



# Portfolio

## REPORT

C21116175 | CMT652 Web Application | 5/01/2022

## UTILISING THE CLIENT-SIDE LANGUAGES TO IMPACT THE DYNAMIC WEB

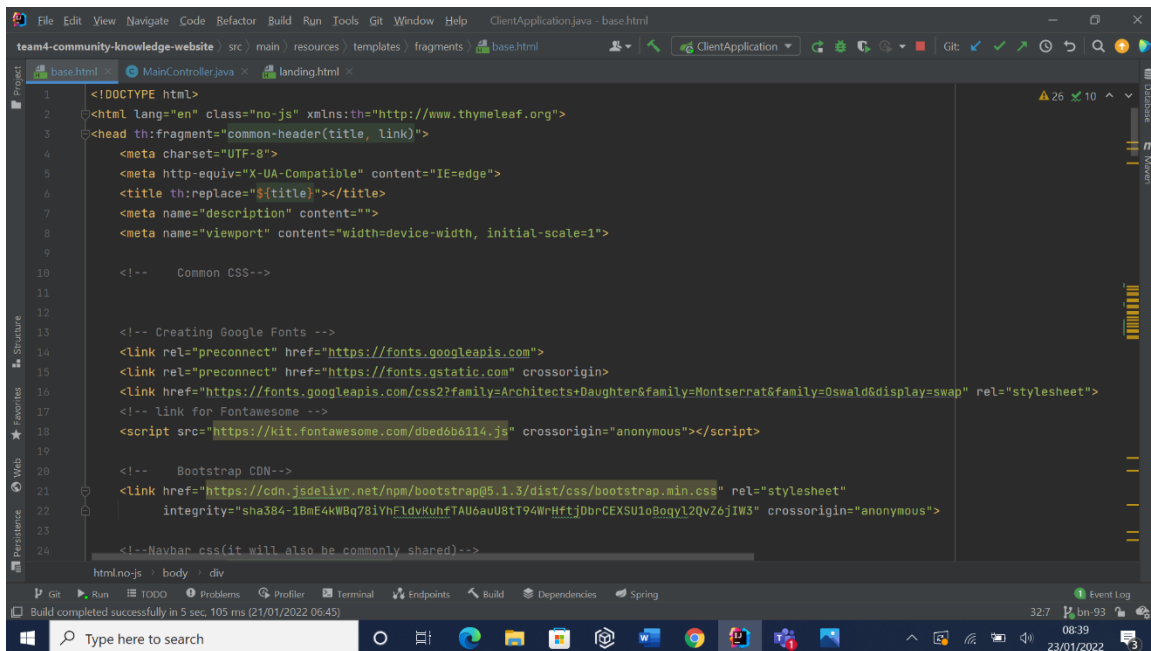
Prior to developing the aesthetics to cater the client's requirements, it is crucial that the fundamentals are being completed. This includes setting a Spring Boot Maven project boiler plate code providing basic dependencies to maximize efficiency and reusability through fragments, which includes: Website, Thymeleaf, Spring-jpa & Mysql-data (). There were certain feature I implemented to connect the users usability and user-friendly experience while navigation through the pages which includes the following:

- I. Main navigation
- II. Landing page
- III. Project donation feature
- IV. About page (Receiving update regarding platform)
- V. Contact page (contacting platform administration team regarding queries)
- VI. Administration page (updating information about the platform, viewing and attending to the user's queries)

There have been various areas within the design and development to complement this project. I was able to utilize the CSS to improve the responsiveness of each web page through media queries. The HTML files have the extension link to include specific text style by using, **bootstrap google font** and **font awesome**, maintaining a constant pattern throughout the website. In addition, I was able to embed **document object models** and create **asynchronous injection** through the backend, async/await fetch functionality was supplied using JavaScript.

Earlier into the project, I considered the landing page of the website is an important feature as it reveals vital, yet relevant information about the platform and it's the first page users encounter(Svaminathan, T. V. 2017).

The landing page consists of 2 elements. First, is a **base template** (fragment – index.html) that contains the functionalities of the header and footer; reduces the reimplementations of code to increase productivity, as it assist in displaying the HTML, CSS and JavaScript of its files with other pages on the platform.



```
1 <!DOCTYPE html>
2 <html lang="en" class="no-js" xmlns:th="http://www.thymeleaf.org">
3 <head th:fragment="common-header(title, link)">
4 <meta charset="UTF-8">
5 <meta http-equiv="X-UA-Compatible" content="IE=edge">
6 <title th:replace="${title}"></title>
7 <meta name="description" content="">
8 <meta name="viewport" content="width=device-width, initial-scale=1">
9
10 <!-- Common CSS -->
11
12
13 <!-- Creating Google Fonts -->
14 <link rel="preconnect" href="https://fonts.googleapis.com">
15 <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
16 <link href="https://fonts.googleapis.com/css2?family=Architects&family=Montserrat&family=Oswald&display=swap" rel="stylesheet">
17 <!-- Link for Fontawesome -->
18 <script src="https://kit.fontawesome.com/dbed6b6114.js" crossorigin="anonymous"></script>
19
20 <!-- Bootstrap CDN -->
21 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
22 integrity="sha384-18mE4kWBq78iYhFtLdVkuhF7AUG6auU8tT94W4rHfjtDbrCEXSU1oBogyl2QvZ6jIW3" crossorigin="anonymous">
23
24 <!--Navbar css(it will also be commonly shared)-->
25
26 </head>
27 <body>
28 <div>
```

Figure 1: Snippet of the 'Base' html file using Thymeleaf and demonstrating how the CSS, JavaScript, Bootstrap, Google Font and Font Awesome are being linked to enhance its beauty

The second element is the body of the landing page. The content within the body is designed to display relative information about the 3 primary features that this platform provides: This consists of communities, events and projects. Each feature is laid out in a grid containing an information text box that gives access to all of the communities, projects and events provided on the platform. The remaining grid cards display the latest post within their category and are updated through the changes made by their designated

ambassadors.

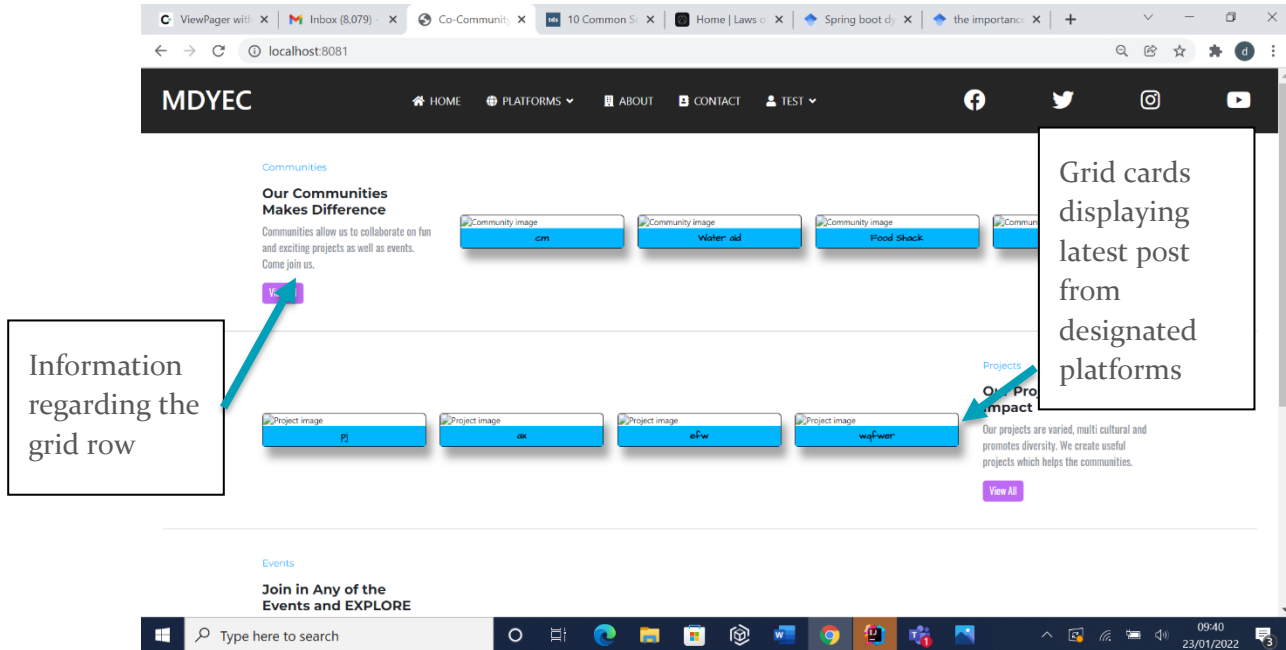


Figure 2: Home page displaying columns of the designated platforms and their latest posts

However, pre-development of the 'landing page', I was able to render the HTML file through the Thymeleaf model and specify its controller through the MainController.java and created its RequestMapping URL. Within its JavaScript, it checks if the cookie is empty to determine if a user is logged in. depending if the user is logged in the main navigation bar will adjust to suit the users.

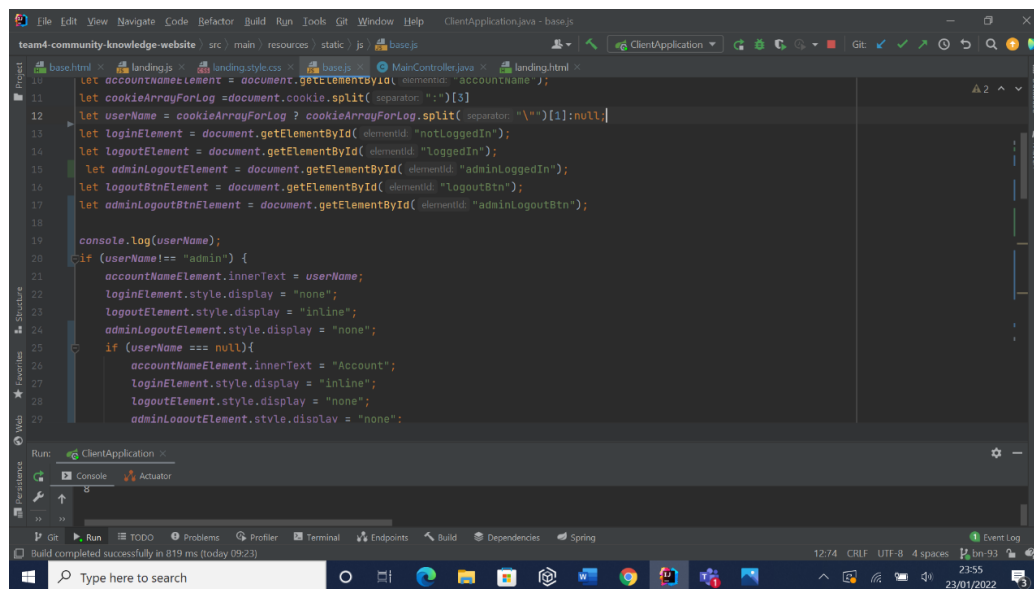


Figure 3: Checking if the user is logged in and display certain links on the navigation bar base on the type of user

The most common feature I developed that is utilized is the **project donation transaction**. Post rendering the HTML file using Thymeleaf outline and defining its controller in the MainController.java and creating its RequestMapping URL. With its JavaScript file I was able to secure the users input through the event handler which includes **change** and **click**. While saving transactions, I was able to create a Fetch API through the backend using an asynchronous Post call.

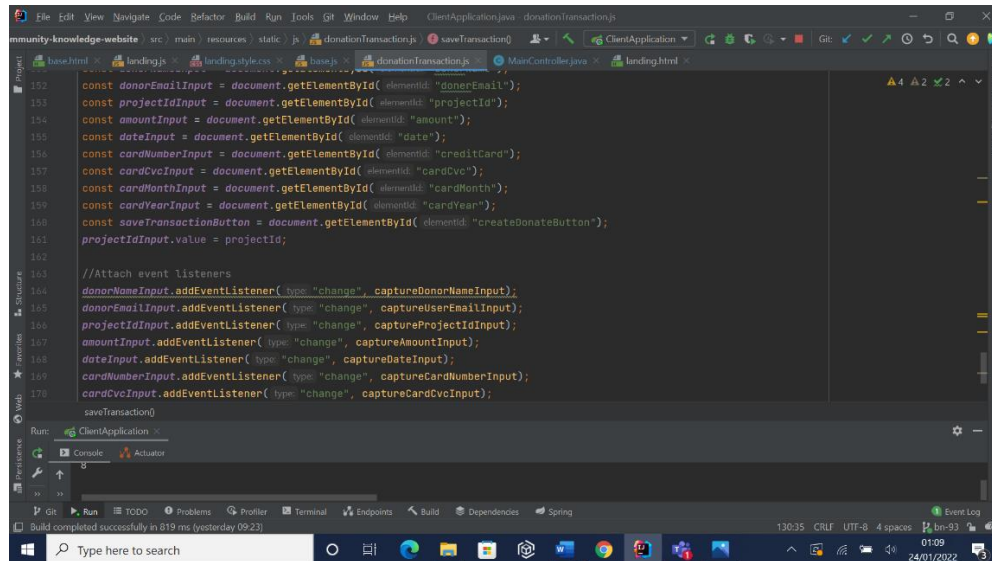


Figure 4: Gathering the input and selection elements from the donation transaction DOM and attaching event listeners within the JS file

In addition, the JS contains **validations** assessing whether the user is inserting the correct information to meet donation transaction criteria.

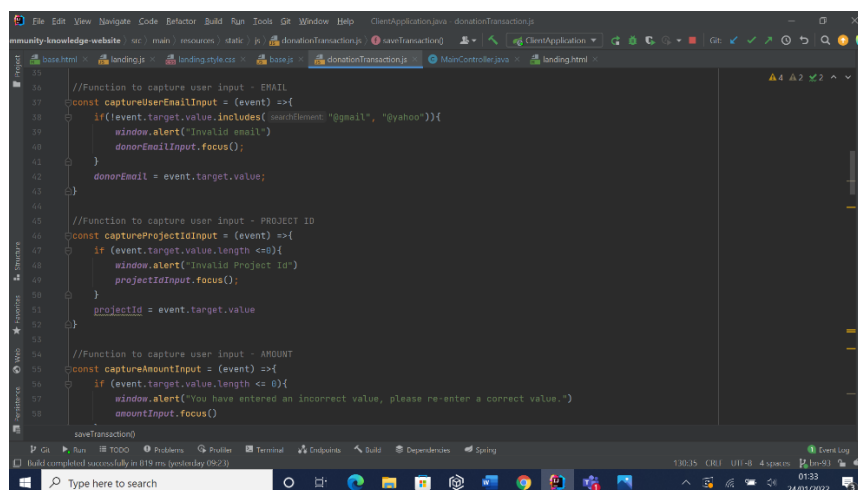


Figure 5: Project donation transaction validation to meet criteria

Applying the JavaScript's fetch API, I am able to create an async POST that demands the backend (Java files); sending a JSON payload utilizing JSON.stringify: This was achieved in the JS file.

```

120   userId: userId
121 }
122 const response = await fetch( input: "http://localhost:8081/api/new-transaction", {
123   method: "POST",
124   headers: {
125     "Content-type": "Application/json"
126   },
127   body: JSON.stringify(transactionObject)
128 })
129 console.log("response---->", response)
130 if(response.status == 200){
131   const data = await response.json();
132   // let addAmount = data.amount;
133   const addFundsUrl = 'http://localhost:8081/api/update-funds/projectId/${projectId}/amount/${amount}'
134   const responseAddFunds = await fetch(addFundsUrl, {
135     method: "PUT",
136     headers: {
137       "Content-type": "Application/json"
138     }
139   })
140   if (responseAddFunds.status == "200") {
141     window.alert("Transaction completed. Redirecting...")
142     window.location.reload();
143   } else {
144

```

Figure 6: Fetching an API for the donation transaction

In addition, a user cannot make a project donation if they are not logged in as they will be denied access. This prevents security issues as a community should be informed about their donors.

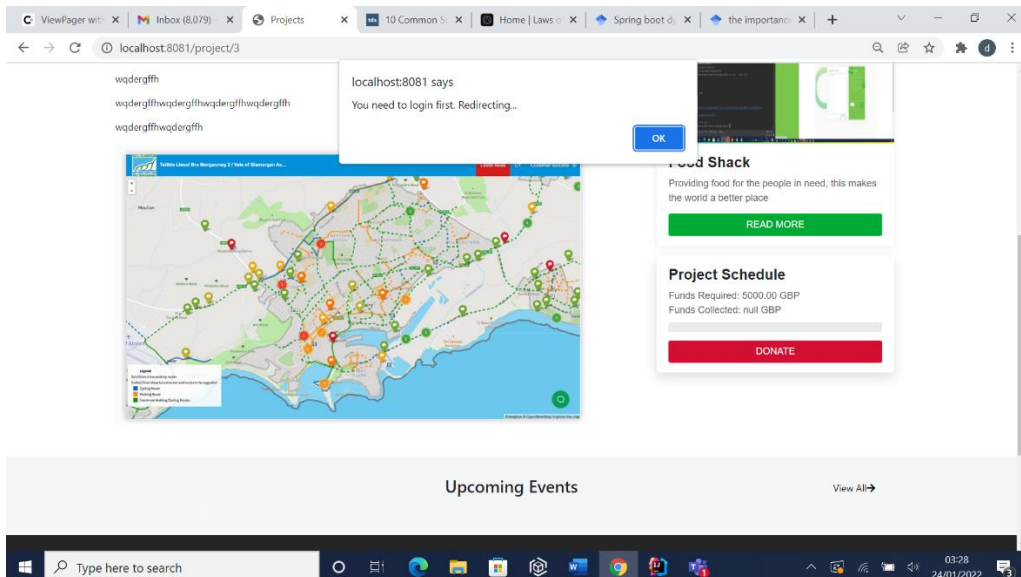


Figure 7: User cannot make a donation as they are not logged in

However once they are logged in, with a click of the donation button a pop-up of the transaction form will appear.

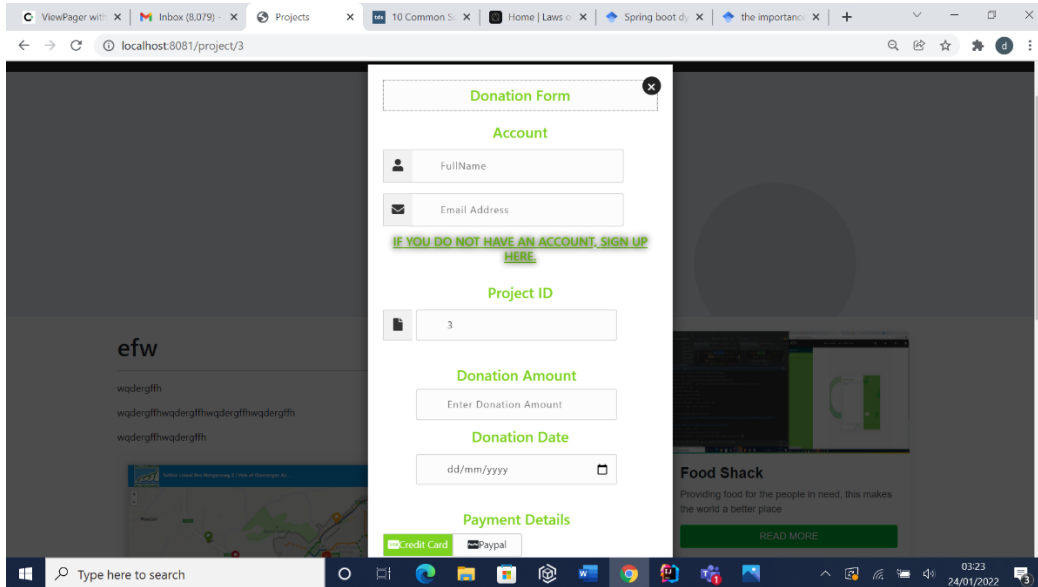


Figure 8: Pop-up donation form appears as a logged in user

This pop-up effect is caused by hiding a particular element (form) within an id tag and displaying once a donate button is clicked.

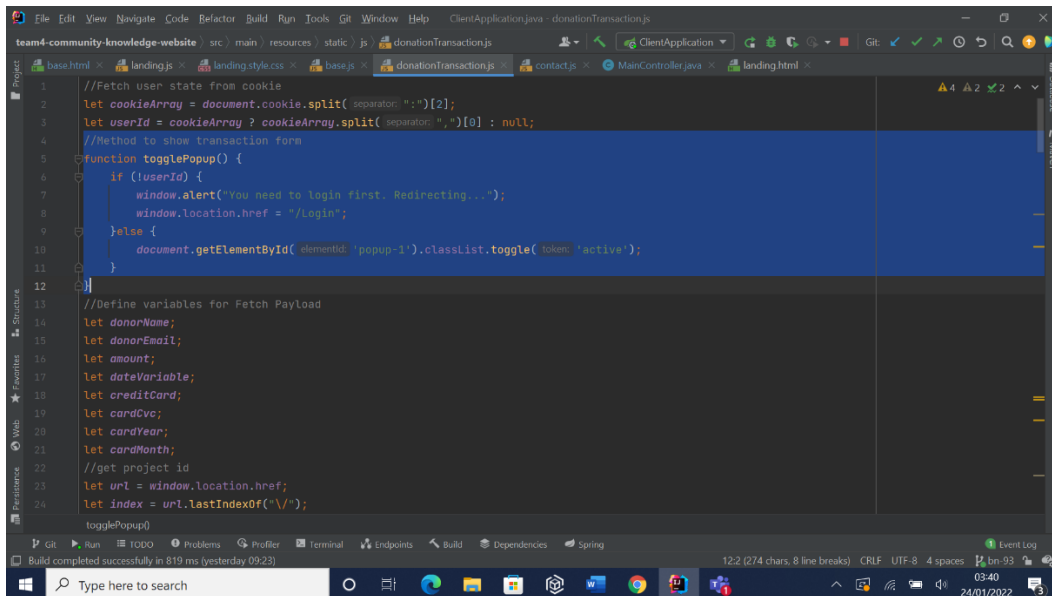


Figure 9: Creating a toggle pop-up to activate when a user is login and activation button is clicked

The Contact page followed the same procedure as the donation transaction page to store users information, however, there is a slight difference as the contact form is able to store documentations. This is caused by capturing the file name using the fetch POST call to



the API/upload-image, as well as updating the <p> tag that displays the content name of the selected documentation.

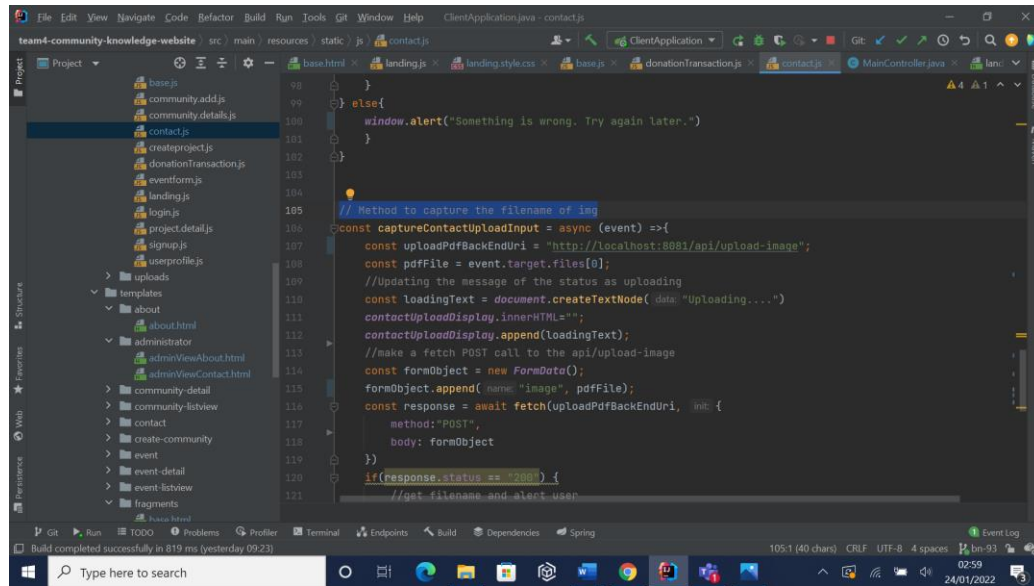


Figure 10: A method to capture the uploaded documentation in the Contact JS

This is how the contact form is presented to the users.

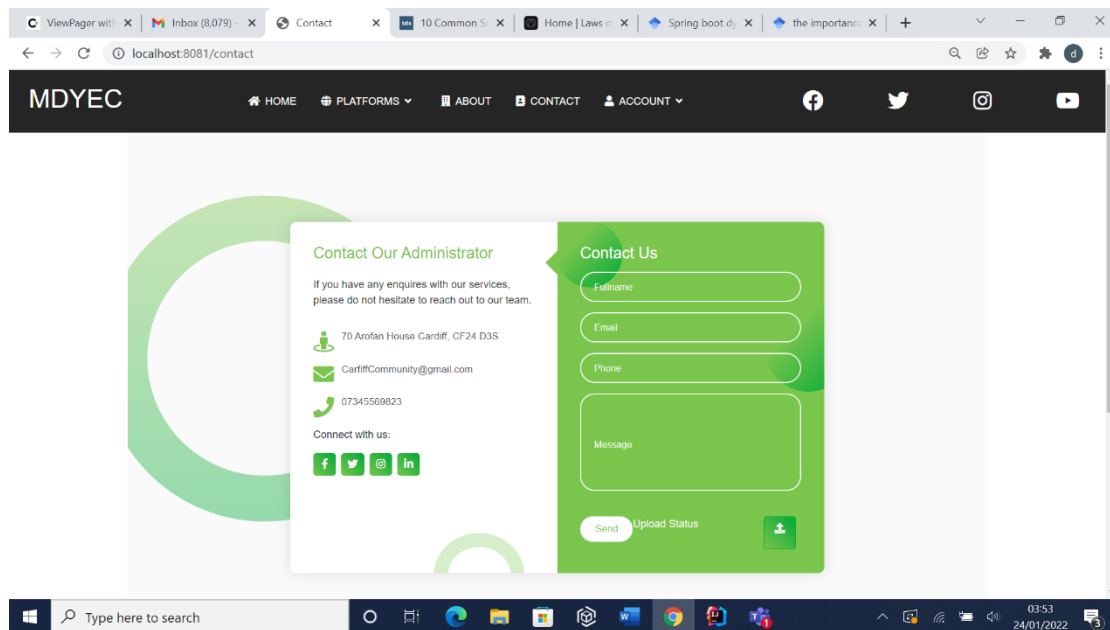
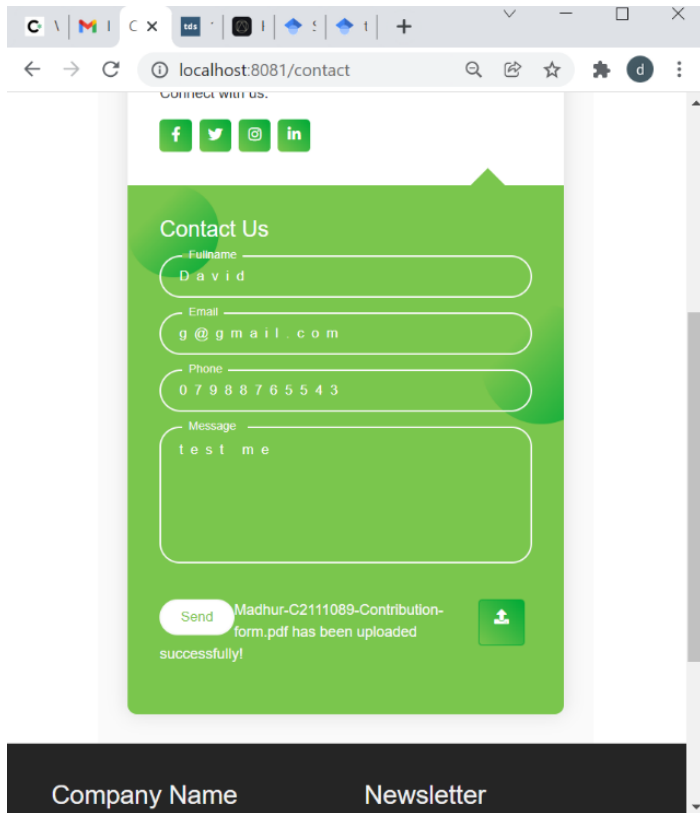


Figure 11: How users view the contact form

When considering the user's interaction while navigating the 'Contact Page', I considered the aesthetic and effectiveness of the contact form. For instance, users are able to view various ways to contact the administration team on the left hand side of the 'contact card'. Whilst the right side allows users to contact the administration team directly on the



platform. When interacting with the form, the users are able to watch the placeholder within the text boxes animate to the top border of their text boxes. I was able to use the browser to amend bugs. These embedded bug tools such as console, element inspector and network were utilized. It was an efficient way to adjust certain elements such as padding, margin and viewing how each responsive page will be shown on various devices.



*Figure 12: A responsive view of the contact form while demonstrating the aesthetics*

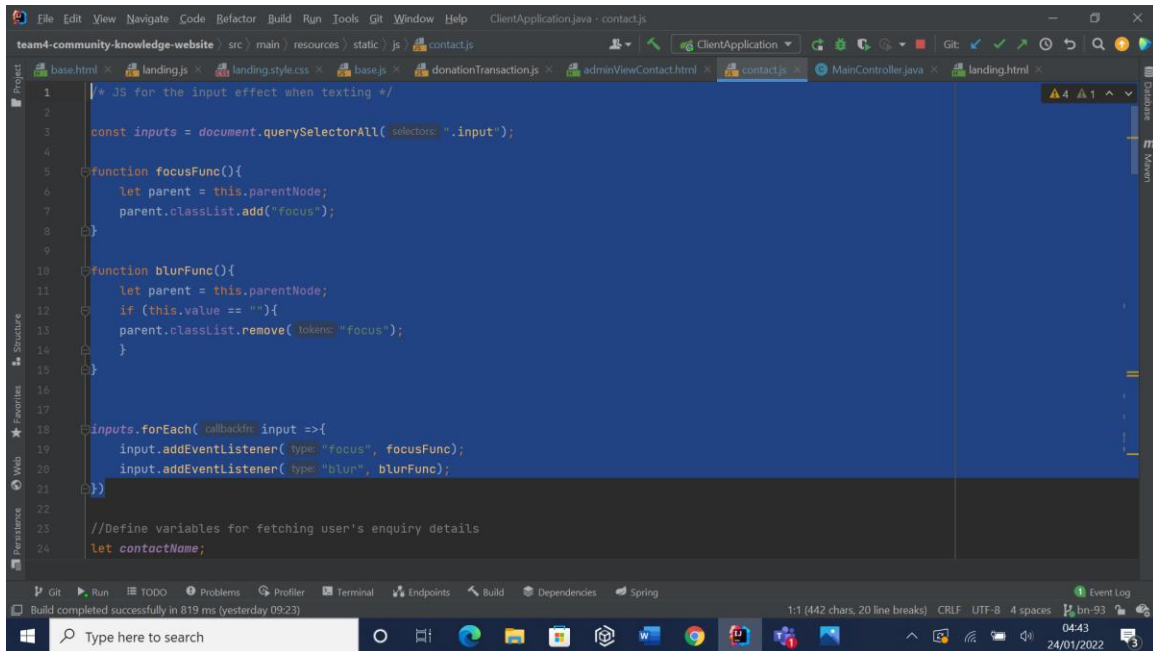


Figure 13: The JavaScript implementation behind the input boxes aesthetics and animation

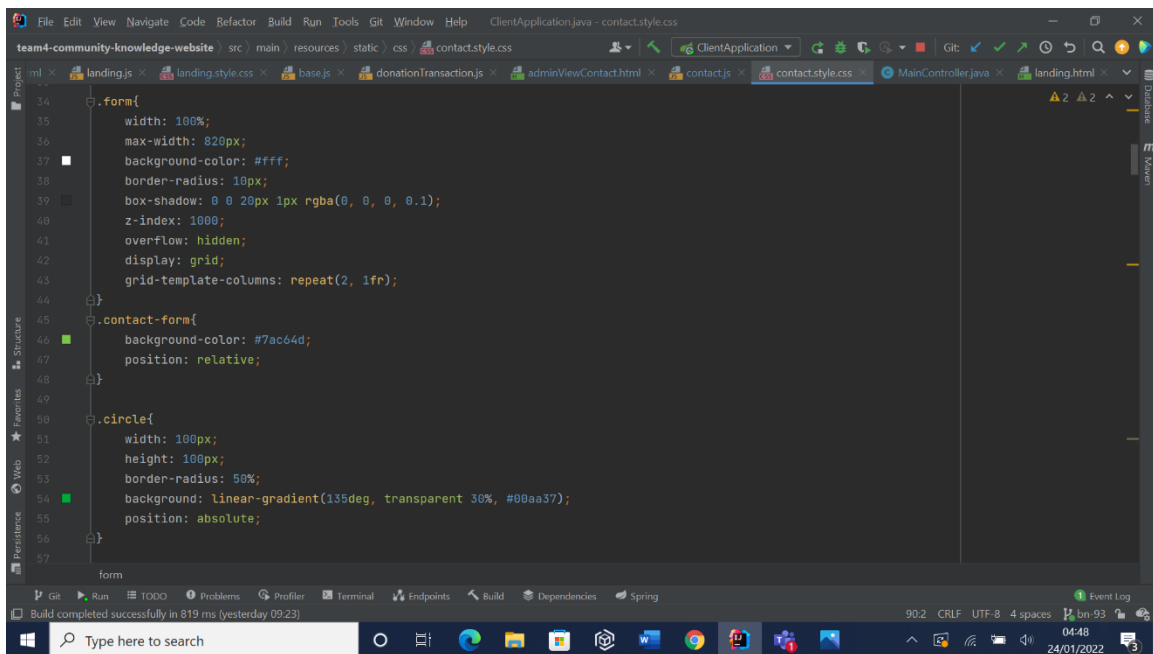


Figure 14: CSS of the Contact page, creating each color and shape

Once the Customer's queries have been sent from the contact page, the administrators receive their queries and contact details. The administrator has the ability to view user's queries in a table using Thymeleaf attributes 'th:each' and 'th:text' and attend to their needs to increase users satisfaction. In addition, they are able to update the about page, display relative information to notify the viewers.

```

<table>
  <thead>
    <th>Fullname</th>
    <th>Email</th>
    <th>Phone No.</th>
    <th>Query</th>
    <th>Documentation</th>
  </thead>
  <tbody>
    <tr>
      <td>{{each="contact: ${contacts}">
        <td th:text="${contact.name}">No Data Stored</td>
        <td th:text="${contact.email}">No Data Stored</td>
        <td th:text="${contact.phone}">No Data Stored</td>
        <td th:text="${contact.message}">No Data Stored</td>
        <td>
          <!-- updated upload button -->
          <label id="download-container" for="downloadInput"><i id="download-icon" name="upload-icon" class="fa fa-down
          <a th:text="${contact.uploadInput}" id="downloadInput" download ></a>
          <p id="downloadStatus"></p>
        </td>
      </tr>
    </tbody>
  </table>

```

Figure 15: Displaying how a table receiving users queries is structured when made with Thymeleaf attributes.

Fullname	Email	Phone No.	Query	Documentation
David	g@gmail.com	07988765543	test me	MSc SE - Autumn - Assessment Map (1).pdf
Sam Jones	Jones@gmail.com	07233980076	ghjklmjhgfdsxgchbjnkmjhgfd	Screenshot (1).png
Allen Jones	AJ@gmail.com	07233986432	I am having issues with my Project page.	Screenshot (3).png
Alex	al@gmail.com	07233987743	The Platform isn't updating	MSc SE - Autumn - Assessment Map (1).pdf
steeve	ste@gmail.com	07988232284	I need some help	MSc SE - Autumn - Assessment Map (1).pdf
Sam	S@gmail.com	07422349987	dsrjsrjsrjnskdskds	Madhur-C2111089-Contribution-form.pdf

Figure 16: The Administration team sees the content of the table based on customer queries

## UTILISING THE SERVER SIDE CODE TO IMPACT THE DYNAMIC WEB

When developing the application there are procedures that were taken, such as using a Model, View, Controller (MVC) architecture which focuses on the interlinking three vital elements that initiates and utilizes OOP software paradigms. There were simple but effective procedures from including the procedural programming paradigm during the implementation of the MVC structure. There are various features I created that use the JSON payload received from the frontend such as donation transaction, contact, administration platform. However, I will elaborate on the adopted process whilst creating the **donation transaction** feature, process listed below:

- I. I created a **controller** java class that focuses on reacting to the users input and interacts with the transaction model objects. This process accomplishes operations that alter the outcome of the model through receiving and validating user inputs within the controller.

```
//API endpoints will be defined here
//1. End point is to save the transaction
@PostMapping("/api/new-transaction")
public DonationTransaction saveNewTransaction(@RequestBody DonationTransaction donationTransaction){
    System.out.println("I am being hit");
    return donationTransactionService.saveTransaction(donationTransaction);
}

//end point to get one project by id
@GetMapping("/api/get-project-transaction/{id}")
private Set<DonationTransaction> getProjectById(@PathVariable String id){
    return donationTransactionService.getProjectById(id);
}
```

Figure 17: A snippet of the donation transaction controller that receives POST requests, RequestBody and pathvariable then return specific HTTP response. In addition, I set up a GET mapping to get a project by ID

- II. I created a **service** java class that has **repository injection** and interacts with other repository classes(querying objects); merging data to create new, enhanced

objects. This enabled me to use abstractions and encapsulation during areas of the web application and the repositories altering autonomously.

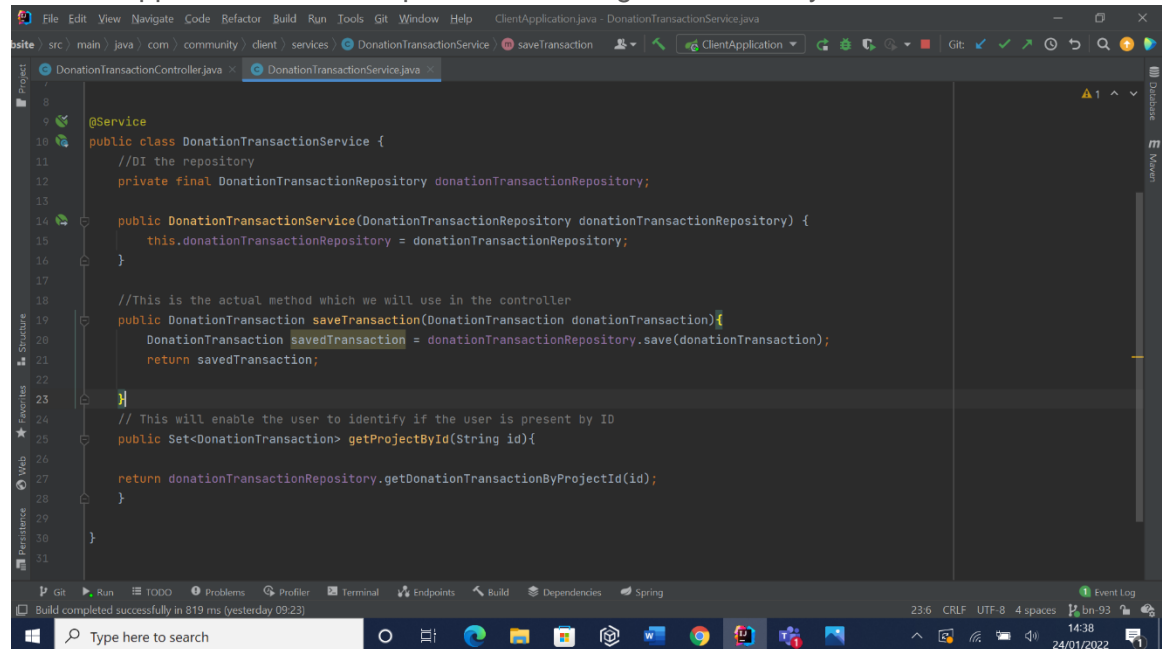


Figure 18: Creating a Dependency Injection for the donation transaction repository. Spring Boot allows DI to initialize the instances and utilizes it when needed.

- III. I created a **Repository** java class(interface) that focuses on the data transaction (sending and receiving) storage. It comes with its own limitations as each repository works alongside an individual **Model** class. Therefore, when I designed the donation transaction, contact, administration platform Model classes, a repository class had to be created for each Models: It is unable to interact with other Model classes.

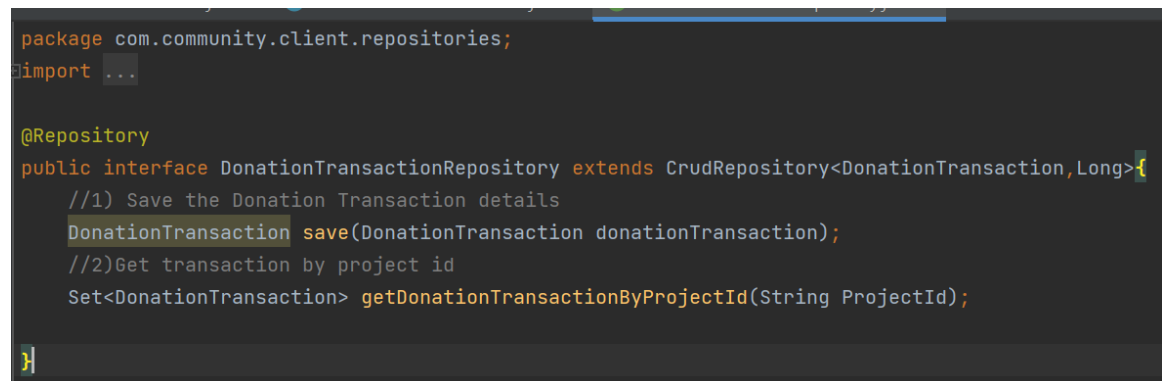


Figure 19: The donation transaction repository extends CrudRepository and save its transaction details by overriding its method implementation. It is also annotated in Spring Boot and creates Spring Bean and the DI is accessible.

- IV. I created another Model which is considered as an Entity object and converted to a database table by creating object relational mapping.

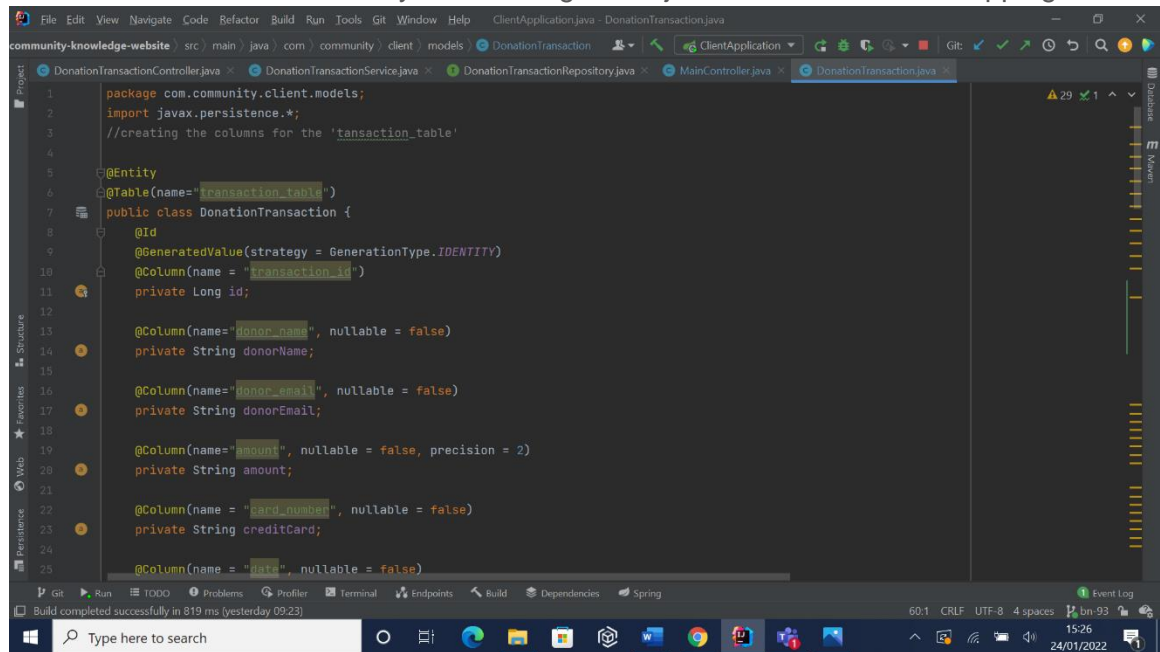


Figure 20: Creating the table using entity and columns within the table as well as set getters and setters

- V. The last step was to merge and connect all the classes together through correct annotations and dependency injection.



Figure 21: Creating a RequestMapping to link with the browser through an URL action and return model and view

## IN THE CONTEXT OF A SOFTWARE ENGINEER, DESIGN AND CREATE A DATABASE THAT INTEGRATES WITH THE WEB PAGES

When creating the database I started with an Entity Relationship Diagram. I was able to identify that the donation transaction table did not require any one-to-one relationship with the user model, however one user can make many donations. Due to the ERD, I was able



to implement a JPA within the java class and Hibernate can convert the Object-Relational

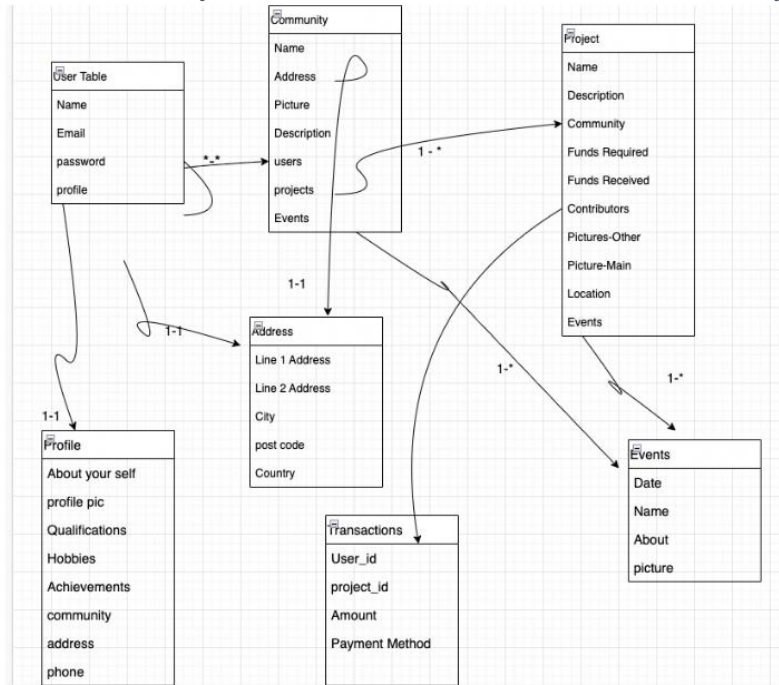


Figure 22: Creating an ER Diagram with my team

Mapping into the DB table I created. Look at **figure20**.

**Figure20** demonstrates how I created and simplified java objects through the implementation of JPA. I chose to create column names to address the various user inputs which stem from the ERD, however, there were additional columns included to improve users information gathered. Afterwards, I created getters and setters based on the java model since they will be needed in the donation transaction service and controller. Annotations were made to the java model with Entity annotations that enables Spring JPA to offer the current java object into the db table. HTTP requests from the controller and converts to dependency injection in the transaction service (influencing the class and methods) would be made through the information in the table being queried. I was able to identify which column is mandatory for users to fill in.

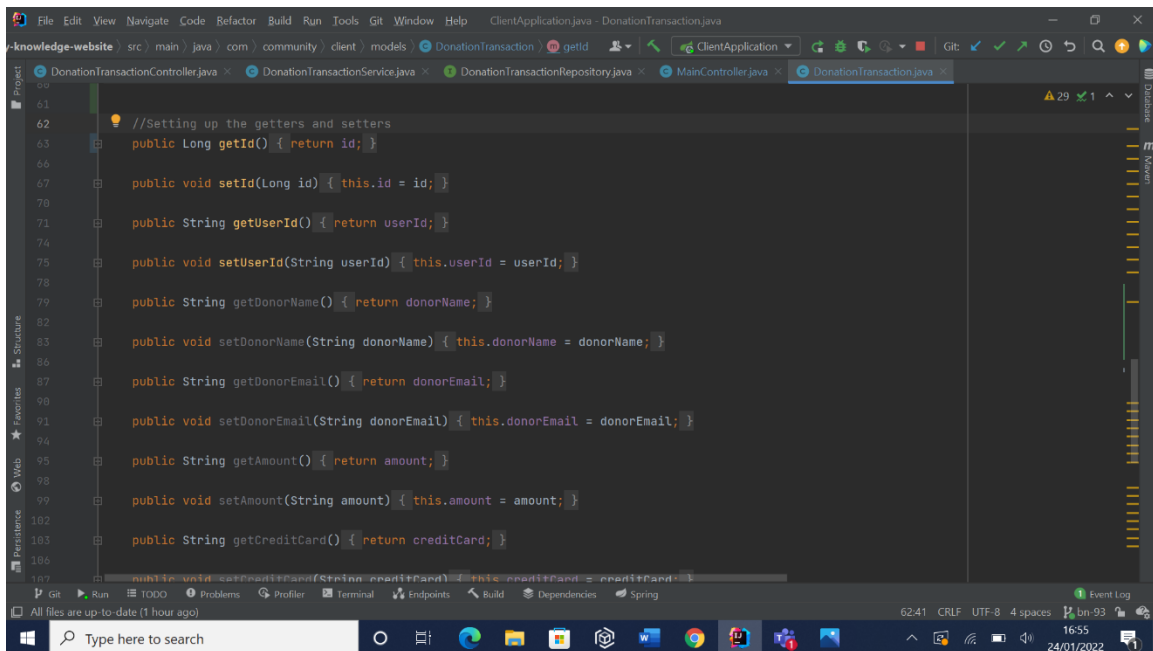


Figure 23: Demonstrating the getters and setters

Furthermore, to create these entity tables there had to be a database connection in the application.properties file. This included specifying a database URL, username and password. This is a default configuration for Spring Boot.

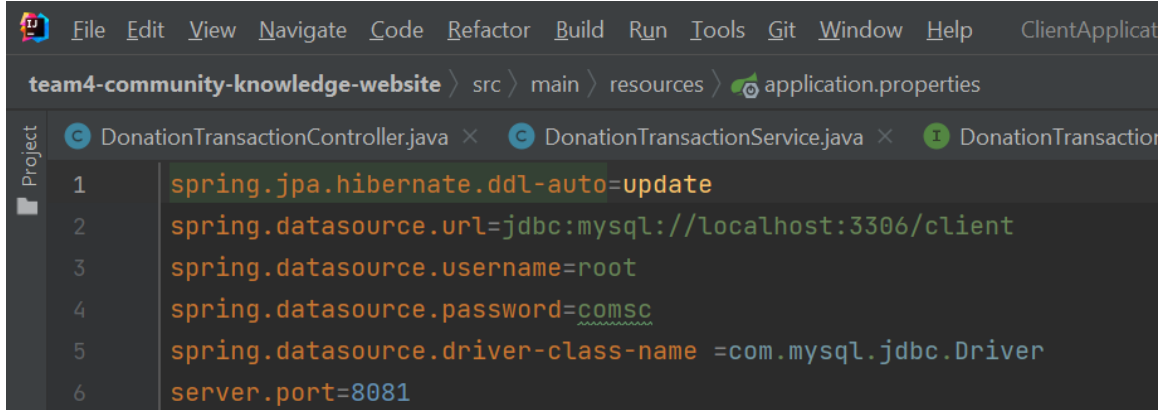
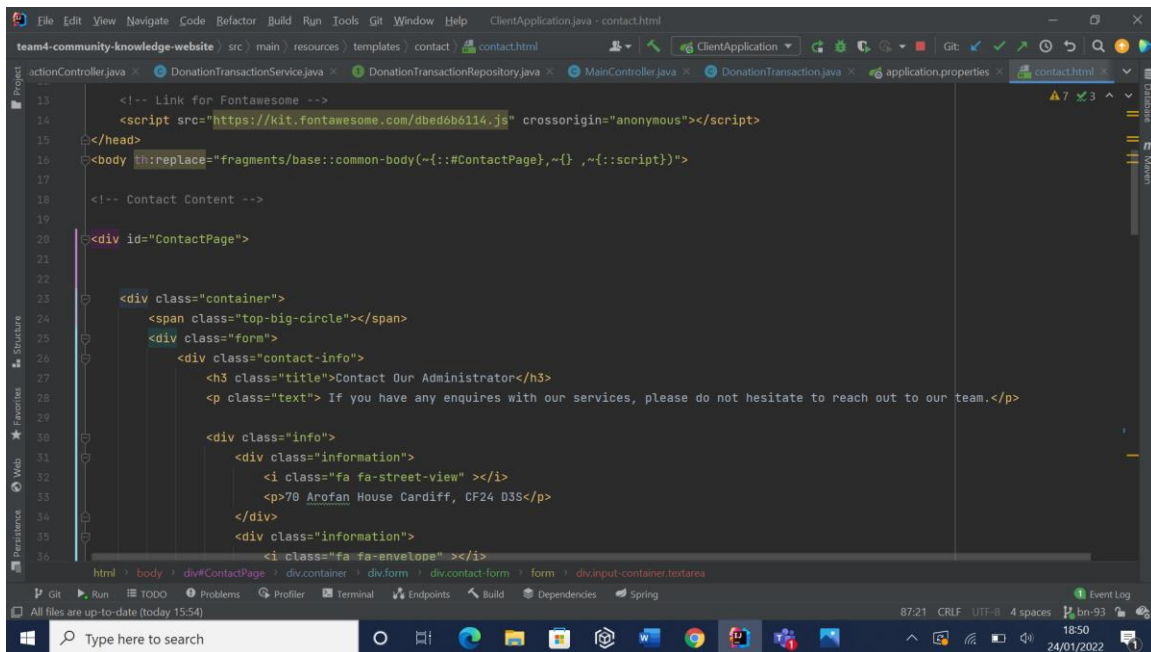


Figure 24: Configuration in application.properties

## DEMONSTRATE A SYSTEMATIC UNDERSTANDING OF COMPUTER CONCEPT RELATING TO WEB APPLICATIONS, BOTH THEORETICAL AND PRACTICAL

*During the development of a dynamic application I was able to identify 3 main components, such as the client-side (what the user interacts with), Server-side (code behind the interaction) and database: These components were utilized during my contribution in the client project. I was able to exercise my skills whilst implementing HTML, CSS, JavaScript code. The application can run in any browser provider and this will not cause any restriction on the users. I was able to simplify my tasks by using **spring boot's** Thymeleaf, it enabled me to send and retrieve data from backend and html code instantly. Bootstrap was not necessary when developing the aesthetics but was used in certain areas. To assist in making the platform responsive I was able to use media queries. JavaScript was a crucial elements since it enabled me to improve users interaction and collaborating with the backend.*



```
13 <!-- Link for Fontawesome -->
14 <script src="https://kit.fontawesome.com/dbed6b6114.js" crossorigin="anonymous"></script>
15 </head>
16 <body th:replace="fragments/base::common-body(~{::#ContactPage},~{::script})">
17
18 <!-- Contact Content -->
19
20 <div id="ContactPage">
21
22
23 <div class="container">
24 <span class="top-big-circle"></span>
25 <div class="form">
26 <div class="contact-info">
27 <h3 class="title">Contact Our Administrator</h3>
28 <p class="text">If you have any enquires with our services, please do not hesitate to reach out to our team.</p>
29
30 <div class="info">
31 <div class="information">
32 <i class="fa fa-street-view"></i>
33 <p>70 Arofan House Cardiff, CF24 03S</p>
34 </div>
35 <div class="information">
36 <i class="fa fa-envelope"></i>
```

Figure 25: HTML file for contact page

Interacting with the client-side (backend) I was able to build the repositories, controllers, model, services classes(MVC model). The repositories are interfaces that utilize any in-built java classes, for instance JPA Repository or Crud Repository. I was able to provide methods for the controllers through using the logics written in the service classes. MySQL

was as my database server and it was designed through the parameters created by the spring boot's configuration file.

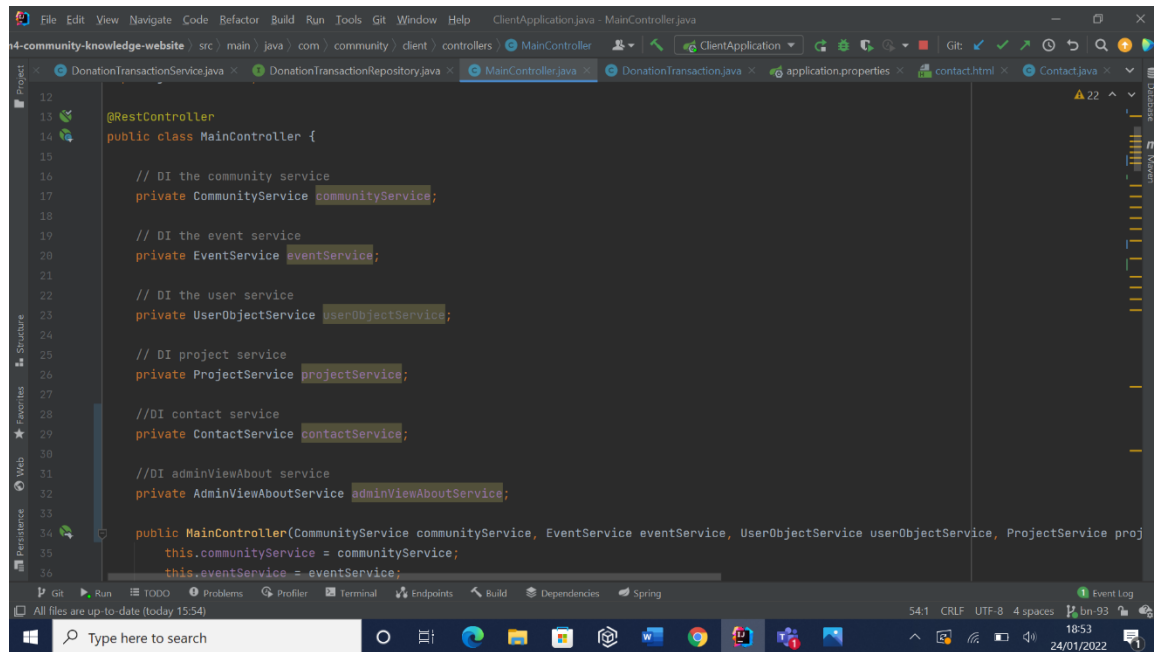


Figure 26: backend, main controller

To create a tidy work environment my team and I decided to put the client-side code in a folder called **resources**, while the java files were located within the main class. This increased efficiency in the work environment, thus locating files easier and readable. The local server was created to display all the static content from the html pages(includes CSS and JavaScript), while gathering/interaction with real-time data which is fetched from MySQL db.

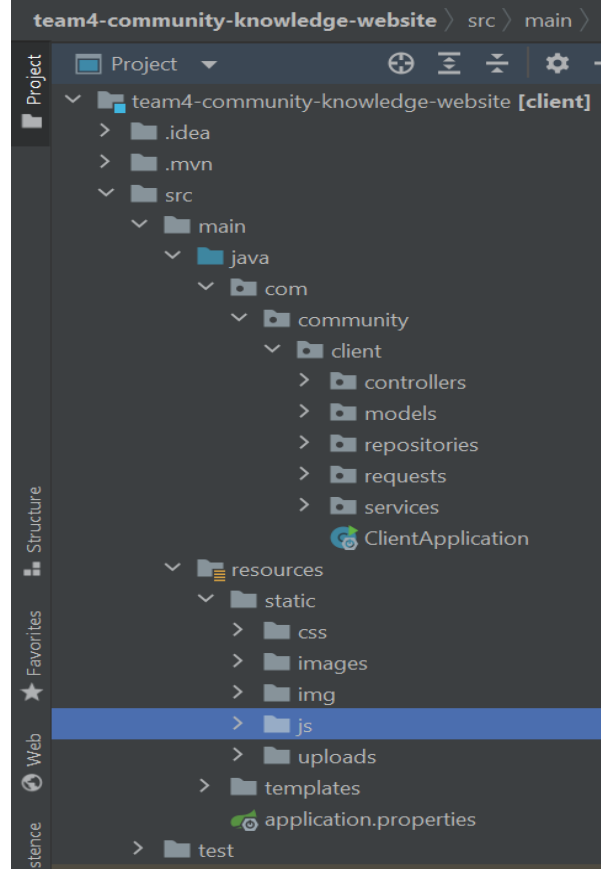


Figure 27: Folder Structuring

## CRITICALLY ANALYZE ARCHITECTURAL DESIGN PATTERNS RELEVANT TO WEB APPLICATION AND REFLECT ON CHOICE MADE

With the necessary resources provided I was able to work on the client-side, backend architecture and the MVC.

In conclusion, I implemented the client project which consists of working on the client-side, server-side and database, this helps the platform run autonomously and switch data through the HTTP requests (browser): This is considered as an API. Implementing the backend, which followed a procedure which enabled me to fetch, store, update the database. This assisted in keeping my code/process cleaner and implementable. I was able to keep the logic in separate sections instead of the static method, for instance CRUD.

I was able to understand the importance of MVC as I was able to use the **model** to represent data or create the database; process of machines, whilst the **view** is a state of the model that visualizes(visualization). Finally, the controller presented facilities to amend the state of the model. These three layers assisted throughout the entire client-side development.

There are future amendments I feel are necessary to improve the functionality of the application such as the administration platform.

For further improvement I would consider creating a delete attribute on the customer's query table located in the administration forum. This will help to discard queries that have been attended too instead of the table being clustered. Also, implementing a search above the customer's query table to filter the table presented: Makes the administration task easier while attending to queries. Instead of using MySQL I would consider using mongo db to hold the database and React for the frontend: This makes the work experience more user friendly.

## Referencing

Svaminathan, T. V. (2017). Quantifying the Relative Importance of Key Drivers of Landing Page. *Indian Journal of Marketing*.