HANSER

Leseprobe

Javid Jamae, Peter Johnson

JBoss im Einsatz

Den JBoss Application Server konfigurieren

Übersetzt aus dem Englischen von Dorothea Heymann Reder

ISBN: 978-3-446-41574-4

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-41574-4

sowie im Buchhandel.

3

3 Anwendungen bereitstellen

Die Themen dieses Kapitels:

- Wie Anwendungen bereitgestellt werden
- Wie Klassen geladen werden
- Häufige Bereitstellungsfehler beheben
- Diverse Anwendungen bereitstellen

In gewisser Weise ähnelt JBoss AS einem neuen Haus, das Sie gekauft haben. Das Haus hat Böden, Türen, Fenster, Wände und ein Dach, ebenso wie JBoss AS Services besitzt. Nun mögen alle diese Dinge zwar den Regen abhalten und vor Wind schützen, aber sie verleihen dem Haus keine Atmosphäre, machen es nicht zu einem Heim.

Damit ein Haus ein Zuhause wird, müssen Möbel, Dekorationen und andere Dinge hinzukommen, die jedem Raum Sinn und Persönlichkeit verleihen. Ebenso müssen Sie JBoss AS Anwendungen hinzufügen, um ihm Persönlichkeit, Nutzen und Sinn zu verleihen. So wie ein Tisch, Stühle und Tischwaren ein leeres Zimmer zu einem Esszimmer machen, in dem Sie Ihre Familie und Freunde empfangen können, kann eine gut programmierte Webanwendung, die auf JBoss AS bereitgestellt wird, den Server zu einer einladenden Website machen, die Ihre Kunden gerne besuchen.

Der ganze Zweck eines Anwendungsservers ist es ja, Anwendungen auszuführen. Doch bevor sie ausgeführt werden, müssen diese Anwendungen bereitgestellt werden. Dieses Kapitel beschreibt, welche Anwendungstypen (bei einer lockeren Verwendung dieses Begriffs) bereitgestellt werden können und wie die Bereitstellung auf JBoss AS funktioniert. Doch schauen wir uns als Erstes an, was es bedeutet, eine Anwendung bereitzustellen.

3.1 Bereitstellung verstehen

Eine Bereitstellung verläuft in zwei Phasen. Zuerst benachrichtigen Sie den Anwendungsserver direkt oder indirekt, dass eine Anwendung bereitgestellt werden muss. Als Zweites trifft der Anwendungsserver die notwendigen Maßnahmen, um eine Anwendung gebrauchsfähig zu machen.

JBoss AS verwendet eine Plugin-Bereitstellungsarchitektur, in der getrennte Deployer dafür zuständig sind, verschiedene Typen von Anwendungen bereitzustellen. Das macht die Bereitstellungsarchitektur modular und erleichtert das Definieren neuer Anwendungstypen.

3.1.1 Eine Anwendung bereitstellen

Vielleicht der einfachste Weg, eine Anwendung bereitzustellen, besteht darin, sie in das Verzeichnis *server/xxx/deploy* zu legen. Wenn der Server läuft, untersucht der Deployment-Scanner dieses Verzeichnis regelmäßig, und wenn er neue oder geänderte Anwendungen sieht, stellt er sie bereit. Wenn der Server gerade nicht läuft, wird das Verzeichnis gescannt, sobald er wieder hochgefahren wurde.

Wenn die Anwendung aktualisiert wird, z.B. indem eine neuere Version der Anwendung mit einem jüngeren Zeitstempel in das *deploy*-Verzeichnis kopiert wurde, entfernt der Deployment-Scanner die alte Version, bevor er die neue bereitstellt. Diese Undeploy-Deploy-Aktion hat den Nebeneffekt, dass der aktuelle Zustand der Anwendung verloren geht, einschließlich des Sitzungszustands für aktive Benutzer. Daher sollten Sie ein solches Hot Deployment in einer Produktionsumgebung nur verwenden, wenn Sie absolut sicher sind, keine laufenden Sitzungen zu unterbrechen. Denn wenn Sie für tausend Benutzer mal schnell den Sitzungsstatus löschen, werden Sie entweder haufenweise böse E-Mails oder Telefonanrufe bekommen oder am Ende ganz ohne Kunden dastehen.

Eine Anwendung zu entfernen ist einfach: Sie löschen sie nur aus dem *deploy*-Verzeichnis, und wenn der Deployment-Scanner das nächste Mal läuft, merkt er, dass die Anwendung nicht mehr existiert, und entfernt sie. Alternativ können Sie zum Bereitstellen oder Entfernen einer Anwendung die Operationen `deploy` oder `redeploy` der MBean `jboss.system:service=MainDeployer` verwenden. Um die Methode aufzurufen, rufen Sie entweder die JMX Console oder Twiddle auf. Sie könnten z.B. mit Twiddle eine Datei namens *myapp.ear* aus dem Verzeichnis */some/path* bereitstellen, indem Sie an der Eingabeaufforderung Folgendes eingeben (alles in einer Zeile):

```
twiddle invoke "jboss.system:service=MainDeployer"
 ➥ deploy /some/path/myapp.ear
```

Die bereitzustellende Anwendung muss für den Server zugänglich sein, denn der Anwendungsserver stellt die Anwendung an ihrem aktuellen Speicherort bereit; die Methode kopiert die Anwendung nicht in das *deploy*-Verzeichnis der Serverkonfiguration. Das bedeutet, dass in unserem Beispiel die Anwendung von ihrem Verzeichnis unter */some/path/myapp.ear* bereitgestellt wird. Zusätzlich gilt: Da die Anwendung nicht ins *deploy*-Ver-

zeichnis kopiert wird, steht sie nach einem Neustart des Anwendungsservers nicht mehr zur Verfügung. Verwenden Sie die Methode `undeploy` auf derselben MBean, um eine auf diese Weise bereitgestellte Anwendung zu entfernen.

Wenn Sie den Anwendungsserver mit einem Skript starten, können Sie jederzeit Twiddle-Anweisungen hinzufügen, um die gewünschten Anwendungen bereitzustellen. Zwei positive Bemerkungen über diesen Mechanismus: Sie können ihn auch dann verwenden, wenn das JBoss AS-Installationsverzeichnis schreibgeschützt ist (weil die Anwendung nicht in das *deploy*-Verzeichnis kopiert wird), und Sie können Anwendungen auch dann mit ihm bereitstellen, wenn der Hot Deployer ausgeschaltet wurde.

Hinweis

Aufmerksame Leser denken jetzt vielleicht an den Java Specification Request 88 (JSR-88), der die Bereitstellung von Java EE-Anwendungen definiert. Es wird Sie freuen zu hören, dass JBoss AS JSR-88 unterstützt. Weniger freuen wird es Sie, dass dieser noch einige Probleme hat, die uns daran hindern, ihn zur Nutzung zu empfehlen. Erstens gibt es kein Tool, das JSR-88 verwendet; wenn Sie eine Anwendung mit JSR-88 bereitstellen möchten, müssen Sie schon selbst programmieren. Zweitens werden Anwendungen, die mit JSR-88 bereitgestellt wurden, in das *tmp*-Verzeichnis geladen und bei einem Neustart des Servers nicht wieder bereitgestellt.

3.1.2 Anwendungen verpacken

Wenn Sie eine Anwendung bereitstellen, stellen Sie immer das Paket mit dieser Anwendung bereit. Ein Paket kann entweder eine Archivdatei oder ein expandiertes Verzeichnis sein. Ein Archiv wäre z.B. eine Web Archive(WAR)-Datei oder eine Enterprise Archive(EAR)-Datei. Doch was ist ein expandiertes Verzeichnis? Schauen wir uns ein Beispiel an. Angenommen, Ihre Anwendung besteht aus einer EJB Java Archive(JAR)-Datei und einer WAR-Datei, die eine Webschnittstelle enthält. Diese Dateien können Sie in eine EAR-Datei verpacken, nennen wir sie *myapp.ear*, und diese *myapp.ear*-Datei in das *deploy*-Verzeichnis kopieren. Dies ist ein Beispiel für die Bereitstellung einer Archivdatei. Sie können aber auch die JAR- und die WAR-Dateien in ein Verzeichnis namens *myapp.ear* legen und dieses gesamte Verzeichnis in das *deploy*-Verzeichnis kopieren. Dies ist ein Beispiel für ein expandiertes Verzeichnis. Sie können sogar noch weiter gehen und die WAR-Datei, die JAR Datei oder beide extrahieren. Abbildung 3.1 zeigt für beide Szenarien die Dateien auf dem Laufwerk.

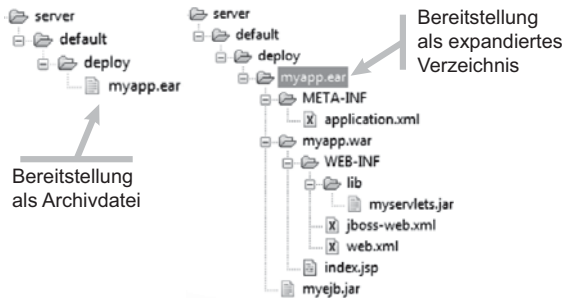

Abbildung 3.1
Bereitstellung als Archivdatei und als expandiertes Verzeichnis

Nun fragen Sie sich vielleicht, welcher Mechanismus der bessere ist. Beide haben gute und schlechte Seiten. Mit einer Archivdatei haben Sie nur eine einzige Datei, um die Sie sich kümmern müssen, und es besteht nicht die Gefahr, dass die Anwendung nur teilweise bereitgestellt wird, weil z.B. eine der Dateien gelöscht wurde.

Ein expandiertes Verzeichnis hat mehrere Vorteile: Erstens sind alle Konfigurationsdateien und Deployment-Dekriptoren im Blickfeld und können einfach bearbeitet werden. Wenn Sie den primären Deskriptor einer Anwendung bearbeiten, z.B. die Datei *application.xml* in einem EAR, dann stellt der Hot Deployer die Anwendung wieder bereit. Tabelle 3.1 listet für jeden Anwendungstyp den primären Deskriptor auf.

Tabelle 3.1 Primäre Deskriptoren für diverse Anwendungstypen

Anwendungstyp	Primärer Deskriptor
WAR	WEB-INF/web.xml
EAR	META-INF/application.xml
SAR	META-INF/jboss-service.xml
JAR	META-INF/ejb-jar.xml
RAR	META-INF/ra.xml

Zweitens können Sie JSPs, Stylesheets und diverse andere Textdateien ändern, und die Anwendung beginnt automatisch, sie zu benutzen (auch wenn der Kunde eventuell für ein Stylesheet oder eine Bilddatei noch auf den Aktualisieren-Button des Browsers klicken müsste, um die Änderung zu sehen). Drittens können Sie ohne Umstände neue Dateien hinzufügen. Vielleicht haben Sie ja ein *doc*-Verzeichnis mit PDF-Dateien und ein Servlet, das anhand des Inhalts des *doc*-Verzeichnisses eine Webseite mit Links auf diese Dateien erstellt. Hier eine neue PDF hinzuzufügen ist einfach; Sie müssen sie nur in dieses Verzeichnis kopieren. Der Nachteil der Bereitstellung als expandiertes Verzeichnis besteht darin, dass Sie nicht nur mit einer, sondern mit vielen Dateien zu tun haben.

Noch etwas ist von Bedeutung: Wenn Sie eine Archivdatei bereitstellen, extrahiert der Deployer sie in das Verzeichnis *server/xxx/tmp/deploy* unter einem generierten Namen, der auf dem Namen der Archivdatei basiert, wie z.B. *myapp28562-exp.ear* für das obige Beispiel. Dieses Verzeichnis enthält die expandierte Version des Archivs, auch wenn die JAR-Dateien so zurückbleiben, wie sie sind, und nicht ausgepackt werden. Aber denken Sie nicht, das sei praktisch und gebe Ihnen das Beste aus beiden Welten, denn bei einem Neustart des Anwendungsservers wird der meiste Inhalt aus dem Verzeichnis *server/xxx/tmp/deploy* gelöscht. Daher sollten Sie sich niemals auf irgendetwas stützen, das im *server/xxx/tmp*-Verzeichnis liegt. Dass es *tmp* heißt, hat schließlich seinen Grund.

3.1.3 Anwendungstypen

Wie schon in der Einführung zu diesem Kapitel gesagt, verwenden wir den Begriff „Anwendung“ relativ weitschweifig. Was ist eine Anwendung? Webster definiert sie als die Verwendung, die einer Sache beschieden ist. Das kann man so verstehen, dass eine An-

wendung jede Verwendung ist, die man dem Anwendungsserver gibt, oder auch alles, das eine nützliche Funktion ausführt, die auf dem Anwendungsserver bereitgestellt werden kann.

Zwei zentrale Anwendungstypen sind Geschäftsanwendungen und Services. Eine Geschäftsanwendung stellt eine Geschäftsfunktion normalerweise für Endanwender zur Verfügung, während ein Service Funktionalität liefert, die andere Anwendungen unterstützt. Oft sind mit dem Begriff Anwendung nur Geschäftsanwendungen gemeint, aber in diesem Kapitel verwenden wir ihn im allgemeinen Sinne.

Die Definition von Webster gibt viel Spielraum zu definieren, was als Anwendung betrachtet wird, aber das ist gut so, weil JBoss AS eine Vielzahl unterschiedlicher Anwendungstypen unterstützt. Wie wird ein Anwendungstyp vom anderen unterschieden? Mithilfe des Suffix' für den Datei- oder Verzeichnisnamen der Anwendung. Tabelle 3.2 listet die verschiedenen Suffixe auf, beschreibt ihren Zweck und zeigt an, in welchem Abschnitt oder Kapitel der Anwendungstyp eingehender behandelt wird.

Tabelle 3.2 Anwendungstypen und ihre Suffixe

Suffix	Anwendungstyp	Siehe
deployer -deployer-beans.xml	Definiert einen Anwendungs-Deployer zur Bereitstellung eines bestimmten Anwendungstyps. Diese Anwendungstypen erscheinen nur im Verzeichnis *server/xxx/deployers.*	Abschnitt 3.1.5
aop -aop.xml	Definiert Aspekte für Klassen, die Funktionalität dieser Klassen erweitern oder neu hinzufügen.	
sar -service.xml	Definiert einen Service, der dem Anwendungsserver Funktionalität hinzufügt.	Kapitel 7 und 8
-jboss-beans.xml	Definiert POJOs für den Microcontainer.	Kapitel 2
rar	Definiert einen Ressourcenadapter, um sich über die Java Connector Architecture (JCA) mit Unternehmensinformationssystemen zu verbinden.	–
-ds.xml	Definiert eine Datenquelle, die für den Zugriff auf Daten in einer Datenbank verwendet wird.	Abschnitt 3.4.1
har	Definiert ein Hibernate-Archiv, das für den Zugriff auf eine Datenbank mittels Hibernate verwendet wird.	Abschnitt 3.4.2
jar	Definiert eine Collection von EJBs, die Geschäftslogik für eine Anwendung beisteuern. Könnte auch eine Klassenbibliothek sein, aber diese wird in der Regel mit anderen Anwendungen mitgeliefert oder in das Verzeichnis *server/xxx/lib* gelegt anstatt in das Verzeichnis *server/xxx/deploy*.	Kapitel 7
zip	Der Deployer untersucht den Inhalt der Zip-Datei, um festzustellen, welchen Typ von Anwendung sie enthält, und stellt sie dann mit dem richtigen Deployer bereit. Enthält die zip-Datei z.B. *WEB-INF/web.xml*, wird die Datei als Webanwendung bereitgestellt. Allerdings ist der	–

(Fortsetzung nächste Seite)

Suffix	Anwendungstyp	Siehe
	Standardkontext für ein WAR, das als Zip-Datei bereitgestellt wurde, der volle Dateiname, z.B. *http://localhgost:8080/someapp.zip*.	
war	Definiert eine Webanwendung, die eine Webschnittstelle für eine Anwendung oder einen Webservice zur Verfügung stellt.	Kapitel 5 und 9
wsr	Ein JBoss-spezifisches Archiv, das einen Webservice definiert. Verwenden Sie dieses Suffix, um den Webservice nach allen anderen WAR-Dateien bereitzustellen.	Kapitel 9
ear	Definiert eine Java EE-Anwendung, die eine Collection von EJBs, WAR-Dateien und Klassenbibliotheken ist.	Kapitel 7
bsh	Definiert einen Service mithilfe eines Bean-Shellskriptes.	–
last	Wird als Verzeichnis (oder Archivdatei) mit bereitzustellenden Anwendungen behandelt, die als letzte in der Reihenfolge bereitgestellt werden, die ihren Suffixen entspricht.	–

Und wie kann der Anwendungsserver bei so vielen Anwendungstypen wissen, in welcher Reihenfolge er sie bereitstellen soll? Diese Frage wollen wir nun beantworten.

3.1.4 Die Reihenfolge der Bereitstellung

Während der Initialisierung, oder wenn der Deployer mehrere Anwendungen bereitstellen soll, werden die Anwendungen je nach Typ in einer bestimmten Reihenfolge bereitgestellt. Zufälligerweise listet Tabelle 3.2 die Anwendungstypen in genau der Reihenfolge auf, in der sie standardmäßig bereitgestellt werden.

Praktischerweise wird das Archiv **.last* zuletzt bereitgestellt. Wenn Sie eine Anwendung haben, die nach allen anderen bereitgestellt werden muss, legen Sie im Verzeichnis *server/xxx/deploy* ein Verzeichnis an, das vielleicht *doit.last* heißen kann, und legen Sie Ihre Anwendung hinein. Dann können Sie sicher sein, dass alle anderen Anwendungen vor dieser bereitgestellt werden.

Der Umgang mit geschachtelten Anwendungen

Sie können auch eine Anwendung erstellen, die weitere, eingeschachtelte Anwendungen enthält. Ein Beispiel dafür ist eine EAR-Datei, die sowohl JAR- als auch WAR-Dateien enthalten kann. Aber darauf sind Sie nicht beschränkt: Sie könnten auch eine JAR-Datei haben, die eine SAR enthält, die ein – na ja, Sie wissen, was ich meine. Wie sehen diese Anwendungen im Matrioschka-Stil in der Bereitstellungsreihenfolge aus?

Hinweis

Eine Matrioschka ist eine russische Puppe (normalerweise aus Holz), in der immer kleinere Puppen eingeschachtelt sind. Im Entwurf bezeichnet man dies als das Matrioschka-Prinzip. Die Verschachtelung von Paketen ist ein ganz ähnliches Konzept.

Zuerst entscheidet der Typ der äußeren Anwendung über die Reihenfolge der Bereitstellung aller Anwendungen auf dieser Ebene. Wenn dann die Zeit kommt, die Matrioschka-

Anwendung bereitzustellen, wird die am tiefsten eingeschachtelte Anwendung als erste bereitgestellt. Befinden sich auf irgendeiner Ebene mehrere Anwendungen, gilt die Reihenfolge der Suffixe. Abbildung 3.2 illustriert die Reihenfolge der Bereitstellung einer fiktiven Menge von Anwendungen. Die Zahlen neben den Anwendungen geben ihre relative Reihenfolge in der Bereitstellung wieder.

Abbildung 3.2
Bereitstellung von Anwendungen im Matrioschka-Stil

Es gibt jedoch Wege, die Reihenfolge der Bereitstellung von Anwendungen in einer Matrioschka-Anwendung außer Kraft zu setzen. So definiert z.B. in einer EAR-Datei die Datei *META-INF/application.xml* die Ordnung für die eingeschachtelten Anwendungen. Spätere Kapitel über Anwendungstypen werden solche Ordnungsmechanismen noch behandeln.

3.1.5 Optionen für die Bereitstellungskonfiguration

Der Deployer wird über die Deskriptordateien *deployers.xml* und *profile.xml* konfiguriert, die beide im Verzeichnis *server/xxx/conf* liegen. Diese Datei definiert mehrere POJOs, die verschiedene Verantwortlichkeiten für die Bereitstellung festlegen. Tabelle 3.3 beschreibt diese POJOs und stellt einige ihrer interessantesten Konfigurationseigenschaften vor.

Tabelle 3.3 Konfigurationseigenschaften von Deployer-POJOs

Bean	Eigenschaft	Beschreibung
`MainDeployer`	`Structural Deployers`	Eine Liste von Beans, die definiert, wie die bereitstellungsfähigen Objekte auf oberster Ebene klassifiziert werden. So sind z.B. eine Reihe von Dateitypen gepackte Archive und somit ein Fall für die `JarStructure`-Bean. Diese Dateien haben Erweiterungen wie .zip, .ear usw. Textdateien, die Services definieren, wie z.B. **-ds.xml* und **-service.xml*, sind für die `FileStructure`-Bean definiert.

(Fortsetzung nächste Seite)

Bean	Eigenschaft	Beschreibung
	`deployers`	Eine Liste der verschiedenen Deployer. Diese behandeln die von den `structuralDeployers` erkannten Dateien. Die Eigenschaft `structuralDeployer` findet alle Dateitypen, die von Interesse sind, während die `deployers`-Eigenschaft die Services findet, die diese Dateitypen bereitstellen.

`DeploymentFilter`	`prefixes`
	`suffixes`
	`matches`
		Stellt fest, welche Dateien oder Verzeichnisse der Deployer ignorieren kann. Die Eigenschaft `matches` erkennt einen vollständigen, einfachen Verzeichnis- oder Dateinamen, und die anderen beiden Eigenschaften erkennen Präfixe und Suffixe. Jede ist eine Liste kommagetrennter Strings. Beachten Sie, dass die Elemente der Liste Dinge wie temporäre Dateien widerspiegeln, die von Editoren oder anderer Software angelegt wurden, sowie Arbeitsdateien oder Verzeichnisse, die von Quellkontrollsystemen wie Subversion verwendet werden, usw.

`VFSDeployment Scanner`	`URIList`	Eine Liste von Speicherorten, die der Deployer nach Anwendungen durchsucht. Ein Beispiel finden Sie im Abschnitt unter dieser Tabelle.
	`URIs`	Wie `URIList`, aber die Liste wird als ein einziger String mit kommagetrennten Werten übergeben.
	`recursiveSearch`	Zeigt an, ob der Scanner rekursiv auch Unterverzeichnisse nach Anwendungen durchsuchen soll. Wird nur verwendet, wenn ein Verzeichnisname keinen Punkt enthält. So liegen z.B. einige Deskriptoren für Messaging-Services im Verzeichnis *server/xxx/ deploy/messaging*. Wenn diese Eigenschaft `true` ist (der Standardwert), wird das Messaging-Verzeichnis nach bereitzustellenden Anwendungen durchsucht. Wird die Eigenschaft auf `false` eingestellt, wird das Verzeich Verzeichnis nicht durchsucht, was Probleme verursachen kann, wenn Sie den Messaging-Service benutzen möchten. Diese Einstellung wirkt sich nicht auf expandierte Anwendungsverzeichnisse wie *jmx-console.war* aus, weil diese immer Punkte im Namen haben.
`VFSBootstrap Scanner`	(wie für `VFSDeployment Scanner`)	Diese Orte werden im Rahmen des Bootstrap-Prozesses durchsucht. Wenn Sie nichts am Bootstrapping ändern möchten, sollten Sie auch diese Einstellung nicht ändern.

Bean	Eigenschaft	Beschreibung
`VFSDeployer Scanner`	(wie für `VFSDeployment Scanner`)	Orte, die nach den verschiedenen Deployern durchsucht werden.
`HDScanner`	`scanEnabled`	Stellen Sie dies auf `true` ein (den Standardwert), um den Hot Deployer zu aktivieren, und auf `false`, um ihn zu deaktivieren. Ist diese Eigenschaft `false`, werden Anwendungen nur bereitgestellt, wenn der Server gestartet oder die `deploy`-Methode auf der MBean `Main-Deployer` aufgerufen wird.
	`scanPeriod`	Gibt an, wie viele Millisekunden der Hot Deployer zwischen zwei Scans wartet. Der Standardwert beträgt 5000 Millisekunden (5 Sekunden). Dieser Wert wird ignoriert, wenn `scanEnabled` `false` ist.
	`scanThreadName`	Hiermit können Sie den Thread-Namen von seinem Standardwert `HDScanner` abändern. Mithilfe des Thread-Namens können Sie den Hot Deployer-Thread erkennen, wenn Sie einen Thread-Dump veranlassen müssen.

Wie in der Tabelle beschrieben, können Sie auch mehrere Bereitstellungsorte angeben. Das schauen wir uns am besten in einem Beispiel an.

Ein anderes Verzeichnis für die Bereitstellung

Angenommen, Sie ziehen es vor, Ihre Anwendungen in einem anderen Verzeichnis als *server/xxx/deploy* bereitzustellen. Vielleicht ist ja der Verzeichniszugriff so eingestellt, dass Sie nur Lesezugriff auf das Installationsverzeichnis des Anwendungsservers haben und in das *deploy*-Verzeichnis nicht schreiben dürfen. In diesem Fall können Sie die Konfiguration so abändern, dass auch Dateien unter (z.B.) */opt/deploy* bereitgestellt werden, indem Sie die Eigenschaft `URIList` der Bean `VFSDeploymentScanner` so wie in Listing 3.1 einstellen.

Listing 3.1 Bereitstellungsverzeichnisse in profile-service.xml

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
 ...
 <bean name="VFSDeploymentScanner" ...>
   ...
  <property name="URIList">
   <list elementClass="java.net.URI">
    <value>${jboss.server.home.url}deploy/</value>
    <value>file:/opt/deploy/</value>
   </list>
  </property>
 </bean>
</deployment>
```

Beachten Sie den angehängten Schrägstrich in */opt/deploy/*. Er bedeutet, dass der angegebene Ort ein Verzeichnis ist, das nach bereitzustellenden Anwendungen gescannt werden sollte. Fehlt der Schrägstrich, so gilt der angegebene Ort als eine einzelne bereitzustellende Anwendung – als Archivdatei oder expandiertes Verzeichnis. In dem Fall müsste der Speicherort eines der gültigen Suffixe haben, wie z.B. */opt/someapp.war*.

Mit dieser Änderung werden jetzt Anwendungen, die Sie in das Verzeichnis */opt/deploy* speichern, so behandelt, als lägen sie in *server/xxx/deploy*. Sie werden sogar automatisch neu bereitgestellt, wenn der Server neu gestartet wird.

Ein Aspekt, den Sie bei der Bereitstellung einer Anwendung bedenken müssen, ist die Frage: Wohin mit den JAR-Dateien, die diese Anwendung benötigt? Legen Sie sie in das Verzeichnis *server/xxx/lib* oder in den Klassenpfad? Bevor wir dies beantworten, müssen Sie zuerst verstehen, wie der Klassenlader funktioniert. Das ist unser nächstes Thema.

3.2 Wie Klassen geladen werden

Die JBoss AS-Dokumentation beschreibt genau, wie der Anwendungsserver Klassen lädt. Und auch mehrere Wiki-Seiten sagen Ihnen, wie Klassen geladen und Probleme, die dabei auftreten, gelöst werden. Anstatt all dies hier zu wiederholen (was in Programmiererkreisen Wiederverwendung, aber unter Autoren schlicht Plagiat heißt, sorry, Herr Lobatschewski), schlagen wir einen etwas anderen Weg ein.

Hinweis

Mit JBoss AS 5.0 wurde ein neuer Klassenlader eingeführt, der auf dem neuen Virtual File System (VFS) basiert. Das VFS wurde implementiert, um den Umgang mit Dateien im Anwendungsserver zu vereinfachen und zu vereinheitlichen. Der neue Klassenlader VFS Class Loader verwendet VFS, um JAR- und Klassendateien zu finden. Auch wenn dies eine deutliche Änderung für das Laden von Klassen in JBoss AS 5.0 bedeutet, ist das Verhalten noch fast dasselbe wie in den Vorläuferversionen von JBoss AS.

Die JBoss AS-Dokumentation verliert sich im Hinblick auf das Laden von Klassen so sehr in haarspalterischen Details, dass es Neulinge schon überwältigen kann. Wir beschreiben daher in vereinfachter Form, wie der Anwendungsserver Klassen lädt. Bitte beachten Sie, dass diese vereinfachte Sicht nicht alle Fälle abdeckt; wenn Sie also auf einen solchen Fall stoßen, schauen Sie in die JBoss AS-Dokumentation, um das Laden von Klassen noch besser zu verstehen. Hoffentlich finden Sie nach der Lektüre unserer vereinfachten Einführung den Detailreichtum der Dokumentation leichter verdaulich.

Wir beginnen mit einer Beschreibung der Klassenlader, denn der Anwendungsserver hat viele Klassenlader, um notfalls zwischen den Klassen richtig unterscheiden zu können. Danach betrachten wir das Scoping von Klassen, das den Anwendungsserver in die Lage versetzt, Klassen besser auseinanderhalten zu können. Zum Schluss betrachten wir Lader-Repositories, in denen mehrere Klasselader Klassen freigeben oder isolieren können.

3.2.1 Multiple Klassenlader

Der Anwendungsserver verwendet mehrere Klassenlader, von denen jeder eine bestimmte Menge von Klassen lädt. Zum Teil wird dies getan, um die bereitgestellten Anwendungen zu trennen. Wenn es nur einen einzigen Klassenlader gäbe und eine Anwendung eine Version und eine andere Anwendung eine andere Version einer Klasse benötigte, hätten Sie ein Problem. Eine der Anwendungen müsste dann die falsche Version benutzen und möglicherweise einen Fehler melden. Bei Verwendung mehrerer Klassenlader kann jede Anwendung ihre eigene Version der Klasse laden.

Der Anwendungsserver beobachtet alle Klassenlader und implementiert Regeln, die nicht nur definieren, welcher Klassenlader welche Klassen lädt, sondern auch, ob Klassen, die von einem Klassenlader geladen wurden, Zugriff oder Sichtbarkeit für Klassen bieten, die von einem anderen Klassenlader geladen wurden. Abbildung 3.3 zeigt vereinfacht eine solche Sichtbarkeit für Klassen, wobei jeder Kasten im Diagramm einen Klassenlader darstellt. Im Diagramm sind Klassen auf einer bestimmten Ebene sichtbar für andere Klassen auf derselben Ebene und für untergeordnete Klassen, aber nicht für Klassen auf einer höheren Ebene

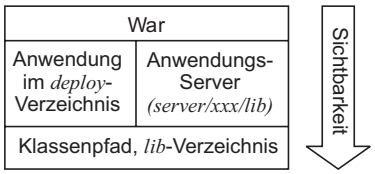

Abbildung 3.3
Eine vereinfachte Darstellung der Sichtbarkeit von Klassen: Hier können Klassen, die in übergeordneten Dateien oder Verzeichnissen liegen, Klassen in untergeordneten Verzeichnissen referenzieren.

Tipp

Keine Anwendungen sind sichtbar für Klassen in einer WAR-Datei, die aufgrund der Servlet-Spezifikation getrennt sein müssen. Dies können Sie ändern, indem Sie die Eigenschaft `useJBossWebLoader` der Bean `WarDeployer` in der Datei *sever/xxx/deployers/jboss-web.deployer/META-INF/war-deployers-jboss -beans.xml* auf `true` einstellen. Wenn Sie dann noch keine volle Sichtbarkeit haben, können Sie auch die Eigenschaft `java2ClassLoadingCompliance` derselben Bean ebenfalls auf `true` einstellen. Die Datei *war-deployers-beans.xml* beschreibt die Verwendung dieser beiden Eigenschaften.

Auf der untersten Ebene liegen die Klassen im Klassenpfad, einschließlich Java Virtual Machine(JVM)-Klassen (wie die in *rt.jar*) und die Bootklassen für den Anwendungsserver im *lib*-Verzeichnis. Auf der nächsten Ebene liegen alle Klassen für die bereitgestellten Anwendungen und die Anwendungsserverklassen aus den JAR-Dateien im Verzeichnis *server/xxx/lib*. Die oberste Ebene stellen Klassen in einer WAR-Datei dar, die im *deploy*-Verzeichnis oder in einer EAR- oder Service Archive(SAR)-Datei liegen könnte.

Diese Sichtbarkeit der Klassen ist praktisch, wenn Sie große Klassenbibliotheken mit mehreren Anwendungen gemeinsam nutzen möchten. Sie brauchen dann weniger Arbeitsspeicher, um diese Anwendungen ausführen zu können. Der Nachteil: Wenn zwei bereitgestellte Anwendungen verschiedene Versionen einer Klassenbibliothek benötigen, können

sie nicht auf diese Weise ausgeführt werden. Um diese beiden Anwendungen ordentlich bereitstellen zu können, müssen Sie etwas über das Scoping erfahren.

3.2.2 Scoping von Klassen

Sie können verlangen, dass eine bestimmte Anwendung ihr eigenes Klassenlader-Repository haben muss und ihre eigenen Klassen denen vorzieht, die in anderen Anwendungen zur Verfügung stehen. Dies bezeichnet man als Scoping. Sie definieren dazu ein Klassenlader-Repository in einer der JBoss AS-spezifischen Konfigurationsdateien. Schauen wir uns zuerst an einem Beispiel an, wie man Klassen-Scoping konfiguriert, um danach Klassenlader-Repositories zu erläutern.

Um für eine SAR-Datei Klassen mit Scoping zu laden, schreiben Sie Folgendes in die Datei *META-INF/jboss-service.xml*:

```
<service>
 <loader-repository>jbia.loader:loader=Loader1</loader-repository>
 ...
</service>
```

Für eine EAR-Datei schreiben Sie Folgendes in die Datei *META-INF/jboss-app.xml*:

```
<jboss-app>
 <loader-repository>jbia.loader:loader=Loader2</loader-repository>
 . . .
</jboss-app>
```

Und für eine WAR-Datei schreiben Sie Folgendes in die Datei *META-INF/jboss-web.xml*:

```
<jboss-web>
 <class-loading>
  <loader-repository>jbia.loader:loader=Loader3</loader-repository>
 </class-loading>
 . . .
</jboss-web>
```

Der einzige wichtige Namensteil ist das `loader`-Attribut; den Rest können Sie nennen, wie Sie möchten. Folgende Versionen sind z.B. alle gültig:

```
<loader-repository>foo.bar:loader=some.stuff</loader-repository>
<loader-repository>com.myorg:loader=org.ear</loader-repository>
```

Außerdem gilt: Wenn Ihre Anwendung verschiedene Versionen der JAR-Dateien aus dem Serververzeichnis *server/xxx/lib* verwendet, können Sie die Eigenschaft `java2ParentDelegation` für das Klassenlader-Repository auf `false` einstellen, damit der Anwendungsserver stattdessen die Klassen in Ihren JAR-Dateien benutzen muss. In der *META-INF/jboss-service.xml*-Datei eines SAR-Archivs würde dies wie folgt aussehen; für die anderen Dateien ist es ganz ähnlich.

```
<service>
 <loader-repository>jbia.loader:loader=Loader1
  <loader-repository-config>java2ParentDelegation=false
   </loader-repository-config>
 </loader-repository>
 . . .
</service>
```

Die Tags in den obigen Konfigurationsdateien beziehen sich auf Loader-Repositories und nicht auf Klassenlader. Und was sind nun Loader-Repositories? Diese Frage werden wir nun beantworten.

3.2.3 Loader-Repositories

Sie fragen sich vielleicht, was Klassenlader-Repositories mit Klassenladern zu tun haben und ob sie das Gleiche oder etwas Unterschiedliches sind. Sie sind unterschiedlich. Der Anwendungsserver erzeugt viele Klassenlader, für jede Anwendung eine. Wenn Sie die JMX Console ausführen, sehen Sie sie, wenn Sie die Agent-Ansichtsseite um ein Drittel nach unten gehen. Ihre Namen beginnen alle mit `jboss.classloader:id=`.

Darüber hinaus pflegt der Anwendungsserver mehrere Loader-Repositories. Dies sind die Orte, von denen der Anwendungsserver Klassen laden kann. Ein Loader-Repository kann von einem Klassenlader benutzt werden, während ein anderes von mehreren Klassenladern benutzt wird. Auch wenn für jede bereitgestellte Anwendung ein separater Klassenlader existiert, verwenden sie alle dasselbe Loader-Repository. Oder: Wenn Sie ein Loader-Repository für eine Anwendung definieren können, dann verwendet der Klassenlader für diese Anwendung sein eigenes Klassenlader-Repository.

Wenn Sie ein Loader-Repository für eine Anwendung definieren, bekommt der Klassenlader für diese Anwendung sein eigenes Repository. Das Endresultat ist, dass Klassen in dieser Anwendung Klassen, die ebenfalls in dieser Anwendung liegen, den Vorzug gegenüber Klassen außerhalb der Anwendung geben. Ein Nebeneffekt: Klassen in anderen Anwendungen sind für Klassen innerhalb dieser Anwendung nicht sichtbar. Das können Sie sich wie einen auf einer Seite durchsichtigen Spiegel vorstellen, der die Anwendung umgibt: Die anderen können hinaussehen, aber niemand kann hineinsehen.

Abbildung 3.4 illustriert Klassenpräferenzen mit und ohne ein Loader-Repository. Die JAR-Datei *three.jar* liegt im Verzeichnis *server/xxx/lib* und enthält Version 1 von Klasse C, während die EAR-Dateien *one.ear* und *two.ear* im Verzeichnis *server/xxx/deploy* liegen und die Klassen A und B enthalten. Zusätzlich enthält *two.ear* Version 2 von Klasse C. Um zu gewährleisten, dass die Klassen in *two.ear* die richtige Version von Klasse C aussuchen, wird ein Loader-Repository in *two.ear* deklariert. Wenn Klasse A Klasse C referenziert, bekommt sie die Version aus *three.jar*, aber wenn Klasse B Klasse C referenziert, bekommt sie die Version aus *two.ear*. Die EAR-Dateien sind nur ein Beispiel; wir hätten auch SAR, JAR oder jeden anderen Archivtyp nehmen können. Das Ergebnis wäre immer dasselbe.

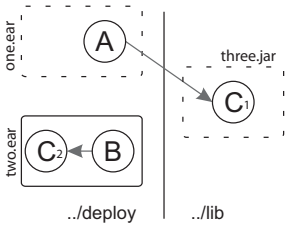

Abbildung 3.4
Ein Beispiel dafür, wie Klassenlader-Repositories den Zugriff auf Klassen beeinflussen. Die Klassen A und B referenzieren beide die erwartete Version der Klasse C.

Wie bereits zu Beginn unserer Erörterung gesagt, haben wir hier nur in vereinfachter Form dargestellt, wie der Anwendungsserver Klassen lädt. Wenn Sie Genaueres darüber wissen möchten, sollten Sie die JBoss AS-Dokumentation lesen. Aber immerhin sind Sie jetzt in der Lage, die häufigsten Probleme beim Laden von Klassen zu durchschauen.

3.3 Häufige Bereitstellungsfehler beheben

Die Anwendungsbereitstellung ist ein Gebiet, auf dem Murphy's Law regiert: Irgendetwas geht immer schief. Doch wenn das geschieht, können Sie sich damit trösten, dass Sie nicht der einzige Unglückliche sind; schon ein kurzer Blick auf die Online-Foren für JBoss-Benutzer lässt erkennen, dass viele andere ebenfalls von Murphy heimgesucht wurden. Wir haben einige der häufigsten Bereitstellungsprobleme aus den Foren herausgesucht und schlagen im Folgenden Lösungen für folgende Situationen vor:

- *Class not found*-Exceptions
- Fehler wegen doppelter JAR-Datei
- Fehler mit Zip-Dateien
- *Class cast*-Exceptions

3.3.1 Class not found-Exception

Ein `ClassNotFoundException`-Fehler liegt normalerweise an einer von zwei Ursachen. Die eine, an die man sofort denkt, ist das Fehlen einer JAR-Datei. Die typische Lösung besteht darin herauszufinden, welches JAR die Klasse enthält, und sie in Ihre WAR- oder EAR-Datei einzubinden. Sie können z.B. Ant mit folgendem Klassenpfad verwenden:

```
<path id="classpath">
 <fileset dir="/" includes="**/*.jar" />
</path>
```

Wer nicht gerne solche drastischen Maßnahmen ergreift, sollte die jarFinder-Utility benutzen. Mit ihr können Sie ein Verzeichnis von JAR-Dateien (einschließlich Unterverzeichnisse) nach einer Klasse, einer Eigenschaft oder einer anderen Art von Datei mit einem bestimmten Namen durchforsten.

Von *http://www.isocra.com/articles/jarFinder.php* können Sie den Quellcode des jarFinders herunterladen. Die Dateien erscheinen ebenfalls in der Quelle für dieses Buch. Nach dem Download extrahieren Sie die Utility und erstellen sie mit dem mitgelieferten Ant-Skript *build.xml*. Die resultierenden Klassen werden im Unterverzeichnis *classes* angezeigt, das Sie in den Klassenpfad einbinden müssen. Das folgende Beispiel zeigt, wie die Utility ausgeführt wird, um die Klasse *org.jboss.aop.advice.Interceptor* im Installationsverzeichnis von JBoss zu finden:

```
java –cp classes com.isocra.utils.jarSearch.DirectorySearcher
➥ $JBOSS_DIR org.jboss.aop.advice.Interceptor.class
```

Sobald Sie die JAR-Datei mit der Klasse gefunden haben, sollten Sie sie in Ihre Anwendung einbinden. Wenn Sie eine Clientanwendung mit diesem Problem haben, sollten Sie die JAR-Datei in den Klassenpfad des Clients legen.

Die eigentliche Ursache wird oft übersehen – nämlich dass der falsche Klassenlader nach der fehlenden Klasse sucht. So ruft z.B. ein Servlet in Ihrer WAR-Datei eine EJB auf, die in einer JAR-Datei in der EAR-Datei liegt, die ihrerseits versucht, auf eine Klasse in der WAR-Datei zuzugreifen. Das Problem dabei ist, dass die Klassen in der EAR-Datei für Klassen in der WAR-Datei nicht sichtbar sind.

Des Rätsels Lösung ist ziemlich einfach: Sie verlagern die JAR-Datei auf eine niedrigere Ebene in der Sichtbarkeitshierarchie der Klassen. In unserem Beispiel verlagern Sie die Klassen aus der WAR- in die EAR-Datei. Vielleicht müssen Sie dazu Ihre Klassen neu verpacken, weil Sie die Servlet-Klassen in der WAR-Datei behalten müssen. Jetzt haben die Servlets und die EJBs Zugriff auf die Klassen.

Eine Variante dieses Problems tritt bei einer Client-Anwendung auf, wenn der Text `no security manager` wie folgt auftritt:

```
javax.naming.CommunicationException [Root exception is
java.lang.ClassNotFoundException: org.jbia.SomeMissingClass
(no security manager: RMI class loader disabled)]
```

Eine Google-Suche würde erbringen, dass Sie einen Security Manager einrichten müssen, aber das ist in der Regel nicht notwendig. Stellen Sie stattdessen die notwendige JAR-Datei im Klassenpfad für die Clientanwendung zur Verfügung.

3.3.2 Doppelte JAR-Dateien

Ein anderes bekanntes Problem besteht darin, dass Sie in Ihre Anwendung eine JAR-Datei einbinden, die bereits im Verzeichnis *server/xxx/lib* des Anwendungsservers zur Verfügung steht (oder anderswo, wie etwa in der Tag-Bibliothek-JAR *jstl.jar*, die im Verzeichnis *server/xxx/deploy/jbossweb.sar* liegt). Manchmal ist das unproblematisch, aber wenn Sie eine `ClassCastException` bekommen, sollten Sie nachschauen, ob Sie JAR-Dateien liefern, die bereits vom Anwendungsserver zur Verfügung gestellt werden.

Eine Variante dieses Problems tritt ein, wenn Sie die Datei *log4j.jar* in Ihre Anwendung einbinden. Dann bekommen Sie folgenden Fehler:

```
10:42:50,093 ERROR [STDERR] log4j:ERROR "org.jboss.logging.util.Only
OnceErrorHandler" was loaded by [org.jboss.system.server.NoAnnotatio
nURLClassLoader@1de3f2d].
10:42:50,249 ERROR [STDERR] log4j:ERROR Could not create an Appender
. Reported error follows.
10:42:50,249 ERROR [STDERR] java.lang.ClassCastException: org.jboss.
logging.appender.DailyRollingFileAppender
```

In beiden Fällen ist die Lösung einfach: Entfernen Sie die störende JAR-Datei aus Ihrem Archiv, oder definieren Sie ein separates Klassenlader-Repository für die Anwendung.

3.3.3 Zip File-Fehler

Archivdateien liegen im Zip-Dateiformat vor, und die von der JVM mitgelieferten Zip File-Klassen sind dazu da, sie auszupacken. Treten dabei Probleme auf, können diverse Exceptions ausgelöst werden. Solche Probleme treten typischerweise auf, wenn ein Archiv in das *deploy*-Verzeichnis kopiert wird und gleichzeitig der Hot Deployer versucht, eine erst teilweise kopierte Datei bereitzustellen.

Angenommen, Sie haben eine 10 MB große WAR-Datei, die so groß ist, weil so viele JAR-Dateien eingebunden werden müssen. Nehmen wir weiter an, dass es 20 Sekunden dauert, die Datei über ein Netzwerk in das *deploy*-Verzeichnis zu kopieren. Erinnern Sie sich, dass der Hot Deployer alle fünf Sekunden läuft, also garantiert versuchen wird, die Datei bereitzustellen, bevor sie vollständig kopiert wurde. Das Ergebnis ist ein Zip File-Fehler.

Eine Variante davon tritt auf, wenn Sie ein expandiertes Verzeichnis bereitstellen. Wenn der Hot Deployer ausgeführt wird, sind noch nicht alle Dateien kopiert, und Sie bekommen eine Fehlermeldung wegen fehlender Dateien. Aber wenn Sie die Datei suchen, ist sie da. Den Hot Deployer deswegen zu verfluchen bringt nichts.

Die Lösung ist es, nichts in das *deploy*-Verzeichnis zu kopieren, sondern das Paket mit der Anwendung in ein temporäres Verzeichnis auf derselben Festplattenpartition zu schreiben, auf der auch das *deploy*-Verzeichnis liegt. Ist der Kopiervorgang abgeschlossen, verschieben Sie das ganze Paket in das *deploy*-Verzeichnis. Eine solche Verlagerung ist eine atomare Operation, weil sie auf derselben Partition stattfindet.

3.3.4 Class Cast-Exception

Class Cast-Exceptions werden von unterschiedlichen Problemen ausgelöst, zuallererst natürlich dadurch, dass das Objekt, das Sie zu konvertieren versuchen, nicht den erwarteten Typ hat. Das können Sie leicht feststellen, indem Sie die Klasse des zu konvertierenden Objekts untersuchen, entweder mit einem Debugging-Tool oder indem Sie Ihrer Anwendung Logging-Code hinzufügen, der ausgibt, zu welcher Klasse das Objekt gehört. Doch es gibt noch zwei weitere, subtilere Gründe, die ebenfalls auftreten können, wenn Sie einen Anwendungsserver benutzen.

Erstens: Vielleicht beschaffen Sie ein Objekt, z.B. eine EJB, von JNDI und versuchen dieses dann zu konvertieren. Betrachten Sie z.B. folgenden Code zum Zugriff auf eine EJB:

```
Context ctx = new InitialContext();
MyEjb ejb = (MyEjb)ctx.lookup("MyEjb");
```

Ein potenzielles Problem mit diesem Code ist, dass Sie den verkehrten JNDI-Namen verwendet haben, um die EJB nachzuschlagen. Nehmen wir an, die EJB hat standardmäßig den Namen `MyEjb/local`. In diesem Fall würde Ihnen die Lookup-Methode ein JNDI Context-Objekt zurückliefern – daher die Class Cast-Exception. Die Lösung ist, den vollen EJB-Namen zu benutzen:

```
MyEjb ejb = (MyEjb)ctx.lookup("MyEjb/local");
```

Ein zweites potenzielles Problem besteht darin, dass der Name der EJB vielleicht `MyEjb` lauten mag, aber nur für die Remote-Schnittstelle. In diesem Fall müssen Sie die Methode `narrow` auf der Klasse `RemoteObject` wie folgt aufrufen:

```
Object obj = ctx.lookup("MyEjb");
MyEjb ejb = (MyEjb)RemoteObject.narrow(obj, MyEjb.class);
```

Eine andere subtile Ursache für die Class Cast-Exception könnte daran liegen, dass das Objekt mit einem Klassenlader erzeugt wurde, aber in eine Klasse umgewandelt werden soll, die von einem anderen Klassenlader geladen wurde. Nehmen wir z.B. an, Sie haben eine Collection namens `coll` und extrahieren daraus mit folgendem Code `org.foo.Widget`-Objekte:

```
Object obj = coll.get(i);
log.debug("obj=" + obj.getClass());
org.foo.Widget w = (org.foo.Widget)obj;
```

Wenn Sie das ausführen, gibt das Programm folgende Zeile in der Logdatei aus:

```
...obj=org.foo.Widget
```

Doch die dritte Codezeile löst eine Class Cast-Exception aus.

Normalerweise geschieht das, wenn Sie in Ihrer Anwendung eine JAR-Datei mit verpacken, die bereits vom Anwendungsserver bereitgestellt wird. Das Objekt wurde wahrscheinlich aus der Klasse erzeugt, die im Anwendungsserver definiert ist, und die Konvertierung verwendet die Klassendefinition aus Ihrer Anwendung. Selbst wenn die beiden Klassen identisch sind, werden sie als unterschiedlich betrachtet, weil sie verschiedene Klassenlader haben. Die Lösung: Entweder verwenden Sie für Ihre Anwendung ein separates Klassenlader-Repository, oder Sie entfernen das Duplikat der Klassenbibliothek aus Ihrer Anwendung.

3.4 Diverse Anwendungen bereitstellen

Nun müssten Sie alles über die Bereitstellung von Anwendungen auf dem Anwendungsserver wissen. Sogar auf Murphys Heimsuchungen sind Sie vorbereitet. In den folgenden Kapiteln werden wir verschiedene Arten von Anwendungen behandeln, aber vorher kümmern wir uns um einige Anwendungstypen, die kein ganzes Kapitel beanspruchen. Diese Anwendungen sind Datenquellen und Hibernate-Archive.

3.4.1 Datenquellen bereitstellen

Irgendwann benötigen Sie einen Mechanismus, um auf Daten in einer Datenbank zuzugreifen. Ihre Anwendung könnte mit Java Database Connectivity (JDBC) direkt auf die Datenbank zugreifen, aber da wäre ein ziemlich großes Problem: Eine Datenbankverbindung einzurichten ist eine aufwendige Operation. Und mit einer Webanwendung stellen Sie dann fest, dass Sie die meiste Zeit mit Datenbankabfragen verbringen, und wünschen sich, nicht so oft eine Verbindung neu aufbauen zu müssen.

Die Lösung besteht darin, den Anwendungsserver Datenbankverbindungen mittels einer Datenquelle verwalten zu lassen. Dann kann der Anwendungsserver einen Verbindungspool aufbauen und der Anwendung bei Bedarf daraus eine Verbindung zuweisen. Die Frage ist nur: Wie können Sie eine Datenquelle auf dem Anwendungsserver deklarieren oder bereitstellen?

Die Antwort liegt in der Deklarierung und anschließenden Bereitstellung einer **-ds.xml*-Datei. Sie können jeden beliebigen Dateinamen verwenden, wenn nur das Suffix `-ds.xml` ist. Listing 3.2 zeigt ein Beispiel.

Listing 3.2 Beispiel einer *-ds.xml-Datei

```
<datasources>
 <local-tx-datasource>
  <jndi-name>MySqlDS</jndi-name>
  <connection-url>jdbc:mysql://mysql-hostname:3306/jbossdb
  ➥ </connection-url>
  <driver-class>com.mysql.jdbc.Driver</driver-class>
  <user-name>x</user-name>
  <password>y</password>
  <min-pool-size>5</min-pool-size>
  <max-pool-size>20</max-pool-size>
  <idle-timeout-minutes>0</idle-timeout-minutes>
  <blocking-timeout-millis>5000</blocking-timeout-millis>
  <exception-sorter-class-name>
  ➥ org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
  ➥ </exception-sorter-class-name>
  <check-valid-connection-sql>SELECT COUNT(*) FROM FooBar
  </check-valid-connection-sql>
  <metadata>
   <type-mapping>MySQL</type-mapping>
  </metadata>
  <connection-property name="xxx" type="java.lang.String">yyy
  ➥ </connection-property>
 </local-tx-datasource>
</datasources>
```

Das Tag `<local-tx-datasource>` definiert einen bestimmten Typ von Datenquelle, der die lokalen Transaktionen behandelt. Es stehen drei verschiedene Datenquellentypen zur Verfügung, und jeder behandelt Transaktionen anders. Tabelle 3.4 beschreibt diese Typen.

Tabelle 3.4 Transaktionstypen für Datenquellen

Tag	Beschreibung
`<local-tx-datasource>`	Eine Datenquelle, die Transaktionen verwendet, sogar verteilte Transaktionen im lokalen Anwendungsserver, verwendet aber keine über mehrere Anwendungsserver verteilten Transaktionen.
`<no-tx-datasource>`	Eine Datenquelle, die keine Transaktionen verwendet. Diese Option wird im Beispiel nicht gezeigt, aber wenn, dann würde sie anstelle des Tags `<local-tx-datasource>` auftreten.
`<xa-datasource>`	Eine Datenquelle, die über mehrere Anwendungsserver verteilte Transaktionen verwendet. Diese Option wird im Beispiel nicht gezeigt, aber wenn, dann würde sie anstelle des Tags `<local-tx-datasource>` auftreten.

Welchen Transaktionstyp sollten Sie verwenden? In den meisten Fällen `<local-tx-data-source>`, weil er Transaktionen in einem einzigen Anwendungsserver bearbeitet. Wenn Sie Ihre Anwendungsserver clustern oder auf mehrere Anwendungsserver verteilte Transaktionen wünschen, dann sollten Sie `<xa-datasource>` verwenden. Beachten Sie, dass `<local-tx-datasource>` und `<xa-datasource>` verteilte Transaktionen behandeln, zu denen mehrere Datenquellen gehören. Der Unterschied ist, dass `<local-tx-datasource>` diese Transaktionen in nur einem einzigen laufenden Anwendungsserver behandelt, während `<xa-datasource>` sie auf mehrere Anwendungsserver verteilt. Am anderen Ende des Spektrums eignet sich `<no-tx-data-source>` für Fälle, in denen Ihre Anwendungen die Datenbank nur lesen müssen.

Hinweis

XA ist eine API, die vom The Open Group's Distributed Transaction Processing-Modell definiert wird. Dieses Modell stellt Kommunikationsmechanismen zwischen einem Transaction Monitor und mehreren Ressourcenmanagern zur Verfügung, die Updates in die Datenbank schreiben. Der Transaction Monitor koordiniert die einzelnen, von den Ressourcenmanagern behandelten Transaktionen, um zu gewährleisten, dass die Transaktionssemantik intakt bleibt, wenn an einer einzelnen Transaktion mehrere Ressourcenmanager beteiligt sind.

Innerhalb des Transaktionstyps können Sie eine Vielzahl von Konfigurationsoptionen angeben. Tabelle 3.5 beschreibt die verschiedenen Konfigurationsoptionen in der `*-ds.xml`-Datei. Eine vollständige Sammlung von Konfigurationsoptionen mit Beschreibungen finden Sie in der Datei *docs/dtd/jboss-ds_5_0.dtd*.

Tabelle 3.5 Datenquellen-Konfigurationsoptionen für *-ds.xml

Tag	Beschreibung
`<jndi-name>`	Unter diesem Namen wird die Datenquelle im JNDI-Namespace nachgeschlagen. Das Präfix `java:` wird dem Namen automatisch hinzugefügt.
`<connection-url>`	Diesen URL verwendet der JDBC-Treiber zur Einrichtung einer Datenbankverbindung. Der URL ist spezifisch für die Datenbank und den zugehörigen Treiber. Im Beispiel heißt die Datenbank *jbossdb*; sie muss eine gültige Datenbank in MySQL sein.
`<driver-class>`	Der Klassenname für den JDBC-Treiber. Gilt nur für `<local-tx-datasource>` und `<no-tx-datasource>`.
`<xa-datasource-class>`	Der Klassenname für die Datenquelle für verteilte Transaktionen Gilt nur für `<xa-datasource>`.
`<user-name>`	Der Benutzername für die Einrichtung der Datenbankverbindung.
`<password>`	Das Passwort für den angegebenen Benutzernamen.
`<security-domain>`	Verweist auf eine Sicherheitsdomäne, die ein in login-config.xml definiertes Identitäts-Loginmodul verwendet. Dies können Sie anstelle von `<username>` und `<password>` verwenden, um der Datenbank ein verschlüsseltes Passwort zu übergeben.

Tag	Beschreibung
`<min-pool-size>`	Mindestzahl der offenen Verbindungen, die der Anwendungsserver behält. Der Anwendungsserver öffnet diese vorgegebene Anzahl Verbindungen bei der ersten Verbindungsanforderung, vorher nicht. Um Verbindungen schon beim Starten des Anwendungsservers aufzubauen, schreiben Sie einen einfachen Service, der nichts anderes tut, als eine Verbindung anzufordern.
`<max-pool-size>`	Höchstzahl der offenen Verbindungen, die der Anwendungsserver behält. Gehen dem Anwendungsserver die Verbindungen aus, weist er eine neue Verbindung zu, um der neuen Anforderung nachzukommen, aber nur bis diese Höchstzahl erreicht ist. Ab dann stellt er die Anforderungen in eine Warteschlange, bis wieder Verbindungen frei werden. Daher ist es wichtig, dass Anwendungen die erhaltenen Verbindungen wieder schließen.
`<idle-timeout-minutes>`	Wenn eine zusätzliche Verbindung für diese Anzahl Minuten unbenutzt bleibt, wird sie geschlossen. Beachten Sie, dass die Anzahl der offenen Verbindungen nie unter den Wert `<min-pool-size>` absinkt.
`<blocking-timeout-millis>`	Gibt an, wie viele Millisekunden ein Client auf eine Verbindung wartet, ehe der Timeout eintritt. Der Client erhält dann eine Exception.
`<exception-sorter-class-name>`	Eine Klasse, die verwendet wird, um festzustellen, ob ein Fehler, den die Datenbank zurückliefert, fatal ist.
`<check-valid-connection-sql>`	Gibt SQL-Code an, der beim Einrichten der Verbindung ausgeführt werden muss, um deren Gültigkeit zu testen. Im Beispiel- muss `FooBar` eine Tabelle in der Datenbank *jbossdb* sein, die im Verbindungs-URL angegeben ist.
`<valid-connection-checker- class-name>`	Kennzeichnet eine Klasse, die verwendet werden kann, wenn die Verbindung auf ihre Gültigkeit geprüft wird. Die angegebene Klasse muss die Schnittstelle `org.jboss.resource.adapter.jdbc. ValidConnectionChecker` implementieren. Verwenden Sie diese Option statt `<check-valid-connection-sql>`, wenn Sie die Gültigkeit der Verbindung mit mehr als nur einer SQL-Anweisung testen möchten. Achtung: Jede Anwendung, die eine Verbindungsanforderung absetzt, aufgrund derer eine Verbindung eingerichtet wird, muss warten, bis der Connection Checker fertig ist.
`<type-mapping>`	Wird vom Container-Managed Persistence(CMP)-Code verwendet, um die Datenbank zu ermitteln und seine Datenbankbehandlung darauf einzustellen. Der Name muss in der Datei *standard-jbosscmp-jdbc.xml* vorhanden sein. Außerdem können Sie neue Einträge zu *standardj-bosscmp-jdbc.xml* hinzufügen, um die Datenbankinteraktion anzupassen. Beachten Sie, dass das nur für EJB 2.1, aber nicht für EJB3 verwendet wird.

Tag	Beschreibung
`<connnection-property>`	Eine Eigenschaft, die an den `java.sql.Driver` beim Einrichten einer Datenbankverbindung übergeben werden muss. Im Beispiel heißt die Eigenschaft *xxx*, und der Wert lautet *yyy*. Schlagen Sie die gültigen Eigenschaften in Ihrer JDBC-Treiberdokumentation nach. Sie können auch mehrere `<connection-property>`-Einträge übergeben. Gilt nur für `<local-tx-datasource>` und `<no-tx-datasource>`.
`<xa-datasource-property>`	Kennzeichnet eine Eigenschaft, die an `javax.sql.DataSource` beim Einrichten einer Datenbankverbindung übergeben wird. Schlagen Sie die gültigen Eigenschaften in Ihrer JDBC-Treiberdokumentation nach. Sie können auch mehrere `<xa-datasource-property>`-Einträge übergeben. Gilt nur für `<xa-datasource>`.

`<transaction-isolation>`	Zeigt die Ebene der Transaktionsisolation für die Datenbank an. Zulässige Werte sind:
	`TRANSACTION_READ_UNCOMMITTED`
	`TRANSACTION_READ_COMMITTED`
	`TRANSACTION_REPEATABLE_READ`
	`TRANSACTION_SERIALIZABLE`
	`TRANSACTION_NONE`
	Die Beschreibung dieser Ebenen und welche Ebenen Ihre Datenbank unterstützt schlagen Sie in der JDBC-Dokumentation Ihrer Datenbank nach. Gilt nicht für `<no-tx-datasource>`.

Sie können in einer einzelnen **-ds.xml*-Datei auch mehrere Datenquellen definieren, aber wir empfehlen nur eine einzige Datenquelle pro Datei, da die Datenquellen dann einfacher zu verwalten sind.

JBoss AS wird mit einer Reihe von **-ds.xml*-Beispieldateien für verschiedene Datenbanken geliefert. Diese finden Sie im Verzeichnis *docs/examples/jca*, das Ihre erste Anlaufstelle sein sollte, wenn Sie eine **-ds.xml*-Datei definieren. Außerdem stellen wir in etlichen Kapiteln dieses Buches **-ds.xml*-Beispieldateien vor.

Da Sie nun die **-ds.xml*-Datei für Ihre Datenbank haben, tun Sie damit zweierlei: Zuerst gehört die **-ds.xml*-Datei in das *deploy*-Verzeichnis. Ja, Sie haben richtig gelesen: Die Datei wird wie eine Anwendung, genauer gesagt wie ein Service behandelt. Zweitens müssen Sie die JAR-Datei für den JDBC-Treiber der Datenbank zur Verfügung stellen. Die Treiber-JAR-Datei legen Sie in das Verzeichnis *server/xxx/lib*. Seien Sie vorsichtig mit JDBC-Treibern, die eine Versionsnummer im Dateinamen haben: Sie möchten ja nicht zwei Versionen desselben Treibers gleichzeitig im *server/xxx/lib*-Verzeichnis liegen haben.

Sobald die Datenquelle bereitgestellt ist, erstellt der Anwendungsserver mehrere MBeans für sie. Diese MBeans sind in Tabelle 3.6 beschrieben, wobei *XXX* der JNDI-Name für die Datenquelle ist.

Tabelle 3.6 MBeans, die für eine Datenquelle erzeugt werden

MBean	Beschreibung
`jboss.jca:name=XXX, service=DataSourceBinding`	Verwaltet die `javax.sql.DataSource`-Objekte.
`jboss.jca:name=XXX, service=LocalTxCM`	Verwaltet den Connection-Manager, der für den Verbindungspool zuständig ist. Mit dieser MBean verwalten Sie diverse Aspekte von verteilten Transaktionen, z.B. den Timeout-Wert für Transaktionen auf der lokalen XA-Ressource. Wird nur für `<local-tx-datasource>` erstellt.
`jboss.jca:name=XXX, service=XATxCM`	Verwaltet den Connection Manager, der für den Verbindungspool zuständig ist. Mit dieser MBean verwalten Sie diverse Aspekte von verteilten Transaktionen, z.B. den Timeout-Wert für Transaktionen auf der lokalen XA-Ressource. Wird nur für `<xa-datasource>` erstellt.
`jboss.jca:name=XXX, service=NoTxCM`	Verwaltet den Connection Manager, der für den Verbindungspool zuständig ist. Wird nur für `<no-tx-datasource>` erstellt.
`jboss.jca:name=XXX, service=ManagedConnectionFactory`	Verwaltet die Connection Factory, in der Datenbankverbindungen hergestellt werden.
`jboss.jca:name=XXX, service=ManagedConnectionPool`	Verwaltet den Pool der Datenbankverbindungen. Mit dieser MBean können Sie die Anzahl der aktiven Verbindungen überwachen und sogar die Höchst- und Mindestzahl der Verbindungen ändern.
`jboss.jdbc:service=metadata, datasource=XXX`	Mit dieser MBean können Sie die Typzuordnung ändern. Erscheint nicht, wenn die Datenquelle nicht für Transaktionen definiert ist.

Wenn Sie eine **-ds.xml*-Datei im *deploy*-Verzeichnis bereitstellen, steht diese Datenquelle allen auf dem Anwendungsserver bereitgestellten Anwendungen zur Verfügung. Sie können die **-ds.xml*-Datei auch mit Ihrer Anwendung verpacken. Als Beispiel schauen wir uns an, wie Sie eine Datenquelle in eine EAR-Datei verpacken können.

Eine Datenquelle in eine EAR-Datei verpacken

In Kapitel 7 werden wir beschreiben, wie Sie Webanwendungen und EJBs in eine EAR-Datei verpacken. Doch auch die **-ds.xml*-Datei lässt sich in die EAR-Datei integrieren.

Nehmen wir z.B. an, dass die Webanwendung in *inventory.war* verpackt ist und die EJBs in *inventory.jar*. Sie können diese Archive und den Deskriptor für die Datenquelle in eine einzige EAR-Datei verpacken, die den in Abbildung 3.5 gezeigten Inhalt hat.

Da die **-ds.xml*-Datei im Paket vorliegt, können Sie außerdem auch noch die JAR-Datei für den JDBC-Treiber dort unterbringen. Wenn Sie dem EAR-Dateideskriptor ein Klassenlader-Repository hinzufügen, hätte nur diese eine Anwendung Zugriff auf den JDBC-Treiber.

Abbildung 3.5
Der Inhalt des Pakets *inventory.ear* zeigt die eingebettete **-ds.xml*-Datei an.

Die Datei *application.xml* führt die Archive auf, die in die EAR-Datei verpackt sind, aber sie ist ein Standarddeskriptor von Java EE, und da Datenquelldeskriptoren nicht Teil der Standard-Java EE sind, können sie folglich in dieser Datei nicht referenziert werden. Stattdessen verwenden Sie eine *META-INF/jboss-app.xml*-Datei, die wie folgt auf den Datenquelldeskriptor verweist:

```
<!DOCTYPE jboss-app PUBLIC "-//JBoss//DTD J2EE Application 1.4//EN"
   "http://www.jboss.org/j2ee/dtd/jboss-app_4_0.dtd">
<jboss-app>
 <module>
  <service>inventory-ds.xml</service>
 </module>
</jboss-app>
```

Da Sie nun die EAR-Datei haben, können Sie sie bereitstellen und die Anwendung ausführen, die nunmehr den Datenquelldeskriptor benutzt, um auf die Datenbank zuzugreifen. Doch schauen wir uns nun an, wie ein Hibernate-Archiv bereitgestellt wird.

3.4.2 Ein Hibernate-Archiv bereitstellen

Auch wenn der Zugriff auf Daten in einer Datenbank eine integrale Anforderung für die meisten Anwendungen ist, möchten Sie diesen Zugriff vielleicht nicht immer nur auf der niedrigen Ebene von SQL-Anweisungen regeln, wie es JDBC verlangt. Hibernate ist eine Schicht für objektrelationale Zuordnung (ORM), die Sie in die Lage versetzt, in Ihrer Anwendung POJOs zu benutzen, während sich Hibernate darum kümmert, diese Objekte den Tabellen in einer relationalen Datenbank zuzuordnen. Dieses Buch ist nicht dazu da zu erklären, was Hibernate ist, wie es funktioniert und wie es genau benutzt wird; dies können Sie in *Java Persistence with Hibernate* von Christian Bauer und Gavin King nachlesen.

JBoss AS unterstützt Hibernate in Ihren Anwendungen durch einen simplen Mechanismus, nämlich das Hibernate-Archiv. Ein solches einfach zu erstellendes Archiv besteht aus den Klassen der Objekte, die Sie in der Datenbank speichern möchten, den Hibernate-Zuordnungsdateien für diese Klassen und einem Hibernate-Servicedeskriptor. Sie müssen noch nicht einmal ein Hibernate-Archiv anlegen, um Hibernate in JBoss AS zu nutzen; das Hibernate-Archiv wird lediglich zur Bequemlichkeit zur Verfügung gestellt, wenn Sie mehrere Anwendungen haben, die mit Hibernate auf dieselben Daten zugreifen.

Um ein Hibernate-Archiv anzulegen, haben Sie folgende Möglichkeiten:

- Ein persistentes Objekt programmieren, das die Daten definiert, die in der Datenbank gespeichert werden sollen

- Eine Zuordnungsdatei programmieren, die zeigt, wie der Inhalt der Objekte auf die Datenbank abgebildet wird
- Die Hibernate-Services-Datei programmieren, die das Hibernate-Archiv definiert

Wenn das Archiv fertig ist, können Sie es verpacken und bereitstellen. Diese Schritte schauen wir uns jetzt genauer an.

Das persistente Objekt programmieren

Zuerst benötigen Sie eine Klasse für persistente Objekte, z.B. eine einfache Klasse, die Informationen über ein Video enthält, das Sie vielleicht von einer Videothek-Website herunterladen, wie in Listing 3.3 gezeigt.

Listing 3.3 Video.java

```
package org.jbia.har;
public class Video {
 private int id;
 private String name;
 private int minutes;
 private float price;
 /*Getter und Setter*/
}
```

Da die Getter- und Setter-Standardmethoden sind, haben wir sie aus Platzgründen weggelassen. Beachten Sie, dass keine speziellen Konstrukte oder Anmerkungen erforderlich sind, alles sieht aus wie eine Standard-JavaBean. Das einzig Besondere ist das `id`-Feld, weil in Hibernate alle persistenzfähigen Klassen eine Objekt-`id` haben sollten.

Die Zuordnungsdatei programmieren

Als Nächstes müssen Sie definieren, wie der Inhalt der `Video`-Klasse auf eine Datenbanktabelle abgebildet werden soll. Hier kommt die Zuordnungsdatei ins Spiel. Listing 3.4 zeigt eine Zuordnungsdatei für die `Video`-Klasse.

Listing 3.4 Video.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
 <class name="org.jbia.har.Video" table="Video">
  <id name="id" type="integer" column="id">
   <generator class="identity" />
  </id>
  <property name="name" type="string" column="name" />
  <property name="minutes" type="integer" column="min" />
  <property name="price" type="float" column="price" />
 </class>
</hibernate-mapping>
```

Beachten Sie die Beschreibung des `id`-Felds. Wird die Generatorklasse auf `identity` eingestellt, so bedeutet dies, dass die Datenbank mit ihrer eingebauten Identity-Fähigkeit automatisch die Objekt-`id` zuweist, wenn das Objekt gespeichert wird. Das funktioniert

mit Datenbanken wie MySQL, aber für Datenbanken, die eine Sequence-Spalte unterstützen, verwenden Sie stattdessen den Klassenwert `sequence`. Beachten Sie das folgende Beispiel:

```
<id name="id" type="integer" column="id">
  <generator class="sequence" />
</id>
```

Andere mögliche Werte für die Generatorklasse können Sie in der Hibernate-Dokumentation nachschlagen.

Die Konfigurationsdatei für das Hibernate-Archiv programmieren

Zum Schluss benötigen Sie eine XML-Datei für die Hibernate-Archivkonfiguration, um das Hibernate-Archiv zu definieren. Listing 3.5 zeigt das XML für unser Videobeispiel.

Listing 3.5 video-hibernate.xml

```
<hibernate-configuration
  xmlns="urn:jboss:hibernate-deployer:1.0">
 <session-factory
  bean="jbia.har:app=Video"  ❶
  name="java:/hibernate/jbia/VideoSF">   ❷
  <property name="datasourceName">
➥  java:/jdbc/deploymentDS</property>   ❸
  <property name="dialect">
➥  org.hibernate.dialect.MySQLDialect   ❹
➥  </property>
 <property name="hbm2ddlAuto">   ❺
➥  create-drop</property>
   <property name="showSqlEnabled">
➥   true</property>
   <depends>jboss:service=Naming</depends>
   <depends>jboss:service=TransactionManager
➥   </depends>
   <depends>jboss.jca:name=jdbc/deploymentDS
➥ ,service=DataSourceBinding</depends>
 </session-factory>
</hibernate-configuration>
```

Dieser Deskriptor erstellt eine MBean, in diesem Fall mit Namen `jbia.har:app=Video` ❶. Sie können der MBean jeden gewünschten Namen geben, solange er sich nur an die Namenskonventionen für MBeans hält und unter allen bereitgestellten MBeans eindeutig ist.

Das `name`-Attribut kennzeichnet den JNDI-Namen für die Hibernate-Session-Factory ❷. Unter diesem Namen schlägt der Client die Session-Factory nach, um mit persistenten Objekten zu arbeiten.

Das Beispiel verweist auf die Datenquelle ❸, die im vorigen Abschnitt angelegt wurde. Daher ist ein Hibernate-Archiv kein Ersatz für eine **-ds.xml*-Datei, sondern baut darauf auf. Da die Datenquelle eine MySQL-Datenbank benutzt, geben wir das in der Eigenschaft `dialect` an ❹. Die Hibernate-Dokumentation listet die verfügbaren Dialekte auf; für fast jede relationale Datenbank ist einer vorhanden.

Die Einstellung `create-drop` für `Hbm2ddlAuto` ❺ zeigt an, dass Hibernate automatisch die in den Zuordnungsdateien beschriebenen Tabellen erzeugt und aus der Datenbank Tabellen löscht, die in keiner der Zuordnungsdateien zu finden sind. Dies ist ein Feature für eine Entwicklungsumgebung, aber nicht für eine Produktionsdatenbank. Der Hibernate-Archiv-Deployer sucht das Hibernate-Archiv automatisch nach Zuordnungsdateien **.hbm.xml* ab; Sie müssen diese nicht extra ausfindig machen. Die Hibernate-Dokumentation beschreibt auch die anderen Einstellungen für das Attribut `Hbm2ddlAuto`.

Die in diesem Beispiel benutzten Attribute sind nicht die einzigen Möglichkeiten. Zu vielen Hibernate-Eigenschaften gibt es entsprechende MBean-Attribute, die in einer Tabelle in der JBoss AS-Dokumentation beschrieben werden. Manche der Hibernate-Eigenschaften erfahren eine Sonderbehandlung, wie in Tabelle 3.6 angegeben.

Tabelle 3.6 Diese Hibernate-Eigenschaften werden besonders behandelt.

Hibernate-Eigenschaft	MBean-Attribut	Sonderbehandlung

hibernate.cache.
provider_class
	`CacheProviderClass`	Standardwert ist
		`org.hibernate.cache.`
		`HashtableCacheProvider`

hibernate.transaction. flush_before_completion	-keine-	Immer `true`
hibernate.transaction. auto_close_session	-keine-	Immer `true`
hibernate.connection aggressive_release	-keine-	Immer `true`
hibernate.connection release_mode	-keine-	Immer `after_statement`

Die Bedeutung dieser Einstellungen entnehmen Sie bitte der Hibernate-Dokumentation. Da Sie nun alle notwendigen Dateien beisammen haben, können Sie das Hibernate-Archiv erstellen.

Das Hibernate-Archiv erstellen

Kompilieren Sie die Java-Quelldatei(en), und legen Sie sie in eine Archivdatei namens *video.har*, wie in Abbildung 3.6 gezeigt. Beachten Sie, dass das Archiv die Erweiterung *.har* hat. Um ein solches Archiv zu erstellen, können Sie die JAR-Utility benutzen und die resultierende Datei *video.har* nennen.

Abbildung 3.6
Der Inhalt des Archivs video.har

Die Zuordnungsdatei *Video.hbm.xml* muss denselben Basisdateinamen (hier *Video*) wie die Klassendatei haben und überdies in demselben Verzeichnis wie diese liegen. Sie können auch Hilfsklassen einbinden, sofern erforderlich, aber bitte liefern Sie keine Zuordnungsdateien für eine Klasse, die nicht persistent ist.

Das Hibernate-Archiv bereitstellen

Um das Hibernate-Archiv bereitzustellen, kopieren Sie es in das *deploy*-Verzeichnis, sodass aus dem Hibernate-Archiv eine Anwendung wird (nach unserer lockeren Definition einer Anwendung). Und wie andere Anwendungen können Sie es auch als Archivdatei oder expandiertes Verzeichnis bereitstellen. Ist das erledigt, kann jede auf dem Anwendungsserver bereitgestellte Anwendung auf die persistenten Objekte zugreifen, die im Hibernate-Archiv definiert sind.

Den Hibernate-Client programmieren

Wie bereits gesagt, schlägt der Client eine Session-Factory anhand des JNDI-Namens nach. Von dort aus kann er eine Hibernate-Sitzung öffnen und benutzen, um persistente Objekte zu bearbeiten. Ist der Client fertig, sollte er seine Sitzung auch wieder schließen, um sie für andere Clients freizugeben.

Listing 3.6 ist ein Programm, das die Session-Factory nachschlägt und die Sitzung erstellt.

Listing 3.6 Hibernate-Sitzung nachschlagen

```
import javax.naming.InitialContext;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
...
 InitialContext ctx = new InitialContext();  ❶
 SessionFactory hsf = (SessionFactory)
  ctx.lookup("java:/hibernate/jbia/VideoSF");   ❷
 Session hs = hsf.openSession();   ❸
 try {
  /* tue etwas mit hs */
 } finally {
  hs.close();   ❹
}
...
```

Dieser Hibernate-Client ist eine auf dem Anwendungsserver bereitgestellte Anwendung; daher sind keine Eigenschaften erforderlich, um den anfänglichen JNDI-Kontext zu bekommen ❶. Der Client verwendet den in der Datei *hibernate-services.xml* angegebenen JNDI-Namen, um die Session-Factory nachzuschlagen ❷. Danach öffnet er eine Sitzung ❸, tut seine Arbeit und schließt die Sitzung wieder ❹.

Sobald Sie das Hibernate-Sitzungsobjekt haben, verwenden Sie es wie in jedem normalen Hibernate-Programm. Der folgende Code würde z.B. eine Liste aller Videos, alphabetisch nach Namen geordnet, abrufen:

```
import org.hibernate.Query;
import java.util.List;
...
```

```
 Query q = hs.createQuery("from org.jbia.har.Video order by name");
 List l = q.list();
...
```

Und der nun folgende Code würde ein neues Video anlegen und in der Datenbank speichern:

```
Video video = new Video();
video.setName("Monty Python and the Holy Grail");
video.setMinutes(91);
video.setPrice(14.99f);
hs.save(video);
```

Dieser Clientcode könnte in einer zustandslosen EJB-Session-Bean erscheinen, die von einem Servlet aufgerufen werden könnte. In Kapitel 7 beschreiben wir, wie Sie Webanwendungen und EJBs in eine EAR-Datei packen oder auch ein Hibernate-Archiv in eine EAR-Datei aufnehmen können.

Ein Hibernate-Archiv in einer EAR-Datei verpacken

Ein Hibernate-Archiv wird genauso wie eine Datenquelle in eine EAR-Datei verpackt. Nehmen wir z.B. an, dass die Webanwendung in *video.war* und die EJBs in *video.jar* verpackt sind. Sie können diese Archive und das Hibernate-Archiv in eine einzige EAR-Datei verpacken, die den in Abbildung 3.7 gezeigten Inhalt hat.

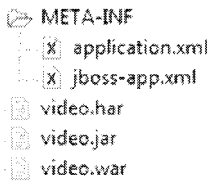

Abbildung 3.7
Inhalt des Pakets video.ear mit einem Hibernate-Archiv

Die Datei *application.xml* ist ein Standarddeskriptor von Java EE, und da Hibernate-Archive nicht zum Java EE-Standard gehören, können sie in dieser Datei nicht referenziert werden. Stattdessen referenzieren sie das Hibernate-Archiv wie folgt in einer *META-INF/jboss-app.xml*-Datei:

```
<!DOCTYPE jboss-app PUBLIC "-//JBoss//DTD J2EE Application 1.4//EN"
 "http://www.jboss.org/j2ee/dtd/jboss-app_4_0.dtd">
<jboss-app>
 <module>
 <har>video.har</har>
 </module>
</jboss-app>
```

Da Sie jetzt eine EAR-Datei haben, können Sie sie bereitstellen und die Anwendung ausführen, die dann das Hibernate-Archiv nutzt, um auf die persistenten Objekte zuzugreifen.

3.5 Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie Sie sowohl Geschäftsanwendungen als auch Services auf dem Anwendungsserver bereitstellen können. Eine Anwendung kann entweder als Archivdatei oder als expandiertes Verzeichnis verpackt werden. Sie haben die verschiedenen Anwendungstypen kennengelernt und festgestellt, dass diese in einer bestimmten Reihenfolge bereitgestellt werden.

Außerdem wissen Sie nun in groben Zügen, wie der Anwendungsserver Klassen lädt und wie Sie diesen Vorgang durch Scoping so begrenzen, dass keine Konflikte zwischen verschiedenen Versionen der Klassenbibliotheken auftreten können. Sie sind auf viele der häufigsten Bereitstellungsfehler vorbereitet, darunter Class Not Found-Exceptions, Class Cast-Exceptions und Fehler aufgrund einer doppelten JAR-Datei.

Als Zugabe haben Sie erfahren, wie Datenquellen und Hibernate-Archive konfiguriert werden. Auf die Konfiguration von Datenquellen werden wir in den restlichen Kapiteln dieses Buches noch mehrmals zurückkommen, weil fast alles Interessante, das mit einem Anwendungsserver getan werden kann, eine Datenbank erforderlich macht.

Die Informationen dieses Kapitels entsprechen in unserer Haus-Analogie dem Schritt, in dem Sie den Umzugslaster ausladen und die Möbel hineintragen. Vielleicht haben Sie auch schon eine Vorstellung davon, welche Dinge in welches Zimmer gehören.

Wir haben jedoch noch nichts behandelt, das dem Schritt entspräche, aus den einzelnen Zimmern funktionelle und einladende Bereiche eines Hauses zu machen. Dies werden wir im folgenden Teil des Buches tun: Wir besprechen bestimmte Anwendungstypen und wie man sie für den Anwendungsserver konfiguriert. Und zwar nehmen wir uns ein Zimmer – eine Anwendung – nach dem anderen vor. Doch bevor wir so weit sind, müssen wir noch ein weiteres, übergreifendes Thema behandeln, auf das wir in diversen Kapiteln zu sprechen kommen: die Sicherheit.

3.6 Quellen

- JBoss AS-Dokumentation – http://www.jboss.org/fileaccess/default/members/jbossas/freezone/docs/Server_Configuration_Guide/beta500/html/index.htmljar
- Finder-Utility – http://www.isocra.com/articles/jarFinder.php