Workshop:
Spring REST + Testing
Übungen

Ablauf

Dieser Workshop gliedert sich in mehrere Vortragsteile, die den Teilnehmern die Thematik Spring Framework und Spring Boot mit Spring Data usw. näherbringen. Im Anschluss daran sind jeweils einige Übungsaufgaben von den Teilnehmern – idealerweise in Gruppenarbeit – am Rechner zu lösen.

Voraussetzungen

1) Aktuelles JDK 11, idealerweise auch JDK 17, installiert
2) Aktuelles Eclipse installiert (Alternativ: Spring Tool Suite oder IntelliJ IDEA)

Teilnehmer

- Entwickler und Architekten mit Java-Erfahrung, die ihre Kenntnisse zu Spring, Spring Boot und Data vertiefen möchten

Kursleitung und Kontakt

Michael Inden
Derzeit freiberuflicher Buchautor und Trainer
E-Mail: michael_inden@hotmail.com
Blog: https://jaxenter.de/author/minden

Weitere Kurse (Java, Unit Testing, ...) biete ich gerne auf Anfrage als Inhouse-Schulung an.

TAG 3: Spring REST + Testing

Aufgabe 1: Spring REST Intro 15 min

Vollziehe das Beispiel unter **https://spring.io/guides/gs/rest-service/** nach.

Aufgabe 2: Product Client 25 min

In der Präsentation haben wir einen Product-Server gesehen. Nun gilt es einen dazugehörigen REST Client zu entwickeln, um Produkte abzufragen und zu erzeugen:

- Erstelle Abfragen als SpringBootApp und als StandAloneApplikation
- Ergänze in der ursprünglichen Applikation noch Validierung

Aufgabe 3: Highscore REST Server 45 min

Erstelle einen Highscore-Server zur Verwaltung und Abfrage von Highscore-Daten. Speichere die Highscores in einer In-Memory-DB. Stelle etwa folgende Endpoints bereit:

- **POST „Highscore“**
- **GET /highscores**
- **GET /highscores/top/<n>**
- GET /highscores/byPerson/<name>
- GET /highscores/byLocalDate/<localdate>
- DELETE /highscores/<n>
- **DELETE /highscores/of/<name>**
- DELETE /highscores/on/< localdate >

Aufgabe 4: Erstelle einen User REST Server 45 min

Gegeben sei ein REST-Server zur Verwaltung und Abfrage von User-Daten, der ausschließlich mit GET arbeitet. Die User werden in einer In-Memory-DB gespeichert und durch folgende Endpoints bereitgestellt:

- `/create?email=[email]&name=[name]: create a new user with an auto-generated id and email and name as passed values.`
- `/delete?id=[id]: delete the user with the passed id.`
- `/get-by-email?email=[email]: retrieve the id for the user with the given email address.`
- `/update?id=[id]&email=[email]&name=[name]: update the email and the name for the user identified by the given id.`

Verbessere das Applikationsdesign durch

a) Nutzung von passsenden Status-Codes sowie HTTP-Verben
b) Integration eines Service
c) sowie durch Einsatz von Error Handling
d) Validierung

Prüfe die Validierung und die anderen Aktionen mit CURL, etwa

```
curl –X POST –H "Content–Type: application/json"
http://localhost:8080/users/ –d '{ "email" : "E–MAIL", "name" :
"Micha" }'
```

Und mit einer gültigen E-Mail:

```
curl –X POST –H "Content–Type: application/json"
http://localhost:8080/users/ –d '{ "email" : "mail@mike.de", "name"
: "Micha" }'
```

Aufgabe 5: Zodiac / Sternzeichen & Architektur 100 min

In dieser Aufgabe wird sowohl die Architektur als auch das Testing geschult. Einige Ideen zum REST Controller sind https://developer.okta.com/blog/2019/03/28/test-java-spring-boot-junit5 entnommen, aber an vielen Stellen speziell angepasst, erweitert und verändert. Das Ganze findet sich als Projekt »**ex05-spring-boot-testing-template**».

Aufgabe 5a: Prüfe die Applikation mit folgenden Aufrufen

Lerne durch einige REST Calls das nach außen sichtbare Interface kennen.

```
curl -X POST \
  http://localhost:8080/birthday/dayOfWeek \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 1971-02-07

curl -X POST \
  http://localhost:8080/birthday/chineseZodiac \
  -H 'Content-Type: text/plain' \
  -H 'accept: text/plain' \
  -d 1971-02-07
```

Aufgabe 5b (Step1): Schreibe passende Integrationstests! Wieso ist das hilfreich? Denke an spätere interne Umbaumaßnahmen! 25 min

Nutze die Klasse `BirthdayInfoControllerIT` als Ausgangsbasis. Nutze zur Vereinfachung `@ParameterizedTest` in einer Klasse `BirthdayInfoControllerIT2`.

Aufgabe 5c (Step2): Analysiere die Architektur und das Design. Benenne Schwachstellen und verbessere das Ganze. Extrahiere dazu einen passenden Service sowie ein dazugehöriges Interface: 25 min

```
public interface BirthdayService
{
      String getBirthDOW(LocalDate birthday);
      String getChineseZodiac(LocalDate birthday);
      String getStarSign(LocalDate birthday);
}
```

Korrigiere die Integration Tests und dabei insbesondere ggf. die geladenen Klassen mithilfe von

```
@ContextConfiguration(classes = { BirthdayInfoController.class,
                                  BasicBirthdayService.class })
```

Aufgabe 5d (Step3): Analyse: Welche Probleme könnten auftreten, wenn der erstellte Service auf andere Services oder ein Repository zugreift? Wie kann man die Berechnung verbessern und noch einfacher testbar machen? 50 min

Erstelle eine weitere Klasse `ImprovedBirthdayService`, die verbesserte, elegantere Varianten der Berechnungen (etwa Lookup-Maps/Arrays, Hilfsmethoden oder sogar geschickte Lambda-Abfragen) enthält. Was muss dann für die Hauptapplikation beim Start festgelegt werden? Womit kann man das erreichen? Sollte man die Berechnungen in eine Util-Klasse auslagern?

Aufgabe 6: Zodiac / Sternzeichen Client 45 min

In ersten Teil von Aufgabe 1 wurden CURL-Aufrufe zum Prüfen der Implementierung genutzt. Oftmals werden Aufrufe von anderen Systemen automatisiert in Form von Clients ausgeführt werden, die etwa einen REST Call mithilfe von Spring's `RestTemplate` absetzen.

In dieser Aufgabe versetzen wir uns in die Lage desjenigen Entwicklers, der einen solchen Client geschrieben hat und diesen nun testen möchte. Wie kann man dazu vorgehen? Was ist ein initiales Problem dabei? Schaue dazu nochmal auf folgende Implementierung:

```
@Service
public class ZodiacClient {

      private final RestTemplate restTemplate;

      @Autowired
      public ZodiacClient(RestTemplateBuilder restTemplateBuilder) {
            restTemplate = restTemplateBuilder.build();
      }

      public String getZodiac(LocalDate localdate) {
            return restTemplate.getForObject("http://localhost:8080/"
                        + "birthday/chineseZodiac?birthdayString="
                        + localdate, String.class);
      }
}
```

Aufgabe 6a: Bringe die Applikation zum Laufen, parallel zum Server.

Aufgabe 6b: Schreibe einen Test für die obige Klasse, die eine Abfrage beim Server lanciert.

Aufgabe 6c: Schreibe einen Test, der unabhängig vom laufenden Server ist. Erinnere dich an `@RestClientTest`.

Aufgabe 6d: Verallgemeinere die akzeptierten Aufrufe (mithilfe von `Matchers.matchesPattern`). Erweitere den `ZodiacClient` um einen POST-Aufruf für «normale» Sternzeichen und liefere immer «DRAGON» als Ergebnis.

Das Grundgerüst findet sich im Projekt »**ex06-spring-boot-rest-client-template**».

Aufgabe 7: Wandle Integration Test in Unit Test 30 min

Gegeben sei die Klasse `OrderService` sowie ein passender Integrationstest. Wandle diesen schrittweise in einen Unit Test um.

Aufgabe 7a: Wie kann man statt gegen die DB mit Mocks arbeiten? Verwende `@MockBean` und eine passende Parametrierung.

Aufgabe 7b: Nutze nun einen reinen Unit Test. Gestalte die Klasse in eine neue Klasse `OrderServiceImproved` um, damit dies möglich wird!