Kapitel 1
Überblick

»Die Neugier steht immer an erster Stelle eines Problems,
das gelöst werden will.«
Galileo Galilei (1564–1642)

In diesem Kapitel gebe ich Ihnen einen Überblick über den Java EE 8-Standard und die Java EE 8-Spezifikation. Dabei erfahren Sie,

- welche Technologien der Java EE 8-Standard definiert,
- in welchen Anwendungsszenarien sie eingesetzt werden und
- wie man die Softwarearchitektur von Java EE 8-Anwendungen konzipiert.

1.1 Einführung

Am 18. September 2017 erschien der Java EE-Standard in der Version 8. Mit der Veröffentlichung von Java EE 8 stellte sich für die IT-Welt erneut die Frage, was die Java EE Expert Group verbessert und welche Probleme sie mit dem aktualisierten Industriestandard gelöst hatte. Die besondere Herausforderung wird jährlich auf IT-Messen und in IT-Magazinen sichtbar, denn fortschreitend etablieren sich mobile Endgeräte und Single-Page-Anwendungen als unverzichtbare Bestandteile einer modernen Webanwendung. Dementsprechend baut ein großer Teil aller geschäftskritischen Unternehmensanwendungen seine Clients mittlerweile mit Angular 2 auf, das unter der Haube TypeScript und somit wiederum JavaScript laufen lässt. Und selbst im geschäftskritischen Umfeld kommt man nicht mehr umhin, diese Evolution gleichermaßen ins Auge zu fassen.

Und stetig drehen sich die Uhren der IT-Welt mit erhöhter Geschwindigkeit weiter. Immer deutlicher zeigt sich, dass nicht nur Menschen mit einem Server kommunizieren, sondern auch Dinge wie Geldautomaten, Fahrzeuge, Ausweiskarten, Türschlösser, Biometrie-Leser oder Herzschrittmacher. Der Fachbegriff hierfür lautet *Internet of Things* (IoT), denn es soll überhaupt keine Rolle mehr spielen, welche Technologien in den Endgeräten zum Einsatz kommen. Der Gedanke ist nicht neu, denn bereits im Jahre 1996 wurde er unter dem Begriff *Service-Oriented Architecture* (*SOA*) vom Marktforschungsunternehmen *Gartner* beschrieben. Schon damals hatte man die Vorstellung, dass zukünftig Geschäftsprozesse in einem weltweiten Verbund miteinander kommunizieren. Dabei sollte es sich um Internetdienste

handeln, die ganz unabhängig voneinander Dateninhalte und Dienste anbieten und konsumieren. Dieser Vision sind Webservices sehr nahegekommen, denn mit ihrer Hilfe findet die Kommunikation ganz unabhängig von der verwendeten Programmiersprache oder dem verwendeten Betriebssystem statt. Auf diese Weise wird plattformübergreifend eine lose Kopplung ermöglicht. Ob nun also Handy oder Getränkeautomat, spielt keine Rolle mehr. Immer üppiger wurde das Angebot mit Echtzeitverarbeitung von Industrie, Wirtschaft und Bankenwelt. Und auch Behörden haben sich mit E-Gouvernement und Open Gouvernement längst zum Ziel gesetzt, Informations- und Verwaltungsdienste über das Web zugänglich zu machen. Die Vorteile liegen auf der Hand, denn eine digitale Prozessautomatisierung mit zentraler Datenhaltung ist wesentlich effektiver als die altbewährten Kartonregister vergangener Tage.

Eine weitere Herausforderung entspringt der Welt des Onlinestreamings, wo eine Unternehmensanwendung nicht mehr nur als großes, alleinstehendes Programm entwickelt, sondern in zahlreiche kleine Dienstanwendungen aufgeteilt wird. Der Fachbegriff hierfür lautet *Microservice-Architektur*.

Die Vorteile der Microservice-Architektur sind:

- Sie dient der besseren Skalierung von Systemen, die einen überhöhten Ressourcenbedarf erfordern.
- Funktionen können unabhängig weiterentwickelt werden, ohne immer wieder den ganzen Monolithen deployen zu müssen.
- Die Gesamtanwendung ist stabil, auch wenn ein Microservice ausfällt.
- Es besteht die Möglichkeit, jeden Microservice durch ein eigenes Entwickler-Team umzusetzen. Die Entwickler dieses Teams können sich dadurch sowohl technologisch als auch fachlich voll auf die Funktionalität dieser einen Dienstanwendung spezialisieren, um ihre Umsetzung bestmöglich zu beherrschen.
- Sie optimiert die Zugriffszeiten von Anwendungen, die wie bei Netflix Millionen von weltweiten Endgerätanfragen gleichzeitig beantworten müssen.

Bei der Microservice-Architektur wird jeder Dienst auf einem eigenen Server betrieben. Ein wichtiges Produkt für die Erstellung solcher Anwendungscontainer nennt sich *Docker*. Docker erstellt minimale Serverinstanzen und automatisiert hierbei das Anwendungs-Setup und die Konfiguration der Serverinstanz.

Der Java EE 8-Standard sollte all diese Entwicklungen berücksichtigen. Die Herausforderungen waren also sehr groß. Technologisch haben sich JSON und RESTful Webservices längst durchgesetzt, wobei der Wunsch nach einer verbesserten Multi-Channel-Kommunikation zwischen Client und Server immer deutlicher wird. Der Java EE 8-Standard unterstützt diese Entwicklung durch die Integration des neuen Protokolls HTTP/2. Ferner wird die API wich-

tiger Java EE-Technologien verbessert, indem Java SE 8-Funktionalität (Lambda Expression, Stream API und Neue Date-Time-API) und optimierte CDI-Annotationen zur Verfügung gestellt werden.

1.1.1 Die Key-Features von Java EE 8

In Abbildung 1.1 sehen Sie die Key-Features der veröffentlichten Java EE-Versionen.

1999	2001	2003	2006	2009	2013	**2017**
J2EE 1.2	J2EE 1.3	J2EE 1.4	Java EE 5	Java EE 6	Java EE 7	**Java EE 8**
Servlets	CMP	Webservices	Annotations	JAX-RS	JAX-RS 2.0	**Servlet 4.0**
JSP	JCA	JAX-RPC	EJB 3.0	CDI	JSON-P	**JSON-P 1.1**
EJB		Management	JPA	Servlet 3.0	WebSocket	**JSON-B**
JMS			JSF	Web Profile	JMS 2.0	**JAX-RS 2.1**
RMI/IIOP			JAXB	Pruning	Batch	**Security API**
			JAX-WS			

Abbildung 1.1 Die veröffentlichten Java EE-Versionen und ihre Key-Features

- **Key-Feature 1** – Servlets 4.0
  Das Besondere an der neuen Servlet-Version 4.0 ist, dass erstmalig das HTTP/2-Protokoll verwendet wird. HTTP/2 beschleunigt die Kommunikation zwischen Client und Server durch Komprimierung, Parallelität und Server-Push. Neben dem Einbau von HTTP/2 bietet die neue Servlet Technologie eine spezielle Mapping API, über die sich die Pfade zu einzelnen Elementen komfortabel ermitteln lassen.
- **Key-Feature 2** – JSON-P 1.1
  JSON-P 1.1 unterstützt die neuen Standards JSON Pointer, JSON Patch und JSON Merge Patch des IETF (Internet Engineering Task Force). Zusätzlich wurden spezielle Hilfsklassen hinzugefügt, mit denen Java SE 8-Streams verwendet werden können.
- **Key-Feature 3** – JSON-B 1.0
  Über JSON-B 1.0 lassen sich einfache Java-Klassen (*POJOs*) in JSON-Dokumente wandeln. Hierfür ist lediglich der Aufruf einer einzigen Methode vonnöten. Genauso einfach ist die Umwandlung eines JSON-Dokuments in ein POJO.
- **Key-Feature 4** – JAX-RS 2.1
  JAX-RS 2.1 unterstützt Server-sent Events, womit Daten vom Server zum Client geschickt werden können. Eine neue Reactive API erlaubt ferner, mit ReactiveX-Frameworks wie RxJava asynchron und ereignisbasiert zu interagieren, und ermöglicht Servern, bei bestehender Verbindung das Senden selbständig zu triggern.

- **Key-Feature 5** – Security API 1.0
  Die Security API 1.0 stellt eine vereinfachte Benutzerschnittstelle für die Authentifizierung und Autorisierung zur Verfügung, die sich auch für die Verwendung mit Clouds und PaaS-basierten Anwendungen eignet.

In den Key-Features wird deutlich, wie sich der Java EE-Standard in Richtung der Microservices und der Cloud-basierten Anwendungen bewegt. Denn immer mehr fokussiert der Java EE-Standard die Weiterentwicklung von RESTful Webservices und der Verarbeitung von JSON, wobei gleichzeitig die Geschwindigkeit über neue Standards wie HTTP/2 optimiert wird.

1.2 Der Java EE 8-Standard

Wenn wir die Uhren zurückdrehen zu einer Zeit, als das Internet noch in den Kinderschuhen steckte, befinden wir uns plötzlich in einer recht einfachen Umgebung. Denn anfangs bestand das Web noch aus einer Reihe von vernetzten Rechnern, die über eine spezielle Webserver-Software namens *Apache* statische Inhalte wie HTML-Seiten oder Bilder zur Verfügung stellten.

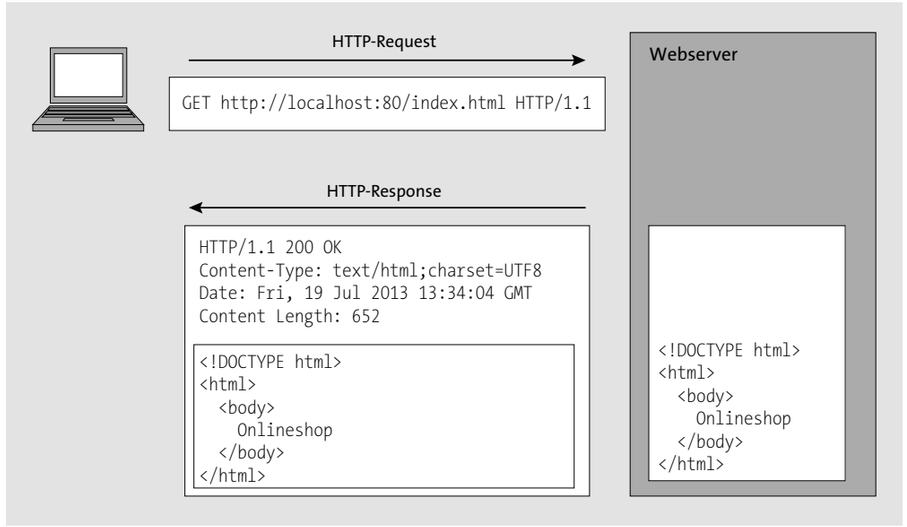

Abbildung 1.2 Die Anzeige von statischen Inhalten

Die Client-Server-Interaktion zwischen dem Webbrowser und der HTTP-Software erfolgt über das HTTP-Protokoll. Dabei löst der Benutzer HTTP-Anfragen aus, die von dem Webbrowser an die HTTP-Server-Software verschickt werden. Der HTTP-Server beantwortet die Anfrage, indem er das statische Dokument einliest und an den Webbrowser verschickt.

In den 90er-Jahren tauchte dann der Bedarf auf, Waren über das Internet zu verkaufen. Man spezifizierte das sogenannte *Common Gateway Interface* (*CGI*). CGI-Programme wurden meistens in der Programmiersprache Perl realisiert. Gleich zu Beginn nutzte man relationale Datenbanken, um die Geschäftsdaten dauerhaft zu speichern. Für den Zugriff auf die relationale Datenbank wurden die sogenannte *X/Open*- und die *Call-Level-Interface*-Spezifikation (*CLI*) definiert. Über sie konnten SQL-Anweisungen an die Datenbank gesendet werden. Abbildung 1.3 zeigt, wie ein CGI-Programm ein statisches HTML-Template verwendet, um dort den dynamischen Inhalt aus der Datenbank (den Text »Onlineshop«) einzufügen. Das Ergebnis wird schließlich per HTTP-Response an den Webbrowser verschickt.

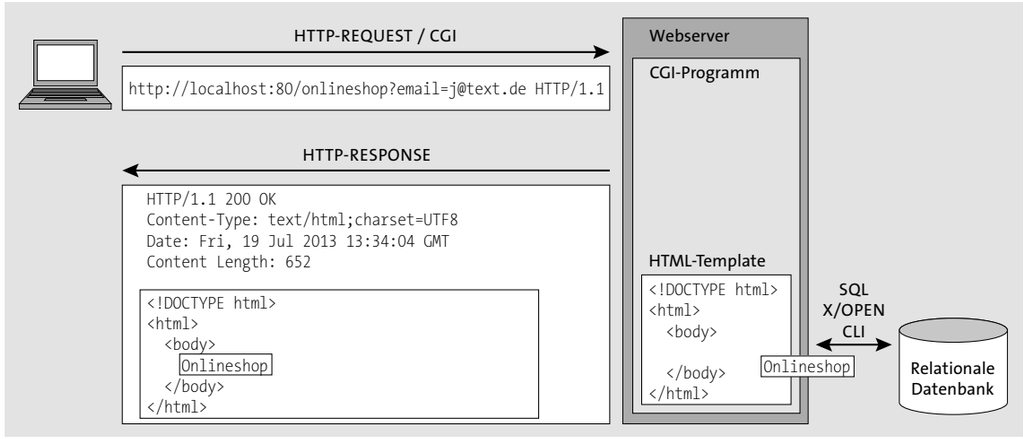

Abbildung 1.3 Eine CGI-Anwendung in der Programmiersprache Perl

Bei solch einer CGI-Anwendung tauchten Sicherheitsprobleme auf, die für eine geschäftskritische Unternehmensanwendung nicht tragbar sind. Diese Sicherheitsprobleme nennen sich Lastsicherheit, Transaktionssicherheit und Korruptionssicherheit:

- **Lastsicherheit**
  Eines der Probleme von herkömmlichen CGI-Anwendungen betrifft die Lastsicherheit, denn sie sind in der Laufzeit sehr kostspielig. Bei jeder Anfrage an den Server wird ein eigenständiges Programm gestartet. Nach der Erzeugung der Antwort wird das Programm zwar wieder beendet, aber wenn Hunderte Benutzer das Programm gleichzeitig mehrere Male hoch- und herunterfahren, kann das zum Absturz des gesamten Rechners führen.
- **Transaktionssicherheit**
  Geschäftskritische Unternehmensanwendungen transferieren sehr häufig Geldbeträge. Transaktionen von Geldbeträgen erfordern, dass ein bestimmter Betrag auf einem Datenbankserver abgezogen und zeitgleich auf einem anderen Datenbankserver hinzugefügt wird. Dieser Prozess muss in einem einzigen Schritt abgearbeitet werden, da es sich um verschiedene Speicherprozesse auf verteilten Systemen handelt. Herkömmliche CGI-

Anwendungen können bei einem Abbruch auf einem der Systeme nicht ohne weiteres gewährleisten, dass auch die Vorgänge auf den anderen Systemen rückgängig gemacht werden. Bei Reisebuchungsportalen kann es noch komplexer werden: Sollen ein Hotelzimmer, ein Mietwagen und ein Flug gleichzeitig gebucht werden, muss nicht nur der Geldtransfer, sondern auch die Hotelzimmerreservierung, die Reservierung des Autos und die Flugbuchung innerhalb der gleichen Transaktion durchgeführt werden. Schlägt eines der Systeme fehl, müssen alle Speicherungen rückgängig gemacht werden. Der gesamte Anwendungsfall muss *atomar*, das heißt in einem »unteilbaren Ganzen«, stattfinden.

- **Korruptionssicherheit**
  Beim Geldtransfer taucht auch noch das Risiko der Korruption auf. Deshalb müssen Geldangelegenheiten besonders sicher vor unbefugtem Zugriff Dritter sein. Hierfür sind spezielle Authentifizierungs- und Autorisierungsmechanismen erforderlich, denn die Identität des Benutzers ist sicherzustellen.

Seit 1999 ist die Java Enterprise Edition (Java EE) der Standard, der die Aufgabe hat, diese besonderen Sicherheitsprobleme der *geschäftskritischen Unternehmensanwendung* von Industrie, Wirtschaft und Behörden zu lösen. Das Ziel von Java EE ist also, eine Plattform anzubieten, mit denen sich hochskalierbare, zuverlässige und sichere Unternehmensanwendungen erstellen lassen.

1.2.1 Die Java EE 8-Spezifikation

Java EE war von Beginn an Teil des Java Community Process. Bei dieser von Sun Microsystems ins Leben gerufenen, freiwilligen Zusammenarbeit haben alle Mitglieder das Recht, eine Erweiterung als sogenannten *Java Specification Request* (*JSR*) einzureichen. Nimmt das Exekutivkomitee den JSR-Vorschlag an, so wird er als JSR freigegeben. Anschließend durchläuft jeder JSR einen formellen Prozess. Zunächst wird eine Expert-Group gebildet, die den jeweiligen JSR in Form eines Dokuments formuliert. Anschließend werden die Anforderungen des JSRs in Form einer Java-Bibliothek von Herstellern implementiert.

Einer der JSRs von Java EE 8, nämlich der JSR 366, nennt sich *Java Platform Enterprise Edition Specification v8*. Bei diesem JSR handelt es sich um das Rahmenwerk des Java EE 8-Standards, das die in den Standard aufgenommenen Java EE-Technologien umreißt. Das Rahmenwerk lässt sich unter folgendem Link als PDF-Dokument herunterladen:

https://www.jcp.org/en/jsr/detail?id=366

Neben den Java EE 8-Technologien definiert das Rahmenwerk das ganze Drum und Dran des Java EE 8-Servers, der das Herzstück der Java EE 8-Systemlandschaft ist. Wenn ein Hersteller bei seinem Java EE 8-Server alle Container und Dienste der Java EE 8-Spezifikation und alle Java EE 8-Technologien fehlerfrei anbietet, kann er ihn als vollständig Java EE 8-konformen Server zertifizieren lassen.

1.2.2 Aus Java EE wird Jakarta EE

Java EE war für viele Jahre Teil des standardisierten Verfahrens des JCP. Hierbei war anfangs Sun Microsystems und später Oracle nicht nur Lizenzinhaber, sondern auch an vielen JSRs als Specification Lead federführend beteiligt. Im September 2017 kündigte Oracle an, die Weiterentwicklung von Java EE der Eclipse Foundation zu übertragen. Seither haben sich alle existierenden Mitglieder der Java EE Expert Groups unter der Eclipse Foundation erneut zusammengefunden, um dort den Industriestandard weiterzuentwickeln.

Weil alle mit Java beginnenden Bezeichner der Lizenz von Oracle unterliegen, schlug Oracle vor, dass die Eclipse Foundation die Weiterentwicklung unter dem Namen *Eclipse Enterprise for Java* (kurz *EE4J*) fortführt. Mit der Zeit wurde aber klar, dass sich dieser Name nicht besonders gut vermarkten lässt. Die Working Group beschloss den Bezeichner lediglich als Top-Level-Schirmprojekt zu setzen und die neuen Java EE/EE4J-Technologien unter einem neuen Bezeichner zu spezifizieren. In einem Wahlentscheid setzte sich der Name *Jakarta EE* durch. Seitdem steht fest, dass der neue Bezeichner für die Java EE-Plattform *Jakarta EE* sein wird. Der Leiter der Eclipse Foundation, Mike Milinkovich, hat die Vorstellung, dass zukünftig die Begriffe *Jakarta EE-Technologien* und *Jakarta EE-Entwickler* fallen werden, wenn man im ursprünglichen Sinne Java EE-Technologien und Java EE-Entwickler meint.

Genauso wie sich der Bezeichner des Standards änderte, haben sich seitdem auch die Namen der einzelnen Technologien geändert, denn die heißen jetzt »Eclipse project for XXX«. Beispielsweise wird sich JAX-RS nun *Eclipse project for JAX-RS* nennen. Genauso erhalten auch entsprechende Implementierungen einer Spezifikation einen »Eclipse«-Vorsatz. Beispielsweise wird Jersey von nun an *Eclipse Jersey* heißen.

Die in der Eclipse Foundation entstehenden Jakarta EE-Technologien sollen aber trotz der Namensänderungen technisch dem aktuellen Standard Java EE 8 entsprechen. Das heißt, dass sich die Technologien unter den neuen Namen technisch nicht von den JSRs aus dem Java EE 8-Standard unterscheiden werden.

In diesem Buch werde ich statt *Jakarta EE* den Bezeichner *Java EE* verwenden, denn dies ist nach wie vor die Plattform, die die gezeigten Technologien abbildet.

1.2.3 Java EE-Profile

Manche Java EE Server beziehen sich nicht auf die vollständige Java EE-Spezifikation, sondern nur auf eine Untermenge ihrer Inhalte. Weil man auch hierfür eine Zertifizierung benötigt, wurden sogenannte *Profile* eingeführt.

Hersteller von Java EE Servern können eine Untermenge der Java EE-Technologien als eigene Spezifikation beim Java Community Process beantragen. Die unter einem neuen Namen gekennzeichnete Variante wird untersucht und gegebenenfalls als sogenanntes *Profile* freigegeben. Anschließend wird das neue Profil in einer eigenen Spezifikation festgeschrieben. Zum Beispiel existiert eine Spezifikation für das sogenannte Java EE Web Profile. Das Java EE

Web Profile enthält lediglich die Untermenge der Java EE 8-Technologien, die für die Implementierung der gebräuchlichsten Java EE-Anwendungen erforderlich sind. Für dieses Buch werden wir keinen Java EE-Web-Profile-Server, sondern einen vollständigen Java EE Server einsetzen.

1.2.4 Komponenten und Container

Java-Klassen werden in der Java EE-Fachsprache als *Komponenten* bezeichnet. Der Java EE-Standard unterscheidet vier Typen von Komponenten:

- Java-Standalone-Komponenten
- Java-Applets
- Webkomponenten (*Servlets*)
- EJB-Komponenten (*EJB*)

Während Java-Standalone-Komponenten und Java-Applets als clientseitige Komponenten angesehen werden, stellen Webkomponenten und EJB-Komponenten serverseitige Komponenten dar. Statt der Begriffe *Webkomponente* und *EJB-Komponente* aus der Spezifikation verwendet man in der Praxis eher die Begriffe *Servlet* und *EJB*. Servlets werden in Webcontainern und EJBs in EJB-Containern ausgeführt.

Die Java EE-Spezifikation legt fest, dass ein vollständig Java EE-konformer Server sowohl einen Webcontainer als auch einen EJB-Container enthalten muss.

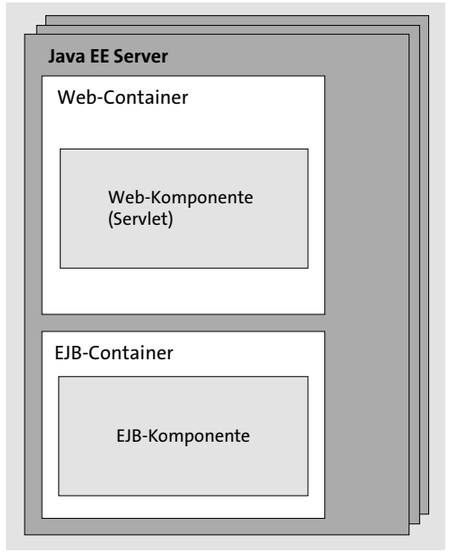

Abbildung 1.4 Die Java EE-Lösung: Die Webkomponente (Servlet) befindet sich in einem Webcontainer.

Falls Ihnen der Ausdruck »Container« vielleicht noch befremdlich erscheint: Selbst, wenn es Ihnen nicht bewusst sein sollte, Container sind Ihnen als Java-Programmierer längst vertraut. Denn auch die *Java Runtime Engine* ist ein Container. In der Java EE-Fachsprache wird hierfür der Begriff *Java Standalone Container* oder eben *Java Runtime Engine* verwendet. Die Java Runtime Engine ist der erste Container, der vom Hersteller Sun Microsystems erfunden wurde. Ein weiterer Container, der Ihnen ebenso bekannt sein könnte, ist das Applet-Plugin im Webbrowser. In der Java EE-Welt spricht man vom *Applet-Container*. Die Erfinder des Java EE-Standards sind dem Konzept von Komponenten und Containern somit treu geblieben, da sie auch serverseitig ein Servlet als Komponente in einem Container ausführen.

Der Webcontainer

Der Webcontainer nimmt den vom Webbrowser eintreffenden HTTP-Request entgegen. Aus den enthaltenen Informationen erstellt er ein Java-Objekt (das `Request`-Objekt), das er als Parameter an das zuständige Servlet weiterleitet. Der Webcontainer verfügt hierbei über einen speziellen HTTP-Dienst, der dem Entwickler der Webkomponente viel Mühe erspart. Dabei begünstigt er auch die Lastsicherheit durch drei Faktoren:

- Der erste Faktor basiert darauf, dass er nicht für jeden HTTP-Request gestartet und heruntergefahren werden muss, sondern im Standby auf eintreffende HTTP-Requests wartet.
- Der zweite Faktor beruht auf dem eingebauten Multi-Threading-Mechanismus. Hierdurch kann der Webcontainer zahlreiche gleichzeitig eintreffende HTTP-Requests entgegennehmen. Und selbst wenn etliche HTTP-Requests auf dieselbe Webkomponente zielen, ist dies für den Webcontainer kein Problem, da er mehrere Instanzen eines Servlets in einem Pool verwalten kann.
- Außerdem ist es möglich, mehrere Remote-Server im Cluster-Verbund zusammenzuschalten. Dies ist der dritte Faktor, der die Aufnahmekapazität der Java EE-Anwendung prinzipiell auf ein unbegrenztes Maß skalieren kann.

Der EJB-Container

Auch EJBs (*Enterprise JavaBeans*) werden serverseitig als Komponenten in einem speziellen Container ausgeführt, der sich EJB-Container nennt. Der EJB-Container verfügt über besondere Dienste, die die Transaktionssicherheit und die Korruptionssicherheit gewährleisten. Diese Dienste sind nach wie vor auch die Argumente, die für den Einsatz von EJBs sprechen.

1.2.5 Die Datenhaltung

Die Geschäftsdaten einer Java EE-Anwendung werden üblicherweise dauerhaft in einer relationalen Datenbank gespeichert. In der Java EE-Spezifikation wird für die dauerhafte Datenhaltung der Fachbegriff *Persistenz* verwendet. Der Zugriff auf eine relationale Datenbank erfolgt über die *Java Database Connectivity* (*JDBC*). JDBC basiert auf der Benutzerschnittstelle

SQL X/Open CLI (*Call Level-Interface*), in der die Kommunikation zwischen einer Anwendung und einer Datenbank spezifiziert ist.

Um bei verteilten Systemen auch noch Transaktionssicherheit zu gewährleisten, wird ein JDBC-Treiber verwendet, der X/Open-XA-Standard-konform ist. Dieser basiert auf der Benutzerschnittstelle *SQL X/Open XA CLI*.

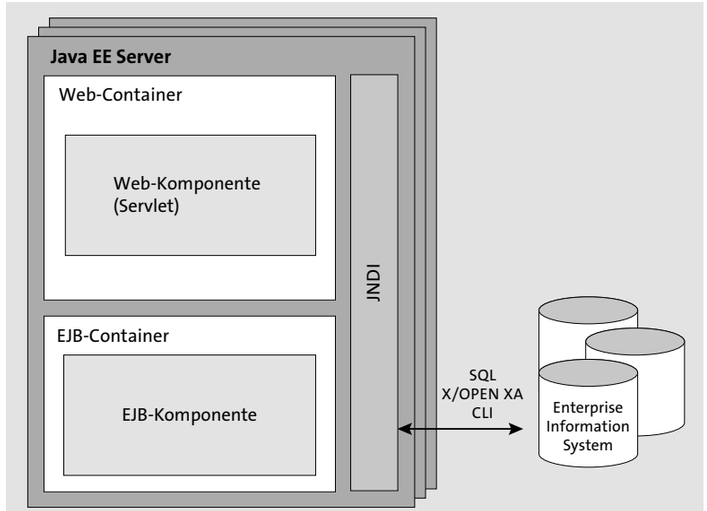

Abbildung 1.5 Der Java EE Server verwaltet die Datenbankzugriffe über SQL X/Open CLI bzw. SQL X/Open XA CLI und stellt die Daten über das Java Naming and Directory Interface (JNDI) zur Verfügung.

Abbildung 1.5 zeigt darüber hinaus, dass die Java EE-Anwendung nicht direkt auf die relationale Datenbank zugreift, denn diese Aufgabe obliegt dem Java EE Server. Der Java EE Server interagiert mit der relationalen Datenbank und stellt die Daten über das *Java Naming and Directory Interface* (*JNDI*) zur Verfügung. JNDI bietet eine einheitliche Schnittstelle für den Zugriff auf Namens- und Verzeichnisdienste. Ferner sehen Sie in der Abbildung, dass die relationale Datenbank als *Enterprise Information System* (*EIS*) bezeichnet wird. Das gesamte Konzept wurde in der Java EE-Connector-Architecture-(JCA-)Spezifikation festgeschrieben. Auf JCA komme ich weiter unten in diesem Kapitel noch einmal zurück.

1.3 Anwendungsszenarien

In diesem Abschnitt zeige ich in jeweiligen Anwendungsszenarien, wie der lokale Client auf unterschiedliche Art und Weise mit dem Java EE Server kommuniziert und dieser wiederum auf eine relationale Datenbank zugreift, um die Daten dauerhaft zu verwalten. Bitte betrachten Sie die unterschiedlichen Varianten nicht voneinander losgelöst, denn häufig ist es tatsächlich so, dass sie miteinander kombiniert werden.

1.3.1 Szenario 1: Browser ↔ Webcontainer

In Szenario 1 verwendet der Benutzer einen Webbrowser, um mit dem Server zu kommunizieren. Die Verarbeitung des HTTP-Requests erfolgt im Webcontainer eines Java EE Servers. Nachdem der Webcontainer den eintreffenden HTTP-Request verarbeitet hat, erstellt er aus den enthaltenen Informationen ein `Request`-Objekt, das er als Parameter an das zuständige Servlet weiterleitet.

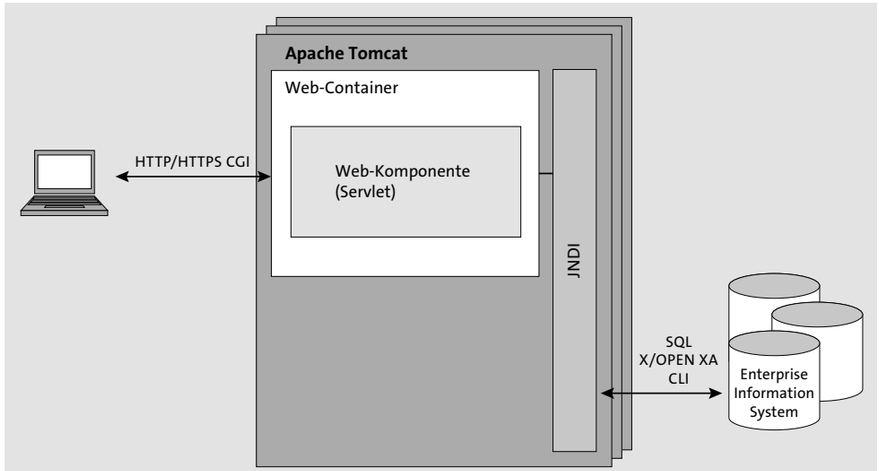

Abbildung 1.6 Die Kommunikation zwischen Webbrowser und Webcontainer in Anwendungsszenario 1

Apache Tomcat, Catalina und Grizzly

In Abbildung 1.6 habe ich keinen vollkonformen Java EE Server abgebildet, denn ein vollkonformer Java EE Server müsste nicht nur einen Webcontainer, sondern auch einen EJB-Container enthalten. Stattdessen habe ich ein einfacheres Produkt dargestellt, das lediglich einen Webcontainer bereitstellt. Diese einfache Variante wird in der Regel mit dem Produkt *Tomcat* der Apache Software Foundation umgesetzt. Tomcat integriert den gängigsten Webcontainer. Er nennt sich *Catalina*. Auch GlassFish enthält ein Catalina-Derivat. Die Catalina-Variante von GlassFish wurde allerdings durch die interne Nutzung der NIO API verbessert und in *Grizzly* umgetauft.

1.3.2 Szenario 2: beliebiger Client ↔ EJB-Container (RMI-IIOP oder JMS)

Eines der Besonderheiten von EJB-Containern ist, dass Sie von Beginn an, genau wie Webcontainer eine direkte Kommunikation mit der Außenwelt anboten. Ursprünglich fand diese Kommunikation lediglich über Session Beans statt, die plattformunabhängig über **RMI-IIOP** angesprochen werden konnten. RMI-IIOP steht für *Java Remote Method Invocation (RMI) interface over the Internet Inter-Orb Protocol (IIOP)*. Um RMI-IIOP von Grund auf zu erklären, begeben wir uns zurück in die 90er-Jahre, denn damals hatte die Informationstechnologie das ehrgeizige Ziel, verteilte Anwendungen zu bauen, die sich über die technologischen Grenzen von heterogenen Systemlandschaften und unterschiedlichen Programmiersprachen hinwegsetzen sollten. Der wichtigste Standard, der für diesen Zweck entstand, nannte sich *CORBA*. CORBA ist die Abkürzung für *Common Object Request Broker Architecture*. Es handelte sich dabei um eine Middleware-Technologie, die mithilfe eines sogenannten *Object Request Brokers* (*ORB*) eine programmiersprachenunabhängige Interaktion zwischen Client und Server ermöglicht. Der ORB ist für die eigentliche Übertragung der Informationen verantwortlich. Wenn beispielsweise eine serverseitige Komponente eine Methode bereitstellen möchte, wird diese bei dem ORB registriert. Der Client kann den ORB anschließend ansprechen, um die registrierte Methode aufzurufen. Weil der ORB in der Mitte als Vermittler erforderlich ist, spricht man auch von einer *Middleware-Technologie*. Die Programmierung der CORBA-eigenen Sprache war aber recht komplex und fehlerträchtig. Daher suchten die IT-Unternehmen nach weiteren Lösungen für die plattformunabhängige Kommunikation. Sun Microsystems hatte unterdessen eine eigene Middleware-Technologie erschaffen, die sich *Remote Method Invocation* (*RMI*) nannte. Auch bei RMI findet die Kommunikation statt, indem eine entfernte Client-Anwendung die Ausführung einer serverseitigen Methode auslöst. Der Begriff aus dem Fachjargon hierfür lautet *Remote Procedure Call* (*RPC*). Um einen RPC zu ermöglichen, wird auch bei RMI ein *Object Request Broker* (*ORB*) benötigt. Beim Java EE-Standard stellt der Java EE Server den ORB dar.

Das Besondere an RMI ist, dass neben einfachen Daten auch serialisierbare Java-Objekte transportiert werden können. Das Übertragungsprotokoll nennt sich *Java Remote Method Protocol* (*JRMP*). Bei einem RMI-Programm werden *Stubs* (Interfaces mit Methodenrümpfen) und *Skeletons* (Klassen mit gleichnamigen Methoden) erstellt. Ruft ein Client eine Methode des Stubs beim ORB auf, delegiert der ORB diesen Aufruf an das serverseitige Objekt, bei dem anschließend die entsprechende Methode aktiviert wird. Der Einsatz von RMI mit dem Transportprotokoll RMI-JRMP setzt voraus, dass sowohl der Client als auch der Server auf der Java-Plattform basieren. Daher musste für Session Beans eine andere Middleware-Technologie eingebaut werden, die auch von Java-fremden Technologien verstanden wird. Aus diesem Grund wurde der damalige Standard für plattformunabhängige Kommunikation, nämlich CORBA, hinzugezogen. Das neue Transportprotokoll nannte man RMI-IIOP. Das Außergewöhnliche hierbei ist aber nicht nur, dass Session Beans über RMI-IIOP plattform- und programmiersprachenunabhängig kommunizieren können, sondern dass Transaktionen automatisch über die verteilten Komponenten hinweg koordiniert werden. Der EJB-Container verwaltet hierbei die EJB-Komponenten und kümmert sich ebenso um diverse Systemdienste, wie beispielsweise den Transaktionsdienst und den Security-Dienst. Aber obwohl RMI-IIOP hiermit ein Bestandteil eines ausgereiften Komponentenmodells ist, setzt sich schon längst eine andere Systemarchitektur durch, die die Plattformunabhängigkeit über Webservices gewährleistet, weshalb die Java EE Expert Group einerseits RMI-IIOP für Java EE 8-Server fest verankert, andererseits vorschlägt, bei künftigen Java EE-Spezifikationen RMI-IIOP als optional zu kennzeichnen, sodass kommende Java EE Server RMI-IIOP nicht mehr anbieten müssen.

Neben RMI-IIOP bietet die EJB-Komponentenarchitektur eine asynchrone, nachrichtenbasierte Kommunikation an. Für diesen Zweck sind keine Session Beans, sondern sogenannte *Message-driven Beans* im Gebrauch. Message-driven Beans verwenden eine standardisierte Benutzerschnittstelle mit dem Namen JMS API. Hierbei handelt es sich also nicht um ein festgelegtes Protokoll, sondern um Schnittstellendefinitionen, die von einem JMS Middleware Provider implementiert werden.

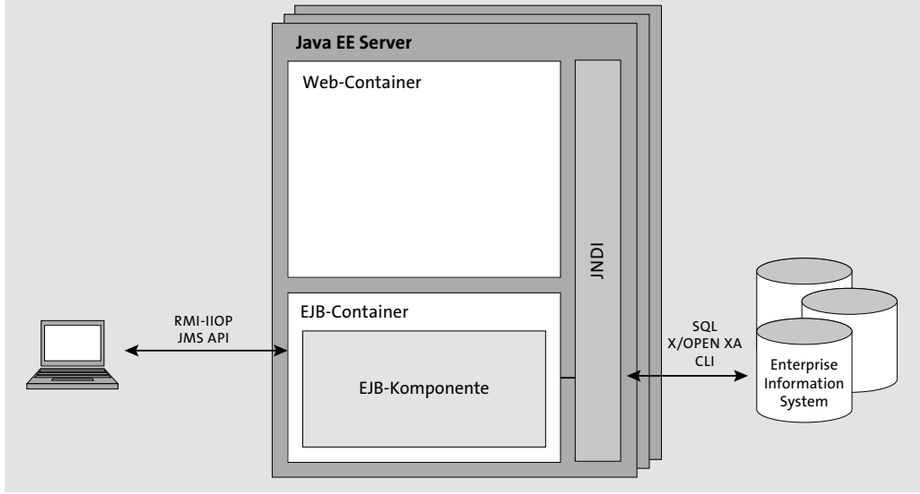

Abbildung 1.7 Die Kommunikation über RMI-IIOP oder die JMS API in Anwendungsszenario 2

1.3.3 Szenario 3: Browser ↔ Webcontainer ↔ EJB-Container

Abbildung 1.8 zeigt ein weiteres Szenario. Darin verwendet die Java EE-Anwendung sowohl Web- als auch EJB-Container. Während die Kommunikation mit dem Webbrowser des Benutzers über den Webcontainer erfolgt, kümmert sich der EJB-Container um die Interaktion mit der Datenbank. Bei diesem Szenario kommunizieren die Webkomponenten mit den EJB-

Komponenten innerhalb der gleichen Anwendung. Die Webkomponente nutzt hierfür die *Context and Dependency Injection for Java* (*CDI*). Weil die Webkomponente in der gleichen Java-Laufzeitumgebung ausgeführt wird wie die EJB-Komponente, bezeichnet man den Zugriff als *lokal*. Der lokale Zugriff wird vom Container intern über eine Referenz auf die Speicheradresse der EJB-Komponente bewerkstelligt.

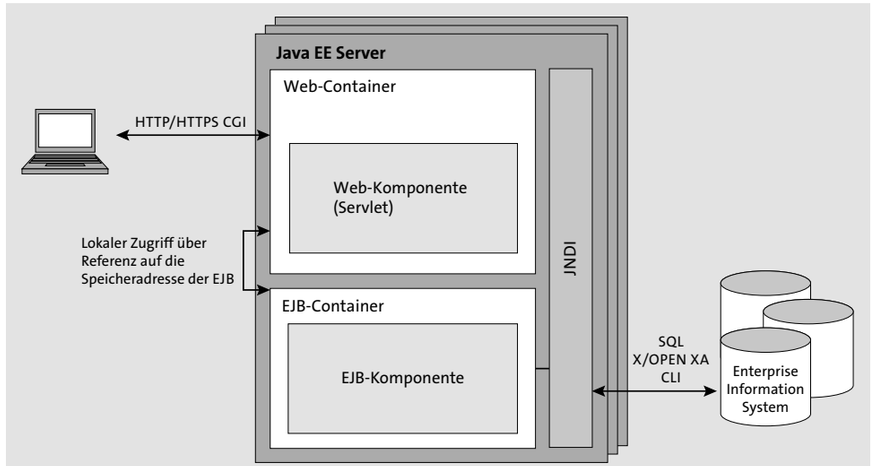

Abbildung 1.8 Die Client kommuniziert mit der Webkomponente. Die Webkomponente kommuniziert wiederum mit einer EJB-Komponente, um die Transaktionssicherheit und die Korruptionssicherheit zu gewährleisten.

1.3.4 Szenario 4: beliebiger Client ↔ Webcontainer ↔ EJB-Container (SOAP oder REST)

Mit Anwendungsszenario 4 kommen wir auf Transportprotokolle zu sprechen, die sich immer mehr zum De-facto-Standard etablieren, denn das Besondere an diesem Anwendungsszenario ist, dass die Komponenten der Java EE-Anwendung über die Transportprotokolle SOAP und REST mit der Außenwelt kommunizieren.

Aber fangen wir ganz von vorne an, denn auch bei dieser Erläuterung werden wir die Uhren wieder zurück in die 90er-Jahre drehen. Damals entstand der Oberbegriff *Service-Oriented Architecture* (*SOA*). Der Begriff SOA wurde erstmalig 1996 vom Marktforschungsunternehmen *Gartner* eingesetzt. Die *Service-Oriented Architecture* beschreibt eine Vision, in der einzelne Geschäftsprozesse in einem weltweiten Komplex voneinander entkoppelten Internetdiensten verwirklicht werden. Die Abhängigkeiten innerhalb des Internets werden dabei so minimiert, dass technische und fachliche Änderungen fast keine Anpassungserfordernisse von einzelnen Clients oder Servern nach sich ziehen. Einzige Voraussetzung ist die Einhaltung von vereinbarten Kommunikationsschnittstellen.

SOA basiert also auf dem Grundgedanken, einen globalen Verbund von lose gekoppelten Systemen zu realisieren. Oder anders gesagt: Im Sinne von SOA ist ein Webservice eine Anwendung, die konsumierbare Dateninhalte ganz unabhängig von der verwendeten Programmiersprache oder dem verwendeten Betriebssystem anbietet. Auf diese Weise wird plattformübergreifend eine lose Kopplung ermöglicht.

Hinter den Webservices stehen Anwendungen, die Informationen zum Beispiel mit XML-Syntax anbieten. Der Anbietende wird *Service Provider* genannt und der Konsumierende *Service Consumer*. Zusätzlich kommen genau wie bei CORBA wieder Object Request Broker (ORBs) ins Spiel, bei denen Services registriert werden müssen.

Heutzutage findet die Kommunikation mit einem Java EE Server immer häufiger über einen Webservice statt, denn immer mehr Clients werden ausschließlich mit JavaScript geschrieben, die mit dem Server über Webservices interagieren. Ferner besteht durch Webservices die Möglichkeit, dass eine Maschine als Client im Einsatz ist. Einzige Voraussetzung ist die Einhaltung von vereinbarten Kommunikationsschnittstellen, wie beispielsweise SOAP oder REST.

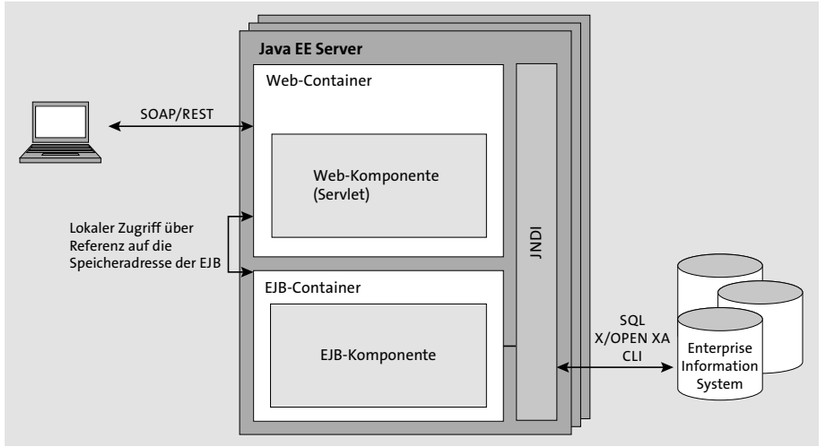

Abbildung 1.9 Anwendungsszenario 4 mit Webservices

1.3.5 Szenario 5: beliebiger Client ↔ EJB-Container (SOAP oder REST)

Anwendungsszenario 5 zeigt, dass sich EJB-Komponenten, genauso wie Webkomponenten, über SOAP und REST ansprechen lassen.

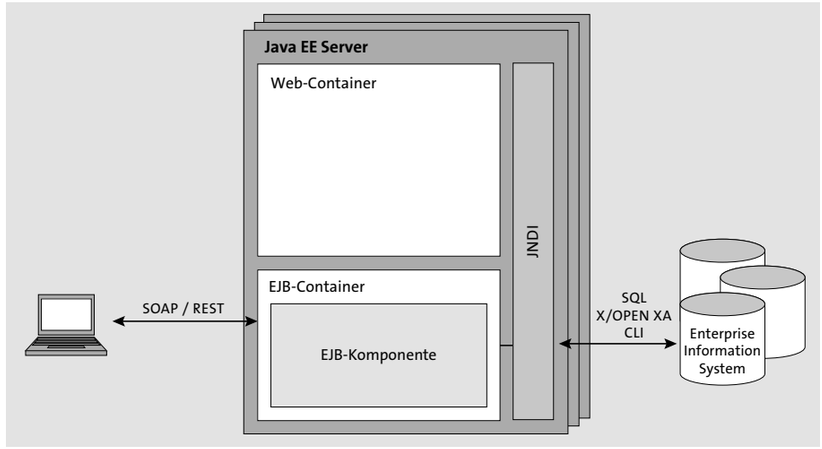

Abbildung 1.10 Nicht nur Webkomponenten, sondern auch EJB-Komponenten lassen sich als Webservices ausbauen.

1.4 Der mehrschichtige Aufbau einer Java EE-Anwendung

In der Praxis ist eine Java EE-Anwendung aus sehr vielen Komponenten aufgebaut. Dabei handelt es sich meistens um eine sehr umfangreiche und komplexe Software. Je größer die Software ist, desto mächtiger wird der Wunsch nach einer sauberen Struktur. In Java EE-Anwendungen wird eine Ordnung erzielt, indem die Komponenten in einer mehrschichtigen Architektur aufgeteilt werden. Darüber hinaus können in einer Java EE-Software unterschiedliche Entwurfsmuster zum Einsatz kommen. Das gängigste Entwurfsmuster beim Frontend einer Java EE-Anwendung nennt sich *Model-View-Controller* oder auch einfach *MVC-Entwurfsmuster*.

Das MVC-Entwurfsmuster trennt die Komponenten einer Java EE-Anwendung in Anwendungsdaten (*Model*), Präsentation der Daten (*View*) und Anwendungslogik (*Controller*). Das Model speichert den Anwendungszustand, der im Controller geändert und in der View dargestellt wird. Dabei werden die Anwendungsdaten als JavaBeans zwischen der View und dem Controller hin- und hergereicht.

Ich habe bereits angemerkt, dass die Java EE-Komponenten typischerweise in mehrere Schichten sortiert sind. Dabei betrachtet man die Ordnung der Schichten gemäß dem Anfrageprozess. Das bedeutet, dass die View als Präsentationsschicht gesehen wird. Der Controller übernimmt darunter die Steuerungsschicht. Die Datenhaltung wird ganz unten angesiedelt und Persistenzschicht genannt.

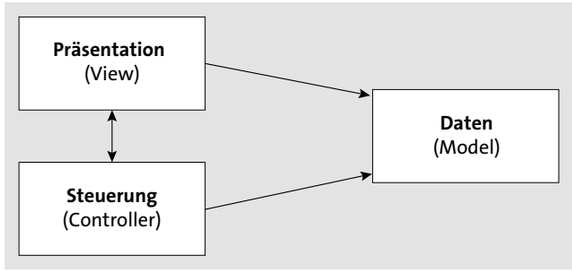

Abbildung 1.11 Das MVC-Entwurfsmuster

Die Schichten kommunizieren miteinander über vereinbarte Schnittstellen. Jede Schicht kennt nur die Schnittstelle der direkt darunterliegenden Schicht. Durch eine saubere Trennung kann eine Schicht komplett ausgewechselt werden. Solange die vereinbarten Schnittstellen vorhanden sind, muss der Quelltext der anderen Schichten nicht angepasst werden. Ein anderer Vorteil der Schichtentrennung ist, dass unterschiedliche Experten an den einzelnen Schichten arbeiten können. Meistens werden für jeden Bereich mehrere Mitarbeiter in Teams gruppiert.

Als Beispiel betrachten wir eine gängige Aufteilung für Anwendungsszenario 1. In Abbildung 1.12 sehen Sie, wie hierbei die Low-Level-Technologien Servlets und JSPs im Frontend zum Einsatz kommen.

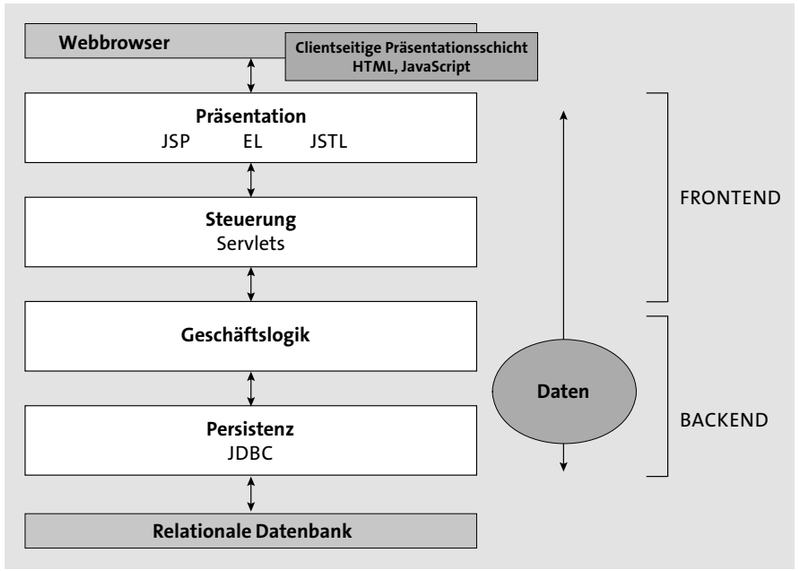

Abbildung 1.12 Die Schichten, die bei einem persistierenden Request durchlaufen werden

Im Backend wird die Geschäftslogik eingebaut. Darunter werden Java-Klassen programmiert, die für die Speicherung der Geschäftsdaten über JDBC zuständig sind. Der englische Fachbegriff lautet *Data Access Objects* (*DAO*). Beachten Sie auch noch, wie die Daten in Form von ganz einfachen Java-Klassen zwischen Frontend und Backend hin- und hergereicht werden. Solche einfachen Java-Klassen werden auch *Plain Old Java Objects* (*POJOs*) oder *JavaBeans* genannt.

Die gleiche Softwarearchitektur können wir (wie in Abbildung 1.13 gezeigt) auch mit den High-Level-Technologien JSF, EJB und JPA verwirklichen.

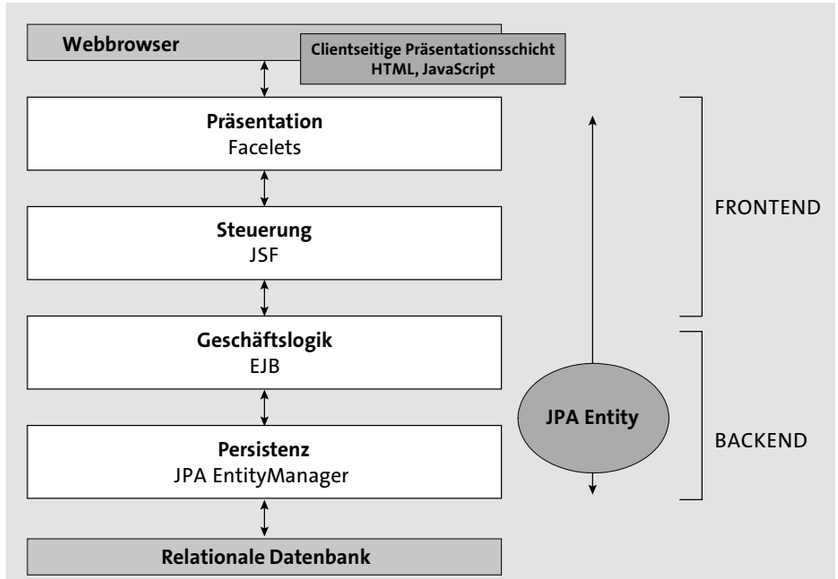

Abbildung 1.13 High-Level-Technologien innerhalb der mehrschichtigen Java EE-Softwarearchitektur

Im Folgenden gehe ich ausführlicher auf die einzelnen Schichten ein.

1.4.1 Schicht 1: Präsentation

Die oberste Schicht bezeichnet man als *Präsentationsschicht* oder als *View*. Weil die Benutzerschnittstelle einer Java EE-Anwendung üblicherweise webbasiert ist, lässt sich diese Schicht in eine Client- und eine Server-Seite aufteilen. Die clientseitige Präsentationsschicht befindet sich physikalisch im Webbrowser. In den Anwendungsszenarien 1 und 3 wird sie als dynamisch erzeugte HTML-Seite innerhalb eines Webbrowsers angezeigt. Über diese

Ansicht kann der Benutzer die Daten in einer Webpage betrachten und mit der Anwendung interagieren. Gleichzeitig kann die Webpage JavaScript enthalten, sodass die Client-Seite aktiv über Ajax mit dem Server kommunizieren kann. In der Literatur von Oracle wird die clientseitige Präsentationsschicht manchmal auch als Client-Layer bezeichnet. Die serverseitige Präsentationsschicht besteht aus den Webkomponenten, mit denen der Entwickler die clientseitige Präsentationsschicht entwirft. Mit Low-Level-Technologien wären das Java Server Pages (JSPs). Mit High-Level-Technologien kämen an dieser Stelle vorzugsweise sogenannte *Facelets* zum Einsatz.

1.4.2 Schicht 2: Steuerung

Die zweite Schicht nimmt die Anfragen des HTTP-Clients entgegen. Darüber hinaus ist sie für die Steuerung im Programmablauf zuständig. Gleichzeitig leitet sie die Anfrage an die tieferen Schichten weiter. Ohne JSF-Framework müssten Sie an dieser Stelle Servlets entwickeln. Wenn hingegen das JSF-Framework zum Einsatz kommt, nimmt das Framework die HTTP-Anfrage entgegen. Im Programm müssen Sie nur die Steuerung vorsehen. Hierfür werden spezielle *Backing Beans* programmiert. Im Prinzip handelt es sich hierbei um JavaBeans, die durch das CDI-Framework vom Webcontainer verwaltet werden.

1.4.3 Schicht 3: Geschäftslogik

In der dritten Schicht wird die Geschäftslogik abgebildet. Diese Schicht kann durch den Einsatz von *Enterprise JavaBeans* realisiert sein. Dies kommt auf das eingesetzte Szenario an. Eine weitere Bezeichnung für diese Schicht ist *Business-Schicht* oder *Business-Layer*. Werden keine EJBs verwendet, spricht man manchmal auch von einer *Service-Schicht*.

1.4.4 Schicht 4: Persistenz

Die Persistenz-Schicht besteht aus den Komponenten, die aus der Anwendung heraus für die dauerhafte Datenhaltung zuständig sind. Die Komponenten der Persistenz-Schicht speichern, ändern und löschen Daten, indem sie auf eine relationale Datenbank zugreifen.

Ohne die JPA werden die Datenbankzugriffe mithilfe von JDBC verwirklicht. Mit High-Level-Technologien wird man die *Java Persistence API* (*JPA*) einsetzen. Das bedeutet, dass die darüber liegende Geschäftslogik-Schicht direkt den EntityManager der JPA nutzt, um die Geschäftsdaten zu persistieren.

Für den Transport der Daten werden bei den Low-Level-Technologien einfache POJOs bzw. JavaBeans verwendet. Häufig handelt es sich bei diesen Objekten um Gegenstücke zu einem Datensatz einer Datenbanktabelle. In so einem Fall spricht man auch von einer *Entity*.

Mit den High-Level-Technologien wird man die spezielleren JPA-Entities einsetzen. Im Prinzip handelt es sich hierbei ebenfalls um JavaBeans, die jedoch bestimmte Voraussetzungen erfüllen müssen.

In Abbildung 1.13 sehen Sie, dass die gleiche JavaBean durchgehend in allen Schichten als Modell bzw. als Datenbehälter verwendet wird. Allerdings ist diese einfache Vorgehensweise in großen Java EE-Anwendungen manchmal mit Vorsicht zu genießen, denn wie Sie in Kapitel 8, »Die Java Persistence API«, noch sehen werden, kann der Datensatz einer Datenbanktabelle sehr große Binärwerte (beispielsweise Bilder) enthalten. Die Entity, die als Gegenstück zu einem solchen Datensatz erzeugt wird, könnte anschließend zur Laufzeit den Arbeitsspeicher verstopfen. Erschwerend kommt hinzu, dass die relationale Abhängigkeit zwischen den Datenbanktabellen als Referenz in den JavaBean-Objekten realisiert wird. Dies kann zur Folge haben, dass bei einer einzigen Abfrage ganze Objektbäume in das Frontend geliefert werden. Um diesen Problemen entgegenzutreten, werden in der Java EE-Welt verschiedene Ansätze verfolgt. Eine Variante besteht darin, große Werte erst zu setzen, wenn sie tatsächlich gebraucht werden. Man spricht hierbei vom *Lazy Fetching*. Ein weiteres Problem entsteht aber auch hierbei durch die Art und Weise, wie das JPA-Framework arbeitet, denn die JPA kann einen Wert nur automatisiert in eine Entity setzen, wenn es im sogenannten *Managed*-Zustand ist. Aufgrund dieser Probleme muss eine gut durchdachte Strategie entworfen werden, die konsequent in der gesamten Java EE-Anwendung eingehalten wird. Eventuell muss zu jeder JPA-Entity ein sogenanntes *Data Transfer Object* (*DTO*) programmiert werden. Mithilfe eines DTO hat der Entwickler die Größe der übertragenden Daten im Griff. Eine zusätzliche DTO-Schicht wird aber nur in sehr großen Java EE-Projekten in Erwägung gezogen, weil die Wartung der zusätzlichen Schicht aufwendig und die Übertragung der Daten zwischen JPA-Entity und DTO kostspielig ist.

1.5 Die Evolution von HTTP/2 und WebSockets

Ich habe zum Anfang dieses Kapitels bereits erwähnt, dass der Java EE 8-Standard moderne Entwicklungen des Internets unterstützt, indem er die Verwendung des HTTP/2-Protokolls ermöglicht. Um die Besonderheit dieser Innovation zu verdeutlichen, reisen wir ins Jahr 1996 zurück und schauen uns die Entwicklung bis ins heutige Zeitalter an.

1.5.1 HTTP/1.0

Als 1996 Sir Timothy John Berners-Lee das HTTP-Protokoll 1.0 veröffentlichte, nahm jeder Request-Response-Zyklus zwischen dem Webbrowser und dem HTTP-Server noch eine eigene TCP-Verbindung in Anspruch. In den Webseiten waren aber immer häufiger zahlreiche Bilder eingebunden, die ebenfalls jeweils über eine eigene TCP-Verbindung besorgt werden mussten.

Abbildung 1.14 Mit HTTP 1.0 benötigte jedes Bild eine eigene TCP-Verbindung.

1.5.2 HTTP/1.1

Das Nachziehen von Bildern des Protokolls HTTP/1.0 war aufwendig. Dies war einer der Grund dafür, warum man im Jahre 1999 das Protokoll HTTP in der Version 1.1 herausgab. Mit HTTP 1.1 kann man die Verbindung offenhalten, sodass der Webbrowser die Bilder einer Webseite über die gleiche TCP-Verbindung beschafft.

Abbildung 1.15 Mit HTTP 1.1 können alle in der Webseite enthaltenen Bilder mit einer einzigen TCP-Verbindung beschafft werden.

In den nachfolgenden Jahren diente das Protokoll HTTP 1.1 stets als solide Grundlage für stabile Webanwendungen.

1.5.3 Ajax

Es war eine Erfindung der Firma Microsoft, die das Internet revolutionierte. Denn durch das `XMLHttpRequest`-Objekt lässt sich ein HTTP-Request an den Server senden, ohne dass der Webbrowser die Webseite verlässt. Der Server antwortet nun nicht mehr mit einer kompletten Webseite, sondern lediglich mit einem kleinen Fragment, das in der Webseite dynamisch eingebaut wird. Seitdem Jesse James Garrett diesen Mechanismus im Jahre 2005 *Asynchronous JavaScript And XML* (kurz *Ajax*) benannte, wurde Ajax in immer mehr Webanwendun-

gen eingebaut. Es entstanden zahllose JavaScript-Funktionen und ganze Ajax-Frameworks, die zu einem immer besseren User-Erlebnis führten. Nur leider beharrte das HTTP-Protokoll auf dem einseitigen Verhältnis zwischen einem Webbrowser und einem HTTP-Server. Das Unerfreuliche hierbei ist, dass nur der Webbrowser Anfragen stellen darf, während der HTTP-Server dazu verdonnert ist, zu antworten. Die Onlineshop-Anwendung aus diesem Buch verdeutlicht das Problem: Beispielsweise könnte man sich vorstellen, dass ein Verkäufer vor seinem Rechner sitzt und darauf wartet, dass seine Artikel verkauft werden. Ohne sein Zutun würde er vergeblich auf eine Änderung an seinem Bildschirm warten.

1.5.4 Short Polling

Grundsätzlich gibt es für dieses Problem die Möglichkeit, einen Zeitgeber in die Webseite einzubauen, der sich über einen Ajax-Request in kurzen zeitlichen Intervallen nach dem aktuellen Stand erkundigt. Der Fachbegriff für die im Intervall immer wieder versendeten Anfragen lautet *Polling* oder auch *Short Polling*, wenn die wiederkehrenden Aufrufe in sehr kurzen Abständen (wie zum Beispiel einer Sekunde) erfolgen. Allerdings sind die wiederkehrenden Aufrufe des Webbrowsers kostspielig, weil jeder Aufruf eine eigene TCP/IP-Verbindung aufbaut. Und das ist auch nicht ganz ungefährlich, denn wenn beispielsweise Tausende Kunden in jeder Sekunde einen HTTP-Request an den Server schicken, reicht das schon, um so manchen Server lahmzulegen.

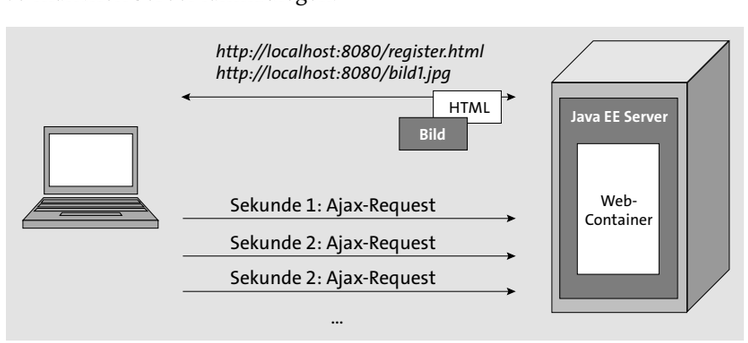

Abbildung 1.16 Das »Short Polling« fragt in zeitlich kurzen Intervallen wiederkehrend nach einem Ergebnis.

1.5.5 Long Polling

Neben dem Short Polling gibt es auch das Long Polling. Hierbei handelt es sich um eine ähnliche Technik, bei der die Kommunikation aber mit einer einzigen TCP/IP-Verbindung auskommt, denn hat der Server für den Client beim ersten Zugriff noch keine Information zur Verfügung, hält der Client die Verbindung offen. Dies gelingt ihm durch den regelmäßigen Versand eines Header-Schnipsels. Sobald die Information beim Server zur Verfügung steht, verschickt er eine abschließende Response. Durch das Long Polling kann der Eindruck einer

Push-Technologie entstehen. Mit Push-Technologie ist gemeint, dass die Informationen nicht vom Server gezogen (Pull), sondern durch den Server gedrückt (Push) werden.

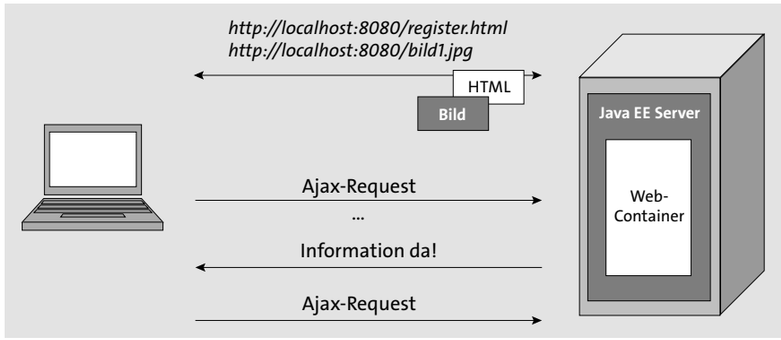

Abbildung 1.17 Beim »Long Polling« hält der Server die Verbindung offen, bis ihm die angefragte Information zur Verfügung steht.

Das Long Polling hat den Vorteil, dass es den Netzwerkverkehr reduziert und es ähnlich wie eine Push-Technologie effizienter mit den Ressourcen umgeht. Aber auch das Long Polling ist nur eine Nachahmung dessen, was man in Wirklichkeit bezweckt, denn der eigentliche Zweck ist, dass die beiden Gesprächsteilnehmer nach Belieben miteinander kommunizieren. Außerdem ist die Implementierung des Long Pollings aufwendig und fehlerträchtig.

1.5.6 SSE mit dem »EventSource«-Objekt

Im Jahre 2014 wurde HTML5 final veröffentlicht, und damit einher gingen zahlreiche Erweiterungen, die die Schwächen von HTML 4 ausmerzen sollten. Eine der Neuerungen war das JavaScript-Objekt `EventSource`, über das bei einem Server ein *Server-sent Event* ausgelöst wird.

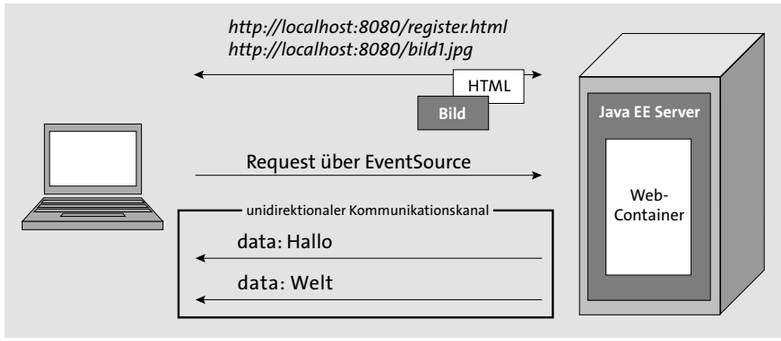

Abbildung 1.18 Über das »EventSource«-Objekt wird ein unidirektionaler Kommunikationskanal geöffnet, über den der Server Daten an den Client senden kann.

Mit Listing 1.1 wird eine Verbindung zu einem Server aufgebaut.

```
<!DOCTYPE html>
<html>
<body>
<div id="ausgabe"></div>
<script>
   if (typeof (EventSource) !== "undefined") {
      var eventSource =
         new EventSource("http://localhost:8080/onlineshop-web/SseServlet");
      eventSource.onmessage = function(event) {
         document.getElementById("ausgabe").innerHTML = event.data;
      };
   } else {
      document.getElementById("ausgabe").innerHTML = "Kein SSE moeglich!";
   }
</script>
</body>
</html>
```

Listing 1.1 sse.html

Sobald die TCP/IP-Verbindung zum Server hergestellt worden ist, bleibt sie dauerhaft bestehen, sodass der Server nach Belieben Daten an den Client versenden kann. Im Servlet erstellen wir die Daten textuell in einem festgelegten Format. Danach werden die Daten in einen Kommunikationskanal gesendet.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.util.concurrent.TimeUnit;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/SseServlet")
public class SseServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;
   protected void doGet(HttpServletRequest request,
```



```
      HttpServletResponse response)
      throws ServletException, IOException {
      response.setContentType("text/event-stream;charset=UTF-8");
      response.setHeader("Cache-Control", "no-cache");
      response.setHeader("Connection", "keep-alive");
      PrintWriter printWriter = response.getWriter();
      while (true) {
         printWriter.print("data: " + LocalDateTime.now() + "\n");
         printWriter.flush();
         try {
            TimeUnit.SECONDS.sleep(1);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Listing 1.2 SseServlet.java

Wenn Sie dieses Beispielprogramm mit einem neueren Google Chrome ausprobieren, sollte im Fenster des Webbrowsers sekündlich eine Uhrzeit untereinandergeschrieben werden. Allerdings funktioniert das Beispiel mit einem Internet Explorer nicht. Dieses Problem könnte mit einem sogenannten *PolyFill* gelöst werden. Ein PolyFill ist ein Programm, das ein bestimmtes Feature eines Webbrowsers implementiert, wenn der Webbrowser hierauf ursprünglich verzichtet. Allerdings ist das eine umständliche Lösung. Ein weiteres Problem besteht darin, dass der Server es nicht sofort bemerkt, wenn der Webbrowser geschlossen wird. Außerdem ist die Kommunikation zwischen Client und Server nicht bidirektional, da der Client nur noch Daten empfängt. Um beim Onlineshop einfach nur den Verkaufsstatus kenntlich zu machen, ist diese Funktionalität ausreichend. Aber nicht nur für die Benutzer des Internet Explorers, sondern auch für zahlreiche Anwendungsfälle, bei denen eine bidirektionale Kommunikation erforderlich ist, stellt das `EventSource`-Objekt keine Option dar, denn das Protokoll HTTP/1.1 ist nun mal nicht für diesen Anwendungsfall ausgelegt.

1.5.7 HTTP/2

Mit der Servlet-Version 4.0 kehrt das HTTP/2-Protokoll in den Java EE-Standard ein und löst alle oben genannten Probleme mit einem Schlag.

Führen wir uns noch einmal HTTP/1.1 vor Augen, um die vorhandenen Probleme zu verdeutlichen. Bei einer HTTP/1.1-Verbindung beantwortet der Server einen HTTP-Request üblicherweise mit einer HTML-Seite und wartet anschließend, bis der Webbrowser das HTML eingelesen hat und fehlende Elemente wie Bilder, CSS-Dateien oder JavaScript-Dateien nach-

fordert. Obwohl HTTP/2 hierzu rückwärtskompatibel ist, benötigt es lediglich eine einzige TCP/IP-Verbindung, um die Daten im sogenannten *Full-Duplex-Multiplexingverfahren* (Full Duplex: im gleichzeitigen Gegenbetrieb; Multiplexing: Bündelung bzw. Auflösung von zusammengefassten Blöcken) auszutauschen. Um diese Verbesserungen, die sich hieraus ergeben, zu nutzen, brauchen Sie bei der Servlet 4.0 Technologie grundsätzlich überhaupt nichts zu tun, denn schon während der Verbindung beginnt der Server zwischenzeitlich mit dem Versand von Ressourcen, von denen er der Meinung ist, dass sie vom Webbrowser gebraucht werden. Je nach Anzahl der Ressourcen kann allein der Server-Push zu einem wesentlich verbesserten Seitenaufbau beim Webbrowser führen. Dieser Geschwindigkeitsvorteil wird auch noch verbessert, indem im Header nur die Bestandteile versendet werden, die sich ändern. Der Zeitgewinn hierdurch ist nicht zu unterschätzen, denn die Verwendung der Cookies führte bei HTTP/1.1 häufig zu einer etwa 1 KB großen Header-Überlast. Hinzu kam, dass die Cookies aufgrund des zustandslosen Protokolls immer wieder versendet wurden und die Natur des TCP/IP-Protokolls diese Rundreisen auch noch verlangsamt, weil zu Beginn der Verbindung die verfügbare Kapazität gemessen wird. Einen weiteren Geschwindigkeitsvorteil bietet HTTP/2, weil es die HTML-Seiten in einem binären Format versendet, während HTTP/1.1 ein rein textuelles Format ist. Weil das Binärformat kompakter ist, muss eine geringere Menge an Bytes verschickt werden als bei einem textuellen Format. Das Binärformat hat darüber hinaus den Vorteil, dass es für das Sniffer-Programm eines Hackers erst nach Entschlüsselung preisgegeben wird. Außerdem setzt HTTP/2 eine SSL-Verschlüsselung über HTTPS für die Kommunikation zwischen Client und Server voraus.

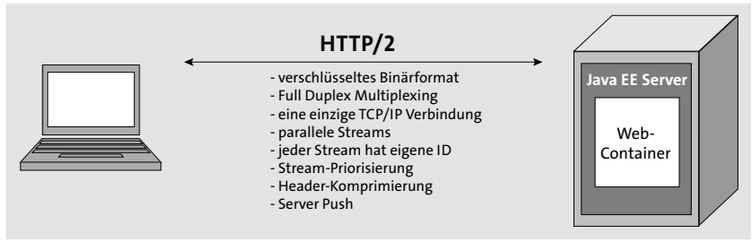

Abbildung 1.19 HTTP/2 bietet zahlreiche Vorteile gegenüber HTTP/1.1.

HTTP/2 ist nicht nur schneller und sicherer, sondern bietet viele weitere Features. Beispielweise wird das sogenannte Head-of-Line-Blocking-Problem gelöst. Das Head-of-Line-Blocking kommt beim HTTP/1.1-Protokoll bei einer Netzwerkunterbrechung auf, weil die Enden des abgeschnittenen Streams einander nicht mehr zugeordnet werden können. Dieses Problem löst HTTP/2, indem jeder Stream durch eine eigene Streaming-ID identifiziert wird. Bricht der Stream ab, lassen sich die beiden Enden eines Streams beim Neuaufbau korrekt zuordnen, denn der Webbrowser kennt ja ihre Stream-IDs. Handelt es sich um mehrere Streams, kann auch eine Priorisierung der einzelnen Streaming-IDs angewiesen werden. Bei dieser Priorisierung handelt es sich nicht einfach um eine unterschiedliche Gewichtung einzelner IDs, sondern um eine komplexe Anordnungsstrategie ganzer ID-Hierarchiebäume.

1.5.8 WebSocket

Das WebSocket-Protokoll wurde bereits im Jahre 2009 von Ian Hickson spezifiziert, einem Mitarbeiter des Unternehmens Google. WebSocket wurde vom IETF im Dezember 2011 als RFC 6455 standardisiert. Das Dokument kann unter folgender URL heruntergeladen werden:

http://tools.ietf.org/html/rfc6455

Parallel dazu beschrieb Ian Hickson auch, wie eine JavaScript-API zum WebSocket-Protokoll aussehen könnte. Das W3C übernahm seine API-Spezifikation, die Sie als *Candidate Recommendation* unter folgender URL herunterladen können:

http://www.w3.org/TR/websockets

WebSockets ist ein Netzwerkprotokoll, bei dem sich Client und Server im *Full-Duplex-Multiplexingverfahren* unterhalten können. Mit WebSockets können beide Kommunikationspartner ohne Einschränkung einen Datenstrom an die Gegenstelle schicken. Bestimmt stellen Sie sich nun die Frage, ob hiermit denn nicht das Gleiche erreicht wird wie mit dem oben beschriebenen HTTP/2-Protokoll, handelt es sich doch in beiden Fällen um ein *Full-Duplex-Multiplexingverfahren*. Die Unterschiede zwischen WebSockets und HTTP/2-Protokoll sind aber eklatant. Als erste Unterscheidung lässt sich vorausschicken, dass HTTP/2 zwar bidirektional funktioniert, dass aber lediglich der Webbrowser eine Anfrage versendet, die der Server anschließend beantwortet. Auch wenn HTTP/2 einen Server-Push ermöglicht, so sind Client und Server also nicht gleichberechtigt. Wir schauen uns dies nun im Detail an.

Da HTTP die Grundlage des Internets darstellt, ist auch das WebSocket-Protokoll so aufgebaut, dass sich der Webbrowser zunächst über einen HTTP-Request mit dem Server verbindet. Dieser anfängliche HTTP-Request wird *Opening Handshake Request* genannt. In Listing 1.3 sehen Sie den HTTP-Header eines Opening Handshake Requests:

```
GET /chat HTTP/1.1
Host: www.marktplatz.de:8080
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: eIverKOqwLAcRSRgr18lTV==
Sec-WebSocket-Protocol: meinwebsocket
Sec-WebSocket-Version: 13
```

Listing 1.3 Der HTTP-Header des Opening Handshake Requests

Der Server beantwortet die Anfrage mit einer Opening Handshake Response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection:Upgrade
Sec-WebSocket-Accept:L6wqtsHk6dzD+kd9NCYT6Wt7OCU=
```

```
Sec-WebSocket-Protocol: meinwebsocket
Upgrade:WebSocket
```

Listing 1.4 Die Opening Handshake Response

Weil es sich hierbei gewissermaßen um einen gewöhnlichen HTTP-Request handelt, wissen Router, Proxys und andere Instrumente des Internets hiermit umzugehen. Im Header des Opening Handshake Requests sind allerdings noch weitere Informationen enthalten, die ihn als Eröffnungstakt für die WebSocket-Kommunikation etikettieren. Kurzum: Bei WebSockets wird das HTTP-Protokoll lediglich dazu verwendet, die Kommunikation zwischen Client und Server anzustoßen. Nach dem Opening Handshake führen die Teilnehmer ihre Unterhaltung mit einem Protokoll fort, das nichts mehr mit HTTP zu tun hat. So gibt es beispielsweise auch keine HTTP-Header mehr. Stattdessen besteht eine TCP/IP-Verbindung, die einen Datenaustausch in beide Richtungen erlaubt. Um diese Unterscheidung hervorzuheben, spricht man bei WebSockets von *Frames* und unterscheidet bei den Frames zwischen *Control Frames* und *Data Frames*.

Control Frames werden eingesetzt, um die Kommunikation zwischen dem Client und dem Server zu koordinieren. Beispielsweise wird zu Beginn der Verbindung ein *Control Frame* in Form des *Opening Handshakes* benötigt. Aber auch wenn einer der Teilnehmer die Verbindung beenden möchte und hierzu einen *Closing Handshake* verschickt, handelt es sich um einen *Control Frame*.

Mit einem *Data Frame* werden die Geschäftsdaten übermittelt. Dabei kann es sich um Texte oder auch um Binärdaten (wie beispielsweise Videodaten) handeln. Die Anzahl der Data Frames ist nicht begrenzt, sodass es sich im Prinzip um eine endlose Verkettung von Data Frames handeln könnte.

Betrachten Sie hierzu Abbildung 1.20.

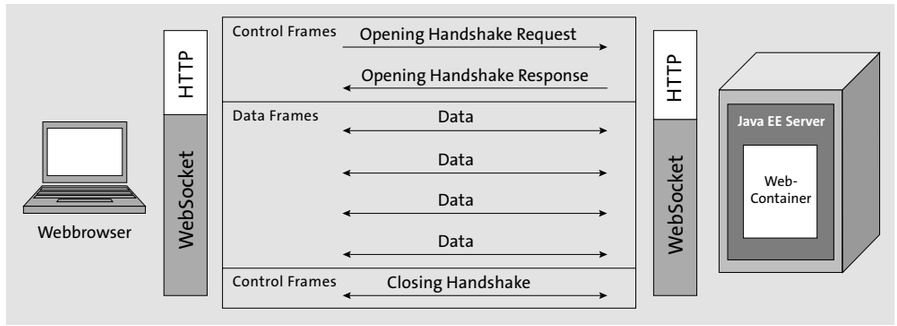

Abbildung 1.20 Die Kommunikation über das WebSocket-Protokoll

1.6 Die Technologien des Java EE 8-Standards

Jede Java EE-Version enthält eine Menge an Java EE-Technologien, und mit jeder Version kommen neue hinzu. Java EE 8 umfasst 32 Technologien, die jeweils in eigenen Spezifikationsdokumenten beschrieben werden.

In Tabelle 1.1 werden alle Spezifikationen des Java EE 8-Standards aufgelistet. Neben dem Rahmenwerk der Java EE 8-Spezifikation »Java Platform Enterprise Edition Specification v8« werden auch die enthaltenen Java EE 8-Technologien in den folgenden Hauptgruppen angezeigt:

- Webtechnologien
- Enterprise-Technologien
- Webservices-Technologien

Neben diesen drei Hauptgruppen enthält das Gesamtpaket Technologien, die unter folgenden Gruppen zusammengefasst werden:

- Management- und Security-Technologien
- Java SE-Technologien, die sich auf Java EE beziehen

Technologietyp	Java Platform Enterprise Edition Specification v8	JSR 366
Web	**Servlets 4.0**	**JSR 369**
	Java Server Pages (JSP) 2.3	JSR 245
	Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52
	Expression Language (EL) 3.0	JSR 341
	JavaServer Faces 2.3	**JSR 372**
	Java API for WebSocket 1.1	**JSR 356**
	Java API for JSON Processing 1.1	**JSR 374**
	Java API for JSON-Binding 1.0	**JSR 367**
Enterprise	**Java Persistence 2.2**	**JSR 338**
	Enterprise JavaBeans 3.2	JSR 345
	Java Message Service API 2.0	JSR 368
	Contexts and Dependency Injection for Java (CDI) 2.0	**JSR 365**

Tabelle 1.1 Die Java EE 8-Technologien. Bei den fett gedruckten Zeilen handelt es sich um die abgeänderten oder hinzugekommenen Technologien.

Technologietyp	Java Platform Enterprise Edition Specification v8	JSR 366
	Dependency Injection for Java 1.0	JSR 330
	Bean Validation 2.0	**JSR 380**
	Java Transaction API (JTA) 1.2	JSR 907
	Java EE Connector Architecture 1.7	JSR 322
	JavaMail 1.6	**JSR 919**
	Interceptors 1.2	JSR 318
	Concurrency Utilities for Java EE 1.0	JSR 236
	Common Annotations for the Java Platform 1.3	**JSR 250**
	Batch Applications for the Java Platform	JSR 352
Webservices	Java API for XML-Based Webservices (JAX-WS) 2.2	JSR 224
	Java API for RESTful Webservices (JAX-RS) 2.1	**JSR 370**
	Implementing Enterprise Webservices 1.3	JSR 109
	Webservices Metadata for the Java Platform	JSR 181
	Java APIs for XML Messaging 1.3	JSR 67
	Java API for XML Registries (JAXR) 1.0	JSR 93
Security	**Java EE Security API 1.0**	**JSR 375**
	Java Authentication Service Provider Interface for Containers 1.1	JSR 196
	Java Authorization Contract for Containers 1.5	JSR 115
Management	J2EE Management 1.1	JSR 77
	Debugging Support for Other Languages 1.0	JSR 45

Tabelle 1.1 Die Java EE 8-Technologien. Bei den fett gedruckten Zeilen handelt es sich um die abgeänderten oder hinzugekommenen Technologien. (Forts.)

Bei den fett dargestellten Technologien handelt es sich um neue oder aktualisierte Technologien.

Folgende Technologien sind neu hinzugekommen:

- **Java API for JSON Binding (JSON-B)**
- **Java EE Security API**

Folgende Technologien wurden mit einem neuen JSR wesentlich überarbeitet:

- **Servlets 4.0**: überarbeitet mit JSR 369
- **Java Server Faces (JSF) 2.3**: überarbeitet mit JSR 372
- **Java API for JSON Processing (JSON-P) 1.1**: überarbeitet mit JSR 374
- **Contexts and Dependency Injection for Java 2.0**: überarbeitet mit JSR 365
- **Bean Validation 2.0**: überarbeitet mit JSR 380
- **Java API for RESTful Web Services (JAX-RS) 2.1:** überarbeitet mit JSR 370

Bei folgenden Technologien handelt es sich um Wartungs-Releases, die mit gleichem JSR wie ihr Vorgänger veröffentlicht wurden:

- **Java API for WebSocket 1.1**: Wartungs-Release, immer noch JSR 356 wie Vorgänger
- **Java Persistence 2.2**: Wartungs-Release, immer noch JSR 338 wie Vorgänger
- **JavaMail 1.6**: Wartungs-Release, immer noch JSR 919 wie Vorgänger

In diesem Buch werde ich die folgenden Technologien behandeln:

Kapitel	Technologie
Kapitel 1, »Überblick«	HTTP/2 WebSocket 1.1 Maven
Kapitel 2, »Die Entwicklungsumgebung«	UTF-8
Kapitel 3, »Planung und Entwurf«	Scrum AMDD-XP
Kapitel 4, »Servlet 4.0«	Servlets 4.0
Kapitel 5, »Java Server Pages«	JSP 2.3 JSTL1.2 JSP-EL 3.0
Kapitel 6, »Die relationale Datenbank«	Oracle DB ANSI SQL-92
Kapitel 7, »JDBC«	JDBC 4.2

Tabelle 1.2 Behandelte Technologien

Kapitel	Technologie
Kapitel 8, »Die Java Persistence API«	JPA 2.2
Kapitel 9, »Java Server Faces«	JSF 2.3 JSF-EL 3.0 CDI 2.0
Kapitel 10, »Enterprise JavaBeans«	EJB 3.2 JMS 2.0
Kapitel 11, »Webservices und JSON«	JSON-P 1.1 JSON-B 1.0 JAX-WS 2.2 JAX-RS 2.1

Tabelle 1.2 Behandelte Technologien (Forts.)

In den folgenden Abschnitten erhalten Sie jeweils eine kurze Beschreibung der Java EE-Technologien, die für den Java EE-Entwickler von besonderer Bedeutung sind.

1.6.1 Servlets 4.0

Historisch gesehen handelt es sich bei einer Java EE-Anwendung in der Regel um eine Webanwendung, d. h. eine Anwendung, die über einen Webbrowser verwendet wird. Obwohl das Frontend typischerweise über die High-Level-Technologie *Java Server Faces* implementiert wird, spielt die Servlet-Technologie auch bei *Java Server Faces* eine zentrale Rolle, da auch sie als High-Level-Technologie auf der Low-Level-Technologie Servlets aufsetzt. Letztlich ist es also auch bei *Java Server Faces* ein Servlet, das im Hintergrund arbeitet und mithilfe des Webcontainers für die Interaktion zwischen der clientseitigen und der serverseitigen Präsentationsschicht verantwortlich ist. Mit Java EE 8 wurde die Servlet-Technologie auf die Version 4.0 angehoben.

Das Konzept der Servlet-Technologie wurde bereits im Jahre 1995 vom Java-Altvater James Gosling vorgestellt und später von Pavani Diwanji überarbeitet. Sun Microsystems stellte die Servlet-Technologie schließlich in der Version 1.0 im Jahre 1997 fertig. Als Java EE 1.0 erstmalig als Sammelspezifikation im Jahre 1999 veröffentlicht wurde, gehörte die Servlet-Technologie in der Version 2.2 gleich als fester Bestandteil dazu. Anfangs hatte man eine Schnittstelle entwickelt, die auf dem *Common Gateway Interface* (*CGI*) basierte, und darüber hinaus konnte man nun mit Java objektbasiert und komfortabel HTTP-Requests und HTTP-Responses verarbeiten.

Die Servlet-Technologie baut im Wesentlichen auf Java-Komponenten auf, die sich ebenfalls Servlets nennen. Aus der Sicht des Webcontainers ist ein Servlet eine Java-Klasse, die das Interface `javax.servlet.Servlet` implementiert. Obwohl dies die einzige Voraussetzung ist, die eine Java-Klasse mitbringen muss, damit sie in einem Webcontainer ausgeführt werden kann, ist es dennoch nicht üblich, auf diese Weise ein Servlet zu programmieren, denn wie ich in Kapitel 4 noch detaillierter zeigen werde, programmiert man Servlets, indem man eine Java-Klasse von der abstrakten Klasse `javax.servlet.http.HttpServlet` ableitet. Denn erst diese Unterklasse enthält alles, was erforderlich ist, damit ein Servlet zu einer vollwertigen Webkomponente wird. Im folgenden Beispiel wird ein Servlet mit dem Parameter `name` aufgerufen. Hierfür wird in der Adressleiste eines Webbrowsers folgende URL eingegeben.

```
http://localhost:8080/onlineshop-web/SigninServlet?name=Alex
```

In einem Servlet wird die Anfrage vom Webcontainer entgegengenommen und verarbeitet. Im vorliegenden Fall wird der Webcontainer ein Servlet suchen, das unter dem Bezeichner `SigninServlet` zu finden ist.

Innerhalb des `SigninServlet` würde der Quelltext aus Listing 1.5 dafür sorgen, dass der Webcontainer die Anfrage mit einer Hallo-Seite beantwortet.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/SigninServlet")
public class SigninServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(
         HttpServletRequest request,
         HttpServletResponse response)
               throws ServletException, IOException {

      PrintWriter out = response.getWriter();
      String name = request.getParameter("name");
      out.println("<html>");
      out.println("<body>");
```

```
        out.println("<h1>Hallo " + name + "</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Listing 1.5 SigninServlet.java

Ist der Webcontainer erst einmal in Betrieb, braucht eine Webkomponente nur ein einziges Mal aufgerufen zu werden. Da der Webcontainer die Parallelprogrammierung beherrscht, kann er die Webkomponenten zur selben Zeit für weitere gleichzeitig aufkommende Anfragen verwenden. Dabei wird die Anfrage nicht direkt an die Webkomponente weitergeleitet, sondern zunächst automatisch vom Webcontainer ausgewertet. Erst im Anschluss kommt die Webkomponente ins Spiel, denn sie ist für die fachliche Bearbeitung der Anfrage zuständig.

Mit der Servlets-Version 4.0 wird das HTTP/2-Protokoll unterstützt. In einem vorangegangenen Abschnitt habe ich bereits gezeigt, wieso die Unterstützung des HTTP/2-Protokolls von so großer Bedeutung ist, und erklärt, dass HTTP/2 mit einer einzigen TCP/IP-Verbindung auskommt, um eine bidirektionale Multiplexingkommunikation zwischen Client und Server anzubieten. Darüber hinaus besteht die Möglichkeit eines Server-Pushs, den Sie innerhalb des Servlets selbst programmieren können. Auch hierbei versenden Sie Server-Inhalte proaktiv an den Cache des Clients. Hierfür wurde in der Servlet API eine neue Klasse eingebaut, die sich `PushBuilder` nennt. Listing 1.6 zeigt, wie ein Objekt des Typs `PushBuilder` ein Bild an den Client sendet:

```
PushBuilder pushBuilder = request.newPushBuilder();
pushBuilder
   .path("images/bild1.jpg")
   .addHeader("content-type", "image/jpg")
   .push();
```

Listing 1.6 PushBuilderServlet.java

In Listing 1.6 sehen Sie, dass Sie das Objekt eines `PushBuilder` erzeugen, indem Sie die Methode `newPushBuilder` bei dem `Request`-Objekt aufrufen. Anschließend können Sie bei dem `PushBuilder` über die Methode `path` das Web-Element anzeigen, das an den Client versendet werden soll. Der `PushBuilder` bietet noch weitere Methoden wie die Methode `addHeader` an, über die weitere Informationen zum Push angezeigt werden. Zuletzt rufen Sie die Methode `push()` auf, mit der das Web-Element an den Webbrowser geschickt wird.

Neben dem Einbau von HTTP/2 bietet die neue Servlet-Technologie eine spezielle Mapping API, über die sich die Pfade zu einzelnen Elementen komfortabel ermitteln lassen.

1.6.2 JSP 2.3

Als Servlets erfunden wurden, entwickelte man Webkomponenten genauso, wie ich es im obigen Servlet-Beispiel gezeigt habe: Man schrieb die HTML-Texte in einen Ausgabestrom. Wenn Sie sich vor Augen halten, dass Webseiten häufig aus Hunderten von HTML-Zeilen bestehen, können Sie sich leicht vorstellen, wie wartungsintensiv Servlets sein können.

Aufgrund des hohen Wartungsaufwands von Servlets fügte man bereits der allerersten Java EE-Version 1.0 die Technologie *Java Server Pages* (*JSP*) hinzu. Eine JSP sieht dem Anschein nach wie eine HTML-Seite aus. Eine spezielle JSP-Engine namens *Jasper* erzeugt aus jeder JSP ein automatisch generiertes Servlet. Dieses Servlet setzt die Zeichenketten der JSP in einen Ausgabestrom, genauso wie es im obigen Beispiel im Servlet durchgeführt werden könnte.

In einer Datei mit dem Namen *signin.jsp* würden wir mit HTML arbeiten und hierbei die dynamischen Inhalte in spezielle Tags einbetten.

```
<html>
<body>
Hallo <%=request.getParameter("name")%>
</body>
</html>
```

Listing 1.7 signin.jsp

In der Adressleiste eines Webbrowsers würden wir nun folgende URL eingeben, um vom Server die Zeichenkette »Hallo Alex« zu erhalten:

http://localhost:8080/onlineshop-war/signin.jsp?name=Alex

Die Umstülpung ermöglicht es, innerhalb des HTML-Quelltextes reinen Java-Code zu programmieren. Dadurch kann ein Webdesigner eine JSP grafisch gestalten, und ein Java-Entwickler kann in der gleichen Datei Java-Anweisungen setzen.

Zusätzlich können individuelle JSP-Elemente als Java-Klassen programmiert werden, die die JSP-Engine als dynamische Java-Anweisungen interpretiert. Die JSP-Elemente können vom Webdesigner innerhalb der JSP genauso wie HTML-Elemente gesetzt werden. Hierdurch werden die statischen HTML-Elemente um dynamische JSP-Elemente erweitert, denn der klassische Webdesigner arbeitet normalerweise nur mit HTML und CSS. Für ihn ist Java »kryptisches Zeug«, mit dem er sich nicht herumschlagen mag.

Für die wichtigsten Zwecke bietet die JSP-API sogenannte JSP-Aktionselemente an, sodass diese nicht mehr individuell programmiert werden müssen. In Listing 1.8 zeige ich, wie die JSP-Aktionselemente `useBean` und `getProperty` genutzt werden, um die E-Mail-Adresse eines Kunden auszugeben.

```
<html>
<body>
<jsp:useBean id="customer"
    class="de.java2enterprise.onlineshop.model.Customer"/>
<jsp:getProperty
        property="email"
        name="customer"/>
</body>
</html>
```

Listing 1.8 signin.jsp

1.6.3 EL 3.0

Das obige Beispiel zeigt, wie kompliziert die Verwendung von JSP-Elementen sein kann. Die Antwort auf dieses Problem kam mit der *Expression Language* (*EL*). Listing 1.9 zeigt, wie einfach die Ausgabe der Kunden-E-Mail-Adresse hiermit programmiert werden kann:

```
<html>
<body>
${customer.email}
</body>
</html>
```

Listing 1.9 index.jsp

Die EL wird nicht nur in JSPs, sondern auch für die Technologie JSF bei sogenannten *Facelets* eingesetzt. Die Untermenge, die man in JSPs verwendet, bezeichnet man als *JSP-EL*. Die Untermenge für Facelets wird *JSF-EL* genannt.

JSF-EL erkennen Sie an der voranstehenden Raute:

```
<html>
<body>
#{customer.email}
</body>
</html>
```

Listing 1.10 index.xhtml

1.6.4 JSTL 1.2

Die *Java Server Pages Standard Tag Library* (*JSTL*) bietet spezielle Elemente an, über die zum Beispiel Verzweigungen, Schleifen oder die Ausgabe von Werten programmiert werden können. In Listing 1.11 sehen Sie, wie mit JSTL geprüft wird, ob die JavaBean mit dem Namen `customer` gesetzt worden ist. Wenn sie keinen Wert enthält, wird `Bitte einloggen!` ausgegeben.


```
<c:if test="${empty customer}">
    Bitte einloggen!
</c:if>
```

Listing 1.11 index.jsp

1.6.5 JSF 2.3

Als man mit den ersten Java EE-Versionen noch Webanwendungen mit Servlets und JSPs programmierte, wurde bald eine klare Aufgabenverteilung deutlich: Während man Servlets einsetzte, um den Ablauf zu steuern, verwendete man JSPs nur noch für die Präsentation der Geschäftsdaten. Eine dritte Aufgabe kam ganz »normalen« Java-Klassen zu, denn man benötigte auch einen Transportbehälter für die Geschäftsdaten. Den ganzen Verbund der Klassen, den man für das Geschäftsmodell entwirft, nennt man *Domänenmodell* oder auch einfach nur *Model*. Deshalb wird das bereits vorgestellte Entwurfsmuster als *Model-View-Controller*-(MVC-)Pattern bezeichnet.

In den ersten Java EE-Versionen zeigte sich, dass bei dem MVC-Entwurfsmuster mit Servlets und JSPs zahlreiche Programmieranweisungen immer wieder gleich blieben. Deshalb entwickelte man Frameworks, die diesen Quelltext automatisch erstellen. Die wichtigsten Erkenntnisse dieser Frameworks wurden in die sogenannten *Java Server Faces* übernommen, denn Java Server Faces enthalten all das, was die Java-Gemeinde aus der jahrelangen Erfahrung mit den älteren Java EE-Open-Source-Frameworks gelernt hat. Das wichtigste Merkmal von JSF ist hierbei, dass es das MVC-Entwurfsmuster in Perfektion umsetzt. Allerdings nimmt das JSF-Framework dem Entwickler noch eine ganze Reihe weiterer Arbeiten ab.

Die mit Java EE 8 neue hinzugekommene JSF 2.3 API bringt zahlreiche Verbesserungen mit. Beispielsweise werden erstmals Lambda Expressions, Streams und die neue Date-Time-API von Java SE 8 unterstützt. Ferner bietet sie eine verbesserte Integrierung der *Context und Dependency Injection* (*CDI*). Darüber hinaus wurden zahlreiche Erweiterungen bei den Facelets eingebaut, sodass sich die Validierung und Konvertierung von Daten verbessert hat. Neben diesen Optimierungen werden Web-Elemente nun per Server-Push an den Webbrowser versendet. Als JSF-Programmierer brauchen Sie hierfür gar nicht aktiv zu werden, denn JSF 2.3 macht das von sich aus. Allerdings setzt HTTP/2 voraus, dass sich der Webbrowser über HTTPS mit dem Java EE Server verbunden hat.

1.6.6 WebSockets 1.1

Das WebSocket-Protokoll ermöglicht eine bidirektionale Full-Duplex-Kommunikation zwischen Client und Server. Seit der Version 7 unterstützt der Java EE-Standard das WebSocket-Protokoll mithilfe der *Java API for WebSocket*. Die Eingliederung erfolgte mit dem JSR 356. Java EE 8 kommt mit dem Wartungs-Release *Java API for WebSocket 1.1* einher. WebSockets werden clientseitig über JavaScript implementiert.

Als Beispiel werden wir ein kleines Chat-Programm erstellen, das das WebSocket-Protokoll verwendet. Zunächst benötigen wir eine Webseite, in der wir ein HTML-Formular mit zwei Eingabefeldern und einem Button unterbringen werden. Wir nennen die Webseite *websocket.html*.

```
<html>
<body>
   <form action="">
      Benutzer:<input id="benutzer">
      Nachricht:<input id="nachricht">
      <input type="button" onclick="sende();" value="OK">
   </form>
   <div id="ausgabe"></div>
   <script type="text/javascript" src="websocket.js">
   </script>
</body>
</html>
```

Listing 1.12 websocket.html

Im ersten Eingabefeld tragen die Chat-Teilnehmer ihren Namen ein. In das zweite Eingabefeld schreiben sie ihre Nachrichten. Durch einen Mausklick auf den Button wird eine JavaScript-Funktion aufgerufen, die sich `sende()` nennt. Unterhalb des HTML-Formulars habe ich ein `<div>`-Element für die Ausgaben hinterlegt. Darunter sehen Sie, dass die JavaScript-Datei *websocket.js* eingebunden wurde. Diese JavaScript-Datei setzt die WebSocket-API ein, um die Texte der beiden Felder per WebSocket-Protokoll an den Server zu verschicken. In Listing 1.13 ist der komplette Quelltext des JavaScript-Programms abgedruckt:

```
var websocket = new WebSocket("ws://localhost:8080/onlineshop-web/websocket");
var ausgabe = document.getElementById("ausgabe");

websocket.onopen = function(evt) {
   ausgabe.innerHTML += "Verbunden mit " + websocket.url + "<br>";
};
websocket.onmessage = function(evt) {
   ausgabe.innerHTML += evt.data + "<br>";
};
```



```
websocket.onerror = function(evt) {
   ausgabe.innerHTML += "Fehler: " + evt.data + "<br>";
};

function sende() {
    websocket.send(benutzer.value + ": " + nachricht.value);
}
```

Listing 1.13 websocket.js

Das wichtigste WebSocket-Objekt des JavaScript-Standards nennt sich `WebSocket`. Der Konstruktor des `WebSocket`-Objekts nimmt eine `ws`-URI als Parameter entgegen und startet hiermit den Opening Handshake Request.

Anschließend werden die Callback-Funktionen `onOpen` und `onMessage` mit Funktionen verbunden, die im jeweiligen Fall automatisch aufgerufen werden. Nach Öffnung der Verbindung wird im Programm der Name des Benutzers mit seiner Nachricht auf den Bildschirm aller Chat-Teilnehmer ausgegeben.

Serverseitig werden wir nun die *Java API for WebSocket* nutzen. In Listing 1.14 sehen Sie, wie eine Klasse den serverseitigen Teil der Chat-Anwendung übernimmt.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import javax.websocket.EncodeException;
import javax.websocket.OnMessage;
import javax.websocket.RemoteEndpoint.Basic;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocket")
public class WebSocket {
   @OnMessage
   public void message(String message, Session session)
           throws IOException, EncodeException {

      for (Session s : session.getOpenSessions()) {
         Basic endpoint = s.getBasicRemote();
         endpoint.sendText(message);
      }
   }
}
```

Listing 1.14 WebSocket.java

Um eine Java-Klasse in einen WebSocket-Endpoint umzuwandeln, muss sie mit der Annotation `javax.websocket.server.ServerEndpoint` versehen werden. In einer Klammer tragen wir den Bezeichner `"/websocket"` ein, über den der Endpoint aufrufbar ist.

Die Methode, die bei einer eintreffenden Nachricht aufgerufen werden soll, muss mit der Annotation `javax.websocket.OnMessage` versehen werden. Als Parameter können Sie der Methode beispielsweise eine Zeichenkette mitgeben. In ihr würde die WebSocket-API automatisch die Nachricht setzen, die sie vom Client erhalten hat.

Um die erhaltene Nachricht für das Chat-Programm an alle angebundenen Clients zu verschicken, muss die Methode einen zweiten Parameter des Typs `javax.websocket.Session` annehmen. Diesen wird die WebSocket-API automatisch als Endpoint des Clients setzen:

Die Klasse `javax.websocket.Session` bietet eine Methode an, die sich `getOpenSessions()` nennt. Diese Methode liefert alle Client-Endpoints zurück, die aktuell mit dem Server-Endpoint verbunden sind. Jeder der Client-Endpoints wird auch wieder als Typ `javax.websocket.Session` erstellt, sodass wir über die Methoden `getBasicRemote()` und `sendText()` die Nachricht an alle Client-Endpoints versenden.

Die neuen Methoden von WebSockets 1.1

Mit dem Java EE 8-Standard wurde das neue WebSockets 1.1 veröffentlicht. WebSockets 1.1 bietet zwei neue Methoden an, um Java SE 8 Lambda Expressions zu unterstützen:

- `public void addMessageHandler(Class<T> clazz, MessageHandler.Whole<T> handler);`
- `public void addMessageHandler(Class<T> clazz, MessageHandler.Partial<T> handler);`

1.6.7 JSON-P 1.1

Über die Java API for JSON-Processing (JSON-P) lassen sich JSON-Quellen einlesen und transformieren. JSON-P wurde erstmalig mit Java EE 7 in den Standard aufgenommen. Mit Java EE 8 wird die Technologie JSON-P auf die Version 1.1 erweitert.

Um JSON-Daten zu generieren oder zu parsen, werden mit JSON-P zwei Möglichkeiten angeboten, nämlich das Object Model und das Streaming Model.

- **Die Model API**
  Mit der Model API werden die JSON-Daten im Arbeitsspeicher als Baumstruktur aufgebaut. Dies hat den Vorteil, dass über die Daten zur Laufzeit komfortabel navigiert werden kann und hierbei Daten wahlfrei verändert werden können. Der Nachteil ist jedoch, dass das Einlesen eines großen Datenbaumes zeitaufwendig ist und viel Speicherplatz in Anspruch nimmt.
- **Die Streaming API**
  Bei der Streaming API werden die JSON-Daten als Datenstrom durchlaufen. Soll ein Schlüsselwertpaar ganz unabhängig vom restlichen Datenbestand verarbeitet werden, ist das Streaming Model die schnellere Variante, die kaum Speicherplatz benötigt.

Das mit Java EE 8 hinzugekommene JSON-P 1.1 unterstützt die neuen IETF-Standards JSON Pointer, JSON Patch und JSON Merge Patch. Zusätzlich wurden spezielle Hilfsklassen hinzugefügt, mit denen Java SE 8-Streams verwendet werden können.

1.6.8 JSON-B 1.0

Die Java API JSON Binding 1.0 (JSON-B 1.0) wurde mit Java EE 8 erstmalig in den Standard aufgenommen. Über JSON-B lassen sich einfache Java-Klassen (sogenannte *POJOs*) in JSON-Dokumente wandeln. Hierfür ist lediglich der Aufruf einer einzigen Methode vonnöten. Genauso einfach ist die Umwandlung eines JSON-Dokuments in ein POJO.

1.6.9 JPA 2.2

Datenbankanwendungen, die mit JDBC programmiert werden, enthalten eine große Menge von Anweisungen, die auch automatisch generiert werden könnten. Zum Beispiel ist es mit JDBC üblich, zu jeder Datenbanktabelle eine Java-Klasse zu programmieren. Zur Laufzeit kann hierdurch jeder Datensatz als Java-Objekt erzeugt werden. Solche Java-Objekte werden auch *Entities* genannt. Weil sich der Aufbau einer Entity logisch gesehen aus der dahinterliegenden Datenbanktabelle ergibt, haben sich zahlreiche Java-Frameworks zur Aufgabe gemacht, den Entwickler bei dieser Arbeit zu entlasten. Selbstverständlich war die Nutzung dieser Frameworks recht unterschiedlich. Deshalb wurde im Jahre 2006 die *Java Persistence API* (*JPA*) ins Leben gerufen. Die JPA ist eine standardisierte Schnittstelle, durch die die Nutzung von Persistenz-Frameworks vereinheitlicht wurde.

Bei der JPA 2.2 handelt es sich um ein Maintenance Release, das weiterhin unter dem JSR 338 der JPA 2.1 verwaltet wird. Dennoch bringt die JPA einige wichtige Änderungen mit. Genauso wie bei den Neuerungen anderer Java EE 8-Technologien spielt auch hier wieder die Verwendung von Java SE 8-Features eine zentrale Rolle. Beispielsweise unterstützt die JPA 2.2 die Verwendung von Lambda Expressions, die Stream API, die neue Date-Time-API und die Annotation `@Repeatable` von Java SE 8.

1.6.10 EJB 3.2

Die *Enterprise JavaBean*-Technologie stellt ein Komponentenmodell dar, in dem verteilte Komponenten plattform- und programmiersprachenunabhängig miteinander kommunizieren können. Gleichzeitig ermöglicht das EJB-Komponentenmodell, dass das Gesamtsystem hierbei hochverfügbar und skalierbar bleibt.

Hauptakteure sind die sogenannten *Session Beans*, in denen die Geschäftslogik der Anwendung implementiert wird. Session Beans haben die Besonderheit, dass sie mit entfernten Komponenten plattform- und programmiersprachenunabhängig kommunizieren können. Allerdings habe ich Ihnen bei den Anwendungsszenarien gezeigt, dass Session Beans auch

von Komponenten, die man als *lokal* bezeichnet, aufgerufen werden können. Und das ist mittlerweile auch die gängigere Variante. Unter lokalen Client-Komponenten versteht man Java-Objekte, die sich in der gleichen JVM wie die Session Bean befinden. Aus diesem Grund können lokale Aufrufe nur von Web- oder EJB-Komponenten der gleichen Java EE-Anwendung erfolgen. Außerdem setzen lokale Aufrufe voraus, dass der Java EE Server nicht über Cluster-Instanzen in einer Rechnerfarm betrieben wird. Auf dieses Thema werde ich in den folgenden Kapiteln noch detailliert eingehen.

Kommen wir nun zu den verschiedenen EJB-Typen. Session Beans lassen sich in Stateless Session Beans, Stateful Session Beans und Singleton Session Beans einteilen.

Die *Stateful Session Bean* bietet gegenüber der *Stateless Session Bean* den Vorteil, dass sie den Zustand der Geschäftsdaten für die Dauer der Sitzung behalten kann. Zum Beispiel kann sie in einem Onlineshop für die Einkäufe der Kunden eingesetzt werden.

Wenn innerhalb der gesamten Applikation (präziser innerhalb einer JVM) nur ein einziges Exemplar einer Session-Bean-Klasse existieren soll, wird eine *Singleton Session Bean* programmiert. Zum Beispiel könnte in einem Onlineshop eine Singleton Session Bean als zentrales Modul verwendet werden, um die zur Verfügung stehenden Artikel-Kategorien wiederzugeben. Da es von dieser Singleton Session Bean nur ein einziges Exemplar gäbe, würde sie von allen Clients gleichzeitig verwendet.

Neben den Session Beans bietet die EJB-Komponentenarchitektur *Message-driven Beans* an. Message-driven Beans nutzen einen Messaging-Broker, um asynchron Nachrichten zu empfangen. In der Regel kommt bei einem Java EE Server ein *Java-Message-Service-*(*JMS-*)*Provider* als Messaging-Broker zum Einsatz.

1.6.11 JMS 2.0

Mit JMS können Java EE-Komponenten asynchron miteinander kommunizieren. Die JMS-Spezifikation besteht aus einem *JMS Service Provider Interface* (*JMS SPI*) und einem *JMS Application Programming Interface* (*JMS API*). Die Erstellung von JMS-SPI-basierten JMS-Providern ist den Herstellern von Messaging-Systemen vorbehalten. Für den Java EE-Applikationsentwickler ist dagegen die JMS API interessant, da sie eine einheitliche Schnittstelle zum JMS-Provider darstellt. Bei JMS wird ferner zwischen den *Point-to-Point-* und *Publish-and-Subscribe*-Kommunikationsmodellen unterschieden. Mit Java EE 7 wurde dem *Java Message Service* mit der Version 2.0 eine vereinfachte (*simplified*) API beigefügt. Als man die Technologien für den Java EE 8-Standard festlegte, stand noch JMS in der Version 2.1 auf der Agenda. Zahlreiche Features sollten umgesetzt werden, wie beispielsweise auch hier wieder die Nutzung der Java SE 8 API. Allerdings wurde JMS 2.1 im November 2016 wieder aus der Liste der Java EE 8-Technologien entfernt.

1.6.12 CDI 2.0 und DI

Die *Contexts and Dependency Injection for Java* 2.0 (CDI 2.0) ermöglicht es, Java-Objekte nicht im Programm selbst, sondern durch den umgebenden Container zu instanziieren. Die Spezifikation wurde dem Standard mit der Java EE-Version 6 erstmalig hinzugefügt. Die Referenzimplementierung nennt sich *Weld*.

Was es mit CDI genau auf sich hat und warum CDI erst so spät in den Java EE-Standard einzog, wird deutlich, wenn wir uns die Entwicklungsgeschichte von Java EE anschauen. Drehen wir also unsere Uhren in die Zeit zurück, als es noch keine Java EE-Technologien gab (also vor 1999). Schon damals war es so, dass im Programmcode ein Java-Objekt mit dem `new`-Operator im Arbeitsspeicher erzeugt wurde.

Als sich der Java EE-Standard im Laufe der Jahre durchsetzte, kamen nicht alle Java EE-Technologien bei den Entwicklern gleich gut an. Der große Gewinner war der Webcontainer, ohne den eine Webanwendung mit Java undenkbar war. Dagegen taten die Entwickler sich mit dem damaligen EJB-Container schwer. Deshalb tauchten damals Open-Source-Frameworks auf, die den Java EE-Standard nur teilweise nutzten. Die bekanntesten Open-Source-Frameworks nannten sich *Struts*, *Hibernate*, *Seam* und *Spring*. Während Struts ein reines Web-Framework und Hibernate ein reines Persistenz-Framework war, versuchten Seam und Spring, gleich mehrere Lücken zu füllen, die der damalige Java EE-Standard noch aufwies. Sowohl Seam als auch Spring beabsichtigten darüber hinaus, die Vorteile des EJB-Containers nachzubilden und gleichzeitig die Programmierung für den Entwickler zu vereinfachen.

Insgesamt waren die Open-Source-Frameworks so erfolgreich, dass man alles, was an ihnen besonders nützlich war, in den Java EE-Standard übernahm. Hierbei ließ man sich von den Gründern und Erfindern der Open-Source-Frameworks helfen. Folgerichtig kaufte man die Stars der Open-Source-Szene (gleich einem großen Fußballverein) ein. Beispielsweise waren Craig McClanahan, Gavin King und Rod Johnson als Mitglieder der Expert Group des JCP wesentlich an den Neuentwicklungen beteiligt. Craig McClanahan ist der Gründer von Struts, Architekt des Webcontainers Catalina und technischer Leiter von JSF. Gavin King ist Gründer von Hibernate und Seam und als Mitglied der Expert Groups an JSF, JPA, EJB und (als technischer Leiter) an CDI beteiligt. Rod Johnson, der bereits im Jahre 2001 bei der Servlet-Spezifikation 2.4 mitwirkte und später Spring entwickelte, wurde im Jahre 2008 Mitglied des Java EE-Exekutiv-Komitees. Das, was auf den ersten Blick ungerecht erscheinen mag, kommt den Java EE-Entwicklern jedoch zugute, denn die anarchistische Programmierung mit Open-Source-Frameworks folgte größtenteils unzuverlässigen Gesetzen. Dieses Thema wurde sehr treffend von der Java-Entwicklergemeinde mit dem Begriff *Jar-Hell* zusammengefasst. Um den chaotischen Verhältnissen ein Ende zu setzen, übernahm Oracle die Neuordnung und verbürgt sich seit der Java EE-Version 6 für eine einheitliche API, die unter der Federführung der anerkanntesten Java-Koryphäen gereift ist.

Kommen wir nun zur Entstehung von CDI zurück. Das Muster *Inversion Of Control* (*IoC*) wurde erstmalig im Jahre 2002 in dem Buch »J2EE Design and Development« von Rod John-

son bekannt gemacht. Dies war auch das Fundament für sein neu gegründetes Framework *Spring*. Für das Muster Inversion Of Control im Zusammenhang mit der Verwendung mit Containern erdachte Martin Fowler den genaueren Begriff *Dependency Injection*. Durch die Dependency Injection werden Java-Klassen nicht mehr im Programm instanziiert. Stattdessen übergibt man die Verantwortung an den umgebenden Container. Man spricht von einer »Injektion«, weil die Instanziierung wie bei einer Einimpfung von außen erfolgt. Dass mit der Dependency Injection viele Vorteile verbunden sind, hat damit zu tun, dass man zum Zeitpunkt der Entwicklung noch nicht darüber entscheiden muss, mit welcher konkreten Klasse man ein Objekt instanziieren möchte. Dies kann beispielsweise auch im Nachhinein per Konfiguration vom Administrator geändert werden. Es handelt sich also um einen weiteren Schritt in Richtung »Loskopplung« – und dabei sieht der Quelltext auch noch wesentlich aufgeräumter aus, da viele Hartverdrahtungen entfallen.

Eine Art der Inversion of Control war ja in gewisser Weise bereits in den Servlet- und EJB-Technologien enthalten. Aber die Dependency Injection, die man aus dem Open-Source-Framework Spring abkupferte, kam mit der Java EE-Version 6 neu hinzu. Dank ihr können die Container nun nicht mehr nur Servlets und EJBs verwalten, sondern auch herkömmliche Java-Objekte. Ich habe bereits angesprochen, dass der englische Fachbegriff für die »ganz normalen Java-Objekte« *Plain Old Java Objects* (*POJOs*) lautet. Wenn ein POJO genauso wie ein Servlet oder eine EJB vom Container verwaltet wird, bezeichnet man es in der Java EE-Spezifikation als *Managed Bean*. Bei den verwalteten Objekten wird somit seither zwischen Servlets, EJBs und Managed Beans unterschieden.

Mithilfe von CDI können normale Java-Klassen in Managed Beans umgewandelt werden. Dabei wird ihr Gültigkeitsbereich per Konfiguration festgelegt. In der Regel wird man den Gültigkeitsbereich über eine Annotation des Packages `javax.enterprise.context` definieren. In Listing 1.15 wird der Gültigkeitsbereich der Klasse `RegisterController` auf `@RequestScoped` gesetzt. Hierdurch verbleibt das erzeugte Objekt für die Dauer der Anfrage eines Webbrowsers als verwaltete Komponente im Webcontainer.

```
@RequestScoped
public class RegisterController {
    ...
}
```

Listing 1.15 RegisterController

Innerhalb einer verwalteten Komponente (Servlet, EJB oder Managed Bean) kann eine andere Komponente injiziert werden. Dabei unterscheidet die Spezifikation zwischen einer *Resource Injection* und einer *Dependency Injection*.

Die Resource Injection ermöglicht die automatische Bereitstellung von JNDI-Ressourcen. Das Servlet aus Listing 1.16 injiziert beispielsweise eine `DataSource` zur Verwendung einer Datenbank:


```
@RequestScoped
public class RegisterController {

    @Resource(name="java:comp/OnlineshopDataSource")
    private DataSource ds;

    ...
}
```

Listing 1.16 RegisterController

Mit der Dependency Injection kann in einer verwalteten Komponente eine andere Klasse (nennen wir sie beispielsweise `Customer`) injiziert werden. Die Annotation, mit der die Einimpfung durchgeführt wird, nennt sich `@Inject`.

```
@RequestScoped
public class RegisterController {

    @Inject
    Customer customer;

    ...
}
```

Listing 1.17 RegisterController

Um den `RegisterController` in einer anderen Komponente einzusetzen, lässt er sich über die Annotation `@Named` hierzu kennzeichnen. Hierdurch wird er auch in einem Facelet über seinen Namen aufgerufen werden können.

In Kapitel 9, »Java Server Faces«, werde ich das gezeigte Beispiel noch wesentlich ausführlicher beschreiben, denn eine der wichtigsten Aufgaben von CDI ist, das Zusammenspiel der unterschiedlichen Schichten in der Softwarearchitektur einer Java EE-Anwendung zu erleichtern. In dem fortgeschrittenen Kapitel wird es besonders interessant, da dort das Backend und das Frontend über verwaltete JavaBeans miteinander interagieren.

Mit der im Java EE 8 veröffentlichten CDI-Version 2.0 wurde die CDI-Spezifikation modularisiert. Der Hauptbestandteil mit Konzepten für Qualifier, Scopes, Dependency Injection, Interceptors und Events wurde in der »Core CDI«-Spezifikation untergebracht. Die API für das Bootstrapping von CDI-Containern, die auch in Java SE gültig sind, wurden in der »CDI in Java SE«-Spezifikation definiert. Die Konzepte für EJBs gelangten in die »CDI in Java EE«-Spezifikation.

Mit CDI 2.0 können nun erstmalig Features von Java SE 8, wie Lambda Expressions, die Stream API, die Date-Time-API und die Annotation `@Repeatable`, verwendet werden. Ferner

lassen sich Observer deaktivieren und über die Annotation `@Priority` sortieren. Darüber hinaus gab es viele weitere Verbesserungen. Hierzu gehören beispielsweise Annotation Literals, die statische Klassen darstellen und eine Instanziierung zahlreicher CDI-Annotationen ermöglichen, und asynchrone Events.

1.6.13 JTA 1.2

Ein wichtiger Grundpfeiler des Java EE-Standards ist die *Java Transaction API* (*JTA*). JTA war von vornherein sehr gut durchdacht und blieb über viele Jahre unverändert, was Sie daran erkennen, dass die Versionsnummer lange gleich blieb, denn bis Java EE 7 reichte die JTA-Version 1.1 aus. Dem Java EE 8-Standard wurde JTA in der Version 1.2 hinzugefügt.

Als erfahrenem Java-Entwickler sind Ihnen Transaktionen bestimmt bekannt. Ich gehe dennoch kurz auf das Thema ein, um eine gemeinsame Ausgangsbasis für alle Leser zu schaffen.

In den Prozessen einer Java-Anwendung werden die Geschäftsdaten häufig nicht nur auf einer, sondern gleich auf mehreren Datenbanktabellen abgespeichert. Dabei kommen sehr schnell Transaktionen ins Spiel, denn in der Regel hängen die Speichervorgänge der gleichzeitig auszuführenden Prozesse voneinander ab. Der ANSI-SQL-Standard hat für dieses Problem bereits vor etlichen Jahren einen Grundstein gelegt, indem er die Durchführung der Speichervorgänge in Transaktionen bündelt.

Bei einer Java EE-Anwendung ist die Nutzung von Transaktionen bald eine Selbstverständlichkeit. Dabei werden die im Programm angewiesenen Speichervorgänge meist automatisch durch den EJB-Container in eine Transaktion eingeschlossen. Der JTA-Dienst des EJB-Containers führt hierbei von sich aus ein COMMIT aus, wenn die Speicherung gelingt. Genauso wird der JTA-Dienst des EJB-Containers ein ROLLBACK ausführen, wenn zwischendurch etwas schiefläuft.

Der JTA-Dienst hat dabei die Aufgabe, die sogenannten *ACID*-Eigenschaften der Transaktionen zu gewährleisten. ACID steht für *Atomicity, Consistency, Isolation and Durability*:

- Mit *Atomicity* ist gemeint, dass mehrere Prozesse als Ganzes durchgeführt werden. Das bedeutet, dass jeder einzelne Speichervorgang gelingen muss; ansonsten werden alle Speichervorgänge rückgängig gemacht.
- Die *Consistency* gewährleistet, dass sich die Geschäftsdaten nach dem Durchlauf der beteiligten Transaktionen nicht widersprechen.
- Die *Isolation* sagt aus, dass die Daten während einer unabgeschlossenen Transaktion abgeschirmt sein müssen, da der zwischenzeitliche Einblick zu falschen Rückschlüssen führt. In Kapitel 6, »Die relationale Datenbank«, werden wir auf diese Anforderungen zurückkommen, denn auch die relationale Datenbank muss auf der Low-Level-Ebene bestimmte Abschirmungen vornehmen.
- *Durability* bedeutet, dass der persistierte Zustand nach der Transaktion für andere Prozesse sichtbar ist.

Konventionelle Transaktionen reichen für das in diesem Kapitel eingangs erwähnte Beispiel des Onlineshops aber nicht aus, denn ein Reisebuchungsportal persistiert nicht nur die Datenbanktabellen eines einzelnen Systems, sondern auch die Datenbanktabellen unterschiedlicher Systeme. Dabei kommt dem *JTA-Dienst* eine schwierige Rolle zu. Um die ACID-Eigenschaften auch auf verteilten Systemen zu gewährleisten, wurde einst der *X/Open XA*-Standard eingeführt. Der X/Open XA-Standard ist hierbei nicht nur auf relationale Datenbanken beschränkt, sondern kann auch andere Informationssysteme (wie zum Beispiel Messaging-Systeme) umfassen.

Ein JDBC-Treiber, der X/Open XA-Standard-konform ist, muss das sogenannte Zwei-Phasen-Commit-Protokoll beherrschen. Als Industriestandard muss selbstverständlich auch ein Java EE Server XA-fähig sein. Auch hierfür ist wieder der JTA-Dienst des EJB-Containers zuständig.

Mit der JTA-Version 1.2 wurde die API der JTA erweitert, sodass sie nun nicht mehr zwingend einen EJB-Container benötigt.

1.6.14 JCA 1.7

Zu Beginn dieses Kapitels habe ich erklärt, dass die Geschäftsdaten einer Java EE-Anwendung üblicherweise in einer relationalen Datenbank persistiert werden. Dabei haben Sie auch gelernt, dass die Java EE-Anwendung nicht direkt auf die relationale Datenbank zugreift, sondern stattdessen die Geschäftsdaten über den Java EE Server bezieht. Der Java EE Server entkoppelt die Verbindung zwischen der Java EE-Anwendung und dem EIS, indem der Java EE-Anwendung lediglich der Zugriff auf das *Java Naming and Directory Interface* (*JNDI*) zur Verfügung gestellt wird. Dies sind Bestandteile eines Gesamtkonzepts des Java EE-Standards, die man in der Java EE Connector Architecture (JCA) festgeschrieben hat. JCA definiert eine standardisierte Softwarearchitektur, bei der die beteiligten Komponenten und Schnittstellen einem festgelegten Muster folgen müssen. Gleichzeitig wurden zur Abstraktion allgemeingültige Begriffe festgelegt. Beispielsweise nennt sich das Medium, auf dem die Geschäftsdaten *persistiert* sind, *Enterprise Information System* (*EIS*). Die Java EE-Spezifikation drückt sich so abstrakt aus, weil die Quelle der dauerhaften Datenhaltung austauschbar sein soll. Man spricht auch von einer *losen Ankopplung*. Dies ist ein wesentliches Kriterium von JCA. Die Grundidee ist, eine zentrale Datenbasis ganz allgemein zur Verfügung zu stellen. Dabei soll es sich um jede Art von Datenhaltung handeln, die das Unternehmen in die Lage versetzt, seine Geschäftsprozesse abzubilden. In der Praxis sieht aber alles viel konkreter aus, denn in der Regel wird die Datenhaltung von geschäftskritischen Unternehmensanwendungen durch eine relationale Datenbank realisiert.

Auch bei der Kommunikation mit der Datenhaltung werden abstrakte Begriffe verwendet. Dort heißt es, dass der Zugriff auf die Ressourcen eines EIS ganz allgemein über einen Ressourcen-Adapter erfolgt, der das *Common Client Interface* (*CCI*) implementiert. Bei der Arbeit mit einer relationalen Datenbank stellt der *Java Database Connectivity*-(*JDBC*-)Treiber das dar, was man in der Connector-Architektur als Ressourcenadapter bezeichnet.

1.6.15 Bean Validation 2.0

Die Bean Validation wurde erstmalig im Jahre 2009 mit Java EE 6 in den Standard aufgenommen. Die Spezifikation legt eine Validation API und insbesondere bestimmte Annotationen fest, mit deren Hilfe eine automatisierte Validierung von JavaBeans erzwungen wird. Die Annotationen können aber auch durch entsprechende XML-Definitionen überschrieben werden. Die Referenzimplementierung der Spezifikation nennt sich *Hibernate Validator Version 6*. Eine der Besonderheiten der Validierungs-API ist, dass sie keiner bestimmten Schicht des Multi-Tier-Aufbaus einer Java EE-Anwendung zugeordnet wurde, sondern stattdessen sowohl in der Webschicht als auch in der Persistenz-Schicht eingesetzt werden kann.

Die mit dem JSR 380 veröffentlichte Bean Validation 2.0 wurde für Java SE 8 erforderlich, weil Java ab der Version 8 eine Bean-Validierung über Annotationen der Bean Validation 2.0 anbieten muss. Beispielsweise lassen sich nun Typ-Argumente annotieren (`List<@Positive Integer> positiveNumbers`).

1.6.16 JAX-WS 2.2

JAX-WS stellt die Standard-Java-API für XML-Webservices dar. Eine andere Bezeichnung lautet *Java API for Web Services over SOAP*.

Wenn JAX-WS als Webservice im Einsatz ist, werden die Daten in einer XML-Struktur – oder genauer gesagt: in SOAP – verschickt. In den ersten Jahren der Erfolgsgeschichte von Webservices basierte die Kommunikation zwischen Clients und Servern grundsätzlich nur auf SOAP. Der Begriff SOAP galt ehemals als Abkürzung für *Simple Object Access Protocol*. Heutzutage steht der Begriff SOAP aber für sich. SOAP setzt eine Schnittstellendefinition voraus, die mithilfe der *Web Services Description Language* (*WSDL*) realisiert wird.

1.6.17 JAX-RS 2.1

Neben SOAP entstand im Laufe der Jahre ein weiteres Protokoll mit dem Namen *Representational State Transfer* (*REST*). REST wurde erstmalig in einer Dissertation von Thomas Roy Fielding erwähnt. Fielding ist auch einer der Erfinder des HTTP-Protokolls und ehemaliger Vorsitzender der *Apache Software Foundation*. REST kommt der Idee von lose gekoppelten Systemen noch näher als SOAP, da es auf die Abhängigkeit von aufwendigen Schnittstellenbeschreibungen mit XML verzichtet und stattdessen die originären HTTP-Request-Typen (beispielsweise `GET`, `PUT`, `POST` und `DELETE`) nutzt.

Die mit Java EE 8 hinzugekommene JAX-RS-Version 2.1 unterstützt Server-sent Events, womit Daten vom Server zum Client geschickt werden können. Eine neue Reactive API ermöglicht ferner, mit ReactiveX Frameworks wie RxJava asynchron und ereignisbasiert zu interagieren.

1.7 Los geht's mit der Praxis

In diesem Kapitel habe ich Ihnen einen ersten Überblick gegeben und Ihnen gezeigt, wie Java-Technologien von einem Container-Modul-Komponenten-Konzept durchgängig begleitet werden. Für Java EE-Anwendungen ist dieses Konzept von großer Bedeutung, da die Java-Komponenten über ihre Container miteinander kommunizieren. Ferner haben Sie erfahren, dass Java EE-Anwendungen nach dem MVC-Entwurfsmuster organisiert und darüber hinaus in horizontalen Schichten geordnet werden.

Da Sie nun ungefähr wissen, was Sie in den einzelnen Teilen des Buches erwartet, geht es im nächsten Kapitel schon mit der Praxis los. Wenn Sie als Java EE-Anfänger das eine oder das andere noch nicht vollends verstanden haben sollten, ist das überhaupt kein Problem, denn die einzelnen Themen werden noch in praktischen Übungen eingehend behandelt. Ihre manuelle Programmierung ist wichtig, denn erst durch das eigene aktive Handeln in den folgenden Kapiteln werden Sie die soeben gesehene »schwere Kost« wirklich verstehen. Schalten Sie deshalb gleich Ihren Computer ein. Am besten brühen Sie sich noch eine Tasse frischen Kaffee auf (schließlich handelt es sich ja um *Java*, was die amerikanische Bezeichnung für eine besondere Kaffeebohnen-Sorte ist).

Kapitel 9
Java Server Faces

»Der Neurotiker zieht sein bekanntes Unglück dem unbekannten Glück vor.«
(Sprichwort)

JavaServer Faces (*JSF*) ist ein High-Level-Framework für Java-basierte Webanwendungen. Wie bei der Low-Level-Technologie Servlets handelt es sich also wieder um die Entwicklung des Frontends einer Java EE-Anwendung. Allerdings bezeichnet die Java EE-Spezifikation die Technologie JSF als High-Level-Framework, denn mithilfe einer weiteren Abstraktionsschicht wird die Entwicklung des Frontends weiter vereinfacht. JSF bietet jedoch ein anderes Paradigma als Servlets an, denn während man bei Servlets von einem anfragebasierten Framework spricht, betrachtet man JSF als ein komponentenbasiertes Framework.

Beim anfragebasierten Framework liegt der Fokus auf dem HTTP-Request des Webbrowsers. Jeder HTTP-Request wird als eigene Aktion betrachtet. Deshalb spricht man bei einem anfragebasierten Framework auch von einem *Action-based Web-Framework*. Beim *Action-based Web-Framework* stellt die Anwendung zunächst die Intention der Anfrage fest, indem sie die mitgelieferten Parameter auswertet, um hierauf die passende HTTP-Response zu liefern. Beispielsweise wird im Servlet die Generierung der clientseitigen Präsentationsschicht mithilfe der manuellen Entwicklung von HTML, JavaScript und CSS-Bestandteilen programmiert.

Das Ziel eines komponentenbasierten Frameworks ist hingegen, den Entwickler von dieser Tätigkeit zu entlassen. Während der HTTP-Request im Verborgenen verarbeitet wird, stehen die JSF-Komponenten als High-Level-Abstraktionsschicht im Fokus.

Die Entwicklung von JSF begann im Jahr 2001. Das Final Release der Version 1.0 erschien im Jahr 2004. Anfangs gehörte JSF noch nicht zum Java EE-Standard. Nachdem jedoch wichtige Erkenntnisse und neue Erfindungen aus der Open-Source-Gemeinde in die JSF-Version 1.2 eingeflossen waren, wurde es schließlich im Jahr 2006 als fester Bestandteil in Java EE 5 integriert. Der eigentliche Durchbruch kam aber erst mit der Spezifikation Java EE 6, als die *Java Server Faces* in Version 2.0 aufgenommen wurden. JSF 2.0 brachte so viele Vereinfachungen mit sich, dass es seither als Standard-Komponenten-Framework für das Frontend einer Java EE-Anwendung nicht mehr wegzudenken ist. Über die Jahre wurde JSF immer weiter optimiert und mit neueren Webtechnologien angereichert. Beispielsweise wurde Ajax eingebaut, und Webflows wurden eingeführt, mit denen der Sitzungszustand (Session-State) über mehrere Anfragen hinweg beibehalten wird.

Bei der Java EE-Version 8 wurde die aktuelle JSF-Version 2.3 eingearbeitet. JSF 2.3 bringt zahlreiche Verbesserungen. Insbesondere lag der Fokus in der Unterstützung der Features von Java SE 8 und CDI. Die Spezifikation können Sie von folgender URL herunterladen: *https://jcp.org/en/jsr/detail?id=372*.

Wenn Sie an dieser Stelle des Buches angelangt sind, brauche ich es nicht mehr zu erwähnen, dass auch diese Spezifikation von Softwareherstellern als Implementierung realisiert werden muss, damit sie als Technologie verwendet werden kann. Die Firma Oracle bietet eine eigene Referenzimplementierung an, die sich *Mojarra* nennt. Dieser Implementierung werden wir im Buch den Vorzug geben. Als wichtigstes Konkurrenzprodukt gilt *MyFaces*. MyFaces stammt aus der Schmiede der *Apache Software Foundation*.

Ein markantes Merkmal von JSF gegenüber der Servlet-Variante ist, dass für die clientseitige Präsentationsschicht statt JSPs *Facelets* eingesetzt werden. Grundsätzlich können auch JSPs verwendet werden. Dies war sogar einmal standardmäßig vorgesehen, denn JSPs stellten anfangs die Default-*View-Definition-Language* (*VDL*) von Java Server Faces dar. Mit der JSF-Version 2.0 (Java EE 6) legte man jedoch fest, dass die JSPs durch Facelets ersetzt werden sollen.

Facelets basieren auf der XML-Technologie und wurden im Jahre 2005 im Rahmen des JSR 252 von Jacob Hookom entwickelt. Genauso wie bei einer JSP können auch bei einem Facelet die Geschäftsdaten mithilfe der *Expression Language* verarbeitet werden. Die Expression Language für Facelets wird *JSF-EL* genannt. Im Unterschied zur JSP-EL verwendet man bei der JSF-EL zur Einleitung kein Dollarzeichen, sondern eine Raute. JSF-EL gehört genauso wie JSP-EL zur *Unified EL 3.0*.

Facelets bieten einige Vorteile gegenüber den Java Server Pages. Die wichtigsten Argumente für Facelets sind:

- **Facelets sind performanter.**
  Weil JSPs erst zur Laufzeit erzeugte Servlets sind, ist die automatische Zwischenspeicherung der Ansicht sehr umständlich. Hiervon bekommt der Entwickler der View-Komponenten zwar zunächst nichts mit, aber diese Altlast führt zu einem trägen Laufzeitverhalten.
- **Facelets sind komfortabler.**
  Attribute von benutzerdefinierten JSP-Tags müssen explizit in einer TLD definiert sein. Dies ist umständlich. Bei Facelets resultieren Attribute hingegen ganz einfach daraus, dass man eine entsprechende Property in der UI-Komponente deklariert.
- **Facelets bieten das Templating.**
  Um die Ansicht kachelartig aufzubauen, können Sie bei Facelets eine spezielle Template-Technik verwenden (ich werde sie in Abschnitt 9.8 zeigen.) Dieses sogenannte *Templating* gehört bei Facelets zu den Kernaufgaben eines Entwicklers. Bei JSPs ist hierfür ein zusätzliches Framework wie *Tiles* erforderlich.

Innerhalb der Facelets können Sie verschiedene UI-Komponenten der JSF-Standardbibliothek einsetzen. Sie wurden in folgende zwei Pakete unterteilt (Tabelle 9.1).

Bezeichnung/URI	Präfix	Beschreibung
JSF-HTML-UI-Komponenten *http://xmlns.jcp.org/jsf/html*	`h`	Tags zur grafischen Darstellung der Benutzeroberfläche
JSF-Core-UI-Komponenten *http://xmlns.jcp.org/jsf/core*	`f`	Tags, die unabhängig von der Darstellung sind. Beispielsweise gehören hierzu die Konvertierung und die Validierung der Geschäftsdaten oder auch die ActionListener.

Tabelle 9.1 Die Pakete für die UI-Komponenten

Zusätzlich zu den oben aufgelisteten UI-Komponenten können sogenannte *HTML5-friendly Pass-through-UI-Komponenten* und HTML5-freundliche Pass-through-Attribute verwendet werden.

Bezeichnung/URI	Präfix	Beschreibung
JSF-Pass-through-UI-Komponenten *http://xmlns.jcp.org/jsf*	`p`	Tags für die HTML5-freundliche Markup-Technik
JSF-Pass-through-Attribute *http://xmlns.jcp.org/jsf/passthrough*	`jsf`	Tags für die HTML5-freundliche Markup-Technik

Tabelle 9.2 Die Pakete für die HTML5-friendly Pass-through-UI-Komponenten und HTML5-friendly Pass-through-Attribute

Außerdem besitzen Facelets eigene UI-Komponenten, die beispielsweise das Templating ermöglichen.

Bezeichnung/URI	Präfix	Beschreibung
JSF-Templating *http://xmlns.jcp.org/jsf/facelets*	`ui`	Tags für das Templating

Tabelle 9.3 Die Pakete für das UI-Templating

Genauso wie bei JSPs wird hin und wieder auch die *Java Standard Tag Library* (*JSTL 1.2*) verwendet. In vielen Fällen können Sie auf den Gebrauch von JSTL-Tags aber verzichten, denn in der Regel bieten die JSF-UI-Komponenten die gleiche Funktionalität bei einer komfortableren Kodiermöglichkeit an.

Bezeichnung/URI	Präfix	Beschreibung
JSTL-Core *http://xmlns.jcp.org/jsp/jstl/core*	`c`	JSTL 1.2-Core-Tags
JSTL-Functions *http://xmlns.jcp.org/jsp/jstl/functions*	`fn`	JSTL 1.2-Function-Tags
JSTL SQL *http://xmlns.jcp.org/jsp/jstl/sql*	`sql`	JSTL 1.2-SQL-Tags
JSTL XML *http://xmlns.jcp.org/jsp/jstl/xml*	`xml`	JSTL 1.2-XML-Tags

Tabelle 9.4 Die JSTL-Pakete

9.1 Ein erstes Beispiel

In diesem Abschnitt werden wir uns nicht weiter mit trockener Theorie auseinandersetzen, sondern stattdessen direkt mit der Praxis beginnen, denn indem Sie aktiv und mit Spaß an einer JSF-Anwendung arbeiten, wird der Lernerfolg erhöht. In dem Programmierbeispiel werden Sie die Willkommensseite des durchgehenden Programmierbeispiels Onlineshop entwickeln. Es handelt sich also um User-Story 0 der durchgehenden Onlineshop-Anwendung (»Als Kunde möchte ich willkommen geheißen werden.«). Hierbei werden wir uns von Eclipse und seinen Wizards helfen lassen. Für das Beispiel werden wir nicht das dynamische Webprojekt aus den vergangenen Kapiteln verwenden. Stattdessen werde ich zeigen, wie Sie ein dynamisches Webprojekt mit JSF »auf der grünen Wiese« erzeugen.

Falls Sie noch die Projekte aus den vorherigen Kapiteln in Ihrem Eclipse vorliegen haben, müssen Sie das alte Projekt *onlineshop-web* nun umbenennen oder entfernen.

Hinweis

Wenn Sie die vorherigen Kapitel übersprungen haben, sind einige Vorbereitungen zu treffen, denn Sie brauchen einen Java EE 8-konformen Server wie *GlassFish 5.0*. Die Installation von GlassFish habe ich in Kapitel 2, »Die Entwicklungsumgebung«, gezeigt. GlassFish 5.0 enthält von Haus aus die JSF-Implementierung namens *Mojarra*.

Neben der Installation von GlassFish müssen Sie den Java EE Server als Server-Runtime und auch die sogenannten *GlassFish Tools* in Eclipse einbinden. Auch diese Schritte habe ich in Kapitel 2 beschrieben.

9.1.1 Die Erstellung eines JSF-Projekts

In Eclipse erzeugen Sie nun ein neues dynamisches Webprojekt mit dem Namen *onlineshop-web*.

In Abbildung 9.1 sehen Sie, dass beim Wizard für das dynamische Webprojekt bei Configuration der Eintrag Default Configuration for GlassFish 5.0 ausgewählt ist. Diese Einstellung ist für unseren GlassFish Server richtig.

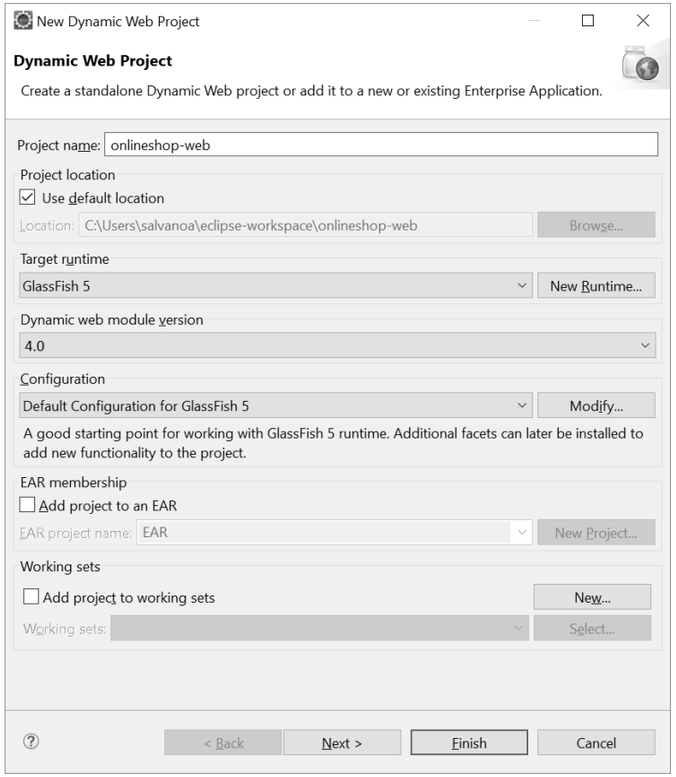

Abbildung 9.1 Das dynamische Webprojekt

Jetzt müssen wir das dynamische Webprojekt noch mit JSF ausstatten. Das bedeutet, dass wir dem Eclipse-Projekt ein entsprechendes *Facet* hinzufügen müssen. Um die Facets anzupassen, klicken Sie im Bereich Configuration auf den Button Modify.

In der Ansicht Project Facets werden auf der linken Seite die Facets angezeigt, die in Ihrer Eclipse-Version zur Auswahl stehen. Klicken Sie auf die Checkbox JavaServer Faces, um das JSF-Facet zu aktivieren. Die Version sollte hierbei auf 2.3 gesetzt werden können.

Hinweis

Bei meiner Eclipse-Oxygen-Version 2 wurde der Wizard noch nicht auf die aktuelle Java EE 8-Version angehoben. Weil die Eclipse Foundation aber derzeitig an der Behebung dieses Problems arbeitet, zeige ich Ihnen in den Screenshots die Java EE 8-Versionen, die Sie zum Zeitpunkt des Buchkaufs bei Ihrer Eclipse-Version vorfinden sollten.

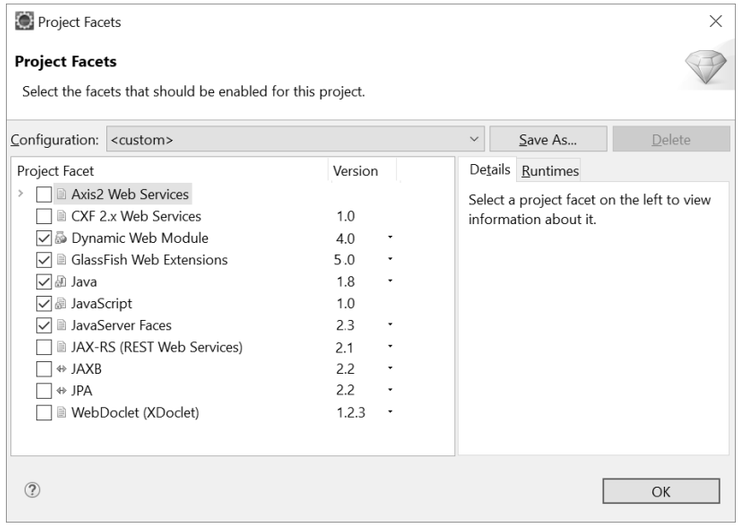

Abbildung 9.2 Über das Hinzufügen und Entfernen von Facelets können Sie den Charakter eines Eclipse-Projekts verändern.

Wenn Sie zur Bestätigung auf den Button OK klicken, gelangen Sie wieder in das Fenster Dynamic Webproject zurück. Klicken Sie dort auf Next. Hierdurch öffnet sich die Ansicht Java. Wenn Sie erneut auf Next klicken, befinden Sie sich in der Ansicht Web Module. Mit nochmaligem Mausklick auf Next sind Sie nun in der Ansicht JSF Capabilities angekommen.

Mit der ersten Auswahlbox wird auf die *JSF Implementation Library* hingewiesen. In der Einführung habe ich bereits angemerkt, dass die Java-Server-Faces-Spezifikation ja erst von einem Hersteller implementiert werden muss, damit sie als Library verwendet werden kann. Da wir in einer vorherigen Auswahl den GlassFish Server als Server-Runtime gesetzt haben, lässt sich an dieser Stelle die GlassFish System Library auswählen (siehe Abbildung 9.3). Hierdurch wird die JSF 2.3-Referenzimplementierung *Mojarra* genutzt.

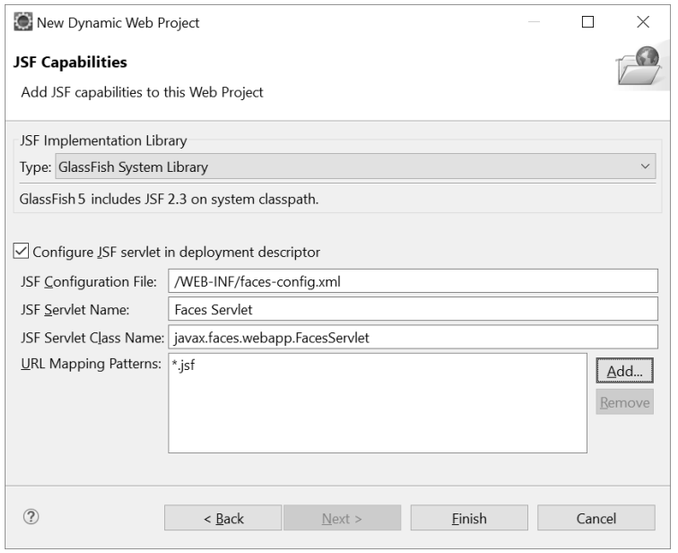

Abbildung 9.3 Die Ansicht »JSF Capabilities«

Hinweis

Wenn Sie in Ihrer Entwicklungsumgebung nicht mit GlassFish arbeiten, wird Ihnen dieser Eintrag nicht angeboten. Stattdessen erhalten Sie dann im Fenster eine Warnmeldung. Zum Beispiel könnte es sein, dass Sie in Ihrem Arbeitsumfeld den Apache Tomcat einsetzen. Für diesen Fall bietet Ihnen Eclipse eigene Wizards an, über die Sie die JSF-Bibliotheken aus dem Internet herunterladen können. Allerdings ist dieser Weg nicht zu empfehlen. Stattdessen nutzen Sie in diesem Fall das Build-Automatisierungs-Tool Maven. Lesen Sie hierzu Abschnitt 2.8, »Maven«.

Im unteren Teil des Fensters können Sie über eine Checkbox den Bereich Configure JSF servlet in deployment descriptor anschalten. Setzen Sie ein Häkchen in diese Checkbox, da wir hierüber die Java Server Faces konfigurieren werden.

Im Eingabefeld URL-Mapping Patterns wird der Eintrag *"/faces/*"* angezeigt. Hiermit weist der Wizard darauf hin, dass alle Anfragen, die den Pfad */faces* enthalten, über das Faces-Servlet erfolgen sollen. Für unsere Anwendung werden wir aber einen gängigeren Weg beschreiten. Selektieren Sie diesen Eintrag, und klicken Sie auf Remove. Über Add erstellen Sie einen neuen Eintrag mit dem URL-Mapping-Pattern *"*.jsf"*. Hiermit sorgen wir dafür, dass alle Anfragen, die die Endung .jsf aufweisen, über das JSF-Framework laufen.

Auf einen weiteren wichtigen Bestandteil einer JSF-Anwendung wird im mittleren Bereich des Fensters hingewiesen. Es handelt sich hierbei um die neu zu erstellende JSF-Konfigura-

tionsdatei *faces-config.xml*. Obwohl diese Konfigurationsdatei optional ist, werden wir sie dennoch erzeugen. Weiter unten werde ich auf diese Datei zurückkommen.

Mit einem Klick auf FINISH wird das JSF-Projekt abschließend angelegt.

9.1.2 Die Anpassung der »web.xml«

Wenn Sie sich jetzt den Deployment-Deskriptor */WEB-INF/web.xml* anschauen, werden Sie feststellen, dass der Wizard ihn so fertiggestellt hat, dass alle Anfragen auf das URL-Pattern *"*.jsf"* zunächst über die Klasse *javax.faces.webapp.FacesServlet* laufen.

Damit die URL des initialen Aufrufs für den Onlineshop möglichst kurz ist, sollten wir der *web.xml* auch noch ein Welcome-File-Element hinzufügen, das automatisch den JSF-Request-Response-Prozess *index.jsf* auslöst:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">
  <display-name>onlineshop-web</display-name>
  <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.jsf
        </welcome-file>
    </welcome-file-list>
</web-app>
```

Listing 9.1 »web.xml« mit »index.jsf« als Willkommensdatei

9.1.3 Die Erzeugung des Facelets mithilfe von Eclipse

Im nächsten Schritt legen wir die View-Komponente an. Die View-Komponente wird standardmäßig als Facelet bereitgestellt. Eclipse unterstützt die Verwendung von Facelets und

der darin verwendbaren UI-Komponenten durch eigene Designer und Wizards. Um sich bei der Erzeugung eines Facelets helfen zu lassen, klicken Sie im Hauptmenü auf FILE • NEW • OTHER. Im Wizard-Fenster öffnen Sie den Ordner WEB und selektieren dort den Eintrag HTML-FILE. Nach einem Klick auf NEXT erscheint das Fenster NEW HTML FILE (siehe Abbildung 9.4). Tragen Sie dort unter FILE NAME den Dateinamen »index.xhtml« ein.

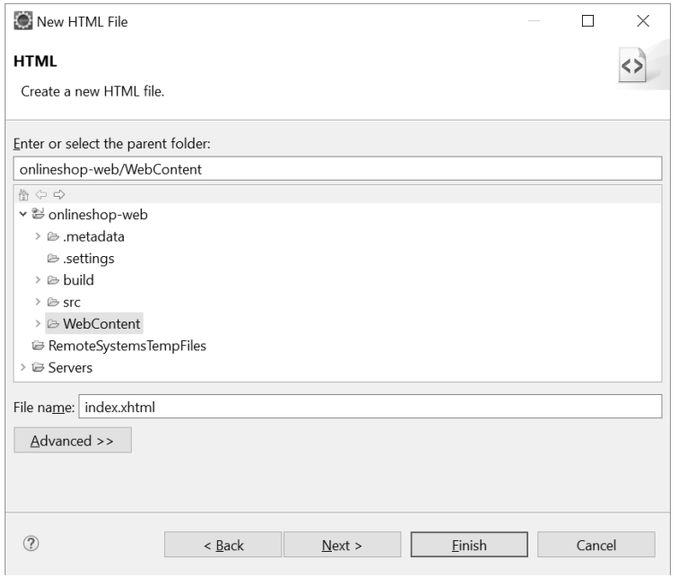

Abbildung 9.4 Die Datei »index.xhtml«

Mit einem Mausklick auf NEXT gelangen Sie zur Ansicht SELECT HTML TEMPLATE (siehe Abbildung 9.5), wo Sie sich zwischen verschiedenen HTML- und XHTML-Templates entscheiden müssen. Da einerseits Facelets XHTML-Dokumente sind und wir andererseits HTML5-Markup-Elemente einsetzen möchten, stellt sich die Frage, welches Template wir für unsere Zwecke nutzen sollten. Die Antwort: Beide Templates sind geeignet. Per Konvention werden jedoch XHTML-Dokumente verwendet.

Weil XHTML-Dateien zu den XML-Dokumenten gehören, ist es wichtig, dass ihr Inhalt im Sinne der XML-Technologie *wohlgeformt* ist.

Das bedeutet:

- Genau ein Root-Element muss vorhanden sein, von dem aus die DOM-Hierarchie des Dokuments streng eingehalten ist.
- Zu jedem öffnenden Tag muss ein schließendes Tag formuliert sein.
- Allen Element-Attributen muss ein in Anführungsstrichen gesetzter Wert zugewiesen werden.

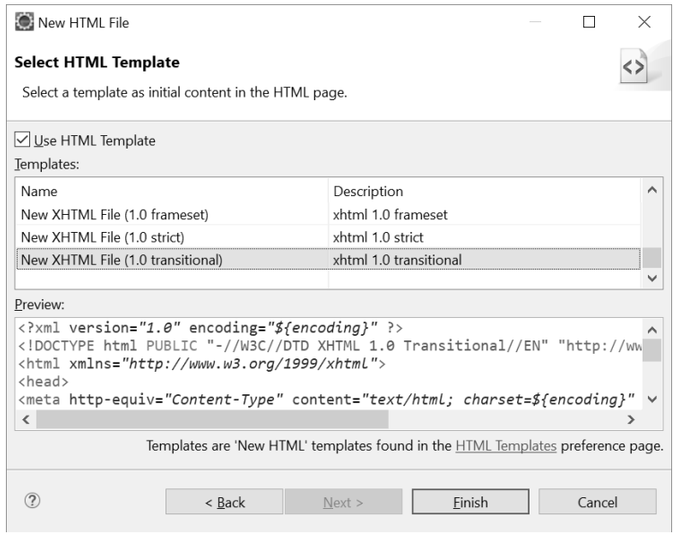

Abbildung 9.5 Die Erstellung der XHTML-Datei

Normalerweise beginnen XML-Dateien mit einer XML-Deklaration. Also beispielsweise so:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Die XML-Version 1.0 und die Zeichenkodierung UTF-8 entsprechen dem Default. Weil eine XML-Deklaration nicht zwingend vorhanden sein muss und wir ohnehin die Zeichenkodierung auf UTF-8 belassen möchten, könnte sie theoretisch auch weggelassen werden. Aus diesem Grund verzichtet der Eclipse-Wizard manchmal auf eine explizite XML-Deklaration. In diesem Buch werden wir so vorgehen, dass wir die XML-Deklaration stets setzen werden, obwohl das W3C an dieser Stelle flexibel ist. Aber auch sonst hat das W3C bereits alles Erdenkliche getan, damit wir in Bezug auf die Kombination aus XHTML 1.0 und dem HTML5-Standard nicht auf Probleme stoßen. Beispielsweise sind HTML5-Markup-Elemente per Default im XHTML-1.0-Namensraum enthalten. In unserem konkreten Fall werden wir dennoch den Eintrag New XHTML File (1.0 Transitional) selektieren, denn dies entspricht nach wie vor der üblichen Vorgehensweise.

Mit einem Mausklick auf Finish erzeugt Eclipse eine XHTML-Datei und zeigt diese in einem *Defaulteditor* an. Allerdings ist dieser Editor für unsere Zwecke nicht zweckmäßig. Stattdessen benötigen wir den *Webpage-Editor*. Deshalb schließen Sie den Defaulteditor mit der Datei *index.xhtml* und klicken im Projekt-Explorer mit der rechten Maustaste auf die Datei *index.xhtml*. Im Kontextmenü klicken Sie auf Open With und im Untermenü auf Webpage-Editor.

Der Webpage-Editor bietet zwei verschiedene Ansichten zur Auswahl an, nämlich Design und Preview. Die Ansicht Design ist zweigeteilt. Im unteren Bereich der Design-Ansicht

können Sie den Quelltext editieren. Auf der oberen Seite können Sie die Webpage mit der Maus gestalten. Die Preview zeigt das Endergebnis an. Gemeinsam mit der View Palette (mit den Tag-Librarys) erhalten wir einen vollständigen GUI-Designer, mit dem wir die Ansicht der View-Komponente grafisch modellieren können.

Bevor es gleich mit der eigentlichen Modellierung losgeht, entfernen Sie das `<head>`- und das `<body>`-Element aus der Datei, denn diese statischen HTML-Markup-Elemente werden wir nun durch dynamische JSF-UI-Komponenten ersetzen.

```
<?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">

</html>
```

Listing 9.2 index.xhtml

Öffnen Sie nun zusätzlich über Window • Show View • Other im Ordner General die View Palette (siehe Abbildung 9.6). Diese View bietet eine Menge an UI-Komponenten an, auf die ich im Folgenden eingehen werde.

Als Nächstes entfernen Sie im Quelltext das `<head>`-Element und das `<body>`-Element, denn JSF bietet hierfür eigene UI-Komponenten an, die wir aus der View Palette in die Datei *index.xhtml* hineinziehen werden.

In der Palette sehen Sie mehrere Reiter. Sie tragen beispielsweise die Aufschrift Facelet Core, HTML 4.0, JSF Core oder JSF HTML. Für diesen Abschnitt sind zunächst nur die Reiter JSF HTML und JSF Core von Bedeutung. Öffnen Sie den Reiter mit der Aufschrift JSF HTML.

Als Erstes werden wir die Head-Komponente in die Datei ziehen. Diese UI-Komponente ist vor allem für die Nutzung von JSF-Komponenten für Ajax von Bedeutung, denn ohne eine Head-Komponente verweigert der Ajax-Mechanismus von JSF ganz einfach seinen Dienst.

Ziehen Sie also die UI-Komponente Head in den Editor-Bereich hinein. Beobachten Sie hierbei, wie Eclipse den Quelltext für die UI-Komponenten automatisch erstellt, denn es sollten anschließend ein öffnendes und ein schließendes `h:head`-Tag in die Datei eingefügt worden sein. Auch die Namensraumdeklarationen für die JSF-UI-Komponenten werden im HTML-Element hinzugefügt. Innerhalb der Tags schreiben Sie manuell die Tags `<title>` und `</title>` hinein. Danach ziehen Sie hinter das öffnende Tag eine JSF-Output-Text-Komponente hinein. Als `value` tragen Sie `"Onlineshop"` ein. Außerdem schreiben Sie manuell das `<meta>`-Tag für das UTF-8-Encoding hinzu.

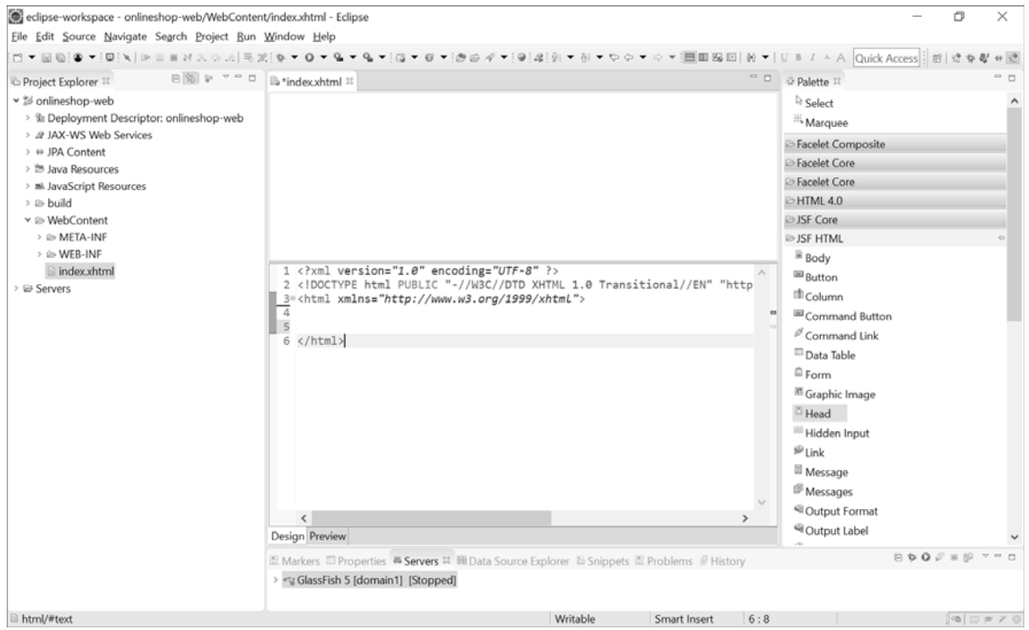

Abbildung 9.6 Die View »Palette« für das Setzen von UI-Komponenten

Dann ziehen Sie eine BODY-Komponente von der Palette in die Preview und setzen dort manuell ein `<h1>`-Element hinein. Innerhalb des `<h1>`-Elements fügen Sie wieder eine JSF-Output-Komponente mit dem Value `Onlineshop` hinzu. Unterhalb des `<h1>`-Elements schreiben Sie `Willkommen im Onlineshop`. Der Quelltext der Datei *index.xhtml* sollte anschließend wie folgt aussehen:

```
<?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:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title><h:outputText value="Onlineshop" /></title>
        <meta
            http-equiv="Content-Type"
            content="text/html; charset=UTF-8" />
    </h:head>
    <h:body>
        <h1>
```



```
            <h:outputText value="Onlineshop" />
        </h1>
        Willkommen im Onlineshop!
    </h:body>
</html>
```

Listing 9.3 index.xhtml

9.1.4 Das Facelet einbauen

In Listing 9.4 sehen Sie ein einfaches Facelet, das den Besucher im Onlineshop willkommen heißt.

```
<?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:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title><h:outputText value="Onlineshop" /></title>
    </h:head>
    <h:body>
        <h1 class="title">
            <h:outputText value="Onlineshop" />
        </h1>
        Willkommen im Onlineshop!
    </h:body>
</html>
```

Listing 9.4 index.xhtml

9.1.5 Die Anwendung deployen

Um die Webanwendung auf den GlassFish Server zu deployen, sollte eine GlassFish-Server-Instanz installiert und das Projekt dieser Instanz als Anwendung hinzugefügt worden sein. All das wurde bereits in vorangegangenen Kapiteln gezeigt. Nach dem Deployment rufen Sie im Browser die URL *http://localhost:8080/onlineshop-web/index.jsf* auf.

9.2 Ein Durchstich mit JSF und JPA

In diesem Abschnitt programmieren Sie User-Story 2 der durchgehenden Onlineshop-Anwendung (»Als Kunde möchte ich mich registrieren«). Der Abschnitt beginnt mit einem

Vergleich der Servlet-Technologie und der JSP-Technologie mit der Technologie JSF, denn durch die Gegenüberstellung von Low Level und High Level wird deutlich, welcher Quelltext mit JSF automatisch erzeugt wird, sodass wir auf eine manuelle Programmierung verzichten können.

Java Server Faces erleichtern die Programmierung der Benutzerschnittstelle einer Java EE-Anwendung, weil sich mit ihnen vieles erübrigt, was mit Servlets und JSPs manuell programmiert werden müsste. Diese Vereinfachung wird bei der Registrierung des Onlineshops besonders deutlich. In Abbildung 9.7 sehen Sie, wie das Frontend einer Kundenregistrierung mit einem Servlet und einer JSP programmiert würde.

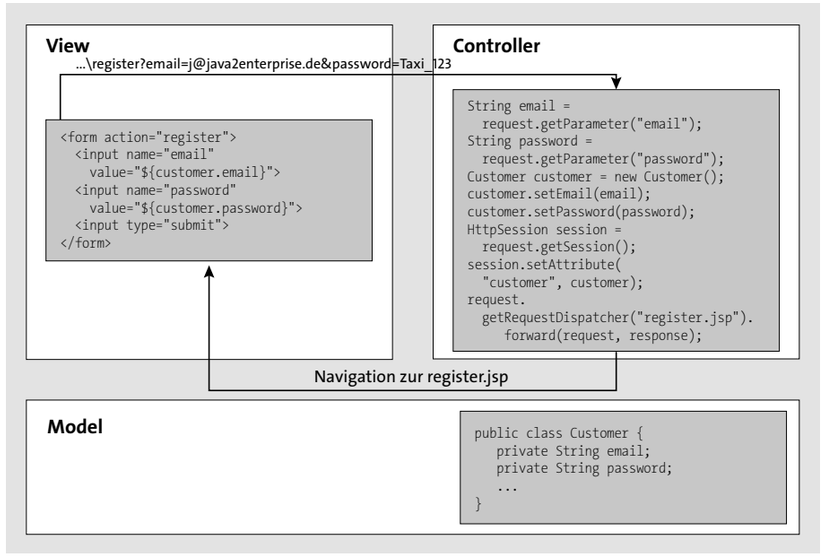

Abbildung 9.7 Das MVC-Pattern mit Servlets und JSPs

Die Komponenten weisen das klassische Entwurfsmuster *Model-View-Controller* (*MVC*) auf. Dabei wird das Servlet als *Controller* eingesetzt. Das Controller-Servlet nimmt die Parameter der Anfrage entgegen und erstellt hiermit eine *JavaBean*. Die JavaBean wird als *Model* in einen Gültigkeitsbereich (beispielsweise in die Session) gesetzt. Dies ist aber nicht die einzige Aufgabe des Controller-Servlets, denn zusätzlich ist es für die Navigation zuständig. In Abbildung 9.7 sehen Sie, wie es den Prozess über einen `RequestDispatcher` an eine JSP weiterleitet. Die JSP ist als *View* für die Präsentation der Daten verantwortlich. Zu diesem Zweck können Sie innerhalb einer JSP die JSP-EL verwenden. In der View zeigen wir aber nicht nur die Geschäftsdaten an, sondern ermöglichen auch die Interaktion mit dem Benutzer. Hierfür werden üblicherweise HTML-Formulare verwendet.

In Abbildung 9.8 sehen Sie die Kundenregistrierung in der JSF-Variante.

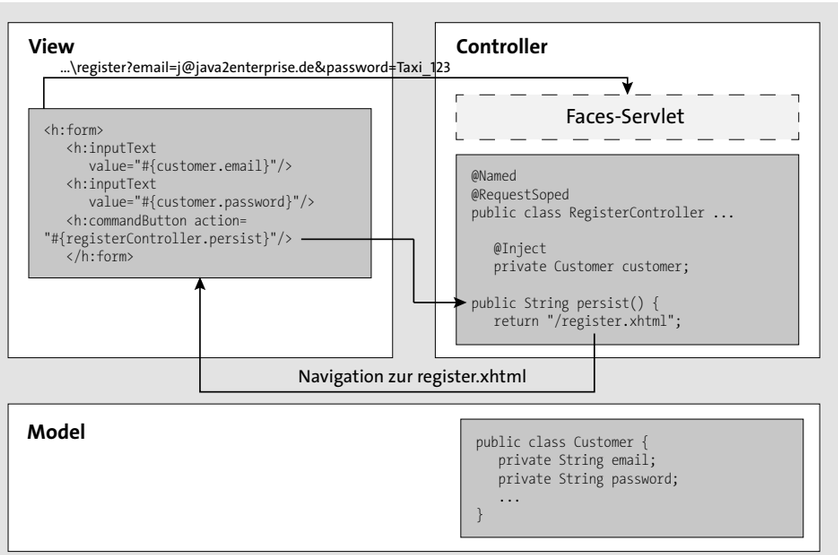

Abbildung 9.8 Das MVC-Pattern mit Java Server Faces

Mit JSF ist der Programmablauf zwischen Model, View und Controller ähnlich, wie ich es in Abbildung 9.7 mit Servlets und JSPs gezeigt habe.

In Abbildung 9.8 sehen Sie auf der linken Seite ein öffnendes und ein schließendes `<h:form>`-Element. Mit diesem Element wird im Facelet ein HTML-Formular gerendert. Das HTML enthält zwei Eingabefelder und einen Command-Button. Dies wurde mit den Elementen `<h:inputText>` und `<h:commandButton>` festgelegt.

In den `<h:inputText>`-Elementen wird über einen EL-Ausdruck auf die Property `customer` der Klasse `RegisterController` referenziert.

Im Command-Button wurde das Attribut `action` mit dem EL-Ausdruck "`#{registerController.persist()}`" programmiert. Hiermit teilt der Command-Button mit, dass die Methode `persist()` der Klasse `RegisterController` ausgeführt werden soll, wenn der Benutzer den Button betätigt.

9.2.1 Die Backing Bean erstellen

Kommen wir nun zur rechten Seite von Abbildung 9.8, wo sich die *Controller*-Komponente befindet. MVC-Puristen bezeichnen diese Komponente auch als *Handler* oder *Vermittler*. In der Literatur des Herstellers Oracle wird meistens der Begriff *Backing Bean* gebraucht. Und auch in diesem Buch verwende ich den Begriff *Backing Bean*.

Die Backing Bean ist normalerweise für die Weiterleitung der fachlichen Belange aus der *View* und für die Kommunikation mit dem Backend verantwortlich. Deshalb muss sie alle Properties und alle Methoden für die View-Komponente definieren.

Ein auffallendes Merkmal hierbei ist, dass im Vergleich zur Servlet- und JSP-Variante im Controller die Aufgabe der manuellen Erzeugung der JavaBean-Objekte wegfällt. Mit Servlets und JSPs hatten wir an dieser Stelle Request-Parameter entgegengenommen und hieraus JavaBean-Objekte erzeugt. Auf diese Kodierung können wir nun verzichten, da sie vom Framework automatisch übernommen wird. Die Property `customer` der Backing Bean wird also automatisch initialisiert und mit den Werten der View gefüllt, ohne dass wir hierfür auch nur eine einzige Zeile Quelltext schreiben müssen.

Selbst die Navigation läuft anders ab, denn sie ist nun viel einfacher. Im Grunde genommen landet der ankommende HTTP-Request nicht mehr bei einem individuell programmierten Servlet, sondern bei dem sogenannten *Faces-Servlet*. Dieses Servlet wird vom JSF-Framework automatisch zur Verfügung gestellt. Allerdings geschieht dies unbemerkt im Hintergrund, sodass sich der Entwickler hierum nicht kümmern braucht. Für den Entwickler ist es viel interessanter, dass im Programmablauf die Methode aufgerufen wird, die im Attribut `action` des Command-Buttons eingetragen ist. Der Begriff aus dem Fachjargon für solche Methoden lautet *Aktionsmethode*. Der englische Fachbegriff heißt *Action Method*. In unserem Beispiel nennt sich die Aktionsmethode `persist()`. In der Aktionsmethode legt der Entwickler das weitere Vorgehen fest. Zuletzt kann er über einen Rückgabewert darüber verfügen, wohin im Fortlauf navigiert werden soll. In Abbildung 9.8 navigiert JSF zur View */register.xhtml*, weil dies als Zeichenkette so eingetragen ist.

9.2.2 Ein POJO als JPA-Entity anlegen

Genauso wie in den Programmierbeispielen vergangener Kapitel werden wir auch hier wieder die Geschäftsdaten durch POJOs modellieren. Dabei werden wir die POJOs als JPA-Entities anlegen, so wie ich es in Kapitel 8, »Die Java Persistence API«, gezeigt habe. Dieser Abschnitt ist für das Studium von JSF nicht zwingend erforderlich. Sie können die hier gezeigten Schritte also auch überspringen. Für manche Entwickler ist die Programmierung eines »echten« Beispiels jedoch von Bedeutung, bei dem man auch wirklich Daten in einer relationalen Datenbank abspeichert. Deshalb zeige ich Ihnen in diesem Abschnitt die hierfür erforderlichen Schritte. Das Beispiel setzt voraus, dass Sie die Datenbanktabelle `CUSTOMER` in Ihrer relationalen Datenbank gespeichert haben (die Erstellung der Onlineshop-Datenbank habe ich in Kapitel 6, »Die relationale Datenbank«, gezeigt), sodass wir nun über die Java Persistence API (JPA) auf sie zugreifen können.

Die JPA habe ich in Kapitel 8, »Die Java Persistence API«, beschrieben. Um die Geschäftsdaten dieses Kapitels mithilfe der JPA in der relationalen Datenbank abzuspeichern, ist dieses Wissen erforderlich. Für den Fall, dass Sie Kapitel 8 übersprungen haben, werde ich Ihnen dennoch zeigen, wie Sie die JPA ohne den großen Umweg über Kapitel 8 einbeziehen können.

Als wichtigste Voraussetzung erfordert die JPA, dass die *.jar*-Bibliothek einer JPA-Implementierung im Klassenpfad eingebunden ist. Wenn Sie GlassFish verwenden, ist dies implizit gegeben, denn GlassFish enthält die JPA-Referenzimplementierung *EclipseLink*.

Unser Ziel, die JPA einzubeziehen, ist schnell erreicht. Als ersten Schritt klicken Sie mit der rechten Maustaste auf das dynamische Webprojekt und wählen im Kontextmenü CONFIGURE • CONVERT TO JPA PROJECT... aus. Dabei öffnet sich das PROJECT FACET-Fenster.

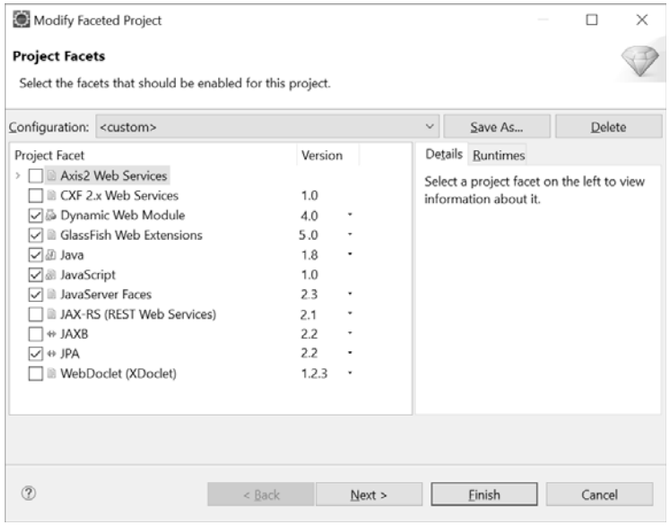

Abbildung 9.9 Das JPA-Facet setzen

Mit einem Klick auf FINISH schließen Sie das Fenster. Eclipse hat hierbei die JPA-Konfigurationsdatei *persistence.xml* in das Unterverzeichnis */META-INF* des Quellordners gespeichert und darin eine Persistenz-Einheit angelegt, die als Namen den Bezeichner des Projekts trägt. Die Persistenz-Einheit trägt deshalb den Namen *onlineshop-web*. Wir werden uns später auf den Namen der Persistenz-Einheit beziehen, wenn wir in der Backing Bean über die JPA auf die Datenbank zugreifen. Beachten Sie, dass wir die Persistenz-Einheit im letzten Kapitel *onlineshop-jpa* genannt hatten. Diesen Bezeichner werde ich auch in diesem Programmierbeispiel der Einheitlichkeit halber beibehalten und ihn in der Datei *persistence.xml* abändern. Allerdings wird dies in den Backing Beans ohnehin keine Rolle spielen, denn da wir nur eine einzige Persistenz-Einheit vorsehen, brauchen wir ihren Namen in der CDI-Annotation überhaupt nicht zu erwähnen. CDI wird ganz einfach davon ausgehen, dass es sich um die eine Persistenz-Einheit handelt, die ihr vorliegt.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="onlineshop-jpa">
    </persistence-unit>
</persistence>
```

Listing 9.5 persistence.xml

Als Nächstes werden wir die Geschäftsdaten für die Registrierung modellieren. Das bedeutet, dass wir pro Domäne ein POJO erzeugen. Das POJO wird nach den gängigen Regeln einer JPA-Entity erstellt. Damit ist beispielsweise gemeint, dass es für jedes Spaltenfeld über eine Property verfügen wird, die über öffentliche Getter- und Setter-Methoden und über einen Defaultkonstruktor erreichbar ist. Was ein POJO tatsächlich erst in eine JPA-Entity verwandelt, ist die entsprechende JPA-Konfiguration. In Listing 9.6 ist die JPA-Entity `Customer.java` abgebildet. Weitere Erläuterungen zu JPA-Entities erhalten Sie in Kapitel 8, »Die Java Persistence API«. Kopieren Sie die folgende Datei in den Quellordner Ihres dynamischen Webprojekts:

```
package de.java2enterprise.onlineshop.model;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(
        schema = "ONLINESHOP",
        name = "CUSTOMER"
)
@NamedQuery(
        name = "Customer.findAll",
        query = "SELECT c FROM Customer c"
)
public class Customer implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
```



```
@SequenceGenerator(
        name = "CUSTOMER_ID_GENERATOR",
        sequenceName = "SEQ_CUSTOMER",
        schema = "ONLINESHOP",
        allocationSize = 1,
        initialValue = 1
)
@GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "CUSTOMER_ID_GENERATOR"
)
private Long id;

private String email;

private String password;

public Customer() {
}

public Long getId() {
    return this.id;
}

public void setId(Long id) {
    this.id = id;
}

public String getEmail() {
    return this.email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPassword() {
    return this.password;
}

public void setPassword(String password) {
    this.password = password;
}
```

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
            + ((id == null) ? 0 : id.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (!(obj instanceof Customer)) {
        return false;
    }
    Customer other = (Customer) obj;
    if (id == null) {
        if (other.id != null) {
            return false;
        }
    } else if (!id.equals(other.id)) {
        return false;
    }
    return true;
}

public String toString() {
    return id + "-" + email + "-" + password;
}
}
```

Listing 9.6 Customer.java

9.2.3 Eine Backing Bean als Controller programmieren

Eingangs habe ich bereits angemerkt, dass bei einer JSF-Anwendung zu jeder View-Komponente auch eine entsprechende Controller-Komponente bzw. eine Backing Bean erstellt wird. Eine Backing Bean ist vom Prinzip her ein POJO, also eine ganz einfache Java-Klasse.

Obwohl es grundsätzlich möglich ist, auch eine Entity-Klasse des Datenmodells in eine Backing Bean umzuwandeln, setzt man (in der »sauberen« JSF-Softwarearchitektur) eine Backing Bean nicht als Modell der Geschäftsdaten, sondern lediglich zur Steuerung und zur Vermittlung ein.

Auch in den Kapiteln über die Low-Level-Technologien Servlets und JSPs haben wir Controller-Komponenten erstellt. Dort haben wir Servlets programmiert, die sich um alle Belange des HTTP-Requests und der HTTP-Response gekümmert haben. Um die Instanziierung eines Servlets haben wir uns nicht gesorgt. Weil Servlets ohnehin vom Webcontainer verwaltet werden, brauchten wir uns um ihre Erzeugung nicht zu kümmern. Bei Backing Beans sieht das anders aus. Weil Backing Beans keine Servlets, sondern ganz einfache Java-Klassen sind, müssen wir explizit die Anweisung geben, dass der Webcontainer die Verwaltung der Instanzen übernehmen soll.

Zu diesem Zweck boten frühere Java EE-Versionen das Package `javax.faces.bean` der JSF-API an. Diese Variante gilt aber seit Java EE 8 nunmehr als *deprecated*. Stattdessen wird die einheitlich geltende *Context-and-Dependency-Injection*-(*CDI*-)Technologie eingesetzt.

Die Nutzung von CDI

In diesem Abschnitt werde ich zeigen, wie Sie CDI einsetzen, um die Klasse `RegisterController` als verwaltete Bean bereitzustellen. Die generelle Bedeutung von CDI habe ich in Kapitel 1, »Überblick«, angesprochen.

Um CDI für ein Webmodul zu aktivieren, musste in früheren Java EE-Versionen eine bestimmte XML-Datei mit dem Namen *beans.xml* geschrieben werden. Die Datei *beans.xml* wurde in einem Webmodul unterhalb des Ordners */WEB-INF* und in einem EJB-Modul unterhalb des Ordners */META-INF* abgelegt. Auf diese Datei kann aber bereits seit Java EE-Version 7 verzichtet werden.

Dennoch ist die Erstellung der Konfigurationsdatei *beans.xml* auch heute noch manchmal nützlich.

Die XML-Datei *beans.xml* muss das Root-Element `beans` enthalten, das den Namensraum des XML-Schemas *beans_1.1.xsd* avisiert. Listing 9.7 zeigt den Inhalt einer *beans.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans 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/beans_2_0.xsd"
       bean-discovery-mode="annotated">
</beans>
```

Listing 9.7 beans.xml

Beachten Sie bei dem obigen Listing das Attribut `bean-discovery-mode="annotated"`. Hierdurch weisen wir an, dass alle verwalteten Komponenten und die Komponenten, die wir injizieren werden, annotiert sein müssen. Dies entspricht der Defaulteinstellung. Aus diesem Grund könnte die oben gezeigte *beans.xml* auch weggelassen werden – das Ergebnis wäre dasselbe.

In manchen Fällen ist es aber sinnvoll, CDI zu deaktivieren. Hierfür setzen Sie das Attribut `bean-discovery-mode="none"`.

Hinweis

Beim Java EE Server GlassFish können Sie CDI auch während des Deployments deaktivieren. Zu diesem Zweck setzen Sie den Optionsparameter `implicitCdiEnabled` auf `false`.

```
asadmin deploy --property implicitCdiEnabled=false ...
```

Ein anderer Grund, der das Anlegen der *beans.xml* rechtfertigt, ist, dass es beim Programmieren zu lästig sein könnte, jede einzelne Komponente, die man injizieren möchte, mit einer CDI-Annotation zu versehen. Denken Sie beispielsweise an die vielen JPA-Entities, die per Reverse Engineering immer wieder automatisiert erzeugt werden. Wenn Sie den Wunsch hegen, alle JPA-Entities grundsätzlich als Beans injizieren zu können, setzen Sie das Attribut `bean-discovery-mode="all"`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans 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/beans_2_0.xsd"
       bean-discovery-mode="all">

</beans>
```

Listing 9.8 beans.xml

Dies werden wir auch in unserem Onlineshop so vornehmen, da unsere JPA-Entities hierdurch keine CDI-Annotationen benötigen.

Um die Klasse `RegisterController` zur Managed Bean zu erklären, setzt man eine Annotation ein, die gleichzeitig den Gültigkeitsbereich der Backing Bean anzeigt. Die Annotationen für den Gültigkeitsbereich befinden sich im Package `javax.enterprise.context`.

Die CDI-Annotationen hierfür lauten:

- **`@ApplicationScoped`**
  Eine einzige Instanz der Backing Bean ist in der gesamten Applikation global gültig. Damit ist gemeint, dass sie beispielsweise in allen HTTP-Sitzungen verschiedener Benutzer sichtbar und gleichermaßen veränderbar ist.
- **`@SessionScoped`**
  Die Backing Bean bleibt während der gesamten HTTP-Sitzung durchgängig bestehen.
- **`@RequestScoped`**
  Die Backing Bean ist ab dem Eintreffen eines HTTP-Requests bis zum finalen Absenden der HTTP-Response gültig.
- **`@ConversationScoped`**
  In manchen Situationen ist der Gültigkeitsbereich `@RequestScoped` zu klein und der Gültigkeitsbereich `@SessionScoped` zu groß. Stellen Sie sich beispielsweise einen Einkaufswagen vor. Wenn der Kunde die Waren bezahlt, die er im Einkaufswagen gesammelt hat, ist der Vorgang beendet. Damit Sie auch für solch eine Situation eine Dauer festlegen können, wurde der Gültigkeitsbereich `@ConversationScoped` zur Verfügung gestellt.
- **`@FlowScoped`**
  `@FlowScoped` bietet die Möglichkeit einer Konversation zwischen Client und Server an. Allerdings gehen die Möglichkeiten von Faces Flows über die bisher realisierbaren Navigationsregeln mit der Annotation `@ConversationScoped` hinaus, sodass auch komplexe Konversationen modelliert und diese sogar auf unterschiedliche Webmodule übertragen werden können.
- **`@Dependent`**
  Beim Modus `bean-discovery-mode="annotated"`, der ja per Default gesetzt ist, wenn die *beans.xml* fehlt, muss der Gültigkeitsbereich auch bei den injizierten Beans gesetzt werden. Allerdings müssen Sie hierbei vorsichtig sein, denn wenn Sie eine der oben gezeigten Annotationen einsetzen, darf ihr Gültigkeitsbereich nicht über dem Gültigkeitsbereich der sie beherbergenden Komponente herausragen. Beispielsweise darf eine Backing Bean des Gültigkeitsbereichs `@SessionScoped` nicht in eine Bean injiziert werden, deren Gültigkeitsbereich lediglich `@RequestScoped` ist. Um sicherzugehen, dass dies immer gewährleistet ist, können Sie die CDI-Annotation `@Dependent` einsetzen. `@Dependent` stellt also einen Pseudo-Gültigkeitsbereich dar. Er wird gesetzt, wenn der Gültigkeitsbereich mit dem Gültigkeitsbereich der aufrufenden Komponente gleichgesetzt werden soll. Mit anderen Worten: Wurde eine Komponente in einer verwalteten Komponente injiziert, endet ihre Gültigkeit mit dem Lebensende der verwalteten Komponente.
- **`@Default`**
  Nur für den Fall, dass Sie diese Annotation mal bei einer Fehlermeldung sehen: In der *beans.xml* des Onlineshops hatten wir über `bean-discovery-mode="all"` dafür gesorgt, dass injizierte Komponenten automatisch mitverwaltet werden. Um auch für die injizierten Komponenten einen Gültigkeitsbereich anzubieten, setzt Ihnen das CDI-Framework

automatisch die Annotation `@Default` vor. `@Default` unterscheidet sich kaum von der Annotation `@Dependent`, denn auch hierbei ist die Gültigkeit von der injizierenden Komponente abhängig.

Kommen wir zurück zu unserer Backing Bean `RegisterController`. Für sie werden wir die Annotation `@RequestScoped` einsetzen, denn im Normalfall sollte der Gültigkeitsbereich einer Backing Bean so straff wie nur möglich gehalten werden.

```
package de.java2enterprise.onlineshop;

import java.io.Serializable;
import javax.enterprise.context.RequestScoped;

@RequestScoped
public class RegisterController implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

Listing 9.9 RegisterController.java

Zusätzlich werden wir der Backing Bean die Annotation `javax.inject.Named` beifügen. Hierdurch weisen wir an, dass die Backing Bean mit einem EL-Ausdruck angesprochen werden kann. Der Bezeichner der Backing Bean gleicht dem Namen der Backing-Bean-Klasse, außer dass der erste Buchstabe kleingeschrieben wird. Die Backing Bean wird in einem Facelet also über `#{registerController}` referenziert werden können:

```
package de.java2enterprise.onlineshop;

import java.io.Serializable;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import de.java2enterprise.onlineshop.RegisterController;

@Named
@RequestScoped
public class RegisterController implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

Listing 9.10 RegisterController.java

Als Nächstes werden wir eine Property des Typs `Customer` in die Backing Bean `RegisterController` setzen. Der Webcontainer muss zusätzlich über eine spezielle CDI-Annotation darüber in Kenntnis gesetzt werden, dass er die `Customer`-Bean als Property vom

`RegisterController` mitverwalten soll. Diese CDI-Annotation nennt sich `@Inject` (`javax.inject.Inject`). Hierdurch kann beispielsweise in der View-Komponente von JSF auch auf die `Customer`-Bean zugegriffen werden:

```
package de.java2enterprise.onlineshop;

import java.io.Serializable;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
import de.java2enterprise.onlineshop.model.Customer;

@Named
@RequestScoped
public class RegisterController implements Serializable {
    private static final long serialVersionUID = 1L;

    @Inject
    private Customer customer;

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
}
```

Listing 9.11 RegisterController.java

Ich habe bereits erwähnt, dass wir den Gültigkeitsbereich der zu injizierenden Klasse definieren könnten. Beispielsweise wäre die Annotation `@Dependent` ganz passend. Dadurch wäre sie von der Lebensdauer der aufrufenden `RegisterController`-Bean abhängig. Weil wir bei der *beans.xml* aber das Attribut `bean-discovery-mode="all"` gesetzt haben, kann diese Annotation auch entfallen.

Die Speicherung in der Datenbank programmieren

Im letzten Listing haben wir die Klasse `RegisterController` fast komplett fertig programmiert. Was aber noch fehlt, ist eine Anweisung, die die Daten der Klasse `Customer` in der Datenbank speichert. Diese Aufgabe werden wir mit den JPA-Kenntnissen erledigen, die wir in Kapitel 8, »Die Java Persistence API«, gewonnen haben. Dort habe ich gezeigt, wie Sie eine

JPA-Entity persistieren. Ohne hier in die Details von Kapitel 8 zu gehen, fassen wir die Anforderung für die JPA in zwei Schritten zusammen. Das Eclipse-Projekt sollte über eine *persistence.xml* im */META-INF*-Verzeichnis des Klassenpfades verfügen. Der Quelltext der *persistence.xml* muss eine `persistence-unit` definieren, damit der EntityManager auf die Datenquelle zugreifen kann. In vorangegangenen Kapiteln habe ich gezeigt, wie Sie hierdurch den Zugriff über den JNDI-Namen `"jdbc/__default"` erzielen. Die Erweiterung der Klasse `RegisterController` ist in Listing 9.12 durch fette Schrift hervorgehoben. Die Klasse `RegisterController` sieht nach der JPA-Erweiterung nun wie folgt aus:

```
package de.java2enterprise.onlineshop;

import java.io.Serializable;

import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.inject.Inject;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.UserTransaction;

import de.java2enterprise.onlineshop.model.Customer;

@Named
@RequestScoped
public class RegisterController implements Serializable {
    private static final long serialVersionUID = 1L;

    @PersistenceContext
    private EntityManager em;

    @Resource
    private UserTransaction ut;
package de.java2enterprise.onlineshop;

import java.io.Serializable;

import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
```



```
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.UserTransaction;

import de.java2enterprise.onlineshop.model.Customer;

@Named
@RequestScoped
public class RegisterController implements Serializable {
    private static final long serialVersionUID = 1L;

    @PersistenceContext
    private EntityManager em;

    @Resource
    private UserTransaction ut;

    @Inject
    private Customer customer;

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public String persist() {
        try {
            ut.begin();
            em.persist(customer);
            ut.commit();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "/register.jsf";
    }
}
```

Listing 9.12 RegisterController.java

9.2.4 Die Webkomponente »register.xhtml« erstellen

In diesem Abschnitt erstellen Sie die Webkomponente *register.xhtml* für die Registrierung des Benutzers. Die Datei *register.xhtml* legen Sie analog zur Datei *index.xhtml* im vorangegangenen Programmierbeispiel an, denn lediglich der Inhalt des `<body>`-Elements wird sich von dem Inhalt in der *index.xhtml* unterscheiden.

```
<?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:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title><h:outputText value="Onlineshop" /></title>
    </h:head>
    <h:body>

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

Listing 9.13 register.xhtml

Das HTML-Formular erzeugen

Fügen Sie nun die UI-Komponente Form in das Body-Rechteck ein. Die JSF-UI-Komponente Form benötigt kein `action`-Attribut, da die angepeilte URL erst beim Submit-Button festgelegt wird. Auch die Angabe des Request-Typs entfällt bei der Form-UI-Komponente. Im Standardfall wird das HTML-Formular mithilfe eines JSF-Command-Buttons über einen POST-Request abgeschickt.

Die Tabelle anlegen

Innerhalb der JSF-UI-Komponente Form werden wir gleich noch die speziellen Form-UI-Komponenten des JSF-Frameworks hinzufügen. Aber ehe wir damit loslegen, benötigen wir noch eine UI-Komponente, die sich um eine ordentliche Gliederung per HTML-Tabelle kümmert. Deshalb fügen Sie innerhalb der Form-UI-Komponente zunächst eine Panel Grid-UI-Komponente hinzu. Die Panel Grid-UI-Komponente wird das JSF-Framework später als HTML-Tabelle rendern.

In der Preview aus Abbildung 9.10 sehen Sie anschließend, dass der Wizard von sich aus das Panel Grid gefüllt hat, denn es wurden automatisch vier Output Text-UI-Komponenten mit den Texten *item1* bis *item4* in das Panel Grid eingefügt. Der Wizard hat außerdem dafür gesorgt, dass die Tabelle über zwei Spalten verfügt. Dies erkennen Sie an dem Attribut `columns="2"`. Das JSF-Framework setzt deshalb automatisch jeweils zwei UI-Komponenten in

eine Reihe. Die darauffolgenden UI-Komponenten werden in der nächsten Zeile hinzugefügt. Hieraus resultiert eine Tabelle, die über zwei Spalten und zwei Zeilen verfügt. Dies wird so auch im oberen Bereich der Design-Ansicht angezeigt.

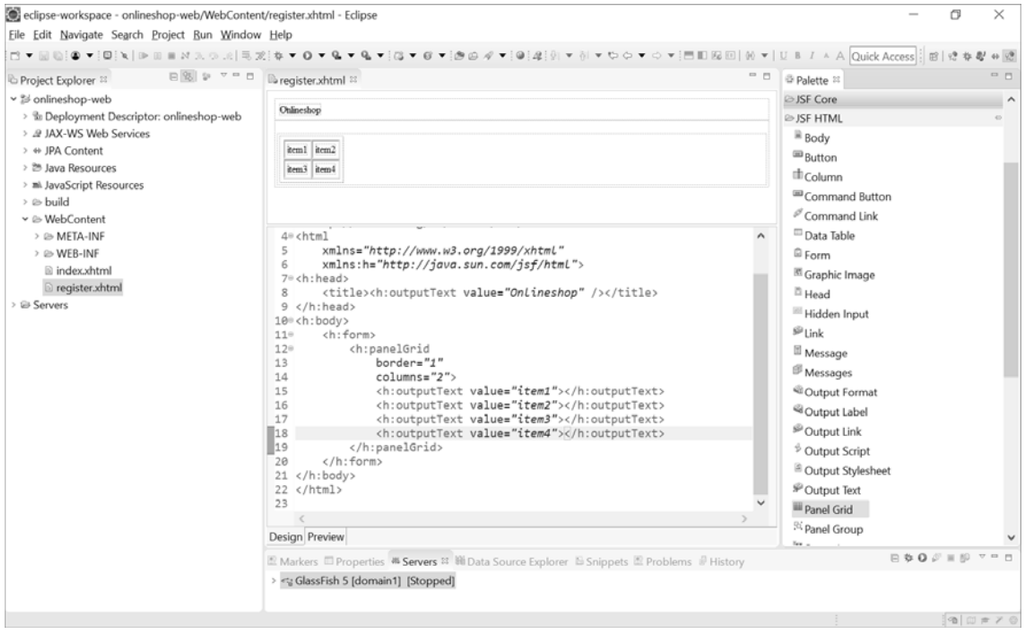

Abbildung 9.10 Die Preview zeigt eine zweispaltige Tabelle an.

Die Bemühungen des Wizards geschehen in bester Absicht, sind jedoch für unsere Zwecke unbrauchbar, denn schließlich möchten wir keine Texte, sondern Formularelemente erzeugen. Darum entfernen Sie die Output Text-UI-Komponente nun aus der Ansicht.

Öffnen Sie danach in der View Palette den Reiter JSF Core. In diesem Reiter befindet sich die JSF-UI-Komponente Facet, mit der wir dem Panel Grid zunächst eine Überschrift geben werden. Mit *Facet* (zu Deutsch *Facette*) ist gemeint, dass es sich um eine Child-Komponente handelt, die einen bestimmten Aspekt einer Parent-Komponente abhandelt. Der Aspekt wird über das Attribut `name` gekennzeichnet. Beispielsweise können wir der UI-Komponente Panel Grid die Facets mit den Namen `"header"` und `"footer"` beifügen. Aus dem Facet mit dem Namen `"header"` wird das JSF-Framework ein HTML-`<th>`-Element erzeugen.

Ziehen Sie nun die Facet-UI-Komponente in das Panel Grid. Beim öffnenden Tag setzen Sie das Attribut `name="header"`. Zwischen das öffnende und das schließende Facet-Tag ziehen Sie eine Output-Text-Komponente aus dem Reiter JSF HTML. Beim `value`-Attribut tragen Sie `"Registrieren"` ein.

Eingabefelder hinzufügen

Für die Registrierung des Benutzers benötigen wir zwei Eingabefelder, nämlich für seine E-Mail-Adresse und für sein Kennwort. Das zweite Eingabefeld für das Kennwort sollte die eingegebenen Zeichen als Wildcards darstellen, deshalb werden wir hierfür ein spezielles Security-Eingabefeld einsetzen. Außerdem sollen die Eingabefelder über Label-UI-Komponenten betitelt sein. Deshalb ziehen Sie nun ein OUTPUT LABEL und ein TEXT INPUT für die E-Mail-Adresse sowie ein OUTPUT LABEL und ein SECRET INPUT für das Kennwort in das PANEL GRID hinein.

Beachten Sie, dass der Wizard stets ein öffnendes und ein schließendes Tag erstellt. Weil der innere Bereich der Elemente leer ist, können wir die öffnenden und schließenden Tags auch zu einem einzigen Tag zusammenfassen. Dies erledigt der Wizard automatisch, wenn Sie im öffnenden Tag vor die schließende Eckklammer einen Querstrich setzen. Der Wizard erkennt anschließend, dass wir kein schließendes Tag mehr brauchen, und entfernt es automatisch.

In den `value`-Attributen der Output Label ändern wir nun den auszugebenden Text. Klicken Sie hierfür in der Vorschau doppelt auf `outputLabel`. Hierdurch sollte sich im unteren Bereich die View PROPERTIES öffnen. Diese View bietet für die Bearbeitung der Attribute zwei verschiedene Editiermodi an, und zwar QUICK EDIT und ATTRIBUTES. Klicken Sie auf QUICK EDIT. Setzen Sie hinter `value:` den Wert `"E-Mail:"`. Genauso gehen Sie beim zweiten Output Label vor, nur dass Sie dort den Wert `"Kennwort:"` eintragen.

Klicken Sie danach doppelt auf das Innere des TEXT INPUT-Elements, und öffnen Sie im unteren Bereich die View PROPERTIES. Dort tragen Sie im Feld `value` den Wert `#{registerController.customer.email}` ein.

Durch den Ausdruck `#{registerController.customer.email}` verwenden wir die JSF-EL. In der Syntax und in den grundlegenden Bestandteilen sind JSF-EL und JSP-EL (aus Kapitel 5, »Java Server Pages«) gleich. Zum Beispiel greift der Ausdruck `#{registerController.customer.email}` auf das Objekt `registerController` der Java-Klasse `RegisterController` zu. In der Klasse `RegisterController` ist eine Property namens `customer` enthalten, deren Wert über eine öffentliche Getter-Methode `getCustomer()` beschafft werden kann. Genauso wie mit JSP-EL lässt sich die Methode `getCustomer()` auch bei JSF-EL aufrufen, indem Sie hinter den Punktoperator den Bezeichner `customer` setzen. Darüber hinaus können Sie wie im Beispiel über weitere Punktoperatoren von Property zu Property wandern. Diese Syntax habe ich bereits in Abschnitt 5.6, »JSP-EL«, beschrieben.

Aber kehren wir nun zu unserem Beispiel zurück. Der Ausdruck `#{registerController.customer.email}` veranlasst, dass eine `RegisterController`-Bean und eine `Customer`-Bean erzeugt werden. Wenn der Benutzer eine Zeichenkette in das E-Mail-Textfeld schreibt und das Formular absendet, wird die Zeichenkette als Parameter über einen POST-Request an den Server versandt. Den Wert, den der Benutzer in das Textfeld einträgt, wird das Faces-

Servlet intern automatisch als Attributwert für die neue `Customer`-Bean verwenden. Weil wir den Gültigkeitsbereich von `RegisterController` und damit auch die `Customer`-Bean auf `RequestScoped` gesetzt haben, sind beide Beans während des HTTP-Requests gültig.

Als Nächstes gehen Sie gleichermaßen für das Passwort-Eingabefeld vor. Ersetzen Sie den vorhandenen Text durch den Text `"Kennwort:"`. Klicken Sie anschließend auch in SECRET INPUT. Danach können Sie in der PROPERTIES-View auch hierfür einen `value`-Wert setzen. Tragen Sie dort `#{registerController.customer.password}` ein.

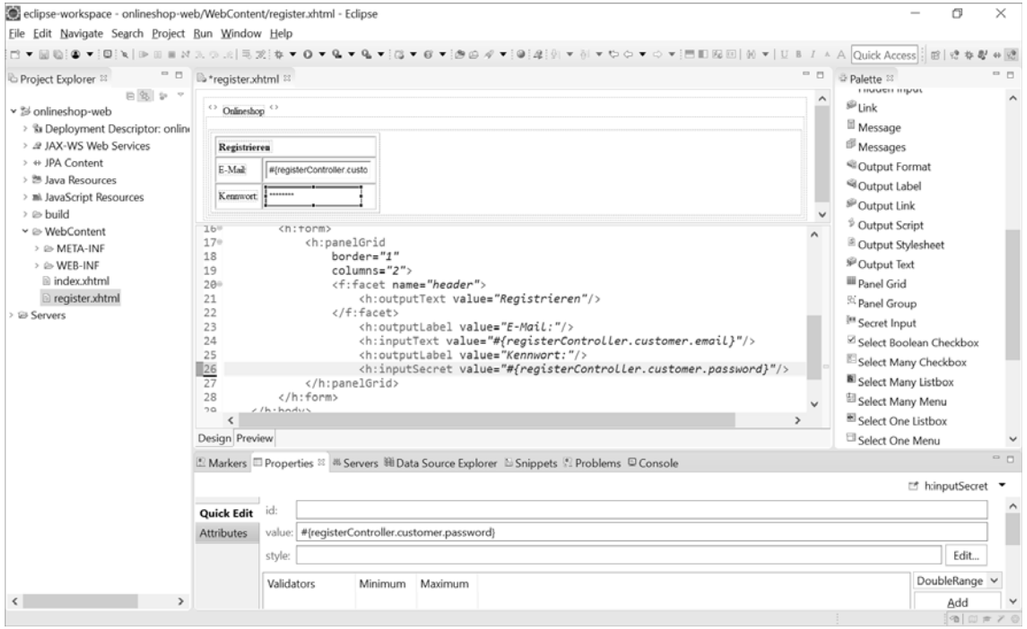

Abbildung 9.11 Die Properties der UI-Komponente »Secret Input«

Die Geschäftsdaten an den Server senden

Damit die Geschäftsdaten des HTML-Formulars an den Server versendet werden, setzen wir im Beispiel einen JSF-Command-Button ein.

```
<h:commandButton
    value="Registrieren"/>
```

Die Aktion, die durch den Mausklick auf die Schaltfläche ausgeführt werden soll, definieren wir mithilfe des Attributs `action`. Der zugewiesene Wert ist der Bezeichner der Aktionsmethode in der Backing Bean ohne die runden Klammern.

Um der Schaltfläche auch noch einen Anzeigetext mitzugeben, fügen Sie das Attribut `value` hinzu.

Hinweis

Weil es schöner aussieht, habe ich im Quelltext das Attribut des Panel Grids `border="1"` entfernt. Beachten Sie im Quelltext auch noch, dass der Wizard den Namespace von sich aus automatisch eingefügt hat.

Der komplette Quelltext der View-Komponente *register.xhtml* sollte nun wie folgt aussehen:

```
<?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:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
    <title>
        <h:outputText value="Onlineshop"/>
    </title>
    <meta
        http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <f:facet name="header">
            <h:outputText value="Registrieren"/>
            </f:facet>
            <h:outputLabel value="E-Mail:" />
            <h:inputText
                value=
                "#{registerController.customer.email}"
            >
            </h:inputText>
            <h:outputLabel value="Kennwort:" />
            <h:inputSecret
                 value=
                "#{registerController.customer.password}"
            >
            </h:inputSecret>
            <h:commandButton
            action="#{registerController.persist}"
            value="Registrieren"/>
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Listing 9.14 register.xhtml

Im Programmierbeispiel haben wir den Command-Button eingesetzt, um die Aktionsmethode in der Backing Bean zu aktivieren. Hierbei war es wichtig, dass der Command-Button von einer `form`-Komponente umgeben ist.

Bei der Ausführung des Programms wird der Command-Button als Schaltfläche `<input type="submit>` gerendert.

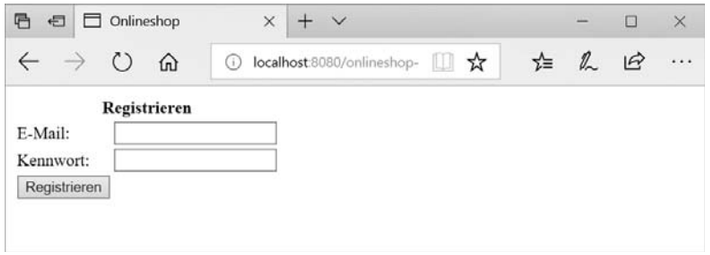

Abbildung 9.12 Der Aufruf der URL »http://localhost:8080/onlineshop-web/register.jsf«

Klickt der Benutzer auf die Schaltfläche, so wird die Anfrage des HTML-Formulars als HTTP-POST-Request an den Server gesendet, was schließlich dazu führt, dass die Methode `persist()` der Klasse `RegisterController` aufgerufen wird.

9.3 Die implizite Navigation

Im Web löst der Benutzer üblicherweise eine Navigation aus, indem er mit der Maus entweder auf einen HTML-Link oder auf einen SUBMIT-Button eines HTML-Formulars klickt. JSF bietet unterschiedliche UI-Komponenten an, über die die Navigation des Webs genutzt wird, um die Abfolge der aufgerufenen Komponenten festzulegen. Hierbei wird der Navigationsmechanismus des JSF-Frameworks verwendet.

9.3.1 Der Command-Button

Im vorangegangenen Programmierbeispiel haben wir bereits den Command-Button eingesetzt. Über den Command-Button hatten wir dafür gesorgt, dass eine Aktionsmethode aufgerufen wird und dadurch den Kursverlauf bestimmt. Somit haben wir auch mit der bereits implementierten Funktionalität eine Navigation eingebaut. Die UI-Komponente Command-

Button lässt sich aber nicht nur für das Aktivieren einer Aktionsmethode verwenden, sondern auch für die Navigation zu einer anderen View-Komponente, denn bei den JSF-Schaltflächen kann ebenfalls der Pfad der anvisierten View-Komponente als Ziel der Aktion gesetzt werden. Der folgende Command-Button steuert beispielsweise die View */index.xhtml* an, wobei auch hier der Navigationsmechanismus von JSF genutzt wird:

```
<?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:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
    <title>
        <h:outputText value="Onlineshop"/>
    </title>
    <meta
        http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <f:facet name="header">
            <h:outputText value="Registrieren"/>
            </f:facet>            <h:outputLabel value="E-Mail:" />
            <h:inputText
                value=
                "#{registerController.customer.email}"
            >
            </h:inputText>
            <h:outputLabel value="Kennwort:" />
            <h:inputSecret
                 value=
                "#{registerController.customer.password}"
            >
            </h:inputSecret>
            <h:commandButton
                action="#{registerController.persist}"
                value="Registrieren"/>
            <h:commandButton
                action="/index.xhtml"
                value="Abbrechen"/>
```



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

Listing 9.15 register.xhtml

Eine festgelegte Logik im JSF-Navigationsmechanismus ermöglicht, dass das Framework die anvisierte View-Komponente von selbst ermittelt. Beispielsweise würde der Command-Button mit **`action="index"`** ebenso die View */index.xhtml* ansteuern. Dies gelingt, weil JSF den eingetragenen Bezeichner `"index"` automatisch als View-ID des VDL-Dokuments (*View Declaration Language*) betrachtet. Der Fachbegriff hierfür lautet *implizite Navigation*.


```
    <h:commandButton
        action="index"
        value="Abbrechen"/>
```

Listing 9.16 register.xhtml

9.3.2 Der Command-Link

Command-Links ähneln den Command-Buttons. Sie werden beispielsweise genauso wie Command-Buttons innerhalb einer Form-UI-Komponente platziert und erhalten zur Navigation auch die gleichen Attribute. Beim Command-Button der Programmierübung habe ich bereits gezeigt, wie Sie das Attribut `action` einsetzen. Auch beim Command-Link tragen Sie entweder eine Aktionsmethode oder auch direkt die anzusteuernde View ein:

```
<?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:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
    <title>
        <h:outputText value="Onlineshop"/>
    </title>
    <meta
        http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
```

```
            <f:facet name="header">
            <h:outputText value="Registrieren"/>
            </f:facet>           <h:outputLabel value="E-Mail:" />
            <h:inputText
                value=
                "#{registerController.customer.email}"
            >
            </h:inputText>
            <h:outputLabel value="Kennwort:" />
            <h:inputSecret
                 value=
                "#{registerController.customer.password}"
            >
            </h:inputSecret>
            <h:commandButton
                action="#{registerController.persist}"
                value="Registrieren"/>
            <h:commandLink
                action="index"
                value="Abbrechen"/>
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Listing 9.17 register.xhtml

Wenn Sie in Ihrem Programm den Command-Button durch einen Command-Link ersetzen, werden statt der Schaltflächen nun Hyperlinks gerendert, sodass dem Anschein nach kein Unterschied zu einem normalen Link besteht. Bei dem gerenderten Hyperlink zeigt sich aber, dass es sich keinesfalls um ein gewöhnliches HTML-Anker-Element handelt, denn der Link verweist auf sich selbst:

```
<a href="#">
```

Außerdem wird die Anfrage genauso wie bei einem Command-Button über einen HTTP-POST-Request vom Client zum Server versendet. Dies ermöglicht ein `onclick`-Attribut, das ein JavaScript-Programm aufruft. Dem Benutzer des Webbrowsers bleiben diese internen Vorgänge weitestgehend verborgen. Dies kann in gewissen Fällen auch nützlich sein. Der Vorteil des Command-Links ist aber gleichzeitig sein Nachteil: Wenn der Benutzer beispielsweise ein Lesezeichen setzen möchte, würde ihm das nicht gelingen, da die URL ja lediglich einen Verweis auf die eigene URL enthält. Auf diesen Makel werden wir später zurückkommen, denn das JSF-Framework bietet aus diesem Grunde weitere UI-Komponenten an, die dieses Problem lösen.

9.3.3 Die implizite Navigation von Facelet zu Facelet

Bei der Navigation über JSF spielt der Rückgabewert der Aktionsmethode eine zentrale Rolle, denn er kann entweder über die View-ID oder aber auch über eine zu konfigurierende Zeichenkette gesteuert werden. Im Fachjargon spricht man vom *Outcome*.

Die Willkommensseite werden wir abändern, sodass sie einen Link zur Registrierung anbietet.

```
<?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:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title><h:outputText value="Onlineshop" /></title>
        <meta
            http-equiv="Content-Type"
            content="text/html; charset=UTF-8" />
    </h:head>
    <h:body>
        <h1 class="title">
            <h:outputText value="Onlineshop" />
        </h1>
        <h:link
            outcome="register"
            value="Registrieren" />
    </h:body>
</html>
```

Listing 9.18 index.xhtml

In der Willkommensseite wird nun ein Link zur Registrierung angeboten (Abbildung 9.13).

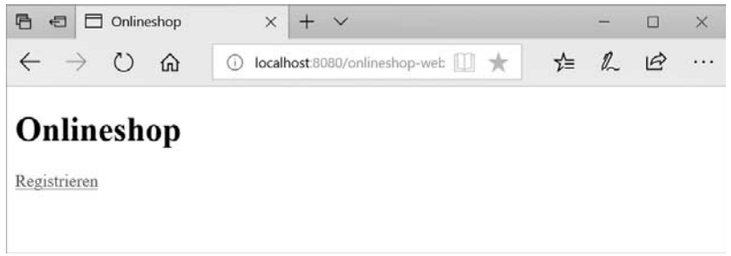

Abbildung 9.13 Die Willkommensseite mit dem Link zur Registrierung

Über den Link gelangen Sie zur Registrierung, wo wir ebenso einen Link einbauen könnten, um die Navigation zwischen Facelet zu Facelet zu ermöglichen.

```
        <h:link
            outcome="index"
            value="Zur Willkommensseite" />
    </h:body>
</html>
```

Listing 9.19 Der Link zur Willkommensseite

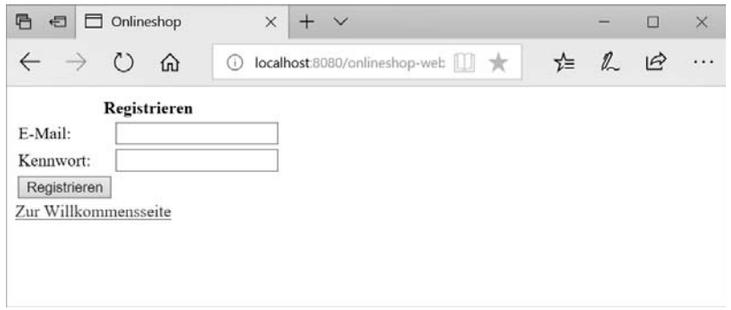

Abbildung 9.14 Die Registrierung mit JSF

9.3.4 In der Aktionsmethode eine View-ID als Ziel ansteuern

In unserem Programmierbeispiel navigieren wir über die Aktionsmethode `persist()` in der Backing Bean zurück in die View-Komponente */register.xhtml* (siehe Abbildung 9.15). Aber bestimmt haben Sie es bereits erahnt: Genau wie bei einem Facelet ist es auch in der Aktionsmethode möglich, ein Facelet über seine View-ID anzusteuern.

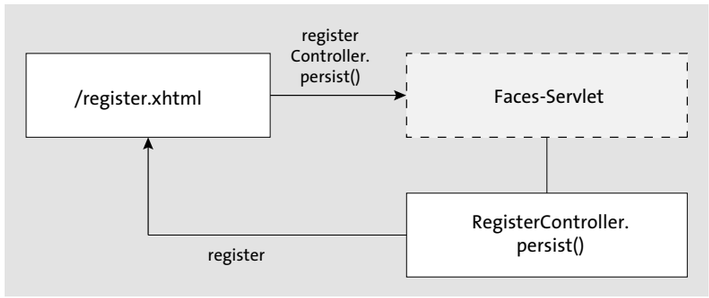

Abbildung 9.15 Die Navigation über die View-ID

Im nächsten Programmierbeispiel werden wir einen Schritt weitergehen und eine `if-else`-Anweisung einbauen, die die Navigation über eine Logik steuert. Als Beispiel hierzu werden wir den folgenden Programmablauf implementieren.

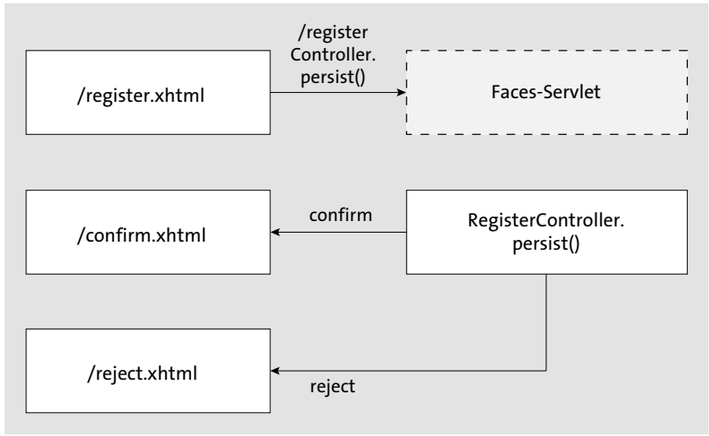

Abbildung 9.16 Die implizite Navigation zu den Views »confirm.xhtm« und »reject.xhtml«

In Abbildung 9.16 navigieren wir nicht mehr zurück zum Facelet */register.xhtml*. Stattdessen steuern wir über eine interne Logik entweder das Facelet */confirm.xhtml* oder das Facelet */reject.xhtml* an.

Legen Sie für das Beispiel zwei weitere XHTML-Dateien an, die Sie *confirm.xhtml* und *reject.xhtml* nennen.

In der Datei *confirm.xhtml* wird die erfolgreiche Registrierung bestätigt. Darunter setzen wir eine Schaltfläche, über die der Benutzer auf die Homepage wechseln kann.

```
<?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:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:body>
    Sie wurden erfolgreich registriert!
    <h:form>
        <h:commandButton
            action="index"
            value="Zur Willkommensseite"/>
    </h:form>
</h:body>
</html>
```

Listing 9.20 confirm.xhtml

Der Inhalt der Datei *reject.xhtml* sieht ähnlich aus, nur dass dort das Scheitern der Registrierung gemeldet wird:

```
<?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:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:body>
    Die Registrierung ist misslungen!
    <h:form>
        <h:commandButton
            action="index"
            value="Zur Willkommensseite"/>
    </h:form></h:body>
</html>
```

Listing 9.21 reject.xhtml

In der Backing Bean `RegisterController.java` werden wir dafür sorgen, dass die folgende View je nach Situation zunächst dynamisch in der Backing Bean ermittelt wird. Nur wenn die Speicherung gelingt, soll die View */confirm.xhtml* aufgerufen werden. Ansonsten soll die View */reject.xhtml* erscheinen:


```
package de.java2enterprise.onlineshop;

import java.io.Serializable;

import javax.annotation.Resource;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.transaction.UserTransaction;

import de.java2enterprise.onlineshop.model.Customer;

@Named
@RequestScoped
public class RegisterController implements Serializable {
    private static final long serialVersionUID = 1L;

    @PersistenceContext
    private EntityManager em;

    @Resource
    private UserTransaction ut;

    @Inject
    private Customer customer;

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public String persist() {
        try {
            ut.begin();
            em.persist(customer);
            ut.commit();
        } catch (Exception e) {
            e.printStackTrace();
```

```
            return "reject";
        }
        return "confirm";
    }
}
```

Listing 9.22 RegisterController.java

9.4 Die explizite Navigation

Ein wesentlicher Bestandteil der JSF-Navigation ist, dass sie sich nicht nur implizit, sondern auch explizit per Konfiguration einstellen lässt. Hierfür wird die JSF-Konfigurationsdatei *faces-config.xml* verwendet.

> **Hinweis**
>
> Ursprünglich war es die wichtigste Aufgabe der *faces-config.xml*, die sogenannten Managed Beans zu konfigurieren. Da die Verwendung von Managed Beans ab der Java EE-Version 8 *deprecated* ist, definieren wir in der *faces-config.xml* vorwiegend Navigationsregeln.

Die *faces-config.xml* wurde von Eclipse bereits beim ersten Programmierbeispiel automatisch im Verzeichnis */WEB-INF* angelegt. Der Wizard von Eclipse sollte sie bei Ihren Programmierbeispielen (in Ihrer aktuellen Eclipse IDE) wie folgt angelegt haben.

```
<?xml version="1.0" encoding="UTF-8"?>
<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_3.xsd"
    version="2.3">
</faces-config>
```

Listing 9.23 faces-config.xml

9.4.1 Navigation-Rules

Bei der expliziten Navigation wird in der JSF-Konfigurationsdatei für jede Navigationsregel ein `navigation-rule`-Element gesetzt. Innerhalb der `navigation-rule`-Elemente werden beliebig viele Navigationsfälle mit dem XML-Element `navigation-case` festgelegt.

Für jeden Navigationsfall wird ein Outcome einer View-ID zugeordnet. Der Outcome wird über das XML-Element `from-outcome` und die View-ID über das XML-Element `to-view-id` spezifiziert.

Mit der folgenden Navigationsregel legen Sie fest, dass bei dem Rückgabewert `success` zur View */confirm.xhtml* navigiert wird:

```
<?xml version="1.0" encoding="UTF-8"?>
<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_3.xsd"
 version="2.3">
    <navigation-rule>
        <from-view-id>*</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/confirm.xhtml</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/reject.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Listing 9.24 faces-config.xml

Die obige Navigationsregel gilt für alle Ausgangs-Views. Innerhalb des XML-Elements `navigation-rule` kann auch das XML-Element `from-view-id` genutzt werden, um die Navigationsregel auf eine Ausgangs-View einzuschränken:

```
...
<navigation-rule>
    <from-view-id>/register.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/confirm.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
...
```

Listing 9.25 faces-config.xml

Beim XML-Element `from-view-id` können Sie auch Wildcards nutzen. Beispielsweise gilt die folgende Regel für alle Ausgangs-View-Komponenten, die sich im Pfad */register* befinden:

```
...
<navigation-rule>
    <from-view-id>/register/*</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/confirm.xhtml</to-view-id>
    </navigation-case>
...
```

Listing 9.26 faces-config.xml

9.4.2 Forward und Redirect

Normalerweise wird die Navigation serverseitig umgeleitet. Die Servlet-Technologie gebraucht hierfür den Fachbegriff *Forward*. Dass es sich um eine serverseitige Weiterleitung handelt, erkennen Sie im Programmierbeispiel an der URL in der Adressleiste Ihres Webbrowsers, denn nach dem Mausklick auf der Schaltfläche bleibt diese unverändert.

Statt der serverseitigen Umleitung mit einem Forward können Sie aber auch eine clientseitige Umleitung erwirken. Der Fachbegriff für eine clientseitige Umleitung ist *Redirect*.

In der `faces-config` steht hierfür das Element `<redirect/>` zur Verfügung:

```
...
<navigation-rule>
    <from-view-id>/register.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/confirm.xhtml</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>
...
```

Listing 9.27 faces-config.xml

Hinweis

Weitere Informationen über die Umleitung mit einem Forward oder einem Redirect finden Sie in Kapitel 4, »Servlet 4.0«.

9.4.3 Das Programmierbeispiel

Wir werden das komplette Beispiel der expliziten Navigation für die Registrierung nun programmieren. Statt der Rückgabewerte `confirm` und `reject`, die ja als View-IDs eine implizite Navigation ermöglichen, soll die Methode `persist()` nun die Zeichenketten `success` und `failure` liefern. Weil die Rückgabewerte jetzt anders lauten als die Facelet-Dateien, ist eine explizite Navigationsregel über die JSF-Konfigurationsdatei erforderlich (siehe Abbildung 9.17).

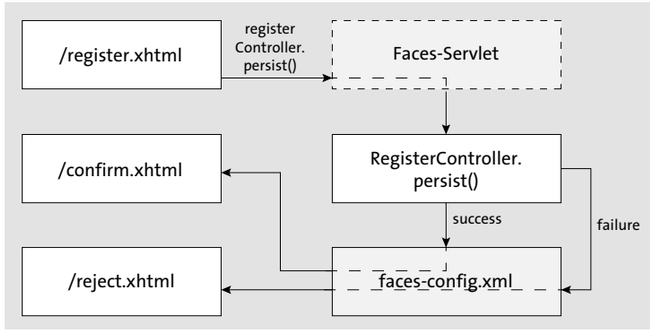

Abbildung 9.17 Die explizite Navigation

Um die explizite Navigation in der JSF-Konfigurationsdatei fertigzustellen, lassen Sie sich von Eclipse helfen. Öffnen Sie hierfür im Webprojekt den Ordner */WebContent/WEB-INF*, und klicken Sie dort doppelt auf die Datei *faces-config.xml*. Selektieren Sie dann den Reiter NAVIGATION-RULE. Beachten Sie, dass Sie die Perspektive JAVA EE benötigen.

Ziehen Sie die Dateien *register.xhtml*, *confirm.xhtml* und *reject.xhtml* aus dem Projekt-Explorer in die Arbeitsfläche der NAVIGATION RULE. Selektieren Sie anschließend in der View PALETTE die Auswahl LINK, denn hiermit können Sie die Navigation Rule grafisch gestalten. Zeichnen Sie jetzt einen Pfeil, indem Sie eine Linie von der *register.xhtml* zur *confirm.xhtml* ziehen. Genauso ziehen Sie danach auch noch einen Pfeil von der *register.xhtml* zur *reject.xhtml*.

Im Anschluss markieren Sie den Pfeil, der von der *register.xhtml* zur *confirm.xhtml* zeigt, indem Sie rechts die Option SELECT auswählen und hiernach auf den oberen Pfeil klicken. Im unteren Bereich von Eclipse öffnen Sie die View PROPERTIES und geben in das Eingabefeld FROM ACTION die Zeichenkette `#{registerController.persist}` ein. Im nächsten Eingabefeld FROM OUTCOME geben Sie den Wert `success` ein. Genauso markieren Sie den zweiten Pfeil, um dort im ersten Eingabefeld ebenfalls `#{registerController.persist}` und im zweiten Eingabefeld die Zeichenkette `failure` einzutragen. Die Ansicht von Eclipse sollte nun so wie in Abbildung 9.18 aussehen.

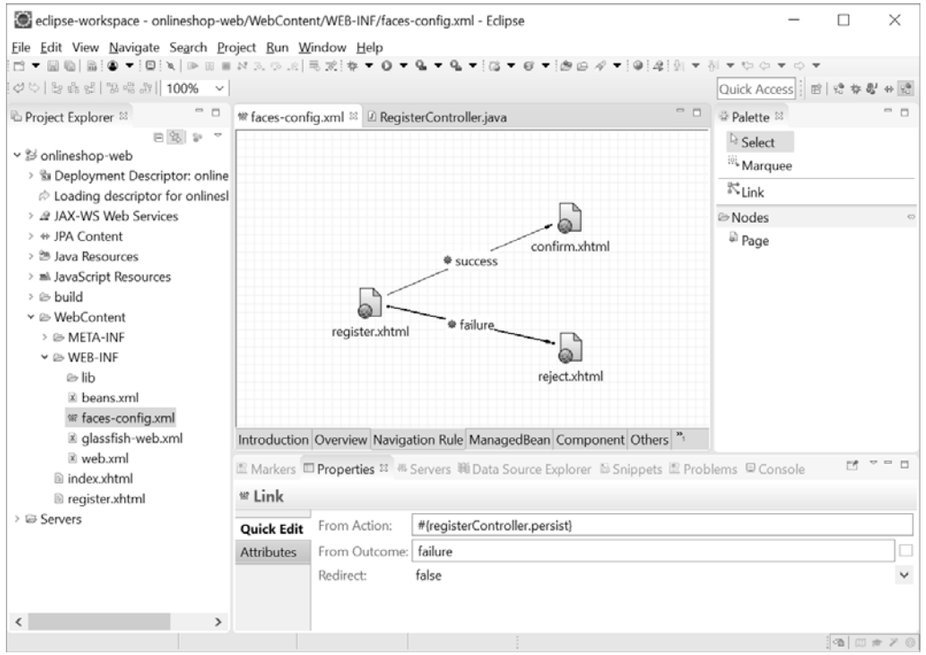

Abbildung 9.18 Der Reiter »Navigation Rule« im »Faces Configuration Editor«

Wenn Sie auf den Reiter SOURCE klicken, sollten Sie sehen, dass der Wizard Folgendes in die Datei *faces-config.xml* hineingeschrieben hat:

```
<?xml version="1.0" encoding="UTF-8"?>
<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_3.xsd"
    version="2.3">
    <navigation-rule>
        <display-name>register.xhtml</display-name>
        <from-view-id>/register.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/confirm.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>
    <navigation-rule>
        <display-name>register.xhtml</display-name>
```



```
        <from-view-id>/register.xhtml</from-view-id>
        <navigation-case>
            <from-outcome>failure</from-outcome>
            <to-view-id>/reject.xhtml</to-view-id>
        </navigation-case>
    </navigation-rule>

</faces-config>
```

Listing 9.28 faces-config.xml

In der Backing Bean müssen Sie die Rückgabewerte nun ebenfalls anpassen:

```
...
public String persist() {
    try {
        ut.begin();
        emf.createEntityManager().persist(customer);
        ut.commit();
        return "success";
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "failure";
}
...
```

Listing 9.29 RegisterController.java

9.5 JSF-Grundkenntnisse

Jetzt, nachdem Sie bereits vier eigene Beispielanwendungen programmiert haben, schauen wir uns die vorgefertigten Werkzeuge von JSF genauer an. Wir beginnen mit den Arbeitsphasen, die die HTTP-Anfrage im JSF-Framework durchläuft. Anschließend betrachten wir JSF-Klassen, die bei der Programmierung von Backing Beans nützlich sind. Danach lernen Sie weitere UI-Komponenten kennen. Abschließend werde ich Ihnen noch zeigen, wie Sie die JSF-Konfigurationsdatei verschieben und unterteilen können.

9.5.1 Die Arbeitsphasen

Zu Beginn des Kapitels habe ich das Zusammenspiel der einzelnen Komponenten im Model-View-Controller-Pattern von JSF erläutert. Wir werden uns diesen Ablauf jetzt noch etwas

detaillierter anschauen, denn wenn Sie die Arbeitsphasen von JSF bei einem HTTP-Request genauer kennen, lassen sich manche Funktionen besser zuordnen.

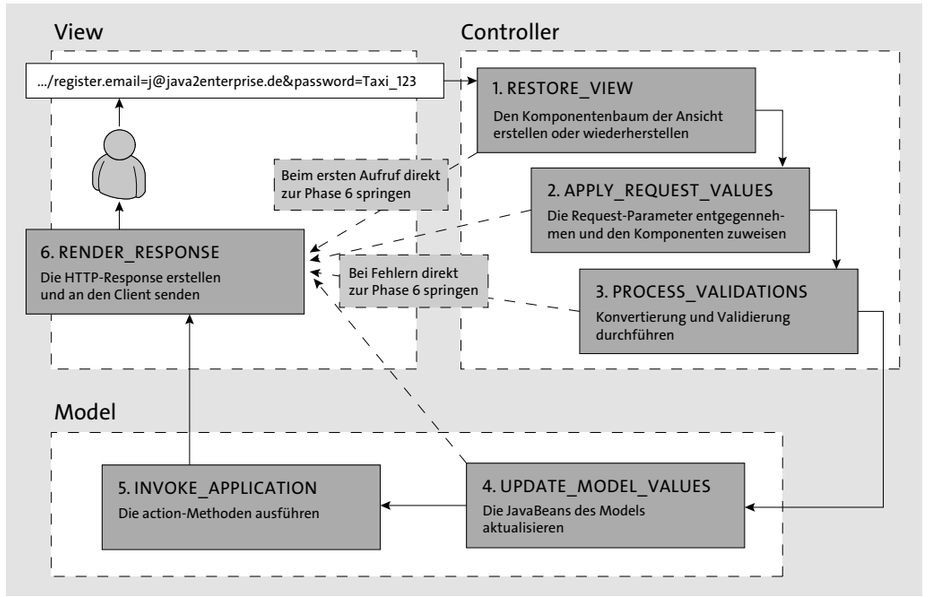

Abbildung 9.19 Die Arbeitsphasen bei einem HTTP-Request

Bei einem HTTP-Request können folgende Arbeitsphasen auftreten:

1. **RESTORE_VIEW**
   Die erste Phase beginnt, wenn ein HTTP-Request beim Server eintrifft. Sie nennt sich `RESTORE_VIEW`. Bei dieser Phase wird die Ansicht erstellt bzw. wiederhergestellt. Hierbei wird eine Instanz der Klasse `UIViewRoot` erzeugt. Sie ist der Wurzelknoten der hierarchischen Baumstruktur der UI-Komponenten. Der Wurzelknoten wird im sogenannten *FacesContext* abgespeichert, sodass von hier aus auf alle UI-Komponenten des Facelets zugegriffen werden kann. Bei den Verarbeitungsschritten einer HTTP-Anfrage spielt die Klasse `FacesContext` eine bedeutende Rolle, da sie die zentrale Stelle darstellt, von der aus Sie auch auf weitere Bestandteile und Funktionalitäten des Frameworks zugreifen können.

   In Abbildung 9.19 sehen Sie, dass von der ersten Phase aus zwei Pfeile abgehen. Der gestrichelte Pfeil, der direkt zur sechsten Phase zeigt, wird durchlaufen, wenn ein Benutzer eine JSF-Seite zum ersten Mal betritt. In diesem Fall spricht man von einem *Initial Request*. Bei einem Initial Request werden nur der erste und der letzte Arbeitsschritt durchlaufen; dies spart Zeit.

   Löst der Benutzer hingegen eine Anfrage durch einen SUBMIT-Button aus, so lautet der Fachausdruck *Postback-Request*. Bei einem Postback-Request verbleibt die Navigation im

aktuellen Lebenszyklus. Dabei wird in der ersten Phase der gespeicherte Zustand der UI-Komponenten wiederhergestellt und weiter zur zweiten Phase gegangen.

2. **APPLY_REQUEST_VALUES**
   In der zweiten Phase des Lebenszyklus werden zunächst die HTTP-Parameter entgegengenommen. Anschließend wird vom Wurzelknoten aus jeder Kindknoten rekursiv durchlaufen. Dabei sucht sich jede UI-Komponente die für sie übermittelten Werte heraus.
3. **PROCESS_VALIDATIONS**
   In der dritten Phase werden alle Werte von UI-Komponenten konvertiert und validiert. Dieser Vorgang ist von Bedeutung, da man in der clientseitigen Präsentationsschicht (also im HTML-Formular) lediglich Zeichenketten eingibt. Aus diesen Zeichenketten werden später beispielsweise Geldbeträge oder Datumswerte erstellt. Tritt hierbei ein Fehler auf, springt der Lebenszyklus von hier aus direkt in die 6. Phase.
4. **UPDATE_MODEL_VALUES**
   In Phase 4 werden die bereits konvertierten und validierten Werte in das Datenmodell übertragen. Beispielsweise werden in der Bean `Customer.java` die Properties `email` und `password` gesetzt, indem die Setter-Methoden der Bean aufgerufen werden. Auch bei dieser Phase können aufkommende Fehler dazu führen, dass der Prozess in Phase 6 springt.
5. **INVOKE_APPLICATION**
   In Phase 5 werden die ActionListener aktiviert und die Aktionsmethoden der Backing Beans aufgerufen. Beispielhaft wurde bei der Klasse `Customer.java` die Aktionsmethode `persist()` vorgesehen, die für die Speicherung der `Customer`-Objekte in die relationale Datenbank zuständig ist. Aktionsmethoden werden nach den ActionListenern aufgerufen.
6. **RENDER_RESPONSE**
   In der sechsten und letzten Phase wird die HTTP-Response erstellt und für spätere Anfragen gesichert. Zuletzt wird die HTTP-Response an den Client versendet.

Über »immediate="true"« den Lebenszyklus abändern

Enthält eine UI-Komponente das Attribut `immediate="true"`, so wird der Ablauf im Lebenszyklus der Anfrage etwas abgeändert.

Bei Eingabefeldern hat das Attribut zur Folge, dass die Konvertierung und die Validierung der Eingabefelder nicht erst in der 3. Phase, sondern bereits in der 2. Phase stattfinden. Treten hierbei Fehler auf, wird anschließend unmittelbar die 6. Phase eingeleitet.

Bei Befehlskomponenten werden ActionListener und Aktionsmethoden in der 2. Phase statt in der 5. Phase aktiv.

Das Attribut kann nützlich sein, wenn ein Teilaspekt der Webseite ganz unabhängig von anderen Komponenten aktiviert werden soll. Das folgende Beispiel zeigt einen CANCEL-(ABBRECHEN-)Command-Button in der *register.xhtml*:

```
        ...
        <h:commandButton
            immediate="true"
            action="/index.xhtml"
            value="Abbrechen"/>
        <h:commandButton
            action="#{registerController.persist}"
            value="Registrieren"/>
    </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Listing 9.30 register.xhtml

9.5.2 Bedeutende JSF-Klassen für die Backing Bean

Auf die besondere Bedeutung der Backing Beans für die Nutzung der JSF-API habe ich bereits hingewiesen. Dabei wurde deutlich, dass Backing Beans ganz einfache POJO-Klassen sind, in die die Geschäftsdaten und die Aktionsmethoden der jeweiligen User-Story eingebaut werden. Hierbei möchte man häufig auch auf weitere Komponenten oder sogar auf die Low-Level-Elemente der darunterliegenden Servlets zugreifen. Für all das hat das JSF-Framework vorgesorgt, indem es entsprechende Java-Klassen zur Verfügung stellt. Im Folgenden werde ich die wichtigsten Klassen beschreiben.

FacesContext

Eine der wichtigsten Klassen ist die Klasse `FacesContext`, denn sie ermöglicht den Zugriff auf zahlreiche interne Informationen des JSF-Frameworks. Für die Arbeit mit der zentralen `FacesContext`-Instanz bietet die abstrakte Klasse `javax.faces.context.FacesContext` die Methode `getCurrentInstance()` an:

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

Beispielsweise können Sie hierüber nachfragen, in welcher Phase sich der Anfrageprozess momentan befindet. Zu diesem Zweck wird die Methode `FacesContext.getCurrentPhaseId()` ausgeführt. Der Rückgabewert ist vom Typ `PhaseId`, der mit einer dieser sechs Konstanten verglichen wird:

- `PhaseId.RESTORE_VIEW`
- `PhaseId.APPLY_REQUEST_VALUES`
- `PhaseId.PROCESS_VALIDATIONS`

- `PhaseId.UPDATE_MODEL_VALUES`
- `PhaseId.INVOKE_APPLICATION`
- `PhaseId.RENDER_RESPONSE`

Im folgenden Listing 9.31 sehen Sie, wie in einer Backing Bean geprüft wird, ob sich der aktuelle Prozess in der Phase `PROCESS_VALIDATION` befindet:

```
PhaseId phaseId = facesContext.getCurrentPhaseId();
if(PhaseId.PROCESS_VALIDATIONS.equals(phaseId)) {
    // do this and that
}
```

Listing 9.31 PhaseId-Überprüfung

Eine weitere wichtige Aufgabe der Klasse `FacesContext` ist die Speicherung des UI-Komponenten-Baums als Instanz des Typs `javax.faces.component.UIViewRoot`. Über die Methode `getViewRoot()` lässt sich diese Instanz als Wurzel des gesamten UI-Komponenten-Baums beschaffen:

```
UIViewRoot viewRoot = facesContext.getViewRoot();
```

Von hier aus lassen sich also ganz komfortabel einzelne UI-Komponenten des UI-Komponenten-Baums bearbeiten.

Mit der Methode `findComponent()` holen Sie eine Referenz auf eine UI-Komponente:

```
UIComponent component = viewRoot.findComponent("registerForm");
```

Eine weitere Aufgabe der Klasse `FacesContext` ist die Nachrichtenübermittlung an die View-Komponente.

ExternalContext

Die zentrale Instanz der Klasse `javax.faces.context.ExternalContext` ermöglicht es Ihnen, sich unmittelbar auf die Low-Level-Ebene der Servlets zu begeben. Den `ExternalContext` holen Sie mit der Methode `getExternalContext()` der Klasse `FacesContext`:

```
ExternalContext externalContext =
    facesContext.getExternalContext();
```

Um die unterschiedlichsten Funktionalitäten der Servlet-API zu besorgen, bedienen wir uns der zahlreichen Methoden der Klasse `ExternalContext`. Beispielsweise beschafft die folgende Anweisung die Information, ob der aktuelle HTTP-Request über SSL gesichert ist:

```
boolean isSecure = externalContext.isSecure();
```

Application

Die zentrale Instanz der Klasse `javax.faces.application.Application` ist für die gesamte Anwendung als Singleton vorhanden. Die Singleton-Instanz besorgen Sie über die Methode `ApplicationFactory.getApplication()`:

```
Application application = ApplicationFactory.getApplication();
```

Durch den Zugriff auf diese Instanz können Sie die für die Webanwendung globalen Einstellungen steuern. Beispielsweise ändert die folgende Anweisung die Spracheinstellung, die per Default für die Webanwendung gesetzt ist:

```
application.setDefaultLocale(Locale.ENGLISH);
```

9.5.3 Weitere UI-Komponenten

In diesem Abschnitt werden wir uns weitere UI-Komponenten der JSF-HTML-Bibliothek anschauen, die clientseitig als Schaltflächen oder Hyperlinks eingebaut werden können.

Der Output-Link

Genauso wie ein Command-Link wird auch ein Output-Link als HTML-Hyperlink gerendert. Anders als bei einem Command-Link setzen Sie die URL, auf die der Hyperlink verweist, aber hier mit dem Attribut `value`, und den Namen des Links setzen Sie zwischen dem öffnenden und dem schließenden Tag.

Die folgende UI-Komponente setzt einen Link auf eine externe URL:

```
<h:outputLink value="http://www.java2enterprise.de">
    Java2Enterprise
</h:outputLink>
```

Das gerenderte HTML-Element sieht wie folgt aus:

```
<a href="http://www.java2enterprise.de">
    Java2Enterprise
</a>
```

Dem Output-Link können Sie auch Parameter mitgeben. Hierfür stecken Sie die UI-Komponente `param` ins Innere des Output-Links:

```
<h:outputLink value="http://www.java2enterprise.de">
    Java2Enterprise
    <f:param name="coming_from" value="marktplatz_de"/>
</h:outputLink>
```


Das gerenderte HTML-Element sieht wie folgt aus:

```
<a href=
"http://www.java2enterprise.de?coming_from=marktplatz_de">
    Java2Enterprise
</a>
```

Obwohl der Output-Link genauso wie ein Command-Link als HTML-Anker-Element gerendert wird, unterscheiden sich die beiden UI-Komponenten gravierend, denn statt eines POST-Requests handelt es sich bei einem Output-Link um einen GET-Request. Deshalb braucht ein Command-Link auch nicht in eine Form-Komponente platziert zu werden. Es wird auch nicht das JSF-Navigationssystem oder eine JSF-Aktion ausgelöst. Diese UI-Komponente eignet sich deshalb lediglich für den Aufruf einer externen URL. Um dennoch eine JSF-Aktion auszuführen, könnten Sie auch Folgendes programmieren:

```
<h:outputLink value="#{facesContext.externalContext.applicationContextPath}/
signin.xhtml">
```

Link

Mit den beiden UI-Komponenten Command-Link und Output-Link können wir bereits die meisten Anwendungsfälle abdecken, bei denen auf der Benutzeroberfläche ein Hyperlink gerendert werden soll. Während der Command-Link Formularelemente über einen POST-Request an eine JSF-Aktion abschickt, rendert der Output-Link ein ganz einfaches HTML-Anker-Element für eine externe URL. Was jetzt noch fehlt, ist ein Link, der einerseits genauso wie ein Command-Link den Navigationsmechanismus von JSF auslöst, der aber andererseits hierbei als HTTP-GET-Request versendet wird, denn auf diese Weise kann die URL beispielsweise als Lesezeichen gesetzt werden. Und genau für diesen Zweck dient die UI-Komponente `link`. Die URL wird hierbei bereits in der ersten Phase des `INITIAL_REQUEST` ermittelt, sodass sie gleich beim ersten Aufruf angezeigt wird.

Über das Attribut `value` tragen Sie den anzuzeigenden Text für den Link ein. Den Outcome, der für die Navigation verwendet werden soll, setzen Sie über das Attribut `outcome`. Genauso wie bei einem Command-Link können Sie hier eine Aktionsmethode, einen vorkonfigurierten Outcome oder auch eine View-ID eintragen. Das ist praktisch, weil der Anwender hier quasi ein Lesezeichen auf eine Funktion setzt, deren Implementierung der Entwickler im Hintergrund ändern kann.

```
<h:link
    value="Suchen"
    outcome="#{searchController.start()}"/>
```

Listing 9.32 index.xhtml

Button

Die Entwickler des JSF-Frameworks haben noch eine weitere UI-Komponente namens `button` verwirklicht, die vom Prinzip her der UI-Komponente Link gleicht. Auch hierbei wird eine Schaltfläche bereits im `INITIAL_REQUEST` so aufbereitet, dass sie genauso wie die UI-Komponente `link` einerseits den Navigationsmechanismus von JSF nutzt, aber andererseits einen GET-Request auslöst. Auch bei dieser UI-Komponente gelingt dies, indem der Mausklick auf die Schaltfläche per `oncick`-Attribut aufgefangen und ein spezielles JavaScript-Programm des JSF-Frameworks aufgerufen wird. Ein `form`-Element ist deshalb nicht erforderlich.

```
<h:button
    value="Suchen"
    outcome="#{searchController.start()}"/>
```

Listing 9.33 index.xhtml

SelectBooleanCheckbox

Mithilfe der `selectBooleanCheckBox` wird eine Checkbox gerendert, die zwischen zwei Werten wechseln kann (siehe Abbildung 9.20). In Listing 9.34 wird eine `selectBooleanCheckbox` gesetzt. Der Wert wird über das Attribut `value` eingetragen.

```
    <h:form>
        <h:selectBooleanCheckbox
            value="#{testController.a}"/>
        <h:commandButton value="Submit"/>
    </h:form>
    <h:outputText value="#{testController.a}"/>
```

Listing 9.34 index.xhtml

Wenn Sie das obige Beispiel ausprobieren möchten, werden Sie auch folgende Backing Bean benötigen. In der Backing Bean wird eine Property des Typs `Boolean` gesetzt:

```
package de.java2enterprise.onlineshop;

import java.io.Serializable;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class TestController implements Serializable {
    private static final long serialVersionUID = 1L;
```



```
    private Boolean a;

    public Boolean getA() {
        return a;
    }

    public void setA(Boolean a) {
        this.a = a;
    }
}
```

Listing 9.35 TestController.java

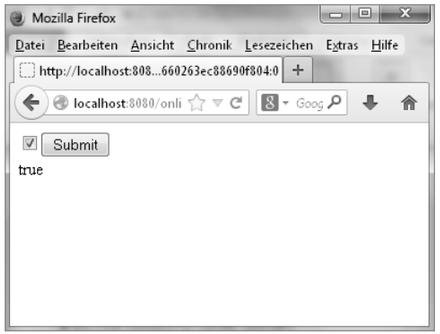

Abbildung 9.20 Die »SelectBooleanCheckbox«

SelectOneRadio, SelectOneListbox und SelectOneMenu

`selectOneRadio` rendert eine Liste von Radio-Buttons, von denen ein einziger selektiert sein kann (siehe Abbildung 9.21). Es gibt verschiedene Möglichkeiten, die Elemente dieser Komponente zur Verfügung zu stellen. Die handlichste Alternative zeige ich in Listing 9.36. Hierbei wird ein Child-Element des Typs `SelectItems` eingesetzt, das die Elemente der Radio-Buttons zur Verfügung stellt.

```
...
<h:form>
    <h:selectOneRadio
        value="#{testController.day}">
        <f:selectItems
            value="#{testController.days}"/>
    </h:selectOneRadio>
    <h:commandButton value="Submit"/>
```

```
</h:form>
<h:outputText value="#{testController.day}"/>
...
```

Listing 9.36 test.xhtml

In der Backing Bean werden zwei Properties programmiert. Eine Property hält den aktuellen Wert des Typs `String`. Die andere Property hat den Typ `java.util.List<String>`. Sie bietet die Liste aller zur Verfügung stehenden String-Werte an:

```
package de.java2enterprise.onlineshop;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import javax.annotation.PostConstruct;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class TestController implements Serializable {
    private static final long serialVersionUID = 1L;

    private List<String> days;

    private String day;

    public List<String> getDays() {
        return days;
    }

    public void setDays(List<String> days) {
        this.days = days;
    }

    public String getDay() {
        return day;
    }

    public void setDay(String day) {
        this.day = day;
    }
```



```
    @PostConstruct
    public void init() {
        days = new ArrayList<String>();
        days.add("Montag");
        days.add("Dienstag");
        days.add("Mittwoch");
        days.add("Donnerstag");
        days.add("Freitag");
        days.add("Samstag");
        days.add("Sonntag");
    }
}
```

Listing 9.37 test.xhtml

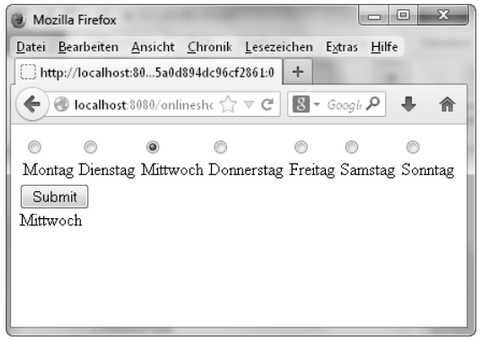

Abbildung 9.21 Die UI-Komponente »SelectOneRadio«

Die UI-Komponenten `selectOneListBox` und `selectMenu` werden gleichermaßen programmiert. In Listing 9.38 sehen Sie die erforderliche Änderung in hervorgehobener Schrift:

```
<h:form>
    <h:selectOne Listbox
        value="#{testController.day}">
        <f:selectItems
            value="#{testController.days}"/>
    </h:selectOne Listbox>
    <h:commandButton value="Submit"/>
</h:form>
```

Listing 9.38 test.xhtml

Wie Sie in Abbildung 9.22 erkennen, hat die kleine Änderung eine große Wirkung.

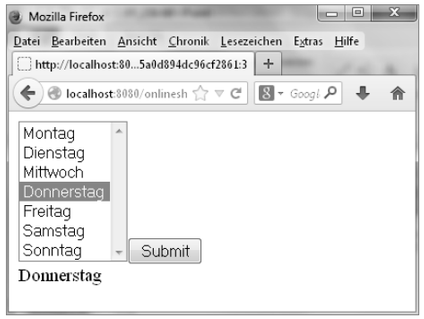

Abbildung 9.22 »SelectOneListbox«

SelectManyCheckbox, SelectManyListbox und SelectManyMenu

Die UI-Komponenten `selectManyCheckbox`, `selectManyListbox` und `selectManyMenu` gleichen den soeben gezeigten `selectOne`-Gegenstücken. Der einzige Unterschied besteht darin, dass die `selectMany`-Komponenten eine Mehrfachauswahl ermöglichen.

```
<h:form>
    <h:selectManyListbox
        value="#{testController.selected}">
        <f:selectItems
            value="#{testController.days}"/>
    </h:selectManyListbox>
    <h:commandButton value="Submit"/>
    <h:outputText value="#{testController.selected}"/>
</h:form>
```

Listing 9.39 text.xhtml

Bei der Backing Bean wird die auswählbare Property nun ebenfalls als `java.util.List<String>` programmiert (siehe Abbildung 9.23).

Abbildung 9.23 Die UI-Komponente »SelectManyListbox«

9.5.4 Die »faces-config.xml« verschieben und aufteilen

Wie bereits angemerkt, liegt die Datei üblicherweise gemeinsam mit dem Deployment-Deskriptor im Verzeichnis */WEB-INF*. Es obliegt der Entscheidung des Entwicklerteams, sie an einem anderen Platz abzulegen. Beispielsweise könnten wir die Datei *faces-config.xml* verlagern, und zwar in ein Unterverzeichnis mit dem Namen /faces unterhalb des Web-Wurzelverzeichnisses (*/WebContent*). Um dies zu verwirklichen, müssten wir das JSF-Framework über den Deployment-Deskriptor *web.xml* hierüber in Kenntnis setzen. Der folgende Ausschnitt aus einem Deployment-Deskriptor zeigt an, wie Sie dies in der JSF-Konfigurationsdatei definieren würden.

```
<context-param>
    <param-name>
        javax.faces.application.CONFIG_FILES
    </param-name>
    <param-value>
        /faces/faces-config.xml
    </param-value>
</context-param>
...
```

Listing 9.40 »web.xml« mit dem Eintrag des Standortes für die »faces-config.xml«

Für unser einfaches Onlineshop-Beispiel reicht es aber vollkommen aus, wenn Sie die JSF-Konfigurationsdatei im Verzeichnis */WEB-INF* belassen.

Neben der soeben gezeigten Abwandlung besteht die Möglichkeit, die JSF-Konfigurationen auf mehrere JSF-Konfigurationsdateien zu verteilen. Das ist vor allem dann sinnvoll, wenn verschiedene *jar*-Bibliotheken mit benutzerdefinierten, d. h. selbst programmierten Komponenten eingebunden werden sollen. Solche *.jar*-Bibliotheken werden in der klassischen Java EE-Anwendung im Verzeichnis */WEB-INF/lib* abgelegt. Deshalb sucht das JSF-Framework beim Start der Anwendung automatisch auch in den */META-INF*-Verzeichnissen der *.jar*-Bibliotheken nach einer JSF-Konfigurationsdatei. In diesem Fall muss eine explizite Standortangabe über den Deployment-Deskriptor nicht mehr angezeigt werden.