JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

# Serverless

Sonderdruck für


www.oio.de

## Microservices mit WildFly Swarm

▶ von Falk Sippach



Microservices mit WildFly Swarm

# Java EE in one JAR

Im Vergleich zur modernen, aufstrebenden Microservices-Bewegung wirken Applikationsserver etwas angestaubt. Viele Unternehmen möchten den neuen Architekturstil einsetzen, können und wollen aber weder auf Application Server noch auf das jahrelang angehäufte Wissen zum Java-Enterprise-Standard verzichten. WildFly Swarm wagt den Brückenschlag und erleichtert das Erstellen von Microservices-Architekturen auf Basis von Java EE. Der Application Server verschwindet dabei nicht ganz, er wird vielmehr restrukturiert und in die Anwendung mit eingepackt.

von Falk Sippach

Klassische Java-Enterprise-Anwendungen – egal ob als Laufzeitmonolith oder in Form von Microservices – baut man gegen eine Vielzahl von APIs aus dem Java-EE-Standard und installiert dann das schlanke Deployment-Artefakt (enthält keine Dependencies) in einen recht schwergewichtigen Applikationsserver, der alle notwendigen Bibliotheken mitbringt. Um Isolations- und Ressourcenkonflikten aus dem Weg zu gehen, wird aber typischerweise je Server genau nur eine Anwendung deployt. Dieses Vorgehen ist aufwendig und unflexibel. Außerdem wird dem Betrieb die Arbeit unnötig erschwert. In Zeiten von Continuous Delivery und DevOps ist die zusätzliche Installation von Anwendungscontainern tatsächlich ein fragwürdiges Vorgehen. Im Fall von Microservices enthält der vollständige Application Server zudem viel mehr Funktionalität als die Anwendung überhaupt benötigt.

Die Konkurrenz in Form von Spring Boot und Dropwizard hat es vorgemacht. Sie verpackt die Anwendung mitsamt den notwendigen Bibliotheken und einem Servlet-Container in ein ausführbares Fat JAR und bietet zudem diverse zusätzliche Dienste, um die Herausforderungen der Microservices-Architektur zu meistern.

### Fat JAR

Ein Fat JAR oder Uber JAR ist ein Java-Archiv, das neben den Klassen und Ressourcen des Projekts auch alle notwendigen Bibliotheken und den zur Ausführung notwendigen Applikationscontainer enthält. Die typischen Build-Management-Tools (Maven, Gradle) bieten Plug-ins an, die die deklarierten Dependencies entsprechend aufbereiten und in das JAR integrieren. Im Fall von WildFly Swarm werden die Abhängigkeiten als internes Maven Repository abgelegt und zur Laufzeit aufgelöst.

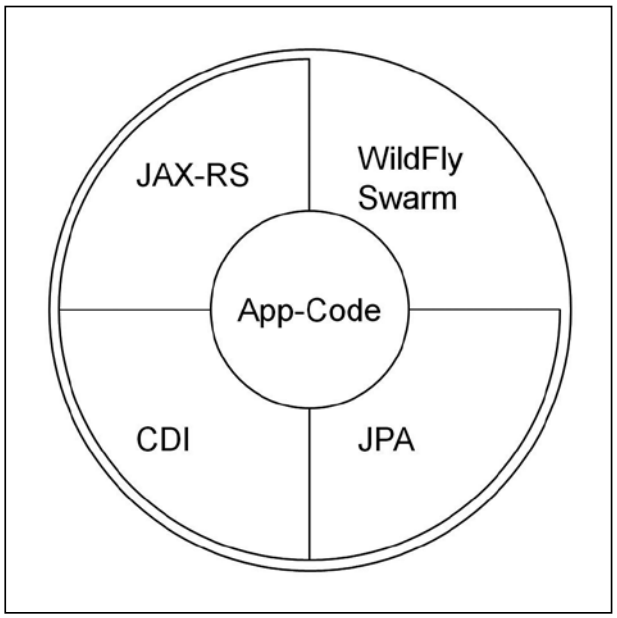


Abb. 1: WildFly-Swarm-Anwendung, die nur JAX-RS, CDI und JPA nutzt

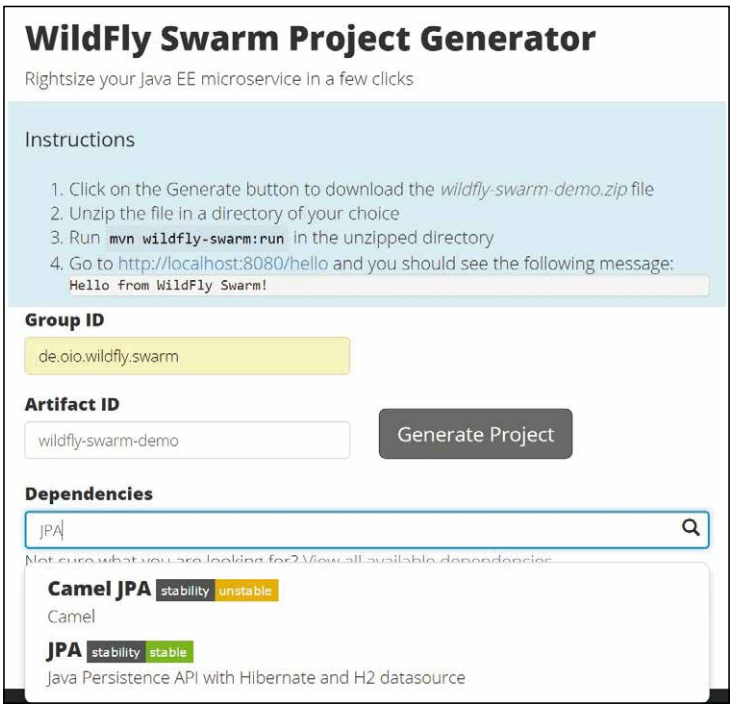


Abb. 2: WildFly Swarm Project Generator vereinfacht das Erstellen neuer Projekte

Mit WildFly Swarm [1] hat nun auch Red Hat/JBoss diesen Weg eingeschlagen. Die Besonderheit: Hier lässt sich der volle Java-EE-Stack nutzen. Im Gegensatz zu der sonst üblichen Integration von Tomcat oder Jetty kommt mit WildFly sogar ein vollwertiger Application Server zum Einsatz. Genau genommen wurde WildFly für Swarm in seine Bestandteile zerlegt. Über diverse Konfigurationsmöglichkeiten kann man steuern, wie viel App-Server man tatsächlich in seiner Anwendung verpacken möchte. Mit WildFly Swarm kann man also genau die APIs auswählen, die gebraucht werden (**Abb. 1**). Das spart Ressourcen und ermöglicht einen schlanken Betriebsprozess.

### Konfiguration: Alles läuft über Fractions

WildFly Swarm bindet die notwendigen Abhängigkeiten über so genannte Fractions ein, die sich entweder als Maven Dependencies oder programmatisch hinzufügen und konfigurieren lassen. Zum Erstellen des Fat JARs muss dabei zunächst nur der Build-Prozess angepasst werden. Das präferierte Vorgehen verwendet Maven (mindestens in der Version 3.3.x). Gradle wird zwar auch unterstützt, erhält aber nicht so viel Aufmerksamkeit seitens der Swarm-Entwickler und ist derzeit nicht so stabil. Um bereits existierende JARs oder WARs als Fat JAR zu verpacken, gibt es außerdem noch das Kommandozeilenwerkzeug swarmtool [2].

Möchte man ein bestehendes Java-EE-Projekt „swarmifizieren“, genügt theoretisch das Hinzufügen des WildFly-Swarm-Maven-Plug-ins (Listing 1). Mit *mvn package* wird dann das Uber JAR *myapp-swarm.jar* erstellt. Ausführen kann man die Anwendung entweder mit *mvn wildfly-swarm:run* oder mit *java -jar myapp-swarm.jar*.

Das Maven-Plug-in versucht automatisch, die benötigten Fractions zu erkennen, indem es die Anwendung scannt. Das ergibt aber nur Sinn, wenn man eine bestehende WAR-basierte Anwendung zu WildFly Swarm migrieren möchte, ohne alle Swarm Dependencies konfigurieren zu müssen.

#### Listing 1: Swarm-Maven-Plug-in

```
<build>
 <plugins>
  <plugin>
   <groupId>org.wildfly.swarm
                    </groupId>
   <artifactId>wildfly-swarm-plugin
                    </artifactId>
   <executions>
    <execution>
     <goals>
      <goal>package</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

#### Fraction

Das zentrale Element bei WildFly Swarm sind die Fractions, über die die notwendigen Abhängigkeiten zu Containerdiensten und externen Bibliotheken konfiguriert werden. Bei den Java-EE-Standard-Implementierungen entspricht eine Fraction typischerweise einem WildFly-Subsystem. Damit hat man zum Beispiel Zugriff auf JPA, JAX-RS, CDI oder JSF. Es können auch eigene Fractions erzeugt werden, die über transitive Abhängigkeiten andere Fractions aggregieren. Somit lassen sich Fractions kombinieren, sodass man sie anschließend nur über eine Abhängigkeit referenzieren kann. Die Liste der verfügbaren Fractions ist lang und wird ständig erweitert. Neben den WildFly-Subsystemen stehen beispielsweise auch Dienste aus dem Netflix-Open-Source-Stack, Logaggregation mit Logstash oder Authentifizierung/Autorisierung mit Keycloak zur Verfügung. Eine aktuelle Übersicht findet man im Swarm Project Generator [3].

Erstellt man ein neues WildFly-Swarm-Projekt, bietet es sich an, Swarm Project Generator [3] zu nutzen (**Abb. 2**). Nach der Eingabe der Maven-Projekt-Koordinaten kann man die verfügbaren Fractions auswählen und erhält anschließend ein fertiges Maven-Projekt als Download, das direkt gebaut und ausgeführt werden kann. Neben dem Swarm-Maven-Plug-in werden bei diesem Ansatz die Fractions explizit als Maven Dependencies gesetzt. Um alle Fractions in einer einheitlichen Versionsnummer einzubinden, bietet sich die Deklarierung des WildFly Swarm BOM (Bill of Materials) im Dependency Management an. Anschließend können die eigentlichen Fractions ohne Versionsangabe als Dependencies hinzugefügt werden (Listing 2).

Damit die Anwendung ausführbar ist, wird durch Swarm eine Klasse mit einer *Main*-Methode zur Verfügung gestellt, in der das Bootstrapping angestoßen wird. Man kann aber auch eine eigene *Main*-Methode implementieren und sie dann im Swarm-Maven-Plug-in referenzieren. Über eine DSL mit einem Fluent Interface lassen sich darin die Fractions feingranular konfigurieren und der Inhalt des Fat JARs festlegen. Letzteres passiert mit ShrinkWrap, einem Java-API zur programmatischen Manipulation von Java-Archiven im Speicher. Entstanden ist ShrinkWrap als ein Modul des Arquillian-Projekts, das auch aus dem Haus Red Hat kommt. In Listing 3 wird ein Container erzeugt und anschließend werden einige Fractions hinzugefügt. Danach wird ein Archiv inklusive REST-Endpunkt und JAX-RS-Application-Klasse gebaut und im Container deployt. Bei den Fractions kann man noch diverse Konfigurationsparameter einstellen, wie man an der JPA und *Datasources* Fraction sehen kann. Viele weitere Beispiele finden sich im WildFly-Swarm-Example-Projekt auf GitHub [4].

Die Konfigurationsmöglichkeiten der Swarm-DSL entsprechen den Parametern der XML-Konfiguration des WildFly Application Servers (*standalone.xml*). Kennt man diese bereits von WildFly-Projekten, findet man sich in Swarm relativ schnell zurecht. Man kann übrigens die Konfiguration auch in einer *standalone.xml* zur Verfügung stellen, die beim Start der Anwendung geladen wird:

```
java -jar myapp-swarm.jar -c configurations/standalone.xml
```

### Listing 2: Konfiguration der Fractions in Maven

```
<properties>
  <version.wildfly.swarm>2016.9</version.wildfly.swarm>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <failOnMissingWebXml>false</failOnMissingWebXml>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.wildfly.swarm</groupId>
      <artifactId>bom-all</artifactId>
      <version>${version.wildfly.swarm}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <!-- Java EE 7 dependency -->
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
  <!-- Wildfly Swarm Fractions -->
  <dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>jaxrs</artifactId>
  </dependency>
[...]
  </dependencies>
</project>
```

### Listing 3: Programmatische Konfiguration der Fractions

```
public static void main(String[] args) throws Exception {
  Swarm container = new Swarm()
    .fraction(new JAXRSFraction())
    .fraction(new CDIFraction())
    .fraction(new LoggingFraction())
    .fraction(new JPAFraction()
      .inhibitDefaultDatasource()
      .defaultDatasource("SpeakerDS"))
    .fraction(new DatasourcesFraction()
      .jdbcDriver(new JDBCDriver("h2")
        .driverName("h2")
        .driverClassName("org.h2.Driver")
        .xaDatasourceClass("org.h2.jdbcx.JdbcDataSource")
        .driverModuleName("com.h2database.h2"))
      .dataSource(new DataSource("SpeakerDS")
        .driverName("h2")
        .jndiName("java:/SpeakerDS")
        .connectionUrl("jdbc:h2:./library;DB_CLOSE_ON_EXIT=TRUE")
        .userName("sa")
        .password("sa")));
  container.start();

  JAXRSArchive appDeployment = ShrinkWrap.create(JAXRSArchive.class);
  appDeployment.addResource(Application.class);
  appDeployment.addResource(SpeakersEndpoint.class);
  appDeployment.addAllDependencies();
  container.deploy(appDeployment);
}
```

Neben der gesamten *standalone.xml* lassen sich auch nur Teile der Konfiguration externalisieren und zum Beispiel für verschiedene Umgebungen explizit überschreiben. Entweder man setzt Systemvariablen, übergibt Environment-Variablen beim Programmstart oder man referenziert Textdateien (Properties oder YAML). Listing 4 zeigt ein YAML-Beispiel, das eine Defaultkonfiguration (erster Block) und weitere Blöcke für unterschiedliche Umgebungen (Stages) zur Verfügung stellt. Den Namen der Stage kann man dann beim Start der Anwendung übergeben:

```
java -jar myapp-swarm.jar -S development
```

Wurde keine Stage angegeben, wird der Defaultblock verwendet. Die Parameter aus den Properties- oder YAML-Dateien können über Platzhalter in der *standalone.xml* oder auch in der programmatischen Konfiguration aufgelöst werden. Weitere Einstellungsmöglichkeiten und die Defaultwerte kann man der Referenzdokumentation entnehmen [5].

### Kommunikation: REST und Messaging

Im Gegensatz zu einem Deployment-Monolithen müssen Microservices über ihre Systemgrenzen hinweg miteinander kommunizieren. Die Entkopplung der einzelnen Services ist ein Kernaspekt dieses Architekturstils. Typischerweise verwendet man für synchrone Aufrufe REST-Schnittstellen über das HTTP-Protokoll und für asynchrone Kommunikation Messaging. Beide Arten werden durch den Java-EE-Standard bereits abgedeckt. Messaging funktioniert derzeit aber noch nicht zur Interprozesskommunikation, sondern nur innerhalb eines einzelnen Service. Als Message Broker kommt ActiveMQ zum Einsatz.

REST-Schnittstellen lassen sich mittels JAX-RS einfach veröffentlichen und mit dem Client-API auf-

**Listing 4: „project-stages.yml“**

```
logger:
  level: DEBUG
swarm:
  port:
    offset: 10
---
project:
  stage: development
logger:
  level: DEBUG
swarm:
  port:
    offset: 50
---
project:
  stage: production
logger:
  level: INFO
swarm:
  port:
    offset: 100
}
```

rufen. Über die Swagger Fraction kann zudem eine API-Beschreibung veröffentlicht und mittels des WildFly-Swarm-Swagger-UI-Servers als dynamisch generierte Dokumentation zur Verfügung gestellt werden.

### Logging, Managementmonitoring und Health Checks

Da die aktuellen Bordmittel des Java-EE-7-Standards nicht für die Entwicklung praxistauglicher Microservices ausreichen, stellt das WildFly-Swarm-Team diverse Zusatzbibliotheken als Fractions zur Verfügung. Die Auswahl ist zwar noch nicht so groß wie beispielsweise bei Spring Boot, aber das Angebot wird ständig ausgebaut. Auch mit den bisher vorhandenen lassen sich bereits produktionsreife Services entwickeln.

Ein wichtiger Aspekt ist das zentralisierte Einsammeln der Loginformationen. Sonst tut man sich mit der manuellen Fehlersuche in den einzelnen Logdateien über die Vielzahl der kleinen Serviceinstallationen keinen Gefallen. WildFly Swarm stellt eine Logstash Fraction zur Verfügung, die alle Logausgaben an einen parallel laufenden Logstash-Server weiterreicht. Das eigentliche Logging funktioniert übrigens mit JBoss Logging und wird standardmäßig auf der Konsole auf dem INFO-Level ausgegeben – ist aber konfigurierbar.

Neben dem zentralen Logging sind Health Checks und Monitoring zur Überwachung der Serviceinstallationen unerlässlich. Dazu muss die Monitor-Fraction hinzugefügt werden. Jeder Knoten bietet dann zur Laufzeit Informationen zum Deployment (*/node*), Speicherverbrauch (*/heap*) und Threadverhalten (*/threads*) an, die wiederum Monitoringsoftware periodisch abrufen kann. Über Health Checks können zudem kritische Zustände sichtbar gemacht werden. Ein REST-Endpunkt wird hierfür mit *@Health* annotiert und liefert eine Instanz von *HealthStatus* zurück. Fällt der freie Speicherplatz in Listing 5 unter einen Schwellwert, wird eine negative Rückmeldung geliefert. Verfügbar ist dieser Health Check dann übrigens unter */health/diskSpace*. Zugriffsmöglichkeiten im laufenden Betrieb bieten die Management-Fraction (nutzt das WildFly-HTTP-Management-Interface) oder der JMX over HTTP Connector Jolokia.

**Listing 5: Health Check**

```
@GET
@Path("/diskSpace")
@Health
public HealthStatus checkDiskspace() {
  File path = new File(".");
  long freeBytes = path.getFreeSpace();
  long threshold = 1024 * 1024 * 100; // 100mb
  return freeBytes>threshold ? HealthStatus.up() :
    HealthStatus.down().withAttribute("freebytes", freeBytes);
}
```

**Listing 6: Konfiguration Security Realm**

```
new ManagementFraction()
  .securityRealm("ManagementRealm", (realm) -> {
    realm.inMemoryAuthentication( (authn)->{
      authn.add( "admin", "geheim", true );
    });
    realm.inMemoryAuthorization( (authz)->{
      authz.add( "admin", "admin" );
    });
  })
```

### Security: einen Security Realm einrichten

Anwendungen muss man gegen unerlaubte Zugriffe absichern. So sollte insbesondere für die Managementkonsole und die Monitoringendpunkte ein Security Realm konfiguriert werden. Neben dem üblichen Weg über die *standalone.xml* und *mgmt-users.properties* kann man über die Management-Fraction auch eine In-Memory-Authentifizierung einrichten (Listing 6).

Für die Absicherung einer Webanwendung wird man aus dem Java-EE-Standard typischerweise JAAS verwenden. Damit lässt sich eine Basic- oder formularbasierte Authentifizierung umsetzen. Es gibt dafür keine spezielle WildFly-Swarm-Unterstützung, auf GitHub findet man aber ein Beispiel dazu [4].

Mit Keycloak lässt sich zudem eine Single-Sign-On-Lösung integrieren, die als zentrale Nutzerverwaltung fungiert und auch die Authentifizierung über Social Log-ins ermöglicht. Keycloak kann dabei als separater Server parallel laufen oder sogar innerhalb der WildFly-Swarm-Anwendung mit hochgefahren werden. Die Konfigurationsoberfläche ist dann unter */auth/admin* verfügbar. So sichert man bestimmte Zugriffspfade der Anwendung ab:

```
JAXRSArchive deployment = ShrinkWrap.create(JAXRSArchive.class);
deployment.as(Secured.class)
  .protect( "/backoffice" )
  .withMethod( "GET" )
  .withRoles( "admin" );
```

### Verteilung und Widerstandsfähigkeit

Die Nachteile des Deployment-Monolithen sind hinlänglich bekannt, durch eine Microservices-Architektur wird aber leider auch nicht alles besser. Denn ein Microservice kommt selten allein daher. Vielmehr steigt die Komplexität zur Laufzeit, wenn sehr viele unabhängige Services in Produktion gebracht werden und untereinander kommunizieren müssen. Das Lokalisieren und Interagieren unter den Services in wechselnden Deployment-Landschaften ist dabei eine nicht zu unterschätzende Herausforderung. Helfen können dabei eine Service Registration und das Auffinden anderer Dienste über Service Discovery. WildFly Swarm abstrahiert von konkreten Implementierungen der Service Discovery über die Topology Fraction. Das Veröffentlichen eines

Service erfolgt über das Konvertieren in ein *Topology-Archive* und dem Aufruf von *advertise()*:

```
JAXRSArchive deployment = ShrinkWrap.create(JAXRSArchive.class);

// advertise this deployment as 'speakers'
deployment.as(TopologyArchive.class).advertise("speakers");
```

Über die Topology-Instanz können dann andere Services gefunden werden. Dazu genügt es, eine Look-up-Methode (funktioniert intern über JNDI) aufzurufen:

```
Topology topology = Topology.lookup();
topology.asMap();
```

Über *topology.asMap()* bekommt man dann Zugriff auf alle derzeit registrierten Services. Zudem kann man *TopologyListener* hinzufügen, um sich auch über die zukünftigen Änderungen der Servicelandschaft informieren zu lassen. Über die Fraction *topology-webapp* lässt sich übrigens auch aus dem Browser per JavaScript auf die Service Discovery zugreifen.

Die Topology Fraction stellt nur ein API zur Verfügung und benötigt eine konkrete Implementierung. Davon gibt es aktuell zwei: JGroups und Consul. JGroups wird im WildFly Application Server für das Clustering verwendet und ergibt bei Swarm vor allem in reinen WildFly-Szenarien Sinn. Es wird keine zusätzliche Serverinstanz benötigt, allerdings muss Multicast zur Verfügung stehen. Für die initiale Konfiguration reicht es, die *topology-jgroups* Fraction hinzuzufügen. Die einzelnen Instanzknoten finden sich dann automatisch. In heterogenen Serviceumgebungen muss man auf Consul zurückgreifen. Im Vergleich zu dem im WildFly bereits integrierten JGroups ist das Einrichten aber deutlich aufwendiger. So muss zunächst ein Consul-Server aufgesetzt werden. Jeder Service benötigt zusätzlich noch einen Consul Agent, um hartcodierte Hostkonfigurationen zu vermeiden. Schöner Nebeneffekt ist der Aufbau eines fehlertoleranten Systems, das auch Ausfälle einzelner Service-Discovery-Server aushält.

Netflix und seine Infrastruktur wird aktuell häufig als Musterbeispiel für eine gute Microservices-Umsetzung herangezogen. In WildFly Swarm kann man aus dem Netflix-Stack mit Ribbon clientseitiges Loadbalancing und mit Hystrix das Circuit-Breaker-Pattern einsetzen. Über die Ribbon Fraction werden automatisch transitive Dependencies wie Hystrix, RxJava und Netty ins Projekt gezogen. So wird ein Service in der Service Registration (je nach Konfiguration JGroups oder Consul) veröffentlicht:

```
deployment.as(RibbonArchive.class).advertise();
```

Auf der Clientseite (Listing 7) wird dann ein Ribbon-Proxy erstellt, der entweder den echten Remoteservice befragt oder alternativ, bei Nichterreichbarkeit (offene Sicherung), über einen Fallback-Handler eine sinnvolle Alternative anbietet. Hystrix prüft regelmäßig auf die Wiedererreichbarkeit des entfernten Service, um den Sicherungskreislauf gegebenenfalls wieder zu schließen.

### Testen mit Arquillian

Java-EE-Komponenten sind in den letzten Versionen immer leichtgewichtiger geworden und lassen sich daher sehr gut mit Unit-Tests auf Korrektheit prüfen. Für Integrationstests bietet der Standard allerdings keine Lösung. Mit Arquillian lassen sich aber auch WildFly-Swarm-Anwendungen einfach testen. Dazu müssen die zu testenden Artefakte zusammengepackt und der integrierte Container hochgefahren werden. Anschließend können von innen mit Integrationstests oder von außen mit funktionalen Tests (z. B. Selenium) Erwartungen geprüft werden. Die Testklasse muss dafür vom Arquillian Test Runner ausgeführt werden. In einer mit *@Deployment* annotierten Methode wird das ShrinkWrap-Archiv zusammengebaut, auf dessen Basis dann die Tests (mit *@Test* annotierte Methoden) ausgeführt werden. Bei Verwendung von CDI lassen sich die Dependencies innerhalb der Testklasse natürlich auch injizieren (Listing 8).

### Fazit

WildFly Swarm scheint in der aktuellen Diskussion rund um Microservices und Java EE gut aufgestellt zu sein. Ende Juni wurde die erste finale Version 1.0.0 zur Verfügung gestellt. Seitdem gibt es monatliche Updates, wenn auch mit einer etwas eigenwilligen Versionierung (2016.8.1 bzw. 2016.9). Auf der Roadmap stehen zu-

**Listing 7: Clientseitiges Load Balancing und Circuit Breaker**

```
@ResourceGroup( name="speaker" )
public interface SpeakerServiceService {
  SpeakerService INSTANCE = Ribbon.from(SpeakerService.class);

  @TemplateName("speakers")
  @Http(method = Http.HttpMethod.GET, uri = "/")
  @Hystrix(fallbackHandler = SpeakerFallbackHandler.class)
  RibbonRequest<ByteBuf> getSpeakers();
}

public class SpeakerFallbackHandler implements FallbackHandler<ByteBuf> {
  @Override
  public Observable<ByteBuf> getFallback(HystrixInvokableInfo<?> hystrixInfo,
                                         Map<String, Object> requestProps) {
    String fallback = "...";
    byte[] bytes = fallback.getBytes();
    ByteBuf byteBuf = UnpooledByteBufAllocator.DEFAULT.buffer(bytes.length);
    byteBuf.writeBytes(bytes);
    return Observable.just(byteBuf);
  }
}
```

dem die Integration mit einem API-Gateway, Verbesserungen bei Cloud-Diensten (OpenShift, Kubernetes), Abstraktionen für verschiedene Umgebungen (Entwicklung, Test, CI, Produktion, Cloud) und Verbesserungen im Tooling (IDEs). Inwieweit die zukünftigen Pläne von Oracle zu Java EE mit der MicroProfile-Initiative zusammenkommen werden [6] [7], bleibt abzuwarten. WildFly Swarm wird aber vermutlich relativ schnell auf Änderungen reagieren können. Daher kann man gespannt sein, wie sich die Situation entwickelt und ob WildFly Swarm seine Position als derzeit bekanntestes Java EE Microservice Framework behaupten wird.

**Listing 8: Arquillian-Test**

```
@RunWith(Arquillian.class)
public class MyTest {

  @Inject
  private SpeakerService service;

  @Test
  public void testSpeakerService() {
    // assert something about this.service
  }

  @Deployment
  public static Archive createDeployment() {
    JARArchive archive = ShrinkWrap.create( JARArchive.class );
    deployment.addResource( SpeakerService.class );
    return archive;
  }
}
```

Nicht vergessen sollte man, dass auch Payara, IBM und Tomitribe an Java-EE-Implementierungen für Microservices-Architekturen arbeiten. Aber Konkurrenz belebt das Geschäft. Und wenn am Ende die Standardisierung schritthält, erhalten wir Programmierer neue und idealerweise austauschbare Alternativen, um Microservices mit Java-EE-Technologien umzusetzen.

**Falk Sippach** hat über fünfzehn Jahre Erfahrung mit Java und ist bei der Mannheimer Firma OIO Orientation in Objects GmbH als Trainer, Softwareentwickler und Projektleiter tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit anderen die örtliche Java User Group.

**@sippsack**

## Links & Literatur

[1] WildFly Swarm Project: http://wildfly-swarm.io/

[2] swarmtool: https://github.com/wildfly-swarm/wildfly-swarm/tree/master/swarmtool

[3] WildFly Swarm Project Generator: http://wildfly-swarm.io/generator/

[4] WildFly-Swarm-Beispiele: https://github.com/wildfly-swarm/wildfly-swarm-examples

[5] WildFly-Swarm-Dokumentation: https://wildfly-swarm.gitbooks.io/wildfly-swarm-users-guide/

[6] Oracle Unveils Plan to Revamp Java EE 8 for the Cloud: https://www.infoq.com/news/2016/08/oracle-java-ee-cloud

[7] Schlosser, Hartmut: „Oracle sieht „reaktives Programmiermodell" für Java EE 8 vor": https://jaxenter.de/oracle-javaee-release-45210