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 Technologien 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**Beispiel für ein**

einfaches Servlet. Hier wird die GET-Methode der HTTP-Anfrage behandelt..

```
public class BeispielServlet extends HttpServlet {
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
      throws ServletException, IOException {
    String name = req.getParameter("name");
    String text = req.getParameter("text");

    res.setContentType("text/html");

    ServletOutputStream out = res.getOutputStream();
    out.println("<html><head><title>");
    out.println(name);
    out.println("</title></head><body>");
    out.println(text);
    out.println("</body></html>");
    out.flush();
  }
}
```

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**Beispiel für eine einfache (jedoch unvollständige) Hilfsklasse zum Schreiben von HTML-Code in Servlets**zeigt.

```
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(">");
  }
  ...
}
```

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**Ein einfaches JSP-Beispiel**. Hier ist HTML die treibende Kraft und in die einzelnen Tags der HTML 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>
```

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 Applika-tionsserver 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 selbstverständlich 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 Seitendeklara-tionssprache 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**Das Model-View-Controller-Prinzip**dargestellt.

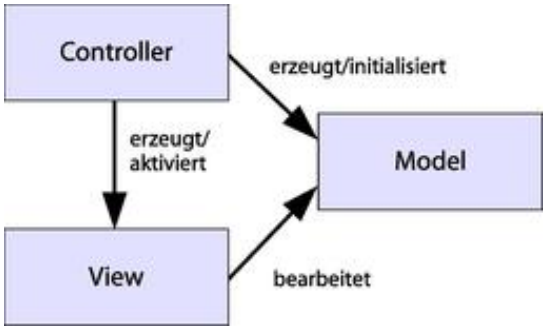


Abbildung:Das Model-View-Controller-Prinzip

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**Das Model2-Prinzip als Spezialisierung der Model-View-Controller-Architektur**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.

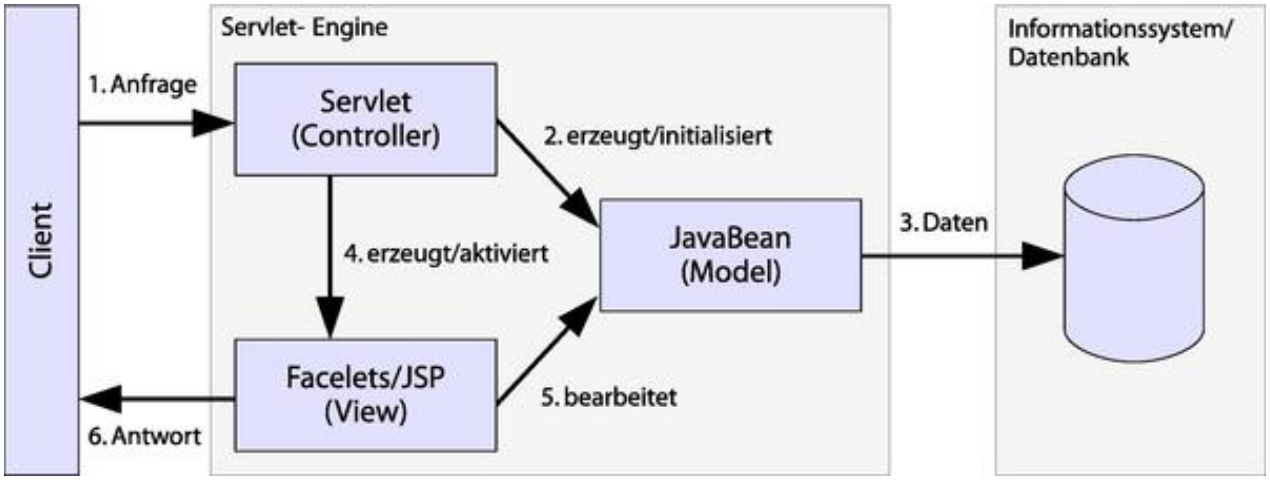


Abbildung:Das Model2-Prinzip als Spezialisierung der Model-View-Controller-Architektur

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 Wiederverwendbarkeit 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**[Sektion: JSF 2.0 und 2.1 im Überblick]**zeigt Neuerungen von JSF 2.0 und 2.1 auf und

Abschnitt[Sektion: JSF 2.2 im Überblick]Neuerungen von JSF 2.2. In Abschnitt[Sektion: Das erste JSF-Beispiel]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**Sektion: Seitendeklarationssprachen**). Abschnitt**Sektion: Advanced Facelets**zeigt weiterführende Informationen zu Facelets und in Abschnitt**Sektion: Templating**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**Sektion: Kompositkomponenten**erläutert. Für Kompositkomponenten gibt es in JSF 2.1 einige kleinere Verbesserungen, die in den Abschnitten**Sektion: Ressourcen in Kompositkomponenten**und**Sektion: Fallstricke in der Praxis**näher erklärt werden.
- Die Integration von Bean-Validation erlaubt eine metadatenbasierte Validierung. Informationen dazu finden Sie in Abschnitt**Sektion: Bean-Validation nach JSR-303**.
- Ajax wurde in den Standard integriert. Eine ausführliche Einführung finden Sie in Kapitel**Kapitel: Ajax und JSF**.
- 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**Sektion: Project-Stage**zeigt die Details.
- JSF 2.0 standardisiert die Verwaltung von Ressourcen wie Skripte oder Stylesheets. Wie Sie davon profitieren, zeigt Kapitel**Kapitel: Verwaltung von Ressourcen**.
- System-Events bieten die Möglichkeit, auf spezielle Ereignisse im Lebenszylus zu reagieren. Details finden Sie in Abschnitt**Sektion: System-Events**.
- Die erweiterte Unterstützung von GET-Anfragen verbessert die Möglichkeit, Bookmarks zu setzen. Abschnitt**Sektion: Bookmarks und GET-Anfragen in JSF**liefert die Details.
- JSF 2.0 vereinfacht das Navigieren mit impliziter Navigation. Näheres dazu finden Sie in Abschnitt**Sektion: Navigation**.
- Partial-State-Saving optimiert das Speichern des Zustands von Ansichten. Details dazu finden Sie in Abschnitt**Sektion: Komponentenklasse schreiben**.
- Mit dem View-Scope gibt es einen neuen Gültigkeitsbereich für Managed-Beans. Mehr dazu erfahren Sie in Abschnitt**Sektion: Managed-Beans**.

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**Sektion: Benutzerdefinierte Validatoren**und**Sektion: Registrieren der Komponenten- und der Rendererklasse**).
- Die Namensräume der JSF-Tag-Bibliotheken beginnen jetzt mit`http://xmlns.jcp.org`anstatt mit`http://java.sun.com`, wie unter anderem Kapitel**Kapitel: Standard-JSF-Komponenten**zeigt.
- `h:dataTable`unterstützt ab JSF 2.2`java.util.Collection`als Typ, wie Abschnitt**Sektion: DataTable-Komponente**zeigt.
- Die neue Komponente mit dem Tag`h:inputFile`ermöglicht endlich den Upload von Dateien (siehe Abschnitt**Sektion: Dateiuploadfeld h:inputFile**).
- Die Unterstützung von GET-Anfragen in JSF wird mit View-Actions weiter vervollständigt. Details dazu finden Sie in Abschnitt**Sektion: View-Actions**.
- Das Verzeichnis, in dem JSF-Ressourcen in der Webapplikation aufgelöst werden, lässt sich jetzt konfigurieren (siehe Abschnitt**Sektion: Identifikation von Ressourcen -- Teil 1**).
- Mit Resource-Library-Contracts wurde die Grundlage für austauschbare Templates geschaffen, wie Abschnitt**Sektion: Resource-Library-Contracts**zeigt.
- Tags für eigene Komponenten können mit JSF 2.2 direkt über`@FacesComponent`definiert werden (siehe Abschnitt**Sektion: Tag-Definition schreiben**).
- Mit JSF 2.2 können Sie in Tag-Bibliotheken Tags für einzelne Kompositkomponenten definieren, wie Abschnitt**Sektion: Die eigene Komponentenbibliothek**zeigt.
- JSF 2.2 ermöglicht das Zurücksetzen von Eingabekomponenten mit`f:resetValues`(siehe Abschnitt**Sektion: Basisfunktionen der Core-Tag-Library**) und dem Attribut`reset-Values`von`f:ajax`(siehe Abschnitt**Sektion: Eingabefelder zurücksetzen**).
- Mit dem neuen Attribut`delay`von`f:ajax`ermöglicht JSF eine genauere Kontrolle der Ajax-Queue (siehe Abschnitt**Sektion: Ajax-Queue kontrollieren**).
- JSF unterstützt ab Version 2.2 mit Pass-Through-Attributen und Pass-Through-Elementen offiziell HTML5 (siehe Kapitel**Kapitel: JSF und HTML5**).
- JSF stellt ab Version 2.2 den View-Scope für CDI zur Verfügung. Mehr dazu erfahren Sie in Abschnitt**Sektion: Beans und Dependency-Injection mit CDI**.
- Mit Faces-Flows ermöglicht JSF die Gruppierung mehrerer Seiten zu wiederverwendbaren Modulen. Details finden Sie in Kapitel**Kapitel: Faces-Flows**.

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 Komponenten-bibliotheken 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**Kapitel: PrimeFaces -- JSF und mehr**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**[Sektion: Softwareumgebung]**die für die Buchbeispiele relevante Soft-wareumgebung. Nachdem alle Beispiele sehr ähnlich aufgebaut sind, werfen wir danach in Abschnitt**[Sektion: Projektstruktur mit Maven]**einen Blick auf die grundlegende Projektstruktur. In Abschnitt**[Sektion: *Hello World*: das erste JSF-Projekt]**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**Kapitel: Eine kurze Einführung in Maven**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**[Sektion: Entwicklung mit Eclipse]**und in Anhang**Kapitel: Eclipse**.
Bei JSF-Anwendungen handelt es sich um klassische Java-Web-applikationen 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**[Sektion: Starten der Anwendung mit Maven]**zeigen wir Ihnen, wie Sie das Beispiel mit*Maven*direkt von der Kommandozeile mit*Jetty 8*starten. In Abschnitt**[Sektion: Entwicklung mit Eclipse]**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**Struktur des Hello-World-Projekts**sehen Sie die komplette Projektstruktur des*Hello World*-Beispiels.

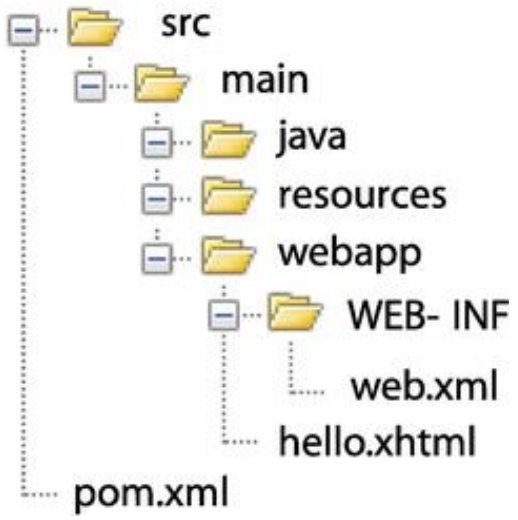


Abbildung:Struktur des Hello-World-Projekts

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**Abhängigkeiten zu Apache MyFacesin der pom.xml**zeigt die Abhängigkeiten für*Apache MyFaces*in Version 2.2.1. 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.

```
<dependencies>
  <dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-api</artifactId>
    <version>2.2.1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-impl</artifactId>
    <version>2.2.1</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

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

```
<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**Kapitel: Eine kurze Einführung in Maven**.
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**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 eingebettetem`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**Kapitel: Standard-JSF-Komponenten**.
web.xml:Im zweiten Schritt erstellen wir die Webkonfigurationsdatei`web.xml`im`/WEB-INF`-Verzeichnis unserer WebanwendungDie Datei`web.xml`wird auch*Deployment*-Deskriptor der Webanwendung genannt.: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**Die Konfigurations-datei web.xml mit der Spezifikation eines FacesServlet sowie des zugehörigen Servlet-Mappings**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**Sektion: Project-Stage**.

```
<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>
```

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, 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 Web-applikation 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**Anwendung im lokalen Repository**zeigt die Verzeichnisstruktur im lokalen Repository.

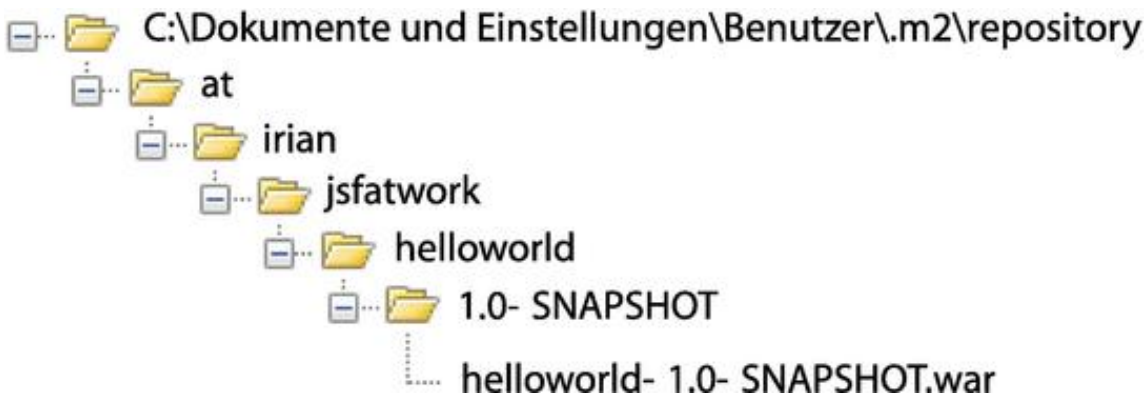


Abbildung:Anwendung im lokalen Repository

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**Kapitel: Eclipse**.

1.5.5.1 Arbeiten mit Eclipse

Nach dem Starten von*Eclipse*sollten Sie sich wie in Abbildung**Eclipse mit geöffnetem Hello World-Projekt**gezeigt in der "Java EE"-Perspektive befinden. Falls dem nicht so ist, können Sie über`Window|Open Perspective|Other...`in diese Perspektive wechseln.

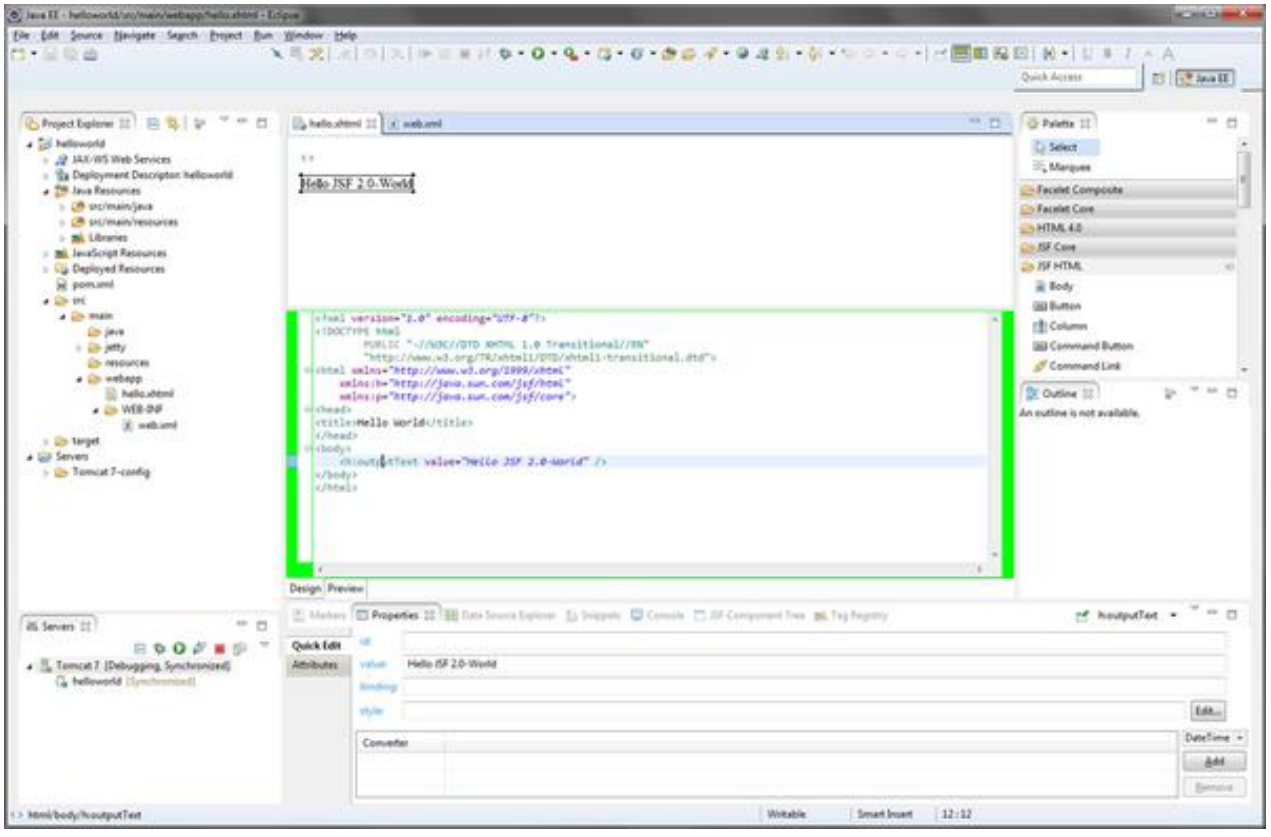


Abbildung:Eclipse mit geöffnetem Hello World-Projekt

Wie Abbildung**Eclipse mit geöffnetem Hello World-Projekt**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**Eclipse WTP Property-Editor**zeigt den Editor mit der hinzugefügten Komponente und deren Attribute im*Properties*-Tab.

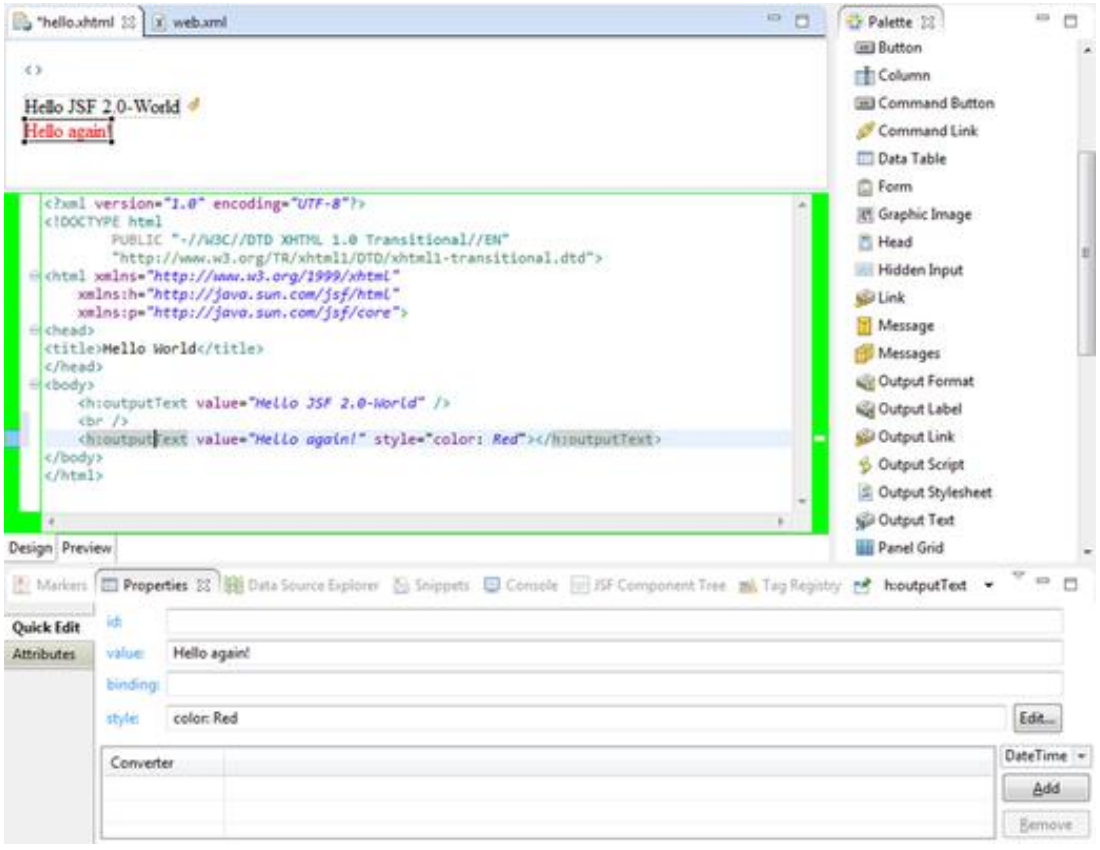


Abbildung:Eclipse WTP Property-Editor

1.5.5.2 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 MenüRunden EintragRun `As|Run on Server`auswählen. Zum Starten im Debug-Modus rufen Sie stattRun `As|Run on Server`den MenüeintragDebug `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**Sektion: Apache Tomcat 7 in Eclipse einrichten**.

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 Klassen-variablen und`lastName`und den dazugehörigen Zugriffsmethoden`getFirstName()`,`setFirstName(String firstName)`,`getLastName()`und`setLastName(String lastName)`reichen dazu aus. Die Klasse ist in Listing**Die Klasse Customer**dargestellt.

```
package at.irian.jsfatwork.gui.page;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class Customer {
  private String firstName;
```

```
  private String lastName;

  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
}
```

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-SeiteFacelets ist seit JSF 2.0 Teil des Standards und JavaServer Pages vorzuziehen, mehr dazu in Abschnitt**Sektion: Seitendeklarationssprachen**.:. 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 RauteAb JSF 1.2 darf auch ein "$"-Zeichen - wie in der früher definierten*JSP Expression Language*- verwendet werden.: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**Die Datei editCustomer.xhtml**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 definiert werden. Ab JSF 2.0 kann dieser Schritt durch das direkte Angeben der Seite entfallen. Weiterführende Informationen zum Thema Navigation finden Sie in Abschnitt**Sektion: Navigation**.
Der komplette Sourcecode der Seite`editCustomer.xhtml`ist in Listing**Die Datei editCustomer.xhtml**zu finden.

```
<!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[MyGourmet 1: Komponenten und 10mmihre Darstellung](#)zeigt die Darstellung der Seite im Browser und den Zusammenhang zu den JSF-Komponenten in der XHTML-Datei.

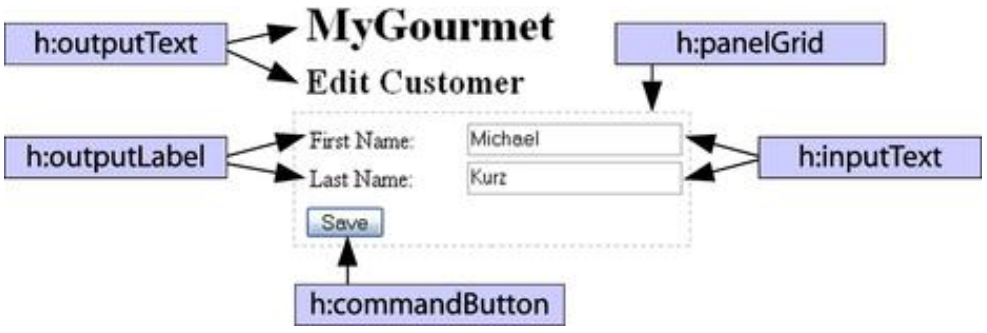


Abbildung:MyGourmet 1: Komponenten und 10mmihre Darstellung

Bevor wir auf die Seite`showCustomer.xhtml`(Listing[Die Datei showCustomer.xhtml](#)) navigieren können, müssen wir sie zuerst erstellen. Die neue Seite soll ähnlich der ersten Seite aussehen, nur ersetzen jetzth`:outputText`-Tags dieh`:inputText`-Elemente und ein zusätzlichesh`:outputText`-Tag gibt die Nachricht "Customer saved successfully!" aus.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">
<head>
  <title>MyGourmet - Show Customer</title>
</head>
```

```
<body>
  <h1><h:outputText value="MyGourmet"/></h1>
  <h2><h:outputText value="Show Customer"/></h2>
  <h:panelGrid id="grid" columns="2">
    <h:outputText value="First Name:"/>
    <h:outputText value="#{customer.firstName}"/>
    <h:outputText value="Last Name:"/>
    <h:outputText value="#{customer.lastName}"/>
  </h:panelGrid>
  <h:outputText value="Customer saved successfully!"/>
</body>
</html>
```

Applikationslogik ausführen:Fertig! Die Applikation funktioniert bereits so wie gewünscht und leitet uns von der ersten Seite durch einen Klick auf die Schaltfläche weiter - auf der zweiten Seite werden die eingegebenen Daten angezeigt. In einer "realen" Applikation würden wir die Daten jetzt abspeichern, dazu müssen wir durch die von der Schaltfläche ausgelöste Aktion auf eine Methode der dahinterliegenden Managed-Bean zugreifen. Auch dieser Schritt ist unkompliziert, statt das Attribut`action`direkt auf eine Zeichenkette zu setzen, verwenden wir eine Method-Expression, die auf eine Methode in der dahinterliegenden Managed-Bean referenziert. Mit der gleichen Syntax, mit der wir vorher auf eine Variable in der Managed-Bean`customer`zugegriffen haben, können wir jetzt auch eine Methode referenzieren. Der geänderte Code der Schaltfläche sieht folgendermaßen aus:

```
<h:commandButton id="save"
  action="#{customer.save}" value="Save"/>
```

Action-Methode:Die referenzierte Methode darf keinen Übergabeparameter haben, muss eine Zeichenkette zurückliefern und zudem mit`public`deklariert werden. Die Methode wird beispielsweise einen Datenbankzugriff ausführen und die Daten des Kunden speichern. Wir stellen diesen Zugriff einfach als Kommentar dar. Schließlich liefert die Methode die Zeichenkette zurück, die wir zuvor direkt in die`action`-Eigenschaft aufgenommen haben, also`/showCustomer.xhtml`:

```
public String save() {
  return "/showCustomer.xhtml";
}
```

Wenn die Speicherung der Kundendaten nicht erfolgreich gewesen ist, sollte eine andere Zeichenkette zurückgeliefert werden. Dadurch wird eine andere Navigation ausgelöst und beispielsweise wieder die Seite`/editCustomer.xhtml`angezeigt.
Im nächsten Kapitel erarbeiten wir gemeinsam die theoretischen Grundlagen zum Verständnis von JSF. Nach einem kurzen Einblick in die Aufgaben von JSF in Abschnitt**Sektion: Aufgaben der JSF-Technologie**und der Definition einiger grundlegender Begriffe in Abschnitt**Sektion: JavaServer Faces in Schlagworten**folgt ein zweiter Teil des Beispiels*MyGourmet 1*in Abschnitt**Sektion: MyGourmet 1: Schlagworte im Einsatz**. Dort wird das Verständnis der zuvor definierten Grundbegriffe im Praxiseinsatz vertieft.

2 Die Konzepte von JavaServer Faces

Auf der Basis des im letzten Kapitel erarbeiteten kleinen Beispiels wenden wir uns jetzt dem zu, was an Theorie hinter dem Quellcode steht. Dazu werden wir uns die einzelnen Bestandteile der*JavaServer Faces*-Technologie ansehen und die Unterstützung von JSF für den Aufbau einer modernen Webapplikation analysieren.

2.1 Aufgaben der JSF-Technologie

Fassen wir noch einmal zusammen, was die JSF-Spezifikation bedeutet - sie definiert ein Framework für die Entwicklung der Benutzerschnittstelle in Java-Webapplikationen. Die Spezifikation dient dazu, den Entwickler in folgenden Bereichen zu unterstützen:

- Komponenten:
  JSF erlaubt es, vollständige Webanwendungen in einfacher Form aus Komponenten aufzubauen. Darüber hinaus kann man Komponenten selbst erstellen und beliebig wiederverwenden.
- Datentransfer:
  JSF macht es sehr einfach möglich, Daten von der Applikation in die Benutzerschnittstelle (und wieder zurück) zu transferieren.
- Zustandsspeicherung:
  JSF ermöglicht die automatische Speicherung des Zustands der Applikation am Server oder am Client.
- Ereignisbehandlung:
  Vom Benutzer am Client generierte Ereignisse können am Server behandelt werden. Dazu werden Ereignisbehandlungsmethoden mit den einzelnen Komponenten verknüpft.

Durch die strikte Trennung der Schichten der Applikation im Sinne der MVC-Architektur können die einzelnen an der Applikation beteiligten Personen (z.B. Webdesigner, Komponentenentwickler und Applikationsentwickler) unabhängig voneinander arbeiten.

2.2 JavaServer Faces in Schlagworten

Wir werden im Verlauf dieses Buches sehr viele Begriffe aus der JSF-Technologie verwenden. Um einen Überblick zu geben, werden wir uns die Definition der wichtigsten Begriffe vorab kurz ansehen.

Komponente (auch`Component`,`UIComponent`oder`Control`)

Eine Komponente ist ein eigenständiger und wiederverwendbarer Baustein, der zusammen mit anderen

Komponenten zum Aufbau einer Seite in einer JSF-Anwendung eingesetzt wird. JSF bietet eine gute Auswahl an vordefinierten Komponenten. Die Palette reicht von einfachen Ausgabekomponenten für Texte oder Bilder über Komponenten zum Erfassen von Benutzereingaben bis hin zu komplexen Komponenten zur Darstellung von Tabellen.

Ansicht und Komponentenbaum

Alle Komponenten einer Seite werden zusammen als Ansicht bezeichnet und sind in Form eines Komponentenbaums miteinander verknüpft. Die Wurzel dieses Baums bildet die`UIViewRoot`-Komponente , alle anderen Komponenten hängen als Kinder und Kinder dieser Kinder unter diesem Element. Alle JSF-bezogenen Vorgänge im Ablauf einer Anfrage an die Anwendung starten mit dem Aufruf einer Methode auf diesem`UIViewRoot`-Element, das den Aufruf rekursiv an seine Kinder weiterreicht.

Renderer

Die eigentliche Ausgabe der Komponente und ihrer Daten und das Entnehmen der vom Benutzer am Client geänderten Daten wird vom`Renderer`erledigt. Eine Komponente kann dabei mit vielen Renderern verbunden werden - je nach Ausgabetechnologie wird ein bestimmter Renderer ausgewählt und so das Aussehen der Komponente verändert. Renderer sind optional - die Komponente kann auch selbst ihre Darstellung bestimmen und führt dann den Ausgabeprozess "selbstständig" durch.

Seitendeklarationssprache (auch View Declaration Language, VDL)

Eine Seitendeklarationssprache (VDL) ist eine Syntax, um Ansichten beziehungsweise Seiten für JSF zu deklarieren. Dieses Konzept wurde in JSF 2.0 im Zuge der Integration von Facelets eingeführt, um von der eingesetzten Technologie zu abstrahieren. Der Standard unterstützt in Version 2.0 mit Facelets und JSP zwei konkrete Implementierungen einer VDL. JSP wird allerdings nur mehr aus Kompatibilitätsgründen unterstützt und bietet lediglich einen Teil der neuen Features.

Validator

Die vom Benutzer eingegebenen Werte müssen nicht immer korrekt sein - beispielsweise könnte der Benutzer einen Wert in einem notwendigen Feld nicht eingegeben oder einen zu langen Wert in einem von der Länge her beschränkten Textfeld eingetragen haben. Für solche Fälle gibt es einfach zu verwendende Validatoren in JSF. Diese Validatoren überprüfen die Gültigkeit der eingegebenen Werte und unterbinden das Zurückschreiben von ungültigen Werten ins Modell.

Konverter

Für Webanwendungen ist es notwendig, die von der Applikationslogik gelieferten Datentypen in eine Zeichenkette zu konvertieren, da der Browser nur Zeichenketten verarbeiten und anzeigen kann. Auch dafür gibt es eine Hilfestellung in der JSF-Technologie, die sich Konverter nennt. Sie konvertiert die von der Geschäftslogik verwendeten Datentypen in Zeichenketten und diese Zeichenketten nach der Veränderung durch den Benutzer wieder zurück in Java-kompatible Datentypen. Wenn ein Fehler in der Konvertierung auftritt, wird wie bei den Validatoren ein Zurückschreiben der Werte ins Modell verhindert.

Managed-Beans (auch*Backing-Beans*genannt)

Hinter den Komponenten liegen Managed-Beans, die die eigentlichen Werte zum Befüllen der Komponenten liefern. Sie werden - wie bereits im ersten Beispiel gezeigt - zentral definiert und können entweder für jeden Benutzer getrennt oder zentral für die gesamte Applikation gültig sein. Managed-Beans sind simple Java-Klassen, auch*POJOs*(Plain Old Java Objects) genannt, die dem*JavaBeans*-Standard genügen müssen.

***Unified Expression Language*(auch*Unified-EL*)**

Die*Unified Expression Language*ist das Bindeglied zwischen den Komponenten einer Ansicht und den dahinterliegenden Managed-Beans. Mit Value-Expressions werden Eigenschaften von Managed-Beans mit Attributen von Komponenten verbunden - und das nicht nur zum Auslesen der Werte einer Bean, sondern auch zum Zurückschreiben von Benutzereingaben. Method-Expressions erlauben das Verknüpfen von Komponenten mit Methoden. Ein Konzept, das zum Beispiel bei der Validierung von Benutzereingaben und der Ereignisbehandlung eingesetzt wird.

Ereignisse und Ereignisbehandlung

Ereignisse sind eines der zentralen Elemente der*JavaServer Faces*-Technologie. Ein Ereignis tritt in einer*JSF*-Anwendung beispielsweise auf, wenn eine Schaltfläche betätigt oder ein Wert geändert wird. Jede Komponente kann Ereignisse auslösen und jede Managed-Bean kann als Interessent für solche Ereignisse registriert werden. Neben den bereits für die Standardkomponenten definierten Ereignissen können neue Ereignisse (für angepasste Komponenten) in die Ereignisbehandlung von JSF aufgenommen werden.

Navigation und Aktionen (`navigation-rules`,`action`)

Die Navigation in einer JSF-Applikation wurde vor JSF 2.0 ausschließlich über sogenannte Navigationsregeln in der`faces-config.xml`definiert. JSF verwendet den Rückgabewert spezieller Methoden, nämlich der "Action"-Methoden (die bei der Behandlung eines*Action*-Ereignisses aufgerufen werden), um eine Weiterleitung von einer auf die nächste Seite zu veranlassen. Dieser Rückgabewert kann der Name einer Navigationsregel oder ab JSF 2.0 direkt der Name der nächsten Seite sein.

Nachrichten

Sollten bei der Abarbeitung von Methoden oder beim Validieren und Konvertieren von Werten Fehler auftreten, müssen diese Fehler dem Benutzer angezeigt werden. Validierungs- und Konvertierungsfehler werden in JSF in Nachrichten umgewandelt, die dann auf der Seite angezeigt werden können.
Das zeitliche Zusammenspiel dieser einzelnen Objekte in der JSF-Technologie ist genau geregelt, und zwar im "Request Processing Lifecycle" genannten Lebenszyklus einer HTTP-Anfrage. Diesen "Lifecycle" werden wir in Abschnitt**[Sektion: Lebenszyklus einer HTTP-Anfrage in JSF]**näher betrachten. Zunächst werden wir aber im folgenden Abschnitt den Zusammenhang zwischen einigen dieser Grundbegriffe anhand des Beispiels*MyGourmet 1*demonstrieren.

2.3 MyGourmet 1: Schlagworte im Einsatz

Im letzten Abschnitt wurden kurz die wichtigsten Grundbegriffe von*JavaServer Faces*erläutert. Nach dieser eher theoretischen Betrachtung wollen wir hier versuchen, diese Schlagworte mit dem bereits bekannten Beispiel*MyGourmet 1*in Verbindung zu bringen. Als Ausgangspunkt ist in Listing**MyGourmet 1: Die Seite editCustomer.xhtml**nochmals der Code der Seitendeklaration`editCustomer.xhtml`abgebildet.

```
<!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:commandButton id="save" action="#{customer.save}"
        value="Save"/>
    </h:panelGrid>
  </h:form>
</body>
</html>
```

Wenn sich ein Benutzer diese Seite im Browser ansehen will, muss er`editCustomer.jsf`in die Adressleiste tippen. Wir haben aber bis jetzt immer XHTML-Dateien mit der Endung`.xhtml`erstellt. Woher kommt dieser Unterschied und warum erscheint im Browserfenster überhaupt eine Ausgabe? Diese Frage ist einfach

beantwortet. Nach außen ist nur die JSF-Ansicht`editCustomer.jsf`sichtbar, die intern auf einer sogenannten Seitendeklaration aufgebaut wird. In "Standard"-JSF ist das eine XHTML-Datei für Facelets oder eine JSP-Datei mit dem gleichen Namen wie die entsprechende JSF-Ansicht. Die JSF-Implementierung weiß, wie der Pfad der zugehörigen Seitendeklaration relativ zum Kontext der Webapplikation aussieht - in unserem Fall`/editCustomer.xhtml`. Dieser Pfad der Seitendeklaration wird auch*View-Identifier*genannt.
Die Seitendeklaration bestimmt den Inhalt und die Struktur des Komponentenbaums und somit auch die Ansicht. Die Tags der XHTML-Seite werden von Facelets in Komponenten umgesetzt und im Komponentenbaum angeordnet. Abbildung**MyGourmet 1: Von der Seitendeklaration zur Komponente**zeigt diese Umsetzung exemplarisch am Tag der Eingabekomponente für den Vornamen.

```
<h:inputText id="firstName" value="#{customer.firstName}"/>
```

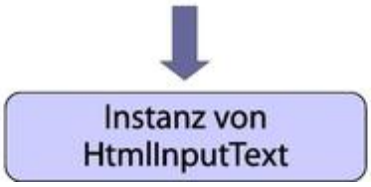


Abbildung:MyGourmet 1: Von der Seitendeklaration zur Komponente

Auf die gleiche Weise werden auch alle anderen Tags der Seite in Komponenten umgesetzt und in den Baum eingefügt. Der fertige Komponentenbaum der Ansicht`editCustomer.jsf`sieht dann in etwa wie in Abbildung**MyGourmet 1: Komponentenbaum**aus. Die Namen der Knoten in der Abbildung sind die Namen der eingesetzten Komponentenklassen.

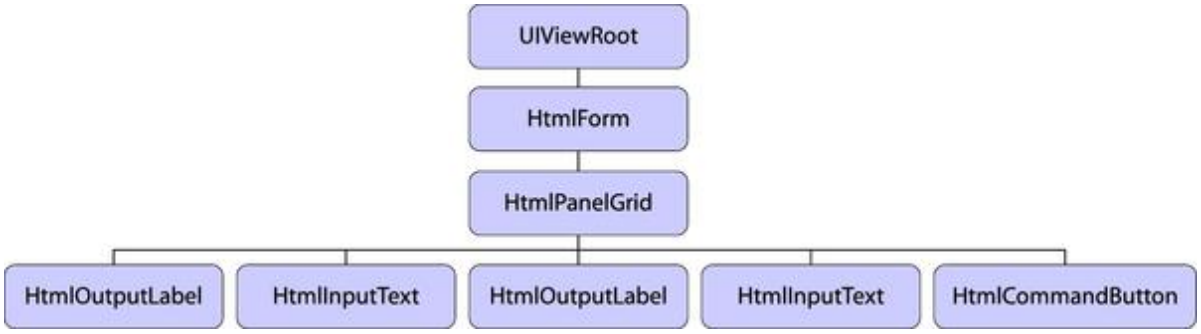
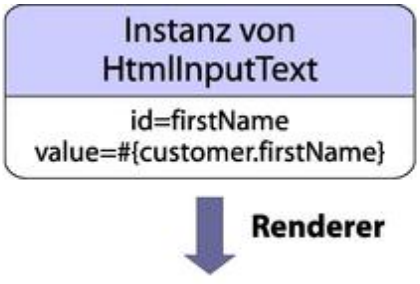


Abbildung:MyGourmet 1: Komponentenbaum

Der fertig aufgebaute Komponentenbaum kann jetzt von einem Renderer in eine Ausgabesprache umgesetzt und dem Benutzer angezeigt werden. In den meisten Fällen wird es sich bei der Ausgabe um HTML-Seiten handeln, durch das Austauschen des Renderers kann aber fast jede beliebige Ausgabetechnologie eingesetzt werden. Wir wollen uns allerdings für unser Beispiel momentan auf HTML beschränken. Der Renderer nimmt also die in der Komponenteninstanz gespeicherten Daten und gibt den entsprechenden HTML-Code für die jeweilige Komponente aus. Abbildung**MyGourmet 1: Von der Komponente zur HTML-Ausgabe**zeigt zum Beispiel, wie die Eingabekomponente für den Vornamen in HTML umgesetzt wird.




```
<input type="text" id="form:firstName" value="Michael"/>
```

Abbildung:MyGourmet 1: Von der Komponente zur HTML-Ausgabe

Die Darstellung der komplett gerenderten Seite im Browser ist in Abbildung**MyGourmet 1: editCustomer.xhtml im Browser**zu sehen.

Abbildung:MyGourmet 1: editCustomer.xhtml im Browser

Die Zusammenhänge zwischen Seitendeklaration, Komponentenbaum und gerenderter Ausgabe dürften somit geklärt sein. Den konkreten Ablauf des gesamten Prozesses in Form des JSF-Lebenszyklus haben wir allerdings noch außen vor gelassen. Diese zeitlichen Zusammenhänge und Abläufe werden in Abschnitt**[Sektion: Lebenszyklus einer HTTP-Anfrage in JSF]**behandelt. Zuvor werfen wir jedoch in Abschnitt**[Sektion: Managed-Beans]**noch einen genaueren Blick auf Managed-Beans und in Abschnitt**[Sektion: Die*Unified Expression Language*]**auf die Verbindung zwischen Modell und Ansicht mit der*Unified Expression Language*.

2.4 Managed-Beans

Die Managed-Beans sind ein zentraler Bestandteil der*JavaServer Faces*. Sie bilden in einer Anwendung das Modell beziehungsweise die Verbindung zum Modell und der Geschäftslogik. Im Hinblick auf eine strikte Trennung von Präsentation und Logik fällt ihnen damit eine sehr wichtige Rolle zu. In der Praxis sind die Aufrufe der Geschäftslogik in einer Anwendung komplett in den Managed-Beans gekapselt. Die Verbindung zu den Eigenschaften und Methoden einer Managed-Bean wird mit*Unified-Expression*-Ausdrücken realisiert.
Der Rest dieses Abschnitts geht auf die Grundlagen und Details von Managed-Beans ein. Detailliertere Informationen zur*Unified Expression Language*, dem Bindeglied zwischen Ansicht und Modell in JSF, finden sich in Abschnitt**[Sektion: Die*Unified Expression Language*]**.

2.4.1 Managed-Beans - die Grundlagen

Wie muss eine Managed-Bean in JSF aussehen, damit sie eingesetzt werden kann? Die Anforderungen sind minimal: Managed-Beans sind simple Java-Klassen, auch*POJOs*(Plain Old Java Objects) genannt, die dem*JavaBeans*-Standard genügen müssen. Für die Klasse an sich bedeutet das nur, dass sie einen Konstruktor ohne Parameter mit Sichtbarkeit`public`haben muss.
Wie bereits erwähnt, wird in JSF auf die Eigenschaften der Managed-Beans zugegriffen, um Daten zu lesen und zu schreiben. Eine Eigenschaft hat einen Namen, einen Typ und Methoden zum Lesen und Schreiben des Werts. Die Namen dieser Methoden müssen folgender Konvention entsprechen:`getEigenschaftsName`ist der Name der Methode für den lesenden Zugriff und`setEigenschaftsName`der Name der Methode für den schreibenden Zugriff. Ob in der Methode auf eine private Variable der Klasse zugegriffen wird oder ob eine komplexe Operation der Geschäftslogik dahinter liegt, ist transparent und letztendlich egal. Nach außen ist in beiden Fällen nur die Eigenschaft der Bean sichtbar.
Listing**Managed-Bean-Klasse Customer aus MyGourmet 1**zeigt noch einmal die Klasse`Customer`aus unserem Beispiel*MyGourmet 1*. Diese Bean hat die Eigenschaften`firstName`und`lastName`vom Typ`String`, die durch

die jeweiligen Getter- und Setter-Methoden definiert sind. Der Zugriff in der Ansicht erfolgt mit den Ausdrücken#{customer.firstName}und#{customer.lastName}. Je nachdem, ob der Wert in der zugreifenden Komponente gelesen oder geschrieben wird, kommt der entsprechende Getter oder Setter zum Einsatz. Die Namen der Eigenschaften leiten sich von den Namen der Getter- und Setter-Methoden ab. Die privaten Felder im Hintergrund haben keinerlei Einfluss auf den Namen - sie nehmen nur den Wert der Eigenschaft auf.
Die Methodesaveerfüllt einen anderen Zweck. Sie wird zur Behandlung von Ereignissen benutzt und ist keiner Eigenschaft zugeordnet - detailliertere Informationen zu diesem Thema finden sich in Abschnitt**[Sektion: Ereignisse und Ereignisbehandlung]**.

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class Customer {
  private String firstName;
  private String lastName;
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String save() {
    return "/showCustomer.xhtml";
  }
}
```

Eigenschaften von Beans müssen nicht unbedingt sowohl eine Getter- als auch eine Setter-Methode haben. Je nachdem, welche der beiden vorhanden ist, handelt es sich dann um eine Eigenschaft, die nur gelesen oder nur geschrieben werden kann.
Genau genommen haben wir in den letzten Absätzen lediglich von*JavaBeans*gesprochen. Zu Managed-Beans werden sie erst, wenn sie tatsächlich von JSF verwaltet werden. Wie das funktioniert, zeigt der nächste Abschnitt.

2.4.2 Konfiguration von Managed-Beans

Einer der Eckpfeiler von JSF ist die zentrale Stelle für die Behandlung von Managed-Beans - die*Managed Bean Creation Facility*. Mit diesem Instrument werden folgende Aufgaben durchgeführt:

- Deklaration sämtlicher Managed-Beans
- Festlegung der Lebensdauer der Managed-Beans
- Automatische Erzeugung, Initialisierung, Verwendung und Löschung der Managed-Bean-Instanzen
- Bereitstellung der Managed-Beans über die Expression Language (EL). Über Value-Expressions und Method-Expressions kann zum Beispiel auf Geschäftsobjekte referenziert werden.

Damit Managed-Beans in der Ansicht verwendet werden können, muss die JSF-Umgebung wissen, unter

welchem Namen und unter welcher Klasse die jeweilige JavaBean zu finden ist. Seit JSF 2.0 gibt es zwei Varianten, um diese Registrierung durchzuführen. Im Einführungsbeispiel haben wir ja bereits gesehen, wie eine Managed-Bean über Annotationen deklariert wird. Listing**Konfiguration der Bean customer aus MyGourmet 1 über Annotationen**zeigt nochmals den relevanten Teil der Klasse. Alternativ können Managed-Beans auch in der`faces-config.xml`deklariert werden - in JSF-Versionen vor 2.0 war das noch die einzige Möglichkeit.

```
@ManagedBean
@SessionScoped
public class Customer {
  ...
}
```

Listing**Konfiguration der Bean customer aus MyGourmet 1 in der faces-config.xml**zeigt, wie die Konfiguration der Bean`customer`aus*MyGourmet 1*in der Datei`faces-config.xml`aussieht.

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>
    at.irian.jsfatwork.gui.page.Customer
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Die beiden in den Listings**Konfiguration der Bean customer aus MyGourmet 1 über Annotationen**und**Konfiguration der Bean customer aus MyGourmet 1 in der faces-config.xml**vorgestellten Varianten führen zu demselben Ergebnis: Die Managed-Bean mit dem Namen`customer`wird definiert. Welche der beiden Sie in Ihrem Projekt einsetzen, ist zum Teil auch eine Geschmacksfrage. Wir haben uns in*MyGourmet*für Annotationen entschieden, um aufgeblähte Konfigurationsdateien zu vermeiden.
In der`faces-config.xml`erfolgt die Deklaration einer Bean in einem Element namens`managed-bean`. Darin verschachtelt folgt im Element`managed-bean-name`zuerst der Name, unter dem die Bean in EL-Ausdrücken referenziert wird - in unserem Beispiel`customer`. Die Klasse der Bean wird im Element`managed-bean-class`festgelegt. Zuletzt folgt im Element`managed-bean-scope`mit`session`die Angabe der Lebensdauer der Bean.
Bei der Deklaration mit der Annotation`@ManagedBean`entspricht der Name der Managed-Bean laut Konvention dem Klassennamen mit einem kleinen Anfangsbuchstaben. In unserem Fall wird zum Beispiel aus der Klasse`Customer`die Bean`customer`. Wollen Sie einen anderen Namen verwenden, können Sie diesen im Element`name`der Annotation setzen. Listing**Konfiguration der Bean customer mit alternativem Namen**zeigt die bereits bekannte Bean mit dem expliziten Namen`customerBean`. Der Zugriff in der Ansicht erfolgt jetzt mit dem Ausdruck`#{customerBean.firstName}`.

```
@ManagedBean(name = "customerBean")
@SessionScoped
public class Customer {
  ...
}
```

In Standard-JSF sind folgende Gültigkeitsbereiche (Scopes) für Beans definiert (in Klammer steht der Wert für die Konfiguration in der`faces-config.xml`und die entsprechende Annotation):

- *None-Scope*(`none`,`@NoneScoped`):
  Die Managed-Bean wird bei jedem Aufruf neu erstellt.
- *Request-Scope*(`request`,`@RequestScoped`):
  Die Managed-Bean lebt für die Zeitdauer einer HTTP-Anfrage.
- *View-Scope*(`view`,`@ViewScoped`):
  Die Lebensdauer der Managed-Bean ist an die Ansicht geknüpft, in der sie verwendet wird.

- *Session-Scope*(`session`,`@SessionScoped`):
  Die Managed-Bean lebt für die Dauer einer Sitzung, in der der Benutzer mit der Anwendung verbunden ist.
- *Application-Scope*(`application`,`@ApplicationScoped`):
  Für die gesamte Lebensdauer der Anwendung ist nur eine für alle Benutzer gleiche Instanz dieser Managed-Bean vorhanden.

Abbildung**Vergleich der Lebensdauer unterschiedlicher Gültigkeitsbereiche**vergleicht die Lebensdauer der von JSF standardmäßig zur Verfügung gestellten Gültigkeitsbereiche.

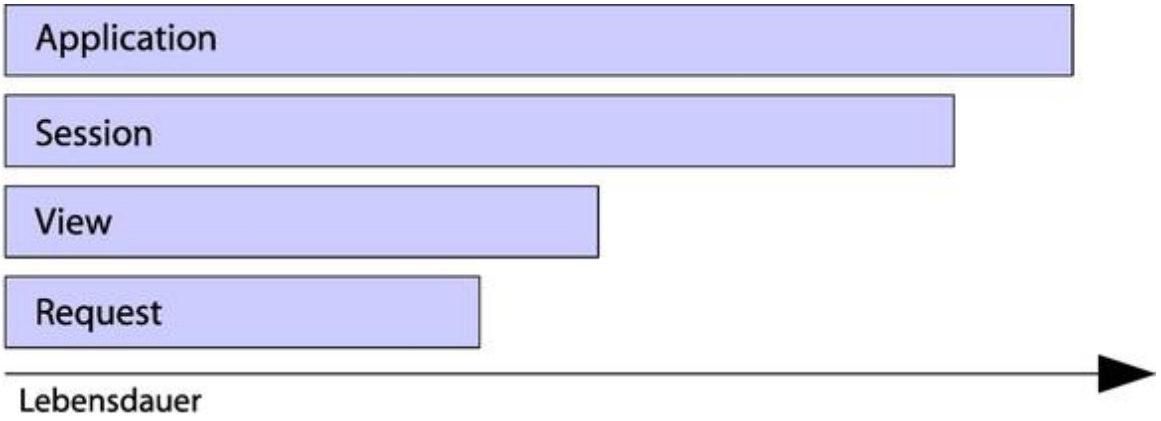


Abbildung:Vergleich der Lebensdauer unterschiedlicher Gültigkeitsbereiche

Nachdem Sie jetzt wissen, wie Managed-Beans deklariert werden, wollen wir Ihnen nicht vorenthalten, wie JSF diese verwaltet. Der interne Ablauf beim Zugriff auf eine Bean sieht wie folgt aus:

1. Beim ersten Zugriff auf die Bean wird diese automatisch durch die*Managed Bean Creation Facility*instanziiert. Ist die Instanz der Bean bereits vorhanden, wird sie zurückgegeben. Die Instanziierung kann nur erfolgen, wenn ein Konstruktor ohne Argumente verfügbar ist.
2. Nach dem Erzeugen der Bean werden alle Managed-Properties ini-tialisiert. Genaueres dazu finden Sie in Abschnitt**[Sektion: Managed-Properties]**.
3. Zu guter Letzt wird die Managed-Bean unter der spezifizierten Lebensdauer gespeichert.

Das erfolgt in unserem Beispiel im Session-Scope, also solange eine logische Verbindung zwischen Benutzer und Applikation in Form einer Sitzung besteht. Verwenden Sie den Session-Scope nur, wenn unbedingt notwendig. Mit dem neuen View-Scope ist es jetzt relativ einfach möglich, Daten über mehrere Requests mitzunehmen. Zumindest so lange, bis auf eine neue Seite navigiert wird.

2.4.3 Managed-Properties

Die*Managed Bean Creation Facility*bietet die Möglichkeit, Eigenschaften von Managed-Beans nach dem Erstellen zu initialisieren (sogenannte Managed-Properties). Neben fixen Werten besteht über Dependency-Injection auch die Möglichkeit, Abhängigkeiten auf andere Beans für die Initialisierung zu verwenden.
Wie bei der Konfiguration der Beans selbst existieren auch für die Deklaration der Managed-Properties zwei Varianten. Mit der Annotation`@ManagedProperty`werden direkt die Felder der Eigenschaft in der Bean annotiert. Der initiale Wert steht bei dieser Methode im Element`value`. Alternativ kann dieselbe Deklaration auch in der`faces-config.xml`gemacht werden. Dazu kommt das Element`managed-property`zum Einsatz, wobei der Name der Eigenschaft und der zu setzende Wert in den Elementen`property-name`und`value`stehen.
Listing**Managed-Properties mittels Annotationen**zeigt den Java-Code einer annotierten Managed-Bean mit Managed-Properties. In Listing**Managed-Properties in der Konfiguration**ist die äquivalente Konfiguration in der`faces-config.xml`zu sehen. Die Annotationen in der Klasse`Login`sind in diesem Fall natürlich nicht notwendig.

```
@ManagedBean
@SessionScoped
```

```
public class Login {
  @ManagedProperty(value = "3")
  private int loginRetries;
  @ManagedProperty(
    value = "#{roleResolver.defaultResolver}")
  private RoleResolver roleResolver;
  ...
}
```

```
<faces-config>
  ...
  <managed-bean>
    <managed-bean-name>login</managed-bean-name>
    <managed-bean-class>
        at.company.webapp.model.Login
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>loginRetries</property-name>
      <value>3</value>
    </managed-property>
    <managed-property>
      <property-name>roleResolver</property-name>
      <value>#{roleResolver.defaultResolver}</value>
    </managed-property>
  </managed-bean>
  ...
</faces-config>
```

Nach der Initialisierung der Bean`login`besitzt deren Attribut`loginRetries`den Wert 3. Das Attribut`roleResolver`zeigt, dass als Wert auch eine Referenz auf eine andere Managed-Bean angeführt sein kann - über eine Value-Expression. Alle anderen Attribute besitzen ihre Standardwerte.
Der Einsatz von Managed-Beans zum Initialisieren von Managed-Properties unterliegt in JSF einer Einschränkung. Eine Managed-Property darf nicht mit einer Managed-Bean mit kürzerer Lebensdauer initialisiert werden. Es ist zum Beispiel nicht erlaubt, eine Bean im Request-Scope in eine Bean im Session-Scope zu injizieren. Die Erklärung dafür ist einfach: Da eine Session länger läuft als eine Anfrage, ist die injizierte Bean nach der ersten Anfrage nicht mehr aktuell. Beans im None-Scope können dagegen immer verwendet werden, da sie in keinem Gültigkeitsbereich abgelegt sind.
Die Initialisierung mit Referenzen über Dependency-Injection bietet einige Vorteile gegenüber der Auflösung von*Beans*im Code. Zum einen lassen sich damit statische Aufrufe im Code vermeiden und zum anderen steigen Wartbarkeit und Übersichtlichkeit durch die zentrale Konfiguration.
Die*Managed Bean Creation Facility*ist die von JSF zur Verfügung gestellte Möglichkeit zur Erstellung von Beans - aber bei Weitem nicht die einzig mögliche. Mit CDI und*Spring*gibt es Alternativen, die in Konfigurationsumfang und Erweiterbarkeit deutlich überlegen sind, ohne jedoch die Komplexität der Anwendung unnötig zu erhöhen. In Abschnitt**Sektion: Beans und Dependency-Injection mit CDI**zeigen wir, wie Managed-Beans mit CDI verwaltet werden und welche Vorteile sich daraus ergeben.

2.4.4 Die Rolle von Managed-Beans

Bis jetzt sind wir davon ausgegangen, dass die Managed-Beans das Modell der Anwendung laut MVC-Entwurfsmuster bilden. Das muss nicht unbedingt so sein - in den meisten Fällen ist es sogar besser, wenn die

Managed-Beans nicht direkt das Modell sind, sondern nur eine Vermittlerrolle zwischen Ansicht und tatsächlichem Modell einnehmen.
Wie kann man sich das vorstellen? Ein einfaches Beispiel, basierend auf*MyGourmet 1*, hilft, diesen Sachverhalt zu klären. In*MyGourmet 1*bildet die Managed-Bean das Modell der Anwendung, ein Zugriff auf den Vornamen des Kunden sieht folgendermaßen aus:`#{customer.firstName}`. Neben den Modelleigenschaften des Kunden beinhaltet die Bean hier auch die Action-Methoden zur Ereignisbehandlung. Der Nachteil dieser Variante ist die enge Kopplung von GUI-Logik und Modell, der sich besonders dann negativ auswirkt, wenn es zu Änderungen kommt.
Eine elegantere Lösung ist, die Modellklasse`Customer`komplett von JSF-Code freizuhalten und eine Managed-Bean`customerBean`einzuführen, die neben der GUI-Logik eine Eigenschaft vom Typ`Customer`besitzt. Ein Zugriff auf den Vornamen des Kunden sieht dann folgendermaßen aus:`#{customerBean.customer.firstName}`. Die Modellklasse kann in dieser Variante unabhängig von der Präsentationsschicht in einer tiefer liegenden Schicht der Anwendung erstellt werden. Die beiden Varianten sind in Abbildung**Die Rolle von Managed-Beans**dargestellt.

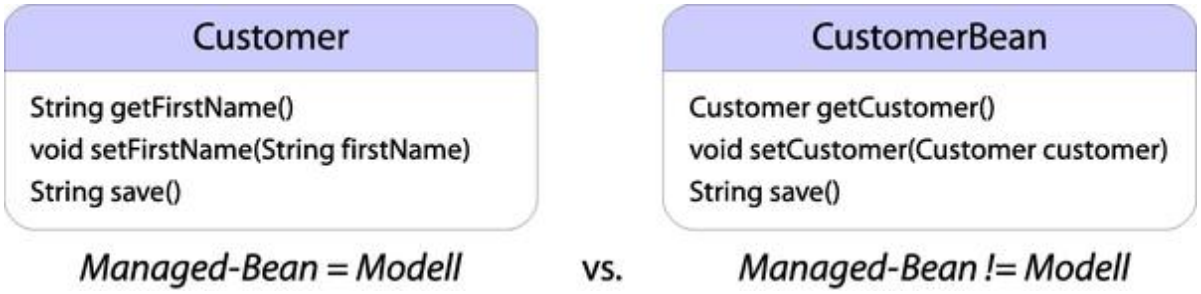


Abbildung:Die Rolle von Managed-Beans

In*MyGourmet*werden wir aus Gründen der Einfachheit vorerst dabei bleiben, die Klasse`Customer`direkt als Managed-Bean zu verwenden. Erst bei den etwas umfangreicheren Beispielen ab*MyGourmet 5*in Abschnitt**[Sektion: *MyGourmet 5*: Konvertierung]**kommt eine eigene Klasse zum Einsatz.

2.5 Die Unified Expression Language

Ein Basiselement der JSF-Spezifikation ist die*Unified Expression Language*(kurz*Unified-EL)*, die es ermöglicht, Komponenten der Benutzerschnittstelle und der Geschäftsdaten dahinter sehr dynamisch zu verbinden. Wir wollen ja Daten aus dem Modell lesen, aber auch Benutzereingaben ins Modell zurückschreiben. Des Weiteren muss definiert werden, welche Methoden welche Ereignisse behandeln. Mithilfe von Value-Expressions werden Komponentenattribute an Managed-Beans und ihre Eigenschaften gebunden und mit Method-Expressions werden Methoden referenziert. Wie das in*MyGourmet 1*erfolgt, sehen wir uns in Abschnitt**[Sektion: *Unified-EL*in*MyGourmet 1*]**etwas genauer an.
Für die Definition eines EL-Ausdrucks ist eine Raute und eine geschwungene Klammer dem Ausdruck voran- und eine geschwungene Klammer dem Ausdruck nachzustellen. Was darf zwischen diesen Begrenzern stehen? Zwischen diesen Begrenzern können der Name von Managed-Beans oder "impliziten" Objekten, Eigenschaften dieser Objekte (von den Elternelementen mit Punkten getrennt) oder auch Operatoren angegeben werden. Dazu zählen sowohl arithmetische Operatoren wie "+" und "-" als auch Vergleichsoperatoren, sogar der "ternäre" Operator (`bedingung ? wenn_wahr : wenn_falsch`) ist erlaubt.
Es lassen sich aber nicht nur Eigenschaften mit der*Unified Expression Language*auslesen, auch Methoden können mit dieser vereinheitlichten "Ausdruckssprache" aufgerufen werden - über die Angabe von Method-Expressions. Beispielsweise werden mit Konstrukten der*Unified Expression Language*Ereignisbehandlungsmethoden aufgerufen.

2.5.1 Unified-EL

in MyGourmet 1

Im Beispiel*MyGourmet 1*haben wir bereits ausführlich von Value-Expressions Gebrauch gemacht, um Kundendaten anzuzeigen und Benutzereingaben abzuspeichern. Nehmen wir den Vornamen des Kunden als Beispiel: Mit dem Ausdruck`#{customer.firstName}`wird das Attribut`value`der Eingabekomponente mit der Eigenschaft`firstName`der Managed-Bean`customer`verbunden. In Abbildung**Value-Expression in Eingabekomponente**ist der Zusammenhang zwischen Tag, Komponente und Managed-Bean noch genauer dargestellt.

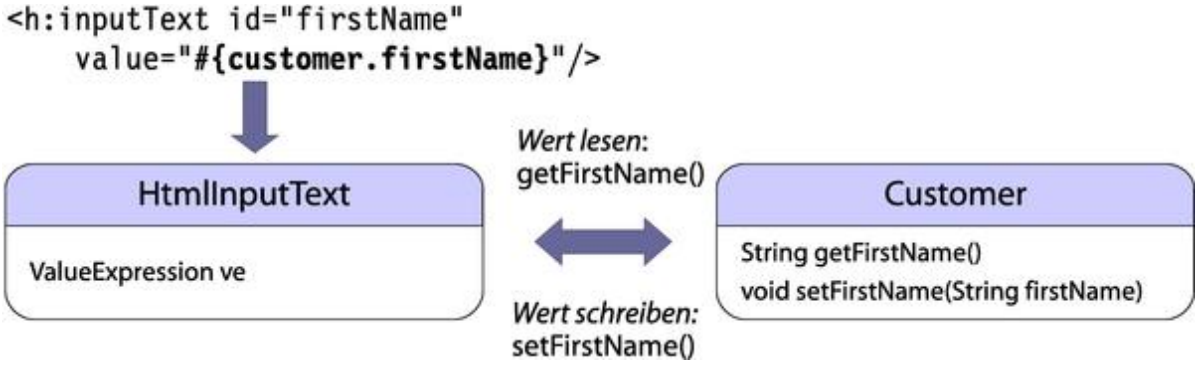


Abbildung:Value-Expression in Eingabekomponente

Eine Value-Expression wird erst beim tatsächlichen Lesen oder Setzen des Werts aufgelöst. Deswegen wird in der Komponente auch die Value-Expression selbst und nicht der Wert abgelegt. Eine genauere Betrachtung, warum das notwendig ist, folgt in Abschnitt**[Sektion: Lebenszyklus einer HTTP-Anfrage in JSF]**über den Lebenszyklus von JSF.
Die Daten werden beim Anzeigen der Seite bereits ordnungsgemäß gelesen und beim Abschicken des Formulars geschrieben. Es fehlt die Möglichkeit, benutzerdefinierten Code beim Bearbeiten der abgeschickten Seite aufzurufen - zum Beispiel um die Eingaben des Benutzers in der Datenbank zu speichern. Hier kommen Method-Expressions ins Spiel. Hat der Benutzer auf der Seite`editCustomer.jsf`die Schaltfläche zum Speichern betätigt, soll serverseitig die Methode`save`der Bean aufgerufen werden. Dazu gibt es bei Befehlskomponenten das Attribut`action`, das eine Method-Expression enthalten kann. In unserem Beispiel ist das der Ausdruck`#{customer.save}`. Die damit referenzierte Methode wird aufgerufen, wenn alle vom Benutzer eingegebenen Daten validiert wurden und gültig sind. Die zurückgelieferte Zeichenkette wird zur Navigation benutzt und bestimmt, welche Seite angezeigt wird. Abbildung**Method-Expression in Befehlskomponente**zeigt das Tag der Komponente und den Weg bis zum Aufruf der mit der Komponente verbundenen Methode.

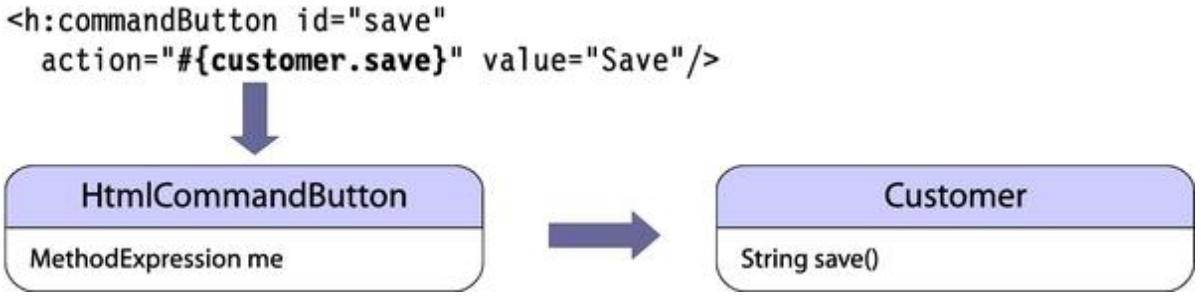


Abbildung:Method-Expression in Befehlskomponente

Ausführlichere Informationen zur*Unified-EL*finden sich im nächsten Abschnitt. Der Navigationsvorgang wird in Abschnitt**[Sektion: Navigation]**näher beleuchtet.

2.5.2 Die Unified-EL im Detail

Im Folgenden finden Sie einen kurzen Überblick über die Möglichkeiten, die die*Unified-EL*bietet.

- `value="#{user.username}"`
  Bindet ein Komponentenattribut an die Eigenschaft`username`eder Managed-Bean mit dem Namen`user`. Das bedeutet, dass beim Rendern der Komponente der Wert über den Aufruf von`getUser-name()`aus der Bean geholt wird und dass in der Aktualisierungsphase der Wert über den Aufruf von`setUsername()`wieder in die Bean zurückgeschrieben wird.
- `rendered="#{user.username != null}"`
  Bindet einen booleschen Wert (`true`oder`false`) an das Attribut der Komponente. Dies kann sehr gut dazu verwendet werden, um durch das Setzen des`rendered`-Attributs die Komponente ein- oder auszublendenUm Probleme zu vermeiden, ist das Verwenden des`rendered`-Attributs dem Einsatz von JSTL-`<c:if/>`-Tags vorzuziehen.:. Bei einem solchen Ausdruck wird keine Aktualisierung des Werts vorgenommen (es gibt ja keinen Setter) - daher kann ein zusammengesetzter Ausdruck auch nicht für das`value`-Attribut von JSF-Komponenten benutzt werden.
- `value="#{bill.sum * 13,7603}"`
  Wenn der Wert einer`h:outputText`-Komponente mit diesem Ausdruck versehen ist, wird in der Komponente immer der aktuelle Wert der Eigenschaft`sum`der Managed-Bean`bill`stehen, multipliziert mit 13,7603. Diese Form der EL-Ausdrücke sollte allerdings nur spärlich eingesetzt werden. Berechnungen jeglicher Art sind in der Geschäftslogik besser aufgehoben.
- `style="#{grid.displayed ? 'display:inline;' : 'display:none;'}"`
  Hier wird das Attribut`style`einer Komponente entweder auf`display:inline;`oder`display:none;`gesetzt, ein sehr häufig verwendeter Trick, um Bereiche einer Seite am Client für gewisse Attributwerte aus- oder einzublenden.
- `value=" Hallo Benutzer #{user.username}"`
  Auch die Kombination von Zeichenketten und*Unified-EL*-Ausdrücken ist möglich. Auf diese Weise lassen sich sehr einfach dynamische Ausdrücke erzeugen. Diese Lösung hat allerdings einen gravierenden Nachteil: Bei der Kombination mit der Zeichenkette wird keine Konvertierung des Werts mehr durchgeführt, sondern einfach`toString()`aufgerufen.
- `action="#{user.storeUser}"`
  Hier wird die Aktion`storeUser`über eine Method-Expression aufgerufen. Die referenzierte Methode muss die Signatur`String storeUser()`aufweisen und als Rückgabewert die Zeichenkette liefern, die die Navigation zur nächsten Seite bestimmt.
- `value="#{mapBean['index']}"`
  Für den Zugriff auf Werte in Objekten, die das Interface Map implementieren, kann der Schlüssel in eckigen Klammern angegeben werden. Diese Notation ist prinzipiell äquivalent zur Punktnotation, es kann also auch mit der Punktnotation auf die Inhalte einer Map referenziert werden. Umgekehrt ist es möglich, mit der Notation der eckigen Klammern auch auf Eigenschaften normaler Beans zuzugreifen. Kleine Denkaufgabe: Warum kann die Notation mit eckigen Klammern in gewissen Situationen von Vorteil sein?
- `value="#{mapBean[user.username]}"`
  Hier die Auflösung der Aufgabe: Value-Expressions kann man auch schachteln und dabei lässt sich dann der Schlüssel für den Zugriff auf die Map oder der Eigenschaftsname für den Zugriff auf die Bean-Eigenschaft über eine Value-Expression angeben.
- `value="#{listBean[5]}"`
  Ein letztes Beispiel: Hier referenziert der Wert der Komponente auf den sechsten Eintrag einer Liste. Auch dieser Index kann wieder eine Value-Expression sein, so wie im obigen Beispiel.

Implizite Objekte:Als Basis für die Auflösung der*EL*-Ausdrücke kann jede Managed-Bean dienen oder eine Liste von impliziten Objekten, die von JSF zur Verfügung gestellt werden. Die folgende Liste enthält die wichtigsten impliziten Objekte:

- `requestScope`
  Zugriff auf die*Request-Map*des External-Contexts.
- `viewScope`
  Zugriff auf die*View-Map*des*View-Roots*.
- `sessionScope`
  Zugriff auf die*Session-Map*des External-Contexts.
- `applicationScope`
  Zugriff auf die*Application-Map*des External-Contexts.
- `view`
  Zugriff auf den*View-Root*.
- `param`
  Zugriff auf die*Request-Parameter-Map*des External-Contexts.
- `paramValues`
  Zugriff auf die*Request-Parameter-Values-Map*des External-Contexts (verhält sich gleich wie die Request-Parameter-Map, nur wird hier ein*String-Array*zurückgegeben).
- `header`
  Zugriff auf die Request-Header-Map des External-Contexts.
- `headerValues`
  Zugriff auf die Request-Header-Values-Map des External-Contexts (verhält sich gleich wie die Request-Header-Map, nur wird hier ein*String-Array*zurückgegeben).
- `facesContext`
  Zugriff auf den Faces-Context
- `initParam`
  Zugriff auf die*Init-Parameter-Map*des External-Contexts und damit auf Kontextparameter der Webapplikation.
- `cookie`
  Zugriff auf die Request-Cookie-Map des External-Contexts.

Beispiel:Ein häufig verwendetes Beispiel für implizite Objekte: Mit dem Ausdruck`#{param.myParam}`kann auf den*Request-Parameter*namens`myParam`zugegriffen werden.

2.5.3 Erweiterungen der Unified-EL in Java EE 6

*Java EE 6*bringt eine neue Version der*Unified-EL*mit einigen lang erwarteten Neuerungen. Mit der neuen Version können endlich in EL-Ausdrücken Methoden mit Parametern verwendet werden. Bislang war das nur für statische Methoden über EL-Funktionen (siehe Abschnitt**Sektion: Definition einer EL-Funktion**) möglich.
Der Einsatz von Method-Expressions mit Parametern eröffnet eine Reihe interessanter Möglichkeiten in JSF. Im Beispiel*MyGourmet 9*in Abschnitt**Sektion: MyGourmet 9: UIData und Detailansicht**sehen Sie, wie eine Action-Methode mit Parametern zum Löschen einer Adresse aus einer Liste eingesetzt werden kann.
Aber das ist noch nicht alles. Als weitere Neuerung sind Value-Expressions nicht mehr auf Eigenschaften von

Beans beschränkt. Die neue*Unified-EL*erlaubt den Aufruf einer beliebigen Methode, deren Rückgabewert den Wert der Value-Expression bildet. Damit sind folgende*Unified-EL*-Ausdrücke möglich:

- `value="#{bean.list.size()}"`
  Mit diesem Ausdruck ist es endlich möglich, die Anzahl der Elemente einer Liste ohne Umwege auszulesen.
- `value="#{bean.text.replaceAll(':', '_')}"`
  Dieser Ausdruck ruft auf der Eigenschaft`text`vom Typ`String`die Methode`replaceAll()`auf, um alle Doppelpunkte durch Unterstriche zu ersetzen, und liefert das Ergebnis zurück.
- `value="#{bean.findOrders(otherBean.customer)}"`
  Der Wert dieses Ausdrucks wird über einen Aufruf der Methode`findOrders()`auf der Bean`bean`bestimmt. Als Parameter kommt dabei die Eigenschaft`customer`der Bean`otherBean`zum Einsatz - auch das ist ohne Probleme möglich.
- `value="#{bean.getName()}"`
  Diese Value-Expression bindet den Rückgabewert der Methode`getName()`an die Komponente. Im Gegensatz zum Ausdruck kann mit`#{bean.getName()}`nur gelesen werden - auch wenn die Methode`setName()`existiert - und ist daher nicht für Eingabefelder geeignet.

Die neue Version der*Unified-EL*ist ein Teil von*Java EE 6*und kommt automatisch mit allen Servern, die Servlet 3.0 und JSP 2.2 unterstützen (dazu zählen zum Beispiel Tomcat 7 oder Jetty 8). Falls Sie einen älteren Server einsetzen, müssen Sie trotzdem nicht auf die wichtigsten Features der neuen*Unified-EL*verzichten. Abschnitt**subsec Konfiguration der Unified-EL**zeigt die Verwendung der alternativen EL-Implementierung von*JBoss*.

2.6 Lebenszyklus einer HTTP-Anfrage in JSF

Aus einer HTTP-Anfrage heraus müssen in JSF einige Verarbeitungsschritte durchgeführt werden, um den Zustand der Applikation wiederherzustellen und die Vorbereitungen dafür zu treffen, dass die Applikationslogik aufgerufen werden kann. Diese Schritte betreffen einmal die Wiederherstellung des Komponentenbaums, das Auslesen der vom Benutzer veränderten Daten aus der HTTP-Anfrage und die Validierung dieser Daten sowie das Übertragen dieser validierten Daten in die Modell-objekte. Nach dem Aufruf von Aktionen in der Geschäftslogik bleibt dann noch der letzte Schritt - die Ausgabe der Antwort auf die HTTP-Anfrage (das sogenannte "Rendering"). Dieser Ablauf ist in der*JavaServer Faces*-Spezifikation genau definiert und bildet eine essenzielle Grundlage für jede auf der JSF-Technologie basierende Anwendung. In Abbildung**Der Lebenszyklus ("`Lifecycle"') einer HTTP-Anfrage**ist der Ablauf dargestellt.

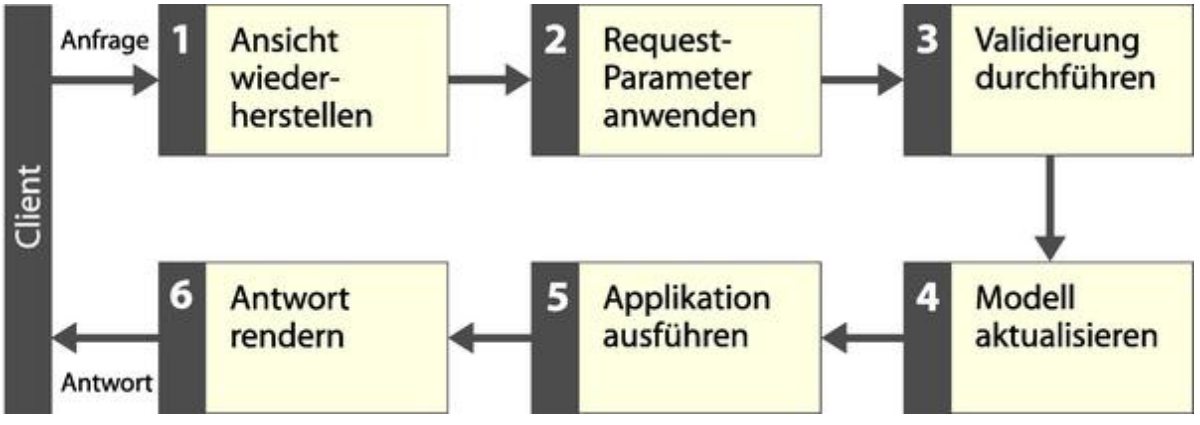


Abbildung:Der Lebenszyklus ("`Lifecycle"') einer HTTP-Anfrage

Wir werden nun die einzelnen Verarbeitungsschritte genauer betrachten. Um das Ganze spannender zu gestalten, werden wir auch gleich auf die praktische Verwertbarkeit der Vorgänge in den einzelnen Phasen eingehen.

Phase 1: Ansicht wiederherstellen (*Restore View*)

Jede Ansicht einer JSF-Anwendung besteht aus Komponenten, die in Form eines Komponentenbaums organisiert sind. Die Abarbeitung einer Anfrage beginnt in der ersten Phase des Lebenszyklus mit dem Aufbau des Komponentenbaums.
Trifft die erste Anfrage auf eine Ansicht ein, existiert der Komponentenbaum noch nicht und JSF baut ihn aus der Seitendeklaration neu auf. Kommt als Seitendeklarationssprache JSP zum Einsatz, leitet JSF die Anfrage an die hinter der Ansicht liegende JSP-Seite weiter. Diese wird abgearbeitet und bei jedem Antreffen eines neuen, noch nicht zu einer initialisierten Komponente gehörenden Tags wird eine neue Komponente erzeugt und mit den Attributwerten aus der JSP-Seite initialisiert. Facelets verfolgt eine ganz ähnliche Strategie und baut den Baum beim Parsen des zugrunde liegenden XHTML-Dokuments auf.
In einem zweiten Durchlauf durch den Komponentenbaum wird die Seite dann gerendert. Eine initiale Anfrage auf eine Ansicht durchläuft also nur die erste und die letzte Phase des Lebenszyklus. Abbildung**Initialzündung des Lebenszyklus**zeigt diesen Ablauf.

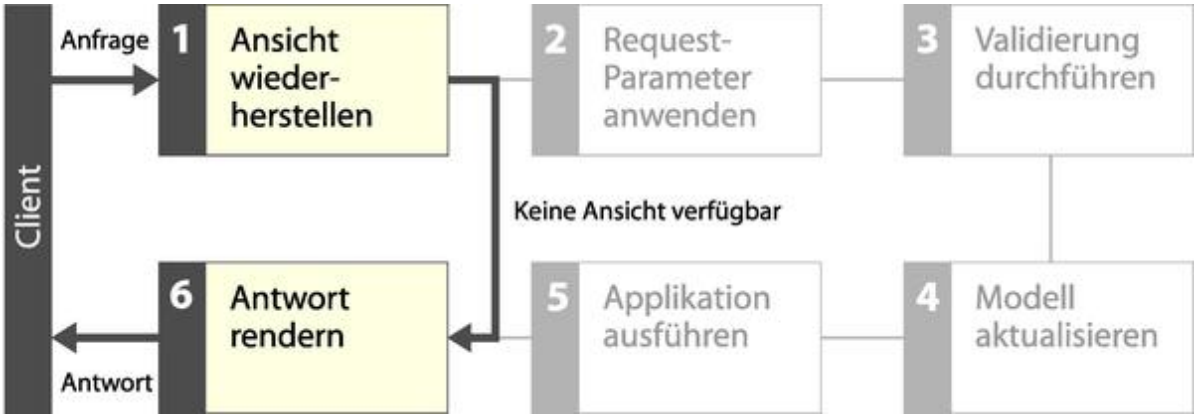


Abbildung:Initialzündung des Lebenszyklus

Wenn eine Anfrage das zweite Mal dieselbe Seite anfordert, wird in JSF-Versionen vor 2.0 der Komponentenbaum komplett aus dem zuvor am Server oder am Client gespeicherten Zustand wieder hergestellt. Zum Komponentenbaum gehören natürlich nicht nur die Komponenten selbst, sondern auch Validatoren, Konverter und die "alten" Werte sowie sämtliche anderen Eigenschaften der Komponenten.
Ab JSF 2.0 gibt es an diesem Punkt einen entscheidenden Unterschied zwischen JSP und Facelets. Mit JSP wird der komplette Komponentenbaum aus dem Seitenzustand rekonstruiert. Beim Einsatz von Facelets baut JSF hingegen zuerst die Ansicht aus der Seitendeklaration neu auf und verarbeitet erst dann den gespeicherten Zustand. Dieser neue Ansatz - auch Partial-State-Saving genannt - bietet einige Vorteile bezüglich Performance und Größe des Seitenzustands. Erste Tests zeigen, dass sich bei aktiviertem Partial-State-Saving die Größe des Seitenzustands ungefähr um den Faktor 3 reduziert.

Phase 2: Request-Parameter anwenden (*Apply Request Values*)

In dieser Phase wird der gesamte Komponentenbaum bearbeitet und die vom Benutzer eingetragenen Werte werden den einzelnen Komponenten zugewiesen. Das geschieht, indem am Wurzelknoten die Methode`processDecodes()`aufgerufen wird - der Wurzelknoten ruft dann die gleiche Methode auf seinen Kindknoten und diese wiederum auf ihren Kindknoten rekursiv auf.
Außerdem sucht sich beim Abarbeiten der Methode jede Komponente (oder genauer gesagt der der Komponente zugeordnete Renderer) aus der HTTP-Anforderung, und zwar aus den Parametern, HTTP-Kopfzeilen (Header) und Cookies, die Werte heraus, die diese Komponente betreffen, und speichert sie als "übermittelter" Wert (Submitted-Value) . Dieser Prozess wird "decodieren" (engl.*decoding*) genannt. Der Submitted-Value ist allerdings noch nicht der Wert, der dann später tatsächlich ins Modell geschrieben wird - er muss zunächst noch in ein fürs Datenmodell geeignetes Format konvertiert und validiert werden. Außerdem kann das Zurückschreiben durch das Fehlschlagen der Konvertierung oder der Validierung noch verhindert werden.
Die Konvertierung und Validierung wird dann in der nächste Phase vorgenommen, allerdings kann diese Phase auch vorgezogen werden: Mit dem Setzen des`immediate`-Attributs wird die Komponente "angewiesen", die Konvertierung und Validierung bereits in Phase 2 durchzuführen. Warum das in manchen Fällen wünschenswert ist und wie das genau funktioniert, wird in Abschnitt**[Sektion: Ändern des Lebenszyklus -- immediate-Attribut]**näher erläutert.
Der Submitted-Value ist auch in der Renderphase, speziell für das Schreiben von benutzerdefinierten Komponenten, wichtig. Wenn nämlich die Validierung oder Konvertierung von Komponenten der Seite nicht

erfolgreich verläuft, muss beim Rendering der Komponenten der Submitted-Value herangezogen werden - es wäre nicht richtig, hier den "alten" Komponentenwert, der in der`value`-Eigenschaft gespeichert ist, heranzuziehen, weil sonst Informationen des Benutzers verloren gehen würden.

Phase 3: Konvertierung und Validierung durchführen (*Process Validations*)

Der aus der HTTP-Anfrage ausgelesene Wert einer Komponente wird in dieser Phase konvertiert und validiert - natürlich wieder am Wurzelknoten startend für den ganzen Komponentenbaum. Die Konvertierung erfolgt vom zeichenkettenbasierten Submitted-Value auf die für das dahinterliegende Datenmodell notwendige Darstellung. Aus der Zeichenkette " 01.01.2012" wird dann zum Beispiel eine Instanz der Klasse`java.util.Date`.
Standardkonverter:Diese Konvertierung wird standardmäßig durchgeführt, ohne dass der Entwickler aktiv werden muss, dazu wird einfach der im Framework für einen bestimmten Datentyp definierte Standardkonverter herangezogen.
Benutzerdefinierter Konverter:Ist das Verhalten des Standardkonverters nicht ausreichend oder unerwünscht (weil zum Beispiel eine spezielle Datumsklasse für die Geschäftsdaten verwendet werden soll), kann man eigene Konverter erstellen und einzelnen Komponenten zuweisen. Das Erzeugen von benutzerdefinierten Konvertern und deren Einbindung erläutern wir in Abschnitt**[Sektion: Konvertierung]**näher.
Sofort nach dem erfolgreichen Abschluss des Konvertierungsvorgangs wird der Wert der Komponente validiert; das erledigen sogenannte Validatoren.
Validatoren:Es gibt im JSF-Standard bereits einige vorgefertigte Validatoren (z.B. ein`LengthValidator`oder ein`DoubleRangeValidator`),*Apache MyFaces*liefert noch einige Validatoren mehr mit (siehe Abschnitt**[Sektion: Validierung]**, beispielsweise gibt es Kreditkarten- und E-Mail-Validatoren). Das Einbinden dieser Validatoren erfolgt über das Hinzufügen von Kindelementen zu Komponenten. Darüber hinaus ist es sehr einfach, selbst Validatoren und auch Methoden, die eine Validierung übernehmen, zu erstellen.
Validierungsmethoden können - ähnlich wie Konverter - mit dem Attribut`validator`an die Komponente gebunden werden. Beim Validieren eines erforderlichen Werts liegt die Sache etwas anders: Hier wird kein Kindelement der Komponente hinzugefügt, sondern das Attribut`required`der Komponente auf`true`gesetzt.
Weiterführende Informationen zur Validierung finden sich in Abschnitt**[Sektion: Validierung]**.
Der nächste Schritt ist nun das Setzen des konvertierten und validierten Werts: Allerdings noch nicht in die Managed-Beans, sondern vorerst nur in die Eigenschaft`value`innerhalb der KomponenteEines der Grundkonzepte von JSF ist, dass die konvertierten und validierten Werte aller Komponenten gemeinsam in der Phase "Modell aktualisieren" in die Managed-Beans übernommen werden.:. Gleichzeitig wird mit dem Setzen von`localValueSet`markiert, dass ein Wert in der Komponente selbst gespeichert wurde. Anschließend kommt auch die Ereignisbehandlung von*JavaServer Faces*ins Spiel: Hat sich der Wert der Komponente geändert, wird ein Value-Change-Event erzeugt und registriert. In der nächsten Ereignisbehandlungsphase werden die Behandlungsroutinen für dieses Ereignis aufgerufen.
Was passiert aber, wenn die Konvertierung und/oder die Validierung fehlschlägt? Ist dies der Fall, werden die entsprechenden Fehlermeldungen generiert und die aktuelle Seite wird inklusive Fehlermeldungen als Antwort gerendert. Das bedeutet, dass alle folgenden Phasen außer der "Antwort rendern"-Phase übersprungen werden - es gibt kein Übertragen der Werte in die Geschäftsdaten und kein Ausführen von Aktionen in der Geschäftslogik.
Abbildung**Lebenszyklus für fehlgeschlagene Validierung**stellt diesen Ablauf grafisch dar.

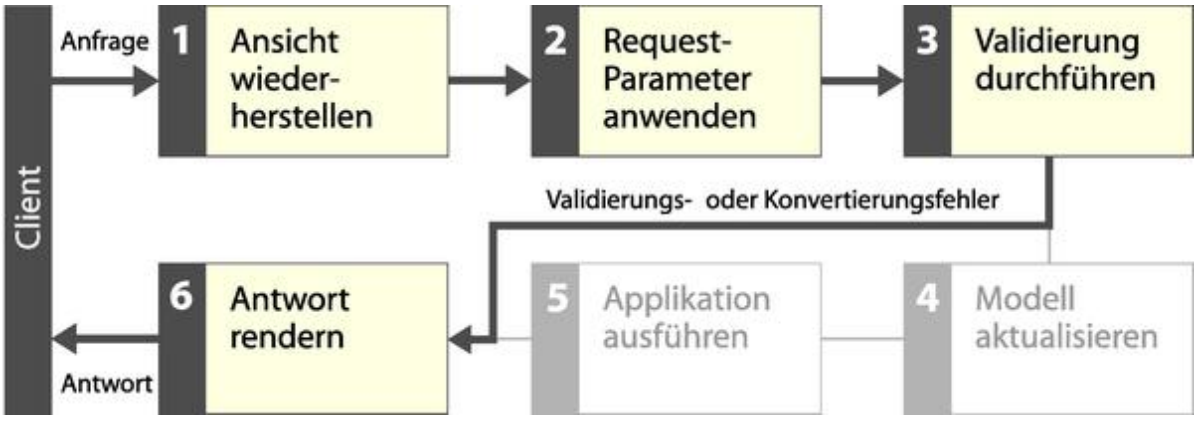


Abbildung:Lebenszyklus für fehlgeschlagene Validierung

Phase 4: Modell aktualisieren (*Update Model Values*)

Unter der Voraussetzung, dass die abgesendeten Werte (Submitted-Values) richtig konvertiert, validiert und als*Local-Value*gespeichert werden konnten, werden diese Werte jetzt auf die von den einzelnen Komponenten referenzierten Eigenschaften der Geschäftsdaten übertragen. Dafür werden die Setter-Methoden aufgerufen, die notwendig sind, um das Modell mit den neuen Daten zu aktualisieren.
Wir haben schon bei der Analyse des Beispiels kurz besprochen, wie eine solche Referenz zwischen

Komponente und Geschäftslogik aussehen kann: Üblicherweise wird dafür die`value`-Eigenschaft der Komponente mit einer Value-Expression an eine Eigenschaft der Geschäftslogik gebunden. Ein Beispiel hierfür ist der bereits bekannte Ausdruck`#{customer.firstName}`aus*MyGourmet*, mit dem die Eigenschaft`firstName`eder{Managed-Bean}`customer`referenziert wird.
Nach dem Ausführen dieser Phase wurde ein konvertierter und validierter Wert in die dahinterliegenden Beans eingetragen - wir mussten dafür noch keine einzige Zeile Applikationslogik schreiben.

Phase 5: Applikation ausführen (*Invoke Application*)

Der nächste Schritt ist das Ausführen von speziellen Ereignissen, den sogenannten Aktionen. Diese speichern beispielsweise geänderte Geschäftsdaten, lesen Geschäftsdaten auf der Basis geänderter Filterkriterien neu aus oder kommunizieren mit anderen Systemen. Jedenfalls bestimmen sie durch ihren Rückgabewert, wohin die Reise in der Anwendung gehen wird - welche Ansicht also als Nächstes aufgerufen wird.
Die Übergänge zwischen den einzelnen Ansichten haben wir durch die geeignete Definition der Navigation bereits vorher festgelegt. Wir registrieren Aktionen durch das Setzen des`action`-Attributs der Befehlskomponenten (von*UICommand*abgeleitete Komponenten).
Zusätzlich zum`action`-Attribut jeder Befehlskomponente gibt es auch ein Attribut`actionListener`. Mit diesem Attribut wird eine Verbindung zu Ereignisbehandlungsmethoden geschaffen, die knapp vor den Action-Methoden aufgerufen werden. Wozu benötigen wir solche Action-Listener? Im Gegensatz zur Action-Methode wird beim Aufruf einer mit dem Attribut`actionListener`gebundenen Methode ein Parameter vom Typ`javax.faces.event.ActionEvent`mitgegeben, und dieser Parameter enthält wiederum ein Element`component`. Damit kann also die Komponente, die diese Aktion ausgelöst hat, sehr schnell aufgefunden werden.
Weitere Informationen zu Ereignissen finden sich in Abschnitt**[Sektion: Ereignisse und Ereignisbehandlung]**.
Die Navigation wird in Abschnitt**[Sektion: Navigation]**genauer betrachtet.

Phase 6: Antwort rendern (Render Response)

In der letzten Phase wird der Komponentenbaum gerendert und die Ausgabe wird als Antwort der JSF-Anfrage zum Client geschickt. Des Weiteren speichert JSF den Zustand des Komponentenbaums für nachfolgende Anfragen auf dieselbe Ansicht.
Das Rendern des Komponentenbaums läuft im Prinzip in zwei Schritten ab:

1. Der Komponentenbaum wird aus der Seitendeklaration aufgebaut. Mit JSP passiert das durch einen Forward auf die JSP-Datei, in Facelets beim Parsen der XHTML-Datei.
2. Der in Schritt 1 erstellte Komponentenbaum wird durch einen Aufruf der Methode`encodeAll`auf dem Wurzelknoten gerendert.

In allen JSF-Versionen kommen beim Rendern der Werte der einzelnen Komponenten wieder die bereits erwähnten Konverter ins Spiel: Der Renderer holt den Wert der Komponente, ruft die Methode`getAsString()`auf dem Konverter auf und rendert das in eine Zeichenkette verwandelte Objekt zurück zum Client. Damit ist der Lebenslauf der Anfrage abgeschlossen.
Ausführlichere Informationen zum Thema Seitendeklarationssprachen finden Sie in Abschnitt**[Sektion: Seitendeklarationssprachen]**.

2.6.1 Ändern des Lebenszyklus - - immediate-Attribut

JSF wäre nicht JSF, wenn es nicht auch beim Ablauf des Lebenszyklus Möglichkeiten zur Einflussnahme gäbe. Bei der Beschreibung der Apply-Request-Values-Phase wurde bereits kurz die vorzeitige Konvertierung und Validierung von Eingabekomponenten über das`immediate`-Attribut erwähnt. Eine detailliertere Erläuterung dieses Themas holen wir jetzt nach.
Mit dem Attribut`immediate`kann das Verhalten von Eingabe- und Befehlskomponenten beim Ablauf des

Lebenszyklus beeinflusst werden. Der Standardwert für `immediate` ist `false`, was einem normalen Ablauf des Lebenszyklus entspricht. Wird der Wert auf `true` gesetzt, ändert sich das Verhalten der Komponenten. Bei Eingabekomponenten bewirkt es die vorgezogene Konvertierung und Validierung des Werts der Komponente in der Apply-Request-Values-Phase. Das Verhalten für Befehlskomponenten wird dahingehend abgeändert, dass Aktionen und Action-Listener nicht nach der Invoke-Application-Phase, sondern bereits nach der Apply-Request-Values-Phase aufgerufen werden. Das gilt wohlgemerkt nur für Komponenten, deren `immediate`-Attribut tatsächlich den Wert `true` hat. Die Abarbeitung aller anderen Komponenten der Seite bleibt unverändert.

2.6.1.1 immediate-Attribut für Eingabekomponenten

Wann ist dieses Verhalten bei Eingabekomponenten erwünscht? Sehr häufig benötigt man die Validierung und Konvertierung nur eines Teils des Komponentenbaums und die Änderung der Seite (oder des Verhaltens der Seite) aufgrund dieser kleinen Änderung. Diese Änderung soll dann unabhängig von der Validierung der anderen Komponentenwerte auf jeden Fall durchgeführt werden. Beispielsweise soll bei der Auswahl einer Zahlung mit Kreditkarte ein neues Feld für den Kartentyp und die Kreditkartennummer eingeblendet werden - diese Einblendung soll natürlich auch erfolgen, wenn im E-Mail-Adressfeld noch keine gültige E-Mail-Adresse steht. Genau dieses Verhalten erreichen Sie, indem Sie das Attribut `immediate` der Komponente zur Auswahl der Zahlungsart auf `true` setzen.
Sehen wir uns dieses kleine Beispiel in der Praxis an: Dazu erstellen wir in einer Seite ein Eingabefeld für den Namen mit verpflichtender Eingabe, ein Auswahlfeld und ein Eingabefeld für den Kreditkartentyp, das nur bei selektiertem Auswahlfeld angezeigt wird. Listing **Value-Change-Listener in der Ansicht** zeigt das Fragment.

```
<h:inputText value="#{customer.lastName}" required="true"/>
<h:selectBooleanCheckbox onclick="this.form.submit()"
    value="#{customer.useCreditCard}" immediate="true"
    valueChangeListener="#{customer.useCreditCardChanged}"/>
<h:inputText value="#{customer.creditCardType}"
    rendered="#{customer.useCreditCard}"/>
```

Die notwendige Logik zum Ein- und Ausblenden des Eingabefelds für den Kreditkartentyp befindet sich im Value-Change-Listener , der für die Komponente registriert ist. Diese Methode wird im Ablauf des Lebenszyklus aufgerufen, wenn sich der Wert der Komponente geändert hat. In unserem Fall wird in dieser Methode die Eigenschaft `useCreditCard` in der Managed-Bean auf den neuen Wert gesetzt. Die Eingabekomponente benutzt die gleiche Eigenschaft für das Attribut `rendered` und wird abhängig vom Wert der Eigenschaft dargestellt oder ausgeblendet.
Würde der Standardlebenslauf einer JSF-Anfrage abgearbeitet werden, könnte das letzte Textfeld nur dann ohne Fehlermeldungen angezeigt werden, wenn der Benutzer bereits einen Namen eingegeben hat. Ist der Name leer, erfolgt zwar die Umschaltung; es wird aber eine Fehlermeldung angezeigt.
Die Lösung ist, das `immediate`-Attribut des Auswahlfelds auf den Wert `true` zu setzen. Dadurch wird die Konvertierung und Validierung des Werts vorgezogen und der Value-Change-Listener wird vor der Konvertierung und Validierung der anderen Eingabekomponenten aufgerufen. Im Value-Change-Listener ist der Aufruf der Methode `FacesContext.getCurrentInstance().renderResponse()` notwendig, um die Konvertierung und die Validierung der anderen Komponenten zu überspringen. Listing **Value-Change-Listener in der Bean** zeigt den Code der Methode.

```
public void useCreditCardChanged(ValueChangeEvent ev) {
  Boolean useCreditCard = (Boolean) ev.getNewValue();
  if (useCreditCard != null) {
    this.useCreditCard = useCreditCard;
  }
  FacesContext.getCurrentInstance().renderResponse();
}
```

Damit funktioniert die Umschaltung ohne die aus der Konvertierung oder Validierung anderer Komponenten resultierenden Fehlermeldungen.
Abbildung **Lebenszyklus für immediate-Eingabekomponenten** zeigt den Ablauf des Lebenszyklus für unser Beispiel. Durch den Aufruf von `renderResponse()` im Value-Change-Listener der Komponente wird die Ausführung nach der Apply-Request-Values-Phase direkt bei der Render-Response-Phase fortgesetzt. Alle dazwischenliegenden Phasen werden übersprungen.

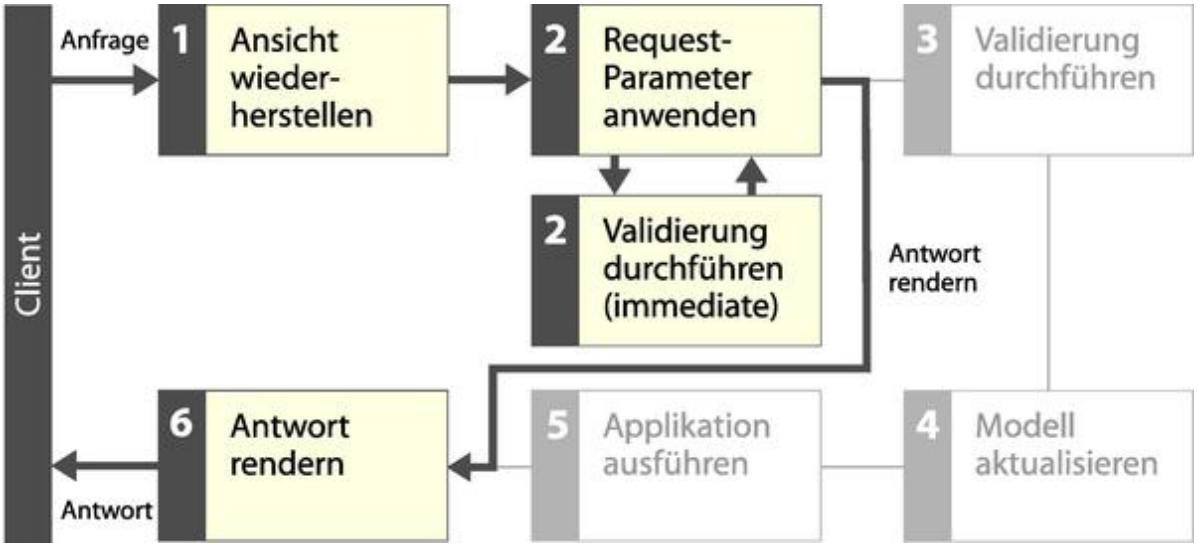


Abbildung:Lebenszyklus für immediate-Eingabekomponenten

Eine detailliertere Beschreibung der Funktionsweise von Value-Change-Ereignissen und deren Behandlung beim Ablauf des Lebenszyklus findet sich in Abschnitt**[Sektion: Value-Change-Events]**und in Beispiel*MyGourmet 3*in Abschnitt**[Sektion: *MyGourmet 3*: Ereignisse]**.

2.6.1.2 immediate-Attribut für Befehlskomponenten

Für Befehlskomponenten bewirkt das Setzen des`immediate`-Attributs auf`true`ebenfalls eine vorgezogene Behandlung der Komponente beim Ablauf des Lebenszyklus. Bei der Komponente registrierte Aktionen und Action-Listener werden dann bereits am Ende der Apply-Request-Values-Phase und nicht mehr nach der Invoke-Application-Phase ausgeführt. Die Validierung und das Update des Modells werden dabei übersprungen. Der Grund dafür ist schnell erklärt: Durch die Bearbeitung der Befehlskomponente wird die Navigation angestoßen und der darauf folgende Schritt im Lebenslauf ist immer die Render-Response-Phase.
Ein Beispiel für den Einsatz von`immediate`-Befehlskomponenten ist eine Abbrechen-Schaltfläche für Formulare. Das Setzen des Attributs`immediate`auf`true`verhindert in diesem Fall, dass der Benutzer alle verpflichtenden Felder eingeben muss, um die Bearbeitung des Formulars überhaupt abbrechen zu können. Mit einer normalen Befehlskomponente sieht der Lebenszyklus bei fehlenden verpflichtenden Eingaben wie in Abbildung**Lebenszyklus für fehlgeschlagene Validierung**aus und die relevante Invoke-Application-Phase wird gar nicht mehr erreicht. Hier die zentrale Codezeile:

```
<h:commandButton action="/cancelled.xhtml"
    value="Cancel" immediate="true"/>
```

Wie man sieht, genügt die Attributdefinition`immediate=true`auf der Komponente, um die gewünschte Funktionalität zu erzielen. Die Codezeile entstammt dem Beispiel*MyGourmet 2*, das in Abschnitt**[Sektion: *MyGourmet 2*: immediate-Attribute]**näher beschrieben wird. Abbildung**Lebenszyklus für immediate-Befehlskomponenten**zeigt den Ablauf des Lebenszyklus für unser Beispiel.

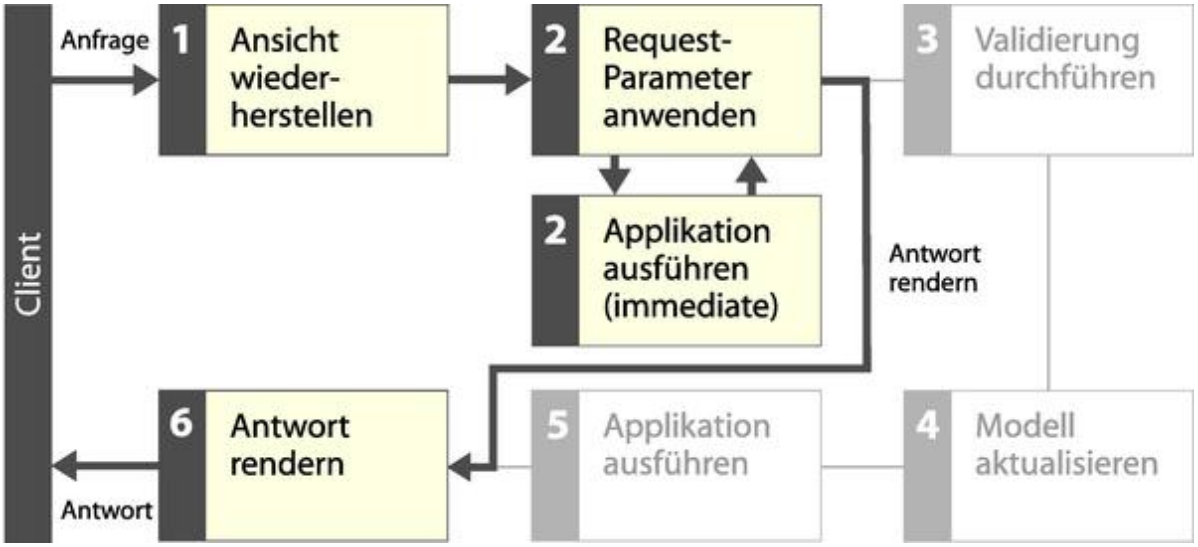


Abbildung:Lebenszyklus für immediate-Befehlskomponenten

Die Bearbeitung der`immediate`-Schaltfläche löst bereits in der Apply-Request-Values-Phase die Navigation aus und die Ausführung wird direkt mit der Render-Response-Phase fortgesetzt. Alle dazwischenliegenden Phasen werden übersprungen.
Details zu Action-Ereignissen und deren Behandlung beim Ablauf des Lebenszyklus finden Sie in Abschnitt**[Sektion: Action-Events]**.

2.6.2 MyGourmet 2: immediate-Attribute

*MyGourmet 2*ist eine kleine Erweiterung des Beispiels*MyGourmet 1*aus Abschnitt**Sektion: MyGourmet 1: Einführung anhand eines Beispiels**. Die Eingabe des Vor- und Nachnamens ist von nun an verpflichtend und das Eingabeformular für die Kundendaten erhält eine zusätzliche Schaltfläche zum Abbrechen der Bearbeitung. Der Zweck der Übung ist die Demonstration des Zusammenhangs zwischen verpflichtenden Eingabekomponenten, deren Validierung und`immediate`-Befehlskomponenten. Nur wenn bei der Befehlskomponente`immediate`auf`true`gesetzt ist, kann die Bearbeitung des Formulars auch im Fall leerer verpflichtender Eingabefelder abgebrochen werden. Andernfalls wäre bei der Ausführung des Lebenszyklus bereits bei der Process-Validations-Phase Schluss und der Benutzer sieht statt einer neuen Seite erneut das Formular - diesmal allerdings mit Fehlermeldungen, wenn die Seiteh`:messages`- oderh`:message`-Komponenten enthält. Ab JSF 2.0 wird automatisch eineh`:messages`-Komponente hinzugefügt, wenn die Project-Stage auf`Development`gesetzt ist (siehe Abschnitt**Sektion: Project-Stage**).
Die notwendigen Änderungen an`editCustomer.xhtml`sind minimal. Bei den beidenh`:inputText`-Komponenten wird das Attribut`required`auf`true`gesetzt und die neue Schaltfläche wird eingefügt. Das Attribut`action`der neuen Befehlskomponente referenziert keine Ereignisbehandlungsmethode, sondern enthält mit der Zeichenkette`cancelled.xhtml`direkt die View-ID der anzuzeigenden Ansicht. Der komplette Sourcecode der modifizierten Seite ist in Listing**MyGourmet 2: Die Seite editCustomer.xhtml**dargestellt.

```
<!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>MyGourmet - Edit Customer</title>
</head>
<body>
  <h1><h:outputText value="MyGourmet"/></h1>
  <h2><h:outputText value="Edit Customer"/></h2>
  <h:messages showDetail="true" showSummary="false"/>
  <h:form id="form">
```

```
    <h:panelGrid id="grid" columns="2">
      <h:outputLabel value="First Name:" for="firstName"/>
      <h:inputText id="firstName" required="true"
          value="#{customer.firstName}"/>
      <h:outputLabel value="Last Name:" for="lastName"/>
      <h:inputText id="lastName" required="true"
          value="#{customer.lastName}"/>
    </h:panelGrid>
    <h:commandButton id="save" value="Save"
        action="#{customer.save}"/>
    <h:commandButton id="cancel" value="Cancel"
        immediate="true" action="/cancelled.xhtml"/>
  </h:form>
</body>
</html>
```

An der zweiten Seite`showCustomer.xhtml`zum Anzeigen der gespeicherten Kundendaten hat sich nichts geändert. Neu ist allerdings die Seite`cancelled.xhtml`- sie wird dargestellt, wenn der Benutzer die Bearbeitung der Daten abbricht. Auf dieser Seite passiert außer der Ausgabe einer Nachricht nichts Aufregendes, weshalb wir auf eine eigene Abbildung verzichten.

2.7 Navigation

Ein wichtiger Teil jeder JSF-Applikation ist die Definition der Navigation zwischen den einzelnen Ansichten. Damit der Benutzer im Browser überhaupt von einer Ansicht der Anwendung zu einer anderen wechseln kann, muss die Seitendeklaration eine Befehlskomponente enthalten. Darunter versteht man eine Komponente, die das Absenden der aktuellen Seite an den Server veranlässt und somit die Abarbeitung des Lebenszyklus am Server anstößt. Von diesen Befehlskomponenten gibt es in JSF zwei:`h:commandButton`und`h:commandLink`. Wie der Name schon verrät, werden sie als Schaltfläche beziehungsweise Link in HTML ausgegeben.
Der ausschlaggebende Faktor für den Einsatz der Navigation ist das Attribut`action`der Befehlskomponenten. Darüber wird am Ende der Invoke-Application-Phase entschieden, welche Ansicht von JSF gerendert und zum Benutzer zurückgesendet wird.
Ab JSF 2.0 kann direkt die View-ID einer Ansicht im`action`-Attribut angegeben oder von der Action-Methode zurückgegeben werden - wir bezeichnen das als implizite Navigation. Sehen wir uns die Navigation in*MyGourmet*dahingehend noch einmal genauer an. Listing**Befehlskomponenten aus MyGourmet**zeigt die beiden Befehlskomponenten in der Ansicht`editCustomer.xhtml`. Die erste Schaltfläche löst eine Navigation zur Ansicht`showCustomer.xhtml`aus, da die Action-Methode`save`diese View-ID zurückgibt. Beim Betätigen der zweiten Schaltfläche landet der Benutzer auf der im Attribut`action`angegebenen Ansicht`cancelled.xhtml`.

```
<h:commandButton id="save" value="Save"
    action="#{customer.save}"/>
<h:commandButton id="cancel" value="Cancel"
    action="/cancelled.xhtml" immediate="true"/>
```

Die klassische Variante der Navigation beruht auf Navigationsregeln. Sie legen fest, wann welche Seite der Anwendung aufgerufen wird. JSF erlaubt die Definition einer beliebigen Anzahl von Navigationsregeln in der`faces-config.xml`. In einer Regel steht vor der Auflistung der einzelnen Navigationsfälle das Element`from-view-id`. Dieses Element bestimmt, von welcher Seite aus die folgenden Navigationsfälle gültig sind.
Listing**Navigationsregel für MyGourmet**zeigt eine Navigationsregel für die Seitendeklaration mit der View-ID`editCustomer.xhtml`, die der aktuell mit impliziter Navigation umgesetzten Navigation in*MyGourmet*entspricht. Die einzelnen Navigationsfälle können nur aktiv werden, wenn sich der Benutzer auf dieser Ansicht befindet.

```
<navigation-rule>
  <from-view-id>/editCustomer.xhtml</from-view-id>
```

```
  <navigation-case>
    <from-outcome>ok</from-outcome>
    <to-view-id>/showCustomer.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancel</from-outcome>
    <to-view-id>/cancelled.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Im Gegensatz zum vorangegangenen Beispiel ist die Navigationsregel in Listing**Globale Navigationsregel**global für alle Ansichten der Anwendung gültig. Das Element`from-view-id`enthält in solchen Fällen einen Ausdruck, um die zutreffenden View-IDs zu definieren. Im einfachsten Fall schließt der Wert*alle Ansichten mit ein. Genauso gut ist es aber beispielsweise möglich, die Regel mit`/secure/*`auf ein Verzeichnis innerhalb der Anwendung einzuschränken.

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>logout</from-outcome>
    <to-view-id>/home.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Im Detailbereich der Navigationsregeln folgen dann ein oder mehrere Navigationsfälle. Diese Navigationsfälle definieren über Filter- und Zielelemente die Seitenübergänge. Die Zielseite ist über das Element definiert. Die Filterelemente sind`from-outcome`und das seltener verwendete`from-action`.
Die Zeichenkette im`from-outcome`-Element der Navigationsregel muss dann dem Rückgabewert der Action-Methode entsprechen. Als Alternative kann direkt eine Zeichenkette im`action`-Attribut stehen, die dann dem Element`from-outcome`entsprechen muss.
Gibt es mehrere gleiche Rückgabewerte kann der Navigationsfall mit dem Element`from-action`noch weiter auf den Aufruf einer bestimmten Action-Methode eingeschränkt werden. Listing**Navigationsfall mit from-action**zeigt einen entsprechenden Navigationsfall.

```
<navigation-case>
  <from-action>#{customer.save}</from-action>
  <from-outcome>ok</from-outcome>
  <to-view-id>/showCustomer.xhtml</to-view-id>
</navigation-case>
```

Einen Fall haben wir noch nicht berücksichtigt. Was passiert, wenn die Action-Methode`null`zurückliefert? Die Antwort darauf ist einfach (zumindest vor JSF 2.0): Die aktuelle Seite wird nochmals angezeigt. Ab JSF 2.0 kann dieses Verhalten durch die bedingte Navigation beeinflusst werden, doch dazu später mehr.
In manchen Fällen ist es erforderlich, die Navigation explizit über den Navigation-Handler von JSF anzustoßen, um etwa aus einem Action-Listener heraus auf eine andere Seite zu navigieren. In direkter Form ist das nicht möglich, da der Action-Listener keine Zeichenkette zurückliefert. Dazu muss zuerst eine globale Navigationsregel in der`faces-config.xml`definiert werden, bei der`from-outcome`auf eine Zeichenkette und`to-view-id`auf die gewünschte Zielseite gesetzt sind. Listing**Programmatische Navigation**zeigt den Aufruf, der die Navigation auf die Zielseite bewirkt. Ab JSF 2.0 ist es natürlich auch möglich, statt der globalen Navigationsregel direkt eine View-ID anzugeben.

```
FacesContext ctx = FacesContext.getCurrentInstance();
fc.getApplication().getNavigationHandler()
    .handleNavigation(ctx, null, "cancel");
```

Seit Version 2.0 bietet JSF zusätzlich die bedingte Navigation. Mit dem Element`if`, das eine Value-Expression aufnimmt, die zu einem booleschen Wert evaluieren muss, können Navigationsfälle abhängig vom Ergebnis

dieses Ausdrucks gemacht werden. Das Element`if`kann in Kombination mit`from-outcome`oder`from-action`oder als einzige Bedingung für einen Navigationsfall eingesetzt werden. So oder so müssen alle angegebenen Bedingungen erfüllt sein, damit der Navigationsfall eintreten kann.
Listing**Bedingte Navigation im Einsatz**zeigt eine Version der bereits bekannten Navigationsregel aus*MyGourmet*mit bedingter Navigation. Die Navigation nach Betätigen der Abbrechen-Schaltfläche ist in dem Beispiel von der Eigenschaft`customer.registered`abhängig. Ist der Benutzer bereits registriert und ändert er seine Daten nur, kommt er beim Betätigen der Schaltfläche auf die Übersichtsseite`/showCustomer.xhtml`zurück. Bricht er ab, bevor die Registrierung abgeschlossen ist, gelangt er auf die Ansicht`/cancelled.xhtml`.

```
<navigation-rule>
  <from-view-id>/editCustomer.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>ok</from-outcome>
    <to-view-id>/showCustomer.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancel</from-outcome>
    <if>#{customer.registered}</if>
    <to-view-id>/showCustomer.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancel</from-outcome>
    <if>#{not customer.registered}</if>
    <to-view-id>/cancelled.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

Die Einführung der bedingten Navigation beeinflusst auch das Verhalten der Navigation, wenn die Action-Methode`null`zurückliefert. In JSF-Versionen vor 2.0 wurde dadurch immer die aktuelle Seite nochmals angezeigt. Ab Version 2.0 prüft JSF in diesem Fall auch alle Navigationsfälle, die ein`if`-Element ohne`from-outcome`oder`from-action`haben.

2.8 Ereignisse und Ereignisbehandlung

Ein essenzieller Bestandteil jeder Webapplikation ist die Behandlung von Eingaben des Benutzers am Client. Wenn der Benutzer zum Beispiel einen Button anklickt oder einen Eintrag aus einer Liste auswählt, muss eine entsprechende Aktion ausgeführt werden. In*JavaServer Faces*kommen dazu Ereignisse (Events) zum Einsatz. Diese Ereignisse werden dann von Event-Listenern behandelt, die vorher bei einer Komponente für einen speziellen Ereignistyp registriert wurden. Als Beispiel dient die Schaltfläche zum Speichern aus*MyGourmet 2*in Listing**MyGourmet 2: Die Seite editCustomer.xhtml**.
Für diese Button-Komponente wird über die Method-Expression im Attribut`action`eine Ereignisbehandlungsmethode registriert. Klickt der Benutzer im Browser auf den Schalter, wird beim Abarbeiten des Lebenszyklus ein Event erzeugt und die registrierte Methode wird aufgerufen.
Die Behandlung der Events und der Aufruf der Event-Listener wird in JSF serverseitig durchgeführt. Beim Durchlaufen des Lebenszyklus können ab der Apply-Request-Values-Phase beim Abarbeiten der Phasen Events generiert und in eine Event-Queue eingefügt werden. Wenn die Phase fertig bearbeitet ist, werden für die Events in der Queue die registrierten Event-Listener aufgerufen. Das darf nicht früher geschehen, damit jeder Event-Listener denselben Status des Komponentenbaums sieht. Abbildung**Der Lebenszyklus ("`Lifecycle"') einer HTTP-Anfrage**bietet einen Überblick über den Lebenszyklus und die möglichen Stellen zur Behandlung von Ereignissen. Aus der Abbildung ist ersichtlich, dass es beim Abarbeiten der Events die Möglichkeit zur Beeinflussung des Lebenszyklus gibt. Event-Listener können mit der Methode`FacesContext.renderResponse()`direkt zur Render-Response-Phase springen oder mit der Methode`FacesContext.responseComplete()`die Ausführung des Lebenszyklus komplett abbrechen. Wie

das konkret in einem Beispiel aussieht, zeigt*MyGourmet 3*in Abschnitt**[Sektion: *MyGourmet 3*: Ereignisse]**.
JSF definiert mehrere Arten von Events (und dazu passende Event-Listener), die durch Benutzeraktionen oder vom System ausgelöst werden. Folgende Events werden durch Benutzeraktionen ausgelöst:

- Value-Change-Events werden ausgelöst, wenn sich der Wert einer Eingabekomponente ändert. Details finden Sie in Abschnitt**[Sektion: Value-Change-Events]**.
- *Action-Events*werden von Befehlskomponenten ausgelöst, wenn sie aktiviert werden. Details finden Sie in Abschnitt**[Sektion: Action-Events]**.

Folgende Events werden vom System ausgelöst:

- *System-Events*werden vom System zu bestimmten Zeitpunkten im Lebenszyklus ausgelöst. Details finden Sie in Abschnitt**[Sektion: System-Events]**.
- *Phase-Events*werden vom System routinemäßig beim Abarbeiten des Lebenszyklus vor und nach jeder Phase ausgelöst. Details finden Sie in Abschnitt**[Sektion: Phase-Events]**.

Eine detailliertere Beschreibung der einzelnen Ereignistypen folgt in den nächsten Abschnitten.

2.8.1 Value-Change-Events

Value-Change-Events werden von Eingabekomponenten ausgelöst, deren Werte sich beim Senden des Formulars geändert haben. Wählt der Benutzer zum Beispiel die Checkbox "Kreditkarte benutzen" aus, und sie war vorher nicht angewählt, wird beim Abarbeiten des Lebenszyklus nach dem Senden der Seite ein Value-Change-Event gefeuert und registrierte Event-Listener werden aufgerufen. Bleibt der Wert gleich, wird kein Event ausgelöst.
Event-Listener für Value-Change-Events können auf zwei Arten auf Komponenten registriert werden: über eine Method-Expression im Attribut`valueChangeListener`der Komponente oder mit dem Kindelement`f:valueChangeListener`. Sehen wir uns zuerst in Listing**Registrieren einer Ereignisbehandlungsmethode**an, wie das Registrieren eines Value-Change-Listeners über eine Method-Expression im Sourcecode aussieht.

```
<h:selectBooleanCheckbox onclick="this.form.submit()"
    value="#{customer.useCreditCard}" immediate="true"
    valueChangeListener="#{customer.useCreditCardChanged}"/>
```

Listing**Code der Ereignisbehandlungsmethode**zeigt die zugehörige Ereignisbehandlungsmethode in der Managed-Bean. Die Signatur einer solchen Methode muss den Rückgabewert`void`haben und einen Parameter vom Typ`ValueChangeEvent`aufnehmen. Der Name spielt keine Rolle.

```
public void useCreditCardChanged(ValueChangeEvent e) {
  Boolean useCreditCard = (Boolean) e.getNewValue();
  if (useCreditCard != null) {
    this.useCreditCard = useCreditCard;
  }
  FacesContext.getCurrentInstance().renderResponse();
}
```

Sehen wir uns nun in Listing**Registrieren einer Value-Change-Listener-Klasse**an, wie das Registrieren eines Value-Change-Listeners über ein Kindelement der Komponente im Sourcecode aussieht.

```
<h:selectBooleanCheckbox onclick="this.form.submit()"
```

```
    value="#{customer.useCreditCard}" immediate="true">
  <f:valueChangeListener
      type="at.irian.CreditCardChangeListener"/>
</h:selectBooleanCheckbox>
```

Der Unterschied ist offensichtlich: In den Kindelementen wird statt einer Method-Expression der Name einer Klasse verwendet, die das Interface`javax.faces.event.ValueChangeListener`implementiert. Diese Form der Registrierung bietet den Vorteil, dass mehrere Listener für eine Komponente registriert werden können. Die Reihenfolge der Aufrufe entspricht dabei der Reihenfolge der Kindelemente. Listing**Code des Value-Change-Listeners**zeigt den Java-Code des Event-Listeners.

```
public class CreditCardChangeListener
    implements ValueChangeListener {
  public void processValueChange(ValueChangeEvent e) {
    Boolean useCreditCard = (Boolean) e.getNewValue();
    FacesContext fc = FacesContext.getCurrentInstance();
    if (useCreditCard != null) {
      ELContext el = fc.getELContext();
      Customer customer = (Customer) el.getELResolver()
          .getValue(el, null, "customer");
      customer.setUseCreditCard(useCreditCard);
    }
    fc.renderResponse();
  }
}
```

Auf den ersten Blick sieht dieser Code komplizierter aus, was allerdings täuscht. Wenn der Value-Change-Listener als eigene Klasse implementiert ist, können wir nicht mehr direkt auf die Eigenschaften der Managed-Bean zugreifen. Das ist aber kein Beinbruch, da wir mithilfe des ELResolvers Managed-Beans auflösen können. In unserem Fall wird zuerst die Bean`customer`geholt und dann der Wert ihrer Eigenschaft`useCreditCard`auf den Wert aus dem Value-Change-Event gesetzt.
In beiden Varianten bekommt der Event-Listener das Ereignis als Instanz der Klasse`javax.faces.event.ValueChangeEvent`übergeben. Diese Klasse bietet folgende für die Bearbeitung des Ereignisses relevante Methoden:

- `Object getNewValue()`liefert den neuen, konvertierten und validierten Wert der Komponente zurück.
- `Object getOldValue()`liefert den alten Wert der Komponente aus der Managed-Bean zurück.
- `UIComponent getComponent()`liefert die Komponente zurück, die das Ereignis ausgelöst hat.

Stellt sich noch die Frage, wie JSF Value-Change-Events intern bearbeitet. Das Ereignis wird nur ausgelöst, wenn sich der neue Wert nach der Validierung tatsächlich vom alten Wert in der Managed-Bean unterscheidet. Value-Change-Events werden im Normalfall am Ende der Process-Validations-Phase ausgelöst, nachdem die Werte aller Eingabekomponenten erfolgreich konvertiert und validiert wurden. Ist das`immediate`-Attribut der Komponente`true`, wird das Value-Change-Event bereits am Ende der Apply-Request-Values-Phase gefeuert.

2.8.2 Action-Events

Action-Events werden von Befehlskomponenten ausgelöst, wenn der Benutzer sie am Client betätigt. Im JSF-Standard sind zwei Befehlskomponenten vorgesehen:`h:commandButton`und`h:commandLink`, doch dazu mehr in Abschnitt**Sektion: Befehlskomponenten**. Wichtig ist hier, dass die Betätigung ein Senden der aktuellen Seite bewirkt. Beim dadurch angestoßenen Ablauf des Lebenszyklus wird dann ein Action-Event für die auslösende Komponente gefeuert.
Bei der Abarbeitung von Action-Events muss man zwischen Actions und Action-Listenern unterscheiden. Im Prinzip werden beide eingesetzt, um Action-Events zu bearbeiten, die von Komponenten ausgelöst wurden. Dabei gilt, dass Action-Listener immer kurz vor Actions aufgerufen werden.

Actions sind der ideale Ort für Aufrufe der Geschäftslogik, wohingegen Action-Listener eher für UI-zentrierte Logik vorgesehen sind. Diese Trennung macht durchaus Sinn, um die Businesslogik und die UI-Logik getrennt zu halten. Ein Aspekt, der besonders bei komplexeren Projekten die Wartbarkeit spürbar erhöht. Wie schon bei den Value-Change-Events gibt es auch für Action-Listener zwei Methoden der Registrierung bei Komponenten: Entweder über eine Method-Expression im Attribut`actionListener`der Komponente oder mit dem Kindelement`f:actionListener`.
In beiden Varianten bekommt die Ereignisbehandlungsmethode das ausgelöste Ereignis als Instanz der Klasse`ActionEvent`übergeben. Für die Bearbeitung des Ereignisses ist besonders die Methode`UIComponent getComponent()`interessant. Sie liefert als Rückgabewert die das Ereignis auslösende Komponente zurück.
Werfen wir auch bei den Action-Events einen kurzen Blick hinter die Kulissen der Bearbeitung durch JSF. Die Bearbeitung beginnt bereits in der Apply-Request-Values-Phase beim Decodieren des Requests. Hat der Benutzer eine Befehlskomponente betätigt, wird ein entsprechendes Event erzeugt und in die Event-Queue eingefügt. Handelt es sich um eine`immediate`-Befehlskomponente, werden die registrierten Event-Listener am Ende der Apply-Request-Values-Phase und ansonsten am Ende der Invoke-Application-Phase ausgeführt.

2.8.3 MyGourmet 3: Ereignisse

*MyGourmet 3*erweitert das Beispiel um einige Angaben zur Kreditkarte des Kunden, genauer gesagt um den Kartentyp und die Nummer der Karte. Der Benutzer kann auf der Seite`editCustomer.xhtml`über eine`h:selectBooleanCheckbox`-Komponente auswählen, ob er die Kreditkarte als Zahlungsmittel verwenden will oder nicht. Abhängig von der dortigen Auswahl sind die entsprechenden Eingabefelder auf der Seite ein- beziehungsweise ausgeblendet. Diese Umschaltung erfolgt mittels eines Value-Change-Listeners, der beim Auswahlfeld registriert ist und die Eigenschaft`useCreditCard`der Managed-Bean`customer`setzt. Genau diese Eigenschaft wird auch in einer Value-Expression im`rendered`-Attribut der Eingabefelder referenziert, um diese ein- und auszublenden.
Sehen wir uns diesen Ablauf kurz in der Praxis an, bevor wir uns näher mit dem Sourcecode beschäftigen. Wir nehmen an, dass beim ersten Aufruf der Seite`editCustomer.xhtml`die Eigenschaft`useCreditCard`der Managed-Bean den Wert`false`hat. Die Anwendung präsentiert sich dem Benutzer dann wie in Abbildung**MyGourmet 3: Kunde bearbeiten ohne Kreditkartendaten**gezeigt. In diesem Fall ist die`h:selectBooleanCheckbox`-Komponente nicht aktiviert und die`h:inputText`-Komponenten für Kartentyp und Kartennummer werden nicht gerendert - so weit, so gut.

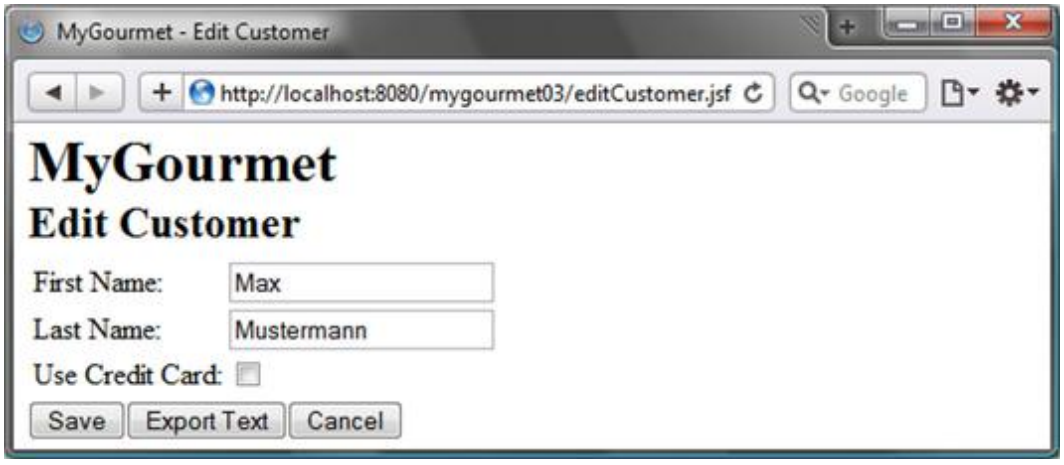


Abbildung:MyGourmet 3: Kunde bearbeiten ohne Kreditkartendaten

Klickt der Benutzer jetzt auf das Auswahlfeld, erfolgt ein Senden des Formulars durch den JavaScript-Code im`onclick`-Attribut und der Lebenszyklus wird am Server gestartet. Dabei kommt der Value-Change-Listener ins Spiel, indem er die Eigenschaft auf den neuen Wert setzt. Als Ergebnis wird die Seite mit aktiviertem Auswahlfeld und den beiden Eingabefeldern am Client angezeigt. Die Anwendung sieht dann im Browser wie in Abbildung**MyGourmet 3: Kunde bearbeiten mit Kreditkartendaten**aus. Der umgekehrte Weg ist genauso möglich, auch wenn der Benutzer noch nicht alle verpflichtenden Felder ausgefüllt hat, da das Auswahlfeld als`immediate`-Komponente realisiert ist.

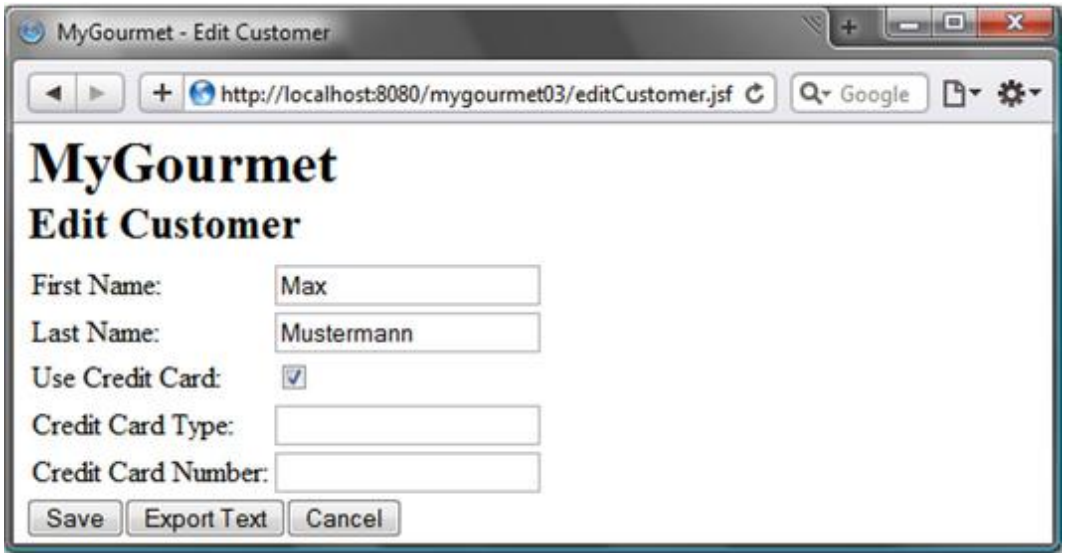


Abbildung:MyGourmet 3: Kunde bearbeiten mit Kreditkartendaten

Eine weitere Neuerung auf der Seite ist die Möglichkeit zum Export der Daten. Ein Klick auf die Schaltfläche macht nichts anderes, als die vom Benutzer eingegebenen Daten als reinen Text in einer eigenen Seite auszugeben. Dazu wird in der entsprechenden Action nach der Ausgabe die Ausführung des Lebenszyklus mit der Methode`FacesContext.responseComplete()`abgebrochen. Doch dazu kommen wir etwas weiter unten. Listing**MyGourmet 3: Änderungen in editCustomer.xhtml**zeigt die zusätzlichen Eingabekomponenten für die Kreditkartendaten in der Seitendeklaration`editCustomer.xhtml`.

```
<h:outputLabel value="Use Credit Card:" for="useCreditCard"/>
<h:selectBooleanCheckbox id="useCreditCard"
    value="#{customer.useCreditCard}"
    valueChangeListener="#{customer.useCreditCardChanged}"
    immediate="true" onclick="this.form.submit()"/>
<h:outputLabel value="Credit Card Type:" for="ccType"
    rendered="#{customer.useCreditCard}"/>
<h:inputText rendered="#{customer.useCreditCard}"
    id="ccType" value="#{customer.creditCardType}"
    required="#{customer.useCreditCard}"/>
<h:outputLabel rendered="#{customer.useCreditCard}"
    value="Credit Card Number:" for="ccNumber"/>
<h:inputText id="ccNumber"
    value="#{customer.creditCardNumber}"
    rendered="#{customer.useCreditCard}"
    required="#{customer.useCreditCard}"/>
```

Die Seite`showCustomer.xhtml`zeigt alle vom Benutzer eingegebenen Daten an. Bei den Kreditkartendaten wird wie bei der Dateneingabe das Attribut`rendered`verwendet, um sie nach Bedarf ein- oder auszublenden. Listing**MyGourmet 3: Änderungen in showCustomer.xhtml**zeigt die Änderungen zum Darstellen der Kreditkartendaten in`showCustomer.xhtml`bezüglich*MyGourmet 2*.

```
<h:outputText value="Credit Card Type:"
    rendered="#{customer.useCreditCard}"/>
<h:outputText value="#{customer.creditCardType}"
    rendered="#{customer.useCreditCard}"/>
<h:outputText value="Credit Card Number:"
    rendered="#{customer.useCreditCard}"/>
<h:outputText value="#{customer.creditCardNumber}"
    rendered="#{customer.useCreditCard}"/>
```

Die Klasse`Customer`hat neben den Getter- und Setter-Methoden der neuen Eigenschaften zwei zusätzliche Methoden zur Bearbeitung von Ereignissen erhalten.
Die Ereignisbehandlungsmethode`useCreditCardChanged`behandelt das Value-Change-Event der

Auswahlkomponente. Zum Ein- und Ausblenden der Eingabefelder für die Kreditkartendaten wird der neue Wert der Komponente aus dem Ereignis ausgelesen und in die Eigenschaft`useCreditCard`gesetzt. Ein darauffolgender Aufruf von`FacesContext.renderResponse()`erzwingt anschließend eine sofortige Ausgabe der Seite. Die Methode ist in Listing**MyGourmet 3: Value-Change-Listener**ersichtlich.

```
public void useCreditCardChanged(ValueChangeEvent e) {
  Boolean useCreditCard = (Boolean) e.getNewValue();
  if (useCreditCard != null) {
    this.useCreditCard = useCreditCard;
  }
  FacesContext.getCurrentInstance().renderResponse();
}
```

Die Methode`export`führt den Export der Daten des Kunden durch. Zur Ausgabe wird direkt in die`HttpServletResponse`geschrieben, auf die man in JSF für Servlet-basierte Anwendungen wie folgt zugreifen kann:

```
(HttpServletResponse)FacesContext.getCurrentInstance().
    getExternalContext().getResponse();
```

Nach der Ausgabe der Daten wird der Ablauf des Lebenszyklus mit einem Aufruf von`FacesContext.responseComplete()`komplett abgebrochen. Die Methode`export`ist in Listing**MyGourmet 3: Ereignisbehandlungsmethode export**zu finden.

```
public String export() {
  FacesContext fc = FacesContext.getCurrentInstance();
  try {
    HttpServletResponse resp = (HttpServletResponse)
        fc.getExternalContext().getResponse();
    resp.setContentType("text/plain");
    PrintWriter writer = resp.getWriter();
    writer.print("First Name: ");
    writer.println(firstName);
    writer.print("Last Name: ");
    writer.println(lastName);
    if (useCreditCard) {
      writer.print("Credit Card Type: ");
      writer.println(creditCardType);
      writer.print("Credit Card Number: ");
      writer.println(creditCardNumber);
    }
    fc.responseComplete();
  } catch (IOException e) {e.printStackTrace();}
  return null;
}
```

An der Seitendeklaration`cancelled.xhtml`und der Konfigurationsdatei`faces-config.xml`hat sich gegenüber dem vorangegangenen Beispiel nichts verändert.

2.8.4 System-Events

System-Events bilden ab JSF 2.0 einen neuen Typ von Ereignissen, die zu bestimmten Zeitpunkten in der JSF-Applikation oder im Lebenszyklus ausgelöst werden. JSF definiert eine ganze Reihe dieser Ereignisse und bietet die Möglichkeit, bei Bedarf dafür Listener zu registrieren. Grundsätzlich unterscheidet JSF zwischen System-

Events, die beim Ausführen des Lebenszyklus für eine spezifische Komponenteninstanz ausgelöst werden, und System-Events, die beim Ausführen der Applikation unabhängig von einer Komponenteninstanz ausgelöst werden.
Folgende System-Events werden für eine bestimmte Komponenteninstanz ausgelöst:

- `PreRenderComponentEvent`wird vor dem Rendern einer Komponente ausgelöst.
- `PreRenderViewEvent`wird vor dem Rendern der kompletten Seite für die Wurzelkomponente des Komponentenbaums ausgelöst.
- `PreValidateEvent`wird vor der Validierung einer Komponente ausgelöst.
- `PostValidateEvent`wird nach der Validierung einer Komponente ausgelöst.
- `PostAddToViewEvent`wird nach dem Hinzufügen einer Komponente zum Komponentenbaum ausgelöst.
- `PreRemoveFromViewEvent`wird nach dem Entfernen einer Komponente aus dem Komponentenbaum ausgelöst.
- `PostRestoreStateEvent`wird nach dem Wiederherstellen des Zustands einer Komponente ausgelöst.
- `PostConstructViewMapEvent`wird nach dem Erstellen der View-Map und`PreDestroyViewMapEvent`vor dem Entfernen der View-Map ausgelöst.

Folgende System-Events werden von JSF unabhängig von einer Komponenteninstanz ausgelöst:

- `PostConstructApplicationEvent`wird beim Starten der Applikation ausgelöst, nachdem die Konfiguration fertig geladen ist.
- `PreDestroyApplicationEvent`wird beim Beenden der Applikation ausgelöst.
- `PostConstructCustomScopeEvent`wird nach dem Erstellen und`PreDestroyCustomScopeEvent`vor dem Entfernen eines benutzerdefinierten Scopes ausgelöst.
- `ExceptionQueuedEvent`wird ausgelöst, wenn beim Ausführen des Lebenszyklus eine Exception an den Exception-Handler übergeben wird.

Das Registrieren von Ereignisbehandlungsmethoden für System-Events auf Komponenten erfolgt deklarativ mit dem Tag`f:event`aus der Core-Tag-Library.`f:event`wird dazu einfach als Kind-Tag in das entsprechende Komponenten-Tag eingebunden. Im Attribut`type`wird dabei der voll qualifizierte Klassenname der Event-Klasse angegeben und im Attribut`listener`eine Method-Expression für die Listener-Methode. Listing**f:event im Einsatz**zeigt, wie ein Listener für das Ereignis`PostValidateEvent`auf einer`h:inputText`-Komponente registriert wird.

```
<h:inputText value="#{bean.name}">
  <f:event type="javax.faces.event.PostValidateEvent"
      listener="#{customerBean.postValidateName}"/>
</h:inputText>
```

Für häufig verwendete System-Events gibt es Kurznamen, die im Attribut`type`statt des Klassennamens verwendet werden können. Tabelletab:sysevent-shortzeigt eine Übersicht der verfügbaren Kurznamen.

Kurzname	Event-Klasse
preRenderComponent	javax.faces.event.PreRenderComponentEvent
preRenderView	javax.faces.event.PreRenderViewEvent
postAddToView	javax.faces.event.PostAddToViewEvent
preValidate	javax.faces.event.PreValidateEvent
postValidate	javax.faces.event.PostValidateEvent

Listing**Listener-Methode für PostValidateEvent**zeigt die in Listing**f:event im Einsatz**referenzierte Ereignisbehandlungsmethode für das System-Event`PostValidateEvent`. Listener-Methoden für System-Events, die einer Komponenteninstanz zugeordnet sind, müssen immer einen Parameter vom Typ`Component-System-Event`aufweisen. Im Beispiel wird aus der übergebenen Event-Instanz mit der Methode`getComponent()`die auslösende Komponente ermittelt und in der Bean abgelegt.

```
public void postValidateName(ComponentSystemEvent ev) {
  inputComponent = ev.getComponent();
}
```

Listener für System-Events, die keiner Komponenteninstanz zugeordnet sind (wie etwa`PostConstructApplicationEvent`), lassen sich nicht über`f:event`registrieren. Solche Listener müssen als eigene Klassen umgesetzt werden, die das Interface`SystemEventListener`implementieren. Die Registrierung der Listener-Klasse erfolgt dann in der`faces-config.xml`. Dazu wird im Element`application`für jede Kombination von Event-Klasse und Listener-Klasse ein`system-event-listener`-Element hinzugefügt. Listing**System-Event-Listener in der faces-config.xml**zeigt, wie das zum Beispiel für die Listener-Klasse`at.irian.jsfatwork.MyListener`und das System-Event`Post-Construct-ApplicationEvent`aussieht.

```
<faces-config>
  <application>
    <system-event-listener>
      <system-event-listener-class>
        at.irian.jsfatwork.MyListener
      </system-event-listener-class>
      <system-event-class>
        javax.faces.event.PostConstructApplicationEvent
      </system-event-class>
    </system-event-listener>
  </application>
</faces-config>
```

In Listing**System-Event-Listener als Klasse**finden Sie die Listener-Klasse`MyListener`. Das Interface`SystemEventListener`definiert zwei zu implementierende Methoden:`processEvent`wird immer dann aufgerufen, wenn das in der`faces-config.xml`angegebene Ereignis ausgelöst wird und ein Aufruf von`isListenerForSource`den Wert`true`zurückliefert. Mit`isListenerForSource`kann eine Listener-Klasse die zu bearbeitenden Ereignisse anhand des auslösenden Objekts einschränken. In unserem Listener ist das zum Beispiel die JSF-Klasse`Application`.

```
public class MyListener implements SystemEventListener {
  public void processEvent(SystemEvent event) {
    // Ereignis behandeln
  }
  public boolean isListenerForSource(Object source) {
    return source instanceof Application;
  }
}
```

Weitere Beispiele zur Verwendung von System-Events finden Sie in*MyGourmet 6*(siehe Abschnitt**[Sektion: *MyGourmet 6*: Validierung]**), in Abschnitt**Sektion: View-Actions**und in Abschnitt**Sektion: Komponentenklasse schreiben**.

2.8.5 Phase-Events

Phase-Events werden routinemäßig bei der Ausführung des Lebenszyklus vor und nach jeder einzelnen Phase ausgelöst. Wie schon bei den bisher besprochenen Ereignistypen können auch für Phase-Events Listener registriert werden, wobei der Fokus allerdings ein etwas anderer ist. Da Phase-Events von JSF als Teil der Ausführung des Lebenszyklus ausgelöst werden, liegt ihr Haupteinsatzgebiet bei JSF-naher Funktionalität oder beim Debugging.
Für die Behandlung von Value-Change-Events und Action-Events werden Listener für Ereignisse bei einzelnen

Komponenten registriert. Die Einbindung von Listenern für Phase-Events unterscheidet sich davon etwas - zumindest vor JSF-Version 1.2. Sie werden direkt in der`faces-config.xml`für den Lebenszyklus registriert. Wie das beispielsweise mit der Klasse`DebugPhaseListener`aussieht, zeigt Listing**Phase-Listener konfigurieren**.

```
<lifecycle>
  <phase-listener>at.irian.DebugPhaseListener</phase-listener>
</lifecycle>
```

Wie sieht eine solche Klasse aus? Jeder Phase-Listener muss das Interface`javax.faces.event.PhaseListener`implementieren. Dieses Interface definiert drei Methoden:

- `void afterPhase(PhaseEvent ev)`wird nach der Ausführung der Phase aufgerufen.
- `void beforePhase(PhaseEvent ev)`wird vor der Ausführung der Phase aufgerufen.
- `PhaseId getPhaseId()`gibt an, für welche Phase der Listener Ereignisse behandelt. Die Klasse`PhaseId`stellt Konstanten zum Identifizieren einzelner Phasen bereit, wie zum Beispiel`Phase-Id.RENDER_RESPONSE`. Für Listener, die in allen Phasenübergängen aufgerufen werden sollen, gibt es zusätzlich die Konstante`Phase-Id.ANY_PHASE`.

Die Klasse`at.irian.DebugPhaseListener`dient als einfaches Beispiel, um die Einsatzmöglichkeiten von Phase-Listenern zu demonstrieren. Der Zweck des Beispiels ist, vor und nach jeder Phase eine Logmeldung auszugeben. Der Sourcecode der Klasse findet sich in Listing**Ein Phase-Listener zum Debuggen**.

```
public class DebugPhaseListener implements PhaseListener {
  static Log log = LogFactory.getLog(DebugPhaseListener.class);

  public void afterPhase(PhaseEvent event) {
    log.debug("After phase: " + event.getPhaseId());
  }
  public void beforePhase(PhaseEvent event) {
    log.debug("Before phase: " + event.getPhaseId());
  }
  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
}
```

Die Anzahl der Listener ist natürlich auch bei Phase-Events nicht begrenzt. Die Reihenfolge der Aufrufe bei der Ausführung des Lebenszyklus entspricht der Reihenfolge der Definition.
In JSF 1.2 wurde eine weitere Möglichkeit zum Registrieren von Phase-Listenern eingeführt. Mit dem Tag`f:phaseListener`kann für einzelne Seiten ein Phase-Listener eingebunden werden. Die Definition der zu verwendenden Klasse erfolgt analog zu den bereits vorgestellten Tags über das Attribut`type`. In unserem kleinen Beispiel sieht das dann so aus:

```
<f:view>
  <f:phaseListener type="at.irian.DebugPhaseListener"/>
```

Phase-Listener werden gerne von externen Frameworks und Bibliotheken für JSF verwendet, um die Funktionalität zu erweitern. Durch die fix definierten Aufrufe vor und nach einzelnen Phasen bieten sie ideale Einstiegspunkte in die Ausführung des Lebenslaufs. Besonders die Möglichkeit, vor den einzelnen Phasen Code auszuführen, erlaubt es Erweiterungen, eigenen Initialisierungscode einzubinden.

2.8.6 MyGourmet 4:

Phase-Listener und System-Events

In*MyGourmet 4*fügen wir keine neue Funktionalität zur Anwendung hinzu, sondern erweitern sie um zwei Phase-Listener und einen Listener für System-Events. Der erste Phase-Listener gibt Debuginformationen zu Beginn und am Ende jeder Phase aus und der zweite listet alle Request-Parameter auf. Anhand der Ausgaben dieser Ereignisbehandlungsmethoden werden wir weiter unten nochmals die Ausführung des Lebenszyklus durch JSF genauer unter die Lupe nehmen. Der System-Event-Listener gibt beim Starten und Beenden der Applikation jeweils eine Logmeldung aus.
Sehen wir uns zuerst in Listing**MyGourmet 4: Phase-Listener zum Debuggen**die Klasse`DebugPhaseListener`an. Wie jeder Phase-Listener implementiert auch sie das Interface`PhaseListener`. Die Methode`getPhaseId()`liefert`PhaseId.ANY_PHASE`zurück, wodurch die beiden Ereignisbehandlungsmethoden vor beziehungsweise nach jeder Phase aufgerufen werden. In`beforePhase`und`afterPhase`werden nur Logmeldungen ausgegeben, mit deren Hilfe wir etwas weiter unten den Ablauf des Lebenszyklus analysieren. Der Listener ist somit einsatzbereit und muss nur noch registriert werden.

```
public class DebugPhaseListener implements PhaseListener {
  static Log log = LogFactory.getLog(DebugPhaseListener.class);

  public void afterPhase(PhaseEvent event) {
    log.debug("After phase: " + event.getPhaseId());
  }
  public void beforePhase(PhaseEvent event) {
    log.debug("Before phase: " + event.getPhaseId());
  }
  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
}
```

Der zweite Listener`ParameterPhaseListener`gibt vor der Ausführung der Apply-Request-Phase alle Request-Parameter in Form von Logmeldungen aus. Seine Methode`getPhaseId()`muss daher den Wert`PhaseId.APPLY_REQUEST_VALUES`zurückliefern. In`beforePhase`wird die`Map`mit allen Request-Parametern über folgenden Code ausgelesen:

```
FacesContext.getCurrentInstance().getExternalContext()
    .getRequestParameterMap();
```

Die Implementierung von`afterPhase`kann leer bleiben. Somit ist auch dieser Listener bereit für die Registrierung. Der Sourcecode von`ParameterPhaseListener`findet sich in Listing**MyGourmet 4: Phase-Listener zum Ausgeben der Request-Parameter**.

```
public class ParameterPhaseListener implements PhaseListener {
  static Log log = LogFactory.getLog(
      ParameterPhaseListener.class);

  public void beforePhase(PhaseEvent event) {
    FacesContext fc = FacesContext.getCurrentInstance();
    Map<String, String> map = fc.getExternalContext().
        getRequestParameterMap();
    for (String key : map.keySet()) {
      StringBuilder param = new StringBuilder();
```

```
        param.append("Parameter: ");
        param.append(key);
        param.append(" = ");
        param.append(map.get(key));
        log.debug(param.toString());
      }
    }
    public void afterPhase(PhaseEvent event) {
    }
    public PhaseId getPhaseId() {
      return PhaseId.APPLY_REQUEST_VALUES;
    }
  }
```

Die Registrierung der beiden Phase-Listener erfolgt in der Datei`faces-config.xml`im`WEB-INF`-Verzeichnis der Applikation. Diese Datei ist der zentrale Punkt zur Konfiguration verschiedenster Aspekte einer JSF-Anwendung. Wir werden im Laufe des Buches noch auf diverse Einstellungen zu sprechen kommen. In der`faces-config.xml`fügen wir für jeden der Listener unter dem Element`lifecycle`ein Kindelement`phase-listener`mit dem Namen der Klasse hinzu. Hier können beliebig viele Listener registriert werden, wobei die Reihenfolge der Ausführung mit der Reihenfolge ihrer Definition übereinstimmt. Die`faces-config.xml`von*MyGourmet 4*ist in Listing**MyGourmet 4: faces-config.xmlmit Registrierung der Phase-Listener**zu finden.

```
<faces-config 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"
    version="2.2">
  <lifecycle>
    <phase-listener>
      at.irian.jsfatwork.gui.jsf.DebugPhaseListener
    </phase-listener>
    <phase-listener>
      at.irian.jsfatwork.gui.jsf.ParameterPhaseListener
    </phase-listener>
  </lifecycle>
</faces-config>
```

2.8.6.1 Analyse des Lebenszyklus

Nachdem die Funktionsweise der Phase-Listener und deren Registrierung jetzt klar sein sollte, folgt jetzt der wirklich interessante Aspekt von*MyGourmet 4*. Mit den Ausgaben der Phase-Listener lässt sich der Ablauf des Lebenszyklus sehr schön nachvollziehen.
Beginnen wir im ersten Schritt beim initialen Aufruf der Seite`edit-Customer.xhtml`. Der Listener gibt während der Ausführung folgende Logmeldungen aus:

```
DEBUG Before phase: RESTORE_VIEW(1)
DEBUG After phase: RESTORE_VIEW(1)
DEBUG Before phase: RENDER_RESPONSE(6)
DEBUG After phase: RENDER_RESPONSE(6)
```

Die Logmeldungen bestätigen das erwartete Ergebnis für den erstmaligen Aufruf einer Seite. Nur die Phasen Restore-View zum Erstellen des Komponentenbaums und Render-Response zum Rendern der Seite werden durchlaufen. Der Benutzer sieht als Ergebnis die Seite zum Editieren der Kundendaten in seinem Browser.
Im nächsten Schritt gibt der Benutzer seinen Vor- und Nachnamen ein und klickt auf die Schaltfläche zum Abspeichern. Dadurch wird das Formular der Seite abgeschickt und am Server wird erneut der Lebenszyklus durchlaufen. Der Phase-Listener gibt diesmal folgende Logmeldungen aus:

```
DEBUG Before phase: RESTORE_VIEW(1)
DEBUG After phase: RESTORE_VIEW(1)
DEBUG Before phase: APPLY_REQUEST_VALUES(2)
DEBUG After phase: APPLY_REQUEST_VALUES(2)
DEBUG Before phase: PROCESS_VALIDATIONS(3)
DEBUG After phase: PROCESS_VALIDATIONS(3)
DEBUG Before phase: UPDATE_MODEL_VALUES(4)
DEBUG After phase: UPDATE_MODEL_VALUES(4)
DEBUG Before phase: INVOKE_APPLICATION(5)
DEBUG After phase: INVOKE_APPLICATION(5)
DEBUG Before phase: RENDER_RESPONSE(6)
DEBUG After phase: RENDER_RESPONSE(6)
```

Beim Senden der Seite werden alle sechs Phasen des Lebenszyklus durchlaufen. Die Liste der Request-Parameter sieht dabei laut Phase-Listener`ParameterPhaseListener`wie folgt aus:

```
DEBUG Parameter: javax.faces.ViewState = ...
DEBUG Parameter: form:lastName = Mustermann
DEBUG Parameter: form_SUBMIT = 1
DEBUG Parameter: form:save = Save
DEBUG Parameter: form:firstName = Max
```

Was passiert aber jetzt genau im Hintergrund? Nach der Wiederherstellung des Komponentenbaums in der Restore-View-Phase - dazu benutzt JSF den Parameter`javax.faces.ViewState`- werden in der Apply-Request-Values-Phase beim Decodieren die Submitted-Values der Komponenten gesetzt. Nachdem die wichtigsten Komponenten der Seite`editCustomer.xhtml`sprechende IDs aufweisen, ist die Aufschlüsselung der Parameter relativ einfach. Der Parameter`form_SUBMIT`sagt dem Formular, dass es übermittelt wurde,`form:firstName`ist der vom Benutzer eingegebene Vorname "Max" und`form:lastName`der Nachname "Mustermann". Durch den Parameter`form:save`weiß die entsprechende Befehlskomponente beim Decodieren, dass das Senden von ihr ausgelöst wurde und dass sie ein entsprechendes Action-Event in die Event-Queue einfügen muss.
Damit sind alle Submitted-Values in den Komponenten abgelegt und die weitere Verarbeitung kann bei der Konvertierung und Validierung fortgesetzt werden. Hier passiert nichts Besonderes, da Vor- und Nachname gültige Werte haben. Im nächsten Schritt - dem Modell-Update - werden die konvertierten und validierten Werte der Eingabekomponenten in das Modell zurückgeschrieben. In unserem Fall bedeutet das einen Aufruf der Methoden`setFirstName`und`setLastName`der Managed-Bean`customer`.
Damit sind wir auch schon bei der Invoke-Application-Phase angekommen. Dort wird als Reaktion auf das in der Apply-Request-Values-Phase erzeugte Ereignis die bei der Befehlskomponente registrierte Ereignisbehandlungsmethode aufgerufen. Für die Speichern-Schaltfläche ist das die Methode`save`der Managed-Bean`customer`. Das Ergebnis dieser Methode bestimmt die nächste anzuzeigende Seite. Bleibt als letzter Schritt nur noch das Rendern dieser Seite - in unserem Fall`show-Customer.xhtml`- und der Lebenszyklus ist abgeschlossen.
Nachdem wir den Standardfall durchexerziert haben, kommen wir nun zu einigen Spezialfällen. Schauen wir uns zunächst an, was passiert, wenn der Benutzer das Auswahlfeld "Use Credit Card" aktiviert. Der JavaScript-Code im Attribut`onclick`sorgt dafür, dass bei jedem Klick auf das Auswahlfeld das Formular übermittelt wird - eine Notwendigkeit, damit der registrierte Value-Change-Listener am Server seine Arbeit verrichten kann. Aber sehen wir uns zuerst die Logmeldungen des Phase-Listeners an, bevor wir weitere Details betrachten:

```
DEBUG Before phase: RESTORE_VIEW(1)
DEBUG After phase: RESTORE_VIEW(1)
DEBUG Before phase: APPLY_REQUEST_VALUES(2)
DEBUG After phase: APPLY_REQUEST_VALUES(2)
DEBUG Before phase: RENDER_RESPONSE(6)
DEBUG After phase: RENDER_RESPONSE(6)
```

Wie Sie sehen können, springt die Ausführung des Lebenszyklus nach der Apply-Request-Values-Phase direkt zur Render-Response-Phase. Der Grund dafür ist schnell erklärt. Die Eingabekomponente ist`immediate`und wird

somit bereits in der Apply-Request-Values-Phase konvertiert und validiert. Das bedeutet weiterhin, dass auch der Value-Change-Listener aufgerufen wird, wenn sich der Wert der Komponente geändert hat. Genau dieser ruft dann auch nach dem Setzen des neuen Werts in der Eigenschaft`useCreditCard`die Methode`FacesContext.renderResponse()`auf, und die Phasen 3 bis 5 werden übersprungen.
Sehen wir uns abschließend noch kurz die Ausgabe des Phase-Listeners bei fehlgeschlagener Validierung an:

```
DEBUG Before phase: RESTORE_VIEW(1)
DEBUG After phase: RESTORE_VIEW(1)
DEBUG Before phase: APPLY_REQUEST_VALUES(2)
DEBUG After phase: APPLY_REQUEST_VALUES(2)
DEBUG Before phase: PROCESS_VALIDATIONS(3)
DEBUG After phase: PROCESS_VALIDATIONS(3)
DEBUG Before phase: RENDER_RESPONSE(6)
DEBUG After phase: RENDER_RESPONSE(6)
```

Die Ausgabe ist wie erwartet. Nach der Process-Validations-Phase wird bei einem Validierungsfehler sofort die Render-Response-Phase angesprungen und die gleiche Seite wird nochmals dargestellt - im Idealfall mit sprechenden Fehlermeldungen.
Neben den beiden Phase-Listenern ist in*MyGourmet 4*zu Demonstrationszwecken noch der Listener`ApplicationListener`für die System-Events`PostConstructApplicationEvent`und`PreDestroyApplicationEvent`in der`faces-config.xml`registriert. Die einzige Funktionalität des Listeners ist, beim Hochfahren und Beenden der Applikation eine Logmeldung auszugeben. Listing**MyGourmet 4: System-Event-Listener**zeigt die Listener-Klasse. Nachdem die Methode`processEvent`für beide Ereignisse aufgerufen wird, muss anhand der übergebenen Instanz der Klasse`SystemEvent`eine Unterscheidung getroffen werden.

```
public class ApplicationListener implements SystemEventListener{
  static Log log = LogFactory.getLog(ApplicationListener.class);

  public void processEvent(SystemEvent event) {
    if (event instanceof PostConstructApplicationEvent) {
      log.debug("application startup ");
    } else if (event instanceof PreDestroyApplicationEvent) {
      log.debug("application shutdown");
    }
  }

  public boolean isListenerForSource(Object source) {
    return source instanceof Application;
  }
}
```

2.9 Seitendeklarationssprachen

Eine Seitendeklarationssprache (*View Declaration Language, VDL*) ist eine Syntax, um Ansichten beziehungsweise Seiten für JSF zu deklarieren. Dieses Konzept wurde in JSF 2.0 im Zuge der Integration von Facelets eingeführt, um von der eingesetzten Technologie zu abstrahieren. Der Standard unterstützt ab Version 2.0 mit Facelets und JSP zwei konkrete Implementierungen einer VDL.
Vor Version 2.0 von*JavaServer Faces*war JSP die primäre Seitendeklarationssprache der Spezifikation. Entwicklern sollte durch den Einsatz einer standardisierten und weitverbreiteten Technologie der Umstieg auf JSF so einfach wie möglich gemacht werden. Ein nobler Ansatz - doch leider ist das Gespann von JSF und JSP nur eine suboptimale Lösung, was hauptsächlich daran liegt, dass die beiden Technologien für verschiedene Einsatzzwecke entworfen wurden.
Sehen wir uns diese Diskrepanz zwischen JSP und JSF etwas genauer an. Wie Sie bereits wissen, wird eine Anfrage in JSF in mehreren Phasen abgearbeitet. Der Aufbau des Komponentenbaums aus der Seitendeklaration

und die Ausgabe der Ansicht finden dabei in verschiedenen Phasen statt. Und genau hier liegt das Problem:*JavaServer Pages*wurden für einen viel einfacheren Lebenszyklus entwickelt.
Eine Anfrage wird mit JSPs in einer einzigen Phase bearbeitet - in der wird auch gleich die Antwort ausgegebenIm Hintergrund wird jede JSP-Datei in ein Servlet mit einer mehr oder weniger komplexen Reihe von`out.println(" ...")`-Anweisungen kompiliert.:. Das ist in Kombination mit JSF besonders dann problematisch, wenn die JSP-Datei neben Komponenten-Tags auch einfache Inhalte wie HTML-Elemente und Text enthält. Diese werden beim Abarbeiten der Datei direkt ausgegeben, während aus den Komponenten-Tags der Komponentenbaum aufgebaut wird, der erst am Ende des Lebenszyklus gerendert wird. Das und andere Gründe machen den Aufbau des Komponentenbaums aus JSP-Seitendeklarationen relativ aufwendig.
Hier kommt Facelets ins Spiel - eine alternative Technologie zur Deklaration von Ansichten, die perfekt in den Lebenszyklus von JSF integriert ist. Die Hauptaufgabe von Facelets besteht im Aufbau von Komponentenbäumen aus XHTML-Dokumenten. Facelets erledigt diese Aufgabe mit Bravour und bietet im Vergleich zu JSP auch noch einige andere Vorteile, auf die wir im nächsten Abschnitt eingehen werden.
In*JavaServer Faces*2.0 hat Facelets den Platz von JSP als primäre Seitendeklarationssprache eingenommen. JSP wird im Standard nur mehr aus Kompatibilitätsgründen unterstützt. Neue Features sind ab JSF 2.0 nur noch mit Facelets verfügbar.

2.9.1 Vorteile von Facelets gegenüber JSP

Der Einsatz von Facelets in einem JSF-Projekt bietet eine Reihe von praktischen Vorteilen gegenüber dem Einsatz von*JavaServer Pages*. Das ist auch nicht weiter verwunderlich, da Facelets speziell für JSF entwickelt wurde. Durch die daraus resultierende, nahtlose Integration in den Lebenszyklus treten viele JSP-inhärente Probleme beim Aufbau der Ansichten erst gar nicht auf.
Die optimale Abstimmung auf den JSF-Lebenszyklus macht das Erstellen und Ausgeben von Ansichten mit Facelets effizienter. Das wirkt sich einerseits in einer höheren Geschwindigkeit beim Abarbeiten der Seiten gegenüber JSP aus. Andererseits ergeben sich auch für Seitenautoren einige Vorteile. Probleme bei der Kombination von Komponenten und Standard-HTML-Inhalten gehören mit Facelets der Vergangenheit an. Der Einsatz von`f:verbatim`erübrigt sich damit genauso wie die Notwendigkeit,`f:view`einzusetzen.
Facelets bietet die Möglichkeit, Seiten und Inhalte auf Templates aufzubauen. Nähere Informationen dazu finden Sie in Abschnitt**Sektion: Templating**. Die Wiederverwendung von Inhalten wird generell groß geschrieben, was die Modularisierung Ihrer Projekte erleichtert und dadurch die Wartbarkeit erhöht. Seitenfragmente lassen sich einfach zentral ablegen und in mehrere Seiten integrieren. Das geht sogar so weit, dass Sie Komponenten erstellen können, ohne eine Zeile Java-Code zu verfassen. Wie das funktioniert, erfahren Sie in Kapitel**Kapitel: Die eigene JSF-Komponente**.
Nach diesem kurzen Überblick über die Vorteile von Facelets sehen wir uns im nächsten Abschnitt die praktischen Aspekte von Seitendeklarationen etwas genauer an.

2.9.2 Seitendeklarationssprachen im Einsatz

JSF 2.0 war ein großer Schritt vorwärts, was die Unterstützung von Seitendeklarationssprachen betrifft. In älteren Versionen musste noch der View-Handler ausgetauscht werden, um Facelets in eigenen Projekten einzusetzen. Mit Version 2.0 wurde dieser Schritt obsolet. JSP und Facelets funktionieren jetzt auf Anhieb ohne weitere Konfiguration.
Bei der Integration von Facelets in den JSF-Standard wurde besonderer Wert auf die Rückwärtskompatibilität zur letzten Version vor JSF 2.0 gelegt. Nach außen hin hat sich für Entwickler, die Facelets als VDL einsetzen, nichts geändert. Lediglich die Implementierung wurde an JSF 2.0 angepasst. Wenn Sie in Ihrem Projekt also Klassen aus dem Package`com.sun.facelets`oder einem Subpackage verwenden, müssen Sie Ihr Projekt an die neuen Klassen anpassen. In allen anderen Fällen wird der Einsatz von Facelets sogar einfacher, da das Einbinden der Jar-Dateien und des Facelets-View-Handlers entfällt.

Bis jetzt haben wir in allen Beispielen Facelets eingesetzt. Es wird Zeit, eine solche Deklaration etwas genauer unter die Lupe zu nehmen. Listing**Seitendeklaration in Facelets**zeigt nochmals`showCustomer.xhtml`aus*MyGourmet 1*in leicht gekürzter Fassung.

```
<!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>Show Customer</title></head>
<body>
  <h1><h:outputText value="Show Customer"/></h1>
  <h:panelGrid id="grid" columns="2">
    <h:outputText value="First Name:"/>
    <h:outputText value="#{customer.firstName}"/>
    <h:outputText value="Last Name:"/>
    <h:outputText value="#{customer.lastName}"/>
  </h:panelGrid>
  <h:outputText value="Customer saved!"/>
</body>
</html>
```

Eine Facelets-Deklaration ist technisch gesehen nichts anderes als ein XHTML-Dokument mit dem Doctype*XHTML Transitional*. Das Einbinden der verwendeten Tag-Bibliotheken wird über Namensräume im Wurzelelement des Dokuments realisiert. Im Beispiel aus Listing**Seitendeklaration in Facelets**sind alle Tags der Core-Tag-Library unter dem Präfix`f:`und die Tags der HTML-Tag-Library unter dem Präfix`h:`verfügbar. Der Rest des Dokuments besteht nur mehr aus Komponenten-Tags und reinem HTML.
JSF entscheidet anhand einiger Regeln, ob eine Seitendeklaration als JSP oder Facelets interpretiert wird. Dieses Verhalten kann über folgende Kontextparameter in der`web.xml`gesteuert werden:

- `javax.faces.DEFAULT_SUFFIX`definiert eine durch Leerzeichen getrennte Liste von Erweiterungen für View-Identifier, die JSF als JSP interpretieren soll. Der Standardwert ist`.jsp`.
- `javax.faces.FACELETS_SUFFIX`definiert eine durch Leerzeichen getrennte Liste von Erweiterungen für View-Identifier, die JSF als interpretieren soll. Der Standardwert ist`.xhtml`.
- `javax.faces.FACELETS_VIEW_MAPPINGS`definiert eine durch Semikolons getrennte Liste von View-Identifiern, die JSF als Facelets interpretieren soll. Diese Liste kann auch Einträge mit Wildcards wie`/secure/*`enthalten.

Prinzipiell spricht nichts dagegen, JSP und Facelets parallel in einer Anwendung einzusetzen. Mit der Standardkonfiguration von JSF ist das auch ohne Probleme möglich. Ob es Sinn macht, einen Teil der Ansichten in JSP und einen anderen mit Facelets zu deklarieren, bleibt fraglich.

2.10 Verwendung des ID-Attributs in JSF

Neben dem`value`-Attribut wird in JSF vermutlich das`id`-Attribut am häufigsten verwendet. Dieses Attribut erlaubt dem Entwickler, eine eindeutige ID für die Komponente festzulegen. Die ID muss dabei aber nicht eindeutig für die gesamte Seite sein, sondern nur innerhalb des aktuellen Naming-Containers, in den die Komponente eingebettet ist. Naming-Container sind in JSF einige Komponenten, die große Bereiche der Seite eingrenzen,

wieh:form,h:dataTableoder sämtliche Kompositkomponenten .
Um den Entwickler dabei zu unterstützen, die Komponenten-IDs einer Webseite eindeutig zu halten, schreibt JSF nicht direkt die vom Benutzer angegebene ID in die gerenderte Ausgabe. Stattdessen kommt die sogenannte Client-ID der Komponente zum Einsatz. Die Client-ID wird durch Doppelpunkte getrennt aus der ID der Komponente und den IDs aller Elternkomponenten, die Naming-Container sind, zusammengesetzt.
Als Beispiel sind in Abbildung**Client-IDs in MyGourmet 3**die Client-IDs aus dem Beispiel*MyGourmet 3*zu finden. Alle Eingabekomponenten und die Schaltflächen sind innerhalb des Formulars mit der IDformangeordnet, was auch an den Client-IDs zu erkennen ist.

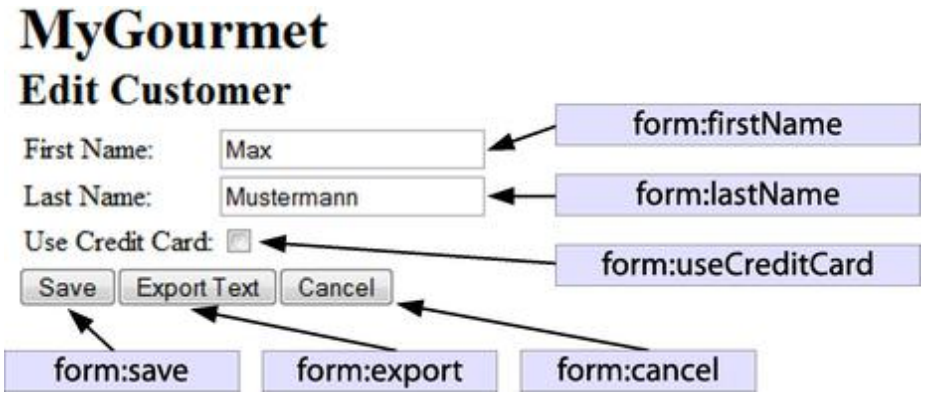


Abbildung:Client-IDs in MyGourmet 3

Bei einer Änderung der Seitenstruktur, bei der Naming-Container als Eltern hinzukommen oder wegfallen, ändert sich auch die Client-ID der eingebetteten Komponente. Ab JSF 1.2 kann dieses Verhalten für Form-Komponenten beeinflusst werden. Das AttributprependIddesh:form-Tags steuert, ob die ID des Formulars den Bezeichnern ihrer Komponenten vorangestellt wird (Wert isttrue) oder nicht (Wert istfalse). Der Defaultwert isttrue.
Abbildung**Client-IDs in MyGourmet 3 mit prependId auf false**zeigt nochmals die Client-IDs aus dem Beispiel*MyGourmet 3*- diesmal ist allerdings das AttributprependIddesh:form-Tags auffalsegesetzt.

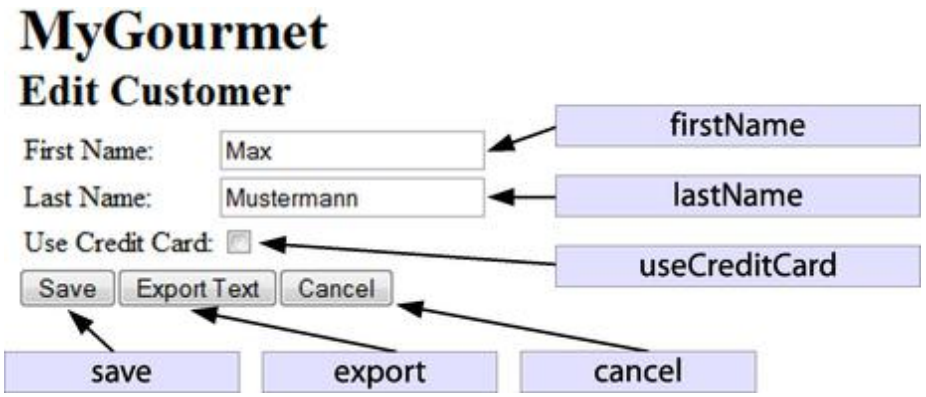


Abbildung:Client-IDs in MyGourmet 3 mit prependId auf false

Natürlich wird die Client-ID auch innerhalb des Quelltextes benötigt - beispielsweise, um die ID einer Komponente zu rendern. Der Zugriff auf die vollständige ID erfolgt über die inUIComponentdefinierte MethodegetClientId(FacesContext fc). Ist beim ersten Aufruf dieser Methode noch keine ID für die Komponente gesetzt, erhält die Komponente eine automatische ID zugewiesen. Diese automatischen IDs sind über das Präfixj_idvon händisch zugewiesenen Bezeichnern unterscheidbar.

2.11 Konvertierung

Die Konvertierung ist ein wichtiger Aspekt im JSF-Lebenszyklus, da Werte am Webclient in Form von Zeichenketten, serverseitig jedoch als Java-Typen dargestellt werden. Den Konvertern fällt dabei die Rolle eines internen Vermittlers zu. Sie kümmern sich um die Umwandlung der vom Benutzer eingegebenen Zeichenketten in Java-Objekte und wandeln Java-Objekte beim Rendern der Ausgabe wieder in Zeichenketten um.
In JSF gibt es für sehr viele Datentypen bereits Standardkonverter, die automatisch zum Einsatz kommen. Wenn wir zum Beispiel in*MyGourmet 3*den Typ der EigenschaftcreditCardNumberder Klasse vonStringaufIntegerändern, müssen wir uns nicht um die Konvertierung kümmern. JSF verwendet, falls kein anderer Konverter definiert ist, automatisch den Standardkonverter für den Datentyp .
Dieser Standardkonverter ist für beide Richtungen der Konvertierung verantwortlich. Zum einen konvertiert er

beim Rendern der Ansicht den`Integer`-Wert der Eigenschaft`creditCardNumber`in eine Zeichenkette. Beim Absenden eines Formulars konvertiert er außerdem die vom Benutzer eingegebene Zeichenkette wieder in einen`Integer`.
Jeder Konverter implementiert das in Listing**Das Interface Converter**dargestellte Interface`javax.faces.convert.Converter`.

```
public interface Converter {
  Object getAsObject(FacesContext context,
      UIComponent component,
      String value) throws ConverterException;
  String getAsString(FacesContext context,
      UIComponent component,
      Object value) throws ConverterException;
}
```

Die zwei Richtungen der Konvertierung sind hier klar zu erkennen: Für die Konvertierung des Java-Datentyps in eine Zeichenkette wird die Methode`getAsString()`aufgerufen, für die Konvertierung aus der Zeichenkette zurück in den Java-Datentyp die Methode`getAsObject()`.
Das Auffinden eines Konverters für eine Komponente funktioniert über folgende Schritte:

- Das Tag der Komponente enthält ein Attribut`converter`, das auf einen Konverter verweist.
- Das Tag der Komponente besitzt ein Kind-Tag, das auf einen Konverter verweist.
- Die Komponente zeigt auf einen Wert mit einem gewissen Datentyp - für diesen Datentyp ist ein Konverter registriert.

Die Registrierung von Konvertern in der`faces-config.xml`behandeln wir weiter unten. Zuerst zeigen wir Ihnen im nächsten Abschnitt, welche Konverter JSF bereits im Standard ausliefert.

2.11.1 Standardkonverter

Von den Standardkonvertern, die über ein Kind-Tag eingebunden werden, sind in der Praxis das Tag`f:convertDateTime`und das Tag`f:convertNumber`relevant.
Das Tag`f:convertDateTime`hat folgende Eigenschaften:

- `type`:
  Dieses Attribut definiert, welche Teile des Datumswertes angezeigt werden. Es kann die Werte`date`,`time`oder`both`annehmen.
- `dateStyle`:
  Dieses Attribut gibt den Typ der Datumsanzeige an, wenn`type`auf`date`oder`both`gesetzt ist. Mögliche Werte sind`default`,`short`, ,`long`,`full`.
- `timeStyle`
  Dieses Attribut gibt den Typ der Zeitanzeige an, wenn`type`auf`time`oder`both`gesetzt ist. Es sind die gleichen Werte wie für die`date-Style`-Angabe gültig.
- `pattern`:
  Statt der Angabe von`type`,`dateStyle`und`timeStyle`kann hier direkt ein Datumsmuster (wie etwa`dd.MM.yyyy`) übergeben werden.
- `timeZone`:
  Die Zeitzone für die Ausgabe kann mit diesem Attribut umgestellt werden, beispielsweise ist der Standardwert für dieses Attribut`GMT`.
- `locale`:
  Hier kann eine Lokalisierung angegeben werden - sowohl als Zeichenkette als auch als Instanz der

Klasse`Locale`über eine Value-Expression. Beispiele für den Wert sind:`en_US`oder`#{session.locale}`.

Das Tag`f:convertNumber`hat folgende Eigenschaften:

- `type`:
  Kann die Eigenschaften`currency`oder`percentage`zusätzlich zum Standardwert`number`annehmen.
- `currencyCode`:
  Mit dieser Eigenschaft kann die Währung über einen international gültigen Code eingestellt werden - für den Euro wäre das beispielsweise`EUR`. Alternativ dazu können Sie auch das Attribut`currencySymbol`setzen.
- `currencySymbol`:
  Alternative zum`currencyCode`. Beispiel wäre die Angabe von für den Euro.
- `groupingUsed`:
  Gibt an, ob ein Separator (beispielsweise ein Tausenderseparator) verwendet wird.
- `locale`:
  Hier kann eine Lokalisierung angegeben werden - sowohl als Zeichenkette als auch als Instanz der Klasse`Locale`über eine Value-Expression. Beispiele für den Wert sind:`en_US`oder`#{session.locale}`.
- `minFractionDigits`:
  Dieses Attribut gibt die minimale Anzahl an Nachkommastellen an.
- `maxFractionDigits`:
  Dieses Attribut gibt die maximale Anzahl an Nachkommastellen an.
- `minIntegerDigits`:
  Dieses Attribut gibt die minimale Anzahl an Vorkommastellen an.
- `maxIntegerDigits`:
  Dieses Attribut gibt die maximale Anzahl an Vorkommastellen an.
- `pattern`:
  Alternativ zur Angabe der anderen Attribute kann mit diesem Attribut direkt ein Muster für die Darstellung festgelegt werden - beispielsweise ist hier die Angabe von`#.###,##`für das "deutsche" Zahlenformat möglich.

Für die einzelnen Attribute der Konverter gilt, dass sie möglichst nahe an die Attribute der Lokalisierungsfunktionen von Java selbst angelehnt sind - in der Java-API-Dokumentation findet sich also mehr über die möglichen Werte für die einzelnen Parameter und ihre Auswirkungen.
Die Behandlung von Fehlern und Fehlermeldungen in Form von Nachrichten beim Konvertieren wird in Abschnitt**[Sektion: Nachrichten]**näher erläutert.
Konverter zum Lokalisieren:Konverter dienen auch der Lokalisierung von Werten. Mit dem Attribut`locale`kann der Lokalisierungscode für die Standardkonverter explizit gesetzt werden - sowohl als Zeichenkette als auch als Instanz der Klasse`Locale`über eine Value-Expression. Ohne explizite Angabe wird der Standard-Lokalisierungscode verwendet. Im Beispiel in Listing**Lokalisierung mit Konvertern**wird das Datum 13. Dezember 2012 aus einer Backing-Bean ausgelesen und für Deutsch und Englisch dargestellt, was zu folgender Ausgabe führt:

```
13.12.12
12/13/12
```

```
<h:outputText id="orderDate" value="#{orderBean.orderDate}">
  <f:convertDateTime dateStyle="short" locale="de"/>
</h:outputText>
<h:outputText id="orderDate" value="#{orderBean.orderDate}">
  <f:convertDateTime dateStyle="short" locale="en"/>
```

```
</h:outputText>
```

Konverter-Binding:Ab JSF 1.2 haben die Standardkonverter-Tags (`f:convertNumber`,`f:convertDateTime`und`f:converter`) ein zusätzliches Attribut`binding`, über das die Konverterinstanz aus einer Managed-Bean bezogen werden kann.

2.11.2 Benutzerdefinierte Konverter

Natürlich kann die Funktionalität des JSF-Frameworks durch Erstellen eigener Konverter erweitert werden: Es ist sowohl möglich, bestehende Konverter durch eigene zu ersetzen als auch neue Java-Datentypen zu verarbeiten oder für einzelne Komponenten-Tags eigene Konverter zu entwickeln.
Wir wollen uns als Beispiel einen Konverter ansehen, der in eingegebenen Zeichenketten Gruppen von Whitespace-Zeichen durch einen Unterstrich ersetzt. Listing**Konverter zum Ersetzen von Zeichen**zeigt die entsprechende Klasse`Replace-Converter`.

```
public class ReplaceConverter implements Converter {
  public Object getAsObject(FacesContext ctx, UIComponent c,
      String value) throws ConverterException {
    if (value == null) return null;
    return value.replaceAll("
s+", "_");
  }
  public String getAsString(FacesContext ctx, UIComponent c,
      Object value) throws ConverterException {
    if (value == null) return null;
    return value.toString();
  }
}
```

Unser Konverter implementiert das Interface`Converter`mit den beiden Methoden`getAsObject()`und`getAsString()`. Nachdem wir vom Benutzer eingegebene Zeichenketten konvertieren wollen, ersetzen wir die Zeichen in der Methode`getAsObject()`. In der Methode`getAsString()`, die beim Rendern aufgerufen wird, geben wir einfach die String-Repräsentation des übergebenen Objekts zurück (es sollte sich ohnehin um einen String handeln).
Ein kleiner Hinweis: Dieser Konverter funktioniert in der oben gezeigten Form nur mit Eingabekomponenten, nicht aber mit Ausgabekomponenten. Warum? Aus dem zuvor erwähnten Grund, dass in`getAsString()`nur die String-Repräsentation des übergebenen Objekts zurückgegeben wird.
Benutzerdefinierte Konverter können auf verschiedene Weise in JSF registriert und eingebunden werden. Eine Möglichkeit ist, einzelne Komponenten einer Ansicht mit Konvertern zu versehen. Das funktioniert bei allen Ein- und Ausgabekomponenten über das Kindelement`f:converter`oder eine Method-Expression im Attribut`converter`. Das Tag`f:converter`wird einfach als Kindelement zum Tag der gewünschten Komponente in der Seitendeklaration hinzugefügt. Dazu muss der Konverter allerdings bereits unter einem eindeutigen Bezeichner im System registriert sein. Dieser Bezeichner wird dann im Attribut`converterId`von`f:converter`eingetragen.
JSF bietet ab Version 2.0 die Möglichkeit, Konverter mit der Annotation`@FacesConverter`im System zu registrieren. Listing**Konverter mit @FacesConverterregistrieren**zeigt, wie der zuvor erstellte Konverter mit dem in der Annotation angegebenen Bezeichner verbunden wird.

```
@FacesConverter("at.irian.ReplaceConverter")
public class ReplaceConverter implements Converter {
  ...
}
```

In JSF vor Version 2.0 erfolgte die Registrierung von Konvertern ausschließlich in der Datei`faces-config.xml`. In JSF 2.0 und neueren Versionen steht diese Variante natürlich immer noch zur Verfügung. Dazu muss für jeden

Konverter ein Element`converter`mit den Kindelementen`converter-id`für den Bezeichner und`converter-class`für die Klasse des Konverters angelegt werden. Listing**Konverter in faces-config.xmlregistrieren**zeigt, wie die zuvor mit der Annotation durchgeführte Registrierung unseres Konverters in der`faces-config.xml`aussieht.

```
<converter>
  <converter-id>at.irian.ReplaceConverter</converter-id>
  <converter-class>
    at.irian.jsfatwork.gui.jsf.ReplaceConverter
  </converter-class>
</converter>
```

JSF bietet noch eine weitere Variante, um einen Konverter mit einer Komponente zu verbinden. Alle Ein- und Ausgabekomponenten verfügen dazu über das Attribut`converter`. In diesem Attribut kann über eine Value-Expression eine Managed-Bean-Eigenschaft vom Typ`javax.faces.convert.Converter`referenziert werden. Die Getter-Methode dieser Eigenschaft muss dann bei jedem Aufruf eine neue Instanz des gewünschten Konverters zurückliefern. Eine Setter-Methode wird hierfür nicht benötigt.
Listing**Benutzerdefinierte Konverter in der Seitendeklaration**zeigt für beide vorgestellten Varianten, wie der Konverter in der Seitendeklaration mit einer Komponente verbunden wird. Beim Einsatz des Attributs`converter`muss es natürlich eine Methode`getReplaceConverter`in der referenzierten Managed-Bean geben.

```
<h:inputText value="#{bean.stringValue}">
  <f:converter converterId="at.irian.ReplaceConverter"/>
</h:inputText>
<h:inputText value="#{bean.otherStringValue}"
    converter="#{bean.replaceConverter}"/>
```

Alternativ bietet es sich an, eine zentrale Managed-Bean für alle benutzerdefinierten Konverter der Anwendung zu erstellen. Diese Bean liegt im Application-Scope und stellt für jeden Konverter eine Getter-Methoden zur Verfügung, die eine neue Instanz des Konverters zurückliefert. Listing**Managed-Bean für Konverter**zeigt, wie eine solche Managed-Bean aussehen könnte.

```
@ManagedBean @ApplicationScoped
public class ConverterProvider {
  public Converter getReplaceConverter() {
    return new ReplaceConverter();
  }
}
```

Die Registrierung eines Konverters mit einem Bezeichner bietet einen entscheidenden Vorteil: Mit Facelets lässt sich das Einbinden benutzerdefinierter Konverter dadurch noch weiter vereinfachen. In Abschnitt**Sektion: Tag-Bibliotheken mit Facelets erstellen**zeigen wir Ihnen, wie Sie auf einfachstem Weg in einer Tag-Bibliothek ein Tag für einen eigenen Konverter definieren.
Neben dem Einbinden benutzerdefinierter Konverter in einzelne Komponenten bietet JSF außerdem die Möglichkeit, einen Konverter global für einen bestimmten Datentyp zu registrieren. Ab JSF 2.0 reicht es dazu aus, das Element`forClass`der Annotation`@FacesConverter`auf den gewünschten Datentyp zu setzen. Listing**Konverter für Datentyp mit @FacesConverter**zeigt das für den Datentyp`Date`.

```
@FacesConverter(forClass = Date.class)
public class MyDateConverter implements Converter {
  ...
}
```

In JSF vor Version 2.0 mussten Konverter für bestimmte Klassen in der`faces-config.xml`registriert werden. Wie das funktioniert, zeigt Listing**Konverter für Datentyp in faces-config.xml**. Wenn unter`converter-for-`

`class`eine Klasse angegeben wird, für die noch kein Konverter existiert, wird ein neuer Konverter registriert. Diese Art der Konfiguration funktioniert natürlich auch weiterhin mit JSF 2.0 und neueren Versionen.

```
<converter>
  <converter-for-class>
    java.util.Date
  </converter-for-class>
  <converter-class>
    at.irian.jsfatwork.MyDateTimeConverter
  </converter-class>
</converter>
```

Im Beispiel*MyGourmet 5*in Abschnitt**[Sektion: *MyGourmet 5*: Konvertierung]**wird ein weiterer benutzerdefinierter Konverter für Postleitzahlen entwickelt und in das Beispiel integriert.

2.11.3 MyGourmet 5: Konvertierung

Das Beispiel*MyGourmet 5*erweitert*MyGourmet 4*aus Abschnitt**[Sektion: *MyGourmet 4*: Phase-Listener und System-Events]**um einige Eingabefelder und zeigt den Einsatz von Standard- und benutzerdefinierten Konvertern.
Beginnend mit diesem Beispiel ändern wir die zugrunde liegende*Backing-Bean*. Bis jetzt hat es eine Bean gegeben, die alle Eigenschaften des Kunden und den Code für die Seite beinhaltet hat. Im Hinblick auf die Erweiterungen im Zuge dieses Abschnitts ist es sinnvoll, diese Aspekte zu trennen. Die Daten des Kunden befinden sich jetzt in der Klasse`Customer`im Package`at.irian.jsfatwork.domain`. Die Managed-Bean`CustomerBean`ersetzt die bisherige Klasse`Customer`im Package`at.irian.jsfatwork.gui.page`. Sie enthält eine Eigenschaft`customer`für den aktuellen Kunden und Code, der für die Seite relevant ist.
Zur Demonstration der Konverter erhält die Klasse`Customer`zusätzlich die Eigenschaften`birthday`,`zipCode`,`city`und`street`. Für die Eingabe des Geburtsdatums auf`editCustomer.xhtml`fügen wir den Standardkonverter`f:convertDateTime`mit einem Muster hinzu. Listing**Eingabekomponente mit Datumskonverter**zeigt das Tag.

```
<h:inputText id="birthday" size="30"
    value="#{customerBean.customer.birthday}">
  <f:convertDateTime pattern="dd.MM.yyyy"/>
</h:inputText>
```

Für die Ausgabe des Geburtsdatums auf`showCustomer.xhtml`wird der gleiche Standardkonverter verwendet. Listing**Ausgabekomponente mit Datumskonverter**zeigt das Tag.

```
<h:outputText
    value="#{customerBean.customer.birthday}">
  <f:convertDateTime pattern="dd.MM.yyyy"/>
</h:outputText>
```

Als Beispiel für einen benutzerdefinierten Konverter wollen wir einen Konverter entwickeln, der die Eingabe von Postleitzahlen mit Ländercodes ermöglicht, ohne dass diese mitgespeichert werden. Dazu werden einfach alle Buchstaben inklusive des ersten Auftretens des Zeichens '-' ignoriert. Der Rest der eingegebenen Zeichenkette muss dann eine Zahl sein. Aus der Eingabe`A-1010`wird dann beispielsweise die Zahl`1010`.
Der neue Konverter`ZipCodeConverter`wird von der JSF-Klasse`IntegerConverter`abgeleitet, was den Vorteil hat, dass die eigentliche Konvertierung der Postleitzahl ohne den Ländercode von diesem erledigt werden kann. Listing**Konverter für Postleitzahlen mit Ländercodes**zeigt den Code der Klasse. Mit der Annotation`@FacesConverter`registrieren wir den Konverter unter dem Bezeichner`at.irian.ZipCode`für die spätere Verwendung im System.

```
@FacesConverter(value="at.irian.ZipCode")
public class ZipCodeConverter
    extends IntegerConverter implements Serializable {
  public Object getAsObject(FacesContext ctx,
      UIComponent component,String value)
      throws ConverterException {
    if (value != null && value.length() > 0) {
      int pos = value.indexOf('-');
      for (int i = 0; i < pos; i++) {
        if (!Character.isLetter(value.charAt(i))) {
          throw new ConverterException("Zip invalid.");
        }
      }
      if (pos > -1 && pos < value.length() - 1) {
        return super.getAsObject(
            ctx, component, value.substring(pos + 1));
      }
    }
    return super.getAsObject(ctx, component, value);
  }
}
```

Interessant ist insbesondere der Teil, der die Fehlerbehandlung übernimmt: Wenn die Postleitzahl nicht dem Muster entspricht, wird eine Exception-Instanz vom Typ`ConverterException`erzeugt. Die Ausnahme wird über ihren Konstruktor mit einer Nachricht verbunden, die dann im Webbrowser als Fehlermeldung ausgegeben wird. Der benutzerdefinierte Konverter`ZipCodeConverter`wird über das Kindelement`f:converter`in`editCustomer.xhtml`eingebunden. Listing**Konverter über Bezeichner einbinden**zeigt das entsprechende Tag.

```
<h:inputText id="zipCode" size="30" required="true"
    value="#{customerBean.customer.zipCode}">
  <f:converter converterId="at.irian.ZipCode"/>
</h:inputText>
```

Nachdem wir jetzt alle Eingaben in die benötigten Datentypen konvertiert haben, wenden wir uns im nächsten Abschnitt der Validierung von Benutzereingaben zu.

2.12 Validierung

Die Eingabe von Daten ist eine Angelegenheit, bei der Benutzer gerne Fehler machen - das Aufzeigen und die Verhinderung der Eintragung fehlerhafter Daten ins Modell muss jedem Entwicklungsframework ein Anliegen sein. JSF bietet "out-of-the-box" bereits viele Möglichkeiten an, solche Fehler zu verhindern.
Für die Überprüfung von Benutzereingaben werden in JSF sogenannte Validatoren eingesetzt. Die Verknüpfung von Validatoren und Eingabekomponenten funktioniert sehr einfach - ähnlich wie bei der Einbindung von Konvertern. Neben einer ganzen Reihe von Standardvalidatoren, die im Lieferumfang von JSF enthalten sind, ist es ohne Weiteres möglich, eigene Validatoren zu schreiben.
Mit JSF ab Version 2.0 gestaltet sich die Validierung durch die Unterstützung des Bean-Validation-Konzepts (JSR-303) sogar noch einfacher. Durch neue Mechanismen ist erstmals eine vollständig metadatenbasierte Validierung ohne Umwege möglich. Abschnitt**[Sektion: Bean-Validation nach JSR-303]**widmet sich diesem hochaktuellen Thema und zeigt, wie Sie Bean-Validation in Kombination mit JSF in eigenen Projekten einsetzen.
Anschließend wenden wir uns in Abschnitt**[Sektion: Standardvalidatoren]**den Standardvalidatoren zu, bevor wir Ihnen in Abschnitt**[Sektion: Benutzerdefinierte Validatoren]**zeigen, wie Sie benutzerdefinierte Validatoren erstellen.

2.12.1 Bean-Validation nach JSR-303

JSF 2.0 war ein großer Schritt vorwärts für die Validierung in Webanwendungen. Davor hat Validierung über die Schichtengrenzen hinweg immer zu Redundanz geführt. In vielen Fällen wurden die gleichen Validierungsregeln in mehreren Schichten der Applikation umgesetzt. Neben einem erhöhten Aufwand bei der Implementierung führt dieser Ansatz auch zu einer höheren Fehleranfälligkeit. Die Validierung wurde dann zum Beispiel sowohl über JSF-Validatoren in der Ansicht als auch beim Persistieren der Entitäten im Service angewandt.
Die Kombination von JSF ab Version 2.0 und Bean-Validation (JSR-303) ändert das radikal. Bean-Validation definiert ein Metadatenmodell und eine Java-API, um die Validierung gezielt in den Domainklassen zu bündeln. Validierungsregeln (Constraints) sind in Form von Validatorklassen implementiert, die mit Annotationen an JavaBeans gebunden werden.
Sehen wir uns am Beispiel von*MyGourmet*an, wie Bean-Validation in der Praxis eingesetzt wird. Existieren für eine Eigenschaft einer Bean, die an eine Eingabekomponente gebunden ist, JSR-303-konforme Metadaten, so wird diese automatisch validiert. Listing**Validierung mit Bean-Validation**zeigt einen Ausschnitt der Klasse`Customer`mit Bean-Validation-Metadaten. Mit der Annotation`@NotNull`wollen wir vermeiden, dass eine der drei Eigenschaften den Wert`null`annimmt. Die Eigenschaft`zipCode`ist zusätzlich mit den Annotationen`@Min`und`@Max`auf einen Wertebereich von 1000 bis 99999 eingeschränkt.

```
public class Customer {
  @NotNull @Min(value = 1000) @Max(value = 99999)
  private Integer zipCode;
  @NotNull
  private String city;
  @NotNull
  private String street;
  ...
}
```

Eine wichtige Voraussetzung ist noch zu beachten: Damit fehlende Benutzereingaben von JavaServer Faces auch wirklich als`null`-Werte interpretiert werden, muss in der`web.xml`der Kontextparameter`javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL`auf`true`gesetzt werden. Standardmäßig ignoriert JSF Eingabekomponenten ohne Benutzereingabe bei der Validierung. Dieses Verhalten macht dann zum Beispiel den Einsatz von`@NotNull`unmöglich.
Das wars! Mehr ist nicht notwendig, um Bean-Validation ab JSF 2.0 zu nutzen, wenn eine Implementierung von JSR-303 in der Laufzeitumgebung verfügbar ist. In*MyGourmet*wird das durch eine Abhängigkeit von*Hibernate Validator v4*, der Referenzimplementierung von JSR-303, in der Maven-Projektdatei`pom.xml`garantiert. Wie Sie sehen, mussten wir keine der Ansichten ändern, um die Validierung zu aktivieren.
Das komplette Beispiel*MyGourmet 6*mit allen Änderungen bezüglich der Validierung finden Sie in Abschnitt**[Sektion: *MyGourmet 6*: Validierung]**.
In der folgenden Aufzählung finden Sie eine Liste aller Standard-Constraints in Bean-Validation:

- `@AssertFalse`
  Das annotierte Element muss`false`sein.
- `@AssertTrue`
  Das annotierte Element muss`true`sein.
- `@DecimalMax`
  Das annotierte Element muss kleiner oder gleich dem in`value`angegebenen Wert sein. Beachten Sie, dass`value`ein String ist, der als`BigDecimal`interpretiert wird (Details finden Sie in der API-Dokumentation zu`BigDecimal`).
- `@DecimalMin`

Das annotierte Element muss größer oder gleich dem in`value`angegebenen Wert sein. Beachten Sie, dass`value`ein String ist, der als`BigDecimal`interpretiert wird (Details finden Sie in der API-Dokumentation zu`BigDecimal`).

- `@Digits`
  Das annotierte Element darf nicht mehr als in`integer`angegebene Stellen vor dem Komma und nicht mehr als in`fraction`angegebene Stellen nach dem Komma haben.
- `@Future`
  Das annotierte Element muss ein Datum in der Zukunft sein.
- `@Max`
  Das annotierte Element muss kleiner oder gleich dem in`value`angegebenen Wert sein. Im Gegensatz zu`@DecimalMax`hat`value`hier den Typ`long`.
- `@Min`
  Das annotierte Element muss größer oder gleich dem in`value`angegebenen Wert sein. Im Gegensatz zu`@DecimalMin`hat`value`hier den Typ`long`.
- `@NotNull`
  Das annotierte Element darf nicht`null`sein.
- `@Null`
  Das annotierte Element muss`null`sein.
- `@Past`
  Das annotierte Element muss ein Datum in der Vergangenheit sein.
- `@Pattern`
  Das annotierte Element muss dem in`regexp`angegebenen regulären Ausdruck entsprechen.
- `@Size`
  Die Größe des annotierten Elements muss zwischen`min`und`max`liegen. Mit diesem Constraint können Strings und alle Collection-Typen validiert werden.

MyFaces ExtVal:JSF bietet leider nicht den vollen Funktionsumfang von Bean-Validation. Alternativ können Sie einen Blick auf das Projekt*MyFaces Extensions Validator*(auch bekannt als ExtVal)Weiterführende Informationen zum Projekt*MyFaces ExtVal*finden Sie unter**http://myfaces.apache.org/extensions/validator**.:werfen, das einen Adapter für JSR-303-konforme Implementierungen zur Verfügung stellt. Dieser Adapter bietet außerdem zusätzliche Funktionalitäten und alternative Konzepte. Beispielsweise wird*MyFaces ExtVal*für die Gruppenvalidierung (als Alternative zur deklarativen Angabe in der Ansicht) Annotationen für die Eigenschaften und Action-Methoden im Backing-Bean zur Verfügung stellen. Dadurch werden Refactorings erheblich einfacher und können zuverlässig durchgeführt werden. Über den Standard hinaus bietet*MyFaces ExtVal*ein eigenes Validierungsmodul mit weiteren Funktionalitäten speziell für JSF. Unter anderem wird komponentenübergreifende Validierung unterstützt.*MyFaces ExtVal*ist auch mit JSF 1.1 und JSF 1.2 einsetzbar.

2.12.1.1 Benutzerdefinierte Constraints mit Bean-Validation

Mit Bean-Validation können Sie sehr einfach eigene Validatoren und Constraints erstellen. Wir wollen dieses Feature in*MyGourmet*nutzen und einen Validator erstellen, der das Geburtsdatum auf einen bestimmten Zeitraum überprüft. Das Datum darf nicht vor dem 1.1.1900 und nicht in der Zukunft liegen.
Alles, was wir dazu brauchen, ist eine Annotation und eine Validatorklasse. Fangen wir mit der Annotation`@Birthday`an. Damit diese Annotation als Constraint für Bean-Validation eingesetzt werden kann, muss sie wiederum mit`@Constraint`annotiert werden. Das Element`validatedBy`stellt die Verbindung zur Validatorklasse, in unserem Fall , her. Eine Constraint-Annotation muss mindestens über die Elemente`message`zur Definition einer Fehlermeldung,`groups`zur Definition der Validatorgruppen und`payload`zum Übergeben zusätzlicher Informationen verfügen. Listing**Annotation für benutzerdefiniertes Constraint**zeigt den Code von`@Birthday`.

```
@Constraint(validatedBy = BirthdayValidator.class)
```

```
@Retention(RUNTIME)
@Target({ANNOTATION_TYPE, METHOD, FIELD})
public @interface Birthday {
  String message() default "Wrong birthday";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Die beiden Annotationen`@Retention`und`@Target`sorgen dafür, dass die Annotation zur Laufzeit ausgewertet wird und auf andere Annotationen, Methoden und Felder angewendet werden kann. Das Element hat einen Defaultwert, der als Fehlermeldung ausgegeben wird, falls er nicht überschrieben wird. Der Defaultwert sollte im Idealfall auf ein Resource-Bundle verweisen, mehr dazu in Abschnitt**[Sektion: Internationalisierung]**.
Zum erfolgreichen Einsatz unseres Constraints fehlt nur noch die Validatorklasse`BirthdayValidator`. Sie muss das Interface`ConstraintValidator`, parametrisiert mit der zuvor definierten Annotation und dem zu validierenden Datentyp, implementieren. Die Methode`initialize`bekommt beim Validierungsvorgang die zugehörige Annotation als Parameter und dient zur Initialisierung des Validators. Die eigentliche Validierung erfolgt dann in der Methode`isValid`, die den zu validierenden Wert und den aktuellen Kontext übergeben bekommt. Im Falle einer erfolgreichen Validierung muss sie`true`zurückliefern. Listing**Validator für benutzerdefiniertes Constraint**zeigt den Sourcecode.

```
public class BirthdayValidator
    implements ConstraintValidator<Birthday, Date> {
  public void initialize(Birthday birthday) {}

  public boolean isValid(Date date,
      ConstraintValidatorContext ctx) {
    boolean dateCorrect = true;
    if (date != null) {
      ctx.disableDefaultConstraintViolation();
      if (date.after(new Date())) {
        ctx.buildConstraintViolationWithTemplate(
            "Birthday is in the future.")
            .addConstraintViolation();
        dateCorrect = false;
      }
      Calendar cal = Calendar.getInstance();
      cal.set(Calendar.YEAR, 1900);
      cal.set(Calendar.MONTH, 0);
      cal.set(Calendar.DAY_OF_MONTH, 1);
      if (date.before(cal.getTime())) {
        ctx.buildConstraintViolationWithTemplate(
            "Birthday is before Jan 1, 1900.")
            .addConstraintViolation();
        dateCorrect = false;
      }
    }
    return dateCorrect;
  }
}
```

Da wir in`isValid`zwei verschiedene Überprüfungen durchführen, wollen wir auch zwei unterschiedliche Fehlermeldungen definieren. Die Standardfehlermeldung soll gar nicht erst angezeigt werden und wird mittels`ctx.disableDefaultConstraintViolation()`abgeschaltet. Jetzt können wir mit folgendem Code beliebige Fehlermeldungen zum Kontext hinzufügen, die dann in JSF-Nachrichten umgewandelt werden:

```
ctx.buildConstraintViolationWithTemplate("Fehlermeldung")
    .addConstraintViolation();
```

Unser benutzerdefiniertes Constraint ist jetzt fertig für den Einsatz. Die Eigenschaft`birthday`der Klasse`Customer`sieht mit der neuen Annotation wie folgt aus:

```
@Birthday
private Date birthday;
```

2.12.2 Standardvalidatoren

Nach der kurzen Einführung in Bean-Validation mit JSF widmen wir uns im Rest des Abschnitts der klassischen JSF-Validierung. Für die Einbindung als Kind-Tag steht bereits im JSF-Standard eine Liste von Validatoren zur Verfügung:

- `f:validateLength`:
  Dieser Validator überprüft die Länge einer Zeichenkette basierend auf den Werten der Attribute`minimum`und`maximum`. Das folgende Beispiel erlaubt die Eingabe einer Zeichenkette mit der minimalen Länge drei und der maximalen Länge sieben:

  ```
  <h:inputText value="#{backingBean.property}">
    <f:validateLength minimum="3" maximum="7"/>
  </h:inputText>
  ```

- `f:validateLongRange`:
  Für die Überprüfung, ob eine Ganzzahl in einem gewissen Bereich liegt, bietet JSF den*LongRange*-Validator an. Auch hier sind die gültigen Attribute`minimum`und`maximum`. Ein Beispiel für die Überprüfung einer Zahl im Bereich 0-1000:

  ```
  <h:inputText value="#{backingBean.property}">
    <f:validateLongRange minimum="0"
          maximum="1000"/>
  </h:inputText>
  ```

- `f:validateDoubleRange`:
  Für die Überprüfung, ob eine Fließkommazahl in einem gewissen Bereich liegt, bietet JSF den*DoubleRange*-Validator an. Wieder sind die gültigen Attribute`minimum`und`maximum`. Ein Beispiel für die Überprüfung einer Zahl im Bereich`0.0`-`1.0`:

  ```
  <h:inputText value="#{backingBean.property}">
    <f:validateDoubleRange minimum="0.0"
          maximum="1.0"/>
  </h:inputText>
  ```

- `f:validateRegex`:
  Dieser Validator überprüft, ob der Zeichenkettenwert dem im Attribut`pattern`angegebenen regulären Ausdruck entspricht. Eine detaillierte Erklärung zu regulären Ausdrücken in Java finden Sie in der Dokumentation zur Java-API. Hier ein kleines Beispiel, bei dem der Wert einer Komponente aus einem Großbuchstaben, gefolgt von einer beliebigen Folge von Buchstaben bestehen muss:

```
<h:inputText value="#{backingBean.property}">
  <f:validateRegex pattern="[A-Z][a-zA-Z]*"/>
</h:inputText>
```

- `f:validateRequired`:
  Dieser Validator überprüft, ob der Benutzer überhaupt einen Wert angegeben hat.

Wie Sie wahrscheinlich bemerkt haben, ist der Validator zum Überprüfen verpflichtender Werte erst ab JSF 2.0 verfügbar. Wie hat das vorher funktioniert? In älteren Versionen wurde der gleiche Effekt über das Setzen des Attributs`required`auf den einzelnen Eingabekomponenten erreicht. Ist das Attribut auf`true`gesetzt, schlägt die Validierung der Komponente bei einem Leerwert fehl. Ab JSF 2.0 können Sie beide Varianten benutzen, der eigene Validator fügt sich allerdings besser in das Validierungskonzept ein.
Bis JSF 2.0 können Sie in benutzerdefinierten Validatoren keine Überprüfung auf`null`durchführen, weil JSF die Validierung erst gar nicht aufruft, wenn das`required`-Attribut einer Komponente auf`false`gesetzt ist und der Benutzer nichts eingegeben hat. Dieses Verhalten war notwendig, weil Validatoren nicht optional gemacht werden konnten.
Ab JSF 2.0 kann dieses Verhalten über den Kontextparameter`javax.faces.VALIDATE_EMPTY_FIELDS`gesteuert werden. Ein Wert von`true`(oder`auto`in Kombination mit Bean-Validation) weist JSF an, auch leere Eingabekomponenten zu validieren.
Validator-Binding:Die Tags der Standardvalidatoren haben ab JSF 1.2 ein zusätzliches Attribut`binding`, über das eine Validatorinstanz mit einer Eigenschaft einer Managed-Bean verknüpft werden kann. Die verknüpfte Eigenschaft muss den Typ`javax.faces.validator.Validator`aufweisen.
Ab JSF 2.0 verfügen alle Standardvalidatoren über das Attribut`dis-abled`, das eine Value-Expression aufnimmt. Wenn der Ausdruck zu`true`evaluiert, kommt der Validator im Lebenszyklus nicht zum Einsatz.
Kombination von Validatoren:Es ist möglich, Validatoren zu kombinieren - also zum Beispiel sicherzustellen, dass eine Zeichenkette angegeben wurde, die zwischen drei und sechs Zeichen lang ist und aus einem Großbuchstaben gefolgt von einer Reihe weiterer Buchstaben besteht (siehe Listing**Kombination von Validatoren**).

```
<h:inputText value="#{backingBean.wert}">
  <f:validateRequired/>
  <f:validateLength minimum="3" maximum="6"/>
  <f:validateRegex pattern="[A-Z][a-zA-Z]*"/>
</h:inputText>
```

2.12.3 Benutzerdefinierte Validatoren

Benötigen Sie Überprüfungen, die von Standardvalidatoren nicht abgedeckt werden, können Sie eigene Validatoren schreiben. Am einfachsten funktioniert das mit Validierungsmethoden. Diese werden mit einer Method-Expression im Attribut`validator`der Eingabekomponente referenziert und müssen die in Listing**Signatur einer Validierungsmethode**dargestellte Signatur aufweisen. Der Name der Methode ist frei wählbar.

```
public void validate(
    FacesContext ctx,
    UIComponent component,
    Object value) throws ValidatorException;
```

Sehen wir uns ein kleines Beispiel im Detail an. Unser benutzerdefinierter Validator soll eine Zeichenkette auf die Werte`ja`und`nein`prüfen und bei allen anderen Eingaben eine Fehlermeldung erzeugen. Listing**Beispiel einer Validierungsmethode**zeigt die entsprechende Methode.

```
public void yesNoValidate(FacesContext ctx, UIComponent comp,
    Object value) throws ValidatorException {
  if (value instanceof String) {
    String strValue = (String) value;
    if (!(strValue.equals("ja")
        && !(strValue.equals("nein")))) {
      throw new ValidatorException(
          new FacesMessage("messageText", null));
    }
  } else {
      throw new ValidatorException(
          new FacesMessage("messageText", null));
  }
}
```

Im Fehlerfall wirft die Methode eine`ValidatorException`mit einer zugeordneten`FacesMessage`. Sie wird dem Benutzer angezeigt. Die Behandlung von Fehlern und Fehlermeldungen in Form von Nachrichten beim Validieren wird in Abschnitt**[Sektion: Nachrichten]**näher erläutert.
Damit diese Methode beim Validierungsvorgang aufgerufen wird, muss sie mit der Eingabekomponente verbunden werden:

```
<h:inputText value="#{backingBean.wert}"
  validator="#{backingBean.yesNoValidate}"/>
```

Wenn neben einer Validierungsmethode noch weitere Validatoren mit Tags in die Komponente eingebunden sind, kommt die Validierungsmethode immer als letzte an die Reihe.
Cross-Component-Validierung:Manche Anwendungsfälle machen Validierungsmethoden erforderlich, die nicht nur den Wert einer Komponente validieren. Die Methode muss dazu mit der letzten zu validierenden Komponente verknüpft sein - nur dann kann auf die konvertierten und validierten Werte aller anderen Komponenten zugegriffen werden. Ein Beispiel für die Cross-Component-Validierung finden Sie in*MyGourmet 6*in Abschnitt**[Sektion: *MyGourmet 6*: Validierung]**.
Benutzerdefinierte Validatoren können auch als eigenständige Klassen implementiert werden, die dann das Interface`javax.faces.validator.Validator`implementieren müssen. In einer Seitendeklaration werden solche Validatoren mit dem Tag`f:validator`als Kindelement in eine Eingabekomponente eingebunden. Das Attribut`validatorId`dieses Tags bezieht sich auf den Bezeichner, unter dem die Validatorklasse im System registriert ist.
Die Registrierung eines Validators kann wie bei Konvertern in der`faces-config.xml`oder ab JSF 2.0 auch mit einer Annotation erfolgen. Listing**Registrierung eines Validators mit Bezeichner**zeigt das entsprechende Fragment der Konfiguration - ist die dazu äquivalente Annotation.

```
<validator>
  <validator-id>at.irian.YesNo</validator-id>
  <validator-class>
    at.irian.jsfatwork.YesNoValidator
  </validator-class>
</validator>
```

JSF 2.2:Ab JSF 2.2 ist das Element`value`der Annotation`@FacesValidator`optional und wird mit einer Namenskonvention ergänzt. Ist es nicht angegeben, benutzt JSF den Klassennamen mit einem kleinen Anfangsbuchstaben als Validator-ID. Für unsere Validatorklasse`YesNoValidator`wäre das die ID`yesNoValidator`.

2.12.4 MyGourmet 6:

Validierung

*MyGourmet 6*erweitert sein Vorgängerbeispiel*MyGourmet 5*aus Abschnitt**[Sektion: *MyGourmet 5*: Konvertierung]**. Der Fokus dieses Beispiels liegt, wie sich aus dem Thema der vorherigen Abschnitte leicht erraten lässt, auf Standard- und benutzerdefinierten Validatoren.
Den Großteil der Validierungsaufgaben in*MyGourmet*erledigt Bean-Validation - wie wir bereits in Abschnitt**[Sektion: Bean-Validation nach JSR-303]**gesehen haben. Listing**MyGourmet 6: Bean Customer mit Annotationen**zeigt alle Eigenschaften der Klasse`Customer`mit den Bean-Validation-Annotationen, jedoch ohne die zugehörigen Getter und Setter.

```
public class Customer {
  @NotNull
  private String firstName;
  @NotNull
  private String lastName;
  @NotNull
  private String email;
  @NotNull @Min(value = 1000) @Max(value = 99999)
  private Integer zipCode;
  @NotNull
  private String city;
  @NotNull
  private String street;
  @Birthday
  private Date birthday;
  private Boolean useCreditCard = Boolean.FALSE;
  @NotNull
  private CreditCardType creditCardType;
  @NotNull
  private String creditCardNumber;
  ...
}
```

Um die Verwendung der JSF-Standardvalidatoren vorzustellen, könnte die Postleitzahl der Adresse in`editCustomer.xhtml`anstatt mit Bean-Validation auch mit dem Tag`f:validateLongRange`auf den Wertebereich 1000 bis 99999 überprüft werden. Listing**MyGourmet 6: Validierung der Postleitzahl mit Standardvalidator**zeigt das zugehörige Fragment der Seite.

```
<h:inputText id="zipCode" size="30" required="true"
    value="#{customerBean.customer.zipCode}">
  <f:converter converterId="at.irian.ZipCode"/>
  <f:validateLongRange minimum="1000" maximum="99999"/>
</h:inputText>
```

Wir wollen Ihnen auch zeigen, wie das Geburtsdatum des Kunden auf klassische Art und Weise validiert wird. Zu diesem Zweck kommt der Validator`BirthdayValidator`(nicht zu verwechseln mit dem Constraint-Validator aus Abschnitt**[Sektion: Bean-Validation nach JSR-303]**!) zum Einsatz, der das Datum auf den Wertebereich überprüft und im System unter dem Name`at.irian.Birthday`registriert ist. Listing**MyGourmet 6: Validator für das Geburtsdatum**zeigt den Code.

```
@FacesValidator(value = BirthdayValidator.VALIDATOR_ID)
public class BirthdayValidator
    implements Validator, Serializable {
  private static final long serialVersionUID = 1L;
  public static final String VALIDATOR_ID = "at.irian.Birthday";

  public void validate(FacesContext ctx, UIComponent component,
      Object value) throws ValidatorException {
```

```
    Date date = (Date)value;
    if (date.after(new Date())) {
      FacesMessage msg = new FacesMessage(
          FacesMessage.SEVERITY_ERROR,
          "Birthday is in the future.", null);
      throw new ValidatorException(msg);
    }
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR, 1900);
    cal.set(Calendar.MONTH, 0);
    cal.set(Calendar.DAY_OF_MONTH, 1);
    if (date.before(cal.getTime())) {
      FacesMessage msg = new FacesMessage(
          FacesMessage.SEVERITY_ERROR,
          "Birthday is before Jan 1, 1900.", null);
      throw new ValidatorException(msg);
    }
  }
}
```

Gibt der Benutzer ein Datum außerhalb dieses Bereichs an, wird eine`ValidatorException`mit einer entsprechenden`FacesMessage`geworfen. Das Einbinden dieses benutzerdefinierten Validators in die Seitendeklaration ist in Listing**MyGourmet 6: Validierung des Geburtsdatums**zu sehen.

```
<h:inputText id="birthday" size="30"
    value="#{customerBean.customer.birthday}">
  <f:convertDateTime pattern="dd.MM.yyyy"/>
  <f:validator validatorId="at.irian.Birthday"/>
</h:inputText>
```

Als Beispiel für eine Cross-Component-Validierung mit einer Validierungsmethode wird die Kreditkartennummer vom gewählten Kartentyp abhängig gemacht. Ist Karte A ausgewählt, muss die Kartennummer vier Zeichen lang sein, ist Karte B ausgewählt, muss sie aus fünf Zeichen bestehen. Die erweiterten Tags für den Typ und die Kartennummer auf der Seite`editCustomer.xhtml`zeigt Listing**MyGourmet 6: Kreditkartendaten mit Validierung**.

```
<h:selectOneListbox id="ccType"
    value="#{customerBean.customer.creditCardType}"
    rendered="#{customerBean.customer.useCreditCard}">
  <f:selectItems value="#{customerBean.creditCardTypes}"/>
  <f:event type="javax.faces.event.PostValidateEvent"
      listener="#{customerBean.postValidateCCType}"/>
</h:selectOneListbox>
<h:inputText id="ccNumber"
    value="#{customerBean.customer.creditCardNumber}"
    rendered="#{customerBean.customer.useCreditCard}"
    validator="#{customerBean.validateCreditNumber}"/>
```

Beachten Sie in Listing**MyGourmet 6: Kreditkartendaten mit Validierung**, dass die Komponenten über das`rendered`-Attribut abhängig vom Wert der Eigenschaft`useCreditCard`ein- oder ausgeblendet werden. Da JSF nur gerenderte Komponenten validiert, können wir damit auch die Validierung steuern.
Für die Validierung der Kreditkartennummer ist die Methode`validateCreditNumber`der Klasse`CustomerBean`zuständig. Die vom Benutzer eingegebene Nummer wird dabei direkt als Argument`value`an die Methode übergeben. Das Auslesen des aktuellen Kreditkartentyps erfordert allerdings etwas mehr Aufwand. Ein einfacher Zugriff auf die Eigenschaft`useCreditCard`des aktuellen Kunden liefert in diesem Fall nicht immer den aktuellen Wert. Der Grund dafür ist einfach: Die Eigenschaft im Modell wird erst nach der erfolgreichen Validierung aller Komponenten aktualisiert und hat zum Zeitpunkt des Aufrufs der Validierungsmethode noch den alten Wert. Zum Auslesen des aktuellen, vom Benutzer eingegebenen Werts gibt

es mehrere Möglichkeiten.
Zum einen kann die Komponente für den Kreditkartentyp mit dem Attribut`binding`an eine Eigenschaft der*Backing-Bean*gebunden werden - ein Vorgang, der auch*Component-Binding*genannt wird. Das Binding ermöglicht den direkten Zugriff auf die Komponente in der Validierungsmethode. Component-Binding kann aber, wenn die Bean wie in unserem Fall im Session-Scope liegt, zu unangenehmen Nebenwirkungen führen, da die Komponente nur einmal erzeugt und dann in der Bean abgelegt wird.
JSF bietet mithilfe des System-Events`PostValidateEvent`eine weitere, sehr elegante Möglichkeit, um den aktuellen Kreditkartentyp auszulesen. Dieses Event wird direkt nach dem Validieren einer Komponente ausgelöst. Wir müssen also nur mit dem Tag`f:event`eine Listener-Methode für dieses Ereignis registrieren (siehe Listing**MyGourmet 6: Kreditkartendaten mit Validierung**) und können dann in dieser Methode auf die Komponente und den aktuellen Wert zugreifen. Listing**MyGourmet 6: System-Event-Listener für Validierung**zeigt die Listener-Methode für das System-Event`PostValidateEvent`in der Klasse`CustomerBean`.

```
public void postValidateCCType(ComponentSystemEvent ev) {
  this.creditCardTypeInput = (UIInput)ev.getComponent();
}
```

In Listing**MyGourmet 6: Validierungsmethode für Kreditkarten-nummer**finden Sie die Implementierung der Validierungsmethode in der Klasse`CustomerBean`. Beachten Sie den Zugriff auf das Feld`creditCardTypeInput`zum Auslesen des aktuellen Kreditkartentyps. Da die Reihenfolge der Validierung einzelner Komponenten in JSF immer durch den Komponentenbaum bestimmt ist, wird dieses Feld garantiert vor dem Aufruf der Validierungsmethode gesetzt.

```
public void validateCreditNumber(FacesContext ctx,
    UIComponent comp, Object value) throws ValidatorException {
  CreditCardType ccType =
      (CreditCardType)creditCardTypeInput.getValue();
  Boolean useCC = customer.getUseCreditCard();
  if (useCC != null && useCC && ccType != null) {
    String ccNumber = (String)value;
    int length;
    if (ccType == CreditCardType.CARD_A) length = 4;
    else length = 5;
    if (!ccNumber.matches("
d{" + length + "}")) {
      String msgText = MessageFormat.format(
          "Card number must consist of {0} digits.", length);
      FacesMessage msg = new FacesMessage(
          FacesMessage.SEVERITY_ERROR, msgText, null);
      throw new ValidatorException(msg);
    }
  }
}
```

2.13 Nachrichten

Der Begriff Nachricht bezeichnet in JSF eine Instanz der Klasse`javax.faces.application.FacesMessage`, die intern für Meldungen aller Art verwendet wird. Nachrichten werden während der Ausführung des Lebenszyklus oder von der Applikation erzeugt und in eine Queue im Faces-Context eingefügt. Gibt es`h:messages`-oder`h:message`-Komponenten auf der Seite, werden die Meldungen gerendert.
Ein Beispiel für die Verwendung von Nachrichten im Lebenszyklus von JSF: Tritt bei der Konvertierung oder Validierung ein Fehler auf, erstellt die betroffene Komponente eine Nachricht und erklärt sich als ungültig. In diesem Fall springt die Ausführung des Lebenszyklus nach der Konvertierung und Validierung direkt zur Render-Response-Phase und die Seite wird mit den Fehlermeldungen erneut angezeigt.
Eine Nachricht kann eine Übersichtsmeldung und Detailinformation aufweisen. Nachrichten müssen aber nicht

immer Fehlermeldungen sein. JSF definiert folgende Schweregrade für`FacesMessage`-Instanzen:

- `SEVERITY_FATAL`
- `SEVERITY_ERROR`
- `SEVERITY_WARN`
- `SEVERITY_INFO`(Standardwert)

In*MyGourmet 6*haben wir Nachrichten in den benutzerdefinierten Konvertern oder Validatoren verwendet. Hier als Beispiel der Code, der eine Fehlermeldung für ein Geburtsdatum in der Zukunft erstellt:

```
if (date.after(new Date())) {
  FacesMessage msg = new FacesMessage(
      FacesMessage.SEVERITY_ERROR,
      "Birthday is in the future.", null);
  throw new ValidatorException(msg);
}
```

Nachrichten können aber auch beim Abarbeiten der Applikationslogik über den Aufruf der Methode`FacesContext.addMessage()`hinzugefügt werden.
Globale und lokale Nachrichten:Im obigen Beispiel wird die Nachricht an den Konstruktor der übergeben. Es handelt sich hier um eine*lokale*Nachricht. Die Nachricht wird beim Abfangen der Exception in JSF direkt mit der betroffenen Komponente verbunden. Wo fällt das ins Gewicht? Wir haben das schon bei den Nachrichtenkomponenten selbst besprochen - für die Ausgabe der Nachrichten gibt es zwei*Tags*: Das eine ist dash`:messages`-Tag, das alle Nachrichten, also sowohl zu einer Komponente gehörende als auch globale Nachrichten, anzeigt, das andere ist das`<h:message for=" komponentenId" />`-Tag, das nur die Nachrichten einer gewissen Komponente darstellt. Mit dem ersten Parameter der`addMessage()`-Methode kann also beeinflusst werden, in welchen Nachrichtenbereichen die Nachricht erscheinen wird.
Die Tags`h:messages`und`h:message`besitzen die Attribute`show-Summary`und`showDetail`. Mit diesen Attributen kann eingestellt werden, ob die Detailinformation oder die Übersichtsmeldung der Nachricht angezeigt wird. Seit Version 1.2 von JSF gibt es für Eingabekomponenten zusätzlich die Attribute`converterMessage`,`requiredMessage`und`validatorMessage`. Mit ihnen lassen sich gezielt Fehlermeldungen einzelner Komponenten mit benutzerdefinierten Zeichenketten überschreiben. Die Namen sind selbsterklärend. Ebenfalls seit JSF 1.2 gibt es das Attribut`label`für Eingabekomponenten, dessen Wert bei Fehlermeldungen statt der ID verwendet wird.
Beim Arbeiten mit JSF ist es sinnvoll, ein`h:messages`-Tag auf der Seite zu haben. Tritt nämlich bei der Konvertierung und Validierung einer Seite ein Fehler auf, wird das Modell nicht verändert und die Applikationslogik nicht ausgeführt - es kann also auch nicht auf eine neue Seite navigiert werden. Ohne dash`:messages`-Tag ist das Auftreten von Fehlern aber für Benutzer (und natürlich auch für Applikationsentwickler) nicht nachvollziehbar. Ab JSF 2.0 wird automatisch ein`h:messages`-Komponente in jeder Seite eingefügt, falls diese noch nicht vorhanden ist - allerdings nur, wenn die Project-Stage auf`Development`gesetzt ist (wie in Abschnitt**Sektion: Project-Stage**gezeigt).
In JSF-Versionen vor 2.0 müssen Sie dash`:messages`-Tag immer selbst in die Seite einbauen.
Der Text der Nachricht wird im Beispiel oben direkt in englischer Sprache angegeben. Nachrichten, die dem Benutzer angezeigt werden, sollten aber auf jeden Fall internationalisiert sein. Abschnitt**[Sektion: Internationalisierung]**zeigt, wie das funktioniert.

2.14 Internationalisierung

Ein wichtiges Thema bei der Entwicklung von Webapplikationen ist die Internationalisierung. Dazu gehört einerseits das Speichern von Zeichenketten unabhängig von der Applikation, damit diese Zeichenketten von Übersetzern in andere Sprachen übertragen werden können. Andererseits kann es je nach Region zu einer anderen Konvertierung von Datumswerten, Zahlen und Währungsangaben kommen. JSF ist dafür gerüstet und macht die Lokalisierung Ihrer Anwendung zum Kinderspiel.

2.14.1

Ermittlung des Lokalisierungscodes

Beim Ausführen einer JSF-Anwendung läuft diese für jeden Anwender mit einem bestimmten Lokalisierungscode, auch`Locale`genannt. Das`Locale`setzt sich aus zwei Angaben zusammen. Einerseits wird mit dem Code die Sprache festgelegt, in der die Anwendung ablaufen soll, andererseits erfolgt die Festlegung eines StaatesEs gibt erhebliche Unterschiede in der Anwendung ein- und derselben Sprache in verschiedenen Staaten.:. Manchmal wird auch in einer dritten Ebene eine Einteilung vorgenommen, beispielsweise nach Dialekten innerhalb der Sprache eines Staates.
Die Definition der unterstützten Lokalisierungscodes einer Applikation erfolgt in der`faces-config.xml`-Datei. Listing**Konfiguration für Internationalisierung**zeigt beispielhaft eine Konfiguration. Die solcherart konfigurierte Applikation unterstützt die Sprachen*Deutsch*,*Englisch*für die USA,*Englisch*für Großbritannien und*Französisch*.

```
<application>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>en_US</supported-locale>
    <supported-locale>en_GB</supported-locale>
    <supported-locale>fr</supported-locale>
  </locale-config>
  <message-bundle>
    at.irian.jsfatwork.messages
  </message-bundle>
</application>
```

Lokalisierung im Request:Welcher Lokalisierungscode jetzt tatsächlich ausgewählt wird, liegt am Benutzer, der sich mit dem System verbindet. Der Browser des Benutzers sendet eine HTTP-Kopfzeile mit der gewünschten Lokalisierung. Wenn die Anwendung diese Lokalisierung unterstützt, wird sie ausgewählt - ansonsten kommt die Standardeinstellung zum Einsatz.
Lokalisierung im View:Will man die Auswahl eines*Locales*nicht von der HTTP-Anfrage abhängig machenBeispiel: Ein deutscher Benutzer sitzt an einem Webclient in einem französischen Internetcafé und will die Benutzerschnittstelle natürlich in seiner Sprache sehen.:, kann der Lokalisierungscode auf einer Seite über die Angabe des*locale*-Attributs am`f:view`-Tag zentral gesetzt werden. Dieses Attribut kann dynamisch an die Eigenschaft einer Managed-Bean gebunden werden. Ein Beispiel:

```
<f:view locale="#{userBean.userLocale}">
    ...
</f:view>
```

2.14.2 Internationalisierung der JSF-Nachrichten

Durch den Eintrag`message-bundle`in der`faces-config.xml`wird festgelegt, wo die Datei für die Nachrichten der Applikation zu finden ist. Es sind also für die Konfiguration aus Listing**Konfiguration für Internationalisierung**folgende Dateien im Package`at.irian.jsfatwork`notwendig, um alle angegebenen Sprachen abzudecken:

- `messages.properties`
- `messages_de.properties`

- `messages_en_US.properties`
- `messages_en_GB.properties`
- `messages_fr.properties`

In diesen Dateien können Sie die Texte für Nachrichten verwalten, die Ihre Anwendung in selbst erstellten Konvertern, Validatoren oder auch Bean-Methoden erzeugt. Des Weiteren können die vom System verwendeten Nachrichten überschrieben werden, wenn das benötigt wird.
Nachdem die Bundles jetzt konfiguriert und erstellt sind, wollen wir sie auch einsetzen. Im Beispiel*MyGourmet 6*haben wir in den benutzerdefinierten Validierungsmethoden im Fehlerfall Nachrichten erzeugt. Der Text ist dort allerdings noch direkt angegeben. Um das zu ändern, erstellen wir eine Hilfsmethode, die eine Instanz der Klasse`FacesMessage`mit einem lokalisierten Text erzeugt. Listing**Internationalisierte Nachrichten in der Backing-Bean**zeigt diese Methode.

```
public static FacesMessage getFacesMessage(
    FacesContext ctx, FacesMessage.Severity severity,
    String msgKey, Object... args) {
  Locale loc = ctx.getViewRoot().getLocale();
  ResourceBundle bundle = ResourceBundle.getBundle(
      ctx.getApplication().getMessageBundle(), loc);
  String msg = bundle.getString(msgKey);
  if (args != null) {
    MessageFormat format = new MessageFormat(msg);
    msg = format.format(args);
  }
  return new FacesMessage(severity, msg, null);
}
```

Der Code ist einfach: Die Methode`getMessageBundle()`liefert den Namen des Bundles - der View-Root das aktuelle Locale. Mit diesen beiden Informationen wird das konkrete Bundle geladen - daraus wird der Text für den angegebenen Schlüssel ausgelesen. Dieser Text wird dann beim Erstellen der Nachricht als Übersichtsmeldung verwendet. Die Detailinformation wird hier nicht angegeben.
Nachrichten, die von Standardkomponenten erzeugt werden, bieten immer eine Übersichtsmeldung und eine Detailinformation. In der`.properties`-Datei existiert daher ein Eintrag mit einem an den Schlüssel angehängten Suffix`_detail`.
Wenn Sie*Apache MyFaces*einsetzen, können Sie auch die Klasse`org.apache.myfaces.shared_impl.util.MessageUtils`zum Anlegen von lokalisierten Nachrichten verwenden. Mit dieser Klasse haben Sie auch den Vorteil, dass Detailinformationen nach dem oben genannten Schema automatisch ausgelesen werden.

2.14.2.1 Nachrichten für Bean-Validation

Wenn Sie eigene Constraints für Bean-Validation erstellen, sollten Sie auch internationalisierte Nachrichten verwenden. Bean-Validation ist ein von JSF unabhängiges System - daher werden die Texte an anderer Stelle definiert. Fehlermeldungen, die in geschwungenen Klammern stehen, werden im Resource-Bundle`ValidationMessages`aufgelöst. Der Text`{validator.msg}`wird zum Beispiel als Schlüssel für das Resource-Bundle interpretiert.
Der Bean-Validator erstellt JSF-Nachrichten standardmäßig in einem etwas anderen Format als andere JSF-Validatoren. Während bei anderen Validatoren die Fehlermeldung mit dem Label der Eingabekomponente und einem Doppelpunkt beginnt, gibt der Bean-Validator den Text aus, der von der Bean-Validation-API zurückkommt. Dieses Verhalten kann einfach geändert werden. Sie müssen dazu die Message-Format-Vorlage des Bean-Validators im Message-Bundle der Applikation austauschen. Fügen Sie dort die Zeilen aus Listing**Bean-Validator-Nachrichten**hinzu. Der Platzhalter`{0}`bezeichnet dabei die eigentliche Fehlermeldung und`{1}`das Label.

```
javax.faces.validator.BeanValidator.MESSAGE={1}: {0}
javax.faces.validator.BeanValidator.MESSAGE_detail=
    {1}: {0}
```

Eine ausführlichere Beschreibung der Internationalisierung von*MyGourmet*mit Beispielen folgt in Abschnitt**[Sektion: *MyGourmet 7*: Internationalisierung]**.

2.14.3 Internationalisierung der Anwendungstexte

Für die anderen Texte in Applikationen (etwa für Beschreibungen oder Labels) ist die Vorgehensweise etwas anders - seit JSF 1.2 gibt es zwei verschiedene Varianten, ein Resource-Bundle zu referenzieren.
Die erste Methode ist das Tag`f:loadBundle`, mit dem ein Resource-Bundle für die Verwendung auf der gleichen Seite verfügbar gemacht wird. Dieses Tag sollte nicht mehr verwendet werden, da es die internationalisierten Daten erst ab der Render-Response-Phase zur Verfügung stellt.
Diese Probleme lassen sich mit der neueren Methode vermeiden. Dabei wird das Resource-Bundle in der`faces-config.xml`mit dem Element`resource-bundle`im Abschnitt`application`referenziert.
Listing**Internationalisierung durch Konfiguration mit resource-bundle**zeigt, wie das Beispiel von vorhin mit dieser Methode umgesetzt wird mit den Vorteilen, dass die Texte in allen Seiten verfügbar sind und dass sie auch mit Ajax funktionieren.

```
<resource-bundle>
  <base-name>de.test.resource.text</base-name>
  <var>text</var>
</resource-bundle>
```

Der Zugriff auf die Elemente des Resource-Bundles ist für beide Methoden identisch und wird über eine Value-Expression im Attribut`value`erledigt. Ein Beispiel:

```
<h:outputText value="#{text.description}"/>
```

Für den ersten Teil der Value-Expression wird der Name der für die`.properties`-Datei vergebenen Variablen eingesetzt und als Attributbezeichnung der Schlüssel des Eintrags in der Datei. Die in JSF verborgene Kraft zeigt sich dadurch, dass man diesen Ausdruck in fast jedem Attribut jedes Tags einsetzen kann - nicht nur im`value`-Attribut.
Für die Internationalisierung von Anwendungen kommt dem`h:outputFormat`-Tag eine besondere Bedeutung zu. Der entscheidende Vorteil gegenüber`h:outputText`ist die Unterstützung von Message-Format-Vorlagen. Dadurch wird es möglich, Texte mit Platzhaltern bereits im Resource-Bundle der Anwendung zu definieren. Die Position der Platzhalter kann dabei sogar an die jeweilige Sprache angepasst werden - ein wichtiger Schritt in Richtung Wartbarkeit und zentraler Internationalisierung von Texten.
Im folgenden Beispiel wird eine Statusmeldung für das Profil des Kunden mit Parametern ausgegeben:

```
<h:outputFormat value="#{msgs.profile_msg}" rendered=
    "#{customerBean.customer.firstName != null}">
  <f:param value="#{customerBean.customer.firstName}"/>
  <f:param value="#{msgs.profile_active}"/>
</h:outputFormat>
```

Im deutschen Resource-Bundle könnten`profile_msg`und`profile_active`wie folgt definiert sein:

```
profile_msg=Ihr Profil ist {1}, {0}.
profile_active=aktiv
```

Bei der englischen Übersetzung könnten sich folgende Zeichenketten ergeben:

```
profile_msg={0}, your profile is {1}.
profile_active=active
```

Die Anpassung des Beispiels an verschiedene Sprachen spielt sich nur im Resource-Bundle der Applikation ab. Mit dem direkten Einsatz der`OutputText`-Komponente ist ein solch flexibles Vorgehen unmöglich.
Zugriff von der Backing-Bean:Wie in der Seitendeklaration kann auch vom Backing-Bean-Code aus auf die internationalisierten Texte zugegriffen werden. Vor JSF 1.2 gab es dafür keine spezifische Lösung - Java stellt den dazu benötigten Code zur Verfügung. In Version 1.2 hat sich das geändert. Mit`resource-bundle`eingebundene Texte können jetzt mit`Applica-tion.getResourceBundle()`direkt über ihren Namen aufgelöst werden. Listing**Internationalisierte Texte in der Backing-Bean**zeigt eine Hilfsmethode, um Texte aus einem Resource-Bundle zu laden.

```
public static String getResourceText(FacesContext ctx,
    String bundleName, String key, Object... args) {
  String text;
  try {
    Application app = ctx.getApplication();
    ResourceBundle bundle = app.getResourceBundle(
        ctx, bundleName);
    text = bundle.getString(key);
  } catch (MissingResourceException e) {
    return "???" + key + "???";
  }
  if (args != null) {
    text = MessageFormat.format(text, args);
  }
  return text;
}
```

Das Beispiel*MyGourmet 7*im nächsten Abschnitt zeigt den praktischen Einsatz von Message- und Resource-Bundle.

2.14.4 MyGourmet 7: Internationalisierung

Das Beispiel*MyGourmet 7*entspricht in der Funktionalität genau dem Vorgängerbeispiel*MyGourmet 6*aus Abschnitt**[Sektion: *MyGourmet 6*: Validierung]**. Die Änderungen beziehen sich nur auf die Internationalisierung der Anwendung.
Der erste Schritt ist die Konfiguration der Lokalisierung in der Datei`faces-config.xml`. In unserem Fall ist Deutsch als Standardwert und Englisch als unterstützte Sprache konfiguriert (Listing**MyGourmet 7: Konfiguration**).

```
<application>
  <locale-config>
    <default-locale>de</default-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
  <message-bundle>
    at.irian.jsfatwork.messages
  </message-bundle>
  <resource-bundle>
    <base-name>
      at.irian.jsfatwork.messages
    </base-name>
```

```
    <var>msgs</var>
  </resource-bundle>
</application>
```

Die Resource-Bundles sind im Package`eat.irian.jsfatwork`unter dem Namen`messages`abgelegt - ein Bundle mit dem Namen`messages_de`für die deutsche Sprache und ein Bundle mit dem Namen`messages_en`für die englische Sprache. Durch unsere Konfiguration des`resource-bundle`-Elements ist das Bundle unter dem Namen`msgs`in allen Seiten der Anwendung in EL-Ausdrücken verwendbar. In Listing**Auszug aus dem deutschen Bundle von MyGourmet**finden Sie einen Auszug aus dem Bundle`messages_de.properties`für die deutsche Sprache.

```
title_main=MyGourmet
title_show_customer=Kunde
title_edit_customer=Kunde bearbeiten
first_name=Vorname
last_name=Nachname
email=E-Mail-Adresse
birthday=Geburtstag
use_credit_card=Kreditkarte angeben
credit_card_type=Kreditkarten-Typ
credit_card_number=Kreditkarten-Nummer
profile_msg=Ihr Profil ist {1}, {0}.
profile_active=aktiv
credit_card_type_CARD_A=Card A
credit_card_type_CARD_B=Card B
validateBirthday.MAXIMUM=Geburtsdatum liegt in der Zukunft.
validateBirthday.MINIMUM=Geburtsdatum ist vor dem 1.1.1900.
validateCreditCardNumber.NUMBER=Kreditkartennummer
    muss aus {0} Ziffern bestehen.
javax.faces.validator.BeanValidator.MESSAGE={1}: {0}
```

Der Zugriff auf die lokalisierten Texte soll exemplarisch anhand des Vornamens des Kunden demonstriert werden. Alle anderen Texte in der Seite werden analog behandelt. Hier das lokalisierte Label für den Vornamen:

```
<h:outputLabel for="firstName" value="#{msgs.first_name}:"/>
```

Es referenziert im`value`-Attribut einen EL-Ausdruck, der sich aus dem Namen des Bundles und dem Schlüssel für den Vornamen zusammensetzt.
Fallweise ist es notwendig, direkt in Java auf lokalisierte Ressourcen zuzugreifen. Die Klasse`GuiUtil`fasst zu diesem Zweck die Methode zum Auslesen eines lokalisierten Textes aus Listing**Internationalisierte Texte in der Backing-Bean**und die Methode zum Erstellen einer lokalisierten Nachricht aus Listing**Internationalisierte Nachrichten in der Backing-Bean**zusammen. In*MyGourmet*kommt`GuiUtil`in`CustomerBean`zum Einsatz. Listing**Einsatz von GuiUtil in MyGourmet**zeigt exemplarisch zwei Methoden aus dieser Managed-Bean.

```
public void validateCreditNumber(FacesContext ctx,
    UIComponent component,
    Object value) throws ValidatorException {
  CreditCardType ccType =
      (CreditCardType)creditCardTypeInput.getValue();
  Boolean useCC = customer.getUseCreditCard();
  if (useCC != null && useCC && ccType != null) {
    String ccNumber = (String)value;
    int length;
    if (ccType == CreditCardType.CARD_A) length = 4;
    else length = 5;
    if (!ccNumber.matches("
d{" + length + "})) {
```

```
        FacesMessage msg = GuiUtil.getFacesMessage(
            ctx, FacesMessage.SEVERITY_ERROR,
            "validateCreditCardNumber.NUMBER", length);
        throw new ValidatorException(msg);
      }
    }
  }

  private String getCCTypeLabel(CreditCardType type) {
    FacesContext c = FacesContext.getCurrentInstance();
    String key = "credit_card_type_" + type.toString();
    return GuiUtil.getResourceText(c, "msgs", key);
  }
```

Die Methode`validateCreditNumber`ist bereits aus*MyGourmet 6*bekannt: Im Unterschied zur vorherigen Version wird hier im Fehlerfall eine lokalisierte Nachricht erstellt. Der Schlüssel des verwendeten Texts lautet`validateCreditCardNumber.NUMBER`. Ein Blick auf Listing**Auszug aus dem deutschen Bundle von MyGourmet**zeigt, dass dieser Text eine Message-Format-Vorlage mit einem Platzhalter ist. Beim Erzeugen der Nachricht wird dieser Platzhalter mit der Länge der Kreditkartennummer ersetzt.
Die Methode`getCCTypeLabel`liefert den lokalisierten Namen für Felder der Enum`CreditCardType`zurück.
Die Resource-Bundles für Bean-Validation sind unter dem Namen`ValidationMessages`abgelegt - ein Bundle mit dem Namen`ValidationMessages_de`für die deutsche Sprache und ein Bundle mit dem Namen`ValidationMessages_en`für die englische Sprache.

3 Standard-JSF-Komponenten

JSF bietet eine gute Auswahl an vordefinierten Komponenten an, die viel Funktionalität zum Erstellen von Benutzeroberflächen mitbringen. Für die meisten Anwendungsfälle beherrschen diese Standardkomponenten das gewünschte Verhalten, angefangen von einfachen Eingabeschaltflächen über Textfelder bis hin zur Darstellung von Daten in Tabellenform.

Alle JSF-UI-Komponentenklassen im Standard erweitern die Klasse`javax.faces.component.UIComponentE` die das grundlegende Verhalten einer UI-Komponente definiert.

Das Einbinden einer JSF-Komponente in eine Seitendeklaration erfolgt in Facelets und JSP über ihr*Tag*. Die Tags für die Standard-JSF-Komponenten und ihre Darstellung als HTML-Ausgabe befinden sich in der*HTML-Custom-Tag-Library*. In der*Core-Tag-Library*finden sich weitere Tags, die Basisaktionen unabhängig von einem speziellen*Render-Kit*bereitstellen.JSF 2.2:In JSF 2.2 haben alle Tag-Bibliotheken neue Namensräume erhalten, die mit`http://xmlns.jcp.org`anstatt`http://java` wie Tabelletab:standard-taglibszeigt. JSF 2.2 unterstützt sowohl die neuen als auch die alten Namensräume. Für eine optimale Zukunftssicherheit empfehlen wir allerdings, auf die neuen Namensräume umzustellen. Zusätzlich sind in der Tabelle die Präfixe dargestellt, mit denen die Namensräume üblicherweise in die Seitendeklaration eingebunden werden.

Name	Namensraum	Präfix
HTML-Custom-Tag-Library	http://xmlns.jcp.org/jsf/html (ab 2.2)	h
	http://java.sun.com/jsf/html (vor 2.2)	h
Core-Tag-Library	http://xmlns.jcp.org/jsf/core (ab 2.2)	f
	http://java.sun.com/jsf/core (vor 2.2)	f

4 Advanced JSF

Nachdem in den vorangegangenen Kapiteln die Grundlagen von JSF im Mittelpunkt standen, wollen wir uns in diesem Kapitel den etwas weiterführenden Themen zuwenden.
Nach einer kurzen Vorstellung der Project-Stage in Abschnitt**[Sektion: Project-Stage]**zeigen wir Ihnen in Abschnitt**[Sektion: Advanced Facelets]**erweiterte Aspekte von Facelets, die hauptsächlich die Wiederverwendung von Inhalten betreffen. Ein zentrales Thema für beinahe jedes Webprojekt ist Templating. Abschnitt**[Sektion: Templating]**zeigt daher ausführlich, wie Templating mit Facelets funktioniert.
Abschnitt**[Sektion: Bookmarks und GET-Anfragen in JSF]**präsentiert anschließend die Unterstützung von GET-Anfragen mit View-Parametern und - neu in JSF 2.2 - View-Actions. Abschließend werfen wir in Abschnitt**[Sektion: Die JSF-Umgebung: Faces-Context und External-Context]**noch einen etwas ausführlicheren Blick auf den Faces-Context und den External-Context, bevor wir das Kapitel mit einigen Details der Konfiguration von JSF in Abschnitt**[Sektion: Konfiguration von JavaServer Faces]**abschließen.
Damit die Praxis nicht zu kurz kommt, werden die vorgestellten Konzepte in den Beispielen*MyGourmet 10*,*MyGourmet 11*und*MyGourmet 12*umgesetzt.

4.1 Project-Stage

Die in JSF 2.0 eingeführte Project-Stage ist an`RAILS_ENV`von*Ruby on Rails*angelehnt und bietet eine Möglichkeit, die aktuelle Phase des Projekts für die Entwicklung bereitzustellen. Die möglichen Werte sind in der Enum`javax.faces.application.ProjectStage`festgelegt und lauten folgendermaßen:

- `Production`(Standardwert)
- `Development`
- `SystemTest`
- `UnitTest`

Sie können die Project-Stage überall dort einsetzen, wo Sie Code abhängig von der aktuellen Projektphase ausführen wollen.
Die Projektphase kann auf folgende Arten auf einen der oben angeführten Werte gesetzt werden:

- Über den Kontextparameter`javax.faces.PROJECT_STAGE`in der`web.xml`
- Über den Namen`java:comp/env/jsf/ProjectStage`mit JNDI

Mit dem Ausschnitt aus der`web.xml`in Listing**Project-Stage in web.xml setzen**wird die Project-Stage zum Beispiel auf den Wert`Development`gesetzt.

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

Zur Laufzeit wird die aktuelle Project-Stage im Application-Objekt abgelegt und kann mit der Methode`getProjectStage()`von dort ausgelesen werden. Listing**Überprüfen der Project-Stage10mm(Variante 1)**zeigt ein kleines Codebeispiel.

```
FacesContext fc = FacesContext.getCurrentInstance();
Application a = fc.getApplication();
```

```
if (a.getProjectStage() == ProjectStage.Development) {
  // Beliebiger Code
}
```

Mit der Hilfsmethode`isProjectStage(ProjectStage)`im`FacesContext`lässt sich das vorherige Codefragment noch weiter vereinfachen. In Listing**Überprüfen der Project-Stage10mm(Variante 2)**finden Sie ein Beispiel.

```
FacesContext fc = FacesContext.getCurrentInstance();
if (fc.isProjectStage(ProjectStage.Development)) {
  // Beliebiger Code
}
```

JSF berücksichtigt die Project-Stage bereits in der Spezifikation an einigen Stellen und das Potenzial für weitere Einsatzgebiete ist groß.
Wenn die Project-Stage auf`Development`gesetzt ist, wird in jede Seite eine`h:messages`-Komponente eingefügt, falls diese nicht vorhanden ist. Damit wird verhindert, dass Validierungsfehler in Formularen unbemerkt bleiben.
Ein weiteres Beispiel findet sich beim Ressourcenmanagement. JSF cacht Ressourcen nur dann, wenn die Project-Stage auf`Production`gesetzt ist. Andernfalls werden sie bei jedem Zugriff neu geladen.
In*MyFaces*steuert die Project-Stage zusätzlich das Überprüfen von Seitendeklarationen auf Änderungen. Ist die Project-Stage auf`Production`gesetzt, aktualisiert Facelets Seitendeklarationen nach dem ersten Aufruf der Seite nicht mehr. Bei allen anderen Project-Stages werden Änderungen nach zwei Sekunden wieder berücksichtigt, was die Entwicklung erheblich vereinfacht. Mehr zu diesem Thema erfahren Sie in Abschnitt**[Sektion: Die Webkonfigurationsdatei web.xml]**bei der Beschreibung des Kontextparameters .

4.2 Advanced Facelets

In allen bisherigen Beispielen haben wir Facelets nur als Seitendeklarationssprache und bessere Alternative zu JSP eingesetzt. Facelets kann aber viel mehr und bietet eine breite Palette an Features, die das Leben eines JSF-Entwicklers einfacher machen. Im Laufe dieses Abschnitts werden wir einige davon vorstellen.
Facelets stellt dazu eine eigene Tag-Bibliothek mit dem Namensraum`http://xmlns.jcp.org/jsf/facelets`bereit. Üblicherweise wird diese Bibliothek mit dem Präfix`ui`in Seitendeklarationen eingebunden. Die wichtigsten Tags daraus werden wir im Rest dieses Abschnitts präsentieren.

4.2.1 Wiederverwendung von Inhalten mit Facelets

Facelets bietet Entwicklern die Möglichkeit, Ansichten modular aufzubauen und wiederkehrende Inhalte an zentraler Stelle zu definieren. Das dafür grundlegende Konzept sind die sogenannten Kompositionen, die einen Teil eines Komponentenbaums gruppieren.
Facelets kann eine Ansicht aus einer beliebigen Anzahl von Kompositionen aufbauen, die jeweils in einem eigenen XHTML-Dokument deklariert sind. Trifft Facelets beim Aufbau des Komponentenbaums auf ein`ui:include`-Tag , wird das Dokument mit dem im Attribut`src`angegebenen Dateinamen in die Ansicht mit aufgenommen. Der Pfad des Dokuments kann absolut oder relativ zur aktuellen Ansicht angegeben sein.
Sehen wir uns das im Kontext von*MyGourmet*an. Listing**Fragment für einen Seitenkopf**definiert eine Komposition für einen Seitenkopf, der unter`/WEB-INF/includes/header.xhtml`abgelegt ist.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<head><title>MyGourmet header</title></head>
<body>
  <ui:composition>
    <h:panelGroup style="width: 100; height: 40px;"
        layout="block">
      <h:graphicImage value="/images/logo.png"
          style="float: left;"/>
      <h1 style="display: inline; margin-left: 5px;">
        #{msgs.title_main}
      </h1>
    </h:panelGroup>
    <h2>#{pageTitle}</h2>
  </ui:composition>
</body>
</html>
```

In Listing**Einfügen des Seitenkopfs mit ui:include**sehen Sie den Ausschnitt der Seitendeklaration`showCustomer.xhtml`mit dem über`ui:include`eingefügten Seitenkopf. Das Tag`ui:param`übergibt den Text für die Überschrift zweiter Ordnung als Parameter an das eingefügte Seitenfragment - doch dazu später mehr.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
  <head>
    <title>#{msgs.title_main}</title>
  </head>
  <body>
    <ui:include src="/WEB-INF/includes/header.xhtml">
      <ui:param name="pageTitle"
          value="#{msgs.title_show_customer}"/>
    </ui:include>
    ...
  </body>
</html>
```

Wie funktioniert in Facelets die Zusammensetzung der Deklaration`showCustomer.xhtml`mit dem eingefügten Fragment`header.xhtml`? Wie Sie vielleicht bemerkt haben, handelt es sich bei beiden Dateien um komplette XHTML-Dokumente. Es soll allerdings nur ein einziges Dokument an den Browser geschickt werden. Des Rätsels Lösung liegt darin, wie Facelets das Tag`ui:composition`behandelt - es ignoriert beim Einfügen des Seitenfragments sämtliche Inhalte außerhalb des Tags`ui:composition`.
Nachdem Facelets das HTML-Grundgerüst des Seitenfragments in Listing**Fragment für einen Seitenkopf**sowieso ignoriert, kann es auch entfernt werden. Das Wurzelelement des XHTML-Dokuments ist nicht länger`html`, sondern`ui:composition`. Streng genommen handelt es sich dann nicht mehr um ein XHTML-Dokument, was aber Facelets nichts ausmacht. Listing**Fragment für einen Seitenkopf**zeigt die optimierte Variante von`header.xhtml`.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
  <h:panelGroup style="width: 100; height: 40px;"
      layout="block">
    <h:graphicImage value="/images/logo.png"
        style="float: left;"/>
    <h1 style="display: inline; margin-left: 5px;">
      #{msgs.title_main}
```

```
    </h1>
  </h:panelGroup>
  <h2>#{pageTitle}</h2>
</ui:composition>
```

Der Parameter`pageTitle`ermöglicht eine individuelle Definition der Überschrift bei jedem Einfügen des Fragments. Werfen Sie nochmals einen Blick auf Listing**Fragment für einen Seitenkopf**, dann sehen Sie die Verwendung dieses Parameters. Mit dem EL-Ausdruck`#{pageTitle}`wird der Wert direkt im`h2`-Element ausgewertet.
Das`ui:component`-Tag bietet die gleiche Funktionalität wie das Tag`ui:composition`- im Komponentenbaum wird aber zusätzlich eine Wurzelkomponente für die Komponentengruppe eingefügt.
Die hier gezeigte Vorgehensweise ist die einfachste Form, Komponentenbäume in Facelets aus mehreren Kompositionen aufzubauen. Wir werden Ihnen im Laufe der nächsten Abschnitte noch weitere Möglichkeiten zeigen, Seitendeklarationen modular aufzubauen.

4.2.2 Tag-Bibliotheken mit Facelets erstellen

Wir haben mittlerweile mit der Core-, der HTML- und der Facelets-Tag-Library drei verschiedene Tag-Bibliotheken kennengelernt. Jede von ihnen bietet unter einem im System eindeutigen Namensraum verschiedenste Tags zum einfachen Aufbau von Seitendeklarationen an. Wie wäre es, wenn Sie für Ihre eigenen Komponenten, Konverter und Validatoren auch eigene Tags definieren könnten? Das würde die tägliche Arbeit mit JSF doch erheblich vereinfachen. Facelets bietet auch dafür eine einfache Lösung an.
Eine benutzerdefinierte Tag-Bibliothek erlaubt die Definition von Tags für eigene Komponenten, Konverter und Validatoren. Wie die Tag-Bibliotheken der Standardkomponenten hat auch jede benutzerdefinierte Tag-Bibliothek einen im System eindeutigen Namensraum, mit dem sie in jede Seitendeklaration eingebunden werden kann. Neben Tag-Definitionen kann eine Tag-Bibliothek auch sogenannte EL-Funktionen enthalten, mit denen statische Funktionen in EL-Ausdrücken verfügbar gemacht werden.
Nachdem wir bis jetzt noch keine eigenen Komponenten erstellt haben, werden wir mit der Definition eines entsprechenden Tags noch bis Kapitel**Kapitel: Die eigene JSF-Komponente**warten. In Abschnitt**Sektion: Die eigene Komponentenbibliothek**finden Sie sogar eine kurze Anleitung zum Aufbau einer eigenen Komponentenbibliothek. Wir wollen den folgenden Abschnitt mit der Definition einer EL-Funktion beginnen, wobei wir Ihnen auch gleich zeigen, wie Sie eine Tag-Bibliothek erstellen und im System registrieren können. Was wir Ihnen überdies nicht vorenthalten wollen, ist das Erstellen eines Tags für einen Konverter und einen Validator.

4.2.2.1 Definition einer EL-Funktion

*JavaServer Pages*ab Version 2.1 und Facelets bieten die Möglichkeit, statische Funktionen in EL-Ausdrücken verfügbar zu machen - und das mit einer beliebigen Anzahl von Parametern. Nachdem die Beispiele im Buch Facelets als Seitendeklarationssprache einsetzen, werden wir uns an dieser Stelle auf die Definition einer EL-Funktion mit Facelets beschränken. In JSP funktioniert die Definition allerdings sehr ähnlich.

Tipp:EL-Funktionen sind mit der neuen Version der*Unified-EL*in*Java EE 6*oft nicht mehr notwendig, da beliebige Methoden - auch mit Parametern - aufgerufen werden können (siehe Abschnitt**Sektion: Erweiterungen der Unified-EL in Java EE 6**).

Als Beispiel implementieren wir eine Funktion, die für ein Geburtsdatum das Alter berechnet und als Zahl zurückliefert. Der dazu notwendige Java-Code beschränkt sich auf wenige Zeilen in der statischen Methode`getAge`der Klasse`MyGourmetUtil`. Diese Klasse ist in Listing**Java-Code der EL-Funktion**zu sehen.

```
public class MyGourmetUtil {
  public static int getAge(Date birthday) {
    Calendar birthCal = Calendar.getInstance();
```

```
    birthCal.setTime(birthday);
    Calendar today = Calendar.getInstance();
    int age = today.get(Calendar.YEAR)
        - birthCal.get(Calendar.YEAR);
    if (today.get(Calendar.DAY_OF_YEAR)
        < birthCal.get(Calendar.DAY_OF_YEAR))
      age--;
    return age;
  }
}
```

Die hinter der EL-Funktion liegende Methode ist damit vorhanden, jetzt muss sie noch in einer Tag-Bibliothek verfügbar gemacht werden. Listing**Tag-Bibliothek mit einer EL-Funktion**zeigt die Tag-Bibliothek`mygourmet.taglib.xml`mit der Definition der EL-Funktion in einem`function`-Element.

```
<facelet-taglib 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-facelettaglibrary_2_2.xsd">
  <namespace>http://at.irian/mygourmet</namespace>
  <function>
    <function-name>getAge</function-name>
    <function-class>
      at.irian.jsfatwork.gui.util.MyGourmetUtil
    </function-class>
    <function-signature>
      int getAge(java.util.Date)
    </function-signature>
  </function>
</facelet-taglib>
```

Der Name, unter dem die Funktion später in EL-Ausdrücken einsetzbar ist, wird im Kindelement`function-name`angegeben. Die Klasse und die aufzurufende Methode werden in den Elementen`function-class`und`function-signature`definiert. Sie müssen in beiden Werten qualifizierte Namen verwenden, damit die jeweiligen Klassen gefunden werden.
Das Einbinden der Tag-Bibliothek in die Anwendung erfolgt mit dem Kontextparameter`javax.faces.FACELETS_LIBRARIES`in der`web.xml`. Facelets interpretiert den Wert dieses Parameters als eine über Semikolons separierte Liste von Tag-Bibliotheken. Nach der Registrierung ist die Tag-Bibliothek über die im Element`namespace`definierte URI`http://at.irian/mygourmet`im System verfügbar. Listing**Tag-Bibliothek in der web.xml einbinden**zeigt den Ausschnitt der`web.xml`-Datei.

```
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>/WEB-INF/mygourmet.taglib.xml</param-value>
</context-param>
```

Facelets bindet Tag-Bibliotheken aus Jar-Dateien im Classpath automatisch ein, wenn sie im`META-INF`-Verzeichnis liegen und ihr Dateiname mit`.taglib.xml`endet.
Dem Einsatz der EL-Funktion steht jetzt nichts mehr im Weg. Die benutzerdefinierte Tag-Bibliothek`mygourmet.taglib.xml`wird ähnlich den bestehenden Tag-Bibliotheken eingebunden. In Listing**EL-Funktion im Einsatz**sehen wir die Ausgabe des Alters unter Zuhilfenahme unserer Funktion. Bitte beachten Sie, dass beim Aufruf der Funktion das Präfix`mg:`mit angegeben werden muss.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
```

```
    xmlns:mg="http://at.irian/mygourmet">
  ...
  <h:outputText value=
      "#{mg:getAge(customerBean.customer.birthday)}"/>
  ...
</html>
```

4.2.2.2 Definition eines Konverter-Tags

In Abschnitt**Sektion: Benutzerdefinierte Konverter**haben wir bereits einen Konverter für die Postleitzahl definiert und unter dem Bezeichner`at.irian.ZipCode`registriert. Eingebunden wurde dieser Konverter mit`f:converter`unter Angabe dieses Bezeichners. Schön wäre es, wenn ein eigenes Tag mit einem sprechenden Namen und der Möglichkeit, Attribute an diesen Konverter zu übergeben, existieren würde - nichts einfacher als das.
Das Hinzufügen der Zeilen in Listing**Definition eines Konverter-Tags**zu unserer Bibliothek reicht aus, um den Konverter unter dem Tag`convertZipCode`zur Verfügung zu stellen. Beim Aufbau des Komponentenbaums fügt Facelets dann für jedes Tag mit dem Namen`convertZipCode`aus unserer Bibliothek den Konverter mit dem Bezeichner`at.irian.ZipCode`ein.

```
<tag>
  <tag-name>convertZipCode</tag-name>
  <converter>
    <converter-id>at.irian.ZipCode</converter-id>
  </converter>
</tag>
```

Listing**Einsatz des benutzerdefinierten Konverter-Tags**zeigt das neue Tag in einer Seitendeklaration. Als Voraussetzung gilt auch hier, dass die Tag-Bibliothek in der Deklaration unter dem Präfix`mg`bekannt gemacht wurde.

```
<h:inputText id="zipCode" size="30"
    value="#{addressBean.address.zipCode}">
  <mg:convertZipCode/>
</h:inputText>
```

In*MyGourmet 12*(siehe Abschnitt**[Sektion: *MyGourmet 12*: GET-Unterstützung]**) erstellen wir einen weiteren benutzerdefinierten Konverter mit eigenem Tag zum Umwandeln von*Collections*in Zeichenketten. Im nächsten Abschnitt über Validatoren zeigen wir Ihnen, wie auch Attribute übergeben werden können.

4.2.2.3 Definition eines Validator-Tags

Der Vorgang der Definition eines Tags für einen Konverter funktioniert in exakt der gleichen Weise auch für Validatoren. Obwohl in*MyGourmet*die Validierung über Bean-Validation abgewickelt wird, werden wir hier einen Validator für das Alter einer Person registrieren. Der Validator soll über die beiden optionalen Eigenschaften`minAge`und`maxAge`steuerbar sein.
Listing**Definition eines Validator-Tags**zeigt die Zeilen für die Definition des Validator-Tags. Der interessante Aspekt an diesem Validator sind die beiden Eigenschaften`minAge`und`maxAge`.

```
<tag>
  <tag-name>validateAge</tag-name>
  <validator>
    <validator-id>at.irian.Age</validator-id>
  </validator>
</tag>
```

Die Werte der beiden Eigenschaften können direkt über Attribute des Tags an den Validator übergeben werden.

Facelets verknüpft diese dann automatisch mit gleichnamigen Eigenschaften der dahinterliegenden Validator-Objekte. In Listing**Einsatz des benutzerdefinierten Validator-Tags**sehen Sie das Tag`mg:validateAge`mit gesetztem Attribut`minAge`im Einsatz.

```
<h:inputText id="birthday" size="30"
    value="#{customerBean.customer.birthday}">
  <f:convertDateTime pattern="dd.MM.yyyy"/>
  <mg:validateAge minAge="18"/>
</h:inputText>
```

4.2.3 MyGourmet 10: Advanced Facelets

Das Beispiel*MyGourmet 10*fasst alle Änderungen aus Abschnitt**[Sektion: Advanced Facelets]**zusammen. Ein Großteil der Neuerungen hat direkt oder indirekt mit der neuen Tag-Bibliothek`/WEB-INF/mygourmet.taglib.xml`zu tun, die unter dem Namensraum`http://at.irian/mygourmet`in der Anwendung verfügbar ist.
Alle Ansichten haben jetzt einen einheitlichen Seitenkopf, der über`mg:pageHeader`eingebunden ist. Genauso gut wäre es möglich, direkt das dahinterliegende Seitenfragment`header.xhtml`aus dem Verzeichnis`/WEB-INF/includes`über`ui:include`zu verwenden.
Der Konverter für die Postleitzahl in`editAddress.xhtml`und der Validator für das Alter des Kunden in`editCustomer.xhtml`sind jetzt direkt über Tags aus der neuen Bibliothek eingebunden. In der Ansicht`showCustomer.xhtml`wird zusätzlich das Alter der Person über die EL-Funktion`mg:getAge`ausgegeben.

4.3 Templating

Layout und Design spielen bei der Entwicklung vieler Webanwendungen eine wichtige Rolle. Neben einem ausgefeilten grafischen Design ist eine konsistente und durchgängige Seitenstruktur oft die Grundvoraussetzung für den Erfolg einer Applikation. Ein einheitliches Seitenlayout vereinfacht nicht nur die Bedienbarkeit für den Benutzer, sondern ermöglicht auch die konsequente Umsetzung eines Unternehmensdesigns (Corporate Identity) auf allen Seiten. Diese Anforderungen lassen sich in der Entwicklung mithilfe von Templates umsetzen.
Der Einsatz von Templates verringert nicht nur die Redundanz der erstellten Anwendung, sondern bietet auch entscheidende Vorteile während der Entwicklung. Templates fördern durch den modularen Aufbau der Seiten die Wiederverwendung von Code und erleichtern die Trennung von Design und Inhalt. Diese Entkopplung unterstützt eine konsequente Durchsetzung des Designs im gesamten Projekt und schwächt die Auswirkung von nachträglichen Änderungen ab. Im Idealfall muss dann nur das Template oder ein zentral definiertes Seitenfragment angepasst werden, was Entwicklungs- und Wartungskosten spart.
Facelets bietet eine sehr elegante Templating-Lösung, die perfekt in den JSF-Lebenszyklus integriert ist. Ein Template ist in Facelets in erster Linie eine XHTML-Datei - wie jede andere Seitendeklaration. Den Unterschied macht das Tag`ui:insert`aus der*Facelets-Tag-Library*, mit dem ersetzbare Bereiche im Template definiert werden können. Eine Seitendeklaration, die auf diesem Template aufbaut (der sogenannte Template-Client), kann diese Bereiche mit dem eigentlichen Content ersetzen. Die komplette Ansicht besteht dann aus dem im Template definierten Inhalt und den ersetzten Bereichen aus dem Template-Client.
Sehen wir uns nun anhand eines kleinen Beispiels an, wie das Templating mit Facelets in der Praxis aussieht. Die Seiten dieses Beispiels sollen über eine Kopfzeile, einen Content-Bereich und eine Fußzeile verfügen. Diese Anforderung setzen wir in Form eines Templates mit der entsprechenden Struktur und drei ersetzbaren Bereichen um. Dadurch ist das Layout zentral definiert und einfach auf alle Seiten anwendbar. Das entsprechende Template mit dem Name`template.xhtml`ist in Listing**Beispiel eines Templates in Facelets**zu finden.

```
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<head>
  <title>MyGourmet</title>
  <link rel="stylesheet" type="text/css" href="style.css"/>
</head>
<body>
  <div id="header">
    <ui:insert name="header">
      <h1>MyGourmet</h1>
    </ui:insert>
  </div>
  <div id="content">
    <ui:insert name="content"/>
  </div>
  <div id="footer">
    <ui:insert name="footer">
      <h:outputText value="Copyright (c) 2012"/>
    </ui:insert>
  </div>
</body>
</html>
```

Das Template ist ein einfaches XHTML-Dokument, in dem die grundlegende Seitenstruktur mit`div`-Elementen abgebildet ist. Die drei`ui:insert`-Tags mit den Namen`header`,`content`und`footer`definieren die ersetzbaren Bereiche. Bei den`ui:insert`-Bereichen für die Kopf- und die Fußzeile nutzen wir die Möglichkeit, Default-Content zu definieren. Falls ein Template-Client den entsprechenden Bereich nicht überschreibt, fügt Facelets den Inhalt innerhalb des`ui:insert`-Tags in die Ausgabe ein. Diese Vorgehensweise ist besonders dann praktikabel, wenn der Inhalt in weiten Teilen der Applikation gleich bleibt.
Im nächsten Schritt werden wir die Seite`showCustomer.xhtml`erstellen. Sie basiert auf unserem Template und definiert einen eigenen Content-Bereich. Facelets bietet dafür die Tags`ui:composition`und`ui:define`an.`ui:composition`stellt eine Verbindung zum Template mit dem im Attribut`template`angegebenen Namen her - in unserem Fall`template.xhtml`. Innerhalb von`ui:composition`können die Zielbereiche des Templates mit`ui:define`-Blöcken überschrieben werden. Welcher mit`ui:insert`definierte Bereich des Templates durch den`ui:define`-Block ersetzt wird, bestimmt das Attribut`name`.
Bevor wir etwas genauer analysieren, wie Facelets eine Ansicht mit einem Template rendert, werfen wir in Listing**Beispiel eines Template-Clients in Facelets**noch einen Blick auf den kompletten Template-Client`showCustomer.xhtml`. Auch hier haben wir auf das XHTML-Grundgerüst verzichtet und direkt das Tag`ui:composition`als Wurzelelement verwendet.

```
<ui:composition template="template.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
  <ui:define name="content">
    <h2>Kundendaten</h2>
    <h:panelGrid id="grid" columns="2">
      <h:outputText value="Vorname:"/>
      <h:outputText value="#{customer.firstName}"/>
      <h:outputText value="Nachname:"/>
      <h:outputText value="#{customer.lastName}"/>
    </h:panelGrid>
  </ui:define>
</ui:composition>
```

Wie baut Facelets die Ansicht aus`showCustomer.xhtml`mit dem Template auf? Wie schon beim Einsatz von`ui:include`ignoriert Facelets auch hier sämtliche Inhalte außerhalb`ui:composition`und der Aufbau des Komponentenbaums beginnt mit dem referenzierten Template. Während des Seitenerstellungsvorgangs werden

die mit`ui:insert`definierten Bereiche im Template ersetzt. In unserem Beispiel kommt der Inhalt der Kopf- und Fußzeile aus dem Template und der Inhalt des Content-Bereichs stammt aus dem`ui:define`-Block in`showCustomer.xhtml`.
Abbildung**Ersetzbare Bereiche des Templating-Beispiels**zeigt die ersetzbaren`ui:insert`-Bereiche des Templates anhand der gerenderten Ausgabe unseres Beispiels. Die Umrahmungen mit den Namen der einzelnen Teile in der linken oberen Ecke dienen nur der besseren Visualisierung und wurden nicht von JSF gerendert.

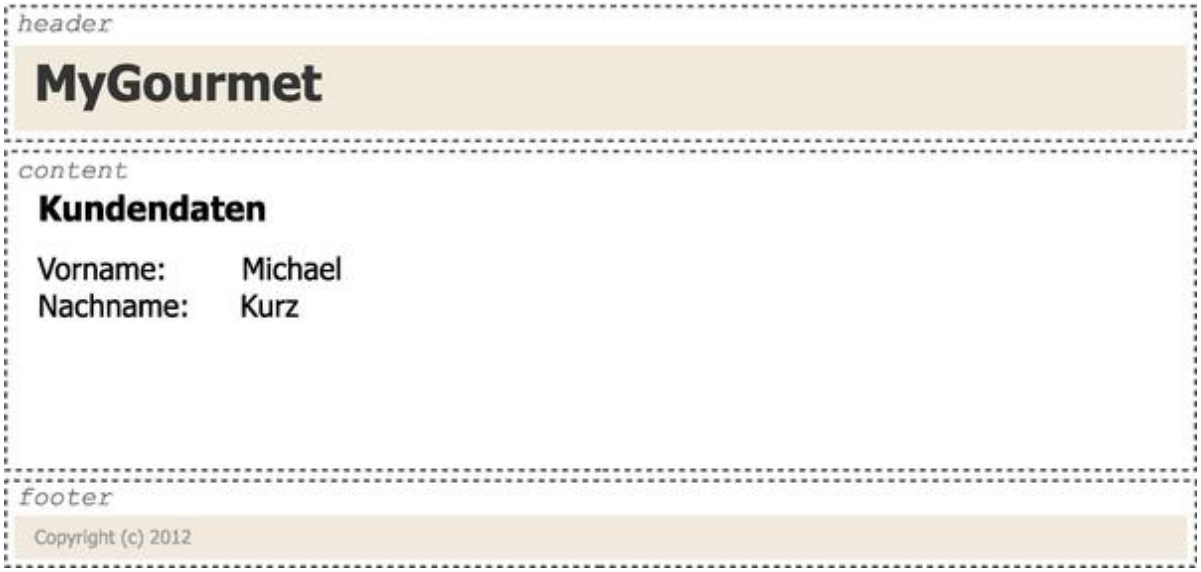


Abbildung:Ersetzbare Bereiche des Templating-Beispiels

Facelets bietet für das Templating noch einiges mehr als die im letzten Abschnitt beschriebene Basisfunktionalität. Nach der Vorstellung von mehrstufigem Templating in Abschnitt**[Sektion: Mehrstufiges Templating]**werden wir in Abschnitt**[Sektion: Mehrere Templates pro Seite]**einen Blick auf den Einsatz von mehreren Templates in einem Template-Client werfen.

4.3.1 Mehrstufiges Templating

Mehrstufiges Templating ermöglicht den Aufbau einer Hierarchie von Templates. Das ist besonders dann praktisch, wenn eine Anwendung in mehrere Bereiche gegliedert ist, die ein gemeinsames Layout, aber unterschiedliche Inhalte haben. Im Fall von*MyGourmet*wäre das zum Beispiel ein Kundenbereich zum Bestellen von Gerichten, ein Bereich für Restaurants und Anbieter und ein allgemeiner Administrationsbereich. Das grundlegende Layout der Seiten mit Kopfzeile, linker Seitenleiste, Content-Bereich und Fußzeile bleibt gleich und wird daher im Haupttemplate aufgebaut. Dort landet auch ein Standardwert für den Inhalt der Kopf- und Fußzeile. In den abgeleiteten Templates wird die linke Seitenleiste überschrieben und mit bereichsspezifischem Inhalt gefüllt. Der Content-Bereich bleibt weiterhin leer und wird erst in den konkreten Seiten überschrieben.
Abbildung**Templating-Hierarchie von MyGourmet**zeigt die mehrstufige Templating-Hierarchie in*MyGourmet*inklusive der bereits bekannten Seite`showCustomer.xhtml`aus dem Kundenbereich.

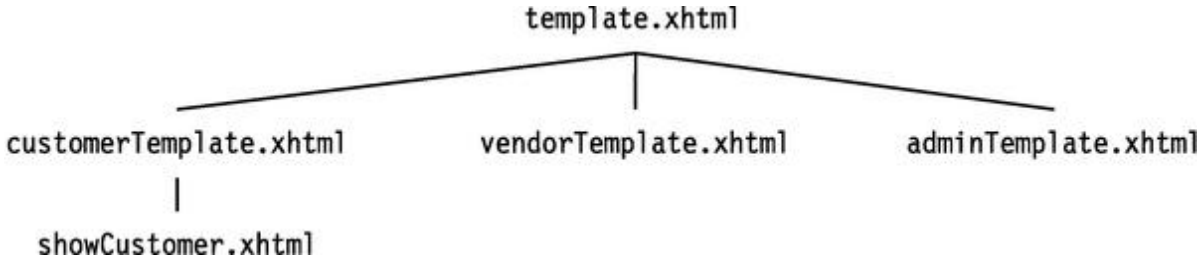


Abbildung:Templating-Hierarchie von MyGourmet

Der Aufbau einer mehrstufigen Templating-Hierarchie gestaltet sich einfach, da jeder Template-Client wiederum die Rolle eines Templates einnehmen kann - in Facelets gibt es keine strikte Trennung zwischen diesen beiden Rollen. Die oben erwähnten und in Abbildung**Templating-Hierarchie von MyGourmet**ersichtlichen Templates für die Anwendungsbereiche nehmen beide Rollen ein. Einerseits sind sie Template-Clients, da sie mit folgendem Code das Haupttemplate referenzieren:

```
<ui:composition template="template.xhtml">
```

Für die konkreten Seiten im Anwendungsbereich sind sie allerdings Templates, die neben den geerbten Inhalten aus dem Haupttemplate den Inhalt der linken Seitenleiste deklarieren. In der Seite wird das Template dann beispielsweise mit folgendem Code referenziert:

```
<ui:composition template="customerTemplate.xhtml">
```

Eine genauere Betrachtung des mehrstufigen Templatings in der Praxis folgt mit Beispiel*MyGourmet 11*in Abschnitt**[Sektion: *MyGourmet 11*: Templating mit Facelets]**.

4.3.2 Mehrere Templates pro Seite

In manchen Fällen macht es Sinn, neben einem Template für die Ansicht selbst zusätzliche Templates für wiederkehrende Bereiche der Seite zu verwenden. Denken Sie zum Beispiel an speziell gestaltete Bereiche in einer Seitenleiste oder an Vorlagen für unterschiedliche Typen von Seiteninhalten. Die bereits bekannte Methode mit`ui:composition`führt in diesem Fall nicht zum Erfolg, da der Content außerhalb des Tags abgeschnitten wird. Bei zwei geschachtelten`ui:composition`-Tags mit gesetztem`template`-Attribut gewinnt immer das innere - dieser Ansatz ist also für unsere Zwecke nicht brauchbar.
Facelets bietet aber auch dafür eine Lösung an. Mit`ui:decorate`existiert eine Variante von`ui:composition`, bei deren Verwendung der außerhalb des Tags liegende Code nicht abgeschnitten wird. Wie der Name bereits sagt, wird der Content innerhalb von`ui:decorate`mit dem Inhalt des referenzierten Templates dekoriert.
Sehen wir uns das anhand eines kleinen Beispiels an. Listing**Template-Client mit mehreren Templates**zeigt einen Ausschnitt aus dem Quelltext eines Template-Clients, der ein Template für die Seite und eines für eine Box in der Seitenleiste beinhaltet.

```
<ui:composition template="template.xhtml">
  ...
  <ui:define name="left_sidebar">
    <ui:decorate template="sideBox.xhtml">
        <ui:param name="title" value="Meldungen"/>
      <h:outputText value="#{bean.msg}"/>
    </ui:decorate>
  </ui:define>
  ...
</ui:composition>
```

Innerhalb von`ui:decorate`wird dem referenzierten Template`sideBox.xhtml`mit dem Tag`ui:param`der Parameter mit dem Name`title`übergeben. Der restliche Inhalt von`ui:decorate`bildet den Inhalt der Box.
Das Template für die Box ist ein XHTML-Dokument mit einer kleinen Besonderheit. Da es sich um ein Template für einen Teil der kompletten Seite handelt, darf das HTML-Grundgerüst nicht gerendert werden. Dazu dient das Tag`ui:composition`- diesmal allerdings ohne das Attribut`template`. Auf diese Weise eingesetzt definiert es einen Teilkomponentenbaum bestehend aus seinem Inhalt. Alle Elemente außerhalb werden abgeschnitten.
Der im Template-Client gesetzte Parameter`title`ist im Template als Variable verfügbar und wird über einen EL-Ausdruck referenziert. Der Inhalt der Box steht direkt im`ui:decorate`-Tag. Durch den Einsatz von`ui:insert`ohne das Attribut`name`fügt Facelets beim Rendern den kompletten Inhalt von`ui:decorate`ein.
Das komplette Template`sideBox.xhtml`für die Box finden Sie in Listing**Template sideBox.xhtml**.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
```

```
    <div class="side_box">
      <p class="header">#{title}</p>
      <ui:insert>Default body</ui:insert>
    </div>
  </ui:composition>
```

Das Beispiel*MyGourmet 11*in Abschnitt**[Sektion: *MyGourmet 11*: Templating mit Facelets]**benutzt ebenfalls mehrere Templates pro Seite, um Boxen in der Seitenleiste zu formatieren.

4.3.3 MyGourmet 11: Templating mit Facelets

Nach der Vorstellung der Templating-Fähigkeiten von Facelets wird es Zeit, diese in unserem Beispiel anzuwenden.*MyGourmet 11*erweitert das Vorgängerbeispiel*MyGourmet 10*um ein einfaches, mithilfe von Templates umgesetztes Layout. Bevor wir allerdings auf die Details der Implementierung eingehen, wollen wir kurz präsentieren, wie die Anwendung im Browser aussieht. Abbildung**MyGourmet 11: Kundenseite im Browser**zeigt die gerenderte Ausgabe der Seite`showCustomer.xhtml`.

Abbildung:MyGourmet 11: Kundenseite im Browser

In diesem Beispiel kommt die bereits kurz vorgestellte mehrstufige Template-Hierarchie aus Abbildung**Templating-Hierarchie von MyGourmet**zum Einsatz. Das Haupttemplate mit dem Namen`template.xhtml`bietet nicht viel Neues. Es definiert das grundlegende Layout der Anwendung mit je einem`div`-Container für die Kopfzeile, die linke Seitenleiste, den Content-Bereich und die Fußzeile. Innerhalb dieser Container befindet sich je ein`ui:insert`-Tag mit einem eindeutigen Namen. Das in Abbildung**MyGourmet 11: Kundenseite im Browser**ersichtliche Design ist in einem verlinkten CSS-Dokument definiert.
Wenden wir uns nun dem Template für die Seiten im Kundenbereich (`customerTemplate.xhtml`) zu. Es leitet sich vom Haupttemplate ab und definiert die Standardinhalte der Kopf- und Fußzeile und der Seitenleiste für den Kundenbereich. Einen Ausschnitt zeigt Listing**MyGourmet 11: Template für Kundenseiten**.

```
  <ui:composition template="template.xhtml">
```

```
  <ui:define name="header">
    <h:graphicImage value="/images/logo.png"/>
    <h1>#{msgs.title_main}</h1>
  </ui:define>
  <ui:define name="left_sidebar">
    <ui:include src="leftSideBar.xhtml"/>
  </ui:define>
  <ui:define name="footer">
    <h:outputText value="#{msgs.footer_left_text}"
        style="float: left;"/>
    <h:outputText value="#{msgs.footer_right_text}"
        style="float: right"/>
  </ui:define>
</ui:composition>
```

Die Definition der Kopf- und Fußzeile erfolgt direkt im Template - die Inhalte sind relativ überschaubar. Für die Seitenleiste kommt jedoch eine andere Vorgehensweise zum Einsatz: Ihr Inhalt wird im separaten XHTML-Dokument`leftSideBar.xhtml`definiert und mit dem Tag`ui:include`in das Template eingebunden.
Die Seitenleiste besteht aus einer Box mit einem kleinen Menü und einer Box mit aktuellen Meldungen. Die Umsetzung entspricht dem in Abschnitt**[Sektion: Mehrere Templates pro Seite]**vorgestellten Beispiel für den Einsatz von mehreren Templates mit`ui:decorate`. Listing**MyGourmet 11: linke Seitenleiste**zeigt einen Ausschnitt der`leftSideBar.xhtml`-Datei.

```
<ui:composition>
  <ui:decorate template="/META-INF/templates/sideBox.xhtml">
    <ui:param name="title" value="#{msgs.menu_title}"/>
    <h:form id="menu">
      <h:panelGrid columns="1">
        <h:commandLink action="showCustomer">
            #{msgs.menu_show_customer}
        </h:commandLink>
      </h:panelGrid>
    </h:form>
  </ui:decorate>
  <ui:decorate template="/META-INF/templates/sideBox.xhtml">
    <ui:param name="title" value="#{msgs.news_title}"/>
    <p>MyGourmet - jetzt mit Facelets und Templating</p>
  </ui:decorate>
</ui:composition>
```

4.4 Bookmarks und GET-Anfragen in JSF

JSF vor Version 2.0 ist nur eingeschränkt in der Lage, bookmarkfähige Links zu erzeugen und mit GET-Anfragen umzugehen. Das liegt vor allem an der Tatsache, dass jeder Klick auf eine`h:commandLink`- oder`h:commandButton`-Komponente eine POST-Anfrage auslöst. Diesem Umstand wird ab Version 2.0 der Spezifikation mit einer erweiterten Unterstützung von GET-Anfragen Rechnung getragen. Das Basispaket, bestehend aus GET-Navigation (siehe Abschnitt**[Sektion: Navigation mit h:link und h:button]**) und View-Parametern (siehe Abschnitt**[Sektion: View-Parameter]**), ist bereits in JSF 2.0 enthalten. JSF 2.2 rundet dieses Paket mit View-Actions (siehe Abschnitt**[Sektion: View-Actions]**) noch weiter ab.

4.4.1 Navigation mit h:link und h:button

Die Basis der GET-Unterstützung bilden mit`h:link`und`h:button`zwei Komponenten, die als Link beziehungsweise Schaltfläche gerendert werden und eine GET-Anfrage absetzen. Der Clou ist, dass dabei trotzdem der Navigationsmechanismus von JSF zum Einsatz kommt. Zu diesem Zweck haben beide Komponenten das Attribut`outcome`, dessen Wert zum Auflösen der URL an den Navigation-Handler übergeben wird. Dieser versucht zuerst einen Navigationsfall zu finden, für den`from-outcome`mit dem Wert von`outcome`übereinstimmt. Bleibt die Suche erfolglos, wird der Wert von`outcome`direkt als View-ID interpretiert. Im Unterschied zur klassischen Navigation wird die View-ID bereits beim Rendern der Ansicht aufgelöst und nicht dynamisch in der Invoke-Application-Phase beim Postback. Dieses Konzept wird daher auch als*präemptive Navigation*bezeichnet.
Zur Demonstration der GET-Fähigkeiten erhält*MyGourmet*zwei neue Ansichten. Die erste neue Seite mit der View-ID`providerList.xhtml`zeigt eine Liste von Anbietern, die Essen ausliefern. Jeder Eintrag dieser Liste verweist mit einem`h:link`-Tag auf die Detailseite`showProvider.xhtml`. Listing**h:link im Einsatz**zeigt einen Ausschnitt der Seitendeklaration`providerList.xhtml`mit dem Link zur Detailseite.

```
<h:dataTable var="provider"
    value="#{providerBean.providerList}">
  <h:column>
    <f:facet name="header">
      <h:outputText value="#{msgs.provider_name}"/>
    </f:facet>
    <h:link outcome="showProvider" value="#{provider.name}">
      <f:param name="id" value="#{provider.id}"/>
    </h:link>
  </h:column>
</h:dataTable>
```

Der Wert des Attributs`value`wird dabei als Linktext gerendert. Wir nutzen an dieser Stelle die implizite Navigation und geben im Attribut`outcome`direkt die View-ID der Detailseite an. Der eindeutige Bezeichner des entsprechenden Anbieters wird mit dem Tag`f:param`als Kind der Linkkomponente bereitgestellt. Listing**Gerenderte Ausgabe von h:link**zeigt, wie JSF die Links rendert. Hier sehen Sie auch, was der Begriff*präemptive Navigation*in der Praxis bedeutet. Bereits beim Rendern der Ansicht wird für jedes`h:link`- oder`h:button`-Komponente die resultierende URL abhängig vom Attribut`outcome`bestimmt. Aktiviert der Benutzer einen der Links beziehungsweise eine der Schaltflächen, schickt der Browser eine simple GET-Anfrage an den Server und es gibt in diesem Fall keinen Postback. Deswegen ist es auch nicht notwendig,`h:link`und`h:button`in ein`h:form`-Tag einzubetten.

```
<a href="/showProvider.jsf?id=1">Pizzeria Venezia</a>
<a href="/showProvider.jsf?id=2">Rhodos</a>
<a href="/showProvider.jsf?id=3">Frying Dutchman</a>
```

Nachdem der Browser eine GET-Anfrage schickt, stimmt auch die URL in der Adressleiste mit der tatsächlich gerenderten Seite überein. Der Anwender kann daher auch ein Lesezeichen auf diese Ansicht setzen. Das hört sich trivial an, trifft aber bei der klassischen Navigation nicht immer zu, da die Befehlskomponenten`h:commandLink`und`h:commandButton`erst einen Postback auf die aktuelle Seite machen, bevor JSF die Navigation ausführt und eine neue Ansicht rendert. Daher hinkt die Adressleiste um eine Ansicht hinterher.

4.4.2

View-Parameter

Wenn der Benutzer den von`h:link`gerenderten Link aktiviert, muss JSF den übergebenen Parameter`id`verarbeiten und den richtigen Anbieter anzeigen. Dabei kommen die sogenannten View-Parameter ins Spiel, die Request-Parameter direkt ans Modell binden. Diese Parameter sind im Grunde nichts anderes als Eingabekomponenten, die über Request-Parameter befüllt werden. Wie bei allen Eingabekomponenten werden auch hier die Werte zuerst konvertiert und validiert.
View-Parameter werden in einer Seitendeklaration innerhalb des Tags`f:metadata`in Form von`f:viewParam`-Komponenten angegeben. Listing**View-Parameter im Einsatz**zeigt den`f:metadata`-Bereich der Ansicht`showProvider.xhtml`. Der darin eingebettete View-Parameter verbindet den Request-Parameter mit dem Name`id`direkt mit der Eigenschaft`providerBean.id`in der Backing-Bean.

```
<f:metadata>
  <f:viewParam name="id" value="#{providerBean.id}"/>
</f:metadata>
```

Sie können überprüfen, ob sich der View-Parameter wie eine Eingabekomponente verhält (oder treffender gesagt eine ist): Rufen Sie die Seite im Browser mit einem nicht numerischen Wert für den Parameter`id`auf. Das Ergebnis ist eine Fehlermeldung des JSF-Number-Konverters, da die Eigenschaft`providerBean.id`den Typ`long`aufweist. Es ist auch ohne Weiteres möglich, einen Validator einzusetzen oder die Eigenschaft mit Bean-Validation-Metadaten zu versehen.
Nachdem der Bezeichner erfolgreich in die Bean übertragen wurde, müssen vor dem Rendern der Ansicht noch die Daten des Anbieters geladen werden. JSF 2.2 bietet dazu mit den neuen View-Actions die eleganteste Lösung an. Alternativ können Sie - speziell wenn Sie noch JSF 2.0 oder 2.1 einsetzen - das Laden der Daten auch über das System-Event`PreRenderViewEvent`realisieren. Wir zeigen Ihnen beide Varianten in Abschnitt**[Sektion: View-Actions]**.
Wenn die Zielseite eines`h:link`- oder`h:button`-Elements View-Parameter enthält, können diese automatisch übernommen werden. Dazu muss lediglich das Attribut`includeViewParams`der Tags`h:link`und`h:button`auf den Wert`true`gesetzt werden. Als Beispiel fügen wir auf`showProvider.xhtml`folgenden Link auf die Seite`editProvider.xhtml`zum Bearbeiten des Anbieters hinzu:

```
<h:link outcome="editProvider" includeViewParams="true"
    value="#{msgs.edit_provider}"/>
```

Beim Rendern der HTML-Ausgabe dieses Links werden die View-Parameter der Zielseite als Parameter hinzugefügt. Vorausgesetzt,`editProvider.xhtml`definiert denselben View-Parameter wie`showProvider.xhtml`, ergibt sich folgende URL:`/editProvider.xhtml?id=1`.

4.4.2.1 Positionierung von f:metadata

Das Tag`f:metadata`muss immer ein direktes Kind von`f:view`sein und darf nicht in ein Template oder ein mit`ui:include`eingefügtes Seitenfragment ausgelagert werden. Die Verbindung von Templating und View-Parametern lässt sich dennoch sehr einfach bewerkstelligen. Dazu muss das Template einen ersetzbaren Bereich für die View-Parameter definieren, der dann im Template-Client ersetzt wird. Listing**Template mit View-Parametern**zeigt ein Beispiel für ein Template mit dem ersetzbaren Bereich`metadata`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:f="http://xmlns.jcp.org/jsf/core">
<body>
  <f:view>
    <ui:insert name="metadata"/>
    <div id="content">
      <ui:insert name="content"/>
    </div>
  </f:view>
</body>
```

```
</html>
```

Listing**Template-Client mit View-Parametern**zeigt einen Template-Client, der auf dem zuvor definierten Template aufbaut und den View-Parameter-Bereich überschreibt. Wie Sie sehen, muss das komplette`f:metadata`-Tag im Template-Client deklariert werden.

```
<ui:composition template="template.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:f="http://xmlns.jcp.org/jsf/core">
  <ui:define name="metadata">
    <f:metadata>
      <f:viewParam name="id" value="#{bean.id}"/>
    </f:metadata>
  </ui:define>
  <ui:define name="content">
    Page content
  </ui:define>
</ui:composition>
```

4.4.2.2 Lebenszyklus mit View-Parametern

Wir klären jetzt noch die Frage, wie sich die Ausführung des Lebens-zyklus ändert, wenn View-Parameter ins Spiel kommen. Bei einer initialen Anfrage auf eine Ansicht handelt es sich ja immer um eine GET-Anfrage. Vor JSF 2.0 sprang bei einer solchen Anfrage die Ausführung des Lebenszyklus nach Phase 1 sofort zu Phase 6 und die Ansicht wurde gerendert.
Dieses Verhalten hat sich verändert: In Phase 1 wird zunächst geprüft, ob es einen Metadatenbereich und View-Parameter gibt. Wenn ja, wird eine neue Ansicht erzeugt, die nur die View-Parameter enthält. Damit wird dann der komplette Lebenszyklus durchlaufen. Gibt es in der Anfrage Parameter, werden die Daten in den folgenden Phasen wie bei einem Postback behandelt: Zuerst werden Sie den einzelnen View-Parametern zugeordnet, dann konvertiert und validiert und ins Modell zurückgeschrieben, falls keine Fehler aufgetreten sind. Abschließend wird die Ansicht wie bisher gerendert.

4.4.3 View-Actions

Bis jetzt haben wir Action-Methoden immer in Kombination mith`:commandButton`oderh`:commandLink`verwendet. Diese klassischen Action-Methoden werden aber nur dann während des Lebenszyklus ausgeführt, wenn ein Benutzer durch einen Klick auf eine Schaltfläche oder einen Link einen Submit ausgelöst hat. View-Actions ermöglichen dahingegen ab JSF 2.2 das Ausführen von Action-Methoden beim initialen Laden einer Seite durch eine GET-Anfrage.
View-Actions waren ursprünglich bereits für JSF 2.0 geplant, da sie einen integralen Bestandteil der GET-Unterstützung bilden. Aus Zeitgründen haben sie es aber erst mit Version 2.2 in die JSF-Spezifikation geschafft.
Wir wollen nun unser weiter oben gegebenes Versprechen einlösen und das Laden der Anbieterdaten mithilfe einer View-Action zeigen. View-Actions werden in einer Seitendeklaration mit dem Tag`f:viewAction`angegeben, das analog zu`f:viewParam`innerhalb von`f:metadata`liegen muss. Die Action-Methode wird dabei wie etwa von`h:commandButton`bekannt als Method-Expression im Attribut`action`referenziert.
Listing**View-Action im Einsatz**zeigt den`f:metadata`-Bereich der Ansicht`showProvider.xhtml`mit der View-Action und dem View-Parameter aus dem letzten Abschnitt. Die in Abschnitt**[Sektion: Positionierung von f:metadata]**definierten Regeln zur Positionierung von`f:metadata`gelten natürlich auch beim Einsatz von View-Actions.

```
<f:metadata>
  <f:viewParam name="id" value="#{providerBean.id}"/>
  <f:viewAction action="#{providerBean.loadProvider}"/>
```

```
</f:metadata>
```

In der referenzierten Action-Methode`loadProvider`werden die Anbieterdaten für die vom View-Parameter gesetzte ID geladen. Nachdem JSF View-Actions standardmäßig in der Invoke-Application-Phase ausführt, können wir ohne Probleme auf den Wert von`id`zugreifen. Dadurch ist aber auch gewährleistet, dass die Methode bei einem Konvertierungsfehler erst gar nicht ausgeführt wird (die Ausführung des Lebenszyklus springt ja vorher schon zur Render-Response-Phase). Wie bei allen Action-Methoden wird der Rückgabewert für die Navigation verwendet. Nachdem wir auf der Seite bleiben wollen, geben wir hier einfach`null`zurück. Listing**Action-Methode für View-Action**zeigt die Action-Methode.

```
public String loadProvider() {
  provider = findProvider(id);
  if (provider == null) {
    GuiUtil.addErrorMessage("error_non_existing_provider", id);
  }
  return null;
}
```

Die Navigation mit View-Actions funktioniert im Grunde genau so wie bei normalen Action-Methoden. Der einzige Unterschied ist, dass eine von View-Actions initiierte Navigation immer automatisch als Redirect ausgeführt wird. Listing**Action-Methode für View-Action mit Navigation**zeigt eine Variante der vorherigen Action-Methode, die bei nicht existierenden Anbieterdaten auf eine Fehlerseite mit der View-ID`error.xhtml`weiterleitet. Auf diese Art und Weise lässt sich zum Beispiel auch eine einfache Loginlösung mit View-Actions basteln: Solange der Benutzer nicht eingeloggt ist, löst die Action-Methode immer einen Redirect auf die Login-Seite aus.

```
public String loadProvider() {
  provider = findProvider(id);
  if (provider == null) {
    return "error";
  }
  return null;
}
```

Wie bereits erwähnt führt JSF View-Actions standardmäßig in der Phase Invoke-Application aus. Bei Bedarf können View-Actions allerdings in jeder Phase von 2 bis 5 durchgeführt werden. Sie müssen dazu lediglich im Attribut`phase`einen der folgenden Werte eintragen:
,`PROCESS_VALIDATIONS`,`UPDATE_MODEL_VALUES`oder`INVOKE_APPLICATION`. Damit unsere View-Action in Phase 2 des Lebenszyklus in die Tat umgesetzt wird, müssten wir`f:viewAction`wie folgt ändern:

```
<f:viewAction action="#{providerBean.loadProvider}"
    phase="APPLY_REQUEST_VALUES"/>
```

Beachten Sie aber, dass in diesem Fall das Feld`id`noch nicht gesetzt ist - das wird erst in Phase 4 erledigt! Die View-Action wird ebenfalls bereits in Phase 2 des Lebenszyklus ausgeführt, wenn das Attribut`immediate`auf`true`gesetzt wird.
View-Actions werden standardmäßig nur bei initialen GET-Requests auf die Seite aufgerufen. Soll die View-Action zusätzlich auch bei Postbacks ausgeführt werden, muss das Attribut`onPostback`auf`true`gesetzt werden.

4.4.3.1 View-Actions im Vergleich zum System-Event PreRenderViewEvent

Mit JSF 2.0 oder 2.1 können Sie als Alternative zu View-Actions das System-Event`PreRenderViewEvent`verwenden. Mit JSF 2.2 funktioniert das natürlich weiterhin - View-Actions sind aber deutlich flexibler. Dazu registrieren wir über das Tag`f:event`direkt in der Seitendeklaration die Methode`preRenderView`der Managed-Bean als Listener für dieses Ereignis. Hier die Registrierung in`showProvider.xhtml`:

```
<f:event type="preRenderView"
    listener="#{providerBean.preRenderView}"/>
```

Listing**Listener-Methode für PreRenderViewEvent**zeigt die zuvor registrierte Listener-Methode in der Klasse`ProviderBean`. Hier erfolgt das Laden der Anbieterdaten, falls beim Konvertieren und Validieren des View-Parameters kein Fehler aufgetreten ist. Seit JSF 2.0 lässt sich diese Abfrage einfach über die Methode`isValidationFailed`am Faces-Context bewerkstelligen. Sollte kein Anbieter mit dem angegebenen Bezeichner existieren, erzeugen wir eine entsprechende Nachricht.

```
public void preRenderView(ComponentSystemEvent ev) {
  FacesContext ctx = FacesContext.getCurrentInstance();
  if (!ctx.isValidationFailed()) {
    this.provider = findProvider(id);
    if (provider == null) {
      GuiUtil.addErrorMessage("error_non_existing_provider", id);
    }
  }
}
```

Anhand des Beispiels zeigt sich auch schon der erste Unterschied zwischen View-Actions und einem Listener für das`PreRenderViewEvent`. Die Listener-Methode wird immer zu Beginn der Phase 6 des Lebenszyklus ausgeführt - und das sowohl für initiale Anfragen als auch für Postbacks. Das würde für unser Beispiel bedeuten, dass die Anbieterdaten bei jedem Request neu geladen werden. Nachdem die Managed-Bean aber im View-Scope liegt, ist das eigentlich nicht notwendig. Mit der View-Action werden die Daten nur beim ersten Zugriff auf die Seite über eine GET-Anfrage geladen. Und das auch nur dann, wenn die Konvertierung des View-Parameters nicht fehlschlägt.
Mit View-Actions können Sie außerdem bei Bedarf ohne Probleme in die Navigation eingreifen.
Ein weiterer Unterschied fällt nicht sofort auf den ersten Blick auf. Im Gegensatz zum Aufruf des Listeners für das`PreRenderViewEvent`ist zum Zeitpunkt des Aufrufs der View-Action der Komponentenbaum noch nicht aufgebaut. Dadurch ist es zwar einerseits nicht möglich auf einzelne Komponenten zuzugreifen, andererseits ergibt sich dadurch aber ein Performance-Vorteil.

4.4.4 MyGourmet 12: GET-Unterstützung

Das Beispiel*MyGourmet 12*fasst alle Änderungen rund um das Thema View-Parameter und View-Actions zusammen. Die augenscheinlichste Neuerung ist der neue Anbieterbereich in der Anwendung. Als Einstiegspunkt dient die Seite`providerList.xhtml`mit einer Übersicht der Anbieter. Diese Liste wird beim Erzeugen der Backing-Bean`ProviderBean`mit einigen Werten initialisiert. Von dieser Seite führt pro Anbieter ein`h:link`-Komponente zu`showProvider.xhtml`. Die Daten eines Anbieters werden in Instanzen der Klasse`Provider`abgelegt.
Beim Aufruf der Seite`showProvider.xhtml`wird die als Request-Parameter mitgeschickte ID des Anbieters mit einem View-Parameter in der Managed-Bean gespeichert. Eine View-Action sorgt dann dafür, dass die Anbieterdaten für diese ID geladen werden.
Da wir seit*MyGourmet 11*Templating benutzen, mussten wir das Template (wie in Abschnitt**[Sektion: Positionierung von f:metadata]**beschrieben) für den Einsatz von View-Parametern anpassen.
Eine weitere kleine Änderung betrifft die linke Seitenleiste. Damit der Anbieterbereich erreichbar ist, haben wir das Menü um einen Link erweitert. Zur besseren Unterstützung von Bookmarking haben wir das Menü auf`h:link`-Komponenten umgestellt. Als angenehmer Nebeneffekt kann dadurch die Form eingespart werden.
Listing**MyGourmet 12: Einsatz des Listenkonverters**zeigt den Einsatz eines neuen Konverters, der*Collections*in Zeichenketten umwandelt. Mit dem Attribut`bundleName`kann ihm der Name eines Resource-Bundles übergeben werden, aus dem die Einträge der Liste aufgelöst werden. Nachdem mittlerweile der Anbieter und der Kunde eine Liste von Kategorien aufweisen, die aber nur symbolische Konstanten enthalten, spart dieser Konverter einiges an

Quelltext.

```
<h:outputText value="#{providerBean.provider.categories}">
  <mg:convertList separator=", " bundleName="msgs"/>
</h:outputText>
```

Listing **MyGourmet 12: Listenkonverter** zeigt die `getAsString`-Methode und die Eigenschaften der Konverterklasse `ListConverter`, die über die Attribute des Custom-Tags `mg:convertList` gesetzt werden. Dieses Tag wurde, wie schon im letzten Beispiel gezeigt, in der Tag-Bibliothek `mygourmet.taglib.xml` definiert.

```
private String separator;
private String bundleName;

public String getAsString(FacesContext ctx,
    UIComponent comp, Object value) {
  StringBuilder builder = new StringBuilder();
  if (value instanceof Collection) {
    for (Object obj : (Collection)value) {
      String item = obj.toString();
      if (builder.length() > 0 && separator != null) {
        builder.append(separator);
      }
      if (bundleName != null && bundleName.length() > 0) {
        builder.append(GuiUtil.getResourceText(
            ctx, bundleName, item));
      } else {
        builder.append(item);
      }
    }
  }
  return builder.toString();
}
```

4.5 Die JSF-Umgebung: Faces-Context und External-Context

Bisher sind wir immer wieder auf den Faces-Context gestoßen. Dieser Kontext stellt die zentrale Schaltstelle einer JSF-Anwendung dar und wird durch die Klasse `javax.faces.context.FacesContext` repräsentiert. Er wird ganz am Anfang jeder HTTP-Anfrage vom *Faces-Servlet* initialisiert und steht dem Entwickler ab dann als Parameter vieler Methoden, aber auch jederzeit über den Aufruf der Methode `FacesContext.getCurrentInstance()` zur Verfügung.
Mit dem Faces-Context ist ein direkter Zugriff auf den EL-Resolver möglich. Damit lassen sich Objekte, die über die *Unified-EL* verfügbar sind, direkt im Java-Code auflösen. Listing **Zugriff auf eine Managed-Bean im Java-Code** zeigt zum Beispiel, wie der Zugriff auf eine Managed-Bean namens `personList` aussieht.

```
FacesContext fc = FacesContext.getCurrentInstance();
```

```
fc.getApplication().getELResolver().getValue(
    fc.getELContext(), null, "personList");
```

Eine wichtige Anwendung des Faces-Contexts ist das Hinzufügen von Nachrichten für die Darstellung auf der Webseite - und die Möglichkeit, auf die bisher hinzugefügten Nachrichten zuzugreifen. Zu diesem Zweck existieren folgende Methoden:

- `addMessage(String clientId, FacesMessage message)`:
  Fügt eine Nachricht zum Faces-Context hinzu. Falls eine Client-ID angegeben wird, bezieht sich die Nachricht auf die entsprechende Komponente, andernfalls ist sie global.
- `Iterator<FacesMessage> getMessages()`:
  Gibt einen`Iterator`über alle Nachrichten im Faces-Context zurück. Es sind auch jene inkludiert, die einer Komponente zugeordnet sind.
- `Iterator<FacesMessage> getMessages(String clientId)`:
  Gibt einen`Iterator`über alle Nachrichten im Faces-Context für die Komponente mit der angegebenen Client-ID zurück.
- `List<FacesMessage> getMessageList()`:
  Gibt eine Liste mit allen Nachrichten im Faces-Context zurück (ab JSF 2.0). Es sind auch jene inkludiert, die einer Komponente zugeordnet sind.
- `List<FacesMessage> getMessageList(String clientId)`:
  Gibt eine Liste über alle Nachrichten im Faces-Context für die Komponente mit der angegebenen Client-ID zurück (ab JSF 2.0).

Weiterführende Details zum Hinzufügen und Verwalten von Nachrichten finden Sie in Abschnitt**Sektion: Nachrichten**.
Applikationsobjekte anlegen:Über den Faces-Context gelangen Sie auch zur*Application*, und die hilft Ihnen sowohl beim Erzeugen von neuen Komponenten als auch von Method- und Value-Expressions (siehe Listing**Applikationsobjekte anlegen**).

```
fc.getApplication().getExpressionFactory().
    createValueExpression(ELContext ctx,
        String expression, Class expectedType);

fc.getApplication().getExpressionFactory().
    createMethodExpression(ELContext ctx, String expression,
    Class expectedReturnType, Class[] params);

fc.getApplication().createComponent(String componentType);
```

Wobei der`ELContext`- wie in Listing**Zugriff auf eine Managed-Bean im Java-Code**bereits gezeigt - als Eigenschaft des Faces-Contexts verfügbar ist. Auch hier zahlt es sich aus, Utility-Methoden für das Erzeugen neuer Elemente vorzusehen.
Lebenslauf beeinflussen:Ein wichtiger Bereich im Faces-Context ist die Möglichkeit, den Lebenslauf einer HTTP-Anfrage zu beeinflussen. Dazu gibt es folgende Methoden, die bereits bei der Ereignisbehandlung in Abschnitt**Sektion: Ereignisse und Ereignisbehandlung**zum Einsatz gekommen sind:

- `renderResponse()`:
  Nach Abschluss der momentan laufenden Phase wird sofort die HTTP-Antwort gerendert (die Ausführung des Lebenszyklus springt sofort zur Render-Response-Phase).
- `responseComplete()`:
  Die Abarbeitung des Lebenszyklus wird nach Abschluss der momentan laufenden Phase beendet.

External-Context:Schließlich bietet der Faces-Context noch die Möglichkeit, auf den External-Context zuzugreifen. Der External-Context ist der Wrapper rund um die der Webanwendungsumgebung zugrundeliegende

Funktionalität; also in den meisten Fällen um entweder den`ServletContext`oder`PortletContext`. Hier der für einen Zugriff notwendige Code:

```
FacesContext fc = FacesContext.getCurrentInstance();
ExternalContext ec = fc.getExternalContext();
```

Auch der External-Context bietet einige interessante Methoden, die sich aus der Funktionalität des Basis-Contexts ergeben. Hier eine selektive Auswahl:

- `Map<String,Object> getRequestMap()`liefert eine veränderbare*Map*mit allen Attributen des Requests zurück. Dazu zählen unter anderem auch alle instanziierten Managed-Beans im Gültigkeitsbereich`request`, nicht aber die Request-Parameter.
- `Map<String,String> getRequestParameterMap()`liefert eine nicht veränderbare*Map*mit allen Request-Parametern zurück. Der Name des Parameters ist dabei der Schlüssel der*Map*.
- `Map<String,Object> getSessionMap()`liefert eine veränderbare*Map*mit allen Attributen der Session zurück. Dazu zählen unter anderem auch alle instanziierten Managed-Beans im Gültigkeitsbereich`session`.
- `String getRemoteUser()`liefert den Namen des Benutzers zurück, der den Request abgesetzt hat, falls dieser vorher authentifiziert wurde.
- `boolean isUserInRole(roleName)`prüft Rollenberechtigungen und liefert`true`zurück, wenn der authentifizierte Benutzer für die Rolle`roleName`authorisiert ist.

Der External-Context bietet noch einige weitere Methoden, die Sie am besten der API-Dokumentation entnehmen.

4.6 Konfiguration von JavaServer Faces

JSF verwendet ein Minimum an zu editierenden XML-Dateien. Sieht man von einer eventuellen Verwendung ergänzender Bibliotheken oder*Portlets*ab, existieren nur zwei Konfigurationsdateien für eine JSF-basierte Webanwendung:

- Webanwendung-Konfigurationsdatei`web.xml`:
  Wie bei allen Java-EE-Webapplikationen dient dieser*Deployment-Deskriptor*zum Setzen zentraler Einstellungen in der Applikation. Hier finden sich Definitionen von Kontextparametern, Listenern, Filtern, das*Faces-Servlet*oder auch das*Servlet-Mapping*wieder. Details dazu finden Sie in Abschnitt**[Sektion: Die Webkonfigurationsdatei web.xml]**.
- JSF-Konfigurationsdatei`faces-config.xml`:
  Diese XML-Datei ist die zentrale Konfigurationsdatei von JSF. Details dazu enthält Abschnitt**[subsec Die JSF-Konfigurationsdatei -- faces-config.xml]**.

Abschnitt**[subsec Konfiguration der*Unified-EL*]**zeigt die Konfiguration zur Verwendung der alternativen EL-Implementierung von*JBoss*. Damit lassen sich auch auf älteren Servern (vor*Java EE 6*) die Features der neuen*Unified-EL*nutzen.

4.6.1 Die

Webkonfigurationsdatei web.xml

Sehen wir uns nun im Folgenden eine typische`web.xml`-Datei einer JSF-Applikation genauer an. Der wichtigste Teil des Deployment-Deskriptors ist die Spezifikation des*Faces-Servlets*, das die Anfragen an die JSF-Anwendung bearbeitet, und dessen*Mapping*. Ein weiterer wichtiger Aspekt der Webapplikation, der in diesem Abschnitt behandelt wird, sind Konfigurationsparameter.

4.6.1.1 Faces-Servlet und Mapping

Jede JSF-Applikation muss ein Faces-Servlet konfigurieren. Listing**Deployment-Deskriptor von MyGourmet 1**zeigt nochmals die`web.xml`-Datei des Beispiels*MyGourmet 1*, in der das von JSF mitgelieferte Servlet`javax.faces.webapp.FacesServlet`inklusive des zugehörigen Servlet-Mappings definiert sind.

```
<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 - MyGourmet 1</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>*.jsf</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Welche Anfrage auf welches Servlet weitergeleitet wird, zeigt das`servlet-mapping`-Element an. Der Wert des`url-pattern`-Elements definiert ein Präfix oder Postfix der Anfrageadresse, das dem*Faces-Servlet*zugewiesen wird. Im Beispiel*MyGourmet 1*wurde die Postfix-Zuordnung (auch*Extension-Mapping*genannt)`*.jsf`gewählt, wobei sämtliche Anfragen mit der Endung`.jsf`durch das*Faces-Servlet*gehandhabt werden. Intern wird diese Adresse dann auf die View-ID der Seitendefinition umgelegt.
Die zweite Möglichkeit, ein*Servlet-Mapping*zu definieren, ist eine Präfix-Zuordnung`<url-pattern>/faces/*</url-pattern>`. Die View-ID der Seitendefinition ergibt sich in diesem Fall direkt aus der URL, nachdem das Präfix entfernt wurde.
Sehen wir uns kurz den entscheidenden Unterschied zwischen Präfix- und Postfix-Mapping anhand eines Beispiels an. Wir wollen dazu die Seite mit der Seitendefinition`/helloWorld.jsp`der fiktiven Webanwendung`http://www.mustermann.org`im Browser laden. Mit einem Postfix-Mapping müssen wir dazu die URL

```
http://www.mustermann.org/helloWorld.jsf
```

in die Adressleiste eingeben. Mit einem Präfix-Mapping sieht die URL folgendermaßen aus:

```
http://www.mustermann.org/faces/helloWorld.jsp
```

In der Regel sind beide Methoden für eine JSF-Anwendung einsetzbar. In manchen Fällen, wie beim Einsatz von Tomahawk oder Trinidad, kann es allerdings von Vorteil sein, ein Präfix-Mapping zu verwenden.

4.6.1.2 Kontextparameter

An erster Stelle in der Konfiguration stehen üblicherweise die Kontextparameter. Die optionale Angabe von`context-param`-Elementen dient der Definition von Parametern zur Initialisierung des*Servlet-Contexts*. Hier können Basiseinstellungen vorgenommen werden. Die folgende Liste zeigt eine Übersicht der wichtigsten Einstellungen für JSF:

- `javax.faces.CONFIG_FILES`:
  Über`param-value`können JSF-Konfigurationsdateien angegeben werden. Auch wenn der Wert dieses Parameters leer bleibt, wird die Standarddatei`WEB-INF/faces-config.xml`immer geladen, daher ist der Parameter optional. Für eine bessere Übersichtlichkeit kann es auch wünschenswert sein, mehrere solcher Dateien einzuführen. Die Namen dieser Dateien müssen dann im`param-value`-Element durch Kommata voneinander getrennt werden. Listing**Einsatz mehrerer Konfigurationsdateien**zeigt ein entsprechendes Beispiel.

```
<param-name>javax.faces.CONFIG_FILES</param-name>
<param-value>
    /WEB-INF/faces-navigation.xml,
    /WEB-INF/faces-I18N.xml
</param-value>
```

- `javax.faces.DEFAULT_SUFFIX`:
  Dieser Parameter definiert eine durch Leerzeichen getrennte Liste von Erweiterungen für View-Identifier, die JSF als JSP-Deklarationen interpretieren soll. Der Standardwert ist`.jsp`.
- `javax.faces.FACELETS_BUFFER_SIZE`:
  Mit diesem Parameter kann die Größe des Buffers der HTTP-Response gesetzt werden. Der Buffer kann mit dem Wert -1 deaktiviert werden, was auch der Standardwert ist.
- `javax.faces.FACELETS_LIBRARIES`:
  Mit diesem Parameter werden Tag-Bibliotheken für Facelets in die Anwendung eingebunden. Als Wert wird eine durch Semikolons separierte Liste von Pfaden relativ zum Wurzelverzeichnis der Anwendung erwartet. Listing**Einbinden mehrerer Tag-Bibliotheken**zeigt, wie zwei im Verzeichnis`WEB-INF`liegende Tag-Bibliotheken eingebunden werden.

```
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>
    /WEB-INF/mygourmet.taglib.xml;
    /WEB-INF/converters.taglib.xml
  </param-value>
</context-param>
```

- `javax.faces.FACELETS_REFRESH_PERIOD`:
  Dieser Parameter definiert die Zeitdauer in Sekunden, nach der Seitendeklarationen bei Anfragen auf Änderungen überprüft. Ein Wert von 10 bedeutet zum Beispiel, dass nach dem Kompilieren der Seitendeklaration für mindestens 10 Sekunden keine Änderungen berücksichtigt werden. Während der Entwicklung sollte der Wert möglichst niedrig sein, damit Änderungen am Server sofort berücksichtigt werden. Ein Wert von -1 schaltet die Überprüfung von Änderungen komplett aus. Diese Einstellung bietet sich speziell für Produktionssysteme an, da die Überprüfung natürlich auch einen minimalen Aufwand darstellt. Beachten Sie aber, dass in diesem Fall Änderungen an Seitendeklarationen ohne Serverneustart

nicht mehr möglich sind. Der Standardwert für diesen Parameter beträgt in*Mojarra*2.*MyFaces*definiert den Standardwert abhängig von der aktuellen Project-Stage . Ist die Project-Stage auf`Production`gesetzt, beträgt der Standardwert -1, ansonsten 2.

- `javax.faces.FACELETS_SKIP_COMMENTS`:
  Wenn dieser Parameter auf den Wert`true`gesetzt ist, entfernt beim Kompilieren sämtliche Kommentare aus den Seitendeklarationen. Damit können Sie sicherstellen, dass von Entwicklern eingefügte Kommentare nicht in der gerenderten Ausgabe auftauchen.
- `javax.faces.FACELETS_SUFFIX`:
  Dieser Parameter definiert eine durch Leerzeichen getrennte Liste von Erweiterungen für View-Identifier, die JSF als Facelets-Deklarationen interpretieren soll. Der Standardwert ist`.xhtml`.
- `javax.faces.FACELETS_VIEW_MAPPINGS`:
  Dieser Parameter definiert eine durch Semikolons getrennte Liste von View-Identifiern, die JSF als Facelets-Deklarationen interpretieren soll. Diese Liste kann auch Einträge mit Wildcards wie`/secure/*`enthalten.
- `javax.faces.FULL_STATE_SAVING_VIEW_IDS`:
  Dieser Parameter definiert eine durch Kommata getrennte Liste von View-Identifiern, für die JSF das aus Version 1.2 bekannte Full-State-Saving verwenden soll.
- `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL`:
  Das Setzen dieses Parameters auf den Wert`true`bewirkt, dass JSF vor der Validierung bei allen Eingabekomponenten, deren Submitted-Value eine leere Zeichenkette ist, den Submitted-Value auf`null`setzt. Dieses Verhalten ist standardmäßig nicht aktiviert.
- `javax.faces.PARTIAL_STATE_SAVING`:
  Mit diesem Parameter lässt sich steuern, ob das mit JSF 2.0 eingeführte Partial-State-Saving in der Anwendung eingesetzt wird. Der Wert`true`aktiviert Partial-State-Saving. Das Gleiche gilt auch, wenn der Parameter nicht gesetzt wird. Alle anderen Werte, wie etwa`false`, deaktivieren das Partial-State-Saving für die gesamte Anwendung.
- `javax.faces.PROJECT_STAGE`:
  Mit diesem Parameter kann die Projektphase der Anwendung auf einen der folgenden Werte gesetzt werden:`Production`(Standardwert),`Development`,`SystemTest`oder`UnitTest`.
- `javax.faces.STATE_SAVING_METHOD`:
  JSF speichert beim Rendern einer Ansicht den Zustand des Komponentenbaums für nachfolgende Anfragen auf dieselbe Ansicht. Die Datenhaltung kann entweder auf dem Client oder auf dem Server geschehen. Ist`param-value`auf`server`gesetzt, verringert sich der Bandbreitenbedarf, dafür wird der Server stärker unter Last gesetzt. Steht`param-value`hingegen auf`client`, steigt das Datenaufkommen und der Komponentenbaum muss clientseitig in-Felder serialisiert (*Base64-encoded*) werden. Ist kein ausdrücklicher Grund vorhanden, den Status der Applikation auf dem Client zu speichern, wird empfohlen, die Standardmethode`server`nicht zu überschreiben.
- `javax.faces.VALIDATE_EMPTY_FIELDS`:
  Dieser Parameter steuert das Validierungsverhalten von JSF für Eingabekomponenten, deren Wert`null`oder leer ist. Folgende drei Werte sind möglich, der Standardwert ist`auto`:
  - `true`: JSF validiert alle Eingabekomponenten.
  - `auto`: JSF validiert Eingabekomponenten, deren Wert`null`oder leer ist, wenn Bean-Validation in der Anwendung verfügbar ist.
  - `false`: JSF validiert keine Eingabekomponenten, deren Wert`null`oder leer ist.
- `javax.faces.WEBAPP_CONTRACTS_DIRECTORY`:
  Definiert das Verzeichnis, aus dem JSF Resource-Library-Contracts in der Webapplikation auflöst. Der Standardwert ist`/contracts`. Details dazu finden Sie in Abschnitt**Sektion: Ein erstes Beispiel**.

- `javax.faces.WEBAPP_RESOURCES_DIRECTORY`:
  Definiert das Verzeichnis, aus dem JSF Ressourcen in der Webapplikation auflöst. Der Standardwert ist`/resources`. Details dazu finden Sie in Abschnitt**Sektion: Identifikation von Ressourcen -- Teil 1**.

4.6.2 Die JSF-Konfigurationsdatei -- faces-config.xml

In diesem Abschnitt geht es darum, zentrale Einstellungen für*JavaServer Faces*zu treffen. Einerseits bietet die`faces-config.xml`die Möglichkeit, alltägliche Konfigurationseinstellungen in der Entwicklung einer JSF-Applikation zu setzen. Dazu gehört die Konfiguration von Managed-Beans, Navigationsregeln sowie Einstellungen zur Applikation selbst. Der zweite Aufgabenbereich, die Registrierung von Komponenten, Renderern, Validatoren und Konvertern, ist für den Entwickler von Komponenten (und Komponentenbibliotheken) interessant. Zuletzt gibt es noch erweiterte Möglichkeiten, wie das Einbinden von*Phase-Listenern*und die Konfiguration von*Factories*für die Erzeugung von JSF-Kernklassen.
JSF bietet ab Version 2.0 für viele Einstellungen in der Konfigurationsdatei`faces-config.xml`alternativ die Möglichkeit, Annotationen einzusetzen.
Die Konfiguration von Managed-Beans erfolgt in jeweils einem`managed-bean`-Element pro Bean oder ab JSF 2.0 mit der Annotation . Eine genauere Beschreibung der umfassenden Einstellmöglichkeiten findet sich in Abschnitt**subsec Konfiguration von Managed-Beans**.
Die Regeln für die Navigation in einer JSF-Anwendung werden mithilfe von`navigation-rule`-Elementen definiert. Wie das genau funktioniert, zeigt Abschnitt**Sektion: Navigation**. Im Laufe der Entwicklung einer JSF-Webapplikation können sehr viele Navigationsregeln anfallen. Um die Übersicht zu behalten, empfiehlt sich die Auslagerung einzelner Teile der Konfiguration in separate Dateien, wie es im Abschnitt der JSF-Konfigurationsdatei`faces-config.xml`beschrieben wurde.

4.6.2.1 Anwendungseinstellungen -- application

Ein zentrales Element in der`faces-config.xml`ist das Tag`application`. Darin werden wichtige Einstellungen für Kernbereiche von JSF getroffen. Das Element`application`kann als die zentrale Schaltstelle einer JSF-Anwendung betrachtet werden. Hier ist es möglich, neben Einstellungen zur Lokalisierung und der Definition von Message- und Resource-Bundles essenzielle Teile wie den View-Handler oder den EL-Resolver von JSF mit eigenen Implementierungen zu dekorieren. Damit wird dem Benutzer ein mächtiges Werkzeug in die Hand gegeben, um die Applikation an eigene Bedürfnisse anzupassen.
Listing**faces-config.xml mit Spring-EL-Resolver**zeigt ein Beispiel für eine`faces-config.xml`, auf deren Elemente wir weiter unten eingehen werden.

```
<faces-config>
<application>
  <navigation-handler>
    at.irian.jsfatwork.MyNavigationHandler
  </navigation-handler>
  <el-resolver>
    org.springframework.web.jsf.el.SpringBeanFacesELResolver
  </el-resolver>
  <message-bundle>
    at.irian.jsfatwork.messages
  </message-bundle>
  <locale-config>
    <default-locale>de</default-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
```

```
    <resource-bundle>
      <base-name>at.irian.jsfatwork.text</base-name>
      <var>text</var>
    </resource-bundle>
  </application>
  </faces-config>
```

In der folgenden Auflistung finden Sie eine Übersicht der wichtigsten Einstellungen. Bei den einzelnen Punkten handelt es sich jeweils um Kindelemente des Tags`application`:

- `locale-config`ist ein Container zur Definition der von der Anwendung unterstützten Sprachen. In Listing**faces-config.xml mit Spring-EL-Resolver**ist zum Beispiel Deutsch als Standardsprache und Englisch als unterstützte Sprache definiert. Die Konfiguration der Internationalisierung wird ausführlicher in Abschnitt**Sektion: Internationalisierung**behandelt.
- `message-bundle`definiert den Namen des Resource-Bundles, das die Nachrichten der Applikation enthält, wie Listing**faces-config.xml mit Spring-EL-Resolver**zeigt. Genauere Informationen hierzu finden Sie in Abschnitt**subsec Internationalisierung der JSF-Nachrichten**.
- `resource-bundle`ist ein Container für die Definition eines Resource-Bundles, das Texte für die Anwendung enthält. Im Beispiel in Listing**faces-config.xml mit Spring-EL-Resolver**wird das Resource-Bundle`at.irian.jsfatwork.text`unter dem Name`text`in Seitendefinitionen verfügbar gemacht. Genauere Informationen hierzu finden sich in Abschnitt**subsec Internationalisierung der Anwendungstexte**.
- `navigation-handler`definiert die Klasse des verwendeten*Navigation-Handlers*. Dieser kann zum Beispiel für erweitertes Logging oder andere spezielle Zwecke im Navigationsbereich überschrieben werden.
- `view-handler`definiert die Klasse des verwendeten*View-Handlers*, für den es eine Grundimplementierung gibt. Mit Facelets und JSF vor Version 2.0 musste hier der Facelets-View-Handler eingetragen werden.
- `state-manager`definiert die Klasse des verwendeten*State-Managers*.
- `el-resolver`definiert die Klasse des verwendeten*EL-Resolvers*, der für die Auflösung der Value-Expressions zuständig ist. Hier kann zum Beispiel der für*Spring*eingetragen werden.

Die weiteren möglichen Elemente der`faces-config.xml`, wie die Registrierung der Renderer, Konverter oder Validatoren, werden in den Abschnitten**Sektion: Konvertierung**und**Sektion: Validierung**beziehungsweise in Kapitel**Kapitel: Die eigene JSF-Komponente**(für die Konfiguration von Komponenten) noch genauer behandelt.

4.6.3 Konfiguration der Unified-EL

Falls Sie einen älteren Server (wie Tomcat 6 oder Jetty 7) einsetzen, müssen Sie trotzdem nicht auf die wichtigsten Features der neuen*Unified-EL*aus*Java EE 6*verzichten.
Die alternative EL-Implementierung von*JBoss*bietet die wichtigsten Features der neuen*Unified-EL*, implementiert aber noch die alte API. Dadurch lassen sich Probleme mit älteren Servern vermeiden. Die*JBoss-EL*in Version 2.0.1.GA erhalten wir über eine Abhängigkeit in der Maven-Projektdatei`pom.xml`(siehe Listing**Maven-Abhängigkeit zur JBoss-EL**).

```
<dependency>
  <groupId>org.jboss.el</groupId>
  <artifactId>jboss-el</artifactId>
  <version>2.0.1.GA</version>
```

```
    <scope>compile</scope>
  </dependency>
```

Im zweiten Schritt müssen wir JSF davon überzeugen, die alternative Implementierung zu verwenden. Dazu muss über einen Kontextparameter die Klasse`org.jboss.el.ExpressionFactoryImpl`als Expression-Factory gesetzt werden. Tabelletab:expression-factoryzeigt die Namen der Kontextparameter für*Apache MyFaces*und*Mojarra*.

5 Verwaltung von Ressourcen

In JSF handelt es sich bei Ressourcen um Artefakte wie Bilder, Skripte oder Stylesheets, die eine Komponente benötigt, um vollständig am Client angezeigt zu werden.
Die Verwaltung von Ressourcen ist eine Anforderung, die in der einen oder anderen Form für jede Anwendung relevant ist. Besonders für Komponentenbibliotheken ist es wichtig, die Abhängigkeiten zwischen Komponenten und Ressourcen wie Skripten oder Stylesheets zu definieren. Außerdem müssen diese Ressourcen für den Anwendungsentwickler transparent zur Verfügung gestellt werden. Vor JSF 2.0 hat es dafür keinen standardisierten Weg gegeben. Viele Anbieter haben daher eigenständige Lösungen entwickelt, die allerdings in den wenigsten Fällen miteinander kompatibel sind.
Ab JSF 2.0 gibt es einen standardisierten Weg, Ressourcen zu verwalten und flexibel in der Ansicht zu positionieren. Davon profitieren nicht nur Komponentenbibliotheken, sondern alle Anwendungen mit eigenen Bildern, Stylesheets oder Skripten. Die Abschnitte**[Sektion: Identifikation von Ressourcen -- Teil 1]**bis**[Sektion: Ressourcen in*MyGourmet 12*]**zeigen ausführlich, wie die Verwaltung von Ressourcen funktioniert.
JSF 2.2 bietet mit den sogenannten Resource-Library-Contracts eine Möglichkeit zur Definition austauschbarer Templates und Ressourcen, wie Abschnitt**[Sektion: Resource-Library-Contracts]**zeigt.

5.1 Identifikation von Ressourcen -- Teil 1

Ressourcen werden in JSF im einfachsten Fall durch ihren Namen identifiziert. Sehen wir uns das am besten anhand eines Beispiels an. Folgender Code fügt das Bild`image.png`in eine Ansicht ein:

```
<h:graphicImage name="image.png"/>
```

Erfahrene JSF-Entwickler werden sofort bemerkt haben, dass in JSF~1.2 noch kein Attribut`name`für dash`:graphicImage`-Tag definiert ist. Der Wert dieses Attributs ist der Name der Ressource, die als Bild ausgegeben werden soll. Es handelt sich dabei nicht um eine Pfadangabe im klassischen Sinn, sondern um einen Bezeichner, der intern in den tatsächlichen Pfad der Ressource aufgelöst wird.
JSF sucht Ressourcen an den folgenden Stellen einer Anwendung (in der angegebenen Reihenfolge):

1. Im Wurzelverzeichnis der Webapplikation unter`/resources`
2. In`/META-INF/resources`und somit auch in allen Jar-Dateien im Classpath

In unserem Fall versucht JSF die Bezeichnung`image.png`an einem der oben genannten Orte aufzulösen. Daher legen wir folgende Datei an:

```
/resources/image.png
```

Das oben angeführte Beispiel zeigt den einfachen Fall einer Ressource, die über ihren Namen referenziert wird. JSF bietet aber darüber hinaus eine Einteilung in Bibliotheken, eine Versionierung und die Lokalisierung von Ressourcen.
Bibliotheken sind eine Möglichkeit, Ressourcen unter einem gemeinsamen Namen zu gruppieren. Der Bibliotheksname wird dann gemeinsam mit dem Ressourcennamen zur Identifikation der Ressource verwendet. Wir erweitern unser Beispiel um eine Bibliothek`images`, in der alle Bilder abgelegt sind. Die Codezeile von oben ändert sich wie folgt ab:

```
<h:graphicImage library="images" name="image.png"/>
```

JSF interpretiert die Bibliothek beim Auflösen der Ressource als zusätzliches Verzeichnis. Der Pfad der Bilddatei sieht folgendermaßen aus:

```
/resources/images/image.png
```

Doch damit noch nicht genug - Ressourcen und Bibliotheken können in mehreren Versionen existieren und lokalisiert werden. Die Funktionsweise zeigt Abschnitt**[Sektion: Identifikation von Ressourcen -- Teil 2]**.
JSF 2.2:In Versionen vor 2.2 sucht JSF Ressourcen in der Webanwendung immer im Verzeichnis`/resources`. Dieser Pfad hat aber den Nachteil, dass er ohne zusätzliche Absicherungen nach außen verfügbar ist. Das ist insbesondere für Kompositkomponenten nicht immer das gewünschte Verhalten. Ab JSF 2.2 lässt sich dieses Verzeichnis daher über den Kontextparameter`javax.faces.WEBAPP_RESOURCES_DIRECTORY`in der`web.xml`anpassen. Im Beispiel in Listing**Konfiguration des Verzeichnisses für Ressourcen**liegt das Verzeichnis innerhalb von`/WEB-INF`, wodurch es von außen nicht zugänglich ist.

```
<context-param>
  <param-name>
    javax.faces.WEBAPP_RESOURCES_DIRECTORY
  </param-name>
  <param-value>/WEB-INF/resources</param-value>
</context-param>
```

Nachdem die Ressourcen nur intern von JSF aus diesem Verzeichnis gelesen werden und nach außen unter einer anderen URL zur Verfügung stehen, ist das kein Problem für die Applikation. Der Pfad für das Beispiel mit der Bibliothek von oben sieht dann folgendermaßen aus:

```
/WEB-INF/resources/images/image.png
```

Das Auflösen der Ressourcen und das Ausliefern der Daten an den Client übernimmt die Klasse`Resource-Handler`. Wie viele andere Teile von JSF kann auch der`Resource-Handler`über die`faces-config.xml`mit einer eigenen Implementierung dekoriert werden. Denkbar wären Implementierungen, die Ressourcen aus einer Datenbank holen oder dynamisch erzeugen - der Fantasie sind keine Grenzen gesetzt.

5.2 Ressourcen im Einsatz

Es gibt mehrere Wege, ab JSF 2.0 Ressourcen in die Seite einzubinden. Folgende Standardkomponenten verfügen über die Attribute`name`und`library`, um direkt die auszugebende Ressource zu referenzieren:

- `h:graphicImage`gibt den Link auf ein Bild aus.
- `h:outputScript`gibt den Link auf ein Skript aus.
- `h:outputStylesheet`gibt den Link auf ein Stylesheet aus.

Ressourcen können auch direkt über einen EL-Ausdruck im Attribut`value`der entsprechenden Komponente referenziert werden - das implizite Objekt`resource`dient diesem Zweck. Eigenschaften dieses Objekts werden beim Auswerten des Ausdrucks als Ressourcenbezeichner interpretiert. Dieser Bezeichner kann der Name der Ressource oder der Name der Bibliothek gefolgt vom Namen der Ressource mit einem Doppelpunkt als Trennzeichen sein.
Hier nochmals das Beispiel aus dem letzten Abschnitt, diesmal allerdings über einen EL-Ausdruck realisiert:

```
<h:graphicImage value="#{resource['images:image.png']}"/>
```

Kommt als Seitendeklarationssprache Facelets zum Einsatz, kann dieser EL-Ausdruck sogar direkt im HTML-Code verwendet werden:

```
<img src="#{resource['images:image.png']}"/>
```

Die dritte Methode zum Einbinden von Ressourcen erfolgt im Java-Code und ist vor allem für Entwickler von Komponenten interessant. Mit den beiden Annotationen`@ResourceDependency`und`@ResourceDependencies`können Abhängigkeiten zu Ressourcen bereits in der Komponenten- beziehungsweise Rendererklasse definiert werden. Folgender Code verknüpft zum Beispiel eine Komponente mit der Ressource`script.js`aus der Bibliothek`scripts`:

```
@ResourceDependency(name="script.js", library="scripts")
public class MyComponent extends UIComponentBase {
...
}
```

Diese Methode wird hier nur der Vollständigkeit halber erwähnt. Weiterführende Informationen zum Thema Komponentenentwicklung und Ressourcen finden Sie in Abschnitt**Sektion: Rendererklasse schreiben**.

5.3 Positionierung von Ressourcen

Einen wichtigen Aspekt des Ressourcenmanagements von JSF haben wir bis jetzt noch nicht erwähnt. Stellen Sie sich vor, Sie benutzen eine Komponente aus einer Komponentenbibliothek, die ein spezielles Skript oder Stylesheet verwendet. Wie kommt diese Ressource dann in die Ansicht? Als Anwender der Bibliothek wollen wir uns nicht darum kümmern. Bei einigen Ressourcen wie Stylesheets und manchen Skripten kommt noch hinzu, dass sie an speziellen Stellen der Ansicht ausgegeben werden müssen, damit beim Benutzer die Seite richtig funktioniert. Wie die Verbindung zwischen Komponente und Ressource modelliert wird, haben wir bereits im letzten Abschnitt gezeigt. Das erklärt aber noch nicht, wie ein Link auf die Ressource in die Ansicht übernommen wird.
JSF erlaubt die Positionierung einzelner Ressourcen in definierten Bereichen der Ansicht wie dem Head oder dem Body. Damit JSF diese Bereiche richtig identifizieren kann, gibt es folgende neue Komponenten

in der*HTML-Tag-Library*:

- `h:head`umschließt den Head-Bereich der Seite.
- `h:body`umschließt den Body-Bereich der Seite.

Beim Einbinden einer Ressource kann einer dieser Bereiche direkt adressiert werden. Dafür existiert in`@ResourceDependency`das Element`target`und in`h:outputScript`ein gleichnamiges Attribut. Die erlaubten Werte sind`head`,`body`und`form`, wobei*Stylesheets*allerdings unabhängig von der Angabe immer im Head ausgegeben werden (gemäß HTML-Standard müssen*Stylesheets*im Head-Bereich verlinkt werden, damit der Quelltext der Seite gültiges HTML bleibt).
Damit das Positionieren von Ressourcen funktioniert, ist es unbedingt nötig,`h:head`und`h:body`in allen Ansichten einzusetzen. Mit Facelets können Sie das in einem Template zentral für alle Seiten erledigen. Wie das funktioniert, erfahren Sie in Abschnitt**Sektion: Templating**.
Sehen wir uns anhand eines Beispiels an, wie das Positionieren von Ressourcen in der Praxis aussieht. Das folgende Dokument enthält im Body-Bereich ein Stylesheet ohne Positionsangabe und ein Skript mit der Position`head`:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
  <title>Ressourcen-Test</title>
</h:head>
<h:body>
  <h:outputStylesheet name="style.css"/>
  <h:outputScript name="test.js" target="head"/>
  <h:outputText value="Test"/>
</h:body>
</html>
```

Hier der gerenderte HTML-Code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Ressourcen-Test</title>
  <link type="text/css" rel="stylesheet"
      href="/app/javax.faces.resource/style.css.jsf"/>
  <script type="text/javascript"
      src="/app/javax.faces.resource/test.js.jsf">
  </script>
</head>
<body>
  Test
</body>
</html>
```

Der Code zeigt, dass beide Ressourcen, das Skript und das Stylesheet, im Head-Bereich der Seite gerendert wurden. Mit Komponenten aus einer Bibliothek funktioniert das genauso - vorausgesetzt, der Entwickler hat die Komponente mit`@ResourceDependency`annotiert und das`target`-Attribut auf den Wert`head`gesetzt. Wenn Sie dann`h:head`und`h:body`auf der Seite einsetzen, erledigt JSF den Rest.

5.4 Identifikation

von Ressourcen -- Teil 2

In Abschnitt[Sektion: Identifikation von Ressourcen -- Teil 1]haben wir kurz darauf hingewiesen, dass JSF bei Ressourcen und Bibliotheken Versionierung und Lokalisierung unterstützt.
Versionsnummern sind bei Bibliotheken und Ressourcen mit durch Unterstrich (_) getrennte Zahlen wie`1_0`oder`1_0_1`angegeben. Sie werden, getrennt durch einen Schrägstrich (/), an den Namen der Ressource oder der Bibliothek angehängt. Bei Ressourcennamen kann die Version zusätzlich eine Dateierweiterung wie`.png`oder`.css`aufweisen. Warum das so ist, sehen wir gleich. Hier unser Beispiel mit Versionsnummern für die Bibliothek und die Ressource:

```
<h:graphicImage library="images/1_0"
    name="image.png/1_1.png"/>
```

JSF interpretiert Bibliotheksversionen beim Auflösen als ein weiteres Verzeichnis im Pfad der Ressource. Ressourcenversionen werden dahingegen etwas anders behandelt. Existiert eine Ressource in mehreren Versionen, wird der Name der Ressource zu einem Verzeichnis, und die einzelnen Versionen sind die eigentlichen Ressourcendateien. Der neue Pfad der Bilddatei lautet dann folgendermaßen:

```
/resources/images/1_0/image.png/1_1.png
```

In den meisten Fällen ist es aber nicht nötig, die Versionsnummern beim Einsatz von Ressourcen zu definieren. Wenn keine expliziten Versionen angegeben sind, verwendet JSF automatisch die Ressource mit der höchsten Versionsnummer.
Um die verschiedenen Kombinationen von Ressourcennamen, Bib-liotheksnamen und Versionsnummern zu veranschaulichen, wollen wir unser Beispiel erweitern. Abbildung**Beispiele von Ressourcen**zeigt das Ressourcenverzeichnis der Anwendung mit einer Bibliothek und den zwei Ressourcen`image.png`und`new.png`. Links neben dem Namen der Bilddatei ist jeweils das Bild selbst dargestellt.

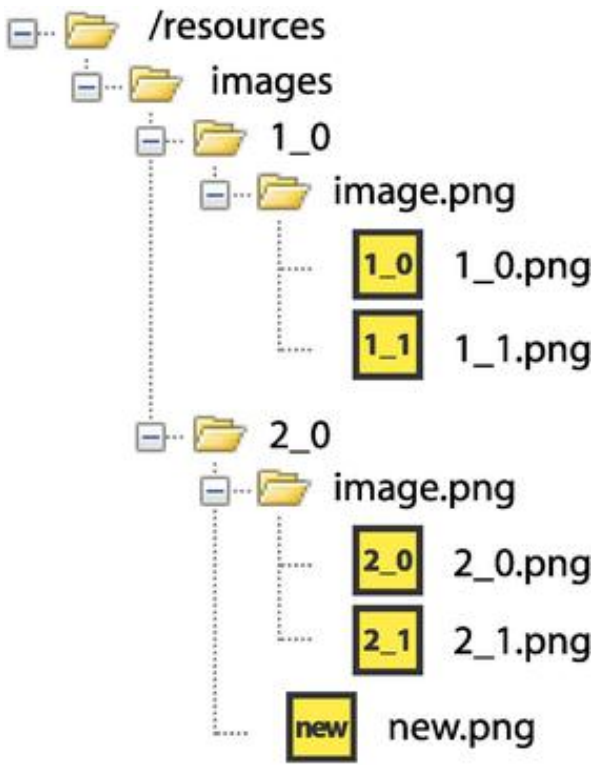


Abbildung:Beispiele von Ressourcen

Basierend auf dem Verzeichnisbaum aus Abbildung**Beispiele von Ressourcen**wollen wir nun verschiedene Ressourcen auflösen und uns das Ergebnis ansehen. Tabelletab:resource-resolutionzeigt für eine Reihe von Kombinationen aus Bibliotheks- und Ressourcenname, welches Bild dadurch angezeigt wird. Beachten Sie bitte vor allem Namen ohne Versionsangaben und wie diese immer zur höchsten Version aufgelöst werden.

Bibliothek	Ressource	
images	image.png	file=grafiken/resources-2_1.eps
images	image.png/2_1.png	file=grafiken/resources-2_1.eps
images/20	– image.png	file=grafiken/resources-2_1.eps
images/20	_ image.png/2_1.png	file=grafiken/resources-2_1.eps
images	image.png/2_0.png	file=grafiken/resources-2_0.eps
images/10	_ image.png/1_0.png	file=grafiken/resources-1_0.eps
images/10	– image.png	file=grafiken/resources-1_1.eps
images	new.png	file=grafiken/resources-new.eps

Zum Schluss wollen wir noch die Lokalisierung von Ressourcen besprechen. JSF sucht beim Auflösen von Ressourcen nach folgendem Eintrag im*Application-Message-Bundle*:

```
javax.faces.resource.localePrefix=<Wert>
```

Falls dieser Eintrag für das aktuelle Locale gesetzt ist, wird dessen Wert als Teil des Pfads der Ressourcendatei interpretiert. Ist der Wert im deutschen Resource-Bundle beispielsweise aufdegesetzt, sieht der Pfad zu unserem Bild wie folgt aus:

```
/resources/de/images/image.png
```

Weiterführende Informationen zur Internationalisierung und zum Application-Message-Bundle finden sich in Abschnitt**Sektion: Internationalisierung**.

5.5 Ressourcen in MyGourmet 12

Die Umstellung von*MyGourmet*auf Ressourcen ist im momentanen Stand der Entwicklung trivial - wir erstellen dafür kein neues Beispiel.
Die wichtigste Änderung ist die Umstellung des Haupttemplates`template.xhtml`auf die Elemente`h:head`und`h:body`. Dafür müssen aber nur die entsprechenden HTML-Elemente ersetzt werden. Sobald das geschehen ist, können auch das Stylesheet und das Logo ins Verzeichnis`/resources`verschoben und als Ressourcen verwendet werden.
Die Ressourcenverwaltung wird erst ab dem nächsten Beispiel*MyGourmet 13*interessant, wenn sich alles um die in JSF 2.0 eingeführten Kompositkomponenten dreht. Mehr dazu erfahren Sie in Abschnitt**Sektion: Kompositkomponenten**.

5.6 Resource-Library-Contracts

JSF 2.2:Die Pläne für JSF 2.2 sahen ursprünglich ein "Multi-Templating" genanntes System zur Unterstützung von Themes vor. Dieses System hat es aber nicht bis in die finale Version der Spezifikation geschafft. Übrig geblieben sind die sogenannten Resource-Library-Contracts, eine Notlösung mit grundlegenden Features zur Unterstützung austauschbarer Templates und mit Potenzial für künftige Erweiterungen.
Resource-Library-Contracts gruppieren Templates und Ressourcen zu einer austauschbaren Einheit. Ein Contract besteht aus einem oder mehreren Facelets-Templates, deren mit`ui:insert`definierten ersetzbaren Bereichen und einer beliebigen Anzahl von Ressourcen. Die Templates, deren ersetzbare Bereiche und die Ressourcen bilden somit eine (informelle) Schnittstelle, aus der hervorgeht, wie ein Resource-Library-Contract eingesetzt werden kann. Alle Resource-Library-Contracts mit derselben Schnittstelle können gegeneinander ausgetauscht werden.

5.6.1 Ein erstes Beispiel

Ein Resource-Library-Contract ist im Grunde genommen ein Container für Templates und Ressourcen, der in einem speziellen Verzeichnis abgelegt wird. JSF sucht Resource-Library-Contracts standardmäßig an folgenden Stellen einer Anwendung (in der angegebenen Reihenfolge):

1. Im Wurzelverzeichnis der Webapplikation unter`/contracts`

2. In `/META-INF/contracts` und somit auch in allen Jar-Dateien im Classpath

Abbildung **Resource-Library-Contract in der Applikation** zeigt den Inhalt des Contracts mit dem Namen `contract1` im Verzeichnis `/contracts` der Webapplikation.

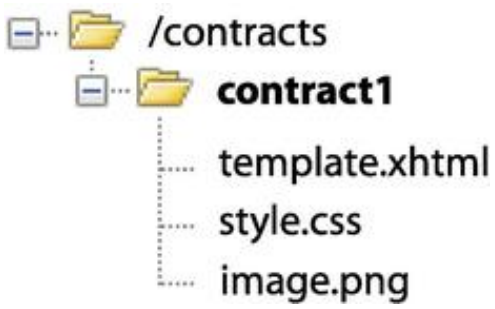


Abbildung: Resource-Library-Contract in der Applikation

Dieser Resource-Library-Contract beinhaltet das Template `tem-plate.xhtml`, das Stylesheet `style.css` und das Bild `image.png`. Das Template definiert die ersetzbaren Bereiche `header` und `content` und bindet das Stylesheet als JSF-Ressource ein, wie Listing **Template für Resource-Library-Contract** zeigt.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head><title>Contract Template</title></h:head>
<h:body>
  <h:outputStylesheet name="style.css"/>
  <div class="header">
    <ui:insert name="header"/>
  </div>
  <div class="content">
    <ui:insert name="content"/>
  </div>
</h:body>
</html>
```

Die Schnittstelle dieses Contracts besteht somit aus dem Template `template.xhtml` und dessen ersetzbaren Bereichen `header` und `content`. Das Stylesheet und das Bild beachten wir momentan nicht weiter (mehr dazu in Abschnitt **[Sektion: Ressourcen aus Resource-Library-Contracts]**). Mit diesem Wissen können wir das Template aus dem Contract bereits in einer Facelets-Seite einsetzen, wie Listing **Template-Client für Resource-Library-Contract** zeigt.

```
<ui:composition template="/template.xhtml">
  <ui:define name="header">
    <h1>Überschrift</h1>
  </ui:define>
  <ui:define name="content">
    <p>Beliebiger Inhalt</p>
  </ui:define>
</ui:composition>
```

Das Template aus dem Resource-Library-Contract wird nur über seinen Namen referenziert. Dabei stellt sich die Frage, wie es aus dem Contract in die Seite kommt. Dazu müssen wir zwei neue Aspekte von JSF 2.2 berücksichtigen. Zum einen stellt JSF standardmäßig alle Resource-Library-Contracts, die beim Starten der Applikation gefunden werden, allen Seiten zur Verfügung. Nachdem es in unserem Beispiel nur einen Contract gibt, ist das genau das gewünschte Verhalten. Zum anderen versucht JSF ab Version 2.2,

XHTML-Dateien und Ressourcen immer zuerst aus den der Seite zugeordneten Contracts zu laden. JSF sucht also in unserem Fall das Template`template.xhtml`wie gewünscht zuerst im Contract`contract1`. Selbst wenn im Wurzelverzeichnis der Applikation auch ein Template mit dem Name`template.xhtml`existiert, kommt zuerst jenes aus dem Contract zum Zug.
Der Einsatz von Resource-Library-Contracts wird erst richtig interessant, wenn mehrere Contracts mit derselben Schnittstelle in einer Applikation zur Verfügung stehen. Damit lassen sich zum Beispiel Templates mit unterschiedlichem Styling für verschiedene Bereiche der Anwendung realisieren. In diesem Fall muss natürlich gewährleistet sein, dass es eine eindeutige Zuordnung von Contracts zu Seiten gibt. Abschnitt**[Sektion: Zuordnung von Resource-Library-Contracts]**zeigt, wie das funktioniert.
Packt man Resource-Library-Contracts in Jar-Dateien, ergeben sich weitere interessante Anwendungsfälle. Contracts lassen sich dadurch sehr einfach durch das Ersetzen einer Jar-Datei austauschen oder in mehreren Applikationen einsetzen. Abschnitt**[Sektion: Resource-Library-Contracts in Jar-Dateien]**zeigt, was Sie dabei beachten müssen.

5.6.1.1 Konfiguration des Contracts-Verzeichnisses

Wie bereits erwähnt sucht JSF Resource-Library-Contracts in der Webanwendung standardmäßig im Verzeichnis`/contracts`. Dieser Pfad kann aber in der`web.xml`über den Kontextparameter`javax.faces.WEBAPP_CONTRACTS_DIRECTORY`angepasst werden. Im Beispiel in Listing**Konfiguration des Verzeichnisses für Contracts**liegt das Verzeichnis innerhalb von`/WEB-INF`, wodurch es auch nicht mehr von außen zugänglich ist.

```
<context-param>
  <param-name>
    javax.faces.WEBAPP_CONTRACTS_DIRECTORY
  </param-name>
  <param-value>/WEB-INF/contracts</param-value>
</context-param>
```

5.6.2 Ressourcen aus Resource-Library-Contracts

Neben Templates können Resource-Library-Contracts auch eine beliebige Anzahl von Ressourcen enthalten. Der Resource-Library-Contract aus Abschnitt**[Sektion: Ein erstes Beispiel]**beinhaltet zum Beispiel neben dem Template auch noch die Ressourcen`style.css`und`image.png`. Diese Ressourcen können über ihren Namen in eine Seite eingebunden werden. Das Stylesheet wird ja bereits mith`:outputStylesheet`im Template verwendet, wie Listing**Template für Resource-Library-Contract**zeigt.
Das Einbinden von Ressourcen aus einem Contract funktioniert aber nicht nur innerhalb des Contracts. Wie schon bei den Tem-plates versucht JSF ab Version 2.2, auch Ressourcen immer zuerst aus den einer Seite zugeordneten Contracts zu laden. Listing**Template-Client für Resource-Library-Contract mit Ressource**zeigt nochmals den Template-Client aus Listing**Template-Client für Resource-Library-Contract**- diesmal allerdings zusätzlich mit dem Bild`image.png`.

```
<ui:composition template="/template.xhtml">
  <ui:define name="header">
    <h1>Überschrift</h1>
  </ui:define>
  <ui:define name="content">
```

```
    <p>Beliebiger Inhalt</p>
    <h:graphicImage name="image.png"/>
  </ui:define>
</ui:composition>
```

5.6.3 Zuordnung von Resource-Library-Contracts

Existieren mehrere Contracts mit derselben Schnittstelle in einer Applikation, muss die Zuordnung von Contracts zu Seiten angepasst werden. Andernfalls stellt JSF wiederum alle Contracts in allen Seiten zur Verfügung, was zu unvorhersehbaren Ergebnissen führen kann.
JSF baut beim Starten der Applikation eine Liste aller verfügbaren Resource-Library-Contracts auf. Diese Liste hat allerdings keine eindeutig definierte Ordnung. Nachdem JSF aber beim Laden von XHTML-Dateien und Ressourcen die Liste der für die Seite verfügbaren Contracts durchgeht und einfach den ersten Treffer verwendet, bleibt es also in gewissem Maß dem Zufall überlassen, welcher Contract zum Einsatz kommt.
Die Zuordnung von Resource-Library-Contracts zu Seiten der Applikation lässt sich auf zwei Arten definieren. Es kann eine grundlegende Zuordnung für einzelne Seiten oder für ganze Seitenbereiche in der`faces-config.xml`gemacht werden. Für einzelne Seiten kann die Zuordnung mit`f:view`zudem gezielt definiert werden.
Die Zuordnung von Contracts zu Seiten in der`faces-config.xml`basiert auf URL-Mustern. Mögliche Werte für die Muster umfassen einzelne Seiten wie`/page1.xhtml`, Seitenbereiche wie`/customer/*`oder die gesamte Applikation mit*. Pro Muster wird eine Liste aller anzuwendenden Contract-Namen angegeben. Bei der Auswertung berücksichtigt JSF zuerst immer exakte Übereinstimmungen für konkrete Seiten und ansonsten das längste zutreffende Muster.
Die Konfiguration erfolgt im Element`resource-library-contracts`innerhalb`application`. Dort wird für jede Kombination von URL-Muster und Contract-Namen ein`contract-mapping`-Element eingefügt. Das URL-Muster kommt dabei ins Element`url-pattern`und die kommaseparierte Liste von Contract-Namen ins Element`contracts`. Mit der in Listing**Zuordnung von Resource-Library-Contracts in der faces-config.xml**gezeigten Konfiguration wird allen Seiten, deren View-ID mit`/special/`beginnt, der Contract`layout-special`und alle anderen Seiten der Contract`layout`zugeordnet.

```
<application>
  <resource-library-contracts>
    <contract-mapping>
      <url-pattern>/special/*</url-pattern>
      <contracts>layout-special</contracts>
    </contract-mapping>
    <contract-mapping>
      <url-pattern>*</url-pattern>
      <contracts>layout</contracts>
    </contract-mapping>
  </resource-library-contracts>
</application>
```

Die Zuordnung von Resource-Library-Contracts kann für einzelne Seiten gezielt überschrieben werden. Dazu gibt es im Tag`f:view`das neue Attribut`contracts`, in dem eine kommaseparierte Liste von Contract-Namen angegeben werden kann. Listing**Zuordnung von Resource-Library-Contracts mit f:view**zeigt ein Beispiel.

```
<f:view contracts="contract1">
  <ui:composition template="/template.xhtml">
    ...
  </ui:composition>
</f:view>
```

Im Attribut`contracts`kann auch eine Value-Expression angegeben werden, wie das folgende Beispiel zeigt:

```
<f:view contracts="#{bean.contracts}">
  ...
</f:view>
```

Die referenzierte Bean-Eigenschaft muss die Liste der Contract-Namen in Form einer Zeichenkette zurückliefern. Damit besteht die Möglichkeit, die einer Seite zugeordneten Contracts dynamisch zu ändern. Wenn Sie jetzt auf die Idee kommen, das global in einem Template zu definieren, müssen wir Sie leider enttäuschen. JSF erlaubt die Definition der Contracts über`f:vie`wexplizit nur in der ersten Datei, die beim Aufbau der Seite verarbeitet wird (also im Template-Client).

5.6.4 Resource-Library-Contracts in Jar-Dateien

Resource-Library-Contracts in Jar-Dateien bieten einige Vorteile. Sie können einerseits in mehreren Applikationen eingesetzt werden und lassen sich andererseits beim Bauen der Applikation einfach austauschen. Dabei sollten Sie allerdings darauf achten, dass der neue Contract dieselbe Schnittstelle definiert.
Jar-Dateien mit Resource-Library-Contracts sind schnell erstellt. Dazu müssen die Contracts nur in das Verzeichis`/META-INF/contracts`kopiert werden. Für den aus Abschnitt**[Sektion: Ein erstes Beispiel]**bekannten Contract`contract1`sieht die Verzeichnisstruktur dann wie in Abbildung**Resource-Library-Contract in Jar-Datei**aus.

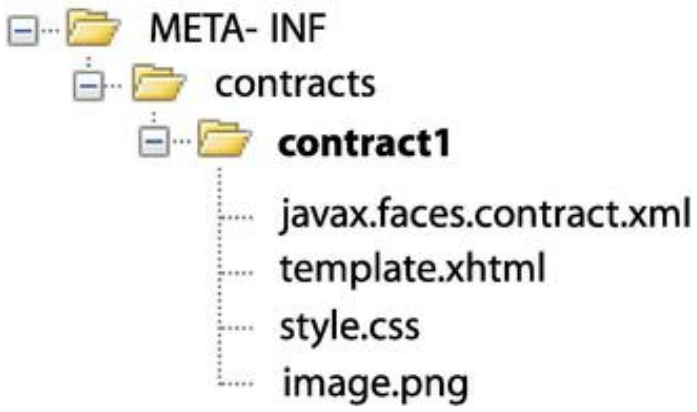


Abbildung:Resource-Library-Contract in Jar-Datei

JSF berücksichtigt Resource-Library-Contracts aus Jar-Dateien nur dann, wenn sie eine Datei mit dem Name`javax.faces.contract.xml`beinhalten. In JSF 2.2 kann diese Datei noch leer sein, was aber in zukünftigen Versionen von JSF nicht so bleiben muss.

5.6.5 MyGourmet 12 mit Resource-Library-Contracts

In diesem Abschnitt präsentieren wir eine Variante von*MyGourmet 12*, bei der das Basis-Template der Applikation mithilfe von Resource-Library-Contracts verwaltet wird. Das erlaubt uns die Definition von zwei Contracts mit unterschiedlichem Design für den Kundenbereich und den Anbieterbereich der Applikation. Die beiden Designs werden jeweils als Contract realisiert:`base-color`für das bereits bekannte Design und`base-gray`für eine in Grau gehaltene Variante. Beide Contracts beinhalten folgende Ressourcen:

- Das bereits bekannte Template`template.xhtml`ohne Änderungen
- Das für jeden Contract angepasste Stylesheet`mygourmet.css`
- Das für jeden Contract farblich angepasste Bild`logo.png`

Abbildung**Resource-Library-Contracts in15mmMyGourmet 12**zeigt die Resource-Library-Contracts`base-color`und`base-gray`in der Verzeichnisstruktur der Applikation.

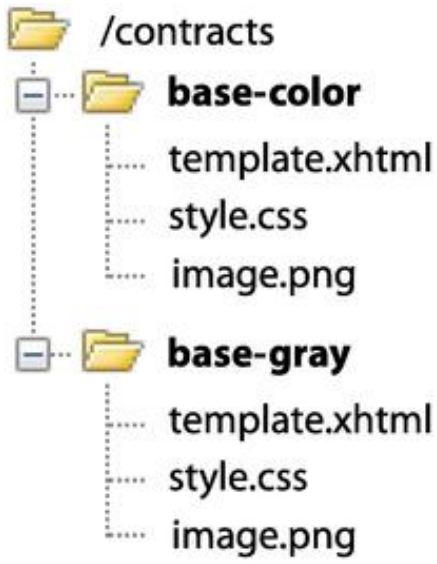


Abbildung:Resource-Library-Contracts in15mmMyGourmet 12

Um die Zuordnung von Contracts zu Seiten einfacher zu machen, haben wir die Seiten im Kundenbereich nach`/views/customer`und die Seiten im Anbieterbereich nach`/views/provider`verschoben. Damit können wir den Kundenseiten den Contract`base-color`und den Anbieterseiten den Contract`base-gray`zuordnen. Listing**Konfiguration in MyGourmet 12**zeigt die entsprechende Konfiguration in der`faces-config.xml`.

```
<resource-library-contracts>
  <contract-mapping>
    <url-pattern>/views/customer/*</url-pattern>
    <contracts>base-color</contracts>
  </contract-mapping>
  <contract-mapping>
    <url-pattern>/views/provider/*</url-pattern>
    <contracts>base-gray</contracts>
  </contract-mapping>
</resource-library-contracts>
```

Die Seiten der Applikation benutzen weiterhin das Template`customerTemplate.xhtml`. Dort müssen wir noch den Pfad des Basis-Templates auf den Wert`/template.xhtml`ändern - dieses Template kommt jetzt wie das Stylesheet aus dem Contract. Das Logo wird als Ressource mit dem Namen`logo.png`ebenfalls aus dem Contract eingebunden.

6 Die eigene JSF-Komponente

Seinem Ruf als Komponentenframework wird*JavaServer Faces*mehr als gerecht: Komponenten sind ein essenzieller Bestandteil von JSF und bilden einen zentralen Erweiterungspunkt. Es gibt wohl bei keinem anderen Webframework so viele Möglichkeiten zur Erweiterung und zum Erstellen von eigenen Komponenten, die harmonisch mit dem Framework interagieren. Die Standardkomponenten und diverse Erweiterungen und Komponentenbibliotheken aus dem JSF-Umfeld decken bereits ein beachtliches Funktionsspektrum ab. Die meisten Entwickler werden aber früher oder später auf Anwendungsfälle stoßen, die damit nicht realisierbar sind. Spätestens dann ist eine eigene Komponente sinnvoll.
Vor Version 2.0 von JSF war die Komponentenentwicklung jedoch relativ aufwendig. Es galt immer abzuwägen, ob sich dieser Schritt im konkreten Fall auszahlt oder sich das Problem nicht auch anderweitig lösen lässt. JSF machte in Version 2.0 in diesem Bereich einen großen Schritt auf die Entwickler zu. Mit den neuen Kompositkomponenten (Composite-Components) existiert eine wirklich sehr einfache Möglichkeit, Komponenten deklarativ zu erstellen - ohne eine Zeile Java-Code oder XML-Konfiguration zu schreiben. Wie das funktioniert, zeigen wir Ihnen in Abschnitt**[Sektion: Kompositkomponenten]**.
So wichtig die Kompositkomponenten auch sind, sie können nicht jedes Problem lösen. Speziell komplexere Aufgaben lassen sich weiterhin nur mit den höchstflexiblen klassischen Komponenten realisieren. Dass aber auch deren Erstellung kein Hexenwerk ist, zeigen wir Ihnen in Abschnitt**[Sektion: Klassische Komponenten]**. Mit JSF ab Version 2.0 und Facelets ist es im Vergleich sogar noch etwas einfacher geworden.
In Abschnitt**[Sektion: Kompositkomponenten und klassische Komponenten kombinieren]**gehen wir noch einen Schritt weiter und kombinieren Kompositkomponenten mit klassischen Komponenten. Mit diesem Ansatz lassen sich Kompositkomponenten bei Bedarf sehr einfach mit Java-Code erweitern. Sie werden sehen, dass die beiden Konzepte perfekt miteinander harmonieren und die Entwicklung von eigenen Komponenten damit noch flexibler wird.
In Abschnitt**[Sektion: Alternativen zur eigenen Komponente]**zeigen wir Ihnen, wie Sie einzelne Teile einer existierenden Komponente ersetzen: Es muss nicht immer eine komplette Komponente sein. Das Beispiel*MyGourmet 13*fasst alle im Laufe des Kapitels gemachten Änderungen zusammen und wird in Abschnitt**[Sektion: *MyGourmet 13*: Komponenten und Services]**behandelt.
Sie wollen eine eigene Komponentenbibliothek mit all Ihren Komponenten erstellen? Kein Problem, Abschnitt**[Sektion: Die eigene Komponentenbibliothek]**zeigt, wie das funktioniert. In Abschnitt**[Sektion: *MyGourmet 13*mit Komponentenbibliothek]**finden Sie nochmals*MyGourmet 13*-diesmal allerdings mit Komponentenbibliothek.

6.1 Kompositkomponenten

Kompositkomponenten waren aus unserer Sicht eines der wichtigsten neuen Features von JSF 2.0. Entwickler erhalten durch die Verbindung von Facelets und Ressourcen die Möglichkeit, Komponenten aus beinahe beliebigen Seitenfragmenten aufzubauen - daher auch der Name. Eine Kompositkomponente ist im Grunde nichts anderes als ein XHTML-Dokument, das in einer Ressourcenbibliothek abgelegt ist und die Komponente deklariert.
Nach einer kurzen Einführung anhand eines Beispiels in Abschnitt**[Sektion: Eine erste Kompositkomponente]**werden wir in den Abschnitten**[Sektion: Der Bereich cc:interface]**und**[Sektion: Der Bereich cc:implementation]**etwas genauer auf die Deklaration von Kompositkomponenten und die einzelnen Tags der Composite-Tag-Bibliothek eingehen. Anschließend zeigt Abschnitt**[Sektion: Ressourcen in Kompositkomponenten]**die Kombination von Kompositkomponenten und Ressourcen. In den Abschnitten**[Sektion: Die Komponente mc:panelBox]**bis**[Sektion: Die Komponente mc:inputSpinner]**finden Sie einige praxisorientierte Beispiele und Abschnitt**[Sektion: Fallstricke in der**

Praxis]zeigt abschließend noch einige Fallstricke beim Einsatz von Kompositkomponenten.

6.1.1 Eine erste Kompositkomponente

Die Definition des Namensraums und des Tags der Komponente ergibt sich per Konvention aus der Ressourcenbibliothek und dem Namen des XHTML-Dokuments. Als erstes Beispiel wandeln wir die Box in der Seitenleiste in eine Kompositkomponente mit dem Namen`panelBox`um. Da alle im Laufe dieses Abschnitts erstellten Komponenten in der Bibliothek`mygourmet`landen, legen wir zuerst dieses Verzeichnis unter`/resources`an. Darin erstellen wir dann die Deklaration der Komponente mit dem Namen`panelBox.xhtml`. Abbildung**Ressource der Kompositkomponente panelBox**zeigt den Verzeichnisbaum. Ausführlichere Informationen zum Thema Ressourcen finden Sie in Kapitel**Kapitel: Verwaltung von Ressourcen**.

Abbildung:Ressource der Kompositkomponente panelBox

Sobald`panelBox.xhtml`in der Bibliothek`mygourmet`existiert, ist die Komponente einsatzfähig. Das Einbinden erfolgt wie bei allen Komponenten über den Namensraum und das Tag. Per Konvention leitet sich der Namensraum von Kompositkomponenten aus dem Präfix`http://xmlns.jcp.org/jsf/composite/`gefolgt vom Bibliotheksnamen - in unserem Fall`mygourmet`- ab. Das Tag erhält seinen Namen von der Deklaration und lautet`panelBox`. Listing**Einbinden einer Kompositkomponente**zeigt, wie die Komponente in einer Seitendeklaration zum Einsatz kommt. Das gewählte Präfix`mc`ist wie immer beliebig, bietet sich aber als Abkürzung von*MyGourmet-Component*san.

```
<html xmlns:mc="http://xmlns.jcp.org/jsf/composite/mygourmet">
  ...
  <mc:panelBox title="Panel-Header">
    <h:outputText value="Ein Text"/>
  </mc:panelBox>
  ...
</html>
```

Da wir die Komponente bereits einbinden können, wird es Zeit, einen Blick auf die Deklaration zu werfen. Listing**Kompositkomponente panelBox**zeigt eine erste Version von`panelBox.xhtml`, deren Inhalt stark an das zuvor eingesetzte Template`sideBox.xhtml`(siehe Abschnitt**Sektion: Mehrere Templates pro Seite**) angelehnt ist.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:cc="http://xmlns.jcp.org/jsf/composite"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<cc:interface>
  <cc:attribute name="title"/>
</cc:interface>
<cc:implementation>
  <div class="side_box">
```

```
    <p class="header">#{cc.attrs.title}</p>
    <cc:insertChildren/>
  </div>
</cc:implementation>
</ui:composition>
```

JSF fasst im Namensraum`http://xmlns.jcp.org/jsf/composite`alle zur Deklaration einer Kompositkomponente notwendigen Tags zusammen. Wenn Sie Listing**Kompositkomponente panelBox**näher betrachten, werden Sie bemerken, dass es sich wie schon beim Templating um ein XHTML-Dokument mit dem Wurzelelement`ui:composition`handelt. Für die Deklaration der Komponente sind aber nur die mit`cc:interface`und`cc:implementation`umschlossenen Bereiche relevant. Facelets ignoriert den Rest.
Der Bereich innerhalb von`cc:interface`definiert die Schnittstelle der Komponente nach außen. Bei unserer Box fällt dieser Bereich relativ klein aus und umfasst nur ein Attribut mit dem Namen`title`. Der Wert dieses Attributs wird über das Tag der Komponente gesetzt, wie bereits Listing**Einbinden einer Kompositkomponente**gezeigt hat. Wie dieses Attribut innerhalb der Komponente verwendet wird, zeigen wir gleich.
Im Bereich`cc:implementation`folgt die Implementierung der Komponente, die aus einem beliebigen Mix aus JSF-Tags, HTML-Tags und Tags der Composite-Tag-Bibliothek zusammengesetzt sein kann. Unsere Box besteht aus einem`div`-Element, in dem als erster Absatz der Titel ausgegeben wird. Da der Titel ein Attribut der Komponente ist, müssen wir irgendwie auf dessen Wert zugreifen. Dazu führt JSF 2.0 das implizite Objekt`cc`als Referenz auf die aktuelle Kompositkomponente ein. Der Zugriff auf das Attribut erfolgt mit der Value-Expression`#{cc.attrs.title}`, wobei die Eigenschaft`attrs`eine Map aller Attribute zurückliefert. Das Tag`cc:insertChildren`veranlasst JSF dazu, sämtliche Inhalte einzufügen, die beim Einsatz der Komponente innerhalb des Tags angegeben werden. Im Beispiel aus Listing**Einbinden einer Kompositkomponente**ist das die`h:outputText`-Komponente mit dem Wert*Ein Text*.
Bleibt noch zu klären, wie JSF mit Kompositkomponenten in einer Seitendeklaration umgeht. Listing**Seitenleiste in MyGourmet mit panelBox**zeigt, wie der Inhalt der Seitenleiste aus*MyGourmet*mit der Komponente`panelBox`aussieht.

```
<mc:panelBox id="menu" title="#{msgs.menu_title}">
  <h:panelGrid columns="1">
    <h:link outcome="providerList"
        value="#{msgs.menu_provider_list}"/>
    <h:link outcome="showCustomer"
        value="#{msgs.menu_show_customer}"/>
  </h:panelGrid>
</mc:panelBox>
<mc:panelBox id="news" title="#{msgs.news_title}">
  <p>MyGourmet - jetzt mit Facelets und Templating</p>
</mc:panelBox>
```

Trifft Facelets beim Aufbau der Ansicht in der Seitendeklaration auf eine Kompositkomponente, wird im Standardfall eine Komponente vom Typ`UINamingContainer`erzeugt und in den Komponentenbaum eingefügt (Abschnitt**[Sektion: Kompositkomponenten und klassische Komponenten kombinieren]**zeigt, wie Sie an dieser Stelle eine eigene Komponente verwenden können). Diese Komponente bildet den Wurzelknoten für sämtliche weitere Inhalte der Kompositkomponente. Ihre Kinder werden allerdings erst beim weiteren Abarbeiten der Seitendeklaration aus dem Inhalt der Kompositkomponente selbst und dem Inhalt des Tags in der aufrufenden Seite erstellt und hinzugefügt. Diese Vorgehensweise ist mit dem Aufbau einer Ansicht mit Templates vergleichbar.
Was heißt das konkret für unser Beispiel? Nach dem Erstellen des Wurzelknotens setzt Facelets die Verarbeitung mit`panelBox.xhtml`fort. Nachdem das`div`-Element und der Absatz mit dem Titel verarbeitet wurden, kommt das Tag`cc:insertChildren`an die Reihe. Wie Sie in Listing**Seitenleiste in MyGourmet mit panelBox**sehen, haben beide`panelBox`-Tags Inhalte, die an dieser Stelle in den Komponentenbaum aufgenommen werden.

6.1.2 Der Bereich cc:interface

Der Bereich`cc:interface`definiert die Schnittstelle der Komponente nach außen und umfasst alle Merkmale, die für Benutzer der fertigen Komponente relevant sind. Neben der Definition von Attributen und Facets ist es auch möglich, das Verhalten einzelner interner Komponenten nach außen bekanntzugeben. Hier eine Liste aller Tags, die in der Schnittstelle einer Kompositkomponente verwendet werden können:

- `cc:attribute`
  Dieses Tag definiert ein Attribut der Kompositkomponente.
- `cc:facet`
  Dieses Tag definiert ein Facet der Kompositkomponente.
- `cc:valueHolder`
  Dieses Tag definiert einen Namen, unter dem eine interne Komponente verfügbar ist, die`ValueHolder`implementiert.
- `cc:editableValueHolder`
  Dieses Tag definiert einen Namen, unter dem eine interne Komponente verfügbar ist, die`EditableValueHolder`implementiert.
- `cc:actionSource`
  Dieses Tag definiert einen Namen, unter dem eine interne Komponente verfügbar ist, die`ActionSource2`implementiert.

Nach diesem kurzen Überblick folgen jetzt noch einige Details zu den Möglichkeiten der einzelnen Tags. Zu diesem Zweck finden Sie in Listing**Kompositkomponente simpleInput**die Kompositkomponente`simpleInput`, die aus einem Eingabefeld und einer Submit-Schaltfläche besteht.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:cc="http://xmlns.jcp.org/jsf/composite"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
  <cc:interface>
    <cc:attribute name="inputLabel" required="false"
        default="Input"/>
    <cc:attribute name="submitLabel" default="Submit"/>
    <cc:attribute name="action" required="true"
        targets="submit"/>
    <cc:attribute name="value" required="true"/>
    <cc:editableValueHolder name="input" targets="inputText"/>
    <cc:actionSource name="submit"/>
  </cc:interface>
  <cc:implementation>
    <h:outputLabel for="input" value="#{cc.attrs.inputLabel}"/>
    <h:inputText id="inputText" value="#{cc.attrs.value}"/>
    <h:commandButton id="submit"
        value="#{cc.attrs.submitLabel}"/>
  </cc:implementation>
</ui:composition>
```

6.1.2.1 Attribute

Wenden wir uns zuerst der Definition von Attributen mit dem Tag`cc:attribute`zu. Die Komponente`simpleInput`definiert eine Reihe von Attributen unterschiedlicher Ausprägung. Mit`required`kann gesteuert werden, ob das Attribut beim Einsatz der Komponente verpflichtend anzugeben ist oder nicht. Die Attribute`inputLabel`und`submitLabel`sind beide optional, da bei dem einen`required`explizit auf`false`gesetzt ist und beim anderen überhaupt fehlt. Für optionale Attribute kann im Attribut`default`ein Standardwert definiert werden.
Der Typ eines Attributs kann im Attribut`type`in Form eines voll qualifizierten Klassennamens angegeben werden. Der Standardwert für`type`ist`java.lang.Object`.
Attribute können auch Method-Expressions für Action-Methoden oder Event-Listener enthalten. Dazu muss lediglich im Attribut`method-signature`die Signatur der Methode angegeben werden. Dabei ist es wichtig, immer voll qualifizierte Klassennamen für den Rückgabewert und die Parameter zu verwenden. Wie das Beispiel in Listing**Method-Expressions als Attribute**zeigt, wird die übergebene Method-Expression analog zu anderen Attributen direkt über`cc.attrs`referenziert. Es macht keinen Sinn,`type`und`method-signature`gleichzeitig in`cc:attribute`zu verwendet. Sollte das dennoch einmal der Fall sein, wird der Wert von`method-signature`einfach ignoriert.

```
<cc:interface>
  <cc:attribute name="submitAction"
      method-signature="java.lang.String action()"/>
</cc:interface>
<cc:implementation>
  <h:commandButton action="#{cc.attrs.submitAction}"/>
</cc:implementation>
```

Attribute mit den Namen`action`,`actionListener`,`validator`und , die bereits aus vorherigen Kapiteln bekannt sind, nehmen hier eine Sonderstellung ein. JSF verknüpft in solchen Attributen Method-Expressions automatisch mit Komponenten im Bereich`cc:implementation`, deren IDs im Attribut`targets`angegeben sind. Der Wert von`targets`muss nicht auf eine ID beschränkt sein, sondern kann auch eine Liste von IDs enthalten, die durch Leerzeichen separiert sind. Das Attribut`method-signature`kann in diesen Fällen leer bleiben, da die Methoden-Signaturen ohnehin von JSF vorgegeben sind.
Im Beispiel aus Listing**Kompositkomponente simpleInput**ist das Attribut mit dem Namen`action`über den Wert in`targets`an die Komponente mit der ID`submit`gebunden. Das Attribut`action`im Tag`h:commandButton`darf in diesem Fall nicht explizit gesetzt sein. Der Vorteil dieser Variante ist, dass Benutzer der Komponente im Attribut`action`sowohl einen String als auch eine Method-Expression angeben können - genau so wie es zum Beispiel auch bei`h:commandButton`funktioniert.
So weit, so gut. Der Ansatz mit dem Attribut`targets`hat in JSF 2.0 allerdings einen entscheidenden Nachteil. Die spezielle Behandlung der oben genannten Attribute`action`,`actionListener`,`validator`und`valueChangeListener`funktioniert nur, wenn das Attribut der Kompositkomponente genau einen dieser Namen hat. Aus diesem Grund ist es mit JSF 2.0 zum Beispiel auch nicht möglich, mehr als ein "echtes"`action`-Attribut zu definieren. Abschnitt**[Sektion: Fallstricke in der Praxis]**zeigt, wie dieses Problem mit JSF 2.0 umgangen und ab JSF 2.1 gelöst werden kann.
An dieser Stelle möchten wir Sie noch darauf hinweisen, dass Facelets alle Attribute in der Kompositkomponente ablegt, die der Benutzer des Tags angibt - auch wenn sie nicht mit`cc:attribute`definiert sind. Wir raten Ihnen aber dazu, alle Attribute zu definieren. Damit geben Sie Benutzern der Kompositkomponente einen klaren Kontrakt über die Schnittstelle und das Verhalten in die Hand.

6.1.2.2 Facets

Mit`cc:facet`kann ein benanntes Facet für eine Kompositkomponente definiert werden. Der Name des Facets wird dabei im Attribut`name`angegeben. Ist das Attribut`required`explizit auf`true`gesetzt, muss der Benutzer der Komponente das Facet verpflichtend hinzufügen. Das Beispiel in Listing**Kompositkomponente simpleInput im Einsatz**zeigt den Bereich`cc:interface`einer

Kompositkomponente mit zwei Facet-Definitionen. Abschnitt[Sektion: Der Bereich cc:implementation]zeigt Details zu Facets in Kompositkomponenten.

```
<cc:interface>
  <cc:facet name="header" required="true"/>
  <cc:facet name="footer"/>
</cc:interface>
```

6.1.2.3 Verhaltensdefinitionen

Interne Komponenten im Bereich`cc:implementation`, die ein spezielles Verhaltens-Interface implementieren (siehe Abschnitt**Sektion: Verhaltens-Interfaces**für Details), können mit den Tags`cc:actionSource`,`cc:editableValueHolder`und`cc:valueHolder`nach außen bekannt gegeben werden. Damit bekommen Benutzer der Kompositkomponente die Möglichkeit, Objekte wie Event-Listener, Konverter oder Validatoren an diese Komponente zu binden. Das Beispiel aus Listing**Kompositkomponente simpleInput**definiert eine Action-Source mit dem Namen`submit`und einen Editable-Value-Holder mit dem Namen`input`.
Der Editable-Value-Holder mit dem Namen`input`ist über den Wert des Attribut`targets`an das Eingabefeld mit der ID`inputText`gebunden - hier ist auch eine Liste von IDs möglich, die durch Leerzeichen separiert sind. Wenn sich der Name und die ID der verbundenen Komponente nicht unterscheiden, kann das Attribut`targets`auch wegfallen. Bei der Action-Source mit dem Namen`submit`trifft genau das zu: Die Schaltfläche mit der ID`submit`ist an die Action-Source gebunden.
Bleibt noch zu klären, wie Benutzer von Kompositkomponenten Event-Listener, Konverter oder Validatoren an die Kompositkomponente anhängen können. In JSF 2.0 haben zu diesem Zweck`f:actionListener`,`f:valueChangeListener`sowie alle Konverter- und Validator-Tags in der Core-Tag-Library das Attribut`for`bekommen. Darin kann der Name eines Action-Listeners, eines Value-Holders oder eines Editable-Value-Holders angegeben werden. Listing**Kompositkomponente simpleInputim Einsatz**zeigt ein Beispiel für den Einsatz der Komponente`simpleInput`mit einem Action-Listener, einem Validator und einem Value-Change-Listener.

```
<mc:simpleInput action="#{customerBean.save}"
    value="#{customerBean.longValue}" submitLabel="Save">
  <f:actionListener for="submit"
      binding="#{customerBean.saveListener}"/>
  <f:validateLongRange for="input" minimum="10"/>
  <f:valueChangeListener for="input"
      binding="#{customerBean.valueChangeListener}"/>
</mc:simpleInput>
```

Aktiviert ein Benutzer im Browser die Schaltfläche, wird in der Process-Validations-Phase zuerst der Validator aufgerufen. Wenn der Wert gültig ist und sich geändert hat, kommt anschließend der Value-Change-Listener zum Zug. In der Invoke-Application-Phase wird dann zuerst der Listener mit`for="submit"`und dann die in`action`angegebene Action-Methode ausgeführt.

6.1.3 Der Bereich cc:implementation

Der Bereich`cc:implementation`enthält alle JSF-Tags, HTML-Elemente und anderweitigen Inhalte, aus denen die Kompositkomponente aufgebaut ist.
Im Implementierungsteil gibt es mehrere Möglichkeiten, Inhalte einzufügen, die ein Benutzer der Kompositkomponente innerhalb der Komponenten-Tags angegeben hat. Folgende Tags der Composite-

Tag-Bibliothek sind dafür relevant:

- `cc:insertChildren`
  Dieses Tag veranlasst JSF dazu, beim Aufbau des Komponentenbaums den Inhalt des Tags aus der Seitendeklaration des Benutzers zu übernehmen.
- `cc:renderFacet`
  Dieses Tag veranlasst JSF dazu, den Inhalt des Facets mit dem Namen`name`in den Komponentenbaum einzufügen. Das Facet muss ein Kind der Kompositkomponente sein.
- `cc:insertFacet`
  Dieses Tag veranlasst JSF dazu, den Inhalt des Facets mit dem Namen`name`als Facet zu einer anderen Komponente hinzuzufügen. Das Facet muss ein Kind der Kompositkomponente sein.

Eine wichtige Rolle spielt im Implementierungsteil das implizite Objekt`cc`, mit dem in EL-Ausdrücken die aktuelle Kompositkomponente referenziert werden kann. Die wichtigsten Eigenschaften dieses Objekts sind`cc.attrs`zum Zugriff auf die Attribute der Kompositkomponente und`cc.facets`zum Zugriff auf Facets. Da beide Eigenschaften vom Typ Map sind, kann direkt mit der Punktnotation auf einzelne Elemente zugegriffen werden. Das implizite Objekt`cc`referenziert die Wurzelkomponente der Kompositkomponente und hat daher den Typ`UIComponent`. Dadurch ist es natürlich möglich, auch andere Eigenschaften der Komponente wie`cc.clientId`zu verwenden.
Auf den praktischen Aspekt des Implementierungsteils werden wir an dieser Stelle nicht weiter eingehen. Was aber nicht heißen soll, dass wir ihn vernachlässigen. Wir möchten Sie für weiterführende Beispiele nur auf die nächsten Abschnitte verweisen, in denen wir einige Kompositkomponenten präsentieren.

6.1.4 Ressourcen in Kompositkomponenten

Bevor wir uns im nächsten Abschnitt tatsächlich auf die konkreten Beispiele stürzen, werfen wir noch einen Blick auf den Einsatz von JSF-Ressourcen in Kompositkomponenten. Nachdem wir uns mit der Komponente bereits in einer Bibliothek befinden, ist es ein Leichtes, dort zusätzliche Bilder, Stylesheets oder Skripte unterzubringen.
Die Ressourcen werden wie in Abschnitt**Sektion: Ressourcen im Einsatz**beschrieben mit den Tags`h:graphicImage`,`h:outputScript`und`h:outputStylesheet`in die Komponente eingebunden. Der Vorteil dieser Lösung ist, dass sich Benutzer der Komponente keine Gedanken darüber machen müssen. Wenn sie`h:head`und`h:body`verwenden, werden die referenzierten Ressourcen automatisch in die Ansicht aufgenommen. Listing**Ressourcen in einer Kompositkomponente**zeigt ein Beispiel mit dem Skript`script.js`und dem Stylesheet`style.css`, die beide zusätzlich zur Kompositkomponente in der Bibliothek`mygourmet`liegen.

```
<cc:implementation>
  <h:outputScript library="mygourmet"
      name="script.js" target="head"/>
  <h:outputStylesheet name="style.css" library="mygourmet"/>
</cc:implementation>
```

Ab JSF 2.1 gibt es die Möglichkeit, mit dem Namen`this`die Bibliothek der aktuellen Kompositkomponente zu referenzieren. Das funktioniert allerdings nur, wenn die Ressource über einen EL-Ausdruck im Attribut`value`referenziert wir (siehe Abschnitt**Sektion: Ressourcen im Einsatz**). Damit lässt es sich vermeiden, den Namen der Bibliothek direkt anzugeben und die Komponente funktioniert auch mit geändertem Bibliotheksnamen ohne Probleme. Listing**Bibliotheksname this**zeigt das Beispiel aus Listing**Ressourcen in einer Kompositkomponente**mit der Bibliothek`this`.

```
<cc:implementation>
  <h:outputScript target="head"
      value="#{resource['this:script.js']}"/>
  <h:outputStylesheet value="#{resource['this:style.css']}"/>
</cc:implementation>
```

Die Referenzierung von Ressourcen mit dem Bibliotheksname`this`funktioniert nur innerhalb von Kompositkomponenten.

6.1.5 Die Komponente mc:panelBox

Zu Beginn dieses Abschnitts haben wir bereits eine einfache Version der Kompositkomponente`panelBox`gezeigt, die wir noch um einige Aspekte erweitern werden. Listing**Finale Version der Kompositkomponente panelBox**zeigt die komplette Komponente.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://xmlns.jcp.org/jsf/html"
        xmlns:cc="http://xmlns.jcp.org/jsf/composite"
        xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
        xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
<cc:interface>
  <cc:attribute name="styleClass" default="box"/>
  <cc:attribute name="style" required="false"/>
  <cc:attribute name="headerClass" default="box-header"/>
  <cc:facet name="header" required="false"/>
</cc:interface>
<cc:implementation>
  <h:outputStylesheet library="mygourmet"
       name="components.css"/>
  <div class="#{cc.attrs.styleClass}"
      style="#{cc.attrs.style}">
    <c:if test="#{!empty cc.facets.header}">
      <p class="#{cc.attrs.headerClass}">
        <cc:renderFacet name="header"/>
      </p>
    </c:if>
  <cc:insertChildren/>
</div>
</cc:implementation>
</ui:composition>
```

In der einfachen Variante der Komponente war der Header noch als Attribut implementiert. Aus Gründen der Flexibilität ist der Header jetzt ein Facet der Komponente. Damit ist es auch möglich, komplexere Inhalte in den Header zu verfrachten. Die Definition des Facets erfolgt im Interface-Bereich der Komponente mit dem Tag`cc:facet`. In der gerenderten Ausgabe soll der Inhalt des Facets in einem Absatz dargestellt werden - allerdings nur, wenn der Benutzer das Facet angegeben hat. Sehen wir uns dieses Problem Schritt für Schritt an. Das Einfügen des Facets erfolgt mit`cc:renderFacet`und macht keine Probleme, wenn der Benutzer nichts angegeben hat. Problematisch ist erst das umschließende`p`-Element, das natürlich nicht in der Ausgabe auftauchen soll, wenn es kein Facet gibt. Die Lösung ist einfach. Mit einem`mc:if`-Tag wird mit dem Ausdruck`#{!empty cc.facets.header}`überprüft, ob das

Facet`header`angegeben wurde. Ist das nicht der Fall, wird der Block innerhalb`c:if`nicht in den Komponentenbaum eingefügt.
Der zweite interessante Aspekt dieser Kompositkomponente ist das Styling mit CSS. Alle Komponenten in der Bibliothek`mygourmet`sollen ein einheitliches Styling erhalten, das allerdings von geändert werden kann. In der Komponente definieren wir mit dem Tag`h:outputStylesheet`eine Abhängigkeit auf das Stylesheet`components.css`. Fürs Styling kommen die Attribute`styleClass`und`headerClass`mit Defaultwerten zum Einsatz. So ist gewährleistet, dass die internen CSS-Klassen bei Bedarf überschrieben werden können.
Die Deklaration der beiden`panelBox`-Komponenten in der Seitenleiste von*MyGourmet 13*ist in Listing**Kompositkomponente panelBox im Einsatz**zu sehen. Abbildung**Gerenderte Ausgabe von panelBox in MyGourmet 13**zeigt die gerenderte Ausgabe mit Default-Styling.

```
<mc:panelBox id="menu">
  <f:facet name="header">
    <h:outputText value="#{msgs.menu_title}"/>
  </f:facet>
  <h:panelGrid columns="1">
    <h:link outcome="providerList"
        value="#{msgs.menu_provider_list}"/>
    <h:link outcome="showCustomer"
        value="#{msgs.menu_show_customer}"/>
  </h:panelGrid>
</mc:panelBox>
<mc:panelBox id="news">
  <f:facet name="header">
    <h:outputText value="#{msgs.news_title}"/>
  </f:facet>
  <p>MyGourmet - jetzt mit Facelets und Templating</p>
</mc:panelBox>
```

Abbildung:Gerenderte Ausgabe von panelBox in MyGourmet 13

6.1.6 Die Komponente mc:dataTable

Die Komponente`mc:dataTable`zeigt, wie Kompositkomponenten die tägliche Arbeit mit JSF erleichtern können.`mc:dataTable`bietet keine neue Funktionalität, sondern stellt eine Art Wrapper für`h:dataTable`dar. Damit ist es möglich, die Standardkomponente mit einem Default-Style zu erweitern. Listing**Kompositkomponente dataTable**zeigt die Deklaration der Komponente.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:cc="http://xmlns.jcp.org/jsf/composite"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:c="http://xmlns.jcp.org/jsp/jstl/core">
<cc:interface>
  <cc:attribute name="var"/>
```

```
    <cc:attribute name="value"/>
    <cc:facet name="header"/>
    <cc:facet name="footer"/>
  </cc:interface>
  <cc:implementation>
    <h:outputStylesheet library="mygourmet"
        name="components.css"/>
    <h:dataTable id="table" value="#{cc.attrs.value}"
        styleClass="mygourmet-table" headerClass=
        "mygourmet-table-header" rowClasses=
        "mygourmet-table-rownobg, mygourmet-table-rowbg"
        columnClasses="mygourmet-table-cell">
      <c:set target="#{component}" property="var"
          value="#{cc.attrs.var}"/>
      <cc:insertFacet name="header"/>
      <cc:insertChildren/>
      <cc:insertFacet name="footer"/>
    </h:dataTable>
  </cc:implementation>
  </ui:composition>
```

Damit die Tabelle auch Daten anzeigt, brauchen wir die Attribute`var`und`value`in der Schnittstelle der Kompositkomponente, die dann im Implementierungsteil an`h:dataTable`weitergegeben werden. Hier ergibt sich allerdings ein Problem. Das Attribut`var`von`h:dataTable`darf nicht über eine Value-Expression gesetzt werden. Deshalb müssen wir hier einen kleinen Trick anwenden. Facelets bietet mit der JSTL-Funktion`c:set`ein Tag, um Eigenschaften von Beans zu setzen. Ab JSF 2.0 steht die aktuelle Komponente direkt mit dem impliziten Objekt`component`zur Verfügung. Mit`c:set`als direktes Kind von`h:dataTable`ist es somit möglich, das Attribut`var`doch zu setzen.
Die CSS-Klassen werden bei`mc:dataTable`direkt gesetzt, könnten aber wie bei der Komponente`panelBox`auch mit Attributen, die über Defaultwerte verfügen, realisiert werden. Das Stylesheet`components.css`wird genau wie zuvor als Ressource eingebunden.
Der Implementierungsteil zeigt deutlich den Unterschied zwischen`cc:renderFacet`und`cc:insertFacet`. Im aktuellen Beispiel kommt`cc:insertFacet`zum Einsatz, um die Facets`header`und`footer`an`h:dataTable`weiterzureichen. Sie sollen ja nicht von der Kompositkomponente, sondern von`h:dataTable`gerendert werden. Zu guter Letzt bleibt noch das Tag`compo-site:insertChildren`übrig, mit dem die Kindelemente des Tags in der aufrufenden Deklaration an`h:dataTable`werden.`mc:dataTable`wird dadurch wie`h:dataTable`verwendet, in dem der Inhalt mit`h:column`-Tags deklariert wird.
Abbildung**Gerenderte Ausgabe von dataTable**zeigt die gerenderte Ausgabe von`mc:dataTable`aus der Ansicht`providerList.xhtml`.

Name	PLZ	Kategorien
Pizzeria Venezia	1010	Italienisch
Restaurant Mykonos	1040	Griechisch
Zur lustigen Wirtin	1010	Italienisch, Hausmannskost

Abbildung:Gerenderte Ausgabe von dataTable

6.1.7 Die Komponente mc:collapsiblePanel

Die Kompositkomponente`mc:collapsiblePanel`rendert einen Bereich der Seite, der über eine Schaltfläche ein- und ausgeblendet werden kann. Diese Schaltfläche wird als Icon gerendert, das sich je

nach Einklappzustand des Panels ändert. Direkt neben dem Icon wird der Inhalt des optionalen Facets`header`dargestellt. Listing**Kompositkomponente collapsiblePanel**zeigt die Deklaration der Komponente.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:cc="http://xmlns.jcp.org/jsf/composite"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<cc:interface>
  <cc:attribute name="model" required="true">
    <cc:attribute name="collapsed" required="true"/>
    <cc:attribute name="toggle" required="true"
        method-signature="java.lang.String f()"/>
  </cc:attribute>
  <cc:actionSource name="toggle"/>
  <cc:facet name="header"/>
</cc:interface>
<cc:implementation>
  <h:panelGroup layout="block"
      styleClass="collapsiblePanel-header">
    <h:commandButton id="toggle"
        action="#{cc.attrs.model.toggle}"
        styleClass="collapsiblePanel-img"
        image="#{resource[cc.attrs.model.collapsed
          ? 'mygourmet:toggle-plus.png'
          : 'mygourmet:toggle-minus.png']}"/>
    <cc:renderFacet name="header"/>
  </h:panelGroup>
  <h:panelGroup layout="block"
      rendered="#{!cc.attrs.model.collapsed}">
    <cc:insertChildren/>
  </h:panelGroup>
</cc:implementation>
</ui:composition>
```

Das Ein- und Ausklappen des Inhalts der Komponente erfolgt mithilfe der Attribute`collapsed`, das den Einklappzustand steuert, und`toggle`, das die Action-Methode zum Umschalten des Einklappzustands aufnimmt. Die beiden Attribute sind in das Attribut`model`eingebettet, wodurch Benutzer der Komponente nur eine Bean übergeben müssen, die über die Eigenschaft`collapsed`und eine Methode mit der in`toggle`definierten Signatur verfügt.
Der Inhalt des Panels ist in eine`h:panelGroup`-Komponente eingebettet, mit deren`rendered`-Attribut das Ein- und Ausblenden realisiert wird. Der Wert dieses Attributs wird dazu mit dem Ausdruck`#{!cc.attrs.model.collapsed}`vom übergebenen Zustand abhängig gemacht. Beachten Sie hier bitte auch den Zugriff auf das geschachtelte Attribut der Kompositkomponente.
Die Schaltfläche zum Umschalten des Einklappzustands wird in Form von Bildern gerendert. Wie schon das Stylesheet aus den letzten Beispielen liegen die beiden Bilder auch direkt in der Ressourcenbibliothek. Sie werden im Attribut`image`des`h:commandButton`-Tags über das implizite Objekt`resource`eingebunden. Die Schaltfläche selbst ist unter dem Namen`toggle`als Action-Source nach außen verfügbar.
Listing**Kompositkomponente collapsiblePanelim Einsatz**zeigt ein Einsatzszenario der Komponente und in Abbildung**Gerenderte Ausgabe von collapsiblePanel**finden Sie die gerenderte Ausgabe beider Einklappzustände.

```
<mc:collapsiblePanel model="#{customerBean}">
  <f:facet name="header"><h3>Information</h3></f:facet>
  Diese Information ist klappbar.<br/>
  Diese Information ist klappbar.
```

```
</mc:collapsiblePanel>
```

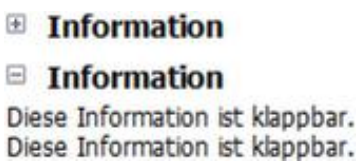


Abbildung:Gerenderte Ausgabe von collapsiblePanel

Die hier gezeigte Kompositkomponente`collapsiblePanel`funktioniert so weit einwandfrei, hat aber noch einen entscheidenden Schönheitsfehler. Die Logik zum Ein- und Ausblenden der Kindkomponenten muss vom Benutzer der Komponente über das Attribut`model`bereitgestellt werden. Das entspricht nicht unserer ursprünglichen Definition von Komponenten als eigenständige und wiederverwendbare Bausteine. Aus diesem Grund werden wir in Abschnitt**[Sektion: Kompositkomponenten und klassische Komponenten kombinieren]**die Komponente um diese Funktionalität erweitern. Dazu zeigen wir Ihnen aber vorher noch in Abschnitt**[Sektion: Klassische Komponenten]**das Erstellen von klassischen Komponenten.

6.1.8 Die Komponente mc:inputSpinner

Die Kompositkomponente`mc:inputSpinner`rendert eine Eingabekomponente mit zwei Buttons, die über JavaScript den Zahlenwert des Eingabefelds erhöhen oder reduzieren. Die Schnittstelle der Komponente nach außen ist sehr übersichtlich. Sie umfasst die Attribute`value`für die Eingabekomponente und`inc`, das den Betrag definiert, der addiert beziehungsweise subtrahiert wird. Zusätzlich ist die Eingabekomponente unter dem Name`input`verfügbar. Listing**Kompositkomponente inputSpinner**zeigt die Deklaration.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://xmlns.jcp.org/jsf/html"
  xmlns:cc="http://xmlns.jcp.org/jsf/composite"
  xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<cc:interface>
  <cc:attribute name="value" required="true"
      type="java.lang.Integer"/>
  <cc:attribute name="inc" default="1"/>
  <cc:editableValueHolder name="input"/>
</cc:interface>
<cc:implementation>
  <h:outputStylesheet library="mygourmet"
      name="components.css"/>
  <h:outputScript library="mygourmet"
      name="inputSpinner.js" target="head"/>
  <h:panelGroup>
    <h:inputText id="input" value="#{cc.attrs.value}"
        styleClass="inputSpinner-input"/>
    <h:panelGroup id="buttons"
        styleClass="inputSpinner-buttons">
      <h:graphicImage styleClass="inputSpinner-button"
          name="spin-up.png" library="mygourmet"
          onclick="return changeNumber(
          '#{cc.clientId}:input', #{cc.attrs.inc});"/>
      <h:graphicImage styleClass="inputSpinner-button"
          name="spin-down.png" library="mygourmet"
          onclick="return changeNumber(
```

```
            '#{cc.clientId}:input', #{-cc.attrs.inc});"/>
      </h:panelGroup>
    </h:panelGroup>
  </cc:implementation>
  </ui:composition>
```

Der Implementierungsteil umfasst das Einbinden der benötigten Ressourcen, die Eingabekomponente und einen `:panelGroup` mit zwei Bildern, die über JavaScript als Schaltfläche zum Ändern des Werts fungieren. Die Komponente lädt also insgesamt vier Ressourcen: Zum einen das bereits bekannte Stylesheet `components.css`, dann das Skript `inputSpinner.js` zum Ändern des Werts und die beiden Bilder `spin-up.png` und `spin-down.png`.
Das Erhöhen und Reduzieren des Werts der Eingabekomponente erfolgt in der Funktion `changeNumber`, die als Parameter die Client-ID der Eingabekomponente und den Wert zum Addieren bekommt. Durch das Übergeben der Client-ID ist es ohne Probleme möglich, mehrere `input-Spinner`-Komponenten in einer Ansicht zu verwenden. Listing **JavaScript für inputSpinner** zeigt die JavaScript-Funktion aus der Ressource `inputSpinner.js`.

```
function changeNumber(clientId, increment) {
  var inc = Number(increment);
  if (isNaN(inc) || inc == 0 ) inc = 1;
  var input = document.getElementById(clientId);
  var val = Number(input.value);
  if (isNaN(val)) val = 0;
  input.value = val + inc;
  return false;
}
```

Die Funktion wird über das Attribut `onclick` der beiden Bilder ausgeführt. Interessant ist dort das Berechnen der Client-ID der Eingabekomponente. Da die Kompositkomponente selbst ein Naming-Container ist, muss auch ihre ID berücksichtigt werden. Diese bekommen wir mit dem Ausdruck `cc.clientId` und durch das Anhängen von `:input` ergibt sich die Client-ID der Eingabekomponente.
Die Komponente wird wie jede andere Eingabekomponente eingesetzt. In Abbildung **Gerenderte Ausgabe von inputSpinner** sehen Sie die gerenderte Ausgabe.

Abbildung:Gerenderte Ausgabe von inputSpinner

6.1.9 Fallstricke in der Praxis

Dieser Abschnitt zeigt eine Reihe von Fallstricken, die beim Einsatz von Kompositkomponenten bedacht werden müssen.

6.1.9.1 Kompositkomponenten im Komponentenbaum

Beim Einsatz von Kompositkomponenten wird wie in Abschnitt **[Sektion: Eine erste Kompositkomponente]** beschrieben immer eine Wurzelkomponente in den Komponentenbaum eingefügt. In den meisten Fällen ist dieses Verhalten auch erwünscht, um zum Beispiel doppelte IDs bei mehrfachem Einsatz der Komponente zu vermeiden. In wenigen Spezialfällen kann diese Wurzelkomponente allerdings

unerwartete Nebeneffekte haben.
Sehen wir uns dazu im folgenden Beispiel die Kompositkomponente`inputField`an, die eine`h:inputText`-Komponente mit einer zugeordneten`h:outputLabel`-Komponente zusammenfasst:

```
<cc:interface>
  <cc:attribute name="value"/>
</cc:interface>
<cc:implementation>
  <h:outputLabel for="input" value="Input:"/>
  <h:inputText id="input" value="#{cc.attrs.value}"/>
</cc:implementation>
```

Im folgenden Beispiel werden zwei der oben gezeigten`inputField`-Komponenten in einer`h:panelGrid`-Komponente angeordnet:

```
<h:panelGrid columns="2">
  <mc:inputField value="#{testBean.value1}"/>
  <mc:inputField value="#{testBean.value2}"/>
</h:panelGrid>
```

Die vermutlich erwartete Ausgabe an dieser Stelle wäre, dass eine Tabelle mit zwei Zeilen gerendert wird, die in der ersten Spalte die Labels und in der zweiten Spalte die Eingabefelder beinhaltet. Das tatsächliche Ergebnis ist allerdings anders: JSF rendert eine Tabelle mit nur einer Zeile. Dieses Verhalten ist aus Sicht von JSF auch korrekt. Das Panel-Grid "sieht" ja nur die Wurzelkomponente der Kompositkomponente und stellt den kompletten Inhalt in einer Zelle dar.
Momentan (einschließlich JSF 2.2) gibt es leider keine Möglichkeit, dieses Verhalten zu umgehen.

6.1.9.2 Verwendung mehrerer Action-Attribute

In Abschnitt**[Sektion: Der Bereich cc:interface]**haben wir gezeigt, wie die Attribute`action`,`actionListener`,`validator`und`valueChangeListener`über das Attribut`targets`mit internen Komponenten verknüpft werden können. Die spezielle Behandlung dieser Attribute funktioniert jedoch nur, wenn das Attribut der Kompositkomponente genau einen dieser Namen hat. Da der Name des Attributs aber eindeutig sein muss, ist es mit JSF 2.0 nicht möglich, mehr als ein "echtes"`action`-Attribut zu definieren.
Spätestens dann, wenn wir zu einer Kompositkomponente eine zweite Schaltfläche hinzufügen wollen, brauchen wir aber ein zweites`action`-Attribut. Als Notlösung kann in JSF 2.0 in diesem Fall ein Attribut mit einer entsprechenden Methodensignatur zum Einsatz kommen wie Abbildung**Mehrere Action-Attribute mit JSF 2.0**zeigt.

```
<cc:interface>
  <cc:attribute name="action" targets="submit"/>
  <cc:attribute name="cancelAction"
      method-signature="java.lang.String action()"/>
</cc:interface>
<cc:implementation>
  <h:commandButton id="submit"/>
  <h:commandButton id="cancel"
      action="#{cc.attrs.cancelAction}"/>
</cc:implementation>
```

Ab JSF 2.1 gibt es eine elegantere Lösung für dieses Problem. Mit dem bereits bekannten Attribut`targets`und dem neuen Attribut`targetAttributeName`kann`cc:attribute`mit einem bestimmten Attribut einer internen Komponente verknüpft werden. Abbildung**Mehrere Action-Attribute mit**

JSF 2.1zeigt diesen Ansatz für das obige Beispiel.

```
<cc:interface>
  <cc:attribute name="submitAction" targets="save"
      targetAttributeName="action"/>
  <cc:attribute name="cancelAction" targets="cancel"
      targetAttributeName="action"/>
</cc:interface>
<cc:implementation>
  <h:commandButton id="submit"/>
  <h:commandButton id="cancel"/>
</cc:implementation>
```

Der Wert des Attributs`submitAction`wird in diesem Beispiel mit dem Attribut`action`der internen Komponente mit der ID`submit`verknüpft. Analog wird der Wert des Attributs`cancelAction`an das Attribut`action`der internen Komponente mit der ID`cancel`weitergereicht.`targetAttributeName`enthält also immer dann den Namen des "Zielattributs", wenn sich dieser vom Namen in`name`unterscheidet.

6.1.9.3 Auflösung des Typs von Attributen

Mit JSF 2.0 und 2.1 kann es zu Problemen mit Attributen von Kompositkomponenten kommen, wenn beim Einsatz der Komponente`null`als Attributwert übergeben wird. JSF kann in diesem Fall den Typ des übergebenen Werts nicht auflösen, was in bestimmten Situationen zu fehlerhaftem Verhalten führt.
Dieses Fehlverhalten lässt sich anhand eines Beispiels sehr einfach demonstrieren. Die Kompositkomponente`inputTest`hat ein Attribut mit dem Namen`value`, das im Implementierungsteil in einer Eingabekomponente benutzt wird:

```
<cc:interface>
  <cc:attribute name="value"/>
</cc:interface>
<cc:implementation>
  <h:inputText id="input" value="#{cc.attrs.value}"/>
</cc:implementation>
```

Der Fehler tritt auf, wenn beim Einsatz der Komponente eine Value-Expression verwendet wird, die zu`null`evaluiert (vorausgesetzt die Eigenschaft`longValue`hat den Typ`java.lang.Long`und nicht`long`):

```
<mc:inputTest value="#{testController.longValue}"/>
```

JSF kann in Version 2.0 und 2.1 den Typ in diesem Fall nicht auflösen. Daher wird in der Eingabekomponente auch kein entsprechender Konverter gesetzt - was sonst automatisch basierend auf dem Typ`java.lang.Long`gemacht werden würde. Der fehlende Konverter kann zu unerwartetem Verhalten im Lebenszyklus führen. Bei einer ungültigen Benutzereingabe wird dann zum Beispiel statt der Anzeige einer Fehlermeldung eine`ClassCastException`geworfen.
Mit JSF 2.2 tritt dieses Problem nicht mehr auf, da der Algorithmus zur Auflösung des Typs von Attributen geändert wurde (in*Apache MyFaces*wird dieser Algorithmus bereits ab den Versionen 2.0.10 und 2.1.4 eingesetzt). Für frühere JSF-Versionen können Sie dieses Problem verhindern, indem Sie in`h:inputText`explizit einen Konverter verwenden (falls der Typ eindeutig ist) oder indem Sie auf den Wert`null`für diese Attribute verzichten.

6.2 Klassische

Komponenten

Nach der Einführung in das Thema Kompositkomponenten zeigen wir Ihnen in diesem Abschnitt, wie Sie mit JSF eine klassische Komponente für den Einsatz mit Facelets erstellen. Auf den ersten Blick kann dieser Vorgang etwas kompliziert wirken, da er eine Reihe von Schritten umfasst - vor allem auch wegen der vielen Webtechnologien, die dabei verwendet werden.
Die Erstellung einer Komponente besteht aus folgenden Schritten:

- Komponentenfamilie, Komponententyp und Renderertyp definieren
- Komponentenklasse schreiben
- Rendererklasse schreiben
- Komponentenklasse und Rendererklasse registrieren
- Tag-Definition schreiben und in Bibliothek aufnehmen
- Tag-Handler-Klasse schreiben
- Bibliothek in die Seite einbinden

Jeden einzelnen dieser Punkte werden wir uns jetzt anhand eines kleinen Beispiels genauer ansehen. Zu Demonstrationszwecken erstellen wir die Kompositkomponente`inputSpinner`nochmals als klassische JSF-Komponente.

6.2.1 Vorarbeiten: Komponentenfamilie, Komponententyp und Renderertyp definieren

Bevor wir mit dem Schreiben der Komponente beginnen, sollten wir uns darüber klar werden, ob und wie die Komponente von einer anderen Komponente ableitbar ist oder ob wir vollständig "von der grünen Wiese" weg beginnen müssen. Üblicherweise ist es sinnvoll, von einer bestehenden Basiskomponente abzuleiten.

6.2.1.1 Die Wahl der Basisklasse

Kandidaten für die Basisklasse sind von`UIComponent`abgeleitete Klassen, im Standard also die folgenden Komponentenklassen:

- `UIOutput`:
  Komponente, die einen Wert darstellt (und keine Eingabe erlaubt).
- `UIInput`:
  Komponente, die die Eingabe eines Werts ermöglicht.
- `UISelectOne`:
  Komponente, die die Auswahl genau eines Werts aus einer Reihe von Werten ermöglicht.
- `UISelectMany`:
  Komponente, die die Auswahl mehrerer Werte aus einer Reihe von Werten ermöglicht.
- `UICommand`:
  Komponente, die eine Aktion ausführt.

- `UIPanel`:
  Komponente, die als Behälter für eine oder mehrere andere Komponenten dient.
- `UIMessage`:
  Komponente, die zur Anzeige einer Nachricht dient.

Häufig wird von einer der drei Komponenten`UICommand`,`UIInput`oder`UIPanel`abgeleitet, weil diese drei Klassen bereits sehr viel Logik für das Arbeiten mit häufig wiederkehrenden Aufgaben bereitstellen.
Wann ist das Ableiten von`UIInput`sinnvoll? Das ist dann angebracht, wenn die Komponente genau einen Wert aus dem Modell präsentieren soll. Dinge wie Konvertierung, Validierung, Schreiben und Lesen des Werts aus und in das Modell werden von dieser Klasse bereits erledigt. Ist das Verhalten der Komponente noch spezieller, kann man eventuell von der Klasse`HtmlInputText`ableiten - diese Klasse und der zugehörige Renderer erledigen auch das Decodieren des Werts aus der HTTP-Anfrage, das Setzen des Werts als Submitted-Value und das Schreiben der Komponentenansicht in die HTTP-Antwort.
Die Klasse`UIInput`bietet sich ebenfalls an, wenn eine vom Benutzer ausgelöste Veränderung eines Komponentenattributs gespeichert werden soll. Damit kann zum Beispiel eine einklappbare Panel-Komponente realisiert werden, die ihren Einklappzustand im Attribut`value`ablegt. Dadurch kann der Zustand auch an eine Bean-Eigenschaft gebunden werden.
Die`UICommand`-Klasse wird man als Basis für die eigene Komponente verwenden, wenn diese "Aktionen" auslöst. Die Behandlung von Action-Methoden, Action-Listenern und das Auslösen von Ereignissen werden durch diese Basisklasse bereits erledigt.
Die Wahl der Basisklasse für den Input-Spinner fällt eindeutig aus. Nachdem es sich um ein "getuntes" Eingabefeld handelt, drängt sich`UIInput`auf.

6.2.1.2 Komponententyp, Komponentenfamilie und Renderertyp

Jede Komponente ist im System eindeutig über ihren Komponententyp identifiziert. Es darf keine andere Komponente mit demselben Komponententyp existieren. Üblicherweise wird der Komponententyp als statische Konstante definiert. Unsere Input-Spinner-Komponente, deren Komponentenklasse`InputSpinner`wir im nächsten Abschnitt erstellen, hat folgenden Komponententyp:

```
public static final String COMPONENT_TYPE =
    "at.irian.InputSpinner";
```

Beim Registrieren der Komponente im System wird ihr Komponententyp als Bezeichner verwendet. Dieser kommt dann zum Beispiel bei der Definition des Tags der Komponente in der Tag-Bibliothek zum Einsatz. Darüber hinaus können wir den Komponententyp übergeben, wenn wir eine Komponente erzeugen wollen und dazu nicht direkt den Konstruktor, sondern die Factory-Methode`createComponent()`der`Application`-Klasse aufrufen. Tatsächlich sollte man immer über diesen Weg gehen, damit zentral die Implementierung einer Komponente ausgetauscht werden kann. Hier das Beispiel dafür:

```
FacesContext.getCurrentInstance().getApplication()
    .createComponent(InputSpinner.COMPONENT_TYPE);
```

Der nächste Schritt ist die Definiton der Komponentenfamilie - hier kann entweder eine bestehende Familie verwendet oder eine neue Familie definiert werden. Die Familie einer Komponente wird mit einem Aufruf von`UIComponent.getFamily()`bestimmt. Wenn Sie eine eigene Familie für Ihre Komponente erstellen wollen, müssen Sie diese Methode überschreiben. In einer Komponentenfamilie können mehrere Komponenten enthalten sein. Nachdem es sich bei unserer Beispielkomponente um ein Eingabefeld handelt, belassen wir es bei der von`UIInput`geerbten Familie`javax.faces.Input`.
Die Komponentenfamilie wird gemeinsam mit dem Renderertyp verwendet, um einen Renderer auszuwählen. Würde hier der Komponententyp benutzt werden, könnte ein Renderer immer nur eine Art von Komponente rendern - durch die Definition der Komponentenfamilie kann für eine ganze Gruppe von Komponenten derselbe Renderer zum Einsatz kommen. Der Renderertyp wird bei der Definition des Tags gemeinsam mit dem Komponententyp angegeben und beim Erzeugen der Komponente gesetzt - damit wird

der oft im Komponentenkonstruktor gesetzte "Default"-Renderertyp überschrieben.
Im Folgenden sehen Sie den Komponententyp, die Komponentenfamilie und den Renderertyp für ausgewählte Standardkomponenten:

- `javax.faces.component.html.HtmlCommandLink`:
  Die Komponente hinter dem Tag `h:commandLink`.
  Komponententyp: `javax.faces.HtmlCommandLink`
  Komponentenfamilie: `javax.faces.Command`
  Renderertyp: `javax.faces.Link`
- `javax.faces.component.html.HtmlCommandButton`:
  Die Komponente hinter dem Tag `h:commandButton`.
  Komponententyp: `javax.faces.HtmlCommandButton`
  Komponentenfamilie: `javax.faces.Command`
  Renderertyp: `javax.faces.Button`
- `javax.faces.component.html.HtmlInputText`:
  Die Komponente hinter dem Tag `h:inputText`.
  Komponententyp: `javax.faces.HtmlInputText`
  Komponentenfamilie: `javax.faces.Input`
  Renderertyp: `javax.faces.Text`

Unsere Input-Spinner-Komponente erhält den eigenen Renderertyp `at.irian.InputSpinner`.

6.2.2 Komponentenklasse schreiben

Die Komponentenklasse ist das "Herzstück" der Komponentenarchitektur von JSF - die Instanz der Komponentenklasse wird im JSF-Komponentenbaum gespeichert und beinhaltet alle Daten einer Komponente. Nachdem die Komponentenklasse die eigentlich "treibende Kraft" im Ablauf einer JSF-Anfrage istIn jeder Phase des Lebenszyklus wird der Komponentenbaum durch rekursive Aufrufe der zuständigen Behandlungsmethoden durchwandert.:, kann man alle Ereignisse in JSF von hier aus steuern. Zwei der wichtigsten Aufgaben sind das Auslesen der für die Komponente relevanten Request-Parameter (Decoding) und das Rendern der Komponente (Encoding). Die Komponente kann - wenn das vom Entwickler so vorgesehen ist - diese Aufgaben selbst übernehmen oder an einen Renderer delegieren. Der größte Vorteil eines eigenen Renderers ist die klare Trennung zwischen den in der Komponente enthaltenen Daten und der daraus gerenderten Ausgabe. Die beiden Ansätze schließen sich nicht gegenseitig aus. So wäre es zum Beispiel vorstellbar, in einer ersten Version das Rendering von der Komponente durchführen zu lassen und erst später einen eigenen Renderer zu erstellen.
Wir werden uns in diesem Abschnitt um die Komponentenklasse kümmern und erst in Abschnitt **[Sektion: Rendererklasse schreiben]**einen genaueren Blick auf die Rendererklasse werfen.

6.2.2.1 Komponentenattribute

Eine Komponentenklasse enthält üblicherweise eine Eigenschaft für jedes Attribut des zugehörigen Tags. Es kann aber auch Attribute des Tags geben, die nicht explizit in der Komponente existieren - diese werden dann in einer speziellen Map in der Komponente abgelegt.
Die Zugriffsmethoden auf die Eigenschaften der Komponente müssen gewährleisten, dass die Eigenschaften sowohl direkt als auch über Value-Expressions angegeben werden können. Dabei gilt, dass das direkte Setzen von Attributen stärker "zieht" als das Setzen einer Value-Expression. Beim Lesen des Komponentenattributs muss dann natürlich berücksichtigt werden, ob es direkt gesetzt wurde. Ist das der Fall, wird der Komponentenwert direkt zurückgegeben, ansonsten wird die Value-Expression evaluiert. Klingt kompliziert - ist aber mit JSF 2.0 zum Kinderspiel geworden.
Ab JSF 2.0 werden als Grundlage für das neue Partial-State-Saving Eigenschaften von Komponenten nicht

mehr in privaten Feldern, sondern in einer Map verwaltet. Zu diesem Zweck hat jede Komponente eine Instanz der Klasse`StateHelper`, die den Zustand intern verwaltet. Was es mit dem Zustand auf sich hat, klären wir etwas weiter unten, jetzt interessiert uns erst einmal das Lesen und Schreiben der Eigenschaften.
Listing**Die Klasse InputSpinner**zeigt die Klasse`InputSpinner`unserer Beispielkomponente. Wie Sie sehen, ist der Code sehr überschaubar und umfasst im Grunde genommen nur die Definition der Eigenschaft`inc`. Den Rest, wie etwa das State-Saving oder das Verwalten der anderen Attribute, erledigen die Basisklassen von JSF. Nachdem wir von`UIInput`abgeleitet haben, können wir auch dessen bereits vorhandenen Funktionsumfang benutzen.

```
public class InputSpinner extends UIInput {

  public static final String COMPONENT_TYPE =
      "at.irian.InputSpinner";

  enum PropertyKeys {inc}

  public InputSpinner() {
    setRendererType("at.irian.InputSpinner");
  }
  public int getInc() {
    return (Integer)getStateHelper().eval(
        PropertyKeys.inc, 1);
  }
  public void setInc(int inc) {
    getStateHelper().put(PropertyKeys.inc, inc);
  }
}
```

Mit der Methode`put()`wird der Wert der Eigenschaft direkt mit dem Schlüssel`PropertyKeys.inc`gesetzt. Die Methode`eval()`liefert, falls vorhanden, den direkt gesetzten Wert zurück. Existiert ein solcher nicht, wird anschließend eine eventuell vorhandene Value-Expression evaluiert. In unserem Fall wird, falls auch eine solche nicht existiert, der Defaultwert 1 zurückgeliefert.

6.2.2.2 State-Saving

Neben den Zugriffsmethoden auf die Eigenschaften der Komponente hat die Komponentenklasse noch eine äußerst wichtige Aufgabe: Sie muss ihren Zustand bis zum nächsten Request sichern können. Das State-Saving ist in früheren JSF-Versionen relativ einfach gestrickt und verfolgt den Ansatz, immer den kompletten Zustand des gesamten Komponentenbaums zu speichern und wiederherzustellen. Da diese Vorgehensweise in Bezug auf Performance und Speicherverbrauch nicht optimal ist, wurde in JSF 2.0 mit dem Partial-State-Saving ein neuer Ansatz realisiert.
Wie der Name bereits erahnen lässt, werden dabei nur noch wirklich relevante Teile des Zustands gespeichert. JSF markiert dazu nach dem Aufbau des Komponentenbaums einen initialen Zustand, der ohnehin durch die Seitendeklaration definiert ist. Der Partial-State setzt sich dann aus allen Änderungen am Komponentenbaum nach Erreichen dieses Zustands zusammen. Voraussetzung für die korrekte Initialisierung ist der Aufbau der Ansicht aus der Seitendeklaration vor jedem Wiederherstellen des Zustands.
Für das State-Saving sind die Methoden`saveState()`und`restoreState()`aus dem Interface`StateHolder`zuständig. Vor JSF 2.0 mussten diese beiden Methoden in mühsamer Kleinarbeit für jede Komponentenklasse erstellt werden - ein fehleranfälliger Prozess, dem wir keine Träne nachweinen. Ab JSF 2.0 sind diese Methoden bereits in`UIComponentBase`implementiert und müssen nur noch bei speziellen Anforderungen überschrieben werden.
Wie funktioniert das? Die Grundlage für die Aufzeichnung des Zustands bildet der im letzten Abschnitt vorgestellte`StateHelper`, der den Zustand der Komponente intern verwaltet. Damit ist auch bereits der Grundstein für das Partial-State-Saving gelegt. Die "zentralisierte" Zustandsverwaltung ermöglicht das Festhalten von Änderungen nach Erreichen des initialen Zustands.

Jede Komponente, die Partial-State-Saving einsetzen will, muss das von`StateHolder`abgeleitete und um Methoden zur Markierung des initialen Zustands erweiterte Interface`PartialStateHolder`implementieren. Nachdem auch diese Methoden bereits in der Klasse`UIComponentBase`implementiert und in einigen anderen Basisklassen erweitert werden, müssen Sie sich auch darum keine Gedanken machen.

6.2.2.3 Komposition klassischer Komponenten

Ein interessanter Aspekt der Komponentenentwicklung ist die Komposition von Komponenten zu einem größeren Ganzen: Dazu ist die geeignete Stelle zu finden, in der Kindkomponenten der Elternkomponente hinzugefügt werden können. Im Wesentlichen gibt es hier zwei Möglichkeiten: Die einfachere Variante ist das Hinzufügen von*transienten*Kindkomponenten beim Rendern der Seite.*Transient*bedeutet, dass das Attribut`transient`der Komponente auf`true`gesetzt wurde, und die Komponente daher im*State-Saving*-Prozess verschwindet. Auch wenn diese Art der Komposition funktioniert, ist sie doch nicht in allen Fällen optimal, weil eine Art Komponentenfunktionalität in den Renderer ausgelagert wird.
Mit JSF ab Version 2.0 ist es möglich, Kindkomponenten mithilfe des System-Events`PostAddToViewEvent`zu einer zusammengesetzten Komponente hinzuzufügen. Dieses Ereignis wird ausgelöst, nachdem eine Komponente in den Komponentenbaum eingefügt wurde. Wir werden einen Listener für dieses Ereignis erstellen, in dem wir die zusätzlichen Komponenten hinzufügen.
Listing**Komposition klassischer Komponenten mit System-Events**zeigt einen Ausschnitt einer Komponentenklasse, die über die Annotation`@ListenerFor`als Listener für das System-Event`PostAddToViewEvent`registriert ist. Tritt das Ereignis beim Einfügen der Komponente in die Ansicht auf, wird die im Interface`ComponentSystemEventListener`definierte Methode`processEvent`aufgerufen. Da dieses Interface bereits von der Klasse`UIComponent`implementiert wird, müssen wir nur die entsprechende Methode überschreiben und erweitern.

```
@FacesComponent("at.irian.MyPanel")
@ListenerFor(systemEventClass = PostAddToViewEvent.class)
public class MyPanel extends HtmlPanelGroup {

  public void processEvent(ComponentSystemEvent ev)
  throws AbortProcessingException {
    if (ev instanceof PostAddToViewEvent) {
      addComponents();
    }
    super.processEvent(ev);
  }
  ...
}
```

6.2.3 Rendererklasse schreiben

Der Begriff Renderer steht in JSF für eine Klasse, die einer Komponente (oder einer Komponentenfamilie) zugeordnet ist und die Ansicht für diese Komponente erstellt, aber auch den Wert einer Komponente wieder aus der HTTP-Anfrage ausliest und in die Komponenteninstanz überträgt. Jede Rendererklasse muss von der abstrakten Klasse`javax.faces.render.Renderer`abgeleitet werden und die Methoden überschreiben, die nicht die Standardfunktionalität besitzen sollen.
Der Renderer ist letztlich die Klasse, deren Schreiben am meisten Aufwand bedeutet, weil in ihr die ganze Ansichtslogik programmmiert werden muss - durch die Vielfalt der in der Webanwendungsentwicklung verwendeten Technologien wie HTML, JavaScript, CSS und XML ist diese Ansichtslogik bei größeren Komponenten sehr komplex und unübersichtlich. Das ist auch der Grund für die Spezifizierung des JSF-Standards - der "normale" Webentwickler muss sich um die Entwicklung dieser Ansichtslogik nicht mehr

kümmern.
Folgende Aufzählung zeigt die Methoden der Klasse`Renderer`:

- `void decode(FacesContext ctx, UIComponent comp)`
  Liest den Wert der Komponente aus den Request-Parametern.
- `void encodeBegin(FacesContext ctx, UIComponent comp)`
  Wird beim Rendern der Komponente zuerst aufgerufen.
- `void encodeChildren(FacesContext ctx, UIComponent comp)`
  Wird beim Rendern der Komponente nach`encodeBegin()`aufgerufen, wenn`getRendersChildren()`den Wert`true`zurückliefert.
- `void encodeEnd(FacesContext ctx, UIComponent comp)`
  Wird beim Rendern der Komponente zuletzt aufgerufen.
- `boolean getRendersChildren()`
  Liefert den Wert`true`zurück, wenn der Renderer alle Kindkomponenten selbst rendert. Der Defaultwert ist`false`.
- `Object getConvertedValue(FacesContext ctx, UIComponent comp, Object submittedValue)`
  Konvertiert den Submitted-Value von einem String in den für die Komponente benötigten Wert.
- `String convertClientId(FacesContext ctx, String clientId)`
  Konvertiert die Client-ID in eine für den Client gültige Form.

Einige dieser Methoden müssen von (fast) jedem Entwickler einer benutzerdefinierten Komponente überschrieben werden. Je nach Typ der Komponente ist das entweder`encodeBegin()`oder`encodeEnd()`. Diese Methoden werden aufgerufen, wenn das zugehörige Tag in der Seitendeklaration geöffnet respektive geschlossen wird. Hier sollte der zur Komponente gehörige HTML-Code geschrieben werden - wenn es sich überhaupt um einen Renderer für HTML handelt. Prinzipiell sind natürlich auch Renderer für andere Ausgabeformate wie WML, XML, XUL oder SVG denkbar.
Der Renderer für unsere Input-Spinner-Komponente wird in der Klasse`InputSpinnerRenderer`implementiert.

6.2.3.1 Rendern (Encoding)

Listing**Die Methode encodeBegin() des Beispielrenderers**zeigt die Methode`encodeBegin()`der Rendererklasse`InputSpinnerRenderer`. Als Parameter werden der Methode der Faces-Context und die Komponente, für die das Rendern erfolgen soll, übergeben. Das Schreiben der Ansicht erfolgt in den beiden privaten Methoden`encodeInput()`zum Rendern des Eingabefelds und`encodeButtons()`zum Rendern der beiden Spin-Buttons. Momentan interessiert uns nur`encodeInput()`, da dort das`input`-Element über Aufrufe der Klasse`ResponseWriter`geschrieben wird - die wiederum erhält man vom momentanen Faces-Context über einen Aufruf der Methode`getResponseWriter()`.

```
public void encodeBegin(FacesContext ctx,
    UIComponent component) throws IOException {
  InputSpinner spinner = (InputSpinner)component;
  String clientId = spinner.getClientId();
  encodeInput(ctx, spinner, clientId);
  encodeButtons(ctx, spinner, clientId);
}
private void encodeInput(FacesContext ctx,
    InputSpinner spinner, String clientId)
    throws IOException {
  ResponseWriter writer = ctx.getResponseWriter();
```

```
    writer.startElement("input", spinner);
    writer.writeAttribute("id", clientId, null);
    writer.writeAttribute("name", clientId, null);
    Object value = getValue(ctx, spinner);
    if (value != null) {
      writer.writeAttribute("value", value.toString(), null);
    }
    writer.writeAttribute("class", "inputSpinner-input", null);
    writer.endElement("input");
  }
```

In vielen existierenden Komponenten wird üblicherweise die Methode`encodeEnd()` und nicht die Methode`encodeBegin()` überschrieben. Das liegt an historischen Gründen in Verbindung mit JSF 1.1 - dort ist erst in der`encodeEnd()`-Methode sichergestellt, dass alle Kindelemente der gerade geschriebenen JSF-Komponente bereits erstellt und in den Komponentenbaum eingehängt wurden - daher wird üblicherweise die`encodeEnd()`-Methode für Komponenten mit Kindelementen verwendet. In JSF-Versionen ab 1.2 ist das nicht mehr notwendig, weil hier in der Render-Response-Phase der gesamte Baum bereits aufgebaut ist.
Standardmäßig wird der Komponentenbaum in JSF rekursiv durchlaufen und jede Komponente wird durch einen einmaligen Aufruf der Methode`encodeAll()` gerendert. Die Methode`encodeAll()` ruft zuerst die Methode`encodeBegin()` der aktuellen Komponente auf und überprüft dann, ob die Methode`getRendersChildren()` den Wert`true` zurückliefert. Wenn dem so ist, werden nicht die einzelnen Kindkomponenten durchgegangen, sondern die Methode`encodeChildren()` aufgerufen - damit kann also die Komponente selbst ihre Kinder in die Ansicht schreiben! Ansonsten wird die Methode`encodeAll()` auf den einzelnen Kindern aufgerufen und damit rekursiv der Baum einen Schritt tiefer in die Ansicht geschrieben. Für Komponenten, die ihre Kindelemente selbst verwalten und in die HTML-Ansicht schreiben wollen, ist es also wichtig, dass ihr Renderer die Methode`getRendersChildren()` überschreibt und`true` zurückliefert.
`ResponseWriter`:Zurück zum Schreiben der Ausgabe unserer Komponente. Dieser Vorgang funktioniert für HTML (und alle von SGML abgeleiteten Dialekte ähnlich) über das Öffnen, Schließen und Schreiben von Attributen von den zur Komponente gehörenden Tags. Das Öffnen eines Tags ist ein einfacher Aufruf des`ResponseWriter`:

```
  writer.startElement("input", spinner);
```

`startElement()`:Zuerst wird an die Methode`startElement()` die Zeichenkette übergeben, die als Name des Tags verwendet werden soll - in unserem Fall`input`. Als zweites Attribut wird die Komponente selbst übergeben. Sehr wichtig - hier soll auf keinen Fall`null` übergeben werden, sondern immer die zugehörige Komponente. Ist das HTML-Tag einer Komponente nicht eins zu eins zuzuordnen - wenn beispielsweise ein Renderer für eine Komponente mehrere HTML-Tags erzeugt -, sollte die Komponente jedem dieser Tags übergeben werden, das Gleiche gilt auch für eventuelle Kind-Tags. Die Information über die dazugehörende Komponente wird von grafischen Entwicklungsumgebungen ausgewertet, um beim Rendering zur Designzeit beispielsweise alle Tags einer Komponente mit einer speziellen Klasse auszuzeichnen.
`writeAttribute()`:Im nächsten Schritt werden die Attribute des Tags in die HTML-Ansicht geschrieben, dazu dient die Methode`writeAttribute()`. Auch hier wird wieder der Name des Attributs zuerst übergeben:

```
  writer.writeAttribute("id", clientId, "id");
```

Auf den Attributsnamen folgt der zu schreibende Wert und schließlich wieder das dazugehörende Attribut der Komponente. Auch diese Verbindung wird von Entwicklungsumgebungen genutzt und sollte nach Möglichkeit gesetzt werden, wenn ein entsprechendes Komponentenattribut vorhanden ist. Ein Beispiel für ein Attribut, wo das nicht möglich ist, ist das`onclick`-Attribut. Dieses Attribut hat keine Entsprechung in der Komponente, es wird nur in die Map der Attribute eingetragen.

```
writer.writeAttribute(HTML.ONCLICK_ATTR,
    onClick.toString(), null);
```

value-Attribut:Wichtig (und etwas anders als die Behandlung der anderen Attribute) ist die Behandlung des`value`-Attributs. Wenn eine JSF-Anfrage am Server ankommt, wird der Wert einer Komponente decodiert und dieser Wert vorerst in das Feld`submittedValue`geschrieben. Tritt beim Konvertieren oder Validieren einer Komponente des Komponentenbaums ein Fehler auf, wird die weitere Behandlung der Anfrage abgebrochen und direkt in die Render-Response-Phase gesprungen. Statt jetzt den Wert der Komponente auszugeben, den man über den Aufruf von`getValue()`erhält, muss man also beim Rendern zuerst prüfen, ob nur der Submitted-Value gesetzt worden ist. Wenn diese Bedingung zutrifft, darf man nur diesen Submitted-Value schreiben.
Konvertierung:Ist nicht der Submitted-Value ausschlaggebend, sondern tatsächlich ein Wert für die Komponente gesetzt, muss vor dem Rendern des Werts noch ein eventuell angegebener Konverter aufgerufen werden, um den Wert in eine Zeichenkette umzuwandeln.
Listing**Auslesen des Werts einer Komponente beim Rendern**zeigt die Methode`getValue()`der Klasse`InputSpin-nerRenderer`, die nach dem soeben beschriebenen Algorithmus den Wert der Komponente für die Anzeige zurückliefert.

```
private Object getValue(FacesContext ctx,
    InputSpinner spinner) {
  Object submittedValue = spinner.getSubmittedValue();
  if (submittedValue != null) {
    return submittedValue;
  }
  Object value = spinner.getValue();
  Converter converter = getConverter(ctx, spinner);
  if (converter != null) {
    return converter.getAsString(ctx, spinner, value);
  } else if (value != null) {
    return value.toString();
  } else {
    return "";
  }
}
private Converter getConverter(FacesContext ctx,
    UIComponent comp) {
  Converter conv = ((UIInput)comp).getConverter();
  if (conv != null) return conv;
  ValueExpression exp = comp.getValueExpression("value");
  if (exp == null) return null;
  Class valueType = exp.getType(ctx.getELContext());
  if (valueType == null) return null;
  return ctx.getApplication().createConverter(valueType);
}
```

Unabhängigkeit von JSF-Implementierung:Üblicherweise kann man Funktionalitäten von der darunterliegenden JSF-API ohne Probleme übernehmen. Man sollte aber darauf aufpassen, sich nicht von der konkreten JSF-Implementierung, auf der man die Komponente entwickelt, abhängig zu machen - man sollte also tatsächlich nur die Funktionalität der`javax.faces.`*-API verwenden.

6.2.3.2 Decodierung (Decoding)

Weiter im Programm: Genauso, wie die Komponente in die HTML-Seite geschrieben wird, muss der Wert der Komponente bei einem Postback wieder aus der HTTP-Anfrage ausgelesen werden können. Auch diese Aufgabe erledigt der Renderer, und zwar mit der Methode`de-code()`. Listing**Decodieren eines**

Wertszeigt die Methode unseres Beispielrenderers.

```
public void decode(FacesContext ctx, UIComponent component) {
  Map<String, String> params = ctx
      .getExternalContext().getRequestParameterMap();
  String clientId = component.getClientId();
  String value = params.get(clientId);
  ((UIInput)component).setSubmittedValue(value);
}
```

Die Vorgehensweise ist recht einfach: Die Map der Request-Parameter wird nach der Client-ID der Komponente durchsucht und der zurückgelieferte Wert wird als Submitted-Value der Komponente gesetzt. Konvertierung und Validierung:Jetzt geht's weiter im Lebenszyklus der HTTP-Anfrage: Die Komponente muss den Submitted-Value jetzt konvertieren und anschließend validieren. Zum Konvertieren des Werts wird die Methode`getConvertedValue()`des Renderers aufgerufen (siehe Listing**Konvertieren des Werts im Beispielrenderer**).

```
public Object getConvertedValue(FacesContext ctx,
    UIComponent component, Object submittedValue)
    throws ConverterException {
  Converter converter = getConverter(ctx, component);
  if (converter != null ) {
    return converter.getAsObject(
        ctx, component, (String) submittedValue);
  } else {
    return submittedValue;
  }
}
```

6.2.3.3 Rendern von Ressourcen

Das Rendern von Ressourcen wie Bildern, Stylesheets oder Skripten ist ein wichtiger Aspekt bei vielen Komponenten. Ab JSF 2.0 gibt es dafür jetzt endlich auch eine standardisierte Lösung, was das Erstellen von Komponenten erheblich vereinfacht. Das Thema Ressourcen haben wir ja schon in Kapitel**Kapitel: Verwaltung von Ressourcen**ausführlich behandelt. Dort wurde bereits kurz erwähnt, dass Abhängigkeiten zwischen Ressourcen und Komponenten in Form von Annotationen auf der Komponenten- oder Rendererklasse abgebildet werden können.
Sehen wir uns das für unsere Input-Spinner-Komponente etwas genauer an. Aus Gründen der Einfachheit werden wir die Ressourcen der Kompositkomponente`inputSpinner`aus der Bibliothek`mygourmet`mitverwenden. Damit die Komponente richtig dargestellt wird und ordnungsgemäß funktioniert, benötigen wir das Stylesheet`components.css`und das Skript`inputSpinner.js`. Um die beiden Bilder zum Erhöhen und Reduzieren des Werts kümmern wir uns später. Für jede der beiden Ressourcen annotieren wir die Rendererklasse mit`@ResourceDependency`unter Angabe der Bibliothek und des Namens. Das Skript wird zusätzlich mit`target="head"`in den Header der gerenderten Ausgabe verfrachtet. Listing**Ressourcenannotationen auf der Rendererklasse**zeigt die Rendererklasse mit den Annotationen. Mehr ist nicht notwendig, um JSF die Ressourcen automatisch verwalten zu lassen - vorausgesetzt natürlich, Sie verwenden`h:head`und`h:body`. Sie können die Komponente auch mehrfach auf einer Seite einsetzen, JSF wird sie immer nur einmal rendern.

```
@ResourceDependencies({
  @ResourceDependency(library = "mygourmet",
      name = "inputSpinner.js", target = "head"),
  @ResourceDependency(library = "mygourmet",
      name = "components.css")}
```

```
)
public class InputSpinnerRenderer extends Renderer {
...
}
```

Die automatische Ressourcenverwaltung stellt auch sicher, dass Ressourcen in der gerenderten Ausgabe immer genau dort landen, wo sie sollen. Ein Stylesheet hat zum Beispiel in einem HTML-Dokument außerhalb von`head`nichts zu suchen.
Die beiden Bilder`spin-up.png`und`spin-down.png`, die wir ebenfalls aus der Bibliothek`mygourmet`der Kompositkomponente übernehmen, können natürlich nicht einfach über eine Annotation mit der Komponente verbunden werden. Nachdem sie einen Teil der gerenderten Ausgabe der Komponente bilden, müssen sie manuell eingefügt werden. Dazu kommt die Klasse`ResourceHandler`zum Einsatz, mit der JSF intern Ressourcen verwaltet. Der Zugriff auf den für die Anwendung zuständigen Resource-Handler erfolgt über das Applikationsobjekt.
Das Rendern der Ressource selbst ist dann relativ einfach. Der wichtigste Schritt ist das Erzeugen der Ressource über die Methode`createResource()`des Resource-Handlers. Diese Methode nimmt als Parameter entweder nur den Namen oder den Namen und die Bibliothek der Ressource und gibt eine Instanz der Klasse`Resource`zurück, über die wir vollen Zugriff erhalten. Das Bild wird über den Response-Writer als`img`-Element gerendert, dessen`src`-Attribut auf eine spezielle Ressource-URL gesetzt ist. Diese URL wird von der Methode`Resource.getRequestPath()`berechnet. Wenn der Browser beim Darstellen der Seite das Bild mit dieser URL nachlädt, liefert JSF den Inhalt der Ressource an den Client aus.
Listing**Direktes Rendern von Ressourcen**zeigt, wie das Rendern eines der beiden Bilder mit dem zugehörigen JavaScript-Code aussieht.

```
Application app = ctx.getApplication();
ResourceHandler handler = app.getResourceHandler();
Resource spinUpRes = handler.createResource(
    "spin-up.png", "mygourmet");
String onclickUp = MessageFormat.format(
    "return changeNumber(''{0}'', {1});",
    clientId, spinner.getInc());
writer.startElement("img", spinner);
writer.writeAttribute("class", "inputSpinner-button", null);
writer.writeAttribute("src", spinUpRes.getRequestPath(), null);
writer.writeAttribute("onclick", onclickUp, null);
writer.endElement("img");
```

Wenn Sie eine große Anzahl von Komponenten schreiben, werden Sie diese wahrscheinlich in Komponenten-bibliotheken als Jar-Dateien verpacken. Mit dem Einsatz von JSF-Ressourcen ist das kein Problem mehr, da die benötigten Klassen und Ressourcen in einer Jar-Datei zusammengefasst werden können. Eine detaillierte Beschreibung dazu finden Sie in Abschnitt**[Sektion: Die eigene Komponentenbibliothek]**.

6.2.4 Registrieren der Komponenten- und der Rendererklasse

Die soeben verfassten Komponenten- und Rendererklassen müssen jetzt noch mit einem Eintrag in der`faces-config.xml`in der Faces-Umgebung registriert werden.

Listing**Registrierung einer Komponentenklasse**zeigt die Registrierung der Komponentenklasse des Beispiels unter dem Komponententyp`at.irian.InputSpinner`.

```
<component>
  <component-type>
    at.irian.InputSpinner
  </component-type>
  <component-class>
    at.irian.jsfatwork.gui.jsf.component.InputSpinner
  </component-class>
</component>
```

Wie schon bei Konvertern und Validatoren erlaubt JSF ab Version 2.0 das Registrieren von Komponenten mit einer Annotation. Das Annotieren der Komponentenklasse mit`@FacesComponent`reicht, um die Komponente unter dem im Element`value`angegebenen Komponententyp zu registrieren.
Listing**Registrierung einer Komponentenklasse mittels Annotation**zeigt den dazu passenden Code.

```
@FacesComponent("at.irian.InputSpinner")
public class InputSpinner extends UIInput {
...
}
```

JSF 2.2:Ab JSF 2.2 ist das Element`value`der Annotation`@FacesComponent`optional und wird mit einer Namenskonvention ergänzt. Ist es nicht angegeben, benutzt JSF den Klassennamen mit einem kleinen Anfangsbuchstaben als Komponenten-ID. Für das Beispiel aus Listing**Registrierung einer Komponentenklasse mittels Annotation**wäre das die ID`inputSpinner`.
Bei der Registrierung des Renderers muss zuerst ein Renderkit ausgewählt werden, für das die Eintragung der zusätzlichen Rendererklasse erfolgt. Die Auswahl ist meistens einfach, und fast immer bleibt es bei der Verwendung des Standard-Renderkits mit der Renderkit-ID`HTML_BASIC`.
Listing**Registrierung einer Rendererklasse**zeigt die Registrierung des Beispielrenderers unter der Komponentenfamilie`javax.faces.Input`und dem Renderertyp`at.irian.InputSpinner`.

```
<render-kit>
  <render-kit-id>HTML_BASIC</render-kit-id>
  <renderer>
    <component-family>javax.faces.Input</component-family>
    <renderer-type>at.irian.InputSpinner</renderer-type>
    <renderer-class>
      at.irian.jsfatwork.gui.jsf.component.InputSpinnerRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

Ein Renderer kann ab JSF-Version 2.0 auch mit der Annotation`@FacesRenderer`im System registriert werden. Die notwendigen Daten werden in den Elementen`renderKitId`,`componentFamily`und`rendererType`angegeben. Die Renderkit-ID kann auch weggelassen werden und wird dann auf das Standard-Renderkit gesetzt. Listing**Registrierung einer Rendererklasse mittels Annotation**zeigt den Code.

```
@FacesRenderer(componentFamily = "javax.faces.Input",
    rendererType = "at.irian.InputSpinner")
public class InputSpinnerRenderer extends Renderer {
  ...
```

```
}
```

6.2.5 Tag-Definition schreiben

Alle bisherigen Schritte, das Erstellen der Komponenten- und Rendererklasse und das Registrieren der beiden Klassen im System, gestalten sich unabhängig von der eingesetzten Seitendeklarationssprache immer gleich. Bei der Definition des Tags der Komponente ist das leider nicht mehr möglich.
Mit JSP ist die Definition des Tags und das damit verbundene Erstellen der Tag-Klasse ein recht mühsames Unterfangen. Da mit Version 2.0 der Spezifikation der Fokus eindeutig auf Facelets gelegt wurde und JSP nur noch eine Nebenrolle spielt, werden wir darauf an dieser Stelle nicht mehr näher eingehen.
Mit Facelets erfordert das Erstellen einer Tag-Definition keinen großen Aufwand. Existiert bereits eine passende Tag-Bibliothek reicht die Angabe des Tag-Namens, des Komponententyps und des Renderertyps, um das Tag zu spezifizieren. Ist das nicht der Fall, muss zuerst eine neue Tag-Bibliothek angelegt und dem System bekannt gemacht werden. Wie das funktioniert, zeigt Abschnitt**Sektion: Tag-Bibliotheken mit Facelets erstellen**. Die komplette Definition des Tags für unsere Beispielkomponente zeigt Listing**Definition des Tags der Beispielkomponente**.

```
<tag>
  <tag-name>inputSpinner</tag-name>
  <component>
    <component-type>at.irian.InputSpinner</component-type>
    <renderer-type>at.irian.InputSpinner</renderer-type>
  </component>
</tag>
```

JSF 2.2:Mit JSF 2.2 lässt sich der Aufwand zum Erstellen eines Komponenten-Tags auf ein absolutes Minimum reduzieren. Im einfachsten Fall reicht es aus, das Element`createTag`von`@FacesComponent`auf`true`zu setzen. JSF stellt die Komponente dadurch per Konvention im Namensraum`http://xmlns.jcp.org/jsf/component`zur Verfügung. Der Name des Tags leitet sich dabei automatisch aus dem Klassenname (mit kleinem Anfangsbuchstaben) ab.
Wenn Sie eigene Werte für den Namensraum oder den Tag-Namen verwenden wollen, müssen Sie diese explizit angeben. Der Tag-Name kann im Element`tagName`und der Namensraum im Element`namespace`gesetzt werden. Abbildung**Definition des Tags mittels Annotation**zeigt ein konkretes Beispiel.

```
@FacesComponent(value="at.irian.InputSpinner",
    createTag=true, tagName="inputSpinner",
    namespace="http://at.irian/test")
public class InputSpinner extends UIInput {
...
}
```

Diese Methode eignet sich gut zur schnellen Definition von Tags für einzelne Komponenten. Spätestens wenn Sie mehr als eine Handvoll Komponenten haben, sollten Sie aber eine vollständige Tag-Bibliothek inklusive XML-Konfiguration in Betracht ziehen.
Attribute über Reflection:Woher weiß Facelets, welche Attribute für das Tag verfügbar sind? Über*Reflection*greift Facelets auf die Komponentenklasse zu und ermittelt die möglichen Attribute aus dieser Klasse.

6.2.6 Tag-Behandlungsklasse schreiben

In seltenen Fällen wird auch für Facelet-Tags eine Behandlungsklasse benötigt. Das ist beispielsweise dann der Fall, wenn hinter dem Tag keine Komponente existiert, wie das für `c:if` aus der JSTL-Bibliothek von Facelets der Fall ist. Listing **Tag-Handler für c:if** zeigt den Code dieses Tag-Handlers.

```
public final class IfHandler extends TagHandler {
  private final TagAttribute test;
  private final TagAttribute var;

  public IfHandler(TagConfig config) {
    super(config);
    this.test = this.getRequiredAttribute("test");
    this.var = this.getAttribute("var");
  }
  public void apply(FaceletContext ctx, UIComponent parent)
      throws IOException, FacesException, ELException {
    boolean b = this.test.getBoolean(ctx);
    if (this.var != null) {
      ctx.setAttribute(var.getValue(ctx), new Boolean(b));
    }
    if (b) this.nextHandler.apply(ctx, parent);
  }
}
```

Im Konstruktor wird der Tag-Handler mit den Werten aus der Seite initialisiert. Mit dem Aufruf von `getRequiredAttribute()` wird garantiert, dass das Attribut in der Seitendeklaration gesetzt ist. Ist das nicht der Fall, wirft Facelets eine Exception. Essenziell ist dann die Methode `apply` - die immer dann aufgerufen wird, wenn das Tag beim Bauen des Komponentenbaums intern aufgerufen wird. Hier findet die eigentliche Logik statt, die im Falle von `c:if` die Kindkomponenten des Tags abhängig vom Wert des Attributs `test` in den Komponentenbaum einfügt oder nicht. Da die Kindkomponenten explizit über einen Aufruf von `this.nextHandler.apply()` abgearbeitet werden, ist es ein Leichtes, diesen Vorgang zu beeinflussen.
Tag-Handler wie der soeben beschriebene werden in der Tag-Bibliothek direkt in einer Tag-Definition verwendet und können so in der Seitendeklaration wie eine Komponente eingesetzt werden. Hier gilt es allerdings, zu beachten, dass ein Tag-Handler nur beim Aufbau des Komponentenbaums aufgerufen wird. Die Definition des Tags sieht in diesem Fall wie folgt aus:

```
<tag>
  <tag-name>if</tag-name>
  <handler-class>IfHandler</handler-class>
</tag>
```

Tag-Handler können aber auch für Komponenten verwendet werden, die ein spezielles Verhalten erfordern. Zur Demonstration erstellen wir einen Tag-Handler für unsere Input-Spinner-Komponente, der das Attribut `inc` verpflichtend macht. Den Sourcecode des Tag-Handlers finden Sie in Listing **Tag-Handler für die Beispielkomponente**.

```
public class InputSpinnerTagHandler extends ComponentHandler {
  private TagAttribute inc;
```

```
  public InputSpinnerTagHandler(ComponentConfig conf) {
    super(conf);
    this.inc = getRequiredAttribute("inc");
  }
}
```

Dieser Tag-Handler kann jetzt zusätzlich bei der Definition des Kompo-nenten-Tags angegeben werden. Listing**Definition des Tags der Beispielkomponente mit Tag-Handler**zeigt die um den Tag-Handler erweiterte Definition.

```
<tag>
  <tag-name>inputSpinner</tag-name>
  <component>
    <component-type>at.irian.InputSpinner</component-type>
    <renderer-type>at.irian.InputSpinner</renderer-type>
    <handler-class>
      at.irian.jsfatwork.gui.jsf.component.InputSpinnerTagHandler
    </handler-class>
  </component>
</tag>
```

6.2.7 Tag-Bibliothek einbinden

Zu guter Letzt bleibt nur noch ein Schritt übrig: Bevor die Komponente in einer Deklaration zum Einsatz kommen kann, muss die Tag-Bibliothek eingebunden werden. Wie das funktioniert, haben wir ja bereits bei den Standardkomponenten und diversen anderen Gelegenheiten gezeigt und muss hier nicht mehr wiederholt werden.
An dieser Stelle möchten wir Ihnen noch einmal veranschaulichen, wie JSF beim Aufbau des Komponentenbaums die Komponenten- und Rendererklasse für ein Tag auflöst. Trifft JSF beispielsweise auf das Tag`mg:inputSpinner`, ist aus der Definition in der Tag-Bibliothek bereits der Komponententyp`at.irian.InputSpinner`und der Renderertyp`at.irian.InputSpinner`bekannt. Über den Komponententyp kann jetzt bereits die Komponenteninstanz erstellt und in den Baum eingefügt werden. Mit dieser Information lässt sich auch die Rendererklasse auflösen. Wenn wir davon ausgehen, dass das Standard-Renderkit mit dem Bezeichner`HTML_BASIC`zum Einsatz kommt, kann mit dem Renderertyp und der Komponentenfamilie die Klasse des Renderers aus der Konfiguration bestimmt werden. Die Komponentenfamilie ist ja jederzeit über einen Aufruf der Methode`getFamily()`der Komponente abrufbar.
Das ist alles, was man über das Schreiben von eigenen Komponenten wissen muss - viel Erfolg beim Erstellen der dynamischsten, interaktivsten und schönsten JSF-Komponenten!

6.3 Kompositkomponenten und klassische Komponenten kombinieren

Wir haben Ihnen in den letzten beiden Abschnitten die Entwicklung von Kompositkomponenten und klassischen Komponenten nähergebracht. In diesem Abschnitt werden wir Ihnen zeigen, dass sich diese Konzepte nicht gegenseitig ausschließen, sondern ganz im Gegenteil sogar sehr gut harmonieren.
Bei der Entwicklung von Kompositkomponenten tritt immer wieder der Fall ein, dass ein gewünschtes Verhalten nur mit Java-Code realisierbar ist. Wir benötigen also einen Erweiterungspunkt zur Integration dieses Codes. Wie wir Ihnen bereits gezeigt haben, sind Kompositkomponenten intern aus klassischen Komponenten aufgebaut. Der naheliegendste Gedanke ist daher, diesen Java-Code in Form einer klassischen Komponente zu realisieren.
JSF verfolgt exakt diesen Gedanken und erlaubt bei Kompositkomponenten die freie Wahl des Typs der Wurzelkomponente. Über das Attribut`componentType`im Element`cc:interface`kann der Komponententyp dieser Komponente explizit angegeben werden. Die eingesetzte Komponentenklasse muss als einzige Voraussetzung das Interface implementieren und in`getFamily()`die Komponentenfamilie`javax.faces.NamingContainer`zurückliefern. Wird`component-Type`nicht gesetzt, erzeugt JSF automatisch eine Komponente vom Typ .
Genau das werden wir jetzt für die in Abschnitt**[Sektion: Die Komponente mc:collapsiblePanel]**vorgestellte Kompositkomponente`collapsiblePanel`machen. Dort haben wir ja bereits kritisch angemerkt, dass Benutzer der Komponente die Logik zum Ein- und Ausblenden selbst bereitstellen müssen. Wir werden diese Funktionalität in der Komponente`CollapsiblePanel`umsetzen, die wir dann mit der Kompositkomponente verknüpfen. Die Komponente selbst kann dabei sehr einfach gehalten werden. Sie muss lediglich über die Eigenschaft`collapsed`und eine Ereignisbehandlungsmethode zum Ein- und Ausblenden verfügen. Listing**Komponente CollapsiblePanel**zeigt die Komponentenklasse, die mit`@FacesComponent`unter dem Komponententyp`at.irian.CollapsiblePanel`registriert wird.

```
@FacesComponent("at.irian.CollapsiblePanel")
public class CollapsiblePanel extends UINamingContainer {
  enum PropertyKeys {collapsed}

  public boolean isCollapsed() {
    return (Boolean)getStateHelper().eval(
        PropertyKeys.collapsed, Boolean.FALSE);
  }
  public void setCollapsed(boolean collapsed) {
    getStateHelper().put(PropertyKeys.collapsed, collapsed);
  }
  public void toggle(ActionEvent e) {
    setCollapsed(!isCollapsed());
    setCollapsedValueExpression();
  }
  private void setCollapsedValueExpression() {
    ELContext ctx = FacesContext.getCurrentInstance()
        .getELContext();
    ValueExpression ve = getValueExpression(
        PropertyKeys.collapsed.name());
    if (ve != null) ve.setValue(ctx, isCollapsed());
  }
}
```

Da die Komponentenklasse von`UINamingContainer`abgeleitet ist, brauchen wir keine weiteren Vorkehrungen treffen und können sie direkt in der Kompositkomponente verwenden. Ist Ihre Komponente von einer anderen Klasse wie etwa`UIInput`abgeleitet, muss sie zusätzlich das Interface`NamingContainer`implementieren und in`getFamily()`den Wert`javax.faces.NamingContainer`zurückliefern. Der Wert der Eigenschaft`collapsed`wird intern mit dem bereits bekannten`StateHelper`verwaltet. Ein Aufruf der Ereignisbehandlungsmethode`toggle`ändert lediglich den Wert dieser Eigenschaft. Hat der Benutzer der Kompositkomponente im Attribut`collapsed`eine Value-Expression angegeben, wird mit der Methode`setCollapsedValueExpression()`zusätzlich der aktuelle Einklappzustand

zurückgeschrieben.
Wenden wir uns nun der Kompositkomponente selbst zu. Die Interna ändern sich im Vergleich zu Abschnitt**[Sektion: Die Komponente mc:collapsiblePanel]**nur minimal und werden etwas anders verdrahtet. Listing**Kompositkomponente collapsiblePanelmit benutzerdefinierter Wurzelkomponente**zeigt die aktualisierte Komponente`collapsiblePanel`mit gesetzem`componentType`-Attribut.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:cc="http://xmlns.jcp.org/jsf/composite"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<cc:interface componentType="at.irian.CollapsiblePanel">
  <cc:attribute name="collapsed"/>
  <cc:actionSource name="toggle"/>
  <cc:facet name="header"/>
</cc:interface>
<cc:implementation>
  <h:panelGroup layout="block"
      styleClass="collapsiblePanel-header">
    <h:commandButton actionListener="#{cc.toggle}"
        id="toggle" styleClass="collapsiblePanel-img"
        image="#{resource[cc.collapsed
          ? 'mygourmet:toggle-plus.png'
          : 'mygourmet:toggle-minus.png']}"/>
    <cc:renderFacet name="header"/>
  </h:panelGroup>
  <h:panelGroup layout="block" rendered="#{!cc.collapsed}">
    <cc:insertChildren/>
  </h:panelGroup>
</cc:implementation>
</ui:composition>
```

Als erster Schritt wird das verpflichtende Attribut`model`durch das optionale Attribut`collapsed`ersetzt. Damit kann der Einklappzustand von außen über eine Value-Expression mit einer Bean-Eigenschaft verknüpft werden. Das eröffnet dem Benutzer die Möglichkeit, einen initialen Wert zu setzen und den aktuellen Einklappzustand abzuspeichern. Das Auswerten des initialen Zustands aus dem Attribut`collapsed`wird dabei intern automatisch vom`StateHelper`der Komponente erledigt.
Die zweite Änderung betrifft die Ereignisbehandlungsmethode`toggle`und die Eigenschaft`collapsed`. Da beide jetzt direkt von der Wurzelkomponente zur Verfügung gestellt werden, ändern sich die EL-Ausdrücke für den Zugriff auf`cc.toggle`und`cc.collapsed`. Das ist möglich, da die mit`cc`referenzierte Komponente jetzt eine Instanz der zuvor erstellten Klasse`CollapsiblePanel`ist.
Damit ist die verbesserte Version der Komponente`collapsiblePanel`auch schon einsatzbereit. Jetzt können wir tatsächlich von einem eigenständigen und wiederverwendbaren Baustein sprechen. Der nächste logische Schritt wäre jetzt, die Komponente inklusive aller als Jar-Datei zur Verfügung zu stellen. Abschnitt**[Sektion: Die eigene Komponentenbibliothek]**zeigt, wie das funktioniert.

6.4 Alternativen zur eigenen Komponente

Eine Komponente besteht aus den Teilen Komponentenklasse, Rendererklasse und einem optionalen Tag-

Handler. Alle diese Teile sind miteinander verbunden, können aber auch getrennt voneinander ausgetauscht werden. Die einfachste Möglichkeit, eine Komponente zu verändern, ohne eine neue Komponente schreiben zu müssen, ist der Austausch der Rendererklasse.

6.4.1 Austausch der Rendererklasse

Um die Rendererklasse auszutauschen, muss zuerst die Konfiguration in der`faces-config.xml`verändert werden. Listing**Konfiguration eines Renderers**zeigt ein Beispiel.

```
<render-kit>
  <render-kit-id>HTML_BASIC</render-kit-id>
  <renderer>
    <component-family>javax.faces.Output</component-family>
    <renderer-type>javax.faces.Label</renderer-type>
    <renderer-class>
      mypackage.RequiredLabelRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

Beispiel: RequiredLabel:Hier wird der Renderer für die`Label`-Komponente durch die Klasse`mypackage.RequiredLabelRenderer`überschrieben. Jetzt bleibt nur noch, diese Klasse zu implementieren. Listing**Rendern eines Labels mit Stern für Pflichtfelder**zeigt eine Implementierung, in der das`required`-Attribut der zum Label gehörenden Komponente ausgewertet wird.

```
public class RequiredLabelRenderer extends HtmlLabelRenderer {
  protected void encodeBeforeEnd(FacesContext facesContext,
      ResponseWriter writer, UIComponent uiComponent)
      throws IOException {
    String forAttr = getFor(uiComponent);
    if (forAttr != null) {
      UIComponent forComponent =
          uiComponent.findComponent(forAttr);
      if (forComponent instanceof UIInput &&
          ((UIInput) forComponent).isRequired()) {
        writer.startElement(HTML.SPAN_ELEM, null);
        writer.writeAttribute(HTML.ID_ATTR,
            uiComponent.getClientId(facesContext)+
            "RequiredLabel", null);
        writer.writeAttribute(HTML.CLASS_ATTR,
            "requiredLabel", null);
        writer.writeText("*", null);
        writer.endElement(HTML.SPAN_ELEM);
      }
    }
  }
}
```

Je nach dem Wert dieses Attributs wird ein Stern an die Label-Beschreibung angefügt. Zu diesem Zweck wird der Pflichtfeldrenderer von`HtmlLabelRenderer`abgeleitet und die Methode`encodeBeforeEnd()`der Basisklasse überschrieben. In dieser Methode wird zuerst die zum

Label gehörende Komponente gesucht; anschließend wird das -Attribut dieser Komponente abgefragt. Gehört die Komponente zu einem Pflichtfeld, wird ein`span`-Tag mit einer CSS-Klasse und einem * als Inhalt ausgegeben. Sehr einfach und sehr effektiv! Beachten Sie aber bitte, dass die Klasse`HtmlLabelRenderer`aus*Apache MyFaces*stammt und nicht im Standard enthalten ist. Nichtsdestotrotz ändert sich, auch wenn Sie*Mojarra*einsetzen, an der grundlegenden Funktionalität nichts.

6.4.2 Austausch der Komponentenklasse

Genauso wie der Austausch der Rendererklasse ist auch der Austausch der Komponentenklasse möglich - in der`faces-config.xml`-Datei ist ein zusätzlicher Eintrag wie folgt vorzunehmen:

```
<component>
  <component-type>javax.faces.HtmlInputText</component-type>
  <component-class>
    mypackage.SpecialHtmlInputText
  </component-class>
</component>
```

Auto-Converter:Mit dieser Vorgehensweise kann beispielsweise automatisch ein Konverter mit der Komponente verbunden werden, ohne dass dazu ein eigenes Konverter-Tag verwendet wird. Ein Beispiel für eine solche Klasse:

```
public class SpecialHtmlInputText extends HtmlInputText {
  public SpecialHtmlInputText() {
    super();
    setConverter(ConverterFactory.getSpecialConverter());
  }
}
```

Damit kommt dieser Spezialkonverter für alle Elemente dieser Komponenten zum Einsatz!

6.4.3 Benutzerdefinierte Komponente aus den Backing-Beans- - Component-Binding

Die beiden bisher verwendeten Tricks gelten für alle Elemente eines Komponententyps - was ist zu tun, wenn man nur einzelne Komponenten mit diesem Spezialverhalten auszeichnen möchte und andere nicht? Beispiel: Zeilenumbruch:Ein Beispiel aus der Praxis: Für eine Applikation wurde eine Verbindung zu einer auf einem AS400-Server laufenden Legacy-Datenbank entwickelt. Die vom Server zurückgegebenen Daten

waren mit einem `r` zur Markierung des Zeilenumbruchs ausgezeichnet - auf dem Frontend sollte diese Markierung ebenfalls zu einem Zeilenumbruch im HTML-Markup führen, musste also als `<br/>` ausgegeben werden. Da nicht alle Textausgaben geparst werden sollten, wurde das Rendering-Verhalten nur für einen Teil der Komponenten ersetzt.
Um diese Ersetzung vorzunehmen, kann man entweder ein eigenes Tag erstellen und über dieses Tag einen neuen Renderer für die Komponente festlegen, man kann aber auch Component-Binding einsetzen, wobei dieser zweite Weg wesentlich einfacher ist. Dazu sind zuerst alle Komponenten, die ein spezielles Rendering-Verhalten aufweisen sollen, mit einem `binding`-Attribut zu versehen. Im nächsten Schritt wird dieses Attribut mit der dahinterliegenden Geschäftslogik verbunden. Das folgende Beispiel zeigt einen Ausschnitt aus einer solcherart veränderten Seitendeklaration:

```
<h:outputText value="#{limitDetail.limitView.comment}"
    binding="#{componentBean.outputWithBreaks}"/>
```

Die referenzierte Methode sieht dabei so aus:

```
UIComponent getOutputWithBreaks() {
  return new OutputTextWithBreaks();
}
```

Jetzt benötigen wir nur noch eine Implementierung dieser Komponente - in Listing **Verändern des Rendering-Verhaltens einer Komponente durch Component-10mmBinding** wird diese gezeigt. Es wird die `encodeEnd()`-Methode überschrieben - wo ja üblicherweise ein Renderer für die Komponente gesucht und dessen `encodeEnd()`-Methode aufgerufen wird. In diesem Fall erledigen wir das Rendering gleich selbst in der Komponente. Das eigentliche Rendering ist in der Abbildung ausgeblendet, da es exakt der Funktionalität in der Rendererklasse entsprechen soll.

```
public static final class OutputText extends HtmlOutputText {
  public OutputText() {
    super();
  }
  public void encodeEnd(FacesContext context)
      throws IOException {
    String text = RendererUtils.getStringValue(context, this);
    text = HTMLEncoder.encode(text, true, true);
    text = text.replaceAll("\r","<br/>");
    renderOutputText(context, this, text, false);
  }
  public static void renderOutputText(
      FacesContext ctx, UIComponent component,
      String text, boolean escape)
      throws IOException {
    ...
  }
}
```

6.5 MyGourmet 13: Komponenten

und Services

Das Beispiel*MyGourmet 13*integriert alle in diesem Kapitel entwickelten Komponenten - sowohl die Kompositkomponenten aus Abschnitt**[Sektion: Kompositkomponenten]**als auch die klassische Komponente aus Abschnitt**[Sektion: Klassische Komponenten]**und deren Kombination aus Abschnitt**[Sektion: Kompositkomponenten und klassische Komponenten kombinieren]**. Neben den vielen neuen Komponenten gibt es noch die neue Ansicht`editProvider.xhtml`zum Bearbeiten eines Anbieters. Im Zuge dieser Änderung haben wir die Architektur der Anwendung ein klein wenig optimiert und eine Serviceklasse für Objekte vom Typ`Provider`eingeführt. Listing**MyGourmet 13: ProviderService**zeigt das Interface`ProviderService`der Serviceklasse.

```
public interface ProviderService {
  Provider createNew();
  boolean save(Provider entity);
  void delete(Provider entity);
  List<Provider> findAll();
  Provider findById(long id);
}
```

Der Grund für diese Optimierung ist schnell erklärt. Nachdem die Managed-Bean`ProviderBean`im View-Scope abgelegt ist, wird sie für jede Ansicht neu erstellt. Das bedeutet aber auch, dass die Liste der Anbieter jedes Mal neu initialisiert wird. Im letzten Beispiel war das noch kein Problem, da die Anbieterdaten nicht veränderbar waren. Mit der neuen Ansicht`editProvider.xhtml`wird es allerdings zum Problem, da die vom Benutzer veränderten Daten beim Verlassen der Seite verloren gehen - sie sind ja nur in der Bean gespeichert. Mit einem Service im Application-Scope wird quasi eine Datenbank simuliert und das Problem tritt nicht mehr auf. Als zusätzlicher Vorteil macht eine bereits existierende Serviceklasse einen Umstieg auf eine echte Datenbank sehr einfach. Mit dieser Änderung ist es auch ohne Weiteres möglich, die für die Ansicht`providerList.xhtml`benötigte Funktionalität in die Managed-Bean`ProviderListBean`im Request-Scope auszulagern.
Die Klasse`ProviderServiceImpl`implementiert das Interface`ProviderService`und stellt den eigentlichen Service dar. Sie steht als Managed-Bean unter dem Namen`providerService`im Application-Scope zur Verfügung. Listing**MyGourmet 13: Implementierung des Service**zeigt den Rumpf der Klasse mit den Annotationen. Die Implementierung ist sehr einfach gehalten und basiert intern auf einer Liste, die beim Erzeugen der Bean mit drei Objekten vom Typ`Provider`initialisiert wird.

```
@ManagedBean(name = "providerService")
@ApplicationScoped
public class ProviderServiceImpl implements ProviderService {
  ...
}
```

Viel interessanter ist da schon das automatische Setzen des Service in der Managed-Bean`ProviderBean`über eine Managed-Property. Listing**MyGourmet 13: Setzen des Service in ProviderBean**zeigt das entsprechende Codefragment. Sie müssen sich in dem Fall keine Gedanken mehr über den Service machen. Nach dem Erstellen einer Managed-Bean vom Typ`ProviderBean`setzt JSF den Service automatisch - und das garantiert vor dem ersten Zugriff des Benutzers.

```
@ManagedProperty(value = "#{providerService}")
private ProviderService providerService;
public void setProviderService(
    ProviderService providerService) {
  this.providerService = providerService;
}
```

Aber jetzt zurück zum eigentlichen Thema dieses Kapitels: Komponenten. Die Liste der zur Verfügung stehenden Kompositkomponenten umfasst`mc:panelBox`,`mc:dataTable`,`mc:collapsiblePanel`und`mc:inputSpinner`. Das Präfix`mc`steht dabei für den Namensraum`http://at.irian/mygourmet`der MyGourmet-Tag-Bibliothek. Abbildung**MyGourmet 13: Ressourcen der Bibliothek mygourmet**zeigt den Inhalt der Ressourcenbibliothek`mygourmet`mit allen Kompositkomponenten, Stylesheets, Bildern und Skripten.

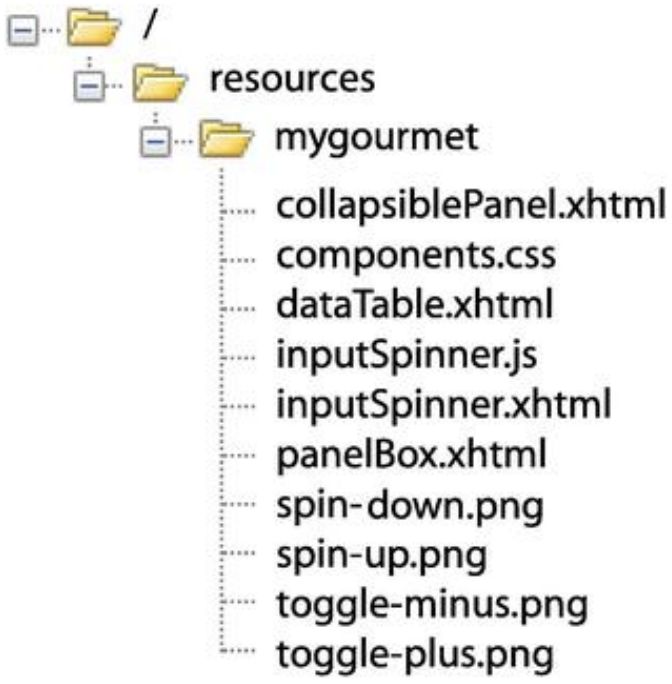


Abbildung:MyGourmet 13: Ressourcen der Bibliothek mygourmet

Die Komponente`mg:inputSpinner`, also die klassische Variante unserer Input-Spinner-Komponente, kann in gleicher Weise wie die Kompositkomponente eingesetzt werden.

6.6 Die eigene Komponentenbibliothek

In diesem Abschnitt zeigen wir, wie einfach das Erstellen einer eigenen Komponentenbibliothek mit JSF 2.0 geworden ist. Dazu packen wir exemplarisch die Kompositkomponente`collapsiblePanel`aus Abschnitt**[Sektion: Kompositkomponenten und klassische Komponenten kombinieren]**inklusive aller benötigten Artefakte in eine Jar-Datei. Nachdem wir die Komponente selbst bereits wiederverwendbar gemacht haben, kann diese Jar-Datei in jeder JSF-Anwendung eingesetzt werden.
Eine erste Version unserer Komponentenbibliothek ist schnell erstellt. Wir müssen lediglich das Verzeichnis der Ressourcenbibliothek`mygourmet`und die Klasse`CollapsiblePanel`der benutzerdefinierten Wurzelkomponente in die Jar-Datei aufnehmen.
Da die Komponente`CollapsiblePanel`mit der Annotation`@FacesComponent`im System registriert wird, brauchen wir eigentlich keine XML-Konfiguration. Im Endeffekt müssen wir aber doch eine leere`faces-config.xml`anlegen. JSF berücksichtigt Annotationen nur in jenen Jar-Dateien, die eine Datei mit dem Namen`faces-config.xml`oder mit der Endung`.faces-config.xml`im Verzeichnis`META-INF`beinhalten. Listing**faces-config.xml für die Komponentenbibliothek**zeigt die leere`faces-config.xml`für JSF 2.2.

```
<faces-config 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"
    version="2.2">
</faces-config>
```

Damit JSF die Ressourcen unserer Bibliothek`mygourmet`aus der Jar-Dateien im Classpath auflösen kann, muss sie im Verzeichnis`META-INF/resources`liegen. Die Komponentenbibliothek ist damit einsatzbereit und kann in jeder beliebigen JSF-Anwendung verwendet werden, sobald das Jar-Archiv im Classpath verfügbar ist. An der Verwendung der Kompositkomponente`collapsiblePanel`hat sich dabei nichts geändert. Sie wird wie bisher über den Namensraum`http://xmlns.jcp.org/jsf/composite/mygourmet`und den Tag-Name`collapsiblePanel`eingebunden.
Diese zugegeben sehr einfache Komponentenbibliothek bietet sich als Basis für Erweiterungen an. Sie kann neben weiteren Kompositkomponenten auch mit klassischen Komponenten, Konvertern und Validatoren ergänzt werden. Die Tags für diese Artefakte müssen allerdings in einer Tag-Bibliothek konfiguriert werden (siehe Abschnitt**Sektion: Tag-Bibliotheken mit Facelets erstellen**). Für unsere Beispiel erstellen wir dazu im Verzeichnis`META-INF`die Tag-Bibliothek`mygourmet.taglib.xml`mit dem Namensraum`http://at.irian/mygourmet`.
Die Tags der Kompositkomponenten und die Tags der Tag-Bibliothek sind dann allerdings über unterschiedliche Namensräume erreichbar. Um das zu vermeiden, erlaubt JSF das Importieren von Kompositkomponenten in Tag-Bibliotheken. Dazu muss im Element`composite-library-name`der Name der Bibliothek - in unserem Fall`mygourmet`- angegeben werden. Dieser Ansatz funktioniert zwar, es lässt sich aber immer nur eine komplette Ressourcenbibliothek pro Tag-Bibliothek einbinden.
JSF 2.2:JSF 2.2 schafft hier Abhilfe und ermöglicht die Definition von Tags für einzelne Kompositkomponenten aus unterschiedlichen Bibliotheken. Dazu muss in der Tag-Bibliothek ein Tag für eine Komponente mit einer Ressourcen-ID im Element`resource-id`hinzugefügt werden. Diese Ressourcen-ID besteht aus dem Bibliotheksnamen und dem durch einen Schrägstrich (`/`) getrennten Ressourcennamen der Kompositkomponente. Listing**Konfiguration der Tag-Bibliothek für die Komponenten-bibliothek**zeigt die Konfiguration des Tags für die Kompositkomponente`collapsiblePanel`aus unserer Tag-Bibliothek.

```
<facelet-taglib 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-facelettaglibrary_2_2.xsd">
  <namespace>http://at.irian/mygourmet</namespace>
  <tag>
    <tag-name>collapsiblePanel</tag-name>
    <component>
      <resource-id>mygourmet/collapsiblePanel.xhtml</resource-id>
    </component>
  </tag>
</facelet-taglib>
```

JSF bindet Tag-Bibliotheken aus Jar-Dateien im Classpath automatisch ein - allerdings nur wenn eine Konfigurationsdatei im Verzeichnis`META-INF`liegt, deren Namen mit der Erweiterung`.taglib.xml`endet. Das Tag`collapsiblePanel`ist jetzt in Applikationen unter dem Namensraum`http://at.irian/mygourmet`verfügbar.
Abbildung**Struktur der eigenen Komponenten-bibliothek**zeigt abschließend noch die Struktur und den Inhalt der Jar-Datei für unser Beispiel.

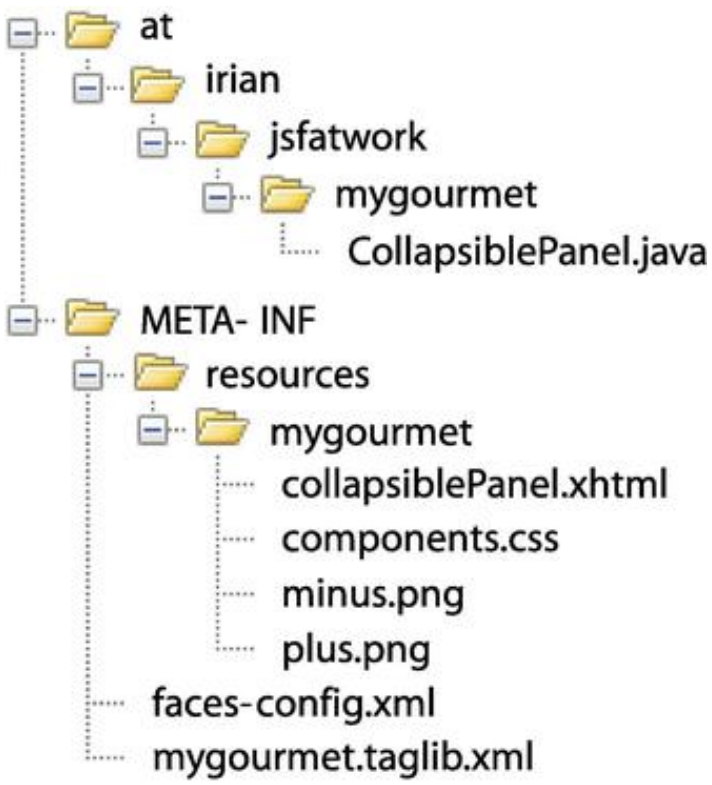


Abbildung:Struktur der eigenen Komponenten-bibliothek

In Abschnitt**[Sektion: *MyGourmet 13*mit Komponentenbibliothek]**finden Sie nochmals das Beispiel*MyGourmet 13*- diesmal allerdings mit einer Tag-Bibliothek, die alle Kompositkomponenten, Komponenten, Validatoren und Konverter in einer eigenen Jar-Datei zusammenfasst.

6.7 MyGourmet 13 mit Komponentenbibliothek

In diesem Abschnitt finden Sie eine kurze Beschreibung zu einer alternativen Version von*MyGourmet 13*. In dieser Version sind die Kompositkomponenten und alle Bestandteile der eingesetzten Tag-Bibliothek in einer Komponentenbibliothek zusammengefasst.
Diese Komponentenbibliothek beinhaltet zum einen im Verzeichnis`META-INF/resources`die Ressourcenbibliothek`mygourmet`und zum anderen die Tag-Bibliothek`mygourmet.taglib.xml`im Verzeichnis`META-INF`. Dazu kommt noch eine Reihe von Java-Klassen wie Komponenten- und Rendererklassen, Konverter und Validatoren. Die Kompositkomponenten sind, wie im letzten Abschnitt beschrieben, in die Tag-Bibliothek importiert. Damit stehen alle Artefakte unter dem Namensraum`http://at.irian/mygourmet`zur Verfügung.
Auf der Ebene des Quellcodes ist die Komponentenbibliothek als Maven-Modul realisiert. Die Projektbeschreibung`pom.xml`des Beispiels*MyGourmet 13 mit Komponentenbibliothek*umfasst zwei Module:`mygourmet13-taglib`enthält alle Bestandteile der Komponentenbibliothek und`mygourmet13-webapp`beinhaltet den Rest der Applikation. Die Verbindung zwischen den beiden Modulen ist in`mygourmet13-webapp`über eine Abhängigkeit auf`mygourmet13-taglib`definiert.
Um die Applikation zu starten, müssen Sie`mvn jetty:run`im Verzeichnis des Moduls`mygourmet13-webapp`aufrufen. Zuvor muss aber das Modul`mygourmet13-taglib`oder das ganze Projekt`mygourmet13`gebaut werden. Das Erstellen von`mygourmet13-taglib`liefert als Ergebnis eine Jar-Datei mit allen Teilen der Komponentenbibliothek.

7 Ajax und JSF

Ajax hat in den letzten Jahren enorm an Bedeutung gewonnen und ist aus der Webentwicklung nicht mehr wegzudenken. Mit der stark zunehmenden Nutzung des Internets in den letzten Jahren stieg auch der Wunsch nach einer interaktiveren und umfangreicheren Benutzung von Webseiten, wie sie im Bereich herkömmlicher Desktop-Applikationen bereits seit vielen Jahren üblich und Benutzern vertraut ist. Diese Lücke soll durch neue Ansätze in der Webtechnologie geschlossen werden. Das grundlegende Ziel muss lauten, das Web noch stärker an die oft sehr unterschiedlichen Bedürfnisse der Benutzer anzupassen.
Geschichte des Web:In der Geschichte des World Wide Web bestand der erste technologische Ansatz (Web in der Version 1.0) darin, dem Benutzer rein statische HTML-Seiten zu präsentieren. Diese waren schwer wartbar, boten kein dynamisches Verhalten und wiesen für den Benutzer eine oft undurchsichtige Benutzerführung auf.
Die nächste Phase wurde durch den Einsatz von dynamischen Inhalten eingeleitet. Der Benutzer bekommt nach Interaktion mit dem Server dynamische und für ihn spezifische Inhalte aufbereitet. Diesem Ansatz für das World Wide Web wird manchmal die Versionsnummer 1.5 zugeschrieben.
Web 2.0:Der Begriff des Web in der Version 2.0 ist eine logische Weiterentwicklung, die den Bedürfnissen der Benutzer folgt. Erstmals im Oktober 2004 für eine Konferenz gebraucht, steht er für vernetzte und personalisierte Plattformen, die Desktop-Applikationen in vielen Bereichen ersetzen sollen.
Web 2.0 ist ein Begriff, dessen Definition recht breit gefasst ist. Ajax ist eine jener Technologien, die die Realisierung von Anwendungen des Web in der Version 2.0 unterstützen sollen. Im Laufe der letzten Jahre erfreute sich Ajax einer immer größeren Beliebtheit, wobei nicht zuletzt auch Google mit seinen auf dieser Technologie aufsetzenden Applikationen zu einem rasanten Aufschwung beitrug.
Bevor wir uns in Abschnitt**[Sektion: Ajax ab JSF 2.0]**auf die Details und erste praktische Beispiele stürzen, werden wir in Abschnitt**[Sektion: Einführung in Ajax -- "`Asynchronous JavaScript And XML"']**den Begriff Ajax etwas genauer unter die Lupe nehmen. In den Abschnitten**[Sektion: Ajax in Kompositkomponenten]**und**[Sektion: Eigene Ajax-Komponenten]**ziehen wir dann alle Register und setzen Ajax mit Kompositkomponenten ein. Abschnitt**[Sektion: *MyGourmet 14*: Ajax]**zeigt anschließend die Integration von Ajax in*MyGourmet*. Zu guter Letzt präsentieren wir in Abschnitt**[Sektion: Werkzeuge für den Ajax-Entwickler]**noch einige Werkzeuge, die Ihnen bei der Arbeit an Ajax-Anwendungen unter die Arme greifen können.

7.1 Einführung in Ajax -- "`Asynchronous JavaScript And XML"'

Ajax ist an sich keine eigenständige Technologie im Webbereich, sondern vielmehr ein Sammelbegriff für eine Vielzahl von Technologien, die bereits seit mehreren Jahren eingesetzt werden. Der NameDefinition von Ajax:wurde zum ersten Mal 2005 in einem Essay über neue Entwicklungen und zukünftige Trends im Bereich von Webapplikationen erwähnt**http://adaptivepath.com/publications/essays/archives/000385.php**:. Ajax steht für

"**A**synchronous**J**avaScript**A**nd**X**ML" und setzt auf Basistechnologien wie XHTML, Cascading Style Sheets (CSS), Document Object Model (DOM), XML, JavaScript und das XMLHttpRequest-Objekt auf.
Der zentrale Baustein einer Ajax-Applikation ist die Skriptsprache JavaScript. Folglich muss JavaScript im Browser aktiviert sein, um die Ajax-Funktionalität nutzen zu können. Üblicherweise greift jede mit Ajax entwickelte Webseite auf eine bereits vorhandene JavaScript-Bibliothek wie etwa*jQuery*zurück. Eine solche Bibliothek bietet Code für die einfache Verwendung des XMLHttpRequest-Objekts an und behandelt oft auch Unterschiede zwischen einzelnen Browsern.
Beim Einsatz von Ajax in einer Webapplikation gibt es Unterschiede hinsichtlich der Intensität der Verwendung. Die einfachste Möglichkeit ist die minimale Integration von Ajax zur Optimierung der herkömmlichen Abläufe einer Applikation. Der Gebrauch mehrerer Ajax-basierter Komponenten ist schon eine Stufe darüber anzusiedeln. Als letzter Schritt ist die Realisierung einer kompletten Applikation auf Basis von JavaScript und Ajax möglich. Für den Bau solcher Anwendungen ist JSF aber nicht mehr die optimale Wahl - dazu eignen sich reine JavaScript-Bibliotheken oder das*Google Web Toolkit-(GWT-)*Framework besser. Wir werden uns daher nur mit den ersten beiden Ansätzen beschäftigen.
Der Vorteil einer typischen Ajax-Webanwendung ist die wesentlich flüssigere Interaktion zwischen dem Browser und dem Webserver. Möglich ist dies durch die Technik, Daten asynchron vom Server zu laden, während clientseitig das System nicht zum Stillstand kommen muss. Außerdem wird die Oberfläche des Browsers nur mit der angeforderten Information aktualisiert, also nicht komplett neu geladen.

7.2 Ajax ab JSF 2.0

Die rasante Entwicklung von Ajax ist natürlich auch an der JSF-Welt nicht spurlos vorübergegangen. Dass JSF und Ajax sich gut vertragen, bewiesen bereits vor JSF 2.0 eine ganze Reihe von Komponenten-bibliotheken und Ajax-Frameworks speziell für JSF. Jede dieser Lösungen funktionierte für sich genommen einwandfrei. Das Problem lag - wie bei vielen Entwicklungen der damaligen Zeit im JSF-Umfeld - an der fehlenden Spezifikation und der daraus entstandenen Inkompatibilität der unterschiedlichen Produkte. Obwohl alle Lösungen das gleiche Ziel - die Integration von JSF und Ajax - verfolgten, unterschieden sich die technische Umsetzung doch teils erheblich. Mit JSF 2.0 war dieses Problem Geschichte, da die Spezifikation eine Standardisierung der Ajax-Unterstützung beinhaltet.
JSF definiert ab Version 2.0 eine JavaScript-Bibliothek, die grundlegende Ajax-Operationen wie das Senden einer Anfrage und das Bearbeiten der Antwort abdeckt. Mit dieser standardisierten Schnittstelle ist gewährleistet, dass alle Komponenten clientseitig dieselbe Funktionalität einsetzen und sich nicht in die Quere kommen.
Diese JavaScript-API bildet auch die Grundlage für die Integration von Ajax in JSF-Anwendungen. JSF bietet grundsätzlich zwei verschiedene Ansätze, um Ansichten mit Ajax-Funktionalität auszustatten. Zum einen können Entwickler zum Absetzen einer Ajax-Anfrage direkt die Funktion der JavaScript-API aufrufen. Zum anderen gibt es mit dem Tag`f:ajax`eine deklarative Variante, um eine Komponente oder sogar einen ganzen Bereich einer Seitendeklaration mit Ajax-Funktionalität auszustatten.
Die Spezifikation kümmert sich allerdings nicht nur um die Clientseite. Auch die Bearbeitung einer Ajax-Anfrage am Server ist umfassend spezifiziert. JSF 2.0 erweiterte den Lebenszyklus so, dass zum Bearbeiten einer Ajax-Anfrage nur die relevanten Teile des Komponentenbaums ausgeführt (*Partial-View-Processing*) und gerendert (*Partial-View-Rendering*) werden.
Im Laufe dieses Abschnitts gehen wir auf die Details der Ajax-Integration in JSF ab Version 2.0 ein. Nach einem ersten Beispiel mit`f:ajax`in Abschnitt**[Sektion: Ein erstes Beispiel mit f:ajax]**folgen in Abschnitt**[Sektion: f:ajax im Einsatz]**noch weitere Beispiele und Abschnitt**[Sektion: Ereignisse und Listener in Ajax-Anfragen]**zeigt, wie sich Ereignisse und Listener bei Ajax-Anfragen verhalten. Abschnitt**[Sektion: JavaScript-API]**widmet sich anschließend der JavaScript-API für Ajax und Abschnitt**[Sektion: Partieller JSF-Lebenszyklus]**gibt Einblicke in den partiellen Lebenszyklus. Abschließend zeigt Abschnitt**[Sektion: Eingabefelder zurücksetzen]**noch wie und vor allem warum, ab JSF 2.2 Eingabefelder bei Ajax-Anfragen zurückgesetzt werden können.

7.2.1 Ein erstes Beispiel mit f:ajax

Als erstes Beispiel werden wir das Ein- und Ausblenden der Kreditkartendaten in der Ansicht`editCustomer.xhtml`auf Ajax umstellen. Bis jetzt löste ein Klick auf das Auswahlfeld "Kreditkarte angeben" über JavaScript ein Übermitteln des Formulars aus. Am Server fand dadurch natürlich ein Durchlauf des Lebenszyklus über den kompletten Komponentenbaum statt. Eigentlich wollen wir aber nur die Eigenschaft`useCreditCard`neu setzen und die zwei Labels und Eingabekomponenten der Kreditkartendaten abhängig davon ein- oder ausblenden.
Wir wollen erreichen, dass ein Klick auf das Auswahlfeld eine Ajax-Anfrage an den Server auslöst. Als Reaktion auf die Anfrage soll der registrierte Value-Change-Listener ausgeführt und der Bereich mit den Kreditkartendaten neu gerendert werden. JSF erleichtert uns ab Version 2.0 diese Aufgabe enorm. Mit dem Tag`f:ajax`kann eine Komponente mit Ajax-Funktionalität ausgestattet werden, indem es als Kind-Tag des mit Ajax-Unterstützung zu versehenden Elements in die Deklaration eingefügt wird. In unserem Beispiel ergibt sich die gewünschte Funktionalität durch das Hinzufügen des`f:ajax`-Elements zur Auswahlkomponente`h:selectBooleanCheckbox`. Listing**Einsatz von f:ajax zum Umschalten der Kreditkartendaten**zeigt die relevanten Teile der Deklaration`editCustomer.xhtml`. Aus Gründen der einfacheren Handhabung haben wir das Formular in zwei Bereiche geteilt: einen für die Basisdaten und einen für die Kreditkartendaten.

```
<h:form id="form">
  <h:panelGrid id="baseData" columns="2">
    ...
    <h:selectBooleanCheckbox id="useCreditCard"
        value="#{customerBean.customer.useCreditCard}"
        valueChangeListener=
            "#{customerBean.useCreditCardChanged}">
      <f:ajax render="ccData"/>
    </h:selectBooleanCheckbox>
  </h:panelGrid>
  <h:panelGrid id="ccData" columns="2">
    ...
  </h:panelGrid>
</h:form>
```

Das Hinzufügen des`f:ajax`-Tags bringt bereits den gewünschten Effekt: Wenn der Benutzer im Browser den Wert des Felds "Kreditkarte angeben" ändert, wird nicht mehr die gesamte Seite neu geladen, sondern nur noch der Bereich mit den Kreditkartendaten neu gezeichnet. Sie können sich davon mit einem Werkzeug wie*Firebug*(siehe Abschnitt**[Sektion: Firebug]**) überzeugen, indem Sie den Typ der abgesetzten Anfrage analysieren. Genauso gut können Sie aber auch die aktuelle Uhrzeit auf der Seite ausgeben - sie wird sich nicht ändern.
Die kleine Änderung hat einen durchschlagenden Effekt. Wie verarbeitet JSF das Tag`f:ajax`? Der gewählte Ansatz ist so einfach wie mächtig. JSF setzt das Tag`f:ajax`beim Rendern unserer Auswahlkomponente als Aufruf der Funktion`jsf.ajax.request()`im Attribut`onchange`um. Mit dem Attribut`render`teilen wir JSF mit, welche Komponente neu gerendert werden soll. In unserem Fall ist das die Panel-Grid-Komponente mit der ID`ccData`.

Tipp: Auch mit Ajax muss weiterhin`h:form`in gewohnter Art und Weise eingesetzt werden.

Beachten Sie bitte auch, dass die`h:selectBooleanCheckbox`-Komponente nicht mehr`immediate`ist. Zuvor war das notwendig, da bei einem Klick auf das Auswahlfeld immer das komplette Formular im

Lebenszyklus ausgeführt wurde und einige der anderen Formularfelder verpflichtend anzugeben sind, und damit unschöne Validierungsmeldungen eingeblendet wurden. Mit Ajax wird der Lebenszyklus nur noch partiell für die eine Komponente ausgeführt, es erscheinen keine Validierungsmeldungen für andere Komponenten.
Schreiben wir das`f:ajax`-Tag in Listing**Einsatz von f:ajax zum Umschalten der Kreditkartendaten**mit den Standardwerten für seine Attribute aus, so erhalten wir:

```
<f:ajax event="valueChange" execute="@this" render="ccData"/>
```

Das Ergebnis der Langschreibweise ist identisch. Der Wert des Attributs`event`bestimmt das Ereignis, von dem die Ajax-Anfrage ausgelöst wird. Mögliche Werte dafür sind`valueChange`für Eingabekomponenten,`action`für Befehlskomponenten und alle anderen HTML-Ereignisse - allerdings ohne das Präfix`on`. Wir könnten also mit dem Wert`click`die Ajax-Anfrage genauso gut durch eine Klick auf das Auswahlfeld auslösen. Das Attribut`execute`definiert, welche Komponenten beim Abarbeiten des Lebenszyklus am Server bearbeitet werden. Die Konstante`@this`bezeichnet dabei die umschließende Komponente. Die Attribute`event`und`execute`können in unserem Fall weggelassen werden, da es sich bei den Werten`valueChange`und`@this`um die Defaultwerte für Eingabekomponenten handelt. Was JSF am Server als Reaktion auf diese Anfrage macht, zeigen wir Ihnen in Abschnitt**[Sektion: Partieller JSF-Lebenszyklus]**.
Listing**Umschalten der Kreditkartendaten mittels JavaScript-API**zeigt die zweite Variante, eine Komponente mit Ajax auszustatten. Nachdem das Absenden einer Ajax-Anfrage immer über die standardisierte JavaScript-API läuft, kann die dafür zuständige Funktion`jsf.ajax.request()`auch direkt verwendet werden. Die Funktion übernimmt als Parameter das auslösende DOM-Element, das Ereignis und ein assoziatives Array mit Optionen. Mehr dazu in Abschnitt**[Sektion: JavaScript-API]**.

```
<h:selectBooleanCheckbox id="useCreditCard"
    value="#{customerBean.customer.useCreditCard}"
    valueChangeListener="#{customerBean.useCreditCardChanged}"
    onchange="jsf.ajax.request(
        this, event, {render: 'form:ccData'});"/>
```

Tipp: Verwenden Sie in allen Seitendeklarationen`h:head`und`h:body`. Nur so kann JSF die für Ajax benötigten Skript-Ressourcen richtig einbinden.

7.2.2 f:ajax im Einsatz

Mit dem Tag`f:ajax`können eine ganze Reihe von Standardkomponenten in JSF mit Ajax-Verhalten ausgestattet werden. Genauer gesagt sind es alle Komponenten, die das Interface`ClientBehaviorHolder`implementieren, wodurch es relativ einfach möglich ist, auch eigene Komponenten oder solche aus Komponentenbibliotheken für`f:ajax`fit zu machen. Damit steht der Kompatibilität von Komponenten aus unterschiedlichen Quellen in Bezug auf Ajax nichts mehr im Wege.
`f:ajax`kann auf zwei Arten zum Einsatz kommen. Zum einen kann eine einzelne Komponente mit Ajax-Verhalten ausgerüstet werden, indem`f:ajax`als Kind-Tag eingefügt wird. Zum anderen ist es auch möglich, mit`f:ajax`einen ganzen Bereich einer Seite auf Ajax umzustellen.
Hier die wichtigsten Attribute des Tags`f:ajax`:

- `event`:
  Name des Ereignisses, das die Ajax-Anfrage auslöst. Mögliche Werte sind`valueChange`für Eingabekomponenten,`action`für Befehlskomponenten und alle anderen HTML-Ereignisse -

allerdings ohne das Präfix`on`. Der Defaultwert dieses Attributs wird von der Komponente bestimmt.

- `execute`:
  Eine durch Leerzeichen separierte Liste der IDs jener Komponenten, die beim Bearbeiten der Ajax-Anfrage durch JSF im Lebenszyklus ausgeführt werden sollen. Kann auch die Konstanten`@this`(das Element selbst),`@form`(das Formular des Elements),`@all`(alle Elemente) und`@none`(kein Element) beinhalten. Wenn das Attribut als Value-Expression gesetzt wird, muss der Typ der Eigenschaft`List<String>`sein. Der Defaultwert ist`@this`.
- `render`:
  Eine durch Leerzeichen separierte Liste der IDs jener Komponenten, die beim Bearbeiten der Ajax-Anfrage durch JSF im Lebenszyklus gerendert werden sollen. Kann auch die Konstanten`@this`,`@form`,`@all`und`@none`beinhalten. Wenn das Attribut als Value-Expression gesetzt wird, muss der Typ der Eigenschaft`List<String>`sein. Der Defaultwert ist`@none`.
- `listener`:
  In diesem Attribut kann eine Methode einer Managed-Bean über eine Method-Expression als Listener registriert werden. Die Methode muss einen Parameter vom Typ`AjaxBehaviorEvent`haben und wird während der Ajax-Anfrage in der Invoke-Application-Phase ausgeführt. Details folgen in Abschnitt**[Sektion: Ereignisse und Listener in Ajax-Anfragen]**.
- `onevent`:
  Erlaubt das Registrieren einer clientseitigen JavaScript-Callback-Funktion für Ajax-Ereignisse. Details folgen in Abschnitt**[Sektion: JavaScript-API]**.
- `onerror`:
  Erlaubt das Registrieren einer clientseitigen JavaScript-Callback-Funktion für Fehler, die beim Bearbeiten der Ajax-Anfrage auftreten. Details folgen in Abschnitt**[Sektion: JavaScript-API]**.
- `disabled`:
  Das Ajax-Verhalten wird "abgeschaltet", wenn dieses Attribut auf`true`gesetzt ist.
- `delay`:
  Erlaubt das Verzögern von Ajax-Anfragen um den in Millisekunden angegebenen Wert. Falls während dieser Zeitspanne mehrere Ajax-Anfrage auftreten, wird nur die aktuellste verarbeitet. Details folgen in Abschnitt**[Sektion: Ajax-Queue kontrollieren]**.
- `resetValues`:
  JSF 2.2 erlaubt das gezielte Zurücksetzen von Eingabefeldern bei Ajax-Anfragen. Details folgen in Abschnitt**[Sektion: Eingabefelder zurücksetzen]**.

Sehen wir uns dazu einige kurze Beispiele an. Das erste Beispiel beinhaltet eine Form mit zwei Eingabefeldern für den Vornamen und den Nachnamen einer Person und einer Schaltfläche. Unterhalb der Schaltfläche befindet sich noch ein Textfeld, mit dem der komplette Name ausgegeben wird. Ohne Ajax bewirkt ein Klick auf die Schaltfläche einen normalen Submit mit anschließendem Neuaufbau der Ansicht. In diesem Fall ist im Textfeld erst nach dem Übermitteln der eingegebene Name sichtbar. Listing**f:ajax im Einsatz: Beispiel 1**zeigt das Beispiel bereits in einer mit Ajax erweiterten Variante. Das Tag`f:ajax`bewirkt, dass die Schaltfläche über Ajax das Ausführen der beiden Eingabekomponenten und das Rendern des Textfelds anstößt. Wenn Sie im Browser einen Vor- und Nachnamen eingeben und die Schaltfläche aktivieren, wird das Textfeld ohne Neuaufbau der Ansicht aktualisiert.

```
<h:form id="form">
  <h:panelGrid columns="1">
    <h:inputText id="first" value="#{test.first}"/>
```

```
    <h:inputText id="last" value="#{test.last}"/>
    <h:commandButton value="Show">
      <f:ajax execute="first last" render="name"/>
    </h:commandButton>
  </h:panelGrid>
  <h:outputText id="name" value="#{test.name}"/>
</h:form>
```

Das Attribut`execute`enthält die IDs`first`und`last`der beiden Eingabekomponenten und das Attribut`render`die ID`name`der Textkomponente. Diese Werte werden in der Ajax-Anfrage inkludiert und JSF führt den Lebenszyklus nur für die beiden Eingabekomponenten aus und rendert anschließend das Textfeld neu. Das Attribut`event`ist nicht explizit gesetzt, für`h:commandButton`ist aber als Defaultereignis`action`definiert.

Tipp: Komponenten, die mittels Ajax neu gerendert werden - deren IDs also im Attribut`render`von`f:ajax`vorkommen -, müssen immer im DOM der Seite vorhanden sein (siehe Abschnitt**[Sektion: Ajax in Kompositkomponenten]**).

Im zweiten Beispiel kommt ein zusätzliches`f:ajax`-Tag zum Einsatz, um einen ganzen Bereich der Deklaration mit Ajax-Verhalten auszustatten. Listing**f:ajax im Einsatz: Beispiel 2**zeigt die erweiterte Deklaration.

```
<h:form id="form">
  <f:ajax render="name">
    <h:panelGrid columns="1">
      <h:inputText id="first" value="#{test.first}"/>
      <h:inputText id="last" value="#{test.last}"/>
      <h:commandButton value="Show">
        <f:ajax execute="first last" render="name"/>
      </h:commandButton>
    </h:panelGrid>
  </f:ajax>
  <h:outputText id="name" value="#{test.name}"/>
</h:form>
```

Da`f:ajax`im Attribut`event`kein Ereignis definiert, werden nur jene Komponenten innerhalb des Tags mit Ajax-Verhalten ausgestattet, die ein Defaultereignis haben. Konkret sind das die beiden Eingabekomponenten mit dem Ereignis`valueChange`. Das Ajax-Verhalten der Schaltfläche für das Defaultereignis bestimmt bereits das innere`f:ajax`-Tag und ändert sich deshalb nicht. Mit der zusätzlichen Ajax-Funktionalität wird jetzt jedes Mal, wenn sich im Browser eines der beiden Eingabefelder ändert, das Textfeld neu gerendert.
Tabelletab:ajax-default-eventszeigt eine Übersicht aller Standardkomponenten mit Defaultereignissen.

Komponente	Defaultereignis
`h:commandButton`	action
`h:commandLink`	action
`h:inputText`	valueChange
`h:inputTextarea`	valueChange
`h:inputSecret`	valueChange
`h:selectBooleanCheckbox`	valueChange
`h:selectOneRadio`	valueChange
`h:selectOneListbox`	valueChange
`h:selectOneMenu`	valueChange
`h:selectManyCheckbox`	valueChange
`h:selectManyListbox`	valueChange

h:selectManyMenu	valueChange

Listing**f:ajax im Einsatz: Beispiel 3**zeigt das gleiche Beispiel wie Listing**f:ajax im Einsatz: Beispiel 2**, mit dem Unterschied, dass im äußeren`f:ajax`-Tag das Attribut`event`auf`dblclick`gesetzt ist.

```
<h:form id="form">
  <f:ajax event="dblclick" render="name">
    <h:panelGrid columns="1">
      <h:inputText id="first" value="#{test.first}"/>
      <h:inputText id="last" value="#{test.last}"/>
      <h:commandButton value="Show">
        <f:ajax execute="first last" render="name"/>
      </h:commandButton>
    </h:panelGrid>
  </f:ajax>
  <h:outputText id="name" value="#{test.name}"/>
</h:form>
```

Durch diese kleine Änderung lösen jetzt alle Komponenten innerhalb`f:ajax`nach einem Doppelklick eine Ajax-Anfrage aus. Das gilt auch für das Panel-Grid und die Schaltfläche, die jetzt für zwei verschiedene Ereignisse mit Ajax-Verhalten ausgestattet ist.
Das nächste Beispiel in Listing**f:ajax im Einsatz: Beispiel 4**ist wiederum nur eine Variante des bereits bekannten Formulars. Diesmal gibt es jedoch zwei Textfelder zur Ausgabe des Namens: Das erste liegt innerhalb der Form und hat die ID`inner`. Das zweite Textfeld liegt außerhalb und hört auf die ID`outer`. Das äußere Feld soll immer dann aktualisiert werden, wenn sich der Wert eines der Eingabefelder ändert, wohingegen das innere durch einen Klick auf die Schaltfläche aktualisiert werden soll. Das sollte an und für sich kein Problem darstellen. Wenn das Attribut`render`des inneren`f:ajax`-Tags auf`inner`und das des äußeren auf`outer`gesetzt wird, sollte sich doch genau dieses Verhalten einstellen. Tut es aber nicht - JSF beschwert sich schon beim Laden der Seite, dass die ID`outer`nicht existiert. Warum das so ist, klären wir gleich.

```
<h:form id="form">
  <f:ajax render=":outer">
    <h:panelGrid columns="1">
      <h:inputText id="first" value="#{test.first}"/>
      <h:inputText id="last" value="#{test.last}"/>
      <h:commandButton value="Show">
        <f:ajax execute="first last" render="inner"/>
      </h:commandButton>
    </h:panelGrid>
  </f:ajax>
  <h:outputText id="inner" value="#{test.name}"/>
</h:form>
<h:outputText id="outer" value="#{test.name}"/>
```

Das Problem wird vom Textfeld außerhalb der Form verursacht und liegt an der Berechnung der Client-IDs für Ajax-Anfragen. JSF setzt dazu die Methode`UIComponent.findComponent()`ein, die einen bestimmten Algorithmus zum Auffinden der Komponente mit der übergebenen ID anwendet. Dieser Algorithmus geht davon aus, dass die übergebene ID relativ zum nächsthöheren Naming-Container ist. In unserem Fall wird also versucht, die Komponente mit der ID`outer`als Kind der Form mit der ID`form`aufzulösen. Nachdem das Textfeld aber außerhalb liegt, kann das nicht funktionieren. Die Lösung dieses Problems ist einfach:`findComponent()`behandelt alle IDs mit einem führenden Doppelpunkt als absolute IDs, die beginnend beim Wurzelknoten des Komponentenbaums aufgelöst werden. Wir müssen also nur das`render`-Attribut im äußeren`f:ajax`-Tag auf`:outer`setzen, um die gewünschte Funktionalität zu erhalten.

7.2.3 Ereignisse und Listener in Ajax-Anfragen

In diesem Abschnitt werfen wir einen genaueren Blick auf Ereignisse und Listener in Ajax-Anfragen. Der Einsatz von Action-Methoden und Ereignisbehandlungsmethoden für Action- oder Value-Change-Events funktioniert grundsätzlich gleich wie in Nicht-Ajax-Anfragen.
Im Beispiel in Listing**h:selectOneMenu mit f:ajax und Value-Change-Listener**zeigen wir Ihnen, wie Sie mit einfachen Mitteln zweih:selectOneMenu-Komponenten miteinander verknüpfen. Die Auswahlmöglichkeiten der zweiten Komponente sollen von der getätigten Auswahl in der ersten Komponente abhängen und über Ajax aktualisiert werden. Ein gutes Beispiel dafür ist die Auswahl eines Landes und eines dazu passenden Bundeslandes. Mit demf:ajax-Tag verpassen wir der ersten Komponente das dazu nötige Ajax-Verhalten.

```
<h:selectOneMenu id="country" value="#{test.country}"
    valueChangeListener="#{test.countryChanged}">
  <f:selectItems value="#{test.countryItems}"/>
  <f:ajax render="state"/>
</h:selectOneMenu>
<h:selectOneMenu id="state" value="#{test.state}">
  <f:selectItems value="#{test.stateItems}"/>
</h:selectOneMenu>
```

Auf derh:selectOneMenu-Komponente zur Auswahl des Landes ist die Methodetest.countryChangedals Value-Change-Listener registriert. Immer wenn ein Benutzer den Wert im Browser ändert, wird eine Ajax-Anfrage abgesetzt und der Lebenszyklus wird partiell für diese Komponente ausgeführt. Da der Benutzer den Wert geändert hat, ruft JSF auch den Value-Change-Listener auf. In dieser Methode wird zuerst die Liste der Bundesländer abhängig vom gewählten Land neu gesetzt. Dann wird noch das aktuell ausgewählte Bundesland aufnullzurückgesetzt. Listing**Value-Change-Listener zum Initialisieren der Bundesländer**zeigt den relevanten Code. Dieh:selectOneMenu-Komponente mit der IDstatewird anschließend neu gerendert und im Browser mit den neuen Werten aktualisiert.

```
public void countryChanged(ValueChangeEvent ev) {
  state = null;
  updateStateItems((String)ev.getNewValue());
}
private void updateStateItems(String country) {
  if (country != null) {
    stateItems = stateItemsMap.get(country);
  } else {
    stateItems = new ArrayList<SelectItem>();
  }
}
```

Das aktuell ausgewählte Land wird in der MethodecountryChangedaus dem übergebenen Ereignis und nicht aus der EigenschaftcountryderManaged-Bean geholt. Der Value-Change-Listener wird ja bereits in der Process-Validations-Phase des Lebenszyklus aufgerufen und der vom Benutzer ausgewählte Wert ist daher noch nicht in die Eigenschaft zurückgeschrieben worden.

Als Alternative zum Value-Change-Listener kann hier auch ein Ajax-Listener zum Einsatz kommen.`f:ajax`bietet ja die Möglichkeit, im Attribut`listener`eine Method-Expression anzugeben. Listing**h:selectOneMenu mit f:ajax und Ajax-Listener**zeigt das Beispiel von oben mit einem Ajax-Listener. Der Value-Change-Listener wird in diesem Fall nicht mehr benötigt.

```
<h:selectOneMenu id="country" value="#{test.country}">
  <f:selectItems value="#{test.countryItems}"/>
  <f:ajax render="state" listener="#{test.countryChanged}"/>
</h:selectOneMenu>
<h:selectOneMenu id="state" value="#{test.state}">
  <f:selectItems value="#{test.stateItems}"/>
</h:selectOneMenu>
```

Eine Ajax-Listener-Methode muss einen Parameter vom Typ`AjaxBehaviorEvent`haben, wie Listing**Ajax-Listener zum Initialisieren der Bundesländer**zeigt. Der Inhalt der Methode unterscheidet sich in einem wesentlichen Punkt vom Value-Change-Listener in Listing**Value-Change-Listener zum Initialisieren der Bundesländer**. Nachdem JSF den Ajax-Listener erst in der Invoke-Application-Phase aufruft, ist der Wert in der Eigenschaft`country`bereits aktuell. Die Methode`updateStateItems`hat sich im Vergleich zu Listing**Value-Change-Listener zum Initialisieren der Bundesländer**nicht geändert.

```
public void countryChanged(AjaxBehaviorEvent ev) {
  state = null;
  updateStateItems(country);
}
```

In Abschnitt**[Sektion: Partieller JSF-Lebenszyklus]**wird der partielle Lebenszyklus bei Ajax-Anfragen noch einmal genauer unter die Lupe genommen.

7.2.4 JavaScript-API

JSF definiert ab Version 2.0 eine JavaScript-Bibliothek als clientseitige Grundlage für die Integration von Ajax. Die Bibliothek ist in der Ressource mit dem Namen`jsf.js`in der Bibliothek`javax.faces`abgelegt. Wenn Sie Ajax deklarativ mit dem Tag`f:ajax`einsetzen, müssen Sie sich um das Einbinden dieser Ressource nicht kümmern. Nur bei einem direkten Einsatz der JavaScript-API müssen Sie Folgendes angeben:

```
<h:outputScript name="jsf.js" library="javax.faces"
    target="head"/>
```

Um die Skript-Ressource`jsf.js`selbst müssen Sie sich keine Gedanken machen. Sie gehört zum Standardlieferumfang von JSF und ist in den Jar-Dateien inkludiert. Auch beim Einsatz der JavaScript-API ist es wichtig,`h:head`und`h:body`in der Seitendeklaration zu verwenden. Nur so kann garantiert werden, dass das Skript richtig in der gerenderten Ausgabe am Browser ankommt.
Hier eine Auflistung der wichtigsten Funktionen der JavaScript-API:

- `jsf.ajax.request(source, event, options)`:
  Diese Methode sendet eine Ajax-Anfrage an den Server.
- `jsf.ajax.response(request, context)`:
  Diese Methode bearbeitet die Anwort des Servers auf die Ajax-Anfrage und ist für Endanwender

nicht relevant.

- `jsf.ajax.addOnError(callback)`:
  Diese Methode registriert eine Callback-Funktion zum Behandeln von Fehlern, die während der Bearbeitung der Ajax-Anfrage aufgetreten sind.
- `jsf.ajax.addOnEvent(callback)`:
  Diese Methode registriert eine Callback-Funktion zum Behandeln von Ajax-Ereignissen.

7.2.4.1 Senden von Ajax-Anfragen

Die Methode`jsf.ajax.request(source, event, options)`ist das Herzstück der JavaScript-API von JSF. Sie allein ist für das Senden der asynchronen Ajax-Anfragen an den Server zuständig. Das Tag`f:ajax`ist nur eine einfachere Variante, um das Ajax-Verhalten auf deklarativem Weg in die Ansicht zu bringen. Beim Rendern macht JSF daraus wieder einen Aufruf der Funktion`jsf.ajax.request()`.
Mit den Parametern`source`und`event`werden das DOM-Element und das DOM-Ereignis übergeben, die für das Auslösen der Ajax-Anfrage verantwortlich sind. Der dritte Parameter`options`ist ein assoziatives Array, mit dem weitere Optionen als Schlüssel-Wert-Paare an die Funktion übergeben werden. Diese Optionen entsprechen weitestgehend den bereits bekannten Attributen aus`f:ajax`. Über`execute`werden IDs auszuführender Komponenten angegeben und über`render`IDs von Komponenten, die neu zu zeichnen sind. Während beim Einsatz von`f:ajax`in den meisten Fällen eine relative ID genügt, muss beim Aufruf der JavaScript-Funktion immer die komplette Client-ID angegeben werden. Die Funktion operiert ja direkt auf DOM-Ebene und kann keine relativen IDs über den Komponentenbaum von JSF auflösen.
Die Funktion`jsf.ajax.request()`baut die Ajax-Anfrage zusammen und schickt sie asynchron an den Server. Was JSF dort genau mit dieser Anfrage macht, verraten wir Ihnen in Abschnitt**[Sektion: Partieller JSF-Lebenszyklus]**. Sobald die Antwort auf die Anfrage vom Server zurückkommt, wird im Erfolgsfall die Funktion`jsf.ajax.response()`aufgerufen, um die Antwort zu verarbeiten und gegebenenfalls Teile der Ansicht neu aufzubauen.
Listing**Beispiel 5 mit Ajax über JavaScript**zeigt noch einmal das Beispiel mit den beiden Auswahllisten aus dem letzten Abschnitt. Diesmal ist das Ajax-Verhalten allerdings direkt über die JavaScript-API realisiert. Genau wie vorher löst das Ändern des Werts in der ersten Komponente eine Ajax-Anfrage aus, die ein Neuzeichnen der zweiten Komponente veranlasst.

```
<h:form id="form">
  <h:selectOneMenu id="country" value="#{test.country}"
      valueChangeListener="#{test.changeCountry}"
      onchange="jsf.ajax.request(
          this, event, {render: 'form:state'})">
    <f:selectItems value="#{test.countryItems}"/>
  </h:selectOneMenu>
  <h:selectOneMenu id="state" value="#{test.state}">
    <f:selectItems value="#{test.stateItems}"/>
  </h:selectOneMenu>
</h:form>
```

7.2.4.2 Status und Fehler von Ajax-Anfragen behandeln

Mit der Funktion`jsf.ajax.addOnEvent(callback)`kann eine Callback-Funktion zum Behandeln des Status von Ajax-Anfragen registriert werden. Die Callback-Funktion bekommt ein Objekt mit genaueren Angaben zum Ereignis übergeben, wobei vor allem die Eigenschaft`status`interessant ist. Sie nimmt die Werte`begin`für das Ereignis, kurz bevor die Anfrage an den Server geschickt wird,`complete`für das Ereignis, nachdem die Antwort vom Server zurückkommt, und`success`für das Ereignis nach erfolgreichem Abschluss der Anfrage an. Die Callback-Funktion wird also dreimal für jede Ajax-Anfrage aufgerufen.
Listing**Callback-Funktion für Ajax-Ereignisse**zeigt eine Funktion, die für jedes Ereignis eine Meldung ausgibt. In Abschnitt**[Sektion: Die Kompositkomponente mc:ajaxStatus]**zeigen wir Ihnen, wie Sie mit einfachsten Mitteln eine Kompositkomponente zum Darstellen einer Ajax-Status-Meldung erstellen.

```
var processEvent = function processEvent(data) {
  if (data.status == "begin") {
    alert('Begin');
  } else if (data.status == "complete") {
    alert('Complete');
  } else if (data.status == "success") {
    alert('Success');
  }
}
jsf.ajax.addOnEvent(processEvent);
```

Eine über`jsf.ajax.addOnEvent()`registrierte Funktion wird bei jeder Ajax-Anfrage aufgerufen.`addOnEvent()`kann auch ohne Probleme mehrfach hintereinander aufgerufen werden, um mehr als eine Funktion zu registrieren. Wenn Sie eine Callback-Funktion für eine bestimmte Ajax-Anfrage benötigen, können Sie diese mit dem Attribut`oneevent`von`f:ajax`oder beim Aufruf von`jsf.ajax.request()`als Option mit dem Schlüssel`oneevent`registrieren.
Mit der Funktion`jsf.ajax.addOnError(callback)`kann eine Callback-Funktion registriert werden, die beim Auftreten eines Fehlers während der Bearbeitung einer Ajax-Anfrage aufgerufen wird. Die Callback-Funktion wird auch hier mit einem Objekt mit Detailinformationen versehen. Die Eigenschaft`status`kann die Werte`httpError`,`serverError`,`malformedXML`oder`emptyResponse`annehmen.
Eine über`jsf.ajax.addOnError()`registrierte Funktion wird im Fehlerfall bei jeder Ajax-Anfrage aufgerufen.`addOnError()`kann auch ohne Probleme mehrfach hintereinander aufgerufen werden, um mehr als eine Funktion zu registrieren. Wenn Sie eine Callback-Funktion für eine bestimmte Ajax-Anfrage benötigen, können Sie diese mit dem Attribut`onerror`von`f:ajax`oder beim Aufruf von`jsf.ajax.request()`als Option mit dem Schlüssel`onerror`registrieren.

7.2.5 Partieller JSF-Lebenszyklus

Eines der entscheidenden Features einer vernünftigen Ajax-Integration ist das partielle Ausführen (*Partial-View-Processing*) und Rendern (*Partial-View-Rendering*) des Komponentenbaums. Als Reaktion auf eine Ajax-Anfrage soll ja im ersten Schritt der Lebenszyklus nicht für den kompletten Komponentenbaum, sondern nur für die in der Anfrage spezifizierten Komponenten ausgeführt werden. Im zweiten Schritt wird als Ergebnis der Anfrage ein weiterer Teil des Komponentenbaums, der nicht mit dem ersten Teil übereinstimmen muss, gerendert.
JSF unterstützt ab Version 2.0 das partielle Ausführen und Rendern direkt mit dem Standardlebenszyklus. Dazu ist der Lebenszyklus, wie in Abbildung**Lebenszyklus mit Ajax**zu erkennen, in die zwei logischen Bereiche*Ausführen*und*Rendern*aufgeteilt.

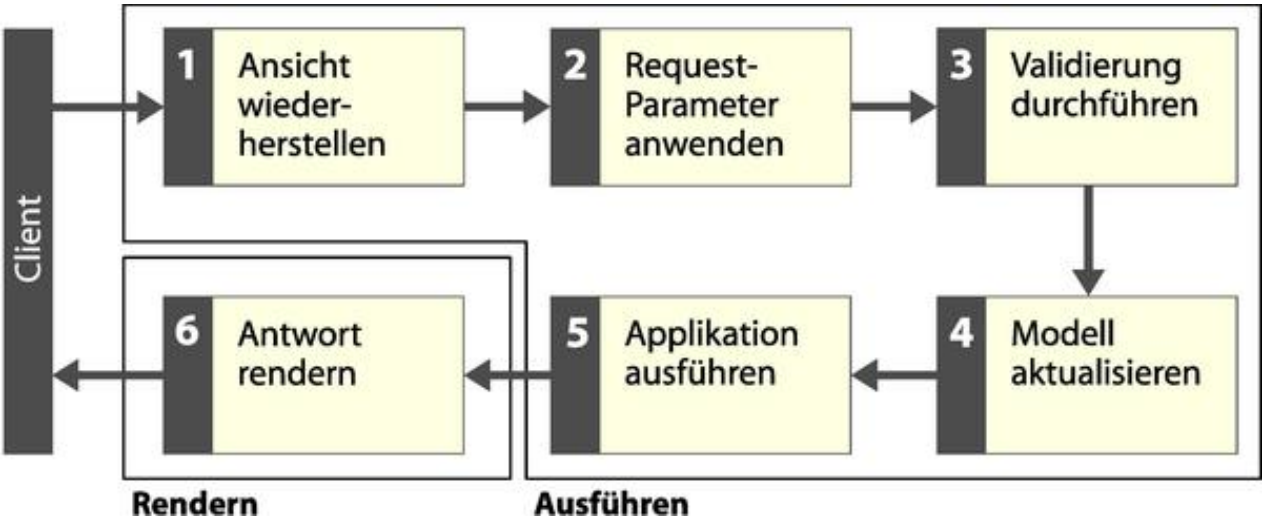


Abbildung:Lebenszyklus mit Ajax

Welche Komponenten in den beiden Bereichen zum Einsatz kommen, wird über die Parameter der Ajax-Anfrage bestimmt. Deren Werte entsprechen den Werten der`f:ajax`-Attribute`execute`und`render`beziehungsweise den gleichnamigen Parametern im assoziativen Array der Funktion`jsf.ajax.request()`.
Tabelletab:ajax-request-paramszeigt die wichtigsten Parameter der Ajax-Anfrage aus dem Einführungsbeispiel in Abschnitt**[Sektion: Ein erstes Beispiel mit f:ajax]**. JSF weiß anhand des Parameters`javax.faces.partial.ajax`, dass es sich um eine Ajax-Anfrage handelt und dass der Lebenszyklus partiell ausgeführt werden muss. Der Parameter`javax.faces.source`gibt an, welche Komponente die Anfrage ausgelöst hat, und die beiden restlichen Parameter bestimmen, welche Komponenten ausgeführt und gerendert werden.

Parameter	Wert
javax.faces.partial.ajax	`true`
javax.faces.source	`form:useCreditCard`
javax.faces.partial.execute	`form:useCreditCard`
javax.faces.partial.render	`form:grid`

Sie müssen sich allerdings keine Gedanken um diese Parameter machen. JSF erledigt das Erstellen und Abschicken der Ajax-Anfrage automatisch im Hintergrund. Sie müssen lediglich spezifizieren, welche Komponenten ausgeführt und gerendert werden sollen - und das auch nur, wenn sich diese Daten von den Defaultwerten unterscheiden.

7.2.6 Ajax-Queue kontrollieren

JSF 2.2:Mit JSF 2.2 ist es möglich, die clientseitige Queue der Ajax-Anfragen zu beeinflussen. Das neue Attribut`delay`von`f:ajax`erlaubt das Verzögern von Ajax-Anfragen um den in Millisekunden angegebenen Wert. Falls während dieser Zeitspanne mehrere Ajax-Anfragen auftreten, wird immer nur die aktuellste verarbeitet. Diese Funktionalität ist wie maßgeschneidert für Ajax-Anfragen, die von Tastaturereignissen ausgelöst werden. Das klassische Einsatzszenario dafür ist das automatische Vervollständigen von Benutzereingaben.
Das Beispiel in Listing**Beispiel für delayvon f:ajax**zeigt eine Eingabekomponente, die bei jedem Tastendruck des Benutzers eine Ajax-Anfrage auslöst. Mit dem Wert 300 für`delay`ist gewährleistet, dass der Server nicht mit Anfragen bombadiert wird, wenn der Benutzer schnell tippt. Treten innerhalb der angegebenen 300 Millisekunden mehrere Ajax-Anfragen auf, wird immer nur die aktuellste verarbeitet.

```
<h:inputText value="#{bean.product}">
  <f:ajax event="keyup" render="result" delay="300"/>
</h:inputText>
<h:panelGroup id="result">
  <ui:repeat value="#{bean.products}" var="p">
    #{p}<br/>
  </ui:repeat>
</h:panelGroup>
```

7.2.7 Eingabefelder zurücksetzen

JSF 2.2:Mit dem neuen Attribut`resetValues`von`f:ajax`lässt sich ab JSF 2.2 ein bereits länger bekanntes

Problem elegant lösen. In einigen Fällen kann JSF während Ajax-Anfragen den Wert von Eingabekomponenten nur dann aktualisieren, wenn er zuvor explizit zurückgesetzt wurde. Dazu muss bei`f:ajax`das neue Attribut`resetValues`auf`true`gesetzt werden. In diesem Fall setzt JSF vor der Render-Phase alle im Attribut`render`angegebenen Eingabekomponenten durch einen Aufruf der Methode`UIViewRoot.resetValues`zurück.
Warum das notwendig sein kann, zeigt das Beispiel in Listing**Beispiel für resetValues**. Der Code sieht auf den ersten Blick einwandfrei aus. Bei näherer Betrachtung zeigt sich aber ein potenzielles Problem.

```
<h:form id="form">
  <h:messages id="msgs"/>
  <h:inputText id="v1" value="#{bean.value1}"/>
  <h:commandLink value="+1" action="#{bean.incValue1}">
    <f:ajax render="v1"/>
  </h:commandLink>
  <h:inputText id="v2" value="#{bean.value2}">
    <f:validateLongRange minimum="10"/>
  </h:inputText>
  <h:commandButton value="Save">
    <f:ajax execute="v1 v2" render="msgs v1 v2"/>
  </h:commandButton>
</h:form>
```

Solange der Benutzer nur den "+1"-Link klickt, funktioniert alles vorbildlich. Während der Ajax-Anfrage wird zuerst in der Action-Methode die Eigenschaft`value1`um eins erhöht und dann das Eingabefeld mit dem neuen Wert gerendert.
Die Probleme beginnen erst, wenn der Benutzer die Schaltfläche zum Speichern mit einem ungültigen Wert für das zweite Eingabefeld betätigt. In diesem Fall schlägt die Validierung für die Komponente mit der IDv2fehl. Nachdem wir aber die ID von`h:messages`vorsorglich in das Attribut`render`mit aufgenommen haben, bekommt der Benutzer wie erwartet eine Fehlermeldung zu sehen.
So weit funktioniert alles wie erwartet, doch jetzt wird bei einem Klick auf den "+1"-Link der aktualisierte Wert nicht mehr angezeigt (obwohl er tatsächlich erhöht wird). Der Grund dafür ist schnell gefunden. Während der Anfrage mit dem ungültigen Wert für`v2`hat JSF in der Process-Validations-Phase für beide Komponenten den Local-Value gesetzt. Wegen des Validierungsfehlers wurde dieser aber nie zurückgesetzt. Nachdem der Local-Value beim Rendern immer Vorrang gegenüber dem Wert aus dem Modell hat, wird der nur im Modell aktualisierte Wert der Eigenschaft`value1`nicht angezeigt.
Genau dieses Problem lässt sich durch das Setzen von`resetValues`vermeiden. Folgendes Beispiel zeigt die einwandfrei funktionierende Variante des "+1"-Links.

```
<h:commandLink value="+1" action="#{bean.incValue1}">
  <f:ajax render="v1" resetValues="true"/>
</h:commandLink>
```

7.3 Ajax in Kompositkomponenten

Im nächsten Beispiel werden wir die in Abschnitt**Sektion: Die Komponente mc:collapsiblePanel**erstellte Kompositkomponente`collapsiblePanel`auf Ajax umstellen. In der aktuellen Variante wird bei jedem Klick auf das Icon zum Ein- und Ausklappen des Inhalts die komplette Seite neu aufgebaut - ein nicht mehr zeitgemäßes Verhalten. Vielmehr soll bei einem Klick auf das Icon nur der Inhalt des Panels angezeigt oder ausgeblendet werden.

Nichts leichter als das. Wie schon im letzten Beispiel lässt sich die Ajax-Unterstützung mit einer einzigen Zeile realisieren. Nachdem wir das von der Komponente`h:commandButton`ausgelöste Übermitteln der Seite auf Ajax umstellen wollen, fügen wir dort das Tag`f:ajax`hinzu. Listing**Ajax in der Kompositkomponente collapsiblePanel**zeigt den Implementierungsteil der Kompositkomponente mit Ajax-Unterstützung.

```
<cc:implementation>
  <h:panelGroup layout="block"
      styleClass="collapsiblePanel-header">
    <h:commandButton id="toggle"
        actionListener="#{cc.toggle}"
        styleClass="collapsiblePanel-img"
        image="#{resource[cc.collapsed ?
           'mygourmet:toggle-plus.png' :
           'mygourmet:toggle-minus.png']}">
      <f:ajax render="@this panel-content"/>
    </h:commandButton>
    <cc:renderFacet name="header"/>
  </h:panelGroup>
  <h:panelGroup id="panel-content" layout="block">
    <h:panelGroup rendered="#{!cc.collapsed}">
      <cc:insertChildren/>
    </h:panelGroup>
  </h:panelGroup>
</cc:implementation>
```

In der Render-Phase des partiellen Lebenszyklus müssen wir in diesem Fall zwei Komponenten neu rendern lassen. Zum einen die Panel-Group mit dem Inhalt der Komponente und zum anderen auch die Schaltfläche selbst, da sich das Icon abhängig vom Einklappzustand ändert. Das gleichzeitige Rendern mehrerer Komponenten stellt JSF erwartungsgemäß vor keine Probleme. Das Attribut`render`nimmt eine beliebige Anzahl mit Leerzeichen getrennter IDs auf, wobei auch hier die Konstante`@this`für die umschließende Komponente steht.
Wie Sie vielleicht schon bemerkt haben, ist im Vergleich zur Nicht-Ajax-Version das Tag`cc:insertChildren`von zwei`h:panelGroup`-Tags umschlossen. Es handelt sich dabei nicht um einen Fehler, sondern um eine notwendige Maßnahme, damit Ajax richtig funktioniert. Die Erklärung ist einleuchtend: Da als Antwort auf die Ajax-Anfrage die Panel-Group mit dem Bezeichner`panel-content`neu gerendert und im Browser ersetzt wird, muss sie dort immer existieren. Mit nur einer Panel-Group würde die gerenderte Ausgabe nach dem ersten Einklappen aus dem DOM verschwinden und könnte nicht mehr eingeblendet werden.
Das`f:ajax`-Tag in Listing**Ajax in der Kompositkomponente collapsiblePanel**ist eine abgekürzte Form des folgenden Tags:

```
<f:ajax event="action" execute="@this"
    render="@this panel-content"/>
```

Wie schon im letzten Beispiel können die Attribute`event`und`execute`auch hier zugunsten der Standardwerte entfallen. Befehlskomponenten definieren`action`als Defaultereignis für`f:ajax`, falls kein anderer Wert angegeben wird.
Doch damit noch nicht genug. Wir gehen noch einen Schritt weiter und stellen auch einen Teil des in die Kompositkomponente eingefügten Inhalts auf Ajax um. In der Ansicht`showCustomer.xhtml`befindet sich die Liste der Adressen eines Kunden in einer Kompositkomponente vom Typ`dataTable`(siehe Abschnitt**Sektion: Die Komponente mc:dataTable**). Die Tabelle ist ihrerseits in eine`collapsiblePanel`-Komponente eingebettet. In der Tabelle gibt es für jede Adresse eine`h:commandLink`-Komponente zum Löschen der Adresse. Ein Klick auf diesen Link soll eine Ajax-Anfrage auslösen und das Neuzeichnen der Tabelle veranlassen.

Listing **Ajax-Kompositkomponente collapsiblePanel im Einsatz** zeigt die relevanten Teile der Seitendeklaration `show-Customer.xhtml` mit dem auf Ajax umgestellten Link.

```
<mc:collapsiblePanel id="addressPanel"
    collapsed="#{customerBean.collapsed}">
  <f:facet name="header">
    <h3>#{msgs.title_addresses}</h3>
  </f:facet>
  <mc:dataTable id="addresses" var="address"
      value="#{customerBean.customer.addresses}">
    ...
    <h:column>
      <h:commandLink action="#{addressBean.edit(address)}"
          value="#{msgs.edit}"/>
      <h:commandLink value="#{msgs.delete}"
          action="#{customerBean.deleteAddress(address)}">
        <f:ajax render=":form:addressPanel:addresses"/>
      </h:commandLink>
    </h:column>
  </mc:dataTable>
</mc:collapsiblePanel>
```

Mit `f:ajax` bekommt die Linkkomponente das Ajax-Verhalten eingeimpft. Wenn der Benutzer den Link im Browser aktiviert, wird eine Ajax-Anfrage an den Server geschickt. JSF führt den Lebenszyklus für die `h:commandLink`-Komponente aus und rendert anschließend die Tabelle neu. Beachten Sie bitte die Form der ID im Attribut `render`: Der führende Doppelpunkt kennzeichnet eine absolute ID. Dieser Schritt wird notwendig, wenn zwischen der auslösenden und der zu rendernden Komponente mehrere Naming-Container liegen.
Dieses Beispiel zeigt sehr schön, wie gut die einzelnen Features in JSF miteinander harmonieren. Die Integration von Ajax in Kompositkomponenten stellt eine einfache Möglichkeit dar, Komponenten mit nach außen transparenter Ajax-Unterstützung zu erstellen.

7.4 Eigene Ajax-Komponenten

In diesem Abschnitt zeigen wir Ihnen anhand von zwei Kompositkomponenten, wie einfach, aber zugleich mächtig die Ajax-Integration von JSF ist. Die Komponenten `ajaxStatus` und `ajaxPoll` nutzen direkt die JavaScript-API, um fortgeschrittene Ajax-Funktionalität anzubieten.

7.4.1 Die Kompositkomponente mc:ajaxStatus

Durch den vermehrten Einsatz von Ajax reduziert sich die Anzahl der Anfragen, die ein komplettes Neuladen der Ansicht nach sich ziehen. Neben den vielen Vorteilen, die Ajax mit sich bringt, gibt es auch einen kleinen Nachteil. Der Browser zeigt keinerlei Statusmeldung an, während eine Ajax-Anfrage bearbeitet wird. Bei jedem kompletten Seitenaufbau sieht der Benutzer einen Fortschrittsbalken oder eine Meldung, die Auskunft über den Status des Seitenaufbaus geben - nicht so bei Ajax-Anfragen. Und obwohl Ajax die Zeit zwischen Anfrage und Antwort erheblich verkürzt, kann es doch zu unklaren Situationen für

den Benutzer kommen. Aus diesem Grund erstellen wir eine Komponente, die eine Statusmeldung anzeigt, solange eine Ajax-Anfrage aktiv ist.
Mit den Informationen über die JavaScript-API aus Abschnitt**[Sektion: JavaScript-API]**ist es ein Leichtes, eine solche Komponente zu erstellen. Wir benötigen im Grunde nichts weiter als eine Statusmeldung und eine JavaScript-Funktion, die mit`jsf.ajax.addOnEvent()`als Callback registriert wird und die Statusmeldung dynamisch ein- und ausblendet. Das packen wir dann gemeinsam in eine Kompositkomponente und fertig ist die Ajax-Status-Komponente. Listing**Kompositkomponente ajaxStatus**zeigt die Deklaration.

```
<cc:interface>
  <cc:attribute name="text" default="Loading"/>
  <cc:attribute name="style"/>
  <cc:attribute name="styleClass"
      default="ajax-progress"/>
</cc:interface>
<cc:implementation>
  <h:outputStylesheet library="mygourmet"
      name="components.css"/>
  <h:outputScript name="jsf.js" library="javax.faces"
      target="head"/>
  <h:outputScript name="ajaxStatus.js"
      library="mygourmet" target="head"/>
  <script type="text/javascript">
    registerAjaxStatus('#{cc.clientId}:msg');
  </script>
  <div id="#{cc.clientId}:msg" class="#{cc.attrs.styleClass}"
      style="display: none;#{cc.attrs.style}">
    #{cc.attrs.text}
  </div>
</cc:implementation>
```

Die Schnittstelle der Komponente ist nicht besonders spannend. Sie besteht nur aus den drei Attributen`text`für die Statusmeldung sowie`style`und`styleClass`, um das Erscheinungsbild an eigene Bedürfnisse anzupassen.
Im Implementierungsteil werden zuerst benötigte Ressourcen geladen. Dazu zählen das Stylesheet`components.css`, die JavaScript-Bibliothek`jsf.js`und das Skript`ajaxStatus.js`mit der Funktionalität der Komponente. Anschließend wird über einen Aufruf der in`ajax-Status.js`definierten Funktion`registerAjaxStatus()`ein Callback für die aktuelle Komponente registriert. Als Parameter bekommt diese Funktion die ID des`div`-Elements mit der Statusmeldung. Schließlich soll dieses Element während einer Ajax-Anfrage ein- und dann wieder ausgeblendet werden. Beachten Sie bei der ID einmal mehr den Zugriff auf`#{cc.clientId}`. Nur so ist gewährleistet, dass auch wirklich die von JSF gerenderte Client-ID zum Einsatz kommt. Das Ein- und Ausblenden ist ganz einfach über die CSS-Eigenschaft`display`realisiert, wobei das Element initial mit`display: none`im Attribut`style`versteckt ist.
Der eigentlich interessante Aspekt der Komponente ist aber der in Listing**JavaScript für Kompositkomponente ajaxStatus**dargestellte JavaScript-Code.

```
function processAjaxUpdate(msgId) {
  function processEvent(data) {
    var msg = document.getElementById(msgId);
    if (data.status == "begin") {
      msg.style.display = '';
    } else if (data.status == "success") {
      msg.style.display = 'none';
    }
  }
  return processEvent;
```

```
};
function registerAjaxStatus(msgId) {
  jsf.ajax.addOnEvent(processAjaxUpdate(msgId));
}
```

Die Funktion`registerAjaxStatus()`registriert mit einem Aufruf von`jsf.ajax.addOnEvent()`eine Callback-Funktion, um das Element mit der übergebenen ID ein- und auszublenden. Es wird allerdings nicht die Funktion`processAjaxUpdate()`registriert, wie es auf den ersten Blick scheinen kann, sondern deren innere Funktion`processEvent()`. Durch diese Schachtelung von Funktionen ist es möglich, von außen die ID des Elements zu übergeben, wohingegen die innere Funktion den Parameter`data`bekommt. Da für die innere Funktion eine*Closure*erstellt wird, geht die übergebene ID nicht verloren und wir haben eine elegante Möglichkeit, auch mehrere unabhängige Ajax-Update-Komponenten in einer Ansicht zu definieren.
Um die Komponente einzusetzen, muss einfach nur das Tag`<mc:ajaxUpdate/>`in die Deklaration eingefügt werden - vorausgesetzt das Präfix`mc`ist entsprechend definiert.

7.4.2 Die Kompositkomponente mc:ajaxPoll

Für manche Anwendungsfälle ist es notwendig, Bereiche einer Seite in periodischen Abständen zu aktualisieren. Klassische Beispiele dafür sind Seiten mit Börsenkursen, Auktionen oder aktuellen Sportergebnissen. JSF stellt diese Funktionalität zwar nicht direkt zur Verfügung, sie lässt sich aber mit wenigen Zeilen JavaScript-Code in einer Kompositkomponente realisieren. Erneut erweist sich die JavaScript-API als äußerst nützlich.
Im Implementierungsteil werden zuerst die benötigten Ressourcen geladen. Dazu zählen die von JSF definierte JavaScript-Bibliothek`jsf.js`und das Skript`ajaxPoll.js`mit der notwendigen Funktionalität für die Komponente. Anschließend wird über einen Aufruf der in`ajaxPoll.js`definierten Funktion`startAjaxPoll()`das Polling gestartet. Als Parameter bekommt diese Funktion die ID des`div`-Elements, das über`cc:insertChildren`mit benutzerdefiniertem Inhalt gefüllt wird, und das Intervall. Beachten Sie bei der ID einmal mehr den Zugriff auf`#{cc.clientId}`, um die korrekte Client-ID zu erhalten. Listing**Kompositkomponente ajaxPoll**zeigt die Deklaration.

```
<cc:interface>
  <cc:attribute name="interval" required="true"/>
</cc:interface>
<cc:implementation>
  <h:outputScript name="jsf.js" library="javax.faces"
      target="head"/>
  <h:outputScript name="ajaxPoll.js"
      library="mygourmet" target="head"/>
  <script type="text/javascript">
  startAjaxPoll('#{cc.clientId}',#{cc.attrs.interval});
  </script>
  <div id="#{cc.clientId}">
    <cc:insertChildren/>
  </div>
</cc:implementation>
```

Die gesamte Ajax-Funktionalität der Komponente steckt in der Skript-Ressource`ajaxPoll.js`. Der Code ist in Listing**JavaScript für Kompositkomponente ajaxPoll**zu sehen.

```
function processPollEvent(interval) {
```

```
    return function(data) {
      if (data.status == 'success') {
        startAjaxPoll(data.source.id, interval);
      }
    };
  }
  function poll(clientId, interval) {
    var element = document.getElementById(clientId);
    element.mgPoll = true;
    jsf.ajax.request(element, null, {render: clientId,
        onevent: processPollEvent(interval)});
  }
  function startAjaxPoll(clientId, interval) {
    setTimeout("poll('"+clientId+"', "+interval+")", interval);
  }
```

Als Einstiegspunkt dient die Funktion`startAjaxPoll()`, die als Parameter die ID des zu aktualisierenden DOM-Elements und das Intervall zwischen den Ajax-Anfragen bekommt. Die Ajax-Anfrage selbst wird in der Funktion`poll()`abgesetzt, die in`startAjaxPoll()`über die JavaScript-Funktion`setTimeout()`nach den im Intervall angegebenen Millisekunden aufgerufen wird. Die Funktion`poll()`sucht zuerst das DOM-Element für die übergebene Client-ID und schickt dann die Abfrage mit`jsf.ajax.request()`ab. Der Clou an der Sache ist, dass über eine Callback-Funktion nach erfolgreichem Abschluss der Anfrage über`startAjaxPoll()`der gesamte Prozess nochmals gestartet wird. Dadurch entsteht der Polling-Effekt und es ist gewährleistet, dass im Fall eines Fehlers keine weiteren Anfragen an den Server abgeschickt werden.
Einen kleinen Schönheitsfehler hat die Komponente noch. Wenn gleichzeitig`ajaxStatus`zum Einsatz kommt, wird bei jedem Update die Statusmeldung angezeigt - ein Verhalten, das nicht immer angebracht ist. Doch auch dafür gibt es eine sehr einfache Lösung. Wie Sie vielleicht schon bemerkt haben, wird in`poll()`dynamisch die Eigenschaft`mgPoll`auf das DOM-Element gesetzt. Diese Eigenschaft kann dann in der Callback-Funktion von`ajaxStatus`mit`data.source.mgPoll`abgefragt werden. Ist sie gesetzt, wird die Meldung nicht eingeblendet.
Um die Komponente einzusetzen, muss nur das Tag`mc:ajaxPoll`mit dem gewünschten Inhalt in die Deklaration eingefügt werden - vorausgesetzt das Präfix`mc`ist entsprechend definiert. Hier ein Beispiel, mit dem eine Art Uhr in die Ansicht eingebaut wird:

```
<mc:ajaxPoll interval="950">#{customerBean.time}</mc:ajaxPoll>
```

7.5 MyGourmet 14: Ajax

*MyGourmet 14*ist vom Funktionsumfang her identisch mit*MyGourmet 13*. Der große Unterschied liegt in der Integration von Ajax, wodurch an einigen Stellen ein komplettes Neuladen der Ansicht vermieden wird. Die Anwendung wirkt dadurch insgesamt flüssiger. Abbildung**MyGourmet 14 mit Ajax-Status**zeigt einen Screenshot der Ansicht`showCustomer.xhtml`mit aktivierter Ajax-Status-Meldung.

Abbildung:MyGourmet 14 mit Ajax-Status

Damit Benutzer immer über den Status einer Ajax-Anfrage informiert sind, fügen wir die Kompositkomponente`ajaxStatus`direkt im Template`customerTemplate.xhtml`in den Header ein. Listing**Die Komponente ajaxStatus im Einsatz**zeigt den aktualisierten Header-Bereich.

```
<ui:define name="header">
  <h:graphicImage name="images/logo.png"/>
  <h1>#{msgs.title_main}</h1>
  <mc:ajaxStatus style="float:right; width: 100px;"/>
</ui:define>
```

Nachdem die Ajax-Anfragen in*MyGourmet*recht zügig bearbeitet werden, blitzt die Statusmeldung nur kurz auf. Wenn Sie sie in ihrer vollen Pracht betrachten wollen, können Sie zum Beispiel mit Firebug einen Breakpoint in das Skript setzen.
Die Klasse`DebugPhaseListener`hat ebenfalls eine kleine Erweiterung bekommen. Die Logmeldungen zu Beginn und am Ende jeder Phase geben jetzt Auskunft darüber, ob es sich bei der Anfrage um eine Ajax-Anfrage handelt oder nicht. Um das festzustellen, rufen wir die Methode`isAjaxRequest()`des Partial-View-Contexts auf, der über den Faces-Context erreichbar ist. Listing**Erweiterte Version des Debug-Phase-Listeners mit Ajax-Support**zeigt den Sourcecode.

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

  public void afterPhase(PhaseEvent ev) {
    String ajax = getAjaxText(ev.getFacesContext());
    PhaseId phaseId = ev.getPhaseId();
    log.debug("After phase: " + phaseId + ajax);
  }
  public void beforePhase(PhaseEvent ev) {
    String ajax = getAjaxText(ev.getFacesContext());
    PhaseId phaseId = ev.getPhaseId();
```

```
    log.debug("Before phase: " + phaseId + ajax);
  }
  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
  private String getAjaxText(FacesContext ctx) {
    return ctx.getPartialViewContext()
        .isAjaxRequest() ? " (Ajax)" : "";
  }
}
```

Die restlichen Änderungen umfassen nur die im Laufe des Kapitels mit Ajax erweiterten Ansichten der Anwendung. Dazu zählt neben dem dynamischen Ein- und Ausblenden der Kreditkartendaten in`editCustomer.xhtml`auch das`collapsiblePanel`mit den Adressen in der Seite`showCustomer.xhtml`. Dort gibt es ja sogar die doppelte Ajax-Funktionalität: In der Kompositkomponente selbst und in der darin eingebetteten Tabelle zum Löschen einer Adresse.
In Abschnitt**[Sektion: Ein erstes Beispiel mit f:ajax]**haben wir bereits eine Lösung zum dynamischen Ein- und Ausblenden der Kreditkartendaten mittels`f:ajax`gezeigt. Die dort präsentierte Umsetzung hat allerdings noch einen kleinen Schönheitsfehler: Wenn im eingeblendeten Zustand die Kreditkartendaten geändert, ausgeblendet und wieder eingeblendet werden, gehen die Änderungen verloren. Ein genauerer Blick auf das eingesetzte`f:ajax`-Tag zeigt den Grund. Nachdem wir das Attribut`execute`nicht angegeben haben (und somit der Standardwert`@this`zum Einsatz kommt), bearbeitet JSF bei der partiellen Ausführung des Lebenszyklus nur das Auswahlfeld. Die geänderten Daten kommen nie am Server an und werden durch das erneute Rendern im Browser mit den alten Werten überschrieben.

7.6 Werkzeuge für den Ajax-Entwickler

Ajax ist durch die Vielzahl an verwendeten Basistechnologien nicht gerade einfach einzusetzen. Will man Ajax-Anwendungen bauen, ist eine Unterstützung durch geeignete Entwicklungswerkzeuge unabdingbar. Die folgende Aufzählung von Tools erhebt keinen Anspruch auf Vollständigkeit. Wir wollen Ihnen eine Reihe von Werkzeugen präsentieren, die sich in der täglichen Arbeit mit JSF-Projekten bewährt haben.

7.6.1 Firebug

*Firebug*Firebugist unter**http://getfirebug.com**und als Add-on erhältlich.:ist eine Erweiterung von*Firefox*, die eine ganze Reihe sehr nützlicher Werkzeuge für Webentwickler direkt in den Browser integriert.*Firebug*ist frei verfügbar und erweist sich in vielen Situationen als unbezahlbar. Besonders dann, wenn es darum geht, den DOM-Baum zu analysieren oder JavaScript-Code zu debuggen. Hier eine Liste der wichtigsten Features:

- Inspizieren und Editieren von HTML-Code
- Inspizieren und Editieren von CSS-Regeln
- Debuggen und Profiling von JavaScript-Code
- Analysieren des DOM
- Analysieren des HTTP-Verkehrs

Der Funktionsumfang und die Benutzerfreundlichkeit verbessern sich mit jeder neuen Version. Mittlerweile gibt es sogar eine Liste von sehr nützlichen Erweiterungen für*Firebug*selbst. Dazu zählt zum Beispiel*YSlow*zur Überprüfung von HTML-Seiten auf Performance-Probleme.

7.6.1.1 Analyse des DOM-Baums

Versucht man die Aktualisierung durch die Ajax-Antwort mit einem Editor nachzuvollziehen, wird keine Änderung des HTML-Codes sichtbar sein. Die Webseite wurde nicht komplett neu geladen, sondern nur der DOM-Baum aktualisiert. Diesen DOM-Baum rendert der Browser und stellt ihn grafisch dar. Die Veränderung kann nur durch die Visualisierung des Baums selbst sichtbar gemacht werden.*Firebug*bietet die visuelle Darstellung des Baums der HTML-Elemente in Form einer Verzeichnisstruktur. Abbildung**Firebug mit aktualisierter Komponente**zeigt die Ansicht`showCustomer.xhtml`mit eingeklapptem Adressen-Panel und aktiviertem*Firebug*.

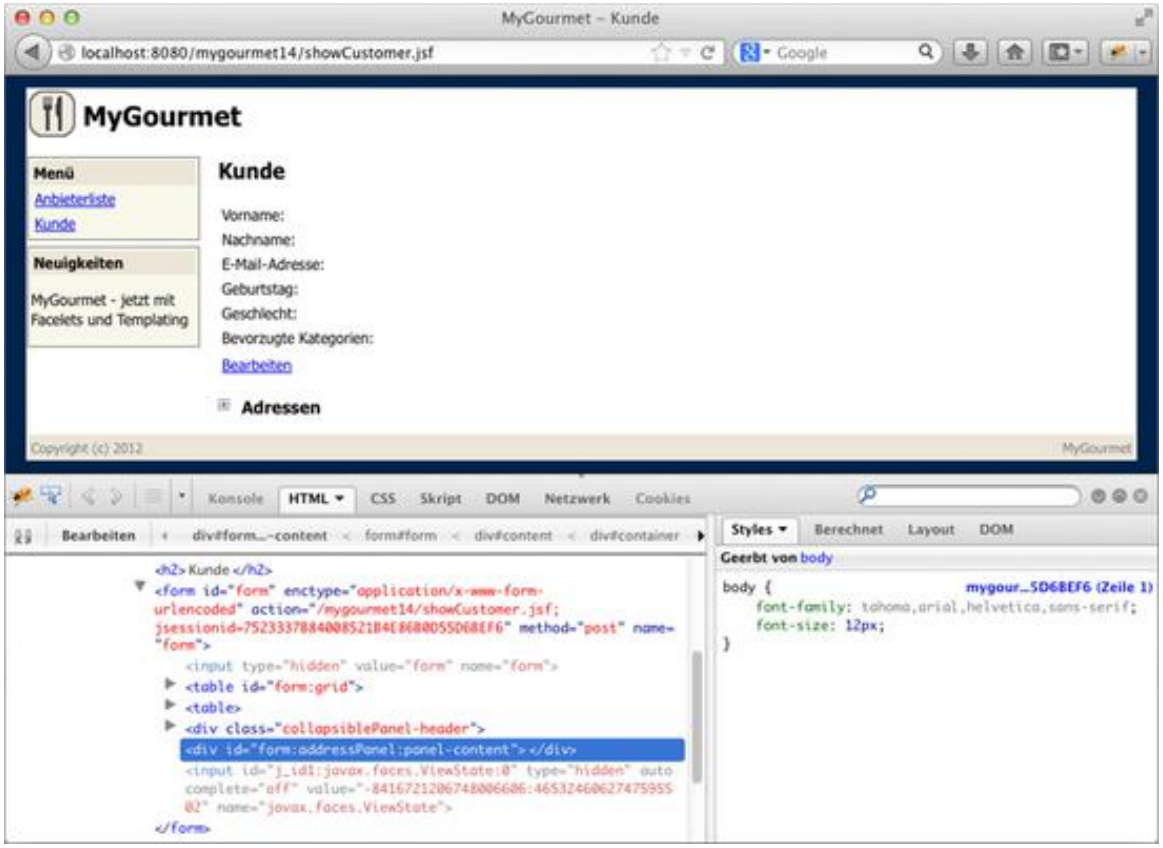


Abbildung:Firebug mit aktualisierter Komponente

Es ist auch möglich, Knoten über deren ID, Tag-Name oder Attribute zu suchen, diese im Browser herausheben zu lassen oder deren Eigenschaften zu ändern.*Firebug*zeigt aber nicht nur den aktuellen HTML-Code, sondern hebt sogar Elemente farblich hervor, die sich gerade geändert haben.
Im unteren Panel in Abbildung**Firebug mit aktualisierter Komponente**ist der mit der Webseite synchron gehaltene DOM-Baum zu sehen. Das hervorgehobene Element ist der`div`-Block des Panels, in dem die Liste der Adressen angezeigt wird. Bei einer initialen Anfrage auf die Seite ist das Panel immer ausgeklappt. Der in der Abbildung erkennbare, eingeklappte Zustand ist erst als Reaktion auf die Ajax-Anfrage nach dem Klick auf das Icon eingetreten.
Sobald man in der Baumansicht ein Element anwählt, wird dieses im Browser farblich hervorgehoben. Es ist auch möglich, direkt aus dem Browserfenster ein Element anzeigen zu lassen. Auf der rechten Seite sind zusätzliche Informationen über die Knoten, wie Name,`ID`oder die Eigenschaften, abrufbar. Es gibt auch die Möglichkeit, in eine JavaScript-Objektansicht umzuschalten. Wie in Abbildung**Ansicht mit Attributen und Methoden in Firebug**zu sehen, kann in sämtliche Methoden und Attribute eines JavaScript-Objektes Einsicht genommen werden.

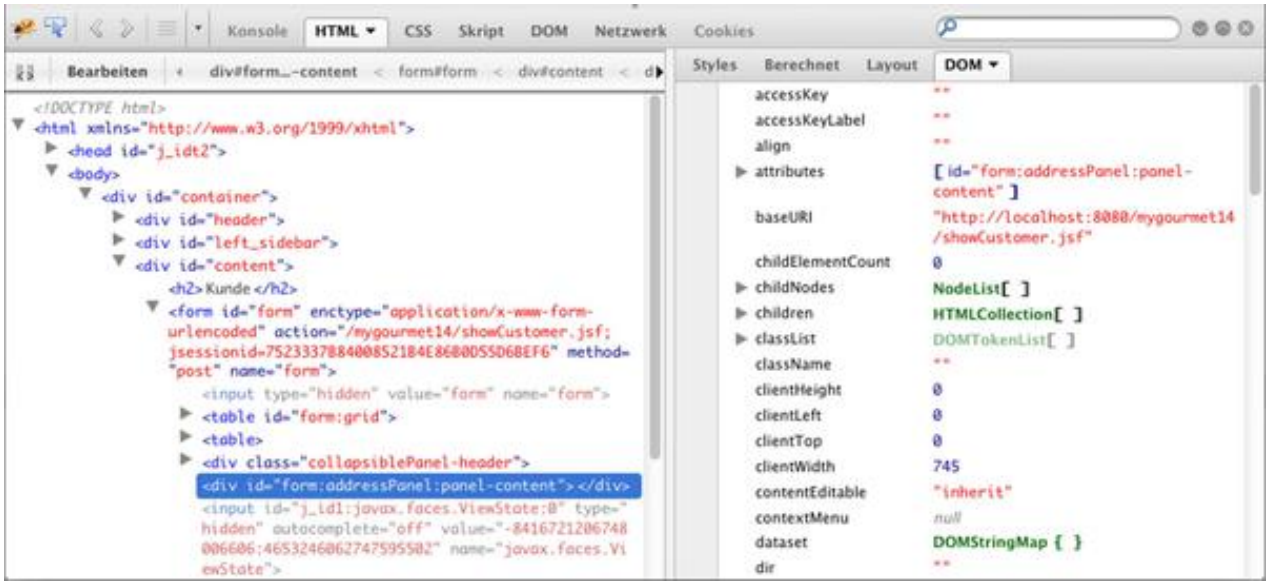


Abbildung:Ansicht mit Attributen und Methoden in Firebug

Auch wenn Sie den Internet Explorer einsetzen, müssen Sie nicht auf einen DOM-Inspektor verzichten. Ab Version 8 ist ein entsprechendes Werkzeug sogar bereits im Standardlieferumfang enthalten. Für ältere Versionen empfiehlt sich die*Internet Explorer Developer Toolbar*Die*Internet Explorer Developer Toolbar*ist unter der Adresse erhältlich.:.

7.6.1.2 JavaScript-Debugging

Ein unersetzliches Werkzeug für Ajax-Entwickler ist ein JavaScript-Debugger, wie ihn zum Beispiel*Firebug*für*Firefox*bietet. Mit diesem kann zur Laufzeit JavaScript-Code der Webseite oder der Ajax-Bibliothek in einzelnen Schritten durchlaufen werden. So wird komplexer Code leichter verständlich, Objekte, ihre Daten und Methoden erkennbar und fehlerhaftes Verhalten einfach ermittelbar. Wie in herkömmlichen Debuggern ist es möglich, Breakpoints zu setzen und sich an dieser Stelle der Ajax-Applikation den Zustand von Variablen und Objekten anzusehen und Funktionen aufzurufen. Ein Praxisbeispiel ist in Abbildung**JavaScript-Debugging in Firebug**zu sehen.

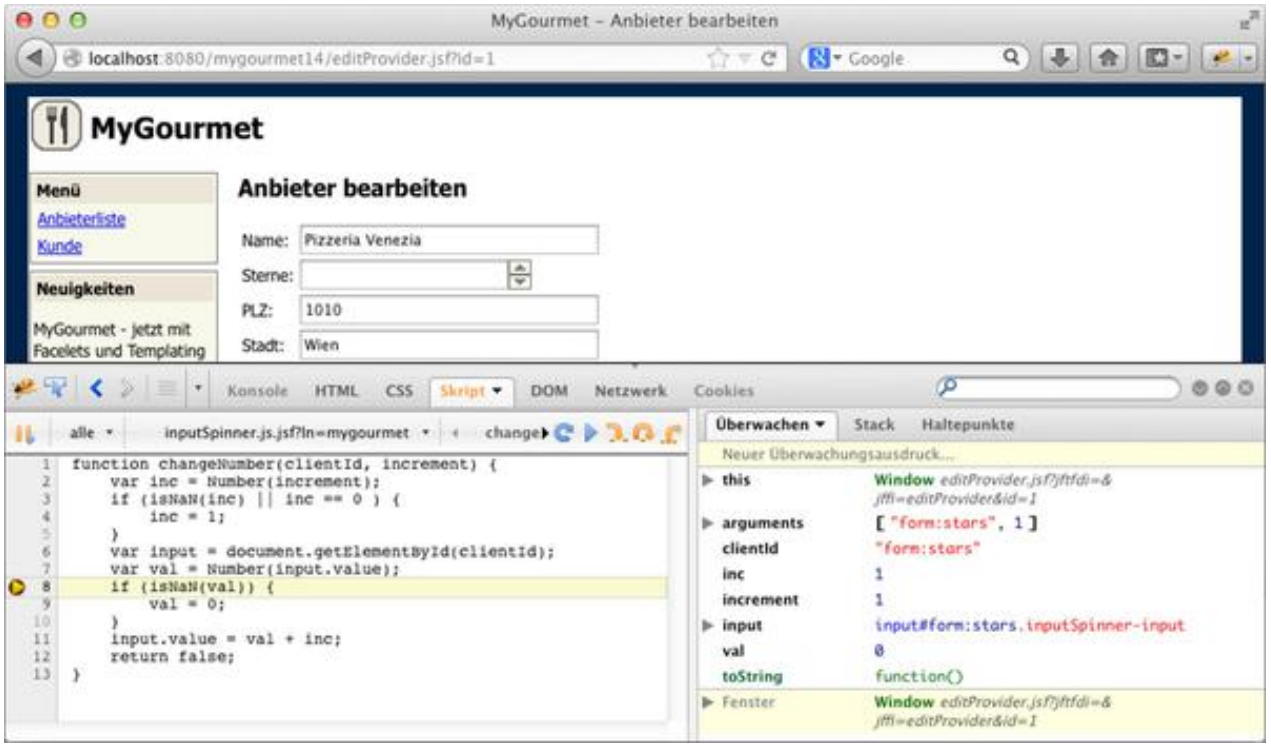


Abbildung:JavaScript-Debugging in Firebug

Hier wurde ein Breakpoint in das Skript`inputSpinner.js`gesetzt. Sobald der Benutzer auf eines der Bilder zum Erhöhen oder Reduzieren des Werts klickt, springt das Fenster des Debuggers auf und die Abarbeitung des Codes hält genau an dieser Stelle an. Im Fenster*Überwachen*kann jeder erdenkliche

JavaScript-Befehl eingegeben werden. Hier zum Beispiel der Code, um das Input-Spinner-Eingabefeld anzuzeigen: <

```
document.getElementById('form:stars')
```

Der Internet Explorer verfügt ab Version 8 bereits im Standardlieferumfang über ein recht brauchbares Entwicklerwerkzeug mit DOM-Browser und JavaScript-Debugger. Für ältere Versionen ist der*Microsoft Script DebuggerMicrosoft Script Debugger*ist unter**http://www.microsoft.com/downloads**erhältlich.:als Freeware erhältlich.

7.6.2 HTTP-Debugger

Manchmal kann es recht nützlich sein, sich den Protokollablauf ein wenig näher anzusehen. Mit einem HTTP-Debugger ist es möglich, den kompletten HTTP-Verkehr mitzuschneiden. Eines der bekanntesten Werkzeuge in diesem Bereich ist der frei verfügbare HTTP-Debugger*Fiddler*(Windows)*Fiddler*ist unter**http://fiddler2.com**erhältlich.:. Die Zusammenarbeit mit dem Browser wird über das Programm selbst geregelt - es stellt einen lokalen Proxy dar. In muss der Proxy unter den Verbindungseinstellungen mit den Daten`127.0.0.1`und Port`8888`eingetragen werden. Im Internet Explorer sind keine weiteren Einstellungen zu treffen, denn die Zwischenstelle wird automatisch erkannt.*Fiddler*kann so konfiguriert werden, dass das Programm die Abarbeitung vor einer HTTP-Anfrage oder nach einer HTTP-Antwort - ähnlich einem gewöhnlichen Debugger - anhält und so der aktuelle Stand der Interaktion zu sehen ist. Dieser wird im linken Fenster von*Fiddler*protokolliert, wie es Abbildung**Der HTTP-Debugger Fiddler protokolliert den HTTP-Verkehr**zeigt. In Version 2 ist es sogar möglich, HTTPS-Verbindungen zu debuggen.

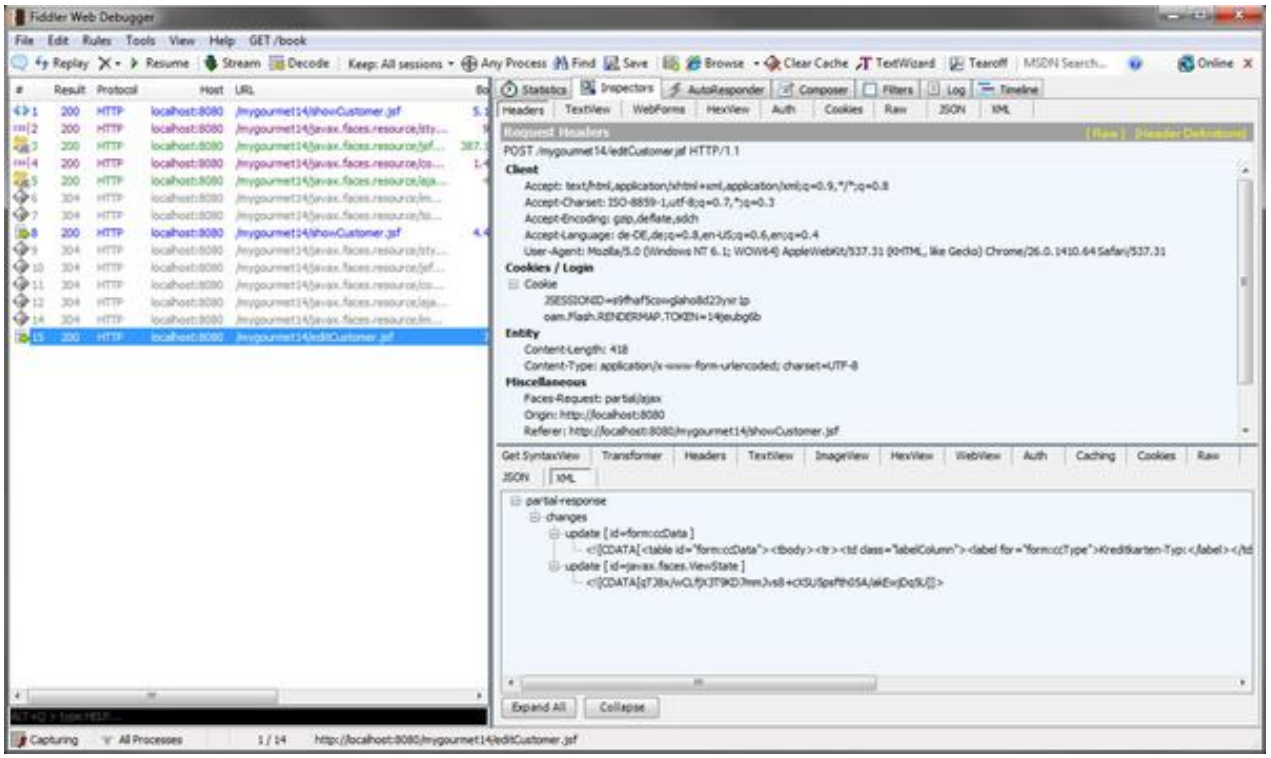


Abbildung:Der HTTP-Debugger Fiddler protokolliert den HTTP-Verkehr

Die angezeigten Daten sind der Stand auf eine Ajax-Anfrage nach der HTTP-Antwort. Im rechten oberen Fenster wird die übliche Protokollinformation der vorhergehenden Anfrage angezeigt. In unserem Beispiel handelt es sich um eine POST-Anfrage mit der Protokollversion 1.1 in einer bestimmten HTTP-Sitzung. Das Fenster rechts unten zeigt die Nutzdaten der Antwort an. Zu sehen ist die Liste, die vomXML-`HttpRequest`-Objekt aufgenommen und in weiterer Folge in den DOM-Baum eingefügt wird.
Für Firefox bietet*Firebug*mittlerweile auch eine sehr brauchbare Übersicht aller abgesetzten HTTP-Requests und ermöglicht das einfache Umschalten zwischen "vollen" und Ajax-Requests.

7.6.3 Web Developer Toolbar

Ein für den Webentwickler unersetzliches Werkzeug für Firefox und Chrome ist das Add-on*Web DeveloperWeb Developer*ist als Add-on zu Firefox und Chrome und unter der Adresse**http://chrispederick.com/work/web-developer/**erhältlich.:. Grob umrissen bietet dieses Add-on folgende Funktionen an:

- Grafische Auszeichnung von einzelnen HTML-Elementen
- Validierung von HTML, CSS und anderen Technologien
- Formatierte Anzeige des Quellcodes
- Frei definierbare Größe des Browserfensters
- Visuelle Ausgabe von HTML-Attributen
- Manipulation von Formularen
- Anzeige der CSS-Stile einzelner HTML-Elemente
- Editieren von CSS und HTML und Anzeigen der Änderungen in Echtzeit

Die wichtigste Funktion der Erweiterung ist die Möglichkeit der Editierung von CSS und die sofortige Anzeige der Änderungen. Das Schreiben und mühsame ständige Umschalten in den Browser kann so vermieden werden. Der Entwickler kann sofort mitverfolgen, welche Auswirkungen einzelne CSS-Befehle haben.

8 JSF und HTML5

HTML5 ist zurzeit in aller Munde und sicher eines der am häufigsten verwendeten und missbrauchten Modewörter im Bereich der Webentwicklung. Aber hinter dem Hype verbergen sich eine ganze Reihe nützlicher Features und Erweiterungen, die das Leben von Webentwicklern einfacher (und aufregender) machen. Höchste Zeit für JSF auf diesen rasenden Zug aufzuspringen.
Es hat zwar bereits in der Vergangenheit einige Ansätze zur Integration von HTML5 in JSF gegeben, die offizielle Variante ist aber erst seit JSF 2.2 Teil der Spezifikation. Unter dem Oberbegriff "HTML5 Friendly Markup" versammelt JSF zwei grundlegend neue Konzepte: Pass-Through-Attribute und Pass-Through-Elemente.
Beginnen werden wir dieses Kapitel allerdings mit einem kur-zen Abstecher zu den verschiedenen Verarbeitungsmodi für Facelets-Dateien in Abschnitt**[Sektion: Verarbeitungsmodi für Facelets-Dateien]**. Danach zeigen wir Ihnen in Abschnitt**[Sektion: HTML5 Pass-Through-Attribute]**aber auch schon die Pass-Through-Attribute und in Abschnitt**[Sektion: HTML5 Pass-Through-Elemente]**die Pass-Through-Elemente. Abschnitt**[Sektion: *MyGourmet 15*: HTML5]**beschreibt abschließend noch eine Variante von*MyGourmet*mit den zuvor vorgestellten Konzepten.

8.1 Verarbeitungsmodi für Facelets-Dateien

Als erste und einfachste Maßnahme zur Unterstützung von HTML5 verarbeitet JSF 2.2 Facelets-Dateien etwas anders als JSF 2.1. Im Vergleich zur Vorgängerversion rendert JSF 2.2 immer den HTML5-Doctype`<!DOCTYPE html>`-unabhängig davon, welcher Doctype in der XHTML-Datei angegeben ist.
Dazu gibt es seit JSF 2.1 die Möglichkeit, Facelets-Dateien in verschiedenen Modi zu verarbeiten. In JSF 2.2 ist das standardmäßig der Modus`html5`. Wenn Sie den HTML5-Doctype nicht verwenden wollen, können Sie den Verarbeitungsmodus in der`faces-config.xml`wie in Listing**Modus xhtml für Facelets-Dateien**auf den Modus`xhtml`setzen (Standardmodus in JSF 2.1).

```
<faces-config-extension>
  <facelets-processing>
    <file-extension>.xhtml</file-extension>
    <process-as>xhtml</process-as>
  </facelets-processing>
</faces-config-extension>
```

Zusätzlich definiert JSF noch die Modi`xml`zur Verarbeitung von Facelets-Dateien im XML-Modus (XML-Deklaration, Doctype und Kommentare werden entfernt) und`jspx`zum Umstieg von JSP.

8.2 HTML5 Pass-Through-Attribute

HTML5 definiert eine ganze Reihe neuer Attribute und Attributwerte für bereits existierende HTML-Elemente. JSF 2.2 liefert mit den sogenannten Pass-Through-Attributen das passende Konzept zur Unterstützung dieser Attribute. Das Prinzip ist einfach: Jede Komponente kann neben den "normalen" Attributen noch eine beliebige Anzahl von Pass-Through-Attributen haben. Diese Attribute werden unverändert an das von der Komponente gerenderte Element weitergereicht und von der Komponente selbst nicht weiterverarbeitet.
Der Vorteil dieses Ansatzes liegt klar auf der Hand. JSF kann damit neue und geänderte HTML5-Attribute unterstützen,

ohne die existierenden Standardkomponenten zu ändern. Die JSF-Spezifikation ist somit auch nicht von der HTML5-Spezifikation abhängig. Ein wichtiger Punkt - besonders wenn man bedenkt, dass die Arbeit an HTML5 noch nicht abgeschlossen ist.
JSF definiert mehrere Varianten zur Definition von Pass-Through-Attributen. Die einfachste besteht darin, das Attribut im Namensraum`http://xmlns.jcp.org/jsf/passthrough`zu definieren. Dazu muss das Attribut lediglich mit dem Präfix des Namensraums versehen werden. Das Beispiel in Listing**Beispiel mit Pass-Through-Attributen**zeigt eine`h:inputText`-Komponente mit den beiden regulären Attributen`id`und`value`und den Pass-Through-Attributen`type`und`placeholder`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">
<h:head><title>JSF 2.2 HTML5</title></h:head>
<h:body>
  <h:form id="form">
    <h:inputText id="email" value="#{bean.email}"
        pt:type="email" pt:placeholder="E-Mail eingeben"/>
    <h:commandButton action="#{bean.save}" value="Save"/>
  </h:form>
</h:body>
</html>
```

Bei den Pass-Through-Attributen handelt es sich um HTML5-Erweiterungen für das Element`input`, die von`h:inputText`nicht direkt unterstützt werden. Mit`placeholder`kann ein Text angegeben werden, der im Browser angezeigt wird, solange der Benutzer noch nichts eingegeben hat. Im Attribut`type`kommt mit`email`einer der neuen Typen für Eingabefelder zum Einsatz. Der gerenderte HTML-Code für das`h:inputText`-Tag beinhaltet wie erwartet alle vier Attribute:

```
<input id="form:email" name="form:email" value=""
    placeholder="E-Mail eingeben" type="email"/>
```

Tabelletab:html5-input-exzeigt einige Beispiele, wie das`input`-Element mit den zusätzlichen HTML5-Attributen im Browser dargestellt wird. Die Validierung der Mailadresse wird dabei vom Browser clientseitig beim Submit des Formulars durchgeführt - JSF bekommt davon nichts mit. Aber Achtung: Nicht alle HTML5-Features werden von allen Browsern gleichermaßen unterstütztEine aktuelle Übersicht, welche Browserversionen welche HTML5-Features unterstützen, finden Sie zum Beispiel unter**http://caniuse.com**.:.

Anwendungsfall	Beispiel
Leeres Eingabefeld	
Ungültige E-Email (Chrome 28)	
Ungültige E-Mail (Firefox 23)	

Neben der Variante mit dem Namensraum können Sie Pass-Through-Attribute auch mit dem Tag`f:passThroughAttribute`definieren. Der Name des Pass-Through-Attributs wird dabei in`name`und der Wert in`value`angegeben, wie Listing**Pass-Through-Attribute über Tags**zeigt. Die gerenderte Ausgabe ändert sich dadurch nicht.

```
<h:inputText id="email" value="#{bean.email}">
  <f:passThroughAttribute name="type" value="email"/>
  <f:passThroughAttribute name="placeholder"
      value="E-Mail eingeben"/>
</h:inputText>
```

Wenn Ihnen diese beiden Varianten noch nicht reichen, können Sie Pass-Through-Attribute mit dem Tag`f:passThroughAttributes`auch aus einer Bean-Eigenschaft mit dem Typ`Map<String, Object>`holen, wie Listing**Pass-Through-Attribute aus Bean-Eigenschaft**zeigt.

```
<h:inputText id="email" value="#{bean.email}">
  <f:passThroughAttributes value="#{bean.attributes}"/>
</h:inputText>
```

Listing**Bean-Eigenschaft für Pass-Through-Attribute**zeigt die zu Listing**Pass-Through-Attribute aus Bean-Eigenschaft**passende Bean-Eigenschaft.

```
public Map<String, Object> getAttributes() {
  Map<String, Object> attrs = new HashMap<String, Object>();
  attrs.put("placeholder", "E-Mail eingeben");
  attrs.put("type", "email");
  return attrs;
}
```

Pass-Through-Attribute ermöglichen auch den Einsatz von HTML5-Custom-Data-AttributenDetails zu HTML5-Custom-Data-Attributen finden Sie zum Beispiel unter**http://html5doctor.com/html5-custom-data-attributes**.:. Diese Attribute sind dazu gedacht, HTML-Elemente mit zusätzlichen Daten zu versehen. Man erkennt sie sofort an ihrem Namen, der immer mit dem Präfix`data-`beginnt. Custom-Data-Attribute werden im DOM abgelegt und stehen dort für die Verarbeitung mit JavaScript zur Verfügung.
Folgendes Beispiel zeigt ein`h:inputText`-Tag mit dem Custom-Data-Attribut`data-required`in Form eines Pass-Through-Attributs.

```
<h:inputText id="email" value="#{bean.email}"
    pt:type="email" pt:data-required="true"/>
```

Dieses Attribut hat keine spezielle Bedeutung und wird weder von JSF noch vom Browser gesondert behandelt. Es kann allerdings im JavaScript-Code der Applikation beispielsweise zur clientseitigen Validierung verwendet werden.

8.3 HTML5 Pass-Through-Elemente

Traditionell bauen JSF-Entwickler die Seiten einer Anwendung überwiegend mit JSF-Tags auf und kommen nur indirekt mit den gerenderten HTML-Elementen in Kontakt. Mit dem vermehrten Einsatz von clientseitigen Technologien wie JavaScript ist es allerdings immer öfter von Bedeutung, die Struktur der Seite im Detail zu kennen. JSF 2.2 bietet mit den sogenannten Pass-Through-Elementen eine Lösung für dieses Problem an.
Bei Pass-Through-Elementen handelt es sich um HTML-Elemente, die mit passenden JSF-Komponenten im Komponentenbaum verbunden sind. Ein HTML-Element wird per Definition genau dann zu einem Pass-Through-Element, wenn mindestens eines seiner Attribute im Namensraum`http://xmlns.jcp.org/jsf`definiert ist.
Listing**Beispiel mit Pass-Through-Elementen**zeigt nochmals das Beispiel aus Listing**Beispiel mit Pass-Through-Attributen**. Bei dieser Version wurden allerdings alle JSF-Tags durch Pass-Through-Elemente ersetzt. An der Funktionalität ändert sich dadurch nichts und selbst der Komponentenbaum am Server hat sich nicht entscheidend verändert.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:jsf="http://xmlns.jcp.org/jsf">
<head jsf:id="head"><title>JSF 2.2 HTML5</title></head>
<body jsf:id="body">
  <form jsf:id="form">
    <input type="text" placeholder="E-Mail eingeben"
```

```
        jsf:id="email" jsf:value="#{bean.email}"/>
      <button jsf:action="#{bean.save}">Save</button>
    </form>
  </body>
  </html>
```

Wie funktioniert das? Wenn mindestens ein Attribut eines HTML-Elements im Namensraum`http://xmlns.jcp.org/jsf`definiert ist, transformiert JSF dieses HTML-Element in ein JSF-Tag und fügt die entsprechende Komponente in den Komponentenbaum ein. Die Entscheidung, welches JSF-Tag verwendet wird, fällt JSF anhand des Elementnamens. Ergibt sich daraus keine eindeutige Zuordnung, wird in einigen Fällen zusätzlich der Wert eines Attributs in die Entscheidung mit einbezogen.
Sehen wir uns das anhand des Beispiels aus Listing**Beispiel mit Pass-Through-Elementen**etwas genauer an. Da das Element`head`mit`id`ein Attribut im Namensraum`http://xmlns.jcp.org/jsf`hat, wird es als Pass-Through-Element behandelt. JSF transformiert das Element`head`daher in das JSF-Tag`h:head`und setzt dessen Attribut`id`auf den angegebenen Wert. Analog wird das Element`body`in`h:body`und`form`in`h:form`transformiert.
Das Element`input`lässt sich nur unter Berücksichtigung des Attributs`type`eindeutig transformieren. Aus der Kombination`input`und`type="text"`macht JSF das Tag`h:inputText`. Werfen wir nochmal einen Blick auf die Attribute von`input`. Sowohl`id`als auch`value`haben ein Präfix und werden somit direkt an die Komponente weitergereicht. Nachdem das HTML5-Attribut`placeholder`aber kein Präfix besitzt, wird es nicht als normales Attribut, sondern als Pass-Through-Attribut zu`h:inputText`hinzugefügt. Anders wäre es auch nicht möglich, dah`:inputText`kein Attribut mit dem Namen`placeholder`definiert. Anhand dieses Beispiels lässt sich schön erkennen, wie sich Pass-Through-Elemente und Pass-Through-Attribute gegenseitig ergänzen.
Tabelletab:html5-pte-mappingzeigt eine komplette Liste der in JSF 2.2 definierten Zuordnungen von Element-Attribut-Kombinationen zu JSF-Tags. Attribute mit dem Präfix`jsf`in der zweiten Spalte bezeichnen Attribute im Namensraum`http://xmlns.jcp.org/jsf`. Ein Stern wiederum bedeutet, dass der Wert des Attributs für die Zuordnung nicht relevant ist.

HTML-Element	Selektor-Attribut	JSF-Tag
a	jsf:action	h:commandLink
a	jsf:actionListener	h:commandLink
a	jsf:value	h:outputLink
a	jsf:outcome	h:link
body		h:body
button		h:commandButton
button	jsf:outcome	h:button
form		h:form
head		h:head
img		h:graphicImage
input	type="button"	h:commandButton
input	type="checkbox"	h:selectBooleanCheckbox
input	type="color"	h:inputText
input	type="date"	h:inputText
input	type="datetime"	h:inputText
input	type="datetime-local"	h:inputText
input	type="email"	h:inputText
input	type="month"	h:inputText
input	type="number"	h:inputText
input	type="range"	h:inputText
input	type="search"	h:inputText
input	type="time"	h:inputText
input	type="url"	h:inputText
input	type="week"	h:inputText
input	type="file"	h:inputFile
input	type="hidden"	h:inputHidden
input	type="password"	h:inputSecret
input	type="reset"	h:commandButton
input	type="submit"	h:commandButton
input	type="*"	h:inputText
label		h:outputLabel

link		h:outputStylesheet
script		h:outputScript
select	multiple="*"	h:selectManyListbox
select		h:selectOneListbox
textarea		h:inputTextArea

Nachdem Pass-Through-Elemente in JSF-Tags transformiert werden, können sie auch wie JSF-Tags behandelt werden. Im folgenden Codefragment wird zum Beispiel ein Validator direkt mit`f:validateLength`zum HTML-Element hinzugefügt. Das sieht zugegebenermaßen auf den ersten Blick etwas befremdlich aus, funktioniert aber einwandfrei.

```
<input type="text" jsf:value="#{bean.name}">
  <f:validateLength minimum="3"/>
</input>
```

Im nächsten Beispiel wird das Elementamit dem Attribut`jsf:outcome`ein ein`h:link`-Tag transformiert. Daher ist es auch möglich, die ID mit`f:param`als Parameter für den gerenderten Link zu definieren.

```
<a jsf:outcome="details" title="Show #{person.name}">
  #{person.name}
  <f:param name="id" value="#{person.id}"/>
</a>
```

Passen Sie genau auf, welche Attribute von Pass-Through-Elementen Sie im Namensraum`http://xmlns.jcp.org/jsf`definieren. Im nächsten Beispiel ist das Attribut`value`ohne Namensraum definiert und wird als Pass-Through-Attribut an die Komponente weitergereicht - ein großer Unterschied, wie wir gleich sehen werden.

```
<input jsf:id="name" type="text" value="#{bean.name}">
  <f:validateLength minimum="3"/>
</input>
```

Auf den ersten Blick funktioniert das Beispiel ohne Probleme. Der gerenderte Wert wird aber nicht von der Komponente verwaltet, sondern immer direkt aus der Value-Expression in die gerenderte Ausgabe geschrieben. Bei einem Validierungsfehler bekommt der Benutzer dadurch aber wieder den alten Wert und nicht den Submitted-Value zu sehen.

Tipp: Definieren Sie bei Pass-Through-Elementen unbedingt alle Attribute, die zur Komponente gehören, im Namensraum`http://xmlns.jcp.org/jsf`.

JSF kann auch mit Pass-Through-Elementen umgehen, für die keine fixe Zuordnung zu einem JSF-Tag definiert ist. In diesem Fall wird eine generische Komponente in den Komponentenbaum eingefügt. Damit ist es auch möglich, HTML5-Elemente wie`progress`zu Pass-Through-Elementen zu machen und sie sogar mit Ajax-Verhalten auszustatten, wie das folgende Beispiel zeigt. Jeder Klick auf das`progress`-Element löst einen Ajax-Request aus und rendert die ihm zugeordnete Komponente neu.

```
<button jsf:action="#{bean.update}">
  Aktualisieren
  <f:ajax render="progress"/>
</button>
<progress jsf:id="progress" max="10" value="#{bean.progress}">
  <f:ajax event="click" render="@this"/>
</progress>
```

8.4 MyGourmet 15: HTML5

Bei*MyGourmet 15*handelt es sich um eine erweiterte Variante von*MyGourmet 14*, die an einigen Stellen mit Pass-Through-Attributen und Pass-Through-Elementen erweitert wurde.
In`editAddress.xhtml`haben zum Beispiel alle Eingabefelder das HTML5-Attribut`placeholder`über`f:passThroughAttribute`bekommen. In`editProvider.xhtml`hingegen wurde`placeholder`direkt als Attribut im Pass-Through-Namensraum hinzugefügt. Des Weiteren wurde die Kompositkomponente`mc:inputSpinner`durch ein Eingabefeld mit dem neuen HTML5-Typ`number`ersetzt, wie das folgende Beispiel zeigt.

```
<h:inputText pt:type="number" pt:min="0" pt:step="1"
    value="#{providerBean.provider.stars}"
    pt:placeholder="#{msgs.enter_stars}"/>
```

Wenn der Browser diesen Typ unterstützt, wird dadurch nativ ein Eingabefeld für Zahlen mit Schaltflächen zum Erhöhen und Reduzieren des Werts angezeigt.
In`editCustomer.xhtml`wurden soweit möglich alle JSF-Tags durch Pass-Through-Elemente ersetzt. Zusätzlich kommt beim Eingabefeld für die Mailadresse der HTML5-Type`email`zum Einsatz.

9 JSF und CDI

Mit*Contexts and Dependency Injection for Java (CDI)*enthält Java EE ab Version 6 einen neuen Standard, um Beans zu verwalten und über Dependency-Injection miteinander zu verbinden. Der Einsatz von CDI in JSF-Projekten bietet eine ganze Reihe von Vorteilen gegenüber dem Einsatz der internen*Managed Bean Creation Facility*. CDI bringt unter anderem einen viel mächtigeren, typsicheren Dependency-Injection-Mechanismus, Unterstützung für Interceptors und Decorators und eine sehr einfache Interaktion über Ereignisse mit, um nur einige der Vorteile zu nennen - und das alles, ohne die Komplexität der Anwendung merklich zu erhöhen. Aus Sicht von JSF macht es keinen Unterschied, ob Beans über CDI oder JSF-intern verwaltet werden. Die Verbindung zwischen der Ansicht und dem Modell erfolgt in beiden Fällen über die*Unified-EL*.
CDI unterstützt den Standard*Dependency Injection for Java*(JSR-330, auch*At Inject*genannt). JSR-330 standardisiert erstmals die wichtigsten Annotationen für die Dependency-Injection und ermöglicht den Einsatz derselben Annotationen aus dem Package`javax.inject`in unterschiedlichen Umgebungen wie*Java EE 6*,*Spring 3*oder*Guice 2*.
Nach einer Einführung in die Konzepte von CDI in Abschnitt**[Sektion: Beans und Dependency-Injection mit CDI]**finden Sie in Abschnitt**[Sektion: Konfiguration von CDI]**Hinweise zur Konfiguration. Im Anschluss daran zeigt Abschnitt**[Sektion: *MyGourmet 16*: Integration von CDI]**Details zum Beispiel*MyGourmet 16*. Abschnitt**[Sektion: Konversationen mit JSF]**widmet sich der Abbildung von Geschäftsprozessen mit Konversationen. Abschließend zeigen wir Ihnen in Abschnitt**[Sektion: Apache MyFaces CODI]**noch, wie Sie Konversationen mit CDI und dem Projekt*Apache MyFaces CODI*einsetzen.

9.1 Beans und Dependency-Injection mit CDI

Die Hauptaufgabe von CDI besteht in der Verwaltung von Beans mit einem definierten Lebenszyklus. CDI stellt zu diesem Zweck Mechanismen zur Verfügung, um Beans zu definieren und über typsichere Dependency-Injection miteinander zu verknüpfen.
CDI unterstützt unterschiedliche Arten von Beans:

- Beans, die von Klassen definiert werden (auch Managed-Beans genannt - nicht zu verwechseln mit JSF-Managed-Beans).
- Beans, die von Producer-Methoden oder -Feldern definiert werden.
- Beans, die von EJB-Session-Beans oder Java EE-Ressourcen definiert werden.

Wir zeigen Ihnen, wie Sie Beans auf Basis von Klassen und mit Producer-Methoden definieren.

9.1.1 Managed-Beans

mit CDI

Der Weg zur ersten CDI-Bean ist ausgesprochen einfach: CDI kennt keine spezielle Annotation, um Beans auf Basis von Klassen zu definieren, und macht aus so gut wie allen Klassen Beans. Ausgenommen sind zum Beispiel nicht statische, innere Klassen oder Klassen, die über keinen geeigneten Konstruktur verfügen. Ein für CDI geeigneter Konstruktor hat entweder keine Parameter oder ist mit`@Inject`annotiert. Jede CDI-Bean hat einen Gültigkeitsbereich (Scope), der mit einer Annotation auf der Bean-Klasse definiert wird. CDI unterstützt standardmäßig folgende Gültigkeitsbereiche und definiert dafür Annotationen im Package`javax.enterprise.context`:

- *Dependent-Scope*(`@Dependent`):
  Der*Dependent-Scope*ist ein sogenannter Pseudo-Scope - er hat keinen eigenen Lebenszyklus und die Lebensdauer hängt von der Verwendung der Bean ab. Wenn Sie keine Scope-Annotation angeben, verwendet CDI standardmäßig diesen Scope. Er entspricht dem*None-Scope*in JSF.
- *Request-Scope*(`@RequestScoped`):
  Die Bean lebt für die Zeitdauer einer HTTP-Anfrage. Dieser Gültigkeitsbereich entspricht dem*Request-Scope*in JSF.
- *Conversation-Scope*(`@ConversationScoped`):
  Die Bean lebt für die Zeitdauer einer Konversation. Die von CDI direkt angebotenen Konversationen sind allerdings nicht besonders praktisch. Wir empfehlen daher den Einsatz der Konversationen von*Apache MyFaces CODI*(siehe Abschnitt**[Sektion: Apache MyFaces CODI]**).
- *Session-Scope*(`@SessionScoped`):
  Die Bean lebt für die Dauer einer Sitzung, in der der Benutzer mit der Anwendung verbunden ist. Dies entspricht dem*Session-Scope*in JSF.
- *Application-Scope*(`@ApplicationScoped`):
  Für die gesamte Lebensdauer der Anwendung ist nur eine für alle Benutzer gleiche Instanz dieser Bean vorhanden. Dieser Gültigkeitsbereich entspricht dem*Application-Scope*in JSF.

Tipp:CDI-Beans in sogenannten Passivating-Scopes wie`@SessionScoped`oder`@ConversationScoped`müssen serialisierbar sein.

Listing**CDI-Bean**zeigt die Definition einer CDI-Bean mit der Klasse`CustomerService`im Application-Scope. Achten Sie bitte speziell auf das Package der Annotation`@ApplicationScoped`. Es handelt sich hier um die von CDI definierte Annotation und nicht um das JSF-Pendant mit dem gleichen Namen.

```
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class CustomerService {
  ...
}
```

In CDI ist jetzt eine Bean vom Typ`CustomerService`im Application-Scope verfügbar, die in jede andere Bean injiziert werden kann. Das Annotieren eines Feldes mit`@Inject`reicht aus, um eine Abhängigkeit auf eine andere Bean zu definieren. Listing**Dependency-Injection mit @Inject**zeigt die Bean`CustomerBean`mit einer Abhängigkeit auf die zuvor definierte Bean vom Typ`CustomerService`.

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
```

```
@Named
@RequestScoped
public class CustomerBean {
  @Inject
  private CustomerService customerService;
  ...
}
```

Beim Erstellen einer Bean-Instanz vom Typ`CustomerBean`löst CDI diese Abhängigkeit auf und injiziert die Bean-Instanz für den Typ`CustomerService`aus dem Application-Scope.
Mit der Annotation`@Named`erhält die Bean zusätzlich einen Namen, unter dem sie in Unified-EL-Ausdrücken verfügbar ist. Auch hier gilt wie bei Managed-Beans in JSF: Wenn im Element`value`von`@Named`kein expliziter Name angegeben wird, erzeugt CDI per Konvention den Namen aus dem Klassennamen mit einem kleinen Anfangsbuchstaben. Aus dem Klassennamen`CustomerBean`wird zum Beispiel der Name`cust-omer-Bean`, der dann wie folgt in einer Ansicht verwendet werden kann:

```
<h:inputText value="#{customerBean.property}"/>
```

Für den Fall, dass es mehrere Beans gleichen Typs gibt, muss die Auswahl weiter eingeschränkt werden. Dazu sieht CDI sogenannte Qualifier vor. Ein Qualifier ist eine beliebige Annotation, die mit`@javax.inject.Qualifier`annotiert ist. Listing**Qualifier in CDI**zeigt als Beispiel den Qualifier`@Special`.

```
@Retention(RUNTIME)
@Target({TYPE, FIELD, METHOD})
@Qualifier
public @interface Special {
}
```

Mit einem solchen Qualifier wird dann einerseits die Klasse der Bean und andererseits der Injektionspunkt annotiert. Aber sehen wir uns das am besten anhand eines Beispiels an. Listing**Service-Beans mit und ohne Qualifier**zeigt zwei unterschiedliche Implementierungen des Interface`Service`. Da beide Beans den Typ`Service`haben (wir gehen davon aus, dass nur das Interface nach außen hin bekannt ist), annotieren wir zur weiteren Unterscheidung die Klasse`SpecialServiceImpl`mit unserem Qualifier`@Special`.

```
@ApplicationScoped
public class ServiceImpl implements Service {
  ...
}

@ApplicationScoped
@Special
public class SpecialServiceImpl implements Service {
  ...
}
```

Dieselbe Qualifier-Annotation wird auch beim Injektionspunkt benutzt, um die Auswahl der Beans vom Typ`Service`auf das gewünschte Exemplar einzuschränken. Listing**Dependency-Injection mit @Inject und Qualifier**zeigt das entsprechende Codefragment.

```
public class MyBean {
  @Inject @Special
```

```
  private Service service;
}
```

Die Annotation`@Named`ist übrigens auch ein Qualifier, der zum Einschränken der Auswahl anhand des Namens der Bean eingesetzt werden kann. Wir raten allerdings davon ab, das zu tun, da damit der Vorteil der Typsicherheit verloren geht.
CDI-View-Scope, JSF 2.2:Falls Sie den View-Scope in CDI vermissen, haben wir eine schlechte und eine gute Nachricht für Sie. Die schlechte Nachricht ist, dass CDI standardmäßig keinen View-Scope kennt. Die gute Nachricht ist, dass JSF 2.2 hier Abhilfe schafft und den View-Scope für CDI nachreicht. Listing**CDI-Managed-Bean im View-Scope**zeigt eine von CDI verwaltete Managed-Bean im neuen CDI-View-Scope von JSF 2.2. Verwechseln Sie die Annotation`javax.faces.view.ViewScoped`für den CDI-View-Scope nicht mit der Annotation`javax.faces.bean.ViewScoped`für den JSF-View-Scope.

```
@javax.inject.Named
@javax.faces.view.ViewScoped
public class CustomerBean {
  @Inject
  private CustomerService customerService;
  ...
}
```

*Apache MyFaces CODI*bietet mit dem View-Access-Scope eine Alternative zum klassischen View-Scope - auch ohne JSF 2.2. Details dazu finden Sie in Abschnitt**[Sektion: Apache MyFaces CODI]**und im Beispiel*MyGourmet 17*.

9.1.2 Producer-Methoden

Die zweite hier vorgestellte Variante zur Definition von Beans mit CDI sind die sogenannten Producer-Methoden. Wie der Name bereits erahnen lässt, werden Beans mit diesem Konzept über spezielle Methoden definiert. Alle nicht abstrakten Methoden einer Managed-Bean oder Session-Bean, sowohl statische wie auch nicht statische, können als Producer-Methoden fungieren. Damit eine Methode zur Producer-Methode wird, muss sie mit`@Produces`annotiert werden. Der Rückgabetyp der Methode definiert dabei den Typ der Bean und der Rückgabewert wird als Bean-Instanz verwendet. Producer-Methoden können wie Bean-Klassen mit Scope- und Qualifier-Annotationen versehen werden. Listing**Producer-Methode für eine Zufallszahl**zeigt ein Beispiel, in dem eine Zufallszahl vom Typ`Integer`mit dem Qualifier`@Random`als Bean zur Verfügung gestellt wird.

```
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;
import javax.inject.Named;
import at.irian.Random;

@ApplicationScoped
public class RandomProducer {
  private java.util.Random random = new java.util.Random();
  @Produces @Named @Random
  public int getRandom() {
    return random.nextInt(1000);
  }
}
```

Der entscheidende Vorteil von Producer-Methoden besteht darin, dass die Erstellung der Bean-Instanz komplett im Verantwortungsbereich der Applikation liegt. Damit lassen sich auch Beans für Klassen definieren, die CDI nicht direkt verwenden kann (wie etwa Klassen aus dem JDK) oder die eine spezielle Initialisierung benötigen.
Die im Beispiel von der Producer-Methode erzeugte Zufallszahl kann in einer anderen Bean verwendet werden, indem ein Feld vom Typ`int`mit`@Inject`und dem Qualifier`@Random`annotiert wird:

```
public class MyBean {
  @Inject @Random
  private int random;
}
```

Nachdem der Scope nicht explizit definiert ist, kommt für die Zufallszahl der Dependent-Scope zum Einsatz. Der Scope der Bean`RandomProducer`und der Scope der erzeugten Bean haben keine direkte Beziehung zueinander. Verwendet eine Producer-Methode aber zum Beispiel Daten aus der Bean, kann deren Scope durchaus eine Rolle spielen.
Da die Producer-Methode zusätzlich mit`@Named`annotiert ist, steht die Zufallszahl auch direkt in Unified-EL-Ausdrücken zur Verfügung:

```
<h:outputText value="#{random}"/>
```

Der Name der Bean leitet sich per Konvention aus dem Methodennamen ab, falls er nicht in`@Named`angegeben ist. Bei Getter-Methoden nach dem*JavaBeans*-Standard wird der Name der Eigenschaft verwendet. Im Beispiel wird daher aus`getRandom()`der Name`random`.
Producer-Methoden können auch Parameter enthalten. Listing**Producer-Methode mit Parameter**zeigt eine Producer-Methode mit einem Parameter vom Typ`UserBean`. Beim Aufruf der Methode löst CDI die aktuelle Bean-Instanz für diesen Typ auf und übergibt sie an die Methode. Dort wird aus der übergebenen Bean der Name des aktuell eingeloggten Benutzers ausgelesen und als eigene Bean mit dem Typ`String`und dem Qualifier`@UserName`im Dependent-Scope zur Verfügung gestellt.

```
@ApplicationScoped
public class UsernameProducer {
  @Produces @Named @UserName
  public String getUserName(UserBean userBean) {
    return userBean.getUserName();
  }
}
```

Listing**Producer-Methode für Konverter**zeigt eine interessante Einsatzmöglichkeit für Producer-Methoden in JSF-Applikationen. Die gezeigte Klasse`ConverterProducer`stellt über die Methode`getCustomConverter()`den Konverter mit der Klasse`CustomConverter`als Bean zur Verfügung.

```
@ApplicationScoped
public class ConverterProducer {
  @Produces @Named
  public CustomConverter getCustomConverter() {
    return new CustomConverter();
  }
}
```

Der Einsatz des Konverters sieht wie im folgenden Beispiel aus:

```
<h:inputText value="#{bean.property}"
    converter="#{customConverter}"/>
```

In Abschnitt**Sektion: Definition mit Java**zeigen wir Ihnen, wie Sie mit JSF 2.2 Faces-Flows über CDI-Producer-Methoden definieren können.

9.2 Konfiguration von CDI

CDI ist wie JSF nur eine Spezifikation, für die es mehrere Implementierungen gibt. Zu den bekanntesten zählen zurzeit*Weld*von*JBoss*(die Referenzimplementierung) und*Apache OpenWebBeans*. Grundsätzlich gibt es zwei Varianten, um CDI einzusetzen. Wenn die Anwendung auf einem Applikationsserver läuft, der Java EE 6 unterstützt (wie*Glassfish 3*oder*JBoss AS 7*), ist CDI bereits integriert und einsatzbereit. Läuft die Applikation hingegen nur auf einem Servlet-Container wie*Tomcat*oder*Jetty*, muss eine CDI-Implementierung manuell integriert werden.
Nachdem unsere*MyGourmet*-Beispiele auf*Jetty*laufen, haben wir uns für die zweite Variante mit*OpenWebBeans*entschieden, da sich die Integration in eine JSF-Anwendung sehr einfach gestaltet. Nach dem Einbinden aller benötigten Jar-Dateien muss nur noch der Listener`WebBeansConfigurationListener`wie in Listing**Listener für OpenWebBeans in web.xml**in der`web.xml`eingetragen werden. Die Liste aller benötigten Abhängigkeiten finden Sie im Quellcode zu Beispiel*MyGourmet 16*in der`pom.xml`.

```
<listener>
  <listener-class>
    org.apache.webbeans.servlet.WebBeansConfigurationListener
  </listener-class>
</listener>
```

Damit CDI Beans in einer Webapplikation findet, muss die Datei im Verzeichnis`WEB-INF`existieren. Beans in Jar-Dateien findet CDI hingegen nur, wenn die Datei`beans.xml`im Verzeichnis`META-INF`existiert. Die Datei`beans.xml`kann Konfigurationen für CDI enthalten, bleibt aber im einfachsten Fall leer, wie Listing**Minimale beans.xml**zeigt.

```
<beans 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/beans_1_0.xsd">
</beans>
```

Tipp:CDI funktioniert nur, wenn die Datei`beans.xml`im Verzeichnis`WEB-INF`oder`META-INF`existiert.

9.3 MyGourmet 16: Integration

von CDI

In*MyGourmet 16*dreht sich alles um die Integration von CDI. Dazu wird*OpenWebBeans*über Abhängigkeiten in der`pom.xml`in das Maven-Projekt eingebunden (Details entnehmen Sie bitte direkt dem Sourcecode). Die zusätzlich notwendige Konfiguration in der`web.xml`entspricht genau der im letzten Abschnitt vorgestellten.
Die Umstellung in*MyGourmet*beschränkt sich auf das Austauschen der JSF-Annotationen durch ihre JSR-330- und CDI-Pendants in den Klassen`ProviderServiceImpl`,`AddressBean`,`CustomerBean`,`ProviderBean`und`ProviderListBean`. Die Bean`ProviderBean`wurde mit der Annotation`javax.faces.view.ViewScoped`auf den CDI-View-Scope von JSF 2.2 umgestellt.
Listing**Bean-Definition und Dependency-Injection mit CDI**zeigt die Konfiguration der Beans mit den Klassen`ProviderService`und`ProviderBean`. Die JSR-330-Annotation`@Inject`auf dem Feld`providerService`der Klasse`ProviderBean`bewirkt, dass beim Erstellen einer Bean mit dem Typ`ProviderBean`die Bean mit dem Typ`ProviderService`in das Feld injiziert wird.

```
@ApplicationScoped
public class ProviderServiceImpl implements ProviderService {
  ...
}

@Named @SessionScoped
public class ProviderBean implements Serializable {
  @Inject
  private ProviderService providerService;
    ...
}
```

Was wir Ihnen hier gezeigt haben, ist die einfachste Form der Bean-Definition mit*CDI*und JSR-330. Sobald CDI läuft und in JSF eingebunden ist, steht Ihnen die komplette Palette der Möglichkeiten offen - und glauben Sie uns, das sind eine Menge. Ein wichtiges Einsatzgebiet für die Verwendung von CDI mit JSF sind Konversationen. In Abschnitt**[Sektion: Konversationen mit JSF]**finden Sie allgemeine Informationen zu diesem Thema und Abschnitt**[Sektion: Apache MyFaces CODI]**zeigt, wie Sie Konversationen mit CDI und*Apache MyFaces CODI*einsetzen.

9.4 Konversationen mit JSF

In vielen Webanwendungen lassen sich die zugrunde liegenden Geschäftsprozesse nicht direkt auf den Seitenfluss abbilden. Viele Prozesse, die aus Sicht des Benutzers eine abgeschlossene Einheit bilden, erstrecken sich in der Anwendung über mehrere Anfragen oder sogar über mehrere Ansichten hinweg. Denken Sie zum Beispiel an die Registrierung eines Benutzers, bei der im ersten Schritt die Login-Daten und im zweiten Schritt Daten zur Person abgefragt werden. Für den Benutzer der Anwendung ist dieser Vorgang eine in sich abgeschlossene Tätigkeit, die mit dem Anzeigen der ersten Ansicht beginnt und durch das Betätigen der Fertigstellen-Schaltfläche im zweiten Schritt abgeschlossen wird. Aus Sicht der Webanwendung handelt es sich dabei allerdings nur um eine Reihe von Anfragen auf zwei verschiedene Seiten.
Dadurch stellt sich die Frage, in welchem Gültigkeitsbereich die Daten während des Prozesses abgelegt werden müssen, damit sie in jedem Schritt zur Verfügung stehen. Managed-Beans im Request-Scope werden nach jeder Anfrage neu erzeugt und eignen sich daher nicht. Der View-Scope ist auch nur dann

ausreichend, wenn der Prozess nicht mehr als eine Ansicht umfasst. Managed-Beans im Application-Scope eignen sich nicht für unsere Zwecke, da sie nur einmal pro Anwendung erzeugt werden und dadurch alle Benutzer dieselben Daten sehen. Bleibt als letzte Alternative nur noch der Session-Scope übrig. Der Session-Scope löst zwar das Verfügbarkeitsproblem während des Prozesses, bringt aber einige entscheidende Nachteile mit sich.
An dieser Stelle kommen Konversationen ins Spiel. Konversationen sind der ideale Speicherort für Managed-Beans, deren Lebensdauer über eine Anfrage oder Ansicht hinausgeht. Bei Webanwendungen tritt dieser Fall öfters ein, da sich Geschäftsprozesse nicht immer direkt auf den Seitenfluss der Applikation abbilden lassen. Konversationen bieten einige entscheidende Vorteile gegenüber der Session:

- Eine Konversation kann im Gegensatz zur Session sehr einfach beendet und aus dem Speicher entfernt werden, ohne andere Konversationen oder Managed-Beans außerhalb der Konversation zu beeinflussen.
- Es kann beliebig viele Konversationen pro Benutzer geben.
- Eine Konversation wird normalerweise für jedes Fenster oder Tab des Browsers neu angelegt. Dadurch kann die Anwendung gleichzeitig in mehreren Fenstern oder Tabs laufen, die sich gegenseitig nicht beeinflussen. Mit der Session ist das nicht so einfach möglich, da bei den meisten Browsern für alle Tabs und oft sogar für Fenster die gleiche Session zum Einsatz kommtBrowser benutzen oft für alle Tabs und Fenster dieselben Cookies, wodurch am Server dieselbe Session verwendet wird.:.

Wenn Sie CDI einsetzen, kann JSF sehr einfach um Konversationen erweitert werden. Das Projekt*Apache MyFaces CODI*bietet neben Konversationen noch eine Vielzahl von Erweiterungen zur nahtlosen Integration von JSF und CDI. Details zu CODI finden Sie in Abschnitt**[Sektion: Apache MyFaces CODI]**.

9.5 Apache MyFaces CODI

Das Projekt*Apache MyFaces Extensions CDI*(kurz CODI) ist eine portable Erweiterung von CDI, die unter dem Dach von*Apache MyFaces*entwickelt wird. CODI bietet eine ganze Reihe von Features, um die Integration von JSF und CDI so einfach wie möglich zu gestalten.
CODI besteht aus mehreren Modulen, die sich je nach Bedarf in die Anwendung einbinden lassen. Neben einem Core-Modul für die Basisfunktionalität gibt es unter anderem noch Module für JSF, JPA, Messaging und Bean-Validation. CODI zeigt sich sehr flexibel, was seine Umgebung betrifft. Es läuft CDI-seitig mit*Apache OpenWebBeans*und*JBoss Weld*und JSF-seitig mit*Apache MyFaces*und*Mojarra*.
CDI kann an und für sich als äußerst gelungen bezeichnet werden, das Konversationskonzept ist allerdings eher dürftig. Als eines der wichtigsten Features bietet CODI daher im JSF-Modul eine erweiterte Unterstützung von Konversationen für CDI. In Abschnitt**[Sektion: Konversationen mit CODI]**werfen wir daher einen etwas genaueren Blick auf dieses Thema.
Ein weiteres interessantes Feature von CODI ist die typsichere Konfiguration von Ansichten und die damit mögliche Definition von Page-Beans. Dabei handelt es sich um Beans, die an eine Ansicht gebunden sind und bei der Ausführung des Lebenszyklus an mehreren Stellen, wie etwa kurz vor dem Rendern der Ansicht, benachrichtigt werden. Details dazu finden Sie in Abschnitt**[Sektion: View-Config und Page-Beans]**.
Abschließend zeigt Abschnitt**[Sektion: *MyGourmet 17*: Apache MyFaces CODI]**das Beispiel*MyGourmet 17*, in dem einige CODI-Features in die Praxis umgesetzt werden.

9.5.1 Konversationen

mit CODI

CODI bietet ein sehr flexibles Konversationskonzept und ermöglicht sogar die Verwendung mehrerer Konversationen auf einer Seite. Eine Konversation ist dabei immer an das aktuelle Browserfenster beziehungsweise an den Browsertab gebunden. Es gibt somit keine Probleme, wenn die Anwendung in mehreren Fenstern oder Tabs läuft.
Eine Konversation ist genau wie die Session oder die HTTP-Anfrage als Gültigkeitsbereich für Managed-Beans einsetzbar. Anders als in CDI beginnt in CODI die Lebensdauer einer Konversation mit dem ersten Zugriff auf eine Bean in der Konversation. Die Lebensdauer hängt in CODI vom Typ der Konversation ab und kann unterschiedlich lange ausfallen. Eine Konversation kann allerdings nie länger als die Session dauern, da sie in der Session abgelegt ist.
Das JSF-Modul von CODI bietet eine ganze Reihe unterschiedlicher Konversationen und bringt auch gleich die passenden Annotationen mit, um sie als Scopes für CDI-Beans zu verwenden:

- Der*Conversation-Scope*(`@ConversationScoped`) definiert eine Konversation mit manueller Lebensdauer.
- Der*View-Access-Scope*(`@ViewAccessScoped`) definiert eine Konversation mit automatischer Lebensdauer. Solange Zugriffe auf eine Bean im View-Access-Scope erfolgen, erstreckt sich ihre Lebensdauer immer über die aktuelle und die nächste Ansicht.
- Der*Window-Scope*(`@WindowScoped`) definiert eine Art Session pro Browserfenster/-tab mit manueller Lebensdauer.

9.5.1.1 Conversation-Scope

Der Conversation-Scope von CODI definiert eine Konversation mit manueller Lebensdauer für CDI-Beans. Listing**CDI-Bean im Conversation-Scope von CODI**zeigt die Bean-Klasse`WizardBean`mit den notwendigen Annotationen. Diese Klasse könnte zum Beispiel als Bean für einen mehrstufigen Wizard mit den Seiten`step1.xhtml`,`step2.xhtml`und`step3.xhtml`verwendet werden.

```
@Named
@ConversationScoped
public class WizardBean implements Serializable {
  ...
}
```

Tipp: Verwechseln Sie die CODI-Annotation`@ConversationScoped`nicht mit der gleichnamigen CDI-Annotation. Sie unterscheiden sich lediglich durch das Package.

Wie verhält sich diese Bean in der Praxis, wenn ein Benutzer zum Beispiel die Seite`step1.xhtml`aufruft? Die Bean und die Konversation werden beim ersten Zugriff auf die Bean erstellt. Da es sich um eine manuelle Konversation handelt, steht sie auch für die Seiten`step2.xhtml`und`step3.xhtml`zur Verfügung, bis sie manuell oder durch einen Timeout beendet wird.
Das manuelle Beenden der Konversation wird über einen Aufruf der Methode`close()`auf der Instanz der aktuellen Konversation erledigt. Die aktuelle Konversation lassen wir uns dabei direkt von CDI in die Bean injizieren. Listing**Manuelles Beenden einer Konversation in CODI**zeigt nochmals die Klasse`WizardBean`mit den Methoden`save()`und`cancel()`. In beiden Methoden wird die Konversation beendet.

```
@Named @ConversationScoped
public class WizardBean implements Serializable {
  @Inject
  private Conversation conversation;
```

```
  public String save() {
    conversation.close();
    return "details.xhtml";
  }
  public String cancel() {
    conversation.close();
    return "overview.xhtml";
  }
}
```

Standardmäßig erstellt CODI für jede Bean im Conversation-Scope eine eigene Konversation. Dadurch ist es auch möglich, mehrere Konversationen auf einer Seite zu haben. Manchmal ist es aber erwünscht, mehrere Beans in einer Konversation zusammenzufassen. Dazu gibt es in CODI das Konzept der sogenannten Konversationsgruppen. Mit der Annotation`@ConversationGroup`kann eine beliebige Java-Klasse als typsichere ID der Konversationsgruppe definiert werden. Listing**Konversationsgruppen in CODI**zeigt die Beans`WizardStep1`und`WizardStep2`, die beide in derselben, durch das Interface`Wizard`identifizierten Konversation liegen.

```
public interface Wizard {}

@ConversationScoped
@ConversationGroup(Wizard.class)
public class WizardStep1 implements Serializable {
  ...
}

@ConversationScoped
@ConversationGroup(Wizard.class)
public class WizardStep2 implements Serializable {
  ...
}
```

Da sich beide Beans eine Konversation teilen, werden beim Schließen der Konversation auch beide Beans aus dem Speicher entfernt. Technisch gesehen ist die Annotation`@ConversationGroup`ein Qualifier und muss daher auch verwendet werden, wenn eine Abhängigkeit auf eine damit annotierte Bean erstellt wird. Listing**@Inject mit Konversationsgruppen**zeigt ein Beispiel.

```
public class Wizard {
  @Inject @ConversationGroup(Wizard.class)
  private WizardStep1 step1;
  @Inject @ConversationGroup(Wizard.class)
  private WizardStep2 step2;
}
```

Wird keine explizite Konversationsgruppe angegeben, verwendet CODI intern die Klasse der Bean als ID der Konversation.

9.5.1.2 View-Access-Scope

Der View-Access-Scope von CODI definiert eine Konversation mit automatischer Lebensdauer für CDI-Beans. Grundsätzlich erstreckt sich die Lebensdauer einer Bean im View-Access-Scope immer auf die Ansicht, in der sie momentan verwendet wird, und auf die nächste Ansicht. Listing**CDI-Bean im View-Access-Scope von CODI**zeigt die Bean`DetailsBean`im View-Access-Scope.

```
@Named
@ViewAccessScoped
public class DetailsBean implements Serializable {
  ...
}
```

Sehen wir uns dazu ein Beispiel an. Findet der erste Zugriff auf`DetailsBean`in der Seite`showDetails.xhtml`statt, werden die Konversation und die Bean erstellt. Die Lebensdauer der Konversation erstreckt sich jetzt per Definition auf die Seite`showDetails.xhtml`und auf die nächste Seite. Solange der Benutzer auf dieser Seite bleibt - etwa weil Ajax-Anfragen ausgeführt werden -, ist die Konversation aktiv. Navigiert der Benutzer im nächsten Schritt auf die Seite`editDetails.xhtml`, bleibt die Konversation weiter aktiv. Der entscheidende Punkt ist jetzt, ob auf dieser Seite ein Zugriff auf die Bean erfolgt. Falls ja, verlängert CODI die Laufzeit um eine weitere Ansicht. Falls nein, wird die Konversation aus dem Speicher entfernt, sobald der Benutzer auf eine Seite mit unterschiedlicher View-ID navigiert. Konversationen mit automatischer Lebensdauer sind zwar sehr praktisch, können aber zu unerwarteten Ergebnissen führen, wenn eine Bean - vielleicht unbeabsichtigt - auf mehreren Seiten referenziert wird.

9.5.1.3 Window-Scope

Der Window-Scope von CODI definiert eine Konversation mit manueller Lebensdauer für CDI-Beans. Die Konversation ist dabei an ein Browserfenster beziehungsweise an einen Browsertab gebunden und fungiert als eine Art Session pro Fenster beziehungsweise Tab. Listing**CDI-Bean im Window-Scope von CODI**zeigt eine Bean im Window-Scope.

```
@Named
@WindowScoped
public class SettingsBean implements Serializable {
  ...
}
```

9.5.1.4 Window-Context

Wenn Sie*MyGourmet 17*starten und im Browser durch die Anwendung klicken, werden Sie bemerken, dass jede URL der Anwendung den Request-Parameter mit dem Namen`windowId`aufweist. CODI benutzt den Wert dieses Parameters, um Fenster und Tabs derselben Browserinstanz zu unterscheiden. Damit sich Konversationen eindeutig auf ein Fenster oder einen Tab des Browsers abbilden lassen, führt CODI den Window-Context ein. Der Wert des Parameters`windowId`ist die ID des aktuellen Window-Contexts, in dem alle Konversationen eines Fensters oder Tabs in der Session abgelegt sind.
Den Window-Context kann man sich wie die Konversation direkt von CDI einimpfen lassen. Ein Schließen des Window-Contexts beendet alle darin befindlichen Konversationen. Listing**CODI Window-Context**zeigt ein Beispiel.

```
@WindowScoped
public class WindowBean implements Serializable {

  @Inject
  private WindowContext windowContext;

  public void closeWindow() {
    windowContext.close();
  }
}
```

9.5.2 View-Config und Page-Beans

Ein weiteres interessantes Feature von CODI ist die typsichere Konfiguration von Ansichten mit View-Config-Klassen. Dank impliziter Navigation erlaubt ja JSF ab Version 2.0 an vielen Stellen die direkte Verwendung der View-ID zur Navigation. Das ist zwar sehr praktisch, kann aber mit steigender Größe der Applikation zu Problemen führen, wenn XHTML-Dateien umbenannt oder umstrukturiert werden.

9.5.2.1 View-Config

Die grundlegende Idee einer View-Config ist, Seiten nicht mehr über ihre View-IDs, sondern über spezielle Klassen zu referenzieren. Eine View-Config-Klasse wird einmal zentral definiert und dann im gesamten Projekt stellvertretend für eine Seite verwendet. Intern bildet CODI die View-Config wieder auf eine View-ID ab.
Listing**View-Config mit CODI**zeigt eine erste View-Config. Die Klasse muss nur das Interface`ViewConfig`implementieren und mit`@Page`annotiert werden.

```
@Page
public class Overview implements ViewConfig {}
```

Die Verbindung zwischen der Klasse und der View-ID erfolgt per Konvention über den Klassennamen. Aus der Klasse`Overview`leitet CODI zum Beispiel die View-ID`/overview.xhtml`ab.
Listing**View-Config im Einsatz**zeigt, wie die View-Config aus Listing**View-Config mit CODI**in einer Action-Methode zur Navigation eingesetzt wird. Statt der View-ID wird jetzt einfach die View-Config-Klasse zurückgeliefert. Dazu muss natürlich der Rückgabewert der Methode angepasst werden, was mit JSF ab Version 2.0 aber kein Problem darstellt.

```
@ViewAccessScoped
public class DetailsBean implements Serializable {
  public Class<? extends ViewConfig> save() {
    return Overview.class;
  }
}
```

Im vorherigen Beispiel sind wir davon ausgegangen, dass alle Seitendeklarationen im Wurzelverzeichnis der Applikation liegen. CODI ermöglicht mit der typsicheren Konfiguration aber auch das Abbilden von Verzeichnisstrukturen in Form von Klassenhierarchien. Sehen wir uns dazu als Beispiel in Listing**View-Config mit CODI**die Konfiguration für die Seiten`details.xhtml`und`overview.xhtml`im Verzeichnis`pages`an.

```
public interface Pages extends ViewConfig {
  @Page
  public final class Overview implements Pages {}
  @Page
  public final class Details implements Pages {}
}
```

Das Interface`Pages`repräsentiert das Verzeichnis`pages`und ist vom Interface`ViewConfig`abgeleitet. Die

konkreten View-Config-Klassen`Overview`und`Details`sind als innere Klassen umgesetzt, die jetzt nicht mehr direkt`ViewConfig`, sondern`Pages`implementieren. Aus der Klasse macht CODI intern die View-ID`/pages/details.xhtml`- genau was wir erreichen wollten.
Als angenehmer Nebeneffekt sind die View-Config-Klassen dadurch übersichtlich in einem Interface zusammengefasst. Eine solche Gruppierung ist übrigens auch dann möglich, wenn die Seiten nicht in einem Verzeichnis liegen. Dazu muss das Interface lediglich mit der Annotation`@Page(basePath="")`versehen werden, wie Listing**View-Config mit Page-Beans**zeigt. Aus der Klasse`Details`wird wieder die View-ID`/details.xhtml`.

9.5.2.2 Page-Beans

Mit den View-Config-Klassen bietet CODI die Möglichkeit, Page-Beans für Ansichten zu definieren. Dabei handelt es sich um eine Bean, die an eine Ansicht gebunden ist und bei der Ausführung des Lebenszyklus an mehreren Stellen benachrichtigt wird. Die Verbindung zwischen Ansicht und Page-Bean wird mit der Annotation`@PageBean`auf der View-Config-Klasse definiert. Listing**View-Config mit Page-Beans**zeigt ein Beispiel.

```
@Page(basePath = "")
public interface Pages extends ViewConfig {
  @Page
  @PageBean(OverviewBean.class)
  public final class Overview implements Pages {}
  @Page
  @PageBean(DetailsBean.class)
  public final class Details implements Pages {}
}
```

Sobald eine Ansicht mit einer Page-Bean verbunden ist, wird die Bean von CODI zu bestimmten Zeitpunkten während des Lebenszyklus benachrichtigt. Dabei werden folgende Methoden aufgerufen:

- Nach dem Erstellen der Ansicht wird die mit`@InitView`annotierte Methode aufgerufen.
- Vor dem Aufrufen der Action-Methode wird die mit`@PrePageAction`annotierte Methode aufgerufen.
- Vor der Render-Response-Phase wird die mit`@PreRenderView`annotierte Methode aufgerufen.
- Nach der Render-Response-Phase wird die mit`@PostRenderView`annotierte Methode aufgerufen.

Listing**Page-Bean**zeigt ein Beispiel für eine Page-Bean mit zwei annotierten Methoden.

```
@ViewAccessScoped
public class OverviewBean {
  @InitView
  public void init() {...}
  @PreRenderView
  public void loadData() {...}
}
```

9.5.3 MyGourmet 17: Apache MyFaces

CODI

*MyGourmet 17*integriert*Apache MyFaces CODI*und zeigt einige Anwendungsfälle. Das Beispiel verwendet das Core- und das JSF-2.0-Modul von CODI. Beide Module werden über Abhängigkeiten in der`pom.xml`in das Maven-Projekt eingebunden. Details entnehmen Sie bitte direkt dem Sourcecode. Auf die Konfiguration von CDI werden wir hier nicht mehr näher eingehen.
In*MyGourmet 17*haben wir den Kundenbereich der Anwendung leicht umgebaut. Die Startseite ist jetzt`customerList.xhtml`, auf der eine Liste aller Kunden mit`mc:dataTable`dargestellt wird. Von dieser Ansicht aus kann der Benutzer zur Detailseite eines Kunden navigieren, einen neuen Kunden anlegen oder einen vorhandenen Kunden löschen.
Die Bean`CustomerListBean`im View-Access-Scope ist als Page-Bean der Ansicht definiert. Listing**MyGourmet 17: Page-Bean der Kunden-Übersichtsseite**zeigt die Klasse. Die Liste der Kunden wird in der Methode`preRenderView`geladen. Da sie mit`@PreRenderView`annotiert ist, wird sie vor jedem Rendern der Ansicht von CODI aufgerufen. Zum Löschen eines Kunden wird über eine`h:commandLink`-Komponente die Methode`deleteCustomer`aufgerufen. Der zu löschende Kunde wird dabei direkt als Parameter übergeben:

```
<h:commandLink value="#{msgs.delete}"
    action="#{customerListBean.deleteCustomer(customer)}">
  <f:ajax render=":form:addressPanel:addresses"/>
</h:commandLink>
```

Das Löschen eines Kunden wird als Ajax-Anfrage ausgeführt. Der View-Access-Scope der Bean bedeutet, dass die Konversation der Bean so lange offen bleibt, bis nicht mehr auf sie zugegriffen wird.

```
@Named @ViewAccessScoped
public class CustomerListBean implements Serializable {
  @Inject
  private CustomerService customerService;
  private List<Customer> customerList;
  @PreRenderView
  public void preRenderView() {
    customerList = customerService.findAll();
  }
  public List<Customer> getCustomerList() {
    return customerList;
  }
  public void deleteCustomer(Customer customer) {
    customerService.delete(customer);
  }
}
```

Wir haben im Zuge der Änderungen in*MyGourmet 17*alle Operationen für Objekte vom Typ`Customer`im Interface`CustomerService`zusammengefasst. Die Implementierung`CustomerServiceImpl`dieses Interface steht als CDI-Bean zur Verfügung. Listing**MyGourmet 17: Page-Bean der Kunden-Übersichtsseite**zeigt, wie die Abhängigkeit zum Service mit`@Inject`definiert wird.
Listing**MyGourmet 17: View-Config**zeigt Teile der View-Config für*MyGourmet 17*im Interface`View`. Aus`showCustomer.xhtml`wird zum Beispiel die Klasse`ShowCustomer`. Da alle Klassen`View`implementieren, muss dort über`basePath`in der Annotation`@Page`der Pfad überschrieben werden. Ohne diese Anpassung würde CODI davon ausgehen, dass die Seiten im Verzeichnis`/view`liegen und die View-IDs dementsprechend anpassen. Alternativ könnten wir natürlich die XHTML-Dateien in das Verzeichnis`/view`verschieben. Die Definition der Page-Beans finden Sie ebenfalls in der View-Config in Listing**MyGourmet 17: View-Config**.

```
@Page(basePath = "")
```

```
public interface View extends ViewConfig {
  @Page @PageBean(AddCustomerBean.class)
  public class AddCustomer1 implements View {}
  @Page @PageBean(AddCustomerBean.class)
  public class AddCustomer2 implements View {}
  @Page @PageBean(CustomerListBean.class)
  public class CustomerList implements View {}
  @Page @PageBean(CustomerBean.class)
  public class ShowCustomer implements View {}
}
```

Die Detailseite`showCustomer.xhtml`ist über eine`h:commandLink`-Komponente in der Übersichtsseite erreichbar. Bei einem Klick auf den Link wird die Methode`showCustomer`der Bean`CustomerBean`mit der ID des Kunden aufgerufen. Die Methode lädt den Kunden und gibt die View-Config der Detailseite für die Navigation zurück. Die Klasse`AddressBean`ist in der Klasse`CustomerBean`aufgegangen. Listing**MyGourmet 17: Page-Bean der Kundenansichten**zeigt die relevanten Teile der Klasse`CustomerBean`.

```
@Named @ViewAccessScoped
public class CustomerBean extends CustomerBeanBase {
  @Inject
  private CustomerService customerService;
  public Class<? extends ViewConfig> showCustomer(long id) {
    this.customer = customerService.findById(id);
    return View.ShowCustomer.class;
  }
}
```

Wenn der Benutzer auf die Detailseite navigiert, wird eine Instanz der Bean`CustomerBean`im View-Access-Scope inklusive der Konversation erstellt. Die Konversation bleibt aktiv, bis während einer Anfrage keine Zugriffe mehr auf die Bean`CustomerBean`erfolgen.
Der Wizard zum Anlegen eines neuen Kunden ist ebenfalls von der Übersichtsseite aus erreichbar. Der Ablauf besteht aus den beiden Ansichten`addCustomer1.xhtml`zur Eingabe der Basisdaten des Kunden und`addCustomer2.xhtml`zur Eingabe einer Adresse. Die Bean`AddCustomerBean`ist die Page-Bean für beide Ansichten und liegt im Conversation-Scope von CODI. Listing**MyGourmet 17: Page-Bean des Wizards zum Anlegen eines Kunden**zeigt die relevanten Teile der Klasse`AddCustomerBean`. Die Bean bekommt den Service`CustomerService`injiziert, der zum Erzeugen einer neuen`Customer`-Instanz in der Methode`createCustomer`verwendet wird. Da diese Methode mit`@InitView`annotiert ist, wird sie von CODI nach der Restore-View-Phase aufgerufen, wenn der Lebenszyklus für eine der beiden verknüpften Ansichten ausgeführt wird. So ist gewährleistet, dass immer eine Instanz der Klasse`Customer`existiert.

```
@Named @ConversationScoped
public class AddCustomerBean extends CustomerBeanBase {
  @Inject
  private CustomerService customerService;
  @Inject
  private Conversation conversation;
  @InitView
  public void createCustomer() {
    if (customer == null) {
      customer = customerService.createNew();
    }
  }
  public Class<? extends ViewConfig> save() {
    customerService.save(customer);
    conversation.close();
```

```
    return View.CustomerList.class;
  }
  public Class<? extends ViewConfig> cancel() {
    conversation.close();
    return View.CustomerList.class;
  }
}
```

Da die Bean`AddCustomerBean`im Conversation-Scope liegt, müssen wir uns selbst um das Beenden der Konversation kümmern. Dazu wird in den Action-Methoden`save()`und`cancel()`die Methode`close()`auf der Konversation aufgerufen, die wir uns von CDI injizieren lassen.

10 PrimeFaces -- JSF und mehr

Komponenten sind ein essenzieller Bestandteil von*JavaServer Faces*und bilden einen zentralen Erweiterungspunkt. Der JSF-Standard definiert bereits eine ganze Reihe von Komponenten und Tags für die grundlegenden Anforderungen einer Webapplikation. Darüber hinausgehend sind im Laufe der letzten Jahre diverse Komponentenbibliotheken entstanden. Die meisten dieser Bibliotheken bieten neben einem erweiterten Angebot von Komponenten auch noch andere Konzepte, die das Entwickeln von JSF-Anwendungen teils erheblich vereinfachen. Eine der zurzeit populärsten und am aktivsten weiterentwickelten Komponentenbibliotheken ist*PrimeFaces*. In diesem Kapitel werden wir Ihnen zeigen, welche Funktionalitäten*PrimeFaces*in Version 3.5 über den JSF-Standard hinaus anbietet.
Nach einem kurzen Überblick in Abschnitt**[Sektion: PrimeFaces -- ein Überblick]**gibt Abschnitt**[Sektion: Komponenten]**einen Einblick in die Welt der*PrimeFaces*-Komponenten. In Abschnitt**[Sektion: Themes]**zeigen wir Ihnen anschließend, wie Sie das Aussehen einer*PrimeFaces*-Applikation mit Themes anpassen.*PrimeFaces*bietet auch im Ajax-Bereich einige Erweiterungen gegenüber dem JSF-Standard - Abschnitt**[Sektion: PrimeFaces und Ajax]**zeigt die Details. Abschließend werfen wir in Abschnitt**[Sektion: *MyGourmet 18: PrimeFaces*]**noch einen Blick auf das Beispiel*MyGourmet 18*.

10.1 PrimeFaces -- ein Überblick

*PrimeFaces*hat sich seit seiner Veröffentlichung im Jahr 2010 zu einer der populärsten Open-Source-Komponentenbibliotheken für JSF entwickelt. Der Kernbestandteil von*PrimeFaces*ist ein Set von momentan etwa 100 aufeinander abgestimmter Komponenten, deren Erscheinungsbild mit Themes an eigene Bedürfnisse angepasst werden kann. Die Palette der angebotenen Funktionalität geht weit über den JSF-Standard hinaus und reicht von diversen Eingabekomponenten über Komponenten zur Darstellung von Daten in Form von Listen, Bäumen oder Tabellen bis hin zu Komponenten für Diagramme oder*Google-Maps*. Als Zugabe gibt es unter anderem noch Erweiterungen im Bereich Ajax und die Unterstützung von Ajax-Push. Mit*PrimeFaces Mobile*existiert zusätzlich eine angepasste Variante, um die Entwicklung von Webapplikationen für mobile Endgeräte zu vereinfachen.
Neben der Funktionalität legen die Entwickler von*PrimeFaces*einen starken Fokus auf Themen wie Leichtgewichtigkeit und Performance - ein Unterfangen, das durchaus als gelungen bezeichnet werden kann. Die Integration von*PrimeFaces*in die eigene Webapplikation gestaltet sich daher äußerst einfach. Für die Basisversion muss lediglich eine einzige Jar-Datei in das Projekt eingebunden werden. Nur beim Einsatz von optionalen Features wie dem Export nach PDF oder Excel oder dem File-Upload sind weitere Abhängigkeiten notwendig. Details dazu entnehmen Sie bitte der Dokumentation von*PrimeFaces*.
Falls Sie in Ihrem Projekt*Maven*einsetzen, müssen Sie nur die in Listing**Maven-Abhängigkeit für PrimeFaces**gezeigte Abhängigkeit zur`pom.xml`hinzufügen. Andernfalls finden Sie die aktuellste Version von*PrimeFaces*unter .

```
<dependency>
  <groupId>org.primefaces</groupId>
  <artifactId>primefaces</artifactId>
  <version>3.5</version>
</dependency>
```

Damit Maven die Abhängigkeit auflösen kann, muss das*PrimeFaces*-Repository mit der URL**http://repository.primefaces.org**in der`pom.xml`eingetragen werden. Details dazu finden Sie im Quellcode zu*MyGourmet 18*.
Ansonsten ist keine weitere Konfiguration notwendig - weder in der`web.xml`noch in der`faces-config.xml`. Hier zeigt sich deutlich der Fokus auf die Leichtgewichtigkeit in allen Belangen.

10.2 Komponenten

*PrimeFaces*bietet in Version 3.5 etwa 100 Komponenten für die unterschiedlichsten Anwendungsfälle an, die unter dem Namensraum`http://primefaces.org/ui`verfügbar sind. Die Vorstellung aller Komponenten würde den Rahmen des Buches sprengen, weshalb wir hier nur eine repräsentative Auswahl zeigen. Eine Übersicht aller verfügbaren Komponenten finden Sie unter der Adresse**http://www.primefaces.org/showcase**.
Listing**PrimeFaces-Beispiel mit p:panel**zeigt ein erstes Beispiel mit der Panel-Komponente von*PrimeFaces*, die über das Tag`p:panel`in die Seite eingebunden wird. Das Präfix`p`muss mit dem Namensraum`http://primefaces.org/ui`verknüpft sein, damit JSF die Tags der*PrimeFaces*-Komponenten auflösen kann. Im Beispiel bekommt das Panel über das Attribut`header`noch eine Überschrift und wird durch das Setzen von`toggleable`auf`true`im Browser ein- und ausblendbar gemacht. Vergessen Sie auch mit*PrimeFaces*nicht,`h:head`und`h:body`zu verwenden: Nur so ist gewährleistet, dass alle Stylesheets und Skripte in die Seite eingebunden werden.

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:p="http://primefaces.org/ui">
<h:head><title>Test</title></h:head>
<h:body>
  <p:panel header="Test" toggleable="true">
    <h:outputText value="Hallo, hier spricht PrimeFaces!"/>
  </p:panel>
</h:body>
</html>
```

Abbildung**p:panel im Browser**zeigt die gerenderte Ausgabe des Beispiels aus Listing**PrimeFaces-Beispiel mit p:panel**. Die Panel-Komponente wird automatisch mit den Einstellungen vom aktuellen Theme dargestellt. Sie müssen dazu weder ein Stylesheet einbinden (das macht JSF automatisch) noch eine CSS-Klasse auf der Komponente setzen. Details zu Themes finden Sie in Abschnitt**[Sektion: Themes]**.

Abbildung:p:panel im Browser

Sie können in einer Applikation und sogar innerhalb einer Seite Komponenten und Tags aus*PrimeFaces*beliebig mit jenen aus dem JSF-Standard kombinieren. In den meisten Fällen müssen Sie das sogar machen, da es für Tags wie`h:form`,`h:outputText`,`h:head`oder`h:body`keine Alternativen gibt.

Probleme sind dadurch nicht zu erwarten, da sich*PrimeFaces*sehr strikt an den JSF-Standard hält. Für andere Standardkomponenten gibt es allerdings sehr wohl Alternativen, wie der nächste Abschnitt zeigt.

10.2.1 Erweiterte Standardkomponenten

*PrimeFaces*bietet für eine ganze Reihe von Komponenten aus dem JSF-Standard erweiterte Varianten an. Komponenten wie`p:inputText`,`p:selectOneRadio`,`p:messages`oder`p:outputLabel`verfügen über dieselbe Grundfunktionalität wie die gleichnamigen Standardkomponenten. In den meisten Fällen sind die erweiterten Komponentenklassen sogar von den JSF-Klassen abgeleitet. Die*PrimeFaces*-Komponenten sollten aber trotzdem bevorzugt zum Einsatz kommen, da sie Themes unterstützen und meist zusätzliche Funktionen bereitstellen.
Listing**Beispiel mit erweiterten Standardkomponenten in PrimeFaces**zeigt ein Beispiel mit erweiterten Standardkomponenten aus*PrimeFaces*. Im Vergleich zu bisherigen Beispielen mit Tags aus dem JSF-Standard hat sich bis auf das Präfix nicht viel geändert - selbst die Namen der Attribute bleiben gleich. Ein genauerer Blick auf das Tag`p:inputText`mit der ID`birthday`zeigt, dass Konverter und Validatoren aus JSF oder aus eigenen Tag-Bibliotheken problemlos weiterverwendet werden können. Ähnliches gilt für`p:selectOneRadio`: Die Definition der Auswahlmöglichkeiten wird wie gehabt mit`f:selectItem`oder`f:selectItems`erledigt.

```
<p:messages showDetail="true" showSummary="false"/>
<h:form id="form">
  <h:panelGrid id="baseData" columns="2">
    <p:outputLabel for="name" value="Name"/>
    <p:inputText id="name" value="#{bean.name}"/>
    <p:outputLabel for="birthday" value="Geburtstag"/>
    <p:inputText id="birthday" value="#{bean.birthday}">
      <f:convertDateTime pattern="dd.MM.yyyy"/>
      <mg:validateAge minAge="18"/>
    </p:inputText>
    <p:outputLabel for="gender" value="Geschlecht"/>
    <p:selectOneRadio id="gender" value="#{bean.gender}">
      <f:selectItems value="#{bean.genderItems}"/>
    </p:selectOneRadio>
  </h:panelGrid>
</h:form>
```

Abbildung**Gerenderte Ausgabe zum Beispiel mit erweiterten Standardkomponenten**zeigt die gerenderte Ausgabe des Beispiels aus Listing**Beispiel mit erweiterten Standardkomponenten in PrimeFaces**mit fehlerhaften Benutzereingaben. Hier sieht man deutlich den Unterschied zu Standard-JSF, da die Komponenten automatisch mit den Einstellungen vom aktuellen Theme dargestellt werden.

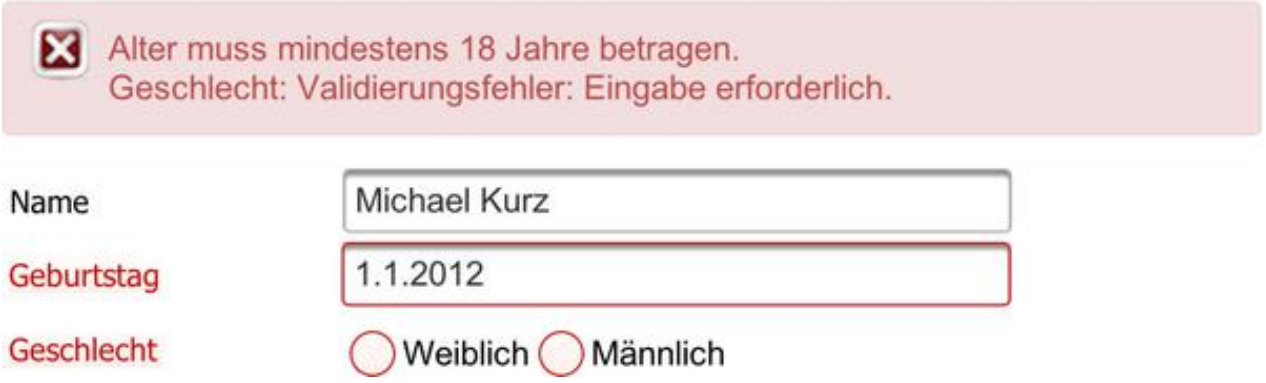


Abbildung:Gerenderte Ausgabe zum Beispiel mit erweiterten Standardkomponenten

In Abbildung**Gerenderte Ausgabe zum Beispiel mit erweiterten Standardkomponenten**lassen sich noch weitere Vorteile von*PrimeFaces*erkennen. Die beiden Komponenten mit fehlerhafter Benutzereingabe werden inklusive der zugeordneten Labels in roter Farbe dargestellt. Wenn Sie ganz genau hinsehen, werden Sie noch etwas entdecken: Die Fehlermeldung für die fehlende Auswahl des Geschlechts enthält automatisch den Wert der zugeordneten Label-Komponente. In Standard-JSF würde hier die Client-ID der Komponente stehen - außer das Attribut`label`ist explizit gesetzt. Gerade solche Details erleichtern die tägliche Entwicklungsarbeit oft enorm.

10.2.2 Auswahl einiger PrimeFaces-Komponenten

*PrimeFaces*stellt eine große Zahl von Komponenten zur Verfügung, deren Funktionalität weit über den JSF-Standard hinausgeht. Im Laufe dieses Abschnitts stellen wir eine zugegeben sehr kleine Auswahl dieser Komponenten vor.

10.2.2.1 AccordionPanel -- p:accordionPanel

Die AccordionPanel-Komponente mit dem Tag`p:accordionPanel`stellt mehrere auf- und zuklappbare Tabs untereinander dar. Ein Tab gruppiert beliebigen Inhalt und wird mit dem Tag`p:tab`definiert. Die einzelnen Tabs können mit einem Klick auf ihre Titelzeile auf- und zugeklappt werden, wobei im Standardfall maximal ein Tab aktiv ist. Klickt der Benutzer auf die Titelleiste eines geschlossenen Tabs, wird dieses aufgeklappt und das zuvor aktive zugeklappt. Abbildung**p:accordionPanel im Browser**zeigt ein Beispiel mit drei Tabs, von denen der erste aktiv ist und der dritte komplett deaktiviert wurde (mehr dazu später).

Abbildung:p:accordionPanel im Browser

Listing**p:accordionPanel**zeigt den Code zum Beispiel aus Abbildung**p:accordionPanel im Browser**. Innerhalb von`p:accordionPanel`wird jeder Tab in einem`p:tab`-Tag mit beliebigem Inhalt definiert. Der Titel wird im Attribut`title`oder alternativ in einem Facet mit dem Name`title`angegeben. Der dritte Tab ist durch das Setzen des Attributs`disabled`auf`true`komplett deaktiviert und kann im Browser nicht aufgeklappt werden. Wenn Sie einen Tab dynamisch deaktivieren wollen, können Sie hier natürlich auch eine Value-Expression verwenden.

```
<p:accordionPanel>
  <p:tab title="Tab 1">
    Inhalt Tab 1
  </p:tab>
  <p:tab title="Tab 2">
    Inhalt Tab 2
  </p:tab>
  <p:tab title="Tab 3" disabled="true">
    Inhalt Tab 3
  </p:tab>
</p:accordionPanel>
```

Standardmäßig werden alle Tabs von`p:accordionPanel`gerendert und im Browser auf- und zugeklappt. Bei komplexeren Tabs kann das aber durchaus die Ladezeit der Seite verlängern.*PrimeFaces*bietet daher die Möglichkeit, Tabs dynamisch über Ajax nachzuladen. Dazu muss lediglich im Tag`p:accordionPanel`das Attribut`dynamic`auf`true`gesetzt werden. Initial inaktive Tabs werden dann erst beim ersten Aktivieren nachgeladen. Um einen Tab bei jedem Aktivieren neu zu laden - zum Beispiel wegen dynamischer Inhalte -, muss zusätzlich das Attribut`cache`auf`false`gesetzt werden. `p:accordionPanel`erlaubt das Aufklappen mehrerer Tabs, wenn das Attribut`multiple`auf`true`gesetzt wird.

10.2.2.2 Calendar -- p:calendar

Eine Komponente zur komfortablen Auswahl eines Datums darf natürlich in keiner Komponentenbibliothek fehlen.*PrimeFaces*macht da keine Ausnahme und bietet zu diesem Zweck die Calendar-Komponente mit dem Tag`p:calendar`an.`p:calendar`hat zwei verschiedene Darstellungsmodi, die über das Attribut`mode`gesetzt werden. Im Modus`inline`wird die Komponente rein als Auswahlfeld für ein Datum ohne Texteingabemöglichkeit durch den Benutzer dargestellt. Im Modus`popup`wird die Komponente hingegen als Eingabefeld angezeigt. Das Auswahlfeld wird dann nur bei Bedarf als Pop-up eingeblendet. Abbildung**p:calendar im Browser**zeigt die gerenderte Ausgabe von`p:calendar`im Modus`popup`mit geöffnetem Auswahlfeld.

Abbildung:p:calendar im Browser

`p:calendar`verfügt über einige Attribute, um das Verhalten und die Darstellung der Komponente anzupassen. Im Modus`popup`lässt sich zum Beispiel über das Attribut`showOn`das Öffnen des Auswahlfeldes steuern: Mit`button`wird die Datumsauswahl über eine Schaltfläche neben dem Eingabefeld geöffnet, mit`focus`, wenn das Eingabefeld im Browser den Fokus erhält, und mit`both`in beiden Fällen. Die Titelleiste des Datumsauswahlfeldes zeigt standardmäßig den ausgewählten Monat und das Jahr an. Alternativ kann an dieser Stelle auch ein Navigator mit Auswahlfeldern für den Monat und das Jahr eingeblendet werden. Der Navigator wird durch das Setzen des Attributs`navigator`auf`true`aktiviert. Listing**p:calendar**zeigt das Tag für das Beispiel in Abbildung**p:calendar im Browser**im Modus`popup`mit Öffnen über eine Schaltfläche und aktiviertem Navigator. Im Attribut`pattern`wird außerdem noch das Datumsformat festgelegt.

```
<p:calendar value="#{bean.date}" mode="popup" navigator="true"
    showOn="button" pattern="dd.MM.yyyy"/>
```

Aus Sicht von JSF ist die Kalenderkomponente eine Eingabekomponente wie jede andere auch (die Klasse`Calendar`ist eine Subklasse von`HtmlInputText`) und unterstützt beliebige Konverter und Validatoren für den Datentyp`java.util.Date`. Wenn die Komponente keinen expliziten Konverter findet, wird der Wert intern konvertiert.

*PrimeFaces*liefert standardmäßig nur eine Lokalisierung für Englisch aus. Wenn Sie den Kalender wie in Abbildung**p:calendar im Browser**mit deutschen Texten (und Montag als ersten Tag) haben wollen, müssen Sie noch ein kurzes Stück JavaScript in Ihre Seite einbinden. Den entsprechenden Code für eine ganze Reihe von Sprachen finden Sie im*PrimeFaces*-Wiki**http://code.google.com/p/primefaces/wiki/PrimeFacesLocales**:.

10.2.2.3 DataTable -- p:dataTable

Ein weiterer Klassiker für Komponentenbibliotheken im JSF-Umfeld ist eine leistungsfähige Komponente zur tabellarischen Darstellung dynamischer Daten. Nachdem die DataTable-Komponente aus dem JSF-Standard Pagination nur halbherzig unterstützt und Features wie Sortierung oder Filterung komplett fehlen, besteht in diesem Bereich enormes Verbesserungspotenzial. In*PrimeFaces*gibt es dazu die DataTable-Komponente mit dem Tag`p:dataTable`, die all diese Features (und noch viel mehr) mitbringt. Abbildung**p:dataTable im Browser**zeigt die gerenderte Ausgabe von`p:dataTable`mit aktivierter Pagination und Sortierung für eine Liste von Personen. Wie Sie sehen, wird auch`p:dataTable`gemäß dem aktuellen Theme dargestellt.

Abbildung:p:dataTable im Browser

Die grundlegende Funktionsweise von`p:dataTable`entspricht jener der DataTable-Komponente aus dem JSF-Standard (wie in Abschnitt**Sektion: DataTable-Komponente**beschrieben). Analog zu`h:dataTable`verfügt auch`p:dataTable`über die Attribute`value`und`var`. In unserem Beispiel referenziert`value`eine Liste von Instanzen der Klasse`Person`mit den Eigenschaften`name`und`email`. Beim Einsatz von`p:dataTable`werden die Spalten der Tabelle mit dem Tag`p:column`definiert. Listing**p:dataTable mit Pagination und Sortierung**zeigt das XHTML-Fragment für die Tabelle aus Abbildung**p:dataTable im Browser**.

```
<p:dataTable var="person" value="#{bean.persons}"
    paginator="true" rows="5">
  <p:column headerText="Name" sortBy="#{person.name}">
    <h:outputText value="#{person.name}"/>
  </p:column>
  <p:column headerText="E-Mail" sortBy="#{person.email}">
    <h:outputText value="#{person.email}"/>
  </p:column>
</p:dataTable>
```

Das Hinzufügen eines Paginators für die seitenweise Darstellung großer Datenmengen ist mit`p:dataTable`äußerst einfach. Dazu muss nur das Attribut`paginator`auf den Wert`true`gesetzt werden. Die Anzahl der Zeilen pro Seite wird dann im Attribut`rows`definiert.*PrimeFaces*rendert standardmäßig einen Paginator mit Schaltflächen für bestimmte Seiten im Umfeld der aktuellen Seite und zur Navigation auf die erste, letzte, vorherige und nächste Seite. Das Umschalten zwischen den einzelnen Seiten erfolgt automatisch mittels Ajax.
Fall Sie den Paginator nicht wie in Abbildung**p:dataTable im Browser**am Anfang und am Ende der Tabelle haben wollen, müssen Sie lediglich das Attribut`paginatorPosition`auf den Wert`top`oder`bottom`setzen

(der Standardwert ist`both`).
Das Sortieren der Daten nach den Werten in einzelnen Spalten ist ähnlich einfach. Für jede Spalte kann im Attribut`sortBy`von`p:column`eine Eigenschaft der dargestellten Objekte als Sortierkriterium angegeben werden. In unserem Beispiel sind das die Eigenschaften`name`für die Spalte mit den Namen und`email`für die Spalte mit den Mailadressen. Intern benutzt die Komponente zum Vergleich zweier Werte einfach einen Java-Comparator. Sobald`sortBy`angegeben ist, rendert*PrimeFaces*ein Icon zum Sortieren der Daten, basierend auf den Werten der Spalte. Die Sortierung der Daten über einen Klick auf das Icon erfolgt ebenfalls mittels Ajax.
`p:dataTable`bietet noch eine Vielzahl weiterer Features wie das Filtern der Daten, die Selektion von Zeilen, das Verschieben von Spalten oder das Verändern der Spaltenbreite. Aus Platzgründen müssen wir für weitere Details allerdings auf die Dokumentation von*PrimeFaces*verweisen.

10.2.2.4 Menu -- p:menu

*PrimeFaces*stellt eine ganze Reihe von Komponenten für die verschiedensten Menüs zur Verfügung. Wir erläutern stellvertretend die Menu-Komponente mit dem Tag`p:menu`- die restlichen Menükomponenten funktionieren sehr ähnlich. Abbildung**p:menu im Browser**zeigt die gerenderte Ausgabe eines mit`p:menu`definierten Menüs mit diversen Menüeinträgen.

Abbildung:p:menu im Browser

Die Einträge eines Menüs werden in*PrimeFaces*mit dem Tag`p:menuitem`zu einer Menükomponente hinzugefügt. Dabei unterstützt`p:menuitem`Einträge mit unterschiedlichem Navigationsverhalten:

- Für einen Menüeintrag mit GET-Navigation im Stil von`h:link`wird das Ziel der Navigation im Attribut`outcome`eingetragen.
- Für einen Menüeintrag im Stil von`p:commandLink`wird wie gewohnt das Attribut`action`verwendet. Standardmäßig wird eine Ajax-Anfrage ausgelöst, wenn das Attribut`ajax`nicht auf`false`gesetzt ist. Im Ajax-Fall können im Attribut`update`IDs von Komponenten angegeben werden, die JSF neu rendern soll.
- Für einen Menüeintrag im Stil von`h:outputLink`muss im Attribut`url`eine komplette URL eingetragen werden.

Mit dem Tag`p:submenu`können zusätzlich mehrere Einträge gruppiert werden. Listing**p:menu mit verschiedenen Menüeinträgen**zeigt das XHTML-Fragment für das Menü aus Abbildung**p:menu im Browser**mit vier unterschiedlichen Typen von Einträgen.

```
<p:menu>
  <p:menuitem value="Anbieter" outcome="providerList"/>
  <p:menuitem value="Kunden" action="#{bean.goToCustomers}"
      ajax="false"/>
  <p:menuitem value="Aktualisieren" action="#{bean.update}"
      update="form"/>
  <p:submenu label="Extern">
    <p:menuitem value="JSF@Work" url="http://jsfatwork.irian.at"
        target="_blank"/>
  </p:submenu>
```

```
</p:menu>
```

Das Aufbauen eines Menüs mit den beiden Tags`p:menuitem`und`p:submenu`funktioniert für andere Menükomponenten wie`p:menubar`,`p:menuButton`oder`p:tabMenu`exakt gleich.

10.2.2.5 PanelGrid -- p:panelGrid

Die PanelGrid-Komponente mit dem Tag`p:panelGrid`bietet eine einfache Möglichkeit, andere Komponenten in Form einer Tabelle anzuordnen.`p:panelGrid`lässt sich auf die gleiche Art und Weise wie`h:panelGrid`aus dem JSF-Standard verwenden (Details dazu finden Sie in Abschnitt**Sektion: Panel-Komponenten**). Die*PrimeFaces*-Alternative verwendet für die Darstellung allerdings automatisch die Einstellungen vom aktuellen Theme.
Der größte Pluspunkt von`p:panelGrid`ist die Unterstützung von Tabellenzellen, die sich über mehrere Zeilen oder Spalten erstrecken. Dazu müssen die Zeilen allerdings mit dem Tag`p:row`und die Spalten mit`p:column`definiert werden. Abbildung**p:panelGrid im Browser**zeigt die gerenderte Ausgabe einer mit`p:panelGrid`definierten Tabelle.

Überschrift 1	Überschrift 2	
Zelle 1	Zelle 2	
	Zelle 3	Zelle 4

Abbildung:p:panelGrid im Browser

In Listing**p:panelGrid**finden Sie das XHTML-Fragment für die Tabelle aus Abbildung**p:panelGrid im Browser**. Die Zeilen und Spalten der Tabelle sind explizit mit den Tags`p:row`und`p:column`definiert - auch im Header-Facet von`p:panelGrid`. Damit sich eine Zelle innerhalb einer Zeile über mehrere Spalten erstreckt, muss im Tag`p:column`das Attribut`colspan`entsprechend gesetzt werden. Die Anzahl der Spalten sollte natürlich in jeder Zeile gleich sein. Analog erstreckt sich eine Zelle über mehrere Zeilen, wenn das Attribut`rowspan`entsprechend gesetzt wird.

```
<p:panelGrid>
  <f:facet name="header">
    <p:row>
      <p:column>Überschrift 1</p:column>
      <p:column colspan="2">Überschrift 2</p:column>
    </p:row>
  </f:facet>
  <p:row>
    <p:column rowspan="2">Zelle 1</p:column>
    <p:column colspan="2">Zelle 2</p:column>
  </p:row>
  <p:row>
    <p:column>Zelle 3</p:column>
    <p:column>Zelle 4</p:column>
  </p:row>
</p:panelGrid>
```

10.2.2.6 Rating -- p:rating

Mit`p:rating`können Bewertungen in Form von anklickbaren Sternen abgegeben werden. Der Wert der Rating-Komponente entspricht der Anzahl der ausgewählten Sterne. Die Komponente kann sowohl für die Eingabe als auch rein für die Ausgabe einer Bewertung dienen. Abbildung**p:rating im Browser**zeigt die gerenderte Ausgabe von zwei Rating-Komponenten: einmal ohne und einmal mit Icon zum Zurücksetzen des Werts auf 0.

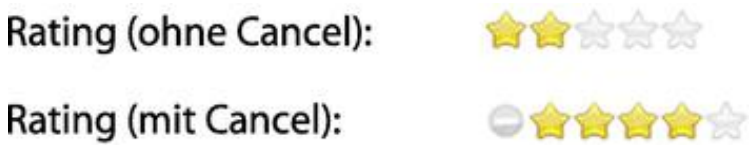


Abbildung:p:rating im Browser

Listing**Beispiele zu p:rating**zeigt einige Beispiele für das Tag`p:rating`. Die Anzahl der auswählbaren Sterne wird im Attribut`stars`angegeben (der Standardwert ist 5). Das Icon zum Zurücksetzen des Werts wird standardmäßig gerendert, wenn nicht das Attribut`cancel`explizit auf`false`gesetzt wird. Mit dem Setzen des Attributs`readonly`auf`true`kann die Komponente zur reinen Ausgabekomponente umfunktioniert werden.

```
<p:rating value="#{bean.value}" cancel="false" stars="5"/>
<p:rating value="#{bean.value}" stars="5"/>
<p:rating value="#{bean.value}" readonly="true" stars="5"/>
```

10.2.2.7 Slider -- p:slider

Die Slider-Komponente mit dem Tag`p:slider`stellt einen Schieberegler für Zahlenwerte zur Verfügung, ist selbst aber keine Eingabekomponente.`p:slider`muss daher immer mit einer Eingabekomponente wie`p:inputText`oder`h:inputHidden`verbunden werden. Abbildung**p:slider im Browser**zeigt zwei mögliche Einsatzszenarien: einmal in Kombination mit einem Eingabefeld und einmal in Kombination mit einer reinen Textausgabe des aktuellen Werts.

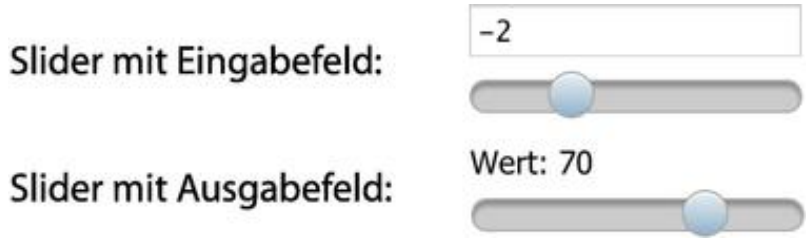


Abbildung:p:slider im Browser

Die ID der verbundenen Eingabekomponente wird im Attribut`for`von`p:slider`eingetragen. Jede Betätigung des Schiebereglers aktualisiert dann den Wert des Eingabefelds im Browser. Umgekehrt funktioniert das natürlich auch: Der Schieberegler wird jedes Mal angepasst, wenn der Benutzer den Wert im Eingabefeld ändert.
Listing**p:slider mit Eingabefeld**zeigt ein Beispiel für die Kombination von`p:slider`und`p:inputText`. Im Tag`p:slider`definieren dabei die Attribute`min-Value`und`max-Value`den minimalen und maximalen Wert des Schiebereglers. Der Regler lässt sich zwischen diesen beiden Werten in Schritten bewegen, die durch das Attribut`step`bestimmt sind. In unserem Beispiel ergibt das die möglichen Werte -5, -4, ..., 5.

```
<p:inputText id="value" value="#{bean.value}"/>
<p:slider for="value" minValue="-5" maxValue="5" step="1"/>
```

Wenn Sie die direkte Eingabe des Werts durch den Benutzer verhindern wollen, können Sie`p:slider`mit`h:inputHidden`verbinden. In diesem Fall kann der aktuelle Wert des Schiebereglers als Text ausgegeben werden. Dazu muss lediglich die ID einer Ausgabekomponente wie`h:outputText`im Attribut`display`eingetragen werden. Listing**p:slider mit Textausgabe**zeigt ein entsprechendes Beispiel.

```
<h:inputHidden id="value" value="#{bean.value}"/>
<h:outputText value="Wert: "/>
<h:outputText id="out" value="#{bean.value}"/>
<p:slider for="value" display="out"
    minValue="0" maxValue="100"/>
```

10.2.2.8 Spinner -- p:spinner

Die Spinner-Komponente mit dem Tag`p:spinner`ist eine Eingabekomponente für Zahlen mit zwei Schaltflächen zum Erhöhen und Reduzieren des Zahlenwerts. Abbildung**p:spinner im Browser**zeigt die gerenderte Ausgabe eines Beispiels.

Abbildung:p:spinner im Browser

In Listing**p:spinner**finden Sie ein Beispiel für das Tag`p:spinner`. Das Attribut`stepFactor`definiert den Betrag, der addiert beziehungsweise subtrahiert wird. Mit den Attributen`min`und`max`lassen sich eine untere und obere Grenze für den Zahlenwert definieren.

```
<p:spinner value="#{bean.value}"
    stepFactor="2" min="1" max="20"/>
```

10.3 Themes

Das visuelle Erscheinungsbild einer*PrimeFaces*-Applikation lässt sich mit sogenannten Themes sehr einfach ändern. Ein Theme definiert das Aussehen einer Anwendung und bestimmt unter anderem das Farbschema oder die verwendeten Zeichensätze.*PrimeFaces*bietet in der aktuellen Version über 30 verschiedene Themes mit etlichen Farbkombinationen für die unterschiedlichsten Geschmäcker an. Eine Übersicht aller verfügbaren Themes finden Sie unter**http://www.primefaces.org/themes.html**. Abbildung**Ausgewählte PrimeFaces-Themes**zeigt einen kleinen Vorgeschmack.

Abbildung:Ausgewählte PrimeFaces-Themes

*PrimeFaces*benutzt standardmäßig das Theme mit dem Namen*Aristo*. Alle weiteren Themes sind nicht im Standardumfang enthalten und müssen als Jar-Datei in die Applikation eingebunden werden. Die einzelnen Themes stehen als Download auf der*PrimeFaces*-Seite und als Maven-Abhängigkeit im*PrimeFaces*-Repository zur Verfügung. Listing**Maven-Abhängigkeit für PrimeFaces-Theme**zeigt zum Beispiel die Abhängigkeit für das Theme*Afterdark*.

```
<dependency>
  <groupId>org.primefaces.themes</groupId>
  <artifactId>afterdark</artifactId>
  <version>1.0.9</version>
</dependency>
```

Das aktuell von*PrimeFaces*verwendete Theme wird über den Kontextparameter`primefaces.THEME`Ein

der`web.xml`definiert. Als Wert des Parameters kommt der kleingeschriebene Theme-Name zum Einsatz. Mit dem Kontextparameter in Listing**Kontextparameter für PrimeFaces-Theme**wird zum Beispiel das Theme*Afterdark*aktiviert.

```
<context-param>
  <param-name>primefaces.THEME</param-name>
  <param-value>afterdark</param-value>
</context-param>
```

*PrimeFaces*unterstützt auch das dynamische Umschalten zwischen Themes zur Laufzeit. Dazu muss lediglich im Kontextparameter`primefaces.THEME`eine Value-Expression angegeben werden, die den Theme-Name als String zurückliefert. Listing**Dynamischer Kontextparameter für PrimeFaces-Theme**zeigt ein Beispiel, in dem die Eigenschaft`theme`der Bean`preferences`referenziert wird. In Listing**CDI-Bean für PrimeFaces-Theme**finden Sie die dazu passende CDI-Bean.

```
<context-param>
  <param-name>primefaces.THEME</param-name>
  <param-value>#{preferences.theme}</param-value>
</context-param>
```

```
@Named @SessionScoped
public class Preferences implements Serializable {
  public String getTheme() {
    return "afterdark";
  }
}
```

Wenn Sie trotz der großen Auswahl kein passendes Theme für Ihre Applikation finden, können Sie mit relativ geringem Aufwand ein eigenes erstellen.

10.3.1 Benutzerdefinierte Themes erstellen

Das Erstellen eigener Themes gestaltet sich mit*PrimeFaces*relativ einfach. Sie können dazu das Online-Tool*jQuery ThemeRollerThemeRoller*ist unter**http://jqueryui.com/themeroller/**verfügbar.:verwenden.*ThemeRoller*ist ein einfach zu bedienender Editor mit eingebauter Vorschau für*jQuery UI*-Themes, die sich auch für den Einsatz in eignen. Abbildung**JQuery-ThemeRoller**zeigt den*ThemeRoller*in Aktion.

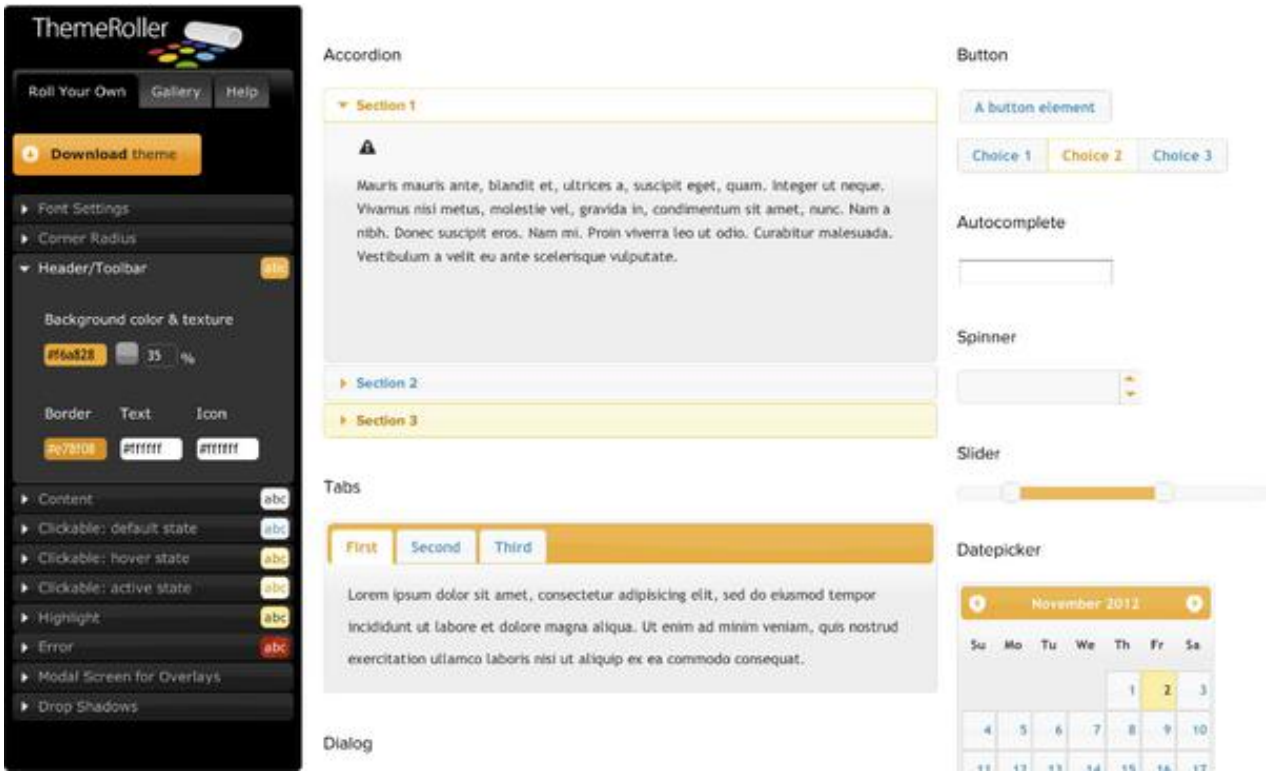


Abbildung:JQuery-ThemeRoller

*ThemeRoller*stellt im Tab`Gallery`eine ganze Reihe vorgefertigter Themes als Grundlage für eigene Entwürfe zur Verfügung. Einige der dort aufgelisteten Themes finden sich auch in*PrimeFaces*wieder. Nachdem Sie das Theme gemäß Ihren Wünschen gestaltet haben, müssen Sie es über einen Klick auf die Schaltfläche als Zip-Datei downloaden. In der daraufhin angezeigten Downloadseite können Sie das Auswahlfeld`Toggle all`deselektieren und im Eingabefeld`Theme Folder Name`den Namen des Theme-Verzeichnisses in der Zip-Datei angeben.
In der von*ThemeRoller*erzeugten Zip-Datei finden Sie im Verzeichnis`css`das Verzeichnis mit dem auf der Downloadseite angegebenen Namen. Dort liegt zum einen die für uns relevante CSS-Datei in einer lesbaren Variante mit der Endung`custom.css`und in einer optimierten Variante mit der Endung`custom.min.css`und zum anderen das Verzeichnis`images`mit Bildern und Icons. Aus diesen Daten erstellen wir jetzt das Theme mit dem Namen`mygourmet`.
*PrimeFaces*lädt die Daten zu einem Theme als JSF-Ressourcen (Details zur Ressourcenverwaltung finden Sie in Kapitel**Kapitel: Verwaltung von Ressourcen**). Per Konvention liegt jedes Theme in einer eigenen Bibliothek, deren Name aus dem Präfix`primefaces-`und dem Theme-Namen zusammengesetzt wird - für unser Beispiel ist das`primefaces-mygourmet`.
Im ersten Schritt legen wir daher das Verzeichnis der Bibliothek`primefaces-mygourmet`unter`/resources`im Wurzelverzeichnis der Webapplikation oder unter`/META-INF/resources`in einer Jar-Dateien an. Dorthin kopieren wir im nächsten Schritt die CSS-Datei mit der Endung`custom.css`und das Verzeichnis`images`aus der von*ThemeRoller*erstellten Zip-Datei. Die CSS-Datei muss unbedingt auf`theme.css`umbenannt werden, damit*PrimeFaces*die Ressource auflösen kann.
Im letzten Schritt müssen in der CSS-Datei`theme.css`noch alle URL-Referenzen für Bilder auf Value-Expressions umgebaut werden. Damit wird sichergestellt, dass die Bilder von JSF als Ressourcen geladen werden. Die Referenz`url(images/icons.png)`muss zum Beispiel folgendermaßen geändert werden:

```
url("#{resource['primefaces-mygourmet:images/icons.png']}")
```

Wie Sie sehen, ist es ohne Probleme möglich, in einer CSS-Datei, die von JSF als Ressource geladen wird, Value-Expressions zu verwenden. Im Beispiel von oben wird damit die Ressource`images/icons.png`aus der Bibliothek`primefaces-mygourmet`referenziert. Diese Art der Referenzierung haben wir bereits in Abschnitt**Sektion: Ressourcen im Einsatz**vorgestellt.
Das Theme ist damit fertig und einsatzbereit. Sie müssen nur den Kontextparameter`primefaces.THEME`auf den Wert`mygourmet`setzen.

10.4 PrimeFaces und Ajax

*PrimeFaces*verfügt über eine sehr leistungsfähige Integration von Ajax, die auf der standardisierten Ajax-Integration von JSF 2 basiert. Die in Kapitel**Kapitel: Ajax und JSF**vorgestellten Grundprinzipien gelten somit auch beim Einsatz von*PrimeFaces*. Wie nicht anders zu erwarten, bietet*PrimeFaces*allerdings auch im Ajax-Bereich einige über den JSF-Standard hinausgehende Features an. Details dazu finden Sie in Abschnitt**[Sektion: Erweiterungen im Vergleich zu Standard-JSF]**.
Bei zahlreichen*PrimeFaces*-Komponenten ist das Ajax-Verhalten bereits eingebaut. Dazu zählen neben Komponenten wie`p:dataTable`und`p:accordionPanel`auch Komponenten mit spezifischer Ajax-Funktionalität wie`p:ajaxStatus`und`p:poll`. In Abschnitt**[Sektion: Ajax-Komponenten]**finden Sie einige dieser Ajax-Komponenten. Eine Reihe von Komponenten verfügt außerdem über spezielle Attribute, um direkt Ajax-Anfragen zum Aktualisieren der Seite auszulösen oder um auf Ajax-Anfragen zu reagieren. Abschnitt**[Sektion: Komponenten mit Ajax-Unterstützung]**gibt dazu einen kurzen Überblick.

10.4.1 Erweiterungen im Vergleich zu Standard-JSF

Das Pendant zu`f:ajax`in*PrimeFaces*ist`p:ajax`. Die Handhabung von`f:ajax`und`p:ajax`sind bis auf ein paar kleine Abweichungen identisch. Der größte Unterschied zwischen den beiden Tags ist die Benennung der Attribute:`execute`und`render`von`f:ajax`werden zu`process`und`update`in`p:ajax`.
Das Beispiel in Listing**f:ajax versus p:ajax**demonstriert den gemeinsamen Einsatz von`f:ajax`und`p:ajax`anhand zweier Eingabekomponenten mit Ajax-Verhalten. Bei beiden Komponenten löst das Ereignis`onChange`eine Ajax-Anfrage aus, um das Textfeld mit der ID`out`neu zu rendern. Im ersten Fall wurde das Ajax-Verhalten allerdings mit`f:ajax`und im zweiten Fall mit`p:ajax`zur Komponente hinzugefügt - das Resultat bleibt gleich.

```
<p:inputText value="#{bean.first}">
  <f:ajax event="change" execute="@this" render="out"/>
</p:inputText>
<p:inputText value="#{bean.last}">
  <p:ajax event="change" process="@this" update="out"/>
</p:inputText>
<h:outputText id="out" value="#{bean.first} #{bean.last}"/>
```

Bei Ajax-Anfragen benachrichtigt*PrimeFaces*standardmäßig eventuell vorhandene Ajax-Status-Komponenten mit dem Tag`p:ajaxStatus`. Ist das nicht gewünscht, muss in`p:ajax`lediglich das Attribut`global`auf den Wert`false`gesetzt werden. Details zu`p:ajaxStatus`finden Sie in Abschnitt**[Sektion: Ajax-Komponenten]**.
Per Definition werden in JSF bei jeder Ajax-Anfrage die Daten des gesamten Formulars an den Server geschickt. Das gilt zum Beispiel auch dann, wenn von 100 Eingabekomponenten nur eine einzige im partiellen Lebenszyklus ausgeführt wird. Standardmäßig zeigt*PrimeFaces*dasselbe Verhalten, bietet aber die Möglichkeit, hier einzugreifen. Wird in`p:ajax`das Attribut`partialSubmit`auf`true`gesetzt, werden nur die Daten der für die Ajax-Anfrage relevanten Komponenten an den Server geschickt. Dieses Verhalten lässt sich auch global aktivieren, indem in der`web.xml`der Kontextparameter`primefaces.SUBMIT`auf den

Wert`partial`gesetzt wird.

10.4.2 Ajax-Komponenten

In diesem Abschnitt stellen wir die Ajax-Komponenten`p:ajaxStatus`und`p:poll`vor.

10.4.2.1 AjaxStatus -- p:ajaxStatus

Die AjaxStatus-Komponente mit dem Tag`p:ajaxStatus`ermöglicht das Einblenden von Statusmeldungen (oder komplexeren Komponenten) für verschiedene Ereignisse während einer Ajax-Anfrage in*PrimeFaces*. Die Komponente unterstützt ein Facet für jedes mögliche Ereignis (eine vollständige Liste finden Sie in der Dokumentation zu*PrimeFaces*). Tritt das entsprechende Ereignis ein, wird der Inhalt des Facets mit dem gleichen Namen eingeblendet.
Im Beispiel in Listing**p:ajaxStatus**wird beim Start einer Ajax-Anfrage (Ereignis`start`) der Text`Loading`eingeblendet. Nach dem Beenden der Ajax-Anfrage (Ereignis`complete`) wird ein leerer Text angezeigt.

```
<p:ajaxStatus>
  <f:facet name="start">
    <h:outputText value="Loading"/>
  </f:facet>
  <f:facet name="complete">
    <h:outputText value=""/>
  </f:facet>
</p:ajaxStatus>
```

10.4.2.2 Poll -- p:poll

Die Poll-Komponente mit dem Tag`p:poll`erlaubt das periodische Senden von Ajax-Anfragen. Das Intervall zwischen zwei Anfragen wird dabei im Attribut`interval`als Sekundenwert angegeben. Die restlichen Attribute zum Steuern der Ajax-Anfrage wie etwa`process`,`update`oder`global`sind identisch zu`p:ajax`. Listing**p:poll**zeigt ein Beispiel für`p:poll`, in dem alle fünf Sekunden eine Ajax-Anfrage gesendet wird.

```
<p:poll interval="5" process="@none" update="time"
    listener="#{bean.touch}" global="false"/>
<h:outputText id="time" value="#{bean.timeStamp}"/>
```

Während der Abarbeitung dieser Anfrage am Server wird zuerst die im Attribut`listener`referenzierte Listener-Methode aufgerufen. Listing**Listener-Methode für p:poll**zeigt die Details der Methode. Anschließend rendert JSF die Komponente mit der ID`time`neu und aktualisiert die Ausgabe am Client. Nachdem`global`auf`false`gesetzt ist, bleibt die Benachrichtigung einer eventuell vorhandenen Ajax-Status-Komponente aus.

```
private Date timestamp = new Date();

public void touch() {
  timestamp = new Date();
}
```

10.4.3 Komponenten mit Ajax-Unterstützung

Bei einigen Komponenten wie`p:commandButton`oder`p:commandLink`sind die Attribute von`p:ajax`bereits integriert.`p:commandButton`und`p:commandLink`senden sogar standardmäßig Ajax-Anfragen, wenn nicht explizit das Attribut`ajax`auf`false`gesetzt wird.
Listing**Ajax-Anfrage mit p:commandLink**zeigt ein Beispiel mit`p:commandLink`. Ein Klick auf diesen Link löst ohne spezielle Vorkehrungen automatisch eine Ajax-Anfrage aus. Durch die Angaben in den Attributen`process`und`update`werden serverseitig die Komponenten mit den IDs`first`und`last`ausgeführt und die Komponente mit der ID`out`wird neu gerendert.

```
<p:inputText id="first" value="#{bean.first}"/>
<p:inputText id="last" value="#{bean.last}"/>
<p:commandLink value="Aktualisieren"
    process="first last" update="out"/>
<h:outputText id="out" value="#{bean.first} #{bean.last}"/>
```

Für Inhalte, die bei jeder Ajax-Anfrage aktualisiert werden müssen, bietet sich die OutputPanel-Komponente mit dem Tag`p:outputPanel`an. Die Komponente wird inklusive aller Kindkomponenten automatisch bei jeder Ajax-Anfrage neu gerendert, wenn das Attribut`autoUpdate`auf`true`gesetzt wird.

10.5 MyGourmet 18: PrimeFaces

Das Beispiel*MyGourmet 18*basiert auf dem Vorgängerbeispiel*MyGourmet 17*und bietet auch denselben Funktionsumfang. Allerdings haben wir in*MyGourmet 18*die komplette Anwendung auf*PrimeFaces*umgestellt. Das Ziel der Migration war eine Anwendung, die in Funktionalität und Aussehen*MyGourmet 17*sehr ähnlich ist, die aber das volle Potenzial von*PrimeFaces*zur Verfügung hat.
Die Umstellung von*MyGourmet 17*auf*PrimeFaces*erfolgt in mehreren Schritten, die in den folgenden Abschnitten näher erklärt werden. Zuerst widmet sich Abschnitt**[Sektion: Integration von*PrimeFaces*]**der Integration von*PrimeFaces*. Anschließend zeigt Abschnitt**[Sektion: Umstellung auf*PrimeFaces*-Komponenten]**in groben Zügen die Umstellung der einzelnen Seiten. Zu guter Letzt finden Sie in Abschnitt**[Sektion: Benutzerdefiniertes Theme]**noch Hinweise zum benutzerdefinierten Theme für*MyGourmet*.

10.5.1 Integration von PrimeFaces

Die Integration von*PrimeFaces*in*MyGourmet*gestaltet sich äußerst einfach und umfasst das Eintragen der entsprechenden Abhängigkeit und des dazu notwendigen Repositories in die`pom.xml`. Details dazu finden Sie in Abschnitt**[Sektion: PrimeFaces -- ein Überblick]**und im Code des Beispiels.
Wie in Abschnitt**[Sektion: Auswahl einiger*PrimeFaces*-Komponenten]**beim Tag`p:calendar`bereits erwähnt, liefert*PrimeFaces*standardmäßig nur eine Lokalisierung für Englisch aus. Das Hinzufügen weiterer

Sprachen ist allerdings kein Problem - ein kurzes Stück JavaScript aus dem*PrimeFaces*-Wiki**http://code.google.com/p/primefaces/wiki/PrimeFacesLocales**:genügt.
Da*MyGourmet*auch auf Deutsch funktionieren soll, haben wir den dazu notwendigen Code in der Ressource`primeFacesLoc.js`in der Bibliothek`scripts`abgelegt und im Template`template.xhtml`mit dem Tag`h:outputScript`eingebunden. Damit ist der Code in allen Seiten verfügbar und der Umstellung der einzelnen Seiten auf*PrimeFaces*-Komponenten steht nichts mehr im Weg.

10.5.2 Umstellung auf PrimeFaces-Komponenten

Die Umstellung der einzelnen Seiten der Anwendung gestaltet sich relativ unspektakulär. Im ersten Schritt haben wir wie in Abschnitt**[Sektion: Erweiterte Standardkomponenten]**beschrieben alle Standardkomponenten - soweit vorhanden - mit erweiterten Alternativen aus*PrimeFaces*ersetzt. Bei vielen Komponenten wie`outputLabel`oder`inputText`genügt es, dazu das Präfix vonhaufpzu ändern - vorausgesetztpist korrekt mit dem Namensraum`http://primefaces.org/ui`verknüpft.
Manche Komponenten verlangen nach zusätzlichen Anpassungen.`p:commandButton`und`p:commandLink`senden zum Beispiel standardmäßig Ajax-Anfragen. Wenn dieses Verhalten nicht gewünscht ist, muss explizit das Attribut`ajax`auf`false`gesetzt werden. Im Ajax-Fall kann dafür aber auf`f:ajax`oder`p:ajax`verzichtet werden, da die Tags bereits die entsprechenden Attribute mitbringen (siehe Abschnitt**[Sektion: Komponenten mit Ajax-Unterstützung]**).
Die Umstellung von`h:dataTable`beziehungsweise`mc:dataTable`auf`p:dataTable`benötigt geringfügig mehr Aufwand. Dafür bietet`p:dataTable`aber auch einige zusätzliche Features, wie Listing**MyGourmet 18: p:dataTable in customerList.xhtml**anhand eines Ausschnitts aus der Seite`customerList.xhtml`zeigt. Details zu`p:dataTable`finden Sie in Abschnitt**[Sektion: Auswahl einiger*PrimeFaces*-Komponenten]**.

```
<p:dataTable value="#{customerListBean.customerList}" var="cust"
    paginator="true" rows="10" paginatorPosition="bottom"
    emptyMessage="#{msgs.customers_empty}">
  <p:column headerText="#{msgs.name}" sortBy="#{cust.fullName}">
    <p:commandLink value="#{cust.fullName}" ajax="false"
        action="#{customerBean.showCustomer(cust.id)}"/>
  </p:column>
  <p:column headerText="#{msgs.email}" sortBy="#{cust.email}">
    <h:outputText value="#{cust.email}"/>
  </p:column>
  <p:column>
    <p:commandLink value="#{msgs.delete}" update="@form"
        action="#{customerListBean.deleteCustomer(cust)}"/>
  </p:column>
</p:dataTable>
```

Ansonsten haben wir noch an einigen Stellen Standard- oder Kompositkomponenten mit*PrimeFaces*-Komponenten ersetzt. In`showProvider.xhtml`finden Sie zum Beispiel`p:rating`, in`editProvider.xhtml``p:spinner`und in`editCustomer.xhtml``p:calendar`. Im Template`customerTemplate.xhtml`wird statt der Kompositkomponente`mc:ajaxStatus`die flexiblere Alternative`p:ajaxStatus`verwendet.
In der linken Seitenleiste`leftSideBar.xhtml`gibt es ebenfalls einige Änderungen. Anstatt der ersten Kompositkomponente`mc:panelBox`mit den`h:link`-Tags zur Navigation kommt`p:menu`zum Einsatz. Die einzelnen Einträge des Menüs werden mit`p:menuitem`hinzugefügt, wobei das Ziel der Navigation im Attribut`outcome`definiert wird. So ist gewährleistet, dass die Navigation weiter über GET-Anfragen läuft.
Das zweite`mc:panelBox`-Tag muss`p:panel`weichen. Listing**MyGourmet 18: leftSideBar.xhtml**zeigt den

relevanten Inhalt der Seitenleiste (ohne lokalisierte Labels).

```
<p:menu>
  <p:submenu label="Menü">
    <p:menuitem outcome="providerList" value="Anbieterliste"/>
    <p:menuitem outcome="customerList" value="Kundenliste"/>
  </p:submenu>
</p:menu>
<p:panel header="Neuigkeiten">
  <p>MyGourmet - jetzt mit Facelets und Templating</p>
</p:panel>
```

Damit sich`p:menu`harmonisch in die Seitenleiste einfügt, muss das Styling noch etwas angepasst werden.*PrimeFaces*definiert dazu für jede Komponente eine ganze Reihe von CSS-Selektoren (eine Übersicht für jede Komponente finden Sie in der Dokumentation). Die gerenderte Ausgabe der Menükomponente kann unter anderem über die CSS-Klasse`.ui-menu`angepasst werden. Listing**MyGourmet 18: Styling für p:menu**zeigt zum Beispiel eine CSS-Regel, um das Menü innerhalb eines Elements mit der ID`left_sidebar`(die ID der Seitenleiste) anzupassen. Alle notwendigen Anpassungen finden Sie in`mygourmet.css`.

```
#left_sidebar .ui-menu {
  width: 134px;
  padding: 2px;
  margin-bottom: 5px;
}
```

10.5.3 Benutzerdefiniertes Theme

Als letzten Schritt haben wir mit*ThemeRoller*ein benutzerdefiniertes Theme für*MyGourmet*erstellt und in die Anwendung integriert. Alles Wissenswerte zu diesem Thema inklusive einer kurzen Anleitung finden Sie in Abschnitt**[Sektion: Themes]**.

11 Faces-Flows

In Webanwendungen gibt es immer wieder Abläufe, die eine abgeschlossene Einheit bilden, sich aber über mehrere Ansichten erstrecken. Denken Sie zum Beispiel an die Registrierung eines Benutzers, bei der in einer ersten Seite die Login-Daten und in einer zweiten Seite Daten zur Person abgefragt werden. Nachdem dieser Vorgang in sich abgeschlossen ist und immer gleich abläuft, liegt die Idee nahe, daraus ein wiederverwendbares Modul zu erstellen. Genau an dieser Stelle kommen die sogenannten Flows ins Spiel.
Bei Flows handelt es sich um eine Gruppierung mehrerer Seiten, die in einer bestimmten Reihenfolge miteinander verknüpft sind. Ein Flow kann von außen über einen definierten Einstiegspunkt gestartet werden. Einmal gestartet werden die Seiten des Flows bis zum Erreichen eines Ausstiegspunkts in der intern definierten Reihenfolge abgearbeitet. Abbildung**Checkout-Flow**zeigt als Beispiel einen Flow zum Auschecken eines Warenkorbs.

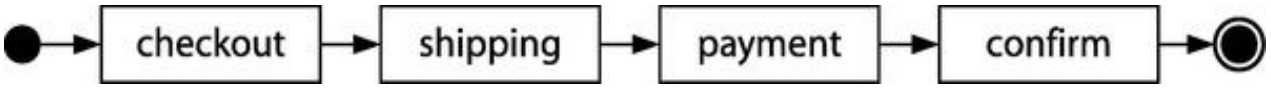


Abbildung:Checkout-Flow

Der Flow besteht aus den Seiten`checkout`,`shipping`,`payment`und`confirm`. Die Seite`checkout`ist dabei als Startseite definiert und bildet den Einstiegspunkt in den Flow. Von der Seite`confirm`aus kann der Flow beendet werden.
Erweiterungen wie*Spring Web Flow*oder*ADF Task Flow*sermöglichen bereits seit mehreren Jahren den Einsatz von Flows in Kombination mit JSF. Mit Version 2.2 haben Flows unter dem Namen "Faces-Flows" auch ihren Weg in die JSF-Spezifikation gefunden.
Im restlichen Kapitel widmen wir uns dem Einsatz von Faces-Flows mit JSF 2.2. Zum Einstieg in das Thema zeigen wir in Abschnitt**[Sektion: Ein erstes Beispiel]**ein erstes, einfaches Beispiel. Im Anschluss daran werfen wir in Abschnitt**[Sektion: Definition von Flows]**einen genaueren Blick auf die Definition von Flows mit XML und mit Java. Zur optimalen Verwaltung von Daten innerhalb von Flows liefert JSF auch gleich den passenden Scope für Managed-Beans mit, wie Abschnitt**[Sektion: Flow-Scope]**zeigt. In Abschnitt**[Sektion: Faces-Flows in Jar-Dateien]**folgt eine kurze Anleitung zum Verpacken von Flows in Jar-Dateien. Abschließend zeigt Abschnitt**[Sektion: Beispiel*Faces-Flows*]**noch ein etwas umfangreicheres Beispiel.

11.1 Ein erstes Beispiel

Ein Faces-Flow besteht aus mehreren Knoten, die über die interne Navigation des Flows miteinander verbunden sind. JSF definiert verschiedene Arten von Knoten - für das erste Beispiel sind allerdings nur Knoten für Seiten und Knoten zum Beenden des Flows von Interesse.
Im ersten Beispiel werden wir den Flow mit der ID`flow1`erstellen. Dieser Flow besteht aus zwei Seiten, die jeweils einen kurzen Text und einen Link zur Navigation auf die nächste Seite enthalten. Aus Sicht der Funktionalität gibt dieses Beispiel nicht viel her - es eignet sich aber sehr gut zur Demonstration der grundlegenden Struktur und der Funktionsweise von Faces-Flows.
Die Definition unseres Flows mit der ID`flow1`erfolgt im gleichnamigen Verzeichnis innerhalb der Webapplikation. Darin legen wir die leere Konfigurationsdatei`flow1-flow.xml`an - JSF kann den Flow ansonsten nicht korrekt auflösen. Bei unserem Beispiel handelt es sich um einen sogenannten impliziten Flow, der auf einigen Konventionen von JSF aufbaut und dadurch mit einer leeren Konfigurationsdatei auskommt. In Abschnitt**[Sektion: Definition von Flows]**füllen wir die Konfiguration dann mit Leben.
Nachdem das Grundgerüst steht, können wir uns um die Seiten des Flows kümmern. Dazu legen wir im

Flow-Verzeichnis die Dateien`flow1.xhtml`und`page2.xhtml`an. Per Konvention erstellt JSF für jede XHTML-Datei im Flow-Verzeichnis einen Seitenknoten, dessen ID dem Dateinamen ohne der Erweiterung`.xhtml`entspricht. JSF definiert automatisch den Knoten mit der selben ID wie der Flow selbst zum Startknoten des Flows. Zusätzlich definiert JSF einen Knoten zum Beenden des Flows, dessen ID sich aus der Flow-ID und der Zeichenkette`-return`zusammensetzt. Unser Flow besteht somit aus den Knoten`flow1`,`page2`und`flow1-return`.
Abbildung**Struktur des Flows flow1**zeigt den Inhalt unseres Flows mit dem Namen`flow1`im Wurzelverzeichnis der Webapplikation.

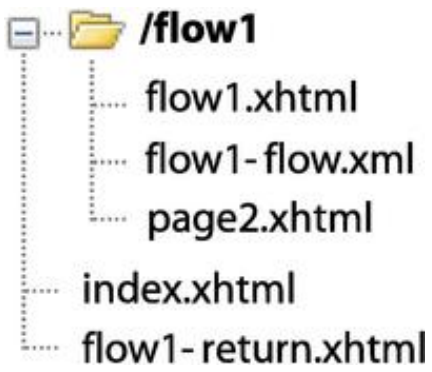


Abbildung:Struktur des Flows flow1

JSF 2.2 integriert Faces-Flows nahtlos in das bestehende Navigationssystem. Zum Starten eines Flows reicht es aus, dessen ID bei der Navigation anzugeben. Dazu gibt es grundsätzlich zwei Varianten. Soll der Flow über eine GET-Anfrage gestartet werden, muss seine ID im Attribut`outcome`von`h:link`oder`h:button`eingetragen werden. Soll der Flow hingegen über eine POST-Anfrage gestartet werden, muss seine ID im Attribut`action`von`h:commandLink`oder`h:commandButton`angegeben werden. Im zweiten Fall kann die Flow-ID auch der Rückgabewert einer Action-Methode sein.
Unser Flow mit der ID`flow1`kann zum Beispiel mit einem der folgenden Tags gestartet werden:

```
<h:link value="Start flow1" outcome="flow1"/>
<h:commandLink value="Start flow1" action="flow1"/>
```

JSF navigiert beim Starten des Flows auf die im Startknoten angegebene Seite`flow1.xhtml`. Für die Navigation innerhalb des Flows werden einfach die IDs der Knoten benutzt. Zum Verlassen des Flows auf der Seite`page2.xhtml`muss die ID`flow1-return`des Knotens zum Beenden benutzt werden. Unser Flow kann dann zum Beispiel mit einem der folgenden Tags beendet werden.

```
<h:button value="Flow beenden" outcome="flow1-return"/>
<h:commandButton value="Flow beenden" action="flow1-return"/>
```

Nach dem Beenden navigiert JSF automatisch auf die Seite`/flow1-return.xhtml`.

11.2 Definition von Flows

Das volle Funktionsspektrum der Faces-Flows lässt sich nur mit einer expliziten Definition des Flows entfalten. In dieser Definition wird der Flow mit seinen Knoten konfiguriert. Eine Übersicht aller verfügbaren Knotentypen finden Sie in Abschnitt**[Sektion: Typen von Flow-Knoten]**. Die Definition von Flows kann wahlweise mit XML (siehe Abschnitt**[Sektion: Definition mit XML]**) oder mit Java (siehe Abschnitt**[Sektion: Definition mit Java]**) erfolgen.

In diesem Abschnitt konzentrieren wir uns auf Flows, die direkt in der Webapplikation definiert sind. Faces-Flows können aber auch in einer Jar-Datei definiert werden. Details dazu finden Sie etwas später in Abschnitt**[Sektion: Faces-Flows in Jar-Dateien]**.

11.2.1 Typen von Flow-Knoten

JSF 2.2 bietet folgende Knotentypen für Faces-Flows:

- *View-Knoten*:
  Knoten für eine Facelets-Seite innerhalb eines Flows
- *Return-Knoten*:
  Knoten zum Beenden eines Flows
- *Flow-Call-Knoten*:
  Knoten zum Starten eines Flows aus einem anderen Flow heraus. Hier besteht die Möglichkeit, Parameter an den aufgerufenen Flow zu übergeben.
- *Method-Call-Knoten*:
  Knoten zum Aufrufen einer Managed-Bean-Methode, deren Rückgabewert - falls vorhanden - die weitere Navigation bestimmt.
- *Switch-Knoten*:
  Knoten zur dynamischen Navigation, basierend auf einer Liste von Value-Expressions mit zugeordneten Navigationszielen. JSF verwendet die erste Value-Expression mit dem Wert`true`zur Navigation.

11.2.2 Definition mit XML

Zur Definition eines Flows mit XML kommt die bereits bekannte XML-Datei im Flow-Verzeichnis zum Einsatz. Der Name dieser Datei setzt sich aus der Flow-ID und der Erweiterung`-flow.xml`zusammen. Syntaktisch handelt es sich bei dieser Konfigurationsdatei um eine stark eingeschränkte`faces-config.xml`. Das Wurzelelement`faces-config`darf nur ein einziges`flow-definition`-Element mit der Definition des Flows enthalten.
Im Element`flow-definition`muss im Attribut`id`die Flow-ID angegeben werden. Innerhalb dieses Elements werden die Knoten des Flows definiert. JSF 2.2 sieht dazu für jeden Knotentyp ein eigenes XML-Element vor. Als erstes Beispiel finden Sie in Listing**Flow flow1 mit XML**eine Konfiguration für unseren Flow`flow1`aus Abschnitt**[Sektion: Ein erstes Beispiel]**.

```
<faces-config 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"
    version="2.2">
  <flow-definition id="flow1">
    <flow-return id="flow1-return">
```

```
      <from-outcome>/index</from-outcome>
    </flow-return>
  </flow-definition>
</faces-config>
```

Die gezeigte Konfiguration definiert im Element`flow-return`einen Return-Knoten mit der ID`flow1-return`. Das Navigationsziel für diesen Knoten wird im Element`from-outcome`auf den Wert`/index`festgelegt. Wird der Flow über diesen Knoten beendet, benutzt JSF den Wert`/index`, um eine Seite außerhalb des Flows aufzulösen. Ohne explizite Navigationsregeln ergibt sich daraus die Seite`/index.xhtml`- mit einer entsprechenden Navigationsregel in der`faces-config.xml`kann dieser Wert aber auf eine beliebige Seite abgebildet werden.
Auch mit einer expliziten Konfiguration bleiben die in Abschnitt**[Sektion: Ein erstes Beispiel]**vorgestellten Konventionen bestehen. Die über die Konventionen definierten Knoten des Flows bleiben also erhalten. In der Konfiguration definierte Knoten haben aber immer Vorrang und überschreiben per Konvention definierte Knoten mit der gleichen ID.

Tipp: Achten Sie darauf, für das Flow-Verzeichnis, für den Namen der Konfigurationsdatei und für die ID in der Konfiguration immer die selbe Flow-ID zu verwenden.

In Listing**Komplette Definition von Flow flow1 mit XML**finden Sie die komplette Definition unseres Flows inklusive aller View-Knoten und der Definition des Startknotens. Bei den View-Knoten wird die View-ID der anzuzeigenden Seite inklusive des Flow-Verzeichnisses im Element`vdl-document`eingetragen. Der Startknoten des Flows wird im Element`start-node`über seine ID bestimmt.

```
<flow-definition id="flow1">
  <start-node>start</start-node>
  <view id="start">
    <vdl-document>/flow1/flow1.xhtml</vdl-document>
  </view>
  <view id="final">
    <vdl-document>/flow1/page2.xhtml</vdl-document>
  </view>
  <flow-return id="return">
    <from-outcome>/index.xhtml</from-outcome>
  </flow-return>
</flow-definition>
```

Mit einem Flow-Call-Knoten kann ein Flow aus einem anderen Flow heraus gestartet werden. Dabei ist es sogar möglich, Parameter an den aufzurufenden Flow zu übergeben. Listing**Flow-Definition mit Flow-Call-Knoten und ausgehendem Parameter**zeigt, wie aus Flow`flow1`heraus der Flow`flow2`mit dem Parameter`param1`aufgerufen wird.

```
TODO remove belowskip (layout)
<flow-definition id="flow1">
  <flow-call id="flow2">
    <flow-reference>
      <flow-id>flow2</flow-id>
    </flow-reference>
    <outbound-parameter>
      <name>param1</name>
      <value>#{flow1Bean.value}</value>
    </outbound-parameter>
  </flow-call>
</flow-definition>
```

Der Parameter wird nur korrekt an den aufgerufenen Flow übergeben, wenn dieser einen eingehenden Parameter mit dem selben Namen definiert. Listing**Flow-Definition mit eingehendem Parameter**zeigt die Definition des eingehenden Parameters`param1`im Flow`flow2`.

```
<flow-definition id="flow2">
  <inbound-parameter>
    <name>param1</name>
    <value>#{flow2Bean.value}</value>
  </inbound-parameter>
</flow-definition>
```

JSF ermöglicht das Registrieren von Methoden, die beim Starten oder Beenden des Flows ausgeführt werden. Die entsprechenden Methoden müssen dazu in der Konfiguration als Method-Expression in den Elementen`initializer`oder`finalizer`eingetragen werden (JSF erwartet eine Methode ohne Parameter mit dem Rückgabewert`void`):

```
<initializer>#{bean.initializeFlow}</initializer>
<finalizer>#{bean.finalizeFlow}</finalizer>
```

11.2.3 Definition mit Java

Nachdem sich ausufernde XML-Konfigurationen immer geringerer Beliebtheit erfreuen, sieht JSF 2.2 zusätzlich die Möglichkeit zur Definition von Flows mit Java vor. JSF setzt dabei ganz auf*Contexts and Dependency Injection*(CDI) , das ab Version 6 ebenfalls Teil von Java EE ist, und erwartet die Definition von Flows in sogenannten Producer-Methoden. Informationen zu CDI finden Sie in Kapitel**Sektion: JSF und CDI**- Abschnitt**sec-cdi-prod-meth Producer-Methoden**zeigt Details zu Producer-Methoden im Speziellen. Jeder Flow wird in einer eigenen Producer-Methode als Instanz vom Typ`javax.faces.flow.Flow`erstellt. Die Producer-Methode kann in einer beliebigen serialisierbaren Klasse definiert werden und muss mit`@Produces`und dem von JSF definierten Qualifier annotiert sein. Der Name der Methode kann ebenfalls beliebig werden. Die Methode muss einen Parameter vom Typ`javax.faces.flow.builder.FlowBuilder`haben, der mit dem Qualifier`@FlowBuilderParameter`annotiert ist. Nur so ist gewährleistet, dass CDI beim Aufruf der Methode eine einsatzbereite Instanz für diesen Typ an die Methode übergibt. Listing**Komplette Definition von Flow flow1 mit Java**zeigt die Definition unseres Flows`flow1`mit Java. Der daraus resultierende Flow entspricht dem in Listing**Komplette Definition von Flow flow1 mit XML**mit XML definierten Flow.

```
public class Flow1 implements Serializable {
  @Produces @FlowDefinition
  public Flow buildFlow(
      @FlowBuilderParameter FlowBuilder flowBuilder) {
    flowBuilder.id("", "flow1");
    flowBuilder.viewNode("start", "/flow1/flow1.xhtml")
        .markAsStartNode();
    flowBuilder.viewNode("final", "/flow1/page2.xhtml");
    flowBuilder.returnNode("return").fromOutcome("/index");
    return flowBuilder.getFlow();
  }
}
```

Der Flow wird innerhalb der Producer-Methode mit dem von CDI übergebenen`FlowBuilder`zusammengestellt. Mit einem Aufruf der Methode`id`wird die ID des zu erstellenden Flows gesetzt. Die Methode erwartet als ersten Parameter die ID des definierenden Dokuments und erst als zweiten Parameter die Flow-ID. Auf diese Dokumenten-ID kommen wir nochmals in Abschnitt**[Sektion: Faces-Flows in Jar-Dateien]**zurück - sie bleibt im Moment leer (Achtung:`null`ist nicht erlaubt).
Die Knoten unseres Flows werden ebenfalls mit Methoden der Klasse`FlowBuilder`definiert. Die Methode`viewNode`erstellt einen View-Knoten und erwartet die Knoten-ID als ersten Parameter und die View-ID der anzuzeigenden Seite inklusive des Flow-Verzeichnisses als zweiten Parameter. Analog erzeugt ein Aufruf der Methode`returnNode`einen Return-Knoten mit der übergebenen Knoten-ID. Das Navigationsziel wird dann über einen Aufruf von`fromOutcome`direkt in der erzeugten Knoteninstanz gesetzt. Über die Methode`markAsStartNode`kann ein beliebiger Knoten des Flows als Startknoten festgelegt werden.
Ganz zum Schluss wird der Flow mit einem Aufruf von`getFlow()`erzeugt und zurückgeliefert.
Die Definition eines Flow-Call-Knotens mit Parameter sieht in der Java-Konfiguration folgendermaßen aus:

```
flowBuilder.flowCallNode("flow2").flowReference("", "flow2")
    .outboundParameter("param1", "#{flow1Bean.value}");
```

Ein eingehender Parameter wird mit Java wie folgt definiert:

```
flowBuilder.inboundParameter("param1", "#{flow2Bean.value}");
```

Auch in der Java-Konfiguration können Methoden registriert werden, die JSF beim Starten oder Beenden des Flows aufruft:

```
flowBuilder.initializer("#{bean.initializeFlow}");
flowBuilder.finalizer("#{bean.finalizeFlow}");
```

11.3 Flow-Scope

Bis jetzt haben wir uns noch keine Gedanken über die innerhalb eines Flows benutzten Daten gemacht. In Abschnitt**Sektion: Konversationen mit JSF**haben wir dieses Thema bereits aus Sicht von Geschäftsprozessen behandelt und den Einsatz von Konversationen empfohlen. Managed-Beans in Konversationen funktionieren natürlich auch im Zusammenspiel mit Faces-Flows. JSF 2.2 bietet aber mit dem Flow-Scope eine maßgeschneiderte Lösung für Managed-Beans, die in Flows zum Einsatz kommen.
In JSF hat jeder Flow seinen eigenen Flow-Scope, dessen Lebensdauer beim Starten des zugeordneten Flows beginnt und sich bis zum Beenden des Flows erstreckt. Der Flow-Scope ist dabei immer an das aktuelle Browserfenster beziehungsweise an den Browsertab gebunden. Es gibt somit keine Probleme, wenn die Anwendung in mehreren Fenstern oder Tabs läuft. Die Unterscheidung von Browserfenstern und Browsertabs wird ab Version 2.2 von JSF intern über sogenannte Client-Windows gemachtAchten Sie auf den Parameter`jfwid`in der URL.:.

11.3.1 Managed-Beans

im Flow-Scope

In JSF ist eine Managed-Bean im Flow-Scope immer genau einem Flow zugeordnet. Dadurch hängt ihre Lebensdauer von der Ausführung des Flows ab. Die Bean-Instanz wird beim ersten Zugriff nach dem Starten des Flows erstellt und beim Beenden des Flows wieder aus dem Speicher entfernt.
JSF setzt auch beim Flow-Scope voll auf CDI und stellt die Scope-Annotation`javax.faces.flow.FlowScoped`für CDI-Beans zur Verfügung. Da jeder Flow seinen eigenen Scope hat, muss im Element`value`von`@FlowScoped`die ID des verknüpften Flows angegeben werden. Listing**Managed-Bean im Flow-Scope**zeigt eine CDI-Bean im Flow-Scope des Flows`flow1`.

```
@Named
@FlowScoped(value="flow1")
public class Flow1Bean {
  ...
}
```

11.3.2 Direkter Zugriff auf den Flow-Scope

Für kleinere Datenmengen muss nicht unbedingt eine Managed-Bean erstellt werden. JSF bietet direkten Zugriff auf den Flow-Scope über Java und über das implizite Objekt`flowScope`in Unified-EL-Ausdrücken.
Java-seitig erfolgt der Zugriff auf den aktuellen Flow-Scope über folgenden Code:

```
FacesContext ctx = FacesContext.getCurrentInstance();
Map<Object, Object> flowScope = ctx.getApplication()
    .getFlowHandler().getCurrentFlowScope();
flowScope.put("userName", "Michael Kurz");
```

Der Zugriff auf den Flow-Scope in einer Value-Expression erfolgt über das implizite Objekt`flowScope`:

```
<h:outputText value="#{flowScope.userName}"/>
```

11.4 Faces-Flows in Jar-Dateien

Spätestens dann, wenn ein Faces-Flow in mehreren Applikationen eingesetzt werden soll, muss man sich Gedanken über dessen Wiederverwendbarkeit machen. JSF 2.2 bietet dazu die Möglichkeit, Faces-Flows in Jar-Dateien zu verpacken - und das unabhängig davon, ob der Flow mit XML oder mit Java definiert ist. Die Definition eines Flows in einer Jar-Datei funktioniert ähnlich einfach wie die Definition in einer Applikation, wenn man einige Details beachtet.
Der erste Unterschied ist der Ablageort des Flow-Verzeichnisses. In Jar-Dateien erwartet JSF das Flow-Verzeichnis mit den XHTML-Dateien immer im Verzeichnis`/META-INF/flows`. Hier ist zu beachten, dass die in Abschnitt**[Sektion: Ein erstes Beispiel]**gezeigten Konventionen für Flows in Jar-Dateien nicht gelten. Daher muss jeder einzelne Knoten explizit in der Konfiguration definiert sein.
Das führt uns auch schon zum nächsten Punkt: Die Definition von Flows mit XML erfolgt in Jar-Dateien zentral für alle Flows in der Datei`/META-INF/faces-config.xml`. In dieser`faces-config.xml`kann im Element`name`ein Name für die Konfiguration bestimmt werden. JSF verwendet diesen Namen als zusätzliche ID für den Flow - die sogenannte*Defining-Document-ID*. Damit lassen sich Namenskonflikte mit Flows aus anderen Jar-Dateien vermeiden. Listing**Definition eines Flows mit Defining-Document-ID**zeigt die Definition des Flows mit der ID`flow1`und der Defining-Document-ID`project1`.

```
<faces-config ...>
  <name>project1</name>
  <flow-definition id="flow1">
    ...
  </flow-definition>
</faces-config>
```

Wenn ein Flow mit einer Defining-Document-ID definiert ist, muss diese beim Starten des Flows angegeben werden. Dazu muss auf der startenden Komponente mit dem Tag`f:attribute`das Attribut`to-flow-document-id`mit der Defining-Document-ID als Wert gesetzt werden. Der Flow aus Listing**Definition eines Flows mit Defining-Document-ID**wird wie folgt gestartet:

```
<h:link value="Flow1 starten" outcome="flow1">
   <f:attribute name="to-flow-document-id" value="project1"/>
</h:link>
```

11.5 Beispiel Faces-Flows

Das Beispiel*Faces-Flows*enthält neben dem bereits bekannten Flow`flow1`in einer XML- und einer Java-Variante zusätzlich den neuen Flow`login`zum Einloggen eines Benutzers. Anhand dieses Login-Flows wollen wir Ihnen zeigen, wie ein Flow aus einem anderen Flow heraus mit Parametern aufgerufen wird. Den kompletten Quellcode des Beispiels*Faces-Flow*finden Sie gemeinsam mit den*MyGourmet*-Beispielen unter**http://jsfatwork.irian.at**.
Listing**Definition von Flow login mit XML**zeigt die Definition des Login-Flows mit XML. Dieser Flow liegt direkt in der Applikation im Verzeichnis`login`. Dort gibt es die Seiten`login.xhtml`,`success.xhtml`und`error.xhtml`, für die JSF per Konvention einen View-Knoten erstellt. Der interessanteste Teil der Definition ist aber der Flow-Call-Knoten für den Flow`forgotPassword`mit dem Parameter`username`. Beim Aufruf des zweiten Flows wird der Wert des Parameters aus der Eigenschaft`username`der Bean`loginBean`im Flow-Scope ausgelesen und an den Flow übergeben.

```
<flow-definition id="login">
```

```
  <flow-return id="login-return">
      <from-outcome>/index</from-outcome>
  </flow-return>
  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>forgotPassword-return</from-outcome>
      <to-view-id>/login/login.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>
  <flow-call id="forgotPassword">
    <flow-reference>
      <flow-id>forgotPassword</flow-id>
    </flow-reference>
    <outbound-parameter>
      <name>username</name>
      <value>#{loginBean.username}</value>
    </outbound-parameter>
  </flow-call>
</flow-definition>
```

Der Flow definiert zusätzlich eine Navigationsregel für den Outcome`forgotPassword-return`, der beim Beenden von Flow`forgotPassword`zurückgeliefert wird. In diesem Fall wird wieder die Seite angezeigt. Listing**Definition von Flow login-java mit Java**zeigt die Java-Variante des Login-Flows.

```
public class LoginJavaFlow implements Serializable {
  @Produces @FlowDefinition
  public Flow buildFlow(
      @FlowBuilderParameter FlowBuilder flowBuilder) {
    flowBuilder.id("", "login-java");
    flowBuilder.viewNode("start", "/login-java/login-java.xhtml")
        .markAsStartNode();
    flowBuilder.viewNode("success", "/login-java/success.xhtml");
    flowBuilder.viewNode("error", "/login-java/error.xhtml");
    flowBuilder.flowCallNode("forgotPassword")
        .flowReference("", "forgotPassword")
        .outboundParameter("username",
            "#{loginJavaBean.username}");
    flowBuilder.navigationCase().fromViewId("*")
        .fromOutcome("forgotPassword-return")
        .toViewId("/login-java/login-java.xhtml");
    flowBuilder.returnNode("login-return").fromOutcome("/index");
    return flowBuilder.getFlow();
  }
}
```

Nachdem beide Varianten in der selben Applikation definiert sind, haben wir für die Java-Variante die Flow-ID`login-java`gewählt. Aus Sicht der Funktionalität gibt es keinen Unterschied zwischen den beiden Ausführungen des Login-Flows.
Der Flow`forgotPassword`liegt in einem eigenen Maven-Modul und wird in der Applikation als Jar-Datei eingebunden. Der Flow besteht nur aus der Seite`forgotPassword.xhtml`, die im Verzeichnis`/META-INF/flows/forgotPassword`abgelegt ist. Die Definition des Flows befindet sich in der Konfigurationsdatei`/META-INF/faces-config.xml`.
Listing**Definition von Flow forgotPassword**zeigt die Definition des Flows mit dem eingehenden Parameter. Der Wert des übergebenen Parameters wird beim Starten des Flows im Flow-Scope unter dem Schlüssel`username`abgelegt.

```
<flow-definition id="forgotPassword">
  <flow-return id="return">
    <from-outcome>forgotPassword-return</from-outcome>
  </flow-return>
  <inbound-parameter>
    <name>username</name>
    <value>#{flowScope.username}</value>
  </inbound-parameter>
</flow-definition>
```

12 MyGourmet Fullstack - - JSF, CDI und JPA mit CODI kombiniert

In diesem Kapitel werden wir anhand unseres*MyGourmet*-Beispiels die Architektur einer JSF-Anwendung mit CDI und*Apache MyFaces CODI*vorstellen, die sich in dieser oder ähnlicher Form in der Praxis bewährt hat. Sie stellt eine optimale Ausgangsbasis für Ihre eigenen JSF-Webapplikationen dar - auch für komplexere Anwendungen. Das Beispiel*MyGourmet Fullstack*ist eine Erweiterung des Beispiels*MyGourmet 17*, in dem zum einen die Funktionalität der Anwendung ausgebaut wird und zum anderen die Architektur der Anwendung auf den derzeitigen Stand der Technologie gebracht wird.
Als JSF-Implementierung kommt standardmäßig*Apache MyFaces*zum Einsatz. Das Beispiel funktioniert aber auch mit der aktuellen Version von*Mojarra*. Wie in allen anderen Beispielen können Sie auch hier die JSF-Implementierung über ein Profil in der Datei`pom.xml`bestimmen (siehe Anhang**Kapitel: Eine kurze Einführung in Maven**für Details). Zur Persistierung von Daten wird die*Java Persistence API*(JPA) in Version 2.0 mit dem*Hibernate EntityManager*als Implementierung eingesetzt. Datenbankseitig steht*HyperSQL Data-Base*(HSQLDB) bereit - es kann aber jede Datenbank, die von*Hibernate*unterstützt wird, verwendet werden.
Den Quellcode zu*MyGourmet Fullstack*finden Sie wie den Code aller bisherigen Beispiele unter**http://jsfatwork.irian.at**.

12.1 Architektur von MyGourmet Fullstack

In*MyGourmet Fullstack*kommt eine bei der Java-Webentwicklung weitverbreitete Architektur mit drei übereinanderliegenden Schichten zum Einsatz. Die*Präsentationsschicht*ist die oberste Schicht und beinhaltet die Benutzerschnittstelle. Sie greift auf die*Serviceschicht*zu, in der die Geschäftslogik der Anwendung definiert ist. Ganz unten liegt die*Datenzugriffsschicht*, in der sämtliche in der Serviceschicht getätigten Zugriffe auf die Daten der Anwendung gekapselt sind. Die Entitäten des hinter der Anwendung liegenden Modells sind schichtübergreifend verfügbar. Abbildung**Architektur von MyGourmet Fullstack**zeigt eine grafische Darstellung der Architektur von*MyGourmet Fullstack*.

Abbildung:Architektur von MyGourmet Fullstack

Ein Vorteil der Schichtenarchitektur ist die strikte Trennung der Zuständigkeiten (Separation of Concerns, SOC). Jede Schicht ist dabei für einen bestimmten Bereich der Anwendung zuständig. Eine Schicht kann auf die Funktionalität der direkt unter ihr liegenden Schicht zugreifen und kann Funktionalität für die direkt über ihr liegende Schicht anbieten. Nachdem es nie Abhängigkeiten nach oben geben darf, bildet jede Schicht mit den unter ihr liegenden Schichten ein abgeschlossenes Subsystem.
Wir empfehlen Ihnen, immer eine saubere Schichtentrennung einzuhalten - auch wenn das im ersten Moment wie ein unnötiger Mehraufwand aussieht. Die Trennung (natürlich nur an vernünftigen Stellen) zahlt sich spätestens in der Wartung aus.
In den nächsten Abschnitten werden wir einen kurzen Blick auf alle Schichten von*MyGourmet Fullstack*inklusive der schichtübergreifenden Entitäten werfen. Die Struktur des Maven-Projekts ist etwas einfacher gehalten und entspricht nicht direkt den Schichten der Anwendung. Aus Gründen der Einfachheit haben wir die Entitäten, die Datenzugriffsschicht und die Serviceschicht im Modul`mygourmet-service`zusammengefasst. Die Präsentationsschicht befindet sich im Modul`mygourmet-webapp`. Für umfangreichere Projekte macht es aber durchaus Sinn, jede Schicht in einem eigenen Maven-Modul unterzubringen.

12.1.1 Entitäten

Entitäten bilden das Modell der Geschäftsobjekte und sind ein zentraler Bestandteil jeder Applikation. Entitäten sind einfache JavaBeans mit Eigenschaften und deren*Getter*-und*Setter*-Methoden. Sie stellen die objektorientierte Repräsentation der Tabellen in der Datenbank dar. Da sie in allen Schichten verwendet werden, sind sie aus Sicht der Architektur außerhalb der Schichten angesiedelt. Die Entitätsklassen finden Sie im Package`at.irian.jsfatwork.domain`im Modul`mygourmet-service`.
Die Information über die Zuordnung zwischen den Tabellen und Spalten der Datenbank und den Entitäten und Eigenschaften ist direkt in den Klassen enthalten. JPA bietet dazu eine Reihe von Annotationen an. Dadurch ist es nicht mehr nötig, Unmengen von XML-Dateien zur Konfiguration dieser Zuordnung anzulegen.
Die benutzerdefinierten Bean-Validation-Validatoren müssen sich ebenfalls im Modul`mygourmet-service`befinden, da sie in den Entitäten benutzt werden. Nachdem Bean-Validation aber nicht von JSF abhängig ist, stellt das kein Problem dar. Die Validatorklassen und Annotationen finden Sie in`at.irian.jsfatwork.validation`.

12.1.2 Datenzugriffsschicht

In der Datenzugriffsschicht wird die Schnittstelle zur Datenbank mit einem generischen CRUD-Service realisiert. Dieser Service ist in der Klasse`CrudService`als CDI-Bean im Application-Scope umgesetzt und nutzt die JPA-Unterstützung von CODI. Der CRUD-Service stellt generische Methoden für die wichtigsten Funktionen des Entity-Managers von JPA bereit. Für*MyGourmet*reicht diese simple Umsetzung aus - in komplexeren Applikationen kann ein solcher CRUD-Service dann zum Beispiel als Grundlage für Repositories dienen. Listing**MyGourmet Fullstack: CrudService**zeigt Teile der Klasse`CrudService`.

```
@ApplicationScoped
public class CrudService {
```

```
  @Inject
  private EntityManager em;

  public <T extends BaseEntity> void persist(T entity) {
    em.persist(entity);
  }

  public <T extends BaseEntity> T findById(
      Class<T> clazz, long id) {
    return em.find(clazz, id);
  }

  public <T extends BaseEntity> void delete(T entity) {
    em.remove(entity);
  }
}
```

Die zentrale Stelle zur Interaktion mit dem Persistenzkontext von JPA ist die Klasse`EntityManager`. In*MyGourmet*wird der Entity-Manager wie üblich in der`persistence.xml`konfiguriert und von CDI verwaltet. Der CRUD-Service bekommt den Entity-Manager dann direkt als CDI-Bean in ein mit`@Inject`annotiertes Feld eingeimpft.
Der Entity-Manager wird allerdings nicht automatisch als CDI-Bean zur Verfügung gestellt - darum müssen wir uns selbst kümmern. Mit einer Producer-Methode ist das aber mit wenigen Zeilen Code erledigt. Listing**MyGourmet Fullstack: Producer für Entity-Manager**zeigt die Klasse`EntityManagerProducer`mit der Producer-Methode`createEntityManager()`, die eine Bean vom Typ`EntityManager`im Request-Scope zur Verfügung stellt. Damit ist gewährleistet, dass alle Beans, die eine Abhängigkeit auf den Entity-Manager definiert haben, pro Request immer dieselbe Instanz eingeimpft bekommen.
Die von der Producer-Methode zur Verfügung gestellte Entity-Manager-Instanz kommt aus der Bean`EntityManagerProducer`. CDI wertet dazu beim Erzeugen der Bean die Annotation`@PersistenceContext`aus und setzt den Entity-Manager in die annotierte Eigenschaft. Die Bean ist im Dependent-Scope definiert, damit die Producer-Methode wirklich für jeden Request einen neuen Entity-Manager zur Verfügung stellt.

```
@Dependent
public class EntityManagerProducer {

  @PersistenceContext(unitName = "mygourmet")
  private EntityManager entityManager;

  @Produces
  @RequestScoped
  public EntityManager createEntityManager() {
    return this.entityManager;
  }

  public void dispose(@Disposes EntityManager em) {
    if (em.isOpen()) {
      em.close();
    }
  }
}
```

Am Ende des Requests wird der Entity-Manager zuerst in der Disposer-Methode`dispose()`ordnungsgemäß geschlossen und dann aus dem Speicher entfernt. CDI unterstützt Disposer-Methoden für alle von Producer-Methoden erzeugten Beans. Die Methode muss genau einen

mit`@Disposes`annotierten Parameter vom Typ der zu beseitigenden Bean haben.
Die Konfiguration des JPA-Persistenzkontexts erfolgt in der Datei`/META-INF/persistence.xml`im Maven-Modul`mygourmet-service`. Dort ist neben der JDBC-Verbindung zu HSQLDB und einigen Einstellungen für*Hibernate*auch ein Connection-Pool, in unserem Fall*C3P0*, eingetragen.*Hibernate*ist so konfiguriert, dass bei jedem Start der Applikation die Datenbank aus den Mappings in den Entitäten neu erstellt wird. Details finden Sie im Sourcecode des Beispiels.

12.1.3 Serviceschicht

Die Geschäftslogik der Anwendung ist in der Serviceschicht untergebracht und von der Benutzerschnittstelle entkoppelt. Die Serviceschicht bekommt den CRUD-Service injiziert und ruft ihn bei Bedarf, wie zum Beispiel bei einer Persistierung, auf. Der Weg zur Datenbank sollte in der Präsentationsschicht ausschließlich über die Serviceschicht und nicht direkt über die Datenzugriffsschicht erfolgen. Die Serviceklassen befinden sich im Package`at.irian.jsfatwork.service`und sind als CDI-Beans im Application-Scope definiert. Listing**MyGourmet Fullstack: Serviceimplementierung**zeigt exemplarisch einen Ausschnitt der Klasse`ProviderService`.

```
@ApplicationScoped
public class ProviderService {

  @Inject
  private CrudService crudService;

  @Transactional
  public void save(Provider entity) {
    if (entity.isTransient()) {
      crudService.persist(entity);
    } else {
      crudService.merge(entity);
    }
  }
  @Transactional
  public void delete(Provider provider) {
    provider = crudService.merge(provider);
    for (Order order : provider.getOrders()) {
      order.setCustomer(null);
      crudService.delete(order);
    }
    provider.getCategories().clear();
    crudService.delete(provider);
  }
  @Transactional
  public Provider findById(long id) {
    return crudService.findById(Provider.class, id);
  }
  @Transactional
  public List<Provider> findAll() {
    return crudService.findAll(Provider.class);
  }
}
```

Die Serviceschicht ist in*MyGourmet Fullstack*auch für die Transaktionskontrolle der Applikation zuständig. Die Transaktionen sind an einzelnen Servicemethoden ausgerichtet und werden von CODI mit einem Interceptor verwaltet. Jede Methode einer Serviceklasse, die mit der CODI-Annotation`@Transactional`annotiert ist, wird in einer Transaktion ausgeführt. Wenn die Serviceklasse

selbst mit`@Transactional`annotiert ist, werden alle Methoden der Klasse in einer Transaktion ausgeführt. Die Serviceschicht ist der ideale Ort zur Definition der Transaktionen, da sie für die Benutzerschnittstelle das Tor zu Geschäftslogik darstellt. Wenn Sie nochmals einen Blick auf Listing**MyGourmet Fullstack: CrudService**werfen, werden Sie bemerken, dass es dort kein`@Transactional`gibt. Nachdem CRUD-Operationen nur im Service verwendet werden, laufen sie automatisch in den dort definierten Transaktionen ab.
Die Schichtentrennung ist beispielsweise auch für das Testen außerordentlich wichtig. Die Applikation kann dann nämlich (ohne Ausführung der GUI selbst) über die Serviceschicht direkt getestet werden. Wir empfehlen Ihnen, die Serviceschicht von Beginn an zu testen und die Tests immer auf gleichem Stand wie die GUI-Logik zu halten. Gerade bei der Entwicklung von Webanwendungen kostet jeder zu einem Neustart des Servers führende Fehler, der mit der Ausführung eines Tests verhindert hätte werden können, wertvolle Entwicklungszeit.

12.1.4 Präsentationsschicht

Die*Präsentationsschicht*umfasst die Benutzerschnittstelle der Applikation und bildet die oberste Schicht der Architektur. In*MyGourmet Fullstack*besteht die Präsentationsschicht aus der JSF-Webanwendung und ist im*Maven*-Modul`mygourmet-webapp`untergebracht. Dieses Modul enthält alle Seitendeklarationen inkusive ihrer Page-Beans sowie alle Komponenten, Konverter, Validatoren und Phase-Listener.
Die Präsentationsschicht ist nur für die Benutzerschnittstelle zuständig und greift zur Ausführung von Geschäftslogik auf die Serviceschicht zu. Umgekehrt darf es aber keine Abhängigkeiten der Serviceschicht oder gar der Datenzugriffsschicht auf die Präsentationsschicht geben. GUI/JSF-Spezifika haben dort natürlich nichts verloren.
Sehen wir uns diesen Vorgang anhand der Anbieter-Übersichtsseite und ihrer Page-Bean etwas genauer an. Listing**MyGourmet Fullstack: Page-Bean der Anbieter-Übersichtsseite**zeigt die Klasse`ProviderListBean`der Page-Bean. Beim initialen Zugriff auf die Ansicht wird die Bean inklusive der Konversation erstellt. Beim Laden der Anbieterliste in der Methode`preRenderView`kommt beim Aufruf von`providerService.findAll()`im dahinterliegenden CRUD-Service der von der Producer-Methode erzeugte Entity-Manager für den aktuellen Request zum Einsatz. Da die Servicemethode mit`@Transactional`annotiert ist, startet CODI vor dem eigentlichen Aufruf der Methode eine Transaktion. Beim Verlassen der Methode kümmert sich CODI weiterhin um das korrekte Beenden der Transaktion mit einem Commit im Normalfall und einem Rollback, falls die Methode eine Exception wirft.

```
@Named
@ViewAccessScoped
public class ProviderListBean implements Serializable {
  @Inject
  private ProviderService providerService;
  private List<Provider> providerList;

  @PreRenderView
  public void preRenderView() {
    providerList = providerService.findAll();
  }
  public List<Provider> getProviderList() {
    return providerList;
  }
  public void deleteProvider(Provider provider) {
    providerService.delete(provider);
  }
}
```

Dasselbe gilt für das Löschen eines Anbieters in`deleteProvider`. Beim Aufruf von`providerService.delete(provider)`wird im dahinterliegenden CRUD-Service ebenfalls die Entity-Manager-Bean aus dem aktuellen Request verwendet. Der zu löschende Anbieter wird beim Aufruf

der Action-Methode direkt über die Method-Expression als eine der zuvor geladenen Entitäten übergeben. Im`ProviderService`muss die Entität daher zuerst mit einem Aufruf der Methode`merge`an den aktuellen Entity-Manager gebunden werden. Nachdem die Methode`delete`im Service mit`@Transactional`annotiert ist, läuft die Operation in einer Transaktion ab.
Damit ist*MyGourmet Fullstack*fertig konfiguriert und einsatzbereit. Wir möchten Sie dazu einladen, den Quellcode der Anwendung genau unter die Lupe zu nehmen. Betrachten Sie das Beispiel als Basis für eigene Experimente und erkunden Sie die Details der Zusammenarbeit von JSF, JPA, CDI und CODI in der Praxis.

13 JSF und Spring

Der Einsatz von*Spring*in JSF-Projekten bietet eine ganze Reihe von Vorteilen gegenüber dem Einsatz der internen*Managed Bean Creation Facility*.*Spring*bringt unter anderem einen viel mächtigeren Dependency-Injection-Mechanismus, Autowiring von Beans und eine sehr ausgefeilte Unterstützung von Aspect-Oriented Programming (AOP) mit, um nur einige der Vorteile zu nennen - und das alles, ohne die Komplexität der Anwendung merklich zu erhöhen. Aus Sicht von JSF macht es keinen Unterschied, ob Beans über*Spring*oder JSF-intern verwaltet werden. Die Verbindung zwischen der Ansicht und dem Modell erfolgt in beiden Fällen über die*Unified-EL*.
Nach einigen Informationen zur Konfiguration von*Spring*in Abschnitt**[Sektion: Konfiguration von Spring]**zeigt Abschnitt**[Sektion: Dependency-Injection mit Spring und JSR-330]**wie JSF-Applikationen mit*Spring*entwickelt werden können. Im Anschluss daran zeigt Abschnitt**[Sektion: *MyGourmet 16 Spring*: Integration von Spring]**Details zum Beispiel*MyGourmet 16 Spring*.
Ein weiteres wichtiges Thema bei der Webentwicklung ist die Abbildung von Geschäftsprozessen. Dazu werden immer häufiger Konversationen eingesetzt. Warum das so ist zeigt Abschnitt**[Sektion: Konversationen mit JSF]**. Wenn Sie in Ihrem Projekt bereits Spring zur Verwaltung der Managed-Beans verwenden, steht einem Einsatz von Konversationen nichts mehr im Weg. Abschnitt**[Sektion: Apache MyFaces Orchestra]**zeigt anschließend, wie Sie mit dem Projekt*Apache MyFaces Orchestra*Konversationen mit Spring verwenden.

13.1 Konfiguration von Spring

Das Einbinden von*Spring*in eine JSF-Anwendung erfolgt in zwei Schritten. Zuerst muss*Spring*selbst konfiguriert und aktiviert werden. Am einfachsten funktioniert das mit dem in*Spring*integrierten Servlet-Context-Listener`ContextLoaderListener`, der in die`web.xml`eingebunden wird und beim Initialisieren eines Servlets den`ApplicationContext`von*Spring*einrichtet. Die dazu benötigten Konfigurationsdateien holt sich der Listener aus dem Kontextparameter`contextConfigLocation`. Dort kann eine Liste von Dateinamen, getrennt durch Leerzeichen, Komma oder Semikolon, angegeben werden. Die Pfade werden relativ zum Root der Webapplikation ausgewertet und es besteht die Möglichkeit, Suchmuster wie`/WEB-INF/*Context.xml`oder`/WEB-INF/**/*Context.xml`einzusetzen. Listing**Konfiguration von Spring in der web.xml**zeigt die notwendige Konfiguration in der`web.xml`.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/spring-config.xml
  </param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<listener>
  <listener-class>
  org.springframework.web.context.request.RequestContextListener
  </listener-class>
```

```
</listener>
```

*Spring*unterstützt ab Version 2.0 die für Webanwendungen relevanten Scopes`request`und`session`. Damit diese richtig funktionieren, muss in der`web.xml`zusätzlich der Servlet-Request-Listener`RequestContextListener`konfiguriert werden. Zusätzlich erlaubt*Spring*ab Version 2.0 die Definition eigener Scopes. Wir werden von dieser Möglichkeit auch Gebrauch machen und den in JSF 2.0 hinzugekommenen View-Scope implementieren (siehe Abschnitt**[Sektion: *MyGourmet 16 Spring*: Integration von Spring]**).*Spring*selbst unterstützt diesen neuen Scope in der momentan aktuellen Version noch nicht.
Im zweiten Schritt müssen wir JSF anweisen, Managed-Beans nicht mehr selbst aufzulösen, sondern die Arbeit an*Spring*zu delegieren.*Spring*bietet zu diesem Zweck einen eigenen EL-Resolver, der zuerst Managed-Beans über den`ApplicationContext`von*Spring*auflöst, bevor er - wenn die Auflösung erfolglos war - die Kontrolle an den Standard-Resolver von JSF weitergibt. Dieser EL-Resolver wird im`application`-Bereich der`faces-config.xml`, wie in Listing**Konfiguration des Spring-EL-Resolvers in der faces-config.xml**gezeigt, konfiguriert.

```
<faces-config>
  <application>
    <el-resolver>
      org.springframework.web.jsf.el.SpringBeanFacesELResolver
    </el-resolver>
    ...
  </application>
</faces-config>
```

Alle hier gemachten Angaben beziehen sich auf*Spring*in Version 2.5 oder höher in Kombination mit JSF ab Version 1.2. Für ältere Versionen von*Spring*(deren Einsatz wir nicht mehr empfehlen) oder JSF in Version 1.1 weicht die Konfiguration an einigen Stellen von der hier gezeigten ab.

13.2 Dependency-Injection mit Spring und JSR-330

*Spring*unterstützt ab Version 3.0 den Standard*Dependency Injection for Java*(JSR-330, auch*At Inject*genannt). Mit JSR-330 ist es erstmals gelungen, die wichtigsten Annotationen für die Dependency-Injection zu standardisieren. Dadurch ist es endlich möglich, in unterschiedlichen Umgebungen wie*Java EE 6*,*Spring 3*oder*Guice 2*die selben Annotationen aus dem Package`javax.inject`einzusetzen. Mit diesem Ansatz lassen sich Abhängigkeiten zu Annotationen eines spezifischen Dependency-Injection-Frameworks vermeiden.
JSR-330 deckt die Kernbereich der Dependency-Injection ausreichend ab. Andere Bereiche wie die Definition von Beans werden in JSR-330 allerdings nur rudimentär berücksichtigt - dafür ist der Einsatz umso einfacher.*Spring*unterstützt zum Registrieren von Beans unter anderem die JSR-330-Annotation`@Named`. Der Bean-Name wird dabei entweder direkt im Element`value`angegeben oder ansonsten vom Klassennamen abgeleitet.
Für den Gültigkeitsbereich einer Bean steht in JSR-330 nur die Annotation`@Singleton`zur Verfügung. Alle anderen Gültigkeitsbereiche können im`value`-Element der Spring-Annotation`@Scope`angegeben werden.
*Spring*unterstützt ab Version 2.0 standardmäßig folgende Gültigkeitsbereiche, die bei der Konfiguration über Annotationen und über XML gleichermaßen gelten:

- *Prototype-Scope*(`prototype`):
  Die Bean wird nicht gespeichert und bei jedem Aufruf neu erstellt. Entspricht dem*None-Scope*in JSF.
- *Request-Scope*(`request`):
  Die Bean lebt für die Zeitdauer einer HTTP-Anfrage. Entspricht dem*Request-Scope*in JSF.
- *Session-Scope*(`session`):
  Die Bean lebt für die Dauer einer Sitzung, in der der Benutzer mit der Anwendung verbunden ist. Entspricht dem*Session-Scope*in JSF.
- *Singleton-Scope*(`singleton`):
  Für die gesamte Lebensdauer der Anwendung ist nur eine für alle Benutzer gleiche Instanz dieser Bean vorhanden. Entspricht dem*Application-Scope*in JSF.

Die Annotation eines Feldes mit`@Inject`reicht aus um einen Injektionspunkt zu definieren. Listing**Dependency-Injection mit @Inject**zeigt die Definition einer Bean mit einem Injektionspunkt.

```
@Named
@Singleton
public class MyBean {
  @Inject
  private Service service;
  ...
}
```

Spring 3.0(oder jedes andere Dependency-Injection-Framework mit Unterstützung für JSR-330) kann diese Abhängigkeit auflösen und eine Bean vom Typ`Service`injizieren. Für den Fall, dass es mehrere Beans mit gleichem Typ gibt, muss die Auswahl weiter eingeschränkt werden. Dazu sieht JSR-330 sogenannte Qualifier vor. Ein Qualifier ist eine beliebige Annotation, die mit`@javax.inject.Qualifier`annotiert ist. Listing**JSR-330 Qualifier**zeigt als Beispiel den Qualifier`@Special`.

```
@Retention(RUNTIME)
@Target({TYPE, FIELD, METHOD})
@Qualifier
public @interface Special {
}
```

Mit einem solchen Qualifier wird dann einerseits die Klasse der Bean und andererseits der Injektionspunkt annotiert. Aber sehen wir uns das am besten anhand eines Beispiels an. Listing**Service-Beans mit und ohne Qualifier**zeigt zwei unterschiedliche Implementierungen des Interfaces`Service`. Beide Klassen werden mit den Spring-Annotationen`@Service`und`@Scope`als Beans im Singleton-Scope definiert. Da beide Beans den Typ`Service`haben (wir gehen davon aus, dass nur das Interface nach außen hin bekannt ist), annotieren wir zur weiteren Unterscheidung die Klasse`SpecialServiceImpl`mit unserem Qualifier`@Special`.

```
@Service("service")
@Scope("singleton")
public class ServiceImpl implements Service {
  ...
}

@Service("specialService")
@Scope("singleton")
@Special
public class SpecialServiceImpl implements Service {
  ...
}
```

Dieselbe Qualifier-Annotation wird auch beim Injektionspunkt benutzt um die Auswahl der Beans vom Typ`Service`auf das gewünschte Exemplar einzuschränken. Listing**Dependency-Injection mit @Injectund Qualifier**zeigt das entsprechende Codefragment.

```
public class MyBean {
  @Inject @Special
  private Service service;
  ...
}
```

JSR-330 stellt mit`@Named`eine konkrete Qualifier-Annotationen zum Einschränken der Auswahl basierend auf dem Namen der Bean zur Verfügung. In Listing**Dependency-Injection mit @Injectund @Named**wird zum Beispiel die Bean vom Typ`Service`mit dem Namen`specialService`aus Listing**Service-Beans mit und ohne Qualifier**injiziert.

```
public class MyBean {
  @Inject @Named("specialService")
  private Service service;
  ...
}
```

*Dependency Injection for Java*ist ein wichtiger Schritt in die richtige Richtung. Im nächsten Abschnitt sehen wir uns an, wie der Einsatz von*Spring*in Kombination mit JSR-330 in*MyGourmet*aussieht.

13.3 MyGourmet 16 Spring: Integration von Spring

In*MyGourmet 16 Spring*dreht sich alles um die Integration von*Spring*. Die dazu notwendigen Bibliotheken werden über Abhängigkeiten in der*pom.xml*in das Maven-Projekt eingebunden. Details entnehmen Sie bitte direkt dem Sourcecode.
Die zusätzlich notwendige Konfiguration in der`web.xml`entspricht genau der im letzten Abschnitt vorgestellten und wird hier nicht mehr wiederholt. Jetzt fehlt für ein funktionstüchtiges System nur mehr die im Kontextparameter`contextConfigLocation`referenzierte XML-Datei`spring-config.xml`und die Konfiguration der Beans. Listing**Spring-Konfiguration von MyGourmet 16**zeigt die komplette*Spring*-Konfiguration für*MyGourmet 16 Spring*(einige Klassennamen und Schema-Locations enthalten aus Platzgründen einen Zeilenumbruch, was aber nicht erlaubt ist).

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/
          spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/
```

```
          spring-context-3.0.xsd">

  <!-- Classpath nach Spring-Komponenten scannen -->
  <context:component-scan base-package="at.irian.jsfatwork"/>

  <!-- View-Scope registrieren -->
  <bean class="org.springframework.beans.factory
      .config.CustomScopeConfigurer">
    <property name="scopes">
      <map>
        <entry key="view">
          <bean class="at.irian.jsfatwork.spring.ViewScope"/>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

Beans werden in*Spring*entweder über Annotationen oder in der XML-Konfiguration definiert. Da wir bisher in allen Beispielen die Managed-Beans über Annotationen konfiguriert haben, sehen wir uns zuerst diese Variante an. Die Umstellung in*MyGourmet*beschränkt sich auf das Austauschen der JSF-Annotationen durch ihre JSR-330- und*Spring*-Pendants in den Klassen`ProviderServiceImpl`,`AddressBean`,`CustomerBean`,`ProviderBean`und`ProviderListBean`. Die Implementierung des View-Scopes für*Spring*erfolgt in der Klasse`at.irian.jsfatwork.spring.ViewScope`. Wir werden die Klasse hier nicht abbilden, bei Interesse werfen Sie bitte einen Blick in den Sourcecode von*MyGourmet 16 Spring*. Das Registrieren des Scopes erfolgt mit einer Bean vom Typ`CustomScopeConfigurer`, deren Eigenschaft`scopes`eine Map mit den benutzerdefinierten Scopes aufnimmt. In Listing**Spring-Konfiguration von MyGourmet 16**sehen Sie die Registrierung des neuen View-Scopes.
Listing**Bean-Definition und Dependency-Injection mit Spring-Annotationen**zeigt die Konfiguration der Beans mit den Namen`providerService`und`providerBean`über die Annotation`@Named`. Das Beispiel beschreibt auch, wie Abhängigkeiten zwischen Beans mit Annotationen konfiguriert werden. Die JSR-330-Annotation`@Inject`auf dem Feld`providerService`der Klasse`ProviderBean`bewirkt, dass beim Erstellen einer Bean mit dem Bezeichner`providerBean`die Bean mit dem Bezeichner`providerService`in das gleichnamige Feld injiziert wird. Mit`@Inject`kann entweder direkt ein Feld der Klasse oder die Setter-Methode einer Eigenschaft annotiert werden.

```
@Named("providerService")
@Singleton
public class ProviderServiceImpl
    implements ProviderService {
...
}

@Named("providerBean")
@Scope("view")
public class ProviderBean {

    @Inject
    private ProviderService providerService;
    ...
}
```

Die Definition von Beans über Annotationen funktioniert allerdings nur, wenn*Spring*in der XML-Konfiguration mit dem Element`context:component-scan`angewiesen wird, Klassen nach Annotationen zu durchsuchen. Das Attribut`base-package`definiert dabei einen Startpunkt, um den Suchvorgang so weit wie möglich einzuschränken und den Start der Anwendung nicht unnötig zu verlängern. Wir beschränken uns in*MyGourmet*auf Klassen, deren vollqualifizierter Name mit`at.irian.jsfatwork`beginnt (siehe

Listing**Spring-Konfiguration von MyGourmet 16**). Als angenehmer Nebeneffekt aktiviert`context:component-scan`auch die Verarbeitung von Standardannotationen wie`@Resource`,`@PostConstruct`und`@PreDestroy`und ab*Spring 3.0*auch von JSR-330-Annotationen.
Zum Abschluss werfen wir noch einen kurzen Blick auf die Konfiguration von Beans über XML. Listing**XML-Konfiguration der Beans in MyGourmet 16 Spring**zeigt, wie eine zu den Annotationen äquivalente Konfiguration in der`spring-config.xml`aussehen würde. Eine Bean wird in*Spring*mit dem Element`bean`deklariert. Der Name wird dabei im Attribut`id`angegeben, der Gültigkeitsbereich im Attribut`scope`und die zugrundeliegende Klasse im Attribut`class`. Das Einbringen der Bean`providerService`in die Bean`providerBean`erfolgt mit dem Kindelement`property`. Das Attribut`name`gibt dazu den Namen der zu setzenden Eigenschaft an und das Attribut`ref`den Bezeichner der zu setzenden Bean.

```
<bean id="providerService"
    class="at.irian.jsfatwork.service.ProviderServiceImpl"/>
<bean id="addressBean" scope="session"
    class="at.irian.jsfatwork.gui.page.AddressBean"/
<bean id="customerBean" scope="session"
    class="at.irian.jsfatwork.gui.page.CustomerBean"/>
<bean id="providerBean" scope="view"
    class="at.irian.jsfatwork.gui.page.ProviderBean">
  <property name="providerService" ref="providerService"/>
</bean>
```

Was wir Ihnen hier gezeigt haben, ist die einfachste Form der Bean-Definition mit*Spring*und JSR-330. Sobald*Spring*läuft und in JSF eingebunden ist, stehen Ihnen alle Möglichkeiten des Spring-Frameworks offen - und glauben Sie uns, das sind eine Menge. Besonders hervorzuheben ist hier die sehr gute und umfangreiche Integration von*Aspect Oriented Programming*(AOP) zur einfachen Implementierung von Querschnittsfunktionalität. Weiterführende Informationen zu*Spring*und*Spring AOP*finden Sie in der sehr gelungenen Dokumentation unter**http://www.springsource.org/documentation**.
Ein wichtiges Einsatzgebiet für die Verwendung von*Spring*mit JSF ist die Möglichkeit, zusätzliche Scopes zu definieren. Insbesondere Konversationen sind hier interessant. In Abschnitt**[Sektion: Konversationen mit JSF]**finden Sie allgemeine Informationen zu diesem Thema und Abschnitt**[Sektion: Apache MyFaces Orchestra]**zeigt wie Sie Konversationen mit*Spring*und*Apache MyFaces Orchestra*einsetzen.

13.4 Konversationen mit JSF

In vielen Webanwendungen lassen sich die zugrunde liegenden Geschäftsprozesse nicht direkt auf den Seitenfluss abbilden. Viele Prozesse, die aus Sicht des Benutzers eine abgeschlossene Einheit bilden, erstrecken sich in der Anwendung über mehrere Anfragen oder sogar über mehrere Ansichten hinweg. Denken Sie zum Beispiel an die Registrierung eines Benutzers, bei der im ersten Schritt die Login-Daten und im zweiten Schritt Daten zur Person abgefragt werden. Für den Benutzer der Anwendung ist dieser Vorgang eine in sich abgeschlossene Tätigkeit, die mit dem Anzeigen der ersten Ansicht beginnt und durch das Betätigen der Fertigstellen-Schaltfläche im zweiten Schritt abgeschlossen wird. Aus Sicht der Webanwendung handelt es sich dabei allerdings nur um eine Reihe von Anfragen auf zwei verschiedene Seiten - das Konzept eines Geschäftsprozesses ist im JSF-Standard voraussichtlich erst ab Version 2.2 enthalten.
Dadurch stellt sich die Frage, in welchem Gültigkeitsbereich die Daten während des Prozesses abgelegt werden müssen, damit sie in jedem Schritt zur Verfügung stehen. Managed-Beans im Request-Scope werden nach jeder Anfrage neu erzeugt und eignen sich daher nicht. Der View-Scope ist auch nur dann ausreichend, wenn der Prozess nicht mehr als eine Ansicht umfasst. Managed-Beans im Application-Scope eignen sich nicht für unsere Zwecke, da sie nur einmal pro Anwendung erzeugt werden und dadurch alle Benutzer die selben Daten sehen. Bleibt als letzte Alternative nur noch der Session-Scope übrig. Der Session-Scope löst zwar das Verfügbarkeitsproblem während des Prozesses, bringt aber einige entscheidende Nachteile mit sich.

An dieser Stelle kommen Konversationen ins Spiel. Konversationen sind der ideale Speicherort für Managed-Beans, deren Lebensdauer über eine Anfrage oder Ansicht hinausgeht. Bei Webanwendungen tritt dieser Fall öfters ein, da sich Geschäftsprozesse nicht immer direkt auf den Seitenfluss der Applikation abbilden lassen. Konversationen bietet einige entscheidende Vorteile gegenüber der Session:

- Eine Konversation kann im Gegensatz zur Session sehr einfach beendet und aus dem Speicher entfernt werden, ohne andere Konversationen oder Managed-Beans außerhalb der Konversation zu beeinflussen.
- Es kann beliebig viele Konversationen pro Benutzer geben.
- Eine Konversation wird normalerweise für jedes Fenster oder Tab des Browsers neu angelegt. Dadurch kann die Anwendung gleichzeitig in mehreren Fenstern oder Tabs laufen, die sich gegenseitig nicht beeinflussen. Mit der Session ist das nicht so einfach möglich, da bei den meisten Browsern für alle Tabs und oft sogar für Fenster die gleiche Session zum Einsatz kommtDer Browser bestimmt natürlich nicht selbst, welche Session zum Einsatz kommt. Wenn aber im Browser für alle Tabs und Fenster dieselben Cookies verwendet werden, kann der Server diese nicht unterscheiden.:.

JSF kann sehr einfach um Konversationen erweitert werden. Wenn Sie Spring einsetzen, stellt das Projekt*Apache MyFaces Orchestra*Konversationen und eine Reihe anderer Erweiterungen für die Entwicklung von Webapplikationen mit JSF und Spring zur Verfügung. Details zu*Orchestra*finden Sie in Abschnitt**[Sektion: Apache MyFaces Orchestra]**.

13.5 Apache MyFaces Orchestra

*Apache MyFaces Orchestra*bietet eine ganze Reihe von Erweiterungen um die Integration von JSF und*Spring*so einfach wie möglich zu gestalten. Das wichtigste Feature ist dabei aber sicherlich die Unterstützung von Konversationen. Abschnitt**[Sektion: Konversationen mit*Orchestra*]**widmet sich daher komplett dem Thema Konversationen mit*Orchestra*und*Spring*.
Ähnlich wie bei Konversationen ist beim Einsatz der*Java Persistence API (JPA)*die Lebensdauer des Persistenzkontexts ein wichtiges Kriterium. Wie bei den Managed-Beans kann es auch hier zu Problemen kommen, wenn die Lebensdauer zu kurz oder zu lang gewählt wird.*Orchestra*bietet auch für dieses Problem eine Lösung, indem es den Persistenzkontext und Konversationen zusammenführt. Weiterführende Informationen zur Persistenzunterstützung von*Orchestra*finden Sie in Abschnitt**[Sektion: Persistenz]**.
Als weiteres interessantes Feature bietet*Orchestra*die Möglichkeit, einen View-Controller für eine Ansicht zu definieren. Dabei handelt es sich um eine Managed-Bean, die über eine Annotation oder eine Namenskonvention an eine Ansicht gebunden ist. Der View-Controller wird dann bei der Ausführung des Lebenszyklus an mehreren Stellen, wie etwa kurz vor dem Rendern der Ansicht, benachrichtigt und kann entsprechend darauf reagieren. Abschnitt**[Sektion: View-Controller]**zeigt Details zu den View-Controllern und beschreibt wie sie in der Praxis eingesetzt werden können.
Abschließend finden Sie in Abschnitt**[Sektion: *MyGourmet 17 Spring*: Apache MyFaces Orchestra]**das Beispiel*MyGourmet 17 Spring*, in dem einige*Orchestra*-Features in die Praxis umgesetzt werden.

13.5.1 Konversationen mit Orchestra

Eine Konversation ist genau wie die Session oder die HTTP-Anfrage als Gültigkeitsbereich für Managed-Beans einsetzbar. Die Lebensdauer einer Konversation beginnt in*Orchestra*mit dem ersten Zugriff auf eine Bean im Conversation-Scope. Wie wird allerdings festgelegt, wie lange eine Konversation dauert? Bei den

Standard-Gültigkeitsbereichen ist das Ende der Lebensdauer klar definiert. Der Request-Scope endet nach jeder Anfrage, der Session-Scope endet mit dem Ende der Session und so weiter.
*Orchestra*definiert zwei verschiedene Strategien, um eine Konversation zu beenden. Konversationen mit manueller Lebensdauer (Manual-Scope) müssen explizit über die Orchestra-API oder ein spezielles Tag in der Seitendeklaration beendet werden. Alternativ bietet*Orchestra*auch eine automatische Verwaltung der Lebensdauer (Access-Scope). Dabei bleibt die Konversation über mehrere Anfragen hinweg aktiv, bis nicht mehr auf die Eigenschaften oder Methoden einer Bean in der Konversation zugegriffen wird.

13.5.1.1 Konfiguration von Orchestra

*Orchestra*setzt für die Verwaltung der Conversation-Scopes auf*Spring*und die ab Spring in Version 2.0 verfügbaren benutzerdefinierten Scopes. Dadurch ist es zwingend erforderlich, dass alle Managed-Beans im Conversation-Scope über*Spring*definiert werden. Sie sollten das allerdings nicht als Einschränkung, sondern als Chance betrachten - Sie können auch sonst von der Konfiguration Ihrer Beans über*Spring*profitieren .
Die Basiskonfiguration von*Spring*haben wir ja bereits in Abschnitt**[Sektion: Konfiguration von Spring]**behandelt. Listing**Spring-Konfiguration der Conversation-Scopes von Orchestra**zeigt alle für*Orchestra*zusätzlich notwendigen Einträge in der Spring-Konfiguration.

```
<import resource=
    "classpath*:/META-INF/spring-orchestra-init.xml"/>

<!-- Configure additional Orchestra scopes -->
<bean class="org.springframework.beans.factory.config
    .CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="manual">
        <bean class="org.apache.myfaces.orchestra
            .conversation.spring.SpringConversationScope">
          <property name="timeout" value="30"/>
          <property name="lifetime" value="manual"/>
        </bean>
      </entry>
      <entry key="access">
        <bean class="org.apache.myfaces.orchestra
            .conversation.spring.SpringConversationScope">
          <property name="timeout" value="30"/>
          <property name="lifetime" value="access"/>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

Zuerst werden mit dem`import`-Element alle im Classpath befindlichen Spring-Konfigurationen für*Orchestra*importiert. In diesen importierten Dateien werden einige Basiseinstellungen vorgenommen. Anschließend kommt erneut eine Bean mit der bereits aus Abschnitt**[Sektion: *MyGourmet 16 Spring*: Integration von Spring]**bekannten Klasse`CustomScopeConfigurer`zum Einsatz, um die beiden Conversation-Scopes von*Orchestra*zu definieren.
Die beiden Gültigkeitsbereiche können mit den im Attribut`key`angegebenen Namen bei der Definition von Managed-Beans verwendet werden. Mit`access`wird eine Bean im Access-Scope und mit`manual`eine Bean im Manual-Scope erstellt. Die beiden Scope-Definitionen unterscheiden sich in der Konfiguration nur durch die unterschiedliche Initialierung der Eigenschaft`lifetime`. Die Eigenschaft`timeout`gibt die Zeitspanne in Minuten an, die*Orchestra*nach dem letzten Zugriff auf eine Konversation abwartet, bevor sie automatisch beendet wird.
Bei der Wahl dieses Timeouts sollten Sie das richtige Verhältnis zum Session-Timeout im Auge behalten. Wenn der Session-Timeout kleiner ist als der Timeout der Konversation, wird bei kompletter Inaktivität des Benutzers die Konversation bereits mit dem Session-Timeout beendet. Arbeitet der Benutzer allerdings mit

einem anderen Bereich der Applikation weiter, ohne auf die offene Konversation zuzugreifen, wird der Session-Timeout bei jeder Anfrage zurückgesetzt. In diesem Fall greift der Timeout der Konversation.
Damit*Orchestra*ordnungsgemäß funktioniert, muss der in Listing**Konfiguration für Orchestra in der web.xml**gezeigte Listener in derweb.xmlkonfiguriert werden.

```
<listener>
  <listener-class>
    org.apache.myfaces.orchestra.conversation.servlet
        .ConversationManagerSessionListener
  </listener-class>
</listener>
```

13.5.1.2 Konversationen im Einsatz

Zur Demonstration der neuen Scopes werden wir zuerst den Gültigkeitsbereich der Managed-BeancustomerBeanvon Session auf Access umstellen. Dazu muss nur der Wert desvalue-Elements der Annotation@Scopevonsessionaufaccessgeändert werden. Listing**Definition einer Bean im Access-Scope**zeigt die Klassendefinition mit den Annotationen. Werden die Beans mit XML-Elementen definiert, muss dort das Attributscopeauf den Wertaccessgesetzt werden.

```
@Named("customerBean")
@Scope("access")
public class CustomerBean {
...
}
```

Wie verhält sich diese Managed-Bean in der Praxis? In unserem Beispiel wird die Bean mit dem NamencustomerBeanin den AnsichtenshowCustomer.xhtmlundeditCustomer.xhtmlverwendet. Durch den initialen Aufruf vonshowCustomer.xhtmlwird neben der Bean auch die Konversation erstellt. Der Name der Konversation leitet sich automatisch vom Namen der Managed-Bean ab und lautet ebenfallscustomerBean. Navigiert der Benutzer dann weiter zueditCustomer.xhtml, um die Daten zu ändern und anschließend wieder zurück, bleibt die Konversation offen. Erst wenn der Benutzer eine Ansicht aufruft, in der kein Zugriff auf die Eigenschaften und Methoden der BeancustomerBeanerfolgt, wird die Konversation beendet. In Abschnitt**[Sektion: *MyGourmet 17 Spring*: Apache MyFaces Orchestra]**werden wir die Lebensdauer von Konversationen anhand von*MyGourmet 17 Spring*noch etwas genauer analysieren.
Konversationen mit automatischer Lebensdauer sind zwar sehr praktisch, können aber in manchen Fällen zu unerwarteten Ergebnissen führen. Wenn zum Beispiel während der Ausführung einer Ajax-Anfrage kein Zugriff auf die Managed-Bean erfolgt, wird sie von*Orchestra*entfernt und beim nächsten Zugriff wieder neu erstellt. Das kann zu sehr unangenehmen Nebeneffekten führen, die nur schwer zu lokalisieren sind. Ebenso gut kann der Fall eintreten, dass eine Managed-Bean sehr lange im Speicher bleibt, wenn sie - vielleicht unbeabsichtigt - in irgendeiner Form auf mehreren Seiten in Folge referenziert wird. Das kann insbesondere in Kombination mit der Persistenzverwaltung von*Orchestra*zu ungewollten Effekten führen. Bei manueller Lebensdauer müssen Sie selbst bestimmen, wie lange die Konversation im Speicher gehalten wird.
In der Praxis ist es nicht immer erwünscht, dass jede Bean eine eigene Konversation erhält. Mit der Annotation@ConversationNamekann deswegen der Name der Konversation explizit im Elementvalueangegeben werden. Wenn zwei Managed-Beans denselben Namen für ihre Konversation definieren und gleichzeitig verwendet werden, landen sie in derselben Konversation.
Das Beispiel in Listing**Definition zweier Beans in derselben Konversation mit Annotation**umfasst die Beans mit den Namenbean1undbean2im Manual-Scope, deren Konversationsname explizit aufconversationgesetzt ist. Die Konversation wird beim ersten Zugriff auf eine der Beans erstellt und bleibt so lange aktiv, bis sie manuell oder durch einen Timeout beendet wird. Werden in diesem Zeitraum beide Beans referenziert, landen sie in derselben Konversation. Ohne@ConversationNamewürde es in diesem Fall eine Konversation mit dem Namenbean1und eine mit dem Namenbean2geben.

```
@Named("bean1")
@Scope("manual")
```

```
@ConversationName("conversation")
public class SomeBean {
...
}

@Named("bean2")
@Scope("manual")
@ConversationName("conversation")
public class AnotherBean {
...
}
```

Der Name der Konversation kann auch in XML über das Attribut`conversationName`gesetzt werden. Listing**Definition zweier Beans in derselben Konversation mit XML**zeigt die XML-Variante des Beispiels inklusive der Definition des Orchestra-Namensraums.

```
<beans ... xmlns:orchestra="http://myfaces.apache.org/orchestra"
    xsi:schemaLocation="http://myfaces.apache.org/orchestra
    http://myfaces.apache.org/orchestra/orchestra.xsd">
...
<bean id="bean1" class="SomeBean" scope="manual"
    orchestra:conversationName="conversation"/>
<bean id="bean2" class="AnotherBean" scope="manual"
    orchestra:conversationName="conversation"/>
</beans>
```

Sehen wir uns auch noch eine Managed-Bean im Manual-Scope an. Als Beispiel werden wir in*MyGourmet 17 Spring*einen kleinen Wizard zum Anlegen eines Kunden in zwei Schritten erstellen. Der Wizard umfasst die Ansichten`addCustomer1.xhtml`und`addCustomer2.xhtml`, in denen die Managed-Bean`addCustomerBean`verwendet wird. Listing**Manuelles Beenden einer Konversation über die Java-API**zeigt den relevanten Teil des Sourcecodes dieser Bean im Manual-Scope. Die Bean und die Konversation werden beim ersten Zugriff erstellt. Da es sich aber diesmal um eine manuelle Konversation handelt, wird sie nicht automatisch beendet. Diese Aufgabe wird manuell beim Speichern oder Abbrechen des Wizards in den Action-Methoden über einen Aufruf der Methode`invalidate()`auf der aktuellen Konversation erledigt. Die aktuelle Konversation wird von`Conversation.getCurrentInstance()`zurückgeliefert. Listing**Manuelles Beenden einer Konversation über die Java-API**zeigt die beiden Methoden`save()`zum Speichern des neuen Kunden und`cancel()`zum Abbrechen des Wizards.

```
@Named("addCustomerBean")
@Scope("manual")
public class AddCustomerBean extends CustomerBeanBase {
  ...
  public String save() {
    customerService.save(customer);
    Conversation.getCurrentInstance().invalidate();
    return ViewIds.CUSTOMER_LIST_VIEW_ID;
  }
  public String cancel() {
    Conversation.getCurrentInstance().invalidate();
    return ViewIds.CUSTOMER_LIST_VIEW_ID;
  }
}
```

Eine Konversation kann auch mit dem Tag`endConversation`aus der Orchestra-Tag-Bibliothek beendet werden, die über den Namensraum`http://myfaces.apache.org/orchestra`verfügbar ist. Dieses Tag wird als Kind-Tag zu einer Befehlskomponente hinzugefügt und bekommt im Attribut`name`den Namen der zu

beendenden Konversation. Listing**Manuelles Beenden einer Konversation in der Seitendeklaration**zeigt die Cancel-Schaltfläche des Beispiels mit dem Tag`endConversation`. Beide Varianten erfüllen denselben Zweck.

```
<h:commandButton id="cancel" value="#{msgs.cancel}"
    action="#{addCustomerBean.cancel}" immediate="true">
  <o:endConversation name="addCustomerBean"/>
</h:commandButton>
```

Wenn Sie*MyGourmet 17 Spring*starten und im Browser durch die Anwendung klicken, werden Sie bemerken, dass jede URL der Anwendung den Request-Parameter mit dem Namen`conversationContext`aufweist.*Orchestra*benutzt den Wert dieses Parameters, um verschiedene Fenster und Tabs derselben Browserinstanz zu unterscheiden. Wie Sie bereits wissen, werden Konversationen in der Session abgelegt. Damit sich diese Konversationen eindeutig auf ein Fenster oder ein Tab des Browsers abbilden lassen, führt*Orchestra*das Konzept des Konversationskontexts ein. Der Wert des Parameters`conversationContext`ist der Bezeichner des aktuellen Kontexts und stellt die Verbindung her. Ein solcher Kontext ist ein Container für alle Konversationen eines Fensters oder Tabs.
Wenn die Anwendung in einem neuen Fenster oder Tab erneut geöffnet wird, fehlt dieser Parameter und*Orchestra*erstellt einen neuen Konversationskontext mit einem neuen Bezeichner. Sie können das überprüfen, indem Sie*MyGourmet 17 Spring*in zwei Tabs Ihres Browsers parallel starten. Sie werden sehen, dass sich der Wert des Parameters`conversationContext`unterscheidet.
Abbildung**Session mit Orchestra-Konversationen**stellt eine Session mit zwei Konversationskontexten dar, in denen jeweils die gleichen Konversationen aktiv sind. Diese Situation entsteht, wenn die Anwendung in zwei Fenstern oder Tabs gleichzeitig benutzt wird. Erst durch den Kontext ist es möglich, dass eine Konversation mehrfach existiert.

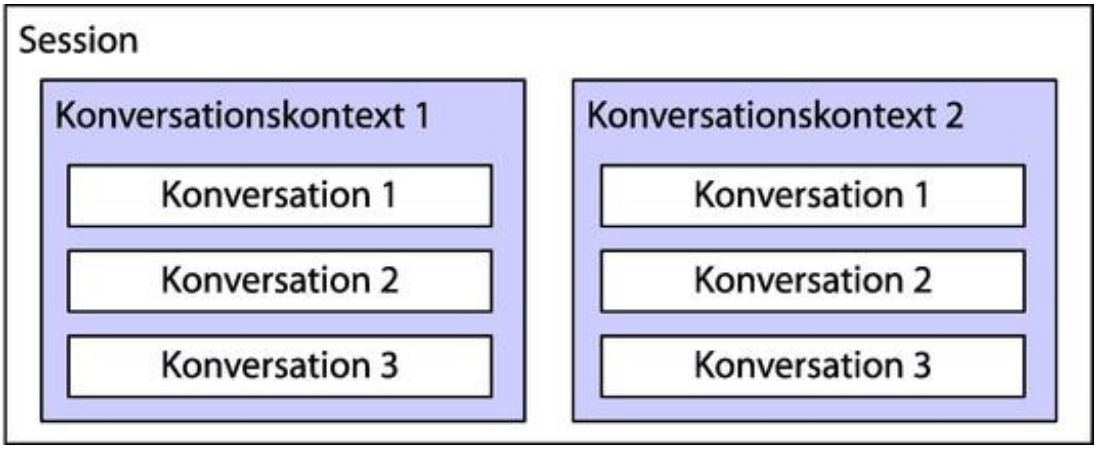


Abbildung:Session mit Orchestra-Konversationen

13.5.2 Persistenz

Bei Webanwendungen, die zum Persistieren der Daten die*Java Persistence API (JPA)*einsetzen, taucht bei der Abbildung von Geschäftsprozessen ein weiteres Problem auf. Neben der Lebensdauer der Managed-Beans ist die Lebensdauer des Persistenzkontexts ein weiteres entscheidendes Kriterium. Ist diese Dauer zu kurz gewählt, kann es zu Problemen beim Laden von Daten kommen (die`LazyInitializationException`dürfte vielen Entwicklern ein Begriff sein). Ist sie hingegen zu lang, steigt der Speicherverbrauch der Anwendung zunehmend an.
Die meisten Probleme rühren daher, dass die Lebensdauer des Persistenzkontexts und die der Managed-Beans nicht übereinstimmen. In vielen Fällen ist die Lebensdauer der Beans länger als die des Persistenzkontexts. Da Managed-Beans oft geladene Entitäten halten, kommt es bei einem Zugriff auf eine Eigenschaft, die vor dem Schließen des Persistenzkontexts noch nicht geladen wurde, zur gefürchteten`LazyInitializationException`.
*Orchestra*bietet auch für dieses Problem eine Lösung an, indem es den Persistenzkontext an eine Konversation bindet und für die Lebensdauer der Konversation offen hält. Wie bei den Konversationen gilt, dass der Persistenzkontext so lange wie benötigt, aber so kurz wie möglich, offen bleiben soll. Mit dem Ende

der Konversation wird auch der Persistenzkontext geschlossen.
Da der Persistenzkontext während der ganzen Lebensdauer der Konversation offen bleibt, kann ohne Probleme direkt mit den von der Datenbank geladenen Entitäten gearbeitet werden.
Ein wichtiger Punkt ist, dass jede Konversation einen eigenen Persistenzkontext und somit auch einen eigenen JPA-Entity-Manager erhält. Entitäten, die in einer Konversation geladen wurden, können daher nicht so ohne Weiteres in einer anderen Konversation verwendet werden, da sie an einen Entity-Manager gebunden sind. Wenn Sie dieselbe Entität gleichzeitig in mehreren Konversationen benötigen, sollte sie auch in jeder Konversation eigenständig geladen werden. Wenn Sie eine Entität von einer Konversation zu einer anderen weiterreichen wollen, zum Beispiel beim Übergang von einer Listenansicht auf eine Detailansicht, sollten Sie nur die ID übergeben und die Entität in der zweiten Konversation neu laden.
Alternativ können Sie auch mehrere Managed-Beans in derselben Konversation ablegen, damit sie automatisch denselben Persistenzkontext verwenden. Wie das funktioniert haben wir ja im letzten Abschnitt gezeigt. Sie sollten sich allerdings bewusst sein, dass der Speicherverbrauch mit jeder geladenen Entität ansteigt.
Wir werden hier nicht weiter auf die zusätzlich notwendige Konfiguration und die praktischen Aspekte der Persistenzunterstützung in*Orchestra*eingehen. Nachdem dieser Teil von*Orchestra*nur in Kombination mit JPA funktioniert, werden wir dieses Thema erst in Kapitel**Sektion: MyGourmet Fullstack Spring -- JSF, Spring, Orchestra und JPA kombiniert**behandeln. Dort zeigen wir Ihnen dann anhand von*MyGourmet*, wie Sie eine komplette Anwendung mit JSF,*Spring*,*Orchestra*und*JPA*erstellen.
Doch bevor es so weit ist, zeigen wir Ihnen im nächsten Abschnitt erst noch, wie Sie in*Orchestra*View-Controller definieren.

13.5.3 View-Controller

Als weiteres interessantes Feature bietet*Orchestra*die Möglichkeit, einen View-Controller für eine Ansicht zu definieren. Dabei handelt es sich um eine Managed-Bean, die an eine Ansicht gebunden ist und bei der Ausführung des Lebenszyklus an mehreren Stellen benachrichtigt wird. Ab JSF 2.0 können Sie alternativ auch System-Events einsetzen, um auf Ereignisse des Lebenszyklus zu reagieren.
Die Verbindung zwischen Ansicht und View-Controller kann mit der Annotation`@ViewController`auf der Klasse der Managed-Bean oder über eine Namenskonvention definiert werden. Die Annotation`@ViewController`nimmt im Element`viewIds`eine Liste der View-IDs auf, für die die Bean als View-Controller fungieren soll. Listing**Annotationsbasierte Definition eines View-Controllers**zeigt die Klasse`AddCustomerBean`, die als View-Controller für die Ansichten mit den View-IDs`/addCustomer1.xhtml`und`/addCustomer2.xhtml`definiert ist.

```
@Named("addCustomerBean")
@Scope("manual")
@ViewController(viewIds = {"/addCustomer1.xhtml",
    "/addCustomer2.xhtml"})
public class AddCustomerBean
    extends CustomerBeanBase {

  @InitView
  public void createCustomer() {
    if (customer == null) {
      customer = customerService.createNew();
    }
  }
  ...
}
```

Ein View-Controller kann auch über eine Namenskonvention mit einer Ansicht verbunden werden. Der Name der Managed-Bean (nicht der Klassenname), die als View-Controller zum Einsatz kommt, wird dabei aus dem Namen der View-ID berechnet. Dazu werden in der View-ID alle Zeichen nach einem Schrägstrich in Großbuchstaben umgewandelt. Anschließend werden alle Schrägstriche und sämtliche Zeichen ab dem ersten

Punkt entfernt. Entsteht dadurch ein ungültiger Bean-Name, wird das Präfix_vorangestellt.
Tabelletab:orchestra-namemapperzeigt einige Beispiele.

View-ID	Bean-Name
`addCustomer1.xhtml`	addCustomer1
`customer/registration1.xhtml`	customerRegistration1
`1addCustomer.xhtml`	_1addCustomer

Die Definition der View-Controller über die Namenskonvention hat einige Nachteile gegenüber der annotationsbasierten Variante. Annotationen bieten neben der beliebigen Wahl der Bean-Namen auch die Möglichkeit, eine Bean als View-Controller für mehrere Ansichten zu verwenden. Das Beispiel aus Listing**Annotationsbasierte Definition eines View-Controllers**kann daher über die Namenskonvention in der Form nicht realisiert werden.
Sobald eine Ansicht mit einem View-Controller verbunden ist, kann*Orchestra*zu bestimmten Zeiten während der Ausführung des Lebenszyklus den View-Controller benachrichtigen. Dabei werden folgende Methoden des View-Controllers aufgerufen:

- Nach der Restore-View-Phase wird die mit`@InitView`annotierte Methode des View-Controllers aufgerufen. Wenn die Methode den Namen`initView()`hat, kann die Annotation entfallen.
- Vor der Invoke-Application-Phase wird die mit`@PreProcess`annotierte Methode aufgerufen. Wenn die Methode den Namen`preProcess()`hat, kann die Annotation entfallen.
- Vor der Render-Response-Phase wird die mit`@PreRenderView`annotierte Methode aufgerufen. Wenn die Methode den Namen`preRenderView()`hat, kann die Annotation entfallen.

Im View-Controller in Listing**Annotationsbasierte Definition eines View-Controllers**ist zum Beispiel die Methode`createCustomer()`mit`@InitView`annotiert. Diese Methode wird für die beiden Ansichten`/addCustomer1.xhtml`und`/addCustomer2.xhtml`nach jeder Restore-View-Phase aufgerufen, um die Variable`customer`zu initialisieren, falls sie`null`ist.
View-Controller bilden die Grundlage für ein weiteres Feature von*Orchestra*. Mit der Annotation`@ConversationRequire`kann eine Reihe von Ansichten mit einer bestimmten Konversation verknüpft werden. Das ist besonders für Geschäftsprozesse interessant, die sich über mehrere Ansichten mit einer definierten Einstiegsseite erstrecken. Solange die Konversation noch nicht existiert, kann der Benutzer nur auf die Einstiegsseite zugreifen. Jeder Zugriff auf eine andere Seite des Prozesses führt, je nach Konfiguration, zu einem Redirect oder dem Auslösen einer Navigation. Erst wenn die Konversation über die Einstiegsseite erzeugt wird, sind auch die anderen Ansichten des Prozesses verfügbar.
Sehen wir uns das anhand des letzten Beispiels an. Listing**View-Controller für einen Prozess mit mehreren Ansichten**zeigt nochmals die Klasse`AddCustomerBean`, diesmals allerdings mit der Annotation`@ConversationRequire`. Im Element`conversationNames`der Annotation werden alle Konversationen angegeben, auf deren Existenz überprüft werden soll, bevor ein Benutzer auf die Ansichten zugreifen kann. Eine Ausnahme bilden alle im Element`entryPointViewIds`angegebenen View-IDs. Diese Ansichten können auch ohne eine existierende Konversation aufgerufen werden - irgendwie muss der Ablauf ja gestartet werden. In unserem Beispiel ist der Einstiegspunkt des Ablaufs die Ansicht`addCustomer1.xhtml`. Erfolgt ein Zugriff auf`addCustomer2.xhtml`, ohne dass die Konversation`addCustomerBean`existiert, wird eine Navigation mit der im Element`navigationAction`angegebenen Zeichenkette ausgeführt. Dabei kann es sich um eine globale Navigationsregel oder ab JSF 2.0 auch direkt um eine View-ID handeln. In unserem Beispiel wird einfach auf die erste Ansicht navigiert. Alternativ kann statt des Elements`navigationAction`auch das Element`redirect`mit einer URL für einen Redirect angegeben werden.

```
@Named("addCustomerBean")
@Scope("manual")
@ViewController(viewIds = {"/addCustomer1.xhtml",
    "/addCustomer2.xhtml"})
@ConversationRequire(conversationNames = "addCustomerBean",
    entryPointViewIds = "/addCustomer1.xhtml",
    navigationAction = "/addCustomer1.xhtml")
public class AddCustomerBean extends CustomerBeanBase {
...
```

```
}
```

13.5.4 MyGourmet 17 Spring: Apache MyFaces Orchestra

MyGourmet 17 Spring integriert *Apache MyFaces Orchestra* und zeigt neben der notwendigen Konfiguration auch einige Anwendungsfälle. *Orchestra* bietet ab Version 1.4 ein neues Modul *Core20*, das bereits an JSF 2.0 angepasst ist. *Core20* fasst die aus älteren Versionen von *Orchestra* bekannten Module *Core* mit der Basisfunktionalität und *Core15* mit den im Laufe des Abschnitts vorgestellten Annotationen zusammen. Das Modul wird über eine Abhängigkeit in der `pom.xml` in das Maven-Projekt eingebunden. Details entnehmen Sie bitte direkt dem Sourcecode. Auf die Konfiguration von *Spring* und *Orchestra* werden wir hier nicht mehr näher eingehen.

In *MyGourmet 17 Spring* haben wir den Kundenbereich der Anwendung leicht umgebaut. Die Startseite ist jetzt `customerList.xhtml`, auf der eine Liste aller Kunden in einer `mc:dataTable`-Komponente dargestellt wird. Von dieser Ansicht aus kann der Benutzer zur Detailseite eines Kunden springen, einen neuen Kunden anlegen oder einen bereits vorhandenen Kunden löschen.

Die Bean `customerListBean` im Access-Scope ist als View-Controller der Ansicht definiert. Listing **MyGourmet 17 Spring: View-Controller der Kunden-Übersichtsseite** zeigt die Klasse `CustomerListBean` der Bean. Die Liste der Kunden wird in der Methode `preRenderView` geladen. Da sie mit `@PreRenderView` annotiert ist, wird sie vor jedem Rendern der Ansicht von *Orchestra* aufgerufen. Zum Löschen eines Kunden wird über einen `h:commandLink`-Komponente die Methode `deleteCustomer` aufgerufen. Der zu löschende Kunde wird dabei direkt als Parameter übergeben:

```
<h:commandLink value="#{msgs.delete}" action=
    "#{customerListBean.deleteCustomer(customer)}">
  <f:ajax render="@form"/>
</h:commandLink>
```

Das Löschen eines Kunden wird als Ajax-Anfrage ausgeführt. Der Access-Scope der Bean bedeutet, dass die Konversation der Bean so lange offen bleibt, bis nicht mehr auf sie zugegriffen wird.

```
@Named("customerListBean")
@Scope("access")
@ViewController(viewIds = {"/customerList.xhtml"})
public class CustomerListBean {

  @Inject
  private CustomerService customerService;
  private List<Customer> customerList;

  @PreRenderView
  public void preRenderView() {
    customerList = customerService.findAll();
  }
  public List<Customer> getCustomerList() {
    return customerList;
  }
  public void deleteCustomer(Customer customer) {
```

```
        customerService.delete(customer);
      }
    }
```

Wir haben im Zuge der Änderungen am Kundenbereich in*MyGourmet 17 Spring*alle Operationen für Objekte vom Typ`Customer`im Interface`CustomerService`zusammengefasst. Die Implementierung`CustomerServiceImpl`dieses Interface steht als Bean mit dem Bezeichner`customerService`zur Verfügung. In Listing**MyGourmet 17 Spring: View-Controller der Kunden-Übersichtsseite**sehen Sie, wie die Abhängigkeit zum Service mit`@Inject`definiert wird.
Die Detailseite`showCustomer.xhtml`ist ebenfalls über ein`h:commandLink`-Komponente in der Übersichtsseite erreichbar. Beim Aktivieren der Komponente wird die Methode`showCustomer`der Bean`customerBean`mit der ID des Kunden als Parameter aufgerufen. Die Methode lädt den Kunden und gibt die View-ID der Detailseite für die Navigation zurück. Die Managed-Bean`customerBean`ist als View-Controller für`editCustomer.xhtml`,`showCustomer.xhtml`und`editAddress.xhtml`definiert und wird im Access-Scope angelegt. Die Klasse`AddressBean`ist in dieser Klasse aufgegangen. Listing**MyGourmet 17 Spring: View-Controller der Kundenansichten**zeigt den Kopf der Klasse`CustomerBean`. Die Klasse`CustomerBean`ist von der abstrakten Klasse`CustomerBeanBase`abgeleitet, die einige Basisfunktionalität für Kundenansichten beinhaltet, die auch von`AddCustomerBean`beim Anlegen eines Kunden verwendet wird - doch dazu später mehr.

```
@Named("customerBean")
@Scope("access")
@ViewController(viewIds = {"/editCustomer.xhtml",
    "/showCustomer.xhtml", "/editAddress.xhtml"})
public class CustomerBean extends CustomerBeanBase {
...
}
```

Werfen wir noch einen Blick auf die Konversation der Managed-Bean`customerBean`. Wenn der Benutzer von der Übersichtsseite auf die Detailseite eines Kunden geht, wird neben der Managed-Bean auch die Konversation`customerBean`mit automatisch verwalteter Lebensdauer erstellt (Access-Scope). Die Konversation wird erst dann beendet, wenn während einer Anfrage keine Zugriffe mehr auf die Bean`customerBean`erfolgen. Abbildung**Lebensdauer der Konversation customerBean**zeigt eine Serie von Zugriffen auf die Ansichten`customerList.xhtml`,`showCustomer.xhtml`,`editCustomer.xhtml`und`editAddress.xhtml`. Der Balken am unteren Ende der Abbildung zeigt die Lebensdauer der Konversation`customerBean`.

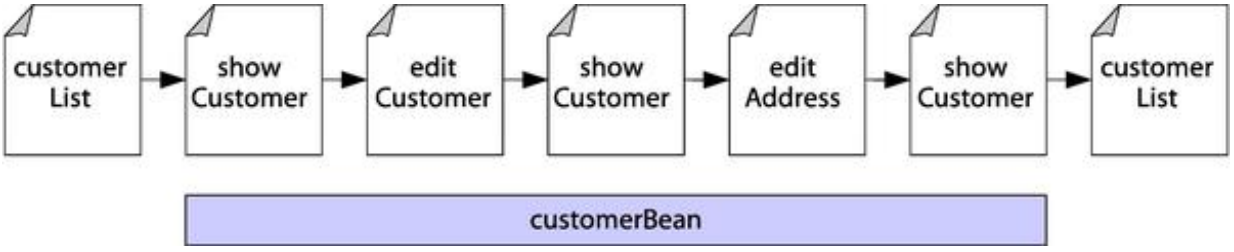


Abbildung:Lebensdauer der Konversation customerBean

Der Wizard zum Anlegen eines neuen Kunden ist ebenfalls von der Übersichtsseite aus erreichbar. Der Ablauf besteht aus den beiden Ansichten`addCustomer1.xhtml`zur Eingabe der Basisdaten des Kunden und`addCustomer2.xhtml`zur Eingabe einer Adresse. Die Managed-Bean`addCustomerBean`ist der View-Controller für beide Ansichten und liegt diesmal im Manual-Scope. Listing**MyGourmet 17 Spring: View-Controller des Wizards zum Anlegen eines Kunden**zeigt die Klasse`AddCustomerBean`. Die Bean bekommt auch den Service`customerService`injiziert, der zum Erzeugen einer neuen`Customer`-Instanz in der Methode`createCustomer`verwendet wird. Da diese Methode mit`@InitView`annotiert ist, wird sie von*Orchestra*nach der Restore-View-Phase aufgerufen, wenn der Lebenszyklus für eine der beiden verknüpften Ansichten ausgeführt wird. So ist gewährleistet, dass immer eine Instanz der Klasse`Customer`existiert.

```
@Named("addCustomerBean")
@Scope("manual")
@ViewController(viewIds = {"/addCustomer1.xhtml",
    "/addCustomer2.xhtml"})
@ConversationRequire(conversationNames = "addCustomerBean",
    entryPointViewIds = "/addCustomer1.xhtml",
    navigationAction = "/addCustomer1.xhtml")
public class AddCustomerBean extends CustomerBeanBase {

  @Inject
  private CustomerService customerService;

  @InitView
  public void createCustomer() {
    if (customer == null) {
      customer = customerService.createNew();
    }
  }
  public String save() {
    customerService.save(customer);
    Conversation.getCurrentInstance().invalidate();
    return ViewIds.CUSTOMER_LIST_VIEW_ID;
  }
  public String cancel() {
    Conversation.getCurrentInstance().invalidate();
    return ViewIds.CUSTOMER_LIST_VIEW_ID;
  }
  public Address getAddress() {
    return customer.getAddresses().get(0);
  }
}
```

Mit der Annotation`@ConversationRequire`auf dem View-Controller wird sichergestellt, dass ein Benutzer im Browser nur auf die Einstiegsseite`addCustomer1.xhtml`des Wizards zugreifen kann, solange die Konversation`addCustomerBean`nicht existiert. Als Einstiegsseiten gelten alle Ansichten, deren View-IDs im Element`entryPointViewIds`der Annotation aufgeführt sind. Ruft der Benutzer im Browser die Ansicht`addCustomer2.xhtml`auf, ohne dass die Konversation`addCustomerBean`existiert, oder tritt nach dem Zugriff auf die erste Seite ein Timeout der Konversation auf, wird der Navigation-Handler mit der in`navigationAction`angegebenen Zeichenkette`/addCustomer1.xhtml`aufgerufen. Durch die implizite Navigation von JSF ab Version 2.0 wird dadurch im Browser direkt die Ansicht mit der entsprechenden View-ID angezeigt. Die Ansicht ist als Einstiegsseite deklariert und*Orchestra*erzeugt beim ersten Zugriff auf die Bean die Konversation. Nach dem Ausfüllen der ersten Seite kann jetzt auch die zweite Seite zum Abschließen der Registrierung aufgerufen werden.
Da die Bean`addCustomerBean`im Manual-Scope liegt, müssen wir uns selbst um das Beenden der Konversation kümmern. Dazu wird in den beiden Action-Methoden`save`und`cancel`die Methode`invalidate`auf der aktuellen Konversation aufgerufen, die wir über`Conversation.getCurrentInstance()`erhalten.

15 Tobago - - JSF und mehr

Für die Entwicklung von Webapplikationen unter Verwendung des JSF-Standards stehen eine Vielzahl von Frameworks zur Verfügung. In der Regel bieten diese Frameworks Controls und weitere Features zur Gestaltung des Frontends einer Applikation an. Eines dieser Frameworks nennt sich*Tobago*, auf das wir im Folgenden genauer eingehen werden.

15.1 Tobago - - ein Überblick

Die Entwicklung von*Tobago*startete im Jahr 2002. Seit 2005 ist*Tobago*ein Open-Source-Projekt und wurde wie das Framework*Trinidad*im Jahr 2006 ein Subprojekt des*Apache MyFaces*-Projekts.
*Tobago*bietet eine komfortable Möglichkeit, Webapplikationen auf Basis weiterentwickelter JSF-Standardcontrols zu erstellen. Die Positionierung der einzelnen Controls wird dabei im Gegensatz zu anderen Frameworks über einen Layout-Manager gesteuert, auf den im weiteren Verlauf noch genauer eingegangen wird.
*Tobago*ist jedoch mehr als nur eine Komponentenbibliothek, die auf den JSF-Standard aufsetzt. Der Fokus von*Tobago*liegt vielmehr darin, Geschäftsanwendungen ohne die direkte Nutzung von HTML, CSS und JavaScript zu erstellen. Zudem folgt eine*Tobago*-Seitenbeschreibung mehr der Entwicklung eines Desktop-User-Interface als einer Webseite. Eine Seitenbeschreibung in*Tobago*benötigt kein HTML, somit auch nicht das typische HTML-Gerüst aus DIV-Tags, um die Controls auf der Seite zu positionieren.
Das Design und die Styles werden in*Tobago*über CSS geregelt. Zurzeit unterstützt*Tobago*lediglich Themes für HTML/CSS-Clients (Renderer für WML und XSL:FO stellen ein*Proof of Concept*dar). Um eine gewisse Unabhängigkeit zu erreichen, sollte in der Seitenbeschreibung weder HTML noch CSS oder JavaScript verwendet werden. Eine Seitenbeschreibung in*Tobago*beinhaltet in der Regel nur*Tobago*- und JSF-Tags.
Das*Tobago*-Projekt steht unter**http://myfaces.apache.org/tobago/**zur Verfügung und bietet neben weiterführender Dokumentation noch Folgendes:

- Controlübersicht Die Demonstration stellt vorhandene Controls und deren Umgang beziehungsweise Konfigurationsmöglichkeiten vor.**http://people.apache.org/repo/m2-snapshot-repository/org/apache/myfaces/tobago/tobago-example-demo/**
- Adressbuchapplikation Das Adressbuch ist eine kleine, in sich geschlossene Anwendung und zeigt den Umgang mit*Tobago*in einer komplexeren Umgebung.**http://people.apache.org/repo/m2-snapshot-repository/org/apache/myfaces/tobago/tobago-example-addressbook/**
- Basis-Setup Das Basis-Setup kann als initiales Projekt-Setup verwendet werden.**http://people.apache.org/repo/m2-snapshot-repository/org/apache/myfaces/tobago/tobago-example-blank/**

15.2 Tobago

und MyGourmet

Um einen Einblick in*Tobago*zu bekommen, gehen wir auf die bereits bekannte Applikation*MyGourmet*ein. Abbildung**Tobago und MyGourmet**zeigt eine mit*Tobago*erstellte Webseite der Applikation.

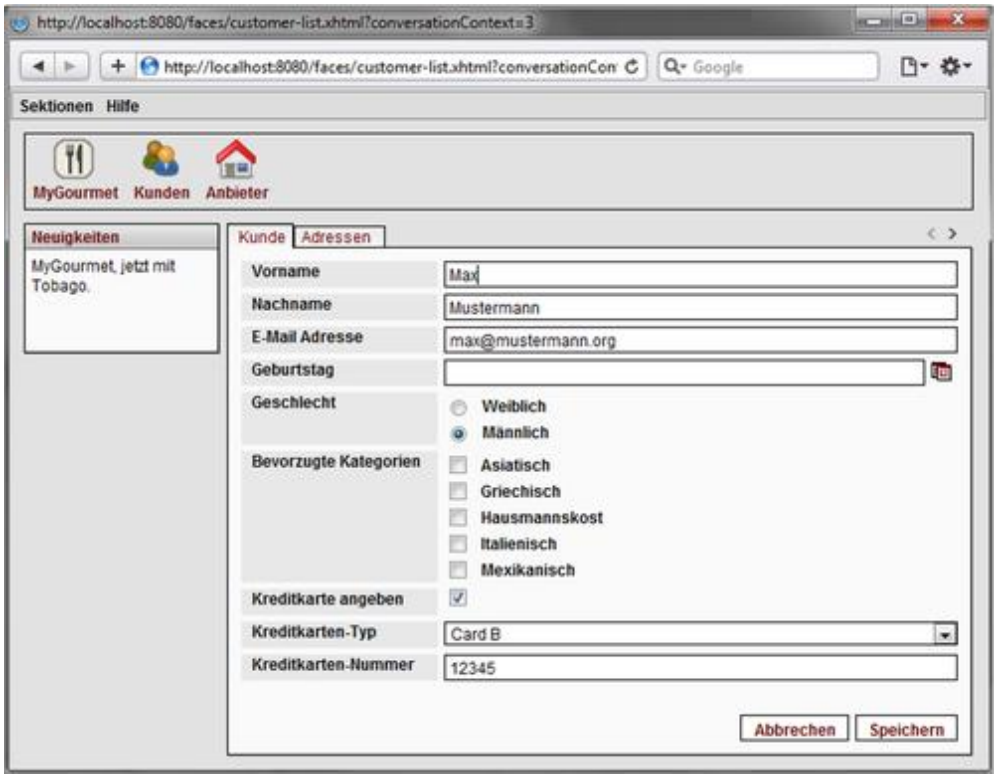


Abbildung:Tobago und MyGourmet

Entwicklern von Webapplikationen wird durch den Einsatz von*Tobago*die Erstellung eines Frontends erleichtert, indem sie sich nicht mit einer Vielzahl verschiedener Technologien auseinandersetzen müssen, sondern sich lediglich auf ein Framework konzentrieren können.
Listing**Seitendeklaration mit Tobago und Facelets**zeigt einen Ausschnitt aus der für die Seitenerstellung verwendeten Datei`customer.xhtml`. Für die Seitenbeschreibung mittels*Tobago*wurde kein HTML verwendet und das generelle Layout der Seite wurde lediglich durch Angabe der vorhandenen Zeilen bzw. Spalten und eine jeweilige Größendefinition geregelt. Auf die möglichen Definitionen und deren Bedeutung wird im Abschnitt**[subsec Grid-Layout]**eingegangen.

```
<tc:tabGroup switchType="reloadTab">
  <tc:tab label="#{msgs.title_show_customer}">
    <tc:panel>
      <f:facet name="layout">
        <tc:gridLayout rows="fixed;fixed;..."/>
      </f:facet>
      <tx:in label="#{msgs.first_name}" required="true"
          value="#{customerController.customer.firstName}"/>
      ....
      <tx:selectBooleanCheckbox label="#{msgs.use_credit_card}"
          value="#{customerController.customer.useCreditCard}">
        <f:facet name="change">
          <tc:command action=
              "#{customerController.changeCreditCardUsage}"/>
        </f:facet>
      </tx:selectBooleanCheckbox>
      ...
      <tc:cell/>
        <tc:panel>
          <f:facet name="layout">
            <tc:gridLayout columns="*;fixed;fixed"/>
```

```
            </f:facet>
            <tc:cell/>
            <tc:button label="#{msgs.cancel}" immediate="true"
                action="#{customerController.cancel}"/>
            <tc:button label="#{msgs.save}" transition="true"
                action="#{customerController.saveCustomer}"/>
          </tc:panel>
        </tc:panel>
    </tc:tab>
    <tc:tab label="#{msgs.title_addresses}">
      ...
    </tc:tab>
  </tc:tabGroup>
```

Um bestehende Applikationen auf*Tobago*zu portieren, sind nur einige Schritte notwendig. In der Applikation*MyGourmet*wurden nur wenige Anpassungen vorgenommen, um sie auf*Tobago*umzustellen. Unter Verwendung der bisherigen Services wurden neben der Seitenbeschreibung folgende Änderungen durchgeführt:

- `web.xml`
  In der`web.xml`wurde das*Resource-Servlet*von*Tobago*eingetragen. Das*Resource-Servlet*wird verwendet, um CSS, JS und Grafiken zu laden. Die Angabe des*Resource-Servlets*ist für den Fall, dass die Ressourcen als Bibliothek eingebunden werden, erforderlich. Bei entpackten Ressourcen ist die Angabe hingegen nicht notwendig. Listing**Resource-Servlet in der web.xml**zeigt die Deklaration eines*Resource-Servlets*innerhalb der`web.xml`.

  ```
  <servlet>
    <servlet-name>ResourceServlet</servlet-name>
    <servlet-class>
      org.apache.myfaces.tobago.servlet.ResourceServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ResourceServlet</servlet-name>
    <url-pattern>
      /org/apache/myfaces/tobago/renderkit/*
    </url-pattern>
  </servlet-mapping>
  ```

- `faces-config.xml`
  *Tobago*unterstützt in der Version 1.0.x kein JSF 2.0 und somit auch keine implizite Navigation. Daher müssen wir die für die Navigation in*MyGourmet*notwendigen Navigationsregeln in der`faces-config.xml`eintragen. Die bereits über Annotationen in*Spring*definierten Services können ohne Änderungen weiter verwendet werden.

Abschließend muss die Konfigurationsdatei`tobago-config.xml`erstellt werden. In der`tobago-config.xml`(siehe Listing**Konfiguration in der tobago-config.xml**) werden das zu verwendende Theme und das Ressourcenverzeichnis eingetragen.

```
<tobago-config>
  <theme-config>
    <default-theme>speyside</default-theme>
  </theme-config>
```

```
    <resource-dir>tobago-resource</resource-dir>
    <resource-dir>
      org/apache/myfaces/tobago/renderkit
    </resource-dir>
  </tobago-config>
```

Als Einstieg für die Entwicklung einer Webapplikation mit Tobago als JSF-Framework kann das bereits in Abschnitt**[Sektion: Tobago -- ein Überblick]**erwähnte Basis-Setup dienen. Es enthält alle notwendigen Konfigurationsdateien zur initialen Erstellung eines Projekts.
Im Folgenden wird auf einzelne der in Abbildung**Tobago und MyGourmet**verwendeten Controls eingegangen.

15.3 Tobago-Controls

Generell bietet*Tobago*zwei Bibliotheken`tc`und`tx`zur Deklaration von Controls. In der`tc`-Bibliothek sind die reinen Controls enthalten, während die`tx`-Bibliothek Kompositionen aus Controls beinhaltet. So ist ein`tx`-Control häufig eine Komposition aus dem eigentlichen Control und einem Label.

15.3.1 Tobago tc:in/tx:in

Ein einzeiliges Eingabefeld wird durch ein`tc:in`dargestellt. Als Erweiterung zu diesem Eingabefeld bietet*Tobago*das`tx:in`. Diese Erweiterung beinhaltet neben dem Eingabefeld noch ein Label. Listing**Das tx:in-Tobago-Control**zeigt die Benutzung des`tx:in`. Das Label wird an das Eingabefeld gebunden und bei einem Klick auf das Label erhält das Eingabefeld den Fokus.

```
<tx:in
  label="First Name"
  value="#{customerController.customer.firstName}"
  required="false" readonly="false"
  disabled="false" rendered="true" />
```

Durch das zugrunde liegende Theme wird die Breite des Labels festgelegt. Jedoch besteht die Möglichkeit, diese Breite durch das Attribut`labelWidth`anzupassen. Ebenfalls werden Access-Keys unterstützt. Beinhaltet das in der Seitenbeschreibung angegebene Label einen Unterstrich, so wird der nachfolgende Buchstabe als Access-Key definiert. Auf der dargestellten Seite wird der Access-Key nunmehr als unterstrichener Buchstabe gekennzeichnet.

15.3.2 Tobago UICommand

*Tobago*bietet mehrere Möglichkeiten zur Unterstützung von Befehlskomponenten. Die Basis-Controls dafür sind das`tc:button`und das`tc:link`. Weitere Controls sind das`tc:menu`und das`tc:toolbar`, auf die wir in den nächsten Abschnitten noch genauer eingehen werden.

```
<tc:button label="Delete Customer" defaultCommand="false"
    action="#{customerController.delete}"
    image="image/delete.png" transition="false">
  <f:facet name="confirmation">
    <tc:out value="Do you want to delete the customer?" />
  </f:facet>
```

```
</tc:button>
```

In Listing**Das tc:button-Tobago-Control**wird die Deklaration eines`tc:button`dargestellt. Das`action`-Attribut beschreibt hier die Funktion, die beim Klick auf den Button ausgeführt werden soll. Zudem kann das`tc:button`-Control mit einem Bild dekoriert werden. Das Bild kann entweder über den Pfad, relativ zur Wurzel der Webapplikation, oder über den Resource-Manager von*Tobago*geladen werden.
Über das Facet`confirmation`wird erreicht, das die Aktion erst ausgeführt wird, wenn die`confirmation message`durch den Benutzer mit einem "OK" bestätigt wurde. Das`transition`-Attribut, per Default auf`true`gesetzt, bewirkt, dass die Seite für die Dauer der Aktion gesperrt ist. Die Seite wird für die Zeit der durchzuführenden Aktion abgedunkelt und eine "Sanduhr" angezeigt. Implizit wird durch diese Vorgehensweise erreicht, dass eine Aktion durch mehrfaches Klicken auf einen Button nicht öfter als gewünscht ausgeführt wird. Ein denkbares Szenario wäre hier das Löschen eines Kunden aus der Applikation`MyGourmet`. Nimmt das Löschen eines Kunden einen längeren Zeitraum in Anspruch, und ein Benutzer klickt währenddessen erneut auf den Löschen-Button, so könnte dies zu einer Fehlermeldung führen, da der Kunde bereits gelöscht ist. Wird das Attribut`defaultCommand`mit dem Wert`true`belegt, so wird die an diesem Button gebundene Aktion auch nach Betätigung der Return-Taste ausgeführt.
Das Tobago-Control`tc:command`kann verwendet werden, um beispielsweise auf Ereignisse zu reagieren. Listing**Das tc:command-Tobago-Control**zeigt die in der*MyGourmet*-Applikation verwendete Deklaration zur Behandlung von Ereignissen.

```
<tx:selectBooleanCheckbox label="Use CC"
    value="#{customerController.customer.useCreditCard}">
  <f:facet name="change">
    <tc:command
        action="#{customerController.changeCreditCardUsage}"/>
  </f:facet>
</tx:selectBooleanCheckbox>
```

15.3.3 Tobago tc:tabGroup

Das Tobago-Control`tc:tabGroup`bietet die Möglichkeit, Inhalte auf verschiedenen Tab-Panels darzustellen. Der Wechsel zwischen den einzelnen Panels erfolgt über die Tab-Reiter. Abbildung**Das tc:tabGroup-Tobago-Control**zeigt das in*MyGourmet*auf der Kundenseite verwendete`tc:tabGroup`-Control.

Abbildung:Das tc:tabGroup-Tobago-Control

Das Ladeverhalten der einzelnen Tab-Panels wird durch das Attribut`switchType`gesteuert. Mögliche Werte sind hier:

- `client`Wird für`switchType`der Wert`client`verwendet, so werden alle Daten für die einzelnen Tab-Panels beim Erstellen der Seite mit in die HTML-Ausgabe geschrieben. Nachdem die Seite auf dem Client geladen wurde, ist der Wechsel zwischen den einzelnen Tab-Panels zwar schneller, das HTML kann aber in Abhängigkeit der darzustellenden Inhalte sehr groß werden. Der Wert`client`ist der Defaultwert.
- `reloadTab`Sollen die Inhalte erst geladen werden, wenn das Tab-Panel tatsächlich angezeigt wird, so kann der Wert`reloadTab`verwendet werden. Wird das Tab-Panel ausgewählt, wird der Inhalt vom Server angefordert und das Tab-Panel wird per Ajax aktualisiert.
- `reloadPage`Auch bei dem Wert`reloadPage`wird der Inhalt der einzelnen Tab-Panels erst geladen, wenn das Tab-Panel angezeigt werden soll. Jedoch wird im Unterschied zum`reloadTab`die gesamte Seite neu geladen.

15.3.4 Tobago tc:menu

Ein gewisser Anteil an Funktionalität wird bei Desktop-Applikationen oftmals über Menüs angeboten. Über das Tobago-Control`tc:menu`kann ein Menü auf einer Seite dargestellt werden. Listing**Das tc:menu-Tobago-Control**zeigt einen Ausschnitt der Deklaration des Controls.

```
<tc:page>
...
  <f:facet name="menuBar">
    <tc:menuBar>
      <tc:menu label="Sections">
        <tc:menuItem label="Customers"
            action="customer-list"
            immediate="true"
            image="resources/images/16/customer.png"/>
        <tc:menuItem label="Providers"
            action="provider-list"
            immediate="true"
            image="resources/images/16/provider.png"/>
      </tc:menu>
      ...
    </tc:menuBar>
  </f:facet>
  ...
<tc:page>
```

Ein Menü wird über das Facet`menuBar`einer`tc:page`zu einer Seite hinzugefügt. Menüeinträge können durch die Tags`tc:menuItem`,`tc:menuCheckBox`und`tc:menuRadio`erstellt werden. Ein Separator zwischen einzelnen Menüeinträgen wird durch ein`tc:separator`eingefügt.

15.3.5 Tobago tc:toolBar

Das`tc:toolBar`-Control wird verwendet, um eine Gruppe von Buttons zu rendern. Wie in Abbildung**Das tc:toolBar-Tobago-Control**dargestellt, gibt es die Möglichkeit, die einzelnen Buttons mit einer Beschriftung beziehungsweise einem Bild zu versehen.

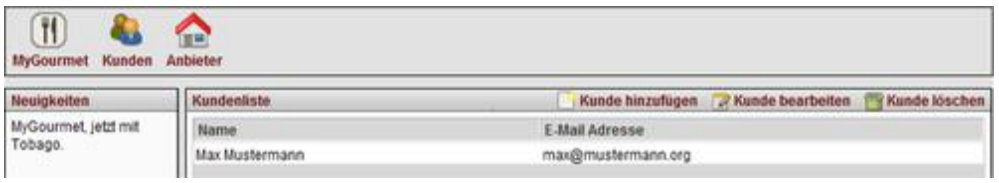


Abbildung:Das tc:toolBar-Tobago-Control

Ein Bild kann in verschiedenen Standardgrößen verwendet werden. Die Größe des darzustellenden Bildes wird durch das Attribut`iconsize`bestimmt. Gültige Werte für dieses Attribut sind`big`,`small`und`off`. Einzelne Buttons werden durch ein`tc:toolBarCommand`erstellt. Soll ein Bild beispielsweise ausgegraut dargestellt werden, so besteht diese Möglichkeit über das Attribut`disabled`. Wird die in diesem Attribut angegebene Bedingung erfüllt, so versucht der Ressourcenmanager ein Bild nach folgendem Namensschema zu laden:

- Normales Bild -`images/edit.png`
- Ausgegrautes Bild -`images/editDisabled.png`

Listing**Deklaration des tc:toolBar-Tobago-Controls**zeigt die Deklaration eines`tc:toolBar`-Controls.

```
<tc:toolBar>
  <tc:toolBarCommand
    label="Add Customer"
    action="customer"
    image="resources/images/new.png"/>
  <tc:toolBarCommand
    label="Edit Customer"
    action="#{customerController.editCustomer}"
    image="resources/images/edit.png"
    disabled="#{customerController.customerAvailable}"/>
</tc:toolBar>
```

15.3.6 Tobago tc:sheet

Tabellen werden in*Tobago*mit dem Tag`tc:sheet`erstellt. Im*Tobago-MyGourmet*-Beispiel wird eine Tabelle verwendet, um die Adressen von Kunden anzuzeigen. Listing**Das tc:sheet-Tobago-Control**zeigt einen Ausschnitt der Deklaration einer Tabelle.

```
<tc:sheet var="address"
    value="#{customerController.addressList}"
    state="#{customerController.selectedAddress}">
  <tc:column label="Zip Code">
    <tc:out value="#{address.zipCode}"/>
  </tc:column>
  <tc:column label="City">
    <tc:out value="#{address.city}"/>
  </tc:column>
  ...
</tc:sheet>
```

Über das`var`-Attribut wird zur Laufzeit auf die Objekte der Liste zugegriffen, die im`value`-Attribut angegeben wurde. Diese Liste enthält die Daten, die in dem`tc:sheet`angezeigt werden sollen. Der Zugriff auf Instanzvariablen bzw. Methoden erfolgt durch einfache Punktnotation. Spalten werden innerhalb vom`tc:sheet`über das Tag`tc:column`hinzugefügt. In Listing**Das tc:sheet-Tobago-Control**wird in jeder einzelnen Spalte ein`tc:out`verwendet, um die einzelnen Adresswerte anzuzeigen. Es können jedoch auch die Tags`tc:in`,`tc:selectOneChoice`oder`tc:selectBooleanCheckbox`in einer Spalte verwendet werden. Über die verschiedenen mit`show`beginnenden Attribute wird das Navigationsverhalten einer Tabelle gesteuert. So kann

über das Attribut`showDirectLinks`direkt zu einer gewünschten Seite in der Tabelle gesprungen werden. Das`state`-Attribut wird verwendet, um aktuelle Informationen über die Tabelle zu erhalten. Die durch`method-binding`angegebene Methode muss als Übergabeparameter ein`SheetState`empfangen können. Über dieses Objekt kann nun beispielsweise die selektierte Zeile ermittelt werden. So hat ein Entwickler die Möglichkeit, auf Ereignisse des Benutzers zu reagieren. In*MyGourmet*kann der Benutzer eine ausgewählte Adresse über die Toolbar durch einen Klick auf`Delete Address`löschen. Die in der Geschäftslogik auszuführende Methode`deleteAddress`wird in Listing**Selektion einer Zeile im tc:sheet-Tobago-Control**dargestellt.

```
public String deleteAddress() {
  Address address = getSingleSelectedAddress();
  if (address != null) {
    customerService.deleteAddress(address);
  }
  return CUSTOMER;
}

private Address getSingleSelectedAddress() {
  List<Integer> selection = selectedAddress.getSelectedRows();
  if (selection.size() != 1) {
    createFacesMessage(FacesMessage.SEVERITY_ERROR,
        "Please select exactly one row.");
    return null;
  }
  return addressList.get(selection.get(0));
}
```

Über das Attribut`sortable`der`tc:column`wird das Sortieren einzelner Spalten aktiviert. Wird für die Sortierung ein eigener Sortieralgorithmus benötigt, so kann dieser über das Attribut`sortActionListener`im`tc:sheet`-Tag ausgeführt werden. Informationen über die derzeitig angewandte Sortierung finden sich auch im`SheetState`wieder.

15.4 Partial-Rendering mit Tobago

Performance ist bei Webapplikationen schon immer ein wichtiges Thema gewesen. Um nicht bei jedem gestellten Request eine Seite komplett neu zu rendern, gibt es die Möglichkeit, nur einzelne Elemente neu zu laden.*Tobago*unterstützt ebenfalls dieses Partial-Rendering. Im Gegensatz zum bereits erwähnten

16 Eine kurze Einführung in Maven

*Apache Maven*ist ein äußerst hilfreiches Werkzeug, um Java-basierte Projekte zu verwalten. Es kann unter anderem Applikationen verwalten, erstellen, verteilen, automatisch testen, Abhängigkeiten verwalten und Webseiten des Projekts erstellen. Neben einer standardisierten Beschreibung von Projekten im*Project Object Model*(`pom.xml`) und einem standardisierten Build-Prozess bietet*Maven*unter anderem noch eine automatische Auflösung von Abhängigkeiten zu anderen Projekten und Bibliotheken. Die Funktionalität lässt sich durch verschiedenste Plug-ins erweitern, von denen wir in weiterer Folge einige zeigen.
Zunächst sollte man wissen, dass*Maven*sämtliche für den Build-Prozess benötigte Dateien, seien es normale Bibliotheken oder Plug-ins, automatisch von zentralen Stellen aus dem Internet - den sogenannten*Maven-Repositories*- lädtDas zentrale*Maven-Repository*finden Sie zum Beispiel unter der Adresse**http://search.maven.org/**. Dort können Sie auch sehr bequem nach*Maven*-Artefakten suchen.:. In einem solchen*Repository*sind die Bibliotheken von vielen Open-Source-Projekten in einer speziellen*Maven*-Struktur (unterteilt nach Group-ID, Artifact-ID und Version) abgelegt.
Ein großer Pluspunkt von*Maven*ist die automatische Verwaltung von Abhängigkeiten auf Bibliotheken. Die Abhängigkeiten eines Projekts werden in seiner`pom.xml`in Form von`dependency`-Elementen definiert. Listing**Maven-Abhängigkeit für MyFaces 2.2.1**zeigt die Abhängigkeiten für*MyFaces*in Version 2.2.1.

```
<dependencies>
  <dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-api</artifactId>
    <version>2.2.1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-impl</artifactId>
    <version>2.2.1</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

Wie Sie sehen, werden Abhängigkeiten auf Bibliotheken über ihre Group-ID, Artifact-ID und Version definiert. Im*Repository*spiegelt sich diese Struktur in der Verzeichnishierarchie wider.
Die benötigten Bibliotheken werden nach dem Ladevorgang in einem lokalen Repository abgelegt und können bei Bedarf von dort bezogen werden. Dieses lokale Verzeichnis befindet sich standardmäßig im Homeverzeichnis des Benutzers, das je nach Betriebssystem variieren kann. Unter Windows liegt das lokale Repository beispielsweise in folgendem Verzeichnis:

```
C:\Benutzer\Benutzer\Benutzername.m2\Benutzername.m2\repository
repository
```

16.1 Installation von

Maven

In diesem Abschnitt beschreiben wir Schritt für Schritt, wie*Maven*auf einem Windows-System eingerichtet wird:

1. Laden von Maven
   Eine komprimierte Datei der aktuellen Version von*Maven*ist auf der Projektwebseite**http://maven.apache.org**zu finden. Die aktuelle Version ist 3.0.5, mit der auch alle Beispiele getestet wurden.
2. Entpacken des Archivs
   Zuerst müssen Sie die heruntergeladene Zip-Datei`apache-maven-3.x.x-bin.zip`in ein Verzeichnis Ihrer Wahl entpacken. Im weiteren Verlauf nehmen wir folgendes Verzeichnis an, das abhängig von der konkreten Versionsnummer auf Ihrem Rechner anders aussehen kann:

   ```
   C:\Programme\Programme\maven-3.x.x
   ```

3. Setzen der benötigten Umgebungsvariablen
   Für den einwandfreien Betrieb von Maven müssen einige Umgebungsvariablen gesetzt werden. Zuerst setzen wir die Variable`MAVEN_HOME`auf das Installationsverzeichnis von Maven. Öffnen Sie dazu

   ```
   Systemeinstellungen (Windows-Taste + Pause)
       -> Erweitert -> Umgebungsvariablen
   ```

   und fügen Sie unter den Benutzervariablen eine Variable mit dem Namen`MAVEN_HOME`und folgendem Wert hinzu:

   ```
   C:\Programme\Programme\maven-3.x.x
   ```

   Des Weiteren muss das`bin`-Verzeichnis der Maven-Installation in den Systempfad aufgenommen werden. Setzen Sie dazu die Benutzervariable`PATH`auf folgenden Wert:

   ```
   MAVEN_HOME\bin;PATH
   ```

   Im selben Dialog muss auch`JAVA_HOME`zum Pfad des Java-JDK gesetzt sein, also zum Beispiel auf:

   ```
   C:\Programme\Programme\Java\Java\jdk1.7.0_21
   ```

4. Maven überprüfenIn der Konsole kann nun überprüft werden, ob*Maven*korrekt aufgesetzt wurde:

   ```
   mvn --version
   ```

   Wird eine Versionsnummer ausgegeben, kann der nächste Schritt erfolgen, falls nicht, müssen die vorherigen Punkte noch einmal auf Korrektheit überprüft werden.

Ihr System verfügt jetzt über eine lauffähige Version von*Maven*. Als Einstiegspunkte für vertiefende Informationen zum Thema*Maven*bieten sich dessen Projektseite**http://maven.apache.org**und die frei verfügbaren Bücher*Maven: The Complete Reference*und*Maven by Example**Maven by Example*und*Maven: The Complete Reference*finden Sie unter**http://www.sonatype.com/resources/books**.:an.

16.2 Maven und MyGourmet

Alle*MyGourmet*-Beispiele sind als*Maven*-Projekte angelegt und dadurch sehr einfach zu handhaben. Wir werden in diesem Abschnitt kurz die Möglichkeiten von*Maven*anhand von*MyGourmet*präsentieren. Der Quellcode aller*MyGourmet*-Beispiele ist unter der URL**http://jsfatwork.irian.at**verfügbar.
Beginnen wir mit der grundlegendsten Aufgabe: Der Build-Prozess des Projekts wird mit folgendem Befehl im Wurzelverzeichnis des Projekts angestoßen:

```
mvn clean install
```

*Maven*löscht daraufhin alle bisher generierten Dateien, kompiliert alle Klassen und packt sie zusammen mit allen Ressourcen und Bibliotheken, die über Abhängigkeiten in der Datei`pom.xml`definiert sind, in ein WAR-Archiv. Diese Dateien landen im ersten Schritt im Verzeichnis`target`. Das WAR-Archiv wird dann im lokalen Repository unter der Group-ID, der Artifact-ID und der Version, die in der`pom.xml`angegebenen sind, abgelegt. Für das Beispiel*MyGourmet 1*ergibt sich daraus der in Abbildung**MyGourmet 1 im lokalen Repository**ersichtliche Verzeichnisbaum.

Abbildung:MyGourmet 1 im lokalen Repository

Standardmäßig benutzen alle*MyGourmet*-Beispiele*Mojarra*als JSF-Implementierung. Wir haben allerdings die Projektbeschreibung in der Datei`pom.xml`so gestaltet, dass Sie ohne Änderungen an der Datei selbst auf*MyFaces*umschalten können. Dazu gibt es zwei Profile mit den jeweils für die Implementierung benötigten Abhängigkeiten. Wenn Sie keines der Profile explizit auswählen, kommt immer*Mojarra*zum Einsatz. Wollen Sie*MyFaces*einsetzen, müssen Sie das Profil mit der ID`myfaces`wie folgt beim Aufruf von*Maven*aktivieren:

```
mvn clean install -P myfaces
```

Falls im Browser Probleme auftreten, wenn Sie dasselbe Projekt direkt hintereinander mit*MyFaces*und*Mojarra*ausführen (oder umgekehrt), sollten Sie den Cache des Browsers löschen. Manchmal führen in diesem Fall gecachte JavaScript-Dateien zu Problemen.
Alle*MyGourmet*-Beispiele sind über ihre`pom.xml`so konfiguriert, dass sie mit folgendem Befehl im Wurzelverzeichnis des Projekts ausgeführt werden können:

```
mvn jetty:run
```

Das*Jetty-Maven-Plug-in*startet daraufhin den Servlet-Container*Jetty*und rollt die Anwendung aus.*MyGourmet 1*kann dann zum Beispiel unter folgender Adresse im Browser aufgerufen werden:

```
http://localhost:8080/mygourmet01
```

Sie müssen das Projekt vorher nicht kompilieren oder im lokalen Repository installieren.*Maven*kümmert sich

automatisch darum, falls das noch nicht geschehen ist.
Auch hier gilt, dass Sie*Apache MyFaces*über das Profil mit der ID`myfaces`aktivieren können:

```
mvn jetty:run -P myfaces
```

Alle gängigen Entwicklungsumgebungen unterstützen mittlerweile den direkten Import von Maven-Projekten. Wie das im Detail mit Eclipse funktioniert, können Sie in Abschnitt**Sektion: Eclipse und MyGourmet**nachlesen.

16.3 Erstellen eines JSF-Projekts

Mit*Maven*ist es sehr einfach, eine*HelloWorld*-Anwendung zu generieren. Für diesen Zweck gibt es in*Maven*das Konzept der so genannten*Archetypes*. Das sind Vorlagen, auf deren Basis sich aus vorhandenen Bibliotheken Maven-Projekte mit der grundlegenden Struktur einer Applikation generieren lassen.
Der Vorlagenkatalog von*MyFaces*enthält zurzeit folgende acht Archetypes zum Generieren von JSF-Projekten:

1. MyFaces-Hello-World
   Vorlage für das Grundgerüst eines Projekts mit*MyFaces*in Version 1.2 inklusive aller Abhängigkeiten und JSP als Seitendeklarations-sprache.
2. MyFaces-Hello-World mit Facelets
   Vorlage für das Grundgerüst eines Projekts mit*MyFaces*in Version 1.2 inklusive aller Abhängigkeiten und Facelets als Seitendeklarationssprache.
3. MyFaces-Hello-World mit Portlets
   Vorlage für das Grundgerüst eines Portlets-Projekts mit*MyFaces*in Version 1.2 inklusive aller Abhängigkeiten und JSP als Seitendeklarationssprache.
4. MyFaces-Hello-World 2.0
   Vorlage für das Grundgerüst eines Projekts mit*MyFaces*in Version 2.1 inklusive aller Abhängigkeiten und Facelets als Seitendeklarationssprache.
5. MyFaces-Hello-World 2.0 mit OWB
   Vorlage für das Grundgerüst eines Projekts mit*MyFaces*in Version 2.1 und*Apache OpenWebBeans**Apache OpenWebBeans*ist eine Implementierung des Standards*Contexts and Dependency Injection for Java*(CDI, JSR-299).:inklusive aller Abhängigkeiten und Facelets als Seitendeklarationssprache.
6. JSF-Komponente
   Eine spezielle Vorlage, aus der Maven-Projekte für Komponentenbibliotheken mit*MyFaces*in Version 1.2 generiert werden können.
7. MyFaces-Hello-World mit Trinidad
   Vorlage für das Grundgerüst eines Maven-Projekts mit*MyFaces*in Version 1.2 und*Trinidad*in Version 1.2 inklusive aller Abhängigkeiten und JSP als Seitendeklarationssprache.
8. MyFaces-Hello-World mit Trinidad 2.0
   Vorlage für das Grundgerüst eines Maven-Projekts mit*MyFaces*in Version 2.1 und*Trinidad*in Version 2.0 inklusive aller Abhängigkeiten und Facelets als Seitendeklarationssprache.

Erstellen wir also jetzt ein Maven-Projekt für eine einfache Webanwendung in JSF. Der Vorgang ist denkbar einfach: Rufen Sie dazu in der Konsole in einem beliebigen Verzeichnis folgenden Befehl auf:

```
mvn archetype:generate
```

```
-DarchetypeCatalog=http://myfaces.apache.org
```

Dieser Befehl startet die Projektgenerierung mit dem Vorlagenkatalog von*Apache MyFaces*. Das Erstellen des Projekts besteht aus folgenden drei Schritten:

1. Maven stellt alle Vorlagen aus dem oben angegebenen Katalog`http://myfaces.apache.org`in einer nummerierten Liste dar. Wählen Sie die Nummer der Vorlage mit dem Namen`myfaces-archetype-helloworld20`aus.
2. Nach der Auswahl müssen Sie die Anwendung konfigurieren. Geben Sie als Group-ID`at.irian.jsfatwork`, als Artifact-ID`helloworld`, als Version`1.0-SNAPSHOT`und als Package des Projekts`at.irian.jsfatwork.helloworld`ein.
3. Nach dem Bestätigen der Einstellungen legt Maven die Projektstruktur unter dem Verzeichnis`/helloworld`an.

Die generierte Applikation baut auf*MyFaces*in Version 2.1 mit allen benötigten Abhängigkeiten auf - diese wurden bereits in die*Maven*-Projektdatei`pom.xml`eingetragen. Genauso befinden sich unter

```
\helloworld\helloworld\src\src\main\main\webapp\webapp\WEB-INF
```

die für die Anwendung benötigten Konfigurationsdateien`web.xml`und`faces-config.xml`mit den erforderlichen Parametern. Je nach Bedarf sind diese wie in Abschnitt**Sektion: Konfiguration von JavaServer Faces**beschrieben abzuändern.

14 MyGourmet Fullstack Spring - - JSF, Spring, Orchestra und JPA kombiniert

In diesem Kapitel werden wir anhand unseres*MyGourmet*-Beispiels die Architektur einer JSF-Anwendung mit*Spring*vorstellen, die sich in dieser oder ähnlicher Form in der Praxis bewährt hat. Sie stellt eine optimale Ausgangsbasis für Ihre eigenen JSF-Webapplikationen dar - auch für komplexere Anwendungen. Das Beispiel*MyGourmet Fullstack Spring*ist eine Erweiterung des Beispiels*MyGourmet 16 Spring*, in dem zum einen die Funktionalität der Anwendung ausgebaut wird und zum anderen die Architektur der Anwendung auf den derzeitigen Stand der Technologie gebracht wird.
Als JSF-Implementierung kommt standardmäßig*Apache MyFaces*zum Einsatz. Das Beispiel funktioniert aber auch mit der aktuellen Version von*Mojarra*. Wie in allen anderen Beispielen können Sie auch hier die JSF-Implementierung über ein Profil in der Datei`pom.xml`bestimmen (siehe Anhang**Kapitel: Eine kurze Einführung in Maven**für Details). In*Spring*wird als Persistenzschicht die*Java Persistence API*(JPA) in Version 2.0 mit dem*Hibernate EntityManager*als Implementierung eingesetzt. Datenbankseitig steht*HyperSQL Data-Base*(HSQLDB) bereit - es kann aber jede Datenbank, die von*Hibernate*unterstützt wird, verwendet werden.
Den Quellcode zu*MyGourmet Fullstack*finden Sie wie den Code aller bisherigen Beispiele unter**http://jsfatwork.irian.at**.

14.1 Architektur von MyGourmet Fullstack

In*MyGourmet Fullstack*kommt eine bei der Java-Webentwicklung weitverbreitete Architektur mit drei übereinanderliegenden Schichten zum Einsatz. Die*Präsentationsschicht*ist die oberste Schicht und beinhaltet die Benutzerschnittstelle. Sie greift auf die*Serviceschicht*zu, in der die Geschäftslogik der Anwendung definiert ist. Ganz unten liegt die*Datenzugriffsschicht*, in der sämtliche in der Serviceschicht getätigten Zugriffe auf die Daten der Anwendung gekapselt sind. Die Entitäten des hinter der Anwendung liegenden Modells sind schichtübergreifend verfügbar. Abbildung**Architektur von MyGourmet Fullstack Spring**zeigt eine grafische Darstellung der Architektur von*MyGourmet Fullstack Spring*.

Abbildung:Architektur von MyGourmet Fullstack Spring

Ein Vorteil der Schichtenarchitektur ist die strikte Trennung der Zuständigkeiten (Separation of Concerns, SOC). Jede Schicht ist dabei für einen bestimmten Bereich der Anwendung zuständig. Eine Schicht kann auf die Funktionalität der direkt unter ihr liegenden Schicht zugreifen und kann Funktionalität für die direkt über ihr liegende Schicht anbieten. Nachdem es nie Abhängigkeiten nach oben geben darf, bildet jede Schicht mit den unter ihr liegenden Schichten ein abgeschlossenes Subsystem.
Wir empfehlen Ihnen, immer eine saubere Schichtentrennung einzuhalten - auch wenn das im ersten Moment wie ein unnötiger Mehraufwand aussieht. Die Trennung (natürlich nur an vernünftigen Stellen) zahlt sich spätestens in der Wartung aus.
In den nächsten Abschnitten werden wir einen kurzen Blick auf alle Schichten von*MyGourmet Fullstack Spring*inklusive der schichtübergreifenden Entitäten werfen. Die Struktur des Maven-Projekts ist etwas einfacher gehalten und entspricht nicht direkt den Schichten der Anwendung. Aus Gründen der Einfachheit haben wir die Entitäten, die Datenzugriffsschicht und die Serviceschicht im Modul`mygourmet-service-spring`zusammengefasst. Die Präsentationsschicht befindet sich im Modul`mygourmet-webapp-spring`. Für umfangreichere Projekte macht es aber durchaus Sinn, jede Schicht in einem eigenen Maven-Modul unterzubringen.

14.1.1 Entitäten

Entitäten bilden das Modell der Geschäftsobjekte und sind ein zentraler Bestandteil jeder Applikation. Entitäten sind einfache JavaBeans mit Eigenschaften und deren*Getter*-und*Setter*-Methoden. Sie stellen die objektorientierte Repräsentation der Tabellen in der Datenbank dar. Da sie in allen Schichten verwendet werden, sind sie aus Sicht der Architektur außerhalb der Schichten angesiedelt. Die Entitätsklassen finden Sie im Package`at.irian.jsfatwork.domain`im Modul`mygourmet-service-spring`.
Die Information über die Zuordnung zwischen den Tabellen und Spalten der Datenbank und den Entitäten und Eigenschaften ist direkt in den Klassen enthalten. JPA bietet dazu eine Reihe von Annotationen an. Dadurch ist es nicht mehr nötig, Unmengen von XML-Dateien zur Konfiguration dieser Zuordnung anzulegen.
Die benutzerdefinierten Bean-Validation-Validatoren müssen sich ebenfalls im Modul`mygourmet-service-spring`befinden, da sie in den Entitäten benutzt werden. Nachdem Bean-Validation aber nicht von JSF abhängig ist, stellt das kein Problem dar. Die Validatorklassen und Annotationen finden Sie in`at.irian.jsfatwork.validation`.

14.1.2 Datenzugriffsschicht

In der Datenzugriffsschicht wird die Schnittstelle zur Datenbank mit einem generischen Crud-Service realisiert. Dieser Service ist in der Klasse`CrudService`als*Spring*-Bean im Singleton-Scope umgesetzt und nutzt die JPA-Unterstützung von*Spring*. Der Crud-Service stellt generische Methoden für die wichtigsten Funktionen des Entity-Managers von JPA bereit. Für*MyGourmet*reicht diese simple Umsetzung aus - in komplexeren Applikationen kann ein solcher Crud-Service dann zum Beispiel als Grundlage für Repositories dienen. Listing**MyGourmet Fullstack Spring: CrudService**zeigt Teile der Klasse`CrudService`.

```
@Named("crudService")
```

```
@Singleton
public class CrudService {

  @PersistenceContext(unitName = "mygourmet")
  private EntityManager em;

  public <T extends BaseEntity> void persist(T entity) {
    em.persist(entity);
  }
  public <T extends BaseEntity> T findById(
      Class<T> clazz, long id) {
    return em.find(clazz, id);
  }
  public <T extends BaseEntity> void delete(T entity) {
    em.remove(entity);
  }
}
```

Die zentrale Stelle zur Interaktion mit dem Persistenzkontext von JPA ist die Klasse`EntityManager`. In*MyGourmet*wird der Entity-Manager über*Spring*konfiguriert und von*Orchestra*verwaltet.*Spring*wertet dazu die Annotation`@PersistenceContext`aus und setzt den Entity-Manager beim Erzeugen der Bean`CrudService`in die annotierte Eigenschaft.
*Orchestra*stellt dabei sicher, dass*Spring*den Persistenzkontext der aktuellen Konversation verwendet. Wenn Sie Listing**MyGourmet Fullstack Spring: CrudService**nochmals genauer betrachten, werden Sie feststellen, dass die Bean im Singleton-Scope definiert ist. Der Persistenzkontext ist allerdings an eine kurzlebigere Konversation gebunden. Wie kann das funktionieren? Der von*Spring*injizierte Entity-Manager ist ein Proxy, der den Zugriff auf den aktuellen Persistenzkontext regelt.
Durch die Persistenzunterstützung von*Orchestra*läuft der Crud-Service im Kontext der aktuellen Konversation ab. Sie bekommen den selben Entity-Manager injiziert, der zu Beginn der Konversation erstellt und bis zu ihrem Ende offen gehalten wird. Dadurch ist es mit*Orchestra*auch ohne Probleme möglich, innerhalb einer Konversation direkt mit den Entitäten zu arbeiten. Exceptions, die durch einen zu kurzlebigen Persistenzkontext hervorgerufen werden, gehören der Vergangenheit an.
Auf die Konfiguration von JPA werden wir erst in Abschnitt**[Sektion: Konfiguration von JPA]**näher eingehen.

14.1.3 Serviceschicht

Die Geschäftslogik der Anwendung ist in der Serviceschicht untergebracht und von der Benutzerschnittstelle entkoppelt. Die Serviceschicht bekommt den Crud-Service injiziert und ruft ihn bei Bedarf, wie zum Beispiel bei einer Persistierung, auf. Der Weg zur Datenbank sollte in der Präsentationsschicht ausschließlich über die Serviceschicht und nicht direkt über die Datenzugriffsschicht erfolgen.
Im Package`eat.irian.jsfatwork.service`gibt es für jede Service-Bean ein Interface. Die zugehörige Implementierung befindet sich im Unterpackage`impl`. Somit ist sichergestellt, dass gegen das Interface programmiert werden kann und beim Zugriff auf die Services konkrete Implementierungsdetails vor der Anwendung verborgen bleiben. Die Implementierungsklassen sind über die JSR-330-Annotation`@Named`als*Spring*-Beans im Singleton-Scope definiert. Listing**MyGourmet Fullstack Spring: Serviceimplementierung**zeigt exemplarisch die Klasse`ProviderServiceImpl`.

```
@Named("providerService")
@Singleton
public class ProviderServiceImpl implements ProviderService {
  @Inject
  private CrudService crudService;
```

```
  public Provider createNew() {
    return crudService.createNew(Provider.class);
  }
  @Transactional
  public void save(Provider entity) {
    crudService.persist(entity);
  }
  @Transactional
  public void delete(Provider provider) {
    for (Order order : provider.getOrders()) {
      order.setCustomer(null);
      crudService.delete(order);
    }
    provider.getCategories().clear();
    crudService.delete(provider);
  }
  @Transactional(readOnly = true)
  public Provider findById(long id) {
    return crudService.findById(Provider.class, id);
  }
  @Transactional(readOnly = true)
  public List<Provider> findAll() {
    return crudService.findAll(Provider.class);
  }
}
```

Die Serviceschicht ist in*MyGourmet Fullstack Spring*auch für die Transaktionskontrolle der Applikation zuständig. Die Transaktionen sind an einzelnen Servicemethoden ausgerichtet und werden von*Spring*verwaltet. Jede Methode einer Serviceklasse, die mit*@Transactional*annotiert ist, wird in einer Transaktion ausgeführt. Die Serviceschicht ist der ideale Ort zur Definition der Transaktionen, da sie für die Benutzerschnittstelle das Tor zu Geschäftslogik darstellt. Wenn Sie nochmals einen Blick auf Listing**MyGourmet Fullstack Spring: CrudService**werfen, werden Sie bemerken, dass es dort keine`@Transactional`-Annotationen gibt. Nachdem Crud-Operationen nur im Service verwendet werden, laufen sie automatisch in den dort definierten Transaktionen ab.
Die Schichtentrennung ist beispielsweise auch für das Testen außerordentlich wichtig. Die Applikation kann dann nämlich (ohne Ausführung der GUI selbst) über die Serviceschicht direkt getestet werden. Wir empfehlen Ihnen, die Serviceschicht von Beginn an zu testen und die Tests immer auf gleichem Stand wie die GUI-Logik zu halten. Gerade bei der Entwicklung von Webanwendungen kostet jeder zu einem Neustart des Servers führende Fehler, der mit der Ausführung eines Tests verhindert hätte werden können, wertvolle Entwicklungszeit.

14.1.4 Präsentationsschicht

Die*Präsentationsschicht*umfasst die Benutzerschnittstelle der Applikation und bildet die oberste Schicht der Architektur. In*MyGourmet Fullstack Spring*besteht die Präsentationsschicht aus der JSF-Webanwendung und ist im*Maven*-Modul`mygourmet-webapp-spring`untergebracht. Dieses Modul enthält alle Seitendeklarationen und ihre View-Controller sowie alle Komponenten, Konverter, Validatoren und Phase-Listener.
Die Präsentationsschicht ist nur für die Benutzerschnittstelle zuständig und greift zur Ausführung von Geschäftslogik auf die Serviceschicht zu. Umgekehrt darf es aber keine Abhängigkeiten der Serviceschicht oder gar der Datenzugriffsschicht auf die Präsentationsschicht geben. GUI/JSF-Spezifika haben dort natürlich nichts verloren. Aufmerksame Leser werden jetzt anmerken, dass wir bereits in Abschnitt**[Sektion: Datenzugriffsschicht]**, über die Datenzugriffsschicht, die Rolle von*Orchestra*bei der Verwaltung des Persistenzkontexts besprochen haben. Haben wir hier bereits das Prinzip der Schichtentrennung verletzt? Die Antwort lautet nein, denn*Orchestra*verwaltet den Persistenzkontext aus der

Präsentationsschicht heraus.
Sehen wir uns diesen Vorgang anhand der Kunden-Übersichtsseite und ihrem View-Controller etwas genauer an. Listing**MyGourmet Fullstack Spring: View-Controller der Kunden-Übersichtsseite**zeigt die Klasse`CustomerListBean`des View-Controllers. Beim initialen Zugriff auf die Ansicht wird die Bean`customerListBean`inklusive der gleichnamigen Konversation mit dem Persistenzkontext erstellt.*Orchestra*stellt ab diesem Zeitpunkt sicher, dass*Spring*den mit der Konversation verbundenen Persistenzkontext verwendet. Beim Laden der Kundenliste in der Methode`preRenderView`kommt beim Aufruf von`customerService.findAll()`im dahinterliegenden Crud-Service bereits der Persistenzkontext der Konversation zum Einsatz.

```
@Named("customerListBean")
@Scope("access")
@ViewController(viewIds = {"/customerList.xhtml"})
public class CustomerListBean {
  @Inject
  private CustomerService customerService;
  private List<Customer> customerList;

  @PreRenderView
  public void preRenderView() {
    customerList = customerService.findAll();
  }
  public List<Customer> getCustomerList() {
    return customerList;
  }
  public String deleteCustomer(Customer customer) {
    customerService.delete(customer);
    return null;
  }
}
```

Dasselbe gilt für das Löschen eines Kunden in`deleteCustomer`. Beim Aufruf von`customerService.delete(customer)`wird im dahinterliegenden Crud-Service ebenfalls der Persistenzkontext der Konversation verwendet. Der zu löschende Kunde wird beim Aufruf der Action-Methode direkt über die Method-Expression als eine der zuvor geladenen Entitäten übergeben. Durch den an die Konversation gebundenen Persistenzkontext ist auch das ohne Probleme möglich. Nachdem die Methode`delete`im Service mit`@Transactional`annotiert ist, läuft die Operation sogar in einer Transaktion ab.
Die automatische Verwaltung des Persistenzkontexts durch*Orchestra*funktioniert natürlich nur für Managed-Beans im Access- oder Manual-Scope.

14.2 Konfiguration der Anwendung

Die Konfiguration von*MyGourmet Fullstack Spring*entspricht weitestgehend der bereits bekannten Konfiguration aus den vorherigen Beispielen. Neu sind lediglich die Einstellungen für JPA und die Persistenzunterstützung von*Orchestra*. Wir werden daher in diesem Abschnitt nicht mehr auf die Basiskonfiguration eingehen.
Im ersten Schritt haben wir auch die Spring-Konfiguration an die neue Struktur des Maven-Projekts angepasst. Es gibt jetzt eine Konfigurationsdatei für jedes Maven-Modul:

- `service.spring.xml`im Modul`mygourmet-service-spring`

- `web.spring.xml`im Modul`mygourmet-webapp-spring`

Listing**MyGourmet Fullstack Spring: Konfiguration von Spring in der web.xml**zeigt den Parameter`contextConfigLocation`mit den Namen der Konfigurationsdateien in der`web.xml`. Die Konfigurationsdatei`service.spring.xml`kommt dabei aus dem Classpath, da das Service-Modul als Maven-Abhängigkeit im Webapp-Modul definiert ist.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/web.spring.xml
    classpath:service.spring.xml
  </param-value>
</context-param>
```

In Abschnitt**[Sektion: Konfiguration von JPA]**werden wir kurz auf die Details der Konfiguration von JPA in*Spring*in der Datei`service.spring.xml`eingehen. Anschließend folgen in Abschnitt**[Sektion: Konfiguration von*Orchestra*]**weiterführende Informationen zur Konfiguration der Persistenzunterstützung von*Orchestra*in`web.spring.xml`.

14.2.1 Konfiguration von JPA

In diesem Abschnitt werden wir Ihnen einige Details der Konfigurationsdatei`service.spring.xml`mit den Einstellungen für JPA unter*Spring*näherbringen. Wir haben den*Hibernate EntityManager*als Implementierung von JPA 2.0 gewählt. Als Datenbank kommt*HyperSQL DataBase*(HSQLDB) zum Einsatz. Zuerst muss eine Data-Source für die Datenbank definiert werden. Dazu kommt am besten ein Connection-Pool, in unserem Fall*C3P0*, zum Einsatz. Listing**MyGourmet Fullstack Spring: Konfiguration der Entity-Manager-Factory**zeigt die Definition der Bean`dataSource`mit den Einstellungen für die Datenbank und den Connection-Pool.

```
<!-- Configure a c3p0 pooled data source -->
<bean id="dataSource" class="com.mchange.v2.c3p0
    .ComboPooledDataSource">
  <property name="user" value="sa"/>
  <property name="password" value=""/>
  <property name="driverClass" value="org.hsqldb.jdbcDriver"/>
  <property name="jdbcUrl" value="jdbc:hsqldb:mem:."/>
  <property name="initialPoolSize" value="1"/>
  <property name="minPoolSize" value="1"/>
  <property name="maxPoolSize" value="10"/>
</bean>
<!-- Hibernate JPA entity manager factory -->
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa
        .LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor
        .HibernateJpaVendorAdapter">
      <property name="showSql" value="false"/>
      <property name="database" value="HSQL"/>
```

```
      <property name="generateDdl" value="true"/>
    </bean>
  </property>
  <property name="persistenceUnitName" value="mygourmet"/>
</bean>
```

Der zentrale Punkt der Konfiguration von JPA in*Spring*ist die Definition der Entity-Manager-Factory. Diese Factory wird zum Beispiel zum Erstellen des Entity-Managers benutzt, der im Crud-Service-Bean verwendet wird. In unserem Beispiel wird die Factory in der Bean`entityManagerFactory`definiert (siehe Listing**MyGourmet Fullstack Spring: Konfiguration der Entity-Manager-Factory**). In der Eigenschaft`dataSource`setzen wir die zuvor definierte Data-Source und in der Eigenschaft`jpaVendorAdapter`folgen Details zur gewählten JPA-Implementierung. Für*Hibernate*kommt dazu eine Bean der Klasse`HibernateJpaVendorAdapter`zum Einsatz. Diese Bean bekommt in der Eigenschaft`database`den Typ der verwendeten Datenbank übergeben. Durch das Setzen der Eigenschaft`generateDdl`auf`true`weisen wir*Hibernate*an, die Datenbank aus den Mappings zu erstellen. Damit*Spring*die Annotation`@PersistenceContext`zum Injizieren des Entity-Managers im Crud-Service richtig verarbeitet, muss das Element`<context:annotation-config/>`zur Konfiguration hinzugefügt werden.
In*MyGourmet*wird die Transaktionskontrolle deklarativ mit der Annotation`@Transactional`realisiert.`@Transactional`kann auf Interfaces, Klassen und Methoden mit der Sichtbarkeit`public`angewandt werden.*Spring*empfiehlt allerdings, nur konkrete Klassen zu annotieren, um Probleme mit Proxies zu vermeiden. Die Annotationen selbst reichen allerdings nicht aus, um Transaktionen zu starten.`Spring`muss erst noch mit dem Tag`tx:annotation-drive`nangewiesen werden, die Transaktionskontrolle basierend auf den Annotationen durchzuführen. Der dazu benötigte Transaktionsmanager wird im Attribut`transaction-manager`des Tags angegeben. In unserem Fall handelt es sich dabei um einen speziellen Transaktionsmanager für JPA. Listing**MyGourmet Fullstack Spring: Konfiguration der Transaktionskontrolle**zeigt die Konfiguration der Transaktionskontrolle.

```
<tx:annotation-driven
    transaction-manager="transactionManager"/>

<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory"
      ref="entityManagerFactory"/>
</bean>
```

14.2.2 Konfiguration von Orchestra

Mit der im letzten Abschnitt gezeigten Konfiguration von JPA können wir die in Abschnitt**Sektion: Konfiguration von Orchestra**beschriebene Basiskonfiguration von*Orchestra*für die Persistenzunterstützung erweitern. Die komplette Konfiguration finden Sie in der Konfigurationsdatei`web.spring.xml`im Modul`mygourmet-webapp-spring`.
Damit die Bindung des Persistenzkontexts an die Konversation funktioniert, müssen die Conversation-Scopes um einen Interceptor erweitert werden. Dazu wird in der Bean-Definition über die Eigenschaft`advices`ein AOP-Advice hinzugefügt, der bei jedem Zugriff auf eine Bean in der Konversation prüft, ob der richtige Persistenzkontext gesetzt ist. Listing**MyGourmet Fullstack Spring: Konfiguration der Conversation-Scopes mit Persistenzunterstützung**zeigt die Konfiguration des Access- und des Manual-Scopes mit dem Interceptor.

```
<bean class="org.springframework.beans.factory.config
    .CustomScopeConfigurer">
  <property name="scopes">
    <map>
      <entry key="manual">
        <bean class="org.apache.myfaces.orchestra
            .conversation.spring.SpringConversationScope">
          <property name="timeout" value="30"/>
          <property name="lifetime" value="manual"/>
          <property name="advices">
            <list><ref bean="persistentContextInterceptor"/>
            </list>
          </property>
        </bean>
      </entry>
      <entry key="access">
        <bean class="org.apache.myfaces.orchestra
            .conversation.spring.SpringConversationScope">
          <property name="timeout" value="30"/>
          <property name="lifetime" value="access"/>
          <property name="advices">
            <list><ref bean="persistentContextInterceptor"/>
            </list>
          </property>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

Die Bean`persistentContextInterceptor`ist der Interceptor für die Scopes und stellt sicher, dass immer der richtige Persistenzkontext verwendet wird. Der Interceptor benötigt eine passende Factory für die eingesetzte Persistenztechnologie, die über die Eigenschaft`persistenceContextFactory`gesetzt wird. Für JPA liefert*Orchestra*bereits eine fertige Factory mit, die zum Erzeugen des Persistenzkontexts eine fertig konfigurierte JPA-Entity-Manager-Factory in der Eigenschaft`entityManagerFactory`erwartet. Mit der Konfiguration der Entity-Manager-Factory verlassen wir*Orchestra*und tauchen in die Welt von JPA ein. Wie das funktioniert, haben wir ja bereits im letzten Abschnitt gesehen. Listing**MyGourmet Fullstack Spring: Konfiguration der Persistenzunterstützung in Orchestra**zeigt die Konfiguration der Persistenzunterstützung in*Orchestra*.

```
<bean id="persistentContextInterceptor"
    class="org.apache.myfaces.orchestra.conversation
    .spring.PersistenceContextConversationInterceptor">
  <property name="persistenceContextFactory"
      ref="persistentContextFactory"/>
</bean>
<bean id="persistentContextFactory"
    class="org.apache.myfaces.orchestra.conversation
    .spring.JpaPersistenceContextFactory">
  <property name="entityManagerFactory"
      ref="entityManagerFactory"/>
</bean>
```

Damit ist*MyGourmet Fullstack Spring*fertig konfiguriert und einsatzbereit. Wir möchten Sie dazu einladen, den Quellcode der Anwendung genau unter die Lupe zu nehmen. Betrachten Sie das Beispiel als Basis für

eigene Experimente und erkunden Sie die Details der Zusammenarbeit von JSF, JPA,*Spring*und*Orchestra*in der Praxis.