Department of Computer Sciences
University of Salzburg

PS Software Praktikum
WS 17/18

Cool Roofs

28. Februar 2018

Project Members:

Michael Nening,	irgendwas,	michael.nening@stud.sbg.ac.at
Martin Schönegger,	irgendwas,	martin.schoenegger@stud.sbg.ac.at
Timo Niederleuthner,	01521540,	timo.niederleuthner@stud.sbg.ac.at
Michael Moser,	irgendwas,	michael.moser@stud.sbg.ac.at

Correspondence to:

Universität Salzburg
Fachbereich Computerwissenschaften
Jakob–Haringer–Straße 2
A–5020 Salzburg
Austria

Inhaltsverzeichnis

1 Kurzfassung

Das Projekt Cool-Roofs ist im Umfang der Lehrveranstaltung Software Praktikum der Universität Salzburg entstanden. Die globale Erwärmung ist ein allgegenwärtiges Problem; mit diesem Projekt soll ein Beitrag dagegen geleistet werden. Es gibt eine spezielle Farbe mit der Dächer angestrichen werden können, damit diese einen Großteil der Sonnenstrahlung zurück reflektieren. Auf diese Weise lässt sich der Treibhauseffekt eindämmen.
Im Rahmen dieses Projekts entwickelten wir eine Webapplikation, mit der sich Hausbesitzer und Investoren in Verbindung setzen können.Ein Hausbesitzer kann sein Dach registrieren, und Investoren finanzieren mit ihrer gezahlten Summe das Streichen des Daches. Dieses hat auch für die Dachbesitzer einen positiven Effekt: Die Häuser wärmen sich bei viel Sonneneinstrahlung nicht mehr so auf wie früher, was in einem angenehmeren Wohnklima resultiert und dazu führt, dass Klimaanlagen weniger oft eingeschalten werden müssen. Jeder Investor/Hausbesitzer soll außerdem eine Anzeige des gesparten CO_2 und eine Google-Maps Ansicht des Hauses bekommen.

2 Einleitung

Mittags an einem klaren Sommertag in den Vereinigten Staaten erhält eine flache (horizontale) Oberfläche etwa 1000 Watt Sonnenlicht pro Quadratmeter. Traditionelle dunkle Dächer absorbieren dieses Sonnenlicht stark und erhitzen sowohl das Gebäude als auch die umgebende Luft. Dies erhöht den Energieverbrauch in klimatisierten Gebäuden und macht nicht klimatisierte Gebäude weniger komfortabel. Heiße dunkle Dächer verschärfen auch die städtischen Wärmeinseln, indem sie die über das Dach strömende Luft erwärmen und durch die Wärmeabstrahlung in die Atmosphäre zur globalen Erwärmung beitragen.
In unserer Webapplikation ist es möglich, sich entweder als Hausbesitzer oder als Investor einzuloggen. Dies ist einfach über einen vorhandenen Facebook/Google -Account möglich oder über eine normale Registrierung. Pro Account ist es nur möglich, entweder Investor oder Hausbesitzer zu sein.
Als Hausbesitzer ist man in der Lage seine Häuser auf einer Google-Maps-Karte einzuzeichnen. Gleichzeitig ist die Angabe von Dachtyp und Alter notwendig. Dabei wird die Fläche der Dächer automatisch über das gekennzeichnete Dach berechnet und muss nicht vom Hausbesitzer eingegeben werden. Außerdem sieht der Hausbesitzer, ob auf seinem Dach bereits eine Investition getätigt wurde und bekommt nach Anstrich des Daches eine Anzeige des gesparten $C0_2$.

Als Investor ist es nun möglich, sich ein Dach auszusuchen und in dieses zu investieren. Der Investor geht dafür auf die entsprechende Seite und kann dort die Region auswählen und wie viel m^2 er investieren möchte. Die Regionen sind dabei beschränkt auf Europe, Asia, North America, Africa, South America, Australia/Oceania, Other. Als nächstes werden ihm eine Reihe von verschiedenen Häusern vorgeschlagen, in welche er Investieren kann. Entweder bestätigt er diese oder er drückt auf Reset und bekommt eine neue Liste von Dächern angezeigt. Weiters hat der Investor Zugriff auf eine Ansicht all seiner Investitionen und sieht dabei das gesparte $C0^2$ pro Dach und in Summe und ein Google-Maps-Karte mit dem jeweiligen Dach.

3 Verwendete Frameworks

Im Projekt Cool-Roofs wurde mit Spring-Boot gearbeitet und damit verbunden mit diversen Frameworks. Hier werden wir Ihnen einen kurzen Überblick über die verwendeten Frameworks geben und anhand einiger Beispiele zeigen wie diese verwendet wurden.

3.1 Spring-Boot MVC

Das Spring-Boot MVC Framework stellt eine Model-View-Controller (MVC) Architektur und fertige Komponenten bereit, mit denen flexible und lose gekoppelte Webanwendungen entwickelt werden können. Das MVC-Muster führt zu einer Trennung der verschiedenen Aspekte der Anwendung, während eine lose Kopplung zwischen diesen Elementen bereitgestellt wird.

3.2 Maven

Maven ist ein Build-Management-Tool der Apache Software Foundation und basiert auf Java. Dies hilft uns bei der Erstellung der ausführbaren .jar Datei. In unserem Fall haben wir die Informationen für unser Softwareprojekt in ein XML-File mit dem Namen pom.xml gepackt. Diese Datei enthält alle Informationen zum Softwareprojekt. Auf den genaueren Aufbau wird noch eingegangen.

3.3 Hibernate

Hibernate ist ein "Object-relational mapping" (ORM) -Tool. Object-relational mapping oder ORM ist eine Programmiermethode zum Zuordnen der Objekte zum relationalen Modell, wobei Entitäten/Klassen Tabellen zugeordnet

werde. Instanzen werden Zeilen zugeordnet und Attribute von Instanzen werden Tabellenspalten zugeordnet.
Es wird eine "virtuelle Objektdatenbank" erstellt, die innerhalb der Programmiersprache verwendet werden kann.
Näheres und Anwendungsbeispiele hierzu sind in der Beschreibung der Repositorys zu finden.

3.4 Thymeleaf

Thymeleaf ist eine moderne serverseitige Java-Template-Engine für Web- und Standalone-Umgebungen. Thymeleafs Hauptziel ist es, Templates in Ihren Entwicklungsworkflow zu bringen - HTML, das in Browsern korrekt dargestellt werden kann und auch als statischer Prototyp funktioniert, was eine stärkere Zusammenarbeit in Entwicklungsteams ermöglicht.
Mit Modulen für Spring Framework ist Thymeleaf ideal für unser Projekt in der HTML5 JVM-Webentwicklung. Mehr hierzu in den entsprechenden HTML-Dateien.

3.5 Bootstrap

Bootstrap ist ein Open-Source-Toolkit zur Entwicklung mit HTML, CSS und JS. Es enthält auf HTML und CSS basierende Vorlagen für Formulare, Buttons, Tabellen, Grid-Systeme, Navigations- und andere Gestaltungselemente. Bootstrap wird standardmäßig mit einem 940 Pixel breiten, zwölfspaltigen Grid-Layout ausgeliefert.
Wir verwendeten Bootstrap um unsere Website so zu gestalten, dass sie sowohl am PC wie auch am Smartphone/Tablet immer einwandfrei aussieht.

3.6 jQuery

jQuery ist eine schnelle, kleine und funktionsreiche JavaScript-Bibliothek. Mit einer einfach zu verwendenden API, die für eine Vielzahl von Browsern geeignet ist.

4 Application Properties File

Um unsere Anwendung lauffähig zu machen müssen vorerst einige Einstellungen vorgenommen werden.

```
# ===============================
# = DATA SOURCE
# ===============================
spring.datasource.url = jdbc:mysql://mydbcoolroof.c6n5ktd61o6t.eu-central-1.rds.amazonaws.com:3306/CoolRoofScheme
spring.datasource.username = masterCoolRoof
spring.datasource.password = DPifSP2017!!
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1

# ===============================
# = JPA / HIBERNATE
# ===============================
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

# ===============================
# = THYMELEAF configurations
# ===============================
spring.thymeleaf.mode=LEGACYHTML5
spring.thymeleaf.cache=false

# ============================================================
# = Spring Security / Queries for AuthenticationManagerBuilder
# ============================================================
spring.queries.users-query=select email, password, active from user where email=?
spring.queries.roles-query=select u.email, r.role from user u inner
join user_role ur on(u.user_id=ur.user_id) inner join role r on(ur.role_id=r.role_id) where u.email=?

# ================
# Load Static Content
# ================
spring.devtools.restart.exclude=static/**,templates/**
```

- **Properties**
  Verschiedenste Eigenschaften können in der application.properties/application.yml Datei angegeben werden

- **Data Source**
  Hier wird die Datenquelle für unsere Web Applikation eingebunden. Für unsere Anwendung haben wir Amazon AWS als Datenbank verwendet. Wichtig hier ist die URL, den Usernamen und das Passwort der Datenbank bereitzustellen.

- **JPA / HIBERNATE**
  Der Befehl spring.jpa.show-sql = true wird verwendet um SQL Statements zu loggen. Die zweite Zeile spring.jpa.hibernate.ddl-auto = update wird verwendet, um die Datenbank bei Modelländerungen automatisch auf dem neuesten Stand zu halten. D.h. es werden automatisch Spalten in die Datenbank eingefügt, falls das Datenmodell verändert

wird. Der dritte Befehl wird verwendet um Namen bei Datenbankanfragen automatisch mit "_" zu versehen, was in Spring Boot bei einigen Datenbank Anfragen benötigt wird. Der vierte Befehl ermöglicht Hibernate, SQL zu generieren, welches für ein bestimmtes DBMS optimiert ist

- **Thymeleaf configurations**
  Hier ist es wichtig den Mode auf „LEGACYHTML5" zu setzen um keine Probleme mit Thymeleaf in Kombination mit HTML5 zu bekommen.
- **Spring Security / Queries for AuthenticationManagerBuilder**
  Hier werden Queries für die Spring Security und den Authentication Manager Builder vordefinert.
- **Load Static Content**
  Diese Entwicklereinstellungen sind nützlich, um das Projekt in der Entwicklungsphase bei Änderung von statischem Inhalt nicht jedes Mal neu kompilieren zu müssen.

4.1 Application.yml

```yaml
facebook:
  client:
    clientId: 384261248599251
    clientSecret: fd7fa1c5f5a267f463263a0ce7ff2025
    accessTokenUri: https://graph.facebook.com/oauth/access_token
    userAuthorizationUri: https://www.facebook.com/dialog/oauth
    tokenName: oauth_token
    authenticationScheme: query
    clientAuthenticationScheme: form
  resource:
    userInfoUri: https://graph.facebook.com/me

google:
  client:
    clientId: 12894100090-tqso3lih5o42isneort886la2pesafmp.apps.googleusercontent.com
    clientSecret: 9xfU16efvxQ-BTMsXT9wOLpw
    accessTokenUri: https://accounts.google.com/o/oauth2/token
    userAuthorizationUri: https://accounts.google.com/o/oauth2/auth
    clientAuthenticationScheme: form
    scope: profile email
  resource:
    userInfoUri:  https://www.googleapis.com/oauth2/v3/userinfo
server:
  port: 3000
```

Diese Datei wird primär für unseren Social Media Login benötigt. Er beinhaltet diverse Einstellungen, welche von Facebook bzw. Google nach erfolgreicher Konfiguration zur Verfügung gestellt werden. Außerdem wurde hier der PORT festgelegt über welchen unsere Anwendung laufen soll.

5 Project Object Model (POM) File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.coolRoofs</groupId>
    <artifactId>coolRoofs</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>Cool Roofs</name>
    <description>Cool Roof Project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.security.oauth</groupId>
            <artifactId>spring-security-oauth2</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
```

Ein Project Object Model oder POM ist die grundlegende Arbeitseinheit in Maven. Es ist eine XML-Datei, die Informationen über die Projekt- und Konfigurationsdetails enthält, die von Maven zum Erstellen des Projekts verwendet werden. Es enthält Standardwerte für die meisten Projekte. Einige der Konfigurationen, die im POM angegeben werden können, sind die Projektabhängigkeiten, die Plugins oder Ziele, die ausgeführt werden können, die Erstellungsprofile und so weiter. Andere Informationen wie die Projektversion, Beschreibung, Entwickler, Mailinglisten und Ähnliches können ebenfalls angegeben werden.

6 SocialMediaController

```
/**
 * @param a principal object containing user data
 * @return the principal object
 * this method creates a new user in our AWS database
 * if a user tries to log in with his Facebook or Google account
 * and is not already registered at our application
 **/
@RequestMapping("/user")
public Principal user(Principal principal) {

    //we only get email, last name and name from social media account
    String email, lastName, name;

    if (principal == null)
        return principal;

    OAuth2Authentication auth = (OAuth2Authentication) principal;
    Object login = auth.getUserAuthentication().getDetails();

    @SuppressWarnings("unchecked")
    LinkedHashMap<String, String> userData = (LinkedHashMap<String, String>) login;
    auth.setAuthenticated(true);

    //sub is included in data which we get from Facebook account
    if (userData.get("sub") == null) {

        //user data for FACEBOOK account
        email = userData.get("id");
        email += "@socialMediaAccount.com";
        Object[] temp = userData.get("name").split(" ");
        name = temp[0].toString();
        lastName = temp[1].toString();

    } else {

        //user data for GOOGLE Account
        email = userData.get("sub");
        email += "@socialMediaAccount.com";
        name = userData.get("given_name");
        lastName = userData.get("family_name");
    }

    User userExists = userService.findUserByEmail(email);

    //if user not exists in database, add it
    if (userExists == null) {
        User newUser = new User();

        newUser.setEmail(email);
        newUser.setPassword("");
        newUser.setName(name);
        newUser.setLastName(lastName);
        newUser.setStreet("notdefined");
        newUser.setZipCode("notdefined");
        newUser.setCity("notdefined");
        newUser.setCountry("notdefined");
        newUser.setActive(1);
        newUser.setRole("notdefined");

        userService.saveUser(newUser);
    }

    return principal;
}
```

Dieser Controller verwaltet, wie der Name schon sagt, den Social Media Login. Er erhält im Prinzip vom Frontend ein Principal (represents the abstract notion of a principal, which can be used to represent any entity, such as an individual, a corporation, and a login id) Objekt, welches von den sozialen Medien nach dem Login zur Verfügung gestellt wird. Dieser Controller überprüft, ob sich der User zum ersten Mal auf unserer Seite einloggt oder ob er bereits in der Datenbank vorhanden ist. Es gibt zwei Optionen welche erfüllt werden:

1. Speichern des Users in die Datenbank und Rückgabe des User Objektes
2. Rückgabe des User Objektes, falls schon in Datenbank vorhanden

7 LoginController

Dieser Controller verwaltet den normalen (nicht social media) Login und die Registrierung von Nutzern. Im Prinzip legt der Nutzer über die Registration Page einen Account an. Die Zugangsdaten, sowie Art des Kontos und die getätigten Investments werden in der Datenbank gespeichert und bei Zugriffen auf die Seite(Login) von der Datenbank abgefragt.

```java
@RequestMapping(value = { "/login" }, method = RequestMethod.GET)
public ModelAndView login() {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("login");
    return modelAndView;
}

/**
 * mapping handler method for /registration
 *
 * @return modelAndView
 **/
@RequestMapping(value = "/registration", method = RequestMethod.GET)
public ModelAndView registration() {
    ModelAndView modelAndView = new ModelAndView();
    User user = new User();
    modelAndView.addObject("user", user);
    modelAndView.setViewName("registration");
    return modelAndView;
}

/**
 * mapping handler method for /registration handles the submission of a user
 * registration
 *
 * @return modelAndView
 **/
@RequestMapping(value = "/registration", method = RequestMethod.POST)
public ModelAndView createNewUser(@Valid User user, BindingResult bindingResult) {

    ModelAndView modelAndView = new ModelAndView();
    User userExists = userService.findUserByEmail(user.getEmail());

    if (userExists != null) {
        bindingResult.rejectValue("email", "error.user",
                "There is already a user registered with the email provided");
    }

    if (bindingResult.hasErrors()) {
        modelAndView.setViewName("registration");
    }else {
        userService.saveUser(user);
        modelAndView.addObject("successMessage", "User has been registered successfully");
        modelAndView.addObject("user", new User());
        modelAndView.setViewName("registration");

    }
    return modelAndView;
}
```

Als (eindeutigen) Benutzernamen wird die E-Mail Adresse des Nutzers verwendet. Das Passwort wird verschlüsselt in unserer Datenbank abgespeichert. Zusätzlich werden bei der Registrierung noch einige persönliche Informationen abgefragt.

8 Grundprinzip ModelViewController (MVC)

MVC ist ein bekanntes Programmierparadigma, bei dem zur besseren Aufteilung das Programm in 3 Programm-Komponenten geteilt wird.

- das Modell (Model)
  Dient zur Speicherung bestimmter Daten, z.B. zur Speicherung von Teilen des aktuellen Zustands der Anwendung.

- die Darstellung des Modells (View)
  Eine Ansicht (View) kann eine beliebige Ausgabe von Informationen sein, z. B. ein Diagramm, und dient damit zur Visualisierung.

- die Beeinflussung des Modells (Controller)
  Der Controller akzeptiert Eingaben und konvertiert sie in Befehle für das Modell (Model) oder die Ansicht (View).

Das Modell ist verantwortlich für die Verwaltung der Daten der Anwendung. Es antwortet auf die Anfrage von der Ansicht und antwortet auch auf Anweisungen von der Steuerung, um sich selbst zu aktualisieren.
Die Ansicht bedeutet die Darstellung von Daten in einem bestimmten Format, ausgelöst durch die Entscheidung eines Controllers, die Daten zu präsentieren.
Der Controller ist dafür verantwortlich, auf die Benutzereingabe zu antworten und Interaktionen an den Datenmodellobjekten durchzuführen.

9 MVC im Projekt

Um das Model-View-Controller-Konzept einzusetzen, werden grob gesagt vier "Klassenarten" benötigt: Controller, Model, Repositoriy und Service. Im Folgenden Abschnitt wird auf die einzelnen Bestandteile näher eingegangen, mit je einer allgemeine Erklärung und einem kleinen Beispiel, wie das ganze in unserer Implementierung funktioniert.

9.1 Controller

Einerseits steuern die Controller-Klassen quasi den "Flow" der Anwendung, also welche Seiten wann angezeigt werden, andererseits kann hier auch die nötige Logik implementiert werden.

Eingesetzt wird dafür die `springframework`-API.

Ein einfaches Beispiel einer typischen Controller-Funktion:

```java
@RequestMapping(value = "/login/roofSuccess", method = RequestMethod.GET)
public ModelAndView roofSuccess() {
    ModelAndView modelAndView = new ModelAndView();

    SecurityContextHolder.getContext().getAuthentication();

User user = lc.getLoggedInUser();

    if(!user.getRole().equals("roofOwner")) {
        modelAndView.setViewName("redirect:/login/failurePage.html");
        return modelAndView;
    }

    modelAndView.addObject("welcome", "Welcome " + user.getName() + " " + user.getLastName());

    return modelAndView;
}
```

Wann wird die Funktion aufgerufen? Dies wird über das `@RequestMapping` gesteuert. In diesem Fall also, wenn ein `GET` auf `/login/roofSuccess` angefragt wird.

Zuerst wird ein ModelAndView-Objekt erzeugt. Diesem können Informationen, die man anzeigen möchte, mitgegeben werden. In diesem einfachen Fall ist das der Name des Users, typischerweise sind das aber auch Model-Objekte, auf deren Attribute dann zugegriffen werden kann (via javascript und thymeleaf) oder die bei einer POST-Anfrage verändert werden können.

Zuvor überprüfen wir noch, ob der Nutzer eingeloggt ist und die richtige Rolle hat, also ob er überhaupt berechtigt ist, die Seite zu sehen. Wenn er das nicht darf, wird er nach /login/failurePage.html weitergeleitet.

Mit `@AutoWired` kann der Controller auf die Datenbank zugreifen und die damit verbundenen Methoden, die im Repository definiert wurden, benutzen.

```java
@Autowired
private RoofService roofService;
```

Das ganze wird als Attribut der Controller-Klasse definiert. Mehr Details zu den Repositories im dazugehörigen Kapitel.

Das dem ModelAndView-Objekt hinzugefügte Object kann dann im html file verwendet werden, in unserem Beipsiel einfach mit

```
<div class="logged-in-user"th:utext= "$welcome"></div>
```

Ein dem ModelAndView hinzugefügtes Roof-Objekt und dessen Felder kann am Frontend dann folgendermaßen verwendet werden:

```
th:object="$roof"
```

Und auf das "age" Feld des Roof-Objekts:

```
<div class="form-group">
<label th:if="${#fields.hasErrors('age')}" th:errors="*{age}"></label>
<label for="age">Age</label>
<select class="form-control" id="age" th:field="*{age}" required pattern="(0-9)">
  <div>
    <option disabled="true" selected="selected" value="">Select</option>
    <option value="0">0-5 years</option>
    <option value="5">5-10 years</option>
    <option value="10">10-15 years</option>
    <option value="15">15-20 years</option>
    <option value="20">20+ years</option>
  </div>
</select>
</div>
```

Eine vollständige Controller-Klasse sollte natürlich für jede mögliche `GET`- und `POST`-Anfrage eine dazugehörige Funktion haben, die steuert, was bei der jeweiligen Anfrage dem Nutzer angezeigt werden soll.

9.2 Model

Das Model stellt vereinfacht gesagt ein Objekt dar, dessen Attribute beispielsweise durch das Ausfüllen eines Formulars gesetzt werden können. Gleichzeitig werden diese Objekte auch zum Eintragen in die Datenbank genutzt.

```
@Entity
@Table(name = "user")
public class User {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = "user_id")
   private int id;

   @Column(name = "email")
   @Email(message = "*Please provide a valid Email")
   @NotEmpty(message = "*Please provide an email")
   private String email;

   @Column(name = "password")
   @Length(min = 5, message = "*Your password must have at least 5 characters")
   @NotEmpty(message = "*Please provide your password")
   @Transient
   private String password;
```

In diesem Beispiel definieren wir mit `@Table(name="user")`, dass ein User-Objekt einem Eintrag in unserer user-Tabelle der Datenbank entspricht. Selbiges natürlich mit `@Column` etc.

Diese Funktionalität wird von `javax.persistence.*`-API bereitgestellt. Eine weitere nützliche Funktion ist `@GeneratedValue(strategy = GenerationType.AUTO)`. Hier werden automatisch eindeutige Zahlen generiert, wie wir sie beispielsweise für die User-id verwenden.

Zusätzlich findet man in der `hibernate.validator.constraints`-API einige sehr nützliche Funktionen:

- `@Length` erlaubt, die Länge des Strings zu beschränken. So fangen wir zu kurze Passwörter ab.
- `@NotEmpty` stellt sicher, dass der übergebene String nicht leer ist.
- Mit `@Min(value)` kann festgesetzt werden, dass einem Integer (beispielsweise) nur Zahlen größer gleich `value` zugewiesen werden können.
- Weitere Funktionen: `@Email`, `@URL`, ...

Mit diesen Constraints kann man sehr einfach am Backend sicherstellen, dass nur erlaubte/gewünschte Werte in die Datenbank gelangen.

Außerdem können direkt, wenn ein Input einen Constraint nicht erfüllt, Nachrichten übergeben werden, die dann beispielsweise der Nutzer sieht und erfährt, wo er einen Fehler gemacht hat. Zum Beispiel mit

`@Email(message="Please provide a valid Email").`

Am Front-End wird das dann folgendermaßen ausgegeben:

```
<!-- email -->
<div class="registration-fields">
    <div class="field-label">Email:</div>
    <div class="field-input">
        <input type="text" th:field="*{email}" placeholder="Email" class="form-control" />
    </div>
</div>
<div class="registration-fields" style="margin-top:0px!important" th:if="${#fields.hasErrors('email')}">
    <div class="field-label"></div>
    <div class="field-error" th:errors="*{email}"></div>
</div>
<!-- end email -->
```

Ansonsten benötigt man in den Model-Klassen nur noch getter- und setter-Methoden.

9.3 Repository & Service

Mit den Repository-Interfaces wird JpaRepository erweitert. Die Repositories sind quasi die echte Schnittstelle zwischen Anwendung und Datenbank. Mit ihrer Hilfe werden die Queries erzeugt und beim Aufruf der entsprechenden Methode ausgeführt.

JpaRepositories stellt viele Teile von Datenbankabfragen bereit. Einige Beispiele sind hier aufgelistet:

Keyword	Sample
`And`	`findByLastnameAndFirstname`
`Or`	`findByLastnameOrFirstname`
`Is,Equals`	`findByFirstname,findByFirstnameIs,findByFirstnameEquals`
`Between`	`findByStartDateBetween`
`LessThan`	`findByAgeLessThan`
`LessThanEqual`	`findByAgeLessThanEqual`
`GreaterThan`	`findByAgeGreaterThan`
`GreaterThanEqual`	`findByAgeGreaterThanEqual`
`After`	`findByStartDateAfter`
`Before`	`findByStartDateBefore`
`IsNull`	`findByAgeIsNull`
`IsNotNull,NotNull`	`findByAge(Is)NotNull`
`Like`	`findByFirstnameLike`
`NotLike`	`findByFirstnameNotLike`
`StartingWith`	`findByFirstnameStartingWith`

`https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/jpa.repositories.html` (Abschnitt 2.3.2)

Man kann sich damit die benötigten Methoden zusammenstellen, zum Beispiel im Format `find[All][Entity]By[Attribute]`.

Eine Beispiel aus unserem Projekt sind die drei Klassen RoofService + RoofServiceImplementation + RoofRepository.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.coolroof.model.Roof;

@Repository("roofRepository")
public interface RoofRepository extends JpaRepository<Roof, Long> {

    List<Roof> findAllRoofsByUserId(int uId);
    List<Roof> findAll();
}
```

Mit `@Repository` wird festgelegt, dass es ein Repository ist. Auf dieses kann, wie man beispielsweise in den Controllern sieht, über `@AutoWired` zugegriffen werden.

```
public interface RoofService {

    public List<Roof> findAllRoofsByUserId(int uId);

    public void saveRoof(Roof roof);

    public List<Roof> findAll();

}
```

In diesem Interface werden die Methoden festgelegt, die dann in der RoofServiceImplementation implementiert werden.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.coolroof.model.Roof;
import com.coolroof.repository.RoofRepository;

@Service("roofService")
public class RoofServiceImpl implements RoofService {

    @Autowired
    private RoofRepository roofRepository;

    @Override
    public List<Roof> findAllRoofsByUserId(int uId) {
        return roofRepository.findAllRoofsByUserId(uId);
    }

    @Override
    public void saveRoof(Roof roof) {
        roofRepository.save(roof);
    }

    @Override
    public List<Roof> findAll() {
        return roofRepository.findAll();
    }
}
```

Hier in der Implementierung ist schön zu sehen, wozu das Repository gut ist und wie man es benutzt. Da JpaRepository die Funktionen bereits bereitstellt, muss in der Implementation-Klasse nichts weiter getan werden als mit `@AutoWired` die Verbindung zum Repository herzustellen, und dann die passenden Funktionen aufzurufen.

10 Exception Handling in Spring-Boot MVC

Spring MVC bietet mehrere komplementäre Ansätze zum Exception Handling, hier werden wir nur kurz auf die wichtigsten Methoden eingehen um ein Grundverständnis zu erlangen.
Wir werden nun die verschiedenen verfügbaren Optionen zeigen. Unser Ziel ist es, Exceptions nicht explizit in Controller-Methoden zu behandeln, wo dies möglich ist. Sie werden besser in einem eigenen Code behandelt.
Die drei Optionen sind:

- per Exception

- per Controller
- oder global.

10.1 HTTP Status Codes

Normalerweise verursacht jede unbehandelte Ausnahme, die bei der Verarbeitung einer Webanforderung ausgelöst wird, dass der Server eine HTTP 500-Antwort zurückgibt. Jede Ausnahme, die man selbst schreibt, kann jedoch mit der Annotation @ResponseStatus versehen werden. Wenn eine mit Anmerkungen versehene Ausnahme von einer Controller-Methode ausgelöst wird und an keiner anderen Stelle behandelt wird, wird automatisch die entsprechende HTTP-Antwort mit dem angegebenen Statuscode zurückgegeben.

10.2 Controller Based Exception Handling

Man kann jedem Controller zusätzliche (@ExceptionHandler)-Methoden hinzufügen, um speziell Ausnahmen zu behandeln, die von Request-Handling-Methoden (@RequestMapping) im selben Controller ausgelöst werden. Solche Methoden könnten sein:

- Ausnahmen ohne die @ResponseStatus-Annotation (normalerweise vordefinierte Ausnahmen, die man nicht geschrieben hat) behandeln.
- Den Benutzer in eine spezielle "error view" umleiten.
- Eine vollständig benutzerdefinierte " error response " erstellen.

10.3 Global Exception Handling

Ein "controller advice" ermöglicht es, genau die gleichen Exception-Handling-Techniken zu verwenden, sie aber auf die gesamte Anwendung anzuwenden, nicht nur auf einen einzelnen Controller. Man kann sich diese als "annotation driven interceptor" vorstellen.
Jede mit @ControllerAdvice annotierte Klasse wird zu einem Controller-Advice und drei Arten von Methoden werden unterstützt:

- Exception-handling Methoden, die mit @ExceptionHandler kommentiert sind.
- Model enhancement methods (zum Hinzufügen zusätzlicher Daten zum Modell), die mit @ModellAttribut kommentiert sind. Zu beachten ist dabei, dass diese Attribute für die "exception handling views" nicht verfügbar sind.

- Binder-Initialisierungsmethoden (zur Konfiguration der Formularverarbeitung) mit @InitBinder Anmerkungen versehen.

11 Frontend Javascript

JavaScript ist eine Skriptsprache, die ursprünglich 1995 von Netscape für dynamisches HTML in Webbrowsern entwickelt wurde, um Benutzerinteraktionen auszuwerten, Inhalte zu verändern, nachzuladen oder zu generieren und so die Möglichkeiten von HTML und CSS zu erweitern. Heute findet JavaScript auch außerhalb von Browsern Anwendung, so etwa auf Servern und in Microcontrollern.

Bei der Programmierung einer Webanwendung kommt man fast nicht um Javascript herum. Der Javascript Code liegt in src/main/resources/static/javascript und ist in den Files "make_investment.js", "my_investment" und "add_roof.js". (siehe Projekt Ordner)

Für das CoolRoof Projekt brauchen wir Javascript hauptsächlich für die Google Maps Integration, Kommunikation mit dem Backend und zum dynamischen Ändern des angezeigten Webinhaltes.

```
//this function finds roof by ID
function findRoofById(id) {
    for(var i=0;i < roofList.length; i++) {
        if(id === roofList[i].roofId) {
            return roofList[i];
        }
    }
};

//this function updates the dropDown menue after each changed investment
function updateDropDownMenue(dropDown) {

    for(var i = 0; i < selectedRoofList.length; i++) {
        var option = document.createElement("option");
        option.value = selectedRoofList[i].roofId;
        //TODO: change the name in drop down menue
        option.text = 'Roof ' + i;
        dropDown.add(option, null);
    }
};

//this function clears the dropDown menue before each changed investment
function clearDropDownMenue(dropDown) {

    var i;
    for(i = dropDown.options.length - 1 ; i >= 0 ; i--) {
        dropDown.remove(i);
    }
};
```

Funktionen um Dropdown-menüs Dynamisch zu verändern und eine Funktion um eine Liste aus Dächern zu durchsuchen.

```
function initMap() {

        var selectedRoof = document.getElementById("selectedInvestment").value;

        //get attributes for selectedMap
        longitude = findRoofById(parseInt(selectedRoof)).longitude;
        latitude = findRoofById(parseInt(selectedRoof)).latitude;
        zoomfactor = findRoofById(parseInt(selectedRoof)).zoomFactor;
        jsonString = findRoofById(parseInt(selectedRoof)).roofPolygon;

        map = new google.maps.Map(document.getElementById(mapName), {
            center: { lat: parseFloat(latitude), lng: parseFloat(longitude) },
            zoom: parseInt(zoomfactor),
            //only show roadmap type of map, and disable ability to switch to other type
            mapTypeId: google.maps.MapTypeId.SATELLITE,
            mapTypeControl: false,
            tilt:0,
            streetViewControl: false
        });
        //disable edit functions
        map.data.setStyle({
            editable: false,
            draggable: false
        });

        //loads polygon
        loadPolygons(jsonString);

        //update fields
        updateFields(document.getElementById("selectedInvestment").value, document.getElementById("selectedSpace").value);
};
```

Google Maps Javascript Function

12 Datenbankmodel

Als Anbieter für unsere Datenbank wählten wir AWS (AmazonWebServices). Hier gibt es spezielle Konten für Studenten, die es uns so möglich machten, alles zu benutzen was wir brauchten. Unser Datenbankmodell ist eine Relationale Datenbank, die MySQL unterstützt.

12.1 RDS MySQL

MySQL ist eine beliebte relationale Open-Source-Datenbank mit einfacher Skalierbarkeit.
Für die Organisation der Daten verwenden wir 5 verschiedene Tabellen.

- Investment
  Beinhaltet Informationen über ein getätigtes Investment. (Fläche, Ort etc.)
- Role
  Dient dazu um festzustellen, welche Rolle ein Benutzer hat.
- Roof
  Beinhaltet Informationen über das Dach selbst. (Preis, Region etc.)

- User
  Beinhaltet Informationen über den Benutzer. (Benutzername, Email etc.)

- User_Role
  Weist User ID einer Role ID zu.

Details hierzu sind in den jeweiligen Tabellen nachzusehen.

•Investment

#	investment_id	co2saved	investment_date	magnitude	roof_ids	rtn_of_interests	space	user_id	values_of_investment
1	1	0	2018-02-01 16:42:22	170	8,10;	0	10	153	NULL
2	2	0	2018-02-01 16:44:00	2890	8,170;	0	170	153	NULL
3	3	0	2018-02-01 16:44:33	6800	9,383;10,17;	0	400	153	NULL

•investemt_id: Eindeutige ID vom Investment
•co2saved: Int Wert; wie viel CO_2 in Tonnen gespart wurde
•investment_date: Datum, zu dem das Investment getätigt wurde
•magnitude: Wert des Investments in $
•roof_id: Fremdschlüssel der Dächer
•space: Fläche in m^2
•user_id: ID des Users

•Role

#	role_id	role
1	1	roofOwner
2	2	investor
3	3	notdefined

•role_id: ID der Role
•role: Welche rolle der Account hat

•Roof

#	roof_id	age	area	area_left	latitude	longitude	material	price_per_sqm	region
1	8	0	180	0	47.781522340710374	13.04420605301857	Shingle	17	Europe
2	9	5	383	0	-33.86910270160048	151.20890758931637	Shingle	17	Austral...
3	10	0	314	276	-33.880932092717465	151.23356111347675	Shingle	17	Austral...

•roof_id: ID des Daches
•age: Alter des Daches (0,5,10,15, >20 Jahre)
•area: Gesamtfläche des Daches
•area_left: Fläche des Daches, in die noch nicht investiert wurde
•latitude: Breitengrad der Position des Daches

•longitud: Längengrad der Position des Daches
•material: Material des Daches
•price_per_sqm: Preis pro Quadratmeter in $
•region: Region, in der sich das Dach befindet

roof_polygon	time_of_investment	user_id	zoom_factor
{"type":"FeatureCollection","feature…	2018-02-01 16:44:00	154	19
{"type":"FeatureCollection","feature…	2018-02-01 16:44:33	154	20
{"type":"FeatureCollection","feature…	NULL	154	20

•roof_polygon: Punkte des Daches in Form eines Strings
•time_of_investment: Datum, zu dem das Investment getätigt wurde
•user_id: ID des Users
•zoom_factor: Zoom Faktor für die Google-Maps Anzeige

•User

#	user_id	active	email	last_name	name	password
1	152	1	MoserM89@gmx.at	Moser	Michael	$2a$10$J7QnSqpnkYeb1UBT4TUr…
2	153	1	1640028136017473@socialMedia…	Moser	Michael	$2a$10$/lBK/GRqYLSK34zQXjlDp…
3	154	1	113688098113480554787@social…	Moser	Michael	$2a$10$SfBHlsUphyVi.XWzDlwzS…

•user_id: ID des Users
•active: Ob noch aktuell
•emai: E-Mailadresse des Benutzers
•last_name: Nachname des Benutzers
•name: Vorname des Benutzers
•password: Hashwert des Passwortes des Benutzers

role	country	city	street	zip_code
investor	Österreich	Gampern	Schwarzmoos 23	4851
investor	notdefined	notdefined	notdefined	notdefined
roofOwner	notdefined	notdefined	notdefined	notdefined

•role: Rolle des Benutzers
•country: Land des Benutzers
•city: Stadt des Benutzers
•street: Straße des Benutzers
•zip-code: zip-code des Benutzers

•User_Role

#	user_id	role_id
1	154	1
2	152	2
3	153	2

•user_id ID des Benutzers
•role_id: Rolle des Benutzers