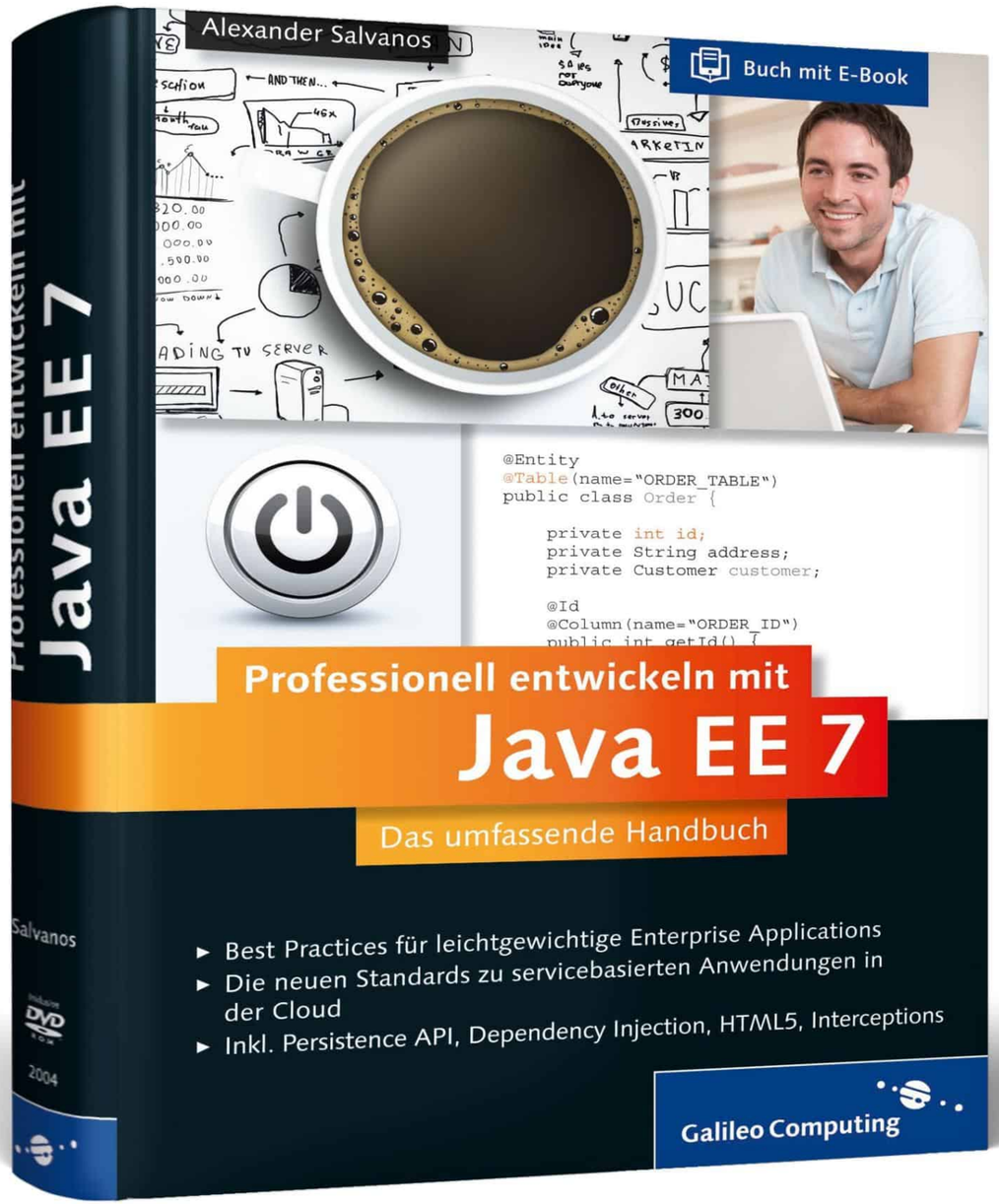


Leseprobe

Dieses Buch ist die zuverlässige Wissensquelle für alle Java-Enterprise-Entwickler. In dieser Leseprobe erhalten Sie eine Einführung in den Java EE-Server und erfahren alles zur Java Database Connectivity (JDBC). Außerdem können Sie einen Blick in das vollständige Inhalts- und Stichwortverzeichnis des Buches werfen.

»Der Java EE-Server«
»JDBC«

Inhalt

Index

Der Autor

Leseprobe weiterempfehlen

Alexander Salvanos

Professionell entwickeln mit Java EE 7 – Das umfassende Handbuch

1.096 Seiten, gebunden, Juni 2014
49,90 Euro, ISBN 978-3-8362-2004-0

www.galileo-press.de/3250

Kapitel 2
Der Java EE Server

»Es ist nicht genug, zu wissen, man muss auch anwenden.«

Johann Wolfgang von Goethe

Der Java EE Server ist die Voraussetzung für den Betrieb einer Java-EE-Anwendung. Gelegentlich wird der Java EE Server auch als »Java-EE-Applikationsserver« oder einfach nur als »Applikationsserver« bezeichnet. In diesem Buch wird der Begriff *Java EE Server* verwendet.

Java EE Server werden von diversen Softwareherstellern gemäß den Vorgaben der Java-EE-Spezifikation angefertigt. Jeder Softwarehersteller entwickelt hierbei seinen eigenen individuellen Java EE Server. Allerdings werden nur diejenigen Java EE Server als Java-EE-konform zertifiziert, die die Anforderungen der Java-EE-Spezifikation vollständig abdecken.

In diesem Kapitel wird gezeigt, wie Sie den *GlassFish Server Open Source Edition 4* als Java-EE-7-Server auf Ihrem Rechner installieren und konfigurieren. Anschließend werden wir auch eine minimale Java-EE-Anwendung erstellen und diese in den Java EE Server installieren. Darüber hinaus wird in diesem Kapitel auch gezeigt, wie GlassFish in einer integrierten Entwicklungsumgebung eingebunden wird. Es wird der Einsatz mit Eclipse und in einem Extra-Abschnitt auch mit NetBeans gezeigt.

2.1 Die Installation

Um den GlassFish-Server zu installieren, installieren Sie zunächst ein *Java Development Kit* und dann den Server selbst.

2.1.1 Das JDK 7 installieren

Als Java-Entwickler ist Ihnen selbstverständlich bekannt, dass jegliches Arbeiten mit Java-Technologien auch eine *Java SE Runtime Engine* (JRE) voraussetzt. Für die Entwicklung ist darüber hinaus auch ein Java-Compiler erforderlich. Aus diesem Grund benötigen wir ein komplettes *Java SE Development Kit* (JDK), denn in ihm ist beides vorhanden, also sowohl die JRE als auch der Java-Compiler.

Für die Arbeiten mit dem Java EE Server *GlassFish 4* benötigen Sie ein Java Development Kit der Version 7u25. Wenn Sie das JDK in dieser oder in einer neueren Version bereits installiert haben, können Sie den folgenden Abschnitt überspringen.

Derzeit erreichen Sie die Java-SE-Download-Seite über folgenden Link:

http://www.oracle.com/technetwork/java/javase/downloads

Wenn Sie das JDK für Ihr Betriebssystem heruntergeladen und installiert haben, müssen Sie noch die Umgebungsvariablen `JAVA_HOME`, `PATH` und `CLASSPATH` setzen.

Als Java-Entwickler werden Ihnen die Unterschiede zwischen Unix-basierten und Windows-basierten Betriebssystemen bekannt sein. Beispielsweise könnte der Pfad auf einem Unix-basierten Rechner (Linux, Solaris, MacOS etc.) so aussehen: */etc/jdk.1.7.0_25*, bei einem Windows-basierten Rechner hingegen so: *C:\jdk1.7.0_25*. (Beachten Sie den Hinweis zum Betriebssystem MacOS im Vorwort dieses Buches.)

- `JAVA_HOME`: Setzen Sie Ihr JDK-Installationsverzeichnis als Wert der Umgebungsvariablen `JAVA_HOME`, damit die nachfolgenden Installationen von GlassFish und Eclipse reibungsfrei funktionieren.
- `PATH`: Die Anpassung der Umgebungsvariablen `PATH` ist für die Verwendung der Textkonsole wichtig. Erweitern Sie Ihre `PATH`-Umgebungsvariable um das Installationsverzeichnis mit einem angehängten `\bin`. Bei Windows ist das *C:\jdk1.7.0_25\bin*. Beachten Sie als Nutzer eines UNIX-basierten Betriebssystems wie Linux, Solaris oder OS X wieder die Unterschiede der Pfadangaben.
- `CLASSPATH`: Als Wert der Umgebungsvariable `CLASSPATH` setzen Sie einen Punkt. Hiermit weisen Sie Java an, im aktuellen Verzeichnis nach Java-Dateien zu suchen.

Sie können testen, ob alles richtig installiert ist, indem Sie (nach dem Setzen der Umgebungsvariablen) eine Textkonsole öffnen und `java -version` eingeben. Wenn Java nicht gefunden werden kann oder Java eine falsche Version anzeigt, sollten Sie dies erst korrigieren, bevor Sie mit der Installation von GlassFish fortfahren.

Auf manchen Rechnern sind mehrere Java-Versionen gleichzeitig installiert. Dies ist theoretisch kein Problem. Allerdings führt es häufig zu einem großen Durcheinander. Wenn man mit mehreren Java-Versionen arbeitet, ist man gezwungen, sehr diszipliniert zu arbeiten, und man braucht genaue Kenntnisse darüber, wann was ausgeführt wird. Unterliegen Sie jedoch keinen Zwängen, deinstallieren Sie lieber vorher alle alten Java-Versionen, bevor Sie mit der Installation des JDK 7 beginnen. Dies ist der sauberste Weg.

2.1.2 »GlassFish Server Open Source Edition 4« herunterladen

Oracle bietet den Java EE Server *GlassFish* in zwei unterschiedlichen Varianten an: kostenpflichtig und nicht kostenpflichtig.

Wenn Sie die URL *https://glassfish.java.net/* in Ihrem Webbrowser aufrufen, sehen Sie im unteren Bereich einen Link zum kostenpflichtigen Oracle GlassFish Server (siehe Abbildung 2.1).

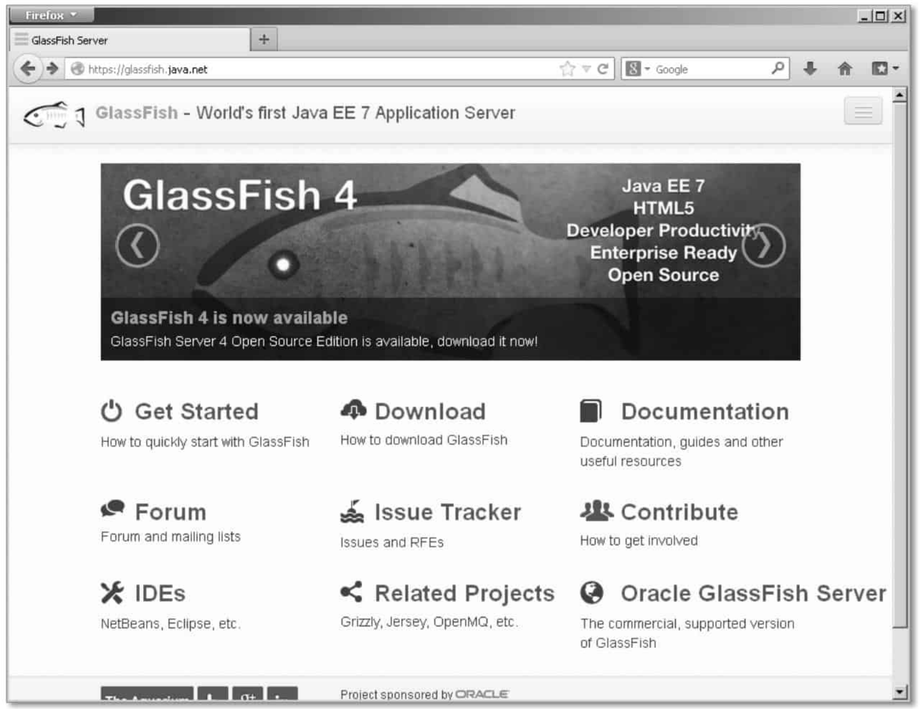


Abbildung 2.1 Die Willkommensseite von GlassFish

Oracle empfiehlt die Verwendung dieses kommerziellen Produkts für eine produktive Umgebung, das heißt für den tatsächlichen Betrieb im professionellen Umfeld. Sollten Sie sich für die Anschaffung der kostenpflichtigen Version entscheiden, unterstützt das Unternehmen Oracle Sie bei der Benutzung des Java EE Servers durch einen gesonderten Support. Die kostenfreie Version nennt sich *GlassFish Server Open Source Edition 4* und bietet diesen Vorteil nicht. Im Kern handelt es sich aber um das gleiche Produkt.

Über einen Mausklick auf den Link Download gelangen Sie zu einer Download-Seite, auf der Sie verschiedene GlassFish-Varianten in sechs Schritten zusammenstellen können (siehe Abbildung 2.2).

Der GlassFish-Download-Website können Sie entnehmen, dass sich die Installationssoftware sowohl als gezippte Datei wie auch als ausführbarer Installer herunterladen lässt.

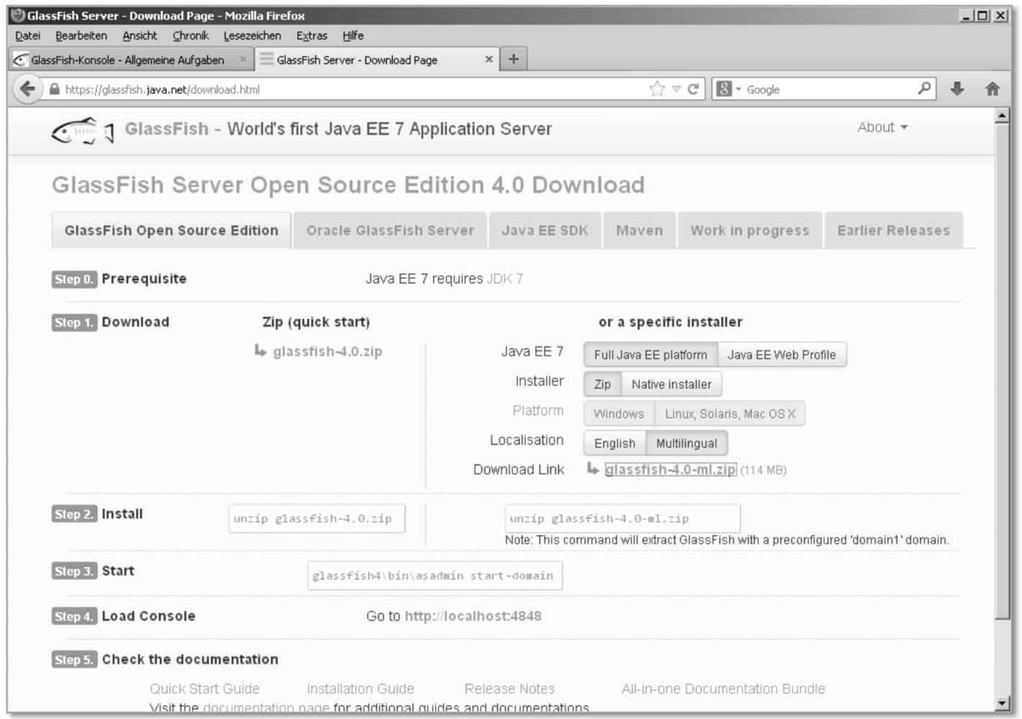


Abbildung 2.2 Die Downloads des GlassFish-Servers

Bei der gezippten Datei profitieren Sie von der kurzen Installationsdauer, denn sie enthält ein komplett vorkonfiguriertes GlassFish-Verzeichnis. Darin enthalten ist ein betriebssystemunabhängiger GlassFish-Server. Mit ihm können Sie direkt loslegen. Weil Ihnen viel Arbeit abgenommen wird, eignet er sich vor allem dann, wenn Sie Zeit sparen möchten. Ihnen bleibt es jedoch nicht erspart, über viele der voreingestellten Konfigurationen Bescheid zu wissen. Aber für dieses Know-how wird gleich gesorgt, denn direkt nach der Installation werden wir detailliert auf die wichtigsten Konfigurations- und Einstellungsmöglichkeiten des GlassFish-Servers eingehen.

Wählen Sie auf der Download-Website nun die FULL JAVA EE PLATFORM aus, denn schließlich brauchen wir einen vollständig Java-EE-konformen Server. Darunter wählen Sie ZIP und MULTILINGUAL aus. Klicken Sie dann auf den Link *glassfish-4.0-ml.zip*, um die Installationsdatei herunterzuladen. In der *.zip*-Datei befindet sich ein Verzeichnis, das *\glassfish4* heißt. Extrahieren Sie dieses Verzeichnis in ein hierfür geeignetes Verzeichnis. Im Rahmen dieses Buches wurde bei dem Betriebssystem Microsoft Windows das Verzeichnis *C:* verwendet. Das bedeutet, dass Sie die Beschreibungen und die Screenshots aus diesem Buch bestmöglich nachvollziehen werden, wenn Ihre Installation mit dem Betriebssystem Microsoft Windows unter dem Ordner *C:\glassfish4* abgespeichert ist. Aber ein anderes Betriebssystem und ein anderer Ordner sind genauso verwendbar. Beachten Sie, dass Sie für den Ordner die vollen Schreib- und Leserechte benötigen.

2.1.3 Die Verzeichnisse von GlassFish 4

Jetzt, da die Installation von GlassFish 4 erfolgreich abgeschlossen ist, sollte die Extraktion der *.zip*-Datei unter Windows ein Verzeichnis mit dem Namen *C:\glassfish4* erzeugt haben. Bevor es gleich mit der Verwaltung des GlassFish-Servers losgeht, gibt dieser Abschnitt Ihnen einen groben Überblick über die Verzeichnisstruktur von GlassFish. Auf Ihrem Rechner begeben Sie sich nun in das Verzeichnis *C:\glassfish4* (siehe Abbildung 2.3).

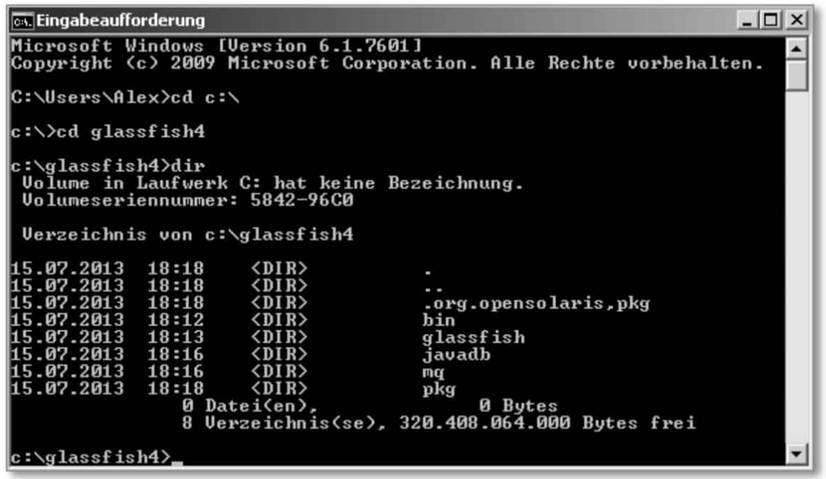


Abbildung 2.3 Das Verzeichnis des GlassFish-Servers

Dieses Verzeichnis stellt den äußeren Rahmen oder auch das sogenannte Parent-Verzeichnis der Installation dar. Wie Sie gleich sehen werden, befindet sich das eigentliche GlassFish-Installationsverzeichnis in einem seiner Unterordner. Diese sind:

- **\bin**
  Das Verzeichnis *\bin* enthält einige Administrationskommandos von GlassFish. Dieses Verzeichnis ist jedoch nicht von großem Belang, denn die vollständige Liste aller Administrationskommandos ist in einem anderen Unterverzeichnis abgespeichert.
- **\glassfish**
  Das Unterverzeichnis *c:\glassfish4\glassfish* stellt das eigentliche Installationsverzeichnis des Java EE Servers dar. Aus diesem Grund wird es in den GlassFish-eigenen Skripten unter der Umgebungsvariablen `AS_INSTALL` abgespeichert. `AS_INSTALL` steht für *Application Server Installation*.

Die wichtigsten Unterordner des Verzeichnisses *c:\glassfish4\glassfish* sind:

- **\bin**
  Das Verzeichnis *\bin* enthält die Administrationskommandos von GlassFish. Das Verzeichnis *c:\glassfish4\glassfish\bin* sollte als Eintrag in die Umgebungsvariable PATH gesetzt werden. Dies ist für den Betrieb des GlassFish-Servers aber keine zwingende Voraussetzung, sondern vereinfacht lediglich die Arbeit mit der Kommandozeile. Denn hierdurch lassen sich anschließend die Kommandos des Java EE Servers von überall aus aufrufen.
- **\config**
  Das Verzeichnis *\config* enthält u. a. das Skript *asenv.bat*, das weitere Umgebungsvariablen für den GlassFish-Server setzt.
- **\docs**
  Das Verzeichnis *\docs* enthält ein Tutorial für die Benutzung des GlassFish-Servers.
- **\domains**
  In diesem Verzeichnis werden sogenannte Domänen untergebracht. Das Verzeichnis *\domains* wird deshalb im Fachjargon auch als *domain-root-dir* bezeichnet. Auf Domänen wird im nächsten Abschnitt ausgiebig eingegangen.
- **\legal**
  Im Verzeichnis *\legal* finden Sie die Lizenzvereinbarungen zum GlassFish-Server.
- **\lib**
  Das Verzeichnis *\lib* enthält die Java-Bibliotheken des GlassFish-Servers. Der GlassFish-Server ist selbst in der Programmiersprache Java geschrieben worden. Deshalb werden Sie hier zahlreiche *.jar*-Dateien vorfinden, die beim Start des GlassFish-Servers ausgeführt werden.
- **\modules und \osgi**
  GlassFish enthält Dienste, die durch OSGi als Module ferngewartet werden können. OSGi ist die Abkürzung für *Open Service Gateway initiative*. Die Module bestehen aus *.jar*-Files, die in sogenannten OSGi-Bundles zur Laufzeit eingespielt, installiert, administriert und auch wieder entfernt werden. Die *.jar*-Files werden in das Verzeichnis *\modules* gespeichert. Das Verzeichnis *\osgi* enthält dagegen das quelloffene OSGi-Framework *Apache Felix*.
- **\javadb**
  Das Verzeichnis *\javadb* enthält ein relationales Datenbankmanagementsystem, das ehemals den Namen *Derby* trug.
- **\mq**
  Das Verzeichnis *\mq* enthält die Ressourcen des JMS-Providers *Open MQ*. Durch JMS können Nachrichten gesendet und auch empfangen werden. Auch hierdurch wird eine lose Kopplung von verteilten Systemen ermöglicht.
- **\pkg**
  Der Ordner *\pkg* enthält einen Unterordner *\pkg\javadocs* und *\pkg\lib*. Den Unterordner *\pkg\lib* benötigt GlassFish für den Installations- bzw. Deinstallationsprozess selbst. Es enthält Dateien zum Framework *Image Packaging System* (IPS). IPS wird zur Erstellung von Paketstrukturen, Installationsverzeichnissen und für die Installationskonsistenz verwendet. Mit anderen Worten: Als Java-EE-Entwickler werden Sie mit diesem Verzeichnis kaum Berührungspunkte haben. Im Unterordner *\pkg\javadocs* befinden sich JavaDocs zum IPS-System.

2.2 Domänen und Instanzen

Wenn Sie auf einem GlassFish-Server eine Java-EE-Anwendung installieren, wird sie nicht direkt im Server erzeugt, sondern in einer sogenannten Domäne.

2.2.1 Domänen

Domänen stellen einen administrierbaren Namensraum dar (siehe Abbildung 2.4). Jede Domäne enthält eine eigene Administrationsanwendung, die durch eine Benutzer-Passwort-Kombination geschützt werden kann.

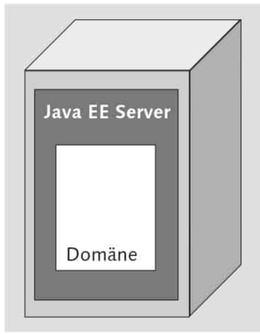


Abbildung 2.4 Eine Domäne

Innerhalb eines GlassFish-Servers können beliebig viele Domänen erzeugt werden. In der Regel richtet man jedoch nur eine einzige Domäne in einem GlassFish-Server ein.

Domänen sind die Java-EE-Maschinen, in denen zum Beispiel Web-Container und EJB-Container laufen. In mancher Hinsicht kann man die Domänen sogar als die eigentlichen Java EE Server betrachten.

2.2.2 Instanzen

Der Hersteller von GlassFish hat sich ein weiteres Konfigurationskonzept ausgedacht, denn der Server arbeitet mit sogenannten Instanzen. Eine Instanz kann man als Laufzeitumgebung für alle Module und Komponenten betrachten, die sich darin befinden.

Eine Instanz wird aber auch als *Konfigurationsinstanz* bezeichnet, denn hiermit wird treffend ausgedrückt, was eine Instanz eigentlich auch ist. Zu jeder Domäne lassen sich beliebig viele Konfigurationsinstanzen einrichten, wobei jede Konfigurationsinstanz zur Laufzeit nur einer einzigen Domäne zugewiesen werden kann.

Damit die Komponenten einer Java-EE-Anwendung in einem Web- und einem EJB-Container verwaltet werden können, muss die Java-EE-Anwendung auf einer konkreten Instanz deployt werden (siehe Abbildung 2.5).

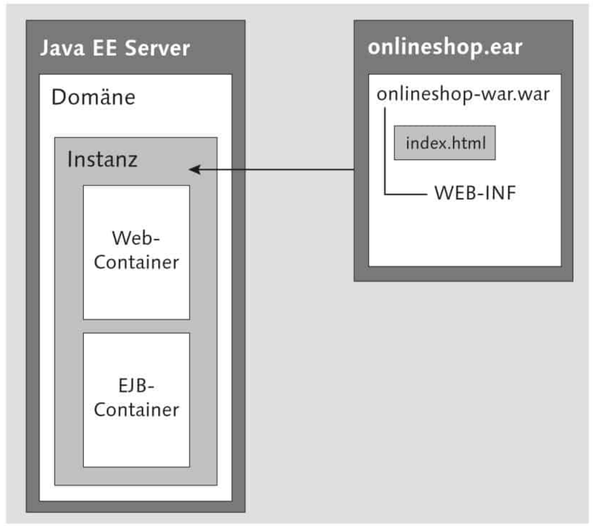


Abbildung 2.5 Das Deployment basiert auf einer Konfigurationsinstanz.

Im Allgemeinen unterscheidet man zwischen:

- Standalone-Instanzen (eigenständigen Instanzen) und
- Cluster-Instanzen.

Standalone-Instanz

Die eigenständige Instanz wird auf einem einzelnen, alleinstehenden Rechner installiert. Zum Beispiel verwenden Sie eine eigenständige Instanz, wenn Sie Ihren Server für die Entwicklungstätigkeit einrichten. Mithilfe mehrerer eigenständiger Instanzen kann zwischen verschiedenen Konfigurationen auch hin- und hergewechselt werden. Das ist praktisch, da man somit eine Entwicklungs- oder eine Testkonfiguration einfach ausprobieren kann.

Cluster-Instanz

Während eigenständige Instanzen nicht mit anderen Instanzen gebündelt werden können, sind Cluster-Instanzen durchaus dazu in der Lage. Und hier liegt auch der eigentliche Zweck

einer Cluster-Instanz, denn mithilfe von Cluster-Instanzen lassen sich sogenannte Cluster einrichten. Dabei wird jede einzelne Cluster-Instanz auf einem eigenen Rechner eines Rechnerverbunds eingesetzt. Den Rechnerverbund bezeichnet man als *Serverfarm*. Auf einer Serverfarm wird jeder Rechner als sogenannter *Node* installiert. Grundsätzlich unterscheidet GlassFish zwischen *CONFIG-Nodes* und *SSH-Nodes*. In einem produktiven System werden *SSH-(Secure Shell-)Nodes* verwendet, um eine verschlüsselte Verbindung zwischen den *Nodes* innerhalb des Clusters zu gewährleisten. Dabei wird der gesamte Cluster wie ein einzelnes System betrachtet, das von außen über eine zentrale Administrationsanwendung fernverwaltet wird.

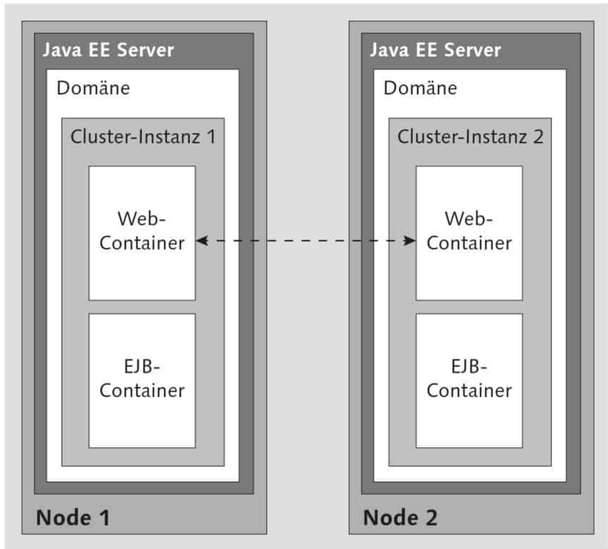


Abbildung 2.6 Cluster-Instanzen

Die Erstellung und Administration eines Clusters ist keine triviale Angelegenheit und für die Entwicklungstätigkeit nicht erforderlich. Jedoch spielt sie in einer produktiven Umgebung eine große Rolle, da hierdurch die Kapazität und die Verfügbarkeit des gesamten Systems verbessert werden kann.

2.3 Die Textkonsole

Die Arbeit mit dem GlassFish-Server beginnt in der Regel mit einem Konsolenprogramm, das sich `asadmin` nennt. `asadmin` ist von entscheidender Bedeutung, da nur auf diesem Weg eine Domäne erzeugt und gestartet werden kann.

`asadmin` ist also eine ausführbare Datei, die unter Windows den Namen *asadmin.bat* trägt (unter Unix-basierten Betriebssystemen: *asadmin*). Dieser Konsolenbefehl befindet sich im Unterverzeichnis *\bin* des GlassFish-Installationsverzeichnisses. Wenn Sie den Pfad *c:\glassfish4\glassfish\bin* als Eintrag in die Umgebungsvariable `PATH` gesetzt haben, sollte *asadmin.bat* von jedem Pfad aus aufrufbar sein.

`asadmin` kann mit einem Optionsparameter aufgerufen werden. Zum Beispiel führt der Aufruf mit `--help` zur Ausgabe eines ausführlichen Benutzerhandbuchs von GlassFish (siehe Abbildung 2.7).



Abbildung 2.7 Der Optionsparameter »--help«

Ruft man hingegen `asadmin` ohne Optionsparameter auf, erhält man eine Textkonsole, über die die Optionsparameter als Kommandos aufgerufen werden können (siehe Abbildung 2.8).

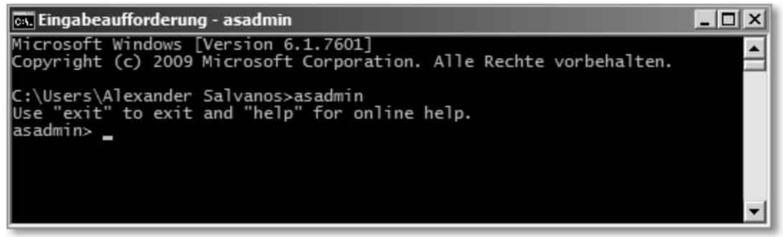


Abbildung 2.8 Die Textkonsole »asadmin«

2.3.1 Eine Domäne erzeugen

Um eine Domäne zu erzeugen, verwendet man das Kommando `create-domain`. Dem Kommando `create-domain` wird der Name der zu erzeugenden Domäne hinzugefügt. Zum Beispiel erzeugen Sie mit folgendem Aufruf eine Domäne, die den Namen *domain1* trägt:

```
create-domain domain1
```

Der Bezeichner *domain1* ist eigentlich auch der Default-Wert. Das bedeutet: Wenn Sie den Bezeichner *domain1* weglassen, wird dieser automatisch gesetzt. GlassFish versucht die Konfiguration des Servers durch Default-Werte zu vereinfachen. Obwohl wir deshalb an vielen Stellen auf das explizite Setzen eines Wertes verzichten können, gebe ich zur Verdeutlichung die Default-Werte im Übungsbeispiel an.

Mit dem Kommando `create-domain` können Sie auch noch weitere Optionsparameter setzen. Auch hierbei führt ein Weglassen zu einer automatischen Konfiguration mit Default-Werten. Zum Beispiel verwendet eine Domäne zahlreiche Ports, um ihre Dienste in einem Netzwerk anzubieten. Wenn Sie die Default-Werte nutzen, ist es wichtig, dass Sie die Default-Ports kennen, um Konflikte zu vermeiden.

Der wichtigste Port, der bei der Erzeugung einer Domäne gesetzt werden muss, ist der `adminport`. Der `adminport` ist von zentraler Bedeutung, da er für den Aufruf des *Domain Administration Servers* (DAS) über einen Webbrowser verwendet wird. Auf den *DAS* gehen wir in Abschnitt 2.4 ein. Der Default-Wert für den `adminport` ist 4848.

Ein weiterer Port, der von wesentlicher Bedeutung ist, nennt sich `instanceport`. Mit dem `instanceport` legen Sie fest, über welchen HTTP-Port die Web-Komponenten Ihrer Java-EE-Anwendung erreichbar sein sollen. Der Default-Wert ist 8080. Da dieser Port auch von *Apache Tomcat* und sogar von dem Administrationstool des relationalen Datenbank-Management-Systems Oracle verwendet wird, könnte es hierbei zu einem Konflikt auf Ihrem Rechner kommen. In so einem Fall setzen Sie den `instanceport` auf einen noch freien Wert.

Alle Default-Werte können Sie Tabelle 2.1 entnehmen:

Dienst	Port
IIOP	3700
IIOP SSL	3820
IIOP MUTUAL AUTH	3920
WEB CONSOLE	4848
OSGI SHELL	6666
JMS	7676
HTTP	8080
HTTPS	8181
JMX	8686
JAVA DEBUGGER	9009

Tabelle 2.1 Default-Werte für »instanceport«

Spätestens wenn Sie mehrere Domains erstellen, müssen Sie für jede Domäne alle Ports per Optionsparameter ändern. Diese Tätigkeit könnte ganz schön mühsam werden. Zur Vereinfachung bietet GlassFish einen weiteren Optionsparameter an, der sich `portbase` nennt. Wenn Sie `portbase` als Optionsparameter verwenden, geben Sie lediglich die ersten zwei Ziffern für alle Ports an. Zum Beispiel könnten Sie mit folgendem Kommando dafür sorgen, dass alle Ports für *domain1* mit 77 beginnen:

```
create-domain --portbase 77 domain1
```

Die übrigen zwei Ziffern fügt GlassFish selbst hinzu. Die Ports der einzelnen Dienste wären in diesem Fall wie folgt:

Dienst	Port
IIOP	7700
IIOP SSL	7720
IIOP MUTUAL AUTH	7720
WEB CONSOLE	7748

Tabelle 2.2 Portnummern ändern Sie mit »portbase«.

Dienst	Port
OSGI SHELL	7766
JMS	7776
HTTP	7780
HTTPS	7781
JMX	7786
JAVA DEBUGGER	7709

Tabelle 2.2 Portnummern ändern Sie mit »portbase«. (Forts.)

Mit dem folgenden Kommando würden Sie eine Domäne mit dem Namen *domain1* mit `adminport 4848` und `instanceport 8080` erzeugen.

```
create-domain --adminport 4848 --instanceport 8080 domain1
```

Sie brauchen dieses Kommando aber nicht mehr auszuführen, da GlassFish dies in der gezippten Installationsdatei bereits für Sie vorgenommen hat.

Das Kennwort von »admin«

Wenn Sie mit `create-domain` eine Domäne erzeugen, werden Sie aufgefordert, ein Kennwort für den Default-Administrator einzugeben. Der Administrator trägt den Namen »admin«. GlassFish hat aber auch diesen Schritt bereits für uns erledigt. Damit wir mit der Arbeit sofort loslegen können, hat GlassFish als Kennwort für den Default-Administrator eine leere Zeichenkette gesetzt.

Das Verzeichnis von »domain1«

Wenn Sie sich auf Ihrem Rechner nun unterhalb des Installationsverzeichnisses von GlassFish das Verzeichnis *\domains* anschauen, werden Sie dort einen Unterordner mit dem Namen *\domain1* vorfinden. Dieses Verzeichnis steht stellvertretend für die Domäne *domain1*.

Unterhalb des Verzeichnisses *\domain1* befinden sich mehrere Unterverzeichnisse, von denen einige noch leer sind. Zu diesem Zeitpunkt sind folgende Verzeichnisse von Bedeutung:

- **\autodeploy**
  Das Verzeichnis *\autodeploy* ist die zentrale Stelle für das sogenannte *Hot Deployment* von GlassFish.

Mit *Hot Deployment* ist gemeint, dass der GlassFish-Server während des Betriebs auf Änderungen in dem Verzeichnis *\autodeploy* achtet.

Eine Java-EE-Anwendung besteht aus einer festgelegten Verzeichnisstruktur, die in der Regel als gezippte Datei paketiert wird. Sobald eine solche Datei in das *\autodeploy*-Verzeichnis gespeichert wird, betrachtet das der GlassFish-Server als Neuinstallation. Eine Löschung dieser Datei ist für den GlassFish-Server eine Deinstallation. In einem späteren Übungsbeispiel wird das *Hot Deployment* mit dem Verzeichnis *\autodeploy* noch gezeigt.

- **\config**
  Im Unterverzeichnis *\config* finden Sie die Konfigurationsdateien der Domäne *domain1* wieder.
- **\docroot**
  Das Unterverzeichnis *\docroot* stellt das Wurzelverzeichnis der Domäne dar. Hier befindet sich eine Datei mit dem Namen *index.html*. Sie sollte im Webbrowser angezeigt werden, wenn die Domäne gestartet worden ist und die URL-Adresse der Domäne (beispielsweise *http:/localhost:8080*) aufgerufen wurde.

2.3.2 Der Start einer Domäne

Um den GlassFish-Server in Betrieb zu nehmen, starten Sie eine Domäne. Das bedeutet in unserem Fall, dass wir die Domäne *domain1* zum Laufen bringen müssen. Hierzu verwenden wir folgendes Kommando:

```
start-domain domain1
```

In Abbildung 2.9 sehen Sie die Ausgaben, die beim Start der Domäne *domain1* auf meiner Konsole geschrieben werden.

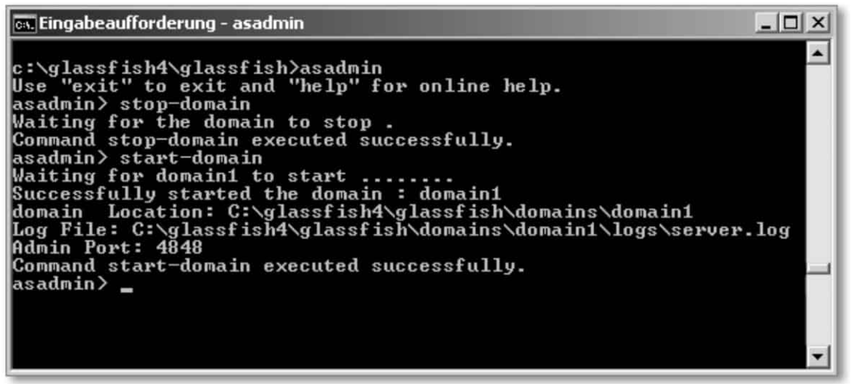


Abbildung 2.9 Das Kommando »start-domain«

Sobald die Domäne in Betrieb ist, kann ihre Startseite aufgerufen werden. Geben Sie hierzu in der Adressleiste Ihres Webbrowsers *http://localhost:8080* ein (siehe Abbildung 2.10).

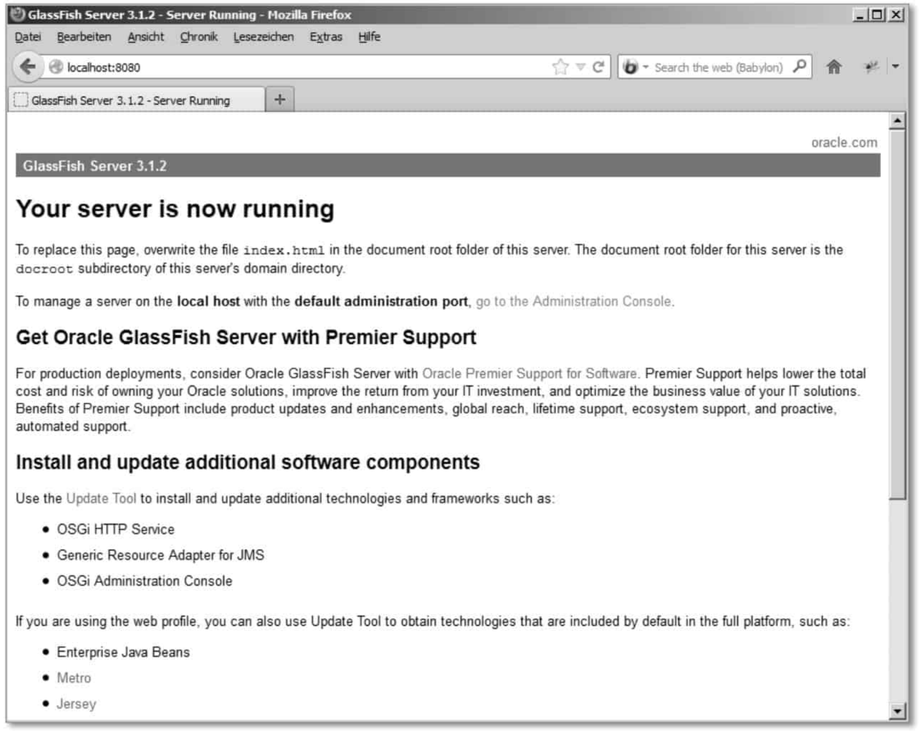


Abbildung 2.10 Die Startseite der Domäne »domain1«

Falls es nicht wie erwartet funktioniert

Wenn das Deployment der Java-EE-Anwendung nicht reibungslos funktioniert, obwohl sich GlassFish ansonsten anstandslos hoch- und herunterfahren lässt, könnte dies an der Verbindungsmöglichkeit über den Hostnamen *localhost* liegen. Versuchen Sie in diesem Fall, die Einstellungen für die Namensauflösung für Hosts auf Ihrem Rechner anzupassen.

Auf einem UNIX-basierten Rechner müssen Sie die Datei */etc/hosts* bearbeiten. Bei dem MacOS ist auch eine Verknüpfung weiter unten, also bei */private/etc/hosts* vorhanden. Bei einem Windows-Rechner ist es die Datei *C:\Windows\System32\drivers\etc\hosts*.

Verbinden Sie die Loopback-IP-Adresse `127.0.0.1` mit dem Namen `localhost`:

```
127.0.0.1      localhost
```

Hinweis

Beim MacOS muss außerdem noch der FQDN des Rechners in der */etc/hosts/* eingetragen werden.

Die Default-Instanz mit dem Namen »server«

Weiter oben wurde bereits gesagt, dass eigentlich jegliches Arbeiten mit einer Domäne auf den Konfigurationen einer Instanz basiert. Als Sie soeben die Domäne gestartet haben, haben Sie bereits eine solche Konfigurationsinstanz verwendet. Dabei handelte es sich um eine Default-Instanz, die aus der Sicht des Administrators den konfigurierbaren Administrationsbereich des Java EE Servers darstellt. Wundern Sie sich deshalb nicht, wenn Sie später mal in den Konfigurationsdateien den Bezeichner »server« sehen sollten, denn dies ist der Alias für diese Default-Instanz. Die Domäne *domain1* wurde von GlassFish so angelegt, dass sie die Instanz mit dem Namen *server* mitverwendet. Dadurch braucht GlassFish keine weitere Konfigurationsinstanz mehr zu erzeugen.

2.4 Der DAS

Sobald Sie eine Domäne gestartet haben, aktivieren Sie ganz automatisch auch einen sogenannten *Domain Administration Server*, der oft auch abgekürzt als *DAS* bezeichnet wird. Jede Domäne beinhaltet einen eigenen DAS, über den sie konfiguriert und administriert werden kann. Obwohl der GlassFish-Server frisch installiert ist, sind bereits zahlreiche Einstellungen vorkonfiguriert. Dadurch ist der GlassFish-Server zur ersten Verwendung bereits vorbereitet. Das ist auf der einen Seite sehr praktisch, hat aber andererseits auch einen Nachteil. Weil der GlassFish-Server zunächst nur für eine Entwicklungsumgebung gerüstet wurde, kann er in diesem Zustand nicht direkt für ein produktives System eingesetzt werden. Aber auch auf einem Entwicklungsrechner sind die vorbelegten Werte nicht immer vorteilhaft und müssen für so manche Java-EE-Anwendung angepasst werden, damit die Leistung im Betrieb des Servers verbessert wird.

2.4.1 Die Bedienung des DAS über »asadmin«

Es wurde bereits angemerkt, dass die Konfiguration und die Administration des GlassFish-Servers über den *Domain Administration Server* (DAS) erfolgt. Die Bedienung des DAS kann entweder auf der Konsole über das `asadmin`-Kommando oder über einen Webbrowser erfolgen. Wir werden zunächst mit `asadmin` arbeiten. Weiter unten wird anschließend die webbasierte Variante eingesetzt.

Bei dem Befehl `asadmin` unterscheidet man zwischen *Local Commands* und *Remote Commands*. Ein *Local Command* haben Sie zum Beispiel verwendet, um die Domäne zu erzeugen

und um sie zu starten. *Remote Commands* hingegen werden an den DAS weitergereicht. Deshalb setzen sie einen aktiven DAS voraus. Daher funktionieren sie auch nur, wenn bereits eine Domäne erzeugt und gestartet worden ist. Durch den Start der Domäne *domain1* wurde auch der DAS aktiviert. Deshalb stehen nun auch die Remote Commands zur Verfügung.

Der DAS bietet zahlreiche Remote Commands an. Alle Remote Commands zu erläutern, würde den Rahmen dieses Kapitels sprengen. Es werden nur einige Kommandos gezeigt, die ein Java-EE-Komponenten-Entwickler im Arbeitsalltag mit seiner Installation benötigt. Durch Eingabe des Kommandos `list-commands` erhalten Sie eine Übersicht über die zur Verfügung stehenden Kommandos (siehe Abbildung 2.11).

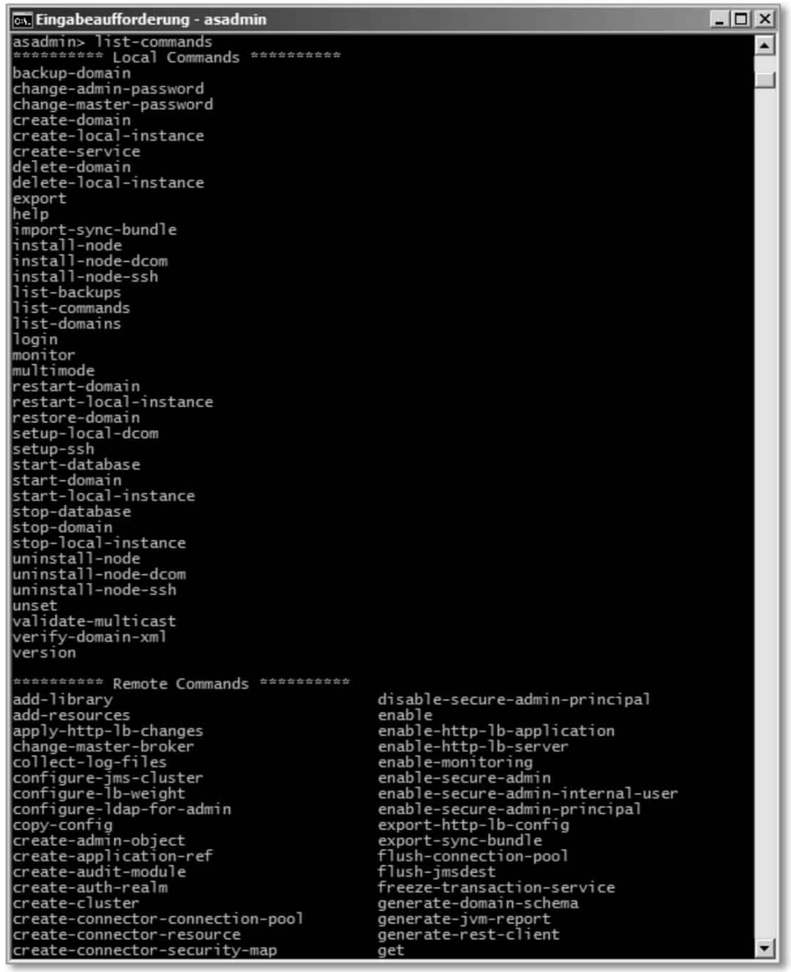


Abbildung 2.11 Auflistung von Local und Remote Commands

Wie Sie der Abbildung entnehmen können, werden die Local Commands und die Remote Commands getrennt aufgeführt.

Die Bedienung des DAS über `asadmin` und seine Remote Commands ist mächtiger als die Bedienung des DAS über den Webbrowser. Sie eignet sich vor allem dann, wenn Sie Kommandos in Shell-Skripten einbeziehen oder andere Administrationsaufgaben erledigen möchten, die über die einfache Konfiguration hinausgehen.

Die Konfiguration des Loggings

Nicht nur für den Administrator, sondern auch für den Entwickler ist das Logging ein wichtiges Instrument der täglichen Arbeit. Der Logging-Mechanismus von GlassFish lässt sich von »ganz schweigsam« bis hin zu »besonders redselig« einstellen. Folgende Logging-Levels lassen sich von niedrig bis hoch wie folgt einstellen: `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER` und `FINEST`. Dabei umfasst ein Level auch die Benachrichtigungen der jeweilig niedrigeren Levels.

Nun können für jede Instanz für verschiedene Packages unterschiedliche Logging-Level gesetzt werden. Gleichzeitig erlaubt GlassFish auch das Setzen eines globalen Logging-Levels. Der globale Logging-Level gilt für alle Logger, wobei die Einstellung eines spezifischen Logging-Levels den globalen Logging-Level für das jeweilige Package überschreibt.

Im Allgemeinen kann man sagen, dass zur Entwicklungszeit ein besonders feiner (redseliger) Logging-Level vorteilhaft ist. Weil das ständige Hinausschreiben von Nachrichten aber auch sehr zeitaufwendig ist, sollte der Logging-Level während des produktiven Betriebs möglichst schweigsam sein.

Für den Onlineshop werden wir den globalen Logging-Level nun auf `FINEST` setzen. Hierfür geben Sie auf der `asadmin`-Konsole Folgendes ein:

```
set-log-levels java.util.logging.ConsoleHandler=FINEST
```

Auf der Konsole sehen Sie, dass der Logging-Level für die Instanz »server« gesetzt wird (siehe Abbildung 2.12).



Abbildung 2.12 Für die Entwicklung setzen wir den Logging-Level der Instanz »server« auf »FINEST«.

2.4.2 Die Bedienung des DAS über einen Webbrowser

Wie bereits angemerkt, bietet der DAS eine grafische Benutzeroberfläche über einen Webbrowser an, um zum Beispiel Java-EE-Anwendungen komfortabel installieren zu können. Der Fachbegriff für die grafische Benutzeroberfläche des DAS lautet *Administration Console*.

Die Administration Console erreichen Sie, indem Sie in der Adressleiste Ihres Webbrowsers die URL *http://localhost:4848* eingeben. Wenn Sie ein Kennwort für den Administrator festgelegt haben, erscheint als Nächstes ein Login-Fenster. Ansonsten sollte sich nach einer Weile die Startseite der Administration Console zeigen (siehe Abbildung 2.13). Auf der linken Seite des Fensters sehen Sie nun eine vertikale Menüleiste, über die Sie die verschiedenen Konfigurationsmöglichkeiten des GlassFish-Servers auswählen können.

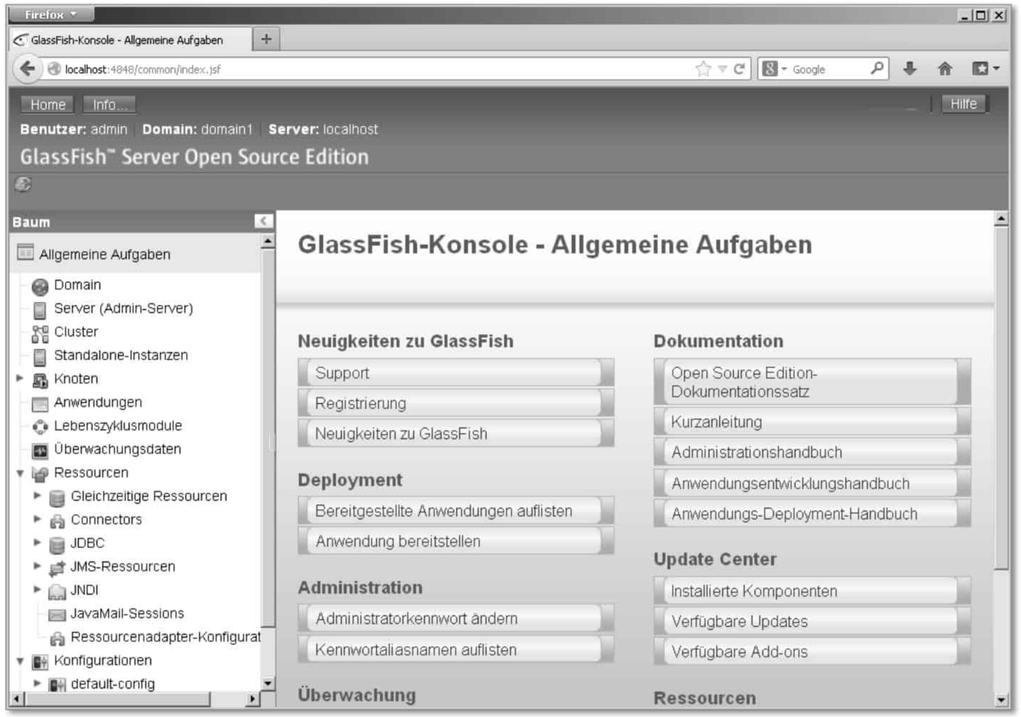


Abbildung 2.13 Die Startseite der Administration Console

Beispielsweise können Sie nun mithilfe des DAS die Logging-Levels setzen. Zu diesem Zweck klicken Sie auf der linken Seite des DAS auf KONFIGURATIONEN • SERVER-CONFIG • LOGGER-EINSTELLUNGEN. Auf der rechten Seite öffnen Sie den Reiter LOGGING-EBENEN (siehe Abbildung 2.14).

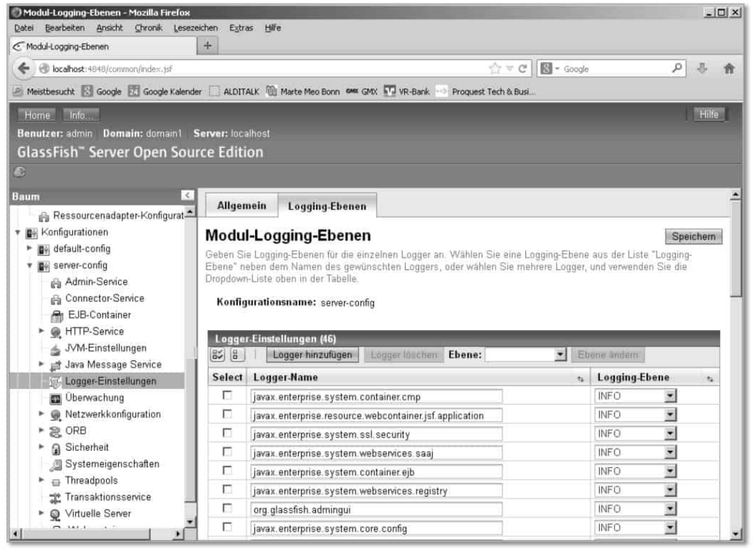


Abbildung 2.14 Die Einstellung des Loggers für die einzelnen Logging-Ebenen im DAS

Das Administratorkennwort setzen bzw. ändern

Um Einstellungen (wie beispielsweise das Passwort des Administrators) zu ändern, klicken Sie links auf DOMAIN (siehe Abbildung 2.15). Auf der rechten Seite erscheint eine neue Ansicht. Dort klicken Sie auf ADMINISTRATORKENNWORT.

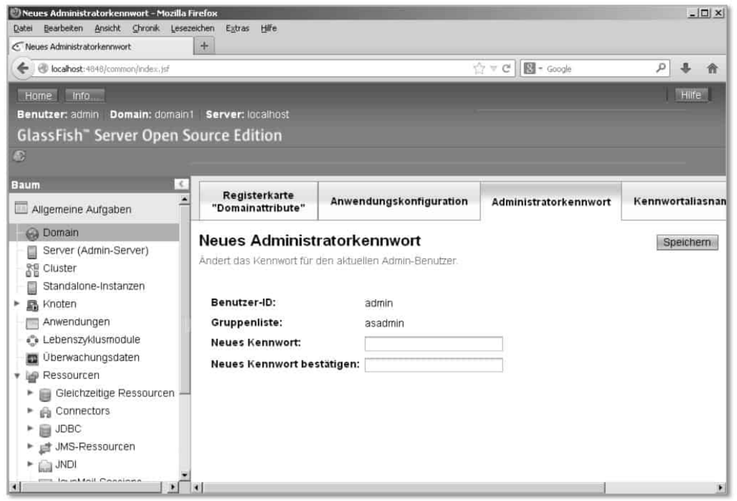


Abbildung 2.15 Der Menüpunkt »Domäne«

Der Heap der Java-Laufzeitumgebung

Sie können eine Leistungsverbesserung bei einem Java EE Server erzielen, indem Sie den Heap der JVM anpassen. Selbstverständlich ist die Größe des Heaps abhängig von dem zur Verfügung stehenden Rechner. Dennoch ist es hilfreich, wenn man gewisse Grundregeln beachtet.

Die wichtigsten Optionsparameter für die Konfiguration des Heap-Speichers sind `-Xmx`, `-Xms` und `-Xmn`.

`-Xmx` setzt die maximale Heap-Größe, `-Xms` setzt die minimale Heap-Größe und `-Xmn` zeigt an, wie viel des verwendeten Heaps für den sogenannten »eden«-Bereich belegt werden soll. In den »eden«-Bereich gelangen Objekte, wenn sie erstmalig erzeugt wurden. Dort schaut der Garbage Collector öfter vorbei und prüft, ob die Objekte noch gebraucht werden. Wenn der Garbage Collector nach einigen Versuchen merkt, dass ein Objekt längerfristig benötigt wird, überführt er es in andere Bereiche, die für ältere bzw. permanente Objekte reserviert sind.

Bei dem frisch installierten GlassFish-Server ist der maximal zur Verfügung stehende Heap-Speicher auf 512 MB(`-Xmx512m`) voreingestellt. Wenn man nun bei einer Minimalanforderung davon ausgehen würde, dass ein 32-Bit-Betriebssystem mit 4 GB vorliegt und dass das Betriebssystem Microsoft Windows 2 GB für eigene Tätigkeiten braucht, ist es angemessen, wenn wir mindestens 1 GB des gesamten Volumens nutzen.

Wenn Sie weniger Arbeitsspeicher als 4 GB Ihr Eigen nennen, trifft diese Bemessungsgrundlage für Sie nicht zu. Dessen ungeachtet könnten Sie die folgende Betrachtung auf Ihr System übertragen. Gehen wir zurück zu der Annahme, dass Sie einen Rechner mit einem 32-Bit-Betriebssystem und mit 4 GB RAM haben und dass Sie 1 GB für den GlassFish-Server opfern können.

Gemäß dieser Annahme setzen wir die maximale Heap-Size auf 1024 MB. Gleichzeitig werden wir die minimale Heap-Größe ebenso auf 1024 MB setzen, damit sich die Java-Laufzeitumgebung auf diesen Wert festlegen muss und keine Zeit damit verschwendet, den Heap-Speicher zu ändern. Für den »eden«-Bereich werden wir die Hälfte des gesamten Speichers einsetzen.

Klicken Sie im DAS auf der linken Seite auf KONFIGURATIONEN • SERVER-CONFIG • JVM-EINSTELLUNGEN (siehe Abbildung 2.16). Auf der rechten Seite klicken Sie auf den Reiter JVM-OPTIONEN. Dort ändern Sie den Wert des maximalen Heaps auf `-Xmx1024m` ab. Dann verwenden Sie den Button JVM-OPTION HINZUFÜGEN, um auch die Optionsparameter `-Xms1024m` und `-Xmn512m` hinzuzufügen.

Abbildung 2.16 zeigt die Optionsparameter im DAS. Klicken Sie oben rechts auf den Speicher-Button.

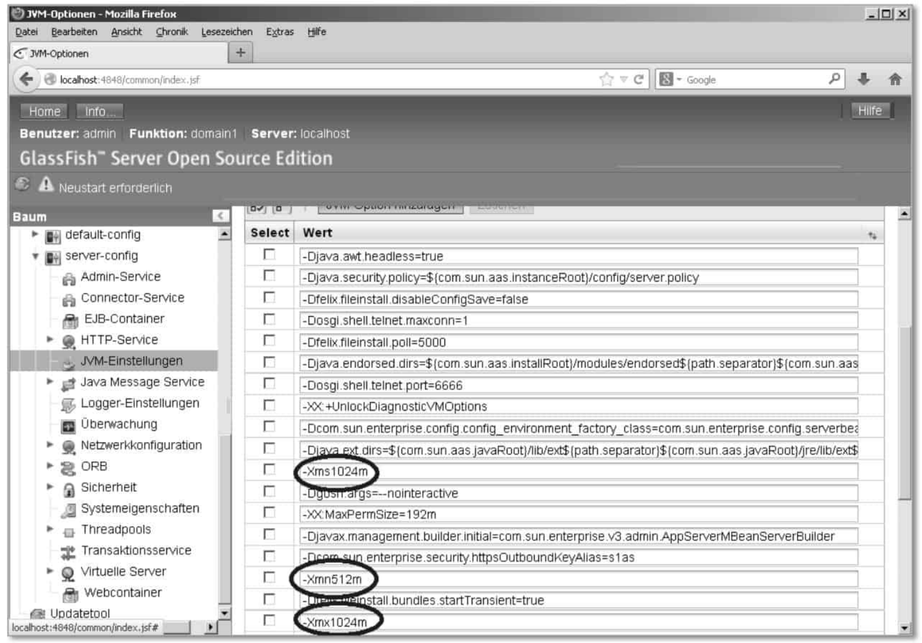


Abbildung 2.16 Der JVM Heap Space für den GlassFish-Server

Die Adressen der Listener anpassen

Eine weitere Konfiguration, die wir nun vornehmen werden, betrifft die IP-Adresse des sogenannten *Listeners*. Wir beginnen mit den HTTP-Listenern. Danach werden wir auch die IP-Adresse des IIOP-Listeners ändern. Zunächst wird kurz erläutert, warum die Listener des GlassFish-Servers angepasst werden.

Jedes Mal, wenn über das Netzwerk eine externe Anfrage eintrifft, »klopft« sie beim Server zuerst an eine bestimmte IP-Adresse und an einem bestimmten Port »an«. Damit der Java EE Server die ankommende Anfrage wahrnimmt, muss er wissen, auf welcher IP-Adresse und auf welchem Port er horchen soll. GlassFish nutzt hierbei nicht die im Netzwerk automatisch vergebene IP-Adresse, sondern eine im DAS konfigurierte IP-Adresse.

Nachdem der GlassFish-Server erstmalig installiert ist, lauscht er zunächst mit vier Listenern. Diese nennen sich `admin-listener`, `http-listener1`, `http-listener2` und `orb-listener` (für IIOP). Während der `admin`-Listener für Administrationsaufgaben gedacht ist, hat der Listener `http-listener1` die Aufgabe, normale HTTP-Anfragen mitzubekommen. Der `http-listener2` ist für die hochverschlüsselten SSL-Anfragen über das HTTPS-Protokoll gedacht. Der `orb`-Listener ist für externe IIOP-Verbindungen von EJB- oder CORBA-Aufrufen zuständig.

Wenn Sie auf KONFIGURATIONEN • SERVER-CONFIG • HTTP-SERVICE • HTTP-LISTENER klicken, erhalten Sie einen Überblick über die vorkonfigurierten HTTP-Listener.

Die HTTP-Listener von GlassFish sind per Default über die *Unspecified IPv4 Address* (bzw. auch *Pseudo IPv4 Adress*) `0.0.0.0` erreichbar. Die Pseudo-IPv4-Adresse `0.0.0.0` bedeutet in diesem Zusammenhang, dass lokale Aufrufe auf alle IP-Adressen entgegengenommen werden.

Hinweis

In einem IPv6-Netzwerk wäre das Pendant `0.0.0.0.0.0` bzw. `::`

Solange Sie sich lokal mit Ihrem GlassFish-Server verbinden, brauchen Sie theoretisch die Pseudo-Adresse nicht abzuändern. Wenn Sie jedoch einen GlassFish-Server remote zur Verfügung stellen, müssen Sie die Pseudo-IP-Adresse `0.0.0.0` zu der externen IP-Adresse korrigieren. Aber auch für den lokalen Zugriff ist die Verwendung der Pseudo-IP-Adresse `0.0.0.0` nicht empfehlenswert; Sie sollten die *Loopback-IP-Adresse* `127.0.0.1` eintragen.

Hinweis

Wenn Sie das neuere IPv6 in Ihrem Netzwerk einsetzen, handelt es sich bei dem `localhost` um die Loopback-Adresse `0.0.0.0.0.1` bzw. `::1`.

Um die IP-Adressen der drei HTTP-Listener abzuändern, klicken Sie im Fenster aus Abbildung 2.17 auf der linken Seite jeweils auf den einzelnen Listener, tragen auf der rechten Seite die IP-Adresse `127.0.0.1` ein und klicken jedes Mal oben rechts auf SPEICHERN (siehe Abbildung 2.17).

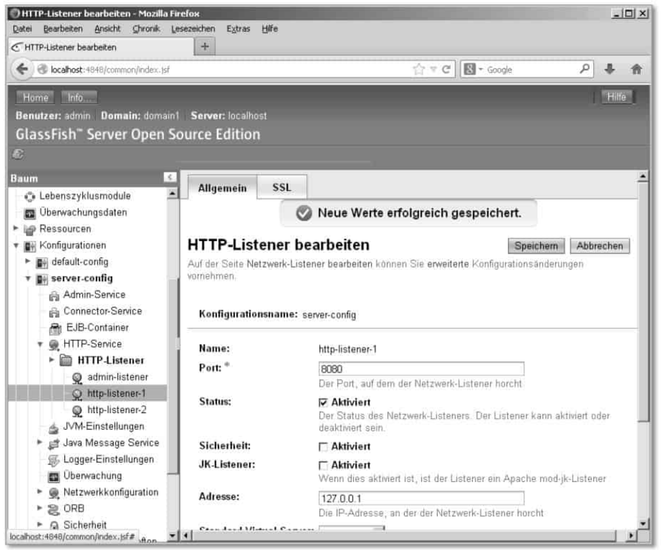


Abbildung 2.17 Die Änderung der IP-Adresse der HTTP-Listener

Als Nächstes kümmern wir uns um den IIOP-listener, der auf EJB-Clients und CORBA-Aufrufe wartet. Klicken Sie hierfür auf KONFIGURATIONEN • SERVER-CONFIG • ORB • ORB-LISTENER-1 und ändern Sie auch dort die Adresse auf die Loopback-Adresse `127.0.0.1` ab (siehe Abbildung 2.18). Klicken Sie abschließend auf SPEICHERN.

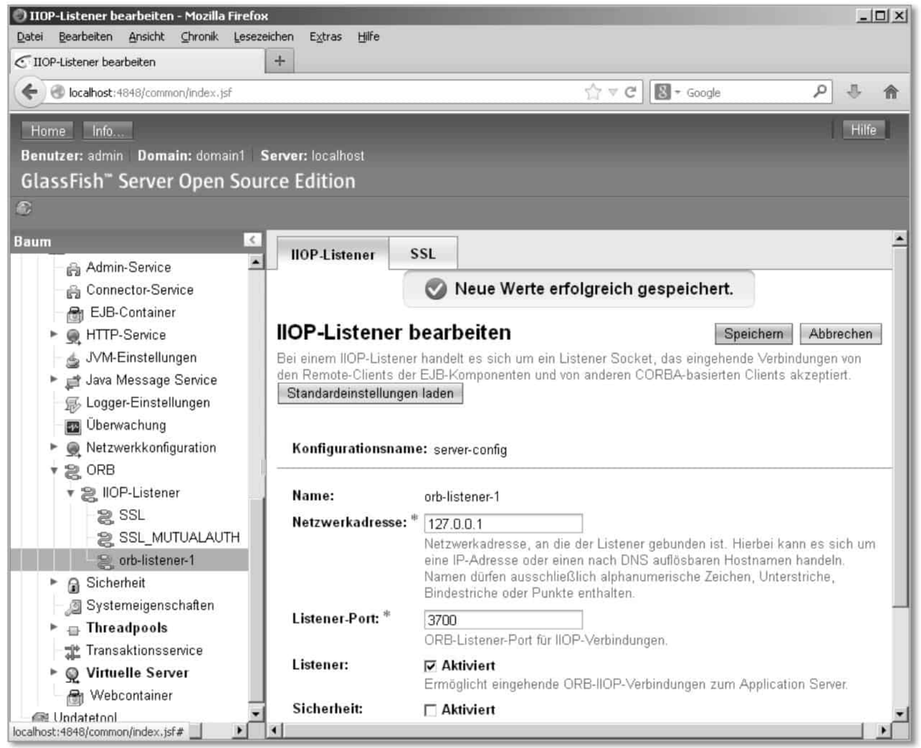


Abbildung 2.18 Die Anpassung der IP-Adresse für den IIOP-Listener

Erhöhung des HTTP-Thread-Pools

Grundsätzlich bietet Java die Möglichkeit, zahlreiche Prozesse in nebenherlaufenden Threads zuzulassen. Bei diesem scheinbar parallel ablaufenden Vorgang werden die einzelnen Prozesse in sehr kleine Zeitfragmente heruntergebrochen. Weil zwischen den »parallelen« Threads ständig hin- und hergeschaltet wird, können zeitlich langsamere Threads die anderen Prozessfragmente nicht blockieren. Dadurch wird die Gesamtgeschwindigkeit deutlich verbessert.

Zur automatischen Verwaltung dieser parallelen Threads wurden sogenannte Thread-Pools erfunden. Als Beispiel für einen Thread-Pool könnte Ihnen der *Connection Pool* von JDBC bekannt sein. Auch hier werden mehrere parallel laufende Threads in einem zentralen

Thread-Pool gehalten. Bei einem Thread-Pool wird zunächst eine minimale Anzahl an Threads vorbereitet, die zur unmittelbaren Verwendung vorliegen. Jeder Prozess kann sich nun eines dieser Threads bedienen. Wenn der Prozess durchlaufen ist, gibt er den Thread wieder frei, sodass der Thread anschließend für weitere Prozesse zur Verfügung steht. Wenn alle Threads des Thread-Pools in Verwendung sind, kann der Thread-Pool die Anzahl der Threads erhöhen, indem er weitere erzeugt. Dies kann er bis zu einer festgelegten maximalen Anzahl an Threads fortführen.

Auch ein Web-Container beinhaltet einen Thread-Pool, weil er mehrere parallel ankommende HTTP-Anfragen nicht aufeinander warten lassen möchte. Bei einem frisch installierten GlassFish-Server können maximal fünf parallel zugreifende HTTP-Anfragen beantwortet werden. Diese Obergrenze ist aber für unsere Zwecke nicht ausreichend, denn eine vollwertige Java-EE-Anwendung stößt hiermit schnell auf ihre Grenzen. Um die Leistung des Web-Containers zu verbessern, werden wir die maximale Anzahl der im Thread-Pool zur Verfügung stehenden Threads erhöhen.

Klicken Sie hierfür im Fenster aus Abbildung 2.19 auf der linken Seite auf KONFIGURATIONEN • SERVER-CONFIG • THREAD POOLS • HTTP-THREAD-POOL, und setzen Sie auf der rechten Seite die MAX. THREADPOOLGRÖSSE und die MIN. THREADPOOLGRÖSSE jeweils auf 200 Threads. Auch hiernach sollte der GlassFish-Server neu gestartet werden.

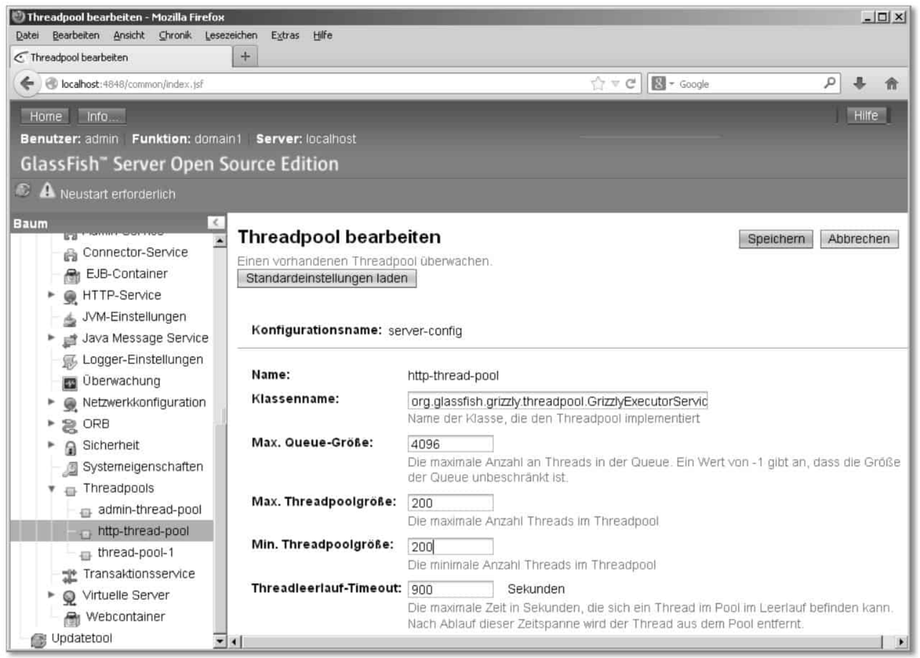


Abbildung 2.19 Das Setzen der maximalen und der minmalen Threadpoolgröße

JMX konfigurieren

Um das Wohlergehen einer Java-EE-Anwendung überwachen zu können, legt die Java-EE-Spezifikation fest, dass ein Java EE Server über eine besondere API mit dem Namen *Java Management Extension* (JMX) verfügen muss.

Intern arbeitet JMX mit sogenannten *Managed Beans* (MBeans). Eine MBean ist eine spezielle Java-Klasse, die beispielsweise Benachrichtigungen oder Warnungen erledigt. Um seine MBeans zu verwalten, stellt der GlassFish-Server den sogenannten *JMX Connector Server* bereit.

Die IP-Adresse und den Port, über die der JMX-Dienst erreichbar ist, können Sie über ALLGEMEINE AUFGABEN • KONFIGURATIONEN • SERVER-CONFIG • ADMIN SERVICE abändern (siehe Abbildung 2.20).

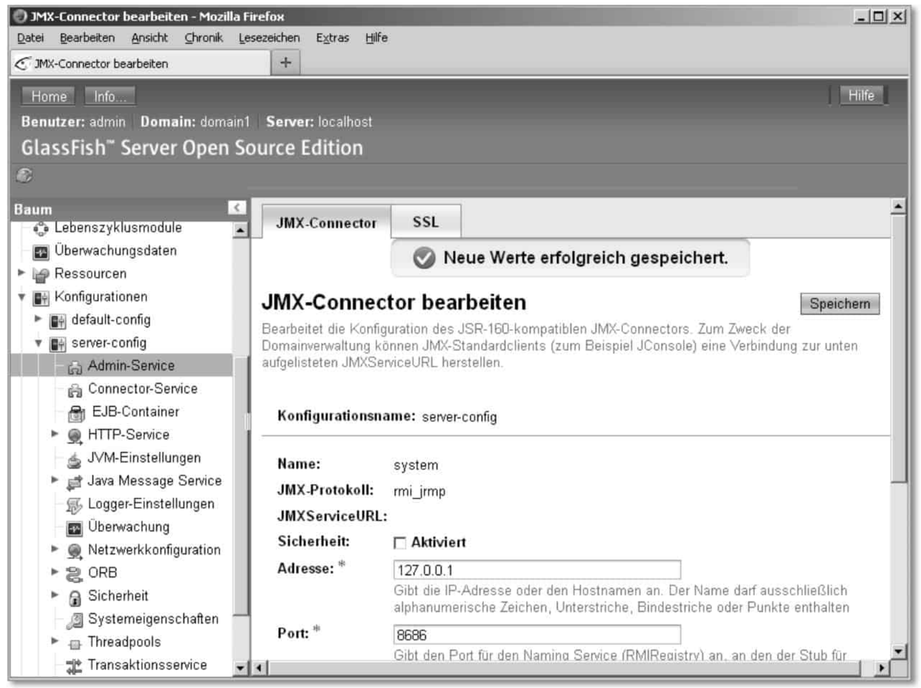


Abbildung 2.20 Die »JMXServiceURL« beim GlassFish-Server

Für den JMX Connector Server von GlassFish ist per Default die *Unspecified IPv4 Address* 0.0.0.0 und der Port 8686 erreichbar. Setzen Sie die IPv4-Adresse auf die Loopback-IP-Adresse `127.0.0.1`.

Hinweis

Wenn Sie IPv6-Adressen in Ihrem Netzwerk verwenden, tragen Sie `0.0.0.0.0.1` in das Feld ein.

Um die Änderungen restlos wirksam werden zu lassen, starten Sie den Server neu.

Das Monitoring über JConsole

Wir werden uns nun die Informationen anschauen, die uns der JMX Connector Server über seine registrierten MBeans meldet. Das gängigste Werkzeug, über das man sich mit dem JMX Connector Server von GlassFish verbinden kann, bietet die JDK. Es nennt sich *JConsole* (*Java Monitoring and Management Console*).

Sie können JConsole beispielsweise über die Eingabeaufforderung aufrufen. Es erscheint ein Fenster, in dem Sie die Verbindungsdaten des JMX Connector Servers von GlassFish eingeben können (siehe Abbildung 2.21).



Abbildung 2.21 JConsole

Klicken Sie zunächst auf REMOTE PROCESS, denn über LOCAL PROCESS werden die MBeans nicht automatisch gestartet. Geben Sie dann im Eingabefeld die IP-Adresse des Rechners und die Portnummer des JMX Connector Servers ein. Auf meinem Rechner ist das `127.0.0.1:8686`. Wenn Sie anschließend auf den Button CONNECT klicken, öffnet sich ein

Fenster mit mehreren Reitern (siehe Abbildung 2.22). Gleich beim ersten Reiter zeigt sich JConsole von seiner besten Seite, da wir in einer grafischen Übersicht den Verbrauch von Speicher- und Prozessorressourcen erkennen können.

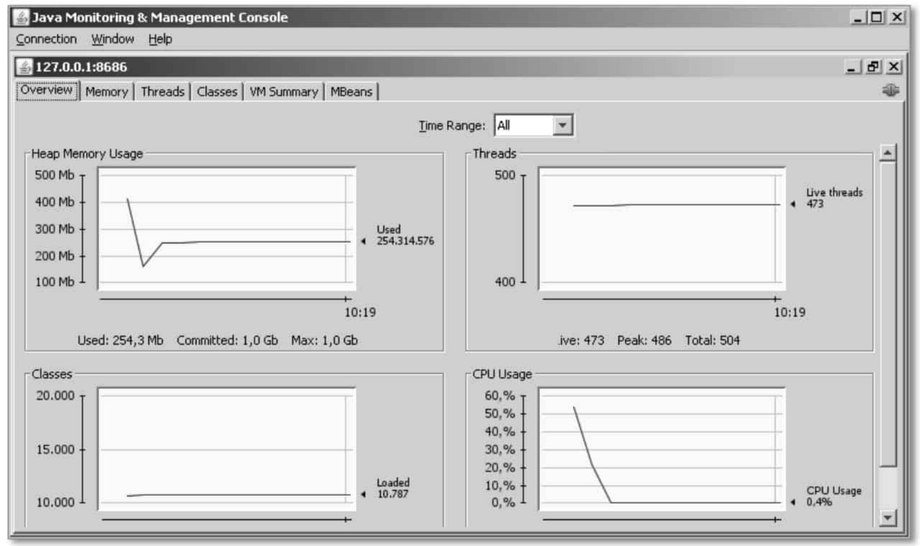


Abbildung 2.22 Das Monitoring des Java EE Servers über JConsole

Die Zeichenenkodierung auf UTF-8 umstellen

Ich gehe später noch auf die Internationalisierung von Texten und die Bedeutung der Zeichenkodierung von UTF-8 ein. Im Extra-Abschnitt über UTF-8 finden Sie bei Interesse wichtige Informationen zu diesem Thema.

Wenn GlassFish mit dem Webbrowser kommuniziert, verwendet er standardmäßig die Enkodierung ISO-8859-1. Weil für internationalisierte Webanwendungen eigentlich nur noch die Zeichenenkodierung UTF-8 verwendet werden sollte, werden wir diese Default-Einstellung abändern. Wie Sie später noch sehen werden, müssen wir diese Änderung noch an weiteren Stellen vorsehen. Aber darum kümmern wir uns erst in späteren Kapiteln.

Um den GlassFish-Server auf die Zeichenenkodierung UTF-8 umzustellen, setzen Sie noch den entsprechenden Wert in die JVM-Optionen. Klicken Sie hierfür im DAS wieder auf der linken Seite auf KONFIGURATIONEN • SERVER-CONFIG • JVM-EINSTELLUNGEN. Auf der rechten Seite klicken Sie auf den Reiter JVM-OPTIONEN. Dort klicken Sie auf den Button JVM-OPTION HINZUFÜGEN und fügen den Optionsparameter `-Dfile.encoding=UTF8` hinzu.

Im Anschluss folgt nun ein kurzer Extra-Abschnitt, in dem die wichtigsten Zeichenenkodierungen – insbesondere UTF-8 – erläutert werden.

2.5 Extra-Abschnitt: UTF-8

Wenn man eine Internetanwendung programmiert, entsteht eine Anwendung, die global (d.h. über Landes- und Sprachgrenzen hinaus) aufrufbar ist. Dabei tauchen häufig auch Probleme auf, denn der Webbrowser erhält den Inhalt einer Webseite in einer Abfolge von Bytes, und die kann je nach Sprache und Zeichenenkodierung sehr divergent ausfallen.

Um hierfür eine solide Lösung zu finden, erfand man die Zeichenenkodierung UTF-8. Heutzutage wird man jede Webanwendung nur noch mit UTF-8 erstellen, denn nur so ist man international und multilingual gewappnet. Im Folgenden wird gezeigt, weshalb UTF-8 für Internetseiten so bedeutend ist und wie diese wichtige Zeichenenkodierung in einer Webseite festgelegt wird. Zunächst schauen wir uns die gängigsten Zeichenenkodierungen an. Denn dadurch wird klar, wieso es nötig war, UTF-8 zu erfinden.

2.5.1 ANSI

Die Abkürzung ANSI steht für das *American National Standards Institute*. Der ANSI-Code wird Ihnen auch unter anderen Namen, wie zum Beispiel Windows-1252, begegnet sein. Sowohl ANSI wie auch Windows-1252 werden aber nur fälschlicherweise als Standards bezeichnet. ANSI basiert auf den (in diesem Abschnitt später gezeigten) Zeichensatz ISO 8859-1, der tatsächlich als Standard normiert wurde.

2.5.2 ASCII

Der ehemals wichtigste Zeichenkodierungsstandard in der Informationstechnologie war der *American Standard Code for Information Interchange* (ASCII). Er wurde in den 60er-Jahren veröffentlicht. Man verwendete eine Abfolge von 7 Bits. Durch unterschiedliche 7-Bit-Kombinationen können theoretisch bis zu 128 verschiedenartige Symbole dargestellt werden. Man spricht auch von 128 Codepoints. Ein *Codepoint* stellt eine Position in der Kodierungstabelle dar (siehe Abbildung 2.23).

Buchstabe	Die 7 Bits des Codepoints						
a	1	1	0	0	0	0	1
b	1	1	0	0	0	1	0
c	1	1	0	0	0	1	1

Abbildung 2.23 ASCII bietet 128 unterschiedliche Codepoints.

Die 128 Codepoints waren damals vollkommen ausreichend, da die 26 englischen Kleinbuchstaben, die jeweiligen Großbuchstaben, die Sonderzeichen und die Steuerzeichen der amerikanischen Tastatur damit abgedeckt werden konnten.

2.5.3 ISO 8859-1

In den 80er-Jahren verbreiteten sich IBM-kompatible Computer auf der ganzen Welt so sehr, dass schlagartig auch anderssprachige Buchstaben abgebildet werden mussten. Da ASCII unterdessen bereits massenhaft eingesetzt wurde, suchte man nach Zeichensätzen, die einerseits in den ersten 128 Zeichen deckungsgleich mit dem eingefahrenen Standard sind, andererseits diesen aber auch länderspezifisch erweitern. Dienlich war dabei, dass damalige IBM-Modelle intern mit 8 Bits bzw. einem Byte rechneten. Da eine auf 8 Bits basierende Kodierung 256 Codepoints ermöglicht, standen die verbliebenen 128 Codepoints für landeseigene Symbole zur Verfügung. Aus diesem Grund wurden zahlreiche Kodierungstabellen mit 256 Codepoints entwickelt, die man *Codepages* nannte. Die wichtigsten Codepages wurden von der ISO normiert. Die heute noch gängigste Norm aus dieser Zeit nennt sich *ISO 8859-1*. Ein anderer Begriff für diese Norm ist auch *latin1*.

Dass unterschiedliche Codepages für die Darstellung verschiedener Sprachzeichen erforderlich sind, birgt allerdings gravierende Nachteile. Abgesehen von der Problematik, dass die Zeichen unterschiedlicher Normen nicht im gleichen Dokument enthalten sein können, wurde eine Jahrtausende alte Tradition aus Ostasien zur Nuss, die es zu knacken galt. Die Urform der ostasiatischen Zeichen stammt nämlich aus China und umfasst weit mehr als 100.000 Symbole. Diese gigantische Menge kann überdies von ihrer Natur her sogar noch erweitert werden.

2.5.4 Unicode

Zunächst versuchte man mit zwei Bytes, also mit einem 16 Bit langen Unicode-Zeichensatz auszukommen. Die nun zur Verfügung stehenden 65.536 Zeichen sollten für alle gängigen Zeichen ausreichen, die in den 80er-Jahren weltweit in der Wirtschaft und der Industrie verbreitet waren. Trotz des exorbitanten Umfangs der chinesischen Zeichen war der Grundgedanke realistisch, denn die ostasiatischen Staaten – allen voran China selbst – bemühten sich bereits seit Jahrzehnten um Reduktion. So kam es dazu, dass Unicode im Jahre 1991 jubelnd veröffentlicht und von Technologien wie zum Beispiel Java oder Windows als internes Format eingesetzt wurde. Die Zeichen von Unicode stimmen nicht nur mit dem ASCII-Zeichensatz überein, sondern sind in den ersten 256 Codepoints auch noch kompatibel mit dem Zeichensatz ISO 8859-1. Leider trübten sich die Aussichten für Unicode, denn er wurde klar, dass man auch die fortgelassenen Symbole brauchte. Gleichzeitig wurden immer mehr Sprachen entdeckt.

2.5.5 UTF-16

Um letztendlich jedes erdenkliche Zeichen darzustellen, wurde UTF-16 entwickelt. UTF-16 enthält mehrere 16-Bit-Ebenen, die *Planes* genannt werden. Genau genommen handelt es sich um 17 Planes. Dies ermöglicht insgesamt 1.114.112 Kombinationen, von denen bis dato bereits mehr als 110.000 belegt wurden. Um zum ersten Unicode-Entwurf kompatibel zu sein, entspricht die erste *Plane* dem älteren Unicode-Entwurf. Aufgrund seiner Bedeutung wird die erste Plane als *Basic Multilingual Plane*(BMP) bezeichnet. Während sich BMP aus zwei Bytes zusammensetzt, nehmen die weiteren Planes vier Bytes ein. Darüber hinaus wird auch noch mit *Big Endian* und *Little Endian* unterschieden, in welcher Reihenfolge die Bytes angelegt werden. Bei Little Endian befinden sich die kleineren Bytes vorne. Die Byte-Reihenfolge erkennt der Webbrowser an der *Byte Order Mark* (BOM). Die BOM ist eine Zeichenfolge, die gleich am Anfang einer Textdatei gesetzt wird. In hexadezimaler Darstellung deutet FF FE auf ein Big Endian und FE FF auf ein Little Endian hin.

UTF-16 konnte sich für die Nutzung im Web jedoch nicht durchsetzen. Laut einer Statistik des W3C verwenden weltweit weniger als 0,01 % der Webseiten UTF-16.

2.5.6 UTF-8

UTF-8 wurde mit dem RFC-3629 durch die ISO-10646 normiert.

Egal welche Normungsinstitute oder Expertengruppen man auch fragt, einhellig ist man der Meinung, dass UTF-8 durchgängig für Webseiten und Webanwendungen eingesetzt werden sollte. Laut W3C sind schon mehr als die Hälfte aller Webseiten auf die Zeichenkodierung UTF-8 umgestellt worden, die etwa zum gleichen Zeitraum wie UTF-16 von Ken Thompson erfunden wurde.

Die Grundidee von UTF-8 ist, die Anzahl der für die Kodierung zur Verfügung stehenden Bytes nicht auf eine bestimmte Menge einzuschränken. Stattdessen sollen die Kodierungen beliebig viele Bytes lang sein können. Demzufolge kann eine UTF-8-Kodierung ein Byte, zwei Bytes, drei Bytes oder länger sein. Mit dem neuen Kodierungswunder könnten somit beliebig viele Codepoints zur Verfügung gestellt werden. Allerdings musste auch noch eine gute Rückwärtskompatibilität her. Aber Ken Thompson lieferte auch hierfür eine Lösung (siehe Abbildung 2.24).

Das Kodierungsmuster von UTF-8 unterscheidet zwischen dem *Startbyte* (d.h. dem ersten Byte) und den *Folgebytes*. Die einen Byte langen UTF-8-Codes bestehen nur aus dem Startbyte und werden am ersten Bit erkannt, denn das muss bei 1 Byte langen Codes immer eine 0 sein. Dadurch bleiben 128 Kombinationsmöglichkeiten, die mit den ASCII-Codes deckungsgleich sind. Alle Kodierungen, die länger als ein Byte sind, also 2 Bytes, 3 Bytes usw., deuten mit beginnenden Einsen des ersten Startbytes die Anzahl der gesamten Bytes an. Folgebytes beginnen immer mit einer 1 und einer 0. Diesem logischen Gerüst fallen

zwar einige Bit-Positionen zum Opfer, denn die braucht das Kodierungssystem ja für seine interne Logik. Aber allein schon mit dieser Methode wäre eine 8 Byte lange Kodierung und damit eine Zuordnung von 4.398.046.511.104 Codepoints möglich. Im Jahre 2003 wurde UTF-8 durch das RFC 3629 auf vier Bytes begrenzt und mit den UTF-16-Codepoints in Übereinstimmung gebracht.

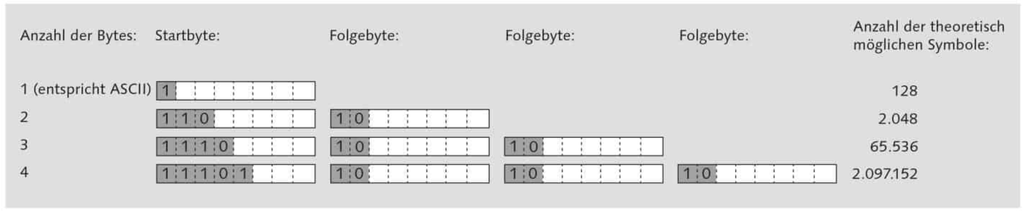


Abbildung 2.24 Dem logischen Gerüst von UTF-8 fallen die in der Darstellung dunkel markierten Bit-Positionen zum Opfer.

Die in Abbildung 2.24 markierten Positionen braucht das Kodierungssystem für seine interne Logik. Die Anzahl der theoretisch möglichen Symbole errechnet sich durch die in den weißen Zellen kombinierbaren Bitfolgen, d.h.: 2^7, 2^11, 2^16 und 2^21.

Die Vormacht von UTF-8 liegt nicht nur an der theoretisch unbegrenzten Anzahl an Codepoints. Vielmehr sorgte die besondere Art der Rückwärtskompatibilität dafür, dass sich UTF-8 bislang als wichtigster Kodierungsstandard im Internet durchsetzen konnte. UTF-8 ist nämlich sowohl in den Positionen der Codepoints wie auch in der Abfolge der acht Bits je Zeichen deckungsgleich zu ASCII.

Bei UTF-8 kann zur Erkennung – genau wie bei UTF16BE und UTF-16LE – ebenfalls ein BOM (`EF BB BF`) am Anfang des Textes stehen, wobei hier nur die Reihenfolge »große Bytes zuerst« definiert ist.

2.5.7 Eine HTML-Seite mit UTF-8-Merkmalen erzeugen

Wenn ein Webbrowser eine Webseite empfängt, versucht er die Enkodierung anhand unterschiedlicher Merkmale zu interpretieren. Weil sich diese Angaben widersprechen könnten, hat das W3C eine Reihenfolge festgelegt, nach der ein Webbrowser priorisieren soll. Die Reihenfolge ist im Folgenden dargestellt.

Priorität 1: Der Content-Type im HTTP-Header

Die höchste Priorisierung wird dem Content-Type im HTTP-Header eingeräumt:

```
Content-Type: text/html;charset=UTF-8
```


Dieser Eintrag ist sehr wichtig, denn wenn ein Webbrowser ein Textdokument einliest, erhält er zunächst nur Einsen und Nullen, die er vorgehend nur als Bits bzw. Bytes versteht. Weil der Content des Dokuments ganz unterschiedlich enkodiert sein kann, muss dem Webbrowser rechtzeitig gezeigt werden, wie er die Bytes zu deuten hat. Weil das Einlesen der Reihe nach von oben nach unten erfolgt und der Content-Type im HTTP-Header vor dem Dokument ankommt, erfährt der Webbrowser vorab, wie der Content zu interpretieren ist. Der Content-Type bedient sich hierbei eines Standards, der in einem ganz anderen Zusammenhang entwickelt wurde, nämlich für den E-Mail-Verkehr. Der Standard nennt sich *Multipurpose Internet Mail Extension* (MIME).

Priorität 2: Das BOM am Anfang der Textdatei

Wurde der Content-Type im HTTP-Header nicht gesetzt, untersucht der Webbrowser den Beginn des HTTP-Bodys, denn hier könnte der Texteditor ein BOM angefügt haben.

Priorität 3: Das META-Tag »http-equiv«

Das META-Element hat die geringste Wertigkeit. Wenn weder der Content-Type eine Auskunft gibt und auch kein BOM gefunden wurde, sucht der Webbrowser nach META-Tags mit dem Attribut `http-equiv`.

```
<meta http-equiv="Content-Type"
    content="text/html; charset=UTF-8"/>
```

Listing 2.1 Das META-Tag »http-equiv«

Im Übrigen wurde für *HTML5* die Syntax etwas minimalisiert, denn dass es sich bei einem HTML-META-Tag um einen HTML-Quelltext handelt, ist eine unnötige Information. Was bleibt, ist ein kurzes Tag, das sich auf den Zeichensatz bezieht:

```
<meta charset="UTF-8"/>
```

Listing 2.2 Das META-Tag »charset="UTF-8"« gilt nur bei HTML5.

In der Regel werden alle drei Merkmale gleichzeitig gesetzt.

2.6 Eine Java-EE-Anwendung deployen

Nachdem Sie den GlassFish-Server installiert und konfiguriert haben, ist es an der Zeit, eine erste Beispielanwendung in der Art eines »Hallo Welt«-Programms zu erstellen und diese zu installieren. Zu früh?

Keineswegs, denn für eine einfache Java-EE- bzw. Webanwendung, die lediglich eine Zeichenkette ausgibt, benötigen Sie kaum weitere Kenntnisse als jene, die Sie als Java-Programmierer ohnehin schon mitbringen. Sobald Sie die Beispielanwendung fertiggestellt haben, kann sie in dem GlassFish-Server bereitgestellt werden. Der englische Fachausdruck für die Bereitstellung ist *Deployment*.

2.6.1 Der Aufbau einer Java-EE-Anwendung

Java-EE-Anwendungen sind hierarchisch verschachtelt, denn zusammengehörende Java-Klassen werden zu Modulen und mehrere Module wiederum zu einer Gesamtdatei zusammengefasst. In diesem Abschnitt werden wir uns diesen Aufbau Schritt für Schritt genau anschauen, da er für die Installation von zentraler Bedeutung ist.

2.6.2 Java-EE-Module

Damit die Web-Komponenten und die EJB-Komponenten in den Containern ausgeführt werden können, müssen sie auf dem Server installiert werden. Die Installation findet statt, indem die Java-EE-Anwendung als gezippte Datei auf den Java EE Server übertragen wird. Der Fachausdruck für die Installation ist *Deployment*.

Das Deployment beginnt, indem logisch zusammenhängende Komponenten zu Modulen zusammengefasst werden. Module werden Ihnen bereits aus Ihrer Java-Praxis bekannt sein, denn auch Java-Standalone-Komponenten und Java-Applets werden mithilfe des `jar`-Werkzeugs zu einer Java-Archivdatei gezippt und mit der Endung *.jar* abgespeichert. In der Java-EE-Fachsprache werden diese beiden *.jar*-Dateien als *Module* bezeichnet. Genauso wie clientseitige Java- und Applet-Dateien werden auch serverseitige Komponenten zu Modulen gepackt (siehe Abbildung 2.25).



Abbildung 2.25 EJB-Komponente in EJB-Modul

Web-Komponenten werden zu Web-Modulen und EJB-Komponenten zu EJB-Modulen paketiert. Allerdings haben Web-Module ein besonderes Merkmal, denn sie erhalten nicht die Endung *.jar*, sondern die Endung *.war*. *war* steht für den englischen Begriff *Web Archive*.

Leider wurde für die EJB-Module keine eigene Endung vorgesehen. Um die Java-EE-Module noch deutlicher zu kennzeichnen, wird deshalb manchmal zusätzlich ein »-ejb« bzw. beim Web-Modul ein »-war« zum Dateinamen hinzugefügt (siehe Abbildung 2.26).



Abbildung 2.26 »war«-Komponente in einem »war«-Modul

Eine weitere Besonderheit von Web-Modulen ist, dass ihre Komponenten nicht im obersten Ordner ihrer Verzeichnisstruktur gesetzt werden, denn dieser Bereich ist statischen Inhalten vorbehalten. Mit statischen Inhalten sind Dateien – wie zum Beispiel HTML-, Bilder, CSS- oder JavaScript-Dateien – gemeint. Es handelt sich also um Dateien, die vom Webbrowser interpretiert werden können. Alle Dateien, die nicht geradeswegs zum Webbrowser geschickt, sondern vielmehr noch vor dem direkten Zugriff geschützt werden sollen, gehören hier nicht hin, sondern werden in einem sicheren Unterordner namens *\WEB-INF* abgelegt (siehe Abbildung 2.27).

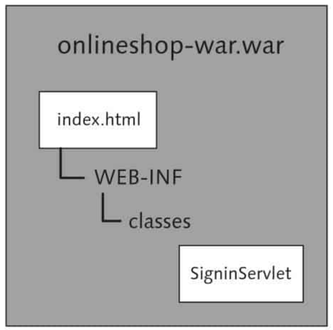


Abbildung 2.27 Die Baumstruktur der »war«-Datei

Eine Ausnahme hierbei bilden *Facelets* und *JSPs*. Die Spezifikation legt fest, dass Facelets und JSPs ebenfalls zu den statischen Inhalten hinzuzufügen sind. Der Grund hierfür ist Folgender: Ursprünglich bezweckte man, dass Facelets und JSPs vom Web-Designer gewartet werden. Um an dieser Stelle eine klare Trennung vorzunehmen, wurden Facelets und JSPs ausnahmsweise dem statischen Bereich zugeordnet.

Dagegen werden Servlets und auch andere Java-Klassen des Frontends in einem Unterverzeichnis mit dem Namen *\WEB-INF\classes* abgelegt. Dabei müssen sie in Packages geordnet sein. Das bedeutet mit anderen Worten, dass eine Web-Komponente zwingend ein Package angeben muss. Das Default-Package ist also nicht zulässig.

Bei der Ausführung des Web-Containers werden alle Packages unterhalb von *\WEB-INF\classes* automatisch in den `CLASSPATH` gesetzt. Oft werden in einer Webanwendung auch weitere Java-Bibliotheken (wie zum Beispiel Treiber) benötigt. Solche Java-Bibliotheken werden im Verzeichnis *\WEB-INF\lib* gespeichert. Alle *.jar*-Dateien, die in dem Ordner *\WEB-INF\lib* abgelegt sind, werden vom Web-Container automatisch in den `CLASSPATH` eingebunden. In unserem Übungsbeispiel benötigen wir aber keine Java-Bibliotheken, deshalb brauchen wir dort das Verzeichnis *\WEB-INF\lib* auch nicht zu erstellen.

Beachten Sie jedoch, dass das Verzeichnis *\WEB-INF* zwingend erforderlich ist, auch wenn das Modul keine Servlets und auch keine Java-Bibliotheken beherbergt.

2.6.3 Die Enterprise-Archiv-Datei

Weiter oben wurde dargelegt, dass manche Java-EE-Anwendungen manchmal nur einen Web-Container benötigen, weil sie lediglich Web-Komponenten enthalten. Solche Web-Module können auch ohne *ear*-Schale deployt werden. Dabei spricht man auch von »einfachen Webanwendungen«. In der Java-EE-Spezifikation wurde hierfür der Begriff *Webapplication* bzw. *Webapp* definiert. Auch beim GlassFish-Server kann ein Web-Modul ohne »ear-Schale« deployt werden. Dies entspricht jedoch nicht der Standardform. Denn normalerweise besteht eine Java-EE-Anwendung aus einem Java-Enterprise-Archiv, das in seinem Inneren alle Module beherbergt, die zu der Java-Enterprise-Anwendung gehören.

Ein Java-Enterprise-Archiv ist ein mit dem `jar`-Tool paketiertes Verzeichnis. Damit die Datei auch als Java-EE-Anwendung erkannt wird, erhält sie die Endung *.ear*. »ear« ist die Abkürzung für *Enterprise-Archiv*. Deshalb bezeichnet man Enterprise-Archiv-Dateien manchmal auch einfach nur als *.ear*-Datei. Eine *.ear* -Datei enthält ein oder mehrere Module. In Abbildung 2.28 sehen Sie ein Java-Enterprise-Archiv mit zwei Modulen.

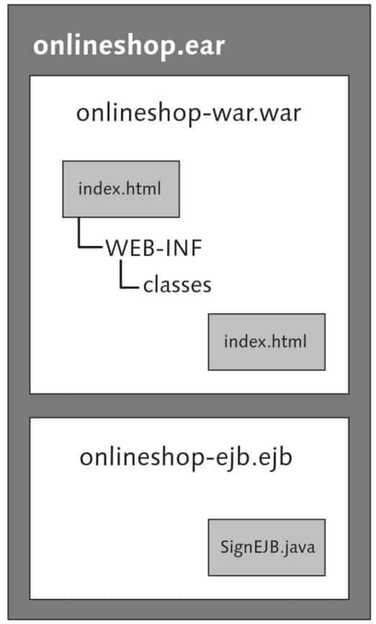


Abbildung 2.28 Die ».ear«-Datei

2.6.4 Die Erstellung der Web-Komponente (mit UTF-8)

Bisher wurde die hierarchische Verschachtelung von Komponenten, Modulen und Applikation gezeigt. Nun gehen wir in die Praxis über und erstellen dieses Konstrukt in Form einer konkreten *.ear*-Datei.

Zunächst erstellen Sie ein Arbeitsverzeichnis mit dem Namen *C:\workspace*. Unterhalb des Arbeitsverzeichnisses setzen Sie ein Projektverzeichnis, beispielsweise *C:\workspace\Onlineshop*. Darunter kommt erst das Verzeichnis für die Webinhalte. Ich nenne es *C:\workspace\Onlineshop\WebContent*, weil ich das durch die Arbeit mit Eclipse so gewohnt bin. Dieses Verzeichnis wird später der oberste Ordner des Web-Moduls sein. Legen Sie noch das Verzeichnis *C:\workspace\Onlineshop\WebContent\WEB-INF* an. Erstellen Sie nun folgenden Quelltext in einer Textdatei:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8"/>
        <title>Onlineshop</title>
    </head>
    <body>
```

```
        <h1>Onlineshop</h1>
    </body>
</html>
```

Listing 2.3 »index.html«

Legen Sie Datei in das Verzeichnis *C:\workspace\Onlineshop\WebContent* ab. Beachten Sie hierbei, dass die Datei mit der Zeichenenkodierung UTF-8 abzuspeichern ist. Wenn Sie beispielsweise das Dokument *index.html* mit dem Texteditor von Microsoft Windows anlegen, wird es normalerweise automatisch mit dem Microsoft-typischen Zeichensatz ANSI voreingestellt. Über das Menü SPEICHERN UNTER können Sie aber auch andere Zeichenkodierungen auswählen. Außer ANSI bietet der Texteditor von Windows zum Beispiel auch noch UNICODE, UNICODE BIG ENDIAN und UTF-8 an (siehe Abbildung 2.29).

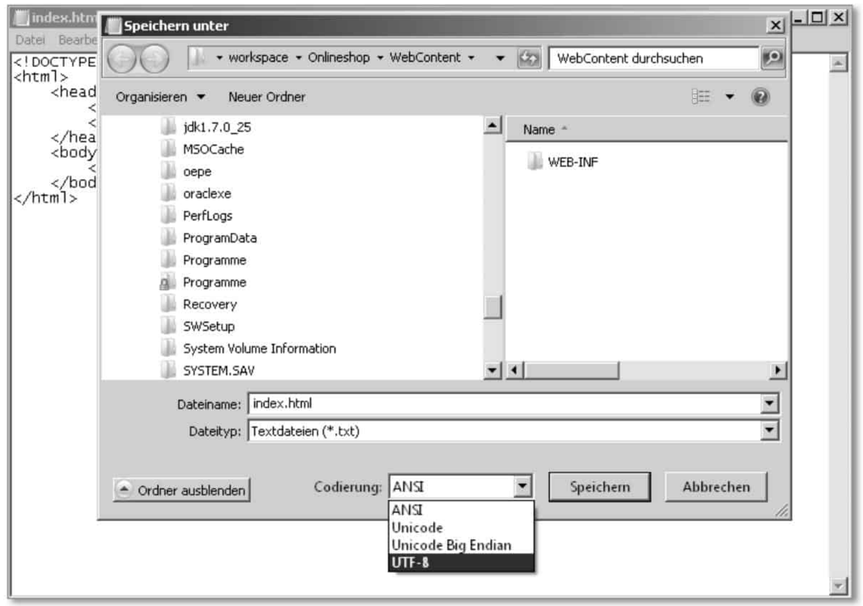


Abbildung 2.29 Die Einstellung der Zeichenkodierung im Texteditor

Für den Webentwickler spielt allerdings nur noch UTF-8 eine entscheidende Rolle.

Hinweis

Warum man der Zeichenenkodierung so viel Wert beimisst, wurde in Abschnitt 2.5, »Extra-Abschnitt: UTF-8«, deutlich. Dort wurde die Bedeutung von Zeichenenkodierungen und insbesondere von UTF-8 erklärt.

2.6.5 Die Erzeugung des Web-Moduls und der ».ear«-Datei

Nun werden wir das Web-Modul erzeugen. Nennen Sie das Modul *onlineshop-war.war*. Die Konvention von Java EE besagt, dass der Name der Webanwendung automatisch durch den Namen des Web-Moduls festgelegt ist. Das Web-Modul wird deshalb den Namen *onlineshop-war* tragen. Das Web-Modul *onlineshop-war.war* entsteht, indem der Inhalt von *C:\workspace\Onlineshop\WebContent* mit dem `jar`-Tool gezippt wird. Gleich darauf erzeugen Sie die Datei *onlineshop.ear*, indem Sie die Datei *onlineshop-war.war* mit dem `jar`-Tool zippen (siehe Abbildung 2.30).

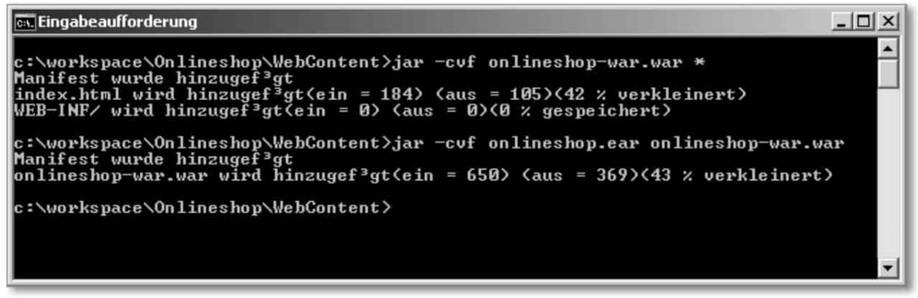


Abbildung 2.30 Die Paketierung der ».ear«-Datei

2.6.6 Die Installation

Nun werden wir die Beispielanwendung *onlineshop.ear* deployen. Für das Deployment einer Java-EE-Anwendung bieten sich grundsätzlich drei Möglichkeiten an. Sie können das Deployment durchführen, indem Sie

- `asadmin` mit Remote Commands einsetzen,
- den DAS über den Webbrowser bedienen oder
- die Java-EE-Anwendung einfach in das Verzeichnis *C:\glassfish4\glassfish\domains\ domain1\autodeploy* kopieren.

In diesem Abschnitt wird die Variante mit dem Webbrowser gezeigt. Für das Deployment der Java-EE-Anwendung klicken Sie im Fenster aus Abbildung 2.31 auf der linken Seite im Menü auf ANWENDUNGEN. Wenn Sie anschließend auf der rechten Seite den BEREITSTELLEN-Button betätigen, erscheint die Ansicht mit dem Titel ANWENDUNGEN ODER MODULE BEREITSTELLEN.

Hier können Sie die Java-EE-Anwendung nun hochladen. Wenn der Java EE Server auf einem entfernten Rechner liegt, muss die Java-EE-Anwendung mit der ersten Option hochgeladen werden. Für unsere Beispielanwendung können Sie die zweite Option verwenden und auf DATEIEN DURCHSUCHEN klicken.

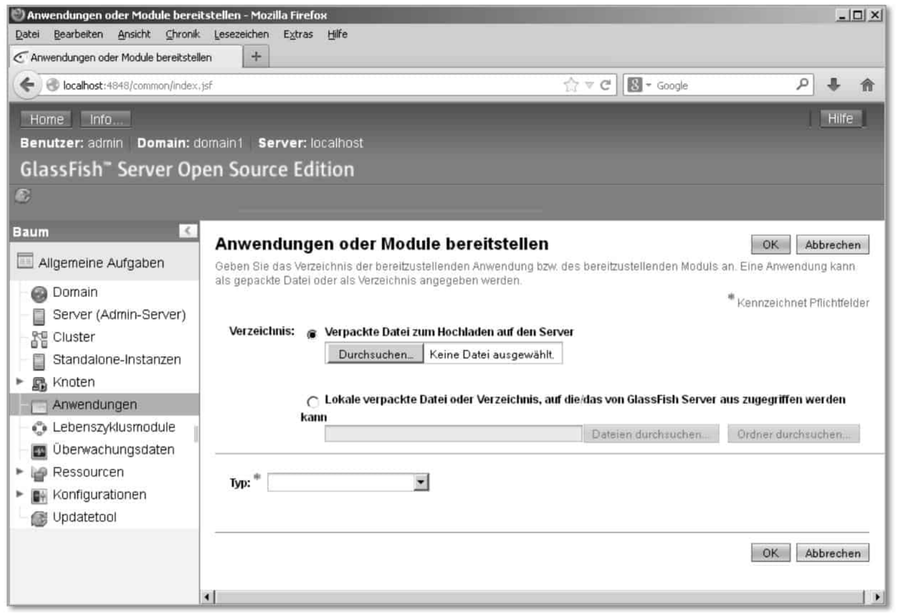


Abbildung 2.31 Das Bereitstellen der Java-EE-Anwendung

Weil sowohl vollständig Java-EE-konforme Anwendungen wie auch einzelne Web-Module installierbar sind, haben wir eigentlich die Wahl, ob wir nun die Datei *onlineshop.ear* oder die Datei *onlineshop-war.war* auswählen (siehe Abbildung 2.32).

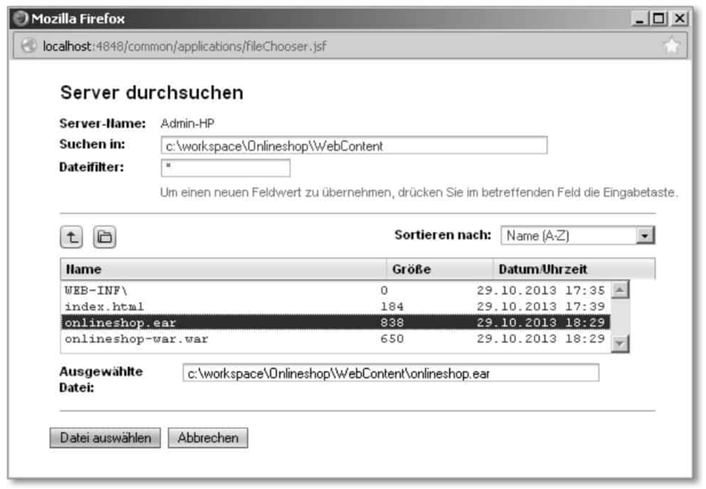


Abbildung 2.32 Die Auswahl der ».ear«-Datei

Selektieren Sie am besten die *.ear*-Datei, um das Deployment der vollständigen *.ear*-Datei einmal zu testen.

Mit einem Klick auf DATEI AUSWÄHLEN gelangen Sie zum vorherigen Fenster zurück. Wählen Sie dort unter den virtuellen Servern den Eintrag SERVER aus (siehe Abbildung 2.33).

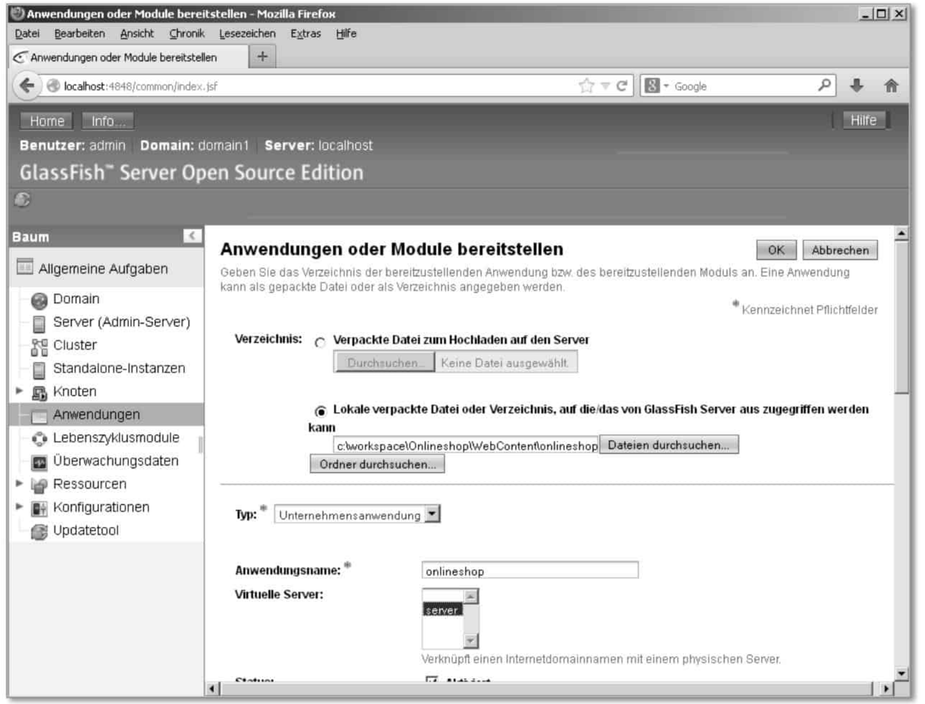


Abbildung 2.33 Die Auswahl des virtuellen Servers

Die übrigen Optionen können Sie in ihrer Voreinstellung belassen. Zuletzt bestätigen Sie mit OK.

Im nächsten Fenster (siehe Abbildung 2.34) sind Sie jetzt wieder auf der Anwendungsseite angekommen. Dort sollten Sie jetzt den Eintrag ONLINESHOP vorfinden.

Wenn Sie auf ONLINESHOP klicken, öffnet sich eine Editierseite zur Java-EE-Anwendung (siehe Abbildung 2.35).

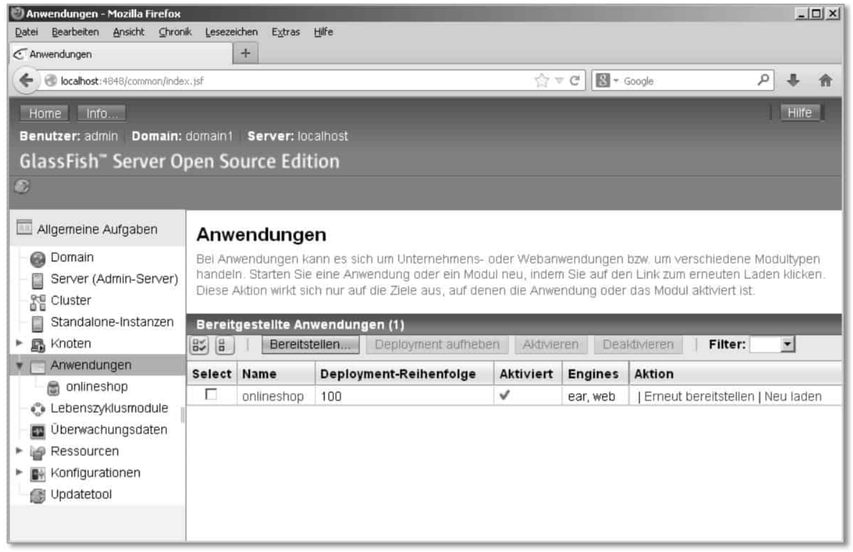


Abbildung 2.34 Die Onlineshop-Anwendung in der Administration Console

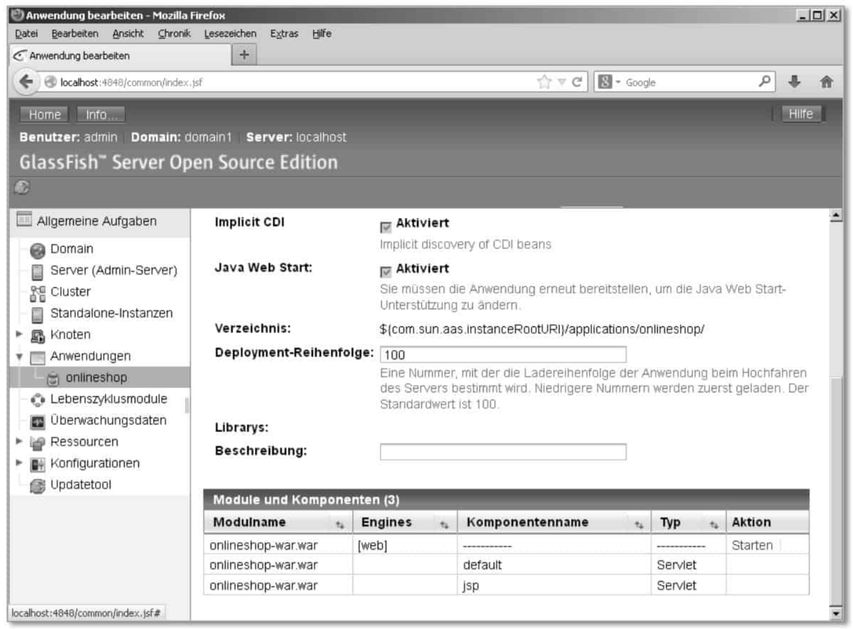


Abbildung 2.35 Anwendung bearbeiten

Klicken Sie dort unten rechts auf STARTEN. Dadurch öffnet sich eine Webseite in einem Internetbrowser, der Ihnen zwei Links anbietet (siehe Abbildung 2.36).



Abbildung 2.36 Links für die Webanwendung

Der erste Link ist für den Zugriff per HTTP, und der zweite für den Zugriff per HTTPS.



Abbildung 2.37 Die Onlineshop-Anwendung im Webbrowser

2.7 Eclipse einsetzen

In den bisherigen Beispielen wurde davon ausgegangen, dass Sie als Leser einen Texteditor für die Eingabe von Quelltexten einsetzen. Das ist selbstverständlich nicht wirklichkeitsnah. Wahrscheinlich sind Sie als Java-Entwickler bereits mit einer integrierten Entwicklungsumgebung (IDE) bestens vertraut. Eine Programmierung ohne IDE ist heutzutage ja auch nicht mehr zeitgemäß. In diesem Buch werden wir *Eclipse* als IDE einsetzen. Zur Zeit der Drucklegung war die Version *LUNA* die aktuellste.

In diesem Abschnitt wird gezeigt, wie GlassFish innerhalb der Eclipse-IDE verwendet werden kann. Damit Eclipse nicht durcheinanderkommt und die Domäne nicht doppelt ausgeführt wird, deinstallieren wir die Java-EE-Anwendung vom GlassFish-Server und stoppen die Domäne. Ein paralleles Arbeiten ist prinzipiell möglich. Allerdings müssten Sie in diesem Fall auf die Konfiguration der Portnummern achten, da eine Portnummer auf einem Host nur einmalig vergeben werden darf. Gehen Sie lieber den einfachen Weg, und neutralisieren Sie GlassFish, indem Sie die im letzten Kapitel vorgenommenen Einstellungen rückgängig machen.

Für die Deinstallation klicken Sie auf der linken Seite der GlassFish-Web-Konsole auf ANWENDUNGEN (siehe Abbildung 2.38). Auf der rechten Seite selektieren Sie die Onlineshop-Anwendung. Durch einen Klick auf DEPLOYMENT AUFHEBEN wird die Anwendung entfernt.

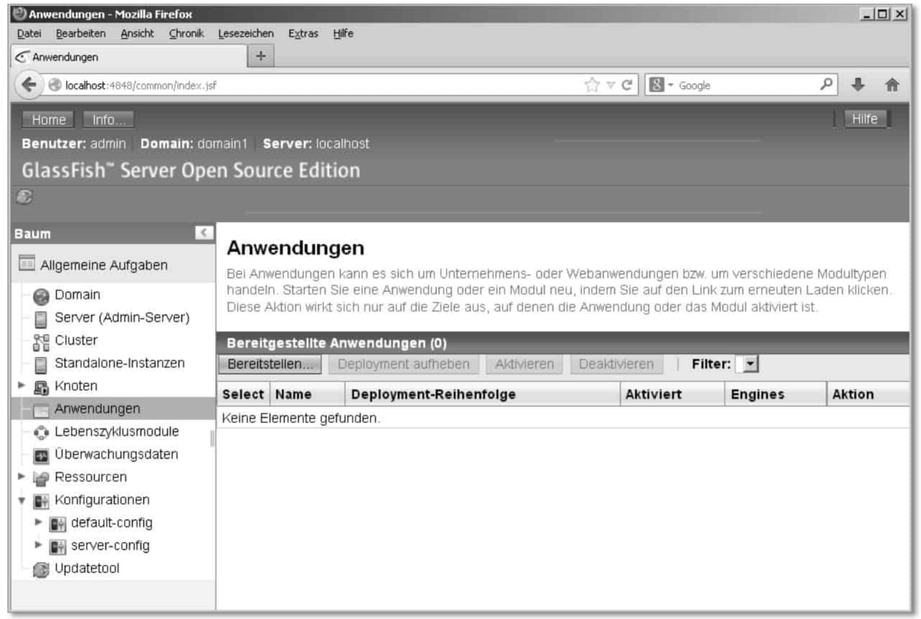


Abbildung 2.38 Der Onlineshop ist entfernt.

Außerdem sollten Sie den GlassFish-Server herunterfahren. Führen Sie hierfür im Programm `asadmin` das Kommando `stop-domain domain1` aus (siehe Abbildung 2.39).



Abbildung 2.39 Mit »stop-domain« fahren Sie die Domäne herunter.

2.7.1 Eclipse installieren

Sie können Eclipse von der Downloadseite der Eclipse-Foundation herunterladen. Sie finden Produkte unter

http://www.eclipse.org/downloads/ (siehe Abbildung 2.40).



Abbildung 2.40 Die Download-Seite von Eclipse

Beachten Sie, dass Sie aus den zahlreichen Download-Möglichkeiten die *Eclipse IDE for Java EE Developers* wählen müssen. Wenn Sie Eclipse heruntergeladen haben, sollte sich eine gezippte Datei in Ihrem Download-Verzeichnis befinden. In der gezippten Datei befindet sich ein Verzeichnis, das sich *eclipse* nennt. Extrahieren Sie dieses Verzeichnis auf Ihre Festplatte. Genau wie zu Anfang dieses Kapitels beschrieben, sollten Sie eine JDK 7 oder höher installiert haben und die Umgebungsvariablen `JAVA_HOME` und `PATH` entsprechend angepasst haben. Sie starten Eclipse, indem Sie die ausführbare Datei *eclipse.exe* im obersten Ordner des Eclipse-Verzeichnisses ausführen.

Fast alle Java-Entwickler haben bereits Erfahrung mit Eclipse sammeln können. Deshalb wird an dieser Stelle auch keine eingehende Bedienanweisung beigefügt. Stattdessen überspringe

ich diesen Schritt und zeige Ihnen direkt, wie GlassFish innerhalb von Eclipse genutzt werden kann.

2.7.2 Eclipse auf UTF-8 umstellen

Weiter oben wurde bereits erklärt, dass UTF-8 eine sehr bedeutende Zeichenenkodierung für Webanwendungen darstellt. Bestenfalls haben Sie auch den Extra-Abschnitt 2.5 durchgelesen, denn dort wurde alles Wichtige zu UTF-8 genauestens erklärt.

UTF-8 ist leider nicht die voreingestellte Zeichenenkodierung von Eclipse. Deshalb müssen wir noch an einige Änderungen vornehmen, bevor es gleich mit der automatisierten Erzeugung von Quelltexten losgehen kann.

Änderung des Default File Encodings

Die erste Stelle, über die wir eine Umstellung auf UTF-8 erwirken werden, findet sich in der Datei *eclipse.ini*. Diese Datei liegt im Eclipse-Hauptordner, d.h. im gleichen Ordner, in dem auch die ausführbare *eclipse.exe* enthalten ist. In dieser Datei sind alle Optionsparameter enthalten, die beim Start von Eclipse berücksichtigt werden. Fügen Sie dem Inhalt dieser Datei folgenden Optionsparameter hinzu:

```
...
-Dfile.encoding=UTF8
```

Listing 2.4 »eclipse.ini«

Danach starten Sie Eclipse neu, damit die Änderung wirksam wird.

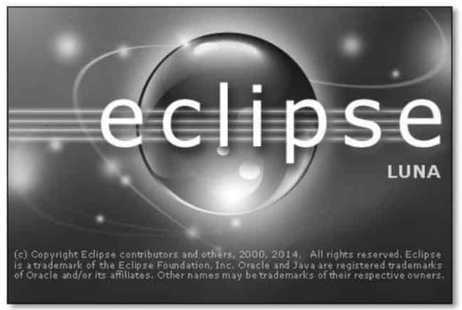


Abbildung 2.41 Eclipse LUNA

In Eclipse öffnen Sie nun das PREFERENCES-Fenster über WINDOW • PREFERENCES (siehe Abbildung 2.42). Dort selektieren Sie auf der linken Seite im Baum den Ordner GENERAL • WORKSPACE. Auf der rechten Seite sollte unter TEXT FILE ENCODING die Option DEFAULT(UTF8) selektiert sein.

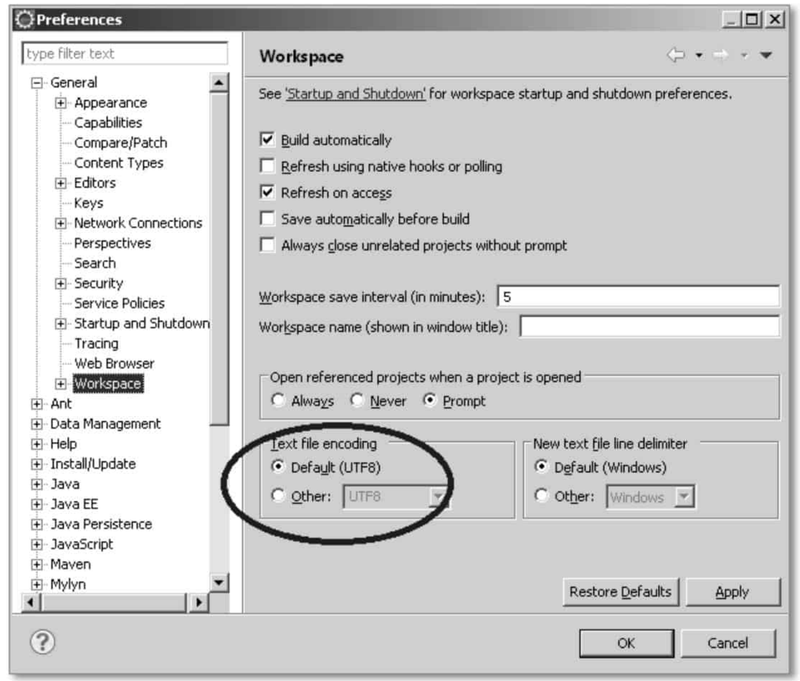


Abbildung 2.42 Default(UTF8) als »Text file encoding«

Änderung des Content Type

Als Nächstes öffnen Sie im PREFERENCES-Fenster den Ordner GENERAL • CONTENT TYPES (siehe Abbildung 2.43). Auf der rechen Seite erscheint danach eine Baumstruktur. Dort öffnen Sie den Eintrag TEXT. Kontrollieren Sie, ob im Eingabefeld DEFAULT ENCODING der Eintrag auf »UTF-8« gesetzt ist. Ändern Sie den Wert gegebenenfalls in »UTF-8« um, und klicken Sie auf UPDATE. Dies sollte auch alle darunter liegende Zweige abändern. Unterhalb von TEXT selektieren Sie den Eintrag HTML und darunter den Eintrag FACELETS. Im Eingabefeld DEFAULT ENCODING kontrollieren Sie, ob »UTF-8« eingetragen ist. Wenn nicht, ändern Sie den Wert wieder in »UTF-8« um und klicken auf UPDATE.

Genauso sollten Sie hier auch noch für andere Dateitypen (wie zum Beispiel für HTML-, CSS-, JavaScript- oder auch XHTML- und JSP-Dateien) kontrollieren, ob der CONTENT TYPE auf UTF-8 steht, bzw. ihn umstellen.

Zur Kontrolle öffnen Sie auf der linken Seite den Ordner WEB (sieheAbbildung 2.44). Unterhalb von WEB klicken Sie beispielsweise auf HTML FILES oder auf JSP FILES. Dort sollte nun ebenfalls »UTF-8« bzw. »ISO 10646« als Enkodierung gesetzt sein.

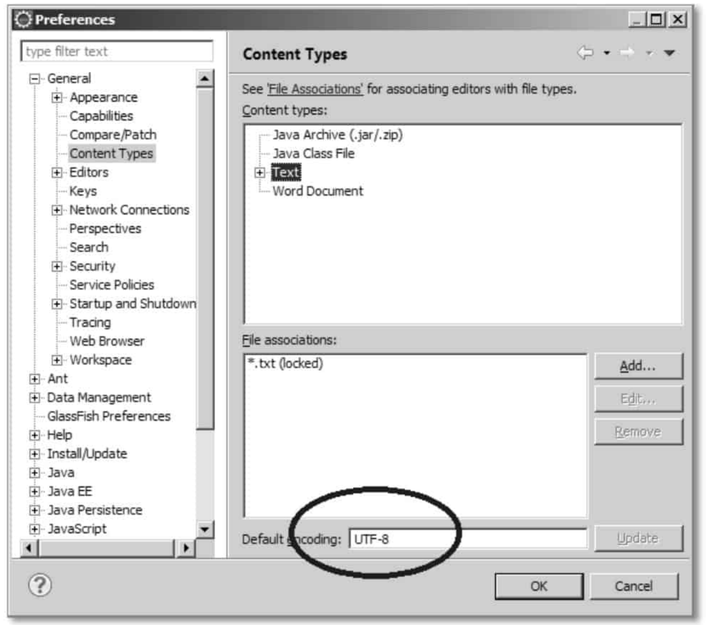


Abbildung 2.43 Die Umstellung des »Content Types«

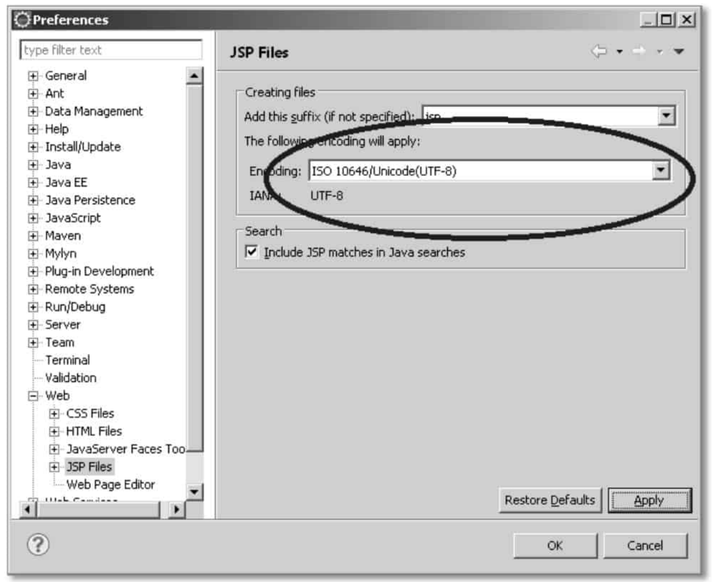


Abbildung 2.44 »ISO 10646/Unicode(UTF-8)« als Enkodierung

2.7.3 Die Installation des GlassFish-Plug-ins

Die Einbindung des GlassFish-Servers in Eclipse erfordert, dass zunächst ein Plug-in installiert wird. Dieser Schritt ist aber schnell durchgeführt. Klicken Sie hierfür in Eclipse auf WINDOW • PREFERENCES (siehe Abbildung 2.45).

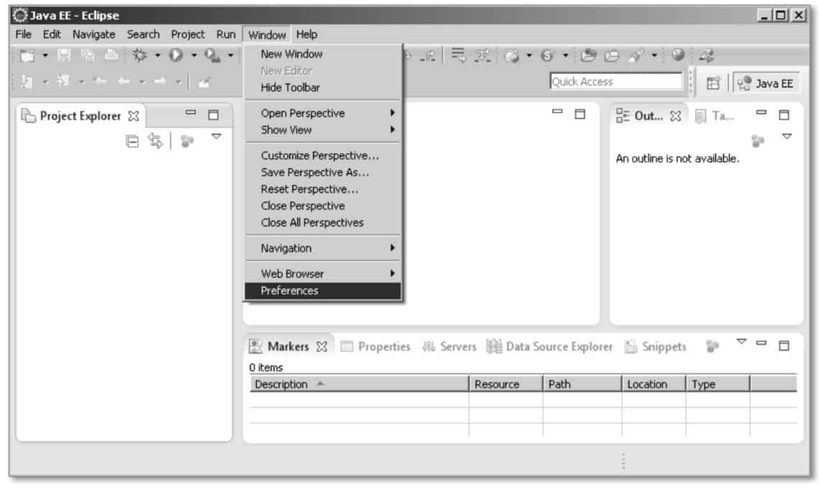


Abbildung 2.45 Die Hauptansicht von Eclipse

Im PREFERENCES-Fenster klicken Sie links auf SERVER und dann auf RUNTIME ENVIRONMENTS (siehe Abbildung 2.46).

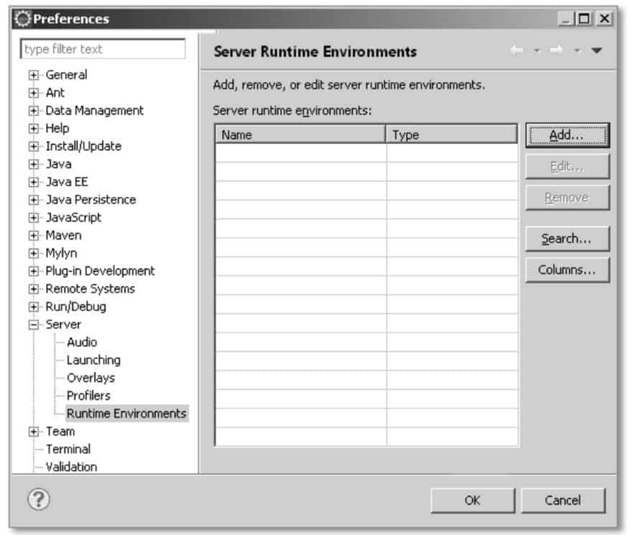


Abbildung 2.46 Die »Server Runtime Environments« im »Preferences«-Fenster

Klicken Sie in diesem Fenster auf der rechten Seite auf ADD, um ein neues Runtime-Environment hinzuzufügen (sieheAbbildung 2.47).

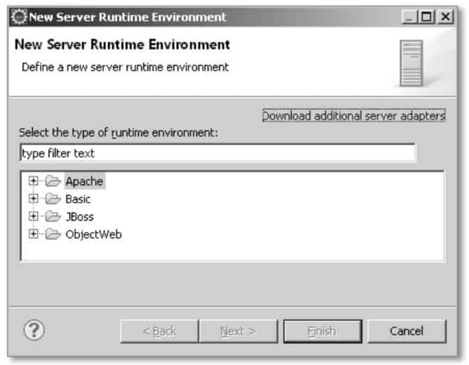


Abbildung 2.47 Das Fenster »New Server Runtime Environment«

Damit ein *Server Runtime Environment* eingerichtet werden kann, benötigt Eclipse spezielle Adapter. In der Ansicht des Fensters NEW SERVER RUNTIME ENVIRONMENT sehen Sie die voreingestellten Adapter. Weil kein GlassFish-Adapter vorgesehen ist, klicken Sie als Nächstes rechts oben auf DOWNLOAD ADDITIONAL SERVER ADAPTERS.



Abbildung 2.48 Die Oracle »GlassFish Tools« für Eclipse

Nach einer kurzen Weile sollten dort nach und nach die im Internet zur Verfügung stehenden Adapter erscheinen. In der kompletten Liste sollten Sie auch den Eintrag GLASSFISH TOOLS vorfinden (siehe Abbildung 2.48).

Klicken Sie dann auf NEXT. Anschließend sollte die Lizenzvereinbarung für diese Erweiterung erscheinen. Bestätigen Sie die Lizenzvereinbarung, und klicken Sie auf FINISH. Wenn die Installation fertiggestellt worden ist, müssen Sie Eclipse neu starten.

Nach dem Neustart öffnen Sie erneut das Fenster NEW SERVER RUNTIME ENVIRONMENT (siehe Abbildung 2.49). Dort sollte nun ein neuer Eintrag namens GLASSFISH vorhanden sein. Öffnen Sie den GLASSFISH-Ordner, und wählen Sie dort GLASSFISH 4.0 aus.

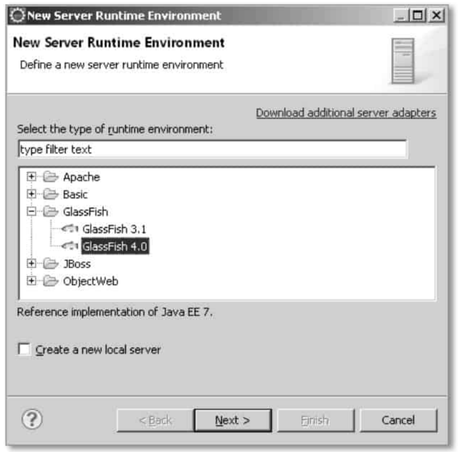


Abbildung 2.49 Die Auswahl des Runtime-Environment-Typs

Wenn Sie auf NEXT klicken, erscheint ein weiteres Fenster, in dem Sie den Ordner des GlassFish-Servers angeben müssen (siehe Abbildung 2.50).

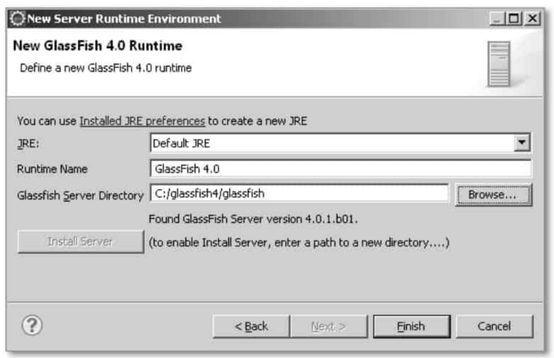


Abbildung 2.50 Die Auswahl des GlassFish-Verzeichnisses

Wenn Sie abschließend auf FINISH klicken, sehen Sie, dass nun ein GlassFish-Server-Runtime-Environment eingerichtet worden ist (siehe Abbildung 2.51).

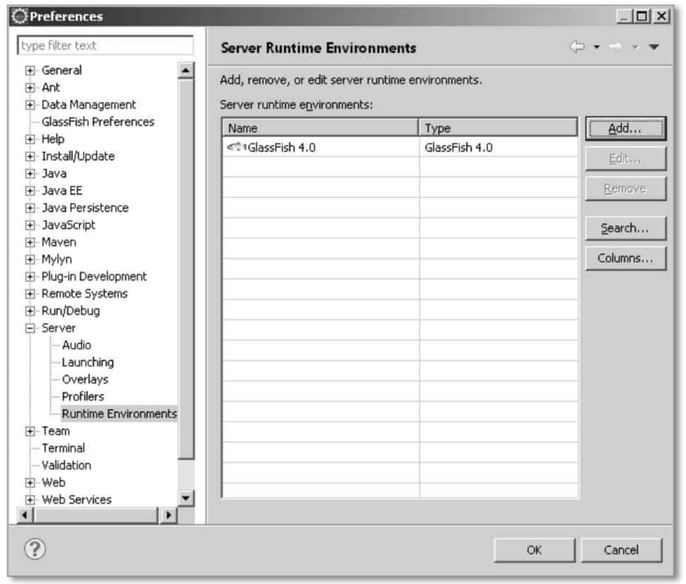


Abbildung 2.51 Das GlassFish-Server-Runtime-Environment

Nachdem das Runtime-Environment eines GlassFish-Servers erstellt ist, muss eine Domäne als virtueller Server innerhalb von Eclipse erzeugt werden. Öffnen Sie hierfür im Hauptfenster (rechts unten) die SERVER VIEW. In Abbildung 2.52 sehen Sie, dass in meinem Server-Register noch kein »virtueller Server« vorhanden ist.

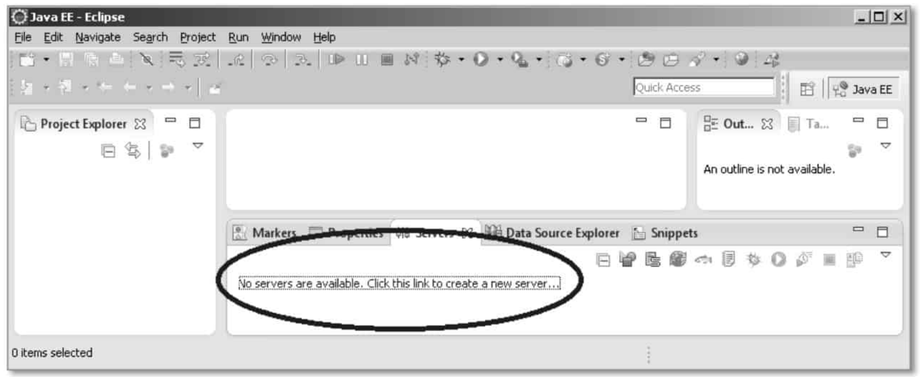


Abbildung 2.52 Die »virtuellen Server« in Eclipse

Wenn Sie im Server-Register auf NEW SERVER WIZARD klicken, öffnet sich ein Fester, in dem Sie den Server-Typ auswählen und dem Server einen Namen geben können. Ich habe meinen »virtuellen Server« GLASSFISH genannt (siehe Abbildung 2.53).

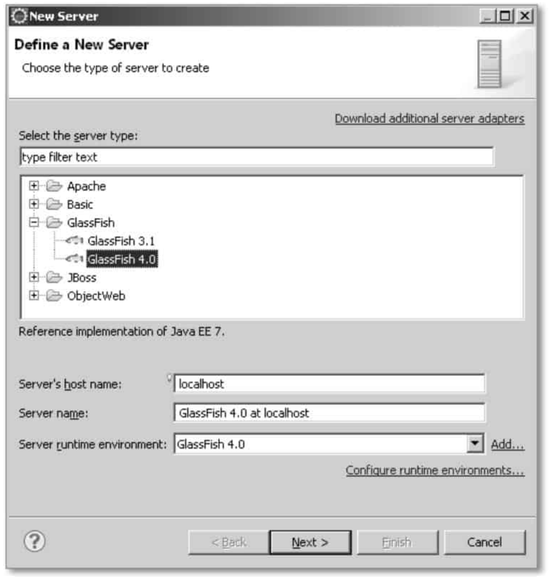


Abbildung 2.53 Die Erstellung eines virtuellen Servers

Anschließend werden Sie noch aufgefordert, einen Benutzernamen und ein Passwort einzugeben. Per Default ist der Benutzer *admin* eingestellt, für den wir noch kein Passwort vergeben haben. Die Checkbox PRESERVE SESSIONS ACROSS REDEPLOYMENT (siehe Abbildung 2.54) bietet die Möglichkeit, dass Ihre Sitzung erhalten bleibt, auch wenn Sie die Anwendung erneut installieren.

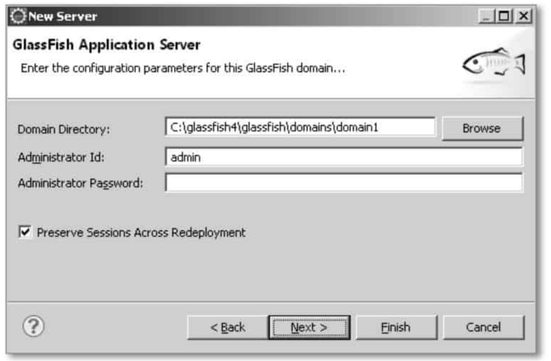


Abbildung 2.54 Die Eingabe von Benutzer und Kennwort

Zuletzt erscheint noch ein Fenster, in dem Sie dem Server vorhandene Projekte hinzufügen können (siehe Abschnitt 2.7.4). Klicken Sie auf FINISH, um die Server-Instanz zu installieren.

Um die Loggings ganz ausführlich in der Eclipse-View CONSOLE mitverfolgen zu können, gehen Sie erneut über das Hauptmenü auf WINDOW • PREFERENCES und wählen dort die GLASSFISH PREFERENCES aus (siehe Abbildung 2.55). Selektieren Sie die beiden Einträge ENABLE GLASSFISH PLUGIN LOG INFORMATION IN IDE LOG FILE und START THE GLASSFISH ENTERPRISE SERVER IN VERBOSE MODE.

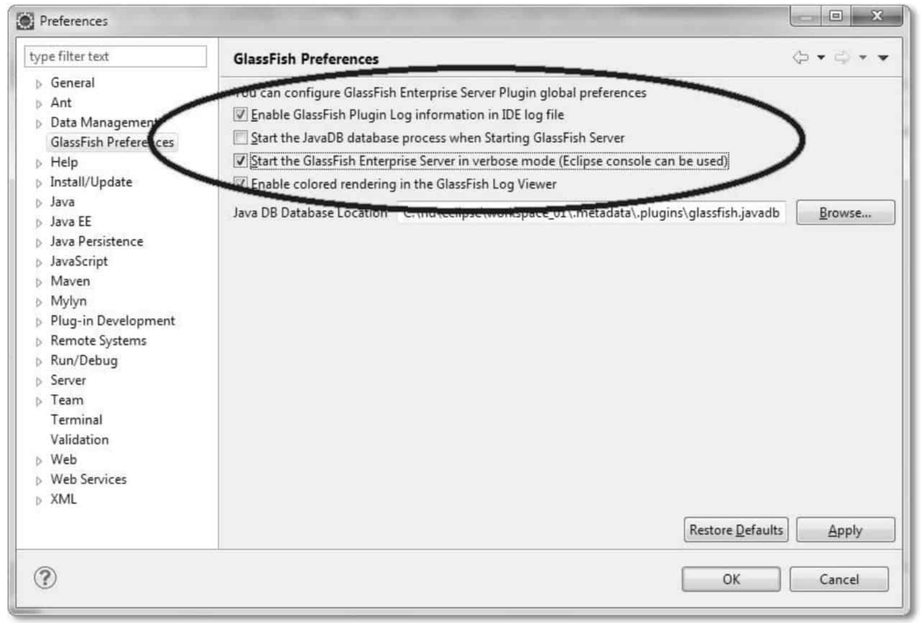


Abbildung 2.55 Die Beschaffung der ausführlichen Logging-Informationen in der View »Console«

2.7.4 Eine Java-EE-Anwendung in Eclipse erzeugen

Obwohl zu Beginn des Kapitels bereits deutlich wurde, dass die manuelle Erzeugung einer Java-EE-Anwendung kein Hexenwerk darstellt, ist es komfortabel, dass Eclipse sie auch automatisch erzeugen kann. Eigentlich benötigt Eclipse für eine Java-EE-Anwendung mit Web-Modul zwei Eclipse-Projekte, denn die Inhalte des Web-Moduls werden separat im sogenannten *dynamischen Webprojekt* gepflegt. Wir werden die Java-EE-Anwendung deshalb in zwei Schritten erstellen. Im ersten Schritt werden wir ein dynamisches Webprojekt erzeugen, das wir *onlineshop-war* nennen, und im zweiten Schritt erzeugen wir das Java-Enterprise-Projekt *onlineshop*. Das dynamische Webprojekt *onlineshop-war* lässt sich als Web-Modul in das Java-EE-Projekt integrieren, sodass das Java-EE-Projekt als vollständige Java-EE-Anwendung mit Web-Modul im GlassFish-Server ausgeführt werden kann.

Die Erzeugung des dynamischen Webprojekts

Um ein dynamisches Webprojekt in Eclipse zu erzeugen, klicken Sie im Hauptmenü auf FILE • NEW • DYNAMIC WEB PROJECT. Dabei erscheint ein Fenster, in dem Sie im oberen Bereich in einem Eingabefeld den Namen des dynamischen Webprojekts eintragen können (siehe Abbildung 2.56).

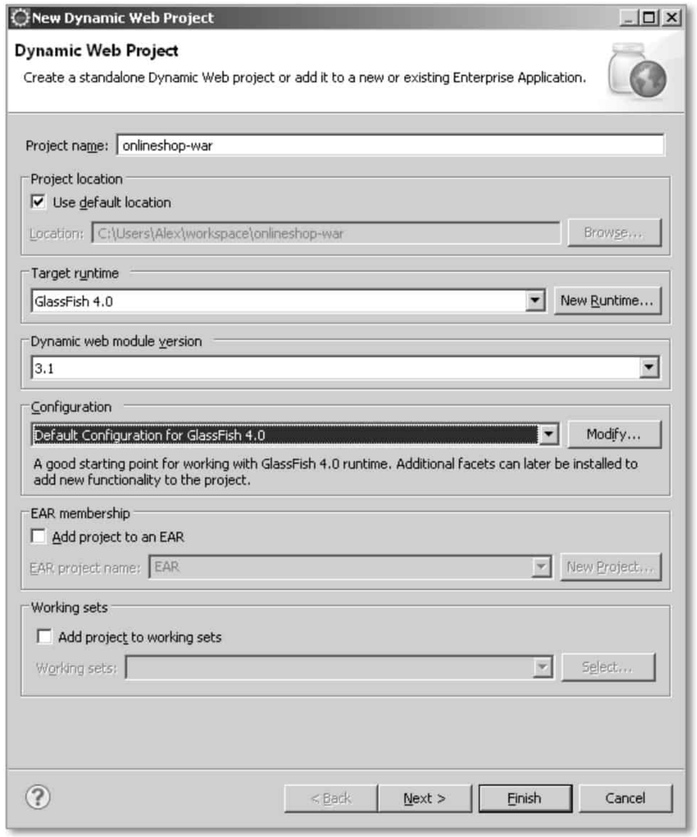


Abbildung 2.56 Die Erzeugung eines dynamischen Webprojekts

Für das Beispiel des Buches wurde der Bezeichner *onlineshop-war* gewählt, wobei alle Buchstaben kleingeschrieben sind. Dieser Bezeichner wird im Standardfall auch gleichzeitig der Name des Web-Moduls und somit per Default auch die aufzurufende *context-root* in der Adressleiste des Browsers sein. Die context-root ist der Teil der URL, der den Namen der Webanwendung ausmacht. In der URL befindet sich die context-root hinter der Domain und dem Port:

```
http://localhost:8080/onlineshop-war/index.html
```

Weiter unten können Sie die TARGET RUNTIME auswählen. Sie sollte die von Ihnen installierte GlassFish-Version enthalten.

In der Auswahl DYNAMIC WEB MODULE VERSION kann die Version der Webapplication selektiert werden. Die Vorauswahl von Eclipse brauchen Sie nicht zu ändern, da Eclipse die für den Server höchste Version bereits vorsieht.

Wenn Sie auf NEXT klicken, gelangen Sie zu einem Fenster, in dem Sie den Quellenordner abändern können. Belassen Sie dort den Wert auf der Vorgabe *src*.

Mit noch einem Klick auf NEXT kommen Sie zu einem Fenster, in dem Sie der CONTEXT ROOT einen Bezeichner geben können (siehe Abbildung 2.57). Obwohl sich die context-root hiermit per Konfiguration ändern lässt, ist es doch eigentlich praktisch, dass dieser Bezeichner eigentlich gar nicht gesetzt werden muss. Aus diesem Grund und auch, um die Beispiele in den Screenshots nachvollziehen zu können, behalten Sie auch dort den voreingestellten Wert bei.

Zusätzlich kann der Name des CONTENT DIRECTORY geändert werden. Es handelt sich hierbei lediglich um den Pfadnamen, den Eclipse für seine interne Verwaltung der Projektdateien nutzt. Am besten ist es, wenn Sie diesen Namen auf dem Default-Wert lassen.

Darunter befindet sich noch eine Checkbox, über die der Deployment-Deskriptor für das Web-Modul generiert werden kann. Der Deployment-Deskriptor ist eine Datei namens *web.xml*, durch die das Web-Modul per XML konfiguriert werden kann. Diese Datei ist optional und braucht jetzt nicht erzeugt zu werden. Auf den Deployment-Deskriptor werden wir in Kapitel 4, »Servlets«, noch zurückkommen.

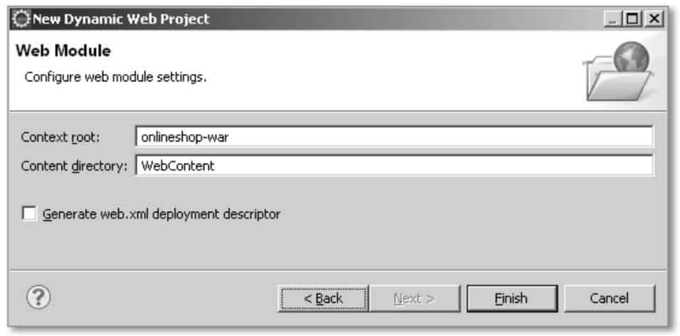


Abbildung 2.57 Die Anpassung von »Context root« und »Content directory«

Mit einem Klick auf FINISH wird das dynamische Webprojekt *onlineshop-war* erzeugt.

Wenn Sie nun auf die linke Seite des Eclipse-Fensters schauen, werden Sie das neue Projekt *onlineshop-war* sehen. Öffnen Sie dort den Ordner WEBCONTENT • WEB-INF. Wenn GlassFish Ihr Runtime-Server ist, sollten Sie in dem Ordner */WEB-INF* die Datei *glassfish-web.xml* (oder

bei älteren GlassFish-Versionen die Datei *sun-web.xml*) entdecken. Diese Datei gehört standardmäßig nicht zu einem Web-Modul. Dennoch hat Eclipse sie für uns angelegt, denn hierdurch erhalten wir einen Hinweis darauf, dass individuelle Einstellungen, die lediglich GlassFish versteht, hier eingetragen werden können. Beispielsweise sollte Eclipse innerhalb der Datei die context-root der Webanwendung definiert haben. Wenn Sie im obigen Wizard den Default-Wert beibehalten haben, sollte der Eintrag wie folgt aussehen:

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

<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish Application Ser-
ver 3.1 Servlet 3.0//EN" "http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
    <context-root>/onlineshop-war</context-root>
 </glassfish-web-app>
```

Listing 2.5 »glassfish-web.xml«

Die Angabe der context-root ist in unserem Fall unnötig. Für eine andere Einstellung ist die *glassfish-web.xml* jedoch hilfreich. Denn über das Element `parameter-encoding` können wir GlassFish anweisen, dass er die von den Benutzern in den HTML-Formularen eingegebenen HTTP-Parameter als UTF-8 Zeichen enkodieren soll. Diese Einstellung brauchen wir, um die Webanwendung international – also auch mit deutschen Umlauten oder chinesischen Symbolen – einsetzen zu können.

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

<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassFish Application Ser-
ver 3.1 Servlet 3.0//EN" "http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
    <context-root>/onlineshop-war</context-root>
    <parameter-encoding default-charset="UTF-8"/>
</glassfish-web-app>
```

Listing 2.6 »glassfish-web.xml«

Die Erzeugung des Enterprise-Application-Projekts

Im nächsten Schritt werden wir das *Enterprise-Application-Projekt* erzeugen. Hierfür klicken Sie im Hauptmenü auf FILE • NEW • ENTERPRISE APPLICATION PROJECT.

Anschließend öffnet sich ein Fenster, in dem Sie oben den Namen des Enterprise-Application-Projekts eintragen (siehe Abbildung 2.59). Auch hier ist es wieder so, dass dieser Name als Bezeichner für die paketierte Datei verwendet wird. Setzen den Namen deshalb auf »onlineshop«. Behalten Sie für die übrigen Einträge die voreingestellten Werte bei.

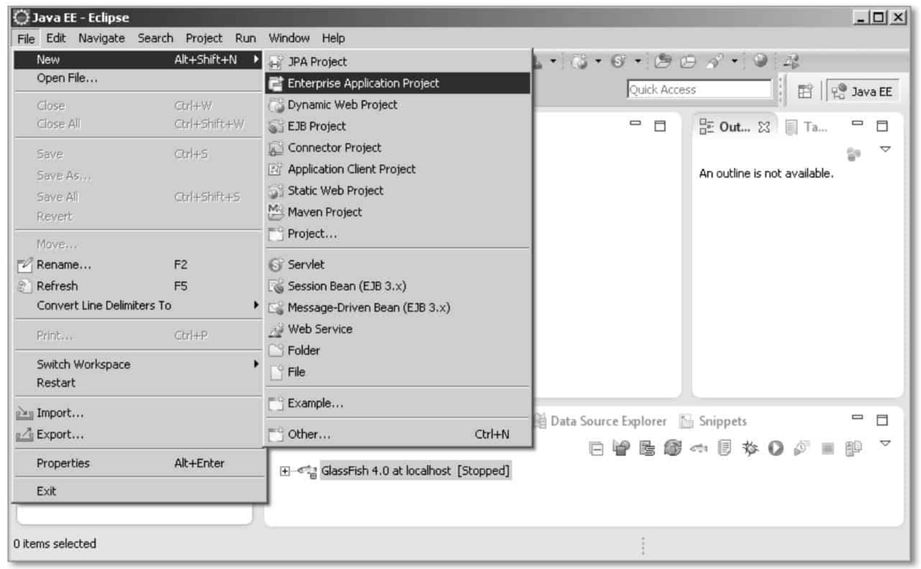


Abbildung 2.58 Mit »File • New • Enterprise Application Project« starten Sie den Wizard für die Erzeugung eines Enterprise-Application-Projekts.

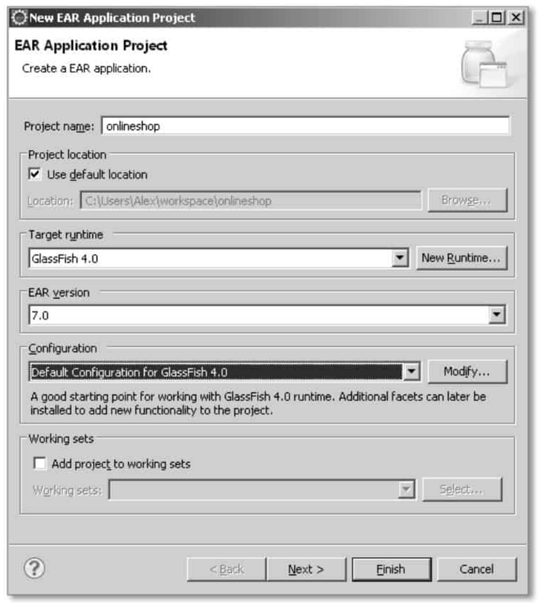


Abbildung 2.59 Die Erzeugung des Enterprise-Application-Projekts

Wenn Sie auf NEXT klicken, erscheint ein neues Fenster, in dem Sie Java-EE-Module dem Java-Enterprise-Application-Projekt zuordnen können (siehe Abbildung 2.60). Da wir vorhin das dynamische Webprojekt *onlineshop-war* erzeugt haben, können wir dieses dem Enterprise-Projekt jetzt hinzufügen. Bei CONTENT DIRECTORY lassen wir »EarContent« stehen.

Auch bei der Erstellung eines Enterprise-Application-Projekts kann wieder ein Deployment-Deskriptor erzeugt werden. Dieser Deployment-Deskriptor nennt sich *application.xml*. Auch er ist optional und braucht für dieses Beispiel nicht erzeugt zu werden.

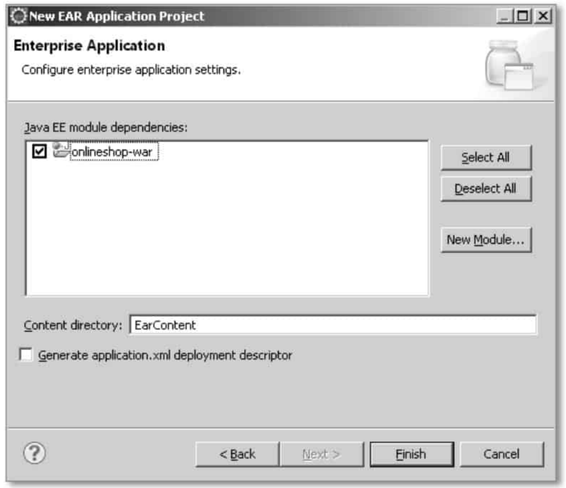


Abbildung 2.60 Das Hinzufügen des Java-EE-Moduls »onlineshop-war«

Die Erstellung einer UTF-8-enkodierten HTML-Datei

Im Hauptfenster von Eclipse müssten nun zwei Projekte, nämlich *onlineshop* und *onlineshop-war.war* zu sehen sein. Markieren Sie das dynamische Webprojekt *onlineshop-war*, und klicken Sie im Menü des Hauptfensters auf FILE • NEW • OTHER. Öffnen Sie im erscheinenden Fenster den Ordner WEB, und wählen Sie den Eintrag HTML FILE aus (siehe Abbildung 2.61).

Mit NEXT gelangen Sie in ein neues Fenster, in dem Sie der HTML-Datei einen Namen geben und den Bestimmungsort festlegen können. Setzen Sie den FILE NAME auf »index.html« (siehe Abbildung 2.62).

Mit einem Klick auf NEXT kommen Sie in das Fenster, in dem Sie sich aus verschiedenen Templates ein Muster für die HTML-Seite aussuchen (siehe Abbildung 2.63). Selektieren Sie dort den Eintrag NEW HTML FILE (5). Mit einem Klick auf FINISH wird die Datei erzeugt.

Öffnen Sie im Projektbaum des Projekts *onlineshop-war* den Unterordner WEBCONTENT. Eclipse sollte dort die Datei *index.html* erzeugt haben. Klicken Sie mit der rechten Maustaste auf die *index.html*, um die Properties der Datei anzuschauen (siehe Abbildung 2.64). Dort sollte unter TEXT FILE ENCODING die Zeichenenkodierung zu sehen sein.

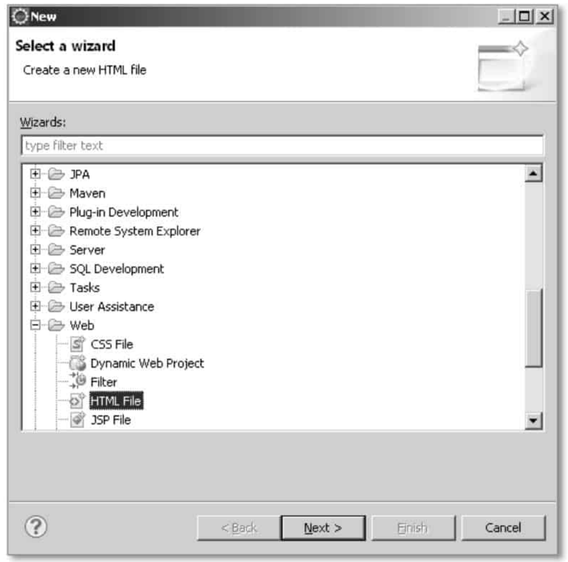


Abbildung 2.61 Die Auswahl von »HTML File«

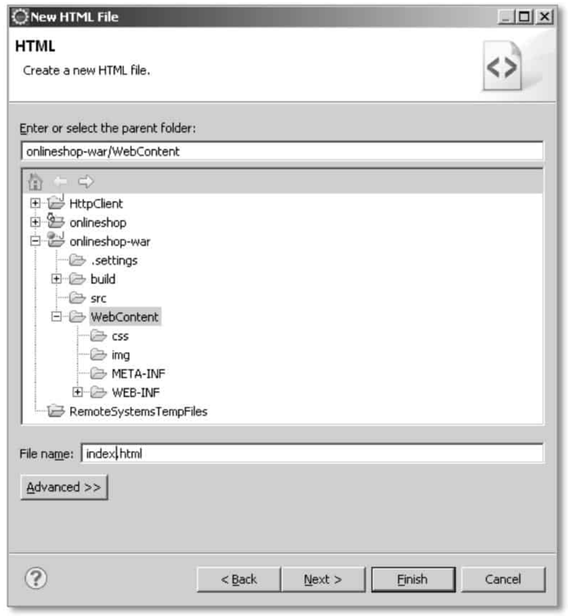


Abbildung 2.62 Den Namen der HTML-Datei eingeben

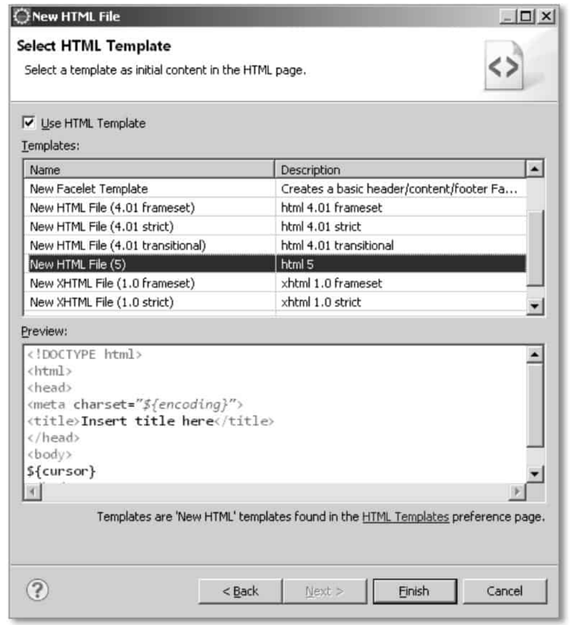


Abbildung 2.63 Die Auswahl von »New HTML File(5)«

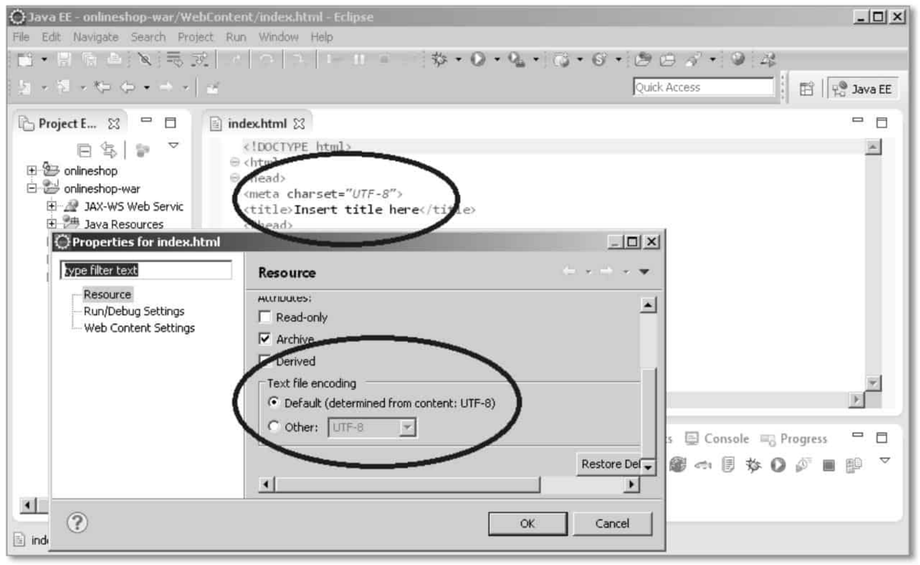


Abbildung 2.64 Die UTF-8-Einstellung in der HTML-Seite

In der Abbildung 2.64 sehen Sie, dass die Datei in der UTF-8-Enkodierung erzeugt wurde, aber dass auch zusätzlich auf den Content-Type UTF-8 hingewiesen wird. Obgleich Eclipse hiermit dem Anschein nach unnötig oft die gleiche Information setzt, ist dies dennoch die übliche Vorgehensweise.

Die Onlineshop-Willkommensseite erstellen

Zur Übung ersetzen Sie den von Eclipse erzeugten Quelltext der Datei *index.html* durch folgenden Quelltext und führen das Programm aus.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8"/>
        <title>Onlineshop</title>
    </head>
    <body>
        <h1>Onlineshop</h1>
    </body>
</html>
```

Listing 2.7 »index.html«

Um die Java-EE-Anwendung zu deployen, fügen Sie sie zunächst der GlassFish-Server-Instanz hinzu. Klicken Sie hierfür mit der rechten Maustaste auf die GlassFish-Server-Instanz. Im Kontextmenü klicken Sie auf ADD AND REMOVE (siehe Abbildung 2.65).

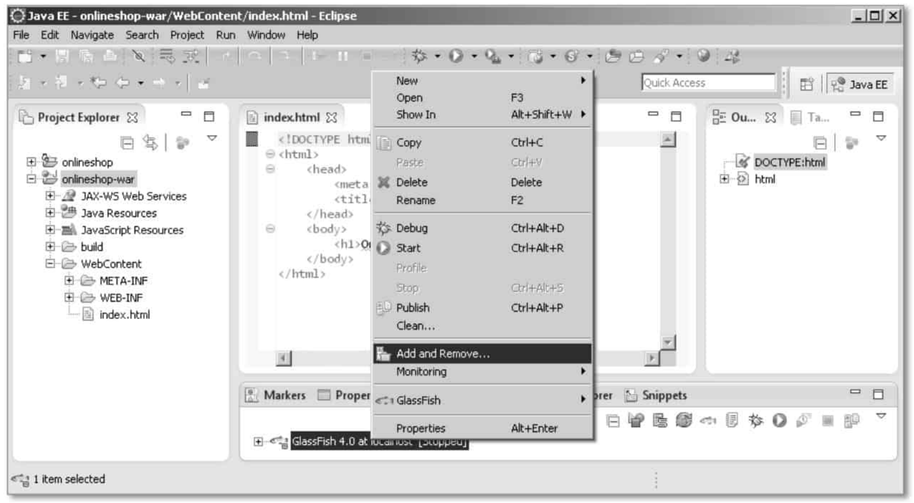


Abbildung 2.65 Der Befehl »Add and Remove«

Im ADD AND REMOVE...-Fenster selektieren Sie auf der linken Seite den Eintrag »onlineshop« und klicken anschließend auf den Button ADD> (siehe Abbildung 2.66).

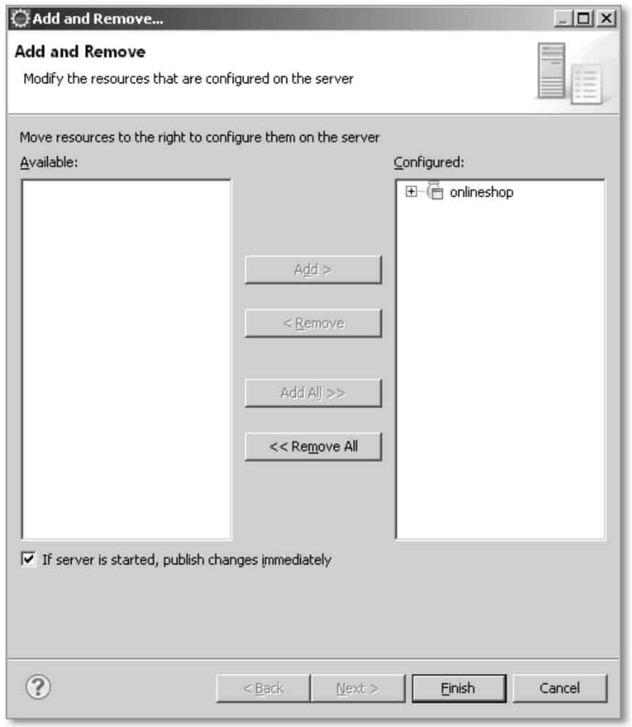


Abbildung 2.66 Das »Add and Remove...«-Fenster

Wenn anschließend der Eintrag »onlineshop« auf die rechte Seite gewandert ist, schließen Sie den Vorgang mit FINISH ab.

In der SERVER-View starten Sie den Server mit einem Klick auf den grünen Play-Button (siehe Abbildung 2.67). Hiermit wird das GlassFish-Plug-in die Anwendung *onlineshop-war* auch deployen. Wenn das mal nicht reibungsfrei gelingt, betätigen Sie in der SERVER-View den Button PUBLISH TO THE SERVER (mit einem Rechner und einem Dokument als Icon). Dieser Button ist eigentlich dafür gedacht, Änderungen an der Anwendung auf den Server zu übernehmen, ohne GlassFish neu starten zu müssen.

Geben Sie nun in der Adressleiste eines Webbrowsers die URL *http://localhost:8080/onlineshop-war/* ein, um die Startseite der Java-EE-Anwendung *Onlineshop* zu sehen (siehe Abbildung 2.68).

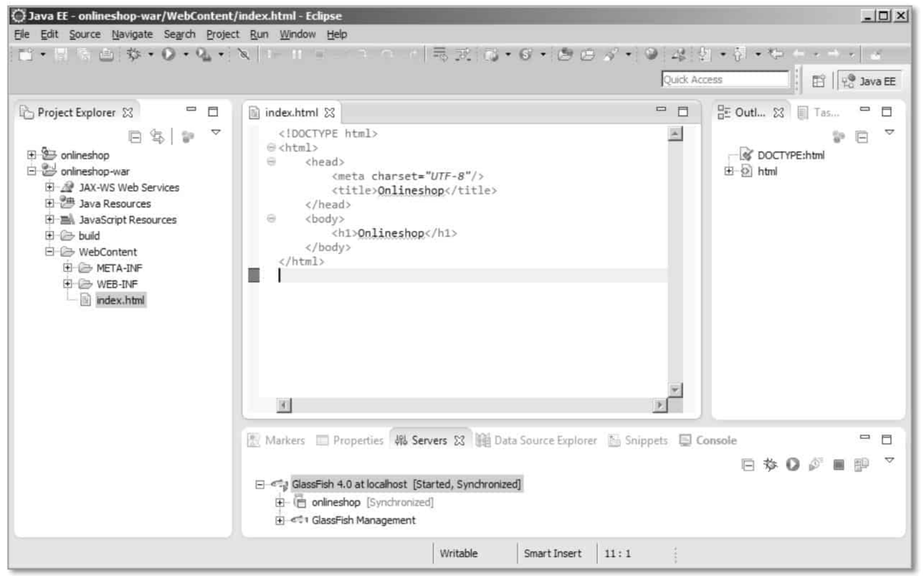


Abbildung 2.67 Den GlassFish-Server starten

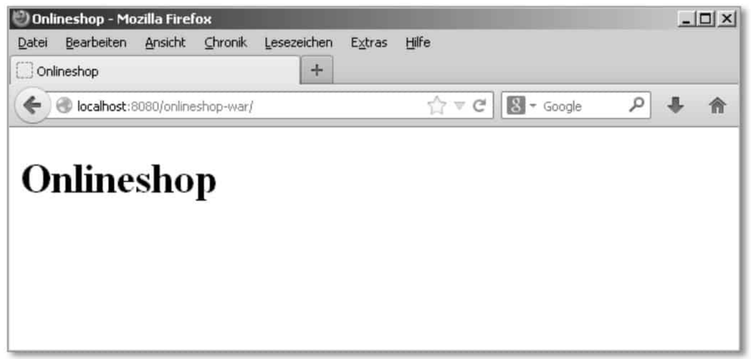


Abbildung 2.68 Die Startseite des Onlineshops

2.8 Extra-Abschnitt: NetBeans einsetzen

In diesem Extra-Abschnitt wird zusätzlich zu der Eclipse-IDE auch noch die integrierte Entwicklungsumgebung *NetBeans* gezeigt. Obwohl dieses Buch den Schwerpunkt auf Eclipse setzt, könnte dieser Abschnitt für den einen oder anderen Entwickler von Interesse sein, der die Entwicklungsumgebung NetBeans ausprobieren möchte.

2.8.1 NetBeans installieren

Während ich diese Zeilen eintippe, ist NetBeans mit der Version *7.4 Beta* das Aktuellste, was Oracle momentan zu bieten hat. Oracle hat auch hier wieder den Fokus auf HTML5 gesetzt.

NetBeans kann unter *https://netbeans.org/downloads/* heruntergeladen werden. Folgen Sie den Links, bis Sie auf der Download-Seite angekommen sind (siehe Abbildung 2.69).

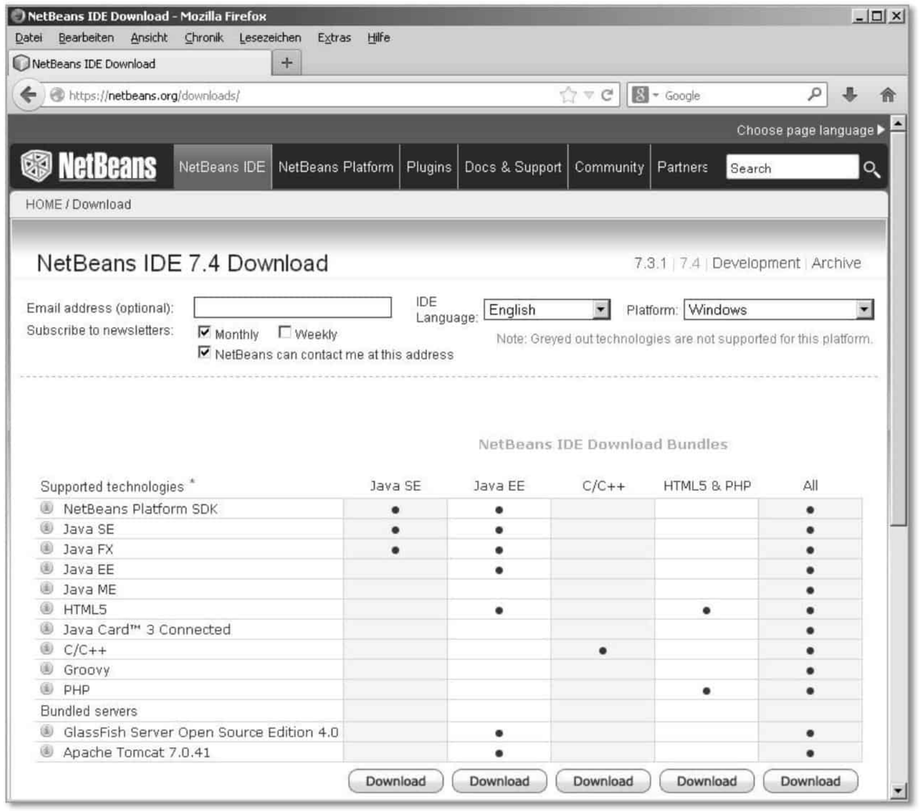


Abbildung 2.69 Die Download-Seite von NetBeans

Auf der Download-Seite werden verschiedene Varianten von NetBeans angeboten. Wählen Sie oben rechts Ihr Betriebssystem aus, und klicken Sie auf den Download-Link für die Java-EE-Plattform. Laden Sie NetBeans herunter, und starten Sie das Installationsprogramm (siehe Abbildung 2.70).

In NetBeans 7.4 wird die Installation des GlassFish-Servers 4.0 angeboten, denn sie gehört als Bestandteil zu dem Gesamtpaket.

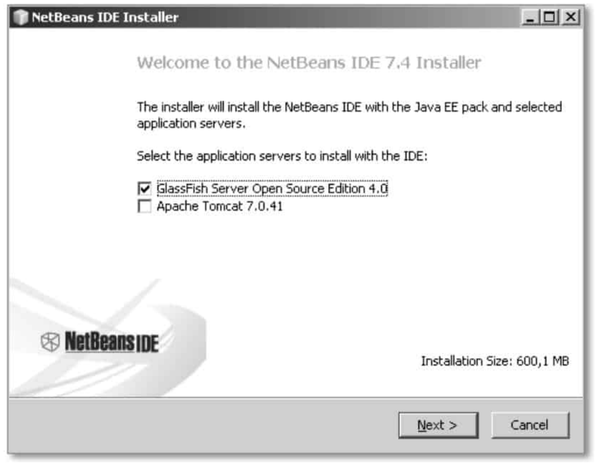


Abbildung 2.70 Die Willkommensseite des NetBeans-Installers

2.8.2 NetBeans auf UTF-8 umstellen

Um bei NetBeans für ein reibungsloses Arbeiten mit UTF-8 zu sorgen, müssen wir vorab mit einer besonderen Default-Option starten. Die Default-Optionen, die Sie gleich ändern werden, finden Sie in der Datei *netbeans\etc\netbeans.conf*. Hierfür benötigen Sie Administrator-Rechte.

Achtung

Bei Microsoft Windows könnte es zu Problemen kommen, wenn Sie die Änderung direkt im Unterordner von *C:\Programme\netbeans* durchführen möchten. Die Lösung ist, die Datei in einem Benutzerverzeichnis (zum Beispiel im Desktop) durchzuführen und die geänderte Datei zurückzukopieren.

Im Inhalt der Datei suchen Sie nun den Schlüssel `netbeans_default_options` auf. Wenn Sie ihn gefunden haben, fügen Sie im Wert die Zeichenkette `-J-Dfile.encoding=UTF-8` als Optionsparameter hinzu. Das Ergebnis sollte beispielsweise wie folgt aussehen:

```
...
netbeans_default_options="-J-Dfile.encoding=UTF-8 -J-client -J-Xss2m -J-Xms32m -J-
XX:PermSize=32m -J-Dapple.laf.useScreenMenuBar=true -J-Dapple.awt.graphics.Use-
Quartz=true -J-Dsun.java2d.noddraw=true -J-Dsun.java2d.dpiaware=true -J-
Dsun.zip.disableMemoryMapping=true -J-Dsun.awt.disableMixing=true"
...
```

Listing 2.8 netbeans\etc\netbeans.conf

Nach der Änderung in der Datei *netbeans.conf* muss NetBeans neu gestartet werden. Von nun an wird NetBeans für Quelltexte in neuen Projekten stets die Zeichenkodierung UTF-8 nutzen.

2.8.3 Eine Java-EE-Anwendung in NetBeans

In diesem Abschnitt werden wir eine Java-EE-Anwendung mit NetBeans erzeugen. In NetBeans klicken Sie auf FILE • NEW PROJECT. Als Java-EE-Projekt-Typ wählen Sie JAVA EE • ENTERPRISE APPLICATION aus (siehe Abbildung 2.71).

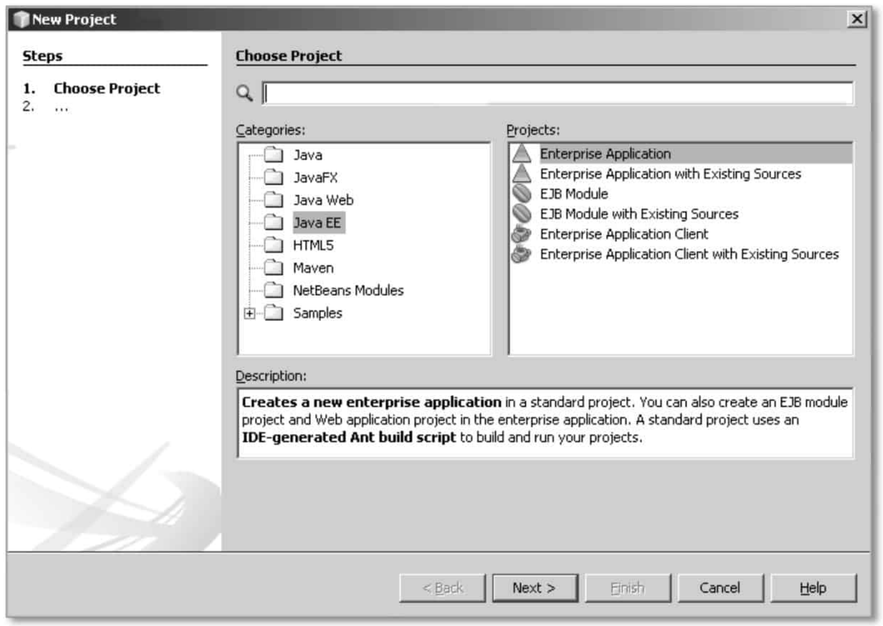


Abbildung 2.71 So erstellen Sie eine Java-EE-Enterprise-Application.

Auf der rechten Seite klicken Sie auf ENTERPRISE APPLICATION und dann auf NEXT.

Im nächsten Fenster geben Sie »Onlineshop« als Projektnamen ein (siehe Abbildung 2.72).

Klicken Sie anschließend auf NEXT.

Im darauf folgenden Fenster (siehe Abbildung 2.73) deaktivieren Sie das Kontrollkästchen CREATE EJB-MODULE. Das andere Kontrollkästchen lassen Sie aktiviert. Klicken Sie dann auf FINISH.

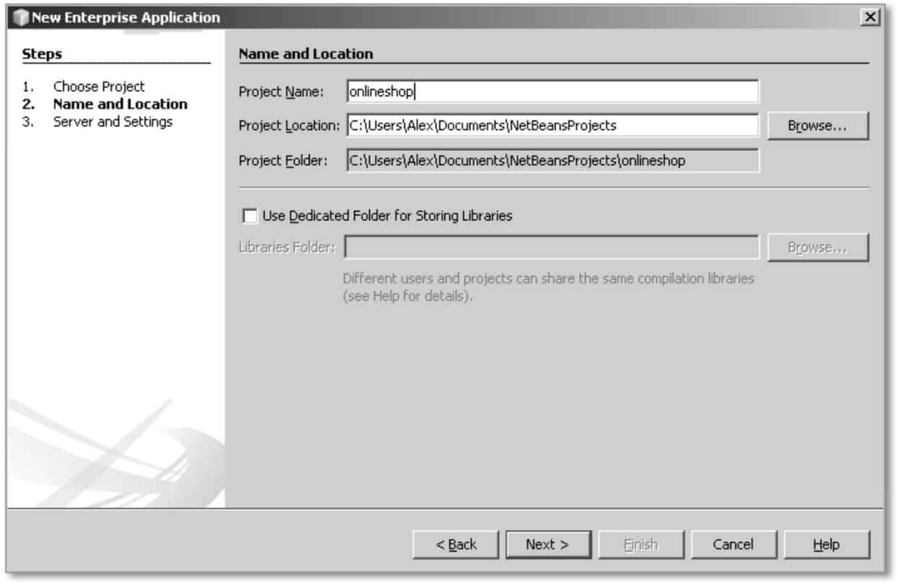


Abbildung 2.72 Die neue Java-EE-Anwendung »Onlineshop«

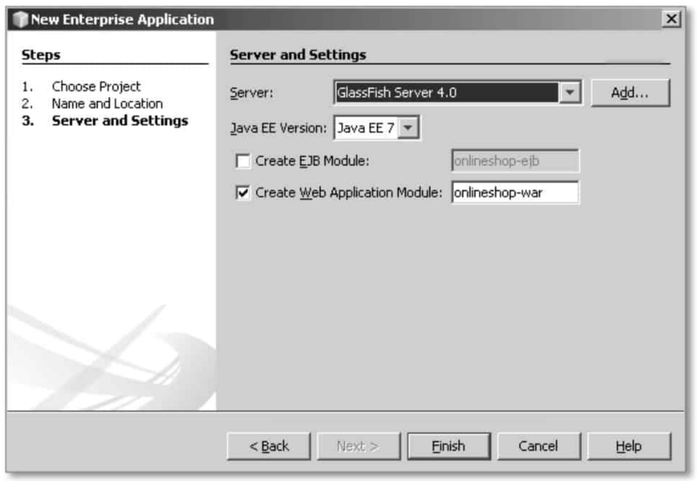


Abbildung 2.73 Deaktivieren Sie das Kontrollkästchen bei »Create EJB-Module«.

Im Hauptfenster von NetBeans sehen Sie jetzt zwei NetBeans-Projekte, nämlich *onlineshop* und *onlineshop-war.war*. Öffnen Sie im Projektbaum die Datei *index.html*, um sie anzuschauen (siehe Abbildung 2.74).

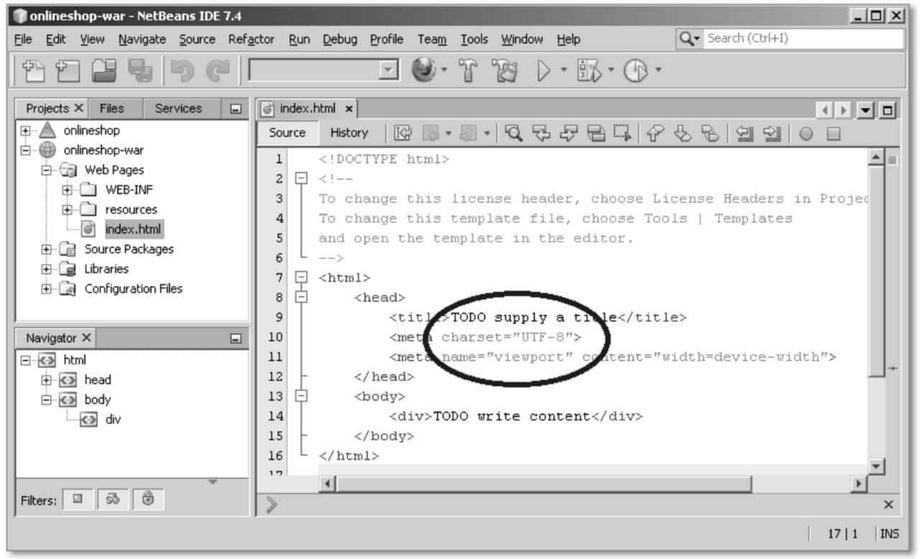


Abbildung 2.74 Die Zeichenenkodierung der »index.html«

Wie Sie sehen, setzt NetBeans das verkürzte HTML5-Meta-Tag `charset` ein.

Kapitel 7
JDBC

»Nur wer sich in Genügsamkeit genügt, hat stets genug.«

Lao-Tse

In diesem Kapitel wird die *Java Database Connectivity* (JDBC) behandelt.

Die JDBC-API stellt eine einheitliche Programmierschnittstelle für den Zugriff auf eine relationale Datenbank dar. Der Quelltext, der durch die Nutzung der JDBC-API entsteht, ist portierbarer. Damit ist gemeint, dass der gleiche Quelltext, der für den Zugriff auf eine Oracle-Datenbank geschrieben wurde, auch für DB2 oder MySQL-Datenbankmanagementsysteme verwendet werden kann.

Die JDBC-API wurde im Jahre 1996 in der ersten Version herausgebracht. Seither wird sie stetig verbessert und mit fortwährend neuen Versionen veröffentlicht. Die Spezifikation der aktuellen Version 4.0 kann hier heruntergeladen werden:

http://download.oracle.com/otn-pub/jcp/jdbc-4.0-fr-eval-oth-JSpec/jdbc-4_0-fr-spec.zip

Obgleich heutzutage meistens Persistenz-Frameworks verwendet werden und die handgeschriebenen SQL-Anweisungen altmodisch erscheinen, wird in manchen Projekten nach wie vor direkt JDBC ohne weiteres Framework eingesetzt. Aber auch wenn Sie in Ihrem Java-EE-Projekt mit einem High-Level-Framework wie JPA arbeiten sollten, gibt Ihnen das Wissen über die Low-Level-API ein tieferes Verständnis dafür, wie die Persistenz auch bei dem High-Level-Framework funktioniert. Außerdem werden JDBC-Kenntnisse bei einem Java-EE-Entwickler in jedem Fall vom Arbeitgeber vorausgesetzt.

7.1 Der direkte Zugriff per JDBC

In diesem Unterkapitel werden wir per JDBC direkt auf die Datenbank zugreifen. Dabei werden wir zunächst administrativ tätig werden. Denn damit eine Anwendung die JDBC-API verwenden kann, muss sie einen JDBC-Treiber einbinden. In diesem Unterkapitel werden wir den Treiber als sogenannte *Driver Definition* einbinden, sodass er von allen Eclipse-Projekten aus über einen definierten Namen verwendet werden kann.

In Kapitel 6, »Die relationale Datenbank«, haben wir das Programm *sqlplus.exe* (d.h. die Kommandozeile) verwendet, um die Datenbank per SQL zu erreichen. In manchen Situatio-

nen wünscht man sich jedoch eine grafische Benutzeroberfläche, weil sie für einfache Vorhaben komfortabler ist. Eclipse bietet von Haus aus eine solche grafische Benutzeroberfläche, die sich *Data Source Explorer* nennt. In diesem Abschnitt werden wir den Data Source Explorer nutzen, um einen raschen Überblick über die Tabellen `CUSTOMER` und `ITEM` der Onlineshop-Datenbank zu erhalten.

Zuletzt werden wir den Datenbankzugriff aber auch noch über ein eigenes Java-Programm realisieren.

7.1.1 Die JDBC-Treibertypen

Für die gängigsten Datenbanken wird der Treiber durch den Hersteller selbst zur Verfügung gestellt. Aber egal für welche Datenbank der Treiber produziert wurde, es handelt sich immer um einen der nachfolgend genannten vier Treibertypen.

JDBC-Treibertyp I (JDBC-ODBC-Bridge)

Ein JDBC-Treiber vom Typ I überbrückt die Verbindung zur Datenbank, indem zusätzlich eine proprietäre Schnittstelle namens *Open Database Connectivity* (ODBC) zwischengeschaltet wird. Die ODBC-Schnittstelle ist typischerweise in der Programmiersprache C geschrieben und wurde ursprünglich nur auf einem Windows-Betriebssystem über DLL-Dateien zur Verfügung gestellt. Heutzutage existieren ODBC-Treiber auch für Unix-Betriebssysteme, allerdings sind sie nur sehr wenig verbreitet.

Häufig wird dieser Verbindungstyp auch *JDBC-ODBC-Bridge* genannt. Die JDBC-ODBC-Bridge ist für professionelle Java-EE-Anwendungen nicht geeignet, da der Umweg über ODBC zusätzliche Ressourcen verschwendet und die Möglichkeiten auf die des proprietären ODBC-Treibers beschränkt sind. Außerdem würde man in dem Java-EE-Projekt voraussetzen, dass eine JDBC-ODBC-Bridge auf dem Rechner des Java EE Servers installiert ist, wovon man bei einem Unix-Betriebssystem jedoch kaum ausgehen kann.

JDBC-Treibertyp II (Native API Driver)

Der JDBC-Treibertyp II ist auch nicht direkt an der Datenbank angekoppelt, sondern nutzt gleichermaßen native Binärdateien als Brücke zur Datenbank. Dieser Treiber wird häufig auch als *Native API Driver* bezeichnet. Der Treibertyp II lässt schnellere Zugriffe zu, als die JDBC-ODBC-Bridge erlaubt. Dennoch ist auch dieser Treiber problematisch, da er ebenso betriebssystemabhängig ist.

JDBC-Treibertyp III (Network Protocol Driver)

Auch der JDBC-Treibertyp III stellt keine direkte Verbindung zur Datenbank her, denn er nutzt Netzwerk-Sockets, um auf eine Middleware zuzugreifen, die den tatsächlichen Datenbankzugriff realisiert. Dieser Treibertyp wird auch als *Network Protocol Driver* bezeichnet.

Der Vorteil des JDBC-Treibertyps III gegenüber den Typen I und II ist, dass er aus reinem Java-Code besteht und deshalb keine proprietäre Installation erfordert.

JDBC-Treibertyp IV (Pure Java Driver)

Ein JDBC-Treiber vom Typ IV wird auch *Pure Java Driver* genannt. Genauso wie Typ III besteht auch der Pure Java Driver aus reinem Java-Code. Der Vorteil des Typ-IV-Treibers ist jedoch, dass er eine unmittelbare Verbindung zur Datenbank herstellt, indem er direkt an die Netzwerkschnittstelle des Datenbankverwaltungssystems ankoppelt. Weitere Software oder Treiber sind nicht erforderlich.

7.1.2 Der JDBC-Treiber für die Oracle Database

Der Hersteller der *Oracle Database* stellt zwei Treiber für die Verwendung von SQL-Anweisungen zur Verfügung, nämlich einen vom Treibertyp II und einen vom Treibertyp IV. (Ein dritter Treiber namens *KPRB Driver* ist für das Handling von Stored Procedures und Triggern vorgesehen. Auf diesen Treiber werden wir nicht eingehen, da er im Rahmen der Java-EE-Entwicklung kaum genutzt wird.) Schauen wir uns nun die beiden Treiber für die Oracle Database genauer an.

Der OCI-Treiber (Treibertyp II)

Der OCI-Treiber ist ein JDBC-Treiber vom Typ II, der eine vorinstallierte Oracle-Client-Software nutzt, um sich mit der Datenbank zu verbinden. Oracle befürwortet den Einsatz dieses Treibers nur in speziellen Fällen, wenn die besonderen Eigenschaften eines Network Protocol Drivers benötigt werden.

Der Oracle Thin Driver (Treibertyp IV)

Der *Oracle Thin Driver* ist ein JDBC-Treiber vom Typ IV. Oracle empfiehlt die Verwendung dieses Treibers, der bereits bei der OracleXE-Installation im Verzeichnis *[ORACLE]/app/oracle/product/11.2.0/server/jdbc/lib* mitgeliefert wird. In diesem Verzeichnis werden Sie die Dateien *ojdbc5.jar*, *ojdbc6.jar* und *ojdbc6_g.jar* vorfinden. Alle produktiven Java-Anwendungen ab der JDK 6 sollten die Datei *ojdbc6.jar* einsetzen.

Der Treiber *ojdbc6_g.jar* wurde mit dem Debug-Optionsparameter (`javac -g`) kompiliert. Dadurch enthalten die JDBC-Klassen zusätzliche Debug-Informationen. Obwohl man den Treiber *ojdbc6_g.jar* in einem produktiven System aus Performance-Gründen nicht einsetzen wird, ist er in der Entwicklungsphase sehr nützlich.

7.1.3 Die Driver Definition in Eclipse

Um den Oracle Thin Driver in einer Java-Anwendung zu verwenden, reicht es eigentlich aus, wenn er in den Klassenpfad eingebunden wird. Diesen Weg werden wir aber nicht gehen.

Denn innerhalb von Eclipse gibt es auch noch die Möglichkeit, den Treiber als *Driver Definition* festzulegen. Dies ist ganz praktisch, da er hierdurch in allen Projekten der gesamten Eclipse-IDE über seinen Namen eingebunden werden kann.

Um den Treiber als Driver Definition zu setzen, klicken Sie im Hauptfenster auf WINDOW • PREFERENCES. Auf der linken Seite des PREFERENCES-Fensters öffnen Sie den Ordner DATA MANAGEMENT • CONNECTIVITY • DRIVER DEFINITIONS. Anschließend können Sie auf der rechten Seite die vorhandenen Driver-Definitionen nach Herstellern filtern. Wenn Sie den Eintrag ORACLE selektieren, sollte die Ansicht leer sein (siehe Abbildung 7.1).

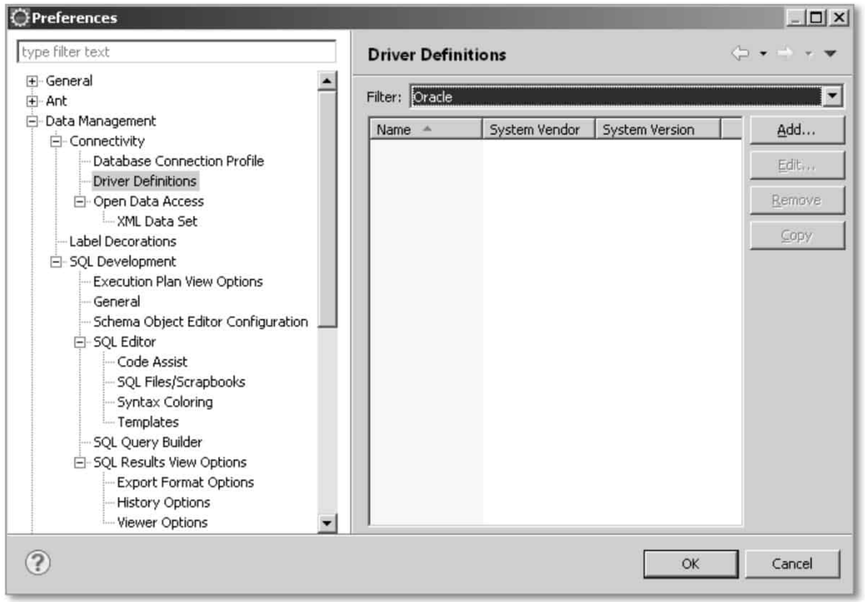


Abbildung 7.1 Die »Driver Definition« im »Preferences«-Fenster

Wenn Sie nun auf der rechten Seite auf ADD klicken, öffnet sich ein neues Fenster, in dem Sie die *Driver Definition* erstellen (siehe Abbildung 7.2). Unter den Herstellern wählen Sie wieder ORACLE aus. Im mittleren Bereich erscheinen die zur Verfügung stehenden *Driver Templates*, unter denen Sie den *Oracle Thin Driver* mit der Versionsnummer 11 auswählen.

Als Nächstes wechseln Sie auf die Registerkarte JAR LIST (siehe Abbildung 7.3). Entfernen Sie dort mit REMOVE JAR/ZIP vorhandene Einträge, und klicken Sie auf ADD JAR/ZIP, um die Datei *ojdbc6_g.jar* im Verzeichnis *[ORACLE]/app/oracle/product/11.2.0/server/jdbc/lib* zu suchen. Bestätigen Sie Ihre Auswahl mit OK.

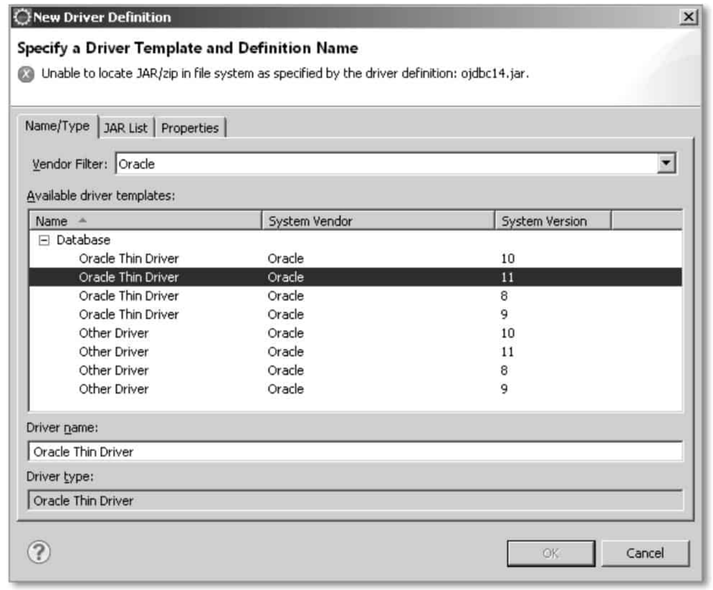


Abbildung 7.2 Die Auswahl des »Driver Templates«

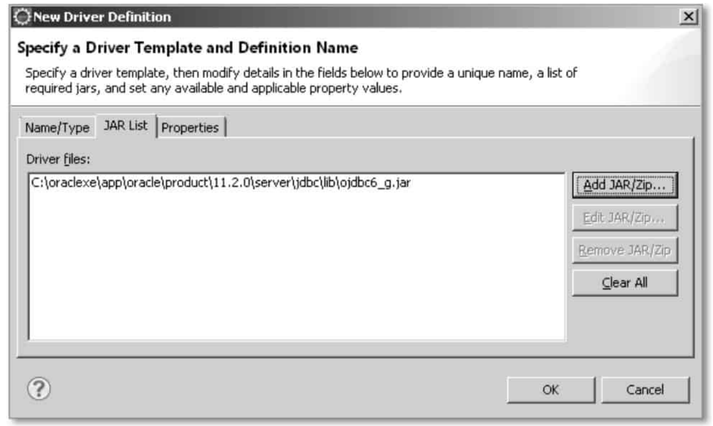


Abbildung 7.3 Die JAR-Liste für die Driver Definition

7.1.4 Der »Data Source Explorer« in Eclipse

Wie eingangs angekündigt, werden wir nun einen direkten Datenbankzugriff über den *Data Source Explorer* von Eclipse einrichten.

Der *Data Source Explorer* ist eine Benutzerschnittstelle, mit der Sie auf Datenbanken zugreifen, SQL-Anweisungen senden, Ergebnismengen anschauen können u.v.m.

Im Data Source Explorer wird eine Liste vorkonfigurierter Verbindungsprofile vorgehalten. Nachdem der Oracle-Treiber als *Driver Definition* eingebunden worden ist, kann die Onlineshop-Datenbank im Data Source Explorer als Verbindungsprofil eingerichtet werden.

Wenn der Data Source Explorer in Ihrer Window Perspective innerhalb von Eclipse nicht bereits geöffnet ist, können Sie ihn als View hinzufügen, indem Sie im Hauptmenü von Eclipse auf WINDOW • SHOW VIEW klicken und ihn unter DATA MANAGEMENT auswählen (siehe Abbildung 7.4).

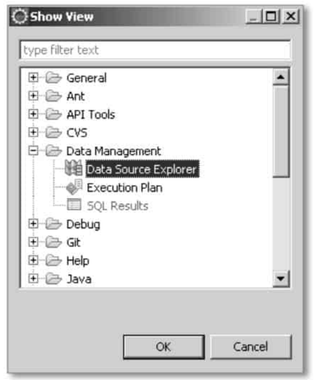


Abbildung 7.4 Die View für den DataSource Explorer

Innerhalb des Data Source Explorers öffnen Sie mit der rechten Maustaste das Kontextmenü und klicken dort auf NEW, um ein neues Verbindungsprofil (*Connection Profile*) einzurichten (siehe Abbildung 7.5).

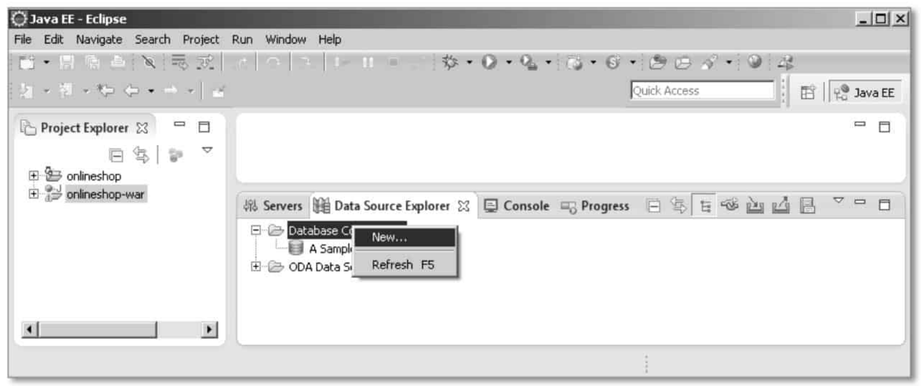


Abbildung 7.5 Mit »New« ein Connection Profile erstellen

Danach erscheint ein Fenster, in dem Sie den Verbindungsprofil-Typ (unter CONNECTION PROFILE TYPES) auswählen und ihm einen Namen geben (siehe Abbildung 7.6). Nennen Sie das Verbindungsprofil »Onlineshop«. Die Beschreibung im unteren Teil des Fensters ist optional.

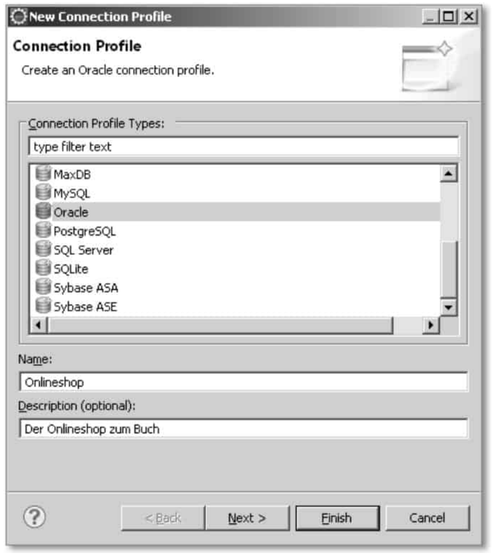


Abbildung 7.6 Die Auswahl eines Verbindungsprofil-Typs im Data Source Explorer

Mit NEXT gelangen Sie zum nächsten Fenster, in dem Sie die Details für das Verbindungsprofil eingeben. Ganz oben müssen Sie den *Oracle Thin Driver* selektieren. Darüber hinaus setzen Sie noch die Daten aus Tabelle 7.1. Ihre Bedeutung wurde in Kapitel 6, »Die relationale Datenbank«, erläutert.

Name	Wert
SID	XE
Host	localhost
Port number	1521
User name	onlineshop_user
Password	geheim_123

Tabelle 7.1 Daten zum Anlegen des Verbindungsprofils

Mit einem Klick auf TEST CONNECTION stellen Sie fest, ob die Einstellungen des Verbindungsprofils korrekt sind und Sie sich mit der Datenbank verbinden können.

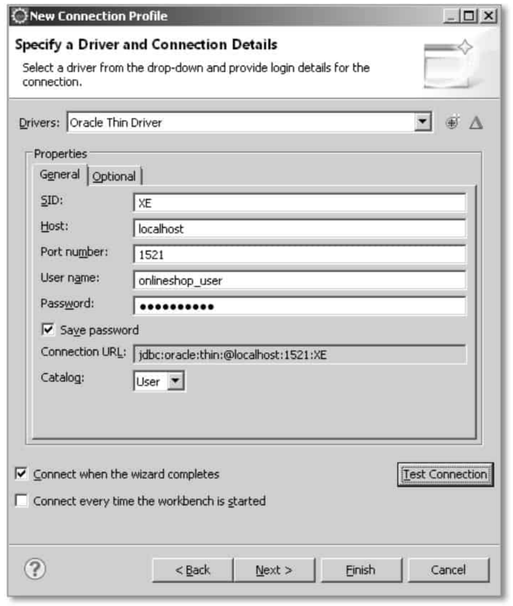


Abbildung 7.7 Die Erstellung eines »Connection Profile«

Wenn Sie auf den Button NEXT klicken, wird Ihnen eine Zusammenfassung zu dem Verbindungsprofil gezeigt (siehe Abbildung 7.8). Bestätigen Sie abschließend die Details mit FINISH.

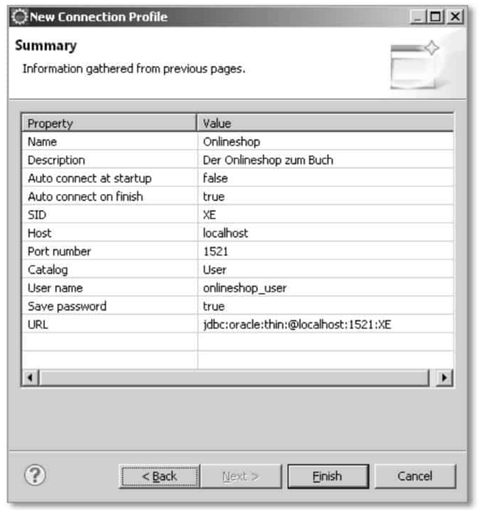


Abbildung 7.8 Die Zusammenfassung zu dem »Connection Profile«

In der Baumstruktur des Data Source Explorers sollte anschließend ein Ordner mit dem Namen Onlineshop zu finden sein (siehe Abbildung 7.9). Öffnen Sie die Baumstruktur bis zum Ordner XE • SCHEMAS • ONLINESHOP • TABLES, und überprüfen Sie dort die Tabellen CUSTOMER und ITEM. Diese Tabellen hatten wir im letzten Kapitel über ein SQL-Skript angelegt.

Daraufhin klicken Sie auf der rechten Seite des Data Source Explorers auf das Scrapbook-Symbol, um mit dem Scrapbook SQL-Anweisungen an die Datenbank zu senden. Dieses Symbol ist in Abbildung 7.9 mit einem Kreis markiert.

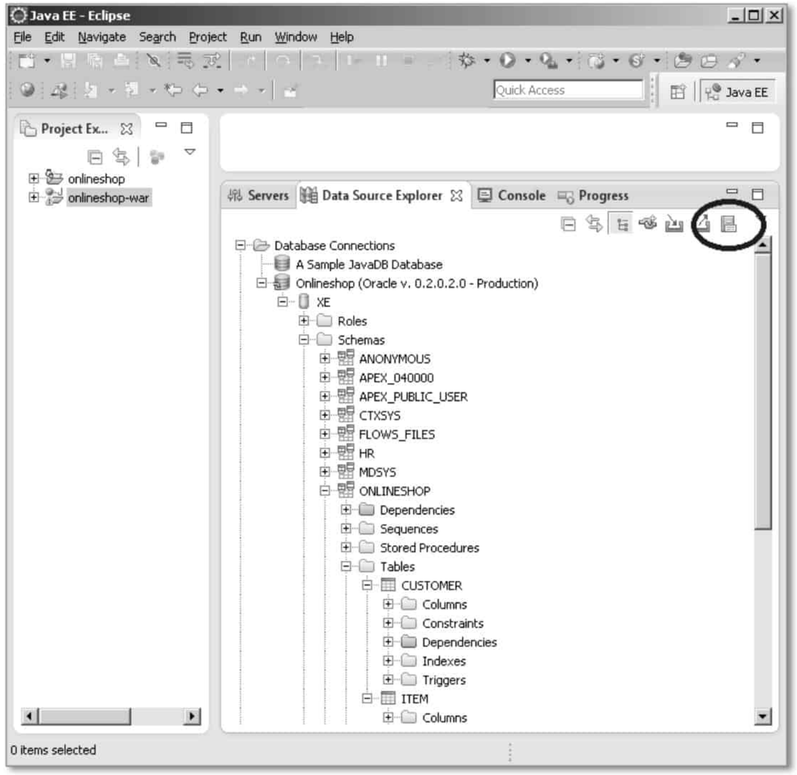


Abbildung 7.9 Rechts oben befindet sich das Symbol für das Scrapbook.

Geben Sie im Scrapbook die SQL-Anweisung SELECT * FROM ONLINESHOP.CUSTOMER; ein. Markieren Sie die SQL-Anweisung. Klicken Sie mit der rechten Maustaste auf die SQL-Anweisung, damit sich das Kontextmenü öffnet. Im Kontextmenü wählen Sie den Menüeintrag EXECUTE SELECTED TEXT aus. Daraufhin sollte rechts unten der Status der Kommandoausführung angezeigt werden. Klicken Sie dort (siehe Abbildung 7.10) auf den Reiter RESULT, um die Daten der Tabelle CUSTOMER zu betrachten.

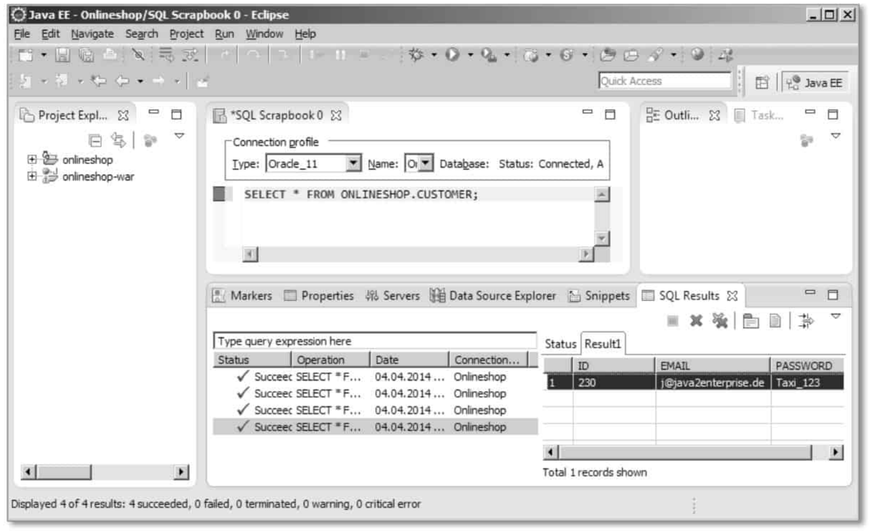


Abbildung 7.10 Die Daten der Tabelle »CUSTOMER«

7.1.5 Die Driver Definition im Java-Projekt verwenden

Damit die *Driver Definition* nun auch im Eclipse-Projekt eingebunden wird, öffnen Sie im PROPERTIES-Fenster Ihres Java-Projekts den Ordner JAVA BUILD PATH (siehe Abbildung 7.11).

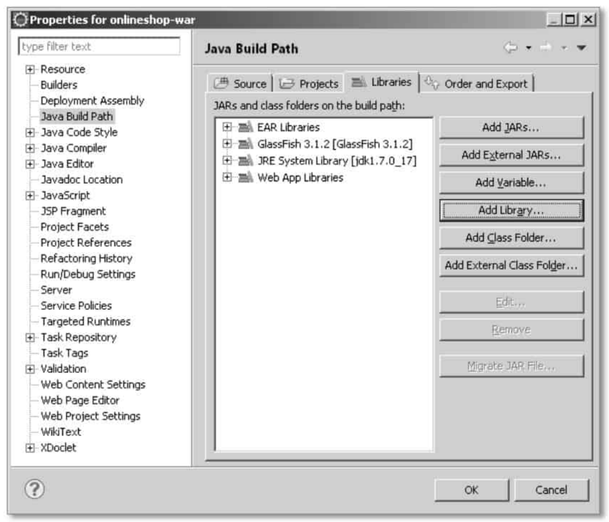


Abbildung 7.11 Der »Java Build Path« in den Properties der Webanwendung

Dann wählen Sie auf der rechten Seite das Register LIBRARIES aus und klicken dort auf ADD LIBRARY. Im ADD LIBRARY-Fenster selektieren Sie CONNECTIVITY DRIVER DEFINITION (siehe Abbildung 7.12).

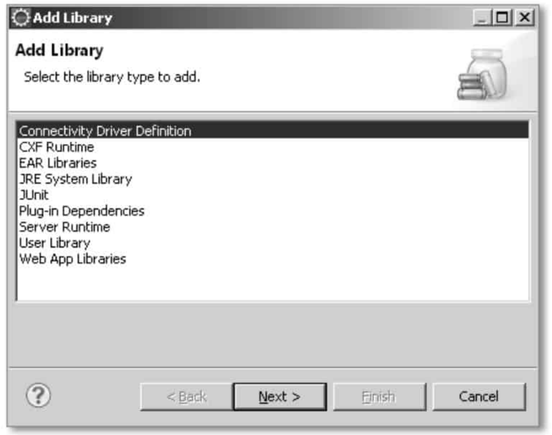


Abbildung 7.12 Das Hinzufügen einer »Connectivity Driver Definition«

Mit NEXT gelangen Sie in ein Fenster, in dem Sie die Driver Definition für die Oracle-Datenbank hinzufügen können (siehe Abbildung 7.13).

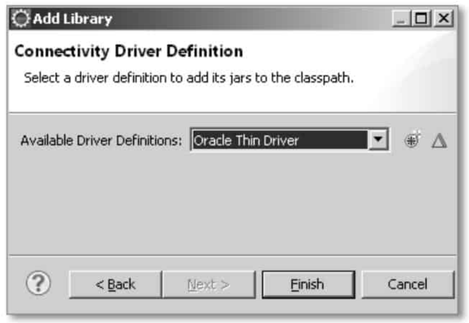


Abbildung 7.13 Die Auswahl des »Oracle Thin Drivers«

Klicken Sie abschließend auf FINISH, damit die Driver Definition als Library hinzugefügt wird.

7.1.6 Der Erstellung einer Datenbankverbindung

Die Grundvoraussetzung, um Datenbankoperationen von Java aus durchzuführen, ist die Erstellung einer Datenbankverbindung. Im folgenden Beispiel werden wir eine kleine Java-

SE-Anwendung schreiben, die eine Datenbankverbindung aufbaut und auf der Konsole ausgibt, ob der Zugriff auf die Datenbank erfolgreich war oder nicht.

javax.sql.DriverManager

Die älteste Klasse der JDBC-API, mit der eine Datenbankverbindung erstellt werden kann, war bereits in der ersten Version der JDBC-API im Package `java.sql` enthalten. Sie nennt sich `DriverManager`. In diesem Abschnitt wird diese herkömmliche Variante in einem Beispiel gezeigt. Betrachten Sie dieses Beispiel nicht als Best Practice: Die Klasse ist veraltet, wird hier aber der Vollständigkeit halber aufgeführt. Heutzutage verwendet man in Java-EE-Anwendungen (aber auch in Java-SE-Anwendungen) die Klasse `javax.sql.DataSource`, bei der die Verbindung über einen JNDI-Lookup ermöglicht wird. Diese Klasse wird im nächsten Abschnitt behandelt, wenn wir im Java EE Server eine Datenquelle über JNDI zur Verfügung haben.

Aber auch im `DriverManager`-Beispiel werden wir die Verbindungsinformationen (wie den Namen der Treiberklasse, die JDBC-URL, den Datenbankbenutzernamen und das Datenbankpasswort) in eine separate Datei namens *C:\onlineshop.properties* auslagern. Dadurch ermöglichen wir, dass Verbindungsdaten für die Entwicklungs-, Test- und Produktionsumgebung außerhalb des Programms ausgetauscht werden können. Die ausgelagerte Textdatei werden wir als Property-Datei anlegen, damit sie durch die Klasse `java.util.Properties` ausgelesen werden kann.

```
driver=oracle.jdbc.OracleDriver
url=jdbc:oracle:thin:@localhost:1521/XE
username=onlineshop_user
password=geheim_123
```

Listing 7.1 c:\onlineshop.properties

Im Eclipse-Projekt werden wir nun eine Java-Klasse erstellen, die sich `DataAccess` nennt. Die Klasse `DataAccess` wird eine Datenbankverbindung aufbauen und die Gültigkeit der Verbindung prüfen.

Nachdem die ausgelagerte Datei eingelesen worden ist, wird zunächst die Haupttreiberklasse des JDBC- Treibers instanziiert. Im Beispiel mit der Oracle Database ist das die Klasse `oracle.jdbc.OracleDriver`. Um gemäß der Java-EE-Connector-Architektur die Datenbank als austauschbaren Informationsservice zu halten, wird der Name des Treibers nicht hartkodiert, sondern ganz generisch mit der Methode `Class.forName(String Treibername)` aufgerufen.

Anschließend wird eine Datenbankverbindung mithilfe der Methode `getConnection()` geholt und mit `isValid()` geprüft, ob die Datenbankverbindung gültig ist. Indem wir der Methode `isValid()` den Wert `10` als Parameter mitgeben, sorgen wir dafür, dass die Verbindung nur valide ist, wenn sie innerhalb von 10 Sekunden dingfest gemacht werden konnte. Für die Ausgabe auf die Konsole werden wir die Klasse `java.util.Logger` einsetzen.


```
package de.java2enterprise.onlineshop;

import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;
import java.util.logging.Logger;

public class DataAccess {
    public static void main(String[] args)
            throws Exception {
        final Logger logger =
            Logger.getLogger(
                DataAccess.class.getName());

        final Properties p = new Properties();
        p.load(new FileInputStream(
                "c:/onlineshop.properties"));
        Class.forName(p.getProperty("driver"));
        final Connection con =
            DriverManager.getConnection(
                p.getProperty("url"),
                p.getProperty("username"),
                p.getProperty("password"));
        if(con.isValid(10)) {
            logger.info("Connected!");
        }
        logger.info("Closing connection!");
        con.close();
        logger.info("Program finished!");
    }
}
```

Listing 7.2 »DataAccess.java«

Nach der Ausführung des Programms sollte auf der Konsole eine ähnliche Ausgabe wie diese erscheinen:

```
Apr 04, 2014 9:20:56 AM de.java2enterprise.onlineshop.DataAccess main
Information: Connected!
Apr 04, 2014 9:20:56 AM de.java2enterprise.onlineshop.DataAccess main
Information: Closing connection!
Apr 04, 2014 9:20:56 AM de.java2enterprise.onlineshop.DataAccess main
Information: Program finished!
```

javax.sql.DataSource

Obwohl auf die oben gezeigte Weise eine Datenbankverbindung erzeugt werden kann, gilt sie in Java-EE-Anwendungen als »deprecated« und wird nicht mehr benutzt. Der Grund dafür ist, dass Java-EE-Anwendungen Datenbankverbindungen nicht direkt, sondern über den Java EE Server erstellen sollen. Denn die Java-EE-Spezifikation fordert eine weitere Entkopplung, die die Beschaffung von Datenbankverbindungen kapselt und abstrahiert. Man fasste diese Anforderung unter der allgemeinen Bezeichnung *DataSource* (zu Deutsch *Datenquelle*) zusammen. Die DataSource wird vom Administrator des Java- EE Servers über einen JNDI-Namen zur Verfügung gestellt. Innerhalb der Java-EE-Anwendung kann anschließend über den JNDI-Namen auf die DataSource Bezug genommen werden. Der Rückgabewert des JNDI-Lookups ist vom Typ `javax.sql.DataSource`.

javax.sql.ConnectionPoolDataSource

Eine weitere Anforderung erwuchs aus dem Problem, dass Java-EE-Anwendungen häufig sehr viele Datenbankverbindungen gleichzeitig benötigen. Weil die Erzeugung einer Datenbankverbindung verhältnismäßig kostspielig ist, musste eine Möglichkeit geschaffen werden, sie durch eine Art Recycling wiederzuverwenden. Man entwickelte sogenannte *Connection-Pools*. Durch einen Connection-Pool wird eine bestimmte Menge an Datenbankverbindungen wie in einem Recycling-Mechanismus verwaltet. Eine DataSource, die einen Connection-Pool einsetzt, wird durch das Interface `javax.sql.ConnectionPoolDataSource` repräsentiert.

javax.sql.XADataSource

Zusätzlich zu den beiden genannten DataSource-Typen erfordert ein Java-EE-konformer Server, dass auf unterschiedliche Systeme gemäß des X/Open-XA-Standards zugegriffen wird. Dabei werden die CRUD-Anweisungen (CREATE, READ, UPDATE und DELETE) innerhalb einer einzigen globalen Transaktion über verteilte Systeme geklammert. Die JDBC-Repräsentation für diese Fähigkeit wird durch das Interface `javax.sql.XADataSource` gekennzeichnet.

Bei einer globalen Transaktion über den X/Open-XA-Standard werden die CRUD-Operationen in den betroffenen Datenquellen nur dann wirklich wirksam, wenn alle Systeme mit der Durchführung einverstanden sind. Auf diese Weise wird gewährleistet, dass entweder bei allen oder bei gar keiner Datenquelle die CRUD-Operationen durchgeführt werden.

Die Durchführung der globalen Transaktion geschieht dabei mithilfe des sogenannten 2-Phasen-Commit-Protokolls. Wenn bei dem 2-Phasen-Commit-Protokoll ein Commit angewiesen wird, versendet der *Transaction Manager* eine PREPARE_TO_COMMIT-Anfrage an alle betroffenen Datenquellen. Daraufhin reagieren die Datenquellen entweder mit einem PREPARED oder mit einem NO. Wenn alle Datenquellen mit einem PREPARED geantwortet haben, sendet der Transaction Manager ein COMMIT. Ansonsten versendet er ein ABORT, was bei den Datenquellen zu einem ROLLBACK führt.

7.2 Der Zugriff über den Java EE Server

In dem letzten Programmierbeispiel haben wir direkt von einer Java-Standalone-Anwendung aus mit der Datenbank kommuniziert. Nun wird gezeigt, wie die Datenbankverbindung durch den Java EE Server besorgt werden kann. Dies ist praktisch, denn dadurch ist die Java-EE-Anwendung von dieser Aufgabe entlastet, und gleichzeitig erzielen wir eine Loskopplung der Anwendung von der Datenbank.

Die gängige Praxis ist, dass der Java EE Server die Datenbankverbindung als DataSource über einen Namensdienst zu Verfügung stellt. Hierfür muss der GlassFish-Server konfiguriert werden. Vor der Konfiguration muss aber zunächst auch noch der JDBC-Treiber in das Unterverzeichnis *c:/glassfish4/glassfish/domains/domain1/lib* kopiert werden (siehe Abbildung 7.14).

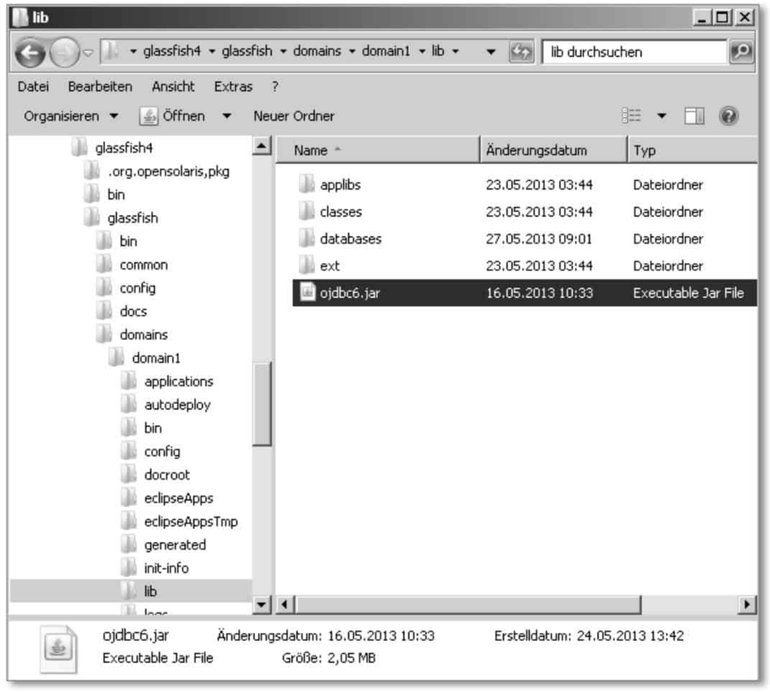


Abbildung 7.14 Der JDBC-Treiber im »lib«-Verzeichnis der Domäne »domain1«

Wenn der GlassFish-Server auf Ihrem Rechner bereits läuft, sollten Sie ihn nun in jedem Fall erneut starten, denn ansonsten erfährt er nicht, dass der JDBC-Treiber im */lib*-Verzeichnis vorhanden ist.

Nach dem Neustart von GlassFish rufen Sie in einem Webbrowser die Seite *http://localhost:4848* auf, um die Administrationskonsole zu öffnen.

Bei vielen GlassFish-Plug-in-Varianten ist es auch möglich, die Admin-Konsole von Eclipse aus aufzurufen. Weil wir das GlassFish-Plug-in in Eclipse installiert haben, könnte hierfür in der View SERVER ein spezieller Button mit einem Fisch-Symbol angeboten sein. Diese Option ist aber nicht in allen Plug-in-Versionen vorhanden. Deshalb werde ich auf die Erläuterung dieses Buttons verzichten. Er ist ohnehin vom Prinzip her mit dem gezeigten Weg gleichwertig.

Innerhalb der Administrationskonsole werden wir nun die Konfiguration für den ConnectionPool und die DataSource vornehmen. Dabei werden wir

1. die JVM-Optionsparameter für die Oracle Database anpassen,
2. einen JDBC-ConnectionPool erstellen und
3. eine DataSource erzeugen (Vorsicht: Eine DataSource wird beim GlassFish-Server als *JDBC Resource* bezeichnet.)

7.2.1 Die JVM-Optionsparameter für den Java EE Server

Im Webbrowser sollte nun die Administrationskonsole des GlassFish-Servers zu sehen sein. Klicken Sie dort auf der linken Seite auf KONFIGURATIONEN • SERVER-CONFIG • JVM EINSTELLUNGEN. Auf der rechten Seite rufen Sie den Reiter JVM-OPTIONEN auf und fügen die beiden Optionsparameter hinzu (siehe Abbildung 7.15).

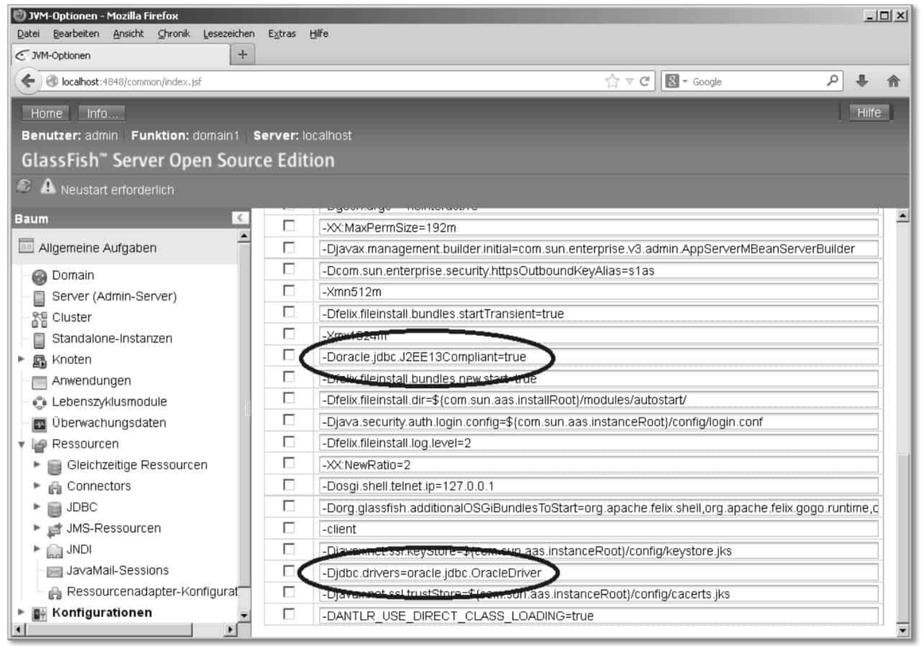


Abbildung 7.15 Die JVM-Optionsparameter für den JDBC-Treiber

Beim GlassFish-Server ist per Default der Optionsparameter

```
-Djdbc.drivers=org.apache.derby.jdbc.ClientDriver
```

gesetzt, da die Datenbank *Derby* per Default bei GlassFish vorinstalliert ist. Da für dieses Buch die Oracle Database verwendet wird, ändern wir diesen Optionsparameter wie folgt:

```
-Djdbc.drivers=oracle.jdbc.OracleDriver
```

Laut Oracle-Online-Dokumentation muss der Optionsparameter `-Doracle.jdbc.J2EE13Compliant=true` gesetzt werden, damit sich der Treiber gemäß dem Java-EE-Standard verhält. Auch darum sollten wir uns nun kümmern.

Zuletzt bestätigen Sie die Änderungen mit einem Mausklick auf den Button SPEICHERN. Außerdem sollten Sie nun den Java EE Server erneut stoppen und wieder starten.

7.2.2 Erstellung eines JDBC-Connection Pools

Als Nächstes erstellen wir einen *Connection Pool* für die Oracle Database. In der Administrationskonsole klicken Sie auf der linken Seite auf RESSOURCEN • JDBC • JDBC-CONNECTION POOLS. Auf der rechten Seite erscheint dann eine Übersicht über die vorhandenen JDBC-Connection Pools (siehe Abbildung 7.16).



Abbildung 7.16 Die Übersicht über die Connection Pools

In dem Übersichtsfenster JDBC-Connection Pools klicken Sie auf Neu, um den Verknüpfungsnamen zu der Datenbank einzugeben.

Die Erstellung des Connection Pools wird in zwei Schritten durchgeführt.

Schritt 1

Im ersten Schritt müssen Sie im obersten Eingabefeld einen Namen vergeben (siehe Abbildung 7.17). Als Ressourcentyp stehende folgende Einträge zur Auswahl:

- `java.sql.Driver`
- `javax.sql.DataSource`
- `javax.sql.ConnectionPooledDataSource`
- `javax.sql.XADataSource`

Den Eintrag `java.sql.Driver` lassen wir mal außen vor. Stattdessen betrachten wir die drei `DataSource`-Varianten. Im letzten Abschnitt wurde der Unterschied dieser drei Varianten beschrieben, als sie als Alternative zur direkten Datenverbindung über den DriverManager aufgezeigt wurden. Gemäß dieser Beschreibung müsste man annehmen, dass der Ressourcentyp `javax.sql.DataSource` kein und nur der Ressourcentyp `javax.sql.ConnectionPooledDataSource` ein Connection-Pooling anbietet. Zumindest lässt dies der Bezeichner des Ressourcentyps vermuten.

Allerdings wurde das mit dem Oracle-Treiber anders gelöst: Um eine Datenverbindung mit ConnectionPooling auf eine einzige Datenquelle zu erhalten, empfiehlt Oracle den Typ `javax.sql.DataSource`. Diese Tatsache ist leider sehr verwirrend und hat schon zu großem Ärger bei so manchem Entwickler geführt. Wie dem auch sei, für unseren Onlineshop wäre diese Auswahl also die richtige.

Nebenbei gesagt: Wenn wir auf mehrere XA-Datenquellen in einer einzigen globalen Transaktion zugreifen wollten, müssten wir den XA-Treiber verwenden und den Eintrag `javax.sql.XADataSource` auswählen, der von Oracle ebenfalls empfohlen wird – und zwar genau dann, wenn eben das 2-Phasen-Commit-Protokoll des X/Open-XA-Standards benötigt wird.

Datenbanktreiber-Hersteller

Bei dem Eingabefeld für den Datenbanktreiber-Hersteller selektieren Sie Oracle. Ganz unten in der Checkbox Introspektieren wird ein nützliches Feature angeboten: Wenn Sie diese Checkbox anklicken, sucht GlassFish für Sie den Treibernamen heraus, der zu dem Datenbankhersteller passt. Oracle bezeichnet dieses Feature als *Introspektion*.

Mit einem Mausklick auf Weiter gelangen Sie zum zweiten Schritt.

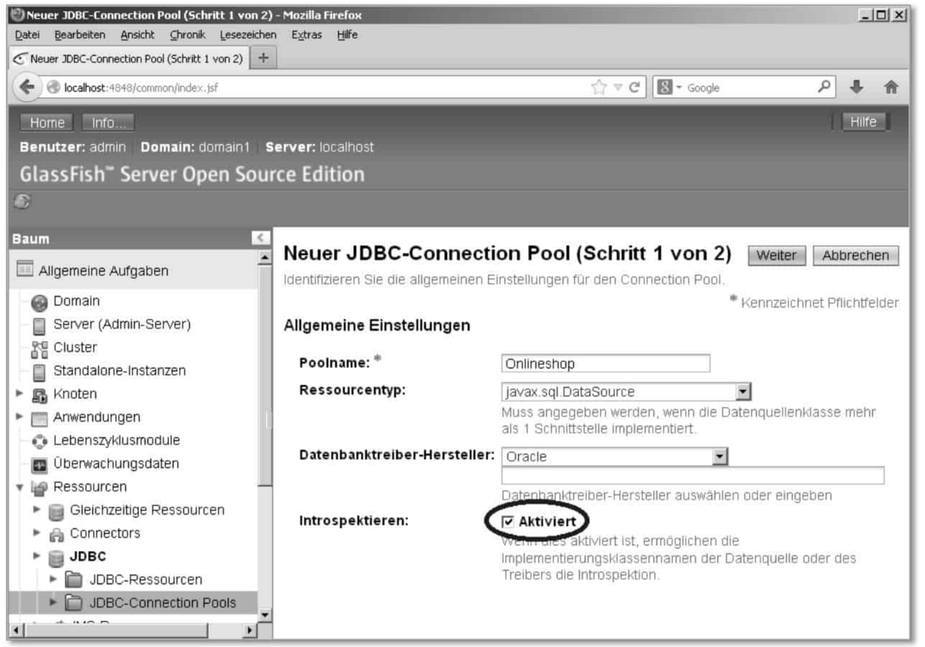


Abbildung 7.17 Der erste Schritt zur Erstellung des JDBC-Connection Pools enthält eine Checkbox für die Introspektion.

Schritt 2

Im zweiten Schritt müssen Sie zunächst den Klassennamen der Datenquelle einstellen (siehe Abbildung 7.18). Setzen Sie den Wert auf `oracle.jdbc.pool.OracleDataSource`. Wenn es sich um eine XA-Datenquelle handelt, tragen Sie `oracle.jdbc.xa.client.OracleXADataSource` ein.

Wenn bei Ihnen diese Einträge nicht angeboten werden, könnte es daran liegen, dass Sie den JDBC-Treiber nicht in das Verzeichnis *c:/glassfish4/glassfish/domains/domain1/lib* kopiert bzw. den GlassFish-Server anschließend nicht gestoppt und erneut hochgefahren haben. Wenn das der Fall ist, korrigieren Sie dies zunächst. Erst, wenn die obige Ansicht in Ihrem Webbrowser zu sehen ist, sollten Sie mit dem nächsten Schritt fortfahren.

Weiter unten haben Sie die Möglichkeit, den Connection Pool zu konfigurieren. Dabei sind vor allem die zur Verfügung stehenden Connections von Interesse. Per Default werden 8 Connections vorbereitet. Wenn diese belegt sind, erhöht der Pool seine Connections auf bis zu 32 Stück. Beide Werte werden wir nun erhöhen, denn wenn wir die Fotos der Artikel in der Datenbank speichern wollen, müssen wir für zahlreiche gleichzeitig aufkommende Anfragen gewappnet sein. Die Datenbank hatten wir hierauf bereits vorbereitet, denn sie ist nach unserer Änderung im letzten Kapitel in der Lage, etwa 300 Anfragen parallel zu beantworten. Deshalb werden wir den Minimalwert nun auf 80 und den Maximalwert auf 320 setzen (siehe Abbildung 7.19). Dies sollte im Vergleich zu der Einstellung in der Datenbank angemessen sein.



Abbildung 7.18 Der zweite Schritt für die Erstellung des JDBC-Connection Pools

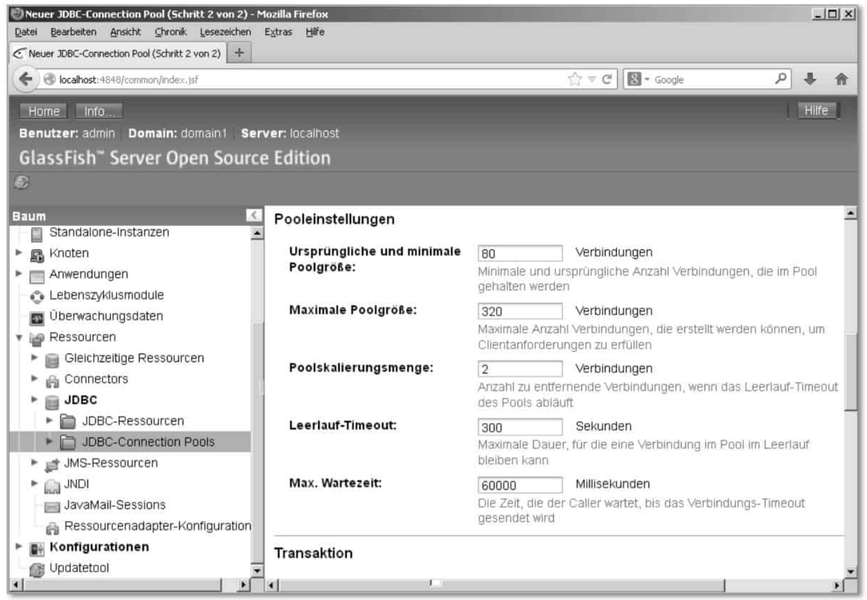


Abbildung 7.19 Die Einstellungen für den Connection Pool

Wenn Sie die Einstellungen für den Pool angepasst haben, scrollen Sie in den unteren Bereich des Fensters, wo Sie in Eingabefeldern die Verbindungsinformationen eingeben können (siehe Abbildung 7.20). Füllen Sie die Eingabefelder USER, PASSWORD und URL, wie Sie es in der Property-Datei eingetragen hatten. Vielleicht wird Ihnen eines Tages in Ihrem Java-EE-Projekt folgende Schreibweise begegnen:

```
jdbc:oracle:thin:@//localhost:1521/XE
```

Hierbei handelt es sich um eine neue Syntax, bei der der Oracle-Service verwendet wird, der auf dem Rechner installiert ist. Wenn Sie die gleiche Entwicklungsumgebung verwenden, die in diesem Buch gezeigt wird, funktioniert die neue Syntax, weshalb ich sie nachfolgend auch verwenden werde. Es könnte aber durchaus sein, dass Sie in Ihrem Arbeitsumfeld eine andere Oracle-Installation vorfinden. In diesem Fall verwenden Sie die alte JDBC-URL-Syntax:

```
url=jdbc:oracle:thin:@localhost:1521/XE
```

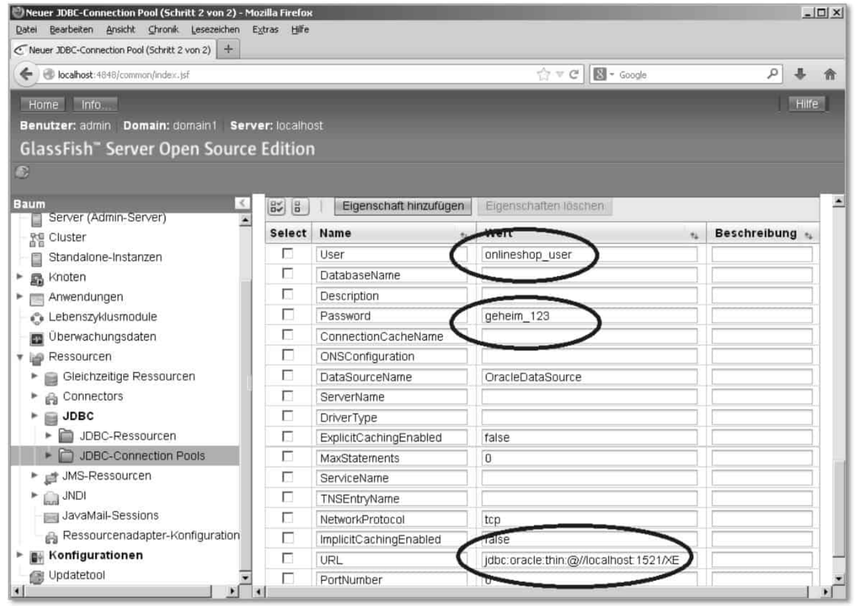


Abbildung 7.20 Die Eingabe der Verbindungsdaten

Wenn Sie auf FERTIG STELLEN klicken, wird der neue JDBC-Connection Pool erstellt. Der neue JDBC-Connection Pool sollte nun im Übersichtsfenster zu sehen sein (siehe Abbildung 7.21).

Klicken Sie im Connection-Pool-Fenster auf ONLINESHOP, und versuchen Sie im Editierfenster mit einem Klick auf PING, die Datenbankverbindung zu testen (siehe Abbildung 7.22).



Abbildung 7.21 Der neue JDBC-Connection Pool zur »Onlineshop«-Datenbank

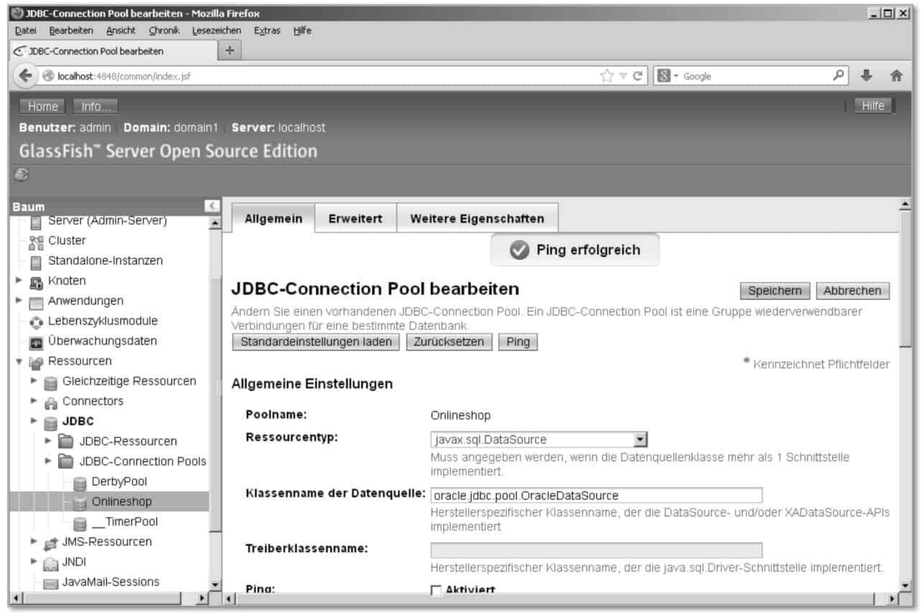


Abbildung 7.22 Der Ping zum Datenbank-Server

7.2.3 Erstellung einer »Default Data Source«

Ich habe bereits angemerkt, dass bei GlassFish der Begriff *JDBC Ressource* ein Synonym für den Begriff *DataSource* ist. Klicken Sie in der Administrationskonsole des GlassFish-Servers deshalb auf RESOURCES • JDBC • JDBC RESSOURCEN.

Auf der rechten Seite sollte danach eine Übersicht über die JDBC-Ressourcen zu sehen sein (siehe Abbildung 7.23). Wenn Sie nun eine nagelneue JDBC-Ressource erstellen wollten, müssten Sie mit der Maus auf NEW klicken. Dies wäre normalerweise auch der nächste Schritt. Wir werden in diesem Buch aber einen alternativen Weg gehen. Der Grund hierfür ist, dass die Java-EE-Spezifikation vorgesehen hat, dass ein EE Server eine sogenannte *Default Data Source* zur Verfügung stellen muss. Die Default Data Source soll per Spezifikation von einer Java-EE-Anwendung aus über den JNDI-Namen `jdbc/__default` erreichbar sein.

Das Besondere an der Default Data Source ist, dass mithilfe einer speziellen Konvention für »Default-Werte« auf die explizite Angabe der Datenquelle verzichtet werden kann.

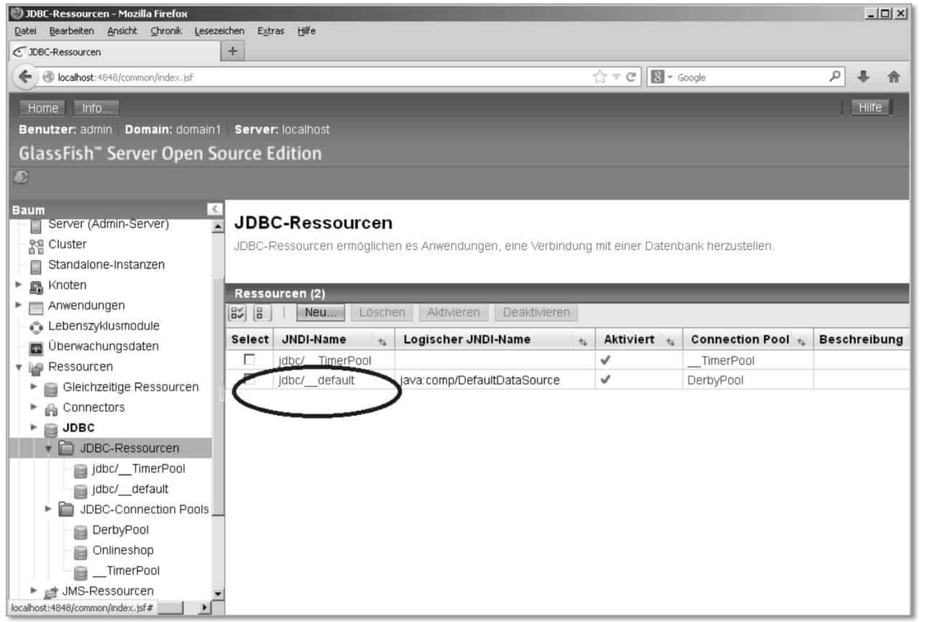


Abbildung 7.23 Die »Default Data Source«, die über »jdbc/__default« oder den logischen JNDI-Namen »java:comp/DefaultDataSource« erreichbar ist

Wie Sie an Abbildung 7.23 erkennen können, wurde die Default Data Source bei dem GlassFish Server bereits erstellt und vorkonfiguriert. Das Problem dieser vorkonfigurierten Default Data Source ist, dass sie sich leider noch nicht auf unsere Oracle Database bezieht, sondern auf einen sogenannten *Derby Pool*. Derby ist eine relationale Datenbank, die mit

dem GlassFish-Server mitgeliefert wird und vorkonfiguriert ist. Unsere Aufgabe ist es nun, die Default Data Source so abzuändern, dass sie nicht mehr auf Derby, sondern auf die Oracle Database mit dem Onlineshop referenziert. Hierfür klicken Sie auf JDBC/__DEFAULT. In dem Dialog aus Abbildung 7.24 selektieren Sie in der Combobox POOLNAME den Eintrag ONLINESHOP.

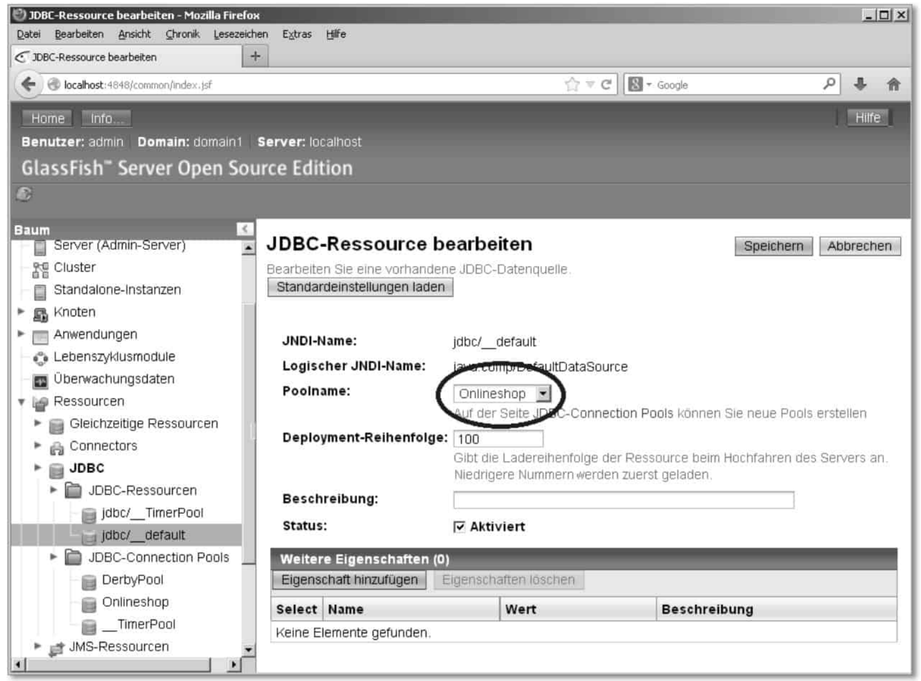


Abbildung 7.24 Die Erstellung der JDBC-Ressource

Klicken Sie anschließend auf SPEICHERN, um die Änderung der Default Data Source zu bestätigen. Schließen Sie danach das Browserfenster, und fahren Sie über `asadmin` auf der Kommandozeile den GlassFish-Server herunter. In Eclipse starten Sie zu guter Letzt den GlassFish-Server wieder. Er sollte nun für die Verwendung der JDBC-Ressource über einen Namensdienst vorbereitet sein.

7.3 Die Kernfunktionalität der JDBC-API

In diesem Abschnitt lernen Sie, wie mit der Kernfunktionalität von JDBC gearbeitet wird. Dabei werden wir die Datenbankverbindungen über den Java EE Server besorgen. Anschließend werden wir Daten speichern und wieder abfragen.

7.3.1 Die Datenbankverbindung über den Java EE Server besorgen

Im nächsten Beispiel wird die Datenbankverbindung über den GlassFish-Server getestet. Deshalb wird sich die Klasse `DataAccess` die Datenbankverbindung nun nicht mehr über den anwendungseigenen JDBC-Treiber holen, sondern über den Namensdienst des Java EE Servers. Das funktioniert allerdings auch nur, wenn die Klasse `DataAccess` vorab als Teil einer Java-EE-Anwendung im GlassFish-Server deployt wurde.

Das bedeutet auch, dass wir nun wieder ein dynamisches Webprojekt vorzugsweise als Teil eines Java-Enterprise-Projekts benötigen. Wenn Sie die Beispiele der letzten Kapitel mitprogrammiert haben, können Sie das *onlineshop-war*-Projekt weiterhin nutzen. Dort erstellen Sie ein Servlet, das die Klasse `DataAccess` aufruft. Es ruft außerdem auch die statische Methode `doLookup()` der Klasse `InitialContext` auf, um die Ressource `"jdbc/__default"` zu finden. Die Klasse `OracleDataSource` können wir allerdings hier nun nicht mehr nutzen, da der Lookup ein Objekt der Klasse `java.sql.DataSource` liefert.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;

import javax.naming.InitialContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

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

    private DataSource ds;

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

        final PrintWriter writer =
                response.getWriter();
        response.setContentType(
                "text/html;charset=UTF-8");
```

```
            writer.println("<!DOCTYPE html>");
            writer.println("<html><body>");
        try {
            ds = (DataSource)
                InitialContext.doLookup(
                "jdbc/__default");
            final Connection con = ds.getConnection();
            if(con.isValid(10)) {
                writer.println("<BR>Connected!");
            }
            con.close();
        } catch (Exception ex) {
            writer.println(ex.getMessage());
        }
            writer.println(
            "<BR>Test finished!</body></html>");
    }
}
```

Listing 7.3 »TestServlet.java«

Beachten Sie, dass Sie nun den JDBC-Treiber im Java-Build-Path (in den Preferences des Webprojekts) nicht mehr benötigen. Führen Sie das Programm aus. Wenn Sie das Servlet im dynamischen Webprojekt *onlineshop-war* programmiert haben, rufen Sie im Webbrowser die Adresse *http://localhost:8080/onlineshop-war/test* auf.

7.3.2 Die Datenquelle durch den Web-Container injizieren lassen

In diesem Abschnitt wird gezeigt, wie der Quelltext der Controller-Klasse mithilfe des sogenannten Injizierens vereinfacht werden kann. Dabei wird der JNDI-Lookup nicht mehr in der Controller-Klasse manuell programmiert, sondern vom Web-Container automatisch zugewiesen.

Um dem Web-Container mitzuteilen, welche Objektvariable er initialisieren soll, können Konfigurationsdateien oder Annotations verwendet werden. Mittlerweile werden meistens Letztere eingesetzt. Im folgenden Listing sehen Sie, wie der Objektvariablen `dataSource` die Datenquelle der Default-Data-Source injiziert wird. Dabei verwenden wir die Annotation `@Resource`:

```
...
@Resource(name="jdbc/__default")
private DataSource ds;
...
```

Listing 7.4 »TestServlet.java«

Ein besonderer Vorteil hierbei ist, dass eine Entkopplung stattfindet. In der Klasse `TestServlet.java` wird nicht festgelegt, woher die Daten kommen. Wenn wir die Klasse in einem anderen Kontext verwenden, würde sie deshalb gleichermaßen funktionieren. Einzig und allein der EE Server entscheidet darüber, ob es sich bei dem Ursprung der Daten um eine Oracle Database, eine MySQL-Datenbank oder gar um ein ganz anderes *Enterprise Information System* (EIS) handelt. Hiervon bekommt die Controller-Klasse aber nichts mit und muss bei einer Änderung auch nicht zwingend angepasst werden.

Das »Injizieren« ist also eine automatische Initialisierung, die der Container vornimmt. Der Fachbegriff hierfür ist *Dependency Injection*. Die Grundidee der Dependency Injection entstand durch die Möglichkeiten der *Reflection API*. Weil der Container per Reflexion in die Java-Klasse hineinschauen und sie manipulieren kann, ist er auch in der Lage, Objektvariablen nachträglich zu initialisieren.

Eigentlich ist die Dependency Injection (oder – wie sie vormals hieß – die *Inversion of Control*) keine Erfindung von Sun Microsystems gewesen, sondern eine Technologie, die durch das Spring-Framework von Rod Johnson im Jahre 2002 bekannt wurde. Später wurde sie mithilfe von Gavin King (dem Hauptentwickler der Hibernate- und JBoss-Seam-Frameworks) in den Java-EE-Standard übernommen (weitere Informationen hierzu finden Sie in Kapitel 1, »Überblick«).

Die Angabe der Datenquelle einfach weglassen

Im nächsten Schritt werden wir die »Default«-Konvention nutzen, um auch auf die Angabe des JNDI-Namens zu verzichten. Der Quelltext aus dem obigen Beispiel funktioniert gleichermaßen, wenn Sie die in Klammern gesetzte Angabe `name="jdbc/__default"` entfernen:

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;

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

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

```
@Resource
private DataSource ds;

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

    final PrintWriter writer =
            response.getWriter();
    response.setContentType(
            "text/html;charset=UTF-8");
        writer.println("<!DOCTYPE html>");
        writer.println("<html><body>");
    try {
        final Connection con = ds.getConnection();
        if(con.isValid(10)) {
            writer.println("<BR>Connected!");
        }
        con.close();
    } catch (Exception ex) {
        writer.println(ex.getMessage());
        }
            writer.println(
            "<BR>Test finished!</body></html>");
    }
}
```

Listing 7.5 »TestServlet.java«

7.3.3 Das Absetzen einer SQL-Anweisung

Die JDBC-Kernfunktionalität bietet mit der Klasse `java.sql.Statement` eine einfache Möglichkeit an, wie mit der Datenbank per SQL kommuniziert werden kann. Ihre wichtigsten Methoden nennen sich `executeUpdate()` und `executeQuery()`, denn in der Regel wird mit einer SQL-Anweisung die Datenbank aktualisiert (`INSERT`, `UPDATE` und `DELETE`) oder abgefragt (`SELECT`). Wenn der Zweck der SQL-Anweisung noch nicht feststeht, kann die `execute()`-Methode verwendet werden.

Im folgenden Listing wird in einer Methode mit dem Namen `persist()` ein `INSERT`-Statement erzeugt. Über die Methode `executeUpdate()` wird die SQL-Anweisung an die Datenbank versendet. Beachten Sie bei dem Beispiel, dass die Datenbank den Primärschlüssel automatisch setzt.


```
...
public void persist(
        String _email,
        String _password)
        throws Exception {
    Connection con = ds.getConnection();
    Statement stmt = con.createStatement();
    stmt.executeUpdate(
    "INSERT INTO onlineshop.customer (" +
        "email, " +
        "password" +
    ") VALUES (" +
        "'" + _email + "', " +
        "'" + _password + "') "
    );
    con.close();
}
...
```

Listing 7.6 »RegisterServlet.java«

7.4 Das MVC-Entwurfsmuster einsetzen

In den Kapiteln über Java-EE-Web-Technologien bin ich bereits auf die Verwendung des MVC-Entwurfsmusters eingegangen. Auch in diesem Abschnitt werden wir dieses Entwurfsmuster noch einmal aufgreifen, denn mithilfe von JDBC können wir die JavaBean-Klassen des Modells nun persistieren.

Hier noch mal ein kurze Wiederholung: JavaBeans sind im Prinzip wiederverwendbare Komponenten, die bestimmten Anforderungen entsprechen müssen. Zum Beispiel müssen sie einen parameterlosen Konstruktor zur Verfügung stellen und serialisierbar sein. Außerdem sollten sie sogenannte Properties enthalten. Properties sind private Objektvariablen, die über öffentliche Setter- und Getter-Methoden nach außen zur Verfügung stehen.

Wenn eine JavaBean als Gegenstück zu einer Tabelle einer relationalen Datenbank betrachtet wird, spricht man eher von einer *Entity* und ihre Properties bezeichnet man als *Attribute* der Entity.

Das Besondere an einer Entity ist, dass der Begriff gleichzeitig auch besagt,

- dass sie zwingend über einen Schlüssel eindeutig identifizierbar ist und
- dass sie persistiert (das heißt: dauerhaft gespeichert) werden kann und somit einen gewissen Lebenszyklus in Bezug auf die Datenhaltung aufweist.

7.4.1 Die Speicherung einer Entity

In einem Programmierbeispiel schaffen wir für die Kunden des Onlineshops eine Möglichkeit, sich zu registrieren. Das Schema für unser Beispiel sehen Sie in Abbildung 7.25.

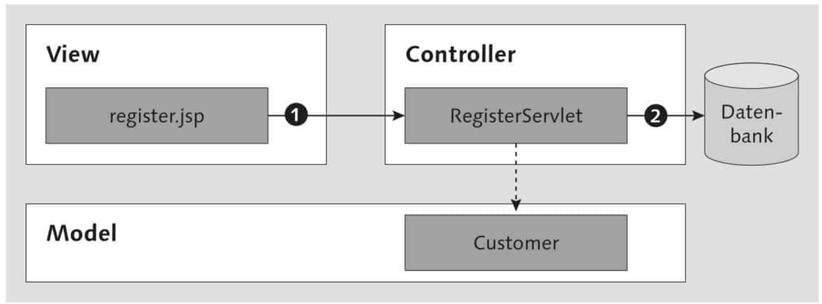


Abbildung 7.25 Das Übungsbeispiel

Für die Übung benötigen Sie die Kenntnisse über dynamische Webanwendungen, die in den vergangenen Kapiteln gezeigt wurden. Außerdem wäre es vorteilhaft, wenn Sie die Beispiele der letzten Kapitel programmiert hätten. Sollten Sie aber erst in diesem Kapitel eingestiegen sein, ist das auch kein Problem, denn die Servlet- und JSP-Kenntnisse, die Sie für das Beispiel benötigen, sind trivial.

Das Model (Customer.java)

Die Klasse mit dem Namen `Customer.java` wurde bereits in Kapitel 4, »Servlets«, gezeigt. In den folgenden Beispielen werden wir diese Klasse als Entity nutzen, um die persönlichen Daten des Kunden in der Tabelle `CUSTOMER` zu speichern.

Die View (register.jsp)

In diesem Beispiel bietet die `register.jsp` ein HTML-Formular an, über das ein Kunde seine E-Mail-Adresse und ein Passwort eingibt. Die `register.jsp` ruft das `RegisterServlet` auf.

```
<%@ include file="head.jspf" %>
    <form action="register" method="post">
    <fieldset>
    <legend>Registrieren</legend>
    <table>
        <tbody>
        <tr>
            <th>
                <label for="email">Email:</label>
            </th>
            <td>
```



```
                <input
                    type="email"
                    name="email"
                    size="40"
                    maxlength="40"
                    title="muster@beispiel.de"
                    placeholder="E-Mail eingeben"
                    pattern=".{6,40}"
                    required="required">
            </td>
        </tr>
        <tr>
            <th>
                <label for="password">
                    Password:
                </label>
            </th>
            <td>
                <input
                    type="password"
                    name="password"
                    size="10"
                    maxlength="10"
                    title="6-10 Zeichen"
                    placeholder=
                        "Passwort eingeben"
                    pattern=".{6,10}"
                    required="required">
                </td>
            </tr>
            <tr>
                <td/><td>
                    <input type="submit">
                    <input type="reset">
                </td>
            </tr>
        </tbody>
        </table>
        </fieldset>
        </form>
<%@ include file="footer.jspf" %>
```

Listing 7.7 »register.jsp«

Der Controller (RegisterServlet.java)

Im `RegisterServlet` werden die Parameter des HTTP-Requests entgegengenommen. Mit den erhaltenen Werten wird ein Objekt der Klasse `Customer` erzeugt und der Methode `persist()` zur Speicherung übergeben. Wenn keine Fehler aufgetreten sind, wird abschließend die Hauptseite `index.jsp` aufgerufen.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import de.java2enterprise.onlineshop.model.Customer;

public class RegisterServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Resource
    private DataSource ds;

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

        response.setContentType(
                "text/html;charset=UTF-8");

        final String email =
            request.getParameter("email");
        final String password =
            request.getParameter("password");

        Customer customer = new Customer();
        customer.setEmail(email);
        customer.setPassword(password);

        try {
            persist(customer);
        } catch (Exception e) {
            throw new ServletException(e.getMessage());
```



```
        }
        final RequestDispatcher dispatcher =
            request.getRequestDispatcher("index.jsp");
        dispatcher.forward(request, response);
    }

    private void persist(Customer customer)
            throws Exception {
        final Connection con = ds.getConnection();
        final Statement stmt = con.createStatement();
        stmt.executeUpdate(
        "INSERT INTO onlineshop.customer (" +
            "email, " +
            "password" +
        ") VALUES (" +
            "'" + customer.getEmail() + "', " +
            "'" + customer.getPassword() + "') "
        );
        con.close();
    }
}
```

Listing 7.8 »RegisterServlet.java«

7.4.2 Die Abfrage von Geschäftsdaten

In diesem Abschnitt wird gezeigt, wie mit JDBC Geschäftsdaten aus einer Datenbanktabelle geholt werden. In dem letzten Programmierbeispiel konnte sich ein Kunde über das Registrierungsformular registrieren. Dabei wurden seine Geschäftsdaten in der Datenbanktabelle `CUSTOMER` als Datensatz abgespeichert. Bei diesem Übungsbeispiel (siehe Abbildung 7.26) soll sich ein Kunde in die Onlineshop-Anwendung einloggen, indem er über ein HTML-Formular eine E-Mail-Adresse und ein Passwort eingibt.

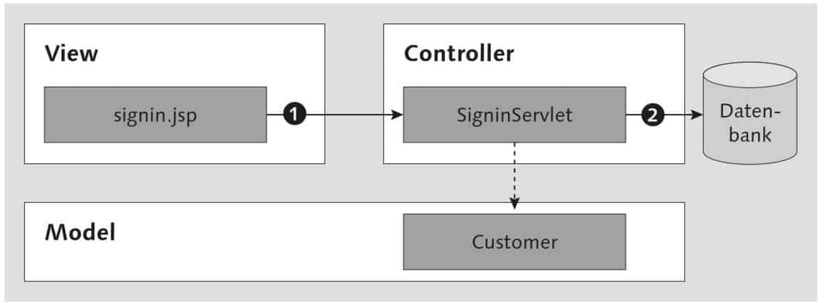


Abbildung 7.26 Das Einloggen eines Kunden

Mit diesen beiden Zeichenketten versucht die Anwendung, einen passenden Datensatz in der Tabelle `CUSTOMER` zu finden. Wenn ein entsprechender Datensatz gefunden werden konnte, wird mit der Ergebnismenge ein Objekt der Klasse `Customer` erzeugt und in der HTTP-Sitzung gespeichert.

Die View (signin.jsp)

Genauso wie die `register.jsp` enthält die `signin.jsp` ebenfalls ein HTML-Formular mit zwei Eingabefeldern mit den Namen `email` und `password`. Der einzige Unterschied zur `register.jsp` ist, dass die `signin.jsp` das `SigninServlet` aufruft.

```
...
    <form action="signin" method="post">
...
```

Listing 7.9 »signin.jsp«

Der Controller (SigninServlet.java)

Die Klasse `SigninServlet` nimmt zunächst die Parameter `email` und `password` entgegen. Anschließend ruft das Servlet mit diesen beiden Parametern die Methode `find()` auf.

Die Methode `find()` sucht in der Datenbanktabelle `CUSTOMER` einen Datensatz, dessen E-Mail-Adress- und Passwortfelder zu den erhaltenen Parametern passen. Wenn ein entsprechender Datensatz gefunden werden konnte, wird hiermit ein Objekt der Klasse `Customer` erzeugt. Die Methode gibt dieses Objekt als Rückgabewert zurück. Wenn kein Kunde unter der übergebenen E-Mail-Passwort-Kombination gefunden werden konnte, wird `null` als Wert zurückgegeben.

Wenn das Objekt ungleich null ist, wird es in der Methode `doPost()` in die aktuelle HTTP-Sitzung abgespeichert. Abschließend wird die Kontrolle an die Willkommensseite *index.jsp* abgegeben.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import de.java2enterprise.onlineshop.model.Customer;
```



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

    @Resource
    private DataSource ds;

    protected void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
                throws ServletException, IOException {
        String _email =
            request.getParameter("email");
        String _password =
            request.getParameter("password");

        Customer customer = null;
        try {
            customer = find(
                _email, _password);
        } catch (Exception e) {
            throw new ServletException(e.getMessage());
        }

        if(customer != null) {
            HttpSession session = request.getSession();
            session.setAttribute("customer", customer);
        }

        RequestDispatcher dispatcher =
                request.getRequestDispatcher("index.jsp");
            dispatcher.forward(request, response);

    }

    public static Customer find(
        String _email, String _password)
            throws Exception {

        Connection con = ds.getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(
            "SELECT " +
                "id, " +
```

```
            "email, " +
            "password " +
        "FROM onlineshop.customer " +
        "WHERE email='" + _email + "'" +
        "AND password='" + _password + "'" +
);

if(rs.next()) {

    Customer customer = new Customer();

    Long id = Long.valueOf(rs.getLong("id"));
    customer.setId(id);

    String email = rs.getString("email");
    customer.setEmail(email);

    String password = rs.getString("password");
    customer.setPassword(password);

    return customer;
}
con.close();
return null;
        con.close();
        return null;
    }
}
```

Listing 7.10 »SigninServlet.java«

Damit wir auch im Frontend der Java-EE-Anwendung ein Ergebnis erhalten, werden wir die E-Mail-Adesse des eingeloggten Anwenders in der Willkommensseite *index.jsp* anzeigen:

```
<%@ include file="head.jspf" %>
Eingeloggt: <c:out value="${customer.email}" default="-"/>
<%@ include file="footer.jspf" %>
```

Listing 7.11 »index.jsp«

7.5 Das objektrelationale Mapping

In diesem Abschnitt wird das *objektrelationale Mapping* (ORM) gezeigt. In der letzten Übung wurde ersichtlich, dass jedes Attribut einer Entity einem Spaltenfeld in der Datenbanktabelle entspricht. Gleichzeitig muss der Datentyp des Attributs zu dem Pendant des Spaltenfeldes

passen. Weil in der Java-Anwendung die Werte aus der Datenbank mit der JDBC-API geholt werden, ist es die Aufgabe des JDBC-Treibers, die Umwandlung von SQL-Datentypen in Java-Datentypen durchzuführen. Bei dieser Umwandlung gibt uns die JDBC-API aber nur selten die grundlegenden Java-Datentypen zurück, sondern bedient sich meistens eigener Klassen. Die Umwandlung der zurückgegebenen Datentypen in Java-Standard-Datentypen gehört hingegen zu den Aufgaben und Pflichten des Anwendungsentwicklers.

7.5.1 Von SQL zu Java

Tabelle 7.2 können Sie von rechts nach links entnehmen, wie das Mapping von den ANSI-SQL- und Oracle-Datentypen auf die grundlegenden Java-Standard-Datentypen der Java-Anwendung über die Klasse `java.sql.ResultSet` erfolgt.

	Java-Anwendung	Rückgabewert	java.spl.Result-Set	ANSI-SQL	ORACLE
Wahrheits-werte	java.lang.Boo-lean	boolean	getBoolean(x)	BOOLEAN	NUMBER(1)
»Kleinste« Ganzzahlen	java.lang.Byte	byte	getByte(x)	TINYINT	NUMBER(3)
»Kurze« Ganzzahlen	java.lang.Short	short	getShort(x)	SMALLINT	NUMBER(5)
»Normale« Ganzzahlen	java.lang.Integer	int	getInt(x)	INTEGER	NUMBER(10)
»Lange« Ganzzahlen	java.lang.Long	long	getLong(x)	BIGINT	NUMBER(19)
»Besonders lange Zahlen«	java.math.Big-Decimal	java.math.Big-Decimal	getBigDecimal(x)	DECIMAL	NUMBER(38)
Geldbeträge und andere Kommazahlen	java.lang.Double	double	getDouble(x)	DOUBLE	NUMBER(p, n)
»Kurze« Zeichenketten, bis 1000 Zeichen	java.lang.String	java.lang.String	getString(x)	VAR-CHAR(n)	VARCHAR (1000 char)
»Lange« Zeichenketten, über 1000 Zeichen	java.lang.String	java.lang.String java.sql.Clob	getString(x) getClob(x)	CLOB	CLOB

Tabelle 7.2 Die Zuordnung der SQL-Datentypen der ANSI-Norm zu der Oracle-Datenbank und den Java-Datentypen

	Java-Anwendung	Rückgabewert	java.spl.Result-Set	ANSI-SQL	ORACLE
Datum	java.util.Date	java.sql.Date	getDate(x)	DATE	DATE
Uhrzeit	java.util.Date	java.sql.Time	getTime(x)	TIME	DATE
Zeitstempel	java.util.Date	java.sql.Time-stamp	getTimestamp(x)	TIME-STAMP	TIME-STAMP(3)
Binärdateien	byte[]	byte[] java.sql.Blob	getBytes(x) getBlob(x)	BLOB	BLOB

Tabelle 7.2 Die Zuordnung der SQL-Datentypen der ANSI-Norm zu der Oracle-Datenbank und den Java-Datentypen (Forts.)

Wahrheitswerte

Für die numerischen SQL-Datentypen bietet die JDBC-API die Methode `getBoolean()` an. Obwohl die Oracle Database den SQL-Datentyp `BOOLEAN` nicht kennt, kann die Methode `getBoolean()` verwendet werden. Die einzige Voraussetzung ist, dass das Datenfeld eine 0 oder eine 1 als numerischen Wert enthält. Der Rest wird wie von Geisterhand erledigt. Der Rückgabewert ist vom primitiven Datentyp `boolean`.

Numerische Typen

Auch für die numerischen SQL-Datentypen bietet die JDBC-API Getter-Methoden an. Allerdings liefern sie stets primitive bzw. elementare Datentypen. Bei der Umwandlung der primitiven Datentypen sollte aber kein Autoboxing verwendet werden. Das Autoboxing birgt eine Gefahr, weil der Vergleichsoperator darin unterscheidet, ob ein Objekt per `new`-Operator oder per Autoboxing instanziiert wurde. Je nachdem, wie das Objekt initialisiert wurde, vergleicht der Operator eine Zahl oder eine Referenzadresse. Dieser Automatismus ist ein Risiko, weil in der Komplexität einer großen Java-EE-Anwendung manchmal nicht überschaubar ist, wie ein Objekt zustande kam. Wenn nicht ganz sicher ist, dass dieses Risiko vermieden werden kann, sollten Sie in geschäftskritischen Anwendungen gänzlich auf Autoboxing und Autounboxing verzichten.

Zeichenketten

Aus Tabelle 7.2 lässt sich entnehmen, dass Zeichenketten mit der Methode `getString()` geholt werden. Deshalb braucht der Anwendungsentwickler die Rückgabewerte nicht umzuwandeln.

Da die gewaltige Datenmenge eines CLOB aber zu Problemen in der Netzwerkschicht führen kann, müssen Sie dies als Anwendungsentwickler berücksichtigen und gegebenenfalls mit einem `java.io.InputStream` oder einem Objekt der Klasse `java.sql.Clob` arbeiten, um

zunächst einen Zeiger auf die eigentliche Zeichenkette zu erhalten. Das Problem wird somit umgangen, weil die Zeichenkette erst dann initialisiert werden muss, wenn die Daten tatsächlich benötigt werden. Dennoch müssen Sie beachten, dass der Zeiger nur innerhalb der Datenbanktransaktion gültig ist.

Zeitpunkte

Bei ANSI-SQL wird bei Zeitpunkten nach Datum (`DATE`), Uhrzeit (`TIME`) und Zeitstempeln (`TIMESTAMP`) unterschieden. Um das Mapping zu den ANSI-SQL-Datentypen zu vereinfachen, wurden bei der JDBC-API die Klassen `java.sql.Date`, `java.sql.Time` und `java.sql.Timestamp` beigefügt. Die zusätzlichen Klassen sind im Kern mit der Klasse `java.util.Date` verwandt, denn alle drei Klassen sind von ihr abgeleitet. Allerdings gehen die JDBC-Klassen spezifischer auf die Eigenarten der SQL-Datentypen ein.

Zum Beispiel beinhaltet der Datentyp `java.sql.Date` nur ein Datum. Die Uhrzeit wird automatisch entfernt. Und der Datentyp `java.sql.Time` führt lediglich eine Uhrzeit mit und verzichtet auf das Datum. Dagegen enthält die Klasse `java.sql.Timestamp` sowohl ein Datum, eine Uhrzeit als auch kleinste Sekundenteile bis hin zu Nanosekunden. Für die gängigsten Anwendungen werden aber lediglich die dreistelligen Millisekunden benötigt.

Binärdateien

Im Onlineshop wurde Java-seitig für die Fotodateien der theoretisch unendlich große Datentyp `byte[]` und in der Oracle Database der 4 Gigabyte große Datentyp `BLOB` festgelegt. Grundsätzlich kann der Wert einer binären Datei ganz einfach mit der Methode `getBytes()` geholt werden. Diese Möglichkeit wird auch im Onlineshop gezeigt. Je nach Anwendungsfall kann es aber hierbei theoretisch zum gleichen Problem kommen, das bereits weiter oben bei den Zeichenketten beschrieben wurde. Denn genauso wie bei der Klasse `String` bergen auch `byte[]`-Objekte das Risiko, dass sehr große Dateien zu Laufzeitengpässen in der Netzwerkschicht führen können. Auch für `byte[]` kann das Problem mit einem Input-Stream oder der Klasse `java.sql.Blob` durch einen Zeiger auf den eigentlichen Wert gelöst werden. Jedoch sind auch diese Zugriffe nur während der Transaktion gültig, weshalb wir mit ihnen behutsam umgehen müssen.

7.5.2 Die Umwandlung der JDBC-eigenen Datentypen

Weil das bisherige Onlineshop-Beispiel mit der Klasse `Customer` lediglich `VARCHAR`-Datentypen (bzw. wegen der Oracle Database `VARCHAR2`-Datentypen) an die Anwendung weiterreichte, brauchte keine Umwandlung vorgenommen zu werden. Anders sähe es bei der Abfrage von Artikeln aus, denn dort muss zum Beispiel auch ein Zeitpunkt besorgt werden.

Im folgenden Listing wird die komplette Datei `Item.java` gezeigt. In dieser Klasse haben wir alle Objektvariablen mit Java-Standard-Datentypen deklariert.

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

import java.util.Date;

public class Item {
    private Long id;
    private String description;
    private byte[] foto;
    private Double price;
    private String title;
    private Date traded;
    private Long seller_id;

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

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

    public String getDescription() {
        return this.description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public byte[] getFoto() {
        return this.foto;
    }

    public void setFoto(byte[] foto) {
        this.foto = foto;
    }

    public Double getPrice() {
        return this.price;
    }

    public void setPrice(Double price) {
        this.price = price;
```



```
    }

    public String getTitle() {
        return this.title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Long getSeller_id() {
        return seller_id;
    }

    public void setSeller_id(Long seller_id) {
        this.seller_id = seller_id;
    }

    public Date getTraded() {
        return this.traded;
    }

    public void setTraded(Date traded) {
        this.traded = traded;
    }
}
```

Listing 7.12 »Item.java«

Nun programmieren wir eine Methode mit dem Namen `find()`. In dieser Methode wird im Titel des Artikels nach einem Zeichenkettenfragment gesucht. Über `ResultSet.getTimestamp()` erhalten wir ein Objekt der JDBC-eigenen Klasse `java.sql.Timestamp`. Dies ist erforderlich, weil die Java-Standard-API keinen Datentyp vorhält, der zu dem SQL-Datentyp `TIMESTAMP` passt. Nun ist es unsere Aufgabe, das Objekt der Klasse `java.sql.Timestamp` in ein Objekt der Klasse `java.util.Date` umzuwandeln:

```
...
public static List<Item> find(String s)
        throws Exception {
    Connection con = ds.getConnection();
    PreparedStatement stmt =
            con.prepareStatement(
            "SELECT " +
```

```
                "id, " +
                "title, " +
                "description, " +
                "price, " +
                "seller_id, " +
                "buyer_id, " +
                "traded " +
            "FROM onlineshop.item " +
            "WHERE title like ?"
            );
stmt.setString(
    1,
    (s==null) ? "%" : "%" + s + "%");
ResultSet rs = stmt.executeQuery();
List<Item> items = new ArrayList<Item>();
while(rs.next()) {

    Item item = new Item();

    Long id = Long.valueOf(rs.getLong("id"));
    item.setId(id);

    String title = rs.getString("title");
    item.setTitle(title);

    String description =
        rs.getString("description");
    item.setDescription(description);

    double price = rs.getDouble("price");
    if(price != 0) {
        item.setPrice(
            Double.valueOf(price));
    }

    long seller_id = rs.getLong("seller_id");
    if(seller_id != 0) {
        item.setSeller_id(
            Long.valueOf(seller_id));
    }

    long buyer_id = rs.getLong("buyer_id");
    if(buyer_id != 0) {
```



```
            item.setBuyer_id(
                Long.valueOf(buyer_id)
            );
        }

        Timestamp ts = rs.getTimestamp("traded");
        if(ts != null) {
            Date traded =
                new Date(ts.getTime());
            item.setTraded(traded);
        }

        items.add(item);
    }
    con.close();
    return items;
}
...
```

Listing 7.13 »SearchServlet.java«

7.6 Vorbereitete SQL-Anweisungen

Im letzten Abschnitt wurden in der Klasse `CustomerDAO` SQL-Anweisungen als Zeichenketten fertiggestellt und an die Datenbank versandt. Dabei haben wir die grundlegende Klasse `java.sql.Statement` eingesetzt. Bei wenigen hintereinander folgenden SQL-Anweisungen ist das Programm im Handumdrehen mit der Aufgabe fertig. Wenn es sich aber um Millionen von INSERTS handelt, ist die Klasse `java.sql.Statement` zu langwierig. Stattdessen wurden für diesen Zweck *vorbereitete* SQL-Anweisungen erfunden.

7.6.1 Die Klasse »PreparedStatement«

Mit der Klasse `java.sql.PreparedStatement` können vorbereitete SQL-Anweisungen erstellt und an die Datenbank verschickt werden. Weil die Datenbank hierdurch bereits ein Grundgerüst erhält, bevor sie die SQL-Anweisung abwickelt, kann sie sich zunächst einen Ausführungsplan anlegen, der für die Leistung optimal ist. Je größer die Anzahl der Datensätze ist, desto stärker macht sich die Geschwindigkeitsoptimierung einer vorbereiteten Anweisung bemerkbar.

Wenn SQL-Anweisungen handgeschrieben erstellt werden und kein Persistenz-Framework eingesetzt wird, nutzt man heutzutage kaum mehr die einfache Variante `java.sql.Statement`, sondern eigentlich nur noch die Klasse `java.sql.PreparedStatement`. Denn schnell wurde

noch ein weiterer Vorzug von vorbereiteten SQL-Anweisungen deutlich. SQL-Anweisungen bergen nämlich die Gefahr, dass sie von Hackern durch eine sogenannte SQL-Injection auf dem Weg zur Datenbank manipuliert werden. Dagegen sind vorbereitete Anweisungen wesentlich besser vor Korruption geschützt.

Die Arbeitsweise mit der Klasse `java.sql.PreparedStatement` ähnelt der mit der Klasse `java.sql.Statement`. Statt der einzufügenden Werte enthalten vorbereitete SQL-Anweisungen aber jeweils nur ein Fragezeichen. Die Klasse bietet für unterschiedliche Datentypen jeweils eine Setter-Methode an. Dadurch kann die Datenbank vor der eigentlichen Ausführung das Fragezeichen durch den konkreten Wert ersetzen. Der erste Parameter der Setter-Methoden zeigt mit einer 1 beginnend die Nummer der zu ändernden Spalte an. Der zweite Parameter enthält den zu setzenden Wert.

In dem folgenden Listing wird die Klasse `java.sql.PreparedStatement` statt der Klasse `java.sql.Statement` verwendet:

```
...
public static void persist(Customer customer)
        throws Exception {
    Connection con = ds.getConnection();
    PreparedStatement stmt = con.prepareStatement(
        "INSERT INTO onlineshop.customer(" +
            "email, " +
            "password" +
        ") VALUES (" +
            "?, " +
            "?)"
        );
    stmt.setString(1, customer.getEmail());
    stmt.setString(2, customer.getPassword());
    stmt.executeUpdate();
    con.close();
}
...
```

Listing 7.14 »RegisterServlet.java«

Genauso wie für eine INSERT-Anweisung kann auch für ein SELECT ein `java.sql.PreparedStatement` eingesetzt werden:

```
...
public static Customer find(
    String _email, String _password)
        throws Exception {
```



```
    Connection con = ds.getConnection();
    PreparedStatement stmt = con.prepareStatement(
            "SELECT " +
                "id, " +
                "email, " +
                "password " +
            "FROM onlineshop.customer " +
            "WHERE email=? " +
            "AND password=?");

    stmt.setString(1, _email);
    stmt.setString(2, _password);

    ResultSet rs = stmt.executeQuery();
    if(rs.next()) {

        Customer customer = new Customer();

        Long id = Long.valueOf(rs.getLong("id"));
        customer.setId(id);

        String email = rs.getString("email");
        customer.setEmail(email);

        String password = rs.getString("password");
        customer.setPassword(password);

        return customer;
    }
    con.close();
    return null;
}
...
```

Listing 7.15 »SigninServlet.java«

7.6.2 Von Java zu SQL

In Abschnitt 7.5.1, »Von SQL zu Java«, wurde gezeigt, wie die Datentypen der Ergebnismenge einer Abfrage extra umgewandelt werden müssen, weil die Klasse `ResultSet` häufig keine Java-Standard-Datentypen liefert.

Und auch bei den Setter-Methoden eines `PreparedStatement` muss eine Umformung gleichermaßen vorgenommen werden. Tabelle 7.3 zeigt von links nach rechts, wie das Mapping

von den grundlegenden Java-Standard-Datentypen der Java-Anwendung auf die ANSI-SQL- und Oracle-Datentypen erfolgt.

	Java-Anwendung	java.sql.PreparedStatement	ANSI-SQL	ORACLE
Wahrheitswerte	java.lang.Boolean	setBoolean(x, boolean)	BOOLEAN	NUMBER(1)
»Kleinste« Ganzzahlen	java.lang.Byte	setByte(x, byte)	TINYINT	NUMBER(3)
»Kurze« Ganzzahlen	java.lang.Short	steShort(x, short)	SMALLINT	NUMBER(5)
»Normale« Ganzzahlen	java.lang.Integer	setInt(x, int)	INTEGER	NUMBER(10)
»Lange« Ganzzahlen	java.lang.Long	setLong(x, long)	BIGINT	NUMBER(19)
»Besonders lange Zahlen«	java.math.BigDecimal	setBigDecimal(x, java.math.BigDecimal)	DECIMAL	NUMBER(38)
Geldbeträge und andere Kommazahlen	java.lang.Double	setDouble(x, double)	DOUBLE	NUMBER(p, n)
»Kurze« Zeichenketten, bis 1000 Zeichen	java.lang.String	setString (x, java.lang.String)	VARCHAR(n)	VARCHAR2 (1000 char)
»Lange« Zeichenketten, über 1000 Zeichen	java.lang.String	setString(x, java.lang.String) setClob(x, java.sql.Clob)	CLOB	CLOB
Datum	java.util.Date	setDate(x, java.sql.Date)	DATE	DATE
Uhrzeit	java.util.Date	setTime(x, java.sql.Time)	TIME	DATE
Zeitstempel	java.util.Date	setTimestamp(x, java.sql.Timestamp)	TIMESTAMP	TIMESTAMP(3)
Binärdateien	byte[]	setBytes(x, byte[]) setBlob(x, java.sql.Blob)	BLOB	BLOB

Tabelle 7.3 Die gängigsten Setter-Methoden der Klasse »java.sql.PreparedStatement«

7.6.3 Das Abrufen automatisch generierter Schlüssel

Im letzten Kapitel wurde gezeigt, wie bei einer INSERT-Anweisung Primärschlüssel automatisch erzeugt werden. Durch die automatische Generierung von Primärschlüsseln haben wir einen Geschwindigkeitsvorteil erzielt. Weil sich die Datenbank selbstständig um die Erzeugung der Schlüssel kümmert, musste sich die Java-EE-Anwendung nicht um deren Konsistenz kümmern. Allerdings hat die automatische Generierung von Primärschlüsseln auch einen Nachteil. Denn wenn die Primärschlüssel in der Datenbank automatisch generiert werden, geschieht dies aus der Sicht der Java-EE-Anwendung im Verborgenen. Das Problem bei den bisher gezeigten Mitteln ist, dass die Java-EE-Anwendung Mühe hat, den Bezug zwischen einem Objekt und einem generierten Schlüssel zu finden.

Stellen Sie sich vor, dass sich ein Kunde mit der Anwendung soeben registriert hat und nun unmittelbar danach auch automatisch eingeloggt sein soll. Um einen Artikel zu erstellen, benötigen wir die ID des Verkäufers, denn die muss ja ins Feld `SELLER_ID` der Tabelle `ITEM` gespeichert werden. Obwohl ein Primärschlüssel automatisch generiert wurde, haben wir seine ID nicht zur Hand. Denn die Methode `executeUpdate()` liefert von Natur aus keinen entsprechenden Rückgabewert.

Eine Möglichkeit ist, den Kunden über die eindeutige E-Mail-Adresse zu suchen. Diese Lösung wird in einer geschäftskritischen Anwendung jedoch kaum Verwendung finden, denn sie ist unsicher und relativ kostspielig. Und eigentlich weist einzig und allein der Primärschlüssel eindeutig auf die Identität eines Geschäftsobjekts hin.

Für diese Problemstellung wurde in der aktuellen JDBC-API jedoch vorgesorgt. Um automatisch generierte Schlüssel unmittelbar nach der Erzeugung zu erhalten, kann die Methode `Connection.prepareStatement()` noch mit einem zweiten Parameter aufgerufen werden.

Wenn als zweiter Parameter zum Beispiel ein String-Array mitgegeben wird, geht der Treiber davon aus, dass es sich um die Namen der automatisch generierten Schlüssel handelt:

```
String[] autogeneratedKeys = new String[]{"id"};

PreparedStatement stmt = con.prepareStatement(
    "INSERT INTO onlineshop.customer(" +
        "email, " +
        "password" +
    ") VALUES (" +
        "?, " +
        "?)", autogeneratedKeys
    );
```

Listing 7.16 Connection.prepareStatement() mit Array aus Schlüsselnamen als zweitem Parameter

Nachdem die Methode `executeUpdate()` der Klasse `java.sql.PreparedStatement` den neuen Datensatz eingefügt hat, gibt die Methode `getGeneratedKeys()` ein `ResultSet` zurück, das die automatisch generierten Schlüssel beinhaltet:

```
ResultSet rs = stmt.getGeneratedKeys();
```

Das folgende Listing zeigt, wie das `Customer`-Objekt einen Primärschlüssel erhält, nachdem es persistiert wurde. Dadurch kann die aufrufende Methode mit dem aktuellen Kunden-Objekt gleich weiterarbeiten.

Als Beispiel werden wir die erhaltene ID nun verwenden. Damit wir im Frontend ein Ergebnis über die im Hintergrund gelaufenen Programmabläufe erhalten, werden wir im Request-Scope eine Nachricht hinterlegen. Wenn das `id`-Feld einen Wert erhalten hat, soll die Meldung »Die Registrierung war erfolgreich!« ausgegeben werden, ansonsten erhält der Benutzer die Meldung »Die Registrierung war erfolglos!«.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import javax.naming.InitialContext;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
import de.java2enterprise.onlineshop.model.Customer;

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

    @Resource
    private DataSource ds;

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



```
        response.setContentType(
                "text/html;charset=UTF-8");

        String email = request.getParameter("email");
        String password =
            request.getParameter("password");

        Customer customer = new Customer();
        customer.setEmail(email);
        customer.setPassword(password);

        try {
            persist(customer);
        } catch (Exception e) {
            throw new ServletException(e.getMessage());
        }

        if(customer.getId() != null) {
            request.setAttribute("message", "Die Registrierung war erfolgreich!");
        } else {
            request.setAttribute("message", "Die Registrierung war erfolglos!");
        }

        RequestDispatcher dispatcher =
            request.getRequestDispatcher("index.jsp");
        dispatcher.forward(request, response);
    }

    public void persist(Customer customer)
            throws Exception {
        String[] autogeneratedKeys = new String[]{"id"};
        Connection con = ds.getConnection();
        PreparedStatement stmt = con.prepareStatement(
            "INSERT INTO onlineshop.customer(" +
                "email, " +
                "password" +
            ") VALUES (" +
                "?, " +
                "?)", autogeneratedKeys
            );

        stmt.setString(1, customer.getEmail());
        stmt.setString(2, customer.getPassword());
```

```
            stmt.executeUpdate();

            ResultSet rs = stmt.getGeneratedKeys();
            Long id = null;
            while(rs.next()) {
                id = rs.getLong(1);
                customer.setId(id);
            }
            con.close();
        }
}
```

Listing 7.17 »RegisterServlet.java«

In der `index.jsp` werden wir die Nachricht aus dem `RegisterServlet` anzeigen:

```
<%@ include file="head.jspf" %>
<p>${message}</p>
<%@ include file="footer.jspf" %>
```

Listing 7.18 »index.jsp«

7.7 Binärdaten

Ein Bild sagt mehr als 1000 Worte, und auch das Programmierbeispiel unseres Onlineshops macht deutlich, dass heutzutage audiovisuelle Medien eine wichtige Stellung einnehmen. Umso wichtiger ist, dass eine Java-EE-Anwendung problemlos Binärdaten speichern kann. Wie Sie gleich sehen werden, ist die Speicherung von Bildern kein Hexenwerk. Technisch gesehen wurden sogar bereits alle hierfür erforderlichen Bestandteile der Java-API beschrieben. In einem Programmierbeispiel werden die verschiedenen Fragmente nun noch einmal zusammengeführt. Denn am praktischen Beispiel wird am besten deutlich, wie die Speicherung eines Bildes vom Webbrowser bis hin zur Datenbank erfolgt.

7.7.1 Bilder speichern

In dem Programmierbeispiel des Onlineshops bieten Kunden Artikel an, indem sie über das Verkaufsformular in der *sell.jsp* die Daten des Artikels eingeben. Dabei bietet das Formular nicht nur die Möglichkeit, den Titel, die Beschreibung und den Preis des Artikels beizufügen, sondern enthält auch einen Hochlademechanismus für Bilddateien.

In dem Programmierbeispiel zu der Übung soll die Klasse `SellServlet` die entgegengenommenen Daten inklusive Bilddatei in der Datenbanktabelle `ITEM` speichern (siehe Abbildung 7.27).

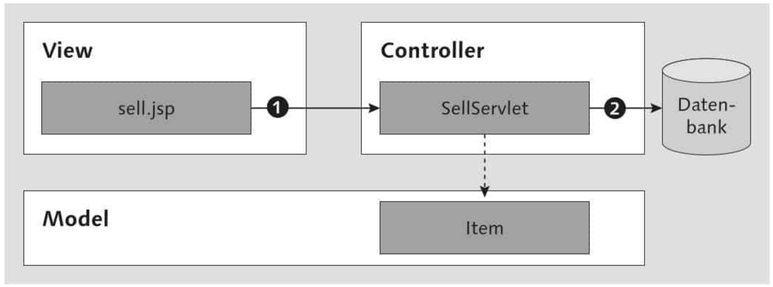


Abbildung 7.27 Das MVC-Entwurfsmuster für das Einstellen von Artikeln

Das Modell (Item.java)

In der Entity `Item.java` werden die Attribute Titel, Beschreibung, Preis und Foto implementiert. Das heißt, dass sie als private Objektvariablen deklariert und über öffentliche Setter- und Getter- Methoden nach außen hin angeboten werden.

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

import java.io.Serializable;
import java.util.Date;

public class Item implements Serializable {
    private static final long
        serialVersionUID = -6604363993041715170L;

    private Long id;
    private String title;
    private String description;
    private Double price;
    private byte[] foto;
    private Long seller_id;
    private Long buyer_id;
    private Date traded;

    public Item() {}

    public Item(
            String title,
            String description,
            Double price,
            Long seller_id) {
```

```
        this.title = title;
        this.description = description;
        this.price = price;
        this.seller_id = seller_id;
    }

    public Long getId() {
        return id;
    }

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

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public byte[] getFoto() {
        return foto;
    }
```



```
    public void setFoto(byte[] foto) {
        this.foto = foto;
    }

    public Long getSeller_id() {
        return this.seller_id;
    }

    public void setSeller_id(Long seller_id) {
        this.seller_id = seller_id;
    }

    public Long getBuyer_id() {
        return this.buyer_id;
    }

    public void setBuyer_id(Long buyer_id) {
        this.buyer_id = buyer_id;
    }

    public Date getTraded() {
        return traded;
    }

    public void setTraded(Date traded) {
        this.traded = traded;
    }

    public String toString() {
        return
            "[" +
            getId() + "," +
            getTitle() + "," +
            getDescription() + "," +
            getPrice() + "," +
            getSeller_id() + "," +
            getBuyer_id() + "," +
            getTraded() +
            "]";
    }
}
```

Listing 7.19 »Item.java«

Die View (sell.jsp)

Die Datei *sell.jsp* wurde bereits in einem vorherigen Kapitel gezeigt. Damit Sie das Listing zu dem Programmierbeispiel nicht suchen müssen, wird es an dieser Stelle noch einmal abgedruckt:

```
<%@ include file="head.jspf" %>
    <form
        action="sell"
        method="post"
        enctype="multipart/form-data">
    <fieldset>
    <legend>Verkaufen</legend>
    <table>
        <tbody>
        <tr>
            <th>
                <label for="title">Titel:</label>
            </th>
            <td>
                <input
                    type="text"
                    name="title"
                    size="40"
                    maxlength="40"
                    title=
                    "Ein Titel für den Artikel"
                    placeholder="Titel eingeben"
                    pattern=".{6,40}"
                    required="required">
    </td>
</tr>
<tr>
    <th>
        <label
            for="description">
            Beschreibung:
        </label>
    </th>
    <td>
        <textarea
            name="description"
            cols="100"
            rows="10"
```



```
            maxlength="1000">
        </textarea>
    </td>
</tr>
<tr>
    <th>
        <label
            for="price">
            Preis:
        </label>
    </th>
    <td>
        <input
            type="number"
            name="price"
            size="40"
            maxlength="40"
            title=
            "Ein Preis für den Artikel"
            placeholder="Preis eingeben"
            pattern=".{1,40}"
            required="required">
        </td>
    </tr>
    <tr>
        <th>
            <label
                for="foto">
                Foto:
            </label>
        </th>
        <td>
            <input type="file" name="foto">
        </td>
    </tr>
    <tr>
        <td/><td>
            <input type="submit">
            <input type="reset">
        </td>
    </tr>
</tbody>
</table>
```

```
    </fieldset>
    </form>
<%@ include file="footer.jspf" %>
```

Listing 7.20 »sell.jsp«

Der Controller (SellServlet.java)

In Kapitel 4, »Servlets«, wurde bereits beschrieben, wie man aus einem Multipart/Form-Data-Request Bilder einliest. Im folgenden Listing der Klasse `SellServlet.java` wird nun gezeigt, wie das Bild des Artikels in der Datenbanktabelle `ITEM` gespeichert wird.

Im Beispiel wird das Bild gemeinsam mit allen anderen Geschäftsdaten des Artikels in der Klasse `Item.java` gekapselt und an die Methode `persist()` zur Speicherung übergeben.

```
package de.java2enterprise.onlineshop;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import javax.annotation.Resource;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.servlet.http.Part;
import javax.sql.DataSource;

import de.java2enterprise.onlineshop.model.Customer;
import de.java2enterprise.onlineshop.model.Item;

@WebServlet("/sell")
@MultipartConfig(
        location="/tmp",
        fileSizeThreshold=1024*1024,
        maxFileSize=1024*1024*5,
        maxRequestSize=1024*1024*5*5)
```



```
public class SellServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Resource
    private DataSource ds;

    public void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        Part part = request.getPart("foto");
        ByteArrayOutputStream baos =
        new ByteArrayOutputStream();
        try {
            InputStream in = part.getInputStream();
            int i = 0;
            while ((i = in.read()) != -1) {
                baos.write(i);
            }
        } catch (IOException ex) {
            throw new ServletException(ex.getMessage());
        }

        HttpSession session = request.getSession();
        Object customer =
            session.getAttribute("customer");
        if(customer != null) {
            String title =
                request.getParameter("title");
            String description =
                request.getParameter("description");
            String price =
                 request.getParameter("price");

            Item item = new Item();
            item.setTitle(title);
            item.setDescription(description);
            item.setPrice(Double.valueOf(price));
            item.setSeller_id(
                ((Customer)customer).getId());
            item.setFoto(baos.toByteArray());
            baos.flush();
```

```
            try {
                persist(item);

            } catch (Exception e) {
                throw new ServletException(
                        e.getMessage());
            }
            RequestDispatcher dispatcher =
                request.getRequestDispatcher(
                    "index.jsp");
            dispatcher.forward(request, response);
        }
}

public void persist(Item item)
            throws Exception {
        String[] autogeneratedKeys = new String[]{"id"};
        Connection con = ds.getConnection();
        PreparedStatement stmt = con.prepareStatement(
            "INSERT INTO onlineshop.item (" +
                "title, " +
                "description, " +
                "price, " +
                "foto, " +
                "seller_id " +
            ") VALUES (" +
                "?, " +
                "?, " +
                "?, " +
                "?, " +
                "?" +
            ") ", autogeneratedKeys
);
stmt.setString(1, item.getTitle());
stmt.setString(2, item.getDescription());
stmt.setDouble(3, item.getPrice());
stmt.setBytes(4, item.getFoto());
stmt.setLong(5, item.getSeller_id());
stmt.executeUpdate();

ResultSet rs = stmt.getGeneratedKeys();
Long id = null;
while(rs.next()) {
    id = rs.getLong(1);
```



```
            item.setId(id);
        }
        con.close();
    }
}
```

Listing 7.21 »SellServlet.java«

Dem Listing kann man entnehmen, wie die Methode `persist()` den Titel, die Beschreibung, den Preis und das Foto in der Datenbanktabelle speichert. Dabei wird auch hier der neue Primärschlüssel als Wert der Objektvariablen `id` übernommen.

Bilder verkleinern

Wenn bei der Entwicklung einer Webanwendung Bilder zum Server hochgeladen werden, tritt sehr schnell ein bestimmtes Problem auf. Denn es ist schwer vorhersehbar, in welchem Format und in welcher Größe das Original losgeschickt wird. Wir werden an dieser Stelle nicht weiter auf die Verschiedenheit der Bildtypen eingehen. Aber die Größe von Bildern soll dennoch kurz besprochen werden.

Als Beispiel werden wir eine Methode programmieren, die vor der Speicherung aufgerufen werden soll, um die Bilder in ihrer längeren Seite einheitlich zu transformieren. Wir nennen die Methode deshalb `scale()`.

Die Methode `scale()` nutzt eine Konstante, der wir den Namen `MAX_IMAGE_LENGTH` geben und die wir ganz oben im Servlet deklarieren werden. Der Wert der Konstanten wurde ganz willkürlich auf 400 Pixel gesetzt:

```
public final static int MAX_IMAGE_LENGTH = 400;
```

Wir legen also fest, dass die längere Seite des Bildes auf die von uns eigenmächtig festgelegte Länge von 400 Pixeln komprimiert bzw. gestreckt wird.

Die kürzere Länge wird mit dem gleichen Faktor verkleinert oder vergrößert, sodass sich das Seitenverhältnis nicht ändert. Auf diese Weise soll bei der Transformation eine Verzerrung des Bildes vermieden werden.

Die Methode `scale()` soll mit dem Bild als `byte[]`-Parameter aufgerufen werden können:

```
public byte[] scale(byte[] foto) throws IOException {
```

Unser Bild ist nun in einem Byte-Buffer in Originalgröße gespeichert. Da die Größe des Bildes unseren Anforderungen eventuell nicht entspricht, müssen wir dafür Sorge tragen, dass nun das Bild skaliert wird. Hierzu müssen wir erst einmal herausfinden, welche Breite und welche Höhe bei dem Bild vorliegt. Dies erreichen wir, indem wir aus den Bilddaten und der Klasse `ImageIO` ein Objekt der Klasse `BufferedImage` erstellen. Die Klasse `ImageIO` benötigt die Bildda-

ten aber als Eingabestrom. Deshalb erzeugen wir zunächst ein Objekt der Klasse `ByteArrayInputStream`:

```
ByteArrayInputStream byteArrayInputStream =
    new ByteArrayInputStream(foto);
```

Die Breite und die Höhe des Bildes sind anschließend einfach zu ermitteln. Wir erhalten sie mithilfe der Convenience-Methoden `getWidth()` und `getHeight()`. Da wir gleich eine Bruchrechnung mit den Werten vorhaben, casten wir beide Werte zum Datentyp `double`:

```
BufferedImage originalBufferedImage =
    ImageIO.read(byteArrayInputStream);

double originalWidth = (double)
    originalBufferedImage.getWidth();
    double originalHeight = (double)
        originalBufferedImage.getHeight();
```

Der für die Skalierung relevante Wert ist der größere der beiden:

```
double relevantLength =
    originalWidth > originalHeight ?
        originalWidth : originalHeight;
```

Der Skalierungsfaktor für die Transformation errechnet sich aus der maximal zulässigen Länge, `MAX_IMAGE_LENGTH`, geteilt durch die relevante Länge:

```
double transformationScale =
    MAX_IMAGE_LENGTH / relevantLength;
```

Die resultierende Breite und Länge des Bildes erhalten wir, indem wir die ursprünglichen Werte mit dem Skalierungswert multiplizieren. Das Ergebnis wird mit `Math.round(double)` zu einer Ganzzahl gerundet. `Math.round(double)` liefert die Werte im Datentyp `long`. So hohe Wertbereiche erwarten wir in Pixeln nicht, deshalb können wir die erhaltenen Ganzzahlen von `long` auf `int` casten:

```
int width = (int) Math.round(
    transformationScale * originalWidth );
    int height = (int) Math.round(
        transformationScale * originalHeight );
```

Nun erzeugen wir ein neues Objekt der Klasse `BufferedImage` und instanziieren es mit der erwünschten und soeben erhaltenen Breite und Länge. Als dritten Parameter erwartet `BufferedImage` eine Angabe zu dem Bildtyp. Wir entscheiden uns hier für den Typ `INT_RGB`, ein 8-Bit-RGB-Bild ohne Alpha-Werte.


```
BufferedImage resizedBufferedImage =
    new BufferedImage(
        width, height, BufferedImage.TYPE_INT_RGB);
```

Für die Transformation benötigen wir die Java-Klasse `Graphics2D`, die zur Verbesserung der ursprünglichen Klasse `Graphics` entwickelt worden ist. Die Klasse `Graphics2D` ist zwar Erbe der Klasse `Graphics`, jedoch gehen ihre Fähigkeiten weit über die alte Klasse `Graphics` hinaus. `Graphics2D` wird für sämtliche zweidimensionalen Umwandlungen (wie Verschiebungen, Skalierungen oder Farbmanipulationen) verwendet. Die Umwandlung basiert auf dem sogenannten benutzerabhängigen Koordinatensystem *Transformation Userspace*. Diese Standardeinstellung ist für Geräte bestimmt, die im Bereich von 72 dpi arbeiten, also 72 Punkte pro Inch darstellen. Da eine Transformation somit geräteabhängig ist, müssten wir die Gerätekonfiguration erst abfragen. Unser Zielgerät ist jedoch ein Unix-Root-Server ohne Grafik-API, deshalb werden wir uns an dieser Stelle mit der gegebenen Schätzung begnügen.

```
Graphics2D g2d =
    resizedBufferedImage.createGraphics();
```

Für grafische Umwandlungen gibt es eine Vielzahl an Vorgehensweisen. Um einen Hinweis darauf zu geben, welche Vorgehensweise oder welcher Algorithmus erwünscht ist, können Sie mithilfe der Methode `setRenderingHint(Key, Object)` Vorlieben nennen. Hierdurch wird insbesondere die Qualität bzw. die Laufzeit des Prozesses beeinflusst. Das heißt, je besser die Qualität des Resultats sein soll, desto länger muss auf das Ergebnis gewartet werden. Der Implementation steht es frei, inwieweit sie sich der Zielvorgabe des Hinweises nähert oder ob sie sogar die Hinweise vollständig ignoriert. Wir setzen an dieser Stelle eine bikubische Interpolation ein:

```
g2d.setRenderingHint(
    RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_INTERPOLATION_BICUBIC);
```

Die eigentliche Skalierung erhalten wir mithilfe der Klasse `AffineTransform`. Sie erwirkt eine lineare Abbildung von 2D-Koordinaten auf andere 2D-Koordinaten, bewahrt also durch ein spezielles Verfahren die »Geradlinigkeit« und »Parallelität«. Wir müssen zwei Parameter angeben. Der erste gibt die Skalierung in der X-Achse an und der zweite die Skalierung in der y-Achse. Da wir eine gleichmäßige Skalierung wünschen, geben wir jeweils den Wert `transformationScale` an.

```
AffineTransform affineTransform =
    AffineTransform.getScaleInstance(
        transformationScale,
        transformationScale);
g2d.drawRenderedImage(
    originalBufferedImage, affineTransform);
```

Für die Speicherung in die Datenbank benötigen wir einen Eingabestrom. Mithilfe der Klasse `ImageIO` schreiben wir unser neues `BufferedImage`-Objekt in einen `ByteArrayOutputStream` und setzen den so erhaltenen Buffer in einen `ByteArrayInputStream`:

```
ByteArrayOutputStream baos =
    new ByteArrayOutputStream();
ImageIO.write(resizedBufferedImage, "PNG", baos);
return baos.toByteArray();
}
```

Sie können die Methode in der Klasse `SellServlet` nutzen, indem Sie die Methode aufrufen, bevor das Byte-Array als Wert des Attributs `Foto` gesetzt wird:

```
...
item.setFoto(scale(baos.toByteArray()));
...
```

7.7.2 Bilder anzeigen

Um Bilder auf einer Webseite anzuzeigen, muss man einen Umweg machen. Der Grund hierfür liegt in der Art und Weise, wie ein Webbrowser Bilder für eine Webseite nutzt.

Ein Webbrowser fragt über das HTTP-Protokoll zunächst nach einem HTML-Dokument. Nachdem er das HTML-Dokument eingelesen hat, merkt der Webbrowser anhand der enthaltenen IMG-Tags, dass er für die Webseite Bilder nachladen muss. Erst dann besorgt er sich über weitere HTTP-Requests die Bilder. Das bedeutet, dass er für jedes Bild einen eigenen HTTP-Request an den Server verschicken muss. Demgemäß müssen wir in der Java-EE-Anwendung zunächst ein Dokument erstellen, das die einzelnen `IMG`-Elemente enthält.

Abbildung 7.28 zeigt alle Komponenten, die für die User-Story benötigt werden.

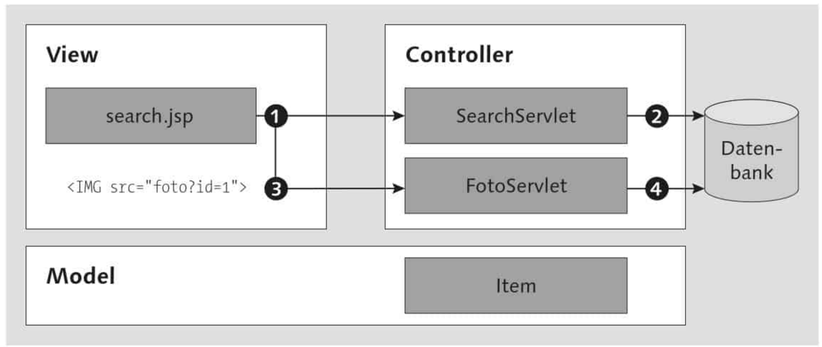


Abbildung 7.28 Bei der Anzeige von Artikeln müssen die Bilder über einen Umweg besorgt werden.

Die View (search.jsp)

Im Programmierbeispiel benötigen wir zunächst die View-Komponente *search.jsp*. Ganz oben enthält die JSP ein HTML-Formular, über das der Kunde Artikel suchen kann.

```
<%@ include file="head.jspf" %>
<article>
    <section>
        <form action="search" method="post">
        <fieldset>
        <legend>Suchen</legend>
        <table>
            <tbody>
            <tr>
                <th>
                    <label for="search">
                        Suche:
                    </label>
                </th>
                <td>
                    <input
                        type="text"
                        name="search"
                        size="40"
                        maxlength="40"
                        title="Suchtext"
                        placeholder=
                        "Suchtext eingeben"
                        >
                </td>
                <td>
                    <input type="submit">
                    <input type="reset">
                </td>
            </tr>
        </tbody>
        </table>
        </fieldset>
        </form>
    </section>
</article>
```

Listing 7.22 »search.jsp«

Unter dem HTML-Formular werden die Daten mit der Expression Language angezeigt. Die Bereiche, in denen die Artikel angezeigt werden, enthalten jeweils ein IMG-Element. Das IMG-Element wird als Quelle das URL-Pattern `"foto"` mitführen. Gleichzeitig wird das URL-Pattern auch die ID des Artikels mitliefern.

Sobald der Webbrowser die *search.jsp* erhalten hat, durchstöbert er das Dokument nach IMG-Elementen. Jedes Mal, wenn er auf ein IMG-Element stößt, versendet der Webbrowser einen GET-Request an die angegebene URL, um das Ergebnis an der Stelle des IMG-Elements anzuzeigen:

```
<c:forEach var="item" items="${items}">
<article>
    <section>
        <form action="buy" method="post">
        <fieldset>
        <legend>ID: ${item.id}</legend>
        <h2>${item.title}</h2>
        <p>${item.description}</p>
        <p><img src="foto?id=${item.id}">
        <p>Preis: ${item.price} Euro</p>
        </fieldset>
        </form>
    </section>
</article>
</c:forEach>
<%@ include file="footer.jspf" %>
```

Listing 7.23 »search.jsp«

Der Controller (SearchServlet.java)

Wenn ein Kunde auf den SUBMIT-Button klickt, wird seine Suche an das `SearchServlet` geleitet. Dabei werden die Artikel aus der Datenbank geholt, bei denen der Suchtext im Titel erscheint. Die gefundenen Artikel werden in der HTTP-Sitzung abgelegt.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```



```
import javax.naming.InitialContext;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.sql.DataSource;
import de.java2enterprise.onlineshop.model.Item;


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

    @Resource
    private DataSource ds;

    public void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

    String s = request.getParameter("search");
    try {
        List<Item> items = find(s);
        if(items!=null) {
            HttpSession session =
                request.getSession();
            session.setAttribute(
                "items", items);
        }
    } catch (Exception e) {
        throw new ServletException(e.getMessage());
    }
    RequestDispatcher dispatcher =
    request.getRequestDispatcher("search.jsp");
    dispatcher.forward(request, response);
}

public static List<Item> find(String s)
        throws Exception {
    Connection con = ds.getConnection();
```

```
PreparedStatement stmt =
        con.prepareStatement(
        "SELECT " +
            "id, " +
            "title, " +
            "description, " +
            "price, " +
            "seller_id, " +
            "buyer_id, " +
            "traded " +
        "FROM onlineshop.item " +
        "WHERE title like ?");
if(s != null) {
    s = "%" + s + "%";
}
stmt.setString(1, s);
ResultSet rs = stmt.executeQuery();
List<Item> items = new ArrayList<Item>();
while(rs.next()) {

    Item item = new Item();

    Long id = Long.valueOf(rs.getLong("id"));
    item.setId(id);

    String title = rs.getString("title");
    item.setTitle(title);

    String description =
        rs.getString("description");
    item.setDescription(description);

    double price = rs.getDouble("price");
    if(price != 0) {
        item.setPrice(
            Double.valueOf(price));
    }

    long seller_id = rs.getLong("seller_id");
    if(seller_id != 0) {
        item.setSeller_id(
            Long.valueOf(seller_id));
    }
```



```
    long buyer_id = rs.getLong("buyer_id");
    if(buyer_id != 0) {
        item.setBuyer_id(Long.valueOf(buyer_id));
    }

    Timestamp ts = rs.getTimestamp("traded");
    if(ts != null) {
        Date traded =
            new Date(ts.getTime());
        item.setTraded(traded);
    }

    items.add(item);
  }
  con.close();
  return items;
 }
}
```

Listing 7.24 »SearchServlet.java«

Der Controller für die Fotos (FotoServlet.java)

Damit die Fotos in der *search.jsp* angezeigt werden können, programmieren wir eine weitere Komponente, die wir `FotoServlet.java` nennen. Dieses Servlet nimmt die Anfrage für das Foto in einem GET-Request entgegen. Nachdem das Foto aus der Datenbank geladen wurde, wird es Byte für Byte in den Ausgabestrom geschrieben.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Blob;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import javax.naming.InitialContext;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
```

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

    @Resource
    private DataSource ds;

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

        try {
            String id = request.getParameter("id");

            Connection con = ds.getConnection();
            PreparedStatement stmt =
                con.prepareStatement(
                "SELECT " +
                "foto " +
                "FROM onlineshop.item " +
                "WHERE id = ?");
            stmt.setLong(1, Long.parseLong(id));
            ResultSet rs = stmt.executeQuery();
            if(rs.next()) {
                Blob foto = rs.getBlob("foto");
                response.reset();
                int length = (int) foto.length();
                response.setHeader(
                    "Content-Length",
                    String.valueOf(length));
                InputStream in =
                    foto.getBinaryStream();
                final int bufferSize = 256;
                byte[] buffer =
                    new byte[bufferSize];

                ServletOutputStream out =
                    response.getOutputStream();
                while (
                    (length = in.read(buffer))
                        != -1
                    ) {
                    out.write(buffer, 0, length);
```



```
                }
                in.close();
                out.flush();
                foto = null;
            }
            stmt.close();
            con.close();
        } catch(Exception ex) {
            throw new ServletException(
                ex.getMessage());
        }
    }
}
```

Listing 7.25 »FotoServlet.java«

7.8 Den Onlineshop fertigstellen

Der Onlineshop ist schon fast fertig programmiert. Und obwohl die *search.jsp* für jeden Artikel ein HTML-Formular bereithält, fehlt dennoch die Möglichkeit, Artikel zu kaufen. Wir brauchen also eine Komponente, die den Speicherwunsch eines Kunden verwirklicht. Hierfür werden wir eine weitere Web-Komponente programmieren, die wir `BuyServlet` nennen werden. Das Servlet wird den Kauf eines Artikels realisieren, indem der Käufer und ein Zeitstempel beim betroffenen Datensatz hinterlegt werden.

Abbildung 7.29 zeigt den Programmablauf, der beim Kauf eines Artikels durchlaufen wird.

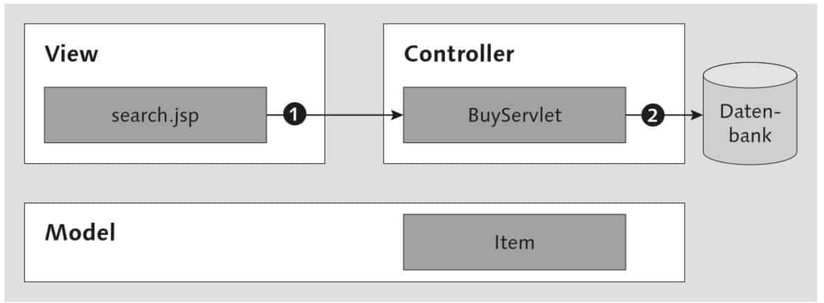


Abbildung 7.29 Der Programmablauf beim Kaufen eines Artikels

7.8.1 Die View (search.jsp)

In der *search.jsp* fügen wir einen zusätzlichen Abschnitt hinzu. In diesem Abschnitt werden wir mit der Expression Language eine Fallunterscheidung durchführen. Sollte der Artikel

noch nicht verkauft worden sein, werden wir den Button für das Verkaufen des Artikels anzeigen. Gleichzeitig werden wir ein INPUT-Element des Typs "Hidden" beifügen, das die ID des Artikels mitführen soll.

```
<%@ include file="head.jspf" %>
<article>
    <section>
        <form action="search" method="post">
        <fieldset>
        <legend>Suchen</legend>
        <table>
            <tbody>
            <tr>
                <th>
                    <label for="search">
                        Suche:
                    </label>
                </th>
                <td>
                    <input
                        type="text"
                        name="search"
                        size="40"
                        maxlength="40"
                        title="Suchtext"
                        placeholder=
                        "Suchtext eingeben"
                        >
                </td>
                <td>
                    <input type="submit">
                    <input type="reset">
                </td>
            </tr>
        </tbody>
        </table>
        </fieldset>
        </form>
    </section>
</article>
<c:forEach var="item" items="${items}">
<article>
    <section>
```



```
<form action="buy" method="post">
<fieldset>
<legend>ID: ${item.id}</legend>
<h2>${item.title}</h2>
<p>${item.description}</p>
 <p>Preis: ${item.price} Euro</p>
<c:if test="${not empty customer}">
    <c:choose>
    <c:when test="${empty item.traded}">
        <input
            type="hidden"
            name="item_id"
            value="${item.id}">
        <input
            type="submit"
            value="Kaufen"/>
    </c:when>
    <c:otherwise>
        <b>Verkauft am ${item.traded}
            an ${item.buyer_id}</b>
    </c:otherwise>
    </c:choose>
</c:if>
<aside>
    <p><img src="foto?id=${item.id}">
</aside>
</fieldset>
</form>
    </section>
</article>
</c:forEach>
<%@ include file="footer.jspf" %>
```

Listing 7.26 »search.jsp«

7.8.2 Der Controller (BuyServlet.java)

Die Klasse `BuyServlet` nimmt die ID des Artikels als Parameter entgegen. Die ID des Kunden haben wir ja bereits, denn die sollte in der HTTP-Sitzung gespeichert sein. Zur Persistierung fügen wir eine Methode mit dem Namen `update()` hinzu. In dieser Methode wird die ID des Käufers im Datenfeld `BUYER_ID` gespeichert. Außerdem werden wir auch einen Zeitstempel im Datenfeld `TRADED` hinterlegen, damit wir dem Verkäufer und dem Käufer mitteilen können, wann der Kauf getätigt worden ist.

```
package de.java2enterprise.onlineshop;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;

import javax.naming.InitialContext;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import javax.sql.DataSource;
import de.java2enterprise.onlineshop.model.Customer;

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

    @Resource
    private DataSource ds;

    public void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        Long item_id =
            Long.parseLong(
                request.getParameter("item_id"));
        HttpSession session = request.getSession();
        Customer customer =
            (Customer)
                session.getAttribute("customer");

        try {
            update(item_id, customer.getId());
        } catch (Exception e) {
            throw new ServletException(e.getMessage());
        }
        RequestDispatcher dispatcher =
```



```
                request.getRequestDispatcher(
                    "buy.jsp");
        dispatcher.forward(request, response);
    }

    public void update(Long item_id, Long buyer_id)
            throws Exception {
        Connection con = ds.getConnection();
        PreparedStatement stmt = con.prepareStatement(
            "UPDATE onlineshop.item " +
            "SET buyer_id = ?, " +
            "traded = SYSTIMESTAMP " +
            "WHERE id = ?"
        );
        stmt.setLong(1, buyer_id);
        stmt.setLong(2, item_id);
        stmt.executeUpdate();
        con.close();
    }
}
```

Listing 7.27 »BuyServlet.java«

Auf einen Blick

1 Überblick 33

2 Der Java EE Server 77

3 Planung und Entwurf 149

4 Servlets 201

5 Java Server Pages 291

6 Die relationale Datenbank 375

7 JDBC 433

8 Die Java Persistence API 509

9 Java Server Faces 701

10 Enterprise JavaBeans 901

Inhalt

Geleitwort des Fachgutachters 21
Über dieses Buch 23

1 Überblick 33

1.1 Der Java-EE-7-Standard 33
1.1.1 Wer definiert den Java-EE-Standard? 35
1.1.2 Die Java-EE-Spezifikation 36
1.1.3 Java EE Server und Java-EE-Profile 36
1.1.4 Komponenten und Container 37
1.1.5 Die Datenhaltung 38
1.1.6 Anwendungsszenarien 39

1.2 Die Java-EE-Software-Architektur 42
1.2.1 Das MVC-Entwurfsmuster 43
1.2.2 Die mehrschichtige Software-Architektur 43

1.3 Die Technologien von Java EE 7 46
1.3.1 Die Web-Technologien des Java-EE-7-Standards 50
1.3.2 Die Enterprise-Technologien des Java-EE-7-Standards 65
1.3.3 Webservices-Technologien 72

1.4 Los geht's mit der Praxis 73

2 Der Java EE Server 77

2.1 Die Installation 77
2.1.1 Das JDK 7 installieren 77
2.1.2 »GlassFish Server Open Source Edition 4« herunterladen 78
2.1.3 Die Verzeichnisse von GlassFish 4 81

2.2 Domänen und Instanzen 83
2.2.1 Domänen 83
2.2.2 Instanzen 83

2.3 Die Textkonsole 85
2.3.1 Eine Domäne erzeugen 87
2.3.2 Der Start einer Domäne 90

2.4 Der DAS ... 92
2.4.1 Die Bedienung des DAS über »asadmin« ... 92
2.4.2 Die Bedienung des DAS über einen Webbrowser ... 95

2.5 Extra-Abschnitt: UTF-8 ... 105
2.5.1 ANSI ... 105
2.5.2 ASCII ... 105
2.5.3 ISO 8859-1 ... 106
2.5.4 Unicode ... 106
2.5.5 UTF-16 ... 107
2.5.6 UTF-8 ... 107
2.5.7 Eine HTML-Seite mit UTF-8-Merkmalen erzeugen ... 108

2.6 Eine Java-EE-Anwendung deployen ... 109
2.6.1 Der Aufbau einer Java-EE-Anwendung ... 110
2.6.2 Java-EE-Module ... 110
2.6.3 Die Enterprise-Archiv-Datei ... 112
2.6.4 Die Erstellung der Web-Komponente (mit UTF-8) ... 113
2.6.5 Die Erzeugung des Web-Moduls und der ».ear«-Datei ... 115
2.6.6 Die Installation ... 115

2.7 Eclipse einsetzen ... 119
2.7.1 Eclipse installieren ... 121
2.7.2 Eclipse auf UTF-8 umstellen ... 122
2.7.3 Die Installation des GlassFish-Plug-ins ... 125
2.7.4 Eine Java-EE-Anwendung in Eclipse erzeugen ... 130

2.8 Extra-Abschnitt: NetBeans einsetzen ... 140
2.8.1 NetBeans installieren ... 141
2.8.2 NetBeans auf UTF-8 umstellen ... 142
2.8.3 Eine Java-EE-Anwendung in NetBeans ... 143

3 Planung und Entwurf 149

3.1 Die Projektplanung mit Scrum ... 150
3.1.1 Die Anforderung analysieren ... 152
3.1.2 Teilanforderungen ermitteln ... 152
3.1.3 Teilanforderungen priorisieren ... 153
3.1.4 Teilanforderungen umsetzen ... 155

3.2 Die Datentypen für den Entwurf ... 157
3.2.1 Wahrheitswerte ... 158
3.2.2 »Kurze« Ganzzahlen ... 158

3.2.3 »Normale« Ganzzahlen ... 159
3.2.4 »Lange« Ganzzahlen ... 159
3.2.5 Kommazahlen (und Geldbeträge) ... 159
3.2.6 Zeichenketten ... 160
3.2.7 Zeitpunkte ... 160
3.2.8 Große Datenmengen ... 161

3.3 Das User-Story-Diagramm ... 162
3.3.1 Einführung ... 163
3.3.2 User-Story 0 »As a user I want to be welcomed« ... 165
3.3.3 User-Story 1 »As a user I want to register« ... 165
3.3.4 User-Story 2 »As a customer I want to sign in« ... 167
3.3.5 User-Story 3 »As a customer I want to sell items« ... 169
3.3.6 User-Story 4 »As a customer I want to search items« ... 170
3.3.7 User-Story 5 »As a customer I want to buy items« ... 170

3.4 Die UI-Prototypen ... 171
3.4.1 Die Datei »index.html« ... 173
3.4.2 Die Datei »register.html« ... 175
3.4.3 Die Datei »signin.html« ... 178
3.4.4 Die Datei »sell.html« ... 180
3.4.5 Die Datei »search.html« ... 183
3.4.6 Die Datei »styles.css« ... 184

3.5 Das HTTP-Monitoring ... 187
3.5.1 Die Verbindung zum Server herstellen ... 188
3.5.2 Der HTTP-Request ... 188
3.5.3 Die HTTP-Response ... 189
3.5.4 Das Monitoring in Eclipse ... 192
3.5.5 Die Firefox-Entwickler-Symbolleiste ... 194

4 Servlets 201

4.1 Ein technischer Überblick ... 202
4.1.1 Die Arbeitsweise des Web-Containers ... 202
4.1.2 Der Lebenszyklus eines Servlets ... 203
4.1.3 Das HttpServlet ... 204
4.1.4 Die Bekanntmachung und Konfiguration eines Servlets ... 205

4.2 Die Servlet-API ... 209
4.2.1 Die Service-Methoden ... 209
4.2.2 Die Beantwortung eines HTTP-Requests ... 211

4.2.3 Formular-Parameter entgegennehmen ... 212
4.2.4 Die Inhalte der Methode »doPost()« programmieren ... 216
4.2.5 Header-Zeilen ... 218
4.2.6 Die Festlegung der Zeichenenkodierung ... 218
4.2.7 Einen Status-Code setzen ... 219
4.2.8 Eine clientseitige Umleitung (Redirect) herbeiführen ... 220
4.2.9 Eine serverseitige Umleitung (Forward) programmieren ... 223
4.2.10 Das Logging in einem Servlet ... 224

4.3 HTTP-Sitzungen ... 227
4.3.1 Die Dauer einer HTTP-Sitzung ... 227
4.3.2 Wie eine HTTP-Sitzung funktioniert ... 227
4.3.3 Wie man eine HTTP-Sitzung nutzt ... 228
4.3.4 Das MVC-Entwurfsmuster ... 229
4.3.5 Eine JavaBean für das Modell erstellen ... 230
4.3.6 Ein Beispiel programmieren ... 232
4.3.7 Die Gültigkeitsbereiche der Attribute ... 236

4.4 Cookies ... 238
4.4.1 Cookies setzen ... 238
4.4.2 Die Gültigkeitsdauer ... 239
4.4.3 Ein Beispiel mit Cookies ... 239

4.5 Initialisierung und Konfiguration ... 242
4.5.1 Parameter für die gesamte Webanwendung ... 242
4.5.2 Parameter für ein einzelnes Servlet ... 243
4.5.3 Initialisierung mit der Annotation »@WebServlet« ... 246
4.5.4 Externe Ressourcen einlesen ... 247

4.6 Dateien hochladen ... 250
4.6.1 Multipart/Form-Data ... 251
4.6.2 Die Konfiguration des Servlets ... 253
4.6.3 Informationen zu einem Part auslesen ... 253
4.6.4 Hochgeladene Dateien konfigurieren ... 254
4.6.5 Den Wertebereich einlesen ... 256

4.7 Asynchrone Servlets und Nonblocking I/O ... 258
4.7.1 Asynchrone Servlets ... 259
4.7.2 Nonblocking I/O ... 266

4.8 Webfilter ... 269
4.8.1 Die Methoden eines Filters ... 270
4.8.2 Die Konfiguration eines Filters ... 272
4.8.3 Mehrere Filter vor ein Servlet setzen ... 275
4.8.4 Filter mit Umleitung ... 277

4.8.5 Filter überspringen ... 279
4.8.6 Einen Filter in Eclipse erzeugen ... 280

4.9 WebListener ... 284
4.9.1 Die Programmierung eines WebListeners ... 285

5 Java Server Pages ... 291

5.1 Grundlagen ... 292
5.1.1 Die HTML-Seiten des Onlineshops in JSP-Dateien umbenennen ... 292
5.1.2 Das automatisch generierte Servlet ... 293
5.1.3 Eine JSP konfigurieren ... 295

5.2 Direktiven ... 296
5.2.1 Die page-Direktive ... 296
5.2.2 Die taglib-Direktive ... 300
5.2.3 Die include-Direktive ... 300
5.2.4 Den Kopfteil und Fußteil der JSPs in gesonderte JSPF-Dateien auslagern ... 301

5.3 Skripting ... 306
5.3.1 Ausdrücke ... 311
5.3.2 Deklarationen ... 312
5.3.3 Kommentare ... 312

5.4 Aktionselemente ... 313
5.4.1 scriptlet ... 313
5.4.2 text ... 313
5.4.3 plugin ... 314
5.4.4 fallback ... 321
5.4.5 params und param ... 321
5.4.6 include ... 322
5.4.7 forward ... 323
5.4.8 useBean ... 324
5.4.9 getProperty und setProperty ... 325

5.5 JSP Custom Tags ... 328
5.5.1 Ein »simples« JSP-Custom-Element ... 329
5.5.2 Tag-Dateien programmieren ... 334

5.6 JSP-EL ... 336
5.6.1 Grundlegende Elemente ... 337
5.6.2 String-Operatoren ... 340
5.6.3 Relationale Operatoren ... 340

5.6.4 Der empty-Operator ... 340
5.6.5 Der konditionale Operator ... 341
5.6.6 Klammern und Semikolon ... 341
5.6.7 Literale ... 341
5.6.8 Implizite Objekte ... 342
5.6.9 Der Zugriff auf eine JavaBean über den Gültigkeitsbereich ... 346
5.6.10 Funktionen ... 347

5.7 Die Java Standard Tag Library ... 347
5.7.1 Anwendungsserver ohne JSTL ... 348
5.7.2 core – Die JSTL-Kernfunktionalität ... 350
5.7.3 fmt – Formatierungen ... 354
5.7.4 functions – Funktionen ... 365
5.7.5 sql – Datenbankzugriffe ... 368
5.7.6 xml – Die Verarbeitung von XML-Strukturen ... 369

6 Die relationale Datenbank 375

6.1 Die Einrichtung der Oracle Database ... 376
6.1.1 Die Installation ... 376
6.1.2 Sprache, Land und Zeichensatz ... 381
6.1.3 Die SQL-Befehlszeile einsetzen ... 382
6.1.4 Transaktionen ... 384
6.1.5 Der Isolation-Level ... 385
6.1.6 Die Änderung des Oracle-HTTP-Ports ... 387

6.2 Die Organisation der Oracle Database ... 390
6.2.1 Objekte ... 390
6.2.2 Tablespaces ... 390
6.2.3 Einen Datenbankbenutzer erzeugen ... 392
6.2.4 Das Berechtigungskonzept ... 393

6.3 Performance-Tuning für die Java-EE-Anwendung ... 396
6.3.1 Datenbank-Sitzungen ... 396
6.3.2 Mit PROCESSES indirekt auch die SESSIONS steuern ... 397

6.4 Die Erstellung der Datenbanktabellen ... 398
6.4.1 SQL- und Datenbankkenntnisse ... 399
6.4.2 Die Erstellung des Datenmodells ... 399
6.4.3 Die speziellen SQL-Datentypen der Oracle Database ... 401
6.4.4 Datenbanktabellen erzeugen, löschen und ändern ... 409
6.4.5 Bedingungen hinzufügen ... 410

6.4.6 Daten hinzufügen, ändern und löschen ... 413
6.4.7 Daten selektieren ... 415
6.4.8 Daten filtern ... 416
6.4.9 Gruppierungsfunktionen ... 418
6.4.10 Zeichenketten verarbeiten ... 419
6.4.11 Sortieren ... 421
6.4.12 Mehrere Datenbanktabellen verbinden ... 421
6.4.13 Unterabfragen ... 424
6.4.14 Primärschlüssel automatisch generieren ... 425

6.5 Die Indizierung ... 428

6.6 Das komplette Onlineshop-Beispiel ... 428

7 JDBC 433

7.1 Der direkte Zugriff per JDBC ... 433
7.1.1 Die JDBC-Treibertypen ... 434
7.1.2 Der JDBC-Treiber für die Oracle Database ... 435
7.1.3 Die Driver Definition in Eclipse ... 435
7.1.4 Der »Data Source Explorer« in Eclipse ... 437
7.1.5 Die Driver Definition im Java-Projekt verwenden ... 442
7.1.6 Der Erstellung einer Datenbankverbindung ... 443

7.2 Der Zugriff über den Java EE Server ... 447
7.2.1 Die JVM-Optionsparameter für den Java EE Server ... 448
7.2.2 Erstellung eines JDBC-Connection Pools ... 449
7.2.3 Erstellung einer »Default Data Source« ... 455

7.3 Die Kernfunktionalität der JDBC-API ... 456
7.3.1 Die Datenbankverbindung über den Java EE Server besorgen ... 457
7.3.2 Die Datenquelle durch den Web-Container injizieren lassen ... 458
7.3.3 Das Absetzen einer SQL-Anweisung ... 460

7.4 Das MVC-Entwurfsmuster einsetzen ... 461
7.4.1 Die Speicherung einer Entity ... 462
7.4.2 Die Abfrage von Geschäftsdaten ... 465

7.5 Das objektrelationale Mapping ... 468
7.5.1 Von SQL zu Java ... 469
7.5.2 Die Umwandlung der JDBC-eigenen Datentypen ... 471

7.6 Vorbereitete SQL-Anweisungen ... 475
7.6.1 Die Klasse »PreparedStatement« ... 475

7.6.2 Von Java zu SQL ... 477
7.6.3 Das Abrufen automatisch generierter Schlüssel ... 479

7.7 Binärdaten ... 482
7.7.1 Bilder speichern ... 482
7.7.2 Bilder anzeigen ... 494

7.8 Den Onlineshop fertigstellen ... 501
7.8.1 Die View (search.jsp) ... 501
7.8.2 Der Controller (BuyServlet.java) ... 503

8 Die Java Persistence API 509

8.1 Einführung ... 510
8.1.1 Wofür braucht man ein Persistenz-Framework? ... 510
8.1.2 JPA und Persistence-Provider ... 511
8.1.3 EclipseLink ... 511
8.1.4 Die ersten Arbeitsschritte bei einer einfachen JPA-Anwendung ... 512

8.2 JPA konfigurieren ... 515
8.2.1 Der Aufbau der »persistence.xml« ... 515
8.2.2 Persistenz-Einheiten definieren ... 515
8.2.3 Transaktionen (mit JTA oder RESOURCE_LOCAL) ... 516
8.2.4 Den Bezeichner vom Namensdienst anzeigen ... 517
8.2.5 Mit »description« eine kurze Beschreibung hinzufügen ... 518
8.2.6 Den Persistence-Provider angeben ... 518
8.2.7 Die Verbindungsdaten hinterlegen ... 519

8.3 Den EntityManager kennenlernen ... 520
8.3.1 Die Erzeugung eines EntityManagers ... 520
8.3.2 Der Persistenz-Kontext ... 524
8.3.3 Die Transaktionen ... 524
8.3.4 Die zwei Ausprägungen des Persistenz-Kontextes ... 529
8.3.5 Wann das Persistieren tatsächlich ausgeführt wird ... 529
8.3.6 Die Status im Lebenszyklus einer JPA-Entity ... 530
8.3.7 Die Methoden für die Datenbankoperationen ... 530

8.4 JPA und Eclipse ... 535
8.4.1 Ein JPA-Projekt in Eclipse erzeugen ... 535
8.4.2 Beim Webprojekt mit JPA arbeiten ... 546
8.4.3 Das ORM-Reverse-Engineering ... 551

8.5 JPA-Entities erstellen ... 554
8.5.1 Der Grundaufbau einer JPA-Entity ... 554
8.5.2 Die Konfiguration einer JPA-Entity ... 555
8.5.3 Die zusätzliche Deklaration in der »persistence.xml« ... 558
8.5.4 Benennung des Tabellennamens ... 559
8.5.5 Benennung des Spaltennamens ... 561
8.5.6 Den Zugriffstyp ändern ... 562

8.6 Primärschlüssel definieren ... 566
8.6.1 Identitätsprobleme von JPA-Entities lösen ... 567
8.6.2 Zusammengesetzte Primärschlüssel ... 570
8.6.3 Die automatische Schlüsselgenerierung ... 573

8.7 Singuläre Attribute konfigurieren ... 579
8.7.1 Zeitliche Basic-Attribute ... 580
8.7.2 Große Datenmengen ... 581
8.7.3 Transiente Attribute ... 582

8.8 Assoziationen anfertigen ... 583
8.8.1 Grundlagen ... 583
8.8.2 Die unidirektionale Many-to-one-Assoziation ... 584
8.8.3 Die bidirektionale One-to-many-many-to-one-Assoziation ... 588
8.8.4 Die unidirektionale One-to-many-Assoziation ... 590
8.8.5 Die unidirektionale Many-to-many-Assoziation ... 594
8.8.6 Die bidirektionale Many-to-many-Assoziation ... 596
8.8.7 Die bidirektionale One-to-one-Assoziation ... 597
8.8.8 Die unidirektionale One-to-one-Assoziation ... 598
8.8.9 Kaskadieren ... 599
8.8.10 CascadeType.REMOVE ... 601
8.8.11 »orphanRemoval« ... 603
8.8.12 Fetching-Strategien ... 604
8.8.13 Listen sortieren ... 605

8.9 JPA-Entities in Eclipse erzeugen ... 606
8.9.1 Die Auswahl der Datenbanktabellen ... 608
8.9.2 Assoziationen hinzufügen ... 608
8.9.3 Den erzeugten Quelltext nachbessern ... 614

8.10 JPQL ... 626
8.10.1 Abfragen programmieren ... 626
8.10.2 Mehrere JPA-Entities verbinden ... 633
8.10.3 Gruppierungsfunktionen ... 639
8.10.4 Unterabfragen ... 641

8.10.5 Zeichenketten auswerten ... 642
8.10.6 Fallunterscheidungen ... 644
8.10.7 Datum und Zeit ... 646
8.10.8 Parameter setzen ... 646
8.10.9 Individuelle Ergebnismengen mit dem »new«-Operator ... 649
8.10.10 Über Assoziationsattribute navigieren ... 651
8.10.11 Änderungen und Löschungen ... 651
8.10.12 Named Queries ... 652

8.11 Die Criteria-API ... 654
8.11.1 Eine einfache Abfrage programmieren ... 655
8.11.2 Eine Criteria-Query mit einer WHERE-Bedingung ... 657
8.11.3 Mit dem Canonical Metamodel arbeiten ... 660

8.12 Fortgeschrittene Themen ... 663
8.12.1 Spaltenfelder auf mehrere Klassen verteilen ... 663
8.12.2 Attribute auf mehrere Tabellen verteilen ... 668
8.12.3 Die Spaltenfelder einer separaten Tabelle als Collection beifügen ... 673
8.12.4 Vererbung und Polymorphie ... 675
8.12.5 Konkurrierende Zugriffe steuern ... 686
8.12.6 Native SQL-Abfragen ... 692

9 Java Server Faces 701

9.1 Einführung ... 701
9.1.1 Vergleich mit Servlets und JSPs ... 702

9.2 Ein JSF-Beispiel ... 705
9.2.1 Die Erstellung eines JSF-Projekts ... 706
9.2.2 Die Anpassung der Konfiguration ... 709
9.2.3 Die JSF-Konfigurationsdatei »faces-config.xml« ... 710
9.2.4 Extra-Abschnitt: JSF mit dem Apache Tomcat ... 711
9.2.5 JPA einbeziehen und eine JPA-Entity als Model anlegen ... 712
9.2.6 Eine Backing Bean als Controller programmieren ... 716
9.2.7 Die Erzeugung des Facelets mithilfe von Eclipse ... 728

9.3 JSF-Grundkenntnisse ... 737
9.3.1 Die Arbeitsphasen ... 737
9.3.2 Bedeutende JSF-Klassen für die Backing Bean ... 740
9.3.3 Die Navigation ... 742
9.3.4 Die UI-Komponenten in einem Facelet ... 752

9.4 Meldungen ... 762
9.4.1 Meldungen für eine UI-Komponente ... 762
9.4.2 Globale Meldungen ... 768
9.4.3 Weitere Informationen anhängen ... 770
9.4.4 Project Stages ... 774

9.5 Internationalisierung ... 776
9.5.1 Die Message Bundles ... 776
9.5.2 Die Anzeige der übersetzten Texte ... 778
9.5.3 Internationalisierte Messages in der Backing Bean ... 780

9.6 Die Validierung ... 781
9.6.1 Das Attribut »required« ... 781
9.6.2 Standard-Validation-Tags ... 782
9.6.3 Benutzerdefinierte Validierungen ... 785
9.6.4 Bean-Validation-Annotationen gemäß der JSR-303 ... 791

9.7 Die Konvertierung ... 793
9.7.1 ConvertDateTime ... 793
9.7.2 convertNumber ... 797

9.8 Templating ... 798
9.8.1 Templating-Grundlagen ... 798
9.8.2 Ein umfangreiches Beispiel mithilfe von Eclipse anfertigen ... 802

9.9 CSS und Ressourcen ... 835
9.9.1 Die Bilder des Onlineshops einbinden ... 836
9.9.2 Die CSS-Datei aus dem Buch ... 838
9.9.3 Die Sprache über Länderflaggen ändern ... 840

9.10 Konversationen ... 844
9.10.1 Die Backing Bean programmieren ... 844
9.10.2 Die View-Komponenten erstellen ... 848

9.11 Faces Flows ... 854
9.11.1 Ein Faces-Flows-Beispiel ... 855
9.11.2 Der Aufruf eines zweiten Flows ... 861
9.11.3 Die User-Story 3 als Flow ... 863

9.12 Events und Listener ... 869
9.12.1 ActionEvent ... 871
9.12.2 ValueChanged-Event ... 873
9.12.3 DataModel-Event ... 876
9.12.4 Phase-Event ... 880
9.12.5 System-Event ... 882

9.13 Ajax einsetzen ... 884
9.13.1 Ajax-Grundlagen ... 884
9.13.2 Die »ajax«-Komponente von JSF ... 887
9.14 HTML5-friendly Markup ... 896
9.14.1 Pass-Through-Attribute ... 896
9.14.2 Pass-Through-Elemente ... 898

10 Enterprise JavaBeans 901

10.1 Session-Bean-Grundlagen ... 901
10.1.1 Was ist eine Session Bean? ... 903
10.1.2 Die Namenskonventionen bei Session Beans ... 904
10.1.3 Die Business-Methoden von Session Beans ... 904
10.1.4 Der entfernte und der lokale Aufruf ... 905
10.1.5 Business-Interfaces ... 909
10.1.6 Die optionalen Deployment-Deskriptoren ... 913
10.2 Session Beans mit Eclipse ... 916
10.2.1 Die Erzeugung des EJB-Projekts ... 916
10.2.2 Die Stateless Session Bean erzeugen ... 919
10.2.3 Die Business-Methode programmieren ... 921
10.2.4 Das Deployment ... 922
10.3 Der entfernte EJB-Client ... 924
10.3.1 Die Erzeugung des Application-Client-Projekts ... 924
10.3.2 Das EJB-Modul einbinden ... 926
10.3.3 Die Bibliothek »gf-client.jar« hinzufügen ... 927
10.3.4 Die Verknüpfung über JNDI ... 928
10.3.5 Der JNDI-Lookup für den entfernten Aufruf ... 930
10.3.6 Die »jndi.properties« ... 932
10.4 Der lokale EJB-Client ... 934
10.4.1 Die Erstellung eines lokalen Business-Interface ... 935
10.4.2 Ein Java-Server-Faces-Projekt erstellen ... 936
10.4.3 Eine JSF-Backing Bean ... 939
10.4.4 Ein Facelet hinzufügen ... 940
10.4.5 JPA einsetzen ... 941
10.5 Transaktionen ... 945
10.5.1 Container-managed Transactions ... 945
10.5.2 Bean-managed Transactions ... 950

10.6 Unterschiedliche Session-Bean-Typen ... 952
10.6.1 Stateless Session Bean ... 952
10.6.2 Stateful Session Beans ... 956
10.6.3 Singleton Session Beans ... 963
10.7 Asynchrone Business-Methoden ... 965
10.8 JMS und Message Driven Beans ... 969
10.8.1 JMS-Grundlagen ... 969
10.8.2 Die serverseitige Konfiguration ... 974
10.8.3 Die clientseitige Konfiguration ... 977
10.8.4 Die Interfaces der Classic JMS API ... 984
10.8.5 Ein einfaches Beispiel mit der Classic JMS API ... 986
10.8.6 Das Classic-JMS-Queue-Beispiel optimieren ... 996
10.8.7 Einen JMS-Classic-Topic-Client erstellen ... 1007
10.8.8 Eine Session Bean für den JMS-Versand ... 1018
10.8.9 Die Simplified JMS API ... 1020
10.8.10 Message Driven Beans ... 1026
10.9 Der Timer-Service ... 1039
10.9.1 Einen Timer programmieren ... 1040
10.9.2 Der zeitplangesteuerte Timer ... 1046
10.10 Webservices und EJB ... 1048
10.10.1 JAX-WS Kurzeinführung ... 1049
10.10.2 Ein einfacher JAX-WS-Webservice in drei Schritten ... 1051
10.10.3 Das WSDL-Dokument ... 1057
10.10.4 Der JAX-WS-Webservice-Client ... 1064
10.10.5 Weitere wichtige JAX-WS-Bestandteile ... 1068
10.10.6 Einen File-Uploader erstellen ... 1071
10.10.7 Big Webservices und die Business Process Execution Language ... 1074
10.10.8 EJB und JAX-WS ... 1074
10.10.9 JAX-RS-Kurzeinführung ... 1075
10.10.10 Ein REST-Beispiel mit EJB und JPA ... 1079

Index ... 1087

Index

@Access 564
@Asynchronous 965
@Basic 580
@DiscriminatorColumn 677
@DiscriminatorValue 677
@EJB 908, 1033
@ElementCollection 673
@Embeddable 663
@Entity 556
@EntityResult 696
@FieldResult 696
@GeneratedValue 574, 577
@Id 556
@IdClass 570
@Inheritance 677
@Inject 70
@JoinColumn 586
@Lob 581
@Local 911
@Local → *Business-Interface lokales*
@LocalBean 912
@ManyToMany 595
@ManyToOne 585
@MappedSuperClass 685
@MessageDriven 1028
@MultipartConfig 257
@NamedNativeQuery 696
@OneWay 1071
@PathParam 1080
@PostActivate 962
@PostConstruct 954, 961, 963
@PreDestroy 954, 962–963
@PrePassivate 961
@PrimaryKeyJoinColumn 670
@QueryParam 1080
@Remote 910
@Resource 458
@Schedule 1046
@SecondaryTable 669–670
@SecondaryTables 670
@SequenceGenerator 577
@SqlResultSetMapping 695
@TableGenerator 576
@Temporal 580
@Timeout 1043
@Transient 582
@WebFilter 272
@WebFilter → WebFilter
@WebParam 1069
@WebService 1052
@WebServlet 246, 295
@XmlRootElement 1070–1071, 1082
[INNER] JOIN 634

A

ACID 71
Action-Event 871
Agile Model Driven Development and Extreme Programming 149
Ajax → Asynchronous JavaScript and XML
Aktionselemente 313
fallback 321
forward 323
getProperty 325
include 322
param 321
params 321
plugin 314
scriptlet 313
setProperty 325
text 313
useBean 324
AL16UTF16 381
AL32UTF8 381
allocationSize 577
AMDD/XP → Agile Model Driven Development and Extreme Programming
American National Standards Institute 105, 381
ANSI → American National Standards Institute
ANSI-SQL 385
Apache CXF 1049
Application 742
Application-managed 521
ASCII 105
Assoziationen 514, 583
anfertigen 583
Asynchrone Servlets 258–259
asyncSupported 260
complete() 261
start() 261
startAsync() 261

Asynchronous JavaScript and XML ... 49
Atomicity ... 71
Attribute
singuläre ... 579
AUTO_INCREMENT ... 425
AUTOCOMMIT ... 384
autoflush ... 299
AUTOINCREMENT ... 425
Automatische Schlüsselgenerierung ... 573
AVG ... 640
Axis 2 ... 1049

B

Backing Beans ... 45, 704, 716
Basic-Attribute ... 514
Batch Applications for the Java Platform ... 46
Bean-managed Transactions ... 950
Bean-Validation ... 791
Berechtigungsrollen ... 395
Bidirektionale Many-to-many-Assoziation ... 596
Bidirektionale One-to-many-many-to-one-Assoziation ... 588
Bidirektionale One-to-one-Assoziation ... 597
Big Webservices ... 1074
Bild
anzeigen ... 494
speichern ... 482
verkleinern ... 491
BOM → Byte-Order-Mark
BPEL → Business Process Execution Language
buffer ... 299
Business Process Execution Language ... 1074
Business-Interface
lokales ... 911
Business-Layer → Business-Schicht
Business-Schicht ... 45
Button ... 757
Byte-Order-Mark ... 107

C

Canonical Metamodel ... 510, 660
CascadeType.ALL ... 601
CascadeType.DETACH ... 600
CascadeType.MERGE ... 600
CascadeType.PERSIST ... 600
CascadeType.REFRESH ... 600
CascadeType.REMOVE ... 600–601
CASE ... 645
CCI → Common Client Interface
CDI → Context and Dependency Injection (for Java)
Celtix ... 1049
CGI → Common Gateway Interface
Classic JMS API ... 969, 986
CLI ... 34
Client-Layer → Präsentationsschicht
Cloud-Computing ... 46
COALESCE ... 645
Code-First ... 1051
Command-Button ... 754
Command-Link ... 754
COMMIT ... 384
Common Client Interface ... 38
Common Gateway Interface ... 34, 51
Common Object Request Broker Architecture ... 41
Concurrency Utilities for Java EE ... 46
ConnectionFactory ... 973
Connection-Pool ... 448, 450
ConnectionPoolDataSource ... 446
ConnectionPooledDataSource ... 450
Consistency ... 71
CONSTRAINT ... 411
Container ... 37
Container-managed ... 521
Container-managed Transactions ... 945
Content-Type ... 109
contentType ... 296–297
Context and Dependency Injection ... 42
Context and Dependency Injection for Java ... 68
Contract-First ... 1051
Control Frame ... 60
Convention Over Configuration ... 49
ConversationScoped ... 844
ConvertDateTime ... 793
convertNumber ... 797
Cookies ... 238
Gültigkeitsdauer ... 239
CORBA → Common Object Request Broker Architecture
COUNT [DISTINCT] ... 639
Criteria-API ... 654
CriteriaBuilder ... 656
CriteriaDelete ... 656
CriteriaQuery ... 656
CriteriaUpdate ... 656
Cross-Cutting-Concerns ... 269
CRUD ... 409, 446
CSS3 ... 49, 184

CURRENT_DATE ... 646
CURRENT_TIME ... 646
CURRENT_TIMESTAMP ... 646

D

DAS → Domain Administration Server
Data Frame ... 60
Data Source Explorer ... 434, 437
Database Management System ... 375
DataModel-Event ... 876
DataSource ... 446, 450
Dateien hochladen ... 250
Wertebereich einlesen ... 256
Datenbank
erstellen ... 398
relationale ... 375
Datentypen ... 157
Datum und Zeit ... 646
DBMS → Database Management System
Default Data Source ... 455
deferredSyntaxAll ... 300
Dependency Injection ... 69, 459
Deployment ... 110
Deployment-Deskriptor ... 205, 242, 296, 913
DETACHED ... 530
Direktiven → JSP-Direktiven
DIRTY READ ... 686
Dirty Read ... 386
Domain Administration Server ... 92
Domänenmodell ... 399
Driver ... 450
Driver Definition ... 433, 436, 442
DriverManager ... 444
Durability ... 71
DurableConsumer ... 1012

E

Eclipse ... 119
Filter ... 280
GlassFish Tools ... 126
GlassFish-Plug-in ... 125
installieren ... 121
TCP/IP Monitor ... 194
UTF-8 ... 122
EclipseLink ... 510, 512
EIS → Enterprise Information System
EJB Entity Beans ... 47
EJB QL ... 47
EJB → Enterprise JavaBeans
EJB-Container ... 37, 40
ejb-jar.xml ... 914
EJB-Komponente ... 37
EJB-Komponentenarchitektur ... 902
EL → JSP-EL
Enterprise Information System ... 38
Enterprise Java Beans ... 37, 66, 901
Timer-Service ... 1039
Transaktionen ... 945
Enterprise Service Bus ... 1049
Entity Relationship Model ... 375
EntityManager ... 511, 514, 520
EntityManagerFactory ... 521
ERM → Entity Relationship Model
errorPage ... 297
ESB → Enterprise Service Bus
Expression Language → JSP-EL
EXTENDED (Persistenz-Kontext) ... 529
extends ... 300
ExternalContext ... 741

F

Facelet Composition Page ... 800
Facelets ... 703
Facelet-Templates ... 798
Faces Flows ... 55, 854
FacesContext ... 740
Facet ... 537
FETCH JOIN ... 638
Fetching-Strategien ... 604
FetchType.EAGER ... 604
FetchType.LAZY ... 604
Field-Annotationen ... 563
FilterChain → WebFilter
Firefox-Entwickler-Symbolleiste ... 194
FOREIGN KEY ... 411
forward ... 224
Forward → Servlets
FTP ... 1050

G

Geschäftskritische Unternehmensanwendungen ... 34
GlassFish
Admin Service ... 102
Administration Console ... 95

Administratorkennwort setzen/ändern ... 96
adminport ... 87
asadmin ... 86
Cluster-Instanz ... 84
Default-Instanz ... 92
Domäne ... 87
Garbage Collector ... 97
Heap ... 97
HTTP-Listener ... 98
IIOP-Listener ... 98
instanceport ... 87
IP-Adresse ... 98
IPv6 ... 99
JMX ... 102
JVM-Einstellungen ... 97
Konfigurationsinstanz ... 84
Local Commands ... 92
Logging ... 94
Max. Threadpoolgröße ... 101
MBean ... 102
Min. Threadpoolgröße ... 101
Minimalanforderung ... 97
Monitoring ... 103
orb-listener ... 98
portbase ... 88
Remote Commands ... 93
server ... 92
Serverfarm ... 85
Standalone-Instanz ... 84
Textkonsole ... 85
Thread-Pool ... 101
GlassFish Server Open Source Edition 4 ... 78
GRANT ... 394
Große Datenmengen ... 581
GROUP BY ... [HAVING] ... 640
Gruppierungsfunktionen ... 418, 639
Gültigkeitsbereiche ... 236

H

Hibernate ... 68, 510
Hot Deployment ... 217
HTML5 ... 47, 49, 55
HTML5-friendly Markup ... 896
http-equiv ... 109
HTTP-Monitoring ... 187
HTTP-Request ... 188
HTTP-Response ... 189
HttpServlet ... 51, 204
HttpSession ... 228
getAttribute() ... 228
jSessionId ... 228
session-timeout ... 227
setAttribute() ... 228
HTTP-Sitzungen → HttpSession

I

IDENTITY ... 425
IDL → Interface Definition Language
IIOP → Internet Inter ORB Protocol
Image Packaging System ... 83
Implizite Objekte ... 306
application ... 308
config ... 310
out ... 306
page ... 308
pageContext ... 308
request ... 307
response ... 307
session ... 307
import ... 299
IN ... 636
include ... 224
include-Direktive ... 296, 300
Indizierung ... 428
info ... 298
initialValue ... 577
Interface Definition Language ... 41
Internet Inter ORB Protocol ... 41
Introspektieren ... 450
Inversion of Control ... 69, 459
IPS → Image Packaging System
isELIgnored ... 300
isErrorPage ... 298
ISO 8859-1 ... 104–106
Isolation ... 71
Isolation-Level ... 385
isThreadSafe ... 298

J

J2EE13Compliant ... 449
Java API for JSON Processing ... 46, 64
Java API for RESTful Web Services ... 72–73, 1048, 1075
@DELETE ... 1077
@GET ... 1077

@POST ... 1077
@PUT ... 1077
Java API for Web Services over SOAP ... 72
Java API for WebSockets ... 46, 56
Java API for XML ... 1048
Java Architecture for XML Binding API ... 1070
Java Community Process ... 35
Java Database Connectivity → JDBC
Java EE Server ... 77
Java Management Extension ... 102
Java Message Service ... 67, 969
AUTO_ACKNOWLEDGE ... 989
ByteMessage ... 990
CLIENT_ACKNOWLEDGE ... 989
DUPS_OK_ACKNOWLEDGE ... 989
DurableConsumer ... 1012
JMSCorrelationID ... 1000
JMSDeliveryMode ... 1000
JMSDeliveryTime ... 1001
JMSDestination ... 1000
JMSExpiration ... 1001
JMSMessageID ... 1000
JMSPriority ... 1001
JMSRedelivered ... 1001
JMSReplyTo ... 1000
JMSTimestamp ... 1001
JMSType ... 1001
Kommunikationsmodelle ... 973
MapMessage ... 990
MessageListener ... 998
ObjectMessage ... 990
onMessage() ... 999
QueueBrowser ... 998
receive() ... 997
receiveNoWait() ... 997
SESSION_TRANSACTED ... 989
SharedConsumer ... 1017
SharedDurableConsumer ... 1018
Simplified JMS API ... 1020
StreamMessage ... 990
TextMessage ... 990
Java Naming and Directory Interface ... 39
Java Persistence API ... 65
Fortgeschrittene Themen ... 663
Java Persistence Query Language ... 626
Java Platform Enterprise Edition 7 Specification ... 36
Java Remote Method Protocol ... 66
Java Server Faces ... 54, 701
Ajax einsetzen ... 884
APPLY_REQUEST_VALUES ... 739
Benutzerdefinierte Validierungen ... 785
CSS und Ressourcen ... 835
Die Arbeitsphasen ... 737
Die Navigation ... 742
Events und Listener ... 869
Faces Flows ... 854
Forward und Redirect ... 749
Globale Meldungen ... 768
Internationalisierung ... 776
INVOKE_APPLICATION ... 739
Konfigurationsdatei ... 747
Konversationen ... 844
Meldungen ... 762
mit dem Apache Tomcat ... 711
PROCESS_VALIDATIONS ... 739
Project Stages ... 774
RESTORE_VIEW ... 738
Templating ... 798
UPDATE_MODEL_VALUES ... 739
Validierung ... 781
Java Server Pages ... 291
Aktionselemente ... 291
Ausdrücke ... 311
Deklarationen ... 312
Direktiven ... 291
Expression Language ... 53, 336
Kommentare ... 312
Standard Tag Library ... 54
Java Specification Request ... 35
Java Standard Tag Library ... 347
Anwendungsserver ohne JSTL ... 348
core ... 350
fmt ... 354
functions ... 365
sql ... 368
xml ... 369
Java Transaction API ... 70
Java-EE-Deployment ... 47
Java-EE-Module ... 110
Java-EE-Profile ... 36
Java-EE-Spezifikation ... 36
Java-EE-Web-Profil ... 37
JAXB ... 1070
JAXB → Java Architecture for XML Binding API
JAX-R ... 47
JAX-RPC ... 47, 1049
JAX-RS → Java API for RESTful Web Services
JAX-WS → Java API for Web Services over SOAP
JAX-WS → Java API for XML
JAX-WS RI ... 1049
JCache ... 46
JConsole ... 103

JCP → Java Community Process
JDBC 38, 433
Driver Templates 436
JDBC-ODBC-Bridge 434
KPRB Driver 435
Native API Driver 434
Network Protocol Driver 434
OCI-Treiber 435
Oracle Thin Driver 435
Pure Java Driver 435
Treibertypen 434
Verbindungsprofil 439
JDBC-Treiber 540
JDBC-URL 453
Jersey 1075
JMS API 67
JMS Provider 970
JMS → Java Message Service
JMS Service Provider Interface 67, 970
JMS SPI → JMS Service Provider Interface
JMSConsumer 1020
JMSContext 1020
JMSProducer 1020
JMX → Java Management Extension
JNDI → Java Naming and Directory Interface
JOIN 634
JOINED 675
JPA → Java Persistence API
JPA-Entities 514, 554
Identitätsprobleme lösen 567
Lebenszyklus 530
JPQL → Java Persistence Query Language
JRMP → Java Remote Method Protocol
JSF → Java Server Faces
JSON → Java API for JSON Processing
JSON-P → Java API for JSON Processing
JSP 2.3 52
JSP Custom Tags 328
SimpleTagSupport 330
JSP → Java Server Pages
JSP-Aktionselemente 291
JSP-Direktiven 291
JSP-EL 336
empty-Operator 340
Funktionen 347
Grundelemente 337
Gültigkeitsbereich 346
implizite Objekte 342
Klammern und Semikolon 341
konditionaler Operator 341
Literale 341
relationale Operatoren 340
String-Operatoren 340
JSPF 301
JSP-Skripting-Elemente 291
JSP-Tag-Datei 334
JSR → Java Specification Request
JSTL → Java Server Pages Standard Tag Library
JSTL → Java Standard Tag Library
JTA → Java Transaction API
JUL 225
JUnit 225

K

Kardinalität 583
Kaskadieren 599
Komponenten 37
Korruptionssicherheit 35

L

language 296–297
Lastsicherheit 34
Lebenszyklus einer JPA-Entity 530
LEFT [OUTER] JOIN 636
LENGTH (s) 643
LIKE 642
LOCATE (s, ges [, start]) 643
Locking-Modi 689
Log4J 225
Logging 94
Long Polling 58
Loopback-IP-Adresse 91, 99
LOWER und UPPER 643

M

MANAGED 530
ManyToMany 595
ManyToOne 584, 588
MAX und MIN 639
Message Broker 969
Message Driven Beans 41, 67, 901, 1026
Message Oriented Middleware 969
Microsoft .NET 66
Model-View-Controller 42–43, 229, 461
Multilingual Plane → UTF-16
MVC-Entwurfsmuster → Model-View-Controller

N

Named Queries 652
Native SQL-Abfragen 692
NetBeans 140
UTF-8 143
NLS_LANG 381
No-Interface View 912
Non Repeatable Read 386
Nonblocking I/O 52, 258, 266
onAllDataRead() 267
onDataAvailable() 267
onError() 267
NULLIF 645

O

Object Management Group 399
Object Oriented Analysis and Design 48, 164
Object Request Broker 66, 72
OMG → Object Management Group
ON DELETE CASCADE 599
ON DELETE NO ACTION 599
ON DELETE SET NULL 599
OneToMany 588
OOAD → Object Oriented Analysis and Design
Open MQ 82, 970
Open Source 68
Opening Handshake Request 59
Opening Handshake Response 60
OpenJPA 510
OPTIMISTIC 689
OPTIMISTIC_FORCE_INCREMENT 689
ORACLE_SID 381
OracleDriver 444
OracleXE 375, 377
Berechtigungskonzept 393
Datenbankbenutzer 392
DBA 396
DEFAULT TABLESPACE 392
Installation 376
Objekte 390
Oracle-Technology-Network-Lizenz (OTN) 378
Organisation 390
Performance-Tuning 396
Port 387
PRIVILEGE 393
PROCESSES 397
ROLES 393
shutdown immediate 387
spfile 398
SQL-Befehlszeile 382
sqlplus.exe 382
startup 387
ORB → Object Request Broker
ORM 468
orm.xml 557
ORM-Reverse-Engineering 551
orphanRemoval 603
Outcome 747
Output-Link 755

P

page-Direktive 296
autoflush 299
buffer 299
contentType 297
deferredSyntaxAll 300
errorPage 297
extends 300
import 299
info 298
isELIgnored 300
isErrorPage 298
isThreadSafe 298
language 297
pageEncoding 297
session 298
trimDirectiveWhitespace 297
pageEncoding 296–297
Part
getContentType() 253
getName() 253
getSize() 253
getSubmittedFileName() 253
passThrough 897
Pass-Through-Element 55
Perl 34
persistence.xml 515, 558
Persistence-Provider 511, 518, 542
PersistenceUnit 522
Persistenz-Kontext 523–524
EXTENDED 529
TRANSACTIONAL 529
PESSIMISTIC 689
PESSIMISTIC_FORCE_INCREMENT 689
PESSIMISTIC_READ 689

PESSIMISTIC_WRITE ... 689
Phantom-Phänomen ... 386
Phase-Event ... 880
Plain Old Java Object → POJO
Planung und Entwurf ... 149
Point-to-Point Messaging ... 973
Point-to-Point-Kommunikationsmodell ... 68
POJO ... 43, 69, 230
Polling ... 58
Präsentationsschicht ... 45
PreparedStatement ... 480
Primärschlüssel ... 425, 566
automatisch generieren ... 425, 427
definieren ... 566
eingebettete Primärschlüsselklasse ... 572
externe Primärschlüssel-Klasse ... 570
zusammengesetzte ... 570
Primary Cache ... 962
Property-Annotationen ... 564
Publish-and-Subscribe Messaging ... 973
Publish-and-Subscribe-Kommunikationsmodell ... 68
Push-Technologie ... 58

Q

Queue ... 973
QueueConnectionFactory ... 973
QueueReceiver ... 985
QueueSender ... 985
QueueSession ... 985

R

READ COMMITTED ... 386, 686
READ UNCOMMITTED ... 386, 686
Redirect → Servlet
Relationale Datenbank ... 375
Remote Method Invocation ... 66
Remote Procedure Call ... 66
Remote-Business-Interface ... 910
REMOVED ... 530
REPEATABLE READ ... 386, 686
REpresentational State Transfer ... 41, 73
RequestDispatcher ... 223
required ... 781
REST → REpresentional State Transfer
RMI over IIOP ... 41, 906
RMI → Remote Method Invocation
RMI-IIOP → RMI over IIOP
RMI-JRMP ... 66
ROLLBACK ... 385
RPC → Remote Procedure Call

S

SCHEMA ... 390
schema ... 577
Scrapbook ... 441
scriptlet ... 313
Scrum ... 149, 164
Burndown Chart ... 157
Product Backlog ... 155
Product Owner ... 152
Product Requirement ... 152
Projektplanung ... 150
Sprint Backlog ... 156
UI-Prototypen ... 171
User-Story-Diagramm ... 149
Seam ... 68
Secondary Cache ... 962
SEI → Service Endpoint Implementation
SelectBooleanCheckbox ... 757
SelectManyCheckbox ... 761
SelectManyListbox ... 761
SelectManyMenu ... 761
SelectOneListbox ... 759
SelectOneMenu ... 759
SelectOneRadio ... 759
SEQUENCE ... 425
sequenceName ... 577
SERIALIZABLE ... 386, 686
Service Consumer ... 72
Service Endpoint Implementation ... 1049
Service Oriented Architecture ... 72, 1048
Service Provider ... 72
ServletContext ... 248
ServletInputStream ... 268
Servlets ... 51, 201
@WebServlet ... 208
ContentType ... 218
Dateien hochladen ... 250
destroy() ... 204
externe Ressourcen ... 247
Formular-Parameter entgegennehmen ... 212
Forward ... 220
GenericServlet ... 205
Header setzen ... 218
init() ... 203

Konfiguration ... 205
Lebenszyklus ... 203
Load-On-Startup ... 207
Logging ... 224
Multipart/Form-Data ... 251
Redirect ... 220
service() ... 204
Servlet-API ... 209
Status-Code ... 219
Zeichenkodierung ... 218
Servlets 3.1 ... 51
session ... 298
Session Beans ... 41, 901, 903
asynchrone Business-Methoden ... 965
Business-Interface ... 904, 909
Business-Methoden ... 904
Client View ... 907
entfernter Aufruf ... 906
Lebenszyklus ... 953
setFlushMode ... 529
SharedConsumer ... 1017
Short Polling ... 58
Simplified JMS API ... 969
SINGLE_TABLE ... 675
Singleton Session Beans ... 67, 903, 963
Singuläre Attribute ... 579
Skeletons ... 66
Skripting ... 306
application ... 308
out ... 306
page ... 308
request ... 307
session ... 307
SLF4j ... 225
SMTP ... 1050
SOA → Service Oriented Architecture
SOAP ... 41, 1050
Software-Architektur ... 42
SpringSource ... 68
SQL ... 376
Standard-Validation ... 782
Stateful Session Beans ... 67, 903, 956
Lebenszyklus ... 960
Stateless Session Beans ... 67, 903
Lebenszyklus ... 953
Static Metamodel ... 510, 655
Statisches Metamodell → Static Metamodel
Status-Code ... 191
Structured Query Language → SQL
Struts ... 54, 68
Stubs ... 66
SUBSTRING(s, start [, anzahl]) ... 644
SUM ... 640
SYS ... 382
SYSDBA ... 383
SYSTEM ... 382
System-Event ... 882

T

TABLE_PER_CLASS ... 675
Tablespaces ... 390
AUTOEXTEND ON ... 391
Tag Library Descriptor ... 330
taglib-Direktive ... 296, 300
TCP/IP ... 1050
Templating ... 798
Temporale Parameter ... 648
Tomcat ... 40, 711
Topic ... 974
TopicConnectionFactory ... 974
TopicPublisher ... 985
TopicSession ... 985
TopicSubscriber ... 985
TopLink ... 510
TRANSACTION (Persistenz-Kontext) ... 529
Transaktionen ... 516, 524
Transaktionssicherheit ... 35
Transaktionstyp ... 542
Transiente Attribute ... 514, 582
TRIM ([[LEADING | TRAILING | BOTH] z FROM] s) ... 644
trimDirectiveWhitespace ... 297
Tupel ... 375
TypedQuery ... 655–656

U

UDDI ... 1051
UIComponent ... 741
Unicode ... 106
Unidirektionale Many-to-one-Assoziation ... 584
Unidirektionale One-to-one-Assoziation ... 598
Unterabfragen ... 424, 641
UPPER ... 643
User-Storys ... 149
UTF-16 ... 107, 381
UTF-8 ... 104–105, 107, 381

V

validateDoubleRange ... 783
validateLength ... 783
validateLongRange ... 784
validateRegex ... 784
validateRequired ... 782
Value-Changed-Event ... 873
Vererbung und Polymorphie ... 675
Versionierung ... 687

W

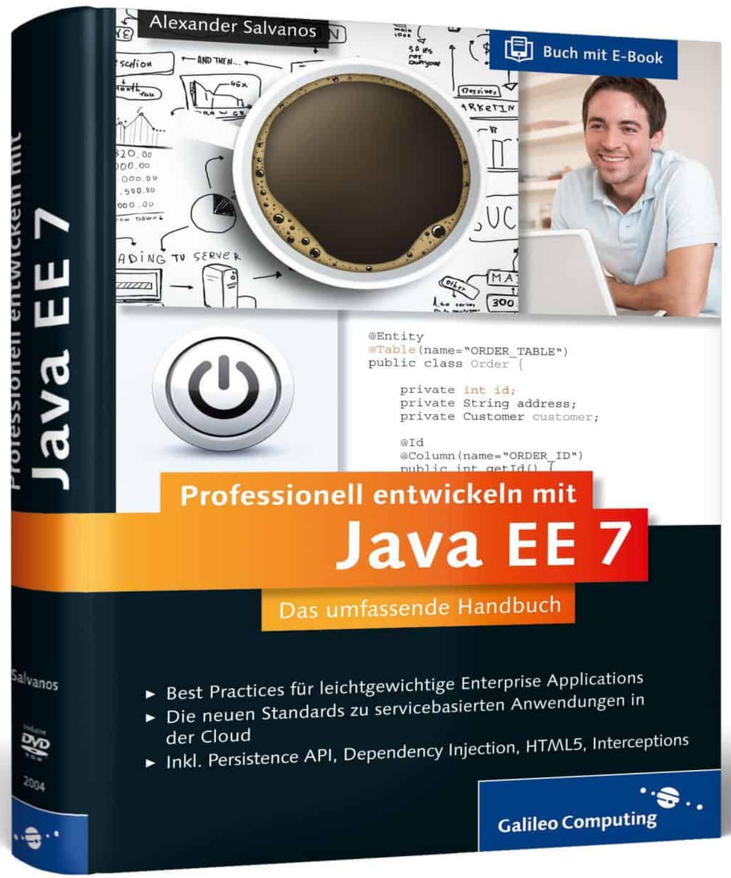
Web Services Description Language ... 72, 1050, 1057
UDDI ... 73
web.xml ... 296
Web-Container ... 40, 202
WebFilter ... 269
destroy() ... 270
DispatcherType ... 280
doFilter() ... 270
init() ... 270
überspringen ... 279
Umleitung ... 277
Web-Komponente ... 37
WebListener ... 284
HttpSessionActivationListener ... 284
HttpSessionAttributListener ... 284
HttpSessionBindingListener ... 284
HttpSessionListener ... 284
ServletContextAttributListener ... 284
ServletContextListener ... 284
ServletRequestAttributeListener ... 285
ServletRequestListener ... 285
Web-Modul ... 111
Webservices ... 72, 1048
Service Consumer ... 72
Service Provider ... 72
WebSockets ... 49, 56
Weld ... 68
Windows-1252 ... 105, 381
WSDL → Web Services Description Language

X

X/Open ... 34
X/Open SQL Call Level-Schnittstelle ... 39
X/Open XA ... 71, 446, 450
XADataSource ... 446, 450
XFire ... 1049
XML-RPC ... 1049

Z

Zeitliche Basic-Attribute ... 580
Zusammengesetzte Primärschlüssel ... 570





Alexander Salvanos kann auf jahrelange Praxiserfahrung als Softwareentwickler und Java-Trainer zurückblicken. In seinen Schulungen gelingt es ihm, auch abstrakte Konzepte wie die Objektorientierte Programmierung anschaulich und verständlich zu vermitteln. Als Mitglied des Java Community Process beteiligt er sich an der Weiterentwicklung von Java 7 und berät Unternehmen beim professionellen Einsatz von modernen Java-Technologien.

Alexander Salvanos

Professionell entwickeln mit Java EE 7 – Das umfassende Handbuch

1.096 Seiten, gebunden, Juni 2014
49,90 Euro, ISBN 978-3-8362-2004-0

www.galileo-press.de/3250

Wir hoffen sehr, dass Ihnen diese Leseprobe gefallen hat. Gerne dürfen Sie diese Leseprobe empfehlen und weitergeben, allerdings nur vollständig mit allen Seiten. Die vorliegende Leseprobe ist in all ihren Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen beim Autor und Verlag.

Teilen Sie Ihre Leseerfahrung mit uns!