PEARSON Education

ADDISON-WESLEY

Marc Fleury
Scott Stark
Norman Richards
JBoss, Inc.

JBoss 4.0

3 JBoss und Namensdienste

Dieses Kapitel erläutert den JNDI-basierten Namensdienst von JBoss, JBossNS, und die Rolle von JNDI in JBoss und J2EE. Außerdem bietet es eine Einführung in die grundlegende JNDI-API sowie allgemeine Nutzungskonventionen. Darüber hinaus werden Sie in diesem Kapitel etwas über die JBoss-spezifische Konfiguration der J2EE-Umgebungen für die Benennung von Komponenten erfahren, die von den Standard-bereitstellungsdeskriptoren konfiguriert werden. Das letzte Thema dieses Kapitels ist die Konfiguration und Architektur des JBoss- Namensdienstes.

In J2EE spielt der JBoss-Namensdienst eine Schlüsselrolle, da er einen Dienst bereitstellt, der dem Benutzer erlaubt, einem Objekt einen Namen zuzuordnen. Dies ist eine grundlegende Notwendigkeit in jeder Programmierumgebung, weil Entwickler und Administratoren in der Lage sein wollen, mit wieder erkennbaren Namen auf Objekte und Dienste zu verweisen. Ein gutes Beispiel für einen allgegenwärtigen Namensdienst ist das **D**omain **N**ame **S**ystem (DNS) im Internet. DNS gestattet Ihnen, auf Hosts zu verweisen, indem Sie logische Namen anstelle der numerischen Internetadressen verwenden. JNDI spielt eine ähnliche Rolle in J2EE, da es Entwicklern und Administratoren erlaubt, Name/Objekt-Bindungen für den Einsatz in J2EE-Komponenten zu erstellen.

3.1 Ein Überblick über JNDI

Bei JNDI handelt es sich um eine Standard-Java-API, die im Paket von JDK 1.3 und höher enthalten ist. JNDI stellt ein gemeinsames Interface für eine Vielzahl vorhandener Namensdienste bereit: DNS, LDAP, Active Directory, die RMI-Registrierung, die COS-Registrierung, NIS und Dateisysteme. Die JNDI-API ist logisch in eine Client-API für den Zugriff auf Namensdienste und ein Dienstanbieter-Interface (**S**ervice **P**rovider **I**nterface, SPI) gegliedert, das dem Benutzer gestattet, JNDI-Implementierungen für Namensdienste zu erstellen.

Die SPI-Schicht ist eine Abstraktion, die die Anbieter von Namensdiensten implementieren müssen, um den JNDI-Kernklassen zu ermöglichen, den Namensdienst mit Hilfe des gemeinsamen JNDI-Clientinterface offen zu legen. Eine Implementierung

von JNDI für einen Namensdienst wird als *JNDI-Provider* bezeichnet. Die Benennung in JBoss ist ein Beispiel für die JNDI-Implementierung auf der Grundlage von SPI-Klassen. Bitte beachten Sie, dass die Entwickler von J2EE-Komponenten das JNDI-SPI nicht benötigen.

Eine ausführliche Einführung in JNDI, worin sowohl das Client- als auch die Dienstanbieter-API behandelt werden, finden Sie in der Anleitung von Sun unter *http://java.sun.com/products/jndi/tutorial/*.

3.1.1 Die JNDI-API

Das Hauptpaket der JNDI-API ist `javax.naming`. Es enthält fünf Interfaces, zehn Klassen und mehrere Exceptions. Es gibt eine Schlüsselklasse, `InitialContext`, und zwei Schlüsselinterfaces, `Context` und `Name`.

Namen in JNDI

Die Aufgabe eines Namens ist von grundlegender Bedeutung in JNDI. Das Benennungssystem bestimmt die Syntax, der der Name entsprechen muss. Diese Syntax erlaubt dem Benutzer, Stringdarstellungen von Namen nach ihren Komponenten zu analysieren. Bei einem Namensdienst wird ein Name dazu eingesetzt, Objekte zu finden. Im einfachsten Sinne ist ein Namensdienst lediglich eine Sammlung von Objekten, die über eindeutige Namen verfügen. Um ein Objekt in einem Namensdienst zu finden, stellen Sie dem System einen Namen bereit, woraufhin das System das unter diesem Namen abgespeicherte Objekt zurückgibt.

Denken Sie zum Beispiel an die Namenskonvention des Unix-Dateisystems. Jede Datei wird durch ihren Pfad relativ zum Stamm des Dateisystems benannt, wobei die einzelnen Komponenten des Pfads durch einen Schrägstrich (`/`) getrennt sind. Der Dateipfad ist von links nach rechts angeordnet. Der Pfadname `/usr/jboss/readme.txt` benennt beispielsweise die Datei `readme.txt` im Unterverzeichnis `jboss` des Verzeichnisses `usr`, das sich im Stammverzeichnis des Dateisystems befindet. Die Benennung in JBoss verwendet einen Namensraum (engl. Namespace) im Unix-Stil und dessen Namenskonvention.

Das Interface `javax.naming.Name` stellt einen generischen Namen als geordnete Sequenz von Komponenten dar. Dabei kann es sich um einen gemischten Namen handeln, der mehrere Namensräume umfasst, oder um einen zusammengesetzten Namen, der innerhalb eines einzelnen hierarchischen Namensdienstes verwendet wird. Die Komponenten eines Namens sind nummeriert. Die Indizes eines Namens mit n Komponenten reichen von 0 bis n. Ausschließlich die signifikanteste Komponente befindet sich am Index 0. Ein leerer Name hat keine Komponenten.

Bei einem gemischten Namen (engl. Composite Name) handelt es sich um eine Folge von Komponentennamen, die mehrere Namensräume umfassen. Ein Beispiel für einen gemischten Namen ist die Kombination aus Host- und Dateiname, die allgemein bei Unix-Befehlen wie `scp` eingesetzt wird. Der folgende Befehl kopiert zum Beispiel `localfile.txt` in die Datei `remotefile.txt` im Verzeichnis `tmp` auf dem Host `ahost.someorg.org`:

```
scp localfile.txt ahost.someorg.org:/tmp/remotefile.txt
```

Ein zusammengesetzter Name (engl. Compound Name) wird von einem hierarchischen Namensraum abgeleitet. Jede Komponente eines zusammengesetzten Namens ist ein Ordnungsname (engl. Atomic Name) – d.h. es handelt sich um einen String, der nicht in kleinere Komponenten aufgelöst werden kann. Der Name des Dateipfads im Unix-Dateisystem ist ein Beispiel für einen zusammengesetzten Namen: `ahost.someorg.org:/tmp/remotefile.txt` ist ein gemischter Name, der sich über die Namensräume von DNS und des Unix-Dateisystems erstreckt. Die Komponenten des gemischten Namens sind `ahost.someorg.org` und `/tmp/remotefile.txt`. Bei einer *Komponente* handelt es sich um einen Stringnamen aus dem Namensraum des Namensdienstes. Stammt die Komponente aus einem hierarchischen Namensraum, kann sie mit Hilfe der Klasse `javax.naming.CompoundName` weiter in ihre elementaren Bestandteile aufgelöst werden. Die JNDI-API stellt die Klasse `javax.naming.Composite Name` als Implementierung des `Name`-Interface für gemischte Namen bereit.

Kontexte

`javax.naming.Context` ist das primäre Interface für die Interaktion mit einem Namensdienst. Es stellt eine Reihe von Name/Objekt-Bindungen dar. Jeder Kontext ist mit einer Namenskonvention verknüpft, die festlegt, wie er Stringnamen in `javax.naming.Name`-Instanzen auflöst. Um eine Name/Objekt-Bindung herzustellen, rufen Sie die Methode `bind()` eines Kontextes auf und geben einen Namen und ein Objekt als Argumente an. Später können Sie das Objekt anhand seines Namens über die Suchmethode von `Context` abfragen. Ein Kontext stellt normalerweise Operationen zur Bindung eines Namens an ein Objekt, zur Auflösung der Namensbindung und zum Abrufen eines Listings aller Name/Objekt-Bindungen bereit. Das Objekt, das Sie in einen Kontext einbinden, kann selbst den Typ `Context` aufweisen. Das gebundene `Context`-Objekt wird als *Unterkontext* des Kontextes bezeichnet, dessen `bind()`-Methode aufgerufen wurde.

Denken Sie beispielsweise an ein Dateiverzeichnis, das den Pfadnamen `/usr` hat und einen Kontext im Unix-Dateisystem darstellt. Ein relativ zu einem anderen Dateiverzeichnis benanntes Dateiverzeichnis ist ein Unterkontext (der im Allgemeinen als *Unterverzeichnis* bezeichnet wird). Ein Dateiverzeichnis mit dem Pfadnamen `/usr/jboss` benennt einen `jboss`-Kontext, bei dem es sich um einen Unterkontext von `usr`

handelt. Als weiteres Beispiel ist eine DNS-Domäne, z.B. `org`, zu nennen, bei der es sich um einen Kontext handelt. Eine relativ zu einer anderen DNS-Domäne benannte DNS-Domäne ist ein weiteres Beispiel für einen Unterkontext. In der DNS-Domäne `jboss.org` ist die DNS-Domäne `jboss` ein Unterkontext von `org`, da DNS-Namen von rechts nach links analysiert werden.

Einen Kontext mit Hilfe von InitialContext erhalten

Alle Operationen des Namensdienstes erfolgen über eine Implementierung des `Context`-Interface. Daher brauchen Sie eine Möglichkeit, einen Kontext für den Namensdienst zu erhalten, an dessen Verwendung Sie interessiert sind. Die Klasse `javax.naming.InitialContext` implementiert das `Context`-Interface und bildet den Ausgangspunkt für die Interaktion mit einem Namensdienst.

Beim Erstellen eines `InitialContext` wird dieser mit Umgebungseigenschaften initialisiert. JNDI bestimmt den Wert der einzelnen Eigenschaften, indem die Werte aus den folgenden beiden Quellen der Reihe nach miteinander verbunden werden:

- Das erste Auftreten der Eigenschaft aus dem Umgebungsparameter des Konstruktors und (bei geeigneten Eigenschaften) die Appletparameter und Systemeigenschaften.
- Alle im Klassenpfad gefundenen `jndi.properties`-Resssourcendateien.

Für jede in diesen beiden Quellen gefundene Eigenschaft wird der Eigenschaftswert wie folgt ermittelt. Wenn es sich bei der Eigenschaft um eine der standardmäßigen JNDI-Eigenschaften handelt, die eine Liste von JNDI-Factorys festlegen, werden alle Werte zu einer einzelnen durch Doppelpunkte getrennten Liste verkettet. Bei anderen Eigenschaften wird nur der erste Wert verwendet. Als bevorzugte Methode gilt das Festlegen der JNDI-Umgebungseigenschaften über eine `jndi.properties`-Datei, die es erlaubt, dass der Code die JNDI-Provider-spezifischen Informationen offen legt, so dass eine Änderung der JNDI-Provider keine Änderungen am Code und keine Neukompilierung nach sich zieht.

Die von der Klasse `InitialContext` intern verwendete `Context`-Implementierung wird zur Laufzeit bestimmt. Die Standardrichtlinie verwendet die Umgebungseigenschaft `java.naming.factory.initial`, die den Klassennamen der `javax.naming.spi.InitialContextFactory`-Implementierung enthält. Den Namen der Klasse `InitialContextFactory` erhalten Sie von dem von Ihnen verwendeten Namensdienst-Provider.

Listing 3.1 zeigte ein Beispiel für eine `jndi.properties`-Datei, die eine Clientanwendung für eine Verbindung zu einem JBossNS-Dienst über Port 1099 auf dem lokalen Host nutzt. Der Clientanwendung muss die Datei `jndi.properties` im Klassenpfad der Anwendung zur Verfügung stehen. Dies sind die Eigenschaften, die für die JBoss-

JNDI-Implementierung erforderlich sind. Andere JNDI-Provider verfügen über eigene Eigenschaften und Werte.

Listing 3.1: Eine `jndi.properties`-Beispieldatei

```
### JBossNS properties
java.naming.factory.initial=org.jnp.interfaces.
NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

3.1.2 J2EE und JNDI: die Umgebung der Anwendungskomponente

JNDI ist ein grundlegender Aspekt der J2EE-Spezifikationen. Eine wichtige Aufgabe von JNDI besteht darin, den Code von J2EE-Komponenten von der Umgebung zu isolieren, in der er bereitgestellt wird. Die Umgebung der Anwendungskomponente zu verwenden ermöglicht eine individuelle Anpassung der Komponente, ohne dabei auf ihren Quellcode zugreifen oder diesen ändern zu müssen. Die Umgebung der Anwendungskomponente wird als **E**nterprise **N**aming **C**ontext (ENC) bezeichnet. Es liegt in der Verantwortung des Containers der Anwendungskomponente, den Containerkomponenten einen ENC in Form des JNDI-`Context`-Interface zur Verfügung zu stellen. Die am Lebenszyklus einer J2EE-Komponente Beteiligten nutzen den ENC auf folgende Weise:

- Der Komponenten-Provider verwendet den Standardbereitstellungsdeskriptor für die Komponente, um die erforderlichen ENC-Einträge festzulegen. Bei den Einträgen handelt es sich um Deklarationen von Informationen und Ressourcen, die die Komponente zur Laufzeit benötigt. Die Geschäftslogik der Anwendungskomponente sollte so kodiert werden, dass sie auf die Informationen ihres ENC zugreift.
- Der Container stellt Werkzeuge zur Verfügung, um dem Deployer einer Komponente zu erlauben, die ENC-Referenzen zuzuordnen, die der Komponentenentwickler auf die entsprechende Entität der Bereitstellungsumgebung gesetzt hat.
- Der Komponenten-Deployer verwendet die Containerwerkzeuge, um eine Komponente zur endgültigen Bereitstellung vorzubereiten.
- Der Komponentencontainer verwendet die Informationen des Bereitstellungspakets, um den vollständigen Komponenten-ENC zur Laufzeit zu erstellen.

Die vollständige Spezifikation für die Verwendung von JNDI auf der J2EE-Plattform finden Sie im Abschnitt 5 der J2EE-1.4-Spezifikation, die unter *http://java.sun.com/j2ee/download.html* zur Verfügung steht.

Die Instanz einer Anwendungskomponente lokalisiert den ENC mit Hilfe der JNDI-API. Die Instanz einer Anwendungskomponente erstellt ein `javax.naming.InitialContext`-Objekt mit Hilfe des argumentlosen Konstruktors und sucht dann die Namensumgebung unter dem Namen `java:comp/env`. Die Umgebungseinträge der Anwendungskomponente werden direkt im ENC oder seinen Unterkontexten gespeichert. Listing 3.2 verdeutlicht die prototypischen Codezeilen, die von einer Komponente für den Zugriff auf ihren ENC verwendet werden

Listing 3.2: Beispielcode für den ENC-Zugriff

```
// Den ENC-Kontext der Anwendungskomponente abrufen
iniCtx = new InitialContext();
Context compEnv = (Context) iniCtx.lookup("java:comp/env");
```

Bei der *Umgebung einer Anwendungskomponente* handelt es sich um eine lokale Umgebung, die nur für die Komponente zugänglich ist, wenn der Steuerthread für den Anwendungsservercontainer mit der Anwendungskomponente interagiert. Das bedeutet, dass eine EJB `Bean1` nicht auf die ENC-Elemente der EJB `Bean2` zugreifen kann und umgekehrt. In ähnlicher Weise kann die Webanwendung `Web1` nicht auf die ENC-Elemente der Webanwendung `Web2` zugreifen – oder von `Bean1` oder `Bean2`, wenn wir schon einmal dabei sind. Außerdem kann ein beliebiger Clientcode unabhängig davon, ob er innerhalb der VM des Anwendungsservers oder extern ausgeführt wird, nicht auf den `java:comp`-JNDI-Kontext einer Komponente zugreifen. Aufgabe des ENCs ist die Bereitstellung eines isolierten, schreibgeschützten Namensraums, auf den sich die Anwendungskomponente stützen kann, und zwar unabhängig von der Art der Umgebung, in der die Komponente bereitgestellt wird. Der ENC muss von anderen Komponenten isoliert werden, da jede Komponente ihren eigenen ENC-Inhalt definiert. Die Komponenten A und B können beispielsweise für einen Verweis auf verschiedene Objekte denselben Namen festlegen. Die EJB `Bean1` kann zum Beispiel den Umgebungseintrag `java:comp/env/red` definieren, um auf den Hexadezimalwert für die RGB-Farbe Rot zu verweisen, während die Webanwendung `Web1` denselben Namen an die Darstellung des Sprachgebietsschemas der Bereitstellungsumgebung in Rot binden kann.

In JBoss gibt es allgemein verwendete Ebenen des Namensgültigkeitsbereichs: Namen unter `java:comp`, Namen unter `java:` und alle anderen Namen. Wie bereits erläutert, stehen der `java:comp`-Kontext und seine Unterkontexte nur der mit einem bestimmten Kontext verknüpften Anwendungskomponente zur Verfügung. Unterkontexte und Objektbindungen direkt unter `java:` sind nur innerhalb der VM des JBoss-Servers, nicht jedoch für Remoteclients sichtbar. Alle anderen Kontexte und Objektbindungen stehen Remoteclients unter der Voraussetzung zur Verfügung, dass der Kontext oder das Objekt die Serialisierung unterstützt. Im nächsten Abschnitt erfahren Sie, wie die Isolierung dieser Namensgültigkeitsbereiche erreicht wird.

Ein Beispiel dafür, wann die Einschränkung einer Bindung an den `java:`-Kontext nützlich ist, bildet eine `javax.sql.DataSource`-Verbindungs-Factory, die nur innerhalb des JBoss-Servers verwendet werden kann, wo sich der mit ihr verknüpfte Datenbankpool befindet. Andererseits würde ein EJB-Home-Interface an einen global sichtbaren Namen gebunden, der für Remoteclients zugänglich ist.

ENC-Konventionen

JNDI wird als API für die Offenlegung zahlreicher Informationen einer Anwendungskomponente genutzt. Der JNDI-Name, den die Anwendungskomponente für den Zugriff auf die Informationen verwendet, wird für EJB-Komponenten im Standardbereitstellungsdeskriptor `ejb-jar.xml` und für Webkomponenten im Standardbereitstellungsdeskriptor `web.xml` deklariert. Es können mehrere Arten von Informationen in JNDI gespeichert und von dort abgefragt werden; dazu gehören folgende:

- Umgebungseinträge, wie von den `env-entry`-Elementen deklariert
- EJB-Referenzen, wie von den Elementen `ejb-ref` und `ejb-local-ref` deklariert
- Referenzen auf Verbindungs-Factorys von Ressourcenmanagern, wie von den `resource-ref`-Elementen deklariert
- Referenzen auf Ressourcenumgebungen, wie von den `resource-env-ref`-Elementen deklariert

Alle Arten von Elementen der Bereitstellungsdeskriptoren verfügen über eine JNDI-Nutzungskonvention für den Namen des JNDI-Kontextes, unter dem die Informationen eingebunden sind. Zusätzlich zu dem Standardelement `deploymentdescriptor` gibt es einen für JBoss-Server spezifischen Bereitstellungsdeskriptor, der den von der Anwendungskomponente verwendeten JNDI-Namen dem JNDI-Namen der Bereitstellungsumgebung zuordnet.

Umgebungseinträge

Umgebungseinträge sind die einfachste Form der in einem Komponenten-ENC gespeicherten Informationen und weisen Ähnlichkeiten mit den Umgebungsvariablen von Betriebssystemen wie Unix oder Windows auf. Bei einem Umgebungseintrag handelt es sich um eine Name/Wert-Bindung, die einer Komponente gestattet, einen Wert offen zu legen und mit Hilfe eines Namens darauf zu verweisen.

Sie können einen Umgebungseintrag deklarieren, indem Sie ein `env-entry`-Element in den Standardbereitstellungsdeskriptoren verwenden. Das Element `env-entry` enthält die folgenden untergeordneten Elemente:

- Ein optionales `description`-Element mit einer Beschreibung des Eintrags
- Das Element `env-entry-name`, das den Namen des Eintrags relativ zu `java: comp/env` vorgibt

- Das Element `env-entry-type`, das den Java-Typ des Eingabewerts vorgibt. Dabei muss es sich um einen der folgenden handeln:
  - `java.lang.Byte`
  - `java.lang.Boolean`
  - `java.lang.Character`
  - `java.lang.Double`
  - `java.lang.Float`
  - `java.lang.Integer`
  - `java.lang.Long`
  - `java.lang.Short`
  - `java.lang.String`
- Das Element `env-entry-value`, das den Wert des Eintrags als String vorgibt

Listing 3.3 zeigt ein Beispiel für ein `env-entry`-Fragment aus einem `ejb-jar.xml`-Bereitstellungsdeskriptor. Es gibt kein JBoss-spezifisches Bereitstellungsdeskriptorelement, da es sich bei `env-entry` um eine vollständige Spezifikation des Namens und des Werts handelt Listing 3.4. zeigt ein Beispielcodefragment für den Zugriff und die im Bereitstellungsdeskriptor deklarierten Werte `maxExemptions` und `taxRate` `env-entry`.

Listing 3.3: Ein Beispiel für ein `env-entry`-Fragment in `ejb-jar.xml`

```
<!-- ... -->
<session>
    <ejb-name>ASessionBean</ejb-name>
    <!-- ... -->
    <env-entry>
        <description>
            The maximum number of tax exemptions allowed
        </description>
        <env-entry-name>maxExemptions</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>15</env-entry-value>
    </env-entry>
    <env-entry>
        <description>The tax rate </description>
        <env-entry-name>taxRate</env-entry-name>
        <env-entry-type>java.lang.Float</env-entry-type>
        <env-entry-value>0.23</env-entry-value>
    </env-entry>
</session>
<!-- ... -->
```

Listing 3.4: Ein Fragment eines `env-entry`-Zugangscodes

```
InitialContext iniCtx = new InitialContext();
Context envCtx = (Context) iniCtx.lookup("java:comp/env");
Integer maxExemptions = (Integer) envCtx.lookup
                        ("maxExemptions");
Float taxRate = (Float) envCtx.lookup("taxRate");
```

EJB-Verweise

Bei EJBs und Webkomponenten ist es üblich, mit anderen EJBs zu interagieren. Da es sich bei dem JNDI-Namen, unter dem ein EJB-Home-Interface eingebunden ist, um eine Entscheidung zur Bereitstellungszeit handelt, muss der Komponentenentwickler eine Möglichkeit haben, eine Referenz auf eine EJB zu deklarieren, die vom Deployer verknüpft wird. EJB-Referenzen erfüllen diese Anforderung.

Bei einer *EJB-Referenz* handelt es sich um eine Verknüpfung in der Namensumgebung einer Anwendungskomponente, die auf ein bereitgestelltes EJB-Home-Interface zeigt. Der von der Anwendungskomponente verwendete Name ist eine logische Verknüpfung, die die Komponente von dem eigentlichen Namen des EJB-Home-Interface in der Bereitstellungsumgebung isoliert. Die J2EE-Spezifikation empfiehlt, alle Referenzen auf Enterprise-Beans im Kontext `java:comp/env/ejb` der Umgebung für die Anwendungskomponente anzuordnen.

Eine EJB-Referenz wird mit Hilfe eines `ejb-ref`-Elements im Bereitstellungsdeskriptor deklariert. Jedes `ejb-ref`-Element beschreibt die Interfaceanforderungen, über die die referenzierende Anwendungskomponente für die referenzierte Enterprise-Bean verfügt. Das Element `ejb-ref` enthält die folgenden untergeordneten Elemente:

- Ein optionales Beschreibungselement, das den Zweck der Referenz angibt.
- Das Element `ejb-ref-name`, das den Namen der Referenz relativ zum Kontext `java:comp/env` festlegt. Um eine Referenz unter den empfohlenen Kontext `java:comp/env/ejb` zu stellen, verwenden Sie die Form `ejb/link-name` für den Wert `ejb-ref-name`.
- Das Element `ejb-ref-type`, das den Typ der EJB angibt. Dabei muss es sich entweder um `Entity` oder um `Session` handeln.
- Das Element `home`, das den vollständig qualifizierten Klassennamen des EJB-Home-Interface angibt.
- Das Element `remote`, das den vollständig qualifizierten Klassennamen des EJB-Remote-Interface angibt.
- Ein optionales `ejb-link`-Element, das die Referenz mit einer anderen Enterprise-Bean aus demselben EJB-JAR oder derselben J2EE-Anwendungseinheit verknüpft. Der Wert von `ejb-link` ist der `ejb-name` der referenzierten Bean. Sind noch andere

Enterprise-Beans mit demselben `ejb-name` vorhanden, verwendet der Wert einen Pfadnamen für den Speicherort der Datei `ejb-jar`, die die referenzierte Komponente enthält. Der Pfadname ist relativ zur referenzierenden `ejb-jar`-Datei angegeben. Der Anwendungsassembler hängt den `ejb-name` der referenzierten Bean durch das #-Zeichen getrennt an den Pfadnamen an. Dies erlaubt die eindeutige Kennzeichnung mehrerer Beans mit demselben Namen.

Der Gültigkeitsbereich einer EJB-Referenz wird auf die Anwendungskomponente gesetzt, deren Deklaration das Element `ejb-ref` enthält. Das bedeutet, dass die EJB-Referenz zur Laufzeit nicht von anderen Anwendungskomponenten aus zugänglich ist und dass andere Anwendungskomponenten `ejb-ref`-Elemente mit demselben `ejb-ref-name` definieren können, ohne einen Namenskonflikt hervorzurufen. Listing 3.5 stellt ein Fragment von `ejb-jar.xml` vor, das die Verwendung des Elements `ejb-ref` verdeutlicht. Listing 3.6 bietet ein Codebeispiel, das den Zugriff auf die in Listing 3.5 deklarierte `ShoppingCartHome`-Referenz zeigt.

Listing 3.5: Ein Beispiel für ein `ejb-ref`-Fragment eines `ejb-jar.xml`-Deskriptors

```
<description>This is a reference to the store products entity
</description>
        <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>org.jboss.store.ejb.ProductHome</home>
    </ejb-ref>
    <remote> org.jboss.store.ejb.Product</remote>
</session>
<session>
    <ejb-ref>
        <ejb-name>ShoppingCartUser</ejb-name>
        <!--...-->
        <ejb-ref-name>ejb/ShoppingCartHome</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>org.jboss.store.ejb.ShoppingCartHome</home>
        <remote> org.jboss.store.ejb.ShoppingCart</remote>
        <ejb-link>ShoppingCartBean</ejb-link>
    </ejb-ref>
</session>
<entity>
    <description>The Product entity bean </description>
    <ejb-name>ProductBean</ejb-name>
    <!--...-->
</entity>
<!--...-->
```

Listing 3.6: Ein Fragment eines `ejb-ref`-Zugangscodes

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ShoppingCartHome home = (ShoppingCartHome) ejbCtx.lookup
                        ("ShoppingCartHome");
```

EJB-Referenzen mit jboss.xml und jboss-web.xml

Der JBoss-spezifische EJB-Bereitstellungsdeskriptor `jboss.xml` beeinflusst EJB-Referenzen in zweierlei Hinsicht. Erstens gestattet das untergeordnete Element `jndi-name` der Elemente `session` und `entity` dem Benutzer, den JNDI-Namen der Bereitstellung für das EJB-Home-Interface festzulegen. Bei fehlender `jboss.xml`-Spezifikation für den `jndi-name` einer EJB wird das Home-Interface unter dem Wert von `ejb-name` aus `ejb-jar.xml` eingebunden. Bei der Session-EJB mit dem `ejb-name` von `ShoppingCartBean` in Listing 3.5 würde das Home-Interface bei fehlender `jndi-name`-Spezifikation in `jboss.xml` beispielsweise unter dem JNDI-Namen `Shopping CartBean` eingebunden.

Der zweite Verwendungszweck des Deskriptors `jboss.xml` im Hinblick auf `ejb-ref`s besteht darin, das Ziel festzulegen, auf das die ENC-`ejb-ref` einer Komponente verweist. Das Element `ejb-link` kann nicht für einen Verweis auf EJBs in anderen Unternehmensanwendungen genutzt werden. Muss `ejb-ref` auf eine externe EJB zugreifen, können Sie den JNDI-Namen des bereitgestellten EJB-Home-Interface durch den Einsatz des `jboss.xml`-Elements `ejb-ref/jndi-name` festlegen.

Der Deskriptor `jboss-web.xml` wird nur verwendet, um das Ziel festzulegen, auf das die `ejb-ref` für den ENC einer Webanwendung verweist. Das Inhaltsmodell für JBoss-`ejb-ref` umfasst folgende Elemente:

- Das Element `ejb-ref-name`, das dem Element `ejb-ref-name` im Standarddeskriptor `ejb-jar.xml` oder `web.xml` entspricht
- Das Element `jndi-name`, das den JNDI-Namen des EJB-Home-Interface in der Bereitstellungsumgebung festlegt

Listing 3.7 zeigt ein Beispiel für ein Fragment eines `jboss.xml`-Deskriptors, das die folgenden Verwendungszwecke verdeutlicht:

- Das `ejb-ref`-Verknüpfungsziel von `ProductBeanUser` wird auf den Bereitstellungsnamen `jboss/store/ProductHome` gesetzt.
- Der Bereitstellungs-JNDI-Name der `ProductBean` wird auf `jboss/store/Product-Home` gesetzt.

Listing 3.7: Ein Beispiel für ein `ejb-ref`-Fragment von `jboss.xml`

```
<!-- ... -->
<session>
    <ejb-name>ProductBeanUser</ejb-name>
    <ejb-ref>
        <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
        <jndi-name>jboss/store/ProductHome</jndi-name>
    </ejb-ref>
</session>

<entity>
    <ejb-name>ProductBean</ejb-name>
    <jndi-name>jboss/store/ProductHome</jndi-name>
    <!-- ... -->
</entity>
<!-- ... -->
```

Lokale EJB-Referenzen

Mit EJB 2.0 wurden lokale Interfaces hinzugefügt, bei denen keine RMI Call-by-Value-Aufrufe verwendet werden. Stattdessen verwenden diese Interfaces Call-by-Reference und benötigen deshalb keinen Overhead für die RMI-Serialisierung. Eine lokale EJB-Referenz ist eine Verknüpfung innerhalb einer Namensumgebung der Anwendungskomponete, die auf ein bereitgestelltes lokales EJB-Home-Interface zeigt. Der von der Anwendungskomponente verwendete Name ist eine logische Verknüpfung, die die Komponente vom eigentlichen Namen des lokalen EJB-Home-Interface in der Bereitstellungsumgebung isoliert. Die J2EE-Spezifikation empfiehlt, alle Referenzen auf Enterprise-Beans im Kontext `java:comp/env/ejb` der Umgebung der Anwendungskomponente anzuordnen.

Eine lokale EJB-Referenz deklarieren Sie, indem Sie das Element `ejb-local-ref` im Bereitstellungsdeskriptor verwenden. Jedes `ejb-local-ref`-Element beschreibt die Anforderungen, die die Anwendungskomponente für die referenzierte Enterprise-Bean an das Interface stellt. Das Element `ejb-local-ref` enthält die folgenden untergeordneten Elemente:

- Ein optionales `description`-Beschreibungselement, das den Zweck der Referenz angibt.
- Das Element `ejb-ref-name`, das den Namen der Referenz relativ zum Kontext `java:comp/env` festlegt. Um die Referenz unter dem empfohlenen Kontext `java:comp/env/ejb` einzuordnen, verwenden Sie die Form `ejb/`*`link-name`* für den Wert von `ejb-ref-name`.
- Das Element `ejb-ref-type`, das den Typ der EJB festlegt. Dabei muss es sich entweder um `Entity` oder um `Session` handeln.

- Das Element `local-home`, das den vollständig qualifizierten Klassennamen des lokalen EJB-Home-Interface angibt.
- Das Element `local`, das den vollständig qualifizierten Klassennamen des lokalen EJB-Interface angibt.
- Das Element `ejb-link`, das die Referenz mit einer anderen Enterprise-Bean in der Datei `ejbjar` oder in derselben J2EE-Anwendungseinheit verknüpft. Der Wert von `ejb-link` ist der `ejb-name` der referenzierten Bean. Sind mehrere Enterprise-Beans mit demselben `ejb-name` vorhanden, verwendet der Wert den Pfadnamen zum Speicherort der Datei `ejb-jar`, die die referenzierte Komponente enthält. Der Pfadname ist relativ zu der referenzierenden `ejb-jar`-Datei angegeben. Der Anwendungsassembler hängt den `ejb-name` der referenzierten Bean, durch das Zeichen # getrennt, an den Pfadnamen an. Damit können mehrere Beans mit demselben Namen eindeutig bezeichnet werden. Das Element `ejb-link` muss in JBoss angegeben werden, um eine Übereinstimmung mit der lokalen Referenz auf die entsprechende EJB zu erzielen.

Der Gültigkeitsbereich einer lokalen EJB-Referenz wird auf die Anwendungskomponente gesetzt, deren Deklaration das Element `ejb-local-ref` enthält. Das bedeutet, dass die lokale EJB-Referenz für andere Anwendungskomponenten zur Laufzeit nicht zugänglich ist und dass andere Anwendungskomponenten `ejb-local-ref`-Elemente mit demselben `ejb-ref-name` definieren dürfen, ohne einen Namenskonflikt zu verursachen. Listing 3.8 stellt ein Fragment von `ejb-jar.xml` vor, das die Verwendung des Elements `ejblocal-ref` verdeutlicht. Listing 3.9 zeigt Beispielcode, der den Zugriff auf die in Listing 3.8 deklarierte `ProbeLocalHome`-Referenz veranschaulicht.

Listing 3.8: Ein Beispiel für ein `ejb-local-ref`-Fragment des Deskriptors `ejb-jar.xml`

```
<!-- ... -->
<session>
    <ejb-name>Probe</ejb-name>
    <home>org.jboss.test.perf.interfaces.ProbeHome</home>
    <remote>org.jboss.test.perf.interfaces.Probe</remote>
    <local-home>org.jboss.test.perf.interfaces.ProbeLocalHome
    </local-home>
    <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
    <ejb-class>org.jboss.test.perf.ejb.ProbeBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Bean</transaction-type>
</session>
<session>
    <ejb-name>PerfTestSession</ejb-name>
    <home>org.jboss.test.perf.interfaces.PerfTestSessionHome
    </home>
    <remote>org.jboss.test.perf.interfaces.PerfTestSession
    </remote>
```

Listing 3.8: Ein Beispiel für ein `ejb-local-ref`-Fragment des Deskriptors `ejb-jar.xml` (Fortsetzung)

```
        <ejb-class>org.jboss.test.perf.ejb.PerfTestSessionBean
        </ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        <ejb-ref>
            <ejb-ref-name>ejb/ProbeHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>org.jboss.test.perf.interfaces.SessionHome</home>
            <remote>org.jboss.test.perf.interfaces.Session</remote>
            <ejb-link>Probe</ejb-link>
        </ejb-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ProbeLocalHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <local-home>org.jboss.test.perf.interfaces.
            ProbeLocalHome</local-home>
            <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
            <ejb-link>Probe</ejb-link>
        </ejb-local-ref>
    </session>
    <!-- ... -->
```

Listing 3.9: Ein Fragment des `ejb-local-ref`-Zugriffscodes

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ProbeLocalHome home = (ProbeLocalHome) ejbCtx.lookup
                      ("ProbeLocalHome");
```

Referenzen auf Verbindungs-Factorys von Ressourcenmanagern

Der Code der Anwendungskomponente kann durch die Verwendung logischer Namen, die man als *Referenzen auf Verbindungs-Factorys von Ressourcenmanagern* bezeichnet, auf Ressourcen-Factorys verweisen. Diese Referenzen werden von den `resource-ref`-Elementen in den Standardbereitstellungsdeskriptoren definiert. Der Deployer bindet die Referenzen auf die Verbindungs-Factorys von Ressourcenmanagern mit Hilfe der Deskriptoren `jboss.xml` und `jboss-web.xml` an die eigentlichen in der Zielbetriebsumgebung vorhandenen Verbindungs-Factorys der Ressourcenmanager.

Jedes `resource-ref`-Element beschreibt eine einzelne Referenz auf die Verbindungs-Factory eines Ressourcenmanagers. Das Element `resource-ref` besteht aus den folgenden untergeordneten Elementen:

- Einem optionalen `description`-Beschreibungselement, das den Zweck der Referenz angibt.
- Dem Element `res-ref-name`, das den Namen der Referenz relativ zum Kontext `java:comp/env` angibt. (Die auf dem Ressourcentyp beruhende Namenskonven-

tion, die besagt, für welchen Unterkontext `res-ref-name` eingefügt werden soll, wird gleich noch erläutert.)

- Dem Element `res-type`, das den vollständig qualifizierten Klassennamen der Verbindungs-Factory des Ressourcenmanagers angibt.
- Dem Element `res-auth`, das angibt, ob der Code der Anwendungskomponente die Ressourcenanmeldung programmatisch vornimmt oder ob der Container die Ressource anhand der vom Deployer gelieferten Hauptzuordnungsinformationen anmeldet. Dabei muss es sich entweder um `Application` oder um `Container` handeln.
- Dem optionalen Element `res-sharing-scope`, das derzeit nicht von JBoss unterstützt wird.

Die J2EE-Spezifikation empfiehlt, alle Referenzen auf Verbindungs-Factorys von Ressourcenmanagern in den Unterkontexten der Umgebung der Anwendungskomponente anzuordnen, wobei für jeden Typ von Ressourcenmanager ein anderer Unterkontext verwendet werden soll. Die empfohlene Zuordnung des Typs zum Namen des Unterkontextes sieht wie folgt aus:

- Referenzen auf JDBC-`DataSource` sollten im Unterkontext `java:comp/env/jdbc` deklariert werden.
- JMS-Verbindungs-Factorys sollten im Unterkontext `java:comp/env/jms` deklariert werden.
- JavaMail-Verbindungs-Factorys sollten im Unterkontext `java:comp/env/mail` deklariert werden.
- URL-Verbindungs-Factorys sollten im Unterkontext `java:comp/env/url` deklariert werden.

Listing 3.10 zeigt ein Beispiel für ein Fragment eines `web.xml`-Deskriptors, das die Verwendung des Elements `resource-ref` verdeutlicht Listing 3.11. stellt ein Codefragment vor, das eine Anwendungskomponente verwenden kann, um auf die von `resource-ref` deklarierte Ressource `DefaultMail` zuzugreifen.

Listing 3.10: Ein `resource-ref`-Fragment des `web.xml`–Deskriptors

```
<web>
    <!-- ... -->
    <servlet>
        <servlet-name>AServlet</servlet-name>
        <!-- ... -->
    </servlet>
    <!-- ... -->
    <!-- JDBC DataSources (java:comp/env/jdbc) -->
```

Listing 3.10: Ein `resource-ref`-Fragment des `web.xml`–Deskriptors (Fortsetzung)

```
    <resource-ref>
        <description>The default DS</description>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
    <!-- JavaMail-Verbindungs-Factorys (java:comp/env/mail) -->
    <resource-ref>
        <description>Default Mail</description>
        <res-ref-name>mail/DefaultMail</res-ref-name>
        <res-type>javax.mail.Session</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
    <!-- JMS-Verbindungs-Factorys (java:comp/env/jms) -->
    <resource-ref>
        <description>Default QueueFactory</description>
        <res-ref-name>jms/QueueFactory</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web>
```

Listing 3.11: Ein Beispiel für ein `resource-ref`-Zugangscodefragment

```
Context initCtx = new InitialContext();
javax.mail.Session s = (javax.mail.Session)
    initCtx.lookup("java:comp/env/mail/DefaultMail");
```

Referenzen auf Verbindungs-Factorys von Ressourcenmanagern mit jboss.xml und jboss-web.xml

Der Zweck des EJB-Bereitstellungsdeskriptors `jboss.xml` von JBoss und des Bereitstellungsdeskriptors `jboss-web.xml` für Webanwendungen besteht darin, die Verknüpfung des vom Element `res-ref-name` definierten logischen Namens mit dem in JBoss bereitgestellten JNDI-Namen der Ressourcen-Factory anzulegen. Dies erfolgt durch die Bereitstellung des Elements `resource-ref` im Deskriptor `jboss.xml` oder `jboss-web.xml`. Das JBoss-Element `resource-ref` besteht aus den folgenden untergeordneten Elementen:

- Dem Element `res-ref-name`, das mit dem `res-ref-name` eines entsprechenden `resource-ref`-Elements aus dem Standarddeskriptor `ejb-jar.xml` oder `web.xml` übereinstimmen muss.
- Dem optionalen Element `res-type`, das den vollständig qualifizierten Klassennamen der Verbindungs-Factory des Ressourcenmanagers angibt.

- Dem Element `jndi-name`, das den JNDI-Namen der Ressourcen-Factory angibt, wie in JBoss bereitgestellt.
- Dem Element `res-url`, das im Falle von `java.net.URL` vom Typ `resource-ref` den URL-String angibt.

Listing 3.12 führt ein Fragment des Deskriptors `jboss-web.xml` vor, das Zuordnungsbeispiele für die in Listing 3.10 vorgegebenen `resource-ref`-Elemente zeigt.

Listing 3.12: Ein `resource-ref`-Beispielfragment für den Deskriptor `jboss-web.xml`

```
<jboss-web>
    <!-- ... -->
    <resource-ref>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <jndi-name>java:/DefaultDS</jndi-name>
    </resource-ref>
    <resource-ref>
        <res-ref-name>mail/DefaultMail</res-ref-name>
        <res-type>javax.mail.Session</res-type>
        <jndi-name>java:/Mail</jndi-name>
    </resource-ref>
    <resource-ref>
        <res-ref-name>jms/QueueFactory</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <jndi-name>QueueConnectionFactory</jndi-name>
    </resource-ref>
    <!-- ... -->
</jboss-web>
```

Referenzen auf Ressourcenumgebungen

Eine Referenz auf Ressourcenumgebungen ist ein Element, das unter Verwendung eines logischen Namens auf ein mit einer Ressource verknüpftes, verwaltetes Objekt verweist (z.B. JMS-Ziele). Referenzen auf Ressourcenumgebungen werden von den `resource-env-ref`–Elementen in den Standardbereitstellungsdeskriptoren definiert. Der Deployer bindet die Referenzen mit Hilfe der Deskriptoren `jboss.xml` und `jboss-web.xml` an den tatsächlichen Speicherort des verwalteten Objekts in der Zielbetriebsumgebung.

Jedes `resource-env-ref`-Element beschreibt die Anforderungen, die die referenzierende Anwendungskomponente an das referenzierte verwaltete Objekt stellt. Das Element `resource-env-ref` besteht aus den folgenden untergeordneten Elementen:

- Einem optionalen `description`-Beschreibungselement, das den Zweck der Referenz angibt.

- Dem Element `resource-env-ref-name`, das den Namen der Referenz relativ zum Kontext `java:comp/env` angibt. Konventionsgemäß wird der Name in einen Unterkontext eingefügt, der mit dem verknüpften Ressourcen-Factory-Typ übereinstimmt. Der `resource-env-ref-name` einer Referenz auf eine JMS-Warteschlange mit dem Namen `MyQueue` sollte beispielsweise `jms/MyQueue` lauten.
- Dem Element `resource-env-ref-type`, das den vollständig qualifizierten Klassennamen des referenzierten Objekts angibt. Im Fall der JMS-Warteschlange lautet der Wert zum Beispiel `javax.jms.Queue`.

Listing 3.13 zeigt ein Beispiel für die Deklaration des Elements `resource-ref-env` anhand einer Session-Bean. Listing 3.14 enthält ein Codefragment, das verdeutlicht, wie die von `resource-env.ref` deklarierte `StockInfo`-Warteschlange gesucht wird.

Listing 3.13: Ein Beispiel für ein `resource-env-ref`-Fragment von `ejb-jar.xml`

```
<session>
    <ejb-name>MyBean</ejb-name>

    <resource-env-ref>
        <description>This is a reference to a JMS queue used in
            the processing of Stock info
        </description>
        <resource-env-ref-name>jms/StockInfo
        </resource-env-ref-name>
        <resource-env-ref-type>javax.jms.Queue
        </resource-env-ref-type>
    </resource-env-ref>
    <!-- ... -->
</session>
```

Listing 3.14: Ein Fragment eines `resource-env-ref`-Zugriffscodes

```
InitialContext iniCtx = new InitialContext();
javax.jms.Queue q = (javax.jms.Queue)
envCtx.lookup("java:comp/env/jms/StockInfo");
```

Referenzen auf Ressourcenumgebungen mit jboss.xml und jboss-web.xml

Der Zweck des Bereitstellungsdeskriptors `jboss.xml` für die EJB und des Bereitstellungsdeskriptors `jboss-web.xml` für Webanwendungen besteht darin, die Verknüpfung des vom Element `resource-env-ref-name` definierten logischen Namens mit dem JNDI-Namen des in JBoss bereitgestellten verwalteten Objekts vorzunehmen. Dies

geschieht durch die Bereitstellung des Elements `resource-env-ref` im Deskriptor `jboss.xml` oder `jboss-web.xml`. Das JBoss-Element `resource-env-ref` besteht aus den folgenden untergeordneten Elementen:

- Dem Element `resource-env-ref-name`, das mit dem `resource-env-ref-name` eines entsprechenden `resource-env-ref`-Elements aus dem Standarddeskriptor `ejb-jar.xml` oder `web.xml` übereinstimmen muss.
- Dem Element `jndi-name`, das den JNDI-Namen der Ressource angibt, wie in JBoss bereitgestellt.

Listing 3.15 enthält ein Fragment eines `jboss.xml`-Deskriptors, das eine `resource-env-ref`-Beispielzuordnung für `StockInfo` zeigt.

Listing 3.15: Ein `resource-env-ref`-Fragmentbeispiel für den Deskriptor `jboss.xml`

```
<session>
    <ejb-name>MyBean</ejb-name>
        <resource-env-ref>
        <resource-env-ref-name>jms/StockInfo
        </resource-env-ref-name>
        <jndi-name>queue/StockInfoQueue</jndi-name>
    </resource-env-ref>
    <!-- ... -->
</session>
```

3.2 Die JBossNS-Architektur

Die JBossNS-Architektur ist eine Java-Socket-/RMI-basierte Implementierung des `javax.naming.Context`-Interface. Es handelt sich um eine Client/Server-Implementierung, auf die ein Remotezugriff erfolgen kann. Die Implementierung ist so optimiert, dass ein Zugriff aus der VM heraus, in der auch der JBossNS-Server betrieben wird, die Sockets nicht einbezieht. Der Zugriff über dieselbe VM erfolgt durch eine Objektreferenz, die als globales Singleton zur Verfügung steht. Abbildung 3.1 zeigt einige der wichtigsten Klassen der JBossNS-Implementierung und ihre Beziehungen zueinander.

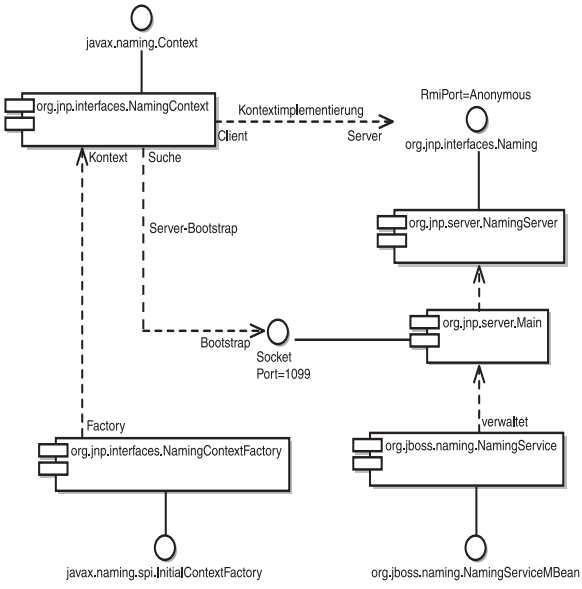

Abbildung 3.1: Die wichtigsten Komponenten der JBossNS-Architektur

Beginnen wir mit der MBean `NamingService`. Sie stellt den JNDI-Namensdienst bereit. Dabei handelt es sich um einen bedeutenden Dienst, der überall von den Komponenten der J2EE-Technolgie verwendet wird. Die konfigurierbaren Attribute für die MBean `NamingService` lauten wie folgt:

- `Port` Der lauschende Port des `jnp`-Protokolls für den `NamingService`. Falls kein anderer Wert angegeben ist, lautet der Standardwert `1099` – wie beim Standardport der RMI-Registrierung.
- `RmiPort` Der RMI-Port, an den die RMI-`Naming`-Implementierung exportiert wird. Ist kein anderer Wert angegeben, lautet der Standardwert `0`, was bedeutet, dass ein beliebiger verfügbarer Port verwendet werden kann.
- `BindAddress` Die Adresse, die der `NamingService` abhört. Dies kann auf einem Host mit mehreren Servern gleichen Typs für einen `java.net.ServerSocket` verwendet werden, der Verbindungsanfragen nur an einer seiner Adressen annimmt.
- `RmiBindAddress` Die Adresse, die der RMI-Serverabschnitt des `NamingService` abhört. Dies kann auf einem Host mit mehreren Servern gleichen Typs für einen `java.net.ServerSocket` verwendet werden, der Verbindungsanfragen nur an einer seiner Adressen annimmt. Ist dieses Attribut nicht festgelegt, aber das Attribut `BindAddress` vorhanden, erhält `RmiBindAddress` standardmäßig den Wert von `BindAddress`.

- `Backlog` Die maximale Warteschlangenlänge für eingehende Verbindungsangaben (eine Verbindungsanfrage), die auf den Parameter `Backlog` gesetzt wird. Wenn eine Verbindungsangabe bei voller Warteschlange eingeht, wird die Verbindung zurückgewiesen.
- `ClientSocketFactory` Eine optionale maßgeschneiderte `java.rmi.server.RMIClientSocketFactory`-Implementierung. Ist dieses Attribut nicht angegeben, wird die standardmäßige `RMIClientSocketFactory` verwendet.
- `ServerSocketFactory` Der Klassenname einer optionalen maßgeschneiderten `java.rmi.server.RMIServerSocketFactory`-Implementierung. Ist dieses Attribut nicht angegeben, wird die standardmäßige `RMIServerSocketFactory` verwendet.
- `JNPServerSocketFactory` Der Klassenname einer optionalen maßgeschneiderten `javax.net.ServerSocketFactory`-Implementierung. Dabei handelt es sich um die Factory für den `ServerSocket`, der zum Starten des Downloads des `Naming`-Interface von JBoss verwendet wird. Ist dieses Attribut nicht angegeben, wird der Wert der Methode `javax.net.ServerSocketFactory.getDefault()` verwendet.

`NamingService` erstellt auch den Kontext `java:comp`, so dass der Zugriff auf diesen Kontext auf der Grundlage des Kontextklassenladers des Threads isoliert wird, der auf den Kontext `java:comp` zugreift. Dadurch wird der private ENC der Anwendungskomponente bereitgestellt, der von der J2EE-Spezifikation gefordert wird. Diese Trennung erfolgt durch die Bindung von `javax.naming.Reference` an einen Kontext, der `org.jboss.naming.ENCFactory` als `javax.naming.ObjectFactory` verwendet. Wenn ein Client eine Suche nach `java:comp` oder einen anderen Unterkontext durchführt, prüft die `ENCFactory` den `ClassLoader` des Threadkontextes und führt unter Verwendung von `ClassLoader` als Schlüssel eine Suche in einer Zuordnung durch.

Wenn eine Kontextinstanz für die Klassenladerinstanz nicht existiert, wird sie erstellt und mit dem Klassenlader in der `ENCFactory`-Zuordnung verknüpft. Daher stützt sich die korrekte Isolierung des ENCs einer Anwendungskomponente darauf, dass die einzelnen Komponenten einen eindeutigen `ClassLoader` erhalten, der mit den Ausführungsthreads der Komponenten verknüpft ist.

Der `NamingService` delegiert seine Funktionalität an die MBean `org.jnp.server.Main`. Doppelte MBeans sind erforderlich, da JBossNS als eigenständige JNDI-Implementierung begonnen hat und auch immer noch als solche ausgeführt werden kann. Die MBean `NamingService` bettet die `Main`-Instanz in den JBoss-Server ein, so dass die Nutzung von JNDI mit derselben VM wie vom JBoss-Server keine zusätzliche Belastung des Sockets mit sich bringt. Die konfigurierbaren Attribute von `Naming Service` sind tatsächlich diejenigen der JBossNS-MBean `Main`. Beim Festlegen von Attributen für die MBean `NamingService` werden einfach die entsprechenden Attribute für die MBean `Main` gesetzt, die in `NamingService` enthalten ist. Wenn der `NamingService`

gestartet wird, führt er die darin enthaltene MBean `Main` aus, um den JNDI- Namensdienst zu aktivieren.

Darüber hinaus legt `NamingService` die `Naming`-Interfaceoperationen über eine nicht typisierte JMX-Aufrufoperation offen. Dies gestattet den Zugang zum Namensdienst über JMX-Adapter für beliebige Protokolle. Weiter hinten in diesem Kapitel werden wir uns ein Beispiel dafür ansehen, wie HTTP für den Zugriff auf den Namensdienst mit Hilfe der Aufrufoperation verwendet werden kann.

Einzelheiten über die Threads und den Klassenlader für den Threadkontext sprengen den Rahmen dieses Buches; die JNDI-Anleitung bietet jedoch eine umfassende Erläuterung. Einzelheiten finden Sie unter *http://java.sun.com/products/jndi/tutorial/beyond/misc/classloader.html*.

Wenn sie gestartet wird, führt die MBean `Main` folgende Aufgaben aus:

1. Sie erstellt eine Instanz des `org.jnp.naming.NamingService` und setzt sie als Serverinstanz der lokalen VM ein. Diese wird von allen `org.jnp.interfaces.NamingContext`-Instanzen verwendet, die innerhalb der JBoss-Server-VM erstellt werden, um RMI-Aufrufe über TCP/IP zu vermeiden.
2. Sie exportiert das `org.jnp.naming.interfaces.Naming`-RMI-Interface der `NamingServer`-Instanz mit Hilfe der konfigurierten Attribute `RmiPort`, `ClientSocketFactory` und `ServerSocketFactoryattributes`.
3. Sie erstellt einen Socket, der das von den Attributen `BindAddress` und `Port` vorgegebene Interface abhört.
4. Sie erzeugt einen Thread, um Verbindungen am Socket anzunehmen.

3.2.1 Die InitialContext-Factorys für Namensdienste

Der JBoss-JNDI-Provider unterstützt derzeit mehrere verschiedene `InitialContext`-Factory-Implementierungen. Die am häufigsten verwendete Factory ist die `org.jnp.interfaces.NamingContextFactory`-Implementierung. Sie hat folgende Eigenschaften:

- `java.naming.factory.initial` Der Name der Umgebungseigenschaft für die Angabe der zu verwendenden Ausgangskontext-Factory. Bei dem Wert der Eigenschaft sollte es sich um einen vollständig qualifizierten Namen der Factory-Klasse handeln, die einen Ausgangskontext erstellt. Ist diese Eigenschaft nicht angegeben, wird beim Erstellen eines `InitialContext`-Objekts eine `javax.naming.NoInitialContextException` ausgelöst.
- `java.naming.provider.url` Der Name der Umgebungseigenschaft für die Angabe des Speicherorts des JBoss-JNDI-Dienstanbieters, den der Client verwenden soll. Die Klasse `NamingContextFactory` nutzt diese Information, um herauszufinden, mit welchem JBossNS-Server eine Verbindung hergestellt werden soll. Bei

dem Wert der Eigenschaft sollte es sich um einen URL-String handeln. Für JBossNS lautet das URL-Format *jnp://host:port/[jndi_path]*. Der URL-Abschnitt `jnp:` ist das Protokoll und verweist auf das von JBoss verwendete Socket-/RMI-basierte Protokoll. Bei dem URL-Abschnitt `jndi_path` handelt es sich um einen optionalen JNDI-Namen, der relativ zum Stammkontext angegeben ist (z.B. `apps` oder `apps/tmp`). Alle Angaben außer der Hostkomponente sind optional. Die folgenden Beispiele sind gleichwertig, da der Standardwert für den Port `1099` lautet:

jnp://www.jboss.org:1099/

www.jboss.org:1099

www.jboss.org

- `java.naming.factory.url.pkgs` Der Name der Umgebungseigenschaft für die Angabe der Liste von Paketpräfixen, die beim Laden von URL-Kontext-Factorys zu verwenden sind. Bei dem Wert der Eigenschaft sollte es sich um eine durch Doppelpunkte getrennte Liste von Paketpräfixen für den Namen der Factory-Klasse handeln, die eine URL-Kontext-Factory erstellt. Für den JBoss-JNDI-Provider muss dies `org.jboss.naming:org.jnp.interfaces` sein. Diese Eigenschaft ist wichtig für die Lokalisierung der URL-Kontext-Factorys `jnp:` und `java:` des JBoss-JNDI-Providers.

- `jnp.socketFactory` Der vollständig qualifizierte Klassenname der `javax.net.SocketFactory`-Implementierung, die für die Erstellung des Bootstrap-Sockets verwendet wird. Der Standardwert lautet `org.jnp.interfaces.Timed SocketFactory`. `TimedSocketFactory` ist eine einfache `SocketFactory`-Implementierung, die die Spezifikation einer Verbindung und eines Lese-Timeouts unterstützt. Diese beiden Eigenschaften werden durch folgende Werte angegeben:

  - `jnp.timeout` Der Verbindungs-Timeout in Millisekunden. Der Standardwert lautet `0`, d.h., dass die Verbindung so lange blockiert wird, bis in der TCP/IP-Schicht der VM ein Timeout auftritt.

  - `jnp.sotimeout` Der Timeout beim Lesen des verbundenen Sockets in Millisekunden. Der Standardwert lautet `0`, d.h., dass der Lesevorgang blockiert wird. Dieser Wert wird an `Socket.setSoTimeout` für den neu verbundenen Socket übergeben.

Wenn ein Client einen `InitialContext` mit diesen JBossNS-Eigenschaften erstellt, wird das Objekt `org.jnp.interfaces.NamingContextFactory` verwendet, um die `Context`-Instanz zu erstellen, die in den folgenden Operationen eingesetzt wird. Bei `NamingContextFactory` handelt es sich um die JBossNS-Implementierung des `javax.naming.spi.InitialContextFactory`-Interface. Wenn die Klasse `Naming ContextFactory` gebeten wird, einen Kontext herzustellen, legt sie eine Instanz des `org.jnp.interfaces.NamingContext`s mit der Umgebung `InitialContext` und dem Namen des Kon-

textes im globalen JNDI-Namensraum an. Die eigentliche Aufgabe der `NamingContext`-Instanz besteht darin, eine Verbindung zum JBossNS-Server herzustellen und das `Context`-Interface zu implementieren. Die `Context.PROVIDER_ URL`-Information von der Umgebung gibt an, von welchem Server eine `NamingServer`-RMI-Referenz abgerufen werden soll.

Die Verknüpfung der `NamingContext`-Instanz mit einer `NamingServer`-Instanz erfolgt bei Bedarf (lazy) mit der ersten ausgeführten `Context`-Operation. Wenn eine `Context`-Operation durchgeführt wird und kein `NamingServer` mit dem `NamingContext` verknüpft ist, schaut dieser nach, ob seine Umgebungseigenschaften einen `Context.PROVIDER_URL` definieren. Ein `Context.PROVIDER_URL` definiert den Host und den Port des JBossNS-Servers, den der `Context` benutzen soll. Ist ein Provider-URL vorhanden, prüft `NamingContext` zunächst, ob eine `Naming`-Instanz mit einem Host/Port-Paar als Schlüssel bereits erstellt wurde, indem er die statische Zuordnung der Klasse `NamingContext` überprüft. Er verwendet einfach die vorhandene `Naming`-Instanz, wenn eine solche bereits für das Host/Port-Paar abgerufen wurde. Wurde keine `Naming`-Instanz für den vorgegebenen Host und den Port erstellt, stellt der `NamingContext` mit Hilfe eines `java.net.Socket` eine Verbindung zu dem Host und dem Port her und ruft einen `Naming`-RMI-Stub vom Server ab, indem er ein `java.rmi.MarshalledObject` vom Socket liest und dessen `get`-Methode aufruft. Die neu erworbene `Naming`-Instanz wird in der Zuordnung im `NamingContext`-Server unter dem Host/Port-Paar im Cache gespeichert. Ist kein Provider-URL in der mit dem Kontext verknüpften JNDI-Umgebung angegeben, verwendet `NamingContext` einfach die von der MBean `Main` in der VM festgelegte `Naming`-Instanz.

Die `NamingContext`-Implementierung des `Context`-Interface delegiert alle Operationen an die mit dem `NamingContext` verknüpfte `Naming`-Instanz. Die Klasse `NamingServer`, die das `Naming`-Interface implementiert, verwendet `java.util.Hashtable` als `Context`-Speicher. Für jeden eindeutigen JNDI-Namen eines vorgegebenen JBossNS-Servers gibt es eine eindeutige `NamingServer`-Instanz. Zu einem gegebenen Zeitpunkt verweisen null oder mehr aktive, flüchtige `NamingContext`-Instanzen auf eine `NamingServer`-Instanz. Der Zweck von `NamingContext` besteht darin, als Kontext für den `Naming`-Interfaceadapter zu fungieren, der die Übersetzung der an `NamingContext` übergebenen JNDI-Namen steuert. Da ein JNDI-Name eine relative Angabe oder ein URL sein kann, muss er im Kontext des JBossNS-Servers, auf den er verweist, in einen absoluten Namen umgewandelt werden. Diese Übersetzung ist eine Schlüsselfunktion von `NamingContext`.

Namenssuche in Clusterumgebungen

Bei der Ausführung in einer JBoss-Clusterumgebung können Sie sich entschließen, keinen `Context.PROVIDER_URL`-Wert anzugeben und den Client das Netzwerk nach verfügbaren Namensdiensten abfragen zu lassen. Dies funktioniert nur bei JBoss-Servern,

die mit der Konfiguration `all` oder einer entsprechenden Konfiguration betrieben werden, die über die bereitgestellten Dienste `org.jboss.ha.framework.server.ClusterPartition` und `org.jboss.ha.jndi.HANamingService` verfügt. Der Suchvorgang besteht darin, ein Multicast-Anfragepaket an die Suchadresse oder den Port zu senden und darauf zu warten, dass irgendein Knoten antwortet. Bei der Antwort handelt es sich um eine HA-RMI-Version des `Naming`-Interface. Die folgenden Eigenschaften von `InitialContext` beeinflussen die Konfiguration der Suche:

- `jnp.partitionName` Der Name der Clusterpartition, auf die die Suche beschränkt werden soll. Wenn Sie sie in einer Umgebung ausführen, die über mehrere Cluster verfügt, wollen Sie die Namenssuche vielleicht auf einen bestimmten Cluster beschränken. Es gibt keinen Standardwert, d.h., dass jede Antwort eines Clusters angenommen wird.
- `jnp.discoveryGroup` Die Multicast-IP-Nummer/-Adresse, an die die Suchanfrage gesendet wird. Der Standardwert lautet `230.0.0.4`.
- `jnp.discoveryPort` Der Port, an den die Suchanfrage gesendet wird. Der Standardwert lautet `1102`.
- `jnp.discoveryTimeout` Die Zeit in Millisekunden, die auf die Antwort zu einer Suchanfrage gewartet wird. Der Standardwert lautet `5000` (5 Sekunden).
- `jnp.disableDiscovery` Ein Flag, das angibt, ob ein Suchvorgang vermieden werden soll. Die Suche erfolgt, wenn entweder kein `Context.PROVIDER_URL` angegeben ist oder wenn kein gültiger Namensdienst unter den angegebenen URLs gefunden werden kann. Hat das Flag `jnp.disableDiscovery` den Wert `true`, wird kein Versuch unternommen, einen Suchvorgang durchzuführen.

Die HTTP-Implementierung der InitialContext-Factory

Auf den JNDI-Namensdienst kann über HTTP zugegriffen werden. Aus der Sicht eines JNDI-Clients ist dies eine unsichtbare Veränderung, da der Client weiterhin das JNDI-`Context`-Interface verwendet. Operationen über das `Context`-Interface werden in HTTP-Posts an ein Servlet übersetzt, das die Anfrage mit Hilfe seiner JMX-Aufrufoperation an `NamingService` weiterleitet. Zu den Vorteilen der Verwendung von HTTP als Zugriffsprotokoll gehören der bessere Zugang über Firewalls und Proxys hinweg, die so eingerichtet sind, dass sie HTTP zulassen, sowie die Möglichkeit, den Zugriff auf den JNDI-Dienst mit Hilfe der standardmäßigen rollenbasierten Sicherheit von Servlets zu schützen. Um über HTTP auf JNDI zuzugreifen, verwenden Sie `org.jboss.naming.HttpNamingContextFactory` als Factory-Implementierung. Im Folgenden sehen Sie den vollständigen Satz der unterstützenden `InitialContext`-Umgebungseigenschaften für diese Factory:

- `java.naming.factory.initial` Dies ist der Name der Umgebungseigenschaft für die Angabe der Ausgangskontext-Factory, bei der es sich um `org.jboss.naming.HttpNamingContextFactory` handeln muss.
- `java.naming.provider.url` (oder `Context.PROVIDER_URL`) Diese Eigenschaft muss auf den HTTP-URL des JMX-Invokerservlets gesetzt werden. Sie hängt zwar von der Konfiguration von `http-invoker.sar` und dem darin enthaltenen WAR ab, die Standardinstallation speichert das JMX-Invokerservlet jedoch unter `/invoker/JMXInvokerServlet`. Der vollständige HTTP-URL ist der öffentliche URL des JBoss-Servletcontainers zuzüglich `/invoker/JMXInvokerServlet`. Es folgen einige Beispiele:

  http://www.jboss.org:8080/invoker/JMXInvokerServlet

  http://www.jboss.org/invoker/JMXInvokerServlet

  https://www.jboss.org/invoker/JMXInvokerServlet

  Das erste Beispiel greift unter Verwendung von Port 8080 auf das Servlet zu. Das zweite verwendet den Standard-HTTP-Port 80 und das dritte nutzt eine SSL-verschlüsselte Verbindung zu dem Standard-HTTPS-Port 443.
- `java.naming.factory.url.pkgs` Für alle JBoss-JNDI-Provider muss dies `org.jboss.naming:org.jnp.interfaces` sein. Diese Eigenschaft ist wichtig zum Auffinden der `jnp:`- und `java:`-URL-Kontext-Factorys für die JBoss-JNDI-Provider.

Die von `HttpNamingContextFactory` zurückgegebene JNDI-`Context`-Implementierung ist ein Proxy, der die über ihn erfolgten Aufrufe an ein Brückenservlet delegiert. Dieses Servlet wiederum leitet den Aufruf über den JMX-Bus an den `NamingService` weiter und die Antwort über HTTP zurück. Um funktionieren zu können, muss der Proxy wissen, wie der URL des Brückenservlets lautet. Dieser Wert kann serverseitig gebunden sein, wenn der JBoss-Webserver über ein bekanntes öffentliches Interface verfügt. Befindet sich der JBoss-Webserver hinter einer oder mehreren Firewalls oder Proxys, kann der Proxy nicht wissen, welcher URL benötigt wird. In diesem Fall wird der Proxy mit einem Systemeigenschaftswert verknüpft, der in der Client-VM festgelegt werden muss. Weitere Informationen über den Betrieb von JNDI über HTTP finden Sie im Abschnitt 3.2.2 »*Zugriff auf JNDI über HTTP*«.

Die Implementierung der InitialContext-Factory für die Anmeldung

JAAS ist die bevorzugte Methode für die Authentifizierung eines Remoteclients bei JBoss. Der Einfachheit halber und um die Migration von anderen Anwendungsserverumgebungen zu erleichtern, die JAAS nicht verwenden, gestattet JBoss jedoch, dass die Sicherheitsinformationen über den `InitialContext` übergeben werden. JAAS wird im Hintergrund weiterhin benutzt, aber es besteht keine offensichtliche Verwendung der JAAS-Interfaces in der Clientanwendung. Bei der Factory-Klasse, die diese Fähig-

keit bereitstellt, handelt es sich um `org.jboss.security.jndi.LoginInitialContext-Factory`. Im Folgenden finden Sie die gesamten unterstützenden Eigenschaften der `InitialContext`-Umgebung für diese Factory:

- `java.naming.factory.initial` Dies ist der Name der Umgebungseigenschaft zur Angabe der Ausgangskontext-Factory, bei der es sich um `org.jboss.security.jndi.LoginInitialContextFactory` handeln muss.
- `java.naming.provider.url` Diese Eigenschaft muss auf den URL eines `NamingContextFactory`-Providers gesetzt werden. `LoginInitialContext` ist nur ein Wrapper um die `NamingContextFactory`, der dem vorhandenen `Naming ContextFactory`-Verhalten eine JAAS-Ameldung hinzufügt.
- `java.naming.factory.url.pkgs` Für alle JBoss-JNDI-Provider muss dies `org.jboss.naming:org.jnp.interfaces` sein. Diese Eigenschaft ist wichtig zur Lokalisierung der `jnp:`- und `java:`-URL-Kontext-Factorys des JBoss-JNDI-Providers.
- `java.naming.security.principal` (oder `Context.SECURITY_PRINCIPAL)` Dies ist der zu authentifizierende Prinzipal. Dabei kann es sich entweder um eine `java.security.Principal`-Implementierung oder einen String handeln, der den Namen eines Prinzipals darstellt.
- `java.naming.security.credentials` (oder `Context.SECURITY_CREDENTIALS)` Dabei handelt es sich um die Nachweise, die zur Authentifizierung des Prinzipals verwendet werden sollen (z.B. Passwort, Sitzungsschlüssel usw.).
- `java.naming.security.protocol` (oder `Context.SECURITY_PROTOCOL`) Diese Eigenschaft gibt den Namen des JAAS-Anmeldemoduls an, das für die Authentifizierung des Prinzipals und für Nachweise verwendet wird.

3.2.2 Zugriff auf JNDI über HTTP

Zusätzlich zu dem veralteten RMI/JRMP mit einem Socket-Bootstrap-Protokoll bietet JBoss die Unterstützung für den Zugriff auf seinen JNDI-Namensdienst über HTTP. Diese Fähigkeit wird von `http-invoker.sar` bereitgestellt. Im Folgenden sehen Sie die Struktur von `http-invoker.sar`:

```
http-invoker.sar
+- META-INF/jboss-service.xml
+- invoker.war
| +- WEB-INF/jboss-web.xml
| +- WEB-INF/classes/org/jboss/invocation/http/servlet/
                          InvokerServlet.class
| +- WEB-INF/classes/org/jboss/invocation/http/servlet/
                          NamingFactoryServlet.class
| +- WEB-INF/classes/org/jboss/invocation/http/servlet
                         /ReadOnlyAccessFilter.class
| +- WEB-INF/classes/roles.properties
| +- WEB-INF/classes/users.properties
| +- WEB-INF/web.xml
| +- META-INF/MANIFEST.MF
+- META-INF/MANIFEST.MF
```

Der Deskriptor `jboss-service.xml` definiert die MBeans `HttpInvoker` und `HttpInvokerHA`. Diese Dienste verarbeiten die Weiterleitung von Methodenaufrufen, die über HTTP an die entsprechende Ziel-MBean auf dem JMX-Bus gesendet werden.

Die Webanwendung `http-invoker.war` enthält Servlets, die die mit dem HTTP-Transport zusammenhängenden Einzelheiten regeln. `NamingFactoryServlet` verarbeitet Anfragen zur Erstellung der Implementierung des JBoss-JNDI-Namensdienstes. `javax.naming.Context`. `InvokerServlet` verarbeitet die von RMI/HTTP-Clients kommenden Aufrufe. `ReadOnlyAccessFilter` erlaubt Ihnen, den JNDI-Namensdienst zu sichern, während Sie einen einzelnen JNDI-Kontext für den schreibgeschützten Zugriff durch nicht authentifizierte Clients zur Verfügung stellen.

Bevor wir uns den Konfigurationen zuwenden, wollen wir uns die Operation der `http-invoker`-Dienste ansehen. Abbildung 3.2 zeigt eine logische Ansicht der Struktur eines JBoss-JNDI-Proxys und dessen Beziehung zu den serverseitigen Komponenten von `http-invoker`. Der Proxy wird vom `NamingFactoryServlet` abgerufen, wozu ein `InitialContext` mit der auf `org.jboss.naming.HttpNamingContextFactory` gesetzten Eigenschaft `Context.INITIAL_CONTEXT_FACTORY` und der auf den HTTP-URL des `NamingFactoryServlet`s gesetzten Eigenschaft `Context.PRO VIDER_URL` verwendet wird. Der daraus resultierende Proxy ist in eine `org.jnp. interfaces.NamingContext`-Instanz eingebettet, die die Implementierung des `Context`-Interface bereitstellt.

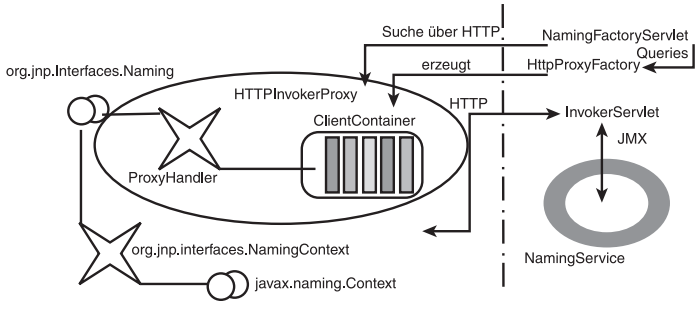

Abbildung 3.2: Die HTTP-Invoker-Proxy/Server-Struktur eines JNDI-Kontextes

Der Proxy ist eine Instanz des `org.jboss.invocation.http.interfaces.HttpInvokerProxy`s und implementiert das `org.jnp.interfaces.Naming`-Interface. Intern enthält der `HttpInvokerProxy` einen Invoker, der die Methodenaufrufe für das `Naming`-Interface über HTTP-Posts an das `InvokerServlet` leitet. Das `InvokerServlet` übersetzt diese Posts in JMX-Aufrufe an den `NamingService` und gibt die Antwort an den Proxy in der Antwort auf das HTTP-Post zurück.

Um all diese Komponenten miteinander zu verbinden, müssen mehrere Konfigurationswerte festgelegt werden. Abbildung 3.3 verdeutlicht die Beziehungen zwischen den Konfigurationsdateien und den entsprechenden Komponenten.

Abbildung 3.3: Die Beziehung zwischen den Konfigurationsdateien und der JNDI/HTTP-Komponente

Der Deskriptor `http-invoker.sar/META-INF/jboss-service.xml` definiert die `HttpProxyFactory`, die den `HttpInvokerProxy` für den `NamingService` erstellt. Zu den Attributen, die für die `HttpProxyFactory` konfiguriert werden müssen, gehören folgende:

- `InvokerName` Der JMX-`ObjectName` des `NamingService`, der im Deskriptor `conf/jboss-service.xml` definiert wird. Die in den JBoss-Distributionen verwendete Standardeinstellung lautet `jboss:service=Naming`.
- `InvokerURL` oder `InvokerURLPrefix` + `InvokerURLSuffix` + `UseHostName` Sie können den vollständigen HTTP-URL zum `InvokerServlet` festlegen, indem Sie das Attribut `InvokerURL` verwenden, oder die vom Hostnamen unabhängigen Teile des URLs festlegen und von der `HttpProxyFactory` einsetzen lassen. Ein Beispiel für einen `InvokerURL`-Wert lautet: *http://jbosshost1.dot.com:8080/invoker/JMXInvokerServlet*. Dieser Wert kann in folgende Bestandteile aufgelöst werden:
  - `InvokerURLPrefix` Das URL-Präfix vor dem Hostnamen. Normalerweise handelt es sich dabei um `http://` bzw. `https://`, wenn SSL verwendet werden soll.
  - `InvokerURLSuffix` Das URL-Suffix hinter dem Hostnamen. Dies schließt die Portnummer des Webservers sowie den bereitgestellten Pfad zum `InvokerServlet` mit ein. Das `InvokerURLSuffix` für den Beispielwert von `Invoker URL` würde `:8080/invoker/JMXInvokerServlet` lauten. Die Portnummer wird von den Einstellungen des Webcontainerdienstes bestimmt. Der Pfad zum `InvokerServlet` wird im Deskriptor `http-invoker.sar/invoker.war/WEB-INF/web.xml` angegeben.
- `UseHostName` Ein Flag, das angibt, ob der Hostname im Hostabschnitt des vollständigen `InvokerURL`s anstelle der IP-Adresse des Hosts erscheinen soll. Lautet der Wert `true`, wird die Methode `InetAddress.getLocalHost().getHostName()`, andernfalls die Methode `InetAddress.getLocalHost().getHost Address()` verwendet.
- `ExportedInterface` Das `org.jnp.interfaces.Naming`-Interface, das der Proxy für die Clients offen legt. Der eigentliche Client dieses Proxys ist die Klasse `NamingContext` für die JBoss-JNDI-Implementierung, die ein JNDI-Client beim Durchsuchen von `InitialContext` erhält, wenn der JBoss-JNDI-Provider verwendet wird.

- `JndiName` Der Name in JNDI, unter dem der Proxy gebunden wird. Er muss auf einen leeren String gesetzt werden, um zu kennzeichnen, dass das Interface nicht in JNDI eingebunden werden soll. Sie können JNDI nicht dazu verwenden, sich selbst zu laden. Das ist die Rolle von `NamingFactoryServlet`.

Der Deskriptor `http-invoker.sar/invoker.war/WEB-INF/web.xml` definiert die Zuordnungen von `NamingFactoryServlet` und `InvokerServlet` zusammen mit ihren Initialisierungsparametern. Die für JNDI/HTTP relevante Konfiguration des `NamingFactoryServlet`s ist der Eintrag `JNDIFactory`, der Folgendes definiert:

- Den Initialisierungsparameter `namingProxyMBean`, der dem Namen der MBean `HttpProxyFactory` zugeordnet wird. Dieser wird von `NamingFactoryServlet` verwendet, um den `Naming`-Proxy abzurufen, der als Antwort auf HTTP-Posts zurückgegeben wird. Für die Standardeinstellungen von `http-invoker.sar/META-INF/jboss-service.xml` lautet der Name `jboss:service=invoker,type=http,target=Naming`.
- Einen Initialisierungsparameter `proxyAttribute`, der den Namen des Attributs der `namingProxyMBean` definiert, um den `Naming`-Proxywert abzufragen. Dabei handelt es sich standardmäßig um den Attributnamen `Proxy`.
- Die Servlet-Zuordnung für die `JNDIFactory`-Konfiguration. Die Standardeinstellung für die ungesicherte Zuordnung lautet `/JNDIFactory/*`. Diese Angabe erfolgt relativ zum Kontextstamm von `http-invoker.sar/invoker.war`, wobei es sich standardmäßig um den Namen des WARs ohne das Suffix `.war` handelt.

Die für JNDI/HTTP relevante Konfiguration von `InvokerServlet` ist das `JMXInvokerServlet`, das die Servletzuordnung von `InvokerServlet` definiert. Die Standardeinstellung für die ungesicherte Zuordnung lautet `/JMXInvokerServlet/*`. Diese Angabe erfolgt relativ zum Kontextstamm von `http-invoker.sar/invoker.war`, wobei es sich standardmäßig um den Namen des WARs ohne das Suffix `.war` handelt.

3.2.3 Zugriff auf JNDI über HTTPS

Um über HTTP/SSL auf JNDI zugreifen zu können, müssen Sie einen SSL-Connector für den Webcontainer aktivieren. Näheres zu diesem Thema wird in Kapitel 9, »*Webanwendungen*«, erläutert. Dieser Abschnitt zeigt die Verwendung von HTTPS für einen einfachen Beispielclient, der einen HTTPS-URL als URL des JNDI-Providers verwendet. Dieses Beispiel umfasst die Konfiguration des SSL-Connectors, so dass es in sich geschlossen ist, sofern Sie sich nicht für die Einrichtung des SSL-Connectors im Einzelnen interessieren.

Außerdem stellt dieses Beispiel eine Konfiguration der `HttpProxyFactory`-Installation für die Verwendung eines HTTPS-URLs bereit. Im Folgenden sehen Sie den Abschnitt des `jboss-service.xml`-Deskriptors `http-invoker.sar`, den das Beispiel zur Bereitstellung dieser Konfiguration installiert:

```
<!-- Das Naming-Dienstinterface über HTTPS offen legen -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
       name="jboss:service=invoker,type=https,target=Naming">
    <!-- Der Naming-Dienst, für den wir einen Proxy einrichten
    -->
    <attribute name="InvokerName">jboss:service=Naming
    </attribute>
    <!-- Den URL des Invokers aus der Adresse des Clusterknotens
         zusammensetzen -->
    <attribute name="InvokerURLPrefix">https://</attribute>
    <attribute name="InvokerURLSuffix">
        :8443/invoker/JMXInvokerServlet
    </attribute>
    <attribute name="UseHostName">true</attribute>
    <attribute name="ExportedInterface">org.jnp.interfaces.
    Naming </attribute>
    <attribute name="JndiName"/>
    <attribute name="ClientInterceptors">
        <interceptors>
            <interceptor>
                org.jboss.proxy.ClientMethodInterceptor
            </interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor
            </interceptor>
            <interceptor>
                org.jboss.naming.interceptors.
               ExceptionInterceptor
            </interceptor>
            <interceptor>
                org.jboss.invocation.InvokerInterceptor
            </interceptor>
        </interceptors>
    </attribute>
</mbean>
```

Alles, was sich im Hinblick auf die HTTP-Standardkonfiguration geändert hat, sind die Attribute `InvokerURLPrefix` und `InvokerURLSuffix`, die mit Hilfe von Port 8443 einen HTTPS-URL einrichten.

Zumindest müssen Sie für einen JNDI-Client unter Verwendung von HTTPS einen HTTPS-URL-Protokollhandler einrichten. In diesem Beispiel wird die **J**ava **S**ecure **S**ocket **E**xtension (JSSE) für HTTPS eingesetzt. Die JSSE-Dokumentation beschreibt sehr gut, was für den Einsatz von HTTPS erforderlich ist. Zum Einrichten des in Listing 3.16 gezeigten Beispielclients müssen Sie die folgenden Schritte beachten:

1. Sie müssen Java einen Protokollhandler für HTTPS-URLs zur Verfügung stellen. Das Paket `com.sun.net.ssl.internal.www.protocol` aus der JSSE-Version enthält einen HTTPS-Handler. Um die Verwendung von HTTPS-URLs zu aktivieren, fügen Sie dieses Paket in die Standardsucheigenschaft `java.protocol.handler.pkgs` für den URL-Protokollhandler ein. Sie legen diese Eigenschaft im Ant-Skript fest.

2. Damit SSL funktioniert, installieren Sie den JSSE-Sicherheitsprovider. Dazu installieren Sie entweder die JSSE-JARs als Erweiterungspaket oder programmatisch. Dieses Beispiel verwendet den programmatischen Ansatz, da es sich dabei um die weniger störende Methode handelt.

3. Der URL des JNDI-Providers muss HTTPS als Protokoll enthalten. Die Zeilen 15-16 des `ExClient`-Codes geben eine HTTP/SSL-Verbindung zu `localhost` an Port 8443 an. Der Hostname und der Port werden vom SSL-Connector des Webcontainers festgelegt.

4. Sie deaktivieren die Validierung des HTTPS-URL-Hostnamens anhand des Serverzertifikats. Standardmäßig führt der JSSE-HTTPS-Protokollhandler anhand des aus dem Serverzertifikat bekannten Namens eine strenge Validierung des Abschnitts aus dem HTTPS-URL durch, der den Hostnamen angibt. Dabei handelt es sich um dieselbe Prüfung, die Webbrowser vornehmen, wenn Sie eine Verbindung zu einer gesicherten Website herstellen. Das Beispiel in Listing 3.16 benutzt ein selbst signiertes Serverzertifikat, das den allgemeinen Namen »Chapter 8 SSL Example« anstelle eines besonderen Hostnamens verwendet, was wahrscheinlich in Entwicklungsumgebungen und Intranets auch so üblich ist. Der `HttpInvokerProxy` von JBoss überschreibt die Standardprüfung des Hostnamens, wenn die Systemeigenschaft `org.jboss.security.ignoreHttpsHost` vorhanden ist und den Wert `true` aufweist. Im Ant-Skript setzen Sie den Wert der Eigenschaft `org.jboss.security.ignoreHttpsHost` auf `true`.

Listing 3.16: Ein JNDI-Client, der für den Transport HTTPS verwendet

```
package org.jboss.chap3.ex1;

import java.security.Security;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Properties env = new Properties();
        env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                   "org.jboss.naming.HttpNamingContextFactory");
        env.setProperty(Context.PROVIDER_URL,
                        "https://localhost:8443/invoker/
                        JNDIFactorySSL");

        Context ctx = new InitialContext(env);
        System.out.println("Created InitialContext, env="
                           + env);

        Object data = ctx.lookup("jmx/invoker/RMIAdaptor");
        System.out.println("lookup(jmx/invoker/RMIAdaptor): "
                           + data);
    }
}
```

Um den Client zu testen, erstellen Sie zunächst das Beispiel zu Kapitel 3, um den `chap3`-Konfigurationsdateisatz anzulegen:

```
[examples]$ ant -Dchap=chap3 config example
```

Im nächsten Schritt starten Sie den JBoss-Server mit Hilfe des Konfigurationsdateisatzes `chap3`:

```
[bin]$ sh run.sh -c chap3
```

Abschließend führen Sie `ExClient` mit Hilfe des folgenden Codes aus:

```
[examples]$ ant -Dchap=chap3 -Dex=1 run-example
...
run-example1:
     [java] Created InitialContext,env={java.naming.provider.url=https://
localhost:8443/invoker/JNDIFactorySSL,java.naming.factory.
initial=org.jboss.naming.HttpNamingContextFactory}
     [java] lookup(jmx/invoker/
RMIAdaptor):org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxy @cac3fa
```

3.2.4 Den HTTP-Zugriff auf JNDI sichern

Ein Vorteil des Zugriffs auf JNDI über HTTP liegt in dem leicht zu sichernden Zugang zu der JNDI-`InitialContext`-Factory sowie in den Namensdienst-Operationen unter Verwendung der deklarativen Standardwebsicherheit. Dies ist möglich, weil die serverseitige Behandlung des JNDI/HTTP-Transports mit zwei Servlets implementiert wird. Diese Servlets befinden sich im Verzeichnis `http-invoker.sar/invoker.war`, das Sie, wie bereits zuvor gezeigt wurde, in den `deploy`-Verzeichnissen der Konfigurationen `default` und `all` finden. Um den gesicherten Zugang zu JNDI zu aktivieren, müssen Sie den Deskriptor `invoker.war/WEB-INF/web.xml` ändern und alle ungesicherten Servletzuordnungen entfernen. Der in Listing 3.17 gezeigte Deskriptor `web.xml` gewährt zum Beispiel nur dann Zugang zu den `invoker.war`-Servlets, wenn der Benutzer authentifiziert wurde und über die Rolle `HttpInvoker` verfügt.

Listing 3.17: Ein Beispiel für einen `web.xml`-Deskriptor für den gesicherten Zugriff auf die JNDI-Servlets

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
          "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
          "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- ### Servlets -->
    <servlet>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <servlet-class>
            org.jboss.invocation.http.servlet.InvokerServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet> <servlet>
```

Listing 3.17: Ein Beispiel für einen `web.xml`-Deskriptor für den gesicherten Zugriff auf die JNDI-Servlets (Fortsetzung)

```
        <servlet-name>JNDIFactory</servlet-name>
        <servlet-class>
            org.jboss.invocation.http.servlet.NamingFactoryServlet
    </servlet-class>
    <init-param>
        <param-name>namingProxyMBean</param-name>
        <param-value>
            jboss:service=invoker,type=http,target=Naming
        </param-value>
    </init-param>
        <init-param>
            <param-name>proxyAttribute</param-name>
            <param-value>Proxy</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <!-- ### Servlet-Zuordnungen -->
    <servlet-mapping>
        <servlet-name>JNDIFactory</servlet-name>
        <url-pattern>/restricted/JNDIFactory/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <url-pattern>/restricted/JMXInvokerServlet/*</url-pattern>
    </servlet-mapping> <security-constraint>
        <web-resource-collection>
            <web-resource-name>HttpInvokers</web-resource-name>
            <description>
                An example security config that only allows users
                with the role HttpInvoker to access the HTTP
                invoker servlets
            </description>
            <url-pattern>/restricted/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>HttpInvoker</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>JBoss HTTP Invoker</realm-name>
    </login-config> <security-role>
        <role-name>HttpInvoker</role-name>
```

Listing 3.17: Ein Beispiel für einen `web.xml`-Deskriptor für den gesicherten Zugriff auf die JNDI-Servlets (Fortsetzung)

```
    </security-role>
</web-app>
```

Der Deskriptor `web.xml` legt lediglich fest, welche Servlets gesichert werden und welchen Rollen der Zugang zu den gesicherten Servlets gewährt wird. Zusätzlich müssen Sie die Sicherheitsdomäne festlegen, die die Authentifizierung und Autorisierung für das WAR verarbeitet. Dies geschieht durch den Deskriptor `jboss-web.xml`. Das folgende Beispiel verwendet die Domäne `http-invoker security`:

```
<jboss-web>
   <security-domain>java:/jaas/http-invoker</security-domain>
</jboss-web>
```

Das Element `security-domain` legt den Namen der Sicherheitsdomäne fest, die für die Konfiguration des JAAS-Anmeldemoduls zur Authentifizierung und Autorisierung verwendet wird.

Weitere Einzelheiten über die Bedeutung und die Konfiguration von Sicherheitsdomänennamen finden Sie in Kapitel 8, *»Sicherheitsaspekte von JBoss«*.

3.2.5 Den JNDI-Zugriff in einem schreibgeschützten, unsicheren Kontext sichern

Eine weitere Funktion, die für den JNDI/HTTP-Namensdienst zur Verfügung steht, ist die Möglichkeit, einen Kontext zu definieren, der für nicht autorisierte Benutzer im schreibgeschützten Modus zugänglich ist. Dies kann bei Diensten von Bedeutung sein, die von der Authentifizierungsschicht genutzt werden. Beispielsweise muss das `SRP-LoginModule` das zur Authentifizierung verwendete SRP-Serverinterface suchen. Sehen wir uns nun an, wie der schreibgeschützte JNDI-Zugriff in JBoss funktioniert.

Zunächst wird die `ReadOnlyJNDIFactory` in `invoker.sar/WEB-INF/web.xml` deklariert. Sie wird `/invoker/ReadOnlyJNDIFactory` zugeordnet:

```
<servlet>
    <servlet-name>ReadOnlyJNDIFactory</servlet-name>
    <description>A servlet that exposes the JBoss JNDI Naming service stub
          through http, but only for a single read-only context.
          The return content is serialized MarshalledValue
          containing the org.jnp.interfaces.Naming stub.
    </description>
<servlet-class>
    org.jboss.invocation.http.servlet.NamingFactoryServlet
</servlet-class>
    <init-param>
        <param-name>namingProxyMBean</param-name>
        <param-value>
            jboss:service=invoker,type=http,target=Naming,
                                        readonly=true
        </param-value>
    </init-param>
```

```
        <init-param>
            <param-name>proxyAttribute</param-name>
            <param-value>Proxy</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <!-- ... -->
    <servlet-mapping>
        <servlet-name>ReadOnlyJNDIFactory</servlet-name>
        <url-pattern>/ReadOnlyJNDIFactory/*</url-pattern>
    </servlet-mapping>
```

Die Factory stellt nur einen JNDI-Stub bereit, der mit einem Invoker verbunden werden muss. In diesem Fall handelt es sich um den Invoker `jboss:service=invoker,type=http,target=Naming,readonly=true`. Dieser wird in der Datei `http-invoker.sar/META-INF/jboss-service.xml file` deklariert:

```
    <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
           name="jboss:service=invoker,type=http,target=Naming,
                 readonly=true">
        <attribute name="InvokerName">jboss:service=Naming</attribute>
        <attribute name="InvokerURLPrefix">http://</attribute>
        <attribute name="InvokerURLSuffix">
            :8080/invoker/readonly/JMXInvokerServlet
        </attribute>
        <attribute name="UseHostName">true</attribute>
        <attribute name="ExportedInterface">org.jnp.interfaces.Naming</attribute>
        <attribute name="JndiName"></attribute>
        <attribute name="ClientInterceptors">
            <interceptors>
                <interceptor>org.jboss.proxy.ClientMethodInterceptor
                </interceptor>
                <interceptor>org.jboss.proxy.SecurityInterceptor
                </interceptor>
                <interceptor>
                    org.jboss.naming.interceptors.ExceptionInterceptor
                </interceptor>
                <interceptor>org.jboss.invocation.InvokerInterceptor
                </interceptor>
            </interceptors>
        </attribute>
    </mbean>
```

Der clientseitige Proxy muss einem bestimmten serverseitigen Invoker-Servlet antworten. Die hier gezeigte Konfiguration leitet die Aufrufe an den `/invoker/readonly/JMXInvokerServlet`-Namensdienst. Dies ist sogar das Standard-`JMXInvokerServlet` mit einem hinzugefügten Schreibschutzfilter:

```
        <filter>
            <filter-name>ReadOnlyAccessFilter</filter-name>
            <filter-class>
                org.jboss.invocation.http.servlet.
                ReadOnlyAccessFilter
            </filter-class>
            <init-param>
                <param-name>readOnlyContext</param-name>
                <param-value>readonly</param-value>
                <description>The top level JNDI context the filter
                    will enforce read-only access on. If specified
                    only Context.lookup operations will be allowed
                    on this context. Other operations or lookups on
                    any other context will fail. Do not associate
                    this filter with the JMXInvokerServlets if you
                    want unrestricted access. </description>
```

```
        </init-param>
        <init-param>
            <param-name>invokerName</param-name>
            <param-value>jboss:service=Naming</param-value>
            <description>
                The JMX ObjectName of the naming service mbean
            </description>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>ReadOnlyAccessFilter</filter-name>
        <url-pattern>/readonly/*</url-pattern>
    </filter-mapping>
    <!-- ... -->
    <!-- Eine Zuordnung für das JMXInvokerServlet, die Aufrufe
         von Lookups nur in schreibgeschützten Kontext erlaubt.
         Wird von ReadOnlyAccessFilter durchgesetzt.
    -->
    <servlet-mapping>
        <servlet-name>JMXInvokerServlet</servlet-name>
        <url-pattern>/readonly/JMXInvokerServlet/*</url-pattern>
    </servlet-mapping>
```

Der Parameter `readOnlyContext` ist auf `readonly` gesetzt, d.h., dass Sie beim Zugang zu JBoss über `ReadOnlyJNDIFactory` nur auf die Daten im Kontext `readonly` zugreifen können. Das folgende Codefragment verdeutlicht diese Verwendungsweise:

```
Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
                "http://localhost:8080/invoker/
                ReadOnlyJNDIFactory");
Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly/data");
```

Versuche, Objekte außerhalb des `readonly`-Kontextes zu suchen, werden fehlschlagen. Beachten Sie, dass der Kontext `readonly` bei der Auslieferung von JBoss keine Daten enthält, so dass Sie ihn erst erstellen müssen, um ihn nutzen zu können.

3.2.6 Weitere Namensdienst-MBeans

Zusätzlich zu der MBean `NamingService`, die einen in JBoss eingebetteten JBossNS-Server konfiguriert, gibt es drei weitere MBean-Dienste für die Benennung, die mit JBoss ausgeliefert werden: `ExternalContext`, `NamingAlias` und `JNDIView`.

Die MBean org.jboss.naming.ExternalContext

Die MBean `ExternalContext` erlaubt Ihnen, externe JNDI-Kontexte im JNDI-Namensraum des JBoss-Servers zu vereinen. Der Begriff *extern* bezieht sich auf alle Namensdienste außerhalb des JBossNS-Namensdienstes, die innerhalb der VM des JBoss-Servers ausgeführt werden. Sie können LDAP-Server, Dateisysteme, DNS-Server usw. integrieren, selbst wenn der Stammkontext des JNDI-Providers nicht serialisierbar ist. Der Zusammenschluss kann Remoteclients zur Verfügung gestellt werden, wenn der Namensdienst den Remotezugriff unterstützt.

Um einen externen JNDI-Namensdienst einzufügen, müssen Sie der Datei `jboss-service.xml` die Konfiguration des MBean-Dienstes `ExternalContext` hinzufügen. Die konfigurierbaren Attribute des Dienstes `ExternalContext` sind im Folgenden aufgeführt:

- `JndiName` Der JNDI-Name, unter dem der externe Kontext gebunden ist.
- `RemoteAccess` Ein Boole'sches Flag, das angibt, ob der externe `InitialContext` in serialisierbarer Form gebunden werden sollte, was einem Remoteclient erlaubt, den externen `InitialContext` zu erstellen. Wenn ein Remoteclient den externen Kontext über JBoss-JNDI-`InitialContext` sucht, erstellt der Client gewissermaßen eine Instanz des externen `InitialContext`s und verwendet dabei dieselben `env`–Eigenschaften, die an die MBean `ExternalContext` übergeben werden. Dies funktioniert nur, wenn der Client einen neuen `InitialContext (env)` von einem entfernten Standort aus erstellen kann. Dazu ist es erforderlich, dass der `Context.PROVIDER_URL`-Wert von `env` in der Remote-VM aufgelöst werden kann, die auf den Kontext zugreift. Dies sollte beim Beispiel für LDAP funktionieren. Beim Beispiel für das Dateisystem funktioniert es wahrscheinlich meistens nicht, solange der Dateisystempfad auf einen allgemeinen Netzwerkpfad verweist. Ist diese Eigenschaft nicht vorgegeben, lautet der Standardwert `false`.
- `CacheContext` Das Flag `cacheContext`. Erhält es den Wert `true`, wird der externe Kontext nur beim Starten der MBean erstellt und dann so lange als speicherinternes Objekt festgehalten, bis die MBean beendet wird. Wird der Wert für `cacheContext` auf `false` gesetzt, so wird der externe `Context` mit Hilfe der MBean-Eigenschaften und der Klasse `InitialContext` bei jeder Suche erstellt. Sucht ein Client den nicht im Cache gespeicherten Kontext, sollte er für diesen Kontext `close()` aufrufen, um Ressourcenlecks zu verhindern.
- `InitialContext` Der vollständig qualifizierte Klassenname der zu verwendenden `InitialContext`-Implementierung. Dabei muss es sich um einen der folgenden Namen handeln: `javax.naming.InitialContext`, `javax.naming.directory.InitialDirContext` oder `javax.naming.ldap.InitialLdapContext`. Im Fall von `InitialLdapContext` wird ein leeres `Controls`-Array verwendet. Der Standardwert lautet `javax.naming.InitialContext`.
- `Properties` Die JNDI-Eigenschaften für den externen `InitialContext`. Bei der Eingabe sollte es sich um einen Text handeln, der dem in die Datei `jndi.properties` eingefügten Text entspricht.
- `PropertiesURL` Die `jndi.properties`-Informationen für den externen `InitialContext` aus einer externen Eigenschaftsdatei. Dabei handelt es sich entweder um einen URL, einen String oder den Namen einer Klassenpfadressource. Im Folgenden sehen Sie einige Beispiele:

```
file:///config/myldap.properties
http://config.mycompany.com/myldap.properties
/conf/myldap.properties
myldap.properties
```

Die folgende MBean-Definition weist eine Bindung an einen externen LDAP-Kontext im JBoss-JNDI-Namensraum unter dem Namen `external/ldap/jboss` auf:

```
<!-- Einen Remote-LDAP-Server binden -->
<mbean code="org.jboss.naming.ExternalContext"
       name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/
     jboss">
    <attribute name="JndiName">external/ldap/jboss</attribute>
    <attribute name="Properties">
        java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
        java.naming.provider.url=ldap://ldaphost.jboss.org:389/
                                 o=jboss.org
        java.naming.security.principal=cn=Directory Manager
        java.naming.security.authentication=simple
        java.naming.security.credentials=secret
    </attribute>
    <attribute name="InitialContext">
        javax.naming.ldap.InitialLdapContext
    </attribute>
    <attribute name="RemoteAccess">true</attribute>
</mbean>
```

Mit dieser Konfiguration können Sie aus der JBoss-VM heraus auf den externen LDAP-Kontext zugreifen, der sich unter `ldap://ldaphost.jboss.org:389/o=jboss.org` befindet, wenn Sie das folgende Codefragment benutzen:

```
InitialContext iniCtx = new InitialContext();
LdapContext ldapCtx = iniCtx.lookup("external/ldap/jboss");
```

Die Verwendung desselben Codefragments außerhalb der VM des JBoss-Servers wird in diesem Fall ebenfalls funktionieren, da der Wert der Eigenschaft `RemoteAccess` auf `true` gesetzt ist. Lautete der Wert `false`, würde das Codefragment nicht funktionieren, da der Remoteclient ein `Reference`-Objekt mit einer `ObjectFactory` erhielte, die nicht in der Lage wäre, den externen `InitialContext` neu zu erstellen:

```
<!-- Das Dateisystemverzeichnis /usr/local binden -->
<mbean code="org.jboss.naming.ExternalContext"
       name="jboss.jndi:service=ExternalContext,
       jndiName=external/fs/usr/local">
    <attribute name="JndiName">external/fs/usr/local</attribute>
    <attribute name="Properties">
        java.naming.factory.initial=com.sun.
        jndi.fscontext.RefFSContextFactory
        java.naming.provider.url=file:///usr/local
    </attribute>
    <attribute name="InitialContext">javax.naming.InitialContext</attribute>
</mbean>
```

Diese Konfiguration beschreibt die Bindung des lokalen Dateisystemverzeichnisses `/usr/local` an den JBoss-JNDI-Namensraum unter dem Namen `external/fs/usr/local`.

Bei dieser Konfiguration können Sie mit Hilfe des folgenden Codefragments aus der JBoss-VM heraus auf den Kontext des externen Dateisystems zugreifen, der sich in der Datei `///usr/local` befindet:

```
InitialContext iniCtx = new InitialContext();
Context ldapCtx = iniCtx.lookup("external/fs/usr/local");
```

Bitte beachten Sie, dass dieser Code einen der JNDI-Dienstanbieter von Sun verwendet, der von *http://java.sun.com/products/jndi/serviceproviders.html* heruntergeladen werden muss. Die JARs des Providers sollten im Serverkonfigurationsverzeichnis `lib` gespeichert werden.

Die MBean org.jboss.naming.NamingAlias

Bei der MBean `NamingAlias` handelt es sich um einen einfachen Hilfsdienst, der Ihnen erlaubt, einen Alias in Form einer JNDI-`javax.naming.LinkRef` von einem JNDI-Namen auf einen anderen zu erstellen. Dies weist Ähnlichkeiten mit einer symbolischen Verknüpfung im Unix-Dateisystem auf. Für einen Alias fügen Sie der Konfigurationsdatei `jboss-service.xml` eine Konfiguration der MBean `NamingAlias` hinzu.

Für den `NamingAlias`-Dienst gelten die folgenden konfigurierbaren Attribute:

- `FromName` Der Speicherort, an dem `LinkRef` unter JNDI eingebunden wird.
- `ToName` Der Empfängername des Alias. Dabei handelt es sich um den Zielnamen, auf den `LinkRef` verweist. Der Name ist ein URL oder ein Name, der relativ zum `InitialContext` aufgelöst werden muss; wenn es sich bei dem ersten Zeichen des Namens um einen Punkt (`.`) handelt, ist der Name relativ zu dem Kontext angegeben, in dem die Verknüpfung eingebunden ist.

Das folgende Beispiel stellt eine Zuordnung des JNDI-Namens `QueueConnectionFactory` zu dem Namen `ConnectionFactory` bereit:

```
<mbean code="org.jboss.naming.NamingAlias"
       name="jboss.mq:service=NamingAlias,fromName=
             QueueConnectionFactory">
    <attribute name="ToName">ConnectionFactory</attribute>
    <attribute name="FromName">QueueConnectionFactory
    </attribute>
</mbean>
```

Die MBean org.jboss.naming.JNDIView

Die MBean `JNDIView` gestattet dem Benutzer, die JNDI-Namensraum-Struktur, wie sie im JBoss-Server vorhanden ist, mit Hilfe des Interface für die JMX-Agentenansicht zu betrachten. Zu diesem Zweck stellen Sie unter Verwendung des HTTP-Interface eine

Verbindung zu der JMX-Agentenansicht her. In der Standardeinstellung finden Sie diese unter *http://localhost:8080/jmx-console/*. Auf dieser Seite sehen Sie einen Abschnitt, in dem die registrierten MBeans nach Domänen sortiert aufgelistet sind. Er sollte in etwa so aussehen wie die Liste in Abbildung 3.4.

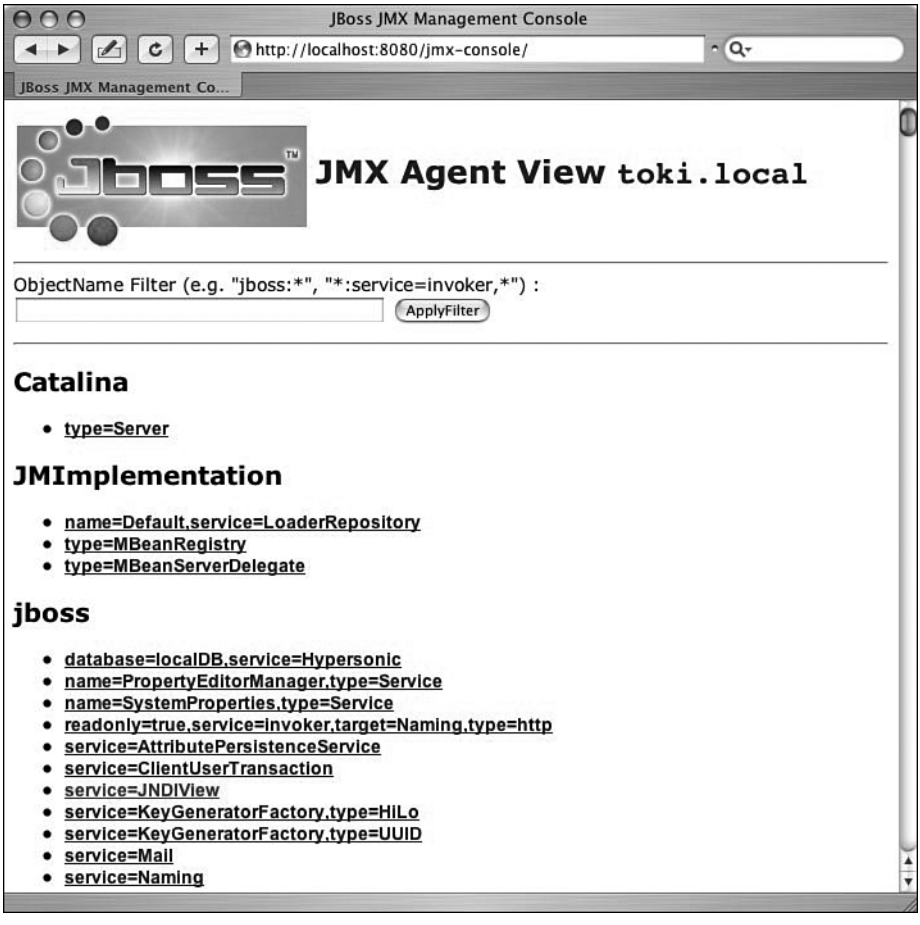

Abbildung 3.4: Die Ansicht der konfigurierten JBoss-MBeans in der JMX-Konsole

Über den Link `JNDIView` gelangen Sie zur Ansicht der MBean `JNDIView`, in der eine Liste von deren Operationen anzeigt wird. Diese Ansicht sollte ähnlich aussehen wie in Abbildung 3.5.

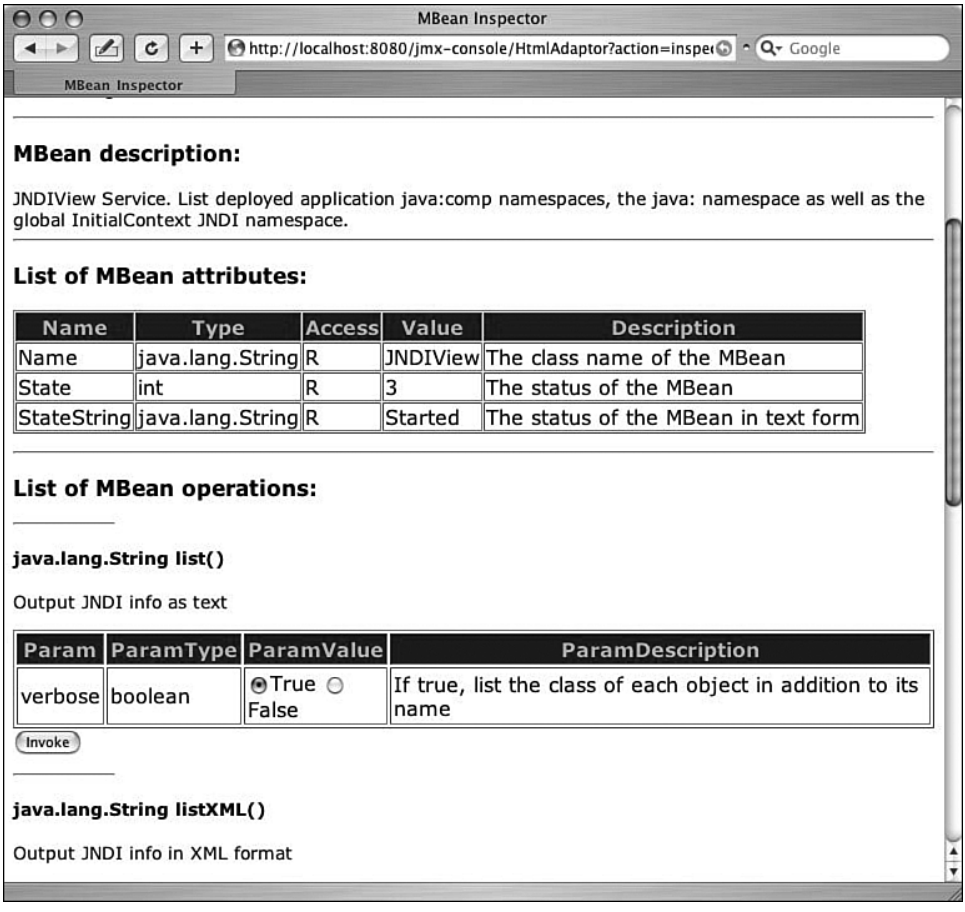

Abbildung 3.5: Die Ansicht der MBean `JNDIView` in der JMX-Konsole

Die Listenoperation lagert den JNDI-Namensraum des JBoss-Servers in eine HTML-Seite aus und verwendet dabei eine einfache Textansicht. Der Aufruf der Listenoperation erzeugt beispielsweise die in Abbildung 3.6 dargestellt Ansicht.

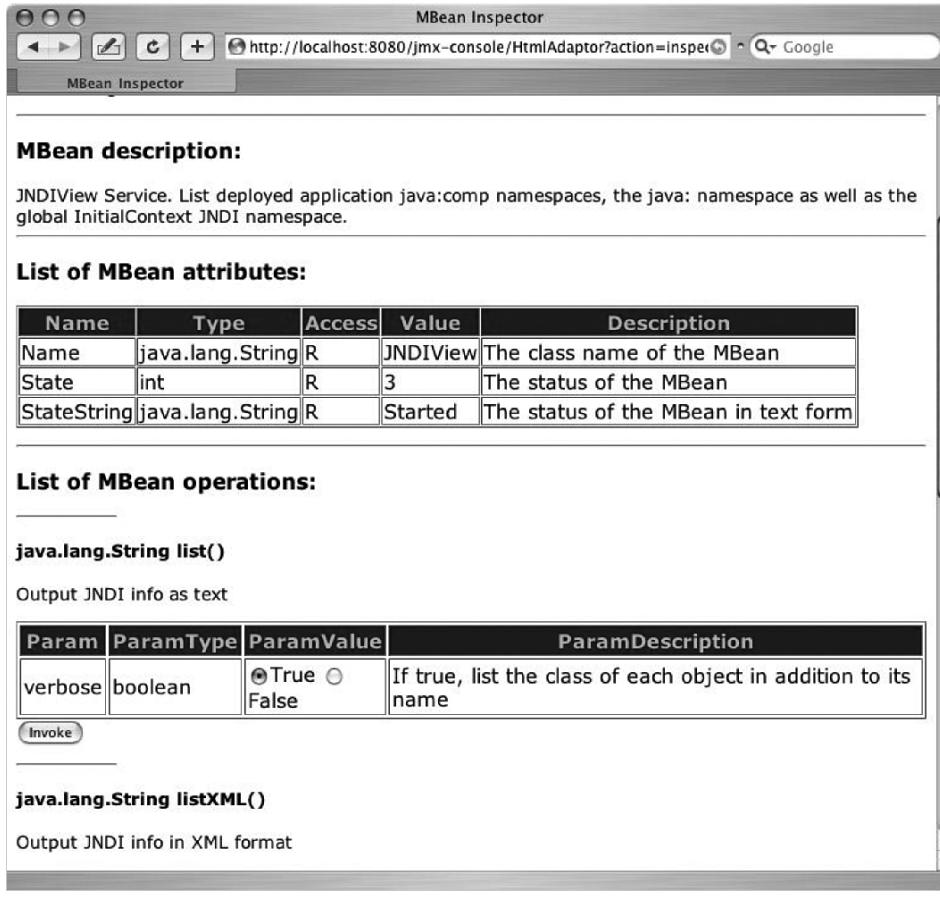

Abbildung 3.6: Die JMX-Konsolenansicht der Ausgabe aus der `JNDIView`-Listenoperation