

## Apache Geronimo, Teil 3: Der Geronimo-basierte Spring Application Server

# Server à la Spring

■ VON KRISTIAN KÖHLER UND CHRISTIAN DEDEK

Ein applikationsspezifisches Anwendungsarchivformat verwenden? Geht nicht? Gibt's nicht! – Geronimo macht's möglich. Die Erstellung eines eigenen Anwendungsarchivs scheint im ersten Moment vielleicht uninteressant, andererseits stellt sich die Frage des Deployment-Formats, falls eine Anwendung, die nicht auf die Servlet- oder gar EJB- Spezifikation aufsetzt, in einen Application Server installiert werden soll. Beispiele für dieses Szenario sind eine Standalone-Spring-Anwendung oder eine von mehreren Web-Anwendungen gleichzeitig nutzbare Instanz einer Spring-Anwendung.

In ihrem Buch „J2EE without EJB“ [1] zeigten Rod Johnson und Jürgen Holler die Schwachpunkte der bis dahin als Standard für Enterprise-Java-Anwendungen geltenden EJB-Technologie auf. Mit dem Spring Framework [2] stellten sie zusätzlich eine Alternative zur Entwicklung von Enterprise-Anwendungen vor, die nicht mehr die Technologie, sondern vielmehr das fachliche Design der Anwendung in den Vordergrund stellt. Das Framework wird inzwischen in sehr vielen Anwendungen eingesetzt und stellt mit den angebotenen Integrationsmöglichkeiten, zum Beispiel Hibernate, eine gute Basis für eigene Enterprise-Anwendungen dar. Für den produktiven Betrieb im Enterprise-Server-Umfeld können Spring-Anwendungen auch in Java EE-Server installiert werden. Dadurch können leicht die Vorteile dieser Umgebungen, wie standardisierte Administration und spezielle Server Features, beispielsweise Clustering, genutzt werden.

Das Deployment der Spring-Anwendungen erfolgt dabei meist innerhalb eines Java EE-Anwendungsarchivs. Für die

Integration in eine Web- oder sogar EJB-Anwendung stehen im Spring Framework komfortable Möglichkeiten zur Verfügung. Falls allerdings die eigene Applikation nicht als Web- oder EJB-Archiv zur Verfügung gestellt werden soll und trotzdem Server-Dienste wie Transaktionsmanager oder Connection Pooling genutzt werden sollen, fällt das Deployment innerhalb eines Application Server jedoch oft schwer. Stellvertretend für diesen Anwendungstyp stehen hier folgende Szenarien:

- ein Integrationsserver, der keine Remote-Schnittstelle besitzt und nur periodisch Aufgaben ausführen soll
- eine von mehreren Webanwendungen gemeinsam genutzte Spring-basierte Business-Logik-Komponenteninstanz ähnlich einem EJB Backend, die über Hibernate Daten in einer Datenbank speichert

Ein für alle Java EE-Server standardisiertes Spring-Anwendungsarchiv, das die Installation innerhalb des Servers analog zu einem WAR-Archiv erleichtern könnte, steht nicht zur Verfügung. Durch eine Erweiterung des Geronimo-eigenen Anwendungsinstallationsmechanismus kann allerdings die Installation eines solchen Archivs ermöglicht werden. Im Folgenden werden wir auf die Geronimo-Installationsarchitektur eingehen und anhand eines Beispiels zeigen, wie ein

solches Archiv aufgebaut und verarbeitet werden könnte.

## Der Mechanismus

Wir hatten bereits in der ersten Folge der Artikelreihe beschrieben, dass Builder-Instanzen die Installation einer Anwendung in den Server übernehmen. Ein entsprechender Builder erzeugt hierbei für ein bestimmtes Anwendungsarchiv die für die Verwaltung benötigten GBeans. Diese sind die kleinsten verwaltbaren Einheiten innerhalb des Geronimo-Servers und werden vom Builder zu einem Modul (in Version 1.0 Konfigurationen) zusammengefasst. Diese Module können anschließend in den Server geladen und angesprochen werden. Nach der Durchführung des Deployments wird vom Serverkern nicht mehr zwischen den ursprünglichen Anwendungstypen unterschieden. Alle installierten Anwendungen lassen sich mittels der angebotenen Managementmöglichkeiten durch die entsprechenden GBeans gleichartig verwalten.

Das Anbieten eines neuen Anwendungsarchivs beschränkt sich demnach

### Apache Geronimo – die Serie

- Teil 1: Einsatz und Aufbau
- Teil 2: Erstellen einer eigenen Geronimo-Distribution und Servererweiterungen
- Teil 3: Der Geronimo-basierte Spring Application Server

### Lesetipp



Bitte beachten Sie auch das im Mai 2006 bei *entwickler.press* frisch erschienene Buch der beiden Artikel-Autoren: **Geronimo – Apache Geronimo im Einsatz**.

auf die Erstellung eines Builders, der das eigene Archivformat erkennt und auswerten kann. Innerhalb des Builders müssen die GBeans für die Verwaltung der Anwendung erzeugt werden. Die Erstellung eines solchen Builders gestaltet sich relativ einfach. Es ist lediglich für eine Implementierung des Interfaces `org.apache.geronimo.deployment.ConfigurationBuilder` Sorge zu tragen. Das Interface verfügt über drei in Listing 1 gezeigte Methoden.

Wird der Server aufgefordert, ein neues Anwendungsarchiv zu laden, werden alle Builder über die Methode `getDeploymentPlan` geprüft, ob sie das übergebene Archiv verarbeiten können. Ist dies der Fall, muss vom jeweiligen Builder ein Deployment-Plan-Objekt zurückgegeben werden. Dies erfolgt in der Regel als eine einfache Objektrepräsentation des XML-Dokuments. Der zurückgelieferte Deployment-Plan wird nicht direkt vom Server ausgewertet. Er dient lediglich als Parameter für weitere Methodenaufrufe an diesem Builder.

### Listing 1

```
public interface ConfigurationBuilder {
    /**
     * Builds a deployment plan specific to this builder
     * from a planFile and/or module if this builder can
     * process it.
     * ...
     */
    Object getDeploymentPlan(File planFile, JarFile
        module, ModuleIDBuilder idBuilder) throws
        DeploymentException;

    /**
     * Checks what configuration URL will be used for the
     * provided module.
     * ...
     */
    Artifact getConfigurationID(Object plan, JarFile
        module, ModuleIDBuilder idBuilder) throws
        IOException, DeploymentException;

    /**
     * Build a configuration from a local file
     * ...
     */
    DeploymentContext buildConfiguration(boolean
        inPlaceDeployment, Artifact configId, Object plan,
        JarFile module, Collection configurationStores,
        ArtifactResolver artifactResolver, Configuration-
        Store targetConfigurationStore) throws IO-
        Exception, DeploymentException;
}
```

Im zweiten Schritt wird der Builder beauftragt, die `ModuleID` des zu installierenden Archivs mitzuteilen. Hierzu wird die Methode `getConfigurationID` aufgerufen. Die Signatur der Methode ist leider momentan nicht aussagekräftig gewählt und vermischt unglücklicherweise die Begriffe `ModuleID` und `ConfigurationID`. Im dritten und letzten Schritt müssen die entsprechenden GBeans erzeugt und innerhalb eines `DeploymentContext`-Objektes zurückgegeben werden.

### Der Archivaufbau

Da der eigene SpringBuilder die Anwendungsarchive erkennen und auswerten können muss, wird ein spezielles Spring-Archivformat benötigt. Das Archiv soll neben allen benötigten Klassen ebenfalls unterhalb des `META-INF`-Verzeichnisses eine Konfigurationsdatei mit Angaben zur Spring-Umgebung enthalten. Listing 2 zeigt ein Beispiel einer solchen Datei.

Die Angaben in der Konfigurationsdatei deklarieren einen eindeutigen Namen, unter dem der Spring-Kontext verwaltet werden soll. Außerdem werden die zu verwendenden Spring-Konfigurationsdateien innerhalb des Archivs angegeben. Sie sollen während des Deployments zu einem gemeinsamen Spring-Kontext zusammgeführt werden.

Beim Aufruf der `getDeploymentPlan`-Methode des Spring-Builders wird im Beispiel die übergebene Datei über XMLBeans [3] eingelesen. Zur Weiterverarbeitung wird an den Serverkern eine Objektrepräsentation zurückgeliefert, die, wie in Listing 3 erkennbar, aus einem

Anzeige

### Listing 2

```
<?xml version="1.0"?>
<spring-config
  xmlns="http://de.oio.geronimo/xml/ns/
    spring-container-1.0"
  xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance">

  <container-name>OIO-Container</container-name>

  <applicationConfigs>
    <applicationConfig>META-INF/spring-beans.
      xml</applicationConfig>
  </applicationConfigs>

</spring-config>
```

speziellen *SpringConfigDocument* erstellt werden kann.

Innerhalb der Methode *getConfigurati- onID* kann eine beliebige, für das Modul jedoch eindeutige *ModuleID* zurückgeliefert werden. Das Format ist hierbei an das von Maven bekannte Namensmuster angelehnt und besteht ebenfalls aus *Group- ID*, *ArtifactID*, Version und Typangabe.

Das Kernstück der Builder-Implementierung stellt die Methode *buildCon- figuration* dar. In dieser müssen die zur

Verwaltung benötigten GBeans erzeugt werden. Im Beispiel bietet es sich an, jedes Spring-Anwendungsarchiv über ein eigenes GBean am Server anzumelden. Ein solches GBean kann Attribute aus der Konfigurationsdatei aufnehmen und stellt einen Verwaltungs-Wrapper zum eigentlich zu erzeugenden gemeinsamen Spring-Kontext dar. Die Anmeldung eines solchen SpringModule GBeans zeigt Listing 4.

### Einbindung in den Server

Der Einbau des eigenen Builders in den Server erfolgt analog zur Aufnahme der eigenen ConfigStore-Implementierung, die im zweiten Teil der Artikelreihe behandelt wurde. Über das Maven 1 Packaging Plug-in und einen Deployment-Plan (Listing 5) wird ein Modul für den Spring-Builder erstellt, das zum Beispiel während einer Server Assembly in den Server installiert werden kann [4].

Der modulare Aufbau des Geronimo Application Server ermöglicht somit eine einfache Integration der neuen Builder-Implementierung. Eine direkte Referenzierung aus anderen Serverkomponenten (Deployer) heraus, die für die Anwendungsinstallation verantwortlich sind, ist nicht nötig. Wie bei der *ConfigStore*-Einbindung aus Teil 2 der Artikelreihe werden dem im Server vorhandenen Deployer

automatisch per Dependency Injection sämtliche Builder-Instanzen übergeben. Der Builder muss lediglich über sein GBeanInfo Object dem Server mitteilen, dass es sich bei ihm um eine Implementierung des Interface ConfigurationBuilder handelt (Listing 6).

### How to use it – deployen

Nachdem eine neue Server-Assembly mit dem zusätzlichen Builder erstellt wurde, können über das Kommandozeilenwerkzeug *DEPLOY* Spring-Anwendungsarchive installiert werden. Folgender Kommandozeilenauftrag erledigt dies:

```
./deploy.sh deploy meinSpringAnwendungsArchiv.jar
meinSpringDeploymentPlan.xml
```

Ab diesem Installationsschritt kann die Spring-Anwendung wie jede andere Anwendung verwaltet werden. In der *config.xml*-Datei befindet sich z.B. ein Moduleintrag, und das Repository des Servers verfügt ebenfalls über ein neues aufgenommenes Archiv. Die Start- und Stopp-Kommandos für das Anwendungsarchiv können ebenfalls eingesetzt werden.

### Weitergehende Überlegungen

Mit dem Dependency-Mechanismus innerhalb von Deployment-Plänen bietet der Geronimo die Möglichkeit, Abhängigkeiten zwischen Serverbestandteilen zu definieren. Ein Spring-Anwendungsarchiv kann z.B. als Dependency in einem Deployment-Plan einer Webanwendung angegeben werden (wie folgt). Auf diese Weise kann die entsprechende Webanwendung auf Ressourcen innerhalb des referenzierten Spring-Archivs zugreifen.

```
...
<dependency>
<groupId>meineSpringGroupId</groupId>
<artifactId>meineSpringArtifactId</artifactId>
<type>spr</type>
</dependency>
...
```

Enthalten mehrere Webanwendungen Verweise auf dieses Spring-Modul, können diese einen gemeinsamen Backend-Spring-Kontext nutzen. Diese einfache Trennung von Deployment-Einheiten bietet interessante Optionen für den Ein-

#### Listing 3

```
public Object getDeploymentPlan(File planFile, JarFile
    module, ModuleIDBuilder idBuilder) throws
    DeploymentException {

    if (planFile == null && module == null) {
        return null;
    }

    try {

        SpringConfigDocument document = SpringConfig-
            Document.Factory.parse(planFile);
        return document.getSpringConfig();

    } catch (Exception e) {
        return null;
    }

}
```

#### Listing 4

```
...
GBeanData springModule = new GbeanData(
    new AbstractName(configId, nameMap),
    SpringModule.getGBeanInfo());

springModule.setAttribute("containerName",
    springConfig.getContainerName());
springModule.setAttribute("applicationConfigs", configs);
springModule.setAttribute("deployedFileName",
    f.getName());

context.addGBean(springModule);
...
```

#### Listing 5

```
<?xml version="1.0"?>
<module xmlns="http://geronimo.apache.org/xml/
    ns/deployment-1.1">

    <gbean name="springBuilder" class="de.oio.
        geronimo.builder.OIOBuilder">
        <reference name="Repositories"/>
    </gbean>

</module>
```

#### Listing 6

```
static {

    GBeanInfoBuilder infoFactory =
        GBeanInfoBuilder.createStatic(OIOBuilder.class,
            NameFactory.CONFIG_BUILDER);

    infoFactory.addAttribute("kernel", Kernel.class, false);
    infoFactory.addAttribute("configurationManager",
        ConfigurationManager.class, false);

    infoFactory.addReference("Repositories", Repository.
        class, "Repository");

    infoFactory.addInterface(ConfigurationBuilder.class);

    infoFactory.setConstructor(
        new String[]{"kernel", "configurationManager",
            "Repositories"});

    GBEAN_INFO = infoFactory.getBeanInfo();
}

public static GBeanInfo getGBeanInfo() {
    return GBEAN_INFO;
}
```

satz in produktiven Szenarien wie z.B. unabhängige Versionierung oder erhöhte ausfallsichere Deployments von Anwendungsbestandteilen. Die Integrationsarchitektur des Geronimo zeigt zum wiederholten Mal Ihre Stärken.

## Geronimo – der neuste Stand

Seit dem letzten Teil der Geronimo-Reihe (JM 6.2006) hat sich rund um den Geronimo einiges getan. Der Artikel referenziert auf Geronimo 1.1. Die wichtigste Änderung dieser Version besteht in der konsistenten Umbenennung von Configuration zu Module innerhalb des gesamten Servers. Die eventuell auftretende Verwirrung durch die Vermischung der Begriffe innerhalb des Artikels ist in der Umbenennung von Configuration (Version 1.0) zu Module (Version 1.1) begründet. Der Begriff Configuration kann innerhalb des Artikels synonym zu Module verwendet werden. Ab der Version 1.1 existiert innerhalb des Geronimo nur noch der Begriff Module. Das führt zu Veränderungen in sämtlichen Werkzeugen und Kommandos innerhalb des Geronimo. Sämtliche Konfigurationen werden nun durch eine Module-ID an Stelle der bisherigen Configuration-ID verwiesen. Auch die Syntax von Deployment-Plänen wurde verändert. Dies führt in der Folge leider zu einem Migrationsbedarf existierender Geronimo-basierter Anwendungen. Dazu sind sämtliche Verwendungen von Configuration-ID zu ändern. Mithilfe des neuen Kommandozeilenwerkzeug *upgrade.jar* sollten große Teile dieser Arbeiten sich automatisieren lassen. Leider ist die vollständige automatische Migration

von Geronimo 1.0-Deployments auf Geronimo 1.1 bisher nicht gelungen.

Eine weitere Veränderung hat sich in einer Umstellung des bisherigen Configuration Store vollzogen. Das Verzeichnis *config-store* wird entfernt. Deployment-Einheiten werden stattdessen im Verzeichnis *repository* abgelegt. Die Konfigurationsdaten von GBeans können dabei innerhalb des CAR-Archivs jetzt auch in einem XML-Format abgelegt werden. Dies erhöht die Administrationsmöglichkeiten installierter Komponenten signifikant.

Weitere Verbesserung hinsichtlich der Administration des Servers stellt das neue Plug-in-Feature dar. Mithilfe der neuen Plug-in-Infrastruktur können ohne Stopp des Servers Anwendungen, Servererweiterungen und Geronimo-basierte Dienste in eine Geronimo-Instanz hinzugefügt werden. Über die spezielle Webseite [www.geronimoplugins.com](http://www.geronimoplugins.com) soll der Austausch und Handel solcher Plug-ins popularisiert werden.

Abschließend wurde die Administrationskonsole des Geronimo mit weiteren Funktionen, wie z.B. einem Memory-Meter, aufgewertet. Besonders interessant ist hier auch die Möglichkeit der direkten Konfiguration eines integrierten Apache httpd Webserver über die Konsole.

Auch die funktionale Weiterentwicklung des Servers schreitet voran. Neben angestrebten Performanceverbesserungen bei Remote-EJB-Zugriffen wurde die Unterstützung von SMTP innerhalb des Servers verbessert. Die Cluster-Implementierungen für Web- und EJB-Container wurden ebenfalls stark weiterentwickelt.

Außerdem werden unter dem Namen Little-G leichtgewichtige Distributionen des Geronimo mit den Webcontainern Jetty bzw. Tomcat angeboten. Diese Distributionen zeichnen sich aufgrund ihrer reduzierten Dienstkonfigurationen durch einen geringen Ressourcenverbrauch aus.

Leider ist die Migration des Build-Management-Systems von Maven 1 auf die Version 2 des Build-Management-Werkzeugs nicht abgeschlossen. Daher gestaltet sich das Assemblieren eigener Geronimo-Distributionen immer noch etwas umständlicher als nach der abgeschlossen Migration erwartbar sein dürfte. Die Migration wird im Rahmen der Anstrengungen zur Erreichung der Java 5-Zertifizierung innerhalb des kommenden Jahres vorangetrieben [5]. Die Kompatibilität des Geronimo mit dem Sun JDK 1.5 ist auf einem guten Weg und bei Verzicht auf einen CORBA-Einsatz für den Betrieb bereits verfügbar. Als Fazit ist festzustellen, dass der Geronimo-Community 2006 keine Zeit zum Ausruhen bleibt.



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

## ■ Links & Literatur

- [1] [www.interface21.com/books/j2eewithoutEJB.html](http://www.interface21.com/books/j2eewithoutEJB.html)
- [2] [www.springframework.org](http://www.springframework.org)
- [3] [xmlbeans.apache.org](http://xmlbeans.apache.org)
- [4] Christian Dedek, Kristian Köhler: Apache Geronimo: Teil 2: Der Geronimo-basierte *Java Magazin* Application Server im Eigenbau: [www.javamagazin.de/itr/online\\_artikel/psecom,id,816,nodeid,11.html](http://www.javamagazin.de/itr/online_artikel/psecom,id,816,nodeid,11.html)
- [5] [opensource.atlassian.com/confluence/oss/display/GERONIMO/Migration+Status](https://opensource.atlassian.com/confluence/oss/display/GERONIMO/Migration+Status)