→ 3., vollständig überarbeitete Auflage

Michael Kurz · Martin Marinschek

JavaServer Faces 2.2

Grundlagen und erweiterte Konzepte

dpunkt.verlag

Michael Kurz studierte Informatik an der Technischen Universität Wien und hat sich seitdem in seiner beruflichen Tätigkeit dem Thema Webentwicklung verschrieben. Seit seinem Wechsel zu IRIAN beschäftigt er sich vorrangig mit JSF, und ist im Unternehmen als Webentwickler für mehrere JSF-Projekte tätig. Weiter leitet er JSF-Schulungen, hält Vorträge auf internationalen Konferenzen und ist Apache MyFaces Committer. Neben der Arbeit als Softwareentwickler schreibt er gerne über JSF und verwandte Themen – unter anderem auf seinem Blog *http://jsflive. wordpress.com*.

Martin Marinschek absolvierte das Studium der Internationalen BWL und der Computertechnik in Wien. Seither leitet er als Geschäftsführer der Firma IRIAN.at verschiedene Software-Engineering-Projekte und bietet Schulungen und Consulting im J2EE-Bereich an. Seit 2003 ist er Mitglied des MyFaces-Projekts und seit 2004 Committer der Apache Software Foundation. Er realisierte eine Vielzahl von großen Webprojekten mit JavaServer Faces, insbesondere mit Apache MyFaces. Gleichzeitig unterrichtet er an Fachhochschulen und Universitäten in Wien Web- und Software-Engineering. Als Mitglied der JSF 2.2 Expertgroup ist Martin Marinschek auch ein Co-Autor der aktuellen JSF-Spezifikation.

Michael Kurz · Martin Marinschek

JavaServer Faces 2.2

Grundlagen und erweiterte Konzepte

3., vollständig überarbeitete Auflage

dpunkt.verlag

Michael Kurz
michael.kurz@irian.at

Martin Marinschek
martin.marinschek@irian.at

Lektorat: Dr. Michael Barabas
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Kurz
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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.

ISBN
Buch 978-3-86490-009-9
PDF 978-3-86491-432-4
ePub 978-3-86491-433-1

3., vollständig überarbeitete Auflage 2014

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Inhaltsverzeichnis

1 Einführung in JavaServer Faces

Vielfalt der Technologien

JavaServer Faces (JSF) ist eine moderne Technologie zur Entwicklung von Webanwendungen. Allerdings steht sie nicht allein auf weiter Flur – es gibt Dutzende als Open Source veröffentlichte und Hunderte proprietäre Frameworks für die Entwicklung von Webapplikationen alleine im Java-Bereich, andere Programmiersprachen außer Acht gelassen. Der Elefant unter diesen Frameworks im Java-Bereich ist sicher Apache Struts, aber auch Apache Wicket und Apache Tapestry sind sehr erfolgreich. Die erste Frage ist also, warum sich solch eine Vielfalt von Frameworks entwickelt hat und weshalb die Notwendigkeit für die Spezifizierung der *JavaServer Faces*-Technologie entstand – immerhin gibt es doch mit der Servlet- und JSP-Technologie schon eine solide Basis für die dynamische Erstellung von Webseiten. Die Beschreibung der geschichtlichen Entwicklung der Webprogrammierung wird hier zum Verständnis beitragen.

1.1 Kurzgeschichte der Webentwicklung

HTML und HTTP

Alles begann mit der Übertragung der ersten Seite in Hypertext Markup Language (HTML) über das Hypertext Transfer Protocol (HTTP) im August 1991. Nur wenige Visionäre ahnten damals, welche Entwicklung das World Wide Web über die Jahre nehmen würde. Exponentielles Wachstum war dem World Wide Web in die Wiege gelegt worden – dies galt für die Verbreitung genauso wie für die technologische Entwicklung. Anfangs war HTML eine einfache Sprache zur Bedeutungsauszeichnung von Textteilen. Durch die vielfache Anwendung in den verschiedensten Bereichen wurde HTML immer mehr hin zur Layoutsprache erweitert. Dem daraus entstandenen Wildwuchs an Auszeichnungselementen für die unterschiedlichsten Zwecke wurde die Layoutsprache CSS (Cascading Style Sheets) entgegengesetzt. Begleitend zu diesen statischen Sprachen wurde auch das dynamische Element durch die Verwendung von JavaScript im Webbrowser immer wichtiger.

Server

Zur selben Zeit war eine ähnliche Revolution der Sprachen und Skriptsprachen am Server im Gange – unzählige serverseitige Techno-

logien kämpften um die Gunst der Webentwickler, Perl, Phyton, PHP, Ruby und natürlich Java sind nur einige Beispiele. Kein Wunder, dass durch diese hohe Anzahl an involvierten Technologien die Entwicklung von großen, hochdynamischen Webanwendungen immer komplexer wurde – die Entwickler mussten bei der Bewältigung dieser Komplexität unterstützt werden.

1.1.1 Geschichte der Webentwicklung mit Java

Servlets

Im Java-Bereich war die Entwicklung der Servlet-Technologie 1997 der erste Schritt zur dynamischen Generierung von HTML-Seiten am Server. Im Wesentlichen beruht diese Technologie darauf, dass in den Java-Code Befehle eingebunden werden, die zur Erzeugung von HTML dienen. Praktisch bedeutet das den Aufruf der Funktion `println` auf einem `OutputStream` wie in Listing 1.1.

Listing 1.1 *Beispiel für ein einfaches Servlet. Hier wird die GET-Methode der HTTP-Anfrage behandelt.*

```
1  public class BeispielServlet extends HttpServlet {
2    public void doGet(HttpServletRequest req,
3                      HttpServletResponse res)
4        throws ServletException, IOException {
5      String name = req.getParameter("name");
6      String text = req.getParameter("text");
7
8      res.setContentType("text/html");
9
10     ServletOutputStream out = res.getOutputStream();
11     out.println("<html><head><title>");
12     out.println(name);
13     out.println("</title></head><body>");
14     out.println(text);
15     out.println("</body></html>");
16     out.flush();
17   }
18 }
```

Sie können sich leicht vorstellen, dass diese Erzeugung von HTML aus normalem Java-Code bei langen HTML-Passagen schwer verständlich und unübersichtlich wird. In einem zweiten Schritt entstanden daher Hilfsklassen, die das Schreiben von HTML-Tags durch den Aufruf gewisser Methoden erleichterten. Die Erstellung von HTML beschränkte sich somit auf den Aufruf dieser Methoden, wie Listing 1.2 zeigt.

JSP

Für komplexe HTML-Seiten war auch diese Vorgehensweise nicht optimal, was zur Entwicklung der *JavaServer Pages*-(JSP-)Technologie führte. Ein Beispiel in dieser Sprache findet sich in Listing 1.3. Hier ist HTML die treibende Kraft und in die einzelnen Tags der HTML

```
public class ServletUtility {
  private HttpServletResponse res;
  public ServletUtility(HttpServletResponse res) {
    this.res = res;
  }
  public void startTag(String tagName) {
    res.print("<");
    res.print(tagName);
  }
  public void endTag(String tagName) {
    res.print("</");
    res.print(tagName);
    res.print(">");
  }
  ...
}
```

Listing 1.2
Beispiel für eine einfache (jedoch unvollständige) Hilfsklasse zum Schreiben von HTML-Code in Servlets

sind in sogenannten Scriptlets die Aufrufe der Java-Methoden zur Ausgabe der dynamischen Teile der HTML-Seite eingebunden. Dieser Ansatz erleichterte die Erstellung von komplexen HTML-Seiten mit viel eingebautem JavaScript-Code und einer hohen Anzahl an CSS-Auszeichnungen ungemein.

```
<%@ page language="java" %>
<html>
  <head>
    <title><%=request.getParameter("name");%></title>
  </head>
  <body>
    <%=request.getParameter("text");%>
  </body>
</html>
```

Listing 1.3
Ein einfaches JSP-Beispiel

Alles, was »benutzt« werden kann, kann allerdings auch »missbraucht« werden, und genau dieser Fall trat für die JSP-Technologie ein. Die Entwickler begannen, immer mehr Code in die einzelnen JSP-Seiten aufzunehmen, bis erneut eine hochkomplexe Mischung aus HTML-Tags und Java-Code entstand. Diese Mischung war genauso schlecht wartbar wie die in Servlet-Code eingebaute HTML-Generierung. Ein weiterer Kritikpunkt an der Verwendung von JSP war, dass der eingebaute Sourcecode erst zum Zeitpunkt des Anwendungsstarts im Applikationsserver kompiliert wurde, und viele Fehler, die normalerweise bei der Erstellung von Java-Klassen aus dem Sourcecode bereits beseitigt worden waren, erst zur Laufzeit auftraten. Die Bedeutung dieses Problems steigt selbst-

verständlich mit der Menge des in die JSP-Seite eingebundenen Sourcecodes.

Webframeworks

Zur Lösung dieses Problems traten Webframeworks auf den Plan. Der Entwickler wird bei Benutzung eines Frameworks dazu angehalten, möglichst große Teile der Layoutbeschreibung in einer Seitendeklarationssprache wie JSP zu erstellen und gleichzeitig möglichst wenig Funktionalität im Sinne von Anwendungslogik zwischen die Elemente der Seitendeklarationssprache einzufügen.

Model-View-Controller (MVC)

Ein klarer Schnitt zwischen den Bereichen Modell, Ansicht und Steuerungslogik ist also notwendig – dieses Entwicklungsmuster wird auch Model-View-Controller-Muster (kurz MVC) genannt und ist in Abbildung 1.1 dargestellt.

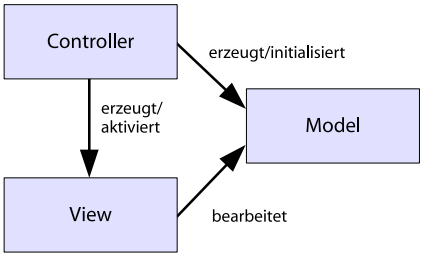

Abbildung 1.1
Das Model-View-Controller-Prinzip

Model2 – MVC für das Web

Bei der Webentwicklung mit Java hat sich eine spezielle Form dieses Entwurfsmusters mit dem Namen *Model2* etabliert. Der Begriff *Model2* stammt aus der Spezifikation des JSP-Standards und beschreibt die Übertragung des MVC-Ansatzes in die Welt der Webentwicklung mit Java. Diese Form ist dem zugrunde liegenden MVC-Muster sehr ähnlich, einzig die verschiedenen Ausprägungen von Model, View und Controller werden hier genauer definiert, wie Abbildung 1.2 zeigt. Für fast alle mit Java arbeitenden Webframeworks dient das Model2-Pattern als Grundlage der Architektur. Als Steuerungslogik (Controller) wird dabei ein Servlet verwendet und meist ist das Modell in Form von Java-Klassen, häufig als Beans oder POJOs (Plain Old Java Objects), ausgeführt. Für die Deklaration der Ansicht gibt es allerdings viele Möglichkeiten – bei Turbine dient hierzu Velocity, bei Cocoon ein XML-Dialekt und bei Struts und JSF vor Version 2.0 *JavaServer Pages* (JSPs). Ab Version 2.0 setzt JSF standardmäßig auf Facelets (XHTML) als Seitendeklarationssprache.

Komponenten

Anfangs stand diese Trennung der einzelnen Schichten einer Applikation als höchste Priorität auf der Aufgabenliste der einzelnen Webframeworks. Alle oben genannten Frameworks haben dieses Problem im Bereich der Webentwicklung auf ihre Art und Weise gelöst. Im Laufe der Zeit war diese Aufteilung allerdings nicht mehr die einzige Notwendigkeit in der Webentwicklung und andere Aspekte wie die Wiederver-

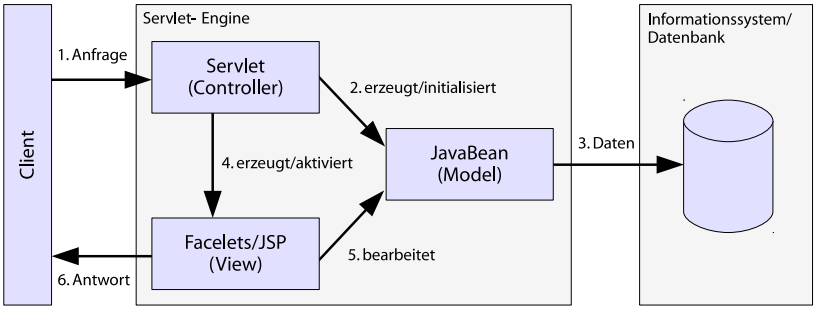

Abbildung 1.2
Das Model2-Prinzip als Spezialisierung der Model-View-Controller-Architektur

wendbarkeit von Komponenten rückten in den Vordergrund. Die Zeit war also mehr als reif für *JavaServer Faces* (JSF) als Basis für eine komponentenorientierte Entwicklung.

1.1.2 Entstehung von JavaServer Faces

JSF als Standard

JavaServer Faces (JSF) wurde nicht zuletzt als Technologie entwickelt, um die vielfältigen Ansätze zur Entwicklung von Webanwendungen unter Java zu standardisieren. Diese Standardisierung wird im Rahmen des Java Community Process (JCP) durchgeführt.

Der Java Community Process definiert die Rahmenbedingungen für die Entwicklung von Spezifikationen zur Erweiterung der Java-Plattform. Vorschläge für Spezifikationen werden dort in Form von Java Specification Requests (JSRs) mit einer fortlaufenden Nummer eingebracht und von einer Expert-Group bearbeitet. Jeder JSR durchläuft dabei einen mehrstufigen Prozess, bis eine finale Version vorliegt.

Die Spezifikation der Version 1.0 von *JavaServer Faces* (JSR-127) wurde 2004 veröffentlicht und nur wenige Monate später durch die fehlerbereinigte Version 1.1 ersetzt. Im Jahr 2006 folgte Version 1.2 der JSF-Spezifikation (JSR-252) als Teil von *Java EE 5*. Version 1.1 und Version 1.2 legten den Grundstein für den Aufstieg von *JavaServer Faces* zur wichtigsten Technologien in der Java-Webentwicklung – speziell JSF 1.2 war über mehrere Jahre sehr stark vertreten.

Mit der Einführung von Version 2.0 (JSR-314) im Jahr 2009 als Teil von *Java EE 6* wurde ein neues Kapitel der JSF-Entwicklung aufgeschlagen. In den drei Jahren zwischen der Veröffentlichung von Version 1.2 und Version 2.0 haben viele neue Trends und Technologien das Licht der Welt erblickt. Mit der steigenden Popularität von JSF hatte sich außerdem eine sehr aktive Community entwickelt. In zahlreichen Projekten wurden neue Komponentenbibliotheken, Bibliotheken zur Integration neuer Technologien oder Lösungen für Unzulänglichkeiten und nicht adressierte Bereiche in der Spezifikation entwickelt.

Die Expert-Group hat beim Entwurf von *JavaServer Faces* 2.0 einige der neuen Features an Lösungen aus damals populären Bibliotheken angelehnt. Durch die Standardisierung verbesserte sich die Kompatibilität von Komponentenbibliotheken und Erweiterungen verschiedener Hersteller, was wiederum das Leben der Entwickler vereinfachte.

JSF 2.1 brachte im November 2010 nur kleinen Änderungen an existierenden Features. Erst JSF 2.2 (JSR-344) brachte im Mai 2013 neben einer Vielzahl von Detailverbesserungen wieder eine ganze Reihe von neuen Features mit sich.

So viel zur geschichtlichen Entwicklung von JavaServer Faces. Wenn Sie bereits mit älteren JSF-Versionen Erfahrungen gesammelt haben, können Sie in den folgenden Abschnitten gezielt nach Informationen suchen. Abschnitt 1.2 zeigt Neuerungen von JSF 2.0 und 2.1 auf und Abschnitt 1.3 Neuerungen von JSF 2.2. In Abschnitt 1.5 geht es dann mit dem ersten Beispiel so richtig los.

1.2 JSF 2.0 und 2.1 im Überblick

In diesem Abschnitt fassen wir kurz die wichtigsten Neuerungen in JSF 2.0 und 2.1 mit Referenzen auf die entsprechenden Stellen im Buch zusammen.

- Facelets ist seit JSF 2.0 Teil des Standards (siehe Abschnitt 2.9). Abschnitt 4.2 zeigt weiterführende Informationen zu Facelets und in Abschnitt 4.3 finden Sie eine Einführung in Templating.
- Kompositkomponenten ermöglichen ab JSF 2.0 das Erstellen von eigenen Komponenten, ohne eine Zeile Java-Code zu schreiben. Wie das funktioniert, wird in Abschnitt 6.1 erläutert. Für Kompositkomponenten gibt es in JSF 2.1 einige kleinere Verbesserungen, die in den Abschnitten 6.1.4 und 6.1.9 näher erklärt werden.
- Die Integration von Bean-Validation erlaubt eine metadatenbasierte Validierung. Informationen dazu finden Sie in Abschnitt 2.12.1.
- Ajax wurde in den Standard integriert. Eine ausführliche Einführung finden Sie in Kapitel 7.
- Eine Reihe neuer Annotationen macht die Konfiguration von JSF-Anwendungen so einfach wie nie zuvor.
- Mit der Project-Stage kann die aktuelle Phase des Projekts im Entwicklungsprozess ermittelt werden. Abschnitt 4.1 zeigt die Details.
- JSF 2.0 standardisiert die Verwaltung von Ressourcen wie Skripte oder Stylesheets. Wie Sie davon profitieren, zeigt Kapitel 5.
- System-Events bieten die Möglichkeit, auf spezielle Ereignisse im Lebenszylus zu reagieren. Details finden Sie in Abschnitt 2.8.4.

- Die erweiterte Unterstützung von GET-Anfragen verbessert die Möglichkeit, Bookmarks zu setzen. Abschnitt 4.4 liefert die Details.
- JSF 2.0 vereinfacht das Navigieren mit impliziter Navigation. Näheres dazu finden Sie in Abschnitt 2.7.
- Partial-State-Saving optimiert das Speichern des Zustands von Ansichten. Details dazu finden Sie in Abschnitt 6.2.2.
- Mit dem View-Scope gibt es einen neuen Gültigkeitsbereich für Managed-Beans. Mehr dazu erfahren Sie in Abschnitt 2.4.

1.3 JSF 2.2 im Überblick

In diesem Abschnitt fassen wir kurz die wichtigsten Neuerungen in JSF 2.2 mit Referenzen auf die entsprechenden Stellen im Buch zusammen.

- Bei den Annotationen `@FacesValidator` und `@FacesComponent` ist das Element `value` jetzt optional und wird durch eine Namenskonvention ergänzt (siehe Abschnitt 2.12.3 und 6.2.4).
- Die Namensräume der JSF-Tag-Bibliotheken beginnen jetzt mit `http://xmlns.jcp.org` anstatt mit `http://java.sun.com`, wie unter anderem Kapitel 3 zeigt.
- `h:dataTable` unterstützt ab JSF 2.2 `java.util.Collection` als Typ, wie Abschnitt 3.4 zeigt.
- Die neue Komponente mit dem Tag `h:inputFile` ermöglicht endlich den Upload von Dateien (siehe Abschnitt 3.6.5).
- Die Unterstützung von GET-Anfragen in JSF wird mit View-Actions weiter vervollständigt. Details dazu finden Sie in Abschnitt 4.4.3.
- Das Verzeichnis, in dem JSF-Ressourcen in der Webapplikation aufgelöst werden, lässt sich jetzt konfigurieren (siehe Abschnitt 5.1).
- Mit Resource-Library-Contracts wurde die Grundlage für austauschbare Templates geschaffen, wie Abschnitt 5.6 zeigt.
- Tags für eigene Komponenten können mit JSF 2.2 direkt über `@FacesComponent` definiert werden (siehe Abschnitt 6.2.5).
- Mit JSF 2.2 können Sie in Tag-Bibliotheken Tags für einzelne Kompositkomponenten definieren, wie Abschnitt 6.6 zeigt.
- JSF 2.2 ermöglicht das Zurücksetzen von Eingabekomponenten mit `f:resetValues` (siehe Abschnitt 3.1) und dem Attribut `resetValues` von `f:ajax` (siehe Abschnitt 7.2.7).
- Mit dem neuen Attribut `delay` von `f:ajax` ermöglicht JSF eine genauere Kontrolle der Ajax-Queue (siehe Abschnitt 7.2.6).
- JSF unterstützt ab Version 2.2 mit Pass-Through-Attributen und Pass-Through-Elementen offiziell HTML5 (siehe Kapitel 8).

- JSF stellt ab Version 2.2 den View-Scope für CDI zur Verfügung. Mehr dazu erfahren Sie in Abschnitt 9.1.
- Mit Faces-Flows ermöglicht JSF die Gruppierung mehrerer Seiten zu wiederverwendbaren Modulen. Details finden Sie in Kapitel 11.

1.4 Das Ökosystem von JavaServer Faces

Wenn wir bisher von *JavaServer Faces* gesprochen haben, war immer die Spezifikation der Technologie gemeint. Zum Erstellen einer Applikation wird aber immer eine konkrete Implementierung dieser Spezifikation benötigt. Zurzeit gibt es mit *Apache MyFaces* und *Mojarra* – der Referenzimplementierung von *Oracle* – zwei frei verfügbare JSF-Implementierungen. Beide Projekte bieten den kompletten Funktionsumfang des JSF-Standards, unterscheiden sich jedoch in Details. Der größte Unterschied ist der Entwicklungsprozess: *MyFaces* wird komplett von einer Open-Source-Community entwickelt, wohingegen die Arbeit an *Mojarra* federführend von *Oracle* vorangetrieben wird.

Die JSF-Implementierung bietet nur ein beschränktes Set an Komponenten an. Im Laufe der letzten Jahre ist daher eine ganze Reihe von Komponentenbibliotheken entstanden, die neben einem erweiterten Angebot von Komponenten auch noch andere Konzepte zur Verfügung stellen, um die Entwicklung so einfach wie möglich zu gestalten.

Eine ausführliche Übersicht aller Komponentenbibliotheken würde den Rahmen dieses Buches sprengen. In Kapitel 10 dreht sich daher alles um den Einsatz von *PrimeFaces* – der wohl zurzeit populärsten Komponentenbibliothek für JSF. Hier noch eine Liste der bekanntesten Komponentenbibliotheken:

- *PrimeFaces*: `http://www.primefaces.org`
- *JBoss RichFaces*: `http://www.jboss.org/richfaces`
- *Apache MyFaces Tobago*: `http://myfaces.apache.org/tobago`
- *Apache MyFaces Tomahawk*: `http://myfaces.apache.org/tomahawk`
- *Apache MyFaces Trinidad*: `http://myfaces.apache.org/trinidad`
- *ICEfaces*: `http://www.icefaces.org`

1.5 Das erste JSF-Beispiel

Nichts ermöglicht einen besseren Einblick in eine Technologie als ein kurzes Beispiel. Daher werden wir den Einstieg in JavaServer Faces direkt mit einem *Hello World*-Beispiel beginnen. In einem ersten Schritt

beschreibt Abschnitt 1.5.1 die für die Buchbeispiele relevante Softwareumgebung. Nachdem alle Beispiele sehr ähnlich aufgebaut sind, werfen wir danach in Abschnitt 1.5.2 einen Blick auf die grundlegende Projektstruktur. In Abschnitt 1.5.3 geht es dann mit dem Beispiel *Hello World* richtig zur Sache. Den kompletten Quellcode aller Buchbeispiele finden Sie unter `http://jsfatwork.irian.at`.

1.5.1 Softwareumgebung

Als Grundlage für alle Beispiele und eingesetzten Tools muss ein Java Development Kit (JDK) in Version 6 oder 7 auf dem Rechner installiert sein.

Für einen einfachen Start mit JSF basieren alle unsere Beispiele auf dem weitverbreiteten Build-Werkzeug *Apache Maven*. *Maven* ist ein äußerst hilfreiches Mittel, um Java-basierte Projekte zu verwalten. Neben einer standardisierten Beschreibung von Projekten im *Project Object Model* (`pom.xml`) und einem standardisierten Build-Prozess bietet *Maven* außerdem eine automatische Auflösung von Abhängigkeiten zu anderen Projekten und Bibliotheken.

Eine detaillierte Einführung in die grundlegenden Konzepte von *Maven* würde den Rahmen dieses Kapitels sprengen – aber keine Sorge, wir lassen Sie nicht im Regen stehen. In Anhang A finden Sie allerhand Wissenswertes zu *Maven* inklusive einer Installationsanleitung. Dort zeigen wir Ihnen auch, wie Sie den Vorgang der Projekterstellung mit *Maven* automatisieren können.

Mit *Maven* ist die Webapplikation auch von der Kommandozeile startbar – außer einem simplen Editor ist theoretisch keine Entwicklungsumgebung notwendig. Leichter geht es allemal mit einer Entwicklungsumgebung wie *IntelliJ IDEA*, *Eclipse* oder *NetBeans*, zumal alle drei mittlerweile direkt mit *Maven*-Projekten umgehen können. Wir konzentrieren uns in diesem Buch auf *Eclipse* – nicht weil es die beste, sondern weil es die am weitesten verbereitete Entwicklungsumgebung für Java ist. *Eclipse* bietet mit der Erweiterung *Web Tools Platform (WTP)* sogar eine brauchbare Unterstützung für die Entwicklung von JSF-Anwendungen an. Details zur JSF-Entwicklung mit *Eclipse* finden Sie in Abschnitt 1.5.5 und in Anhang B.

Bei JSF-Anwendungen handelt es sich um klassische Java-Webapplikationen nach dem Servlet-Standard. Die Buchbeispiele benötigen als Laufzeitumgebung einen Servlet-Container, der mindestens Servlet 3.0 unterstützt wie *Apache Tomcat 7* oder *Jetty 8*. In Abschnitt 1.5.4 zeigen wir Ihnen, wie Sie das Beispiel mit *Maven* direkt von der Kommandozeile mit *Jetty 8* starten. In Abschnitt 1.5.5 finden Sie eine Anleitung zum Starten der Beispiele mit *Apache Tomcat 7* aus *Eclipse* heraus.

1.5.2 Projektstruktur mit Maven

Für *Maven* gilt das Motto: »Kennen Sie ein Projekt, kennen Sie alle«. Der Grund dafür ist, dass *Maven* per Konvention eine Struktur definiert, die von allen Projekten eingehalten werden sollte. Der Aufbau dieser Projektstruktur läuft also immer nach demselben Schema ab.

Im Projektverzeichnis wird neben der Beschreibung des Projekts in der Datei `pom.xml` noch das Verzeichnis `src` mit dem Unterverzeichnis `main` angelegt. Dort legen wir den Quellcode unseres Projekts in drei weiteren Unterverzeichnissen ab. Sämtliche Java-Klassen kommen ins Unterverzeichnis `java`, alle Ressourcen wie `.properties`-Dateien kommen ins Unterverzeichnis `resources` und alle für die Webapplikation relevanten Dateien ins Unterverzeichnis `webapp`. In Abbildung 1.3 sehen Sie die komplette Projektstruktur des *Hello World*-Beispiels.

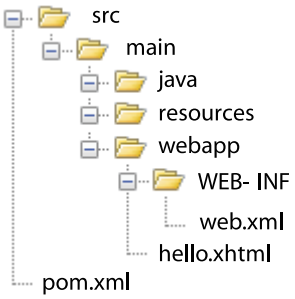

Abbildung 1.3
Struktur des Hello-World-Projekts

1.5.3 *Hello World*: das erste JSF-Projekt

Wenn Sie ein neues JSF-Projekt starten, sollten Sie sich zu Beginn für eine der beiden Implementierungen entscheiden. Wir möchten hier keine klare Empfehlung abgeben, da diese Entscheidung von vielen Faktoren abhängt. Nur so viel: Sie können sowohl mit *MyFaces* als auch mit *Mojarra* tolle JSF-Anwendungen bauen.

Dank *Maven* gestaltet sich das Einbinden der JSF-Implementierung als Kinderspiel. Sie muss lediglich als Abhängigkeit zur Beschreibung des Projekts in der Datei `pom.xml` hinzugefügt werden. Die komplette Datei finden Sie im Quellcode der Anwendung, für uns ist momentan nur der Teil mit der JSF-Implementierung interessant. Listing 1.4 zeigt die Abhängigkeiten für *Apache MyFaces* in Version 2.2.0-SNAPSHOT. `SNAPSHOT` signalisiert, dass es sich noch um eine Entwicklungsversion handelt – zur Zeit der Drucklegung dieses Buches stand die Version 2.2.0 kurz vor der Veröffentlichung. Die Bibliothek mit der Artifact-

ID `myfaces-api` beinhaltet die standardisierte API von JSF 2.2 und die Bibliothek mit der Artifact-ID `myfaces-impl` die Implementierung.

Listing 1.4 *Abhängigkeiten zu Apache MyFaces in der* `pom.xml`

```
<dependencies>
  <dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-api</artifactId>
    <version>2.2.0-SNAPSHOT</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-impl</artifactId>
    <version>2.2.0-SNAPSHOT</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

Listing 1.5 zeigt als Alternative die Abhängigkeiten für *Mojarra* in Version 2.2.2. Die Bibliothek mit der Artifact-ID `jsf-api` beinhaltet die standardisierte API von JSF 2.2 und die Bibliothek mit der Artifact-ID `jsf-impl` die konkrete Implementierung von *Mojarra*.

Listing 1.5 *Abhängigkeiten zu Mojarra in der* `pom.xml`

```
<dependencies>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.2</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.2</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

Standardmäßig verwenden alle Beispiele *Mojarra*. Wenn Sie zu Testzwecken die JSF-Implementierungen ändern wollen, müssen Sie dazu nicht die Datei `pom.xml` editieren. Alle *MyGourmet*-Beispiele definieren Profile für *Mojarra* (ist standardmäßig aktiv) und für *MyFaces*. Wie Sie diese Profile verwenden können, zeigt Anhang A.

Deklaration der Ansicht

Jetzt kommen wir zum wichtigsten Teil unserer Anwendung: Wie es sich für eine *Hello World*-Anwendung gehört, wollen wir auf der

Startseite unserer Anwendung den Text »Hello JSF 2.2-World« ausgeben. Dazu legen wir im Verzeichnis `webapp` die JSF-Seitendeklaration `hello.xhtml` an (siehe Listing 1.6).

Listing 1.6 *Die Seitendeklaration* `hello.xhtml`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://xmlns.jcp.org/jsf/html">
<head>
  <title>Hello World</title>
</head>
<body>
  <h:outputText value="Hello JSF 2.2-World"/>
</body>
</html>
```

Das Grundgerüst dieser Seite ist ein gewöhnliches XHTML-Dokument mit einem im `body`-Element eingebetteten `h:outputText`-Tag zur Ausgabe der Meldung. Dieses von JSF zur Verfügung gestellte Tag gibt den im Attribut `value` angegebenen Text aus. Das Präfix `h:` ist dabei mit dem in JSF 2.2 neu definierten Namensraum `http://xmlns.jcp.org/jsf/html` verbunden und kennzeichnet die HTML-Tag-Bibliothek von JSF. Sie enthält neben dem Tag `h:outputText` noch eine Reihe weiterer Tags für Standard-JSF-Komponenten und ihre Darstellung als HTML-Ausgabe – doch dazu später mehr in Kapitel 3.

web.xml

Im zweiten Schritt erstellen wir die Webkonfigurationsdatei `web.xml` im `/WEB-INF`-Verzeichnis unserer Webanwendung[1] so, dass auf die JSF-Technologie zugegriffen werden kann. Das geschieht durch die Einbindung des JSF-Servlets in Form einer Servlet-Definition und eines Servlet-Mappings, wie es Listing 1.7 zeigt. Durch das angegebene `servlet-mapping`-Element werden sämtliche Anfragen mit der Endung `.xhtml` von genau diesem JSF-Servlet bearbeitet.

Über den Kontextparameter `javax.faces.PROJECT_STAGE` wird die Project-Stage auf `Development` gesetzt. Damit teilen wir JSF mit, dass wir uns aktuell in der Entwicklungsphase des Projekts befinden. Welche Auswirkungen das mit sich bringt, erfahren Sie in Abschnitt 4.1.

Zu guter Letzt definieren wir noch die Seite `hello.xhtml` als Welcome-File der Anwendung. Damit ist gewährleistet, dass die Seite immer dann angezeigt wird, wenn ein Benutzer im Browser die URL der Anwendung ohne Angabe einer speziellen Seite eingibt.

Herzlichen Glückwunsch – Sie haben soeben Ihre erste Webanwendung mit *JavaServer Faces* verfasst! Im nächsten Abschnitt zeigen wir Ihnen,

[1] Die Datei `web.xml` wird auch *Deployment*-Deskriptor der Webanwendung genannt.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   version="3.0">
  <description>JSF 2.0 - Hello World</description>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
      javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>hello.xhtml</welcome-file>
  </welcome-file-list>
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
</web-app>
```

Listing 1.7
Die Konfigurationsdatei `web.xml` *mit der Spezifikation eines* `FacesServlet` *sowie des zugehörigen Servlet-Mappings*

wie Sie die Anwendung direkt mit *Maven* starten können. Dieses Beispiel war selbstverständlich erst der Einstieg, wenn Sie also noch Fragen haben, laden wir Sie zum Weiterlesen ein.

1.5.4 Starten der Anwendung mit Maven

Zum Starten der *Hello World*-Applikation kommt das *Jetty-Maven-Plug-in* zur Anwendung. *Jetty* ist ein Servlet-Container, der als Laufzeitumgebung für unsere JSF-Applikation dient und von der Konsole aus zu starten ist. Schnelles Prototyping für erste Versionen der Webapplikation kann hiermit perfekt zum Zug kommen. Der Befehl, um den Server zu starten, lautet:

```
mvn clean jetty:run
```

Eingegeben werden muss dieser ebenfalls wieder im Projektverzeichnis. Die benötigten Dateien werden durch *Maven* erneut automatisch in das lokale Repository geladen. Danach startet der Server und die Applikation kann in der Adresszeile des Browsers wie folgt aufgerufen werden:

```
http://localhost:8080/helloworld/
```

Der Build-Prozess des Projekts kann in weiterer Folge mit diesem Befehl neu angestoßen werden:

```
mvn install
```

Maven erstellt dabei den Unterordner `target` mit den kompilierten Klassen und dem `.war`-Archiv. Die `.war`-Datei enthält alle zur Ausführung der Webapplikation benötigten Bibliotheken, die *Maven* über die Abhängigkeiten in der `pom.xml`-Projektdatei eingefügt hat. Das Projekt wurde ins lokale Repository unter der Group-Id `at.irian.jsfatwork` und der Artifact-Id `helloworld` installiert. Abbildung 1.4 zeigt die Verzeichnisstruktur im lokalen Repository.

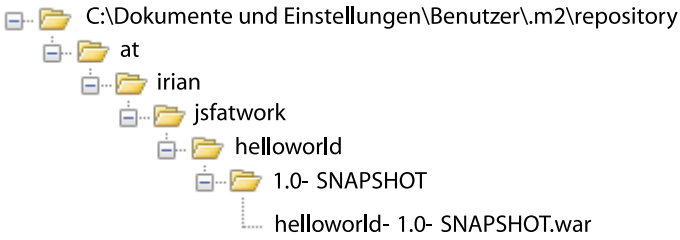

Abbildung 1.4 *Anwendung im lokalen Repository*

1.5.5 Entwicklung mit Eclipse

Mit *Maven* verfügen wir bereits über eine solide Basis für die einfache und effiziente Verwaltung von JSF-Projekten. Bis jetzt haben wir *Maven* allerdings nur von der Kommandozeile aus benutzt. Die tägliche Entwicklungsarbeit gestaltet sich jedoch mit einer Entwicklungsumgebung wie *IntelliJ IDEA*, *Eclipse* oder *NetBeans* erheblich einfacher. Zum Glück ist das mittlerweile kein Problem mehr, da alle oben genannten Entwicklungsumgebungen den direkten Umgang mit Maven-Projekten unterstützen.

Wir konzentrieren uns in diesem Abschnitt auf die JSF-Entwicklung mit *Eclipse*, da es frei verfügbar und sehr weit verbreitet ist. Eine ausführliche Anleitung, um *Eclipse* für die Arbeit mit JSF und den Buchbeispielen einzurichten, findet sich in Anhang B.

Arbeiten mit Eclipse

Nach dem Starten von *Eclipse* sollten Sie sich wie in Abbildung 1.5 gezeigt in der »Java EE«-Perspektive befinden. Falls dem nicht so ist, können Sie über `Window|Open Perspective|Other...` in diese Perspektive wechseln.

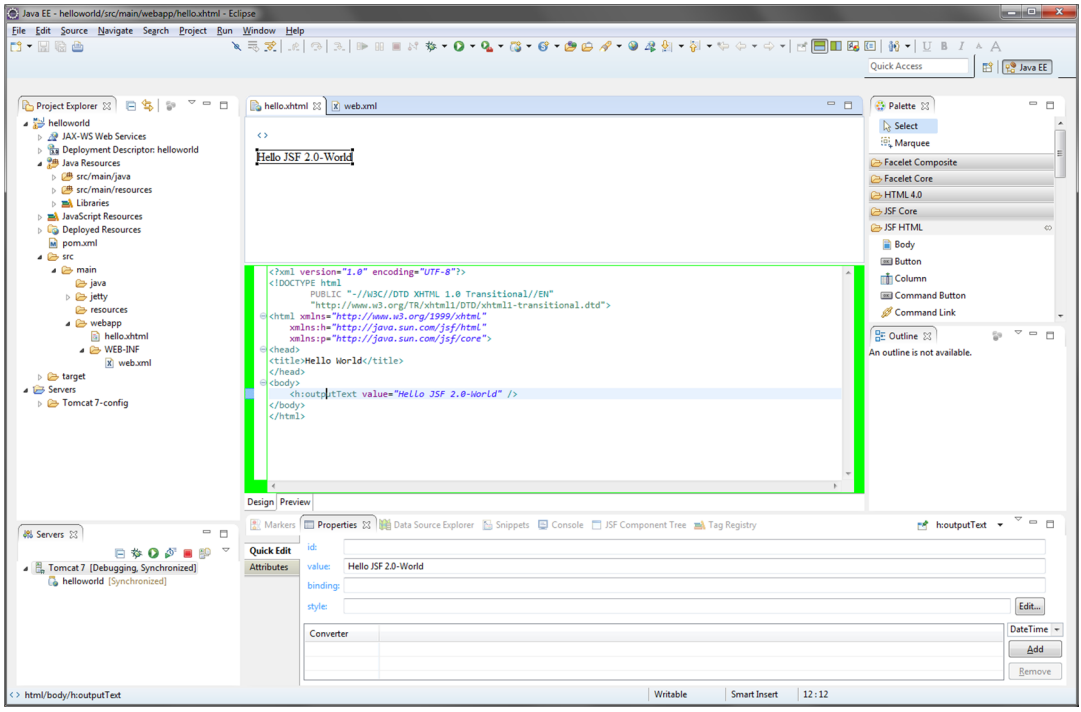

Abbildung 1.5
Eclipse mit geöffnetem Hello World-Projekt

Wie Abbildung 1.5 zeigt, stellt *Eclipse* mittlerweile einen Editor und eine WYSIWYG-Ansicht für die einzelnen JSF-Seiten zur Verfügung. Mit diesem Editor ist es kinderleicht, JSF-Seiten selbst zu erstellen und Komponenten auf diesen Seiten einzubinden. Der Editor wird über einen Doppelklick auf eine JSF-Datei gestartet. Dadurch öffnet sich der JSF-Editor mit einer Quellcode- und einer WYSIWYG-Ansicht. Von der Werkzeugleiste rechts im Bild können Komponenten direkt in den oberen oder unteren Teil gezogen werden, die entstandenen Komponenten werden dann automatisch von der WYSIWYG-Ansicht dargestellt.

Eine weiteres hilfreiches Feature ist das *Properties*-Tab. Dort finden Sie eine Auflistung aller Attribute der im Editor selektierten Komponente mit der Möglichkeit zum Bearbeiten der Werte. Sollte das *Properties*-Tab nicht angezeigt werden, können Sie es über `Window|Show View|Properties` einblenden.

Als Beispiel werden wir in unserer XHTML-Datei mit dem Namen `hello.xhtml` eine neue Komponente einfügen. Durch einen Doppelklick auf die Datei öffnet sich der Editor. Falls *Eclipse* die Datei in einem »normalen« Editor öffnet, müssen Sie den Editortyp im Kontextmenü über `Open With|Web Page Editor` umstellen. Selektieren Sie dann in der Komponentenpalette im Tab »JSF HTML« das Element »Output

Text« und ziehen Sie es in die Quellcodeansicht oder in die WYSIWYG-Ansicht. Mit einem Klick auf die Komponente im Editor werden die Attribute im *Properties*-Tab angezeigt. Geben Sie dort für das Attribut `value` beispielsweise den Wert »Hello again!« ein. Sie können zusätzlich das Aussehen der Komponente ändern, indem Sie für das Attribut `style` zum Beispiel den Wert »color: Red« eintragen. Die Darstellung in der WYSIWYG-Ansicht wird sofort angepasst. Abbildung 1.6 zeigt den Editor mit der hinzugefügten Komponente und deren Attribute im *Properties*-Tab.

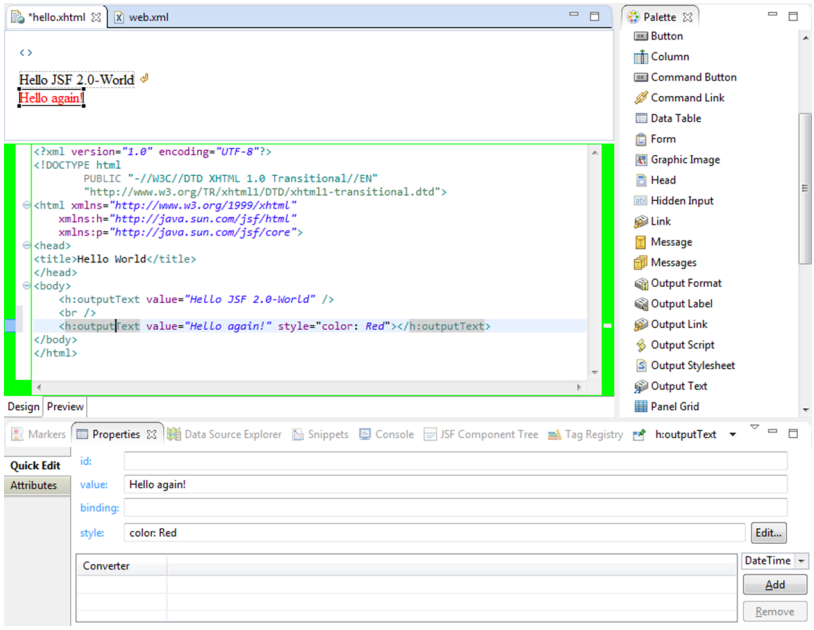

Abbildung 1.6
Eclipse WTP Property-Editor

Starten der Anwendung mit Eclipse

Aus *Eclipse* heraus können Sie JSF-Anwendungen direkt auf einer ganzen Reihe unterschiedlicher Server starten und debuggen. Dazu müssen Sie zuerst das zu startende Projekt im Projekt-Explorer selektieren und dann im Kontextmenü oder im Menü `Run` den Eintrag `Run As|Run on Server` auswählen. Zum Starten im Debug-Modus rufen Sie statt `Run As|Run on Server` den Menüeintrag `Debug As|Debug on Server` auf. Falls Sie noch keinen Server konfiguriert haben, öffnet *Eclipse* an dieser Stelle einen Assistenten zum Einrichten. Für die Buchbeispiele eignet sich *Apache Tomcat 7.0* besonders gut – eine detaillierte Anleitung zum Einrichten finden Sie im Anhang in Abschnitt B.3.

Beim Hochfahren des Servers werden sämtliche Logmeldungen in einem eigenen Konsolenfenster angezeigt. Nach erfolgreichem Start öffnet *Eclipse* standardmäßig ein Browserfenster mit der Applikation. Sie können die Webanwendung aber auch wie folgt in einem Browser Ihrer Wahl aufrufen:

```
http://localhost:8080/helloworld/
```

Der Port `8080` und der Kontextpfad `helloworld` beziehen sich dabei auf unser *Hello World*-Beispiel. Die konkrete Konfiguration eines Servers können Sie mit einem Doppelklick auf den entsprechenden Eintrag im *Servers*-Tab öffnen und bearbeiten.

Manchmal kann es trotz korrekten Codes zu unerklärlichen Fehlern in der JSF-Applikation kommen. In solchen Fällen ist es oft hilfreich, den Verteilungsprozess neu in Gang zu setzen, um Probleme durch unvollständig oder gar nicht neu verteilte Dateien zu lösen. Selektieren Sie dazu den betroffenen Server im *Servers*-Tab und wählen Sie `Clean...` im Kontextmenü.

Hilft auch diese Maßnahme nicht, bleibt in zweiter Instanz nur das Neustarten von Eclipse. Abhilfe kann auch das Löschen und Neuerstellen des *Server*-Eintrags im *Servers*-Tab schaffen.

Nach diesem Abstecher in die Welt der Build-Werkzeuge und Entwicklungsumgebungen widmen wir den nächsten Abschnitt der ersten Version unseres *MyGourmet*-Beispiels.

1.6 *MyGourmet 1*: Einführung anhand eines Beispiels

Im Laufe des Buches wird schrittweise eine kleine Beispielapplikation mit dem Namen *MyGourmet* aufgebaut. Die Anwendung soll einen Online-Bestellservice für lukullische Genüsse jeglicher Art darstellen. Der Fokus liegt dabei verständlicherweise weniger auf vollständiger Funktionalität oder perfektem Design, sondern auf der Vermittlung der Basiskonzepte von *JavaServer Faces*. Jeder Schritt erweitert *MyGourmet* um die im jeweiligen Kapitel vorgestellten Aspekte von JSF. Sie finden den Sourcecode für alle Beispiele dieses Buches unter der Adresse `http://jsfatwork.irian.at`.

Im ersten Schritt erweitern wir unser *Hello World*-Beispiel um ein einfaches Formular zur Eingabe der Daten eines Kunden. Es existiert ein Feld für die Eingabe des Vornamens und des Nachnamens und eine Absendeschaltfläche. Nach dem Betätigen der Schaltfläche werden die gerade eingegebenen Daten noch einmal dargestellt, und zwar in

entsprechenden Ausgabefeldern mit einer zusätzlich eingeblendeten Erfolgsmeldung.

Zuerst sollten wir die Klassen unseres Datenmodells so fertigstellen, dass wir sie in der Webapplikation verwenden können. Das ist einfach – eine simple Java-Klasse `Customer` mit den zwei Klassenvariablen `firstName` und `lastName` und den dazugehörigen Zugriffsmethoden `getFirstName()`, `setFirstName(String firstName)`, `getLastName()` und `setLastName(String lastName)` reichen dazu aus. Die Klasse ist in Listing 1.8 dargestellt.

Listing 1.8 *Die Klasse* `Customer`

```
1  package at.irian.jsfatwork.gui.page;
2
3  import javax.faces.bean.ManagedBean;
4  import javax.faces.bean.SessionScoped;
5
6  @ManagedBean
7  @SessionScoped
8  public class Customer {
9    private String firstName;
10   private String lastName;
11
12   public String getFirstName() {
13     return firstName;
14   }
15   public void setFirstName(String firstName) {
16     this.firstName = firstName;
17   }
18   public String getLastName() {
19     return lastName;
20   }
21   public void setLastName(String lastName) {
22     this.lastName = lastName;
23   }
24 }
```

Managed-Bean

Der Zugriff auf das Datenmodell erfolgt in JSF über sogenannte Managed-Beans. In JSF versteht man darunter *JavaBeans*, die unter einem eindeutigen Namen in der Anwendung zur Verfügung stehen. Um eine Managed-Bean vom Typ `Customer` zu registrieren, genügt es ab JSF 2.0, die Klasse mit `@ManagedBean` zu annotieren. Der Name, unter dem die Bean zur Verfügung steht, wird vom Klassennamen abgeleitet und lautet in unserem Fall `customer`.

Die Bean ist dabei einem zeitlich eingeschränkten und auf den Benutzer bezogenen Gültigkeitsbereich zugeordnet. Mit der ebenfalls in

Version 2.0 eingeführten Annotation `@SessionScoped` weisen wir JSF an, die Managed-Bean einmal pro HTTP-Session neu zu erzeugen.

Deklaration der Ansicht

Jetzt kommen wir zum wichtigsten Teil unserer Anwendung: Irgendwo muss auf diese Managed-Bean zugegriffen werden, und das machen wir in einer Facelets-Seite[2]. In *MyGourmet 1* ist das die Seite `editCustomer.xhtml` zum Erfassen des Vor- und Nachnamens des Kunden. Das Grundgerüst der Seite ist wie schon beim *Hello World*-Beispiel ein HTML-Dokument mit eingebetteten JSF-Tags im `body`-Element.

Damit wir mit unserer Seite überhaupt Benutzereingaben verarbeiten können, brauchen wir ein Formular. JSF stellt dazu in der HTML-Tag-Bibliothek das Tag `h:form` zur Verfügung. Die Eingabefelder für den Vor- und den Nachnamen des Kunden werden mit dem Tag `h:inputText` innerhalb des Formulars in die Seite eingefügt. Damit Benutzer der Anwendung die Eingabefelder unterscheiden können, bekommen sie über das Tag `h:outputLabel` ein Label. Die Verbindung zwischen dem Label und dem Eingabefeld erfolgt, indem die ID des Eingabefelds in das `for`-Attribut von `h:outputLabel` eingetragen wird. Zum Ausrichten der einzelnen Elemente in einer tabellenförmigen Struktur kommt `h:panelGrid` zum Einsatz.

Interessant ist bei diesen Tags das `value`-Attribut von `h:inputText`. Es beinhaltet eine Value-Expression, über die der Wert einer Komponente mit einer Eigenschaft einer Managed-Bean verbunden werden kann. Das geschieht mit folgender Syntax: Nach einer Raute[3] folgt in geschwungenen Klammern der Name der Eigenschaft in der Form `bean.eigenschaft`. Allgemein ergibt das einen Ausdruck in der Form `#{managedBean.eigenschaft}` – wie in Listing 1.9 mehrfach zu sehen.

Diese Applikation können wir bereits ausführen, wir werden eine Seite mit den von uns definierten Eingabefeldern erhalten. Der nächste Schritt ist das Weiterleiten des Benutzers auf die Seite `showCustomer.xhtml`, was in unserem Fall durch eine Schaltfläche erfolgen soll. Wir fügen also eine Schaltfläche zu unserer XHTML-Seite hinzu. Das entsprechende JSF-Tag heißt `h:commandButton`. Diese Schaltfläche versehen wir mit dem Attribut `action`, das den Wert `/showCustomer.xhtml` erhält, und dem Attribut `value` mit der im Browser darzustellenden Beschriftung `Save`. Ein Klick auf die Schaltfläche bewirkt, dass JSF den Benutzer auf die im Attribut `action` angegebene Seite weiterleitet.

Vor JSF 2.0 musste die Navigation noch verpflichtend in der Konfigurationsdatei `faces-config.xml` in Form von Navigationsregeln defi-

[2] Facelets ist seit JSF 2.0 Teil des Standards und JavaServer Pages vorzuziehen, mehr dazu in Abschnitt 2.9.

[3] Ab JSF 1.2 darf auch ein »$«-Zeichen – wie in der früher definierten *JSP Expression Language* – verwendet werden.

niert werden. Ab JSF 2.0 kann dieser Schritt durch das direkte Angeben der Seite entfallen. Weiterführende Informationen zum Thema Navigation finden Sie in Abschnitt 2.7.

Der komplette Sourcecode der Seite `editCustomer.xhtml` ist in Listing 1.9 zu finden.

Listing 1.9 *Die Datei* `editCustomer.xhtml`

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">
<head>
  <title>MyGourmet - Edit Customer</title>
</head>
<body>
  <h1><h:outputText value="MyGourmet"/></h1>
  <h2><h:outputText value="Edit Customer"/></h2>
  <h:form id="form">
    <h:panelGrid id="grid" columns="2">
      <h:outputLabel value="First Name:" for="firstName"/>
      <h:inputText id="firstName"
          value="#{customer.firstName}"/>
      <h:outputLabel value="Last Name:" for="lastName"/>
      <h:inputText id="lastName"
          value="#{customer.lastName}"/>
    </h:panelGrid>
    <h:commandButton id="save" value="Save"
        action="/showCustomer.xhtml"/>
  </h:form>
</body>
</html>
```

Abbildung 1.7 zeigt die Darstellung der Seite im Browser und den Zusammenhang zu den JSF-Komponenten in der XHTML-Datei.

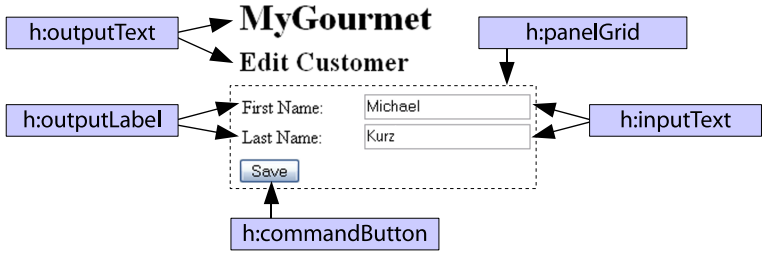

Abbildung 1.7 *MyGourmet 1: Komponenten und ihre Darstellung*