

## Apache Geronimo, Teil 2: Der Geronimo-basierte *Java Magazin* Application Server im Eigenbau

# Do it yourself

■ VON CHRISTIAN DEDEK UND KRISTIAN KÖHLER

Der eigene Application Server? Geht nicht? Gibt's nicht! – Geronimo macht's möglich. Die Erstellung einer eigenen Serverdistribution scheint im ersten Moment eventuell völlig überflüssig. Doch bei näherem Betrachten macht eine optimierte Ausführungsplattform für die eigene Anwendung Sinn. Wir zeigen die Konfektionierung einer eigenen Geronimo-basierten Serverdistribution. Der vorgestellte *Java Magazin* Application Server zum Selbstübersetzen wird zum Download angeboten [1].

Der erste Teil dieser Artikelreihe betrachtete den grundlegenden Aufbau des Geronimo Application Server. Neben der Administration und Installation des Servers wurde seine Verwendung als Java EE-konformer Application Server vorgestellt. Ein Schwerpunkt lag im Deployment von Java EE-Anwendungsarchiven sowie der grundsätzlichen Architektur des Servers.

Der Geronimo arbeitet wie gezeigt konfigurationsbasiert. Geronimo-Serverbestandteile werden in gleicher Weise wie eigene Anwendungen als Konfigurationen innerhalb des Servers verwaltet. Über die im Server enthaltenen Konfigurationen wird dessen Funktionsumfang bestimmt. Die Erstellung einer Serverdistribution beschränkt sich auf die Zusammenstellung von Konfigurationen, über die der vollständige Funktionsumfang definiert wird. Der Aufwand zur Herstellung einer auf Geronimo basierenden Distribution wird durch die zur Verfügung gestellten Werkzeuge sowie die zugrunde liegende Apache Software Licence gering gehalten und stellt ein reizvolles Einsatzgebiet für den Geronimo dar. In der Versionsverwaltung der Apache Software Foundation stehen bereits verschiedene Serverdistribuitionen zur Verfügung:

- Java EE Jetty – Java EE-konformer Server mit einem vorkonfigurierten Jetty-Web-Container

- Java EE Tomcat – Java EE-konformer Server mit einem vorkonfigurierten Tomcat-Web-Container
- Minimal Tomcat – Distribution mit reiner Web-Container-Unterstützung
- Web JMS Tomcat Server – Distribution mit Web-Container und JMS-Unterstützung
- Java EE Installer – selbstextrahierendes Installer-Archiv, mit dem ein Java EE-konformer Server installiert werden kann. Bei der Installation kann der Funktionsumfang des Servers angegeben werden.

Wie im ersten Teil der Reihe bereits erwähnt, nutzen einige Firmen den Vorzug einer eigenen Serverdistribution für interessante kommerzielle Szenarien. Das prominenteste Beispiel ist sicherlich IBM, die mit der WebSphere Application Server Community Edition [2] einen auf Geronimo basierenden Java EE-konformen Application Server als eigenes Produkt im Marktsegment für Klein- und mittelständische Unternehmen positioniert.

Die Erstellung einer eigenen Distribution bietet sich allerdings nicht nur für Schwergewichte der IT-Branche an. Der Geronimo Application Server kann in seinem Funktionsumfang auf die eigenen Anforderungen zugeschnitten werden. Eigene Anwendungen können über diesen Mechanismus innerhalb einer optimierten Serverumgebung ausgeführt werden. Werden z.B. keine EJBs im Server genutzt, muss die Kon-

figuration für die EJB-Unterstützung nicht in die Distribution aufgenommen werden.

Eine auf Geronimo basierte Produkt-distribution kann sich aus einer optimierten Serverdistribution sowie darin installierten Anwendungen zusammensetzen. Beispiele für optimierte Serverdistribuitionen wären:

- reiner Java EE-Web-Container mit prozesseigener Datenbank
- EAI-Server aus JMS-Implementierung, Datenbank und Routing Engine
- Web Service Engine zur generischen Bereitstellung vorhandener Dienste innerhalb einer serviceorientierten Architektur
- Server mit Funktionalität für Spring Deployment

### Apache Geronimo – die Serie

Teil 1: Einsatz und Aufbau

Teil 2: Erstellen einer eigenen Geronimo-Distribution und Servererweiterungen

Teil 3: Ein eigenes Anwendungsdeployment (voraussichtlich *Java Magazin* 8.2006)

### Lesetipp



Bitte beachten Sie auch das im Mai 2006 im Software & Support Verlag frisch erschienene Buch der beiden Artikel-Autoren: **Geronimo – Apache Geronimo im Einsatz.**

- Portalserver aus JSR 168-konformem Portlet-Container und Content Management System

Die in die jeweiligen Server installierbaren Anwendungen könnten sowohl standardkonforme Java EE-Archive

**Zur Registrierung und konkreten Konfiguration der GBeans verwendet der Geronimo denselben Mechanismus wie beim Deployment von Java EE-Standardkomponenten.**

(EAR, WAR, RAR ...) oder auch eigene Anwendungsformate (verschlüsseltes ZIP ...) sein.

## Der Bauplan

Die Konfektionierung einer Distribution funktioniert nach dem Baukastenprinzip. Dadurch ist es möglich, aus einfachen Bausteinen komplexe Strukturen aufzubauen. Im ersten Schritt werden die einfachen Bausteine gefertigt. In diesem Bild entspricht ein Bau-

## ConfigStore

Konfigurationen werden im Geronimo-Kernel durch Objekte vom Typ *Configuration Store*, hier kurz ConfigStore, verwaltet. Der ConfigStore bietet das Installieren, Laden, Suchen und Deinstallieren der Konfigurationen, die über ihre eindeutige *configId* adressiert werden, an. Die Daten einer Konfiguration werden zwischen ConfigStore und Geronimo-Kernel in Form von *ConfigurationData*-Objekten ausgetauscht. Mit der CAR-Datei ist ein spezielles Archivformat zur dauerhaften Speicherung der Konfigurationen definiert. Ein CAR-Archiv enthält dabei jeweils genau eine Konfiguration. Ein ConfigStore unterscheidet dabei nicht, ob das CAR-Archiv Java EE-Komponenten oder eigene Geronimo-Services enthält. Die Standardimplementierung des ConfigStore nutzt das lokale Dateisystem zur Ablage sämtlicher Konfigurationsdaten. Die Konfigurationen werden in separaten Unterverzeichnissen des Verzeichnisses *config-store* abgespeichert. Der Inhalt dieser Verzeichnisse entspricht dem Inhalt eines entpackten CAR-Archivs.

stein einer Geronimo-Konfiguration. Benötigte Dienste und Funktionen der optimierten Serverumgebung sowie die zu integrierenden Anwendungen müssen hierzu als Konfigurationen erstellt werden. Größe und Umfang eines Bausteins sind vom Entwickler hierbei selbst zu bestimmen. Indem man z.B. eine Datenbank-Anbindung in Form einer eigenständigen Konfiguration definiert, erhält man eine kleine, leicht wiederverwendbare Einheit. Wird die gleiche Datenbank-Anbindung in eine komplexere Konfiguration integriert, ergibt sich ein größerer Baustein, der sich durch einen größeren Funktionsumfang auszeichnen kann.

Im zweiten Schritt der Konfektionierung werden die Bausteine zur endgültigen Produktdistribution zusammengestellt. Art und Herkunft der Bausteine spielen für den Prozess des Zusammensetzens keine Rolle.

Im folgenden Abschnitt wird die Herstellung eigener Bausteine demonstriert. Zuerst wird ein eigener Dienst für eine optimierte Serverumgebung vorgestellt. Als zweiter Baustein dient eine Java EE-konforme Anwendung. Beide Bausteine werden im Anschluss zusammen mit weiteren Konfigurationen zu einem eigenen

Geronimo-basierten Produkt zusammengestellt.

## Der eigene Baustein

Im ersten Artikel wurde die Basisarchitektur des Geronimo behandelt. Grundlegendes Element bilden dabei die GBeans. Sie sind die kleinste verwaltbare Einheit innerhalb des Servers. Ein eigener Dienst wird deshalb auch durch mindestens eine GBean in den Server integriert. Aufgabe der GBean ist die Definition und Bereitstellung der administrativen und konfigurativen Schnittstelle zwischen dem Dienst und dem Application Server. Der Geronimo-Kernel unterscheidet dabei nicht zwischen Java EE-Application-Komponente oder einem proprietären Dienst. Die GBeans einer Konfiguration werden innerhalb des Servers im ConfigStore gespeichert. Er stellt in unserer Metapher den Baukasten zur Aufbewahrung der Bausteine dar (siehe Kasten ConfigStore).

Die eigentliche Implementierung eines eigenen Dienstes kann ohne spezielle Berücksichtigung des Application Server erfolgen. Die Integration wird dann nachträglich durch die Bereitstellung einer oder mehrerer GBeans durchgeführt. An dieser Stelle setzt unser Beispiel der Entwicklung und Integration

## Listing 1

### GBean Beispiel

```
public class BeispielGBean implements GBeanLifecycle {
    private String serviceName;

    public String getServiceName() {
        return serviceName;
    }

    public void setServiceName(String serviceName) {
        this.name = serviceName;
    }

    public static final GBeanInfo GBEAN_INFO;

    static {
        GBeanInfoBuilder builder = new GBeanInfoBuilder
            (BeispielGBean.class);
        builder.addAttribute("serviceName",
            String.class, true);
        GBEAN_INFO = builder.getBeanInfo();
    }

    public static GBeanInfo getGBeanInfo() {
        return GBEAN_INFO;
    }

    public void doStart() throws Exception {
        System.out.println("doStart");
    }

    public void doStop() throws Exception {
        System.out.println("doStop");
    }

    public void doFail() {
    }
}
```

einer GBean an. Die Verwaltungsschnittstelle zwischen Geronimo und Dienst ist möglichst simpel durch ein String-Feld *serviceName* repräsentiert. Die einzige zu erfüllende Aufgabe der GBean ist die Implementierung der Methode *getGBeanInfo()*, die ein *GBeanInfo*-Objekt zur Beschreibung der Schnittstelle zurückgibt. Statt das Metadaten-Objekt selbst zu erzeugen, kann mit der Klasse *GBeanInfoBuilder* eine Implementierung des Builder Patterns[3] zur Vereinfachung der Objektinitialisierung eingesetzt werden. Die GBean des Beispiels nutzt zusätzlich noch die Möglichkeit, einen eigenen Dienst mit Signalen aus dem Lebenszyklusmanagement des Application Server zu versorgen. Dazu wird das Interface *GBeanLifeCycle* mit den Methoden *doStart()*, *doStop()* und *doFail()* implementiert (Listing 1).

Zur Registrierung und konkreten Konfiguration der GBeans verwendet der Geronimo denselben Mechanismus wie beim Deployment von Java EE-Standardkomponenten. Die an den JSR 88 angelehnten Deployment-Pläne beschreiben in Form einer XML-Datei die Umgebungseinstellungen aller GBeans einer Konfiguration. Für unser Beispiel muss also ein Deployment-Plan angelegt werden,

## Listing 2

### Deployment-Plan des GBean-Beispiels

```
<?xml version="1.0"?>

<configuration xmlns="http://geronimo.apache.org/
  xml/ns/deployment-1.0"
  configId="de/oio/gbean-sample/1.0">

  <dependency>
    <groupId>de.oio.geronimo</groupId>
    <artifactId>gBeanBeispiel</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>

  <gbean name="OIOSample" class="de.oio.geronimo.
    SampleGBean">
    <attribute name="serviceName">Kristian</
      attribute>
  </gbean>
</configuration>
```

welcher die GBean und ihre Anbindung an den Server beschreibt (Listing 2).

Eine Konfiguration wird durch ein Element *configuration* beschrieben. Dieses muss mindestens einen innerhalb der Serverumgebung eindeutigen Wert für das Attribut *configId* zur Identifikation der Konfiguration enthalten. Eine GBean wird durch das Element *gbean* definiert. Durch die Attribute *name* und *class* werden ein eindeutiger Name innerhalb der Konfiguration sowie die Implementierungsklasse der GBean bestimmt. Über ein *attribute*-Element kann ein Attribut der Managementschnittstelle der GBean durch Dependency Injection initialisiert werden. Die Implementierungsklasse der

## Repository

Der Geronimo verwaltet innerhalb eines Repository sämtliche vom Server benötigten Bibliotheken. Durch das Repository können Konfigurationen Bibliotheken gemeinsam nutzen. Diese müssen nicht in jeder Konfiguration redundant gehalten werden. Allerdings besitzen Konfigurationen dadurch eine Laufzeitabhängigkeit und können nur in einem Server mit passendem Repository ausgeführt werden. Die Struktur ist an Maven angelehnt. Ressourcen im Repository werden durch einen eindeutigen Identifier aus vier Bestandteilen referenziert.

- *groupId*: Kurzname der zugehörigen Gruppe. Gruppen dienen zur Zusammenfassung mehrerer jar-Archive eines Herstellers.
- *type*: gibt den Typ der Datei als Dateiendung an. Im Falle von Bibliotheken handelt es sich hierbei um JAR.
- *artifactId*: Kurzname der Datei. Innerhalb einer Gruppe werden einzelne Bibliotheken über die *artifactId* unterschieden.
- *version*: Angabe zur Version der Datei.

Die Standard-Repository-Implementierung legt die Dateien unterhalb des Repository-Verzeichnisses ab. Der entsprechende Dateiname setzt sich hierbei wie folgt zusammen: *<groupId>/<type>s/<artifactId>-<version>.<type>*. Eine Referenz auf eine Datei *de.geronimo/jars/gBeanBeispiel-1.0-SNAPSHOT.jar* unterhalb des *Repository*-Verzeichnisses einer Geronimo-Installation hat innerhalb eines Deployment-Plans dann folgende Form:

```
<groupId>de.oio.geronimo</groupId>
<artifactId>gBeanBeispiel</artifactId>
<version>1.0-SNAPSHOT</version>
```

Anzeige

GBean und die damit verbundenen Klassen des Dienstes werden dem Geronimo-Server über ein Repository zur Verfügung gestellt. Durch *dependency*-Elemente werden im Deployment-Plan Verweise auf die Archive, welche diese Klassen enthalten, eingerichtet (siehe Kasten Repository).

Zur Fertigstellung des ersten Bausteins muss noch das CAR-Archiv angelegt werden. Zu diesem Zweck kann das

### Das Zusammensetzen der einzelnen Bausteine kann mittels des Maven Geronimo Assembly Plug-in erfolgen.

Maven-basierte Packaging-Plug-in genutzt werden. In der Geronimo-Version 1.0 handelt es sich um ein Plug-in für Maven 1, welches in der Version 1.1 durch ein Plug-in für Maven 2 erweitert wird. Eine alternative Werkzeugunterstützung durch Ant ist in Planung.

Ein Deployment zur Laufzeit wäre eine grundsätzliche Alternative bei der Erstellung der CAR-Archive. Allerdings muss dafür eine bereits existierende Geronimo-Distribution gestartet werden. Daher wird hier nicht weiter auf diese Möglichkeit eingegangen.

Befindet sich die GBean-Implementierung in einem Maven 1-basierten Projekt, kann durch Aufnahme des folgenden XML-Ausschnitts aus Listing 3 das Packaging-Plug-in nach der Übersetzung des Projektes aufgerufen werden. Die resultierende CAR-Datei wird automatisch in das Maven Repository kopiert.

```
<project default="default">
  <goal name="default">
    <attainGoal name="car:install"/>
  </goal>
</project>
```

Das Plug-in greift auf den bereits aus dem ersten Artikel bekannten ConfigurationBuilder des Geronimo-Kerns zurück. Dazu wird durch das Plug-in ein minimaler Server gestartet, dem der Deployment-Plan übergeben wird. Der für den

Deployment-Plan zuständige Builder erzeugt ein *ConfigurationData*-Objekt, welches als *config.ser* im CAR-Archiv abgelegt wird.

Der eigene Dienst in Form des CAR-Archivs ist der erste fertig gestellte Baustein und kann als Bestandteil einer optimierten Serverumgebung in späteren Distributionen verwendet werden. Sämtliche Geronimo-Bestandteile werden über diesen Mechanismus erstellt.

### Der Java EE-Baustein

Beim Deployment eines Java EE-Archivs wird ebenfalls eine Konfiguration angelegt. Diese wird auch durch den ConfigStore in Form einer CAR-Datei abgelegt. Im Bereich des Standard-Deployments waren wir nicht näher auf dieses Serverspezifikum eingegangen. Der Deployment-Plan unterscheidet sich vom Deployment-Plan des vorherigen Bausteins durch ein anderes Wurzelement *application*. Innerhalb der Deployment-Pläne können gleiche Umgebungskonfigurationen von GBeans durch Elemente der gleichen Namensräume identisch deklariert werden. Auch hier unterscheidet der Server nicht zwischen GBeans aus Java EE-Archiven und speziellen Geronimo-Services. Der Aufbau des Plans ist an einen Java EE-Standard-Deployment-Deskriptor angelehnt. Mithilfe des Attributs *configId* wird wieder ein Name zur eindeutigen Identifikation des späteren Bausteins definiert. Im Beispiel handelt es sich um ein EAR-Archiv *ear-sample-1.0-SNAPSHOT.ear*, in welchem eine Webapplikation als *web-sample-1.0-SNAPSHOT.war* abgelegt ist (Listing 3).

Das Maven-Plug-in kann ebenfalls zur Herstellung der CAR-Archivs aus einem Java EE-Standardarchiv verwendet werden. Hierzu kann ein eigenständiges Maven-Projekt, mit dessen Hilfe die Konfiguration erstellt wird, angelegt werden. Die darin enthaltene Datei *project.xml* muss, wie in der folgenden Quellcodepassage gezeigt, um die Abhängigkeiten zu den entsprechenden Java EE-Archiven erweitert werden. Die *maven.xml*-Datei des Projektes muss wieder die Packaging-Plug-in-Angabe enthalten. Nach Aufruf des Maven-Systems

liegt auch hier die CAR-Datei im Maven Repository ab.

```
<dependency>
  <groupId>de.oio.geronimo</groupId>
  <artifactId>ear-sample</artifactId>
  <version>1.0-SNAPSHOT</version>
  <type>ear</type>
</dependency>
```

Den gravierendsten Unterschied zur Erzeugung des CAR-Archivs eines Serverdienstes bildet der zuständige Builder des Serverkerns. Für Java EE-Archive ist nicht der ConfigurationBuilder, sondern ein ModuleBuilder zuständig. Er wertet den Deployment-Plan aus und erstellt das entsprechende *ConfigurationData*-Objekt. Die Auswahl des korrekten Builder übernimmt der Geronimo-Kernel und ist für den Nutzer transparent. Der ModuleBuilder wird dabei nur zur Auswertung des Deployment-Plans benötigt. Eine bereits erstellte Konfiguration innerhalb des ConfigStore benötigt zu ihrer Benutzung keinen ModuleBuilder. Durch Entfernen des ModuleBuilder entsteht eine Distribution ohne Deployment-Möglichkeit für Java EE-Archive. Eine Java EE-Anwendung in Form eines CAR-Archivs stellt eine bereits gültige Konfiguration dar. Ein solcher Baustein ist auch in einer Distribution ohne Java EE Deployment einsetzbar.

### Stein auf Stein ...

Das Zusammensetzen der einzelnen Bausteine kann mittels des Maven Geronimo Assembly Plug-in erfolgen. Das

#### Listing 3

##### Deployment-Plan für ein EAR-Archiv

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://geronimo.apache.org/xml/ns/j2ee/application-1.0"
  configId="de.oio.geronimo/application/1.0/car">
  <module>
    <web>web-sample-1.0-SNAPSHOT.war</web>
    <web-app xmlns="http://geronimo.apache.org/xml/ns/j2ee/web/jetty-1.0" configId="Web">
      </web-app>
    </module>
  </application>
```

Plug-in wird ebenfalls innerhalb des Geronimo-Projektes zur Konfektionierung der angebotenen Serverdistributionen eingesetzt.

Jede Serverdistribution sollte als separates Maven-Projekt angelegt werden. In der entsprechenden *project.xml*-Datei müssen sämtliche benötigten Bausteine als Abhängigkeiten eingetragen werden. Dies erstreckt sich vom Geronimo-Kernel bis zu den Start und Stopp-Skripten des Servers. Eigene Anwendungen müssen in diesem Zusammenhang ebenfalls angegeben werden. Einen Ausschnitt der Datei zeigt der folgende Quellcode. In diesem Beispiel wird Jetty als Konfiguration mit in den Server aufgenommen. Die CAR-Datei aus dem Maven Repository wird durch das Assembly-Plug-in in den ConfigStore installiert und steht somit im Server zur Verfügung.

```
<dependency>
<groupId>geronimo</groupId>
<artifactId>jetty</artifactId>
<type>car</type>
<version>1.2-SNAPSHOT</version>
<properties>
<geronimo.assemble>install</geronimo.assemble>
</properties>
</dependency>
...
```

Alle eingebundenen Konfigurationen müssen als CAR-Archiv zur Verfügung stehen. Diese Abhängigkeiten können entweder durch das lokale Maven Repository oder über ein entferntes Maven Repository aufgelöst werden. Die Konfigurationen für Geronimo 1.0 findet man z.B. auch online unter [www.ibiblio.org/maven/geronimo/cars/](http://www.ibiblio.org/maven/geronimo/cars/). Dies ermöglicht eine Assemblierung eines Geronimo-Servers ohne eigene Kompilierung der Geronimo-Quellen.

Unterhalb des *src*-Verzeichnisses können weitere Dateien, die mit in die Distribution kopiert werden, hinterlegt werden. Unter *var/configs* sollte sich die *config.xml*-Datei befinden. Diese Datei wird, wie im letzten Artikel angesprochen, zum Startzeitpunkt des Servers ausgewertet und die zu startenden Konfigurationen daraus ermittelt. Sie sollte bei einer eigenen Distribution ebenfalls die zu startenden Konfigurationen enthalten.

Abb. 1:  
Serverkonsole

```
Booting Geronimo Kernel (in Java 1.4.2_07)...
Started configuration 1/3 1s: geronimo/jetty/1.2-SNAPSHOT/car
Started configuration 2/3 1s: de.ocio/geronimo/application/1.0/car
Started configuration 3/3 0s: geronimo/j2ee-security/1.2-SNAPSHOT/car
Startup completed in 4 seconds
Listening on Ports:
 0 127.0.0.1 JMX Remoting Connector
1099 0.0.0.0 RMI Naming
4242 0.0.0.0 Remote Login Listener
8019 127.0.0.1 Jetty Connector AJP13
8080 0.0.0.0 Jetty Connector HTTP
8443 0.0.0.0 Jetty Connector HTTPS

Started Application Modules:
EAR: de.ocio/geronimo/application/1.0/car

Web Applications:
http://LAP0023:8080/web-sample

Geronimo Application Server started
```

Die Einbindung des Maven-Plug-ins in das Projekt kann über die Aufnahme des folgenden XML-Blocks in die *maven.xml*-Datei erfolgen.

```
<project default="default">
<goal name="default" prereqs="assemble:install"/>
</project>
```

Nach dem erfolgreichen Maven-Aufruf findet man im *target*-Verzeichnis des Projektes die eigene Serverdistribution als *TAR*, *GZ*- und *ZIP*-Archiv. Der

Name der Distribution wird über das Maven Property *geronimo.assembly.name* definiert. Die Abbildung 1 zeigt die Konsolenausgabe des angebotenen assemblierten *Java Magazin*-Servers nach dem erfolgreichen Serverstart über das *startup*-Skript.

### Der Meisterbrief

In der bisher beschriebenen Produktdistribution werden Java EE-Anwendungsarchive in der Standardimplementierung des ConfigStore abgelegt. Verschiedene

## Anzeige

Anforderungen an die Softwareverteilung eines Produktes lassen sich mit diesem Standardmechanismus nicht abbilden. Die Architektur des Servers erlaubt allerdings den Austausch der

Den Geronimo-basierten *Java Magazin* Application Server finden Sie unter [www.javamagazin.de/geronimo](http://www.javamagazin.de/geronimo)

ConfigStore-Implementierung zur Realisierung unterschiedlichster Distributionsszenarien. Folgende Szenarien sind z.B. denkbar:

- zentraler Distributionsserver
- zentraler Lizenzierungsmechanismus auf Basis von Anwendungsmodulen
- Farming von geclusterten Applikationen

### ObjectNameGroup

Über eine *ObjectNameGroup* kann eine im Geronimo installierte GBean eindeutig referenziert werden. Die Gruppe setzt sich aus mehreren Bestandteilen zusammen, deren einzelne Namen sich an die Java EE-Management-Spezifikation (JSR 77) anlehnen.

- Domain
- Server
- Application
- Module
- Type
- Name

Die *ObjectNameGroup* wird wiederum auf einen JMX ObjectName abgebildet, unter dem die GBean am JMX MBeanServer angemeldet wird. Der Name der *ObjectNameGroup* Domain wird hierbei auf die JMX ObjectName Domain abgebildet. Die weiteren Namensbestandteile werden als „Key Property List“ des JMX ObjectName angegeben.

Beispiel eines resultierenden JMX ObjectName:

```
geronimo.server:EJBModule=daytrader-ejb-1.2-SNAPSHOT.jar,J2EEApplication=geronimo/daytrader-derby-tomcat/1.2-SNAPSHOT/car,J2EEServer=geronimo,j2eeType=EntityBean,name=AccountEJB
```

- Verschlüsselung der Anwendungsarchive

### Eigener Baukasten

Neben den normalen Anforderungen an eine GBean muss der eigene ConfigStore das Interface *org.apache.geronimo.kernel.config.ConfigurationStore* implementieren.

Das Interface enthält Methoden zum Installieren, Laden, Suchen und Deinstallieren von Konfigurationen. Diese werden hierbei über ihre eindeutige *configId* adressiert und in Form von *ConfigurationData*-Objekten übergeben. Für die Ablage der Konfiguration können *ConfigurationData*-Objekte über Geronimo-Hilfsklassen recht einfach serialisiert werden.

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
GBeanData configurationGBeanData = ExecutableConfigurationUtil.getConfigurationGBeanData(configurationData);
configurationGBeanData.writeExternal(oos);
```

```
byte[] bites = baos.toByteArray();
//Speicherung mit eventueller Verschlüsselung
```

In den Methoden zum Laden der Konfiguration könnten Lizenzprüfungen eingebaut werden, die prüfen, ob eine Konfiguration überhaupt über diesen

ConfigStore geladen werden kann oder nicht.

Da es sich bei der ConfigStore-Implementierung um eine GBean handelt, muss die Methode *getGBeanInfo* erstellt werden. Dies kann wie im obigen Beispiel mithilfe der Klasse *GBeanInfoBuilder* erfolgen.

```
GBeanInfoBuilder infoFactory =
    GBeanInfoBuilder.createStatic(AlternativeConfigStore
        Impl.class, "ConfigurationStore");
```

Wichtigster Wert in diesem Beispiel ist der zweite Parameter im Aufruf der Methode *createStatic*. Es handelt sich hierbei um die Angabe des Java EE-Typs (*ConfigurationStore*), die sich im eindeutigen Namen (siehe Kasten *ObjectNameGroup*) der GBean, unter der es im Server angemeldet wird, widerspiegelt.

Der Dependency-Injection-Mechanismus für GBeans innerhalb des Geronimo unterstützt neben einer festen Referenzangabe zu einer einzelnen GBean auch die Definition einer Referenz zu einer Menge von GBeans. Über diesen Mechanismus kann eine GBean Interesse an weiteren GBeans anmelden. Wird eine entsprechende GBean gestartet bzw. wieder gestoppt, wird eine Referenz über Dependency Injection gesetzt. Die Angabe der Ziel-Bean erfolgt über ein Namensmuster, das Teile der *ObjectNameGroup* enthalten kann. Die Angabe von *\*:j2eeType=WebModule,\** stellt z.B.

### Listing 4

#### Deployment-Plan-Ausschnitt für ConfigurationManager

```
<!-- Configuration Manager service -->
<gbean name="ConfigurationManager"
    class="org.apache.geronimo.kernel.config.
        EditableConfigurationManagerImpl">
    <reference name="Stores">
        <gbean-name>*:j2eeType=ConfigurationStore,*</gbean-name>
    </reference>
    <reference name="AttributeStore">
        <name>AttributeManager</name>
    </reference>
    <reference name="PersistentConfigurationList">
        <type>AttributeStore</type>
        <name>AttributeManager</name>
    </reference>
</gbean>
```

### Listing 5

#### Deployment-Plan für eigenen ConfigStore

```
<configuration
    xmlns="http://geronimo.apache.org/xml/ns/
        deployment-1.0">
    configId="de.oio.geronimo/alternative-config-
        Store-config/1.0-SNAPSHOT/car"
    domain="geronimo.server"
    server="geronimo">
    <gbean name="AlternativeConfigStore"
        class="de.oio.geronimo.config.AlternativeConfig-
            StoreImpl">
    </gbean>
</configuration>
```

eine Referenzangabe zu allen GBeans mit einem *j2eeType*-Webmodul dar. Auf diese Weise kann sich der Web-Container über neu installierte Webmodule informieren lassen und entsprechend darauf reagieren. Eine direkte Referenz zwischen Installationsmechanismus und Ausführungsumgebung ist nicht nötig.

In der Standardkonfiguration des Geronimo verwaltet der *ConfigurationManager* auf diese Weise alle *ConfigStore*-Instanzen. Er lässt sich über den Start bzw. das Stoppen aller GBeans, die den Typ *ConfigurationStore* besitzen, informieren (Listing 4).

Durch die oben gezeigte Java EE-Typ-Angabe beim Erstellen des *GBeanData*-Objektes innerhalb der eigenen *ConfigStore*-Implementierung besitzt die GBean den Typ *ConfigurationStore*. Beim Start der Bean wird der *ConfigurationManager* entsprechend darüber informiert. Eine weitere Anmeldung des *ConfigStore* ist nicht erforderlich.

## Richtfest

Damit die eigene *ConfigStore* GBean im Server zum Einsatz kommen kann, muss eine eigene Konfiguration für den Store erstellt werden. Hierzu kann wieder das Geronimo Maven 1 Packaging Plug-in genutzt werden. Der Deployment-Plan ist in Listing 5 dargestellt. Die

Konfiguration kann anschließend über ein eigenes Assemblieren des Servers genutzt werden. Eine Anpassung der bestehenden Serverkonfigurationen ist nicht nötig.

## Der Einzug

Innerhalb des Geronimo werden alle *ConfigStores* als Targets verwaltet. Dieser Begriff stammt aus der Java EE-Application-Deployment-Spezifikation und bezeichnet zur Verfügung stehende Installationsziele. Mit dem Geronimo-eigenen Kommandozeilenwerkzeug *deploy* lassen sich diese Targets eines Servers ausgeben. Die Ausführung von *deploy.bat list-targets* liefert im Beispiel:

Available Targets:

```
geronimo.server:J2EEApplication=null,J2EEModule=
geronimo/j2ee-system/1.2-SNAPSHOT/car,J2EEServer=
geronimo,j2eeType=ConfigurationStore,
name=Local
geronimo.server:J2EEApplication=null,J2EEModule=
de.oio.geronimo/alternative-configStore-config/
1.0-SNAPSHOT/ar,J2EEServer=geronimo,j2eeType=
ConfigurationStore,name=AlternativeConfigStore
```

Beim zweiten Eintrag handelt es sich um den eigenen *ConfigStore* der unter der ConfigId *de.oio.geronimo/alternative-configStore-config/1.0-SNAPSHOT/ar* angemeldet wurde. Eine Ausgabe aller in

diesem Target installierten Module lässt sich mit folgendem Aufruf erzielen:

```
deploy.bat list-modules geronimo.server:J2EEApplication=
null,J2EEModule=de.oio.geronimo/alternative-config-
Store-config/1.0-SNAPSHOT/car,J2EEServer=geronimo,
j2eeType=ConfigurationStore,name=
AlternativeConfigStore
```

## Nach dem Bau ist vor dem Bau

Die integrationsorientierte Softwarearchitektur des Geronimo erleichtert die Erstellung einer eigenen Serverdistribution. Der unter [www.javamagazin.de/geronimo](http://www.javamagazin.de/geronimo) bereitgestellte *Java Magazin* Application Server kann als Basis für ein eigenes Produkt eingesetzt werden. Viel Spaß beim „Heimwerken“!



**Christian Dedek** und **Kristian Köhler** beschäftigen sich bei Orientation in Objects mit der Architektur, Entwicklung und Beratung von Java EE-Applikationen. Kontakt: [geronimo@oio.de](mailto:geronimo@oio.de).

## Links & Literatur

- [1] [www.oio.de/public/geronimo/jm-artikel](http://www.oio.de/public/geronimo/jm-artikel)
- [2] [www-306.ibm.com/software/de/websphere/was-ce.html](http://www-306.ibm.com/software/de/websphere/was-ce.html)
- [3] Erich Gamma et al.: Entwurfsmuster, Addison-Wesley, 1996, 119 ff.
- [4] Christian Köhler, Christian Dedek: "Ich warte auf die Wende". Apache Geronimo, Teil 1, in *Java Magazin* 5.2006

## Anzeige