HANSER

Bernd Müller

JBoss Seam

Die Web-Beans-Implementierung

ISBN-10: 3-446-41190-9
ISBN-13: 978-3-446-41190-6

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/978-3-446-41190-6
sowie im Buchhandel

Kapitel 2

Seam als modernes Komponenten-Framework

Die *Java Platform, Enterprise Edition* (Java-EE) ist in der aktuellen Version 5 ein modernes Komponenten-Framework, das eine ganze Reihe verschiedener Spezifikationen zu einem Ganzen bündelt. Es sind dies die Spezifikationen Servlets 2.5 (JSR 154), JavaServer Pages 2.1 (JSR 254), JavaServer Faces 1.2 (JSR 252), Enterprise JavaBeans 3.0 / Java Persistence API (JSR 220) und einige andere mehr.

Insbesondere die in J2EE 1.4 verwendete EJB-Spezifikation 2.1 hat zu starker Kritik an komplexen, schwergewichtigen Komponentenmodellen geführt, so dass EJB 3.0 ein völlig neues und zur Version 2.1 nicht kompatibles Persistenzmodell einführte. Dieses ist leichtgewichtig und kann z. B. für den Bereich der Persistenz (JPA) auch außerhalb eines Java-EE-Containers, d. h. in Java-SE-Umgebungen verwendet werden. Viele der Java-EE-Spezifikationen machen vom sogenannten *Dependency-Injection*-Muster Gebrauch. Der Begriff wird im Allgemeinen Martin Fowler zugeschrieben. Unter Dependency-Injection versteht man vereinfacht ausgedrückt das Einspeisen eines Wertes (Objekts) in ein Objekt-Property von außerhalb, d. h. ohne aktives Zutun des Objekts. Wir werden später noch viele Beispiele dafür sehen, da JBoss-Seam ausgiebig von diesem Muster Gebrauch macht. Im Internet ist eine Diskussion [URL-DI] des Begriffs von Fowler verfügbar.

JBoss-Seam versucht, den mit Java-EE 5 eingeschlagenen Weg eines modernen Komponenten-Frameworks weiterzugehen. Weiter als ihn Java-EE 5 geht. Java-EE 5 definiert JSF als Komponenten-Framework für Oberflächen und EJB 3.0 als Komponenten-Framework für die Geschäftslogik und die Per-

sistenz. Die Integration dieser beiden Frameworks geschieht händisch durch die Definition von JSF-Managed-Beans, die in beliebiger Art und Weise auf Session- und Entity-Beans zugreifen dürfen. JBoss-Seam integriert JSF und EJB in einem einheitlichen Komponentenmodell und definiert damit einen Standard zur Integration von JSF und EJB, der in Java-EE so nicht vorhanden ist. Möglich wird dies, da sowohl JSF als auch EJB zwei Frameworks sind, die bereits entsprechende Erweiterungsmöglichkeiten vorgesehen haben.

Die Firma JBoss, mittlerweile eine Tochter von Red Hat, entwickelt den JBoss-Application-Server (JBoss-AS) als Open-Source-System. Auch Hibernate, ein weit verbreiteter OR-Mapper (object-relational), wird von JBoss als Open-Source-System entwickelt, nachdem er ursprünglich von Gavin King konzipiert und implementiert wurde. Gavin King ist Mitarbeiter von JBoss und maßgeblicher Kopf der Seam-Entwicklung. Das bei der Implementierung eines der innovativsten Application-Server und einem der innovativsten OR-Mapper gewonnene Know-How fließt auch in die Entwicklung von Seam ein. Die entwickelten Konzepte sind so attraktiv, dass innerhalb des Java-Community-Prozesses der schon erwähnte Java-Specification-Request 299 [URL-WB] ins Leben gerufen wurde. Unter dem Namen *Web-Beans* wird von Red Hat, Sun, Oracle, Google, Adobe, Apache und anderen versucht, Seam-Konzepte in eine offizielle Java-Spezifikation einfließen zu lassen. Man kann davon ausgehen, dass Web-Beans Bestandteil einer zukünftigen Java-EE-Spezifikation sein wird.

Seams Standardplattform ist der JBoss-AS. Wir erwähnten aber in Kapitel 1, dass Seam auch in einem Servlet-Container lauffähig ist, wenn der einbettbare EJB-Container von JBoss verwendet wird. Da wir davon ausgehen, dass Seam in Form von Web-Beans seinen Weg in die Java-EE-Welt finden wird, konzentrieren wir uns im weiteren Verlauf dieses Buches auf Java-EE. Alle Anwendungen, deren Code im Buch abgebildet ist, wurden auf dem JBoss-AS entwickelt und getestet. Sie sind in der Regel sofort auf dem JBoss-AS lauffähig. In dem von uns verwendeten Release von Seam sind alle in der Distribution mitgelieferten Beispielanwendungen sowohl im JBoss-AS als auch im Tomcat deploybar. Falls Sie Glassfish oder einen anderen Container verwenden wollen, konsultieren Sie bitte die entsprechende Dokumentation der Distribution.

Bei der Einführung von Seam werden wir in diesem Kapitel Konzepte beleuchten, die dem Leser mit Erfahrungen im Bereich von JSF und EJB/JPA zumindest in abgeschwächter Form bekannt sind. Im Kapitel 4 befassen wir uns mit neuen Konzepten, die in dieser Form noch nicht Bestandteil von Java-EE sind.

2.1 JSF und Facelets

Wir können in diesem Buch keine vollständige Einführung in JavaServer-Faces geben, wollen aber einige grundlegende Konzepte erläutern. Es gibt eine ganze Reihe guter JSF-Bücher, z. B. [Man05, SB07] oder deutschsprachig etwa [Mü06], die wir dem Leser für eine ausführliche Einführung in JavaServer-Faces empfehlen.

JavaServer-Faces hatten zum Ziel, ein Komponentenmodell für Oberflächenelemente zu entwickeln, das es ermöglicht, technisch hochwertige, den Erfordernissen moderner Anwendungen entsprechende Oberflächen auf der Basis des MVC-Patterns zu entwickeln. Zu den für diese Einführung wichtigsten im JSR 127 genannten Anforderungen gehören:

- die Verwaltung der Zustände von UI-Komponenten über mehrere Anfragen hinweg
- die Darstellungs- und Client-Unabhängigkeit
- die Realisierung von UI-Komponenten auf dem Server durch leichtgewichtige JavaBeans, die auf client-generierte Events reagieren können
- ein API zur Validierung von Benutzereingaben und automatische Konvertierung der Eingaben von Strings in andere Typen und zurück

Um diese Anforderungen zu realisieren, definiert JSF ein Komponenten- und ein Bearbeitungsmodell. Das Komponentenmodell stellt UI-Komponenten als JavaBeans dar, die über zugeordnete, aber austauschbare Views in beliebige Darstellungstechniken gerendert werden. Jede JSF-konforme Implementierung muss als Darstellungstechnik mindestens HTML auf Basis von JSPs bereitstellen. Das Bearbeitungsmodell definiert sechs Phasen für eine Anfragebearbeitung. In der ersten erfolgt das Wiederherstellen des letzten Zustands einer Komponente bzw. aller Komponenten der betreffenden Seite und in der letzten Phase das Abspeichern des aktuellen Zustands. Andere Phasen sind für Validerung, Konvertierung und Event-Verarbeitung zuständig. Die letzte Phase erzeugt schließlich noch die Darstellung der Seite. Dieses exakt definierte, modularisierte und konfigurierbare Komponenten- und Bearbeitungsmodell erlaubt es Seam, sich an den entsprechenden Bearbeitungspunkten einzuklinken und Seam-eigene Komponenten ins Spiel zu bringen.

Die genannte Darstellungsunabhängigkeit ermöglicht es, den in der sechsten Bearbeitungsphase eingesetzten View-Handler, im Standardfall JSP/HTML, auszutauschen. Eine Alternative ist *Facelets*, eine von Jacob Hookom entwickelte View-Technik auf XHTML-Basis. Jacob Hookom ist Entwickler bei

Sun und arbeitet unter anderem an der JSF-Referenzimplementierung und an Glassfish [URL-GF], der Open-Source-Implementierung eines Java-EE-Containers von Sun.

Obwohl einige JSF-Bücher teilweise auch auf Facelets eingehen (z. B. [Mü06] und [SB07]), wollen wir hier Facelets kurz einführen, da im Augenblick kein explizites Buch über Facelets existiert. Für umfassende Informationen sei der Leser auf die Facelets-Homepage [URL-FL] verwiesen. JBoss-Seam enthält Facelets, es ist also nicht notwendig, Facelets gesondert herunterzuladen.

Facelets verwenden als Auszeichnungssprache XHTML und nicht JSP. Die Verwendung von JSP zur Definition von JSF-Seiten führt zu einigen Problemen, die z. B. von Hans Bergsten in seinem Artikel *Improving JSF by Dumping JSP* [URL-BE] diskutiert werden. Auch wenn einige dieser Probleme mit JSF 1.2 der Vergangenheit angehören, bleibt der Umstand, dass JSF auf Basis von JSP einen JSP-Container benötigt, während Facelets ohne einen solchen Container auskommt. Die Seam-Entwickler raten zur Verwendung von Facelets, und nur wenige Beispiele im Seam-Download sind mit JSP, der überwiegende Teil ist mit Facelets implementiert. Wir gehen nun kurz auf die syntaktischen Unterschiede zwischen JSP und Facelets ein und verwenden im weiteren Verlauf des Buches bis auf eine Ausnahme dann ausschließlich Facelets.

In Abschnitt 2.4 werden wir eine Online-Banking-Anwendung mit Seam implementieren. Die Login-Seite der Anwendung besteht aus zwei Eingabemöglichkeiten für den Benutzernamen und das Passwort sowie einer Schaltfläche zum Abschicken des Formulars. Die Implementierung der Seite mit JSP zeigt der folgende Code-Ausschnitt:

```
<h:form id="anmeldung">
  <h:panelGrid columns="2" styleClass="entry">
    <h:outputText value="Kundennummer:" styleClass="label" />
    <h:inputText id="kundennummer" value="#{handler.username}"
                 styleClass="input" />
    <h:outputText value="Passwort:" styleClass="label" />
    <h:inputSecret id="passwort" value="#{handler.password}"
                   styleClass="input" />
    <h:commandButton action="#{handler.login}" value="Anmelden"
                     styleClass="button" />
    <h:panelGroup />
    </h:panelGrid>
</h:form>
```

Man erkennt die übliche Vorgehensweise bei der Erstellung einer JSF-Seite mit JSP, nämlich HTML- und JSP-Tags zu vermeiden, da dies zu Problemen führen kann. Das `<h:panelGrid>`-Tag wird durch JSF in ein HTML-`<table>`-Tag gerendert. Komplexere Layouts werden in JSF/JSP durch geschachtelte

`<h:panelGrid>` realisiert und sind damit nach dem Rendern verschachtelte HTML-Tabellen. Der modernere Ansatz für das Layout von Web-Seiten ist der Einsatz des HTML-`<div>`-Tags und von Cascading Style Sheets (CSS). Mit Facelets ist dies problemlos möglich. Die Login-Seite liest sich in ihrer Facelets-Realisierung wie folgt:

```
<h:form id="anmeldung">
  <fieldset>
    <div class="entry">
      <div class="label">Kundennummer:</div>
      <div class="input">
        <h:inputText id="kundennummer"
                     value="#{handler.username}" />
      </div>
    </div>
    <div class="entry">
      <div class="label">Passwort:</div>
      <div class="input">
        <h:inputSecret id="passwort"
                       value="#{handler.password}" />
      </div>
    </div>
  </fieldset>
  <fieldset>
    <div class="entry">
      <h:commandButton action="#{handler.login}"
                       value="Anmelden" class="button" />
    </div>
  </fieldset>
</h:form>
```

Diese Version der Login-Seite ist zwar deutlich umfangreicher als die JSP-Version, jedoch nur aufgrund der vielen `<div>`-Tags. Diese erlauben andererseits jedoch ein praktisch beliebiges Layout der Seite mit CSS.

Facelets geht aber einen Schritt weiter und erlaubt die Verwendung des Attributs `jsfc` in jedem HTML-Tag. Mit diesem Attribut wird für das HTML-Tag angezeigt, dass es von JSF speziell zu behandeln ist, nämlich als die UI-Komponente, die als Wert des `jsfc`-Attributs angegeben ist. Die folgende Code zeigt die Überarbeitung mit dem `jsfc`-Attribut.

```
<form jsfc="h:form" id="anmeldung">
  <fieldset>
    <div class="entry">
      <div class="label">Kundennummer:</div>
      <div class="input">
        <input type="text" jsfc="h:inputText" id="kundennummer"
               value="#{handler.username}" />
```

```
        </div>
      </div>
      <div class="entry">
        <div class="label">Passwort:</div>
        <div class="input">
          <input type="password" jsfc="h:inputSecret"
                 id="passwort" value="#{handler.password}" />
        </div>
      </div>
    </fieldset>
    <fieldset>
      <div class="entry">
        <input type="submit" jsfc="h:commandButton"
               action="#{handler.login}"
               value="Anmelden" class="button" />
      </div>
    </fieldset>
  </form>
```

Die JSF-Seite ist nun „reines“ HTML bzw. XHTML und kann daher mit gängigen Entwicklungswerkzeugen für Web-Seiten bearbeitet werden.

Die bisher beschriebenen Unterschiede zwischen JSF mit JSP und JSF mit Facelets waren hauptsächlich syntaktischer Natur und zum Teil vernachlässigbar. Einer der Hauptgründe für die Verwendung von Facelets ist ein mächtiger Template-Mechanismus, der deutlich über die Möglichkeiten der Tags `<jsp:include>` und `<%@ include %>` in JSP hinausgeht. So ist es zum Beispiel möglich, Parameter von der inkludierenden zur inkludierten Datei zu übergeben. Wir werden diese Möglichkeit in Abschnitt 2.4 nutzen, um ein einheitliches Layout aller Seiten der Anwendung mit Kopf- und Fußzeile zu realisieren.

Wir beschließen diesen Abschnitt mit einem kleinen Beispiel zur Demonstration des Template-Mechanismus. Das Beispiel verwendet die Tags `<ui:composition>`, `<ui:define>`, `<ui:include>`, `<ui:insert>` und `<ui:param>`. Als Referenz der Facelets-Tags verwenden Sie bitte den Anhang D. Wir überarbeiten die bekannte Login-Seite und verwenden Facelets Template-Mechanismus. Die Login-Seite erhält dazu eine Menü- und eine Fußzeile. Listing 2.1 zeigt die Template-Seite `template.xhtml`.

Listing 2.1: Die Seite `template.xhtml`

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2       xmlns:ui="http://java.sun.com/jsf/facelets"
3       xmlns:h="http://java.sun.com/jsf/html"
4       xmlns:f="http://java.sun.com/jsf/core"
```

```
5        xmlns:s="http://jboss.com/products/seam/taglib">
6  <head>
7    <title><ui:insert name="title" /></title>
8    <link type="text/css" rel="stylesheet" href="css/login.css" />
9    <meta http-equiv="Content-Type"
10         content="text/html; charset=UTF-8" />
11 </head>
12 <body>
13   <ui:include src="menu.xhtml">
14     <ui:param name="param1" value="Template-Demo-Menü"/>
15   </ui:include>
16   <div class="body">
17     <ui:insert name="body"/>
18   </div>
19   <div class="footer">
20     Das ist die Fußleiste.
21   </div>
22 </body>
23 </html>
```

Interessant sind die beiden `<ui:insert>`-Tags in den Zeilen 7 und 17 sowie das `<ui:include>`-Tag in den Zeilen 13–15 mit dem geschachtelten `<ui:param>`-Tag. Die Datei, deren mit `<ui:define>` definierten Bereiche die `<ui:insert>`-Tags ersetzten, nennt man *Template-Client*. Ein solcher Template-Client ist die Datei `login-mit-template.xhtml`, dargestellt in Listing 2.2.

Listing 2.2: Die Seite `login-mit-template.xhtml`

```
1  <html>
2  <body>
3  Was hier steht wird nicht gerendert
4  <ui:composition template="template.xhtml">
5
6    <ui:define name="body">
7      <h:form id="anmeldung">
8        <div class="entry">
9          <div class="label">Kundennummer:</div>
10         <div class="input">
11           <h:inputText id="kundennummer"
12                        value="#{handler.username}" />
13         </div>
14       </div>
15       <div class="entry">
16         <div class="label">Passwort:</div>
17         <div class="input">
18           <h:inputSecret id="passwort"
```

```
19                                  value="#{handler.password}" />
20              </div>
21          </div>
22          <div class="entry">
23              <h:commandButton action="#{handler.login}"
24                                  value="Anmelden" class="button" />
25          </div>
26      </h:form>
27  </ui:define>
28
29  <ui:define name="title">
30          Login der Template-Demo
31  </ui:define>
32
33 </ui:composition>
34 Was hier steht wird nicht gerendert
35 </body>
36 </html>
```

In den Zeilen 6–27 und 29–31 werden die Bereiche mit Namen `body` und `title` definiert. Im Tag `<ui:composition>` in Zeile 4 wird das Template aus Listing 2.1 referenziert. Diese Komposition wird durch das Template ersetzt, wobei die im Template verwendeten `<ui:insert>`-Tags wiederum durch die beiden mit `<ui:define>` definierten Bereiche ersetzt werden. Markup und Text außerhalb des `<ui:composition>`-Elements wird dabei nicht gerendert.

Als letztes noch nicht analysiertes Konstrukt bleibt das `<ui:include>`-Tag mit geschachteltem `<ui:param>` in den Zeilen 13–15 des Listings 2.1. Die dort inkludierte Datei `menu.xhtml` ist im Folgenden dargestellt:

```
<body>
<div class="menu">
   <h:outputText value="#{param1}:"/>
   <s:link view="/dummy-1.xhtml" value="Dummy 1" />
   <s:link view="/dummy-2.xhtml" value="Dummy 2" />
   <s:link view="/dummy-3.xhtml" value="Dummy 3" />
</div>
</body>
```

Interessant ist hier die Verwendung des an die inkludierte Datei übergebenen Parameters in der dritten Zeile als Wert des `value`-Attributs. Im Beispiel war der Wert des `<ui:param>`-Tags ein String-Literal. Erlaubt sind jedoch allgemeine EL-Ausdrücke, so dass dieses Konstrukt sehr weitreichende Möglichkeiten bietet. Die Abbildung 2.1 zeigt die Browserdarstellung der template-basierten Login-Seite.

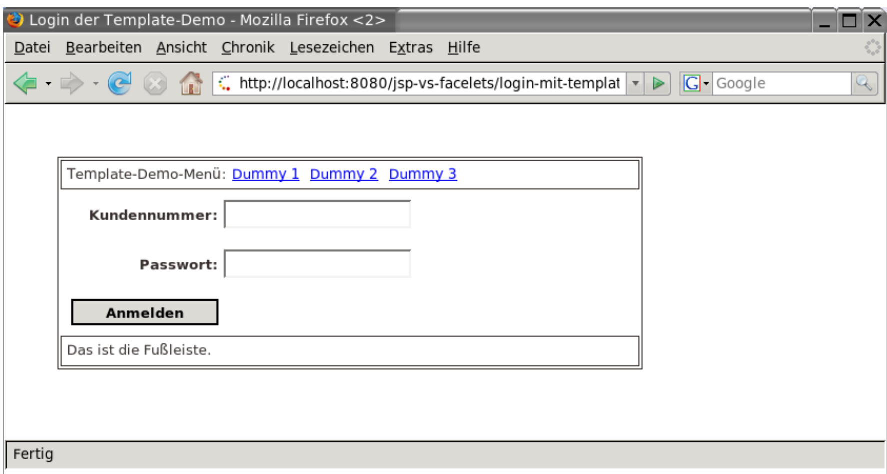

Abbildung 2.1: Template-basierte Login-Seite

Wir verwenden Facelets Template-Mechanismus in unserer Online-Banking-Anwendung nicht und gehen hier eher „konservativ" vor. Sie finden aber viele Beispiele zum Template-Mechanismus im Seam-Download. Der Seam-Anwendungsgenerator, den wir in Abschnitt 7.4 vorstellen, verwendet ebenfalls den Template-Mechanismus.

2.2 EJBs und JPA

Die Java-Enterprise-Edition (Java-EE) definiert eine Reihe zentraler Komponenten unter dem Oberbegriff Enterprise-JavaBeans (EJB). Darunter werden Session-Beans, Entity-Beans und Message-Driven-Beans verstanden. In der Version 5 der Java-EE-Spezifikation werden EJBs in der Version 3.0 definiert. Das in früheren EJB-Versionen (bis einschließlich Version 2.1) verwendete Komponentenmodell insbesondere für Entity-Beans wurde von vielen Entwicklern als unhandlich und umständlich empfunden. Dies führte zu alternativen Ansätzen, die sich z. B. in Buchtiteln wie *J2EE Development without EJB* [JH04], Anwendungs-Frameworks wie Spring [URL-SPRING] oder Persistenz-Frameworks wie Hibernate [URL-HIBERNATE] niedergeschlagen haben. EJB 3.0 kann als Reaktion auf diese Kritik angesehen werden und wird allgemein als leichtgewichtiges und modernes Framework empfunden. Im Rahmen dieses Buches über Seam kann keine Einführung in Java-EE 5 bzw. EJB 3.0 gegeben werden. Dies ist umfangreicheren Büchern, wie etwa dem gelungenen Buch von Burke und Monson-Haefel [BMH06] oder dem Java-EE-Tutorial [JBC+06] vorbehalten. Wir werden aber kurz auf Session- und Entity-Beans eingehen und

diese in Grundzügen vorstellen. Die Definition von Entity-Beans erfolgt innerhalb von EJB 3.0 in einer eigenen Spezifikation, der Java-Persistence-API (JPA), Version 1.0. JPA wurde stark von Hibernate beeinflusst. Als Einführung in JPA und Hibernate empfehlen wir das Buch von Bauer und King [BK07b] bzw. die deutsche Übersetzung [BK07a]. King ist der konzeptionelle Kopf hinter Hibernate und Seam. Alternativ empfehlen wir unser eigenes Buch [MW07], in dem wir uns auf JPA konzentrieren.

Seam verwendet Session- und Entity-Beans. Message-Driven-Beans werden für den asynchronen Nachrichtenaustausch verwendet und gehören damit standardmäßig in einem GUI-basierten System wie Seam nicht zu den zentralen Komponenten. Seam erlaubt die Verwendung von Message-Driven-Beans, worauf wir jedoch nicht weiter eingehen. Session-Beans werden verwendet, um Geschäftsprozesse, Anwendungsfälle und Benutzerinteraktionen zu implementieren, Entity-Beans, um persistente Daten zu repräsentieren, die Dinge des Geschäftsmodells darstellen.

Session-Beans definieren die Schnittstelle zur Client-Anwendung und realisieren Anwendungsfälle und Geschäftsprozesse. Eine Session-Bean erzeugt z.B. ein neues Konto, das durch ein Entity-Bean repräsentiert wird. Oder sie realisiert den Anwendungsfall einer Überweisung, der in der Regel mehrere Entity-Beans manipuliert. Die durch Session-Beans realisierte Funktionalität ist typischerweise transient. Obwohl innerhalb einer Überweisung Änderungen an Konten persistent zu machen sind und eine Transaktionsnummer (TAN) in der Datenbank als bereits verwendet zu kennzeichnen ist, hat die Session-Bean selbst keinen *persistenten* Zustand. Man unterscheidet jedoch sehr wohl zwischen zustandslosen (stateless) und zustandsbehafteten (stateful) Session-Beans. Eine zustandslose Session-Bean hält keinen für die Anwendung wichtigen inneren Zustand und kann daher vom EJB-Container beliebig häufig instantiiert und eingehenden Client-Requests zugeordnet werden. Die EJB-Spezifikation sieht zur Definition einer zustandslosen Session-Bean die `@Stateless`-Annotation vor. Die Annotation ist im Package `javax.ejb` enthalten und daher qualifiziert als `@javax.ejb.Stateless` zu schreiben. Wir werden im Folgenden aber bei allen Annoationen die nicht qualifizierte Schreibweise verwenden, da Mehrdeutigkeiten ausgeschlossen sind.

Eine zustandsbehaftete Session-Bean repräsentiert mit ihrem Zustand anwendungsrelevante Informationen, z. B. den Inhalt eines Warenkorbs in einem Online-Shop. Typischerweise werden zustandsbehaftete Session-Beans in Web-Anwendungen einer HTTP-Session zugeordnet. Da Seam aber eine Reihe weiterer Kontexte einführt, sind auch andere Kontextzuordnungen möglich. Eine

zustandsbehaftete Session-Bean wird durch die `@Stateful`-Annotation definiert.

Session-Beans werden durch eine normale POJO-Klasse implementiert und mit den bereits genannten Annotationen versehen. Optional kann eine Session-Bean drei Interfaces implementieren: ein remote Interface, ein lokales Interfaces und ein Endpoint-Interface. Endpoint-Interfaces definieren die Methoden, die eine Session-Bean über das *Simple Object Access Protocol* (SOAP) und die *Web Service Description Language* (WSDL) als Web-Service zur Verfügung stellt und sind daher für den von uns betrachteten Seam-Bereich nicht relevant. Remote Interfaces definieren Methoden, die von Clients außerhalb des EJB-Containers aufgerufen werden können. Das Interface wird mit `@Remote` annotiert. Lokale Interfaces definieren Methoden, die von Beans aufgerufen werden können, die sich im selben EJB-Container befinden. Sie werden mit `@Local` annotiert. Wir verwenden im Folgenden ausschließlich lokale Interfaces.

Interface- und Bean-Namen sind frei wählbar. Es ist jedoch sinnvoll, sich in einem Projekt auf eine einheitliche Namenskonvention zu einigen. Die im Seam-Download enthaltenen Beispiele verwenden für das lokale Interface beliebige Namen und bilden den Namen der implementierenden Bean-Klasse durch Anhängen von „`Action`“. Im Java-EE-Tutorial von Sun wird die Bean-Klasse mit dem Suffix „`Bean`“ versehen. Wir finden beide Alternativen nicht optimal und werden den Charakter einer Session-Bean, nämlich die Realisierung eines bestimmten Vorgangs durch den Namensbestandteil „`Handler`“ ausdrücken. Das lokale Interface erhält den Suffix „`Local`“. Für eine Session-Bean, die den Login-Vorgang realisiert, würde dies also zur Bean-Klasse `LoginHandler` und zum lokalen Interface `LoginHandlerLocal` führen.

Entity-Beans werden verwendet, um Objekte des Business-Modells persistent verwalten zu können. In Abschnitt 2.4 implementieren wir beispielhaft eine Online-Banking-Anwendung. Entity-Beans dieses Business-Modells sind z. B. Kunden, Konten und Buchungen. Zur bereits angesprochenen sehr einfachen Definition eines Entity-Beans in EJB 3.0 bzw. der enthaltenen Spezifikation JPA 1.0 genügt die `@Entity`-Annotation für die Bean-Klasse und die `@Id`-Annotation für das Primärschlüssel-Property. Ansonsten ist die Bean-Klasse ein POJO, und als Default werden alle Properties persistent gemacht.

Diese sehr kurze Einführung in EJB 3.0 soll hier genügen. Der interessierte Leser findet in [BMH06] und [MW07] ausführliche Einführungen in Java-EE und JPA.

2.3 Seam-Komponenten

Komponenten-Frameworks wurden vor allem durch Web-Frameworks wie Struts, Tapestry oder Spring populär. Es ist heute allgemeiner Konsens, dass Komponenten-Frameworks die Verwendung möglichst einfacher Objekte ermöglichen sollen. Gängig ist der Begriff eines *Plain Old Java Object*s (POJO), manchmal auch *Plain Ordinary Java Object* genannt, der für ein einfaches Java-Objekt steht, das keine Interfaces implementieren muss und auch möglichst von keiner anderen Klasse abgeleitet werden muss. Als Gegenbeispiel hierzu ist EJB 2.1 zu nennen, wo Beans bestimmte Interfaces mit Callback-Methoden implementieren mussten. Ein weiterer Aspekt ist die Verwendung von XML zur Konfiguration von Komponenten. Dies wird in EJB 2.1 aber auch in den anfangs genannten Frameworks sehr umfangreich praktiziert. Es besteht auch hier Konsens, dass ausufernde XML-Konfigurationen möglichst vermieden werden sollten.

Mit JSF wurde zum ersten Mal eine offizielle Java-Spezifikation (JSR 127) erstellt, die den Komponentengedanken realisiert. Zwar sind die internen JSF-Komponenten gezwungen, bestimmte APIs zu realisieren. Die anwendungsorientierten Komponenten, die Managed-Beans, müssen dies jedoch nicht. Managed-Beans müssen kein bestimmtes Interface implementieren und werden vom Servlet-Container bzw. der JSF-Laufzeitumgebung verwaltet. Sie werden bei Bedarf automatisch erzeugt und, falls nicht mehr benötigt, auch wieder entfernt. Sie verwenden das Dependency-Injection-Muster, um z. B. untereinander über Properties verbunden werden zu können. Die Definition erfolgt allerdings auf Basis einer XML-Konfigurationsdatei.

Mit der Spezifikation EJB 3.0 wurde diese Entwicklung weiter vorangetrieben. Der Anteil von XML-Konfigurationsdateien wurde stark reduziert. Die benötigten Konfigurationsinformationen werden durch Annotationen direkt in den Java-Klassen definiert, und die Verwendung des Dependency-Injection-Musters wurde auf breiter Basis etabliert.

Seam perfektioniert diese Entwicklung und definiert eine Reihe von Komponenten (besser eines Komponentenmodells), die den genannten Anforderungen entsprechen:

- keine (wenig) XML-Konfiguration
- Konfiguration durch Annotationen
- keine API-Anforderungen, POJOs genügen
- automatisierte Verwaltung

- Auflösen von Abhängigkeiten oder Verbindungen durch Dependeny-Injection bzw. einer Erweiterung dieses Musters

Wir führen an dieser Stelle das Seam-Komponentenmodell zunächst im Überblick ein und stellen im weiteren Verlauf des Buches die meisten dieser Komponenten detailliert vor. Seam-Komponenten sind POJOs und als mögliche Ausprägung dieses Musters einfache JavaBeans sowie Session-, Entity- und Message-Driven-Beans entsprechend EJB 3.0. Seam-Komponenten sind mit Ausnahme von Message-Driven-Beans jeweils einem *Kontext* zugeordnet. Je nach Kontext ist es einer Komponente möglich, einen eigenen Zustand zu haben und diesen innerhalb der Anwendung sinnvoll einzusetzen. Mögliche Kontexte sind:

- **Stateless**
  Komponenten dieses Kontextes können keinen Zustand verwalten.
- **Event**
  Komponenten dieses Kontextes halten ihren Zustand während der Bearbeitung einer einzelnen JSF-Anfrage.
- **Page**
  Komponenten diese Kontextes halten ihren Zustand bzgl. einer gerenderten JSF-Seite.
- **Conversation**
  Komponenten dieses Kontextes halten ihren Zustand über eine Konversation, d. h. eine Reihe von JSF-Anfragen.
- **Session**
  Komponenten diese Kontextes halten ihren Zustand in der entsprechenden HTTP-Session.
- **Business-Process**
  Komponenten dieses Kontextes halten ihren Zustand über einen länger andauernden Geschäftsprozess.
- **Application**
  Komponenten diese Kontextes sind statisch und entsprechend global.

Der Event-, Session- und Application-Kontext entsprechen dem Request-, Session- und Application-Kontext des Servlet-APIs und damit auch des darauf basierenden JSF-Kontext-Modells. Der Conversation- und der Business-Process-Kontext sind Seam-Innovationen und als solche besonders hervorzuheben, da sie auf der Anwendungsebene und nicht auf der Technikebene definiert sind. Dies ist eine Seam-Besonderheit und kommt in anderen Frameworks so nicht

vor. Konversationen werden ausführlich in Kapitel 3, Geschäftsprozesse in Kapitel 5 eingeführt.

2.4 Das Anwendungsbeispiel

In diesem Buch führen wir Seam anhand eines Beispiels ein. Ein solches Beispiel darf nicht trivial sein, sonst können fortgeschrittene Merkmale Seams nicht eingesetzt werden. Es darf aber auch nicht zu komplex und umfangreich sein, sonst verwehrt die Fachlichkeit den Blick auf das Wesentliche. Viele positive Meinungen zu unserem JSF-Buch [Mü06] haben uns ermutigt, auch hier eine Online-Banking-Anwendung als zentrales Beispiel zu verwenden. Die Fachlichkeit ist vielen Lesern bekannt, die Daten und Prozesse geben genügend Spielraum für die Demonstration von Seam-Merkmalen.

Natürlich kann ein Lehrbeispiel nicht die Realität abbilden. Wir bitten daher alle Leser, die als Entwickler bereits ähnliche Anwendungen realisiert haben, um Nachsicht: Die Fachlichkeit ist von einem Bank-Laien definiert, und zwar mit Hinblick auf eine möglichst gute Präsentation von Seam-Merkmalen und nicht eine möglichst realitätsgetreue Darstellung der Bankfachlichkeit.

Die nun zu entwickelnde Seam-Anwendung realisiert gundlegende Anwendungsfälle und führt in die zentralen, in jeder Seam-Anwendung vorhandenen Komponenten und deren Zusammenspiel ein. In den nächsten Kapiteln vertiefen wir dies und setzen auch Komponenten ein, die nicht notwendigerweise in jeder Anwendung vorhanden sind. Grundlage ist ein Anwendungmodell mit den Klassen `Kunde`, `Konto` und `Buchung`, wobei es mehrere Ausprägungen eines Kontos gibt. Abbildung 2.2 zeigt das UML-Klassendiagramm der Fachlichkeit.

Wir beginnen mit einem möglichst einfachen Anwendungsfall, der Stammdatenverwaltung. Listing 2.3 zeigt das Formular der Stammdatenverwaltung, das in der Facelets-Datei `stammdaten.xhtml` enthalten ist.

Listing 2.3: Das Formular für die Stammdatenverwaltung

```
1  <h:form id="stammdaten">
2
3    <s:validateAll>
4      <f:facet name="afterInvalidField">
5        <s:div styleClass="errors">
6          <h:graphicImage url="/images/error_small.png" />
7          <s:message/>
8        </s:div>
```

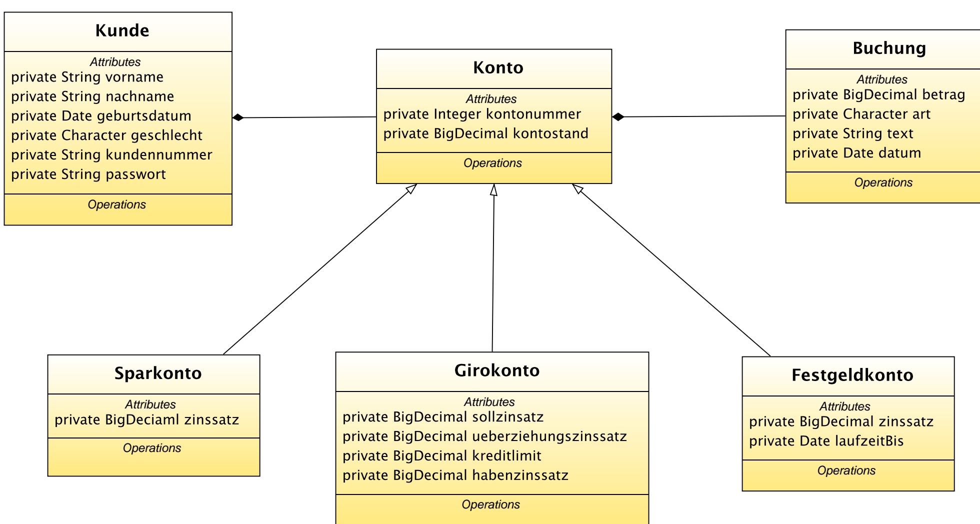

Abbildung 2.2: Das Klassendiagramm der Anwendung

```
 9      </f:facet>
10
11      <div class="usrFormEntry">
12        <h:outputLabel for="nachname" styleClass="usrFormLabel">
13          Anrede:
14        </h:outputLabel>
15        <h:selectOneMenu id="anrede" value="#{kunde.geschlecht}"
16                         styleClass="usrFormValue">
17          <f:selectItem itemLabel="Frau" itemValue="w" />
18          <f:selectItem itemLabel="Herr" itemValue="m" />
19        </h:selectOneMenu>
20      </div>
21      <div class="usrFormEntry">
22        <h:outputLabel for="nachname" styleClass="usrFormLabel">
23          Vorname:
24        </h:outputLabel>
25        <s:decorate>
26          <h:inputText id="vorname" value="#{kunde.vorname}"
27                       required="true" styleClass="usrFormValue" />
28        </s:decorate>
29      </div>
30      <div class="usrFormEntry">
31        <h:outputLabel for="nachname" styleClass="usrFormLabel">
32          Nachname:
33        </h:outputLabel>
34        <s:decorate>
```

```
35          <h:inputText id="nachname" value="#{kunde.nachname}"
36                       required="true" styleClass="usrFormValue" />
37        </s:decorate>
38      </div>
39      <div class="usrFormEntry">
40        <h:outputLabel for="geburtsdatum"
41                       styleClass="usrFormLabel">
42          Geburtsdatum:
43        </h:outputLabel>
44        <s:decorate>
45          <h:inputText id="geburtsdatum"
46                       value="#{kunde.geburtsdatum}" required="true"
47                       styleClass="usrFormValue" />
48        </s:decorate>
49      </div>
50    </s:validateAll>
51
52    <h:commandButton action="#{stammdatenHandler.speichern}"
53                     value="Speichern" styleClass="usrFormSubmit"/>
54
55  </h:form>
```

Das Formular erweist sich als (fast) normale Facelet-Seite mit JSF-Tags der HTML- und Kernbibliothek. Tags der HTML-Bibliothek haben das Präfix `h`, etwa `<h:form>` in Zeile 1, Tags der Kernbibliothek das Präfix `f`, etwa `<f:facet>` in Zeile 4. Lediglich das `<s:validateAll>`-Tag in Zeile 3, das `<s:div>`-Tag in Zeile 5, das `<s:message>`-Tag in Zeile 7 und das mehrfach vorkommende `<s:decorate>`-Tag, z. B. in Zeile 25, sind keine JSF-Standard-Tags, sondern Seam-Tags. `<s:validateAll>` validiert alle enthaltenen JSF-Eingabekomponenten anhand der Validierungsbedingungen der den Eingabekomponenten zugewiesenen Properties. Dies wird gleich klarer. Das Tag `<s:decorate>` als Auszeichnungselement um eine Eingabe verwendet zeigt beim negativen Ausgang einer Validierung eine entsprechende Meldung an. Im Beispiel verwenden wir dazu die Facette `afterInvalidField`, die als JSF-Facette in den Zeilen 4-9 definiert wird. Es wird sowohl eine kleine Graphik als auch eine Nachricht (`<s:message>`) angezeigt. Alle Eingaben (`<h:selectOneMenu>` und `<h:inputText>`) verwenden als Wertebindung Properties der Bean `kunde`. Der Command-Button in Zeile 52 verwendet die Action-Methode `speichern` der Bean `stammdatenHandler`. Die Bean `kunde` ist eine Entity-Bean, die Bean `stammdaten` eine Session-Bean. Wir erläutern beide Beans in Kürze.

Das in Listing 2.3 auf Seite 22 dargestellte Formular ist der Facelets-Datei `stammdaten.xhtml` entnommen. Listing 2.4 auf der nächsten Seite zeigt diese

Datei ohne das Formular. Zunächst wird im Doctype XHTML als Inhaltstyp der Datei definiert. Dann werden in den Zeilen 4 bis 8 die Namensräume der XML-Datei eingeführt. Dies sind `ui` für die Facelets-Bibliothek, `h` und `f` für die beiden JSF-Standard-Bibliotheken und schließlich `s` für die Bibliothek der Seam-Tags.

Listing 2.4: Facelet Stammdatenverwaltung (`stammdaten.xhtml`)

```
1  <!DOCTYPE html PUBLIC
2          "-//W3C//DTD XHTML 1.0 Transitional//EN"
3          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5            xmlns:ui="http://java.sun.com/jsf/facelets"
6        xmlns:h="http://java.sun.com/jsf/html"
7        xmlns:f="http://java.sun.com/jsf/core"
8        xmlns:s="http://jboss.com/products/seam/taglib">
9
10 <head>
11 <link type="text/css" rel="stylesheet" href="css/style.css" />
12 <link type="text/css" rel="stylesheet" href="css/menu.css" />
13 <title>Stammdaten</title>
14 </head>
15
16 <body>
17
18 <ui:include src="menu.xhtml" />
19
20 <div id="cTitle">Stammdaten</div>
21 <div id="...">
22
23 <h:form id="stammdaten">
24
25   ...
26   ...
27
28 </h:form>
29 </div>
30
31 <ui:include src="footer.xhtml" />
32
33 </body>
34 </html>
```

Das Layout des Formulars in Listing 2.3 wurde ohne HTML-Tabellen implementiert. Die `<div>`-Tags wurden mit `class`-Attributen, die JSF-Tags mit `styleClass`-Attributen versehen, so dass sich das Layout mit Cascading Style

Sheets (CSS) realisieren lässt. In den Zeilen 11 und 12 werden zwei CSS-Dateien eingebunden, die das Format der Seite bzw. des Menüs definieren.

In Abschnitt 2.1 hatten wir Facelets als die JSP überlegene Render-Technologie motiviert. Ein Vorteil von Facelets, die Möglichkeit der Definition eines Layout-Templates, kommt durch die <ui:include>-Tags in den Zeilen 18 und 31 zum Einsatz. In den Dateien menu.xhtml und footer.xhtml werden das Menü und die Fußzeile der Anwendung realisiert. Wir gehen hier darauf nicht näher ein und verweisen den Leser auf Abschnitt 2.1. Die Zeilen 23 bis 28 enthalten das Formular aus Listing 2.3. Die Darstellung der Seite zeigt Abbildung 2.3.

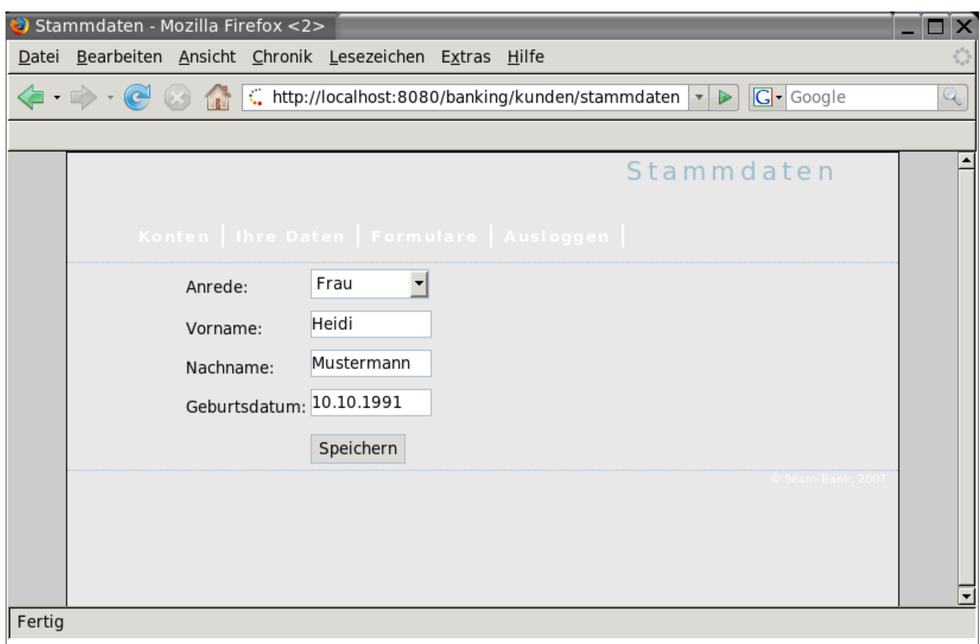

Abbildung 2.3: Stammdaten im Browser (stammdaten.xhtml)

Seam verwendet die in Abschnitt 2.2 eingeführten Entity-Beans als Persistenzobjekte. Entity-Beans sind gewöhnliche POJOs, d. h. sie müssen kein Interface implementieren und von keiner Klasse ableiten. Listing 2.5 auf der nächsten Seite zeigt das Entity-Bean Kunde, das das Interface Serializable implementiert. Dies ist allerdings keine Forderung von Seam, sondern eine sinnvolle Konvention. In diesem Beispiel ist die Serialisierbarkeit nicht notwendig. Bei einer Erweiterung der Anwendung und einem Entity-Zugriff z. B. über RMI ist dies aber notwendig, so dass wir dem Leser diese Konvention empfehlen.

Entity-Beans sollten das `Serializable`-Interface implementieren, um eine möglichst allgemeine Verwendung zu ermöglichen. Dazu muss keine spezielle Methode implementiert werden. Es genügt die Deklaration von „`... implements Serializable`“.

Listing 2.5: Das Entity-Bean `Kunde`

```
1  @Entity
2  @Scope(SESSION)
3  @Name("kunde")
4  public class Kunde implements Serializable {
5
6      private Integer id;
7      private String vorname;
8      private String nachname;
9      private Date geburtsdatum;
10     private Character geschlecht;
11     private String kundennummer;
12     private String passwort;
13     private Set<Konto> konten = new HashSet<Konto>();
14
15     public Kunde() {
16         super();
17     }
18
19     ...
20
21     // ab hier Getter und Setter
22
23     @Id
24     @GeneratedValue(strategy = GenerationType.IDENTITY)
25     public Integer getId() {
26         return this.id;
27     }
28     public void setId(Integer id) {
29         this.id = id;
30     }
31
32
33     @Column(nullable = false)
34     @Length(min = 3, max = 30,
35         message =
36             "Vorname muss zwischen 3 und 30 Zeichen lang sein")
37     public String getVorname() {
38         return this.vorname;
39     }
40     public void setVorname(String vorname) {
```

```
41          this.vorname = vorname;
42      }
43
44
45      @Column(nullable = false)
46      @Length(min = 3, max = 30,
47          message =
48              "Nachname muss zwischen 3 und 30 Zeichen lang sein")
49      public String getNachname() {
50          return this.nachname;
51      }
52      public void setNachname(String nachname) {
53          this.nachname = nachname;
54      }
55
56
57      @Column(nullable = false)
58      @Temporal(TemporalType.DATE)
59      public Date getGeburtsdatum() {
60          return this.geburtsdatum;
61      }
62      public void setGeburtsdatum(Date geburtsdatum) {
63          this.geburtsdatum = geburtsdatum;
64      }
65
66
67      @Column(nullable = false)
68      public Character getGeschlecht() {
69          return geschlecht;
70      }
71      public void setGeschlecht(Character geschlecht) {
72          this.geschlecht = geschlecht;
73      }
74
75
76      @Column(nullable = false, unique = true)
77      public String getKundennummer() {
78          return kundennummer;
79      }
80      public void setKundennummer(String kundennummer) {
81          this.kundennummer = kundennummer;
82      }
83
84
85      @Column(nullable = false)
86      public String getPasswort() {
87          return passwort;
88      }
89      public void setPasswort(String passwort) {
```

```
90          this.passwort = passwort;
91      }
92
93      @OneToMany(mappedBy="kunde", cascade=CascadeType.ALL)
94      public Set<Konto> getKonten() {
95          return this.konten;
96      }
97      public void setKonten(Set<Konto> konten) {
98          this.konten = konten;
99      }
100 }
```

Die Minimalforderungen von JPA, um aus einem einfachen POJO ein persistentes Entity-Bean zu machen, sind trivial. Die Klasse selbst ist mit `@Entity`, das den Datenbankprimärschlüssel darstellende Property mit `@Id` zu annotieren. Das ist alles.

Wir können an dieser Stelle keine Einführung in JPA geben und verweisen den interessierten Leser auf die EJB-Spezifikation, die im Java-Specification-Request 220 [URL-EJB] beschrieben ist, sowie auf unser JPA-Buch [MW07]. An dieser Stelle führen wir nur in die wichtigsten Konzepte ein.

JPA ist innerhalb von Java-EE syntaktisch im Package `javax.persistence` definiert. Die qualifizierte Schreibweise für `@Entity` ist also `@javax.persistence.Entity`. Wir werden immer die Kurzform mit einer entsprechenden Import-Anweisung verwenden. EJB verwendet wie Seam das Konzept *Convention over Configuration* oder (gleichbedeutend) *Configuration by Exception*, so dass es für viele Fälle sinnvolle Default-Annahmen gibt und nur bei Abweichungen von diesen Annahmen eine explizite Angabe erforderlich ist. Eine Entity-Bean wird auf eine Tabelle mit demselben Namen abgebildet. Alle Properties sind persistent und werden auf Spalten mit denselben Namen und festgelegten Typen abgebildet. Sollen abweichende Tabellen- oder Spaltennamen verwendet werden, so sind diese mit den Annotationen `@Table` und `@Column` zu realisieren. Nicht persistente Properties werden mit `@Transient` annotiert.

Der Zugriff der JPA-Implementierung auf ein Property kann entweder direkt auf die Instanzvariable oder über das Getter/Setter-Paar erfolgen. Dies wird durch die Position der Annotation bestimmt. Wir werden in unseren Beispielen immer das Getter/Setter-Paar annotieren, so dass der Zugriff der JPA-Implementierung auch über das Getter/Setter-Paar erfolgt.

Beim Entity `Kunde` haben wir zusätzlich zur `@Id`-Annotation noch die Art der Primärschlüsselgenerierung mit der `@GeneratedValue`-Annotation definiert.

Der Wert `IDENTITY` bedeutet, dass der datenbankeigene Mechanismus zur automatischen Inkrementierung des Primärschlüssels verwendet wird. Ohne Verwendung der `@GeneratedValue`-Annotation erfolgt keine Schlüsselgenerierung, so dass die Anwendung einen Schlüssel vergeben muss. Die Verwendung der Annotation ohne Option entspricht der Default-Option `AUTO`, was der JPA-Implementierung die freie Wahl eines Generierungsverfahrens erlaubt.

Die Properties `vorname` und `nachname` sind identisch annotiert. Mit `@Column (nullable = false)` (Zeilen 33 und 45) werden Null-Werte für diese beiden Properties verboten. Die Annotation `@Length` (Zeilen 34 und 46) definiert Bedingungen für die Länge eines Strings. Im Beispiel werden sowohl eine Untergrenze als auch eine Obergrenze festgelegt. Zusätzlich wird noch der im Fehlerfalle anzuzeigende Text definiert.

Während `@Column` eine JPA-Annotation ist, ist `@Length` eine Hibernate-Validator-Annotation und daher im Package `org.hibernate.validator` enthalten. Hibernate-Validator [URL-HIBVAL] ist ein im Hibernate-Umfeld entstandenes System zur Validierung von Entity-Properties. Es ist mittlerweile nicht nur mit Hibernate, sondern mit jeder JPA-Implementierung und darüber hinaus auch ohne JPA einsetzbar, da die Ebene der Validierung konfigurierbar ist. Bei Seam wird die Validierung auf JSF-Ebene durchgeführt, wenn wie in unserem Beispiel die JSF-Seite ein `<s:validateAll>` enthält. Es erscheint dann durch die Facette `afterInvalidField` die in der `@Length`-Annotation definierte Meldung, wie in Abbildung 2.4 auf der nächsten Seite dargestellt. Ohne dieses Tag wird die Validierung auf der Persistenzebene durchgeführt und führt zu einer Exception in der Persistenzschicht.

Hibernate-Validator genießt eine große Popularität im Bereich der Validierungs-Frameworks und beeinflusst inhaltlich den Java-Specification-Request 303, Bean-Validation [URL-BEANVAL]. Der JSR 303 soll nach aktueller Planung in Java-SE 7 enthalten sein.

Während die JPA-Annotationen in jedem JPA-Buch nachzulesen sind, ist Hibernate-Validator nicht in Buchform dokumentiert. Wir führen daher alle Annotationen des Hibernate-Validators in Abschnitt B.9 auf. Als besonders hervorzuhebende Eigenschaft des Hibernate-Validators ist die einfache Integration eigener Validierer zu nennen, worauf wir jedoch nicht eingehen. Beispiele zur Definition eigener Validierer findet der Leser in unserem Buch [MW07] über JPA.

Seams Formularvalidierung verwendet das Hibernate-Validierungs-Framework, das server-seitig arbeitet und nicht etwa durch client-seitiges JavaScript realisiert ist. Validierungen sollten immer server-seitig erfolgen, um z. B. bösartige Clients auszuschließen. Optional kann zusätzlich eine client-seitige Validierung erfolgen, um Falscheingaben früher zu erkennen.

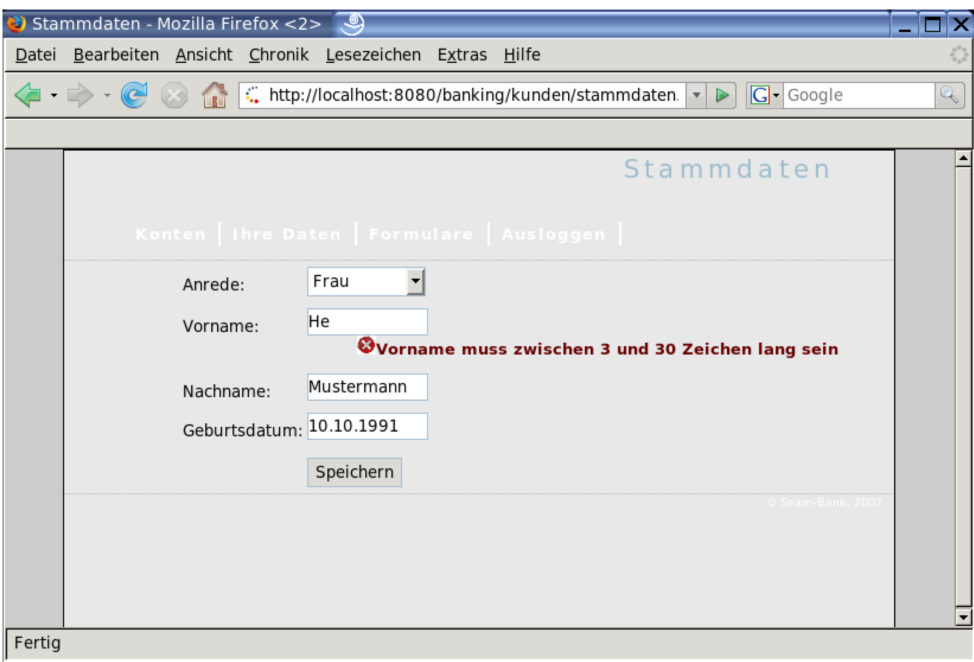

Abbildung 2.4: Durch `@Length`-Annotation generierte Fehlermeldung

Das Property `geburtsdatum` ist vom Typ `java.util.Date`, eine Standardmöglichkeit Javas zur Datumsdarstellung. Ein `Date` ist ein Zeitstempel in Millisekundengenauigkeit. Für die Abbildung auf verschiedene SQL-Datentypen kann mit der `@Temporal`-Annotation ein entsprechender Typ ausgewählt werden. Die Enumeration `TemporalType` kennt die Werte `DATE`, `TIME` und `TIMESTAMP`, die den SQL-Typen für Datum, Zeit und Zeitstempel entsprechen.

Das Property `kundennummer` unterscheidet sich von den anderen Properties durch den zusätzlichen Parameter `unique = true` in der `@Column`-Annotation. Dieser Parameter stellt die Eindeutigkeit des Properties auf Datenbankebene sicher.

Wir schließen die Erläuterung des Entitys `Kunde` mit dem Property `konten` ab. Das Property `konten` ist mengenwertig (`Set<Konto>`) und realisiert eine 1:n-Beziehung zur Klasse `Konto`. JPA unterstützt alle Arten von Beziehungen, sowoh uni- als auch bidirektional. Die `@OneToMany`-Annotation realisiert die gewünschte 1:n-Beziehung und definiert mit dem Attribut `mappedBy` die Spalte `kunde` in der das Konto realisierenden Tabelle als Fremdschlüssel in die Tabelle für `Kunde`. Durch den Wert `CascadeType.ALL` für das Attribut `cascade` werden Einfüge-, Update- und Löschoperationen von `Kunde` zu `Konto` kaskadiert, d. h. beim Anlegen eines neuen Kunden mit einem Konto genügt das Speichern des Kunden, das Konto wird automatisch mitgespeichert.

Seam verwendet neben dem schon erwähnten Hibernate-Validator auch das System Hibernate-Tools [URL-HIBTOOLS]. Mit diesen Werkzeugen ist es z. B. möglich, aus einem JPA-Entity die für ein bestimmtes Datenbanksystem benötigte DDL-Anweisung zur Tabellenerzeugung zu generieren. Seam bzw. Hibernate macht dies auf Wunsch automatisch. Im herunterladbaren Projekt ist die DDL-Generierung konfiguriert. Abschnitt 7.2 erläutert die Konfiguration der DDL-Generierung und des Datanbanksystems im Application-Server. Für das Entity `Kunde` ist die entsprechende DDL-Anweisung für HSQLDB die folgende:

```
create table Kunde (
  id integer generated by default as identity (start with 1),
  vorname varchar(30) not null,
  nachname varchar(30) not null,
  geburtsdatum date not null,
  geschlecht char(1) not null,
  kundennummer varchar(255) not null,
  passwort varchar(255) not null,
  primary key (id),
  unique (kundennummer)
)
```

Nachdem wir nun die JSF-Seite und das Entity definiert haben (ganz ohne Seam-Funktionalität), stellt sich die Frage, wie GUI und Persistenz zueinanderfinden. Seam definiert sein Komponentenmodell auf Basis von POJOs. Damit die POJOs untereinander und mit dem Container bzw. den Container-Diensten kommunizieren können, greift Seam auf das Dependency-Injection-Muster zurück. Seam verwaltet ein Netz von Abhängigkeiten (Dependencies), die bei Bedarf aufgelöst werden. Existiert ein Objekt noch nicht, so wird es erzeugt, d. h. Seam verwaltet auch die Lebenszyklen von Objekten. Erzeugte Objekte werden nicht nur zur Auflösung einer Abhängigkeit verwendet, sondern können auch an Seam zurückgegeben werden. Sie lassen sich dann von dritten Objekten verwenden. Die Abhängigkeiten werden also in beide Richtungen

injiziert, so dass die Seam-Entwickler in Anlehnung an Dependency-Injection den Begriff *Dependency-Bijection* oder kurz *Bijection* geprägt haben. Als weiteres Kunstwort wurde als Umkehrung von Injection die *Outjection* kreiert.

In unserem Beispiel verwendet die JSF-Seite die Komponente `kunde` in mehreren JSF-EL-Ausdrücken, z. B. `"#{kunde.vorname}"` in Zeile 26 in Listing 2.3. Die Komponente `kunde` wird durch die `@Name`-Annotation des Kunden-Entities (Zeile 3 in Listing 2.5) definiert. Seam besitzt die Möglichkeit, bestimmte Komponenten initial zu erzeugen. In unserem Fall ist dies jedoch nicht sinnvoll, da Seam nicht wissen kann, welchen Kunden es erzeugen soll. Um das Bild abzuschließen, benötigen wir noch eine Login-Seite, die es einem Kunden ermöglicht, sich zu authentifizieren, und die damit bestimmt, um welchen Kunden es sich handelt. Um die programmatische Erzeugung einer Seam-Komponente möglichst einfach zu demonstrieren, werden wir eine einfache und nicht sichere Möglichkeit der Authentifizierung realisieren. In Abschnitt 4.1 gehen wir dann auf die korrekte Authentifizierung mit Hilfe einer spezialisierten Seam-Komponente ein.

Den zur Anmeldung relevanten Teil der verwendeten JSF-Seite `login.xhtml` zeigt Listing 2.6.

Listing 2.6: Ausschnitt der JSF-Seite `login.xhtml`

```
1  <h:form id="anmeldung">
2    <div class="usrFormEntry">
3      <h:outputLabel for="kundennummer" styleClass="usrFormLabel">
4          Kundennummer:
5      </h:outputLabel>
6      <h:inputText id="kundennummer"
7                   value="#{loginHandler.kundennummer}"
8                   required="true" styleClass="usrFormValue" />
9    </div>
10   <div class="usrFormEntry">
11     <h:outputLabel for="passwort" styleClass="usrFormLabel">
12         Passwort:</h:outputLabel>
13     <h:inputSecret id="passwort" value="#{loginHandler.passwort}"
14                   required="true" styleClass="usrFormValue" />
15   </div>
16   <div class="usrFormEntry">
17     <h:commandButton action="#{loginHandler.login}"
18         value="Anmelden" styleClass="usrFormSubmit" />
19   </div>
20 </h:form>
```

Man erkennt die in JSF übliche Verwendung von Komponentennamen in EL-Ausdrücken, etwa `"#{loginHandler.kundennummer}"` in Zeile 7 und zur Angabe von Action-Methoden in Zeile 17. Die Komponente mit Namen `loginHandler` ist eine zustandslose Session-Bean, deren Quell-Code in Listing 2.7 dargestellt ist. Durch die `@Name`-Annotation in Zeile 3 wird auch hier eine Seam-Komponente definiert, die den Namen `loginHandler` erhält. Im Unterschied zu einer Entity-Bean kann eine zustandslose Session-Bean erzeugt werden, ohne weitere Informationen verwenden zu müssen.

Listing 2.7: Die zustandslose Session-Bean `LoginHandler`

```
1  @Stateless
2  @Scope(SESSION)
3  @Name("loginHandler")
4  public class LoginHandler implements LoginHandlerLocal {
5
6      @Logger
7      private Log log;
8
9      @PersistenceContext
10     private EntityManager em;
11
12     @Out
13     private Kunde kunde = new Kunde();
14
15     private String kundennummer;
16     private String passwort;
17
18     public String login() {
19         log.info("login() aufgerufen");
20         Query query = em.createQuery(
21                 "select k from Kunde k "
22                 + "where k.kundennummer = :kundennummer "
23                 + "and k.passwort = :passwort");
24         List l = query.setParameter("kundennummer", kundennummer)
25                      .setParameter("passwort", passwort)
26                      .getResultList();
27         if (l.size() == 1) {
28             kunde = (Kunde) l.get(0);
29             return "/stammdaten.xhtml";
30         } else {
31             FacesMessages.instance()
32                .add("Falsche Kundennummer/Passwort-Kombination");
33             return null;
34         }
35     }
36     ...
```

Zur Laufzeit versucht Seam bei der Anfrage der JSF-Seite `login.xhtml` die Variable `loginHandler` aufzulösen. Bei der ersten Anfrage dieser Art existiert `loginHandler` noch nicht, so dass Seam eine Instanz von `LoginHandler` erzeugt und unter dem Namen `loginHandler` registriert. In Zeile 12 wird mit der `@Out`-Annotation das Property `kunde` als Seam-Komponente definiert, die von der Session-Bean zu erzeugen und an das Seam-Laufzeitsystem zurückzugeben ist. Standardmäßig wird als Seam-Komponentenname der Name des Properties angenommen. Da nach dem erstmaligen Rendern der JSF-Seite noch kein sinnvoller Kunde existiert, erzeugen wir in Zeile 13 einen Kunden mit dem Default-Konstruktor, der dann nach dem ersten Request an Seam zurückgegeben wird. Die Annotation `@Out` soll an *Outjection* erinnern. Die entsprechende Annotation für Injection heißt `@In`.

Das Verhalten zur Erzeugung eines Kunden ändert sich, wenn die JSF-Seite nicht durch Eingabe des URLs, sondern durch Betätigung der Schaltfläche aufgerufen wird. Es erfolgt dann der Aufruf der Action-Methode `login()`. In der ersten Anweisung der Methode in Zeile 19 sehen wir, wie einfach die Ausgabe einer Logging-Nachricht ist. Grundlage ist die Injektion der `log`-Variable über die `@Logger`-Annotation in den Zeilen 6/7. In Zeile 20 wird auf einen JPA-Entity-Manager zugegriffen, der ebenfalls als Seam-Komponente injiziert wird, und zwar in den Zeilen 9/10 mit der `@PersistenceContext`-Annotation. Die erzeugte Datenbankanfrage ist eine einfache parametrisierte JPA-Query, die leicht zu verstehen ist. Die Parametrisierung erfolgt über die beiden Properties `kundennummer` und `passwort`, die über die JSF-Seite mit Werten versehen werden. Falls Kundennummer und Passwort übereinstimmen, wird der entsprechende Kunde der Variablen `kunde` zugewiesen und am Ende des Requests auf Grund der `@Out`-Annotation an Seam zurückgegeben.

Seam erweitert das von JSF zur Navigation zwischen JSF-Seiten vorgesehene Konzept der Konfiguration der Navigation in der Datei `faces-config.xml`. Navigationsregeln können wie von JSF vorgesehen in dieser Datei definiert werden, es ist aber auch eine direkte Codierung als Rückgabewert einer Action-Methode möglich. In Zeile 29 wird Seam angewiesen, nach geglückter Ausführung der Action-Methode zur Seite `/stammdaten.xhtml` zu wechseln. Falls die Authentifizierung nicht glückt, wird eine Fehlermeldung generiert und `null` zurückgegeben, was zur Darstellung derselben Seite, dem JSF-Default-Fall, führt.

Als letzter Teil des Beispiels fehlt noch die in der Stammdatenverwaltung verwendete Session-Bean names `stammdatenHandler` (Zeile 52 in Listing 2.3). Die Bean ist in Listing 2.8 dargestellt.

Listing 2.8: Die zustandsbehaftete Session-Bean `StammdatenHandler`

```
1  @Stateful
2  @Scope(SESSION)
3  @Name("stammdatenHandler")
4  public class StammdatenHandler
5         implements StammdatenHandlerLocal {
6
7      @PersistenceContext(type=EXTENDED)
8      private EntityManager em;
9
10     @In @Out
11     private Kunde kunde;
12
13
14     public String speichern() {
15         em.merge(kunde);
16         return null;
17     }
18
19     @Destroy @Remove
20     public void destroy() {}
21 }
```

Wir erkennen die Deklaration als zustandsbehaftete Session-Bean durch die `@Stateful`-Annotation und die Deklaration des Gültigkeitsbereiches Session durch die `@Scope`-Annotation. Die Verwendung einer zustandsbehafteten Bean, der Gültigkeitsbereich Session und der erweiterte Persistenzkontext sind als Gesamtpaket zu sehen. Der erweiterte Persistenzkontext ist eine mit EJB 3.0 eingeführte Erweiterung des Persistenzkontextes, so dass dieser länger als eine Transaktion dauern kann. Bei der `merge()`-Methode in Zeile 15 muss daher kein erneuter Datenbankzugriff erfolgen, was zu einem Performanzgewinn führt. Die Verwendung des erweiterten Persistenzkontexts ist nur bei zustandsbehafteten, nicht aber bei zustandslosen Session-Beans erlaubt.

Die Variable `kunde` ist sowohl mit `@In` als auch mit `@Out` annotiert, da die Weitergabe in beide Richtungen erfolgt. Seam fordert, dass alle zustandsbehafteten Session-Beans eine Methode besitzen, die mit `@Destroy` und `@Remove` annotiert sind. Die entsprechende Methode kann genutzt werden, um verschiedene Aufräumarbeiten zu realisieren, die am Ende des Lebenszyklus der Session-Bean benötigt werden. `@Remove` ist eine EJB-Annotation und teilt dem EJB-Container mit, dass eine zustandsbehaftete Session-Bean inklusive ihres Zustands gelöscht werden soll. `@Destroy` ist eine Seam-Annotation, die die Methode ebenfalls als „Aufräummethode“ kennzeichnet.

Die beiden Session-Beans benötigen nach EJB-Spezifikation noch die entsprechenden Local-Interfaces, wie in Abschnitt 2.2 erläutert. Diese umfassen aber lediglich die öffentlichen Methoden der Beans, so dass wir auf eine gesonderte Darstellung verzichten können.

Wir beenden hiermit zunächst unsere Beispielanwendung. In diesem Beispiel haben wir vordefinierte Seam-Komponenten, den Logger und den Entity-Manager sowie selbst definierte Seam-Komponenten, zwei Session- und eine Entity-Bean, kennengelernt. Die beiden erstgenannten werden von Seam automatisch beim Systemstart erzeugt, die drei letztgenannten zwar ebenfalls automatisch erzeugt, aber nur bei Bedarf. Allen Komponenten ist gemeinsam, dass sie durch ein Beziehungsgeflecht basierend auf Bijection verbunden sind und gegenseitig aufeinander zugreifen können. Der zu implementierende Programm-Code hat einen sehr geringen Umfang und reduziert sich im Beispiel auf das Business-Modell eines Kunden und zwei einfache Session-Beans. Der Leser möge sich bitte überlegen, wie viel Code für eine rein JSF- und Hibernate-basierte Lösung oder alternativ mit einem anderen, dem Leser bekannten Web-Framework nötig wäre.

Fast alle Beispiele des Buches stammen aus dem Bankenbereich. Wir werden im nächsten Kapitel an einem etwas außerhalb stehenden Beispiel darstellen, wie Seam Zustände verwaltet und Konversationen und Transaktionen realisiert. In Kapitel 4 kehren wir wieder zum Online-Banking zurück und vervollständigen es mit einer Authentifizierungs- und Autorisierungskomponente sowie der Möglichkeit, PDF-Dokumente zu erzeugen und E-Mails zu verschicken.