bernd MÜLLER

JavaServer™ Faces und Jakarta Server Faces 2.3

Ein Arbeitsbuch für die Praxis

3. Auflage

Im Internet: Quell-Code zu den Beispielen und Lösungen der Übungen

HANSER

Müller

JavaServer™ Faces und Jakarta Server Faces 2.3

Bernd Müller

JavaServer™ Faces und Jakarta Server Faces 2.3

Ein Arbeitsbuch für die Praxis

3., überarbeitete Auflage

HANSER

Der Autor:

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

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Layout: der Autor mit LaTeX
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Max Kostopoulos
Datenbelichtung, Druck und Bindung: Eberl & Kœsel GmbH & Co. KG, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-45670-9
E-Book-ISBN: 978-3-446-45977-9
E-Pub-ISBN: 978-3-446-47008-8

Inhalt

Vorwort zur 3. Auflage

In der 2. Auflage haben wir die vielen Neuerungen von JSF 2.0 integriert. Die Version 2.1 war eher kosmetischer Natur. Die Version 2.2 brachte wiederum sehr viel Neues, die Version 2.3 ebenso. Es ist also dringend an der Zeit für eine 3. Auflage.

Wie bereits in der 2. Auflage praktiziert, soll auch diese 3. Auflage keine Darstellung aller JSF-Versionen sein. Sie werden daher in der Regel keine Bezüge auf ältere JSF-Versionen, sondern ausschließlich den aktuellen Stand 2.3 beschrieben finden. Damit geht einher, dass einige Abschnitte der früheren Auflagen verschwunden sind, andere erheblich überarbeitet wurden und das Buch damit eine neue Struktur erhalten hat. Mehr noch, Sie halten praktisch ein komplett neues Buch in der Hand.

Während dieses Buch entstand, wurde aus *JavaServer Faces* 2.3 *Jakarta Server Faces* 2.3. Diese beiden Spezifikationen sind praktisch identisch, so dass das Buch beide Systeme beschreibt. Wir sprechen im Folgenden immer von JavaServer Faces, meinen aber beide Spezifikationen.

In der 2. Auflage nutzten wir Eclipse als Build-System. Wir haben nun alle Projektbeispiele auf Maven umgestellt und verwenden WildFly als Application-Server. Da die Projektbeispiele nur offizielle APIs verwenden, sind sie auf allen zertifizierten Servern verwendbar.

Bernd Müller, April 2021

bernd.mueller@ostfalia.de

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 „Arbeitsbuchs 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 Website des Buchs 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.

Bernd Müller Im Sommer 2006

1 Einleitung

1.1 Ziel dieses Buchs

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 weit über zwanzig Jahren, ist festzustellen, dass beide Ziele nicht erreicht wurden. Java wird zwar zur Programmierung von Hardware verwendet, so etwa in jedem Blu-ray-Player, doch keineswegs in dem Ausmaß, wie ursprünglich prognostiziert. Der Siegeszug von Googles Android trägt ebenfalls stark zur Verbreitung von Java bei, da Java die Standardsprache bei der Entwicklung von Software unter Android ist. Jedoch ist diese Art der Verwendung nicht die 1995 vorhergesagte, da sie auf der Anwendungs- und nicht der Hardware-Ebene stattfindet. Das zweite genannte Ziel, mit Applets das Web interaktiver zu machen, krankte an den vielen durch Applets verursachten Sicherheitsproblemen. Applets sind mittlerweile tot und aus dem JDK verbannt.

Java ist jedoch sicher nicht als Fehlschlag zu werten, ganz im Gegenteil. 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 sogenannter *Web-Anwendungen*, d.h. Anwendungen, bei denen der Benutzer durch einen HTML-Browser die Anwendung bedient, die Anwendungslogik aber auf dem Server ausgeführt wird, wird mittlerweile von mehreren Java-Spezifikationen unterstützt, von denen JavaServer Faces die aktuellste, je 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 ist 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 also 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 Buchs ist es, dem Leser die Kenntnisse zu vermitteln, die man benötigt, um mit JavaServer Faces moderne Benutzerschnittstellen zu entwickeln. Dem im Untertitel genannten Anspruch von Praxisnähe und dem Charakter eines Arbeitsbuchs wird das Buch gerecht, indem wir eine größere Anwendung entwickeln, anhand derer 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 wahrscheinlich nicht die letzte sein. Wir gehen auf Versionen und auf die größten Konkurrenten von JSF, die aktuell sehr angesagten JavaScript-Frameworks später noch ein.

1.2 Was sind JavaServer Faces?

JavaServer Faces sind ein Teil der Java-EE-Spezifikation (Java Enterprise Edition) und zwar seit der Version 5. Tabelle 1.1 zeigt in der Übersicht die jeweiligen Versionszugehörigkeiten. Die rechte Spalte zeigt die Veröffentlichungstermine von Java EE. Die JSF-Spezifikationen wurden in der Regel kurz vorher veröffentlicht.

Tabelle 1.1 Java-EE und JSF-Spezifikationen

	JSF-Version	Veröffentlicht
Java EE 5	1.2	Mai 2006
Java EE 6	2.0	Dezember 2009
Java EE 7	2.2	Mai 2013
Java EE 8	2.3	September 2017

JavaServer Faces sind unabdingbarer Bestandteil von Java EE und damit in jedem Application-Server enthalten, der als Java-EE-konform zertifiziert ist. Dies gilt auch und gerade für das mit Java EE 6 erstmalig eingeführte Web-Profil, das zum Ziel hatte, die Anzahl der Teilspezifikationen, die unter Java EE zusammengefasst sind, auf eine möglichst kleine, aber noch sinnvoll einzusetzende Teilmenge zu beschränken. Das Web-Profil beschreibt

sich selbst als „The Web Profile is targeted at developers of modern web applications“, also die Spezifikationen, die für die Umsetzung moderner Web-Anwendungen benötigt werden.

Die beiden weiteren GUI-relevanten Spezifikationen innerhalb von Java EE sind Servlets und JavaServer Pages (JSP). Das Einsatzgebiet von Servlets ist jedoch die Realisierung von Frameworks, wie JSF es eines ist, und nicht die Erstellung und Umsetzung kompletter Geschäftsanwendungen mit komplexen Benutzungsoberflächen. JavaServer Pages werden bereits seit einigen Jahren 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.

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 und JavaFX ein Komponentenmodell für lokale Oberflächen definieren. 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- und JavaFX-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 geschickte HTML-Seite (andere Antwortarten sind möglich) stellen ein Abbild des Komponentenbaums des Servers dar.

Bei der Entwicklung grafischer Benutzungsoberflächen (GUI, Graphical User Interface) hat sich das Model-View-Controller-Entwurfsmuster (MVC-Pattern) durchgesetzt, das, entsprechend angepasst, auch von JavaServer Faces realisiert wird. MVC trennt streng zwischen *Model*, *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), JPA-Entitys oder beliebige andere Java-Objekte sind möglich. Die View wird durch die JSF-Komponenten (Unterklassen der Klasse `UIComponent`) und Facelets (XHTML) realisiert. Was Facelets angeht, ist dies lediglich die Default-Technologie der sogenannten *View Declaration Language*, kurz *VDL*. Die Spezifikation schreibt sie als Mindestanforderung vor, andere sind aber ebenfalls denkbar. So haben wir etwa in der ersten Auflage des Buchs JavaServer Pages (JSP) als VDL verwendet, was zur damaligen Zeit der Standard war, aber auch XULFaces beschrieben, das als Seitenbeschreibungssprache XUL besaß, eine von Mozilla definierte Sprache zur Beschreibung von Benutzungsoberflächen. Auch für WML (Wireless Markup Language) existierte eine prototypische

Implementierung. WML ist eine stark reduzierte Version von HTML, die in Mobiltelefonen der ersten Generation Verwendung fand. Mittlerweile gibt es keine ernstzunehmenden alternativen Sprachen mehr, da HTML 5 sich zum Standard entwickelt hat, was sehr zu begrüßen ist und Facelets in seiner aktuellen Form HTML 5 vollständig unterstützt. JSP wird nicht mehr unterstützt und sollte nicht mehr verwendet werden, da Facelets zum JSF-Standard geworden sind. „JSF auf Basis von XHTML“ im Sinne einer VDL und „Facelets“ sind damit praktisch synonym und werden von uns nicht mehr weiter unterschieden. Der Controller-Teil des MVC-Patterns wird schließlich durch das Faces-Servlet realisiert, auf das wir später noch ausführlich eingehen. Prinzipiell nimmt es Benutzerinteraktionen und Dateneingaben des Benutzers entgegen und führt entsprechende Aktionen aus und wählt die entsprechende View zur Anzeige aus. Dies wird durch Controller-Funktionalität, die in Managed Beans implementiert ist, ergänzt. Diesen ist Abschnitt 2.3 gewidmet.

1.3 Der Leser

Das intendierte Einsatzgebiet von JavaServer Faces ist die Entwicklung von Oberflächen in unternehmenskritischen Anwendungen. Jeder, der in diesem Bereich zurzeit 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 Buchs 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 und der Software-Entwicklung 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 all zu sehr strapazieren zu müssen. Eine Ausnahme sind Eigennamen, so dass wir „JavaServer Faces“ in der

Originalschreibung übernehmen. Eigennamen in Wortverbindungen sind dann das nächste Problem. „Java EE 8“ schreiben wir getrennt, „Java-EE-8-Application-Server“ zusammen. Eine Getrenntschreibung wäre nach unserer Meinung sehr unleserlich. Wörter, die englische und deutsche Bestandteile besitzen, schreiben wir in der Regel ebenfalls mit Bindestrich. Falls Sie selbst schreibend tätig sind, ist das Buch von Peter Rechenberg [Rec06] eine hervorragende und sehr zu empfehlende Hilfe, wenn es um Zweifelsfragen der Grammatik und des Ausdrucks geht.

Ein weiteres sprachliches Problem ist die Fachlichkeit der Anwendung. In großen, insbesondere multinationalen Projekten ist es unbestritten sehr sinnvoll, die gesamte Fachlichkeit in Englisch auszudrücken. In unserem Fall wären das vor allem Klassen-, Methoden- und Variablenbezeichner sowie Dateinamen. Nicht jeder Leser dieses Buchs wird jedoch in allen unseren Beispielen die jeweils entsprechenden englischen Fachbegriffe parat haben – und dies gilt vor allem auch für den Autor! Der Leser möge es uns bitte nachsehen, wenn einige Klassen englische, andere deutsche Bezeichner verwenden, wir also munter mischen.

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 Buchs 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, ganz 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].

1.4 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-, häufig auch in den Major-Versionsnummern während des Schreibens des Buchs Aktualisierungen erhalten beziehungsweise wurden erst während des Schreibens veröffentlicht. Es ist nicht sinnvoll, einem Buch über JavaServer Faces oder andere aktuelle IT-Themen wie früher eine CD beizulegen oder gar API-Dokumentationen abzudrucken. Als Zugang zu Informationen dieser Art betreiben wir die Website

```
https://www.jsfpraxis.de
```

Sie finden dort den lauffähigen Quell-Code aller Beispiele sowie Lösungen zu den Übungsaufgaben als Maven-Projekte und, wenn nötig, ein Druckfehlerverzeichnis. Neben diesen buchspezifischen Inhalten finden Sie eine Fülle von Informationen zu JavaServer Faces,

aber auch zu den im Buch verwendeten Systemen. Wir stellen diese in Kapitel 7 detaillierter dar, empfehlen dem Leser aber zusätzlich die eventuell aktuelleren Informationen auf der Website.

Als Alternative zur Website `www.jsfpraxis.de` und dem aktuellen Zeitgeist folgend finden Sie die Projekte auch auf Github:

```
https://github.com/BerndMuller
```

Da sämtlicher Quell-Code online verfügbar ist, kürzen wir Listings im Buch auf das für das Verständnis Nötigste. Dies erspart Ihnen unnötiges Blättern und führt zu einem günstigeren Buchpreis.

Der Quell-Code ist in Maven-Projekten organisiert, die der Kapitelstruktur des Buchs folgen. Einzige Ausnahme ist das Projekt zum Thema mobiler Endgeräte. Die Tabelle 1.2 zeigt die Zuordnung der Projekte zu den Kapiteln bzw. Abschnitten.

Tabelle 1.2 Projekte und deren Zuordnung

Kapitel/Abschnitt	Projekt
2 JavaServer Faces im Detail	jsf-im-detail
3 Contexts and Dependency Injection	jsf-cdi
4 Weiterführende Themen	jsf-advanced
5 JavaServer Faces im Einsatz: Classic Models	classic-models
6 Spezialthemen	jsf-special
6.7 Mobile Endgeräte	jsf-mobile

Das Buch ist ein Arbeitsbuch für die Praxis. Der erzielbare Nutzen beim „einfachen“ Lesen des Buchs ist auf konzeptionelle Erkenntnisse und Aha-Effekte beschränkt. Wenn Sie den Nutzen des Buchs maximieren wollen, müssen Sie die Projekte herunterladen und selbst ausprobieren. Es empfehlen 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.

Als ganz konkrete Aufforderung, etwa ein bestimmtes Feature eines Projekts auszuprobieren, etwas zu ändern oder eine neue Lösung zu entwickeln, werden wir im Verlauf Textboxen einbauen, die mit einem Werkzeug-Icon versehen sind die konkrete Aufgabe benennen.

Dies ist eine Aufgabe. Hier müssen Sie entwerfen, programmieren, testen, ausprobieren.

Vervollständigt wird dies durch weitere Textboxen, die mit einer Glühbirne einen Hinweis oder Tipp symbolisieren und mit einem Ausrufungszeichen vor Problemen, möglichen Fehlerquellen oder Herausforderungen warnen.

Dies ist ein Hinweis oder ein Tipp. Befolgen Sie ihn bitte!

Achtung! Hier machen wir Sie auf Probleme aufmerksam.

1.5 Versionen, Versionen, Versionen ...

Wie bereits erwähnt, haben die Systeme, die wir verwenden, während der Buchentstehung viele Versionen gesehen bzw. wurden erst veröffentlicht. Für die Version von JavaServer Faces, die wir beschreiben, gilt dies nicht. JavaServer Faces 2.3 ist bei Drucklegung die aktuelle Version, hat aber wie in Tabelle 2 dargestellt eine längere Historie. Zudem wird dies auch die letzte Version von JavaServer Faces sein, da JavaServer Faces wie das komplette Java EE der Eclipse Foundation übergeben wurde und nun unter diesem neuen Dach weiterentwickelt wird. Der neue Name lautet *Jakarta Server Faces* 2.3. Zu Beginn des Kapitels 7 gehen wir auf die Hintergründe und Gegebenheiten der Übergabe etwas detaillierter ein.

Eine Unterscheidung und vor allem textuelle Kenntlichmachung, welches vorgestellte Feature von JavaServer Faces in welcher Version vorhanden ist oder nicht, ist unserer Meinung nach nicht sinnvoll, da dies den Lesefluss sehr hemmt und – wahrscheinlich – für die meisten Leser nicht relevant ist.

Wir beschreiben hier ausschließlich die Version 2.3 von JavaServer Faces. Leser, die sich für frühere Versionen interessieren, finden die erste und zweite Auflage des Buchs sicher noch in Bibliotheken. Bis zur Version 2.0 sollten Sie die erste, bis zur Version 2.3 die zweite Auflage des Buchs verwenden.

1.6 Spezifikationen, Implementierungen, Systeme

Zunächst ein wenig Geschichte. 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 Definition 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 JCP. 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.

Zurück zum Hier und Heute. Um die in diesem Buch entwickelte Software auszuführen, benötigt man einen *Servlet-Container* und eine Implementierung von JavaServer Faces. Die in Java EE 8 verwendete Servlet-Version 4.0 wird durch den Java Specification Request 369 (JSR 369) [URL-JSR369] spezifiziert. Die in Java EE 8 verwendete Version von JavaServer Faces ist 2.3 und damit im JSR 372 [URL-JSR372] festgelegt.

Eine von vielen JSF-Entwicklern verwendete Systemkonfiguration ist die Verwendung von Apache Tomcat [URL-TOM] als Servlet-Container. Als Implementierung von JavaServer Faces kommen Mojarra [URL-MOJ] und MyFaces [URL-MF] in Betracht, die beide zertifiziert sind.

Mit der Version 2.3 von JavaServer Faces muss für eine vollumfängliche Verwendung von JavaServer Faces jedoch nicht nur eine Servlet-Implementierung vorhanden sein, sondern auch CDI 2.0 [URL-JSR365], Expression Language 3.0 [URL-JSR341], WebSocket 1.1 [URL-JSR356] und JSON-Processing 1.1 [URL-JSR353].

Implementierungen dieser Spezifikationen zusammen mit Ihrer Anwendung im Tomcat zu deployen, ist unserer Meinung nach nicht sinnvoll. Viel sinnvoller ist die Verwendung eines Application-Servers, der Java EE 8 vollständig implementiert. Als einzige Maven-Abhängigkeit reicht dann „Java EE 8“ aus, als entsprechender Maven-Scope `provided`. Das erzeugte WAR ist sehr klein, die benötigte Zeit für das Deployment sehr kurz.

Die Referenzimplementierung von Java EE 8 ist *GlassFish* in der Version 5.0. GlassFish, aber auch jeder andere Application-Server, enthält Implementierungen der genannten Spezifikationen, so dass kein weiterer Konfigurationsaufwand erforderlich ist.

Oracle, als Hersteller des GlassFisch-Application-Servers, hat bereits vor einigen Jahren angekündigt, den kommerziellen Support für den Application-Server auslaufen zu lassen und sich auf den WebLogic-Application-Server für kommerzielle Kunden zu konzentrieren. Gleichwohl wurde die Referenzimplementierung für Java EE 8 im September 2017 noch fertiggestellt und kann unter [URL-GF] heruntergeladen werden. Allerdings gab es seither keine weiteren Releases mehr, was Bände über die Bedeutung dieses Angebots für Oracle spricht. Da die Entwicklung des Application-Servers aber bereits als Open-Source erfolgte, übernahm die Payara Foundation [URL-PF] den Quell-Code und entwickelt ihn fortlaufend weiter. Kommerzielle Unterstützung ist über die Firma Payara Services Ltd [URL-PAY] erhältlich. Falls für Sie kein Application-Server vorgegeben ist, finden Sie in Kapitel 7 eine Kurzvorstellung von drei Alternativen und ein paar weitere Informationen zur Historie.

Wir selbst verwenden für die Beispiele dieses Buchs WildFly, da dieser nach unserem Erachten die modernste und aktuellste Implementierung eines Application-Servers ist.

1.7 Totgesagte leben länger

Der in Abschnitt 1.1 beschriebene Siegeszug von Java ist nicht einzigartig. Auch JavaScript hat einen ganz ähnlich fulminanten Siegeszug hinter sich, wenn nicht sogar einen noch

erfolgreicheren. Ältere Leser werden sich noch erinnern, dass vor etlichen Jahren dazu geraten wurde, JavaScript im Browser aus Sicherheitsgründen abzuschalten. Heute, 10 bis 20 Jahre danach, ist praktisch keine Web-Seite mehr verwendbar, wenn JavaScript ausgeschaltet ist. Hippe JavaScript-Frameworks für Benutzungsschnittstellen, etwa Angular/AngularJS, React, Vue.js usw. befinden sich in breiter Verwendung. Dieser Erfolg geht aber auch einher in einem Kommen und vor allem schnellen Gehen dieser Frameworks. Auch hier werden sich nur noch ältere Leser an Namen wie Ample SDK, Enyo, Ext JS, script.aculo.us oder YUI Library erinnern können. Allesamt JavaScript-Frameworks, die ehemals sehr angesagt waren, heute aber vom Erdboden verschwunden sind. Das gilt auch für neuere JavaScript-Frameworks und wurde auch schon auf Stack Overflow thematisiert. [URL-LCJS]

JavaServer Faces sind damit nicht zu vergleichen. Die initiale Version 1.0 wurde im März 2004 veröffentlicht, die dann auch in Produktivsystemen tatsächlich nutzbare Version 1.1 im Mai 2004. Seitdem sind eine Reihe von weiteren Versionen veröffentlicht worden, die zum Teil sehr wertvolle Neuerungen brachten und die Verwendung von JavaServer Faces erheblich vereinfachten. Trotzdem wurde die Rückwärtskompatibilität praktisch vollständig gewahrt. Auf neueren Application-Servern können zehn Jahre alte JSF-Anwendungen praktisch unverändert ausgeführt werden, was für gängige JavaScript-Frameworks völlig undenkbar ist.

Trotz dieser sehr attraktiven Eigenschaften wurde JavaServer Faces schon mehrfach der Tod vorhergesagt. Eine sehr gravierende Vorhersage, besser Rat stammt von ThoughtWorks in ihrem Technology Radar zu JavaServer Faces [URL-TWTR] aus dem Jahr 2014, in dem von der Verwendung von JavaServer Faces abgeraten wird und JavaServer Faces auf „*HOLD*“ gesetzt werden. Als Grund wird lediglich genannt, dass JavaServer Faces vom zugrunde liegenden Web-Programmiermodell abstrahiert. Wir sind der Meinung, dass alle höheren Programmiersprachen und Frameworks ganz inhärent auf diese Form der Abstraktion setzen, um Dinge einfacher lösen zu können. Es ist also insbesondere kein Argument gegen JavaServer Faces.

Andererseits gibt es aber auch Stimmen, die JavaServer Faces sehr positiv sehen, etwa ein Artikel von Reza Rahman [URL-JRL], einem bekannten Java-EE-Protagonisten. Die Überschrift des Artikels lautet *Survey Confirms JSF Remains Leading Web Framework*, beruht auf einer von DZone durchgeführten Befragung im Jahr 2015 und sieht JavaServer Faces auf Platz 1 der Verwendungsstatistik von Web-Frameworks.

Die Väter von JavaServer Faces wollten mit JSF nur Spezifikation und Implementierungen eines Basissystems definieren. Dieses Basissystem sollte die Funktionalität von zunächst HTML 4, mittlerweile von HTML 5 bereitstellen. Auf dieser Basis sollte es möglich sein, weitere Komponenten und umfangreiche Komponentenbibliotheken zu entwickeln, die darüber hinausgehende Funktionalitäten anbieten, z.B. eine Baumkomponente oder Drag-and-Drop. Dieses Ziel wurde erreicht. Es gibt mehrere Anbieter, die derartige Komponentenbibliotheken kommerziell oder auf Open-Source-Basis anbieten und vertreiben. Zu den bekanntesten gehört PrimeFaces der Firma PrimeTek. Unter [URL-WUPF] führt PrimeTek Kunden auf, die PrimeFaces verwenden. Wir empfehlen dem Leser einen Blick auf diese Kundenliste, um die weite Verbreitung von JavaServer Faces tatsächlich einschätzen zu können.

1.8 Aufbau des Buchs

Die zentralen Konzepte und Komponenten der JavaServer Faces werden in

- *JavaServer Faces im Detail*, Kapitel 2,
- *Weiterführende Themen*, Kapitel 4 und
- *Spezialthemen*, Kapitel 6

beschrieben. JavaServer Faces sind mittlerweile ein so umfassendes Framework, dass eine solche Aufteilung angezeigt ist. Kapitel 2 beschreibt die ganz grundlegende Funktionsweise von JSF und die Konzepte und Komponenten, ohne die praktisch keine JSF-Anwendung auskommt. Die einzelnen Komponenten von JSF werden untereinander, vor allem aber auch mit anderen Komponenten von Java EE mit Hilfe von CDI (*Contexts and Dependency Injection*) verbunden, das in Kapitel 3 eingeführt wird. Weiterführende Themen von JSF sind dann Gegenstand des Kapitels 4. Auch diese Themen bzw. die damit verbundenen Konzepte und Programmierkonstrukte werden in JSF-Anwendungen häufig verwendet, gehen aber über das typische Hello-World-Niveau hinaus. Aus didaktischen Gründen werden die einzelnen JSF-Features der Kapitel 2 und 4 exemplarisch an kleinen Beispielen aber in der Regel ohne größeren Anwendungskontext dargestellt. Eine jeweilige Einbettung in einen größeren Anwendungskontext würde den Rahmen sprengen. In Kapitel 5 wird dies nachgeholt, indem eine größere Anwendung auf der Basis der Kapitel 2 und 4 entwickelt und vorgestellt wird. Im nächsten und letzten Abschnitt dieses Kapitels geben wir bereits einen kleinen Einblick in diese Anwendung, das *Classic Models ERP*.

Kapitel 6, *Spezialthemen*, schließt unsere direkten Ausführungen zu JavaServer Faces ab. Die Inhalte sind in dem Sinne speziell, dass nicht alle JSF-Anwendungen davon Gebrauch machen werden. Auch hier gilt aber, dass die Grenzen fließen sind, da z.B. auch Single-Page-Applications und mobile Endgeräte thematisiert werden, die zurzeit eher Hype-Themen und weit verbreitet sind. Man benötigt hierzu jedoch keine neuen Features, sondern kann mit JSF-Bordmitteln sehr erfolgreich sein, was die Einordnung erklärt.

In Kapitel 7 gehen wir auf die verwendeten Systeme ein und beschließen die Inhaltskapitel mit einem Ausblick in Kapitel 8.

Im Anhang A werden schließlich die Tags der Standardbibliotheken im Überblick kurz aufgeführt. Warum kurz? Wir glauben, dass ein Buch nicht mit den Recherchemöglichkeiten des Internets konkurrieren kann und es auch nicht versuchen sollte. Moderne IDEs blenden das JavaDoc der Tags ein und die Suchmaschinen des Internets finden relativ verlässlich weitere Informationen. Sie können aber keinen Überblick geben. Genau dies versuchen wir mit dem Anhang A zu erreichen.

Wir sind nicht nur Autor, sondern auch Buchliebhaber und selbst fleißiger Leser. Ein Buch kann, wie bereits erwähnt, die Recherchemöglichkeiten des Internets nicht einmal ansatzweise anbieten. Ein solider Aufbau, Hinweise auf verwandte Themen innerhalb des Buchs und ein umfassendes Stichwortverzeichnis können die praktische Verwendbarkeit des Buchs aber deutlich erhöhen. Wir haben deshalb viel Aufwand in die Erstellung des Stichwortverzeichnisses investiert und hoffen, dass Sie es gewinnbringend verwenden können. Bitte geben Sie dem Stichwortverzeichnis eine Chance, wenn Sie das Buch durcharbeiten, und verwenden Sie es danach als Recherchemöglichkeit.

Wenn Sie das Buch erworben haben, können Sie die E-Book-Version des Buchs unentgeltlich herunterladen. Wir raten Ihnen sehr, dies zu tun. Das E-Book ist als verlinktes, mehr-

farbiges Hypertextsystem erstellt. Sie können mit Ihrem Reader innerhalb des Buchs navigieren und Links in das Internet automatisch öffnen. Bitte probieren Sie dies aus.

1.9 Classic Models

In den weiteren Kapiteln dieses Buchs werden viele JSF-Features exemplarisch an kleinen Beispielen, in der Regel jedoch ohne Einbettung in einen größeren Anwendungskontext erläutert. Software-Entwicklung ist aber mehr als das Zusammenpacken einzelner Code-Teile. Eine sinnvolle Architektur kann so z.B. nicht entstehen. Zusätzlich müssen wir konstatieren, dass eine solche Aneinanderreihung von Features nicht didaktisch geschlossen erfolgen kann, zumindest sehen wir nicht wie. In Kapitel 5 entwickeln wir daher eine größere Anwendung, das ERP-System *Classic Models*, um einen größeren Anwendungskontext und eventuell sogar Architekturfragen diskutieren zu können. Die Daten der Anwendung basieren auf einem Eclipse-Projekt und repräsentieren typische Geschäftsobjekte eines Großhändlers von Oldtimer-Modellautos: Produkte, Kunden, Bestellungen, Zahlungen. In Kapitel 5 werden wir zeigen, wie verschiedene Anforderungen an ein solches System mit JSF und Java EE umgesetzt werden können. Hier soll lediglich ein kleiner und motivierender erster Einblick erfolgen. Das Bild 1.1 zeigt die Liste aller Mitarbeiter des Händlers im Browser.

JavaServer Faces — Kapitel 5: Classic Models

Home Mitarbeiter Kunden Bestellungen Produkte Datenbank

Mitarbeiter

Vorname	Nachname	Niederlassung	Stellenbezeichnung	E-Mail	Vorgesetzter		
Andy	Fixter	Sydney	Sales Rep	afixter@classicmodelcars.com	William Patterson	Ändern	Löschen
Anthony	Bow	San Francisco	Sales Manager (NA)	abow@classicmodelcars.com	Mary Patterson	Ändern	Löschen
Barry	Jones	London	Sales Rep	bjones@classicmodelcars.com	Gerard Bondur	Ändern	Löschen
Diane	Murphy	San Francisco	President	dmurphy@classicmodelcars.com		Ändern	Löschen
Foon Yue	Tseng	NYC	Sales Rep	ftseng@classicmodelcars.com	Anthony Bow	Ändern	Löschen
George	Vanauf	NYC	Sales Rep	gvanauf@classicmodelcars.com	Anthony Bow	Ändern	Löschen
Gerard	Bondur	Paris	Sale Manager (EMEA)	gbondur@classicmodelcars.com	Mary Patterson	Ändern	Löschen
Gerard	Hernandez	Paris	Sales Rep	ghernande@classicmodelcars.com	Gerard Bondur	Ändern	Löschen
Jeff	Firrelli	San Francisco	VP Marketing	jfirrelli@classicmodelcars.com	Diane Murphy	Ändern	Löschen
Julie	Firrelli	Boston	Sales Rep	jfirrelli@classicmodelcars.com	Anthony Bow	Ändern	Löschen
Larry	Bott	London	Sales Rep	lbott@classicmodelcars.com	Gerard Bondur	Ändern	Löschen
Leslie	Jennings	San Francisco	Sales Rep	ljennings@classicmodelcars.com	Anthony Bow	Ändern	Löschen
Leslie	Thompson	San Francisco	Sales Rep	lthompson@classicmodelcars.com	Anthony Bow	Ändern	Löschen
Loui	Bondur	Paris	Sales Rep	lbondur@classicmodelcars.com	Gerard Bondur	Ändern	Löschen
Mami	Nishi	Tokyo	Sales Rep	mnishi@classicmodelcars.com	Mary Patterson	Ändern	Löschen
Martin	Gerard	Paris	Sales Rep	mgerard@classicmodelcars.com	Gerard Bondur	Ändern	Löschen
Mary	Patterson	San Francisco	VP Sales	mpatterso@classicmodelcars.com	Diane Murphy	Ändern	Löschen
Pamela	Castillo	Paris	Sales Rep	pcastillo@classicmodelcars.com	Gerard Bondur	Ändern	Löschen
Peter	Marsh	Sydney	Sales Rep	pmarsh@classicmodelcars.com	William Patterson	Ändern	Löschen
Steve	Patterson	Boston	Sales Rep	spatterson@classicmodelcars.com	Anthony Bow	Ändern	Löschen
Tom	King	Sydney	Sales Rep	tking@classicmodelcars.com	William Patterson	Ändern	Löschen
William	Patterson	Sydney	Sales Manager (APAC)	wpatterson@classicmodelcars.com	Mary Patterson	Ändern	Löschen
Yoshimi	Kato	Tokyo	Sales Rep	ykato@classicmodelcars.com	Mami Nishi	Ändern	Löschen

Classic Models | © Bernd Müller

Bild 1.1 Anzeige aller Mitarbeiter

Für eine Implementierung dieser Mitarbeiterliste benötigt man mindestens

- eine Klasse zur Repräsentation eines Mitarbeiters und idealerweise einen Mechanismus, um einen solchen Mitarbeiter in relationalen Datenbanken zu verwalten,
- eine Klasse, die als Dienst typische CRUD-Operationen (Create, Read, Update, Delete) auf diesen Mitarbeitern anbietet, idealerweise mehrbenutzerfähig und transaktional,
- einen JSF-Controller, der über diesen Dienst die Liste aller Mitarbeiter erstellen lässt und sie in einer JSF-Seite aufbereitet sowie die JSF-Seite selbst.

Wir beginnen mit der Klasse zur Repräsentation eines Mitarbeiters. Die Daten sind durch das bereits erwähnte Eclipse-Projekt vorgegeben und werden in Kapitel 5 genauer beschrieben. Wenn Java EE zur Verfügung steht, ist Java Persistence API, kurz JPA, das Mittel der Wahl, wenn Daten in relationalen Datenbanken verwaltet werden sollen. Wir gehen auf JPA ebenfalls in Kapitel 5 näher ein. Die in den folgenden Listings dargestellten Code-Ausschnitte entsprechen nicht dem Original, sondern sind wegen des einführenden Charakters dieses Abschnitts zum Teil vereinfacht dargestellt. Das Listing 1.1 zeigt die Klasse `Employee`, die mit entsprechenden JPA-Annotationen versehen ist.

Listing 1.1 Das Entity `Employee` (Ausschnitt)

```
@Entity
@NamedQuery(name = "Employee.findAll",
            query = "SELECT e FROM Employee e")
public class Employee {

  @Id
  private Integer id;

  private String firstName;
  private String lastName;
  private String email;
  private String jobTitle;
  ...
```

Durch die Annotation `@Entity` wird die Klasse zu einem JPA-Entity und damit zu einer persistenten Klasse. Die Annotation `@NamedQuery` definiert für eine Anfrage der Java Persistence Query Language (JPQL) einen Namen, unter dem die Anfrage später ausgeführt werden kann. Um den SQL-Primärschlüssel zu kennzeichnen, wird die Annotation `@Id` verwendet. Die in Bild 1.1 dargestellte Spalte Niederlassung ist eine Assoziation zur Klasse `Office`, die Spalte Vorgesetzter ist eine rekursive Selbstassoziation. Beide sind in Listing 1.1 aus Vereinfachungsgründen nicht dargestellt.

Eine Möglichkeit der einfachen Verwendung von JPA-Entities sind Enterprise Java Beans (EJBs). In Listing 1.2 wird eine solche EJB definiert.

Listing 1.2 Die EJB `EmployeeService` (Ausschnitt)

```
@Stateless
public class EmployeeService {
```

```
  @PersistenceContext
  EntityManager em;

  // CRUD-Operationen nicht dargestellt, da im Beispiel nicht verwendet

  public List<Employee> findAll() {
    return em.createNamedQuery("Employee.findAll", Employee.class)
             .getResultList();
  }

}
```

Die Annotation `@Stateless` definiert eine stateless EJB, die vom EJB-Container verwaltet wird. Mit der Annotation `@PersistenceContext` wird eine vom Application-Server verwaltete Instanz des Interface `EntityManager` injiziert, die die Verbindung zur Datenbank darstellt. Mit ihrer Hilfe wird die in Listing 1.1 definierte JPQL-Anfrage tatsächlich erzeugt und ausgeführt. Resultat ist eine Liste von Employees, die nun von JSF verwendet werden kann. Der entsprechende JSF-Controller ist in Listing 1.3 abgebildet.

Listing 1.3 Der JSF-Controller `EmployeesController` (Ausschnitt)

```
1  @Named
2  @RequestScoped
3  public class EmployeesController {
4
5    @Inject
6    EmployeeService employeeService;
7
8    public List<Employee> getEmployees() {
9      return employees = employeeService.findAll();
10   }
11
12 }
```

Hier wird in den Zeilen 5/6 die EJB des Listings 1.2 injiziert, um dann in Zeile 9 die entsprechende Methode zum Finden aller Mitarbeiter aufzurufen. In Zeile 1 wird mit der Annotation `@Named` ein Name für eine Instanz dieser Klasse vergeben, um diesen Namen in Expression-Language-Ausdrücken innerhalb von JSF-Seiten verwenden zu können. Die Klasse erscheint mit ihrer einzigen Methode, die an ein anderes Objekt deligiert, zunächst wenig sinnvoll. Aus Architekturüberlegungen ist es jedoch sehr sinnvoll, zwischen Controller-Klassen des UIs und Diensteklassen zu unterscheiden und diese nicht zu vermengen. Unabhängig von solchen Überlegungen zeigt Listing 1.4 den entsprechenden Ausschnitt der JSF-Seite, deren gerendertes Resultat in Bild 1.1 auf Seite 11 der Ausgangspunkt unserer Untersuchung war.

Listing 1.4 JSF-Code zur Anzeige aller Mitarbeiter, dargestellt in Bild 1.1

```
1  <h:panelGrid>
2    <f:facet name="header">Mitarbeiterübersicht</f:facet>
3    <h:dataTable value="#{employeesController.employees}"
```

```
4                 var="employee">
5       <h:column>
6         <f:facet name="header">Vorname</f:facet>
7         #{employee.firstName}
8       </h:column>
9       <h:column>
10        <f:facet name="header">Nachname</f:facet>
11        #{employee.lastName}
12      </h:column>
13      <h:column>
14        <f:facet name="header">Niederlassung</f:facet>
15        #{employee.office.city}
16      </h:column>
17      <h:column>
18        <f:facet name="header">Stellenbezeichnung</f:facet>
19        #{employee.jobTitle}
20      </h:column>
21      <h:column>
22        <f:facet name="header">E-Mail</f:facet>
23        #{employee.email}
24      </h:column>
25      <h:column>
26        <f:facet name="header">Vorgesetzter</f:facet>
27        #{employee.reportsTo.firstAndLastName}
28      </h:column>
29      <h:column>
30        <h:button outcome="employee" value="Ändern">
31          <f:param name="employeeId" value="#{employee.id}" />
32        </h:button>
33      </h:column>
34    ...
35    </h:dataTable>
36  </h:panelGrid>
```

Der mit der `@Named`-Annotation vergebene Name ist im Default-Fall der kleingeschriebene Klassenname, hier also `employeesController`. Dieser wird in Zeile 3 in einem Expression-Language-Ausdruck, syntaktisch durch „`#{ ...}`“ definiert, verwendet. Die Methode `getEmployees()` wird durch den zweiten Teil des Expression-Language-Ausdrucks `employees` aufgerufen. Das JSF-Tag `<h:dataTable>` erzeugt eine Tabelle, die mit `<h:column>` ihre Spalten definiert. Der Expression-Language-Ausdruck `#{employeesController.employees}` definiert durch das Attribut `value` den Wert der Tabelle. Mit dem Attribut `var` wird eine Variable definiert, die über die Werte des `value`-Attributs iteriert. In den Spalten der Tabelle werden so leicht erkennbar die Property-Werte der Employees, Zeile für Zeile, aufgebaut.

Wir wollen an dieser Stelle die Diskussion des Beispiels abbrechen, wohl wissend, dass nicht alle Detailfragen beantwortet sind. Das Beispiel hat hier ausschließlich motivierenden Charakter und wird nach den weiteren Kapiteln des Buchs für den Leser sicher voll und ganz verständlich. An dieser Stelle soll lediglich eines festgestellt werden: der Aufwand, um Daten aus einer Datenbank zu lesen und als HTML-Seite darzustellen, ist sehr gering, eventuell minimal. Es lohnt sich, JSF und Java EE etwas genauer anzuschauen.

2 JavaServer Faces im Detail – die Grundlagen

Nachdem wir mit Classic-Models-ERP einen ersten Kontakt mit JavaServer Faces hatten, ist es nun an der Zeit, die Grundlagen der JavaServer Faces detailliert zu erörtern. Dieses Kapitel beschreibt 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 Anfragebearbeitung wird die Antwort an den Client geschickt. Diese Antwort ist aber immer ein Spiegelbild der Komponenten auf dem Server und kein selbstständiges Artefakt.

Doch wie kann ein über die Jahre gereiftes und damit auch über die Jahre gewachsenes Framework, wie JavaServer Faces, dem Leser nahegebracht werden, ohne den Leser zu erschlagen? Die JSF-Spezifikation umfasste in der Version 1.0 298 Seiten, die Version 2.3 mittlerweile 468 Seiten. Dazu sollte die Einführung natürlich didaktisch geschickt und motiviert sowie sehr praxisnah sein. Die Praxisnähe könnte durch ein praxisnahes Beispiel, wir wir es in Abschnitt 1.9 mit dem Classic-Models-ERP-System versucht haben, erreicht werden. Dies aber mit konzeptionell zusammenhängenden Themen und einer gewissen Entwicklung von Thema zu Thema einhergehen zu lassen? Sehr schwierig!

Wir haben uns entschlossen, JavaServer Faces in diesem Kapitel an Themen orientiert einzuführen und damit in relativ kleinen, nicht direkt zusammenhängenden Häppchen. So gibt es etwa einen Abschnitt über das Bearbeitungsmodell einer JSF-Anfrage, einen Abschnitt über die Expression-Language und einen Abschnitt über die Validerung und Konvertierung. Diese werden *in Summe* in einer praxisnahen Anwendung eher weniger benötigt, erleichtern allerdings das Verständnis ganz ungemein. Hinzu kommt, dass wir durch sehr kleine und modulare Beispiele das Vertrautmachen mit der Materie unterstützen, eventuell sogar erst ermöglichen. Schließlich kommen wir sicher dem allgemeinen Zeitgeist von Copy/Paste aus Stack-Overflow entgegen, wenn wir Konzepte, Probleme und deren Lösungen zusammenhängend in kurzer Form darstellen. Wir sind allerdings auch davon überzeugt, dass wir uns, was die thematische Gesamtheit angeht, ganz ungemein von Stack-Overflow unterscheiden.

2.1 Bearbeitungsmodell einer JSF-Anfrage

JavaServer Faces werden mit Hilfe des Servlet-API 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 Umgehung 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. Konvertierungen und Validierungen 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 2.1 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 2.1 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*, da die Formularinhalte per HTTP POST an dasselbe URL des Formulars zurückgeschickt werden. 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 2.1 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 Bearbeitung einer JSF-Anfrage beginnt, wenn das Faces-Servlet den HTTP-Request erhalten hat. Es gibt insgesamt sechs zu unterscheidende Bearbeitungsphasen, die wiederum in zwei übergeordnete Phasen zusammengefasst werden: die Phasen 1 bis 5 in die *Ausführen*-Phase (Execute Phase), die Phase 6 in die *Rendern*-Phase (Render Phase). Zwischen diesen sind Event-Verarbeitungsphasen vorgesehen. Bild 2.1 veranschaulicht dies grafisch. Dabei zeigt das Bild nur das Standardverhalten beim Durchlauf der Phasen. Einige Möglichkeiten, von diesem Standardverhalten abzuweichen, beschreiben wir im Text.

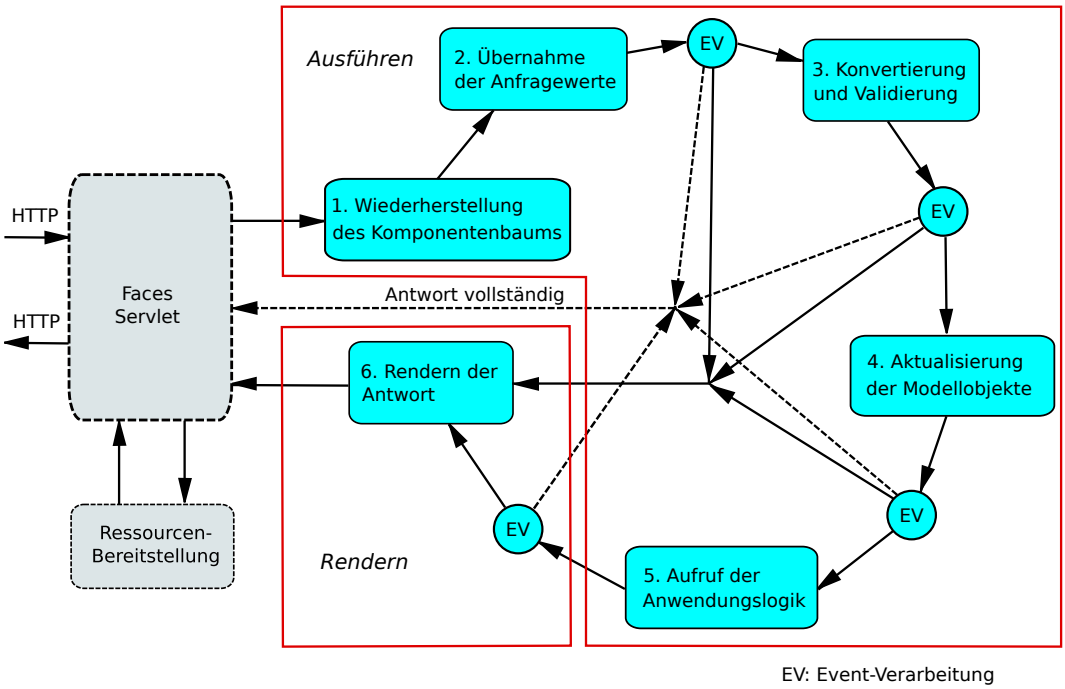

Bild 2.1 Bearbeitungsmodell einer JSF-Anfrage

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 4.9) bzw. in einer Servlet-Umgebung der Version 3.0 und höher 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 interessierte 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 durch JSF verzichten und direkt an das Ende der Bearbeitung springen. Dies ist durch die gestrichelten Linien in Bild 2.1 angedeutet. Beispiele hierfür sind binäre Daten, wie etwa ei-

ne 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 die Fehlerursache enthält.

Mit der Einführung von Ajax-Requests ändert sich das dargestellte Verhalten etwas, da der Baum partiell ausgeführt (Partial-View-Processing, Phasen 1 bis 5) und partiell gerendert (Partial-View-Rendering, Phase 6) werden kann. Dies bedeutet, dass sowohl das Ausführen als auch das Rendern auf bestimmte Komponenten des Baums – auch andere – beschränkt wird. Das Rendern wird auch nicht komplett bis zu einer kompletten HTML-Seite durchgeführt, sondern die genannten Komponenten werden als XML an den Client zurückgegeben. Wir gehen auf die Ajax-Besonderheiten in Abschnitt 4.5 ein.

2.1.1 Wiederherstellung des Komponentenbaums

JSF-Komponenten besitzen einen Zustand, der z.B. bei einer Eingabe den aktuellen Wert, bei einer Drop-down-Liste 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, genauer deren Zustände, 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 4.9.3 erläutert. Jede View hat eine View-Id, die das URI der Anfrage ohne Anwendungsname bzw. der Pfad der zugehörigen Seitendefinition auf dem Server darstellt. Die View-Id wird in der Session gespeichert und über ein verstecktes Formularfeld mit Name `javax.faces.ViewState` identifiziert. Bei einer Anfrage kann daher entschieden werden, ob die Anfrage von einer JSF-Seite initiiert wurde (Alternative 1 in Tabelle 2.1) oder ob die Seite zum ersten Mal besucht wird (Alternative 2 in Tabelle 2.1). 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 Komponentenbaum 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.

Wird die Seite zum ersten Mal besucht (Alternative 2 in Tabelle 2.1), sind die Phasen 2 bis 5 nicht sinnvoll. Es wird daher zur sechsten und letzten Phase, dem Rendern der Antwort, gesprungen. Es gibt allerdings eine Ausnahme von der Ausnahme. Werden sogenannte View-

Parameter verwendet, die wir in Abschnitt 4.6.2 einführen, wird ein Baum, der ausschließlich die View-Parameter enthält, erzeugt und damit alle Folgephasen durchlaufen.

Zu einem großen Teil aller Fälle einer JSF-Anwendung wird jedoch durch das Abschicken eines JSF-Formulars die JSF-Seite aufgerufen, so dass mit der Phase 2, der Übernahme der Anfragewerte weitergemacht wird.

2.1.2 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: Es soll eine ganze Zahl eingegeben werden können. Dies kann relativ einfach mit dem `<h:inputText>`-Element erfolgen:

```
<h:inputText id="in" value="#{lifecycleBean.input}" required="true" />
```

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

```
<input id="form:in" type="text" name="form:in" />
```

Man erkennt, dass sowohl der Wert des Attributs `id` als auch der Wert des Attributs `name` generiert wurden. Die Art der Generierung ist in der Spezifikation beschrieben und besteht in diesem Fall aus der Id des umgebenden Formulars (`form`), einem Doppelpunkt und dem in `<h:inputText>` verwendeten Wert des Attributs `id`. Wir gehen in Abschnitt 4.3 näher auf den Algorithmus dieser Bezeichnergenerierung ein. Der folgende Code ist dem POST-Request entnommen:

```
form=form&form%3Ain=22&...
```

Man erkennt zwischen den beiden &-Zeichen eingeschlossen die Zeichen `&form%3Ain=22&`. Der Doppelpunkt ist durch `%3A` codiert, so dass der übertragene Wert „`22`“ ist. Es ist Aufgabe der Phase 2, 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 Befehlskomponenten besitzen ein boolesches Attribut `immediate`. Ist dieses bei Eingabekomponenten auf `true` gesetzt, finden Validierung und Konvertierung bereits in der Phase 2 *Übernahme der Anfragewerte* und nicht in der nächsten Phase, der Validierung, statt. Ist das Attribut `immediate` bei Befehlskomponenten gesetzt, findet der Aufruf der Action-Methoden bzw. Action-Listener am Ende dieser Phase und nicht in der Phase 5 *Aufruf der Anwendungslogik* statt. Wir gehen auf beide Alternativen in den entsprechenden Abschnitten 2.4.9 und 2.5.5 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, die Bearbeitung der Anfrage komplett beenden oder mit der Validierungsphase beginnen.

2.1.3 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.

Das Beispiel könnte etwa um eine Bereichsprüfung erweitert werden:

```
<h:inputText id="in" value="#{lifecycleBean.input}" required="true">
  <f:validateLongRange minimum="0" maximum="100" />
</h:inputText>
```

Hier wird nicht nur der eingegebene Wert mit `<f:validateLongRange>` auf einen bestimmten Bereich validiert, sondern zunächst durch das `required`-Attribut der Eingabekomponente geprüft, ob überhaupt eine Eingabe vorhanden ist. Falls keine Eingabe vorhanden ist, wird dies durch die Fehlermeldung

```
form:in: Überprüfungsfehler: Wert ist erforderlich.
```

angezeigt. Falls eine Eingabe vorhanden ist, sich diese allerdings nicht im erwarteten Bereich befindet, lautet die Fehlermeldung

```
form:in: Überprüfungsfehler: Das angegebene Attribut liegt
nicht zwischen den erwarteten Werten 0 und 100.
```

XML-Namensräume

Im Beispiel wurden die JSF-Tags `<h:inputText>` und `<f:validateLongRange>` verwendet. Damit die Verwendung in syntaktisch korrektem XML, genauer XHTML, resultiert, müssen zwei XML-Namensräume in die Quelldatei eingebunden werden. Es sind dies die HTML- und die Core-Bibliothek mit den XML-Namensräumen `http://xmlns.jcp.org/jsf/html` und `http://xmlns.jcp.org/jsf/f`, die typischerweise mit den Präfixen `h` und `f` verwendet werden. Wir raten, dieser Konvention zu folgen. In den Listings verzichten wir auf Unnötiges und somit auch auf Namensräume.

Doch warum erfolgt die Fehlermeldung in deutscher Sprache? JavaServer Faces unterstützen ein sehr mächtiges Lokalisierungsmodell, das wir in Abschnit 4.2 erläutern. Das Beispiel nutzt dies jedoch nicht, so dass die Default-Lokalisierung der zugrunde liegenden JVM verwendet wird, die hier *deutsch* zurückliefert.

Die in der Spezifikation *Process Validation* genannte Phase besteht neben der Validierung in der Regel auch aus einer zuvor vorzunehmenden Konvertierung. Im Beispiel muss etwa aus dem vom Benutzer im Browser eingegebenen String `"20"` die Integer-Zahl `20` erzeugt werden, um die Bereichsvalidierung durchführen zu können. Die primitiven Java-Typen, deren Wrapper sowie einige weitere Konvertierungen werden von JSF automatisch vorgenommen. Benutzerdefinierte Konvertierungen sind ebenfalls möglich. Wir widmen uns dem Bereich der Validierung und Konvertierung im Abschnitt 2.4.

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-Change-Event geworfen und an registrierte Listener weitergegeben. Diese Listener können nun zur Render-Phase springen, direkt die Antwort erzeugen oder zur Aktualisierung der Modellobjekte übergehen.

2.1.4 Aktualisierung der Modellobjekte

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

```
<h:inputText id="in" value="#{lifecycleBean.input}"
             required="true" />
```

ist der Ausdruck `"#{lifecycleBean.input}"` dafür zuständig. Der Ausdruck ist ein *Werteausdruck* der *Expression Language (EL)*, auf die wir gleich im nächsten Abschnitt 2.2 näher eingehen.

Der erste Teil `lifecycleBean` ist der Name einer Managed Bean. Die JSF-Implementierung, genauer die EL-Implementierung, sucht also nach einer Managed Bean mit diesem Namen. Das Property `input` des Ergebnisobjekts bekommt dann den Wert der UI-Komponente zugewiesen. Die entsprechende Klasse sollte also mindestens die folgende Form haben:

```
@Named
@RequestScoped
public class LifecycleBean {

  private Integer input;

  public LifecycleBean() { }

  public Integer getInput() {
    return input;
  }

  public void setInput(Integer input) {
    this.input = input;
  }
}
```

Dabei sorgt die Annotation `@Named` dafür, dass die erzeugte Instanz den kleingeschriebenen Klassennamen als EL-Namen zugewiesen bekommt. Auf die Beschreibung der Annotation `@RequestScoped` verzichten wir zunächst.

Wie am Ende jeder Phase werden möglicherweise wieder Events geworfen, Listener informiert, und eventuell wird an das Ende der Bearbeitung gesprungen.

2.1.5 Aufruf der Anwendungslogik

Bis zu diesem Zeitpunkt haben wir noch keinen anwendungsspezifischen 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 mit ins Spiel kommen. Dies geschieht durch Listener, die auf Action-Events registriert wurden, die durch das Betätigen von Schaltflächen oder Hyperlinks ausgelöst werden können. Es gibt zwei Arten von Action-Listenern, wobei die einfachste Verwendung die der automatisch für Befehlskomponenten registrierten Default-Action-Listener ist. Die Verwendung erfolgt durch Bindung einer Action-Methdode an die Befehlskomponente, im folgenden Beispiel ein `<h:commandButton>`.

```
<h:commandButton action="#{lifecycleBean.assign}"
                 value="Abschicken" />
```

Hier wird die Action-Methode `assign()` registriert. Eine Action-Methode hat als Rückgabetyp `Object` oder `void` und keinen Parameter:

```
public void assign() {
  output = input;
}
```

Im Falle des Rückgabetyps `Object` wird der Wert der `toString()`-Methode des Objekts zur Navigation verwendet. In der Regel wird als Rückgabetyp daher eher `String` als `Object` verwendet. In unserem Fall wird nicht navigiert. Action-Methoden sind nicht auf die Anwendungslogik beschränkt. Sie können z.B. auch Events generieren, Anwendungsmeldungen erzeugen oder gar die Antwort selbst rendern.

2.1.6 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. Es ist nun jedoch zuerst an der Zeit, das Rätsel um unser einführendes Beispiel zu lösen und den kompletten Source-Code darzustellen. Listing 2.1 zeigt das JSF-Seitenfragment, das die Eingabe einer Zahl, deren Prüfung auf einen bestimmten Bereich und die anschließende Ausgabe realisiert.

Listing 2.1 JSF-Seitenfragment zur Ein-/Ausgabe

```
Bitte geben Sie eine ganze Zahl zwischen 0 und 100 ein:
<br />
<h:inputText id="in" value="#{lifecycleBean.input}" required="true">
  <f:validateLongRange minimum="0" maximum="100" />
</h:inputText>
<br />
<h:commandButton id="action" action="#{lifecycleBean.assign}"
                 value="Abschicken" />
```

```
<br />
Ausgabe: <h:outputText id="out" value="#{lifecycleBean.output}" />
```

Das Listing 2.2 zeigt die JSF-Bean, die hinter dem Formular die Arbeit macht. Man erkennt die beiden Properties `input` und `output` sowie die bereits erwähnte Action-Methode `assign()`.

Listing 2.2 Die JSF-Bean `LifecycleBean`

```
@Named
@RequestScoped
public class LifecycleBean {

  private Integer input;
  private Integer output;

  public LifecycleBean() { }

  public void assign() {
    output = input;
  }

  public Integer getInput() {
    return input;
  }
  public void setInput(Integer input) {
    this.input = input;
  }

  public Integer getOutput() {
    return output;
  }
  public void setOutput(Integer output) {
    this.output = output;
  }

}
```

Zurück zum Bearbeitungsmodell einer JSF-Anfrage. Die Zielsprache des Renderns der Antwort in Phase 6 ist durch die Spezifikation bewusst offen gelassen worden. Es sind verschiedene Alternativen denkbar, z.B. HTML, WML, SVG oder gar PDF. Seit JSF 2.2 ist HTML 5 der Default.

Während in der zweiten Phase, der Übernahme der Anfragewerte, die Komponentenwerte decodiert werden mussten, müssen sie nun codiert werden. Im Beispiel ist das einfach, da nur die beiden Ein- und Ausgabewerte in Strings überführt und dann HTML 5 für die komplette Seite erzeugt werden muss.

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

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.

Bekanntmachen mit der Technik

- Laden Sie das Projekt *jsf-im-detail* herunter.
- Erzeugen Sie die Anwendung durch `mvn package`.
- Deployen Sie die Anwendung, z.B. durch Kopieren in das entsprechende Application-Server-Verzeichnis.
- Rufen Sie die Anwendung im Browser auf: `localhost:8080/jsf-im-detail`.
- Falls Sie Hilfe bei den jeweiligen Schritten benötigen, finden Sie diese in Kapitel 7.

Nachvollziehen des Lebenszyklus

- Studieren Sie die Klassen `LifecycleBean` und `LifecycleObserver` sowie die JSF-Seite `lifecycle-observer.xhtml`.
- Die Klasse `LifecycleObserver` ist ein sogenannter Phase-Listener. Diesen müssen Sie in der Datei `faces-config.xml` aktivieren, indem Sie die entsprechenden Kommentarzeichen entfernen.
- Bauen und deployen Sie das Projekt nochmals.
- Beobachten Sie die Log-Datei Ihres Application-Servers und führen Sie das Beispiel im Browser aus.

Lebenszyklus mit Ajax

Das Beispiel existiert im Projekt auch in einer Ajax-Variante mit Verwendung von `<f:ajax>`. Welche Änderungen im Log können Sie erkennen? Falls Sie keine Erfahrung mit Ajax besitzen, können Sie die Aufgabe nach Lektüre von Abschnitt 2.7 durchführen.

Lebenszyklus mit anwendungsdefinierten Konvertierern und Validierern

Als Vorgriff auf die entsprechenden Abschnitte zur anwendungsdefinierten Konvertierung und Validierung in Abschnitt 2.4 können Sie das Beispiel auch in dieser Variante durchspielen.

2.2 Expression-Language

Die JSP Standard Tag Library (JSTL) und JavaServer Pages (JSP) führten lange vor der Geburt von JavaServer Faces 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 als JSP an eine Expression-Language, so dass für JSF eine eigene Expression-Language definiert wurde. Es stellte sich jedoch schnell heraus, dass die Existenz zweier derartiger Sprachen wenig sinnvoll ist, was letztendlich zur Definition der *Unified Expression-Language* führte.

Die Unified Expression-Language wurde als Teildokument der JavaServer-Pages-Spezifikation 2.1 definiert [URL-JSR245] und wird in dieser Form seit JSF 1.2 verwendet. Die Unified Expression-Language unterscheidet zwischen sofortiger Auswertung (die sogenannte *Immediate Evaluation*, beginnend mit einem $-Zeichen) zum Zeitpunkt des Renderns der Seite (bei JSPs der Compile-Zeitpunkt) und der verzögerten oder zeitversetzten Auswertung (die sogenannte *Deferred Evaluation*, beginnend mit einem #-Zeichen) zur Laufzeit. Die zeitverzögerte Auswertung macht sich in JSF insbesondere dadurch bemerkbar, dass EL-Ausdrücke zweimal ausgewertet werden können, und zwar während der Übernahme der Anfragewerte (Phase 2) als auch des Renderns der Antwort (Phase 6). Werteausdrücke werden daher sowohl schreibend (Phase 2) als auch lesend (Phase 6) verwendet.

Da JSF-Seiten nur noch mit XHTML und nicht mehr mit JSP definiert werden und JavaServer Pages nicht mehr weiterentwickelt werden, ist für uns nur noch die JSF-artige Verwendung von EL sinnvoll, also die mit Verwendung des #-Zeichens. Außerdem wurde die Expression-Language in eine komplett eigenständige Spezifikation ausgelagert, die Expression Language Specification 3.0 [URL-JSR341]. Diese ist bereits seit Java EE 7 fester Bestandteil der Enterprise-Edition und damit auch in allen aktuellen Application-Servern enthalten. Sie wird von uns hier verwendet, auch wenn sie in der JSF-Spezifikation als *optional* gekennzeichnet ist.

EL-Versionen

Alle uns bekannten Application-Server verwenden die Expression-Language in der Version 3.0. Wir verwenden im Folgenden ebenfalls diese Version, auch wenn sie in der Spezifikation als *optional* gekennzeichnet ist und die für JSF 2.3 verpflichtende Version die Version 2.2 ist.

2.2.1 Syntax

Die Expression-Language enthält Konzepte, wie man sie auch in JavaScript und XPath findet. In der Expression-Language navigiert man durch eine Punktnotation über Objekt-Properties, so wie man in XPath im Baum des XML-Dokuments navigiert. JSF folgt dabei

der JavaBeans-Spezifikation [URL-JB], die einen Default-Konstruktor und ein Property voraussetzt. Ein Property ist dabei ein `private` Field und ein `public` Getter/Setter-Paar. Im EL-Ausdruck wird nur der Property-Name verwendet, der bei der Auswertung entsprechend seiner Verwendung mit den Präfixen `get` und `set` erweitert wird. Die Großschreibung wird den Java-Konventionen angepasst. Wir gehen darauf später noch genauer ein.

Objekt-Properties

Obwohl JSF der JavaBeans-Spezifikation mit `private` Field und `public` Getter/Setter-Paar folgt, können Sie auf den Setter verzichten, wenn nur lesend auf das Property zugegriffen wird.

Zunächst wollen wir aber die grundlegende Syntax der Expression-Language einführen. Ausdrücke der Expression-Langage 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 komplexeren arithmetischen oder logischen Ausdruck enthalten, sogar Lambda-Ausdrücke sind möglich. Kombinationen der genannten Alternativen sind in gewisser Form ebenfalls möglich. Das Verschachteln von Ausdrücken der Art `#{feld[#{i}]}` ist allerdings nicht erlaubt. Anstatt Anführungszeichen ist alternativ die Verwendung einfacher Apostrophe erlaubt. Wir empfehlen die konsequente Verwendung von Anführungszeichen.

2.2.2 Werteausdrücke

Über einen *Werteausdruck* (engl. Value Expression) kann der Wert einer UI-Komponente 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 und in Abschnitt 2.2.4 näher erläutert.

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*.

Um kleinere Beispiele für Werteausdrücke entwickeln zu können, benötigen wir zunächst eine *Managed Bean*. Wir haben bereits bei der Classic-Models-ERP-Anwendung in Abschnitt 1.9 und bei der Vorstellung des Bearbeitungsmodells einer JSF-Anfrage in Abschnitt 2.1 Managed Beans verwendet, ohne allerdings das Konzept einer Managed Bean

einzuführen und die wichtige Frage nach einer sinnvollen Namensgebung zu diskutieren. Auch an dieser Stelle wollen wir den Leser lediglich auf die Problematik hinweisen und widmen uns den genannten Aufgaben dann in Abschnitt 2.3.

Listing 2.3 JSF-Bean für EL-Ausdrücke

```
@Named("elController")
@RequestScoped
public class ExpressionLanguageController {

    private String name = "Übungen mit der Expression-Language";

    private Integer year = LocalDate.now().getYear();

    private String[] array = new String[]{ "eins", "zwei", "drei" };

    private List<Integer> list = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8,
        9);

    private Map<String, String> map = new HashMap<String, String>() {{
            put("eins", "Erster Map-Eintrag");
            put("zwei", "Zweiter Map-Eintrag");
            put("drei", "Dritter Map-Eintrag");
    }};

    public ExpressionLanguageController() { }

    // noch Getter und Setter
    ...
}
```

Die Klasse `ExpressionLanguageController` in Listing 2.3 ist eine einfache Managed Bean, die wir kurz *JSF-Bean* nennen wollen, und deren Properties in den folgenden EL-Beispielen verwendet werden. Bei den verwendeten Typen haben wir ein repräsentatives Spektrum von String und Integer sowie Array, Liste und Map. Anzumerken ist, dass für alle gezeigten privaten Fields nach der JavaBean-Spezifikation und den JSF-Anforderungen öffentliche Getter und Setter existieren sollten. Da unsere Beispiele nur lesenden Zugriff auf die Properties haben, kann auf die Getter verzichtet werden.

Das Listing 2.4 zeigt die Verwendung der JSF-Bean in einer JSF-Seite. Da der Default-Name der Bean der kleingeschriebene Klassenname ist, haben wir ihn als `value`-Attribut der `@Named`-Annotation etwas abgekürzt.

Listing 2.4 Die ersten Zeilen der Datei `value-expressions.xhtml`

```
<h:panelGrid columns="2">
  <f:facet name="header">Einfache Werteausdrücke</f:facet>

  <h:outputText value="\#{elController.name}" />
  <h:outputText value="#{elController.name}" />
```

```
    <h:outputText value="\#{elController['name']}" />
    <h:outputText value="#{elController['name']}" />
    ...
  </h:panelGrid>
```

In diesem Abschnitt zur Expression-Language wollen wir den Code der JSF-Seiten und das Ergebnis des Renderns der Seite etwas kompakter und direkt gegenübergestellt darstellen. Dazu werden alle Beispiele so aufgebaut sein, dass der Quell-Code den EL-Ausdruck jeweils doppelt enthält, wobei er bei der ersten Verwendung mit einem umgekehrten Schrägstrich entwertet wird. Das `<h:panelGrid>`-Element erzeugt eine Tabelle, die durch die Verwendung des `columns`-Attributs zwei Spalten erhält: Die erste enthält den EL-Ausdruck, die zweite deren Wert. Das Bild 2.2 zeigt die Darstellung des Seitenausschnitts von Listing 2.4, wobei das Listing nur für die ersten drei Zeilen der Tabelle im Bild verantwortlich zeichnet.

Einfache Werteausdrücke	
#{elController.name}	Übungen mit der Expression-Language
#{elController['name']}	Übungen mit der Expression-Language
Dies sind einfache #{elController.name} im Jahr #{elController.year}	Dies sind einfache Übungen mit der Expression-Language im Jahr 2018
#{elController.array[0]}	eins
#{elController.list[4]}	4
#{elController.list.get(5)}	5
#{elController.map['zwei']}	Zweiter Map-Eintrag
#{elController.map[elController.array[2]]}	Dritter Map-Eintrag

Bild 2.2 Beispiele für Werteausdrücke (`value-expressions.xhtml`)

Im Beispiel verwenden wir nur Ausgabekomponenten, die Werteausdrücke werden daher ausschließlich lesend interpretiert. Bei der Verwendung einer Eingabekomponente könnten sie auch zum Setzen eines Property verwendet werden. Alle EL-Ausdrücke verwenden die JSF-Bean mit Namen `elController`. Das erste Beispiel zeigt, dass das Property `name` durch eine einfache Punktnotation referenziert wird. Bitte verdeutlichen Sie sich noch einmal, dass die Auswertung des Ausdrucks den Getter `getName()` aufruft. Properties können alternativ auch mit der Klammernotation referenziert werden, so dass die beiden ersten Zeilen semantisch äquivalent sind. Wir raten zur Verwendung der Punktnotation. Die dritte Zeile verbindet (mehrfach) ein String-Literal mit einem Werteausdruck.

Für den Zugriff auf Arrays, Listen oder Maps übernimmt die Expression-Language die Java-Array-Notation der eckigen Klammern, weshalb wir von deren Verwendung bei einfachen Properties abgeraten hatten. Der Zugriff auf Elemente eines Arrays erfolgt in der Java-üblichen Notation, die auch für Listen übernommen wird. Auf Listenelemente könnte aber auch über die `get()`-Methode zugegriffen werden. 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. Die Verwendung von Anführungszeichen ist bei JSF nicht sinnvoll, da der ganze `value`-Wert ja durch Anführungszeichen eingeschlossen wird.

Zu guter Letzt ist die Expression-Language eine komplette Ausdruckssprache, so dass sich Ausdrücke auch verschachteln lassen. Die letzte Zeile ist ein Beispiel hierfür, wobei wir von derartigen Ausdrücken aus offensichtlichen Gründen abraten.

Boolesche Properties

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.

2.2.3 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, bei komplexeren Ausdrücken aber angeraten ist. Tabelle 2.2 führt die Operatoren der Expression-Language auf. Die Reihenfolge innerhalb der Tabelle gibt die Präzedenz wieder; runde und eckige Klammern und Punktoperator binden also z.B. stärker als arithmetische Operatoren, diese wieder stärker als Vergleichsoperatoren.

Wir beginnen mit einfachen arithmetischen Ausdrücken, die in Bild 2.3 dargestellt sind.

Einfache arithmetische Ausdrücke	
#{17 + 4}	21
Das übernächste Jahr ist #{elController.year + 2}	Das übernächste Jahr ist 2020
#{elController.year} ist #{((elController.year % 4) == 0 ? 'ein' : 'kein')} Schaltjahr	2018 ist kein Schaltjahr

Bild 2.3 Komplexere Werteausdrücke (1) (`complex-expressions.xhtml`)

Die erste Zeile zeigt die Verwendung von Literalen, hier allerdings relativ sinnlos, da die Summe natürlich auch direkt hätte verwendet werden können. Die zweite Zeile verwendet das Bean-Property `elController.year` in einem arithmetischen Ausdruck und demonstriert gleichzeitig die gemischte Verwendung eines String-Literals und eines EL-Ausdrucks. Die dritte Zeile zeigt schließlich die Verwendung des bedingten Ausdrucks. 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.

Bild 2.4 zeigt einige Beispiele für Vergleiche und logische Ausdrücke. `true` und `false` in der ersten Zeile sind boolesche Literale. Der Vergleich des Jahres mit 2017, einmal als Ganzzahl, einmal als String, demonstriert implizite Typkonvertierungen des EL-Interpreters vor der Durchführung des Vergleichs, so dass es zu keinem Typfehler kommt. Da die Konvertierungen relativ intuitiv sind, verzichten wir auf eine Erläuterung und verweisen den interessierten Leser auf die EL-Spezifikation.

Die letzten drei Zeilen widmen sich der Verwendung des Und-Operators. In der XHTML-Syntax ist & ein reserviertes Zeichen, das die Definition einer Entity-Referenz einleitet. Der