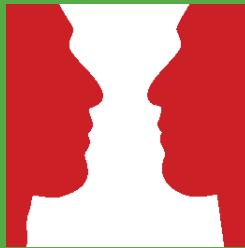


JAVA SERVER FACES 2.0



bernd MÜLLER

EIN ARBEITSBUCH FÜR DIE PRAXIS



2. Auflage

HANSER



Müller

JavaServer Faces 2.0



Bleiben Sie einfach auf dem Laufenden:
www.hanser.de/newsletter

Sofort anmelden und Monat für Monat
die neuesten Infos und Updates erhalten.

Bernd Müller

JavaServer Faces 2.0

Ein Arbeitsbuch für die Praxis

HANSER

Prof. Dr. Bernd Müller
Ostfalia Hochschule für angewandte Wissenschaften
Hochschule Braunschweig/Wolfenbüttel – Fakultät für Informatik
bernd.mueller@ostfalia.de

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht, auch nicht für die Verletzung von Patentrechten und anderen Rechten Dritter, die daraus resultieren könnten. Autor und Verlag übernehmen deshalb keine Gewähr dafür, dass die beschriebenen Verfahren frei von Schutzrechten Dritter sind.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

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.

Dieses Werk ist urheberrechtlich geschützt. Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2010 Carl Hanser Verlag München Wien (www.hanser.de)
Lektorat: Margarete Metzger
Herstellung: Irene Weilhart
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Stephan Rönigk
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-41992-6

Inhaltsverzeichnis

Vorwort zur 2. Auflage	XI
Vorwort	XIII
1 Einleitung	1
1.1 Ziel dieses Buches	1
1.2 Der Leser	3
1.3 Das Buch im Netz	4
1.4 Verwendete Software	5
1.5 Aufbau des Buches	6
1.6 JSF-Versionen vor 2.0	7
2 Motivation	9
2.1 Was sind JavaServer Faces?	9
2.2 JavaServer Faces und Zukunftssicherheit	10
2.3 Neuerungen in JavaServer Faces 2.0	11
2.4 Spezifikationen und Implementierungen	12
2.5 Die erste JSF-Anwendung: Tic-Tac-Toe	12
3 Die Anwendung <i>Comedians</i>	21
3.1 JSF-Seiten	22
3.2 Managed Beans	25
3.3 JPA	32
4 JavaServer Faces im Detail	35
4.1 Bearbeitungsmodell einer JSF-Anfrage	35

4.1.1	Wiederherstellung des Komponentenbaums	38
4.1.2	Übernahme der Anfragewerte	39
4.1.3	Validierung	40
4.1.4	Aktualisierung der Modellobjekte	41
4.1.5	Aufruf der Anwendungslogik	41
4.1.6	Rendern der Antwort	42
4.2	Expression-Language	43
4.2.1	Syntax	44
4.2.2	Bean-Properties	44
4.2.3	Vordefinierte Variablen	47
4.2.4	Vergleiche, arithmetische und logische Ausdrücke	49
4.2.5	Methodenaufrufe und Methodenparameter	52
4.2.6	Verwendung der Expression-Language in Java	53
4.3	Managed Beans	54
4.3.1	Architekturfragen	55
4.3.2	Automatische Verwaltung von Managed Beans	56
4.3.3	Initialisierung	58
4.3.4	Komponentenbindung	64
4.3.5	Java-EE-5-Annotationen	68
4.3.6	JSF-Annotationen	68
4.4	Validierung und Konvertierung	72
4.4.1	Standardkonvertierer	72
4.4.2	Konvertierung von Kalenderdaten und Zahlen	77
4.4.3	Konvertierung von Aufzählungstypen	81
4.4.4	Anwendungsdefinierte Konvertierer	84
4.4.5	Standardvalidierer	87
4.4.6	Validierungsmethoden	92
4.4.7	Anwendungsdefinierte Validierer	94
4.4.8	Eingabekomponenten und das <code>immediate</code> -Attribut	95
4.4.9	Bean-Validierung mit JSR 303	98
4.4.10	Fehlermeldungen	105
4.5	Event-Verarbeitung	114
4.5.1	JSF-Events und allgemeine Event-Verarbeitung	115

4.5.2	Action-Events	117
4.5.3	Befehlskomponenten und das <code>immediate</code> -Attribut	121
4.5.4	Value-Change-Events	122
4.5.5	Data-Model-Events	127
4.5.6	Phase-Events	128
4.5.7	System-Events	130
4.6	Navigation	133
4.6.1	Implizite Navigation	134
4.6.2	View-to-View-Regeln	135
4.6.3	Regeln für mehrere Seiten	138
4.6.4	Regeln für Action-Methoden	139
4.6.5	Regeln zur bedingten Navigation	139
4.6.6	Redirects	140
4.6.7	Verweise auf Nicht-JSF-Seiten	141
4.6.8	View-Parameter und Lesezeichen	143
4.6.9	Die technische Sicht	146
4.7	Internationalisierung	147
4.7.1	Lokalisierung	148
4.7.2	Dynamische und explizite Lokalisierung	155
4.7.3	Klassen als Resource-Bundles	156
4.7.4	Managed Beans und Lokalisierung	159
4.8	Konfiguration	160
4.8.1	Die Servlet-Konfiguration	161
4.8.2	Die JSF-Konfiguration	170
4.8.3	XML-Konfigurationsdatei versus Annotationen	179
4.9	Client-Ids und Komponenten-Ids	180
4.9.1	Id-Arten und Namensräume	180
4.9.2	Client- und server-seitige Programmierung mit Ids	184
4.10	Verwendung allgemeiner Ressourcen	187
4.10.1	Einfache Ressourcen	188
4.10.2	Versionierte Ressourcen und Ressourcen-Bibliotheken	189
4.10.3	Positionierung von Ressourcen	191
4.11	JSTL-Bibliotheken	194

5 Die UI-Komponenten	199
5.1 Die Standardkomponenten	200
5.2 Render-Sätze	204
5.3 Die JSF-Standard-Bibliotheken	204
5.4 Die HTML-Bibliothek	206
5.5 Die Kernbibliothek	208
5.6 Die Facelets-Bibliothek	210
5.7 Die Composite-Component-Bibliothek	211
5.8 Die JSTL-Kern- und Funktionsbibliothek	212
 6 Facelets	215
6.1 Templating mit Facelets	215
6.2 Ein Template-Beispiel: UPN-Rechner	218
6.3 Dynamische Templates	223
6.4 Weitere Facelets-Tags	225
6.5 Entwicklung eigener Komponenten	228
6.6 JSF-Tags als Attribute der Standard-HTML-Tags	234
 7 Ajax	237
7.1 Motivation	238
7.2 Die Grundlagen von Ajax in JSF 2.0	239
7.2.1 JSFs JavaScript-Bibliothek	240
7.2.2 Das <f:ajax>-Tag	242
7.2.3 Das überarbeitete Bearbeitungsmodell einer JSF-Anfrage	245
7.3 Weiterführende Themen	247
7.3.1 Navigation	247
7.3.2 JavaScript mit Java	248
7.3.3 Nicht gerenderte Komponenten	250
7.3.4 Abgekürzte Komponenten-Ids	252
7.4 Ajax mit RichFaces	253
7.4.1 Die <a4j:support>-Komponente	254
7.4.2 Die <a4j:outputPanel>-Komponente	256
7.4.3 Die <a4j:region>-Komponente	257

7.4.4	Die <a4j:commandButton>- und <a4j:commandLink>-Komponenten	259
7.4.5	Die <a4j:poll>-Komponente	259
7.4.6	Die <a4j:log>-Komponente	260
7.5	RichFaces-Komponenten mit eingebauter Ajax-Unterstützung	261
7.5.1	Drag and Drop	262
7.5.2	Bäume	267
7.5.3	Darstellung großer Datenmengen	278
8	JavaServer Faces im Einsatz: Das Online-Banking	285
8.1	Der Seitenaufbau	286
8.2	Das Geschäftsmodell	289
8.3	Authentifizierung und Autorisierung	292
8.3.1	Realisierung der Authentifizierung	293
8.3.2	Die Abmeldung	295
8.4	Pflege der Stammdaten	296
8.5	Überweisungen	299
8.6	Anzeige aller Konten	303
8.7	Anzeige der Umsätze	306
8.8	Export der Umsätze im PDF- und Excel-Format	310
9	JavaServer Faces und Java-EE	317
9.1	Java-EE 5	318
9.2	Java-EE 6	323
9.3	CDI und Weld	324
9.4	Konversationen mit CDI	327
9.5	Weitere Neuerungen in Java-EE 6	333
9.5.1	Servlet 3.0	333
9.5.2	Web-Profile	334
9.5.3	Managed Beans	335
9.6	Authentifizierung und Autorisierung mit JBoss Seam	336
10	Systeme und Werkzeuge	345
10.1	GlassFish	345

10.1.1	Installation und Betrieb	346
10.1.2	Die Datenbank JavaDB	348
10.1.3	Konfiguration	350
10.2	Eclipse	352
10.2.1	Installation	353
10.2.2	GlassFish-Plugin	354
10.2.3	JBoss Tools	356
10.2.4	Projekte	359
10.3	Firebug	359
10.4	Selenium	362
10.4.1	Selenium-IDE	362
10.4.2	Selenium-RC	365
10.5	JSFUnit	368
A	Annotationen	375
A.1	JSF-Annotationen	375
A.2	Annotationen für Managed Beans	378
B	Die Tags der Standardbibliotheken	383
B.1	HTML-Attribute	384
B.2	JavaScript-basierte HTML-Attribute	386
B.3	HTML- und JSF-Attribute für CSS	388
B.3.1	Regeln	388
B.3.2	Klassen- und Id-Selektoren	389
B.4	HTML-Tag-Bibliothek	390
B.5	Kernbibliothek	432
B.6	Facelets	465
C	URL-Verzeichnis	477
	Literaturverzeichnis	483
	Sachverzeichnis	485

Vorwort zur 2. Auflage

In der ersten Auflage haben wir JavaServer Faces in der Version 1.1 beschrieben. Die Änderungen der Version 1.2 waren nicht so umfangreich, dass eine neue Auflage notwendig gewesen wäre. Mit der Version 2.0 sieht dies anders aus. Obwohl die Kompatibilität zur Version 1.2 praktisch vollständig gewährleistet ist, sind zentrale und gewichtige Erweiterungen in JSF eingeflossen. Zu den wichtigsten und von der Anwendergemeinschaft am intensivsten geforderten zählen sicherlich die Definition von Facelets zur StandardbeschreibungsSprache, die auf definierten Schnittstellen basierende Integration von Ajax und die Verwendung von Annotationen. Viele weitere Themen ließen sich noch aufführen. Derartig große Änderungen im JSF-Framework implizieren auch große Änderungen in unserem Buch. So wurden einige Kapitel neu geschrieben, etwa zu den Themen Facelets und Ajax, andere wurden stark überarbeitet, etwa das Kapitel *JSF im Detail* und das Kapitel zum Thema Werkzeugunterstützung. Wir beschreiben in diesem Buch JavaServer Faces in der Version 2.0, vermeiden aber in der Regel die explizite Erwähnung, wann dieses oder jenes Feature in JSF aufgenommen wurde. Falls Sie ein Buch über den Stand vor 2.0 suchen, finden Sie unsere erste Auflage in vielen Bibliotheken. Der Erwerb der ersten Auflage als E-Book ist ebenfalls weiter möglich.

Sommer 2010, *Bernd Müller*

bernd.mueller@ostfalia.de

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Vorwort

JavaServer Faces sind ein Framework für die Entwicklung von Benutzerschnittstellen *für* oder besser *als Teil* einer Java-Web-Anwendung. Die vorliegende Darstellung führt JavaServer Faces nach dieser Definition ein. Es beschreibt, was JavaServer Faces sind und wie man JavaServer Faces für die Entwicklung moderner Benutzerschnittstellen einsetzt, aber auch, wie JavaServer Faces in eine Java-Web-Anwendung zu integrieren sind. Wir behandeln so elementare Themen wie die Anbindung an ein Persistenz-Framework oder die Anbindung an das Authentifizierungs- und Autorisierungssystem des Servlet-Containers. Ein unverzichtbarer Bestandteil kommerzieller Web-Anwendungen ist die Erzeugung von PDF, z.B. für auszudruckende Rechnungen oder Verträge; auch hierfür werden Lösungen entwickelt. Dem aktuellen Trend *Ajax* ist ebenfalls Platz gewidmet.

Dem Anspruch eines „Arbeitsbuches für die Praxis“ wird das Buch gerecht, indem JavaServer Faces anhand eines umfassenden und praxisnahen Beispiels eingeführt werden, ohne zuvor ganze Kapitel über JSF-Grundlagen zu verlieren. Um JavaServer Faces praxisnah einzuführen, werden in den ersten Buchkapiteln keine Grundlagen über Servlets, JSP und JSTL benötigt. Genauso wenig werden Kenntnisse über innere Funktionen einer JVM benötigt, um Java zu programmieren, oder Kenntnisse über B-Baum-Implementierungen, um mit JDBC zu arbeiten. Wo man Servlets und JSPs benötigt, erfolgt jeweils eine kurze Einführung in die benötigten Details. Damit nicht genug, behandelt das Buch umfassend und praktisch alle Details zur Entwicklung von Oberflächen mit JavaServer Faces.

Dieses Buch richtet sich an Leser, die wissen wollen, was JavaServer Faces sind. Es richtet sich aber vor allem an solche, die mit JavaServer Faces entwickeln wollen. „Programmieren lernt man durch Programmieren“. Diesem alten Informatikerspruch werden wir gerecht, weil unser Buch viele praxisnahe Code-Stücke enthält und eine komplette Anwendung implementiert. Alle im Buch dargestellten Code-Stücke stammen aus verschiedenen Beispielprojekten,

die von der Web-Site des Buches heruntergeladen und somit auch praktisch nachvollzogen werden können. Es ist für die Ausbildung von Studenten verschiedener Informatikstudiengänge und für den sich weiterbildenden und im Berufsleben stehenden Praktiker geeignet.

Sommer 2006, *Bernd Müller*
bernd.mueller@fh-wolfenbuettel.de

Kapitel 1

Einleitung

1.1 Ziel dieses Buches

Java wurde 1995 von Sun mit zwei Hauptzielen vorgestellt: zum einen ging es um die portable Programmierung hardware-naher Steuerungen von Geräten, z. B. von Kaffee- und Waschmaschinen, zum anderen sollte das stark auflebende Web bunter und interaktiver werden. Dem ersten Ziel verdanken wir die Plattformunabhängigkeit von Java durch die Definition der JVM (Java Virtual Machine) und des Java-Byte-Codes, dem zweiten Ziel verdanken wir die Applets. Heute, nach über fünfzehn Jahren, ist festzustellen, dass beide Ziele nicht erreicht wurden. Java wird zwar zur Programmierung von Hardware verwendet, doch keineswegs in dem Ausmaß, wie ursprünglich prognostiziert. Die sehr starke Verbreitung von Java in Handys und die Verwendung als Standardsprache von Android ist davon zu unterscheiden. Hier ist Java lediglich Grundlage für eine Vielzahl von Spielen beziehungsweise Anwendungen, wird jedoch nicht zur Steuerung des Handys eingesetzt. Auch die Verwendung von Applets in Web-Anwendungen ist äußerst begrenzt. Hier sind vor allem die Notwendigkeit der Installation eines Java-Plug-In im Web-Browser, und zwar in der vom Applet benötigten Version, und die benötigte Zeit zum Herunterladen des Applets die hemmenden Eigenschaften.

Java ist jedoch sicher nicht als Fehlschlag zu werten. Die Sprache Java – oder genauer die *Plattform Java* – trat einen unvergleichlichen, in der Geschichte der Programmiersprachen noch nie da gewesenen Siegeszug an. Dieser Siegeszug fand zwar nicht im Hardware-Bereich oder als Applet-Sprache statt, sehr wohl aber als server-seitige Sprache zur Entwicklung unternehmenskritischer Anwendungen. Das aktuell sehr beliebte Modell so genannter *Thin-Client*-Anwendungen, d. h. Anwendungen, bei denen der Benutzer durch einen

HTML-Browser die Anwendung bedient, wird mittlerweile von mehreren Java-Spezifikationen unterstützt, von denen JavaServer Faces die aktuellste, ja, nach Interpretation eventuell sogar die einzige ist. Die Beliebtheit von Java im Server-Bereich ist vor allem durch die Plattformunabhängigkeit zu erklären. Unternehmen verfügen im Server-Bereich häufig über eine sehr heterogene Hardware-Landschaft, und die Migration einer Anwendung von einer Hardware und einem Betriebssystem auf eine andere Hardware und ein anderes Betriebssystem sind in der Regel mit sehr viel Aufwand verbunden. Mit Java entfällt dieser Aufwand trotz des Slogans „Write once, run anywhere“ zwar nicht völlig, reduziert sich aber erheblich.

JavaServer Faces werden in der jüngsten Spezifikation aus dem Bereich der Entwicklung von Benutzerschnittstellen für Java-Web-Anwendungen definiert. Sie sind als moderner, komponentenbasierter Ansatz zu sehen, dessen Ziel wir der Spezifikation entlehnen:

“... to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client.”

Ziel dieses Buches ist es, dem Leser die Kenntnisse zu vermitteln, um mit JavaServer Faces moderne Benutzerschnittstellen zu entwickeln. Dem im Untertitel genannten Anspruch von Praxisnähe und dem Charakter eines Arbeitsbuches wird das Buch gerecht, indem wir eine größere Anwendung entwickeln, anhand deren wir die verschiedenen Aspekte von JavaServer Faces und deren Integration in eine Gesamtarchitektur einführen und erläutern. Das Buch ist nicht als überarbeitete Version der Spezifikation zu sehen, sondern es versucht die praktische Anwendung der Spezifikation auf reale Probleme, die in jeglicher Anwendungsentwicklung immer wieder auftauchen. Es enthält wichtige Ratschläge zur Lösung dieser Probleme und blickt über den Tellerrand hinaus. JavaServer Faces genügen nicht, um unternehmenskritische Anwendungen zu realisieren. JavaServer Faces sind immer in einem größeren Kontext eingebettet und müssen z. B. an Business-Modelle oder Datenbanken angeschlossen werden. Dies wird unter anderem innerhalb verschiedener Java-EE-Spezifikationen thematisiert, worauf wir ebenfalls eingehen.

JavaServer Faces sind zwar die aktuellste Spezifikation im Bereich der Java-Web-Anwendungen, werden aber sicher nicht die letzte sein. Bereits kurze Zeit nach der ersten Veröffentlichung der Spezifikation wurden JavaServer Faces erweitert bzw. in andere Systeme integriert. Viele dieser populären Erweiterungen sind in JavaServer Faces 2.0 eingeflossen. Die Version 2.0 ist somit kein „Papier-Tiger“, der am grünen Tisch entstanden ist, sondern sie standardisiert

Entwicklungen, die in anderen Projekten erfolgreich praktiziert wurden. Wir gehen darauf im nächsten Kapitel ausführlicher ein.

1.2 Der Leser

Das intendierte Einsatzgebiet von JavaServer Faces ist die Entwicklung von Oberflächen in unternehmenskritischen Anwendungen. Jeder, der in diesem Bereich zur Zeit tätig ist oder in Zukunft sein wird, kann mit diesem Buch in die Tiefen der Software-Entwicklung mit JavaServer Faces einsteigen. Dies sind zum einen Studenten der Informatik und angrenzender Studiengänge, vor allem aber auch Praktiker, die in diesem Bereich arbeiten.

Unabdingbare Voraussetzung für den effektiven Einsatz des Buches ist die Kenntnis der Sprache Java. Wir gehen davon aus, dass der Leser objektorientierte Konzepte verinnerlicht und eine gewisse Erfahrung bei deren Einsatz hat. Die Entwicklung mit JavaServer Faces erfolgt zu einem großen Teil durch Entwicklungsarbeit mit Java. Der Leser sollte daher Java wirklich beherrschen. Vorteilhaft ist es, wenn Grundkenntnisse über HTML und HTTP vorhanden und die Probleme der Verbindung objektorientierter Anwendungen mit relationalen Datenbanken bekannt sind.

Englische Fachbegriffe

An dieser Stelle seien noch einige Anmerkungen zur Sprache erlaubt. Die Sprache der Informatik ist Englisch. Einige Fachbegriffe können ins Deutsche übersetzt werden, andere nicht. Wenn eine gute deutsche Begrifflichkeit existiert, werden wir diese auch verwenden. Bei einigen Begriffen sind wir der Meinung, dass sie als Eigennamen zu interpretieren und somit nicht zu übersetzen sind. Ein Beispiel sind *Managed Beans*, die nicht als *verwaltete Bohnen* übersetzt werden sollten, oder *UI components* (user interface components), die wir nicht *BS-Komponenten* (Benutzerschnittstellenkomponenten), sondern *UI-Komponenten* nennen wollen. Womit wir beim nächsten Punkt wären: Im Deutschen werden Substantiv-Verbindungen zusammen oder mit Bindestrich geschrieben. Eine getrennt geschriebene Aneinanderreihung, wie sie im Englischen üblich ist, gibt es im Deutschen nicht. Die „user interface components“ müssen also zu „User-Interface-Komponenten“ werden, um die deutsche Grammatik nicht zu sehr zu strapazieren. Eine Ausnahme sind Eigennamen, so dass wir „JavaServer Faces“ in der Originalschreibung übernehmen. Falls Sie selbst schreibend tätig sind, ist das Buch von Peter Rechenberg [Rec06] eine her-

vorragende und sehr zu empfehlende Hilfe, wenn es um Zweifelsfragen der Grammatik und des Ausdrucks geht.

Die englische Sprache hat der deutschen eine weitere Vereinfachung voraus: die der geschlechtsunspezifischen Ansprache. Im Deutschen ist es mittlerweile üblich, beide Geschlechter zu benennen. „Entwicklerinnen und Entwickler“, „EntwicklerInnen“ und „Entwickler/innen“ sind aber allesamt keine stilistischen Meisterleistungen, die uns befriedigen. Es muss eine andere Lösung geben. Die Informatik besitzt eine zentrale Methode, die sowohl in der theoretischen als auch der praktischen Informatik vielerorts eingesetzt wird: die der Abstraktion. Wir versuchen, Aussagen so allgemein wie möglich zu formulieren, um den Anwendungsbereich der Aussage zu vergrößern. Beim Lesen dieses Buches ist es wenig sinnvoll, zwischen männlichen und weiblichen Lesern zu unterscheiden, etwa, indem wir Sie mit „Leser und Leserin“ ansprechen. Im Folgenden verstehen wir unter einem Leser eine Person, die liest, und unter einem Entwickler eine Person, die entwickelt, unabhängig vom Geschlecht. Die Problematik der zweigeschlechtlichen Anrede ist übrigens keineswegs auf die Informatik oder Technik beschränkt. Auch im journalistischen Bereich wird sie zum Teil kritisiert, etwa von Bastian Sick [Sic04].

1.3 Das Buch im Netz

Nichts veraltet in der modernen IT schneller als Software-Systeme. Praktisch alle in diesem Buch verwendeten Systeme haben zumindest in den Minor-Versionsnummern während des Schreibens des Buches Aktualisierungen erhalten. Es ist nicht sinnvoll, einem Buch über JavaServer Faces oder andere aktuelle IT-Themen eine CD beizulegen oder gar API-Dokumentationen abzudrucken. Als Zugang zu Informationen dieser Art betreiben wir im WWW die Web-Site

www.jsfpraxis.de

Sie finden dort den lauffähigen Quell-Code aller Beispiele sowie teilweise Lösungen zu den Übungsaufgaben und, wenn nötig, ein Druckfehlerverzeichnis. Neben diesen buchspezifischen Inhalten finden Sie eine Fülle von Dokumentationen zu JavaServer Faces, aber auch zu den im Buch verwendeten Systemen. Wir stellen diese im nächsten Abschnitt übersichtsartig und im Kapitel 10 etwas detaillierter dar, empfehlen dem Leser aber zusätzlich die aktuelleren Informationen auf der Web-Site.

Das Buch ist ein Arbeitsbuch für die Praxis. Der erzielbare Nutzen beim „einfachen“ Lesen des Buches ist auf konzeptionelle Erkenntnisse und „Aha-Effekte“

beschränkt. Wenn Sie den Nutzen des Buches maximieren wollen, müssen Sie die Beispiele herunterladen, installieren und ausprobieren. Es empfiehlt sich die Durchsicht der Quellen und der Versuch, Dinge anders zu realisieren, als wir es getan haben, bzw. der Versuch, die vorgestellten Realisierungen zu optimieren. Wenn Sie eine bessere Lösung oder gar die optimale Lösung gefunden haben, sind wir sehr daran interessiert. Schreiben Sie uns eine E-Mail, und berichten Sie bitte über Ihre Lösung.

1.4 Verwendete Software

Um die in diesem Buch entwickelte Software auszuführen, benötigt man einen *Servlet-Container* und eine Implementierung von JavaServer Faces. Servlets werden in der neuesten Version 3.0 durch den Java Specification Request 315 (JSR-315) [URL-JSR315] spezifiziert. Die Referenzimplementierung dieser Spezifikation ist im Application-Server *GlassFish* enthalten. Dieser Application-Server enthält auch alle weiteren Implementierungen der Standards, die in Java-EE 6 enthalten sind, z. B. das Java Persistence API (JPA) und Contexts and Dependency Injection (CDI), vor allem aber *Mojarrra* als Referenzimplementierung von JavaServer Faces 2.0. GlassFish empfiehlt sich daher als Grundlage zur Ausführung der Beispiele des Buches. Wir beschreiben die Installation und Verwendung von GlassFish in Kapitel 10.

Neben einem Servlet-Container bzw. einem Application-Server empfehlen wir Ihnen die Installation einer Java-IDE. Die populärste Open-Source-Alternative in diesem Bereich ist im Augenblick *Eclipse*; Sie können allerdings auch jede andere IDE verwenden. Die Beispiele des Buches können von den Buchseiten im Netz (siehe Abschnitt 1.3) heruntergeladen werden. Sie sind als Eclipse-Projekte organisiert, so dass der Einsatz von Eclipse zur Ausführung der Beispiele mit dem geringsten Aufwand verbunden ist.

Die in diesem Buch entwickelte Software verwendet eine ganze Reihe von Systemen. Die schon erwähnten Implementierungen von JPA und CDI sind in GlassFish enthalten. Andere Systeme, etwa zur Erzeugung von PDF oder Excel, werden als zusätzliche Bibliothek benötigt. Wir gehen an den entsprechenden Stellen darauf ein. Alle verwendeten Systeme sind Open-Source-Systeme, die Sie lizenzkostenfrei verwenden können.

1.5 Aufbau des Buches

In Kapitel 2 erfolgt zunächst eine Bestandsaufnahme zum Thema JavaServer Faces: Was sind JavaServer Faces, woher kommen sie und, soweit plausibel begründbar, wohin gehen sie? Die mit JavaServer Faces 2.0 eingeführten Neuerungen werden ebenfalls erwähnt. Das Kapitel wird mit einem einführenden Beispiel, dem Spiel Tic-Tac-Toe, abgeschlossen. In Kapitel 3 wird eine einfache Anwendung zur Verwaltung von Comedian-Daten entwickelt. Anhand dieser Anwendung wird in die grundlegenden Strukturen der Entwicklung mit JavaServer Faces eingeführt. Die dabei gewonnenen, zunächst noch oberflächlichen Einblicke in die innere Funktionsweise von JavaServer Faces motivieren Kapitel 4, in dem eine detaillierte Darstellung aller relevanten Aspekte der Entwicklung mit JavaServer Faces erfolgt. In diesem Kapitel werden das grundlegende Bearbeitungsmodell einer Anfrage, die Validierung und Konvertierung von Benutzereingaben, die Navigation innerhalb einer Anwendung, die Internationalisierung sowie weitere zentrale Themen ausführlich dargestellt. Kapitel 5 stellt den Aufbau der UI-Komponenten dar und gibt einen Überblick über die zur Verfügung stehenden Komponenten.

In den Kapiteln 6 und 7 werden zwei gewichtige Neuerungen der Version 2.0 vorgestellt: Facelets und Ajax. Im Kapitel 8 wird am Beispiel *Online-Banking* eine größere Anwendung implementiert, um Anforderungen realisieren zu können, die in typischen Hello-World-Programmen nicht vorkommen. So werden z. B. PDF- und Excel-Dokumente erzeugt.

JavaServer Faces haben als intendiertes Einsatzgebiet die Entwicklung unternehmenskritischer Anwendungen und sind Bestandteil der Enterprise-Edition von Java. Das Kapitel 9 zeigt, wie JavaServer Faces in Java-EE integriert sind und welche Verwendungsalternativen existieren. Mit JBoss-Seam stellen wir ein System vor, das auf JavaServer Faces basiert, aber in vielen Aspekten über den Standard hinausgeht.

Zur Entwicklung von Anwendungen mit JavaServer Faces werden ein paar wenige grundlegende Systeme benötigt. Andere Systeme sind zur Unterstützung im Entwicklungsprozess in ihrer Verwendung dringend zu empfehlen. Im Kapitel 10 geben wir einen Überblick über derartige Systeme, der allerdings nicht vollständig sein kann.

Der Anhang A fasst die durch JSF 2.0 eingeführten Annotationen zusammen. In Anhang B werden die Tags der Standardbibliotheken detailliert beschrieben. Neben der eigentlichen Bibliotheksdarstellung erläutern wir auch Aspekte der Integration von HTML, JavaScript und CSS. Der Anhang C zählt eine

Reihe wichtiger und interessanter Internet-Quellen auf, die wir dem Leser als Ausgangspunkt eigener Recherchen dringend empfehlen.

Wenn Sie das Buch erworben haben, können Sie die E-Book-Version des Buches unentgeltlich herunterladen. Wir raten Ihnen sehr, dies zu tun. Das E-Book ist als vollständig verlinktes, mehrfarbiges Hypertextsystem erstellt. Sie können mit dem Acrobat-Reader innerhalb des Buches navigieren und Links in das Internet automatisch öffnen. Bitte probieren Sie dies aus.

1.6 JSF-Versionen vor 2.0

Eine Unterscheidung zwischen JSF-Eigenschaften der Version 2.0 und den Versionen vor 2.0 ist unserer Meinung nach textuell zu aufwändig, beeinflusst den Lesefluss negativ und ist vor allem für die meisten Leser nicht relevant. Wir beschreiben in diesem Buch die Version 2.0 von JSF und merken in der Regel an, wenn in Version 2.0 Neues hinzugekommen ist. Wir verzichten aber auf die Darstellung der Eigenschaften in den Versionen vor 2.0 und eine vollständige „Versionierung“ aller Aussagen und Darstellungen.

Wenn Sie Informationen zu JSF vor der Version 2.0 suchen, können Sie die erste Auflage dieses Buches zwar nicht mehr in Papierform, sehr wohl aber als E-Book erwerben. Außerdem ist die erste Auflage in vielen Bibliotheken im Bestand. Auf den Web-Seiten zum Buch finden Sie weiterhin den Quell-Code aller Beispiele der ersten Auflage.

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Kapitel 2

Motivation

JavaServer Faces sind als Teil der Spezifikationen Java-EE 5 und Java-EE 6 unabdingbarer Bestandteil eines jeden entsprechenden Application-Servers. Die beiden weiteren GUI-relevanten Spezifikationen innerhalb von Java-EE sind Servlets und JavaServer Pages. Das Einsatzgebiet von Servlets ist jedoch die Realisierung von Frameworks, wie JSF es eines ist, und nicht die Erstellung kompletter Geschäftsanwendungen. JavaServer Pages werden nicht mehr weiterentwickelt. JavaServer Faces sind also die einzige praxisrelevante Spezifikation für die Entwicklung von Geschäftsanwendungen innerhalb von Java-EE und müssen daher eigentlich nicht motiviert werden. Wir wollen dies hier trotzdem tun, um dem Leser bereits an dieser Stelle einen Überblick über einige der herausragenden Eigenschaften von JSF zu geben.

2.1 Was sind JavaServer Faces?

JavaServer Faces übernehmen nicht nur die Routineaufgaben, die in einer Servlet- oder JSP-Anwendung zu programmieren sind. JavaServer Faces definieren ein Komponentenmodell für Benutzerschnittstellenelemente einer Web-Anwendung, so wie Swing ein Komponentenmodell für lokale Oberflächen definiert. So existieren etwa Komponenten für die Texteingabe, zur Auswahl von Menü-Einträgen oder für die Anzeige von Fehlermeldungen. Diese Komponenten können ganz analog zu Swing-Komponenten hierarchisch angeordnet werden, indem man sie in Containern verschachtelt. Neue Komponenten, z.B. ein Kalender, können entwickelt und dann wie vordefinierte Komponenten verwendet werden. JSF-Komponenten leben auf dem Server, sind in Java programmiert und in einer Anwendung mit Java als Implementierungssprache

verwendbar. Die als Antwort einer HTTP-Anfrage an einen Client geschickten HTML-Seite (andere Antwortarten sind möglich) stellen ein Abbild des Komponentenbaums des Servers dar.

Bei der Entwicklung grafischer Benutzeroberflächen (GUI, Graphical User Interface) hat sich das Model-View-Controller-Entwurfsmuster (MVC Pattern) durchgesetzt. Es wird durch JavaServer Faces komplett realisiert. MVC trennt streng zwischen *Modell*, *View* und *Controller*. Das Modell ist das Geschäftsmodell der Anwendung. In ihm sind die fachlichen Klassen, etwa *Kunde* und *Rechnung*, sowie die fachliche Logik enthalten. Die View ist ausschließlich für die Darstellung verantwortlich, sie darf keine Anwendungslogik enthalten. Der Controller ist schließlich die Instanz, die alle Steuerungsvorgänge der Oberfläche mit dem Benutzer regelt und Änderungen des Benutzers in der View mit dem Modell konsistent hält. Sie enthält damit zum einen die Oberflächenlogik, etwa „Knopf A kann nur gedrückt werden, wenn im Eingabefeld B ein Wert größer 100 steht“, aber auch die entsprechenden Methoden, um die Eingabe einer neuen Kundenadresse bis zum Modell durchzureichen.

Bei JavaServer Faces ist das Modell beliebig wählbar. POJOs (Plain Old Java Objects), EJBs (Enterprise JavaBeans) oder beliebige andere Java-Objekte sind möglich. Die View ist ebenfalls nicht festgelegt. Zwar gibt die Spezifikation als mögliche Realisierung HTML vor, d. h. jede JSF-Implementierung muss mindestens eine HTML-View bereitstellen, andere, etwa WML oder Flash, sind aber ebenfalls denkbar und möglich. Der Controller wird in JSF zum einen durch die schon erwähnten Komponenten realisiert, zum anderen durch so genannte *Handler*, die in Java zu realisieren sind.

Es lässt sich trefflich darüber streiten, ob die Komponenten nicht zur View gehören. Sind nicht sie es, die Eingabefelder und Knöpfe darstellen? Dies ist einerseits sicher richtig, andererseits sieht der Benutzer aber nicht die Komponente an sich, sondern die aus der Komponente generierte Darstellung, die sich z. B. im Web und auf dem Handy sehr unterscheiden kann. Hinzu kommt, dass in der Komponente angegeben wird, ob sie überhaupt darzustellen ist, und, wenn ja, ob sie für Benutzeraktionen empfänglich ist.

2.2 JavaServer Faces und Zukunftssicherheit

Der *Java-Community-Process (JCP [URL-JCP])* wurde 1998 ins Leben gerufen, um Spezifikationen der Java-Plattform zu entwickeln und fortzuschreiben. Sun versuchte damit die Furcht vieler Firmen vor einer monopolistischen und undemokratischen Definiton der Java-Plattform zu entkräften. Mitglieder können Personen und Firmen werden, die dann bei der Entwicklung

und Fortschreibung von *Java-Specification-Requests (JSR)* mitarbeiten können. Ein Java-Specification-Request ist eine Spezifikation für einen Teil der Java-Plattform. Im Augenblick sind mehrere Hundert Personen und Firmen Mitglieder des JCPs. Zu den Firmen gehören praktisch alle großen Software-Häuser.

Im Jahr 2001 wurde der JSR-127 initiiert, der die JSF-Spezifikation zum Inhalt hat. Neben einigen Privatpersonen waren Firmen und Organisationen wie die Apache Software Foundation, BEA Systems, Borland, Fujitsu, Hewlett-Packard, IBM, ILOG, IONA Technologies, Macromedia, Novell, Oracle, Siemens und Sun beteiligt, um einige zu nennen. Diese breite Unterstützung macht klar, welchen Stellenwert JavaServer Faces haben.

Im März 2004 wurde das *final Release* der JSF-Spezifikation 1.0 veröffentlicht. Die Java-2-Enterprise-Edition, die unter anderem Servlets, JavaServer Pages und Enterprise-JavaBeans umfasst, wurde in der Version 1.4 im November 2003, also *vor* der JSF-Spezifikation, veröffentlicht. Sie konnte JavaServer Faces daher nicht enthalten. Die Version 1.5 (nach neuer Sprach- und Zählweise Java-EE 5) wurde im Mai 2006 freigegeben. Die Java-EE-Spezifikation umfasst JavaServer Faces in der Version 1.2. Dies bedeutet, dass alle Hersteller von Java-EE-Application-Servern JavaServer Faces unterstützen müssen, um spezifikationskonform zu sein. Es ist unzweifelhaft, dass dies ein weiterer Schub für die Verbreitung von JavaServer Faces war. Im Dezember 2009 wurde Java-EE 6 veröffentlicht, in der JSF 2.0 enthalten ist.

2.3 Neuerungen in JavaServer Faces 2.0

Der Sprung in der Versionsnummer von JSF 2.0 ist durch eine große Zahl von Neuerungen, Erweiterungen und Änderungen begründet. Zu den grundlegenden Themenbereichen dieser Art gehört sicher die Festlegung auf *Facelets* als Standardseitenbeschreibungssprache und die Integration von *Ajax*, d. h. die Möglichkeit, über ein definiertes JavaScript-API Teilebereiche einer Seite zu aktualisieren und eine erhöhte Benutzerinteraktivität zu ermöglichen. Viele andere Neuerungen, Erweiterungen und Änderungen, die wir aufzählen ohne eine Gewichtung vornehmen zu wollen, flossen ebenfalls ein: Die Möglichkeit der Verwendung von *Annotationen* statt einer XML-Konfiguration; die Mitteilung über bestimmte JSF-Laufzeit-Ereignisse durch *System-Events*; eine einfache Möglichkeit zur Definition zusammengesetzter Komponenten (*Composite Components*); erweiterte Möglichkeiten zur *Validierung*, unter anderem durch die eigenständige Spezifikation Bean-Validation; eine *Steigerung der Effizienz*

der JSF-Implementierungen, da nur noch geänderte Bereiche des Komponentenbaums zwischen Anfragen gespeichert werden, und einige mehr.

2.4 Spezifikationen und Implementierungen

Wie bereits erwähnt, wurde die JSF-Spezifikation Version 1.0 im März 2004 veröffentlicht. Bereits im Mai 2004 folgte Version 1.1. Diese wurden im Rahmen des JSR 127 [URL-JSR127] veröffentlicht. Die Spezifikationen hat man jedoch mittlerweile aus dem dortigen Download-Bereich entfernt. Die Version 1.2 wurde ebenfalls im Rahmen des JCP bearbeitet, und zwar als JSR 252 [URL-JSR252], die Version 2.0 als JSR 314 [URL-JSR314].

Im Augenblick existieren zwei Implementierungen der JSF-Spezifikation 2.0. Dies ist zum einen die so genannte Referenzimplementierung von Sun, häufig als JSF-RI (Referenzimplementierung) bezeichnet, und zum anderen eine Implementierung der Apache Software Foundation, *MyFaces* genannt. Beide Implementierungen sind Open-Source-Entwicklungen, d. h. neben den binären Versionen sind auch die Quell-Codes erhältlich.

Für einen JSR wird jeweils ein *Technology-Compatibility-Kit* (TCK, für weitere Informationen siehe [URL-TCK]) entwickelt. Ein TCK besteht aus einer Menge von Tests, deren Bestehen Voraussetzung dafür ist, dass ein System als Implementierung der jeweiligen Spezifikation gelten darf.

Während wir in der ersten Auflage dieses Buches MyFaces als JSF-Implementierung und Tomcat als Servlet-Container verwendet haben, verwenden wir nun für die entwickelten Beispiele die JSF-Referenzimplementierung mit Code-Namen Mojarra [URL-MOJ]. Als Servlet-Container verwenden wir Glassfish [URL-GF], der allerdings nicht nur ein Servlet-Container, sondern in der Version 3.0 ein vollständiger Java-EE 6 Application-Server ist. Nähere Einzelheiten dazu finden Sie in Kapitel 10.

2.5 Die erste JSF-Anwendung: Tic-Tac-Toe

Während wir im nächsten Kapitel ein „richtiges“ Informationssystem – und damit ein System aus dem intendierten Einsatzgebiet von JSF – erstellen, soll hier als einführendes Beispiel eine eher spielerische JSF-Anwendung entwickelt werden. Diese geht jedoch bereits deutlich über ein *Hello-World* hinaus und belegt die zu Beginn dieses Kapitels postulierten Aussagen: JavaServer Faces vereinfachen die Entwicklung von Web-Oberflächen erheblich, da eine komponentenbasierte, auf dem MVC-Modell aufbauende Architektur den Entwick-

lungsaufwand für den Controller reduziert und den Entwicklungsaufwand zur Verbindung von Modell, View und Controller minimiert.

Das Spiel Tic-Tac-Toe dürfte spätestens nach dem Film *War Games* allgemein bekannt sein. Zwei Spieler versuchen, durch abwechselndes Wählen eines Feldes auf einem 3×3 -Spielfeld drei in einer horizontalen, vertikalen oder diagonalen Linie befindliche Felder zu besetzen. Die Anwendung soll dieses Spiel realisieren, wobei der Benutzer den einen Spieler, der Rechner den anderen Spieler darstellt.

Da aus Sicht des Buches das Business-Modell, hier also die Implementierung des Spielfelds und der Spielregeln, nicht relevant ist, abstrahieren wir von dieser Implementierung und betrachten nur die Schnittstelle. Das komplette Spiel kann, wie alle Beispiele, von den Web-Seiten des Buches heruntergeladen werden. In Listing 2.1 stellen wir die Schnittstelle zum Spiel dar.

Listing 2.1: Das Interface Brett

```
/**  
 * Eine Instanz der Klasse Brett stellt eine Spielsituation auf  
 * dem Brett dar. Der Computer ist durch den Kreis dargestellt,  
 * der Spieler durch das Kreuz.  
 *  
 * Die Methoden sind aus der Sicht des Computers definiert,  
 * z.B. liefert {@code isGewonnen()} {@code true}, falls der  
 * Computer gewonnen hat.  
 *  
 * Layout:  
 * 0 1 2  
 * 3 4 5  
 * 6 7 8  
 *  
 * @author Bernd Mueller  
 * @version 1.0  
 */  
public interface Brett {  
  
    public static final Boolean KREIS = Boolean.TRUE; // Rechner  
    public static final Boolean KREUZ = Boolean.FALSE; // Spieler  
    public static final Boolean LEER = null;  
  
    public boolean isGewonnen();  
    public boolean isVerloren();  
    public boolean isFertig();  
  
    /**  
     * Computer wählt den nächsten freien 'guten' Zug.  
     */
```

```
/*
public void waehleZug();

/**
 * Spieler setzt auf Feld i
 * @param i Brett-Index, zwischen 0 und 8
 * @exception IllegalArgumentException Feld bereits belegt
 */
public void setze(int i) throws IllegalArgumentException;

public Boolean[] getBrett();
}
```

Ein Feld des Spielbretts ist entweder leer oder von einem der beiden Spieler belegt. Diese drei Alternativen können einfach durch boolesche Werte dargestellt werden, was mit den drei Konstanten KREIS, KREUZ und LEER geschieht. Die Methoden `isGewonnen()` und `isVerloren()` testen den Spielzustand aus Sicht des Computers, `isFertig()` wird zur Erkennung des Spielendes verwendet. Die Methode `waehleZug()` lässt den Computer ein Feld belegen, die Methode `setze(int)` belegt ein Feld für den Spieler. Die Methode `getBrett()` gibt das Brett als Array von booleschen Werten entsprechend den drei Konstanten zurück.

Listing 2.2 zeigt die Datei `ttt.xhtml`, die die JSF-Seite für Tic-Tac-Toe enthält.

Listing 2.2: Tic-Tac-Toe als JSF-Seite

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<head>
    <title>Tic Tac Toe</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
</head>
<body>
    <h:form>
        <h:panelGrid columns="3">
            <h:commandButton id="feld-0" image="#{tttHandler.image[0]}"
                            actionListener="#{tttHandler.zug}" />
            <h:commandButton id="feld-1" image="#{tttHandler.image[1]}"
                            actionListener="#{tttHandler.zug}" />
            <h:commandButton id="feld-2" image="#{tttHandler.image[2]}"
                            actionListener="#{tttHandler.zug}" />
            <h:commandButton id="feld-3" image="#{tttHandler.image[3]}"
                            actionListener="#{tttHandler.zug}" />
            <h:commandButton id="feld-4" image="#{tttHandler.image[4]}"
```

```
    actionListener="#{tttHandler.zug}" />
<h:commandButton id="feld-5" image="#{tttHandler.image[5]}" 
    actionListener="#{tttHandler.zug}" />
<h:commandButton id="feld-6" image="#{tttHandler.image[6]}" 
    actionListener="#{tttHandler.zug}" />
<h:commandButton id="feld-7" image="#{tttHandler.image[7]}" 
    actionListener="#{tttHandler.zug}" />
<h:commandButton id="feld-8" image="#{tttHandler.image[8]}" 
    actionListener="#{tttHandler.zug}" />
</h:panelGrid>
<br />
<h:outputText id="meldung" value="#{tttHandler.meldung}" />
<br /><br />
<h:commandButton value="Neues Spiel"
    action="#{tttHandler.neuesSpiel}"/>
</h:form>
</body>
</html>
```

An dieser Stelle soll keine Einführung in JavaServer Faces erfolgen, das ist den weiteren Kapiteln vorbehalten. Wir wollen lediglich einen Überblick über JavaServer Faces geben, um ein Gefühl für die Technik zu vermitteln. Man erkennt in Listing 2.2, dass die Seite im `<html>`-Tag einen Namensraum `h` einbindet. Alle Tags, die diesen Präfix verwenden, sind JSF-Tags, alle anderen Standard-HTML. JSF besteht aus zwei Bibliotheken, der HTML-Bibliothek und der Kernbibliothek, die in der Regel auf die Namensräume `h` und `f` abgebildet werden. Da wir keine Elemente der Kernbibliothek verwenden, war das Einbinden dieser Bibliothek nicht notwendig.

Das erste JSF-Element ist `<h:form>`, aus dem später ein HTML-Formular wird. Das nächste Element, `<h:panelGrid>`, stellt eine Tabelle mit drei Spalten dar. In diesem Element enthaltene Tags werden zeilenweise auf die drei Spalten aufgeteilt. In unserem Beispiel entsteht so aus neun Schaltflächen das 3×3 -Spielfeld. Es besteht aus Schaltflächen (die `<h:commandButton>`-Tags) mit den Attributen `id`, `image` und `actionListener`, die den eigentlich interessanten Teil der Seite ausmachen.

Zuletzt folgt ein `<h:outputText>` für die Ausgabe des Spielresultats sowie ein weiteres `<h:commandButton>`, um ein neues Spiel beginnen zu können.

Die `<h:commandButton>`- und das `<h:outputText>`-Tag enthalten jeweils Ausdrücke der JSF-Expression-Language (JSF-EL). Diese Sprache kennt arithmetische und boolesche, aber auch, wie in diesem Fall, methodenwertige Ausdrücke. `tttHandler` ist ein normales Java-Objekt, mit den Methoden

```
public String getMeldung()
public String[] getImage()
public void zug(ActionEvent)
public String neuesSpiel()
```

Im allgemeinen Fall werden Properties eines Java-Objekts durch solche Ausdrücke an UI-Komponenten gebunden. Unter einem Property versteht JSF ein Attribut nach der JavaBean-Spezifikation mit entsprechendem Getter und Setter, im obigen Fall also ein Attribut `image`, dem Getter `getImage()` und dem Setter `setImage()`. Das in diesem Beispiel verwendete JavaBean besitzt nur den Getter, was aber unerheblich ist. Der Wert des in diesem Fall Array-wertigen Property wird für eine entsprechende Grafik-Datei verwendet. Die Methode `zug` ist eine Event-Listener-Methode. JSF hat das Event-Source/Event-Listener-Modell von AWT und Swing übernommen. Für eine Event-Quelle können daher Listener registriert werden. In diesem Fall ist die Methode `zug` ein Event-Listener für das Drücken der neun Schaltflächen.

Als nächstes Teil im Puzzle fehlt noch das Objekt, das in der JSF-Seite `tttHandler` genannt wurde. Listing 2.3 zeigt die Klasse `TicTacToeHandler`, jedoch ohne die Methoden `zug` und `getImage`, die später separat erläutert werden. Das Objekt `tttHandler` ist eine Instanz der Klasse `TicTacToeHandler`. Dies wird durch die `@ManagedBean`-Annotation in der ersten Zeile erreicht.

Listing 2.3: Die Klasse `TicTacToeHandler` (Ausschnitt)

```
@ManagedBean(name = "tttHandler")
@SessionScoped
public class TicTacToeHandler {

    private static final String KREIS = "/pages/images/kreis.gif";
    private static final String KREUZ = "/pages/images/kreuz.gif";
    private static final String LEER   = "/pages/images/leer.gif";

    private Brett brett;
    private String meldung;

    public TicTacToeHandler() {
        brett = new BrettImpl();
    }

    public String neuesSpiel() {
        brett = new BrettImpl();
        meldung = "";
        return "success";
    }
}
```

```
public String getMeldung() {
    return meldung;
}

public void setMeldung(String meldung) {
    this.meldung = meldung;
}
}
```

Die Klasse **TicTacToeHandler** besitzt die beiden Instanzvariablen **brett** und **meldung**, von denen allerdings nur **meldung** eine Property im Sinne der Java-Bean-Spezifikation mit entsprechendem Getter und Setter ist. Die Variable **brett** hält eine Instanz der Modell-Klasse **Brett**.

Die Methode **zug(ActionEvent)** in Listing 2.4 ist eine typische Controller-Methode. Als Mittler zwischen Modell und View wird sie immer aufgerufen, wenn der Spieler ein einzelnes Spielfeld mit der Maus anklickt. In der Methode wird zunächst die Event-Quelle bestimmt (**ae.getComponent()**), über deren Id man dann das Spielfeld bestimmt. Der Rest der Methode ist selbsterklärend.

Listing 2.4: Die Methode **zug(ActionEvent)**

```
public void zug(ActionEvent ae) {
    if (brett.isFertig())
        return;
    try {
        brett.setze(
            new Integer(ae.getComponent().getId().split("-")[1]));
        if (brett.isVerloren()) {
            meldung =
                "Herzlichen Glueckwunsch, Sie haben gewonnen";
            return;
        }
        brett.waehleZug();
        if (brett.isGewonnen()) {
            meldung = "Sie haben leider verloren";
        }
    } catch (Exception e) {
        log.info("Kein Spielerzug ausgefuehrt");
    }
}
```

Die Methode **getImage()**, dargestellt in Listing 2.5, liefert ein Array von Dateinamen zurück, das zur Anzeige der drei verschiedenen Spielfeldbelegungen

verwendet wird. Dem aufmerksamen Leser wird nicht entgangen sein, dass in der JSF-Seite für jede Schaltfläche der Aufruf der Methode erfolgt, doch immer nur eines der neun Felder Verwendung findet. Hier sind sicherlich effizientere Lösungen möglich, die aber nicht Gegenstand unserer Erörterungen sind.

Listing 2.5: Die Methode `getImage()`

```
public String[] getImage() {
    String[] feld = new String[9];
    for (int i = 0; i < brett.getBrett().length; i++) {
        if (brett.getBrett()[i] == Brett.KREIS) {
            feld[i] = KREIS;
        } else if (brett.getBrett()[i] == Brett.KREUZ) {
            feld[i] = KREUZ;
        } else {
            feld[i] = LEER;
        }
    }
    return feld;
}
```

Zum Schluss bleibt noch die Frage offen, wer zu welchem Zeitpunkt das Objekt `tttHandler` erzeugt. Die JSF-Implementierung erledigt dies automatisch, und zwar vor dem ersten Zugriff auf das Objekt. Erreicht wird dies durch die Definition einer automatisierten, durch die `@ManagedBean`-Annotation realisierten Bean-Verwaltung. In diesem Fall wird die Bean vor dem ersten Zugriff einer HTTP-Session erzeugt und bleibt über die Lebenszeit der Session bestehen. Die Definition dieser Lebensdauer erfolgt über die `@SessionScoped`-Annotation in der zweiten Zeile des Listings 2.3 auf Seite 16. Eine ausführliche Darstellung von Managed Beans erfolgt in Abschnitt 4.3. Einen Eindruck vom Spiel vermittelt Abbildung 2.1.



Projekt

Das Tic-Tac-Toe-Spiel können Sie im Internet von der Web-Site des Buches herunterladen. Es ist im Projekt *tic-tac-toe* enthalten.

Aufgabe 2.1

Laden Sie das Tic-Tac-Toe-Projekt von der Web-Site des Buches herunter. Installieren und testen Sie es. Nähere Informationen zur Installation des Application-Servers und der Buchprojekte finden Sie im Kapitel 10 und auf der Web-Site des Buches.

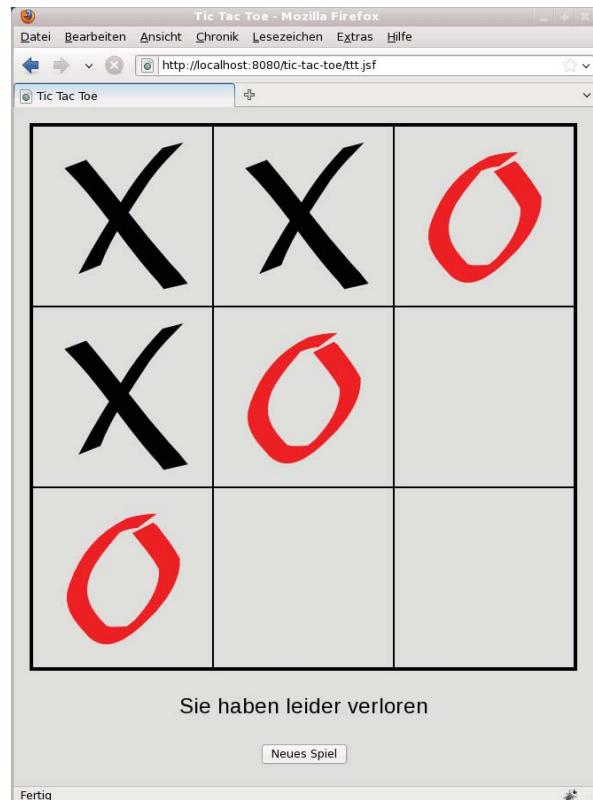


Abbildung 2.1: Browser-Darstellung des Spiels Tic-Tac-Toe

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Kapitel 3

Die Anwendung *Comedians*

Obwohl Tic-Tac-Toe als Einführung in JSF durchaus sinnvoll und motivierend ist, entstammt es nicht dem intendierten Einsatzgebiet von JSF und Java-EE. Java-EE besteht aus mehreren Spezifikationen, auf deren Basis Unternehmensanwendungen realisiert werden können. Unternehmensanwendungen zeichnen sich durch verschiedene Eigenschaften wie etwa Datenkonsistenz im Mehrbenutzerbetrieb, Sicherheit, Skalierbarkeit und Verfügbarkeit aus. Im Mittelpunkt steht die Verarbeitung betrieblicher Daten, denen wir uns nun in der zweiten Beispielanwendung widmen wollen. Dazu entwickeln wir eine kleine Anwendung zur Verwaltung von Stammdaten.

Ein weiterer Grund für das nun zu entwickelnde Beispiel ist das Kapitel 4, *JavaServer Faces im Detail*, das als grundlegendes Kapitel dieses Buches den Aufbau und die Funktionsweise von JavaServer Faces erläutert. Wir glauben, dass dem Leser das Verständnis des Kapitels 4 leichter fallen wird, wenn zuvor ein etwas realistischeres Beispiel als Tic-Tac-Toe besprochen wurde.

Wir beschränken uns in der Beispielanwendung auf die Stammdatenverwaltung von Comedians. Ein Comedian soll in der Anwendung lediglich aus Vor- und Nachnamen sowie seinem Geburtstag bestehen. Die Anwendung soll neue Comedians anlegen und existierende Comedians ändern und löschen können. Dieses einfache Anwendungsgebiet genügt bereits, um einige über die im Tic-Tac-Toe-Spiel hinausgehende Eigenschaften von JSF einführen zu können. Die Einführung erfolgt überblicksartig und wenig detailliert. Die detaillierte Darstellung aller Konzepte holen wir dann in Kapitel 4 nach. Neben weiteren JSF-Eigenschaften werden wir zusätzlich den Standardfall der Verwendung einer relationalen Datenbank für die Datenhaltung in unser Beispiel einbauen.

3.1 JSF-Seiten

JSF-Seiten werden in einer bestimmten Syntax beschrieben. Erstaunlicherweise ist diese Syntax in der Spezifikation nicht eindeutig festgelegt, sondern nur als Minimalforderung formuliert. Bis zur Version 1.2 mussten JSF-Implementierungen JavaServer Pages (JSP) unterstützen. Ab Version 2.0 werden *Facelets* vorausgesetzt, und JSP wird nur noch aus Kompatibilitätsgründen unterstützt, jedoch nicht mehr weiterentwickelt. Facelets, die neue sogenannte Page-Description-Language (PDL), besitzen gegenüber JSP einige Vorteile, auf die wir später eingehen werden. Facelets verwenden XHTML als Sprache und ist somit sicher für viele Leser intuitiv verständlich.

Wir beginnen mit der JSF-Seite zur Darstellung aller Comedians. Eine reale Anwendung wird eine Möglichkeit zur Auswahl anzugebender Comedians bieten, auf die wir aber verzichten und immer alle Comedians anzeigen. Eine Möglichkeit zur Darstellung von a priori unbekannt vielen Zeilen ist die Datentabelle `<h:dataTable>`. Die zentralen Attribute sind `value` und `var`. Das Attribut `value` repräsentiert die Daten, über die iteriert wird, um die Zeilen der Tabelle zu erhalten. Daher muss der erlaubte EL-Ausdruck als Wert des `value`-Attributs zu einer Menge von Objekten evaluieren, z.B. ein Array oder eine Liste (`java.util.List`), aber auch eine Instanz der Klasse `java.faces.model.DataModel`, wie wir sie in unserem Beispiel verwenden. Das Attribut `var` deklariert die Iterationsvariable, die dann innerhalb der Datentabelle verwendet werden kann. Listing 3.1 zeigt die JSF-Seite mit der zentralen Datentabelle.

Listing 3.1: Anzeige aller Comedians (`anzeige-comedians.xhtml`)

```
1 <!DOCTYPE html PUBLIC
2      "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5       xmlns:h="http://java.sun.com/jsf/html"
6       xmlns:f="http://java.sun.com/jsf/core">
7 <head>
8   <title>Anzeige Comedians</title>
9   <link rel="stylesheet" href="style.css" type="text/css" />
10 </head>
11 <body>
12 <h:form>
13   <h:dataTable value="#{comedianHandler.comedians}"
14     var="comedian" rowClasses="odd,even"
15     headerClass="header" footerClass="footer">
16     <f:facet name="header">Anzeige aller Comedians</f:facet>
```

```
17      <f:facet name="footer">#{resourceBundle.copyright}</f:facet>
18      <h:column>
19          <f:facet name="header">Vorname</f:facet>
20          #{comedian.vorname}
21      </h:column>
22      <h:column>
23          <f:facet name="header">Nachname</f:facet>
24          #{comedian.nachname}
25      </h:column>
26      <h:column>
27          <f:facet name="header">Geburtstag</f:facet>
28          <h:outputText value="#{comedian.geburtstag}">
29              <f:convertDateTime pattern="dd.MM.yyyy"/>
30          </h:outputText>
31      </h:column>
32      <h:column>
33          <h:commandButton action="#{comedianHandler.aendern}"
34                          value="Ändern" />
35      </h:column>
36      <h:column>
37          <h:commandButton action="#{comedianHandler.loeschen}"
38                          value="Löschen" />
39      </h:column>
40  </h:dataTable>
41  <h:commandButton action="#{comedianHandler.neuanlage}"
42                          value="Neuanlage" />
43 </h:form>
44 </body>
45 </html>
```

Man erkennt in den Zeilen 5 und 6 die Deklaration der XML-Namensräume für die bereits bekannten beiden JSF-Komponentenbibliotheken HTML und Core. Da wir keine Facelets-eigenen Tags verwenden, wird der Facelets-Namensraum nicht deklariert. Das bereits angesprochene `<h:dataTable>`-Tag befindet sich in den Zeilen 13 bis 40. Der Wert des `value`-Attributs (ein EL-Ausdruck) evaluiert zu einer Methode, die alle Comedians zurückliefert. Das Attribut `var` deklariert die lokale Variable `comedian`, die im Rumpf der Datentabelle als Iterationsvariable verwendet wird. Die für die Formatierung der Seite benötigten CSS-Klassen werden mit den Attributen `headerClass`, `footerClass` und `rowClasses` eingebunden. Wir gehen auf CSS und CSS-Formatierungen an dieser Stelle nicht ein.

Eine Datentabelle kann aus beliebig vielen Spalten bestehen, die das `<h:column>`-Tag definiert. Spalten können wie Tabellen sowohl Über- als auch Unterschriften besitzen. Diese werden nicht über eigene Tags unterschieden, sondern

über Facetten. Eine Facette wird mit dem `<f:facet>`-Tag definiert. Der Wert des Attributs `name` entscheidet, welcher Art die Facette ist. Im Beispiel sind `header` und `footer` als Attributwerte verwendet worden.

Den Datenteil der beiden ersten Spalten bilden die EL-Ausdrücke `#{}comedian.vorname}` und `#{}comedian.nachname}`, die den Vor- und Nachnamen eines Comedian darstellen (Zeilen 20 und 24). Wir werden die dahinter stehende Java-Implementierung gleich näher betrachten. Die dritte Spalte ist etwas komplizierter aufgebaut. Damit der *Konvertierer* `<f:convertDateTime>` verwendet werden kann, muss ein anderes Tag dieses umschließen. Im `<h:outputText>`-Tag wird ebenfalls auf ein Property des Comedian zugegriffen, diesmal mit `#{}comedian.geburtstag}`. Die beiden weiteren Spalten bestehen aus Schaltflächen, die wir bereits aus dem Tic-Tac-Toe-Spiel kennen. Unter der Tabelle befindet sich eine weitere Schaltfläche für das Anlegen eines neuen Comedians. Abbildung 3.1 zeigt die Darstellung der Seite im Browser.

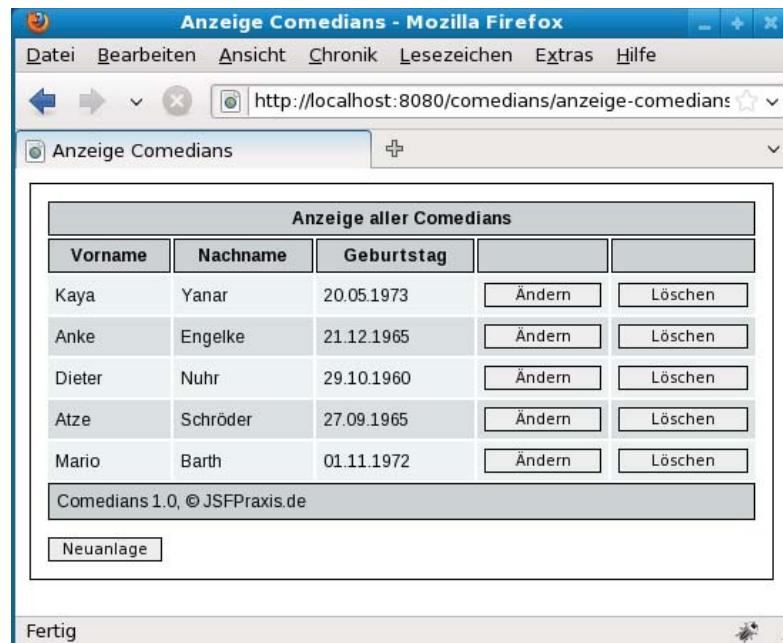


Abbildung 3.1: Anzeige aller Comedians

Die interessanten Fragen sind nun:

- Wie sieht das Property `comedians` aus oder, anders ausgedrückt, wie kommen die Daten aus der Datenbank in die JSF-Anwendung?

- Wie weiß JSF, für welche Comedians die Schaltflächen zum Ändern und Löschen zuständig sind?

3.2 Managed Beans

Listing 3.2 zeigt die ersten Zeilen der Managed Bean ComedianHandler.

Listing 3.2: Die Managed Bean ComedianHandler

```
@ManagedBean
@SessionScoped
public class ComedianHandler {

    @PersistenceContext
    private EntityManager em;

    @Resource
    private UserTransaction utx;

    private DataModel<Comedian> comedians;
    private Comedian aktuellerComedian = new Comedian();
    ...
}
```

Durch die Annotation `@ManagedBean` wird die Klasse `ComedianHandler` zu einer Managed Bean. Als Name wird standardmäßig der Klassenname verwendet und entsprechend den Java-Namensregeln angepasst. Im obigen Fall wird als Bean-Name `comedianHandler` verwendet. Eine explizite Namensvergabe ist ebenfalls möglich. Wir haben dies beim Tic-Tac-Toe bereits praktiziert. Die Annotation `@SessionScoped` deklariert als Scope der Managed Bean die Session.

Mit Java-EE 5 [URL-JSR254] wurde das Java-Persistence-API (JPA) 1.0 als Teil der EJB-3-Spezifikation [URL-JSR220] eingeführt. Es vereinfacht die Realisierung der Persistenz von Java-Klassen im Vergleich zu früheren Java-EE-Spezifikationen ganz erheblich. Ein zentrales Element von JPA ist das Konzept eines Entity-Managers, der für die Persistenz von Entities verantwortlich ist. Die Klasse `EntityManager` realisiert einen Entity-Manager, und die Instanz `em` wird über die Annotation `@PersistenceContext` in die Managed Bean *injiziert*. Java-EE 5 macht stark vom Konzept der *Dependency Injection* Gebrauch, deren Definition im Allgemeinen Martin Fowler [URL-DI] zuschrieben wird. Der Container injiziert dabei eine Entity-Manager-Instanz

ohne Zutun der Bean in die entsprechend annotierte Variable. Während die Annotation `@PersistenceContext` dem Namen entsprechend eine festgelegte Bedeutung hat, wird die Annotation `@Resource` verwendet, um verschiedene Arten von Ressourcen, in diesem Fall eine Transaktion, zu injizieren. Wir gehen darauf später ausführlicher ein.

Wir können uns nun der ursprünglichen Frage widmen, nämlich, wie das Property `comedians` die Comedians der Datenbank repräsentiert. Man erkennt die Deklaration der Instanzvariablen `comedians` vom Typ `DataModel<Comedian>`. Die Klasse `DataModel`, bzw. eine ihrer konkreten Unterklassen, ist ein Wrapper für eine Menge von Zeilenobjekten, in unserem Fall eine Menge von Comedians. Diese wird über den einfachen Getter `getComedians()` im EL-Ausdruck der JSF-Seite zurückgeliefert:

```
public DataModel<Comedian> getComedians() {  
    return comedians;  
}
```

Die Methode `getComedians()` könnte den Datenbankzugriff realisieren. Wir haben jedoch eine andere Alternative realisiert, um zunächst initiale Daten in die Datenbank zu schreiben. Die Methode `init()` realisiert genau dies:

```
@PostConstruct  
public void init() {  
    try {  
        utx.begin();  
        em.persist(new Comedian("Mario", "Barth",  
            new GregorianCalendar(1972, 10, 1).getTime()));  
        em.persist(new Comedian("Atze", "Schröder",  
            new GregorianCalendar(1965, 8, 27).getTime()));  
        em.persist(new Comedian("Dieter", "Nuhr",  
            new GregorianCalendar(1960, 9, 29).getTime()));  
        em.persist(new Comedian("Anke", "Engelke",  
            new GregorianCalendar(1965, 11, 21).getTime()));  
        em.persist(new Comedian("Kaya", "Yanar",  
            new GregorianCalendar(1973, 4, 20).getTime()));  
        comedians = new ListDataModel<Comedian>();  
        comedians.setWrappedData(  
            em.createNamedQuery("SelectComedians").getResultList());  
        utx.commit();  
    } catch (Exception e) {  
        Logger.getAnonymousLogger().log(Level.SEVERE,  
            "'init()' nicht geklappt", e.getMessage());  
    }  
}
```

Zunächst werden über die JPA-Methode `em.persist()` fünf Comedians eingefügt, um anschließend mit der JPA-Named-Query wieder gelesen zu wer-

den. Das beteiligte JPA-Entity erläutern wir in Abschnitt 3.3. Die Methode `setWrappedData()` der Klasse `DataModel` setzt schließlich die durch den Wrapper einzuschließenden Daten. Die Annotation `@PostConstruct` ist eine Java-EE-Annotation, die bewirkt, dass die annotierte Methode nach dem Aufruf des Konstruktors und nach der Ausführung aller Injektionen aufgerufen wird. Die zweite offene Frage war die der Identifikation des zu ändernden bzw. zu löschen Comedian, da die Schaltflächen in der JSF-Seite zeilenunabhängig definiert wurden. Der Schlüssel zur Antwort liegt in der Klasse `DataModel`, die eine Menge von Zeilenobjekten und ein *aktuelles* Zeilenobjekt repräsentiert. Das aktuelle Zeilenobjekt kann mittels `getRowData()` ermittelt werden. Diese Methode – sowie das zugrunde liegende Event-Model – erläutern wir in Abschnitt 4.5.5. So lässt sich in den Methoden zum Ändern und Löschen das aktuelle, d. h. das durch Betätigen der Schaltfläche ausgewählte Zeilenobjekt ermitteln. Die Methode `aendern()` ist damit sehr einfach aufgebaut:

```
public String aendern() {  
    aktuellerComedian = comedians.getRowData();  
    return "/comedian.xhtml";  
}
```

Es wird lediglich die Variable `aktuellerComedian` mit dem ausgewählten Comedian belegt. Doch warum liefert die Action-Methode einen String zurück, und wo wird dieser verwendet?

Das Tic-Tac-Toe-Spiel bestand aus nur einer JSF-Seite. Die Comedian-Anwendung besteht jedoch aus der bereits bekannten Seite zur Anzeige aller Comedians sowie einer zweiten für die Neuanlage eines Comedians bzw. das Ändern eines bereits existierenden Comedians. JSF besitzt eine Komponente, die die Navigation zwischen den einzelnen Seiten realisiert. Die Basis bilden Navigationsregeln, die in der JSF-Konfigurationsdatei `faces-config.xml` hinterlegt werden. Alternativ kann seit JSF 2.0 die Folgeseite als Seitenname der Action-Methode zurückgegeben werden. Im Beispiel wird also von der Seite `/anzeige-comedians.xhtml` im Fall des Wertes `"/comedian.xhtml"` als Ergebnis des Action-Methoden-Aufrufs zur Seite `/comedian.xhtml` navigiert. Diese ist in Listing 3.3 dargestellt, wobei wir uns auf das Formular beschränken.

Listing 3.3: Neuanlage/Ändern eines Comedians (`comedian.xhtml`)

```
<h:form>  
    <h:panelGrid columns="2" headerClass="header"  
                  footerClass="footer">  
        <f:facet name="header">Comedian anlegen / ändern</f:facet>
```

```
<f:facet name="footer">#{resourceBundle.copyright}</f:facet>
<h:outputLabel value="Vorname:" for="vorname" />
<h:panelGroup>
    <h:inputText id="vorname" required="true"
        value="#{comedianHandler.aktuellerComedian.vorname}" />
        <h:message for="vorname" styleClass="message" />
</h:panelGroup>
<h:outputLabel value="Nachname:" for="nachname" />
<h:panelGroup>
    <h:inputText id="nachname" required="true"
        value="#{comedianHandler.aktuellerComedian.nachname}" />
        <h:message for="nachname" styleClass="message" />
</h:panelGroup>
<h:outputLabel value="Geburtstag:" for="geburstag" />
<h:panelGroup>
    <h:inputText id="geburstag" required="true"
        value="#{comedianHandler.aktuellerComedian.geburstag}">
        <f:convertDateTime pattern="dd.MM.yyyy"/>
    </h:inputText>
    <h:message for="geburstag" styleClass="message" />
</h:panelGroup>
</h:panelGrid>
<h:commandButton action="#{comedianHandler.speichern}"
    value="Speichern" />
</h:form>
```

Man erkennt ein einfaches Panel-Grid wie im Tic-Tac-Toe-Spiel, das zu einer HTML-Tabelle gerendert wird. Die weiteren JSF-Tags sind intuitiv verständlich. Abbildung 3.2 zeigt die Seite, nachdem die Schaltfläche Ändern für Mario Barth betätigt wurde.

Die Schaltflächen, die sich in der fünften Spalte der Comedian-Übersicht befinden, sind an die Action-Methode `loeschen()` gebunden:

```
1  public String loeschen() {
2      aktuellerComedian = comedians.getRowData();
3      try {
4          utx.begin();
5          aktuellerComedian = em.merge(aktuellerComedian);
6          em.remove(aktuellerComedian);
7          comedians.setWrappedData(
8              em.createNamedQuery("SelectComedians").getResultList());
9          utx.commit();
10     } catch (Exception e) {
11         Logger.getAnonymousLogger().log(Level.SEVERE,
12             "'loeschen()' nicht geklappt", e.getMessage());
13     }
14     return null;
15 }
```

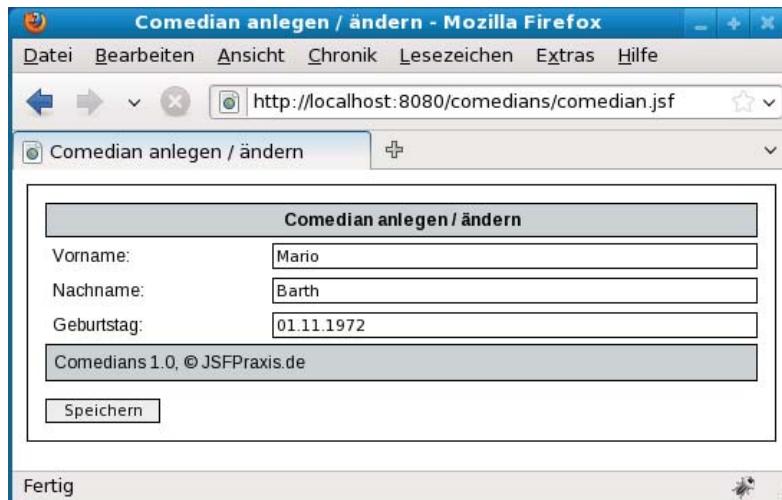


Abbildung 3.2: Ändern eines Comedians

Man erkennt auch hier den Zugriff auf den Comedian der aktuellen Tabellenzeile mit der Methode `getRowData()` in Zeile 2. Der injizierte Transaktionsmanager `utx` klammert die JPA-Zugriffe in einer Transaktion (`begin()`, `commit()`). Ohne in die Tiefen der JPA-Programmierung einsteigen zu wollen, bemerken wir hier nur, dass der in einer HTTP-Response angezeigte Comedian im nächsten HTTP-Request gelöscht werden soll. Der Comedian wird zwischen den HTTP-Requests der Kontrolle des Entity-Managers entzogen, so dass er zunächst wieder der Kontrolle des Entity-Managers zugeführt werden muss. Dies geschieht durch die Methode `merge()` in Zeile 5. Die Methode `remove()` löscht den Comedian. Der Code in den Zeilen 7 und 8 ist identisch mit dem entsprechenden Code der `init()`-Methode. JPA ermöglicht es, mit sogenannten Named-Queries Code-Redundanz auf Query-Ebene zu vermeiden. Die Action-Methode `loeschen()` liefert `null` zurück. Dieser Rückgabewert bewirkt, dass die dargestellte Seite nicht gewechselt wird, also keine Navigation stattfindet. Als letztes noch nicht analysiertes Element der Seite zur Anzeige aller Comedians bleibt die Schaltfläche Neuanlage. Sie ist an die Action-Methode `neuanlage()` gebunden:

```
public String neuanlage() {  
    aktuellerComedian = new Comedian();  
    return "/comedian.xhtml";  
}
```

Die Methode erzeugt lediglich einen neuen Comedian und belegt die Variable `aktuellerComedian` mit diesem Objekt. In der JSF-Seite `comedian.xhtml`

werden die EL-Ausdrücke `#{}comedianHandler.aktuellerComedian.vorname}` und `#{}comedianHandler.aktuellerComedian.nachname}` daher zu einem leeren String ausgewertet, so dass die Seite wie in Abbildung 3.2, jedoch mit leeren Eingabefeldern erscheint.

Die drei Einfabefelder sind mit dem Attribut `required="true"` versehen. Dies bedeutet, dass JSF überprüft, ob eine Eingabe vorhanden ist. Im negativen Fall wird mit einer Fehlermeldung reagiert. Abbildung 3.3 zeigt die Seite im Browser für den Fall, dass das Eingabefeld für den Nachnamen frei gelassen wurde. Wir gehen auf die Standardfehlermeldung und die Möglichkeiten, diese zu ändern, nicht ein.

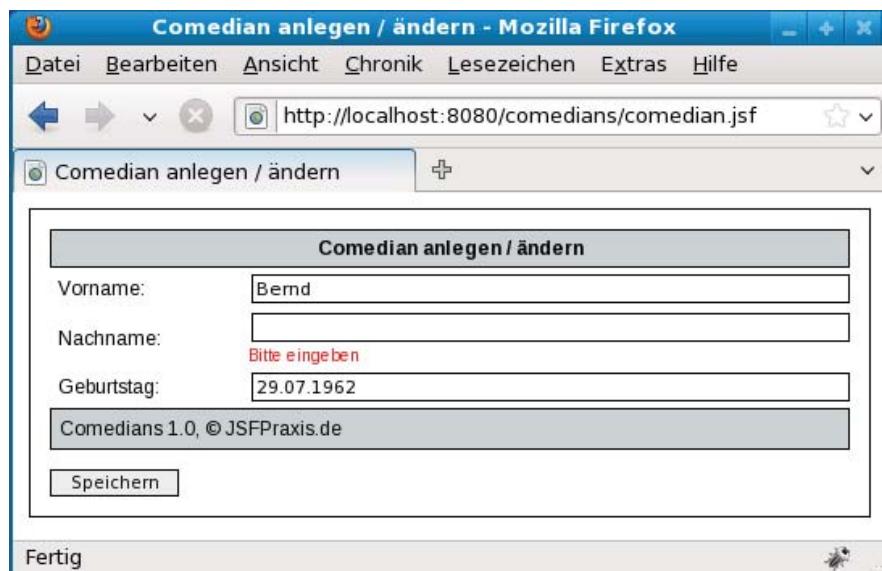


Abbildung 3.3: Fehler beim Anlegen eines Comedians

Falls sowohl Vorname, Nachname und Geburtsdatum eingegeben wurden, führt das Betätigen der Schaltfläche Speichern zum Aufruf der Methode `speichern()`:

```
public String speichern() {
    try {
        utx.begin();
        aktuellerComedian = em.merge(aktuellerComedian);
        em.persist(aktuellerComedian);
        comedians.setWrappedData(
            em.createNamedQuery("SelectComedians").getResultList());
        utx.commit();
    } catch (Exception e) {
        Logger.getAnonymousLogger().log(Level.SEVERE,
```

```
        ''speichern()', nicht geöffnet", e.getMessage());
    }
    return "/anzeige-comedians.xhtml";
}
```

Der Code ähnelt dem Code für die Löschen-Methode mit Ausnahme der `persist()`-Methode. Diese wird in JPA verwendet, um ein Entity in die Datenbank zu schreiben. Da die `speichern()`-Methode sowohl bei der Änderung als auch bei der Neuanlage eines Entitys verwendet wird, bedarf es einer genaueren Analyse.

Der Parameter der `merge()`-Methode ist entweder ein durch die Methode `neuanlage()` neu erstelltes und nicht persistentes Comedian-Objekt oder ein durch die `aendern()`-Methode selektiertes und somit persistentes Comedian-Objekt. Im ersten Fall wird durch die `persist()`-Methode ein SQL-INSERT-Statement, im zweiten Fall durch die `merge()`-Methode ein SQL-UPDATE-Statement erzeugt und ausgeführt. Wir gehen an dieser Stelle nicht weiter auf die Tiefen der JPA-Programmierung ein und verweisen auf den nächsten Abschnitt über JPA und unser Buch [MW07].

Zum Abschluss des Beispiels soll die Fußzeile der beiden Tabellen näher analysiert werden. Java und JSF sind von Haus aus auf Internationalisierung und Lokalisierung vorbereitet. Über ein Resource-Bundle können für Variablen Texte mehrsprachig hinterlegt werden. Das haben wir nicht gemacht, sondern den Mechanismus verwendet, um Code-Redundanzen zu reduzieren. Über den sowohl in der Übersichts- als auch der Detailseite vorhandenen EL-Ausdruck `#{{resourceBundle.copyright}}` wird ein in einer Datei definiertes Property referenziert:

```
copyright=Comedians 1.0, \u00a9 JSFPraxis.de
```

Der Name des Resource-Bundles, `resourceBundle`, wird in der JSF-Konfigurationsdatei `faces-config.xml` definiert. Die Verwendung von Resource-Bundles zur Internationalisierung wird in Abschnitt 4.7 erläutert und soll hier nicht weiter vertieft werden. Facelets stellen einen sehr effektiven Template-Mechanismus bereit, mit dem man Code-Redundanzen auf Seitenebene wirkungsvoll vermeidet. Falls keine mehrsprachige Anwendung zu entwickeln ist, sollte man die Facelets-Lösung der hier dargestellten vorziehen. Wir gehen im Kapitel 6 auf Facelets ein.

3.3 JPA

In diesem Kapitel haben wir eine JSF-Anwendung vorgestellt, um das Interesse am nächsten Kapitel zu wecken, in dem wir detailliert die Funktionsweise von JSF darstellen. Hier wollen wir lediglich der Vollständigkeit halber auf das JPA-Entity Comedian eingehen und stellen es überblicksartig vor. Die Annotation `@Entity` bewirkt, dass Instanzen der Klasse von JPA verwaltet werden können. Mit `@Id` und `@GeneratedValue` wird das annotierte Property zum Primärschlüsselproperty, und JPA wird angewiesen, den Primärschlüssel automatisch zu vergeben. Die Annotation `@NamedQuery` definiert schließlich einen Namen für eine JPA-Query. Diese Query haben wir mehrmals in den entsprechenden Methoden des Comedian-Handlers verwendet.

```
    @NamedQuery(name="SelectComedians",
                 query="Select c From Comedian c")
    @Entity
    public class Comedian implements Serializable {

        private Integer id;
        private String vorname;
        private String nachname;
        private Date geburtstag;

        public Comedian() {
        }

        public Comedian(String vorname, String nachname,
                        Date geburtstag) {
            this.vorname = vorname;
            this.nachname = nachname;
            this.geburtstag = geburtstag;
        }

        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        public Integer getId() {
            return id;
        }
        public void setId(Integer id) {
            this.id = id;
        }

        // restlichen Getter/Setter
    }
```

Mit dem Tic-Tac-Toe-Spiel und der Comedian-Verwaltung realisierten wir zwei kleine Beispielanwendungen mit JSF. Bei den Erläuterungen zum Code haben

wir Begriffe wie Eingabekomponente, Action-Methode, EL-Ausdruck, Navigation, Validierung, Konvertierung, Fehlermeldung, Resource-Bundle und einige mehr verwendet. Im Kapitel 4 werden diese Begriffe ausführlich eingeführt.



Projekt

Der beschriebene Code für die Comedian-Verwaltung ist im Projekt *comedians* enthalten.

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Kapitel 4

JavaServer Faces im Detail

Nachdem wir die Comedian-Anwendung erfolgreich entwickelt haben, ist es an der Zeit, die Grundlagen der JavaServer Faces zu erörtern. Dieses Kapitel beschreibt detailliert die für ein vollständiges Verständnis von JavaServer Faces benötigten Konzepte und Hintergründe, z.B. wie die einzelnen Verarbeitungsschritte einer JSF-Anfrage aussehen, wie existierende Validierer und Konvertierer eingesetzt und eigene entwickelt, wie Events verarbeitet werden und vieles mehr.

Zentral für das Verständnis der inneren Arbeitsweise von JSF ist es, zu verinnerlichen, dass JSF ein server-seitiges Komponenten-Framework ist. Alle JSF-Komponenten, auch wenn sie letztendlich z.B. als Texteingaben oder Schaltflächen auf dem Client dargestellt werden, sind Instanzen bestimmter Java-Klassen auf dem Server. Die komplette Verarbeitung und der Umgang mit diesen Komponenten geschieht auf dem Server, und lediglich am Ende einer Anfrage-Bearbeitung wird die Antwort an den Client geschickt. Diese Antwort ist aber immer ein Spiegelbild der Komponenten auf dem Server und kein selbstständiges Artefakt.

4.1 Bearbeitungsmodell einer JSF-Anfrage

JavaServer Faces werden mit Hilfe des Servlet-APIs realisiert. Servlets wiederum basieren auf dem Request-Response-Modell des zugrunde liegenden HTTP-Protokolls. JavaServer Faces erben somit die Eigenheiten einer HTTP-Anfrage und einer HTTP-Antwort, versuchen aber möglichst viel von diesem Erbe zu verstecken. Insbesondere die Zustandslosigkeit von HTTP, für deren Umgebung bei einer Servlet-Anwendung viel Aufwand investiert werden

muss, wird durch ein vollständiges MVC-Konzept ersetzt, bei dem Zustände eine wichtige Rolle spielen. So kann etwa der Controller mittels Change-Event die Änderung einer Benutzereingabe zwischen zwei HTTP-Anfragen erkennen. Um dies zu ermöglichen, muss ein Abbild des Zustands gespeichert und bei jeder neuen Anfrage mit dem dann neuen, aktuellen Zustand verglichen werden. Validierungen und Konvertierungen müssen durchgeführt, Events verarbeitet, aber eventuell auch neue erzeugt werden. Dieses durchaus komplexe und umfangreiche Verfahren wird in der Spezifikation „*Request Processing Lifecycle*“ genannt; es ist Gegenstand dieses Abschnitts. Wir sprechen im Folgenden von *Bearbeitungsmodell* oder *Lebenszyklus*.

Tabelle 4.1: Möglichkeiten von Anfragen und Antworten

	JSF-Anfrage	andere Anfrage
JSF-Antwort	1	2
andere Antwort	3	4

Prinzipiell kann eine HTTP-Anfrage einer JSF-Seite von einer JSF-Seite oder einer Nicht-JSF-Seite kommen. Genauso kann eine JSF-Seite eine JSF-Antwort oder eine Nicht-JSF-Antwort generieren. Man kann also vier Fälle unterscheiden, die in Tabelle 4.1 dargestellt sind. Unter einer JSF-Anfrage versteht man eine Anfrage, die durch eine zuvor generierte JSF-Antwort, z. B. ein durch JavaServer Faces erzeugtes HTML-Formular, ausgelöst wurde. Man spricht auch von einem *Post-Back*. Eine solche Anfrage enthält immer die Id einer View. Eine JSF-Anfrage kann aber auch eine Anfrage nach Teilen einer Seite sein, etwa einer CSS- oder JavaScript-Datei. Eine andere (Nicht-JSF-)Anfrage ist z. B. ein gewöhnlicher HTML-Verweis.

Eine JSF-Antwort ist eine Antwort, die von der letzten Phase der Anfragebearbeitung, der Render-Phase, erzeugt wurde. Eine andere (Nicht-JSF-)Antwort ist z. B. eine normale HTML-Seite, ein PDF-Dokument oder Teile einer HTML-Seite, z. B. eine CSS- oder JavaScript-Datei. Die Spezifikation spricht in diesem Zusammenhang von „Faces Request“ und „Faces Response“ bzw. von „Non-Faces Request“ und „Non-Faces Response“. Für den Fall von Anfragen oder Antworten von JSF-Teilbereichen spricht die Spezifikation von „Faces Resource Request“ und „Faces Resource Response“.

Es ist offensichtlich, dass die vierte Möglichkeit der Tabelle 4.1 nichts mit JavaServer Faces zu tun hat. Auch die zweite und dritte Möglichkeit sind Sonderfälle. Die folgenden Ausführungen beziehen sich auf die erste Möglichkeit, bei der eine JSF-Anfrage eine JSF-Antwort nach sich zieht. Die seit der Version 2.0 möglichen Ajax-Requests behandeln wir gesondert in Kapitel 7.

Die Bearbeitung einer JSF-Anfrage beginnt, wenn das JSF-Servlet den HTTP-Request erhalten hat. Es gibt insgesamt sechs zu unterscheidende Bearbeitungsphasen. Zwischen diesen sind Event-Verarbeitungsphasen vorgesehen. Abbildung 4.1 veranschaulicht dies grafisch.

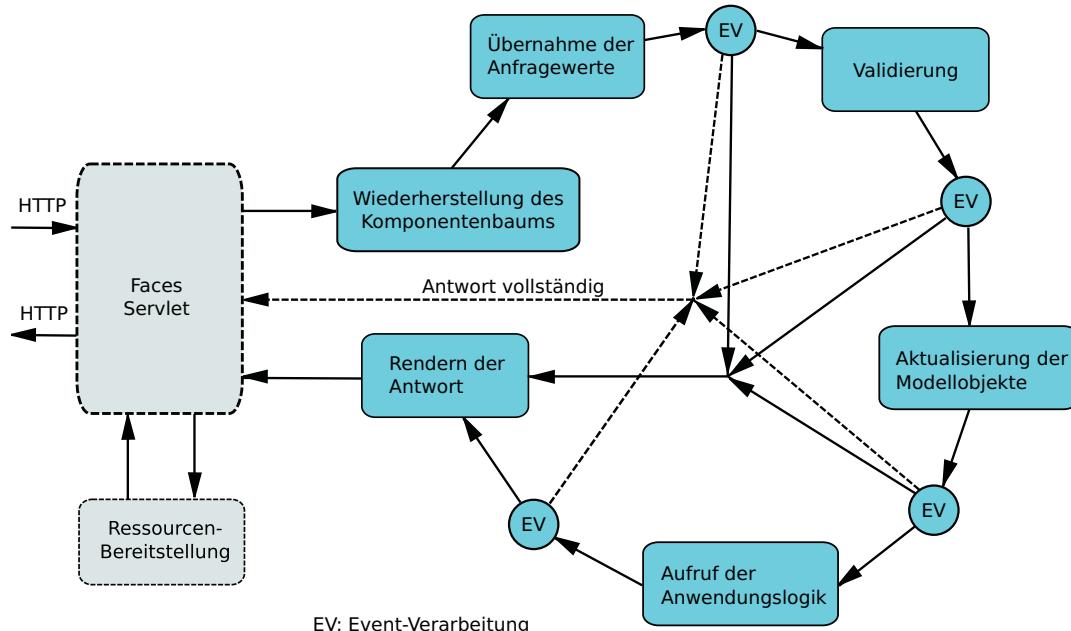


Abbildung 4.1: Bearbeitungsmodell einer JSF-Anfrage

Zunächst beginnen wir mit dem Faces-Servlet. JavaServer Faces sind mit einem Servlet realisiert, das im Deployment-Deskriptor der Servlet-Anwendung definiert werden muss (Abschnitt 4.8.1) bzw. in einer Servlet-Umgebung der Version 3.0 automatisch erkannt wird. Dieses Servlet nimmt HTTP-Anfragen entgegen. Zunächst wird entschieden, ob es sich um eine Ressourcen-Anfrage handelt, beispielsweise um eine JavaScript-Datei. Wenn ja, wird diese Ressource als HTTP-Antwort zurückgeschickt. Falls der Request keine Ressourcen-Anfrage ist, wird das Bearbeitungsmodell angestoßen.

Im Modell werden die sechs Phasen als Rechtecke dargestellt. Man erkennt, dass nach den inneren Phasen (zwei bis fünf) jeweils Events an interessierende Event-Listener übergeben werden können. Bei deren Abarbeitung können JSF-Komponenten verändert oder Anwendungsdaten verarbeitet werden. Man

kann auch komplett auf das Rendern der Antwort verzichten und direkt an das Ende der Bearbeitung springen. Dies ist durch die gestrichelten Linien in Abbildung 4.1 angedeutet. Beispiele hierfür sind binäre Daten, wie etwa eine JPG-Grafik oder ein PDF-Dokument, die als komplette Antwort zurückgeliefert werden und nicht Bestandteil einer HTML-Seite sind. Bei Validierungs- und Konvertierungsfehlern werden die nachfolgenden Phasen übersprungen und nur noch die Antwort generiert, die dann in der Regel Meldungen über den Fehler enthält.

Mit der Einführung von Ajax-Requests in JSF 2.0 ändert sich das dargestellte Verhalten etwas. Die sechste Phase, das Rendern der Antwort-Seite, entfällt, da lediglich eine XML-Struktur und keine komplette Seite zurückgegeben wird. Die ersten fünf Phasen werden auf einen oder mehrere Teilbereiche der Seite beschränkt und arbeiten nicht auf dem gesamten Komponentenbaum. Wir gehen auf die mit Ajax verbundene Komplexität in den nächsten Abschnitten zunächst nicht ein, holen dies aber in Kapitel 7 nach.

4.1.1 Wiederherstellung des Komponentenbaums

JSF-Komponenten besitzen einen Zustand, der bei einer Drop-Down-Liste z.B. die aktuelle Selektion in der Liste enthält. Die View ist eine nicht sichtbare Komponente, die die Wurzel des Baums aller Komponenten dieser Seite darstellt. Da die Zeit zur Beantwortung einer Anfrage wesentlich kürzer als die Zeit zwischen zwei Anfragen derselben Session ist und man nicht alle Komponenten einer Seite zwischen zwei Anfragen benötigt, werden die Komponenten einer Seite nach Beantwortung einer Anfrage gespeichert. Auf diese Weise spart man bei Anwendungen mit Tausenden von Benutzern viel Speicherplatz. Zu Beginn einer Anfragebearbeitung muss daher zunächst der Komponentenbaum wiederhergestellt werden.

Die Speicherung der Komponenten und deren Zustände kann auf dem Client oder dem Server erfolgen. Die Konfiguration der Speicherart wird in Abschnitt 4.8 erläutert. Jede View hat eine View-ID, die das URI der Anfrage darstellt. Bei der Bearbeitung eines Comedians in der Anwendung aus Kapitel 3 (Abbildung 3.2 auf Seite 29) gehört „/comedians“ noch zum frei konfigurierbaren Anwendungsnamen, die View-ID ist somit „/comedian.jsf“. Die View-ID wird in der Session gespeichert. Bei einer Anfrage kann daher entschieden werden, ob die Anfrage von einer JSF-Seite initiiert wurde (Alternative 1 in Tabelle 4.1) oder ob die Seite zum ersten Mal besucht wird (Alternative 2 in Tabelle 4.1). Bei der ersten Alternative wird der gespeicherte Komponentenbaum im alten Zustand wiederhergestellt, bei der zweiten wird ein neuer Komponentenbaum erstellt. Der wiederhergestellte oder neu erstellte Kompo-

nentenbaum wird dann im aktuellen `FacesContext` gespeichert. Ein Objekt der Klasse `FacesContext` im Package `javax.faces.context` enthält alle Informationen, die im Zusammenhang der Verarbeitung *einer* JSF-Anfrage stehen. Wir werden den `FacesContext` noch häufiger verwenden.

Die Wiederherstellung des Komponentenbaums umfasst nicht nur die Zustände der Komponenten, sondern auch das Wiederherstellen aller mit den Komponenten verbundenen Event-Listener, Validierer, Konvertierer und Managed Beans. In dieser Phase erfolgt ebenfalls die Lokalisierung für die View. Besucht man die Seite zum ersten Mal (Alternative 2), wird dann zur letzten Phase, der Render-Phase, gesprungen, ansonsten erfolgt die Übernahme der Anfragewerte.

4.1.2 Übernahme der Anfragewerte

Einige UI-Komponenten lassen die Eingabe von Werten durch den Benutzer zu, sei es als Text oder als Auswahl von Alternativen. Diese Eingaben werden durch das zugrunde liegende Formular als POST-Parameter des HTTP-Requests codiert. Das JSF-Framework muss sie in dieser Phase *decodieren*, d. h. dem Request entnehmen, und der entsprechenden UI-Komponente zuweisen. Ein Beispiel: In der Comedian-Anwendung wird bei der Detailseite zur Eingabe des Comedian-Vornamens ein `<h:inputText>`-Element verwendet:

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

Der durch JavaServer Faces generierte HTML-Code sieht dann so aus :

```
<input id="j_idt3:vorname" type="text" value=""  
name="j_idt3:vorname"/>
```

In diesem Fall ist er von der Referenzimplementierung generiert worden. Eine andere Implementierung erzeugt evtl. anderen Code. Die Id `j_idt3:vorname` ist durch die Spezifikation strukturell definiert, wobei der Präfix generiert wurde, da das Formular selbst keine explizite Id hat. Der folgende Code ist dem POST-Request entnommen:

```
...&j_idt3%3Avorname=Mario&j_idt3%3Anachname=Barth&...
```

Man erkennt im hinteren Teil, dass dem Parameter `j_idt3:vorname` (der Doppelpunkt ist durch „%3A“ codiert) der Wert „Mario“ zugewiesen wird. Es ist Aufgabe dieser Phase, den gesamten POST-String zu parsen und alle Parameter mit ihren jeweiligen Werten herauszufiltern, zu decodieren. Die Werte werden dann vorläufig den entsprechenden Komponenten zugewiesen. Vorläufig

deshalb, weil in den nachfolgenden Validierungen und Konvertierungen noch Fehler auftreten können.

Alle Eingabe- und Steuerkomponenten besitzen ein boolesches Attribut `immediate`. Ist dieses bei Eingabekomponenten gesetzt, finden Validierung und Konvertierung bereits in dieser Phase (Übernahme der Anfragewerte) und nicht in der nächsten Phase, der Validierung, statt. Ist das Attribut `immediate` bei Steuerkomponenten gesetzt, findet der Aufruf der Action-Methoden bzw. Action-Listener am Ende dieser Phase und nicht in der Phase *Aufruf der Anwendungslogik* statt. Wir gehen auf beide Alternativen an den entsprechenden Abschnitten ausführlich ein.

Am Ende der Phase zur Übernahme der Anfragewerte werden alle existierenden Events an die interessierten Listener weitergereicht. Die JSF-Implementierung kann zur Render-Phase springen oder die Bearbeitung der Anfrage komplett beenden.

4.1.3 Validierung

Zu Beginn der Validierungsphase ist sichergestellt, dass alle aktuellen Anfrageparameterwerte für ihre UI-Komponenten bereitstehen. Die JSF-Implementierung durchläuft nun den Komponentenbaum und stellt sicher, dass alle Werte valide sind. Dazu werden alle registrierten Validierer (einige Komponenten besitzen zusätzlich eigene Validierer) zur Validierung aufgefordert. Eventuell müssen vor der Validierung noch Konvertierungen vorgenommen werden.

In der Comedian-Anwendung wird für jede Eingabe (Vorname, Nachname, Geburtstag) validiert, ob das Eingabefeld einen Wert enthält (`required="true"`). Weitere Validierungen sind möglich. So wäre es z. B. sinnvoll zu prüfen, ob der Vor- und Nachname eine Minimal- und eine Maximallänge hat. Dies ist mit JSF leicht möglich:

```
<h:inputText id="nachname" required="true"
    value="#{comedianHandler.aktuellerComedian.nachname}">
    <f:validateLength minimum="3" maximum="20" />
</h:inputText>
```

Die in der Spezifikation *Process Validation* genannte Phase besteht neben der Validierung in der Regel auch aus einer zuvor vorzunehmenden Konvertierung. In der Comedian-Anwendung wird etwa ein den Geburtstag eines Comedians repräsentierender String in ein Java-Objekt der Klasse `java.util.Date` konvertiert, während Vor- und Nachname ohne Konvertierung übernommen werden können. Nach der Datumskonvertierung könnte man eine weitere Validierung, etwa die Überprüfung, dass das Geburtsdatum in der Vergangenheit

liegt, vornehmen. Hierfür stellt JSF jedoch keinen vordefinierten Validierer bereit. Mit der ab JSF 2.0 verfügbaren Bean-Validierung, die wir in Abschnitt 4.4.9 vorstellen, ist diese Überprüfung allerdings direkt realisierbar.

Nachdem alle Validierungen und Konvertierungen mit Erfolg durchgeführt wurden, wird der Wert nun endgültig der Komponente zugewiesen. Sollte sich der Wert seit der letzten Anfrage geändert haben, wird ein Value-ChangeEvent geworfen und an registrierte Listener weitergegeben. Diese Listener können nun zur Render-Phase springen, direkt die Antwort erzeugen oder zur Aktualisierung der Modellobjekte übergehen.

4.1.4 Aktualisierung der Modellobjekte

Bis zu diesem Zeitpunkt haben alle Vorgänge in den UI-Komponenten stattgefunden. Die Werte sind valide und vom richtigen Typ und können nun den Modellobjekten zugewiesen werden. In unserem Beispiel

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

ist der Ausdruck "#{comedianHandler.aktuellerComedian.vorname}" dafür zuständig. Die JSF-Implementierung sucht nach einer Managed Bean mit dem Namen `comedianHandler`. Dann wird das `aktuellerComedian`-Property dieses Objekts ausgewertet. Das Property `vorname` des Ergebnisobjekts bekommt den Wert der UI-Komponente zugewiesen. Wie am Ende jeder Phase werden wieder Events geworfen, Listener informiert, und eventuell wird an das Ende der Bearbeitung gesprungen.

4.1.5 Aufruf der Anwendungslogik

Bis zu diesem Zeitpunkt haben wir noch keinen Applikations-Code verwendet, obwohl bereits relativ viel Aufwand betrieben wurde: Benutzereingaben wurden konvertiert und validiert und die Properties der Managed Beans aktualisiert. Die JSF-Implementierung hat alles automatisch erledigt.

Nun kann die Anwendungslogik aufgerufen werden. Dies geschieht durch Listener, die sich auf Action-Events registriert haben, die durch Schaltflächen oder Hyperlinks ausgelöst werden können. Dabei sind zwei Arten von Listener zu unterscheiden. Die erste Art sind die mit dem Attribut `actionListener` registrierten „richtigen“ Listener. Ein Beispiel sind die neun Schaltflächen des Tic-Tac-Toe-Spiels (siehe Listing 2.2 auf Seite 14):

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

Sie registrieren die Listener-Methode `zug`. Eine solche Listener-Methode muss als Parameter ein `ActionEvent` haben (siehe Listing 2.4 auf Seite 17):

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

Die zweite Art sind die automatisch für Steuerkomponenten registrierten Default-Action-Listener. Dies wird in der Komponente durch ein `action`-Attribut und im Handler durch eine Methode ohne Parameter erfüllt (Beispiel: Speichern eines Comedians):

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

Hier wird die Action-Methode `speichern()` registriert. Eine Action-Methode muss einen String zurückliefern und hat keinen Parameter:

```
public String speichern() {  
    ...
```

Seit JSF 1.2 ist der Rückgabewert von Action-Methoden auf `Object` erweitert worden, um Enumerations als mögliche Rückgabetypen zu ermöglichen. Für das Rückgabeobjekt wird die `toString()`-Methode aufgerufen, um den so erhaltenen String zur Navigation verwenden zu können.

Action-Methoden sind nicht auf die Anwendungslogik beschränkt. Sie können z. B. auch Events generieren, Anwendungsmeldungen erzeugen oder gar die Antwort selbst rendern.

4.1.6 Rendern der Antwort

Nach der Abarbeitung der Anwendungslogik bleiben in der letzten Phase der Anfragebearbeitung noch das Rendern der Antwort und das Abspeichern des Komponentenbaums als zentrale Aufgaben.

Die Zielsprache des Renderns der Antwort ist durch die Spezifikation bewusst offen gelassen worden. Es sind mehrere Alternativen denkbar, z. B. XHTML, JSP, WML oder SVG. Bis einschließlich JSF 1.2 war festgelegt, dass jede Implementierung mindestens JSP als Anzeigetechnik unterstützen muss. Mit JSF 2.0 kam die *Page-Description-Language (PDL)* auf der Basis von Facelets hinzu, auf die wir ausführlich in Abschnitt 5.6 eingehen. Die Neuerungen der Version 2.0 sind nur in Facelets enthalten, JSP wird nur noch aus Kompatibilitätsgründen weitergeführt.

Während bei der zweiten Phase, der Übernahme der Anfragewerte, die Komponentenwerte decodiert werden mussten, müssen Sie nun codiert werden. Das

Geburtsdatum eines Comedians ist etwa ein `Date`-Objekt, das allerdings in der HTML-Seite als String darzustellen ist.

Da der Komponentenbaum programmatisch in der Phase 5, *Aufruf der Anwendungslogik*, verändert werden kann, ist es möglich, dass mehr, aber auch weniger Komponenten zu rendern sind als im Ursprungs-Code der JSF-Seite vorhanden.

Zuletzt muss das Abspeichern des Komponentenbaums so erfolgen, dass bei einer erneuten Anfrage der Seite der Komponentenbaum in seinem dann ursprünglichen Zustand wiederhergestellt werden kann.

Die Version 2.0 führt das sogenannte *Partial State Saving* ein. Da der Komponentenbaum in der Regel nur selten und in geringem Umfang verändert wird, speichert die JSF-Implementierung nur die Änderungen am Komponentenbaum. Die aktuelle Version kann dann über die textuelle Repräsentation der JSF-Seite und die abgespeicherten Änderungen konstruiert werden. Ob das partielle Speichern des Komponentenbaums tatsächlich stattfindet, wird über eine Konfigurationsoption bestimmt.

4.2 Expression-Language

Die JSTL und JSP führten eine Expression-Language (kurz EL) ein, um Entwicklern von JSP-Seiten eine Möglichkeit für den einfachen Zugriff auf Anwendungsdaten zu ermöglichen, ohne den Weg über Java gehen zu müssen. Bei der Definition von JSF wurden die Möglichkeiten einer Expression-Language ebenfalls als sehr wichtiges Element einer Seitenbeschreibungssprache erkannt, und eine Expression-Language sollte integraler Bestandteil von JSF sein. JSF stellt jedoch andere Anforderungen an eine Expression-Language, so dass für JSF eine eigene Expression-Language definiert wurde.

Die Verwendung von JSP als Seitenbeschreibungssprache für JSF erlaubt die Verwendung der JSP-EL als auch der JSF-EL innerhalb einer Seite. Dies führte in den JSF-Versionen 1.0 und 1.1 zu Problemen und letztendlich zur Definition der *Unified Expression-Language* [URL-UEL].

Die Unified EL ist als Teildokument der JavaServer-Pages-Spezifikation 2.1 definiert [URL-JSR245] und wird in dieser Form in JSF 1.2 und ein wenig erweitert in JSF 2.0 verwendet. Die Unified EL unterscheidet zwischen sofortiger Auswertung (sogenannte *Immediate Expressions*, beginnend mit einem `$`-Zeichen) zum Zeitpunkt des Renderns der Seite (bei JSPs der Compile-Zeitpunkt) und der verzögerten oder zeitversetzten Auswertung (sogenannte *Deferred Expressions*, beginnend mit einem `#`-Zeichen) zur Laufzeit. Die zeit-

verzögerte Auswertung macht sich insbesondere dadurch bemerkbar, dass EL-Ausdrücke zweimal ausgewertet werden, und zwar während der Übernahme der Anfragewerte (Phase 2) als auch dem Rendern der Antwort (Phase 6). Werteausdrücke werden daher sowohl schreibend (Phase 2) als auch lesend (Phase 6) verwendet.

Durch die Einführung der auf Facelets basierenden PDL in JSF 2.0 ist es in der Regel nicht notwendig, JSP in JSF-Anwendungen zu verwenden. Wir raten dem Leser, auf die Verwendung von JSP vollständig zu verzichten, um nach wie vor vorhandene Probleme beim Mischen beider Technologien zu vermeiden. Im Folgenden verwenden wir ausschließlich Facelets und zeitversetzte Ausdrücke und können daher auf eine Unterscheidung verzichten.

4.2.1 Syntax

Die Expression-Language enthält Konzepte, wie man sie auch in JavaScript und XPath findet. In der Expression-Language navigiert man durch eine Punkt-Notation über Objekt-Properties, so wie man in XPath im Baum des XML-Dokuments navigiert. Die Properties werden bei der Auswertung automatisch mit `get` und `set` erweitert und die Großschreibung den Java-Konventionen angepasst. Wir gehen darauf später noch genauer ein.

Zunächst wollen wir aber die grundlegende Syntax der Expression-Language einführen. Ausdrücke der Expression-Language schreiben wir als String und schließen sie durch das Rautezeichen (#, üblich sind auch die Bezeichnungen „Hash“, Nummernzeichen oder Lattenzaun) und geschweifte Klammern ein:

```
"#{expr}"
```

Innerhalb dieser Klammern steht der eigentliche EL-Ausdruck. Er kann einen Werteausdruck, einen Methodenausdruck, aber auch einen einfachen arithmetischen oder logischen Ausdruck enthalten. Beliebige Kombinationen der genannten Alternativen (ohne Methodenausdrücke) sind ebenfalls möglich. Das Verschachteln von Ausdrücken der Art `#{field[#{i}]}` ist nicht erlaubt. Anstatt Anführungszeichen ist alternativ die Verwendung einfacher Apostrophe erlaubt. Wir empfehlen die konsequente Verwendung von Anführungszeichen.

4.2.2 Bean-Properties

Über einen *Werteausdruck* (engl. Value Expression) kann der Wert einer UI-Komponente an eine Bean-Property oder die UI-Komponente selbst an eine Bean-Property gebunden werden. Werteausdrücke werden auch verwendet, um Properties einer UI-Komponente zu initialisieren.

Über einen *Methodenausdruck* (engl. Method Expression) lässt sich eine Bean-Methode referenzieren. Dies wird z. B. bei Event-Handlern und Validierungsmethoden verwendet.

Wir konzentrieren uns zunächst auf den einfachsten Fall: den Werteausdruck. Ein Werteausdruck muss zu einer einfachen Bean-Property, einem Element eines Arrays, einer Liste (`java.util.List`) oder einem Eintrag in einer Map (`java.util.Map`) evaluieren. Die Instanz, die der EL-Ausdruck referenziert, kann sowohl gelesen als auch geschrieben werden. Das Schreiben in die Instanz geschieht in der Phase *Aktualisierung der Modellobjekte*, das Lesen in der Phase *Rendern der Antwort*.

Einige kleine Beispiele für verschiedene Property-Zugriffe sind in der JSF-Seite `el.xhtml` enthalten, die wir uns ausschnittweise anschauen:

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">Zugriff auf Bean-Properties</f:facet>
3   <h:outputText value="#{elHandler.name}" />
4   <h:outputText value="#{elHandler['name']}" />
5   <h:outputText value="Dies sind tolle #{elHandler.name}" />
6   <h:outputText value="#{elHandler.array[0]}" />
7   <h:outputText value="#{elHandler.map['zwei']}" />
8   <h:outputText value="#{elHandler.map[elHandler.array[2]]}" />
9 </h:panelGrid>
```

Zunächst müssen wir aber noch den Handler vorstellen. Die Seite verwendet einen einfachen Handler, der in Listing 4.1 in Teilen dargestellt ist. Die Managed Bean `elHandler` der JSF-Seite ist eine Instanz dieser Klasse.

Listing 4.1: Handler für EL-Ausdrücke

```
public class ELHandler {

    private String name = "Übungen mit der Expression-Language";
    private Integer jahr = new Integer(
        new java.text.SimpleDateFormat("yyyy")
            .format(new java.util.Date()));

    private String[] array =
        new String[]{ "eins", "zwei", "drei" };
    private Map<String, String> map =
        new HashMap<String, String>();

    public ELHandler() {
        super();
        map.put("eins", "Erster Map-Eintrag");
        map.put("zwei", "Zweiter Map-Eintrag");
        map.put("drei", "Dritter Map-Eintrag");
    }
}
```

```
}
```

```
// ab hier nur einfache Getter und Setter
```

Im Beispiel verwenden wir nur Ausgabekomponenten, die Werteausdrücke werden daher ausschließlich lesend interpretiert. Bei der Verwendung einer Eingabekomponente könnten sie auch zum Setzen eines Properties verwendet werden. Der EL-Ausdruck des ersten Beispiels in Zeile 3 evaluiert zum Wert des Getters `getName()`, in diesem Fall also „Übungen mit der Expression-Language“. Dafür wird zunächst der Wert des Teilausdrucks links vom ersten Punkt bestimmt, in diesem Fall `elHandler`. Dies ist der Name einer Managed Bean, einer Instanz der Klasse `ELHandler`. Das Property `name` wird in den schon erwähnten Getter überführt und der Wert durch Aufruf des Getters bestimmt.

Für den Zugriff auf Arrays, Listen oder Maps übernimmt die Expression-Language die Java-Array-Notation der eckigen Klammern. Properties können alternativ auch mit der Klammernotation verwendet werden, so dass die Zeilen 3 und 4 semantisch äquivalent sind. In Zeile 5 wird eine String-Konstante mit einem Werteausdruck verbunden. Eine String-Konstante ohne das #-Zeichen wird wie üblich als Literal oder Literalausdruck bezeichnet.

Der Zugriff auf Elemente eines Arrays erfolgt in der Java-üblichen Notation, wie in Zeile 6 dargestellt. Auch Listen werden nach diesem Schema indiziert. Beim Zugriff auf Maps muss der Schlüssel als Konstante in eckigen Klammern geschrieben werden. Die Expression-Language lässt sowohl den Apostroph als auch das Anführungszeichen zur Kennzeichnung von String-Konstanten zu. Da wir als JSF-Implementierung aber Facelets verwenden und diese der XML-Syntax zu gehorchen haben, müssen Tag-Attribute und EL-Ausdrücke verschiedene Quotierungen verwenden. Wie von Ausdrücken nicht anders zu erwarten, können diese beliebig geschachtelt werden. Zeile 8 ist ein Beispiel hierfür.

Die Darstellung der JSF-Seite im Browser zeigt Abbildung 4.2 auf der nächsten Seite. Allerdings haben wir im Augenblick nur die ersten Zeilen der Seite durchgesprochen. Die weiteren folgen später.

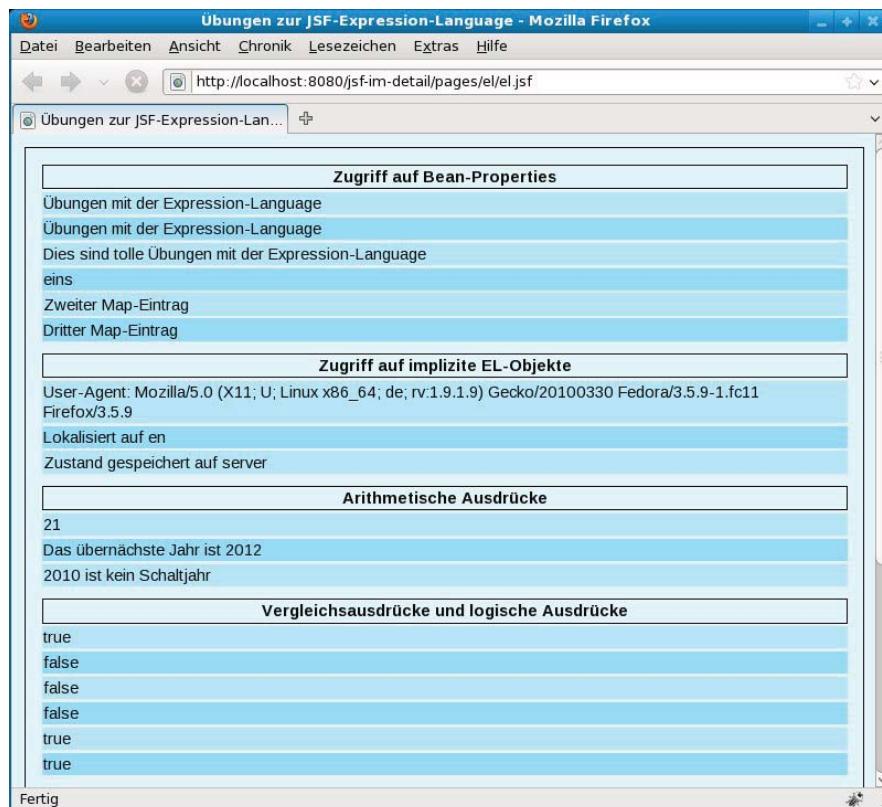


Abbildung 4.2: Beispiele für EL-Ausdrücke



Wir haben in diesem Abschnitt und auch bei der Vorstellung des Bearbeitungsmodells einer JSF-Anfrage immer von Gettern und Settern mit der Syntax `getXxx()` und `setXxx()` gesprochen. Die JavaBean-Spezifikation sieht für boolesche Properties zusätzlich die Möglichkeit eines Getters in der Form `isXxx()` vor. JSF unterstützt diese Form ebenfalls.

4.2.3 Vordefinierte Variablen

JSF definiert einige Objekte implizit, die in der Expression-Language verwendet werden können. Diese vordefinierten Variablen bezeichnen also verschiedene Objekte der zugrunde liegenden Servlet- und JSF-Implementierung. So ist es etwa möglich, auf den HTTP-Request-Header über einen vordefinierten Namen zuzugreifen. Tabelle 4.2 zeigt diese vordefinierten Variablen und erläutert sie kurz.

Tabelle 4.2: Vordefinierte Variablen der Expression-Language

Variablenname	Beschreibung
header	Eine Map von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist <i>ein</i> String.
headerValues	Eine Map von Request-Header-Werten. Schlüssel ist der Header-Name, Rückgabewert ist ein Array von Strings.
cookie	Eine Map von Cookies (Klasse Cookie im Package <code>javax.servlet.http</code>). Schlüssel ist der Cookie-Name.
initParam	Eine Map von Initialisierungsparametern der Anwendung (Application-Scope). Diese werden im Deployment-Deskriptor definiert.
param	Eine Map von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist <i>ein</i> String.
paramValues	Eine Map von Anfrageparametern. Schlüssel ist der Parametername. Rückgabewert ist ein Array von Strings.
facesContext	Die <code>FacesContext</code> -Instanz der aktuellen Anfrage.
component	Die im Augenblick bearbeitete Komponente.
cc	Die im Augenblick bearbeitete zusammengesetzte Komponente.
resource	Eine Map von Ressourcen.
flash	Eine Map von temporären Objekten für die nächste View.
view	Das View-Objekt (View-Scope).
viewScope	Eine Map von Variablen mit View-Scope.
request	Das Request-Objekt (Request-Scope).
requestScope	Eine Map von Variablen mit Request-Scope.
session	Das Session-Objekt (Session-Scope).
sessionScope	Eine Map von Variablen mit Session-Scope.
application	Das Application-Objekt (Application-Scope).
applicationScope	Eine Map von Variablen mit Application-Scope.

Als Beispiel zur Verwendung vordefinierter Variablen dient das folgende Code-Fragment (Fortsetzung der JSF-Seite `el.xhtml`):

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">
3     Zugriff auf implizite EL-Objekte
4   </f:facet>
5   <h:outputText value="User-Agent: #{header['User-Agent']}"/>
6   <h:outputText value="Lokalisiert auf #{view.locale}"/>
7   <h:outputText value="Zustand gespeichert auf \\"/>

```

```
8      #{initParam['javax.faces.STATE_SAVING_METHOD']}> />
9  </h:panelGrid>
```

In Zeile 5 wird der User-Agent des Request-Headers ausgegeben. Jede HTTP-Anfrage enthält den Client, der die Anfrage gestellt hat. In Abbildung 4.2 ist dies ein Mozilla unter Linux. In Zeile 6 wird die Lokalisierung der View erfragt. Im Beispiel ist dies eine deutsche Lokalisierung. In den Zeilen 7/8 wird schließlich auf einen Initialisierungsparameter der Anwendung zugegriffen. Eine Servlet-Anwendung wird durch den Deployment-Deskriptor `web.xml` konfiguriert. Im Beispiel enthält diese Datei die Zeilen

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>
```

die die Speicherung des Zustands auf dem Server definieren. Der Parameterwert `server` ist im Beispiel der Wert des EL-Ausdrucks. Auf Fragen der Konfiguration gehen wir in Abschnitt 4.8 ein. Die in Tabelle 4.2 genannten Scopes führen wir in Abschnitt 4.3 ein.

Wir beschließen den Abschnitt mit einem Beispiel für einen weiteren Zugriff auf Konfigurationsinformationen. Der Name einer Servlet-Anwendung wird zum Deployment-Zeitpunkt festgelegt und ist nicht durch JSF definiert. Über die vordefinierte Variable `request` kann jedoch der Kontextpfad ermittelt werden:

```
<head>
  <title>Übungen zur JSF-Expression-Language</title>
  <link rel="stylesheet" type="text/css"
    href="#{request.contextPath}/css/style.css" />
</head>
```

In diesem Beispiel wird über den Kontextpfad eine Style-Sheet-Datei eingebunden. Dies hätte man alternativ auch kontextrelativ realisieren können, und dient lediglich als Beispiel. Seit Version 2.0 gibt es eine weitere Alternative, Ressource-Dateien einzubinden, auf die wir im Detail in Abschnitt 4.10 eingehen.

4.2.4 Vergleiche, arithmetische und logische Ausdrücke

Die Expression-Language umfasst ein vollständiges Repertoire an Vergleichsausdrücken sowie arithmetischen und logischen Ausdrücken, so dass ein Rückgriff auf Java in der Regel nicht notwendig ist. Tabelle 4.3 führt die Operatoren der Expression-Language auf.

Tabelle 4.3: Operatoren der JSF-Expression-Language

Operator	Alternative	Beschreibung
.		Zugriff auf ein Property, eine Methode oder einen Map-Eintrag
[]		Zugriff auf ein Array- oder Listen-Element oder einen Map-Eintrag
()		Klammerung für Teilausdrücke
?:		Bedingter Ausdruck: <expr> ? <true-value> : <false-value>
+		Addition
-		Subtraktion oder negative Zahl
*		Multiplikation
/	div	Division
%	mod	Modulo
==	eq	gleich
!=	ne	ungleich
<	lt	kleiner
>	gt	größer
<=	le	kleiner-gleich
>=	ge	größer-gleich
&&	and	logisches UND
	or	logisches ODER
!	not	logische Negation
empty		Test auf null, einen leeren String, oder Test auf Array, Map oder Collection ohne Elemente

Beispiele mit Vergleichen sowie arithmetischen und logischen Ausdrücken sind etwa (Fortsetzung der JSF-Seite el.xhtml):

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">Arithmetische Ausdrücke</f:facet>
3   <h:outputText value="#{17 + 4}" />
4   <h:outputText
5     value="Das übernächste Jahr ist #{elHandler.jahr + 2}" />
6   <h:outputText
7     value="#{elHandler.jahr} ist #{{(elHandler.jahr % 4) == 0 \
8       ? 'ein' : 'kein'}} Schaltjahr" />
9 </h:panelGrid>
10 <h:panelGrid columns="1" rowClasses="odd,even">
```

```
12 <f:facet name="header">
13     Vergleichsausdrücke, arithmetische und logische Ausdrücke
14 </f:facet>
15 <h:outputText value="#{'eins' == elHandler.array[0]}" />
16 <h:outputText value="#{2009 == elHandler.jahr}" />
17 <h:outputText value="#{'2009' == elHandler.jahr}" />
18 <h:outputText value="#{2008 == elHandler.jahr}" />
19 <h:outputText value="#{elHandler.jahr > 2000}" />
20 <h:outputText value="#{elHandler.jahr > 2000 \
21                         and (elHandler.jahr %4 != 0)}" />
22 </h:panelGrid>
```

In Zeile 3 werden zwei Konstanten addiert. In Zeile 5 ist ein Summand ein Werteausdruck. Die Zeilen 7/8 sind ein Beispiel für den bedingten Ausdruck. Dieser ternäre Ausdruck hat dieselbe Semantik wie in Java: ergibt die Auswertung des ersten Teilausdrucks **true**, so ist der zweite Teilausdruck der Wert des Gesamtausdrucks, sonst der dritte. Bitte beachten Sie, dass der String-Wert des Attributs **value** in einer Zeile stehen muss. Der Zeilenumbruch am Ende von Zeile 7 ist der Drucktechnik geschuldet und würde zu einem Fehler führen. Auch der Algorithmus zur Schaltjahrbestimmung ist nicht ganz korrekt.

In Zeile 15 beginnen die Beispiele für Vergleiche mit einem String-Vergleich. Der Vergleich in Zeile 16 enthält Integer als Operanden. In Zeile 17 wird ein String mit einem Integer verglichen, was zu keinem Fehler führt. Die EL-Spezifikation definiert, welche impliziten Typkonvertierungen vor der Durchführung des Vergleichs zu erfolgen haben. Da die Konvertierungen relativ intuitiv sind, verzichten wir auf eine Darstellung und verweisen den interessierten Leser auf die EL-Spezifikation. Im Falle der Zeile 17 wird der Integer in einen String konvertiert und dann der Vergleich durchgeführt.

Wir haben in den Beispielen bereits verschiedene Literale wie z. B. Strings und Zahlen verwendet. Strings werden entweder in Anführungszeichen oder Apostrophe eingeschlossen, Zahlen als einfache Zahlen geschrieben, gebrochene Zahlen mit Dezimalpunkt. Die booleschen Literale werden als **true** und **false** geschrieben. Das Schlüsselwort **null** repräsentiert einen „nicht vorhandenen Wert“.

Wie alle Sprachen besitzt auch die Expression-Language reservierte Wörter, die nicht in Ausdrücken verwendet werden dürfen. Neben den in der Tabelle 4.3 in der Spalte *Alternative* genannten Wörtern sind dies die schon erwähnten Wörter **empty**, **true**, **false** und **null**. Weiterhin ist die Verwendung von **instanceof** nicht erlaubt, obwohl es (noch) keine Verwendung in der Expression-Language findet.

4.2.5 Methodenaufrufe und Methodenparameter

Die Unified Expression-Language als Teil von JSP 2.1 erhielt im Rahmen von Java-EE 6 das zweite Maintenance-Review [URL-JSR245II]. In diesem Maintenance-Review wird die Syntax von EL-Ausdrücken um die Möglichkeit von Methodenaufrufen mit Parametern erweitert. Da JSF 2.0 lediglich die Unified EL von JSP 2.1, nicht aber explizit das zweite Maintenance-Review voraussetzt, beschreiben wir diese Erweiterung in einem eigenen Abschnitt. Bitte vergewissern Sie sich, ob Ihre JSF-Implementierung und Ihr Container das zweite Maintenance-Review unterstützen.

Methodenaufrufe in Werteausdrücken waren bisher lediglich in Form von Getter- und Setter-Aufrufen erlaubt. Mit den Erweiterungen im zweiten Maintenance-Review sind Methodenaufrufe und die Verwendung von Methodenparametern erlaubt. Zur Konstruktion eines Beispiels definieren wir drei einfache Methoden in der Bean `elHandler`:

```
public String methodWithOneParam(String param) {  
    return param + " " + param;  
}  
  
public String methodWithTwoParams(String param1, int param2) {  
    return param1 + " " + param2;  
}  
  
public List<Integer> getList() { ... }
```

Die beiden ersten Methoden können nun mit Parametern in der üblichen Syntax versehen werden. Die dritte Methode dient zur Demonstration des Aufrufs der `Collection.size()`-Methode, die mit den bisherigen EL-Mitteln nicht aufgerufen werden konnte, da sie nicht dem Getter-Schema (`getSize()`) entspricht. Die Darstellung im Browser zeigt Abbildung 4.3.

```
<h:outputText  
    value="#{elHandler.methodWithOneParam('text')}" />  
<h:outputText  
    value="#{elHandler.methodWithTwoParams('text', 127)}" />  
<h:outputText value="#{elHandler.list.size()}" />
```

Analog zu Werteausdrücken können auch Methodenausdrücke parametrisiert werden:

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

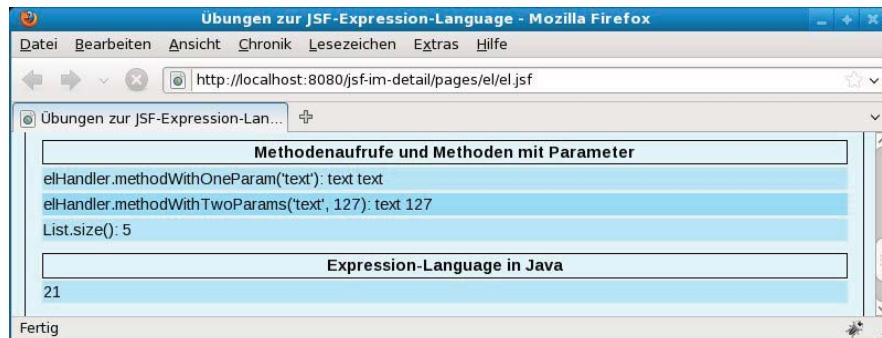


Abbildung 4.3: Weitere Beispiele für EL-Ausdrücke

4.2.6 Verwendung der Expression-Language in Java

Ziel der Expression-Language ist es, komplexe Ausdrücke direkt in JSF-Seiten zu formulieren, ohne auf Java zurückgreifen zu müssen. Damit erscheint die Verwendung der Expression-Language innerhalb von Java wenig sinnvoll. Es finden sich allerdings auch sinnvolle Einsatzgebiete, etwa der Zugriff auf eine Managed Bean, wie wir ihn in Abschnitt 4.3 kennenlernen werden.

Der folgende Ausschnitt der Seite `el.xhtml` zeigt den Quell-Code für die letzte Zeile der Browser-Darstellung in Abbildung 4.3:

```
1 <h:panelGrid columns="1" rowClasses="odd,even">
2   <f:facet name="header">Expression-Language in Java</f:facet>
3   <h:outputText value="#{elHandler.testAusdruck}" />
4 </h:panelGrid>
```

Der Werteausdruck in Zeile 3 dient als Grundlage zur Realisierung der Auswertung eines EL-Ausdrucks in Java. Bei der Auswertung wird die Methode `getTestAusdruck()` aufgerufen, auf die wir nun eingehen. Um einen Werteausdruck in Java zu erzeugen und auszuwerten, wird auf das `FacesContext`- und das `Application`-Objekt zurückgegriffen. Für das `Application`-Objekt kann mit der Methode `getExpressionFactory()` die `ExpressionFactory`-Instanz der Anwendung erfragt werden. Über diese Fabrik lässt sich bezüglich des aktuellen Kontextes ein Werteausdruck erzeugen und auswerten, wie der folgende Java-Code demonstriert:

```
public Integer getTestAusdruck() {
    FacesContext faces = FacesContext.getCurrentInstance();
    Application app = faces.getApplication();
    ExpressionFactory expressionFactory =
        app.getExpressionFactory();
    ELContext el = faces.getELContext();
    Integer val = (Integer) expressionFactory
```

```
    .createValueExpression(el, "#{17 + 4}", Integer.class)
    .getValue(el);
return val;
}
```

Die Klasse `ExpressionFactory` wurde mit der Unified Expression-Language eingeführt. Das dargestellte Verfahren ist daher seit JSF 1.2 zu verwenden. Die vor der Version 1.2 zu verwendende Methode `createValueBinding()` der `Application`-Klasse zur Erzeugung eines Werteausdrucks ist deprecated und sollte nicht mehr verwendet werden. Weiterhin sind alle Klassen des Package `javax.faces.el`, also Klassen, die die JSF-EL bis Version 1.1 realisiert haben und nun durch die Unified Expression-Language realisiert werden, deprecated. Im Beispiel wurde aus Gründen der Einfachheit der konstante Ausdruck `17 + 4` verwendet. Alle bisher eingeführten Konstrukte der Expression-Language sowie die impliziten Objekte aus Tabelle 4.2 können selbstverständlich ebenfalls verwendet werden.

Aufgabe 4.1

Laden Sie das Expression-Language-Projekt von der Web-Site des Buches herunter, und installieren Sie es. Erstellen Sie Ausdrücke für vom Client

- akzeptierte Sprachen
- akzeptierte Übertragungs-Codierungen
- akzeptierte Zeichensatz-Codierungen



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für die Expression-Language sind im Projekt *jsf-im-detail* enthalten.

4.3 Managed Beans

Managed Beans sind einfache Java-Beans oder POJOs, die Daten von UI-Komponenten sammeln, Event-Listener-Methoden implementieren und ähnliche Unterstützungsauflagen für UI-Komponenten wahrnehmen können. Des Weiteren ist es möglich, dass sie Referenzen auf UI-Komponenten beinhalten. Wir haben Managed Beans schon mehrfach in EL-Ausdrücken verwendet. Die Integration in die Expression-Language machen Managed Beans zu dem Werkzeug, mit dem Seiten-Entwickler ohne Java-Kenntnisse auf Java zurückgreifen können.

JavaServer Faces verwalten Managed Beans automatisch, was ihnen ihren Namen gab. Sie werden bei Bedarf erzeugt und existieren entsprechend ihrer deklarierten Lebensdauer. Die Verwaltung von Beans durch den Container – sowohl was den Lebenszyklus von Beans als auch die Verbindungen von Beans untereinander angeht – wurde in den letzten Jahren unter der Bezeichnung *Inversion of Control (IoC)* populär. Inversion of Control wird auch als *Dependency Injection* bezeichnet, weil POJOs auf deklarative Art und Weise miteinander verbunden werden können. Java-EE 5 definiert eine Reihe von Annotationen, mit denen bestimmte Objekte in andere Objekte injiziert werden können, wobei Managed Beans für uns besonders interessant sind. Wir gehen darauf in Abschnitt 4.3.5 ein.

Das Einsatzgebiet von Managed Beans, insbesondere die Aufgabe der Reaktion auf Benutzeraktionen, haben uns zu einem einheitlichen Namensschema veranlasst. Wir kennzeichnen Managed Beans dadurch, dass wir als letzten Namensbestandteil **Handler** verwenden. Beim Tic-Tac-Toe und der Comedian-Anwendung in den Kapiteln 2 und 3 also etwa die Klassen **TicTacToeHandler** und **ComedianHandler**.

4.3.1 Architekturfragen

Nach dem MVC-Design-Pattern ist eine Managed Bean ein Controller, genauer gesagt, *ein Teil* des Controllers, da viele Managed Beans zusammen und Teile des JSF-Laufzeitsystems die Verbindung zwischen View und Modell verwalten. Es liegt allerdings im Ermessen des Entwicklers, zwischen welchen Views und Modelldaten er die Managed Beans anordnet. Eine für die Stammdatenverwaltung sehr einfache Architektur ist die Verbindung einer Managed Bean zu einer View und der direkte Zugriff der Managed Bean auf die Datenbank. Diese Alternative ist in der Abbildung 4.4 links dargestellt. Das ist auch die Art und Weise, in der Entwicklungswerzeuge wie Visual-Studio oder Delphi die sehr schnelle Konstruktion von Oberflächen und deren Verbindung zum Datenbestand eines Datenbanksystems unterstützen. Für kleine und einfache Anwendungen ist dieser Ansatz praktikabel. Er skaliert jedoch nicht und wird bei komplexen Unternehmensanwendungen mit in der Regel sehr langen Lebenszeiten zu unwartbaren Systemen führen.

In Kapitel 3 haben wir eine sehr einfache „Architektur“ für unser Beispiel einer Comedian-Anwendung mit JavaServer Faces definiert. Diese ist kompatibel mit der in Abbildung 4.4 rechts dargestellten Architektur. Managed Beans greifen nicht direkt auf Datenbanken zu, sondern über eine Schicht von Geschäftsobjekten, die im Beispiel mit Hilfe von JPA-Entities realisiert waren. Die Zuordnung von Managed Beans zu Views muss nicht eins zu eins

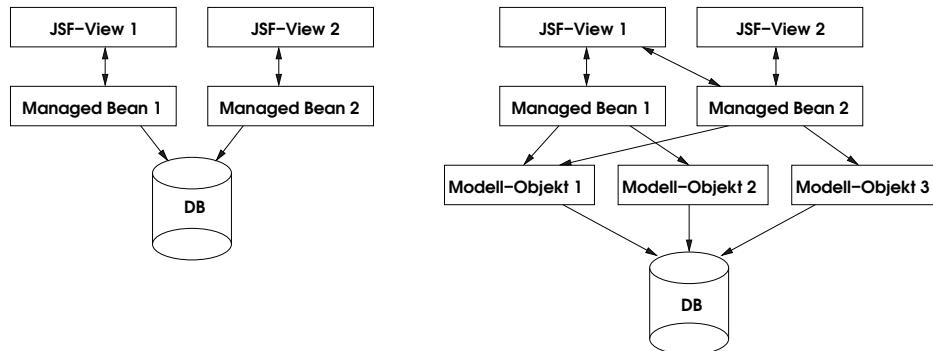


Abbildung 4.4: Einordnung Managed Beans

erfolgen, sondern kann beliebig sein. Auch die verwendeten Geschäftsobjekte sind frei wählbar. In der Comedian-Anwendung wurde die Managed Bean `ComedianHandler` in der Übersichts- und in der Detail-Seite für Comedians verwendet. Die Erfahrung zeigt jedoch, dass eine wirklich beliebige und damit unsystematische Zuordnung von Managed Beans zu JSF-Seiten zu Wartungsproblemen führt. Für größere Anwendungen wird für den Regelfall eine Managed Bean pro JSF-Seite empfohlen, wobei die Namensgebung der Managed Bean und der JSF-Seite sinnvollerweise diese Beziehung widerspiegelt. Ein weiterer Aspekt, der insbesondere das Speichermanagement und damit auch die Performance von JSF beeinflusst, ist die Lebensdauer einer Managed Bean, auf die wir nun näher eingehen.

4.3.2 Automatische Verwaltung von Managed Beans

Managed Beans werden vom JSF-Laufzeitsystem erzeugt und nach Gebrauch wieder gelöscht. Die Konfiguration von Managed Beans erfolgt in einer Konfigurationsdatei oder per Annotation. Die Konfigurationsdatei ist im Standardfall die Datei `faces-config.xml` im WEB-INF-Verzeichnis der Applikation. Wir erläutern im Abschnitt 4.8.2 den allgemeinen Aufbau dieser Konfigurationsdatei und die Verwendung alternativer Dateinamen. Wenn wir im Folgenden von der *JSF-Konfigurationsdatei* sprechen, ist damit die Datei `faces-config.xml` bzw. deren Alternativen gemeint.

Durch die automatische Verwaltung der Managed Beans in der JSF-Konfigurationsdatei können/kann

- die Beans an einer zentralen Stelle deklariert,
- die Beans an einer zentralen Stelle initialisiert,
- die Lebensdauer einer Bean deklarativ beschrieben,

- die Konfiguration von Beans konfigurativ und nicht programmatisch erfolgen und
- Wertebindungen zur Property-Initialisierung verwendet werden.

Seit JSF 2.0 ist die Deklaration von Managed Beans auch über Annotations möglich, was wir in der Comedian-Anwendung bereits eingesetzt haben. Man verliert dabei allerdings den Vorteil der Konzentration von Konfigurationen auf eine Stelle, also die beiden ersten Punkte der obigen Aufzählung. Wir beschreiben zunächst die Konfiguration über XML und führen dann die Annotationen in Abschnitt 4.3.6 ein. Für den JSF-Anfänger empfiehlt sich vor den ersten praktischen Programmierbeispielen auf jeden Fall die Lektüre des Abschnitts 4.3.6, da die Verwendung von Annotationen einfacher als die Verwendung der entsprechenden XML-Konstrukte ist.

Die Deklaration einer Managed Bean erfolgt im <managed-bean>-Element. An dieser Stelle wollen wir nicht noch einmal das XML-Schema darstellen. Moderne IDEs sollten eine Syntaxunterstützung bzgl. des Schemas bzw. entsprechende Wizards bereitstellen. Statt des Schemas versuchen wir in verständlicher Art und Weise die Darstellung der jeweiligen Ausschnitte. Beim <managed-bean>-Element stellt sich dies so dar:

```
<managed-bean>
  <description>*
  <icon>*
  <display-name>*
  <managed-bean-name>
  <managed-bean-class>
  <managed-bean-scope>
  <managed-bean-extension>
    (   <managed-property>* | 
        <map-entries> |
        <list-entries> )
```

Das *-Zeichen steht für die Wiederholung, das |-Zeichen für eine Alternative. Die Deklaration einer Managed Bean enthält also zunächst drei optionale und wiederholbare Elemente. Das <description>-Element sollte eine textuelle Beschreibung der Bean sein. <display-name> und <icon> werden in unseren Beispielen nicht benutzt. Sie können in IDEs zur Visualisierung verwendet werden. Die Elemente <managed-bean-name> sowie <managed-bean-class> und <managed-bean-scope> müssen vorhanden sein.

Beans, die derart deklariert wurden, erzeugt das JSF-Laufzeitsystem automatisch beim ersten Zugriff. Dazu wird der Default-Konstruktor, d. h. der Konstruktor ohne Parameter, aufgerufen. Je nach deklariertem Scope wird

die Bean mit dem deklarierten Namen unter keinem Scope oder dem Scope *Request*, *View*, *Session* oder *Application* gespeichert. Die zu verwendenden Elementinhalte des Tags `<managed-bean-scope>` sind entsprechend `none`, `request`, `view`, `session` und `application`. Im ersten Fall ist die Bean keinem Scope zugeordnet und lässt sich nur in der Konfigurationsdatei bei Initialisierungen anderer Beans verwenden. Wir werden dazu in Abschnitt 4.3.3 ein Beispiel entwickeln. Bei den weiteren Fällen wird die Bean entweder für einen Request, eine View oder eine Session erzeugt und ist dann für den Request, die View oder die Session verfügbar. Beim letzten Fall, dem Application-Scope, wird die Bean nur einmal erzeugt und ist quasi als globale Variable für die ganze Anwendung verfügbar. Wird beispielsweise als Name `myBean` und als Scope `session` deklariert, so kann über den EL-Ausdruck `"#{myBean.myProp}"` auf das Property `myProp` dieser Bean zugegriffen werden. Dies ist eine Kurzform für `"#{sessionScope.myBean.myProp}"`, wobei `sessionScope` eine vordefinierte Variable der JSF-Expression-Language ist (siehe Tabelle 4.2 auf Seite 48). Eine Ausnahme von der automatischen Erzeugung beim ersten Zugriff gilt für Managed Bean des Application-Scopes. Falls das Attribut `eager` des `<managed-bean>`-Elements auf `true` gesetzt ist, wird die Bean bereits beim Start der Anwendung und nicht beim ersten Zugriff erzeugt.

Mit JSF 2.0 sind auch sogenannte *Custom-Scopes* möglich. Diese werden durch einen EL-Ausdruck im `<managed-bean-scope>`-Element definiert. Der Ausdruck muss zu einer Map (`java.util.Map`) evaluieren, in der die Bean gespeichert wird. Die Map repräsentiert also den Scope. Wir gehen hierauf nicht weiter ein.

Die nächsten Elemente der Syntax werden für die Initialisierung von Properties bzw. von Beans, die Maps oder Listen sind, verwendet. Wir behandeln sie im nächsten Abschnitt.

Bemerkung: Der erfahrene Leser wird die Scopes Request, Session und Application aus der Servlet-Spezifikation wiedererkennen. Die Scopes None und View sind JSF-spezifisch und werden später ausführlich erläutert.

4.3.3 Initialisierung

Bei der Beschreibung eines Properties kann man zunächst optional eine Beschreibung, einen Bezeichner oder ein Piktogramm angeben, wie bei der Deklaration der Beans selbst. Diese können von IDEs zu Anzeigezwecken benutzt werden. Danach folgt der Name der Property und optional deren Klasse bzw. primitiver Typ. Strukturell ergibt sich folgende Syntax:

```
<managed-bean>
...
<managed-property>*
  <description>*
  <display-name>*
  <icon>*
  <property-name>
  <property-class>?
  (   <value> |
      <null-value> |
      <map-entries> |
      <list-entries>
  )
)
```

In der Regel erübrigt sich die Angabe der Klasse, weil das System den Typ des Properties über Reflection erhält und automatisch eine Typkonvertierung durchführt. Listing 4.2 zeigt eine Managed Bean, deren Properties **title** und **pi** initialisiert werden. Die Properties sind in der Klasse **MBHandler** deklariert als

```
private String title;
private Double pi;
```

Die Konvertierung des Wertes für **pi** von **String** nach **Double** wird automatisch vorgenommen.

Listing 4.2: Beispiel einer Managed Bean

```
<managed-bean>
  <description>
    Einfache Managed Bean
  </description>
  <managed-bean-name>mbHandler</managed-bean-name>
  <managed-bean-class>
    de.jsfpraxis.mb.MBHandler
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <description>
      Der Titel der Seite
    </description>
    <property-name>title</property-name>
    <value>Beispiel zu Managed Beans</value>
  </managed-property>
  <managed-property>
    <description>
      Eine Zahl zur Demonstration der automatischen Konvertierung
    </description>
  </managed-property>

```

```
</description>
<property-name>pi</property-name>
<value>3.1415</value>
</managed-property>
</managed-bean>
```

Die Initialisierung einer Property vom Typ *Array*, *List* oder *Map* ist etwas umfangreicher. Bei einem Array oder einer Liste werden statt der Angabe eines *<value>*-Elements beliebig viele *<value>*-Elemente innerhalb eines *<list-entries>* geschachtelt. Bei einer Map werden *<map-entry>*-Elemente, die jeweils ein *<key>*- und ein *<value>*-Element enthalten, in ein *<map-entries>*-Element eingeschlossen. Listing 4.3 zeigt ein Beispiel für eine Liste.

Listing 4.3: Beispiel einer Managed Bean

```
<managed-bean>
  <managed-bean-name> ... </managed-bean-name>
  <managed-bean-class> ... </managed-bean-class>
  <managed-bean-scope> ... </managed-bean-scope>
  <managed-property>
    <description>Wichtige URLs</description>
    <property-name>urls</property-name>
    <list-entries>
      <value>http://java.sun.com/docs/books/jls/index.html</value>
      <value>http://java.sun.com/j2ee/javaserverfaces</value>
      <value>http://java.sun.com/javase</value>
      <value>http://www.jsfpraxis.de</value>
    </list-entries>
  </managed-property>
  ...
</managed-bean>
```

Die entsprechende Property-Definition lautet:

```
private List urls = new ArrayList();
```

Die URLs können dann in einer JSF-Seite z.B. mit einem *<h:dataTable>* ganz einfach ausgegeben werden:

```
<h:dataTable var="url" value="#{mbHandler.urls}">
  <f:facet name="header">
    <h:outputText value="Wichtige URLs" />
  </f:facet>
  <h:column>
    <h:outputText value="#{url}" />
  </h:column>
</h:dataTable>
```

Bei einer Map ist für jeden Eintrag der Schlüssel und der Wert anzugeben. Diese werden daher noch einmal durch ein weiteres Element (<map-entry>) eingeschlossen. Da die Initialisierung einer Bean-Property und einer Bean vom Typ List oder Map bis auf die Deklaration der Beans selbst identisch sind, initialisieren wir im nächsten Beispiel eine ganze Map. Listing 4.4 zeigt eine Bean, die vom Typ java.util.Map ist, und initialisiert diese mit Schlüsseln und Werten.

Listing 4.4: Beispiel einer Managed Bean der Klasse Map

```
<managed-bean>
  <description>
    Eine Map als Managed Bean
  </description>
  <managed-bean-name>wichtigeUrls</managed-bean-name>
  <managed-bean-class>java.util.HashMap</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <map-entries>
    <map-entry>
      <key>JLS</key>
      <value>http://java.sun.com/docs/books/jls/index.html
    </value>
    </map-entry>
    <map-entry>
      <key>JSF Home</key>
      <value>http://java.sun.com/j2ee/javaserverfaces</value>
    </map-entry>
    <map-entry>
      <key>SDK</key>
      <value>http://java.sun.com/javase</value>
    </map-entry>
    <map-entry>
      <key>Das Buch</key>
      <value>http://www.jsfpraxis.de</value>
    </map-entry>
  </map-entries>
</managed-bean>
```

Eine solche Managed Bean kann man wie eine anwendungsdefinierte Managed Bean verwenden, z. B. als "#{wichtigeUrls}". Hier wird jedoch die Default-Implementierung der `toString`-Methode der Map-Implementierung aufgerufen, was zum Ergebnis

```
{Das Buch=http://www.jsfpraxis.de,
JLS=http://java.sun.com/docs/books/jls/index.html,
SDK=http://java.sun.com/javase,
```

```
JSF Home=http://java.sun.com/j2ee/javaserverfaces}
```

führt. Die Ausgabe erfolgt ohne Zeilenumbrüche, sodass die Einbettung in HTML das Layout bestimmt. Maps besitzen leider keine Getter, die eine sinnvolle Weiterverarbeitung mit JSF vor der Version 2.0 ermöglichen würden. Sind die Schlüssel jedoch bekannt, kann über die Schlüssel auf die Werte des Eintrags zugegriffen werden. So lässt sich z.B. eine Tabelle erzeugen:

```
<h:panelGrid columns="2">
    <h:outputText value="Java Language Specification" />
    <h:outputText value="#{wichtigeUrls['JLS']}" />
    <h:outputText value="JSF Home" />
    <h:outputText value="#{wichtigeUrls['JSF Home']}" />
    <h:outputText value="SDK" />
    <h:outputText value="#{wichtigeUrls['SDK']}" />
    <h:outputText value="JSF-Praxis - Das Buch" />
    <h:outputText value="#{wichtigeUrls['Das Buch']}" />
</h:panelGrid>
```

Mit der in Abschnitt 4.2.5 eingeführten Version der Expression-Language können beliebige Methodenaufrufe, nicht nur der Aufruf von Gettern und Settern erfolgen. So kann etwa explizit auf die Schlüssel und Werte der Map zugegriffen werden:

```
<h:panelGrid columns="2">
    Schlüssel:
    <h:outputText value="#{wichtigeUrls.keySet()}" />
    Werte:
    <h:outputText value="#{wichtigeUrls.values()}" />
</h:panelGrid>
```

Wenden wir uns abschließend den Scopes zu, die nicht von der Servlet-Spezifikation geerbt wurden. Der None-Scope ist ein besonderer Scope. Er ist der „Nichts-Scope“ d.h. er existiert nicht in dem Sinne, dass Managed Beans dieses Scopes über eine Map verwaltet werden und zugreifbar sind. Dies ist auch der Grund, warum es keine vordefinierte Variable `noneScope` analog etwa zum `sessionScope` gibt (siehe Tabelle 4.2 auf Seite 48). Der None-Scope wird ausschließlich innerhalb der Datei `faces-config.xml` verwendet, um Properties anderer Managed Beans zu initialisieren. Die Managed Bean `mb2Handler` mit der Deklaration

```
<managed-bean>
    <description>
        Bean mit Managed Bean-Property
    </description>
    <managed-bean-name>mb2Handler</managed-bean-name>
    <managed-bean-class>
```

```
de.jsfpraxis.mb.MB2Handler
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
<managed-property>
    <property-name>urls</property-name>
    <value>#{wichtigeUrls}</value>
</managed-property>
</managed-bean>
```

initialisiert ein Property, das in der Klassendefinition als

```
private Map urls;
```

definiert ist. Dieses Property wird mit der Managed Bean `wichtigeUrls` initialisiert. Die Bean `wichtigeUrls` hat im Beispiel einen Session-Scope, da es in einem anderen Beispiel direkt verwendet wird. Für die Initialisierungsaufgabe würde aber der None-Scope ausreichen.

Wenn eine Managed Bean über `<managed-property>` eine andere Managed Bean referenziert, kann es zu Konflikten bezüglich der Lebenszyklen der beiden Beans kommen. Eine Managed Bean darf nur eine Bean mit „größerem“ Scope referenzieren. Dabei sind die Scopes intuitiv als Request, View, Session und Application angeordnet. Der None-Scope bzw. eine Managed Bean dieses Scopes darf immer referenziert werden.

Der View-Scope ist in JSF 2.0 neu hinzugekommen. Er liegt als Scope zwischen dem Request- und dem Session-Scope. Bis zur Version 2.0 war es nicht möglich, zwischen verschiedenen Tabs einer Anwendung in einem Browser zu unterscheiden. Sollten Beans einzelne HTTP-Requests überleben, so mussten sie im Session-Scope leben. Die einzelnen Tabs einer Browser-Sitzung liegen aber in derselben Session, so dass auf JSF-Ebene keine Unterscheidung zwischen den Tabs möglich war.

Zur Demonstration des View-Scopes dienen uns zwei einfache Managed Beans, der `MBViewScopeHandler` und der `MBSessionScopeHandler`. Beide besitzen keine sinnvolle Anwendungslogik und überschreiben lediglich die `toString()`-Methode. Die Klasse `MBViewScopeHandler` definiert sich durch:

```
public class MBViewScopeHandler {

    public String toString() {
        return getClass().getSimpleName() + "@" + hashCode();
    }
}
```

Die Klasse `MBSessionScopeHandler` ist analog aufgebaut. Die einzige Unterscheidung der beiden Managed Beans ist deren Scope. Wie die Namensgebung vermuten lässt, ist der `MBViewScopeHandler` mit Scope `view` und der

MBSessionScopeHandler mit Scope session definiert. Der Unterschied zwischen beiden Beans wird im Browser deutlich, wenn dieselbe Seite in verschiedenen Tabs angezeigt wird. In der JSF-Seite `view-vs-session.xhtml` haben wir zusätzlich eine identisch aufgebaute Bean mit Request-Scope eingebaut:

```
<h:panelGrid columns="1" rowClasses="odd,even">
    <f:facet name="header">View-Scope und Session-Scope</f:facet>
    <h:outputText value="Request: #{requestScopeHandler}" />
    <h:outputText value="View: #{viewScopeHandler}" />
    <h:outputText value="Session: #{sessionScopeHandler}" />
    <h:commandButton value="OK" action="success"/>
</h:panelGrid>
```

Abbildung 4.5 zeigt diese Seite in zwei Tabs. Das Request-Scope-Objekt ist bei jedem neuen Request ein anderes. Das View-Scope-Objekt ist bei mehreren nacheinander erfolgenden Requests in einem Tab dasselbe, im anderen Tab jedoch ein anderes Objekt. Das Session-Scope-Objekt schließlich ist in beiden Tabs dasselbe. Es ist also mit JSF 2.0 wirklich möglich, in der Anwendungslogik zwischen Tabs zu unterscheiden.

4.3.4 Komponentenbindung

Mit den in Abschnitt 4.2.2 eingeführten Wertearausdrücken ist es möglich, Werte einer Komponente an eine Bean-Property zu binden. Eine besondere Art eines Wertearausdrucks ist die Komponentenbindung (engl. Component Binding), mit der sich eine UI-Komponente an eine Bean-Property binden lässt. In der Bean kann dann im Programm-Code lesend, aber auch schreibend auf die Komponente zugegriffen werden. Im `<h:dataTable>`-Element auf Seite 60 haben wir zur Erzeugung der Überschrift der Datentabelle

```
<h:outputText value="Wichtige URLs" />
```

verwendet. Mit dem Attribut `binding` lässt sich die Komponente an eine Bean-Property binden:

```
<h:outputText value="Wichtige URLs"
    binding="#{mbHandler.ueberschrift}" />
```

Die Definition der Property in der Bean-Klasse lautet:

```
private javax.faces.component.html.HtmlOutputText ueberschrift;
```

Diese Variable `ueberschrift` ist eine gewöhnliche Instanzvariable und lässt sich ganz normal verwenden; z.B. kann ihr Wert in einem Setter innerhalb einer Action-Methode gesetzt werden:

```
ueberschrift.setValue("Ganz wichtige URLs");
```

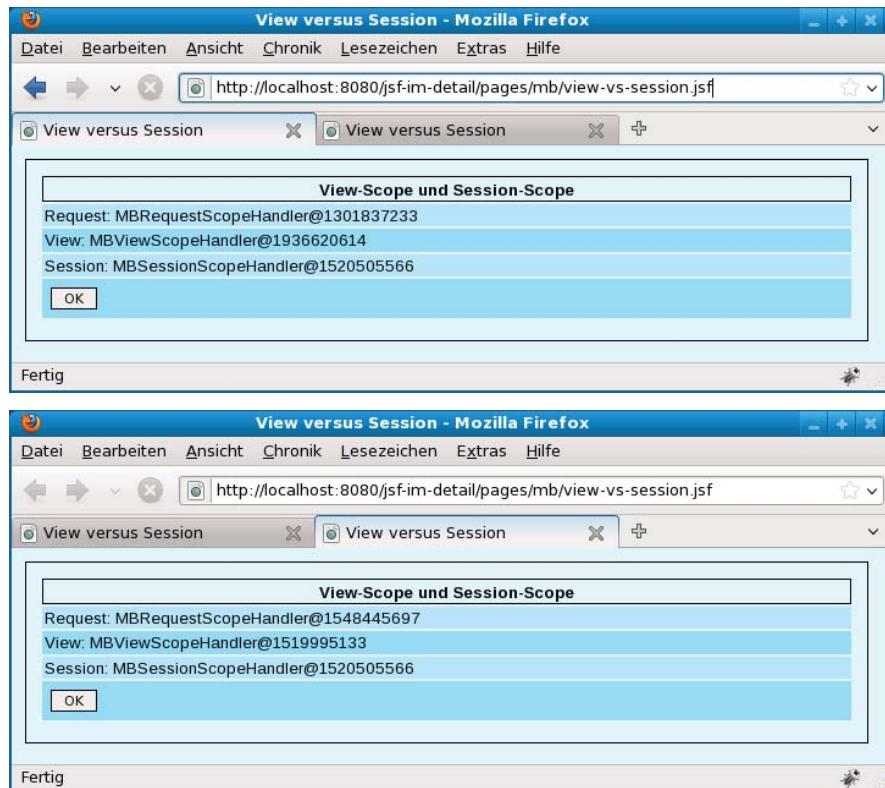


Abbildung 4.5: View- versus Session-Scope

Bei der erneuten Anzeige der Seite ist die Überschrift dann aktualisiert und enthält den neuen Text. Diese Funktionalität hätte man ganz offensichtlich auch durch ein String-Property und einen einfachen Werteausdruck realisieren können.

Als etwas anspruchsvolleres Beispiel soll nun eine dynamische Auswahlmöglichkeit realisiert werden. Hierzu dient die JSF-Seite `component-binding.xhtml`, deren interessanter Teil im Folgenden dargestellt ist.

```
1  <h:panelGrid columns="2" style="padding: 10px;">
2      <f:facet name="header">Löschen eines Select-Items</f:facet>
3      <h:selectOneMenu binding="#{componentBindingHandler.menu}">
4          <f:selectItem itemLabel="bitte auswählen"
5                      itemValue="bitte auswählen" />
6      </h:selectOneMenu>
7      <h:inputText value="#{componentBindingHandler.item}" />
8
9      <h:commandButton value="Löschen"
10         action="#{componentBindingHandler.delete}"/>
```

```
11      <h:commandButton value="Hinzufügen"  
12          action="#{componentBindingHandler.add}"/>  
13  </h:panelGrid>
```

Die initiale Darstellung im Browser ist in Abbildung 4.6 wiedergegeben. Nach einigem Löschen und Hinzufügen von Alternativen stellt Abbildung 4.7 eine weitere Momentaufnahme dar.



Abbildung 4.6: Beispiel zur Komponentenbindung (1)



Abbildung 4.7: Beispiel zur Komponentenbindung (2)

Man erkennt in den Zeilen 3 bis 6 das `<h:selectOneMenu>`-Tag, mit dem JSF eine Drop-Down-Auswahl realisiert. Als einzige Alternative existiert initial der Text "bitte auswählen". Durch Betätigen der Schaltfläche Hinzufügen wird der im Texteingabefeld vorhandene String als neue Alternative der Auswahl hinzugefügt. Durch Betätigen der Schaltfläche Löschen wird die im Drop-Down-Menü selektierte Alternative gelöscht.

Die Manipulation des Komponentenbaums erfolgt server-seitig in Java. Die Klasse `ComponentBindingHandler` mit ihren beiden Action-Methoden ist in Listing 4.5 dargestellt.

Listing 4.5: Beispiel für Komponentenbindung

```
1 public class ComponentBindingHandler {  
2  
3     private String item;  
4     private HtmlSelectOneMenu menu;  
5  
6     public String delete() {  
7         UIComponent found = null;  
8         List<UIComponent> children = menu.getChildren();  
9         for (UIComponent component : children) {  
10             UISelectItem item = (UISelectItem) component;  
11             if (menu.getValue().equals(item.getItemValue())) {  
12                 found = component;  
13             }  
14         }  
15         menu.getChildren().remove(found);  
16         return null;  
17     }  
18  
19     public String add() {  
20         UISelectItem si = new UISelectItem();  
21         si.setItemLabel(item);  
22         si.setItemValue("value");  
23         menu.getChildren().add(si);  
24         return null;  
25     }  
26     ...  
27 }
```

Die beiden Instanzvariablen `item` und `menu` in den Zeilen 3 und 4 werden in der JSF-Seite in den EL-Ausdrücken der Zeilen 7 und 3 verwendet. Die Methode `delete()` erfragt zunächst die Liste aller Kinder (Zeile 8). Über diese wird in einer Schleife iteriert und nach der selektierten Auswahl gesucht. Das Löschen (Zeile 15) der Auswahl muss außerhalb der Iteration erfolgen, so dass zum dargestellten Muster eines Zwischenspeicherns gegriffen wird. Die Methode `add()` ist sehr einfach aufgebaut, so dass wir die Analyse dem Leser überlassen.

Bemerkung: Falls Sie das Beispiel ausprobieren, wird Ihnen auffallen, dass die Auswahl "bitte auswählen" nicht löschar ist. Die Methode `delete()` löscht die Komponente zwar, durch das Vorhandensein des Select-Items in der JSF-Seite wird die Auswahl aber wieder hinzugefügt.

4.3.5 Java-EE-5-Annotationen

Mit der Version 5 von Java-EE nahmen Annotationen Einzug in die Spezifikation. Diese Annotationen sind von den mit JSF 2.0 eingeführten JSF-spezifischen Annotationen zu unterscheiden, die wir in Abschnitt 4.3.6 einführen werden. EE-Annotationen dienen vor allem der sogenannten *Resource-Injection* und sind unter anderem in der Servlet-Spezifikation 2.5 [URL-JSR154] definiert. Unter einer Ressource versteht man in diesem Zusammenhang z. B. Persistenzkontexte, JDBC-Datenquellen oder JMS-Nachrichtenziele. Neben diesen EE-spezifischen Annotationen wurde im JSR *Common Annotations* [URL-JSR250] ein gemeinsamer Satz von Annotationen definiert, die in anderen Spezifikationen Verwendung finden können. Diese Annotationen dokumentieren ihren allgemeinen Charakter durch ihr Package: `javax.annotation`. Wir haben Annotationen aus beiden Spezifikationen bereits in der Comedian-Anwendung in Abschnitt 3 kennengelernt. Tabelle 4.4 zeigt die Annotationen aus beiden Spezifikationen. Wir stellen die Paketnamen explizit dar, um die Zugehörigkeit zur jeweiligen Spezifikation klarzumachen.

Die in Tabelle 4.4 genannten Annotationen sind mit Ausnahme von `@PostConstruct` und `@PreDestroy` immer in einer Singular- und in einer Pluralform vorhanden. Entgegen der Darstellung in der Tabelle sind die Singularformen nicht nur zur Annotation auf Attribut- oder Methodenebene, sondern auch auf Klassenebene erlaubt. Die Pluralversionen fassen mehrere solcher Annotationen zusammen. Die `@EJB`-Annotation, also die Injektion einer Session-Bean, ist in Java-EE 5 von zentraler Bedeutung. Im Abschnitt 9.1 entwickeln wir ein Beispiel, das diese Annotation verwendet.

4.3.6 JSF-Annotationen

Wie bereits erwähnt, hielten mit JSF 2.0 Annotationen Einzug in die JSF-Spezifikation. Sie können als Alternative zur Beschreibung in der Konfigurationsdatei `faces-config.xml` dienen. Falls es zu Widersprüchen zwischen XML und Annotation kommt, erhält XML den Vorzug. Anders ausgedrückt: Die Konfigurationsdatei überschreibt Annotationen im Quell-Code, was im Sinne konfigurierbarer Systeme ohne Quell-Code-Änderung, -Compilation und Deployment sehr sinnvoll ist.

Wir stellen hier nur die Annotationen vor, die im Zusammenhang mit Managed Beans stehen. Es sind dies Annotationen, um Managed Beans zu deklarieren und Properties zu injizieren, sowie Annotationen, um den Scope einer Managed Bean zu definieren. Die weiteren Annotationen werden im jeweiligen Zusammenhang eingeführt.

Tabelle 4.4: Java-EE-Annotationen

Annotation	Beschreibung
Package javax.annotation	
@PostConstruct	Die annotierte Methode wird aufgerufen, nachdem alle Injektionen für die Instanz erfolgt sind und bevor die erste Anwendungsmethode der Instanz aufgerufen wird.
@PreDestroy	Die annotierte Methode wird aufgerufen, bevor das Objekt gelöscht wird.
@Resource	Injiziert eine allgemeine Ressource in das Attribut oder Methode.
@Resources	Fasst mehrere @Resource-Annotationen zusammen.
Package javax.ejb	
@EJB	Injiziert eine Session-Bean in das Attribut oder Methode.
@EJBs	Fasst mehrere @EJB-Annotationen auf Klassenebene zusammen.
Package javax.persistence	
@PersistenceContext	Injiziert einen Entity-Manager als Persistenzkontext.
@PersistenceContexts	Fasst mehrere @PersistenceContexts-Annotationen zusammen.
@PersistenceUnit	Injiziert eine Entity-Manager-Fabrik als Persistenzeinheit.
@PersistenceUnits	Fasst mehrere @PersistenceUnit-Annotationen zusammen.
Package javax.xml.ws	
@WebServiceRef	Injiziert eine Referenz auf einen Web-Service.
@WebServiceRefs	Fasst mehrere @WebServiceRef-Annotationen auf Klassenebene zusammen.

Listing 4.6: Die annotierte Klasse MBAnnotationHandler

```
@ManagedBean(name = "mbAnnotationHandler")
@SessionScoped
public class MBAnnotationHandler {

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

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

...
}
```

Listing 4.6 zeigt die mit `@ManagedBean` und `@SessionScoped` annotierte Klasse `MBAnnotationHandler`. Die `@ManagedBean`-Annotation erzeugt aus einem POJO eine Managed Bean analog zum `<managed-bean>`-Element der JSF-Konfigurationsdatei. Die `@SessionScoped`-Annotation bedingt, dass diese Bean im Session-Scope existiert und entspricht daher dem `<managed-bean-scope>`-Element mit Wert `session`. Die den anderen Scopes entsprechenden Annotationen sind `@NoneScoped`, `@RequestScoped`, `@ViewScoped` und `@ApplicationScoped`. Falls keine Scope-Annotation angegeben wird, findet der Default `@RequestScoped` Verwendung.

Während bei der XML-Konfiguration von Managed Beans der Name der Bean explizit angegeben werden muss, wird über die `@ManagedBean`-Annotation automatisch ein Name vergeben. Dieser entsteht aus dem Klassennamen durch Kleinschreibung des ersten Buchstabens, im Listing 4.6 also `mbAnnotationHandler`. Dieser kann, wie im Beispiel geschehen, durch das `name`-Attribut überschrieben werden.

Auf Ebene der Bean-Deklaration können somit Annotationen die XML-Konfiguration vollständig ersetzen. Auf der Ebene der Initialisierung von Bean-Properties ist dies nicht der Fall. Die Annotation `@ManagedProperty` injiziert den Wert des `value`-Attributs in das annotierte Property. Als Wert sind Strings, wie im Beispiel in Listing 4.6, oder EL-Wertearausdrücke erlaubt. Falls das Property nicht vom Typ String ist, erfolgt eine Konvertierung der String-Darstellung des Wertes in den Property-Typ. Dies gilt als Faustformel für alle einfachen Java-Datentypen, für die JSF Standardkonvertierer vorhält. Wir stellen diese in Abschnitt 4.4 ausführlich dar.

Falls das Property einen komplexeren Datentyp hat, kann keine Konvertierung mit den Standardkonvertierern erfolgen. Dieses Problem gibt uns die Gelegenheit, den oben erwähnten EL-Wertearausdruck als Wert der `@ManagedProperty`-Annotation in einem Beispiel zu verwenden. Wir erweitern die Klasse `MBAnnotationHandler` und stellen die Erweiterungen in Listing 4.7 dar.

Listing 4.7: Die erweiterte Klasse `MBAnnotationHandler`

```
@ManagedBean(name = "mbAnnotationHandler")
@SessionScoped
```

```
public class MBAnnotationHandler {  
  
    ...  
  
    @ManagedProperty(value = "#{urls.urls}")  
    private List<String> urls;  
  
    ...  
  
    @ManagedBean(name = "urls")  
    @NoneScoped  
    public static classUrls {  
  
        public List<String> getUrls() {  
            return Arrays.asList(new String[] {  
                "http://java.sun.com/docs/books/jls/index.html",  
                "http://java.sun.com/j2ee/javaserverfaces",  
                "http://java.sun.com/javase",  
                "http://www.jsfpraxis.de" });  
        }  
    }  
}
```

Man erkennt, dass die `@ManagedBean`-Annotation nicht nur für Top-Level-Klassen erlaubt ist und die `@ManagedProperty`-Annotation EL-Werteausdrücke als Werte erlaubt. Das Beispiel ist nicht sehr sinnvoll, da die Initialisierung des Propertys `urls` in Java sehr viel einfacher erfolgen könnte. Da der EL-Werteausdruck aber nicht eingeschränkt ist, können z.B. auch Managed Beans, die in der Datei `faces-config.xml` deklariert sind, verwendet werden, so dass sinnvollere Verwendungsmöglichkeiten vorstellbar sind. Wir sind allerdings der Meinung, dass die Möglichkeit der Initialisierung von Properties wegen Fragen der Konfigurierbarkeit ohne erneute Compilierung eher in die XML-Konfiguration als in Java-Quell-Code gehört. In der Praxis wird der dargestellte Code daher sicher nicht so häufig Verwendung finden.

Aufgabe 4.2

In der Comedian-Anwendung in Kapitel 3 werden fünf Comedians initial in die Datenbank geschrieben. Wenn man das Beispiel mit zwei Browsern gleichzeitig ausprobiert, sind diese fünf Comedians doppelt vorhanden, da die Initialisierungsmethode per Session aufgerufen wird. Überarbeiten Sie die Anwendung, so dass die Comedians nicht dupliziert werden.



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für Managed Beans sind im Projekt *jsf-im-detail* enthalten.

4.4 Validierung und Konvertierung

Benutzereingaben müssen in der Regel validiert werden. Prinzipiell lässt sich zwischen einer syntaktischen und einer semantischen Validierung unterscheiden. Im Fall der Comedian-Anwendung wurde eine syntaktische Validierung vorgenommen, nämlich der Test, ob überhaupt eine Eingabe vorhanden ist. Eine semantische Validierung ist etwa die Prüfung, ob das Geburtsdatum in der Vergangenheit liegt. Dies kann man auf mehrere Eingaben ausweiten, um etwa zu prüfen, ob das Von-Datum vor dem Bis-Datum bei der Eingabe eines Zeitraums liegt.

JSF stellt eine kleine Zahl von Validierungsmöglichkeiten bereit, die sowohl syntaktischer als auch in geringem Umfang semantischer Natur sind, z.B. die Länge von Texteingaben oder ein Zahlenbereich bei numerischen Eingaben. Diese Möglichkeiten reichen gewöhnlich nur für sehr einfache Anwendungen aus, so dass in der Regel eigene Validierer erstellt werden müssen.

Für viele Validierungen muss zunächst eine Konvertierung stattfinden, da HTML/HTTP String-basiert arbeitet, einige Validierer und Bean-Properties aber andere Typen erwarten. Es gibt für die Standarddatentypen eine Reihe von vordefinierten Konvertierern, die man bei Bedarf um eigene Konvertierer erweitern kann.

4.4.1 Standardkonvertierer

Bei der Darstellung des Bearbeitungsmodells einer JSF-Anfrage wurde die Validierungsphase in Abschnitt 4.4 beschrieben, auf die wir nun zurückgreifen. Falls eine Komponente an eine Bean-Property gebunden ist, deren Typ nicht String ist, besteht der erste Teil der Validierungsphase in einer Konvertierung in den Typ des Property. JavaServer Faces bringen eine Reihe von Konvertierern von Haus aus mit, z.B. für Zahlen und Kalenderdaten. Diese Konvertierer befinden sich im Package `javax.faces.convert`. Jeder Konvertierer implementiert das Interface `Converter` desselben Packages. Die beiden einzigen Methoden des Interface sind

```
public Object getAsObject(
    javax.faces.context.FacesContext context,
```

```
javax.faces.component.UIComponent component,
java.lang.String value);

public String getAsString(
    javax.faces.context.FacesContext context,
    javax.faces.component.UIComponent component,
    java.lang.Object value);
```

mit denen ein `String` in ein `Object` und umgekehrt konvertierbar ist.

Im JSF-Framework sind Konvertierer für die Typen `BigDecimal`, `BigInteger`, `Character`, `Boolean`, `Byte`, `Integer`, `Short`, `Long`, `Float`, `Double` und `Enum` vorhanden. Diese befinden sich, wie bereits erwähnt, im Package `javax.faces.convert`. Die Klassennamen leiten sich systematisch von den genannten Typen und dem Suffix „Converter“ ab, für den `Integer`-Konvertierer also der Klassenname `IntegerConverter`. Diese Konvertierer werden automatisch verwendet, wenn der Property-Typ einem dieser Typen entspricht. Neben den Wrapper-Typen sind auch die primitiven Typen `char`, `boolean`, `byte`, `int`, `short`, `long`, `float` und `double` möglich. In jeder der genannten Klassen ist eine String-Konstante `CONVERTER_ID` definiert, deren Wert sich aus dem Präfix „`javax.faces.`“ und dem Typ zusammensetzt, für den `Integer`-Konvertierer also das String-Literal `"javax.faces.Integer"`.

Die Verwendung der genannten Konvertierer geschieht automatisch immer dann, wenn durch eine Wertebindung eine Eingabekomponente an eine Bean-Property mit entsprechendem Typ gebunden wird. Der JSF-Code in Listing 4.8 erzeugt Eingabefelder und bindet diese an Properties, die die verschiedenen ganzzahligen Datentypen Javas besitzen.

Listing 4.8: Konvertierung ganzer Zahlen

```
<h:panelGrid columns="2">
    <h:outputLabel for="byteValue" value="Byte-Wert:" />
    <h:inputText id="byteValue" size="30"
        value="#{ganzeZahlenHandler.byteValue}" />
    <h:outputLabel for="shortValue" value="Short-Wert:" />
    <h:inputText id="shortValue" size="30"
        value="#{ganzeZahlenHandler.shortValue}" />
    <h:outputLabel for="intValue" value="Int-Wert:" />
    <h:inputText id="intValue" size="30"
        value="#{ganzeZahlenHandler.intValue}" />
    <h:outputLabel for="longValue" value="Long-Wert:" />
    <h:inputText id="longValue" size="30"
        value="#{ganzeZahlenHandler.longValue}" />
    <h:outputLabel for="bigIntValue" value="BigInteger-Wert:" />
    <h:inputText id="bigIntValue" size="30"
```

```
        value="#{ganzeZahlenHandler.bigIntValue}" />
</h:panelGrid>
```

Der folgende Code zeigt die Definition der Properties.

```
private Byte byteValue;
private Short shortValue;
private Integer intValue;
private Long longValue;
private BigInteger bigIntValue;
```

Für gebrochene Zahlen stehen ebenfalls Konvertierer bereit. Auch hier gilt die Aussage, dass monetäre oder andere Werte, die eine exakte Darstellung benötigen, nicht mit `Double` oder `Float`, sondern mit `BigDecimal` zu realisieren sind. Listing 4.9 zeigt einen Seitenausschnitt, der Ein- und Ausgaben mit diesen drei Datentypen realisiert. Dabei erfolgt eine dreispaltige Ausgabe mit Label, Ein-/Ausgabe und Ausgabe des Quadrats der Eingabe im jeweiligen Datentyp.

Listing 4.9: Konvertierung gebrochener Zahlen (JSF)

```
<h:panelGrid columns="3">
    <h:outputText value="Typ" style="font-weight: bold;" />
    <h:outputText value="Ein-/Ausgabe"
                  style="font-weight: bold;" />
    <h:outputText value="Quadrat der Eingabe"
                  style="font-weight: bold;" />

    <h:outputLabel for="floatValue" value="Float-Wert:" />
    <h:inputText id="floatValue"
                 value="#{bruecheHandler.floatValue}" />
    <h:outputText value="#{bruecheHandler.floatValueQuadrat}" />

    <h:outputLabel for="doubleValue" value="Double-Wert:" />
    <h:inputText id="doubleValue"
                 value="#{bruecheHandler.doubleValue}" />
    <h:outputText value="#{bruecheHandler.doubleValueQuadrat}" />

    <h:outputLabel for="bigDecimalValue"
                  value="BigDecimal-Wert:" />
    <h:inputText id="bigDecimalValue"
                 value="#{bruecheHandler.bigDecimalValue}" />
    <h:outputText
        value="#{bruecheHandler.bigDecimalValueQuadrat}" />

    <h:commandButton value="Abschicken"
```

```
action="#{bruecheHandler.abschicken}">
<h:panelGroup /><h:panelGroup />
</h:panelGrid>
```

Den Code der entsprechenden Managed Bean zeigt Listing 4.10. Für die drei Properties existieren entsprechende Getter und Setter, die jedoch aus Platzgründen nicht dargestellt sind. In drei nach dem Getter-Pattern erstellten Methoden wird jeweils das Quadrat der Eingabe berechnet.

Listing 4.10: Konvertierung gebrochener Zahlen (Java)

```
public class BruecheHandler {

    private Float floatValue;
    private Double doubleValue;
    private BigDecimal bigDecimalValue;

    public String abschicken() {
        return "success";
    }

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

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

    public BigDecimal getBigDecimalValueQuadrat() {
        if (bigDecimalValue == null)
            return new BigDecimal(0);
        return bigDecimalValue.multiply(bigDecimalValue);
    }

    // ab hier nur Getter und Setter
}
```

Abbildung 4.8 zeigt die Darstellung der JSF-Seite, deren Hauptteil wir in Listing 4.9 vorgestellt hatten. Das obere Browser-Fenster zeigt die Darstellung beim ersten Besuch der Seite, das untere nach der Eingabe des Wertes 0.1 in jedem der drei Eingabefelder und nach Betätigung der Schaltfläche. Der Bruch

0.1 ist binär nicht exakt darstellbar. Trotzdem wird er beim Eingabefeld nach Betätigen der Schaltfläche korrekt dargestellt. Dies ist jedoch in einer intelligenten Ausgabe der dafür verantwortlichen `println()`-Methode begründet. Nach der Multiplikation mit sich selbst ist die Darstellung bereits in einem Bereich, in dem die Ausgabe keine derartige Korrektur mehr vornimmt.

The figure consists of two screenshots of a Mozilla Firefox browser window. Both windows have the title "Validierer und Konvertierer: Brüche - Mozilla Firefox".
The top window shows the following data:

Validierer und Konvertierer: Brüche		
Typ	Ein-/Ausgabe	Quadrat der Eingabe
Float-Wert:	<input type="text"/>	0.0
Double-Wert:	<input type="text"/>	0.0
BigDecimal-Wert:	<input type="text"/>	0

A button labeled "Abschicken" is visible at the bottom left.
The bottom window shows the same fields after input:

Validierer und Konvertierer: Brüche		
Typ	Ein-/Ausgabe	Quadrat der Eingabe
Float-Wert:	<input type="text" value="0.1"/>	0.010000001
Double-Wert:	<input type="text" value="0.1"/>	0.01000000000000002
BigDecimal-Wert:	<input type="text" value="0.1"/>	0.01

A button labeled "Abschicken" is visible at the bottom left.

Abbildung 4.8: Darstellung von Listing 4.9

Aufgabe 4.3

Vergewissern Sie sich, dass die inexakte Darstellung von Brüchen keine Eigenschaft der JSF-Implementierung, sondern eine Eigenschaft der Sprache Java und der JVM ist. Genauer gesagt, liegt es sogar an der Definition der Binär-Repräsentation von Brüchen, die im IEEE-Standard 754-1985 festgelegt ist.

Diese ist Grundlage sehr vieler Programmiersprachen. Führen Sie den folgenden Programm-Code direkt in Java aus:

```
System.out.println(0.1);
System.out.println(0.1 * 0.1);
```

4.4.2 Konvertierung von Kalenderdaten und Zahlen

JavaServer Faces enthalten zwei weitere Standardkonvertierer, die Klassen `DateTimeConverter` und `NumberConverter`, beide ebenfalls im Package `javax.faces.convert`. Während die bisher vorgestellten Konvertierer keine Konfiguration erlauben, sind die Konfigurationsmöglichkeiten dieser beiden Konvertierer äußerst vielfältig. Wir gehen zunächst auf die Klasse `DateTimeConverter` ein. Der in Listing 4.11 dargestellte JSF-Code verwendet das Tag `<f:convertDateTime>` mit dem Attribut `type`.

Listing 4.11: Datums- und Zeitdarstellung in englischer Lokalisierung

```
<f:view locale="en">
    <h:panelGrid columns="2" rowClasses="odd,even">
        <f:facet name="header">DateTimeConverter (englisch)</f:facet>
        <h:outputText value="Ohne Attribute" />
        <h:outputText value="#{dtHandler.date}">
            </h:outputText>
            <h:outputText value="type="date"" />
            <h:outputText value="#{dtHandler.date}">
                <f:convertDateTime type="date" />
            </h:outputText>
            <h:outputText value="type="time"" />
            <h:outputText value="#{dtHandler.date}">
                <f:convertDateTime type="time" />
            </h:outputText>
            <h:outputText value="type="both"" />
            <h:outputText value="#{dtHandler.date}">
                <f:convertDateTime type="both" />
            </h:outputText>
    </h:panelGrid>
</f:view>
```

Im ersten Beispiel wird auf die Verwendung von `type` verzichtet. In den weiteren Beispielen werden die Werte `date`, `time` und `both` verwendet. Abbildung 4.9 zeigt die Darstellung im Browser.

Anzumerken ist noch, dass der EL-Ausdruck eine Instanz des Typs `java.util.Date` zurückliefert und in der ersten Zeile die Lokalisierung



Abbildung 4.9: Darstellung von Listing 4.11

der View auf *Englisch* eingestellt wird. Alternativ erlaubt das Tag `<f:convertDateTime>` selbst die Verwendung des Attributs `locale`.

Das Tag `<f:convertDateTime>` erlaubt neben den schon erwähnten Attributen `type` und `locale` die Verwendung der Attribute `dateStyle`, `timeStyle`, `timeZone` und `pattern`. Alle Attribute sind vom Typ `String` mit Ausnahme des `Locale`-Objekts für das `locale`-Attribut. Tabelle 4.5 zeigt die Attribute mit ihren möglichen Werten.

Tabelle 4.5: Attribute des `<f:convertDateTime>`-Tags

Attribut	Werte und Beschreibung
<code>type</code>	<code>date</code> (Default), <code>time</code> oder <code>both</code> . Anzeige von Datum, Zeit oder Datum und Zeit.
<code>dateStyle</code>	<code>short</code> , <code>medium</code> (Default), <code>long</code> und <code>full</code> . Formatangabe für den Datumteil, falls <code>type</code> gesetzt.
<code>timeStyle</code>	<code>short</code> , <code>medium</code> (Default), <code>long</code> und <code>full</code> . Formatangabe für den Zeitteil, falls <code>type</code> gesetzt.
<code>timeZone</code>	Angabe der Zeitzone. Falls nicht gesetzt, ist der Default Greenwich-Mean-Time (GMT).
<code>locale</code>	Lokalisierung, entweder als Instanz von <code>java.util.Locale</code> oder als String.
<code>pattern</code>	Angabe eines Patterns zur Formatierung. Alternative zu <code>type</code> . Details siehe Tabelle 4.6.

Die Abbildung 4.10 auf der nächsten Seite zeigt Beispiele zur Konfiguration des `DateTimeConverters`. Dabei ist die deutsche Lokalisierung für die View

eingestellt. Da wir in der JSF-Seite auch den formatierungsspezifischen Quell-Code angeben, verzichten wir auf eine explizite Darstellung der JSF-Seite.

Attribute	Darstellung
ohne Attribute	Sat Feb 06 18:51:06 CET 2010
type="date"	06.02.2010
type="time"	17:51:06
type="both"	06.02.2010 17:51:06
type="both" timeZone="America/Los_Angeles"	06.02.2010 09:51:06
type="both" timeZone="Europe/Berlin"	06.02.2010 18:51:06
type="date" dateStyle="short"	06.02.10
type="date" dateStyle="medium"	06.02.2010
type="date" dateStyle="long"	6. Februar 2010
type="date" dateStyle="full"	Samstag, 6. Februar 2010
type="time" dateStyle="short"	17:51:06
type="time" dateStyle="medium"	17:51:06
type="time" dateStyle="long"	17:51:06
type="time" dateStyle="full"	17:51:06
type="date" pattern="dd.MM.yyyy"	06.02.2010
type="date" pattern="dd. MMM yyyy"	06. Feb 2010
type="date" pattern="Heute, EEEE 'der' dd. MMMM yyyy"	Heute, Samstag der 06. Februar 2010
type="date" pattern="dd.MM.yyyy G, HH:mm:ss:SSS"	06.02.2010 n. Chr., 17:51:06:911

Abbildung 4.10: Beispiele zur Konfiguration des DateTimeConverters

Die im Muster verwendbaren Zeichen sind in Tabelle 4.6 dargestellt. Die Alternativen sind sehr umfangreich. Wir verzichten daher auf eine komplette Darstellung und verweisen auf die Dokumentation des Java-SDKs. JavaServer Faces haben für das Muster die Definition in der Klasse `java.text.SimpleDateFormat` übernommen, so dass die dortige Dokumentation auch auf Java-Server Faces zutrifft.

Auf die Verwendung des Attributs `timeZone` gehen wir nicht ein. Als Beispiel wurde in Abbildung 4.10 „America/Los_Angeles“ und „Europe/Berlin“ verwendet. Das herunterladbare Projekt enthält eine JSF-Seite, die alle bekannten Zeitzonen ausgibt.

Die Verwendung des Zahlenkonvertierers `<f:convertNumber>` gestaltet sich ähnlich flexibel wie die des Datumskonvertierers. Auch seine Aufgabe ist vor allem in der Lokalisierung und in der Spezifikation eines Zahlenformats zu sehen. Tabelle 4.7 zeigt die möglichen Attribute der Komponente. Im Fall der

Tabelle 4.6: Mögliche Zeichen zur Verwendung im pattern-Attribut

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

Verwendung eines Musters kann das Muster analog zum Verfahren bei der Klasse `java.text.DecimalFormat` spezifiziert werden. Wir verzichten daher auf eine komplette Darstellung und führen in Tabelle 4.8 nur die wichtigsten Zeichen auf. Abbildung 4.11 zeigt Beispiele für verschiedene Konfigurationen.

Grundlage des Beispiels ist die Bean-Property

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

die immer nach dem Muster

```
<h:inputText value="#{numberHandler.bigDecimalValue}">  
  <f:convertNumber type="currency" currencyCode="EUR"/>  
</h:inputText>
```

in einer Eingabekomponente verwendet wird. Dadurch ist es möglich, dass Sie nach dem Herunterladen die JSF-Seite für eigene Übungen verwenden.

Aufgabe 4.4

Verwenden Sie die in den Beispielen noch nicht verwendeten Attribute `groupingUsed` und `integerOnly`.

Tabelle 4.7: Attribute des <f:convertNumber>-Tags

Attribut	Werte und Beschreibung
<code>type</code>	number (Default), currency oder percent. Anzeige einer Zahl, einer Währung oder eines Prozentsatzes.
<code>locale</code>	Lokalisierung, entweder als Instanz von <code>java.util.Locale</code> oder als String.
<code>currencyCode</code>	Dreistelliger Währungs-Code nach ISO 4217. Nur möglich, wenn <code>type=currency</code> . Alternative zu <code>currencySymbol</code> .
<code>currencySymbol</code>	Währungssymbol, nur möglich, wenn <code>type=currency</code> . Alternative zu <code>currencyCode</code> .
<code>minFractionDigits</code>	Minimale Anzahl von Nachkommastellen.
<code>maxFractionDigits</code>	Maximale Anzahl von Nachkommastellen.
<code>minIntegerDigits</code>	Minimale Stellenzahl der Ganzzahl.
<code>maxIntegerDigits</code>	Maximale Stellenzahl der Ganzzahl.
<code>groupingUsed</code>	Schalter zur Anzeige des Gruppierungszeichens. Default ist <code>true</code> .
<code>integerOnly</code>	Schalter, ob nur ganzzahliger Anteil der Eingabe verarbeitet werden soll. Default ist <code>false</code> .
<code>pattern</code>	Angabe eines Patterns zur Formatierung. Alternative zu <code>type</code> . Details siehe Tabelle 4.8.

Tabelle 4.8: Mögliche Zeichen zur Verwendung im pattern-Attribut

Zeichen	Bedeutung
0	Eine Ziffer. Bei Nichtexistenz wird '0' dargestellt.
#	Eine Ziffer. Bei Nichtexistenz keine Darstellung.
.	Dezimalseparator der aktuellen Lokalisierung.
,	Gruppierungssymbol der aktuellen Lokalisierung.
,	Fluchtzeichen für Text.

4.4.3 Konvertierung von Aufzählungstypen

Nach der Einführung von Aufzählungstypen in Java 5 wurde in JSF 1.2 ein Konvertierer für Aufzählungstypen eingeführt. Mit diesem können Strings sehr einfach in Werte von Aufzählungstypen und umgekehrt konvertiert werden. Für den Aufzählungstyp **Familienstand**

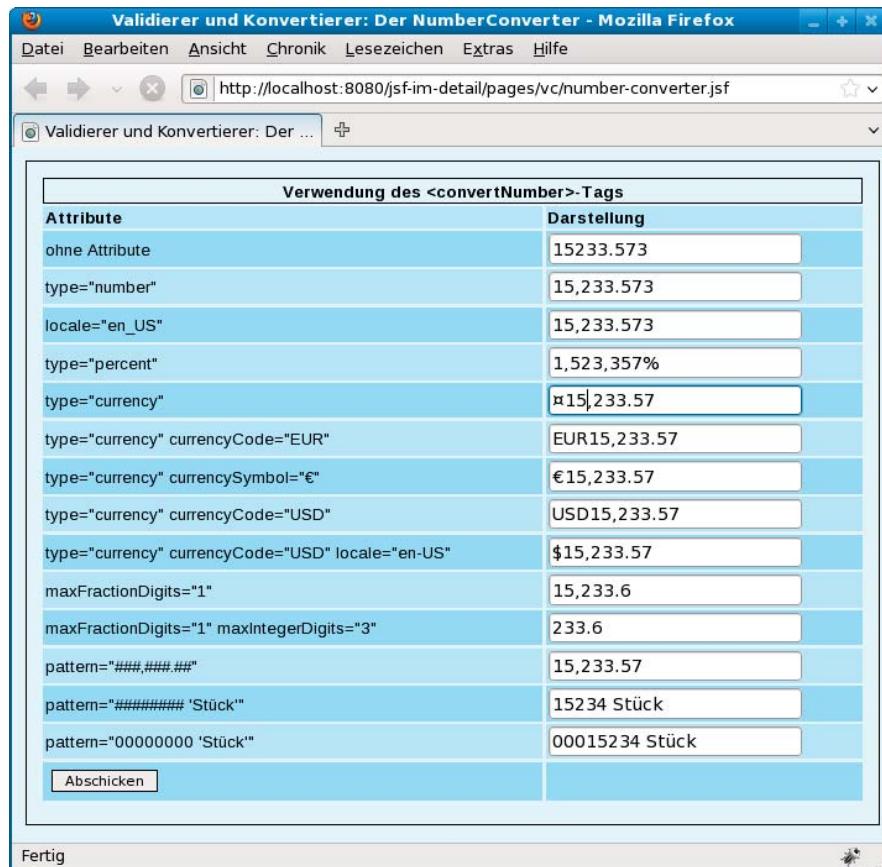


Abbildung 4.11: Beispiele zur Konfiguration des NumberConverters

```
public enum Familienstand {  
    LEDIG, VERHEIRATET, GESCHIEDEN, VERWITWET  
}
```

können dessen Werte in JSF-Seiten als Strings eingegeben werden. Werden die oben genannten Werte in der Großschreibung in die folgende Eingabekomponente

```
<h:inputText value="#{enumHandler.familienstand}" />
```

eingegeben, so erfolgt eine automatische Konvertierung für das entsprechende Property der Managed Bean.

```
@ManagedBean  
public class EnumHandler {  
  
    private Familienstand familienstand;
```

```
// Getter und Setter  
}
```

Aufzählungstypen werden aber typischerweise nicht in textuellen Eingaben, sondern in Auswahlmenüs verwendet. Das Listing 4.12 zeigt die Verwendung des Aufzählungstyps in einem Auswahlmenü, das mit dem <h:selectOneMenu>-Tag realisiert wird.

Listing 4.12: Aufzählungstyp in Auswahlmenü

```
<h:panelGrid columns="2">  
    <f:facet name="header">  
        Validierer und Konvertierer: Enum-Konvertierer mit Auswahl  
    </f:facet>  
    <h:outputLabel for="enum" value="Enum:" />  
    <h:selectOneMenu id="enum"  
                      value="#{enumHandler.familienstand}">  
        <f:selectItem itemLabel="ledig" itemValue="LEDIG" />  
        <f:selectItem itemLabel="verheiratet"  
                      itemValue="VERHEIRATET" />  
        <f:selectItem itemLabel="geschieden"  
                      itemValue="GESCHIEDEN" />  
        <f:selectItem itemLabel="verwitwet" itemValue="VERWITWET" />  
    </h:selectOneMenu>  
    <h:commandButton value="Abschicken"  
                      action="#{enumHandler.abschicken}" />  
    <h:panelGroup />  
</h:panelGrid>
```

Die auszuwählenden Elemente werden mit <f:selectItem>-Tags realisiert. Das Attribut **itemLabel** definiert den anzuzeigenden Text, das Attribut **itemValue** den server-seitigen Wert. Wie wir vorher gesehen haben, werden die großgeschriebenen String-Konstanten automatisch konvertiert.

Diese Verwendung ist jedoch nicht besonders wartungsfreundlich. Änderungen an den Werten des Aufzählungstyps – in diesem Beispiel eher unwahrscheinlich – sind im allgemeinen Fall durchaus möglich und erfordern Änderungen in allen JSF-Seiten, die eine derartige Auswahl verwenden. Eine wartungsfreundlichere Implementierungsalternative verwendet das <f:selectItems>-Tag wie folgt:

```
<h:selectOneMenu id="enum"  
                  value="#{enumHandler.familienstand}">  
    <f:selectItems value="#{enumHandler.values}" />  
</h:selectOneMenu>
```

Hiermit wird die Bereitstellung der Anzeigetexte und Auswahlwerte auf den Server verlagert und somit auf eine Stelle konzentriert. Über die Wertebindung wird die Methode `getValues()`, die ein Array von `SelectItem`-Instanzen zurückgibt, verwendet.

```
public Object[] getValues() {
    SelectItem[] items =
        new SelectItem[Familienstand.values().length];
    for (int i = 0; i < items.length; i++) {
        items[i] = new SelectItem(Familienstand.values()[i],
            Familienstand.values()[i].toString().toLowerCase());
    }
    return items;
}
```

Der mehrfach überladene Konstruktor der Klasse `SelectItem` verwendet in der von uns verwendeten Version als ersten Parameter ein beliebiges Objekt und als zweiten den anzugebenden String.

4.4.4 Anwendungsdefinierte Konvertierer

Die von JSF bereitgestellten Konvertierer für die Standarddatentypen und Aufzählungstypen reichen bei komplexeren Anwendungen häufig nicht aus. Daher sieht JSF die Möglichkeit vor, anwendungsspezifische Konvertierer definieren zu können. Als Beispiel verwenden wir Kreditkartennummern, die wir als eigenen Datentyp definieren. Um tatsächlich konvertieren zu müssen, definieren wir das API etwas umständlich und lassen als Nummer einen einfachen String nicht zu. Die Implementierung Klasse `Kreditkartennummer` besteht lediglich aus einem Konstruktor mit vier Parametern und der überschriebenen `toString()`-Methode:

```
public class Kreditkartennummer {

    public Kreditkartennummer(String q1, String q2,
                               String q3, String q4) {
        ...
    }

    @Override
    public String toString() {
        ...
    }
}
```

Da die innere Struktur unerheblich ist, verzichten wir auf eine Darstellung. Die JSF-Anwendung soll nun die Eingabe einer Kreditkartennummer in zwei

Formaten erlauben. Die Quadrupel, die nur aus Ziffern bestehen, dürfen durch jeweils ein Leerzeichen oder ein Minuszeichen voneinander getrennt werden. Alle anderen Formate sind nicht erlaubt.

JSF-Konvertierer müssen das Interface `Converter` im Package `javax.faces.convert` implementieren. Dieses besteht aus den beiden Methoden `getAsObject()` und `getAsString()`. Listing 4.13 zeigt die Klasse `KreditkartennummerConverter`, die diese beiden Methoden implementiert.

Listing 4.13: Konvertierer für Kreditkartennummern

```
public class KreditkartennummerConverter implements Converter {

    public Object getAsObject(FacesContext context,
                               UIComponent component, String value) {
        // wir prüfen nur ein bisschen und lassen den Rest
        // den Konstruktor machen
        Kreditkartennummer nummer = null;
        String[] quads = null;
        if (value.indexOf('-') != -1) {
            quads = value.trim().split("-");
        } else {
            quads = value.trim().split(" ");
        }
        if (quads.length != 4) {
            FacesMessage message =
                new FacesMessage(FacesMessage.SEVERITY_ERROR,
                                 "Keine Kreditkartennummer",
                                 "Keine Kreditkartennummer");
            throw new ConverterException(message);
        }
        try {
            nummer = new Kreditkartennummer(quads[0], quads[1],
                                            quads[2], quads[3]);
        } catch (Exception e) {
            FacesMessage message =
                new FacesMessage(FacesMessage.SEVERITY_ERROR,
                                 "Keine Kreditkartennummer",
                                 "Keine Kreditkartennummer");
            throw new ConverterException(message);
        }
        return nummer;
    }

    public String getAsString(FacesContext context,
                             UIComponent component, Object value) {
        return ((Kreditkartennummer) value).toString();
    }
}
```

Wir vernachlässigen hier die Diskussion zur Prüfung des korrekten Datenformats, da die Implementierung evident ist. Interessanter sind die erzeugten Objekte vom Typ **FacesMessage**, die JSF-spezifische Fehlermeldungen bzw. Warnungen repräsentieren. Diese werden im Abschnitt 4.4.10 erläutert, weshalb wir dies ebenfalls nicht diskutieren. Es bleibt noch zu erwähnen, dass die Signatur der beiden Konvertierermethoden die Exception-Klasse **ConverterException** zur Signalierung von Fehlern verwendet, wenngleich im Beispiel nur bei der ersten Methode verwendet.

Der nun definierte Konvertierer kann auf verschiedene Arten verwendet werden. Zum einen global zur Konvertierung aller Eingaben von Kreditkartennummern oder lokal zur Konvertierung einer Eingabe. Wir beginnen mit der globalen Alternative, bei der in der JSF-Seite kein konvertiererspezifischer Code zu verwenden ist:

```
<h:inputText  
    value="#{kreditkartenHandler.kreditkartennummer}" />
```

Seit JSF 2.0 kann die Annotation **@FacesConverter** verwendet werden, die mit dem Attribut **forClass** einen Konvertierer global für die angegebene Klasse registriert; also:

```
@FacesConverter(forClass = Kreditkartennummer.class)  
public class KreditkartennummerConverter  
    implements Converter {  
    ...
```

Die alternative Definition in der JSF-Konfigurationsdatei erfolgt mit:

```
<converter>  
    <converter-class>  
        KreditkartennummerConverter  
    </converter-class>  
    <converter-for-class>  
        Kreditkartennummer  
    </converter-for-class>  
</converter>
```

Soll der Konverter nicht global verwendet werden, so ist in der Eingabekomponente der Konvertierer zu referenzieren. Dies kann direkt über eine Wertebindung geschehen:

```
<h:inputText value="#{kreditkartenHandler.kreditkartennummer}"  
    converter="#{kreditkartenHandler.converter}" />
```

Der Getter **getConverter()** muss hier die Konvertierer-Instanz zurückliefern. Dies kann bei Bedarf durch eine anonyme Klasse realisiert werden, da die entsprechenden Methoden auch auf die privaten Daten der Managed Bean

zugreifen können. Eine weitere Alternative ist die Vergabe eines Bezeichners für den Konvertierer, der sich dann analog zu oben im converter-Attribut verwenden lässt:

```
<h:inputText value="#{kreditkartenHandler.kreditkartennummer}"
    converter="kreditkartenkonvertierer" />
```

Alternativ kann das <f:converter>-Tag verwendet werden:

```
<h:inputText value="#{kreditkartenHandler.kreditkartennummer}">
    <f:converter converterId="kreditkartenkonvertierer"/>
</h:inputText>
```

Die Frage ist nun, wie der Bezeichner vergeben wird. Dies kann wiederum über die Annotation `@FacesConverter` oder über XML erfolgen.

```
@FacesConverter("kreditkartenkonvertierer")
public class KreditkartennummerConverter
    implements Converter {
    ...

    <converter>
        <converter-id>
            kreditkartenkonvertierer
        </converter-id>
        <converter-class>
            KreditkartennummerConverter
        </converter-class>
    </converter>
```

Abschließen wollen wir die Ausführungen zu anwendungsdefinierten Konvertierern mit dem converterMessage-Attribut, das alle Eingabekomponenten besitzen. Falls die im Konvertierer definierten Fehlermeldungen (siehe Listing 4.13 auf Seite 85) nicht erwünscht sind oder nicht in der gewünschten Lokalisierung vorliegen, kann mit dem converterMessage-Attribut explizit eine andere Fehlermeldung definiert werden. Abbildung 4.12 zeigt ein Beispiel im Browser, das durch die folgende Verwendung des Attributs entstand:

```
<h:inputText value="#{kreditkartenHandler.kreditkartennummer}"
    converter="#{kreditkartenHandler.converter}"
    converterMessage="Falscheingabe"/>
```

4.4.5 Standardvalidierer

Die Validierung von Benutzereingaben ist eine der Hauptaufgaben eines GUIs. JavaServer Faces verfügen daher bereits über eine Reihe von Validierern, und es ist sehr einfach, eigenentwickelte Validatorer hinzuzufügen. Die Standardvalidierer sind im Package `javax.faces.validator` enthalten. Es sind dies die

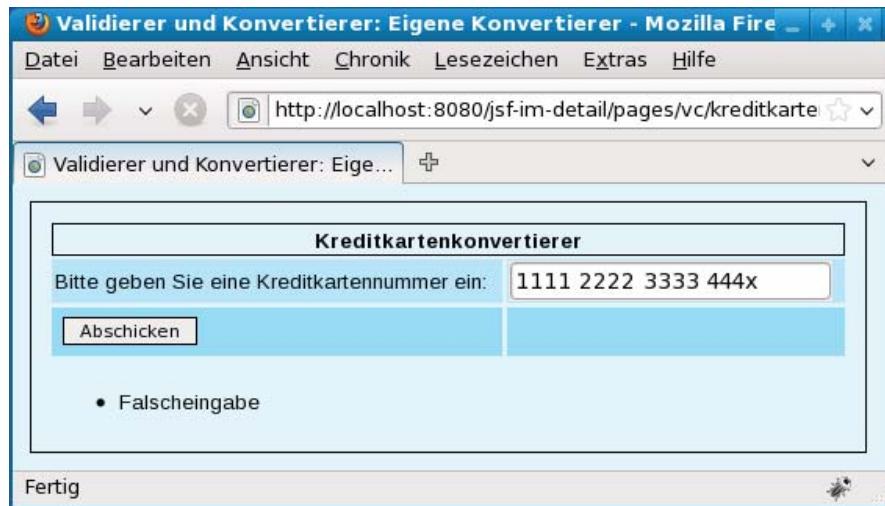


Abbildung 4.12: Kreditkartenkonvertierung mit Fehler

Klassen `LengthValidator`, `LongRangeValidator` und `DoubleRangeValidator`. In Version 1.2 kam die Klasse `MethodExpressionValidator` hinzu, die intern verwendet und von uns nicht weiter beachtet wird. In Version 2.0 kamen die Klassen `RegexValidator`, `RequiredValidator` und `BeanValidator` hinzu. Der auf dem `BeanValidator` basierenden Bean-Validierung ist Abschnitt 4.4.9 gewidmet.

Wir beginnen mit den einfachen Längen- und Bereichsvalidierern. Der `LengthValidator` überprüft die Länge, die beiden Bereichsvalidierer den Wert einer Eingabe. Sowohl die Länge als auch der Wert werden jeweils durch zwei Attribute begrenzt, `minimum` und `maximum`. Es gibt keine Default-Werte für diese Attribute, d. h. falls sie nicht angegeben werden, ist der entsprechende Bereich nicht begrenzt. Die Verwendung ist simpel:

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

In diesem Beispiel wird die Eingabe an die Bean-Property `longValue` gebunden. Diese muss vom Typ `Long` oder `long` sein. Der erlaubte Zahlenbereich für den Long-Wert ist der Bereich zwischen 100 und 500. Weil die Standardvalidierer eine nicht vorhandene Eingabe bis zur JSF Version 2.0 als valide ansehen, wird eine Eingabe verlangt (`required="true"`). Ab 2.0 ist die Behandlung leerer Eingaben konfigurationsabhängig und wird von uns später erläutert.

Die Validierer `LengthValidator` und `DoubleRangeValidator` werden durch die Tags `<f:validateLength>` und `<f:validateDoubleRange>` deklariert. Ein Beispiel, das alle drei Validierer enthält und nach obigem Schema erstellt wurde, zeigt Abbildung 4.13. Die obere Darstellung zeigt die erneute Anzeige nach korrekten Eingaben, die untere nach im Sinne der Validierung falschen Eingaben. Die Fehlermeldungen wurden durch ein `<h:messages showDetail="true"/>` erzeugt, worauf wir in Abschnitt 4.4.10 näher eingehen. Die View ist englisch lokalisiert. Abschnitt 4.7 erläutert, wie dies geändert werden kann.

Der mit JSF 2.0 hinzugekommene Validierer `<f:validateRegex>` überprüft die Übereinstimmung einer Eingabe mit einem regulären Ausdruck. Ebenfalls neu in 2.0 ist der Validierer `<f:validateRequired>`, der überprüft, ob eine Eingabe existiert. Wir haben eine derartige Überprüfung in unseren Beispielen bereits häufig vorgenommen, allerdings mit Hilfe des Attributs `required`. Beide Alternativen sind in der Default-Konfiguration gleichwertig verwendbar. Die Unterschiede sprechen wir in Abschnitt 4.4.9 an. Die Alternative über den expliziten Validierer ergibt ein homogeneres Bild, verursacht aber mehr Schreibaufwand.

Als Beispiel der beiden neuen Validierer sollen nun zwei E-Mail-Eingaben überprüft werden. Listing 4.14 zeigt zwei Eingabemöglichkeiten für E-Mail-Adressen.

Listing 4.14: Die Validierer `<f:validateRegex>` und `<f:validateRequired>`

```
<h:panelGrid columns="2">
    <f:facet name="header">Standardvalidierer (ab 2.0)</f:facet>

    <span> E-Mail-1: </span>
    <h:inputText id="email1" value="#{eingabeHandler.email1}"
        required="true">
        <f:validateRegex pattern=".+@.+\..+" />
    </h:inputText>

    <span> E-Mail-2: </span>
    <h:inputText id="email2" value="#{eingabeHandler.email2}">
        <f:validateRequired />
        <f:validateRegex pattern=".+@.+\..+" />
    </h:inputText>
    <h:commandButton action="#{eingabeHandler.abschicken}"
        value="Abschicken" />
</h:panelGrid>
```

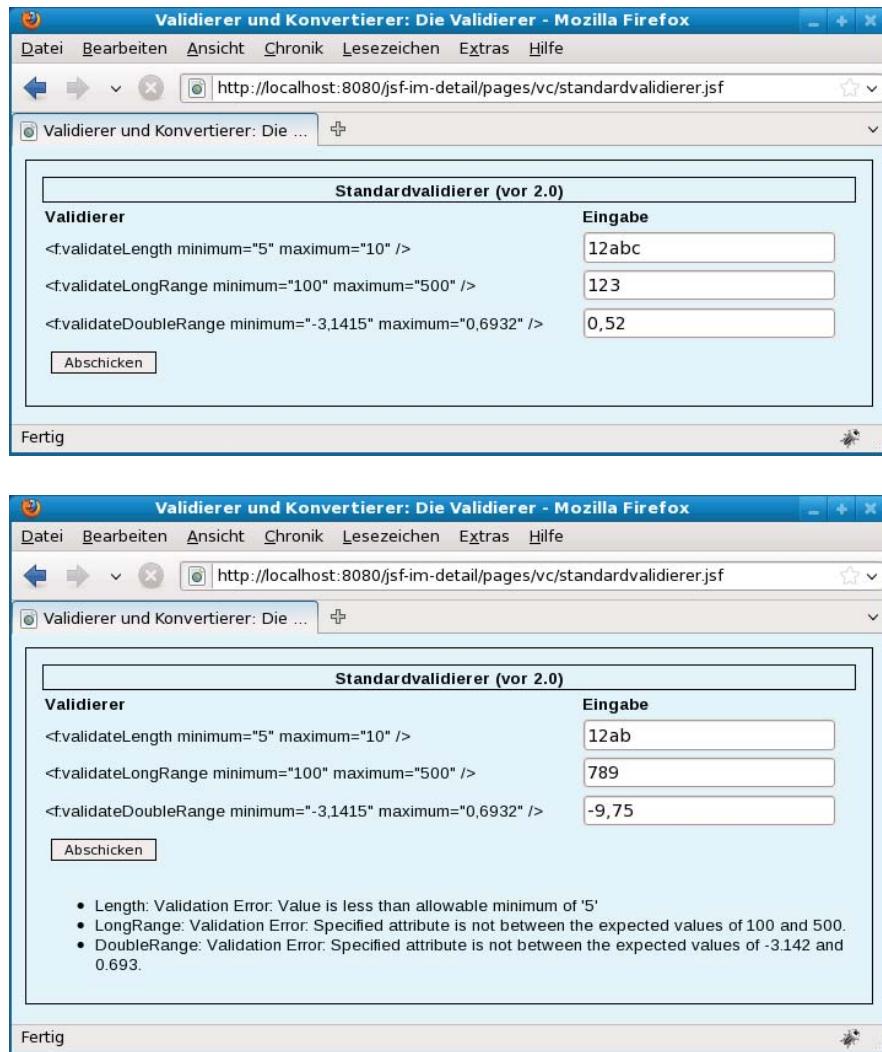


Abbildung 4.13: Standardvalidierer (positive und negative Validierung)

Beide prüfen die Eingabe über einen regulären Ausdruck unter Verwendung von `<f:validateRegex>`. Die erste Eingabe prüft das Vorhandensein einer Eingabe mit `required="true"`, die zweite mit `<f:validateRequired>`.

Das `<f:validateRequired>`-Tag kann nicht nur wie im Beispiel innerhalb einer Eingabekomponente verwendet werden, sondern auch mehrere Tags umfassen. Es erspart so die mehrfache Verwendung des `required`-Attributs. Das folgende Code-Beispiel deutet die Verwendung an.

```
<f:validateRequired>
  ...
  <h:inputText id="email1" ...
    <f:validateRegex pattern=".+@.+\..+" />
  </h:inputText>
  ...
  <h:inputText id="email2" ...
    <f:validateRegex pattern=".+@.+\..+" />
  </h:inputText>
  ...
</f:validateRequired>
```

Die Verwendung der Standardvalidierer ist nicht auf einfache Texteingaben beschränkt, sondern mit allen Eingabekomponenten möglich. In der Regel ist die Auswahl einer Drop-Down-Liste, einer Check-Box oder eines Radio-Buttons aber nur mit sehr wenigen potenziellen Fehlermöglichkeiten verbunden, da ja die Anwendung die auszuwählenden Alternativen vorgibt. Es ist jedoch denkbar, dass die Auswahl der Werte durch bestimmte Bedingungen eingeschränkt wurde, als Auswahlmöglichkeiten jedoch mehr Alternativen angezeigt werden, als valide sind. Eine solche Eingabe realisiert der JSF-Code in Listing 4.15.

Listing 4.15: Drop-Down-Liste mit Validator

```
1 <h:panelGrid columns="2" styleClass="borderTable" border="1">
2   <h:outputText value="Wählen Sie einen Wert zwischen"
3     "#{eingabeHandler.min} und #{eingabeHandler.max}" />
4   <h:selectOneMenu id="zahlenauswahl" required="true"
5     value="#{eingabeHandler.menuauswahl}">
6     <f:selectItem itemValue="" itemLabel="" />
7     <f:selectItem itemValue="1" itemLabel="eins" />
8     <f:selectItem itemValue="2" itemLabel="zwei" />
9     <f:selectItem itemValue="3" itemLabel="drei" />
10    <f:selectItem itemValue="4" itemLabel="vier" />
11    <f:selectItem itemValue="5" itemLabel="fünf" />
12    <f:selectItem itemValue="6" itemLabel="sechs" />
13    <f:selectItem itemValue="7" itemLabel="sieben" />
14    <f:selectItem itemValue="8" itemLabel="acht" />
15    <f:selectItem itemValue="9" itemLabel="neun" />
16    <f:validateLongRange minimum="#{eingabeHandler.min}"
17      maximum="#{eingabeHandler.max}" />
18  </h:selectOneMenu>
19  <h:commandButton action="#{eingabeHandler.abschicken}"
20    value="Abschicken" />
21  <h:outputText
22    value="Auswählt wurde: #{eingabeHandler.menuauswahl}" />
23 </h:panelGrid>
24 <h:messages showDetail="true" />
```

Zunächst erfolgt in Zeile 5 eine Wertebindung des Property `menueauswahl` an die Komponente `<h:selectOneMenu>`. Die anzugezeigenden Alternativen der Drop-Down-Liste beginnen mit der leeren Auswahl, gefolgt von den dargestellten Werten `eins` bis `neun`, die intern die Zahlen 1 bis 9 repräsentieren. Dies ist ein häufig verwendetes Muster, um den Benutzer zur expliziten Auswahl eines Wertes zu zwingen. Standardmäßig ist die erste Auswahl, in diesem Fall die leere, selektiert. Durch das Attribut `required="true"` ist diese aber nicht erlaubt. Der Benutzer wird so gezwungen, eine andere Alternative auszuwählen, und kann nicht unüberlegt auf die Schaltfläche klicken. Interessant ist nun die Verwendung des `LongRangeValidators` in den Zeilen 16 und 17. Der selektierte Wert muss zwischen zwei durch die Managed Bean definierten Grenzen liegen, die dynamisch durch die Anwendung definiert werden können. Dieselben Grenzen werden auch bei der textuellen Eingabeaufforderung in Zeile 3 verwendet. Abbildung 4.14 zeigt die Darstellung im Browser: zunächst mit einer Auswahl außerhalb der vorgeschriebenen Grenzen und dann mit einer korrekten Auswahl. Zur Kontrolle der korrekten Übernahme des Wertes in die Managed Bean wird der Wert in den Zeilen 21/22 durch eine Ausgabekomponente ausgegeben.

4.4.6 Validierungsmethoden

Die bisherigen Validierungen wurden von den Standardvalidierern vorgenommen und sind daher auf syntaktische und einfache semantische Validierungen beschränkt. JavaServer Faces bieten mit Methoden zur Validierung und mit komplett selbstdefinierten Validierern die Möglichkeit, anwendungsbezogene Validierer zu entwickeln, die wesentlich komplexere Validierungen vornehmen können. Beispiele sind etwa die Überprüfung einer E-Mail-Adresse auf korrekte Syntax auf Basis von bekannten Domains oder die Überprüfung eines Überweisungsbetrages im Online-Banking derart, dass bei Durchführung der Überweisung der Überziehungskredit nicht überschritten wird. Die Entwicklung eigenständiger Validierer wird in Abschnitt 4.4.7 beschrieben; die Entwicklung von Validierungsmethoden erläutern wir im Folgenden.

Jede Eingabekomponente erlaubt die Bindung einer Validierungsmethode einer Managed Bean an sich. Dies geschieht analog zur Methodenbindung, mit dem Unterschied, dass das zu verwendende Attribut der Komponente das Attribut `validator` ist. So wird etwa mit

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

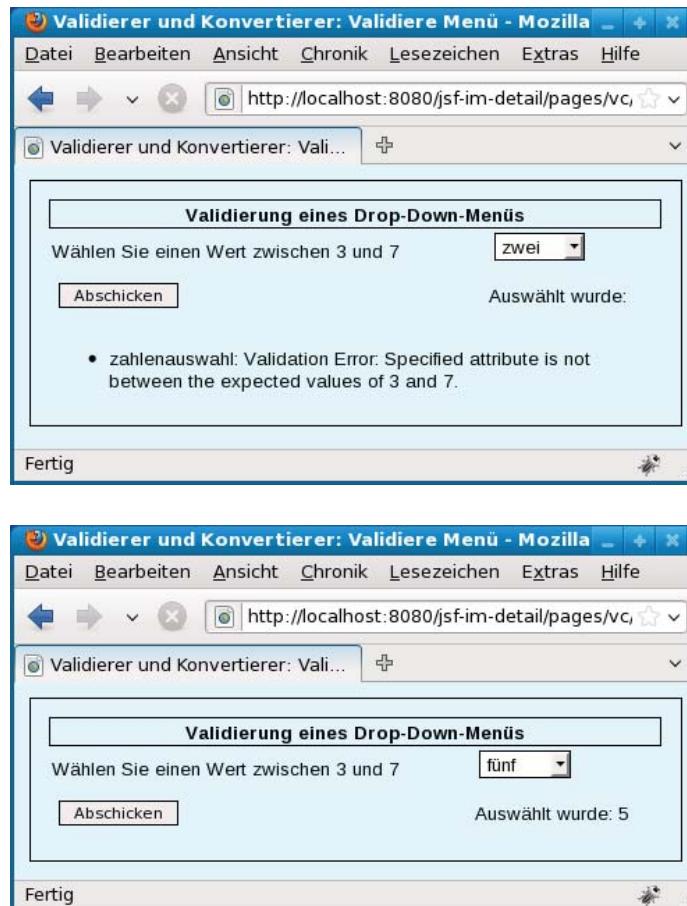


Abbildung 4.14: Validierung einer Drop-Down-Liste

die Methode `validateEmail` an das Property `textValue` gebunden. Die Signatur der Validierungsmethode entspricht der Methode `validate` des Interface `javax.faces.validator.Validator`. Als Methodenbezeichner sind allerdings beliebige Namen erlaubt. Die Methode `validateEmail` ist definiert als

```
public void validateEmail(FacesContext context,
                           UIComponent component,
                           Object value)
        throws ValidatorException {
    if (!((String) value).matches(".+@.+\\..+")) {
        throw new ValidatorException(
            new FacesMessage("Fehlerhafte E-Mail-Syntax"));
    }
}
```

Als Parameter bekommt die Methode den Faces-Context des aktuellen Requests, die Komponente, deren Wert zu validieren ist, und den Wert selbst übergeben. Wir verwenden in der Methode nur den Wert. Die Überprüfung auf eine korrekte E-Mail-Adresse ist stark simplifiziert und reduziert sich im Prinzip auf ein einfaches Muster mit Klammeraffen und einem Punkt im rechten Teil für die Domain, wie wir es bereits im Beispiel für `<f:validateRegex>` verwendet haben. Die Methode `matches()` der Klasse `String` testet auf Übereinstimmung mit einem regulären Ausdruck. Wenn ein Wert valide ist, terminiert die Methode. Wenn nicht, wirft sie eine `ValidatorException`, eine Klasse, die ebenfalls im Package `javax.faces.validator` enthalten ist. Der Parameter des Exception-Konstruktors ist eine Fehlermeldung, auf die wir im nächsten Abschnitt eingehen.

Aufgabe 4.5

Schreiben Sie eine Validierungsmethode, die prüft, ob eine Texteingabe eine Kreditkartennummer ist. Kreditkartennummern sind 16 Ziffern, die in Vierergruppen angeordnet und durch ein Minus- oder Leerzeichen getrennt sind. Also z. B. „1234-5678-9876-5432“ oder „1234 5678 9876 5432“. Eine inhaltliche Überprüfung soll nicht vorgenommen werden. Verknüpfen Sie die Validierungsmethode mit einer JSF-Eingabe, und testen Sie die Seite.

4.4.7 Anwendungsdefinierte Validierer

Analog zur Definition anwendungsdefinierter Konvertierer (siehe Abschnitt 4.4.4) erlaubt JSF auch die Definition von anwendungsdefinierten Validierern. Im Gegensatz zu Konvertierern, die für eine bestimmte Klasse definiert werden können, können Validierer nur über eine vergebene Id verwendet werden. Sowohl in der XML- als auch in der Annotations-Definition ist daher nur die Verwendung über den Namen erlaubt.

Die Klasse `EmailValidator` implementiert das Interface `Validator`.

```
@FacesValidator("emailvalidator")
public class EmailValidator implements Validator {

    public void validate(FacesContext context,
                         UIComponent component, Object value)
        throws ValidatorException {
        if (!((String) value).matches(".+@.+\\..+")) {
            throw new ValidatorException(
                new FacesMessage(FacesMessage.SEVERITY_ERROR,
                                 "Fehlerhafte E-Mail-Syntax",
                                 "Fehlerhafte E-Mail-Syntax"));
        }
    }
}
```

Mit der Annotation `@FacesValidator` wird eine Instanz der Klasse unter dem Namen `emailvalidierer` registriert. Die Verwendung in der JSF-Seite erfolgt dann über das `validator`-Attribut, analog zur Validierungsmethode,

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

oder über das `<f:validator>`-Tag:

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

Soll die Definition und Namensvergabe des Validierers nicht über die Annotation, sondern über die JSF-Konfigurationsdatei vorgenommen werden, so ist der folgende Eintrag zu verwenden:

```
<validator>
    <validator-id>emailvalidierer</validator-id>
    <validator-class>EmailValidator</validator-class>
</validator>
```

4.4.8 Eingabekomponenten und das `immediate`-Attribut

Bei der Darstellung der Phase 2 des JSF-Anfragezyklus in Abschnitt 4.1.2 haben wir erwähnt, dass bei Eingabekomponenten, deren boolesches Attribut `immediate` gesetzt ist, Konvertierung und Validierung bereits bei der Übernahme der Anfragewerte und nicht in der Validierungsphase stattfinden. Doch wo wird dieses Verhalten benötigt?

Ein JSF-Request ist bis zur Version 2.0 immer ein vollständiger Request, d.h. alle Komponenten des Komponentenbaums werden vollständig entsprechend dem Lebenszyklus behandelt. Wenn nun eine der Eingabekomponenten auf eine Eingabe besteht (`required="true"` oder `<f:validateRequired>`), hat dies Auswirkungen auf die komplette Seite. Eine Verarbeitung eines Teils der Seite ist nicht möglich, wenn diese Eingabe nicht vorhanden ist. Wenn z.B. neben den obligatorischen Daten eines Kunden noch die Daten für die Bezahlweise einer Bestellung einzugeben sind, diese sich jedoch abhängig von der Auswahl der Bezahlweise (Kreditkarte, Bankeinzug, Rechnung) unterscheiden, so kann eine Änderung der Auswahl nicht als JSF-Request abgeschickt werden, solange die obligatorischen Daten nicht angegeben sind.

Mit der Einführung von Ajax in Version 2.0 ist eine teilweise Verarbeitung des Komponentenbaums möglich. Wir werden für ein ähnliches Problem in Abschnitt 7.3.3 eine Ajax-Lösung entwickeln. Vor der Version 2.0 kommt als

Lösungsalternative nur die Verwendung des **immediate**-Attributs in Frage. Wir entwickeln ein minimales Beispiel mit zwei Eingabekomponenten, von denen die eine Komponente zwingend vorgeschrieben ist und die andere die JSF-Seite verändert. Listing 4.16 zeigt eine Tabelle mit Beschriftungen und Eingabemöglichkeiten als Teil der Seite `immediate.xhtml`. Abbildung 4.15 zeigt die Darstellung im Browser.

Listing 4.16: Verwendung des `immediate`-Attributs

```
1 <h:panelGrid columns="2">
2   <h:outputLabel value="#{immediateHandler.addressLabels[
3     immediateHandler.language]}"/>
4   <h:inputText value="#{immediateHandler.address}"
5     required="true" />
6   <h:outputLabel value="#{immediateHandler.languageLabels[
7     immediateHandler.language]}"/>
8   <h:selectOneMenu id="menu" value="immediateHandler.language"
9     onchange="this.form.submit()" immediate="true"
10    valueChangeListener="#{immediateHandler.languageChanged}">
11      <f:selectItems value="#{immediateHandler.languages}" />
12    </h:selectOneMenu>
13    <h:commandButton action="#{immediateHandler.action}"
14      value="OK" />
15  </h:panelGrid>
```



Abbildung 4.15: Eingabe mit `immediate`-Attribut (`immediate.xhtml`)

Der Eingabetext ist verpflichtend (`required="true"`, Zeile 5). Sowohl das Auswahlmenü (`onchange="this.form.submit()"`, Zeile 9) als auch der Command-Button schicken das Formular ab. Die Reaktion auf eine Änderung der Sprachauswahl soll die Beschriftung der Eingaben ändern. Dies ist je-

doch bei einer leeren Anschrift nicht möglich. Abhilfe schafft hier das Attribut **immediate** (Zeile 9), das die Konvertierung und Validierung für die mit **immediate** versehene Komponente in die Phase 2 – Übernahme der Anfragewerte – vorzieht. Listing 4.17 zeigt die Managed Bean **ImmediateHandler**.

Listing 4.17: Der ImmediateHandler

```
@ManagedBean
@SessionScoped
public class ImmediateHandler implements Serializable {

    private static final String LANG1 = "Deutsch";
    private static final String LANG2 = "Englisch";

    private String address;
    private String language = LANG1;
    private Map<String, String> addressLabels;
    private Map<String, String> languageLabels;

    public ImmediateHandler() {
        addressLabels = new HashMap<String, String>(2);
        addressLabels.put(LANG1, "Anschrift");
        addressLabels.put(LANG2, "Address");
        languageLabels = new HashMap<String, String>(2);
        languageLabels.put(LANG1, "Sprache");
        languageLabels.put(LANG2, "Language");
    }

    public void languageChanged(ValueChangeEvent vce) {
        language = (String) vce.getNewValue();
        FacesContext.getCurrentInstance().renderResponse();
    }

    public String[] getLanguages() {
        return new String[] { LANG1, LANG2 };
    }

    // Getter und Setter
}
```

Hier erkennt man die Methode **languageChanged()**, die in der JSF-Seite (Listing 4.16 auf der vorherigen Seite) in Zeile 10 als Value-Change-Listener registriert wurde. Wird das Formular nach einer Änderung des Auswahlmenüs über die JavaScript-Funktion **submit()** abgeschickt, wird in Phase 2 – Übernahme der Anfragewerte – diese Listener-Methode aufgerufen. Die Methode

setzt die Sprache und springt dann sofort zu Phase 6, Rendern der Antwort. Dies ist notwendig, da sonst in der 3. Phase die Validierung der Adress-Eingabe erfolgen und bei nicht vorhandener Adresse zu einem Fehler führen würde.

Die beiden Beschriftungen (`<h:outputLabel>`) werden jeweils über eine Map mit Texten versorgt. Dies ist im Prinzip auch die Grundlage der JSF-Lokalisierung, wie wir sie in Abschnitt 4.7 kennenlernen werden.

Aufgabe 4.6

Nachdem Sie das Kapitel 7 durchgearbeitet haben, überarbeiten Sie das Beispiel. Verwenden Sie Ajax, und verzichten Sie auf das `immediate`-Attribut.

4.4.9 Bean-Validierung mit JSR 303

Die Validierung ist als zentraler Baustein einer jeden Anwendung zu sehen. In der Praxis konnte jedoch *eine zentrale* Validierung in einer Java-EE-Anwendung häufig nicht realisiert werden, und es wurde z. B. in der Präsentations- und in der Persistenz-Schicht validiert. Mit der Spezifikation *Bean Validation* als JSR 303 [URL-JSR303] wird versucht, diese redundaten Validierungen zu vermeiden. Dem liegt die Idee zugrunde, wonach die Properties eines Geschäftsobjekts in POJO-Form (eine Bean) mit Validierungsannotationen versehen und diese unabhängig von der Verwendung des Objekts, sei es in der Präsentations-, der Persistenz- oder einer anderen Schicht, überwacht werden. Tabelle 4.9 zeigt die Annotationen, die die Bean Validation als Constraints bereitstellt. Wir verzichten auf eine ausführlichere Darstellung, da die Constraints sehr intuitiv benannt wurden, und ermuntern den Leser zu einer Lektüre der API-Doc des Package `javax.validation.constraints` in der Java-EE-6-Dokumentation. Einige Constraints werden wir im Folgenden verwenden.

Da der JSR 303 in Java-EE 6 enthalten ist, können wir die Bean Validation in Glassfish ohne weitere Konfiguration direkt verwenden. Listing 4.18 zeigt einen Ausschnitt der JSF-Seite `bean-validation.xhtml`. Die Eingaben sind weder mit `required="true"` noch mit `<f:validateRequired>` versehen.

Listing 4.18: Dateneingabe der Seite `bean-validation.xhtml`

```
1 <h:panelGrid columns="2">
2   <f:facet name="header">
3     Bean-Validierung eines Kunden
4   </f:facet>
5
6   <h:outputLabel value="Vorname" for="vor" />
7   <h:inputText id="vor"
8     value="#{kundeHandler.kunde.vorname}" />
```

Tabelle 4.9: Constraint-Annotationen des JSR 303

Annotation	Beschreibung
@AssertFalse	Property muss <code>false</code> sein.
@AssertTrue	Property muss <code>true</code> sein.
@DecimalMax	Property kleiner oder gleich Wert von <code>value</code> .
@DecimalMin	Property größer oder gleich Wert von <code>value</code> .
@Digits	Property darf nicht mehr Vor- und Nachkommastellen haben, als in <code>integer</code> und <code>fraction</code> angegeben.
@Future	Property muss Datum in der Zukunft sein.
@Max	Property kleiner oder gleich Wert von <code>value</code> .
@Min	Property größer oder gleich Wert von <code>value</code> .
@NotNull	Property darf nicht <code>null</code> sein.
@Null	Property muss <code>null</code> sein.
@Past	Property muss Datum in der Vergangenheit sein.
@Pattern	Property muss Pattern in <code>regexp</code> entsprechen.
@Size	Größe des Property muss zwischen <code>min</code> und <code>max</code> liegen.

```
9   <h:outputLabel value="Nachname" for="nach" />
10  <h:inputText id="nach"
11    value="#{kundeHandler.kunde.nachname}" />
12  <h:outputLabel value="Geburtsdatum" for="gebu" />
13  <h:inputText id="gebu"
14    value="#{kundeHandler.kunde.geburtstag}">
15    <f:convertDateTime pattern="dd.MM.yyyy" />
16  </h:inputText>
17  <h:outputLabel value="Kreditlimit" for="limit" />
18  <h:inputText id="limit"
19    value="#{kundeHandler.kunde.kreditlimit}" />
20  <h:commandButton action="#{kundeHandler.speichern}"
21    value="Speichern" />
22 </h:panelGrid>
```

Die Managed Bean `kundeHandler` besteht lediglich aus dem Property `kunde` und der nicht dargestellten Methode `speichern()`:

```
@ManagedBean
public class KundeHandler {

    private Kunde kunde;

    public KundeHandler() {
        kunde = new Kunde();
    }

    // Getter und Setter
}
```

Die Validierungsannotationen werden an den Kunden-Properties angebracht. Sie dürfen sowohl die Variablen direkt als auch das Getter/Setter-Paar annotieren. Wir annotieren die Variablen und verzichten auf die Darstellung der Getter und Setter.

```
public class Kunde {

    @NotNull
    private String vorname;

    @Size(min = 3, max = 30)
    private String nachname;

    @Past
    private Date geburtstag;

    @Min(0)
    @Max(1000)
    private Integer kreditlimit;
    ...
}
```

Das Beispiel dient der Vorstellung einiger der möglichen Annotationen und ist im Anwendungskontext eventuell nur bedingt sinnvoll. Die Annotation `@NotNull` verhindert, dass im Vornamen kein Wert steht. Die Annotation `@Size` erzwingt eine Länge des Nachnamen zwischen 3 und 30 Zeichen. Durch die Annotation `@Past` wird gewährleistet, dass der Geburtstag in der Vergangenheit liegt. Mit den Annotationen `@Min` und `@Max` wird schließlich das zu gewährende Kreditlimit auf 0 bis 1000 (Euro) beschränkt.

Werden nun Werte eingegeben, die nicht den Bedingungen entsprechen, erzeugt die Implementierung der Bean Validation entsprechende Fehlermeldungen. Diese werden an JSF weitergegeben und können mit `<h:message>` und `<h:messages>` wie JSF-eigene Nachrichten angezeigt werden. Abbildung 4.16 zeigt für alle vier Validierungen den Fehlerfall als Beispiel. Die Meldungen wurden jeweils mit einem `<h:message>` erzeugt.

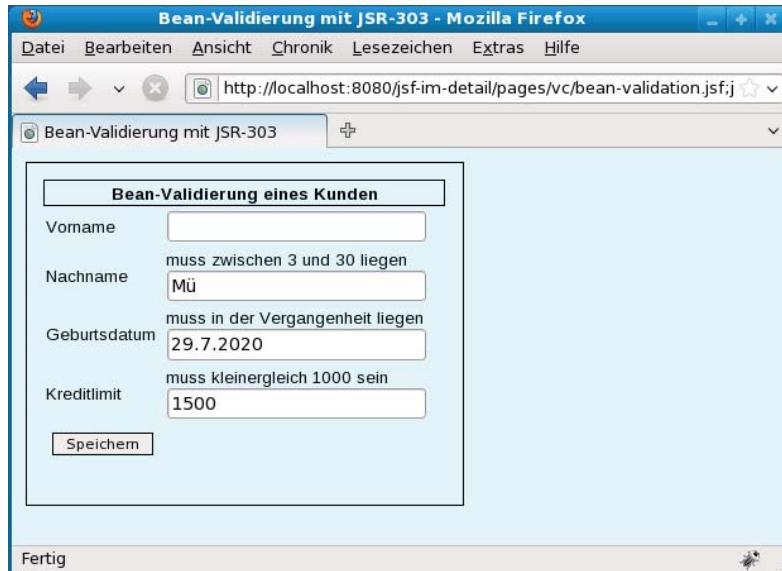


Abbildung 4.16: Die Seite `bean-validation.xhtml` mit Validierungsfehlern

Ohne Eingabe eines Vor- und Nachnamens wird für den Vornamen keine Fehlermeldung erzeugt. Für den Nachnamen wird bemängelt, dass die Eingabe nicht zwischen 3 und 30 Zeichen lang ist. Dieses Verhalten ist das Default-Verhalten, das für eine nicht vorhandene Texteingabe einen leeren String als Wert annimmt. Damit die Überprüfung korrekt arbeitet und mit sinnvollerem Fehlermeldungen reagiert, muss der Kontextparameter

`INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL`

auf `true` gesetzt werden. In der überarbeiteten Version werden nun sowohl Vor- als auch Nachname auf Existenz und Größe geprüft:

```
@NotNull  
@Size(min = 3, max = 30)  
private String vorname;  
  
@NotNull  
@Size(min = 3, max = 30)  
private String nachname;
```

Weitere Informationen zu Kontextparametern findet man im Abschnitt 4.8.1, insbesondere in Tabelle 4.12 auf Seite 163. Erwähnt werden soll an dieser Stelle noch der Kontextparameter `VALIDATE_EMPTY_FIELDS`, der steuert, ob leere Eingaben validiert werden. Der Default-Wert ist `false`. Beim Vorhandensein einer Bean-Validation-Implementierung wird er jedoch automatisch auf `true`

gesetzt, so dass die oben beschriebene Funktionalität von `@Null` und `@NotNull` tatsächlich umgesetzt wird.

Neben den in Tabelle 4.9 dargestellten Constraints erlaubt Bean Validation auch benutzerdefinierte Constraints. Hierzu benötigt man eine Annotation und einen Constraint-Validierer. Die vollständige Darstellung des JSR 303 liegt außerhalb der Ziele dieses Buches, so dass wir lediglich den Code darstellen, ihn aber nicht vollständig erläutern. Im Augenblick existiert kein Buch zum Thema Bean Validation. Da die Spezifikation selbst als Einführung relativ ungeeignet ist, schlagen wir dem Leser die Dokumentation des Hibernate Validator [URL-HIBVAL] vor. Der Hibernate Validator ist in der Version 4.0 die Referenzimplementierung der Bean-Validation-Spezifikation und als Open-Source-Implementierung frei verfügbar.

Als Beispiel für einen benutzerdefinierten Constraint wollen wir die Volljährigkeit eines Kunden prüfen. Bei der Code-Darstellung beginnen wir mit der Annotation. Wir definieren die Annotation `@Volljaehrig`, die zukünftig anstatt der Annotation `@Past` für den Geburtstag verwendet werden soll:

```
@Constraint(validatedBy = VolljaehrigValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Volljaehrig {
    String message() default "Nicht volljährig";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Die Definition verwendet die Bean-Validation-eigene Annotation `@Constraint`, um den noch vorzustellenden Constraint-Validierer `VolljaehrigValidator` zu binden. Bei der Implementierung verweisen wir den interessierten Leser auf die oben genannte Informationsquelle, merken aber an, dass die Fehlermeldung besser mit den in Abschnitt 4.7 noch einzuführenden Resource-Bundles realisiert werden sollte.

Bleibt noch die Implementierung des Constraints in der Klasse `Volljaehrig-Validator` mit der Validierungsmethode `isValid()`:

```
public class VolljaehrigValidator
    implements ConstraintValidator<Volljaehrig, Date> {

    @Override
    public void initialize(Volljaehrig constraintAnnotation) {
        // TODO Auto-generated method stub
    }

    @Override
```

```
public boolean isValid(Date date,
    ConstraintValidatorContext constraintValidatorContext) {
    // heute
    Calendar cal = new GregorianCalendar();
    // heute vor 18 Jahren
    cal.add(Calendar.YEAR, -18);
    return date.before(cal.getTime());
}
```

Auch hier wollen wir den Rahmen unseres Buches nicht sprengen und verweisen den Leser auf die entsprechenden anderweitigen Informationen. Durch die Definition der Annotation und des Constraint-Validierers kann man nun die Annotation bequem verwenden:

```
@Volljaehrig
private Date geburtstag;
```

Als letzten interessanten Punkt der Bean Validation wollen wir uns die Gruppenvalidierung anschauen. Häufig kommt es vor, dass eine Menge von Benutzereingaben nur zu validieren sind, wenn bestimmte Bedingungen erfüllt sind. Beispielsweise werden die Kreditkartendaten nur validiert, wenn diese Bezahlart auch ausgewählt wurde. Bean Validation führt das Konzept der Gruppenvalidierung ein, bei der eine Menge von Constraints unter einem Namen zusammengefasst werden. Die Validierung dieser Gruppen kann dann eingeschaltet werden.

Um das vorherige Beispiel zu erweitern, soll das Kreditlimit für Volljährige entfallen, für Minderjährige aber weiterhin gelten. Hierzu definieren wir zwei Interfaces, `VolljaehrigGruppe` und `NichtVolljaehrigGruppe`, die beide von der Klasse `Default` im Package `javax.validation.groups` ableiten müssen:

```
public interface VolljaehrigGruppe extends Default {
}
```

Da auch die Klasse `NichtVolljaehrigGruppe` einen leeren Rumpf hat, verzichten wir auf eine Darstellung.

Mit Hilfe dieser beiden Gruppen können nur Validierungen selektiv durchgeführt werden. Im folgenden Code werden die Constraints `@Past` und `@Volljaehrig` in Abhängigkeit der angegebenen Gruppen aktiviert.

```
@Past(groups = NichtVolljaehrigGruppe.class)
@Volljaehrig(groups = VolljaehrigGruppe.class)
private Date geburtstag;

@Min(0)
@Max(value = 1000, groups = NichtVolljaehrigGruppe.class)
private Integer kreditlimit;
```

Das Attribut **groups** erlaubt als optionales Attribut aller Validierungs-Annotatiionen die Angabe einer Liste von Gruppen-Interfaces. In unserem Beispiel sind dies die beiden oben definierten Interfaces, die in Kombination die oben formulierte Anforderung eines unbegrenzten Kreditlimits für volljährige und ein maximales Kreditlimit vom 1000 Euro für minderjährige Kunden realisiert.

Der Default-Wert für **groups** ist die leere Gruppe, die wir in der Definition der Annotation **@Volljaehrig** auf Seite 102 bereits verwendet haben. Validierungen ohne **groups**-Attribut werden also immer durchgeführt.

Es stellt sich nun die Frage, wo die Steuerung der Validierungsgruppen erfolgt. Hierzu wird das letzte noch nicht vorgestellte Validierungs-Tag **<f:validateBean>** verwendet. Im folgenden Code-Ausschnitt werden die Eingaben von Geburtstag und Kreditlimit auf Basis der ausgewählten Validierungsgruppe realisiert.

```
<h:outputLabel value="Geburtsdatum" />
<h:inputText id="gebu"
              value="#{kundeHandler.kunde.geburtstag}">
    <f:convertDateTime pattern="dd.MM.yyyy" />
    <f:validateBean validationGroups=
        "de.jsfpraxis.detail.vc.NichtVolljaehrigGruppe" />
</h:inputText>

<h:outputLabel value="Kreditlimit" />
<h:inputText id="limit"
              value="#{kundeHandler.kunde.kreditlimit}">
    <f:validateBean validationGroups=
        "de.jsfpraxis.detail.vc.NichtVolljaehrigGruppe" />
</h:inputText>
```

Da als Parameter des Attributs **validationGroups** allgemeine EL-Ausdrücke zugelassen sind, kann anders als im Beispiel eine anwendungsfallabhängige Steuerung der Validierung erfolgen.

Wir beschließen hiermit unsere Ausführungen zur Bean Validation. Da Bean Validation eine eigenständige Spezifikation ist, konnten wir nur die Oberfläche ansprechen. Bean Validation wurde explizit mit der Anforderung einer optimalen Integration in JSF und JPA entworfen. Wir empfehlen dem Leser, sich näher mit Bean Validation zu beschäftigen, da damit ein großer Schritt in Richtung einfache, redundanzfreie Validierung innerhalb komplexer Unternehmensanwendungen mit Java-EE 6 möglich ist.

Falls keine vollständige Java-EE-6-Umgebung zur Verfügung steht, weil z.B. Tomcat verwendet wird, kann Hibernate Validator [URL-HIBVAL] als Referenzimplementierung der Bean Validation zusätzlich verwendet werden.

4.4.10 Fehlermeldungen

Konvertierungen und Validierungen können gelingen, sie können aber auch mit einem Fehler enden. Diese Fehler und andere wichtige Informationen sollten im Allgemeinen dem Benutzer mitgeteilt werden. JavaServer Faces verwenden dazu ein definiertes Verfahren, das sowohl festlegt, wie der Fehler intern signalisiert als auch, wie er dem Benutzer mitgeteilt wird.

Die JSF-Spezifikation definiert in Abschnitt 2.5.2.4 einen Satz von Fehlermeldungen, die in einer Ressourcen-Datei definiert und für mehrere Sprachen lokalisiert werden müssen bzw. können. Ressourcen-Dateien und Lokalisierung behandeln wir detailliert in Abschnitt 4.7. Ressourcen-Dateien bestehen aus Schlüsseln und Werten. Einige Fehlermeldungen existieren sowohl in einer Kurz- als auch einer Langform. Der Schlüssel der Langform ergibt sich durch Anhängen von „`_detail`“ an die Kurzform. Die Spezifikation gibt eine Reihe von Schlüsseln definitiv und Werte für diese Schlüssel empfehlend vor. Diese Schlüssel und Werte sind im Resource-Bundle `javax.faces.Messages` zu realisieren. Ein Resource-Bundle ist die Zusammenfassung von Ressourcen-Dateien nach einem bestimmten Schema, das, wie bereits erwähnt, in Abschnitt 4.7 erläutert wird.

Die JSF-Referenzimplementierung Mojarra enthält Lokalisierungen für Englisch, Deutsch, Französisch, Spanisch, Portugiesisch, Koreanisch, Japanisch und Chinesisch. Die Tabelle 4.10 führt die Schlüssel der Fehlermeldungen auf, um einen quantitativen Eindruck zu ermöglichen. Zeilen mit angehängtem „`(detail)`“ bedeuten, dass dies Zeile im Original aus zwei Zeilen besteht. Die Schlüssel stammen aus der Datei `Messages.properties`.

Tabelle 4.10: Schlüssel für Standardfehlermeldungen

<code>javax.faces.component.UIInput.CONVERSION</code>
<code>javax.faces.component.UIInput.REQUIRED</code>
<code>javax.faces.component.UIInput.UPDATE</code>
<code>javax.faces.component.UISelectOne.INVALID</code>
<code>javax.faces.component.UISelectMany.INVALID</code>
<code>javax.faces.converter.BigDecimalConverter.DECIMAL (detail)</code>
<code>javax.faces.converter.BigIntegerConverter.BIGINTEGER (detail)</code>
<code>javax.faces.converter.BooleanConverter.BOOLEAN (detail)</code>
<code>javax.faces.converter.ByteConverter.BYTE (detail)</code>
<code>javax.faces.converter.CharacterConverter.CHARACTER (detail)</code>
<code>javax.faces.converter.DateTimeConverter.DATE (detail)</code>

Tabelle 4.10: Schlüssel für Standardfehlermeldungen (Fortsetzung)

javax.faces.converter.DateTimeConverter.TIME (detail)
javax.faces.converter.DateTimeConverter.DATETIME (detail)
javax.faces.converter.DateTimeConverter.PATTERN_TYPE
javax.faces.converter.DoubleConverter.DOUBLE (detail)
javax.faces.converter.EnumConverter.ENUM (detail)
javax.faces.converter.EnumConverter.ENUM_NO_CLASS (detail)
javax.faces.converter.FloatConverter.FLOAT (detail)
javax.faces.converter.IntegerConverter.INTEGER (detail)
javax.faces.converter.LongConverter.LONG (detail)
javax.faces.converter.NumberConverter.CURRENCY (detail)
javax.faces.converter.NumberConverter.PERCENT (detail)
javax.faces.converter.NumberConverter.NUMBER (detail)
javax.faces.converter.NumberConverter.PATTERN (detail)
javax.faces.converter.ShortConverter.SHORT (detail)
javax.faces.converter.STRING
javax.faces.validator.DoubleRangeValidator.MAXIMUM
javax.faces.validator.DoubleRangeValidator.MINIMUM
javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE
javax.faces.validator.DoubleRangeValidator.TYPE
javax.faces.validator.LengthValidator.MAXIMUM
javax.faces.validator.LengthValidator.MINIMUM
javax.faces.validator.LongRangeValidator.MAXIMUM
javax.faces.validator.LongRangeValidator.MINIMUM
javax.faces.validator.LongRangeValidator.NOT_IN_RANGE
javax.faces.validator.LongRangeValidator.TYPE
javax.faces.validator.NOT_IN_RANGE
javax.faces.validator.RegexValidator.PATTERN_NOT_SET (detail)
javax.faces.validator.RegexValidator.NOT_MATCHED (detail)
javax.faces.validator.RegexValidator.MATCH_EXCEPTION (detail)
javax.faces.validator.BeanValidator.MESSAGE

Zur Erläuterung geben wir die Fehlermeldungen des Boolean-Konvertierers und des Längenvalidierers an:

```
# Ausschnitt aus Messages.properties
javax.faces.converter.BooleanConverter.BOOLEAN=\
```

```
{1}: '{0}' must be 'true' or 'false'.
javax.faces.converter.BooleanConverter.BOOLEAN_detail=\
{1}: '{0}' must be 'true' or 'false'. \
Any value other than 'true' will evaluate to 'false'.

javax.faces.validator.LengthValidator.MAXIMUM=\
{1}: Validation Error: Length is greater than allowable \
maximum of '{0}'
javax.faces.validator.LengthValidator.MINIMUM=\
{1}: Validation Error: Length is less than allowable \
minimum of '{0}'
```

Die Fehlermeldung für den Boolean-Konvertierer existiert sowohl in Kurz- als auch in Langform. Für den Längenvalidierer gibt es nur die Kurzform, jedoch sowohl für die obere als auch die untere Grenze des zulässigen Intervalls.

Als Vorgriff auf Abschnitt 4.7 zeigt der folgende Dateiausschnitt die deutschen Fehlermeldungen für den Längenvalidierer.

```
# Ausschnitt aus Messages_de.properties
javax.faces.validator.LengthValidator.MAXIMUM=\
{1}: \u00dcberpr\u00fcfung fehler: L\u00e4nge ist \
gr\u00f6ßer als der zul\u00e4ssige Maximalwert "{0}"
javax.faces.validator.LengthValidator.MINIMUM=\
{1}: \u00dcberpr\u00fcfung fehler: L\u00e4nge ist \
kleiner als der zul\u00e4ssige Minimalwert "{0}"
```

Man erkennt bei diesen, aber auch bei den vorherigen Fehlermeldungen, dass die Fehlermeldungen parametrisiert sind. Die Parametrisierung erfolgt nach einem Standardverfahren für Meldungen, das in der Java-SE-Klasse **MessageFormat** im Package `java.text` festgelegt wird. Dabei werden Zahlen als Positionsparameter verwendet und zur Markierung als Parameter in geschweifte Klammern eingeschlossen. Woher diese Parameter kommen, erläutern wir später.

Bei den Fehlermeldungen unterscheidet JSF zwischen speziell für eine Komponente erzeugten Meldungen und Meldungen für die gesamte JSF-Seite. Die komponentenbasierte Meldung wird mit `<h:message>` erzeugt, die globale Meldung mit `<h:messages>`. Das erstgenannte Tag hat daher auch das Attribut **for**, das den Komponentenbezeichner enthalten muss, für den die Fehlermeldung steht. Das letztgenannte Tag besitzt das boolesche Attribut **globalOnly**, das anzeigt, ob alle Meldungen einer Seite angezeigt werden oder nur die, die keiner Komponente zugewiesen, also quasi global sind.

Der folgende Code-Ausschnitt definiert für die Input-Komponente mit der Id **length** (Zeile 4) einen Längenvalidierer. Für die Input-Komponente wird in Zeile 8 durch das `<h:message>`-Tag eine Fehlermeldungskomponente re-

gistriert. In Zeile 11 wird zusätzlich global eine Fehlermeldungkomponente registriert. Dies ist in diesem Fall nicht sinnvoll und dient nur zu Demonstrationszwecken.

```
1   <h:form id="form">
2     <h:panelGrid columns="2">
3       <h:outputLabel for="length" value="Text (5-10 Zeichen)" />
4       <h:inputText id="length" value="#{meldungenHandler.text}"
5         required="true">
6         <f:validateLength minimum="5" maximum="10" />
7       </h:inputText>
8       <h:message for="length" styleClass="meldung" />
9     </h:panelGrid>
10    <h:commandButton action="sucess" value="Abschicken" />
11    <h:messages showDetail="true" showSummary="false"/>
12  </h:form>
```

Das obere Browser-Fenster in Abbildung 4.17 zeigt die Darstellung nach einer zu kurzen Texteingabe. Im Quell-Code wird in Zeile 7 als Style die CSS-Klasse **meldung** verwendet, die wie folgt definiert ist.

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

Während **<h:message>** standardmäßig die detaillierte Meldung anzeigt, stellt **<h:messages>** die Zusammenfassung dar. In Zeile 10 werden daher die Attribute **showDetail** und **showSummary** entsprechend gesetzt.



Während der Entwicklung einer JSF-Seite empfehlen wir die Verwendung des Tags **<h:messages>** z. B. am Ende der Seite. So werden alle auftretenden Fehler auch tatsächlich angezeigt. Falls Sie JSF in der Version 2.0 verwenden, erreichen Sie dies mit dem Wert **Development** für den Kontextparameter **PROJECT_STAGE**.

Im oberen Browser-Fenster fällt unangenehm auf, dass die Fehlermeldung mit der Id der Komponente eingeleitet wird. Die Id, in diesem Fall **form:length**, wird nach bestimmten Regeln, die wir in Abschnitt 4.9 erläutern, gebildet. Diese Id sagt aber in der Regel dem Benutzer der Anwendung nichts, weil sie ein technisches Detail darstellt. Die Spezifikation legt die Fehlermeldungen nicht fest, sie können sich daher von Implementierung zu Implementierung unterscheiden. Die von uns verwendete Referenzimplementierung Mojarra verwendet für die Unterschreitung der Mindestlänge das bereits bekannte Property

```
javax.faces.validator.LengthValidator.MINIMUM_detail
```

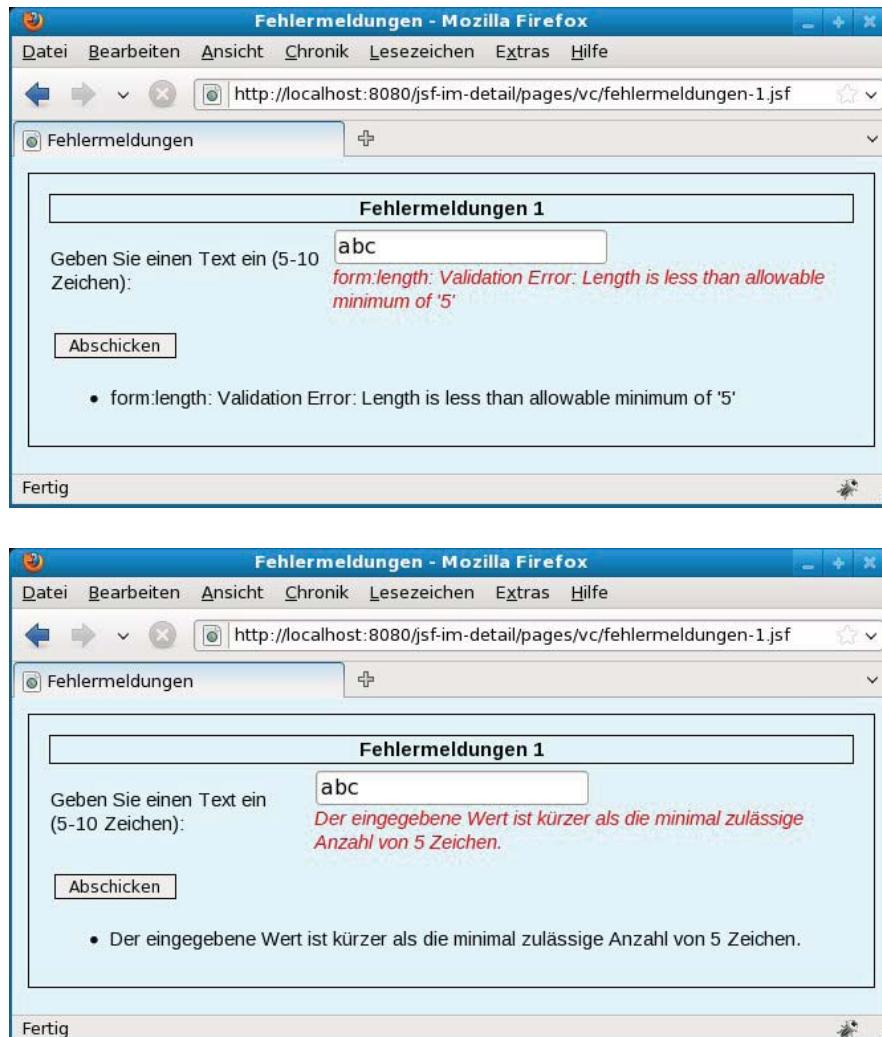


Abbildung 4.17: Fehlermeldung durch Validierungskomponente

mit dem Wert

```
{1}: \u00dcberpr\u00fcfungsfehler: L\u00fclnge ist \
kleiner als der zul\u00fclngige Minimalwert "{0}"
```

JSF sieht die Möglichkeit vor, eigene Fehlermeldungen zu definieren bzw. die vordefinierten zu überschreiben. Dazu muss ein eigenes Message-Bundle in der Konfigurationsdatei `faces-config.xml` deklariert werden:

```
<application>
  <message-bundle>
```

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

Für eine internationalisierte Anwendung müssen dann die entsprechenden Dateien, wie in Abschnitt 4.7 beschrieben, angelegt werden. Im Augenblick beschränken wir uns beispielhaft auf eine deutsche Meldung. Die Datei `meine-meldungen.properties` im Package `de.jsfpraxis.detail_vc` enthält die Zeile

```
javax.faces.validator.LengthValidator.MINIMUM_detail = \
Der eingegebene Wert ist k\ufe0fcrzer als die minimal \
zul\ufe0e4ssige Anzahl von {0} Zeichen.
```

Mit dieser Konfiguration erfolgt dann die Darstellung im Browser, wie im unteren Fenster in Abbildung 4.17 dargestellt.

Die den Fehlermeldungen zugrunde liegende Klasse `FacesMessage` im Package `javax.faces.application` unterscheidet verschiedene Schweregrade (engl. `severity`) von Meldungen. Es sind dies die in `FacesMessage` definierten Konstanten

- `SEVERITY_ERROR`
- `SEVERITY_FATAL`
- `SEVERITY_INFO`
- `SEVERITY_WARN`

Mit ihnen ist es möglich, verschiedene Ebenen von Meldungen zu unterscheiden. Die Tags `<h:message>` und `<h:messages>` stellen dazu die Attribute `errorClass`, `fatalClass`, `infoClass` und `warnClass` zur Angabe einer CSS-Klasse sowie die Attribute `errorStyle`, `fatalStyle`, `infoStyle` und `warnStyle` zur Angabe eines CSS-Styles bereit.

Alle Fehler der Standardvalidierer sind „richtige“ Fehler (`SEVERITY_ERROR`), so dass wir zu Demonstrationszwecken auf eine Eigenentwicklung zurückgreifen müssen. Ziel ist die Weiterverwendung des Längenvalidierers und die Entwicklung einer informativen Meldung, falls die Eingabe exakt die Länge einer der beiden Grenzen (Minimum/Maximum) hat.

Wir ändern das Beispiel so ab, dass die Eingabekomponente einen zusätzlich über das `validator`-Attribut zugewiesenen Validierer bekommt, was im folgenden Listing in Zeile 5 geschieht.

```
1 <h:panelGrid columns="3">
2   <h:outputLabel for="length" value="Text (3-7 Zeichen)" />
3   <h:inputText id="length" value="#{meldungHandler.text}"
```

```
4           required="true"
5           validator="#{meldungenHandler.validateText}>
6   <f:validateLength minimum="3" maximum="7" />
7   </h:inputText>
8   <h:message for="length" errorClass="meldung"
9       infoClass="info" />
10  </h:panelGrid>
11  <h:commandButton action="success" value="Abschicken" />
```

In Zeile 8 wird für einen Fehler über das Attribut **errorClass** die schon bekannte CSS-Klasse **meldung** zugewiesen. Für rein informative Meldungen wird über das Attribut **infoClass** die CSS-Klasse **info** zugewiesen, die als

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

definiert ist. Abbildung 4.18 zeigt im oberen Fenster bei der Eingabe von drei Zeichen die informative Meldung in Blau, das untere Fenster zeigt bei der Eingabe von zwei Zeichen die Fehlermeldung in Rot und größerem Font.

Listing 4.19 enthält die Implementierung der Validierungsmethode **validateText()**.

Listing 4.19: Die Validierungsmethode validateText()

```
1  public void validateText(FacesContext context,
2                      UIComponent component,
3                      Object value)
4      throws ValidatorException {
5  int min = 0, max = 0;
6  int length = ((String) value).length();
7  Validator[] validator = ((UIInput)component).getValidators();
8  String mb = context.getApplication().getMessageBundle();
9  ResourceBundle rb = ResourceBundle.getBundle(mb);
10 for (int i = 0; i < validator.length; i++) {
11     if (validator[i] instanceof LengthValidator) {
12         LengthValidator lv = (LengthValidator) validator[i];
13         min = lv.getMinimum();
14         max = lv.getMaximum();
15         if (length == min || length == max) {
16             String message = rb.getString("de.jsfpraxis.MINMAX");
17             String messageDetail =
18                 rb.getString("de.jsfpraxis.MINMAX_detail");
19             context.addMessage(component.getClientId(context),
20                               new FacesMessage(FacesMessage.SEVERITY_INFO,
21                                               message, messageDetail));
22     }
23 }
```

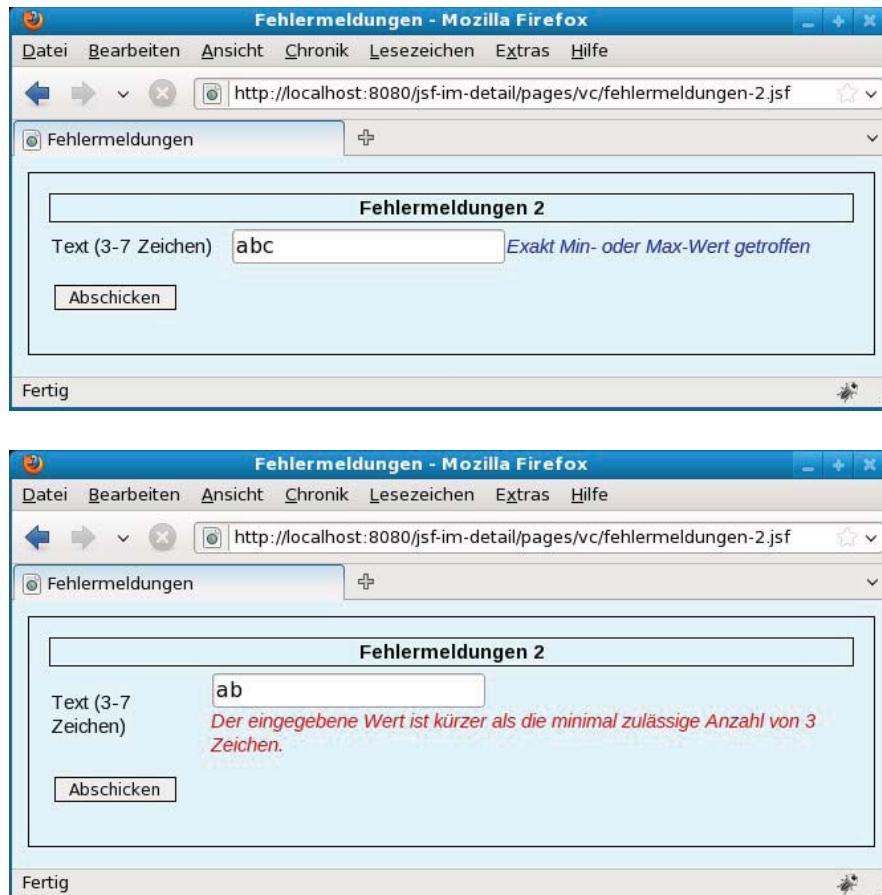


Abbildung 4.18: Fehlermeldung durch Validierungskomponente

```
23      }
24  }
25 }
```

Die Methode `validateText()` muss zwei Probleme lösen: den Zugriff auf den Minimal- und Maximalwert des Längenvalidierers und den Zugriff auf die Meldungen des Message-Bundles. In Zeile 7 wird nach den Validierern der Eingabekomponente gefragt. Im Allgemeinen können dies mehrere sein. In den Zeilen 8 und 9 erfolgt dann der Zugriff auf das Message- bzw. Resource-Bundle. Die For-Schleife iteriert durch alle registrierten Validierer und prüft für einen gefundenen Längenvalidierer, ob die Länge der Eingabe gleich dem Minimal- oder Maximalwert des Validierers ist. Wenn ja, erfolgt ein Zugriff auf

die Properties `de.jsfpraxis.MINMAX` und `de.jsfpraxis.MINMAX_detail`, deren Werte als Parameter für den `FacesMessage`-Konstruktorauftruf in Zeile 20 verwendet werden. Wir verzichten auf die offensichtliche Definition der Properties. Der Konstruktor `FacesMessage()` ist überladen. Die Version mit zwei String-Parametern nimmt als Default-Wert für den Schweregrad den Wert `SEVERITY_INFO`, so dass dies im Beispiel auch entfallen könnte. Die Methode `addMessage()` des Kontext-Objekts fügt die erzeugte Meldung der Liste aller Meldungen für diese Seite hinzu. Weil die Meldung der Eingabekomponente zugeordnet werden soll, ist die Client-Id der Komponente der erste Parameter der Methode. Die Methode `addMessage()` ist nicht überladen. Soll die Meldung keiner Komponente zugeordnet werden, so muss der erste Parameter `null` sein. Schließlich ist für die Namen der beiden Properties noch anzumerken, dass diese frei wählbar sind. Es ist jedoch insbesondere in größeren Projekten mit vielen Entwicklern sinnvoll, eine systematische Vergabe der Namen anzustreben. Im Beispiel erfolgt dies durch eine an Java-Packages angelegte Präfix-Struktur, wie dies auch bei den Standardmeldungen der Fall ist.

Das vorgestellte Verfahren für individuelle Fehlermeldungen ist relativ aufwändig. Seit JSF 2.0 gibt es ein deutlich einfacheres Verfahren. Zum einen kann mit dem Attribut `label` eine Kennzeichnung einer Eingabe über ein Label erfolgen. Statt der internen Komponenten-Id wird dann dieses Label bei Fehlermeldungen verwendet. Im obigen Beispiel also etwa:

```
<h:inputText id="length" label="Text" ... >
<f:validateLength ... />
```

Statt der Fehlermeldung

```
form:length: Überprüfungsfehler: Länge ist ...
```

erscheint die folgende Meldung, da die Id durch das Label ersetzt wird:

```
Text: Überprüfungsfehler: Länge ist ...
```

Neben dem `label`-Attribut wurden mit JSF 2.0 die Attribute `requiredMessage`, `validatorMessage` und `converterMessage` für Eingabekomponenten eingeführt. Mit ihnen lassen sich die entsprechenden Fehler einfach mit Meldungen versehen. Für das obige Beispiel kommen sowohl `requiredMessage` als auch `validatorMessage` in Betracht:

```
<h:inputText value="#{meldungHandler.text}" required="true"
    requiredMessage="Der Text muss eingegeben werden"
    validatorMessage="Die Textlänge liegt außerhalb des \
        zulässigen Bereichs">
<f:validateLength minimum="5" maximum="10" />
```

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

Diese Alternative ist jedoch nicht gleichwertig mit dem Message-Bundle-Ansatz. Die Fehlermeldungen auf Basis eines Message-Bundles sind global in allen Seiten gültig, während die drei genannten Attribute nur für die sie verwendende Eingabekomponente gültig ist. Da als Argument der Attribute jedoch beliebige EL-Ausdrücke vom Typ String zulässig sind, lässt sich ein ähnliches Verhalten simulieren.



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für die Validierung und Konvertierung sind im Projekt *jsf-im-detail* enthalten.

4.5 Event-Verarbeitung

Im Kapitel 2 hatten wir als ein Ziel von JavaServer Faces die Realisierung des MVC-Entwurfsmusters für Web-Anwendungen genannt. Als Mittel zur Realisierung von MVC-basierten GUIs hat sich die event-basierte Verarbeitung von Benutzerinteraktionen durchgesetzt. Praktisch alle graphischen Oberflächen, egal ob mit Java erstellt oder nativ auf UNIX-Systemen mit X11 oder auf Windows mit MFC oder WPF, verwenden das Verarbeitungsmuster, bei dem das Drücken einer Schaltfläche, das Selektieren eines Menüeintrags oder das Füllen eines Eingabefeldes zu verschiedenen Events führt. Diese Events werden vom Controller verarbeitet und ziehen in der Regel den Aufruf von Methoden nach sich. Damit ist es möglich, verschiedene Events verschiedenen Methoden zuzuordnen und so den Code zur Behandlung einzelner Benutzeraktionen in getrennten Code-Stücken zu lokalisieren. Dies entspricht einem Grundgedanken moderner Software-Strukturen.

Bei traditionellen Oberflächen ist die Verarbeitung der Events nahezu trivial, da sie im selben Prozess verarbeitet werden, in dem sie auch erzeugt wurden. Bei Web-Anwendungen sieht dies anders aus. Das Event wird (logisch) im Browser erzeugt, weil der Benutzer mit dem Browser interagiert. Die Verarbeitung findet aber im Server statt. Erschwerend kommt hinzu, dass das HTTP-Protokoll nur für eine sehr kurze Zeit eine Verbindung zwischen Client und Server herstellt, die von wesentlich längeren, nicht verbundenen Zeitabschnitten unterbrochen wird. Eine JSF-Implementierung muss daher einen nicht unerheblichen Aufwand treiben, um dieses Event-Verarbeitungsmuster dennoch zu realisieren. Das im Browser logisch entstandene Event wird erst auf

dem Server zu einem „richtigen“ Java-Event, das dann in üblicher Java-Manier abgearbeitet wird. Grundlage ist das Bearbeitungsmodell in Abschnitt 4.1, das an fast allen Stellen die Generierung und Verarbeitung von Events erlaubt. In der Abbildung 4.1 auf Seite 37 ist dies durch die zwischen den Phasen 2 bis 6 angedeuteten Möglichkeiten des Aufrufs von Event-Listener angedeutet. Diese Event-Listener haben prinzipiell die drei Alternativen,

- das Bearbeitungsmodell unverändert zu durchlaufen,
- in die Phase 6 *Rendern der Antwort* zu springen oder
- die Antwort selbst zu generieren und den Lebenszyklus komplett zu beenden.

Der Sprung in die Phase 6 erfolgt durch den Aufruf der Methode `renderResponse()`, das Beenden des Lebenszyklus durch den Aufruf der Methode `responseComplete()`. Beides sind Methoden der Klasse `FacesContext`. Verwendungsbeispiele für die Methoden finden Sie im weiteren Verlauf des Buches.

4.5.1 JSF-Events und allgemeine Event-Verarbeitung

Das JSF-Framework kannte bis zur Version 1.2 vier Events: Value-Change-Events, Action-Events, Data-Model-Events und Phase-Events. Mit der Version 2.0 wurde diese Anzahl vervielfacht. Abbildung 4.19 auf der nächsten Seite zeigt die Vererbungshierarchie der JSF-Events, wobei aus Gründen der Übersichtlichkeit auf die Unterklassen von `SystemEvent` verzichtet wurde. Die bisher noch nicht erwähnte Klasse `FacesEvent` ist die abstrakte Oberklasse der Verhaltens- und Anwendungs-Events. Verhaltens-Events (`BehaviorEvent` und `AjaxBehaviorEvent`) wurden mit JSF 2.0 eingeführt, um auf Ajax-Requests reagieren zu können. Wir gehen darauf im Kapitel 7 ein. JSF unterscheidet zwischen Anwendungs-Events, zu denen auch Verhaltens-Events gehören, und System-Events. Zu den System-Events gehört die Klasse `SystemEvent` sowie deren Unterklassen. Phase-Events und Data-Model-Events können in erweitertem Sinn auch zu den System-Events gezählt werden; die Übergänge sind fließend.

Zu den Anwendungs-Events gehören die Klassen `ActionEvent` und `ValueChangeEvent`. Das Event `PhaseEvent` ist nicht an Komponenten, sondern an die einzelnen Bearbeitungsphasen einer JSF-Anfrage gebunden. Das Event `DataModelEvent` wird bei Datenänderungen von Datenmodellkomponenten geworfen. Wir haben Datenmodellkomponenten bereits über das Tag `<h: dataTable>` kennengelernt. Im Unterschied zu Action- und Value-Change-

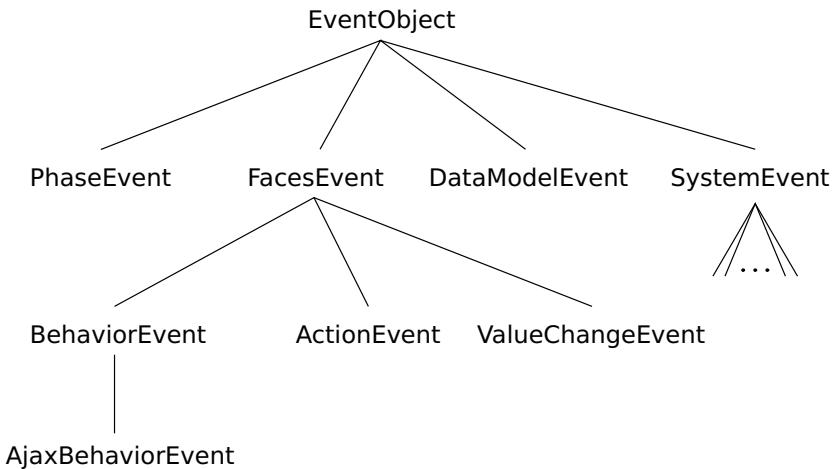


Abbildung 4.19: Hierarchie der JSF-Events

Events werden Datenmodell- und System-Events nicht direkt durch Benutzeraktionen ausgelöst.

Alle Events der Abbildung 4.19 sind bis auf wenige Ausnahmen Klassen des Package `javax.faces.event`. Die Klasse `EventObject` ist im Package `java.util` enthalten und somit eine Java-SE-Klasse und nicht JSF-spezifisch. Die Klasse `DataModelEvent` ist im Package `javax.faces.model` enthalten.

Das Event-Verarbeitungsmodell von JavaServer Faces unterscheidet sich nicht vom bekannten Verarbeitungsmodell Javas. Verschiedene Klassen können bestimmte Events auslösen bzw. werfen, andere Klassen sind daran interessiert, von diesem Umstand zu erfahren. Die an Events interessierten Klassen werden *Event-Listener* genannt. Neben den genannten Event-Klassen enthält das Package `javax.faces.event` auch entsprechende Listener-Interfaces, die Event-Listener-Klassen zu implementieren haben.

Die für uns wichtigsten Listener-Interfaces sind `ActionListener`, `ValueChangeListener`, `DataModelListener` (im Package `javax.faces.model`) und `PhaseListener`. Wir verzichten auf eine Aufzählung der weiteren Listener-Interfaces, da sich deren Namen über das Java-übliche Muster `<name>Event` und `<name>Listener` einfach ergeben und gehen im Folgenden auf die für uns wichtigen Events und deren Listener-Interfaces ein.

4.5.2 Action-Events

Action-Events werden durch Benutzeraktionen ausgelöst, die einem Befehlsmuster folgen, z.B. dem Drücken einer Schaltfläche oder dem Klicken eines Links. Action-Events werden durch Action-Listener bearbeitet. Prinzipiell gibt es zwei Arten von Action-Listenern. Die eine hat einen Rückgabewert, der die Navigation beeinflusst, die andere hat keinen. Beide Arten werden in der Regel eine Programmlogik enthalten, die die Zustände von Managed Beans und Modellobjekten ändert.

Um den Umgang mit Action-Events zu vereinfachen, registriert die JSF-Implementierung implizit einen Action-Listener, der alle Action-Events konsumiert und an anwendungsspezifische Listener weitergibt. Damit müssen Sie keine Klassen realisieren, die das Interface **ActionListener** mit der dort deklarierten Methode

```
public void processAction(javax.faces.event.ActionEvent event)
    throws AbortProcessingException
```

implementiert. Um eine so genannte *Action-Listener-Methode* zu implementieren, genügt es für eine Methode, über einen **void**-Rückgabewert und einen **ActionEvent**-Parameter zu verfügen. Der Methodenname ist frei wählbar, die Exception-Deklaration entfällt. Wir haben eine Action-Listener-Methode im Tic-Tac-Toe-Spiel verwendet (siehe Listing 2.4 auf Seite 17). Die Verwendung einer solchen Methode in den Tags **<h:commandButton>** oder **<h:commandLink>** wird mit dem Attribut **actionListener** (siehe Listing 2.2 auf Seite 14) vorgenommen.

Noch einfacher ist die Verwendung einer so genannten *Action-Methode*, deren Name wiederum frei wählbar ist, die einen **String** zurückgeben muss und keinen Parameter hat. Wir haben für alle Schaltflächen in der Comedian-Anwendung Action-Methoden verwendet. Als Attribut von **<h:commandButton>** und **<h:commandLink>** ist hierfür **action** zu wählen. Seit JSF 1.2 ist als Rückgabetyp **Object** erlaubt, was vor allem in der Integration von Enumerationstypen in JSF 1.2 begründet liegt. Da der Rückgabewert zu Navigationszwecken verwendet wird und diese auf Strings basieren, wird bei allgemeinen Objekten die **toString()**-Methode aufgerufen. In der Regel werden ausschließlich Strings oder Enumerationstypen als Rückgabetyp verwendet. Aus Gründen der Vereinfachung sprechen wir im Folgenden häufig nur von Strings.

An dieser Stelle sei angemerkt, dass der Wert des **action**-Attributs allgemein ein JSF-EL-Ausdruck sein muss, der zu einem String evaluiert. Bei der Verwen-

dung von EL-Literalen kann man daher String-Konstanten direkt verwenden, wie das folgende Beispiel zeigt.

```
<h:commandButton action="#{bean.doSomething}" ...  
<h:commandButton action="success" ...
```

Der String, den eine Action-Methode zurückgibt, ist frei wählbar. Sie können daher Ihrer Anwendung entsprechend sinnvolle Strings verwenden. Für die boolesche Anzeige eines Methodenaufrufs haben sich die beiden Konstanten "success" und "failure" eingebürgert. Auf die Verwendung der Strings innerhalb der Navigation gehen wir im Abschnitt 4.6 detailliert ein.

Interessant in diesem Zusammenhang ist die Parameterlosigkeit von Action-Methoden. Diese ist in der JSF-Spezifikation explizit so formuliert. JSF definiert die Expression-Language aber nicht selbst, sondern verwendet die Unified Expression-Language. Das zweite Maintenance-Review der Unified Expression-Language führt Methodenbindungen mit Parametern ein, wie wir dies bereits für allgemeine Methoden in Abschnitt 4.2.5 dargestellt haben. Für Action-Methoden gilt dies ebenfalls. Für die beiden Action-Methoden

```
public String action() {  
    ...  
}  
  
public String action(String text) {  
    ...  
}
```

kann damit in der JSF-Seite zwischen diesen ausgewählt werden:

```
<h:commandButton value="action()"  
    action="#{actionMethodParamHandler.action}" />  
<h:commandButton value="action('Ein Text')"  
    action="#{actionMethodParamHandler.action('Ein Text')}" />
```

Die Methoden dürfen aber in der augenblicklichen Version der Unified Expression-Language nicht bezüglich der Parametertypen, sondern nur bezüglich der Parameteranzahl überlagert sein. Die Methode

```
public String action(Integer i) {  
    ...  
}
```

wird daher für Integer-Werte nur verwendet, wenn die Version mit String-Parameter nicht vorhanden ist. Eine mögliche Verwendung ist dann z.B. in einer Iteration mit Hilfe des <ui:repeat>-Tags vorstellbar, einem Facelets-Tag, das wir in Kapitel 6 einführen.

```
<ui:repeat var="number"
    value="#{actionMethodParamHandler.numbers}">
    <h:commandButton value="#{number}"
        action="#{actionMethodParamHandler.action(number)}" />
    <br />
</ui:repeat>
```

Die Methode `getNumbers()` liefert ein Array oder eine Liste von Zahlen, über die mit der Variablen `number` iteriert wird. Jeder Command-Button ruft die obige Action-Methode mit einem Integer-Parameter auf.

Man kann sich prinzipiell die Frage stellen, wann Action-Methoden und wann Action-Listener-Methoden zu verwenden sind. Als Faustregel kann man sagen, dass Action-Methoden zu verwenden sind, wenn die Navigation zu steuern ist, und Action-Listener-Methoden zu verwenden sind, wenn möglichst einfach auf das Action-Event und die das Action-Event erzeugende Komponente zugegriffen werden soll.

Die bisher dargestellte vereinfachte Form der Action-Event-Verarbeitung hat einen Nachteil. JSF-Seiten sind ein XML-Dialekt, und XML erlaubt die Verwendung von maximal einem Attribut pro Tag. Das heißt, dass es mit den beiden vorgestellten Alternativen nicht möglich ist, mehrere Action- bzw. Action-Listener-Methoden an einen Benutzerbefehl zu binden. Dies lässt sich mit dem `<f:actionListener>`-Tag erreichen, das innerhalb eines `<h:commandButton>` oder `<h:commandLink>` erlaubt ist. Der folgende Code einer JSF-Seite zeigt eine Schaltfläche mit zwei registrierten Action-Listenern:

```
<h:panelGrid id="panel" columns="1">
    <h:inputTextarea id="textarea" rows="6" cols="50"
        value="#{aeHandler.text}" />
    <h:commandButton id="button" value="OK" action="success">
        <f:actionListener
            type="de.jsfpraxis.events.ActionListener1" />
        <f:actionListener
            type="de.jsfpraxis.events.ActionListener2" />
    </h:commandButton>
</h:panelGrid>
```

Um zu überprüfen, dass die Action-Listener tatsächlich aufgerufen werden, verwenden wir die Eingabekomponente `<h:inputTextarea>`, die eine mehrzeilige Texteingabe erlaubt. Diese Komponente erhält die Id `textarea`, auf die in den Action-Listenern zugegriffen wird. Im `type`-Attribut des `<f:actionListener>` wird der Klassenname des Action-Listeners angegeben. Listing 4.20 zeigt den ersten der beiden verwendeten Action-Listener.

Listing 4.20: Die Klasse ActionListener1

```
1 public class ActionListener1 implements ActionListener {
2
3     public void processAction(ActionEvent ae)
4             throws AbortProcessingException {
5         HtmlInputTextarea text = null;
6         List<UIComponent> components =
7             ae.getComponent().getParent().getChildren();
8         for (UIComponent element : components) {
9             if (element.getId().equals("textarea")) {
10                 text = (HtmlInputTextarea) element;
11             }
12         }
13         text.setValue(text.getValue()
14             + "processAction() von ActionListener1 aufgerufen.\n");
15     }
16 }
```

Die Klasse definiert die im `ActionListener`-Interface geforderte Methode `processAction(ActionEvent)` und ist damit ein `ActionListener`. Die Methode definiert zunächst eine Variable `text` vom Typ `HTMLInputTextarea`. Dies ist die JSF-Komponente, die das `<h:inputTextarea>`-Element realisiert. Dann wird in die Variable `components` eine Liste von Komponenten derselben Ebene geschrieben, da alle Kinder der Elternkomponente erfragt werden. Diese Liste enthält die Schaltfläche und die Texteingabe. In Zeile 7 wird dazu das `ActionEvent`-Objekt nach der erzeugenden Komponente (`getComponent()`), die Komponente nach ihrem Container (`getParent()`) und der Container wiederum nach seinen enthaltenen Komponenten (`getChildren()`) gefragt. Danach wird durch die Liste iteriert und die Komponente mit der Id `textarea` gefunden, die dann der Variablen `text` zugewiesen wird. An den aktuellen Wert der Texteingabekomponente hängen wir in Zeile 14 die String-Konstante an. Die Klasse `ActionListener2` ist identisch mit der Klasse `ActionListener1` – mit dem Unterschied, dass im String die 1 durch die 2 ersetzt wird. Abbildung 4.20 zeigt die Seite im Browser, und zwar nach einmaligem und zweimaligem Betätigen der Schaltfläche. Wie man erkennen kann, funktioniert unser Programm tatsächlich. Die Ausführung der registrierten Action-Listener erfolgt immer in derselben – wie in der JSF-Seite angegebenen – Reihenfolge.

Aufgabe 4.7

Schreiben Sie eine Action-Listener-Methode, die die Beschriftung einer Schaltfläche ändert. Die Beschriftung „`einschalten`“ soll nach Betätigung der Schaltfläche auf „`ausschalten`“ geändert werden und umgekehrt.

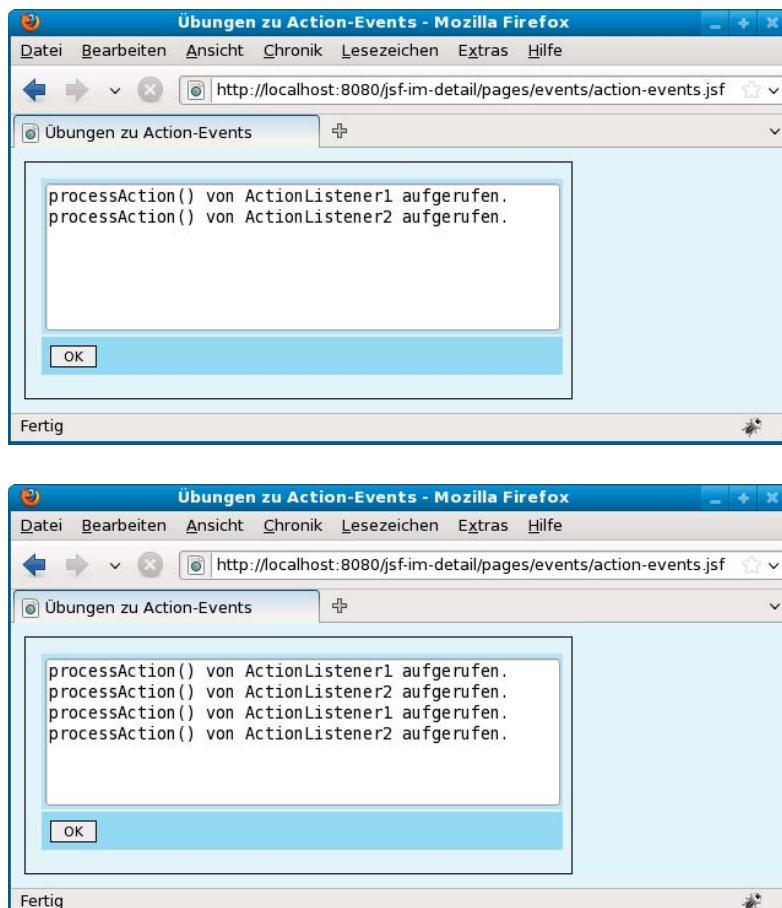


Abbildung 4.20: Verwendung von zwei Action-Listenern

Aufgabe 4.8

In Listing 4.20 musste über die Komponenten iteriert werden, um die Komponente mit der richtigen Id zu finden. Eine Alternative ist die Verwendung der Methode `UIComponent.findComponent(String)`. Implementieren Sie diese Alternative.

4.5.3 Befehlskomponenten und das `immediate`-Attribut

Bei der Darstellung des Bearbeitungsmodells einer JSF-Anfrage haben wir in Abschnitt 4.1 gesehen, dass Action- und Action-Listener-Methoden für Komponenten mit gesetztem `immediate`-Attribut bereits in der Phase 2 *Übernahme der Anfragewerte* und nicht erst in der Phase 5 *Aufruf der Anwendungslogik*

aufgerufen werden. Die Verarbeitung von Action-Events wird also vorgezogen. Was könnte hierfür ein geeigneter Anwendungsfall sein?

Bei der Eingabe von Daten kann zwischen obligatorisch und optional anzugebenden Daten unterschieden werden. Eingabekomponenten für obligatorische Eingaben werden mit einem `required="true"` oder `<f:validateRequired>` versehen. Damit wird aber der Aufruf einer Action-Methode für das gesamte Formular verhindert, wenn nicht alle obligatorischen Eingaben erfolgt sind, etwa auch eine Action-Methode für das Abbrechen des Vorgangs. Der Grund ist, dass in der Phase 3 *Validierung* die Überprüfung der benötigten Eingabe fehlschlägt und somit die Phase 5 gar nicht aufgerufen wird. Abhilfe schafft hier das `immediate`-Attribut für Befehlskomponenten. Die Phase 5 wird für diese Komponenten vorgezogen und in der Phase 2 *Übernahme der Anfragewerte* durchgeführt. Im folgenden Code-Ausschnitt wird die Action-Methode `abbrechen()` auch aufgerufen, wenn die obligatorischen Daten nicht eingegeben wurden. Somit ist die Möglichkeit der Navigation zu einer anderen Seite gegeben.

```
<h:panelGrid columns="2">
    <f:facet name="header">
        Verwendung des immediate-Attributs
    </f:facet>
    <h:outputLabel value="Obligatorische Daten:" for="data" />
    <h:inputText id="data" value="#{iaHandler.data}"
        required="true" />
    <h:panelGroup>
        <h:commandButton action="#{iaHandler.speichern}"
            value="Speichern" />
        <h:commandButton action="#{iaHandler.abbrechen}"
            value="Abbrechen" immediate="true"/>
    </h:panelGroup>
</h:panelGrid>
```

4.5.4 Value-Change-Events

Alle JSF-Eingabekomponenten erzeugen ein Value-Change-Event, wenn ein neuer Wert eingegeben wurde. Da wir uns in einer Web-Anwendung bewegen, bedeutet *neu* immer „von einem HTTP-Request zum nächsten“, wenn man Ajax zunächst nicht betrachtet. Aber auch diese Form von Werteänderung, die im Gegensatz etwa zu Swing sehr eingeschränkt ist, kann durchaus nützlich sein. In der Comedian-Anwendung können etwa die Stammdaten eines Comedians geändert und durch Betätigung eines Command-Buttons in die Datenbank geschrieben werden. Dies ist nicht nötig, sofern in keinem der Eingabefelder eine Änderung vorgenommen wurde, und erzeugt daher eine un-

nötige Belastung der Datenbank. Auch wenn dies in der Comedian-Anwendung dank des *automatic Dirty-Checkings* von JPA nicht wirklich ein Problem ist, kann dieser Anwendungsfall im Allgemeinen vorkommen. Mit Hilfe von Value-Change-Listenern, die an Eingabekomponenten gebunden sind, lässt sich sehr einfach feststellen, ob entsprechende Änderungen erfolgt sind. Die Action-Methode der Schaltfläche kann dieses Ergebnis dann verwenden, um nur im Falle tatsächlicher Änderungen eine Datenbanktransaktion oder anderweitige Aktionen zu veranlassen.

Auch bei Value-Change-Events gibt es mehrere Methoden, um Listener zu registrieren. Die erste Alternative ist die Registrierung eines Value-Change-Listeners als Methodenbindung, die zweite ist die Implementierung einer Listener-Klasse, also einer Klasse, die das Interface `ValueChangeListener` mit der Methode

```
public void processValueChange(
    javax.faces.event.ValueChangeEvent event)
throws AbortProcessingException
```

implementiert. Wir beginnen auch hier mit der einfachen Variante einer Methodenbindung.

Im folgenden Code-Ausschnitt wird die Listener-Methode `textChanged()` der Bean `vceHandler` an eine Texteingabekomponente gebunden. Ziel ist eine Anwendung, die eine Änderung der Eingabe bemerkt und dies in der Ausgabe anzeigt.

```
<h:inputText value="#{vceHandler.text}"
    valueChangeListener="#{vceHandler.textChanged}" />
<h:commandButton value="OK" action="#{vceHandler.ok}" />
<h:outputText value="#{vceHandler.ausgabe}" />
```

Die Texteingabe wird an die Bean-Property `text`, die Textausgabe an die Bean-Property `ausgabe` gebunden. Mit einer Methodenbindung wird die Schaltfläche an die Methode `ok` gebunden. Listing 4.21 zeigt die Klasse `ValueChangeEventHandler`, deren Instanz die Managed Bean `vceHandler` ist.

Listing 4.21: Die Klasse `ValueChangeEventHandler`

```
@ManagedBean(name = "vceHandler")
@SessionScoped
public class ValueChangeEventHandler {

    private String text = "";
    private String ausgabe = "";
    private int reqNb = 0;
```

```
private int reqNbChanged = 0;

public void textChanged(ValueChangeEvent vce) {
    ausgabe = "Text hat sich geändert. "
        + "War: '" + vce.getOldValue() + "'."
        + "Ist: '" + vce.getNewValue() + "'.";
    reqNbChanged = reqNb;
}

public String ok() {
    if (reqNb != reqNbChanged)
        ausgabe = "";
    reqNb++;
    return "success";
}
```

Da JavaServer Faces den Zustand von Komponenten speichern, führt der erstmalige Aufruf des Event-Listeners zwar zum gewollten Ergebnis, und in der Ausgabe erscheint die entsprechende Meldung. Diese ist aber auch in allen weiteren Darstellungen der Seite vorhanden. Eine JSF-Anfrage, die keinen Value-Change-Event enthält, muss die Meldung also wieder entfernen. Wir verwenden dazu zwei Zähler: `reqNb` zählt alle Aufrufe der Seite innerhalb einer Session mit, da die Managed Bean Session-Scope hat. In `reqNbChanged` steht die Nummer des Aufrufs, in dem das letzte Value-Change-Event geworfen wurde. Falls in der Action-Methode die beiden Zahlen nicht identisch sind, wird die Meldung zurückgesetzt.

Als Abschluss des Beispiels seien noch die beiden verwendeten Methoden der Klasse `ValueChangeEvent` erwähnt: `getOldValue()` gibt den alten, `getNewValue()` den aktuellen Wert der Eingabekomponente zurück.

Value-Change-Events werden von *allen* Eingabekomponenten erzeugt. Als weiteres Beispiel soll eine `<h:selectOneMenu>`-Komponente überwacht werden. Als Motivation nehmen wir wiederum an, dass eine komplette Anfragebearbeitung „teuer“ ist, weil in den Anwendungsmethoden umfangreiche Berechnungen oder Datenbankzugriffe vorgenommen werden. Diese sollen nur im tatsächlichen Fall einer Eingabeänderung stattfinden.

Wir realisieren die Berechnung der Funktion $n!$, also das Produkt der ersten n natürlichen Zahlen. Listing 4.22 zeigt einen Teil der JSF-Seite zur Berechnung der Fakultätsfunktion.

Listing 4.22: JSF-Code zur Fakultätsberechnung

```
<h:panelGrid columns="2" styleClass="borderTable">
    <h:panelGrid columns="1">
        <h:selectOneMenu style="width: 100%" value="#{fakHandler.n}"
            valueChangeListener="#{fakHandler.nChanged}">
            <f:selectItems value="#{fakHandler.arguments}" />
        </h:selectOneMenu>
        <h:commandButton value="Berechne n!" />
    </h:panelGrid>
    <h:panelGrid columns="1">
        <h:inputTextarea cols="30" rows="1"
            value="#{fakHandler.fakultaet}" />
    </h:panelGrid>
</h:panelGrid>
```

Die Komponente `<h:selectOneMenu>` ist über eine Wertebindung an die Bean-Property `n` gebunden. Das Ergebnis wird in der `<h:inputTextarea>` dargestellt, die über eine Wertebindung an die Property `fakultaet` gebunden ist. Die Listener-Methode `nChanged()` wird bei einer Änderung des Menüs aufgerufen. Als Menü wird die Liste natürlicher Zahlen dargestellt, die die Methode `arguments` als Liste von `SelectItems` zurückliefert.

Listing 4.23: Die Managed Bean FakultaetHandler

```
@ManagedBean(name = "fakHandler")
@SessionScoped
public class FakultaetHandler {

    private static final int N_MAX = 1000;
    private int n;
    private BigInteger fakultaet;

    public void nChanged(ValueChangeEvent vce) {
        fakultaet = Fakultaet.fakultaet(
            new BigInteger(vce.getNewValue().toString()));
    }
    /**
     * Liste von SelectItems potentieller Argumente
     */
    public List getArguments() {
        List l = new ArrayList();
        for (int i = 0; i < N_MAX; i++)
            l.add(new SelectItem(i, String.valueOf(i)));
        return l;
    }
}
```

Die Klasse **FakultaetHandler** der Managed Bean **fakHandler** zeigt Listing 4.23. Damit auch große Fakultäten berechenbar sind, verwenden wir als Datentyp **BigInteger**. Der Event-Listener weist dem Property **fakultaet** das Ergebnis der Fakultätsberechnung zu. Auf die Darstellung der Klasse **Fakultaet** verzichten wir und verweisen auf das herunterladbare Projekt. Abbildung 4.21 gibt die Darstellung der Seite im Browser wieder, und zwar für die Berechnung der Fakultät von 999. Bitte beachten Sie die geringe Größe des Schiebereglers der Text-Area. Bei wiederholter Betätigung der Schaltfläche ist die Seite innerhalb kürzester Zeit wieder dargestellt. Wird ein neuer Wert ausgesucht und dann die Schaltfläche betätigt, ist eine deutlich längere Antwortzeit zu beobachten. Unser Ziel ist also erreicht.

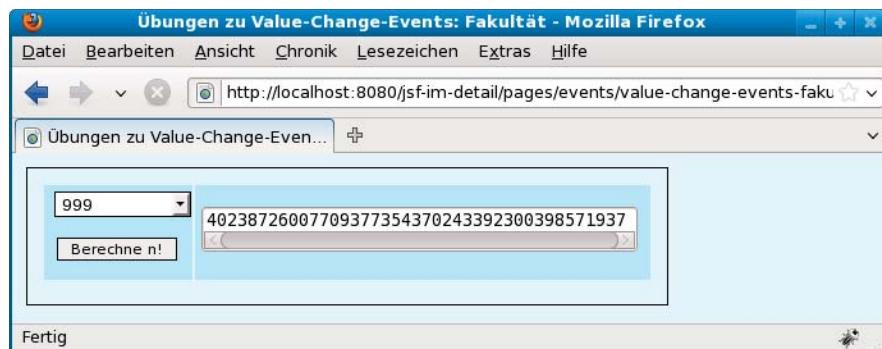


Abbildung 4.21: Berechnung der Fakultät von 999

Bei Value-Change-Events existiert dasselbe Problem wie bei Action-Events. Weil XML-Tags die mehrfache Verwendung ein und desselben Attributs nicht erlauben, ist nur die Registrierung von maximal einem Event-Listener möglich. Abhilfe schafft hier die mehrfache Verwendung des Tags `<f:valueChangeListener>` innerhalb der Eingabekomponente:

```
<h:inputText value="...">
    <f:valueChangeListener type="de.jsfpraxis.MyListener1" />
    <f:valueChangeListener type="de.jsfpraxis.MyListener2" />
</h:inputText>
```

Die beiden Klassen `MyListener1` und `MyListener1` müssen dabei das Interface `ValueChangeListener` mit der Methode

```
public void processValueChange(
        javax.faces.event.ValueChangeEvent event)
    throws AbortProcessingException
```

implementieren. Neben der angesprochenen Notwendigkeit bei der Verwendung von mehr als einem Listener erhöht die Erstellung einer eigenen Listener-Klasse die Möglichkeiten der Wiederverwendung im selben, aber auch in anderen Projekten, so dass diese Option zu überlegen ist. Wir verzichten hier auf ein ausprogrammiertes Beispiel.

Aufgabe 4.9

Lösen Sie mit Hilfe eines Value-Change-Listeners folgende Aufgabe: Wenn eine Benutzereingabe in ein `<h:inputText>` erfolgt ist, darf keine weitere ändernde Eingabe mehr erfolgen. Tipp: Die Java-seitige Komponente ist ein `HtmlInputText`. Diese kennt die Methode `setReadOnly(boolean)`.

Aufgabe 4.10

Realisieren Sie die Aufgabe zur Texteingabeänderung ohne die Zähler in der Managed Bean. Tipp: Verwenden Sie eine zusätzliche Managed Bean mit Request-Scope.

4.5.5 Data-Model-Events

Eine in betrieblichen Informationssystemen häufig anzutreffende Art der Informationsdarstellung ist die mittels Listen und Tabellen. JavaServer Faces verwendet hierzu auf der Komponentenebene als Modellobjekte Instanzen der Klasse `javax.faces.model.DataModel`, die im Standard-Render-Kit mit einer HTML-Tabelle dargestellt werden. Die Interaktion des Benutzers mit einer der Zeilen dieser Tabelle wird über Data-Model-Events realisiert. Ein Event der Klasse `javax.faces.model.DataModelEvent` wird durch ein `DataModel`-Objekt geworfen, wenn ein Zugriff auf eine Zeile des Objektes erfolgt. Damit unterscheiden sich Data-Model-Events von Action- und Value-Change-Events, die durch UI-Komponenten geworfen werden. Weil `DataModel`-Objekte nicht in JSF-Tags vorkommen, kann auch die Registrierung eines Listeners für `DataModelEvents` nicht in der JSF-Seite erfolgen, sondern muss im Java-Code vorgenommen werden. `DataModelEvents` kennen die folgenden Methoden

- `public DataModel getDataModel()`
- `public Object getRowData()`
- `public int getRowIndex()`

die das gesamte Model-Objekt, die Daten oder den Index der betroffenen Zeile zurückgeben.

Ein zu implementierender Listener für `DataModelEvents` muss das Interface `javax.faces.model.DataModelEvent` implementieren, das aus der einzigen Methode

```
public void rowSelected(javax.faces.model.DataModelEvent event)
    throws AbortProcessingException
```

besteht. Die Implementierung einer eigenen Listener-Klasse für DataModel-Events ist in der Regel nicht notwendig. Wir verzichten an dieser Stelle auf ein Beispiel, da wir bereits in der Comedian-Anwendung in Abschnitt 3.2 ein derartiges Beispiel, basierend auf der Methode `getRowData()`, entwickelt haben.

Der Vollständigkeit des Abschnitts halber sei noch erwähnt, dass die Klasse `DataModel` abstrakt ist. Nicht abstrakte Unterklassen, die als Datenmodelle verwendet werden können, sind `ArrayDataModel`, `ListDataModel`, `ResultDataModel`, `ResultSetDataModel` und `ScalarDataModel`, die alle im Package `javax.faces.model` definiert sind. Mit ihnen können als Daten Arrays, Listen (`java.util.List`), SQL-Ergebnismengen (`javax.servlet.jsp.jstl.sql.Result` als JSTL-Result und `java.sql.ResultSet` als JDBC-Result) sowie einfache Java-Objekte (`java.lang.Object`) verwendet werden.

4.5.6 Phase-Events

In Abschnitt 4.1 haben wir die einzelnen Phasen der Bearbeitung einer JSF-Anfrage kennengelernt. In jeder dieser Phasen wird zu Beginn und am Ende ein `PhaseEvent` geworfen. Auch für diese Event-Art existiert ein entsprechendes Listener-Interface, das `PhaseListener`-Interface im selben Package. Das Interface deklariert die Methoden

```
public javax.faces.event.PhaseId getPhaseId()

public void beforePhase(javax.faces.event.PhaseEvent event)

public void afterPhase(javax.faces.event.PhaseEvent event)
```

Die Methode `getPhaseId()` liefert eines von sieben konstanten Objekten (nach dem Enumeration-Muster) der Klasse `PhaseId`. Es sind dies die sechs Repräsentanten der Bearbeitungsphasen sowie ein Repräsentant für „alle Phasen“. Die Bezeichner sind

- `PhaseId.ANY_PHASE`
- `PhaseId.RESTORE_VIEW`
- `PhaseId.APPLY_REQUEST_VALUES`
- `PhaseId.PROCESS_VALIDATIONS`
- `PhaseId.UPDATE_MODEL_VALUES`
- `PhaseId.INVOKE_APPLICATION`
- `PhaseId.RENDER_RESPONSE`

Bei der Entwicklung eines eigenen Phase-Listeners muss die Methode `getPhaseId()` eine dieser sieben Konstanten zurückgeben und zeigt somit, an welcher Phase des Bearbeitungszyklus der Listener interessiert ist. Die beiden Methoden `beforePhase()` und `afterPhase()` können anwendungsbezogen realisiert werden.

Bei der Erstellung eines Informationssystems wird es in der Regel wenig Verwendungsmöglichkeiten für Phase-Events geben. Phase-Events sind aber prädestiniert, bei der Erweiterung des JSF-Frameworks eingesetzt zu werden. Eine Möglichkeit ist etwa eine bestimmte Art der Authentifizierung. Gleich zu Beginn der Bearbeitung einer JSF-Anfrage könnte überprüft werden, ob der Benutzer (bzw. die Session), der die JSF-Seite anfragt, sich authentifiziert hat und die entsprechenden Rechte für den Zugriff auf diese Seite besitzt. Diese Form der Authentifizierung gilt dann aber nur für JSF-Seiten, so dass GIFs, JPEGs oder PDF-Dokumente, deren URL bekannt ist, direkt ohne Authentifizierung angefordert werden können. Auch ohne dieses Manko ist diese Authentifizierungs- und Autorisierungsprüfung jedoch nicht zu empfehlen; man sollte auf ausgereifte Frameworks wie JBoss Seam oder Acegi zurückgreifen.

Aufgabe 4.11

Eine sehr sinnvolle Verwendungsmöglichkeit für Phase-Listener ist die Unterstützung des Erlernens von JSF. Schreiben Sie einen Phase-Listener, der die Funktionsweise des `immediate`-Attributs wie in Abschnitt 4.5.3 beschrieben validiert. Man sollte also erkennen können, dass das Betätigen der Abbrechen-Schaltfläche von Phase 2 zu Phase 6 führt, während das Betätigen der Speichern-Schaltfläche zu einem vollständigen Durchlaufen des Lebenszyklus bzw. zur Phasenfolge 2, 3, 6 führt, je nachdem, ob eine Eingabe vorhanden ist oder nicht.

Als Beispiel für die Verwendung von Phase-Events wollen wir lediglich kleine Status-Nachrichten in das Log-File des Containers schreiben. Listing 4.24 zeigt einen sehr einfachen Phase-Listener, dessen Phasen-Methoden in jeder der sechs Phasen aufgerufen werden.

Listing 4.24: Ein einfacher Phase-Listener

```
public class MyPhaseListener implements PhaseListener {  
  
    private static Log log =  
        LogFactory.getLog(MyPhaseListener.class);  
  
    public PhaseId getPhaseId() {
```

```
// wir sind an allen Phasen interessiert
return PhaseId.ANY_PHASE;
}

public void afterPhase(PhaseEvent pe) {
    log.info("After Phase : " + pe.getPhaseId());
}

public void beforePhase(PhaseEvent pe) {
    log.info("Before Phase : " + pe.getPhaseId());
}
}
```

Registriert wird ein Phase-Listener global für die gesamte Anwendung in der Konfigurationsdatei `faces-config.xml`. Der entsprechende Eintrag ist ein `<phase-listener>`-Element, das im `<lifecycle>`-Element enthalten ist. Es gibt den qualifizierten Namen der Phase-Listener-Klasse an:

```
<lifecycle>
    <phase-listener>
        de.jsfpraxis.MyPhaseListener
    </phase-listener>
</lifecycle>
```

Für eine einzelne Seite kann ein Phase-Listener mit dem `<f:phaseListener>`-Tag registriert werden:

```
<f:phaseListener type="de.jsfpraxis.MyPhaseListener" />
```

4.5.7 System-Events

Frameworks, die JSF erweitern oder auf JSF basieren, benötigen eine feinere Granularität des Event-Systems, als es Phase-Events bereitstellen. JSF 2.0 führt daher System-Events ein, die systemspezifische Ereignisse anzeigen. Dabei wird zwischen einer globalen Systemebene und der Komponentenebene unterschieden. Da wir den Anwendungsentwickler und nicht den Framework-Entwickler als Leser im Fokus haben, stellen wir System-Events nur übersichtsartig vor.

Die in Abbildung 4.19 auf Seite 116 dargestellte Klasse `SystemEvent` ist die Oberklasse der System-Events. Sie enthält als direkte Unterklassen die folgenden Events:

- `ComponentSystemEvent`
- `ExceptionQueuedEvent`

- `PostConstructApplicationEvent`
- `PostConstructCustomScopeEvent`
- `PreDestroyApplicationEvent`
- `PreDestroyCustomScopeEvent`

Die Klasse `ComponentSystemEvent` dient als abstrakte Oberklasse für die folgenden Komponenten-Events.

- `PostAddToViewEvent`
- `PostConstructViewMapEvent`
- `PostRestoreStateEvent`
- `PostValidateEvent`
- `PreDestroyViewMapEvent`
- `PreRemoveFromViewEvent`
- `PreRenderComponentEvent`
- `PreRenderViewEvent`
- `PreValidateEvent`

Wir beginnen mit einem Beispiel für System-Events. Das Event `PostConstructApplicationEvent` wird geworfen, nachdem das JSF-Laufzeitsystem alle Konfigurationsdaten gelesen und verarbeitet hat. Analog dazu wird das Event `PreDestroyApplicationEvent` geworfen, bevor alle Fabriken, die der Anwendung zugeordnet sind, freigegeben werden. Wenn die Anwendung nach der Initialisierung der Anwendungskonfiguration bestimmte Funktionen durchzuführen hat, wird ein Listener für das `PostConstructApplicationEvent` registriert. Dies erfolgt in der JSF-Konfigurationsdatei innerhalb des `<application>`-Elements.

```
<application>
    <system-event-listener>
        <system-event-class>
            javax.faces.event.PostConstructApplicationEvent
        </system-event-class>
        <system-event-listener-class>
            de.jsfpraxis.detail.events.MySystemEventListener
        </system-event-listener-class>
    </system-event-listener>
</application>
```

Der Listener muss das Interface `SystemEventListener` implementieren, das aus zwei Methoden besteht:

```
public boolean isListenerForSource(Object object)  
  
public void processEvent(SystemEvent se)
```

Als Beispiel zur Verwendung von Komponenten-Events wählen wir das Rendern der View bzw. einer Output-Komponente. Die entsprechenden Klassen sind `PreRenderViewEvent` und `PreRenderComponentEvent`. Die Registrierung eines Listeners erfolgt mit dem `<f:event>`-Tag. Dessen `type`-Attribut enthält den voll qualifizierten Klassennamen des Events oder eine Kurzform des Namens. Diese ergibt sich aus den nicht qualifizierten Klassennamen mit – sofern vorhanden – abgeschnittenem „Event“ sowie kleingeschriebenem Anfangsbuchstaben.

```
<f:view>  
    <f:event type="javax.faces.event.PreRenderViewEvent"  
              listener="#{systemEventHandler.handleEvent}" />  
    <f:event type="preRenderView"  
              listener="#{systemEventHandler.handleEvent}" />  
    ...  
    <h:outputText value="Ausgabe">  
        <f:event type="preRenderComponent"  
                  listener="#{systemEventHandler.beforeEncode}" />  
    </h:outputText>  
    ...
```

Im obigen Beispiel wird also eine doppelte Registrierung auf das Event `preRenderView` vorgenommen. Die Listener-Methoden erhalten als Parameter ein `ComponentSystemEvent`:

```
@ManagedBean  
public class SystemEventHandler {  
  
    public void handleEvent(ComponentSystemEvent event) { ... }  
  
    public void beforeEncode(ComponentSystemEvent event) { ... }  
    ...
```

Wir beschließen diesen Abschnitt mit der Nennung der alternativ zu verwendenden Annotationen. Statt des Eintrags in der JSF-Konfigurationsdatei kann ein System-Event-Listener mit

```
@ListenerFor(systemEventClass = <event-class>)
```

bzw. – falls er für mehrere Events registriert werden soll – mit `@ListenersFor` annotiert werden. Falls eigene Event-Klassen als Unterklassen von `ComponentSystemEvent` definiert werden, sind diese mit `@NamedEvent` zu annotieren, um sie als Komponenten-Events im `<f:event>`-Tag verwenden zu können.

Die hier sehr oberflächliche Einführung von System-Events wird in den nachfolgenden Kapiteln vervollständigt. In Abschnitt 8.3 realisieren wir auf Basis des `PreRenderViewEvent` eine einfache Überprüfung, ob der Benutzer eingeloggt ist. In Abschnitt 9.4 verwenden wir dasselbe Event, um eine Benutzerkonversation zu realisieren.



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für die Event-Verarbeitung sind im Projekt `jsf-im-detail` enthalten.

4.6 Navigation

JavaServer Faces sind im Rahmen von Java-EE-Anwendungen als Benutzerschnittstellen-Framework für die Verwendung in Unternehmensanwendungen vorgesehen. Unternehmensanwendungen unterscheiden sich erheblich von den einfach verlinkten Web-Seiten des frühen World Wide Web. Neben der eigentlichen Anwendungslogik ist hier vor allem die Abkehr von „verlinkten“ Web-Seiten zu nennen. JSF-Anwendungen besitzen in der Regel keinen wesentlichen Anteil an Verlinkungen dieser Art. Eine JSF-Seite wird nach dem Auslösen eines Action-Events und der server-seitigen Verarbeitung im Default-Fall nochmals dargestellt. Dies ist in Unternehmensanwendungen häufig sinnvoll, denken wir nur an die Stammdatenverwaltung. Andererseits muss aber eine Navigationskomponente vorhanden sein, die *anwendungsbezogene* Workflows unterstützen sollte.

JavaServer Faces verfügen über eine eigenständige Navigationskomponente, den *Navigation-Handler*. Technisch gesehen, reagiert die Navigationskomponente auf in der Regel durch Benutzerinteraktionen ausgelöste Action-Events. Die Navigation ist entweder an eine Action-Methode gebunden und wird durch das logische Ergebnis der Methodenausführung gesteuert oder durch einen hart codierten Wert in der JSF-Seite. Die Basis für die Navigation ist eine Menge von Regeln in XML, in der alle möglichen Navigationspfade hinterlegt sind. Eine solche Regel definiert, wie man von einer Seite (oder mehreren) auf eine andere gelangt. Mit JSF 2.0 kamen implizite Regeln hinzu, die direkt String-Literale als View-Ids interpretieren. Da dies für JSF-Anfänger sicher die einfachste Alternative darstellt, beginnen wir unsere Darstellung der JSF-Navigationsmöglichkeiten mit dieser Alternative.

4.6.1 Implizite Navigation

Unter impliziter Navigation versteht man die direkte Angabe der View-Id, zu der navigiert werden soll, entweder als String-Literal im JSF-Tag oder als Rückgabewert der Action-Methode. Bei der Angabe im JSF-Tag wird das Attribut `action` bei `<h:commandButton>` und `<h:commandLink>` bzw. das Attribut `outcome` bei `<h:button>` und `<h:link>` verwendet.

Bei der direkten Angabe der View-Id des Navigationsziels reduziert sich der JSF-Code auf lediglich:

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

Wird eine Action-Methode verwendet, so ist bei Verwendung der Methode `action` in der JSF-Seite

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

auf der Java-Seite die Methode mit dem entsprechenden Rückgabewert zu versehen.

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

Die Angabe der Dateinamenserweiterung ist optional. Ist keine Dateinamenserweiterung vorhanden, so wird die der aktuellen View verwendet und an die Ziel-View-Id angehängt. Im obigen Beispiel hätte also auch

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

oder

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

verwendet werden können. Entgegen den Angaben in der Spezifikation konnten wir bei unseren Tests auch andere Dateinamenserweiterungen verwenden, etwa `"ziel.jsf"` oder `"ziel.irgendwas"`. Wir raten dem Leser, dies nicht zu tun, sondern konsequent die View-Id zu verwenden, alternativ deren Kurzform ohne Dateinamenserweiterung.

Die Verwendung der impliziten Navigation ist für den JSF-Anfänger und in kleinen Projekten sicher angezeigt. JSF besitzt aber sehr viel mächtigere Möglichkeiten der Definition anwendungsbezogener Workflows, die durch ein System von Navigationsregeln in der JSF-Konfigurationsdatei realisiert werden.

Wir stellen diese Regeln in den folgenden Abschnitten vor. Zum Abschluss des Abschnitts zur impliziten Navigation sei noch angemerkt, dass Navigationsregeln in der JSF-Konfigurationsdatei die impliziten Navigationsregeln überschreiben, so dass der Leser beim Debuggen eines augenscheinlich falschen Navigationsverhaltens dies beachten sollte.



Navigationsregeln in der JSF-Konfigurationsdatei überschreiben Regeln zur impliziten Navigation. Verhält sich das System nicht so, wie es sich auf Basis der impliziten Navigation verhalten sollte, so sind die expliziten Navigationsregeln in der JSF-Konfigurationsdatei zu überprüfen.

4.6.2 View-to-View-Regeln

Eine JSF-Seite beschreibt die Struktur und das Layout einer View. Durch die eigenständige Navigationskomponente können Anwendungen mit einer Menge von Views definiert werden, deren Navigation völlig unabhängig von den JSF-Seiten ist. Insbesondere müssen z. B. bei Änderungen der Seitenabfolgen oder der Wiederverwendung von Seiten in einer anderen Anwendung keine Änderungen an den JSF-Seiten vorgenommen werden. Dies erhöht die Wartbarkeit der Anwendung erheblich, da die Navigation aufgrund deklarativer Navigationsregeln und logischer Ergebniswerte der Action-Methoden erfolgt und nicht hart in den Seiten codiert ist.

Die folgende Syntax zeigt die Struktur des Teils der Konfigurationsdatei, der für die Navigation zuständig ist. Wir beginnen mit der Regelbeschreibung für *View-to-View-Regeln*, gehen dann auf Regeln für mehrere Seiten ein und schließen mit Verweisen auf Nicht-JSF-Seiten und Redirects.

```
<navigation-rule>
    <from-view-id>
        <navigation-case>*
            <from-action>?
            <from-outcome>?
            <if>?
            <to-view-id>
            <redirect>?
```

Navigationsregeln können einzelne Views benennen oder mit Hilfe des ***-Zeichens für mehrere Views gelten. Grundlage ist die eindeutige Bezeichnung einer View. Eine *View-Id* ist der Kontext-relative Pfad zu einer JSF-Seite. Die View-Id kann als Quelle und Ziel in einer Navigationsregel verwendet werden.

Eine Navigationsregel wird durch ein `<navigation-rule>`-Element definiert. Im `<from-view-id>`-Element wird die Ursprungsseite, für die die Regel definiert wird, bestimmt. Als Nächstes erfolgt dann mit meist mehreren `<navigation-case>`-Elementen die Angabe, unter welchen Bedingungen welche Zielseiten anzusteuern sind. Die Angabe des Ziels dieses Navigationsfalls wird obligatorisch im `<to-view-id>`-Element angegeben. Alle weiteren Elemente sind optional. Mit dem `<from-outcome>`-Element wird der einfachste Fall definiert, bei dem das Ergebnis einer Action-Methode Verwendung findet. Dies kann zum einen der tatsächliche Rückgabewert der Action-Methode sein, aber auch innerhalb des `<h:commandButton>`-Tags eine String-Konstante für das `action`-Attribut. Listing 4.25 zeigt eine Navigationsregel, die von der Seite `hauptseite.xhtml` abhängig vom Wert einer Action-Methode zu den Seiten `rot.xhtml`, `gelb.xhtml` und `blau.xhtml` führt.

Listing 4.25: Beispiel für Navigationsregeln

```
<navigation-rule>
  <description>
    Beispiel View-to-View-Regeln
  </description>
  <from-view-id>/pages/hauptseite.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>rot</from-outcome>
    <to-view-id>/pages/nav/rot.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>gelb</from-outcome>
    <to-view-id>/pages/nav/gelb.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>blau</from-outcome>
    <to-view-id>/pages/nav/blau.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Der Quell-Code der JSF-Seite, die die Navigationsregeln verwendet, sieht folgendermaßen aus:

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

Hier wird an einen Command-Button die Action-Methode `verteiler()` gebunden. Diese entscheidet anwendungslogisch über die weitere Navigation. Die Logik greift auf die Eingabe des Input-Texts zurück, was allerdings aus der JSF-Seite nicht hervorgeht. Die Methode `verteiler()` verwendet dazu das Property `eingabe`:

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

Die Methode macht eine Fallunterscheidung bezüglich des Property `eingabe`. Für die Eingaben `rot`, `gelb`, `blau` wird jeweils "`rot`", "`gelb`" und "`blau`" zurückgegeben. Bei anderen Eingaben wird in das `eingabe`-Property der Text "`bitte nochmal`" geschrieben und "`error`" zurückgegeben. Für dieses Action-Ergebnis gibt es keinen Navigation-Case, so dass die Default-Regel, die erneute Anzeige der aktuellen Seite, angewendet und eine Fehlermeldung erzeugt wird. Bis auf das Erkennen einer Falscheingabe hätte alternativ also auch

```
public String verteiler() {  
    return eingabe;  
}
```

geschrieben werden können. Die drei JSF-Seiten sind identisch aufgebaut, verwenden jedoch jeweils eine andere Hintergrundfarbe. Die Seite `rot.xhtml` ist in Listing 4.26 dargestellt.

Listing 4.26: Die rote Seite mit konstantem action-Attribut (rot.xhtml)

```
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://java.sun.com/jsf/html"  
      xmlns:f="http://java.sun.com/jsf/core">  
<head>  
    <title>Übungen zur Navigation: Rote Seite</title>  
</head>  
<body style="background-color: red;">  
  <f:view>  
    <h:form id="form">  
      <h:commandButton value="zurück" action="zurück" />
```

```
</h:form>
</f:view>
</body>
</html>
```

Hier wird als Wert des **action**-Attributs keine Methodenbindung verwendet, sondern eine String-Konstante, die analog zum Rückgabewert einer Action-Methode in der Navigationsregel Verwendung findet. Da eine entsprechende Navigationsregel in der JSF-Konfigurationsdatei existiert, wird keine implizite Navigation verwendet.

4.6.3 Regeln für mehrere Seiten

Das obige Beispiel verzweigt von der Hauptseite entsprechend der Benutzereingabe zu den drei Seiten mit verschiedenen Hintergrundfarben. Die Rückkehr von diesen Seiten zur Hauptseite müsste mit drei Navigationsregeln mit einem jeweils anderen **<from-view-id>**-Element erfolgen. Dies kann jedoch erheblich vereinfacht werden, da Navigationsregeln im **<from-view-id>**-Element Wildcards zulassen, die mehrere Seiten zu *einer* für die Navigation logischen View zusammenfassen. Listing 4.27 zeigt die Navigationsregel von den drei (oder mehr) Farb-Seiten zurück zur Hauptseite.

Listing 4.27: Navigationsregel mit Wildcard für mehrere Seiten

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

Dem aufmerksamen Leser wird nicht entgangen sein, dass das Muster des **<from-view-id>**-Elements die View-Id des Beispiels in Listing 4.25 auf Seite 136 umfasst. Überlappungen zwischen den View-Ids verschiedener Navigationsregeln sind zulässig und werden über deren Reihenfolge aufgelöst. Die Navigationskomponente durchsucht alle Navigationsregeln und nimmt die erste passende. Für das Beispiel muss die zur Hauptseite zurückführende Regel also nach der Regel für die Hauptseite stehen.

4.6.4 Regeln für Action-Methoden

Ein weiteres mögliches Element innerhalb eines `<navigation-case>` ist das `<from-action>`-Element. Das Element erwartet eine Action-Methode in Form eines EL-Ausdrucks. Die Navigationsregel wird ausgeführt, wenn die entsprechende Action-Methode aufgerufen wird. Die folgende Regel wird ausgeführt, wenn die Action-Methode `rot()` aufgerufen wird.

```
<from-view-id>/pages/nav/hauptseite.xhtml</from-view-id>
<navigation-case>
    <from-action>#{naviHandler.rot}</from-action>
    <to-view-id>/pages/nav/ganzrot.xhtml</to-view-id>
</navigation-case>
```

4.6.5 Regeln zur bedingten Navigation

Das `<if>`-Element wurde mit JSF 2.0 eingeführt und ermöglicht eine bedingte Navigation. Damit ist es möglich, über den Zustand einer Managed Bean die Navigation direkt zu beeinflussen. Als Text des Elements sind boolesche EL-Ausdrücke erlaubt, also etwa ein boolesches Property einer Managed Bean. Wir wählen als Beispiel kein boolesches, sondern ein Integer-Property. Abhängig vom Wert dieses Properties kann nun eine Fallunterscheidung getroffen werden.

```
<from-view-id>...</from-view-id>
<navigation-case>
    <from-outcome>bedingt</from-outcome>
    <if>#{naviHandler.wert lt 100}</if>
    <to-view-id>/pages/nav/bedingt1.xhtml</to-view-id>
</navigation-case>
<navigation-case>
    <from-outcome>bedingt</from-outcome>
    <if>#{naviHandler.wert ge 100}</if>
    <to-view-id>/pages/nav/bedingt2.xhtml</to-view-id>
</navigation-case>
```

In diesem Fall wird die erste Navigationsalternative gewählt, wenn das Action-Ergebnis `bedingt` ist und der Wert des Property `wert` < 100 bzw. die zweite Navigationsalternative, wenn der Wert ≥ 100 ist.

Doch wie funktioniert nun JSFs Navigation im Detail? Die einzelnen Navigationsfälle (`<navigation-case>`) einer Navigationsregel werden der Reihe nach ausgewertet, und zwar am Ende der Phase 5, Aufruf der Anwendungslogik. Die aufgerufene Action-Methode, deren Rückgabewert oder das String-Literal des JSF-Tags werden vom Navigations-Handler verwendet, um die Bedingungen der einzelnen Navigationsfälle zu überprüfen. Die Elemente `<from-action>`,

<from-outcome> und <if> sind optional, können einzeln, aber auch in Kombination verwendet werden. Beim ersten passenden Navigationsfall wird die View ausgewählt, die durch das <to-view-id>-Element festgelegt ist. Alle weiteren Fälle werden nicht mehr betrachtet. Die einzelnen Navigationsfälle müssen also sinnvoll im Rahmen der Anwendungslogik angeordnet werden. Ein Rückgabewert der Action-Methode von `null` führt zur nochmaligen Anzeige derselben Seite. Seit JSF 2.0 kann aber durch das neu eingeführte <if>-Element auch bei einem Rückgabewert von `null` navigiert werden, da das <if> unabhängig von der Action-Methode ist.

4.6.6 Redirects

Als letztes optionales Element eines <navigation-case> kann man das leere <redirect>-Element verwenden. Dadurch wird die JSF-Implementierung angewiesen, dem Browser eine Redirect-Antwort zu schicken. Diese Antwort enthält als URL die View-Id der neuen View, die der Browser dann automatisch in einer weiteren Anfrage verwendet.



Für den HTTP-kundigen Leser: JSF schickt einen Response-Code 302, *Moved Temporarily*, zurück.

Man muss sich darüber im Klaren sein, dass die Verwendung des <redirect> zwei Request/Response-Zyklen verursacht und somit mehr Zeit benötigt. Außerdem ist der Aufwand auf dem Server größer, weil zwei Anfragen anstatt einer zu beantworten sind.

Positiv ist allerdings hervorzuheben, dass die URL in der Adresszeile des Browsers tatsächlich der angezeigten View entspricht. Bei JavaServer Faces steht normalerweise in der Adresszeile immer die URL der View, an die die Anfrage geschickt wurde, und nicht die URL der angezeigten View. Dies ist in der Regel nicht kritisch, führt jedoch eventuell bei Benutzern, die mit der Browser-Zeile arbeiten, zu Verwirrung und Problemen, z.B. wenn die Seite erneut geladen oder als Lesezeichen abgelegt wird. Eine JSF-Anwendung sollte aus diesem Grund immer die komplette Navigation abbilden und den Benutzer auffordern, die Navigation durch den Browser zu unterlassen.

Als Beispiel verwenden wir einen Command-Button, dessen `action`-Attribut den Wert "redirect" hat.

```
<h:commandButton action="redirect" value="Ganz rot" />
```

Die entsprechende Fallunterscheidung in der Navigationsregel lautet dann:

```
<navigation-case>
    <from-outcome>redirect</from-outcome>
    <to-view-id>/pages/ganzrot.xhtml</to-view-id>
    <redirect />
</navigation-case>
```

Redirects können auch bei der impliziten Navigation verwendet werden. Dazu wird `faces-redirect` als Teil des HTTP-Query-Strings an die URL des Requests angehängt, und zwar mit dem Wert `true`. Mit dem Beispiel aus Abschnitt 4.6.1 also etwa

```
<h:commandButton value="..." 
    action="ziel.xhtml?faces-redirect=true"/>
```

Das Beispiel der impliziten Navigation in Java wird dann zu

```
public String action() {
    ...
    return "ziel.xhtml?faces-redirect=true";
}
```

4.6.7 Verweise auf Nicht-JSF-Seiten

In der Tabelle 4.1 auf Seite 36 erwähnten wir auch den Fall, dass aus einer JSF-Anfrage eine Nicht-JSF-Antwort erzeugt wird (Fall 3). Dies kann durch einen einfachen Verweis erfolgen, und zwar mit einem `<h:outputLink>`:

```
<h:outputLink value="http://www.jsfpraxis.de">
    <h:outputText value="Das Buch" />
</h:outputLink>
```

In diesem Fall wird die JSF-Navigationskomponente nicht eingebunden, weil der dafür erzeugte HTML-Code ein einfaches `<a>`-Element mit entsprechendem `href` ist.

Da JSF-Anwendungen häufig analog zu nativen Anwendungen erstellt werden, die nur Schaltflächen, aber keine Verweise kennen, kann eine solche Weiterleitung auf eine Nicht-JSF-Seite, aber auch mit einer Schaltfläche und der Einbindung der JSF-Navigationskomponente erfolgen. Man erhält damit die Flexibilität der normalen JSF-Navigation auch für externe Seiten. Die Schaltfläche

```
<h:commandButton action="das_buch" value="Das Buch" />
```

ist eine gewöhnliche Schaltfläche, wie wir sie in früheren Beispielen schon eingesetzt haben. Die Fallunterscheidung der Navigation enthält den Fall

```
<navigation-case>
    <from-outcome>das_buch</from-outcome>
```

```
<to-view-id>/pages/das-buch.xhtml</to-view-id>
</navigation-case>
```

Auch dies entspricht den bisherigen Beispielen. Der Unterschied muss also in der Seite `das-buch.xhtml` liegen. Der entsprechende HTML-Code lautet:

```
<html>
  <head>
    <meta http-equiv="Refresh"
          content="0; URL=http://www.jsfpraxis.de" />
  </head>
  <body>
  </body>
</html>
```

Hier wird das sogenannte Meta-Refresh von HTML verwendet, um eine Seite erneut bzw. optional eine andere Seite anzuzeigen. Obwohl veraltet (deprecated) funktioniert dies auf allen gängigen Browsern.

Zum Abschluss des Abschnitts sei noch erwähnt, dass man wie immer bei Java-Server Faces auch bei der Navigation direkt in Java arbeiten kann. Listing 4.28 zeigt eine Java-Methode, die als Action-Methode an ein `<h:commandButton>` gebunden werden kann. Die Methode holt sich ein Objekt der Klasse `ExternalContext`, die dem Kontext des zugrunde liegenden Containers (Servlet- oder Portlet-Container) entspricht. Mit diesem Objekt erfolgt ein Redirect auf die angegebene URL. Die beiden Rückgaben sind ohne Bedeutung für die Navigation und nur enthalten, weil sie die Signatur einer Action-Methode verlangt.

Listing 4.28: Action-Methode zur direkten Navigation

```
public String dasBuch() {
    FacesContext context = FacesContext.getCurrentInstance();
    ExternalContext ec = context.getExternalContext();
    try {
        ec.redirect("http://www.jsfpraxis.de");
    } catch (IOException e) {
        return "failure";
    }
    context.responseComplete();
    return "success";
}
```

Der Vollständigkeit halber sei hier erwähnt, dass mit der Integration von Ajax mit JSF 2.0 die Aktualisierung einer JSF-Seite ohne vollständigen HTTP-

Request-Response-Zyklus möglich ist. Die Beschränkung auf die Aktualisierung einer Seite impliziert, dass der Navigation-Handler bei Ajax-Requests nicht involviert ist. Trotzdem ist mit ein wenig Aufwand auch bei einem Ajax-Request eine Navigation möglich, wie wir in Abschnitt 7.3.1 zeigen. Wir verwenden dort auch die Methode `redirect()` des `ExternalContext`.

4.6.8 View-Parameter und Lesezeichen

JSF 2.0 führt View-Parameter ein, die auch mit GET-Requests verwendet werden können. View-Parameter sind nicht mit normalen Request-Parametern zu verwechseln, die man mit dem `<f:param>`-Tag definiert:

```
<h:commandButton action="..." value="...">
    <f:param name="fparam" value="Der-F-Param" />
</h:commandButton>
```

Auf der Zielseite erhält man mit dem EL-Ausdruck `#{{param['fparam']}}` den Wert des Request-Parameters, in diesem Beispiel „Der-F-Param“. Der Name `param` ist hier die in Tabelle 4.2 auf Seite 48 beschriebene vordefinierte Variable aller Request-Parameter.

View-Parameter werden als Sohnknoten der Metadaten einer View definiert. Metadaten sind wiederum als direkte Söhne der View mit `<f:metadata>` zu definieren:

```
<f:view>
    <f:metadata>
        <f:viewParam name="param" value="#{{data.wert}}" />
    </f:metadata>
    <div>
        Übergebener Parameter: #{{data.wert}}
    </div>
</f:view>
```

In obigem Beispiel wird der View-Parameter `param` definiert. Die Komponentenklasse `UIViewParameter` ist eine Unterklasse von `UIInput`, so dass deren Parameter verwendet werden können. Wir zeigen dies gleich anhand eines Beispiels. Ein GET-Request der dargestellten Seite mit angehängtem „`?param=23`“ weist dem View-Parameter den Wert „23“ zu. Im Beispiel besitzt die Managed Bean ein Integer-Property `wert`, das über die Wertebindung den Wert 23 zugewiesen bekommt. Zur Validierung geben wir diesen Wert im zweiten Teil der Seite wieder aus.

Während ein GET-Request einer normalen JSF-Seite zur Ausführung der Phase 1 *Wiederherstellung des Komponentenbaums* mit anschließendem Aufruf der Phase 6 *Rendern der Antwort* führt, wird bei einem GET-Request einer Sei-

te mit View-Parameter der volle JSF-Bearbeitungszyklus aller sechs Phasen durchlaufen. Damit ist es möglich, Lesezeichen für eine Seite zu erstellen, deren Verwendung zu Aktionen auf dem Server führt, oder Konvertierungen durchzuführen, wie im Beispiel mit dem String "23" geschehen.

Wir erweitern das Beispiel und weisen einem Bean-Property einen Prozentwert zur weiteren Verarbeitung zu. Zusätzlich verwenden wir einen Validierer und einen Konvertierer, so dass sowohl Phase 3 als auch Phase 4 involviert sind. Die Seite `view-parameter.xhtml` enthält den folgenden Code-Ausschnitt:

```
<f:metadata>
    <f:viewParam id="prozent" name="prozent"
        value="#{data.percentage}" required="true"
        requiredMessage="Prozentsatz erforderlich"
        converterMessage="Kein Prozentsatz (Zahl)"
        validatorMessage="Satz zwischen 0 und 100">
        <f:validateLongRange minimum="0" maximum="100"/>
        <f:event type="postValidate"
            listener="#{naviHandler.handleEvent}" />
    </f:viewParam>
</f:metadata>
<div>
    Prozentsatz: #{data.percentage}
</div>
```

Hier wird das Bean-Property `percentage` über eine Wertebindung an den View-Parameter gebunden, eine Konvertierung und zwei Validierungen durchgeführt, außerdem ein Event-Listener registriert. Weitere Anwendungsfälle, die auf dem Property `percentage` basieren, sind denkbar. So kann z.B. eine Liste aller möglichen ganzzahligen Prozentsätze bis zum Wert des View-Parameters generiert werden:

```
private Integer percentage;

public List<Integer> getData() {
    List<Integer> list = new ArrayList<Integer>(percentage);
    for (int i = 0; i < percentage; i++) {
        list.add(i);
    }
    return list;
}
```

Diese Liste lässt sich dann einfach zur Anzeige verwenden:

```
<h:dataTable value="#{data.data}" var="d">
    <h:column>#{d}</h:column>
</h:dataTable>
```

Bei diesem Beispiel wird also durch einen GET-Request-Parameter eine Liste von Zahlen von 0 bis zum Wert des Parameters generiert. Für den Anwendungstest erstellen wir die folgenden einfachen GET-Requests in einer HTML-Seite.

```
<a href="view-parameter.jsf?prozent=44"> ...
<a href="view-parameter.jsf"> ...
<a href="view-parameter.jsf?prozent=444"> ...
```

Für das erste Beispiel wird die Liste von Prozentpunkten ausgegeben. Das zweite und dritte Beispiel führt jeweils zu Fehlern, da im zweiten Beispiel kein Parameter **prozent** angegeben und im dritten Beispiel der Wert für den Parameter **prozent** größer als 100 ist.

Die URL inklusive Request-Parameter kann im Browser eingegeben oder als Lesezeichen angelegt werden. Es stellt sich jedoch die Frage, ob die URL zum Aufruf der Seite oder zur Ablage als Lesezeichen manuell erstellt werden muss oder ob JSF hier Hilfe leisten kann. Die beiden JSF-Tags **<h:link>** und **<h:button>**, die mit JSF 2.0 eingeführt wurden, realisieren genau diese Hilfestellung. Mit

```
<h:link outcome="view-parameter.xhtml"
    value="..." includeViewParams="true">
    <f:param name="prozent" value="12" />
</h:link>
```

oder alternativ **<h:button>** werden durch das Attribut **includeViewParams** die enthaltenen Parameterdefinitionen des **<f:param>**-Tags als View-Parameter der Zielseite an die generierte URL angehängt. Bei einem tatsächlichen Aufruf des Links oder der Schaltfläche kann die URL dann als Lesezeichen gespeichert werden.

Interessant bei der Verwendung der Komponenten **<h:button>** und **<h:link>** ist der Zeitpunkt, zu dem die URLs der Navigationsziele berechnet werden. Dies geschieht beim Rendern der Seite; die Ziel-URL ist damit in der gerenderten Seite literal enthalten. Dies steht im Gegensatz zu den Komponenten **<h:commandButton>** und **<h:commandLink>**, bei denen im Post-Request dynamisch das Ziel bestimmt wird. Das Tag **<h:link>** wird als **** gerendert. Das Tag **<h:button>** wird als **<input type="button" onclick="...">** gerendert. Beide Tags benötigen also kein Formular (**<h:form>**), um ausgeführt werden zu können.

4.6.9 Die technische Sicht

Die JSF-Navigation beruht auf der Verarbeitung von Action-Events bzw. dem Ergebnis dieser Verarbeitung. Nur UI-Komponenten, die das `ActionSource2`-Interface im Package `javax.faces.component` implementieren, können Action-Events auslösen. In der Standardimplementierung sind das die Komponenten `HTMLCommandButton` und `HTMLCommandLink`, beide Unterklassen von `UICommand`. Diese werden durch die JSF-Tags `<h:commandButton>` und `<h:commandLink>` repräsentiert. Eine exakte Darstellung des Zusammenhangs und des Komponentenbegriffs im Allgemeinen erfolgt in Kapitel 5.

Die Verarbeitung der Action-Events und deren Einfluss auf die Navigation erfolgt nun nach dem folgenden Verfahren. Zunächst werden alle Action-Listener aufgerufen, die sich auf ein Action-Event mit `<f:actionListener>` registriert haben. Einzelheiten dazu findet man in Abschnitt 4.5.2. Dann werden die Action-Listener aufgerufen, die über ein `actionListener`-Attribut der beiden genannten Komponenten registriert wurden. Diese beide Schritte beeinflussen die Navigation nicht. Im letzten Schritt wird nun der Action-Listener der Navigations-Komponente aufgerufen. Dieser verwendet die Konstanten der `action`-Attribute der beiden Komponenten, um nach entsprechenden Fällen in den Navigationsregeln zu suchen, oder er ruft die an die `action`-Attribute gebundenen Methoden auf und verwendet deren Rückgabewerte für die Navigation. Falls keine expliziten Navigationsregeln gefunden werden, wird die implizite Navigation angewendet.

Diese Darstellung des Navigationsverhaltens ist eine starke Vereinfachung des tatsächlichen Navigationsalgorithmus, genügt aber in der Regel für das Verständnis. Die vollständige Darstellung benötigt in der Spezifikation zweieinhalb Seiten (Abschnitt 7.4.2) und sei dem interessierten Leser als zusätzliche Lektüre empfohlen.

Aufgabe 4.12

Bei sehr vielen einzugebenden Daten werden diese häufig auf mehrere Seiten verteilt. Realisieren Sie eine solche Eingabe (mit beliebigen Daten) über drei Seiten, wobei

- die erste Seite die Schaltflächen „Weiter“ und „Abbruch“,
- die zweite Seite die Schaltflächen „Weiter“, „Zurück“ und „Abbruch“ und
- die dritte Seite die Schaltflächen „Zurück“, „Abbruch“ und „Abschluss“

enthalten. Die Daten werden sinnvollerweise in einer Managed Bean gehalten.



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für die Navigation sind im Projekt *jsf-im-detail* enthalten.

4.7 Internationalisierung

Die Sprache Java und die Java-Plattform wurden von Anfang an mit dem Anspruch eines weltweiten Einsatzes entwickelt. Java war somit Vorreiter auf diesem Gebiet. Andere Sprachen und Systeme, die vor Java entstanden, wurden erst nachträglich im Zeitalter globaler Unternehmen und des Internets mit entsprechenden Eigenschaften ausgestattet. Wichtig in diesem Zusammenhang sind z. B. die Klasse `Locale`, die eine geographische, politische oder kulturelle Region repräsentiert, die Klasse `ResourceBundle`, die das Vorhalten von Texten und Bildern für verschiedene Regionen ermöglicht, und die Klasse `Calendar`, die als abstrakte Klasse verschiedene Datums- und Zeitdarstellungen erlaubt. Eine nichtabstrakte Unterklasse ist `GregorianCalendar`, die in den meisten Ländern der Erde gängige Darstellungen erlaubt. Alle genannten Klassen sind im Package `java.util` enthalten, sind also Standard-Java-SE-Klassen.

JavaServer Faces erfinden die genannten Konzepte und Sprachmittel daher nicht neu, sondern bauen auf den Möglichkeiten von Java-SE auf. Bei Web-Anwendungen ist es offensichtlich, dass diese häufig globale Verwendung finden, so dass die Möglichkeit eines internationalen Einsatzes zwingend erforderlich ist. Unter *Internationalisierung* (engl. *Internationalization*) versteht man den Entwurf eines Systems derart, dass alle in verschiedenen Regionen variierenden Elemente identifiziert und so implementiert werden, dass sie austauschbar anstatt hart codiert sind. Beispiele für solche Elemente sind Beschriftungen und Texte, Piktogramme, Währungssymbole, Datums-, Zeit- und Zahlenformate. Als Abkürzung für den Begriff Internationalisierung hat sich *I18N* (oder *i18n*) eingebürgert, was für das beginnende *I*, 18 folgende Buchstaben und das schließende *N* im Englischen steht.

Eine internationalisierte Anwendung kann für verschiedene Regionen *lokaliert* werden, ohne den Programm-Code zu ändern. Man spricht von einer *Lokalisierung*. Im Detail bedeutet dies, dass die genannten Beschriftungen, Texte usw. für mehrere Regionen verfügbar sind. Analog zur Abkürzungsregel für I18N hat sich hier die Abkürzung L10N (engl. Localization) eingebürgert.

Grundlage für eine Lokalisierung ist die Möglichkeit der Verwendung verschiedener Zeichensätze. Der früher häufig verwendete ASCII-Zeichensatz verwen-

det 7 Bits zur Codierung, was jedoch nur für die englische Sprache ausreicht. Es wurde daher eine Familie von Zeichensätzen definiert, die mit 8 Bits viele Sprachen der Erde codieren kann. Die Familie der ISO-8859-Zeichensätze kann westeuropäische, aber auch kyrillische und türkische Zeichen codieren. Der Zeichensatz ISO-8859-1 definiert die westeuropäischen Zeichen, die z.B. für Deutsch, Englisch, Französisch und Spanisch ausreichen. ISO-8859-1 ist der Default-Zeichensatz für HTTP. Für Chinesisch, Koreanisch und Japanisch reicht aber auch dies bei Weitem nicht aus, so dass Unicode 16 Bits verwendet, bzw. verwendet hat. Neuere Versionen von Unicode verwenden eine bedarfsabhängige Länge von 1 bis 4 Bytes. Im Folgenden gehen wir von der Verwendung von UTF-8 als Unicode-Codierung aus. Verweise für eigene Recherchen zu diesem Thema findet man im Anhang C, dem URL-Verzeichnis. Anzumerken bleibt, dass der in Java definierte Datentyp `char` bzw. seine Wrapper-Klasse `Character` ursprünglich als 16-Bit-Wert definiert war. Seit Java 5 wird Unicode 4.0 unterstützt, was eine Anpassung an die bedarfsabhängige Code-Länge von Unicode erforderte. Wir gehen darauf nicht näher ein und verweisen den Leser auf die Dokumentation der jeweils aktuellen Character-Implementierung einer Java-Version, die im API-Doc der Klasse `Character` erfolgt. Ebenfalls lesenswert ist ein Kurzüberblick im Web [URL-U4S].

4.7.1 Lokalisierung

Die Lokalisierung ist nicht JSF-spezifisch, sondern basiert auf der Java-SE-Lokalisierung, die durch die Klasse `java.util.Locale` repräsentiert wird. Eine Lokalisierung besteht aus einem Code für die Sprache und einem Code für das Land. Dabei ist der Landes-Code optional. Die Klasse `Locale` besitzt entsprechende ein- und zweistellige Konstruktoren. In der textuellen Darstellung wird die Sprache durch zwei Kleinbuchstaben und das Land durch zwei Großbuchstaben mit einem Minus oder Unterstrich als Trennzeichen dargestellt. Tabelle 4.11 zeigt Beispiele für Lokalisierungs-Codes. Die im Rahmen dieser Codierung zu verwendenden Sprachen sind in der ISO-639 (ISO Language Codes) normiert. Sie können diese unter [URL-ISO-639] einsehen. Die Länder sind in der ISO-3166 (ISO Country Codes) normiert und können unter [URL-ISO-3166] nachgeschlagen werden.

Jeder View einer JSF-Anwendung ist eine Lokalisierung zugeordnet. Diese kann explizit durch die Verwendung des `locale`-Attributs gesetzt werden.

```
<f:view locale="de">
```

Wenn keine Lokalisierung gesetzt ist, verwendet das JSF-Laufzeitsystem die in der HTTP-Anfrage codierte Lokalisierung. HTTP schickt in der Anfrage das

Tabelle 4.11: Beispiele für Lokalisierungs-Codes

Code	Sprache	Land
en	Englisch	—
en-US	Englisch	USA
en-GB	Englisch	Großbritannien
es	Spanisch	—
de	Deutsch	—
de-DE	Deutsch	Deutschland
de-AT	Deutsch	Österreich
de-CH	Deutsch	Schweiz

Header-Element `Accept-Language`, das eine Präferenzliste von Lokalisierungen enthält, also z. B.

`Accept-Language: de, en-US, en`

Bei diesem Beispiel würde JavaServer Faces nach Lokalisierungen in der Reihenfolge Deutsch, US-amerikanisches Englisch und Englisch suchen.

Wie definiert man nun aber Lokalisierungen in JavaServer Faces? Lokalisierungen werden für eine Anwendung und somit nicht weiter verwunderlich in der JSF-Konfigurationsdatei definiert. Innerhalb des `<application>`-Elements werden im `<locale-config>`-Element alle unterstützten Lokalisierungen sowie die Default-Lokalisierung angegeben. Dies geschieht mit den Elementen `<supported-locale>` und `<default-locale>`. Listing 4.29 zeigt ein Beispiel.

Listing 4.29: Konfiguration von Lokalisierungen

```
<application>
  <locale-config>
    <default-locale>de</default-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
</application>
```

In diesem Beispiel werden Deutsch und Englisch als unterstützte Lokalisierungen und Deutsch als die Default-Lokalisierung definiert. Die zu verwendende Lokalisierung ergibt sich aus der Angabe der Lokalisierung in der Anfrage und den verfügbaren Lokalisierungen. Ist die in der Anfrage gewünschte Lokalisierung verfügbar (es werden alle im Header genannten Lokalisierungen der

Reihe nach durchprobiert), so wird diese auch verwendet. Ist die in der Anfrage gewünschte Lokalisierung nicht vorhanden, wird die Default-Lokalisierung verwendet. Die explizite Verwendung einer Lokalisierung im <f:view>-Tag überschreibt die nach diesem Verfahren festgelegte Lokalisierung.

Als letzter Teil einer Lokalisierung bleibt noch die Angabe der Texte für eine Lokalisierung. Die Texte einer Lokalisierung werden in einer Datei hinterlegt, die ein bestimmtes Format vorschreibt. Mehrere solcher Dateien, die nach einem bestimmten Schema zu benennen sind, werden zu einem *Resource-Bundle* zusammengefasst. Die abstrakte Klasse `java.util.ResourceBundle` erlaubt die Definition verschiedener lokalisierungsspezifischer Objekte, in der Regel Strings. Die Klasse `java.util.PropertyResourceBundle` als konkrete Unterklasse von `ResourceBundle` schreibt das von der Klasse `java.util.Properties` übernommene Dateiformat vor. Dieses Format ist sehr einfach und besteht aus Schlüssel/Wert-Paaren, die durch ein Gleichheitszeichen getrennt sind. Kommentare werden mit dem Nummernzeichen (#) begonnen und gehen bis zum Zeilenende. Die Listings 4.30 und 4.31 zeigen zwei Dateien mit deutschen und englischen Texten, die zusammen ein Resource-Bundle bilden. Die Schlüssel stehen links vom Gleichheitszeichen und sind in beiden Dateien identisch, die Werte stehen rechts vom Gleichheitszeichen und stellen die lokalisierungstypischen Texte dar.

Listing 4.30: Die Datei `messages.properties`

```
title = Übungen zur Internationalisierung
welcome = Willkommen zu I18N mit JSF-Praxis
input = Wert (mit Dezimalkomma)
save = Speichern
delete = Löschen
cancel = Abbrechen
```

Listing 4.31: Die Datei `messages_en.properties`

```
title = Exercises for Internationalization
welcome = Welcome to i18n with JSF Practice
input = Value (with decimal point)
save = Save
delete = Delete
cancel = Cancel
```

Die von Java verwendeten Methoden zum Lesen von Properties-Dateien arbeiten auf der ISO-8859-1-Codierung. Wollen Sie Zeichen verwenden, die in ISO-8859-1 nicht dargestellt werden können, müssen Sie Unicode-Escape-Sequenzen (\uxxxx) verwenden. Die in der Properties-Datei verwendeten deutschen Umlaute sind als solche erlaubt. Zu Demonstrationszwecken zeigt Listing 4.32 die alternative Darstellung mit Unicode-Escape-Sequenzen.

Listing 4.32: Die Datei `messages_de.properties` mit Unicode-Escape-Sequenzen

```
title = \u00DCbungen zur Internationalisierung
welcome = Willkommen zu I18N mit JSF-Praxis
input = Wert (mit Dezimalkomma)
save = Speichern
delete = L\u00F6schen
cancel = Abbrechen
```

Dateien eines Resource-Bundles haben immer die Dateiendung `properties`. Die Basis der Dateinamen ist frei wählbar, doch wird für jede in der Konfigurationsdatei angegebene Lokalisierung der Name der Lokalisierung mit Unterstrich angehängt. In unserem Beispiel sind dies die Dateien `messages_de.properties` und `messages_en.properties`. Die Default-Datei ist `messages.properties`, die die deutschen Texte enthält. Wird ein Schlüssel in der gesuchten Lokalisierungsdatei nicht gefunden, wird in der Default-Datei nach ihm gesucht. Im Beispiel kann daher die Datei `messages_de.properties` (Listing 4.33) leer sein.

Listing 4.33: Die Datei `messages_de.properties`

```
# ist der Default
```

Es sind nun alle Vorkehrungen getroffen, um die lokalisierten Texte des Resource-Bundles verwenden zu können. Hierzu wird die Anwendung in der JSF-Konfigurationsdatei um das Resource-Bundle erweitert. Listing 4.34 zeigt das entsprechende Element `<resource-bundle>`. Das enthaltene Element `<basename>` benennt den Basisnamen der Dateien, die die lokalisierten Texte enthalten. Das Element `<var>` deklariert eine Variable, die in der JSF-Datei in EL-Ausdrücken verwendet werden kann.

Listing 4.34: Lokalisierungen und Resource-Bundle

```
<application>
    <locale-config>
        <default-locale>de</default-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
    <resource-bundle>
        <base-name>de.jsfpraxis.detail.i18n.messages</base-name>
        <var>msg</var>
    </resource-bundle>
</application>
```

Da die JVM Properties-Dateien mit Hilfe des Class-Loaders lädt, entspricht der Basisname einer Properties-Datei den für Java-Klassen gültigen Regeln bezüglich der Übereinstimmung von Package-Präfix und Verzeichnisstruktur im Dateisystem. Der in Listing 4.34 genannte Name `messages` steht also für die Dateien `messages.properties`, `messages_en.properties` und `messages_de.properties` im Verzeichnis `i18n`, das sich wiederum hierarchisch in den anderen genannten Verzeichnissen befindet.

Listing 4.35 zeigt die Seite `intro.xhtml`, die das definierte Resource-Bundle verwendet. Die im `<resource-bundle>`-Element der JSF-Konfigurationsdatei definierte Variable `msg` kann nun in EL-Ausdrücken analog zu einer Managed Bean oder vordefinierten Variablen verwendet werden. In Zeile 3 etwa als Seitentitel, in den Zeilen 19 bis 22 als Beschriftung der Schaltflächen.

Listing 4.35: Die Seite `intro.xhtml`

```
1 <f:view>
2     <head>
3         <title>#{msg.title}</title>
4     </head>
5     <body>
6         <h:form>
7             <h:panelGrid columns="1">
8                 <h:outputText value="Accept-Language: \
9                               #{header['Accept-Language']}"/>
10            <h:panelGroup />
11            <h:outputText value="#{msg.welcome}" />
12            <h:panelGroup>
13                <h:outputLabel for="input" value="#{msg.input}: " />
14                <h:inputText id="input" value="#{i18nHandler.value}">
15                    <f:convertNumber pattern="00000.00" />
16                </h:inputText>
```

```
17      </h:panelGroup>
18      <h:panelGroup>
19          <h:commandButton value="#{msg.save}"
20              action="#{i18nHandler.save}" />
21          <h:commandButton value="#{msg.delete}" />
22          <h:commandButton value="#{msg.cancel}" />
23      </h:panelGroup>
24  </h:panelGrid>
25  <h:messages />
26 </h:form>
27 </body>
28 </f:view>
```

Um zu überprüfen, welche Lokalisierungen der Client der HTTP-Anfrage mitgibt, wird in den Zeilen 8/9 die Liste dieser Lokalisierungen ausgegeben (siehe die vordefinierten Variablen in Tabelle 4.2 auf Seite 48). Interessant ist noch die Verwendung des Zahlenkonvertierers in Zeile 15. Hier wird im Pattern der Punkt verwendet, der für das lokalisierte Dezimaltrennzeichen steht (siehe Tabelle 4.8 auf Seite 81). Die Managed Bean `i18nHandler` ist im Augenblick nicht relevant. Sie definiert lediglich ein Property `value` mit initialem Wert 0,0 und eine Action-Methode `save()`.

Für die Darstellung im Browser können nun die beiden Alternativen einer deutschen und englischen Lokalisierung getestet werden. Abbildung 4.22 zeigt beide Alternativen. Wenn Sie das Beispiel selbst nachvollziehen wollen, müssen Sie Ihren Browser dazu überreden, die Lokalisierungen `de` und `en` in der entsprechenden Reihenfolge zu übermitteln. Im Internet-Explorer können Sie dies unter dem Menü *Extras → Internetoptionen* und dann über die Schaltfläche *Sprachen* einstellen. Im Firefox finden Sie im Menü *Bearbeiten → Einstellungen* ebenfalls eine Schaltfläche *Sprachen* für diesen Zweck. Vergleichen Sie die beiden Browser-Darstellungen und den Quell-Code der JSF-Seite.

Die dargestellte Verwendung des Resource-Bundles und die Deklaration der Variablen `msg` wurde mit JSF 1.2 eingeführt. In den früheren Versionen musste das JSF-Tag `<f:loadBundle>` benutzt werden. Die Verwendung in Listing 4.36 entspricht im Ergebnis der oben dargestellten Alternative: Danach kann die Variable `msg` in der JSF-Seite verwendet werden.

Listing 4.36: Verwendung des JSF-Tags `<f:loadBundle>`

```
<f:loadBundle basename="de.jsfpraxis.detail.i18n.messages"
               var="msg" />
```

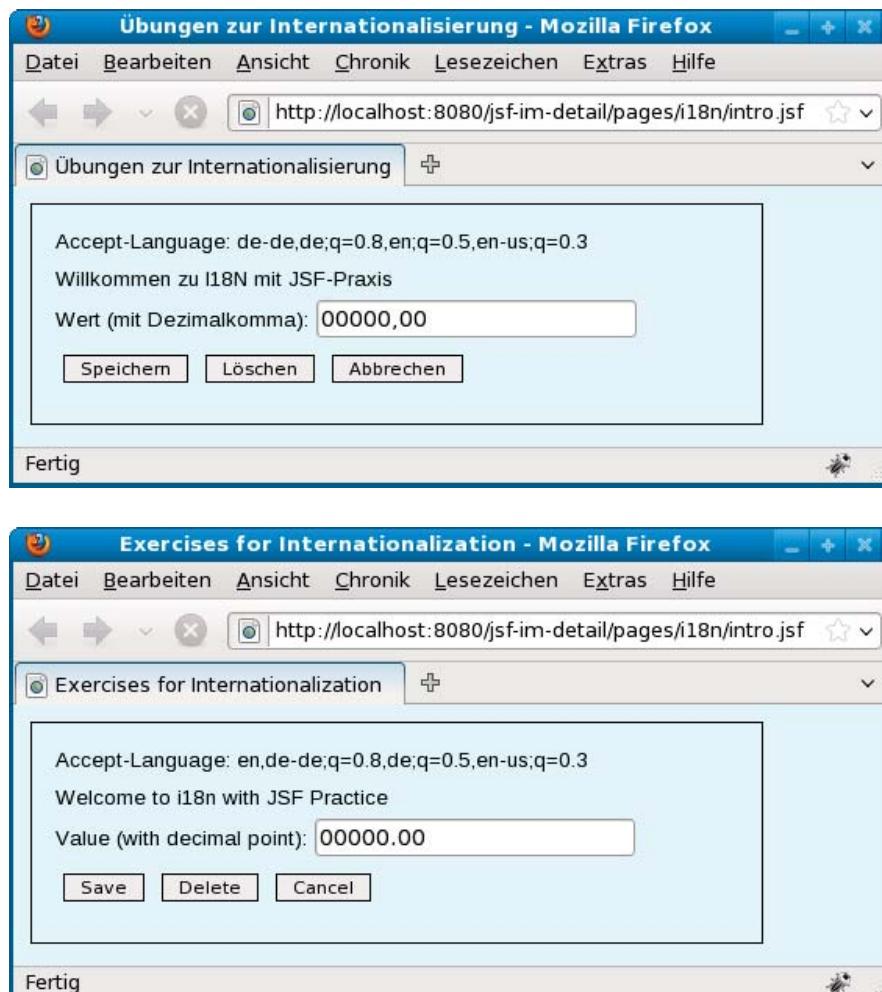


Abbildung 4.22: Die Seite `intro.xhtml` im Browser

Worin sich die beiden Alternativen unterscheiden, ist ihr Ausführungszeitpunkt und damit ihr Performanzverhalten. Während das über die JSF-Konfigurationsdatei geladene Resource-Bundle beim Anwendungsstart einmal gelesen wird, liest `<f:loadBundle>` bei jedem Request das Resource-Bundle. In der Regel sind Sie also mit der ersten Alternative besser beraten.



Das Laden eines Resource-Bundles sollte in der Regel über die JSF-Konfigurationsdatei und nicht über das Tag `<f:loadBundle>` vorgenommen werden, da dies wesentlich effizienter ist.

4.7.2 Dynamische und explizite Lokalisierung

Wir haben bereits die dynamische Lokalisierung auf Basis des HTTP-Headers der Browser-Anfrage und die explizite Lokalisierung mit dem `locale`-Attribut des `<f:view>`-Tags dargestellt. An dieser Stelle soll nun die Möglichkeit der expliziten Lokalisierung auf Benutzerwunsch und somit ebenfalls einer dynamischen Lokalisierung dargestellt werden. Grundlage ist das `locale`-Attribut, das nicht nur Konstanten, sondern auch EL-Ausdrücke zulässt. Als Anwendungsfall einer solchen Möglichkeit ist die Personalisierung einer Anwendung zu nennen, indem die vom Benutzer ausgewählte Lokalisierung bei späteren Seitenzugriffen jeweils beachtet wird.

Listing 4.37 zeigt eine JSF-Seite, deren Lokalisierung durch einen EL-Ausdruck über ein Bean-Property definiert wird. Das Drop-Down-Menü in Zeile 6 erlaubt es dem Benutzer, dieses Property zu ändern.

Listing 4.37: Die Seite locale-prog.xhtml

```
1 <f:view locale="#{myLocaleHandler.locale}">
2   <h:form id="form">
3     <h:commandButton value="#{msg.save}" />
4     <h:commandButton value="#{msg.delete}" />
5     <h:commandButton value="#{msg.cancel}" />
6     <h:selectOneMenu value="#{myLocaleHandler.locale}">
7       <f:selectItems value="#{myLocaleHandler.locales}" />
8     </h:selectOneMenu>
9     <h:commandButton value="#{msg.change}"
10        action="#{myLocaleHandler.changeLocale}" />
11   </h:form>
12 </f:view>
```

Interessant ist noch die Bestimmung der in der Anwendung bekannten Lokalisierungen. Die Methode `getLocales()` des Handlers wird in Zeile 7 im `<f:selectItems>`-Tag verwendet; sie ist im Listing 4.38 dargestellt und verwendet die Methode `getSupportedLocale()` des `Application`-Objekts, die einen Iterator über die in der JSF-Konfigurationsdatei deklarierten Lokalisierungen zurückliefert.

Listing 4.38: Der Lokalisierungs-Handler (Klasse MyLocaleHandler)

```
@ManagedBean
@SessionScoped
public class MyLocaleHandler {
```

```
private String locale;

public MyLocaleHandler() {
    locale = FacesContext.getCurrentInstance()
        .getApplication().getDefaultLocale().toString();
}

public List<String> getLocales() {
    List<String> l = new ArrayList<String>();
    Iterator<Locale> iter = FacesContext.getCurrentInstance()
        .getApplication().getSupportedLocales();
    for (; iter.hasNext(); ) {
        l.add(iter.next().toString());
    }
    return l;
}

public String changeLocale() {
    FacesContext.getCurrentInstance().getViewRoot()
        .setLocale(new Locale(locale));
    return null;
}
}
```

Neben den konfigurierten Lokalisierungen kann auch die Default-Lokalisierung bestimmt werden. Die Methode `getDefaultLocale()` des `Application`-Objekts gibt die Default-Lokalisierung zurück, die im Bean-Konstruktor verwendet wird, um das `locale`-Property initial mit einem Wert zu versehen.

Um die Anwendung in allen Details verstehen zu können, sei der Leser noch einmal auf Abschnitt 4.1 verwiesen. Das Setzen der Lokalisierung der View erfolgt in der Phase 1, Wiederherstellung des Komponentenbaums. Die vom Benutzer ausgewählte Lokalisierung wird dann in Phase 4, Aktualisierung der Modellobjekte, in das Bean-Property geschrieben. Ohne weitere Programmierung würde in Phase 6, Rendern der Antwort, die alte Lokalisierung im View-Objekt stehen und somit mit der alten Lokalisierung gerendert werden. Die Methode `changeLocale()` muss daher in der View-Root explizit die Methode `setLocale()` in Phase 5, Aufruf der Anwendungslogik, aufrufen, um sicherzustellen, dass das View-Objekt in Phase 6 den korrekten Lokalisierungswert besitzt.

4.7.3 Klassen als Resource-Bundles

Auf Seite 150 haben wir die abstrakte Klasse `ResourceBundle` als Grundlage von Javas Resource-Bundles eingeführt. Die Klasse `PropertyResourceBundle`

dient intern als Realisierung der vorgestellten Lokalisierung mittels Property-Dateien. Die Klasse `ListResourceBundle`, wie `PropertyResourceBundle` ebenfalls im Package `java.util`, kann als Grundlage einer Lokalisierung mittels Java-Klassen dienen. Somit ist es z.B. möglich, die Texte der Lokalisierungen in einer Datenbank statt in mehreren Dateien zu verwalten und damit verschiedenen Anwendungen dieselben Lokalisierungen zur Verfügung zu stellen.

Wir verzichten an dieser Stelle auf einen Datenbankzugriff und beschränken uns auf ein reines Java-Beispiel. Die Listings 4.39 und 4.40 stellen die Lokalisierungsklassen als Unterklasse von `ListResourceBundle` dar und überschreiben die Methode `getContents()`, die für das Mapping der Lokalisierungsschlüssel auf Werte zuständig ist.

Listing 4.39: Die Lokalisierungsklasse `Numbers.java`

```
public class Numbers extends ListResourceBundle {

    private static final Object[][] contents = {
        { "_1", "eins"}, { "_2", "zwei"}, { "_3", "drei"},
        { "_4", "vier"}, { "_5", "fünf"}, { "_6", "sechs"},
        { "_7", "sieben"}, { "_8", "acht"}, { "_9", "neun"},
    };

    @Override
    protected Object[][] getContents() {
        return contents;
    }
}
```

Listing 4.40: Die Lokalisierungsklasse `Numbers_en.java`

```
public class Numbers_en extends ListResourceBundle {

    private static final Object[][] contents = {
        { "_1", "one"}, { "_2", "two"}, { "_3", "three"},
        { "_4", "four"}, { "_5", "five"}, { "_6", "six"},
        { "_7", "seven"}, { "_8", "eight"}, { "_9", "nine"},
    };

    @Override
    protected Object[][] getContents() {
        return contents;
    }
}
```

Die Namensgebung der Klassen erfolgt nach dem bereits vorgestellten Schema für Resource-Bundles durch Anhängen eines Unterstrichs und des Länder-Codes. Die Deklaration im Deployment-Descriptor erfolgt nach dem ebenfalls bereits bekannten Verfahren. Listing 4.41 zeigt dies beispielhaft.

Listing 4.41: Lokalisierungen und Resource-Bundle

```
<application>
    <locale-config>
        <default-locale>de</default-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
    <resource-bundle>
        <base-name>de.jsfpraxis.detail.i18n.Numbers</base-name>
        <var>number</var>
    </resource-bundle>
</application>
```

Die Verwendung der Variablen `number` erfolgt ebenfalls wie gewohnt. Listing 4.42 zeigt dies an einem einfachen Beispiel.

Listing 4.42: Verwendung der Variablen `number`

```
<h:panelGrid columns="2" rowClasses="odd,even">
    <h:outputText value="1"/> <h:outputText value="#{number._1}" />
    <h:outputText value="2"/> <h:outputText value="#{number._2}" />
    <h:outputText value="3"/> <h:outputText value="#{number._3}" />
    <h:outputText value="4"/> <h:outputText value="#{number._4}" />
    <h:outputText value="5"/> <h:outputText value="#{number._5}" />
    <h:outputText value="6"/> <h:outputText value="#{number._6}" />
    <h:outputText value="7"/> <h:outputText value="#{number._7}" />
    <h:outputText value="8"/> <h:outputText value="#{number._8}" />
    <h:outputText value="9"/> <h:outputText value="#{number._9}" />
</h:panelGrid>
```

Aufgabe 4.13

Erzeugen Sie eine lokalisierte Seite mit einer Graphik. Die Graphik muss in mehreren Versionen vorliegen. Den lokalisierten Pfad zu den Graphiken verwenden Sie als URL für das Tag `<h:graphicImage>`. Die Dokumentation des `<h:graphicImage>`-Tags finden Sie in der HTML-Tag-Bibliothek auf Seite 402.

4.7.4 Managed Beans und Lokalisierung

Die Verwendung der Lokalisierungen fand bisher ausschließlich in der JSF-Seite statt. Es gibt jedoch Situationen, in denen eine Managed Bean oder eine andere Java-Klasse Zugriff auf die Lokalisierung oder das Resource-Bundle benötigt, sie aber nicht wie im Abschnitt 4.7.3 selbst implementiert. Denkbar ist z. B. die dynamische Erstellung einer lokalisierten `SelectItems`-Liste für die Verwendung in Menüs oder eine an regionale Gegebenheiten angepasste Abfolge von Seiten in einem Arbeitsablauf. Listing 4.43 zeigt die Methode `getTitle()`, die in Java auf das Resource-Bundle zugreift, das in der Konfigurationsdatei im `<resource-bundle>`-Element angegeben ist und für die Lokalisierung der aktuellen `UIViewRoot` den lokalisierten String für `title` zurückgibt.

Listing 4.43: Zugriff auf lokalisierten Text

```
public String getTitle() {
    FacesContext context = FacesContext.getCurrentInstance();
    ResourceBundle rb = context.getApplication()
        .getResourceBundle(context, "msg");
    return rb.getString("title");
}
```

Für ein weiteres Beispiel gehen wir davon aus, dass sich in einer bestimmten Lokalisierung die Abfolge von Seiten in einem Arbeitsablauf zur normalen Abfolge unterscheidet, indem z. B. zusätzliche Seiten anzuzeigen und vom Benutzer auszufüllen sind. Es gilt also, eine lokalisierte Navigation zu erstellen. Hierzu wird eine Action-Methode definiert, die abhängig von der Lokalisierung verschiedene Ergebnisse liefert, die dann in den Navigationsregeln verwendet werden. Dies ist sehr einfach zu realisieren, wie Listing 4.44 zeigt.

Listing 4.44: Action-Methode für lokalisierte Navigation

```
/**
 * Gibt die aktuelle Lokalisierung zurück. Im Beispiel
 * <code>de</code> oder <code>en</code>.
 */
public String save() {
    Locale locale = FacesContext.getCurrentInstance()
        .getViewRoot().getLocale();
    return locale.toString();
}
```

Die Navigationsregel in Listing 4.45 verwendet die Rückgabe der Action-Methode zur Navigation und verzweigt für das Ergebnis „en“ zur Seite `page.xhtml` und für das Ergebnis „de“ zur Seite `seite.xhtml`.

Listing 4.45: Navigationsregeln für lokalisierte Seitenfolgen

```
<navigation-rule>
  <from-view-id>/pages/i18n/intro.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>en</from-outcome>
    <to-view-id>/pages/i18n/page.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>de</from-outcome>
    <to-view-id>/pages/i18n/seite.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele für die Konfiguration sind im Projekt *jsf-im-detail* enthalten.

4.8 Konfiguration

Bis zur Version 1.2 von JSF wurde die gesamte Konfiguration einer JSF-Implementierung über XML-Dateien realisiert. Die mit EJB 2 erkannten Probleme komplexer und überfrachteter XML-Konfigurationen wurden mit EJB 3 und Java-EE 5 zum großen Teil überwunden. Die Konfiguration kann nun alternativ mit Annotationen erfolgen. Mit Version 2.0 vollzieht JSF ebenfalls diesen Schritt und ermöglicht es, viele Konfigurationseinstellungen durch Annotationen vorzunehmen, was wir unter anderem bereits bei Managed Beans, Konvertierern und Validierern getan haben.

Man muss bei JavaServer Faces zwischen zwei Konfigurationen unterscheiden: einmal die Optionen, die mit der Konfiguration des zugrunde liegenden Servlet-Systems zu tun haben, und dann die mit JavaServer Faces direkt involvierten Optionen. Von Letzteren haben wir in diesem Kapitel schon mehrere Beispiele gesehen, z. B. die Deklaration von Managed Beans, Validierern und Konvertierern sowie die Definition von Navigationsregeln. In diesem Abschnitt

beschreiben wir sowohl die Servlet- als auch die JSF-Konfiguration, wobei wir uns bei der Servlet-Konfiguration auf JSF-relevante Details beschränken.

4.8.1 Die Servlet-Konfiguration

Servlets werden in Spezifikationen innerhalb des JCP detailliert beschrieben. JSF 2.0 erfordert eine Servlet-Implementierung der Version 2.3 oder höher. Die im Rahmen von Java-EE 6 entwickelte Version der Servlet-Spezifikation ist 3.0. Die Spezifikationen 2.3 bis 3.0 können unter [URL-JSR53, URL-JSR154, URL-JSR315] eingesehen werden. Für unsere Zwecke genügt ein Blick in den *Deployment-Deskriptor*.

Eine Web-Anwendung muss nach der Servlet-Spezifikation ein bestimmtes Verzeichnis-Layout haben. Im Wurzelverzeichnis der Applikation existiert das Verzeichnis WEB-INF, in dem die zentrale Konfigurationsdatei web.xml, der so genannte *Deployment-Deskriptor*, liegt. In ihm werden alle Servlets der Anwendung sowie andere Ressourcen konfiguriert. JavaServer Faces sind eine Web-Anwendung, die durch *ein* Servlet, das *Faces-Servlet*, realisiert wird. Es muss also zumindest dieses Servlet deklariert und konfiguriert werden. Mit der Servlet-Version 3.0 wurden diese Anforderungen optional. Wir gehen darauf in Abschnitt 9.5.1 ein und beschränken uns hier auf den Stand vor Version 3.0.

Listing 4.46 zeigt den Deployment-Deskriptor web.xml, wie er in den meisten Anwendungen und Beispielen des Buches so oder in ähnlicher Form verwendet wird.

Listing 4.46: Der Deployment-Deskriptor web.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="2.5"
3      xmlns="http://java.sun.com/xml/ns/javaee"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee \
6                          http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
7
8
9      <description>Standard-Deskriptor JavaServer Faces</description>
10
11     <!-- Servlet Definition -->
12     <servlet>
13         <servlet-name>Faces Servlet</servlet-name>
14         <servlet-class>
15             javax.faces.webapp.FacesServlet
16         </servlet-class>
17     </servlet>
```

```
18      <!-- Servlet Mapping -->
19      <servlet-mapping>
20          <servlet-name>Faces Servlet</servlet-name>
21          <url-pattern>*.jsf</url-pattern>
22      </servlet-mapping>
23
24
25 </web-app>
```

Zunächst wird im Wurzelement `<web-app>` ein Namensraum und ein Schema angegeben. Bis zur Version 2.3 der Servlet-Spezifikation wurde der Deployment-Deskriptor mit einer DTD beschrieben. Seit Version 2.4 erfolgt die Beschreibung mit XML-Schema. Im Beispiel verwenden wir das Schema zur Version 2.5. Mittlerweile unterstützen alle modernen IDEs die schema-basierte Entwicklung von XML-Dateien, so dass bestimmte Eigenschaften, wie z.B. die Auto-Vervollständigung von Element- und Attributnamen, unterstützt werden.

Im Beispiel definieren wir zunächst das Faces-Servlet. Dabei ist die implementierende Servlet-Klasse vorgeschrieben, der Servlet-Name frei wählbar. Dieser wird im Servlet-Mapping auf ein URL-Pattern abgebildet. Im Beispiel verwenden wir das sogenannte *Extension-Mapping*. Alternativ kann das *Präfix-Mapping* verwendet werden, bei dem eine Verzeichnisstruktur als Präfix vorgegeben wird, also z. B.

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

Der Deployment-Deskriptor definiert in der Version 2.5 61 Elemente, von denen allerdings für JavaServer Faces in der Regel nur das `<context-param>`-Element Verwendung findet. Dieser *Kontextparameter* wird allgemein verwendet, um den Servlets einer Web-Anwendung Zugriff auf Initialisierungsparametern zu gewähren. Ein Kontextparameter besteht aus der geschachtelten Definition eines Namens und eines Werts:

```
<context-param>
    <param-name>Name</param-name>
    <param-value>Wert</param-value>
</context-param>
```

Tabelle 4.12 auf der nächsten Seite zeigt die Kontextparameter zur JSF-Konfiguration. Da alle den Präfix `javax.faces` besitzen, haben wir diesen

in der Darstellung weggelassen. Der erste Kontextparameter der Tabelle, `CONFIG_FILES` ist also korrekt als `javax.faces.CONFIG_FILES` zu schreiben.

Tabelle 4.12: Kontextparameter zur Konfiguration

Parameter	Beschreibung
<code>CONFIG_FILES</code>	Kontextrelative Liste von Dateien, falls mehrere JSF-Konfigurationsdateien verwendet werden sollen.
<code>DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE</code>	Flag zur Verwendung der Default-Zeitzone anstatt GMT in <code>DateTimeConverter</code> .
<code>DECORATORS</code>	Liste von Klassennamen (Type <code>TagDecorator</code>), durch Semikolon getrennt.
<code>DEFAULT_SUFFIX</code>	Liste von Alternativ-Suffixen für JSF-Dateien, durch Leerzeichen getrennt. Default ist „ <code>.xhtml .jsp</code> “.
<code>DISABLE_FACELET_JSF_VIEWHANDLER</code>	Flag für den JSF 1.2 Kompatibilitätsmodus des View-Handlers.
<code>FACELETS_LIBRARIES</code>	Liste von Dateien für Facelets-Tag-Bibliotheken, durch Semikolon getrennt.
<code>FACELETS_BUFFER_SIZE</code>	Buffer-Größe des generierten <code>ResponseWriter</code> .
<code>FACELETS_REFRESH_PERIOD</code>	Zeitdauer in Sekunden, für die der Facelets-Compiler nach geänderten Facelets-Seiten sucht. Für <code>-1</code> wird nicht geprüft. Dies ist in der Regel für Produktionssysteme zu empfehlen.
<code>FACELETS_RESOURCE_RESOLVER</code>	Qualifizierter Klassenname des zu verwendenden <code>ResourceResolvers</code> .
<code>FACELETS_SKIP_COMMENTS</code>	Flag, das das Rendern eines XML-Kommentars in einer Facelets-Seite verhindert.
<code>FACELETS_SUFFIX</code>	Alternativer Suffix für Facelets-Seiten.

Tabelle 4.12: Kontextparameter zur Konfiguration (Fortsetzung)

Parameter	Beschreibung
FACELETS_VIEW_MAPPINGS	Liste von Resource-Patterns, durch Semikolon getrennt. Passende Ressourcen werden als Facelets-Seiten interpretiert. Beispiel: „/pages/*;*.xhtml“.
FULL_STATE_SAVING_VIEW_IDS	Liste von View-Ids, durch Komma getrennt, deren Zustand komplett, d. h. wie in JSF 1.2 gespeichert werden.
INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL	Leere Strings in Eingabefeldern werden in <code>null</code> umwandelt.
LIFECYCLE_ID	Qualifizierter Name der zu verwendenden <code>Lifecycle</code> -Implementierung.
PARTIAL_STATE_SAVING	Flag zur Steuerung der partiellen Zustandsspeicherung.
PROJECT_STAGE	Aktueller Entwicklungsstand des Systems. Mögliche Werte: <code>Development</code> , <code>UnitTest</code> , <code>SystemTest</code> , <code>Production</code> .
RESOURCE_EXCLUDES	Liste von Dateinamenerweiterungen von Ressourcen, die nicht ausgeliefert werden dürfen. Default-Wert: <code>.class .jsp .jspx .properties .xhtml</code> .
STATE_SAVING_METHOD	Speicherort für den Komponentenbaum. Mögliche Werte sind <code>server</code> (Default) und <code>client</code> .
SEPARATOR_CHAR	Trennzeichen für Client-Ids, die mehrere Bestandteile haben. Default ist der Doppelpunkt, der laut Spezifikation nicht geändert werden darf.
VALIDATE_EMPTY_FIELDS	Validierung leerer Eingaben, falls <code>true</code> . Für den Wert <code>auto</code> wird bei Vorhandensein einer JSR303-Implementierung <code>true</code> , sonst <code>false</code> gesetzt.

Tabelle 4.12: Kontextparameter zur Konfiguration (Fortsetzung)

Parameter	Beschreibung
DISABLE_DEFAULT_BEAN_VALIDATOR	Flag, das die Verwendung einer JSR303-Implementierung steuert.

Einige der in Tabelle 4.12 beschriebenen Kontextparameter haben wir als *Flag* bezeichnet. Die Spezifikation vergleicht den Wert dieser Flags durch den Ausdruck `toLowerCase().equals("true")` mit dem Wahrheitswert `true`. Alle anderen Werte als verschiedene Schreibungen von `true` werden daher als `false` interpretiert. Wir empfehlen dem Leser, trotzdem nur die Werte `true` und `false` zu verwenden.

Im Folgenden diskutieren wir die wichtigeren Kontextparameter im Detail.

Zustandsspeicherung

Der Zustand einer View (der in Abschnitt 4.1.1 auf Seite 38 beschriebene Komponentenbaum) inklusive seiner Konvertierer und Validierer muss zwischen zwei Requests gespeichert werden. Dies kann auf dem Server oder auf dem Client erfolgen. Der entsprechende Kontextparameter ist `STATE_SAVING_METHOD`. Im folgenden Beispiel wird der Server als Speicherort definiert.

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
</context-param>
```

Die Speicherung auf dem Client erfolgt in einem versteckten Input-Element:

```
<input type="hidden"
       name="javax.faces.ViewState" id="javax.faces.ViewState"
       value="...'" />
```

Die Speicherung auf dem Server erfolgt in der Regel in der `HttpSession` des Servlets, wobei zur Identifikation ebenfalls das oben dargestellte Verfahren verwendet wird.

Um zu entscheiden, welche der beiden Alternativen für eine bestimmte Anwendung die bessere ist, muss man sich über die Vor- und Nachteile der Alternativen im Klaren sein. Eine Speicherung auf dem Server benötigt offensichtlich mehr Speicherplatz auf dem Server. Dies ist bei kleinen Anwendungen mit wenigen gleichzeitigen Benutzern nicht problematisch, kann aber bei Tausenden von Benutzern durchaus zum Tragen kommen. Eine Speicherung auf dem Client erhöht das Kommunikationsvolumen, und zwar in beiden Richtungen. Falls eine View viele Komponenten hat, könnte dies bei geringen Bandbreiten

zu einer Verlangsamung führen. Man hat also auch hier den oft anzutreffenden Zeit/Platz-Trade-Off der Informatik.

Ebenfalls zu bedenken ist, dass die Speicherung auf dem Client problematisch bezüglich der Sicherheitsanforderungen der Anwendung sein kann, falls z. B. Benutzername und Passwort in der Session gespeichert sind. JSF-Implementierungen bieten hier wahrscheinlich Lösungsmöglichkeiten an. Für GlassFish haben wir diese in Abschnitt 10.1.3 beschrieben.

Konfigurationsdateien

Im Kontextparameter `CONFIG_FILES` werden die Konfigurationsdateien für JavaServer Faces durch Kommata getrennt angegeben. Default ist hier die Datei `/WEB-INF/faces-config.xml`, d. h. diese Datei wird ohne Definition des Kontextparameters verwendet. Die Beispiele dieses Kapitels, das Sie als Projekt *jsf-im-detail* herunterladen können, haben wir in verschiedenen Verzeichnissen abgelegt. Die Konfiguration ist in mehrere Dateien aufgeteilt. Die folgende Definition des Kontextparameters `CONFIG_FILES` ist dem Beispielprojekt entnommen.

```
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>
        /WEB-INF/faces-config-config.xml,
        /WEB-INF/faces-config-el.xml,
        /WEB-INF/faces-config-mb.xml,
        /WEB-INF/faces-config-vc.xml,
        /WEB-INF/faces-config-events.xml,
        /WEB-INF/faces-config-nav.xml,
        /WEB-INF/faces-config-i18n.xml
    </param-value>
</context-param>
```

Man erkennt für die einzelnen Abschnitte des Kapitels separate Konfigurationsdateien. Falls die Datei `/WEB-INF/faces-config.xml` in der Liste auftaucht, ist sie laut Spezifikation zu ignorieren.

Die Reihenfolge des Ladens der JSF-Konfigurationsdateien erfolgt durch Überprüfung der Existenz des Kontextparameters `CONFIG_FILES`. Ist er vorhanden, werden die über ihn definierten Konfigurationsdateien geladen, sonst die Datei `/WEB-INF/faces-config.xml`. Dies gilt jedoch nur, falls Sie keine weiteren Bibliotheken verwenden. Im allgemeinen Fall wird dieser Reihenfolge noch das Durchsuchen aller Jar-Dateien im `/WEB-INF/lib`-Verzeichnis vorausgestellt. Sollten sich JSF-Bibliotheken mit `/META-INF/faces-config.xml`-

Dateien darunter befinden, werden diese Konfigurationsdateien als Erste gelesen.

Projektphasen

JSF erlaubt seit der Version 2.0 die Klassifizierung eines Projekts bezüglich seines Entwicklungsstands. Die möglichen Entwicklungsphasen `Development`, `Production`, `SystemTest` und `UnitTest` sind als Werte des Aufzählungstyps `javax.faces.application.ProjectStage` für den Kontextparameter `PROJECT_STAGE` verwendbar, wie etwa in folgendem Beispiel:

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

Der Default für `PROJECT_STAGE` ist `Production`.

Die konfigurierte Projektphase ist programmatisch abfragbar. Mit der Methode `getProjectStage()` der Klasse `Application` können damit leicht Tests realisiert werden, die abhängig von der Projektphase verschiedene Funktionalitäten steuern.

Wir empfehlen Ihnen die Konfiguration der Phase `Development` während der Programmentwicklung sehr, da die Referenzimplementierung in dieser Entwicklungsphase mehr Warnungen in das Log ausgibt als im Default-Fall. Zusätzlich erhalten JSF-Seiten, die kein `<h:messages>`-Tag enthalten, automatisch ein solches Tag eingefügt. Unser Tipp zur Verwendung eines `<h:messages>`-Tags auf Seite 135 lässt sich somit konfigurativ und ohne Editieren der JSF-Seite realisieren.



Die Verwendung der Phase `Development` während der Programmentwicklung ist sehr zu empfehlen, da mehr Meldungen in das Log geschrieben und Validierungs- und Konvertierungsfehler automatisch angezeigt werden.

Optimierte Zustandsspeicherung

Ebenfalls eine Neuerung der Version 2.0 ist die optimierte Zustandsspeicherung der View, die mit dem Kontextparameter `PARTIAL_STATE_SAVING` konfiguriert wird. Eine bekannte Schwäche von JSF war die Größe des für die Zustandsspeicherung benötigten Speicherplatzes. Alle Komponenten, Validierer und Konvertierer einer View müssen wie in Abschnitt 4.1.1 dargestellt zwischen den Aufrufen gespeichert werden. Sowohl alle Standard-Komponenten als auch Standard-Validierer und -Konvertierer implementieren das Interface

`StateHolder` im Package `javax.faces.component`. Mit Version 2.0 implementieren sie auch das neu eingeführte Interface `PartialStateHolder` im selben Package, das für die optimierte Zustandsspeicherung zuständig ist. Dabei werden nur die Änderungen bezüglich der JSF-Seite gespeichert, so dass der aktuelle Komponentenbaum aus der Seite und der gespeicherten Änderungen wiederhergestellt werden kann. Für eine JSF-2.0-Implementierung ist die optimierte Zustandsspeicherung der Default. Die Implementierer der Referenzimplementierung und von MyFaces sprechen von einer Verteilung des benötigten Speichers bei Verwendung der optimierten Zustandsspeicherung.

Zugriff auf Konfigurationsdaten

In der Regel benötigt man eher selten Zugriff auf die Konfigurationsdaten. Eine Ausnahme bilden die Projektphasen, deren Kenntnis zur Laufzeit sinnvolle Verwendungsmöglichkeiten erlauben. Aber auch andere Konfigurationsparameter können eventuell nützlich sein, so dass wir uns kurz über den Zugriff auf Konfigurationsdaten Gedanken machen wollen.

Offensichtlich können diese in einer JSF-Seite direkt über die in Tabelle 4.2 auf Seite 48 dargestellte vordefinierte Variable `initParam` zugegriffen werden:

```
Zustandsspeicherung:  
    "#{initParam['javax.faces.STATE_SAVING_METHOD']}" <br />  
Konfigurationsdateien:  
    "#{initParam['javax.faces.CONFIG_FILES']}" <br />  
Projektphase:  
    "#{initParam['javax.faces.PROJECT_STAGE']}" <br />
```

Die Entsprechung in Java erfolgt über den Servlet-Kontext und dessen Methode `getInitParameter()`:

```
FacesContext.getCurrentInstance().getExternalContext()  
    .getInitParameter("javax.faces.PROJECT_STAGE");
```

Für einige Kontextparameter existieren aber auch entsprechend spezialisierte Methoden, wie etwa für die schon erwähnten Projektphasen, die optimierte Zustandsspeicherung und den Speicherort. Wir geben die Signaturen der Methoden inklusive typischen Aufruf an:

- Projektphase:

```
ProjectStage Application.getProjectStage()
```

- Direkter Zugriff auf Kontextparameter:

```
Object FacesContext.getAttributes().get("partialStateSaving")
```

- Zustandsspeicherung:

```
boolean Application.getStateManager()  
    .isSavingStateInClient(FacesContext ctx))
```

Sichern der JSF-Quelltexte

Als abschließendes Thema aus dem Bereich der Servlet-Konfiguration wollen wir uns einem Sicherheitsaspekt widmen. Das Faces-Servlet kann durch die Konfiguration eines Security-Constraint den Zugriff auf Seiten beschränken. In der Regel wird dies über eine rollenbasierte Benutzerautorisierung realisiert. So können bestimmte Seiten z. B. ausschließlich durch den Administrator, andere durch Sachbearbeiter einer bestimmten Abteilung verwendet werden. Die Integration dieser Möglichkeiten in JSF ist jedoch nur bedingt möglich, so dass sie als grundlegendes Sicherheitskonzept ausscheiden. Wir sprechen dies in Abschnitt 8.3 noch einmal an.

An dieser Stelle wollen wir jedoch das generelle Problem der Auslieferung von JSF-Quelldateien angehen. Die Quelldateien von JSF-Seiten sind in der Regel vor dem Zugriff durch Benutzer zu schützen. Die Seiten können über ein URL-Pattern als schützenswerte Web-Ressource deklariert werden. Das folgende XML-Segment in der Datei `web.xml` verhindert, dass XHTML-Dateien direkt ausgeliefert werden.

```
<security-constraint>  
    <display-name>Schutz der Facelets-Quellen</display-name>  
    <web-resource-collection>  
        <web-resource-name>XHTML</web-resource-name>  
        <url-pattern>*.xhtml</url-pattern>  
    </web-resource-collection>  
    <auth-constraint />  
</security-constraint>
```

Im Rumpf des `<auth-constraint>`-Elements sind üblicherweise die Benutzerrollen aufzuführen, für die der Zugriff auf die Web-Ressource erlaubt ist.

Dem aufmerksamen Leser wird aufgefallen sein, dass das beschriebene Verfahren einfacher über den Kontextparameter `RESOURCE_EXCLUDES` realisiert werden kann. Dieser ist jedoch erst in JSF 2.0 verfügbar, so dass in früheren Versionen das hier dargestellte Verfahren zu verwenden ist.

Aufgabe 4.14

Laden Sie das Projekt *jsf-im-detail* herunter, und installieren Sie es. Machen Sie sich mit den Konfigurationsdateien vertraut, und probieren Sie verschiedene Einstellungen aus.

Aufgabe 4.15

Ändern Sie den Ort der Zustandsspeicherung im Deployment-Deskriptor auf

client, und vergleichen Sie den generierten HTML-Code der beiden Alternativen. Achten Sie auf das versteckte Form-Feld.

Aufgabe 4.16

Ändern Sie das Suffix des Servlet-Mappings auf „.html“ ab, so dass in der URL-Zeile des Browsers für den Anwender der Eindruck entsteht, er würde statische HTML-Seiten aufrufen.

4.8.2 Die JSF-Konfiguration

JavaServer Faces sind als ein Framework konzipiert worden, bei dem fast alle – auch sehr zentrale Komponenten – austauschbar sein sollen. Dieses Ziel wurde erreicht. Man hat praktisch keine Komponente fest implementiert, sondern Komponenten sind durch alternative Implementierungen mit kompatiblem API austauschbar.

Ziel des Buches ist es, mit JavaServer Faces anspruchsvolle Benutzerschnittstellen zu entwickeln und aufzuzeigen, wie diese in eine Gesamtanwendung zu integrieren sind. Die Erweiterung von JavaServer Faces um mehr als – etwa – selbst entwickelte UI-Komponenten geht über dieses Ziel hinaus. Wir beschreiben im Folgenden die vollständigen Konfigurationsmöglichkeiten von JavaServer Faces daher nur bis zu der Ebene, die der Zielsetzung des Buches entspricht. Beispiele für Konfigurationen findet man über das ganze Buch verteilt. Wir wiederholen die Beispiele hier nicht, sondern verweisen auf sie. Der JSF-Anfänger kann diesen Abschnitt beim ersten Durcharbeiten getrost überspringen.

Die JSF-Konfiguration erfolgt wie die Servlet-Konfiguration in einer XML-Datei, deren Syntax für neuere JSF-Versionen durch ein XML-Schema definiert ist. Die zu verwendende Schema-Deklaration für JSF Version 2.0 ist:

```
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee \
        http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
    version="2.0">
```

Die Schema-Deklaration für Version 1.2 erhält man durch Ersetzen von „2_0“ in der vorletzten Zeile mit „1_2“ und Ersetzen von „2.0“ in der letzten Zeile mit „1.2“. Für die Versionen 1.0 und 1.1 sind DTDs statt Schemata zu verwenden, auf die wir aber nicht eingehen.

Wie bereits erwähnt, wollen wir keine komplette Darstellung der möglichen Konfigurationsoptionen geben, was der Umfang des Schemas automatisch ver-

bietet. Die Schema-Datei besteht aus über 2700 Zeilen, die in der Spezifikation enthaltene Textform des Schemas umfasst 61 Seiten. Wir wollen an dieser Stelle viele, aber nicht alle Konfigurationsoptionen möglichst anschaulich erläutern. Zu diesem Zweck greifen wir auf eine an reguläre Ausdrücke angelehnte Symbolik zurück, ohne den Anspruch, korrekte reguläre Ausdrücke formulieren zu wollen. So verwenden wir die Zeichen |, ? und * zur Kenzeichnung von Alternativen, optionalen Vorkommen und Wiederholung. Weitere Bedingungen werden im Text formuliert.

Für eine komplette Darstellung verweisen wir auf das Schema und das Werk von [vdV03] als Einführung in das Thema. Moderne IDEs unterstützen die Erstellung der Konfigurationsdatei unter Beachtung des Schemas und bieten Ihnen kontextsensitive Hilfen an, so dass Sie auch ohne Vorliegen des Schemas effizient arbeiten können. Wir empfehlen Ihnen trotzdem, diesen Abschnitt durchzuarbeiten, um ein Gefühl für die vielen Konfigurationsoptionen zu bekommen.

Nach dem oben dargestellten Schema ist das <faces-config>-Element die Wurzel der XML-Struktur. In diesem Element können 15 verschiedene Elemente enthalten sein, auf die wir im Folgenden kurz eingehen.

```
<faces-config>
    <absolute-ordering>?
    <application>*
    <behavior>*
    <component>*
    <converter>*
    <faces-config-extension>*
    <factory>*
    <lifecycle>*
    <managed-beans>*
    <navigation-rule>*
    <ordering>*
    <referenced-bean>*
    <render-kit>*
    <validator>*
```

Wir haben die Elemente alphabetisch sortiert dargestellt. Die Reihenfolge der Elemente ist jedoch beliebig. Entgegen der gewählten Darstellungsart ist eine abwechselnde Verwendung der Elemente erlaubt. Es kann also z. B. zunächst ein Konvertierer, dann ein Validierer und anschließend wieder ein Konvertierer definiert werden. Eine korrekte Darstellung der Syntax wäre jedoch zu aufwändig, so dass wir, wie bereits erwähnt, die aus regulären Sprachen bekannte Symbolik verwenden und diese umgangssprachlich verfeinern.

Bei der Erläuterung der einzelnen Elemente gehen wir nicht alphabetisch vor, sondern ordnen die Elemente inhaltlich an und gruppieren sie, wenn nötig.

Das `<application>`-Element enthält grundlegende Konfigurationen, etwa die unterstützten Lokalisierungen, das zu verwendende Render-Kit, den View-Handler und die Implementierungen zur Auflösung von Variablennamen der EL bzw. zur Auswertung von EL-Ausdrücken.

```
<application>
    <action-listener>*
    <application-extension>*
    <default-render-kit-id>*
    <defaultValidators>*
    <el-resolver>*
    <locale-config>*
    <message-bundle>*
    <navigation-handler>*
    <partial-traversal>*
    <property-resolver>*
    <resource-handler>*
    <state-manager>*
    <system-event-listener>*
    <variable-resolver>*
    <view-handler>*
```

Die einzelnen Elemente können mehrmals in verschiedener Reihenfolge innerhalb des `<application>`-Elements verwendet werden. Dies erscheint zunächst wenig sinnvoll, da z. B. beim View-Handler auf den ersten Blick nur eine Klasse als Implementierung verwendet werden kann. JSF definiert jedoch einen Delegationsmechanismus, der die Definition eines View-Handlers erlaubt, der bei der Methodenimplementierung auf den bis zu diesem Zeitpunkt aktuellen View-Handler zurückgreift. So kann eine Sequenz von View-Handlers definiert werden, die eigenen Implementierungen der View-Handler-Methoden (der abstrakten Klasse `ViewHandler` im Package `javax.faces.application`) liefern oder auf Methoden des zuvor definierten Handlers zurückgreifen. Der beschriebene Delegationsmechanismus gilt nicht nur für den View-Handler, sondern für alle angegebenen Elemente, bei denen er sinnvoll verwendbar ist. Den interessierten Leser verweisen wir auf den Abschnitt 11.4.6 *Delegation Implementation Support* der JSF-Spezifikation.

Die beschriebenen Sohn-Elemente des `<application>`-Elements machen JSF zu einem sehr flexiblen und erweiterbaren System. So wurde etwa Facelets vor der Version 2.0 von JSF mit Hilfe des `<view-handler>`-Elements als View-Handler definiert:

```
<application>
    <view-handler>
        com.sun.facelets.FaceletViewHandler
    </view-handler>
</application>
```

Mit der Version 2.0 wurden Facelets zum Default-View-Handler erhoben. Systeme wie JBoss Seam (Abschnitt 9.6) können die JSF-EL erweitern, indem die Auflösung von Variablen (`<variable-resolver>`) oder die Auswertung von Ausdrücken (`<el-resolver>`) der Expression-Language neu definiert werden. Da derartige Erweiterungen von JSF über die Zielsetzung des Buches hinausgehen, verzichten wir auf eine weitere Darstellung und widmen uns den weiteren Konfigurationsmöglichkeiten.

Das `<managed-bean>`-Element wird verwendet, um Managed Beans zu konfigurieren. Managed Beans werden automatisch erzeugt und initialisiert, wenn sie das erste Mal in einem JSF-EL-Ausdruck verwendet werden. Wir haben Managed Beans ausführlich in Abschnitt 4.3 diskutiert, so dass wir hier auf eine Darstellung verzichten

```
<managed-bean>
    <managed-bean-name>
    <managed-bean-class>
    <managed-bean-scope>
        (   <managed-property>* | 
            <map-entries> |
            <list-entries>
        )
    <managed-bean-extension>
```

Seit JSF 2.0 erlaubt die Annotation `@ManagedBean` die Deklaration einer Managed Bean. Mit dem Attribut `name` lässt sich explizit ein Name vergeben, ansonsten wird der Klassenname verwendet, dessen erster Buchstabe klein geschrieben wird. Da eine Klasse annotiert wird, entfällt deren explizite Nennung. Für die Deklaration des Scopes stehen die Annotationen `@ApplicationScoped`, `@SessionScoped`, `@ViewScoped`, `@RequestScoped`, `@NoneScoped` und `@CustomScoped` zur Verfügung. Properties bekommen über die Annotation `@ManagedProperty` Werte injiziert. Auch hier verweisen wir für eine ausführliche Darstellung auf Abschnitt 4.3.

Das `<referenced-bean>`-Element definiert einen Namen, unter dem eine Bean angesprochen werden kann, die nicht durch das JSF-Laufzeitsystem, sondern anderweitig erzeugt und für einen bestimmten Scope der Applikation bekannt gemacht wurde. Da keine Garantie besteht, dass das Objekt zur Laufzeit tat-

sächlich erzeugt wird, ist die Verwendung des `<referenced-bean>`-Elements für diesen Fall auszulegen.

```
<referenced-bean>
  <referenced-bean-name>
    <referenced-bean-class>
```

Das `<navigation-rule>`-Element definiert die Navigationsmöglichkeiten innerhalb einer JSF-Anwendung, d. h. nach welcher View welche anderen Views folgen können. Das Element ist ausführlich in Abschnitt 4.6 dargestellt. Beispiele finden sich über das ganze Buch verteilt.

```
<navigation-rule>
  <from-view-id>
    <navigation-case>*
      <navigation-rule-extension>*
```

Das `<component>`-Element registriert eine UI-Komponente (eine Unterklasse von `UIComponent`) in der JSF-Laufzeitumgebung. Wir gehen auf die Entwicklung eigener Komponenten mit Java nicht ein, so dass wir dieses Element nicht verwenden. Die Entwicklung eigener Komponenten auf der Basis von Facelets, die wir in Abschnitt 6.5 beschreiben, benötigt das `<component>`-Element nicht.

```
<component>
  <component-type>
  <component-class>
  <facet>*
  <attribute>*
  <property>*
  <component-extension>*
```

Das `<render-kit>`-Element wird verwendet, um neue Renderer dem Standard-HTML-Render-Kit hinzuzufügen oder ein neues Render-Kit zu erzeugen.

```
<render-kit>
  <render-kit-id>?
  <render-kit-class>?
  <renderer>*
  <client-behaviour-renderer>*
  <render-kit-extension>*
```

Das `<validator>`-Element definiert einen Bezeichner für eine Validiererkasse. Wir haben in Abschnitt 4.4.7 eine solche Klasse definiert und sie sowohl mit dem `<validator>`-Element als auch per Annotation dem JSF-Laufzeitsystem bekannt gemacht.

```
<validator>
  <validator-id>
  <validator-class>
  <attribute>*
  <property>*
  <validator-extension>*
```

Das **<converter>**-Element definiert einen Bezeichner für eine Konvertiererklasse. Ein Beispiel für einen Konvertierer finden Sie in Abschnitt 4.4.4.

```
<converter>
  (<converter-id> | <converter-for-class>)
  <converter-class>
  <attribute>*
  <property>*
  <converter-extension>*
```

Das **<phase-listener>**-Element erlaubt die Registrierung von Phase-Listernern, die vor oder nach einer der sechs Phasen des Bearbeitungsmodells einer JSF-Anfrage aufgerufen werden. Ein Beispiel für Phase-Listener finden Sie in Abschnitt 4.5.6. Über das **<lifecycle-extension>**-Element können Framework-Entwickler Erweiterungen des Lebenszyklus injizieren.

```
<lifecycle>
  <phase-listener>*
  <lifecycle-extension>*
```

Alle Klassen des JSF-Kernsystems werden entweder über einen Plug-In-Mechanismus oder über Fabriken erzeugt. Mit dem **<factory>**-Element können neue Fabriken bekannt gemacht werden. Sie werden diese Fabriken nur benötigen, wenn Sie in die innere Funktionsweise von JavaServer Faces eingreifen und z. B. einen eigenen Exception-Handler realisieren wollen.

```
<factory>
  <application-factory>*
  <exception-handler-factory>*
  <external-context-factory>*
  <faces-context-factory>*
  <partial-view-context-factory>*
  <lifecycle-factory>*
  <view-declaration-language-factory>*
  <tag-handler-delegate-factory>*
  <render-kit-factory>*
  <visit-context-factory>*
  <factory-extension>*
```

Neben der Möglichkeit, eigene Fabriken für zentrale JSF-Komponenten zu definieren und so JavaServer Faces zu erweitern, ist die Möglichkeit der Verwendung von Erweiterungselementen hervorzuheben. Das zuletzt aufgeführte `<factory>`-Element erlaubt etwa als direktes Sohnelement die `<factory-extension>`. Dieses Element und alle weiteren, in Tabelle 4.13 vorgestellten Erweiterungselemente erlauben die anwendungsspezifische Erweiterung von JavaServer Faces ohne definiertes XML-Schema. Sie können also valide, aber ansonsten *beliebige* XML-Strukturen für Ihre Erweiterungen verwenden.

Tabelle 4.13: Erweiterungselemente

Übergeordnetes Element	Erweiterungselement
application	application-extension
attribute	attribute-extension
component	component-extension
converter	converter-extension
faces-config	faces-config-extension
facet	facet-extension
factory	factory-extension
lifecycle	lifecycle-extension
managed-bean	managed-bean-extension
navigation-rule	navigation-rule-extension
property	property-extension
render-kit	render-kit-extension
renderer	renderer-extension
validator	validator-extension

Mit Version 2.0 von JavaServer Faces wurde eine Möglichkeit geschaffen, bei mehreren Konfigurationsdateien eine Reihenfolge zwischen diesen zu definieren. Bisher haben wir lediglich die in Abschnitt 4.8.1 dargestellte Möglichkeit zur Verwendung mehrerer Konfigurationsdateien dargestellt. Wenn jedoch wieder verwendbare Anwendungsteile oder Komponentenbibliotheken eigene Konfigurationsmöglichkeiten benötigen, ist der Weg über den Servlet-Deployment-Deskriptor ein wenig geeigneter und aufwändiger Mechanismus. JSF sucht daher auch in JAR-Archiven nach Konfigurationsdateien, was wir gleich näher erläutern. Die Reihenfolge von Konfigurationsdateien in dieser Menge war bisher nicht festgelegt. Seit Version 2.0 existiert eine Möglichkeit, sowohl eine relative als auch eine vollständige Ordnung zu definieren. Bevor wir dies

darstellen können, benötigen wir aber noch eine genauere Analyse der Möglichkeiten, wie JSF Konfigurationsdateien behandelt.

Bei Anwendungsstart werden zunächst alle Anwendungs-Ressourcen durchsucht. In der Regel sind dies die JAR-Dateien der Anwendung im Verzeichnis /WEB-INF/lib. Falls im META-INF-Verzeichnis einer JAR-Datei eine Datei mit Namen faces-config.xml existiert oder Dateien, deren Namen auf .faces-config.xml enden, werden diese zu aktuellen Ressource-Dateien.

Im zweiten Schritt werden alle im Servlet-Deployment-Deskriptor unter dem Kontextparameter javax.faces.CONFIG_FILES genannten Dateien zu Resource-Dateien, wie wir es in Abschnitt 4.8.1 beschrieben.

Im dritten Schritt wird die Datei /WEB-INF/faces-config.xml als Ressource-Datei aufgenommen, sofern sie existiert.

Die in den beiden ersten Schritten bestimmten Dateien wollen wir im Folgenden *Ressource-Dateien* nennen. Die im dritten Schritt bestimmte Datei nennen wir *Konfigurationsdatei*.

Zur Definition von Reihenfolgen werden als direkte Sohnknoten des <faces-config>-Elements die Elemente <name>, <absolute-ordering> und <ordering> verwendet. Das Element <name> wird zur Identifizierung einer Datei verwendet. Mit <absolute-ordering> wird eine vollständige, mit <ordering> eine relative Ordnung definiert.

Wir beginnen mit der Darstellung der vollständigen Ordnung. Das <absolute-ordering>-Element kann nur in der Konfigurationsdatei verwendet werden. In ihm wird die Reihenfolge der Dateien durch explizite Nennung der definierten Namen festgelegt. Das folgende Beispiel besteht aus drei Dateien.

```
<faces-config><!-- /WEB-INF/faces-config.xml -->
  <name>conf</name>
  <absolute-ordering>
    <name>A</name>
    <name>B</name>
  </absolute-ordering>
  ...
</faces-config>

<faces-config>
  <name>A</name>
  ...
</faces-config>

<faces-config>
  <name>b</name>
  ...
</faces-config>
```

In obigem Beispiel erhält man die Reihenfolge A, B, conf, da die Konfigurationsdatei immer als letzte verwendet wird. Alle nicht explizit genannten Konfigurationen werden ignoriert. Falls diese Konfigurationen nicht ignoriert werden sollen, ihre Position in der Reihenfolge jedoch nicht relevant ist, verwendet man das <others>-Element, also z. B.

```
<faces-config><!-- /WEB-INF/faces-config.xml -->
<name>conf</name>
<absolute-ordering>
  <name>A</name>
  <name>B</name>
  <others />
</absolute-ordering>
...
</faces-config>
```

Zur Definition einer relativen Ordnung werden das <ordering>-Element sowie die Elemente <before>, <after> und <others> verwendet. Mit <before> und <after> wird die relative Position einer Konfiguration bzgl. einer anderen Konfiguration festgelegt. Im folgenden Beispiel wird die Konfiguration A nach B durchgeführt.

```
<faces-config>
<name>A</name>
<ordering>
  <after>
    <name>B</name>
  </after>
</ordering>
...
</faces-config>
```

Es ist offensichtlich, dass man mit diesem Mechanismus nur eine partielle Ordnung definiert, so dass es der JSF-Implementierung freisteht, aus der Menge der möglichen Ordnungen eine auszuwählen.

Wir schließen unsere unvollständigen Ausführungen zur Reihenfolgedefinition mit der Anmerkung, dass eine JSF-Implementierung zirkuläre und anderweitig inkonsistente Deklarationen erkennen und mit einem Fehler quittieren muss.

Ebenfalls eine Neuerung von JSF 2.0 sind Behaviors, also Verhalten, die an Objekte gebunden werden können. Behaviors wurden eingeführt, um Ajax-Funktionalitäten in JSF 2.0 einfacher integrieren zu können, sind aber nicht auf Ajax beschränkt. Wir widmen dem Thema Ajax das Kapitel 7, benötigen die folgenden Elemente jedoch nicht, da wir keine zusätzlichen Verhaltensmuster definieren.

```
<behavior>
  <behavior-id>
  <behavior-class>
  <attribute>*
  <property>*
  <behavior-extension>*
```

4.8.3 XML-Konfigurationsdatei versus Annotationen

Die Motivation zur Einführung von Annotationen in JSF 2.0 war die Vereinfachung der Konfiguration und die Vermeidung der sogenannten „XML-Hölle“, also der ausufernden Verwendung von XML-basierten Konfigurationen. Da Annotationen aber im Java-Quell-Code verwendet werden, ist die Deployment-spezifische Konfiguration mit dem Ändern des Quell-Codes und einem, zumindest in Teilen, durchlaufenden Entwicklungszyklus verbunden. Deployment-spezifische Konfigurationen sind daher nach wie vor sinnvollerweise in XML vorzunehmen. Es ergibt sich also eventuell eine Überlagerung bzw. Konkurrenz der Konfiguration über XML oder Annotationen, die aufgelöst werden muss.

Generell gilt, dass eine Konfiguration in der JSF-Konfigurationsdatei eine Konfiguration per Annotation überschreibt. Wird also etwa eine Managed Bean `bean` über eine Annotation erzeugt

```
@ManagedBean(name = "bean")
public class ...
```

aber auch über XML

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

so gilt die XML-Konfiguration.

Weiterhin ist zu klären, wo und wann das JSF-Laufzeitsystem nach Klassen mit Konfigurations-Annotationen zu suchen hat. Hierzu wird zunächst das boolesche Attribut `metadata-complete` des `<faces-config>`-Elements in der Datei `/WEB-INF/faces-config.xml` betrachtet. Ist dieses auf `true` gesetzt, unterbleibt das Suchen nach Klassen mit Konfigurations-Annotationen völlig. Ist das Attribut nicht vorhanden oder explizit auf `false` gesetzt, wird nach entsprechenden Klassen gesucht. Dabei werden

- Klassen direkt im Verzeichnis `/WEB-INF/classes` und
- Klassen, die sich in Jar-Dateien im Verzeichnis `/WEB-INF/lib` befinden,

untersucht. Letztere allerdings nur, wenn die Jar-Datei eine Datei /META-INF/faces-config.xml oder Dateien, die mit faces-config.xml enden, enthält.



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele zur Konfiguration sind im Projekt *jsf-im-detail* enthalten.

4.9 Client-Ids und Komponenten-Ids

4.9.1 Id-Arten und Namensräume

JSF ist ein server-seitiges Komponenten-Framework. Um mit einzelnen Komponenten arbeiten zu können, muss jede Komponente eindeutig identifizierbar sein. Die Java-internen Mittel zur Identifikation von Objekten reichen hierzu nicht aus, da nach der Wiederherstellung des Komponentenbaums Komponenten an anderen Stellen der VM liegen können und somit eine andere Id haben. Daher enthält jede Komponente eine Id in Form eines Strings.

Das Abbild des Komponentenbaums auf dem Client benötigt ebenfalls eine Möglichkeit, die einzelnen Komponenten zu identifizieren. XML, DOM, JavaScript und CSS besitzen aber andere syntaktische Regeln, so dass hier eventuell andere Ids zu verwenden sind. Die JSF-Spezifikation regelt die Abbildung nicht explizit bis ins letzte Detail, so dass eine JSF-Implementierung und der konfigurierte Renderer gewisse Freiheiten haben, z. B. bei der Generierung impliziter Ids, auf die wir später eingehen.

Bei der Erzeugung einer Komponente auf dem Server erhält diese eine Id zugewiesen. Die Id ist als Property (`getId()`, `setId()`) in der Klasse `UIComponent` definiert. Sie wird dem `id`-Attribut des JSF-Tags entnommen, falls es verwendet wurde. Ansonsten wird sie generiert. Um die Eindeutigkeit von Ids zu gewährleisten, werden Namensräume eingeführt, die die Eindeutigkeit erzwingen. Das Interface `javax.faces.component.NamingContainer` stellt einen Container dar, der die Eindeutigkeit für alle Kinder garantiert. Implementierende Klassen des Interface sind `UIData`, `UIForm` und `UINamingContainer`, alle aus dem Package `javax.faces.component`. Eindeutigkeit bedeutet hier, dass innerhalb einer `HtmlForm`-Komponente (eine Unterklasse von `UIForm`, erzeugt durch das `<h:view>`-Tag) jede Kindkomponente eine andere Id haben muss, falls keine weiteren Container verwendet werden.

Listing 4.47 zeigt eine JSF-Seite, die zur Erläuterung von Komponenten- und Client-Ids dienen soll. Man erkennt, dass alle HTML-Tags explizit das `id`-Attribut setzen.

Listing 4.47: Einfache JSF-Seite mit expliziten Ids (`demo.xhtml`)

```
<f:view>
    <h:form id="form">
        <h:outputText id="outtext"
                      value="Bitte geben Sie einen Wert ein!" />
        <h:panelGrid id="panel" columns="2">
            <h:outputLabel id="outlabel" for="input"
                           value="Eingabe: " />
            <h:inputText id="input" value="#{demoHandler.eingabe}"/>
            <h:commandButton id="button" value="Und ab ..." ...>
            <h:panelGroup id="dummy" />
        </h:panelGrid>
    </h:form>
</f:view>
```

Die gewählten Ids werden identisch auch als Komponenten-Ids in Java verwendet. Die Client-Ids werden mit dem Präfix der Container-Id, in diesem Fall `form`, versehen. Tabelle 4.14 auf der nächsten Seite stellt die drei Bezeichner einander gegenüber.

Die erste Spalte gibt den Quell-Code der JSF-Seite wieder. Die Einrückung der ersten Spalte stellt die Tiefe im Komponentenbaum dar, auf die in den beiden anderen Spalten aus Platzgründen verzichtet wurde. Die zweite Spalte zeigt die jeweiligen interessanten Ausschnitte der generierten HTML-Seite. Man erkennt, dass die Ids aller Komponenten außer dem Formular den Präfix `form` besitzen. Weiterhin erkennt man bei `<form>`- und den beiden `<input>`-Elementen das Attribut `name`, das als Wert die Id des jeweiligen Elements enthält. Die Ids werden verwendet, um Elemente im DOM, die Namen, um das Formular und die Formulareingaben zu identifizieren. Die dritte Spalte gibt schließlich die Id der Java-Komponenten wieder, wobei in der ersten Zeile die Komponentenklasse und in der zweiten die Id aufgeführt ist. Die Ids in JSF und Java sind identisch, so dass in Java direkt mit den JSF-Ids gearbeitet werden kann. Die Ids in HTML unterscheiden sich jedoch, so dass bei der Verwendung von JavaScript die generierten Ids bekannt sein müssen. Wir zeigen dies im Abschnitt 4.9.2 beispielhaft.

JSF erzwingt nicht die Vergabe expliziter Ids und generiert, falls keine Ids explizit angegeben wurden, implizite Ids. Zur Demonstration dieses Verhaltens

Tabelle 4.14: Client- und Komponenten-Ids (explizit)

JSF-Id	HTML-Id	Java-Id
<h:form id="form"	<form id="form" name="form"	HTMLForm form
<h:outputText id="outtext"	<span id="form:outtext"	HTMLOutputText outtext
<h:panelGrid id="panel"	<table id="form:panel"	HTMLPanelGrid panel
<h:outputLabel id="outlabel"	<label id="form:outlabel"	HTMLOutputLabel outlabel
<h:inputText id="input"	<input type="text" id="form:input" name="form:input"	HTMLInputText input
<h:commandButton id="button"	<input type="submit" id="form:button" name="form:button"	HTMLCommandButton button
<h:panelGroup id="dummy"	<span id="form:dummy"	HTMLPanelGroup dummy

entfernen wir aus der JSF-Seite `demo.xhtml` (Listing 4.47 auf der vorherigen Seite) alle Ids mit Ausnahme der Id für die Texteingabe. Diese Id darf nicht entfernt werden, da sie im `<h:outputLabel>` referenziert wird. Tabelle 4.15 auf der nächsten Seite präsentiert die dann existierenden Ids.

Man erkennt in der zweiten und dritten Spalte die generierten Ids, aber auch ihre unterschiedliche Schreibung. Außer den Elementen `<form>` und `<input type="text">` haben alle anderen Elemente ihre Ids verloren. Die beiden ``-Elemente sind sogar ganz verschwunden.

Das vorgestellte Konzept von Client- und Komponenten-Ids wird vom JSF-Anwendungsentwickler für die Versionen vor JSF 2.0 in der Regel selten benötigt. Für Systementwickler im Bereich JavaScript und Ajax ist dieses Wissen jedoch unverzichtbar. Mit der Einführung von Ajax in Version 2.0 hat sich dies geändert, und auch der JSF-Anwendungsentwickler benötigt nun ein entsprechendes Grundlagenwissen. Bevor wir im nächsten Abschnitt ein Beispiel mit einem Komponentenzugriff per Client-Id ohne das Ajax-API von JSF 2.0 entwickeln, wollen wir aber noch einmal kurz auf die bereits erwähnten Namensräume eingehen. Das Kapitel 7 ist ausschließlich Ajax gewidmet und erlaubt eine Überarbeitung des Beispiels mit dem Resultat einer einfacheren Implementierung.

Tabelle 4.15: Client- und Komponenten-Ids (implizit)

JSF-Id	HTML-Id	Java-Id
<h:form	<form id="j_idt4" name="j_idt4"	HTMLForm j_idt-4
<h:outputText		HTMLOutputText j_idt-5
<h:panelGrid	<table	HTMLPanelGrid j_idt-6
<h:outputLabel	<label	HTMLOutputLabel j_idt-7
<h:inputText id="input"	<input type="text" id="j_idt4:input" name="j_idt4:input"	HTMLInputText input
<h:commandButton	<input type="submit" name="j_idt4:j_idt8"	HTMLCommandButton j_idt-8
<h:panelGroup		HTMLPanelGroup j_idt-9

Durch das Konzept der Namens-Container können in einer JSF-Seite zwei und mehr Komponenten dieselbe Id haben. Sie müssen sich jedoch in verschiedenen Namens-Containern befinden. Im folgenden Beispiel haben die beiden Input-Komponenten dieselbe Id, was zulässig ist, weil sie sich jeweils in einem anderen Formular befinden.

```
<f:view>
    <h:form id="form1">
        <h:inputText id="eingabe" ...
        ...
    </h:form>
    ...
    <h:form id="form2">
        <h:inputText id="eingabe" ...
        ...
    </h:form>
</f:view>
```

Aufgabe 4.17

Das Projekt *jsf-im-detail* enthält die JSF-Seiten `demo.xhtml` und `demo-wo-ids.xhtml`, die erste Seite explizite Ids für alle Komponenten, die zweite Seite nur eine Id für die Input-Komponente. Laden Sie beide Seiten, und studieren Sie den erzeugten HTML-Code. Betätigen Sie die Schaltflächen beider Seiten, und beobachten Sie das Log. Es werden die Ids der Komponenten ausgegeben.

Aufgabe 4.18

Die Komponenten der JSF-Seite `demo-wo-ids.xhtml` im Projekt `jsf-im-detail` besitzen keine Ids. Schreiben Sie eine Methode, die nach der Textausgabe „Bitte geben Sie einen Wert ein!“ sucht und den Text in „Bitte geben Sie einen Text ein!“ ändert.

4.9.2 Client- und server-seitige Programmierung mit Ids

Mit Kenntnis der Regeln für Client- und Komponenten-Ids können diese sowohl in der client-seitigen als auch in der server-seitigen Programmierung verwendet werden. Client-seitig finden vor allem JavaScript und CSS Verwendung, server-seitig wird Java verwendet.

Wir beginnen mit dem Client. Es soll ein Formular erstellt werden, das ein Label und ein Eingabefeld besitzt und durch Betätigen einer Schaltfläche abgeschickt wird. Wenn sich die Maus über dem Label befindet, soll ein Hilfetext im Eingabefeld angezeigt werden. Dies kann durch JavaScript-Event-Handler für die HTML-Tags `onmouseover` und `onmouseout` realisiert werden. Eine Einführung in JavaScript-basierte HTML-Attribute finden Sie im Anhang B.2 auf Seite 386. Eine vollständige Beschreibung von JavaScript findet man z. B. im Buch von Flanagan [Fla01].

Listing 4.48 zeigt das Formular der Seite `input.xhtml`, die das Gewünschte leistet. Für das Label werden zwei JavaScript-Event-Handler definiert. Das Attribut `onmouseover` (Zeilen 3/4) enthält den aufzurufenden Code, wenn sich die Maus über dem Label befindet. Hier wird dem Eingabefeld der Text „*Ihr Name*“ zugewiesen. Bitte beachten Sie, dass der Zeilenumbruch im String am Ende der Zeile 3 dem Layout geschuldet ist und nicht im Original stehen darf. Das Attribut `onmouseout` (Zeilen 5/6) enthält den aufgerufenen Code, wenn sich die Maus aus dem Bereich des Labels herausbewegt. Das Attribut `onfocus` in Zeile 9 erläutern wir später. Abbildung 4.23 auf der nächsten Seite zeigt im oberen Fenster die Darstellung des Eingabefelds, wenn sich die Maus über dem Label befindet. Der Text wurde *nicht* in das Eingabefeld eingegeben.

Listing 4.48: Einfache Eingabe mit Hilfefunktion in JavaScript (`input.jsp`)

```
1 <h:form id="meinFormular">
2   <h:outputLabel for="eingabe" value="Name: "
3     onmouseover="document.forms.meinFormular
4       ['meinFormular:eingabe'].value = ' Ihr Name'"
5     onmouseout= "document.forms.meinFormular
6       ['meinFormular:eingabe'].value = ''"
7   />
```

```
8   <h:inputText id="eingabe" value="#{inputHandler.eingabe}"  
9     onfocus="this.style.backgroundImage = 'url(null)'"  
10    />  
11    <br/>  
12    <h:commandButton action="#{inputHandler.abschicken}"  
13      value="Abschicken" />  
14  </h:form>
```



Abbildung 4.23: Darstellung der Seite `input.xhtml` (Listing 4.48)

Als weitere Anforderung an das Formular definieren wir eine Eingabeüberprüfung. Ohne das Attribut `required="true"` zu verwenden, soll eine leere Eingabe erkannt werden und der Benutzer eine entsprechende Meldung erhalten. Auch soll für die Meldung nicht auf das Tag `<h:message>` zurückgegriffen werden. Wir realisieren die Anforderung server-seitig mit Java und CSS. Listing 4.49 zeigt die Klasse `InputHandler`, die in der Action-Methode `abschicken()` diese Anforderung erfüllt.

Listing 4.49: Die Klasse `InputHandler`

```
1  public class InputHandler {  
2  
3    private String eingabe;
```

```
4
5     public String abschicken() {
6         UIViewRoot view =
7             FacesContext.getCurrentInstance().getViewRoot();
8         HtmlInputText eingabeKomponente =
9             (HtmlInputText) view.findComponent("meinFormular:eingabe");
10
11        if (eingabe.equals("")) {
12            //eingabeKomponente.setStyle("background-color: red;");
13            eingabeKomponente.setStyle(
14                "background-image: \
15                 url(../../images/eingabe-erforderlich.png);");
16        } else {
17            eingabeKomponente.setStyle("background-image: null;");
18        }
19        return "";
20    }
```

Dazu wird in den Zeilen 8/9 zunächst die `HtmlInputText`-Komponente ausfindig gemacht. Die View einer Seite ist die Wurzel des Komponentenbaums. Die Klasse `UIViewRoot` enthält daher die Methode `findComponent()`, die diesen Baum durchsucht und die Komponente findet, deren Client-Id als Parameter angegeben wurde.

In Zeile 11 wird getestet, ob die Eingabe leer war. Wenn ja, wird in den Zeilen 13–15 mit der Methode `setStyle()` über eine CSS-Deklaration das Hintergrundbild gesetzt. Das Bild `eingabe-erforderlich.png` erhält in roter Schrift den Text „Eingabe erforderlich“. Dies ist im unteren Fenster von Abbildung 4.23 zu erkennen. Alternativ könnte in der auskommentierten Zeile 12 die Hintergrundfarbe auf Rot gesetzt werden. Falls nach dieser Anzeige ein Text eingegeben wird, muss man den Hintergrund wieder in den ursprünglichen Zustand versetzen, er darf also kein zugewiesenes Bild besitzen. Dies wird in Zeile 17 implementiert.

Die Anwendung hat eine unangenehme Eigenschaft. Wird nach einer leeren Eingabe die Meldung als Hintergrund des Eingabefeldes angezeigt, so erfolgt das Schreiben in das Feld auf einem textuellen Hintergrund. Der eingegebene Text ist dann sehr schlecht zu lesen. Daher sollte der Hintergrund neutralisiert werden, wenn das Eingabefeld den Fokus erhält. Dies ist wiederum etwas, das client-seitig zu geschehen hat und mit den schon bekannten JavaScript-Event-Handlern realisiert wird. Das Attribut `onfocus` der Eingabekomponente in Listing 4.48 auf Seite 184 leistet genau dies. Hier ist anzumerken, dass in JavaScript das Minuszeichen einen Operator darstellt und daher in Bezeichnern nicht verwendet werden kann. Die CSS-Eigenschaft `background-image`

(und alle anderen CSS-Schlüsselwörter mit enthaltenem Minuszeichen) muss daher an die JavaScript-Syntaxregeln angepasst werden. Dies geschieht durch Entfernen des Minuszeichens und Großschreibung des Folgezeichens, so dass `backgroundImage` entsteht.

Die Realisierung der Anforderungen ist zugegebenermaßen aufwändig und hätte auch einfacher erfolgen können. Wir wollten aber die Kombination von JavaServer Faces, Java, JavaScript und CSS demonstrieren, ohne auf die Möglichkeiten von JSF 2.0 mit Ajax einzugehen.

Zum Abschluss der Ausführungen zu Ids soll das Attribut `prependId` des `<h:form>`-Tags Erwähnung finden. Der Wert dieses booleschen Attributs bestimmt, ob die Id des Formulars als Präfix der Ids der enthaltenen Komponenten verwendet wird. Der Default-Wert ist `true`, entspricht also dem bisher von uns dargestellten Verhalten. Im Abschnitt 7.3.4 entwickeln wir ein Beispiel im Zusammenhang mit Ajax, das ohne den Präfix arbeitet.

Aufgabe 4.19

Nicht nur die Texteingabe auf einem textuellen Hintergrund ist unschön, sondern auch die Anzeige des Hilfetextes, wenn sich die Maus über dem Label befindet. Erweitern Sie die Anwendung, so dass bei der Anzeige des Hilfetextes ebenfalls ein neutraler Hintergrund angezeigt wird.



Projekt
Die in diesem Abschnitt beschriebenen Code-Beispiele zu Client- und Komponenten-Ids sind im Projekt *jsf-im-detail* enthalten.

4.10 Verwendung allgemeiner Ressourcen

Unter Ressourcen versteht man allgemeine Betriebs- oder Hilfsmittel, die man zur Realisierung von Anwendungen benötigt. Java-EE bezeichnet z. B. Persistenzkontakte oder JDBC-Datenquellen als Ressourcen. JSF selbst verwendet den Begriff z. B. für Dateien und Klassen zur Lokalisierung, wie wir dies in Abschnitt 4.7.1 gesehen haben. Mit der Version 2.0 wurde der Begriff von Ressourcen verallgemeinert, und es wurden Mechanismen eingeführt, um Ressourcen in JSF-Anwendungen standardisiert verwenden zu können.

Unter derartigen Ressourcen versteht man Dateien für Graphiken, JavaScript oder Stylesheets. Für deren Verwendung wurden mit JSF 2.0 neue Tags definiert, bestehende Tags mit neuen Attributen versehen und das JSF-API erwei-

tert. Auf der API-Ebene ist hier vor allem die Klasse `ResourceHandler` zu nennen, die den Zugriff auf Ressourcen regelt.

4.10.1 Einfache Ressourcen

Einfache – das heißt nicht versionierte – Ressourcen werden durch ihre Resourcennamen identifiziert. Dabei werden Graphik-, Stylesheet- und JavaScript-Dateien über die Tags `<h:graphicImage>`, `<h:outputStylesheet>` und `<h:outputScript>` in eine JSF-Seite eingebunden. Das Tag `<h:graphicImage>` existierte bereits vor Version 2.0 und erhielt für die Verwendung von Ressourcen neue Attribute. Die Tags `<h:outputStylesheet>` und `<h:outputScript>` wurden mit JSF 2.0 eingeführt. Graphiken werden in der gerenderten Seite an der über `<h:graphicImage>` definierten Stelle platziert. Stylesheets und Skriptdateien werden in der gerenderten Seite an anderen Stellen platziert, als im Quell-Code verwendet. Die Spezifikation spricht von *Relocation*. Wir behandeln die Tags `<h:outputStylesheet>` und `<h:outputScript>` daher gesondert in Abschnitt 4.10.3.

Ressource-Dateien werden im Wurzelverzeichnis der Web-Applikation im Verzeichnis `resources` abgelegt. Falls Ressource-Dateien in Jar-Dateien gepackt werden, so ist das Verzeichnis `/META-INF/resources` zu verwenden. Um Bibliotheken von Ressourcen zu erstellen, ordnet man diese in einfachen Verzeichnissen an. Zur beispielhaften Verwendung existieren im Verzeichnis `resources/images` die Dateien `girl-1.png`, `girl-2.png` und `girl-3.png`.

Die erste Datei kann dann ganz konventionell mit

```
<h:graphicImage value="/resources/images/girl-1.png" />
```

eingebunden werden und wird in der gerenderten Antwort mit

```

```

als Link auf die Graphikdatei unter Anfügen des Anwendungsnamens dargestellt. Die Ressourcen-Verwaltung von JSF 2.0 erlaubt mit der vordefinierten EL-Variablen `resource` eine alternative Verwendung, die vom Dateisystem abstrahiert. Die Variable `resource` ist eine Map, die über den Dateinamen mit vorangestelltem und durch Doppelpunkt getrennten Bibliotheksnamen indiziert wird, in unserem Beispiel also mit `images`.

```
<h:graphicImage value="#{resource['images:girl-2.png']}> />
```

Die gerenderte Antwort macht von der Ressourcen-Schnittstelle Gebrauch und gibt den Bibliotheksnamen als expliziten Parameter an:

```

```

Dieses Ergebnis wird auch über die direkte Verwendung des HTML-Tags erreicht.

```

```

Die dritte Alternative zur Verwendung des `<h:graphicImage>`-Tags verwendet die in JSF 2.0 neu hinzugekommenen Attribute `library` und `name`. Mit ihnen werden Bibliotheks- und Ressourcename getrennt angegeben.

```
<h:graphicImage library="images" name="girl-3.png"/>
```

Abbildung 4.24 zeigt noch einmal die drei Alternativen in einer übersichtlichen Darstellung mit Quell-Code und Resultat.

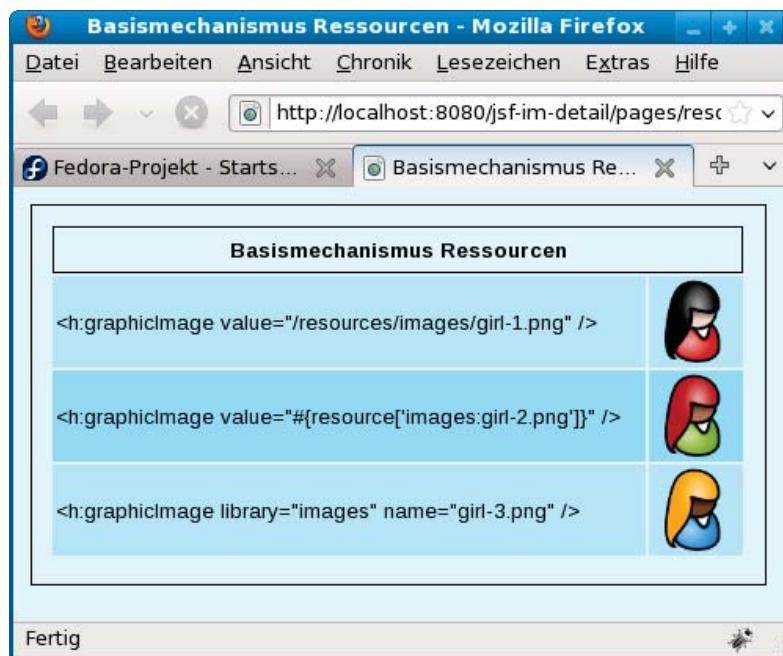


Abbildung 4.24: Alternativen zur Ressourceneinbindung

4.10.2 Versionierte Ressourcen und Ressourcen-Bibliotheken

JSF 2.0 unterstützt die Verwendung lokalisierter und versionierter Ressourcen. Die Verwendung einer Ressource erfolgt über einen *Ressourcen-Identifikator*, der nach dem folgenden Schema aufgebaut ist:

```
[locale/] [libName/] [libVersion/] resourceName [/resourceVersion]
```

Man erkennt die entsprechenden Bestandteile für die Lokalisierung, die Bibliothek sowie die Ressource selbst. Sowohl Bibliothek als auch Ressourcename sind versionierbar. Werden die Lokalisierung sowie die beiden Versionsangaben nicht verwendet, wird die Default-Lokalisierung (siehe Abschnitt 4.7.1) und die höchste Versionsnummer, falls vorhanden, verwendet. Wird lediglich der unverzichtbare Ressourcename angegeben, erhält man die einfache Form eines Ressource-Identifikators, wie wir ihn in Abschnitt 4.10.1 angewandt haben.

Die Versionierung geschieht mit ganzen Zahlen, die durch einen Unterstrich getrennt werden. Wird eine Version für die Bibliothek angegeben, so muss auch eine Bibliothek angegeben werden. Das zunächst komplex erscheinende Schema wird anhand eines Beispiels schnell anschaulich. Abbildung 4.25 zeigt das Ressourcen-Verzeichnis mit der Bibliothek `images`. Diese ist in der Version 1.0 und 1.1 vorhanden und wird im Dateisystem jeweils durch die Verzeichnisse `1_0` und `1_1` repräsentiert. Beide Versionen enthalten die Graphik-Datei `boy.png` allerdings in unterschiedlichen Versionen. Die Graphik-Datei wird als Verzeichnis repräsentiert und die Versionen durch die jeweiligen Dateien in diesen Verzeichnissen, wobei der Dateiname nach obigem Schema aufgebaut ist.

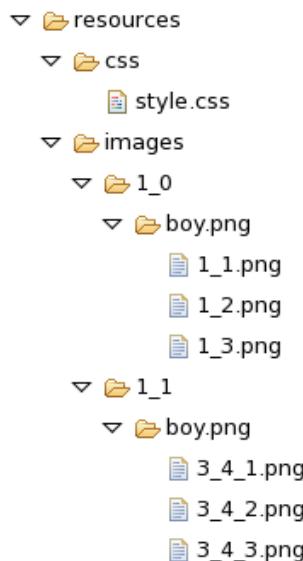


Abbildung 4.25: Versionierte Bibliotheksdateien

Wir haben in diesem Beispiel auf eine Lokalisierung verzichtet, da wir auf Internationalisierung und Lokalisierung bereits im Abschnitt 4.7 eingegangen sind. Soll lokalisiert werden, so muss das zur Lokalisierung verwendete Verzeichnis

vor den Bibliotheksverzeichnissen stehen. Für eine deutsche Lokalisierung also z. B. `resources/de/images/....` Im Message-Bundle muss der verwendete Verzeichnisnamen dann für den Wert `javax.faces.resource.localePrefix` angegeben werden, im Beispiel also

```
javax.faces.resource.localePrefix=de
```

Nachdem die versionierten Graphiken installiert sind, können sie nun in JSF-Seiten verwendet werden. Die bereits bekannten Attribute `library` und `name` der `<h:graphicImage>`-Komponente lassen sich wie in Abschnitt 4.10.1 ohne Versionsangaben verwenden. Es wird dann die Graphik mit der höchsten Versionsnummer der Bibliothek sowie der Graphik selbst verwendet. Für

```
<h:graphicImage library="images" name="boy.png" />
```

ist dies die Datei `3_4_3.png`. Versionen können auch explizit angegeben werden. Dabei entspricht die zu verwendende Syntax den Verzeichnis- und Dateinamen. Soll etwa die Graphik in der Version 1.1 aus der Bibliothek 1.0 verwendet werden, sind die folgenden Attributwerte zu verwenden.

```
<h:graphicImage library="images/1_0" name="boy.png/1_1.png" />
```

Weitere Beispiele sind in Abbildung 4.26 dargestellt. Links von der Graphik sind die beiden Attribute, rechts davon die verwendete Datei angegeben.

4.10.3 Positionierung von Ressourcen

Graphiken werden mit dem `<h:graphicImage>`-Tag am Ort der Verwendung dargestellt. Ressourcen wie JavaScript und CSS sind in der Regel im Kopf einer HTML-Seite, eventuell aber auch im Rumpf zu positionieren. Dies kann durch einfache Tags realisiert werden. Problematisch wird eine derartige Verwendung aber bei der Erstellung eigener Komponenten oder bei der Verwendung von Komponentenbibliotheken. Wie lassen sich JavaScript- oder CSS-Dateien in solchen Komponenten an der richtigen Stelle positionieren?

Mit der Version 2.0 führt JSF die Möglichkeit zur Positionierung von Ressourcen ein. Die beiden Tags `<h:outputScript>` und `<h:outputStylesheet>` repräsentieren externe JavaScript- bzw. Stylesheet-Ressourcen. Das Tag `<h:outputStylesheet>` kann an einer beliebigen Stelle verwendet werden, das Stylesheet selbst wird in der gerenderten Antwortseite immer im Kopf platziert. Bei JavaScript-Dateien kann zwischen einer Platzierung im Kopf oder im Rumpf bzw. dem Formular der Seite ausgewählt werden, da dies die korrekte Funktionsweise beeinflusst. Die Auswahl erfolgt über das Attribut `target`, das die Werte `head`, `body` und `form` erlaubt. Um die gewählte Platzierung zu realisieren, müssen die entsprechenden Tags `<h:head>`, `<h:body>` und `<h:form>` in der Seite vorhanden sein.



Abbildung 4.26: Versionierte Ressourcen

Die CSS-Datei für das Beispiel versionierter Ressourcen wird mit folgendem Code in die Seite integriert:

```
<h:head>
    <title>Versionierung von Bibliotheken</title>
</h:head>
<body>
    <h:outputStylesheet library="css" name="style.css" />
    <h:panelGrid ...>
```

Der generierte Code hat folgendes Aussehen:

```
<head>
    <link type="text/css" rel="stylesheet"
          href="/jsf-im-detail/javax.faces.resource/
              style.css.jsf?ln=css" />
    <title>Versionierung von Bibliotheken</title>
</head>
<body>
    <table>
```

Da Stylesheets im Seitenkopf platziert werden, muss man `<h:head>` als entsprechendes Tag verwenden. Die Verwendung des Standard-HTML-Tags `<head>` führt zu einem Fehler.

Wenn im obigen Beispiel eine JavaScript-Datei im Seitenrumpf verlinkt werden soll, so muss statt des HTML-Tags `<body>` das JSF-Tag `<h:body>` verwendet werden. Wir verzichten hier auf ein Beispiel und verweisen den Leser auf Abschnitt 7.2.1 auf Seite 240, in dem wir ein solches Beispiel entwickeln.

Sollen Ressourcen nicht direkt im Dateisystem existieren, sondern etwa aus Gründen der Wiederverwendbarkeit in Jars gepackt werden, so ist innerhalb des Jars `/META-INF/resources` als Verzeichnis zu wählen. In der Referenzimplementierung ist etwa die JavaScript-Bibliothek `jsf.js` zur Ajax-Unterstützung (siehe Abschnitt 7.2.1) in der Bibliothek `javax.faces` untergebracht. Die Datei `jsf-impl.jar` der Referenzimplementierung enthält daher die Dateien

```
/META-INF/resources/javax.faces/jsf-uncompressed.js  
/META-INF/resources/javax.faces/jsf.js
```

Beide Dateien enthalten den identischen JavaScript-Code, einmal in lesbarer, das andere Mal in optimierter, für Menschen weniger gut lesbarer Form.

Zum Abschluss unserer Ausführungen zur Verwendung allgemeiner Ressourcen sei noch die Möglichkeit zur Ressourcen-Verwendung in Java erwähnt. Da diese Art der Verwendung in der Regel nur bei der Entwicklung eigener Komponenten insbesondere der entsprechenden Renderer benötigt wird, wir in diesem Buch die Entwicklung eigener Komponenten jedoch nicht behandeln, erfolgt dies hier nur der Vollständigkeit halber. Falls Sie eigene Komponenten entwickeln, können Sie Ressourcen mit den Annotationen `@ResourceDependency` und `@ResourceDependencies` für diesen Renderer deklarieren. JSF liefert für jede mit diesem Renderer gerenderte Seite die entsprechende Ressource aus.

Aufgabe 4.20

Überzeugen Sie sich davon, dass die JavaScript-Ressource-Dateien in Ihrem Release der Referenzimplementierung vorhanden und bis auf die Komprimierung identischen Inhalts sind.

	<p>Projekt</p> <p>Die in diesem Abschnitt beschriebenen Code-Beispiele zur Verwendung allgemeiner Ressourcen sind im Projekt <i>jsf-im-detail</i> enthalten.</p>
---	---

4.11 JSTL-Bibliotheken

Zur Erweiterung von JavaServer Pages wurde die *JavaServer Pages Standard Tag Library* (JSTL) entworfen, deren Spezifikation unter [URL-JSR52] eingesehen werden kann. Erstaunlicherweise enthalten Facelets in JSF 2.0 Teile der JSTL, während JSP als VDL diese nicht enthalten. Dies ist umso erstaunlicher, als JSTL nicht mehr aktiv weiterentwickelt wird und das letzte Release aus dem Jahr 2006 stammt. Wir gehen hier der Vollständigkeit halber mit einigen Beispielen auf JSTL ein und beschreiben JSFs JSTL-Bibliothek in Kapitel 5. Insgesamt raten wir von einer Verwendung von JSTL ab, da in der Regel eine entsprechende Implementierungsalternative mit reinen JSF-Mitteln existiert.

JavaServer Faces, genauer Facelets, enthalten eine teilweise Implementierung der JSTL-Kernbibliothek sowie eine vollständige Implementierung der JSTL-Funktionsbibliothek. Die Kernbibliothek besteht aus den Tags `<c:if>`, `<c:when>`, `<c:choose>` und `<c:otherwise>` zur Definition von Fallunterscheidungen, aus dem Iterations-Tag `<c:forEach>` sowie den Tags `<c:catch>` und `<c:set>` zur Verarbeitung von Exceptions und zur Wertzuweisung an Variablen. Insbesondere die beiden letztgenannten Tags sollten unserer Meinung nach nicht verwendet werden, da JSF eine eigene Exception-Behandlung vorsieht, ja sogar die Definition eines anwendungsdefinierten Exception-Handler erlaubt und mit Managed Beans, deren Scopes und der EL mächtige Alternativen bietet.

Ein sinnvolles Tag der JSTL ist `<c:forEach>`, mit dem Iterationen sehr einfach zu realisieren sind. Der folgende Code-Ausschnitt zeigt die Deklaration des XML-Namespace der Kernbibliothek (Endung `core` mit Präfix `c`) und das Erzeugen von Tabellenzeilen, die ausschließlich aus einem hochgezählten Zahlenwert bestehen.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:c="http://java.sun.com/jsp/jstl/core">
...
<h:panelGrid columns="1">
    <f:facet name="header">
        Hochzählen in Tabellenzeilen
    </f:facet>
    <c:forEach var="i" begin="1" end="10">
        #{i}
    </c:forEach>
</h:panelGrid>
...
```

Dieses Schema kann z. B. für die Erzeugung von Umrechnungstabellen für verschiedene Währungen oder, wie im folgenden Beispiel, für einfache Quadratzahlen verwendet werden.

```
<h:panelGrid columns="2">
    <f:facet name="header">Hochzählen von Quadraten</f:facet>
    <c:forEach var="i" begin="1" end="10">
        <h:outputText value="#{i}" />
        <h:outputText value="#{i * i}" />
    </c:forEach>
</h:panelGrid>
```

Da als weitere Option die Schrittweite der Iteration über das `<c:forEach>`-Attribut `step` angegeben werden kann, sind auch kompliziertere Tabellen vorstellbar.

Eine ebenfalls sinnvolle Verwendungsmöglichkeit besteht in der Kombination mit der erweiterten Expression-Language, die Methoden mit Parametern erlaubt. Wir haben dies in Abschnitt 4.2.5 vorgestellt. Das folgende Beispiel erzeugt Schaltflächen, deren Action-Methoden-Aufrufe systematisch mit Integer-Werten parametrisiert sind.

```
<c:forEach var="i" begin="1" end="10">
    <h:commandButton action="#{jstlHandler.action(i)}" ... />
</c:forEach>
```

Die entsprechende Action-Methode verwendet einen Integer-Parameter:

```
@ManagedBean
public class JstlHandler {
    ...
    public String action(int i) {
        ...
    }
    ...
}
```

Da die Attribute `begin`, `end` und `step` als Werte beliebige EL-Ausdrücke erlauben, liegen sinnvolle Einsatzmöglichkeiten auf der Hand.

Kommen wir nun zur zweiten JSTL-Bibliothek, der Funktionsbibliothek. Diese enthält vor allem Funktionen zur String-Manipulation, z. B. den Test, ob Strings bestimmte Teil-Strings enthalten, die Längenberechnung von Strings oder die Umwandlung in Groß- oder Kleinbuchstaben. In Kapitel 5 stellen wir alle Funktionen vor.

Als erstes Beispiel wollen wir Strings in ihre Großbuchstabendarstellung umwandeln. Der folgende Code-Ausschnitt beginnt wieder mit der Deklaration des XML-Namensraums der Funktionsbibliothek (Endung `functions` mit Präfix `fn`).

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions">
    ...
    <h:panelGrid columns="1" rowClasses="odd,even">
        <c:forEach var="item" items="#{jstlHandler.array}">
            #{fn:toUpperCase(item)}
        </c:forEach>
    </h:panelGrid>
```

Man erkennt hier, dass das `<c:forEach>`-Tag zwei weitere Attribute hat: `var` und `items`. Diese entsprechen den bereits bekannten Attributen `var` und `value` des `<h:dataTable>`-Tags, definieren also eine Iterationsvariable und binden einen Iterationsausdruck. Im Beispiel wird wieder die Managed Bean `JstlHandler` mit dem Property `array` verwendet.

```
@ManagedBean
public class JstlHandler {

    private String[] array;

    public JstlHandler() {
        array = new String[] {"eins", "zwei", "drei"};
    }

    public String[] getArray() {
        return array;
    }
    ...
}
```

Das Beispiel gibt also die Strings EINS, ZWEI und DREI aus.

Als letztes Beispiel verwenden wir das `<c:if>`-Tag, um die Umwandlung in Großbuchstaben vom aktuellen String-Wert abhängig zu machen. Das `<c:if>`-Tag der JSTL-Bibliothek kennt kein Else-Äquivalent, wie viele andere Sprachen. Daher muss der Else-Zweig explizit mit der negierten Bedingung in einem weiteren `<c:if>` getestet werden:

```
<c:forEach var="item" items="#{jstlHandler.array}">
    <c:if test="#{not fn:endsWith(item, 'ei')}">
        #{item}
    </c:if>
    <c:if test="#{fn:endsWith(item, 'ei')}">
        #{fn:toUpperCase(item)}
    </c:if>
</c:forEach>
```

Strings, die auf „ei“ enden, werden in Großbuchstaben umgewandelt, andere nicht. Als Ergebnis erhalten wir eins, ZWEI und DREI.

Aufgabe 4.21

Überarbeiten Sie die Beispiele, so dass Sie auf die Verwendung von `<c:if>` und `fn:toUpperCase()` verzichten können. Die Verwendung von `<c:forEach>` ist weiterhin erlaubt.



Projekt

Die in diesem Abschnitt beschriebenen Code-Beispiele zu den JSTL-Bibliotheken sind im Projekt *jstl* enthalten.

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Kapitel 5

Die UI-Komponenten

Wir haben bisher den Begriff *Komponente* sehr häufig verwendet, meist jedoch nicht präzise im Sinne von JavaServer Faces. Nachdem wir im letzten Kapitel die Strukturen und Funktionalitäten einer JSF-Implementierung detailliert darstellten, ist es nun an der Zeit, dies zu korrigieren. Wir zitieren zunächst aus der Spezifikation:

“A JSF user interface component is the basic building block for creating a JSF user interface. A particular component represents a configurable and reusable element in the user interface, which may range in complexity from simple (such as a button or text field) to compound (such as a tree control or table).”

JSF-Komponenten sind allgemeine UI-Komponenten (User Interface). Sie repräsentieren Konzepte wie *Eingabe*, *Ausgabe* und *Befehl*, wie sie in jeder Benutzerschnittstelle vorkommen. JSFs UI-Komponenten existieren auf dem Server und sind *darstellungsunabhängig*, d. h. an keine Render-Technologie gebunden. Die für den Client erzeugte Darstellung ist ein Spiegelbild der auf dem Server existierenden Komponenten. Die JSF-Spezifikation lässt beliebige Darstellungen zu, fordert jedoch von jeder JSF-konformen Implementierung, dass mindestens HTML als Ergebnis der Render-Phase erzeugt wird. Bis zur Version 1.2 wurden JSPs als Seitenbeschreibungssprache und HTML 4.01 als Render-Ergebnis vorausgesetzt. Mit Version 2.0 kamen Facelets als Seitenbeschreibungssprache und XHTML 1.0 als Render-Ergebnis hinzu und wurden zum Default erhoben.

Um von der Seitenbeschreibungssprache zu abstrahieren, wurde der Begriff der *View Declaration Language* (VDL) oder, seltener, der *Page Declaration Language* (PDL) eingeführt. Bei der Entwicklung von Anwendungen mit Ja-

vaServer Faces wird nun eine VDL verwendet, und zwar in Form einer Tag-Bibliothek. Die UI-Komponenten selbst sowie weitere zentrale Elemente der JSF-Anwendung, z. B. der Kontext einer Anfrage oder das die Anwendung repräsentierende Application-Objekt, sind über das JSF-API verwendbar. Die VDL-Implementierung besitzt ebenfalls ein öffentliches API, das aber in der Regel äußerst selten verwendet wird. Als Grundlage dient letztendlich das Servlet-API, das ebenfalls für die Anwendung zur Verfügung steht. Hinzu kommen weitere Bibliotheken, so dass sich eine Anwendungsarchitektur ergibt, deren Aufbau Abbildung 5.1 verdeutlicht.

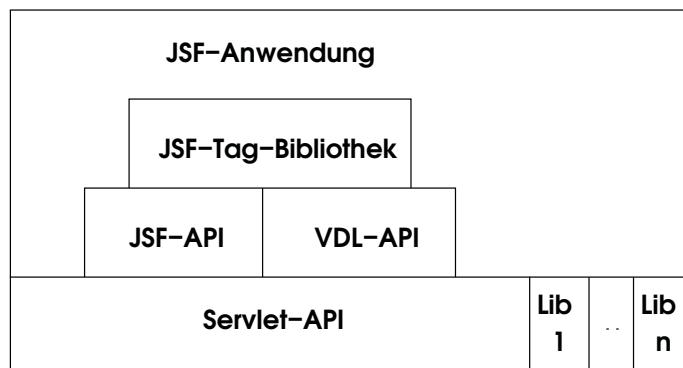


Abbildung 5.1: Schichtenarchitektur einer JSF-Anwendung

Wir stellen in diesem Kapitel die Tag-Bibliothek – genauer: sechs Tag-Bibliotheken – und teilweise deren Verbindungen mit dem JSF-API vor. Die Bibliotheken umfassen die HTML- und die Kernbibliothek in der Facelets-Version der VDL sowie die mit Version 2.0 hinzugekommenen Bibliotheken für Facelets selbst, für zusammengesetzte Komponenten und für JSTL (Kern und Funktionen). Wir verzichten auf die Darstellung der JSP-Version der VDL.

5.1 Die Standardkomponenten

Sämtliche Standardkomponenten sind Klassen des Package `javax.faces.component`. Die Darstellung der Vererbungshierarchie dieser Klassen erfolgt in Abbildung 5.2. Man erkennt verschiedene Komponentenarten, z. B. Befehle (`Command`), Einfachselektion (`SelectOne`), Mehrfachselektion (`SelectMany`) und Eingaben (`Input`). Dies sind Konzepte, wie sie in jeder Oberfläche vorkommen. Sie sind, wie bereits erwähnt, darstellungsunabhängig. Wir charakterisieren jede Komponente stichpunktartig.

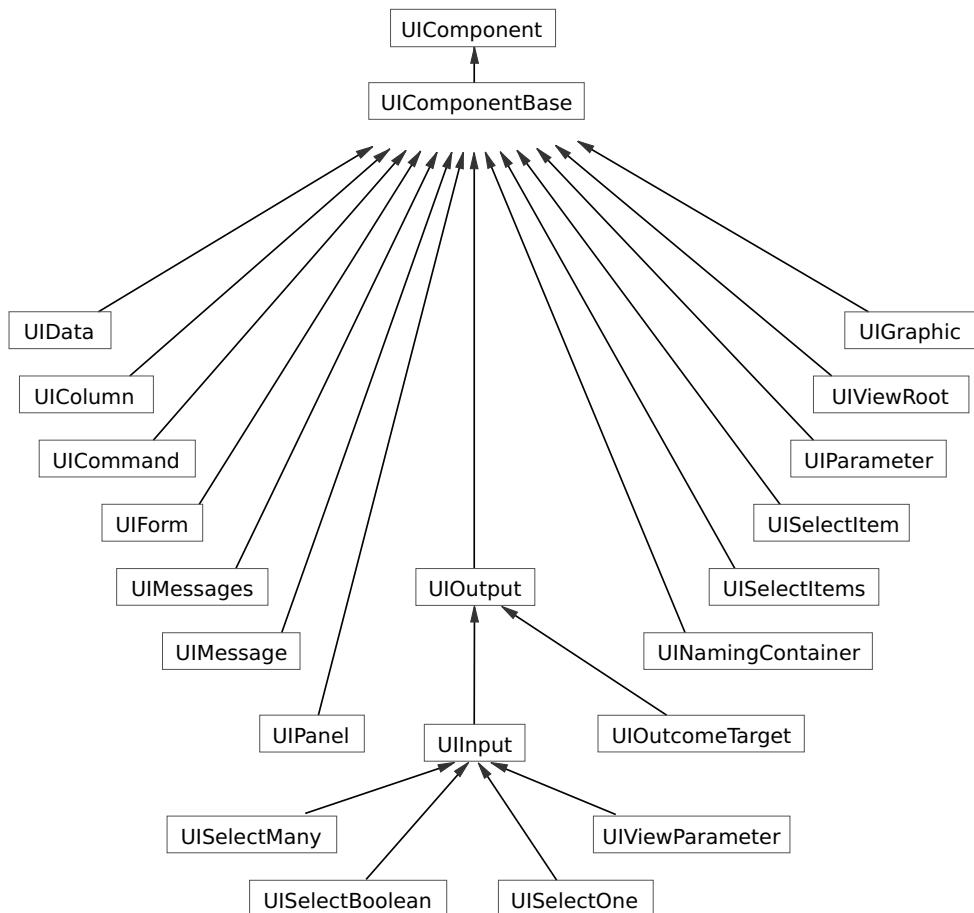


Abbildung 5.2: Die UI-Komponenten

- **UIComponent**
Abstrakte Oberklasse aller Komponenten.
- **UIComponentBase**
Abstrakte Klasse mit Implementierungen aller abstrakten Methoden von **UIComponent**.
- **UIColumn**
Repräsentiert eine Spalte in einer Tabelle. Muss in eine **UIData**-Komponente eingebettet sein.
- **UICommand**
Repräsentiert einen Befehl, der durch einen Benutzer ausgelöst werden kann.

■ **UIData**

Eine datengebundene Komponente, die durch die Datenquelle iteriert, um zeilenweise Daten darzustellen. Benötigt eine oder mehrere **UIColumn**-Komponenten.

■ **UIForm**

Ein Eingabeformular, das andere Komponenten enthalten kann. Eingabekomponenten *müssen* in einem **UIForm** enthalten sein.

■ **UIGraphic**

Anzeige einer Grafik, die durch eine URL identifiziert wird.

■ **UIInput**

Eine Komponente, die Daten entgegennimmt und anzeigt.

■ **UIMessage**

Anzeige von Meldungen für eine bestimmte Komponente.

■ **UIMessages**

Anzeige aller Meldungen; sowohl komponentenbezogen als auch anwendungsbezogen.

■ **UINamingContainer**

Basisklasse für Klassen mit Namensraumeigenschaften.

■ **UIOutcomeTarget**

Komponente zur Erzeugung eines Get-Requests für eine URL.

■ **UIOutput**

Anzeige von Daten, die nicht benutzeränderbar sind.

■ **UIParameter**

Repräsentiert einen Parameter einer übergeordneten Komponente.

■ **UIPanel**

Ordnet enthaltene Komponenten an.

■ **UISelectBoolean**

Eingabe und Anzeige eines booleschen Wertes.

■ **UISelectItem**

Repräsentiert ein Element oder eine Elementgruppe. Wird gewöhnlich von **UISelectMany** und **UISelectOne** verwendet.

■ **UISelectItems**

Repräsentiert mehrere Elemente oder Elementgruppen. Wird gewöhnlich von **UISelectMany** und **UISelectOne** verwendet.

- **UISelectMany**
Anzeige von verschiedenen Elementen und Auswahl von mehreren dieser Elemente durch den Benutzer.
- **UISelectOne**
Anzeige von verschiedenen Elementen und Auswahl eines dieser Elemente durch den Benutzer.
- **UIViewRoot**
Repräsentiert die ganze View. Sie enthält alle anderen Komponenten einer Seite, hat selbst aber keine sichtbare Darstellung.
- **UIViewParameter**
Repräsentiert eine Bindung zwischen einem Request-Parameter und einem Modell- oder UIViewRoot-Property.

Die Oberklasse der JSF-Standardkomponenten ist **UIComponent**. Die in dieser Klasse abstrakt definierten Methoden werden von allen Komponenten implementiert. Beispiele sind etwa die Methode `decode()`, die aus einer JSF-Anfrage die Request-Parameter extrahiert (siehe Abschnitt 4.1.2, *Übernahme der Anfragewerte*), die zur entsprechenden Komponente gehört, oder die Methode `getChildren()`, die eine Liste aller Unterkomponenten eines Containers liefert.

Um die Entwicklung eigener Komponenten zu vereinfachen, implementiert die Klasse **UIComponentBase**, eine direkte Unterklasse von **UIComponent**, die abstrakten Methoden der Oberklasse. Falls Sie eigene Komponenten entwickeln, verwenden Sie in der Regel diese Klasse als Oberklasse. Die weiteren Klassen realisieren die verschiedenen UI-Konzepte, wie Ein- und Ausgabe. Aus der Reihe heraus treten die beiden Komponenten **UIViewRoot** und **UIForm**. **UIViewRoot** ist die Wurzel des in Abschnitt 4.1.1, *Wiederherstellung des Komponentenbaums*, erwähnten Komponentenbaums. Die Komponente **UIForm** enthält alle für Benutzerinteraktionen empfänglichen Komponenten. Wird die Komponente nach HTML gerendert, so entspricht dies einem HTML-<form>. Alle Eingabekomponenten, die außerhalb eines **UIForms** liegen, werden daher nicht in die HTTP-Anfrage aufgenommen und erreichen den Server nicht.

Als Entwickler JSF-basierter Anwendungen verwendet man die vorgestellten Komponentenklassen eher selten. Sollten Sie jedoch eigene Komponenten entwickeln, werden diese in der Regel von **UIComponentBase** abgeleitet.

5.2 Render-Sätze

JavaServer Faces sind zunächst eine darstellungsunabhängige Technik. Die oben dargestellten UI-Komponenten können mit beliebigen Darstellungsmechanismen versehen werden. Der Default-Mechanismus, den jede JSF-Implementierung enthalten muss, ist die Darstellung mit HTML- bzw. XHTML-Seiten. Damit Darstellungen auswechselbar sind, gibt es in JavaServer Faces das Konzept austauschbarer Renderer. Jede Komponente bekommt einen Renderer zugeordnet. Eine Komponente kann auch durch mehrere Renderer dargestellt werden. Die Funktionalität der Komponente bleibt dieselbe, die Darstellung variiert. Ein Beispiel ist die Befehlskomponente, die in HTML als Schaltfläche oder als Link dargestellt werden kann. Alle Renderer für ein bestimmtes Ausgabemedium werden schließlich zu einem Render-Satz (Render-Kit) zusammengefasst.

Alle Render-relevanten Klassen sind im Package `javax.faces.render` zusammengeführt. Dies sind die Klassen `Renderer`, `RenderKit`, `RenderKitFactory` und `ResponseStateManager`. Sie definieren die Schnittstelle für den Zugriff auf Renderer. Wir gehen auf die Erstellung eigener Renderer nicht ein.

Als Abschluss sei noch erwähnt, dass der Standard-Render-Satz als Id die String-Konstante "HTML_BASIC" hat. Diese kann über die Methode

```
Application.getDefaultRenderKitId();
```

erfragt werden. Die Id des aktuellen Render-Satzes einer JSF-Seite liefert die Methode

```
UIViewRoot.getRenderKitId();
```

zurück.

5.3 Die JSF-Standard-Bibliotheken

Der Standard-Render-Satz für JSF erzeugt HTML, genauer XHTML 1.0 Transitional. Wir haben ihn in unseren bisherigen Beispielen verwendet. Die vorgestellten Komponentenarten und die verschiedenen Render-Arten werden systematisch zu JSF-Tag-Namen kombiniert. Tabelle 5.1 zeigt Beispiele für die Namensgebung der JSF-Tags, die aus der Komponentenart und der Renderer-Art zusammengesetzt werden. An die Komponentenart wird die Renderer-Art angehängt. Die Groß/Kleinschreibung folgt den Regeln für Java-Methodenbezeichner, d. h. die Tags beginnen mit Kleinbuchstaben, angehängte Namen folgen mit Großbuchstaben. Eine Ausnahme ist die Form-Komponente mit dem Form-Renderer. Hier ist das Tag `<h:form>` und nicht `<h:formForm>`. Weite-

Tabelle 5.1: Beispiele der Namensgebung für JSF-Tags

JSF-Tag	Standardkomponente	Komponentenart	Renderer-Art
<h:commandButton>	HtmlCommandButton	Command	Button
<h:commandLink>	HtmlCommandLink	Command	Link
<h:selectOneMenu>	HtmlSelectOneMenu	SelectOne	Menu
<h:selectManyMenu>	HtmlSelectManyMenu	SelectMany	Menu

re Ausnahmen sind <h:column> und <h:message(s)>, sowie die mit JSF 2.0 eingeführten Tags <h:button> und <h:link>. Die Klassen, die die Standard-JSF-Tags implementieren, nennt man die *HTML-Standardkomponenten*. Ihre Klassennamen ergeben sich aus dem Tag-Namen mit vorangestelltem *Html* unter Berücksichtigung der Java-Namensregeln. Alle HTML-basierten Komponentenklassen sind im Package `javax.faces.component.html` enthalten.

Die HTML-Standardkomponenten generieren XHTML 1.0 Transitional mit Cascading-Style-Sheets (CSS) und JavaScript. Als Seitenbeschreibungssprache wird Facelets verwendet, also ebenfalls XHTML. Die JSF-Tags werden daher über XML-Namensräume deklariert. Als *JSF-Standard-Bibliotheken* werden üblicherweise die HTML- und die Kernbibliothek bezeichnet. Wir nehmen hier als Beispiel die Facelets-Bibliothek ebenfalls auf. Eine JSF-Seite beginnt insofern in der Regel mit

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:ui="http://java.sun.com/jsf/facellets">
```

Dies deklariert die HTML- und die Kernbibliothek sowie die Facelets-Bibliothek. Die in diesem Beispiel verwendeten Präfixe sind frei wählbar. Sie sollten jedoch dem Beispiel folgen und *h*, *f* und *ui* verwenden. Wir sind bisher immer so verfahren und verwenden diese Präfixe auch im Sachverzeichnis. Falls Sie im Internet nach JSF-Beispielen suchen, werden Sie in der Regel ebenfalls diese Präfixe vorfinden.

Die JSF-Tags erlauben die Verwendung eines Großteils der zur jeweiligen Komponente passenden HTML-Attribute. Die HTML-Tabellen-Attribute *cellpadding* und *cellspacing* können etwa bei tabellengenerierenden Tags wie <h:panelGrid> und <h:dataTable> verwendet werden. Sinnvoller ist hier jedoch die Verwendung der JSF-eigenen Attribute *columnClasses* und *row-*

Classes mit entsprechendem CSS. Auch auf JavaScript basierende dynamische Attribute können verwendet werden. Dies sind z.B. `onmouseover` und `onmouseout`.

Im Folgenden stellen wir alle JSF-Tags der Standardbibliotheken vor. Wir beschränken uns jedoch auf eine Beschreibung der Funktionalität. Sie sollten diese Beschreibungen einmal durcharbeiten, um einen Überblick über den Funktionsumfang der Bibliothek zu erhalten. Eine detaillierte Darstellung der kompletten Syntax sowie eine Übersicht über die HTML-Attribute erfolgt in Anhang B. Dieser Anhang dient als Nachschlagehilfe bei der konkreten Anwendungsentwicklung.

5.4 Die HTML-Bibliothek

Die HTML-Bibliothek umfasst alle Tags, die UI-Komponenten in die Zielsprache HTML rendern. Tabelle 5.2 beschreibt diese kurz.

Tabelle 5.2: Tags der HTML-Bibliothek

Funktion	JSF-Tag	Beschreibung
Formular	<code><h:form></code>	Bindet Eingabekomponenten einer Seite in ein Formular ein. Es sind mehrere <code><h:form></code> in einer Seite erlaubt.
Eingabe	<code><h:inputHidden></code>	Ein nicht sichtbares Eingabefeld (<code>type="hidden"</code>)
	<code><h:inputSecret></code>	Ein Eingabefeld mit maskierter Eingabe (<code>type="password"</code>)
	<code><h:inputText></code>	Einfaches Eingabefeld (<code>type="text"</code>)
	<code><h:inputTextarea></code>	Mehrzeilige Eingabe
Auswahl	<code><h:selectBoolean-Checkbox></code>	Boolesche Checkbox
	<code><h:selectMany-Checkbox></code>	Gruppe von Checkboxen
	<code><h:selectMany-Listbox></code>	Auswahl mehrerer Elemente einer Liste
	<code><h:selectManyMenu></code>	Auswahl mehrerer Elemente eines Drop-Down
	<code><h:selectOne-Listbox></code>	Auswahl eines Elements einer Liste

Tabelle 5.2: Tags der HTML-Bibliothek (Fortsetzung)

Funktion	JSF-Tag	Beschreibung
	<h:selectOneMenu>	Auswahl eines Elements eines Drop-Down
	<h:selectOneRadio>	Auswahl einer Alternative; Alternativen dargestellt als Radio-Buttons
Befehl	<h:commandButton>	Eine Schaltfläche mit optionaler Möglichkeit der Bindung an eine Action-Methode
	<h:commandLink>	Ein Link mit optionaler Möglichkeit der Bindung an eine Action-Methode
Get-Request	<h:button>	Eine Schaltfläche, die einen Get-Request mit Anfrageparameter erzeugt
	<h:link>	Ein Link, der einen Get-Request mit Anfrageparameter erzeugt
Gruppierung	<h:panelGrid>	Eine Tabelle von Komponenten mit der Möglichkeit der Definition von Kopf- und Fußzeilen
	<h:panelGroup>	Gruppierung von Komponenten zur Verwendung innerhalb anderer Komponenten oder als Möglichkeit der einheitlichen Formatierung
Tabelle	<h:dataTable>	Eine Datentabelle mit Bindung an eine Datenquelle und der Möglichkeit der Definition von Kopf- und Fußzeilen
	<h:column>	Eine Tabellenspalte zur Verwendung innerhalb einer <h:dataTable>
Ausgabe	<h:outputFormat>	Ausgabe von parametrisiertem Text
	<h:outputLabel>	Eine Ausgabe zur Kennzeichnung eines Eingabefeldes
	<h:outputLink>	Ein Hyperlink, der mit keiner Benutzeraktion verknüpft ist
	<h:outputText>	Einfache textuelle Ausgabe; Formatierung durch CSS

Tabelle 5.2: Tags der HTML-Bibliothek (Fortsetzung)

Funktion	JSF-Tag	Beschreibung
Nachrichten	<h:message>	Fehlermeldungen oder Informationen für ein bestimmtes Eingabefeld
	<h:messages>	Fehlermeldungen oder Informationen für eine Seite; die Meldungen können komponenten- oder anwendungs- bezogen sein
Graphik	<h:graphicImage>	Darstellung einer Grafik, die durch eine URL identifiziert wird
Verwendung von Ressourcen	<h:body>	Repräsentiert HTML-Body als Ziel für Ressourcen
	<h:head>	Repräsentiert HTML-Head als Ziel für Ressourcen
	<h:outputScript>	Einfügen von JavaScript im Head oder Body
	<h:outputStylesheet>	Einfügen von CSS im Head

5.5 Die Kernbibliothek

Die Kernbibliothek umfasst alle Tags, die von keinem Render-Satz abhängig sind. Sie besitzen daher auch keine Darstellung, sondern sind rein funktionale Komponenten.

Tabelle 5.3: Tags der Kernbibliothek

Funktion	JSF-Tag	Beschreibung
View	<f:view>	Repräsentiert die gesamte View; enthält alle Komponenten der Seite.
	<f:subview>	Repräsentiert eine Teil-View.
Listener	<f:actionListener>	Registriert einen Action-Listener.
	<f:setProperty-ActionListener>	Registriert einen Action-Listener für ein Property.
	<f:valueChange-Listener>	Registriert einen Value-Change-Listener.
	<f:phaseListener>	Registriert einen Phase-Listener.
	<f:event>	Registriert einen System-Event-Listener.

Tabelle 5.3: Tags der Kernbibliothek (Fortsetzung)

Funktion	JSF-Tag	Beschreibung
Konvertierer	<f:converter>	Bindet einen allgemeinen Konvertierer an die übergeordnete Komponente.
	<f:convertDateTime>	Bindet den Standard-Datumskonvertierer an die übergeordnete Komponente.
	<f:convertNumber>	Bindet den Standard-Zahlenkonvertierer an die übergeordnete Komponente.
Ajax	<f:ajax>	Registriert einen Ajax-Handler.
Validierung	<f:validateDoubleRange>	Bindet einen Bereichsvalidierer für Double-Werte an die übergeordnete Komponente.
	<f:validateLongRange>	Bindet einen Bereichsvalidierer für Long-Werte an die übergeordnete Komponente.
	<f:validateLength>	Bindet einen Validierer bzgl. der Länge der Eingabe an die übergeordnete Komponente.
	<f:validator>	Bindet einen allgemeinen Validierer an die übergeordnete Komponente.
	<f:validateBean>	Validierung wird an Bean-Validation-API delegiert.
	<f:validateRequired>	Validierer, der Eingabewert erzwingt.
	<f:validateRegex>	Validierer, der regulären Ausdruck verwendet.
Auswahl-elemente	<f:selectItem>	Bindet ein Auswahllement ein.
	<f:selectItems>	Bindet mehrere Auswahllemente ein.
Internationalisierung	<f:loadBundle>	Lädt ein Resource-Bundle und vergibt Bezeichner dafür.
Facette	<f:facet>	Definiert eine Facette als Kind eines Containers, z.B. die Spaltenüberschrift einer Tabelle.
Parameter	<f:param>	Definiert einen Parameter für das umgebende Action-Element.
	<f:attribute>	Definiert ein Attribut für das umgebende Action-Element.

Tabelle 5.3: Tags der Kernbibliothek (Fortsetzung)

Funktion	JSF-Tag	Beschreibung
	<f:metadata>	Definiert Medadaten für eine View mit Hilfe von <f:viewParam>.
	<f:viewParam>	Definiert Parameter für View.
Unveränderte Ausgabe	<f:verbatim>	Ermöglicht die Verwendung von (Nicht-JSF-) Tags, die von der Verarbeitung ausgeschlossen werden.

5.6 Die Facelets-Bibliothek

Die Facelets-Bibliothek wurde mit JSF 2.0 in den Standard aufgenommen. Sie enthält vor allem Möglichkeiten zur template-basierten Definition von Seiten und ermöglicht damit die einheitliche Seitengestaltung einer Anwendung. Im Gegensatz zur HTML- und Kern-Bibliothek haben wir die Facelets-Bibliothek bisher noch nicht verwendet, so dass beim Überblick über die einzelnen Tags sicher zunächst einige Fragen offen bleiben. Wir haben dem Thema Facelets das Kapitel 6 gewidmet, das diese Fragen beantworten wird.

Tabelle 5.4: Tags der Facelets-Bibliothek

Funktion	JSF-Tag	Beschreibung
Templating	<ui:composition>	Definiert eine Komposition mit optionalem Template und ignoriert Umgebung.
	<ui:define>	Definiert einen Seitenbereich, der in ein Template eingesetzt wird.
	<ui:insert>	Fügt Inhalt in ein Template ein.
	<ui:include>	Inkludiert eine JSF-Seite.
	<ui:param>	Parameterübergabe für JSF-Seite oder Template.
	<ui:decorate>	Definiert eine Komposition mit optionalem Template, ohne Umgebung zu ignorieren.
	<ui:component>	Definiert Komposition ohne Template-Mechanismus und ignoriert Umgebung.

Tabelle 5.4: Tags der Facelets-Bibliothek (Fortsetzung)

Funktion	JSF-Tag	Beschreibung
	<ui:fragment>	Definiert Komposition ohne Template-Mechanismus ohne Umgebung zu ignorieren.
Kommentierung	<ui:remove>	Entfernt eingeschlossenen Seitenbereich und ermöglicht Kommentar.
Debugging	<ui:debug>	Öffnet das Debug-Fenster.
Iteration	<ui:repeat>	Iteriert über Collection.

5.7 Die Composite-Component-Bibliothek

Die Composite-Component-Bibliothek erlaubt die einfache Definition eigener Komponenten. Die Bibliothek wird üblicherweise mit dem Präfix `composite` über die folgende Namensraum-Deklaration eingeführt:

```
<... xmlns:composite="http://java.sun.com/jsf/composite" />
```

Eigene Komponenten werden über Schnittstelle und Implementierung realisiert, so dass die beiden wichtigsten Tags `<composite:interface>` und `<composite:implementation>` sind. Die Übersicht in Tabelle 5.5 kann lediglich als Kurzfassung und Skizzierung der Grundgedanken dienen. Eine ausführliche Darstellung erfolgt im Abschnitt 6.5.

Tabelle 5.5: Tags der Composite-Component-Bibliothek

Funktion	JSF-Tag	Beschreibung
Schnittstelle	<composite:interface>	Beschreibung der Schnittstelle.
	<composite:attribute>	Deklaration eines Schnittstellen-Attributs.
	<composite:facet>	Deklaration einer Facette.
	<composite:valueHolder>	Deklariert Namen für Attribut, das das Interface <code>ValueHolder</code> implementiert.
	<composite:editableValueHolder>	Deklariert Namen für Attribut, das das Interface <code>EditableValueHolder</code> implementiert.

Tabelle 5.5: Tags der Composite-Component-Bibliothek (Fortsetzung)

Funktion	JSF-Tag	Beschreibung
	<composite: actionSource>	Deklariert Namen für Attribut, das das Interface ActionSource2 implementiert.
	<composite: extension>	Erweiterung für JSR 276 Metadaten [URL-JSR276]
Implementie- rung	<composite: implementation>	Implementierung der Komponente.
	<composite: insertFacet>	Fügt benannte Facette einer anderen Komponente hinzu.
	<composite: renderFacet>	Fügt Facette der Komponente hinzu.
	<composite: insertChildren>	Fügt Inhalt dem Komponentenbaum hinzu.

5.8 Die JSTL-Kern- und Funktionsbibliothek

Wir haben in Abschnitt 4.11 die JSTL-Kern- und die JSTL-Funktionsbibliothek exemplarisch eingeführt. Beide Bibliotheken sind in der Facelets-VDL-Version von JSF 2.0 enthalten, in der JSP-Version jedoch nicht. Dieser Abschnitt dient der vollständigen Nennung der enthaltenen Tags und Funktionen. Da die Verwendung der Bibliotheken in der Regel nicht notwendig ist und wir von der Verwendung abraten, enthält Anhang B keine Beschreibung der Tags und Funktionen. Wir verweisen den Leser auf das API-Doc der JSF-Implementierung.

Als Namensräume werden typischerweise `c` und `fn` entsprechend der folgenden Namensraum-Deklaration verwendet:

```
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://java.sun.com/jsf/html"  
      xmlns:c="http://java.sun.com/jsp/jstl/core"  
      xmlns:fn="http://java.sun.com/jsp/jstl/functions">
```

Tabelle 5.6: Tags der JSTL-Kernbibliothek

Funktion	JSTL-Tag	Beschreibung
Fallunter- scheidung	<c:if>	Einfacher Test mit Rumpfauswertung bei positivem Ergebnis. Kein Else-Zweig.

Tabelle 5.6: Tags der JSTL-Kernbibliothek (Fortsetzung)

Funktion	JSTL-Tag	Beschreibung
	<c:choose>	Mehrfachunterscheidung mit enthaltenen <c:when> und <c:otherwise>.
	<c:when>	Im <c:choose> enthaltener Test für einen Fall.
	<c:otherwise>	Alternative, falls kein vorheriger Test erfolgreich war.
Exception-Handling	<c:catch>	Abfangen eines Fehlers.
Iteration	<c:forEach>	Iteration ähnlich <h:dataTable> und <ui:repeat>.
Zuweisung	<c:set>	Setzt Wert einer Variablen.

Tabelle 5.7: Funktionen der JSTL-Funktionsbibliothek

Funktion	JSTL-Funktion	Beschreibung
Test	fn:contains()	Test auf enthaltenen Teil-String.
	fn:containsIgnoreCase()	Test auf enthaltenen Teil-String, ohne Groß/Kleinschreibung zu beachten.
	fn:endsWith()	Test eines Strings auf Suffix.
	fn:startsWith()	Test eines Strings auf Präfix.
String-Verarbeitung	fn:escapeXml()	Erzeugt Escape-Sequenzen für XML-Markup.
	fn:join()	Umwandlung aller Array-Elemente in einen String.
	fn:replace()	Ersetzung von Teil-Strings.
	fn:split()	Aufteilen eines Strings in Teil-Strings.
	fn:substring()	Teil-String bzgl. Start- und End-Index.
	fn:substringAfter()	Teil-String nach Muster.
	fn:substringBefore()	Teil-String vor Muster.
	fn:toLowerCase()	String-Umwandlung in Kleinbuchstaben.
	fn:toUpperCase()	String-Umwandlung in Großbuchstaben.
	fn:trim()	Entfernen von Leerzeichen am String-Anfang und -Ende.

Tabelle 5.7: Funktionen der JSTL-Funktionsbibliothek (Fortsetzung)

Funktion	JSTL-Funktion	Beschreibung
Funktionen	<code>fn:indexOf()</code>	Index eines Teil-Strings im Such-String.
	<code>fn:length()</code>	Berechnung der String- oder Collection-Länge.

Kapitel 6

Facelets

In der ersten Auflage des Buches befand sich im Kapitel *Ausblick* ein Abschnitt über Facelets und die Empfehlung, Facelets statt JSPs zu verwenden. Facelets wurde als Open-Source-Implementierung von Jacob Hookom, einem Sun-Mitarbeiter und Mitglied der JSF-Expertengruppe, initiiert [URL-FL]. Mit JSF 2.0 ist Facelets die Standardseitenbeschreibungssprache. JSPs werden nur noch aus Kompatibilitätsgründen weitergeführt. Praktisch alle Neuerungen von JSF 2.0 sind nur mit Facelets und nicht mit JSPs möglich. Beispiele hierfür sind etwa Get-Requests (`<h:button>`, `<h:link>`), Ajax (`<f:ajax>`) und das Resource-Handling (`<h:head>`, `<h:body>`, `<h:outputScript>`, `<h:outputStylesheet>`). Diese Tags sind zwar in der HTML-Standardbibliothek enthalten, aber nur in der Facelets- und nicht in der JSP-Version. Die Möglichkeit, eigene Komponenten zu erstellen, wurde in JSF 2.0 stark vereinfacht. Die zu verwendende Tag-Bibliothek mit dem Präfix `composite` ist ebenfalls nur für Facelets verfügbar. Diese und viele weitere Gründe sprechen für den Einsatz von Facelets.

6.1 Templating mit Facelets

Neben den oben genannten neuen Tags der HTML-Standardbibliothek, für die Facelets benötigt wird, führt Facelets eigene Tags ein, um verschiedene Anforderungen erfüllen zu können. Zu den ganz zentralen Herausforderungen moderner Web-Anwendungen gehört ein konsistentes Look-and-Feel über die gesamte Anwendung hinweg. Aus technischer Sicht resultiert dies in einem Template-Mechanismus, um redundanzfreie, aber trotzdem einheitliche An-

wendungsseiten gestalten zu können. In diesem Abschnitt führen wir die für das Templating benötigten Facelet-Tags ein.

Zur Realisierung des Template-Mechanismus benötigt man ein *Template* und ein *Template-Client*. Das Template definiert logische Bereiche in einer Seite, die einen Default-Inhalt besitzen, der jedoch austauschbar ist. Der Template-Client injiziert einen beliebigen Inhalt in das Template. Zur Identifizierung der Bereiche und Inhalte werden Namen verwendet. Abbildung 6.1 auf der nächsten Seite verdeutlicht dies schematisch. Die Datei `template.xhtml` stellt das Template, die Datei `template-client.xhtml` den Template-Client dar. Im Template wird mit `<ui:insert>` ein Seitenbereich als austauschbar definiert. Er erhält im Beispiel den Bezeichner `area`. Der Template-Client definiert mit dem `<ui:composition>`-Tag eine Komposition, die alle eingeschlossenen HTML- und JSF-Elemente enthält. Alle umgebenden Komponenten oder, wie im Beispiel, Texte werden ignoriert. Im Beispiel hätten also auch das `<html>`- und das `<body>`-Tag weggelassen werden können. Die Verwendung von syntaktisch korrektem HTML ermöglicht jedoch die Nutzung von HTML-Editoren und kann daher sinnvoll sein.

Über das `template`-Attribut des `<ui:composition>`-Tags wird das zu verwendende Template definiert. Beim Rendern des Template-Clients, der aus JSF-Sicht die tatsächliche, View-Id-definierende Seite darstellt, wird für jedes `<ui:define>`-Tag im Template-Client nach einem `<ui:insert>`-Tag im Template mit demselben Namen gesucht. Falls ein solches Tag existiert, wird es mit dem Inhalt des `<ui:define>` ersetzt.

Um die Wiederverwendung von Seitenelementen zu erhöhen, ist es häufig sinnvoll, den `<ui:define>`-Inhalt mit Hilfe des `<ui:include>`-Tags in eine eigene Datei auszulagern. Im Beispiel der Abbildung 6.1 also etwa

```
<ui:define name="area">
    <ui:include src="area.xhtml" />
</ui:define>
```

Der in `area.xhtml` definierte JSF-Code kann dann auch an anderen Stellen eingebunden werden.

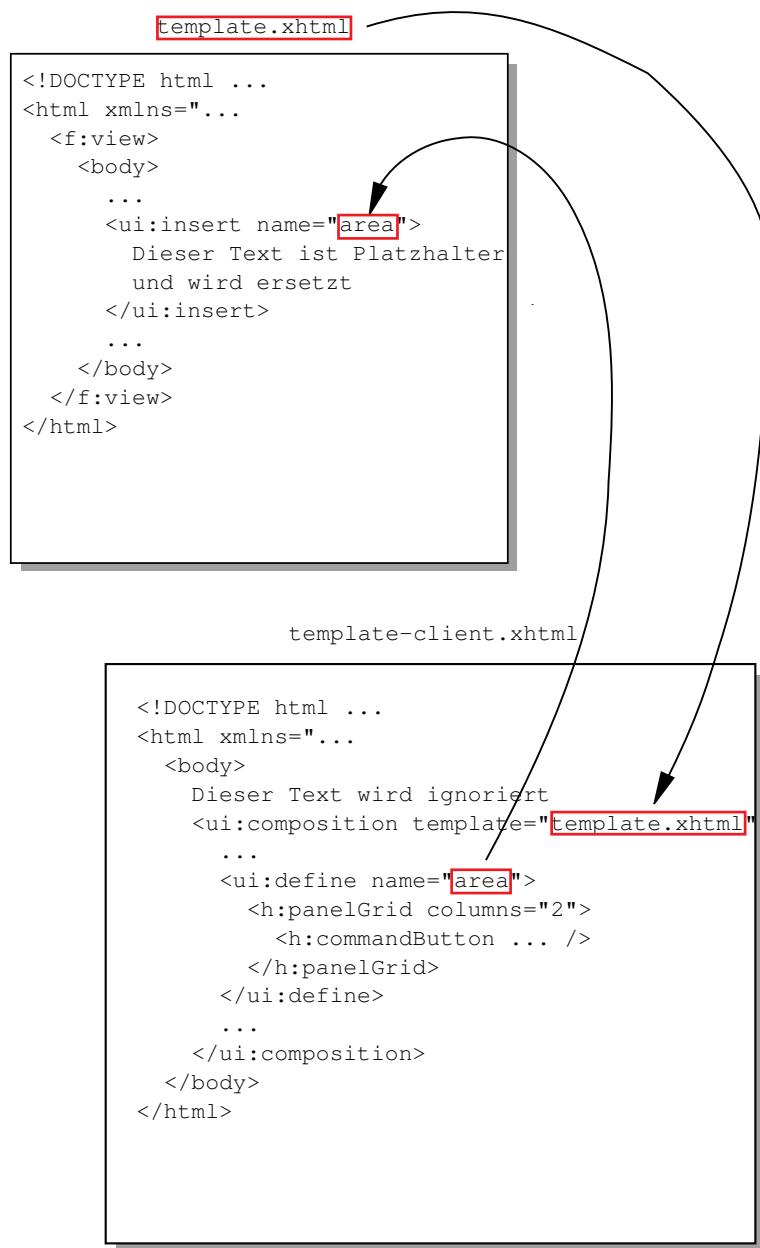


Abbildung 6.1: Darstellung des Template-Mechanismus

6.2 Ein Template-Beispiel: UPN-Rechner

Um den Template-Mechanismus zu veranschaulichen, erstellen wir als Beispieldatenwendung einen Taschenrechner. Eine früher durchaus weiter verbreitete Rechnerart als heute waren die UPN-Rechner. UPN steht für *Umgekehrt Polnische Notation* und ist eine Postfix-Notation für arithmetische Ausdrücke. Statt $a + b$ schreibt man $a\ b\ +$. Die Realisierung erfolgt über einen Stack, der die Operanden sukzessive aufnimmt. Beim Auftreten eines Operators werden die beiden obersten Operanden des Stacks zur Durchführung der Operation verwendet. Auf einem realen Rechner werden statt der Tastenfolge

die folgenden Tasten betätigt:

Da Operanden direkt nacheinander stehen dürfen, müssen sie mit einem speziellen Befehl, der Enter-Taste, auf den Stack gelegt werden. Für kompliziertere Ausdrücke verwendet man in der üblichen Infix-Notation Klammern. Die UPN kommt ohne Klammern aus. Statt

schreibt man

Ziel des Beispiels ist jedoch nicht die Einführung in die Umgekehrt Polnische Notation, sondern der Einsatz des Template-Mechanismus. Hierzu definieren wir für den Rechner ein Layout, das aus getrennten Bereichen für die Eingabe und Fehlermeldungen, für die Darstellung des Stacks, für spezielle UPN-relevante Befehle sowie einem Zifferblock mit Operationszeichen besteht. Abbildung 6.2 auf der nächsten Seite zeigt die Darstellung im Browser. Man erkennt oben ein Eingabefeld und unten ein Ausgabefeld für Fehlermeldungen. Im linken Teil ist der Platz für die Darstellung der Stack-Inhalte, im rechten ein Befehlsblock zu erkennen. Zentral in der Mitte befinden sich der Ziffernblock und die Operationszeichen.

Offensichtlich besteht das Layout aus fünf unabhängigen Teilen. Das verantwortliche Template ist in Listing 6.1 dargestellt.

Listing 6.1: Template-Datei template.xhtml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

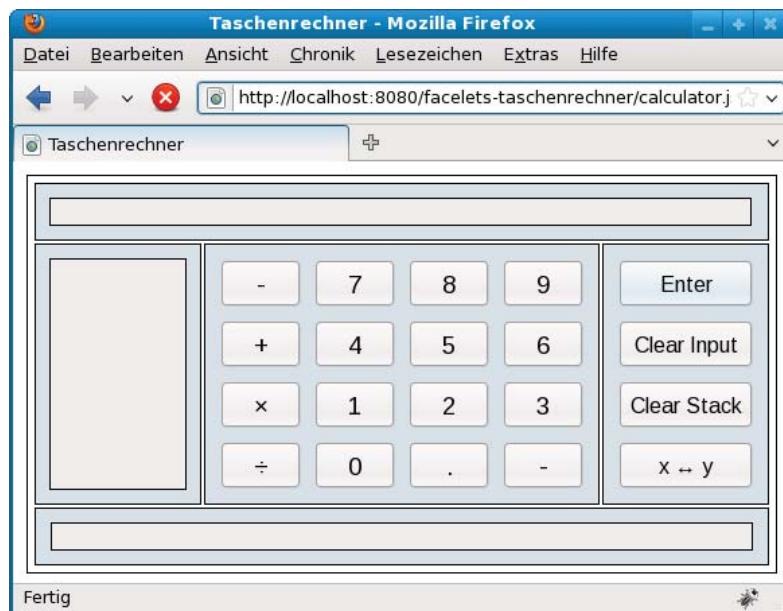


Abbildung 6.2: Layout für den UPN-Rechner

```
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4 <html xmlns="http://www.w3.org/1999/xhtml"
5   xmlns:ui="http://java.sun.com/jsf/facelets"
6   xmlns:h="http://java.sun.com/jsf/html">
7 <head>
8   <title>
9     <ui:insert name="title">
10    Default-Title
11   </ui:insert>
12 </title>
13   <link rel="stylesheet" type="text/css" href="calculator.css" />
14 </head>
15 <body>
16   <h:form>
17     <div id="calc">
18       <div id="input">
19         <ui:include src="input.xhtml" />
20       </div>
21       <div id="stack">
22         <ui:include src="stack.xhtml" />
23       </div>
24       <div id="keypad">
25         <ui:insert name="keypad">
26           Keypad-Platzhalter
27         </ui:insert>
```

```
28      </div>
29      <div id="control">
30          <ui:include src="control.xhtml" />
31      </div>
32      <div id="error">
33          <ui:insert name="error">
34              Fehlermeldungen
35          </ui:insert>
36      </div>
37  </div>
38  </h:form>
39 </body>
40 </html>
```

Man erkennt darin insgesamt sechs Teilbereiche, da auch der Titel der Seite variabel sein soll. Das Template verwendet sowohl `<ui:insert>`- als auch `<ui:include>`-Tags. Das `<ui:include>`-Tag (Zeilen 19, 22 und 30) ist das zentrale Tag zur Wiederverwendung von JSF-Seiten und nimmt den Inhalt der über das `src`-Attribut referenzierten Datei als Teil der aktuellen JSF-Seite auf. Die Verwendungen im Beispiel haben dabei Dateinamen als String-Literale benutzt. Es sind jedoch auch EL-Ausdrücke als Wert des `src`-Attributs erlaubt, wofür wir später noch ein Beispiel entwickeln werden. Wir verzichten an dieser Stelle auf eine vollständige Darstellung der drei inkludierten Dateien und beschränken uns auf die Datei `input.xhtml`, deren Inhalt in Listing 6.2 dargestellt ist.

Listing 6.2: Inkludierte Datei `input.xhtml`

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5      xmlns:ui="http://java.sun.com/jsf/facelets"
6      xmlns:h="http://java.sun.com/jsf/html">
7
8  <body>
9      Alles ausserhalb eines ui:composition wird ignoriert
10     <ui:composition>
11         <h:inputText value="#{calc.input}" disabled="true" />
12     </ui:composition>
13 </body>
14 </html>
```

Man erkennt in den Zeilen 10 bis 12 die Verwendung des `<ui:composition>`-Tags. Dieses fasst die umschlossenen Komponenten zu einer UI-Komponente zusammen, die in den Komponentenbaum aufgenommen wird. Im Beispiel wird lediglich ein `<h:inputText>` umschlossen. Dies hätte auch ohne das `<ui:composition>` als einziges Element in der Datei stehen können, ohne dass sich der entstehende Komponentenbaum geändert hätte. Im Beispiel wurde jedoch nicht so verfahren, um eine vollständige XHTML-Datei zu erhalten, die dann mit gängigen HTML-Werkzeugen, wie etwa Dreamweaver, problemlos verwendet werden kann. Damit demonstrieren wir eine weitere Eigenschaft des `<ui:composition>`-Tags: Die JSF-Implementierung ignoriert alle außerhalb des `<ui:composition>`-Tags liegenden einfachen Texte, aber auch HTML- oder JSF-Tags.

Kommen wir aber zum eigentlichen Template-Mechanismus zurück, und schauen uns den Template-Client genauer an. Der Quell-Code für den Template-Client `calculator.xhtml` ist im Listing 6.3, die Darstellung im Browser in der bereits bekannten Abbildung 6.2 auf Seite 219 dargestellt. Wir verzichten hier und in weiteren Listings dieses Kapitels auf die Darstellung der Zeichen-Codierung und des Dokumententyps.

Listing 6.3: Template-Client `calculator.xhtml`

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html">
4 <body>
5   Dieser Text wird ignoriert.
6   <ui:composition template="template.xhtml">
7     <ui:define name="title">
8       Taschenrechner
9     </ui:define>
10    <ui:define name="keypad">
11      <ui:include src="keypad.xhtml" />
12    </ui:define>
13    <ui:define name="error">
14      <ui:include src="error.xhtml" />
15    </ui:define>
16  </ui:composition>
17  Dieser Text wird auch ignoriert.
18 </body>
19 </html>
```

Man erkennt auch hier die Verwendung des `<ui:composition>`-Tags, allerdings mit dem Attribut `template`. Mit ihm wird das Template aus Listing 6.1

eingebunden, das drei `<ui:insert>`-Tags verwendet, die jeweils mit ihrem `name`-Attribut die Namen `title`, `keypad` und `error` verwenden. Die drei `<ui:define>`-Tags des Template-Clients definieren gerade diese drei einzubindenden Bereiche. Von besonderem Interesse ist hier die Datei `keypad.xhtml`, die die Ziffern-Tastatur in Anlehnung an eine Computer-Tastatur enthält. Da diese lediglich aus einfachen `<h:commandButton>`-Tags bestehen, verzichten wir auf eine Darstellung.

Interessanter an dieser Stelle ist der zweite Template-Client, mit dem wir die Ziffern-Tastatur in Handy-Manier auslegen wollen. Abbildung 6.3 zeigt die Darstellung im Browser. Man erkennt im Vergleich zu Abbildung 6.2 einen anderen Fenstertitel sowie das geänderte Tastenfeld. Der dafür verantwortliche JSF-Code ist in Listing 6.4 dargestellt.

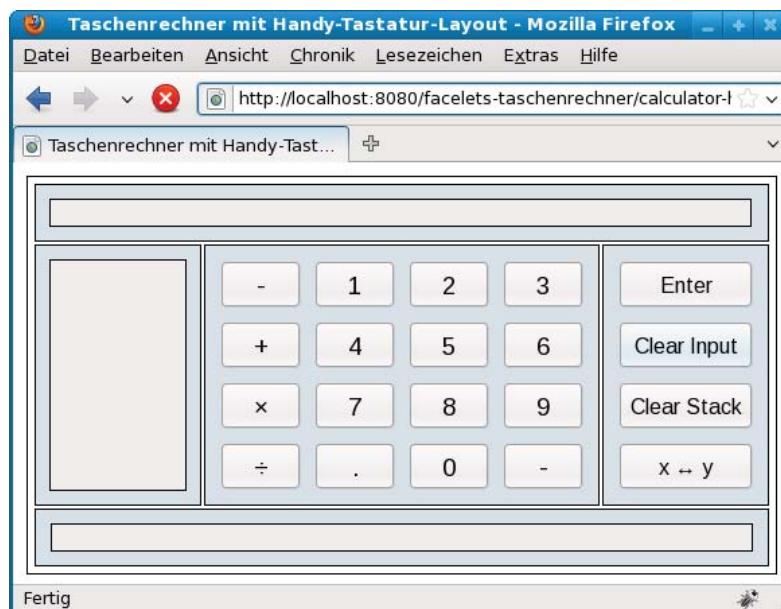


Abbildung 6.3: UPN-Rechner mit Handy-Tastenfeld

Listing 6.4: Zweiter Template-Client `calculator-handy.xhtml`

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html">
4 <body>
5   <ui:composition template="template.xhtml">
6     <ui:define name="title">
```

```
7      Taschenrechner mit Handy-Tastatur-Layout
8  </ui:define>
9  <ui:define name="keypad">
10    <ui:include src="keypad-handy.xhtml" />
11  </ui:define>
12  <ui:define name="error">
13    <ui:include src="error.xhtml" />
14  </ui:define>
15 </ui:composition>
16 </body>
17 </html>
```

Listing 6.4 zeigt die Verwendung eines anderen Fenstertitels in Zeile 7 sowie einer anderen Ziffern-Tastatur in Zeile 10 über die Datei `keypad-handy.xhtml`.

6.3 Dynamische Templates

Der Wert des `src`-Attributs darf ein beliebiger EL-Ausdruck sein. Statt der bisher verwendeten String-Literale kann man alternativ auf das Property einer Managed Bean zurückgreifen und somit sehr dynamische Templates realisieren. Wir überarbeiten dafür unseren Template-Client für das Tastenfeld und die Fehlermeldungen:

```
<ui:define name="keypad">
  <ui:include src="#{calc.zehnerOrHandy}" />
</ui:define>
<ui:define name="error">
  <ui:include src="error-switch.xhtml" />
</ui:define>
```

Der Grund für die Überarbeitung der Fehlermeldungen ist eine zusätzliche Schaltfläche für das Umschalten des Tastenfelds. Um das Gesamt-Layout nicht verändern zu müssen, haben wir diese Schaltfläche in das Fehler-Panel aufgenommen. Abbildung 6.4 auf der nächsten Seite zeigt die neue Version.

Dabei ist die Schaltfläche an die Action-Methode `changeNumericHandy()` gebunden:

```
<h:commandButton value="Tastatur wechseln"
                  action="#{calc.changeNumericHandy}" />
```

Die Action-Methode wechselt zwischen den beiden Dateinamen hin und her.

```
private static final String KEYPAD = "keypad.xhtml";
private static final String KEYPAD_HANDY = "keypad-handy.xhtml";
```

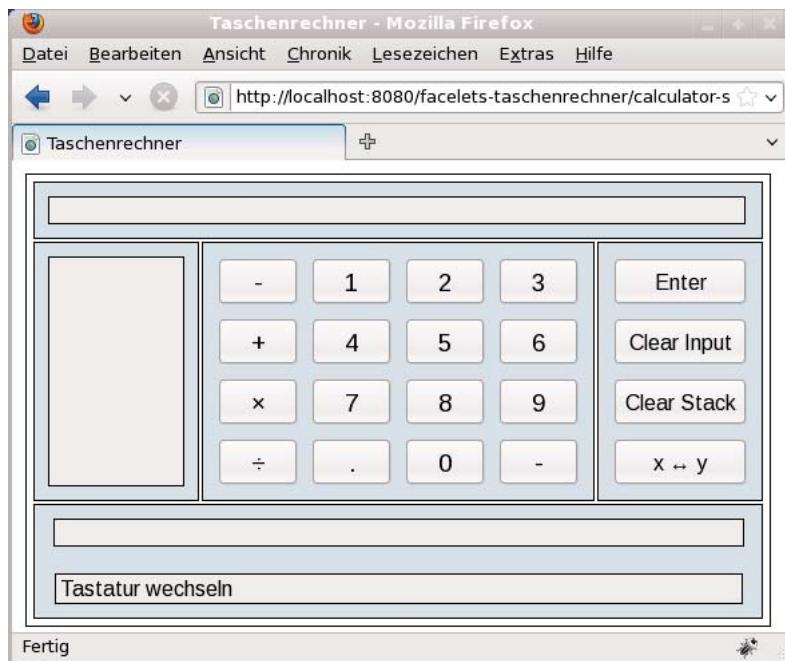


Abbildung 6.4: UPN-Rechner mit auswechselbarem Tastenfeld

```
...
/***
 * Wechselt zwischen Ziffernblock und Handy-Layout
 */
public void changeNumericHandy() {
    if (zehnerOrHandy.equals(KEYPAD)) {
        zehnerOrHandy = KEYPAD_HANDY;
    } else {
        zehnerOrHandy = KEYPAD;
    }
}
```

Durch die Verwendung eines EL-Ausdrucks im `src`-Attribut des `<ui:include>`-Tags können also hoch dynamische Seiten auf Basis des Template-Mechanismus erstellt werden.



Projekt

Der beschriebene Code für die drei alternativen UPN-Rechner ist im Projekt `facelets-taschenrechner` enthalten.

6.4 Weitere Facelets-Tags

Neben den bisher für das Templating verwendeten Tags `<ui:composition>`, `<ui:define>`, `<ui:insert>` und `<ui:include>` definiert die Facelets-Bibliothek sieben weitere Tags, die wir nun vorstellen. Es sind dies die Tags `<ui:component>`, `<ui:debug>`, `<ui:decorate>`, `<ui:fragment>`, `<ui:param>`, `<ui:repeat>` und `<ui:remove>`.

Das `<ui:component>`-Tag erzeugt wie das `<ui:composition>`-Tag eine Komponente. Wie beim `<ui:composition>`-Tag wird das Markup außerhalb der Komponente ignoriert. Im Unterschied zum `<ui:composition>`-Tag besitzt das `<ui:component>`-Tag jedoch kein `template`-Attribut.

Das `<ui:fragment>`-Tag verhält sich analog zum `<ui:component>`, außer dass umgebendes Markup nicht ignoriert wird.

Das `<ui:decorate>`-Tag erzeugt wie das `<ui:composition>`-Tag eine Komponente und besitzt das `template`-Attribut. Es ignoriert jedoch nicht umgebendes Markup.

Das `<ui:param>`-Tag erlaubt die Weitergabe eines Parameters an eine inkludierte Datei (`<ui:include>`) oder ein Template (`<ui:composition>` oder `<ui:decorate>`) und ist nur innerhalb dieser Tags erlaubt. Im folgenden Beispiel wird eine Datei inkludiert und dieser inkludierten Datei ein Parameter mit Namen `param` übergeben.

```
<ui:include src="/included.xhtml">
    <ui:param name="param" value="Der Wert des Parameters" />
</ui:include>
```

In der inkludierten Datei kann dann auf den Parameter in der üblichen EL-Syntax zugegriffen werden, wie der folgende Code demonstriert. Der Parametername wird zur lokalen Variable.

```
<h:body>
    <ui:composition>
        Der inkludierte Text. <br/>
        Der Wert des Parameters: "#{param}"
    </ui:composition>
</h:body>
```

Das `<ui:repeat>`-Tag wird verwendet, um über eine Collection zu iterieren. Als mögliche Collection-Typen sind Arrays und `java.sql.ResultSet`-Objekte sowie List-Implementierungen erlaubt, ganz ähnlich wie beim Standard-Tag `<h:dataTable>`. Genau wie bei diesem wird auch beim `<ui:repeat>` über das `value`-Attribut ein collection-wertiges Property gebunden und mit dem `var`-Attribut eine Iterationsvariable definiert. Während jedoch das `<h:da-`

`taTable`-Tag eine Tabelle mit den entsprechenden HTML-Tags generiert, ist das `<ui:repeat>`-Tag lediglich für die Iteration zuständig, und der Entwickler muss für das entsprechende Markup sorgen. Das Tag ist somit universeller einsetzbar. Wir wollen als Beispiel eine ungeordnete Liste (``) erzeugen. Für den Getter

```
public String[] getDigits() {  
    return new String[] {"eins", "zwei", "drei", "vier", "fünf",  
                        "sechs", "sieben", "acht", "neun", "null"};  
}
```

kann mit Hilfe des `<ui:repeat>` eine solche ungeordnete Liste erzeugt werden:

```
<ul>  
    <ui:repeat var="data" value="#{tagHandler.digits}">  
        <li><h:outputText value="#{data}" /></li>  
    </ui:repeat>  
</ul>
```

Mit dem optionalen Attribut `offset` kann der Startindex der Iteration, mit dem ebenfalls optionalen Attribut `step` die Schrittweite der Iteration angegeben werden, so dass `<ui:repeat>` sehr flexibel verwendbar ist.

Aufgabe 6.1

Erweitern Sie das obige Beispiel mit der Methode `getDigits()` um eine Methode `getColors()`, die ein Array von Farben zurückliefert. Im `<ui:repeat>` testen Sie im EL-Ausdruck auf einen Parameter, den Sie in der inkludierenden Datei mit Hilfe eines `<ui:param>` definiert haben. Über den Parameter kann damit gesteuert werden, ob eine Liste von Ziffern oder von Farben angezeigt wird.

Das `<ui:remove>`-Tag lässt sich für das „Auskommentieren“ von Seitenteilen verwenden. Das durch dieses Tag eingeschlossene Markup wird nicht in die View aufgenommen. Während der Kontextparameter `FACELETS_SKIP_COMMENTS`, den wir in Abschnitt 4.8.1 vorgestellt haben, das Rendern eines XML-Kommentars verhindert, aber den Kommentar in die generierte XHTML-Seite aufnimmt, arbeitet das `<ui:remove>`-Tag auf Komponentenebene und ist somit ein vollwertiger Mechanismus, um auszukommentieren.

Das letzte noch nicht vorgestellte Facelets-Tag verhält sich etwas anders. Das Tag `<ui:debug>` sammelt als Komponente Debug-Informationen für die enthaltende JSF-Seite. Über eine Tastenkombination, im Standardfall Strg-Shift-D kann ein neues Fenster geöffnet werden, das diese Debug-Informationen enthält. Zu den Informationen gehören der Zustand des Komponentenbaums sowie die in der Seite verwendeten Managed Beans. Abbildung 6.5 zeigt das Debug-Fenster im initialen Zustand.



Abbildung 6.5: Facelets-Debug-Fenster (initial)

Abbildung 6.6 zeigt das Debug-Fenster mit der Darstellung des Komponentenbaums.

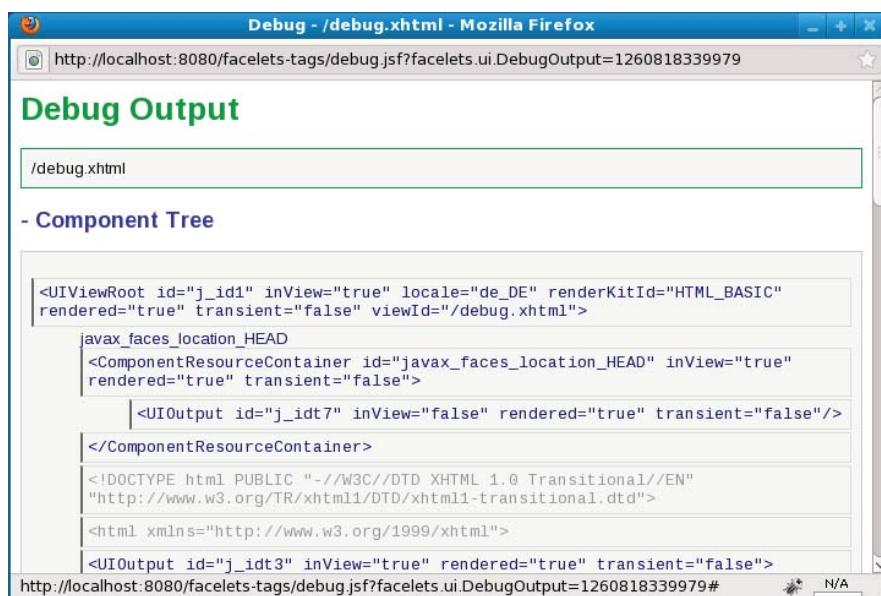


Abbildung 6.6: Facelets-Debug-Fenster mit geöffnetem Komponentenbaum

Das `<ui:debug>`-Tag besitzt die Attribute `hotkey` und `rendered`. Mit dem erstgenannten ist eine alternative Tastenkombination für das Öffnen des Debug-Fensters definierbar, also z.B. `hotkey="A"` für Strg-Shift-A. Das boolesche Attribut `rendered` bestimmt, ob die Komponente gerendert wird und der Benutzer somit die Möglichkeit hat, das Debug-Fenster zu öffnen.

Eine sehr sinnvolle Verwendung des `<ui:debug>`-Tags ist die Positionierung innerhalb des Haupt-Templates der Anwendung und Bindung eines Bean-Property über einen einfachen EL-Ausdruck an das `rendered`-Attribut. Dadurch kann die Möglichkeit für das Öffnen des Debug-Fensters innerhalb aller Seiten der Anwendung an einer einzigen zentralen Stelle ein- und ausgeschaltet werden.



Projekt

Im Projekt *facelets-tags* sind für alle dargestellten Facelet-Tags kleine Beispiele vorhanden.

6.5 Entwicklung eigener Komponenten

Die Entwicklung eigener Komponenten war bis zu JSF 2.0 ein kompliziertes und umfangreiches Unterfangen. So war es z. B. nötig, neben der eigentlichen Komponentenklasse eine Tag-Handler- und eine Render-Klasse zu schreiben. Außerdem wurde eine Tag-Definition auf XML-Basis erstellt, und die Komponenten inklusive Renderer mussten in der JSF-Konfiguration bekannt gemacht werden. Falls noch Ajax-Funktionalität benötigt wurde, stieg der Entwicklungsaufwand weiter an.

Diese Art der Entwicklung eigener Komponenten existiert in JSF 2.0 nach wie vor und wurde etwas vereinfacht. Wir gehen aber darauf nicht ein, da wir als potenzielle Leser Entwickler von Anwendungen und nicht Entwickler von Komponenten ansprechen wollen. Falls Sie derartige Komponenten trotzdem entwickeln möchten, nutzen Sie z. B. das Buch von Ed Burns [Bur10]. Burns leitete die Expertengruppe der JSF-Spezifikation, und sein Buch enthält ein entsprechendes Kapitel zum Thema.

Wir wollen an dieser Stelle die mit JSF 2.0 eingeführte Alternative sogenannter *zusammengesetzter Komponenten* (Composite Components) darstellen. Diese bilden eine einfache, auf Facelets basierende Möglichkeit zur Definition neuer Komponenten, ohne Java-Code oder XML-Konfigurationen schreiben zu müssen. Wir beginnen mit einem einfachen Beispiel, um die zugrunde liegende Idee herauszuarbeiten. Anschließend entwickeln wir das Beispiel weiter und verallgemeinern seinen Anwendungsbereich ein wenig.

Zusammengesetzte Komponenten machen von einer grundlegenden Regel der Software-Technik Gebrauch und unterscheiden explizit zwischen der Schnittstelle und der Implementierung, für die spezielle Tags existieren. Li-

sting 6.5 zeigt eine zusammengesetzte Komponente mit den Tags `<composite:interface>` und `<composite:implementation>`.

Listing 6.5: Zusammengesetzte Komponente login-comp-ft.xhtml

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:h="http://java.sun.com/jsf/html"
3   xmlns:composite="http://java.sun.com/jsf/composite">
4
5   <composite:interface>
6     <composite:attribute name="handler" />
7   </composite:interface>
8
9   <composite:implementation>
10    <h:form>
11      <h:panelGrid columns="2">
12        Benutzer
13        <h:inputText value="#{cc.attrs.handler.name}"
14          required="true" />
15        Passwort
16        <h:inputSecret value="#{cc.attrs.handler.password}"
17          required="true" />
18      </h:panelGrid>
19      <h:commandButton value="Einloggen"
20        action="#{cc.attrs.handler.login}" />
21    </h:form>
22  </composite:implementation>
23 </html>
```

In Zeile 3 von Listing 6.5 erkennt man den eigens für zusammengesetzte Komponenten eingeführten Namensraum `http://java.sun.com/jsf/composite`. Als Präfix sollte `composite`, oder, etwas kürzer, `cc` verwendet werden. Da sich die Definition der zusammengesetzten Komponente in der Datei `login-comp-ft.xhtml` befindet, wird die Komponente mit Namen `login-comp-ft` (Login-Komponente, first try) definiert. Die Zeilen 5 bis 7 legen die Schnittstelle fest. Im Beispiel ist dies lediglich ein Attribut namens `handler`.

In den Zeilen 9 bis 22 erfolgt die Implementierung der Komponente. Innerhalb des `<composite:implementation>`-Tags sind JSFs HTML-, Kern- und Facelets-Tags sowie einfaches HTML und Text erlaubt. Der Beispiel-Code ist bis auf die EL-Ausdrücke selbsterklärend. Der Werteausdruck `#{cc.attrs.handler}` besteht aus der vordefinierten Variablen (dem Schlüsselwort) `cc`, die die aktuelle zusammengesetzte Komponente (Composite Component) darstellt. Weiterhin sind die Properties der Komponente in der Map `attrs` verfügbar.

bar. Da wir ein Interface-Attribut `handler` definiert haben, ist das aktuell für diesen Attributnamen übergebene Objekt über diesen Schlüssel in der Map referenzierbar. Wir werden später sehen, dass das Objekt eine Managed Bean mit den Properties `name` und `password` sowie der Action-Methode `login()` ist. Die Facelets-Implementierung behandelt `<composite:interface>` und `<composite:implementation>` ähnlich wie ein `<ui:composition>`: Jeglicher Text außerhalb der Tags wird ignoriert.

Die Klasse `LoginHandlerFT` wird später bei der aktuellen Benutzung der zusammengesetzten Komponente als Wert von `handler` übergeben. Ihre Implementierung ist trivial:

```
@ManagedBean
public class LoginHandlerFT {

    private String name;
    private String password;

    public String login() {
        if (name.equals(password)) {
            return "login-ok.xhtml";
        } else {
            return "login-not-ok.xhtml";
        }
    }

    // Getter und Setter
}
```

Zur Vervollständigung des Beispiels fehlt noch die Verwendung der zusammengesetzten Komponente. Listing 6.6 zeigt die Datei `login-ft.xhtml`, die die Verwendung realisiert.

Listing 6.6: Verwendung der Komponente (Datei `login-ft.xhtml`)

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2      xmlns:jp="http://java.sun.com/jsf/composite/jsfpraxis">
3
4     <head>
5         <title>Login mit Composite Components (first try)</title>
6     </head>
7     <body>
8         <jp:login-comp-ft handler="#{loginHandlerFT}" />
9     </body>
10    </html>
```

In Zeile 2 wird ein neuer Namensraum verwendet: der schon bekannte Namensraum für zusammengesetzte Komponenten mit angehängtem /jsfpraxis. Wie ist dies zu verstehen? Zusammengesetzte Komponenten werden wie andere Ressourcen (siehe Abschnitt 4.10 auf Seite 187 zur Verwendung allgemeiner Ressourcen) unter dem Verzeichnis **resources** direkt unterhalb des Anwendungsverzeichnisses abgelegt. Um Bibliotheken zusammengesetzter Komponenten erstellen zu können, wird ein Unterverzeichnis mit beliebigem Namen angelegt, das die Komponentendefinitionen enthält. Im Beispiel befindet sich die Komponentendefinition also in der Datei **/resources/jsfpraxis/login-comp-ft.xhtml**. Das Ende des Namensraums in Zeile 2, Listings 6.6, spiegelt den Verzeichnisnamen der Bibliothek wider. Die eigentliche Verwendung der zusammengesetzten Komponente erfolgt in Zeile 8, in der als Wert des Komponentenattributs **handler** die Managed Bean **loginHandlerFT** übergeben wird.

Wie wir gesehen haben, reduziert sich der Aufwand zur Entwicklung eigener Komponenten auf die Definition der Schnittstelle. Man gewinnt dadurch die Möglichkeit der Wiederverwendung des Implementierungs-Codes. Wir wollen an einem zweiten Beispiel einige weitere Möglichkeiten, die zusammengesetzte Komponenten bieten, demonstrieren. Dazu überarbeiten wir das Login-Beispiel und verwenden neue Tags bzw. Attribute.

Wir trennen die Diskussion von Schnittstelle und Implementierung und beginnen mit der Schnittstelle. Der folgende Code zeigt die Schnittstelle der Komponente **login-comp**.

```
<composite:interface>
    <composite:actionSource name="login" targets="form:login" />
    <composite:attribute name="loginAction"
        method-signature="java.lang.String action()" />
    <composite:attribute name="credentials" />
    <composite:attribute name="salutation" default="Anmeldung" />
</composite:interface>
```

Das Tag **<composite:actionSource>** zeigt an, dass die zusammengesetzte Komponente Action-Events wirft und somit in der benutzenden Seite Action-Event-Listener registriert werden können. Als Registrierungsname des Listeners ist dann **login** zu verwenden. Da die zusammengesetzte Komponente mehrere Action-Event-Quellen haben kann, ist die Quelle für diese Action-Source im **targets**-Attribut anzugeben.

Im ersten Beispiel hat die Managed Bean sowohl Properties für Benutzernamen und Passwort als auch die Login-Methode bereitgestellt. Um die Flexibilität zusammengesetzter Komponenten zu demonstrieren, trennen wir die Daten und die Methode. Im Attribut **loginAction** wird die Login-Methode übergeben. Daher muss das Attribut mit **method-signature** versehen werden. Als

Wert ist die Methodensignatur mit voll qualifiziertem Klassennamen anzugeben. Das Attribut `credentials` ist als ein Objekt zur Aufnahme von Benutzernamen und Passwort vorgesehen. Im Attribut `salutation` wird schließlich gezeigt, dass Attribute Default-Werte haben können.

Wenden wir uns nun der Komponentenimplementierung zu. Listing 6.7 zeigt die Implementierung der zusammengesetzten Komponente `login-comp`, die wir nun diskutieren.

Listing 6.7: Zusammengesetzte Komponente (`login-comp`)

```
1 <composite:implementation>
2   <h:head>
3     <title>#{cc.attrs.salutation}</title>
4   </h:head>
5   <h:body>
6     <h:outputStylesheet library="jsfpraxis"
7       name="login-style.css" />
8     <h:form id="form">
9       <div style="font-weight: bold; text-align: center">
10         #{cc.attrs.salutation}
11       </div>
12       <br />
13       <h:panelGrid columns="2">
14         Benutzer
15         <h:inputText id="user" required="true"
16           value="#{cc.attrs.credentials.name}" />
17         Passwort
18         <h:inputSecret id="password" required="true"
19           value="#{cc.attrs.credentials.password}" />
20       </h:panelGrid>
21       <br />
22       <div style="text-align: center">
23         <h:commandLink id="login"
24           action="#{cc.attrs.loginAction}">
25           <h:graphicImage library="jsfpraxis"
26             name="button.png" />
27         </h:commandLink>
28       </div>
29     </h:form>
30   </h:body>
31 </composite:implementation>
```

In den Zeilen 3 und 10 wird das Attribut `salutation`, in den Zeilen 16 und 19 das Attribut `credentials` verwendet. Die Action-Methode `loginAction`, in der Schnittstelle ebenfalls ein Attribut, jedoch mit zusätzlicher Methoden-

signatur, wird schließlich in Zeile 24 verwendet. Nachdem alle Attribute der Schnittstelle verwendet wurden, steht noch die Action-Source der Schnittstelle aus. Der Wert des **targets**-Attributs legt das Ziel des noch zu definierenden Action-Listeners fest. Im Beispiel ist dies der Command-Link in Zeile 23. Im **name**-Attribut wird der Name definiert, unter dem sich Action-Listener registrieren können, was wir im Folgenden gleich vorführen.

Sie werden sich vielleicht fragen, warum wir einen Command-Link und keinen Command-Button verwendet haben. Das Beispiel dient zur Verdeutlichung des Bibliotheksgedankens zusammengesetzter Komponenten. Diese werden in Verzeichnissen unterhalb `/resources` abgelegt, was auch der Fall für allgemeine Ressourcen ist (siehe Abschnitt 4.10). Aus diesem Grund haben wir die CSS-Datei (Zeilen 6/7) und die Graphik der Schaltfläche (Zeilen 25/26) in dasselbe Verzeichnis gelegt wie die zusammengesetzte Komponente. Command-Buttons lassen zwar auch Graphiken für die Schaltfläche zu, erlauben jedoch keine explizite Nennung über ein **library**-Attribut. Über die vordefinierte EL-Variable **resource** hätte allerdings ein Zugriff auf die Ressource realisiert werden können, wie in Abschnitt 4.10 dargestellt.

Kommen wir nun zur Verwendung unserer zusammengesetzten Komponente. Die folgende JSF-Seite verwendet sie über den XML-Namensraum `jp` (`jsf-praxis`).

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:jp="http://java.sun.com/jsf/composite/jsfpraxis">

    <jp:login-comp salutation="Anmeldung zum Online-Banking"
                   loginAction="#{loginHandler.login}"
                   credentials="#{credentials}">
        <f:actionListener for="login"
                          type="de.jsfpraxis.MyLogger" />
    </jp:login-comp>
</html>
```

Wird das **salutation**-Attribut nicht verwendet, kommt der in der Komponentenschnittstelle definierte Default-Wert zum Einsatz. Die Managed Beans `loginHandler` und `credentials` sind einfache Klassen, die sich auf das Login bzw. die Authentifizierungsdaten beschränken. Die Klasse `Credentials` besteht lediglich aus den Properties `name` und `password`, so dass wir auf eine Darstellung verzichten können. Die Klasse `LoginHandler` besteht im Wesentlichen aus der Methode `login()`:

```
public String login() {
    FacesContext fc = FacesContext.getCurrentInstance();
    ELContext elc = fc.getELContext();
```

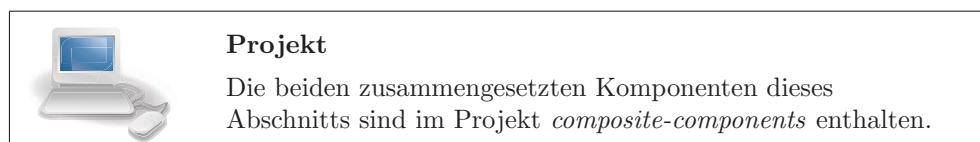
```
Credentials credentials = (Credentials) fc.getApplication()
    .getExpressionFactory().createValueExpression(elc,
        "#{credentials}", Credentials.class).getValue(elc);
if (credentials.getName().equals(credentials.getPassword())) {
    return "login-ok.xhtml";
} else {
    return "login-not-ok.xhtml";
}
}
```

Als letztes Element der Verwendung der zusammengesetzten Komponente bleibt noch der Action-Listener zu diskutieren. Dieser registriert auf die login-Action-Source die Klasse MyLogger. Mit dieser Klasse werden alle Login-Versuche protokolliert. Ein Action-Listener muss das Interface ActionListener und somit die Methode `processAction(ActionEvent e)` implementieren:

```
public void processAction(ActionEvent event)
throws AbortProcessingException {
    UIInput user = (UIInput) event.getComponent()
        .getParent().findComponent("form:user");
    UIInput password = (UIInput) event.getComponent()
        .getParent().findComponent("form:password");
    log.info("Benutzer: " + user.getValue() +
        ", Passwort: " + password.getValue());
}
```

Auch hier hätte der Zugriff auf die Daten über die `credential`-Bean erfolgen können. Wir haben jedoch einen alternativen Weg gewählt, um die Alternative über den JSF-Komponentenbaum noch einmal zu demonstrieren.

Falls zusammengesetzte Komponenten wiederverwendet werden sollen, empfiehlt sich das Paketieren in eine Jar-Datei. Dabei werden Komponenten, wie andere Ressourcen auch, im Verzeichnis `/META-INF/resources` abgelegt.



6.6 JSF-Tags als Attribute der Standard-HTML-Tags

Eine besondere Eigenschaft von Facelets ist die Verwendung der JSF-Tag-Namen als Attribute bzw., genauer, als Attributwerte des `jsfc`-Attributs der gewöhnlichen HTML-Tags. Das `<h:commandButton>`-Tag

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

wird nach dem Rendern in HTML zu

```
<input type="submit" value="Abschicken" ... />
```

Das **id**- und **name**-Attribut haben wir der Übersichtlichkeit halber im generierten HTML-Code untergeschlagen. Dieser Code lässt sich auch als JSF-Code verwenden, wobei die üblichen JSF-Attribute, im Falle einer Schaltfläche etwa das **action**-Attribut, weiter verwendet werden. Hinzu kommt das **jsfc**-Attribut (steht für JSF-Compile), das als Wert den JSF-Tag-Text erhält. Im Beispiel einer Schaltfläche also:

```
<input type="submit" jsfc="h:commandButton"  
      value="Abschicken" action="..." />
```

Dieser Code entspricht unserem ersten JSF-Code-Ausschnitt, obwohl kein JSF-Tag verwendet wurde.

Wozu sollte aber eine derartige Verwendung von JSF dienen? In den Anfängen von Facelets wurde motiviert, dass damit die Weiterverwendung herkömmlicher HTML-Editoren, etwa von Dreamweaver, möglich sei. Mittlerweile ist die IDE-Unterstützung für JSF und Facelets aber so weit gediehen, dass dies in der Regel nicht mehr nötig ist. Wir gehen auf die IDE-Unterstützung für JSF-Tags in Kapitel 10 ein.



Projekt

Das Beispiel zur Verwendung des **jsfc**-Attributs befindet sich im Projekt *facelet-tags*.

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Kapitel 7

Ajax

Ajax ist als eine Menge zusammenhängender Technologien zur client-seitigen Erhöhung der Interaktivität von Web-Anwendungen mittlerweile eine unabdingbare Voraussetzung. Eine neue Anwendungsgeneration, die man häufig unter dem Begriff *Web 2.0* subsummiert, ist ohne Ajax nicht vorstellbar. Als Akronym für *Asynchronous JavaScript and XML* stellt Ajax dabei namentlich nur einen Teil der verwendeten Technologien dar. Neben JavaScript und XML sind auch HTML/XHTML, DOM (Document Object Model) und vor allem das *XMLHttpRequest*-Objekt [URL-XMLHTTP] als zentrale Bestandteile zu nennen.

Wir gehen davon aus, dass der Leser bereits Erfahrung mit Ajax-unterstützten Web-Anwendungen hat, so dass wir von weiteren Ausführungen absehen. Im nächsten Abschnitt motivieren wir zunächst an kleinen Beispielen die Integration der neuen Ajax-Funktionalitäten in JSF 2.0. Im Abschnitt 7.2 stellen wir die Grundlagen dieser Integration dar. Abschnitt 7.3 widmet sich weiterführenden Themen im Zusammenhang mit Ajax.

Die in JSF nun verfügbaren Ajax-Funktionalitäten erlauben die Erstellung von interaktiveren Anwendungen. Der Markt an JSF-Komponentenbibliotheken hat jedoch nicht stillgestanden, und praktisch alle Bibliotheken bieten mittlerweile Komponenten an, die weit über das hinausgehen, was mit den Ajax-Grundprimitiven in JSF 2.0 möglich ist. In den beiden abschließenden Abschnitten dieses Kapitels stellen wir die JSF-Komponentenbibliothek *RichFaces* vor. Zunächst eine Basisbibliothek, deren Komponenten die bestehenden JSF-Standardkomponenten mit Ajax-Funktionalität ausstatten, dann eine umfassende Komponentenbibliothek, die z. B. Bäume und Drag and Drop zur Verfügung stellt.

7.1 Motivation

Als erstes Beispiel für unseren Ajax-Einsatz wählen wir die Aufgabe, einen eingegebenen Text zu replizieren. Und zwar soll der eingegebene Text doppelt und dreifach ausgegeben werden. Die zu verwendende JSF-Seite besitzt drei Texteingabekomponenten, wobei nur die erste zur Eingabe, die beiden anderen lediglich zur Ausgabe verwendet werden.

```
<h:form id="form">
    <h:panelGrid columns="1">
        <h:inputText id="in1" value="#{simpleHandler.text1}" />
        <h:inputText id="in2" value="#{simpleHandler.text2}" />
        <h:inputText id="in3" value="#{simpleHandler.text3}" />
        <h:commandButton id="button" value="Replizieren"
            action="#{simpleHandler.action}" />
    </h:panelGrid>
</h:form>
```

Die Properties `text1`, `text2` und `text3` sind einfache String-Variablen. Man verwendet sie in der Methode `action()`, um das Ziel der Replikation zu erreichen.

```
public void action() {
    text2 = text1 + " " + text1;
    text3 = text1 + " " + text1 + " " + text1;
}
```

Die Frage ist nun: Wie kann diese einfache Anwendung auf der Basis einer Ajax-Unterstützung so geändert werden, dass kein kompletter JSF-Bearbeitungszyklus der gesamten Seite nötig ist, die Anwendungslogik aber erhalten bleibt? Die Antwort ist erstaunlich einfach: durch die Verwendung des `<f:ajax>`-Tags innerhalb des Command-Buttons:

```
<h:commandButton id="button" value="Replizieren"
    action="#{simpleHandler.action}">
    <f:ajax />
</h:commandButton>
```

Durch diese kleine Änderung wird beim Betätigen der Schaltfläche kein HTTP-Request des Formulars, sondern ein XMLHttpRequest des entsprechenden JavaScript-Objekts ausgelöst, leider aber noch ohne den gewünschten Effekt; der Text wird nicht repliziert. Eine erneute Änderung zu

```
<f:ajax execute="@form" render="@form" />
```

hat den gewünschten Erfolg.

Zu den zentralen Fragen bezüglich einer Ajax-Anfrage gehört zum einen: Was wird in der Anfrage an den Server geschickt bzw. was wird auf dem Server

verarbeitet? Und zum anderen: Was wird in der Antwort an den Client geschickt bzw. was wird auf dem Client verarbeitet? Das Attribut `execute` legt die auf dem Server zu verarbeitenden Daten fest. Das Attribut `render` definiert, welche Teile der Seite nach Empfang der Antwort neu darzustellen sind. Das Literal `@form` als Wert für beide Attribute steht für das umgebende Formular, so dass alle Daten des Formulars auf dem Server aktualisiert werden und auch in der Antwort enthalten sind.

Wir haben nun das Replizieren der Daten realisiert, aber das genannte Teilziel einer optimierten, besser minimierten Datenübertragung und -verarbeitung noch nicht erreicht. Die Attribute `execute` und `render` erlauben alternativ die Verwendung einer Liste von Komponentenbezeichnern anstatt des von uns verwendeten Literals `@form`. Da lediglich das Property `text1` auf dem Server verarbeitet und die Properties `text2` und `text3` im Browser aktualisiert werden sollen, überarbeiten wir den Code-Ausschnitt, indem wir die entsprechenden Komponentenbezeichner angeben:

```
<h:form id="form">
    <h:panelGrid columns="1">
        <h:inputText id="in1" value="#{simpleHandler.text1}" />
        <h:inputText id="in2" value="#{simpleHandler.text2}" />
        <h:inputText id="in3" value="#{simpleHandler.text3}" />
        <h:commandButton id="button" value="Replizieren"
            action="#{simpleHandler.action}">
            <f:ajax execute="form:in1" render="form:in2 form:in3" />
        </h:commandButton>
    </h:panelGrid>
</h:form>
```

Damit werden in der noch ausführlich darzustellenden Ajax-Version des Bearbeitungsmodells einer JSF-Anfrage lediglich das Property `text1` der Managed Bean auf dem Server mit dem aktuellen Anfragewert versehen und nach Übertragung der Antwort, die ausschließlich Daten für die beiden Komponenten `id2` und `id3` enthält, auch nur diese beiden Komponenten im Browser aktualisiert.

7.2 Die Grundlagen von Ajax in JSF 2.0

Nach der zwar motivierenden, aber beispielhaften und eher oberflächlichen Einführung von Ajax-Funktionalität im letzten Abschnitt wollen wir nun die Grundlagen von Ajax in JSF 2.0 erarbeiten. Das Ziel der Bemühungen der JSF-Spezifikationsgruppe war die möglichst transparente Integration einer JSF-

spezifischen JavaScript-Bibliothek, um grundlegende Ajax-Operationen ausführen zu können.

Elementare Fragen einer solchen Integration sind etwa

- Wie wird die Bibliothek eingebunden?
- Welche Daten werden an den Server geschickt und auf dem Server verarbeitet?
- Wie wird das Bearbeitungsmodell einer JSF-Anfrage (Abschnitt 4.1) für Ajax-Anfragen überarbeitet?
- Welche Action-Methoden und Event-Listener werden aufgerufen?
- Welche Daten werden als Antwort an den Client geschickt?
- Welche Teile der Seite müssen nach erhalten der Antwort aktualisiert werden?

Wir wollen uns nun der Beantwortung dieser Fragen widmen.

7.2.1 JSFs JavaScript-Bibliothek

JSF 2.0 definiert eine JavaScript-Bibliothek als Ressource (siehe Abschnitt 4.10) mit Ressource-Namen `jsf.js` und Bibliotheksnamen `javax.faces`. Im Sinne einer tatsächlichen Standardisierung wurde der Namensraum `jsf` bei der *OpenAjax-Alliance* [URL-OAA] für JSF registriert.

Die Resource wird über das `<h:outputScript>` eingebunden und kann dann über JavaScript-Attribute der Standard-Tags (siehe Abschnitt B.2 auf Seite 386) verwendet werden. Um das Beispiel aus Abschnitt 7.1 direkt mit Hilfe der JavaScript-Bibliothek zu realisieren, verwenden wir den folgenden Code:

```
1 <h:head>
2   <title>Direkte Verwendung von JavaScript</title>
3 </h:head>
4 <h:body>
5   <h:form id="form">
6     <h:outputScript name="jsf.js" library="javax.faces"
7       target="head"/>
8     <h:panelGrid columns="1">
9       <h:inputText id="in1" value="#{simpleHandler.text1}" />
10      <h:inputText id="in2" value="#{simpleHandler.text2}" />
11      <h:inputText id="in3" value="#{simpleHandler.text3}" />
12      <h:commandButton id="button" value="Button"
13        onclick="jsf.ajax.request(this, event,
14          { execute: 'form:button form:in1',
15            render: 'form:in2 form:in3' });
16          return false;"
```

```
17      action="#{simpleHandler.action}" />
18  </h:panelGrid>
19  </h:form>
20 </h:body>
```

Man erkennt in den Zeilen 6 und 7 das Einbinden der Bibliothek und in den Zeilen 13–16 die Verwendung der JavaScript-Funktion `request()`. Diese erhält als dritten Parameter eine Menge von Schlüssel/Werte-Paaren. Als Schlüssel werden die beiden Literale `execute` und `render` verwendet, die wir schon aus dem ursprünglichen Beispiel kennen. Als Wert des Schlüssels `execute` ist zusätzlich die Komponenten-Id des Command-Buttons angegeben, da JavaScript diese nicht wie im Beispiel automatisch erkennen kann.

Das überarbeitete Beispiel ist in seiner Funktion identisch zum Beispiel in Abschnitt 7.1, wurde aber programmatisch auf der Basis von JavaScript realisiert. Wir ziehen die in Abschnitt 7.1 dargestellte Alternative auf der Basis des `<f:ajax>`-Tags vor und werden diese im Folgenden ausschließlich verwenden. Für den JavaScript-interessierten Leser stellen wir die in der Bibliothek definierten Funktionen in Tabelle 7.1 überblicksartig zusammen.

Tabelle 7.1: JavaScript-Funktionen der JSF-Bibliothek

Funktion	Beschreibung
<code>jsf.getProjectStage()</code>	Wert der Methode <code>Application.getProjectStage()</code>
<code>jsf.getViewState(form)</code>	Kodierter Zustand aller Input-Elemente des Formulars
<code>jsf.ajax.addOnError(callback)</code>	Registriert Callback für Fehlerbehandlung
<code>jsf.ajax.addOnEvent(callback)</code>	Registriert Callback für Event-Behandlung
<code>jsf.ajax.request(source, event, options)</code>	Sendet asynchrone Anfrage an Server
<code>jsf.ajax.response(request, context)</code>	Empfängt Antwort vom Server
<code>jsf.util.chain(source, event)</code>	Aufruf von Skripten (mit Varargs)

Die Verwendung des JavaScript-APIs ist insbesondere bei der Entwicklung eigener Komponenten mit Ajax-Funktionalität erforderlich, auf die wir jedoch nicht eingehen.

Zum Schluss dieses Abschnitts sei uns noch eine Bemerkung erlaubt. Die in Tabelle 7.1 dargestellten JavaScript-Funktionen sind in den API-Dokumenten

für JSF 2.0 enthalten. Damit ist JSF 2.0 die erste Java-EE-Spezifikation, die offiziell JavaScript verwendet und die Verwendung dokumentiert.

7.2.2 Das <f:ajax>-Tag

Mit dem <f:ajax>-Tag werden eine oder mehrere UI-Komponenten mit Ajax-Funktionalität versehen. Die Verwendung innerhalb oder als Wrapper um mehrere andere Komponenten bestimmt diese Funktionalitäten. Die Ajax-Funktionalität richtet sich nach der Komponentenart, kann aber in einem gewissen Bereich überschrieben werden. Um diese unterschiedlichen Einsatzgebiete exemplarisch darstellen zu können, entwickeln wir ein weiteres Beispiel, basierend auf dem Ziel der Eingabereplikation aus Abschnitt 7.1. Der folgende Code-Ausschnitt kommt ohne Command-Button aus, da wir die oben erwähnte komponentenartabhängige Ajax-Funktionalität verwenden.

```
<h:inputText id="in" value="#{simpleHandler.text1}">
    <f:ajax listener="#{simpleHandler.ajaxBehaviorListener}"
        render="out1 out2" />
</h:inputText>
<h:outputText id="out1" value="#{simpleHandler.text2}" />
<h:outputText id="out2" value="#{simpleHandler.text3}" />
```

Während JSFs Steuerungskomponenten (<h:commandButton> und <h:commandLink>) im Ajax-Standardfall auf Action-Events reagieren, reagieren Eingabekomponenten auf Value-Change-Events. Zu den Eingabekomponenten zählen <h:inputText>, <h:inputTextarea>, <h:inputSecret> und alle Komponenten, die mit <h:select...> beginnen. In obigem Beispiel wird das <f:ajax>-Tag innerhalb der Texteingabe mit der Id **in** verwendet. Wirft diese Komponente ein Value-Change-Event, wird die registrierte Listener-Methode **ajaxBehaviorListener()** aufgerufen. Diese Listener-Methode verdoppelt bzw. verdreifacht den Text im Property **text1** und schreibt das Ergebnis in die Properties **text2** bzw. **text3**. Wir haben also dasselbe Verhalten wie im Einführungsbeispiel ohne Command-Button erreicht.

Wo oder besser wie wird aber das Standardverhalten einer Komponenten definiert? Über das Verhaltensmodell einer Komponente (Component Behavior Model) können zusätzliche Verhaltensweisen einer Komponente zugewiesen werden, obwohl diese bei der Implementierung der Komponente noch nicht vorgesehen waren. Realisiert wird dies über das Interface **Behavior**, das in der Version 2.0 als einziges Sub-Interface **ClientBehavior** besitzt. Als bisher einzige Implementierung existiert die Klasse **AjaxBehavior**, über die das angesprochene Ajax-Verhalten realisiert wird. Die Klasse **UIComponentBase** enthält entsprechende Methoden, die in diesem Zusammenhang relevant

sind, z. B. `addClientBehavior()`, `getClientBehaviors()`, `getEventNames()` und `getDefaultEventName()`. Dies sind gerade die vier Methoden, die das Interface `ClientBehaviorHolder` enthält. Alle UI-Komponenten, die dieses Interface realisieren, können mit Hilfe des `<f:ajax>`-Tags mit Ajax-Funktionalitäten versehen werden. Da die Implementierung dieses Interfaces aber nur für eigenentwickelte Ajax-Komponenten interessant ist, wollen wir den Ausflug in die Ajax-Implementierung von JSF an dieser Stelle abbrechen und uns wieder dem Beispiel widmen.

Wir kehren nun zum ursprünglichen Beispiel mit dem Command-Button zurück, verwenden aber das `<f:ajax>`-Tag als Wrapper sowohl um die drei Texteingaben als auch um den Command-Button:

```
<f:ajax execute="form:in1" render="form:in2 form:in3">
    <h:panelGrid columns="1">
        <h:inputText id="in1" value="#{simpleHandler.text1}" />
        <h:inputText id="in2" value="#{simpleHandler.text2}" />
        <h:inputText id="in3" value="#{simpleHandler.text3}" />
        <h:commandButton id="button" value="Replizieren"
            action="#{simpleHandler.action}" />
    </h:panelGrid>
</f:ajax>
```

Das Panel-Grid hat kein Default-Event für das Ajax-Verhalten. Die Input-Texte haben jedoch das Value-Change-Event und der Command-Button das Action-Event als Default. Im obigen Beispiel wird also der folgende Code erzeugt.

```
...
<input id="form:in1" type="text"
    onchange="mojarra.ab(this, event, 'valueChange', ...
...
<input id="form:in2" type="text"
    onchange="mojarra.ab(this, event, 'valueChange', ...
...
<input id="form:in3" type="text"
    onchange="mojarra.ab(this, event, 'valueChange', ...
...
<input id="form:button" type="submit"
    onclick="mojarra.ab(this, event, 'action', ..."
```

Gibt man nun in der Eingabe mit der Id `in1` einen Text ein und betätigt danach die Schaltfläche, werden zwei Ajax-Events gefeuert. Zum einen das Value-Change-Event für die erste Eingabe und zum anderen das Action-Event für die Schaltfläche. Es erfolgen also zwei XMLHttpRequests inklusive zweier Antworten.

Sie können sich von diesem Verhalten mit entsprechenden Werkzeugen überzeugen. Ein sehr mächtiges Werkzeug für die Darstellung von HTML und CSS, das Monitoren von Netzwerkaktivitäten und das Debuggen von JavaScript ist *Firebug*. Wir stellen Firebug in Abschnitt 10.3 detailliert vor und ermuntern den interessierten Leser, die dort vorgestellten Analysemöglichkeiten nachzuvollziehen, da sie das aktuelle Beispiel verwenden.

Im Beispiel ist kein Listener für das Value-Change-Event registriert, so dass lediglich die Action-Methode des Command-Buttons aufgerufen wird. Eine einfache Möglichkeit zur Verifikation, dass tatsächlich zwei XMLHttpRequests abgeschickt werden, ist die Registrierung eines Listeners. Im Attribut `listener` des `<f:ajax>`-Tags wird eine Listener-Methode angegeben, die das `AjaxBehaviorListener`-Interface erfüllt.

```
<f:ajax execute="form:in1" render="form:in2 form:in3"
        listener="#{simpleHandler.ajaxBehaviorListener}">
```

Über den Parameter der Listener-Methode vom Typ `AjaxBehaviorEvent` lässt sich die UI-Komponente – die Quelle des Events – einfach ermitteln.

Die Default-Events können mit dem `event`-Attribut leicht überschrieben werden. Durch den Code

```
<f:ajax execute="form:in1" render="form:in2 form:in3"
        listener="#{simpleHandler.ajaxBehaviorListener}"
        event="click">
```

werden die Default-Events überschrieben und alle enthaltenen UI-Komponenten, die das `ClientBehaviorHolder`-Interface implementieren für das Click-Event als Ajax-Verhalten registriert. In diesem Fall also auch das Panel-Grid.

Aufgabe 7.1

Überzeugen Sie sich davon, dass durch das `event`-Attribut mit Wert `click` das Panel-Grid click-sensitiv wird.

Aufgabe 7.2

Ändern Sie den Wert des `event`-Attributs der Texteingabe im ursprünglichen Beispiel auf `blur` oder `mouseout`, um diese Events zu testen.

Aufgabe 7.3

Ändern Sie den Wert des `event`-Attributs des Command-Button im ursprünglichen Beispiel auf `mouseover`, um den Request durch einfaches Überfahren der Schaltfläche mit der Maus auszulösen.

Aufgabe 7.4

Ändern Sie das Spiel Tic-Tac-Toe aus Abschnitt 2.5 so ab, dass die Spielerzüge per Ajax übermittelt werden.

Aufgabe 7.5

In Aufgabe 4.20 haben Sie die Existenz der Dateien `jsf.js` und `jsf-uncompressed.js` in der Referenzimplementierung verifiziert. Überzeugen Sie sich nun davon, dass in der konfigurierten Projektphase **Development** (siehe Tabelle 4.12 auf Seite 163 und Abschnitt **Projektphasen** auf Seite 167) die unkomprimierte, in den anderen Projektphasen die komprimierte Version zum Einsatz kommt.

7.2.3 Das überarbeitete Bearbeitungsmodell einer JSF-Anfrage

Das in Abschnitt 4.1 eingeführte Bearbeitungsmodell einer JSF-Anfrage, kurz der *Lebenszyklus*, wurde durch die Einführung der Ajax-Funktionalität in JSF 2.0 überarbeitet. Das Modell, das wir noch einmal in Abbildung 7.1 dargestellt haben, bleibt in seinen sechs Phasen erhalten. Einzelne Phasen werden jedoch nur noch für Teile des Komponentenbaums und nicht mehr für den kompletten Komponentenbaum durchgeführt. Die Spezifikation spricht von einer partiellen Traversierungsstrategie (Partial Traversal Strategy). Es wird zwischen der Verarbeitung des Requests (Phasen 1 bis 5) und dem Rendern der Antwort (Phase 6) unterschieden. Die Phasen 1 und 5 (Wiederherstellung des Komponentenbaums und Aufruf der Anwendungslogik) nehmen jedoch an der partiellen Verarbeitung nicht teil, so dass die Phasen 2, 3 und 4 (Partial View Processing) sowie 6 (Partial View Rendering) als Ajax-relevante Phasen übrig bleiben.

Die in Abbildung 7.1 dargestellte Unterteilung in einen Verarbeitungs- (Execute) und einen Render-Teil spiegelt sich auch im Ajax-API wider. Es sind dies genau die Bezeichnungen, die wir in unseren bisherigen Beispielen als Attribute für das `<f:ajax>`-Tag bzw. als Schlüssel der Parameter-Map für den JavaScript-Aufruf verwendet haben. Im XMLHttpRequest werden die Komponenten, die in den Phasen 2, 3 und 4 verwendet werden sollen, im Execute-Parameter übergeben. Die in Phase 6 zu rendernden Komponenten werden im Render-Parameter übergeben. Die Parameter werden in der Anfrage jedoch nicht direkt verwendet, sondern bekommen den Präfix `javax.faces.partial`. Abbildung 7.2 zeigt einen Post-Request des Beispiels aus Abschnitt 7.2.2. Man erkennt, dass die Quelle des Requests die Texteingabe mit Id `form:in1` war (`javax.faces.source`). Dies ist auch die einzige Komponente, die in den Verarbeitungsphase zu aktualisieren und zu validieren ist (`javax.faces.partial.execute`). Die zu rendernden Komponenten sind die Texteingaben mit den Ids `form:id2` und `form:id3` (`javax.faces.partial.render`). In Phase 6 werden die entsprechenden Kom-

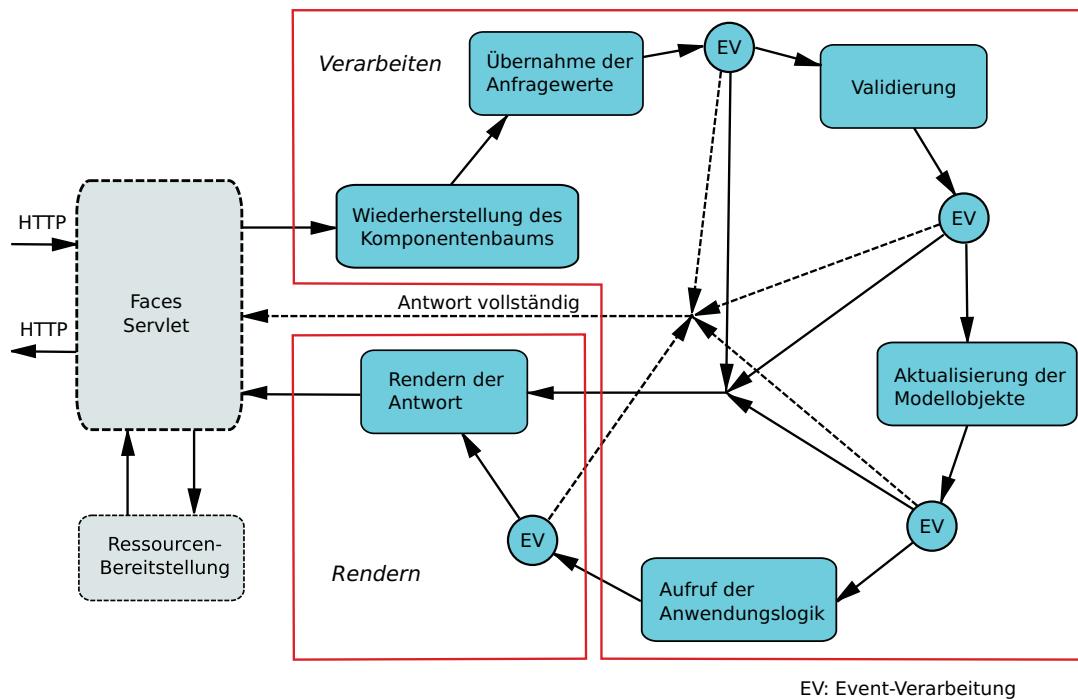


Abbildung 7.1: Lebenszyklus einer Anfrage

ponenten in ein XML-Format gerendert und an den Client geschickt. Das JSF-JavaScript hat dann die Aufgabe, diese Antwort in das DOM des Browsers einzubauen.

Für die meisten Aufgaben, die im Bereich der partiellen Verarbeitung und des partiellen Rendern realisiert werden müssen, existiert die Klasse **PartialViewContext** im Package `javax.faces.context`. Diese enthält unter anderem die booleschen Methoden `isAjaxRequest()`, `isPartialRequest()`, aber auch die Methoden `getExecuteIds()` und `getRenderIds()`, die jeweils eine Collection der entsprechenden Ids zurückliefern. Mit diesen und weiteren Methoden der Klasse **PartialViewContext** ist server-seitig eine entsprechende Verarbeitung auf Ajax-Ebene möglich.

Aufgabe 7.6

Implementieren Sie einen Phase-Listener (siehe Abschnitt 4.5.6 auf Seite 128),

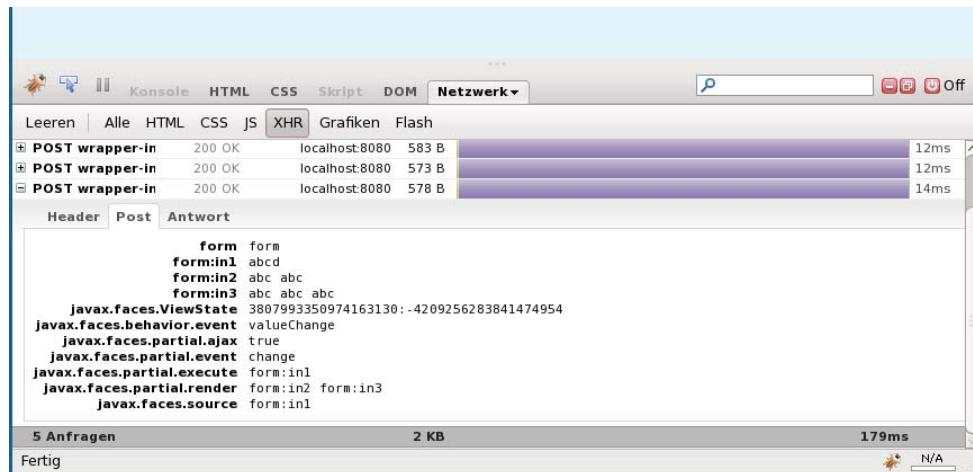


Abbildung 7.2: Post-Request-Informationen in Firebug

und überzeugen Sie sich mit dessen Hilfe, dass auch bei einem Ajax-Request alle sechs Phasen der JSF-Request-Bearbeitung durchlaufen werden.

Aufgabe 7.7

Erweitern Sie den Phase-Listener, und geben Sie die Parameter des Post-Requests aus. Verwenden Sie dazu die Methode `getRequestParameterMap()` der Klasse `ExternalContext`.

Aufgabe 7.8

Geben Sie für einen Ajax-Request die Komponenten-Ids für die Execute- und Render-Phase mit Hilfe der Methoden `getExecuteIds()` und `getRenderIds()` aus.

Aufgabe 7.9

Überarbeiten Sie das Beispiel aus Abschnitt 4.9.2, mit dem wir client- und server-seitige Ids verdeutlicht haben.

7.3 Weiterführende Themen

In diesem Abschnitt wollen wir einige Tipps im Zusammenhang mit Ajax geben und potenzielle Fallstricke ansprechen.

7.3.1 Navigation

Der bisher vorgestellte Ajax-Ansatz ging davon aus, dass das client-seitige JavaScript nach Erhalt der Antwort diese in das DOM der JSF-Seite einbaut,

die die Anfrage gestellt hat. Es wird also die Seite aktualisiert, die die Anfrage initiiert hat, und es findet keine Navigation statt. Dies ist der Standardfall eines Ajax-Requests. Es ist jedoch auch eine Navigation mit Ajax-Requests möglich. Der folgende Code gibt die Action-Methode einer Managed Bean wieder, die zu einer anderen Seite navigiert.

```
@ManagedBean
@RequestScoped
public class RedirectHandler {

    public void action() {
        FacesContext ctx = FacesContext.getCurrentInstance();
        ExternalContext extContext = ctx.getExternalContext();
        String url = extContext.encodeActionURL(
            ctx.getApplication().getViewHandler()
            .getActionURL(ctx, "/simple-button.xhtml"));
        try {
            extContext.redirect(url);
        } catch (IOException ioe) {
            throw new FacesException(ioe);
        }
    }
}
```

Wird obige Action-Methode aufgerufen, etwa durch das folgende <f:ajax>-Tag, so findet eine Navigation zur View /simple-button.xhtml statt.

```
<h:commandButton value="Redirect"
                  action="#{redirectHandler.action}">
    <f:ajax execute="@this" render="@none"/>
</h:commandButton>
```

7.3.2 JavaScript mit Java

Die Zuständigkeit von JavaScript auf dem Client und von Java auf dem Server ist nicht unumstößlich. Mit geringem Aufwand kann auf dem Server JavaScript-Code als Teilantwort für den Client aufbereitet werden. Die nachfolgende Methode `javaScriptWindow()` öffnet mit der JavaScript-Funktion `alert()` ein neues Fenster, in dem der vom Benutzer eingegebene Text des Properties `text1` erscheint.

```
1  public void javaScriptWindow() {
2      FacesContext ctx = FacesContext.getCurrentInstance();
3      ExternalContext extContext = ctx.getExternalContext();
4      if (ctx.getPartialViewContext().isAjaxRequest()) {
5          try {
6              extContext.setResponseContentType("text/xml");
7              extContext.addResponseHeader("Cache-Control", "no-cache");
```

```
8     PartialResponseWriter writer =
9     ctx.getPartialViewContext().getPartialResponseWriter();
10    writer.startDocument();
11    writer.startEval();
12    writer.write("alert('Die Eingabe ist \\\""
13                  + text1 + "\\\"');");
14    writer.endEval();
15    writer.endDocument();
16    writer.flush();
17    ctx.responseComplete();
18 } catch (Exception e) {
19     logger.warning("Fehler in 'javaScriptWindow()' ");
20 }
21 }
22 }
```

Man erkennt in den Zeilen 12/13 die Konstruktion der `alert()`-Funktion und deren Parameter. Wenn die dargestellte Methode über einen Ajax-Aufruf, etwa durch das Blur-Event wie im folgenden Code, aufgerufen wird, öffnet sich ein JavaScript-Fenster mit der entsprechenden Meldung.

```
<h:inputText value="#{simpleHandler.text1}">
  <f:ajax listener="#{simpleHandler.javaScriptWindow}"
    event="blur" />
</h:inputText>
```

Das Browser- sowie das JavaScript-Fenster stellt Abbildung 7.3 dar.

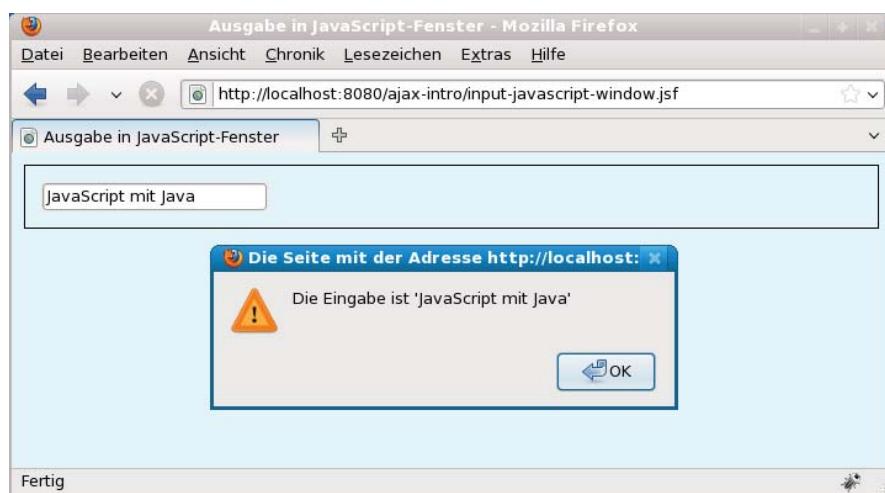


Abbildung 7.3: Texteingabe in JavaScript-Fenster

Die Verwendung des umgekehrten Schrägstrichs und der Apostrophe in der Funktion `alert()` erscheint auf den ersten Blick unnötig und kompliziert. JavaScript hat jedoch direkten Zugang zum Browser-API, so dass eine potentielle Verwundbarkeit einer Anwendung durch Cross-Site-Scripting besteht. Wenn man die Zeilen 12/13 der oben dargestellten Methode `javaScriptWindow()` durch den folgenden Code ersetzt, kann dies zu Sicherheitsproblemen führen.

```
writer.write("alert('Die Eingabe ist ' + " + text1 + ");");
```

Der in `text1` enthaltene Text wird direkt von JavaScript interpretiert. Enthält er Ausdrücke, die JavaScript-Objekte benennen oder JavaScript-Funktionen aufrufen, kann dies zur Kompromittierung des Systems führen. Abbildung 7.4 zeigt dies am Beispiel des Ausdrucks `document.URL`.

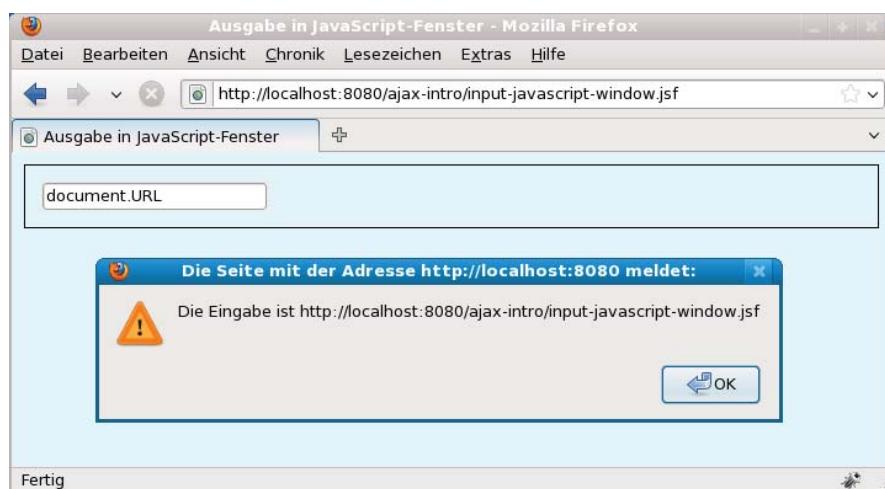


Abbildung 7.4: Verwundbarkeit durch Cross-Site-Scripting

7.3.3 Nicht gerenderte Komponenten

Mit dem `render`-Attribut des `<f:ajax>`-Tags werden die auf dem Client zu aktualisierenden Komponenten bestimmt. Wir haben dies bereits in mehreren Beispielen praktiziert. Problematisch wird dies bei JSF-Seiten, deren Inhalte sich ändern, weil das Rendern von Komponenten über ihr `rendered`-Attribut gesteuert wird. Wir erläutern dies an einem Beispiel. Die JSF-Seite in Listing 7.1 auf der nächsten Seite steuert über drei Radioknöpfe die Eingabemöglichkeiten für eine Zahlung per Bankeinzug, Kreditkarte oder Rechnung.

Listing 7.1: radio.xhtml

```
1 <h:form id="form">
2   <h:panelGrid columns="1">
3     <h:outputText value="Zahlung per" />
4     <h:selectOneRadio value="#{radioHandler.radio}">
5       <f:ajax render="@form" />
6       <f:selectItem itemValue="1" itemLabel="Bankeinzug"/>
7       <f:selectItem itemValue="2" itemLabel="Kreditkarte"/>
8       <f:selectItem itemValue="3" itemLabel="Rechnung"/>
9     </h:selectOneRadio>
10   </h:panelGrid>
11   <h:panelGrid id="einzug" columns="2"
12     rendered="#{radioHandler.radio == 1}">
13     <h:outputLabel value="Kontoinhaber" />
14     <h:inputText value="#{radioHandler.kontoinhaber}" />
15     <h:outputLabel value="Kontonummer" />
16     <h:inputText value="#{radioHandler.kontonummer}" />
17     <h:outputLabel value="Bankleitzahl" />
18     <h:inputText value="#{radioHandler.blz}" />
19   </h:panelGrid>
20   <h:panelGrid id="karte" columns="2"
21     rendered="#{radioHandler.radio == 2}">
22     <h:outputLabel value="Kreditkarte" />
23     <h:inputText value="#{radioHandler.kkart}" />
24     <h:outputLabel value="gültig bis" />
25     <h:inputText value="#{radioHandler.gueltigBis}" />
26     <h:outputLabel value="Kartennummer" />
27     <h:inputText value="#{radioHandler.nummer}" />
28   </h:panelGrid>
29   <h:panelGrid id="rechnung" columns="2"
30     rendered="#{radioHandler.radio == 3}">
31     <h:outputLabel value="Vor-/Nachname" />
32     <h:inputText value="#{radioHandler.name}" />
33     <h:outputLabel value="Straße Nummer" />
34     <h:inputText value="#{radioHandler.strasseNr}" />
35     <h:outputLabel value="PLZ Ort" />
36     <h:inputText value="#{radioHandler.plzOrt}" />
37   </h:panelGrid>
38 </h:form>
```

Die drei Zahlungsalternativen werden jeweils mit einem `<h:panelGrid>`-Tag realisiert. Diese haben die Ids `einzug`, `karte` und `rechnung` (Zeilen 11, 20, 29). Die Darstellung der Panel-Grids wird dynamisch in Abhängigkeit des jeweiligen `rendered`-Attributs gesteuert. Abbildung 7.5 zeigt die Darstellung für den Bankeinzug und die Kreditkarte.

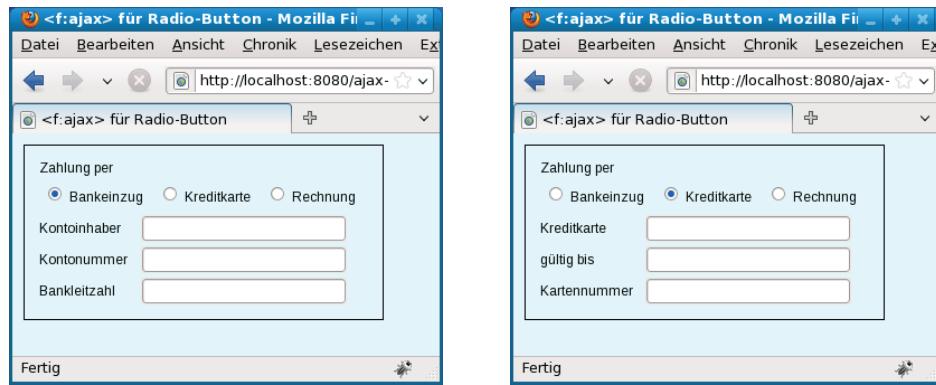


Abbildung 7.5: Auswahl der Zahlungsart über Radio-Button

In Zeile 5 wird das `render`-Attribut des `<f:ajax>` auf das komplette Formular gesetzt:

```
<f:ajax render="@form" />
```

Sinnvoll wäre jedoch ein ausschließliches Aktualisieren der drei Panel-Grids durch:

```
<f:ajax render="form:einzug form:karte form:rechnung" />
```

Dies führt jedoch nicht zum gewünschten Ergebnis, da maximal eines der drei Panel-Grids gerendert wird, die anderen jedoch nicht. Daher sind die nicht gerenderten Panel-Grids nicht im DOM enthalten und lassen sich auch nicht per Ajax aktualisieren. Es muss also das ganze Formular gerendert werden.

7.3.4 Abgekürzte Komponenten-Ids

Bei der Einführung des `<f:ajax>`-Tags haben wir im `execute`- und `render`-Attribut die Komponenten-Ids in der Präfix-Schreibweise verwendet, wie es JSF vorsieht (siehe Abschnitt 4.9 auf Seite 180). Dies ist im folgenden Code-Ausschnitt noch einmal beispielhaft vorgestellt.

```
<h:form id="form">
    <h:panelGrid columns="1">
        <h:inputText id="in1" ... />
        <h:inputText id="in2" ... />
        <h:inputText id="in3" ... />
        <h:commandButton id="button" ... >
            <f:ajax execute="form:in1" render="form:in2 form:in3" />
        </h:commandButton>
    </h:panelGrid>
</h:form>
```

Das Voranstellen der Formular-Id geschieht automatisch, falls das `prependId`-Attribut nicht auf `false` gesetzt ist, wie im folgenden Code-Ausschnitt:

```
<h:form id="form" prependId="false">
    <h:panelGrid columns="1">
        <h:inputText id="in1" ... />
        <h:inputText id="in2" ... />
        <h:inputText id="in3" ... />
        <h:commandButton id="button" ... >
            <f:ajax execute="in1" render="in2 in3" />
        </h:commandButton>
    </h:panelGrid>
</h:form>
```

Man erkennt hier, dass die Werte des `execute`- und `render`-Attributs ohne den Präfix auskommen. Bei der Referenzimplementierung können alle möglichen Kombinationen verwendet werden, also etwa auch die Verwendung von Präfixen trotz `prependId="false"`. Sogar Mischformen sind möglich:

```
<h:form id="form" prependId="false">
    ...
        <f:ajax execute="in1" render="in2 form:in3" />
    ...

```

Wir empfehlen Ihnen aber eine konsequente Anwendung einer der beiden Alternativen. Der JSF-Entwickler muss sich hier keine Gedanken machen, sehr wohl aber der JavaScript-Entwickler.

Aufgabe 7.10

Verwenden Sie das Attribut `prependId` mit beiden Wahrheitswerten und abwechselnder Präfix-Schreibweise. Inspizieren Sie den generierten HTML- und JavaScript-Code, und achten Sie auf die `id`-Attribute.



Projekt

Der in den Abschnitten 7.2 und 7.3 beschriebene Code ist im Projekt *ajax-intro* enthalten.

7.4 Ajax mit RichFaces

Seit der Veröffentlichung der ersten JSF-Version im Jahre 2004 hat sich das Web rasant weiterentwickelt. Die modulare und für Erweiterungen offene Architektur von JSF haben dazu beigetragen, dass sich eine ganze Reihe von JSF-Komponentenbibliotheken mit Ajax-Unterstützung etabliert haben, lange bevor JSF selbst in der Version 2.0 Ajax-Unterstützung einführte. ICEfaces

[URL-ICE] und RichFaces [URL-JRF] sind wahrscheinlich die bekanntesten und am weitesten verbreiteten Bibliotheken, deren Komponenten Ajax unterstützen bzw. die es ermöglichen, bestehende JSF-Komponenten mit Ajax-Funktionalität zu versehen. Beide Bibliotheken wurden ursprünglich im kommerziellen Umfeld entwickelt und von den Firmen ICEsoft und Exadel auch kommerziell vertrieben, sind jedoch mittlerweile den Schritt zu Open-Source-Lizenzen gegangen. Wir beschreiben hier und im Abschnitt 7.5 Teile der RichFaces-Bibliothek. Dies sollte jedoch nicht als qualitative Wertung gesehen werden, und wir ermuntern den Leser ausdrücklich, sich auch die ICEfaces-Bibliothek näher anzuschauen.

Die RichFaces-Bibliothek besteht aus zwei Bibliotheken: der Ajax4jsf- und der eigentlichen RichFaces-Bibliothek. Angaben zur unglücklichen Namensgebung und Entstehungsgeschichte von RichFaces findet man in [Kat08]. Zusammen mit [Fil09] sind dies im Augenblick die einzigen Bücher zum Thema RichFaces. Die Ajax4jsf-Bibliothek besteht aus Komponenten, die es ähnlich wie `<f:ajax>` erlauben, bestehende konventionelle JSF-Komponenten um Ajax-Funktionalität zu erweitern. Im Folgenden wollen wir diese Komponenten in Anlehnung an den XML-Namensraum als A4J-Komponenten bezeichnen. Die eigentliche RichFaces-Bibliothek besteht aus komplexen und mächtigen eigenständigen Komponenten, wie etwa Baum- und Menü-Komponenten und Komponenten zur Realisierung von Drag and Drop. Diese wollen wir ebenfalls in Anlehnung an den XML-Namensraum als Rich-Komponenten bezeichnen.

In Version 3.3 besteht die A4J-Bibliothek aus 24, die Rich-Bibliothek aus 72 Komponenten. Wir können an dieser Stelle daher nur einen sehr kleinen Teil der Komponenten ansprechen, beginnen mit den A4J-Komponenten und widmen Abschnitt 7.5 den Rich-Komponenten.

Bemerkung: Die RichFaces-Bibliothek ist im Sommer 2010 noch nicht für JSF 2.0 verfügbar. Hier verwenden wir die Version 3.3.3 Beta 1, die zwar auf einer JSF-2.0-Implementierung lauffähig ist, doch ohne 2.0-spezifische Eigenschaften und Komponenten zu unterstützen. So muss etwa Facelets explizit als zusätzliche Bibliothek verwendet werden und nicht die eingebaute Facelets-Version. Weiterhin sind alle Tags der Version 2.0, etwa `<f:ajax>`, nicht verfügbar. Mit der im Jahresverlauf 2010 erwarteten Version 4.0 wird JSF 2.0 dann vollständig unterstützt.

7.4.1 Die `<a4j:support>`-Komponente

Die `<a4j:support>`-Komponente erlaubt es, bestehende konventionelle Komponenten mit Ajax-Funktionalität zu versehen, ähnlich wie die `<f:ajax>`-Komponente. Da die A4J-Bibliothek eigene Komponenten für Schaltflächen

(`<a4j:commandButton>`) und Links (`<a4j:commandLink>`) bereitstellt, ist die `<a4j:support>`-Komponente für JSF-Eingaben prädestiniert. Das folgende Beispiel verbindet einen Input-Text und einen Output-Text ähnlich unserem `<f:ajax>`-Beispiel in Abschnitt 7.2.2. Die Attribute `action` und `actionListener` für das Binden einer Action- bzw. Action-Listener-Methode sind ebenfalls verfügbar.

```
<h:inputText value="#{utilHandler.text}">
    <a4j:support event="onkeyup" reRender="text" />
</h:inputText>
<h:outputText id="text" value="#{utilHandler.text}" />
```

Im Beispiel wird nach jedem Tastaturanschlag die Komponente mit der Id `text` mit dem neuen Property-Wert der Eingabe versehen. Die Ähnlichkeit zum JSF-Tag `<f:ajax>` ist offensichtlich. In der Beta-Version von RichFaces 4.0 wurde das Tag und das `reRender`-Attribut daher umbenannt. Bei der Nennung des Events ging man außerdem auf die JSF-Schreibweise ohne das JavaScript-Präfix `on` über, so dass sich das Beispiel in Version 4.0 folgendermaßen darstellt:

```
<h:inputText value="#{utilHandler.text}">
    <a4j:ajax event="keyup" render="text" />
</h:inputText>
<h:outputText id="text" value="#{utilHandler.text}" />
```

Die `<a4j:support>`-Komponente kennt insgesamt 25 Attribute. Diese – und die Attribute der noch vorzustellenden weiteren Komponenten – können wir nicht vollständig beschreiben. Zwei Attribute erscheinen jedoch besonders erwähnenswert: `requestDelay` und `eventsQueue`. Während Ajax aus Benutzersicht eine kürzere Antwortzeit und somit eine höhere Interaktivität bietet, ist auf der Server-Seite eine andere Sichtweise angebracht. Im Beispiel wird bei *jedem* Tastaturanschlag ein XMLHttpRequest an den Server geschickt und eventuell eine Action-Methode aufgerufen. Dies kann eine immense Belastung des Servers bedeuten. Mit den Attributen `requestDelay` und `eventsQueue`, die auch bei anderen Komponenten dieses Abschnitts (`<a4j:poll>`, `<a4j:commandLink>`, `<a4j:commandButton>`) vorhanden sind, kann ein Fein-Tuning der Request-Folgen erfolgen, um die Server-Belastung zu reduzieren.

Mit dem Attribut `eventsQueue` wird der Name einer Queue definiert. Die Requests aller Komponenten, die diese Queue verwenden, werden zunächst in dieser Queue zwischengespeichert. Das Abschicken eines Requests geschieht erst, wenn der vorherige Request und seine Antwort verarbeitet wurden. Dabei wird nur der letzte aktuelle Request nach Eingang der Antwort abgeschickt, die an-

deren Request werden übergangen. Man nutzt also den Queue-Mechanismus, um Sequenzen von kurz hintereinander erzeugten Requests zu optimieren.

Eine weitere Möglichkeit, das durch Ajax verursachte Übertragungsvolumen zu reduzieren, ist das Attribut `requestDelay`. Der Wert des Attributs wird als die Zeitdauer in Millisekunden interpretiert, die der Request vor dem Abschicken warten muss. Auch hier wird nur der aktuellste Request tatsächlich gesendet, Duplikate werden ignoriert.

Zur Verdeutlichung des Verhaltens erweitern wir obiges Beispiel zu

```
<a4j:support event="onkeyup" reRender="text"
    eventsQueue="myQueue" requestDelay="2000" />
```

Nach Betätigen der ersten Taste wartet RichFaces zunächst zwei Sekunden, bis der Request verschickt wird. Benötigt die Verarbeitung auf dem Server etwas länger, sagen wir 5 Sekunden, so werden alle Tastaturanschläge während des Wartens auf die Antwort ignoriert, außer dem letzten. Dieser wird unter Beachtung der zweisekündigen Wartezeit als Request verschickt.

Das beschriebenen Verhalten wäre auch in JSFs `<f:ajax>`-Tag sehr sinnvoll. Nach einem Blog-Eintrag von Jim Driscoll [URL-RA], einem der Entwickler der Referenz-Implementierung, kann dies selbst realisiert werden und wird voraussichtlich in JSF 2.1 Einzug finden.

7.4.2 Die `<a4j:outputPanel>`-Komponente

In Abschnitt 7.3.3 auf Seite 250 haben wir das Problem nicht gerenderter und somit nicht im Komponentenbaum vorhandener Komponenten angesprochen. Dieses Problem löst RichFaces mit der `<a4j:outputPanel>`-Komponente. Ein Beispiel für `<a4j:outputPanel>` gibt uns die Möglichkeit, anzumerken, dass die normale JSF-Request-Bearbeitung auch mit RichFaces beibehalten wird und z. B. Validierungen vorgenommen werden. Im folgenden Programmausschnitt verwenden wir eine Validierungsmethode (`validateMinMax()`) und ein `<h:message>`-Tag zur Anzeige einer etwaigen Fehlermeldung.

```
Geben Sie eine ganze Zahl ein. <br/>
Das 1,5-fache wird neben der Eingabe wiederholt. <br/>
<h:inputText id="eingabe" value="#{utilHandler.ganzzahl}"
    validator="#{utilHandler.validateMinMax}">
    <a4j:support event="onkeyup" reRender="produkt" />
</h:inputText>
<br/>
<h:outputText id="produkt"
    value="#{utilHandler.ganzzahl * 1.5}" />
<br/>
<a4j:outputPanel ajaxRendered="true">
```

```
<h:message for="eingabe" />
</a4j:outputPanel>
```

Die Validierungsmethode, auf deren Darstellung wir verzichten, prüft auf einen Wert zwischen 0 und 10. Bei einem anderen Wert wird eine JSF-Message erzeugt, die mit `<h:message>` angezeigt wird. Das `<a4j:outputPanel>`-Tag setzt einen per Ajax realisierten Ausgabebereich um. Im Beispiel entsteht das in Abschnitt 7.3.3 diskutierte Problem nicht, da wir auf das Attribut `rendered` verzichten. Im allgemeinen Fall kann aber das `<a4j:outputPanel>`-Tag das Darstellungsproblem lösen. In unserem Beispiel ist noch erwähnenswert, dass durch das `ajaxRendered`-Attribut die Komponente bei jeder Ajax-Antwort für diese Seite gerendert wird, obwohl im `reRender`-Attribut des `<a4j:support>`-Tags die Komponente nicht aufgeführt ist.

7.4.3 Die `<a4j:region>`-Komponente

Die bisherigen RichFaces-Beispiele besaßen jeweils nur eine Eingabekomponente. Falls das Formular aus mehreren Eingabekomponenten besteht, werden alle Eingabekomponenten auf dem Server aktualisiert. RichFaces sieht mehrere Alternativen vor, um nur Teile des Komponentenbaums zu aktualisieren, d. h. die Phasen 2 bis 4 der JSF-Bearbeitung durchzuführen. Eine Alternative ist die Verwendung `<a4j:region>`-Komponente.

Das folgende Beispiel besitzt vier textuelle Eingabekomponenten und drei Möglichkeiten, eine Anfrage zu initiieren. Zwei davon bestehen in Vorgriff auf Abschnitt 7.4.4 aus dem `<a4j:commandButton>`-Tag, die dritte aus einem `<h:commandButton>`-Tag.

```
1  <a4j:region>
2      <h:outputLabel value="Text 1:" for="text1" />#160;
3      <h:inputText id="text1" value="#{regionHandler.text1}" />
4      <h:outputLabel value="Text 2:" for="text2" />#160;
5      <h:inputText id="text2" value="#{regionHandler.text2}" />
6      <br />
7      <a4j:commandButton action="#{regionHandler.action}"
8          value="Aktualisieren von Text 1 und Text 2"
9          reRender="output" />
10     </a4j:region>
11     <br />
12     <a4j:region>
13         <h:outputLabel value="Text 3:" for="text3" />#160;
14         <h:inputText id="text3" value="#{regionHandler.text3}" />
15         <h:outputLabel value="Text 4;" for="text4" />#160;
16         <h:inputText id="text4" value="#{regionHandler.text4}" />
17         <br />
18         <a4j:commandButton action="#{regionHandler.action}"
```

```
19         value="Aktualisieren von Text 3 und Text 4"
20         reRender="output" />
21     </a4j:region>
22     <br/><br/>
23     Aktualisiert wurden: <br/>
24     <a4j:outputPanel id="output">
25       Text 1: <h:outputText value="#{regionHandler.text1}" /><br/>
26       Text 2: <h:outputText value="#{regionHandler.text2}" /><br/>
27       Text 3: <h:outputText value="#{regionHandler.text3}" /><br/>
28       Text 4: <h:outputText value="#{regionHandler.text4}" /><br/>
29   </a4j:outputPanel>
30   <br />
31   <h:commandButton action="#{regionHandler.action}"
32     value="Alle Eingaben aktualisieren" />
```

Die `<a4j:region>`-Komponente definiert in den Zeilen 1 bis 10 und 12 bis 21 zwei strukturell jeweils identische Bereiche, die aus je zwei Texteingaben und einer Schaltfläche bestehen. Die Schaltflächen sind als `<a4j:commandButton>`-Tags und insofern mit Ajax-Unterstützung ausgeführt. Ein Klick auf eine solche Schaltfläche führt zu einem Request, der nur die Eingaben innerhalb der Ajax-Region bearbeitet, in diesem Fall also `text1` und `text2` in der ersten und `text3` und `text4` in der zweiten Region. Zur Kontrolle werden die Werte der Eingaben in den Zeilen 25 bis 28 ausgegeben.

Im Unterschied dazu führt die Schaltfläche, die mittels `<h:commandButton>` in Zeile 31/32 realisiert ist, zur Verarbeitung aller Eingaben auf dem Server.

Eine weitere Möglichkeit zur Beschränkung der Komponenten, die auf dem Server verarbeitet werden, ist das boolesche Attribut `ajaxSingle`. Es beschränkt die Verarbeitung auf die Komponente, die den Ajax-Request ausgelöst hat. Das Attribut ist für die beiden Steuerkomponenten `<a4j:commandButton>` und `<a4j:commandLink>` definiert, jedoch insbesondere bei der `<a4j:support>`-Komponente sinnvoll einsetzbar.

```
<h:inputText value="#{utilHandler.text}">
  <a4j:support ajaxSingle="true" ... />
</h:inputText>
```

In diesem Beispiel wird die Verarbeitung auf die dargestellte Eingabekomponente beschränkt, auch wenn sich noch weitere Eingabekomponenten in der Seite befinden.

Als letzte Möglichkeit einer Definition der auf dem Server zu verarbeitenen Komponenten erlaubt das `process`-Attribut die direkte Benennung dieser Komponenten. Es ist sowohl für `<a4j:support>` als auch für `<a4j:commandButton>` und `<a4j:commandLink>` definiert.

Im folgenden Beispiel wird durch das `ajaxSingle`-Attribut zunächst die Verarbeitung auf die Eingabe `text1` eingeschränkt, um danach durch das `process`-Attribut auf die Eingabe `text3` erweitert zu werden. Die einzige nicht verarbeitete Eingabe ist `text2`.

```
<h:inputText id="text1" value="#{regionHandler.text1}">
    <a4j:support event="onkeyup" ajaxSingle="true"
        reRender="output" process="text3"/>
</h:inputText><br/>
<h:inputText id="text2" value="#{regionHandler.text2}" /><br/>
<h:inputText id="text3" value="#{regionHandler.text3}" />
```

7.4.4 Die `<a4j:commandButton>`- und `<a4j:commandLink>`-Komponenten

Die beiden Komponenten `<a4j:commandButton>` und `<a4j:commandLink>` entsprechen ihren JSF-Äquivalenten `<h:commandButton>` und `<h:commandLink>`. Der einzige Unterschied ist, dass das Formular als Ajax-Request abgeschickt wird und die Antwort für das dynamische und Ajax-basierte Rendern von Komponenten verwendet werden kann.

Im folgenden Beispiel wird sowohl die Eingabe- als auch die Ausgabekomponente erneut gerendert, nachdem die Ajax-Antwort eingegangen ist.

```
<h:inputText id="eingabe" value="#{utilHandler.text}" />
<br/>
<a4j:commandLink value="Bitte klicken"
    action="#{utilHandler.ajaxAction}"
    reRender="eingabe,ausgabe" />
<br/>
<h:outputText id="ausgabe" value="#{utilHandler.text}" />
```

7.4.5 Die `<a4j:poll>`-Komponente

Die durch Ajax erhoffte höhere Interaktivität einer Anwendung lässt sich mit der `<a4j:poll>`-Komponente um eine weitere Facette bereichern. Mit dieser Komponente wird eine periodische Interaktivität zwischen Client und Server ohne Benutzeraktivität realisiert. Durch zyklisches Polling des Servers können beliebige Server-Daten auf den Client übertragen und dargestellt werden. Wir zeigen dies im folgenden Beispiel anhand der aktuellen Uhrzeit des Servers:

```
<h:form>
    <a4j:poll id="poll" interval="2000"
        enabled="#{uhrzeitHandler.pollen}" reRender="uhrzeit" />
</h:form>
<h:form>
```

```
<h:outputText id="uhrzeit"
    value="Zeit auf dem Server: #{uhrzeitHandler.zeit}" />
<br/><br/>
<a4j:commandButton id="umschalten"
    value="Pollen \"
        #{uhrzeitHandler.pollen ? 'aus' : 'ein'}schalten"
    action="#{uhrzeitHandler.umschalten}"
    reRender="poll, umschalten"/>
</h:form>
```

In diesem Beispiel wird das Polling auf ein Intervall von 2000 ms gesetzt. Der Default beträgt 1000 ms. Mit dem `reRender`-Attribut wird die Ausgabekomponente `uhrzeit` aktualisiert. Das Beispiel erlaubt das Ein- und Ausschalten des Pollings, was über das boolesche Attribut `enabled` erfolgt. Das entsprechende Bean-Property, das noch zu verwendende Property für die Uhrzeit sowie die Action-Methode für das Ein- und Ausschalten sind einfach zu realisieren:

```
public class UhrzeitHandler {

    private boolean pollen = true;

    public String umschalten() {
        pollen = ! pollen;
        return null;
    }

    public String getZeit() {
        DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
        return dateFormat.format(new Date());
    }
    ...
}
```

Da das Polling über einen gewöhnlichen Ajax-Request erfolgt, muss sich das Tag innerhalb eines Formulars befinden. Um eine Beeinflussung zwischen Polling und Anwendungslogik zu vermeiden, ist für die Anwendungslogik ein zweites Formular zu verwenden. Im Beispiel enthält dieses die Ausgabe der Uhrzeit sowie ein `<a4j:commandButton>`-Tag, um die Action-Methode für das Ein- und Ausschalten des Pollings zu realisieren.

7.4.6 Die `<a4j:log>`-Komponente

Wir haben das Firefox-Plugin Firebug bereits als geeignetes System für die Analyse und das Debugging von Ajax-Request empfohlen. RichFaces erlaubt darüber hinaus mit der `<a4j:log>`-Komponente das Nachvollziehen der durch einen Ajax-Request initiierten client-seitigen JavaScript-Aktivitäten. Durch Betätigen von *Strg-Shift-L* in einer JSF-Seite, die die `<a4j:log>`-Komponente

enthält, öffnet sich ein Fenster, in das RichFaces client-seitige Informationen, z. B. Request- und Response-Daten, vorgenommene Änderungen am DOM-Baum oder die Suche nach Komponenten mit bestimmten Ids, schreibt.

Wichtige Attribute der Komponente sind das boolesche `popup`, mit dem man das Öffnen eines neuen Fensters steuert (Default) oder die Log-Meldungen in das bestehende Browser-Fenster schreibt oder das `level`-Attribut, mit dem der Log-Level eingestellt wird. Mögliche Werte sind `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG` und `ALL`.



Projekt

Das Projekt `richfaces-a4j` enthält Beispiele für alle im Abschnitt 7.4 vorgestellten A4J-Komponenten, also `<a4j:support>`, `<a4j:outputPanel>`, `<a4j:region>`, `<a4j:commandLink>`, `<a4j:commandButton>`, `<a4j:poll>` und `<a4j:log>`.

Aufgabe 7.11

Überzeugen Sie sich mit Hilfe einer zeitaufwändigen Berechnung innerhalb einer Action-Methode auf dem Server (Tipp: `Thread.sleep()`), dass die Verwendung der Attribute `eventsQueue` und `requestDelay` tatsächlich zu einer Reduktion der Ajax-Request-Anzahl führt.

Aufgabe 7.12

Ändern Sie die Default-Tastenkombination für das Öffnen des JavaScript-Debug-Fensters in `Strg-Shift-D` ab. Verwenden Sie hierzu das Attribut `hotkey` des `<a4j:log>`-Tags.

7.5 RichFaces-Komponenten mit eingebauter Ajax-Unterstützung

RichFaces stellt eine große Auswahl an Komponenten mit eingebauter Ajax-Unterstützung zur Verfügung. Dies reicht von grundlegenden Elementen, wie etwa die Auswahl von Alternativen mit einer Combo-Box oder der Zahleneingabe mit einem Schieberegler über anwendungsorientierte Elemente, wie etwa Menükomponenten, bis zur Integration externer Funktionalitäten wie etwa Google Maps und Microsoft Virtual Earth.

Wir stellen an dieser Stelle drei Komponenten bzw. Komponentengruppen vor, deren Funktionalität weit über konventionelle Web-Anwendungen hinausgeht. Dies sind Komponenten zur Realisierung von Drag and Drop, eine

Komponente zur Realisierung von Bäumen, d. h. zur Visualisierung hierarchischer Datenstrukturen, und eine Komponente, um durch umfangreiche Datensets blättern zu können. Für weitere Komponenten verweisen wir auf das RichFaces-Manual sowie auf eine Online-Demo aller Komponenten, die über die RichFaces-Home-Page [URL-JRF] erreicht werden kann.

7.5.1 Drag and Drop

Zur Realisierung von Drag and Drop stellt RichFaces die folgenden Komponenten zur Verfügung:

- `<rich:dragSupport>`
- `<rich:dropSupport>`
- `<rich:dragIndicator>`
- `<rich:dndParam>`
- `<rich:dragListener>`
- `<rich:dropListener>`

Für die Realisierung eines sinnvollen Anwendungsfalls benötigt man mindestens die ersten drei Komponenten. Unser Beispiel wird auch die vierte Komponente verwenden, kommt aber ohne die beiden letztgenannten aus.

Als Beispiel entwickeln wir eine kleine Shop-Anwendung. Der Benutzer kann zwischen verschiedenen Artikeln, in unserem Fall Obstsorten, auswählen und diese mit Hilfe von Drag and Drop in einen Warenkorb legen. Der initiale Zustand der Anwendung ist in Abbildung 7.6 auf der nächsten Seite wiedergegeben.

Man erkennt in dem mit *Artikelwahl* überschriebenen Bereich acht Obstsorten, die drag-sensitiv sind, und in dem mit *Warenkorb* überschriebenen Bereich den eigentlichen Warenkorb, der drop-sensitiv ist. Zusätzlich enthält dieser Bereich eine leere Fläche, die später die textuelle Repräsentation des Warenkorbinhalts wiedergeben wird.

Die Artikel werden durch die Klasse `Artikel` repräsentiert, die lediglich aus einem Namen und einem Preis sowie den entsprechenden Konstruktoren und Getter/Setter-Paaren besteht:

```
public class Artikel {  
  
    private String name;  
    private BigDecimal preis;  
    ...
```

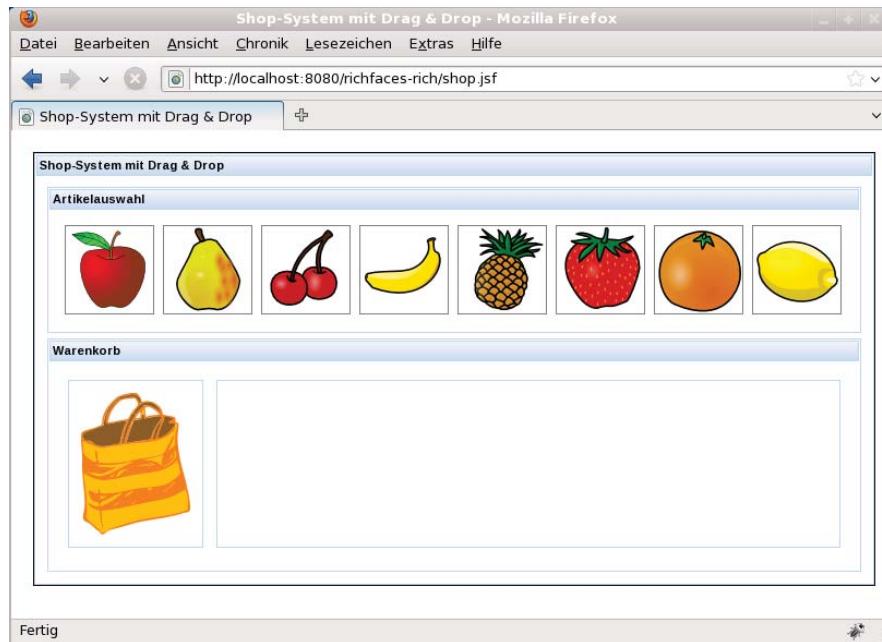


Abbildung 7.6: Initialer Zustand der Shop-Anwendung

Da die JSF-Seite relativ umfangreich ist, analysieren wir zunächst den oberen Bereich, die Artikelauswahl, die die Drag-Unterstützung enthält.

```
1  <rich:dragIndicator id="indicator">
2      <f:facet name="single">
3          {marker} {label}
4      </f:facet>
5  </rich:dragIndicator>
6
7  <h:form id="form">
8      <rich:panel>
9          <f:facet name="header">
10             Shop-System mit Drag & Drop
11         </f:facet>
12         <rich:panel styleClass="auswahl">
13             <f:facet name="header">Artikelauswahl</f:facet>
14             <a4j:repeat value="#{dndBean.artikel}" var="artikel">
15                 <a4j:outputPanel styleClass="dragPanel">
16                     <rich:toolTip>
17                         <span style="white-space:nowrap">
18                             <strong>#{artikel.name}</strong><br />
19                             Preis: #{artikel.preis}
20                         </span>
21                     </rich:toolTip>
22             <rich:dragSupport dragIndicator="indicator"
```

```
23             dragType="Artikel" dragValue="#{artikel}">
24     <rich:dndParam name="label"
25         value="#{artikel.name}" />
26   </rich:dragSupport>
27   <h:graphicImage value="/images/#{artikel.name}.png"
28     height="80" width="80" alt="#{artikel.name}" />
29   </a4j:outputPanel>
30 </a4j:repeat>
31 </rich:panel>
```

Zunächst wird in den Zeilen 1 bis 5 ein Drag-Indicator definiert. Mit einem Drag-Indicator definiert RichFaces, was unter der Maus während einer Drag and Drop Operation angezeigt wird. Dazu wird die Facette **single** (für ein einzelnes Objekt) verwendet, innerhalb deren vordefinierte und selbst definierter Parameter beliebig angeordnet werden können. Im Beispiel ist „**{marker}**“ ein vordefinierter Parameter, der während des Drag and Drop verschiedene Zustände annehmen kann. Wenn die Maus in einen Bereich kommt, in den das gezogene Objekt fallen gelassen werden kann, symbolisiert dies eine optische Zustandsänderung. Der zweite verwendete Parameter, „**{label}**“, wird in den Zeilen 24/25 definiert.

Im den weiteren Zeilen werden einige RichFaces-Tags verwendet, die nichts mit Drag and Drop zu tun haben. Ein **<rich:panel>** definiert ein Panel mit einem Standard-Layout, das unter anderem mit einer Header-Facette eine Überschrift des Panels ermöglicht (Zeilen 8 und 12). Das **<a4j:repeat>** (Zeile 14) stellt eine Iterationskomponente dar, die die Verwendung der Parameter **value** und **var** analog zu **<h:dataTable>** erlaubt. Das **<a4j:outputPanel>**-Tag, das wir schon in Abschnitt 7.4.2 eingeführt haben, verwenden wir, weil die umgebende Vaterkomponente von **<rich:dragSupport>** und **<rich:dropSupport>** die Attribute **onmouseover** und **onmouseout** unterstützen muss.

Schließlich definieren wir mit **<rich:Tooltip>** ein Tooltip. Im Beispiel (Zeilen 16 bis 21) wird der Name und der Preis des Artikels angezeigt. Zur Verdeutlichung zeigt Abbildung 7.7 auf der nächsten Seite den aktivierten Tooltip für die Banane.

Zurück zu Drag and Drop: Mit **<rich:dragSupport>** in den Zeilen 22 bis 26 werden die ziehbaren Objekte definiert. Das Attribut **dragIndicator** benennt den zu verwendenden Drag-Indicator, den wir in den Zeilen 1 bis 5 definiert hatten. RichFaces ermöglicht eine Unterscheidung zwischen verschiedenen Objekten für die Operationen des Drag and Drop. Mit dem Attribut **dragType** wird ein Objekt mit einem Typ versehen. Dieser kann dann verwendet werden, um in verschiedenen Drop-Zonen bestimmte Typen zuzulassen und andere auszuschließen. In unserem Beispiel wird nicht unterschieden, und

es gibt nur den einen Typ **Artikel**. Da als Werte des Attributs `dragType` beliebige EL-Ausdrücke zugelassen sind, können z.B. über Properties einfache Unterscheidungen vorgenommen werden. Das Attribut `dragValue` bestimmt das eigentliche Objekt der Operation. Mit `<h:graphicImage>` werden schließlich die Graphiken für die Artikel erzeugt. Man erkennt, dass der Dateiname für die Graphik dem Artikelnamen entsprechen muss.

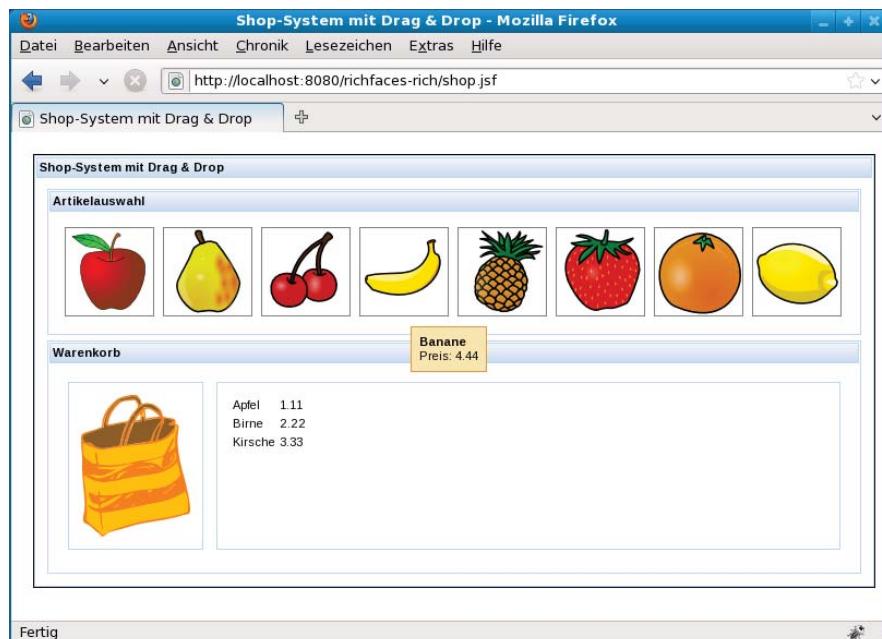


Abbildung 7.7: Gefüllter Warenkorb und aktiver Tooltip für den Artikel *Banane*

Der untere Teil der Seite ist zweigeteilt und enthält in den Zeilen 39 bis 46 den drop-sensitiven Warenkorb und in den Zeilen 47 bis 53 eine einfache Tabelle zur Darstellung der Warenkorbinhalte.

```
35      <rich:panel styleClass="warenkorb">
36          <f:facet name="header">Warenkorb</f:facet>
37          <h:panelGrid id="tab"
38              columns="2" columnClasses="wkBild ,wkTabelle">
39              <rich:panel>
40                  <rich:dropSupport id="warenkorb"
41                      acceptedTypes="Artikel"
42                      dropListener="#{eventBean.processDrop}"
43                      reRender="warenkorbtablelle" />
44                  <h:graphicImage value="/images/Einkaufstuetje.png"
45                      height="130" width="100" alt="Einkaufstüte" />
```

```
46      </rich:panel>
47      <rich:panel styleClass="warenkorbtafelle">
48          <h: dataTable id="warenkorbtafelle"
49              value="#{dndBean.warenkorb}" var="art">
50              <h: column>#{art.name}</h: column>
51              <h: column>#{art.preis}</h: column>
52          </h: dataTable>
53      </rich:panel>
54  </h: panelGrid>
55  </rich:panel>
56  </rich:panel>
57 </h: form>
```

In den Zeilen 40 bis 43 wird mit dem `<rich:dropSupport>`-Tag die Warenkorbgraphik der Zeilen 44/45 drop-sensitiv gemacht. Mit dem Attribut `acceptedTypes` werden die für Drop-Operationen erlaubten Typen definiert. Das Attribut `reRender` benennt die nach Erhalt der Ajax-Antwort zu aktualisierende Komponente. Mit dem Attribut `dropListener` wird die Listener-Methode für das Drop-Event definiert, auf die wir gleich eingehen. Abgeschlossen wird die Seite durch eine einfache Tabelle, die mit `<h: dataTable>` erzeugt wird.

Die Listener-Methode für das Drop-Event wird in der Klasse `EventBean` realisiert. Die Methode `processDrop()` ist die einzige im Interface `DropListener` geforderte Methode. Über den Parameter vom Typ `DropEvent` kann auf den Wert des fallen gelassenen Objekts zugegriffen werden:

```
@ManagedBean
@RequestScoped
public class EventBean implements DropListener {

    @ManagedProperty(" #{dndBean}")
    private DndBean dndBean;

    public void processDrop(DropEvent dropEvent) {
        dndBean.zuWarenkorbHinzufuegen(dropEvent.getDragValue());
    }
    ...
}
```

Interessant ist hier die `@ManagedProperty`-Annotation, über die ein `DndBean`-Objekt injiziert wird. Alternativ wäre in der JSF-Konfigurationsdatei das Tag `<managed-property>` zu verwenden. Erläuterungen hierzu finden sich in Abschnitt 4.3.

Die Klasse `DndBean` ist ebenfalls recht einfach aufgebaut. Sie besteht im Wesentlichen aus zwei Properties für die Artikelauswahl und den Warenkorb so-

wie der Methode `zuWarenkorbHinzufuegen()`, die in obiger `processDrop()`-Methode aufgerufen wird.

```
@ManagedBean  
@SessionScoped  
public class DndBean {  
  
    private List<Artikel> artikel;  
    private List<Artikel> warenkorb;  
  
    public void zuWarenkorbHinzufuegen(Object obj) {  
        warenkorb.add((Artikel) obj);  
    }  
    ...
```

Wir beschließen hiermit unsere Ausführungen zum Thema Drag and Drop mit RichFaces und verweisen den Leser für weitere Recherchen auf das RichFaces-Handbuch und die RichFaces-Web-Seite, die eine Demo-Anwendung für Drag and Drop enthält.

7.5.2 Bäume

Die RichFaces-Tree-Komponente realisiert die Darstellung hierarchischer Daten. Sie muss daher an ein Modellobjekt gebunden werden, das diese Daten ebenfalls in hierarchischer Form bereitstellt. Das Modellobjekt muss das Interface `TreeNode` im Package `org.richfaces.model` implementieren. Alternativ kann man die Klasse `TreeNodeImpl` im selben Package verwenden, die als Dateninhalte beliebige Typen zulässt und das Interface implementiert. Wir verwenden hier `TreeNodeImpl` und entwickeln eine Reihe kleiner Anwendungen, um verschiedene Aspekte der RichFaces-Tree-Komponente zu erläutern.

Für das erste Beispiel erstellen wir zunächst server-seitig das Datenmodell mit Hilfe der Klasse `TreeNodeImpl`. Die Daten sind einfache Strings, die die Hierarchie anhand eines einfachen Nummerierungsschemas repräsentieren. Die Klasse `TreeHandler1` ist eine Managed Bean, deren Methode `getTreeNode()` die Wurzel des Baums zurückgibt.

```
@ManagedBean  
@SessionScoped  
public class TreeHandler1 {  
  
    private TreeNode<String> rootNode = null;  
  
    public TreeNode<String> getTreeNode() {  
        if (rootNode == null) {  
            loadTree();  
        }  
    }
```

```
        return rootNode;
    }

    private void loadTree() {
        rootNode = new TreeNodeImpl<String>();
        TreeNodeImpl<String> node1 = new TreeNodeImpl<String>();
        node1.setData("Knoten 1");
        rootNode.addChild(1, node1);
        TreeNodeImpl<String> node11 = new TreeNodeImpl<String>();
        node11.setData("Knoten 11");
        node1.addChild(3, node11);
        TreeNodeImpl<String> node12 = new TreeNodeImpl<String>();
        node12.setData("Knoten 12");
        node1.addChild(4, node12);
        TreeNodeImpl<String> node121 = new TreeNodeImpl<String>();
        node121.setData("Knoten 121");
        node12.addChild(5, node121);
        TreeNodeImpl<String> node122 = new TreeNodeImpl<String>();
        node122.setData("Knoten 122");
        node12.addChild(6, node122);
        TreeNodeImpl<String> node123 = new TreeNodeImpl<String>();
        node123.setData("Knoten 123");
        node12.addChild(7, node123);
        TreeNodeImpl<String> node2 = new TreeNodeImpl<String>();
        node2.setData("Knoten 2");
        rootNode.addChild(2, node2);
    }
    ...
}
```

Dies ist auf Java-Seite ausreichend, um den Baum erstellen zu können. In JSF genügt bereits die Verwendung des Tags `<rich:tree>`, dessen Attribut `value` mit einer Wertebindung diesen Getter aufruft.

```
<h:form>
    <rich:panel>
        <rich:tree value="#{treeHandler1.treeNode}" />
    </rich:panel>
</h:form>
```

Abbildung 7.8 zeigt die Darstellung des Baums.

Das Tag `<rich:tree>` erlaubt die Verwendung von fast 90 Attributen, die das Aussehen des Baums und sein Anwendungsverhalten bestimmen. Wir erweitern `<rich:tree>` um die Attribute, die für eine einfache Selektion eines Knotens nötig sind und das Übertragungsverhalten beeinflussen.

```
<rich:panel>
    <f:facet name="header">
        Einfacher Baum mit &#60;rich:tree&#62;
    </f:facet>
```

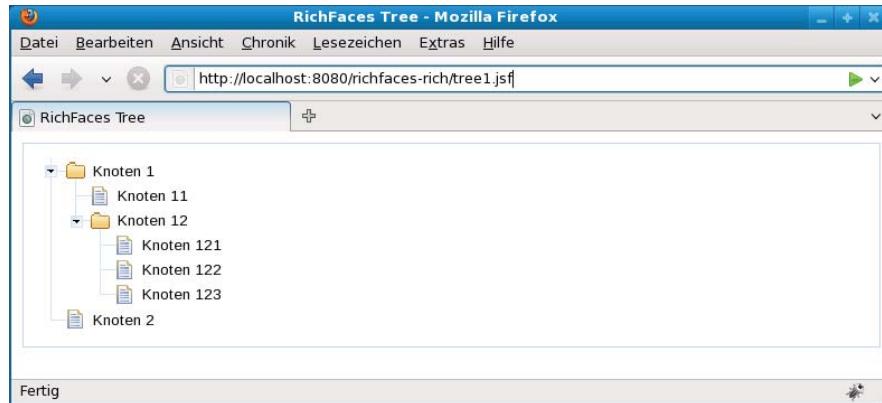


Abbildung 7.8: Einfacher Baum mit <rich:tree>

```
<h:outputText id="selectedNode"
    value="Ausgewählter Knoten: #{treeHandler1.nodeType}" />
<rich:tree reRender="selectedNode" ajaxSubmitSelection="true"
    nodeSelectListener="#{treeHandler1.processSelection}"
    switchType="client" value="#{treeHandler1.treeNode}">
</rich:tree>
</rich:panel>
```

Das Attribut `reRender` ist bereits bekannt und benennt die zu aktualisierende Komponente. Über `nodeSelectListener` wird eine Listener-Methode gebunden, die bei der Selektion eines Knotens aufgerufen wird. Der Aufruf erfolgt per Ajax, da `ajaxSubmitSelection` gesetzt ist. Über den `switchType` wird bestimmt, ob das Aufklappen eines Knotens und damit die Darstellung der Sohnknoten ausschließlich auf dem Client oder mit einem JSF- oder Ajax-Request realisiert wird. Wir gehen darauf später ausführlich ein.

Die Verbindung zwischen dem selektierten Knoten und der Anzeige des Knoteninhalts erfolgt über die schon erwähnte Listener-Methode, im Beispiel `processSelection()`. Die Klasse `TreeHandler1` benötigt zusätzlich das Property `nodeTitle`:

```
private String nodeTitle;

public void processSelection(NodeSelectedEvent event) {
    UITree tree = (UITree) event.getComponent();
    nodeTitle = (String) tree.getRowData();
}
```

Abbildung 7.9 zeigt die Darstellung des Baums mit selektiertem Knoten 122.

Als nächsten Schritt erweitern wir unser Beispiel um dynamische Daten, das heißt Daten, die wir nicht in der Managed Bean erzeugen. Als Datenquel-

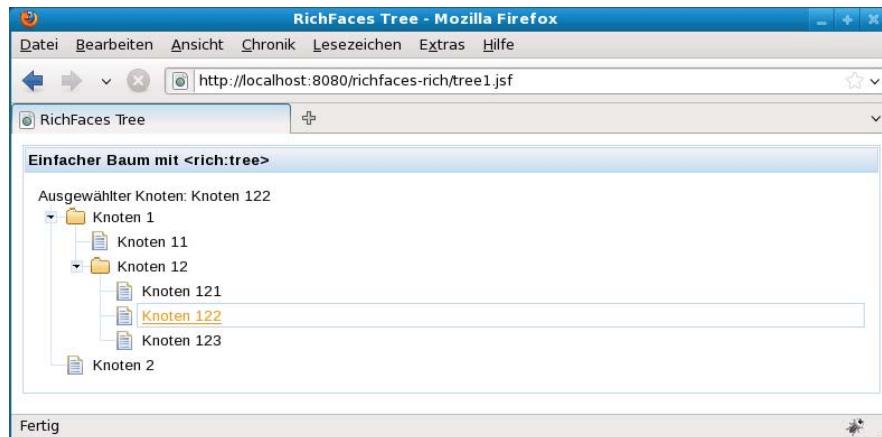


Abbildung 7.9: Selektierter Knoten 122

le wählen wir die JSF-Konfigurationsdatei `faces-config.xml`, die als XML-Datei eine hierarchische Struktur besitzt. Die JSF-Seite selbst erfährt keine Änderungen. In der Managed Bean muss für die Erzeugung des Modellbaums die Datei `faces-config.xml` geparsst und entsprechend aufbereitet werden. Diese XML-Verarbeitung hat jedoch nichts mit dem RichFaces-Tree zu tun, so dass wir auf eine Darstellung des Java-Codes verzichten und dem Leser das Studium des herunterladbaren Quell-Codes empfehlen. Die Darstellung der XML-Datei veranschaulicht Abbildung 7.10.

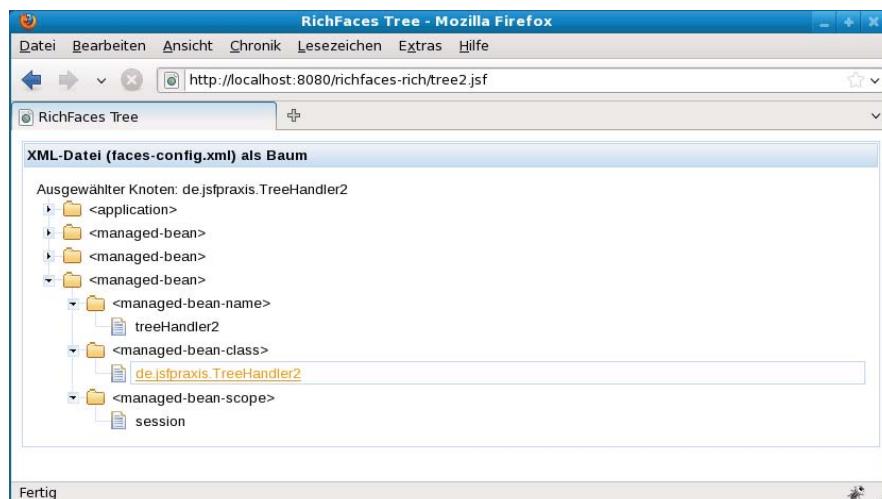


Abbildung 7.10: Darstellung der Datei `faces-config.xml`

Die Baumkomponente erlaubt Anpassungen der Visualisierung der Baumknoten. Hierzu wird die bisher als Datentyp verwendete String-Klasse durch eine anwendungsdefinierte Klasse ersetzt. Für unser drittes Beispiel wählen wir die Klasse MyModel, so dass sich der folgende Code für die Managed Bean ergibt.

```
@ManagedBean
@SessionScoped
public class TreeHandler3 {

    private TreeNode<MyModel> rootNode = null;

    private MyModel selected;

    public TreeNode<MyModel> getTreeNode() {
        if (rootNode == null) {
            loadTree();
        }
        return rootNode;
    }

    ...

    public static class MyModel {

        private String name;
        private String data;

        ...

        public String getType() {
            return name.substring(name.lastIndexOf("-") + 1);
        }
    }
}
```

Die zentrale Methode ist der Getter `getType()`. Das Property `name` enthält jeweils die Namen der XML-Tags, das Property `data` ihre Inhalte. Der Getter `getType()` liefert für die Tags zur Definition von Managed Beans je nach Tag die Strings "bean", "class", "name" und "scope" zurück. Im `<rich:tree>`-Tag können dann mit dem Attribut `nodeFace` dieser Getter verwendet, in den enthaltenen `<rich:treeNode>`-Tags mit dem Attribut `type` eine Fallunterscheidung vorgenommen und je nach Typ verschiedene Icons zur Darstellung verwendet werden. Mit dem Attribut `icon` wird das Icon eines inneren Knotens, mit dem Attribut `iconLeaf` das Icon eines Blattes definiert. Die Darstellung der Seite im Browser zeigt Abbildung 7.11.

```
<rich:tree reRender="selectedNode" ajaxSubmitSelection="true"
```

```
nodeSelectListener="#{treeHandler3.processSelection}"
switchType="client" value="#{treeHandler3.treeNode}"
var="item" nodeFace="#{item.type}">
<rich:treeNode type="bean" icon="/images/bean.png">
    &lt;#{item.name}&gt;
</rich:treeNode>
<rich:treeNode type="class" iconLeaf="/images/class.png">
    &lt;#{item.name}&gt;#{item.data}&lt;#{item.name}&gt;
</rich:treeNode>
<rich:treeNode type="name" iconLeaf="/images/name.png">
    &lt;#{item.name}&gt;#{item.data}&lt;#{item.name}&gt;
</rich:treeNode>
<rich:treeNode type="scope" iconLeaf="/images/scope.png">
    &lt;#{item.name}&gt;#{item.data}&lt;#{item.name}&gt;
</rich:treeNode>
<rich:treeNode type="default">
    &lt;#{item.name}&gt;
</rich:treeNode>
</rich:tree>
```

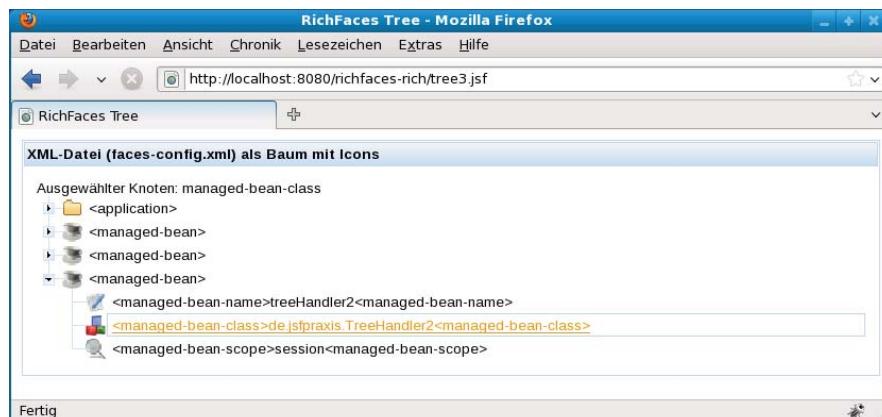


Abbildung 7.11: Darstellung der Datei faces-config.xml mit Icons

Im vierten Beispiel verzichten wir auf die Verwendung der Klasse `TreeNode-Impl` und bauen das Baummodell mit Standarddatenstrukturen des Java-SDK, namentlich Arrays und Listen. Auf JSF-Seite verwenden wir das Tag `<rich:treeNodesAdaptor>`-Tags, das als Adapter zwischen dem RichFaces-Baum und der Nicht-Baumstruktur auf dem Server dient. Wir beginnen mit der Java-Klasse `TreeHandler4`.

```
@ManagedBean
@SessionScoped
public class TreeHandler4 {
```

```
public String[] getNodes() {
    return new String[] { "aaa", "bbb", "ccc" };
}
public List<NodeLevel1> getNodesLevel1() {
    List<NodeLevel1> nodes = new ArrayList<NodeLevel1>();
    nodes.add(new NodeLevel1("nat",
        new Integer[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }));
    nodes.add(new NodeLevel1("prim",
        new Integer[] { 2, 3, 5, 7, 11, 13,
            17, 19, 23, 29, 31 }));
    nodes.add(new NodeLevel1("fibonacci",
        new Integer[] { 0, 1, 1, 2, 3, 5,
            8, 13, 21, 34, 55 }));
    return nodes;
}

public static class NodeLevel1 {

    private String label;
    private Integer[] nodes;

    // Getter und Setter ...
}

}
```

Die Managed Bean `TreeHandler4` besitzt die beiden Getter `getNodes()` und `getNodesLevel1()`. Die erste Methode gibt ein einfaches Array von Strings, die zweite eine Liste von `NodeLevel1`-Objekten zurück. Ein solches Objekt besteht aus einem Property `label` (ein String) und einem Property `nodes` (ein Array von Integern). Diese beiden Datenstrukturen können unter Verwendung der Tags `<rich:treeNodesAdaptor>` und `<rich:treeNode>` sehr einfach als Baum dargestellt werden, wie die folgende JSF-Seite zeigt.

```
1  <rich:panel>
2      <f:facet name="header">
3          Beispiel &lt;rich:treeNodesAdaptor&gt;
4      </f:facet>
5      <rich:tree switchType="client" ajaxSubmitSelection="true"
6          reRender="status">
7          <rich:treeNodesAdaptor id="einfach"
8              nodes="#{treeHandler4.nodes}" var="node">
9              <rich:treeNode>
10                 #{node}
11             </rich:treeNode>
12         </rich:treeNodesAdaptor>
13         <rich:treeNodesAdaptor id="level1"
14             nodes="#{treeHandler4.nodesLevel1}" var="l1">
```

```
15      <rich:treeNode>
16          #{l1.label}
17      </rich:treeNode>
18      <rich:treeNodesAdaptor id="level2"
19          nodes="#{l1.nodes}" var="l2">
20          <rich:treeNode>
21              #{l2}
22          </rich:treeNode>
23      </rich:treeNodesAdaptor>
24      </rich:treeNodesAdaptor>
25  </rich:tree>
26  <rich:panel>
27      <f:facet name="header">Status</f:facet>
28      <h:outputText value="#{treeHandler4.status}" id="status" />
29  </rich:panel>
30 </rich:panel>
```

In den Zeilen 7 bis 12 wird in einem `<rich:treeNodesAdaptor>` als Knotenmenge mit dem Attribut `nodes` der erste Getter des Handlers verwendet. Das Attribut `var` definiert eine Iterationsvariable, die innerhalb des enthaltenen `<rich:treeNode>` verwendet wird.

Der Code-Abschnitt in den Zeilen 13 bis 24 enthält zwei `<rich:treeNodesAdaptor>`-Komponenten, wobei die zweite innerhalb der ersten verwendet wird. Dies ist nötig, weil der Getter `getNodesLevel1()` eine baumartige Struktur mit zwei Ebenen liefert, die jeweils innerhalb der Darstellung zu verwenden sind. Die Darstellung des Baums zeigt Abbildung 7.12.

Für einen Baum der Höhe h müssen h `<rich:treeNodesAdaptor>`-Tags verwendet werden. Für beliebige Bäume ist diese Methode also nicht praktikabel. Hier verwendet man das Tag `<rich:recursiveTreeNodesAdaptor>`, das wir auch in unserem letzten Beispiel für RichFaces-Bäume verwenden.

Das fünfte Beispiel nutzen wir zur Darstellung und Diskussion des Attributs `switchType` des `<rich:tree>`-Tags. Mit ihm wird das Verhalten beim Aufklappen eines Knotens festgelegt. Mögliche Werte des Attributs sind `client`, `server` und `ajax`. Default ist `ajax`. Beim Wert `client` wird der komplette Baum in den Browser geladen. Beim Aufklappen eines Knotens findet daher keine Kommunikation mit dem Server statt, so dass dies das schnellste Arbeiten auf dem Client erlaubt. Allerdings benötigt der Browser mehr Platz, um den kompletten Baum vorzuhalten. Beim Wert `server` findet ein normaler JSF-Request statt, so dass ein sichtbarer Neuaufbau des Baums im Browser erfolgt und z.B. bei einem eventuell vorhandenen Rollbalken die Positionierung verloren geht. Beim Wert `ajax` findet ein Ajax-Request statt, der die soeben genannten Nachteile vermeidet, da nur die beim Aufklappen eines Knotens neu benötigten Sohnknoten erfragt und in die Darstellung eingebaut

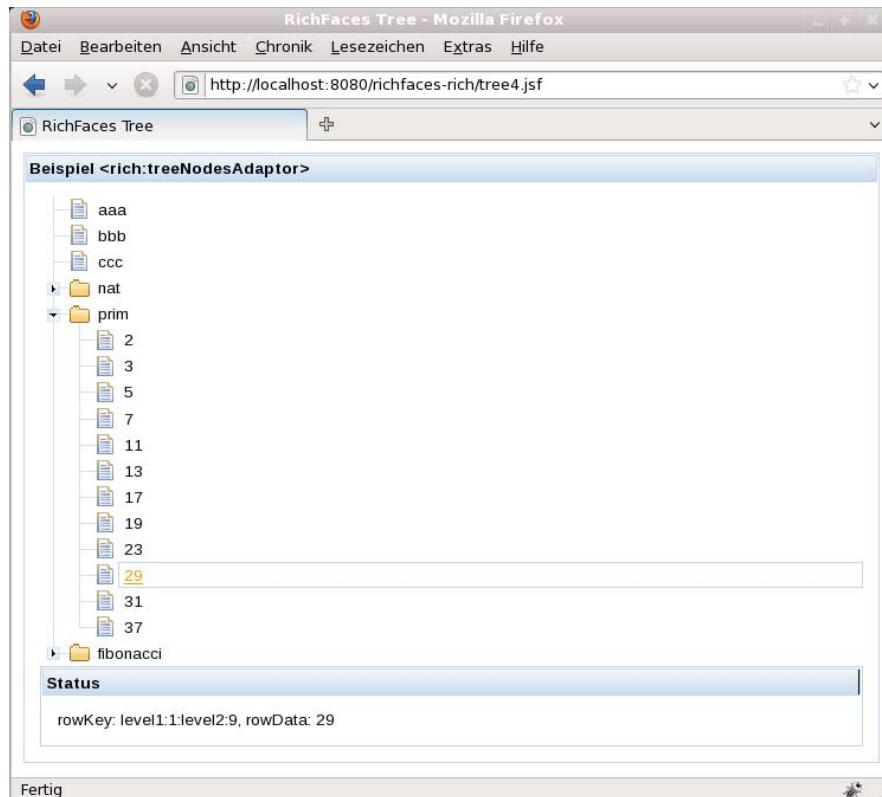


Abbildung 7.12: Baum mit `<rich:treeNodeAdaptor>`

werden. Der folgende Code zeigt das Beispiel mit den Tags `<rich:tree>` und `<rich:recursiveTreeNodesAdaptor>`.

```
<rich:panel>
    <f:facet name="header">
        Beispiel: Baum mit &lt;rich:recursiveTreeNodesAdaptor&gt;
        und #{treeHandler5.anzahlKnoten} Knoten
    </f:facet>
    <rich:tree switchType="client" ajaxSubmitSelection="true">
        <rich:recursiveTreeNodesAdaptor id="rek" var="node"
            roots="#{treeHandler5.roots}" nodes="#{node.children}">
            <rich:treeNode>
                #{node.data}
            </rich:treeNode>
        </rich:recursiveTreeNodesAdaptor>
    </rich:tree>
</rich:panel>
```

Man erkennt bei `<rich:recursiveTreeNodesAdaptor>` das Attribut `nodes`, das den Zugriff auf Sohnknoten beschreibt, und `roots`, das eine Collection von Wurzelknoten enthält. Um beliebig viele Knoten im Baum erzeugen zu können, definieren wir eine rekursive Methode, die die Söhne eines Knotens erzeugt. Dabei werden in der n -ten Ebene $n+1$ Söhne erzeugt. Die Gesamtgröße des Baums berechnet sich also aus der Summe der Fakultäten $\sum_{i=1}^n n!$. Als Knotendaten wählen wir eine einfache textuelle Form von aneinandergehängten Nummern, die jeweils die Stellung eines Knotens innerhalb der jeweiligen Ebene darstellen. Der folgende Code zeigt die Implementierung, die Abbildung 7.13 auf der nächsten Seite die Darstellung mit einem Baum der Höhe 9.

```
@ManagedBean
@SessionScoped
public class TreeHandler5 {

    private static final int TREE_DEPTH = 7;
    private Node root;
    private int counter = 0;

    public List<Node> getRoots() {
        List<Node> roots = new ArrayList<Node>(1);
        roots.add(getRoot());
        return roots;
    }

    private Node getRoot() {
        if (root == null) {
            root = init(1, new Node("1"));
        }
        return root;
    }

    private Node init(int level, Node node) {
        if (level < TREE_DEPTH) {
            node.children = new ArrayList<Node>(level + 1);
            for (int i = 1; i < level + 2; i++) {
                node.children.add(
                    init(level + 1, new Node(node.data + "." + i)));
            }
        }
        return node;
    }

    public static class Node {
        private String data;
```

```
private List<Node> children;  
...  
}  
}
```

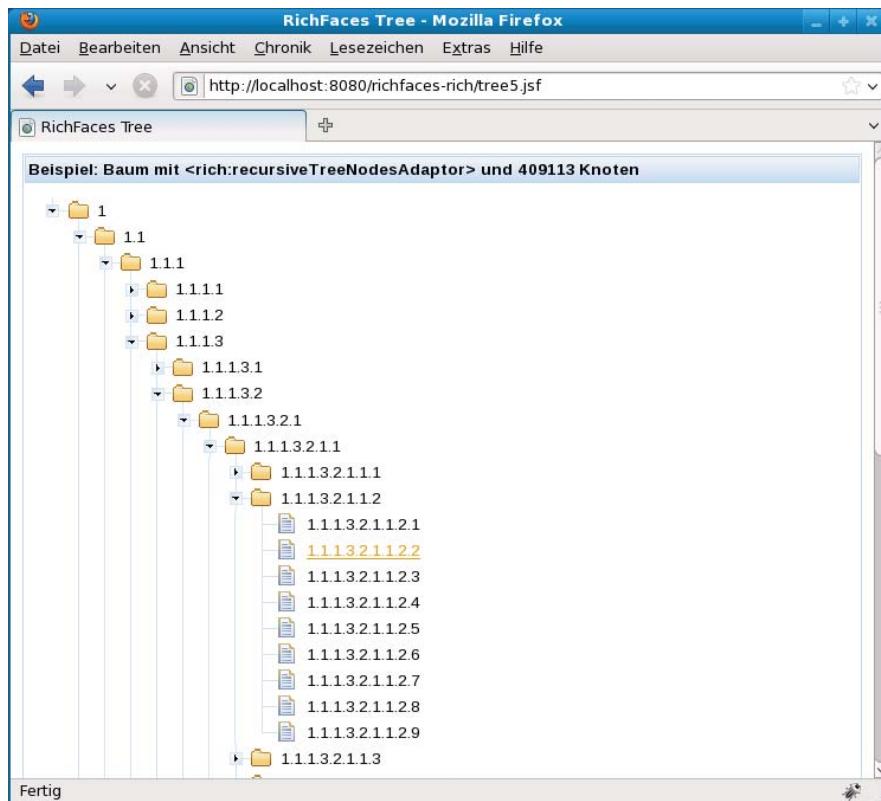


Abbildung 7.13: Baum mit generierten Knoten

Aufgabe 7.13

Das Attribut `adviseNodeOpened` des `<rich:tree>`-Tags erlaubt über eine Methodenbindung die programmatische Statusänderung eines Baumknotens. Die Methode hat die Signatur

```
public Boolean adviseNodeOpened(UITree tree)
```

und steuert über den Rückgabewert, ob ein Knoten auf- oder zugeklappt ist. Schreiben Sie eine solche Methode, die für jeden Knoten `TRUE` zurückliefert, und überzeugen Sie sich, dass alle Knoten des Baums beim ersten Aufruf aufgeklappt sind.

Aufgabe 7.14

Überzeugen Sie sich durch alternative Verwendung der Werte `client`, `server` und `ajax` für das Attribut `switchType` des `<rich:tree>`-Tags vom oben beschriebenen Verhalten.

Aufgabe 7.15

Beim Autor erzeugt ein Wert von 9 für die Konstante `TREE_DEPTH` einen `OutOfMemoryError` auf dem Server. Beim Herunterladen des gesamten Baums in den Browser beläuft sich das Datenvolumen bei einem Wert von 7 auf über 10 MB. Bei einem Wert von 8 bricht Firefox das Herunterladen ab. Finden Sie für Ihre Systemumgebung die entsprechenden Werte heraus.

7.5.3 Darstellung großer Datenmengen

Häufig sind Datenmengen so groß, dass eine vollständige Darstellung in einer Web-Seite nicht sinnvoll ist. RichFaces bietet mehrere Möglichkeiten, große Datenmengen ausschnittsweise darzustellen und in ihnen zu navigieren. Wir stellen hier zwei Möglichkeiten für Datentabellen vor. Zum einen eine Leiste mit Vor- und Zurück-Schaltflächen, zum anderen einen herkömmlichen Scrollbar. Als Testdaten verwenden wir das offizielle Bankleitzahlenverzeichnis der Bundesbank [URL-BLZ] vom 7. Juni 2010, das 19859 Datensätze enthält.

Die Verwendung der Schaltflächenleiste zum Blättern in der Datentabelle gestaltet sich mit RichFaces äußerst einfach. Es muss lediglich das Tag `<rich:datascroller>` als Facette der Datentabelle verwendet werden. Als Datentabelle verwenden wir nicht `<h:dataTable>`, sondern das RichFaces-eigene `<rich:dataTable>`, da diese Komponente die zeilenweise Aktualisierung der Tabelle erlaubt.

```
<rich:dataTable value="#{blzHandler.blzs}" var="blz" rows="15">
    <f:facet name="footer">
        <rich:datascroller />
    </f:facet>
    <rich:column id="blz">
        <f:facet name="header">Bankleitzahl</f:facet>
        #{blz.blz}
    </rich:column>
    <rich:column id="bezeichnung">
        <f:facet name="header">Bezeichnung</f:facet>
        #{blz.bezeichnung}
    </rich:column>
    ...
</rich:dataTable>
```

Man erkennt, dass sowohl `<rich: dataTable>` als auch `<rich: datascroller>` mit minimaler Attributanzahl verwendet werden. Trotzdem führt diese Variante schon zu einer attraktiven Darstellung, wie Abbildung 7.14 zeigt. Das Performanzverhalten ist ebenfalls sehr gut.

The screenshot shows a Mozilla Firefox browser window with the title bar "`<rich:datascroller> - Mozilla Firefox`". The address bar displays "`http://localhost:8080/richfaces-rich/bankleitzahlen-1.jsf`". The main content area contains a table titled "Große Datenmengen mit <rich:datascroller>". The table has columns: Bankleitzahl, Bezeichnung, PLZ, Ort, and Kurzbezeichnung. The data includes entries for various banks in Berlin and Brandenburg. Below the table is a navigation bar with page numbers from 1 to 10 and buttons for first, previous, next, and last pages. The status bar at the bottom left says "Fertig".

Bankleitzahl	Bezeichnung	PLZ	Ort	Kurzbezeichnung
10000000	Bundesbank	10591	Berlin	BBk Berlin
10010010	Postbank	10916	Berlin	Postbank Berlin
10010111	SEB	10789	Berlin	SEB Berlin
10010222	The Royal Bank of Scotland, Niederlassung Deutschland	10105	Berlin	RBS NDL Deutschland
10010424	Aareal Bank	10666	Berlin	Aareal Bank
10019610	Dexia Kommunalbank Deutschland	10969	Berlin	Dexia Berlin
10020000	Berliner Bank - alt-	10890	Berlin	Berliner Bank -alt-
10020200	BHF-BANK	10117	Berlin	BHF-BANK Berlin
10020400	Parex Bank Berlin	10117	Berlin	Parex Bank Berlin
10020500	Bank für Sozialwirtschaft	10178	Berlin	Bank für Sozialwirtschaft
10020890	UniCredit Bank - HypoVereinsbank	10896	Berlin	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	14532	Kleinmachnow	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	16515	Oranienburg	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	14776	Brandenburg an der Havel	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	15711	Königs Wusterhausen	UniCredit Bank-HypoVereinbk

Abbildung 7.14: Tabelle mit `<rich:datascroller>`

Wie bei RichFaces üblich, besitzt auch `<rich:datascroller>` die Möglichkeit zur Anpassung an spezielle Anforderungen, zum einen über Attribute, zum anderen über Facetten. Das folgende Beispiel zeigt, wie die Default-Anzeige der aktuellen Seitenzahl über die `pages`-Facette verändert werden kann. Im Beispiel erfolgt die Darstellung nach dem Muster „aktuelle Seite / maximale Seitenanzahl“. Abbildung 7.15 zeigt die Browser-Darstellung.

```
<rich: dataTable value="#{blzHandler.blzs}" var="blz" rows="15">
    <f:facet name="footer">
        <rich:datascroller pageIndexVar="currentPage"
            pagesVar="maxPages">
            <f:facet name="pages">
                #{currentPage} / #{maxPages}
            </f:facet>
        </rich:datascroller>
    </f:facet>
```

...

```
</rich:dataTable>
```

The screenshot shows a Mozilla Firefox browser window with the title "<rich:datascroller> - Mozilla Firefox". The address bar displays "http://localhost:8080/richfaces-rich/bankleitzahlen-1.jsf". The main content area contains a table titled "Große Datenmengen mit <rich:datascroller>". The table has columns: Bankleitzahl, Bezeichnung, PLZ, Ort, and Kurzbezeichnung. The data consists of 15 rows of bank information. At the bottom of the table, there is a navigation bar with buttons for "Vorher", "Nächste", "1 / 1324", "Nächste Seite", and "Letzte Seite". Below the table, the status bar says "Fertig".

Bankleitzahl	Bezeichnung	PLZ	Ort	Kurzbezeichnung
10000000	Bundesbank	10591	Berlin	BBk Berlin
10010010	Postbank	10916	Berlin	Postbank Berlin
10010111	SEB	10789	Berlin	SEB Berlin
10010222	The Royal Bank of Scotland, Niederlassung Deutschland	10105	Berlin	RBS NDL Deutschland
10010424	Aareal Bank	10666	Berlin	Aareal Bank
10019610	Dexia Kommunalbank Deutschland	10969	Berlin	Dexia Berlin
10020000	Berliner Bank - alt-	10890	Berlin	Berliner Bank -alt-
10020200	BHF-BANK	10117	Berlin	BHF-BANK Berlin
10020400	Parex Bank Berlin	10117	Berlin	Parex Bank Berlin
10020500	Bank für Sozialwirtschaft	10178	Berlin	Bank für Sozialwirtschaft
10020890	UniCredit Bank - HypoVereinsbank	10896	Berlin	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	14532	Kleinmachnow	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	16515	Oranienburg	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	14776	Brandenburg an der Havel	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	15711	Königs Wusterhausen	UniCredit Bank-HypoVereinbk

Abbildung 7.15: <rich:datascroller> mit geänderter Seitenzahl

Mit dem vorgestellten Verfahren über Facetten können die Schaltflächen auch mit alternativen Graphiken oder Texten versehen werden. Dabei wird zwischen aktiven und inaktiven Schaltflächen unterschieden. Schaltflächen werden automatisch inaktiv, wenn sie eine nicht sinnvolle Navigation darstellen, wie z. B. die Schaltfläche für die nächste Seite, wenn bereits die letzte Seite dargestellt ist.

Für die zweite Alternative verwenden wir das RichFaces-Tag <rich:scrollableDataTable>. Diese Tabelle erlaubt unter anderem das spaltenweise Sortieren, die Größenanpassung von Spalten durch Draggen der Spaltenbegrenzungen sowie die Selektion einzelner, aber auch mehrerer Zeilen. Die Verwendung erfolgt ganz analog zur JSF-Datentabelle, wie der folgende Code-Ausschnitt belegt. Die Darstellung im Browser zeigt Abbildung 7.16.

```
<rich:scrollableDataTable value="#{blzHandler.blzs}" var="blz">  
    rows="15" height="300px" width="100%">  
        <rich:column id="blz">  
            <f:facet name="header">Bankleitzahl</f:facet>
```

```
#{blz.blz}
</rich:column>
...
</rich:scrollableDataTable>
```

The screenshot shows a Mozilla Firefox browser window with the title '<rich:scrollableDataTable> - Mozilla Firefox'. The address bar displays 'http://localhost:8080/richfaces-rich/bankleitzahlen-2.jsf'. The main content area contains a scrollable table titled 'Große Datenmengen mit <rich:scrollableDataTable>'. The table has columns: Bankleitzahl, Bezeichnung, PLZ, Ort, and Kurzbezeichnung. The data includes entries for various banks like Bundesbank, Postbank, SEB, and several UniCredit branches. The table is scrollable, with a vertical scrollbar visible on the right side. At the bottom left, there is a status bar with the text 'Fertig'.

Bankleitzahl	Bezeichnung	PLZ	Ort	Kurzbezeichnung
10000000	Bundesbank	10591	Berlin	BBk Berlin
10010010	Postbank	10916	Berlin	Postbank Berlin
10010111	SEB	10789	Berlin	SEB Berlin
10010222	The Royal Bank of Scotland, Niederlassung Deutschland	10105	Berlin	RBS NDL Deutschland
10010424	Aareal Bank	10666	Berlin	Aareal Bank
10019610	Dexia Kommunalbank Deutschland	10969	Berlin	Dexia Berlin
10020000	Berliner Bank -alt-	10890	Berlin	Berliner Bank -alt-
10020200	BHF-BANK	10117	Berlin	BHF-BANK Berlin
10020400	Parex Bank Berlin	10117	Berlin	Parex Bank Berlin
10020500	Bank für Sozialwirtschaft	10178	Berlin	Bank für Sozialwirtschaft
10020890	UniCredit Bank - HypoVereinsbank	10896	Berlin	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	14532	Kleinmachnow	UniCredit Bank-HypoVereinbk
10020890	UniCredit Bank - HypoVereinsbank	16515	Oranienburg	UniCredit Bank-HypoVereinbk

Abbildung 7.16: Scrollbare Tabelle mit <rich:scrollableDataTable>

Wir haben in Abbildung 7.16 bereits eine Anpassung der Spaltenbreiten durch Draggen der Spaltenbegrenzungen vorgenommen. Eine solche Anpassung wird über eine EL-Wertebindung des Attributs `componentState` an den Server übertragen und könnte benutzerdefiniert gespeichert und in neuen Sitzungen wiederverwendet werden, was wir hier allerdings nicht realisieren. Die Sortierung der Tabelle nach Spalteninhalten steht ohne weiteren Aufwand zur Verfügung. Über das Attribut `sortMode` kann mit den Werten `single` und `multi` das Sortierverhalten über eine bzw. mehrere Spalten eingestellt werden. Abbildung 7.17 zeigt die Tabelle nach einem Klick in die Tabellenüberschrift *Bezeichnung*.

Als etwas anspruchsvollere Aufgabe soll nun die Selektion einer Tabellenzeile zu einer server-seitigen Aktion führen. Die Komponente unterstützt die Event-Arten `onSelectionChange`, `onRowClick`, `onRowDb1Click`, `onRowMouseUp` und `onRowMouseDown`. Wir verwenden die einfache Selektion mit `onRowClick` und registrieren mittels `<a4j:support>` einen entsprechenden Listener. Für den Zugriff auf die Selektion und die Tabelle wird das Attribut `selection` sowie eine einfache Komponentenbindung verwendet:

The screenshot shows a Mozilla Firefox browser window with the title '<rich:scrollableDataTable> - Mozilla Firefox'. The address bar displays 'http://localhost:8080/richfaces-rich/bankleitzahlen-2.jsf'. The page content is a table titled 'Große Datenmengen mit <rich:scrollableDataTable>'. The table has columns: Bankleitzahl, Bezeichnung, PLZ, Ort, and Kurzbezeichnung. The rows show various bank entries. A vertical scrollbar is visible on the right side of the table. At the bottom left, there is a button labeled 'Fertig'.

Bankleitzahl	Bezeichnung	PLZ	Ort	Kurzbezeichnung
39060180	Aachener Bank	52001	Aachen	Aachener Bank
39060180	Aachener Bank	52401	Jülich	Aachener Bank
39060180	Aachener Bank	52459	Inden b Jülich	Aachener Bank
39060180	Aachener Bank	52499	Baesweiler	Aachener Bank
39060180	Aachener Bank	52463	Alsdorf, Rheinl	Aachener Bank
39060180	Aachener Bank	52146	Würselen	Aachener Bank
39060180	Aachener Bank (Gf P2)	52001	Aachen	Aachener Bank
39020000	Aachener Bausparkasse	52001	Aachen	Aachener Bauspk Aachen
10010424	Areal Bank	10666	Berlin	Areal Bank
20010424	Areal Bank	20009	Hamburg	Areal Bank
25010424	Areal Bank	30014	Hannover	Areal Bank
36010424	Areal Bank	45009	Essen, Ruhr	Areal Bank Essen
50010424	Areal Bank	60284	Frankfurt am Main	Areal Bank

Abbildung 7.17: Tabelle nach Spalte *Bezeichnung* sortiert

```
<rich:scrollableDataTable value="#{blzHandler.blzs}" var="blz"
    selection="#{blzHandler.selection}"
    binding="#{blzHandler.table}" ...>

    <a4j:support event="onRowClick"
        actionListener="#{blzHandler.rowSelected}" />
    <rich:column id="blz">
        <f:facet name="header">Bankleitzahl</f:facet>
        #{blz.blz}
    </rich:column>
    ...
</rich:scrollableDataTable>
```

Den Code der entsprechenden Managed Bean zeigt die folgende Klassendefinition. Man erkennt die beiden Properties `selection` und `table`. Im allgemeinen Fall muss für die Verarbeitung von client-seitigen Selektionen das RichFaces-Interface `Selection` implementiert werden. RichFaces stellt jedoch bereits eine einfache Implementierung dieses Interfaces in der Klasse `SimpleSelection` bereit, die für unsere Anforderungen ausreicht.

```
@ManagedBean
@SessionScoped
public class BlzHandler implements Serializable {

    private SimpleSelection selection = new SimpleSelection();
    private UIScrollableDataTable table;
```

```
private List<Blz> blzs;

public void rowSelected(ActionEvent ae) {
    // Achtung: klappt nur bei single selection:
    table.setRowKey(selection.getKeys().next());
    Blz blz = (Blz) table.getRowData();
    // hier kann jetzt was mit der BLZ gemacht werden
}

// Getter und Setter
...
}
```

Die Beispiel-Bean verwendet den Session-Scope, um den Code nicht zu sehr aufzublähen. Sinnvoller wäre die Erzeugung der Daten (Property `blzs`) in einer Managed Bean mit Application-Scope und die Reaktion auf Benutzerinteraktionen möglichst in einer Bean mit Request-Scope.

Die Klasse `SimpleSelection` liefert mit der Methode `getKeys()` einen Iterator über die selektierten Tabellenzeilen zurück. Da wir uns auf eine einzelige Selektion beschränken, können wir direkt auf das einzige Element dieses Iterators zugreifen und diese Zeile im Tabellenmodell markieren. Über die Methode `getRowData()` kann dann auf die Daten dieser Zeile zugegriffen werden.

Die ausschließliche Bestimmung des Selektionsindex und das direkte Arbeiten auf der Liste `blzs` führt nicht zum Ziel, da die Darstellung der Tabellenzeilen durch einen etwaigen Sortiervorgang nicht mehr in ihrer ursprünglichen Reihenfolge erfolgen muss.

Wir schließen mit diesem Beispiel unsere Vorstellung von RichFaces-Komponenten ab. Mit Drag and Drop, Baumdarstellungen und der Darstellung großer Datenmengen konnten wir nur einen kleinen Teil der RichFaces-Komponenten oberflächlich ansprechen und fordern den interessierten Leser auf, sich selbst in diese umfangreiche und komplexe Komponentenbibliothek einzuarbeiten. Das RichFaces-Handbuch und die Dokumentation der Komponenten ist über [URL-JRF] erhältlich. Als weiterführende Literatur existieren im Augenblick die Bücher von Katz [Kat08] und Filocamo [Fil09].



Projekt

Das Projekt *richfaces-rich* enthält die Beispielanwendung zu Drag and Drop, die fünf Beispiele der Baum-Komponenten sowie die beiden alternativen Datentabellen.

Kapitel 8

JavaServer Faces im Einsatz: Die Anwendung Online-Banking

In Kapitel 3 haben wir eine sehr einfache JSF-Anwendung zur Verwaltung von Comedians gebaut. Kapitel 4 legte dann die Funktionsweise von JavaServer Faces detailliert dar, während Kapitel 5 die zur Verfügung stehenden Komponenten aufzählte. In den Kapiteln 6 und 7 wurden zwei grundlegende Neuerungen von JSF 2.0 eingeführt, zum einen Facelets mit dem darauf basierenden Template-Mechanismus, zum anderen Ajax. Mit diesem Wissen ist es nun möglich, eine größere, zusammenhängende Anwendung zu entwickeln, die einer „richtigen“ Anwendung nahekommt, zumindest im Bereich der Möglichkeiten eines praxisnahen Lehrbuches.

Als Anwendung wählen wir das Online-Banking einer fiktiven Bank, da viele Leser diesen Anwendungstyp aus eigener Erfahrung kennen werden. Die Anwendung soll die folgenden Anforderungen realisieren:

Authentifizierung

Eine einfache Anmeldung des Benutzers mit Kundennummer und Kennwort. Die erfolgreiche Anmeldung ist Voraussetzung für die weitere Nutzung der Anwendung.

Pflege der Stammdaten

Im Falle eines Umzugs, einer Heirat mit Namensänderung etc. soll der Bankkunde seine Stammdaten selbst aktualisieren können.

Überweisung durchführen

Das eigentliche Bankgeschäft: Überweisung eines bestimmten Betrages von einem eigenen Konto auf ein beliebiges anderes Konto.

Anzeige aller Konten

Alle Konten mit ihren Kontoständen werden angezeigt und dienen zur Selektion eines Kontos für eine Überweisung.

Anzeige der Umsätze

Für ein Konto werden alle Umsätze angezeigt. Die anzuzeigenden Umsätze können bezüglich verschiedener Kriterien eingeschränkt werden.

Export der Umsätze im PDF- und Excel-Format

Die Umsätze werden für den Druck als PDF oder alternativ für die weitere Nutzung auf dem PC als Excel exportiert.

Ziel dieses Kapitels ist es nicht, eine möglichst realistische Online-Banking-Anwendung zu erstellen, vielmehr wird die Anwendung als Mittel zum Zweck genutzt. Der Zweck ist die Demonstration der Einsatzmöglichkeit verschiedener bekannter Konzepte und Modelle im Umfeld von JavaServer Faces. Es kann daher durchaus vorkommen, dass die Realitätsnähe zugunsten einer vorteilhafteren Didaktik etwas leidet. Auch können wir nicht alle Eigenschaften von JSF nutzen, da dies den Umfang der Anwendung zu sehr aufblähen würde. So verzichten wir z. B. auf eine Mehrsprachenfähigkeit, deren Umsetzung wir in Abschnitt 4.7 beschrieben haben.

8.1 Der Seitenaufbau

Bevor wir uns mit der Realisierung der fachlichen Anforderungen beschäftigen, definieren wir den Aufbau der einzelnen JSF-Seiten der Anwendung. Da alle Seiten identisch aufgebaut sein sollen, kommt der Template-Mechanismus von Facelets zum Einsatz. Abbildung 8.1 zeigt im Vorgriff auf Abschnitt 8.4 die Seite zur Pflege der Stammdaten mit einer rot hervorgehobenen Strukturierung der Seite. Die zu erkennenden Teile sind Kopf- (1) und Fußzeile (4) sowie links das Menü (2) und rechts der Hauptteil (3) der Seite. Das Facelets-Template besitzt daher vier Bereiche, die in der Datei `layout.xhtml` in Listing 8.1 zu erkennen sind.



Abbildung 8.1: Der Seitenaufbau der Anwendung

Listing 8.1: Das Banking-Layout layout.xhtml

```
<html>
    <h:head>
        <title>Online-Banking</title>
    </h:head>
    <h:body>
        <ui:insert name="header"/>
        <h:form>
            <div id="menu">
                <ui:include src="menu.xhtml" />
            </div>
            <div id="main">
                <ui:insert name="main"/>
            </div>
        </h:form>
        <div id="footer">
            © JavaServer Faces - Banking
        </div>
    </h:body>
</html>
```

Die Kopfzeile und der Hauptteil einer Seite werden über Template-Clients definiert, so dass diese beiden Bereiche über ein `<ui:insert>` realisiert sind. Die Fußzeile ist bei allen Seiten identisch und relativ klein. Daher wird sie di-

rekt im Template definiert. Das Menü ist ebenfalls bei allen Seiten identisch. Durch den etwas größeren Umfang ist jedoch eine Auslagerung in eine gesonderte Datei mit Hilfe des `<ui:include>`-Tags sinnvoll. Die inkludierte Datei `menu.xhtml` ist in Listing 8.2 dargestellt.

Listing 8.2: Das Menü der Anwendung (Datei `menu.xhtml`)

```
<ui:composition xmlns="..." >
    <ul>
        <li>
            <h:commandLink value="Kontenübersicht" immediate="true"
                action="kontenuebersicht.xhtml?faces-redirect=true" />
        </li>
        <li>
            <h:commandLink value="Umsatzanzeige" immediate="true"
                action="umsatzanzeige.xhtml?faces-redirect=true" />
        </li>
        <li>
            <h:commandLink value="Überweisung" immediate="true"
                action="ueberweisung.xhtml?faces-redirect=true" />
        </li>
        <li>
            <h:commandLink value="Persönliche Daten" immediate="true"
                action="stammdaten.xhtml?faces-redirect=true" />
        </li>
        <li>
            <h:commandLink value="Abmelden" immediate="true"
                action="#{kundenHandler.logout}" />
        </li>
    </ul>
</ui:composition>
```

Alle fünf Menüpunkte sind jeweils über ein `<h:commandLink>`-Tag realisiert. Das in Abbildung 8.1 erkennbare schaltflächenartige Aussehen der Menüpunkte wird über CSS realisiert. Wir gehen hier und in den weiteren Darstellungen der Anwendung nicht auf das verwendete CSS ein und verweisen den interessierten Leser auf das herunterladbare Projekt, das den kompletten Code enthält.

Die ersten vier Menüpunkte verwenden im `action`-Attribut direkt die Anwendungsseiten, die als Template-Client fungieren. Der letzte Menüpunkt realisiert die Abmeldung von der Anwendung über eine Action-Methode, die wir in Abschnitt 8.3.2 vorstellen. Alle Steuerkomponenten verwenden das `immediate`-Attribut, um etwaige Validierungsfehler in den Anwendungsseiten auszuschließen.

8.2 Das Geschäftsmodell

Die Comedian-Anwendung in Kapitel 3 verwendete JPA, um ihre Daten zu persistieren. Auch für das Online-Banking verwenden wir JPA. Da die Daten etwas umfangreicher sind, stellen wir die entsprechenden JPA-Entities an dieser Stelle kurz vor. Abbildung 8.2 zeigt die Entities und ihre Beziehungen. Ein

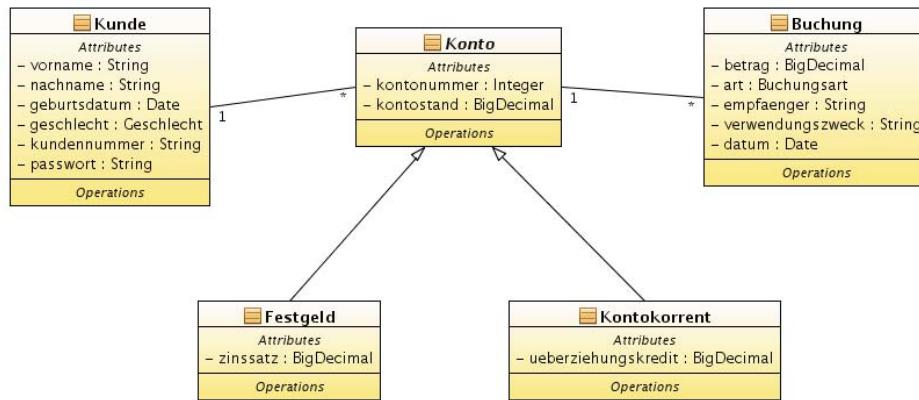


Abbildung 8.2: Das Geschäftsmodell als UML-Klassendiagramm

Kunde kann mehrere Konten besitzen. Die Klasse **Konto** ist abstrakt. Konkrete Ausprägungen sind Festgeld- und Kontokorrentkonten. Für jedes Konto existieren eine Reihe von Buchungen, die die üblichen Attribute enthalten. Die Typen **Geschlecht** in der Klasse **Kunde** und **Buchungsart** in der Klasse **Buchung** sind Aufzählungstypen (**enum**), auf deren Darstellung wir verzichten.

Im Gegensatz zu den Comedians in Kapitel 3 sind Kunden und Konten sowie Konten und Buchungen mit 1:n-Assoziationen verbunden. Diese werden in JPA mit entsprechenden Annotationen modelliert. Listing 8.3 zeigt einen Ausschnitt der Klasse **Kunde**, die durch die `@Entity`-Annotation zu einem JPA-Entity wird. Die 1:n-Assoziation zur Klasse **Konto** wird über die `@OneToMany`-Annotation am Getter `getKonten()` realisiert. Über das `mappedBy`-Attribut wird das die Assoziation realisierende Attribut auf der Kontenseite angegeben. Das `cascade`-Attribut setzt die Kaskadierungseigenschaft von CRUD-Operationen, so dass z. B. beim Persistieren eines neuen Kunden automatisch auch alle Konten des Kundenpersistiert werden.

Listing 8.3: Das JPA-Entity Kunde (unvollständig)

```
@Entity
public class Kunde implements Serializable {

    private Integer id;
    private String vorname;
    private String nachname;
    private Date geburtsdatum;
    private Geschlecht geschlecht;
    private String kundennummer;
    private String passwort;
    private Set<Konto> konten = new HashSet<Konto>();

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }

    ...

    @OneToMany(mappedBy="kunde", cascade=CascadeType.ALL)
    public Set<Konto> getKonten() {
        return konten;
    }
    public void setKonten(Set<Konto> konten) {
        this.konten = konten;
    }

    ...
}
```

Ein Konto wird durch das JPA-Entity Konto realisiert, dessen Quell-Code in Listing 8.4 ausschnittsweise dargestellt ist. Da Konto die Oberklasse einer Vererbungshierarchie darstellt, ist das Entity mit `@Inheritance` annotiert. Als Vererbungsstrategie ist `SINGLE_TABLE` eingestellt. Dies bedeutet, dass die Attribute aller Klassen der Hierarchie in einer Tabelle abgelegt werden.

Listing 8.4: Das JPA-Entity Konto (unvollständig)

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Konto implements Serializable {
```

```
private Integer kontonummer;
private BigDecimal kontostand;
private Kunde kunde;
private Set<Buchung> buchungen = new HashSet<Buchung>();

@Id
public Integer getKontonummer() {
    return kontonummer;
}
public void setKontonummer(Integer kontonummer) {
    this.kontonummer = kontonummer;
}

...
@ManyToOne
@JoinColumn(name="kunde")
public Kunde getKunde() {
    return kunde;
}
public void setKunde(Kunde kunde) {
    this.kunde = kunde;
}
...
}
```

Aus Sicht eines Kontos ist die Beziehung zum Kunden eine n:1-Assoziation. Der Getter `getKunde()` wird daher mit `@ManyToOne` annotiert. Zusätzlich muss die Fremdschlüsselpalte mit der `@JoinColumn`-Annotation angegeben werden. Die Assoziation zwischen `Konto` und `Buchung` entspricht der Assoziation zwischen `Kunde` und `Konto`, so dass sich eine explizite Darstellung erübrigkt. Als letzter offener Punkt im Geschäftsmodell bleiben die beiden nichtabstrakten Kontenarten `Festgeld` und `Kontokorrent`. Das Entity `Festgeld` ist in Listing 8.5 dargestellt. Da es von `Konto` erbt, wird keine weitere JPA-Annotation benötigt.

Listing 8.5: Das JPA-Entity `Festgeld` (unvollständig)

```
@Entity
public class Festgeld extends Konto implements Serializable {

    private BigDecimal zinssatz;
    ...
}
```

Wir schließen die Vorstellung des Geschäftsmodells an dieser Stelle ab. Weitere Darstellungen der Möglichkeiten von JPA erfolgen bei Bedarf in den nächsten Abschnitten. Dem JPA-interessierten Leser legen wir unser eigenes JPA-Buch [MW07] sowie das Buch von Bauer und King als deutsche Übersetzung [BK07a] oder als englische Originalversion [BK07b] nahe.

8.3 Authentifizierung und Autorisierung

In praktisch allen Informationssystemen werden sensible Informationen verarbeitet, so dass je nach Anwendungstyp verschiedene Anforderungen an die Sicherheit des Informationssystems und auch des verwendeten Kommunikationssystems gestellt werden. Grundsätzlich wird im Bereich der Sicherheit Unterschiede zwischen:

- *Authentifizierung (Authentication)*
Die Möglichkeit der Identifizierung der beteiligten Parteien.
- *Autorisierung (Authorization)*
Beschränkung der Zugriffsmöglichkeiten auf bestimmte Ressourcen für bestimmte Benutzer oder Programme.
- *Vertraulichkeit (Confidentiality)*
Sicherstellen, dass ausschließlich die beteiligten Parteien an der Kommunikation teilnehmen.
- *Integrität (Integrity)*
Die Möglichkeit zu verifizieren, dass der Kommunikationsinhalt sich während der Übertragung nicht verändert hat.

In der Regel sind häufig mehrere, meist sogar all diese Punkte in einer Anwendung von Interesse. Wenn Sie z. B. beim Online-Banking Ihre PIN eingeben, wollen Sie sicher sein, dass Ihr Gegenüber tatsächlich Ihre Bank ist, und umgekehrt will die Bank sicher sein, dass tatsächlich Sie der aktuelle Benutzer sind (Authentifizierung). Zudem soll Ihre PIN nicht von anderen abgehört werden können (Vertraulichkeit). Die Bank wiederum möchte nur Ihnen auf Ihr Konto Zugriff gewähren (Autorisierung), und der PDF-Kontoauszug sollte den tatsächlichen Kontostand widerspiegeln und nicht während der Übertragung verfälscht worden sein (Integrität).

Alle vier Themen werden von JSF nicht direkt angesprochen. Java-SE und Java-EE sowie die entsprechenden Übertragungsprotokolle stellen jedoch an verschiedenen Stellen Lösungen für diese Themen bereit. Wir können in diesem Buch nicht auf alle Aspekte eingehen und beschränken uns an dieser Stelle auf

eine sehr einfache Möglichkeit zur Authentifizierung und Autorisierung auf Basis von JSF-Bordmitteln. Im Abschnitt 9.6 stellen wir das Java-EE-Framework JBoss- Seam dar und gehen insbesondere auf die sehr fortgeschrittenen Aspekte der Authentifizierung und Autorisierung mit JBoss- Seam ein.

8.3.1 Realisierung der Authentifizierung

Die zentrale Idee unserer Realisierung ist die Überprüfung der Zugangsdaten, die im Entity **Kunde** in den Attributen **kundennummer** und **passwort** hinterlegt sind. Die Authentifizierung erfolgt also über JPA und damit datenbankbasiert. Bei einer erfolgreichen Authentifizierung wird in der HTTP-Session das Kundenobjekt gespeichert. Das Vorhandensein dieses Objekts wird von einem System-Event zur Autorisierungsprüfung aller Seiten der Anwendung verwendet.

Wir beginnen mit der Eingabe der Authentifizierungsdaten. Der folgende JSF-Code stellt den Hauptteil der Authentifizierungsseite dar.

```
<h:panelGrid columns="2">
    <f:facet name="header">Login Online-Banking</f:facet>
    <h:outputLabel value="Kundennummer:" for="kundennummer" />
    <h:inputText id="kundennummer" required="true"
        value="#{kundenHandler.kundennummer}"
        requiredMessage="Bitte Kundennummer eingeben" />
    <h:outputLabel value="Passwort:" for="passwort" />
    <h:inputSecret id="passwort" required="true"
        value="#{kundenHandler.passwort}"
        requiredMessage="Bitte Passwort eingeben" />
    <h:panelGroup />
    <h:commandButton action="#{kundenHandler.login}"
        value="Anmelden" />
</h:panelGrid>
```

Man erkennt ein **<h:inputText>**- und **<h:inputSecret>**-Tag, deren Werte an die Properties **kundennummer** und **passwort** der Managed Bean **kundenHandler** gebunden sind. Die Steuerkomponente ist an die Action-Methode **login()** der Managed Bean gebunden. Listing 8.6 zeigt die entsprechenden Code-Stellen der Klasse **KundenHandler**.

Listing 8.6: Die Managed Bean KundenHandler

```
1  @ManagedBean
2  @SessionScoped
3  public class KundenHandler implements Serializable {
4
5      private String kundennummer;
6      private String passwort;
7      private Kunde kunde;
8
9      @PersistenceContext
10     private EntityManager em;
11
12    public String login() {
13        Query query = em.createQuery("select k from Kunde k \
14                                where k.kundennummer = :kundennummer \
15                                and k.passwort = :passwort");
16        query.setParameter("kundennummer", kundennummer);
17        query.setParameter("passwort", passwort);
18        List<Kunde> kunden = query.getResultList();
19        if (kunden.size() == 1) {
20            kunde = kunden.get(0);
21            return "/kontenuebersicht.xhtml?faces-redirect=true";
22        } else {
23            return null;
24        }
25    }
26    ...
27 }
```

In den Zeilen 16 und 17 werden die Benutzereingaben innerhalb der JPA-Query verwendet. Zeile 19 prüft, ob diese Eingaben valide sind. Wenn ja, wird zur Kontenübersicht navigiert, die wir in Abschnitt 8.6 vorstellen.

Da die Managed Bean einen Session-Scope hat, enthält das Property `kunde` nach erfolgreicher Authentifizierung den Kunden als aktuelle JPA-Entity-Instanze. Dies wird zur Autorisierung des Zugriffs auf alle Seiten der Anwendung verwendet. Dazu wird das Facelets-Template aus Listing 8.1 auf Seite 287 überarbeitet. Der folgende Code-Ausschnitt zeigt den mit dem `<f:event>`-Tag eingefügten Event-Listener, der eine sehr einfache Form der Authentifizierung realisiert.

```
<h:body>
  <ui:insert name="header"/>
  <f:event listener="#{kundenHandler.checkLoggedIn}" 
           type="preRenderView" />
  <h:form>
    <div id="menu">
```

```
<ui:include src="menu.xhtml" />
</div>
...

```

Durch die Verwendung im Template werden alle Seiten der Anwendung mit diesem System-Event versehen. Der Event-Listener überprüft *vor* dem Rendern einer Seite, ob das **kunde**-Property gesetzt ist, und navigiert bei negativem Ausgang auf die Login-Seite:

```
public void checkLoggedIn(ComponentSystemEvent cse) {
    FacesContext context = FacesContext.getCurrentInstance();
    if (kunde == null) {
        context.getApplication().getNavigationHandler()
            .handleNavigation(context, null,
                "/login.xhtml?faces-redirect=true");
    }
}
```

Das vorgestellte Verfahren zur Authentifizierung und Autorisierung ist für unsere Zwecke ausreichend. Wie bereits erwähnt, ist ein so einfaches Verfahren für reale Anwendungen in der Regel jedoch nicht ausreichend, so dass nach Alternativen gesucht werden muss. In Abschnitt 9.6 stellen wir mit JBoss Seam eine solche Alternative vor.

8.3.2 Die Abmeldung

Die Abmeldung von der Anwendung gestaltet sich recht einfach. Da ein angemeldeter Kunde durch das **kunde**-Property des Kunden-Handlers repräsentiert wird, muss dem Property lediglich **null** zugewiesen werden. Die in Listing 8.2 auf Seite 288 im letzten **<h:commandLink>** für die Abmeldung verwendete Action-Methode **logout()** lässt sich daher als einfache Zuweisung realisieren:

```
public String logout() {
    kunde = null;
    return "/login.xhtml?faces-redirect=true";
}
```

Da der Kunden-Handler Session-Scope hat, bleibt der Kunden-Handler weiter in der Session bestehen. Eine bessere Alternative der Abmeldung ist die komplette Löschung der Session. Hierzu wird die Methode **invalidateSession()** des **ExternalContext** aufgerufen. Das Überschreiben des Property **kunde** mit **null** erübrigkt sich damit:

```
public String logout() {
    FacesContext.getCurrentInstance()
        .getExternalContext().invalidateSession();
    return "/login.xhtml?faces-redirect=true";
}
```

8.4 Pflege der Stammdaten

Als erste Seite unserer Anwendung realisieren wir die Pflege der Stammdaten. Listing 8.7 zeigt die zu erstellende Seite `stammdaten.xhtml`. Man erkennt in den Zeilen 3 und 9 die `<ui:define>`-Tags, mit deren Hilfe die Kopfzeile (Bereich 1 in Abbildung 8.1 auf Seite 287) und der Hauptteil der Seite (Bereich 2 in Abbildung 8.1) erzeugt werden. Alle weiteren Seiten werden ebenfalls diese Struktur besitzen.

Listing 8.7: Die Verwaltung der Stammdaten (`stammdaten.xhtml`)

```
1 <ui:composition xmlns="..." >
2
3     <ui:define name="header">
4         <div id="header">
5             Persönliche Daten
6         </div>
7     </ui:define>
8
9     <ui:define name="main">
10        <h:panelGrid columns="2" columnClasses="sdl,sdr" >
11            <f:facet name="header">Ihre persönlichen Daten</f:facet>
12
13            <h:outputLabel value="Anrede:" for="anrede" />
14            <h:selectOneMenu id="anrede"
15                value="#{kundenHandler.kunde.geschlecht}">
16                <f:selectItem itemLabel="Herr" itemValue="MAENNlich" />
17                <f:selectItem itemLabel="Frau" itemValue="WEIBLICH" />
18            </h:selectOneMenu>
19
20            <h:outputLabel value="Vorname:" for="vorname" />
21            <h:panelGroup>
22                <h:message id="mvorname" for="vorname" />
23                <h:inputText id="vorname" required="true"
24                    value="#{kundenHandler.kunde.vorname}"
25                    requiredMessage="Bitte Vorname eingeben" />
26            </h:panelGroup>
27
28            <h:outputLabel value="Nachname:" for="nachname" />
29            <h:panelGroup>
30                <h:message id="mnachname" for="nachname" />
31                <h:inputText id="nachname" required="true"
32                    value="#{kundenHandler.kunde.nachname}"
33                    requiredMessage="Bitte Nachname eingeben" />
34            </h:panelGroup>
35
36            <h:outputLabel value="Geburtsdatum:" for="gebu" />
```

```
37      <h:panelGroup>
38          <h:message id="mgebu" for="gebu" />
39          <h:inputText id="gebu" required="true"
40              value="#{kundenHandler.kunde.geburtsdatum}"
41              requiredMessage="Bitte Geburtsdatum eingeben">
42                  <f:convertDateTime pattern="dd.MM.yyyy"/>
43          </h:inputText>
44      </h:panelGroup>
45
46      <h:panelGroup />
47          <h:commandButton value="Speichern"
48              action="#{kundenHandler.aktualisieren}" />
49      </h:panelGrid>
50
51  </ui:define>
52 </ui:composition>
```

Die Seite verwendet eine zweispaltige Darstellung für die Labels und Eingaben. Die Darstellung im Browser ist in Abbildung 8.3 dargestellt. Um die zweispaltige Struktur trotz der `<h:message>`-Tags gewährleisten zu können, sind diese mit den Eingaben in eine Panel-Group eingeschlossen. Abbildung 8.4 zeigt die Seite im Fehlerfall.



Abbildung 8.3: Die Stammdatenseite

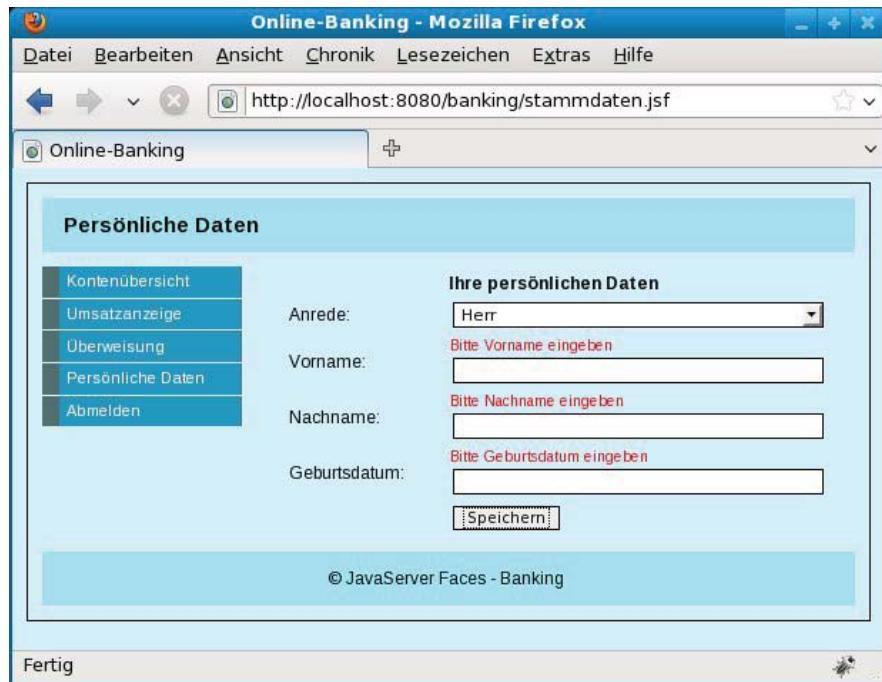


Abbildung 8.4: Die Stammdatenseite mit Fehlern

Eine interessante Optimierungsmöglichkeit ist die Aktualisierung der Stammdaten über Ajax. Dazu wird der Code für den `<h:commandButton>` erweitert zu:

```
<h:commandButton value="Speichern"
    action="#{kundenHandler.aktualisieren}"
    <f:ajax execute="@form" render="@form" />
</h:commandButton>
```

Dabei werden alle UI-Eingabekomponenten des Formulars auf dem Server und nach Erhalt der Antwort alle UI-Komponenten auf dem Client aktualisiert. Hierbei ist insbesondere die Aktualisierung z. B. des Menüs oder der Labels auf dem Client aus Optimierungsgründen nicht sinnvoll. Abhilfe schafft hier die explizite Nennung der zu aktualisierenden Komponenten. Eine Analyse ergibt, dass die vom Benutzer eingegebenen Daten nicht aktualisiert werden müssen, sehr wohl aber die eventuell existierenden Fehlermeldungen. Es ergibt sich also für das `<f:ajax>`-Tag die folgende Konfiguration:

```
<h:commandButton value="Speichern"
    action="#{kundenHandler.aktualisieren}"
    <f:ajax execute="anrede vorname nachname gebu"
    render="mvorname mnachname mgebu" />
```

```
</h:commandButton>
```

Zum Abschluss der Erläuterung der Stammdatenseite fehlt noch die Action-Methode `aktualisieren()`. Diese ist unter Verwendung von JPA-Funktionalität realisiert. Da der Persistenzkontext im Default-Fall ein transaktionaler und kein erweiterter Persistenzkontext ist, werden alle Entities des Persistenzkontextes am Ende der Transaktion vom Persistenzkontext gelöst (detached). Damit im nächsten HTTP-Request das existierende Entity wieder an einer neuen Transaktion teilnehmen kann, muss es wieder in den Persistenzkontext eingebunden werden. Dies geschieht mit der `merge()`-Methode. Da JPA ein sogenanntes *Automatic Dirty Checking* realisiert, also Entities bei Bedarf ohne weiteres Zutun (z.B. Aufruf einer `update()`-Methode) am Ende einer Transaktion in der Datenbank aktualisiert, wird kein weiterer Methodenaufruf benötigt. Der folgende Code-Ausschnitt macht dies noch einmal deutlich.

```
@PersistenceContext  
private EntityManager em;  
  
@Resource  
private UserTransaction utx;  
  
public String aktualisieren() {  
    try {  
        utx.begin();  
        kunde = em.merge(kunde);  
        utx.commit();  
    } catch (Exception e) {  
        // Fehlermeldung fuer Benutzer und Log  
    }  
    return null;  
}
```

8.5 Überweisungen

Bei der Implementierung einer Überweisung soll die Möglichkeit der Auswahl eines Kontos sowie zur Eingabe der üblicherweise benötigten Daten – wie Empfänger, Bankverbindung, Verwendungszweck und Betrag – bestehen. Bei der Bankverbindung beschränken wir uns auf die Kontonummer und verzichten auf die Bankleitzahl. Wir realisieren jedoch nur die Aktualisierung des eigenen Kontos, verwenden also selbst die Kontonummer des Empfängers nicht weiter.

Das Layout der Seite erfolgt analog zur Stammdatenseite zweispaltig, so dass wir uns bei der Darstellung auf die in dieser Seite neu verwendeten Eigenschaf-

ten von JSF beschränken, um Wiederholungen zu vermeiden. Listing 8.8 zeigt daher nur die entsprechenden Ausschnitte des Hauptteils der Seite.

Listing 8.8: Die Überweisungsseite (ueberweisung.xhtml)

```
1 <ui:define name="main">
2   <f:validateRequired>
3     <h:panelGrid columns="2" ...>
4       <f:facet name="header">
5         Bitte Überweisungsdaten eingeben
6       </f:facet>
7
8       <h:outputLabel value="Konto (bitte auswählen)" for="konto" />
9
10      <h:selectOneMenu id="konto"
11        value="#{ueberweisungsHandler.konto}">
12        <f:selectItems value="#{kundenHandler.konten}" />
13      </h:selectOneMenu>
14      ...
15      <h:outputLabel value="Betrag (Euro)" for="betrag" />
16      <h:panelGroup>
17        <h:message for="betrag" styleClass="message" />
18        <h:inputText id="betrag" label="Betrag"
19          value="#{ueberweisungsHandler.betrag}">
20          <f:convertNumber locale="DE" />
21        </h:inputText>
22      </h:panelGroup>
23      ...
24      <h:panelGroup />
25      <h:commandButton value="Überweisung ausführen"
26        action="#{ueberweisungsHandler.ueberweisen}" />
27    </h:panelGrid>
28  </f:validateRequired>
29 </ui:define>
```

Die Auswahl des für die Überweisung zu verwendenden Kontos erfolgt über ein Drop-Down-Menü. Dies ist in den Zeilen 10–13 mit einem `<h:selectOneMenu>` realisiert. Bei den anzuzeigenden Alternativen machen die zu verwendenden JPA-Entities Probleme. Ein zu erstellender Konvertierer hat losgelöste Entities, eventuell zu überschreibende `equals()`- und `hashCode()`-Methoden etc. zu beachten. Wir lösen das Problem durch die Verwendung der Kontonummer als einfachen String und umgehen damit die Erstellung eines Konvertierers. Dem interessierten Leser sei auch hier der Blick in JBoss-Seam empfohlen. Seam enthält mit `<s:convertEntity>` ein Tag, das in allen Auswahlkomponenten verwendet werden kann und JPA-Entities automatisch konvertiert. Zurück

zum Programm-Code: Das <f:selectItems>-Tag in Zeile 12 ruft die Methode `getKonten()` auf, die im Folgenden dargestellt ist.

```
public List<String> getKonten() {  
    List<String> konten = new ArrayList<String>();  
    for (Konto konto : kunde.getKonten()) {  
        String classname = konto.getClass().getCanonicalName();  
        konten.add(konto.getKontonummer() + " (" +  
            classname.substring(classname.lastIndexOf('.') + 1)  
            + ")");  
    }  
    return konten;  
}
```

Man erkennt, dass wir nicht nur die Kontonummer, sondern auch die Kontoart (Festgeld oder Kontokorrent) in Klammern angeben. Abbildung 8.5 zeigt dies mit einem Kontokorrentkonto.



Abbildung 8.5: Die Überweisungsseite

Das <h:inputText>-Tag in den Zeilen 18–21 enthält zwei erwähnenswerte Merkmale. Zum einen die Verwendung des Attributs `label`, zum anderen eine deutsche Lokalisierung. Mit dem `label`-Attribut kann ein lokalisierte Name für ein Eingabefeld vergeben werden, der dann in Fehlernachrichten verwen-

det wird. Im Beispiel haben wir lediglich ein String-Literal verwendet, die Lokalisierung mit den in Abschnitt 4.7 vorgestellten Möglichkeiten ist jedoch leicht möglich. Die Lokalisierung des Zahlenformats für den Überweisungsbetrag wird über das `<f:convertNumber>` mit `locale="DE"` vorgenommen und sorgt dafür, dass das Komma statt des Punktes als Dezimaltrenner verwendet wird.

Als letzter Teil der Überweisung definieren wir die Aktion-Methode `ueberweisen()`. Der folgende Code-Ausschnitt aus der Klasse `ÜberweisungsHandler` zeigt die Properties, die über Wertebindungen durch den Benutzer eingegeben werden, und die Methode `ueberweisen()`. Bei dieser muss zunächst die Kontonummer aus der Darstellung des Auswahlmenüs extrahiert werden, um sie danach als Primärschlüssel der JPA-Methode `find()` verwenden zu können. Da wir den String zur Darstellung des Kontos selbst zusammengebaut haben, können wir für die Extraktion der Kontonummer auf die eventuell nicht ganz saubere Suche nach einem Leerzeichen zurückführen.

```
@ManagedBean
public class UeberweisungsHandler {

    private String empfaenger;
    private String verwendungszweck;
    private BigDecimal betrag;
    private String konto; // Syntax: <Nummer> (<Art>)
    private String kontoEmpfaenger; // nicht verwendet

    public String ueberweisen() {
        String kontonummer = konto.substring(0,konto.indexOf(' '));
        try {
            utx.begin();
            Konto konto = em.find(Konto.class,
                                   new Integer(kontonummer));
            Buchung buchung = new Buchung(betrag, Buchungsart.SOLL,
                                           empfaenger, verwendungszweck, konto);
            utx.commit();
            return "weitere-ueberweisung.xhtml?faces-redirect=true";
        } catch (Exception e) {
            // Fehlerbehandlung
            return null;
        }
    }
}
```

Der Konstruktor `Buchung()` ruft für das Konto, das als letzter Parameter übergeben wird, die Methode `addBuchung()` der Klasse `Konto` auf. In dieser erfolgt das Anlegen der bidirektionalen Beziehung zwischen Konto und Buchung sowie die Aktualisierung des Kontostands:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Konto implements Serializable {
    ...
    public void addBuchung(Buchung buchung) {
        buchung.setKonto(this);
        buchungen.add(buchung);
        if (buchung.getArt().equals(Buchungsart.HABEN)) {
            kontostand = kontostand.add(buchung.getBetrag());
        } else {
            kontostand = kontostand.subtract(buchung.getBetrag());
        }
    }
    ...
}
```

8.6 Anzeige aller Konten

Die Anzeige aller Konten eines Kunden ist relativ einfach zu realisieren. Um diese Anforderung etwas anspruchsvoller zu gestalten, soll jedes angezeigte Konto die Möglichkeit besitzen, für dieses Konto zur Überweisungsseite zu navigieren, wobei das selektierte Konto als Überweisungsquelle vorausgewählt ist. Abbildung 8.6 auf der nächsten Seite zeigt alle Konten des Herrn Mustermann mit der Möglichkeit, über eine Schaltfläche eines der Konten für die Überweisungsseite auszuwählen.

Der Quell-Code für die Kontenübersichtsseite ist in Listing 8.9 dargestellt.

Listing 8.9: Die Kontenübersicht (`kontenuebersicht.xhtml`)

```
1 <ui:define name="main">
2     <div>
3         Konten für #{kontenuebersichtHandler.kunde.vorname}
4         #{kontenuebersichtHandler.kunde.nachname}
5     </div>
6     <h:dataTable var="konto"
7         value="#{kontenuebersichtHandler.konten}"
8         style="width: 100%;" rowClasses="row1, row2"
9         columnClasses="colRechtsbuendig, colLinksbuendig, ... ">
10    <h:column>
11        <h:outputText value="#{konto.kontonummer}">
12            <f:convertNumber pattern="0000000000" />
13        </h:outputText>
14    </h:column>
15    <h:column> #{konto.kontoart} </h:column>
16    <h:column>
```

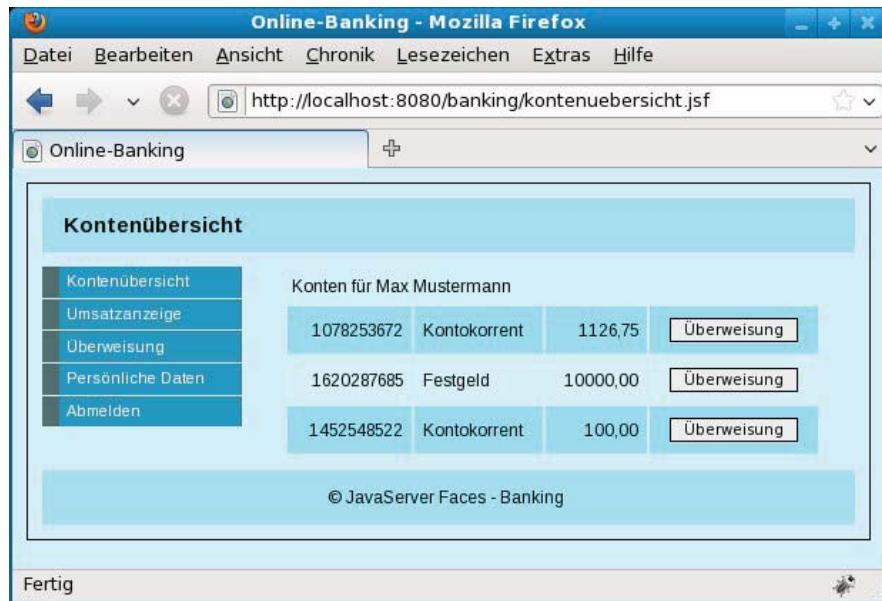


Abbildung 8.6: Kontenübersicht

```
17     <h:outputText value="#{konto.kontostand}">
18         <f:convertNumber locale="DE" pattern="#####.00" />
19     </h:outputText>
20 </h:column>
21 <h:column>
22     <h:commandButton value="Überweisung"
23         action="ueberweisung.xhtml?faces-redirect=true">
24         <f:param name="kontonummer"
25             value="#{konto.kontonummer}" />
26     </h:commandButton>
27 </h:column>
28 </h:dataTable>
29 </ui:define>
```

Interessant ist die direkte Navigation zur Überweisungsseite (Zeile 23) und die Verwendung des `<f:param>`-Tags (Zeilen 24/25) innerhalb der Steuerkomponente. Die als Request-Parameter übergebene Kontonummer kann nun in einem System-Event verwendet werden, um das Drop-Down-Menü der Überweisungsseite vorzubelegen. Dazu erweitern wir die Überweisungsseite (Listing 8.8) mit einem `<f:event>`-Tag und registrieren den Listener auf das Pre-Render-View-Event:

```
<ui:define name="main">
<f:event listener="#{ueberweisungsHandler.auswahl}">
```

```
        type="preRenderView" />
<f:validateRequired>
<h:panelGrid columns="2" ...>
    ...

```

Die Event-Listener-Methode `auswahl()` ist im Folgenden dargestellt. Zunächst wird über den Faces-Context eine Expression-Factory erzeugt, die den EL-Ausdruck `#{{param['kontonummer']}} auswertet und somit die als Request-Parameter übergebene Kontonummer erhält.`

```
public void auswahl(ComponentSystemEvent cse) {
    ELContext context = FacesContext.getCurrentInstance()
        .getELContext();
    Application application = FacesContext.getCurrentInstance()
        .getApplication();
    ExpressionFactory exFactory =
        application.getExpressionFactory();
    ValueExpression ve = exFactory.createValueExpression(context,
        "#{{param['kontonummer']}}", Integer.class);
    String kontonummer = ve.getValue(context).toString();
    List<String> konten = kundenHandler.getKonten();
    for (String kto : konten) {
        if (kto.startsWith(kontonummer)) {
            konto = kto; // Drop-Down setzen
        }
    }
}
```

Zum Abschluss der Diskussion der Kontenübersichtsseite steht noch die Darstellung der Klasse `KontenuebersichtHandler` aus. Diese erfolgt im Listing 8.10.

Listing 8.10: Die Managed Bean `KontenuebersichtHandler`

```
1  @ManagedBean
2  @SessionScoped
3  public class KontenuebersichtHandler implements Serializable {
4
5      @ManagedProperty(value = "#{kundenHandler.kunde}")
6      private Kunde kunde;
7
8      private DataModel<Konto> konten = new ListDataModel<Konto>();
9
10     public KontenuebersichtHandler() { }
11
12     public DataModel<Konto> getKonten() {
13         konten.setWrappedData(
14             new ArrayList<Konto>(kunde.getKonten())));

```

```
15     return konten;
16 }
17 ...
18 }
```

Basis der Kontenübersicht bildet die Klasse `DataModel` als Wrapper um die Kontenliste. Der Getter `getKonten()` liefert eine Instanz dieser Klasse, die über die gleichnamige Methode der Entity-Klasse `Kunde` die entsprechende Kontenliste beinhaltet. Obwohl die Methode `Kunde.getKonten()` (letzte Methode in Listing 8.3 auf Seite 290) einen normalen Getter darstellt, erfolgt beim Methodenaufruf durch den JPA-Provider eine entsprechende `Select`-Anfrage an die Datenbank.

Ebenfalls interessant ist der Zugriff auf den aktuellen Kunden. Dieser wird über `@ManagedProperty` in den Kontenübersichts-Handler injiziert.

Dem aufmerksamen Leser wird aufgefallen sein, dass wir in der Comedian-Anwendung im Kapitel 3 einen ähnlichen Anwendungsfall auf ganz andere Art gelöst haben. Dort wurden alle Comedians ebenfalls in ein `DataModel` eingebettet und über eine Schaltfläche einer der Comedians ausgewählt. Dies könnte hier auch realisiert werden, etwa über eine Action-Methode `ueberweisung()`, die statt der direkten Navigation im `action`-Attribut des `<h:commandButton>` verwendet wird:

```
public String ueberweisung() {
    Konto selektiertesKonto = konten.getRowData();
    return "ueberweisung.xhtml";
}
```

Im Gegensatz zur Comedian-Anwendung haben wir im Online-Banking jedoch zwei Managed Beans für die beiden Seiten, so dass der obige Code nur exemplarisch für die Selektion des Kontos steht. Er müsste entsprechend erweitert werden, um das selektierte Konto dem Überweisungs-Handler zu übergeben. Wir überlassen dies dem Leser zur Übung.

8.7 Anzeige der Umsätze

Bei der Anzeige der Umsätze haben wir zu Beginn des Kapitels als Anforderung die Einschränkung der Umsätze bezüglich verschiedener Kriterien formuliert. Als konkrete Kriterien definieren wir nun

- das Konto selbst
- Zeitraum (von und/oder bis)
- Soll- und/oder Habenumsätze
- Betrag (größer und/oder kleiner)
- Verwendungszweck (als Teil-String)

Abbildung 8.7 zeigt die Umsatzseite im Browser. Außer der Kontoauswahl wurden keine weiteren Suchkriterien angegeben, so dass alle Buchungen für das Konto angezeigt werden. Die Schaltfläche **Aktualisieren** ist für die Anzeige der Buchungsdaten zuständig. Da der Quell-Code der Seite zur Eingabe der Suchkritieren keine weiteren JSF-Elemente enthält, verzichten wir auf eine Darstellung.

Die in der Schaltfläche **Aktualisieren** gebundene Action-Methode `aktualisieren()` initialisiert die DataModel-Variable `buchungen`. Sie greift hierfür auf die Methode `getBuchungenIntern()` zurück, die auch für die Erzeugung der PDF- und Excel-Dokumente verwendet wird und deshalb ausgelagert wurde.

```
@ManagedBean
public class UmsaetzeHandler {

    private DataModel<Buchung> buchungen =
        new ListDataModel<Buchung>();
    ...

    public String aktualisieren() {
        buchungen.setWrappedData(getBuchungenIntern());
        return null;
    }
    ...
}
```

Die Methode `getBuchungenIntern()` ist relativ umfangreich, da die Konstruktion der JPA-Query-Strings abhängig von der Verwendung der verschiedenen Eingabemöglichkeiten durch den Benutzer ist. Wir beschränken uns daher exemplarisch auf die Properties `von`, `bis` und `verwendungszweck`:

```
private List<Buchung> getBuchungenIntern() {

    StringBuilder queryString = new StringBuilder(
        "select b from Buchung b where b.konto = :konto ");
    ...
    if (von != null) {
        queryString.append("and b.datum >= :von ");
    }
    if (bis != null) {
        queryString.append("and b.datum <= :bis ");
    }
}
```

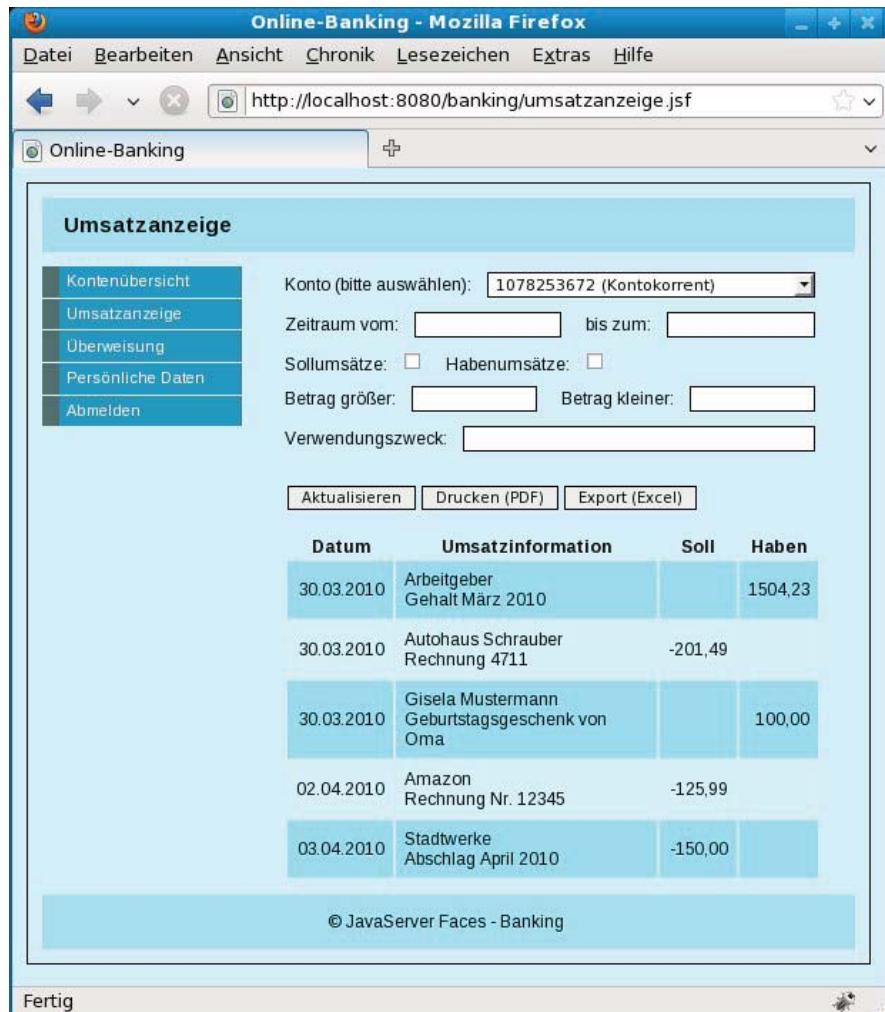


Abbildung 8.7: Anzeige der Umsätze

```
}

if (! verwendungszweck.trim().equals("")) {
    queryString.append(
        "and b.verwendungszweck like :verwendungszweck");
}

Query query = em.createQuery(queryString.toString());
Konto konto = em.find(Konto.class,
    new Integer(this.konto.substring(0,
        this.konto.indexOf(' '))));
query.setParameter("konto", konto);
...

if (von != null) {
```

```
        query.setParameter("von", von, TemporalType.DATE);
    }
    if (bis != null) {
        query.setParameter("bis", bis, TemporalType.DATE);
    }
    if (! verwendungszweck.trim().equals("")) {
        query.setParameter("verwendungszweck",
                           "%" + verwendungszweck.trim() + "%");
    }
    ...
    return query.getResultList();
}
```

Zu bemerken ist hier, dass das mit JPA 2.0 eingeführte Criteria API oder alternativ Hibernates Criteria API die Konstruktion der Anfrage wesentlich vereinfachen würde.

Ebenfalls angemerkt werden muss hier die Möglichkeit, per Konfiguration leere Strings als `null` zu behandeln. Dies realisiert, wie in Abschnitt 4.8 beschrieben, ein Kontextparameter in der `web.xml`:

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

Die obige Kontextparameterdefinition führt dann im Ausdruck `verwendungs-zweck.trim()` zu einer Null-Pointer-Exception.

Ein häufiges Muster bei der Anzeige von a priori unbekannt vielen Daten mit einer Datentabelle (`<h:dataTable>`) ist die Verwendung des `rendered`-Attributs mit einem Test auf Existenz der Daten. Im Falle der Buchungen kann es sein, dass keine Buchung für die verwendeten Selektionskriterien existiert. In einem solchen Fall würden die Spaltenüberschriften der Tabelle (Datum, Umsatzinformation, Soll, Haben) dargestellt werden, jedoch keine Buchungen. Um diesen unschönen Zustand zu vermeiden, wird im folgenden Code geprüft, ob die Liste der Buchungen (`getWrappedData()`) leer ist, und nur bei vorhandenen Buchungen wird die Tabelle inklusive der Spaltenüberschriften gerendert.

```
<h:dataTable var="buchung" value="#{umsaetzeHandler.buchungen}"
             rendered="#{not empty umsaetzeHandler.buchungen.wrappedData}"
             rowClasses="row1, row2"
             columnClasses="colRechtsbuendig, colLinksbuendig, ... ">
    <h:column>
        <f:facet name="header">Datum</f:facet>
```

```
<h:outputText value="#{buchung.datum}">
    <f:convertDateTime pattern="dd.MM.yyyy" />
</h:outputText>
</h:column>
<h:column>
    <f:facet name="header">Umsatzinformation</f:facet>
    #{buchung.empfaenger}<br/>#{buchung.verwendungszweck}
</h:column>
<h:column>
    <f:facet name="header">Soll</f:facet>
    <h:outputText value="#{-buchung.betrag}"
        rendered="#{buchung.soll}">
        <f:convertNumber locale="DE" pattern="#####.00" />
    </h:outputText>
</h:column>
<h:column>
    <f:facet name="header">Haben</f:facet>
    <h:outputText value="#{buchung.betrag}"
        rendered="#{buchung.haben}">
        <f:convertNumber locale="DE" pattern="#####.00" />
    </h:outputText>
</h:column>
</h: dataTable>
```

Für die Darstellung der Buchungsbeträge unterscheiden wir in zwei Spalten zwischen Soll- und Habenbuchungen. Da Buchungsbeträge in der Entity-Klasse **Buchung** durch das vorzeichenlose Property **betrag** und das Enum-Property **art** mit den Werten **SOLL** und **HABEN** repräsentiert werden, definieren wir in der Klasse **Buchung** noch die beiden Methoden **isSoll()** und **isHaben()**:

```
public boolean isSoll() {
    return art.equals(Buchungsart.SOLL);
}

public boolean isHaben() {
    return art.equals(Buchungsart.HABEN);
}
```

Die zweispaltige Darstellung der Buchungsbeträge inklusive des negativen Vorzeichens kann dann wie oben dargestellt erfolgen.

8.8 Export der Umsätze im PDF- und Excel-Format

Um den Kunden den Ausdruck der Umsätze und deren Weiterverarbeitung auf dem heimischen PC zu ermöglichen, soll die Anwendung die Liste der Umsätze

im PDF- und Excel-Format exportieren können. Mittlerweile existieren eine ganze Reihe von Java-Implementierungen für diese Aufgaben. Wir verwenden hier die bekannte und weit verbreitete PDF-Bibliothek *iText* [URL-ITEXT] von Bruno Lowagie und empfehlen für das weitere Literaturstudium das Buch zum Thema vom selben Autor [Low07]. Als Excel-Bibliothek verwenden wir das Java Excel API [URL-JXL].

Wir beginnen mit der Erzeugung und Auslieferung des PDF-Dokuments. Dabei konzentrieren wir uns auf die Auslieferung, da diese JSF-spezifisch ist, und erläutern die Erzeugung nur kurz. Listing 8.11 zeigt die Action-Methode `exportPdf()`, die über die Schaltfläche Drucken (PDF) in Abbildung 8.7 auf Seite 308 aufgerufen wird.

Listing 8.11: Methode `exportPdf()`, Teil 1

```
1 public void exportPdf() {
2
3     ByteArrayOutputStream baos = new ByteArrayOutputStream();
4     PdfPCell cell;
5     Document document = new Document();
6     PdfWriter.getInstance(document, baos);
7     Font defaultFont = new Font(Font.HELVETICA, 10);
8     document.open();
9     document.add(new Paragraph("Umsatzanzeige für "
10                     + kunde.getVorname() + " " + kunde.getNachname()
11                     + ", erstellt am " + format.format(new Date())
12                     + " für Konto " + konto, defaultFont));
13
14     PdfPTable table = new PdfPTable(new float[]{1f, 4f, 1f, 1f});
15     table.setWidthPercentage(100f);
16
17     // Spaltenüberschriften ausgelassen
18
19     List<Buchung> buchungen = getBuchungenIntern();
20     for (Buchung buchung : buchungen) {
21         cell = new PdfPCell(
22             new Phrase(format.format(buchung.getDatum()),
23                         defaultFont));
24         cell.setHorizontalAlignment(Element.ALIGN_RIGHT);
25         table.addCell(cell);
26         cell = new PdfPCell(
27             new Phrase(buchung.getEmpfaenger() + ", "
28                         + buchung.getVerwendungszweck(),
29                         defaultFont));
30         table.addCell(cell);
31         cell = new PdfPCell(
32             new Phrase(buchung.isSoll() ?
```

```
33             " - " + buchung.getBetrag().toString()
34             : " ", defaultFont));
35         cell.setHorizontalAlignment(Element.ALIGN_RIGHT);
36         table.addCell(cell);
37         cell = new PdfPCell(
38             new Phrase(buchung.isHaben() ?
39                 buchung.getBetrag().toString() : " ",
40                 defaultFont));
41         cell.setHorizontalAlignment(Element.ALIGN_RIGHT);
42         table.addCell(cell);
43     }
44     document.add(table);
45     document.close();
```

Ein PDF-Dokument wird in iText durch eine Instanz der Klasse `Document` repräsentiert. Dies wird in Zeile 5 erzeugt und in Zeile 6 mit einem `ByteArrayOutputStream` verbunden. Letzterer wird später verwendet, um die HTTP-Response zu erzeugen. In Zeile 14 wird eine Tabelle angelegt, deren Inhalt in Zeile 20 und folgende durch eine Iteration über die gefundenen Umsätze ganz analog zur `<h:dataTable>`-Komponente erzeugt wird.

Der JSF-spezifische Teil der Methode `exportPdf()` ist in Listing 8.12 dargestellt. Zentrale Idee hierbei ist der Zugriff auf den externen Kontext des JSF-Laufzeitsystems. Dies kann entweder ein Servlet- oder ein Portal-Kontext sein, wobei wir uns hier auf den Ersteren beschränken. Über diesen wird die Antwort ausgegeben und die letzte Phase des JSF-Lebenszyklus (Renderen der Antwort) übersprungen.

Listing 8.12: Methode `exportPdf()`, Teil 2

```
47     FacesContext context = FacesContext.getCurrentInstance();
48     HttpServletResponse response =
49         (HttpServletResponse) context.getExternalContext()
50             .getResponse();
51     response.setContentType("application/pdf");
52     response.setContentLength(baos.size());
53     response.setHeader("Content-disposition",
54         "inline;filename=\"umsaetze.pdf\"");
55     OutputStream out = response.getOutputStream();
56     baos.writeTo(out);
57     out.flush();
58     response.flushBuffer();
59     context.responseComplete();
```

In den Zeilen 48–50 wird über den `ExternalContext` auf das Antwortobjekt des Servlet-Requests (`HttpServletResponse`) zugegriffen. Für dieses Objekt wird zunächst der Content-Type gesetzt (`setContentType()` in Zeile 51). Der Content-Type stellt den Datentyp einer HTTP-Antwort dar und wird durch einen *MIME-Typ* (*Multipurpose Internet Mail Extension*) definiert. Im Beispiel ist der MIME-Typ `application/pdf`, das binäre PDF-Datenformat. Die generelle Struktur eines MIME-Typs wird in den RFCs 1341, 1521 und 1522 festgelegt. Diese und andere RFCs finden Sie im Internet [URL-RFC]. Falls Sie andere Datenformate als HTTP-Antwort zurückgeben wollen, finden Sie eine Liste der aktuell verwendeten MIME-Typen ebenfalls im Internet unter [URL-MIME]. Im Folgenden wird die Größe der Antwort und der HTTP-Header gesetzt. Schließlich werden die eigentlichen Daten geschrieben sowie – aus Sicht der JSF-Implementierung – die Beantwortung der Anfrage beendet und somit die letzte der sechs Bearbeitungsphasen nicht durchgeführt (`responseComplete()` in Zeile 59). Abbildung 8.8 zeigt den oberen Teil des erzeugten PDF-Dokuments.

Umsatzanzeige für Max Mustermann, erstellt am 19.05.2010 für Konto 1078253672 (Kontokorrent)			
Datum	Umsatzinformation	Soll (Euro)	Haben (Euro)
30.03.2010	Arbeitgeber, Gehalt März 2010		1504.23
30.03.2010	Autohaus Schrauber, Rechnung 4711	- 201.49	
30.03.2010	Gisela Mustermann, Geburtstagsgeschenk von Oma		100.00
02.04.2010	Amazon, Rechnung Nr. 12345	- 125.99	
03.04.2010	Stadtwerke, Abschlag April 2010	- 150.00	

Abbildung 8.8: Umsätze als PDF-Dokument

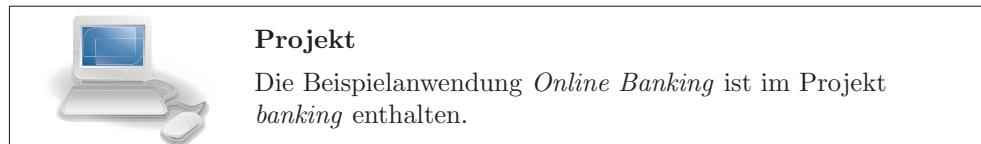
Die Erzeugung der Excel-Datei geschieht völlig analog zur Erzeugung der PDF-Datei. Listing 8.13 zeigt die Action-Methode `exportExcel()`, die über die Schaltfläche **Export (Excel)** aufgerufen wird.

Listing 8.13: Methode `exportExcel()`

```
1 public void exportExcel() {  
2     FacesContext context = FacesContext.getCurrentInstance();  
3     HttpServletResponse response =  
4         (HttpServletResponse) context.getExternalContext()
```

```
5             .getResponse();
6     response.setContentType("application/vnd.ms-excel");
7     response.setHeader("Content-Disposition",
8                         "attachment; filename=\"umsaetze.xls\";");
9     OutputStream out = response.getOutputStream();
10
11    WorkbookSettings ws = new WorkbookSettings();
12    ws.setLocale(new Locale("de", "DE"));
13    WritableWorkbook workbook = Workbook.createWorkbook(out, ws);
14    WritableSheet sheet = workbook.createSheet("Sheet1", 0);
15    sheet.addCell(new Label(0, 0, "Datum"));
16    sheet.addCell(new Label(1, 0, "Umssatzinformation"));
17    sheet.addCell(new Label(2, 0, "Soll"));
18    sheet.addCell(new Label(3, 0, "Haben"));
19    int row = 1;
20    List<Buchung> buchungen = getBuchungenIntern();
21    for (Buchung buchung : buchungen) {
22        sheet.addCell(new Label(0, row,
23                               format.format(buchung.getDatum())));
24        sheet.addCell(new Label(1, row, buchung.getEmpfaenger()
25                        + ", " + buchung.getVerwendungszweck()));
26        sheet.addCell(new Label(2, row,
27                       buchung.isSoll() ? "-" + buchung.getBetrag() : ""));
28        sheet.addCell(new Label(3, row,
29                       buchung.isHaben() ? buchung.getBetrag().toString() : ""));
30        row++;
31    }
32    workbook.write();
33    workbook.close();
34    response.flushBuffer();
35    context.responseComplete();
```

Im Unterschied zur Methode `exportPdf()` ist die Reihenfolge der Anweisungen eine etwas andere. Der Grund hierfür liegt in der Methode `createWorkbook()` in Zeile 13, die als ersten Parameter einen `OutputStream` benötigt, der zuvor erzeugt werden muss. Als weiterer Unterschied liegt im Mime-Type `application/vnd.ms-excel`. Die JSF-spezifischen Anweisungen folgen ansonsten dem Schema, das wir auch in `exportPdf()` verwendet haben. Die für das Java Excel API spezifischen Methoden sind selbsterklärend. Das Resultat der Excel-Erzeugung ist in Abbildung 8.9 in der OpenOffice-Anwendung Calc dargestellt.



The screenshot shows a Microsoft Excel-like spreadsheet titled "umsaetze.xls" in OpenOffice.org Calc. The data is organized into columns A through E:

	A	B	C	D	E
1	Datum	Umsatzinformation	Soll	Haben	
2	30.03.2010	Arbeitgeber, Gehalt März 2010		1504.23	
3	30.03.2010	Autohaus Schrauber, Rechnung 4711	- 201.49		
4	30.03.2010	Gisela Mustermann, Geburtstagsgeschenk von Oma		100.00	
5	02.04.2010	Amazon, Rechnung Nr. 12345	- 125.99		
6	03.04.2010	Stadtwerke, Abschlag April 2010	- 150.00		
7					
8					
9					

Abbildung 8.9: Umsätze im Excel-Format in OpenOffice

Aufgabe 8.1

Setzen Sie den Kontextparameter `INTERPRET_EMPTY_STRING_ SUBMITTED_VALUES_AS_NULL` auf `true`, und testen Sie die Anwendung, insbesondere die Umsatzanzeige. Beseitigen Sie den Fehler.

Aufgabe 8.2

Bei der Erzeugung der Excel-Datei wurden die Umsätze als `Label` und somit als Text erzeugt. Ein Sortieren der Spalten in Excel sortiert daher alphabetisch und nicht numerisch. Überarbeiten Sie den Export, so dass die Beträge numerisch sortierbar sind.

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Kapitel 9

JavaServer Faces und Java-EE

JavaServer Faces sind als Bestandteil der Java Enterprise Edition für die Entwicklung von Unternehmensanwendungen vorgesehen. Es ist daher für ein JSF-Buch sehr sinnvoll, die Verknüpfungspunkte und Integrationsmöglichkeiten von JSF und Java-EE vorzustellen. Durch die Komplexität von Java-EE ist eine vollständige Vorstellung im Rahmen dieses Buches jedoch nicht möglich. Dem interessierten Leser empfehlen wir die Bücher von Panda, Rahman und Lane [PRL07], von Eberling und Leßner [EL08] sowie von Backschat und Rücker [BR08].

In den ersten drei Abschnitten dieses Kapitels stellen wir JavaServer Faces im Rahmen von Java-EE 5, Java-EE 6 und CDI vor. Um über eine solide Basis für den Vergleich zu verfügen, formulieren wir als Ziel die Überarbeitung der in Kapitel 3 entwickelten Anwendung zur Verwaltung von Comedians. Dabei sollen die beiden JSF-Seiten der Anwendung unverändert übernommen werden. Es sind dies zum einen die Seite `anzeige-comedians.xhtml` zur Anzeige aller Comedians, dargestellt in Listing 3.1 auf Seite 22, und zum anderen die Seite `comedian.xhtml` zur Bearbeitung der Daten eines Comedians, dargestellt in Listing 3.3 auf Seite 27.

Im Ergebnis erhalten wir vier alternative Implementierungen einer einfachen CRUD-artigen Anwendung, die dieselben JSF-Seiten, aber durchaus recht unterschiedliche Java-Realisierungen enthalten. Dass die Verwendung derselben JSF-Seiten in diesen vier Alternativen möglich ist, macht den modularen und komponentenbasierten Aufbau von Java-EE im Allgemeinen und JavaServer Faces im Speziellen noch einmal deutlich.

In einem weiteren Abschnitt reißen wir JSF-angrenzende Themen innerhalb von Java-EE kurz an. Wir beschließen das Kapitel mit einem Einblick in die

Sicherheitsmechanismen von JBoss Seam, einem Framework, das innovative Konzepte in Java-EE 5 integriert und die in CDI definierten Konzepte maßgeblich beeinflusst hat.

9.1 Java-EE 5

Java-EE 5 wurde im Mai 2006 im JSR-244 definiert [URL-JSR244]. Die Spezifikation umfasst als Dach-Spezifikation weitere 24 Spezifikationen bzw. APIs, unter anderem JSF 1.2, Servlet 2.5, EJB 3.0 und JPA 1.0. Servlets benötigen als Ablaufumgebung einen Web-Container, EJBs einen EJB-Container. Application-Server wie etwa die beiden populären, quelloffenen Implementierungen *Glassfish* und *JBoss-AS* stellen diese beiden Container-Arten bereit und integrieren die anderen APIs. Die strukturellen Zusammenhänge sind in Abbildung 9.1 schematisch dargestellt.

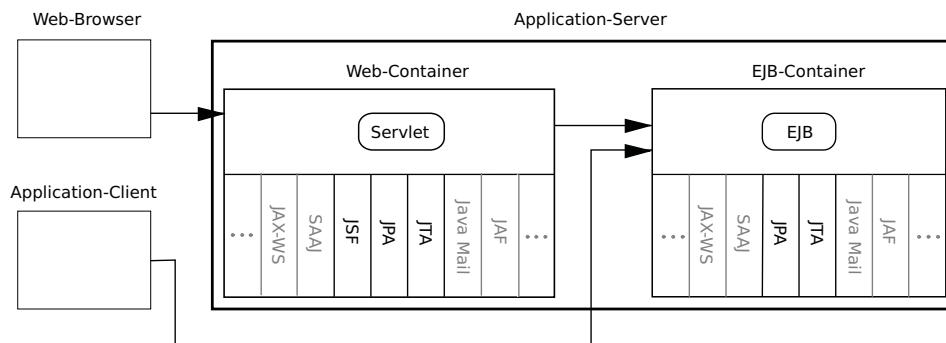


Abbildung 9.1: Application-Server und Container

Der Web-Container stellt eine Ablaufumgebung für Servlets dar. Da Java-Server Faces über ein Servlet realisiert sind, ist dies der Container, in dem unsere bisherigen JSF-Anwendungen betrieben wurden. Sowohl die Comedian-Anwendung als auch das Online-Banking machten neben JSF auch von den APIs JPA (Java Persistence API) und JTA (Java Transaction API) Gebrauch.

Der EJB-Container realisiert eine Ablaufumgebung für Enterprise JavaBeans. Durch den Container sowie die Interaktion von Container und EJBs wird die Erstellung von verteilten, transaktionalen, sicheren und portablen Anwendungen ermöglicht. Diese erwünschten Eigenschaften müssen dabei nicht vom Entwickler explizit programmiert werden, sondern werden als Querschnittsfunktionen (cross cutting concerns) vom Container automatisch bereitgestellt.

Dem Aspekt der Verteilung wird Java-EE unter anderem dadurch gerecht, dass bei sehr großen Anwendungen vom Regelfall eines Application-Servers auf einer JVM abgewichen werden kann. Es können Cluster von Application-Servern inklusive eines Load-Balancings erstellt, jedoch auch einzelne Web- oder EJB-Container auf jeweils einer JVM betrieben werden. Dies benötigt eine Kommunikation zwischen Web- und EJB-Container, aber auch zwischen mehreren EJB-Containern. Die Kommunikation erfolgt in der Regel über Remote Method Invocation (RMI) oder äquivalente Verfahren.

Durch die Möglichkeit des Deployments von Web- und EJB-Anwendung auf verschiedenen JVMs und Rechnern definiert Java-EE 5 jeweils ein Deployment-Archiv für Web- als auch für EJB-Anwendungen. Wir gehen an dieser Stelle auf die beiden Archivarten nur so weit ein, wie es die Entwicklung und das Deployment unserer Comedian-Anwendung erfordert, und verweisen den interessierten Leser für weitere Informationen auf die zu Beginn des Kapitels genannte weiterführende Literatur.

Das Deployment einer Java-EE-Anwendung erfolgt in Form einer sogenannten *Enterprise-Archive-Datei* (EAR). Dies ist eine gewöhnliche JAR-Datei mit der Dateiendung „.ear“. Sie enthält die Web-Anwendung als *Web-Archive-Datei* (WAR) mit der Endung „.war“ und die EJB-Anwendung als EJB-Jar mit der Endung „.jar“. Diese beiden Anwendungen müssen als eigenständige Module definiert werden, was in der Datei `application.xml` im META-INF-Verzeichnis erfolgt. Wird die Anwendung als `comedians.ear` erstellt, so enthält diese Datei also mindestens die folgenden drei Dateien:

```
comedians.war  
comedians.jar  
META-INF/application.xml
```

Bei der Realisierung der Anwendung mit Eclipse erstellen wir die Projekte

- `comedians-ee5` als *Enterprise Application Project*
- `comedians-ee5-web` als *Dynamic Web Project*
- `comedians-ee5-ejb` als *EJB Project*

wobei die Projektarten die Eclipse-spezifischen Bezeichnungen darstellen. Da die Projektnamen standardmäßig die Archivnamen bestimmen, muss das obige Namensschema überarbeitet werden. Als Anwendungsarchiv ergibt sich der Name `comedians-ee5.ear` mit den folgenden Dateien:

```
comedians-ee5-web.war  
comedians-ee5-ejb.jar  
META-INF/application.xml
```

Nach der Darstellung der technischen Infrastruktur können wir uns nun der Entwicklung der Anwendung widmen. Wir beginnen mit der Web-Anwendung. Da die JSF-Seite unverändert aus Kapitel 3 übernommen wird, bleibt hier nur die Managed Bean. Sie ist in Listing 9.1 angegeben.

Listing 9.1: Die Managed Bean ComedianJsfMb

```
1  @ManagedBean(name = "comedianHandler")
2  @SessionScoped
3  public class ComedianJsfMb implements Serializable {
4
5      private DataModel<Comedian> comedians;
6      private Comedian aktuellerComedian;
7
8      @EJB
9      private ComedianEjb comedianEjb;
10
11     public ComedianJsfMb() {
12         comedians = new ListDataModel<Comedian>();
13         aktuellerComedian = new Comedian();
14     }
15
16     public String speichern() {
17         comedianEjb.speichern(aktuellerComedian);
18         return "anzeige-comedians?faces-redirect=true";
19     }
20
21     public String aendern() {
22         aktuellerComedian = comedians.getRowData();
23         return "comedian?faces-redirect=true";
24     }
25
26     public String neu anlage() {
27         aktuellerComedian = new Comedian();
28         return "comedian?faces-redirect=true";
29     }
30
31     public String loeschen() {
32         comedianEjb.loeschen(comedians.getRowData());
33         return null;
34     }
35
36     public DataModel<Comedian> getComedians() {
37         comedians.setWrappedData(comedianEjb.getComedians());
38         return comedians;
39     }
40     ...
41 }
```

Da wir neben der JSF Managed Bean und dem Entity auch eine Session Bean realisieren, auf die die Namensendung *Handler* ebenfalls gut passen würde, ändern wir unser Namensschema und machen im Namen die Art der Bean deutlich. Da die JSF-Seite unverändert übernommen wird, muss der EL-Name geändert werden (Zeile 1). Im Vergleich zur Managed Bean in Listing 3.2 auf Seite 25 fehlen der injizierte Entity-Manager sowie die injizierte Transaktion. Stattdessen wird mit @EJB eine Enterprise JavaBean injiziert. Diese wird in den Zeilen 17, 32 und 37 verwendet, um die Bearbeitung der Geschäftslogik von der JSF Managed Bean zur EJB weiterzureichen. Der weitere Code ist selbsterklärend.

Sowohl die EJB als auch das Entity, das unverändert aus Kapitel 3 übernommen wird, sind in der EJB-Anwendung enthalten. Da EJBs sowohl lokal als auch entfernt (remote), d. h. in einer anderen JVM verwendet werden können, müssen wir für diese beiden Verwendungsarten gesonderte Interfaces erstellen. Ein lokales Interface wird mit @Local, ein entferntes Interface mit @Remote annotiert. Da wir die EJB im selben Container verwenden, nutzen wir das lokale Interface:

```
@Local  
public interface ComedianEjb {  
    public void speichern(Comedian comedian);  
    public void loeschen(Comedian comedian);  
    public List<Comedian> getComedians();  
}
```

Die Implementierung des Interfaces erfolgt in der Session Bean *ComedianEjbImpl*, die in Listing 9.2 dargestellt ist. Bei Session Beans wird zwischen zustandslosen (stateless) und zustandsbehafteten (stateful) Beans unterschieden. Die erste Art hält keinen für den Client sichtbaren Zustand, die zweite hält einen Zustand und wird an die HTTP-Session gebunden. Man spricht häufig auch von SLSB (Stateless Session Beans) und von SFSB (Stateful Session Beans). Im Beispiel ist die Session Bean zustandsbehaftet und wird damit derselben Session zugeordnet wie die JSF Managed Bean.

Listing 9.2: Die Stateful Session Bean *ComedianEjbImpl*

```
1  @Stateful  
2  public class ComedianEjbImpl implements ComedianEjb,  
3                                              Serializable {  
4      private List<Comedian> comedians;  
5  
6      @PersistenceContext(type = PersistenceContextType.EXTENDED)  
7      private EntityManager em;
```

```
8
9     public void speichern(Comedian comedian) {
10        em.persist(comedian);
11        rereadComedians();
12    }
13
14    public void loeschen(Comedian comedian) {
15        em.remove(comedian);
16        rereadComedians();
17    }
18
19    public void rereadComedians() {
20        comedians = em.createNamedQuery("SelectComedians")
21                           .getResultList();
22    }
23
24 }
```

In der Managed Bean wurde die Session Bean über das Interface definiert:

```
@EJB
private ComedianEjb comedianEjb;
```

Existiert nur eine Implementierung des Interfaces, wie in unserem Fall, wird eine Instanz dieser Implementierung injiziert. Falls es mehrere Implementierungen gibt, muss über das **beanName**-Attribut der **@EJB**-Annotation eine Implementierung bestimmt werden.

Ein interessantes Detail der Implementierung ist die Verwendung des erweiterten Persistenzkontextes (**EXTENDED** in Zeile 6). Der transaktionale Persistenzkontext löst (detached) alle vom Entity-Manager verwalteten Entities am Ende einer Transaktion. Beim nächsten Request muss ein geändertes Entity daher dem Entity-Manager über die **merge()**-Methode wieder hinzugefügt werden. Beim erweiterten Persistenzkontext werden die Entities nicht losgelöst, so dass auch kein **merge()**-Aufruf benötigt wird. Da der erweiterte Persistenzkontext sich die Entities über einzelne Requests merken muss, ist seine Verwendung nur bei Stateful Session Beans erlaubt.

In der Methode **speichern()** wird sowohl ein neuer Comedian angelegt als auch ein bestehender Comedian aktualisiert. Da beim Anlegen eines neuen Comedians, aber auch beim Löschen eines bestehenden Comedians die Liste aller Comedians zu aktualisieren ist, wird am Ende der Methoden **speichern()** und **loeschen()** jeweils die Methode **rereadComedians()** aufgerufen, die diese Aktualisierung über die aus Abschnitt 3.3 bekannte **NamedQuery** realisiert.

Projekte



Die Comedians-Anwendung besteht aus drei Projekten:

- *comedians-ee5*
- *comedians-ee5-web*
- *comedians-ee5-ejb*

wobei das Projekt *comedians-ee5* die beiden anderen umfasst und das zu deployende Projekt darstellt.

9.2 Java-EE 6

Java-EE 6 wurde im Dezember 2009 als JSR-316 [URL-JSR316] veröffentlicht, nachdem der zunächst eingereichte JSR-313 [URL-JSR313] wieder zurückgezogen wurde. Java-EE 6 enthält unter anderem die Spezifikationen JSF 2.0, Servlet 3.0, EJB 3.1 und JPA 2.0. Die mit Java-EE 5 begonnene Vereinfachung der Entwicklung von Unternehmensanwendungen wurde mit Java-EE 6 fortgeführt. Die für unsere Comedian-Anwendung in Betracht kommenden Vereinfachungen sind die

- Möglichkeit zur Paketierung von EJBs in das Web-Archiv und der
- Verzicht auf die Angabe eines lokalen Interfaces.

Beim ersten Punkt muss lediglich beachtet werden, dass die EJBs und andere Ressourcen in das Verzeichnis `/META-INF/classes` zu packen sind. Dies ist jedoch der Standardfall und erfolgt bei gängigen IDEs automatisch. Beim zweiten Punkt wird bei einer EJB ohne Interface davon ausgegangen, dass alle öffentlichen Methoden der Bean das lokale Interface definieren. Man bezeichnet dies als die *No-Interface View* der Bean.

Für die Comedian-Anwendung bleiben die JSF-Seiten und die JSF Managed Bean unverändert erhalten. Das lokale Interface `ComedianEjb` entfällt. Damit das Injizieren der EJB in die Managed Bean ohne Code-Änderung erfolgen kann, wird die Session Bean `ComedianEjbImpl` in `ComedianEjb` umbenannt. Damit ist die Comedian-Anwendung vollständig auf Java-EE 6 umgestellt und wurde dabei deutlich vereinfacht.



Projekt

Das Projekt *comedians-ee6* enthält den Code für die Comedian-Verwaltung mit Java-EE 6.

9.3 CDI und Weld

Der von JBoss eingereichte JSR-299 identifizierte unter anderem die schlechte Integration von JavaServer Faces und EJB-Komponenten als unnötige Einschränkung im Entwicklungsmodell von Java-EE 5. Der Request basierte aus JBoss' Erfahrungen mit dem Framework *JBoss Seam* [URL-SEAM], das die angesprochenen Einschränkungen obsolet machte. Der ursprüngliche JSR-Na-*me Web Beans* wurde in *Contexts and Dependency Injection for Java* geändert. Nach der Etablierung des JSR-330 mit Namen *Dependency Injection for Java* [URL-JSR330] wurde der JSR-Name zur Vermeidung der offensichtlichen Verwechslungsgefahr ein weiteres Mal geändert, und zwar in *Contexts and Dependency Injection for the Java EE Platform* [URL-JSR299], und wird allgemein mit *CDI* abgekürzt. Die Referenzimplementierung von CDI wird von JBoss unter dem Namen *Weld* [URL-WELD] entwickelt und ist im Glassfish v3 enthalten.

Die Überarbeitung der Comedian-Anwendung aus Kapitel 3 in den Abschnitten 9.1 und 9.2 haben vor allem eine Verlagerung von Code aus der JSF Managed Bean in die Session Bean nach sich gezogen. Da CDI die Verwendung von Session Beans als JSF Managed Beans und insofern auch die Verwendung in EL-Ausdrücken erlaubt, basiert unsere im Folgenden dargestellte Überarbeitung auf der Originalversion aus Kapitel 3, da diese nur wenig geändert werden muss. Listing 9.3 zeigt die überarbeitete Version.

Listing 9.3: Die CDI Bean ComedianHandler

```
1  @Named
2  @SessionScoped
3  @Stateful
4  public class ComedianHandler implements Serializable {
5
6      @PersistenceContext(type = PersistenceContextType.EXTENDED)
7      private EntityManager em;
8
9      private DataModel<Comedian> comedians;
10     private Comedian aktuellerComedian;
11
12    public ComedianHandler() {
13        comedians = new ListDataModel<Comedian>();
14        aktuellerComedian = new Comedian();
15    }
16
17    public String speichern() {
18        em.persist(aktuellerComedian);
```

```
19     rereadComedians();
20     return "anzeige-comedians?faces-redirect=true";
21 }
22
23 public String aendern() {
24     aktuellerComedian = comedians.getRowData();
25     return "comedian?faces-redirect=true";
26 }
27
28 public String neuanolage() {
29     aktuellerComedian = new Comedian();
30     return "comedian?faces-redirect=true";
31 }
32
33 public String loeschen() {
34     aktuellerComedian = comedians.getRowData();
35     em.remove(aktuellerComedian);
36     rereadComedians();
37     return null;
38 }
39
40 private void rereadComedians() {
41     comedians.setWrappedData(
42         em.createNamedQuery("SelectComedians").getResultList());
43 }
44
45 ...
46 }
```

Um die Unterschiede deutlich zu machen, geben wir noch einmal die ersten Zeilen der Definition der JSF Managed Bean (Listing 3.2) aus Kapitel 3 wieder, diesmal jedoch mit voll qualifizierter Schreibweise der Annotationen:

```
@javax.faces.bean.ManagedBean
@javax.faces.bean.SessionScoped
public class ComedianHandler implements Serializable {
```

Für den direkten Vergleich qualifizieren wir auch die Annotationen aus Listing 9.3:

```
@javax.inject.Named
@javax.enterprise.context.SessionScoped
@javax.ejb.Stateful
public class ComedianHandler implements Serializable {
```

Man erkennt, dass die bei der Managed Bean verwendeten Annotationen JSF-Annotationen aus dem Package `javax.faces.bean` sind. Demgegenüber ist keine der Annotationen der CDI-Bean eine JSF-Annotation. Mit der Annota-

tion `@Named` wird ein Name für eine Bean vergeben. Dieser ist als Wert des `value`-Attributs anzugeben. Falls das Attribut nicht angegeben ist, wird der Klassenname mit kleingeschriebenem Anfangsbuchstaben als Default-Name verwendet. Die `@Named`-Annotation ist im JSR-330 definiert. Diese bildet eine Basisspezifikation für Dependency Injection und wird zur Zeit von Weld, Spring und Google Guice unterstützt. Durch diese Namensdefinition kann die Bean in EL-Ausdrücken der JSF-Seiten verwendet werden, so dass die Anforderung unveränderter JSF-Seiten erfüllt ist.

Die Annotation `@Stateful` ist die schon bekannte EJB-Annotation zur Definition einer zustandsbehafteten Session Bean. Die Annotation `@SessionScoped` definiert schließlich den Session-Kontext als Gültigkeitsbereich der Bean. Das Package `javax.enterprise.context` wird in der CDI-Spezifikation definiert.

Die Deklaration des Entity-Managers in den Zeilen 6/7 entspricht der JSF Managed Bean. Statt des transaktionalen Entity-Managers verwenden wir jedoch den erweiterten Entity-Manager. Dies ist nur in Stateless Session Beans möglich und erspart uns im weiteren Code den Aufruf der `merge()`-Methode. Da Methodenaufrufe von SFSB im Default-Fall innerhalb von Transaktionen erfolgen, kann auf die Benutzertransaktionen aus Listing 3.2 verzichtet werden. Die weiteren Methoden der Bean sind selbsterklärend und in dieser oder ähnlicher Form bereits in den bisherigen Implementierungen verwendet worden.

Zum Abschluss der Diskussion der CDI-Bean kann noch eine Laufzeitoptimierung erfolgen. Da wir eine SFSB verwenden, erfolgen alle Methodenaufrufe innerhalb von Transaktionen. Für die im Quell-Code der Bean nicht angegebenen Getter und Setter gilt dies ebenfalls. Da Getter im JSF-Bearbeitungszyklus häufiger aufgerufen werden, ist ein Aufruf außerhalb einer Transaktion sinnvoll, falls keine Änderungen in der Datenbank erfolgen. Dies gilt in unserem Beispiel auch für den Setter. Mit der Annotation `@TransactionAttribute` lässt sich dies einfach erreichen:

```
@TransactionAttribute(TransactionAttributeType.NEVER)
public DataModel<Comedian> getComedians() {
    return comedians;
}
@TransactionAttribute(TransactionAttributeType.NEVER)
public Comedian getAktuellerComedian() {
    return aktuellerComedian;
}
@TransactionAttribute(TransactionAttributeType.NEVER)
public void setAktuellerComedian(Comedian aktuellerComedian) {
    this.aktuellerComedian = aktuellerComedian;
}
```



Projekt

Das Projekt *comedians-cdi* enthält den Code für die Comedian-Verwaltung mit CDI/Weld.

Bemerkung: Mit CDI ist es möglich, die häufig künstlich erscheinende Trennung zwischen JSF und EJBs oder, allgemeiner formuliert, zwischen GUI und Anwendungslogik zu umgehen und zu vermeiden. Aus dieser Möglichkeit entsteht jedoch kein Zwang. Wenn Sie eine Architektur verwenden, die diverse Schichten und Schnittstellen definiert, so ist dies ganz offensichtlich auch mit CDI zu realisieren. Sie sind der Architekt und werden nicht durch APIs in bestimmte Architekturzwänge gedrängt.

Die vorgestellte Anwendung auf Basis von JSF und CDI verwendet keine JSF-eigenen Annotationen für Managed Beans. Es gibt bereits erste Stimmen im Internet und auf Konferenzen, die dies propagieren: Da CDI einen allgemeineren Ansatz zur Definition und Verwendung von Managed Beans verfolgt, sollten die CDI-Annotationen verwendet und auf JSF-Annotationen verzichtet werden. Alle bisher entwickelten Beispiele im Buch könnten ohne `@ManagedBean` auskommen und alternativ `@Named` verwenden. Dasselbe gilt für die Annotationen zur Definition des Scopes einer Bean. Wir können dem Leser zur Zeit noch nicht raten, so vorzugehen, und empfehlen, die weitere Entwicklung abzuwarten.

9.4 Konversationen mit CDI

Das Servlet-API definiert die Kontexte Request, Session und Application. JavaServer Faces haben dies so übernommen. In der Version 2.0 kamen die View und ein Custom-Scope als Kontexte hinzu. Diese beiden sind jedoch nicht direkt als Zeitspanne definiert, wie die ursprünglichen Kontexte und daher im Zusammenhang mit Konversationen nicht relevant. Ein Kritikpunkt der drei Kontexte war die Lücke zwischen Request- und Session-Kontext. Häufig bedingen Anwendungsfälle eine Folge von Interaktionen, die logisch zusammenhängen und nur in ihrer kompletten Durchführung zum Ziel führen. Ein Request ist damit zu kurz, mehrere Requests hängen nicht zusammen, und die Session ist zu lang, um dies adäquat zu realisieren. In der Vergangenheit mussten daher derartige Anwendungsfälle explizit programmiert werden.

Eine *Konversation* weist ähnliche Charakteristika auf wie eine Session, da sie einen Zustand besitzt und sich über mehrere Requests erstreckt. Im Gegensatz zu einer Session wird der Beginn und das Ende einer Konversation aber explizit

durch die Anwendung definiert, und Konversationen in verschiedenen Tabs einer Browser-Sitzung haben eine eigene Identität, können also unterschieden werden. Abbildung 9.2 macht diesen Zusammenhang deutlich.

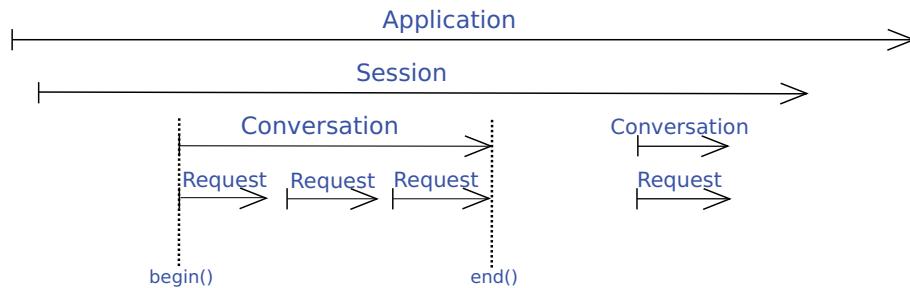


Abbildung 9.2: Kontexte

Die CDI-Spezifikation führt das Konzept eines Konversationskontextes ein und definiert im Package `javax.enterprise.context` neben den Annotationen `@RequestScoped`, `@SessionScoped` und `@ApplicationScoped` auch die Annotation `@ConversationScoped`, die als Kontext einer Bean die Konversation definiert.

Einem JSF-Request ist bei der Verwendung von CDI immer ein Konversationskontext zugeordnet, der sich im Default-Fall im Zustand *transient* befindet. Am Ende der Request-Bearbeitung werden eine solche transiente Konversation und alle in diesem Kontext befindlichen Objekte gelöscht. Anders sieht es bei einer *langlaufenden* Konversation aus. Die Konversation und alle assoziierten Objekte überleben das Ende der Request-Bearbeitung und sind beim nächsten Request derselben Konversation wieder verfügbar.

Der Übergang einer transienten in eine lang laufende Konversation erfolgt durch den Aufruf der Methode `begin()` des Interfaces `Conversation`. Der umgekehrte Übergang einer lang laufenden in eine transiente Konversation erfolgt durch den Aufruf der Methode `end()` desselben Interfaces. Diese programmatiche Kontextabgrenzung ist in Abbildung 9.2 durch die beiden Methoden angedeutet. Die Propagierung einer Konversation über Request-Grenzen hinweg erfolgt in der Regel automatisch, kann aber auch explizit programmatisch definiert werden. Wir gehen auf die explizite Propagierung nicht ein und verweisen den Leser auf die Spezifikation oder das der Referenzimplementierung [URL-WELD] beiliegende Weld-Handbuch.

Als einführendes Beispiel für die Verwendung von Konversationen realisieren wir die wizard-ähnliche Anlage eines Neukunden. Während in der Praxis mehrere Daten einzugeben und Überprüfungen vorzunehmen sind, besteht unser Wizard lediglich aus der Eingabe von Vornamen, Nachnamen und Geburtstag des Kunden. Obwohl sicher auch in einer JSF-Seite zu realisieren, verwenden wir drei Seiten, um den Einsatz einer Konversation zu demonstrieren. Die Abbildungen 9.3 und 9.4 zeigen den Wizard beispielhaft für den Fall einer erfolgreichen Anlage eines Kunden.

Da die JSF-Seiten relativ simple Strukturen aufweisen, verzichten wir auf eine Darstellung der Seiten und konzentrieren uns auf die Managed Bean. Um sämtliche Navigationsstrukturen an einer Stelle zu konzentrieren, enthalten die Action-Attribute der Schaltflächen keine expliziten Navigationsangaben sondern rufen die entsprechenden Methoden der Bean auf. Die Namensgebung der Methoden folgt dabei den einzugebenden Daten mit Ausnahme der Methoden, die die Konversationsgrenzen definieren. Listing 9.4 zeigt den Code der Managed Bean `KundenHandler`. Man erkennt an den Annotationen (Zeilen 1 und 2), dass es sich nicht um eine JSF Managed Bean, sondern um eine CDI Managed Bean mit Konversationslebensdauer handelt.

Listing 9.4: Die Bean `KundenHandler` mit Konversationskontext

```
1  @Named
2  @ConversationScoped
3  public class KundenHandler implements Serializable {
4
5      @Inject
6      private Conversation conversation;
7
8      private String vorname;
9      private String nachname;
10     private Date geburtstag;
11
12    public String neuAnlage() {
13        conversation.begin();
14        return "vorname?faces-redirect=true";
15    }
16
17    public String vorname() {
18        return "vorname?faces-redirect=true";
19    }
20
21    public String nachname() {
22        return "nachname?faces-redirect=true";
23    }
```

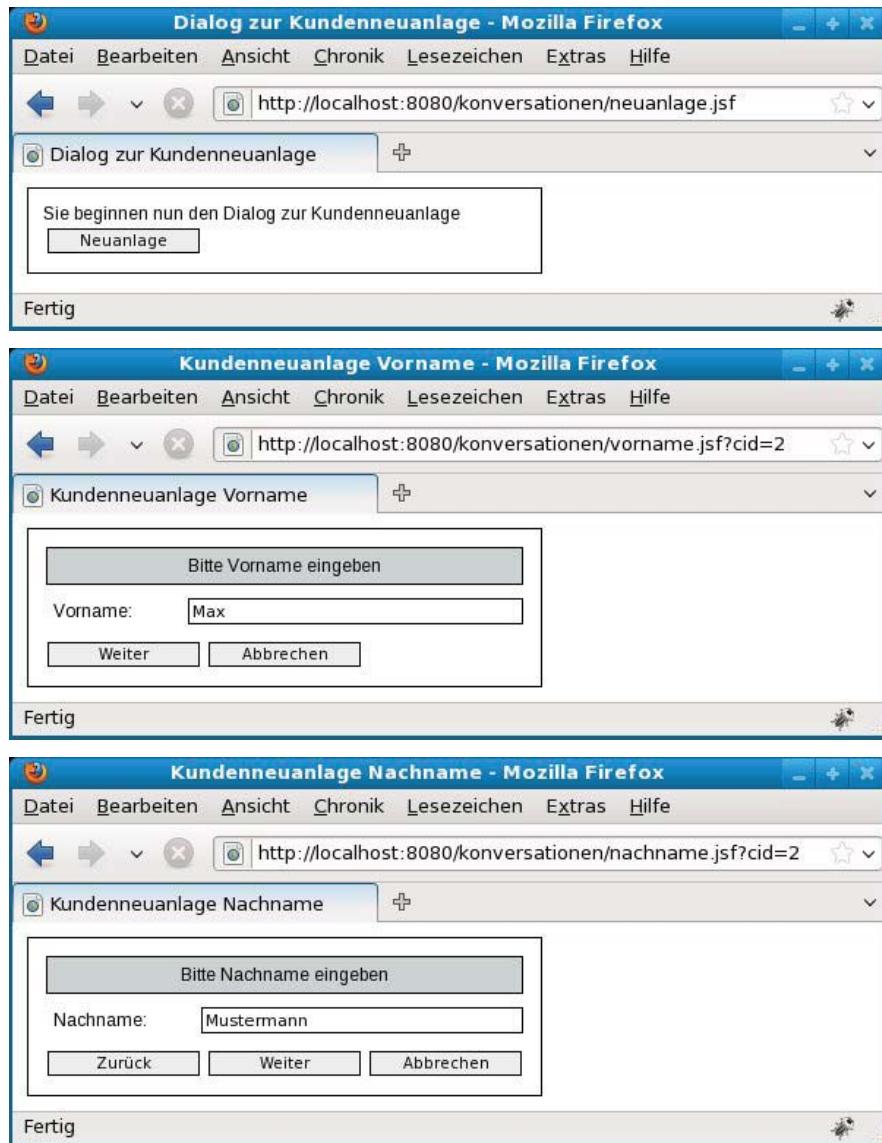


Abbildung 9.3: Wizard zur Neukundenanlage (Teil 1)

```
24  
25     public String geburtstag() {  
26         return "geburtstag?faces-redirect=true";  
27     }  
28  
29     public String anlegen() {  
30         // hier vorname, nachname, geburtstag verarbeiten
```

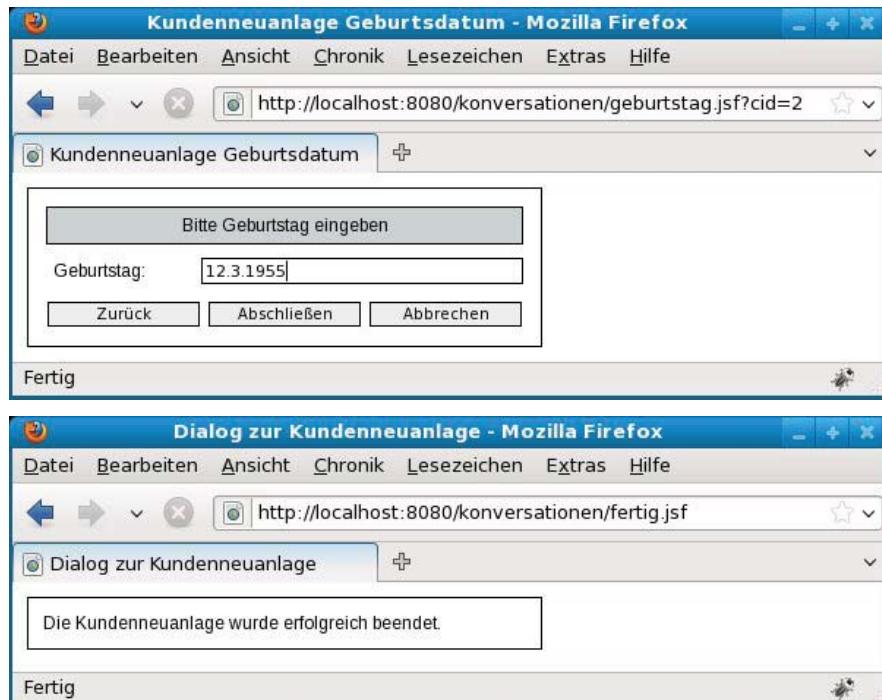


Abbildung 9.4: Wizard zur Neukundenanlage (Teil 2)

```
31     conversation.end();
32     return "fertig?faces-redirect=true";
33 }
34
35 public String abbrechen() {
36     conversation.end();
37     return "abgebrochen?faces-redirect=true";
38 }
39
40 // Getter und Setter
41 ...
42 }
```

In den Zeilen 5/6 wird mit Hilfe der Annotation `@Inject` eine Conversation-Instanz injiziert, um an geeigneter Stelle deren `begin()`- und `end()`-Methoden aufrufen zu können. Alle dargestellten Methoden der Bean sind Action-Methoden und werden an die entsprechenden `<h:commandButton>`-Tags der JSF-Seiten gebunden. Die Methoden `vorname()`, `nachname()` und `geburstag()` dienen der reinen Navigation, die Methoden `neuanlage()`, `anlegen()` und `abbrechen()` rufen zusätzlich die `begin()`- bzw. `end()`-Methode auf.

Dem aufmerksamen Leser wird aufgefallen sein, dass sowohl die Methode `neuanlage()` als auch die Methode `vorname()` zur View `vorname.xhtml` navigieren. Könnte hier nicht optimiert werden und in der Zurück-Schaltfläche der View `nachname.xhtml` ebenfalls die Methode `neuanlage()` aufgerufen und damit auf die Methode `vorname()` verzichtet werden? Nein! Der Aufruf der Methode `neuanlage()` von der View `nachname.xhtml` führt zu einer Exception. Da die View `nachname.xhtml` sich innerhalb der Konversation befindet, wird `begin()` nach dem initialen Aufruf ein zweites Mal aufgerufen, was nicht erlaubt ist, da Konversationen nicht verschachtelt werden können.

Die dargestellte Bean hat als Lebensdauer die Konversation. Würden in der Bean weitere Objekte erzeugt, so wären diese am Ende der Konversation ebenfalls entfernt worden. Sie führen daher nicht zu einem zusätzlichen Speicherbedarf, wie im Falle der Verwendung des Session-Scopes.

Wir haben eine Konversation als Folge von fünf Seiten erstellt, deren innere drei Seiten als Wizard zur Vervollständigung der benötigten Daten dienen. Es muss nun verhindert werden, dass unvollständige Daten am Ende der Konversation verarbeitet werden. Also beispielsweise, dass durch einen direkten Sprung zur Nachnamenseite lediglich Nachname und Geburtstag eingegeben und gespeichert werden. Dies würde im aktuellen Zustand zu einem Fehler führen, da die `end()`-Methode aufgerufen wird, ohne zuvor die `begin()`-Methode aufgerufen zu haben. Wir wollen dem Benutzer diesen Fehler ersparen und die Vollständigkeit auf Anwendungsebene gewährleisten. Dazu verwenden wir ein System-Event, das überprüft, ob die Konversation begonnen wurde. Wenn nicht, wird zur Anfangsseite des Wizards gesprungen. Die Idee ist also dieselbe wie bei der Prüfung auf erfolgte Authentifizierung in Kapitel 8. Hierzu werden die JSF-Seiten `vorname.xhtml`, `nachname.xhtml` und `geburtstag.xhtml` mit einem Event-Listener versehen, und zwar, bevor die Seite gerendert wird:

```
<f:event listener="#{kundenHandler.checkConversation}"  
        type="preRenderView" />
```

Die entsprechende Listener-Methode verwendet die Methode `isTransient()` des Interface `Conversation`, um zu prüfen, ob die Konversation transient oder langlaufend ist:

```
public void checkConversation(ComponentSystemEvent cse) {  
    FacesContext context = FacesContext.getCurrentInstance();  
    if (conversation.isTransient()) {  
        context.getApplication().getNavigationHandler()  
            .handleNavigation(context, null,  
                "/neuanlage.xhtml?faces-redirect=true");  
    }  
}
```

Mit diesen Änderungen ist die Vollständigkeit bzw. die Alles-oder-Nichts-Eigenschaft der Konversation gewährleistet und unser Beispiel einer Konversation abgeschlossen.



Projekt

Das Projekt *konversationen* enthält den Code für den Wizard zur Kundenneuanlage.

Aufgabe 9.1

Installieren Sie das Projekt *konversationen*. Führen Sie in zwei Browser-Tabs parallel zwei Kundenneuanlagen durch, und achten Sie auf den Parameterteil der URL-Leiste des Browsers. Sie erkennen „?cid=<n>“ für ein bestimmtes *n*. Dieser Parameterteil ist auch in den Abbildungen 9.3 und 9.4 mit dem Wert 2 zu erkennen. Die Zahl ist die Conversation-Id (cid) und wird intern zur Unterscheidung der Konversationen verwendet. Ändern Sie die Methode `anlegen()` so ab, dass die Conversation-Id in das Log geschrieben wird. Dazu steht im Interface `Conversation` die Methode `getId()` zur Verfügung.

9.5 Weitere Neuerungen in Java-EE 6

Java-EE 6 enthält eine Reihe weiterer Neuerungen, auf die wir noch nicht eingegangen sind und auch nicht vollständig eingehen können, da sie zu zahlreich sind. Wir wollen an dieser Stelle noch drei für JSF relevante Entwicklungen ansprechen. Es sind dies Servlet 3.0, das Web-Profile sowie die im Zusammenhang mit CDI bereits angesprochene alternative Möglichkeit zur Definition von Managed Beans.

9.5.1 Servlet 3.0

Die in Java-EE 6 enthaltene Servlet-Spezifikation 3.0 ist als JSR-315 [URL-JSR315] veröffentlicht. Sie versucht neben servlet-spezifischen Zielen auch eine allgemeine Vereinfachung der Konfiguration zu realisieren. Dazu wird, wie bei JSF, auf die Annotation von Klassen zurückgegriffen, so dass die Konfiguration über XML optional wird. Über die Annotationen `@WebServlet`, `@WebFilter` und `@WebListener` können Servlets, Filter und Listener deklariert werden. Falls derart annotierte Klassen im Verzeichnis `/WEB-INF/classes` liegen oder in Jars in `/WEB-INF/lib` gepackt sind, werden sie vom Servlet-Container automatisch gefunden und entsprechend behandelt.

Für das Faces-Servlet bedeutet dies, dass die Konfiguration mit `<servlet>` und `<servlet-mapping>`, wie wir sie in Abschnitt 4.8.1 vorgenommen haben, optional ist. Anfragen mit den Mustern `*.faces`, `*.jsf` und `/faces/*` werden automatisch an das Faces-Servlet weitergeleitet. Da mit JSF 2.0 auch die JSF-Konfigurationsdatei `faces-config.xml` optional ist, kann eine JSF-Anwendung sogar völlig ohne XML auskommen.

In der Praxis wird eine JSF-Anwendung jedoch nicht ohne die Verwendung von Kontextparametern zur Konfiguration (siehe Tabelle 4.12 auf Seite 163) bestehen können, so dass die Datei `web.xml` benötigt wird. Ähnliches gilt für die Datei `faces-config.xml`.



Projekt

Das Projekt *servlet-3.0* enthält eine JSF-Anwendung der Art *Hello World*, die ohne `web.xml` und ohne `faces-config.xml` auskommt.

9.5.2 Web-Profile

Durch die Integration neuer Features und wegen der für Unternehmensanwendungen sehr wichtigen Aspekte Kontinuität und Kompatibilität wuchsen die Java-EE ausmachenden Spezifikationen sowohl in ihrer Anzahl als auch in ihrem Umfang von Version zu Version. Mit Java-EE 6 wurde zum ersten Mal versucht, alte, nicht mehr verwendete APIs aus der Spezifikation zu entfernen und die Möglichkeit zur Profil-Bildung zu definieren. Unter einem Profil versteht man eine Teilmenge der Java-EE-Spezifikationen plus eventuell weiterer Spezifikationen, die für eine bestimmte Art von Anwendungen ausgerichtet sind. Das erste und bisher einzige solche Profil ist das *Web-Profile*, das innerhalb des JSR-316 [URL-JSR316] definiert wurde. Als Zielgruppe des Web-Profiles werden in der Spezifikation „*developers of modern web applications*“ genannt. Tabelle 9.1 zählt die das Web-Profile definierenden Spezifikationen auf. Die EJB-Spezifikation verzichtet in der Lite-Variante auf die Möglichkeit der Verwendung entfernter Objekte, wie wir sie in Abschnitt 9.1 im Zusammenhang mit der `@Remote`-Annotation kurz angesprochen haben. Die weiteren von uns in diesem Buch verwendeten APIs sind im Web-Profile enthalten: Bean-Validation, JPA, JTA, CDI. Bis auf die Comedian-Anwendung, die wir in Abschnitt 9.1 auf der Basis von Java-EE 5 als Enterprise-Anwendung (EAR-Format) entwickelt haben, sind damit alle Beispiele des Buches im WAR-Format auch im Web-Profile lauffähig.

Der Glassfish-Container, auf den wir in Abschnitt 10.1 näher eingehen, ist auch als spezialisierter Container für das Web-Profile verfügbar. Die Distributionsdatei dieser Version ist 47 MB groß. Der Start des Containers und das Deployen einer Anwendung benötigt wenige Sekunden. Die Comedian-Anwendung, die wir in Abschnitt 9.3 mit JSF, EJB, JPA und CDI entwickelt haben, hat als deploybares WAR-Archiv eine Größe von 10 kB. Man kann daher sicher mit Fug und Recht behaupten, dass Java-EE 6 im Allgemeinen, insbesondere aber im Web-Profile einen leichtgewichtigen Ansatz für Unternehmensanwendungen darstellt.

Tabelle 9.1: Spezifikationen des Java-EE-6-Web-Profiles

Spezifikation	Version
Servlet	3.0
JavaServer Pages (JSP)	2.2
Expression Language (EL)	2.2
Debugging Support for Other Languages (JSR-45)	1.0
Standard Tag Library for JavaServer Pages (JSTL)	1.2
JavaServer Faces (JSF)	2.0
Common Annotations for Java Platform (JSR-250)	1.1
Enterprise JavaBeans (EJB) Lite	3.1
Java Transaction API (JTA)	1.1
Java Persistence API (JPA)	2.0
Bean Validation	1.0
Managed Beans	1.0
Interceptors	1.1
CDI (JSR-299)	1.0
DI for Java (JSR-330)	1.0

9.5.3 Managed Beans

Java-EE 6 führt die Spezifikation *Managed Beans 1.0* [URL-JSR316] ein, die das Konzept einer Managed Bean, wie man es von JSF kennt, auf alle Bereiche von Java-EE ausdehnt. Die Spezifikation definiert die Annotation `@ManagedBean` und einige wenige weitere. Als Package wird `javax.annotation` verwendet. Neben der in Abschnitt 9.3 eingeführten `@Named`-Annotation und der nativen JSF-Annotation `@ManagedBean` gibt es nun also drei Alternativen,

um eine Klasse als container-verwaltet zu kennzeichnen und deren Instanzen in EL-Ausdrücken verwenden zu können.

Als Entwickler haben Sie die Qual der Wahl. Wir gehen davon aus, dass der Alternative des Packages `javax.annotation` wenig Erfolg beschieden sein wird, und raten Ihnen zur Verwendung der JSF-eigenen Annotation. Da die hinter CDI stehenden Konzepte und das Programmiermodell sehr attraktiv sind, wird das noch junge CDI wahrscheinlich weite Verbreitung finden. In diesem Fall wird auch die Verwendung der JSF-eigenen Annotationen zurückgehen, da CDI einen vollständigen Ersatz und in vielen Bereichen weitere Einsatzmöglichkeiten bietet. Dies kann zur Zeit jedoch nicht verbindlich festgestellt werden, so dass wir den Leser nur aufmuntern können, sich mit beiden Alternativen zu beschäftigen und selbst zu entscheiden.

9.6 Authentifizierung und Autorisierung mit JBoss Seam

Die Sicherheit von Unternehmensanwendungen ist von hoher Relevanz, so dass die im Java-EE- und EJB-Bereich vorhandenen rudimentären Möglichkeiten häufig nicht ausreichen und explizite Sicherheits-Frameworks zum Einsatz kommen oder individuelle Lösungen realisiert werden. Auf der Web-Ebene enthält die Servlet-Spezifikation Mechanismen, um den Zugriff auf Seiten über ihre URLs zu beschreiben. Die EJB-Spezifikation enthält Mechanismen, um über den Deployment-Deskriptor, über Annotationen oder programmatisch den Zugriff auf Klassen und Methoden zu überwachen. Seam enthält mächtigere und über die EJB-Spezifikation hinausgehende Mechanismen zur Authentifizierung und Autorisierung, die wir im Folgenden vorstellen.

Das in der EJB-Spezifikation beschriebene deklarative Sicherheitsmanagement wird auch von Seam verfolgt und auf JSF-Ebene über EL-Ausdrücke, auf Bean-Ebene über Annotationen und EL-Ausdrücke realisiert. Die hier beschriebenen Mechanismen sind in Seam Version 2.1 enthalten.

Authentifizierung

Die Authentifizierung in Seam erfolgt über eine JAAS-basierte (Java Authentication and Authorization Service) Anwendungsschicht, die sehr flexibel auf projektspezifische Anforderungen angepasst werden kann. So kann etwa eine beliebige Methode als Authentifizierungsmethode konfiguriert werden, die z. B. über einen Datenbankzugriff oder einen Namensdienst die Authentifizie-

rungsdaten prüft. Alternativ lassen sich Annotationen verwenden, so dass keine explizite Programmierung mehr erforderlich ist.

Authentifizierungs- und Autorisierungsmechanismen gehen in der Regel von Benutzern und Rollen aus. Diese müssen als JPA-Klassen realisiert und Schlüssel-Properties durch Annotationen gekennzeichnet werden. Die Entity-Klasse **Benutzer** unseres zu entwickelnden Beispiels enthält ein Property als eindeutige Benutzerkennung, den sogenannten *Principal*. Dieses Property ist mit `@UserPrincipal` annotiert.

```
private String name;

@Column(unique = true, nullable = false)
@UserPrincipal
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
```

Das Passwort wird mit `@Password` annotiert.

```
private String passwort;

@UserPassword
public String getPassword() {
    return passwort;
}
public void setPassword(String passwort) {
    this.passwort = passwort;
}
```

Optional kann eine Verschlüsselung des Passworts durch Verwendung des Attributs `hash` erfolgen. Mögliche Werte sind `md5` und `sha`, also z.B. `@UserPassword(hash = "md5")`.

Zwischen Benutzern und Rollen existiert in der Regel eine n-zu-m-Beziehung. Das entsprechende Property wird mit `@UserRoles` annotiert.

```
private Set<Rolle> rollen;

@UserRoles
@ManyToMany
@JoinTable(
    name = "Benutzer_Rolle",
    joinColumns = { @JoinColumn(name = "benutzer") },
    inverseJoinColumns = { @JoinColumn(name = "rolle") }
)
public Set<Rolle> getRollen() {
```

```
        return rollen;
    }
    public void setRollen(Set<Rolle> rollen) {
        this.rollen = rollen;
    }
```

Die Rollen-Klasse gestaltet sich noch einfacher. Hier ist lediglich das Property des Rollennamens mit `@RoleName` zu annotieren.

```
private String rolle;

@RoleName
public String getRolle() {
    return rolle;
}
public void setRolle(String rolle) {
    this.rolle = rolle;
}
```

Damit Seam erkennt, dass diese beiden JPA-Klassen zur Authentifizierung zu verwenden sind, muss dies im Komponentendeskriptor `components.xml` deklariert werden.

```
<security:identity />
<security:jpa-identity-store
    user-class="de.jsfpraxis.Benutzer"
    role-class="de.jsfpraxis.Rolle"/>
```

Das erste Element weist Seam an, die Identity-Komponente zur Authentifizierung zu nutzen. Wir werden diese gleich in einer JSF-Seite verwenden. Das zweite Element definiert die zu verwendenden JPA-Klassen. Bei der Verwendung von LDAP ist alternativ `<security:ldap-identity-store>` zu deklarieren.

Die Verwendung innerhalb einer JSF-Seite ist einfach, es sind lediglich die zur Authentifizierung benötigten Daten einzugeben und die `login()`-Methode der Identity-Komponente aufzurufen. Die Dateneingabe erfolgt für die Credentials-Komponente, wie der folgende Code-Ausschnitt zeigt:

```
<h:panelGrid columns="2">
    <h:outputLabel for="username">Benutzername</h:outputLabel>
    <h:inputText id="username" value="#{credentials.username}" />
    <h:outputLabel for="password">Passwort</h:outputLabel>
    <h:inputSecret id="password" value="#{credentials.password}" />
</h:panelGrid>
<h:commandButton value="Login" action="#{identity.login}" />
```

Die dargestellten Code-Ausschnitte realisieren die Authentifizierung komplett. Durch die Annotationen in den Entity-Klassen können Benutzername, Passwort und Rollen identifiziert werden. Die `login()`-Methode macht hiervon automatisch Gebrauch. Es ist keine JPA-Query oder Ähnliches zu implementieren.

Autorisierung für Seiten und Seitenfragmente

Nachdem der Benutzer sich authentifiziert hat und ihm eine oder mehrere Rollen zugewiesen wurden, können JSF-Seiten und Seitenfragmente sowie Klassen oder Methoden bezüglich des Benutzers oder seiner Rollen im Zugriff beschränkt werden.

Ganze Seiten werden in der Seitenkonfigurationsdatei `pages.xml` vor nicht autorisiertem Zugriff geschützt. Soll die Seite nur für bereits authentifizierte Benutzer erreichbar sein, so wird die Seite über das boolesche Attribut `login-required` geschützt. Die bei nicht angemeldetem Benutzer geworfene `NotLoggedInException` kann verarbeitet oder durch eine Default-Log-In-Seite vermieden werden.

```
<pages login-view-id="/login.xhtml">
    <page view-id="/comedians.xhtml" login-required="true" />
    <page view-id="/delete.xhtml" login-required="true" />
</pages>
```

In diesem Fall führt der Zugriff eines nicht angemeldeten Benutzers auf eine der beiden Seiten zur automatischen Anzeige der Login-Seite. Das Verhalten entspricht also dem von uns im Abschnitt 8.3 entwickelten Verfahren mit einem expliziten Event-Listener. Nach einer erfolgreichen Anmeldung kann man in einer Seam-Anwendung zur ursprünglichen Seite automatisch zurückkehren. Dazu müssen im Komponenten-Deskriptor die beiden folgenden Events-Aktionen registriert werden.

```
<event type="org.jboss.seam.security.notLoggedIn">
    <action execute="#{redirect.captureCurrentView}" />
</event>
<event type="org.jboss.seam.security.loginSuccessful">
    <action execute="#{redirect.returnToCapturedView}" />
</event>
```

Tritt die `NotLoggedInException` auf, wird das `notLoggedIn`-Event geworfen. Mit der ersten Regel beginnt die Build-In-Komponente `redirect` eine lang laufende Konversation, die konzeptionell der CDI-Konversation aus Abschnitt 9.4 entspricht, und merkt sich die angefragte Seite. Nach erfolgreichem Anmelden wird das Event `loginSuccessful` geworfen. Mit der zweiten Regel wird auf

dieses Event reagiert, zur zuvor gespreicherten Seite weitergeleitet und die Konversation beendet.

Falls nicht komplette Seiten, sondern Seitenfragmente nicht dargestellt oder nicht verwendbar sein sollen, können Sie zwei EL-Funktionen verwenden, die rollen- bzw. rechtebasiert arbeiten. Es sind dies `s:hasRole()` und `s:hasPermission()`.

```
<h:commandButton rendered="#{s:hasRole('admin')}"  
    action="..." value="..." />  
<h:commandButton disabled="#{not s:hasRole('admin')}"  
    action="..." value="..." />
```

Die erste Schaltfläche wird nur dargestellt, wenn der angemeldete Benutzer die Rolle `admin` hat. Die zweite Schaltfläche wird für alle Benutzer, die nicht die Rolle `admin` haben, als nicht aktiv dargestellt. Die EL-Funktion `s:hasRole()` kann auch in komplexeren Ausdrücken mit logischen Operatoren auftreten, etwa `"#{s:hasRole('rolle1') and not s:hasRole('rolle2')}"`.

Autorisierung für Komponentenauftrufe

Die bisher eingeführte Rechtevergabe für den Zugriff auf JSF-Komponenten ist analog auch für Seam-Komponenten verfügbar. Die Annotation `@Restrict` erlaubt im `value`-Attribut beliebige EL-Ausdrücke und somit auch `s:hasRole()` und das noch vorzustellende `s:hasPermission()`. Es können sowohl Klassen als auch Methoden annotiert werden. Annotiert man die Klasse, wird der Zugriff auf alle Methodenaufrufe bzgl. des `value`-Attributs überprüft, ansonsten nur für die annotierte Methode.

```
@Name("manager")  
@Restrict("#{s:hasRole('admin')})  
public class Manager {  
  
    public void insert() { ... }  
    public void delete() { ... }  
    ...  
}
```

In obigem Beispiel können die Methoden der Komponente `Manager` nur aufgerufen werden, wenn der angemeldete Benutzer die Rolle `admin` hat. Dies entspricht in der Grundversion der in EJB3 vorhandenen Rechtevergabe, geht aber mit der Möglichkeit der Konstruktion von logischen Ausdrücken insbesondere mit `s:hasPermission()` deutlich über EJB3 hinaus.

Bei JSF-Seiten haben wir bisher nur zwischen angemeldeten und nicht angemeldeten Benutzern mit dem Seitenattribut `login-required` unterscheiden

können. Ein ähnliches Konstrukt, wie die `@Restrict`-Annotation auf Java-Ebene, ist auch im Seitendeskriptor `pages.xml` mit dem Element `<restrict>` möglich. Das folgende Beispiel erlaubt den Zugriff auf die Seite `benutzerverwaltung.xhtml` nur für Benutzer der Rolle `admin`.

```
<page view-id="/benutzerverwaltung.xhtml">
    <restrict>#{s:hasRole('admin')}</restrict>
</page>
```

Kontextabhängige Rechtevergabe

Für einige Anwendungsfälle ist eine Autorisierungskomponente auf Rollenbasis nicht granular genug. Seam erlaubt es, regelbasierte Zugriffsrechte zu definieren, und dies sogar kontextabhängig. Das verwendete Framework *Drools*, ebenfalls ein JBoss-Projekt, ist ein regelbasiertes Inferenzsystem, das auf Basis des Rete-Algorithmus die Definition vorwärtsverkettender Regeln erlaubt. Drools ist in Seam integriert und erlaubt die Definition der entsprechenden Regeln, in deren Syntax und Semantik wir hier nur sehr oberflächlich einführen können.

Vorwärtsverkettende Regelssysteme basieren auf Wenn-Dann-Regeln. Wenn eine oder mehrere Prämisse erfüllt sind, folgt daraus die Konklusion. Drools verwendet `when` und `then`, um diese Regelstruktur zu realisieren. Zusätzlich wird eine Regel mit einem Namen versehen. Wir erläutern Details an einem Beispiel:

```
rule Test
    no-loop
    activation-group "permissions"
when
    check: PermissionCheck(target == "button",
                            action == "render", granted == false)
    Role(name == "admin")
then
    check.grant();
end
```

Die abgebildete Drools-Regel verwendet `PermissionCheck()` und `Role()` als Prämisse, die implizit mit einem logischen *Und* verknüpft sind. Die beiden Prämisse werden als Fakten in den Drools-Arbeitsspeicher eingefügt, und Drools versucht, die Regel anzuwenden. Die Regel feuert, wenn beide Fakten zu *wahr* evaluieren. Der erste Fakt ist immer wahr und hat als Seitenefekt, dass die Regel mit Namen `Test` mit Target `button` und Action `render` in der EL-Funktion `s:hasPermission()` verwendet werden kann und als Erlaubnis den Wert `false` hat. Der Drools-Arbeitsspeicher wird in der Session verwaltet. Er enthält bei einer erfolgreichen Authentifizierung eines Benutzers

den Benutzer-Principal (Klasse `java.security.Principal`) und dessen Rolle (Klasse `org.jboss.seam.security.Role`) ebenfalls als Fakten. Im Beispiel verwenden wir die Rolle mit dem Test, ob der Rollenname `admin` ist. Wenn die Prämisse wahr ist, feuert die Regel, und der Zugriff wird erlaubt. Wir können also schreiben:

```
<h:commandButton  
    rendered="#{s:hasPermission('button', 'render')}"  
    action="..." value="..." />
```

Das Beispiel ist statisch im Sinne einer nicht kontextabhängigen Auswertung der Zugriffsregel. Als Beispiel einer kontextabhängigen Rechtevergabe überarbeiten wir die Anzeige aller Comedians mit dem `<h:dataTable>`-Tag derart, dass sich auch Comedians als Benutzer authentifizieren können und ein solcher Benutzer nur seine eigenen Daten verändern darf. Eine Schaltfläche in einer Spalte der Tabelle soll also abhängig von der Zeile bzw. dem in dieser Zeile dargestellten Comedian gerendert werden oder nicht.

Wir gehen für das Beispiel davon aus, dass der Vorname des Comedians (Property `vorname` der Entity-Klasse `Comedian`) und der Benutzername (Property `name` der Entity-Klasse `Benutzer`) übereinstimmen müssen. Die folgende Regel leistet diese Prüfung auf Übereinstimmung:

```
rule EditAllowed  
    no-loop  
    activation-group "permissions"  
when  
    comedian: Comedian($vorname: vorname)  
    check: PermissionCheck(target == comedian, action == "test",  
                            granted == false)  
    Principal(name == $vorname)  
then  
    check.grant();  
end
```

Zunächst ist im Prädikat `PermissionCheck` der Wert des Attributs `target` keine String-Konstante mehr, sondern eine Variable. Diese Variable steht für eine `Comedian`-Instanz im aktuellen Drools-Arbeitsspeicher. Das Property `$vorname` bekommt den Wert des Bean-Property `vorname` zugewiesen. Für den angemeldeten Benutzer existiert, wie schon erwähnt, ein Principal-Prädikat im Drools-Arbeitsspeicher. Dessen `name`-Property wird mit `$vorname` auf Gleichheit getestet. Drools arbeitet hier korrekt und verwendet intern `equals()` für den String-Vergleich.

Falls die Drools-eigenen Programmierkonstrukte nicht ausreichen, kann etwa innerhalb des `eval()`-Prädikats auf Java zurückgegriffen werden. Wenn

für den Vergleich von Vorname des Comedian und Benutzername keine Unterscheidung auf Groß/Kleinschreibung stattfinden soll, so könnte der oben dargestellte Wenn-Teil der Regel alternativ auch folgendermaßen formuliert werden:

```
when
    comedian: Comedian()
    check: PermissionCheck(target == comedian, action == "test",
                           granted == false)
    principal: Principal()
    eval (comedian.getVorname().toLowerCase()
          .equals(principal.getName().toLowerCase()))
```

Interessant ist nun die Verwendung innerhalb von JSF oder Java. Wir überarbeiten die JSF-Seite zur Darstellung aller Comedians, geben aber nur die wichtigen Bereiche an:

```
<h:dataTable var="comedian" value="#{comedians}">
    <h:column>
        <f:facet name="header">Vorname</f:facet>
        #{comedian.vorname}
    </h:column>
    ...
    <h:column>
        <h:commandButton
            rendered="#{s:hasPermission(comedian, 'test')}"
            value="Ändern" action="#{comedianHandler.edit}" />
    </h:column>
</h:dataTable>
```

Man erkennt die Verwendung der Iterationsvariablen `comedian` des `<h:dataTable>`-Tags als ersten Parameter von `s:hasPermission()`. Der zweite Parameter ist der Wert des `action`-Attributs. Der eigentliche Name der Regel findet keine Verwendung und wird nur intern von Drools zur Unterscheidung verwendet. In diesem Beispiel wird also für einen dem System bekannten und aktuell eingeloggten Comedian nur die Zeile in der Datentabelle mit der Schaltfläche versehen, die seine eigenen Daten enthält.

Wir beschließen an dieser Stelle unsere Ausführungen zu den von Seam bereitgestellten Mechanismen im Bereich Authentifizierung und Autorisierung. Wir konnten nur einen sehr kleinen Ausschnitt dieser Mechanismen darstellen. Andere interessante Aspekte Seams haben wir nicht angesprochen. So ist es etwa möglich, Konversationen ineinander zu verschachteln, oder Entities in Auswahlmenüs, z.B. `<h:selectOneMenu>` direkt ohne Konvertierer zu verwenden. Wir raten dem interessierten Leser, einen Blick auf Seam zu werfen [URL-SEAM].

JBoss hat auf Basis der Erfahrungen mit Seam den JSR-299 (CDI) initiiert. Der geistige Vater von Seam, Gavin King, war Leiter der Spezifikationsgruppe des JSR-299. Während dieses Buch entsteht, wird Seam überarbeitet und CDI als Grundlage der internen Seam-Mechanismen verwendet. Die CDI-Referenzimplementierung *Weld* wird als Teilprojekt auf den Seam-Seiten [URL-SEAM] entwickelt.



Projekt

Das Projekt *comedians-seam* enthält den Code für die Comedian-Verwaltung mit JBoss Seam und den dargestellten Sicherheitsmechanismen.

Kapitel 10

Systeme und Werkzeuge

Die Zeiten von Vi und Emacs sind vorbei, zumindest wenn es um die effiziente Erstellung von Anwendungen auf der Basis komplexer Frameworks geht, wie es Java-EE eines ist. Je komplexer die Frameworks werden, desto komplexer werden auch die unterstützenden Systeme. Wir können daher an dieser Stelle nur einen kurzen Einblick in einige der interessantesten Systeme geben, die im Umfeld der Entwicklung von Anwendungen mit JavaServer Faces existieren.

Problematisch bei der Beschreibung von Systemen und Werkzeugen ist die Schnelllebigkeit der Branche und damit verbunden die häufigen Release-Wechsel. Alle folgenden Beschreibungen beziehen sich auf die im Sommer 2010 aktuellen Versionen. Es sind dies GlassFish 3.0, Eclipse 3.5, Firebug 1.5.4, Selenium 1.0.7 und JSFUnit 1.2.0.

10.1 GlassFish

Der Application-Server GlassFish wird unter einer Open-Source-Lizenz entwickelt und ist im Netz unter [URL-GF] frei verfügbar. Ebenfalls verfügbar ist eine von Oracle erweiterte Version, die für Evaluationzwecke frei verwendet werden kann, für den produktiven Betrieb jedoch eine kommerzielle Lizenz benötigt. Wir haben für alle Beispiele des Buches den *GlassFish Server Open Source Edition* in der Version 3.0 verwendet. Sie können alternativ auch die Version verwenden, die nur das Web-Profile realisiert.

10.1.1 Installation und Betrieb

Wir beschreiben zunächst die Installation und den Betrieb im kommandozeilenorientierten Stand-Alone-Modus und werden später, in Abschnitt 10.2, den Betrieb mit Eclipse vorstellen.

Für die Installation stehen drei Versionen bereit: eine für Windows, eine für Unix-artige Betriebssysteme und eine betriebssystemunabhängige. Wir verwenden die betriebssystemunabhängige Version, die aus einer Zip-Datei besteht. Nach dem Entpacken der Datei befinden sich im Installationsverzeichnis `glassfishv3` mehrere Verzeichnisse, von denen `bin` und `glassfish` die wichtigsten sind. Im Verzeichnis `bin` befindet sich die zentrale Kommando- datei `asadmin`, jeweils in Unix- und Windowsversion, über die alle weiteren GlassFish-Befehle erfolgen.

Um den Server in der Default-Konfiguration zu starten, genügt der Befehl

```
$ asadmin start-domain
```

Hierauf werden einige informative Meldungen erzeugt. Falls keine Fehlermeldungen erscheinen, findet man unter `http://localhost:4848` die graphische Administrationskonsole, die in Abbildung 10.1 abgebildet ist.

GlassFish verwendet das Konzept einer Domain. Domains erlauben es, eine Menge von zusammengehörenden Applikationen gemeinsam zu deployen und zu betreiben. Die Default-Domain `domain1` wird bei der Installation von GlassFish automatisch angelegt. Sie befindet sich im Verzeichnis

```
glassfishv3/glassfish/domains
```

Der oben verwendete Befehl zum Start des Servers startet implizit die Domain `domain1`, so dass die Befehle

```
$ asadmin start-domain
```

und

```
$ asadmin start-domain domain1
```

beide die Domain `domain1` starten. Um eine Domain herunterzufahren, verwendet man analog einen der Befehle

```
$ asadmin stop-domain  
$ asadmin stop-domain domain1
```

Das Deployen einer Anwendung erfolgt durch einfaches Kopieren der War- oder Ear-Datei in das Verzeichnis

```
glassfishv3/glassfish/domains/domain1/autodeploy
```

Nach dem Kopieren sollten Sie den Inhalt der Log-Datei

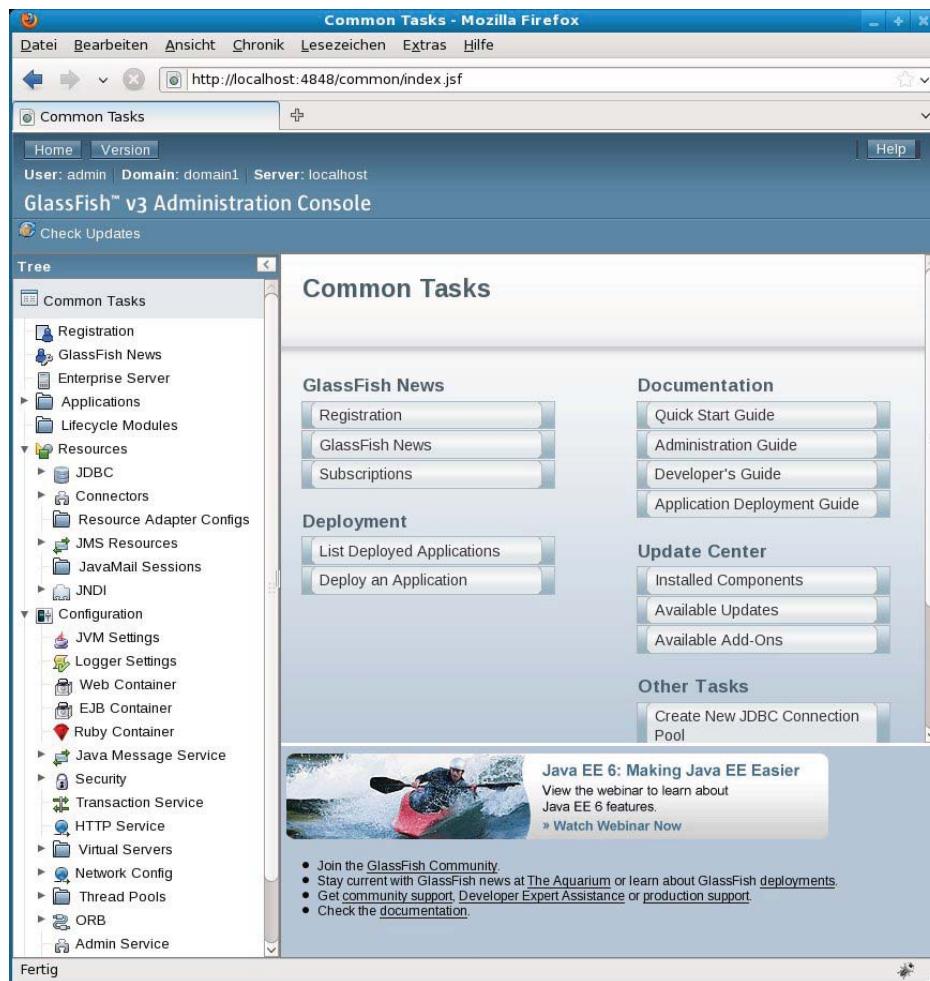


Abbildung 10.1: Administrationskonsole des GlassFish-Servers

```
glassfishv3/glassfish/domains/domain1/logs/server.log
```

beobachten. Die Anwendung steht nun unter `http://localhost:8080/<anwendung>` zur Verfügung.

Aufgabe 10.1

Laden Sie die Datei `ttt.war` von der Web-Seite des Buches herunter. Kopieren Sie sie in das Verzeichnis `glassfishv3/glassfish/domains/domain1/auto-deploy`, und beobachten Sie die Datei `glassfishv3/glassfish/domains/domain1/logs/server.log`. Öffnen Sie im Browser die URL `http://localhost:8080/ttt/`, und spielen Sie Tic-Tac-Toe.

Neben dem Befehl `start-domain` innerhalb von `asadmin` stehen noch eine ganze Reihe weiterer Befehle zur Verfügung. Diese sind im *Sun GlassFish Enterprise Server v3 Administration Guide* [URL-GFADMIN] dokumentiert. Wir wollen hier als Beispiel eine zweite Domain einrichten. Neben der Administrationskonsole und dem Web-Server für die Anwendungen werden eine Reihe weiterer Ports pro Domain verwendet, die alle explizit gesetzt werden können. Einfacher ist es jedoch, einen Basisport anzugeben, aus dem sich die anderen Ports durch Addition definierter Konstanten ergeben. Für den Befehl

```
$ asadmin create-domain --portbase 9000 domain2
```

wird die Domain `domain2` erzeugt und mehrere Dienste aufgesetzt, deren Ports in Tabelle 10.1 aufgeführt sind.

Tabelle 10.1: Verwendete GlassFish-Ports für Basis 9000

Dienst	Port
Administrationskonsole	9048
HTTP	9080
Java Messaging-System (JMS)	9076
Internet Inter-ORB (IIOP)	9037
Secure HTTP (HTTPS)	9081
Secure IIOP	9038
Mutual Authorization IIOP	9039
Java Management Extension (JMX)	9086
OSGI Shell	9066

Nach dem Start der Domain mit

```
$ asadmin start-domain domain2
```

und Deployen der Anwendung, d. h. Kopieren der Datei `ttt.war` nach

```
glassfishv3/glassfish/domains/domain2/autodeploy
```

ist unter `http://localhost:9048` die Administrationskonsole und unter `http://localhost:9080/ttt/` die Anwendung verfügbar.

10.1.2 Die Datenbank JavaDB

Die Comedian-Anwendungen in den Kapiteln 3 und 9 sowie die Online-Banking-Anwendung in Kapitel 8 verwenden die in GlassFish enthaltene Datenbank JavaDB, deren Entstehungsgeschichte erwähnenswert ist. 1996 von der Firma Cloudscape unter demselben Namen als in Java implementiertes relatio-

nales Datenbankmanagementsystem vorgestellt, ging das System 1999 durch einen Firmenaufkauf an Informix über. Informix wurde wiederum von IBM übernommen, die das Produkt unter dem Namen *IBM Cloudscape* weiter vermarktete. 2004 übertrug IBM das Datenbanksystem an die Apache Foundation, die es unter dem Namen *Derby* weiterführte [URL-DERBY]. Sun integriert Derby seit Version 6 in das Java SDK, so dass es wahrscheinlich zu den am meisten verbreiteten Datenbanksystemen gehört. Sun integriert Derby ebenfalls in den GlassFish-Application-Server, und zwar im Verzeichnis `glassfishv3/javadb` und macht JavaDB als Default-Datenbank verfügbar, so dass wir sie in unseren Anwendungen ohne weiteren Konfigurationsaufwand sofort verwenden können.

Abbildung 10.2 zeigt die JDBC-Ressourcen des GlassFish-Servers in der Administrationskonsole. Man erkennt unter dem JNDI-Namen die oben angesprochene Default-Datenbank des Servers. Diese haben wir in unseren datenbank-

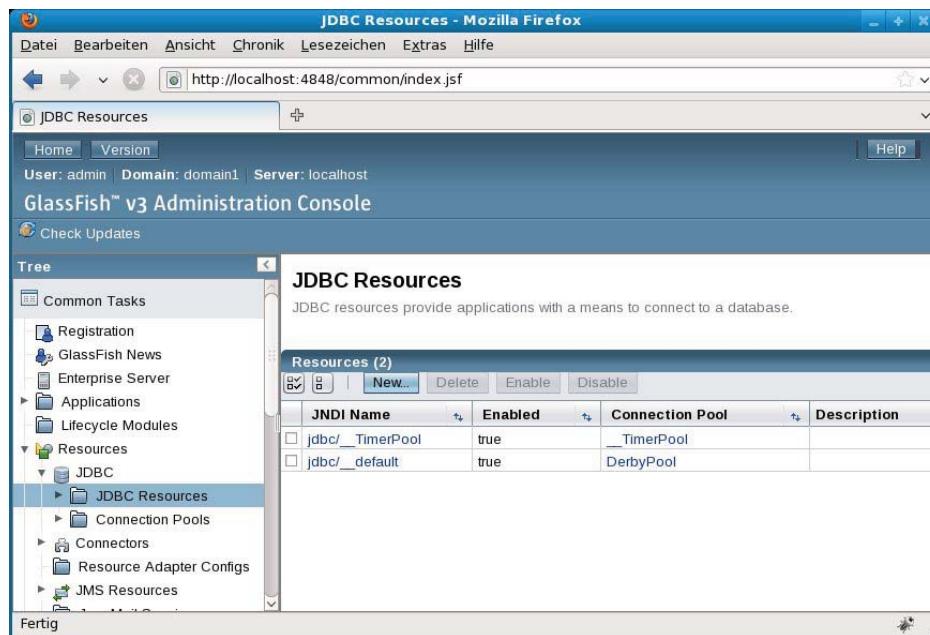


Abbildung 10.2: JavaDB als Default-Datenbank in GlassFish

basierten Anwendungen im JPA-Deployment-Deskriptor `persistence.xml` als JTA-Datenquelle verwendet, wie das folgende Beispiel zeigt:

```
<persistence version="1.0"
    xmlns="..."
    xsi:schemaLocation="...">
```

```
<persistence-unit name="comedians" transaction-type="JTA">
    <jta-data-source>jdbc/_default</jta-data-source>
    <properties>
        <property name="eclipselink.ddl-generation"
            value="drop-and-create-tables" />
        <property name="eclipselink.logging.level"
            value="FINE" />
    </properties>
</persistence-unit>
</persistence>
```

GlassFish startet die Datenbank nicht automatisch, so dass bei den datenbank-basierten Anwendungen neben dem Application-Server selbst auch die Datenbank gestartet werden muss. Dies geschieht mit dem Befehl **start-database**:

```
$ asadmin start-database
```

Entsprechend wird die Datenbank mit **stop-database** wieder heruntergefahren. Der Default-Port der Datenbank ist 1527, was für die Verwendung externer Werkzeuge wichtig zu wissen ist.

Wir beschließen unsere Ausführungen zu den in GlassFish vorhandenen Befehlen mit **list-commands**:

```
$ asadmin list-commands
```

Dieser Befehl zeigt die möglichen Befehle zur Administration des Application-Servers an. Für ausführliche Erläuterungen zu den einzelnen Befehlen konsultieren Sie bitte das Administrationshandbuch [URL-GFADMIN].

10.1.3 Konfiguration

Die Konfiguration des Application-Servers ist in weiten Bereichen möglich, kann hier aber nicht detailliert werden. Selbst die Konfigurationsmöglichkeiten von Mojarra, der JSF-Referenzimplementierung, sind so vielfältig, dass wir sie hier nicht darstellen können. Das Mojarra-Wiki beschreibt einen Teil dieser Möglichkeiten [URL-MOJCP]. Einen guten Einstieg bildet der Kontextparameter **displayConfiguration**, der dafür verantwortlich ist, dass die Konfigurationsparameter und ihre Werte im Log erscheinen. Im Gegensatz zu den in Tabelle 4.12 auf Seite 163 beschriebenen JSF-Kontextparametern beginnen die Mojarra-Kontextparameter nicht mit **javax.faces**, sondern mit **com.sun.faces**. Daher muss im Servlet-Deskriptor **web.xml** der Parameter wie folgt gesetzt werden:

```
<context-param>
    <param-name>com.sun.faces.displayConfiguration</param-name>
    <param-value>true</param-value>
```

```
</context-param>
```

Wir geben im Folgenden die darauf basierenden Ausgaben der Log-Datei wieder. Sie können dem interessierten Leser durch Interpretation der Parameternamen als Motivation für eigene Recherchen dienen. Wir führen sowohl die offiziellen JSF- als auch die Mojarra-Parameter und ihre Werte an.

Listing 10.1: Log-Meldungen für JSF- und Mojarra-Parameter (alphabetisch)

```
'com.sun.faces.allowTextChildren' - DISABLED
'com.sun.faces.autoCompleteOffOnViewState' - ENABLED
'com.sun.faces.clientStateWriteBufferSize' ist auf
    '8192' festgelegt.
'com.sun.faces.compressJavaScript' - ENABLED
'com.sun.faces.compressViewState' - ENABLED
'com.sun.faces.defaultResourceMaxAge' ist auf
    '604800000' festgelegt.
'com.sun.faces.disableUnicodeEscaping' ist auf 'auto' festgelegt.
'com.sun.faces.displayConfiguration' - ENABLED
'com.sun.faces.enableCoreTagLibValidator' - DISABLED
'com.sun.faces.enableGroovyScripting' - DISABLED
'com.sun.faces.enableHtmlTagLibValidator' - DISABLED
'com.sun.faces.enableJSStyleHiding' - DISABLED
'com.sun.faces.enableLazyBeanValidation' - ENABLED
'com.sun.faces.enableRestoreView11Compatibility' - DISABLED
'com.sun.faces.enableScriptsInAttributeValues' - ENABLED
'com.sun.faces.enableThreading' - DISABLED
'com.sun.faces.enableViewStateIdRendering' - ENABLED
'com.sun.faces.enabledLoadBundle11Compatibility' - DISABLED
'com.sun.faces.expressionFactory' ist auf
    'com.sun.el.ExpressionFactoryImpl' festgelegt.
'com.sun.faces.forceLoadConfiguration' - ENABLED
'com.sun.faces.generateUniqueServerStateIds' - ENABLED
'com.sun.faces.number0fConcerrentFlashUsers' ist auf
    '5000' festgelegt.
'com.sun.faces.number0fFlashesBetweenFlashReapings' ist auf
    '5000' festgelegt.
'com.sun.faces.number0fLogicalViews' ist auf '15' festgelegt.
'com.sun.faces.number0fViewsInSession' ist auf '15' festgelegt.
'com.sun.faces.preferXHTML' - DISABLED
'com.sun.faces.registerConverterPropertyEditors' - DISABLED
'com.sun.faces.resourceBufferSize' ist auf '2048' festgelegt.
'com.sun.faces.resourceUpdateCheckPeriod' ist auf '5' festgelegt.
'com.sun.faces.responseBufferSize' ist auf '1024' festgelegt.
'com.sun.faces.sendPoweredByHeader' - ENABLED
'com.sun.faces.serializeServerState' - DISABLED
'com.sun.faces.validateXml' - ENABLED
'com.sun.faces.verifyObjects' - DISABLED
```

```
'com.sun.faces.writeStateAtFormEnd' - ENABLED
'javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_\
    SYSTEM_TIMEZONE' - DISABLED
'javax.faces.DEFAULT_SUFFIX' ist auf '.xhtml .jsp' festgelegt.
'javax.faces.DISABLE_FACELET_JSF_VIEWHANDLER' - DISABLED
'javax.faces.FACELETS_BUFFER_SIZE' ist auf '1024' festgelegt.
'javax.faces.FACELETS_REFRESH_PERIOD' ist auf '2' festgelegt.
'javax.faces.FACELETS_SKIP_COMMENTS' - ENABLED
'javax.faces.FACELETS_SUFFIX' ist auf '.xhtml' festgelegt.
'javax.faces.PARTIAL_STATE_SAVING' - ENABLED
'javax.faces.PROJECT_STAGE' ist auf 'Development' festgelegt.
'javax.faces.RESOURCE_EXCLUDES' ist auf
    '.class .jsp .jspx .properties .xhtml .groovy' festgelegt.
'javax.faces.STATE_SAVING_METHOD' ist auf 'client' festgelegt.
'javax.faces.VALIDATE_EMPTY_FIELDS' ist auf 'auto' festgelegt.
'javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR' - DISABLED
```

Eine weitere interessante Konfigurationsmöglichkeit betrifft die client-seitige Repräsentation des Komponentenbaums zwischen zwei JSF-Anfragen. Standardmäßig erfolgt die Repräsentation in einer Base64-Codierung. Durch einen JNDI-Eintrag wird die Repräsentation verschlüsselt, so dass ein höheres Maß an Sicherheit gegeben ist. Diese bereits mit JSF 1.2 in Mojarra eingeführte Option ist in [URL-MOJF] dokumentiert.

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
</context-param>

<env-entry>
    <env-entry-name>ClientStateSavingPassword</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>Ein geheimes Passwort</env-entry-value>
</env-entry>
```

10.2 Eclipse

Eclipse wird häufig (von uns auch) als Synonym für eine Java-IDE der Eclipse Foundation verwendet. Mittlerweile ist die Eclipse Foundation [URL-ECLIPSE] jedoch eine sehr agile, unabhängige Non-Profit-Organisation, die neben einer Java-IDE eine ganze Reihe weiterer Projekte betreibt. Wir haben z. B. mit EclipseLink [URL-ELINK] die Referenzimplementierung von JPA verwendet, die in GlassFish enthalten ist.

Für die Entwicklung von Java-EE-Anwendungen empfiehlt sich Eclipse in der Version *Eclipse IDE for Java EE Developers*. Diese Version enthält unter anderem die Web-Tools-Platform und unterstützt die Entwicklung von Anwendungen auf der Basis von EJBs, JPA und JSF.

10.2.1 Installation

Eclipse ist nicht 100% Pure Java, enthält also plattformspezifischen Code. Darauf müssen Sie die zu Ihrem Betriebssystem passende Version herunterladen. Das **downloads**-Verzeichnis von [URL-ECLIPSE] zeigt automatisch nur die passenden Alternativen an. Das heruntergeladene Archiv (Zip für Windows, Tar für Linux und Mac OS X) kann an beliebiger Stelle entpackt werden.

Nach dem Entpacken steht unter Linux und Max OS X die Datei **eclipse**, unter Windows die Datei **eclipse.exe** zum Start der IDE zur Verfügung. Wir empfehlen, diese nicht direkt für den Programmstart zu verwenden, sondern eine Shell- bzw. Bat-Datei zu erzeugen, die das native Programm aufruft. In ihr können Laufzeit-Optionen für die JVM übergeben werden, um z. B. den von der JVM zu verwendenden Speicher zu vergößern. Alternativ zur Shell- bzw. Bat-Datei können Sie eine Verknüpfung angelegen, die die Optionen enthält. Eine weitere Alternative ist die Datei **eclipse.ini**, die sich im Installationsverzeichnis von Eclipse befindet. Hier können Sie dieselben Optionen verwenden. Tabelle 10.2 zeigt einige wichtige Optionen. Ein vollständiges Verzeichnis aller Optionen finden Sie in der Eclipse-Hilfe.

Tabelle 10.2: Start-Parameter für Eclipse

Parameter	Bedeutung
<code>-vm <Pfad zur VM></code>	zu verwendende JVM
<code>-data <Pfad zum Workspace></code>	zu verwendender Workspace
<code>-clean</code>	löscht den OSGI-Cache
<code>-vmargs</code>	nachfolgende Parameter an JVM weiterreichen

Das Shell-Skript auf dem Linux-Rechner des Autors enthält beispielsweise die Optionen

```
-vm $JAVA -data $WORKSPACE -vmargs -Xms1024m -Xmx1024m \
-XX:PermSize=256M -XX:MaxPermSize=512M
```

wobei `$JAVA` und `$WORKSPACE` Variablendefinitionen für die entsprechenden Verzeichnisse darstellen.

Um zu kontrollieren, dass Eclipse die Optionen korrekt übernommen hat, öffnen Sie über die Menüfolge *Help* ⇒ *About Eclipse* ⇒ *Installation Details* den Karteireiter *Configuration* und überprüfen die Optionen.

10.2.2 GlassFish-Plugin

Eclipse besitzt in der EE-Version bereits eine Reihe von Adaptoren für verschiedene Java-Application-Server, leider jedoch von Haus aus nicht für GlassFish. Sun entwickelt einen solchen Adaptor, der im Netz frei erhältlich ist [URL-GFP]. Die Installation erfolgt am einfachsten über die Auswahl der Laufzeitumgebung durch die Menüfolge *Window* ⇒ *Preferences* ⇒ *Server* ⇒ *Runtime Environments*. Es öffnet sich ein entsprechendes Fenster, wie in Abbildung 10.3 dargestellt. Da keine Laufzeitumgebung für GlassFish exi-

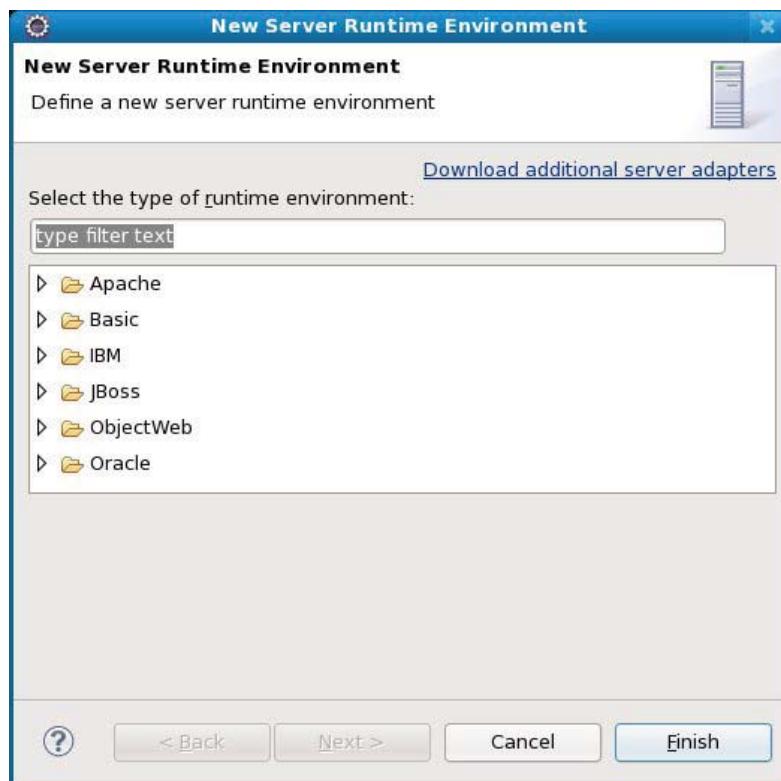


Abbildung 10.3: Auswahl einer Server-Laufzeitumgebung

stiert, wird über den Link oben rechts das Herunterladen und Installieren der Laufzeitumgebung begonnen. Im dann neu geöffneten Fenster wird GlassFish wie in Abbildung 10.4 abgebildet ausgewählt. Nach der Installation kann

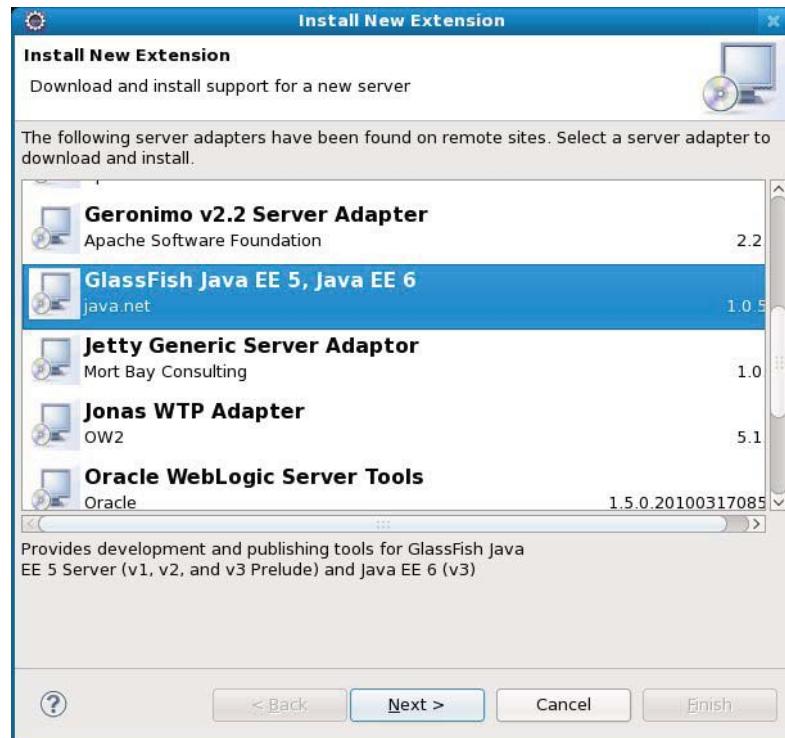


Abbildung 10.4: Installation eines neuen Server-Adaptors

nun nach erneutem Öffnen des Fensters zur Auswahl der Laufzeitumgebung GlassFish selektiert werden. Dies führt zu einem Fehler, da Eclipse das Installationsverzeichnis von GlassFish nicht kennt. Wie in Abbildung 10.5 gezeigt, muss das Installationsverzeichnis aus Abschnitt 10.1 angegeben werden. Es wird jedoch das Unterverzeichnis `glassfish` benötigt, so dass der Pfad mit `glassfishv3/glassfish` endet. Beim Anlegen eines neuen Projekts vom Typ *Dynamic Web Project* wird nun diese Laufzeitumgebung vorausgewählt, da sie die einzige ist. Werden nur Java-EE-APIs verwendet, was in fast allen unseren Projekten der Fall ist, müssen keine weiteren Bibliotheken eingebunden werden. Der Leser möge sich davon überzeugen, dass sich im Verzeichnis `glassfishv3/glassfish/modules` unter anderem die Dateien `jsf-api.jar`, `jsf-impl.jar` (JSF), `weld-osgi-bundle.jar` (CDI), `bean-validator.jar` (Bean Validation) und mehrere `org.eclipse.persistence.*.jar` (JPA) befinden, die wir in vielen unserer Projekte verwendet haben.

Bemerkung: Falls die automatische Suche nach dem Server-Plugin nicht zum gewünschten Erfolg führt, kann alternativ die URL <https://ajax.dev.java.net/>



Abbildung 10.5: Angabe des Installationsverzeichnisses

va.net/eclipse als neues Software-Repository für Eclipse ausgewählt und das Plugin installiert werden.

Wie wir gesehen haben, werden von GlassFish unabhängige Domains betrieben. Nach obiger Definition einer Laufzeitumgebung muss man daher eine oder mehrere Instanzen dieser Umgebung definieren. Eclipse bietet in der Java-EE-Perspektive im unteren Teil ein Subfenster an, das ein Servers-Tab besitzt. In diesem wird über die rechte Maus die Definition einer Server-Instanz eingeleitet. Im sich öffnenden Fenster, das in Abbildung 10.6 dargestellt ist, ist die einzige Laufzeitumgebung bereits voreingestellt. Im nächsten Schritt (Abbildung 10.7) wird die Domain definiert. Standardmäßig ist dies domain1. Nach Abschluss der Definition lässt sich in diesem Tab der Server bzw. die Domain starten und stoppen (Abbildung 10.8). Ein Projekt kann jetzt im Project Explorer über ein kontextsensitives Menü auf dem Server deployt und betrieben werden; bei Bedarf auch im Debug-Modus.

Falls Sie das Starten und Stoppen des Servers über Eclipse und nicht über die Kommandozeile durchführen, sollten Sie dies auch für die JavaDB (Abschnitt 10.1.2) so tun. Dazu öffnen Sie über *Window ⇒ Preferences ⇒ GlassFish Preferences* die GlassFish-Einstellungen und aktivieren die Check-Box „*Start the JavaDB Database Process when Starting GlassFish Server*“.

Weitere Informationen zur Verwendung von Eclipse finden Sie auf www.jsf-praxis.de.

10.2.3 JBoss Tools

Die JSF-Unterstützung im Eclipse-Teilprojekt WTP (Web Tools Project) ist, insbesondere was die Version 2.0 von JSF angeht, nicht optimal. Wir empfehlen

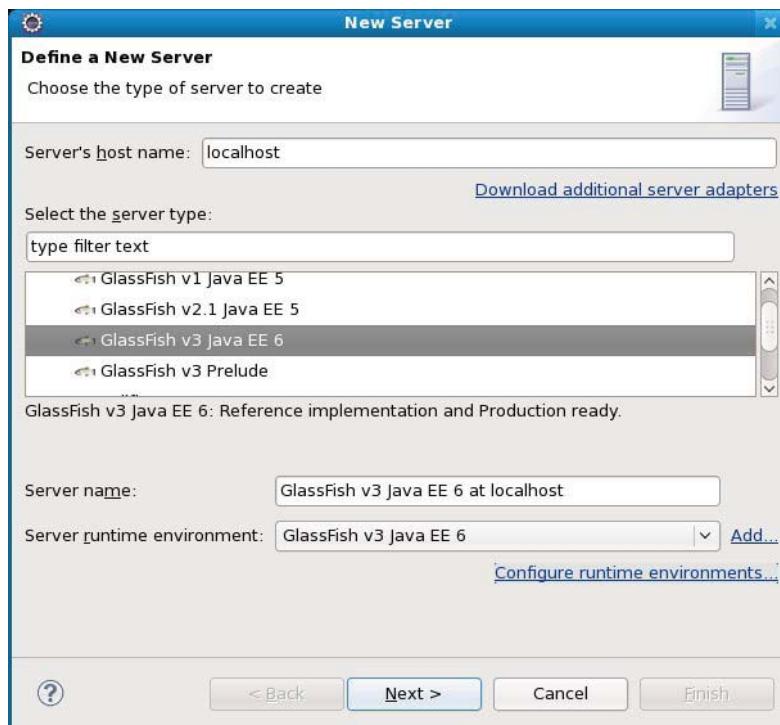


Abbildung 10.6: Definition einer Server-Instanz



Abbildung 10.7: Definition einer Server-Instanz, Teil 2

daher die Installation der JBoss-Tools [URL-JBTOOLS]. Die JBoss Tools unterstützen neben JSF auch Seam, Drools, jBPM, JPA, Hibernate und weitere aktuelle Systeme. Die Installation erfolgt mit einem der in [URL-JBTOOLS]

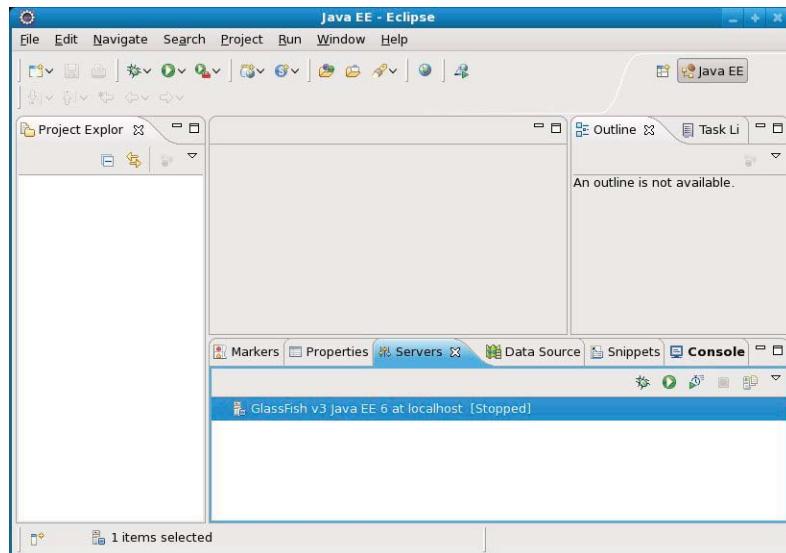


Abbildung 10.8: Server-Tab mit Start/Stopp-Möglichkeit

angegebenen Verfahren. Wir hatten mit dem Eclipse-Update-Verfahren Erfolg, allerdings nur mit stabilen Releases sowohl bei Eclipse als auch bei den JBoss-Tools.

Nach der Installation der JBoss-Tools können JSF-Seiten mit dem JBoss-Tools-JSP-Editor geöffnet werden. Die Abbildung 10.9 zeigt Eclipse in der Web-Development-Perspektive. Der Editor in der Mitte ist der eben genannte JSP-Editor. Dieser ist trotz des irreführenden Namens auch für Facelets verwendbar. Man erkennt an der Unterseite des Editors drei Tabs, die für die Darstellung der JSF-Seite als Quell-Code, Vorschau oder im gemischten Modus dienen. Die Abbildung zeigt den gemischten Modus, bei dem in der oberen Hälfte der Quell-Code, in der unteren Hälfte die Vorschau dargestellt wird. Ebenfalls erwähnenswert ist die View *JBoss-Tools-Palette* ganz rechts, die neben den beiden Standard-JSF-Tag-Bibliotheken auch Tag-Bibliotheken von Ajax4JSF, RichFaces und Seam bereitstellen. Einzelne Tags können per Drag-and-Drop in den Editor gezogen werden.

Neben dieser Unterstützung auf der Komponentenebene bieten die JBoss-Tools eine Vervollständigungsfunktion für EL-Ausdrücke an. Damit können Properties und Methoden von Managed Beans in gewohnter Art und Weise mit wenig Tippaufwand verwendet werden. Die JBoss-Tools bieten eine Reihe weiterer Werkzeuge an, auf die wir aber nicht eingehen können. Der Leser möge sich selbst ein Bild von der Leistungsfähigkeit der JBoss-Tools machen.

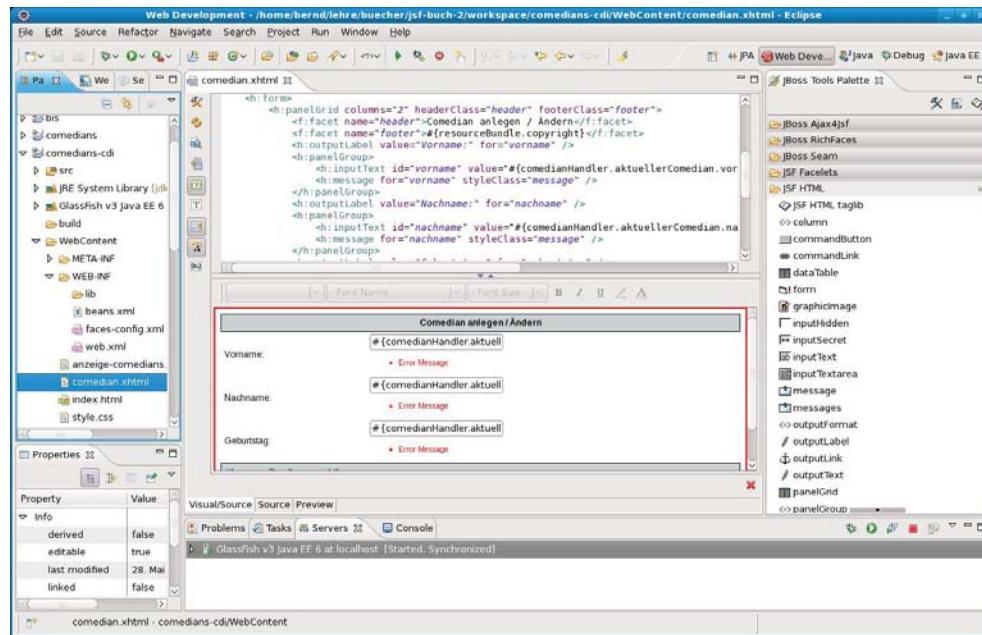


Abbildung 10.9: Die JBoss-Tools in Aktion

10.2.4 Projekte

Die im Buch entwickelten Projekte wurden als *Dynamic Web Project* angelegt. Einzige Ausnahme ist das in Abschnitt 9.1 verwendete EJB-Projekt. Unabhängig von der verwendeten Projektart importieren Sie die Projekte, indem Sie in der Package-View mit der rechten Maus das Kontextmenü öffnen und dann mit der Menüfolge *Import... ⇒ General ⇒ Existing Project into Workspace* das entsprechende Archiv auswählen.

10.3 Firebug

Obwohl JSF versucht, vom Servlet-API und dem zugrunde liegenden HTTP-Protokoll zu abstrahieren, ist die Kenntnis sowohl des APIs als auch des Protokolls bei der Entwicklung von JSF-Anwendungen hilfreich. Dies gilt vor allem bei der Fehlersuche und dem Aufspüren von Performanzproblemen in JSF-Anwendungen. Ein sehr wertvolles Werkzeug bei der Unterstützung dieser Tätigkeiten ist Firebug [URL-FB], ein Firefox-Plugin. Zusätzlich bietet Firebug die Möglichkeit der Analyse der von JSF generierten HTML-Seiten und die Darstellung und Manipulation von CSS. Insbesondere die Möglichkeiten der

Veränderung von CSS-Definitionen mit sofortigem Rendern im Firefox erspart häufige Edit-Deploy-Test-Zyklen.

Wir haben Firebug bereits in den Abschnitten 7.2.2 und 7.2.3 verwendet, wollen hier aber noch einmal kurz auf die Möglichkeiten von Firebug eingehen. Abbildung 10.10 zeigt die HTML-Darstellung einer JSF-Seite, die oben im Browser gerendert, unten im HTML-Quell-Code dargestellt wird. Die JSF-

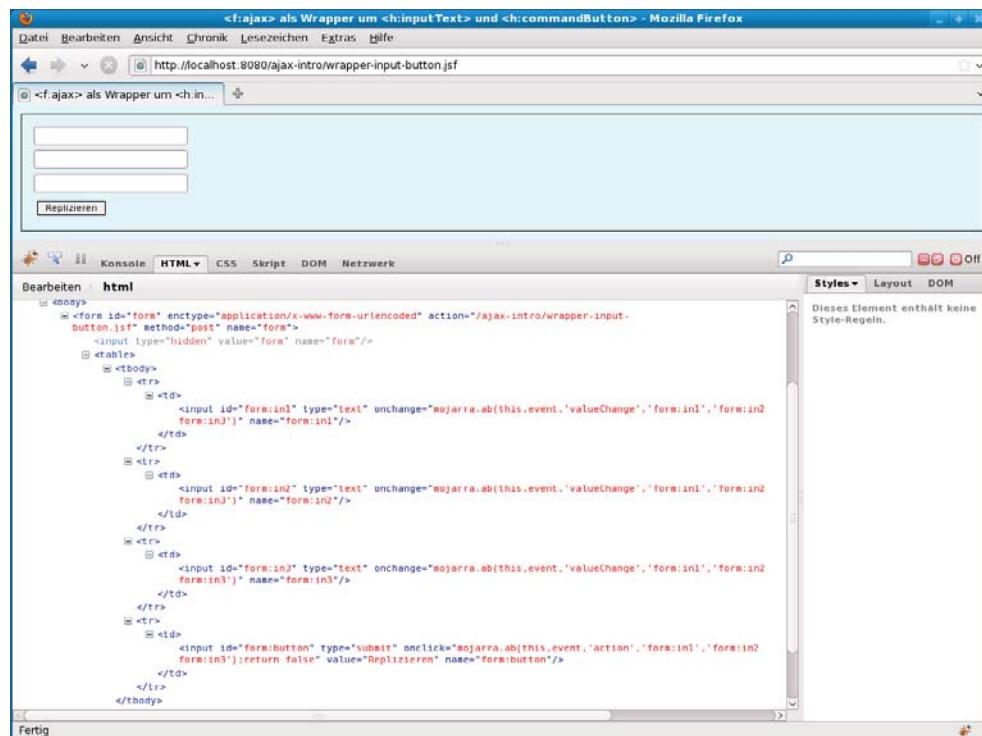


Abbildung 10.10: HTML-Darstellung in Firebug

Seite wurde im Abschnitt 7.2.2 auf Seite 243 beschrieben und besteht aus drei Texteingaben und einer Schaltfläche, die mit einem `<f:ajax>`-Tag umschlossen wurden. Man erkennt in der Abbildung, dass die drei Texteingaben über das `onchange`- und die Schaltfläche über das `onclick`-Event mit dem Server verbunden sind. Ebenfalls erkennbar sind die client-seitigen Ids, z. B. `form:in1`.

Eine weitere Verwendungsmöglichkeit von Firebug ist die Analyse des Netzwerkverkehrs. Abbildung 10.11 zeigt die Darstellung der Post-Tabs zweier Requests. Im ersten Request wurden nur Daten im HTML-Element mit der Id `form:in1` übertragen, während im zweiten Request auch Daten in den Elementen `form:in2` und `form:in3` übertragen wurden. Der erste Request wurde

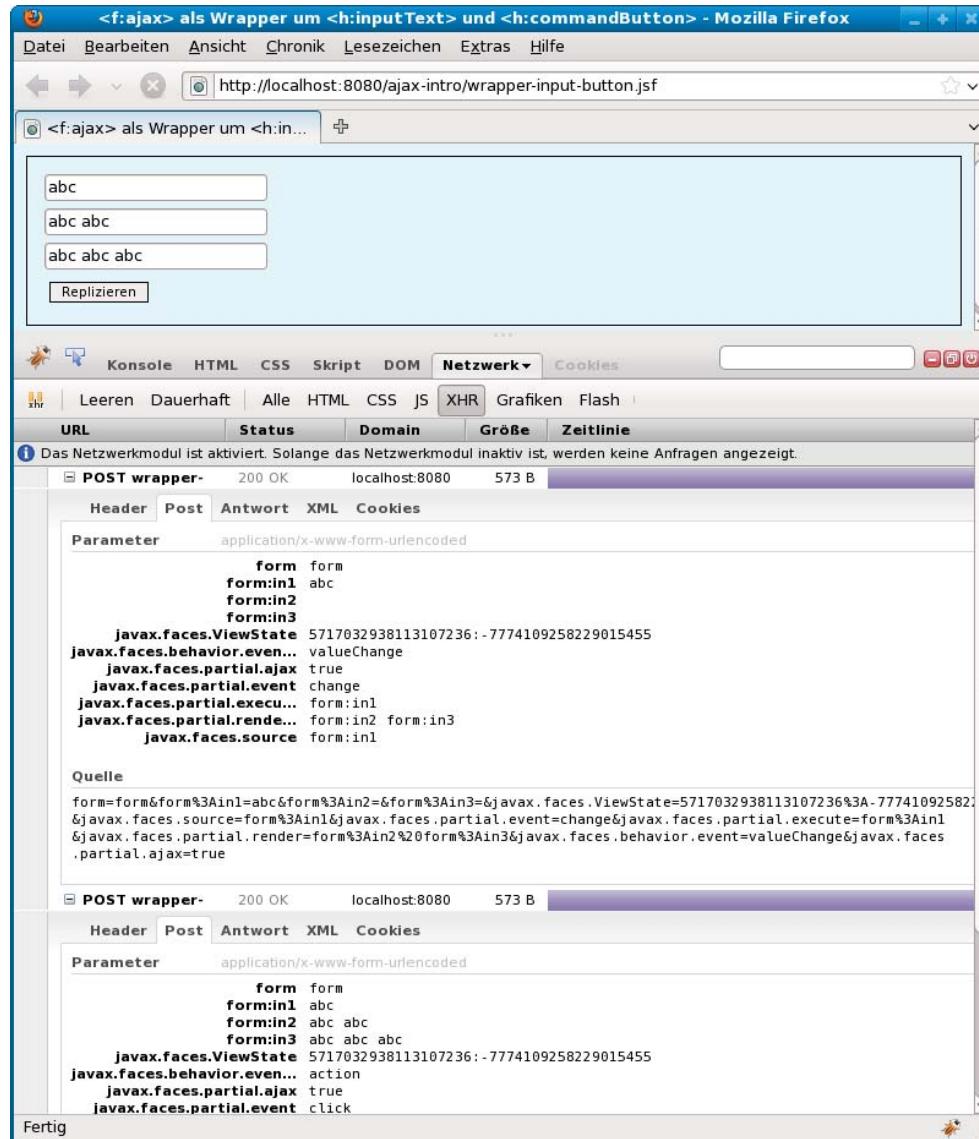


Abbildung 10.11: Post-Requests in Firebug

durch ein change-Event (Wert des Parameters `javax.faces.partial.event`), der zweite Request durch ein click-Event ausgelöst.

Wir können an dieser Stelle leider nicht ausführlicher auf die Möglichkeiten von Firebug eingehen, hoffen aber, dass wir den Leser motivieren konnten, sich eingehender mit Firebug zu beschäftigen. Die dargestellten Verwendungsmög-

lichkeiten lassen nur erahnen, welch große Hilfe Firebug bei der Erstellung von Web-Anwendungen, nicht nur auf Basis von JSF, sein kann.

10.4 Selenium

Selenium ist ein Werkzeugkasten zur Entwicklung von automatisierten Tests für Web-Anwendungen. Die wichtigsten Werkzeuge sind die Selenium-IDE und Selenium-RC (Remote Control), die wir beide vorstellen. Selenium-IDE ist ein Firefox-Plugin zur Entwicklung und Durchführung einzelner Tests oder Test-Suiten. Die erstellten Tests können als wiederverwendbare Skripte abgespeichert und somit später wiederholt werden. Durch einen Export in verschiedene Sprachen und die Programmierung von Tests in diesen Sprachen mit dem Selenium-API lassen sich umfangreichere Tests entwickeln. Zu den unterstützten Sprachen gehört neben Java HTML, C#, Perl, PHP, Python und Ruby. Obwohl die Selenium-IDE als Firefox-Plugin realisiert ist, können die erstellten Tests mit praktisch allen gängigen Browsern in den aktuellen Versionen durchgeführt werden. Selenium-RC kann in jeder entsprechenden IDE, auf der Kommandozeile oder im Build-Werkzeug verwendet werden.

10.4.1 Selenium-IDE

Die Selenium-IDE ist ein Firefox-Plugin und kann, wie alle anderen Selenium-Werkzeuge, unter [URL-SEL] heruntergeladen werden. Nach der Installation in Firefox öffnet sich durch die Menüfolge *Extras ⇒ Selenium IDE* die Selenium-IDE, wie in Abbildung 10.12 dargestellt. Man erkennt eine Schaltfläche mit symbolisiertem roten Punkt. Diese Schaltfläche wird verwendet, um den Aufnahmemodus der Selenium-IDE ein- und wieder auszuschalten. Beim Öffnen der IDE befindet sich die Selenium-IDE bereits im Aufnahmemodus, so dass alle Maus- und Tastaturaktivitäten erfasst und aufgezeichnet werden.

In einer speziell für den Abschnitt 10.5, *JSFUnit*, entwickelten Comedian-Anwendung finden Sie eine Menüseite, die Abbildung 10.13 zeigt. Diese Seite besitzt die Möglichkeit, Daten zu generieren und wieder zu löschen. Sie enthält außerdem den Link *Anzeige aller Comedians*, den wir betätigen. Die Anwendung navigiert nun zur Anzeige aller Comedians, wie in Abbildung 10.14 zu sehen ist. Im Beispiel sind noch keine Comedians vorhanden, so dass auch keine angezeigt werden können.

Wird ein Text auf der Seite selektiert (in Abbildung 10.14 der Text JSFPraxis.de), so kann kontextsensitiv mit der rechten Maustaste ein Menü geöffnet werden, das unter anderem den Befehl *verifyTextPresent* für die aktive Textse-

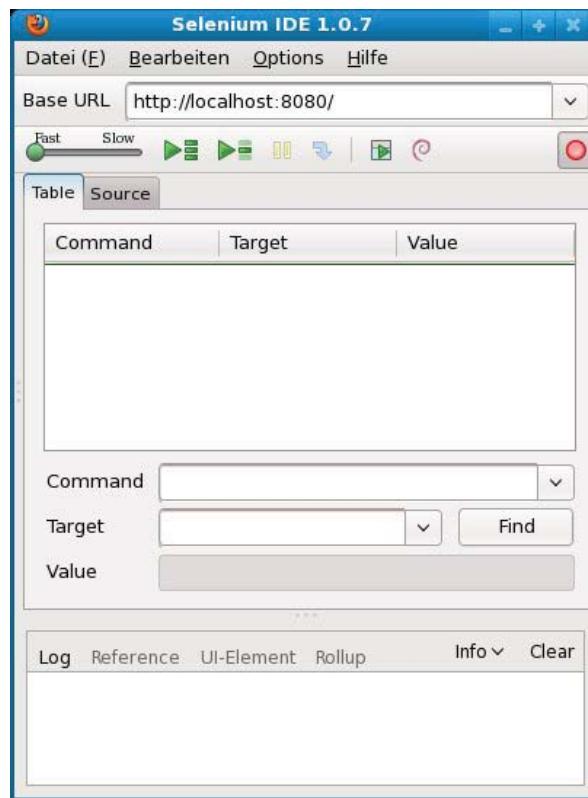


Abbildung 10.12: Selenium-IDE in initialem Zustand

lektion enthält. Wir wählen diesen Menüpunkt aus und erhalten den Zustand der Selenium-IDE, wie er in Abbildung 10.15 dargestellt ist. Der Befehl `verifyTextPresent` überprüft, ob der Parametertext *irgendwo* innerhalb der Seite vorhanden ist. Durch Betätigen des roten Knopfes wird der Aufnahmemodus der Selenium-IDE beendet.

Die Selenium-IDE in Abbildung 10.15 zeigt nun innerhalb des Table-Tabs die verwendeten Befehle. Der Befehl `open` wurde durch den initialen Klick auf den Link erzeugt und hat als Parameter die zu öffnende URL. Die Basis-URL ist ganz oben angegeben. Der Klick auf den Link ist jedoch nicht nur für die Aktivierung der Seite verantwortlich, sondern es muss auch die eigentliche Semantik des Klicks nachgebildet werden. Dies geschieht durch den Befehl `clickAndWait`, der in der zweiten Zeile zu erkennen ist. Schließlich erkennt man in der dritten Zeile den schon erwähnten Befehl `verifyTextPresent` mit entsprechendem Parameter.



Abbildung 10.13: Menüseite der Comedian-Anwendung für Tests



Abbildung 10.14: Anzeige aller Comedians

Dieser Test kann nun mit den beiden Pfeiltasten ausgeführt werden. Die linke Taste führt die komplette Test-Suite, die rechte nur den selektierten Test aus. Falls Sie den Test nachvollziehen wollen, müssen Sie den Schieberegler in Richtung **Slow** schieben, da der Test sonst zu schnell durchgeführt wird.

Nach diesem einführenden, sehr einfachen Test, der lediglich die Navigation auf eine andere Seite und das Vorhandensein eines Textes prüfte, soll nun ein etwas realistischerer Test entwickelt werden. Wir verzichten auf die Bildschirmfotos und beschreiben den Test verbal: Auf der Seite `/comedians-jsfunit/comedian.jsf` werden Vorname, Nachname und Geburtstag eingegeben und danach auf die Speichern-Schaltfläche gedrückt. Auf der dann neu dargestellten Übersicht aller Comedians ist der neu eingegebene Comedian vorhanden. Die Prüfung auf Vorhandensein eines Textes erfolgt diesmal über die Ids der HTML-Elemente. Abbildung 10.16 zeigt diesen Test in der Selenium-IDE.

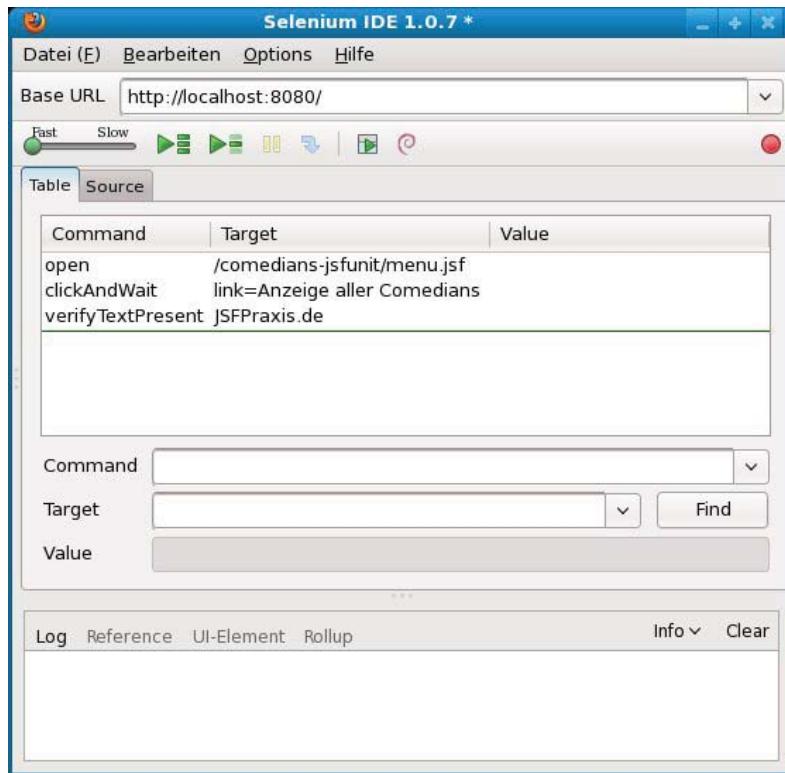


Abbildung 10.15: Selenium-IDE mit Test-Skript

Man erkennt in Abbildung 10.16 zwei neu hinzugekommene Befehle, `type` und `verifyText`. `type` steht für eine Benutzereingabe, `verifyText` für die Verifikation eines Textes. Beide Befehle benötigen ein UI-Element, das über XPath- oder DOM-Syntax spezifiziert werden kann. Im Beispiel erfolgt die Spezifikation über die DOM-Ids, wobei für die Eingaben der Container, in diesem Fall das Formular, keine explizite JSF-Id zugewiesen wurde, so dass JSF eine Id generiert. Für die Anzeige haben wir sowohl für das Formular als auch die Datentabelle (`<h:dataTable>`) Ids in JSF vergeben, was die Lesbarkeit erhöht. Wir gehen hier auf die Ids nicht weiter ein, holen das aber im Abschnitt 10.5 nach, wo wir denselben JSF-Code verwenden. Im unteren Teil von Abbildung 10.16 erkennt man das Logging eines durchgeführten Tests.

10.4.2 Selenium-RC

Selenium-RC erlaubt über die einfachen browser-basierten Aktionen der Selenium-IDE hinausgehende Testmöglichkeiten. Zu diesen gehören z. B. bedingte Verzweigungen und Iterationen oder etwa die Wiederholung fehlgeschlagener

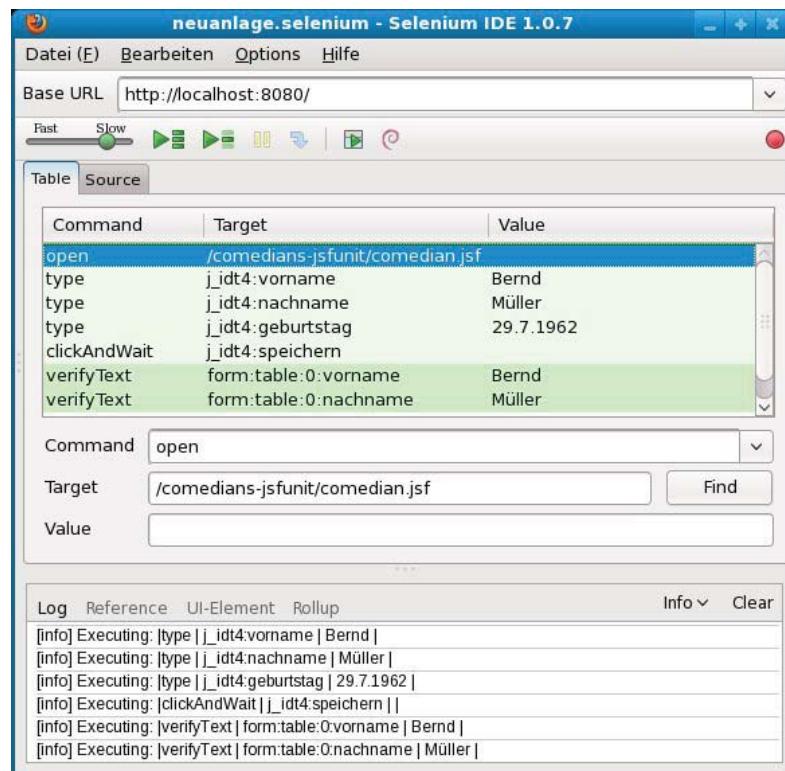


Abbildung 10.16: Die Selenium-IDE mit geladenem Test

Tests. Selenium-RC besteht zum einen aus einem Server, der einen Browser startet und beendet und als Proxy zwischen dem Browser und der Web-Anwendung HTTP-Nachrichten verifiziert. Zum anderen besteht Selenium-RC aus Client-Bibliotheken zur Erstellung der Tests. Als Basis der Definition von Tests dient die Sprache *Selenese*, die Dialekte für HTML, Java, C#, Perl, PHP, Python und Ruby besitzt. Der in Abbildung 10.16 im Table-Tab dargestellte Test-Code entspricht dem HTML-Dialekt. Im Source-Tab wird der entsprechende HTML-Quell-Code angezeigt, den wir zu Dokumentationszwecken für die ersten drei Selenese-Befehle angeben:

```
<html xmlns="...">
<link rel="selenium.base" href="http://localhost:8080/" />
...
<body>
<table ... >
...
<tr>
<td>open</td><td>/comedians-jsfunit/comedian.jsf</td>
<td></td>
```

```
</tr>
<tr>
    <td>type</td><td>j_idt4:vorname</td><td>Bernd</td>
</tr>
<tr>
    <td>type</td><td>j_idt4:nachname</td><td>Müller</td>
</tr>
...
```

Für JSF-Entwickler ist der Java-Dialekt interessanter. Für den Test aus Abbildung 10.16 kann mit der Menüfolge *Datei* ⇒ *Export Test Case As ...* ⇒ *Java* die Klasse **Neuanlage** erzeugt werden, die in Listing 10.2 wiedergegeben ist. Beim Export nach Java kann zwischen den Alternativen JUnit und TestNG gewählt werden. Die Klasse **Neuanlage** wurde als JUnit-Test erzeugt.

Listing 10.2: Selenium-Test als Klasse Neuanlage

```
public class Neuanlage extends SeleneseTestCase {

    public void setUp() throws Exception {
        setUp("http://localhost:8080/", "*firefox");
    }

    public void testNeuanlage() throws Exception {
        selenium.open("/comedians-jsfunit/comedian.jsf");
        selenium.type("j_idt4:vorname", "Bernd");
        selenium.type("j_idt4:nachname", "Müller");
        selenium.type("j_idt4:geburtstag", "29.7.1962");
        selenium.click("j_idt4:speichern");
        selenium.waitForPageToLoad("30000");
        assertEquals("Bernd",
                     selenium.getText("form:table:0:vorname"));
        assertEquals("Müller",
                     selenium.getText("form:table:0:nachname"));
    }
}
```

Man erkennt im Rumpf der Methode **testNeuanlage()** Methodennamen, die identisch oder ähnlich zu den in Abbildung 10.16 und im HTML-Code erkennbaren Befehlsnamen sind. Um den Test durchzuführen, muss der Selenium-RC-Server mit `java -jar selenium-server.jar` gestartet werden. Dann wird die Java-Klasse als JUnit- bzw. TestNG-Test durchgeführt. Bei der Testdurchführung wird Firefox (Methode `setUp()`) oder einer der anderen unterstützten Browser gestartet und die den Selenese-Befehlen entsprechenden JavaScript-

Methoden ausgeführt. Die Testdurchführung kann in Eclipse, auf der Kommandozeile oder, im Build-Prozess integriert, automatisch erfolgen.

Selenium besitzt keine eigenen Werkzeuge für die Berichterstattung. Bei der Verwendung von Java können die von JUnit bzw. TestNG bereitgestellten Werkzeuge verwendet werden.

Wir konnten die Leistungsfähigkeit von Selenium hier nur ansatzweise darstellen und raten dem interessierten Leser, sich intensiv mit Selenium zu beschäftigen, falls Sie auf der Suche nach einem mächtigen Testwerkzeug für Web-Anwendungen sind.



Projekt

Die Tests für die Selenium-IDE als auch für Selenium-RC sind im Projekt *selenium* enthalten.

Aufgabe 10.2

Starten Sie GlassFish, und deployen Sie die Anwendung `comedian-jsfunit`. Installieren Sie die Selenium-IDE in Firefox, und öffnen Sie die IDE. Laden Sie den Test in der Datei `neuanlage.selenium` aus dem Projekt `selenium` in die IDE, und führen Sie den Test aus.

Aufgabe 10.3

Starten Sie GlassFish, und deployen Sie die Anwendung `comedian-jsfunit`. Starten Sie den Selenium-RC-Server über das Ant-Build-File im Projekt `selenium`. Führen Sie die Klasse `Neuanlage` (Listing 10.2) als JUnit-Test innerhalb von Eclipse aus. Führen Sie nun alternativ die Klasse `Neuanlage` über das Ant-Build-File aus. Überprüfen Sie den erzeugten Test-Report.

10.5 JSFUnit

Während das im letzten Abschnitt vorgestellte Selenium ein reines Black-Box-Testwerkzeug ist, können mit JSFUnit [URL-JSFUNIT] Black-Box-Tests, aber auch sogenannte Acryl-Box-Tests durchgeführt werden. Die JSFUnit-Entwickler verstehen darunter Tests, die die Verwendung von client-seitigem HTML und server-seitigem Zustand in einem einzigen Test erlauben. Der server-seitige Zustand wird dabei vom laufenden Container erfragt und nicht, wie häufig üblich, über Mocks simuliert.

JSFUnit-Tests werden in der Regel durch ein `JSFSession`-Objekt gestartet. Dieses repräsentiert einen realen HTTP-Request an eine anwendungsrelative URL. Über dieses Objekt lassen sich Repräsentanten des Clients (Klasse

JSFClientSession) und des Servers (Klasse JSFServerSession) erzeugen, über die dann die eigentlichen Testbedingungen verifiziert werden. Für unsere Beispiele verwenden wir die Comedians-Anwendung, haben jedoch an einigen Stellen explizite Ids vergeben, um auf die einzelnen Komponenten zugreifen zu können. Wir beginnen mit dem Test einer einfachen Navigation.

```
public void testNavigation() throws IOException {
    JSFSession jsfSession = new JSFSession("/comedian.jsf");
    JSFClientSession client = jsfSession.getJSFClientSession();
    JSFServerSession server = jsfSession.getJSFServerSession();
    client.setValue("vorname", "Bernd");
    client.setValue("nachname", "Müller");
    client.setValue("geburtstag", "29.7.1962");
    client.click("speichern");
    assertEquals("/anzeige-comedians.xhtml",
                server.getCurrentViewID());
}
```

Nachdem die Initialisierungsarbeiten durchgeführt sind, wird im Client (der HTML-Seite) im Element vorname der Wert Bernd eingesetzt. Die JSF-Seite muss daher eine Texteingabe mit entsprechender Id (<h:inputText id="vorname" ... />) enthalten. Nach zwei weiteren Eingaben wird das Formular mit einem Klick auf das Element mit der Id speichern (<h:commandButton id="speichern" ... />) abgeschickt. Zum Schluss wird geprüft, ob serverseitig richtig navigiert wurde, d. h. ob die aktuelle View-Id korrekt ist.

Der nächste Test überprüft, ob auf einen Fehlerfall, hier: auf eine vergessene Eingabe, richtig reagiert wird.

```
public void testErrorMessage() throws IOException {
    boolean messageFound = false;
    JSFSession jsfSession = new JSFSession("/comedian.jsf");
    JSFClientSession client = jsfSession.getJSFClientSession();
    JSFServerSession server = jsfSession.getJSFServerSession();
    // Vorname fehlt. Meldung: "Bitte eingeben"
    client.setValue("nachname", "Müller");
    client.setValue("geburtstag", "29.7.1962");
    client.click("speichern");
    for (Iterator<FacesMessage> iterator =
         server.getFacesMessages("vorname");
         iterator.hasNext();) {
        FacesMessage message = (FacesMessage) iterator.next();
        if (message.getDetail().equals("Bitte eingeben")) {
            messageFound = true;
            break;
        }
    }
    assertTrue(messageFound);
}
```

Da es mehrere Faces-Messages für eine JSF-Komponente geben kann, muss über die Menge dieser Meldungen iteriert werden, um sicherzustellen, dass die benötigte Meldung existiert.

JSFUnit erlaubt den vollständigen Zugriff auf den Zustand der JSF-Anwendung auf dem Server, und somit insbesondere auf Managed Beans. Im nächsten Beispiel soll nochmals ein Comedian angelegt und danach geprüft werden, ob die Anzahl der Comedians sich um eins erhöht hat.

```
public void testNeuanlage() throws IOException {
    JSFSession jsfSession = new JSFSession("/comedian.jsf");
    JSFClientSession client = jsfSession.getJSFClientSession();
    JSFServerSession server = jsfSession.getJSFServerSession();
    int anzahlVor = (Integer) server.getManagedBeanValue(
        "#{comedianHandler.comedians.rowCount}");
    client.setValue("vorname", "Bernd");
    client.setValue("nachname", "Müller");
    client.setValue("geburtstag", "29.7.1962");
    client.click("speichern");
    int anzahlNach = (Integer) server.getManagedBeanValue(
        "#{comedianHandler.comedians.rowCount}");
    assertEquals(1, anzahlNach - anzahlVor);
}
```

Der Code dieses Tests erweitert die bereits bekannten Tests um den Zugriff auf die Managed Bean `comedianHandler` über die Methode `getManagedBeanValue()`, die als Parameter einen EL-Ausdruck erwartet. Im Beispiel wird über die Methode `getComedians()` ein DataModel zurückgegeben. Das API dieser Klasse enthält die Methode `getRowCount()`, die die Anzahl der enthaltenen Datenzeilen zurückgibt. Wird nach dem Speichern des Comedians diese Anzahl wiederholt berechnet, muss die Differenz der beiden Anzahlen 1 ergeben.

Im letzten Test soll von der Anzeige aller Comedians durch Selektion eines Comedians auf die Ändern-Seite navigiert werden. Der Test soll weiter prüfen, dass der in der Übersichtsseite selektierte Comedian tatsächlich angezeigt wird. Problematisch an diesem Test ist die Betätigung einer Schaltfläche innerhalb einer `<h:dataTable>`. Um eine bestimmte Tabellenzeile in einem Test verwenden zu können, müssen die Element-Ids vollständig sein und dürfen nicht automatisch generiert werden. Wir erweitern daher den Code der JSF-Seite zur Anzeige aller Comedians um die entsprechenden Ids. Der folgende Code-Ausschnitt zeigt die wichtigen Stellen der JSF-Seite.

```
<body>
<h:form id="form">
    <h:dataTable id="table" var="comedian"
        value="#{comedianHandler.comedians}" ... >
        <h:column>
```

```
<f:facet name="header">Vorname</f:facet>
<h:outputText id="vorname"
               value="#{comedian.vorname}" />
</h:column>
...
<h:column>
    <h:commandButton id="aendern"
                     action="#{comedianHandler.aendern}" .../>
</h:column>
...
```

Man erkennt die Ids `form`, `table`, `vorname` und `aendern`. Die einzelnen Tabellezeilen werden fortlaufend nummeriert. Die Nummerierung beginnt bei 0. Die einzelnen Ids werden nach den in Abschnitt 4.9 erläuterten Regeln mit Doppelpunkt getrennt zur Gesamt-Id verbunden, so dass die Id des Vornamens der ersten Zeile „`form:table:0:vorname`“ und der Schaltfläche „`form:table:0:aendern`“ lautet. Der Zugriff auf die Zeile erfolgt wiederum über `getManagedBeanValue()`, diesmal jedoch über die Methode `getWrappedData()` der Klasse `DataModel`.

```
public void testDataTable() throws IOException {
    JSFSession jsfSession =
        new JSFSession("/anzeige-comedians.jsf");
    JSFClientSession client = jsfSession.getJSFClientSession();
    JSFServerSession server = jsfSession.getJSFServerSession();
    List<Comedian> comedians =
        (List<Comedian>) server.getManagedBeanValue(
            "#{comedianHandler.comedians.wrappedData}");
    if (comedians.size() > 0) {
        Comedian comedian = comedians.get(0);
        client.click("form:table:0:aendern");
        assertEquals("/comedian.xhtml", server.getCurrentViewID());
        HtmlTextInput vorname =
            (HtmlTextInput) client.getElement("vorname");
        assertEquals(comedian.getVorname(), vorname.getText());
    }
}
```

Das erste `assertEquals()` prüft die korrekte Navigation, das zweite, ob der selektierte Comedian tatsächlich angezeigt wird. Wir beschränken uns hier auf die Prüfung des Vornamens. Dem aufmerksamen Leser wird aufgefallen sein, dass die Verwendung des Index 0 sowohl in der `get()`-Methode der `Comedian`-Liste als auch in der Id der Schaltfläche als Literal erfolgt. Eine schönere Lösung bleibt dem Leser als Übung überlassen.

Erwähnenswert beim letzten Test ist die Verwendung der Klasse `HtmlTextInput`. Dies ist kein Schreibfehler und keine Verwechslung mit der JSF-Kompo-

nentenklasse `HtmlInputText`. `HtmlTextInput` ist eine Klasse des `HtmlUnit`-Frameworks [URL-HTMLUNIT], das von JSFUnit verwendet wird, um auf Elemente einer HTML-Seite zuzugreifen.

Nach der Definition der vier Tests stellt sich die Frage, wie die Tests ausgeführt werden. Eine Ausführung innerhalb von Eclipse oder innerhalb eines Build-Prozesses ist nicht direkt möglich, da ein Teil des Tests innerhalb des Containers ausgeführt wird. JSFUnit wird mit einem Servlet ausgeliefert, so dass die Tests browser-basiert durchgeführt werden können. Die Abbildung 10.17 auf der nächsten Seite zeigt die durch das Servlet erstellte Ergebnisseite für unsere vier Tests. Man erkennt in der URL-Leiste, dass an die Anwendungs-URL noch

```
ServletTestRunner?suite=de.jsfpraxis.ComedianHandlerTest
```

angehängt wurde. Die Klasse `ComedianHandlerTest` enthält die vier Tests. Die URL ist nicht vollständig dargestellt, sondern endet mit der Angabe eines XSL-Stylesheets, mit dem die Testergebnisse in das in Abbildung 10.17 dargestellte Format überführt werden. Wir haben dies hier als unerhebliches Detail unterschlagen. Das herunterladbare Projekt enthält die vollständige Version in Form eines Links.

Die JSFUnit-Homepage nennt das Anhängen von `/jsfunit` an die Anwendungs-URL in einer Servlet-3.0-Umgebung als ausreichend, um die Tests auszuführen. Bei uns führte dies leider nicht zum Erfolg.

Bemerkung: Das von uns oben angesprochene Problem der Testdurchführung innerhalb Eclipse oder eines Build-Prozesses ist nicht real. JSFUnit verwendet Cactus [URL-CACTUS] als server-seitiges Test-Framework, so dass alle Möglichkeiten von Cactus zur Durchführung von Tests zur Verfügung stehen. Außerdem enthält JSFUnit eine Ant-Task, und die Homepage von JSFUnit beschreibt, wie JSFUnit aus Eclipse heraus aufgerufen werden kann.

Wir konnten hier JSFUnit nur oberflächlich einführen. JSFUnit ist ein sehr mächtiges Werkzeug, das z. B. auch JSF-Anwendungen auf der Basis von Rich-Faces testen kann. Zusätzlich stehen statische Analysen bereit, die z. B. prüfen, ob Managed Beans in der JSF-Konfigurationsdatei mehrfach definiert werden oder ob Managed Beans mit Session-Scope serialisierbar sind. Wir können hier nicht darauf eingehen und verweisen auf die Originaldokumentation [URL-JSFUNIT].



Projekt

Der Code dieses Abschnitts ist im Projekt *comedians-jsfunit* enthalten.

The screenshot shows a Mozilla Firefox browser window displaying the JSFUnit test results for the suite `de.jsfpraxis.ComedianHandlerTest`. The URL in the address bar is `http://localhost:8080/comedians-jsfunit/ServletTestRunner?suite=de.jsfpraxis.ComedianHandlerTest`. The page title is "Unit Test Results". A note at the top right states "Designed for use with Cactus".

Summary

Tests	Failures	Errors	Success rate	Time
4	0	0	100.00%	0.455

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

TestCase de.jsfpraxis.ComedianHandlerTest

Name	Status	Type	Time(s)
testNeuanlage	Success		0.112
testNavigation	Success		0.116
testErrorMessage	Success		0.075
testDataTable	Success		0.151

[Back to top](#)

Fertig

Abbildung 10.17: JSFUnit im Browser

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Anhang A

Annotationen

In diesem Anhang werden die Annotationen von JSF 2.0 beschrieben. Wir folgen dabei dem im API-Doc eingeschlagenen Weg und unterteilen die Annotationen in einen allgemeinen Teil (Abschnitt A.1) und Annotationen, die für die Deklaration von Managed Beans (Abschnitt A.2) bereitstehen.

Die Anwendbarkeit der Annotation für Klassen, Properties oder Methoden geht aus der Beschreibung hervor. Mit einem Stern gekennzeichnete Attribute sind verpflichtend anzugeben.

A.1 JSF-Annotationen

JSF bietet eine Reihe von Annotationen an, die in den verschiedensten Teilbereichen von JSF Verwendung finden. Einige werden bei der Definition eigener, nativer Komponenten eingesetzt. Da die Entwicklung nativer Komponenten, die nicht zu verwechseln ist mit der Entwicklung eigener Komponenten auf der Basis von Facelets, nicht Thema dieses Buches ist, führen wir diese Annotationen der Vollständigkeit halber auf, ohne sie detailliert zu beschreiben. Es sind dies die Annotationen in Tabelle A.1.

Es bleiben lediglich zwei Annotationen übrig, die nichts mit der Definition von Komponenten zu tun haben. Zum einen ist dies die Annotation `@FacesConverter` im Package `javax.faces.converter` und zum anderen die Annotation `@FacesValidator` im Package `javax.faces.validator`, deren Syntax im Folgenden angegeben ist.

@FacesConverter(...)			
Deklariert eine Klasse als Konvertierer.			
Attribut	Typ	Default	Beschreibung
value	String	—	Definition der Id dieses Konvertierers
forClass	Class	Object	Klasse, für deren Konvertierung der Konvertierer zuständig ist

Die mit **@FacesConverter** annotierte Klasse, die das Interface **Converter** implementieren muss, wird vom JSF-Laufzeitsystem als Konvertierer registriert. Das Attribut **value** definiert die Konvertierer-Id und entspricht damit dem **<converter-id>**-Tag in der JSF-Konfigurationsdatei. Das Attribut **forClass** benennt die Klasse, für die der Konvertierer global registriert wird und entspricht damit dem **<converter-for-class>**-Tag.

Beispiele: Die Klasse **KreditkartennummerConverter** auf Seite 86 als global definierter Konvertierer und dieselbe Klasse auf Seite 87, verwendbar über die Konvertierer-Id.

@FacesValidator(...)			
Deklariert eine Klasse als Validierer.			
Attribut	Typ	Default	Beschreibung
value	String	—	Definition der Id dieses Validierers

Die mit **@FacesValidator** annotierte Klasse, die das Interface **Validator** implementieren muss, wird vom JSF-Laufzeitsystem als Validierer registriert. Das Attribut **value** definiert die Validierer-Id und entspricht somit dem **<validator-id>**-Tag in der JSF-Konfigurationsdatei.

Beispiel: Die Klasse **EmailValidator** auf Seite 94.

Tabelle A.1: JSF-Annotationen zur Komponentenentwicklung

Annotation	Beschreibung
Package javax.faces.application	
@ResourceDependency	Die annotierte Unterklasse von UIComponent oder Renderer erhält eine Ressourcen-Abhängigkeit, so dass die genannte Ressource innerhalb der Seite verwendet werden kann.
@ResourceDependencies	Fasst mehrere @ResourceDependency-Annotationen in Array-Notation zusammen.
Package javax.faces.component	
@FacesComponent	Die annotierte Unterklasse von UIComponent wird beim JSF-Laufzeitsystem als JSF-Komponente registriert.
Package javax.faces.component.behavior	
@FacesBehavior	Die annotierte Unterklasse von BehaviorBase wird beim JSF-Laufzeitsystem als Verhalten (Interface Behavior) registriert.
Package javax.faces.event	
@ListenerFor	Die annotierte Unterklasse von UIComponent oder Renderer wird als Listener für das als Attribut angegebene System-Event registriert.
@ListenersFor	Fasst mehrere @ListenerFor-Annotationen in Array-Notation zusammen.
@NamedEvent	Die annotierte Unterklasse von ComponentSystemEvent wird als System-Event registriert, so dass sie im <f:event>-Tag verwendet werden kann.
Package javax.faces.render	
@FacesRenderer	Die annotierte Unterklasse von Renderer wird als Renderer registriert.
@FacesBehaviorRenderer	Die annotierte Unterklasse von ClientBehaviorRenderer wird als Renderer für Verhalten registriert.

A.2 Annotationen für Managed Beans

Die in JSF definierten Annotationen für Managed Beans, die wir in diesem Abschnitt vorstellen, sind im Package `javax.faces.bean` enthalten. Das Package besteht nur aus diesen Annotationen und enthält keine weiteren Klassen. Dies ist durch die bereits in Abschnitt 9.5.3 angesprochene Problematik redundanter Annotationen für Managed Beans begründet, da sich drei JSRs der Definition von Managed Beans angenommen haben.

Falls sich nun etwa der CDI-Ansatz zur Definition von Managed Beans mittel- und langfristig durchsetzen würde, wären die JSF-eigenen Annotationen als *deprecated* zu kennzeichnen. Um diesen Makel in zukünftigen JSF-Spezifikationen zu vermeiden, wurden die Annotationen bzw. das Package als optional gekennzeichnet. Die Annotationen müssen also in einer JSF-Implementierung nicht vorhanden sein. Sowohl Mojarra als auch MyFaces enthalten jedoch den vollständigen Satz von Annotationen, den wir nun vorstellen.

`@ApplicationScoped`

Deklariert den Application-Scope für eine Managed Bean.

Die `@ApplicationScoped`-Annotation ist nur in Kombination mit der `@ManagedBean`-Annotation erlaubt. Sie deklariert die Anwendung als Lebensdauer der Managed Bean und entspricht somit der Verwendung von `<managed-bean-scope>application</managed-bean-scope>` in der JSF-Konfigurationsdatei.

Beispiel: Die Klasse `DBInit` in der Online-Banking-Anwendung in Kapitel 8. Die Klasse ist im Buch nicht abgebildet, lässt sich aber von der Web-Site des Buches im Projekt *banking* herunterladen. Weitere Verwendungen finden sich im Projekt *jsf-im-detail*, das ebenfalls heruntergeladen werden kann.

`@CustomScoped(...)`

Deklariert einen Custom-Scope für eine Managed Bean.

Attribut	Typ	Default	Beschreibung
<code>value</code>	<code>String</code>	—	Die Map des Custom-Scopes als EL-Ausdruck

Die `@CustomScoped`-Annotation ist nur in Kombination mit der `@ManagedBean`-Annotation erlaubt. Sie deklariert den Scope der Managed Bean über Schlüssel/Wert-Paare einer Map und entspricht somit der Verwendung von

<managed-bean-scope> EL-Ausdruck </managed-bean-scope> in der JSF-Konfigurationsdatei.

Wir verwenden den Custom-Scope im Buch nicht, da er keinen Scope im Sinne der automatischen Verwaltung durch das JSF-Laufzeitsystem darstellt. Der Entwickler ist dafür verantwortlich, dass Referenzen der Map entsprechend den Scopes valide sind. Die Klasse, die das Objekt im Custom-Scope verwaltet, ist verantwortlich dafür, dass die Events `PreDestroyCustomScopeEvent` und `PostConstructCustomScopeEvent` an den entsprechenden Stellen geworfen werden.

@ManagedBean(...)

Deklariert ein POJO als Managed Bean.

Attribut	Typ	Default	Beschreibung
eager	boolean	false	Zeitpunkt der Erzeugung für Application-Scope
name	String	Klassenname	Name der Managed Bean

Die `@ManagedBean`-Annotation registriert die annotierte Klasse beim JSF-Laufzeitsystem als Managed Bean entsprechend der <managed-bean>-Deklaration in der JSF-Konfigurationsdatei. Als Default-Name wird der Klassennname mit kleinem Anfangsbuchstaben verwendet. Ist keine Scope-Annotation vorhanden, wird `@RequestScoped` als Default verwendet. Ist das `eager`-Attribut `true`, wird die Managed Bean mit verwandelter `@ApplicationScoped`-Annotation beim Start der Anwendung instanziert, sonst bei der ersten Verwendung.

Beispiele finden sich über das ganze Buch verteilt, z. B. in Listing 8.6 auf Seite 294 oder in Listing 8.10 auf Seite 305.

@ManagedProperty(...)

Injiziert Wert in ein Property einer Managed Bean.

Attribut	Typ	Default	Beschreibung
value*	String	—	Der zu injizierende Wert
name	String	—	Alternativer Property-Name

Die `@ManagedProperty` veranlasst das JSF-Laufzeitsystem, den als EL-Ausdruck angegebenen Wert in das annotierte Property zu injizieren, und entspricht somit dem <managed-property>-Element in der JSF-Konfigura-

tionsdatei. Falls die Klasse nicht mit `@ManagedBean` annotiert ist, hat die `@ManagedProperty` keine Auswirkungen.

Beispiele: Die Klasse `EventBean` auf Seite 266 und Listing 8.10 auf Seite 305.

@NoneScopd

Deklariert den None-Scope für eine Managed Bean.

Die `@NoneScopd`-Annotation ist nur in Kombination mit der `@ManagedBean`-Annotation erlaubt. Sie deklariert die Managed Bean als zu keinem Scope zugehörig. Die Managed Bean kann daher nur zur Initialisierung anderer Managed Beans verwendet werden. Die Verwendung von `@NoneScopd` entspricht `<managed-bean-scope>none</managed-bean-scope>` in der JSF-Konfigurationsdatei.

Beispiel: Listing 4.7 auf Seite 70.

@ReferencedBean(...)

Deklariert ein POJO als referenzierte Bean.

Attribut	Typ	Default	Beschreibung
<code>name</code>	<code>String</code>	Klassenname	Name der referenzierten Bean

Eine referenzierte Bean ist eine Bean, die nicht durch das JSF-Laufzeitsystem, sondern anderweitig erzeugt wird. Als Default-Name wird der Klassename mit kleinem Anfangsbuchstaben verwendet.

@RequestScopd

Deklariert den Request-Scope für eine Managed Bean.

Die `@RequestScopd`-Annotation ist nur in Kombination mit der `@ManagedBean`-Annotation erlaubt. Sie deklariert den Request als Lebensdauer der Managed Bean und entspricht somit der Verwendung von `<managed-bean-scope>request</managed-bean-scope>` in der JSF-Konfigurationsdatei. Falls keine Scope-Annotation für eine Managed Bean verwendet wird, gilt der Request-Scope als Default.

Beispiel: Die Klasse `EnumHandler` auf Seite 82 und die Klasse `EventBean` auf Seite 266.

@SessionScoped

Deklariert den Session-Scope für eine Managed Bean.

Die `@SessionScoped`-Annotation ist nur in Kombination mit der `@ManagedBean`-Annotation erlaubt. Sie deklariert die Session als Lebensdauer der Managed Bean und entspricht somit der Verwendung von `<managed-bean-scope>session</managed-bean-scope>` in der JSF-Konfigurationsdatei.

Beispiele: Listing 2.3 auf Seite 16 und Listing 3.2 auf Seite 25.

@ViewScoped

Deklariert den View-Scope für eine Managed Bean.

Die `@ViewScoped`-Annotation ist nur in Kombination mit der `@ManagedBean`-Annotation erlaubt. Sie deklariert die View als Lebensdauer der Managed Bean und entspricht somit der Verwendung von `<managed-bean-scope>view</managed-bean-scope>` in der JSF-Konfigurationsdatei.

Beispiel: Die Klasse `MBViewScopeHandler` auf Seite 63, deren Scope alternativ über die JSF-Konfigurationsdatei deklariert wurde.

Anhang B

Die Tags der Standardbibliotheken

Dieser Anhang beschreibt die Tags von drei der insgesamt sechs JSF-Standard-Bibliotheken. Dies sind die HTML-Bibliothek und die Kernbibliothek, die vor der Version 2.0 von JSF die einzigen Bibliotheken waren, sowie die Facelets-Bibliothek. Wir beschreiben nicht die Composite-Bibliothek sowie die beiden JSTL-Bibliotheken für den JSTL-Kern und für JSTL-Funktionen. Wir verzichten auf die Composite-Bibliothek, da deren Darstellung den Umfang des Buches sprengen würde, und auf die beiden JSTL-Bibliotheken, weil ihre Verwendung mit JSF 2.0 in der Regel nicht nötig ist und – insbesondere – von uns nicht empfohlen wird. Einen kurzen Überblick über diese drei hier nicht beschriebenen Bibliotheken finden Sie in den Abschnitten 5.7 und 5.8.

Die Bibliotheken werden über XML-Namensräume wie folgt definiert.

Bibliothek	URI	Präfix
HTML	http://java.sun.com/jsf/html	h
Kern	http://java.sun.com/jsf/core	f
Facelets	http://java.sun.com/jsf/facelets	ui
Composite	http://java.sun.com/jsf/composite	composite
JSTL-Kern	http://java.sun.com/jsp/jstl/core	c
JSTL-Funktionen	http://java.sun.com/jsp/jstl/functions	fn

Die Präfixe sind frei wählbar. Die angegebenen Präfixe entsprechen jedoch verbreiteten Konventionen und sollten verwendet werden. Statt `composite` kann alternativ auch `cc` für *Composite Components* genutzt werden.

Das von JSF erzeugte HTML ist XHTML 1.0. Dies bedeutet, dass ältere Browser von JavaServer Faces erzeugte HTML-Seiten möglicherweise nicht oder nicht korrekt darstellen. Man kann allerdings davon ausgehen, dass der Anteil dieser Browser sehr gering ist und sich weiter verringern wird.

B.1 HTML-Attribute

Die Tags der HTML-Bibliothek haben direkt an die erzeugten HTML-Tags durchgereichte Attribute. Man spricht auch von *pass-through* Attributen. Um diese sinnvoll einzusetzen, benötigt man entsprechende HTML-Kenntnisse. Man muss wissen, welche Attribute welche möglichen Werte zulassen und was diese bewirken, außerdem, durch welche JSF-Tags ein bestimmtes HTML-Tag erzeugt wird. In diesem Abschnitt führen wir die statischen Attribute auf. Unter „statisch“ verstehen wir in diesem Fall „nicht JavaScript-basiert“. Den JavaScript-basierten Attributen ist Abschnitt B.2 gewidmet.

Wir beschreiben die Attribute aus der Sicht von JavaServer Faces, d.h. ausschließlich in der Verwendung durch JSF-Tags. Beispielsweise ist die Verwendung des `height`-Attributs in mehreren HTML-Tags möglich, bei der Verwendung in JavaServer Faces aber nur im ``-Tag (durch `<h:graphicImage>`). Wir beschreiben daher `height` lediglich als ``-Attribut. Eine gute Einführung in HTML findet man in dem Buch von Muscioano und Kennedy [MK00].

Alle Attribute dieses und des nächsten Abschnitts sind optional. Ihnen können Konstanten oder JSF-EL-Ausdrücke zugewiesen werden.

Tabelle B.1: Unterstützte HTML-Attribute

Attribut	Beschreibung
accept	Liste von MIME-Typen zur Definition selektierbarer Dateiarten in einem Formular.
acceptCharset	Angabe der Zeichen-Codierungen, mit denen der Browser die Formulardaten zum Server schickt. Wird in HTML zu <code>accept-charset</code> . Beispiel: <code>acceptCharset="ISO-8859-1 ISO-8859-2"</code> .
accesskey	Definition eines Tastaturkürzels (ein Zeichen), um ein Formularelement direkt anzuspringen.
alt	Alternativtext einer Grafikeinbindung, falls die Grafik nicht angezeigt werden kann.
autocomplete	Speichert Eingabe und bietet beim nächsten Ausfüllen alte Eingaben zur Auswahl an. Werte sind <code>on</code> (Default) oder <code>off</code> .

Tabelle B.1: Unterstützte HTML-Attribute (Fortsetzung)

Attribut	Beschreibung
bgcolor	Die Hintergrundfarbe des Dokuments oder einer Tabellenzelle. Veraltet.
border	Dicke des Rahmens einer Tabelle in Pixel.
cellpadding	Raum zwischen Zellenrahmen und Zelleninhalt in Pixel.
cellspacing	Raum zwischen Zellen bzw. zwischen Zelle und Tabellenrahmen in Pixel.
charset	Character-Encoding des Zieldokuments eines Links.
cols	Anzahl der Zeichen einer Zeile in einer Textarea.
coords	Bildschirmkoordinaten eines Fensters, das über einen Link neu zu öffnen ist.
dir	Textfluss. Entweder <code>ltr</code> (left-to-right) oder <code>rtl</code> (right-to-left).
disabled	Die nicht textbasierte Input-Komponente kann den Fokus nicht bekommen und wird bei Betätigen der Tabulatortaste übersprungen. Der Inhalt der Komponente wird nicht als Formularparameter übertragen.
enctype	Codierungsformat, in dem Formulardaten an den Server übertragen werden. Standard ist „application/x-www-form-urlencoded“. Wird in der Regel nicht gesetzt werden müssen, außer das Formular enthält einen File-Upload oder eine <code>mailto</code> -URL.
frame	Rahmenbeschreibung einer Tabelle. Mögliche Werte: <code>void</code> , <code>above</code> , <code>below</code> , <code>hsides</code> , <code>vsides</code> , <code>lhs</code> , <code>rhs</code> , <code>box</code> , <code>border</code> .
height	Höhe eines Bildes in Pixel.
hreflang	Zweizeichen-ISO-Sprach-Code des referenzierten Dokuments.
ismap	Angabe, dass das Image eine Image-Map ist.
lang	Basissprache als Zweizeichen-ISO-Sprach-Code.
longdesc	URL einer ausführlichen Beschreibung eines Image.
readonly	Kann durch den Benutzer nicht geändert werden.
rel	Verwendung im <code><a></code> -Element, um Quelle und Ziel zu verbinden.
rev	Verwendung im <code><a></code> -Element, um Ziel und Quelle zu verbinden.
rules	Beschreibung des Zellrahmens einer Tabelle. Mögliche Werte sind: <code>rows</code> , <code>cols</code> , <code>groups</code> , <code>none</code> , <code>all</code> .
shape	Definiert den Umriss eines <code><a></code> -Links.

Tabelle B.1: Unterstützte HTML-Attribute (Fortsetzung)

Attribut	Beschreibung
size	Breite eines Eingabefeldes in Anzahl Zeichen oder Anzahl der sichtbaren Alternativen einer Auswahl.
style	CSS-Deklaration (<i>property : wert</i>).
styleClass	CSS-Klassen, durch Komma getrennt.
summary	Text als Zusammenfassung einer Tabelle.
tabindex	Position der Tabulatorordnung einer Seite.
target	Name eines Frames als Link-Ziel.
title	Kurzbeschreibung der Komponente zur Anzeige als Tool-Tip.
usemap	Client-side Image-Map.
width	Breite einer Tabelle in Punkten oder als Prozentsatz des verfügbaren Platzes.

B.2 JavaScript-basierte HTML-Attribute

Der HTML-Standard beschreibt, wie client-seitige Script-Sprachen in HTML zu integrieren sind. Die am häufigsten verwendete Script-Sprache ist *JavaScript*, die in jedem Browser – wenn auch in geringfügig verschiedenen Implementierungen – vorhanden ist. VB-Script ist eine Alternative, doch nur für die Windows-Plattform verfügbar. Durch die Dominanz von JavaScript begründet, sprechen wir hier nur von JavaScript, wohl wissend, dass auch Alternativen möglich sind. Abschnitt 18.2.3 des Standards zählt eine Reihe von *Intrinsic Events* (immanente, spezifische Events) auf, an die Event-Handler gebunden werden können. Die meisten Events sind Tastatur- und Maus-Events.

JavaServer Faces erlaubt es, solchen Event-Handler-Code an JSF-Komponenten bzw. die dann gerenderten HTML-Elemente zu binden. In Tabelle B.2 beschreiben wir diese Attribute allgemein. Bei der Vorstellung der JSF-Tags in Abschnitt B.4 werden dann die für das Tag erlaubten Event-Handler-Attribute aufgezählt. Mit der Integration von Ajax in JSF 2.0 gehört eine Verwendung der JavaScript-Events als Attributnamen der Vergangenheit an, wenngleich sie nach wie vor möglich ist. Beim Einsatz des `<f:ajax>`-Tags, das Ajax-Funktionalität in JSF 2.0 bereitstellt, sind für das Attribut `event` die in Tabelle B.2 verwendeten JavaScript-Events ohne das führende „on“ zu verwenden, also z. B. `blur` statt `onblur` und `click` statt `onclick`.

Eine Einführung in JavaScript geht über den Rahmen des Buches hinaus. Der interessierte Leser findet z. B. in [Fla01] eine ausführliche Darstellung der Sprache.

Bitte beachten Sie, dass der Anwender JavaScript im Browser aktiviert haben muss, damit die Event-Handler ausgeführt werden können.

Tabelle B.2: Unterstützte JavaScript-Attribute

Attribut	Beschreibung
onblur	Event-Handler-Code, der aufgerufen wird, wenn das Element den Fokus verliert.
onchange	Event-Handler-Code, der aufgerufen wird, wenn das Element den Fokus verliert und sich der Wert des Elements geändert hat.
onclick	Event-Handler-Code, der aufgerufen wird, wenn das Element angeklickt wird.
ondblclick	Event-Handler-Code, der aufgerufen wird, wenn das Element doppelgeklickt wird.
onfocus	Event-Handler-Code, der aufgerufen wird, wenn das Element den Fokus bekommt.
onkeydown	Event-Handler-Code, der aufgerufen wird, wenn eine Taste über dem Element gedrückt wird.
onkeypress	Event-Handler-Code, der aufgerufen wird, wenn eine Taste über dem Element gedrückt und wieder losgelassen wird.
onkeyup	Event-Handler-Code, der aufgerufen wird, wenn eine Taste über dem Element losgelassen wird.
onmousedown	Event-Handler-Code, der aufgerufen wird, wenn ein Mausknopf über dem Element gedrückt wird.
onmousemove	Event-Handler-Code, der aufgerufen wird, wenn die Maus über dem Element bewegt wird.
onmouseout	Event-Handler-Code, der aufgerufen wird, wenn die Maus das Element verlässt.
onmouseover	Event-Handler-Code, der aufgerufen wird, wenn die Maus in das Element bewegt wird.
onmouseup	Event-Handler-Code, der aufgerufen wird, wenn ein Mausknopf über dem Element losgelassen wird.
onreset	Event-Handler-Code, der aufgerufen wird, wenn ein Formular zurückgesetzt wird.
onselect	Event-Handler-Code, der aufgerufen wird, wenn Text selektiert wird.

Tabelle B.2: Unterstützte JavaScript-Attribute (Fortsetzung)

Attribut	Beschreibung
onsubmit	Event-Handler-Code, der aufgerufen wird, wenn das Formular abgeschickt wird.

B.3 HTML- und JSF-Attribute für CSS

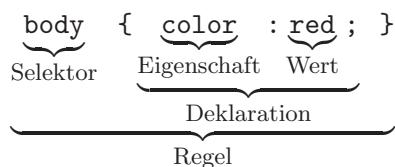
In Tabelle B.1 wurden bereits die HTML-Attribute `style` und `styleClass` eingeführt. Mit ihnen ist es möglich, *Cascading Style-Sheets* (CSS) zu definieren, die zur Formgestaltung von HTML-Elementen dienen. Die genannten beiden HTML-Attribute werden von JSF unverändert in das generierte HTML übernommen. Einige JSF-Tags, vor allem die Tags für Fehlermeldungen und die Tags, die tabellarische Strukturen definieren, ermöglichen die Verwendung einer Reihe weiterer Attribute mit CSS-Angaben. Diese Attribute werden bei der Vorstellung der einzelnen JSF-Tags im nächsten Abschnitt jeweils angegeben. Hier gehen wir auf CSS allgemein ein.

Um optisch ansprechende Anwendungen mit JavaServer Faces entwickeln zu können, kommt man an CSS nicht vorbei. Wir haben in den Beispielen des Buches immer versucht, den CSS-Teil relativ gering und einfach zu halten. Für kommerzielle JSF-Anwendungen benötigt man einen höheren Anteil an CSS. Eine Einführung in CSS geht aber weit über den Fokus des Buches hinaus. Der interessierte Leser findet in [Mey00] eine fundierte und umfassende Einführung in die Sprache.

Wir stellen im Folgenden kurz die syntaktischen Grundregeln vor.

B.3.1 Regeln

Eine CSS-Regel besteht aus einem *Selektor* und einer *Deklaration*. Eine Deklaration besteht aus einer *Eigenschaft* (engl. Property), gefolgt von einem Doppelpunkt und einem *Wert*. Eine Deklaration erfolgt in einem Deklarationsblock, der durch geschweifte Klammern begrenzt wird. Es ergibt sich die folgende Struktur:



Als einfache Selektoren sind die HTML-Tags erlaubt. Der Wert kann ein einziges Schlüsselwort sein, aber auch aus mehreren Schlüsselwörtern bestehen, falls die Eigenschaft dies zulässt. Beispiele sind

```
h2 {color: purple;}  
p {font: sans-serif;}  
table {border: 2px solid #940839;}
```

die für Überschriften der Ebene 2 die Textfarbe auf Violett, für Paragraphen den Font auf einen serifelosen Font und für Tabellen einen durchgezogenen Rahmen von 2 Pixeln in einer hexadezimal definierten Farbe setzen. Die darstellte Formatierung ohne Leerzeichen vor dem Doppelpunkt und dem Semikolon wurde von frühen Browsern so erzwungen und hat sich mittlerweile als Standard etabliert.

Sowohl Selektoren als auch Deklarationen können gruppiert werden:

```
h1 {color: purple; background: white;}  
h1 h2 h3 {color: purple;}  
h1 h2 h3 {color: purple; background: white;}
```

Ein zentrales Konzept von CSS ist die Vererbung. Die HTML-Elemente eines Dokumentes sind als Knoten in einem Baum (DOM) angeordnet. CSS-Formatierungen für einen Knoten vererben sich auf alle Unterknoten und gelten daher für den gesamten Teilbaum, können aber in Sohnknoten überschrieben werden.

B.3.2 Klassen- und Id-Selektoren

Bisher haben wir als Selektoren einfache HTML-Elemente verwendet. Weitere Selektorarten sind *Klassenselektoren* und *Id-Selektoren*. Bei einem Klassenselektor wird eine Klasse definiert, z. B. die Klasse `info`

```
.info: {font-weight: bold; color: blue;}
```

Diese kann dann mit dem `class`-Attribut in allen HTML-Tags verwendet werden

```
<p class="info">  
    Dieser Text wird fett und blau  
</p>
```

Bei der obigen Definition der Klasse wird das Style-Sheet für *jedes* HTML-Tag angewendet, das die Klasse verwendet. Man kann dies auf spezielle Tags eingrenzen, indem das Tag vor den Punkt geschrieben wird:

```
p.info: {font-weight: bold; color: blue;}  
h2.info: {font-weight: bold; color: red;}
```

Der Text im Paragraphen bleibt unverändert, in der Überschrift erscheint er rot.

Falls auch diese Beschränkung noch zu allgemein ist, kann mit einem ID-Selektor eine im Dokument eindeutige Id für die Klasse vergeben werden. Die Syntax verwendet an Stelle des Punktes das Rautezeichen.

```
#format1: {font-style: italic; font-size: smaller;}
```

Diese Klasse wird mit dem `id`-Attribut verwendet, das fast alle HTML-Tags erlauben. Meist erfolgt die Verwendung in einem `<div>`

```
<div id="format1">  
...  
</div>
```

Es gibt noch eine ganze Reihe weiterer Details zum Thema CSS. Der interessierte Leser sei noch einmal auf [Mey00] verwiesen.

B.4 HTML-Tag-Bibliothek

Die Tags der HTML-Tag-Bibliothek werden als HTML gerendert. Wir beschreiben in diesem Abschnitt alle Tags der HTML-Bibliothek, indem wir zunächst das Tag und die zugehörige Komponente benennen. Die Komponenten befinden sich im Package `javax.faces.component.html` mit der einzigen Ausnahme von `UIColumn`, die sich im Package `javax.faces.component` befindet. Es folgt die Angabe der Syntax, bei der jedoch nur die JSF-Attribute angegeben werden. Viele der Attribute sind in mehreren Tags erlaubt. Wir fassen daher die Beschreibung der gemeinsamen Attribute in Tabelle B.3 zusammen. Attribute, die nur bei einem Tag erlaubt sind, werden auch dort beschrieben. Es folgt die Aufzählung der HTML-Attribute, die in statische und dynamische Attribute unterschieden werden. Die statischen Attribute sind in Tabelle B.1, die dynamischen in Tabelle B.2 beschrieben.

Tabelle B.3: Gemeinsame Attribute der HTML-Tag-Bibliothek

Attribut	Typ	EL-Typ	Beschreibung
<code>action</code>	<code>String</code>	Methodenausdruck	Methode, die zur Verarbeitung des Action-Events aufgerufen wird; Return-Typ ist <code>Object</code> , keine Parameter. Meist wird <code>String</code> als Return-Typ verwendet.

Tabelle B.3: Gemeinsame Attribute der HTML-Tag-Bibliothek (Fortsetzung)

Attribut	Typ	EL-Typ	Beschreibung
action-Listener	String	Methodenausdruck	Methode, die zur Verarbeitung des Action-Events aufgerufen wird; Return-Typ ist <code>void</code> , ein Parameter vom Typ <code>ActionEvent</code> .
binding	String	Werteausdruck	Werteausdruck für ein Property.
collection-Type	String	Werteausdruck	Voll qualifizierter Klassenname oder Klasse, die das Collection-Interface implementiert.
converter	String oder Converter	Werteausdruck	Die Id eines registrierten Converters oder ein Objekt, das das Converter-Interface implementiert.
converter-Message	String	beliebig	Fehlermeldung für den Konvertierer. Überschreibt die Standardmeldung.
disabled-Class	String	beliebig	CSS-Klasse, die dem <code><label></code> -Element einer gesperrten Auswahl zugewiesen wird.
enabled-Class	String	beliebig	CSS-Klasse, die dem <code><label></code> -Element einer möglichen Auswahl zugewiesen wird.
escape	boolean	beliebig	Sonderzeichen werden als HTML- und XML-Entities dargestellt. Default ist <code>true</code> .
id	String	Keine EL möglich	Die Id der Komponente.
hide-NoSelection-Option	String	beliebig	Verwendung in <code><h:select*></code> -Komponenten. Das enthaltene Select-Item mit gesetztem <code>noSelectionOption</code> -Attribut wird nicht gerendert, falls <code>true</code> .

Tabelle B.3: Gemeinsame Attribute der HTML-Tag-Bibliothek (Fortsetzung)

Attribut	Typ	EL-Typ	Beschreibung
immediate	boolean	beliebig	Bei Eingabekomponenten erfolgt die Konvertierung und Validierung bereits nach der Phase 2, <i>Übernahme der Anfragewerte</i> , falls true , sonst in der Validierungsphase. Bei Steuerkomponenten erfolgt der Aufruf von Action- und Action-Listener-Methoden ebenfalls vorgezogen nach der Phase 2, falls true , sonst in der Phase 5.
rendered	boolean	beliebig	Angabe, ob Komponente gerendert werden und ob sie an der Anfrageverarbeitung teilnehmen soll.
required	boolean	beliebig	Verifikation, ob Eingabe vorhanden ist.
required-Message	String	beliebig	Fehlermeldung für diese Validierung. Überschreibt die Standardmeldung.
validator	String	Methodenausdruck	Aufzurufende Methode mit Rückgabetyp void und Parametern FacesContext , UIComponent und Object , um den Eingabewert zu validieren.
validator-Message	String	beliebig	Fehlermeldung für den Validierer. Überschreibt die Standardmeldung.
value	Object	Werteausdruck	Der Wert der Komponente.
value-Change-Listener	String	Methodenausdruck	Aufzurufende Methode mit Rückgabetyp void und einem Parameter vom Typ ValueChangeEvent , um das Event der Komponente zu verarbeiten.

<h:body>	HTMLBody
-----------------------	-----------------

Syntax

```
<h:body [binding="componentBinding"] />
```

HTML-Attribute

statisch	dir, lang, style, styleClass, title
dynamisch	onclick, ondblclick, onkeydown, onkeypress, onkeyup, onload, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onunload

Beschreibung

Das **<h:body>**-Tag repräsentiert den Seitenrumpf (**<body>**) in HTML. Das HTML-**<body>**-Tag darf auch direkt in einer JSF-Seite verwendet werden. Wollen Sie jedoch ein Skript mit **<h:outputScript>** und **target="body"** einbinden, so muss das **<h:body>**-Tag verwendet werden; siehe auch Abschnitt 4.10.3 auf Seite 191.

<h:button>	HTMLOutcomeTargetButton
-------------------------	--------------------------------

Syntax

```
<h:button [id="id"] [binding="componentBinding"]
           [rendered="true|false"] [value="value"]
           [image="imageURL"] [includeViewParams="true|false"]
           [outcome="outcome"] [fragment="fragmentId"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
fragment	String	beliebig	Id des Seitenfragments. Wird mit # an das Ende des Ziel-URLs angehängt.
image	String	beliebig	Absolute oder relative URL eines Bildes; im generierten HTML wird type="image" gesetzt.
include-ViewParams	boolean	beliebig	Seitenparameter werden im Ziel-URI eingebunden, falls true.
outcome	String	beliebig	Ergebnis, das für die Navigation verwendet wird. Bei Verwendung einer View-Id wird diese als Ziel für die implizite Navigation generiert.

HTML-Attribute

statisch	accesskey, alt, dir, lang, readonly, style, styleClass, tabIndex, title
dynamisch	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Das **<h:button>**-Tag erzeugt eine Schaltfläche, die bei Betätigung einen GET-Request erzeugt. Das Tag muss daher nicht in ein Formular (**<h:form>**) eingebettet sein. Das Navigationsziel wird über das Attribut **outcome** bestimmt. Die Berechnung des Navigationsziels erfolgt entsprechend den JSF-Navigationsregeln über das **<from-outcome>** einer Navigationsregel in der JSF-Konfigurationsdatei oder über implizite Navigation. Die Berechnung findet in der Phase

6, *Rendern der Antwort*, und nicht, wie üblich, in Phase 5, *Aufruf der Anwendungslogik*, des nachfolgenden Post-Backs statt. Enthaltene `<f:param>`-Tags werden als „`name=wert`“-Paare an das Navigationsziel angehängt.

Beispiel:

```
<h:button outcome="view-parameter.xhtml" value="...">
    <includeViewParams="true">
        <f:param name="prozent" value="12" />
    </h:button>
```

<code><h:column></code>	HTMLColumn
-------------------------------	-------------------

Syntax

```
<h:column [id="id"] [binding="componentBinding"] [rendered="true|false"]
           [footerClass="styleClass"] [headerClass="styleClass"]
           [rowHeader="true|false"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
<code>footerClass</code>	String	beliebig	Liste von CSS-Klassen, durch Komma getrennt, die jedem Spalten-Footer dieser Tabelle zugewiesen werden.
<code>headerClass</code>	String	beliebig	Liste von CSS-Klassen, durch Komma getrennt, die jedem Spalten-Header dieser Tabelle zugewiesen werden.
<code>rowHeader</code>	boolean	beliebig	Spalte wird als Zeilen-Header (<code>th</code> statt <code>td</code>) gerendert.

HTML-Attribute

statisch	—
dynamisch	—

Beschreibung

Ein `<h:column>`-Tag stellt eine Spalte in einem `<h:dataTable>`-Element dar. Die Komponente wird durch den Renderer des `<h:dataTable>`-Tags gerendert und verfügt über keinen eigenen assoziierten Renderer. Eine Spalte kann mit einer Facette (`<f:facet>`) versehen werden, die als Kopf- (`name="header"`) oder Fußzeile (`name="footer"`) der Spalte gerendert wird. Die enthaltenen Komponenten werden als Daten der Spalte dargestellt.

Beispiele: Listing 3.1 auf Seite 22 und Listing 8.9 auf Seite 303.

<h:commandButton>	HTMLCommandButton
--------------------------------	--------------------------

Syntax

```
<h:commandButton [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [type="submit|button|reset"]
    [image=" imageURL"] [action="actionMethodBinding"]
    [actionListener="actionListenerBinding"] [immediate="true|false"]
    [label="label"] [readonly="true|false"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
image	String	beliebig	Absolute oder relative URL eines Bildes; im generierten HTML wird type="image" gesetzt.
label	String	beliebig	Name der Komponente.
type	String	beliebig	Legt die Art der Schaltfläche fest; entweder submit, button oder reset. Keine Auswirkung, wenn das image-Attribut gesetzt ist.

HTML-Attribute

statisch	accesskey, alt, dir, disabled, lang, readonly, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein **<h:commandButton>**-Tag stellt eine Steuerkomponente dar und wird als Schaltfläche gerendert. Sie wird in HTML als **<input>**-Tag mit einem type-Attribut und den Werten **submit**, **reset**, **button** oder **image** dargestellt, abhängig von den JSF-Attributen **type** und **image**. Das Betätigen der Schaltfläche schickt das Formular ab, in dem sich die Schaltfläche befindet, und löst ein **ActionEvent** aus.

Beispiele: Listing 3.1 auf Seite 22 und Listing 8.8 auf Seite 300.

<code><h:commandLink></code>	HTMLCommandLink
------------------------------------	------------------------

Syntax

```
<h:commandLink [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [immediate="true|false"]
    [action="actionMethodBinding"]
    [actionListener="actionListenerBinding"]
/>
```

HTML-Attribute

statisch	accesskey, charset, coords, dir, hreflang, lang, rel, rev, shape, style, styleClass, tabindex, target, title, type
dynamisch	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Ein `<h:commandLink>`-Tag stellt eine Steuerkomponente dar und wird als HTML-`<a>`-Tag dargestellt. Das `href`-Attribut hat als Wert „#“. Das `onclick`-Attribut enthält JavaScript-Code, der das Formular mit einem `ActionEvent` an den Server schickt. Enthaltene `<f:param>`-Elemente werden als zusätzliche Parameter an den Request gebunden und können in einem registrierten Action-Listener verwendet werden.

Beispiele: Listing 6.7 auf Seite 232 und Listing 8.2 auf Seite 288.

<h:dataTable>	HTMLDataTable
----------------------------	----------------------

Syntax

```
<h:dataTable [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"]
    [var="var"] [first="first"] [rows="rows"] [bodyrows="bodyrows"]
    [captionClass="styleClass"] [captionStyle="style"]
    [headerClass="styleClass"] [footerClass="styleClass"]
    [rowClasses="styleClass"] [columnClasses="styleClass"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
bodyrows	String	Werte-ausdruck	Liste von Zeilenindizes, durch Komma getrennt; nach jedem Index wird ein neues (<tbody>) begonnen.
captionClass	String	Werte-ausdruck	Liste von CSS-Klassen, durch Komma getrennt; werden als CSS-Klassen der caption (<caption>) zugewiesen.
captionStyle	String	Werte-ausdruck	CSS-Style für die caption (<caption>).
columnClasses	String	Werte-ausdruck	Liste von CSS-Klassen, durch Komma getrennt; werden als CSS-Klassen den Spalten (<td>) zugewiesen.
first	int	beliebig	Angabe der ersten Zeile des Datenmodells, die anzuzeigen ist; fängt bei 0 an.
footerClass	String	beliebig	CSS-Klasse für die Footer.
headerClass	String	beliebig	CSS-Klasse für den Header.
rowClasses	String	beliebig	Liste von CSS-Klassen, durch Komma getrennt; werden iterativ als CSS-Klassen den Zeilen (<tr>) zugewiesen.
rows	int	beliebig	Die Anzahl der darzustellenden Zeilen.
var	String	Keine EL möglich	Der Name der Variablen, die die aktuelle Zeile enthält.

HTML-Attribute

statisch	bgcolor, border, cellpadding, cellspacing, dir, frame, lang, rules, style, styleClass, summary, title, width
dynamisch	onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Ein `<h:dataTable>`-Tag stellt eine Tabelle dar. Sie wird in HTML als `<table>`-Tag dargestellt. Enthaltene `<h:column>`-Tags bilden die Spalten der Tabelle. Als Werttyp des `value`-Attributs sind beliebige Typen zugelassen. Ein `javax.faces.model.DataModel` wird unverändert übernommen. Andere Datentypen werden automatisch in verschiedene spezifische Typen wie folgt eingepackt:

- `Object[]` in `javax.faces.model.ArrayDataModel`
- `java.util.List` in `javax.faces.model.ListDataModel`
- `javax.servlet.jsp.jstl.sql.Result` in `javax.faces.model.ResultDataModel`
- `java.sql.ResultSet` in `javax.faces.model.ResultSetDataModel`
- jeder andere Typ in `javax.faces.model.ScalarDataModel`

Die Tabelle kann mit Facetten (`<f:facet name="...">`) versehen werden. Mögliche Facetten sind `header`, `footer` und `caption`. Header, Footer, Caption sowie Zeilen und Zellen lassen sich mit CSS-Style-Klassen versehen und die erste darzustellende Zeile sowie die Anzahl der darzustellenden Zeilen ebenfalls angeben.

Beispiele: Listing 3.1 auf Seite 22 und Listing 8.9 auf Seite 303.

<h:form>	HTMLForm
-----------------------	-----------------

Syntax

```
<h:form [id="id"] [binding="componentBinding"]
         [rendered="true|false"] [prependId="true|false"]
      />
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
prependId	boolean	beliebig	Id des Formulars wird als Präfix für die Ids der enthaltenen Komponenten verwendet, falls true . Siehe Abschnitt 4.9.

HTML-Attribute

statisch	accept, acceptcharset, enctype, dir, lang, style, styleClass, target, title
dynamisch	onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onreset, onsubmit

Beschreibung

Ein **<h:form>**-Tag bindet Eingabekomponenten einer Seite in ein HTML-Formular ein. Es sind mehrere **<h:form>**-Tags in einer Seite erlaubt, doch werden in einer JSF-Anfrage immer nur die Eingabekomponenten des Anfrage stellenden Formulars an die Anfrage gebunden. Ein **<h:form>**-Tag muss mindestens eine Steuerkomponente (**<h:commandButton>**, **<h:commandLink>**), Ajax-Unterstützung (**<f:ajax>**) oder JavaScript mit einer **submit()**-Funktion enthalten. Das entstehende HTML-<form> wird mit einem POST-Request an den Server geschickt.

Beispiele: Listing 3.1 auf Seite 22 und Listing 7.1 auf Seite 251.

<code><h:graphicImage></code>	HTMLGraphicImage
-------------------------------------	-------------------------

Syntax

```
<h:graphicImage [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [url="imageURL"]
    [library="name"] [name="imageURL"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
library	String	beliebig	Bibliotheksname der Graphik. Siehe Abschnitt 4.10.
name	String	beliebig	Ressource-Name der Graphik. Siehe Abschnitt 4.10.
url	String	beliebig	Die URL des Bildes. Identisch zum value-Attribut.

HTML-Attribute

statisch	alt, dir, height, ismap, lang, longdesc, style, styleClass, title, usemap, width
dynamisch	onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Ein `<h:graphicImage>`-Tag wird in HTML als ``-Tag dargestellt. Das HTML-`src`-Attribut erhält den Wert des `value`- oder `url`-Attributs zugewiesen. Bei der Verwendung der mit JSF 2.0 eingeführten Attribute `library` und `name` wird das `src`-Attribut mit dem in Abschnitt 4.10 dargestellten Verfahren bestimmt.

Beispiele: Mehrere Verwendungen in den Abschnitten 4.10.1 und 4.10.2.

<h:head>	HTMLHead
-----------------------	-----------------

Syntax

```
<h:head [binding="componentBinding"] />
```

HTML-Attribute

statisch	dir, lang
dynamisch	—

Beschreibung

Das **<h:head>**-Tag repräsentiert den Seitenkopf (**<head>**) in HTML. Das HTML-<head>-Tag darf auch direkt in einer JSF-Seite verwendet werden. Will man jedoch ein Skript mit **<h:outputScript>** und **target="head"** oder ein Stylesheet mit **<h:outputStylesheet>** einbinden, so muss das **<h:head>**-Tag verwendet werden; siehe auch Abschnitt 4.10.3 auf Seite 191. Dies impliziert die Verwendung von **<h:head>**, falls Ajax mit **<f:ajax>** genutzt wird. Verwendetes JavaScript und CSS werden im Seitenkopf platziert.

Beispiele: Code-Ausschnitt auf Seite 192 und Listing 6.7 auf Seite 232.

<code><h:inputHidden></code>	HTMLInputHidden
------------------------------------	------------------------

Syntax

```
<h:inputHidden [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
/>
```

HTML-Attribute

statisch	—
dynamisch	—

Beschreibung

Ein `<h:inputHidden>`-Tag wird in HTML als `<input type="hidden">`-Tag dargestellt. Das Tag wird für die Interaktion mit dem Benutzer nicht benötigt, kann aber verwendet werden, um Daten zwischen Seiten auszutauschen.

<h:inputSecret>	HTMLInputSecret
-----------------	-----------------

Syntax

```
<h:inputSecret [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [redisplay="true|false"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
redisplay	boolean	beliebig	Falls auf true gesetzt, wird der Wert beim Rendern dem <input>-Element als Wert zugewiesen.

HTML-Attribute

statisch	accesskey, alt, autocomplete, dir, disabled, lang, maxlength, readonly, size, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:inputSecret>-Tag stellt eine Passworteingabe dar und wird in HTML als <input>-Tag mit type= "password" gerendert.

Beispiele: Listing 6.7 auf Seite 232 und Code-Ausschnitt auf Seite 293.

<h:inputText>	HTMLInputText
---------------	---------------

Syntax

```
<h:inputText [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
/>
```

HTML-Attribute

statisch	accesskey, alt, autocomplete, dir, disabled, lang, maxlength, readonly, size, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:inputText>-Tag stellt eine Texteingabe dar und wird in HTML als <input>-Tag mit type="text" gerendert.

Beispiele: Listing 3.3 auf Seite 27 und Listing 8.8 auf Seite 300.

<code><h:inputTextarea></code>	HTMLInputTextarea
--------------------------------------	--------------------------

Syntax

```
<h:inputTextarea [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
/>
```

HTML-Attribute

statisch	accesskey, cols, dir, disabled, lang,readonly, rows, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein `<h:inputTextarea>`-Tag wird in HTML als `<textarea>`-Tag dargestellt und dient zur mehrzeiligen Texteingabe. Die HTML-Attribute `rows` und `cols` werden zur Größenbestimmung in das generierte HTML durchgereicht.

Beispiel: Listing 4.22 auf Seite 125.

<code><h:link></code>	HTMLOutcomeTargetLink
-----------------------------	------------------------------

Syntax

```
<h:link [id="id"] [binding="componentBinding"]
         [rendered="true|false"] [value="value"]
         [image="imageURL"] [includeViewParams="true|false"]
         [outcome="outcome"] [fragment="fragmentId"]
    />
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
fragment	String	beliebig	Id des Seitenfragments. Wird mit # an das Ende des Ziel-URLs angehängt.
image	String	beliebig	Absolute oder relative URL eines Bildes; im generierten HTML wird type="image" gesetzt.
include-ViewParams	boolean	beliebig	Seitenparameter werden im Ziel-URI eingebunden, falls true.
outcome	String	beliebig	Ergebnis, das für die Navigation verwendet wird. Bei Verwendung einer View-Id wird diese als Ziel für die implizite Navigation generiert.

HTML-Attribute

statisch	accesskey, alt, charset, coords, dir, disabled, hreflang, lang, rel, rev, shape, style, styleClass, tabindex, target, title
dynamisch	onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Das `<h:link>`-Tag erzeugt einen Link (HTML-Tag `<a>`), der bei Betätigung einen GET-Request erzeugt. Das Tag muss daher nicht in ein Formular eingebettet sein. Das Navigationsziel (HTML-Attribut `href`) wird über das Attribut `outcome` bestimmt. Die Berechnung des Navigationsziels erfolgt entsprechend den JSF-Navigationsregeln über das `<from-outcome>` einer Navigationsregel

in der JSF-Konfigurationsdatei oder über implizite Navigation. Die Berechnung findet in der Phase 6, *Rendern der Antwort*, und nicht wie üblich in der Phase 5, *Aufruf der Anwendungslogik*, des nachfolgenden Post-Backs statt. Enthaltene `<f:param>`-Tags werden als „`name=wert`“-Paare an das Navigationsziel angehängt.

Beispiel:

```
<h:link outcome="view-parameter.xhtml" value="..."  
    includeViewParams="true">  
    <f:param name="prozent" value="12" />  
</h:link>
```

<code><h:message></code>	HTMLMessage
--------------------------------	--------------------

Syntax

```
<h:message for="componentId" [id="id"] [binding="componentBinding"]
[rendered="true|false"] [tooltip="true|false"]
[showDetail="true|false"] [showSummary="true|false"]
[errorClass="styleClass"] [errorStyle="style"]
[fatalClass="styleClass"] [fatalStyle="style"]
[infoClass="styleClass"] [infoStyle="style"]
[warnClass="styleClass"] [warnStyle="style"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
<code>errorClass</code>	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Error-Meldung ist.
<code>errorStyle</code>	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Error-Meldung ist.
<code>fatalClass</code>	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Fatal-Meldung ist.
<code>fatalStyle</code>	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Fatal-Meldung ist.
<code>for</code>	String	beliebig	Id der Komponente, für die die Meldung angezeigt werden soll.

infoClass	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Info-Meldung ist.
infoStyle	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Info-Meldung ist.
showDetail	boolean	beliebig	Stellt die detaillierte Meldung dar.
showSummary	boolean	beliebig	Stellt die kurze Meldung dar.
tooltip	boolean	beliebig	Der Meldungstext wird im <code>title</code> -Attribut des <code></code> -Elements verwendet (so genannter Tooltip).
warnClass	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Warn-Meldung ist.
warnStyle	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Warn-Meldung ist.

HTML-Attribute

statisch	style, styleClass, title
dynamisch	—

Beschreibung

Die erste Fehlermeldung für die im **for**-Attribut angegebene Komponente wird angezeigt. Die Attribute **showDetail** und **showSummary** geben an, ob die ausführliche Meldung oder die Kurzmeldung angezeigt wird.

Die im **for**-Attribut angegebene Id entspricht der Client-Id der Komponente (siehe Abschnitt 4.9), kann in der Regel aber auch abgekürzt werden, z.B. durch die Komponenten-Id. Der exakte Algorithmus zur Bestimmung der Komponente ist relativ umfangreich und wird hier nicht dargestellt. Er ist in der API-Doc für die Methode **findComponent()** der Klasse **UIComponent** dokumentiert.

Beispiele: Listing 3.3 auf Seite 27 und Listing 8.7 auf Seite 296.

<h:messages>	HTMLMessages
---------------------------	---------------------

Syntax

```
<h:messages [id="id"] [binding="componentBinding"]
            [rendered="true|false"] [tooltip="true|false"]
            [globalOnly="true|false"] [layout="list|table"]
            [showDetail="true|false"] [showSummary="true|false"]
            [errorClass="styleClass"] [errorStyle="style"]
            [fatalClass="styleClass"] [fatalStyle="style"]
            [infoClass="styleClass"] [infoStyle="style"]
            [warnClass="styleClass"] [warnStyle="style"]
        />
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
errorClass	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Error-Meldung ist.
errorStyle	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Error-Meldung ist.
fatalClass	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Fatal-Meldung ist.
fatalStyle	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Fatal-Meldung ist.

for	String	beliebig	Id der Komponente, für die die Meldung angezeigt werden soll. Die Id wird als Client-Id im selben Format wie für die Methode <code>findComponent()</code> angegeben.
globalOnly	boolean	beliebig	Nur Meldungen anzeigen, die keiner Komponenten-Id zugeordnet sind.
infoClass	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Info-Meldung ist.
infoStyle	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Info-Meldung ist.
layout	String	beliebig	Werte <code>table</code> oder <code>list</code> . Meldungen werden als Zeilen einer Tabelle beziehungsweise als Liste (<code>...</code>) dargestellt.
showDetail	boolean	beliebig	Stellt die detaillierte Meldung dar.
showSummary	boolean	beliebig	Stellt die kurze Meldung dar.
tooltip	boolean	beliebig	Der Meldungstext wird im <code>title</code> -Attribut des <code></code> -Elements verwendet (so genannter Tooltip).
warnClass	String	beliebig	CSS-Klassennamen, durch Komma getrennt, die als CSS-Klassen (Attribut <code>class</code>) dem <code></code> -Element zugewiesen werden, falls die Meldung eine Warn-Meldung ist.
warnStyle	String	beliebig	CSS-Deklaration, die dem <code>style</code> -Attribut des <code></code> -Elements zugewiesen wird, falls die Meldung eine Warn-Meldung ist.

HTML-Attribute

statisch	style, styleClass, title
dynamisch	—

Beschreibung

Alle Fehlermeldungen werden angezeigt (oder falls `globalOnly` auf `true` gesetzt ist, nur die ohne Komponenten-ID). Die Attribute `showDetail` und `showSummary` geben an, ob die ausführlichen Meldungen oder die Kurzmeldungen angezeigt werden. Das `layout`-Attribut steuert die Anzeige der Meldungen als Liste oder als Tabelle.

Beispiele: Listing 4.15 auf Seite 91 und Code-Ausschnitt auf Seite 108.

<code><h:outputFormat></code>	HTMLOutputFormat
-------------------------------------	-------------------------

Syntax

```
<h:outputFormat [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [converter="converter"]
    [escape="true|false"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
----------	-----	--------	--------------

HTML-Attribute

statisch	dir, lang, style, styleClass, title
dynamisch	—

Beschreibung

Ein `<h:outputFormat>`-Tag rendert parametrisierten Text, der in HTML ebenfalls als normaler Text dargestellt wird. Wenn eines der CSS-Attribute oder das `id`-Attribut gesetzt ist, erfolgt die Darstellung innerhalb eines ``-Elements. Die Parametrisierung erfolgt über das `value`-Attribut. Die aktuellen Parameter müssen dann geschachtelte `<f:param>`-Tags sein. Die Parameterplatzhalter im `value`-Attribut werden durch Indizes bei 0 beginnend in geschweiften Klammern notiert.

Beispiel:

```
<h:outputFormat value="Sehr geehrte/r {0} {1} {2}">
    <f:param value="#{nameHandler.anrede}" />
    <f:param value="#{nameHandler.vorname}" />
    <f:param value="#{nameHandler.nachname}" />
</h:outputFormat>
```

<h:outputLabel>	HTMLOutputLabel
-----------------	-----------------

Syntax

```
<h:outputLabel [id="id"] [binding="componentBinding"] [for="component"]
    [rendered="true|false"] [value="value"] [converter="converter"]
    [escape="true|false"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
for	String	beliebig	Id der Komponente, für die das Label angezeigt werden soll. Die Id wird als Client-Id im selben Format wie für die Methode <code>findComponent()</code> angegeben.

HTML-Attribute

statisch	accesskey, dir, lang, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Ein `<h:outputLabel>`-Tag wird in HTML als `<label>`-Tag dargestellt. Das `for`-Attribut wird in das HTML übernommen. Der Rumpf des `<label>`-Tags wird aus dem Rumpf des `<h:outputLabel>`-Tags oder dem `value`-Attribut erzeugt.

Die im `for`-Attribut angegebene Id entspricht der Client-Id der Komponente (siehe Abschnitt 4.9), kann in der Regel aber auch abgekürzt werden, z. B. durch die Komponenten-Id. Der exakte Algorithmus zur Bestimmung der Komponente ist relativ umfangreich und wird hier nicht dargestellt. Er ist in der API-Doc für die Methode `findComponent()` der Klasse `UIComponent` dokumentiert.

Beispiele: Listing 3.3 auf Seite 27 und Listing 8.7 auf Seite 296.

<h:outputLink>	HTMLOutputLink
----------------	-----------------------

Syntax

```
<h:outputLink [id="id"] [binding="componentBinding"]
              [rendered="true|false"] [value="value"] [converter="converter"]>
    ...
</h:outputLink>
```

HTML-Attribute

statisch	accesskey, charset, coords, dir, disabled, hreflang, lang rel, rev, shape, style, styleClass, tabIndex, target, title, type
dynamisch	—

Beschreibung

Ein `<h:outputLink>`-Tag wird in HTML als `<a>`-Tag dargestellt. Das `href`-Attribut besteht aus dem Komponentenwert. Falls das Element `<f:param>`-Tags enthält, werden diese als Anfrageparameter an das `href`-Attribut angehängt. Dabei werden sowohl Name als auch Wert URL-codiert.

Beispiel:

```
<h:outputLink value="http://www.google.de/search">
    <!-- Leerzeichen und # werden URL-codiert -->
    <f:param name="q" value="Java C#" />
    Google-Suche nach 'Java' und 'C#'
</h:outputLink>
```

<h:outputScript>	UIOutput
-------------------------------	-----------------

Syntax

```
<h:outputScript name="resourceName" [id="id"]  
    [library="libraryName"] [target="head|body|form"]  
    [binding="componentBinding"] [converter="converter"]  
    [rendered="true|false"] [value="value"] [binding="componentBinding"]  
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
library	String	beliebig	Der Bibliotheksname der Skript-Ressource.
name	String	beliebig	Der Name der Skriptdatei.
target	String	beliebig	Die Werte head, body oder form, die das Ziel der Skriptpositionierung angeben. Ohne das Attribut wird das Skript inline gerendert (<script>).

HTML-Attribute

statisch	—
dynamisch	—

Beschreibung

Das **<h:outputScript>**-Tag positioniert eine Skriptdatei an der mit **target** angegebenen Stelle innerhalb der generierten HTML-Seite. Bei der Verwendung der **target**-Werte muss sichergestellt sein, dass sich das entsprechende Tag **<h:head>**, **<h:body>** oder **<h:form>** ebenfalls in der View befindet. Weitere Details findet man im Abschnitt 4.10.3 auf Seite 191.

Beispiel: Code-Ausschnitt auf Seite 240.

<code><h:outputStylesheet></code>	UIOutput
---	-----------------

Syntax

```
<h:outputStylesheet name="resourceName" [id="id"]  
    [library="libraryName"] [binding="componentBinding"]  
    [binding="componentBinding"] [converter="converter"]  
    [rendered="true|false"] [value="value"]  
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
library	String	beliebig	Der Bibliotheksname der Stylesheet-Ressource.
name	String	beliebig	Der Name der Stylesheet-Datei.

HTML-Attribute

statisch	—
dynamisch	—

Beschreibung

Das `<h:outputStylesheet>`-Tag positioniert ein Stylesheet im Kopf der generierten HTML-Seite. Die JSF-Seite muss daher ein `<h:head>` enthalten. Weitere Details findet man im Abschnitt 4.10.3 auf Seite 191.

Beispiel: Code-Ausschnitt auf Seite 192.

<h:outputText>	HTMLOutputText
-----------------------------	-----------------------

Syntax

```
<h:outputText [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [converter="converter"]
    [escape="true|false"]
/>
```

HTML-Attribute

statisch	dir, lang, style, styleClass, title
dynamisch	—

Beschreibung

Ein `<h:outputText>`-Tag wird in HTML als Text dargestellt. Verwendet man eines der HTML-Attribute oder das `id`-Attribut, wird der Text in ein `` eingeschlossen. Während mit JSP das `<h:outputText>`-Tag verwendet werden musste, um den Wert von EL-Ausdrücken in Seiten einzubauen, können mit Facelets EL-Ausdrücke direkt verwendet werden. Die Notwendigkeit zur Verwendung von `<h:outputText>` reduziert sich damit in der Regel auf enthaltene Konvertierer und die Vergabe expliziter Ids.

Beispiele: Listing 3.1 auf Seite 22 und Code-Ausschnitt auf Seite 7.2.2.

<h:panelGrid>	HTMLPanelGrid
---------------	----------------------

Syntax

```
<h:panelGrid [id="id"] [binding="componentBinding"]
    [columns="noOfColumns"] [rendered="true|false"]
    [headerClass="styleClass"] [footerClass="styleClass"]
    [rowClasses="styleClass"] [columnClasses="styleClass"]
    [captionClass="styleClass"] [captionStyle="style"]
    [bodyrows="bodyrows"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
bodyrows	String	Werte- ausdruck	Liste von Zeilenindizes, durch Komma getrennt; nach jedem Index wird ein neues (<tbody>) begonnen.
captionClass	String	Werte- ausdruck	Liste von CSS-Klassen, durch Komma getrennt; werden als CSS-Klassen der caption (<caption>) zugewiesen.
captionStyle	String	Werte- ausdruck	CSS-Style für die caption (<caption>).
columnClasses	String	beliebig	Liste von CSS-Klassen, durch Komma getrennt; werden als CSS-Klassen den Spalten (<td>) zugewiesen.
footerClass	String	beliebig	Die CSS-Klasse für den Footer.
headerClass	String	beliebig	Die CSS-Klasse für den Header.
rowClasses	String	beliebig	Liste von CSS-Klassen, durch Komma getrennt; werden iterativ als CSS-Klassen den Zeilen (<tr>) zugewiesen.

HTML-Attribute

statisch	bgcolor, border, cellpadding, cellspacing, dir, frame, lang, rules, style, styleClass, summary, title, width
dynamisch	onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup

Beschreibung

Ein `<h:panelGrid>`-Tag stellt eine Tabelle dar und wird in HTML als `<table>`-Tag dargestellt. Die Anzahl der Spalten ergibt sich aus dem Wert des `columns`-Attributs. Der Tag-Inhalt wird sukzessive für die einzelnen Tabellenzellen verwendet. Optional kann die Tabelle eine Über- oder/und Unterschrift durch Facetten (`<f:facet>`) erhalten.

Beispiele: Listing 2.2 auf Seite 14 und Listing 3.3 auf Seite 27.

<code><h:panelGroup></code>	HTMLPanelGroup
-----------------------------------	-----------------------

Syntax

```
<h:panelGoup [id="id"] [binding="componentBinding"]
              [rendered="true|false"] [layout="layout"]
            />
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
layout	String	beliebig	Bestimmt den Layout-Typ des generierten HTML. Falls der Wert <code>block</code> ist, wird ein HTML-<div> erzeugt, für andere Werte ein HTML-.

HTML-Attribute

statisch	style, styleClass
dynamisch	—

Beschreibung

Ein `<h:panelGroup>`-Tag stellt einen gruppierenden Container dar. Er wird immer dann verwendet, wenn nur ein JSF-Tag erlaubt ist, aber mehr Tags verwendet werden sollen. Falls eines der CSS-Attribute oder das `id`-Attribut gesetzt ist, erfolgt die Darstellung innerhalb eines ``-Elements, sonst als einfacher Text. Mit dem `layout`-Attribut lässt sich ein `<div>`-Block erzeugen. Mit einem leeren `<h:panelGroup>/` können unvollständige Zeilen eines `<h:panelGrid>` aufgefüllt werden.

Beispiele: Listing 3.3 auf Seite 27 und Listing 4.9 auf Seite 74 .

<h:selectBooleanCheckbox>	HTMLSelectBooleanCheckbox
---------------------------	---------------------------

Syntax

```
<h:selectBooleanCheckbox [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
/>
```

HTML-Attribute

statisch	accesskey, dir, disabled, lang, readonly, style, styleClass, tabIndex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:selectBooleanCheckbox>-Tag wird in HTML als <input>-Tag mit type="checkbox" dargestellt.

Beispiel:

```
Bereits registriert?
<h:selectBooleanCheckbox value="#{handler.flag}" />
```

<h:selectManyCheckbox>	HTMLSelectManyCheckbox
------------------------	------------------------

Syntax

```
<h:selectManyCheckbox [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
    [disabledClass="styleClass"] [ensabledClass="styleClass"]
    [layout="lineDirection|pageDirection"] [collectionType="type"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
layout	String	beliebig	Für lineDirection werden die Checkboxen horizontal, für pageDirection vertikal angeordnet.

HTML-Attribute

statisch	accesskey, border, dir, disabled, lang, readonly, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:selectManyCheckbox>-Tag wird in HTML als Tabelle von <input>-Elementen mit type="checkbox" dargestellt. Die einzelnen Tabellenzellen stellen die enthaltenen <f:selectItem>- und <f:selectItems>-Tags dar. Da mehrere Check-Boxen selektiert werden können, muss der Typ des value-Wertes ein Array sein.

Beispiel:

```
<h:selectManyCheckbox value="#{handler.farben}">
    <f:selectItem itemValue="rot" itemLabel="rot" />
    <f:selectItem itemValue="gelb" itemLabel="gelb" />
    <f:selectItem itemValue="blau" itemLabel="blau" />
</h:selectManyCheckbox>
```

<h:selectManyListbox>	HTMLselectManyListbox
-----------------------	-----------------------

Syntax

```
<h:selectManyListbox [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
    [disabledClass="styleClass"] [ensabledClass="styleClass"]
    [hideNoSelectionOption="true|false"] [collectionType="type"]
/>
```

HTML-Attribute

statisch	accesskey, dir, disabled, lang, readonly, size, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:selectManyListbox>-Tag wird in HTML als ein <select>-Tag mit multiple-Attribut dargestellt. Die Auswahl (je ein HTML-<option>-Tag) besteht aus den enthaltenen <f:selectItem>- oder <f:selectItems>-Tags. Falls die Select-Items vom Typ UISelectItemGroup sind, wird dies in HTML als <optgroup> dargestellt.

Beispiel:

```
<h:selectManyListbox value="#{handler.farben}"
    hideNoSelectionOption="#{handler.flag}">
    <f:selectItem itemLabel="Bitte auswählen"
        noSelectionOption="true" />
    <f:selectItem itemValue="rot" itemLabel="rot" />
    <f:selectItem itemValue="gelb" itemLabel="gelb" />
    <f:selectItem itemValue="blau" itemLabel="blau" />
</h:selectManyListbox>
```

<h:selectManyMenu>	HTMLSelectManyMenu
--------------------	--------------------

Syntax

```
<h:selectManyMenu [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
    [disabledClass="styleClass"] [ensabledClass="styleClass"]
    [hideNoSelectionOption="true|false"] [collectionType="type"]
/>
```

HTML-Attribute

statisch	accesskey, dir, disabled, lang, readonly, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:selectManyMenu>-Tag wird in HTML als ein <select>-Tag mit gesetztem **multiple**-Attribut und **size**-Attribut mit Wert 1 dargestellt. Die Auswahl (je ein HTML-<option>-Tag) besteht aus den enthaltenen <f:selectItem>- oder <f:selectItems>-Tags. Falls die Select-Items vom Typ UI-SelectItemGroup sind, wird dies in HTML als <optgroup> dargestellt.

Beispiel:

```
<h:selectManyMenu value="#{handler.farben}"
    hideNoSelectionOption="#{handler.flag}">
    <f:selectItem itemLabel="Bitte auswählen"
        noSelectionOption="true" />
    <f:selectItem itemValue="rot" itemLabel="rot" />
    <f:selectItem itemValue="gelb" itemLabel="gelb" />
    <f:selectItem itemValue="blau" itemLabel="blau" />
</h:selectManyMenu>
```

<code><h:selectOneListbox></code>	HTMLSelectOneListbox
---	-----------------------------

Syntax

```
<h:selectOneListbox [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
    [disabledClass="styleClass"] [ensabledClass="styleClass"]
    [hideNoSelectionOption="true|false"]
/>
```

HTML-Attribute

statisch	accesskey, dir, disabled, lang, readonly, size, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein `<h:selectOneListbox>`-Tag wird in HTML als ein `<select>`-Tag mit `size`-Attribut dargestellt, das als Wert die Anzahl der Auswahlelemente erhält (`size` kann aber überschrieben werden). Die Auswahl (je ein HTML-`<option>`-Tag) besteht aus den enthaltenen `<f:selectItem>` oder `<f:selectItems>`-Tags. Falls die Select-Items vom Typ `UISelectItemGroup` sind, wird dies in HTML als `<optgroup>` dargestellt.

Beispiel:

```
<h:selectOneListbox value="#{handler.farbe}">
    <f:selectItem itemValue="rot" itemLabel="rot" />
    <f:selectItem itemValue="gelb" itemLabel="gelb" />
    <f:selectItem itemValue="blau" itemLabel="blau" />
</h:selectOneListbox>
```

<h:selectOneMenu>	HtmlSelectOneMenu
-------------------	-------------------

Syntax

```
<h:selectOneMenu [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
    [disabledClass="styleClass"] [ensabledClass="styleClass"]
    [hideNoSelectionOption="true|false"]
/>
```

HTML-Attribute

statisch	accesskey, dir, disabled, lang, readonly, style, styleClass, tabindex, title
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein <h:selectOneMenu>-Tag stellt eine Drop-Down-Liste dar. Sie wird in HTML als <select>-Tag mit einem <size>-Attribut mit Wert 1 dargestellt. Enthaltene <f:selectItem>- und <f:selectItems>-Tags erzeugen in HTML <option>-Elemente.

Beispiele: Listing 4.12 auf Seite 83 und Listing 4.16 auf Seite 96.

<h:selectOneRadio>	HTMLSelectOneRadio
--------------------	--------------------

Syntax

```
<h:selectOneRadio [id="id"] [binding="componentBinding"]
    [rendered="true|false"] [value="value"] [label="label"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"] [immediate="true|false"]
    [disabledClass="styleClass"] [ensabledClass="styleClass"]
    [layout="lineDirection|pageDirection"]
/>
```

JSF-Attribute

Attribut	Typ	EL-Typ	Beschreibung
layout	String	beliebig	Für <code>lineDirection</code> werden die Auswahlknöpfe horizontal, für <code>pageDirection</code> vertikal angeordnet.

HTML-Attribute

statisch	accesskey, border, dir, disabled, lang, readonly, style, styleClass, tabindex, title.
dynamisch	onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect

Beschreibung

Ein `<h:selectOneRadio>`-Tag wird in HTML als Tabelle von `<input>`-Tags mit `type="radio"` dargestellt. Für `layout="pageDirection"` wird jede Auswahl in einer eigenen Tabellenzeile dargestellt, sonst alle Auswahlen in einer Zeile. Falls die Select-Items vom Typ `UISelectItemGroup` sind, werden diese in einer geschachtelten Tabelle dargestellt.

B.5 Kernbibliothek

Die Kernbibliothek (Core Tag Library) enthält Elemente, die unabhängig vom Renderer-Kit (HTML beim Standard-Renderer-Kit) sind. Sie repräsentieren Objekte, die man Komponenten hinzufügen kann. Zu diesen Objekten zählen Konvertierer und Validierer, aber auch Attribute, Listener, Facetten, Parameter und Select-Items und seit JSF 2.0 Ajax-Behaviors.

Da die Aufgabe von Konvertierern und Validierern die Bearbeitung und Prüfung von Werten ist, müssen die übergeordneten Komponenten von Konvertierern und Validierern Komponenten sein, die das `ValueHolder`-Interface implementieren. In der Regel sind dies Ein- und Ausgabekomponenten.

Die Select-Item-Tags werden innerhalb von `UISelectOne`- und `UISelectMany`-Komponenten verwendet.

Auch die anderen Tags sind in ihrer Verwendung nur innerhalb bestimmter anderer Tags erlaubt. Dies geht aus den Beispielen hervor.

<f:actionListener>

Syntax

```
<f:actionListener [type="className"] [binding="componentBinding"]
    [for="component"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
for	String	beliebig	Der qualifizierte Klassenname einer Klasse, die das ActionListener-Interface implementiert.
type	String	beliebig	Der qualifizierte Klassenname einer Klasse, die das ActionListener-Interface implementiert.

Beschreibung

Das <f:actionListener>-Tag erzeugt eine Instanz der angegebenen Klasse und registriert diese als Action-Listener der umgebenden Befehls-Komponente. Es muss entweder type oder binding verwendet werden.

Beispiele: Seite 119 mit type-Attribut, Seite 233 zusätzlich mit for-Attribut.

<f:ajax>

Syntax

```
<f:ajax [disabled="true|false"] [event="event"] [execute="executelds"] />
  [immediate="true|false"] [listener="/listenerMethod"] />
  [onevent="eventFunktion"] [onerror="errorFunktion"] />
  [render="true|false"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
event	String	beliebig	Der Event-Typ, für den die Ajax-Unterstützung registriert wird. Es werden DOM-Event-Namen und nicht JavaScript-Event-Namen verwendet, also z. B. <i>blur</i> und nicht <i>onblur</i> . Die unterstützten Events sind Browser-abhängig.
execute	String	beliebig	Liste von Komponenten-Ids, durch Leerzeichen getrennt, die an den Ajax-Verarbeitungsphasen teilnehmen (siehe Abschnitt 7.2.3). Zusätzlich sind die Literale <code>@this</code> , <code>@form</code> , <code>@all</code> und <code>@none</code> erlaubt.
listener	String	Methodenausdruck	Methode mit Parameter <code>AjaxBehaviorEvent</code> .
onevent	String	beliebig	Name der JavaScript-Funktion zur Event-Behandlung.
onerror	String	beliebig	Name der JavaScript-Funktion zur Fehlerbehandlung.
render	String	beliebig	Liste von Komponenten-Ids, durch Leerzeichen getrennt, die an der Ajax-Render-Phase teilnehmen (siehe Abschnitt 7.2.3). Zusätzlich sind die Literale <code>@this</code> , <code>@form</code> , <code>@all</code> und <code>@none</code> erlaubt.

Beschreibung

Das `<f:ajax>`-Tag versieht die verbundenen UI-Komponenten mit Ajax-Funktionalität. Die verbundenen Komponenten sind die Komponenten, die das `<f:ajax>` enthalten oder in ihm enthalten sind.

Beispiele: Code-Ausschnitt auf Seite 239 und Code-Ausschnitt auf Seite 242.

<f:attribute>

Syntax

```
<f:attribute name="name" value="value" />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
name	String	beliebig	Der Name des Attributs.
value	Object	beliebig	Der Wert des Attributs.

Beschreibung

Das <f:attribute>-Tag setzt ein generisches Attribut der umgebenden Komponente mit Name und Wert.

Beispiel:

```
<h:commandButton binding="#{handler.button}"
                  action="#{handler.action}">
    <f:attribute name="attr" value="4711" />
</h:commandButton>

@ManagedBean
public class Handler {

    private HtmlCommandButton button;

    public String action() {
        logger.info("Attribut: " + Integer.valueOf((String)
                                                 button.getAttributes().get("attr")));
        return "";
    }
    ...
}
```

<f:convertDateTime>

Syntax

```
<f:convertDateTime  
    [binding="converter"] [for="id"]  
    [dateStyle="default|short|medium|long|full"]  
    [timeStyle="default|short|medium|long|full"]  
    [pattern="pattern"] [type="time|date|both"]  
    [locale="locale"] [timeZone="timeZone"]  
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	beliebig	Bean-Property vom Typ javax.faces.convert.Converter
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
dateStyle	String	beliebig	Eine der vordefinierten, lokalisierungsabhängigen Datumsformate.
locale	String oder java.util.Locale	beliebig	Die zu verwendende Lokalisierung.
pattern	String	beliebig	Ein Pattern, dessen Syntax durch die Klasse java.text.SimpleDateFormat festgelegt wird. Siehe auch Tabelle 4.6 auf Seite 80.
timeStyle	String	beliebig	Eine der vordefinierten, lokalisierungsabhängigen Zeitformate.
timeZone	String oder java.util.TimeZone	beliebig	Die zu verwendende Zeitzone.
type	String	beliebig	Zu verwendender Teil der Datums/Zeitangabe. Default ist date, falls dateStyle gesetzt ist, time, falls timeStyle gesetzt ist, oder both, falls beide Style-Attribute gesetzt sind.

Beschreibung

Das <f:convertDateTime>-Tag erzeugt eine Instanz des registrierten Datum/Zeit-Konvertierers mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Ein- oder Ausgabekomponente.

Beispiel:

```
<h:inputText value="#{handler.geburtsdatum}">
    <f:convertDateTime pattern="dd. MMM yyyy" />
</h:inputText>
<h:inputText value="#{handler.geburtsdatum}">
    <f:convertDateTime locale="de_DE" dateStyle="full" />
</h:inputText>
<h:inputText value="#{handler.zeit}">
    <f:convertDateTime type="time" timeZone="Europe/Berlin"/>
</h:inputText>
```

<f:convertNumber>

Syntax

```
<f:convertNumber  
    [binding="converter"] [for="id"]  
    [pattern="pattern"] [locale="locale"]  
    [type="number|currency|percent"]  
    [minIntegerDigits="min"] [maxIntegerDigits="max"]  
    [minFractionDigits="min"] [maxFractionDigits="max"]  
    [groupingUsed="true|false"] [integerOnly="true|false"]  
    [currencyCode="currencyCode"] [currencySymbol="currencySymbol"]  
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	beliebig	Bean-Property vom Typ javax.faces.convert.Converter
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
currencyCode	String	beliebig	Der ISO 4217 Währungs-Code.
currencySymbol	String	beliebig	Das Währungssymbol.
groupingUsed	boolean	beliebig	Gruppierungszeichen werden verwendet, falls true.
integerOnly	boolean	beliebig	Beschränkung auf den ganzzahligen Anteil, falls true.
locale	String oder java.util.Locale	beliebig	Die zu verwendende Lokalisierung.
maxFraction-Digits	int	beliebig	Die maximale Anzahl Ziffern im gebrochenen Anteil.
maxInteger-Digits	int	beliebig	Die maximale Anzahl Ziffern im ganzzahligen Anteil.
minFraction-Digits	int	beliebig	Die minimale Anzahl Ziffern im gebrochenen Anteil.
minInteger-Digits	int	beliebig	Die minimale Anzahl Ziffern im ganzzahligen Anteil.

pattern	String	beliebig	Ein Pattern, dessen Syntax durch die Klasse <code>DecimalFormat</code> (Package <code>java.text</code>) festgelegt wird; siehe auch Tabelle 4.8 auf Seite 81.
type	String	beliebig	Angabe, ob als Zahl, Währung oder Prozentsatz formatiert werden soll.

Beschreibung

Das `<f:convertNumber>`-Tag erzeugt eine Instanz des registrierten Zahlen-Konvertierers mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Ein- oder Ausgabekomponente.

Beispiel:

```
<h:inputText value="#{numberHandler.bigDecimalValue}">
    <f:convertNumber type="currency" currencyCode="EUR" />
</h:inputText>
<h:inputText value="#{numberHandler.anzahl}">
    <f:convertNumber pattern="##### 'Teile'" />
</h:inputText>
```

<f:converter>

Syntax

```
<f:converter [converterId="converterId"] [binding="converter"] [for="id"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
converterId	String	beliebig	Die Id der Konvertiererkasse.
binding	String	beliebig	Bean-Property vom Typ Converter
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.

Beschreibung

Das <f:converter>-Tag erzeugt eine Instanz einer Klasse, die durch eine Id bestimmt wird. Die Klasse muss das Interface `javax.faces.convert.Converter` implementieren. Die Instanz wird an die umgebende Ein-/Ausgabekomponente gebunden.

Beispiel:

```
<h:inputText value="#{handler.kreditnr}">
  <f:converter converterId="kreditkartenkonvertierer" />
</h:inputText>
```

<f:event>

Syntax

```
<f:event listener="listenerMethod" type="event" />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
listener	String	Methoden-ausdruck	Listener-Methode mit Parameter ComponentSystemEvent.
type	String	beliebig	Name des Events, für das der Listener registriert wird. Es können die Literale preRenderComponent, preRenderView, postAddToView, preValidate, postValidate sowie der voll qualifizierte Klassenname von Events, die ComponentSystemEvent erweitern, verwendet werden.

Beschreibung

Das <f:event>-Tag registriert einen ComponentSystemEventListener für eine UI-Komponente.

Beispiele: Code-Ausschnitt auf Seite 144 und Code-Ausschnitt auf Seite 294.

<f:facet>

Syntax

```
<f:facet name="facetName">  
    JSF-Komponente  
</f:facet>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
name	String	keine EL möglich	Der Name der Facette.

Beschreibung

Das <f:facet>-Tag fügt die im Rumpf enthaltene Komponente als Facette der umgebenden Komponente hinzu. Facetten werden durch den Namen identifiziert. Komponenten lassen nur bestimmte Facetten zu. Der Rumpf darf nur aus einer Komponente bestehen, was durch ein <h:panelGroup> immer gewährleistet werden kann.

Beispiel:

```
<h:panelGrid columns="5">  
    <f:facet name="header">  
        <h:panelGroup>  
            <h:outputText value="Statistik Jahr " />  
            <h:outputText value="#{bb.jahr}" />  
        </h:panelGroup>  
    </f:facet>  
    ...
```

<f:loadBundle>

Syntax

```
<f:loadBundle [basename="basename"] var="var" />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
basename	String	beliebig	Der Name des Resource-Bundles.
var	String	keine EL möglich	Ein frei zu wählender Variablenbezeichner.

Beschreibung

Das <f:loadBundle>-Tag lädt das im basename-Attribut angegebene Resource-Bundle (eine oder mehrere Properties-Dateien) in ein neu erzeugtes Objekt vom Typ `java.util.Map`. Für die aktuelle Lokalisierung wird eine der Properties-Dateien geladen. Die Dateien werden im Klassenpfad gesucht und müssen die Dateinamenserweiterung „.properties“ aufweisen. Die Schlüssel können in EL-Ausdrücken wie normale Objekt-Properties der Variable verwendet werden.

Das <f:loadBundle>-Tag sollte aus den in Abschnitt 4.7.1 genannten Gründen nicht mehr verwendet werden. Alternativ erfolgt die Verwendung lokalisierter Ressourcen mit dem <resource-bundle>-Element in der JSF-Konfigurationsdatei.

Beispiel:

```
<f:loadBundle basename="alle-texte" var="text" />
<h:outputText value="#{text.begruebung}" />
```

```
<f:metadata>
```

Syntax

```
<f:metadata />
```

Beschreibung

Das `<f:metadata>`-Tag deklariert Meta-Daten für eine View und muss daher direkt innerhalb eines `<f:view>`-Tags verwendet werden. Das Tag darf nicht in einem Template verwendet werden. Die Verwendung in einem `<ui:composition>` oder `<ui:include>` ist erlaubt, aber nur in der im API-Doc beschriebenen Art und Weise.

Beispiel in Abschnitt 4.6.8.

<f:param>

Syntax

```
<f:param [id="id"] [binding="componentBinding"]
          [name="name"] value="value"
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property.
id	String	keine EL möglich	Die Id der Komponente.
name	String	beliebig	Der Name des Parameters.
value	String	beliebig	Der Wert des Parameters.

Beschreibung

Das <f:param>-Tag erzeugt eine Instanz der Klasse `UIParameter` mit entsprechendem Namen und Wert und bindet sie an die umgebende Komponente. Die gültigen Parameter sind in der vordefinierten EL-Variablen `param` vom Typ `Map` enthalten.

Beispiel:

```
<!-- Seite /param.xhtml -->
<h:commandButton value="Submit" action="/param-value.xhtml">
    <f:param name="attr" value="4711" />
</h:commandButton>

<!-- Seite /param-value.xhtml -->
...
Wert des Parameters: #{param['attr']}
```

Weiteres Beispiel: Verwendung mit <h:outputLink>-Tag auf Seite 418.

<f:phaseListener>

Syntax

```
<f:phaseListener [binding="componentBinding"] [type="Listener"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung eines Objekts, das das Interface PhaseListener implementiert.
type	String	beliebig	Voll qualifizierter Klassenname des zu registrierenden Phase-Listeners.

Beschreibung

Das <f:phaseListener>-Tag registriert einen Phase-Listener für die umgebende UIViewRoot-Komponente. Siehe auch Abschnitt 4.5.6.

Beispiel:

```
<f:phaseListener type="de.jsfpraxis.MyPhaseListener" />
```

<f:selectItem>

Syntax

```
<f:selectItem [id="id"] [binding="componentBinding"]
    [itemValue="itemValue"] [itemLabel="itemLabel"]
    [itemDescription="description"] [itemDisabled="true|false"]
    [value="selectItem"] [escape="true|false"]
    [noSelectionOption="true|false"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property.
escape	boolean	beliebig	Flag, ob Zeichen von itemLabel entwertet werden müssen. Default ist true.
id	String	keine EL möglich	Die Id der Komponente.
itemDescription	String	beliebig	Die Beschreibung des Items zur Verwendung in Werkzeugen.
itemDisabled	String	beliebig	Schalter, um Item zu sperren.
itemLabel	String	beliebig	Das Label des Items.
itemValue	Object	beliebig	Der Wert des Items.
noSelection-Option	boolean	beliebig	Flag, ob diese Auswahl die „Nicht-Auswahl“ darstellt.
value	SelectItem	Werte-ausdruck	Eine Instanz von javax.faces.model.SelectItem als Wert.

Beschreibung

Das <f:selectItem>-Tag erzeugt eine Instanz der Klasse `UISelectItem` mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Komponente.

Beispiel:

```
<h:selectOneMenu value="#{handler.tag}">
    <f:selectItem itemLabel="Montag" itemValue="1" />
    <f:selectItem itemLabel="Dienstag" itemValue="2" />
    <f:selectItem itemLabel="Mittwoch" itemValue="3" />
    ...
</h:selectOneMenu>
```

<f:selectItems>

Syntax

```
<f:selectItems [id="id"] [binding="componentBinding"] value="value"  
    [var="varName"] [itemValue="itemValue"] [itemLabel="itemLabel"]  
    [itemDescription="description"] [itemDisabled="true|false"]  
    [itemLabelEscaped="true|false"] [noSelectionValue="true|false"]  
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werteausdruck	Bindung einer Komponenten-Property.
id	String	keine EL möglich	Die Id der Komponente.
itemLabel-Escaped	boolean	beliebig	Flag, ob Zeichen von itemLabel entwertet werden müssen. Default ist true.
itemDescription	String	beliebig	Die Beschreibung des Items zur Verwendung in Werkzeugen.
itemDisabled	String	beliebig	Schalter, um Item zu sperren.
itemLabel	String	beliebig	Das Label des Items. Falls nicht angegeben, wird das Ergebnis der <code>toString()</code> -Methode zur Anzeige verwendet.
itemValue	Object	beliebig	Der Wert des Items.
noSelection-Value	boolean	beliebig	Flag, ob diese Auswahl die „Nicht-Auswahl“ darstellt.
value	SelectItem, Array, Collection oder Map	Werteausdruck	Ein oder mehrere Instanzen von SelectItem oder POJOs (ab JSF 2.0) als Wert.
var	String	Konstante	Iterationsvariable über value zur Verwendung in den item*-Attributen.

Beschreibung

Das `<f:selectItem>`-Tag erzeugt eine Instanz der Klasse `UISelectItems` mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Komponente. Arrays und Collections müssen aus `UISelectItems` bestehen, bei einer `Map` werden Schlüssel und Werte als Labels und Werte verwendet. Seit JSF 2.0 ist die Beschränkung auf `UISelectItems` aufgehoben, und es können beliebige POJOs verwendet werden.

Beispiel:

```
<h:selectOneMenu>
    <f:selectItems value="#{selectItemHandler.items}" var="item"
        itemValue="#{item}"
        itemLabel="#{item.label}"
        itemDescription="#{item.description}"
    />
</h:selectOneMenu>
```

<f:setPropertyActionListener>

Syntax

```
<f:setPropertyActionListener value="value" target="target" [for="id"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
target	String	Werte-ausdruck	Ziel-Property für den Wert von value.
value	String	beliebig möglich	Wert für das Ziel-Property.

Beschreibung

Das <f:setPropertyActionListener>-Tag registriert einen Action-Listener, der einen Wert in ein Property schreibt.

Beispiel:

```
<h:commandButton action="#{todoHandler.add}" value="Add">
    <f:setPropertyActionListener value="#{todoItem}"
        target="#{todoHandler.itemToAdd}" />
</h:commandButton>
```

<f:subView>

Syntax

```
<f:selectItems id="id" [binding="componentBinding"]  
    [rendered="true|false"]  
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property.
id	String	keine EL möglich	Die Id der Komponente.
rendered	boolean	beliebig	Angabe, ob die Komponente gerendert werden soll.

Beschreibung

Das <f:subView>-Tag erzeugt eine Instanz der Klasse `UINamingContainer` mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Komponente. Die Verwendung mit Facelets ist nicht sinnvoll, da Facelets <ui:include>-Tag bzw. Facelets Templating-Möglichkeiten weit über die Möglichkeiten von <f:subView> hinausgehen.

<f:validateBean>

Syntax

```
<f:validateBean [binding="component"] [for="id"]
    [disabled="true|false"] [validationGroups="groups"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werteausdruck	Bindung einer Komponenten-Property vom Typ BeanValidator.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, ob das umgebende oder die enthaltenen Eingaben validiert werden sollen.
validation-Groups	String	beliebig	Liste von voll qualifizierten Klassennamen, durch Komma getrennt.

Beschreibung

Das <f:validateBean>-Tag delegiert die Validierung von Bean-Properties an die Implementierung der Bean-Validierung (JSR 303), siehe Abschnitt 4.4.9.

Beispiele ebenfalls in Abschnitt 4.4.9.

<f:validateDoubleRange>

Syntax

```
<f:validateDoubleRange [binding="component"] [for="id"]
    [disabled="true|false"] [minimum="min"] [maximum="max"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werteausdruck	Bindung einer Komponenten-Property vom Typ DoubleRangeValidator.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, dass nicht validiert werden soll.
maximum	double	beliebig	Der erlaubte Maximalwert.
minimum	double	beliebig	Der erlaubte Minimalwert.

Beschreibung

Das <f:validateDoubleRange>-Tag erzeugt eine Validierer-Instanz für Double-Werte mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Eingabekomponente. Von den Attributen `minimum` und `maximum` muss mindestens eines verwendet werden.

Beispiele:

```
<h:inputText value="#{handler.skalierungsfaktor}">
    <f:validateDoubleRange minimum="0.0" maximum="1.0" />
<h:inputText>
<h:inputText value="#{handler.preis}">
    <!-- Preis darf nicht negativ sein -->
    <f:validateDoubleRange minimum="0.0" />
<h:inputText>
```

<f:validateLength>

Syntax

```
<f:validateLength [binding="component"] [for="id"]  
    [disabled="true|false"] [minimum="min"] [maximum="max"]  
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werteausdruck	Bindung einer Komponenten-Property vom Typ LengthValidator.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, dass nicht validiert werden soll.
maximum	int	beliebig	Die maximal erlaubte Anzahl von Zeichen.
minimum	int	beliebig	Die minimal erlaubte Anzahl von Zeichen.

Beschreibung

Das <f:validateLength>-Tag erzeugt eine Validierer-Instanz für Eingabelängen mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Eingabekomponente. Von den Attributen `minimum` und `maximum` muss mindestens eines verwendet werden.

Beispiel:

```
<h:inputText value="#{handler.passwort}">  
    <f:validateLength minimum="8" maximum="12" />  
<h:inputText>  
<h:inputText value="#{handler.postleitzahl}">  
    <f:validateLength minimum="5" maximum="5" />  
<h:inputText>
```

<f:validateLongRange>

Syntax

```
<f:validateLongRange [binding="component"] [for="id"]
    [disabled="true|false"] [minimum="min"] [maximum="max"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property vom Typ LongRangeValidator.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, dass nicht validiert werden soll.
maximum	long	beliebig	Der erlaubte Maximalwert.
minimum	long	beliebig	Der erlaubte Minimalwert.

Beschreibung

Das <f:validateLongRange>-Tag erzeugt eine Validierer-Instanz für Long-Werte mit der in den Attributen angegebenen Konfiguration und bindet sie an die umgebende Eingabekomponente. Von den Attributen `minimum` und `maximum` muss mindestens eines verwendet werden.

Beispiel:

```
<h:inputText value="#{handler.umsatz}">
    <f:validateLongRange minimum="100000" maximum="100000000" />
<h:inputText>
```

<f:validateRegex>

Syntax

```
<f:validateRegex [binding="component"] [for="id"]
    [disabled="true|false"] [pattern="pattern"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property vom Typ <code>RegexpValidator</code> .
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, dass nicht validiert werden soll.
pattern	String	beliebig	Ein regulärer Ausdruck.

Beschreibung

Das `<f:validateRegex>`-Tag registriert einen Validator für reguläre Ausdrücke. Die Instanz wird an die umgebende Komponente gebunden.

Beispiel:

```
<h:inputText value="#{handler.email}" required="true">
    <f:validateRegex pattern=".+@.+\..+" />
</h:inputText>
```

<f:validateRequired>

Syntax

```
<f:validateRequired [binding="component"] [for="id"]
    [disabled="true|false"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werteausdruck	Bindung einer Komponenten-Property vom Typ RequiredValidator.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, dass nicht validiert werden soll.

Beschreibung

Das <f:validateRequired>-Tag erzeugt einen Validator, der eine Eingabe erzwingt. Es entspricht daher dem **required**-Attribut der Eingabekomponenten. Das <f:validateRequired>-Tag kann innerhalb einer Eingabekomponente enthalten sein oder mehrere Eingabekomponenten umschließen. Zusätzlich ist die Verwendung in zusammengesetzten Komponenten möglich.

Beispiele in Abschnitt 4.4.5.

<f:validator>

Syntax

```
<f:validator [binding="component"] [for="id"]
             [disabled="true|false"] [validatorId="validatorId"]
             />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property vom Typ Validator.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
disabled	boolean	beliebig	Flag, das angibt, dass nicht validiert werden soll.
validatorId	String	beliebig	Die Id der Validiererkasse.

Beschreibung

Das <f:validator>-Tag erzeugt eine Instanz des **Validator** -Interface. Das Tag lässt sich innerhalb einer Eingabekomponente oder mehrere Eingabekomponenten umschließend verwenden; es sind entweder **validatorId** oder **binding** zu verwenden.

Beispiel in Abschnitt 4.4.7.

<f:valueChangeListener>

Syntax

```
<f:valueChangeListener [binding="component"] [for="id"]  
[type="className"]  
/>
```

Syntax

```
<f:validator validatorId="validatorId" />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werteausdruck	Bindung einer Komponenten-Property vom Typ ValueChangeListener.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.
type	String	beliebig	Der voll qualifizierte Name einer Klasse, die das ValueChangeListener-Interface implementiert.

Beschreibung

Das <f:valueChangeListener>-Tag registriert einen Value-Change-Listener und bindet diesen an die umgebende Komponente.

Beispiel:

```
<h:selectOneMenu value="#{handler.familienstand}">  
    <f:valueChangeListener  
        type="de.jsfpraxis.Familienstandaenderung" />  
</h:selectOneMenu>
```

<f:verbatim>

Syntax

```
<f:verbatim [escape="true|false"] [rendered="true|false"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
escape	boolean	beliebig	Sonderzeichen werden als HTML- und XML-Entities dargestellt, falls true.
rendered	boolean	beliebig	Angabe, ob die Komponente gerendert werden soll.

Beschreibung

Das <f:verbatim>-Tag erzeugt eine Instanz der Klasse UIOutput, deren Wert die Auswertung des Rumpfes ist. Die Instanz wird an die umgebende Komponente gebunden.

Das <f:verbatim>-Tag wurde in JSF 1.x verwendet, um HTML-Code verwenden zu können. Mit Facelets (JSF 2.0) wurde die Verwendung von <f:verbatim> sinnlos, da man HTML-Tags direkt verwenden kann.

Beispiel:

```
<f:verbatim>
    <h1>Regel</h1>
    <p>Mit JSF 2.0 nicht mehr verwenden.</p>
</f:verbatim>
```

<f:view>

Syntax

```
<f:view [afterPhase="listener"] [beforePhase="listener"]
        [locale="locale"] [renderKit="locale"]
    />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
afterPhase	String	Methodenausdruck	Listenermethode mit Parameter PhaseEvent, die außer nach der Phase 1 nach jeder Phase des Lebenszyklus aufgerufen wird.
beforePhase	String oder Locale	beliebig	Listenermethode mit Parameter PhaseEvent, die außer vor der Phase 1 vor jeder Phase des Lebenszyklus aufgerufen wird.
locale	String oder Locale	beliebig	Die zu verwendende Lokalisierung.
renderKitId	String	beliebig	Id des für diese Seite zu verwendenden Render-Kits.

Beschreibung

Das <f:view>-Tag erzeugt eine Instanz der Klasse `UIViewRoot` und setzt die Lokalisierung. Alle enthaltenen Elemente werden zu Sohn-Komponenten der View.

Beispiel:

```
<f:view locale="#{user.locale}">
    ...
</f:view>
```

<f:viewParam>

Syntax

```
<f:viewParam [id="id"] [binding="componentBinding"]
    [value="value"] [maxlength="length"] [for="id"]
    [required="true|false"] [requiredMessage="requiredMessage"]
    [converter="converter"] [converterMessage="converterMessage"]
    [validator="validatorMethod"] [validatorMessage="validatorMessage"]
    [valueChangeListener="listenerMethod"]
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte- ausdruck	Bindung einer Komponenten-Property vom Typ UIComponent.
for	String	beliebig	Id der zusammengesetzten Komponente, die das Tag enthält.

Beschreibung

Das <f:viewParameter>-Tag deklariert Parameter für eine View als Instanzen von **UIViewParameter** und darf nur innerhalb eines <f:metadata>-Tags verwendet werden. Da **UIViewParameter** eine Unterklasse von **UIInput** ist, dürfen alle Attribute von **UIInput** verwendet werden.

Beispiele in Abschnitt 4.6.8.

B.6 Facelets

Die Facelets-Bibliothek stellt Komponenten für einen mächtigen Templating-Mechanismus und weitere, ergänzende Komponenten bereit. Da Facelets in JSF 2.0 die Standardseitenbeschreibungssprache ist, müssen keine Konfigurationseinstellungen vorgenommen werden.

Falls Sie Facelets bereits vor Version 2.0 eingesetzt und Klassen des Paketes `com.sun.facelets` bzw. dessen Subpackages direkt verwendet haben, ist dies mit JSF 2.0 nicht kompatibel. Um die Package-Migration zu vermeiden, muss das alte Facelets-Jar explizit deployt und der Kontextparameter `DISABLE_FACELET_JSF_VIEWHANDLER` auf `true` gesetzt werden.

<ui:component>

Syntax

```
<ui:component [id="id"] [binding="componentBinding"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property vom Typ UIComponent.
id	String	beliebig	Id der Komponente.

Beschreibung

Das <ui:component>-Tag definiert eine Komponente und bindet sie in den Komponentenbaum ein. Text und andere Komponenten außerhalb des Tags werden ignoriert. Die definierte Komponente wird über <ui:include> verwendet. Eine bessere Möglichkeit der Definition und Wiederverwendung von Komponenten bieten zusammengesetzte Komponenten.

Beispiel:

```
Dieser Text wird ignoriert.  
<ui:component binding="#{handler.component}">  
  <p>Hello World</p>  
</ui:component>  
Dieser Text wird ebenfalls ignoriert.
```

<ui:composition>

Syntax

```
<ui:composition [template="template"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
template	String	beliebig	Dateiname (Pfad) des Templates.

Beschreibung

Das <ui:composition>-Tag definiert eine Komponente und bindet sie in den Komponentenbaum ein. Wird das template-Attribut verwendet, definiert <ui:composition> den Template-Client, wie in Abschnitt 6.1 erläutert. Da mehrere <ui:composition>-Tags dasselbe Template verwenden können, ist dies die Grundlage für das einheitliche Layout einer Anwendung. Text außerhalb des Tags wird ignoriert.

Beispiel:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <body>
    Dieser Text wird ignoriert.
    <ui:composition template="/layout.xhtml">
      <ui:define name="title">#{msg.title}</ui:define>
      <ui:define name="heading">#{msg.heading}</ui:define>
      <ui:define name="content">
        <ui:include src="inhalt.xhtml" />
      </ui:define>
    </ui:composition>
    Dieser Text wird ebenfalls ignoriert.
  </body>
</html>
```

<ui:debug>

Syntax

```
<ui:debug [hotkey="key"] [rendered="true|false"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
hotkey	String	Konstante	Einzelner Buchstabe, der mit Strg und Shift zusammen das JSF-Debug-Fenster öffnet. Default ist "d".
rendered	boolean	beliebig	Angabe, ob die Komponente gerendert werden soll.

Beschreibung

Das <ui:debug>-Tag definiert die Debug-Komponente. Die Komponente öffnet durch eine definierbare Tastenkombination (Default: Strg-Shift-d) ein Fenster, das Debug-Informationen anzeigt. Dazu gehören der Komponentenbaum sowie Request-Parameter und Managed Beans.

Beispiel:

```
<ui:debug hotkey="x" rendered="#{not handler.inProduction}" />
```

<ui:decorate>

Syntax

```
<ui:decorate [template="template"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
template	String	beliebig	Dateiname (Pfad) des Templates.

Beschreibung

Das <ui:decorate>-Tag verhält sich identisch zum <ui:composition>-Tag, außer dass umgebender Text und Markup nicht ignoriert werden. Der Zwang zu umgebendem Markup ergibt sich z. B. bei geschachtelten Templates. Bei der Verwendung zweier ineinander geschachtelter <ui:composition>-Tags würde das äußere ignoriert werden.

Beispiel:

```
<body>
    <h:form>
        <ui:composition template="/template-outer.xhtml">
            <ui:define name="header">Client Header</ui:define>
            <ui:define name="body">
                Der innere Bereich:
                <ui:decorate template="/template-inner.xhtml">
                    <ui:define name="inner">Inner</ui:define>
                </ui:decorate>
            </ui:define>
            <ui:define name="footer">Client Footer</ui:define>
        </ui:composition>
    </h:form>
</body>
```

<ui:define>

Syntax

```
<ui:define name="name" />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
name	String	beliebig	Zuweisung eines Names für den Inhalt des Tags.

Beschreibung

Das <ui:define>-Tag definiert einen Inhalt zur Verwendung in Templates. Das Tag kann innerhalb der Tags <ui:composition>, <ui:component>, <ui:decorate> und <ui:fragment> verwendet werden. Der so definierte Inhalt wird durch das <ui:insert>-Tag in eine Seite eingefügt.

Beispiel:

```
<ui:define name="body">
    <p>Der Seitenrumpf</p>
    ...
</ui:define>
```

```
<ui:fragment>
```

Syntax

```
<ui:fragment [id="id"] [binding="componentBinding"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
binding	String	Werte-ausdruck	Bindung einer Komponenten-Property vom Typ UIComponent.
id	String	beliebig	Id der Komponente.

Beschreibung

Das `<ui:fragment>`-Tag definiert eine Komponente und bindet sie in den Komponentenbaum ein. Es verhält sich somit wie das `<ui:component>`-Tag, ignoriert aber umgebenden Text und Markup nicht.

Beispiel:

```
Dieser Text wird nicht ignoriert.  
<ui:fragment binding="#{handler.component}">  
    <p>Hello World</p>  
</ui:fragment>  
Dieser Text wird ebenfalls nicht ignoriert.
```

<ui:include>

Syntax

```
<ui:include src="file" />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
src	String	beliebig	Name der zu inkludierenden Datei.

Beschreibung

Das <ui:include>-Tag inkludiert den Inhalt einer Facelets-Datei. Die Angabe des Dateinamens erfolgt relativ zur inkludierenden Datei.

Beispiel:

```
<ui:define name="footer">
    <ui:include src="footer.xhtml" />
</ui:define>
```

```
<ui:insert>
```

Syntax

```
<ui:insert [name="name"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
name	String	beliebig	Name des einzufügenden Inhalts.

Beschreibung

Das `<ui:insert>`-Tag fügt Inhalt in eine Template ein. Das `name`-Attribut referenziert den über ein `<ui:define>` definierten Inhalt. Wird das Attribut nicht verwendet, wird der ganze Template-Client eingefügt.

Beispiel:

```
<body>
    <h:form>
        <div style="border: 1px solid;">
            <ui:insert name="header">Header-Platzhalter</ui:insert>
        </div>
        <div style="border: 1px solid;">
            <ui:insert name="body">Body-Platzhalter</ui:insert>
        </div>
        <div style="border: 1px solid;">
            <ui:insert name="footer">Footer-Platzhalter</ui:insert>
        </div>
    </h:form>
</body>
```

<ui:param>

Syntax

```
<ui:param [name="name"] [value="wert"] />
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
name	String	beliebig	Name des Parameters.
value	String	beliebig	Wert des Parameters.

Beschreibung

Das <ui:param>-Tag erlaubt die Übergabe von Parametern in eine inkludierte Datei (<ui:include>) oder ein Template (<ui:composition> oder <ui:decorator>).

Beispiel:

```
<ui:include src="/included.xhtml">
    <ui:param name="param" value="Der Wert des Parameters" />
</ui:include>
```

```
<ui:remove>
```

Syntax

```
<ui:remove />
```

Attribute

Das <ui:remove>-Tag erlaubt keine Attribute.

Beschreibung

Das <ui:remove>-Tag entfernt Inhalte (einschließlich sich selbst) aus einer Seite. Es kann daher zum „Auskommentieren“ verwendet werden. Die Verwendung der XML-Kommentarzeichen „“ funktioniert alternativ nur, wenn der Kontextparameter FACELETS_SKIP_COMMENTS auf **true** gesetzt ist.

Beispiel:

```
<h:outputText value="eins" /><br />
<h:outputText value="zwei" /><br />
<h:outputText value="drei" /><br />
<ui:remove>
    <h:outputText value="vier" /><br />
    <h:outputText value="fünf" /><br />
    <h:outputText value="sechs" /><br />
</ui:remove>
<h:outputText value="sieben" /><br />
<!-- nur, falls FACELETS_SKIP_COMMENTS true ist
    &lt;h:outputText value="acht" /&gt;&lt;br /&gt;
--&gt;
&lt;h:outputText value="neun" /&gt;&lt;br /&gt;</pre>
```

<ui:repeat>

Syntax

```
<ui:repeat value="Wert" var="variable"  
          [offset="offset"] [size="anzahl"]  
          [step="schrittweite"] [varStatus="variable"]  
/>
```

Attribute

Attribut	Typ	EL-Typ	Beschreibung
value	String	beliebig	Wert, über den iteriert wird.
var	String	beliebig	Die Iterationsvariable
offset	String	beliebig	Anzahl der zu überspringenden Elemente bei Beginn der Iteration.
size	String	beliebig	Anzahl der Iterationen; muss kleiner als die Collection-Größe sein.
step	String	beliebig	Die Schrittweite der Iteration.
varStatus	String	beliebig	Eine Statusvariable mit den Properties: begin, end, index, step, even, odd, first und last. Bug in Mojarra 2.0.3.

Beschreibung

Das <ui:repeat>-Tag realisiert eine Iteration über ein Array, eine Liste oder ein ResultSet. Es verhält sich ähnlich wie die <h:dataTable>- und <c:forEach>-Tags.

Beispiel:

```
<ul>  
  <ui:repeat var="data" value="#{tagHandler.digits}">  
    <li> #{data} </li>  
  </ui:repeat>  
</ul>
```

Anhang C

URL-Verzeichnis

- [URL-BE] Hans Bergsten, Improving JSF by Dumping JSP
<http://www.onjava.com/pub/a/onjava/2004/06/09/jsf.html>
- [URL-BLZ] Bankleitzahlen der Bundesbank
http://www.bundesbank.de/zahlungsverkehr/zahlungsverkehr_bank-leitzahlen_download.php
- [URL-CACTUS] Cactus Test-Framework
<http://jakarta.apache.org/cactus/index.html>
- [URL-DEVSTUDIO] Red-Hat Developer-Studio
<http://www.redhat.com/developers/rhds/index.html>
- [URL-DERBY] Apache Derby
<http://db.apache.org/derby/>
- [URL-DI] Dependency Injection, Inversion of Control
<http://www.martinfowler.com/articles/injection.html>
- [URL-DOJO] Dojo-Framework (JavaScript)
<http://www.dojotoolkit.org/>
- [URL-ECLIPSE] Eclipse Homepage
<http://www.eclipse.org/>
- [URL-ELINK] EclipseLink
<http://www.eclipse.org/eclipselink/>
- [URL-EEJB] Embeddable EJB-3-Container (JBoss)
<http://labs.jboss.com/jbossejb3/>
- [URL-FL] Facelets
<https://facelets.dev.java.net/>

- [URL-FB] Firebug
<http://getfirebug.com/>
- [URL-GF] GlassFish
<https://glassfish.dev.java.net/>
- [URL-GFADMIN] GlassFish Enterprise Server v3 Administration Guide
<http://docs.sun.com/app/docs/doc/820-7692/>
- [URL-GFP] GlassFish Plugins
<https://glassfishplugins.dev.java.net/>
- [URL-GFWIKI] GlassFish Wiki
<http://docs.sun.com/app/docs/doc/820-7692/>
- [URL-HIBERNATE] Hibernate
<http://www.hibernate.org>
- [URL-HIBCOMP] Hibernate Compatibility Matrix
<http://www.hibernate.org/6.html#A3>
- [URL-HIBTOOLS] Hibernate Tools
<http://www.hibernate.org/255.html>
- [URL-HIBVAL] Hibernate Validator
<http://validator.hibernate.org/>
- [URL-HTMLUNIT] HtmlUnit
<http://htmlunit.sourceforge.net/>
- [URL-ICE] ICEfaces Komponentenbibliothek
<http://www.icefaces.org>
- [URL-ISO-639] ISO Language Codes
http://www.loc.gov/standards/iso639-2/php/code_list.php
- [URL-ISO-3166] ISO Country Codes
http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm
- [URL-ITEXT] iText PDF
<http://www.itextpdf.com/>
- [URL-JA4J] JBoss Ajax4jsf
<http://labs.jboss.com/jbossajax4jsf/>
- [URL-JBPM] JBoss jBPM
<http://labs.jboss.com/jbossjbpm/>
- [URL-JBPMREF] Benutzerhandbuch jBPM und jPDL
<http://docs.jboss.com/jbpm/v3/userguide/>

- [URL-JBTOOLS] JBoss Tools
<http://www.jboss.org/tools>
- [URL-JCAPTCHA] JCaptcha
<http://jcaptcha.sourceforge.net/>
- [URL-JCP] Java Community Process
<http://www.jcp.org>
- [URL-JEM] JEMS-Installer
<http://labs.jboss.com/jemsinstaller/>
- [URL-JRF] JBoss-RichFaces Komponentenbibliothek
<http://labs.jboss.com/jbosssrichfaces/>
- [URL-JSE] Java Platform, Standard Edition
<http://java.sun.com/docs/books/jls/index.html>
- [URL-JSFUNIT] JSFUnit
<http://www.jboss.org/jsfunit>
- [URL-JSR52] JSR 52: JavaServer Pages Standard Tag Library Specification
<http://jcp.org/en/jsr/detail?id=52>
- [URL-JSR53] JSR 53: Servlet 2.3 and JavaServer Pages 1.2 Specifications
<http://jcp.org/en/jsr/detail?id=53>
- [URL-JSR127] JSR 127: JavaServer Faces 1.0 und 1.1
<http://jcp.org/en/jsr/detail?id=127>
- [URL-JSR154] JSR 154: Servlet 2.4 und 2.5
<http://jcp.org/en/jsr/detail?id=154>
- [URL-JSR220] JSR 220: Enterprise JavaBeans 3.0
<http://jcp.org/en/jsr/detail?id=220>
- [URL-JSR244] JSR 244: Java Platform, Enterprise Edition 5
<http://jcp.org/en/jsr/detail?id=244>
- [URL-JSR245] JSR 245: JavaServer Pages 2.1
<http://jcp.org/en/jsr/detail?id=245>
- [URL-JSR245II] JSR 245: JavaServer Pages 2.1, Part II
http://jcp.org/aboutJava/communityprocess/maintenance/jsr245/245-MR2_2.html
- [URL-JSR250] JSR 250: Common Annotations for the Java Platform
<http://www.jcp.org/en/jsr/detail?id=250>
- [URL-JSR252] JSR 252: JavaServer Faces 1.2
<http://www.jcp.org/en/jsr/detail?id=252>

- [URL-JSR254] JSR 254: Java Platform, Enterprise Edition (Java EE) 5
<http://www.jcp.org/en/jsr/detail?id=254>
- [URL-JSR276] JSR 276: Design-Time Metadata for JavaServer Faces Components
<http://www.jcp.org/en/jsr/detail?id=276>
- [URL-JSR299] JSR 299: Contexts and Dependency Injection for Java EE Platform
<http://www.jcp.org/en/jsr/detail?id=299>
- [URL-JSR303] JSR 303: Bean Validation
<http://jcp.org/en/jsr/detail?id=303>
- [URL-JSR313] JSR 313: Java Platform, Enterprise Edition 6
<http://jcp.org/en/jsr/detail?id=313>
- [URL-JSR314] JSR 314: JavaServer Faces 2.0
<http://jcp.org/en/jsr/detail?id=314>
- [URL-JSR315] JSR 315: Servlet 3.0 Specification
<http://jcp.org/en/jsr/detail?id=315>
- [URL-JSR316] JSR 313: Java Platvorm, Enterprise Edition 6
<http://jcp.org/en/jsr/detail?id=316>
- [URL-JSR330] JSR 330: Dependency Injection for Java
<http://jcp.org/en/jsr/detail?id=330>
- [URL-JUNIT] JUnit
<http://www.junit.org/>
- [URL-JXL] Java Excel API
<http://jexcelapi.sourceforge.net/>
- [URL-MIME] Mime Types
<http://www.ltsw.se/knbase/internet/application.htm>
- [URL-MOJ] Project Mojarra
<https://javaserverfaces.dev.java.net>
- [URL-MOJCP] Context-Parameter der Referenz-Implementierung (Mojarra)
<http://wiki.glassfish.java.net/Wiki.jsp?page=JavaServerFaces-RI#section-JavaServerFacesRI-WhatContextParametersAreAvailableAndWhatDoTheyDo>
- [URL-MOJF] Summary of New Features in JavaServer Faces 1.2 Technology
http://java.sun.com/developer/technicalArticles/J2EE/jsf_12/
- [URL-NICE] Niceforms JavaScript-Bibliothek
<http://www.badboy.ro/articles/2007-01-30/niceforms/>

- [URL-OAA] OpenAjax Alliance
<http://www.openajax.org>
- [URL-RA] Request Aggregation in JSF 2 Ajax
<http://weblogs.java.net/blog/driscoll/archive/2009/10/19/request-aggregation-jsf-2-ajax>
- [URL-RULES] JBoss Rules
<http://labs.jboss.com/jbossrules/>
- [URL-RFC] RFC Editor Homepage
<http://www.rfc-editor.org/>
- [URL-SEAM] JBoss Seam
<http://www.seamframework.org/>
- [URL-SEL] Selenium Web Application Testing System
<http://seleniumhq.org/>
- [URL-SPRING] Spring-Framework
<http://www.springframework.org/>
- [URL-TCK] Java Compatibility Test Tools
<http://www.jcp.org/en/resources/tdk>
- [URL-TESTNG] TestNG
<http://www.testng.org/doc/>
- [URL-U4S] Unicode 4.0 Support in J2SE 1.5
http://weblogs.java.net/blog/joconner/archive/2004/04/unicode_40_supp.html
- [URL-UEL] Unified Expression Language
<http://java.sun.com/products/jsp/reference/techart/unified-EL.html>
- [URL-WELD] Weld Home
<http://seamframework.org/Weld>
- [URL-XMLHTTP] XMLHttpRequest
<http://www.w3.org/TR/XMLHttpRequest/>

Lizenziert für l.gruenwoldt@web.de.

© 2010 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

Literaturverzeichnis

- [Amm03] Dirk Ammelburger. *XML — Grundlagen der Sprache und Anwendungen in der Praxis*. Hanser Verlag, 2003.
- [BK07a] Christian Bauer und Gavin King. *Java Persistence mit Hibernate*. Hanser, 2007.
- [BK07b] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications, 2007.
- [BR08] Martin Backschat und Bernd Rücker. *Enterprise JavaBeans 3.0*. Elsevier, 2008.
- [Bur01] Eric M. Burke. *Java and XSLT*. O'Reilly, 2001.
- [Bur10] Eric M. Burke. *JavaServer Faces 2.0, The Complete Reference*. McGraw-Hill, 2010.
- [EL08] Werner Eberling und Jan Leßner. *Enterprise JavaBeans 3*. Hanser, 2008.
- [Fil09] Demetrio Filocamo. *JBoss RichFaces 3.3*. Packt Publishing, 2009.
- [Fla01] David Flanagan. *JavaScript — The Definitive Guide*. O'Reilly, 4. Auflage, 2001.
- [Fla05] David Flanagan. *Java in a Nutshell*. O'Reilly, 5. Auflage, 2005.
- [HM05] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly, 3. Auflage, 2005.
- [Hun01] Jason Hunter. *Java Servlet Programming*. O'Reilly, 2. Auflage, 2001.
- [JH04] Rod Johnson and Juergen Hoeller. *J2EE Development without EJB*. Wiley Publishing, 2004.
- [Kat08] Max Katz. *Practical RichFaces*. Apress, 2008.
- [Low07] Bruno Lowagie. *iText in Action*. Manning Publications, 2007.
- [McL01] Brett McLaughlin. *Java & XML*. O'Reilly, 2. Auflage, 2001.

- [Mey00] Eric A. Meyer. *Cascading Style Sheets — The Definitive Guide*. O'Reilly, 2000.
- [MK00] Chuck Musciano and Bill Kennedy. *HTML & XHTML, The Definitive Guide*. O'Reilly, 4. Auflage, 2000.
- [MW07] Bernd Müller und Harald Wehr. *Java Persistence API mit Hibernate*. Addison-Wesley, 2007.
- [Paw02] Dave Pawson. *XSL-FO*. O'Reilly, 2002.
- [PRL07] Debu Panda, Reza Rahman, and Derek Lane. *EJB3 in Action*. Manning Publications, 2007.
- [Rec06] Peter Rechenberg. *Technisches Schreiben — (nicht nur) für Informatiker*. Hanser Verlag, 3. Auflage, 2006.
- [Sic04] Bastian Sick. *Der Dativ ist dem Genitiv sein Tod. Ein Wegweiser durch den Irrgarten der deutschen Sprache*. Kiepenheuer & Witsch, Spiegel Online, 2004.
- [Tid01] Doug Tidwell. *XSLT*. O'Reilly, 2001.
- [vdV03] Eric van der Vlist. *XML Schema*. O'Reilly, 2003.

Sachverzeichnis

<a4j:commandButton> 257–259
<a4j:commandLink> 259
<a4j:log> 260
<a4j:outputPanel> 256
<a4j:poll> 259
<a4j:region> 257
<a4j:repeat> 264
<a4j:support> 254, 281
<absolute-ordering> 171, 177
accept 384
acceptCharset 384
accesskey 384
Acegi 129
<action-listener> 172
Action-Listener-Methode 117
Action-Methode 42, 117
 Rückgabetyp 42
 zur direkten Navigation 142
ActionEvent 42, 115, 283, 397, 398
actionListener 117
ActionListener 116, 117, 433
ActionSource2 146
addClientBehavior() 243
addMessage() 113
<after> 178
afterPhase() 129
Ajax
 Execute-Teil 245
 Render-Teil 245
AjaxBehavior 242
AjaxBehaviorEvent 244, 434
AjaxBehaviorListener 244
ajaxSingle 258
alt 384
Anfragewerte 39
Anwendungslogik 41
Applet 1
<application> 149, 171
application
 vordefinierte EL-Variable 48
<application-extension> 172
<application-factory> 175
application.xml 319
applicationScope
 vordefinierte EL-Variable 48
@ApplicationScoped 70, 173, 328,
 378
ArrayDataModel 128
@AssertFalse 99
@AssertTrue 99
Asynchron JavaScript and XML
 siehe AJAX
Attribut
 HTML- 384
 pass-through 384
<attribute> 174, 175, 179
<auth-constraint> 169
Authentication 292

- Authentifizierung 292
- Authorization 292
- autocomplete 384
- Automatic Dirty Checking 299
- Autorisierung 292
- AWT 16

- <base-name> 151
- Bean Validation 98
 - benutzerdefinierte Constraints 102
- BeanValidator 88
- Bearbeitungsmodell 36
- <before> 178
- beforePhase() 129
- <behavior> 171, 179
- Behavior 242, 377
- <behavior-class> 179
- <behavior-extension> 179
- <behavior-id> 179
- BehaviorBase 377
- bgcolor 385
- border 385
- Brett 13
- Business Process *siehe* Geschäftsprozess
- Business Process Execution Language *siehe* BPEL

- <c:forEach> 194, 196
- <c:if> 196
- Calendar 147
- Cascading Style Sheets *siehe* CSS
- cc 48, 229, 232
 - vordefinierte EL-Variable 48
- CDI 324
- cellpadding 385
- cellspacing 385
- charset 385

- class 389
- <client-behaviour-renderer> 174
- ClientBehavior 242
- ClientBehaviorHolder 243
- collectionType 391
- cols 385
- @Column 337
- <component> 171, 174
- component
 - vordefinierte EL-Variable 48
- Component Behavior Model *siehe* Komponente, Verhaltensmodell
- Component-Binding 64
- <component-class> 174
- <component-extension> 174
- <component-type> 174
- ComponentSystemEvent 130–132, 377, 442
- composite
 - Präfix 229
- <composite:actionSource> 231
- <composite:implementation> Inhalt 229
- <composite:implementation> 229
- <composite:interface> 229
- Confidentiality 292
- CONFIG_FILES 163, 166
- @Constraint 102
- <context-param> 162
- Contexts and Dependency Injection *siehe* CDI
- Conversation 328, 331, 332
- @ConversationScoped 328, 331
- <converter> 87, 171, 175
- Converter 441
- <converter-class> 87, 175
- <converter-extension> 175
- <converter-for-class> 175

<converter-id> 87, 175
ConverterException 86
converterMessage 87, 113, 144
cookie
 vordefinierte EL-Variable 48
Cookie 48
coords 385
Create, Read, Update, Delete *siehe*
 CRUD
createValueBinding() 54
createValueExpression() 305
createValueExpression() 54,
 234
Cross-Site-Scripting 250
CSS 184, 205, 388
CSS-Regel 388
@CustomScoped 173, 378

DataModel 22, 127, 306
DataModelEvent
 DataModelEvent 115
DataModelEvent 127
DataModelListener 116
dateStyle 78
DateTimeConverter 77
DATETIMECONVERTER_DEFAULT_
 TIMEZONE_IS_SYSTEM_
 TIMEZONE 163
DecimalFormat 80, 440
@DecimalMax 99
@DecimalMin 99
decode() 203
DECORATORS 163
Default 103
<default-locale> 149
<default-render-kit-id> 172
<defaultValidators> 172
DEFAULT_SUFFIX 163
Deklaration
 CSS- 388
Dependency Injection 55, 326
Deployment-Deskriptor 161
@Digits 99
dir 385
DISABLE_DEFAULT_BEAN_
 VALIDATOR 165
DISABLE_FACELET_JSF_
 VIEWHANDLER 163, 465
disabled 385
<div> 390
Document 312
DOM 389
DoubleRangeValidator 88
DTD 162

eager 58, 379
Eclipse 5
EclipseLink 352
Eigenschaft
 CSS- 388
EJB 10, 321
@EJB 69, 321
@EJBs 69
EL 43
 Methodenparameter 52
<el-resolver> 172
enctype 385
Enterprise Archive *siehe* EAR
Enterprise JavaBean *siehe* EJB
@Entity 32, 289
Entity-Manager 25
errorClass 111
Event
 Action- 117
 Data-Model- 127
 System- 130
 Value-Change- 122
Event-Listener 116
EventObject 116
eventsQueue 255

<exception-handler-factory>
 175
ExceptionQueuedEvent 130
Expression
 Deferred- 43
 Immediate- 43
Expression Language *siehe* EL
ExpressionFactory 53
Extension-Mapping 162
<external-context-factory>
 175
ExternalContext 142, 247, 295,
 312, 314

<f:actionListener> 119, 433
<f:ajax> 238, 239, 241, 242, 298,
 386, 434
<f:attribute> 436
<f:convertDateTime> 24, 77, 78,
 437
<f:converter> 87, 441
<f:convertNumber> 79, 302, 439
<f:event> 132, 144, 304, 442
<f:facet> 24, 443
<f:loadBundle> 153, 444
<f:metadata> 143, 445
<f:param> 143, 145, 304, 398, 446
<f:phaseListener> 130, 447
<f:selectItem> 83, 448
<f:selectItems> 83, 155, 301, 450
<f:setPropertyActionListener>
 452
<f:subView> 453
<f:validateBean> 104, 454
<f:validateDoubleRange> 89,
 455
<f:validateLength> 89, 456
<f:validateLongRange> 88, 457
<f:validateRegex> 89, 458

<f:validateRequired> 89, 459
<f:validator> 95, 460
<f:valueChangeListener> 126,
 461
<f:verbatim> 462
<f:view> 463
<f:viewParam> 464
Facelets 22
FACELETS_BUFFER_SIZE 163
FACELETS_LIBRARIES 163
FACELETS_REFRESH_PERIOD 163
FACELETS_RESOURCE_RESOLVER
 163
FACELETS_SKIP_COMMENTS 163,
 226, 475
FACELETS_SUFFIX 163
FACELETS_VIEW_MAPPINGS 164
<faces-config-extension> 171
<faces-config> 179
<faces-context-factory> 175
faces-redirect 141, 321
Faces-Servlet 161
@FacesBehavior 377
@FacesBehaviorRenderer 377
@FacesComponent 377
facesContext
 vordefinierte EL-Variable 48
FacesContext 488
115
FacesContext 39, 48, 142, 312, 314
@FacesConverter 86, 87, 375
FacesEvent 115
FacesMessage() 113
FacesMessage 86, 110
@FacesRenderer 377
@FacesRenderKit 377
@FacesValidator 95, 376
<facet> 174
<factory> 171, 175
<factory-extension> 175
Familienstand 81

find() 302
findComponent() 121, 186, 412,
 414, 417
findComponent() 234
Firebug 359
flash 48
 vordefinierte EL-Variable 48
fn:toUpperCase() 196
frame 385
<from-action> 139
<from-outcome> 136, 140
<from-view-id> 136, 174
 Wildcard 138
FULL_STATE_SAVING_VIEW_IDS
 164
@Future 99

@GeneratedValue 32
getApplication() 305
getApplication() 234
getAsObject() 73, 85
getAsString() 73, 85
getAttributes() 168
getChildren() 203
getComponent() 234
getContents() 157
getCurrentInstance() 305
getDataModel() 127
getDefaultLocale() 156
getElContext() 305
getExecuteIds() 246
getExpressionFactory() 53, 234
getId() 180, 333
getInitParameter() 168
getLocale() 159
getManagedBeanValue() 370
getNavigationHandler() 295
getParent() 234
getPhaseId() 128, 129
getProjectStage() 167, 168
getRenderIds() 246
getRequestParameter() 247
getResourceBundle() 159
getRowCount() 370
getRowData() 127, 306
getRowindex() 127
getStateManager() 169
getSupportedLocale() 155
getViewRoot() 159
getWrappedData() 309, 371
globalOnly 107
GregorianCalendar 147
groups 104
GUI 10, 87

<h:body> 191, 393
<h:button> 145, 394
<h:column> 23, 396
<h:commandButton> 397
<h:commandLink> 398
 mit CSS 288
<h:dataTable> 22, 266, 399
<h:form> 191, 401
<h:graphicImage> 188, 265, 402
<h:head> 191, 403
<h:inputHidden> 404
<h:inputSecret> 293, 405
<h:inputText> 406
<h:inputTextarea> 407
<h:link> 145, 408
<h:message> 107, 410
<h:messages> 107, 413
 automatisch einfügen 167
<h:outputFormat> 416
<h:outputLabel> 417
<h:outputLink> 418
<h:outputScript> 191, 240, 419
<h:outputStylesheet> 188, 420
<h:outputStylesheet> 191
<h:outputText> 421

<h:panelGrid> 422
<h:panelGroup> 424
<h:selectBooleanCheckbox> 425
<h:selectManyCheckbox> 426
<h:selectManyListbox> 427
<h:selectManyMenu> 428
<h:selectOneListbox> 429
<f:selectOneMenu> 83
<h:selectOneMenu> 66, 300, 430
<h:selectOneRadio> 431
handleNavigation() 295
Handler 10
header
 vordefinierte EL-Variable 48
headerClass-Attribut
 <h:dataTable> 23
headerValues
 vordefinierte EL-Variable 48
height 385
<h:outputScript> 188
hreflang 385
HTML 10
HtmlForm 180
HtmlInputText 372
HTMLInputTextarea 120
HtmlOutputText 64
HtmlTextInput 371
HTTP
 -Protokoll 35
 -Request-Header 47
 Accept-Language-Header 149
 Zeichensatz für 148
HttpServletRequest 312, 314

id-Attribut 390
@Id 32
IEEE 754-1895 76
<if> 139, 140
immediate 40

Befehlskomponente 121
Eingabekomponente 95
immediate 288
includeViewParams 145
infoClass 111
@Inheritance 290
initParam
 vordefinierte EL-Variable 48
@Inject 331
Integrität 292
Integrity 292
Internationalisierung 147
INTERPRET_EMPTY_STRING_
 SUBMITTED_VALUES_AS_
 NULL 164, 309
invalidateSession() 295
Inversion of Control *siehe* IoC
IoC 55
isAjaxRequest() 246
ismap 385
ISO
 639 148
 Country Codes 148
 Language Codes 148
 3166 148
 4217 81
 8859 148
isPartialRequest() 246
isSavingStateInClient() 169
isTransient() 332
iText 311

JAAS 336
Java Archive *siehe* JAR
Java Authentication and
 Authorization Service
 siehe JAAS
Java Community Process *siehe*
 JCP
Java Naming and Directory
 Interface *siehe* JNDI

Java Persistence API *siehe* JPA
Java Specification Request *siehe*
JSR
Java Transaction API *siehe* JTA
`java.util`
 Package 147
JavaScript 184, 205, 386
JavaServer Faces *siehe* JSF
JavaServer Pages *siehe* JSP
JavaServer Pages Standard Tag
 Library *siehe* JSTL
`javax.annotation`
 Package 68
`javax.enterprise.context`
 Package 326, 328
`javax.faces`
 Ressourcen-Bibliothek 193, 240
`javax.faces.bean`
 Package 378
`javax.faces.component`
 Package 200, 390
`javax.faces.component.html`
 Package 205, 390
`javax.faces.convert`
 Package 72, 85
`javax.faces.el`
 Package deprecated 54
`javax.faces.event`
 Package 116
`javax.faces.Messages`
 Resource-Bundle 105
`javax.faces.model`
 Package 116, 128
`javax.faces.render`
 Package 204
`javax.faces.validator`
 Package 87
`javax.persistence`
 EntityManager 25
`javax.validation.constraints`
 Package 98
`javax.validation.groups`
 Package 103
JBoss Seam 129, 173
JCP 10, 161
`@JoinColumn` 291
`@JoinTable` 338
JPA 25, 289, 318
 automatic Dirty-Checking 123
 Criteria API 309
 Named-Query 26
JSF-EL 15, 43
JSF-Konfigurationsdatei
 Navigation 134
JSF-Konfigurationsdatei 56
`jsfc` 234
JSFSession 368
JSFUnit 368
JSP 9
JSR-330 326
JSTL 128, 194, 200
JTA 318, 349
JUnit 367
JVM 1
Komponente 199
 Verhaltensmodell einer 242
Komponenten
 HTML-Standard- 205
Komponentenbaum 38
 optimierte
 Zustandsspeicherung 168
 Wiederherstellung 38, 39
Komponentenbindung 64
Kontext
 Portlet- 312
 Servlet- 312
Konversation *siehe auch* Klasse
 Conversation

- Anmeldeweiterleitung mit
 - Seam 340
 - mit CDI 327
- Konvertierer
 - für Aufzählungstypen 81
 - für Standarddatentypen 73
- Konvertierung 72
 - label 113, 301
 - lang 385
 - Lebenszyklus 36
 - und immediate 95
 - Lebenszyklus
 - mit immediate 122
 - LengthValidator 88
 - Lesezeichen 144
 - <lifecycle> 130, 171, 175
 - <lifecycle-extension> 175
 - <lifecycle-factory> 175
 - LIFECYCLE_ID 164
 - lineDirection 426, 431
 - <list-entries> 57, 60, 173
 - ListDataModel 128
 - Listener
 - Action- 117
 - @ListenerFor 132, 377
 - @ListenersFor 132, 377
 - ListResourceBundle 157
 - @Local 321
 - locale 78
 - Locale 147, 148
 - <locale-config> 149, 172
 - localePrefix 191
 - Lokalisierung 148
 - longdesc 385
 - LongRangeValidator 88
 - Managed Bean 39, 54
 - und Lokalisierung 159
 - <managed-bean-class> 57, 173
 - <managed-bean-extension> 57, 173
 - <managed-bean-name> 57, 173
 - <managed-bean-scope> 57, 173
 - <managed-property> 57, 173, 266
 - @ManagedBean 25, 70, 71, 173, 335, 379
 - versus @Named 327
 - @ManagedProperty 70, 71, 173, 266, 306, 379
 - @ManyToMany 338
 - @ManyToOne 291
 - <map-entries> 57, 60, 173
 - <map-entry> 60
 - @Max 99, 100
 - merge() 29, 299, 322
 - <message-bundle> 172
 - MessageFormat 107
 - Messages.properties 105
 - metadata-complete 179
 - Method Expression 45
 - method-signature 231
 - MethodExpressionValidator 88
 - Methodenausdruck 45
 - MIME-Typ 313
 - application/vnd.ms-excel 313, 314
 - @Min 99, 100
 - Mojarra 5, 12
 - Konfiguration 350
 - MVC 12, 55
 - <name> 177
 - index 70
 - name-Attribut
 - <f:facet> 24
 - @Named 326, 331
 - @NamedEvent 132, 377
 - @NamedQuery 32
 - Namens-Container 183

- NamingContainer 180
- Navigation 133
 - Handler 133
 - bedingte 139
 - implizite 134
 - lokalisierte 159
 - View-to-View 135
- <navigation-case> 136, 139, 174
- <navigation-handler> 172
- <navigation-rule> 136, 174
- <navigation-rule-extension> 174
- No-Interface View 323
- @NoneScoped 70, 173, 380
- @NotNull 99, 100
- @Null 99
 - <null-value> 59
- NumberConverter 77
- Object Relational Mapper *siehe* OR-Mapper
- offset 226
- onblur 387
- onchange 387
- onclick 387
- ondblclick 387
- @OneToMany 289
- onfocus 186, 387
- onkeydown 387
- onkeypress 387
- onkeyup 387
- onmousedown 387
- onmousemove 387
- onmouseout 184, 387
- onmouseover 184, 387
- onmouseup 387
- onreset 387
- onselect 387
- onsubmit 388
- OpenAjax-Alliance 240
- <option> 430
- <ordering> 171, 177, 178
- Ordnung
 - relative - 177, 178
 - vollständige - 177
 - von Konfigurationsdateien 177
- <other> 178
- Page Description Language *siehe* PDL
- pageDirection 426, 431
- param
 - vordefinierte EL-Variable 48
- <param-name> 162
- <param-value> 162
- paramValues
 - vordefinierte EL-Variable 48
- Partial State Saving 43
- <partial-traversal> 172
- <partial-view-context-factory> 175
- PARTIAL_STATE_SAVING 164, 167
- PartialStateHolder 168
- PartailViewContext 246
- @Past 99, 100
- pattern 78
- @Pattern 99
- PDL 22, 42, 199, *siehe auch* VDL
- persist() 31
- persist() 26
- persistence.xml 349
- @PersistenceContext 25, 69
- @PersistenceContexts 69
- @PersistenceUnit 69
- @PersistenceUnits 69
- Persistenz 25
- Persistenzkontext
 - erweiterter 322
 - transaktionaler 322
- Phase-Event

Verwendungsbeispiel 129
<phase-listener> 130, 175
PhaseEvent 128
PhaseId 128
 ANY_PHASE 128
 APPLY_REQUEST_VALUES 128
 INVOKE_APPLICATION 128
 PROCESS_VALIDATIONS 128
 RENDER_RESPONSE 128
 RESTORE_VIEW 128
 UPDATE_MODEL_VALUES 128
PhaseListener 116, 128, 447
Plain Old Java Object *siehe* POJO
POJO 10, 54
Post-Back 36, 395, 409
PostAddToViewEvent 131
@PostConstruct 27, 69
PostConstructApplicationEvent
 131
PostConstructCustomScopeEvent
 131, 379
PostConstructViewMapEvent 131
PostRestoreStateEvent 131
PostValidateEvent 131
Präfix-Mapping 162
@PreDestroy 69
PreDestroyApplicationEvent
 131
PreDestroyCustomScopeEvent
 131, 379
PreDestroyViewMapEvent 131
prependId 187, 253
PreRemoveFromViewEvent 131
PreRenderComponentEvent 131,
 132
PreRenderViewEvent 131, 132
PreValidateEvent 131
process 258
processAction() 120
PROJECT_STAGE 164, 167
ProjectStage 167
Properties 150
<property> 174, 175, 179
<property-resolver> 172
PropertyResourceBundle 150
readonly 385
<redirect> 140
redirect() 142
Redirect 140
<referenced-bean> 174
<referenced-bean-class> 174
<referenced-bean-name> 174
@ReferencedBean 380
RegexValidator 88
rel 385
Relocation 188
@Remote 321, 334
Remote Method Invocation *siehe*
 RMI
remove() 29
<render-kit> 171, 174
Render-Kit 174, 204
<render-kit-class> 174
<render-kit-extension> 174
<render-kit-factory> 175
<render-kit-id> 174
Render-Satz 204
<renderer> 174
Renderer 204
RenderKit 204
RenderKitFactory 204
renderResponse() 115
request
 vordefinierte EL-Variable 48
requestDelay 256
requestScope
 vordefinierte EL-Variable 48
@RequestScoped 70, 173, 328, 380
required 89

requiredMessage 113, 144
RequiredValidator 88
resource 48, 188, 233
 vordefinierte EL-Variable 48
@Resource 26, 69
<resource-bundle> 151
Resource-Bundle 150
<resource-handler> 172
Resource-Injection 68
RESOURCE_EXCLUDES 164, 169
ResourceBundle 147, 150, 159
@ResourceDependencies 193, 377
@ResourceDependency 193, 377
ResourceHandler 188
@Resources 69
responseComplete() 115, 313
ResponseStateManager 204
Ressourcen-Identifikator 189
@Restrict 340
ResultDataModel 128
ResultSetDataModel 128
rev 385
RFC
 1341 313
 1521 313
 1522 313
<rich:datascroller> 278
<rich:dataTable> 278
<rich:dndParam> 262
<rich:dragIndicator> 262
<rich:dragListener> 262
<rich:dragSupport> 262, 264
<rich:dropListener> 262
<rich:dropSupport> 262, 266
<rich:panel> 264
<rich:recursiveTreeNodesAdaptor>
 274

<rich:scrollableDataTable>
 280
<rich:toolTip> 264
<rich:treeNode> 271
<rich:treeNodesAdaptor> 272
RMI 319
@RoleName 338
rules 385
ScalarDataModel 128
Scope 58
 Application- 58
 Custom- 58
 None- 62
 Request- 58
 Session- 58, 283
 View- 58, 63
<security-constraint> 169
<select> 430
SelectItem 84, 125, 126
Selektor
 CSS- 388
 Id- 389
 Klassen- 389
Selenese 366
Selenium 362
 -IDE 362
 -RC (Remote Control) 365
SEPARATOR_CHAR 164
<servlet> 162, 334
Servlet 9
 <servlet-class> 162
 <servlet-mapping> 162, 334
 <servlet-name> 162
session
 vordefinierte EL-Variable 48
Session Bean 321
sessionScope
 vordefinierte EL-Variable 48
@SessionScoped 25, 70, 173, 326,
 328, 381
setContentType() 313
setId() 180

setStyle() 186
setWrappedData() 27
SFSB 326
shape 385
showDetail 108
showSummary 108
Simple Object Access Protocol
 siehe SOAP
SimpleDateFormat 79, 437
size 386
@Size 99, 100
<state-manager> 172
STATE_SAVING_METHOD 164, 165
@Stateful 326
StateHolder 168
step 226
style 386
styleClass 386
summary 386
<supported-locale> 149
SVG 42
Swing 16
<system-event-class> 131
<system-event-listener> 131,
 172
<system-event-listener-class>
 131
SystemEvent 115, 130
SystemEventListener 131

tabindex 386
<tag-handler-delegate-factory>
 175
target 386
TCK 12
Template 216
 -Client 216
 geschachtelt 469
TestNg 367

TicTacToeHandler 16
timeStyle 78
timeZone 78
TimeZone 437
title 386
<to-view-id> 136, 140
@TransactionAttribute 326
TreeNode 267
type 78

<ui:component> 225, 466
<ui:composition> 216, 221, 467
 template 221
<ui:debug> 226, 227, 468
<ui:decorate> 225, 469
<ui:define> 216, 222, 296, 470
<ui:fragment> 225, 471
<ui:include> 216, 220, 472
<ui:insert> 216, 220, 473
<ui:param> 225, 474
<ui:remove> 226, 475
<ui:repeat> 118, 225, 476
UIColumn 201
UICommand 201
UIComponent 180, 201, 203, 412,
 417
UIComponentBase 201, 203, 242
UIData 180, 202
UIForm 180, 202, 203
UIGraphic 202
UIInput 202
UIMessage 202
UIMessages 202
UINamingContainer 180, 202, 453
UIOutcomeTarget 202
UIOutput 202, 462
UIParameter 202, 446
UISelectBoolean 202
UISelectItem 202, 448
UISelectItems 202, 451
UISelectMany 203, 432

UISelectOne 203, 432
UISqlParameter 203
UIViewRoot 186, 203, 463
Unified Expression-Language 43,
 52, 118
URI 38
<url-pattern> 162
usemap 386
@UserPassword 337
@UserPrincipal 337
@UserRoles 338
UTF-8 148

validate 93
VALIDATE_EMPTY_FIELDS 101, 164
validateEmail 93
validationGroups 104
<validator> 95, 171, 175
validator 92
Validator 93, 94, 460
<validator-class> 95, 175
<validator-extension> 175
<validator-id> 95, 175
ValidatorException 94
validatorMessage 113, 144
Validierung 40, 72
value 22
Value Expression 44
Value-Change-Event 41
ValueChangeEvent 115
 getNewValue() 124
 getOldValue() 124
ValueChangeEventHandler 123
ValueChangeListener 116, 123,
 126
ValueHolder 432
<var> 151
var 22
<variable-resolver> 172
VDL 199, siehe auch PDL
Vertraulichkeit 292
view
 vordefinierte EL-Variable 48
View 38
 -Id 135
 <UIViewRoot> 186
 in MVC 10
View Declaration Language *siehe*
 VDL
<view-declaration-language-
 factory> 175
<view-handler> 172
View-Parameter 143
ViewHandler 172
viewScope
 vordefinierte EL-Variable 48
@ViewScoped 70, 173, 381
<visit-context-factory> 175

Web 2.0 237
Web Service Description Language
 siehe WSDL
<web-app> 162
Web-Profile 334
<web-resource-collection> 169
<web-resource-name> 169
Web-Tools-Platform 353
web.xml 49, 161
 JSF-Anwendung ohne 334
@WebFilter 333
@WebListener 333
@WebServiceRef 69
@WebServiceRefs 69
@WebServlet 333
Weld 324
Wert
 CSS- 388
Wertearausdruck 44, 51, 64
width 386
WML 10, 42

XHTML 22
XMLHttpRequest 237
XPath 44

Zeitzone 79

HANSER

Alles wird gut.



Bernd Müller
JBoss Seam
Die Web-Beans-Implementierung
280 Seiten.
ISBN 978-3-446-41190-6

Seam, das innovative und agile Framework im Bereich von Java-EE, integriert JavaServer Faces, Enterprise JavaBeans und Java Persistence-API und macht viele Konfigurations- und Codierungsarbeiten überflüssig.

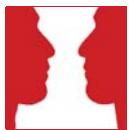
Mit Seam können Sie Webapplikationen deutlich schneller und einfacher entwickeln und bauen gleichzeitig auf eine skalierbare und standardisierte Java-EE-Architektur auf. Dieses Arbeitsbuch führt Sie in die Funktionsweise von Seam ein. Sie lernen die wichtigsten Konzepte und Komponenten im Detail kennen. Dazu zählen lang laufende Transaktionen, Authentifizierung und Autorisierung, PDF-Erzeugung, Geschäftsprozesse, AJAX und das Generieren und Testen von Seam-Anwendungen.

Mehr Informationen zu diesem Buch und zu unserem Programm
unter www.hanser.de/computer



DAS ARBEITSBUCH FÜR JAVA-WEB-ENTWICKLER //

- Steigen Sie mit diesem fundierten Arbeitsbuch in die Entwicklung von Benutzerschnittstellen mit JavaServer Faces 2.0 ein.
- Anhand einer kompletten Beispielanwendung werden alle wichtigen Aspekte von JSF erläutert.
- Vertiefen und erweitern Sie Ihre Fertigkeiten mit den zahlreichen Übungen.
- Verwendet werden ausschließlich Open-Source-Systeme, so dass Sie alle Übungen und Beispiele ohne weitere Lizenzkosten nachvollziehen können.
- Im Internet: Quell-Code zu den Beispielen, Lösungen der Übungen, Dokumentationen zu JavaServer Faces und den verwendeten Systemen auf www.jsfpraxis.de



JavaServer Faces 2.0 // JavaServer Faces 2.0 sind ein Framework für die Entwicklung von Benutzerschnittstellen für oder besser als Teil einer Java-Web-Anwendung. Dieses Arbeitsbuch führt Sie Schritt für Schritt in die Programmierung mit JavaServer Faces ein. Sie erfahren, wie Sie damit moderne Benutzerschnittstellen für die Praxis entwickeln. Und natürlich geht es auch darum, wie JavaServer Faces in eine Java-Web-Anwendung zu integrieren sind. Behandelt werden auch Themen wie die Anbindung an eine Datenbank mit JPA und die Verwendung von CDI, das Erzeugen von PDFs und Ajax.

Verfolgen Sie Schritt für Schritt die Entwicklung einer Online-Banking-Anwendung und lernen Sie so mit Hilfe realer Aufgabenstellungen alle wichtigen Aspekte von JavaServer Faces 2.0 kennen. Mit Hilfe der Übungen, deren Lösungen Sie sich von der Website zum Buch herunterladen können, können Sie das Gelernte selbst ausprobieren und umsetzen.

AUS DEM INHALT // Einleitung und Motivation // JavaServer Faces im Detail // Die UI-Komponenten // Facelets // Ajax // JavaServer Faces im Einsatz: Das Online-Banking // JavaServer Faces und Java-EE // Systeme und Werkzeuge // Anhang: Annotationen und Tags // Anhang: Die Tags der Standardbibliotheken //

Prof. Dr. Bernd **MÜLLER** unterrichtet Software-Technik an der Fakultät Informatik der Hochschule Braunschweig/Wolfenbüttel und ist Autor von Fachbüchern und zahlreichen Veröffentlichungen in Fachzeitschriften und auf Konferenzen.

HANSER

www.hanser.de/computer

ISBN 978-3-446-41992-6

9 783446 419926

Java-Web-Entwickler
Studenten der Informatik