JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

Web Services in Java

Systemtechnik Labor
5BHITT 2015/16, Gruppe X

Philipp Adler

Version 1.0

Note:

Betreuer: Michael Borko

Begonnen am 12. Februar 2016

Beendet am 18. Februar 2016

Inhaltsverzeichnis

1 Einführung

Diese Übung zeigt die Anwendung von mobilen Diensten in Java.

1.1 Ziele

Das Ziel dieser Übung ist eine Webanbindung zur Benutzeranmeldung in Java umzusetzen. Dabei soll sich ein Benutzer registrieren und am System anmelden können.

Die Kommunikation zwischen Client und Service soll mit Hilfe von JAX-RS (Gruppe1) umgesetzt werden.

1.2 Voraussetzungen

- Grundlagen Java und Java EE
- Verständnis über relationale Datenbanken und dessen Anbindung mittels JDBC oder ORM-Frameworks
- Verständnis von Restful Webservices

1.3 Aufgabenstellung

Es ist ein Webservice mit Java zu implementieren, welches eine einfache Benutzerverwaltung implementiert. Dabei soll die Webapplikation mit den Endpunkten /register und /login erreichbar sein.

Registrierung
Diese soll mit einem Namen, einer eMail-Adresse als BenutzerID und einem Passwort erfolgen. Dabei soll noch auf keine besonderen Sicherheitsmerkmale Wert gelegt werden. Bei einer erfolgreichen Registrierung (alle Elemente entsprechend eingegeben) wird der Benutzer in eine Datebanktabelle abgelegt.

Login
Der Benutzer soll sich mit seiner ID und seinem Passwort entsprechend authentifizieren können. Bei einem erfolgreichen Login soll eine einfache Willkommensnachricht angezeigt werden.

Die erfolgreiche Implementierung soll mit entsprechenden Testfällen dokumentiert werden. Es muss noch keine grafische Oberfläche implementiert werden! Verwenden Sie auf jeden Fall ein gängiges Build-Management-Tool..

2 Ergebnisse

2.1 Datenmodell

Der User besteht wie in der Aufgabenstellung beschrieben aus 3 Attributen. Das ist der Name, die Email und das Passwort des Users. Da die Email-Adresse weltweit nur einmal vergeben ist und auch als BenutzerID. welche für das Anmelden benötigt wird, dienen soll, stellt es unseren Primary Key des Modells dar.

Die Registrierung und der Login der Applikation werden durch verschiedene Endpoints getrennt.

2.2 Registrierung

In File *UserRegisterEndpoint* werden alle notwendigen Methoden für die Registrierung implementiert.

```
@Named
@Path("/register")
@Produces({MediaType.APPLICATION_JSON})
public class UserRegisterEndpoint {

   @Inject
   private UserService userService;

   @POST
   public Response post(User user) {
      if(userService.findByEmail(user.getEmail()) != null){
         if(!(user.getEmail().isEmpty() || user.getName().isEmpty() || user.getPassword().isEmpty())) {
            this.userService.createUser(user);
            return Response.status(Response.Status.CREATED).entity("Erfolgreich registriert").build();
         }else{
            return Response.status(Response.Status.NOT_ACCEPTABLE).entity("Bitte füllen Sie alle Felder
aus").build();
         }
      }else{
         return Response.status(Response.Status.FORBIDDEN).entity("Der User existiert bereits").build();
      }
   }

}
```

Code 1 Registrierung

Die Methode *post* ist unter /register erreichbar. Definiert wird dies durch die @Path Angabe. An die *post* Methode wird ein User übergeben. Zuerst wird überprüft ob sich der User bereits in der Datenbank registriert hat. Falls der User nicht existiert, überprüfe ich sicherheitshalber, ob alle Attribute befüllt sind. Wenn alles passt, liefert die Applikation ein Response, indem dem User mitgeteilt wird, dass sich dieser erfolgreich registriert hat. Im Falle, dass es vorher zu anderen Problemen kam, wird ihm mitgeteilt, dass ein Fehler aufgetreten ist.

2.3 Login

In File *UserLoginEndpoint* werden alle notwendigen Methoden für den Login implementiert.

```
@Named
@Path("/login")
@Produces({MediaType.APPLICATION_JSON})
public class UserLoginEndpoint {

   @Inject
   private UserService greetingService;

   @POST
   public Response post(User user) {
      User loginuser = this.greetingService.findByEmail(user.getEmail());
      try{
         if(loginuser != null) {
            if (loginuser.getPassword().equals(user.getPassword())) {
               return Response.status(Response.Status.OK).entity("Herzlich Willkommen: " + user.getEmail()).build();
            }
         }
      }catch(NullPointerException e){
      }
      return Response.status(Response.Status.FORBIDDEN).entity("Falsche Anmeldedaten").build();
   }

}
```

Code 2 Login

Ähnlich wie oben ist die *post* Methode unter /login, definiert im @Path, erreichbar. Auch hier wird überprüft, ob der User bereits in der DB gespeichert ist. Danach wird das Passwort des Users von der DB, mit der Benutzereingabe verglichen. Bei Übereinstimmungen wird eine einfache Willkommensnachricht angezeigt.

2.4 Endpoints

Wie oben bereits erwähnt hat die Registrierung und der Login jeweils einen eigenen Endpoint. Diese werden im JerseyConfig.java definiert.

```
@Named
public class JerseyConfig extends ResourceConfig {

   public JerseyConfig() {
      this.register(UserRegisterEndpoint.class);
      this.register(UserLoginEndpoint.class);
      this.register(JacksonFeature.class);
   }
}
```

Code 3 Deklaration der Endpoints

2.5 Build-Managment-Tool

Als Build-Managment-Tool entschied ich mich für Maven. Maven wir mit einem POM(project onject model) File gemanagt. Da bei einem .pom File der wichtigste Teil die Dependencies sind, habe ich einen Codeausschnitt vorbereitet, welcher meine Wahl darstellt.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jersey</artifactId>
  </dependency>
  <dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>${jersey.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
      <exclusion>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

Code 4 Maven Dependencies

2.6 VM Connection

Auf dem Github-Repository https://github.com/padler-tgm/WebServices findet man unter mysql ein Vagrant-File, welches mit dem Befehlt vagrant up die MySQL-DB startet. Damit sich der Webservice auf die Datenbank verbinden kann müssen unter ressources/application.properties alle notwendigen Informationen hinterlegt werden.

```
spring.datasource.url=jdbc:mysql://10.0.1.20:3306/Webservice
spring.datasource.username=web
spring.datasource.password=web
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create-drop
```

Code 5 Connection data for vagrant vm

2.7 Testfälle

Registrierung

User Registrierung, wo alle Felder ausgefüllt sind

Request

POST /register Accept: application/json

Content-Type: application/json

{“name“:“Philipp“, “email“:“padler@student.tgm.ac.at“, “password“:“1234“}

Response

HTTP/1.1 201 Created

Content-Type: application/json

User Registrierung, wo der Name des Users nicht angegeben wurde

Request

POST /register Accept: application/json

Content-Type: application/json

{“name“:“ “, “email“:“padler@student.tgm.ac.at“, “password“:“1234“}

Response

HTTP/1.1 406 Not Acceptable

Content-Type: application/json

User Registrierung, wo der User bereits registriert ist

Request

POST /register Accept: application/json

Content-Type: application/json

{“name“:“Philipp“, “email“:“padler@student.tgm.ac.at“, “password“:“1234“}

Response

HTTP/1.1 403 Forbidden

Content-Type: application/json

Login

User Login, wo alle Felder ausgefüllt sind und richtige Userdaten

Request

POST /login Accept: application/json

Content-Type: application/json

{“email“:“padler@student.tgm.ac.at“, “password“:“1234“}

Response

HTTP/1.1 200 OK

Content-Type: application/json

User Login, wo der die email des Users nicht angegeben ist

Request

POST /login Accept: application/json

Content-Type: application/json

{“email“:“ “, “password“:“1234“}

Response

HTTP/1.1 403 Forbidden

Content-Type: application/json

User Login, wo das Passwort nicht zur Email-Adresse passt

Request

POST /login Accept: application/json

Content-Type: application/json

{“email“:“padler@student.tgm.ac.at“, “password“:“5678“}

Response

HTTP/1.1 403 Forbidden

Content-Type: application/json

[2]

2.8 Zeitaufwand

Web Services	Datum	geschätzter Aufwand	Zeitaufwand
Research & Einlesen	12.02.16	20min	60min
Build-Management-Tool	12.02.16	15min	70min
Datenmodell & Datenbank	12.02.16	30min	50min
Registrierung	12.02.16	30min	30min
Login	12.02.16	30min	30min
Testfälle	12.02.16	30min	15min
Summe		155min	255min

2.9 Probleme

Das Build-Management-Tool Maven importierte die Dependencies nicht automatisch. Nach löschen des .m2/ im Home-Verzeichnis und erneuten erstellen des Projekts und dank der Hilfe von Paul Kalauner, ließen sich die Dependencies schlussendlich importieren. Das hat mich 1/3 meiner Arbeitszeit gekostet. Vielleicht lag es auch daran, dass wir Maven nicht in der 4.Klasse durchgenommen haben.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Josh Long, (November 2014) „Bootiful“ Java EE Support in Spring Boot 1.2. Abgerufen am 12.Februar 2016 von http://spring.io/blog/2014/11/23/bootiful-java-ee-support-in-spring-boot-1-2

[2] tutorialspoint. Acceptance Testing, Abgerufen am 18.Februar von http://www.tutorialspoint.com/software_testing_dictionary/acceptance_testing.htm