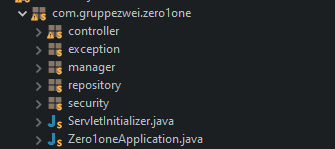
# Code Beschreibung Backend

Im Backendteil der Anwendung der Fallstudie wird das Framework Spring verwendet. Für einen übersichtlichen Aufbau wurde die gesamte Anwendung in einzelne Teile unterteilt. Diese Unterteilung spiegelt sich in den Package-Struktur wider. Unter dem Hauptpackage com.gruppezwei.zero1one befindet sich der gesamte Java Quellcode. Direkt unter diesem Package befinden sich nur zwei Klassen. Beide erfüllen die gleiche Aufgabe, jedoch für unterschiedliche Umgebungen. Der ServletInitizializer wird für den Start auf einem Server benötigt. Die andere Klasse ist für den lokalen Start auf einem Computer zuständig.



//Hier sollte noch was über die Auslieferung stehen

# Zentrales Management

Der wichtigste Teil der Anwendung ist der manager, er ist das Rückgrat des Programms. Dieses Package ist der Dreh- und Angelpunkt. Der Manager kümmert sich um die Verwaltung der Datenbankabfragen, das Mapping der Abfrageergebnisse und das Persistieren der Configuration Items. Im Inneren befinden sich fünf Manager, die jeweils auf einen Aufgabebereich spezialisiert sind. Der ReadManager liefert die Methoden, die später aufgerufen werden, um die HTML-Oberfläche mit Daten zu füllen. Die Methoden, welche die eigentlichen Abfragen ausführen stehen in Repositorys.

|  |
| --- |
| @Autowired  ConfigitemRepository confItemRepo; |

Diese werden in allen Managern mit der @Autowired- Annotation instanziiert. Wie im theoretischen Teil zu Spring erläutert wird dies verwendet, um die Lebenszyklen der Instanzen bzw. Beans voneinander zu trennen. Damit Spring diese Annotationen im Gesamtkontext erkennt muss die gesamte Manager-Klasse mit @Component annotiert werden. Die einzelnen Methoden rufen die Methode des Prepositorys auf und triggern die Query. Diese Liefert eine Liste der abgefragten Objekte. Diese werden nun im Manager gemappt und so zusammengeführt, damit sie im Frontend präsentiert werden können.

|  |
| --- |
| public List<CiRecord> getCiRecordAll() {  return confItemRepo.findAll()  .stream()  .map(this::convertToCiRecord)  .collect(Collectors.toList());  } |

Das Mapping erfolgt für jedes einzelne Objekt in der Liste. Diese wird im Quellcode mit Lambda-Ausdrücken realisiert, welche sein Java 8 den Code übersichtlicher gestalten und Aufgaben wie das Mapping erheblich erleichtern. Im Wesentlichen wird für jedes Objekt der Liste, hier ConfigItems, die Methode convertToCiRecord ausgeführt. Diese Liefert als Antwort ein CiRecord. Im Unterschied zu einem ConfigItem besitzt ein CiRecord noch eine Liste mit allen Attributen, die zu diesem ConfigItem gehören. Diese Liste wird ebenfalls in der Mappingmethode abgefragt und in ihren entsprechenden typ gepackt. In der Praxis wird also eine Abfrage für die Liste der ConfigItems abgesetzt und dann für jedes weitere Item noch eine Abfrage. Dies ist für einen Einsatz mit vielen Benutzern sehr unpraktisch und beeinträchtigt die Geschwindigkeit der Datenbank und somit die der Anwendung erheblich, wenn viele Nutzer gleichzeitig eine Anfrage senden. Da diese Anwendung nur lokal mit höchstens einem Benutzer eingesetzt werden soll ist jedoch kein Performancenachteil zu bemerken. Da sich zudem Joins mit Spring nur mit erhöhtem Aufwand umsetzen lassen ist diese Vorgehensweise zu bevorzugen.

Der PersistenceManager und der UpdateManager kümmern sich um das Speichern der angelegten oder veränderten Configuration Items. Die grundlegende Funktion ist analog zum ReadManager, mit einer umgekehrten Richtung des Datenflusses. Hier werden die Objekte wieder in umgekehrter Reihenfolge gemappt und mit entsprechenden Querys abgespeichert. Der größte Unterschied der beiden Speicher-Manager ist, dass der PersistenceManager neue Objekte abspeichert und somit neue Id‘s zu den Items hinzufügen muss, damit diese im weiteren Gebrauch eindeutig identifiziert werden können.

Der DeleteManager leitet die Löschanfragen. An dieser Stelle werden keine Objekte abgefragt oder umgewandelt. Die einzelnen Items werden über die ID gelöscht. Der AuthenticationManager kümmert sich um die Abfrage und Verwaltung der Benutzerdaten. Es gibt also in der gesamten Anwendung kein Vorgang, der nicht durch den Manager gesteuert wird.

# Datenbankzugriff

Der Wurzel der Datenhaltung bilden die Repositories im repository-Package. Neben den Repositories befinden sich hier auch die POJOs die zur Abfrage benötigt werden. Diese Plain Old Java Objects (POJO) haben die Aufgabe den Inhalt einer Datenbankabfrage wieder zu geben. Im Regelfall bilden die POJOs eine Zeile der Tabelle ab, die abgefragt wird. Sie stellen also eine Entität dar, die in dieser Tabelle gespeichert ist. Aus diesem Grund werden diese Klassen auch mit @Entity annotiert.

|  |
| --- |
| @Entity  public class Configitem {  @Id  private int id;  private String configitemtypname;  private String name;  //Getter and Setters  } |

Durch diese Annotation weiß Spring, dass diese Klasse eine Entität auf der Datenbank wiederspiegelt. Damit eine saubere Abfrage erstellt werden kann muss der Schlüssel auf der Datenbanktabelle mit @Id gekennzeichnet werden. Die Typen der Attribute müssen hier nicht zwangsläufig mit denen der Datenbank übereinstimmen. Eine Zahl oder ein Datum können auch als String abgespeichert werden. Umgekehrt funktioniert dies in Java nicht. Der Name der Attribute sollte hingegen mit den Spaltennamen der Datenbank übereinstimmen. Dies gilt auch für den Namen der Klasse. Falls diese nicht übereinstimmen, oder man sicher gehen möchte, dass die richtige Tabelle und die richtigen Spalten angesprochen werden kann man auch Namen zuordnen. Auch hier arbeitet Spring mit Annotationen. Man kann bereits hinter @Entity den Namen spezifizieren oder nutzt die eigene Annotation @Table:

|  |
| --- |
| @Table(name=”Spaltenname”) |

Analog gibt es auch eine Annotation @Column. Speziell für die Tabelle kann man auch den Namen des Schemas angeben. Stimmen die Namen aber mit den Klassen- und Attributnamen überein, besteht kein bedarf für diese Annotationen. Nur @Entity und @Id sind zwingend erforderlich.

|  |
| --- |
| @Repository  public interface ConfigitemRepository   extends JpaRepository<Configitem, Integer> {    List<Configitem> findByName(String name);    List<Configitem> findTopByOrderByIdDesc();  } |

Ein Repository übernimmt die eigentliche Aufgabe der Abfrage. Im Gegensatz zu den anderen Teilen wird ein Repository als Interface implementiert. In dieser Anwendung erben alle Repositories von einem JpaRepository. Dieses erbt ursprünglich von einem CRUD-Repository. Wie im Namen CRUD schon zu erkennen ist, ermöglicht dieses Repository die grundlegenden Operationen Create, Read, Update und Delete. Das Repository ist generisch aufgebaut und benötigt zwei Parameter: Die Entity, die Abgefragt werden soll, und den Datentyp des Schlüssels. Im obigen Beispiel sind dies die Entity Configitem und der Primärschlüssel vom Typ Integer. Mit diesen Informationen kann Spring bereits Datenbankabfragen durchführen. Voraussetzung ist, dass in der application.properties die Zugangsdaten und der Treiber der Datenbank hinterlegt sind.

|  |
| --- |
| spring.datasource.url=jdbc:h2:file:./src/main/resources/data/h2.hdb  spring.datasource.driverClassName=org.h2.Driver  spring.datasource.username=user  spring.datasource.password=password |

Wie im Manager bereits erwähnt wird ein Repository immer autowired. Es wird als vom Spring-Anwendungskontext als Bean zur Verfügung gestellt. An dieser Stelle wird auch deutlich welche große Aufgabe das Framework übernimmt. Es baut mit den Informationen über die Datenbank eine Connection auf. Diese muss nicht manuell erzeugt werden. Dadurch entfällt das Aufbauen der Verbindung vor der Abfrage und das Abbauen danach. Durch diese Verwaltungsarbeit des Frameworks bleibt hier der Quellcode übersichtlich. Auch entfällt eine Implementierung des Interfaces, auch diese Aufgabe wird von Spring übernommen.

Das Repository beherrscht bereits einfache Abfragen wie findAll() oder findById(<S> id). Die Methoden antworten entweder mit einer Liste von Configitems, oder mit einem Optional-Objekt, je nachdem wie viele Ergebnissätze erwartet werden. Zudem sind auch Methoden delete() zum Löschen und zum Speichern save() vorgesehen. Ein Objekt kann entweder über die Id oder über die Entität selbst gelöscht werden. Die Speichermethoden können sowohl zum Neuanlegen als auch zum Aktualisieren verwendet werden. Gibt man ein Objekt mit, dessen Primärschlüssel schon existiert, werden alle restlichen Parameter überschrieben.

Sind jedoch kompliziertere Abfragen nötig, um gewünschte Ergebnisse zu erhalten hat man zwei Möglichkeiten. Erstens kann man eine Methode im Repository mit @Query versehen und schreibt dahinter in Klammern eine eigene Query. Diese muss als Java Database Connectivity (JDBC)-konforme Query geschrieben werden, damit Spring diese, mit dem in der pom.xml hinterlegten Treiber, zu einer ausführbaren Abfrage umwandeln kann.

Als zweite Möglichkeit kann man Methoden vorgeben, die bestimmte Schlüsselwörter im Namen tragen. Ein simples Beispiel ist oben im ConfigitemRepository gegeben. Nennt man die Methode findBy und anschließend das gewünschte Attribut, nachdem gesucht werden soll, dann bildet Spring eine Query die genau dies macht. Diese Schlagworte können wesentlich komplizierter werden. Im Beispiel darunter wird die oberste Entität zurückgegeben, nachdem die Tabelle absteigend nach ihrer ID geordnet wurde. Es wird also das Element mit der größten ID zurückgegeben.

Um an die Attribute zu kommen, die zu einem CiRecord gehören, muss man normalerweise einen Join durchführen. Da wir aber in einem Repository immer nur exakt eine Tabelle wiederspiegeln ist dies nicht so einfach. Es kann natürlich ein Repository erstellt werden, dass ein solches Ergebnis entgegennimmt. Dies ist aufwändiger als mehrere Abfragen durchzuführen. Da wie bereits erwähnt die Geschwindigkeit der Abfragen bei dieser Single-User-Anwendung eine untergeordnete Rolle spielt wurde auf die zweite einfachere Variante zurückgegriffen.

# Oberflächenverwaltung

//Controller und Teil Zum Frontend (Max)

# Sicherheit

Das Sicherheitskonzept und die Benutzerverwaltung sind mit Spring Security umgesetzt. Dieser sichert alle Seite ab, die von unserer Anwendung verwaltet weden, Im eigenen Package, der security, wird die SecurityConfig implementiert, welcher die zentralen Sicherheitsfunktionen verwaltet. Sie erbt vom WebSecurityConfigurerAdapter. Zum einen wird hier die Methode configure überschrieben, die dafür zuständig ist den Nutzer zu authentifizieren. Dazu ruft sie den CustomAuthenticationprovider auf. In der anderen configure-Methode wird die HttpSecurity konfiguriert.

|  |
| --- |
| @Override  protected void configure(HttpSecurity http) throws Exception{  http.authorizeRequests().antMatchers("/resources/\*\*")  .permitAll().anyRequest().authenticated().and()  .formLogin().loginPage("/login")  .defaultSuccessUrl("/dashboard",true)  .permitAll().and()  .logout().permitAll();  } |

Hier werden die essenziellen Verhaltensweisen der Sicherheit festgelegt. Als erstes wir fesgelegt, dass alle Requests akzeptiert werden. Mit .authenticated() wird konfiguriert, dass jeder Request authentifiziert sein muss, sonst wird er nicht bearbeitet. Mit .formLogin() wird eine eigenes Loginformular registriert, welches unter der Url /login aufgerufen wird. Damit diese aufgerufen werden kann wird die Url im SecurityController gemappt und festgelegt welches Loginformular aufgerufen werden soll, wenn eine Getanfrage eingeht. Mit .defaultSuccessUrl(..) wird die url hinterlegt, die nach dem Login aufgerufen werden soll.

Im CustomAuthenticationprovider wird der standard AuthenticationProvider überschrieben. An dieser Stelle wird durch den AuthenticationVerifier der AuthenticationManager aufgerufen. Dieser fragt den Nutzer und das Passwort ab. Das Passwort selbst wird nur als Hashwert gespeichert und mit dem verhashten Wert der Eingabe abgeglichen. Der AuthenticationVerifier gibt nach erfolgreicher Anmeldung einen AuthenticationToken züruck, der als Cookie gespeichert wird. Über diesen authentifiziert sich der Browser nun auf allen Seiten. Die Eingabe des Passworts ist clientseitig nicht komplett geschützt. Die Eingabe wird erst auf dem Server verhasht. Diese ist für eine Online-Anwendung sehr schlecht. Aber aufgrund des Anwendungsfelds des Programms ist dies keine große Sicherheitslücke, da die Anwendung lokal betrieben wird.

# Fehlerhandling

Um das Programm der Fallstudie abzurunden muss noch ein geeignetes Fehlerhandling implementiert werden. Auch hier bietet Spring einen Default an, der optisch jedoch wenig ansprechend ist. Für eine eigene Fehlerseite wird für clientseitige Fehler die Standardposition für eigene Fehlerseiten benutzt. Legt man unter src/main/ressources/templates eine HTML-Seite mit dem Namen error.html ab, ruft Spring diese automatisch im Fehlerfall auf. Durch einfache Thymeleaf-Befehle werden auf dieser der Fehlerstatus und die textuelle Beschreibung des Fehlercodes ausgegeben. Da im Fall eines internen Fehlers so nur eine unschöne 500 Internal Server Error Meldung ausgegeben wird, werden diese Fehler auf eine andere Weise gehandled.

Damit eine individuelle Fehlermeldung für interne Fehler ausgegeben werden kann wird im Package exception ein ExceptionHandler implementiert. Dieser wird als @ControllerAdvice annotiert. Damit wird sichergestellt, dass Spring die hier verwalteten Fehler überprüft, bevor die Standardmeldung gesetzt wird.

|  |
| --- |
| @ExceptionHandler(EmptyResultDataAccessException.class)  public String handleException(EmptyResultDataAccessException ex){  return "errormanual"  } |

Das Beispiel zeigt eine vereinfachte Handler-Methode. Diese ExceptionHandler reagiert auf eine EmptyResultDataAccessException. Wird dieses Exception irgendwo im Programm geworfern, schaut Thymeleaf nach einer errormanual.html im templates-Ordner und gibt diese HTML-Seite aus. Die weiteren Methoden wurden wie in den Controller-Klassen um ein Model erweitert, welches eine passende Fehlermeldung mit ausgeben kann.

Im gleichen Package sind auch alle eigenen Exceptions abgelegt. Durch die Erstellung eigener Exceptions kann die Ausgabe im Fehlerfall sehr genau ausfallen, da jeder Fehlerfall seine eigene Fehlermeldung hat. So kann ein gewährleistet werden, dass der Nutzer weiß, was nicht funktioniert, ohne diesem eine Auskunft über den internen Aufbau der Anwendung zu geben. Dies wäre besonders relevant, wenn die Anwendung der Fallstudie auf einem Webserver für externe Nutzer zugänglich wäre.