ImplementationProject - OpenData

Lucia-Eve Berger (lucia.berger@mail.mcgill.ca)
European Masters in Software Engineering
code: `https://github.com/luciaeveberger/open_visualization/`

May 1, 2020

Contents

Part I

Application Domain:

Solution. The environmental application relies on the design of a secure API. Using the local data as well as refreshed real time data, the user can perform standard visualization of charts and request custom features. The application focuses on Bolzano environmental data, so it could used for tourists, environmentalists or others.

Part II

Design Spec - Overview of the key features

- Data visualization selection API (Java spring)
- User management API (registration, user profile, login, logout, sessions) (Java Spring Security)
- Data export (csv) (JQuery)
- User preferences datasets (liking, requests for new features) (Cart)
- Security features (Java Spring Security)

Part III

Technologies and the overall architecture

0.1 Frontend

The open data application uses HTML, CSS, jQuery, and HighCharts.js library for the charting the data. The CSS implements a grid for displaying the entry (bootstrap). jQuery is implemented for front-end interactions as well at the AJAX calls. The AJAX calls are used to render the original data that is then plotted by HighCharts after creating a series object. (Dictionary).

0.2 Backend

Java (REST protocol with json payloads) with a MYSQL database. The backend uses Java Controllers and Models to serve as endpoints. Querying on the database is done with Java Data and MYSQL queries on a MYSQL database.

0.3 Dependencies

Using maven, the dependencies of the Java application are managed by the *pom.xml* file. The Java project uses spring-boot-starter-parent, spring-data, mysql-connector-java, thymeleaf-layout-dialect and thymeleaf-extras-springsecurity4. Each of these dependencies is managed by Java Maven. The Maven framework allows for easy install and easy deployment in different environments. Dependencies are added with a new entry formatted in xml.

For example, the MYSQL connector that we explored in class is sourced with the $< dependency >$ tag and is served at runtime (Figure 1).

The application is run with

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figure 1: XML dependency

Architecture Diagram

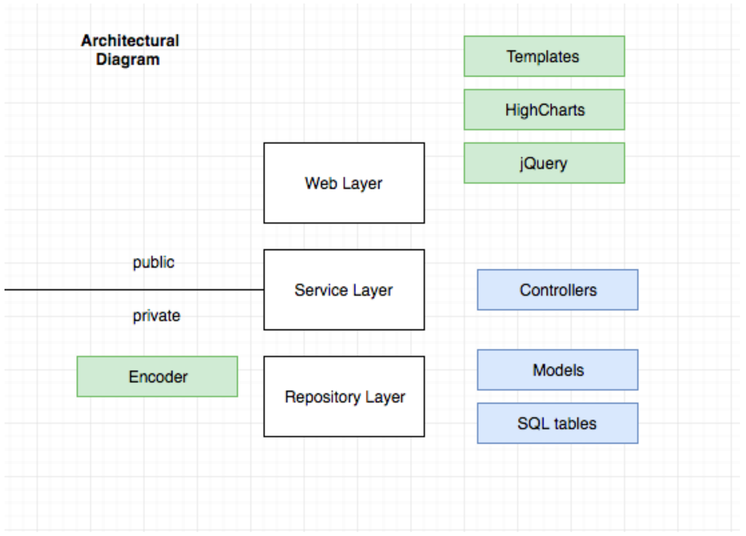


Figure 2: The Spring Java architecture showing the layers

```
mvn clean install && mvn spring-boot:run
```

Part IV
System Architecture:

0.4 Application Structure

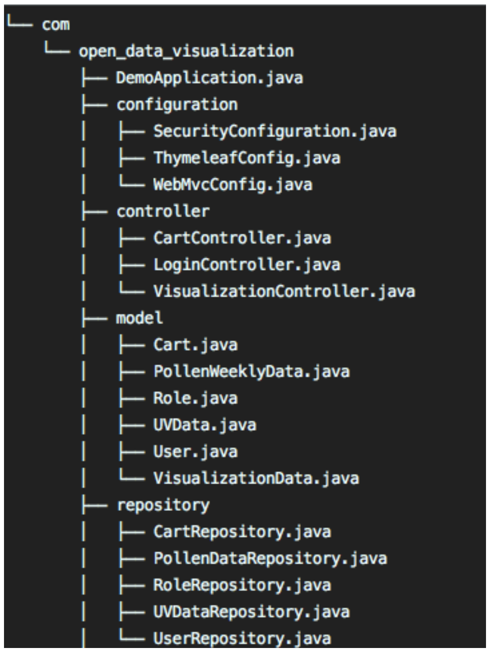
The Open Data application follows a Model-View-Controller Pattern in Java Spring. As Figure 3 shows, each of the function groups or tables have a separate model though they may share a controller. For example, the visualization data is both passed through the *VisualizationController*. The below section describes each function (Model, Controller, and View).

0.5 Database Design (Models)

The database is written in MYSQL, with Java Spring Model overlay. The Tables are USER, ROLES, UVDATA, POLLENWEEKLYDATA, and CART. On each of the tables the column types are specified in the model layer. An example on the POLLENWEEKLYDATA is shown in figure 5.

The database is populated with MYSQL scripts for the data visualization tables. The USER and ROLE table are interacted with via a service. The details of which will be explained in the Data Access portion.

MVC pattern with Javaspring



```
└── com
    └── open_data_visualization
        ├── DemoApplication.java
        ├── configuration
        │   ├── SecurityConfiguration.java
        │   ├── ThymeleafConfig.java
        │   └── WebMvcConfig.java
        ├── controller
        │   ├── CartController.java
        │   ├── LoginController.java
        │   └── VisualizationController.java
        ├── model
        │   ├── Cart.java
        │   ├── PollenWeeklyData.java
        │   ├── Role.java
        │   ├── UVData.java
        │   ├── User.java
        │   └── VisualizationData.java
        ├── repository
        │   ├── CartRepository.java
        │   ├── PollenDataRepository.java
        │   ├── RoleRepository.java
        │   ├── UVDataRepository.java
        │   └── UserRepository.java
```

Figure 3: Screen-grab tree in the terminal

Database Design with a diagram

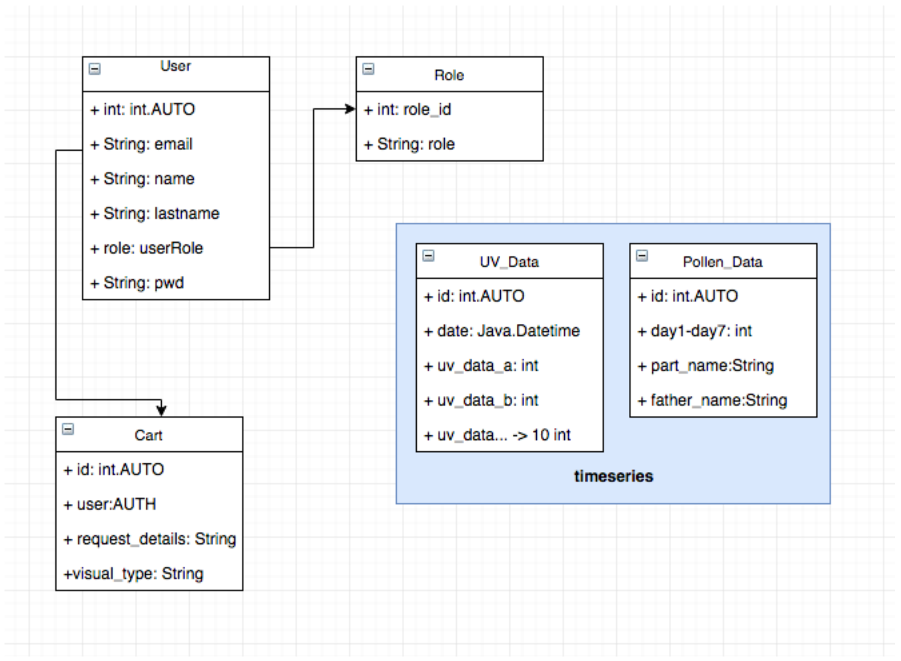


Figure 4:

Entity model Code

```
@Entity
    @Table(name = "pollen_weekly_data")
    public class PollenWeeklyData {

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

    @Column(name="STAT_ID")
    private int STAT_ID;

    @Column(name="STAT_CODE")
    private String STAT_CODE;
```

Figure 5:

0.6 Data Access

0.6.1 Visualization Tables

As in Java, the column data are accessed and altered through getters and setters methods on the model objects. The getters and setters allow bidirectional data transfer.

```
public String getSTAT_NAME_D() {
return STAT_NAME_D;
}
```

Following the Spring implementation, CRUD methods are accessed on the repository level from the Controller. The JpaRepository provides out of box querying methods.In this application, the most common querying method is $findAll()$ which maps to the MYSQL query of $SELECT * FROMTABLE$. The application makes use of both out-of-the-box queries and custom queries. An example of a custom query is below. The custom queries are used on the CART Table and enforce the security parameters.

```
  @Query("select u from Cart u where u.username = ?1")
    List<Cart> findByUsername(String username);
```

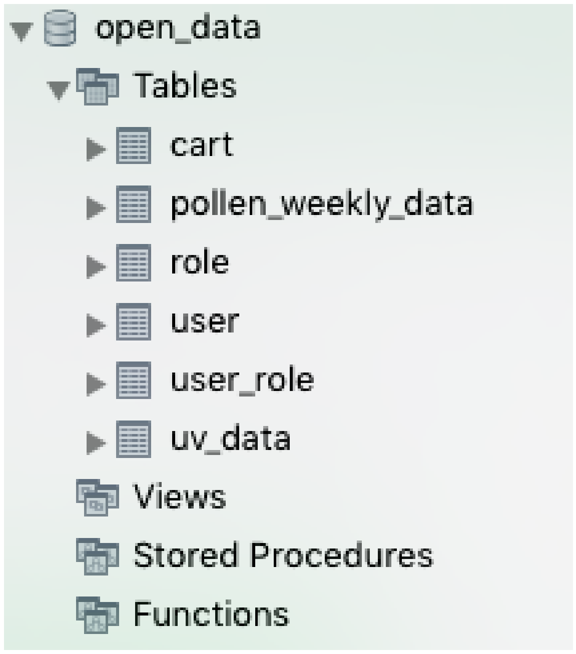
By using the Query Annotation, SQL query statement can be executed on the database. The database is organized in the below tables.

0.7 Controllers

The controllers use the @$RequestMapping$ feature to make the endpoints. Using the servlet library, the controllers use $ModelAndView$ to render the models into views that can be accessed by the front-end.

```
@RequestMapping(value={"/", "/login"}, method = RequestMethod.GET)
```

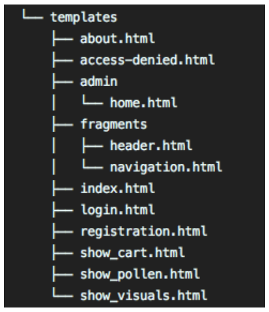
OpenVisualization tables



```
open_data
  Tables
    cart
    pollen_weekly_data
    role
    user
    user_role
    uv_data
  Views
  Stored Procedures
  Functions
```

Figure 6: Screen-grab from MySQLWorkBench

Collection of the Views



```
└── templates
    ├── about.html
    ├── access-denied.html
    ├── admin
    │   └── home.html
    ├── fragments
    │   ├── header.html
    │   └── navigation.html
    ├── index.html
    ├── login.html
    ├── registration.html
    ├── show_cart.html
    ├── show_pollen.html
    └── show_visuals.html
```

OpenVisualization landing page

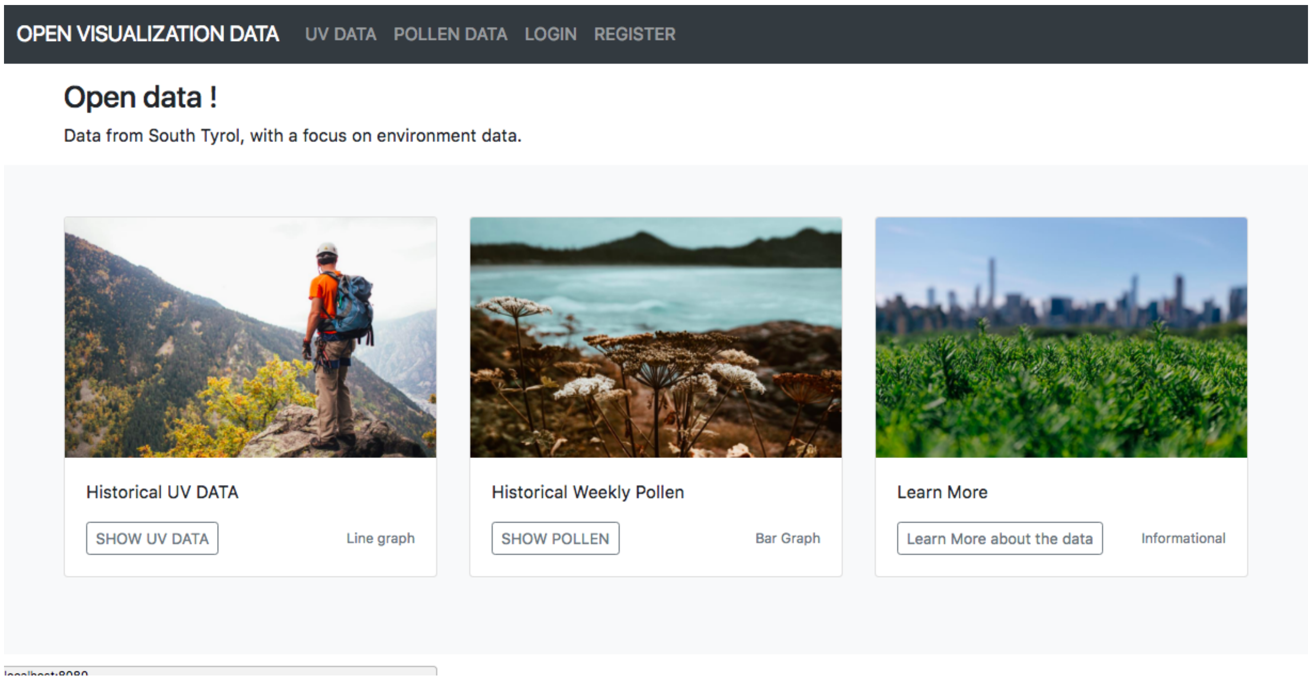


Figure 7:

0.8 Templates and Views

The views of the site were all configured by Java's *@RequestMapping* routes. For many of these routes, a template in HTML was rendered with the corresponding name as the request mapping endpoint.

For the formation of the templates, thymeleaf partials were used. Rather than copying the code for the header and navigation to each of the templates, I created fragments. Each of these fragments were imported into each template. By using partials, the logic (i.e. logged in/not logged in) was isolated only to the top level fragments. It also reduced the amount of code copying. The below code was used in each template.

```
<head th:replace="fragments/header :: header"> </head>
<body class="set_height">
<div th:replace="fragments/navigation :: navigation"></div>
```

An example template (route index) is shown below.

Part V
Functionalities

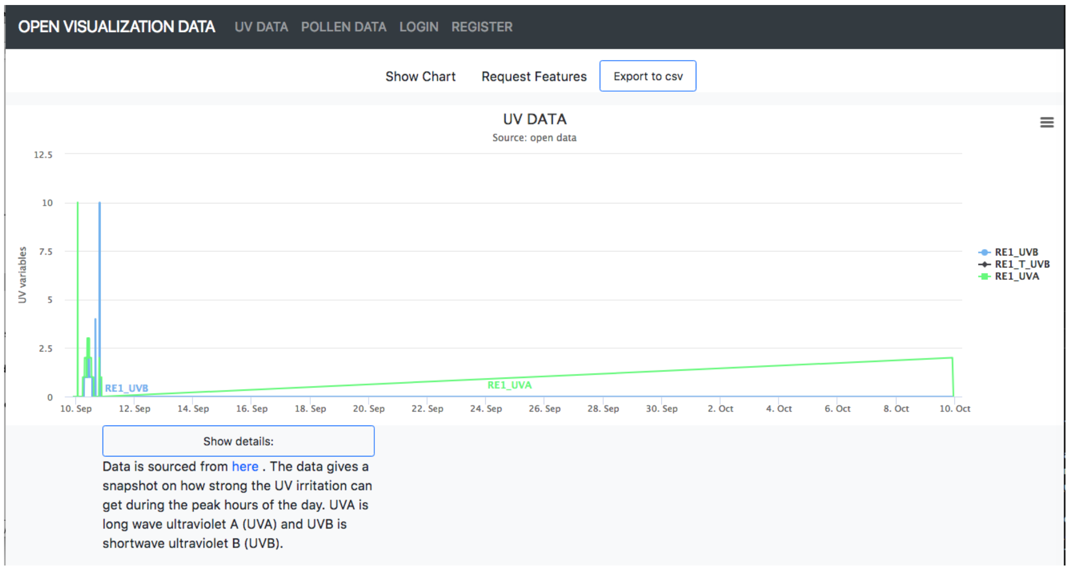
0.9 JQuery-driven

JQuery is used through the application with several functions.

- **Ajax calls:** The JQuery Ajax calls are used for *get*. The *get* functionality calls a java endpoint.

```
$.ajax({url: "/all_uv_data", success: function(result) {
```

UV Functionalities



Data is sourced from here . The data gives a snapshot on how strong the UV irritation can get during the peak hours of the day. UVA is long wave ultraviolet A (UVA) and UVB is shortwave ultraviolet B (UVB).

Figure 8: The UV data showing the functionality

On the success callback of the function, the data is injected into the template and the HighCharts graphic is made from it. This functionality is used in the UV and Pollen Data instance.

- **OnReady():** Throughout each of the jQuery instances, the document ready is used. This ready functionality checks if jQuery is added as a dependency and insures dependencies are loaded.

```
$(document).ready(function(){
```

- **IDClick():** The ID-click functionality is used throughout the html ID's.

```
$("#export_csv").click(function()
```

  An example of the ID-click functionality is the export CSV. The click of the button initiates the creation of a CSV blob and wraps the data response body into a CSV object.

- **Hide/Show functionality:** The hide and show functionality was used to keep parts of the html body hidden at site rendering. For example, the table portion of the site was hidden on default with.

```
 $("#table").hide();
```

0.10 HighCharts

The highcharts javascript library is used to chart the data points. Upon the AJAX request, the data is injected into a series object. The series object is a list of dictionaries. Each of the dictionaries has the name of the item and it's captured data. An empty series is shown below:

```
var series = [{
      name:"RE1_UVB",
      data: []
      },
      {
          name:"RE1_T_UVB",
```

Pollen bar graphic, highcharts

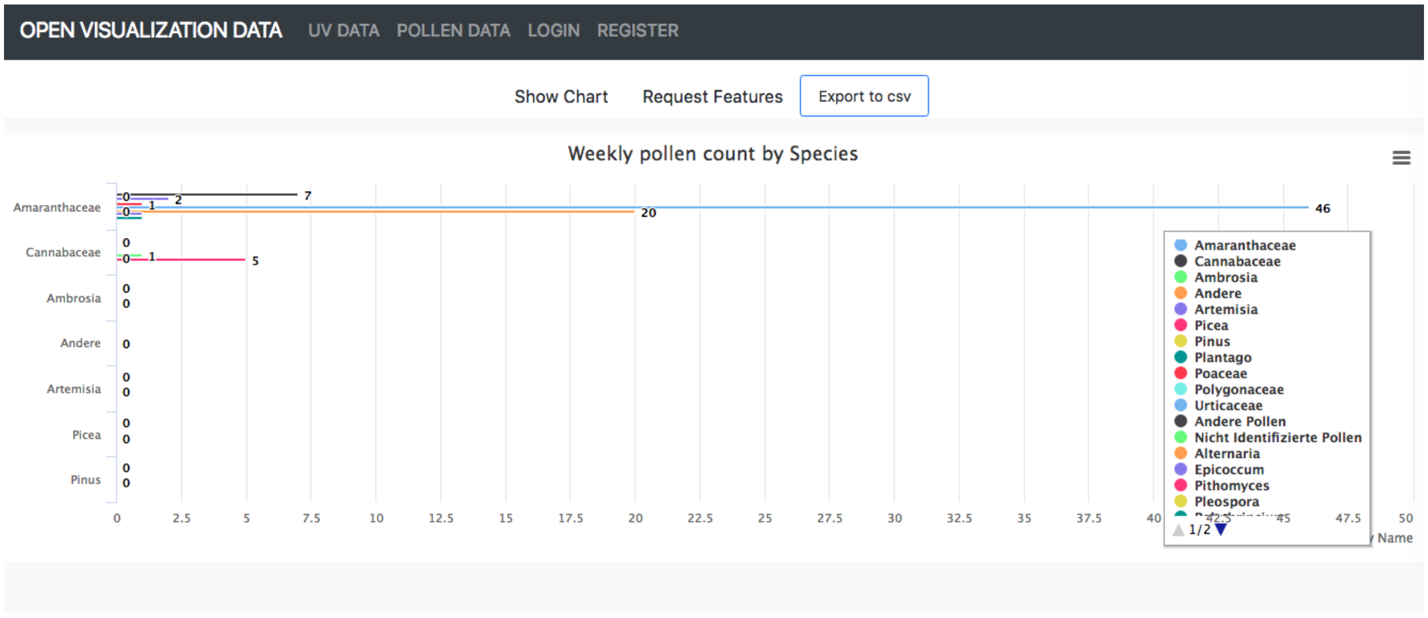


Figure 9: Pollen data

```
            data: []
        },
        {
            name:"RE1_UVA",
            data: []
        }
    ];
```

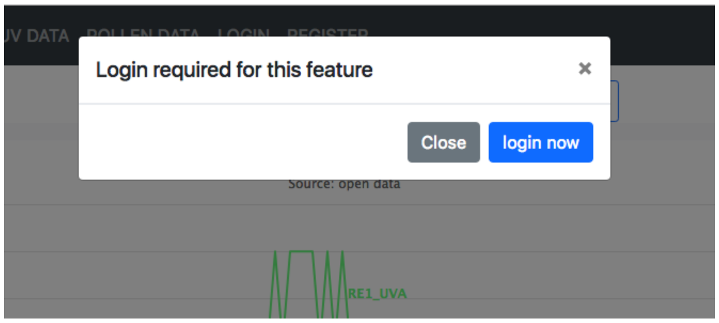
On the callback of the AJAX, the series is populated then the Highcharts chart is configured to an ID. The Highcharts api allows many different parameters. In this application, the responsive parameter as well as the line and bar chart were used. The figure below shows the bar graph of the sample data from OpenData. In further exploration, I would like to use some of the other features as there are many referenced.

0.11 CSS

As mentioned above, bootstrap4 is used in the application. This decision was taken to use the already nicely designed features and responsiveness of the framework. It was supplemented with some local feature css attributes. The following features were used:

- **Grid:** The grid feature allowed for rows of columns. The columns are each responsive (as seen in the templates above)
- **Button Design:** The buttons from bootstrap were quick to implement and aesthetically pleasing. The modifying their colours was easy.
- **Modals Popouts:** As shown in the figure above, the bootstrap modal popout is shown. This popout indicates the user does not have sufficient accesses to a particular feature (In this case, the access to custom features without being logged in).

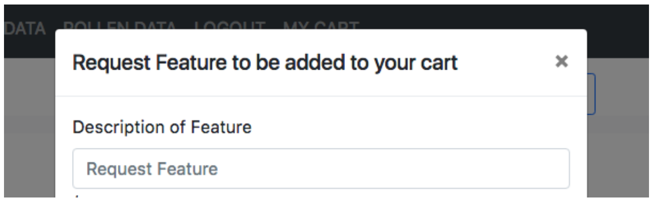
Bootstrap modal popout



Login required for this feature

Figure 10: Indicating access to page

Request Features



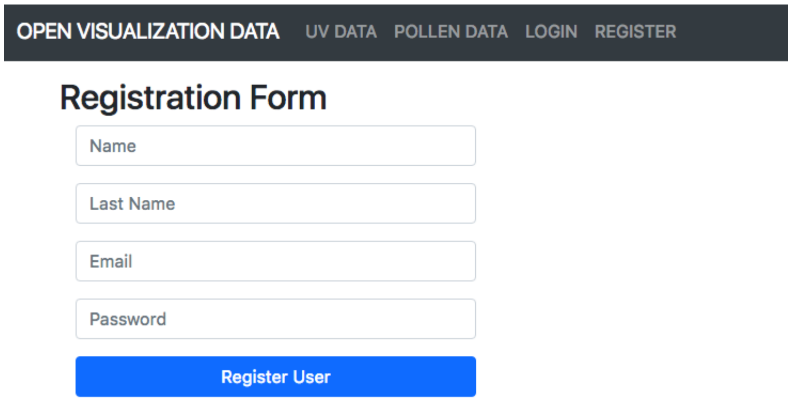
Request Feature to be added to your cart

Description of Feature

Request Feature

Figure 11: Text field to show request

Registration



Registration Form

Name

Last Name

Email

Password

Figure 12: Indicating access to registration

0.12 Login/Registration

The user can register securely on the site and then login with their credentials. When they register, they can add their details on the screen. As mentioned in the security portion, their information is validated before being posted to the backend. Validation includes :

- Does not match other users
- Does not include SQL query
- Password field matches criteria (length).

0.13 Cart features

The cart is a table in the database that is accessed only by a logged-in user. The cart Object has the following features:

```
public Cart(){}
public Cart(String username, String contents, String userRequest){
this.username = username;
this.contents = contents;
this.user_request = userRequest;
}
```

The specific features of the Cart are:

- **Request New Features:** The request new features is an endpoint on the Cart Controller. It takes the user instance and creates a new Cart Object with the userRequest field.
- **Add/Remove from Cart:** Similar to the above, the add and remove from the cart allow a user to add a dataset to the cart or remove it.

Part VI
Performance Analysis:

The app loads quickly. To help the performance of the application:

- Minimized the size of the photos
- Lazy loaded the JQuery if they were specific to one page
- Minified the CSS from bootstrap

Part VII
UX/UI Analysis

The application is responsive and renders well at the mobile and the tablet range. The figure shows the mobile view. The breakpoints are set as we did in the exercise and the nav becomes vertical. I decided to keep it open as it was a sticky nav and should always be present to the user.

Mobile view

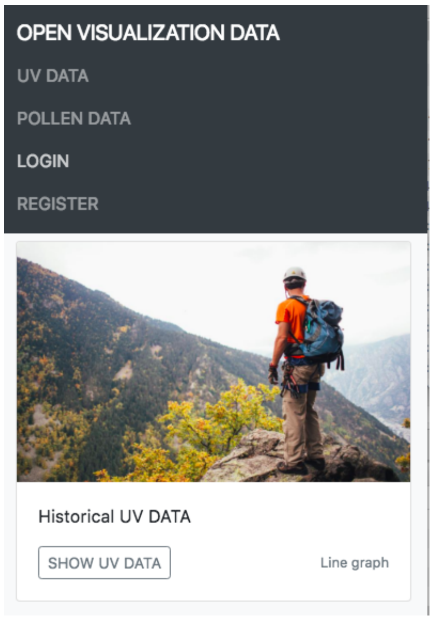


Figure 13: showing the responsive navigation

Part VIII

Security Analysis:

0.14 User Information Security

To ensure the user's security of information in the app, encoding, validators and HTTP security measures are used.

- For passwords within the app, encoding is done through the *BCryptPasswordEncoder* Library.
- To prevent SQL injections, input provided by the user is validated by the templating validator. In this way, before controller receives messages they have to be valid or they will not be passed through the application backend. The same is true on the requests to the cart.
- Figure shows the HTTP security authorization. Using antMatchers, the application routes are permissioned based on the ROLE or login. The *permitAll*() method permits full use while the *hasAuthority*() validates the appropriate accesses. This access structure protects the user's information, particularly their cart choices.

□

0.15 Application Security

Configuration for the application is stored in the applications.properties file. This file is configured with environmental variables, which are not included in the github but rather depend on a hidden file that configures the database.

```
protected void configure(HttpSecurity http) throws Exception {
http.
authorizeRequests()
.antMatchers("/").permitAll()
.antMatchers("/login").permitAll()
.antMatchers("/index").permitAll()
.antMatchers("/show_visuals").permitAll()
.antMatchers("/show_pollen").permitAll()
.antMatchers("/all_uv_data").permitAll()
                .antMatchers("/all_pollen_data").permitAll()
.antMatchers("/about").permitAll()
.antMatchers("/registration").permitAll()
.antMatchers("/admin/**").hasAuthority("ADMIN").anyRequest()
.authenticated().and().csrf().disable().formLogin()
.loginPage("/login").failureUrl("/login?error=true")
.defaultSuccessUrl("/index")
.usernameParameter("email")
.passwordParameter("password")
.and().logout()
.logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
.logoutSuccessUrl("/").and().exceptionHandling()
.accessDeniedPage("/access-denied");
}
```

Figure 14: AntMatcher Http Request Processor

```
spring.datasource.url = {DB_URL}
spring.datasource.username = {USR_NAME}
spring.datasource.password = {PWD}
spring.datasource.testWhileIdle = true
```

Figure 15: Application DB environmental variables

Part IX
Lessons Learned

As it was my first time working with Spring Java framework, I struggled at first how to configure the application. I found there was a lot of documentation online that I followed. I was able to use the templating feature fast from what we learned in class. I was familiar with SQL before so it was quick to implement the databases. The frontend was more challenging deciding on the asthetic. I took photos from an open-source library to help the environmental feel overall. Overall, it was a great learning experience to focus on this application.