
Laborprotokoll

Cloud-Datenmanagement

Systemtechnik Labor
5BHIT 2017/18

Martin Wölfer

Note:
Betreuer: Michael Borko

Version 2.7
Begonnen am 16. November 2017
Beendet am 23. November 2017

Inhaltsverzeichnis

1	Einführung	1
1.1	Ziele	1
1.2	Voraussetzungen	1
1.3	Aufgabenstellung	1
2	Ergebnisse	2
2.1	Allgemein	2
2.1.1	Hibernate	2
2.1.2	Spring	2
2.1.3	Beans	2
2.1.4	In-Memory-Datenbank	3
2.1.5	@Autowired Annotation	3
2.2	Voraussetzungen	3
2.3	Verwendung	3
2.4	Implementation	4
2.4.1	Dependencies	4
2.4.2	Entities	4
2.4.3	Repositories	5
2.4.4	UserDetailsService	5
2.4.5	Security Service	5
2.4.6	UserService	6
2.4.7	Validator	7
2.4.8	Controller	7
2.5	Tests	8
2.5.1	Unittests	8
2.5.2	Regressiontests	9

1 Einführung

Diese Übung zeigt die Anwendung von mobilen Diensten.

1.1 Ziele

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

Die Kommunikation zwischen Client und Service soll mit Hilfe einer REST Schnittstelle umgesetzt werden.

1.2 Voraussetzungen

- Grundlagen einer höheren Programmiersprache
- Verständnis über relationale Datenbanken und dessen Anbindung mittels ODBC oder ORM-Frameworks
- Verständnis von Restful Webservices

1.3 Aufgabenstellung

Es ist ein Webservice 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 Datenbanktabelle 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 (Acceptance-Tests bez. aller funktionaler Anforderungen mittels Unit-Tests) dokumentiert werden. Verwenden Sie auf jeden Fall ein gängiges Build-Management-Tool (z.B. Maven oder Gradle). Dabei ist zu beachten, dass ein einfaches Deployment möglich ist (auch Datenbank mit z.B. file-based DBMS).

2 Ergebnisse

Da der erste Versuch nicht funktionierte, wurde auf eine Realisierung in JavaEE gesetzt mit Unterstützung von **Spring Boot**, **Spring Security**, **Spring Data JPA** und der Datenbank **HSQL**. Es wurde mit folgendem Tutorial gearbeitet:

<https://hellokoding.com/registration-and-login-example-with-spring-security-spring-boot-spring-data-jpa-hsql-jsp/>

Das Endergebnis ist ein RESTful Webservice bei welchem man sich einloggen und registrieren kann und mit einer Willkommensnachricht begrüßt wird.

2.1 Allgemein

Zuerst mussten gewisse Themengebieten recherchiert werden. Vor allem das Themengebiet **Hibernate** musste von Grund auf gelernt und verstanden werden. Folgend werden einige Themen erklärt beziehungsweise Begriffe definiert.

2.1.1 Hibernate

src: <https://howtoprogramwithjava.com/hibernate-persistence-beginners/>

Hibernate "steht" zwischen objektorientiertem Java und einem **Relational DataBase Management System**.

Grundsätzlich dient Hibernate dazu, Java Objekte zu *persistieren*, also diese "permanent" erhältlich zu machen.

2.1.2 Spring

src: <https://howtoprogramwithjava.com/podcast-episode-33-intro-to-spring-framework/>

Hibernate wird sehr oft in Verbindung mit **Spring** verwendet, Spring kümmert sich um die Kernfunktion einer Rest-Applikation, und zwar dem **Controller**. Mit Spring können auf bestimmte Links oder Link-patterns Funktionen **gemapped** werden, welche wiederum beispielsweise eine **.jsp** page aufrufen.

2.1.3 Beans

Zwar ist dieses Thema Grundwissen von JavaEE, trotzdem hatte ich große Probleme Erklärungen zu verstehen, da ich nicht wusste im Kontext von Java was eine **Bean** ist.

Eine Bean ist lediglich ein Standard, welcher 3 folgende Eigenschaften vorschreibt:

1. Alle Attribute sind **private** (nur Getter/Setter)
2. Ein **public** Konstruktor ohne Parameter
3. Muss **Serializable** implementieren

Serializable: Beschreibt die Eigenschaft dass das Objekt in ein String umgewandelt werden kann

2.1.4 In-Memory-Datenbank

src: <https://de.wikipedia.org/wiki/In-Memory-Datenbank>
Da in dem Beispiel mit HSQL gearbeitet, einer In-Memory-Datenbank, muss verstanden werden wie diese funktioniert, bzw. was die wichtigen Eigenschaften.

Der wichtigste Unterschied ist, dass nicht wie bei einem herkömmlichen DBMS die Datenbanken auf der Festplatte gespeichert werden, sondern im RAM. Dies führt dazu, dass wenn der Server neu gestartet wird, also die Datenbank auch neu geladen wird, alle persistierten Daten verloren gehen.

2.1.5 @Autowired Annotation

src: <https://stackoverflow.com/questions/19414734/understanding-spring-autowired-usage>
<https://stackoverflow.com/questions/19414734/understanding-spring-autowired-usage>

Dies war die größte Verwirrung welche aufgetreten ist. Diese Annotation kommt sehr oft vor in dem Beispielprojekt, und zwar meistens vor einem Attribut:

```
1 @Service
  public class UserDetailsServiceImpl implements UserDetailsService{
3     @Autowired
    private UserRepository userRepository;
```

Wenn `UserDetailsServiceImpl` gestartet wird, wird das Attribut `userRepository` vom Typ `UserRepository` definiert, und mit der `@Autowired` Annotation versehen. Dies bedeutet, dass nach der `Bean` `UserRepository` gesucht wird, und diese dann anschließend in dieses Attribut "injected" wird. Ich habe noch immer nicht ganz verstanden was "injecten" bedeutet in dem Kontext.

2.2 Voraussetzungen

Um das Projekt ausführen zu können, müssen JDK 1.7(oder neuer) und Maven 3(oder neuer) vorhanden sein.

Um den Server starten zu können in der CLI, muss maven der **PATH** Variable hinzugefügt sein.

2.3 Verwendung

Der Server ist zu starten mit `mvn spring-boot:run`

Tests sind auszuführen mit `mvn test`

2.4 Implementation

2.4.1 Dependencies

Die dependencies, d.h. welche Framework verwendet wird. Normalerweise würde das bedeuten die jeweiligen .jar Files herunterzuladen und dem Build-Path hinzuzufügen, aber dank Maven kann man dies ganz einfach im `pom.xml` File definieren.

2.4.2 Entities

Mit `hibernate` wird eine Tabelle durch die annotation `@Entity` definiert. Es wird eine Klasse `User` erstellt, welche folgende Attribute besitzt:

- Long id
- String username
- String password
- String passwordConfirm
- Set<Role> roles

Danach werden Getter- und Settermethoden definiert. Zu beachten ist, dass id den eindeutigen Primary Key repräsentiert und somit bei `getId()` die Annotations `@Id` und `@GeneratedValue(strategy = GenerationType.AUTO)` benötigt werden.

```
1 package com.hellokoding.auth.model;
2
3 import javax.persistence.*;
4 import java.util.Set;
5
6 @Entity
7 @Table(name = "user")
8 public class User {
9     private Long id;
10    private String username;
11    private String password;
12    private String passwordConfirm;
13    private Set<Role> roles;
14
15    @Id
16    @GeneratedValue(strategy = GenerationType.AUTO)
17    public Long getId() {
18        return id;
19    }
20
21    ....
22 }
```

Klassen, welche mit `hibernate` in Datenbanken gemapped werden, werden sehr oft im sogenannten `Plain Old Java` geschrieben. Dies bedeutet, dass lediglich Attribute, Konstruktor und getter und setter Methoden definiert werden.

2.4.3 Repositories

Es werden 2 Repositories definiert, eines für die **User** und eines für die **Roles**.

Normalerweise werden in den Repositories Funktionen wie **findOne**, **findAll**, **save** implementiert, allerdings ist es möglich mit Spring vom **JpaRepository** zu erben, welches die Grundfunktionalitäten bereits implementiert hat.

Es wird lediglich folgende die Funktion **findByUsername** definiert:

```
1 public interface UserRepository extends JpaRepository<User, Long> {  
    User findByUsername(String username);  
3 }
```

Das Rolerepository benötigt keine zusätzlichen Definition, da die Grundimplementationen ausreichen:

```
1 public interface RoleRepository extends JpaRepository<Role, Long>{  
}
```

2.4.4 UserDetailsService

Um Spring Security zu verwenden, wird

org.springframework.security.core.userdetails.UserDetailsService implementiert:

```
1 @Service  
public class UserDetailsServiceImpl implements UserDetailsService{  
3 ....
```

In der Implementation von **UserDetailsService** wird eine Funktion **loadUserByUsername** definiert. Diese Funktion, lädt den User mit der Funktion des Userrepositorys (**findByUsername**), lädt die Authorities aus den Rollen, welche auch im User Objekt gespeichert sind.

Es wird anschließend ein **org.springframework.security.core.userdetails.User** Objekt returned, mit **Username**, **Password** und **Authorities**

```
1 @Override  
@Transactional(readOnly = true)  
3 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
    User user = userRepository.findByUsername(username);  
  
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();  
    7 for (Role role : user.getRoles()) {  
        grantedAuthorities.add(new SimpleGrantedAuthority(role.getName()));  
    9 }  
  
    11 return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword()  
        , grantedAuthorities);  
}
```

2.4.5 Security Service

Der nächste Schritt ist des den Security Service zu implementieren, dieser Service kümmert sich darum, dass der Name des momentan eingeloggten User bekannt gegeben werden kann und man sich einloggen kann.

Um den momentan eingeloggten User zu finden, werden aus dem `SecurityContext` die Authentication-Details ausgelesen:

```
Object userDetails = SecurityContextHolder.getContext().getAuthentication().getDetails();
```

Falls diese Variable nun tatsächlich der Klasse `UserDetails` entspricht, bedeutet dies das tatsächlich ein User eingeloggt ist und sein Username wird returned, andernfalls wird null returned:

```
1  if (userDetails instanceof UserDetails) {
3      return ((UserDetails) userDetails).getUsername();
5
    return null;
```

Die Funktion `autologin` wird mit 2 Parametern aufgerufen, `username` und `password`.

Mit dem Username wird wieder aus dem `UserRepository` der User geladen, anschließend wird damit ein

`UsernamePasswordAuthenticationToken` erstellt:

```
1  UserDetails userDetails = userDetailsService.loadUserByUsername(username);
    UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new
        UsernamePasswordAuthenticationToken(userDetails, password, userDetails.getAuthorities());
```

Als nächstes wird mit dem `AuthenticationManager` und dem token authentifiziert mit der Funktion `authenticate`. Es wird mit `isAuthenticated` überprüft, ob die Anmeldung funktioniert hat, wird der `SecurityContext` gesetzt und eine Log-Message ausgegeben:

```
authenticationManager.authenticate(usernamePasswordAuthenticationToken);
2
if (usernamePasswordAuthenticationToken.isAuthenticated()) {
4    SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
    logger.debug(String.format("Auto login %s successfully!", username));
6 }
```

2.4.6 UserService

Ein weiterer wichtiger Dienst ist der `UserService`. Dieser kümmert sich darum, dass die Methoden vom `UserRepository` korrekt aufgerufen werden.

Es wird eine `save` Methode implementiert, welche bevor sie die `save` Methode vom Repository aufruft noch das Passwort **verschlüsselt** und die Rollen setzt.

Bei `findByUsername` wird lediglich die gleichnamige Funktion des repositories aufgerufen:

```
@Override
2 public void save(User user) {
    user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
4    user.setRoles(new HashSet<>(roleRepository.findAll()));
    userRepository.save(user);
6 }

8 @Override
public User findByUsername(String username) {
10     return userRepository.findByUsername(username);
}
```


2.4.7 Validator

Diese Klasse kümmert sich bei der Registrierung darum, dass passende Werte für Passwort und Username angegeben werden.

Zuerst wird die `supports` Methode überschrieben, um sicherzustellen das `User` verglichen werden und nicht andere Objekte:

```
1 @Override
2 public boolean supports(Class<?> aClass) {
3     return User.class.equals(aClass);
4 }
```

Anschließend wird die `Validate` Funktion überschrieben, in welcher alle Restriktionen definiert werden, der Code ist selbsterklärend:

```
1 @Override
2 public void validate(Object o, Errors errors) {
3     User user = (User) o;
4
5     ValidationUtils.rejectIfEmptyOrWhitespace(errors, "username", "NotEmpty");
6     if (user.getUsername().length() < 6 || user.getUsername().length() > 32) {
7         errors.rejectValue("username", "Size.userForm.username");
8     }
9     if (userService.findByUsername(user.getUsername()) != null) {
10        errors.rejectValue("username", "Duplicate.userForm.username");
11    }
12
13    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password", "NotEmpty");
14    if (user.getPassword().length() < 8 || user.getPassword().length() > 32) {
15        errors.rejectValue("password", "Size.userForm.password");
16    }
17
18    if (!user.getPasswordConfirm().equals(user.getPassword())) {
19        errors.rejectValue("passwordConfirm", "Diff.userForm.passwordConfirm");
20    }
21 }
```

2.4.8 Controller

Es muss auch ein Controller definiert werden, welcher sich um das `mappen` der Links kümmert.

Bei `/registration` wird zuerst die `validate` Funktion mit dem `user` objekt und dem Fehler-Objekt als Parameter aufgerufen. Falls Fehler vorhanden sind, wird lediglich die gleiche Seite zurückgegeben, d.h. neu geladen. Wenn allerdings alle Angaben richtig angegeben wurden, wird der User mit dem Dienst `UserService` abgespeichert in der Datenbank, es wird ein login durchgeführt und zur Willkommensseite weitergeleitet.

```
1 @RequestMapping(value = "/registration", method = RequestMethod.POST)
2 public String registration(@ModelAttribute("userForm") User userForm, BindingResult bindingResult, Model
3     model) {
4     validator.validate(userForm, bindingResult);
5
6     if (bindingResult.hasErrors()) {
7         return "registration";
8     }
9
10    userService.save(userForm);
11
12    securityService.autologin(userForm.getUsername(), userForm.getPasswordConfirm());
13
14    return "redirect:/welcome";
15 }
```

Bei **/login** wird zuerst überprüft, ob Fehler vorliegen, falls nicht wird noch überprüft ob der User sich ausgeloggt hat, und falls dies auch nicht der Fall ist wird zur `login.jsp` Seite weitergeleitet, wo man sich mit dem User anmelden kann.

```
1  @RequestMapping(value = "/login", method = RequestMethod.GET)
2  public String login(Model model, String error, String logout) {
3      if (error != null)
4          model.addAttribute("error", "Your username and password is invalid.");
5
6      if (logout != null)
7          model.addAttribute("message", "You have been logged out successfully.");
8
9      return "login";
10 }
```

Zum Schluss wird noch definiert was bei **/welcome** passiert, wo einfach nur auf die `welcome.jsp` Seite weitergeleitet wird, welche sich um das Rendering der Daten kümmert.

2.5 Tests

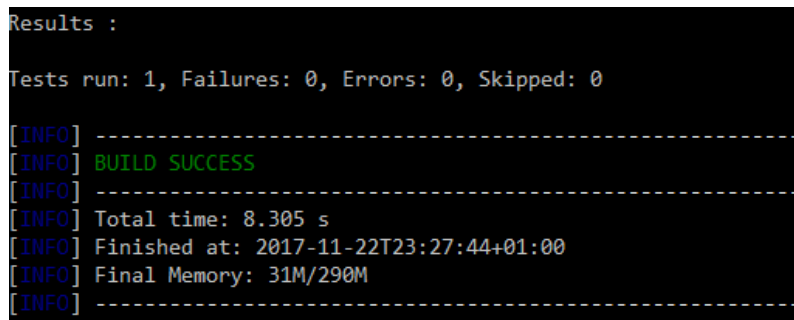
2.5.1 Unittests

Die Testklasse `UserTest` ermöglicht Unit-Tests laufen zu lassen. Beispielsweise gibt es folgenden Test:

```
1  @Test
2  public void saveAndFindUser() {
3      // given
4      User u = new User();
5
6      u.setUsername("mwoelfer01");
7      u.setPassword("12345678");
8
9      userService.save(u);
10
11     // when
12     User found = userRepository.findByUsername(u.getUsername());
13
14     // then
15     assertThat(found.getUsername())
16         .isEqualTo(u.getUsername());
17 }
```

Dieser Test legt einen neuen User an, speichert diesen ab, liest ihn anschließend aus der Datenbank und überprüft die Daten die aus der DB ausgelesen wurden mit denen die angelegt wurden.

Wenn man den Test mit `mvn test` ausführt, erhält man folgendes Ergebnis:



```
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 8.305 s
[INFO] Finished at: 2017-11-22T23:27:44+01:00
[INFO] Final Memory: 31M/290M
[INFO] -----
```

Abbildung 1: Unittest wurde erfolgreich ausgeführt

2.5.2 Regressionstests

Die Regressionstests wurden mit **Postman** durchgeführt. Mit Postman können **POST** und **GET** requests gesendet und getestet werden.

Das erste Problem auf welches gestoßen wurde war, dass kein **CSRF-Token** vorhanden war beim Versuch ein Post-Request zu senden. Grund ist, dass dieser token nur mitgesendet wird, wenn die Seite mit dem Browser aufgerufen wird. Um dieses Problem zu umgehen, wird **CSRF** in der **WebSecurityConfig** disabled:

```
1 .and()
   .csrf().disable();
```

Nun konnte eine leere request gesendet werden ohne Fehlermeldungen.

Es wurde eine POST-request erstellt, auf der url **localhost:8080/registration**, welches ein Form mit **username**, **password** und **passwordConfirm** mitsendet, um einen User zu erstellen.

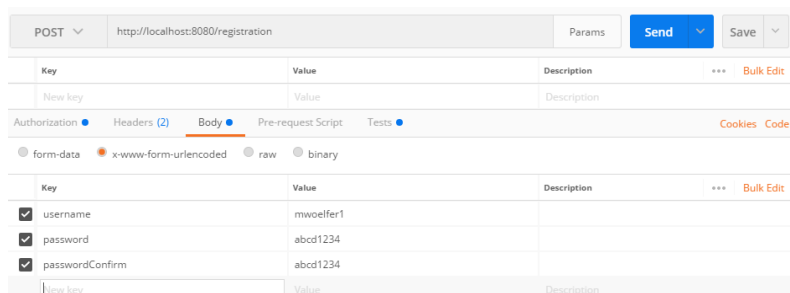


Abbildung 2: Es wird ein Form mitgesendet mit Daten

Es können anschließend Tests geschrieben werden, in dem Fall wird überprüft ob auf die welcome Page umgeleitet wurde:

```
pm.test("Status code is 200", function () {
2   pm.response.to.have.status(200);
});

4
pm.test("Body matches string", function () {
6   pm.expect(pm.response.text()).to.include("Welcome mwolfer");
});
```

Und bei der Ausführung sollten die Tests funktionieren:

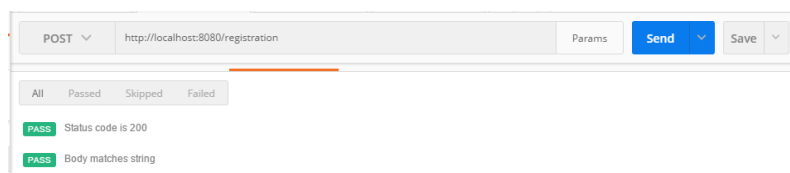


Abbildung 3: Registrations-Test wurde erfolgreich ausgeführt

Es wurde ein weiterer Test für login geschrieben, welcher test ob auf die welcome-page weitergeleitet wird wenn ein Form mit **user** und **password** mitgesendet wird:

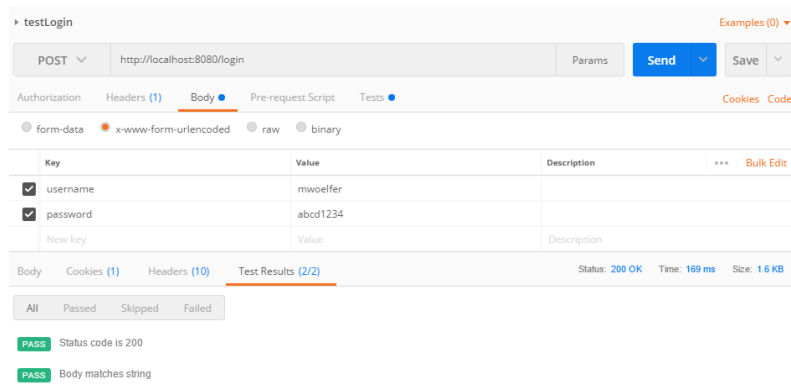


Abbildung 4: Login mit user und password funktioniert

Abbildungsverzeichnis

1	Unittest wurde erfolgreich ausgeführt	8
2	Es wird ein Form mitgesendet mit Daten	9
3	Registrations-Test wurde erfolgreich ausgeführt	9
4	Login mit user und password funktioniert	10