Micro-Services wider Monolithen

Micro-Services in Java realisieren – Teil 1: Leichtgewichtige Web-Apps mit DropWizard

Phillip Ghadir

Dieser zweiteilige Beitrag stellt zwei interessante Zutaten für den Bau von Micro-Services vor: Docker.io zum Bereitstellen von definierten Umgebungen für unsere Services und DropWizard zum Realisieren von Webanwendungen. Im ersten Teil beschäftigen wir uns zuerst einmal mit der Realisierung und einem dafür passenden Framework.

Warum sprechen alle von Micro-Services?

Micro-Services sind derzeit in aller Munde. Man verspricht sich unabhängige Teams und das Lösen unterschiedlichster Probleme mit großen Systemen.

Große monolithische Systeme neigen tendenziell dazu, viele Verantwortlichkeiten zu erfüllen und damit gegen das Single Responsibility Principle [Mart08] zu verstoßen. Die Folge ist eine geringe Kohäsion auf Systemebene. Des Weiteren binden Monolithen alle fachlichen Bestandteile an alle technischen Abhängigkeiten des Monolithen. Der Zyklus für die technische Pflege wird dadurch in der Realität an den der fachlichen Entwicklung gebunden, selbst wenn beide theoretisch voneinander unabhängig sind.

Diese wechselseitigen Abhängigkeiten können durchbrochen werden, wenn jede Funktionalität nur von ihrer Infrastruktur abhängt und gleichzeitig unabhängig von allen anderen Infrastrukturelementen ist, die sie nicht tangieren.

Micro-Services = Single-Purpose Monolithen

Mit einem Micro-Service verbindet man im Allgemeinen, dass er genau einem Zweck dient, über Remote-Schnittstellen verwendet werden kann und über einfache Start-/Stopp-Funktionalität verfügt. Im Normalfall sind dies Starten über die Kommandozeile und Stoppen per Senden eines SIG TERM-Signals. Alles, was ein Micro-Service benötigt, sollte entweder direkt im Service – statisch gebunden – enthalten sein oder aber selbst wiederum remote aufgerufen werden.

Eigenschaften von Micro-Services

Radikale Vertreter verbinden mit Micro-Services eine absolute maximale Anzahl von Code-Zeilen. Während ich das fixe Limitieren der Code-Größe ablehnen würde, teile ich die dahinterliegende Motivation durchaus: Bei Änderungsanforderungen am Service sollte es recht einfach möglich sein, ihn vollständig neuzuimplementieren.

Der Fokus von Micro-Services liegt auf dem Bereitstellen der Funktionalität für (teilweise) unbekannte Clients. Während sich viele Entwicklungsorganisationen darauf konzentrieren zu implementieren, verschiebt der Service-Gedanke den Fokus auf das produktive Bereitstellen des Service in der definierten Zielumgebung. Mein alter Entwicklungsleiter hatte bereits in den 90-ern eine klare Vorstellung davon, was „das Feature ist fertig" bedeutete: „Der Kunde nutzt das Feature seit drei Monaten täglich ohne schwere Fehler gemeldet oder Änderungen gefordert zu haben."

Diese Philosophie teilen auch Unternehmen wie Facebook, während Unternehmen wie Netflix sogar noch weiter gehen: „Ein Service ist fertig, wenn er nicht mehr produktiv verwendet wird." (siehe [Cock14])

Das Bereitstellen eines Service ist nicht auf seine Installation in der Produktionsumgebung beschränkt, sondern erstreckt sich auch auf dessen Betrieb. Damit ergeben sich ein paar Herausforderungen.

Systeme von Systemen

Wer ein System aus einer Menge anderer Systeme zusammenbaut, muss den Zustand des Gesamtsystems überwachen. Dazu gilt es eine Reihe von Fragen beantworten zu können: Welche Services haben gerade Überlast oder sind gerade nicht verfügbar? Welche Services liefern gerade zuverlässig Daten, die nicht verarbeitet werden können? Und so weiter.

Die Liste der Fragen ist lang. Die Anforderung ist für alle gleich: Ein aus vielen Systemen bestehendes System kann nur dadurch überwacht werden, dass jedes einzelne überwacht werden kann.

In dem letzten Beitrag in dieser Kolumne haben wir das Hystrix-Framework kennengelernt, das wir im Micro-Service dazu verwenden können, die Abhängigkeiten zu überwachen und zu entkoppeln.

Eigenschaften von Micro-Services

Ein Micro-Service
- dient genau einem Zweck
- bietet Funktionalität über standardisierte Remote-Schnittstellen
- orientiert sich an den Business-Capabilities

DropWizard – Produktionstaugliche Web-Apps bauen

DropWizard ist eine Auswahl produktionserprobter, robuster Frameworks und Bibliotheken zum Bauen von eigenständigen, produktionstauglichen Webanwendungen. Was in DropWizard Applikation genannt wird, passt hier gut zu der Vorstellung eines Micro-Service. Um mit DropWizard Anwendungen zu bauen, genügt im Wesentlichen das Hinzufügen einer einzigen Abhängigkeit in der Maven-POM

```
<dependencies>
  <dependency>
    <groupId>io.dropwizard</groupId>
    <artifactId>dropwizard-core</artifactId>
    <version>0.7.0</version>
  </dependency>
</dependencies>
```

Damit zieht man die transitiven Abhängigkeiten ein:

- den Webserver Jetty,
- das JaxRS-Framework Jersey,
- die JSON-Bibliothek Jackson,
- das Metrik-Framework Metrics

sowie:

- Guava,
- LogBack und slf4j,
- Hibernate-Validator,
- Apache HttpClient und Jersey-Client,
- JDBI,
- Liquibase,
- Freemarker und Mustache sowie
- Joda Time.

Bausteine einer DropWizard-App

DropWizard erfordert das Schreiben einer Applikationsklasse, die von **`Application<C extends Configuration>`** erben muss. Darüber hinaus werden spezifische Konfigurationsschalter durch eine eigens zu implementierende und von Configuration zu erbende Klasse gekapselt.

In unserer Application-Ableitung müssen zwei Methoden definiert werden: **`initialize()`** und **`run(C, Environment)`**. Während **`initialize`** sicherstellen soll, dass die umgebungsspezifische Konfiguration korrekt ist, dient **`run()`** dazu, die einzelnen anwendungsspezifischen Komponenten – wie Web-Ressourcen, Servlets und HealthChecks – zu registrieren.

Typischerweise werden DropWizard-Anwendungen als eigen- und vollständige JARs zusammengepackt, die alles Notwendige enthalten, um den Service zu starten. Der Kasten „Fette JARs bauen" erläutert, wie man dies in der Maven-POM konfiguriert. Ein solches Micro-Service-JAR können wir dann aus der Shell starten – per

```
java -jar <fettes_micro_service_jar> server <config_file.yml>
```

Fehlen die Kommandozeilenparameter **`server`** und **`config_file.yml`**, gibt DropWizard eine Hilfe-Meldung über die Nutzung aus. Diese Art der Verwendung ist konform zu den Kriterien, die für 12 Factors Apps gelten (siehe gleichnamiger Kasten).

Damit unser Micro-Service auch die richtigen Komponenten initialisieren kann, muss die **`Main`**-Klasse in der **`main`**-Methode die Applikation erzeugen und deren **`run`**-Methode aufrufen. Für das Beispiel des Dokumentenarchivs aus dem letzten Beitrag [Ghad14] könnten wir zum Beispiel eine einfache Implementierung bereitstellen, die noch nicht viel tut. Die **`main`**-Methode erzeugt und startet dann einfach unsere Applikation **`SimpleDocumentStore`**

```
package com.innoq.sds;
import io.dropwizard.Application;
import io.dropwizard.Configuration;
import io.dropwizard.setup.Bootstrap;
import io.dropwizard.setup.Environment;

public class SimpleDocumentStore extends Application<SDSConfig> {
  @Override
  public void initialize(Bootstrap<SDSConfig> bootstrap) {
    // intentionally left blank
  }

  @Override
  public void run(
    SDSConfig sdsConfig,
    Environment environment) throws Exception {
      environment.jersey().register(new DocumentResource());
      environment.getApplicationContext().addServlet(
        DirServlet.class, "/dir" );
  }

  public static void main(String[] args) throws Exception {
    new SimpleDocumentStore().run(args);
  }
}
```

Fette JARs bauen

Um einen mit DropWizard geschriebenen Micro-Service mit allen Abhängigkeiten in ein JAR zu verpacken, sodass er eigenständig funktioniert, verwenden wir das Maven-Shade-Plug-in, das wir in der POM unter /project/build/plugins konfigurieren

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.6</version>
  <configuration>
    <createDependencyReducedPom>true</createDependencyReducedPom>
    <filters>
      <filter>
        <artifact>*:*</artifact>
        <excludes>
          <exclude>META-INF/*.SF</exclude>
          <exclude>META-INF/*.DSA</exclude>
          <exclude>META-INF/*.RSA</exclude>
        </excludes>
      </filter>
    </filters>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <transformers>
          <transformer implementation=
  "org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>
          <transformer implementation=
  "org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <mainClass>com.innoq.SimpleDocumentStore</mainClass>
          </transformer>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Interessant ist die Implementierung der `run`-Methode, die eine Configuration- und Environment-Instanz übergeben bekommt. In unserem Beispiel registrieren wir eine `DocumentResource` (eine gewöhnliche Jersey-Resource) sowie `DirServlet` (ein gewöhnliches HttpServlet). Wie Jersey-/JaxRS-Resourcen implementiert werden können, erläutert beispielsweise der Artikel „REST in Peace" [Tödt13].

12 Factors App

- Der gesamte Code eines Service wird im Versionsverwaltungssystem verwaltet und für viele Deploys genutzt.
- Jeder Service deklariert explizit seine Abhängigkeiten, kapselt sie als Bibliotheken oder aber als Back-End-Services, die remote aufgerufen werden.
- Die Konfiguration jedes Service liegt in der Umgebung, sodass kritische Konfigurationselemente nicht aus Versehen mit eingecheckt werden.
- Backing Services werden als eigene Ressourcen behandelt.
- Die Phasen Build, Release und Run werden strikt voneinander getrennt.
- Eine App (hier: ein Service) wird als eine nicht-leere Menge zustandsloser Prozesse ausgeführt.
- Services werden explizit über Remote-Schnittstellen angesprochen, die über gebundene Ports aufgerufen werden können.
- Skaliert wird über das Starten zusätzlicher paralleler Prozesse (Strategie: horizontal scale out).
- Maximiert die Stabilität durch kurze Startzeiten und graceful shutdown.
- Entwicklungs-/Test- und Produktionsumgebungen sollten so ähnlich wie möglich sein.
- Verwende Logs als Event-Streams.
- Alle Admin-Tasks sollten als einfach startbare Skripte oder Funktionen automatisiert werden

Systemüberwachung durch HealthChecks

Wie wichtig den DropWizard-Autoren das Entwickeln produktionstauglicher Services ist, wird durch die Warnung deutlich, die DropWizard beim Start ausgibt, sofern wir keine spezifischen HealthChecks registrieren

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! THIS APPLICATION HAS NO HEALTHCHECKS. THIS MEANS YOU WILL NEVER KNOW !
! IF IT DIES IN PRODUCTION, WHICH MEANS YOU WILL NEVER KNOW IF YOU'RE !
! LETTING YOUR USERS DOWN. YOU SHOULD ADD A HEALTHCHECK FOR EACH OF YOUR !
! APPLICATION'S DEPENDENCIES WHICH FULLY (BUT LIGHTLY) TESTS IT. !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Anwendungsspezifische HealthChecks können wir einfach in der `run`-Methode der Applikation registrieren. DropWizard verwendet dafür das Metrics-Framework [Metrics]. Wir fügen das Registrieren unserer `StorageHealthCheck`-Instanz in der `run`-Methode der `SimpleDocumentStore`-Klasse so hinzu

```
environment.healthChecks().register(
  "storageDir",
  new StorageHealthCheck( sdsConfig.getStorageDir() ) );
```

Anwendungsspezifische HealthChecks

Für ein Archivsystem könnten wir eine Reihe verschiedener HealthChecks sinnvoll finden: Ist das Storage-Back-End noch da? – Funktioniert der Zugriff auf das Autorisierungsmodul? – Verfügt der Storage noch über ausreichend Speicherplatz? – Wieviele File-Handles sind noch verfügbar?

Die Liste sinnvoller Checks ist lang. Wir beschränken uns hier auf einen trivialen Check: Wir prüfen, ob das konfigurierte Verzeichnis für den Back-End-Storage existiert und tatsächlich ein Verzeichnis ist.

Unsere Implementierung dafür haben wir bereits registriert. Die Implementierung der Klasse `StorageHealthCheck` sieht so aus

```
package com.innoq.sds;
import com.codahale.metrics.health.HealthCheck;
import java.io.File;

public class StorageHealthCheck extends HealthCheck {
  enum StorageCheckResults {
    okay(HealthCheck.Result.healthy("StorageDir is not a directory")),
    notDirectory(HealthCheck.Result.unhealthy(
      "StorageDir is not a directory")),
    doesNotExist(HealthCheck.Result.unhealthy(
      "storageDir is configured but doesn't exist!"));
    final Result result;

    StorageCheckResults( Result result) {
      this.result = result;
    }
  }
  final String storageDir;

  public StorageHealthCheck(String storageDir) {
    this.storageDir = storageDir;
  }

  @Override
  protected Result check() throws Exception {
    File dir = new File( storageDir );
    StorageCheckResults r =
      dir.isDirectory() ? StorageCheckResults.okay :
      dir.exists()      ? StorageCheckResults.notDirectory :
                          StorageCheckResults.doesNotExist;
    return r.result;
  }
}
```

Die Ausgabe des HealthChecks mit korrekt konfigurierter Umgebung

```
{
  "Deadlocks" : {"healthy":true},
  "StorageDir": {"healthy":true}
}
```

Eine Ausgabe des HealthChecks mit falsch konfigurierter Umgebung

```
{
  "Deadlocks" : {"healthy":true},
  "StorageDir": {
    "Healthy":false,
    "message":"storageDir is configured but doesn't exist!"
  }
}
```

Micro-Service produktiv stellen

Ist der Micro-Service als fettes JAR gebaut, kann er in die Zielumgebung kopiert werden, eine umgebungsspezifische Konfiguration angelegt werden und dann wie gezeigt über die Kom-

mandozeile gestartet werden. Der Micro-Service wird über Strg+C beziehungsweise SIG TERM beendet.

Das Template für die umgebungsspezifische Konfiguration kommt normalerweise aus der Entwicklung. Die konkrete Anpassung der Konfiguration (Ports, DB-Zugänge, IPs für Systems-Monitoring usw.) findet typischerweise beim Aufsetzen statt.

Jeder in DropWizard geschriebene Micro-Service bietet direkt zwei Ports an. Auf dem ersten können Anwender den Service nutzen und darüber die im Service realisierten Komponenten verwenden. Der zweite Port bietet einen Admin-Zugang, über den der Service angewiesen werden kann, Tasks – wie zum Beispiel Garbage Collection – auszuführen.

Eigene Admin-Tasks

DropWizard bietet über den Admin-Zugang alle Tasks an, die registriert sind. Eigene Tasks müssen von `io.dropwizard.servlets.tasks.Task` erben und deren `execute`-Methode überschreiben. Darüber hinaus muss ein Konstruktor definiert werden, der dem Super-Konstruktor den Namen der Task übergibt. Über diesen Namen ist die Task dann am Admin-Port verfügbar, nachdem sie in der Application-run-Methode über

```
environment.admin().addTask( new ZipLogFilesTask() );
```

registriert wurde.

Aufsetzen der Umgebung

Es bleibt nun zu klären, wie die Abhängigkeiten zwischen den einzelnen Systemen eines Gesamtsystems reproduzierbar definiert werden können. Mit dieser Fragestellung beschäftigt sich der zweite Teil dieses Beitrags, der in der kommenden Ausgabe erscheint.

Dort werden wir uns Docker anschauen. Docker ist eine junge Plattform, die sich dazu eignet, sehr leichtgewichtige Umgebungen (auf einem Linux-Container-ähnlichen Mechanismus) aufzusetzen.

Fazit

In DropWizard sind verschiedene erprobte und robuste Frameworks zur Entwicklung zusammengestellt, mit denen sich leicht eigenständige Micro-Services in Java realisieren lassen. Es fällt sehr leicht, mit DropWizard konform zu den Kriterien der 12 Factors Apps zu entwickeln. Ein Micro-Service wird tendenziell in einem alles enthaltenden ausführbaren JAR verpackt. Die Konfiguration wird beim Service-Start auf der Kommandozeile übergeben. Mehrere Instanzen einer Anwendung parallel zu starten, fällt somit leicht. DropWizard legt nahe, anwendungsspezifische HealthChecks zu realisieren, sodass die Micro-Services dazu neigen, gut überwacht werden zu können. Darüber hinaus bietet das Framework standardmäßig einen separaten Admin-Zugang, über den Verwaltungsaufgaben angestoßen werden können.

Damit ist DropWizard ein gelungenes leichtgewichtiges Paket, das wir in der kommenden Ausgabe mit Docker abrunden .

Literatur und links

[12FApp] A. Wiggins, The Twelve-Factors App, `http://12factor.net/`

[Cock14] A. Cockcroft, Migrating to Micro-Services, QCon 2014, `http://qconlondon.com/london-2014/presentation/Migrating%20to%20Microservices`

[Dropwizard] von CodaHale, `https://dropwizard.github.io/dropwizard/`

[Ghad14] Ph. Ghadir, Hystrix – Wider den Totalausfall, in: JavaSPEKTRUM, 3/2014

[Mart08] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008

[Metrics] `http://metrics.codahale.com/`

[MicroServices] `http://martinfowler.com/articles/microservices.html`

[TiGh] S. Tilkov, Ph. Ghadir zum Thema Monolithen, `http://www.infoq.com/presentations/Breaking-the-Monolith` und `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2011/Architekturen/tilkov_ghadir_OS_Architekturen_11.pdf`

[Tödt13] K. Tödter, REST in Peace: Eine Client/Server-Chat-Applikation mit Jersey, Atmosphere, JavaScript und JavaFX, in: JavaSPEKTRUM, 2/2013

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com