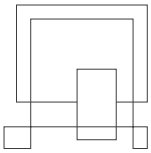


Food and Recipe Assistant

Project report

Students:

Patrik Kucerka (260026)

Patrik Ihnat (260010)

Supervisor: Michael Viuff

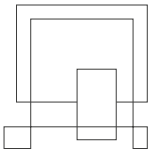
VIA University College

Number of characters without spaces: 43 557

ICT Engineering

7th semester

Date: 1.6.2020



Abstract

As the research in the field of healthy lifestyle makes progress, people care about what they eat more and more. This results in the need for a system that would be a solution for handling related task. The aim of this project is to build a recipe management system which would be a cross platform solution that would eliminate the need for many different applications to satisfy the arising needs.

The solution is a system that consists of a single page web application and a RESTful server implemented in Spring Boot that handles the database and file system interactions. The recipe image compression is also handled by the server. The client side is implemented in AngularJS and its main functionalities are creating, deleting and updating recipes. Adding ingredients to a shopping list and searching through the recipes by applying filters as well as possibility to share the recipes with friends. Due to the responsive design of the system the application is available on multiple screen resolutions.

All requirements have been accomplished on time and tested to assure the proper behaviour of the system.

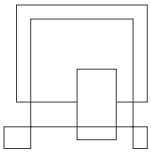
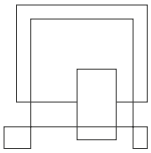
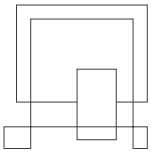
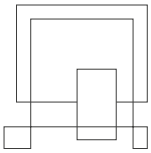


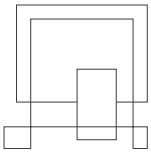
Table of content





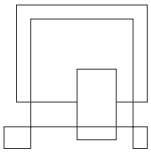
List of figures and tables





1 Introduction

This document contains a detailed information about the steps taken during the process of development of the Food and Recipe assistant which is an advanced management system for recipes. The first four chapters are dedicated to the four stages of the development which are analysis, design, implementation and testing providing an insight to the decisions made during them. Analysis points out the actions behind the extraction and formulation of the software requirements specifications. Next chapter dives into the planning of the system design including the design of algorithms as well as architectural design. This chapter is followed by implementation describing how the designed functionalities were implemented from the perspective of writing the source code providing more information about what techniques and patterns were used and the reasoning behind them. This chapter also includes a section about the technologies used during the implementation phase and the considerations that led to these choices. Last of these four stages is testing. The chapter about the testing detailly describes proceeding towards the high quality and reliability of the product with respect to the testing models and methods selected to ensure these attributes. Chapter results and discussion states the outcome and goals achieved during these stages. Finally, this report states the conclusion and discusses the future of the project.



2 Analysis

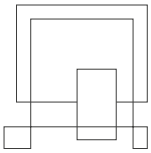
The issues discussed in the Project description – Appendix A were identified and broke down into the smaller tasks called requirements to gradually proceed towards the solution. These were retrieved from the customer to fulfil their expectations and needs throughout the process called requirements engineering since the main aim is to deliver a high quality software that satisfies the customer's needs.

2.1 Requirements

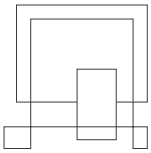
This section contains the lists of requirements including both functional and non-functional. The functional requirements were prioritized after considering which ones are the core requirements and which ones are the most important. This was also done to ensure the correct flow of the development process as some requirements might be dependent on the others e.g. adding a user as a friend is a precondition of sharing a recipe with another user thus it has a higher priority and shall be implemented first.

2.2 Functional Requirements

- FR-1.1: The user shall be able to create a new user account by filling in the required information and submitting it.
- FR-1.2: The user shall be able to log in to the application.
- FR-2.1: The user should be able to add a new recipe by filling in the required information and submitting.
- FR-2.2: New recipe should appear in the list of recipes immediately without the need to refresh the page.
- FR-2.3: When the user selects the recipe from the list, more details should be displayed.
- FR-3.1: The user shall be able to update an existing recipe by providing new values.
- FR-4.1: The user shall be able to remove an existing recipe from his list of recipes.



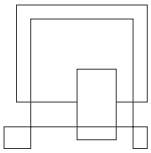
- FR-4.2: If the user shared this recipe with some of the other users - friend, it shall not be deleted from their lists.
- FR-5.1: The user shall be able to add other users as „friends“ by email address.
- FR-5.2: The user should be able to confirm a friend request.
- FR-5.3: The user should be able to reject a friend request.
- FR-5.4: If the fried request is confirmed, users should be able to mutually see each other in their „Friend lists “.
- FR-6.1: The user shall be able to send a recipe to other users from his „ Friend list “.
- FR-7.1: The user shall be able to search in the list of recipes by the recipe name.
- FR-7.2: The user shall be able to search in the list of recipes by the ingredient.
- FR-7.3: The user shall be able to search in the list of recipes by the category.
- FR-8.1: The user shall be able to get a recipe suggestion.
- FR-9.1: The user shall be able to add items to the shopping list directly from the list of ingredients of a recipe.
- FR-9.1: The user shall be able to remove items from the shopping list.
- FR-10.1: The user shall be able to receive notifications about friend requests.
- FR-10.2: The user shall be able to receive a notification about recipes received from friends.
- FR-11: Manage user profile



2.3 Non-Functional Requirements

Even though these requirements do not have a functional character they are still of a high importance as they ensure certain qualities of the system. They dictate the system attributes that need to be achieved by non-functional approach.

- **NFR-1:** The application should be independent of the operating system of the device.
- **NFR-2:** Client should have a responsive character.
- **NFR-3:** The response time of the system should be less than 3 seconds 90% of the time.
- **NFR-4:** The time to learn and operate the system shall be below 15 minutes.
- **NFR-5:** Menu of the system should be designed for easy use on touch screens.
- **NFR-6:** System language shall be English as it is most widely spoken world language.



2.4 User Stories

The user stories were created in order to fulfil the requirements specified above and they reflect the interaction of the user with the system. Since there is need only for one type of the user, there are no restrictions in the system and all functionalities are accessible to all the users. All user stories are displayed in the use case diagram of the system in the *Figure 2.1*. It pictures all allowed actions that the user can perform as well as the relationship between them.

1. As a user I want to be able to create a profile.
2. As a user I want to log in.
3. As a user I want to update the profile, so it contains an actual information.
4. As a user I want to add a recipe so I can access it whenever needed.
5. As a user I want to manage recipes so I can adjust them or remove them depending on my needs.
6. As a user I want to search through the recipes, so I do not waste time on scrolling through them manually.
7. As a user I want to add another user as a friend to keep track of the users I am interest in sharing recipes with.
8. As a user I want to share a recipe with other users, so the recipes do not need to be typed in manually.
9. As a user I want to manage received recipes, so I can decide whether to keep them or not.
10. As a user I want to get recipe suggestions, so it is easier to make a decision about what to cook.
11. As a user I want to receive notifications, so I am informed about the incoming recipes and friend requests.

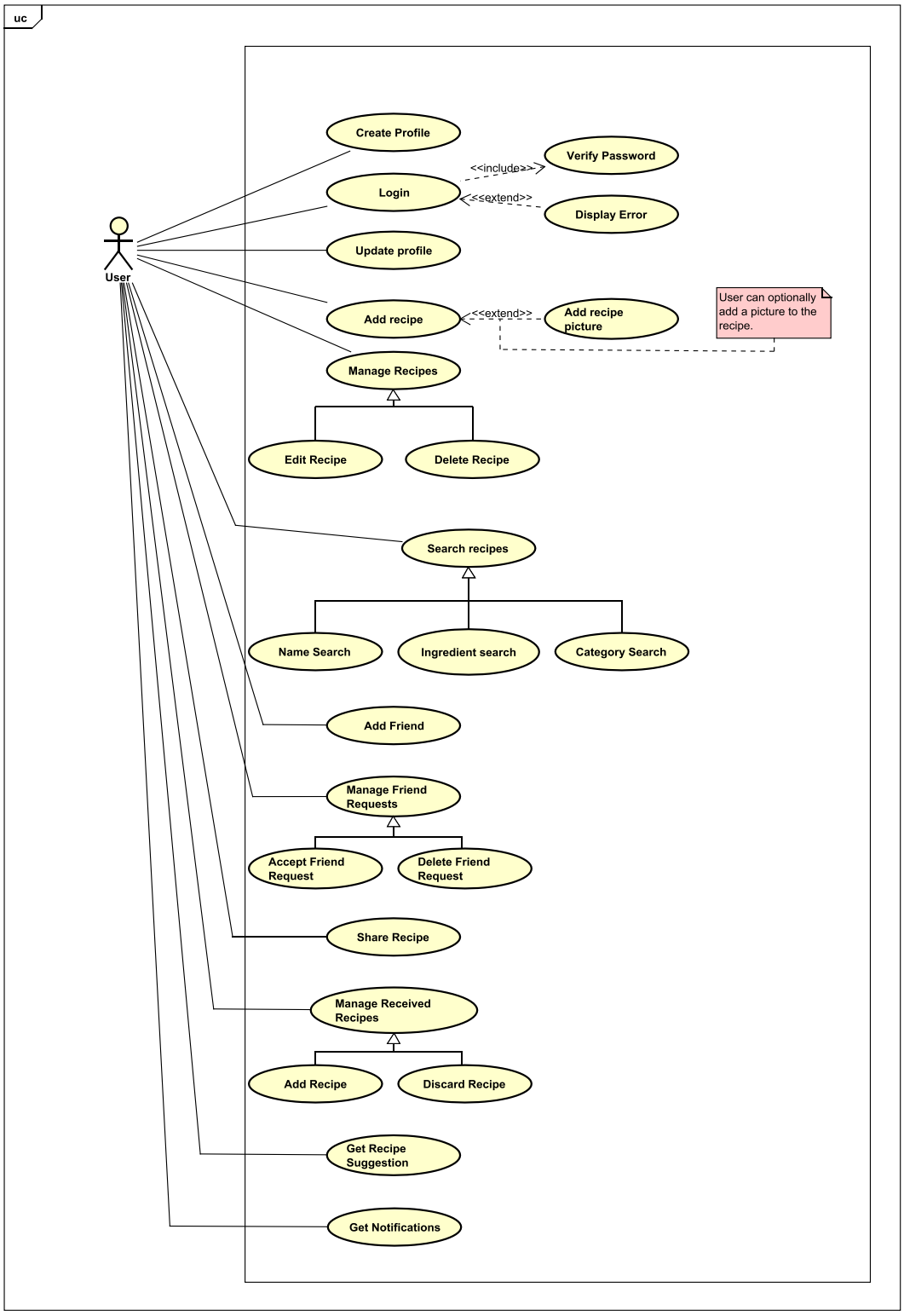
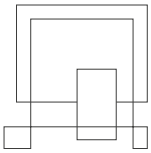
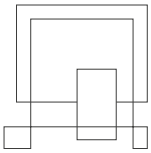


Figure 2.1 - Use Case Diagram



2.5 Use cases

The more detailed information about each of the use cases are present in the use case descriptions. These providea closer look at the steps behind each actions as well as the conditions to finish the action sucessfuly. Preconditions are the conditions that need to be fulfilled otherwise the action can not becompleted. Postconditions are the results of successfully proceedint through the steps in base sequence which lists the smallest units that the action consists of. When these steps are followed and the preconditions are met, the user will achieve the expected result. If the preconditions are not met, the exception will arise and the steps to solve the issue are listed in the exceptions sequence. Use case description for creating a user profile can be seen in the *Figure 2.2*.



ITEM	VALUE
UseCase	Create Profile
Summary	The user is able to create a profile by filling in all the necessary information in the registrat ion form and submiting it.
Actor	User
Precondition	The user email and password are confirmed.
Postcondition	1. The user profile is created. 2. The user is able to log in.
Base Sequence	1. The user loads the web application. 2. The user clicks on "Register" button on the mainpage. 3. Registration Form appears. 4. User fills in all the information. 5. The user clicks "Register" 6. The user profile is successfuly created and stored in the database. 7. The form disappears.
Branch Sequence	
Exception Sequence	Exception: email or password are not matching values in the confirmationfields 1. The field with mismatching value changes color to red. 2.a The user enters a matching value and the field changes color to green. 2.b The user clicks "Cancel" button. 3.a The user profile is created successfuly and the fomr disappears. 3.b The form disappears.
Sub UseCase	
Note	

Figure 2.2 - Use Case Description

2.6 Activity Diagram

Another tool that was used to provide a better insight to the dynamics of the system are activity diagrams. Activity diagrams were used to depicture the flow between the individual activities. It also catches the behaviour of the system specifically the actions that will follow after a certain activity was executed, depending on the users input. Following activity diagram - *Figure 2.3 - Activity Diagram,* shows activities involved in the recipe suggestion functionality.

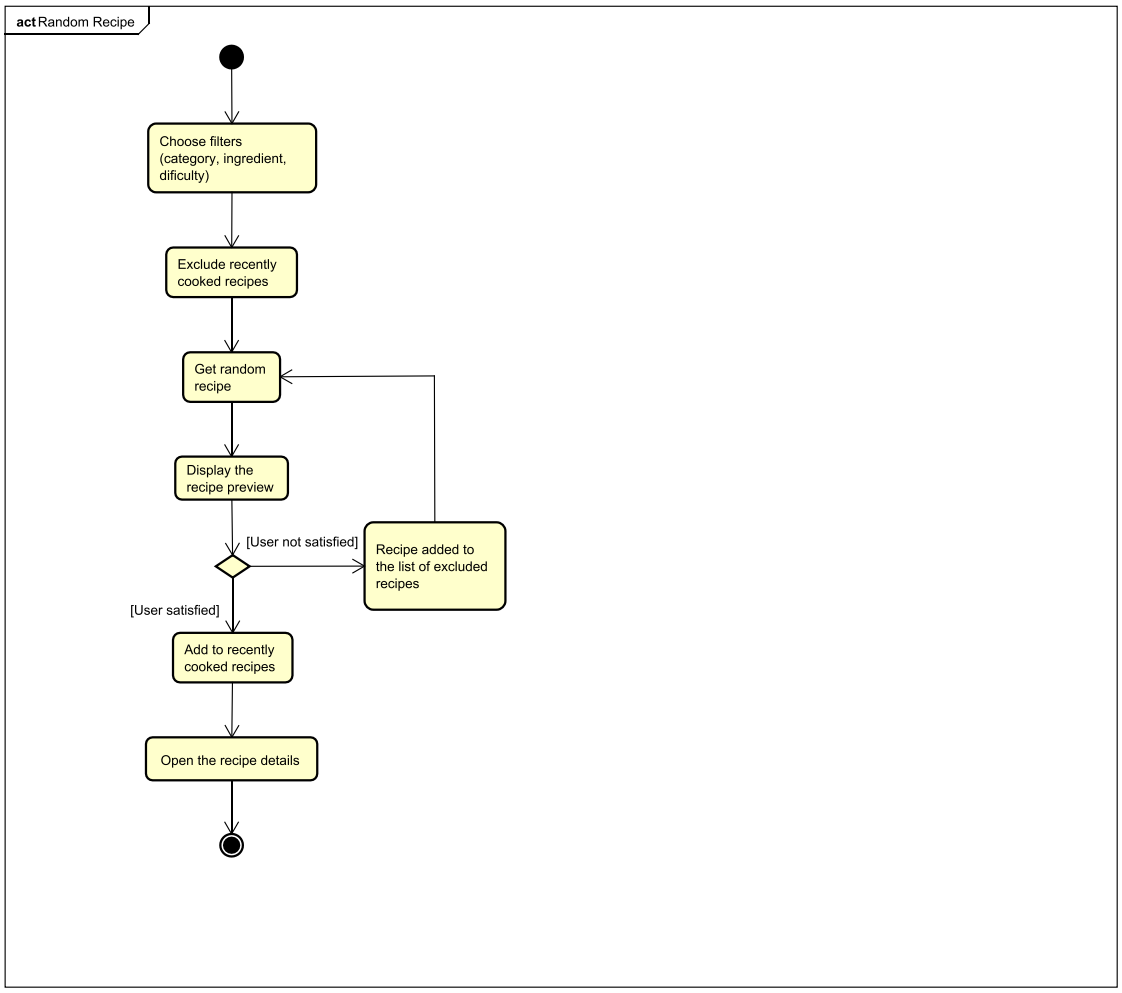


Figure 2.3 - Activity Diagram

The diagram starts with the user providing filter parameters to give a more relevant result. After the irrelevant recipes are filtered out, the recently cooked recipes are removed as well as the rejected results, so the user does not get any unwanted suggestions. In the next step the system randomly picks a recipe that fulfils the criterion. The recipe preview is displayed to the user who can decide whether to open it to see the full details or can opt for a new recipe suggestion. If the user decides to get another suggestion, the displayed recipe is added to the list of rejected recipes to prevent repetition of the suggestions. If the user decides to open the recipe, the recipe is added to the list of recently cooked recipes and the full recipe information is displayed.

2.7 Domain model

Domain Model was created to understand the relationships between the object that take part of the system. The users can add other users as a friend so they can easily share recipes with each other. They also receive notifications, so they are up to date with the incoming events. Users can also add and manage their own recipes as they wish. These recipes consist of ingredients and these ingredients can also be added to the shopping list. Each user has their own shopping list which they can manage any time. This can be seen in the *Figure 2.4* below.

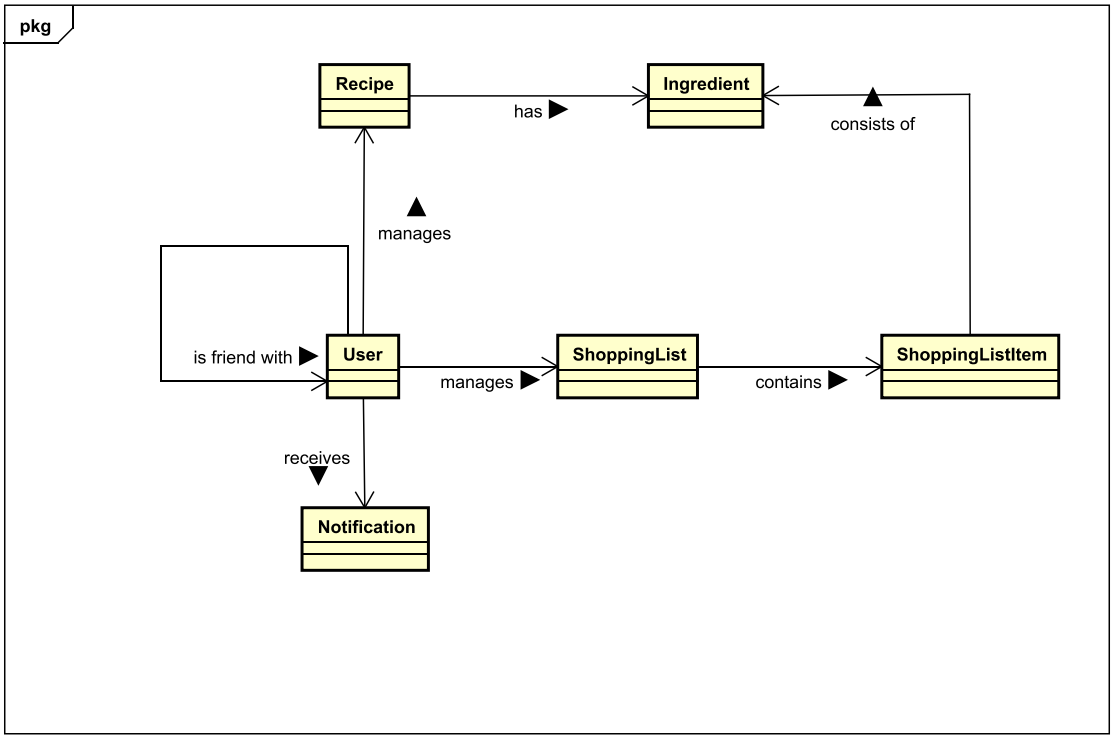


Figure 2.4 - Domain Model Diagram

3 Design

This chapter is dedicated to the decisions made when deciding upon the system architecture as well as the user interface design. It exaplains the reasons behind it and the steps that led towards them. Moreover it discusses individual parts of the system and their responsibilities.

3.1 System Architecture

The system consists of three main parts which are MySQL Database, Appliaction Server and finally the Web Application Client. Since one of the non-functional requirements is that the client application should be platform independent, it targets all the standard screen resolutions, so it is available at personal computers, tablets and mobile phones as well. The approache and means to achieve this goal will be discussed in more detail later in the implementation section. The *Figure 3.1* below, displays the system architecture and the communication between the individual parts.

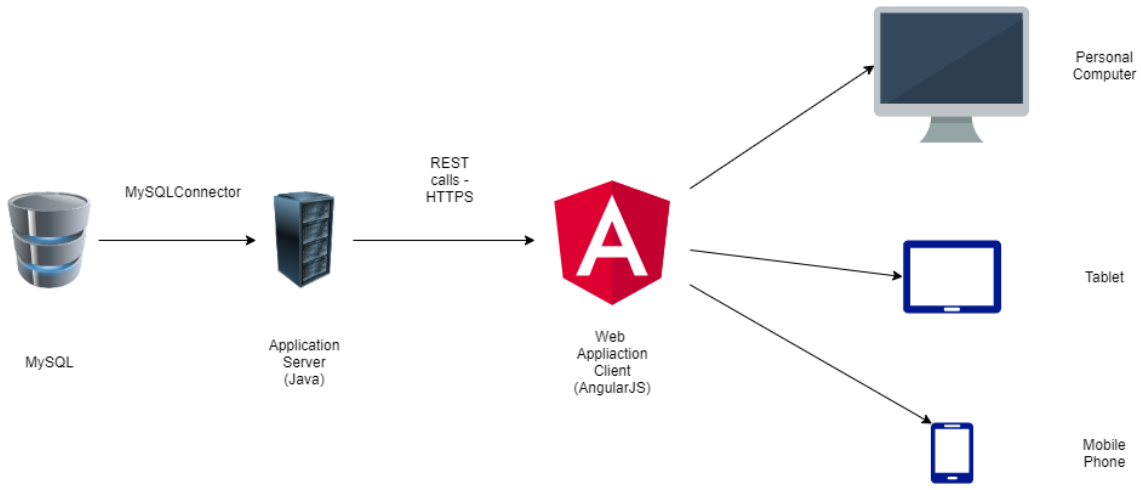
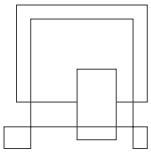


Figure 3.1 - System Architecture Overview

Firs part of the system MySQL database takes the responsibility for the handleling the data and enforces that data logic is persistant.
Second part of the system is the application server. Server hadndles the communication with the database as well as with the file storage system. Server is also



responsible for the computationally heavier tasks so the devices with less resources are not overloaded.
Last but not least is the web application client which is also responsible for all the computationally lightweight tasks. The client is also responsible for the user interface and handleling the user interaction.

3.1.1 Client-Server Communication Architecture

One of the most importat decisions is the client-server communication. This needs to be selected carefuly because different means have different advantages and disadvantages. On one hand web sockets provide bi-directional and full-duplex communication but on the other hand if the communication is not frequent it is just waste of resources. After the evalution of the system's requirements the RESTful architecture was chosen for the following considerations:

1) The communication between the client and the server is rather occasional, so with REST service the server would not be overloaded with the persisten connections.
2) The notifications are the only communication that would be initiated from the server side and since it is only recipes and friend requests (that will not be that frequent), it is reasonable to implement it by polling rather then having open connection all the time.

3.2 Database design

Database design was developed in order to fulfil the requirements stated in the analysis. A relational database was chosen as it fits the needs of this project. As the focus of this project is on the web application a relatively simple database was required. It stores mainly the recipes, shopping list items, notifications etc. Each of them is stored in separated relation – table. These tables are the result of examining the functional requirements. Entity relationship diagram in *Figure 3.2* demonstrates the relationships between the database entities and how they depend on each other.

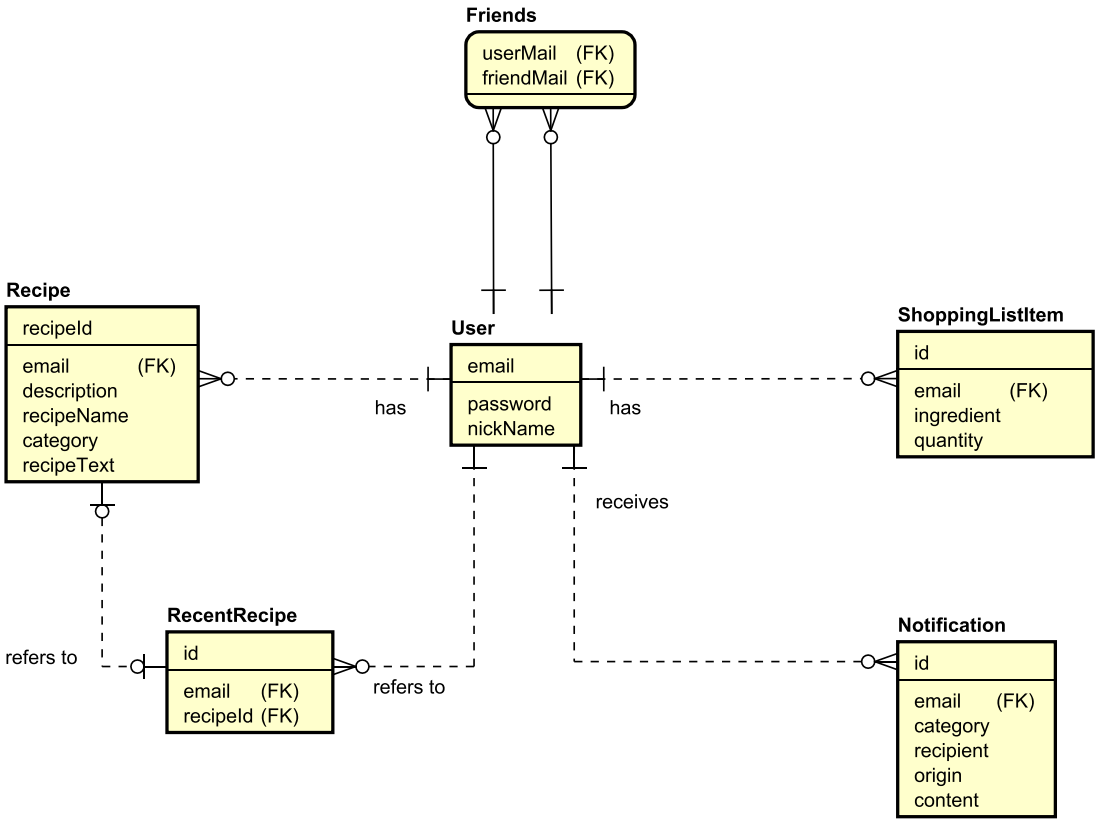
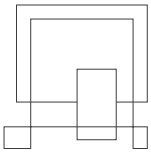


Figure 3.2 - Entity Relationship Diagram

3.3 Class diagram

Class diagram of the back-end in the *Figure 3.3* provides an overview of the infrastructure of the system. All the data is being stored in the MySQL database except the images which are stored in the file system. The server is designed as RESTful server and its end points are separated into the controllers. Each of the controllers is responsible for a specific functionality of the system using mainly GET, POST, PUT and DELETE which are HTTP requests to handle them. Also, they use objects that represent the data when communicating with the database to either store them or forward them to the client. This is done throughout the data access object (DAO) class which is responsible for the CRUD functions to manipulate the data. Achieving this goal is secured by executing the SQL statements. Another important part of the system is ImageManager class. Its purpose is to handle the images of the recipes, specifically to



store, retrieve and remove them from the servers file system. In addition, ImageManager also performs compression of the images to reduce their size and thus help to save resources of the file system.

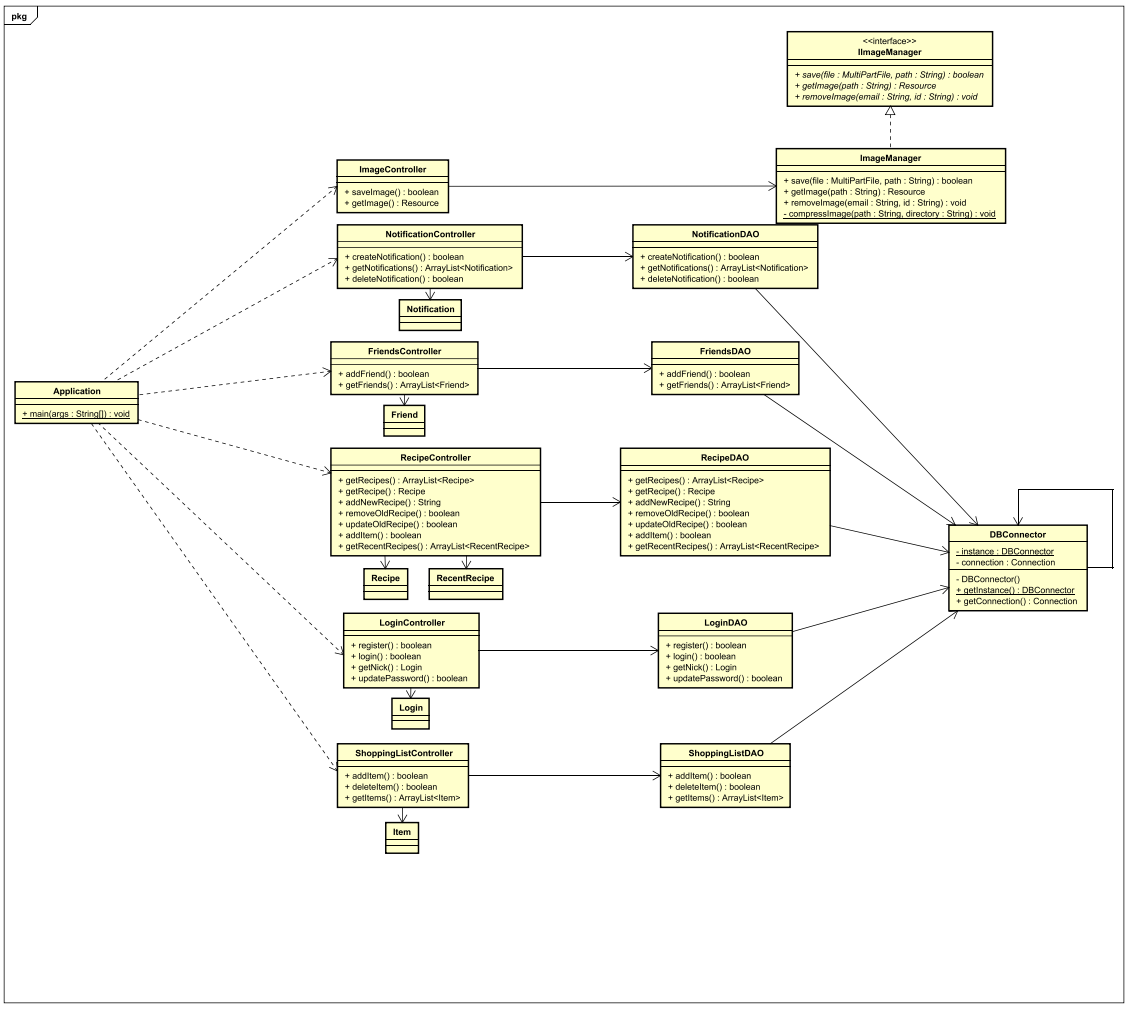
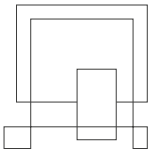


Figure 3.3 - Server Class Diagram

3.4 Sequence Diagram

To demonstrate the interactions between the individual layers of the system a sequence diagram was created. Sequence diagrams contain lifelines of objects and messages exchanged between them. They provide an overview of the order in which the functionalities are executed, depicturing the behaviour of the system over the time. The diagram in the *Figure 3.4 - Create Picture Sequence Diagram* represents the flow



and the messages exchanged between the layers involving the user when creating a new recipe in the system.

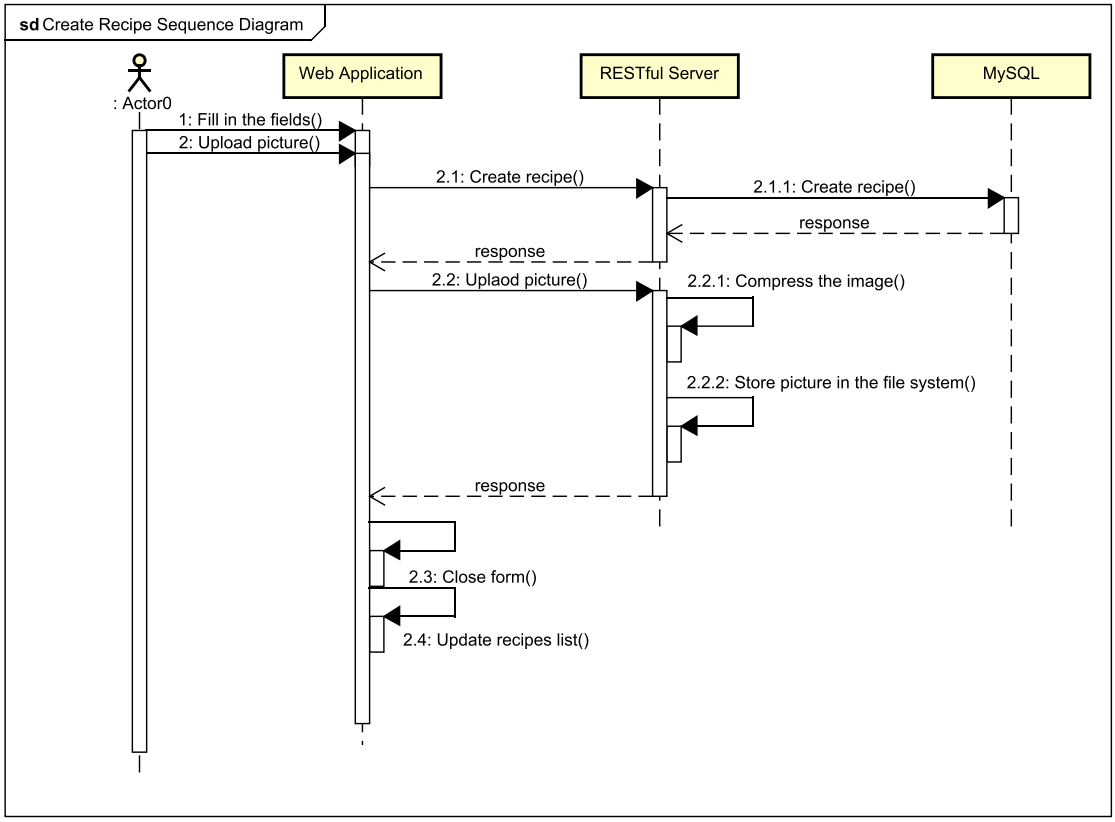
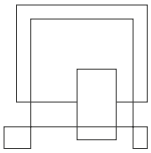


Figure 3.4 - Create Picture Sequence Diagram

Firstly, the user fills in the form and uploads the picture from their device. When the form is submitted, the recipe information is forwarded to the server. Server then tries to store the information in the database and returns the result to the client with the id assigned to the recipe. If the database stores the recipe information successfully, the client uploads the picture linked to the recipe by id to the server where it is compressed and stored in the file system. On completion a response is sent to the client which acts on it by closing the form and updating the list of displayed recipes with the newly created one.



3.5 Design patterns

The focus of this section are the design patterns used in this project to contribute to a cleaner architecture. Design patterns are usually chosen as a standardized solution for certain type of issues. This means they are well known, and the further development is fast even when new developers take over since they are already familiar with it.

3.5.1 Dao

Data access objects (DAOs) are one of the design patterns. Their purpose is to separate the application logic from the data persistence. This means that the application is independent of the implementation of the data retrieval from the database system. Separate DAOs classes were created for these purposes in relation to the business logic. They contain functions to manipulate data in the database using SQL statements. Database is accessed through the DBConnector class.This can be seen at the Figure 3.5 below.

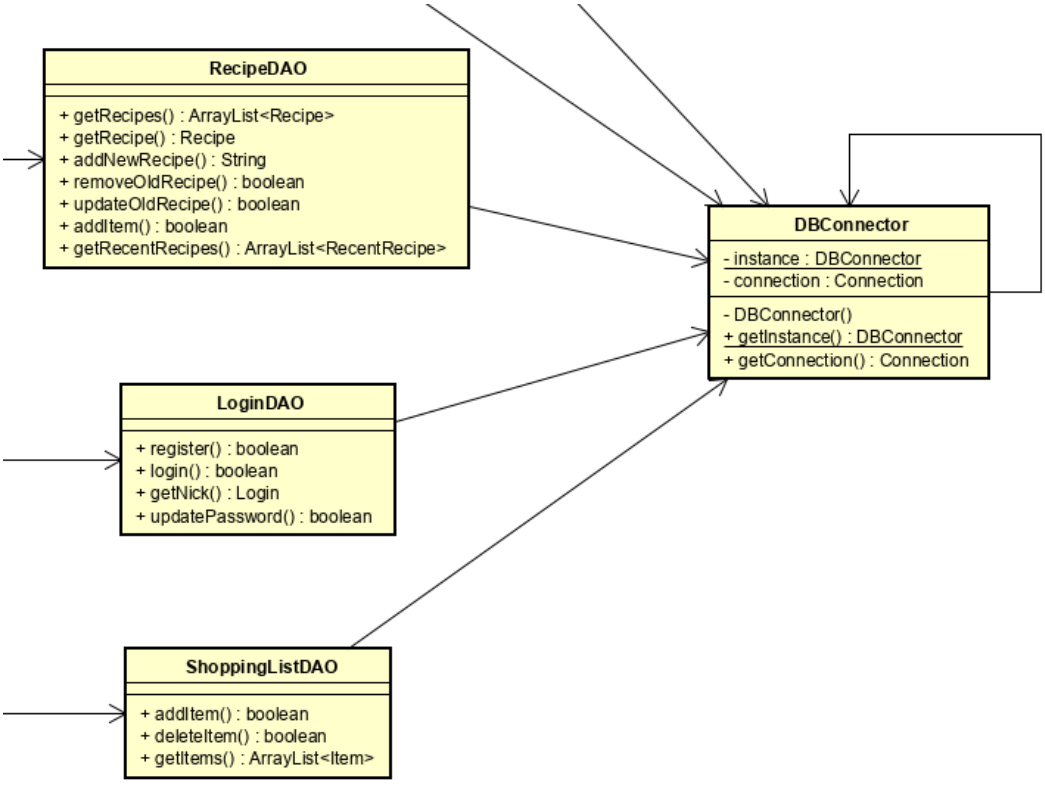
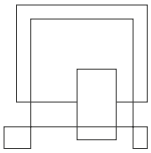


Figure 3.5 Data Access Object Example



3.5.2 MVC

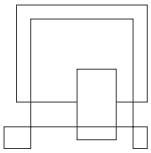
Model view controller architectural design pattern is one of the standards in the software engineering. Its aim is to separate the application logic to provide readability and maintainability. One of the reasons the AngularJS which is a JavaScript framework was chosen in this project is that it enforces the MVC-like pattern in the application and also it makes its implementation easier.
The view consists of HTML templates which handle the interaction with the user. It takes the user input and displays the results of the user actions. It also communicates with the model and updates it when new information is obtained.
The controllers contain the functions and handle the responses to the users input. Controllers also manage the views and display the relevant once. They also define the initial state and behaviour of the data. The main purpose of the controllers is to handle the business logic.
Model in AngularJS is implemented by default as a singleton. This means that there is only one instance containing the data. Model is also a single source of truth which is due to the two-way data binding implemented by AngularJS. This means that the view updates the model and vice versa the model updates the view in real time.

3.5.2.1 AngularJS Service

Switching between different views in the application results in destroying the controllers with the corresponding model. This means that a way to preserve the data is required. This is handled by creating an AngularJS service. The service is view independent and is able to transfer the data from one controller to another. This is used for example to verify if the user is logged in, when trying to access any route of the application.

3.5.3 Interfaces

Last design decision is use of the interfaces. Interfaces assure that certain functionalities of the system will be implemented without the need to modify any other code except its implementation. E.g. IImageManager interface was created and used in the rest endpoints. This means that the REST endpoint is independent of the solution to handle the image management. If the file system is changed to e.g. some external



system, only implementation of the IImageManager Interface needs to be changed with the REST endpoints remining intact.

3.6 Web Application Client Components

Component diagram in the *Figure 3.6* was created to show the individual components of the application and their relationships. Except the Status Service, each of the components is made of the HTML template with corresponding SASS styling and JavaScript controller. Status service holds the information about the user login as well as other data that need to be transferred among the components. Its value is set when the user logs in. As the menu is present in each part of the application after logging in, it checks for the user login every time there is a switch in views using the Status Service.

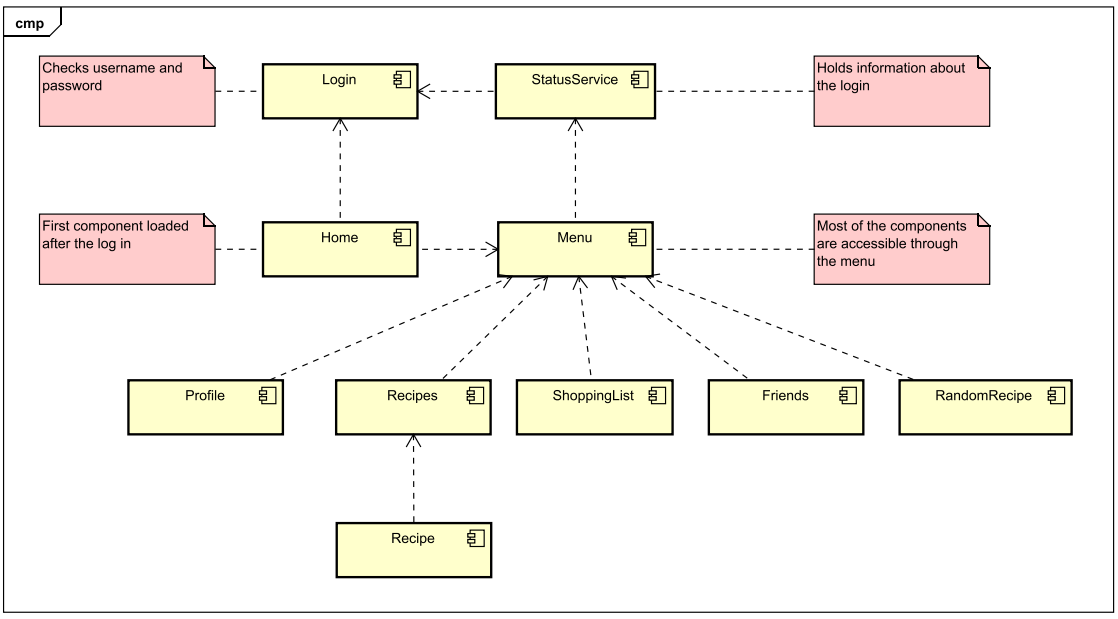
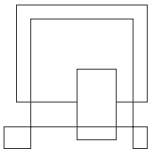


Figure 3.6 - Client Component Diagram



4 Implementation

This chapter provides the technical details and answers the question about how the design was implemented in the code. Code snippets are present, to reveal how the system works under the hood and explain how the different divisions of the system interact with each other. Also, it gives the reasoning behind the choices of the technologies and explains the server configurations as well as the project structure..

4.1 Technologies

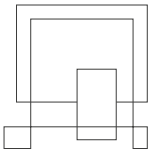
This chapter is dedicated to the technologies used during this project and explains in a higher detail the purpose and the reasons why these tools were selected to achieve a specific properties and qualities of the product. On one hand the main motivation during the tool selection was making the application development as effective as possible but on the other hand the other needs of the project needed to be considered as some other aspects require specific solutions that can be achieved most beneficially by specific tools.

4.1.1 JavaScript

JavaScript was selected as a main programming language for this project as it fits the needs of the single page web applications very well and is supported by most of the modern browsers. It is a high level, multi-paradigm, programming and scripting language suitable for object-oriented programming as well as functional programming. One of its highlights is the ability to dynamically update the content of the webpage without the need to refresh, embracing the interactivity it adds to the web pages. Another benefit is the full integration with the HTML and CSS giving the possibility to change and manipulate the HTML elements as well as to change their styling.

4.1.2 AngularJS

In simplicity AngularJS is a JavaScript framework also written in JavaScript. Possibility to extend HTML attributes with AngularJS directives is one of the core functionalities



among the others. Moreover, it offers option to create customize directives as well. These directives are used to create components which introduces another level of code reusability together with option to control behaviour and complex DOM structures. Another advantage is the way AngularJS handles two-way data binding easily with directives or expressions. Furthermore, it implements the MVC pattern in HTML for the view and in JavaScript for the model with controller. This framework was selected to make a noticeable progress in shorter time, provide better code organization and maintainability.

4.1.3 Jasmine

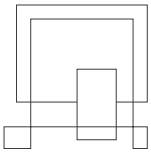
Jasmine is a BDD (behaviour-driven development) framework for JavaScript that was chosen as a tool for creating the unit tests in the client-side of the application. Jasmine is open-source and due to no external dependencies also adds value by its low overhead. Plus, its build-in and optionally custom matchers make it relatively simple to implement a unit test in a short period of time.

4.1.4 Karma

Karma is an opensource tool that is able to spawn a web server and run the tests in a browser. It also offers a possibility to set up which browser it should be tested on. The results and the count of how many tests passed and the errors of failed tests are displayed in the command line. One if the greatest advantages is option to register the files in its configuration which are being watched. If a file changes a test is triggered so the developers are able to see the effect of their code changes basically in real time. Also, the spawned web server collects the results of the tests and provides an overview to the developers.

4.1.5 HTML

HTML also known as Hyper Text Mark-up Language is a standard mark-up language which was used to build a skeleton of the front-end part of this project. It can be said



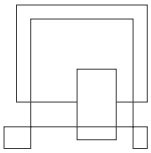
that it provides a depth of the view by defining the hierarchy of the items. It structures and defines all elements on the website from meta tags through document body to smallest paragraphs. This is done with the use of the so-called tags which are specific for each element type on the HTML page (e.g. images, links, divisions, paragraphs etc.).

4.1.6 CSS and SCSS

To achieve a higher code organization and readability the SCSS was chosen instead of the classical CSS for the project. SCSS stands for Sassy CSS and is similarly like SASS (Syntactically Awesome Style Sheets) an extension of CSS. One of the main advantages of SCSS is a possibility to create variables which adds flexibility and more importantly maintainability to the code. Another advantage is a support of nested syntax which can be used to reflect the HTML tag hierarchy and provide a more natural structure of the code. And least not last is the option to divide the code into files separated by business logic which adds even better code structure. The important functionality is that these files are afterwards merged during the compilation into a single CSS file. This results into a single HTTP request to obtain all the styles from the server.

4.1.7 Java

Java is object oriented, class-based, general-purpose and statically-typed programming language which is not compiled directly into the machine code but instead into the so-called byte code. This byte code is further processed by JVM (Java Virtual Machine) introducing platform independency. Java was used for the backend of the project and together with Spring Boot provide a solution to handle all the request from the client side (e.g. image storage in the file system, accessing the database etc.).



4.1.8 JUnit

Similarly like in JavaScript, the need for unit testing also needs to be fulfilled. That is when JUnit comes in. It is one of the most often used unit testing frameworks for Java. The tests can be easily implemented in the test classes using the relevant annotations. The version utilized in this project is JUnit 5.

4.1.9 Spring Boot

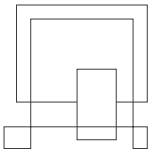
Spring Boot is a Java based, open source framework picked for this project since it reduces the development process of web applications and testing significantly. It is used together with Java to create RESTful web service which is the core of the project architecture. It also provides an embedded web server (Tomcat) by default. Even though this server is pre-configured it is possible to change the configuration settings to fulfil desired behaviour.

4.1.10 Apache Tomcat

Apache Tomcat is an opensource Java server container used as a web server or for hosting Java servlets.

4.1.11 IntelliJ IDEA

IntelliJ IDEA is an integrated development environment. It was chosen for this project since it provides a powerful tools and support during the process of software development of a single page web application as well as its back-end side Spring Boot server. Essentially error checking, syntax highlighting and autocompletion contribute to accelerating the overall performance. Remote debugger and integration of the version control system speeds up the process tremendously.



4.1.12 GitHub

GitHub is primarily a hosting service for repository of version control system Git. This tool's main purpose is to ease the collaboration between the team members and any other contributors to the project. With the same importance it provides a system to control the versions of the system as well as track the progress and check the changes done. Among its leading functionalities is opportunity to make so called pull request which adds extra layer of quality assurance.

4.1.13 MySQL and SQL

SQL (Structured Query Language) is a standard language designed for data manipulation. These data are stored in objects called relations or tables which consist of tuples / rows and attributes / columns. MySQL is an SQL relational database management system (RDBMS). Security and high reliability are one of the reasons why it became globally popular and created a massive community that a developer can benefit from speeding up the whole process. MySQL is opensource and in this project it serves as a web database benefiting from its high performance and good scalability.

4.2 Serving Static Web Content and Settings

4.2.1 Project Structure

In Gradle project, the project structure has a high importance. By default, the static content is loaded from "static" folder created in "resources" directory. This means that whole client is stored in this folder.

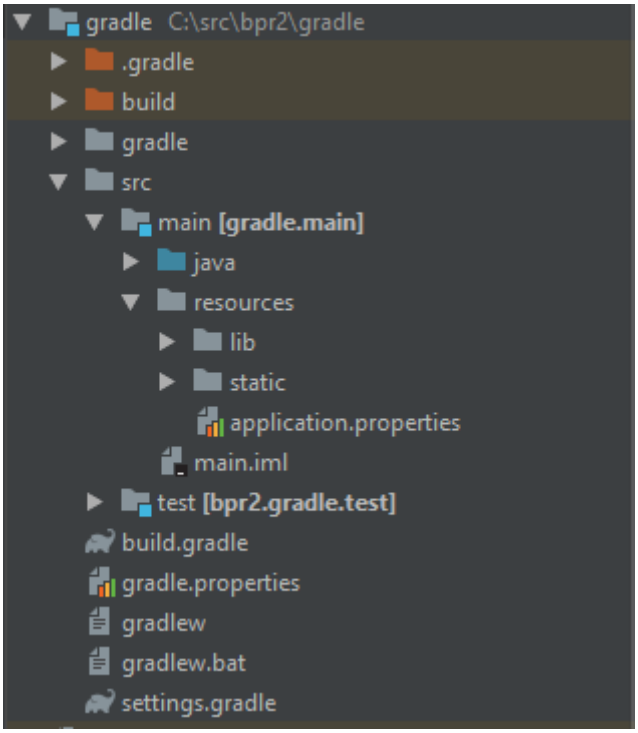


Figure 4.1 - Gradle Project Structure

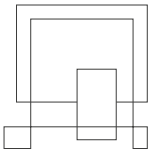
4.2.2 Configurations

Another setting required to provide static content are server configurations. These are set in WebConfig class. Following Figure 4.2 displays how the system recognizes the required files.

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Bean
    public ViewResolver jspViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/static/");
        viewResolver.setSuffix(".html");
        return viewResolver;
    }
```

Figure 4.2 - Web Configuration

The routing when the page is loaded is handled by Home Controller class. It is configured to open index.html file when the user access the default or “/home” URL.



```
@Controller
public class HomeController {
    @RequestMapping("/home")
    public String home() { return "index"; }
}
```

Figure 4.3 - Home Controller

Additional spring configurations are set in the “application.properties” file. Specifically, the size of the images that are posted or requested is set to 4MB.

```
spring.servlet.multipart.max-file-size=4MB
spring.servlet.multipart.max-request-size=4MB
```

Figure 4.4 - Image Size Restriction Configuration

4.3 Code Implementation

4.3.1 AngularJS and JavaScript

4.3.1.1 Login

Login functionality was implemented to verify the registered users. The login form in *Figure 4.5* is located at the main page after visiting the web application. User can log in by filling in the email with password and pressing the Login button.

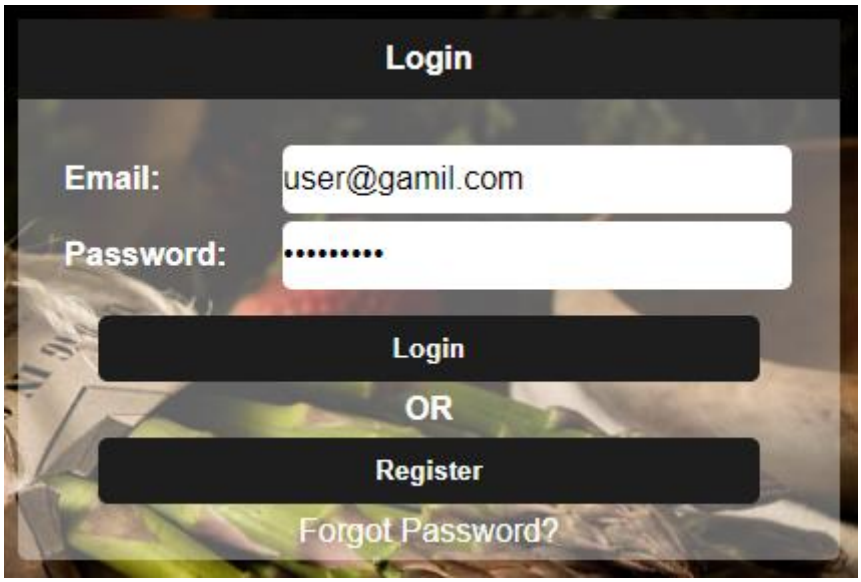
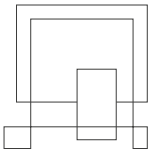


Figure 4.5 - Login Form



This functionality is contained in the login controller which was created with AnglurJS function as shown in the *Figure 4.6* below. Status service is added to the controller via dependency injection for a later use.

```
angular.module( name: 'foodAssistant')
    .controller('LoginCtrl', ['$scope', '$window', 'statusService', '$http',
        function($scope, $window, statusService, $http) {
```

Figure 4.6 - Login Controller

In the *Figure 4.7* we can see the implementation of the login function. Inside the function an HTTP post request sends the data encapsulated in an object to the server using the REST endpoint. If the user was verified successfully the status in the Status Service is set to true as well as additional information that are used in different parts of the applications like email and nickname. Last step is redirection to home view using $window directive.

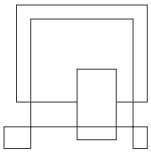
```
$scope.login = async function () {
    $http.post( url: SERVER_URL + '/login', JSON.stringify(
        value: {email:$scope.emailLogin, password:$scope.passwordLogin}))
        .then(function (result) {
            if (result.data) {
                statusService.setLoggedIn(true);
                statusService.setEmail($scope.emailLogin);
                statusService.setNick();
                $window.location.href = "#!home";
            }
        }).catch( onrejected: function (e) {
        console.error(e);
    });
};
```

Figure 4.7 - Login Function

As displayed in the *Figure 4.8* , the direct accessing of the application components is not possible without logging in at first. The user will be redirected automatically to the main page.

```
if(!statusService.getLoggedIn()) {
    $window.location.href = "#!/";
}
```

Figure 4.8 - User Redirection



This functionality is presented in the menu directive as it is present in all the components. Menu is a custom directive created in AngluarJS and is linked to its own HTML template as pictured in the following *Figure 4.9*.

```
.directive('topMenu', function () {
    return {
        templateUrl: 'top-menu.html',
        controller: 'TopMenuCtrl'
    };
});
```

Figure 4.9 - Custom Directive

As mentioned in the design section, Status Service holds the data that are needed outside of the Angular controllers. It consists of a few fields and the functions to manipulate them which can be seen in *Figure 4.10*.

```
angular.module( name: 'foodAssistant')
    .service('statusService', ['$http', function ($http) {
        let loggedIn = false;
        let email = '';
        let nick = '';
        let recipe = {};

        return {
            getLoggedIn: function () {
                return loggedIn;
            },
            setLoggedIn: function(value) {...},
            getEmail: function () {...},
            setEmail: function(value) {...},
            getRecipe: function () {...},
            setRecipe: function(value) {...},
            getNick: function () {...},
            setNick: function() {...}
        };
    }]);
```

Figure 4.10 - Status Service

4.3.1.2 Shopping List

It is possible to add ingredients into the shopping list directly from the recipe description. If the user clicks on one of the ingredients a pop-up window appears with field for quantity as shown in the *Figure 4.11 - Quantity Field pop-up*.

Figure 4.11 - Quantity Field pop-up

This was implemented by creating an HTML template containing input and button elements. This HTML code is then compiled so the AngularJS is aware of the new code and afterwards it is appended to the document body. As result a pop-up window appears when the code is executed. CSS styling was used to position the pop-up box right next to the ingredient for more intuitive user experience. This is achieved by using event element which contains the information about the ingredients text position. The pop-up disappears when it loses the focus or when the ingredient is added to the shopping list. This is captured in the *Figure 4.12*.

```
$scope.addToShoppingList = function (ingredient, event) {
    const htmlString = '\n' +
        '<div id="quantity-input">\n' +
            'Quantity: <input type="text" id="add-item-input" ng-model="itemQuantity" ng-init="0">\n' +
            `<button id="addItemButton" class="orange-button" ng-click="addItem('${ingredient}')">` + 'Add</button>\n' +
        '</div>';

    const html = $compile(htmlString)($scope);
    angular.element(document.body).append(html);

    $('#quantity-input').css( name: { top: event.clientY, left: event.clientX });
    const inputField = $('#add-item-input');
    inputField.focus();
    inputField.focusout(function (e) {
        if (e.relatedTarget === null || e.relatedTarget.id !== 'addItemButton') {
            $('#quantity-input').remove();
        }
    });
};
```

Figure 4.12 - Quantity Field Pop-up Implementation

4.3.1.3 Create recipe

A new recipe can be created by filling in a recipe form in *Figure 4.14*. This can be accessed from any part of the system as this functionality is part of the top menu as can be seen in the figure below.

Figure 4.13 - Top Menu - Add Recipe Button

The form contains input fields as well as an option to upload an image. Also form for adding ingredients can be accessed from here – *Figure 4.11*. The ingredients can be added by typing in the ingredient name and pressing the “plus” button in the right bottom corner.

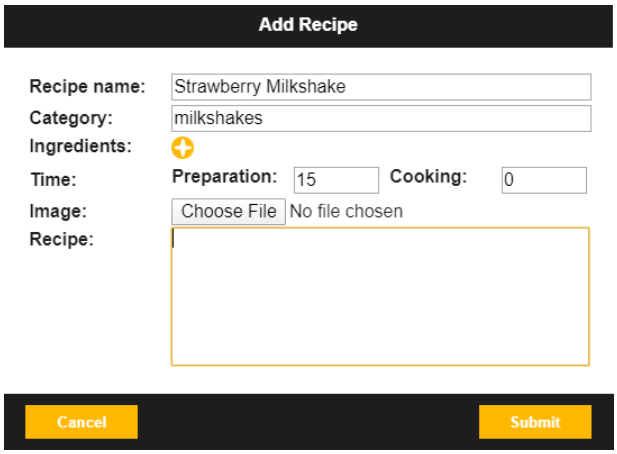


Figure 4.14 - Add Recipe Form

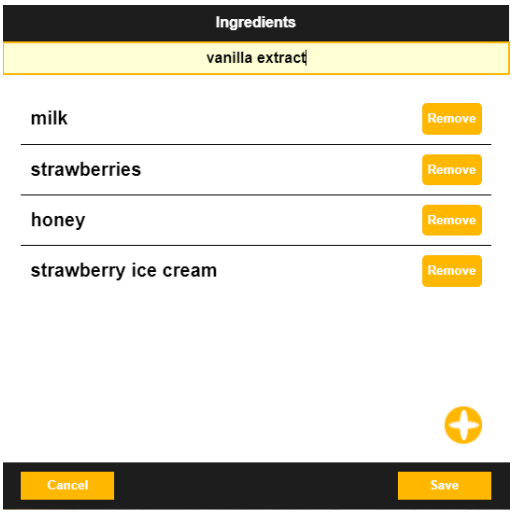
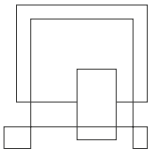


Figure 4.15- Add Ingredients Form

Similarly like the ingredient pop-up, the form for adding recipes and ingredients are HTML strings compiled and appended to the document body. This can be seen in the following *Figure 4.16*.



```
$scope.ingredientsForm = function () {
    const htmlString = '\n' +
        '<div id="ingredients-form">\n' +
            '<div id="reg-top"><b>Ingredients</b>\n' +
            '</div>' +
            '<div id="ingredient-content">\n' +
                '<div id="ingredient-adder">\n' +
                    '<input id="ingredient-input" ng-model="ingredientInput">\n' +
                '</div>' +
                '<ul id="ingredients-list">\n' +
                    '<li ng-repeat="ing in ingredientsList track by $index"><b>{{ing}}</b> ' +
        '<button class="orange-button" ng-click="removeIngredient($index)">Remove</button>\n' +
                    '</li>' +
                '</ul>' +
                '<button id="add-new-ingredient" ng-click="addToList()">\n' +
                '</button>' +
            '</div>' +
            '<div class="bottom-form dark-bottom">\n' +
                '<button id="regCancel" ng-click="closeIngredientsForm()"><b>Cancel</b></button>\n' +
                '<button id="regSubmit" ng-click="saveIngredients()"><b>Save</b></button>\n' +
            '</div>\n' +
        '</div>';

    const html = $compile(htmlString)($scope);
    angular.element(document.getElementById( elementId: "recipe-modal")).append(html);
};
```

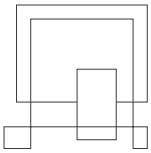
Figure 4.16 - Ingredients Pop-up

When the ingredients are saved, they are converted to a single string holding all the recipe ingredients. This is done by concatenation of the ingredient strings and separating them by coma - *Figure 4.17*.

```
$scope.saveIngredients = function () {
    let ingredientsString = '';
    $scope.ingredientsList.forEach( callbackfn: ing => {
        ingredientsString += ing + ',';
    });
    $scope.ingredients = ingredientsString.slice(0, -1);
    $scope.closeIngredientsForm();
};
```

Figure 4.17 - Save Ingredients Function

As shown in the sequence diagram in the previous chapter in *Figure 3.4*, when adding a recipe at first the recipe information is sent to the server. This is done via HTTP request consisting of defining attributes. Method specifies the request type to be POST as the intention is to add new data. URL sets the address of the server with the route for specific end point. Headers identify the application content and data attribute holds the information that the server will process further. The rest calls return a promise, if



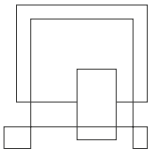
the promise is resolved successfully the list of recipes is updated. This is followed by selecting the recipe picture uploaded by the user and appending it to the form data object. Image in such format can be send through the subsequent POST request. If the request completes successfully the form is closed as displayed in *Figure 4.18*.

```
$scope.addNewRecipe = function () {
    const req = {
        method: 'POST',
        url: SERVER_URL + '/addNewRecipe',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        data: JSON.stringify( value: {email:statusService.getEmail(), recipename:$scope.recipename,
            category: $scope.category, recipetext: $scope.recipetext, ingredients: $scope.ingredients,
            preparation: $scope.preparation, cooking: $scope.cooking})
    };

    $http(req).then(function(response){
        if (typeof $scope.getRecipes === "function") {
            $scope.getRecipes();
        }

        const i = $('#recipeImage').prop( name: 'files')[0];
        let image = new FormData();
        image.append( name: 'file', i,  fileName: response.data + '.jpg');

        $http.post( url: SERVER_URL + '/image?email=' + statusService.getEmail(), image,  config: {
            headers: {'Content-Type': undefined}
        }).then(function () {
            //$('#recipeImage')
            $('#recipe-modal').remove();
        }).catch( onrejected: function (e) {
            console.log(e);
        });
    }).catch( onrejected: function (e) {
        console.error(e);
    });
};
```

Figure 4.18 - Add Recipe Implementation

4.3.2 SASS

One of the non-functional requirements is that the system shall be responsive. This was achieved by implementing the media query technique using the @media rule. This way the properties of the HTML elements can be changed automatically when certain conditions are met. In this case the rule was the screen resolution below 1000px and 600px.The implementation in the code can be seen in the *Figure 4.19* for the resolution of 1000px and in the *Figure 4.16* for the resolution of 600px.



```
@media only screen and (max-width: 1000px) {
  #top-menu {

    #inline-menu {
      display: none;
    }

    #search-field {
      margin-left: 15px;
      min-height: 30px;
      width: 26%;
    }
    #search-button {
      margin-right: 15px;
    }

    .small-icon {
      display: inline-block;
    }
  }

  .recipe-content {
    width: 80%;
  }

  #ingredients-form {
    width: 70%;
  }
}
```

Figure 4.19 - Max Width 1000px

```
@media only screen and (max-width: 600px) {
  .large-screen {...}

  .small-screen {...}

  #notifications-button {
    width: 40px;
    height: 60px;
    font-size: 20px;
  }

  #top-menu {
    #search-field {
      width: calc(100% - 250px);
    }

    .small-icon {
      display: none;
    }
  }

  .recipe-content {
    min-height: 380px;
    width: 100%;
  }

  #ingredients-form {
    width: 100%;
  }
}
```

Figure 4.20 - Max Width 600px

In the ***Error! Reference source not found.*** you can see the form for adding recipe before any break point is reached.

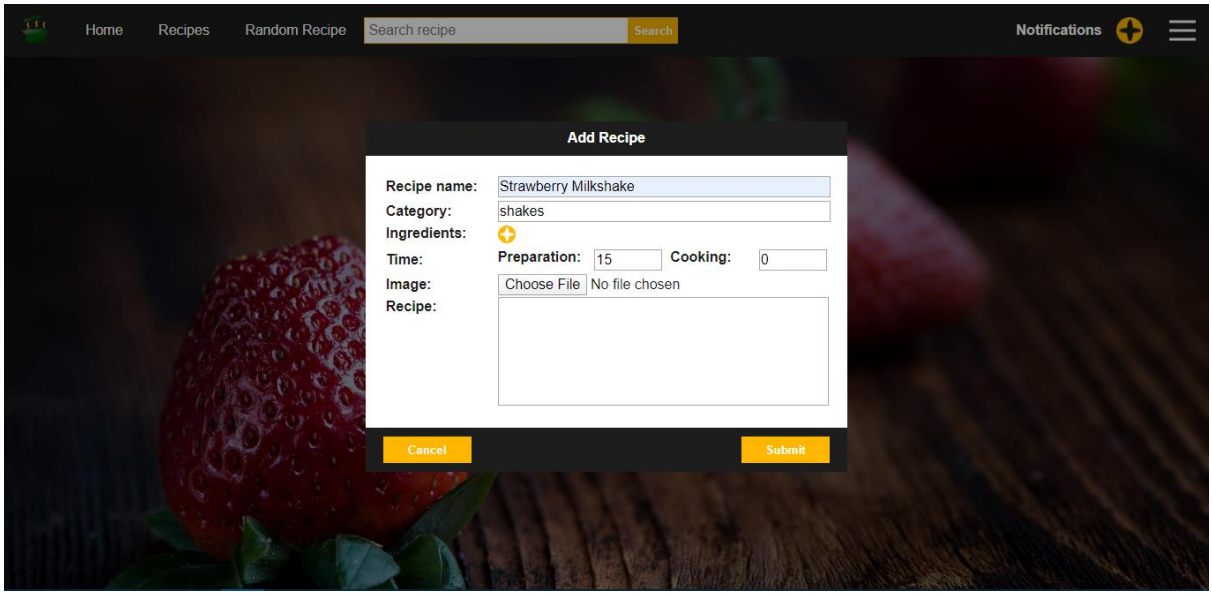


Figure 4.21 - Resolution above 1000 pixels

Figure 4.22 shows how the user interface changes when the screen resolution is below 1000 pixels and *Figure 4.23* displays UI when it is below 600 pixels.

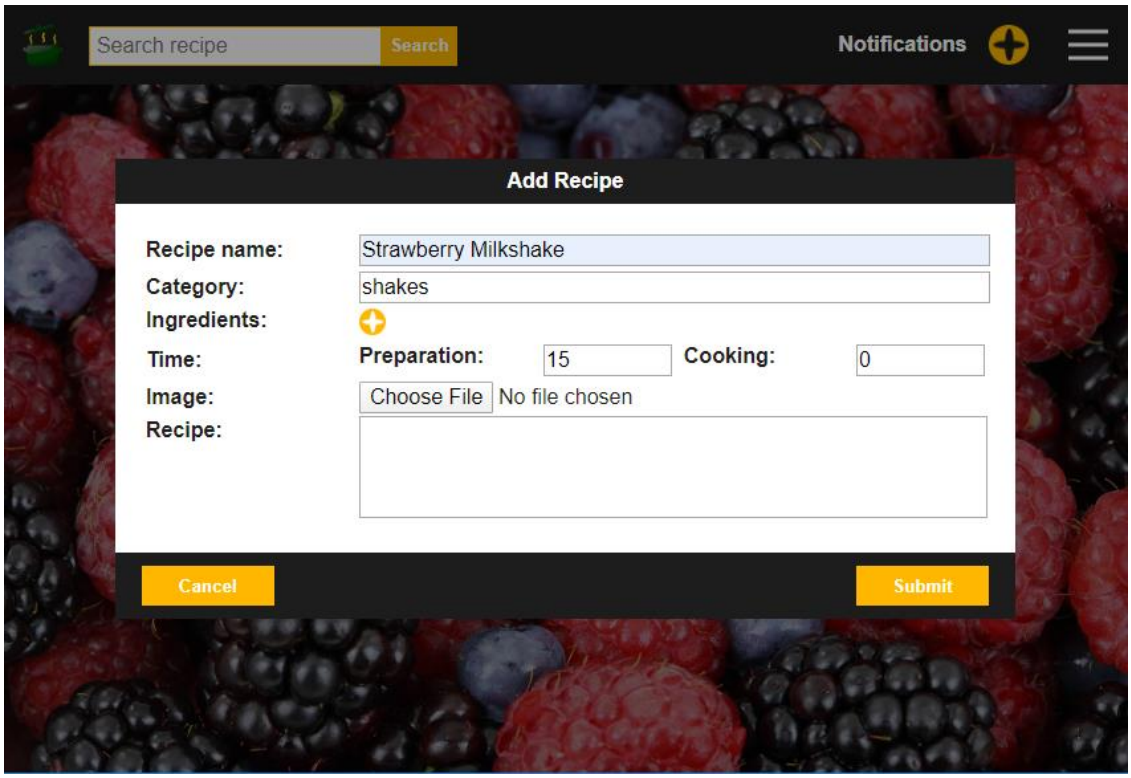


Figure 4.22 - Resolution below 1000 and above 600 pixels

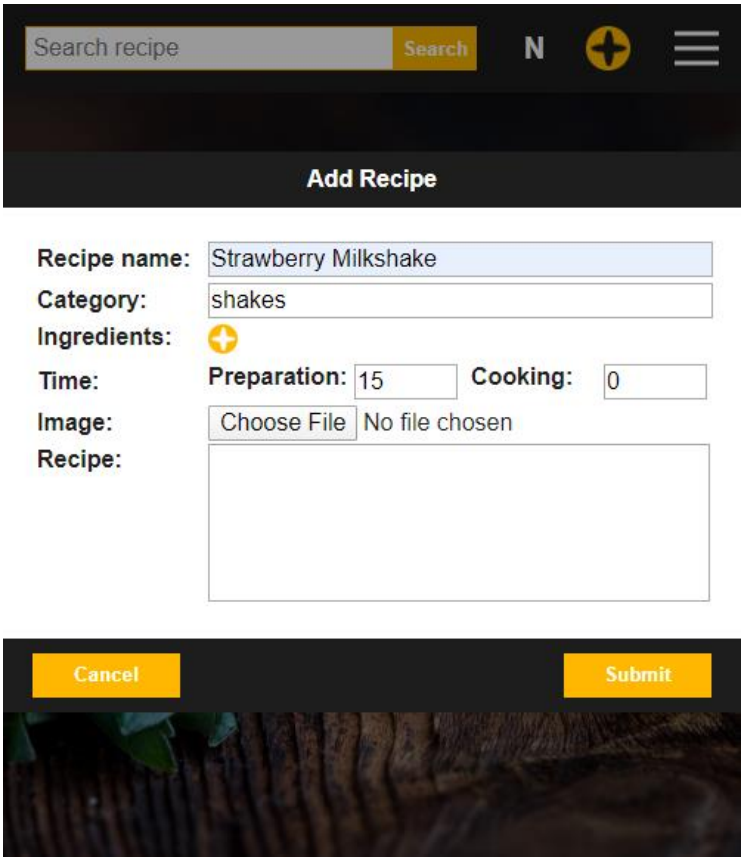


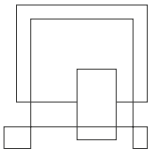
Figure 4.23 - Resolution below 600 pixels

4.3.3 Java

The server implementation is done using Spring framework and consists of REST endpoints. Each of the endpoints has specific path mapping through which can be accessed from the client. The request method and type of the data it deals with is also defined here using @RequestMapping decorator. The function handling the request returns a string containing the id of the recipe created because it is needed for saving the linked picture of the recipe. In the function body we can see that DAO is used to handle the data.

```
@CrossOrigin(origins = URL)
@RequestMapping(path = "/addNewRecipe", method = RequestMethod.POST, consumes = "application/json")
public String addNewRecipe(@RequestBody Recipe recipe) {
    return d.addRecipe(recipe.getEmail(), recipe.getRecipename(), recipe.getCategory(),
            recipe.getRecipetext(), recipe.getIngredients(), recipe.getPreparation(), recipe.getCooking());
}
```

Figure 4.24 - REST Endpoint



Inside the DAO class, the data is used to create an SQL statement which is then executed and stores the new data in the database - *Figure 4.25*. It is followed by query that retrieves the id of the newly created recipe. It needs to be done separately as the database assigns the IDs automatically.

```
public String addRecipe(String emailValue, String name, String category, String text,
                        String ingredients, int preparation, int cooking) {
    try{
        Statement stmt = connection.createStatement();
        String sql = "INSERT INTO assistant.recipes (email, recipename, category, recipetext," +
                "  ingredients, preparation, cooking) " +
                "VALUES (\"" + emailValue + "\", \"" + name + "\", \"" + category + "\", \"" +
                text + "\",\""+ ingredients + "\", \"" + preparation + "\", \"" + cooking + "\")";
        stmt.executeUpdate(sql);

        Statement stmt2 = connection.createStatement();
        String sql2 = "SELECT LAST_INSERT_ID();";
        ResultSet rs2 = stmt2.executeQuery(sql2);
        rs2.next();
        return rs2.getString( columnIndex: 1);
    }catch(Exception e){ System.out.println(e); return "";}
}
```

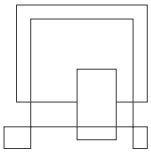
Figure 4.25 - Data Access Object Implementation

Next the image is received through another endpoint. This is implemented also in its own rest controller with a difference that it uses Image Manager class instead of the DAO as it is not accessing the database but the file system instead - *Figure 4.26*.

```
@RestController
public class ImageController {
    private IImageManager imageManager = new ImageManager();

    @RequestMapping(path = "/image", method = RequestMethod.POST, consumes = "multipart/form-data")
    public boolean saveImage(@RequestParam("file") MultipartFile file, @RequestParam(value = "email") String email) {
        return imageManager.save(file,  path: "C:\\assistant\\pictures\\" + email);
    }

    @RequestMapping(path = "/image", method = RequestMethod.GET, produces = MediaType.IMAGE_JPEG_VALUE)
    public Resource getImage(@RequestParam(value = "email") String email, @RequestParam(value = "id") String id) {
        return  imageManager.getImage( path: "file:C:\\assistant\\pictures\\" + email + "\\" + id);
    }
}
```

Figure 4.26 - Image Controller

The image is saved to the folder created specifically for each user by their email address. Afterwards the image is compressed.

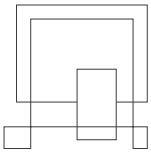


```
public boolean save(MultipartFile file, String path) {
    Path filepath = Paths.get(path, file.getOriginalFilename());
    try (OutputStream os = Files.newOutputStream(filepath)) {
        os.write(file.getBytes());
        compressImage(filepath.toString(), path);
        return true;
    }
    catch (Exception e) {
        System.out.println(e);
        return false;
    }
}
```

Figure 4.27 - Save Function

4.3.3.1 Compression

As can be seen in the *Figure 4.28* below, in the first step the picture is located, new name is created, and the target picture is renamed so the compressed version can keep the name linked to the recipe. New file object is created, and the compression quality is defined as 0.6. The compression is done using javax package specifically ImageIO class. After the initialization of the streams (input for the original image and output for the new one) a suitable image writer is found. The parameters are set on the ImageWriteParam object. The writer is used to write the compressed image and takes the ImageWriteParam as parameter. The result is a new compressed image. The final step is closing the streams and the old image deletion.

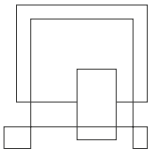


```
private static void compressImage(String path, String directory) throws Exception{
    File target = new File(path);
    File newName = new File( pathname: directory + "\\target.jpg");
    target.renameTo(newName);
    File oldImg = new File( pathname: directory + "\\target.jpg");
    File newImg = new File(path);
    float quality = 0.6f;

    InputStream inputStream = new FileInputStream(oldImg);
    OutputStream outputStream = new FileOutputStream(newImg);
    BufferedImage bufferedImage = ImageIO.read(inputStream);

    Iterator<ImageWriter> imageWriters = ImageIO.getImageWritersByFormatName("jpg");
    if (!imageWriters.hasNext()) {
        throw new IllegalStateException("System could no find any writer.");
    }

    ImageWriter imageWriter = imageWriters.next();
    ImageOutputStream imageOutputStream = ImageIO.createImageOutputStream(outputStream);
    imageWriter.setOutput(imageOutputStream);
    ImageWriteParam imageWriteParam = imageWriter.getDefaultWriteParam();
    imageWriteParam.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
    imageWriteParam.setCompressionQuality(quality);
    imageWriter.write( streamMetadata: null, new IIOImage(bufferedImage,  thumbnails: null,  metadata: null), imageWriteParam);

    inputStream.close();
    outputStream.close();
    imageOutputStream.close();
    imageWriter.dispose();
    oldImg.delete();
}
```

Figure 4.28 - Compression Implementation

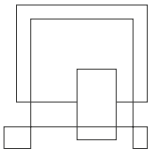
4.3.4 SQL

The *Figure 4.29* displays how the relation holding the recipes was created by following the *Figure 3.2 - Entity Relationship Diagram* . Recipeid column, which is the primary key, is as mention before, handled by the database by autoincrementing. All the attributes cannot be null to assure all the functionalities of the system will be available to the user. All of the columns have a data type specified as well as restrictions on their sizes. The email in the recipe is a foreign key from the UserLogin table to assure a relationship between the users and their recipes.



```
CREATE TABLE Recipes (
recipeid int PRIMARY KEY NOT NULL AUTO_INCREMENT,
email VARCHAR(40) NOT NULL,
recipename VARCHAR(70) NOT NULL,
category VARCHAR(25) NOT NULL,
recipetext TEXT NOT NULL,
ingredients VARCHAR(100) NOT NULL,
preparation int(11) NOT NULL,
cooking int (11) NOT NULL,
FOREIGN KEY (email) REFERENCES UserLogin(email)
)
```

Figure 4.29 - Recipes Table Implementation



5 Test

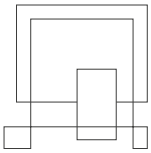
People in general including the developers are prone to make logical mistakes. This natural property might lead to situations that are undesirable and should be avoided. One of the preventions is to make sure that the actions done did not lead to any unsatisfactory scenarios – simply by testing. This chapter talks about how the software was tested to assure that the product was implemented correctly in the preceding chapter and that the customer's needs will be fulfilled in the broadest scale.

5.1 Unit Testing

The purpose of unit testing is to test the smallest parts – functions of the system. Localizing any theoretical system failures in smallest possible unites leads to easier problem identification. Also, the bigger the project gets the harder is to keep track of the relation between the code and the functionalities which might result in breaking one functionality by fixing other one. This approach for error detection in early stages can prevent the system failures at critical stages.

5.1.1 Java unit testing

The functionalities of JUnit were embraced when implementing the unit tests for java classes. The code snippet in the *Figure 5.1 - JUnit 5 Test Implementation* shows how the test for rest and point for retrieving a recipe from the database was implemented.



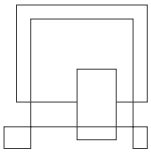
```
public class RecipeControllerTest {

    private HttpResponse response;

    @BeforeEach
    public void init() throws IOException {
        HttpUriRequest request = new HttpGet( uri: "http://localhost:8080/getRecipes?email=mail@test.com" );
        response = HttpClientBuilder.create().build().execute( request );
    }

    @Test
    public void mimeType() {
        assertEquals( expected: "application/json", response.getEntity().getContentType().getElements()[0].toString() );
    }

    @Test
    public void statusCode() {
        assertEquals( response.getStatusLine().getStatusCode(), HttpStatus.SC_OK );
    }
```

Figure 5.1 - JUnit 5 Test Implementation

As you can notice a new class was created specifically for this purpose. It contains the test cases which are annotated with the @Test annotation. Inside this tests a JUnit's function *asserEquals* is used to compare the expectation to the real result of a function. To be more specific in the first case the REST call response is tested to return a result with the correct content type which is expected to be an application/json. In the second test the status code of the response is tested. In this case the expected value is two hundred which stands for successful request. As you can see, if any code needs to be executed multiple times or shall be repeated in the individual unit tests it can be moved to a function that is executed before each of the tests. This function however needs to be annotated with the @BeforeEach annotation as ordered in the JUnit 5 documentation.

5.1.2 JavaScript unit testing

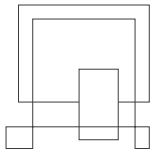
As stated in the Technology section, Jasmine framework was used to implement the unit tests for JavaScript part of the project. The following code snippet in the Figure 5.2 displays the implementation.



```
describe( description: 'TopMenuCtrl', specDefinitions: function() {
    beforeEach(function () {
        window.module('foodAssistant');
    });

    let $controller, $rootScope, statusService;

    beforeEach(inject(function(_$controller_, _$rootScope_, _statusService_){
        $controller = _$controller_;
        $rootScope = _$rootScope_;
        statusService= _statusService_;
    }));

    describe( description: 'should', specDefinitions: function() {
        let $scope, controller;

        beforeEach(function() {
            $scope = $rootScope.$new();
            controller = $controller('TopMenuCtrl', { $scope: $scope, statusService: statusService});
        });
```

Figure 5.2 - Jasmine Unit Test

The test implementation is initiated with the “describe” function which takes in two parameters. First parameter is a string which states what is being tested. The second parameter is a function in which the more specific parts of the test are implemented. The “beforeEach” function is used at the beginning to set the module to be tested. Next, it is used again to inject the tests dependencies. Another “describe” function follows also taking a string and a function as parameters. Now the function contains the narrowed down perimeter of test cases as well as one more “beforeEach” function which initializes the components that are required for testing the functionalities. Among these components we can see “$scope” which is needed to represent the model and “status service” which will be required to test the log in functionality as they are tightly connected. The *Figure 5.3 - Jasmine Test Cases* shows the testcase implementations.



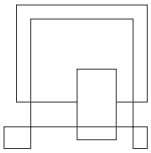
```
it('remove ingredient from the list', function() {
    $scope.ingredientsList = ['water', 'milk', 'flour'];
    $scope.removeIngredient( position: 1);
    expect($scope.ingredientsList).toEqual( util: ['water', 'flour']);
});

it('save ingredients from the list', function() {
    $scope.ingredientsList = ['water', 'milk', 'flour'];
    $scope.saveIngredients();
    expect($scope.ingredients).toContain( util: 'water');
});

it('log out the user', function() {
    $scope.logOut();
    expect(statusService.getLoggedIn()).toBe( util: false);
});
```

Figure 5.3 - Jasmine Test Cases

The function “it” is used to define individual test cases. It takes a string specifying the expected behaviour and a function with the test implementation. First the model was loaded with fake data to simulate the system being used by a user. Then a tested function is executed which is in this case “removeIngredient”. Lastly the “expect” function is called taking the tested function’s result together with “toEqual” function taking the expected result as a parameter. The function validates whether the expected behaviour is met or not. The results of these three tests are next displayed in the command line when running the karma’s run file as can be seen in the *Figure 5.4 - Karma Test Output.*

```
# karma start
13 05 2020 18:12:08.021:WARN [watcher]: Pattern "C:/src/bpr2/app/lib/jasmine-3.5.0/jasmin.js" does not match any file.
13 05 2020 18:12:08.028:WARN [watcher]: Pattern "C:/src/bpr2/app/lib/jasmine-jquery.js" does not match any file.
13 05 2020 18:12:08.072:WARN [karma]: No captured browser, open http://localhost:9876/
13 05 2020 18:12:08.079:INFO [karma-server]: Karma v3.1.1 server started at http://0.0.0.0:9876/
13 05 2020 18:12:08.081:INFO [launcher]: Launching browsers Chrome with concurrency unlimited
13 05 2020 18:12:08.099:INFO [launcher]: Starting browser Chrome
13 05 2020 18:12:11.346:INFO [Chrome 81.0.4044 (Windows 10 0.0.0)]: Connected on socket jCkES_dQqZStT1nfAAAA with id 58480996
Chrome 81.0.4044 (Windows 10 0.0.0): Executed 3 of 3 SUCCESS (0.062 secs / 0.047 secs)
TOTAL: 3 SUCCESS
```

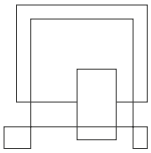
Figure 5.4 - Karma Test Output



5.2 Test Specifications

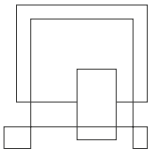
Since the purpose of unit tests is to test small functions, the test for verification whether the requirements were fulfilled or not is needed. Achieving this goal was done doing scenario system test. This test was performed following the test cases in the Figure 5.5 derived from the use cases. Use case descriptions served as a guide step by step for the test execution. This way testing will cover all the functionalities from the functional requirements list.

	Functional requirements	Passed system test
1.	The user can create new account.	Yes.
2.	The user can log in to the application.	Yes.
3.	The user can add new recipe by filling the required information and submitting.	Yes.
4.	The recipe appears in list of recipes immediately without refreshing.	Yes.
5.	When user selects recipe from list more details are displayed.	Yes.
6.	The user can update his existing recipe.	Yes.
7.	The user can remove recipe from his list of recipes.	Yes.
8.	The system doesn't delete shared recipes from other users only from owner.	Yes.
9.	The user can add other users as "friends" by email address.	Yes.
10.	The user can confirm a friend request.	Yes.
11.	The user can reject friend request.	Yes.
12.	If the friend request is confirmed users can mutually see each other in their "Friend list".	Yes.
13.	The users can send a recipe to other users from "Friend list".	Yes.



14.	The user can search in the list of recipes by the recipe name.	Yes.
15.	The user can search in the list of recipes by the ingredient.	Yes.
16.	The user can search in the list of recipes by the category.	Yes.
17.	The user can get recipe suggestion.	Yes.
18.	The user can add items to the shopping list directly from the list of ingredients of a recipe.	Yes.
19.	The user can remove items from the shopping list.	Yes.
20.	The user can receive notifications about friend requests.	Yes.
21.	The user can receive a notification about recipes received from friends.	Yes.
22.	The user can manage his own profile.	Yes.

Figure 5.5 - Test cases

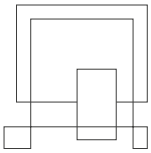


6 Results and Discussion

In conclusion all of the system features were successfully implemented, and the requirements were satisfied. To assure certain qualities, the system test was conducted.

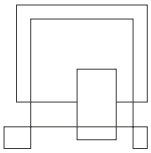
The system provides the users with possibility to manage their recipes. This means they can add new ones, delete them or update them fully replacing old school paper-based technique. Also, user can share them with their friends simply through email address. Moreover, the option to add ingredients directly from the recipe description by only clicking on the ingredient is making everyday life easier. User can also add a picture of the image to their recipe. Image compression was implemented in such a way that it saves the systems resources, but the user does not suffer from any noticeable image quality loss. Furthermore, notification service was implemented so the users get “push” notifications whenever they receive a new friend request, or another user shares a recipe with them. Another implemented functionality is that user can get a recipe suggestion. This suggestion can be influenced by specified parameters if there are some undesired recipes. This feature also takes into consideration recently cooked recipes, so it does not show the same results all the time.

The result of this project is a fully functional application that can be valuable for many users. It can be said that after some minor code fixes, the application would be fully ready for deployment and future development.



7 Conclusions

The product created is a complete recipe management system. All the features arising from the customer's requirements were fulfilled. To assure availability on different platforms, the system was designed as a single page web application. Rest architecture was picket over the web sockets as the persistent connection is not required. Most of the system functionalities are contained within the client which was created using AngularJS framework. This ensures that certain architecture pattern is followed and opens the door for easy continuation of the development in the future.
The capabilities of the whole system were tested following the use case descriptions to guarantee correct behavior during the system test. Couple of unit tests were implemented as well to catch code errors in the early stages.
The system replaces the use of physical cookbooks and shopping lists as well as it eliminates the need for social networks to share recipes. It offers options to store and manage recipes, search through them by specifying filters, share them with friends, get suggestion for recipes and construct shopping lists directly from the recipe descriptions and access them from any device with supported browser, independently from the screen resolution.



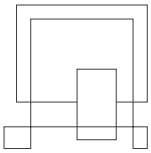
8 Project future

Even though all the functionalities were successfully implemented, there is still space for improvements in the future. Missing unit tests for all parts of the system need to be created to assure easier progress in the future. Also, some minor bugs would need to be fixed. The product can be considered as a prototype that has all the basic functionalities for the recipe management. Due to the lack of the time the system is not special in the terms of functionalities it offers. This means that all the “unique” functionalities that would differentiate it from other systems on the market would be yet to be implemented.

Another change would be redesigning the system. The list of recipes could be displayed in more hierarchical way e.g. nested categories to provide better overview. Also, user login could support login through other system like Facebook, Gmail etc. Moreover, other common functionalities shall be added e.g. possibility to add friend is only for recipe sharing, in the future this functionality could be expanded so the users can communicate with each other as well. Since currently the friends section of the application only displays the list of the friend, another extension could be displaying all the friends recipes and possibility to search through them.

As described in the implementation, the notification system is implemented as polling. However, this is not the most efficient way for handling push notifications. Long polling or Comet approach could replace it in the future.

Lastly the system requires some security upgrades. The passwords are currently stored as a characters in the database. Some kind of password encryption is necessary to protect the user’s privacy. Further the password recovery function is another common standard that should be implemented. Second consideration is securing the database from unwanted data manipulation.



9 Sources of information

Anon 2020. *AngularJS: Developer Guide: Unit Testing*. [online] Available at: <https://docs.angularjs.org/guide/unit-testing> [Accessed 15 May 2020].

Anon 2020. *Sass: Sass Basics*. [online] Available at: <https://sass-lang.com/guide> [Accessed 20 March 2020].

Anon 2020. *Responsive Web Design Media Queries*. [online] Available at: <https://www.w3schools.com/css/css_rwd_mediaqueries.asp> [Accessed 20 March 2020].

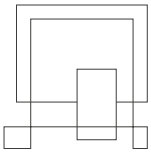
Gaddis, T., 2015. *Starting out with Java: early objects*. Fifth5. ed. Upper Saddle River, NJ: Pearson.

Anon 2020. *Getting Started | Building a RESTful Web Service*. [online] Available at: <https://spring.io/guides/gs/rest-service/> [Accessed 1 February. 2020].

Anon 2020. *@Before vs @BeforeClass vs @BeforeEach vs @BeforeAll | Baeldung*. [online] Available at: <https://www.baeldung.com/junit-before-beforeclass-beforeeach-beforeall> [Accessed 15 May. 2020].

Anon 2020. *Java DIP - Image Compression Technique - Tutorialspoint*. [online] Available at: <https://www.tutorialspoint.com/java_dip/image_compression_technique.htm> [Accessed 3 April. 2020].

Anon 2020. *Serving Static Web Content with Spring Boot*. [online] Available at: <https://spring.io/blog/2013/12/19/serving-static-web-content-with-spring-boot> [Accessed 1 Jun. 2020].



10 Appendices

Appendix J Project Description

Project description

Food and recipe assistant

Patrik Kucerka 260026
Patrik Ihnat 260010

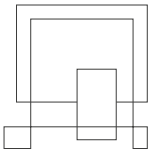
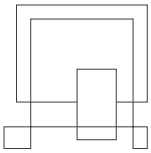
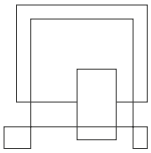


Table of content

Contents





Background description

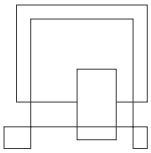
Food is something that people need for their survival. From the food we get energy and nutrients needed for the body maintenance, fighting viruses and bacteria, for the body regeneration and stimulation of the growth. The amount of food people need differs from person to person depending on the sex, weight, height, age and physical activity of the individual. Energy gained from the food is measured in calories and the average for an adult is 2,700. This knowledge allows as to plan the proper diet that makes us vital and full of energy as well as ensures that body gets all stances needed. All the nutrients can be put into five groups. Those are carbohydrates, fats, proteins, vitamins and minerals.

Carbohydrate are the primary source of energy. The significant amount comes from the plants especially starchy ones like beans, potatoes and gains. However, people should be careful about the intake of sugars since the high amounts can lead to obesity and other health problems.

Fats are the second main source of energy. Even though many people avoid fats thinking they are dangerous for the weight maintenance, they are as important as sugars. The presence of same fats is even needed for the body to absorb some vitamins.

Proteins are the building blocks of the human body and thus are very important. The biggest resource of proteins are animal products and beans.

The times when food was just food, just a basic need is gone. As people evolved, their interest in food processing grew and became one of the most popular interests because everybody has to eat. People started to write down the recipes and started to share them. That is how cookbooks came into this world. As the modern technology progressed this part of human culture did not remain unimpacted. A lot of new technologies and applications were introduced to ease the everyday struggle with keeping and managing recipes. Original ways like paper cookbooks, newspaper cuttings, sticky notes on the fridge etc. were replaced with mobile applications and online forums where people can discuss and share their ideas with others. On the one



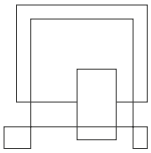
hand it turned all the original issues into very simple tasks but on the other hand a new problem arose.

We have different applications likes recipe management systems, shopping lists, applications for keeping track of the expiry date of the groceries, forums for sharing recipes etc. This can be fine if people need just one or two applications but for the people with bigger interest in culinary arts or for people with special needs or diets a problem arises. People have to jump from one application to another and sometimes they even need to transfer data by themselves since these applications are not interconnected.

Moreover nowadays, in the world of internet, people have so many options that it might get difficult to make right decisions regarding their diet and they hard times picking the right recipe.

(food | Definition & Nutrition | Britannica.com, 2019)

(food | National Geographic Society, 2019)



Purpose

The goal of this project is to reduce the overwhelming number of different applications needed to manage, share recipes, create shopping lists and maintain diets by replacing them with a single solution so the user does not need to jump from one application to another.

Problem statement

Currently people need to install different applications with different purposes regarding the recipe management which takes resources and is time consuming. The challenge in this project is to provide a solution for the overwhelming amount of applications for the people with special needs or diets or for people that require advanced recipe management system.

- How can the user store the recipes and the user information?
- How can the user share the recipes with other users?
- How can the user access the application on different platforms?
- How can the user get recipe suggestions?
- How can the user keep their recipes sorted in some way?
- How can the user create and store a shopping list?

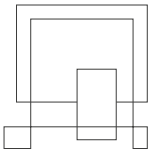
Delimitation

The listed functionalities will be out of our scope due to the limited time and resources for this project.

1. Functionality for keeping track of the expiry date of the groceries.
2. Functionality replacing forums for sharing recipes and discussions.
3. System for data exchange between different applications.

Methodology

To achieve a high agility and good overview the SCRUM was chosen for the development process management. Moreover, it will improve the project planning as well as give us an option to track and monitor the progress. This means that daily



scrum meetings will be involved in the process and will take 10-15 minutes every morning to discuss work done from the previous day and to plan the work to be done that day, so everybody has an overview of what is currently going on. It will also provide a space for questions and discussion of problems. We also decided for UP to ensure high agility for the development process.

Project phases

Inception

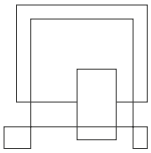
In this phase we would like to achieve that all the stakeholders will understand the lifecycle and all objectives of this project. We want to focus on key things such as understanding what we want to build in which order. Identify and understand all functionalities of our system. If needed also do estimation on costs for project.

Elaboration

This phase of the project is going to be dedicated to gather and identify more detailed and in-depth system/project requirements, because having a detailed understanding of requirements and dependencies makes us stricter and more accurate on plan. Part of this phase is also going to be dedicated to design and validate the foundation of the system architecture.

Construction

For this phase we would like to start coding foundations for the system. Which also should provide is with more accurate estimation on time and costs. In this phase the strict timeline for sprints will come to hand since to be able to follow all our visions we must precisely follow our planned events. Also, all the diagrams and visualizations of our system will take place in this phase.



Transition

At this stage of project, we want to have all our project specifications and diagrams done and up to date according to system. The whole code will be documented etc. The team who was in development phase is also going to be responsible for transition.

Scrum roles

Scrum master

Patrik Ihnat

Product owner

Patrik Kucerka

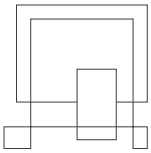
Development team

Patrik Kucerka, Patrik Ihnat

Sprint backlog

This backlog is going to be created at very beginning of each sprint and afterwards may be modified during or after sprint to adjust it according to outcomes.

Question	Objective	Model
How can we store the recipes and user information?	It is important to choose a suitable system for storing the information about the user as well as recipes.	JSON objects stored in the database.
How can the user share recipes with other users?	We need to satisfy the need to share recipes with other users.	Through the user email.
How can we make the application available on different platforms?	It is necessary that the users can access their recipes from any computer.	The solution will be a web application.



How can the user keep their recipes sorted in some way?	It is essential that the user can sort their recipes and organize them.	Option to apply filters to a list of recipes.
How can the application suggest recipes?	We have to find an algorithm that will try to predict and generate a suggestion for the users.	The application will collect data about recently cooked recipes and take into consideration user's requirements.
How can the user create and store a shopping list?	Possibility of creating a shopping list is of great importance.	Application UI will allow the user add items to shopping list directly from recipes. These items will be stored in the database as JSON objects

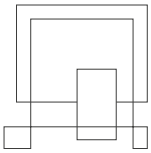
Time schedule

The time this project will take is estimated to be 554 hours per person. The following time schedule displays the ideal working time over the.

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
8 –16	8-16	8-16	0	8-16	0	0

Estimated durations of the phases of UP:

1.2 - 15.2 - Inception
16.2 – 7.3 - Elaboration
8.3 – 31.4 - Construction
1.5 – 30.5 – Transition



Schedule of SCRUM sprints:

Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7	Sprint 8
1.2 - 15.2	16.2 - 29.2	1.3 - 15.3	16.3 - 30.3	1.4 - 15.4	16.4 - 31.4	1.5 - 15.5	16.5 - 4.6

Miles stones/sprint durations:

1.sprint: 70 hours

2 sprint: 64 hours

3 sprint: 70 hours

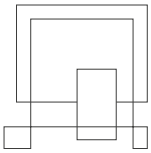
4 sprint: 70 hours

5 sprint: 70 hours

6 sprint: 70 hours

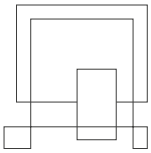
7 sprint: 70 hours

8 sprint: 70 hours



Risk assessment

Risk	Description	Likelihood Scale: 1-5	Severity Scale: 1-5	Risk mitigation	Identifiers
Miscommunication	Communication failure	2	4	Morning meetings	Task are not being achieved
Unpredictable conditions	Unexpected work shifts	3	5	Back-up plan	Task are wrongly implemented
Lack of time	Extra classes	3	4	Precise planning	Task implementation is delayed
Redesigning of system	System has to be redesigned due to the lack of knowledge	2	5	Careful design	Design cannot be implemented



References

Anon 2019. *food | Definition & Nutrition | Britannica.com*. [online] Available at: <https://www.britannica.com/topic/food> [Accessed 28 Oct. 2019].
Anon 2019. *food | National Geographic Society*. [online] Available at: <https://www.nationalgeographic.org/encyclopedia/food/> [Accessed 28 Oct. 2019].