FULLSTACKDEVELOPMENT MIT SPRING BOOT

Fortgeschrittene Software-Entwicklung

Name: Julian Meilinger

Datum: 20.10.2021

# Einleitung

Spring Boot ist ein Open-Source-Micro-Framework. Es bietet Java-Entwicklern eine Plattform für den Einstieg in eine automatisch konfigurierbare Spring-Anwendung in Produktionsqualität.

Die Merkmale von Spring Boot lassen sich wie folgt zusammenfassen:

* direktes Einbetten von Webserver-/Container-Anwendungen wie Apache Tomcat oder Jetty möglich, wodurch kein Einsatz von WAR-Dateien (Web Application Archive) erforderlich ist
* vereinfachte Maven-Konfiguration dank „Starter“-POMs (Project Object Models)
* automatische Spring-Konfiguration, wann immer dies möglich ist
* Bereitstellung nichtfunktionaler Features wie Metriken oder ausgelagerter Konfigurationen

# Spring Boot Fullstack User-Management

In den folgenden Absätzen wird die erste CRUD-Anwendung in Spring Boot erstellt

## Projekt erstellen

Als erstes wurde ein GitHub-Projekt erstellt. <https://github.com/meilinger7/SbFirstCrud>

Spring Boot kann man direkt in der IntelliJ IDE erstellen, oder man legt sie über den Spring-Initializr an. <https://start.spring.io/>

Mit der IDE von IntelliJ:

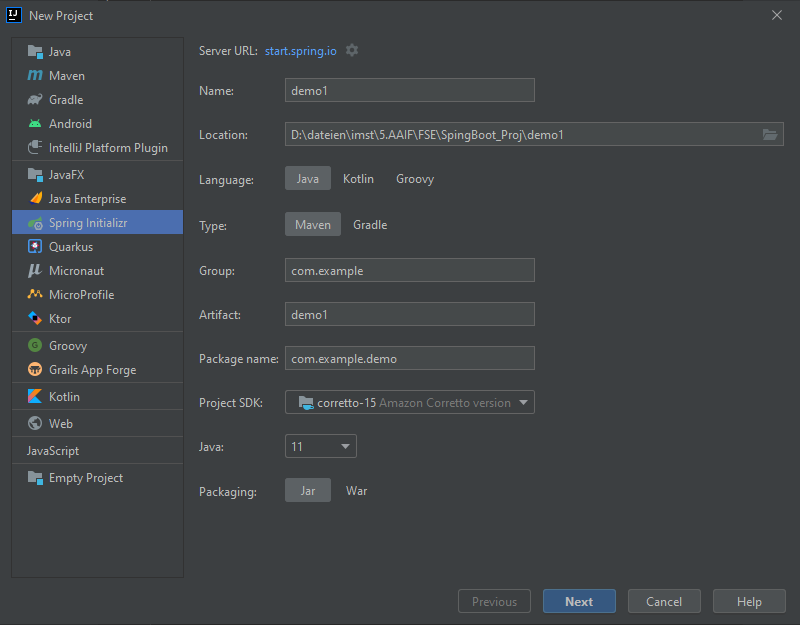


Abbildung 1: Settings des Spring Boot Projekt einstellen

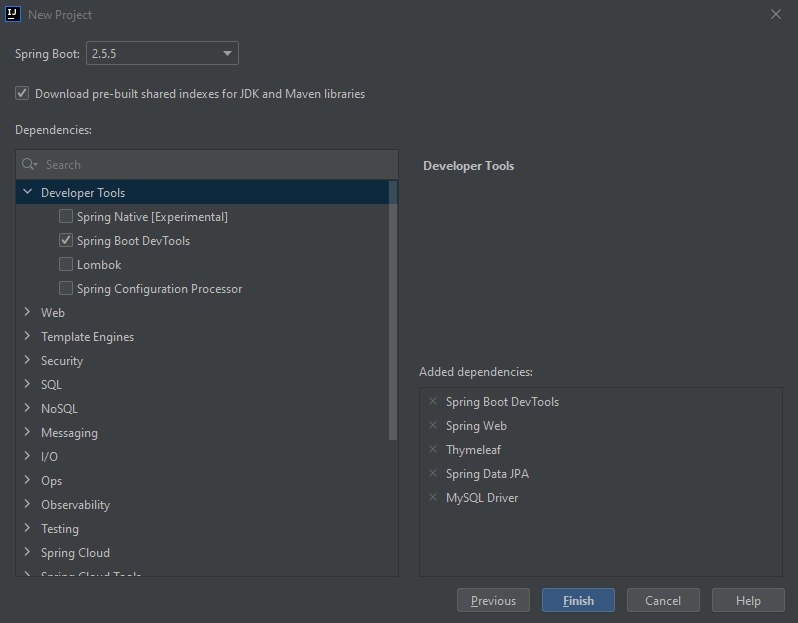


Abbildung 2: Dependencies von Spring Boot einstellen

Damit man nicht immer das Projekt neu starten muss, wenn man Veränderungen anzeigen möchte setzt man einen Hacken unter *File>Settings>Advanced Settings>Compiler.*

## Datenbank verbinden

Damit man sich mit einer Datenbank verbinden kann muss zuerst ein Datenbankservice laufen. Dafür habe ich den XAMPP MySQL Dienst gestartet.

In IntelliJ muss man mit dem Database-Tool sich nur noch mit der Datenbank verbinden und ein neues Schema erstellen.

### DataSource konfigurieren

Um eine Verbindung aufzubauen muss in der *application.properties* einige Metadaten zum Zugang der Datenbank eingefügt werden.

|  |
| --- |
| spring.datasource.url=jdbc:mysql://localhost:3306/userdb  spring.datasource.username=root  spring.datasource.password=  spring.jpa.hibernate.ddl-auto=update  spring.jpa.properties.hibernate.show\_sql=true |

## Homepage programmieren

Zuerst muss eine neue Java Klasse als Controller erstellt werden, damit die *index.html* beim Aufruf der Webseite aufgerufen wird.

|  |
| --- |
| @GetMapping("")  public String showHomePage() {  return "index";  } |

Eine index.html muss als nächstes im *templates* Ordner erstellt werden.

Beim Ausführen der main-funktion soll nun die Webseite über localhost:8080 erreichbar sein.

## Bootstrap hinzufügen

In der *pom.xml* müssen folgende Dependencies eingetragen werden:

|  |
| --- |
| <dependency>  <groupId>org.webjars</groupId>  <artifactId>bootstrap</artifactId>  <version>4.3.1</version>  </dependency>  <dependency>  <groupId>org.webjars</groupId>  <artifactId>webjars-locator-core</artifactId>  </dependency> |

Bei Fehlermeldungen muss Maven neu geladen werden.

In der index.html muss folgendes hinzugefügt werden:

|  |
| --- |
| <html lang="en" xmlns:th="http://www.thymeleaf.org">  …  <link rel="stylesheet" type="text/css" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}"> |

Somit ist Bootstrap auch eingebunden.

## Data Access Layer programmieren

Damit man Objekte der User-Tabelle mit dessen Entitäten erstellen kann, muss es eine Klasse mit den Passenden Eigenschaften geben.

Dazu wird die Klasse User im Subpackege user erstellt. Danach muss die @Entity und die @Table(name = „users“) Annotation hinzugefügt werden.

In dieser Klasse müssen folgende Eigenschaften samt Annotation eingefügt werden.

|  |
| --- |
| @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Integer id;  @Column(nullable = false, unique = true, length = 45)  private String email;  @Column(length = 15, nullable = false)  private String password;  @Column(length = 45, nullable = false, name = "first\_name")  private String firstname;  @Column(length = 45, nullable = false, name = "last\_name")  private String lastname; |

Die Annotationen dienen dazu die Tabellen-Entitäten einzustellen.

Es müssen auch automatisch Getter und Setter erstellt werden.  
Rechtsklick auf Eigenschaft > Refactor > Encapsulate Fields > Alle Eigenschaften auswählen > Refactor

### User Ropository

Es muss ein Interface mit der Bezeichnung UserRepository erstellt werden. Diese erweitert sich nach CrudRepository<User, Integer>.

Wenn man nun den Code Kompiliert erstellt Hibernate automatisch die Tabelle mit entsprechenden Eigenschaften.

## Unit-Tests programmieren

Die Automatisch erstellte Klasse im *test* Verzeichnis kann gelöscht werden.

Danach wird die Klasse UserRepositoryTests erstellt. In dieser Klasse werden Test-CRUD Operationen ausgeführt.

Folgende Annotationen müssen angefügt werden

|  |
| --- |
| @DataJpaTest  @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)  @Rollback(false)  public class UserRepositoryTests {  @Autowired private UserRepository repo; |

### Datensätze hinzufügen

Die erste CRUD Methode ist zum hinzufügen eines Users in die Datenbank

|  |
| --- |
| @Test  public void testAddNew(){  User user = new User();  user.setEmail("cooliannmeilinger@gmail.com");  user.setPassword("testitest");  user.setFirstname("Coolian");  user.setLastname("Meislinger");  User savedUser = repo.save(user);  Assertions.assertThat(savedUser).isNotNull();  Assertions.assertThat(savedUser.getId()).isGreaterThan(0);  } |

Es wird ein User Objekt erstellt und Parameter übergeben.

Damit das User Objekt in der Datenbank gespeichert werden kann muss die repo.save(user) Methode aufgerufen werden.

Zum Schluss wird noch überprüft ob savedUser nicht Null ist und ob die ID größer wie 0 ist.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

### Alle Datensätze auslesen

|  |
| --- |
| @Test  public void testListAll(){  Iterable<User> users = repo.findAll();  Assertions.assertThat(users).hasSizeGreaterThan(0);  for (User user : users) {  System.out.println(user);  }  } |

### Datensätze Updaten/Bearbeiten

|  |
| --- |
| @Test  public void testUpdate(){  Integer userId = 1;  String newPassword = "Hello2000";  Optional<User> optionalUser = repo.findById(userId);  User user = optionalUser.get();  user.setPassword(newPassword);  repo.save(user);  User updatedUser = repo.findById(userId).get();  Assertions.assertThat(updatedUser.getPassword()).isEqualTo(newPassword);  } |

### Datensatz per ID auslesen

|  |
| --- |
| @Test  public void testGet() {  Integer userId = 2;  Optional<User> optionalUser = repo.findById(userId);  Assertions.assertThat(optionalUser).isPresent();  System.out.println(optionalUser.get());  } |

### Datensatz per ID löschen

|  |
| --- |
| @Test  public void testDelete(){  Integer userId = 2;  repo.deleteById(userId);  Optional<User> optionalUser = repo.findById(userId);  Assertions.assertThat(optionalUser).isNotPresent();  } |

## User-Seite Programmieren

Bei der User-Seite sollen alle Benutzer in einer Tabelle aufgelistet sein.

Dazu muss die *UserService* und die *UserController* Klasse erstellt werden.

Die UserService Klasse ist dazu dar die die Daten vom UserRepository zu bekommen.

|  |
| --- |
| @Service  public class UserService {  @Autowired private UserRepository repo;  public List<User> listAll(){  return (List<User>) repo.findAll();  }  } |

Mit dem UserController stellt man die Verbindung von der User-View zu den Daten dar. Desweiterem leitet sie die URL zu der dementsprechenden user.html Seite weiter.

|  |
| --- |
| @Controller  public class UserController {  @Autowired private UserService service;  @GetMapping("/users")  public String showUserList(Model model){  System.out.println("Comiliert User ...");  List<User> listUsers = service.listAll();  model.addAttribute("listUsers", listUsers);  return "users";  }  } |

### Users View

Die user.html gibt eine Tabelle mit ID, E-Mail, Vorname und Nachname aus. Es gibt auch die Möglichkeit Datensätze zu erstellen, bearbeiten und zu löschen

Die Userdaten sollen in einer Tabelle ausgegeben werden. Die Daten werden in einer For-Each Schleife Zeile für Zeile ausgegeben.

In der Letzten Spalte soll für jede Reihe die Möglichkeit bestehen die Datensätze zu bearbeiten oder zu löschen.

|  |
| --- |
| <tbody>  <th:block th:each="user : ${listUsers}">  <tr>  <td>[[${user.id}]]</td>  <td>[[${user.email}]]</td>  <td>[[${user.firstName}]]</td>  <td>[[${user.lastName}]]</td>  <td>[[${user.enabled}]]</td>  <td>  <a class="h4 mr-3" th:href="@{'/users/edit/' + ${user.id}}">Edit</a>  <a class="h4" th:href="@{'/users/delete/' + ${user.id}}">Delete</a>  </td>  </tr>  </th:block>  </tbody> |

## Add User Function programmieren

Es muss ein Form erstellt werden, wo man Daten eingibt um einen neuen User anzulegen.

In der UserController Klasse muss für die neue HTML-Seite ein Eintrag gemacht werden.

|  |
| --- |
| @PostMapping("/users/save")  public String saveUser(User user, RedirectAttributes ra) {  service.save(user);  ra.addFlashAttribute("message", "The user has been saved succsesfully");  return "redirect:/users";  } |

Die Daten bezieht die Methode vom Form-Feld.

Danach muss die user\_form.html Datei erstellt werden. Hier wird ein Post-Formular erstellt, das die Daten an den Controller weitergibt. Damit keine falschen Einträge gemacht werden muss Validierung beachtet werden.

|  |
| --- |
| <form th:action="@{/users/save}" method="post" th:object="${user}"  style="max-width: 500px; margin: 0 auto;">  <input type="hidden" th:field="\*{id}">  <div class="border border-secondary rounded p-3">  <div class="form-row row">  <label class="col-sm-4 col-form-label">E-Mail</label>  <div class="col-sm-8">  <input type="email" th:field="\*{email}" class="form-control" required minlength="8" maxlength="45">  </div>  </div>  **…**  <div class="text-center">  <button type="submit" class="btn btn-primary m-2">Save</button>  <button type="button" class="btn btn-secondary m-2" onclick="cancelForm()">Cancel</button>  </div>  </div>  </form>  </div>  <script type="text/javascript">  function cancelForm() {  window.location = "[[@{/users}]]";  }  </script> |

Table 1: user\_form.html

**ACHTUNG: Diese Tabelle zeigt aus Platzgründen nicht alle Eingabefelder, sondern nur das zum Eingeben der E-Mail. Der gesamte Code ist im GIT auffindbar.**

Um zurück zur User-Seite zu gelangen wurde der JavaScript-Code eingefügt.

### Daten speichern

Damit die Daten vom Post-Formular in der Datenbank gespeichert werden, muss im UserController und in der User Service Klasse jeweils eine Funktion hinzugefügt werden.

Die UserController Klasse nummt das PostFormular an und gibt sie der Service-Klasse weiter.

|  |
| --- |
| @PostMapping("/users/save")  public String saveUser(User user, RedirectAttributes ra) {  service.save(user);  ra.addFlashAttribute("message", "The user has been saved succsesfully");  return "redirect:/users";  } |

Table 2: UserController.java

In der UserService Klasse werden die Daten dann im UserRepository als Objekt angelegt und danach in die Datenbank gespeichert.

|  |
| --- |
| public void save(User user) {  repo.save(user);  } |

Table 3: UserService.java

### Meldung fürs Speichern

Wenn das Speicher erfolgreich war soll in der user.html eine Meldung erscheinen.

|  |
| --- |
| <div th:if="${message}" class="alert alert-success text-center">  [[${message}]]  </div> |

Table 4: user.html

## Datensätze Editieren

In der UserService Klasse muss eine neue Methode zum Auslesen der Daten mit einer bestimmten ID erstellt werden.

|  |
| --- |
| public User get(Integer id) throws UserNotFoundException {  Optional<User> result = repo.findById(id);  if (result.isPresent()){  return result.get();  }  throw new UserNotFoundException("Could not find any Users With ID" + id);  } |

Wenn kein User gefunden wurde, soll eine Exception geworfen werden.

Somit muss auch eine Exception-Klasse im user-Package erstellt werden.

|  |
| --- |
| public class UserNotFoundException extends Throwable {  public UserNotFoundException(String message) {  super(message);  }  } |

Damit die Abfragen auf eine HTML-Seite weitergeleitet werden, muss die UserController Datei bearbeitet werden.

|  |
| --- |
| @GetMapping("/users/edit/{id}")  public String showEditForm(@PathVariable("id") Integer id, Model model, RedirectAttributes ra) {  try {  User user = service.get(id);  model.addAttribute("user", user);  model.addAttribute("pageTitle", "Edit User (ID: " + id + ")");  return "user\_form";  } catch (UserNotFoundException e) {  ra.addFlashAttribute("message", "The user has been saved successfully");  return "redirect:/users";  }  } |

## Datensätze löschen

Um Datensätze zu löschen, müssen wieder Delete-Methoden im Service- und im Controller-Layer erstellt werden.

Desweiterem wird eine Hilfsfunktion im UserRopository Interface erstellt, damit man weis ob es die Funktion schon gibt.

|  |
| --- |
| public void delete(Integer id) throws UserNotFoundException {  Long count = repo.countById(id);  if (count == null || count == 0){  throw new UserNotFoundException("Could not find any users with ID " + id);  }  repo.deleteById(id);  } |

Table 5: UserService.java

|  |
| --- |
| public Long countById(Integer id); |

Table 6:UserRepository.java

|  |
| --- |
| @GetMapping("/users/delete/{id}")  public String deleteUser(@PathVariable("id") Integer id, RedirectAttributes ra) {  try {  service.delete(id);  ra.addFlashAttribute("message", "The user ID " + id + "has been deleted")  } catch (UserNotFoundException e) {  ra.addFlashAttribute("message", e.getMessage());  }  return "redirect:/users";  } |

Table 7: UserController.java