Kommentar: Wie praxistauglich ist JSF wirklich? ▸74

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 5.2014

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

MongoDB
Big Data schemafrei verwalten ▸56

WildFly 8
Neue Features plus Interview ▸64

JAX 2014
Alle Infos hier im Heft!
▸35

REACTIVE PROGRAMMING

Event-basierte Architekturen und ihre Vorteile › 18

„Die reaktive Zukunft der JVM sieht rosig aus.“ Interview mit Jonas Bonér › 26

4 194586 709801 05

Clojure Testing: Alternatives Testframework Midje ▸14

Design for Diagnosability: Wenn das System sagt, was ihm weh tut ▸44

Softwarearchitektur: Beschränkte Mittel ▸36

Teil 1: Grundlagen und Neuigkeiten

Ein OSGi Container wird erwachsen

NEUE SERIE

Diese Artikelserie beleuchtet die nächste Generation von Apache Karaf, die gerade veröffentlichte Version 3.0. Seit den Vorgängern 2.2.x und 2.3.x hat sich einiges getan, und so wartet das 3.0-Release mit einigen Neuerungen auf.

von Achim Nierbeck

Zuerst gehen wir nochmal zurück auf Los und schauen uns das Umfeld an. Was ist Apache Karaf? Ganz einfach: ein Container für OSGi. Hört sich im ersten Augenblick nicht wirklich spektakulär an. Die Besonderheiten liegen im Detail. Apache Karaf lässt sich am einfachsten mit einem Tomcat bzw. dessen Aufgaben vergleichen. Tomcat ist ein Container für Webapplikationen, Karaf das Pendant für OSGi, d. h. die infrastrukturellen Voraussetzungen, die notwendig sind, um eine OSGi-Anwendung betreiben zu können, sind bereits gegeben. Eine dieser Voraussetzungen kann zum Beispiel das Logging sein: Wie kann meine Anwendung in diesem Container loggen und wie einfach ist es für mich als Benutzer, dieses Loggen zu beeinflussen?

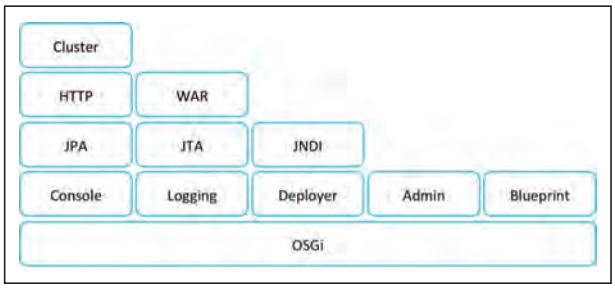

Abb. 1: Karaf-Modulübersicht

Artikelserie

Teil 1: Grundlagen und Neuigkeiten
Teil 2: Aufbau einer JPA-Anwendung
Teil 3: Webvisualisierung im Zusammenspiel mit Pax Web

In diesem Falle verwendet Apache Karaf das OPS4j. pax.logging-Framework. Es bietet eine einheitliche Logging-Schnittstelle zu vielen Logging-APIs (log4j, slf4j oder auch juli logging). Alle diese Logging-APIs werden unterstützt und landen nachher konsistent in einem Logging Appender. In diesem Fall wird intern auf Log4j (in Version 1) gesetzt.

Soweit die Einschätzung, wie Apache Karaf mit anderen Containern zu vergleichen ist. Alternativ gibt es im OSGi-Umfeld Eclipse Virgo als reinen OSGi Container, Oracles GlassFish als Java-EE-Server basierend auf Felix mit OSGi-Fähigkeiten und IBM WebSphere, ebenfalls ein Java-EE-Server mit OSGi-Fähigkeiten.

Vorstellung von Karaf

Im Folgenden werden die wichtigsten Komponenten des Karaf-Servers beleuchtet und in diesem Zusammenhang die Besonderheiten von Karaf 3.0 herausgestellt.

Das Fundament von Karaf bildet das OSGi Framework. Dabei bleibt es dem Nutzer überlassen, ob er lieber auf Apache Felix (Default) oder Eclipse Equinox setzt. Aufbauend auf dieser Basis bietet Karaf ein einheitliches Logging, das an einer Stelle konfiguriert wird. Karaf hat die GoGo Shell des Felix-Frameworks im Einsatz und erweitert diese um einige weitere Kommandos, die bei der Diagnose von Problemen äußerst hilfreich sind.

Deployments

Karaf bietet eine Vielzahl von unterschiedlichen Deployments an, zum Beispiel die Installation von Bundles direkt aus dem Maven Repository heraus, aus der Shell oder das Hot Deployment eines Bundles im Deploy-Verzeichnis. Eine weitere Alternative zum „händischen“

Installieren von Bundles ist das Installieren von so genannten *Features*. Features (Listing 1) beschreiben eine Liste von Bundles, definiert in einem einfachen XML, das unterscheiden kann zwischen wichtigen Bundles der Applikation (zwingend zu installieren) und Bundles, die Abhängigkeiten sind und die ggf. nicht neu installiert werden, wenn bereits vorhanden oder eine aktuellere Version vorhanden ist.

Hierbei definiert das Feature XML zum einen die zu installierenden Bundles, die logisch zusammengehören; zum anderen können hiermit auch gleich die benötigten Konfigurationen für einen Dienst mit angegeben werden. Diese Konfiguration wird beim Einlesen des Features-Service an den Configuration-Admin-Service übergeben und steht somit direkt dem zu konfigurierenden Service zur Verfügung. Neben den vielen unterstützten Möglichkeiten, eines oder mehrere Bundles in Karaf zu installieren, gibt es natürlich auch noch eine Vielzahl von unterstützten URL-Handlern:

- *mvn* – Handler für Maven-Koordinaten
- *file* – Filesystem
- *webbundle* – für Nicht-OSGi-Webapplikationsarchive (WAR), damit diese automatisch umgesetzt werden
- *wrap* – automatisches „Verpacken“ eines Nicht-OSGi-JARs in ein OSGi Bundle

Hier profitiert Karaf vor allem vom Pax-URL-Projekt, das die speziellen URL-Handler bereitstellt. So gibt es zum einen den *wrap* Handler, der dazu führt, dass Nicht-OSGi-JARs automatisch OSGi-fiziert werden. Dies geschieht dank eingebettetem BND (Das „Schweizer Armeemesser für OSGi“, [1]). Ein anderer, gern genutzter URL-Handler ist der *webbundle* Handler. Dieser macht aus einem einfachen WAR ein OSGi-fiziertes WAB (Web Application Bundle – auch bekannt als WAR mit OSGi-Manifest). Dieses wird intern auch vom Pax-Web-Deployer verwendet, der für die Installation von WAR-Dateien aus dem *Deploy*-Ordner verantwortlich ist.

Genau so einfach wie Bundles installiert werden können, kann auch die Konfiguration dem Container mitgegeben werden. Hierzu werden alle Konfigurationen im *etc*-Verzeichnis gehalten. Dabei gilt, dass alle Konfigurationen, die auf *.cfg* enden, automatisch vom Container erkannt und verwaltet werden. Verwendung findet hierbei der Felix *FileInstaller*. Dieser überwacht genau, wie beim *Deploy*-Ordner, das Verzeichnis, das die Konfigurationsdateien enthält. Dabei arbeitet der *FileInstaller* eng mit dem Configuration-Admin-Service zusammen. Für jede Datei wird eine entsprechende PID (persistent ID) angelegt. Die persistent ID ist der Grund für das Muster des Konfigurationsfiles, zum Beispiel *org.ops4j.pax.logging.cfg*. Hierbei wird der Service mit der PID *org.ops4j.pax.logging* konfiguriert. Eine Beispielkonfiguration per Configuration-Admin-Service anhand von Pax Logging zeigt Listing 2.

Es lassen sich auch Managed Service Factories damit umsetzen. Dazu sollte man aber die Dokumentation des Apache-Felix-Configuration-Admin-Service beachten [2].

Auch eine an Spring angelehnte Dependency Injection, vor allem für die Blueprint-Service-Spezifikation, wird dank Apache Aries von Haus aus mitgeliefert. Aufgrund des mit 3.0 neu hinzugekommenen Regionenkonzepts [3] ist es aber auch möglich, Eclipse Gemini Blueprints zu verwenden.

Ein weiteres Kennzeichen von Karaf sind die speziellen Deployer, die unter anderem im *Deploy*-Ordner vorhandene Blueprint-XML-Dateien behandeln und diese als OSGi Bundles direkt im Server installieren (Listing 3). Daraus ergibt sich der Vorteil, dass zum Beispiel spezielle Konfigurationen wie eine *DataSource* (mit oder ohne Pooling) für unterschiedliche Umgebungen erstellt werden können und diese auch einfach von einem Systemadministrator adaptiert werden können, ohne jedes Mal die Entwicklung zu Hilfe rufen zu müssen.

Listing 1

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<features name="java-magazin-sample-feature" xmlns:xsi="http://www.w3.org/2001/
                                                        XMLSchema-instance"
  xsi:schemaLocation="http://karaf.apache.org/xmlns/features/v1.0.0 http://karaf.
                                                    apache.org/xmlns/features/v1.0.0">
  <repository>mvn:org.apache.cxf.karaf/apache-cxf/${dependency.cxf.version}/xml/
                                                        features</repository>

  <feature name="my-first-sample" version="1.0.0"
  description="sample feature">
    <details>This is just a sample feature for being used in the JavaMagazin</details>
    <config name="de.inovex.javamagazin.sample ">
      de.inovex.javamagazin.sample.Config=true
    </config>
    <bundle>mvn:de.inovex.javamagazin.sample/my-sampe/1.0.0</bundle>
  </feature>
</features>
```

Listing 2

```
# Root logger
log4j.rootLogger=INFO, out, osgi:*
log4j.throwableRenderer=org.apache.log4j.OsgiThrowableRenderer

...

# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ISO8601} | %-5.5p | %-16.16t |
        %-32.32c{1} | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.data}/log/karaf.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=1MB
log4j.appender.out.maxBackupIndex=10
```

Neues

Soviel zu den Basics. Im Folgenden geht es nun um die neuen Features und Verbesserungen in Version 3.0. Eines der Key-Features, das aber eher unter der Haube steckt, ist die neue OSGi-5.0-Unterstützung von Karaf 3.0. Dementsprechend gibt es jetzt Unterstützung für die Frameworkneuerungen wie Resolver-Service mit dem generischen „capabilities and requirement model", Subsystem-Service, Repository-Service (Verbesserungen an OBR – OSGi Bundle Repository), Service Loader Mediator, verbesserte Unterstützung von JMX und Configuration-Admin-Service.

Auch ganz neu und nur so mit Karaf 3.0 möglich ist die komplette Absicherung des Karaf-Servers mit JAAS bzw. mit Role-Based Access Controls (RBAC). Alle Dienste (auch über JMX) und Shell-Kommandos können dadurch jetzt abgesichert werden. Besonders interessant ist hierbei, dass einzelne Shell-Kommandos abgesichert werden können.

So zeigt die Konfiguration für das *bundle*-Shell-Kommando, wie eine solche Konfiguration für ein einzelnes Shell-Kommando definiert wird (Listing 4). Es gilt: *command[option]=role*. Es werden alle Befehle mit *–f* (für force) nur für die Rolle *admin* freigegeben. Alle Befehle ohne diese Schalter sind für die Rolle *manager* freigegeben.

Listing 3

```
<blueprint xmlns=http://www.osgi.org/xmlns/blueprint/v1.0.0 xmlns:xsi=http://www.
    w3.org/2001/XMLSchema-instance xmlns:ext=http://aries.apache.org/blueprint/xmlns/
    blueprint-ext/v1.0.0 xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
        http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd http://cxf.apache.org/
                blueprint/core http://cxf.apache.org/schemas/blueprint/core.xsd">
  <ext:property-placeholder />
  <bean id="h2DataSource" class="org.h2.jdbcx.JdbcDataSource">
    <property name="URL" value="jdbc:h2:${karaf.data}/database/h2Test" />
    <property name="user" value="sa" />
    <property name="password" value="" />
  </bean>

  <service id="dataSourceService" interface="javax.sql.DataSource" ref="h2DataSource">
    <service-properties>
      <entry key="osgi.jndi.service.name" value="jdbc/route-test-ds" />
    </service-properties>
  </service>
</blueprint>
```

Listing 4

```
install = admin
refresh[/.*[-][f].*/] = admin
refresh = manager
restart[/.*[-][f].*/] = admin
restart = manager
start[/.*[-][f].*/] = admin
start = manager
stop[/.*[-][f].*/] = admin
stop = manager
uninstall[/.*[-][f].*/] = admin
uninstall = manager
update[/.*[-][f].*/] = admin
update = manager
watch = admin
```

Neues in Aktion

Nach dem Auspacken ist das Erste, was beim Starten auffällt, die neue Shell (Listing 5). Diese bietet dem Nutzer nicht nur das bekannte Verwenden von Kommando-„Scopes", sondern auch das Nutzen von Subshells. Konkret am Beispiel von *log* ist es möglich, sich mit dem Kommando *log:display* das aktuelle Logfile in der Shell anzuzeigen.

Neu ist, dass man durch das einfache Eingeben von *log* in den *log scope* wechseln kann (Listing 6). Dadurch bietet einem die Shell an dieser Stelle auch nur noch die Befehle innerhalb dieses Scopes an; um beim Beispiel von *log* zu bleiben: *display*. Anhand des geänderten Prompts kann man auch sofort erkennen, dass man gerade in einer Subshell ist.

Verlassen kann man die Subshell mit einem *exit*. Die Sichtbarkeit der Kommandos lässt sich, wie so vieles in Karaf, auch konfigurieren. Hierzu muss man lediglich das Kommando *shell:completion* mit dem gewünschten Modus ausführen. Hierzu stehen drei mögliche Modi zur Verfügung:

- *first*, der auch das Default-Verhalten ist, zeigt nur die in dieser Subshell gültigen Kommandos an.
- *global*, der alle Komplettierungen zulässt.
- *subshell*, der eine gut ausbalancierte Zwischenlösung ist; denn so werden einem im Kontext einer Subshell alle weiteren Subshells zur Verfügung gestellt, um so schnell zwischen den einzelnen Subshells wechseln zu können.

Wie unschwer zu erkennen ist, ist einer der größten Vorteile von Karaf die Shell inklusive der damit verbundenen Möglichkeiten. Gerade für die Entwicklung gibt es viele kleine Helferlein in der Shell, die einem bei der Entwicklung gerade von OSGi-Applikationen sehr behilflich sein können. Dies fängt mit *list* und *grep* an: *list* listet einem alle Nichtsystem-Bundles (möchte man auch die System-Bundles gelistet bekommen, dann kann man mit *list –l 0* alle ab Startlevel 0 sehen, dafür gibt es auch einen Alias *la* – für *list all*); *grep* sollte einem aus dem Unix-Umfeld bereits bekannt sein und bildet die gleiche bekannte Funktionalität ab. Beim Auflisten der Bundles sieht man zuerst den Zustand des Bundles. Es lässt sich einfach erkennen, ob das fragliche Bundle erfolgreich gestartet (*resolved* und *active*) ist. Aber auch bei Problemen gibt diese Auflistung bereits Informationen über

den Zustand (seit Karaf 3.0). Besitzen Bundles Blueprint- oder Spring-XML-Files, die durch die entsprechenden Extender gestartet werden, zeigt im Falle eines Blueprint GracePeriods die Auflistung diesen Zustand im Listing anstelle des Active-/Failed-Zustands an (Listing 7).

Um weiterführende Diagnoseinformationen zu bekommen, hilft im Falle eines nicht aktiven Bundles das *bundle:diag*-Kommando. Es zeigt, warum zum Beispiel das im Bundle enthaltene *Blueprint.xml* vom Blueprint Extender nicht gestartet werden konnte. Es könnte beispielsweise auf fehlende Services gewartet haben und ist dann in den Timeout gelaufen.

Mit *bundle:info* oder *bundle:header* bekommt man Detailinformationen über das Bundle selbst und darüber, welche Package-Imports für dieses Bundle aufgelöst wurden bzw. zu welchen Bundles diese gehen. *bundle:diag* (seit Karaf 3.0) hilft beim Finden von Fehlern in Bezug auf Bundles bzw. bei Startfehlern. Zum einen kann der Grund für eine Grace Period bei Blueprint Bundles angezeigt werden, zum anderen wird die entscheidende Exception, die zum Fehlschlagen der Bundle-Aktivierung geführt hat, dargestellt.

Listing 5

```
        __ __                  ____
       / //_/ ____ ___________/ __ /
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (3.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.

karaf@root()> log:display
```

Listing 6

```
        __ __                  ____
       / //_/ ____ ___________/ __ /
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (3.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.

karaf@root()> log
karaf@root(log)> display
```

Für den Entwickler ist der *bundle:watch*-Befehl ein wichtiger Helfer. Gerade bei der Entwicklung von OSGi Bundles kann Karaf dadurch automatisch das gerade neu gebaute Bundle aktualisieren. Hierzu registriert man die *BundleID* mit *bundle:watch 50*. Karaf untersucht dieses Bundle nach dem Ursprung der Installation, zum Beispiel die Maven-Koordinate oder die *file-location*, und überwacht somit das lokale Maven Repository bzw. das Filesystem auf Änderungen. Im Falle einer neu zur Verfügung stehenden Version des Bundles wird diese automatisch aktualisiert. Dies ist äquivalent zu *osgi:update BundleID*.

Bis jetzt haben wir die Möglichkeiten von Karaf, wie sie nach der Installation zur Verfügung stehen, verwendet. Quasi out of the box ist es bereits möglich, OSGi Bundles mit Blueprint zu verwenden, z. B. auch, um eigene Shell-Kommandos zu erstellen. Wenn nun aber Enterprise-Features wie zum Beispiel JPA/JTA verwen-

Listing 7

```
        __ __                  ____
       / //_/ ____ ___________/ __ /
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (3.0.0)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.

karaf@root()> la
START LEVEL 100 , List Threshold: 0
 ID | State    | Lvl | Version          | Name
-------------------------------------------------------------------------------
  0 | Active   |   0 | 4.2.1            | System Bundle
  1 | Active   |  25 | 3.0.0            | Apache Karaf :: Features :: Core
  2 | Active   |  20 | 1.1.0            | Apache Aries Util
  3 | Active   |  30 | 3.0.0            | Apache Karaf :: System :: Core
  4 | Active   |  20 | 1.3.0            | Apache Aries Blueprint Core, Fragments: 46
  5 | Active   |   5 | 1.6.0            | OPS4J Pax Url - Maven Commons
  6 | Active   |  30 | 0.2.1            | JLEdit :: Core
  7 | Active   |  20 | 1.0.2            | Apache Aries Proxy Service
  8 | Active   |  30 | 1.4.0            | OPS4J Base - Util - Collections
  9 | Active   |  24 | 3.0.0            | Apache Karaf :: Deployer :: Wrap Non OSGi Jar
 10 | Active   |  30 | 3.0.0            | Apache Karaf :: Bundle :: Core
...
```

Extender Pattern – Bundle Tracker

Das Extender Pattern wird im OSGi-Kontext vor allem zum Erweitern von Bundles verwendet. So kümmert sich der Blueprint Bundle Tracker darum, *blueprint.xml* in gestarteten Bundles zu finden und den Blueprint-Kontext zu starten.

det werden sollen, so müssen diese Features noch nachträglich installiert werden. Dazu muss man wieder in die Shell wechseln. Mit *feature:install jpa* z. B. wird das JPA-Feature von Aries installiert. Es stellt allerdings nur die Möglichkeit her, Karaf als Container für JPA zu verwenden – die konkrete JPA-Implementierung wie OpenJPA, EclipseLink oder Hibernate muss zusätzlich installiert werden. Hier bietet Apache Karaf eine weitere Neuerung: Als neues Enterprise-Feature wird ein OpenJPA-Feature mitgeliefert. Dadurch lässt sich mit einem einfachen Befehl alles Benötigte installieren.

Ein weiteres Enterprise-Feature ist *war*. Es setzt sich aus anderen Features wie zum Beispiel *http* und *http-whiteboard* zusammen. Mit der Installation des *http*-Features wird dem Karaf ein *OSGi HttpService* zur Verfügung gestellt. Ab diesem Zeitpunkt ist es möglich, nach OSGi-Spezifikation Servlets zu registrieren, JSPs sind in diesem Schritt noch nicht möglich. Mit der Installation des *http-whiteboard*-Features wird das Whiteboard Pattern (siehe gleichnamiger Kasten) zum *HttpService* hinzugefügt.

Karaf-Kommandos

Es gibt 281 Standardkommandos in Karaf. Hier ein Auszug der wichtigsten:

- *help* – Gibt eine Übersicht über alle Kommandos.
- *system:framework* – Frameworkoptionen, wie zum Beispiel *debug*.
- *system:shutdown* – Shutdown-Kommando, wenn mit *reboot* und *–c* kombiniert, wird das Datenverzeichnis beim Neustart gelöscht; vor allem für die Entwicklung interessant.
- *package:exports/imports* – Listet alle exportierten/importierten Packages eines Bundles.
- *jaas:** – Kommandos, um User/Gruppen und Rollen zu pflegen.
- *instance:** – Kommandos, um weitere Karaf-Instanzen zu verwalten, anzulegen oder zu löschen.
- *feature:** – Alles, was mit Features zu tun hat.
- *config:** – Vom Listen bis zum Neuanlegen von Konfigurationen für den Configuration-Admin-Service.
- *bundle:** – Alle Kommandos, die mit Bundles interagieren.
- zahlreiche Aliasse als Kurzform für die meist verwendeten Kommandos.

Whiteboard Pattern

Dieses Pattern beschreibt eine gängige Praxis im OSGi-Kontext. Hierbei geht es um eine Inversion of Control auf Serviceebene. Der Konsument eines bestimmten Dienstes stellt das entsprechende Interface bereit (bzw. es kann auch ein Public-API sein) und wartet auf die Bereitstellung des Dienstes. Das Bundle, das den Dienst bereitstellt, wartet auf die Verwendung durch den Konsumenten.
Einfaches Beispiel: Bundle A stellt Interface A zur Verfügung und hat einen Service-Listener auf diesem Interface. Bundle B implementiert Interface A und stellt diesen Dienst zur Verfügung. Der Service-Listener von Bundle A wird informiert und verwendet den bereitgestellten Dienst.

Ab jetzt ist es möglich, auch JSPs zu registrieren. Mit dem *web*-Feature stehen einem alle Möglichkeiten offen, auch ein *war* in Karaf zu deployen. Als weitere Neuerung werden von Karaf 3.0 auch Servlet-3.0-Annotationen unterstützt. Dies geht sogar soweit, dass ein JAR nur mit Servlets und dem *WebContext*-Path im Manifest reicht, um eine Webapplikation zu starten. Weitere Enterprise-Features sind:

- *hibernate* (als Alternative zu OpenJPA)
- *jndi*
- *jdbc*
- *jms* (liefert weitere Kommandos für JMS, z. B. Apache ActiveMQ)
- *openwebbeans* (CDI – benötigt zusätzlich Pax CDI)
- *weld* (CDI – benötigt zusätzlich Pax CDI)
- *application-without-isolation* (OSGi EBA spec)

Subprojekte

Neben Karaf als Top-Level-Projekt bei Apache bestehen auch Subprojekte, die Karaf erweitern. So gibt es, um beim Weintrinken zu bleiben, noch die Projekte Cellar und Cave. Mit Cave kann Karaf so erweitert werden, dass Karaf nun auch als OSGi Bundle Repository dienen kann. Dies ist vor allem interessant für Projekte, in denen der OBR Resolver zum Einsatz kommt und man eine der Karaf-Instanzen im Netzwerk als Repository dafür zur Verfügung stellen möchte.

Mit Cellar ist es möglich, mehrere Instanzen von Karaf innerhalb eines Netzwerksegments oder auch in der Cloud zu clustern. Dazu sprechen sich die einzelnen Instanzen untereinander ab bgzl. installierter Bundles, Konfigurationen und verteilen ggf. auch Events.

Ausblick

Nachdem die Grundlagen und die Neuerungen von Karaf behandelt wurden, kann hierauf in den folgenden Artikeln aufgebaut werden. Der nächste Artikel wird sich mit dem Aufbau einer JPA-Anwendung unter Verwendung der neuen Funktionalitäten und Kommandos befassen. Der dritte Teil dieser Serie wird basierend auf dem JPA-Beispiel die Möglichkeiten der Webvisualisierung im Zusammenspiel mit Pax Web zeigen.

Achim Nierbeck ist Senior-Entwickler/Architekt/Scrum Master bei der inovex GmbH und beackert das Feld der EE-Entwicklung schon seit über zwölf Jahren. Seit nunmehr drei Jahren beschäftigen ihn die Apache Karaf PMC OSGi und vor allem das Thema Karaf.

Links & Literatur

[1] http://www.aqute.biz/Bnd/Bnd

[2] http://felix.apache.org/documentation/subprojects/apache-felix-config-admin.html

[3] OSGi Compendium Specification 134-3 Subsystem Region