Dirk Weil

Java EE 7

Enterprise-Anwendungsentwicklung leicht gemacht

entwickler.press

Dirk Weil
Java EE 7
Enterprise-Anwendungsentwicklung leicht gemacht

ISBN: 978-3-86802-290-2

© 2013 entwickler.press

Ein Imprint der Software & Support Media GmbH

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.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Media GmbH
entwickler.press
Darmstädter Landstraße 108
60598 Frankfurt am Main
Tel.: +49 (0)69 630089-0
Fax: +49 (0)69 630089-89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat: Sebastian Burkart
Korrektorat: Frauke Pesch
Satz: Dominique Kalbassi

Belichtung, Druck & Bindung: M.P. Media-Print Informationstechnologie GmbH, Paderborn

Cover: © Henvry | istockphoto.com

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktion jeglicher Art (Fotokopie, Nachdruck, Mikrofilm, Erfassung auf elektronischen Datenträgern oder anderen Verfahren) nur mit schriftlicher Genehmigung des Verlags. Jegliche Haftung für die Richtigkeit des gesamten Werks kann, trotz sorgfältiger Prüfung durch Autor und Verlag, nicht übernommen werden. Die im Buch genannten Produkte, Warenzeichen und Firmennamen sind in der Regel durch deren Inhaber geschützt.

Inhaltsverzeichnis

1 Java EE im Überblick **15**

1.1 Aufgabenstellung 15
1.2 Architekturmodell 15
1.3 Anwendungsbestandteile und Formate 16
1.4 Profile 19
1.5 Plattformen 20

2 CDI **21**

2.1 Was ist das? 21
2.2 Wozu braucht man das? 21
2.3 Bereitstellung und Injektion von Beans 24
2.3.1 CDI Beans 24
2.3.2 Field Injection 25
2.3.3 Bean Type 26
2.3.4 Method Injection 27
2.3.5 Constructor Injection 28
2.3.6 Bean Name 29
2.3.7 Bean Scan 30
2.4 Lifecycle Callbacks 31
2.5 Qualifier 32
2.6 Alternatives 35
2.7 Nutzung der Java-EE-Umgebung 37
2.7.1 Java EE Resources 37
2.7.2 Built-in Beans 38
2.8 Producer und Disposer 38
2.8.1 Producer Methods 38
2.8.2 Producer Fields 40
2.8.3 Disposer Methods 40
2.8.4 Introspektion des Injektionsziels 41

2.9 Kontexte und Scopes 42
2.9.1 Request Scope 43
2.9.2 Session Scope 44
2.9.3 Application Scope 44
2.9.4 Conversation Scope 45
2.9.5 Bean Proxies 46
2.9.6 Dependent Scope 46
2.9.7 Qualifier @New 47
2.9.8 Transaction Scope 47
2.10 Interceptors 47
2.10.1 Interceptor Class 48
2.10.2 Interceptor Binding 49
2.10.3 Aktivierung eines Interceptors 50
2.10.4 Transaktions-Interceptor 51
2.11 Decorators 52
2.11.1 Decorator Class 52
2.11.2 Aktivierung eines Decorators 54
2.12 Stereotypes 54
2.13 Eventverarbeitung 56
2.13.1 Events erzeugen 56
2.13.2 Events verarbeiten 58
2.14 Programmgesteuerter Zugriff auf CDI Beans 61
2.14.1 Injektion von Bean-Instanzen 61
2.14.2 Bean Manager 62
2.15 Integration von JPA, EJB und JSF 63
2.16 Portable Extensions 64
2.16.1 Entwicklung eigener Extensions 64
2.16.2 Verfügbare Extensions 66
2.17 CDI in SE-Umgebungen 68

3 Java Persistence 71

3.1 Worum geht's? 71
3.1.1 Lösungsansätze 72
3.1.2 Anforderungen an O/R-Mapper 73
3.1.3 Entwicklung des Standards 74
3.1.4 Architektur von Anwendungen auf Basis von JPA 75

3.2 Die Basics 76
3.2.1 Entity-Klassen 76
3.2.2 Konfiguration der Persistence Unit 78
3.2.3 CRUD 80
3.2.4 Detached Objects 82
3.2.5 Entity-Lebenszyklus 83
3.2.6 Mapping-Annotationen für einfache Objekte 84
3.2.7 Custom Converter 90
3.2.8 Generierte IDs 91
3.2.9 Objektgleichheit 94
3.2.10 Basisklassen für Entity-übergreifende Aspekte 97
3.3 Objektrelationen 99
3.3.1 Unidirektionale n:1-Relationen 99
3.3.2 Unidirektionale 1:n-Relationen 102
3.3.3 Bidirektionale 1:n-Relationen 104
3.3.4 Uni- und bidirektionale 1:1-Relationen 107
3.3.5 Uni- und bidirektionale n:m-Relationen 109
3.3.6 Eager und Lazy Loading 110
3.3.7 Entity Graphs 112
3.3.8 Kaskadieren 114
3.3.9 Orphan Removal 116
3.3.10 Anordnung von Relationselementen 117
3.4 Queries 118
3.4.1 JPQL 118
3.4.2 Native Queries 131
3.4.3 Criteria Queries 134
3.5 Vererbungsbeziehungen 142
3.5.1 Mapping-Strategie SINGLE_TABLE 143
3.5.2 Mapping-Strategie TABLE_PER_CLASS 145
3.5.3 Mapping-Strategie JOINED 146
3.5.4 Non-Entity-Basisklassen 146
3.5.5 Polymorphe Queries 147
3.6 Dies und das 148
3.6.1 Secondary Tables 148
3.6.2 Zusammengesetzte IDs 149
3.6.3 Dependent IDs 151

3.6.4 Locking 153
3.6.5 Callback-Methoden und Listener 157
3.6.6 Bulk Update/Delete 159
3.7 Caching 160
3.8 Erweiterte Entity Manager 164
3.8.1 Extended Entity Manager 164
3.8.2 Application Managed Entity Manager 165
3.9 Java Persistence in SE-Anwendungen 169
3.9.1 Konfiguration der Persistence Unit im SE-Umfeld 170
3.9.2 Erzeugung eines Entity Managers in SE-Anwendungen 171
3.9.3 Transaktionssteuerung in Java-SE-Anwendungen 172
3.9.4 Schema-Generierung 172

4 BeanValidation 175

4.1 Aufgabenstellung 175
4.2 Plattformen und benötigte Bibliotheken 176
4.3 Validation Constraints 177
4.3.1 Attribute Constraints 177
4.3.2 Method Constraints 178
4.3.3 Vordefinierte Constraints 178
4.3.4 Transitive Gültigkeit 179
4.3.5 Constraint Composition 180
4.3.6 Constraint Programming 181
4.4 Objektprüfung 184
4.5 Internationalisierung der Validierungsmeldungen 185
4.6 Validierungsgruppen 186
4.7 Integration in JPA, CDI und JSF 187
4.8 Bean Validation in SE-Umgebungen 189

5 JavaServer Faces 191

5.1 Einsatzzweck von JSF 191
5.2 Die Basis: Java-Webanwendungen 191
5.2.1 Grundlegender Aufbau 191
5.2.2 Servlets 192
5.2.3 JavaServer Pages 194

5.3 JSF im Überblick 195
5.3.1 Model View Controller 195
5.3.2 Facelets 196
5.3.3 Request-Verarbeitung 197
5.4 Konfiguration der Webanwendung 199
5.5 Benötigte Bibliotheken und Plattformen 201
5.6 Programmierung der Views 201
5.6.1 JSF Tag Libraries 202
5.7 Managed Beans 208
5.8 Unified Expression Language 210
5.8.1 Methodenbindung 211
5.8.2 Wertebindung 211
5.8.3 Vordefinierte Variablen 213
5.8.4 Arithmetische Ausdrücke 214
5.9 Navigation 215
5.9.1 Regelbasierte Navigation 215
5.9.2 Inline-Navigation 216
5.9.3 Programmgesteuerte Navigation 217
5.10 Scopes 217
5.11 Verarbeitung tabellarischer Daten 218
5.12 Internationalisierung 221
5.12.1 Locale 221
5.12.2 Resource Bundles 222
5.12.3 Programmgesteuerter Zugriff auf Texte 223
5.13 Ressourcenverwaltung 224
5.13.1 Internationalisierung von Ressourcen 225
5.14 GET Support 225
5.14.1 Verarbeitung von GET-Request-Parametern 226
5.14.2 Erzeugung von GET-Requests 226
5.15 Eventverarbeitung 227
5.15.1 Faces Events 227
5.15.2 Phase Events 228
5.15.3 System Events 229

5.16 Konvertierung 230
5.16.1 Vordefinierte Konverter 231
5.16.2 Custom Converter 232
5.16.3 Ausgabe von Converter- oder Validierungsmeldungen 233
5.17 Validierung 234
5.17.1 Validierung von Eingabewerten 234
5.17.2 Feldübergreifende Validierung 235
5.18 Immediate-Komponenten 242
5.18.1 immediate für Eingabekomponenten 242
5.18.2 immediate für Aktionskomponenten 242
5.19 AJAX 242
5.19.1 AJAX für Aktionselemente 243
5.19.2 AJAX Events 244
5.19.3 AJAX Callbacks 245
5.19.4 JavaScript API 246
5.20 Templating mit Facelets 246
5.20.1 Template 247
5.20.2 Template Client 248
5.20.3 Mehrstufige Templates 249
5.20.4 Mehrere Templates pro Seite 249
5.21 Eigene JSF-Komponenten 250
5.21.1 Composite Components 251
5.21.2 Composite Components mit Backing Beans 255
5.22 Faces Flows 257
5.22.1 Einfache, konventionsbasierte Flows 257
5.22.2 Deskriptorbasierte Flows 258
5.22.3 Producer-basierte Flows 259
5.22.4 Flow Scope 261
5.22.5 Extern definierte Flows 261
5.23 Resource Library Contracts 262
5.24 Komponentenbibliotheken 262
5.25 Security 263
5.25.1 Log-in-Konfiguration 263
5.25.2 Security-Rollen 264
5.25.3 Zugriffsregeln 265

6 Enterprise JavaBeans 267

6.1 Aufgabenstellung 267
6.2 Aufbau von Enterprise JavaBeans 267
6.2.1 EJB-Typen 268
6.2.2 EJB Lifecycle 270
6.3 EJB Deployment 270
6.4 Lokaler Zugriff auf Session Beans 272
6.4.1 Local Interface 272
6.4.2 No-Interface View 273
6.5 Remote-Zugriff 273
6.5.1 Remote Interface 274
6.5.2 Eintrag von EJBs im Namensdienst des Servers 275
6.5.3 Remote Lookup und clientseitige Nutzung von EJBs 276
6.6 Transaktionssteuerung 277
6.6.1 Transaction Management und Transaction Attribute 278
6.6.2 Application und System Exceptions 279
6.6.3 @Transactional vs. EJB Transactions 280
6.7 Asynchrone Methoden 280
6.8 Timer 282
6.9 Security 284
6.9.1 Deklarative Security 284
6.9.2 Programmgestützte Security 285

7 Ein „Real World"-Projekt 287

7.1 Aufgabenstellung 287
7.2 Anwendungsarchitektur 289
7.3 Persistenz 291
7.4 Views 301
7.5 Fachliche Injektion 305

Stichwortverzeichnis 307

Vorwort

Java steht uns als leistungsfähige Basis für die Softwareentwicklung schon seit mehr als siebzehn Jahren zur Verfügung, einen großen Teil davon auch inklusive der Enterprise Edition, d. h. der Plattform für Unternehmensanwendungen, die verteilte Systeme mit oder ohne Browser als Client, (Web) Services, Clustering, integrierte Systemlandschaften etc. beherrschbar machen soll. Warum dann jetzt ein Buch über die Softwareentwicklung mit Java EE?

Java EE ist zweifellos schon seit vielen Jahren eine verlässliche und tragfähige Plattform zur Entwicklung von Enterprise-Anwendungen. Die hohe Komplexität der ersten Versionen hat aber viele bewogen, sich ganz oder teilweise zugunsten anderer Frameworks abzuwenden. Die Kritik war bis zur Version 1.4 auch durchaus berechtigt – nun aber sind die Weichen neu gestellt in Richtung Einfachheit der Softwareentwicklung.

Ein Wert der Java EE besteht sicher in ihrer abwärtskompatiblen Weiterentwicklung. Für ältere Versionen erstellte Anwendungen bleiben also weiter lauffähig. Als Kehrseite der Medaille sind somit auch die alten Strukturen der Plattform noch vorhanden. Um in den Genuss der leichtgewichtigen Softwareentwicklung zu kommen, muss man sich somit auf die neuen Wege konzentrieren und alten Ballast rechts und links liegen lassen.

Dieses Buch soll Ihnen bei der Auswahl aus dem großen Angebot der Java EE eine Hilfe sein. Es hat nicht den Anspruch einer allumfassenden Darstellung, sondern beschränkt sich auf die Teile der Gesamtspezifikation, mit denen sich ein leistungsfähiger, aber überschaubarer Stack für Enterprise-Anwendungen zusammensetzen lässt. Es zeigt anhand vieler Beispiele, wie einfach Software für die Java-EE-Plattform erstellt werden kann. In einem durchgängigen „Real World"-Projekt werden am Ende alle behandelten Teile zu einer kompletten Anwendung zusammengesetzt, so wird das Zusammenspiel der Einzelteile verdeutlicht.

Begleitprojekte

Die im Buch gezeigten Beispiele stammen aus den Begleitprojekten zu den jeweiligen Buchkapiteln. Sie stehen unter *http://www.gedoplan.de/veroeffentlichungen/javaee7buch* zum Download bereit. Dort ist auch das erwähnte übergreifende Projekt enthalten.

Die Projekte sind als Maven-3-Projekte aufgebaut, um möglichst unabhängig von einem bestimmten Betriebssystem oder einer konkreten IDE zu sein. Im Ordner *readme* finden Sie Hinweise zum Build der Projekte, zur Einrichtung der Laufzeitumgebung und zum Import in Eclipse.

Die Implementierungen der verschiedenen Provider für CDI, JPA etc. waren zum Zeitpunkt der Drucklegung des Buches teilweise noch unvollständig und fehlerbehaftet im Hinblick auf die neuen Features der Java EE 7. Dies ist sicher zu erklären mit der relativ kurzen Zeit seit dem Release der Spezifikation. Die Hinweise im Ordner *readme* enthalten auch eine Liste der aufgetretenen Bugs. Aktualisierungen dazu, aber auch allgemeine Informationen finden Sie ebenfalls in unserem Java-EE-Blog: *http://javaeeblog.wordpress.com*.

Weiterführende Dokumentation

Im Sinne der Auswahl der zur leichtgewichtigen Softwareentwicklung benötigten Teile der Java EE geht das Buch nicht auf sämtliche Details der Plattformbestandteile ein, sondern verweist für weniger häufig genutzte Informationen häufig auf die jeweiligen Spezifikationen. Sie können über *http://jcp.org* auf diese Texte zugreifen. Am Anfang der Buchkapitel finden Sie jeweils einen entsprechenden Link.

Wir haben weitgehend darauf verzichtet, die Java-Dokumentation der Plattformklassen im Buch abzudrucken, da sie online verfügbar ist. Sie kann unter *http://docs.oracle.com/javaee/7/api/* direkt gelesen werden und steht auch unter *http://www.oracle.com/technetwork/java/javaee/documentation/* zum Download bereit.

5 JavaServer Faces

5.1 Einsatzzweck von JSF

Der überwiegende Teil von Enterprise-Anwendungen ist mit einem grafischen Benutzerinterface ausgestattet, mit dem Anwendungsdaten visualisiert, Eingaben durch den Benutzer gemacht und Aktionen ausgelöst werden können. Häufig wird zu diesem Zweck ein Webbrowser verwendet, d. h. die Präsentation ist bspw. HTML-basiert und die Kommunikation zwischen Browser und Anwendung geschieht mittels HTTP.

Der Vorteil dieser Konstellation liegt u. a. darin, dass der Anwender keinerlei Softwareinstallation über die Bereitstellung eines Webbrowsers hinaus durchführen muss. Im Gegenzug muss die Anwendung die Aufbereitung des zur Präsentation genutzten HTML-Texts übernehmen und die vom Browser ausgelösten HTTP-Requests verarbeiten.

Technisch lässt sich diese Anforderung mit Webanwendungen und den darin enthaltenen Servlets realisieren. Der nächste Abschnitt geht auf diese Anwendungsbasis im Überblick ein. Der Aufbau einer kompletten Anwendung ausschließlich mit Servlets wäre allerdings sehr aufwändig. Hier wird eher eine Anwendungsschicht mit einem höheren Abstraktionsgrad gebraucht, die in geeigneter Weise die Präsentation in HTML mit Daten und Methoden von Java-Objekten verbindet und eine Verknüpfung der Views der Anwendung untereinander ermöglicht. Der Java-EE-Standard bietet dazu JavaServer Faces – kurz JSF – an. Anwendungen lassen sich damit aus HTML-ähnlichen Seitenbeschreibungen aufbauen, die Daten und Logik von Java-Objekten verwenden und referenzieren.

5.2 Die Basis: Java-Webanwendungen

5.2.1 Grundlegender Aufbau

Webanwendungen bestehen zunächst aus einer Menge von Dokumenten, die im Browser zur Anzeige gebracht werden sollen. Es können HTML-Dokumente sein oder auch Grafiken, PDF-Dateien etc. Sie befinden sich im Startverzeichnis der Anwendung oder in passend benannten Unterverzeichnissen.

Der statische Teil der Webanwendung wird ergänzt um kompilierte Java-Klassen und Ressource-Dateien. Sie finden im Verzeichnis *WEB-INF/classes* Platz oder im Fall von Bibliotheken in *WEB-INF/lib*. Schließlich wird die Anwendung mit einem Deployment De-

scriptor namens *web.xml* und ggf. weiteren Parameterdateien im Verzeichnis *WEB-INF* konfiguriert.

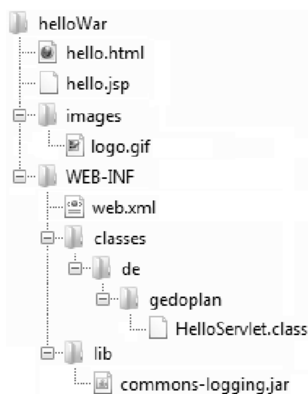

Abbildung 5.1: Aufbau einer Webanwendung

Ein Einstiegsbeispiel zeigt Abb. 5.1: Im willkürlich benannten Startverzeichnis *helloWar* und dem Unterverzeichnis *images* befinden sich einige statische Dokumente. *WEB-INF* enthält den Deployment Descriptor, eine kompilierte Klasse sowie eine Bibliothek.

Webanwendungen werden zum Deployment auf einem geeigneten Server mit *jar* gepackt und erhalten dabei die Dateiendung *.war*. Das Deployment-Verfahren ist abhängig vom Server. In vielen Fällen reicht es, eine Kopie der *war*-Datei in ein dafür bestimmtes Verzeichnis zu speichern.

Der Name der Deployment-Datei ohne ihre Endung bestimmt den sog. Web Context, der Teil der URL zur Adressierung der Anwendungsteile wird. Würde obiges Beispiel als Datei *helloWar.war* auf einem lokalen GlassFish- oder JBoss-Server deployt, müsste man im Browser die Adresse *http://localhost:8080/helloWar/hello.html* nutzen, um die entsprechende Webseite angezeigt zu bekommen.

5.2.2 Servlets

Die gezeigte Struktur wird allerdings erst dann zu einer richtigen Anwendung, wenn sie dynamische Anteile enthält, d. h. Java-Klassen, deren Methoden während der Request-Verarbeitung aufgerufen werden, sodass zu diesem Zeitpunkt dynamischer Inhalt erzeugt und im Browser angezeigt werden kann. Diese Aufgabe übernehmen Servlets. Sie werden im Folgenden skizziert, allerdings ohne auf Details einzugehen. Diese können in der Servlet-Spezifikation[1] nachgelesen werden.

1 Java Servlet Specification, Version 3.1, Rajiv Mordani, December 2009, *http://jcp.org/en/jsr/detail?id=340* → Final Release Download → servlet-3_1-final.pdf

Unter Servlets versteht man Klassen, die i. d. R. von *HttpServlet*[2] abgeleitet werden. Ihre Methode *doGet* wird zur Verarbeitung eines HTTP-GET-Requests aufgerufen. Ähnliche Methoden sind auch für die anderen HTTP-Verben (POST etc.) vorgesehen. Die beiden Parameter vom Typ *HttpServletRequest*[3] und *HttpServletResponse*[4] stellen Ein- und Ausgabe des Servlets dar: Die jeweilige Methode verarbeitet die Request-Parameter und füllt das Response-Objekt u. a. mit dem Text, den der Browser schließlich anzeigen soll (Listing 5.1).

```
public class HelloServlet extends HttpServlet
{
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
  {
    resp.setContentType("text/html");   // Response ist HTML
    PrintWriter out = resp.getWriter();
    out.println("<html>");             // Anzeigetext ausgeben
    out.println("<body>");
    out.println("  Hallo, Welt!");
    out.println("</body>");
    out.println("</html>");
    out.close();
  }
}
```

Listing 5.1: Servlet[5]

Eines fehlt allerdings noch: Die Klasse muss als Servlet registriert und mit einem URL verknüpft werden. Dies ist mithilfe der Annotation *@WebServlet* möglich (Listing 5.2) oder alternativ durch Einträge im Descriptor *web.xml* (Listing 5.3).

```
@WebServlet(urlPatterns = "/helloServlet")
public class HelloServlet extends HttpServlet
{
  …
```

Listing 5.2: Servlet-Registrierung per Annotation

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
  xmlns=http://xmlns.jcp.org/xml/ns/javaee
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
```

2 *javax.servlet.http.HttpServlet*
3 *javax.servlet.http.HttpServletRequest*
4 *javax.servlet.http.HttpServletResponse*
5 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-jsf*

```
  <servlet>
    <servlet-name>helloServlet</servlet-name>
    <servlet-class>de.gedoplan.….HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>helloServlet</servlet-name>
    <url-pattern>/helloServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing 5.3: Servlet-Registrierung im Descriptor „web.xml“

Ein Servlet kann wie im Listing gezeigt mit einem exakten Pfad in der Webanwendung verknüpft werden: Das URL-Pattern */helloServlet* bewirkt, dass das Servlet in unserer Beispielanwendung unter der URL *http://localhost:8080/helloWar/helloServlet* aufrufbar ist. Alternativ sind Verknüpfungen mit Verzeichnissen (*/somePath/**) oder Endungen (**.xhtml*) möglich. Einem Servlet können mehrere URL-Patterns zugeordnet werden.

5.2.3 JavaServer Pages

Es stellte sich in der Vergangenheit schnell heraus, dass Servlets zwar ein mächtiges Werkzeug für die dynamische Request-Verarbeitung sind, die Gestaltung von Webseiten damit aber recht mühsam ist. Schließlich muss der gesamte HTML-Text mithilfe von Ausgabeanweisungen im Servlet erstellt werden.

Abhilfe verspricht die Umkehrung der Vorgehensweise: Statt im Java-Code HTML-Text auszugeben, gestaltet man eine HTML-Seite mit speziellen Tags, die Java-Code enthalten. Der Java-EE-Standard bietet dazu JavaServer Pages – kurz JSP – an. Sie werden in Webanwendungen wie statische Dokumente integriert, allerdings mit der Dateiendung *.jsp* statt *.html*. Zur Laufzeit der Anwendung werden JSPs spätestens bei der ersten Benutzung in äquivalente Servlets übersetzt.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h2>"Hallo, Welt!"</h2>
    <br/>
    Dies ist eine JSP. Es ist <%= new java.util.Date() %>
  </body>
</html>
```

Listing 5.4: JavaServer Page

Wird das in Listing 5.4 gezeigte Beispiel als *hello.jsp* in unsere Beispielanwendung integriert, so führt der URL *http://localhost:8080/helloWar/hello.jsp* zur Anzeige einer Seite im Browser, in der im Text der Zeitpunkt des Requests eingeflochten ist. Dies geschieht durch

den im Tag <%= *new java.util.Date()* %> enthaltenen Java-Code. JSP kennt diverse weitere Tags zur Integration von Deklarationen und Anweisungen, die für die weitere Betrachtung aber unerheblich sind. Bei Interesse finden Sie die Details in der JSP-Spezifikation[6].

JSPs verfolgen zwar durch die beschriebene Umkehr der Vorgehensweise den richtigen Ansatz, bescheren dem Entwickler aber neue Probleme: Zum einen wird der in JSPs enthaltene Java-Code zur Laufzeit der Anwendung kompiliert, Übersetzungsfehler treten somit erst dann auf. Zudem beziehen sich die Fehlermeldungen nicht direkt auf das JSP-Dokument, sondern auf einen vom Server temporär daraus erstellten Java-Quelltext. Zum anderen ist die Gefahr sehr groß, durch die Mischung von HTML- und Java-Code unübersichtliche Programme zu schreiben. Viele reale Projekte sahen sich dadurch am Ende mit praktisch unwartbarem Code konfrontiert.

Eine Lösung dieser Problematik besteht darin, Java-Code nicht mehr direkt in die Webdokumente zu integrieren, sondern ihn in Bibliotheken auszulagern und mit speziellen Tags zu referenzieren. Somit entsteht auf der Seite der Webdokumente eine erweiterte, HTML-ähnliche Sprache, was sich mithilfe von XML und Namespaces gut und ohne konzeptionellen Bruch realisieren lässt. Die andere Seite des referenzierten Java-Codes ist nun in normalen Bibliotheken enthalten, die mit herkömmlichen Mitteln im normalen Build-System erstellt werden können. Eine Ausprägung dieser Vorgehensweise sind die weiter unten beschriebenen Facelets, die in JavaServer Faces als präferierte View-Beschreibung eingesetzt werden.

5.3 JSF im Überblick

5.3.1 Model View Controller

Model View Controller oder auch kurz MVC ist ein Architekturmuster für die Softwareentwicklung. Es trennt drei Aspekte der Software und weist ihnen klare Gestaltungsarten zu:

- Das Model enthält die bearbeiteten Daten. Die Geschäftslogik ist hier mit enthalten oder wird von hier aufgerufen.
- Die View dient der Darstellung der Daten. Sie übernimmt auch die Interaktion mit dem Benutzer, also insbesondere die Annahme von Benutzeraktionen wie Eingaben und Kommandos, ist aber nicht für die eigentliche Verarbeitung zuständig.
- Der Controller verwaltet Views und Models und verknüpft dabei insbesondere die von den Views gelieferten Benutzerkommandos mit den Daten und der Logik der Models.

Das Ziel von MVC ist eine höhere Flexibilität für Änderungen und Erweiterungen und eine größere Wiederverwendbarkeit der einzelnen Komponenten. Das Konzept wurde am Ende der 1970er Jahre zunächst für Benutzeroberflächen in Smalltalk beschrieben (Abb. 5.2).

6 JavaServer Pages, Version 2.1, Pierre Delisle, Jan Luehe, Mark Roth, May 2006, *http://jcp.org/en/jsr/detail?id=245* → Final Release Download → jsp-2_1-fr-spec.pdf

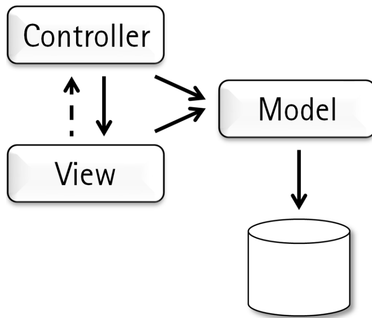

Abbildung 5.2: Model View Controller

Im Fall von JavaServer Faces kommt MVC in der folgenden Ausprägung zum Einsatz: Der Controller wird durch das Faces Servlet implementiert. Die Views sind Facelets, d. h. XHTML-Dokumente, die einige besondere Tag-Bibliotheken verwenden. In JSF können auch andere View-Technologien zum Einsatz kommen, worauf aber hier nicht weiter eingegangen werden soll. Die Models schließlich werden als POJOs (Plain Old Java Objects) bereitgestellt, d. h. als einfache Java-Objekte.

Alle Anwendungs-Requests werden vom Faces Servlet behandelt. Aufgrund des Status der Anwendung wird entschieden, welche Beans und Views verwendet werden sollen. Die Antwort wird schließlich von einer View erzeugt und im Browser angezeigt (Abb. 5.3).

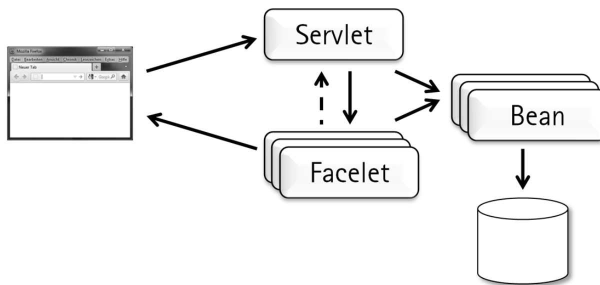

Abbildung 5.3: MVC in JSF

5.3.2 Facelets

Die Sprache zur Seitenbeschreibung ist in der JSF-Spezifikation[7] nicht festgelegt. Jede Implementierung muss aber zumindest Facelets und JSP unterstützen. Seit JSF 2.0 werden Facelets präferiert. Das sind XHTML-Dokumente entsprechend der Definition des W3C.

7 JavaServer Faces Specification, Version 2.2, Oracle Inc., March 2013, *http://jcp.org/en/jsr/detail?id=344* → Final Release Download → javax.faces-api-2.2-FINAL.zip

Sie können um Tags aus weiteren Namensräumen ergänzt werden, um die spezielle Funktionalität von Facelets zu nutzen:

- *http://xmlns.jcp.org/jsf/html*
  UI-Komponenten (Ein/Ausgabeelemente, Aktionselemente, …)
- *http://xmlns.jcp.org/jsf/core*
  Strukturelemente, Parameterelemente; modifizieren das Verhalten von anderen Tags, die sie enthalten oder umschließen
- *http://xmlns.jcp.org/jsf/facelets*
  Tags zur Definition und Nutzung von Templates
- *http://xmlns.jcp.org/jsp/jstl/core und http://xmlns.jcp.org/jsp/jstl/functions*
  Facelet-Version der JSTL-Tags (JSP Standard Tag Library)
- *http://xmlns.jcp.org/jsf/composite*
  Tags zur Definition von eigenen Komponenten

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<head>
  <title>First Facelet</title>
</head>
<body>
  <h:outputText value="Hello World!" />
  <br/>
  <h:outputText value="Ihr Browser: #{header['User-Agent']}" />
</body>
</html>
```

Listing 5.5: Einfaches Facelet

Ein Einstiegsbeispiel für ein Facelet zeigt Listing 5.5. Es benutzt das Tag *<h:outputText>* aus der HTML-Bibliothek zur Ausgabe von Text. Der Ausdruck *#{…}* stammt aus der JSF Expression Language – kurz JSF-EL – und wird zur Request-Zeit ausgewertet. Die weiteren Abschnitte dieses Kapitels gehen genauer auf die verfügbaren Tags und die JSF-EL ein.

5.3.3 Request-Verarbeitung

Wie oben dargestellt, werden Anwendungs-Requests vom zentralen Faces Servlet angenommen. Die Verarbeitung geschieht dann im sog. Request Processing Lifecycle. Dieser Lebenszyklus umfasst sechs Phasen von der Initialisierung der Verarbeitung über die Annahme von Request-Parametern bis schließlich zum Rendern der Antwort (Abb. 5.4).

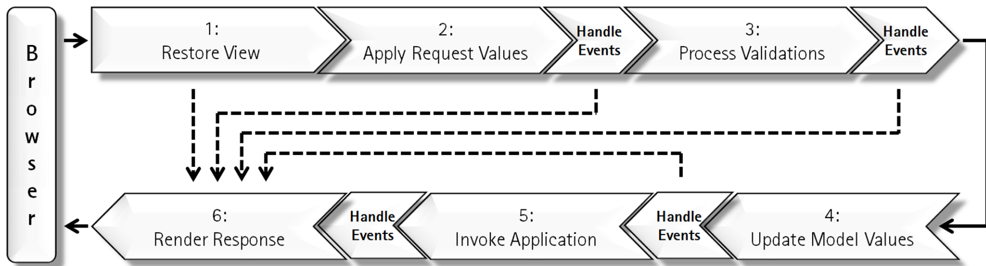

Abbildung 5.4: Request Processing Lifecycle

An die Phasen schließt sich meist eine Event-Verarbeitung an. Hierin kann man mit verschiedenen Event Listenern auf Zustandswechsel o. ä. reagieren.

Beim Auftreten von Verarbeitungsfehlern (Validierungsfehler etc.) wird die Request-Verarbeitung direkt mit der letzten Phase zur Anzeige der View abgeschlossen. Dies kann in der Event-Verarbeitung auch programmatisch ausgelöst werden.

Während der Event-Verarbeitung kann zudem auf das Rendern einer Antwort komplett verzichtet werden (in der Abbildung nicht dargestellt). Dies ist bspw. der Fall, wenn binäre Daten als Antwort an den Client gesendet werden, z. B. Bilder oder PDF-Dokumente.

In der Phase 1 (Restore View) wird die interne Darstellung der aktuellen View hergestellt. Darunter versteht man eine baumartige Objektstruktur, die den ineinander verschachtelten Komponenten der View entspricht. Wird eine View zum ersten Mal benutzt, existiert dieser interne Komponentenbaum noch nicht. Dann wird er entsprechend der Struktur der Seite aufgebaut. Die restlichen Verarbeitungsschritte bis auf die Rendering-Phase entfallen dann. Bei Folgeaufrufen wird der Komponentenbaum aus dem gespeicherten Status heraus wieder hergestellt und der komplette Lebenszyklus durchlaufen.

Der Komponentenbaum einer View wird zwischen zwei Requests gespeichert. Die Ablage geschieht i. d. R. serverseitig in der Session, kann aber auch clientseitig mithilfe von versteckten Feldern in den Views geschehen.

Einige Komponenten lassen die Eingabe von Werten zu (Texte, Listenauswahl, …). Die Eingabewerte werden im Request als Parameter mitgeliefert. Die Phase 2 (Apply Request Values) übernimmt die Eingabewerte als sog. Submitted Values in die zugehörigen Objekte des Komponentenbaums. Die alten Werte der Komponenten werden nicht verändert, um einen späteren Vergleich noch zu ermöglichen.

In der Phase 3 (Process Validations) werden die übermittelten Werte in den gewünschten Zieldatentyp konvertiert und auf Gültigkeit geprüft. Einige Komponenten haben implizite Konverter und Validatoren, weitere können bei Bedarf registriert werden. Verlaufen Konvertierung und Validierung positiv, werden die Eingabewerte endgültig in den Komponenten abgespeichert. Ergibt sich dabei eine Werteänderung, wird ein entsprechender

Event ausgelöst. Bei Konvertierungs- oder Validierungsfehlern werden die Phasen 4 und 5 übersprungen und die Request-Bearbeitung wird mit der letzten Phase fortgesetzt.

In den bisherigen Phasen waren nur die Objekte des Komponentenbaums beteiligt. In der Phase 4 (Update Model Values) werden nun die Werte von dort in die assoziierten Model-Objekte kopiert. Durch die bisherige Verarbeitung ist sichergestellt, dass die Werte valide sind.

Einige Komponenten haben einen natürlichen Aktionscharakter, z. B. Buttons oder Links. Ihnen werden normalerweise Methoden zugeordnet, die die entsprechende Anwendungslogik enthalten. Sie werden in der Phase 5 (Invoke Application) aufgerufen, wenn der Request von einem solchen Aktionselement ausgelöst wurde. Anschließend findet die Navigation in der Anwendung statt, d. h. die Festlegung der als Nächstes anzuzeigenden View.

Als Abschluss der Request-Bearbeitung wird in der Phase 6 (Render Response) der anzuzeigende Inhalt erzeugt. In dieser Phase wird zudem der Komponentenbaum für weitere Anfragen der gleichen View abgespeichert.

Der dargestellt Lebenszyklus kann über die Attribute der beteiligten Komponenten noch partiell modifiziert werden. Darauf gehen die nachfolgenden Abschnitte 5.18 (Immediate-Komponenten) und 5.19 (A) noch ein.

5.4 Konfiguration der Webanwendung

Eine JSF-Anwendung unterscheidet sich im Aufbau nicht von einer herkömmlichen Webanwendung, wie sie oben in Abb. 5.1 dargestellt wurde. In ihrem Deployment Descriptor *WEB-INF/web.xml* wird das Faces Servlet registriert und alle Request-URLs zugeordnet, die auf *.xhtml* enden (Listing 5.6).

```
<web-app version="3.1"
  xmlns=http://xmlns.jcp.org/xml/ns/javaee
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>

  <context-param>
```

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

Listing 5.6: Grundkonfiguration der Webanwendung zur Nutzung von JSF

Das Faces Servlet wird durch die Angabe *<load-on-startup>1</load-on-startup>* schon beim Start der Anwendung geladen, was die jeweilige Implementierung zur Initialsierung des Systems nutzen kann. Die Verknüpfung mit der URL-Endung *.xhtml* ist üblich, aber nicht zwingend. Häufig findet man auch die Endungen *.faces* oder *.jsf* vor.

Mit *<context-param>*-Elementen kann die Anwendung weiter konfiguriert werden. Die Angaben sind optional, meist existieren gute Default-Werte. Das Listing zeigt beispielhaft den Parameter *javax.faces.PROJECT_STAGE*, mit dem der Entwicklungsstand der Anwendung deklariert werden kann. Die möglichen Werte sind *Development*, *UnitTest*, *SystemTest* und *Production*. Die gewählte Einstellung führt dazu, dass umfangreichere Meldungen im Fehlerfall angezeigt werden, was während der Entwicklung recht hilfreich ist. Einige praxisrelevante Parameter sind in Tabelle 5.1 zusammengefasst. Weitere Informationen finden Sie in der JSF-Spezifikation (Abschnitt 11.1.3, Application Configuration Parameters).

Parameter javax.faces.…	Bedeutung	Default
DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE	Für die Konvertierung von Date-Werten nicht UTC, sondern die Systemzeitzone verwenden	*false*
FACELETS_SKIP_COMMENTS	XML-Kommentare in den Views nicht zum Client senden	*false*
INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL	Leere Eingabetexte als null weiterverarbeiten	*false*
PROJECT_STAGE	Entwicklungsstand der Anwendung	*Production*

Tabelle 5.1: JSF-Konfigurationsparameter

Zusätzlich zu dieser allgemeinen Konfiguration des JSF-Systems in der Webanwendung kann die Datei *WEB-INF/faces-config.xml* genutzt werden, um das Verhalten der JSF-Anwendung einzustellen. So finden dort bspw. die später beschriebenen Navigationsregeln oder Internationalisierungsparameter ihren Platz. Als Ausgangspunkt kann die in Listing 5.7 gezeigte quasi-leere Datei dienen. Seit JSF 2.0 ist sie sogar optional, kann also im ersten Schritt auch komplett entfallen.

```
<faces-config version="2.2"
  xmlns=http://xmlns.jcp.org/xml/ns/javaee
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
```

```
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
</faces-config>
```

Listing 5.7: Quasi-leere Konfigurationasdatei „faces-config.xml"

5.5 Benötigte Bibliotheken und Plattformen

Zur Entwicklung von JSF-Anwendungen wird die JSF-Standardbibliothek benötigt. Sie kann z. B. als Maven Dependency im Projekt eingebunden werden (Listing 5.8). Die Bibliothek ist im Zielserver bereits vorhanden, sodass der Scope *provided* ausreicht, wodurch die Bibliothek nicht in die Webanwendung integriert wird.

```
<dependency>
  <groupId>javax.faces</groupId>
  <artifactId>javax.faces-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>
```

Listing 5.8: Maven Dependency für das JSF-API

Alternativ kann auch die umfassende Dependency *javax:javaee-api:7.0* genutzt werden. Als Plattform kann jeder Java-EE-7-Server dienen, wobei das Web Profile ausreichend ist, also bspw. GlassFish 4 oder WildFly 8. Die darin eingesetzte JSF-Implementierung ist zumeist die Referenzimplementierung JSF-RI oder Apache MyFaces. Bei einigen Servern sind sogar mehrere Implementierungen enthalten. Für die Entwicklung von JSF-Anwendungen spielt dies aber keine Rolle – soweit die Implementierung keine Fehler enthält.

5.6 Programmierung der Views

Die JSF-Views werden in einer geeigneten Template-Sprache beschrieben. Alle Implementierungen müssen hier JSP und Facelet unterstützen. In diesem Buch werden ausschließlich Facelets verwendet. Sie bilden seit JSF 2.0 den Standard und bedürfen keiner weiteren Konfiguration.

Die Seiten werden als Dateien mit der Endung *.xhtml* programmiert, wobei die im Folgenden beschriebenen JSF-Tags eingesetzt werden können.

Der Aufruf geschieht allerdings nicht direkt, sondern über einen URL mit der Endung, auf die das Faces Servlet gemappt ist, also bspw. **.xhtml*. Selbst wenn diese mit der Endung der Seitendateien übereinstimmt, werden sie nicht direkt ausgeliefert. Vielmehr wird immer das Faces Servlet aufgerufen, was zum Rendern der Ergebnisseite die entsprechende Seitendatei benutzt.

5.6.1 JSF Tag Libraries

Die Views benötigen i. d. R. zwei grundlegende Tag-Bibliotheken, die über die weiter oben angeführten Namensräume referenziert werden: Die HTML-Bibliothek enthält Komponenten wie Ein- und Ausgabefelder, Buttons etc., aus denen die Benutzeroberfläche zusammengestellt wird. Ihr Namensraum *http://xmlns.jcp.org/jsf/html*[8] wird meist mit dem Präfix *h* in die Views eingebunden. Die Tags der Core-Bibliothek werden nicht direkt zur Anzeige gebracht. Vielmehr dienen sie der Gruppierung oder Parametrierung von anderen Elementen, die sie umschließen oder enthalten. Der zugehörige Namensraum *http://xmlns.jcp.org/jsf/core* wird üblicherweise mit dem Präfix *f* in die Seiten integriert. Ein Facelet hat somit grundsätzlich den in Listing 5.9 gezeigten Aufbau.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://xmlns.jcp.org/jsf/html"
  xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
  <title>…</title>
</h:head>
<h:body>
  …


</h:body>
</html>
```

Listing 5.9: Grundsätzlicher Aufbau eines Facelets

HTML-Bibliothek

Die HTML-Bibliothek enthält Tags für UI-Komponenten wie Textausgabe oder diverse Varianten von Eingabe- und Aktionselementen. In der Rendering-Phase werden sie durch die entsprechenden HTML-Elemente dargestellt. Facelets dürfen darüber hinaus beliebigen HTML-Text enthalten, wodurch sich automatisch Überschneidungen mit der HTML-Bibliothek ergeben. In einem solchen Fall muss dem jeweiligen Tag aus der Bibliothek der Vorzug gegeben, also bspw. *<h:head>* und *<h:body>* genutzt werden und nicht *<head>* bzw. *<body>*. Einige Tags werden nämlich nicht nur als entsprechendes HTML-Element gerendert, sondern führen bspw. zur Integration von Stylesheets oder Skripten in die Ausgabe.

Die HTML-Bibliothek enthält u. a. einige Tags zur Ausgabe von Texten, Links und Grafiken (Tabelle 5.2).

8 Ältere Versionen nutzen die Namespaces *http://java.sun.com/...* Sie sind auch mit JSF 2.2 noch nutzbar.

Tag	Beschreibung	Beispiel
h:outputText	Textausgabe	*<h:outputText value="Hallo"/>*
h:outputFormat	Ausgabe eines parametrisierten Texts	*<h:outputFormat value="Sehr geehrte{0} {1}">* *<f:param value="…"/>* *<f:param value="…"/>* *</h:outputFormat>*
h:outputLabel	Ausgabe eines Labels	*<h:outputLabel for="price" value="Preis"/>* *<h:inputText id="price" …/>*
h:outputLink	Ausgabe eines Hyperlinks	*<h:outputLink value="http://www.gedoplan.de"/>*
h:graphicImage	Ausgabe einer Grafik	*<h:graphicImage url="/images/logo.gif"/>*

Tabelle 5.2: Ausgabe-Tags in der HTML-Bibliothek

Sie funktionieren alle in ähnlicher Weise: Ihr Attribut *value* (*url* bei *graficImage*) bestimmt, was sie zur Anzeige bringen. Mit weiteren Parametern kann das Verhalten weiter gesteuert werden. Als Referenz für die Tags kann die – allerdings nicht sehr übersichtliche – Dokumentation der JSF-Referenzimplementierung genutzt werden: *http://javaserverfaces.java.net/nonav/docs/2.0/pdldocs/facelets/index.html*.

Alle Tags akzeptieren das Attribut *id* zum Setzen der Komponenten-ID sowie den Boole'schen Parameter *rendered*. Wenn er *false* ist, wird die Anzeige der Komponente unterdrückt.

```
<html …>
<h:head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
  <title>Ausgabe-Tags h:outputXxx</title>
</h:head>
<h:body>
  <h:outputText value="Einfacher Text" />
  <hr />
  <h:outputFormat value="Sehr geehrte{0} {1}" rendered="false">
    <f:param value=" Frau" />
    <f:param value="Mustermann" />
  </h:outputFormat>
  <hr />
  <h:outputLabel for="farbe" value="Farbe:" />
  <h:outputText id="farbe" value="rot" />
  <hr />
  <h:outputLink value="http://www.gedoplan.de">
    <h:graphicImage url="/resources/logos/gedoplan-logo.gif" />
  </h:outputLink>
</h:body>
</html>
```

Listing 5.10: Demonstration diverser Ausgabe-Tags

Die Nutzung der Ausgabe-Tags ist in Listing 5.10 beispielhaft gezeigt. Die angezeigten Werte sind allerdings sämtlich statisch, was natürlich in praktischen Anwendungen nur begrenzt sinnvoll ist. Wir werden später sehen, dass an die Stelle fester Werte Referenzen zu Java-Objekten treten können.

Die HTML-Bibliothek beherbergt auch zwei Tags, mit denen andere Elemente gruppiert werden können (Tabelle 5.3): *h:panelGroup* packt seine Unterelemente zu einem neuen Element zusammen. Die HTML-Ausgabe für den Browser enthält dann abhängig vom Attribut *layout* ein *span*- oder *div*-Element. *h:panelGrid* erzeugt eine Tabelle mit der angegebenen Spaltenanzahl. Die Unterelemente füllen die Spalten in der gegebenen Reihenfolge auf, wodurch sich implizit eine Zeilenanzahl für die Tabelle ergibt.

Tag	Beschreibung	Beispiel
h:panelGroup	Gruppierung von mehreren Elementen zu einem	*<h:panelGroup> <h:outputText value="Hallo, "/> <h:outputText value="Welt"/> </h:panelGroup>*
h:panelGrid	Gruppierung und Gitternetz-anordnung	*<h:panelGrid columns="2"> <h:outputLabel value="Bezeichnung"/> <h:outputText value="DVD-Rohling"/> <h:outputLabel value="Preis"/> <h:outputText value="0,60"/> </h:panelGrid>*

Tabelle 5.3: Gruppierungs-Tags in der HTML-Bibliothek

Zu diesen Gruppierungselementen oder sogar zur HTML-Bibliothek allgemein muss man allerdings einwerfen, dass die Gestaltungsmöglichkeiten und der Umfang in der Standardbibliothek eher gering sind. Wenn Sie Ihre Anwendung attraktiv gestalten wollen, müssen Sie in der Praxis Zusatzbibliotheken wie RichFaces oder PrimeFaces einsetzen.

Weitere Bestandteile der HTML-Bibliothek betreffen Eingabe- und Aktionselemente (Tabelle 5.4 und Tabelle 5.5). Sie sind nur innerhalb eines *h:form*-Elements nutzbar. Die Eingabeelemente haben alle eine ähnliche Grundfunktion: Sie zeigen Werte in einer Form an und übermitteln sie wieder beim nächsten Request, wenn dieser durch ein Aktionselement in der Form ausgelöst wurde. Die Nutzung dieser Elemente wird weiter unten in den Abschnitten 5.7 Managed Beans und 5.8 Unified Expression Languages in einigen Beispielen erläutert.

Tag	Beschreibung	Beispiel
h:inputText *h:inputTextarea* *h:inputSecret* *h:inputHidden*	Eingabefeld standard mehrzeilig geheim versteckt	*<h:inputText value="#{bean.valueProperty}"/>*
h:selectBooleanCheckbox	Checkbox	*<h:selectBooleanCheckbox value="#{bean.valueProperty}" />*
h:selectOneListbox *h:selectOneMenu* *h:selectOneRadio*	Einfachauswahl Listbox Drop-down Radioboxes	*<h:selectOneListbox value="#{bean.valueProperty}" >* *<f:selectItems value="#{bean.listProperty}" />* *</h:selectOneListbox>*
h:selectManyListbox *h:selectManyMenu* *h:selectManyCheckbox*	Mehrfachauswahl Listbox Drop-down Checkboxes	*<h:selectManyListbox value="#{bean.valueProperty}" >* *<f:selectItems value="#{bean.listProperty}" />* *</h:selectManyListbox>*

Tabelle 5.4: Eingabe-Tags in der HTML-Bibliothek

Tag	Beschreibung	Beispiel
h:commandButton *h:commandLink*	Button Link	*<h:commandButton value="text" action="#{bean.method}" />*

Tabelle 5.5: Aktionselemente in der HTML-Bibliothek

Der Vollständigkeit halber zeigt Tabelle 5.6 die restlichen Bestandteile der HTML-Bibliothek. Sie wurden entweder schon erwähnt (*h:head*, *h:body*, *h:form*) oder haben Spezialaufgaben, die erst später in diesem Kapitel thematisiert werden können.

Tag	Beschreibung	Beispiel
h:head *h:body*	HTML-Rahmen- elemente	*<html >* *<h:head>* … *</h:head>* *<h:body>* … *</h:body>* *</html>*

Tag	Beschreibung	Beispiel
h:form	Formular	*<h:form>* *<h:inputText …/>* *<h:commandButton …" />* *</h:form>*
h:dataTable *h:column*	Tabellarische Datendarstellung	*<h:dataTable var="item"…>* *<h:column>* *<h:outputText value="#{item.name}"/>* *</h:column>* *…* *</h:dataTable>*
h:message *h:messages*	Meldungsausgabe	*<h:inputText id="name" …/>* *<h:message for="name"/>* *…* *<h:messages/>*
h:button *h:link*	Erzeugung von GET-Requests	*<h:button value="…" outcome="…">*
h:outputScript	Script-Ausgabe	*<h:outputScript name="…" library="…"* *target="head"/>*
h:outputStylesheet	Stylesheet-Ausgabe	*<h:outputStylesheet name="…" library="…"/>*

Tabelle 5.6: Weitere Elemente in der HTML-Bibliothek

Core-Bibliothek

Die Tags der Core-Bibliothek kommen nicht unmittelbar zur Anzeige, sondern fügen den UI-Komponenten Metadaten, Parameter oder Funktionalität hinzu. Ein Beispiel ist in Tabelle 5.2 bereits zu sehen: *f:param* übergibt dort Werte, die die Platzhalter des Texts im Tag *h:outputMessage* füllen.

Ein zentrales Tag ist *f:view*. Es ist der Container für die anderen JSF-Tags, muss also den kompletten Seiteninhalt umschließen – einmal abgesehen vom *html*-Tag. In Facelets ist es allerdings optional, d. h. es wird implizit an die richtige Stelle platziert, wenn eine View es nicht explizit enthält. Insofern können Sie dieses Tag auch gleich wieder vergessen…[9]

Ein interessanteres Tag ist *f:facet*. Es kann als Unterelement diverser *h*-Tags eingesetzt werden, um ihnen bestimmte Aspekte hinzuzufügen. Jeder Aspekt ist benannt mit einem vom Einsatzzweck abhängigen Namen. So kann man bspw. einer mit *h:panelGrid* erstellten Tabelle die Aspekte *header* und *footer* mitgeben, um eine Tabellenüberschrift bzw. Fußnote zu erzeugen. Die Aspektinhalte können einfache Texte oder auch beliebige Kombinationen von JSF-Tags sein.

9 Mit JSF 2.2 wurden sog. Stateless Views eingeführt, die den Zustand ihrer Komponenten in Phase 6 nicht speichern (d. h. in Phase 1 muss der Komponentenbaum stets neu erstellt werden). Dieses in der Praxis selten genutzte Feature kann so aktiviert werden: *<f:view transient="true"> … </f:view>*

```
<h:panelGrid …>
  <f:facet name="header">
    <h:outputText value="#{bean.headerText}" />
  </f:facet>
  <f:facet name="footer">
    Footer-Text
  </f:facet>
…
```

Listing 5.11: Aspekte für Header und Footer

Tabelle 5.7 fasst die Tags der Core-Bibliothek zusammen. Ihr Einsatz wird später an den entsprechenden Stellen erläutert.

Tag	Bedeutung	s. Abschnitt
f:actionListener, f:event, f:phaseListener, f:setPropertyActionListener, f:valueChangeListener	Registrierung von Listenern für diverse Events	5.15
f:convertDateTime, f:convertNumber, f:converter	Konvertierer angeben	5.16
f:attribute	Attribute zu umgebendem Tag hinzufügen	
f:ajax	AJAX-Funktionalität hinzufügen	5.19
f:facet	Aspekt hinzufügen	5.6.1, 5.11
f:metadata, f:viewParam	Metadaten und Parameter für die View definieren	5.14
f:loadBundle	Resource Bundle laden (besser: Resource Bundle global deklarieren)	5.12
f:param	Parameter hinzufügen	
f:selectItem, f:selectItems	Selektionswerte bestimmen	5.8
f:subview	Neuen ID-Namensraum definieren	
f:validateDoubleRange, f:validateLength, f:validateLongRange, f:validateBean, f:validateRegex, f:validateRequired, f:validator	Validatoren angeben (besser: Bean Validation einsetzen)	5.17
f:verbatim	HTML- und JSP-Text zu JSF-Komponente zusammenfassen (wird in Facelets nicht benötigt)	
f:view	View definieren (wird in Facelets nicht benötigt)	5.6.1

Tabelle 5.7: Tags der Core-Bibliothek

5.7 Managed Beans

Die bisherigen Einstiegsbeispiele waren eher statischer Natur. Um dort Dynamik hineinzubringen, benötigen wir Daten und Logik, die wir über sog. Managed Beans mit den Views verknüpfen. Den Managed Beans fällt somit eine entscheidende Rolle für die klare Trennung von Präsentation und Geschäftslogik zu. Man kann für gut strukturierte Software sogar noch weiter gehen und Seitenbeschreibung, Präsentationslogik und Geschäftslogik voneinander trennen:

- Die Seitenbeschreibung deklariert den Seitenaufbau, komponiert eine View also aus ihren Einzelkomponenten. Dazu stehen uns Facelets zur Verfügung.
- Die Geschäftslogik umfasst Abläufe, Persistenz, Berechnungen – eben alles, was eine Anwendung unabhängig von ihrer konkreten Oberfläche tut. Hierfür stehen uns bspw. CDI und Java Persistence zur Verfügung.
- Managed Beans im Sinne von JSF verknüpfen diese beiden Anwendungsseiten miteinander und implementieren die ggf. noch fehlende Präsentationslogik, d. h. sie bereiten Daten aus der Geschäftslogik für die Präsentationskomponenten auf oder entscheiden aufgrund von Aufrufen der Geschäftslogik über die Navigation innerhalb der Seiten der Anwendung.

Zur Bereitstellung von Managed Beans gibt es durch die Historie der Java-EE-Plattform bedingt mehrere Möglichkeiten: Zum einen hat JSF eigene Managed Beans, die mithilfe der Annotation *@ManagedBean*[10] oder gleichwertiger Einträge im Konfigurations-File *faces-config.xml* definiert werden. Zum anderen können CDI Beans mit der Annotation *@Named* diese Rolle übernehmen. Die Überschneidung ist nahezu 100 %ig. Die erste Variante wird in einer zukünftigen JSF-Version entfallen. Im Folgenden werden daher ausschließlich CDI Beans verwendet.

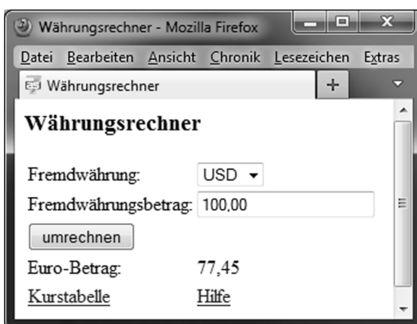

Abbildung 5.5: Währungsrechner

Angenommen, wir wollten den in Abb. 5.5 gezeigten Dialog als Webanwendung realisieren. Die HTML-Bibliothek bietet uns dazu einige Eingabekomponenten an (s. Tabelle 5.4),

10 *javax.faces.bean.ManagedBean*

von denen wir hier zwei verwenden können: *h:selectOneMenu* für die Währungsauswahl und *h:inputText* für die Eingabe des Betrags.

Eingabeelemente erhalten ihren aktuellen Wert während der Rendering-Phase, indem eine Property einer Managed Bean gelesen wird. Umgekehrt wird der eingegebene Wert während der Phase 4 (*Update Model Values*) in die Property gespeichert. Der Begriff Property entstammt der JavaBeans-Spezifikation und bezeichnet ein Methodenpaar: Eine Getter-Methode zum Lesen und eine Setter-Methode zum Setzen des Werts. Diese Methoden haben die vorgegebenen Namen *getName* und *setName*, wobei der Name der Property *name* ist. Unsere Währungsrechner-View wird später auf die Property *fremdWaehrungsBetrag* zugreifen, d. h. in Phase 4 wird *setFremdWaehrungsBetrag* aufgerufen und in Phase 6 *getFremdWaehrungsBetrag*.

Analog benötigen wir für die Währungsauswahl und den Euro-Betrag die Properties *fremdWaehrungsKuerzel* und *euroBetrag*. Im letzten Fall können wir auf die Setter-Methode verzichten, da der Betrag nur ausgegeben wird.

Für den Button – eines der Aktionselemente aus der HTML-Bibliothek – gilt eine ähnliche Überlegung: Hier benötigen wir eine Methode, die in der Phase 5 (Invoke Application) aufgerufen wird, wenn der Button betätigt wurde. Für die Managed Bean des Währungsrechners ergibt sich somit die in Listing 5.12 gezeigte Klasse.

```
@Named @SessionScoped
public class WaehrungsRechnerModel implements Serializable
{
  private String fremdWaehrungsKuerzel;
  private double fremdWaehrungsBetrag;
  private double euroBetrag;

  public String getFremdWaehrungsKuerzel()
  {
    return this.fremdWaehrungsKuerzel;
  }

  public void setFremdWaehrungsKuerzel(String fremdWaehrungsKuerzel)
  {
    this.fremdWaehrungsKuerzel = fremdWaehrungsKuerzel;
  }

  // weitere Getter und Setter
  // …

  public void umrechnen()
  {
    …
  }
…
```

Listing 5.12: Managed Bean mit Properties und Aktionsmethode

Einen Fallstrick stellen die Scope-Annotationen dar – wir haben sie im Kapitel über CDI bereits kennen gelernt. Leider gibt es im Paket *javax.faces.bean* gleichnamige Annotationen. Achten Sie daher bei Verwendung von CDI Beans darauf, dass sie die CDI-Annotationen aus *javax.enterprise.context* verwenden.

Die Seitenbeschreibung für den Eingabeteil zeigt Listing 5.1. Nur angedeutet sind darin allerdings die Verknüpfungen zwischen View und Managed Bean, die mithilfe der Unified Expression Language realisiert werden.

```
<html …>
…
<h:body>
  …
  <h:form>
    <h:panelGrid columns="2">
      <h:outputLabel for="fremdWaehrung" value="Fremdwährung: " />
      <h:selectOneMenu id="fremdWaehrung" value="#{…}">
        <f:selectItems value="#{…}" …"/>
      </h:selectOneMenu>

      <h:outputLabel for="fremdBetrag" value="Fremdwährungsbetrag: " />
      <h:inputText id="fremdBetrag" value="#{…}"/>

      <h:commandButton value="umrechnen" action="#{…}" />
      <h:panelGroup />

      <h:outputLabel for="euroBetrag" value="Euro-Betrag: " />
      <h:outputText id="euroBetrag" value="#{…}"/>
…
```

Listing 5.13: Eingabeteil der Währungsrechner-View

5.8 Unified Expression Language

Die JSF Expression Language ist das Bindeglied zwischen der textbasierten Seitenbeschreibung und der objektorientierten Welt der Managed Beans. Mit ihr ist es möglich, die Werte von UI-Komponenten mit Properties von Managed Beans zu verknüpfen oder auch Aktionselemente mit den Methoden zu verbinden, die bei ihrer Betätigung aufgerufen werden sollen.

Die grundlegende Syntax von JSF-EL-Ausrücken ist *#{expression}*. Als *expression* können darin eine Wertebindung, eine Methodenbindung oder eine arithmetischer Ausdruck stehen.

Wenn hier von JSF-EL gesprochen wird, ist das eigentlich nicht mehr ganz richtig: Früher definierten sowohl die JSP-Spezifikation als auch JSF getrennte Expression Languages. JSP-EL-Ausdrücke folgen dabei dem Format *${expression}*, sind also syntaktisch bis auf das Startzeichen identisch mit JSF-EL-Ausdrücken. Der Unterschied lag darin, dass JSP-

EL-Ausdrücke an jeder Stelle einer JSP genutzt werden konnten und schon beim Seitenaufbau ausgewertet wurden. JSF-EL-Ausdrücke sind dagegen nur innerhalb der JSF-Tags erlaubt und werden in den verschiedenen Phasen der Request-Verarbeitung ausgewertet.

Mittlerweile (seit JSP 2.1 und JSF 1.1) hat man diese beiden Sprachen aber vereinigt zur Unified Expression Language. Das einleitende Zeichen – *#* oder *$* – kann nun frei gewählt werden und die Platzierung innerhalb der Seitenbeschreibung bestimmt den Auswertezeitpunkt.

5.8.1 Methodenbindung

Um eine Aktionskomponente wie bspw. einen Button mit der nach Betätigung auszuführenden Methode zu verknüpfen, bedient man sich eines EL-Ausdrucks zur Methodenbindung mit der Syntax *#{bean.method}*. *bean* verweist darin auf eine Managed Bean mit dem entsprechenden Bean-Namen. Im CDI-Kapitel wurde beschrieben, wie einer CDI Bean mithilfe der Annotation *@Named* ein Name zugewiesen wird.

method bezeichnet die Methode innerhalb der Bean. Sie muss die Signatur *public void method()* oder *public Object method()* haben. Es dürfen auch Parameter übergeben werden. Der EL-Ausdruck ist dann *#{bean.method(parameter)}*, und die Methode muss eine passende Parameterliste aufweisen.

Um den Button des Währungsrechners mit der Methode *WaehrungsRechnerModel.umrechnen* zu verbinden, muss das Attribut *action* des Buttons den Wert *#{waehrungsRechnerModel.umrechnen}* haben (Listing 5.14). Betätigt der Benutzer den Button zur Laufzeit, wird damit einerseits ein Request ausgelöst. Durch die Methodenbindung kommt es dann in Phase 5 zum Aufruf der gewünschten Methode.

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

Listing 5.14: Bindung des Buttons an die aufzurufende Bean-Methode

Mit dem Attribut *actionListener* und den Unterelementen *f:actionListener* lassen sich weitere Methoden registrieren, die ebenfalls aufgrund der Betätigung des Buttons aufgerufen werden. Darauf geht der Abschnitt über die Eventbehandlung weiter unten ein.

5.8.2 Wertebindung

Möchte man eine UI-Komponente mit einem Wert in einer Managed Bean verbinden, kommt ein EL-Ausdruck in Form einer Wertebindung zum Einsatz. Er folgt der allgemeinen Syntax *#{bean.property}*. Darin verweist *bean* auf eine Managed Bean mit dem entsprechenden Bean-Namen, *property* ist der Name einer Property innerhalb der referenzierten Bean.

Im Währungsrechner werden in dieser Art die beiden Währungsfelder mit den zugehörigen Properties aus *WaehrungsRechnerModel* verknüpft (Listing 5.15). Die Wertebindung ist bidirektional, d. h. sie wirkt sowohl lesend als auch schreibend: In Phase 6 wird die

Getter-Methode der Property aufgerufen, um den Wert zu lesen. Bei einem Eingabeelement wird die Setter-Methode in Phase 4 aufgerufen, um den aktuellen Wert in die Bean Property zu speichern.

```
<h:inputText value="#{waehrungsRechnerModel.fremdWaehrungsBetrag}"/>
…
<h:outputText value="#{waehrungsRechnerModel.euroBetrag}">
```

Listing 5.15: Wertebindung für Ein- und Ausgabefelder

Das Auswahlelement für die Währung wird analog behandelt, allerdings wird hier noch eine Liste der zur Verfügung stehenden Werte benötigt. Dazu dient das Element *f:selectItems* im Body des Auswahlelements. Sein Attribut *value* bestimmt die Menge der Auswahlwerte. Über eine Wertebindung werden hier ein Array oder eine *Collection* von Objekten zugeordnet. Die entsprechende Property wird nur gelesen, d. h. es reicht aus, die Getter-Methode in der Bean bereitzustellen.

Die Auswahlelemente *h:selectXxx* unterscheiden zwischen dem *label*, das dem Benutzer angezeigt wird, und dem *value*, der durch die Auswahl schließlich eingegeben wird. Je nach Typ der mittels *f:selectItems* angelieferten Werte bestimmen sich *Label* und *Value* in unterschiedlicher Weise:

- Bei einem Array oder einer *Collection* von Werten des Typs *SelectItem*[11] bestimmen deren Attribute *label* und *value*, was angezeigt und eingegeben wird. *SelectItem* bietet passende Konstruktoren und Zugriffsmethoden zum Setzen der Attribute an. Listing 5.16 zeigt eine beispielhafte Property, in der eine solche Auswahlliste mit numerischen Eingabewerten und Beschriftungstexten erstellt wird.

```
public List<SelectItem> getNotenListe()
{
  List<SelectItem> notenList = new ArrayList<>();
  notenList.add(new SelectItem(1, "sehr gut"));
  notenList.add(new SelectItem(2, "gut"));
  notenList.add(new SelectItem(3, "befriedigend"));
  notenList.add(new SelectItem(4, "unbefriedigend"));
  return notenList;
}
```

Listing 5.16: Mithilfe von „SelectItem“-Objekten aufgebaute Auswahlliste

- Bei einem Array oder einer *Collection* von anderen Java-Objekten entsprechen Anzeige- und Eingabewerte der String-Repräsentation der Objekte (d. h. *toString()*). *f:selectItems* kann allerdings mit weiteren Parametern versehen werden, um *label* und *value* explizit zu bestimmen. Dazu wird mit *var* ein nur innerhalb des Tags gültiger Variablenname deklariert und mit *itemLabel* und *itemValue* jeweils eine Bindung an eine Property in Be-

11 *javax.faces.model.SelectItem*

zug auf diese Variable angegeben. Die Variable wirkt ähnlich einer Schleifenvariablen, mit der bei jedem Schleifendurchlauf ein *label* und *value* für die Auswahl ermittelt wird.

Im Währungsrechner steht die Methode *getWaehrungen* zur Verfügung, die alle bekannten Währungen in einer Liste liefert. Deren Elemente vom Typ *Waehrung* haben wiederum Getter-Methoden *getId* und *getEuroValue*, die das Währungskürzel (z. B. *CHF*) und der zugehörigen Eurowert liefern. Damit ist eine Parametrierung des Auswahlelements in der View wie in Listing 5.17 gezeigt möglich. *itemLabel* und *itemValue* werden hier beide aus der Währungs-ID gefüllt. Es wäre aber auch möglich gewesen, *itemValue="#{waehrung.euroValue}"* anzugeben, um in die Ziel-Property der Eingabe den Umrechnungsfaktor anstelle des Währungskürzels zu speichern.

```
<h:selectOneMenu
       value="#{waehrungsRechnerModel.fremdWaehrungsKuerzel}">
  <f:selectItems value="#{waehrungsRechnerModel.waehrungen}"
                var="waehrung"
                itemLabel="#{waehrung.id}"
                itemValue="#{waehrung.id}"/>
</h:selectOneMenu>
```

Listing 5.17: Auswahlelement mit expliziter Angabe der Anzeige- und Eingabewerte

Im Falle einer Property, die nur gelesen wird, kann statt einer Wertebindung auch ein Methodenaufruf im EL-Ausdruck genutzt werden. Im Beispiel oben wäre somit statt *<h:outputText value="#{waehrungsRechnerModel.euroBetrag}">* auch *<h:outputText value="#{waehrungsRechnerModel.getEuroBetrag()}">* möglich gewesen. Diese Variante ermöglicht es, auf Methoden zuzugreifen, die nicht dem Namensschema von Properties folgen, z. B. *value="#{bean.list.size()}"* oder *rendered="#{bean.text.contains('abc')}"*.

5.8.3 Vordefinierte Variablen

Die Unified Expression Language definiert einige Variablen vor (Tabelle 5.7). Sie können in EL-Ausdrücken wie Bean-Namen verwendet werden, werden allerdings nicht häufig benötigt.

Variable	Type	Bedeutung
initParam	*Map*	Initialisierungswerte (Context-Parameter der Webanwendung)
facesContext	*FacesContext*[1]	Statusinformationen der aktuellen Request-Verarbeitung
requestScope *viewScope* *sessionScope* *applicationScope*	*Map<String, Object>*	Scope-Objekte

Variable	Type	Bedeutung
view	*UIViewRoot*[2]	Aktuelle View
header *headerValues*	*Map<String, String>* *Map<String, String[]>*	Header-Werte des Requests als Strings bzw. String[]
param *paramValues*	*Map<String, String>* *Map<String, String[]>*	Parameter-Werte des Requests als Strings bzw. String[]
cookie	*Map<String, Object>*	Cookies des Requests

Tabelle 5.8: Vordefinierte EL-Variablen

5.8.4 Arithmetische Ausdrücke

Für EL-Ausdrücke sind einige Operatoren definiert, mit denen sich arithmetische Ausdrücke kombinieren lassen (Tabelle 5.8).

Bei ihrem Einsatz sollten Sie allerdings Vorsicht walten lassen, da Berechnungen in den allermeisten Fällen Teil der Geschäftslogik sind und somit in einer View fehlplatziert wären. Bedenklich in diesem Sinne sind sämtliche Beispiele der Tabelle mit Ausnahme von *rendered="#{empty bean.list}"* und ggf. auch *rendered="#{bean.number != 0}"*.

Für den Zugriff auf Map-Elemente gibt es noch eine weitere Möglichkeit: *#{(bean.map.key}* ist äquivalent zu *#{(bean.map['key']}*.

<table>
<tr><th>Operator</th><th>Bedeutung</th><th>Beispiel</th></tr>
<tr><td>+, -, *, /, %</td><td>Grundrechenarten</td><td>value="#{bean.number * 1.19}"</td></tr>
<tr><td>==, !=, <, <=, >, >=
eq, ne, lt, le, gt, ge</td><td>Vergleiche</td><td>rendered="#{bean.number != 0}"</td></tr>
<tr><td>empty</td><td>Test auf Leerheit</td><td>rendered="#{empty bean.list}"</td></tr>
<tr><td>?:</td><td>Bedingter Ausdruck</td><td>value="#{(bean.number%2)==0 ? 'even' : 'odd'}"</td></tr>
<tr><td>&&, ||, !
and, or, not</td><td>Logische Operatoren</td><td>rendered="#{bean.number1 != 0
&& bean.number2 > 100}"</td></tr>
<tr><td>()</td><td>Klammerung</td><td>value="#{(bean.number + 1) * 2}"</td></tr>
<tr><td>[]</td><td>Array-, Listen-, Map-Zugriff</td><td>value="#{(bean.array[2]}"
value="#{(bean.list[5]}"
value="#{(bean.map['key']}"</td></tr>
</table>

Tabelle 5.9: EL-Operatoren

5.9 Navigation

Anwendungen besitzen in den meisten Fällen nicht nur eine View. Übergänge zwischen den Anzeigeseiten sind immer dann möglich, wenn ein Request ausgelöst wird, wenn der Benutzer also eines der Aktionselemente bedient.

5.9.1 Regelbasierte Navigation

Ein wichtiger Parameter zur Festlegung der nächsten View ist das sog. Outcome der auslösenden Aktion. Darunter versteht man einen Text, der als Ergebnis der aufgerufenen Aktionsmethode geliefert wird oder auch direkt im Attribut *action* des Aktionselements angegeben wird (Listing 5.18). Das Outcome ist implizit *null*, wenn die Aktionsmethode kein Ergebnis liefert (*void*-Methode) bzw. das Attribut *action* fehlt.

```
<h:form>
  <h:commandButton value="ok" action="#{someBean.doOk}" />
  <h:commandLink value="Hilfe" action="help" />
</h:form>

@Model
public class SomeBean
{
  public String doOk()
  {
    return "goOn";
  }
  …
```

Listing 5.18: Outcome als Aktionsmethodenergebnis oder direkte Angabe

Vor der Rendering-Phase wird festgelegt, welche View als Nächstes anzuzeigen ist. Dazu werden bei der regelbasierten Navigation Regeln der Form „Falls auf Seite *x* das Outcome *a* erzeugt wird, geht es auf Seite *y* weiter“ verwendet. Diese befinden sich in der Konfigurationsdatei *WEB-INF/faces-config.xml* in *navigation-rule*-Elementen (Listing 5.19).

```
<faces-config … >
  <navigation-rule>
    <from-view-id>/pages/waehrungsRechner*</from-view-id>
    <navigation-case>
      <from-outcome>help</from-outcome>
      <to-view-id>/pages/waehrungsRechner_help.xhtml</to-view-id>
    </navigation-case>
    …
```

Listing 5.19: Navigation Rule

Jede View der Anwendung besitzt eine View-ID, die dem Pfad der Seitenbeschreibung innerhalb der Anwendung entspricht.

Jedes dieser Elemente fasst die Regeln für eine oder mehrere Ausgangsseiten zusammen, wobei das Element *from-view-id* bestimmt, für welche. Hier kann entweder eine exakte View ID angegeben werden – z. B. */pages/waehrungsRechner.xhtml* – oder wie im Beispiel ein Präfix der gewünschten View IDs, gefolgt von einem *. Ein *navigation*-rule ohne *from-view-id* oder mit *<from-view-id>*</from-view-id>* passt auf alle Views der Anwendung.

Die Regel kann beliebig viele *navigation-case*-Elemente enthalten, die jeweils ein Outcome mit einer Zielseite verknüpfen. Das Beispiel in Listing 5.18 liest sich also so: Falls auf einer der Währungsrechnerseiten das Outcome *help* erzeugt wird, ist die nächste View */pages/waehrungsRechner_help.xhtml*.

Ein *navigation-case* kann durch weitere Elemente ergänzt werden:

- *from-action*: Angabe einer Aktionsmethode in Form einer EL-Methodenbindung. Die Regel gilt dann nur, wenn das Outcome von der genannten Methode erzeugt wurde. Bei Angabe von *from-action* kann *from-outcome* sogar entfallen, wenn die Regel für alle Outcomes der genannten Methode gelten soll.
- *if*: Angabe einer Bedingung in Form eines Boole'schen EL-Ausdrucks. Die Regel gilt dann nur, wenn der Audruck *true* ergibt.
- *redirect*: Wird dieses Element (ohne Inhalt) angegeben, geschieht der Übergang zur nächsten Seite durch Redirect, d. h. der Browser wird angewiesen, die nächste Seite durch einen neuen Request anzufordern. Ohne *redirect* wird ein serverseitiges Forward durchgeführt, wodurch kein neuer Request benötigt wird, der im Browser angezeigte URL allerdings um einen Zyklus nachhinkt.

Durch die Möglichkeit, die Ausgangs-View mit einem Präfix bestimmen und in den *navigation-case*s unterschiedliche Bedingungen angeben zu können, kann es zu Mehrdeutigkeiten kommen, d. h. auf eine Ausgangssituation passen ggf. mehrere Regeln. In der JSF-Spezifikation ist exakt beschrieben, welche Regel dann angewendet wird. Grob kann man das so zusammenfassen: Die Regel mit der genauesten Angabe gewinnt.

Gibt es keine passende Regel, wird die aktuelle Seite erneut angezeigt.

5.9.2 Inline-Navigation

Es ist möglich, auf die Navigationsregeln in *faces-config.xml* zu verzichten. Dann müssen anstelle von Outcomes direkt View IDs verwendet werden – entweder im Attribut *action* der Aktionselemente oder als Ergebnis der Aktionsmethoden (Listing 5.20). Mit dem Zusatz *?faces-redirect=true* kann wiederum ein Redirect angefordert werden.

```
<h:commandButton value="tue was" action="#{someBean.doSomething}" />
<h:commandLink value="Kurstabelle"
               action="/pages/waehrungsRechner_rates.xhtml" />

@Model
public class SomeBean
```

```
{
  public String doSomething()
  {
    return "/somePage.xhtml?faces-redirect=true";
  }
  …
```

Listing 5.20: Inline-Navigation

Inline-Navigation führt schnell zu unübersichtlichem und schlecht wartbarem „Spaghetticode“. Setzen Sie sie daher nur mit Bedacht ein!

5.9.3 Programmgesteuerte Navigation

Manchmal ist es wünschenswert, in einer Methode der Anwendung eine Navigation auszulösen, z. B. um in einer der weiter unten gezeigten Listener-Methoden auf besondere Situationen mit einer Umschaltung auf eine neue Seite zu reagieren. Für solche Fälle kann der *NavigationHandler* der JSF-Implementierung direkt aufgerufen werden. Die Methode *handleNavigation* erhält als zweiten und dritten Parameter die Werte, die den oben angesprochenen Elementen *from-action* und *from-outcome* entsprechen (Listing 5.21).

```
if (this.fremdWaehrungsBetrag < 0)
{
  FacesContext facesContext = FacesContext.getCurrentInstance();
  NavigationHandler navigationHandler
    = facesContext.getApplication().getNavigationHandler();
  navigationHandler.handleNavigation(facesContext, null, "help");
}
```

Listing 5.21: Aufruf des Navigations-Handlers

5.10 Scopes

Die Lebensdauer der Managed Beans wird über ihren Scope festgelegt. Da in diesem Buch CDI Beans als Managed Beans genutzt werden, stehen die im CDI-Kapitel beschriebenen Scopes zur Verfügung. Tabelle 5.9 fasst das dort Gesagte nochmals zusammen.

Scope-Annotation	Bean-Lebensdauer
@RequestScoped	Ein Request
@TransactionScoped	Eine Transaktion
@ConversationScoped	Bei transienter Konversation wie *@RequestScoped*, sonst bis zum Ende der Konversation durch explizite Terminierung oder durch Timeout

Scope-Annotation	Bean-Lebensdauer
@SessionScoped	Vom ersten Zugriff bis zum Ende der Sitzung durch explizite Terminierung oder durch Timeout
@ApplicationScoped	Laufzeit der Anwendung

Tabelle 5.10: Scopes

Da JSF auch eigene Scope-Annotationen im Paket *javax.faces.bean* enthält, müssen Sie darauf achten, diejenigen aus *javax.enterprise.context* zu verwenden.

JSF 2.2 definiert zusätzlich den sog. View Scope. Beans, die mit *@ViewScoped*[12] annotiert sind, bleiben so lange aktiv, bis die aktuelle View ID sich ändert, d. h. bis auf eine neue Page navigiert wird. Eine weitere Neuerung der aktuellen Version sind die später beschriebenen Faces Flows. Mit *@FlowScoped* annotierte Beans bleiben solange aktiv, bis der zugehörige Flow beendet wird.

Es sei nicht verschwiegen, dass die Überdeckung zwischen JSF und CDI in Bezug auf die Scopes nicht vollständig ist: JSF kennt neben den genannten Scopes noch einen Flash Scope. Zudem können neue Scopes auf recht einfache Weise mit *@CustomScoped* definiert werden. Die Lücke wird allerdings durch diverse CDI Extensions geschlossen – bspw. durch die im CDI-Kapitel erwähnten Ergänzungen Apache MyFaces CODI oder Delta-Spike.

5.11 Verarbeitung tabellarischer Daten

Viele Anwendungen verarbeiten Daten, die übersichtlich in Form einer Tabelle dargestellt werden können. Das Tag *h:panelGrid* erzeugt zwar eine HTML-Tabelle, aber deren Dimensionen werden in der Seitenbeschreibung festgelegt und nicht dynamisch entsprechend dem Umfang der anzuzeigenden Daten.

Hier kommt *h:dataTable* zum Einsatz. Dieses Tag verarbeitet ein Array oder eine *Collection*[13] von Objekten zur Anzeige einer entsprechend langen Tabelle. Sein Attribut *var* definiert einen nur innerhalb des Tags gültigen Namen für eine Variable, die wie eine Schleifenvariable arbeitet: *h:dataTable* iteriert über die als *value* übergebenen Werte. Für jeden Eintrag wird der Inhalt des Tags zur Anzeige gebracht, wobei die Variable den aktuellen Wert enthält.

Die Unterelemente *h:column* beschreiben je eine Spalte der Tabelle. Die darin befindlichen Ein- und Ausgabeelemente können die Iterationsvariable in ihren EL-Ausdrücken verwenden, um auf eine Property des jeweils aktuellen Eintrags zuzugreifen.

12 *javax.faces.view.ViewScoped* (Achtung: nicht *javax.faces.bean.ViewScoped*!)
13 Allgemeine Collections sind erst seit JSF 2.2 erlaubt, während vorher nur List unterstützt wurde.

Listing 5.22 zeigt eine beispielhafte Anwendung: Das Element *h:dataTable* iteriert mithilfe der lokalen Variablen *bank* über eine Liste von *Bank*-Objekten. Diese haben u. a. die Properties *blz* und *name*, die in den beiden *h:column*-Elementen ausgegeben werden.

```
<h:dataTable var="bank" value="#{bankModel.searchResult}" >
  <h:column>
    <h:outputText value="#{bank.blz}" />
  </h:column>
  <h:column>
    <h:outputText value="#{bank.name}" />
  </h:column>
  …
</h:dataTable>

@Model
public class BankModel
{
  public List<Bank> getSearchResult() { … }
  …
}

public class Bank
{
  public String getBlz() { … }
  public void setBlz(String blz) { … }

  public String getName() { … }
  public void setName(String name) { … }
  …
}
```

Listing 5.22: Anzeige tabellarischer Daten

h:dataTable erlaubt mit seinen Attributen *rowClasses* und *columnClasses* die Angabe von Stilinformationen für die Zeilen und Spalten der Tabelle. Es kann jeweils eine kommagetrennte Liste von CSS-Stilen angegeben werden, die für die Zeilen bzw. Spalten der Reihe nach verwendet werden. So kann bspw. durch Angabe von zwei verschiedenen Zeilenstilen ein alternierender Zeilenhintergrund erzeugt werden, um das Lesen der Tabelle zu erleichtern (Listing 5.23, Abb. 5.6).

```
<h:dataTable var="bank" value="#{bankModel.searchResult}"
             rowClasses="standardTable_Row1,standardTable_Row2" >
```

Listing 5.23: Angabe alternierender Zeilenstile

BLZ	Name	PLZ	Ort
10000000	Bundesbank	10591	Berlin
10010010	Postbank	10916	Berlin
10010111	SEB	10789	Berlin
10010222	ABN AMRO Bank Ndl Deutschland	10105	Berlin
10010424	Aareal Bank	10666	Berlin
10020000	Berliner Bank Ndl d Landesbank Berlin	10890	Berlin
10020200	BHF-BANK	10117	Berlin
10020400	Parex Bank Berlin	10117	Berlin
10020500	Sozialbank	10178	Berlin
10020890	Bayer Hypo- und Vereinsbank	10896	Berlin
10022200	Bankgesellschaft Berlin	10777	Berlin

Abbildung 5.6: Beispiel für eine Datentabelle

Die ausgegebene Tabelle kann mit *f:facet*-Elementen um Header und Footer ergänzt werden. Dies kann wie bei *h:panelGrid* für die Gesamttabelle geschehen oder innerhalb der *h:column*-Elemente für jede Spalte getrennt (Listing 5.24).

```
<h:dataTable …>
  <f:facet name="footer">Quelle: Deutsche Bundesbank</f:facet>
  <h:column>
    <f:facet name="header">BLZ</f:facet>
    <h:outputText value="#{bank.blz}" />
  </h:column>
…
```

Listing 5.24: Datentabelle mit Header und Footer

Die Inhalte der *h:column*-Elemente können beliebige JSF-Tags sein, inkl. Eingabe- oder Aktionselementen. Damit ist es bspw. problemlos möglich, in die Tabellenzeilen einen Button aufzunehmen, der zu einer Bearbeitung des aktuellen Werts navigiert (Listing 5.25).

```
<h:dataTable var="bank" …>
  <h:column>
    <h:commandButton value="bearbeiten"
                     action="#{bankModel.edit(bank)}" />
    </h:column>
…
</h:dataTable>

@Model
```

```
public class BankModel
{
  public String edit(Bank bank) { … }
  …
}
```

Listing 5.25: Datentabelle mit Aktionselement

5.12 Internationalisierung

Viele Tags lassen sich lokalisieren, d. h. sie lassen sich den Gepflogenheiten einer Sprache und eines Landes anpassen. Das umfasst Formate von Zahlen oder Datumsangaben und natürlich Texte, die von einer internationalisierten Anwendung in mehreren Übersetzungen vorgehalten werden müssen.

5.12.1 Locale

Der virtuelle Schauplatz einer Anwendung wird durch ein Objekt des Typs *Locale*[14] angegeben. Es kann vielen Tags mithilfe des Attributs *locale* mitgegeben werden. Dabei kann ein *Locale*-Objekt oder auch eine übliche *String*-Repräsentation der Form *ll*, *ll_CC* oder *ll_CC_VV* übergeben werden. Darin wird die Sprache mit zwei Kleinbuchstaben *ll* angegeben, optional ein Land mit zwei Großbuchstaben *CC* und ggf. zusätzlich eine Variante *VV* (Tabelle 5.11).

de	Deutsch	**en**	Englisch
de_AT	Deutsch (Österreich)	**en_AU**	Englisch (Australien)
de_CH	Deutsch (Schweiz)	**en_GB**	Englisch (Großbritannien)
de_DE	Deutsch (Deutschland)	**en_IN**	Englisch (Indien)
de_LU	Deutsch (Luxemburg)	**en_US**	Englisch (USA)

Tabelle 5.11: Auszug aus den im Java-Standard verfügbaren Locales

Tags ohne explizite *Locale*-Angabe übernehmen die Lokalisierung von ihren umschließenden Tags. Ist auch dort keine Angabe vorhanden, wird die *Locale* aus dem Request ermittelt. Der Browser übermittelt dazu im Header *Accept-Language* eine Liste von akzeptierten *Locale*s. Welche das sind, lässt sich in der Konfiguration des Browsers einstellen.

Für die Anwendung lässt sich umgekehrt in der Konfigurationsdatei *faces-config.xml* einstellen, welche *Locale*s sie bedienen kann. Das geschieht mit dem Element *locale-config*,

14 *java.util.Locale*

in dem eine Vorgabe-*Locale* eingetragen werden kann und eine Liste aller unterstützter *Locale*s (Listing 5.26).

```
<faces-config …>
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de_CH</supported-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>en</supported-locale>
```

Listing 5.26: Konfiguration der unterstützten „Locales“

Zwischen diesen beiden Angaben wird nach der besten Übereinstimmung gesucht: Die vom Browser übermittelten *Locale*s werden in der angegebenen Reihenfolge mit denen der Anwendung verglichen. Dabei wird bei Bedarf jeweils auch ein Fallback innerhalb der Familie gemacht, d. h. die Wunsch-*Locale en_US* passt zur angebotenen *Locale en*.

5.12.2 Resource Bundles

Internationalisierte Texte werden üblicherweise in Form von Properties zur Verfügung gestellt, d. h. die Anwendung referenziert die Texte über logische Schlüssel, deren zugehörige Werte die anzuzeigenden Texte darstellen. Für jede Sprache wird ein Satz von Properties bereitgestellt, die die gleichen Schlüssel verwenden. Zur Laufzeit wird der zur aktuellen Sprache passende Satz von Properties ausgewählt.

Dieses Verfahren wird von der Java-Standard-Klasse *ResourceBundle* implementiert. Eine Möglichkeit der Ablage der Texte sind Properties-Dateien im Classpath: Eine Datei namens *basename.properties* definiert das Resource Bundle und enthält i. d. R. alle benötigten Texte als Default-Werte. Weitere Dateien namens *basename_locale.properties* enthalten die Übersetzungen für eine bestimmte *Locale*. Abb. 5.7 zeigt eine solche Struktur. Das oberste Verzeichnis liegt im Classpath.

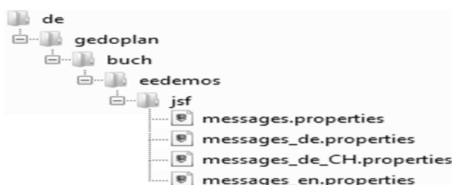

Abbildung 5.7: Properties-Dateien eines Resource Bundles

Die Dateien spannen durch ihre Zuordnung zu Locales eine Art Suchbaum auf. Wird ein Wert in der Datei der aktuellen Locale nicht gefunden, wird die Suche im nächsthöheren Knoten des Baums fortgesetzt, z. B.: *messages_de_CH.properties* → *messages_de.properties* → *messages.properties*.

Resource Bundles können in *faces-config.xml* global deklariert werden. Dabei wird ein Variablenname frei gewählt, der in JSF-EL-Ausdrücken verwendet werden kann, um auf die lokalisierten Texte zuzugreifen (Listing 5.27).

```
<faces-config …>
  <application>
    <resource-bundle>
      <base-name>de.gedoplan.buch.eedemos.jsf.messages</base-name>
      <var>messages</var>
    </resource-bundle>
…
```

Listing 5.27: Deklaration der Variablen „messages" für ein Resource Bundle

Durch diese zentrale Deklaration kann in jeder View der Anwendung mit *#{messages.key}* auf den lokalisierten Text mit dem Schlüssel *key* zugegriffen werden.

Alternativ zur globalen Deklaration kann ein Resource Bundle in einer View explizit geladen werden. Dazu dient das Tag *f:loadBundle*, das bereits vor JSF 2 vorhanden war. Es sollte in neuen Anwendungen allerdings nicht mehr verwendet werden.

Ein besonderes Resource Bundle ist das sog. Application Message Bundle. In ihm befinden sich die Meldungen des JSF-Systems, die bspw. bei Validierungsfehlern generiert werden. Es ist Bestandteil der JSF-Implementierung, kann aber mit einer Deklaration in *faces-config.xml* durch ein eigenes Resource Bundle ersetzt werden (Listing 5.28).

```
<faces-config …>
  <application>
    <message-bundle>
      de.gedoplan.buch.eedemos.jsf.messages
    </message-bundle>
…
```

Listing 5.28: Deklaration eines eigenen Application Message Bundle

5.12.3 Programmgesteuerter Zugriff auf Texte

Manchmal benötigt man innerhalb einer Managed Bean Zugriff auf die lokalisierten Texte. Neben der Standardmöglichkeit mithilfe von *ResourceBundle.getBundle* gibt es seit JSF 2 die in Listing 5.29 gezeigte Möglichkeit, auf ein Resource Bundle über seinen in *faces-config.xml* eingetragenen Variablennamen zuzugreifen.

```
FacesContext ctx = FacesContext.getCurrentInstance();
ResourceBundle bundle
  = ctx.getApplication().getResourceBundle(ctx, "messages");
String text = bundle.getString("key");
```

Listing 5.29: Zugriff auf lokalisierte Texte im Code einer Managed Bean