `UIOutput`. Da als gewünschtes Ergebnis lediglich textueller Inhalt ausgegeben werden soll, eignet sich die `UIOuput`-Komponente ideal. Das Rendering selbst wird nicht in eine separate Renderer-Klasse ausgelagert, sondern findet in der Komponentenklasse selbst statt. Dies ist in diesem Fall so auch akzeptabel, da die Komponente ausschließlich im Zusammenhang mit der XML-Ausgabe verwendet wird. Ansonsten wäre es vom Architekturdesign eleganter, hierfür eine eigene Rendererklasse anzulegen. Das eigentliche Rendering findet in der Methode `encodeEnd` statt, die die entsprechende Methode der Superklasse damit überschreibt. Es wird dabei auf das ItemBean zurückgegriffen, das über ein item-Attribut übergeben wurde. Als Ausgabe wird die XML-Syntax für das `item`-Tag geliefert.

Natürlich ist zur Verwendung der Komponente ein entsprechendes Tag sowie eine Eintragung im TLD (Tag-Library-Descriptor) notwendig.

```
<tag>
  <name>outputItem</name>
  <tag-class>com.edu.jsf.WebLog.tag.OutputItemTag</tag-class>
  <body-content>none</body-content>
  <attribute>
    <name>item</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
```

Listing 8.26: Definition des Tags im TLD

Im Ausschnitt des TLDs von Listing 8.26 ist nochmals zu erkennen, dass das `outputItem`-Tag ein Attribut `item` aufweist, über das ein `ItemBean`-Objekt mit den notwendigen Angaben der Komponente bzw. dem Renderer übergeben wird.

```
package com.edu.jsf.WebLog.tag;

import javax.faces.component.UIComponent;

import com.sun.faces.taglib.Html_basic.OutputTextTag;

/*
 * Tag-Klasse. Dient zur Ausgabe eines Items
 * im XML-Format.
 */
public class OutputItemTag extends OutputTextTag {

  private String item;

  /*
```

```
     * liefert den Renderer zurück
     */
    public String getRendererType() {
      return null;
    }

    /*
     * liefert den Komponententyp zurück
     */
    public String getComponentType() {
      return "OutputItem";
    }

    /*
     * setzt die Properties in die Komponente
     */
    protected void setProperties(UIComponent component) {
      super.setProperties(component);
      UIOutputItem outputcomponent = (UIOutputItem)component;
      if ( item!=null && outputcomponent.getItem()==null )
        outputcomponent.setItem( item );
    }

    /*
    * Setter- und Getter-Methoden
     */
    public String getItem() {
      return item;
    }

    public void setItem(String string) {
      item = string;
    }
  }
```

Listing 8.27: Die Taghandler-Klasse für die UIOutputItem-Komponente

In der `Taghandler`-Klasse aus Listing 8.27 liefert die Methode `getRendererType` als Rückgabewert `null` zurück. Dies bedeutet, dass kein separater Renderer hinterlegt ist, sondern das Rendering in der Komponente vorgenommen wird. Der Rückgabewert der Methode `getComponentType` ist ein Bezeichner, der in der Anwendungskonfigurationsdatei hinterlegt sein muss, da an dieser Stelle die Zuordnung zur Komponentenklasse vorgenommen wird.

```
<component>
  <component-type>OutputItem</component-type>
  <component-class>
```

```
    com.edu.jsf.WebLog.tag.UIOutputItem
  </component-class>
</component>
```

Listing 8.28: Definition der Komponente in faces-config.XML

Ausgabe der item-Liste

Die zweite Komponente, die für die Ausgabe von XML angepasst werden muss, ist eine Form der `UIPanel`-Komponente. Anstatt jedoch eine neue Komponente zu kreieren, genügt es in diesem Fall, einen neuen Renderer zu definieren, das Verhalten der `UIPanel`-Komponente, die in dieser Form in JSF standardmäßig enthalten ist, muss nicht verändert oder angepasst werden. Damit ein neuer Renderer zugewiesen werden kann, wird ein entsprechendes Tag samt `Taghandler`-Klasse definiert.

```
<tag>
  <name>itemList</name>
  <tag-class>com.edu.jsf.WebLog.tag.ItemListTag</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>var</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <type>java.lang.String</type>
  </attribute>
  <attribute>
    <name>value</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <type>java.lang.String</type>
  </attribute>
</tag>
```

Listing 8.29: Tag-Definition

Um nochmals die Vorgehensweise zu erläutern: Es wird für die Ausgabe der einzelnen Einträge in Form von `item`-Objekten eine Komponente benötigt, die lediglich die einzelnen Werte ausgibt. Der Renderer, der dieser Komponente zugeordnet ist, sollte somit keine weitere Formatierung in Form von Tabellen mit einbauen. Die `UIPanel`-Komponente ist hierfür die geeignete (Basis-)Komponente, es ist somit lediglich ein eigener Renderer zu entwickeln, der sämtliche HTML-Tag-Ausgaben unterbindet.

```
package com.edu.jsf.WebLog.tag;

import javax.faces.component.UIComponent;

import com.sun.faces.taglib.Html_basic.DataTableTag;
```

```
/*
 * Tag-Klasse
 */
public class ItemListTag extends DataTableTag {

  /*
   * liefert den Renderer zurück
   */
  public String getRendererType() {
    return "ItemList";
  }

  /*
   * setzt die Properties in die Komponente
   */
  protected void setProperties(UIComponent component) {
    super.setProperties(component);
  }
}
```

Listing 8.30: Die Taghandler-Klasse

In der Taghandler-Klasse (vgl. Listing 8.30) wird von der JSF-Taghandler-Klasse `DataTableTag` geerbt und nur die notwendigen Methoden überschrieben. So liefert die Methode `getRendererType` einen Bezeichner für den neu erstellten Renderer zurück. Dieser Bezeichner ist wiederum in der Anwendungskonfigurationsdatei zu finden. Wichtig ist die Methode `getComponentType`. Hier wird der Standardbezeichner *DataTable* zurückgeliefert, der bewirkt, dass keine eigene Komponentenklasse angezogen, sondern die übliche Komponente von JSF verwendet wird. Im Gegensatz zu der neu entwickelten `UIOutputItem`-Komponente wird in der `UIPanel`-Komponente keinerlei neue Funktionalität benötigt, so dass ein neuer Renderer für die XML-Ausgabe vollkommen genügt.

```
<renderer>
  <component-family>javax.faces.Data</component-family>
  <renderer-type>ItemList</renderer-type>
  <renderer-class>
    com.edu.jsf.WebLog.tag.ItemListRenderer
  </renderer-class>
</renderer>
```

Listing 8.31: Hinterlegung des Renderers

In Listing 8.31 ist nochmals der Ausschnitt aus der Anwendungskonfigurationsdatei *faces-config.XML* zu sehen, in dem die Renderer-Klasse zu dem Bezeichner, der in der Taghandler-Klasse in der Methode `getRendererType` zurückgeliefert wird, definiert wird.

Das zentrale Element stellt jedoch die Renderer-Klasse selbst dar. Prinzipiell würde der Standard-Renderer von JSF passen, würde anstelle der HTML-Ausgaben einfach keine Ausgabe erfolgen. Die neue Renderer-Klasse ist somit dahingehend anzupassen, dass sämtliche HTML-Ausgaben einfach entfernt werden. Da in der RSS-XML-Datei keinerlei zusätzliche Elemente angegeben werden, sondern ausschließlich eine Liste von `item`-Tags aneinander gereiht werden sollen, ist es die Aufgabe des Renderers, keinerlei Ausgaben zu erzeugen, sondern lediglich die einzelnen item-Elemente aneinander zu reihen.

```
package com.edu.jsf.WebLog.tag;

import java.io.IOException;
import java.util.Iterator;

import javax.faces.component.UIColumn;
import javax.faces.component.UIComponent;
import javax.faces.component.UIData;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

import com.sun.faces.renderkit.Html_basic.TableRenderer;
import com.sun.faces.util.Util;

/**
 * Rendererklasse für die XML-Listausgabe
 */
public class ItemListRenderer extends TableRenderer {

  public void encodeBegin(FacesContext context,
      UIComponent component)
      throws IOException {
    if (context == null || component == null)
      throw new NullPointerException(
    Util.getExceptionMessage(
      "com.sun.faces.NULL_PARAMETERS_ERROR"));
  }

  public void encodeChildren(FacesContext context,
      UIComponent component)
      throws IOException {
    if (context == null || component == null)
      throw new NullPointerException(
    Util.getExceptionMessage(
      "com.sun.faces.NULL_PARAMETERS_ERROR"));

    if (component.isRendered()) {
      UIData data = (UIData) component;
      ResponseWriter writer = context.getResponseWriter();
      Iterator kids = null;
      Iterator grandkids = null;
```

```
        kids = data.getChildren().iterator();
        int rowIndex = data.getFirst() - 1;
        int rows = data.getRows();
        int processed = 0;
        while (rows <= 0 || ++processed <= rows) {
          data.setRowIndex(++rowIndex);
          if (!data.isRowAvailable())
            break;
          kids = data.getChildren().iterator();
          while (kids.hasNext()) {
            UIComponent kid = (UIComponent) kids.next();
            if (kid instanceof UIColumn) {
              UIColumn column = (UIColumn) kid;
              grandkids = column.getChildren().iterator();
              while (grandkids.hasNext())
                encodeRecursive( context,
                  (UIComponent) grandkids.next());
            } // if
          } // while
        } // while
      } // if
    }

    public void encodeEnd(FacesContext context,
        UIComponent component)
        throws IOException {
      if (context == null || component == null)
        throw new NullPointerException();
    }
  }
```

Listing 8.32: Renderer-Klasse für die Panel-Komponente

Die in Listing 8.32 aufgelistete Renderer-Klasse beinhaltet keinerlei Ausgaben, weder HTML noch XML. Dies ist auch durchaus in dieser Form gewollt, soll doch ausschließlich eine Iteration über die Kindelemente erfolgen, die durch den Renderer der `UIOutputItem`-Komponente mit den notwendigen XML-Tags gerendet werden.

Aufbau der RSS-Datei

Nachdem sämtliche Komponenten, Renderer und sonstige Programmfunktionalitäten bereitgestellt sind, kann der eigentliche Kern der Aufgabe, die RSS-Datei, aufgebaut werden. Mit Hilfe der neu definierten Tags kann sehr schnell die RSS-Datei umgesetzt werden.

```
<?XML version="1.0" encoding="ISO-8859-1" ?>
<RSS version="0.91">

<%@ taglib uri="http://java.sun.com/jsf/HTML" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

```
<%@ taglib uri="/WEB-INF/WebLog.tld" prefix="cst" %>

<f:view>

  <channel>
    <title><h:outputText value="#{Channel.title}" /></title>
    <link><h:outputText value="#{Channel.link}" /></link>
    <description>
      <h:outputText value="#{Channel.description}" />
    </description>

    <cst:itemList var="singleitem" value="#{ItemList.items}">
      <h:column>
        <cst:outputItem item="#{singleitem}" />
      </h:column>
    </cst:itemList>
  </channel>

</f:view>

</RSS>
```

Listing 8.33: Die RSS-/JSF-Datei

Am Anfang der Datei werden – wie bei JSF-basierten Seiten generell – die notwendigen Tag-Bibliotheken mit eingebunden. Wichtig ist in diesem Zusammenhang auch, dass die eigens erstellte Bibliothek mit eingebunden wird.

An Stelle der bisher gewohnten HTML-Tags, die den Beginn einer HTML-Seite aufzeigen, erfolgen im Anschluss die notwendigen XML-Angaben für eine XML-Datei. Für die Ausgabe der einzelnen Channel-Angaben werden die in JSF vorhandenen `outputText`-Tags verwendet, die auf ein Modellobjekt `Channel` zurückgreifen. Dieses soll im Vorfeld durch das FrontController-Servlet bereitgestellt werden.

Zieleinlauf

Nach dieser zugegebenermaßen relativ aufwändigen Entwicklungsarbeit können jetzt alle Komponenten in einem zusammenhängenden Ablauf getestet werden. Eventuelle Fehlermeldungen sind zumeist relativ aussagekräftig, so dass mögliche Tippfehler oder sonstige Konfigurationsfehler sehr schnell lokalisiert und behoben werden können.

Sollte nicht sogleich die RSS-Datei in einem Newsreader geöffnet werden, kann mit einem Browser ebenfalls die XML-Datei angezeigt werden. Im Internet Explorer gibt es zum Teil Probleme bei der Darstellung, in anderen Browsern wie z.B. Mozilla funktioniert die XML-Ansicht ohne Probleme. Wichtig ist der Aufruf des FrontController-Servlets über die Url

http://localhost:8080/JSFWebLog/faces/RSS?user=1000

Die Angabe des Parameters `user=1000` ist zwingend notwendig, da hierüber das Blog gewählt wird, für das die letzten Eintragungen angezeigt werden sollen. Gegebenenfalls muss in der Datenbank nach einer vorhandenen Id gesucht werden.

Um die einzelnen News auch im RSS-Newsreader betrachten zu können, muss unter HotSheet ein neuer Channel mit obiger Adresse eingegeben werden. Nach einem Aktualisieren der einzelnen Channels sollten dann auch die neuen Einträge des WebLogs der ausgewählten Person auftauchen.

8.13 Fazit

Der praktische Einsatz von JavaServer Faces war das Ziel dieses Kapitels. Es wurde in einem umfangreichen Beispiel gezeigt, wie in einer konkreten Webanwendung JavaServer Faces eingesetzt werden kann. Es wurde auf Fallstricke hingewiesen (z. B. Vor-Initialisieren von Managed-Beans) und wir haben Lösungsansätze dargelegt.

Zudem wurde ein Beispiel gezeigt, wie mit Hilfe von Renderern ein anderes Ausgabeformat erzeugt werden kann. Im RSS-Newsreader wurde eine XML-Ausgabe erzeugt anstelle der »üblichen« HTML-Ausgabe.

Außerdem wurde zur Anbindung einer Datenbank das Open-Source-Werkzeug Torque vorgestellt, mit dessen Hilfe sehr elegant und mit relativ geringem Aufwand eine relationale Datenbank angesprochen werden kann.

9 JSF erweitern und anpassen

Kapitelziel

Dieses Kapitel könnte man auch gut mit »JavaServer Faces für Fortgeschrittene« betiteln. Es geht um Möglichkeiten, JavaServer Faces in verschiedenen Bereichen zu erweitern und an eigene Anforderungen anzupassen.

Zunächst können eigene, benutzerdefinierte Komponenten mit eingebracht sowie eigene Renderer sowohl mit vorhandenen als auch mit eigenen Komponenten verwendet werden. Zur Überprüfung von Eingabedaten können Sie neben den Standardvalidatoren benutzerdefinierte Validatoren erzeugen und einsetzen. Dies wird anhand eines Beispiels zur Überprüfung von Kreditkartennummern demonstriert.

Aber auch im Bereich der Datenkonvertierung können Sie eigene Konverter in JSF einbauen. Mittels der in JSF vorhandenen Konverter werden meist schon die häufigsten Umwandlungen abgedeckt, es kann aber durchaus die Notwendigkeit gegeben sein, an dieser Stelle eigene Konverter zu erstellen.

9.1 Erstellen eigener UI-Komponenten

In den letzten Kapiteln haben Sie gelernt, wie Sie mit Hilfe von JavaServer Faces eigene Anwendungen schnell und effektiv mit den vorhandenen Komponenten erstellen können. JavaServer Faces ist jedoch mehr als ein Basisframework, es ist zugleich ein Baukasten zur Erstellung moderner und performanter Webanwendungen. Obwohl bereits eine große Zahl an Komponenten standardmässig in JSF vorhanden ist, kommen Sie sicherlich einmal in die Lage, eigene benutzerspezifische Komponenten erstellen zu müssen.

Die Architektur von JavaServer Faces ist dabei so ausgelegt, dass eine Erweiterung von Komponenten jederzeit möglich ist. Das Konzept sieht es vor, dass neue Komponenten erstellt und in eine Anwendung problemlos integriert werden können. Eine Erweiterung der Komponenten gehört dem Rollenverständnis von JSF nach zu den Aufgaben der Komponenten- oder Basisentwickler. Sie entwickeln eine neue Komponente mit den gewünschten Funktionalitäten und der gewünschten Darstellung. Die fertigen Komponenten können wiederum von Anwendungsentwicklern oder auch Webdesignern in JSF-Seiten verwendet werden.

Aber nicht nur die Entwicklung neuer Komponenten ist möglich, es kann genauso gut lediglich ein neuer Renderer für eine bereits bestehende Komponente entwickelt werden. Durch den modularen Aufbau von JSF ist es jederzeit möglich, an eine bestehende vorhandene Komponente weitere Renderer anzuhängen, ohne die Funktionalität der Komponente selbst neu entwickeln zu müssen.

Vor der Entscheidung, eine benutzerspezifische Komponente zu erstellen, sollte jedoch zunächst einmal gründlich überprüft werden, ob das gewünschte Verhalten nicht auch mit den Standard-Komponenten erreicht werden kann. Liegt das Anliegen hauptsächlich darin, ein neues Erscheinungsbild zu erzeugen, ist keine komplette Neuentwicklung einer Komponente notwendig, es genügt, einen speziellen Renderer dafür zu entwickeln. Auch wenn es darum geht, Werte in einem bestimmten Format aus der Komponente zurückgeliefert zu bekommen, ist dies eher eine Aufgabe für benutzerspezifische Konverter anstatt für eine komplett neue Komponente. Wenn jedoch ein komplett neuartiges Verhalten oder eine komplett neuartige Funktionsweise einer Komponente benötigt wird, ist die Entscheidung, eine benutzerspezifische Komponente zu entwickeln, genau richtig.

Bei der Entwicklung benutzerspezifischer Komponenten gibt es verständlicherweise einige wichtige Punkte, die beachtet werden müssen. So haben Sie im Kapitel über Datenkonvertierung bereits gelesen, dass eine ständige Datenkonvertierung zwischen den beiden Sichten – der Modellsicht und der Präsentationssicht – stattfindet. Aufgrund der Tatsache, dass sämtliche Daten in einer HTML-Oberfläche lediglich in Textform vorliegen, im zugrunde liegenden Modell jedoch als native Datentypen bereitgehalten werden, müssen Sie für die Umwandlung eine Komponente die notwendigen Methoden bereitstellen. So muss zum einen für die Darstellung, also für die Wandlung aus dem Modell in die Präsentation, eine entsprechende Codierung vorgenommen werden, genauso wie für den umgekehrten Weg, wenn Daten aus der Präsentation in die Komponente bzw. in das Modell zurückgeschrieben werden.

Dafür müssen beim Einsatz eigener UI-Komponenten durch diese mindestens zwei Operationen bereitgestellt werden:

- Codierung: Die aktuellen Werte der Komponente müssen in die passende Darstellungsform gewandelt werden.
- Decodierung: Setzt die Werte eines Requests in die Komponente.

Um diese zwei Operationen durchzuführen, gibt es in JSF zwei Programmiermodelle, zwischen denen sich ein Entwickler entscheiden kann:

- Direkte Implementierung: Die Komponentenklasse selbst ist für die Codierung und Decodierung zuständig.
- Implementierung in einem Renderer (*Delegated Implementation*): Die Rendererklasse einer Komponente übernimmt die Aufgabe der Codierung und Decodierung.

Neben der Codierung und Decodierung ist das Thema der Zustandsspeicherung (State Saving) ebenfalls sehr wichtig. Da das im Web eingesetzte Protokoll HTTP (Hypertext Transfer Protocol) verbindungslos ist, müssen zwischen einzelnen Requests die Zustände von Komponenten gespeichert werden. Nachdem eine View gerendert wurde, wird diese im FacesContext abgelegt. Bei einem erneuten Request wird diese View aus dem FacesContext heraus wiederhergestellt. Um eine Komponente (bzw. den Zustand einer Komponente) zu speichern, müssen daher bei der Entwicklung einer benutzerdefinierten Komponente einige Aspekte berücksichtigt werden.

Die Entwicklung einer benutzerspezifischen UI-Komponente vollzieht sich in mehreren Arbeitsschritten. In den folgenden Abschnitten werden Sie dazu exemplarisch eine neue Komponente erstellen und an diesem Beispiel jeden Arbeits- und Entwicklungsschritt näher erläutert bekommen. Insgesamt werden in den Beispielen zwei Komponenten entwickelt, wobei eine davon mehrere Ausbaustufen erfährt. Dies hat den Vorteil, dass Sie genau erkennen können, wie das Konzept von benutzerdefinierten Komponenten erdacht ist und wie Sie damit schnell und zielgerichtet Ihre eigenen Anforderungen umsetzen können. Durch den schrittweisen Ausbau einer Beispielkomponente wird zudem deutlich, wie Sie auch bereits bestehende Komponenten ohne Probleme um weitere Funktionalitäten und Darstellungsformen elegant ergänzen können.

9.1.1 Die Prozentanzeige-Komponente

Als erstes einfaches Beispiel für eine benutzerdefinierte Komponente wird eine Komponente zur Prozentanzeige verwendet. Diese stellt anhand eines übergebenen prozentualen Werts diesen in Form eines Balkens dar. Somit wird eine textuelle bzw. numerische Information graphisch aufbereitet, was die Benutzerfreundlichkeit von Webanwendungen deutlich erhöht. Prozentanzeige-Komponenten werden recht zahlreich im Internet eingesetzt. Ein Beispiel ist im Bereich der Mailboxen von *http://www.web.de* zu finden. Dabei wird der verbrauchte Speicherplatz für Mails prozentual dargestellt. In welchen Bereichen eine Komponente später eingesetzt werden kann, ist zum Zeitpunkt des Entwurfs einer Komponente zunächst einmal zweitrangig. Sobald eine Komponente mit ihrer Funktionalität und ihrer Darstellung fertig gestellt ist, kann sie in jedem Kontext eingesetzt werden, ohne programmiertechnische Entwicklungen dabei vornehmen zu müssen. Dies ist der große Vorteil von Komponenten – write once, run anywhere.

Diese Komponente eignet sich sehr gut als erste benutzerdefinierte Komponente, da sie den Vorteil hat, lediglich Daten darzustellen. Es muss somit eine Konvertierung und Überführung der Eingabedaten in ein Modellobjekt noch nicht berücksichtigt werden. Die Komponente selbst verwendet für die Darstellung eine spezielle Rendererklasse. Darin ist der Quellcode für die eigentliche Darstellung, also die Umsetzung in HTML, gekapselt. Dadurch ist es möglich, bei Bedarf einfach einen anderen Renderer

zu entwickeln, ohne die Funktionalität der Komponente selbst neu implementieren zu müssen (auch wenn gerade bei dieser Komponente die Funktionalität nicht besonders groß ist).

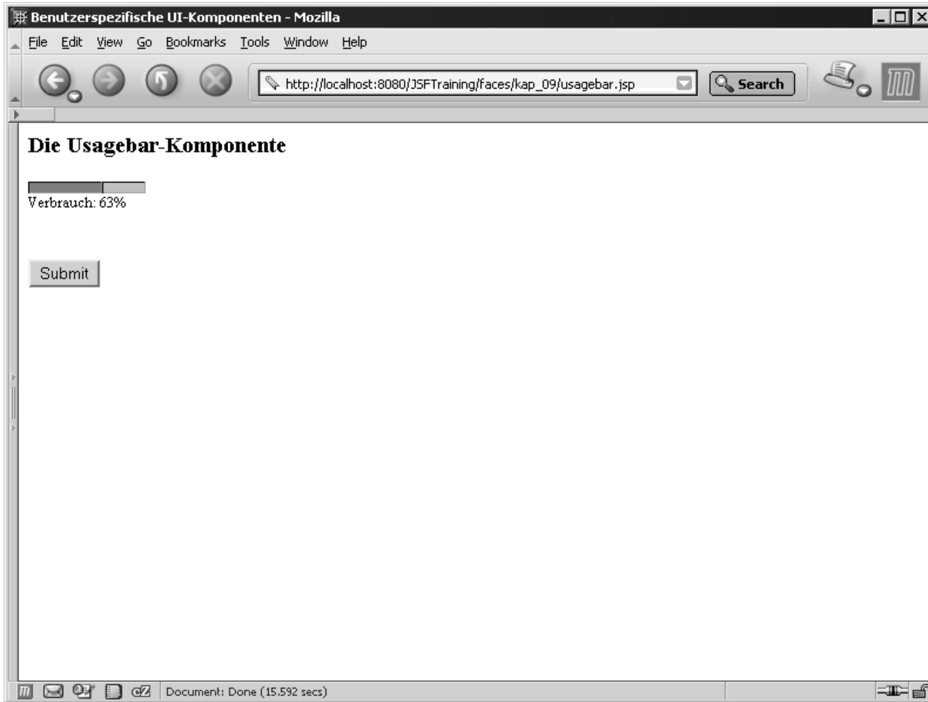


Abbildung 9.1: Die Prozentanzeige-Komponente in Aktion

Interessant ist die Realisierung der Komponente. Obwohl es der erste Eindruck vermitteln könnte, ist die Darstellung keine Graphik, sondern wird durch Tabellen und Hintergrundbilder erzeugt. Es werden zwei Tabellenzellen mit den jeweiligen Hintergründen so in der Breite angeordnet, dass die Balken den prozentualen Werten entsprechen. Da in HTML eine Breitenangabe einer Zelle ohne Inhalt nicht immer korrekt dargestellt wird, wird in der Zelle ein transparentes Bild auf dieselbe Breite wie die gewünschte Zellenbreite selbst gesetzt. Damit erreicht man das erwünschte Ergebnis, dass nämlich die Zellenbreite, und damit die Breite des roten bzw. grünen Balkens, eine berechnete Breite annimmt.

Die Realisierung der Komponente selbst ist im Vergleich zu der Kreditkartenkomponente, die in Abschnitt 9.1.2 beschrieben wird, ein großes Stück einfacher, da die Prozentanzeige-Komponente eben ausschließlich Ausgabefunktionalität besitzt. Es müssen somit keinerlei Werte nach einer Anzeige aus dem Request ausgelesen, überprüft und in ein Modellobjekt übernommen werden.

Vorgehensweise

Für die Entwicklung einer benutzerspezifischen Komponente werden folgende Entwicklungsschritte durchlaufen:

1. Schreiben einer Taghandler-Klasse, die dafür verantwortlich ist, den dazugehörigen Renderer zurückzuliefern sowie den Typ der Komponente zu bestimmen
2. Erzeugen eines TLDs (Tag Library Deskriptors), der die Beschreibung des verwendeten Tags enthält
3. Entwicklung der Komponentenklasse
4. Implementieren der Renderer-Klasse. Diese bestimmt das letztendliche Aussehen der Komponente im Browser.
5. Einbinden der neuen Taglib in eine JSF-Seite sowie Verwendung und Test der Komponente
6. Erweiterung um die Funktionalität der Zustandsspeicherung

Diese Vorgehensweise gilt zunächst einmal für die Prozentanzeige-Komponente. Sie werden in späteren Beispielen sehen, dass bei komplexeren Komponenten weitere Arbeitsschritte hinzukommen. So ist das Thema des Zurückschreibens von Werten in das Modellobjekt sowie das Thema Eventverarbeitung in dieser Komponente noch ohne Berücksichtigung.

Die Taghandler-Klasse

Die Taghandler-Klasse selbst ist »erster Ansprechpartner«, sobald ein benutzerspezifisches Tag in eine JSF-Seite eingebunden wurde. So regelt die Taghandler-Klasse, von welchem Typ ein Tag ist sowie welcher Renderer zuständig ist – vorausgesetzt, es gibt dafür überhaupt einen. Wie Sie in späteren Beispielen noch sehen werden, ist die Angabe eines separaten Renderers nicht immer notwendig, da ein Rendering genauso gut in der Komponente selbst erfolgen kann. Bei der Prozentanzeige-Komponente jedoch wird eine separate Renderer-Klasse verwendet. Zusätzlich werden in der Tag-Klasse die notwendigen Attribute gesetzt, die über die JSF-Seite in den Attributen des Tags angegeben wurden. Dafür stehen übliche setter- und getter-Methoden bereit.

```
package com.edu.jsf.bsp.tag;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentTag;

/**
 * Tag-Klasse für die UsageBar-Komponente.
 * Diese zeigt einen Verbrauchsbalken in Abhängigkeit
 * der übergebenen Prozentzahl.
 */
```

```
public class UsageBarTag extends UIComponentTag {

  private int inuse;

  /**
   * liefert den Renderer zurück
   */
  public String getRendererType() {
    return "UsageBar";
  }

  /**
   * liefert den Komponententyp zurück.
   * Eine entsprechende Eintragung dazu ist ind
   * der faces-config.xml zu finden.
   */
  public String getComponentType() {
    return "UsageBar";
  }

  /*
   * gibt alle Ressourcen frei
   */
  public void release() {
    super.release();
    inuse = 0;
  }

  /**
   * Getter- und Setter-Methode für die
   * Eigenschaft 'inuse'
   */
  public int getInuse() {
    return inuse;
  }

  public void setInuse(int i) {
    inuse = i;
  }

  /**
   * setzt die Properties in die Komponente
   */
  protected void setProperties(UIComponent component) {
    super.setProperties(component);
    UIUsageBar cmp = (UIUsageBar) component;
    cmp.setInuse( inuse );
  }
}
```

Listing 9.1: Taghandler-Klasse für die Prozentanzeige-Komponente

In Listing 9.1 ist die Taghandler-Klasse abgebildet. Die Klasse `UsageBarTag` ist von der Klasse `javax.faces.webapp.UIComponentTag` abgeleitet. Die Klasse `UIComponentTag` ist die Basisklasse für alle Tag-Klassen, die in Zusammenhang mit den UI-Komponenten stehen. Entsprechend dem Standard-Taglib-Verfahren existiert eine Klasse `UIComponent BodyTag`, die dann einzusetzen ist, wenn ein Tag den Body-Content, also einen Wert zwischen dem Start- und dem Endetag, berücksichtigen muss.

Wichtig ist die Methode `getRendererType`. Diese liefert einen Bezeichner zurück, der die Verbindung zur entsprechenden Renderer-Klasse herstellt. Wenn in dieser Methode ein konkreter Wert zurückgeliefert wird, besagt dies, dass das Rendering der Komponente nicht in der Komponente selbst, sondern durch eine separate Rendererklasse übernommen wird. Welche Klasse für das Rendering mit dem Bezeichner `UsageBar` zum Einsatz kommt, wird in der Anwendungskonfigurationsdatei geregelt. Darauf wird jedoch an anderer Stelle nochmals näher eingegangen. Neben dem Rendering wird durch die Methode `getComponentType` geregelt, welche Komponente zum Einsatz kommt. Auch an dieser Stelle wird ein Bezeichner zurückgeliefert, dessen Verbindung zur Komponentenklasse in der Anwendungskonfigurationsdatei hinterlegt ist.

Die Komponente hat zudem eine Eigenschaft `inuse`, die als Ganzzahl den Wert für die Verbrauchsanzeige beinhaltet (Wertebereich von 0 bis 100). Wichtig ist, dass in der Methode `setProperties` die Werte aus der Tagklasse in die Werte der Komponente übergeben werden. Es führt zunächst einmal zu keinem Fehler, wenn diese Methode nicht vollständig implementiert wird, jedoch stehen in der Komponentenklasse dann die notwendigen Werte nicht zur Verfügung, was zu unerwünschten Laufzeitfehlern führen kann. Auch ist der Aufruf der Superklasse wichtig, da unter Umständen dort weitere Eigenschaften in die Komponente gesetzt werden.

Empfohlen wird zudem die Verwendung einer `release`-Methode. In dieser werden alle Ressourcen freigegeben, die eventuell bei Verwendung der Klasse angezogen wurden. Objektreferenzen können hier explizit auf `null` gesetzt werden. Die im Beispiel angedeutete Anweisung, `inuse` auf `0` zu setzen, ist an sich sinnlos, soll aber andeuten, dass sämtliche Ressourcen der Klasse an dieser Stelle freigegeben werden sollen.

Der Tag-Library-Deskriptor

Sämtliche Tags, die in den JSF-Seiten verwenden werden, müssen in einer Xml-Beschreibungsdatei, dem Tag-Library-Deskriptor (kurz TLD) hinterlegt sein. Somit ist auch für die benutzerspezifische Komponente ein entsprechender Eintrag in einem TLD vorzunehmen. Da es verständlicherweise nicht sinnvoll ist, die vorhandenen JSF-Standard-TLDs zu erweitern, da bei einem Einspielen einer neuen Version diese Änderungen verloren gehen würden, wird eine neue Datei zur Speicherung der eigenen Custom-Tags angelegt. Ein TLD beinhaltet neben der Beschreibung der Tag-Bibliothek

selbst Informationen über jedes einzelne Tag der Bibliothek. Diese Beschreibung verwendet der Web-Container, um die in den JSF-Seiten verwendeten Tags zu validieren und gegebenenfalls einen Fehler zu erzeugen.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"HTTP://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>Custom Tag Library for JSF</short-name>
  <uri>HTTP://java.sun.com/jsf/HTML</uri>
  <description>
    Further tags for JSF
  </description>

  <tag>
    <name>usageBar</name>
    <tag-class>com.edu.jsf.bsp.tag.UsageBarTag</tag-class>
    <body-content>none</body-content>
    <attribute>
      <name>inuse</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Listing 9.2: Der erweiterte Tag-Library-Deskriptor

Wie in Listing 9.2 abgebildet, wird ein Tag mit der Bezeichnung `usagebar definiert`. Mit dieser Bezeichnung kann es in einer JSF-Seite angesprochen werden. Als Tag-Klasse ist die Klasse `com.edu.jsf.tag.UsageBarTag` angegeben, die im vorherigen Abschnitt entwickelt wurde. Der Klassenname ist immer vollqualifizierend, d.h. mit kompletten Package-Angaben, zu hinterlegen. Wichtig bei der Definition des Prozentanzeige-Tags ist, dass das Attribut `inuse` ein zwingendes Attribut ist (`required` ist `true`). Ansonsten kann es zu Fehlern in der Komponente selbst kommen. Weitere Attribute könnten an dieser Stelle bei Bedarf hinzugenommen werden. Für das Beispiel wird lediglich ein Attribut verwendet. Achten Sie beim Anlegen des TLDs auf die Groß- und Kleinschreibung. Basierend auf den Angaben im Deskriptor wird zur Laufzeit eine Methode `getInuse` in der Tag-Klasse gesucht. Haben Sie jedoch in der Tagklasse eine Variable `inUse` (großes U) angelegt, so existiert dafür eine getter-Methode `getInUse()`. Dies führt zu einem Programmfehler, da sich die Methodennamen unterscheiden und das Framework somit nicht die passende Methode ermitteln kann.

Das Tag `body-content` gibt an, ob zwischen einem Beginn- und einem Endetag Angaben stehen dürfen. Da das Usebar-Tag jedoch von der Klasse `UIComponentTag` erbt, wird an dieser Stelle das Vorhandensein eines `body`-Elements verneint.

Entwicklung der Komponentenklasse

Eine Komponentenklasse ist für das Verhalten und den Zustand einer UI-Komponente verantwortlich. In einer Komponentenklasse ist somit die Logik einer Komponente enthalten. In der Komponentenklasse sind die Eingabedaten gespeichert sowie einzelne Bezeichner. Zum Verständis nochmals kurz ein Rückblick auf die Verarbeitung von Eingabedaten in JSF:

Die Daten liegen immer in zwei Sichten vor, in der Modellsicht und der Präsentationssicht. Die Komponentenklasse speichert daher vorläufig die Daten, bis diese korrekt in das Modell geschrieben wurden. Erst danach werden die Daten in der Präsentationsschicht wieder gelöscht.

Im Einzelnen ist eine Komponentenklasse für folgende Aufgaben zuständig:

- Codierung
- Decodierung
- Zustandsspeicherung der Komponente
- Validierung
- Modellupdate
- Eventverarbeitung

Bei der Codierung müssen lokale Daten, also Daten, die in der Komponente selbst vorhanden sind, umgewandelt werden in die Präsentationsicht, in Normalfall in eine entsprechende Markup-Syntax. Bei der Decodierung dagegen werden die Werte aus den Requestparametern extrahiert und wiederum in den lokalen Variablen abgespeichert. Nach einer erfolgreichen Validierung auf den (neuen) lokalen Variablen wird ein Modellupdate durchgeführt, bei dem die lokalen Werte in die entsprechenden Modellobjekte überführt werden. Gegebenenfalls werden dabei Events ausgelöst, wie z. B. ein *ValueChangeEvent* bei einer Werteänderung.

Wenn eine Codierung bzw. eine Decodierung nicht in der Komponente selbst stattfinden soll, kommen hierbei die Renderer-Klassen zum Einsatz. Damit wird erreicht, dass die Komponente die Funktionalität unabhängig von der Darstellung kapselt. In diesem Falle werden sämtliche Aufgaben nicht in der Komponentenklasse, sondern in der Renderer-Klasse abgearbeitet.

Bei der Speicherung des Zustandes einer Komponente kann der aktuelle Zustand für den Fall gespeichert werden, dass mehrere Requests stattfinden. So kann durch die Komponente der Zustand gespeichert und für einen späteren erneuten Zugriff wieder erzeugt werden.

Da auch im aktuellen Beispiel die Komponentenklasse nur die (spärlich vorhandene) Funktionalität kapselt, ist die Klasse daher relativ unspektakulär. Auch ein Modellupdate ist – wie Sie bereits gesehen haben – für eine reine Anzeigekomponente nicht notwendig. Aus diesem Grund beinhaltet die Komponentenklasse lediglich die Eigenschaft für den Wert der Verbrauchsanzeige sowie die notwendigen getter- und setter-Methoden.

```
package com.edu.jsf.bsp.tag;

import javax.faces.component.UIComponentBase;

/*
 * Komponentenklasse für die Usage-Komponente
 */
public class UIUsageBar extends UIComponentBase {

  private int inuse;

  /*
   * liefert den Wert der Eigenschaft 'inuse' zurück
   */
  public int getInuse() {
    return inuse;
  }

  /*
   * setzt einen neuen Wert für 'inuse'
   */
  public void setInuse(int i) {
    inuse = i;
  }

  /*
   * liefert einen Bezeichner für die
   * Komponentenfamilie zurück
   */
  public String getFamily() {
    return( "Graph" );
  }
}
```

Listing 9.3: Komponentenklasse für die Prozentanzeige-Komponente

Eine UI-Komponentenklasse ist immer von der Klasse `javax.faces.component .UIComponentBase` abgeleitet. Diese Klasse stellt bereits Basisfunktionalitäten bereit, auf die alle Unterklassen zugreifen können. Daher ist es natürlich möglich und auch vorgesehen, dass benutzerspezifische Komponenten von dieser Klasse erben. Häufig ist es jedoch ratsamer, von einer bereits bestehenden Komponentenklasse zu erben, beispielsweise direkt von `UICommand` oder `UIInput`, da man hierbei nur noch Anpassungen für die eigene benutzerspezifische Komponente vornehmen muss. Ansonsten ist eine komplette Implementierung der Komponente notwendig.

UI-Komponentenklassen, die mit JSF bereits ausgeliefert werden, implementieren standardmäßig bereits einige Interfaces, die im Falle eines Ableitens der eigenen Komponente somit ebenfalls implementiert sind. Häufig verwendete Interfaces im Zusammenhang mit Komponenten sind u.a.:

- `ActionSource`: Bei Implementierung dieses Interfaces wird die Komponente in die Lage versetzt, selbst ActionEvents auslösen zu können.
- `NamingContainer`: Dieses Interface bewirkt, dass alle Komponenten eine eindeutige Id besitzen müssen. Daher muss dieses Interface von sämtlichen UI-Komponenten implementiert werden.
- `StateHolder`: Nicht alle Komponenten müssen ihren Zustand zwischen verschiedenen Requests speichern. Wird dieses Verhalten jedoch benötigt, ist die Verwendung dieses Interfaces notwendig.
- `ValueHolder`: Die Komponente besitzt sowohl einen lokalen Wert, hat aber auch die Möglichkeit, auf ein Modellobjekt zugreifen zu können. Ebenso kann eine Konvertierung zwischen der Daten- und Präsentationssicht vorgenommen werden.

Je nach Komponentenklasse, von der eine benutzerdefinierte Komponente ableitet, sind daher bereits verschiedene Interfaces implementiert. Natürlich ist es jederzeit möglich, weitere Interfaces zu implementieren und damit das Verhalten einer benutzerdefinierten Komponente weiter zu beeinflussen.

Rendering

Die Entwicklung der Prozentanzeige-Komponente war bislang noch nicht allzu ereignisreich. Dies zeigt jedoch, dass eine Entwicklung einer benutzerspezifischen Komponente bei einer schrittweisen Vorgehensweise sehr einfach und überschaubar ist. Im Beispiel wird mit der Entwicklung der Renderer-Klasse jetzt das zentrale Element umgesetzt. Da der Schwerpunkt der Komponente ausschließlich in der Anzeige liegt, ist die Darstellung in eine separate Renderer-Klasse ausgelagert.

```
package com.edu.jsf.bsp.tag;

import java.io.IOException;
```

```
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;

/**
 * Rendererklasse für die Usagebar-Komponente
 */
public class UsageBarRenderer extends Renderer {

  private int total_length = 100;

  /**
   * Decodierung
   */
  public void decode(FacesContext context, UIComponent component) {
  }

  /**
   * Codierung
   */
  public void encodeBegin( FacesContext context,
                           UIComponent component )
      throws IOException {
    ResponseWriter writer = context.getResponseWriter();

    UIUsageBar uibar = (UIUsageBar)component;

    int red_length =  total_length / 100 * uibar.getInuse();
    int grey_length = total_length - red_length;

    writer.write("<table cellpadding=\"0\"
                   cellspacing=\"0\" border=\"0\">");
    writer.write("<tr>");
    writer.write("<td height=\"10\" bgcolor=\"#FF5807\">
        <img src=\"line.gif\" width=\"" + red_length
        + "\" height=\"10\" border=\"0\" alt=\"\"></td>");
    writer.write("<td height=\"10\" bgcolor=\"#C0C0C0\">
        <img src=\"line.gif\" width=\"" + grey_length
       + "\" height=\"10\" border=\"0\" alt=\"\"></td>");
    writer.write("</tr>");
    writer.write("<tr><td colspan=\"2\"><font size=\"2\">
       Verbrauch: " + uibar.getInuse() + "%</font></td></tr>");
    writer.write("</table>");
  }

  public void encodeChildren(FacesContext ctx, UIComponent comp)
      throws IOException {
  }
```

```
    public void encodeEnd(FacesContext ctx, UIComponent comp)
      throws IOException {
    }
  }
```

Listing 9.4: Renderer-Klasse für die Prozentanzeige-Komponente

Wichtig bei der Realisierung eigener Renderer-Klassen ist, dass diese von der Klasse `javax.faces.render.Renderer` ableiten. Die abstrakte Klasse `Renderer` beinhaltet insgesamt sieben Methoden, wovon für das Prozentanzeigebeispiel lediglich die Methode `encodeBegin` von Interesse ist. Weitere Methoden der Klasse sind:

- `decode`
- `encodeBegin`
- `encodeChildren`
- `encodeEnd`
- `getRendersChildren`
- `getConvertedValue`
- `convertClientId`

Während die `decode`-Methode einzig für die Decodierung zuständig ist, stehen für die Codierung drei Methoden zur Verfügung. Die Methode `encodeBegin` wird – wie der Name bereits vermuten lässt – zu Beginn der Codierungsphase aufgerufen. Sollte die Komponente – also quasi das Tag – Kindelemente aufweisen, wird eine Methode `encodeChildren` aufgerufen. Danach erfolgt noch der Aufruf der `encodeEnd`-Methode. Da im aktuellen Fall keine untergeordneten Elemente vorhanden sind, bleibt es dem Entwickler überlassen, die Codierung in der Beginn- oder Endephase einzubauen. Im Beispiel wird die `encodeBegin`-Methode verwendet.

Die `decode`-Methode ist dafür zuständig, Werte aus einem Request unter Zuhilfenahme entsprechender Konverter in das zugrunde liegende Objekt zurückzuschreiben. Die `encode`-Methoden dagegen sind für die eigentliche Darstellung – das Rendering – der Komponente zuständig. Da im Beispiel der Prozentanzeige-Komponente bei der Decode-Phase keine Werte der Komponente in das Modellobjekt zu übernehmen sind, ist die Methode nicht weiter zu implementieren. Der gesamte Quellcode für das Rendering ist im Beispiel in der Methode `encodeBegin` zu finden. Anhand der Eigenschaft `inuse` wird die Breite der linken (roten) Spalte sowie der rechten (grauen) Spalte berechnet. Die nach diesen Berechnungen erzeugte Tabelle wird über die `write`-Methode ausgegeben. Damit die Einhaltung der Zellenbreite korrekt funktioniert, wird in jeder Zelle ein transparentes Bild *line.gif* eingebaut. Dieses wird durch die explizite Angabe der Breite genau auf die gewünschte Breite gesetzt und steuert damit die Breite der Tabellenzelle. Im Beispiel wird davon ausgegangen, dass das benötigte Bild im Hauptverzeichnis der Webanwendung zu finden ist.

Die Methode `convertClientId`, die im Beispiel nicht implementiert ist, kann dazu verwendet werden, um die Id, die eine Komponente auf Clientseite später trägt, in eine andere Form umzuwandeln. Im Normalfall kann jedoch die Standardimplementierung verwendet werden, die eine Client-Id unverändert zurückliefert.

Im Beispiel der Kreditkartenkomponente später wird auf diese Methoden insgesamt nochmals explizit eingegangen, da im Kreditkartenbeispiel mehr passiert als in einer reinen Anzeigekomponente. Für den ersten Eindruck genügt es deshalb zu wissen, dass es in der Renderer-Klasse mehrere Methoden gibt, die für eine Codierung und Decodierung verantwortlich sind.

Bekanntmachen der Komponente und des Renderers

Bevor die Komponente in einer JSF-Seite eingesetzt werden kann, muss diese zunächst dem Framework bekannt gemacht werden. Zudem ist es notwendig, auch den separaten Renderer zu hinterlegen. Bisher wird lediglich in der Tagklasse ein Bezeichner `UsageBar` zurückgeliefert, eine Zuordnung des Bezeichners zur eigentlichen Rendererklasse steht jedoch noch aus. Beide Informationen, sowohl das Bekanntmachen der Komponente als auch des Renderers, sind in der Anwendungskonfigurationsdatei enthalten.

```
...
<component>
  <component-type>UsageBar</component-type>
  <component-class>
    com.edu.jsf.bsp.tag.UIUsageBar
  </component-class>
</component>

<render-kit>
  <renderer>
    <component-family>Graph</component-family>
    <renderer-type>UsageBar</renderer-type>
    <renderer-class>
      com.edu.jsf.bsp.tag.UsageBarRenderer
    </renderer-class>
  </renderer>
</render-kit>
...
```

Listing 9.5: Auszug aus der faces-config.xml

Jedes Render-Kit, das in der Anwendungskonfigurationsdatei hinterlegt ist, hat im Normalfall eine eindeutige Bezeichnung. Wird kein Bezeichner angegeben, gehört ein Renderer zum Standard-Render-Kit. Soll explizit ein Bezeichner angegeben werden, kann mittels des `render-kit-id`-Tags eine entsprechende Id vergeben werden.

Für die Komponente selbst wird über das Tag `component-type` ein Bezeichner definiert, der in der Tagklasse in der Methode `getComponentType` wiederum zurückgeliefert wird. Diesem Typ ist die Klasse `com.edu.jsf.bsp.tag.UIUsageBar` zugeordnet.

Der Renderer `UsageBar` wird durch die Renderer-Klasse `com.edu.jsf.bsp.tag.UsageBar-Renderer` realisiert. Auch hier erfolgt wieder eine einfache Zuordnung zwischen Bezeichner und konkreter Klasse. Ebenfalls wird eine Komponentenfamilie für einen Renderer mit angegeben.

Verwendung der Komponente

Nach diesen Vorarbeiten kann die Komponente in einer JSF-Seite angewendet werden. Die Einbindung des Tags ist relativ einfach, da lediglich ein Attribut für die Prozentanzeige mit übergeben werden muss.

```
<%@ taglib uri="HTTP://java.sun.com/jsf/HTML" prefix="h" %>
<%@ taglib uri="HTTP://java.sun.com/jsf/core" prefix="f" %>

<%@ taglib uri="/WEB-INF/custom-jsf.tld" prefix="cu" %>
<HTML>
<head>
  <title>Benutzerspezifische UI-Komponenten</title>
</head>

<body>
<f:view>
  <h3>Die Usagebar-Komponente</h3>
  <h:form id="frmTest">
    <cu:usageBar inuse="63" /><br>
    <br>
    <h:commandButton value="Submit" action="submit" />
    <br><br>
  </h:form>
</f:view>
</body>
</HTML>
```

Listing 9.6: Einbindung der Prozentanzeige-Komponente

Wichtig ist in Listing 9.6, dass die Tag-Bibliothek *custom-jsf.tld* über eine entsprechende Anweisung mit eingebunden wird, da ansonsten die Angabe `<cu:usagebar>` ins Leere läuft. Als Erweiterung für diese Komponente wäre denkbar, anstatt eines fixen Wertes eine Rechenoperation bzw. weitere Modellobjekte zu übergeben, aus denen sich die Komponente selbst den prozentualen Wert berechnen kann. Auch könnte eine interessante Alternative sein, die Darstellung dreidimensional wählen zu können. Da es möglich ist, für ein und dieselbe Komponente verschiedene Renderer einzusetzen, wird dieser Fall in der folgenden Kreditkartenkomponente genauer betrachtet.

Zustandsspeicherung

In Kapitel 6.11 wurde bereits ein kurzer Überblick über das Thema Zustandsspeicherung (State Saving) gegeben. In JSF wird grundsätzlich eine View immer gespeichert und bei einem erneuten Request wieder hergestellt (in der Phase *Restore View*). Dabei existieren zwei Möglichkeiten: die View wird serverseitig oder clientseitig (im Browser) gespeichert.

Wird die View serverseitig gespeichert, bleibt das Viewobjekt als solches in der Session erhalten. In Nicht-Cluster-Umgebungen ist es im Normalfall nicht notwendig, die Session zu serialisieren und z.B. in eine Datenbank zu speichern. Wird dagegen eine clientseitige Speicherung der View verwendet, muss das Viewobjekt serialisiert und im Browser mit Hilfe so genannter versteckter Felder (Hidden Fields) abgelegt werden. Bei einem erneuten Request erfolgt die Wiederherstellung der View basierend auf den serialisierten Objekten. Dieser grundsätzliche Unterschied läuft im aktuellen Beispiel auf eine interessante Konsequenz hinaus:

Bisher war noch keine Funktionalität in der Komponente vorhanden, die eine Zustandsspeicherung unterstützt. Wird somit eine serverseitige Speicherung verwendet, funktioniert die Anwendung erstaunlicherweise dennoch. Interessant wird es erst, wenn im Deployment Deskriptor eine clientseitige Speicherung eingestellt und mit dieser Einstellung die Testseite für die Prozentanzeigekomponente aufgerufen wird. Zunächst ist die Darstellung korrekt, es wird z.B. der Wert 63% angezeigt. Wird jedoch der Abschicken-Knopf betätigt, wird in der Anwendungskonfigurationsdatei nachgeschaut, welches die Folgeseite ist. Da hierfür keine Regel hinterlegt ist, wird wieder auf dieselbe Seite verwiesen. Weil diese jedoch bereits aktuell ist, wird sie somit zunächst aus den gespeicherten Informationen wiederhergestellt. Da jedoch die Eigenschaft der Komponente, dass der Prozentwert 63 beträgt, nicht mit serialisiert wurde und damit auch nicht in den Informationen der versteckten Felder enthalten ist, erfolgt die Darstellung wie in Abbildung 9.2.

Das Verständnis für das »Fehlverhalten« in Abbildung 9.2 ist sehr wichtig. Es kommt dadurch sehr deutlich zum Ausdruck, was eine Zustandsspeicherung bewirkt: Eine Komponente wird komplett in ihrem aktuellen Zustand serialisiert und bei Bedarf wiederhergestellt. Um auch mit clientseitiger Zustandsspeicherung des Beispiel funktionsfähig zu machen, muss die Komponentenklasse `UIUsageBar` wie folgt erweitert werden.

```
public Object saveState(FacesContext context) {
  Object values[] = new Object[2];
  values[0] = super.saveState(context);
  values[1] = new Integer(inuse);
  return (values);
}

public void restoreState(FacesContext context, Object state) {
```

```
    Object values[] = (Object[]) state;
    super.restoreState(context, values[0]);
    inuse = ((Integer)values[1]).intValue();
  }
```

Listing 9.7: Erweiterung der Komponentenklasse UIUsageBar

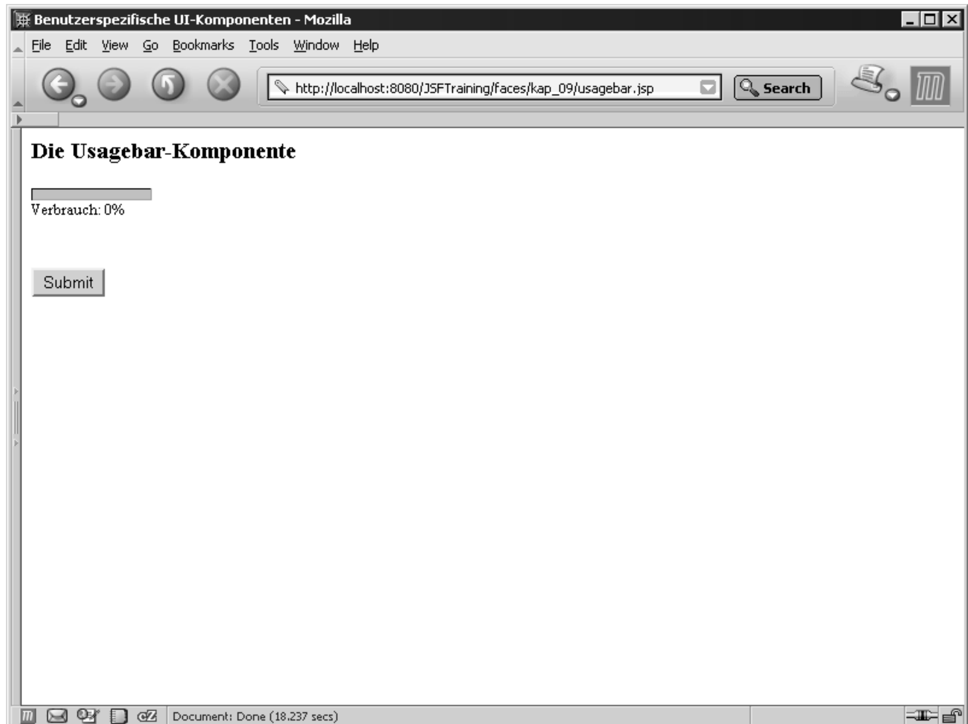


Abbildung 9.2: Fehlerhafte Zustandsspeicherung

Die beiden in Listing 9.7 aufgeführten Methoden gehören zum Interface `javax.faces.component.StateHolder`, das von Klassen implementiert werden muss, die ihren Zustand zwischen verschiedenen Requests speichern müssen. Im Fall der Prozentanzeigekomponente wird die Eigenschaft inuse abgespeichert, die den prozentualen Wert beinhaltet. Mit dieser Erweiterung ist die Prozentanzeigekomponente auch mit clientseitiger Zustandsspeicherung lauffähig.

Um es nochmals zu betonen: Dass das Beispiel auch ohne die zuletzt hinzugefügten Methoden mit serverseitiger Zustandsspeicherung funktioniert, soll nicht bedeuten, dass diese Methode vorzuziehen ist. Vielmehr ist es als glücklicher Zustand zu verstehen, dass es auf diese Weise funktioniert. Dies ist jedoch nicht garantiert. Da die Einstellung, ob serverseitige oder clientseitige Zustandsspeicherung verwendet wird,

nicht immer vom Entwickler vorgegeben werden kann, muss sichergestellt werden, dass eine Anwendung immer lauffähig ist. Somit sind die besagten Methoden *immer* zu implementieren, wenn ein Zustand abgespeichert werden soll.

9.1.2 Eine Eingabekomponente für Kreditkartendaten

Im letzten Abschnitt wurde die Erstellung einer benutzerspezifischen Komponente am Beispiel einer Prozentanzeigekomponente beschrieben. Dabei war das Rendering in einer separaten Klasse ausgelagert. Im folgenden Beispiel wird das Rendering exemplarisch in der Komponente selbst durchgeführt, wodurch zwar die Modularisierung einer Anwendung etwas eingeschränkt wird, aber dennoch eine legale Möglichkeit darstellt. Es ist beim Erstellen benutzerspezifischer Komponenten am Anfang daher immer zu überlegen, ob sich eventuell die Darstellung ändern kann, was für den Einsatz einer separaten Rendererklasse spricht. Ist dagegen die Darstellung relativ konstant, kann ohne Bedenken das Rendering in der Komponentenklasse selbst stattfinden. Stellt man jedoch im Verlauf der Entwicklung fest, dass das Rendering, das in der Komponentenklasse fixiert ist, doch variabler gehandhabt werden muss, können dennoch der Komponente weitere Renderer zugeordnet werden, obwohl die Komponente selbst ebenfalls das Rendering übernimmt. Wie genau dies funktioniert, werden Sie in Kapitel 9.2.3 erfahren.

Als neue Komponente wird eine Eingabekomponente für Kreditkartendaten entwickelt. Bei Bezahlungen über das Internet wird häufig eine Kreditkarte benötigt. Dabei sind verschiedene Angaben zur Kreditkarte selbst zu treffen. Das ist zum einen der Kreditkartentyp (American Express, Mastercard, Visa, ...), die Kreditkartennummer sowie das Gültigkeitsdatum. Seit neuestem wird auch häufig eine so genannte Kontrollnummer abgefragt. Dies ist eine 4-stellige etwas kleiner eingedruckte Nummer auf der Vorderseite der Karte. Dies verdeutlicht auch wiederum, wie wichtig und sinnvoll es ist, hier mit Komponenten zu arbeiten. Würden in einer oder mehreren Webanwendungen die Kredikartendaten mit einzelnen, nicht zusammengehörigen Eingabefeldern für die einzelnen Kredikartendaten gearbeitet, müssten bei Änderungen (beispielsweise dass seit neuestem die Kontrollnummer mit abgefragt wird) alle Eingabemasken verändert werden. Es kann dabei auch passieren, dass die eine oder andere Eingabemaske eventuell übersehen wird oder Fehler beim Ändern der vielen Masken passieren. Jedenfalls ist dies mit sehr viel Aufwand verbunden. Bei Einsatz einer eigenen Komponente genügt es, nur diese Komponente (bzw. den Renderer) selbst zu erweitern. Die geänderte Komponente steht damit automatisch jeder Anwendung zur Verfügung, die auf diese Komponente zurückgreift.

Ziel in den folgenden Abschnitten ist es daher, eine Komponente zu entwerfen, die alle notwendigen Kreditkarteninformationen abfragt und diese validiert, wo dies ohne Zugriff auf externe Anbieter möglich ist.

Für die Entwicklung einer benutzerspezifischen Komponente werden daher folgende Entwicklungsschritte durchlaufen, die dem Prinzip der Entwicklung einer Prozentanzeige-Komponente entsprechen, jedoch mit einigen Erweiterungen versehen sind:

1. Schreiben einer Taghandler-Klasse, wobei hierbei kein separater Renderer zurückgeliefert wird
2. Erzeugen eines TLDs, der die Beschreibung der Komponente enthält
3. Entwicklung der Komponentenklasse selbst
4. Implementieren des Codes für das Rendering in der Komponentenklasse
5. Einbinden der neuen Taglib in eine JSF-Seite sowie Verwendung und Test der Komponente

Der Ablauf unterscheidet sich nicht wesentlich von dem der Prozentanzeige-Komponente. Da die Kreditkartenkomponente jedoch Werte von einem Benutzer entgegennehmen kann, anstatt nur darstellenden Charakter zu haben, sind einige Entwicklungsschritte mehr einzuplanen.

Die Taghandler-Klasse

Auch für die Kreditkarten-Komponente ist wiederum eine Taghandler-Klasse notwendig. Im Gegensatz zur Kreditkartenkomponente muss in der Tag-Klasse jedoch angegeben werden, dass kein separater Renderer dafür verwendet wird, sondern das Rendering in der Komponente selbst durchgeführt wird.

```
package com.edu.jsf.bsp.tag;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentTag;

/**
 * Tag-Klasse für die Kreditkartenkomponente
 */
public class CreditCardTag extends UIComponentTag {

  private String value = "";

  /**
   * liefert den Renderer zurück
   */
  public String getRendererType() {
    return null;
  }

  /**
   * liefert den Komponententyp zurück
```

```
     */
    public String getComponentType() {
      return "CreditCardArea";
    }

    /**
     * liefert den Wert der Eigenschaft 'value' zurück
     */
    public String getValue() {
      return value;
    }

    /**
     * setzt einen neuen Wert für 'value'
     */
    public void setValue( String value ) {
      this.value = value;
    }

    /**
     * setzt die Properties in die Komponente
     */
    protected void setProperties(UIComponent component) {
      super.setProperties( component );
      UICreditCard cmp = (UICreditCard) component;
      if ( UIComponentTag.isValueReference(value) ) {
        ValueBinding vb =
          FacesContext.getCurrentInstance().getApplication()
          .createValueBinding(value);
        cmp.setValueBinding("value", vb);
      } // if
  }
```

Listing 9.8: Die Tag-Klasse für die Kreditkartenkomponente

In Listing 9.8 ist die Tag-Klasse für die Kreditkartenkomponente abgebildet. Die Klasse `CreditCardTag` ist wieder von der Klasse `javax.faces.webapp.UIComponentTag` abgeleitet.

Die Klasse besitzt als einzige Eigenschaft den String `value`. Die setter- und getter-Methoden in der Tagklasse werden entsprechend durch den Tag-Handler aufgerufen und die angegebenen Werte gesetzt bzw. ausgelesen. Dieser Wert, der in der JSF-Seite gesetzt wird, wird in der Methode `setProperties` an die zuständige Komponente weitergereicht. Da im Fall der Kreditkartenkomponente der `value`-Wert eigentlich immer einen `ValueBinding`-Ausdruck darstellen sollte, wird der Ausdruck mittels

```
setValueBinding
```

direkt in die Komponente gesetzt. In der Methode `getComponentType` wird ein Bezeichner für die Komponente zurückgeliefert, die diese Methode bereitstellt. Der Bezeichner `CreditCardArea` ist wiederum in der Anwendungskonfigurationsdatei hinterlegt.

In der Methode `getRendererType` wird festgelegt, ob eine separate Klasse für das Rendering zuständig ist. Da in diesem Beispiel die Darstellung in der Komponente selbst implementiert wird, wird an dieser Stelle `null` zurückgeliefert. Zur Erinnerung: Im Beispiel der Prozentanzeige-Komponente wurde an dieser Stelle ein Bezeichner zurückgeliefert, der in der Anwendungskonfigurationsdatei eine Zuordnung zu einer Renderer-Klasse hatte, die für das Rendering zuständig war.

Der Tag-Library-Deskriptor

Damit ein entsprechendes Tag für die Kreditkarten-Komponente in der Anwendung selbst verwendet werden kann, muss das Tag über den Tag-Library-Deskriptor zunächst bekannt gemacht werden. Dazu kann eine neue Datei angelegt oder aber der während der Entwicklung der Prozentanzeige-Komponente erzeugte Deskriptor erweitert werden. In Listing 9.9 ist der entsprechend erweiterte Deskriptor abgebildet.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"HTTP://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">

<taglib>

  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>Custom Tag Library for JSF</short-name>
  <uri>HTTP://java.sun.com/jsf/HTML</uri>
  <description>
    Further tags for JSF
  </description>

  <tag>
    <name>usageBar</name>
    <tag-class>com.edu.jsf.bsp.tag.UsageBarTag</tag-class>
    <body-content>none</body-content>
    <attribute>
      <name>inuse</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>

  <tag>
    <name>creditCard</name>
    <tag-class>com.edu.jsf.bsp.tag.CreditCardTag</tag-class>
    <body-content>none</body-content>
    <attribute>
      <name>value</name>
      <required>true</required>
```

```
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>

</taglib>
```

Listing 9.9: Der erweiterte Deskriptor für die Kreditkartenkomponente

Wie in Listing 9.9 zu sehen ist, enthält der Deskriptor die Beschreibung für das zusätzliche Tag `creditCard`. Zusätzlich ist dem Tag das Attribut `value` mit anzugeben, das eine Muss-Angabe darstellt. Die Tagklasse `com.edu.jsf.bsp.tag.CreditCardTag` ist vollqualifizierend anzugeben. Das Verzeichnis, in dem der Deskriptor abgespeichert wird, ist prinzipiell dem Entwickler überlassen, es empfiehlt sich jedoch, die Datei direkt im *WEB-INF*-Verzeichnis abzulegen.

Entwicklung der Komponentenklasse

Im Gegensatz zur Prozentanzeige-Komponente ist die Komponentenklasse bei der Kreditkartenkomponente wesentlich komplexer. Nicht nur, dass hierbei Daten aus dem Request in das Modell zurückgeschrieben werden müssen, es erfolgt auch die komplette Darstellung der Komponente in der Klasse selbst.

Wie bereits bei der Komponentenklasse für die Prozentanzeige-Komponente erwähnt, ist eine Komponentenklasse für folgende Aufgaben zuständig:

- Codierung
- Decodierung
- Validierung
- Speicherung des Zustandes
- Modellupdate
- Eventverarbeitung

Die Codierungs- und Decodierungsmethoden sind äußerst wichtige Methoden. In ihnen findet die Transformation aus der Präsentationssicht in die Modellsicht und umgekehrt statt. Während bei der Codierung lokale Daten aus der Komponente in die Präsentationssicht umgewandelt werden, werden bei der Decodierung die Werte aus den Requestparametern extrahiert und wiederum in den lokalen Variablen abgespeichert. Unter Umständen findet noch eine Datenvalidierung statt. Danach werden sämtliche Daten gesammelt in das Modellobjekt überführt. Bei Bedarf werden während dieses Ablaufs Events ausgelöst, wie z.B. ein *ValueChangeEvent* bei einer Werteänderung.

Die Kreditkartenkomponente besitzt mehrere Eigenschaften. So müssen eine Kreditkartennummer, eine Kontrollnummer sowie der Monat und das Jahr der Gültigkeitsdauer eingegeben werden. Natürlich kann in einer Anwendung für jede Eigenschaft ein gesondertes Attribut definiert werden, das den Bezug auf ein Modellobjekt hält. Dies ist jedoch nicht sehr elegant und zudem sehr fehleranfällig. Daher wird eine Klasse `CreditCardData` `verwendet`, die die entsprechenden Daten gesammelt vorhält. Dies hat den Vorteil, dass komfortabel über die Datenklasse zugegriffen werden kann anstatt über viele Einzelwerte. Daher ist bereits im Tag sowie in der Tagklasse nur ein Attribut `value` vorgesehen. Hiermit wird eine Eigenschaft übergeben, die sich auf ein Objekt der Klasse `CreditCardData` bezieht.

```
package com.edu.jsf.bsp.tag;

/**
 * Klasse für Kreditkarten-Angaben
 */
public class CreditCardData {

  private String number;
  private String check ;
  private int validMonth;
  private int validYear;

  /**
   * liefert die Kreditkartennummer zurück
   */
  public String getNumber() {
    return number;
  }

  /**
   * liefert die Kontrollnummer einer Kreditkarte zurück
   */
  public String getCheck() {
    return check;
  }

  /**
   * liefert zurück, bis zu welchem Monat die Kreditkarte
   * gültig ist
   */
  public int getValidMonth() {
    return validMonth;
  }

  /**
   * liefert zurück, bis zu welchem Jahr die Kreditkarte
   * gültig ist
   */
```

```
  public int getValidYear() {
    return validYear;
  }

  /**
   * setzt eine neue Kreditkartennummer
   */
  public void setNumber(String string) {
    number = string;
  }

  /**
   * setzt eine neue Kontrollnummer
   */
  public void setCheck(String string) {
    check = string;
  }

  /**
   * setzt den Monat für die Gültig-Bis-Angabe
   */
  public void setValidMonth(int i) {
    validMonth = i;
  }

  /**
   * setzt das Jahr für die Gültig-Bis-Angabe
   */
  public void setValidYear(int i) {
    validYear = i;
  }
}
```

Listing 9.10: CreditCardData-Klasse

Listing 9.10 zeigt eine einfache Beanklasse, die für die Datenhaltung von Kreditkartendaten zuständig ist. Diese weist lediglich vier Eigenschaften für die Kreditkartennummer, die Kontrollnummer, den Gültigkeitsmonat sowie für das Gültigkeitsjahr aus. Zusätzlich existieren die notwendigen setter- und getter-Methoden.

Eine eigentliche UI-Komponentenklasse ist wiederum von der Klasse `UIComponentBase` abgeleitet. Im Gegensatz zur Prozentanzeige-Komponente ist diese sehr umfangreich, da sowohl das komplette Rendering sowie auch der Modellupdate darin erfolgen.

```
package com.edu.jsf.bsp.tag;

import java.util.HashMap;
import java.util.Map;

import javax.faces.FacesException;
```

```
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.component.ValueHolder;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.Converter;
import javax.faces.el.ValueBinding;

/*
 * Komponentenklasse für die Kreditkartenkomponente
 */
public class UICreditCard extends UIInput implements ValueHolder {

  private Object localvalue = null;
  private boolean valid;

  public UICreditCard() {
    setRendererType( null );
  }

  /*
   * Codierung
   */
  public void encodeBegin( FacesContext context)
     throws java.io.IOException {
  }

  public void encodeChildren( FacesContext context)
    throws java.io.IOException {
  }

  public void encodeEnd( FacesContext context)
    throws java.io.IOException {

    HashMap valueMap = (HashMap)getValue();

    // Rendering
    ResponseWriter writer = context.getResponseWriter();
    writer.write("<table border=\"0\">");
    writer.write("<tr>");
    writer.write("<td>Kartennummer:</td>");
    writer.write("<td><input type=\"text\" ");
    writer.write(" name=\"" + getClientId(context) + "_number\"");
    writer.write(" value=\"" + valueMap.get("number")
      + "\" size=\"20\"></td>");
    writer.write("</tr><tr>");
    writer.write("<td>Kontrollnummer:</td>");
    writer.write("<td><input type=\"text\" ");
    writer.write(" name=\"" + getClientId(context) + "_control\"");
    writer.write(" value=\"" + valueMap.get("check")
      + "\" size=\"4\"></td>");
```

```
  writer.write("</tr><tr>");
  writer.write("<td>Gültig bis:</td>");
  writer.write("<td><input type=\"text\" ");
  writer.write(" name=\"" + getClientId(context) + "_month\"");
  writer.write(" value=\"" + valueMap.get("month")
    + "\" size=\"3\">");
  writer.write("<input type=\"text\" ");
  writer.write(" name=\"" + getClientId(context) + "_year\"");
  writer.write(" value=\"" + valueMap.get("year")
    + "\" size=\"3\"></td>");
  writer.write("</tr>");
  writer.write("</table>");
}

/*
 * Decodierung
 */
public void decode(FacesContext context)
    throws NullPointerException {

  if (context == null) {
    throw new NullPointerException();
  } // if

  UIComponent comp = (UIComponent)this;

  Map requestMap =
    context.getExternalContext().getRequestParameterMap();

  try {
    // Zunächst die Werte des Requests lokal
    // zwischenspeichern
    localvalue = new HashMap();
    ((HashMap)localvalue).put( "number",
      requestMap.get(getClientId(context) + "_number") );
    ((HashMap)localvalue).put( "check",
      requestMap.get(getClientId(context) + "_control") );
    ((HashMap)localvalue).put( "month",
      requestMap.get(getClientId(context) + "_month") );
    ((HashMap)localvalue).put( "year",
      requestMap.get(getClientId(context) + "_year") );

    // Danach den Werte validieren
    String newValue;
    newValue = (String) requestMap.get(getClientId(context)
      + "_month");
    Integer.parseInt(newValue);

    newValue = (String) requestMap.get(getClientId(context)
      + "_year");
    Integer.parseInt(newValue);
```

```
      setValid( true );

    } catch (Exception exc) {
      context.addMessage(this.getClientId( context ),
        new FacesMessage() {
          public String getDetail() {
            return "Die Kreditkartendaten sind nicht korrekt.";
          }

          public Severity getSeverity() {
            return FacesMessage.SEVERITY_ERROR;
          }

          public String getSummary() {
            return "Die Kreditkartendaten sind nicht korrekt.";
          }
        });

      setValid( false );
    } // catch
  }

  /*
   * Aktualisiert das Modell
   */
  public void updateModel( FacesContext context ) {
    super.updateModel( context );

    if (context == null)
      throw new NullPointerException();

    if ( isValid() ) {
      if (getValue() != null) {
        try {
          ValueBinding binding = getValueBinding( "value" );
          HashMap valueMap = (HashMap)getValue();
          CreditCardData ccdata = new CreditCardData();

          ccdata.setNumber( valueMap.get("number").toString() );
          ccdata.setCheck( valueMap.get("check").toString() );
          ccdata.setValidMonth(
            Integer.parseInt(valueMap.get("month").toString() ) );
          ccdata.setValidYear(
            Integer.parseInt(valueMap.get("year").toString() ) );

          binding.setValue(context, ccdata );
          localvalue = null;
        } catch (FacesException e) {
          setValid( false );
          throw e;
        } catch (IllegalArgumentException e) {
```

```
          setValid( false );
          throw e;
        } catch (Exception e) {
          setValid( false );
          throw new FacesException(e);
        } // catch
      } // if
    } // if
  }

  /*
   * setzt einen neuen Wert
   */
  public void setValue( Object value ) {
    if ( value!=null && value instanceof CreditCardData ) {
      CreditCardData ccData = (CreditCardData)value;
      ((HashMap)localvalue).put("number", ccData.getNumber() );
      ((HashMap)localvalue).put("check", ccData.getCheck() );
      ((HashMap)localvalue).put("month",
        new Integer(ccData.getValidMonth()) );
      ((HashMap)localvalue).put("year",
        new Integer(ccData.getValidYear()) );
    } else {
      localvalue = value;
    } // else
  }

  public CreditCardData getCreditCardData() {
    CreditCardData cData = new CreditCardData();
    if ( getValue()!=null ) {
      cData.setNumber(
        ((HashMap)getValue()).get("number").toString() );
      cData.setCheck(
        ((HashMap)getValue()).get("check").toString() );
      cData.setValidMonth( Integer.parseInt(
        ((HashMap)getValue()).get("month").toString()) );
      cData.setValidYear( Integer.parseInt(
        ((HashMap)getValue()).get("year").toString()) );
    } // if
    return cData;
  }

  /*
   * liefert den aktuellen Wert zurück
   */
  public Object getValue() {
    if (localvalue != null) {
      return localvalue;
    } // if
    ValueBinding vb = getValueBinding("value");
    if (vb != null) {
      CreditCardData ccdata = (CreditCardData)
```

```
          vb.getValue(getFacesContext());
        localvalue = new HashMap();
        ((HashMap)localvalue).put("number", ccdata.getNumber() );
        ((HashMap)localvalue).put("check", ccdata.getCheck() );
        ((HashMap)localvalue).put("month",
          new Integer(ccdata.getValidMonth()) );
        ((HashMap)localvalue).put("year",
          new Integer(ccdata.getValidYear()) );

        return localvalue;
      } // if
      return null;
    }

    /*
     * liefert den lokalen Wert zurück
     */
    public Object getLocalValue() {
      return localvalue;
    }

    /*
     * zeigt den Zustand der Komponente an
     */
    public boolean isValid() {
      return valid;
    }

    public void setValid( boolean v ) {
      valid = v;
    }

    /*
     * Setter- und Getter für eventuelle Converter
     */
    public Converter getConverter() {
      return null;
    }

    public void setConverter(Converter arg0) {
    }

    /*
     *  Bezeichner der Komponentenfamilie.
     */
    public String getFamily() {
      return( "CreditCard" );
    }
  }
```

Listing 9.11: Die Komponentenklasse UICreditCard

In Listing 9.11 ist die Komponentenklasse für die Kreditkartenkomponente zu sehen. Sämtliche Funktionalität zur Darstellung sowie zur Verarbeitung der Ein- und Ausgabewerte sind darin enthalten. Für ein tiefer gehendes Verständnis von JSF wird daher in den folgenden Abschnitten auf jede wichtige Methode im Einzelnen eingegangen.

Codierung

Betrachtet man den Lebenszyklus einer JSF-Seite, so wird am Ende der Verarbeitung die Phase *Render Response* durchgeführt. In dieser Phase wird anhand des Komponentenbaums jede einzelne Komponente, die auf der Seite enthalten ist, entsprechend codiert und als Markup-Darstellung im Browser zur Anzeige gebracht. Innerhalb der Komponente muss daher eine Codierung stattfinden, die einerseits die Darstellung der Komponente in die Markup-Sprache durchführt, andererseits auch die lokalen Werte mit in die Markup-Darstellung einfließen lässt.

Die Klasse `UIComponentBase` stellt für die Codierung bereits drei Methoden zur Verfügung, die bei Bedarf überschrieben werden können.

- `encodeBegin`
- `encodeChildren`
- `encodeEnd`

Die Methode `encodeBegin` wird zu Beginn der Codierungsphase aufgerufen. Sollte das Tag Kindelemente aufweisen, wird eine Methode `encodeChildren` aufgerufen. Danach erfolgt der Aufruf der `encodeEnd`-Methode. Da in obigem Fall keine untergeordneten Elemente vorhanden sind, bleibt es dem Entwickler überlassen, die Codierung in der Beginn- oder Endephase einzubauen. Im Beispiel wird die `encodeEnd`-Methode verwendet.

Als ersten wichtigen Schritt wird mittels

```
HashMap valueMap = (HashMap)getValue();
```

eine Map mit den aktuell gesetzten Werten abgefragt. Die Methode `getValue` wird einerseits auf ein Objekt `localvalue` abgefragt, andererseits auch auf den Wert über ein `ValueBinding`. Dies hat den folgenden Hintergrund:

Angenommen, im Beispiel der Kreditkartenkomponente werden eine gültige Kreditkartennummer und eine gültige Kontrollnummer eingegeben. Leider wurde jedoch ein falsches Jahr (z.B. ein Buchstabe) eingegeben. Daraufhin wird ein Fehler erzeugt und die Komponente zusammen mit einer Fehlermeldung nochmals dargestellt. Beim erneuten Darstellen sollen natürlich die urspünglichen (falschen) Werte in den Eingabefeldern eingetragen bleiben, genauso wie die bereits korrekt gefüllten Felder. Da aber z.B. bei einer Jahreszahl kein Buchstabe im Modellobjekt gespeichert werden kann (die Jahreszahl ist ein `int`-Wert), werden in der Komponente (also lokal) die

durch den Benutzer eingegebenen Werte so lange vorgehalten, bis diese korrekt in das Modell übernommen und somit die lokalen Werte gelöscht werden können. Die lokalen Werte sind somit temporäre Speicher.

Sind keine lokalen Werte gesetzt, erfolgt ein Zugriff auf das Modellobjekt, bei dem die einzelnen Werte für die Kreditkartennummer, die Kontrollzahl, der Gültigkeitsmonat sowie das Jahr ausgelesen werden.

Liegen jedoch bereits lokale Werte vor, werden diese verwendet. Es erfolgt in diesem Fall kein Zugriff auf das Modellobjekt. Eine Validierung einer Seite schlägt fehl, sobald eine Komponente einen Fehler zurückliefert. Übertragen auf das Kreditkartenbeispiel bedeutet dies, dass ein Formular nicht abgeschickt werden kann, sollte z.B. das Jahr nicht korrekt eingegeben worden sein. Tritt ein Fehler auf einer Seite auf, werden keine Daten in das Modell zurückgeschrieben. Daher wird in einem Fehlerfall auch nicht auf das Modellobjekt zurückgegriffen, sondern die lokalen (weil aktuelleren) Werte angezogen. Nach einer erfolgreichen Validierung werden die Daten in das Modellobjekt zurückgeschrieben und die lokalen Werte gelöscht.

Aufgrund der Tatsache, dass die durch den Benutzer eingegebenen und möglicherweise falschen Werte jeden beliebigen Inhalt annehmen können, werden die lokalen Werte in einer `HashMap` als Objekte vorgehalten und können bei Bedarf auf den richtigen Datentyp gecastet werden.

Nachdem somit die zu setzenden Werte ermittelt wurden (lokale Werte oder Werte aus dem Modellobjekt), erfolgt über den Aufruf

```
ResponseWriter writer = context.getResponseWriter();
```

ein Zugriff auf einen `ResponseWriter`, über den mit Hilfe der `write`-Methode Ausgaben in den Ausgabestrom an den Browser geschickt werden können. Da jede Komponente in JSF mit einer eindeutigen Id identifiziert werden kann, ist es sehr wichtig, diese Id im `name`-Tag mit einfließen zu lassen. Die aktuelle Id der Komponente wird über den Aufruf `getClientId` abgefragt. Da im Beispiel die Komponente aus vier Eingabefeldern besteht, wird an jedes Eingabefeld zur Id noch ein weiterer Bezeichner angehängt, um diese vier Eingabefelder später wieder eindeutig identifizieren zu können.

Decodierung

Die `decode`-Methode ist das Gegenstück zur `encode`-Methode. Nachdem eine JSF-Seite auf Benutzerseite an den Server zurückgeschickt wurde, wird wieder anhand des Komponentenbaums für jede einzelne Komponente die `decode`-Methode aufgerufen und die im Request enthaltenen Werte werden verarbeitet. Sollte eine Komponente keine Werte entgegennehmen sowie auch keine Events auslösen, muss diese Methode nicht implementiert werden. Da die Kreditkartenkomponente die Eingabedaten aus

dem Request in das Modellobjekt übernehmen soll, ist in dieser Methode zu prüfen, ob die Werte in geeigneter Form vorliegen und konvertiert werden können. Im Beispiel ist die Überprüfung auf das Notwendigste beschränkt. So wird für den Monat und das Jahr einfach nur versucht, den Wert aus dem Request in einen `int`-Wert zu wandeln, schlägt dies fehl, wird ein Fehler erzeugt. In einer »richtigen« Anwendung kann an dieser Stelle eine weitere Überprüfung beispielsweise der Kreditkartennummern (Minimal- oder Maximallänge) sowie eine erste Prüfung der Kontrollziffer erfolgen.

Vor den Tests werden jedoch alle Werte aus dem Request in die lokalen Werte übernommen. Dazu dient wie bereits erläutert eine `HashMap`. Die lokalen Werte dienen dazu, im Fehlerfall den eingegebenen Wert dem Benutzer nochmals anzeigen zu können.

Laufen alle Konvertierungen ohne Fehler durch, stehen alle Werte des Requests in den lokalen Werten zur Verfügung. Des weiteren wird die Komponente mittels

```
setValid( true );
```

in einen gültigen Zustand versetzt. Bei einem Fehler wird eine Meldung erzeugt, die eine kurze Fehlerbeschreibung beinhaltet. Zudem wird der Zustand durch

```
setValid( false );
```

auf ungültig gesetzt. Damit wird kein Update des Modellobjekts durchgeführt, die lokalen Werte bleiben erhalten und werden bei einem erneuten Rendern der Komponente nochmals angezeigt.

Die Werte des Requests werden mit dem Aufruf

```
Map requestMap =
    context.getExternalContext().getRequestParameterMap();
```

in eine Map gestellt. Darauf kann über `get` auf die einzelnen Werte zugegriffen werden. Da in der Kreditkartenkomponente in der Codierungsphase die vier Eingabefelder mit der Id sowie einem Zusatz bezeichnet wurden, müssen die Werte auch wieder mit denselben Bezeichnern ausgelesen werden.

Aktualisieren des Modells

Ist während des Einlesens der Werte aus dem Request in die lokalen Variablen der einzelnen Komponenten kein Fehler aufgetreten, wird ein Modellupdate durchgeführt. Dies bedeutet, dass die lokalen Werte, die noch in der Komponente zwischengespeichert sind, in die zugrunde liegenden Modellobjekte überführt werden. Dazu wird die Methode `updateModel` verwendet. Auch an dieser Stelle kann wieder ein Fehler auftreten, bei dem die Komponente in einen invaliden Zustand gesetzt wird.

War der Update erfolgreich, können die lokalen Werte wieder mittels

```
localvalue = null;
```

auf `null` gesetzt werden, da bei einer erneuten Darstellung auf die aktuellen Werte im Modellobjekt zugegriffen werden kann.

Das Interface ValueHolder

Die Komponentenklasse implementiert das Interface `ValueHolder`. ValueHolder ist ein Interface, das von allen UI-Komponenten verwendet werden kann, die sowohl einen lokalen Wert speichern wie auch Zugriff auf ein Modellobjekt haben und eine entsprechende Konvertierung unterstützen. Für die Kreditkartenkomponente werden folgende Methoden überschrieben:

- `public java.lang.Object getLocalValue()`: Liefert den aktuell gesetzten lokalen Wert der Komponente zurück.
- `public java.lang.Object getValue()`: Liefert einen Wert zurück. Zunächst wird dabei der lokale Wert abgefragt. Ist dieser ungleich null, wird er zurückgeliefert. Wenn nicht, wird überprüft, ob ein ValueBinding-Ausdruck vorliegt, und gegebenenfalls dieser zurückgeliefert. Trifft auch dies nicht zu, wird null zurückgegeben.
- `public void setValue(java.lang.Object value)`: Setzt einen neuen (lokalen) Wert.

Bekanntmachen der Komponente

Die Hauptarbeit bei der Entwicklung der Kreditkartenkomponente ist vollbracht. Damit jedoch eine Anwendung diese Komponente benutzen kann, muss sie dem Framework zuerst noch bekannt gemacht werden. Dies erfolgt über die Anwendungskonfigurationsdatei *faces-config.xml*.

```
...
<component>
  <component-type>CreditCardArea</component-type>
  <component-class>
    com.edu.jsf.bsp.tag.UICreditCard
  </component-class>
</component>
...
```

Listing 9.12: Auszug aus der faces-config.xml

In der Konfigurationsdatei werden lediglich die Komponentenklasse sowie der Typ hinterlegt, der in der Tagklasse `CreditCardTag` in der Methode `getComponentType` zurückgeliefert wird. So schließt sich dann auch wieder der Kreis: Über die Tagklasse wird eine Typbezeichnung zurückgeliefert, wodurch damit automatisch die Komponentenklasse bestimmt ist.

Einsatz der Komponente

Jetzt sind aber definitiv alle Vorarbeiten geleistet und die Komponente kann in die erste JSF-Seite ingegriert werden. Wichtig ist natürlich, dass die Tag-Bibliothek mitangegeben wird, in der sich die Definition für das Kreditkarten-Tag befindet.

```
<%@ taglib uri="/WEB-INF/custom-jsf.tld" prefix="cu" %>
```

Listing 9.13: Einbinden der Tag-Bibliothek

Als Namensraum wurde in Listing 9.13 `cu` für `custom` gewählt, es kann jedoch auch ein beliebiges anderes Präfix verwendet werden.

```
<cu:creditCard value="CreditCardHolder.ccdata" />
```

Listing 9.14: Einbau der Kreditkartenkomponente

Die Kreditkartenkomponente weist nur ein Attribut `value` auf, das allerdings eine Muss-Angabe darstellt. Hier ist die Referenz auf ein Modellobjekt anzugeben, das die entsprechenden Daten aufnehmen kann. Wichtig ist, dass die Eigenschaft von der Klasse `CreditCardData` abstammt, da es ansonsten zu einem Fehler kommt. Natürlich ist der Bezeichner `CreditCardHolder` in der Anwendungskonfigurationsdatei im Bereich der Managed-Beans hinterlegt.

```
<managed-bean>
  <managed-bean-name>CreditCardHolder</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.CreditCardHolderBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Listing 9.15: Auszug aus faces-config.xml

Der Vollständigkeit halber ist in Listing 9.16 noch der Quellcode der Klasse `CreditCard-HolderBean` abgebildet.

```
package com.edu.jsf.bsp.bean;

import com.edu.jsf.bsp.tag.CreditCardData;

/**
 * Ein einfaches Bean für einen
 * Kreditkarteninhaber
 */
public class CreditCardHolderBean {
```

```
    private String firstname;
    private String lastname;
    private CreditCardData ccdata;

    /**
     * getter-Methoden
     */
    public String getFirstname() {
      return firstname;
    }

    public String getLastname() {
      return lastname;
    }

    public CreditCardData getCcdata() {
      return ccdata;
    }

    /**
     * setter-Methoden
     */
    public void setFirstname(String string) {
      firstname = string;
    }

    public void setLastname(String string) {
      lastname = string;
    }

    public void setCcdata(CreditCardData data) {
      ccdata = data;
    }
  }
```

Listing 9.16: CreditCardHolder-Bean

Nachdem die erste JSF-Seite erstellt wurde, kann ein erster Test im Browser durchgeführt und das (hoffentlich) erfolgreiche Ergebnis betrachtet werden.

In Abbildung 9.3 ist die Kreditkartenkomponente in Aktion zu sehen. Natürlich kann diese jetzt in ein beliebiges Umfeld eingebaut werden, in der beispielsweise ein Zahlungsvorgang bearbeitet wird. Wird bei der Eingabe des Monats oder des Jahres ein ungültiger Wert eingegeben (z. B. ein Buchstabe), so wird ein Fehler erzeugt und zur Anzeige gebracht. Voraussetzung ist natürlich, dass das `messages`-Tag auf der Seite eingebaut wurde.

Es kann ebenfalls nachvollzogen werden, dass im Falle eines Fehlers die weiteren eingegebenen Werte nicht gelöscht werden, sondern in der Komponente bestehen bleiben.

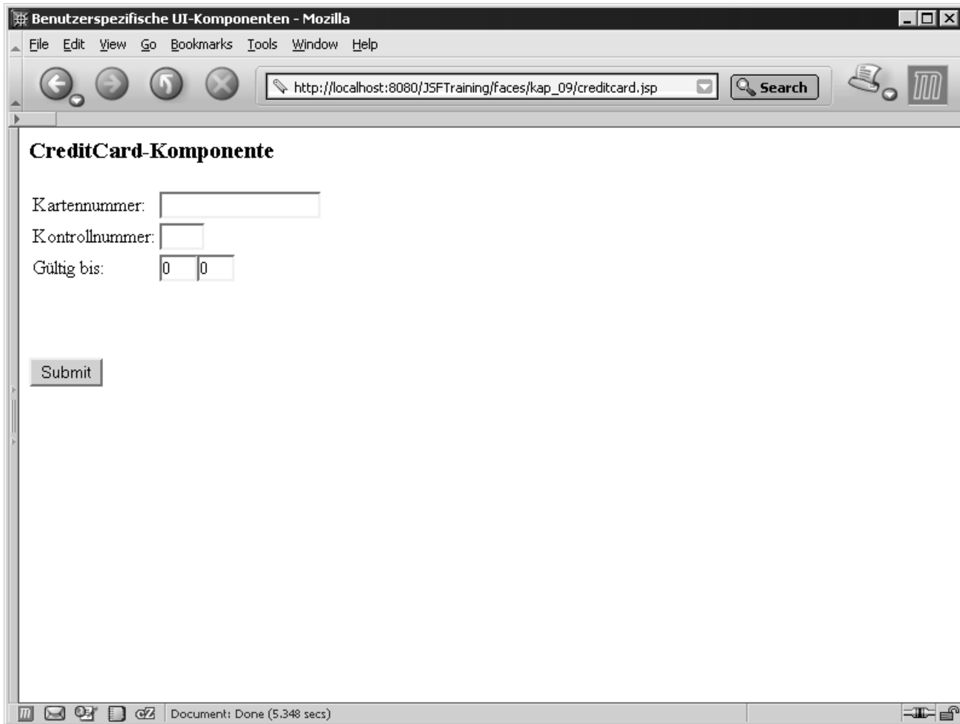


Abbildung 9.3: Die Kreditkartenkomponente

9.1.3 Die Kreditkartenkomponente II

In Abschnitt 9.2.2 wurde bereits eine recht komplexe benutzerdefinierte Komponente entwickelt. Im Gegensatz zur Prozentanzeige-Komponente wurde das Rendering in der Komponente selbst realisiert. Eventuell ist es vielleicht wünschenswert, noch einige kleinere Änderungen vorzunehmen. So ist die Eingabe des Monats und des Jahres über Texteingaben nicht allzu elegant, eine Auswahl über so genannte Drop-Down-Boxen wäre hierbei benutzerfreundlicher. Möchte man jedoch auch die Darstellungsform mit den Eingabefeldern nicht ablösen, sondern bedarfsweise die ein oder andere Darstellungsform einsetzen, ist es möglich, einen Renderer zu implementieren, der nur nach Bedarf angezogen wird. Auch dass das primäre Rendering in der Komponente selbst bereits implementiert ist, ist hierbei *kein* Hindernis. Vielmehr stellt das Rendering in der Komponentenklasse quasi das Standard-Verhalten dar, wenn kein spezieller Renderer mitangegeben wird. Dieses ist eine interessante Erweiterung der Taghandler-Klasse. Es wird daher in den folgenden Beispielen gezeigt, wie unterschiedliche Renderer angezogen werden können, ohne dafür eine spezielle Komponente erzeugen zu müssen.

Eine weitere Anforderung an die Kreditkartenkomponente II ist das Auslösen von `ValueChangeEvents`. Bisher wurde das gesamte Thema Event-Handling noch unberührt gelassen. Deshalb wird im Folgenden ebenfalls darauf eingegangen.

Vorgehensweise

Natürlich wird wieder auf die vorhandenen Klassen der Kreditkartenkomponente zurückgegriffen und es werden nur an bestimmten Stellen Erweiterungen vorgenommen. Daher sind das Anbinden eines weiteren Renderers sowie die Integration des Eventhandlings relativ schnell zu realisieren. Folgende Arbeitsschritte werden dabei unternommen:

1. Erweiterung der Taghandler-Klasse, die bei Bedarf einen speziellen Renderer zurückliefert.
2. Erweitern des TLD.
3. Entwickeln des neuen Renderers.
4. 4.Eintragen des Renderers in der Anwendungskonfigurationsdatei.
5. Anpassungen vornehmen in der Komponentenklasse.

Taghandler-Klasse

Die Taghandler-Klasse ist so zu erweitern, dass sie einmal den neuen Renderer zurückliefert, andererseits aber auch den Standard-Renderer in der Komponentenklasse verwendet. Zur Steuerung wird ein neues Attribut `style` im `CreditCard`-Tag eingeführt. Damit kann gesteuert werden, welcher Renderer zum Einsatz kommt.

```
...
public String getRendererType() {
  if ( style==null || style.equals("") )
    return null;
  else
    return "CreditCard";
}
...
```

Listing 9.17: Die erweiterte Methode zur Bestimmung des Renderers

Natürlich sind in der Tagklasse eine Variable `style` sowie die dazu passenden setter- und getter-Methoden definiert. In der Methode selbst wird nur abgefragt, ob ein Wert für `style` gesetzt ist. Dies bedeutet, dass hier ein beliebiger String übergeben werden kann. An dieser Stelle kann die Taghandler-Klasse auch dahingehend erweitert werden, dass abhängig vom Wert von `style` ein spezieller Renderer zurückgeliefert wird.

Ist dagegen das Attribut `style` nicht angegeben oder weist eine leere Zeichenkette auf, so wird `null` zurückgeliefert, was ein Rendering in der Komponentenklasse selbst bewirkt.

Anpassen des Tag-Library-Deskriptors

In der Taghandler-Klasse wird ein zusätzliches Attribut `style` eingeführt. Dieses muss natürlich im entsprechenden Deskriptor ebenfalls noch eingetragen werden. Das Attribut selbst muss nicht zwingend angegeben werden, da der Standard-Renderer der Komponentenklasse verwendet wird, wenn es nicht vorhanden ist. Es wird somit der Wert für `required` auf `false gesetzt`.

```
...
<tag>
  <name>creditCard</name>
  <tag-class>com.edu.jsf.bsp.tag.CreditCardTag</tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>value</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>style</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
...
```

Listing 9.18: Der erweiterte TLD

Entwickeln des neuen Renderers

Die Renderer-Klasse entspricht weitestgehend der Vorgehensweise aus dem Rendering-Quellcode der ersten Kreditkarten-Variante. Kleinere Unterschiede existieren in den Parametern der Methoden selbst, da im Falle einer separaten Renderer-Klasse die Komponente jeweils mitübergeben werden muss.

Methode in Renderer-Klasse	Methode in Komponentenklasse
`encodeBegin( FacesContext context, UIComponent component )`	`encodeBegin( FacesContext context )`

Tabelle 9.1: Gegenüberstellung von Renderer- und Komponentenklasse

Methode in Renderer-Klasse	Methode in Komponentenklasse
`encodeChildren( FacesContext context, UIComponent component )`	`encodeChildren( FacesContext context )`
`encodeEnd( FacesContext context, UIComponent component )`	`encodeEnd( FacesContext context )`

Tabelle 9.1: Gegenüberstellung von Renderer- und Komponentenklasse (Fortsetzung)

Des Weiteren sind zwei zusätzliche Methoden `writeMonthList` und `writeYearList` enthalten. Diese geben in einer Schleife die jeweiligen Auswahlwerte als einzelne Elemente einer Drop-Down-Liste aus. Eine Drop-Down-Liste wird in HTML mittels des `select`-Tags dargestellt, einzelne Elemente durch ein `li`-Tag. Im aktuell selektierten Tag ist für eine Vorbelegung ein Attribut `selected` einzusetzen. Dies erfolgt ebenfalls in den beiden Methoden für die Monats- und Jahresliste.

```
package com.edu.jsf.bsp.tag;

import java.io.IOException;
import java.util.Calendar;
import java.util.HashMap;
import java.util.Map;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.event.ValueChangeEvent;
import javax.faces.render.Renderer;

/*
 * Rendererklasse für die Usagebar-Komponente
 */
public class CreditCardRenderer extends Renderer {

  private int total_length = 100;

  /*
   * Decodierung
   */
  public void decode(FacesContext context, UIComponent component)
      throws NullPointerException {

    if (context == null) {
      throw new NullPointerException();
    } // if
```

```
UICreditCard cc_component = (UICreditCard) component;
Map requestMap =
  context.getExternalContext().getRequestParameterMap();
try {
  // Zunächst die Werte des Requests lokal
  // zwischenspeichern
  HashMap tempvalue = new HashMap();
  tempvalue.put( "number",
    requestMap.get(cc_component.getClientId(context)
      + "_number") );
  tempvalue.put( "check",
    requestMap.get(cc_component.getClientId(context)
      + "_control") );
  tempvalue.put( "month",
    requestMap.get(cc_component.getClientId(context)
      + "_month") );
  tempvalue.put( "year",
    requestMap.get(cc_component.getClientId(context)
      + "_year") );

  // Danach den Werte validieren
  String newValue;

  newValue =
    (String) requestMap.get(cc_component.getClientId(context)
      + "_month");
  Integer i = new Integer(newValue);
  // Wert wird als Integer nochmals in tempvalue
  // gesetzt, um folgend mit dem richtigen Datentyp
  // die equals-Prüfung zwecks valuechange zu machen.
  tempvalue.put( "month", i );

  newValue =
    (String) requestMap.get(cc_component.getClientId(context)
      + "_year");
  i = new Integer(newValue);
  tempvalue.put( "year", i );

  // Bei Werteänderung ein ValueChangeEvent erzeugen
  if ( !tempvalue.equals( cc_component.getValue()) ) {
    CreditCardData newCCData = new CreditCardData();
    cc_component.queueEvent(
      new ValueChangeEvent(cc_component,
        cc_component.getValue(), tempvalue) );
  } // if

  cc_component.setValue( tempvalue );
  cc_component.setValid( true );

} catch (Exception exc) {
  context.addMessage(cc_component.getClientId( context ),
```

```
        new FacesMessage() {
          public String getDetail() {
            return "Die Kreditkartendaten sind nicht korrekt.";
          }

          public Severity getSeverity() {
            return FacesMessage.SEVERITY_ERROR;
          }

          public String getSummary() {
            return "Die Kreditkartendaten sind nicht korrekt.";
          }
      });

      cc_component.setValid( false );
    } // catch
  }

  /*
   * Codierung
   */
  public void encodeBegin( FacesContext context,
    javax.faces.component.UIComponent component)
      throws java.io.IOException {
  }

  public void encodeChildren( FacesContext context,
    UIComponent component)
      throws java.io.IOException {
  }

  public void encodeEnd( FacesContext context,
      UIComponent component) throws java.io.IOException {
    UICreditCard cc_component = (UICreditCard) component;

    HashMap valueMap = (HashMap)(cc_component.getValue());
    // Rendering
    ResponseWriter writer = context.getResponseWriter();
    writer.write("<table border=\"0\">");
    writer.write("<tr>");
    writer.write("<td>Kartennummer:</td>");
    writer.write("<td><input type=\"text\" ");
    writer.write(
      " name=\"" + cc_component.getClientId(context)
      + "_number\"");
    writer.write(" value=\"" + valueMap.get("number")
      + "\" size=\"20\"></td>");
    writer.write("</tr><tr>");
    writer.write("<td>Kontrollnummer:</td>");
    writer.write("<td><input type=\"text\" ");
    writer.write(" name=\"" + cc_component.getClientId(context)
```

```
        + "_control\"");
      writer.write(" value=\"" + valueMap.get("check")
        + "\" size=\"4\"></td>");
      writer.write("</tr><tr>");
      writer.write("<td>Gültig bis:</td>");
      writer.write("<td><select ");
      writer.write(" name=\"" + cc_component.getClientId(context)
        + "_month\">");
      writeMonthList(writer,
        Integer.parseInt(valueMap.get("month").toString() ) );
      writer.write("</select>");
      writer.write("<select ");
      writer.write(" name=\"" + cc_component.getClientId(context)
        + "_year\">");
      writeYearList(writer,
        Integer.parseInt( valueMap.get("year").toString() ) );
      writer.write("</select>");
      writer.write("</td>");
      writer.write("</tr>");
      writer.write("</table>");
    }

    /*
     * erzeugt eine Monatsliste mit dem aktuellen
     * Monat aktiviert
     */
    private void writeMonthList(ResponseWriter writer, int curVal)
        throws IOException {
      String sSel = "";
      for (int c = 1; c <= 12; c++) {
        if (c == curVal)
          sSel = "selected";
        else
          sSel = "";
        writer.write("<option value=\"" + c + "\" "
          + sSel + ">" + c);
      } // for
    }

    /*
     * erzeugt eine Jahresliste mit dem aktuellen
     * Jahr aktiviert
     */
    private void writeYearList(ResponseWriter writer, int curVal)
        throws IOException {
      String sSel = "";
      int curYear = Calendar.getInstance().get(Calendar.YEAR);
      for (int c = curYear; c <= curYear + 6; c++) {
        if (c == curVal)
          sSel = "selected";
```

```
        else
          sSel = "";
        writer.write("<option value=\"" + c + "\" "
          + sSel + ">" + c);
      } // for
    }
  }
```

Listing 9.19: Rendering-Klasse für die Kreditkartenkomponente

In Listing 9.19 fällt ebenfalls auf, dass die Komponente in den Methodenaufrufen mit übergeben wird. Diese Komponente der Klasse `UIComponent` kann dann auf die eigentliche Kreditkartenkomponente mit

```
UICreditCard cc_component = (UICreditCard) component;
```

gecastet werden. Daraufhin kann der aktuelle Wert über die Komponente mittels `getValue` abgefragt werden. Der Rendering Quellcode selbst ist größtenteils identisch mit dem der Kreditkartenkomponente der ersten Variante. Es wird lediglich eine Drop-Down-Box für das Jahr und den Monat verwendet.

Damit die Komponente auch `ValueChange`-Events erzeugen kann, wird in der `decode`-Methode eine entsprechende Abfrage vorgenommen. Dabei wird überprüft, ob die neuen Werte den bislang gesetzten Werten entsprechen. Ist dem nicht der Fall, liegt eine Werteänderung vor und es wird ein Event erzeugt. Über den Aufruf

```
cc_component.queueEvent( new ValueChangeEvent(cc_component,
    cc_component.getValue(), tempvalue) );
```

wird ein entsprechendes `ValueChange`-Event an eine Komponente angehängt. Im Event selbst ist sowohl der ursprüngliche Wert als auch der neue Wert hinterlegt.

Eintragen des Renderers in der Anwendungskonfigurationsdatei

Da im aktuellen Beispiel der Renderer als separate Klasse aufgerufen wird, ist die Definition des Renderers in der Anwendungskonfigurationsdatei zu hinterlegen. Die Methode `getRendererType` der Tagklasse liefert bereits einen Bezeichner *CreditCard* zurück. Dieser Bezeichner wird in der Anwendungskonfigurationsdatei ebenfalls verwendet und einer Rendererklasse zugeordnet.

```
...
<renderer>
  <component-family>CreditCardArea</component-family>
  <renderer-type>CreditCard</renderer-type>
  <renderer-class>
    com.edu.jsf.bsp.tag.CreditCardRenderer
```

```
    </renderer-class>
  </renderer>
  ...
```

Listing 9.20: Erweiterte Anwendungskonfigurationsdatei

Anpassen der Komponentenklasse

Da der Rendering-Code in der Komponentenklasse beibehalten werden soll, der sozusagen das Standard-Rendering übernimmt, muss einzig eine Abfrage eingesetzt werden, die auf einen eventuellen ausgelagerten Renderer hin überprüft. Dies wird realisiert, indem am Beginn der `encode`- bzw. `decode`-Methode auf den Renderer-Typ abgefragt wird:

```
if (getRendererType() != null) {
  super.encodeEnd( context );
  return;
} // if
```

Listing 9.21: Abfrage auf einen separaten Renderer

Wie in Listing 9.21 zu sehen, wird durch den Aufruf `getRendererType` überprüft, ob ein separater Renderer hinterlegt ist. Wenn ja, wird dieser durch den Aufruf der Superklasse angezogen. Eine entsprechende Abfrage ist sowohl in der `encode`- wie auch in der `decode`-Methode ganz zu Anfang zu finden.

Liefert die Methode einen `null`-Wert zurück, bedeutet dies, dass kein separater Renderer hinterlegt ist und das Rendering in der Komponente selbst stattfindet.

Verwendung der Komponente mit neuem Renderer

Um die erweitere Komponente ansprechen zu können, wird eine neue Faces-Seite erzeugt, auf der die Komponente mit dem zusätzlichen Attribut `style` aufgerufen wird. Dies sollte bewirken, dass der separate Renderer verwendet und das erweitere Aussehen angezeigt wird.

Wird jetzt ein neuer Wert eingegeben, und das Formular mit der Kreditkartenkomponente abgeschickt, wird ein `ValueChange`-Event erzeugt. In der Konsole sollte eine entsprechende Ausgabe erscheinen.

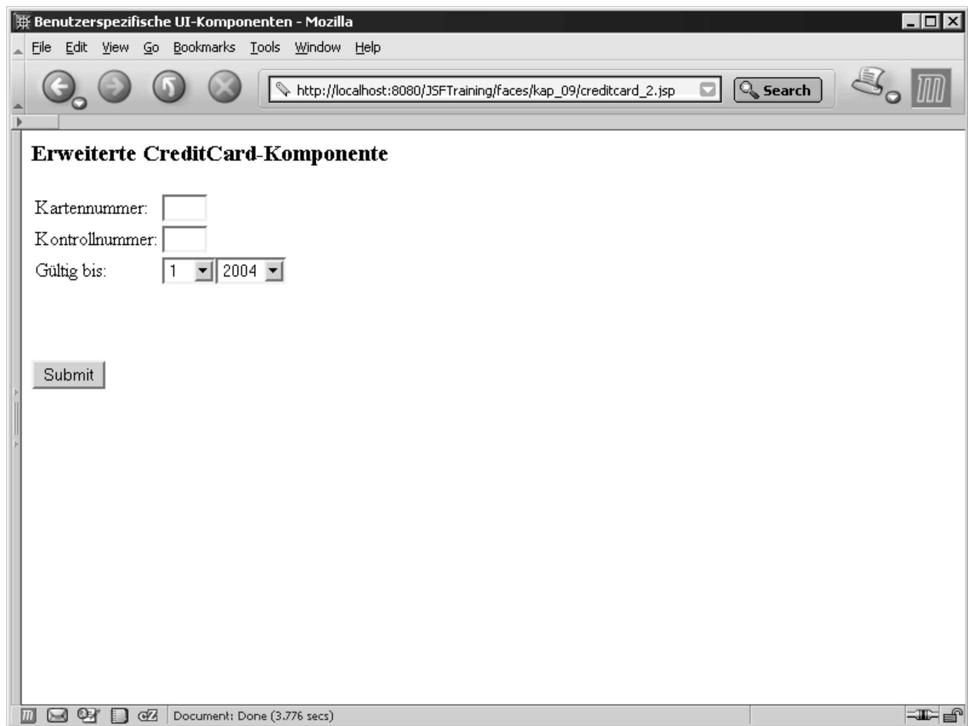


Abbildung 9.4: Kreditkartenkomponente mit verbesserter Jahres- und Monatsauswahl

9.2 Benutzerspezifische Validatoren

Validatoren dienen der Überprüfung von Eingabewerten. Im Kapitel über die Standardvalidatoren wurden bereits die häufigsten Validatoren erläutert, die in Webanwendungen regelmäßig auftreten. So kann mittels der Standardvalidatoren abgeprüft werden, ob ein eingegebener Wert eine bestimmte Länge aufweist oder auch innerhalb eines vorgegebenen Wertebereichs liegt. Die Standardvalidatoren in JSF sind bereits sehr mächtig und decken durch die Möglichkeit, diese auch in Kombination einsetzen zu können, einen Großteil der Feldprüfungen bereits ab.

Dennoch existiert die Notwendigkeit, bei Bedarf eigene Validatoren zu implementieren. Auch hier zeigt sich das JSF-Framework als sehr offen und erweiterungsfreundlich. Daher werden in den folgenden Abschnitten zwei benutzerspezifische Validatoren entwickelt. Der erste, eine einfache Überprüfung von E-Mail-Adressen, wird an ein Standard-Ausgabefeld angehängt. Beim zweiten Validator wird eine Kreditkartennummer auf ihre Gültigkeit hin überprüft. Dieser Validator baut auf der im vorherigen Kapitel entwickelten Kreditkartenkomponente auf.

Grundsätzlich existieren zwei Möglichkeiten, eine eigene Validierung zu realisieren. Zum einen kann eine eigene Validatorklasse geschrieben werden, in der die entsprechenden Prüfungen stattfinden. Zum anderen ist es bei Backing Beans auch möglich, eine einfache Validatormethode innerhalb eines Beans zu implementieren.

9.2.1 Überprüfung von E-Mail-Adressen

In vielen Formularen wird die Eingabe einer E-Mail-Adresse benötigt. Daher liegt die Anforderung nahe, eine spezielle Prüfung von Adressen durchzuführen. Die im folgenden Beispiel gezeigte Lösung ist relativ einfach umzusetzen. Es wird getestet, ob ein @-Zeichen vorhanden ist und vor diesem Zeichen mindestens ein Buchstabe steht. Weiterhin muss ein Punkt nach dem @-Zeichen stehen. Diese Prüfung ist mit Sicherheit nicht vollständig und könnte noch um weitere Prüfungen erweitert werden. Es wird jedoch deutlich, wie einfach benutzerspezifische Validatoren in eine Anwendung eingebaut werden können.

Für die Entwicklung eines eigenen Validators sind folgende Entwicklungsschritte notwendig:

- Erzeugen der Validator-Klasse, die das validator-Interface implementiert
- Registrieren von Fehlermeldungen in der Anwendungskonfigurationsdatei
- Bekanntmachen des neuen Validators
- Verwenden des Validators in einer JSF-Seite

In diesem ersten Beispiel wird die Möglichkeit vorgestellt, eine eigene Validatorklasse zu erzeugen und darin die erforderlichen Prüfungen abzuhandeln.

Erzeugen der Validator-Klasse

Eine Validator-Klasse muss zwingend das Interface `javax.faces.validator.Validator` implementieren. Dieses definiert eine Methode `validate`, in der die eigentliche Überprüfung stattfindet. Weist das Validator-Tag zusätzliche Attribute auf, müssen entsprechende getter- und setter-Methoden zusätzlich bereitgestellt werden. In Fall der Mailüberprüfung werden keine zusätzlichen Attribute im Validator-Tag benötigt, daher existiert in der Klasse `MailValidator` ausschließlich die `validate`-Methode.

```
package com.edu.jsf.bsp.validate;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.component.UIOutput;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
```

```
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

/*
 * Benutzerspezifischer Validator für Mailadressen
 */
public class MailValidator implements Validator {

  public MailValidator() {
  }

  /*
   * führt die Validierung durch
   */
  public void validate(FacesContext context,
      UIComponent component, Object checkValue )
        throws ValidatorException {
    boolean isvalid = true;
    if ((context == null) || (component == null)) {
      throw new NullPointerException();
    } // if
    if (!(component instanceof UIOutput)) {
      return;
    } // if

    if ( checkValue==null || checkValue.equals("") ) return;
    String value = checkValue.toString();

    if ( value.length()<5 ) isvalid = false;
    int at = value.indexOf("@");
    if ( at<=0 ) isvalid = false;
    int dot = value.lastIndexOf(".");
    if ( dot<0 || (value.length()-dot>4) ) isvalid = false;

    if ( isvalid==true ) {
      ((UIInput)component).setValid(true);
    } else {
      ((UIInput)component).setValid(false);
      FacesMessage errMsg =
        MessageFactory.getMessage( context, "mailValidate" );
      throw new ValidatorException( errMsg );
    } // else
  }
}
```

Listing 9.22: Validator-Klasse zur Mailüberprüfung

Mittels der Prüfung

```
(component instanceof UIOutput)
```

wird sichergestellt, dass die Mailvalidierung ausschließlich für Texteingabefelder verwendet wird. Sollte der Validator z.B. an einer Checkbox anhängen, wird keine Validierung durchgeführt.

Im Parameter `checkValue` ist der Wert hinterlegt, den es zu validieren gilt. Im Falle, dass dieser `null` ist oder eine Länge von 0 hat, erfolgt keine Prüfung, die Eingabe wird als valide gewertet (frei nach dem Motto: nichts eingegeben ist zumindest nicht falsch).

Anschließend erfolgt eine einfache Zeichenkettenprüfung, in der einige markante Eigenschaften einer Mailadresse abgefragt werden. An dieser Stelle ist mit Sicherheit viel Raum für weitere Prüfungen gegeben. Sollte aufgrund der Prüfungen festgestellt werden, dass die Eingabe nicht valide ist, wird die Komponente mit

```
component.setValid( false )
```

in einen invaliden Zustand gesetzt. Zusätzlich wird eine Meldung erzeugt, die auf eine fehlerhafte Eingabe hinweist. Für das Anziehen eines Textes für die Fehlermeldung wird eine Methode `getMessage` angezogen. Diese ist in der Klasse `MessageFactory` zu finden. Diese Klasse ist jedoch keine offizielle Klasse von JSF, wird jedoch in der Referenzimplementierung mit ausgeliefert. Die erzeugte Meldung wird dann als Parameter einer `ValidatorException` übergeben, die im Fehlerfalle in der Methode geworfen wird.

Registrieren von Fehlermeldungen

Bereits im Kapitel über die Standardvalidatoren wurde gezeigt, wie Fehlermeldungen überschrieben werden können. Für eigene, neue Fehlermeldungen wird die gleiche Vorgehensweise verwendet. Es kann in der Anwendungskonfigurationsdatei eine Ressourcendatei hinterlegt werden, die bei Programmstart eingelesen wird. Auf diese kann dann im Fehlerfalle zugegriffen werden.

```
...
<application>
  <message-bundle>mymessages</message-bundle>
</application>
...
```

Listing 9.23: Auszug aus der faces-config.xml

Wichtig ist, dass die im Tag `message-bundle` angegebene Ressourcendatei im Klassenpfad zu finden ist, z.B. direkt im `classes`-Unterverzeichnis des `WEB-INF`-Verzeichnisses. In der Datei ist dann die Fehlermeldung in der Form

```
key=value
```

zu hinterlegen. Beim Starten der Anwendung wird dieser Bereich der Anwendungskonfigurationsdatei ausgelesen und kann über die in der Validator-Klasse beschriebenen Methoden angezeigt werden.

Bekanntmachen des Validators

Als letzten Schritt muss dem Framework noch mitgeteilt werden, dass es einen benutzerspezifischen Validator gibt und wo dieser zu finden ist. Auch dieses erfolgt wiederum über die Anwendungskonfigurationsdatei *faces-config.xml*.

```
...
<validator>
  <description>Validator for Mailadresses</description>
  <validator-id>MailValidator</validator-id>
  <validator-class>
    com.edu.jsf.bsp.validate.MailValidator
  </validator-class>
</validator>
...
```

Listing 9.24: Registrieren des Validators

Wie in Listing 9.24 zu sehen, wird der Validator mit dem Bezeichner `MailValidator` eingebunden. Diesem zugeordnet ist die Klasse `com.edu.jsf.bsp.validate.MailValidator`. Der an dieser Stelle hinterlegte Bezeichner wird in der JSF-Seite später wiederverwendet werden, daher sollte der Bezeichnertext aussagekräftig gewählt werden.

Einbau des MailValidators

Nach den notwendigen Deklarationen sowie dem Implementieren der Validator-Klasse kann das `validator`-Tag an eine entsprechende Eingabekomponente angehängt und getestet werden. Dies geschieht mittels:

```
<h:inputText id="mail" value="">
  <f:validator validatorId="MailValidator" />
</h:inputText>
```

Es wird hierbei ein einfaches Eingabefeld erzeugt, an das zur Prüfung das Validator-Tag vom Typ `MailValidator` angehängt wird. Die Klasse für diesen Typ ist wiederum in der Anwendungskonfigurationsdatei hinterlegt und wird bei Bedarf angezogen. Bei korrekter Umsetzung erscheint im Fall einer Fehleingabe ein kurzer Hinweistext, wie in Abbildung 9.5 zu sehen ist.

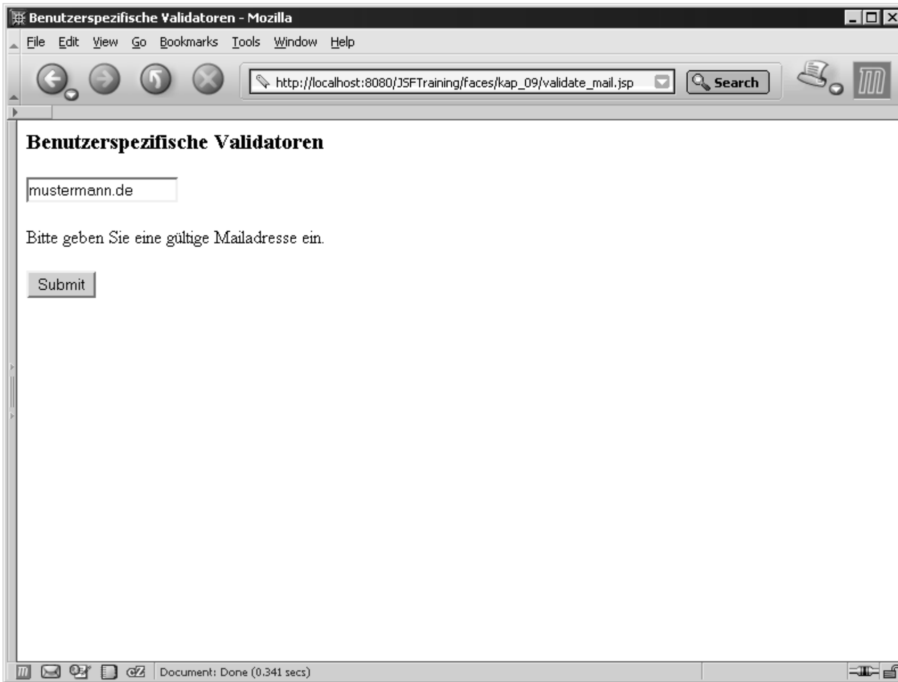


Abbildung 9.5: Der Mailvalidator in Aktion

Validierung bei Backing Beans

Statt eine eigene Validierungsklasse zu entwickeln ist es auch möglich, lediglich eine Validierungsmethode in ein Modellobjekt zu integrieren. Müssen dem Validator keine weiteren Parameter mitgegeben werden, ist dies eine sehr schnelle Möglichkeit, eine Validierungsroutine auf einem Eingabefeld durchzuführen. In Listing 9.25 wird davon ausgegangen, dass in der Anwendungskonfigurationsdatei ein Modellobjekt `Visitor` hinterlegt wurde.

```
<h:inputText value="#{Visitor.mail}"
   validator="#{Visitor.validateMail}" />
```

Listing 9.25: Aufruf einer Validierungsroutine

Anstelle eines verschachtelten Validator-Tags wird im `inputText`-Tag ein Attribut `validator` mitgegeben, in dem die entsprechende Validierungsroutine in JSF EL-Syntax angegeben ist.

```
public void validateMail(FacesContext context,
    UIComponent component, Object checkValue )
    throws ValidatorException {
```

```
    if ( checkValue==null ) return;

    String email = checkValue.toString();
    if (email.indexOf('@') == -1) {
      ((UIInput)component).setValid( false );
      FacesMessage errMsg =
        MessageFactory.getMessage( context, "mailValidate" );
          throw new ValidatorException( errMsg );
    } else {
      ((UIInput)component).setValid( true );
    } // else
  }
```

Listing 9.26: Validierungsroutine in der Klasse VisitorBean

Der Methode `validateMail` werden als Argument der `FacesContext` sowie die Komponente als Objekt der Klasse `UIInput` übergeben. Ebenfalls wird auch der zu prüfende Wert mit übergeben. In der Methode kann im Fehlerfalle die Komponente auf invalide gesetzt werden und eine entsprechende Fehlermeldung mit einer `ValidatorException` geworfen werden.

Vorteil dieses Verfahrens, eine Validierung durchzuführen, ist es, dass keine separate Validatorklasse erzeugt, sondern lediglich eine Methode gemäß einer vorgegebenen Signatur in einem Backing Bean implementiert werden muss. Sollte es jedoch notwendig sein, einem Validator zusätzliche Attribute übergeben zu können (in diesem Beispiel wäre es denkbar, dass nur de-Domains zulässig sind), so muss auf jeden Fall eine eigene Validatorklasse erzeugt werden.

9.2.2 Überprüfung von Kreditkartenangaben

In den bisherigen Beispielen war es ausreichend, einen Validator als Klasse bzw. als Methode aufzurufen. Die Validierung erfolgte daraufhin direkt auf den eingegebenen Werten des zugrunde liegenden Eingabefeldes. Doch es kann durchaus vorkommen, dass einem Validator zusätzlich weitere Parameter zur Prüfung mitzugeben sind. Dies wurde im Fall der Standardvalidatoren z.B. beim LongRange-Validator gezeigt, dem sowohl ein Minimalwert als auch ein Maximalwert mitgegeben werden konnten.

Bei Verwendung benutzerdefinierter Validatoren ist es natürlich auch möglich, zusätzliche Attribute für die Validierung mitzugeben. Dies wird im folgenden Beispiel demonstriert.

Bei vielen eCommerce-Seiten im Internet ist die Bezahlung stets ein aktuelles Thema. Oftmals ist es aus Kostengründen nicht möglich, eine Bezahlung direkt online während des Bezahlvorgangs durchzuführen. Vielmehr werden die Kreditkartenangaben gespeichert und in einem täglichen oder wöchentlichen Turnus vom Anbieter belastet. Stellt sich zu diesem Zeitpunkt erst heraus, dass eine Kreditkartennummer ungültig

ist, ist die bestellte Ware meist schon verschickt. Daher ist es wichtig, bereits bei der Eingabe von Kreditkartenangaben durch den Kunden zu erkennen, ob die Daten gültig sind oder nicht. Das Schöne an Kreditkartendaten ist, dass die Nummern nicht willkürlich entstanden sind, sondern einem mathematischen Algorhithmus folgen. Bei der Erstellung eines Validators für die Kreditkartenkomponente wird daher das so genannte *Luhn-Verfahren* zur Überprüfung der Kreditkartennummer angewendet. Um diesen Validator an die bestehende Komponente anhängen zu können, müssen an der Komponente selbst noch kleinere Änderungen vorgenommen werden.

Die Luhn-Formel

Die Luhn-Formel ist eine mathematische Formel, mit deren Hilfe die Echtheit von Kreditkartennummern geprüft werden kann. Der Algorithmus testet, ob die letzte Ziffer der Kreditkartennummer zu den übrigen angegebenen Nummern passt. Natürlich ist es mit diesem Verfahren möglich, eigene Kreditkartennummern »herzustellen«. Da dieses Verfahren doch schon seit Ende der 60er Jahre veröffentlicht ist, wird an dieser Stelle auch kein Geheimnis verraten.

Des Weiteren ist an der Kreditkartennummer selbst zu erkennen, welcher Hersteller sich hinter einer Nummer verbirgt. So fangen Kreditkartennummern von American Express beispielsweise mit den Ziffern 34 oder 37 an; bei Visa mit einer 4. Somit ist es möglich, auch nur bestimmte Kreditkarten in einer Bezahlfunktion zuzulassen, z.B. wenn die Gebühren für das Inkasso unangemessen hoch sind. Es wird daher die Überprüfung der Kreditkartenangaben dahingehend erweitert, dass im Validator-Tag eine Liste mit Herstellern mitübergeben werden kann, für die eine Eingabe zulässig ist.

Hersteller	Anfang	Gesamtlänge der Nummer
Visa	4	13 oder 16
American Express	34, 37	15
Mastercard	51, 52, 53, 54, 55	16
Diner's Club	30, 36, 38	14

Tabelle 9.2: Kreditkartenhersteller und ihre Kennungen

Anforderung an die Kreditkartenprüfung ist es daher, im Validator-Tag eine beliebige Anzahl an Herstellern in einer Komma-separierten Liste übergeben zu können, die als Eingabe zulässig sind. Da aber das Standard-Validator-Tag keine benutzerspezifischen Attribute zulässt, muss für diese Anforderung ein eigenes Validator-Tag festgelegt werden.

Erzeugen der Validator-Klasse

Der prinzipielle Aufbau der Validator-Klasse ist analog zur Validator-Klasse der Prozentanzeige-Komponente. Die Klasse selbst implementiert das `Validator`-Interface und muss dadurch eine Methode `validate` bereitstellen. Hierin erfolgt die Validierung der Nummer anhand des Luhn-Verfahrens. Dieses ist in einer separaten Methode `luhnTest` ausgelagert. Des Weiteren werden der Monat und das Jahr der Gültigkeit überprüft. Eine Prüfung der Kontrollzahl findet nicht statt.

Da die Möglichkeit geschaffen werden soll, nur bestimmte Kreditkartenanbieter zuzulassen, muss dem Validator auch eine Liste von erlaubten Kreditkartenanbietern übergeben werden. Dazu dienen die setter- und getter-Methode für die Eigenschaft `institutes`. Für die Zulässigkeit einer bestimmten Kreditkartennummer existiert eine separate Prüfmethode. Darin werden anhand des Beginns der Kreditkartennummer die erlaubten Anfangsziffern verglichen. Diese Vorgehensweise ist zwar nicht 100%ig korrekt, es müsste zusätzlich auch die Länge der Nummer überprüft werden, sie reicht jedoch für einen ersten Kurztest vollkommen aus.

Schlägt eine der Prüfungen fehl, wird die gesamte Komponente in einen invaliden Zustand versetzt und die einzelnen Fehlermeldungen über eine Exception dem Benutzer zur Anzeige gebracht. Für die Fehleranzeige wurde in diesem Fall ein Verfahren gewählt, bei dem einer Fehlermeldung zusätzliche Parameter hinzugefügt werden können. Dieser weitere Parameter wird im Aufruf `getMessage` mit übergeben.

Achtung:

Der Validator hat im Gegensatz zum Mailvalidator eine Eigenschaft `institutes`. Diese speichert die zulässigen Kreditkartenarten ab. Da in JSF jede View zwischen den Requests gespeichert wird, werden natürlich in einer View auch die darin enthaltenen Validatoren zusammen mit den Komponenten abgespeichert. Im Kapitel (6.11 Zustandsspeicherung) über die Zustandsspeicherung wurde bereits erläutert, dass es einen grundsätzlichen Unterschied zwischen serverseitiger und clientseitiger Zustandsspeicherung gibt.

Das aktuelle Kreditkartenbeispiel funktioniert nur, wenn eine serverseitige Zustandsspeicherung aktiv ist. Grund dafür ist, dass die notwendigen Methoden für das Speichern und Wiederherstellen des Zustandes nicht in der Validatorklasse implementiert sind. Da bei serverseitiger Speicherung im Normalfall (d.h. keine Clusterumgebung) die View als Objekt in der Session vorhanden bleibt, hat dies keine Auswirkung. Bei clientseitiger Speicherung wird die View jedoch zwangsläufig serialisiert und verliert damit die Eigenschaften des Validators, da dieser seinen Zustand nicht korrekt abspeichert.

Dies ist eine schlechte Programmierarbeit. In »real-live«-Anwendungen sollte daher immer darauf geachtet werden, zum Zwecke der Zustandsspeicherung die notwendigen Methoden bereitzustellen. Dies wäre daher auch im aktuellen Fall des Kreditkartenvalidators ratsam.

```
package com.edu.jsf.bsp.validate;

import java.util.HashMap;
import java.util.StringTokenizer;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

import com.edu.jsf.bsp.tag.UICreditCard;
import com.sun.faces.util.MessageFactory;

/*
 * Benutzerspezifischer Validator für Mailadressen
 */
public class CreditCardValidator implements Validator {

  private String institutes;

  public CreditCardValidator() {
  }

  /*
   * Setter- und Getter-Methode für die
   * Eigenschaft institutes
   */
  public String getInstitutes() {
    return institutes;
  }

  public void setInstitutes(String string) {
    institutes = string;
  }

  /*
   * führt die eigentliche Validierung durch
   */
  public void validate(FacesContext context,
      UIComponent component, Object checkValue )
      throws ValidatorException {
    boolean ccnumber_isvalid = true;
    boolean ccmonth_isvalid = true;
    boolean ccyear_isvalid = true;
    boolean ccinstitutes_isvalid = true;

    if ((context == null) || (component == null)) {
      throw new NullPointerException();
    } // if
```

```
if (!(component instanceof UICreditCard)) {
  return;
} // if
UICreditCard creditCardComp = (UICreditCard)component;

String value = ((HashMap)checkValue).get("number").toString();

// 1. Prüfung der Nummer
ccnumber_isvalid = luhnTest( value );

// 2. Prüfung des Jahres
int year = Integer.parseInt(((HashMap)checkValue)
  .get("year").toString());
ccyear_isvalid = (year>1950 && year<=2010);
// 3. Prüfung des Monats
int month = Integer.parseInt(((HashMap)checkValue)
  .get("month").toString());
ccmonth_isvalid = ( month>0 && month <=12 );
// 4. Prüfung auf Institutionen
ccinstitutes_isvalid = testInstitutions( value );

if ( ccnumber_isvalid && ccyear_isvalid
    && ccmonth_isvalid && ccinstitutes_isvalid ) {
  creditCardComp.setValid( true );
} else {
  creditCardComp.setValid( false );

  if ( ccnumber_isvalid==false ) {
    FacesMessage errMsg =
      MessageFactory.getMessage( context, "creditcardValidate",
        "Kreditkartennummer" );
      throw new ValidatorException( errMsg );
  } // if
  if ( ccyear_isvalid==false ) {
    FacesMessage errMsg =
      MessageFactory.getMessage( context,
      "creditcardValidate", "Gültigkeit bis (Jahr)" );
    throw new ValidatorException( errMsg );
  } // if
  if ( ccmonth_isvalid==false ) {
    FacesMessage errMsg =
      MessageFactory.getMessage( context,
      "creditcardValidate", "Gültigkeit bis (Monat)" );
    throw new ValidatorException( errMsg );
  } // if
  if ( ccinstitutes_isvalid==false ) {
    FacesMessage errMsg =
      MessageFactory.getMessage( context,
      "creditcardValidate", "Kreditkarten-Art" );
    throw new ValidatorException( errMsg );
  } // if
```

```
  } // else
}

/*
 * Prüfung einer Kreditkartennummer nach dem Luhn-Verfahren
 */
private boolean luhnTest( String number ) {
  int len = number.length();
  int digits[] = new int[len];
  int digit = 0;

  try {
    for ( int i=0; i<len; i++ )
      digits[i] = Integer.parseInt( number.substring(i,i+1) );

    int sum = 0;
    int factor = 1;

    for (int i = 0; i < len; i++) {

      digit = digits[len-i-1];
      digit *= factor;
      if (digit >= 10)
        sum += (digit % 10) + 1;
      else
        sum += digit;
      if (factor == 1)
        factor++;
      else
        factor--;
    }
    if ((sum % 10) == 0)
      return true;
    else
      return false;

  } catch (Exception exc) {
    return false;
  } // catch
}

/*
 * Prüfung auf einen bestimmten Kreditkartenhersteller
 */
private boolean testInstitutions( String ccnumber ) {
  StringTokenizer st = new StringTokenizer( institutes, "," );
  String cur = "";
  while ( st.hasMoreElements() ) {
    cur = st.nextElement().toString();
    if ( ccnumber.startsWith( cur ) )
```

```
        return true;
      } // while
      return false;
    }
  }
```

Listing 9.27: Die Validator-Klasse für Kreditkartenangaben

Die Liste der Kreditkartenhersteller, die über die Eigenschaft `institutes` übergeben werden, ist eine Komma-separierte Zeichenkette. Diese Zeichenkette wird über einen `StringTokenizer` in die einzelnen Elemente zerlegt und in einer `while`-Schleife darüber iteriert.

Registrieren von Fehlermeldungen

Bei der Kreditkartenüberprüfung wird ausschließlich eine einzige Fehlermeldung verwendet, die durch einen Parameter ergänzt werden kann. Der Platzhalter wird in der Ressourcendatei durch eine geschwungene Klammer `{}` und eine darin enthaltene Nummer dargestellt. So kann eine Meldung durch nahezu beliebig viele Parameter ergänzt werden.

Bekanntmachen des Validators

Der Validator wird wiederum in der Anwendungskonfigurationsdatei bekannt gemacht. Als Validator-Id wird *CreditCardValidator* verwendet. Dieser Bezeichner wird in der Tag-Klasse durch den Aufruf

```
super.setValidatorId ("CreditCardValidator")
```

gesetzt. Somit kann das Framework eine Zuordnung zwischen dem Bezeichner und der Tag-Klasse herstellen.

```
...
<validator>
  <description>Validator for Credit Cards</description>
  <validator-id>CreditCardValidator</validator-id>
  <validator-class>
    com.edu.jsf.bsp.validate.CreditCardValidator
  </validator-class>
</validator>
...
```

Listing 9.28: Bekanntmachen des Validators in der faces-config.xml

Tag-Handler und Tag-Bibliothek

Bei der Entwicklung eines benutzerspezifischen Validators ohne zusätzliche Attribute konnte das bereits in JSF definierte `validator`-Tag verwendet werden. Da jedoch für den Kreditkartenvalidator ein Attribut mit übergeben werden muss, welches eine Liste mit erlaubten Kreditkartenarten enthält, muss hierfür ein eigenes Tag und damit auch eine Tag-Klasse bereitgestellt werden.

```
...
<tag>
  <name>creditCardValidator</name>
  <tag-class>
    com.edu.jsf.bsp.tag.CreditCardValidatorTag
  </tag-class>
  <attribute>
    <name>institutes</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
...
```

Listing 9.29: Definition des Kreditkarten-Validator-Tags

In Listing 9.29 ist ein Auszug aus dem Tag-Deskriptor zu sehen. Es wird dabei ein Tag definiert, das über den Namen `creditCardValidator` angesprochen wird und eine dazugehörige Tag-Klasse `com.edu.jsf.bsp.tag.CreditCardValidatorTag` besitzt. Zusätzlich hat dieses Tag ein Attribut `institutes`, das zwingend anzugeben ist. Die Definition selbst kann in den bereits vorhandenen TLD eingefügt werden oder aber in eine neue Datei. In obigem Beispiel wird wiederum die für die benutzerdefinierten Komponenten vorhandene *custom-jsf.tld* verwendet.

Nachdem das Tag definiert wurde, muss die dazu passende Tag-Klasse entwickelt werden. Wichtig ist hierbei, dass diese von der Klasse `ValidatorTag` ableitet. Die Klasse `ValidatorTag` ist die Basisklasse für alle benutzerspezifischen Validatoren. Sie ist verantwortlich dafür, dass der passende Validator zur richtigen Zeit instanziert wird und für die Komponente registriert ist.

```
package com.edu.jsf.bsp.tag;

import javax.faces.validator.Validator;
import javax.faces.webapp.ValidatorTag;
import javax.servlet.jsp.JspException;

import com.edu.jsf.bsp.validate.CreditCardValidator;

/**
```

```
 * Tag-Klasse für die UsageBar-Komponente.
 * Diese zeigt einen Verbrauchsbalken in Abhängigkeit
 * der übergebenen Prozentzahl.
 */
public class CreditCardValidatorTag extends ValidatorTag {

  private String institutes;

  /**
   * Konstruktor
   */
  public CreditCardValidatorTag() {
    super();
    super.setValidatorId("CreditCardValidator");
  }

  /**
   * erzeugt den Validator und setzt Institutes
   * in den Validator ein
   */
  protected Validator createValidator() throws JspException {
    CreditCardValidator result = null;
    result = (CreditCardValidator) super.createValidator();
    result.setInstitutes( institutes );

    return result;
  }

  /**
   * Setter- und Getter-Methode für die Kreditkartenarten
   */
  public String getInstitutes() {
    return institutes;
  }

  public void setInstitutes(String string) {
    institutes = string;
  }
}
```

Listing 9.30: Tag-Klasse für den Validator

Wichtig bei der Entwicklung der Tag-Klasse ist, dass im Konstruktor durch den Aufruf

```
super.setValidatorId("CreditCardValidator")
```

der korrekte Bezeichner gesetzt wird, der auch schon in der Anwendungskonfigurationsdatei verwendet wurde. In der Methode `createValidator` werden der eigentliche Validator erzeugt und die Liste der erlaubten Kreditkartentypen übergeben.

Einbau des Validators

Der Einbau des Kreditkarten-Validators ist wiederum relativ unspektakulär. Es muss jedoch darauf geachtet werden, dass die entsprechende Tag-Bibliothek in der JSF-Seite eingebunden wird, da ansonsten das Validator-Tag nicht erkannt wird.

```
<cu:creditCard value="#{CreditCardHolder.ccdata}">
  <cu:creditCardValidator institutes="12,13,14,37" />
</cu:creditCard>
```

Listing 9.31: Einbau des Kreditkarten-Validators

Das Validator-Tag wird als Tag zwischen die Start- und Ende-Tags der Kreditkartenkomponente gesetzt. Eine eventuelle Fehlerausgabe erfolgt wieder an der dafür vorgesehenen Stelle, die durch die Angabe von `<h:messages />` bezeichnet wurde.

Erweiterung der Kreditkartenkomponente

Die Kreditkartenkomponente ist eine benutzerdefinierte Komponente. Das Verhalten zur Codierung und Decodierung sowie die Validierung und ein eventueller Modellupdate wurden neu implementiert. Damit ein benutzerdefinierter oder ein Standardvalidator von der Komponente überhaupt aufgerufen werden, muss in der `decode`-Methode der Komponentenklasse noch eine Erweiterung eingebaut werden. Dabei wird eine Liste von allen an der Komponente angehängten Validatoren erzeugt und die entsprechende `validate`-Methode aufgerufen.

```
Validator[] validators = super.getValidators();
for ( int i=0; i<validators.length; i++ ) {
  Validator validator = (Validator) (validators[i]);
  validator.validate(context, this, localvalue);
} // for
```

Listing 9.32: Erweiterung der decode-Methode

Da die Kreditkartenkomponente von `UIInput` ableitet, stellt diese Klasse bereits die notwendigen Methoden bereit, mit denen sich Validatoren an der Komponente registrieren können. Über den Aufruf `getValidators` wird ein `Array` zurückgeliefert, in dem alle relevanten Validatoren enthalten sind. Von diesen wird in einer Schleife die jeweilige `validate`-Methode aufgerufen.

9.3 Benutzerspezifische Konverter

In klassischen Webanwendungen liegen die Benutzereingaben in Form von Stringwerten vor. Daten werden üblicherweise in Eingabefeldern in HTML-Masken eingegeben. Sobald die Daten durch den Benutzer (meist durch einen Submit-Button) abgeschickt

werden, müssen die Daten aus der Präsentationssicht in das zugrunde liegende Datenmodell übernommen werden. Hier ist meist eine Datenkonvertierung notwendig. Beispielsweise wird das Alter einer Person in Jahren in einem Texteingabefeld an der Oberfläche eingegeben. Im Datenmodell ist die Jahreszahl meist ein Zahlenwert.

JSF beinhaltet bereits eine große Anzahl an Standardkonvertern, die einem Entwickler viel Arbeit abnehmen können. Es kann jedoch auch der Fall auftreten, dass die Standardkonvertierungen nicht ausreichen und somit eigene benutzerspezifische Konverter entwickelt werden müssen.

Im folgenden Beispiel wird eine Eingabemaske für Adressangaben eines Kunden verwendet. Ein Kunde kann übliche Daten wie Name und Adresse eingeben. Er hat jedoch auch die Möglichkeit, eine E-Mail-Adresse einzugeben. Da heutzutage die meisten Menschen bereits mehrere E-Mail-Adressen besitzen (eine geschäftliche, eine private, eine für die Ehefrau, eine für die Freundin, ...), soll die Möglichkeit geschaffen werden, dass diese beliebig vielen Adressen alle hinterlegt werden können. Von der Darstellung soll ein einfaches Textfeld gewählt werden, in dem die einzelnen Adressen durch ein Komma getrennt eingegeben werden können. Diese kommaseparierte Liste soll beim Einlesen der Daten in einem `Vector` des Datenmodells gespeichert werden. Umgekehrt muss aus dem `Vector` bei der Anzeige auch wieder eine Komma-separierte Liste erstellt werden können.

An dieser Stelle greifen keine Standardkonverter mehr. Es muss ein eigener Konverter entwickelt werden, der aus einer Komma-separierten Liste einen `Vector` erzeugt und umgekehrt.

Vorgehensweise

Die Entwicklung eines Konverters ist im Vergleich zur Entwicklung einer kompletten benutzerdefinierten Komponente sehr einfach. Es sind im Wesentlichen drei Arbeitsschritte:

- Erstellen der Konverterklasse, die das Interface `Converter` implementiert
- Bekanntmachen des Konverters in der Konfigurationsdatei
- Einbau des Konverters in der Faces-Seite

Erstellen der Konverterklasse

Die Konverterklasse ist dafür verantwortlich, aus einem übergebenen String den im Modell verwendeten korrekten Datentyp zu erstellen. Ebenfalls muss aus einem Wert eines Modells eine Stringdarstellung erzeugt werden. Eine Konverterklasse muss daher das Interface `javax.faces.convert.Converter` implementieren. Dieses Interface definiert genau zwei Methoden:

- `public Object getAsObject( FacesContext context, UIComponent component, String value )`: für die Konvertierung eines übergebenen Stringwertes aus der Benutzersicht in das korrekte Datenobjekt
- `public String getAsString( FacesContext context, UIComponent component, Object value )`: für die Konvertierung eines Datenobjektes in eine Stringdarstellung

Diese beiden Methoden regeln die Konvertierung in beide Richtungen. Weitere eventuell benötigte Hilfsmethoden können natürlich in der Klasse dazu eingebaut werden.

```
package com.edu.jsf.bsp.convert;

import java.util.Enumeration;
import java.util.StringTokenizer;
import java.util.Vector;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;

/*
 * Custom converter für Mailadressen
 */
public class MailListConverter implements Converter {

  public Object getAsObject( FacesContext context,
      UIComponent component, String value ) {
    Vector v = new Vector();
    if ( value!=null && value.length()>0 ) {
      StringTokenizer st = new StringTokenizer( value, "," );
      String mail;
      while ( st.hasMoreElements() ) {
        mail = st.nextElement().toString();
        v.addElement( mail );
      } // while
    } // if
    return v;
  }

  public String getAsString( FacesContext context,
      UIComponent component, Object value) {

    String ret = "";
    if ( value!=null && value instanceof Vector ) {
      Vector v = (Vector)value;
      for ( Enumeration enum = v.elements();
        enum.hasMoreElements(); ) {
          ret += "," + enum.nextElement().toString().trim();
      } // for
    } // if
```

```
      // Anfangskomma abschneiden
      if ( ret.length()>0 )
        ret = ret.substring( 2 );

      return ret;
    }
  }
```

Listing 9.33: Der Maillisten-Konverter

Für die Konvertierung in ein Datenobjekt, im konkreten Fall in einen `Vector`, wird zunächst überprüft, ob der übergebene Wert eine Länge größer 0 aufweist. Ist dies der Fall, wird der String mit Hilfe des `StringTokenizers` zerlegt und in einer `while`-Schleife Element für Element durchlaufen. Jedes Element wird dabei einzeln einem `Vector` zugefügt. Dieser (temporäre) `Vector` wird nach Durchlaufen aller Elemente zurückgeliefert.

Im umgekehrten Fall wird in der Methode `getAsString` der Vektor selbst durchlaufen und somit der resultierende Stringwert aufgebaut. Aufgrund der Vorgehensweise ist es notwendig, am Ende des Schleifendurchlaufs die ersten zwei Zeichen mit dem Komma zu löschen, um eine korrekte Darstellung zu bekommen.

Bekanntmachen des Konverters

Um den neuen Konverter für die Anwendung verfügbar zu machen, muss dieser zunächst wieder dem Framework und somit auch der Anwendung bekannt gemacht werden. Dies geschieht durch einen Eintrag in der Anwendungskonfigurationsdatei:

```
<converter>
  <description>
    Konvertiert eine Mailliste in einen Vector
  </description>
  <converter-id>maillist</converter-id>
  <converter-class>
    com.edu.jsf.bsp.convert.MailListConverter
  </converter-class>
</converter>
```

Listing 9.34: Eintrag des Konverters in der Anwendungskonfigurationsdatei

Neben einem Beschreibungstext wird dem Konverter ein eindeutiger Bezeichner `maillist` zugewiesen sowie die dazugehörige Konverterklasse vollqualifizierend angegeben.

Einbau des Konverters

Um den Konverter in einer Anwendung nutzen zu können, genügt es, im `converter`-Attribut eines Texteingabe-Tags den Konverter anzugeben. Dieser wird dann automatisch aufgerufen und die Eingabedaten entsprechend konvertiert.

```
<h:inputText value="#{Adress.mails}" converter="maillist" />
```

Listing 9.35: Verwendung des Konverters

Dies war dann auch schon alles. Jetzt kann der Konverter eingesetzt werden. Falls eine korrekte Eingabe vorgenommen wurde, werden die Werte des Feldes Mailadressen künftig in einen Vektor des Datenmodells überführt.

9.4 Fazit

JavaServer Faces ist nicht nur ein Framework mit umfangreichen Tagbibliotheken, sondern bietet einem Anwendungsentwickler auch sehr viel Raum für eigene Erweiterungen. So wurde in diesem Kapitel gezeigt, wie benutzerdefinierte Komponenten erstellt und in eine Anwendung integriert werden können. Des Weiteren können benutzerdefinierte Komponenten mit Standardrenderern oder auch eigenen benutzerdefinierten Renderern verwendet werden. Ebenso ist es auch möglich, vorhandene Standardkomponenten mit eigenen Renderern darzustellen.

Ebenso wurde demonstriert, wie benutzerdefinierte Konverter erstellt werden können. Konverter dienen dazu, Daten zwischen der Präsentationssicht (das, was der Benutzer z.B. im Browser sieht) und der Modellsicht zu konvertieren und umgekehrt. Dazu existieren bereits eine gewisse Anzahl an Standardkonvertern, die um weitere ergänzt werden können, wo Sonderfunktionen benötigt werden.

Zur Sicherstellung einer korrekten Benutzereingabe dienen Validatoren. Mit Validatoren kann eine Benutzereingabe z.B. auf eine korrekte Länge oder einen korrekten Datentyp überprüft werden. Für umfangreichere Prüfungen können auch an dieser Stelle benutzerdefinierte Validatoren eingebracht werden.

Insgesamt gesehen bietet JavaServer Faces somit eine Vielzahl an Möglichkeiten, eigene Erweiterungen mit einzubringen und somit das Framework an eigene projektspezifische Bedürfnisse anpassen zu können.

10 Beispiele aus der Praxis – Little Best Practices

Kapitelziel

Die Syntax einer Technologie zu lernen ist eine Sache, diese auch richtig anzuwenden eine andere. Deswegen hat sich im Laufe der Zeit der Begriff der *Best Practices* etabliert. Unter Best Practices versteht man erprobte und bewährte Vorgehensweisen beim Einsatz einer Technologie. Best Practices werden von keinem Komitee gekürt und auch von keiner Firma offiziell herausgegeben. Vielmehr entwickeln sich Best Practices im Laufe der Zeit in der Entwicklergemeinde automatisch. Nützliche Erfahrungen werden weitergegeben und in künftigen Projekten wieder verwendet. Ziel dieses Kapitels ist es daher, Ihnen erste praktische Tipps sowie Empfehlungen an die Hand zu geben, wie Anwendungen mit JavaServer Faces umgesetzt werden können.

Ein Buch bzw. ein Kapitel zum Thema *Best Practices* erscheint im Normalfall erst einige Zeit, nachdem sich eine Technik in der Praxis bewährt hat und demzufolge sich gewisse Routinen und Vorgehensweisen bereits etablieren konnten. Jetzt ist JSF noch recht neu, und dennoch existiert bereits ein Kapitel Best Practices in diesem Buch – wenn auch mit dem Zusatz »Little«. Dieses Kapitel beruht nicht darauf, JSF-Beispiele aus jahrelanger Erfahrung darzustellen, sondern soll Hinweise und vor allem auch praktische Tipps geben, wie Sie elegant häufig auftretende Problemstellungen mit JSF-Anwendungen anpacken können. Eine Vielzahl der Beispiele beruht ebenfalls auf zahlreichen Diskussionen in einschlägigen Internet-Foren, in denen bereits seit Erscheinen der ersten Early-Access-Version eifrig über Lösungsansätze und Konzepte diskutiert wurde.

Ein Teil der im Folgenden geschilderten Hinweise beruht nicht ausschließlich auf JSF, sondern gilt übergreifend auch für den allgemeinen Entwicklungsvorgang. Für Sie als Einsteiger oder Neuling (allgemein in der Programmierung oder speziell in JavaServer Faces) soll dieses Kapitel Tipps und Anregungen geben, Ihre Entwicklungsergebnisse eventuell ein wenig zu verbessern oder ausbauen zu können.

10.1 Bereitstellen von Managed-Beans

Eine häufig auftretende Aufgabenstellung bei Webanwendungen ist es, dass Daten für eine Seitenanzeige bereitgestellt werden müssen, bevor irgendeine Aktion stattgefunden hat. Ein klassisches Beispiel ist eine Seite mit aktuellen Nachrichten. Auf diese Seite wird von anderen Adressen verlinkt. Ein Benutzer wird somit direkt auf die betreffende Seite geleitet. Nun ist JavaServer Faces in gewisser Weise reaktiv, was bedeutet, dass zunächst einmal eine Aktion stattfinden muss, in der eventuell Beans befüllt und für eine folgende Seitenanzeige bereitgestellt werden können.

Eine ähnliche Aufgabenstellung war in der Beispielanwendung JSF-WebLog gegeben. Es sollte eine Liste mit vorhandenen WebLogs angezeigt werden. Die Lösung in diesem Fall war, eine Startseite vorzuschalten, auf der ein Link zur eigentlichen »richtigen« Startseite eingesetzt war. Betätigte ein Benutzer diesen Link, wurde eine Aktion angestoßen, in der die notwendigen Beans bereitgestellt werden konnten. Doch nicht immer kann eine Seite vorgeschaltet werden, oftmals muss direkt auf die betreffende Seite verwiesen werden.

Dieser Fall war im WebLog-Beispiel bei der Realisierung des RSS-Newsfeeds gegeben. Es wurde direkt eine Xml-Datei angesprochen, die als Parameter den betreffenden WebLog-Bezeichner mitgegeben hat. Anhand dieses Bezeichners mussten dann die entsprechenden Beans befüllt werden. In diesem Fall verwendete man ein FrontController-Servlet, das den Request zunächst entgegennahm, die darin übergebenen Informationen auswertete und darauf aufbauend die notwendigen Beans bereitstellte. Danach wurde direkt auf die eigentliche Seite weiterverwiesen.

Im Beispiel des RSS-Newsfeeds bestand die Schwierigkeit darin, dass je nach Anfrage andere Bean-Informationen bereitgestellt werden mussten (unterschiedlich je nach angefragter WebLog-Id). Bezogen auf das Beispiel einer Meldungsanzeige aktueller Nachrichten verhält es sich jedoch so, dass für jeden Benutzer dieselben Informationen bereitgestellt werden. Daher kann in diesem Fall eine dritte Variante zur Bereitstellung von Beans verwendet werden. Diese Variante verwendet einen *ServletContextListener*.

Ein ServletContextListener ist nichts JSF-Spezifisches, sondern gehört in den Bereich der Servletprogrammierung. Ein *ServletContextListener* wird aufgerufen, sobald ein Servletkontext vollständig initialisiert wurde bzw. sobald der Kontext auch wieder entfernt wurde. Wird somit ein *ServletContextListener* an den Kontext der Faces-Anwendung angehängt, können somit durch den Listener Informationen (sprich die notwendigen Beans) bereitgestellt werden, sobald die Anwendung geladen wird. Damit stehen bereits vor dem ersten Request die notwendigen Beans in befülltem Zustand bereit.

```
package com.edu.jsf.bsp.listener;

import java.util.ArrayList;
```

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import com.edu.jsf.bsp.bean.NewsBean;
import com.edu.jsf.bsp.bean.NewslistBean;

/*
 * Vorfüllen einer Newsliste
 */
public class NewsServletContextListener implements ServletContextListener {

  /*
   * wird aufgerufen beim Initialisieren des Kontexts
   */
  public void contextInitialized(ServletContextEvent event) {
    ArrayList entrylist = new ArrayList();

    // Hier könnte ein Datenbankzugriff erfolgen.
    // Exemplarisch werden die Beans mit Testdaten
    // befüllt.
    NewsBean aNews = new NewsBean();
    aNews.setTitle("JavaServer Faces: Neues Tutorial online");
    aNews.setTeaser("Auf http://www.jsf-forum.de steht
                     ein neues Tutorial bereit.");
    aNews.setText("Auf den Seiten des deutschen JSF-Forums ...");
    entrylist.add( aNews );

    aNews = new NewsBean();
    aNews.setTitle("Final Release der Spezifikation");
    aNews.setTeaser("Soeben wurde das Final Release
                     der Spezifikation freigegeben.");
    aNews.setText("Mit heutigem Datum steht auf den Seiten ...");
    entrylist.add( aNews );

    aNews = new NewsBean();
    aNews.setTitle("Buch 'JavaServer Faces' erschienen");
    aNews.setTeaser("Neuerscheinung von Addison-Wesley zum
                     Thema JSF");
    aNews.setText("Fast zeitgleich mit Erscheinen der
                   Final Spezifikation ...");
    entrylist.add( aNews );

    NewslistBean nlbean = new NewslistBean();
    nlbean.setEntries( entrylist );

    event.getServletContext().setAttribute("Newslist", nlbean);
  }

  /*
   * wird aufgerufen beim Zerstören des Kontexts
```

```
     */
    public void contextDestroyed(ServletContextEvent event) {
    }
}
```

Listing 10.1: Verwendung eines ServletContextListeners

Listing 10.1 zeigt die Implementierung eines *ServletContextListener.* Das Interface sieht vor, dass zwei Methoden bereitgestellt werden müssen: `contextInitialized` und `contextDestroyed`, die jeweils nach der vollständigen Initialisierung des Kontextes bzw. nach dessen Entladen aktiviert werden.

Es werden zunächst mehrere Beans vom Typ `NewsBean` erzeugt und in Form einer `ArrayList` einem `NewsListBean` hinzugefügt. Entscheidend ist das Setzen des Beans in den ServletContext:

```
event.getServletContext().setAttribute("Newslist", nlbean);
```

Nur auf diese Weise ist es möglich, die Beans dem `VariableResolver` von JavaServer Faces zugänglich zu machen. Es ist an dieser Stelle noch nicht möglich, die Beans über den FacesContext bereitzustellen. Daher wird auch nicht die Schreibweise `#{...}` verwendet, sondern lediglich der eigentliche Bezeichner. Natürlich muss auch der Bezeichner (in diesem Fall *Newslist*) als Managed-Bean in der Anwendungskonfigurationsdatei eingetragen sein.

Um den *ServletContextListener* zu registrieren, muss im Deployment Deskriptor *web.xml* eine Eintragung vorgenommen werden:

```
<listener>
  <listener-class>
    com.edu.jsf.bsp.listener.NewsServletContextListener
  </listener-class>
</listener>
```

Listing 10.2: Registrieren des ServletContextListeners

Somit stellt die Verwendung eines *ServletContextListener* eine weitere mögliche Variante dar, Managed-Beans vor dem ersten Aufruf eines Faces-Seite bereitzustellen. Als Anwendungsentwickler muss somit die Entscheidung getroffen werden, welche Variante verwendet werden soll: Vorschalten einer Startseite, Einsatz eines FrontController-Servlets oder eines ServletContextListeners.

10.2 JSF und JSTL SQL – Eine Alternative?

Eine einfache Webanwendung kommt meist ohne eine dauerhafte Speicherung von Daten aus. Sollen in einer Anwendung lediglich einfache Berechnungen durchgeführt, kurze Abläufe demonstriert oder Benutzereingaben nicht weiter gespeichert werden, erfolgt eine Datenspeicherung meist in der Session, quasi im Hauptspeicher des Servers. Oftmals ist es jedoch wichtig, Benutzereingaben dauerhaft zu speichern oder aber auf Datenbestände einer Datenbank zugreifen zu müssen.

Für die Anbindung einer Datenbank in eine Webanwendung gibt es unterschiedlichste Möglichkeiten. »Die beste« Alternative existiert wie so häufig nicht, es kommt vielmehr immer auf die projektspezifischen Eigenheiten sowie auch auf Vorlieben und Erfahrungen der Entwickler und Softwarearchitekten an. In der Beispielanwendung *JSF-WebLog*, die in Kapitel ((8. Die Beispielapplikation JSF-WebLog)) erstellt wurde, ist eine Datenbankanbindung mittels des Open-Source-Werkzeugs Torque realisiert. Da JSF jedoch mit der JSTL verwandt ist, liegt es nahe, die JSTL SQL-Klassenbibliothek für Datenbankzugriffe zu verwenden. Über die JSTL-SQL-Befehle können Datenbankzugriffe sehr komfortabel ebenfalls mit speziellen Tags beschrieben und mit der *JSP Expression Language* ausgewertet werden. Dieses Konzept hat den Vorteil, dass sowohl für die Darstellung der Oberfläche selbst wie auch für die notwendige Datenbankabfrage Taglibs zum Einsatz kommen. Es wird somit eine durchgängige Technologie verwendet, technische Konflikte sind dadurch auf ein Minimum beschränkt.

Es soll jedoch gleich zu Anfang darauf hingewiesen werden, dass der Einsatz von JSTL SQL in der Entwicklergemeinde stark diskutiert wird. Der Einsatz von JSTL SQL bietet zwar einige Vorteile, jedoch auch einige (nicht unwichtige) Nachteile. Dennoch kann ein Einsatz dieser Technik in manchen Projekten durchaus sinnvoll sein.

Ziel soll es daher sein, Ihnen ein Praxisbeispiel zur Integration von JSF und JSTL SQL zu liefern und näher zu bringen. Im Anschluss an das Beispiel werden die Vor- und Nachteile dieses Ansatzes diskutiert. Auf dieser Basis können Sie entscheiden, ob der Einsatz dieser Vorgehensweise für Ihr Projekt sinnvoll erscheint oder nicht.

Im Beispiel wird eine einfache Seminarverwaltung realisiert. Dabei kann ein Benutzer aus einer Liste von Schulungsseminaren sich für ein Seminar durch Eingabe von seinem Namen und seiner Mailadresse anmelden. Die Liste der angebotenen Seminare liegt in einer Datenbank, die Speicherung der Seminaranmeldungen erfolgt ebenfalls in einer separaten Datenbanktabelle.

Die Anwendung besteht aus zwei JSF-Seiten. Auf der ersten Seite wird eine Liste mit allen in der Datenbank vorhandenen Seminarangeboten (Tabelle `course`) angezeigt sowie ein Formular zur Eingabe der Anmeldedaten. Nach Betätigen des Anmeldebuttons wird auf eine zweite Seite verzweigt, in der ein Datenbankeintrag erfolgt, sowie ein kurzer Bestätigungstext im Browser ausgegeben.

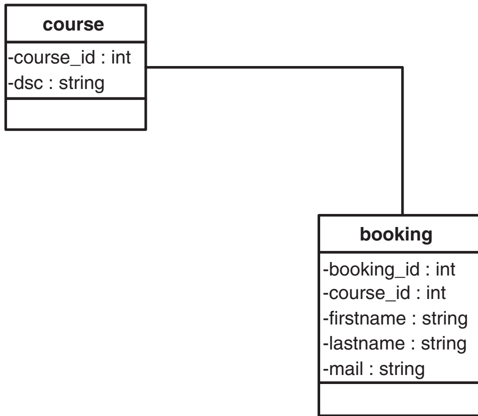


Abbildung 10.1: Schema für das JSF-JSTL SQL Beispiel

```
package com.edu.jsf.bsp.bean;

/**
 * Bean, das sämtliche Daten für eine
 * Kursanmeldung enthält
 */
public class RegisterBean {

  private String firstname;
  private String lastname;
  private String mail;
  private String event;

  /**
   * Getter-Methoden
   */
  public String getEvent() {
    return event;
  }

  public String getFirstname() {
    return firstname;
  }

  public String getLastname() {
    return lastname;
  }

  public String getMail() {
    return mail;
  }

  /**
```

```
     * Setter-Methoden
     */
    public void setEvent(String string) {
      event = string;
    }

    public void setFirstname(String string) {
      firstname = string;
    }

    public void setLastname(String string) {
      lastname = string;
    }

    public void setMail(String string) {
      mail = string;
    }
  }
```

Listing 10.3: Bean für die Anmeldedaten

Das in Listing 10.3 abgebildete Bean ist natürlich entsprechend in der Anwendungskonfigurationsdatei *faces-config.xml* zu hinterlegen. Da eine Navigation eingebaut wird (Wechsel von einer auf die nächse Seite), ist ebenfalls der Navigationsfall von der ersten zur zweiten Seite einzutragen.

```
<navigation-rule>
  <from-view-id>/kap_10/register.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/kap_10/register_thx.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 10.4: Auszug aus faces-config.xml

Der Vorteil bei Einsatz von JSTL SQL ist, dass ein Datenbankzugriff mit nur sehr wenigen Zeilen Quellcode zu realisieren ist. Es sind so gut wie keine Vorarbeiten notwendig, auch müssen keinerlei zusätzliche Klassen generiert oder spezielle Bibliotheken einbunden werden (die erforderlichen JSTL-Bibliotheken sind sowieso bereits integriert).

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
```

```
<html>
<head>
  <title>JSF und JSTL SQL</title>
</head>

<sql:setDataSource
  var="coursesDB"
  driver="org.gjt.mm.mysql.Driver"
  url="jdbc:mysql://localhost/courses"
  user="master"
  password="" />

<body>
<f:view>
  <h3>Zusammenspiel von JSF und JSTL SQL</h3>
  <i>Dieses Beispiel demonstriert das Zusammenspiel
  von JSF und der JSTL-SQL-Klassenbibliothek</i>

  <h:form>

    <u>Seminaranmeldung:</u><br><br>
    Folgende Seminare werden angeboten:

    <sql:transaction dataSource="${coursesDB}">
      <sql:query var="myresult">
        SELECT * FROM course
      </sql:query>
    </sql:transaction>
    <ul>
    <c:forEach items="${myresult.rows}" var="row">
      <li value="${row.course_id}">${row.dsc}</li>
    </c:forEach>
    </ul>

    Hiermit melde ich mich für folgendes Seminar an
    (Seminarnummer 1 - ...):
    <h:inputText value="#{CourseRegister.event}" size="5" />
    <br><br>

    <h:panelGrid columns="2">
      <h:outputText value="Vorname" />
      <h:inputText value="#{CourseRegister.firstname}" size="20" />
      <h:outputText value="Nachname" />
      <h:inputText value="#{CourseRegister.lastname}" size="20" />
      <h:outputText value="E-Mail" />
      <h:inputText value="#{CourseRegister.mail}" size="20" />
    </h:panelGrid>

    <br>
    <h:commandButton action="success" value="Anmelden" />
```

```
  </h:form>

</f:view>
</body>
</html>
```

Listing 10.5: Integration von JSF und JSTL SQL

Wichtig bei der Verwendung von JSTL SQL ist, dass über eine Taglib-Anweisung die erforderlichen Bibliotheken in die Seite eingebunden werden.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
```

Damit werden zusätzlich zu den von JSF benötigten Bibliotheken zwei weitere JSTL-Bibliotheken in eine Seite integriert.

Die Verbindung zur Datenbank wird über das Tag

```
<sql:setDataSource
  var="coursesDB"
  driver="org.gjt.mm.mysql.Driver"
  url="jdbc:mysql://localhost/courses"
  user="master"
  password="" />
```

realisiert. Wird auf mehreren JSF-Seiten eine Datenbankanbindung benötigt, empfiehlt es sich, diesen Teil in einer separaten JSP-Datei auszulagern und über `include`-Tags diese damit von zentraler Stelle in jeder Seite anzuziehen.

Im Anschluss wird über ein `<sql:query>`-Tag eine Datenbankabfrage gestartet. An dieser Stelle fällt sogleich (negativ) auf, dass SQL-Befehle direkt in der JSF-Seite enthalten sind. Dies ist bei dem gezeigten einfachen SQL-Befehl auch bei einem Wechsel auf eine andere Datenbank noch kein Problem, bei komplexeren Datenbankabfragen kann jedoch bei einem Datenbanksystemwechsel ein Anpassen aller SQL-Befehle notwendig sein. Dann müssten sämtliche JSF-Seiten auf enthaltene SQL-Befehle untersucht werden!

Die Ausgabe der Ergebniszeilen erfolgt mit Hilfe der Expression Language, die Teil der JSP-Spezifikation ist. Über

```
${row.course_id}
```

wird jedem Eintrag eine eindeutige Kennung zugewiesen.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
```

```
<html>
<head>
  <title>JSF und JSTL SQL</title>
</head>

<sql:setDataSource
  var="coursesDB"
  driver="org.gjt.mm.mysql.Driver"
  url="jdbc:mysql://localhost/courses"
  user="master"
  password="" />

<body>
<f:view>

  <h3>Die UIInput-Komponente</h3>
  <i>Diese Beispiel demonstriert das Zusammenspiel
  von JSF und der JSTL SQL-Klassenbibliothek</i>

  <br><br><br>
  <sql:transaction dataSource="${coursesDB}">
    <sql:update var="booking" sql="insert into booking
      (course_id, firstname, lastname, mail) values (?, ?, ?, ?)" >
      <sql:param value="${CourseRegister.event}" />
      <sql:param value="${CourseRegister.firstname}" />
      <sql:param value="${CourseRegister.lastname}" />
      <sql:param value="${CourseRegister.mail}" />
    </sql:update>
  </sql:transaction>

  Vielen Dank, wir haben Ihre Anmeldung erhalten.

</f:view>
</body>
</html>
```

Listing 10.6: Zweite Seite der Registrierung mit JSF und der JSTL SQL

Nachdem die Angaben der ersten Seite abgeschickt wurden, erfolgt gemäß der Navigation die Seite mit der Bestätigung der Anmeldung. Auf dieser Seite werden ebenfalls die JSTL-Bibliotheken eingebunden sowie die Datenquelle mit dem entsprechenden Tag angesprochen. Das Speichern der Eingaben erfolgt über ein `<sql:update>`-Tag, die Werte werden wieder mit der Expression Language aus der Session angezogen.

Zusammenspiel von JSF und der JSTL

In beiden Listings fällt auf, dass mittels der JSTL-Befehle auf JSF-Objekte zugegriffen werden kann. Dies ist jedoch nur bedingt möglich. Zwar kann über JSTL-Tags auf JSF-Beans zugegriffen werden (z. B. für eine Ausgabe), jedoch ist in der JSP-Spezifikation kein Managed-Bean-Konzept hinterlegt. Dies bedeutet, dass Beans nicht automatisch angelegt werden, sollte auf diese via JSTL zugegriffen werden. Im Gegensatz dazu regelt das Managed-Bean-Konzept in JSF, dass automatisch die benötigten Beans bereitstehen. In künftigen Versionen der JSP- und der JSF-Spezifikation wird es möglicherweise realisiert werden, dass eine bessere Integration gegeben ist. Daher ist die Kombination von JSF- und JSTL-Tags immer mit Vorsicht zu genießen.

Wichtig ist jedoch, dass im Falle eines Datenbankfehlers oder eines sonstigen Problems kein vernünftiges Fehlerbehandlungskonzept existiert. Es erfolgt keine Kontrolle, ob die Daten tatsächlich in die Datenbank geschrieben wurden, und auch kein automatisches Wiederholen der Aktion, wenn z. B. die Datenbank kurzfristig nicht verfügbar ist.

Fazit- SQL-Funktionalitäten stellt zunächst eine sehr einfache und in der Verwendung über Tag-Befehle auch elegante Möglichkeit dar, eine relationale Datenbank in eine JSF-Anwendung mit einzubinden. Jedoch wird zu Recht vor einem produktiven Einsatz gewarnt, da Fehlerkonzepte fehlen sowie von der Architektur betrachtet die Einbindung von SQL-Befehlen in der JSF-Seite selbst nicht einer sauberen Trennung von Funktionalitäten entspricht. Zumal ist es in den aktuellen Spezifikationen noch nicht möglich, eine wirkliche Integration von JSF- und JSTL-Tags zu erreichen.

Für eine einfache Prototypenentwicklung oder für die schnelle (»quick and dirty«) Entwicklung kleinerer Anwendungen stellt die JSTL-SQL-Bibliothek jedoch eine durchaus mögliche Alternative dar. Für den produktiven Betrieb oder eine professionelle Kundenanwendung sollte jedoch von einem Einsatz der JSTL-SQL-Taglib abgesehen werden.

10.3 Session oder Datenbank?

Das Konzept in JSF ist verführerisch. Modellobjekte werden in der Anwendungskonfigurationsdatei *faces-config.xml* einmal definiert und stehen dann während des gesamten Ablaufs der Webanwendung zur Verfügung. Daher liegt es oftmals nahe, alle notwendigen Objekte, auf die während einer Anwendungssitzung zugegriffen werden könnte, prophylaktisch in der Session abzulegen – unterstützt vom Managed-Bean-Konzept. Diese Vorgehensweise birgt jedoch etliche Gefahren in sich, auch wenn JSF bereits so schlau ist, Modellobjekte erst bei Bedarf zu instanzieren, anstatt alle Objekte gleich zu Beginn anzulegen (*lazy loading*).

Prinzip von Sessions

Sessions werden vom Servlet-Container bzw. JSP-Container bereitgestellt und bieten eine einfache Möglichkeit, temporär Daten und Objekte zu speichern. Sessions werden benutzerspezifisch angelegt und verwaltet. Startet ein Benutzer über seinen Browser eine Webanwendung, so wird ihm eine individuelle Session zugeteilt. Damit eine Session eines Benutzers, die auf dem Server gehalten wird, bei weiteren Requests genau dem gleichen Benutzer wieder zugewiesen werden kann, wird in der Regel ein Cookie im Browser des Benutzers gespeichert, der eine Id mit der zugewiesenen Session beinhaltet. Im Laufe der Anwendung können somit durch einen Benutzer Aktionen durchgeführt werden, die gewisse Informationen in der Session ablegen. Als Beispiel kann ein Online-Shop betrachtet werden. Die einzelnen Waren, die ein Benutzer in den Warenkorb ablegt, können in einer Session gespeichert werden.

Die Lebensdauer einer Session ist jedoch beschränkt. Da der Server und damit die Session nicht mitbekommt, wann ein Benutzer seinen Browser schließt, und damit die temporären Daten gelöscht werden können, wird eine Session automatisch nach einer bestimmten Zeit gelöscht, wenn keine Aktion mehr stattgefunden hat. Die Zeit, die vergehen muss, bis eine (nicht mehr verwendete) Session gelöscht wird, kann in den meisten Fällen in einem Applikationsserver hinterlegt werden. Bei der Festlegung der Timeout-Zeit ist jedoch viel Erfahrung notwendig. Wird die Zeit zu gering gewählt, kann es passieren, dass eine Session gelöscht wird, obwohl ein Benutzer noch in der Anwendung arbeitet und vielleicht nur eine spezielle Seite etwas länger betrachtet hat. Wird die Zeit zu groß gewählt, sammeln sich unter Umständen zu viele Sessions auf dem Server an und der Speicher stößt an seine Grenzen.

Was denn jetzt?

Aufgrund der oben genannten Limitierungen sollte somit stets darauf geachtet werden, dass Sessions möglichst klein gehalten werden. Wichtige oder umfangreiche Daten gehören in eine Datenbank und nicht in eine Session. Da es jedoch immer eine Ermessensfrage ist, welche Daten in der Session abgelegt und welche über die Datenbank angezogen werden, folgend ein Fallbeispiel einer Online-Community.

Als Beispiel dient eine Community zum Thema JSF. Es ist eine Webseite, auf der für registrierte und nicht registrierte Besucher Informationen angeboten werden sowie ein Diskussionsforum existiert. Besucher können sich auf der Plattform registrieren und anmelden und haben damit die Schreibberechtigung für das Forum. Nicht registrierte Besucher haben ausschließlich eine Leseberechtigung.

Bei der Anmeldung auf der Plattform sollte ausschließlich ein Datenbankzugriff in Aktion treten, der die eingegebenen Login-Daten mit dem Datenbestand vergleicht. Es könnte leicht die Serverdimensionen sprengen, wenn bei einer großen Community sämtliche Benutzerdaten in der Session vorgehalten werden. Sobald sich ein Benutzer

erfolgreich angemeldet hat, empfiehlt es sich, diese Information in der Session abzulegen, sprich mit welchem Namen er angemeldet ist und welchen Berechtigungsstatus er aufweist (u.U. gibt es auf der Plattform nochmals unterschiedliche Berechtigungsstufen). Das Forum sollte für die Anzeige der Beiträge immer aus der Datenbank generiert werden. Es macht keinen Sinn, diese Daten in der Session abzuspeichern. Sind mehrere Besucher online, wären sämtliche Forenbeiträge mehrfach in den Sessions abgespeichert. Für die Abfrage der Berechtigung, ob ein Besucher in das Forum schreiben darf oder nicht, genügt wiederum die Information in der Session, hier ist kein Datenbankzugriff notwendig.

Das geschilderte Szenario ist **eine** Sicht der Dinge. Es gibt jedoch auch Beispiele im Internet, die die Sessioninformationen bzw. die Datenbankinformationen genau anders verwalten. So gibt es zahlreiche Shops, bei denen der Warenkorbinhalt in einer Session vorgehalten wird. Dies ist zunächst einmal einleuchtend, sind doch Warenkorbinhalte bis zu einer endgültigen Bestellung transiente, also flüchtige Daten. Andererseits gibt es auch Shops, in denen ein Warenkorbinhalt in der Datenbank gespeichert wird (damit sind diese Daten persistent). Vorteil hiervon ist, dass der Kunde zwischendurch seinen Browser schließen kann und auch an einem anderen Computer seinen Einkauf fortsetzen könnte, ohne den Warenkorbinhalt zu verlieren.

Fazit

Wichtig in dem geschilderten Beispiel war es nicht, eine allgemeingültige Lösung zu präsentieren, die ohne Anpassung auf jede andere Webanwendung übertragbar ist. Vielmehr soll ein Gefühl dafür entstehen, wo bei der Entwicklung einer Anwendung mit JSF etwas länger nachgedacht werden sollte und wo eventuelle Fallstricke zu finden sind. Als guter Entwickler sollte man die Pros und Kontras sehr genau kennen, um die für das Projekt passende Lösung finden zu können.

10.4 Gültigkeitsbereich von Modellobjekten

Die Deklaration von Modellobjekten in der Anwendungskonfigurationsdatei ist mit Sicherheit ein großer Vorteil von JSF. Jedoch ist damit auch eine gewisse Gefahr verbunden. Bei komplexen Projekten ist es meist so, dass die Anzahl der Modellobjekte, die im Laufe einer Anwendung verwendet werden, sehr groß ist. Zudem sind die Objekte bzw. Klassen selbst bereits recht umfangreich. Da es aber leider bequem ist, alles was benötigt wird, einfach mal in die Session abzuspeichern, kann hier schnell ein Engpass auftreten. Benutzersessions werden nämlich im Applikationsserver verwaltet und abgelegt. Wird jetzt für jeden Benutzer einer zahlreich besuchten Webanwendung eine umfangreiche Session angelegt, belastet dies die Ressourcen des Applikationsservers in erheblichem Maße. Daher sollte nicht alles in die Session abgelegt werden, sondern nur Objekte, die tatsächlich über die gesamte Anwendungssitzung zur Verfügung stehen müssen.

Gültigkeitsbereich	**Beschreibung**
none	Ein Bean wird bei jeder Verwendung neu instanziert und danach wieder entfernt.
request	Das Bean steht nur während der Request-Verarbeitung zur Verfügung. Bei einem neuen Request wird dafür eine neue Instanz erzeugt.
session	Ein Bean wird einmalig für die Lebensdauer einer Session erzeugt und darin abgespeichert.
application	Das Bean steht wärend der gesamten Lebensdauer der Anwendung benutzerübergreifend zur Verfügung.

Tabelle 10.1: Gültigkeitsbereiche

Tabelle 10.1 zeigt noch einmal die möglichen Werte für den Gültigkeitsbereich in der Anwendungskonfigurationsdatei *faces-config.xml*. Oftmals reicht es auch aus, ein Bean nur während der Requestverarbeitung im Zugriff zu haben. Zentrale Objekte können auch im `application`-Gültigkeitsbereich angelegt werden. Hierbei stehen sie allen Benutzern automatisch zur Verfügung und müssen nicht in jeder Session redundant abgelegt werden.

Häufig genügt es jedoch, den Gültigkeitsbereich `request` zu wählen. Die benötigten Daten für eine Seitenanzeige werden in diesem Fall im Vorfeld (z. B. durch eine Datenbankabfrage) bereitgestellt und für den Lebenszyklus einer Request-Verarbeitung vorgehalten. Danach wird der Speicher wieder freigegeben. Bei einem erneuten Zugriff auf die Informationen werden diese neu erzeugt und wiederum zur Verfügung gestellt. Häufig kommt diese Vorgehensweise bei sehr dynamischen Webanwendungen vor, bei denen die Inhalte der Webseite oft verändert werden. Ein typisches Beispiel wäre ein Diskussionsforum. Die Liste der Beiträge kann sich sekündlich ändern. Daher wäre es eine denkbare Lösung, die Liste der Beiträge im Forum für jeden Request aus der Datenbank zu ermitteln und dem Request bereitzustellen.

Die Verwendung der Session wurde bereits im letzten Abschnitt näher erleutert. Sie dient der Speicherung benutzerbezogener Daten. Da für jeden Benutzer eine eigene Session angelegt wird, geht dies stark zu Lasten des Speichers auf dem Server. Daher ist genau zu prüfen, ob Daten tatsächlich in der Session abgelegt werden müssen und damit längerfristig vorgehalten werden oder ob nicht ein Gültigkeitsbereich lediglich im Request ausreichend ist.

10.5 Ressourcendateien

Durch die Möglichkeit, in UI-Komponenten auf Ressourcendateien zugreifen zu können, sind Anforderungen an eine mehrsprachige Anwendung sehr einfach zu realisieren. Das in Java bereits bewährte Konzept der Ressourcendateien wurde 1:1 in JSF übernommen. Der Mehraufwand, der sich bei einem Einsatz von Ressourcendateien ergibt, ist minimal,

wenn bereits von Beginn einens Projektes an darauf Wert gelegt wird. Aber auch in ein bestehendes (JSF-)Projekt können im Nachhinein Ressourcendateien eingebaut werden.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
  <title>JSF-Beispiel</title>
</head>

<f:loadBundle basename="txtBundle" var="txtbundle"/>
<f:loadBundle basename="msgBundle" var="msgbundle"/>

<body>
<f:view>
  <h3>Ressourcendateien</h3>
  <i>Dieses Beispiel demonstriert die Verwendung von
     ResourceBundles</i>
  <br><br>
  <h:form>

    <h:outputText value="#{txtbundle.txt}" />
    <br><br>
    <h:outputFormat value="#{msgbundle.msgInfo}">
      <f:param value="param" />
    </h:outputFormat>
    <br><br>
    <h:commandButton value="#{txtbundle.savebutton}" />
    <p>
    <h:messages />
  </h:form>

</f:view>
</body>
</html>
```

Listing 10.7: Verwendung von Ressourcendateien

In Listing 10.7 werden gleich zwei Ressourcendateien in eine JSF-Seite eingebunden. Das kann auch durchaus Sinn machen, wenn sich z.B. die Texte logisch in verschiedene Blöcke unterteilen lassen. Bevor Angaben in einer Resourcendatei in einer UI-Komponente verwendet werden können, müssen zunächst einmal sämtlichen Ressourcendateien mit der Anweisung

```
<f:loadBundle basename="msgBundle" var="msgbundle"/>
```

in die JSF-Seite eingebunden werden. Wichtig ist das Attribut `var`. Es regelt, über welchen Namen die Resourcendatei angesprochen werden kann. Die Ressourcendatei selbst ist nach dem Namensschema

```
???.properties
```

aufgebaut. Die Fragezeichen können durch einen beliebigen Namen ersetzt werden. Dieser Name taucht dann in genau gleicher Schreibweise im `loadBundle`-Tag wieder auf.

Die Ressourcendatei selbst ist eine einfache Textdatei, in der die einzelnen Bezeichner, Textblöcke oder Ausgaben untereinander angeordnet sind. Sollen mehrere Sprachvarianten einer Ressourcendatei vorgehalten werden, wird lediglich die Länderkennung (z.B. `de` für Deutschland) in den eigentlichen Dateinamen mit eingebaut. So wäre in letzterem Beispiel auch eine Datei *msgbundle_de.properties* denkbar. Dass die korrekte Sprachdatei angezogen wird, dafür sorgt Java selbst. Hierüber muss sich der Entwickler keine Gedanken machen.

Sämtliche UI-Komponenten unterstützen die Verwendungen von Ressourcendateien. In Listing 10.7 wird dies anhand von Ausgabetexten, Ausgabemeldungen und einem Commandbutton demonstriert. Die Ressourcendateien selbst müssen im Klassenpfad der Anwendung zu finden sein, ansonsten können keine Texte angezogen werden. Die Ressourcendateien können in beliebig vielen Sprachvarianten vorhanden sein, es wird jeweils die nach den aktuellen Einstellungen korrekte Datei angezogen. Auch wenn eine Anwendung in einer ersten Ausbaustufe nicht mehrsprachig ausgerichtet sein muss, empfiehlt sich dennoch der Einsatz von Ressourcendateien. Der Mehraufwand ist minimal, der Gewinn jedoch, der bei mehrsprachigen Anwendungen erzielt werden kann (zeitlich und bezogen auf das Budget), ist enorm.

10.6 Anlegen einer index.html

Beim Lesen der Überschrift mag man vielleicht zuerst einmal etwas verwirrt den Kopf schütteln, was wohl ein Thema wie eine *index.html* in einem Kapitel für Best Practices zu suchen hat.

Dieses Thema ist jedoch in der Tat sehr wichtig. Nicht, um dadurch einen besseren Quellcode zu erlangen, sondern um mögliche Fehler auf Seiten der Benutzer auszuschließen. Aufgrund der Eigenheit von JSF, dass sämtliche Requests über ein Faces-Servlet geleitet werden müssen, passiert es daher recht häufig, dass JSF-Seiten über eine URL direkt angesprochen werden. Ein typisches Beispiel ist, wenn eine Webanwendung unter der URL `http://www.firma.de/shopanwendung` läuft. Technisch korrekt müsste der Aufruf jedoch `http://www.firma.de/shopanwendung/faces/start.jsp` lauten (wenn nicht etwa ein Prefix-Mapping angelegt ist). Um hier einen Fehler im Vorfeld auszuschließen, empfieht es sich, im Hauptverzeichnis der Webanwendung eine Datei *index.html* anzulegen, die auf die eigentliche JSF-Startseite verweist.

```
<html>
  <head>
  <meta http-equiv="Refresh"
        content="0;URL=/shopanwendung/faces/start.jsp">
  </head>
  <body>
  </body>
</html>
```

Listing 10.8: index.html zur Weiterleitung

Existieren in einer Webanwendung Unterverzeichnisse, die u.U. von außen auch direkt aufgerufen werden können, empfiehlt es sich auch hier mit index-Dateien zu arbeiten. Die Angabe im Header-Bereich einer Html-Seite bewirkt, dass sofort nach dem Laden der html-Seite auf die *start.jsp*-Seite weitergeleitet wird. Die Zahl `0` besagt, dass eine Wartezeit von 0 Sekunden gewartet werden soll, bevor die Weiterleitung aktiv wird.

Eine weitere Möglichkeit, Fehler auszuschließen, die aufgrund eines falschen URL-Aufrufes entstanden sind, ist es, das Mapping auf das Faces-Servlet umzubauen. Grundsätzlich existieren zwei Möglichkeiten, Faces-Requests abzufangen. Zum einen mittels eines *Prefix-Mappings*, indem durch die Angabe von */faces* vor der eigentlichen Seite damit angezeigt wird, dass dies ein Faces-Request darstellt.

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

Eine andere Alternative ist die Verwendung des so genannten *Extension Mappings*. Dabei wird eine Endung definiert, die immer auf das Faces-Servlet verweist.

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

Aber auch bei dieser Variante kann es passieren, dass einzelne JSF-Seiten falsch angesprochen werden. Ein Ausschluss aller (fehlerhaften) Möglichkeiten, die ein Benutzer bei der Angabe von Adressen machen kann, ist zwar nicht möglich, aber bei Verwendung von index-Dateien wird zumindest ein Großteil von möglichen Fehlern abgefangen.

10.7 Steuerung der Komponentensichtbarkeit

Eine sehr elegante Eigenschaft von JSF ist es, einzelne Komponenten wahlweise ein- und ausblenden zu können. So können Sie erreichen, dass auf einer komplexen Seite nur bestimmte Komponenten dargestellt werden. Damit kann eine intuitive Benutzer-

führung realisiert werden, da nur solche Elemente sichtbar sind, die für einen momentanen Arbeitsschritt von Bedeutung sind. Sicherlich kann dieses Resultat auch erzielt werden, wenn eine Folge von Seiten als eine Art Wizard nacheinander geschalten wird. Statt einer einzigen Seite, in der je nach Bedarf einzelne Komponenten ein- und ausgeblendet werden, existieren in diesem Falle eben eine Vielzahl von Einzelseiten, die je nach Situation angezeigt werden.

Wie so häufig gibt es auch hier keine generelle Empfehlung, welcher Weg der bessere ist. Es hängt vielmehr von den projektspezifischen Einzelheiten ab, ob man eine Vielzahl an Einzelseiten oder eine steuerbare Gesamtseite realisiert. Im folgenden Beispiel wird die Steuerung der Sichtbarkeit anhand einer Tabelle mit darin enthaltenen Ausgabefeldern demonstriert.

Im folgenden Beispiel handelt es sich um die Anzeigeseite einer Teilnehmerliste. So kann in einer Anwendung z. B. für Sportveranstaltungen eine Teilnehmerliste betrachtet werden. Die Teilnehmerliste stellt standardmäßig eine komplette Teilnehmerbeschreibung dar. Es werden neben dem Vor- und Nachnamen auch die kompletten Adressdaten angezeigt. Es soll nun eine Möglichkeit geschaffen werden, die Ansicht derart anzupassen, dass lediglich die Namen der Teilnehmer angezeigt werden.

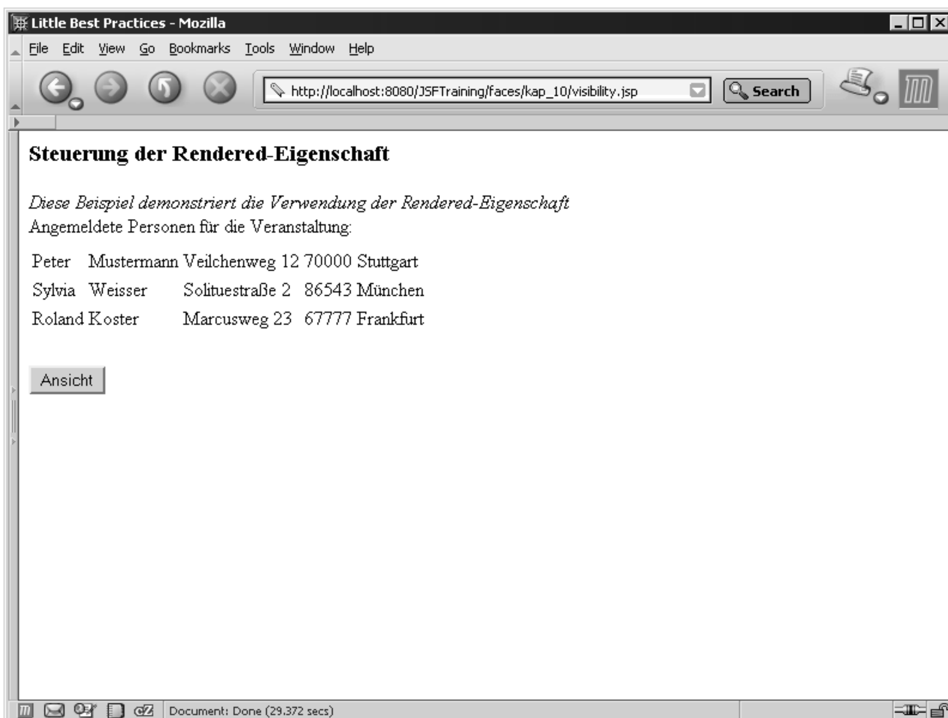


Abbildung 10.2: Anzeige aller Adressspalten

In Abbildung 10.2 ist die standardmäßige Darstellung zu sehen. Nach Umschalten mittels des `Ansicht`-Knopfes soll folgende Darstellung erscheinen:

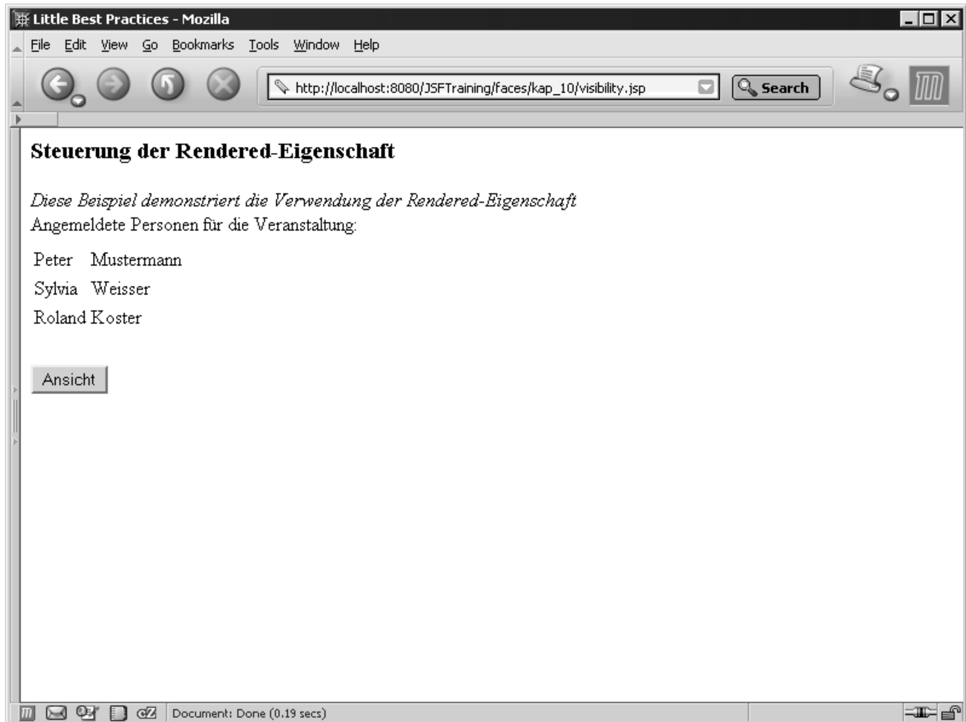


Abbildung 10.3: Komprimierte Ansicht

Für die Speicherung der Daten wird ein simples Bean angezogen, das natürlich auch wieder in der Anwendungskonfigurationsdatei hinterlegt ist.

```
package com.edu.jsf.bsp.bean;

import java.util.ArrayList;

/*
 * Bean zur Darstellung einer Teilnehmerliste
 */
public class ParticipantListBean {

  private ArrayList participants;


  public ParticipantListBean() {
    participants = new ArrayList();
```

```
    PersonBean aPerson = new PersonBean();
    aPerson.setFirstname("Peter");
    aPerson.setLastname("Mustermann");
    aPerson.setStreet("Veilchenweg 12");
    aPerson.setZip( 70000 );
    aPerson.setCity("Stuttgart");
    aPerson.setMale( true );
    participants.add( aPerson );

    aPerson = new PersonBean();
    aPerson.setFirstname("Sylvia");
    aPerson.setLastname("Weisser");
    aPerson.setStreet("Solituestraße 2");
    aPerson.setZip( 86543 );
    aPerson.setCity("München");
    aPerson.setMale( false );
    participants.add( aPerson );

    aPerson = new PersonBean();
    aPerson.setFirstname("Roland");
    aPerson.setLastname("Koster");
    aPerson.setStreet("Marcusweg 23");
    aPerson.setZip( 67777 );
    aPerson.setCity("Frankfurt");
    aPerson.setMale( true );
    participants.add( aPerson );
  }

  public ArrayList getParticipants() {
    return participants;
  }

  public void setParticipants(ArrayList list) {
    participants = list;
  }
}
```

Listing 10.9: Bean zur Speicherung von Personendaten

Im Bean `ParticipantList` wird lediglich eine `ArrayList` mit Objekten der Klasse `PersonBean` befüllt. Das Befüllen ist direkt im Konstruktor hinterlegt, was für diesen Fall für Demonstrationszwecke vollkommen ausreichend ist.

Da in beiden Abbildungen (mit und ohne eingeblendeten Adressspalten) die gleiche Faces-Seite verwendet wird, ist in dieser einen Datei die ausführlichere Variante mit allen Spalten zu entwickeln. Das Ausblenden der anderen Spalten wird später durch einen *ActionListener* angestoßen.

Die zugrunde liegende JSF-Seite sieht wie folgt aus:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head>
  <title>Little Best Practices</title>
</head>

<body>
<f:view>
  <h3>Steuerung der Rendered-Eigenschaft</h3>
  <i>Dieses Beispiel demonstriert die Verwendung
     der Rendered-Eigenschaft</i>

  <h:form id="participantForm">

    Angemeldete Personen für die Veranstaltung:<br>

    <h:dataTable id="tableParticipants"
       value="#{ParticipantList.participants}" var="participant">
      <h:column>
        <h:outputText value="#{participant.firstname}" />
      </h:column>
      <h:column>
        <h:outputText value="#{participant.lastname}" />
      </h:column>
      <h:column id="colStreet">
        <h:outputText value="#{participant.street}" />
      </h:column>
      <h:column id="colZip">
        <h:outputText value="#{participant.zip}" />
      </h:column>
      <h:column id="colCity">
        <h:outputText value="#{participant.city}" />
      </h:column>
    </h:dataTable>

    <br>

    <h:commandButton id="btnChange" value="Ansicht">
      <f:actionListener
       type="com.edu.jsf.bsp.listener.VisibilityListener" />
    </h:commandButton>

  </h:form>

</f:view>
</body>
</html>
```

Listing 10.10: Anzeige von Teilnehmerdaten

Wichtig bei der Erstellung der Faces-Seite ist die Angabe von Ids für die relevanten Komponenten. An den Button, der für das Wechseln der Ansicht zuständig ist, wird ein Listener angehängt, in dessen Methode der Wechsel der Ansicht stattfinden soll. Prinzipiell wäre es natürlich genauso möglich gewesen, eine `Action`-Methode nach Drücken des Buttons aufzurufen.

```
package com.edu.jsf.bsp.listener;

import java.util.Iterator;

import javax.faces.FactoryFinder;
import javax.faces.application.Application;
import javax.faces.application.ApplicationFactory;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

/**
 * Implementierung eines ActionListeners
 */
public class VisibilityListener implements ActionListener {

  private static boolean showAll = true;

  /**
   * wird bei Auslösen eines Events aufgerufen
   */
  public void processAction(ActionEvent event)
      throws AbortProcessingException {

    ApplicationFactory factory =
      (ApplicationFactory) FactoryFinder.getFactory(
      FactoryFinder.APPLICATION_FACTORY);
    Application application = factory.getApplication();
    ValueBinding binding = null;
    FacesContext context = FacesContext.getCurrentInstance();

    showAll = showAll==true ? false : true;
    toggleFields( showAll );
  }

  /**
   * schaltet die Sichtbarkeit der Komponenten
   * wahlweise um
   */
  private void toggleFields( boolean bShowLong ) {
    FacesContext context = FacesContext.getCurrentInstance();

    UIComponent formComponent = null;
```

```
        UIComponent tableComponent = null;
        UIComponent component = null;

        formComponent =
          context.getViewRoot().findComponent("participantForm");
        tableComponent =
          formComponent.findComponent( "tableParticipants" );

        // Spalte Straße
        component = tableComponent.findComponent("colStreet");
        component.setRendered(bShowLong);
        // untergeordnete Elemente verändern
        Iterator itr = component.getChildren().iterator();
        while (itr.hasNext()) {
          UIComponent child = (UIComponent) itr.next();
          child.setRendered(bShowLong);
        } // while

        // Spalte PLZ
        component = tableComponent.findComponent("colZip");
        component.setRendered(bShowLong);
        itr = component.getChildren().iterator();
        while (itr.hasNext()) {
          UIComponent child = (UIComponent) itr.next();
          child.setRendered(bShowLong);
        } // while

        // Spalte Stadt
        component = tableComponent.findComponent("colCity");
        component.setRendered(bShowLong);
        itr = component.getChildren().iterator();
        while (itr.hasNext()) {
          UIComponent child = (UIComponent) itr.next();
          child.setRendered(bShowLong);
        } // while

        context.renderResponse();
      }
    }
```

Listing 10.11: Die ActionListener-Klasse

Die ActionListener-Klasse ist so aufgebaut, dass in der Klasse eine statische Variable `showAll` vorgehalten wird, in der der aktuelle Anzeigestatus festgehalten wird. Hierbei ist zu beachten, dass diese Deklaration nicht thread-safe ist, doch sollte für dieses Beispiel die Vorgehensweise genügen. In der eigentlich interessanten Methode `toggleFields` wird somit ein Boolean-Wert übergeben, der die zusätzlichen Spalten entweder ein- oder ausschaltet. Über den Aufruf

```
context.getViewRoot().findComponent("participantForm");
```

erhält man als Ergebnis eine Komponente zurück. Der Bezeichner `participantForm` ist in der Faces-Seite in genau dieser Form hinterlegt. Von der `Form`-Komponente wiederum wird die `Table`-Komponente abgefragt und mittels der `Table`-Komponente wiederum kann direkt auf die einzelnen Spalten zugegriffen werden. Da eine Spaltenkomponente wiederum mehrere Kindelemente aufweist (u. a. das eigentliche Textausgabefeld), wird über sämtliche Kindelemente iteriert und durch den Aufruf

```
child.setRendered( bShowLong );
```

der Komponente erklärt, dass diese (nicht) gerendert werden soll. Dies erfolgt analog für jede der zusätzlichen Adressspalten. Am Ende der Verarbeitung wird mittels

```
context.renderResponse();
```

die Render Response-Phase aufgerufen, in der die übrigen Komponenten gerendert und damit in der Seite dargestellt werden.

10.8 Wohin mit der Business-Logik?

Für ein gutes Design einer Webanwendung gehört es ebenfalls dazu, auch die in einer Anwendung vorhandene Geschäftslogik an einer zenralen Stelle vorzuhalten. Die Geschäftslogik darf dabei auch nicht mit anderen Belangen wie der GUI vermischt werden. JSF ist von der Definition her ein Oberflächenframework, d. h. genau betrachtet, ist darin überhaupt nichts mit Geschäftslogik zu finden. Dies ist auch richtig so. Geschäftslogik hat nichts mit einer Oberflächenarchitektur zu tun. Es muss eine strikte Trennung erfolgen. Bei Einsatz von *Enterprise JavaBeans (EJB)* ist allein schon durch die Technologie eine Trennung vorgegeben. Ohne EJBs ist jedoch ebenfalls darauf zu achten, dass Geschäftslogik nicht mit der Oberflächenentwicklung vermischt wird.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head>
  <title>JavaServer Faces</title>
</head>
<body>
<f:view>
  <h3>Wohin mit der Business-Logik?</h3>
  <i>Dieses Beispiel zeigt, wie die Geschäftslogik
     in JSF-Anwendungen untergebracht werden kann:</i>

  <h:form>

    Erste Zahl: <h:inputText value="#{Calculate.numberOne}" /><br>
    Zweite Zahl: <h:inputText value="#{Calculate.numberTwo}" /><br>
```

```
      <br>
      Ergebnis: <h:outputText value="#{Calculate.result}" /><br>
      <br>
      <h:commandButton value="Berechnen"
         action="#{Calculate.calculate}" />

    </h:form>

  </f:view>
  </body>
  </html>
```

Listing 10.12: Ein einfaches Rechenbeispiel

In Listing 10.12 ist ein einfaches Formular abgebildet, in dem ein Benutzer zwei Zahlen eingeben kann. Der »Geschäftsprozess«, der nach Abschicken des Formulars angestoßen werden soll, ist der, dass beide Zahlen addiert werden und auf der Seite das Ergebnis wieder angezeigt wird. Diesen Prozess abzubilden ist mit Sicherheit eine leichte Übung. Dennoch soll von der Architektur bedacht werden, welches die bestmögliche Stelle ist, diese Geschäftslogik einzubauen. In Listing 10.12 wird durch den Commandbutton eine Aktionsmethode aufgerufen. Diese Methode ist jedoch in der Beanklasse selbst enthalten. Daher ist es nicht zu empfehlen, die eigentliche Berechnungslogik auch in der Beanklasse zu implementieren, sondern eine separate Klasse dafür bereitzustellen.

```
package com.edu.jsf.bsp.bean;

import com.edu.jsf.bsp.srv.ServiceFacade;

/**
 * Bean, das eine einfache Addition durchführt
 */
public class CalculateBean {

  private int numberOne;
  private int numberTwo;
  private int result;

  /**
   * Getter-Methoden
   */
  public int getNumberOne() {
    return numberOne;
  }

  public int getNumberTwo() {
    return numberTwo;
  }
```

```
  public int getResult() {
    return result;
  }

  public String calculate() {
    ServiceFacade service = ServiceFacade.getInstance();
    Integer calcRes = service.calculate( numberOne, numberTwo );
    result = calcRes.intValue();

    return null;
  }

  /**
   * Setter-Methoden
   */
  public void setNumberOne(int i) {
    numberOne = i;
  }

  public void setNumberTwo(int i) {
    numberTwo = i;
  }

  public void setResult(int i) {
    result = i;
  }
}
```

Listing 10.13: Das Bean zur Berechnung

In Listing 10.13 wird durch den Commandbutton eine Aktion ausgelöst. Die `Aktionsmethode` wiederum beinhaltet nicht die Geschäftslogik selbst, sondern ruft eine entsprechende *Service-Methode* einer Fassadenklasse auf. Statt dieses Aufrufes könnte an dieser Stelle auch ein EJB-Aufruf oder bei verteilten Systemen auch ein RMI- oder Corba-Aufruf erfolgen. Wichtig ist jedenfalls, dass die Geschäftslogik nach Möglichkeit nicht in der Beanklasse enthalten ist, da diese zur GUI gehört (oder gemäß dem MVC-Prinzip zum Modell) und damit nichts mit Geschäftslogik zu tun hat.

```
package com.edu.jsf.bsp.srv;

/**
 * ServiceFacade, in der Berechnungen
 * durchgeführt werden
 */
public class ServiceFacade {

  private static ServiceFacade singleton;
```

```
    private ServiceFacade() {
    }

    public static ServiceFacade getInstance() {
      if ( singleton==null )
        singleton = new ServiceFacade();

      return singleton;
    }

    public Integer calculate( int valueA, int valueB ) {
      return new Integer( valueA + valueB );
    }
  }
```

Listing 10.14: Die ServiceFacade-Klasse

In dieser Vorgehensweise sind gleich mehrere Design-Patterns zu finden. Zum einen ist die `ServiceFacade` selbst als *Singleton* aufgebaut, was bedeutet, dass anwendungsweit lediglich eine Instanz dieser Klasse existiert. Der Aufruf der Geschäftslogik über die `ServiceFacade` ist selbst wiederum das Facade-Pattern. Dieses besagt, dass Funktionalität ausgelagert wird, damit wie in diesem Beispiel nicht die gesamte Funktionalität der Berechnung (es könnten ja auch mehrere Rechenschritte sein) in der Aktionsmethode enthalten ist, sondern in einer separaten Klasse ausgelagert ist. Eine Fassadenklasse ist somit eine Art Schnittstelle, welche die Benutzung umfangreicherer Funktionalität vereinfacht, indem »vor« die umfangreiche Funktionalität eben eine Fassade gesetzt wird.

10.9 Aktions-Methode oder ActionEvent?

In den Einführungskapiteln haben Sie zwei grundlegende Methoden kennen gelernt, um auf Benutzerereignisse zu reagieren: Zum einen können durch Listener, die an Kommando-Komponenten wie einen Commandbutton oder einen Hyperlink angehängt werden, Events erzeugt und in einer Listenerklasse verarbeitet werden. Zum anderen besteht bei UICommand-Komponenten auch die Möglichkeit, eine Aktion mit Hilfe des *Method Bindings* direkt auszuführen.

Rein technisch gesehen kann mit beiden Varianten ein gewünschtes Ergebnis erreicht werden. Jedoch ist nicht jede Variante gleich elegant. Grundsätzlich ist es empfehlenswert, konkrete Aktionen, die auf ein Ereignis folgen sollen, in Aktionsmethoden zu setzen. Häufig bestimmen Aktionsmethoden auch das weitere Navigationsverhalten. Dieses kann durch einen passenden Rückgabewert einer Aktionsmethode festgelegt werden.

ActionEvents beziehen sich des Weiteren primär auf ein bestimmtes Objekt, das eine Aktion ausgelöst hat. Daher empfiehlt es sich, ActionListener nur zu verwenden, wenn bestimmte Benachrichtigungen speziell für ein Objekt verarbeitet werden müssen. Ein Beispiel dafür ist das Kapitel (10.7 Steuerung der Komponentensichtbarkeit), in dem in der Listenerklasse die Sichtbarkeit von Tabellenspalten gesteuert wurde.

10.10 Verwendung von Stylesheets

Cascading Style Sheets (CSS) existieren bereits seit vielen Jahren, gehören somit schon fast zu den Dinosauriern der Webtechnologie. Dennoch wurden sie einige Zeit sehr stiefmütterlich behandelt und finden erst langsam größeren Zuspruch in der Entwickler- und Webdesignerbranche.

Mit Stylesheets können Gestaltungsaspekte wie Farben, Kantenverläufe, Schriftarten etc. näher spezifiziert werden. Da Stylesheets zusätzlich zu den Html-Dateien vorliegen, kann mit deren Verwendung eine Trennung von Inhalt und Layout erreicht werden. Während Html prinzipiell als Auszeichnungssprache erdacht ist, kann die Darstellung mit Hilfe von Stylesheet-Angaben näher bestimmt werden.

Historie

Leider wurde während des großen Browserkriegs zwischen Microsoft und Netscape die Stylesheet-Technik außen vor gelassen, in dem viele Layoutelemente direkt in proprietären Erweiterungen in Html eingeflochten wurden. Das bekannteste Beispiel ist das `font`-Tag, mit dem eine Schriftart und Schriftgröße (sogar auch die Schriftfarbe) in einem Html-Tag direkt in der Seite gesetzt werden konnte. Natürlich verhalten sich jeder Browser und teilweise oftmals auch die einzelnen Browserversionen auch des gleichen Herstellers bei diesen proprietären Erweiterungen mitunter sehr verschieden. Oft kommt es zu völlig verschobenen Darstellungen oder bestimmte Textblöcke erscheinen überhaupt nicht.

Sicherlich, Html war nie für graphische Ausgaben bestimmt, sondern diente anfangs als Beschreibungssprache für (wissenschaftliche) Dokumente. Erst mit dem Aufschwung des World Wide Web auch im privaten Bereich wurden Anforderungen nach mehr Gestaltungsmöglichkeiten laut, die die großen Browserhersteller mit eigenen proprietären Erweiterungen zu befriedigen versuchten. Leider war das Ergebnis meist, dass eine Html-Seite je nach verwendetem Browser komplett unterschiedlich dargestellt wurde. Sobald ein ansprechendes Design auf einem Browser fertig erstellt war, konnte es in einem anderen Browser teilweise nicht einmal mehr angezeigt werden.

Um diesen Problemen aus dem Weg zu gehen, wurden bestimmte Layouts anstelle von Html einfach in einem Grafikprogramm erstellt und als Bild im Web publiziert. Dies war jedoch vorsichtig ausgedrückt eine sehr bescheidene Lösung, stiegen die Ladezei-

ten doch dadurch extrem an. Ein Ausweg aus dieser Misere ist die Verwendung von Stylesheets, die durch das W3C-Konsortium (World Wide Web-Konsortium) standardisiert und damit streng geregelt sind.

Der schleppende Einsatz von Stylesheets liegt sicherlich zum einen darin begründet, dass die Verwendung von Stylesheets ein gewisses Grundwissen und Verständnis für die Technologie erfordern, zum anderen aber auch darin, dass ältere, aber weit verbreitete Browser wie Netscape 4.7x erhebliche Probleme mit CSS-Angaben haben.

Da die aktuellen Browser der letzten Jahre fast durchweg Stylesheets in akzeptabler Weise implementieren, sollte bei künftigen Webprojekten verstärkt auf deren Einsatz gedrängt werden, erleichtern sie doch erheblich die Arbeit speziell von Webdesignern.

Funktionsweise

Stylesheets können auf insgesamt drei unterschiedliche Weisen mit einer Html-Seite (bzw. einer JSF-Seite) verbunden werden.

- Definition im Kopf-Bereich einer Html-Seite
- Definition in den Tags direkt
- Definition in einer separaten Datei

Meist wird die Definition der Stylesheet-Angaben in einer separaten Datei vorgenommen, da hier eine zentrale Datei vorliegt, die dann wiederum in jede einzelne Html-Seite eingebunden werden kann.

In den Stylesheet-Dateien werden dann so genannte Stilvorlagen (das deutsche Wort ist ein wenig gewöhnungsbedürftig) hinterlegt. Durch die Angabe von

```
H3 {
  font-family: sans-serif;
  font-size: 24px;
  font-weight: bold;
}
```

wird beispielsweise festgelegt, dass alle Überschriften der Stufe 3 in der Schriftart *Sans-Serif* in der Größe *24 Pixel* in *fetter* Schreibweise dargestellt werden.

In dieser Art ist die gesamte Syntax von Stylesheets aufgebaut. Innerhalb einer geschweiften Klammer können Attribute zu einem Html-Tag näher beschrieben werden. Insgesamt weist allein die CSS1-Spezifikation über 70 Eigenschaften auf, die über Stylesheet-Angaben gesetzt werden können.

JSF und Stylesheets

Die Integration von Stylesheets in JSF ist ein wichtiger Bestandteil der Spezifikation. So gibt es in allen Standard-Tags, die eine HTML-Ausgabe erzeugen, die Möglickeit, Stylesheet-Attribute mitanzugeben. Diese werden durch die Rendererklassen gemäß der Html- bzw. CSS-Spezifikation in der HTML-Seite ausgegeben.

Abbildung 10.4: Verwendung von Stylesheets

Abbildung 10.4 zeigt eine einfache Faces-Seite, in der Stylesheet-Angaben in JSF-Tags integriert wurden. Die unterschiedlichen Ausgaben des Speichern-Knopfes resultieren lediglich in einer anderen Styleklasse, in allen Fällen wurde der gleiche Rendering-Code verwendet.

```
Im Folgenden wird ein Text mittels des
outputText-Tags ausgegeben, dem eine Styleklasse
zugeordnet ist:
<h:outputText value="Testausgabe" styleClass="styleBlue" />
<br><br>

Das nächste Beispiel verwendet lediglich eine andere
```

```
Styleklasse:
<h:outputText value="Testausgabe" styleClass="styleRed" />
<br><br>

Jetzt wird ein Button durch ein Stylesheet verändert:
<h:commandButton value="Speichern" styleClass="btnStyle" />
<br><br>

Wiederum das gleiche Tag, lediglich mit anderen Stylesheet-Angaben:
<h:commandButton value="Speichern" styleClass="btnStyleExt" />
```

Listing 10.15: Verwendung von Stylesheet-Angaben in Faces-Tags

Listing 10.15 zeigt die eigentliche JSF-Seite. Es werden hierbei die Tags `outputText` und `commandButton` verwendet. Jedoch weisen fast sämtliche Standard-Tags in JSF die Möglichkeit auf, Stylesheet-Angaben mitanzugeben. Das verwendete Stylesheet ist natürlich wie alle Beispiele dieses Buches auf der beigelegten CD zu finden.

Ein großer Vorteil bei Verwendung von Stylesheets ist auch, dass das Design angepasst werden kann, ohne das Rendering in der Rendererklasse anpassen zu müssen. Somit kann ein Renderingcode einer Komponente je nach verwendetem Stylesheet zu unterschiedlichen Darstellungen führen. Da Stylesheets leicht anzupassen sind, ist damit eine sehr einfach Möglichkeit gegeben, das Design einer Anwendung auch nach Fertigstellung der eigentlichen Programmierung noch anzupassen.

10.11 Seitenverlinkung

Eine Webanwendung besteht meist aus einer umfangreichen Ansammlung von einzelnen (JSF) Seiten, die miteinander durch so genannte Hyperlinks verbunden sind. Oftmals gelangt man auch zur nächsten Seite, indem ein Formular über einen Submit-Button abgeschickt wird. Häufig sind jedoch Hyperlinks anzutreffen, die entweder lediglich zu einer Folgeseite führen oder auch konkrete Aktionen in einer Webanwendung auslösen. Zur Darstellung von Hyperlinks existieren in JSF zwei Tags, das `commandLink`-Tag und das `outputLink`-Tag.

Beide Tags sehen auf Seiten der Benutzeroberfläche vollkommen identisch aus, jedoch existiert ein gravierender Unterschied in der Verwendung und dem Zweck beider Tags. Das `commandLink`-Tag löst – wie der Name bereits andeutet – nach Auslösen des Links eine Aktion aus. Dabei wird das Formular, in dem das Tag integriert ist, an den Server zurückgeschickt (ein so genannter *Post Back*), verarbeitet und danach auf eine entsprechende Seite weitergeleitet. Die Folgeseite ergibt sich dabei aus der Anwendungskonfigurationsdatei der Webapplikation.

Das `outputLink`-Tag stellt einen meist externen Link dar, der den Benutzer auf eine entfernte Seite führt. Es erfolgt in diesem Fall keine Aktionsverarbeitung auf Serverseite.

Was auf keinen Fall vorkommen sollte, ist, dass mittels des `outputLink`-Tags auf eine JSF-Seite innerhalb der eigenen Applikation verwiesen wird. Die korrekte Vorgehensweise ist es, einen Seitenwechsel innerhalb der eigenen Applikation mittels des `commandLink`-Tags und einer Eintragung in der Anwendungskonfigurationsdatei zu definieren.

10.12 JSF und EJBs

Enterprise Java Beans (EJBs) haben sich mittlerweile in der Systemlandschaft von Unternehmen etabliert. Waren vor einigen Jahren nur sehr wenige ehrgeizige und innovative Firmen damit beschäftigt, J2EE-Anwendungen inklusive EJBs zu realisieren, gehören EJBs heute bereits zu einer etablierten Technologie.

Hintergrund

Der EJB-Standard wurde entwickelt, um damaligen (und natürlich auch heutigen) Anforderungen nach verteilten Lösungen in Enterprise-Anwendungen nachkommen zu können. Unternehmenskritische Anwendungen sollten in die Lage versetzt werden, mit Hilfe der Java-Technologie performante, ausfallsichere und robuste Anwendungen realisieren zu können. Da Enterprise Java Beans in einem so genannten EJB-Container ausgeführt werden, ist zunächst einmal ein entsprechender Application Server mit einem EJB-Container die Grundvoraussetzung für den Einsatz von EJBs.

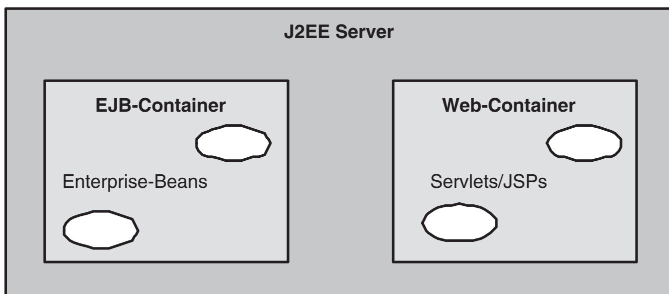


Abbildung 10.5: Aufbau eines J2EE-Servers

Über den EJB-Container stehen Services für Connection-Pooling von Datenbankverbindungen, Naming-Dienste, Security-Dienste, Data Caching und noch einige weitere Dienste zur Verfügung. Der Fokus von EJBs liegt daher ganz im Gegensatz zu JSF nicht auf der Präsentationsseite, sondern vielmehr in den Abläufen im Hintergrund. Sprich

es werden Prozesse angestoßen, Datenbankzugriffe ausgeführt oder komplexe Algorithmen abgearbeitet, das Ergebnis wird dann über ein UI-Framework wie z. B. JSF zur Anzeige gebracht. Diese Architektur, in der einzelne Schichten (so genannte *Tiers*) sauber voneinander getrennt arbeiten, wird als Schichtenarchitektur bezeichnet.

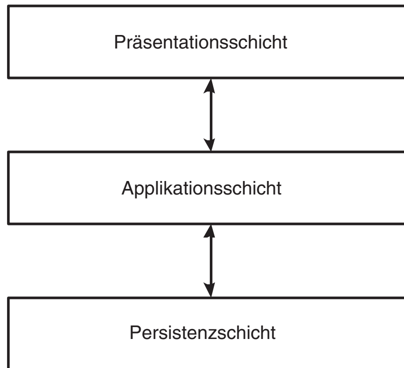


Abbildung 10.6: 3-Schichten-Architektur

In einer 3-Tier-Architektur wird in der Datenzugriffsschicht die Logik für den Zugriff auf Datenbankinhalte gekapselt, während in der Schicht für die Geschäftslogik (Applikationsschicht) die eigentlichen Geschäftsprozesse und Abläufe stattfinden. In der Präsentationsschicht ist die Schnittstelle zum Nutzer enthalten.

Während mittels J2EE und EJBs die unteren Schichten abgebildet werden können, ist JSF optimal zur Realisierung der Präsentationsschicht geeignet.

In einer mehrschichtigen Architektur können weitere Schichten z. B. für Zugriffe auf EIS (Enterprise Information Services)-Systeme integriert werden. Ein typisches Beispiel wäre ein Zugriff auf die betriebswirtschaftliche Standardsoftware SAP. Aber auch in einer n-Tier-Architektur kann JSF für die Präsentationsschicht verwendet werden.

Application Server

EJB-Container werden durch den einen Application Server bereitgestellt. Statt eines Application Servers sind auch die Bezeichnungen EJB-Server oder J2EE-Server gebräuchlich. Während ein EJB-Container die Laufzeitumgebung für Enterprise Beans darstellt, ist ein J2EE-Server die Laufzeitumgebung für EJB-Container. Dies impliziert, dass in einem J2EE-Server unter Umständen mehrere EJB-Container vorherrschen können. Der Server selbst ist dabei für verschiedene Aufgaben zuständig, wie beispielsweise:

- Namens- und Verzeichnisdienste
- Ausfallsicherheit
- Lastverteilung
- - ...

Mittlerweile gibt es eine kaum mehr überschaubare Anzahl von Herstellern, die J2EE-konforme Server anbieten. Das Preisspektrum dieser Produkte reicht dabei von komplett kostenfrei bis hin zu einigen 100.000 €. Für die in diesem Buch gezeigten Beispiele wird der *Jboss*-Server verwendet. Jboss ist ein auf Open-Source basierter J2EE-Server, der kostenfrei über die Internetseite

http://www.jboss.org

heruntergeladen werden kann. Auch aufgrund der Tatsache, dass dieser als Open-Source-Produkt keinen Lizenzzahlungen des Anwenders unterliegt, hat er sich mittlerweile stark verbreitet. Gute und ausführliche Dokumentationen sind mittlerweile reichlich vorhanden, wenn auch seit kurzem die Dokumentation über die Jboss-Homepage kostenpflichtig angeboten wird. Diese minimale Investition sollte man auch möglichst tätigen, wenn man bedenkt, welchen großen Gegenwert in Form eines ausgereiften und professionellen Servers man dafür bekommt.

Die Installation des Server erfordert im Normalfall lediglich ein Entpacken der heruntergeladenen Zip-Datei (bzw. tar), weitere Schritte sind nur bei Bedarf notwendig. Zu bedenken ist jedoch, dass bei der Jboss-Installation zugleich ein Webcontainer (ebenfalls Tomcat) mitinstalliert wird. Dies ist dann zu beachten, wenn die für die Beispielanwendung erstellten JSF-Seiten und Klassen deployed, d.h. ausgeliefert werden. Entweder wird der in Jboss vorhandene Webcontainer verwendet, oder aber dieser wird deaktiviert, damit der bisher benutzte Webcontainer (der auch in der Entwicklungsumgebung konfiguriert ist) verwendet werden kann.

Tomcat in Jboss deaktivieren

Jboss hat in der Standardinstallation Tomcat als Webcontainer mitinstalliert und aktiviert. Wird eine separate Tomcatinstallation verwendet (im gezeigten Beispiel die Version, die in der Entwicklungsumgebung eingebunden ist), kann entweder der Dienst in Jboss komplett deinstalliert oder einfach der in Jboss vorhandene Tomcat auf einen anderen Port gelegt werden. Da die letzte Alternative am schnellsten und einfachsten zu bewerkstelligen ist, wird in der Datei *jboss-service.xml* im Verzeichnis *server/default/deploy/jboss-web-tomcat41.sar/META-INF* der Eintrag `port` von `8080` auf `8081` verändert. Diese Angaben beziehen sich auf die Version 3.22 des Jboss-Servers. In anderen Versionen ist die Angabe des Ports eventuell an einer anderen Stelle vorzunehmen.

Schnittstelle

Im vorangegangenen Beispiel wurde bereits erörtert, an welcher Stelle die Geschäftslogik eingebaut werden kann. Diese Frage kann auch unabhängig von EJBs beantwortet werden. Es empfiehlt sich, in den Action-Methoden, die durch bestimmte Aktionen eines Benutzers über die Oberfläche angestoßen werden, einen entsprechenden Aufruf einzubauen. Für die Realisierung ohne EJBs wurde beispielsweise in der Action-Methode eine entsprechende Service-Klasse aufgerufen, die wiederum das Ergebnis zurückgeliefert hat. Auf keinen Fall sollte die Logik in der Action-Klasse des Beans selbst beinhaltet sein.

Im Falle von EJBs empfiehlt es sich daher auch, den EJB-Aufruf in Action-Klassen zu integrieren, eventuell können die Aufrufe auch über Service-Klassen erfolgen, die wiederum in den Action-Klassen verwendet werden.

Der EJB-Container

Wichtig für das Verständnis ist es, die Unterschiede zwischen einem EJB- und einem Webcontainer zu kennen. Es wurde bereits erläutert, dass EJBs in einem so genannten EJB-Container ablaufen, dieser stellt sozusagen die lebensnotwendige Umgebung für Enterprise Beans dar. Genauso, wie ein Enterprise Bean von einem EJB-Container abhängig ist, benötigt ein EJB-Container wiederum einen entsprechenden Server als Laufzeitumgebung.

EJBs sind als Teil der J2EE-Spezifikation genau spezifiziert. Neben der EJB-Spezifikation sieht die J2EE-Spezifikation APIs (d.h. Programmierschnittstellen) auch noch für folgende Dienste vor:

- API für das JDK (Java Development Kit)
- API für JNDI (Java Naming and Directory Interface)
- JDBC-API (für den Datenbankzugriff)
- JavaMail
- ...

Diese Auflistung ist natürlich in keinster Weise vollständig, soll jedoch zeigen, wie komplex das Thema J2EE ist. EJBs sind sicherlich eines der für einen Neueinsteiger schwierigsten und umfangreichsten Themen. Daher ist es nicht verwunderlich, wenn für eine EJB-Entwicklung meist sehr erfahrene Entwickler gesucht werden.

Der EJB-Container stellt den Enterprise Beans jedoch nicht nur die Laufzeitumgebung zur Verfügung, sondern übernimmt zudem viele Dienste wie z.B. das Instanzen-Pooling oder das Erzeugen und Löschen von Beans generell. Je nach Art des Beans werden auch Themen wie Persistenz, also die Speicherung von Beans, durch den Container abgewickelt.

Typen von Beans

Ebenfalls in der EJB-Spezifikation sind unterschiedliche Arten von Enterprise Beans definiert. Hierbei wird grundsätzlich zwischen *Entity Beans* und *Session Beans* unterschieden. Bei Entity-Beans wiederum wird zwischen *bean-managed persistence* sowie *container-managed persistence* unterschieden, bei Session Beans gibt es die Unterscheidung in *stateful* und *stateless*.

Session Beans werden meist zur Abbildung von Geschäftsprozessen verwendet. So werden mit Hilfe eines Session Beans Berechnungen durchgeführt, Transaktionen in einem Warenwirtschaftssystem abgehandelt oder auch Daten aus einer Datenbank gelesen. Die Unterscheidung in *stateful* und *stateless* bezieht sich dabei auf den Zustand des Beans. Bei stateless (zustandslos) werden im Bean selbst keine Daten zwischen verschiedenen Methodenaufrufen gespeichert. Ein Beispiel wäre eine Addition, in der beim Methodenaufruf die notwendigen Parameter übergeben werden und das Ergebnis der Berechnung zurückgeliefert wird. Der Zustand nach dem Methodendurchlauf ist noch der gleiche wie davor. Bei stateful (zustandsbehaftet) werden Daten im Bean gespeichert und stehen auch nach einem Methodenaufruf noch zur Verfügung.

Im Gegensatz zu Session Beans stellen Entity Beans meist konkretere Dinge wie z. B. einen Kunden, eine Zahlung, eine Anlage oder ein Fahrzeug dar. Zusätzlich repräsentieren Entity Beans Daten aus einer Datenbank. In Anwendungen werden Entity Beans meist in Zusammenarbeit mit Session Beans verwendet, indem Geschäftsabläufen in Session Beans als Parameter Entity Beans mitgegeben werden. Die Unterscheidung der Entity Beans in bean-managed persistence und container-managed persistence bezieht sich dabei auf die Verantwortlichkeit, wer für die Datenspeicherung zuständig ist. Bean-managed bedeutet, dass das Bean selbst für eine korrekte Speicherung seines Zustandes in der Datenbank verantwortlich ist, bei container-managed muss sich der EJB-Container darum kümmern.

Benötigte Klassen

Für ein funktionsfähiges Enterprise Bean werden mindestens drei Klassen/Interfaces benötigt:

- Home-Interface
- Remote-Interface
- Bean-Klasse

Aufgrund der Tatsache, dass Enterprise Beans keine graphische Repräsentation haben und nicht »einfach so« lauffähig sind, wird zudem ein Client-Programm benötigt, das Funktionalitäten des Enterprise Beans aufruft bzw. anstößt.

Zusätzlich zu den verwendeten Klassen werden weiterhin mindestens zwei weitere Konfigurationsdateien verwendet. Die erste Datei, der Deployment-Deskriptor *ejb-jar.xml*, ist in der EJB-Spezifikation genau beschrieben und beinhaltet beanspezifische Angaben. Die zweite Datei ist meist ein containerspezifischer Deskriptor, der sich von Application Server zu Application Server unterscheidet. In den gezeigten Beispielen wird der Jboss-Server verwendet, dort trägt der Deskriptor den Namen *jboss.xml*.

Im Home-Interface sind Methoden definiert, die für das Erzeugen, Suchen oder Löschen des Beans notwendig sind. Diese Schnittstelle wird vom Client verwendet, um konkrete Instanzen zu erzeugen und diese danach zu verwenden. Das Interface selbst ist von `java.ejb.EJBHome` abgeleitet.

Das Remote-Interface ist die entfernte Schnittstelle einer Bean-Klasse. Darin werden die beanspezifischen Methoden hinterlegt, die für die Anwendungslogik benötigt werden. Soll beispielsweise ein Enterprise Bean zwei Zahlen addieren, so wird im Remote-Interface eine Methode `summiereZahlen( int a, int b )` definiert. Im Remote-Interface wird somit die eigentliche Funktionalität des Beans festgelegt.

Die konkrete Implementierung der benötigten Zentralfunktionen wird in der Bean-Klasse vorgenommen. Alle Methoden des Home- und Remote-Interfaces werden in der Bean-Klasse implementiert, ohne jedoch dass die beiden Interfaces konkret über eine `implements`-Anweisung eingebunden werden.

Beispielanwendung

Um die Funktionsweise exemplarisch aufzuzeigen, wie aus einer JSF-Anwendung EJB-Funktionalitäten verwendet werden können, wird anhand einer kleinen Beispielanwendung die Vorgehensweise im Folgenden näher erläutert.

Als Geschäftsprozess soll die Verzinsung eines Kapitaleinsatzes berechnet werden. Es wird davon ausgegangen, dass ein bestimmter Geldbetrag auf ein festverzinsliches Sparbuch anlegt und die jährlich anfallenden Zinsen ebenfalls dort wieder angespart werden, anstatt dass diese ausbezahlt werden. Anhand eines Basiszinssatzes soll der Endbetrag in n Jahren berechnet werden, der sich bis dahin angesammelt hat. Dies ist somit eine klassische Zinseszinsrechnung.

Über die in JSF realisierte Oberfläche werden die benötigten Parameter Startkapital, Anlagedauer in Jahren sowie der Zinssatz eingegeben. Daraufhin werden die Parameter an ein Session Bean übergeben, das die notwendige Zinseszinsrechnung durchführt. Das Ergebnis der Berechnung wird wieder auf einer JSF-Seite zur Anzeige gebracht.

Vorarbeiten

Das Grundgerüst für die Anwendung ist schnell entwickelt. Es wird für die Anzeige der Daten in der Oberfläche ein Modellobjekt verwendet, das entsprechende Bean ist in Listing 10.16 abgebildet.

```
package com.edu.jsf.bsp.bean;

import java.util.Properties;

import com.edu.jsf.bsp.ejb.Interest;
import com.edu.jsf.bsp.ejb.InterestHome;

/**
 * Bean, das eine einfache Zinsrechnung anstößt.
 */
public class InterestBean {

  private double capital;
  private double rate;
  private int years;
  private double targetcapital;

  /**
   * Getter-Methoden
   */
  public double getCapital() {
    return capital;
  }

  public double getRate() {
    return rate;
  }

  public int getYears() {
    return years;
  }

  public double getTargetcapital() {
    return targetcapital;
  }

  public String calculate() {

    // An dieser Stelle den
    // EJB-Aufruf einsetzen

    return null;
  }

  /**
```

```
     * Setter-Methoden
     */
    public void setCapital(double d) {
      capital = d;
    }

    public void setRate(double d) {
      rate = d;
    }

    public void setYears(int i) {
      years = i;
    }

    public void setTargetcapital(double d) {
      targetcapital = d;
    }
  }
```

Listing 10.16: Das Zins-Bean

Die Methode `calculate` bietet eine passende Möglichkeit, einen EJB-Aufruf einzusetzen. Die Stelle, an der dieser Aufruf erfolgen soll, ist in Listing 10.16 noch ausgelassen. Dies wird in den nächsten Schritten erarbeitet.

Die JSF-Seite stellt an sich nichts Außergewöhnliches dar, es werden lediglich die Daten des Interest Bean zur Anzeige gebracht. Da keine Navigationsregeln hinterlegt sind, wird nach dem Auslösen des Buttons auf dieselbe Seite verwiesen und damit das in der `calculate` –Methode berechnete Zinsergebnis zur Anzeige gebracht.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head>
  <title>JSF-EJB-Zusammenspiel</title>
</head>

<body>
<f:view>
  <h3>Zusammenspiel von JSF und EJB:</h3>
  <i>Dieses Beispiel demonstriert den Aufruf eines
     Enterprise Java Beans aus einer JSF-Anwendung
     heraus:</i>
  <br><br>

  <h:form id="ejbForm">

    <h:panelGrid columns="2">
      <f:facet name="header">
        <h:outputText value="Zinseszinsrechnung" />
```

```
        </f:facet>

        <h:outputText value="Kapital" />
          <h:inputText value="#{Interest.capital}" />
        <h:outputText value="Zins" />
          <h:inputText value="#{Interest.rate}" />
        <h:outputText value="Jahre" />
          <h:inputText value="#{Interest.years}" />

        <h:outputText value="Ergebnis" />
          <h:outputText value="#{Interest.targetcapital}" />
      </h:panelGrid>

      <br><br>
      <h:commandButton value="Weiter"
         action="#{Interest.calculate}" />

    </h:form>

  </f:view>
  </body>
  </html>
```

Listing 10.17: JSF-Seite für EJB-Beispiel

Home-Interface

Das Home-Interface wird dazu benutzt, Bean-Instanzen zu erzeugen bzw. diese gegebenenfalls auch wieder zu löschen. Ebenso können Methoden für das Suchen enthalten sein, was ausschließlich bei Entity Beans einen Sinn ergibt, da nur diese in einer Datenbank gespeichert und damit auch über Suchfunktionen eindeutig identifiziert werden können.

```
package com.edu.jsf.bsp.ejb;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface InterestHome extends EJBHome
{
  /**
   * liefert eine Interest-Instanz zurück
   */
  public Interest create()
    throws RemoteException, CreateException;
}
```

Listing 10.18: Das Home-Interface InterestHome

Das in Listing 10.18 abgebildete Home-Interface ist bewusst einfach gehalten, es soll in diesem Beispiel lediglich gezeigt werden, wie schnell eine Kombination von JSF und EJBs realisiert werden kann.

Remote-Interface

Das Remote-Interface definiert die eigentlichen Methoden, auf die es in einem Bean ankommt, sprich die Berechnungsmethode für den Zinseszins. Wichtig ist die Vererbungshierarchie, das Remote-Interface leitet von `javax.ejb.EJBObject` ab.

```
package com.edu.jsf.bsp.ejb;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Interest extends EJBObject
{
  /**
   * berechnet den Zinseszins basierend auf einem
   * Startkapital und einem Zinssatz sowie der
   * Laufzeit
   */
  public double calculateCapital( double basecapital,
         double rate, int years ) throws RemoteException;
}
```

Listing 10.19: Das Remote-Interface Interest

Die Methode zur Berechnung des verzinsten Kapitals `calculateCapital` liefert einen `double`-Wert zurück. Eventuell könnte das gesamte Verfahren auf `BigDecimal` umgestellt werden, für einfache Beispiele sollte der `double`-Datentyp jedoch genügen. Des weiteren kann eine `RemoteException` geworfen werden, diese ist somit ebenfalls für die Methode anzugeben.

Bean-Klasse

In der Bean-Klasse erfolgt die Implementierung der in den Interfaces definierten Methoden. So wird auch das verzinste Kapitel konkret berechnet und das Ergebnis als `double`-Wert zurückgeliefert. Das Bean selbst ist ein *SessionBean*, da es lediglich für eine Berechnung zuständig ist und daher nicht als Datenbankobjekt abgespeichert werden muss. Deswegen wird in der Bean-Klasse das Interface `javax.ejb.SessionBean` implementiert.

```
package com.edu.jsf.bsp.ejb;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
```

```
import java.rmi.RemoteException;

public class InterestBean implements SessionBean {
  /**
   * berechnet den Zinseszins basierend auf einem
   * Startkapital und einem Zinssatz sowie der
   * Laufzeit
   */
  public double calculateCapital( double basecapital,
         double rate, int years ) throws RemoteException {

    double temp = basecapital
                  * Math.pow( (1 + (rate/100d)), years );
    return temp;
  }

  public void ejbCreate() {
  }

  public void ejbPostCreate() {
  }

  public void ejbRemove() {
  }

  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void setSessionContext( SessionContext sc ) {
  }
}
```

Listing 10.20: Die Bean-Klasse InterestBean

Die Methode `calculateCapital` nimmt als Argument das Basiskapital, das zum Startzeitpunkt vorliegt. Des Weiteren werden der Zinssatz und die Anzahl an Jahre mit übergeben, die das Kapital verzinst werden soll. Die Formel ist die klassische Zinseszinsformel, wie sie sicherlich jeder in der Schule gelernt hat und sicherlich im Schlafe noch beherrscht.

Aufruf in der Action-Klasse

Der eigentliche EJB-Aufruf ist bei der Entwicklung der `Action`-Klasse noch außen vor geblieben. Dies wird jetzt nachgeholt, indem die notwendigen Aufrufe in die `calculate`-Methode eingebaut werden. Zunächst werden die benötigten Systemeigen-

schaften über `Properties` gesetzt und dem `InitialContext` zur Initialisierung übergeben. Es wird in diesem Beispiel davon ausgegangen, dass Jboss auf der lokalen Maschine installiert ist (`localhost`) und über den Standardport `1099` angesprochen werden kann.

```
public String calculate() {
  try {
    // Einstellungen für Kontext
    Properties props = new Properties();
    props.put( Context.INITIAL_CONTEXT_FACTORY,
      "org.jnp.interfaces.NamingContextFactory");
    props.put(Context.PROVIDER_URL, "localhost:1099");

    // Kontext holen
    InitialContext jndiContext = new InitialContext(props);

    // Referenz Interest-EJB
    Object ref = jndiContext.lookup("Interest");

    InterestHome home = (InterestHome) PortableRemoteObject.narrow(
      ref, InterestHome.class);
    Interest interest = home.create();

    targetcapital =
      interest.calculateCapital(capital, rate, years);
  } catch (Exception e) {
    System.out.println(e.toString());
  } // catch
  return null;
}
```

Listing 10.21: EJB-Aufruf in der Action-Klasse

Das Enterprise Bean bzw. eine Referenz darauf wird über den Aufruf

```
jndiContext.lookup("Interest")
```

ermittelt. *Interest* ist hierbei der Name des Enterprise Beans, über den die Identifizierung erfolgt.

Verbesserungspotenzial besteht in der `calculate`-Methode, da die gesamte EJB-Funktionalität für den entfernten Aufruf in der Methode implementiert ist. Eleganter wäre eine Lösung, in der in dieser Methode selbst nur ein weiterer Aufruf einer Methode vorhanden ist, in der die benötigte Funktionalität gekapselt ist und somit die Bean-Klasse, in der die Action-Methode enthalten ist, nicht zu überladen wird. Diese Vorgehensweise wird durch das Design Pattern *Facade* beschreiben.

Deployment

Im Gegensatz zu den bisher gezeigten Beispielen muss vor der Ausführung der EJB-Beispielanwendung ein wenig mehr Vorarbeit geleistet werden. Zum einen existieren jetzt zwei Server, ein Application Server (Jboss) für die Enterprise Beans sowie Tomcat für die Ausführung der JSP-Seiten und Servlets. Zum anderen reicht es nicht mehr aus, ausschließlich Java-Klassen in einem bestimmten Verzeichnis bereitzustellen. Vielmehr sind alle notwendigen EJB-Klassen zusammen mit einem Deployment Deskriptor zu packen und gebündelt an den Application Server zu übergeben. Dieser Prozess läuft in der Praxis selten von Hand ab, vielmehr stehen entsprechende Skripte dafür zur Verfügung.

Für das JSF-EJB-Beispiel ist in Listing 10.22 ein kurzes *Ant-Skript* abgebildet, mit dessen Hilfe die notwendige jar-Datei erstellt und zum Jboss-Server übertragen werden kann.

```
<?xml version="1.0"?>

<project name="ejb-jsf" default="deploy-jboss" basedir=".">

  <!-- ================================================== -->
  <!-- Wichtige Pfadangaben                               -->
  <!-- ================================================== -->
  <property name="jboss.home" value="C:\\server\\jboss-322RC4" />
  <property name="src.package" value="com\\edu\\jsf\\bsp\\ejb" />

  <!-- ================================================== -->
  <!-- "Abgeleitete" Pfadangaben                          -->
  <!-- ================================================== -->
  <property name="jboss.lib" value="${jboss.home}/lib" />
  <property name="jboss.configuration" value="default" />
  <property name="jboss.deploy.dir"
    value="${jboss.home}/server/${jboss.configuration}/deploy" />
  <property name="src.dir" value="${basedir}/src"/>
  <property name="lib.dir" value="${basedir}/lib"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.src" value="${basedir}/build/src"/>
  <property name="build.jboss" value="${build.dir}/jbossbuild"/>
  <property name="deploy.jboss" value="${build.dir}/jbossdeploy"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>
  <property name="build.bin.dir" value="${build.dir}/bin"/>
  <property name="build.javadocs.dir"
    value="${build.dir}/docs/api"/>

  <!-- ================================================== -->
  <!-- Ueberpruefung von JBoss-Home                       -->
  <!-- ================================================== -->
  <target name="check-jboss" unless="jboss.home">
    <fail>
```

```
      Property "jboss.home" nicht gefunden. Bitte
      entsprechende Einstellungen vornehmen.
    </fail>
  </target>

  <target name="init" depends="check-jboss">
    <echo message="user.home = ${user.home}"/>
    <echo message="java.home = ${java.home}"/>
    <echo message="ant.home = ${ant.home}"/>
    <echo message="jboss.home = ${jboss.home}"/>
    <echo message=""/>
    <available property="jdk1.4+"
     classname="java.lang.StrictMath" />
  </target>

  <!-- =================================================== -->
  <!-- Kopieren der Dateien                                 -->
  <!-- =================================================== -->

  <target name="copysrc" depends="init">
    <echo message="Copying source files ..." />
    <mkdir dir="${build.src}"/>

    <copy todir="${build.src}/${src.package}">
      <fileset dir="${src.dir}/${src.package}" />
    </copy>

    <mkdir dir="${build.dir}/META-INF"/>
    <copy todir="${build.dir}/META-INF">
      <fileset dir="${basedir}/../kap_10"
         includes="ejb-jar.xml" />
    </copy>
  </target>

  <!-- =================================================== -->
  <!-- Compile-Vorgang                                      -->
  <!-- =================================================== -->

  <target name="compile" depends="copysrc">
    <echo message="Compiling ..." />
    <mkdir dir="${build.classes.dir}"/>
    <javac
      destdir="${build.classes.dir}"
      debug="on"
      deprecation="off"
      optimize="on"
      classpath="${lib.dir}/jbossall-client.jar">
      <src path="${build.src}"/>
    </javac>
  </target>
```

```
<!-- ================================================== -->
<!-- JBoss jar-Datei erzeugen                           -->
<!-- ================================================== -->

<target name="jboss-jar" depends="compile">
  <mkdir dir="${deploy.jboss}"/>
  <jar jarfile="${deploy.jboss}/interestEJB.jar">
    <fileset dir="${build.classes.dir}" />
    <fileset
       dir="${build.dir}"
       includes="META-INF/ejb-jar.xml" />
  </jar>
</target>

 <!-- ================================================== -->
 <!-- jar-Datei deployen                                 -->
 <!-- ================================================== -->

<target name="deploy-jboss" depends="jboss-jar">
  <copy todir="${jboss.deploy.dir}">
    <fileset dir="${deploy.jboss}" includes="interestEJB.jar"/>
  </copy>
</target>

<!-- ================================================== -->
<!-- Am Ende wieder sauber aufraeumen                   -->
<!-- ================================================== -->

<target name="clean" depends="init">
  <echo message="Cleaning up ..." />
  <delete dir="${build.dir}"/>
</target>
</project>
```

Listing 10.22: Ant-Build-Skript

Eine ausführliche Beschreibung der Build-Datei bzw. des Build-Werkzeuges *Ant* würde mit Sicherheit den Rahmen dieses Buches sprengen. Gute Literatur zu diesem Thema ist u.a. auch über die Website *http://ant.apache.org* zu finden. Ebenfalls ist dort eine Kurzreferenz über die weiteren Erläuterungen zu jedem einzelnen Befehl zu finden.

Es empfiehlt sich auf alle Fälle, das Deployment mit Hilfe von Ant-Skripten zu realisieren. Das erste Einrichten dauert zwar ein wenig länger, dafür sind jedoch alle folgenden Deployments sehr schnell durchgeführt; ein Start des Skriptes genügt.

Nachdem die Anwendung erfolgreich geliefert wurde, steht einem Test nichts mehr im Wege. Über einen Browser wird zunächst die Start-JSF-Seite wie gewohnt aufgerufen und es werden einige Beispielwerte hinterlegt. Nach Betätigen des Buttons wird dann

eine Verbindung zum Jboss-Server aufgebaut und die Berechnung des Zinseszins im EJB-Container durchgeführt. Das Ergebnis erscheint bei fehlerfreier Installation wieder auf der JSF-Seite.

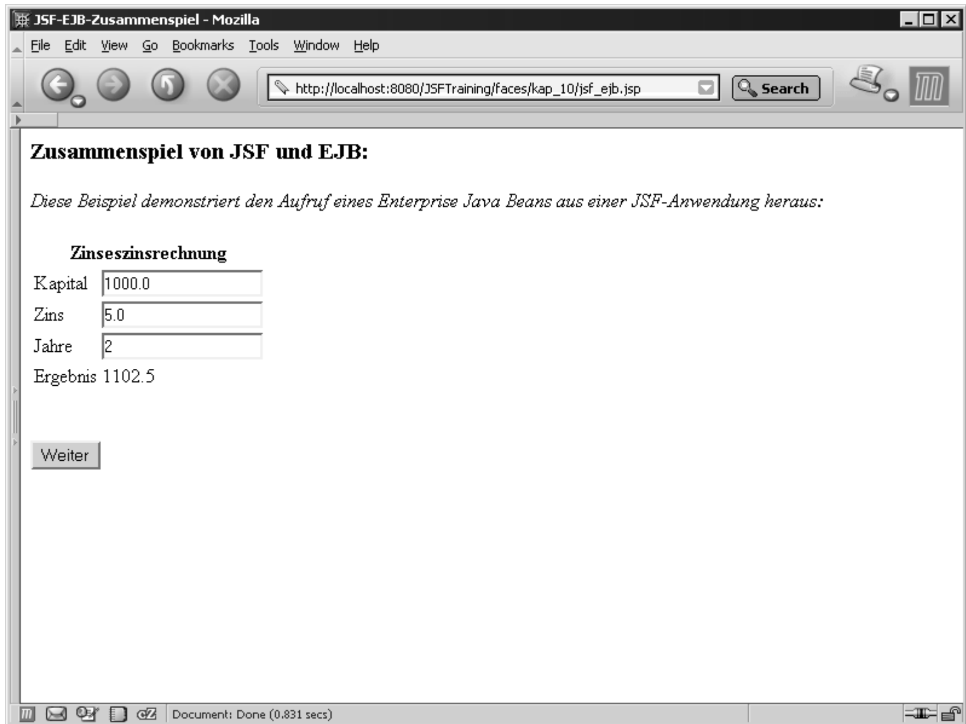


Abbildung 10.7: Erfolgreicher EJB-Aufruf

Fazit

Das Beispiel der Zinseszinsrechnung verdeutlicht in einem kurz gefassten Beispiel das Zusammenspiel von JSF und EJB. Es wurde dargestellt, dass durch EJBs wichtige Geschäftsprozesse anstatt im Webcontainer besser im EJB-Container aufgehoben sind. Zumal stellen EJBs eine sehr gute Ergänzung zu JSF dar, da JSF primär auf eine Oberflächenverarbeitung zielt, EJBs dagegen Funktionalitäten serverseitig abdecken können.

Wie bereits angedeutet kann das Beispiel noch weiter ausgebaut und eleganter aufbereitet werden, indem z. B. der EJB-Aufruf in eine separate Klasse ausgelagert wird, was dem Entwurfsmuster einer Fassade entspricht.

Letzten Endes ist jedoch deutlich geworden, dass bei einem EJB-Einsatz zwar weitere Planungen und Architekturüberlegungen notwendig sind, EJBs auf alle Fälle aber eine Bereicherung für die Entwicklung von leistungsfähigen und performanten Webanwendungen darstellen.

10.13 Cancel-Buttons

Eine in Webanwendungen vollkommen übliche Funktionalität ist, dass eine Eingabe z.B. bei einer Benutzeranmeldung oder bei einem Bestellvorgang jederzeit abgebrochen werden kann. Sicherlich ist es ein Leichtes, dazu einfach einen weiteren Button einzubauen, dessen `action`-Attribut auf eine entsprechende Seite zurückverweist. Das Problem ist jedoch, dass im Falle eines Abbruchs einer Eingabe oder einer Verarbeitung im aktuellen Formular keine Validierung und auch kein Modellupdate durchgeführt werden darf.

Es muss somit im Falle eines Cancel-Buttons dem Framework mitgeteilt werden, dass es sich um eine spezielle Aktion handelt, in der keine Validierung und kein Modellupdate stattfinden soll, sondern lediglich zu einer anderen Seite verzweigt werden soll.

Daher haben alle Command-Komponenten die Eigenschaft `immediate`, die anzeigt, dass der Standard ActionListener direkt ausgeführt werden soll, ohne die *Invoke Application* Phase explizit zu durchlaufen.

```
<h:form id="frmValidator">
  <h:inputText value="" required="true" size="10" />
  <h:commandButton value="Submit" />
  <h:commandButton value="Cancel" immediate="true" />

  <h:messages />
</h:form>
```

Listing 10.23: Verwendung des immediate-Attributs

In Listing 10.23 existiert ein Eingabefeld, an das ein Required-Validator angehängt ist. Versucht man nun, ohne Eingabe das Formular mittels *Submit* abzuschicken, wird ein entsprechender Fehler geworfen. Bei Betätigen des *Cancel*-Buttons jedoch wird das Formular ohne Fehlermeldung abgeschickt.

10.14 Fazit

Dieses Kapitel demonstrierte eine Reihe von Anwendungsbeispielen, wie JavaServer Faces in konkreten Situation eingesetzt werden kann. Es wurden Techniken vorgestellt, effektiver mit dieser neuen Technologie umzugehend sowie grobe Fehler zu vermeiden.

Dieses Kapitel stellt natürlich keinen Anspruch auf Vollständigkeit, sondern soll eher einige exemplarische Hinweise geben, an welchen Stellen eventuell etwas genauer über den Einsatz von JavaServer Faces nachgedacht werden sollte.

Es wurde gezeigt, dass technologisch eine Datenbankanbindung in JSF-Anwendungen mit der JSTL-SQL-Bibliothek zwar hervorragend zusammenpasst (beides wird über Tags gesteuert), jedoch sollte diese Vorgehensweise nicht unbedingt in produktiven und komplexen Anwendungen zum Einsatz kommen.

Es wurde zudem nochmals Wert auf die Verwendung von Ressourcendateien gelegt, um somit sehr einfach eine Anwendung auch international einsetzbar machen zu können. Mittels Ressourcendateien werden sämtliche Textausgaben, die sprachabhängig sind, in einer separaten Datei ausgelagert und können somit problemlos in andere Sprachen übersetzt werden.

Auch wurde gezeigt, an welcher Stelle am besten die Geschäftslogik untergebracht werden kann und ob eher mit Aktionsmethoden oder mit ActionEvents gearbeitet werden soll. Ebenfalls haben Sie gesehen, wie mit Hilfe von EJBs Geschäftsprozesse außerhalb von JSF in einem EJB-Container durchgeführt werden können und das Ergebnis wiederum in JSF-Seiten zur Anzeige gebracht werden kann.

Wichtig ist mit Sicherheit auch der Abschnitt über den Cancel-Button, der einen Abbruch einer Formularverarbeitung bewirkt, ohne dass eine Datenvalidierung oder eine Übernahme der Daten ins Modell erfolgt. Dieser Anwendungsfall tritt in der Praxis mit Sicherheit sehr häufig auf.

11 JSF und Struts

Kapitelziel

Bereits in den einleitenden Kapiteln wurde auf die Verbindung von JavaServer Faces und Struts hingewiesen. Struts ist ein Open-Source-Framework, das ebenfalls wie JSF die Entwicklung von Webanwendungen unterstützt. Allerdings ist Struts bereits einige Jahre länger auf dem Markt verfügbar, so dass heute Anwendungen bereits vielfach mit Struts realisiert wurden. Im folgenden Kapitel soll daher keine generelle Aussage erarbeitet werden, welches das (vermeintlich) bessere oder ausgereiftere Framework ist, sondern vielmehr ein Zusammenspiel beider Technologien skizziert werden. Natürlich stellt dieses Kapitel keine Einführung in das Thema Struts dar, dies würde den Rahmen des Buches bei weitem sprengen. Es werden zunächst eine kurze Einführung sowie ein Überblick zum Thema Struts gegeben und im Anschluss daran wird auf das Zusammenspiel beider Frameworks eingegangen.

11.1 Was ist Struts?

Struts ist mit Sicherheit eines der beliebtesten Web-Frameworks, die zurzeit im Markt existieren. Struts hat mittlerweile eine sehr große Entwickler- und Fangemeinde. Gerade die große Entwicklergemeinde hat auch einen Großteil dazu beigetragen, dass Struts in seiner heutigen Version als sehr stabil und ausgereift bezeichnet werden kann.

Durch die sehr schnelle Einarbeitung, die mit zahlreichen im Internet verfügbaren Tutorials unterstützt wird, sind sehr viele Anwender ebenfalls auf Struts umgestiegen bzw. haben neue Projekte basierend auf Struts gestartet.

Vergleichbar zu JSF bietet auch Struts die Möglichkeit,

- schnell dynamische Webseiten zu generieren,
- basierend auf der MVC-Architektur Webanwendungen zu realisieren
- und mittels der umfangreichen Struts-Tag-Bibliothek effektiv JSP-Seiten umzusetzen.

Mit Craig McClanahan ist der Hauptentwickler und Visionär von Struts ebenfalls im Spezifikations-Gremium für JSF, so dass auch hier eine enge Verzahnung von JSF und Struts deutlich wird.

Struts hat den Vorteil, dass es 100% Open-Source ist, d.h. sämtlicher Quellcode frei verfügbar ist und bei Bedarf eingesehen und (unter bestimmten Voraussetzungen) verändert und angepasst werden kann. Auch existiert mittlerweile eine große Anzahl an Fachliteratur sowie eine aktive Gemeinde im Internet, so dass eine Unterstützung bei Fragen oder Problemen durchaus gegeben ist.

11.2 Weitere Entwicklung von Struts

JSF ist *kein* offizieller Nachfolger von Struts, auch wenn viele dies immer wieder behaupten. JSF und Struts werden weiterhin parallel existieren. Während JSF primär durch Sun bzw. durch die JSF-Community vorangetrieben wird, wird Struts durch eine entsprechende Open-Source-Gemeinde und nicht zuletzt auch durch Craig McClanahan weiterentwickelt.

Zudem verhält es sich so, dass JSF nicht sämtliche Funktionen von Struts gleichwertig oder sogar besser abbildet. Vielmehr hat jedes Framework weiterhin seinen Hauptfokus: Während JSF sich primär als Framework für das User-Interface positioniert, ist Struts allgemeiner aufgestellt als Web-Framework.

Zurzeit noch etwas nachteilig für Struts ist, dass die in Struts verwendete Tag-Bibliothek proprietär ist. Dies soll heißen, dass es mittlerweile durch die JSTL eine standardisierte und akzeptierte Tag-Bibliothek gibt, die ebenfalls wie die Struts-Bibliothek Logik-Tags, Bean-Tags etc. bereitstellt. Auch Struts wird in künftigen Versionen mit Sicherheit verstärkt auf die Funktionalität der JSTL aufbauen, anstatt eigene Bibliotheken zu integrieren.

In der Praxis wird es daher mit großer Wahrscheinlichkeit so sein, dass projektspezifisch unterschieden wird, welches Framework zum Einsatz kommt. Zudem ist es auch möglich (und auch tatsächlich gewollt), JSF und Struts parallel einzusetzen. Während JSF sicherlich starke Vorzüge in der Verwaltung und Verarbeitung auf Seiten des User-Interfaces aufweist, hat Struts eine mächtige Controller-Funktionalität, die JSF in dieser Form nicht beinhaltet.

11.3 Zusammenspiel von JSF und Struts

Um die erwähnten Vorteile von JSF und Struts nutzen zu können, wurde bereits frühzeitig mit der Arbeit an einer so genannten Integrationsbibliothek begonnen. Damit ist es möglich, JSF und Struts parallel zu nutzen, um somit die Vorteile beider Frameworks anwenden zu können.

Einerseits können damit die UI-Komponenten von JSF zum Einsatz kommen, aber auch die Controller von Struts verwendet werden. Gerade für Projekte, die bereits mit Struts realisiert wurden und mit Erscheinen von JSF eine langsame Integration bzw. Migration anstreben, ist die Integrationsbibliothek eine große Vereinfachung. Allerdings muss dazu erwähnt werden, dass die Integrationsbibliothek auf der Struts-Version 1.1 aufsetzt. Anwender von früheren Struts-Versionen müssen daher zunächst auf die aktuelle Struts-Version wechseln, bevor mit einer Integration von JSF und Struts begonnen werden kann.

Um eine Integration beider Frameworks zu erreichen, sind grundsätzlich folgende Arbeitsschritte notwendig:

- Einbinden der Bibliothek *struts-faces.jar*, indem die Datei in das *WEB-INF/lib* Verzeichnis kopiert und in den Build-Pfad aufgenommen wird
- Anpassen des Deployment-Deskriptors *web.xml*
- Umstellung der JSP-Seiten, indem für die UI-Komponenten neue JSF-Komponenten verwendet werden.

In den folgenden Abschnitten wird von einer auf Struts basierenden Anwendung die Migration bzw. Integration von JSF erläutert. Dabei wird zunächst die Struts-Anwendung kurz vorgestellt, wobei an dieser Stelle jedoch keine vollständige Einführung in Struts gegeben werden kann. Vielmehr wird davon ausgegangen, dass Sie bereits Grundlagen-Kenntnisse in Struts besitzen. Alternativ sind im Anhang auch entsprechende Literaturhinweise aufgeführt, in denen weitere Informationen zum Thema Struts nachgeschlagen werden können.

11.4 Von der Theorie in die Praxis

Bei der Migration von Struts zu JSF gibt es prinzipiell zwei Alternativen:

- harte Migration, indem sämtliche Struts-Funktionalität gegen JSF-Funktionalität ausgetauscht wird. Dies bedeutet, dass sämtliche JSP-Seiten neu entwickelt werden müssen ebenso wie alle Controller-Klassen in entsprechende Aktions-Klassen gewandelt bzw. überführt werden müssen. Ebenso müssen die Beans angepasst sowie die Ablaufsteuerung neu konzipiert werden.
- weiche Migration, indem durch Verwendung der Struts-Faces Integrationsbibliothek beide Frameworks aktiv sind und das jeweilige Framework in seiner Paradedisziplin die Führung übernimmt.

Eine harte Migration ist sicherlich nur zu empfehlen, wenn die Anwendung entweder sehr klein ist und daher der Aufwand auch überschaubar bleibt oder aber aufgrund von Funktionsanforderungen oder Architektur-Fehlern eine Neuentwicklung sowieso durchgeführt werden müsste.

Im Folgenden wird anhand einer Struts-Anwendung eine weiche Migration vorgestellt, indem unter Verwendung der Struts-Faces-Integrationsbibliothek ein Großteil der Struts-Funktionalität erhalten bleibt und durch Faces-Komponenten erweitert wird. Das Beispiel selbst ist eine *Newsletter-Anmeldung*, die als Wizard aufgebaut ist. Es werden über drei Schritte hinweg Informationen von einem Benutzer gesammelt und anschließend weiter verarbeitet.

Abbildung 11.1: Newsletter-Anwendung

Bei der Eingabe des Vor- und Nachnamens sowie der E-Mail-Adresse findet eine entsprechende Validierung statt. Diese ist recht einfach gehalten, es wird lediglich überprüft, ob überhaupt eine Eingabe vorhanden ist. Eventuelle Fehler werden in einer separaten Meldungszeile angezeigt.

Wichtig bei Newsletter-Anmeldungen ist es, dass sich ein Benutzer entscheiden kann, ob er diesen als reinen Text (Plain Text) oder in formatierter HTML-Form erhalten möchte. Gerade Anwender älterer Mailprogramme sind oftmals verärgert, wenn ein HTML-Newsletter nicht korrekt dargestellt werden kann, weil eventuell HTML-Befehle verwendet werden, die das Mailprogramm nicht unterstützt.

Abbildung 11.2: Wahl des Newsletter-Typs

Abbildung 11.3: Abschluss der Newsletter-Anmeldung

Als letzter Schritt wird der Newsletter-Abonnent in die Datenbank eingetragen, so dass dieser künftig im Verteiler aufgenommen ist.

Diese Anwendung gilt es im Folgenden auf JavaServer Faces zu bringen, ohne jedoch die komplette Struts-Funktionalität neu entwickeln zu müssen. Es wird daher zunächst auf einige Struts-Grundlagen eingegangen, um darauf aufbauend die Struts-Faces-Integrationsbibliothek erläutern zu können.

11.5 Einblick in die Struts-Anwendung

Das Arbeiten mit der Struts-Faces-Integrationsbibliothek setzt voraus, dass Sie sich bereits im Vorfeld mit Struts beschäftigt haben. Dieses Buch kann natürlich keine Einführung in die Arbeit mit Struts geben. Schwerpunkt dieses Kapitels ist es, deutlich zu machen, wie basierend auf einer reinen Struts-Anwendung eine weiche Integration der JSF-Komponenten möglich ist. Im Folgenden wird daher nur kurz die Funktionsweise der Struts-Anwendung erläutert, um daraufhin die Arbeit mit der Integrationsbibliothek vorstellen zu können.

11.5.1 Struts-Komponenten

Struts basiert (analog zu JavaServer Faces) auf dem Model-View-Controller-Prinzip sowie auf einer Tag-Bibliothek. Im Gegensatz zu JSF ist die Struts-Tag-Bibliothek nicht die JSTL, sondern eine Struts-spezifische Bibliothek. Von der Funktionalität gesehen sind die Struts- und die JSTL-Bibliothek jedoch recht ähnlich. In beiden Bibliotheken existieren Logik-Tags, die einfache Bedingungsabfragen erlauben, sowie Tags für Iterationen. Ziel der Verwendung der Integrationsbibliothek ist es daher, die Struts-spezifischen Tags durch die spezifizierten Tags der JSTL zu ersetzen, ohne die Funktionalität der Struts-Controller im Hintergrund einzuschränken.

Die MVC-Architektur in Struts ist so realisiert, dass es ein Modell in Form eines Beans gibt, ebenso einen (fachlichen) Controller, der die Auswertung der Benutzereingaben vornimmt sowie die Steuerung der Views lenkt und auch die View in Form einer JSP-Seite. Bei den Controllern werden in Struts prinzipiell zwei Arten von Controllern unterschieden: fachliche und technische Controller.

Sämtliche Anfragen des Browsers an eine Struts-Anwendung werden (analog zu JSF) zunächst an ein zentrales Servlet geleitet, das für die weitere Verarbeitung verantwortlich ist. Dieses Servlet stellt den technischen Controller dar. Dieser leitet eine Anfrage u. U. an weitere Controller weiter, die Geschäftslogik beinhalten oder entfernte Services aufrufen. Diese Controller sind die fachlichen Controller.

Die View ist wie bereits angedeutet wiederum als JSP-Seite aufgebaut. Es werden (auch analog zu JSF) Tag-Bibliotheken zur Darstellung von UI-Elementen verwendet. Eine View wird niemals direkt aufgerufen, sondern durch Einstellungen in einer Konfigurationsdatei (wieder analog zu JSF) gesteuert.

11.5.2 Struts-Seiten

Die Struts-Anwendung besteht aus insgesamt drei JSP-Seiten. Diese sind relativ einfach aufgebaut, eine Tabelle mit dazugehörigem Stylesheet sowie einige Struts-Tags genügen, um die Wizard-Funktion abzubilden. Das JSP-Grundgerüst ist in Listing 11.1 zu sehen.

```
<%@ page contentType="text/html; charset=iso-8859-1"
    language="java" %>

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html:html locale="true">
<head>
  <title>Struts-Newsletter-Anmeldung</title>
  <link rel="stylesheet" type="text/css" href="default.css" />
  <html:base />
</head>

<body bgcolor="white"><p>
<center>
<br>
<h3>Newsletter-Anmeldung</h3>
</center>

<html:form action="/register.do">

  <table border="1" cellpadding="3" frame="box" rules="none"
   align="center">
  <tr>
    <th colspan="2">Newsletter-Anmeldung 1/3</th>
  </tr>
  <tr>
    <td>Vorname:</td>
    <td>
      <html:text property="firstname" size="16"
       maxlength="16"/>
    </td>
  </tr>
  <tr>
    <td>Nachname:</td>
    <td>
```

```
        <html:text property="lastname" size="16"
         maxlength="16"/>
      </td>
    </tr>
    <tr>
      <td>E-Mail:</td>
      <td><html:text property="mail" size="16" maxlength="16"/></td>
    </tr>
    <tr>
      <td colspan="2" class="error">
        <html:errors/>
      </td>
    </tr>
    <tr>
      <td colspan="1"><br />
        <html:submit property="submit" value="Submit"/>
      </td>
      <td colspan="1"><br />
        <html:reset/>
      </td>
    </tr>
    </table>
  </html:form><br>

  </body>
  </html:html>
```

Listing 11.1: Struts-JSP-Seite

Zunächst werden die Struts-Tag-Bibliotheken in die JSP-Seite eingebunden. In Struts werden Bibliotheken für Bean-Tags, Html-Tags und Logik-Tags unterschieden. Dies verdeutlicht nochmals, dass die Struts-Bibliotheken sehr mächtig sind und auch sehr viel Funktionalität beinhalten. Nicht nur aufgrund dieser Tatsache ist Struts sehr beliebt in der Entwicklergemeinde.

In der Seite selbst wird über ein `form`-Tag ein Formular definiert. Dieses Formular kann über den Submit-Button ausgelöst werden. Dabei wird die Aktion `register.do` übergeben. Auf Basis dieser *action* kann mit Hilfe der Struts-Konfigurationsdatei die Folgeseite bestimmt werden. Auch in Struts werden die Dateinamen der einzelnen JSP-Seiten nicht direkt in den einzelnen Seiten hinterlegt, sondern ausschließlich in der Konfigurationsdatei.

Durch die Angabe von

```
<html:text property="firstname" size="16" maxlength="16" />
```

wird ein Texteingabefeld definiert. Die Attribute `size` und `maxlength` werden direkt an die HTML-Komponente weitergereicht und legen die Länge des Eingabefeldes sowie die maximale Eingabelänge fest. Interessant beim Ansatz von Struts ist, dass die Eigen-

schaft (*property*), die in einem Feld angezeigt wird, nicht wie in JSF über *ManagedBean.Attributname* angesprochen wird, sondern lediglich über das Attribut. Die Zuordnung, zu welchem Bean das Attribut letztendlich gehört, ist über die Konfigurationsdatei geregelt. Einem Formular ist dabei ein so genanntes *Formular-Bean* zugeordnet, das automatisch verwendet wird. An dieser Stelle existiert somit ein kleiner, aber feiner Unterschied zum Ansatz in JSF.

Das Abschicken des Formulars geschieht durch ein entsprechendes Tag.

```
<html:submit property="submit" value="Submit"/>
```

Diese Angabe bewirkt die Ausgabe eines Submit-Buttons, mit dem die Formularinhalte abgesendet werden können.

Da in Formularen auch in Struts-Anwendungen Fehler auftauchen können (ein Muss-Feld wird nicht befüllt, ungültige E-Mail-Adresse etc.), wird das Tag

```
<html:errors />
```

verwendet, um einen Bereich zu markieren, in dem Fehlermeldungen ausgegeben werden können. Die Fehlermeldungen werden im Beispiel in der `validate`-Methode des Formular-Beans erzeugt und im Formular an der vorgesehenen Stelle als Textstring ausgegeben.

Insgesamt ist auf Seiten der JSP-Erstellung der prinzipielle Unterschied zwischen Struts und JSF nicht allzu groß, es werden hauptsächlich andere Tag-Bibliotheken verwendet. Die Logik der Zuordnung der Werte eines Formulares zu den Beans ist jedoch ein wenig unterschiedlich.

11.5.3 Formular-Beans

Auch wieder vergleichbar mit JSF werden die Daten eines Formulars bei dessen Abschicken in einem Bean gespeichert. In Struts wird dafür der Begriff des *Formular-Beans* verwendet. Ein Formular-Bean entspricht zunächst einmal den Anforderungen eines Formulars in einer HTML-Seite. So kann ein Formular-Bean durchaus auch Eigenschaften aufweisen, die bei einer normalen Modellierung in verschiedenen Entitäten zusammengefasst wären. Ein Formular-Bean stellt somit nicht zwangsläufig eine Entität wie Person, Fahrzeug, Konto etc. dar, sondern spiegelt lediglich die Eingabefelder eines Formulars wider. Ein Formular-Bean ist somit eine Art Sammelcontainer.

```
package com.edu.letter.bean;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionError;
```

```
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public class NewsletterBean extends ActionForm {

  private String firstname;
  private String lastname;
  private String mail;
  private String type;

  /**
   * Getter-Methoden
   */
  public String getFirstname() {
    return firstname;
  }

  public String getLastname() {
    return lastname;
  }

  public String getMail() {
    return mail;
  }

  public String getType() {
    return type;
  }

  /**
   * Setter-Methoden
   */
  public void setFirstname(String string) {
    firstname = string;
  }

  public void setLastname(String string) {
    lastname = string;
  }

  public void setMail(String string) {
    mail = string;
  }

  public void setType(String string) {
    type = string;
  }

  /**
   * führt die Validierung durch
```

```
   */
  public ActionErrors validate(ActionMapping arg0,
                               HttpServletRequest arg1) {
    ActionErrors errors = new ActionErrors();

    if ( firstname==null || firstname.equals("") )
      errors.add( "firstname", new ActionError("errorFirstname") );

    if ( lastname==null || lastname.equals("") )
      errors.add( "lastname", new ActionError("errorLastname") );

    if ( mail==null || mail.equals("") )
      errors.add( "mail", new ActionError("errorMail") );

    return errors;
  }
}
```

Listing 11.2: Das Formular-Bean für den Newsletter

Auch in Struts ist ein Formular-Bean ein JavaBean mit den Eigenschaften und den dazugehörigen Getter- und Setter-Methoden. Wichtig ist jedoch, dass Formular-Beans von der Klasse `ActionForm` ableiten.

Ebenfalls fällt auf, dass eine Methode `validate` existiert. Damit werden die Formularwerte auf ihre Gültigkeit hin überprüft. Im Beispiel wird lediglich überprüft, ob ein Wert eingegeben wurde, falls nicht, wird ein entsprechender Fehler erzeugt und einem Objekt der Klasse `ActionErrors` übergeben. Struts ruft die `validate`-Methode automatisch auf, wenn eine entsprechende Validierung in der Konfigurationsdatei eingestellt wurde.

11.5.4 Controller

Die Controller nehmen eine zentrale und damit auch sehr wichtige Stellung im Struts-Framework ein. In den Controllern findet quasi das eigentliche Geschehen statt, an dieser Stelle wird die Business-Logik durchgeführt bzw. aufgerufen.

```
package com.edu.letter.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class LetterAction extends Action {
```

```
  public ActionForward execute(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest req,
    HttpServletResponse res)
      throws Exception {

    boolean bRet = true;

    // An dieser Stelle kann beispielsweise ein
    // Datenbankzugriff erfolgen, um zu testen, ob
    // eine Mail-Adresse bereits vorliegt.
    // bRet = DBManager.testMailadress( ... );
    if ( bRet==false )
      return mapping.findForward("failure_mail_exists");

    // Falls die Adresse noch nicht vorlag, können an dieser
    // Stelle die neuen Angaben in die Datenbank eingefügt
    // werden.
    // bRet = DBManager.insertAdress( ... );
    if ( bRet==false )
      return mapping.findForward("failure_db_access");

    // Ansonsten lief alles glatt, auf Erfolgsseite
    // weiterleiten.
    return mapping.findForward("success");
  }
}
```

Listing 11.3: Controller-Klasse

Die in Listing 11.3 abgebildete Controller-Klasse ist sehr einfach gehalten. Es ist lediglich der Rahmen angegeben, in dem ein Datenbankzugriff erfolgen kann. Bei Fehlern soll auf eine entsprechende Fehlerseite verwiesen werden, ansonsten soll der Benutzer eine Bestätigungsseite angezeigt bekommen.

Auch in der Controller-Klasse wird an keiner Stelle direkt auf JSP-Seiten verwiesen. Vielmehr wird über ein Mapping ein Bezeichner zurückgeliefert, über den aus der Konfigurationsdatei die entsprechende Seite bestimmt wird.

11.5.5 Struts-Konfigurationsdatei

Die Struts-Konfigurationsdatei lässt sich wiederum sehr gut mit der Anwendungskonfigurationsdatei in JSF vergleichen. Auch in Struts werden darin die Navigation sowie die verwendeten Beans hinterlegt. Im Unterschied zu JSF wird jedoch pro Seite ein Formular-Bean hinterlegt, wobei ein Formular-Bean natürlich in mehreren Seiten verwendet werden kann.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <!-- ======== Form Bean Definitions ======================= -->
  <form-beans>
    <form-bean name="NewsletterForm"
          type="com.edu.letter.bean.NewsletterBean"/>
  </form-beans>

  <!-- ========== Action Mapping Definitions ================ -->
  <action-mappings>
    <action path      = "/register"
            forward   = "/newsletter_names.jsp"
            name      = "NewsletterForm"
            scope     = "session"
            validate  = "true"
            input     = "/newsletter_mail.jsp" />
    <action path      = "/register_2"
            type      = "com.edu.letter.action.LetterAction"
            name      = "NewsletterForm"
            scope     = "session"
            validate  = "true"
            input     = "/newsletter_names.jsp"
    >
      <forward name="success" path="/newsletter_thx.jsp"  />
    </action>

  </action-mappings>

  <!-- ========== Message Resources Definitions ============== -->
  <message-resources parameter="ApplicationResources"/>

</struts-config>
```

Listing 11.4: Konfigurationsdatei struts-config.xml

In der Konfigurationsdatei wird ein Formular-Bean `Newsletterform` definiert, das in allen drei Seiten des Newsletter-Wizards verwendet wird. Es müssen somit nicht immer alle Eigenschaften eines Formular-Beans in einem Html-Formular vorhanden sein. Diejenigen Eigenschaften, die im Formular vorhanden sind und durch den Benutzer befüllt werden, werden beim Abschicken eines Fomulars automatisch in das Formular-Bean übertragen.

Basierend auf Bezeichnern, die in den Controllern zurückgeliefert werden, wird dann auf eine konkrete JSP-Seite verwiesen. Dies ist die einzige Stelle, an der der vollständige Dateiname einer JSP-Seite auftaucht.

Über das Tag

```
<message-resources parameter="ApplicationResources" />
```

wird eine Ressourcendatei mit eingebunden, in der die Fehlertexte hinterlegt sind, die in der Formular-Bean-Klasse angezogen werden. Wichtig ist es auch hier, dass die Ressourcendatei im Klassenpfad der Anwendung abgelegt ist.

11.6 Verwendung der Integrationsbibliothek

Nachdem in den letzten Kapiteln die Struts-Anwendung vorgestellt wurde (der vollständige Quellcode ist auf der beigelegten CD zu finden), wird in den folgenden Kapiteln die Anwendung schrittweise mit Hilfe der Integrationsbibliothek umgebaut.

Im Beispiel, das auf der CD mit enthalten ist, wurde dazu ein zweites Projekt aufgesetzt, um den Vergleich zwischen der Original-Struts-Anwendung und der neuen Struts-JSF-Anwendung erkennen zu können.

Da zum Zeitpunkt der Drucklegung des Buches noch keine Final-Version der Integrationsbibliothek verfügbar war, basieren die Beispiele auf einem so genannten *Nightly Build*. Ein Nightly Build ist eine Zwischenversion, die regelmäßig bei der Erstellung von Software gebaut wird. Ein Nightly Build sollte auf keinen Fall in einer produktiven Umgebung eingesetzt werden, da die Stabilität sowie der Funktionsumfang selbst noch nicht vollständig ausgetestet wurden. Für die folgenden Demonstrationszwecke funktioniert das Nightly Build vom 9. April 2004 jedoch ohne Probleme.

Ziel des Einsatzes der Struts-Faces-Integrationsbibliothek ist es primär, die Vorteile, die JSF bietet, zu nutzen, ohne jedoch (aufwändig) entwickelte Funktionalität von Struts komplett neu programmieren zu müssen. Die vorhandene Controller-Logik soll daher so weit wie möglich erhalten bleiben, die Komponenten in den JSP-Seiten jedoch sollen zu JSF migriert werden.

Die Migration einer Struts-Anwendung verläuft zumeist in folgenden Schritten:

1. Erweitern des Klassenpfades um die benötigten Bibliotheken der Integrationsbibliothek
2. Erweitern des Klassenpfades um die benötigten JSF-Bibliotheken
3. Erweitern des Klassenpfades um die Bibliotheken der JSTL
4. Anpassen des Deployment Deskriptors

5. Anpassen der JSP-Seiten
6. Anpassen der Struts-Konfigurationsdatei

Im Folgenden werden die einzelnen erforderlichen Arbeitsschritte ausführlich besprochen und anhand des konkreten Beispiels demonstriert.

Erweitern des Klassenpfades

Zunächst wird der Klassenpfad um die zentrale Bibliothek der Integrationsbibliothek selbst erweitert. Das *lib*-Verzeichnis der Integrationsbibliothek beinhaltet dabei die jar-Datei *struts-faces.jar*. Diese muss in das Projekt in das Verzeichnis *WEB-INF/lib* kopiert und gegebenenfalls in den Build-Pfad mit aufgenommen werden.

Um auf JSF-Funktionalität zugreifen zu können, müssen aus den Verzeichnissen der JavaServer Faces Referenzimplementierung die Dateien *jsf-api.jar* und *jsf-impl.jar* ebenfalls in das *WEB-INF/lib*-Verzeichnis kopiert werden.

Um auf die JSTL zugreifen zu können, müssen ebenfalls zwei weitere jar-Bibliotheken übernommen werden: *jstl.jar* und *standard.jar*.

Anpassen des Deployment Deskriptors

Der Deployment Deskriptor ist bei reinen Struts-Anwendungen nur auf die Verarbeitung hinsichtlich des Struts-Servlets eingestellt. Daher ist das Faces-Servlet zusätzlich zu hinterlegen sowie ein dazugehöriges Mapping.

```
<!-- JavaServer Faces Servlet Configuration -->
<servlet>
  <servlet-name>faces</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

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

Listing 11.5: Erweiterung des Deployment Deskriptors web.xml

Die Eintragungen entsprechen den üblichen Faces-Eintragungen, wie sie auch in reinen JSF-Anwendungen zu finden sind. Wird das `load-on-startup`-Tag verwendet, ist darauf zu achten, dass im Struts-Servlet der Wert auf `2` gesetzt wird, so dass das Faces-Servlet zuerst initialisiert wird.

Anpassen der JSP-Seiten

Das Anpassen der einzelnen JSP-Seiten ist mit Sicherheit das aufwändigste Arbeitspaket im gesamten Umstellungsprozess. Aufgrund der Tatsache, dass jede einzelne JSP-Seite angepasst werden muss, kann dies bei größeren Webanwendungen mit sehr viel Aufwand verbunden sein. Es müssen *alle* Struts-spezifischen Tags gegen Faces-Tags, Tags der JSTL und Tags der Integrationsbibliothek ausgetauscht werden.

Anstelle des Imports der Struts-Tags im Kopf einer jeden JSP-Seite müssen nun folgende Taglibs importiert werden:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>
<%@ taglib prefix="s"
      uri="http://jakarta.apache.org/struts/tags-faces" %>
```

Listing 11.6: Verwendete Taglibs

Zu beachten ist, dass zusätzlich zu den »üblichen« JSF-Tags einige Tags aus den Integrationsbibliotheken verwendet werden. Diese haben das Präfix `s`. Diese Tags ersetzen einige Struts-Html-Tags, zu denen es keine direkte Entsprechung in JavaServer Faces gibt. Zum Beispiel wird aus

```
<html:form action="/register.do">
```

das Tag

```
<s:form action="/register">
```

Des Weiteren sind sämliche Tags für die Datenein- und -ausgabe mit JSF-Tags zu ersetzen. So wird aus dem Eingabefeld für den Vornamen

```
<html:text property="firstname" size="16" maxlength="16"/>
```

das JSF-Tag

```
<h:inputText value="#{NewsletterForm.firstname}" />
```

Dabei fällt auf, dass im `inputText`-Tag wiederum ein Ausdruck der JSF Expression Language verwendet wird. Ebenso ist der Name `NewsletterForm` mitanzugeben, was in der Struts-Darstellung nicht notwendig war.

Sollen weitere Struts-spezifische Tags verwendet worden sein (z.B. die Logik-Tags), so sind diese durch die entsprechenden Tags der JSTL zu ersetzen.

Last but not least darf auch nicht vergessen werden, die komplette Seite in ein `view`-Tag zu setzen, da ansonsten keine korrekte Darstellung der Seite erfolgt.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib
   uri="http://jakarta.apache.org/struts/tags-faces" prefix="s" %>

<html>
<head>
<title>Struts-Newsletter-Anmeldung</title>
<link rel="stylesheet" type="text/css" href="default.css" />
</head>
<body bgcolor="white"><p>

<center>
  <br>
  <h3>Newsletter-Anmeldung</h3>
</center>

<f:view>
  <s:form action="/register">

  <table border="1" cellpadding="3" frame="box" rules="none"
    align="center">
  <tr>
    <th colspan="2">Newsletter-Anmeldung 1/3</th>
  </tr>
  <tr>
    <td>Vorname:</td>
    <td>
      <h:inputText value="#{NewsletterForm.firstname}" />
    </td>
  </tr>
  <tr>
    <td>Nachname:</td>
    <td>
      <h:inputText value="#{NewsletterForm.lastname}" />
    </td>
  </tr>
  <tr>
    <td>E-Mail:</td>
    <td>
      <h:inputText value="#{NewsletterForm.mail}" />
    </td>
  </tr>
  <tr>
    <td colspan="2" class="error">
      <s:errors />
    </td>
  </tr>
```

```
    <tr>
      <td colspan="1"><br />
        <h:commandButton action="success" value="Submit" />
      </td>
      <td colspan="1"><br />
        <h:commandButton action="reset" value="Reset" />
      </td>
    </tr>
    </table>

  </s:form>
  </f:view>
  <br>

  </body>
  </html>
```

Listing 11.7: Die Index-Seite basierend auf Struts und JSF

Listing 11.7 zeigt die Indexseite einer auf JSF umgestellten Struts-Seite. Es ist darauf zu achten, sämtliche Struts-Tags gegen die entsprechenden Tags aus JSF, der JSTL und der Integrationsbibliothek auszutauschen.

Anpassen der Struts-Konfigurationsdatei

Als letzter Schritt ist die Struts-Konfigurationsdatei *struts-config.xml* anzupassen, so dass der korrekte Struts-Faces Requestprozessor angezogen werden kann, der für eine korrekte Verarbeitung der Requests zuständig ist.

```
<controller>
  <set-property property="processorClass"
value="org.apache.struts.faces.application.FacesRequestProcessor"/>
</controller>
```

Listing 11.8: Hinterlegung des Requestprozessors

Hierbei gibt es jedoch eine Besonderheit zu beachten. Wird in der Anwendung die Template Tag-Bibliothek *Tiles* eingesetzt, ist folgender Eintrag notwendig:

```
<controller>
  <set-property property="processorClass" value=
"org.apache.struts.faces.application.FacesTilesRequestProcessor"/>
</controller>
```

Listing 11.9: Eintragung bei Verwendung von Tiles

Als weitere Änderung ist in der Struts-Konfigurationsdatei noch anzupassen, dass sämtliche Requests zunächst über das Faces-Servlet geleitet werden. Dies erfolgt, indem ein `/faces` vor jede Pfadangabe gestellt wird und somit das entsprechende Mapping den Request auf das Faces-Servlet umleitet.

```
<action-mappings>
  <action  path      = "/register"
           forward   = "/faces/newsletter_type.jsp"
           name      = "NewsletterForm"
           scope     = "session"
           validate  = "true"
           input     = "/index.jsp" />
  <action  path      = "/register_2"
           type      = "com.edu.letter.action.LetterAction"
           name      = "NewsletterForm"
           scope     = "session"
           validate  = "true"
           input     = "/newsletter_type.jsp">
    <forward name="success" path="/faces/newsletter_thx.jsp" />
  </action>
</action-mappings>
```

Listing 11.10: Auszug aus der Struts-Konfigurationsdatei

Wie in Listing 11.10 zu sehen, genügt es im Normalfall, in jedem `forward`-Element ein `/faces` voranzustellen. Natürlich kann auch an dieser Stelle das *Extension Mapping* verwendet werden, so dass das Voranstellen des */faces* nicht mehr notwendig ist.

Sämtliche weiteren Struts-Klassen müssen im Normalfall nicht weiter angepasst werden und sollten auch noch nach einer Migration zu JSF problemlos funktionieren.

11.7 Fazit

Struts und JavaServer Faces sind keine Gegenspieler, von denen auch langfristig gesehen nur einer überleben kann – das sollte in diesem Kapitel deutlich geworden sein.

Struts hat eine etwas andere Zielrichtung als Framework verglichen mit JavaServer Faces, bei Kombination beider Frameworks können sehr leistungsfähige Anwendungen realisiert werden. Struts wird sich mit Sicherheit weiterentwickeln, wobei hier der Trend zur Verwendung der JSTL gegeben ist. Daher zielt auch der Einsatz der Integrationsbibliothek darauf ab, mittels JSF- und JSTL-Tags die Oberfläche zu definieren, dabei aber die Funktionalität in Form der Controller- und Beanklassen beizubehalten.

Der Aufwand, eine reine Struts-Anwendung auf JSF zu migrieren, ist dabei überschaubar. Der meiste Aufwand entsteht sicherlich in der Umsetzung der einzelnen Seiten. Da jedoch die komplette Funktionalität beibehalten werden kann, ist diese Vorgehensweise durchaus akzeptabel.

Anhang

A.1 Inhalt der CD

Um einen schnellen Start in die Entwicklung mit JavaServer Faces zu ermöglichen, wurden die meisten Programme und Werkzeuge der CD beigelegt. Zudem sind die beigelegten Pakete auf ihre Kompatibilität untereinander getestet und funktionieren in dieser Konstellation.

A.2 Konfigurationsdatei faces-config.xml

Die Beispiele im Buch basieren im Wesentlichen auf zwei Projekten, die jeweils eine eigene Anwendungskonfigurationsdatei aufweisen. Dabei sind sämtliche Kurzbeispiele und Codefragmente in einem Projekt zusammengefasst, ebenso wie die Beispielanwendung JSF-Weblog ein eigenes Projekt darstellt.

faces-config.xml für die Kurzbeispiele

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
```

```
<faces-config>

  <!-- ############################################# -->
  <!-- Navigationseintragungen fuer die Kurzbeispiele -->
  <!-- ############################################# -->

  <navigation-rule>
    <from-view-id>/kap_05/eingabe.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/kap_05/ausgabe.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/kap_06/userlogin.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/kap_06/userlogin_ok.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>failed</from-outcome>
      <to-view-id>/kap_06/userlogin_failed.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>content_2</from-outcome>
      <to-view-id>/kap_06/frame_content_2.jsp</to-view-id>
      <redirect />
    </navigation-case>
    <navigation-case>
      <from-outcome>content_1</from-outcome>
      <to-view-id>/kap_06/frame_content.jsp</to-view-id>
      <redirect />
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/kap_10/register.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/kap_10/register_thx.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <!-- ############################################# -->
  <!-- Managed-Bean-Deklaration fuer Kurzbeispiele    -->
  <!-- ############################################# -->
```

```
<managed-bean>
  <managed-bean-name>Person</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.PersonBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>city</property-name>
    <value>München</value>
  </managed-property>
  <managed-property>
    <property-name>zip</property-name>
    <value>#{initParam.defaultZip}</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>Customer</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.CustomerBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Visitor</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.VisitorBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Demologin</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.LoginBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Pricelist</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.PricelistBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Survey</managed-bean-name>
  <managed-bean-class>
```

```
      com.edu.jsf.bsp.bean.SurveyBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>CreditCardHolder</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.bsp.bean.CreditCardHolderBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Adress</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.bsp.bean.AdressBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>CourseRegister</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.bsp.bean.RegisterBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Payment</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.bsp.bean.PaymentBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>ParticipantList</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.bsp.bean.ParticipantListBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Calculate</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.bsp.bean.CalculateBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
```

```
<managed-bean>
  <managed-bean-name>Interest</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.InterestBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Newslist</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.bsp.bean.NewslistBean
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<!-- ######### -->
<!-- Validator -->
<!-- ######### -->
<validator>
  <description>Validator for Mailadresses</description>
  <validator-id>MailValidator</validator-id>
  <validator-class>
    com.edu.jsf.bsp.validate.MailValidator
  </validator-class>
</validator>
<validator>
  <description>Validator for Credit Cards</description>
  <validator-id>CreditCardValidator</validator-id>
  <validator-class>
    com.edu.jsf.bsp.validate.CreditCardValidator
  </validator-class>
</validator>

<!-- ################# -->
<!-- Message-Resourcen -->
<!-- ################# -->
<application>
  <message-bundle>mymessages</message-bundle>
</application>

<!-- ################################## -->
<!-- Benutzerspezifische UI-Komponenten -->
<!-- ################################## -->
<component>
  <component-type>UsageBar</component-type>
  <component-class>
    com.edu.jsf.bsp.tag.UIUsageBar
  </component-class>
</component>
<component>
```

```
    <component-type>CreditCardArea</component-type>
      <component-class>
        com.edu.jsf.bsp.tag.UICreditCard
     </component-class>
  </component>
  <component>
    <component-type>TestComp</component-type>
    <component-class>
      com.edu.jsf.bsp.tag.testcomp
    </component-class>
  </component>

  <!-- ############################## -->
  <!-- Benutzerspezifische Konverter -->
  <!-- ############################## -->
  <converter>
    <description>
      konvertiert eine Mailliste in einen Vector.
    </description>
    <converter-id>maillist</converter-id>
    <converter-class>
      com.edu.jsf.bsp.convert.MailListConverter
    </converter-class>
  </converter>

  <render-kit>
    <renderer>
      <component-family>Graph</component-family>
      <renderer-type>UsageBar</renderer-type>
      <renderer-class>
        com.edu.jsf.bsp.tag.UsageBarRenderer
      </renderer-class>
    </renderer>
    <renderer>
      <component-family>CreditCardArea</component-family>
      <renderer-type>CreditCard</renderer-type>
      <renderer-class>
        com.edu.jsf.bsp.tag.CreditCardRenderer
      </renderer-class>
    </renderer>
  </render-kit>

  <lifecycle>
  <phase-listener>
    com.edu.jsf.bsp.listener.MyLifecycleListener</phase-listener>
  </lifecycle>

</faces-config>
```

Listing A.1: faces-config.xml für die Kurzbeispiele

faces-config für die Beispielanwendung JSF-WebLog

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

  <!-- ################################################## -->
  <!-- Navigationseintragungen fuer die WebLog-Anwendung -->
  <!-- ################################################## -->

  <navigation-rule>
    <from-view-id>/home.jsp</from-view-id>
    <navigation-case>
      <from-outcome>show_register</from-outcome>
      <to-view-id>/register.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/register.jsp</from-view-id>
    <navigation-case>
      <from-outcome>save</from-outcome>
      <to-view-id>/home.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>show_weblog</from-outcome>
      <to-view-id>/show_weblog.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>show_poster</from-outcome>
      <to-view-id>/show_poster.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>show_guestbook</from-outcome>
      <to-view-id>/show_guestbook.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>edit_weblog</from-outcome>
      <to-view-id>/edit_weblog.jsp</to-view-id>
    </navigation-case>
```

```
    <navigation-case>
      <from-outcome>edit_intro</from-outcome>
      <to-view-id>/edit_intro.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>edit_poster</from-outcome>
      <to-view-id>/edit_poster.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>home</from-outcome>
      <to-view-id>/home.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <!-- ################################################## -->
  <!-- Managed-Bean-Deklaration fuer die WebLog-Anwendung -->
  <!-- ################################################## -->
  <managed-bean>
    <managed-bean-name>Login</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.weblog.bean.LoginBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Register</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.weblog.bean.RegisterBean
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Blog</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.weblog.dbbean.Blog
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>Poster</managed-bean-name>
    <managed-bean-class>
      com.edu.jsf.weblog.dbbean.Poster
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
```

```
<managed-bean>
  <managed-bean-name>Bloglist</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.weblog.bean.Bloglist
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Blogentry</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.weblog.dbbean.BlogEntry
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>Blogentrylist</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.weblog.bean.Blogentrylist
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>User</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.weblog.dbbean.User
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>GuestbookentryList</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.weblog.bean.Guestbookentrylist
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>GuestbookEntry</managed-bean-name>
  <managed-bean-class>
    com.edu.jsf.weblog.dbbean.Bookentry
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>ItemList</managed-bean-name>
  <managed-bean-class>
```

```
        com.edu.jsf.weblog.bean.ItemList
      </managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>Channel</managed-bean-name>
      <managed-bean-class>
        com.edu.jsf.weblog.bean.ChannelBean
      </managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <!-- ################################## -->
    <!-- Benutzerspezifische UI-Komponenten -->
    <!-- ################################## -->
    <component>
      <component-type>InputErrorText</component-type>
      <component-class>
        com.edu.jsf.weblog.tag.UIInputError
      </component-class>
    </component>
    <component>
      <component-type>OutputItem</component-type>
      <component-class>
        com.edu.jsf.weblog.tag.UIOutputItem
      </component-class>
    </component>

    <!-- ############################# -->
    <!-- Benutzerspezifische Konverter -->
    <!-- ############################# -->
    <render-kit>
      <renderer>
        <component-family>javax.faces.Input</component-family>
        <renderer-type>InputErrorText</renderer-type>
        <renderer-class>
          com.edu.jsf.weblog.tag.InputErrorRenderer
        </renderer-class>
      </renderer>
      <renderer>
        <component-family>javax.faces.Data</component-family>
        <renderer-type>ItemList</renderer-type>
        <renderer-class>
          com.edu.jsf.weblog.tag.ItemListRenderer
        </renderer-class>
      </renderer>
    </render-kit>
  </faces-config>
```

Listing A.2: faces-config.xml für die Weblog-Anwendung

A.3 Webadressen und -ressourcen

Im Internet existiert eine Fülle von Informationen zum Thema JavaServer Faces. Keine Liste kann daher auch nur im Überblick das aktuelle Angebot abbilden. Nachfolgend daher nur einige Adressen, über die ein Einstieg möglich ist und von dort aus zu anderen Seiten navigiert werden kann.

- *http://java.sun.com/j2ee/javaserverfaces/*
  Offizielle Seite von Sun zu JavaServer Faces
- *http://forum.java.sun.com/forum.jsp?forum=427*
  Das offizielle Forum von Sun zu JavaServer Faces. Es können dort Fragen zum Thema JSF gestellt werden. Oftmals finden auch sehr interessante und lehrreiche Diskussionen zu Architekturfragen statt.
- *http://www.jcp.org/en/jsr/detail?id=127*
  Der Java Specification Request Nummer 127, der die Basis für die JavaServer Faces Spezifikation bildet
- *http://sourceforge.net/projects/myfaces/*
  Open-Source-Implementierung der JavaServer Faces Spezifikation
- *http://www.jamesholmes.com/JavaServerFaces/*
  Homepage der JavaServer Faces Console
- *http://jakarta.apache.org/struts/*
  Offizielle Homepage des Projektes Apache Struts
- *http://www.jsf-forum.de*
  Deutschsprachige Seite zum Thema JSF

A.4 Faces Console

Die Faces Console wurde von James Holmes entwickelt und kann als freie Software lizenzkostenfrei in Projekten eingesetzt werden. Durch Plug-Ins für die gängigsten Entwicklungsumgebungen ist es möglich, die Faces Console in nahezu jedem Projekt einsetzen zu können. Die Faces Console stellt dabei eine Oberfläche sowohl für die Anwendungskonfigurationsdatei *faces-config.xml* bereit wie auch eine Oberfläche für Tag-Libs. Somit ist es möglich, mit Hilfe einer komfortablen Oberfläche die Eintragungen in den Dateien vorzunehmen, anstatt dies z. B. mit Hilfe eines einfachen Editors zu erledigen.

Die Faces Console kann über den Link, der in (12.3 Webadressen und -ressourcen) aufgelistet ist, kostenfrei heruntergeladen werden. Die Faces Console hat dabei eine eigene Lizenzbestimmung, in der u.a. geregelt ist, dass dies kein Open Source ist, ein Anwender jedoch jederzeit kostenfrei das Programm verwenden kann.

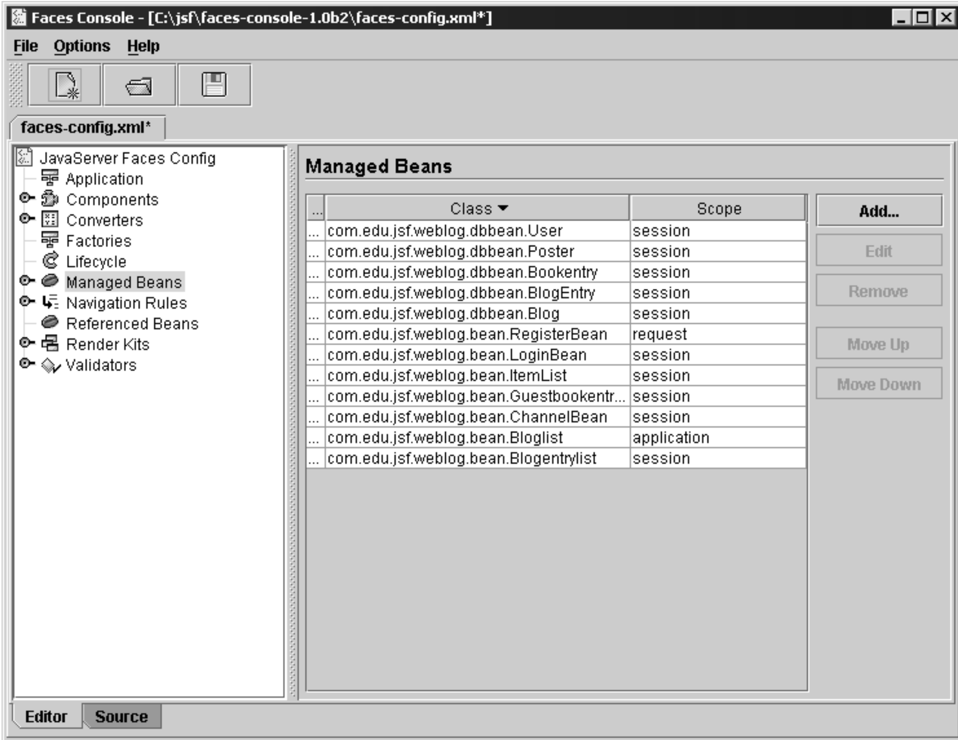


Abbildung A.4: Faces Console

Die Faces Console ist in der Version 1.0 ebenfalls der CD beigelegt.

Index

A

Action-Events 163
  Registrieren eines Actionlisteners 165
ActionListener 390
  Implementieren 163
Adapter-Klassen 36
Ant-Skript 414
Anwendungsfall 245
Anwendungskonfigurationsdatei 78, 80, 95
Apache Torque 244
Application Server 13, 403
Apply Request Values 70
Architektur 30
  Architekturmodell 1 30
  Architekturmodell 2 31
Architekturmodelle 17

B

Backing Beans 105, 114
Bean Management 85, 109
Beispielapplikation 243
benutzerdefinierte Komponenten 307
Benutzerspezifische Konverter 366
Benutzerspezifische Validatoren 351
Best Practices 15, 55, 371
Bezeichner
  Bezeichner auf Clientseite 104
  Komponentenbezeichner 104
Blogging-Community 243
Blogging-Engine 243
Build-Pfad 81
Business-Tier 22

C

Cancel-Buttons 418
Cascacing Style Sheets 398
Channels 287
Client Site Saving 171
Community 56
Component Binding 127
Component Tree 156
Contentprovider 284, 286
Convenience-Klasse 104
Cookie 382
Cookies 171
CSS 398

D

Datenbank 13
  MySQL 13
Datenbankanbindung 254
Datenbankschema 255, 257
Datenbanktreiber 256
Datenkonvertierung 142
Datenmodellierung 253
DateTimeConverters 148
Delegated Implementation 308
Demoanwendungen 49
Deployment Deskriptor 20, 78, 94
Design Pattern 45
Designpattern
  Model-View-Controller 31
  separation of concerns 32
Dialoggedächtnis 69
DoubleRange-Validator 136

E

Eclipse 13, 75
Einführung 53
  JavaServer Faces 53
EJB 402
EJB-Container 402
EJBs 65
Enterprise Information Services 403
Enterprise Java Beans 402
Enterprise JavaBeans 394
Entity-Beans 406
Entity-Relationship-Diagramm 253
Entwicklungsumgebung
  Build-Pfad 81
  Eclipse 13, 75
  PlugIn 76
  Sysdeo 76
Entwurfsmuster 45, 79
  FrontController-Pattern 79
Eventhandling 161
Eventlistener 97
Eventobjekt 162
Eventtypen 162
Expression Language
  JSP Expression Language 39
Extension Mapping 95
Extension Mappings 387
ExternalContext 150

F

Facade-Pattern 397
FacesContext 69, 149
Faces-Konsole 96
FacesMessage 139
Faces-Request 67
Faces-Servlet 94
Fehlermeldungen 136, 354
  Darstellung 131
  Fehlermeldungen überschreiben 138
Framework 11, 54
Frontcontroller 94
FrontController Servlet 292
FrontController-Pattern 79
FrontController-Servlet 273, 372

G

Grid 226
Gültigkeitsbereich 111, 383

H
html_basic Taglib 97
HTMLCommandButton-Komponente 187
HTMLCommandLink-Komponente 191
HtmlInputHidden-Komponente 200
HTMLInputSecret-Komponente 199
HtmlInputTextArea-Komponente 200
HTMLInputText-Komponente 196
HTML-Komponenten 183
HtmlOutputFormat-Komponente 220
HtmlOutputLabel-Komponente 219
HtmlOutputLink-Komponente 221
HtmlOutputText-Komponente 217
HtmlPanelGrid-Komponente 226
HtmlPanelGroup-Komponente 233
HTMLSelectManyCheckbox-Komponente 213
HTMLSelectManyListbox-Komponente 216
HTMLSelectManyMenu-Komponente 214
HTMLSelectOneListbox-Komponente 208
HTMLSelectOneMenu-Komponente 209
HTMLSelectOneRadio-Komponente 206
Hyperlinks 401

I
IDBroker-Service 257
Implementation 62
Implizite Variablen 24
Initialisierungsparameter 117
Installation
  Demoanwendungen 49
  Vorbereitungen 49
Integrationsbibliothek 15, 422
Internationalisierung 141, 189
Invoke Application 71

J
J2EE-Spezifikation 11
J2EE-Standard 63
Jakarta-Tomcat 13
Java Collections Frameworks 215
Java Community Process 62
Java-Applets 17
JavaBeans 82
  Verwendung von JavaBeans 25
JavaServer Faces
  Java Community Process 62
  Specification Requests 127 59
  Standard 59
JavaServer Pages
  Grundlagen 23
Jboss-Server 404
jsf_core Taglib 97
JSF-Weblog
  Beispielanwendung 244
JSP Expression Language 39
JSP-Architektur-Modelle 30
Jsp-Elemente
  Aktion 24
  Ausdruck 24
  Deklaration 24
  Direktive 24
  Skriplet 24
JSTL 17, 32, 37
JSTL SQL 375

K
Klassenschema 248
Komponenten
  gruppieren 233
Komponentenarchitektur 102
Komponentenbaum 106, 181
Komponentenmodell 93
Komponentensichtbarkeit 38 7
Konfigurationsdatei
  Deployment Deskriptor 20
Konfigurationsdateien 78, 94
  Anwendungskonfiguration sdatei 78
  Deployment Deskriptor 78
Konverter 307, 366
Konvertierung
  DateTimeConverters 148
  Konvertierungsattribute 1 47
  NumberConverters 148

L
Längen-Validator 133
lazy loading 113, 381
LongRange-Validator 135
Luhn-Verfahren 358

M
Managed Bean 85
Managed Beans 372
  Initialisierungsparameter 117
Managed-Bean Konzept 109
Managed-Beans
  Einschränkungen 113
  Verwendung 110
  Vorteile 112
Marketplace 56
Method Binding 162, 397
MethodBinding 124
Modellobjekte 82
Modellobjekten 109
Modellsicht 308
Modellupdate 315
Model-View-Controller 31, 53
Model-View-Controller-Entwurfsmuster 45
MyFaces 63

N
Namensraum 98
Namensraumes 36, 104
NamingContainer 105
Navigation
  Dynamische Navigation 158
  Festlegung 89

Funktionsweise 156
Navigationsregeln 153
Reihenfolge 157
Rückgabewerte 154
Standardverhaltens 155
Statische Navigation 158
Wildcards 154
Navigationsfälle 153
Navigationskonzept 151
Navigationsregeln 153
NumberConverters 148

O

O/R-Mapping 13
Objektorientierte Modellierung 245
Objekt-Relationales-Mapping 244
Open Source 62

P

Peerklassen 261
Phase Events 169
PhaseListener 170, 297
PlugIn 13
Sysdeo 13, 76
Portal 54
Portale 56
Präsentationssicht 308
Prefix-Mapping 95
Prefix-Mappings 387
Process Validation 70
Prozentanzeige-Komponente 309

R

Referenzimplementierung 63
Render Response 71
Renderer 93, 107, 182, 307
Rendering
delegated implementation 108
direct implementation 108
Rendering-Modell 93, 107
Render-Kit 107, 181
Request 18
Faces-Request 67
Non-Faces-Request 67
Required-Validator 133
Resourcebundles 175
Ressourcendatei 137, 189
RessourcesBundles 141
Restore View 70
Rollenteilung 59
RSS 284
RSS-Channel 284
RSS-Feed 284
RSS-Newsfeed 284
Rss-Newsfeed 274
Rückgabewert 153
Rückgabewerte 154

S

Screenflow 249
Screenflows 245
Seitenverlinkung 401
separation of concerns 32
Server Site Saving 171
Servlet-API 150
Servlet-Chaining 22
Servlet-Container 17, 18
ServletContextListener 273, 372
Servlet-Mapping 94
Servlets 17
Session 381
Session-Beans 406
Singleton 174, 397
Software-Architekturmodell 57
Spezifikation 62
Standard 59
Standard Lebenszyklus 68, 69
Apply Request Values 70
Invoke Application 71
Render Response 71
Restore View 70
Update Model Values 71
Standard Lebenzyklus
Process Validation 70
Standard-Render-Kit 320
State Saving 171, 309, 322
Client Site Saving 171
Server Site Saving 171
Struts 421
Struts-Faces 15
Stylesheets 398
Subviews 184
Systemvoraussetzungen 12

T

Tag-Bibliotheken 32, 97
Core-Tags 98
Einbinden 98
HTML-Tags 100
Tag-Library-Deskriptor 35, 313
Taglibs 32
benutzerdefinierten Tags 34
Funktionsweise 33
Namensraumes 36
Tag-Library-Deskriptor 35
Timeout 382
TLD 313
Toolunterstützung 64
Torque 13, 244, 254

U

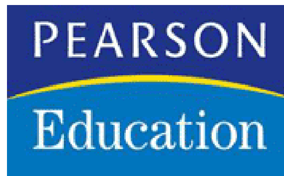
UICommand-Komponente 187
UIData-Komponente 234
UIForm-Komponente 185
UIGraphic-Komponente 224
UIInputError-Komponente 279
UIInput-Komponente 194
UI-Komponenten 53, 179
Attribute 180
Grundaufbau 180
UI-Komponentenmodell 102
UIMessage-Komponente 222
UIOutput-Komponente 217
UIPanel-Komponente 226
UISelectBoolean-Komponente 201
UISelectMany-Komponente 210
UISelectOne-Komponente 203
Update Model Values 71
Use-Cases 245

V

Validatoren 97
DoubleRange-Validator 136
Fehlermeldungen 137
Längen-Validator 133
LongRange-Validator 135
Required-Validator 133
Validierung 130
ValueBinding 105, 124, 174
Value-Change-Events 166
ValueHolder Interface 339
VariableResolver 374
View-Objekt 69
Views 184
Voraussetzungen
des Lesers 12
Systemvoraussetzungen 12

W
Web Services Developer Pack 49
Webanwendung 54
Webfrontend 54
Webserver 17
Wildcards 154
Wizard 388

Z
Zielgruppe 61
Zinseszinsrechnung 407
Zustandsspeicherung 171, 309, 322



Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.
Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!
Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Platzierung im Internet, in Intranets, in Extranets anderen Websites, der Veränderung, des Weiterverkaufs und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.
Bei Fragen zu diesem Thema wenden Sie sich bitte an:
mailto:info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen