Multi-Users Backend

Thema: ÜK-223

Dokumentinformationen

Dateiname: Dokumentation\_ÜK223.docx

Speicherdatum: 25.02.2022

Autoreninformationen

Autor: Venturini Alessio, Hartmann Ronnie

E-Mail: alessio.venturini@noser.com, ronnie.hartmann@noser.com

Inhaltsverzeichnis

[Abbildungsverzeichnis 2](#_Toc96695565)

[Tabellenverzeichnis 2](#_Toc96695566)

[Änderungsgeschichte 2](#_Toc96695567)

[1 Einleitung 3](#_Toc96695568)

[1.1 Sinn und Zweck 3](#_Toc96695569)

[1.2 Referenzdokumente 3](#_Toc96695570)

[1.3 Abkürzungen 3](#_Toc96695571)

[2 Vorbereitungen 4](#_Toc96695572)

[2.1 Spring Boot Skelett 4](#_Toc96695573)

[2.2 GitHub Repository 4](#_Toc96695574)

[3 Arbeit am Projekt 4](#_Toc96695575)

[3.1 MyListEntry 4](#_Toc96695576)

[3.2 Logging 6](#_Toc96695577)

Abbildungsverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

Tabellenverzeichnis

[Tabelle 1 Versionen 2](#_Toc449701436)

[Tabelle 2 Abkürzungen 3](#_Toc449701437)

Änderungsgeschichte

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Datum** | **Autor** | **Details** |
| 1.0 | 23.02.2022 | Alessio Venturini | Dokument erstellt |
| 1.1 | 23.02.2022 | Ronnie Hartmann | Einleitung geschrieben |
| 1.2 | 24.02.2022 | Ronnie Hartmann | Vorbereitungen geschrieben |
| 1.3 | 24.02.2022 | Ronnie Hartmann | Arbeit am Projekt   * MyListEntry Doku begonnen |
| 1.4 | 25.02.2022 | Ronnie Hartmann | Doku zu Logging geschrieben |
| 1.5 | 25.02.2022 | Ronnie Hartmann | Doku zu Encoding geschrieben |

Tabelle 1 Versionen

# Einleitung

## Sinn und Zweck

Im ÜK 223 müssen wir in einer Gruppenarbeit ein Multi-User Projekt mit Spring Boot erstellen. Ziel ist es, ein Backend für eine Blog-Website zu entwickeln. In diesem Dokument beschreiben wir unsere Planung, Probleme und unser Vorgehen bei der Projektarbeit.

## Referenzdokumente

1. …
2. …

## Abkürzungen

|  |  |
| --- | --- |
| **Abkürzung** | **Beschreibung** |
| NYP | **N**oser **Y**oung **P**rofessionals |
| ÜK | **Ü**berbetrieblicher **K**urs |

Tabelle 2 Abkürzungen

# Vorbereitungen

Zu Beginn mussten wir ein paar Vorbereitungen treffen, um mit dem Spring Boot Projekt zu beginnen. Wir mussten das Spring Boot Projekt, welches Luca uns geschickt hat, zum Laufen bringen und ein GitHub Repository erstellen. Ausserdem mussten wir ein Docker Image für die PostgreSQL Datenbank erstellen.

## Spring Boot Skelett

Luca hat uns ein Spring Boot Skelett gegeben, womit wir für das Projekt arbeiten werden. Dieses Skelett enthält bereits User mit Rollen.

Beim Starten des Projektes kam es zu einer Exception. Dieses Problem lösten wir, indem wir bei der DemoApplication Klasse das Keyword **public** hinzugefügt haben.

## GitHub Repository

Auf GitHub haben wir ein Repository mit einem Main- und Develop Branch erstellt. Wir werden mit Feature Branches arbeiten, welche wir dann jeweils mit Pull Request in den Develop Branch mergen.

# Arbeit am Projekt

Nach ein paar Inputs von Luca haben wir mit dem Projekt begonnen. Wir bekamen die Aufgabe ein neues Model «MyListEntry» mit verschiedenen Endpoints zu erstellen. Die Logik muss sich im Service befinden. Der Service holt dann die Daten vom JPA Repository, welches auf die Datenbank zugreift.

## MyListEntry

Als erstes haben wir in unserem Spring Boot Skelett ein neues Package für das MyListEntry erstellt. Darin befindet sich ein Model, ein Controller, ein Service mit seinem Interface und ein Repository.

### Model: MyListEntry.java

Das Model enthält die Felder «id», «titel», «text», «erstellungsdatum» und «wichtigkeit». Mit Annotations wie **@NotNull()** oder **@Size()** werden die Felder validiert. Diese Validierungen werden dann automatisch in der Datenbank übernommen.

Da jeder User mehrere MyListEntrys haben kann, mussten wir eine Zwischentabelle erstellen. Dafür haben wir die Annotations **@ManyToOne()** und **@JoinTable()** verwendet.

@ManyToOne(fetch = FetchType.*EAGER*)  
@JoinTable(  
 name = "mylistentry\_user",  
 joinColumns = @JoinColumn(  
 name = "mylistentry\_id", referencedColumnName = "id"),  
 inverseJoinColumns = @JoinColumn(  
 name = "user\_id", referencedColumnName = "id"))  
private User user;

Der Join ist unidirectional, also kann man von MyListEntry direkt auf den User zugreifen, aber nicht vom User auf seine MyListEntrys.

### Controller: MyListEntryController.java

Im Controller werden die Endpoints definiert. Wir haben die Endpoints «/api/myListEntry» und «/api/myListEntry/{id}» erstellt. Es gibt eine CREATE Methode, um ein MyListEntry zu erstellen, zwei GET Methoden, um alle oder ein bestimmtes MyListEntry zu bekommen, eine DELETE Methode, um ein MyListEntry zu löschen und eine PUT Methode, um ein MyListEntry zu überschreiben.

Später haben wir noch einen Endpoint erstellt, welcher die MyListEntries nach User filtert und nach Wichtigkeit sortiert.

@GetMapping("/{username}")  
public ResponseEntity<Collection<MyListEntry>> findAllByUser(@PathVariable String username) {  
 return new ResponseEntity<>(myListEntryService.findAllByUser(username), HttpStatus.*OK*);  
}

Die Logik geschieht nicht im Controller, sondern im Service. Deshalb gibt es hier nicht viel zu sehen. In diesem Fall habe ich noch einen Custom Query im Repository geschrieben.

Wir haben auch noch einen Endpoint erstellt, welcher alle MyListEntrys in Seiten aufteilt. Jede Seite ist 5 Entrys lang und man kann die Seitenzahl in der URL angeben.

@Operation(summary = "Get all MyListEntries pageable")  
@GetMapping("/page/{page}")  
public ResponseEntity<Collection<MyListEntry>> findAllPageable(@PathVariable int page) {  
 return new ResponseEntity<>(myListEntryService.findAllPageable(page), HttpStatus.*OK*);  
}

Auch hier befindet sich die Logik im Service, weshalb die Methode nur eine einzige Zeile enthält.

### Service: MyListEntryServiceImpl.java & MyListEntryService.Java

Für einen Service erstellt man jeweils ein Interface und die eigentliche Implementation. Unser Interface sieht so aus:

public interface MyListEntryService {  
 List<MyListEntry> findAll();  
 List<MyListEntry> findAllPageable(int page);  
 List<MyListEntryDTO> findAllDTO();  
 MyListEntry findById(UUID id);  
 MyListEntry createMyListEntry(MyListEntry myListEntry);  
 void deleteMyListEntry(UUID id);  
 MyListEntry putMyListEntry(MyListEntry myListEntry, UUID id);  
 List<MyListEntry> findAllByUser(String username);  
 MyListEntry saveMyListEntry(MyListEntry myListEntry);  
}

Hier ist beispielsweise die Implementation der **findAllPageable()** Methode. Als Argument gibt man die Seitenzahl mit. Die Länge der Seite ist immer 5.

@Override  
public List<MyListEntry> findAllPageable(int page) {  
 Pageable pageable = PageRequest.*of*(page, 5, Sort.*by*("erstellungsdatum"));  
 return myListEntryRepository.findAll(pageable).getContent();  
}

Die Entrys werden nach Erstellungsdatum sortiert. Somit wird das älteste MyListEntry als erstes angezeigt.

### Repository: MyListEntryRepository.java

Unser Repository erbt vom JPA Repository. Dieses enthält bereits die CRUD Methoden (Create, Read, Update, Delete). Wenn man trotzdem eigene Querys schreiben möchte, kann man diese dort schreiben.

@Repository  
public interface MyListEntryRepository extends JpaRepository<MyListEntry, UUID> {  
 @Modifying  
 @Query("SELECT m FROM mylistentry m WHERE m.user.id = :userId ORDER BY m.wichtigkeit")  
 List<MyListEntry> findAllByUser(@Param("userId") UUID userId);  
}

Wir haben einen custom Query geschrieben, um alle MyListEntrys von einem bestimmten Benutzer zu bekommen. Die MyListEntrys werden nach Wichtigkeit sortiert.

## Logging

Wir loggen alle Änderungen in der Datenbank. Bei den GET Methoden wird nichts geloggt, da wir finden, dass es nicht so wichtig ist. Wenn beispielsweise ein Fehler, während dem Erstellen eines MyListEntrys auftreten würde, könnte man in den Logs trotzdem nachverfolgen, dass jemand ein MyListEntry erstellen wollte.

Das Logging geschieht im Service, da wir unsere Controller so kompakt wie möglich haben wollen. Um zu loggen haben wir über unserem Service die Annotation **@Log4j2** hinzugefügt. Somit hat man das Objekt «log», womit man loggen kann.

@Override  
public void deleteMyListEntry(UUID id) {  
 *log*.info("Delete MyListEntry with ID " + id);  
 myListEntryRepository.deleteById(id);  
}

Hier wird im Service geloggt, dass ein MyListEntry gelöscht wurde. In den application.properties mussten wir den Pfad des Log Ordners definieren.

logging.file.path=logs

## Encoding

Zu Beginn wurden alle Passwörter unverschlüsselt in der Datenbank gespeichert. Um das Programm etwas sicherer zu machen, haben wir die Passwörter mit BCrypt verschlüsselt. Dazu mussten wir zuerst eine Bean in der DemoApplication Klasse erstellen. Diese gibt uns einen PasswordEncoder zurück.

@Bean  
PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
}

Diese Bean brauchen wir dann in den Security Konfigurationen und im User Service, um die Passwörter zu verschlüsseln.

### User Service

Dadurch, dass wir eine Bean für unseren PasswordEncoder erstellt haben, können wir ihn im Service mit Hilfe von Dependency Injection initialisieren. Diesen PasswordEncoder verwenden wir dann, um beim Erstellen eines Users das Passwort zu verschlüsseln.

@Override  
public User saveUser(User user) throws InstanceAlreadyExistsException{  
 if (userRepository.findByUsername(user.getUsername()) != null) {  
 throw new InstanceAlreadyExistsException("User already exists");  
 }  
 else {  
 user.setPassword(passwordEncoder.encode(user.getPassword()));  
 return userRepository.save(user);  
 }  
}

### SecurityConfig

In den Konfigurationen habe ich eine Methode erstellt, welche einem einen authProvider zurückgibt.

public DaoAuthenticationProvider authProvider() {  
 DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();  
 authProvider.setUserDetailsService(userDetailsService);  
 authProvider.setPasswordEncoder(passwordEncoder);  
 return authProvider;  
}

Wir brauchen ihn in der **configure(AuthenticationmanagerBuilder auth)** Methode. In dieser Methode definieren wir auch den PasswordEncoder.

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder).and()

.authenticationProvider(authProvider());  
}

Auch in der **configureGlobal()** Methode brauchen wir den PasswordEncoder.

@Autowired  
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder);  
}

Nun werden die Passwörter beim Erstellen eines Users verschlüsselt. Auch beim Einloggen gibt es keine Fehler. So sieht dann unsere Passwort Spalte mit zwei Usern aus:

|  |
| --- |
| **password** |
| **$2a$10$1ueuiH4G4g7aIGUcv7gRh.nLQtDIzWrbgjI.g2ytZH2ubFBdXOFlu** |
| **$2a$10$7icq1QqGpB8FUJQmrcZ2jeUNAO8V4LuS1ZhS7VpRz3jQtT1VGh7SK** |