Badminton Tournament Platform (BTP)

Testen des Dropwizard Frameworks im Zuge meiner Projektarbeit.

Starten des Beispielprojektes

- Das GIT Projekt klonen / entpacken.
- In der Kommandozeile in das Root Verzeichnis (btp) navigieren und folgenden Maven Befehl ausführen

  ```
  mvn clean package
  ```

- Nach erfolgreichem build hat man nun ein sogenanntes *fat JAR*, welches man mit folgendem Befehl starten kann

  ```
  java -jar target/matchmaking.jar server config.yml
  ```

Der Parameter **server** wird von Dropwizard erkannt und sorgt dafür, dass die Anwendung als HTTP server läuft. Die **config.yml** ist eine optionale Konfigurationsdatei (In diesem Fall wird sie jedoch benötigt).

- Jetzt sollte die Anwendung im Browser über folgenden Link erreichbar sein:

  ```
  http://localhost:8080
  ```

Hinweise zum Beispielprojekt

Das Backend wurde mit Dropwizard erstellt, das Frontend mit AngularJS und die API Dokumentation mit Swagger. Warum ein *fat JAR*? Es hat den Vorteil, dass alle benötigten Abhängigkeiten bereits in dem JAR enthalten sind und man diese Datei einfach auf einen Server deployen kann. Das JAR wird mit Hilfe des Maven Shade Plugins konfiguriert.

Über Dropwizard

Dropwizard ist ein Java Framework für RESTful Web Services. Es erlaubt ein sehr einfaches aufsetzen dieser Services (Lediglich die Maven dependencies müssen gesetzt werden). DW ist ein leichtgewichtiges Framework, welches sehr resourcen-schonend ist. DW bietet ein gebündeltes Gesamtpaket an, was den Vorteil hat, dass vieles bereits in der richtigen Version vorliegt und man dadurch nicht alles selbst konfigurieren muss und was mit welcher Version kompatibel ist. Dabei verwendet DW unter anderem folgende Frameworks:

- Jetty als HTTP Server
- Darauf läuft Jersey, der die REST Schnittstellen einbindet
- Jackson für die JSON Transformation (Mapping von JSON Objekte auf Java Modelle um mit POJOs arbeiten zu können)
- Hibernate für die Datenbank

Erweiterungen sind unter anderem:

- Metrics (Statistiken - wie viel Zugriffe pro Sekunde?, Helthchecks - Ist die Datenbank online? Service überlastet?)
- Logging mit SLF4J und LOGBACK

Aufbau der Klassen am Beispiel Team

`Team.java`

Das POJO Object mit den getter/setter-Methoden. Falls die Datenbank nicht durch ein Migrationsskript erzeugt werden soll, dann müssen die Annotationen @Entity auf Klassenebene, bzw. @Column auf den Attributen aus dem Package javax.persistence verwendet werden. Dadurch erfolgt ein automatisiertes OR-Mapping.

`TeamResource.java`

Ist die Resource, die konsumiert wird bzw. das JSON produziert. Wichtig ist hier die Klassenannotation *@Path* um den Pfad zur Resource festzulegen. Produces/Consumes gibt an, in welchem Format die Requests/Responses vorliegen sollen (xml/JSON/text/..). HTTP requests wie *GET POST PUT DELETE* sind als Annotation an einzelne Methoden gesetzt. Zusätzlich mit dem Response Rückgabetyp erhält man entpsrechende Status Codes (201, 400, ...).

`BTPApplication.java`

Bis jetzt weiß der Server noch nicht, dass Resource Klassen existieren. Das Programm würde so nicht funktionieren. Daher müssen die Resource Klassen der Anwendung bekannt gemacht werden. Dies geschieht in der BTPApplication Klass, die von Application erbt. Hier werden alle wichtigen Services registriert und Bundles initialisiert.

- Zuerst wird ein HibernateBundle erzeugt.

- In der initialize Methode fügt man dieses Bundle dann der bootstrap Konfiguration hinzu.
- In der run Methode erzeugt man eine SessionFactory für das TeamDAO.
- Zum Schluss fügt man der Environment die TeamResource hinzu.

Die Registrierung der Resourcen kann über ein GuiceBundle auch automatisiert werden, in dem man ein Package angibt, in dem alle Resourcen hinterlegt sind (auto discovery). Sieht in etwa so aus:

```
bootstrap.addBundle(GuiceBundle.builder()
.enableAutoConfig("packagename")
.modules(new GuiceModule())
.build());
```

Hinweise

- Man kann bei Resource Klassen die Pfade kombinieren. Wenn die Klasse den Pfad *@Path("/teams")* definiert ist und in der get-Methode *@Path("/{id}")*, lautet der Endpunkt für den GET beispielsweise *localhost:8080/teams/id*.

```
@GET
@Path("{id}")
public string getTeam(@PathParam("id") String id) { ... }
```

- Query Parameter wie etwa *localhost:8080/teams?teamname=bvm* sind ebenfalls möglich. Dazu muss man die Annotation *@QueryParam* als Parameter der Methode setzen.

```
@GET
public string getTeam(
  @QueryParam("teamname") String teamname) { ... }
```

- Validierungen, wie etwa das Prüfen einer gültigen Email Adresse oder der Wertebereich eines Query Parameters werden durch Annotation abgefragt. Auch die Validierung durch eine eigene Implementierung ist mit der Annotation *@Valid* möglich. Sobald DW erkennt, dass die *@Valid* Annotation gesetzt ist, wird das Verzeichnis auf eigene Implementierungen durchsucht und entsprechend angewandt. Voraussetzung hierfür ist, dass der eigene Validator zuvor in der application environment registriert wurde.

- HATEOAS Unterstützung - DW bietet die Unterstützung sogenannter Linkings. Damit werden REST APIs maschinenlesbar gemacht. Wenn eine Resource zurückgeliefert wird, werden Meta Informationen übertragen. (Unter welchem Endpunkt finde ich die nächste/vorherige Resource). Ziel ist es einen Parse Tree aufzubauen, damit man durch Aufruf eines Links, alle anderen erhält.
- Es gibt die Möglichkeit Swagger in Dropwizard einzubauen. Swagger ist ein API Dokumentationsframework, welches mit wenigen Annotationen eine sehr ausführliche und ansprechende Dokumentation erstellt. Um Swagger in Aktion zu sehen, folgendem Link nach starten der Anwendung folgen.

  localhost:8080/api/swagger

Starten des Servers

Nach dem Bauen des Projekts mit dem Maven clean install/package Befehl wird die Anwendung über die Kommandozeile gestartet.

```
java -jar target/xxx.jar server xxx.yml
```

Wenn das Logging entsprechend eingestellt ist, sieht man beim starten direkt die Endpunkte. Hier werden zusätzlich zwei weitere Endpunkte generiert.

```
/tasks/gc
/tasks/log-level
```

Tasks/gc ist der Garbage Collector, den man bei Bedarf über die entsprechende URI von Hand anstoßen kann. Der zweite Endpunkt stellt sicher, dass man den Logging level on the fly, also während des laufenden Servers geändert werden kann.

Zusammenfassung

Zum Schluss noch ein kurzes Fazit zu Dropwizard, in dem ich offensichtliche Vor- und Nachteile anspreche und meine persönliche Erfahrung wiedergebe.

- Positiv:
  - Permanente Weiterentwicklung, dadurch werden BUGS relativ schnell gefunden und beseitigt.
  - Aktives Mitgestalten möglich, da Open Source.

  - Leicht zu erlernen, dank ausreichend Dokumentation und Google Groups.
  - HATEOAS Unterstützung.
  - Swagger Unterstützung.
  - Zusätzliche Möglichkeiten mit Authorizierung und Authentifizierung.
  - Healthchecks über admin UI (seperater Port).
  - Jede Resource kann über einen API Endpoint via Healthcheck überwacht werden.
  - Logger bereits integriert.

- Negativ
  - Dokumentation recht ausführlich aber nicht immer auf dem neusten Stand und teilweise mit Fehlern bzw. falschen Dependencies.
  - Aufbau einer Testumgebung durch zusätzliche Dependencies recht komplex und zeitintensiv.
  - Erstellen eines WAR files nicht out of the Box. (Zusätzliches Plugin - "Wizard in a Box" notwendig).
  - Wenig Dokumentation oder Beispiele für andere JPA Provider wie etwa EclipseLink.

- Probleme
  - Nach mehrmaligem starten der Anwendung, gab es merkwürdiges Verhalten der Daten und Resourcen. Der Browser Cache hatte hier noch Daten gehalten. Am besten immer im Inkognito Modus starten.
  - Einbinden des Swagger Bundles recht einfach. Jedoch gab es hier enorme Probleme mit dem Abgleich der verschiedenen Module und Maven Plugins. Nach einigen Exclusions und Einbinden neuer Plugins lief Swagger endlich.
  - Erstellen des Frontends mit AngularJS hat viel zu lange gedauert. Z.b. Routing Probleme mit neuer Version alt: #/about, neu: !#/about
  - Datenbank zuerst mit Liquibase und dem Migration Befehl erstellt.

- TODO
  - HATEOAS einbauen.
  - Testklassen vervollständigen.
  - Health Checks vervollständigen.
  - m:n Beziehung zwischen Team und Game verbessern und ausbauen.
  - Eclipselink statt Hibernate als JPA Provider.

- Fazit

Anfangs war ich nicht wirklich überzeugt von Dropwizard, mitlerweile arbeitet es sich sehr gut damit. Da ich bisher nur mit DW zu tun hatte und mir der Vergleich zu anderen REST Frameworks fehlt kann ich hier keine direkten Vergleiche ziehen. Je länger man mit DW arbeitet, desto mehr fasziniert es einen.

Man entdeckt immer mehr Features und Funktionalitäten, die sehr nützlich sind (Angefangen vom Healthchecks/ Logging, über eigene Validatoren bis hin zu Swagger oder HATEOAS). Daher kann ich jedem nur empfehlen Dropwizard als REST Framework zumindest einmal anzutesten und sich von seinen stärken zu überzeugen.