Take-In

Project Report R&D Team-31

Group Members:

Jens Broeren,	s1118059
Aditya Desai,	s1114348
Tijn Giesberts,	s1121678
ÖZGÜN Mutlu,	s1115680

Table of Contents

1. Introduction

Take-In is an application designed to promote sustainable food consumption while building a community of users who have a shared interest in taking on behaviour that supports this cause. At its core, it is a platform to bring people together by letting them share their recipes and leftovers, but Take-In also incorporates communication features like rating other users and maps to find sustainable ingredients, to name a few.

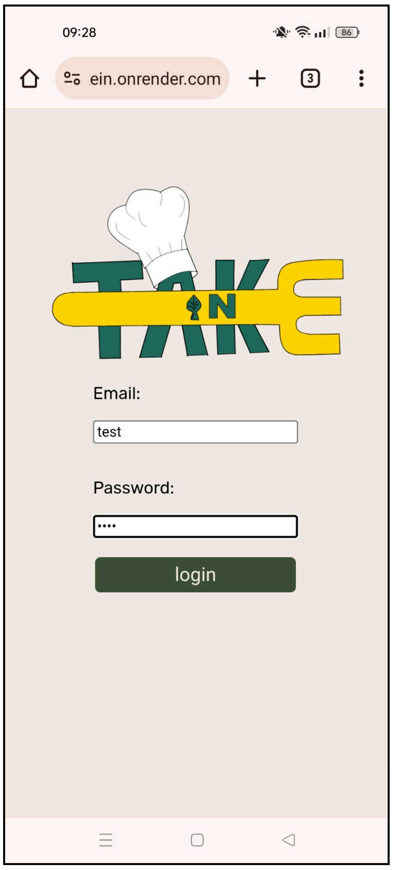
Our users can be categorised into four broad groups, all of whom have different reasons to be interested in using Take-In:

- **Cooking users**: Our favoured users, towards which most of our application is geared. They are the ones that use Take-In to find new recipes and will sometimes share their own recipes. These users will probably make up most of our user base.
- **Administrators**: The people who will have to check submitted recipes and approve them in order to let other users see them. We aren't sure if we will implement administrator operations in our final submission.
- **Less privileged users**: These users can use our app to find sustainable ingredients with as low a price as possible, potentially letting them save some money.
- **Privileged users**: These users occasionally use Take-In to find new recipes or to just try something new, and their main goal in using the app would be to promote sustainable practices.

2. Description

2.1 Focus on Properties

Our application has several "pages", from which its different features can be accessed. As we designed it to be primarily used with a phone, the screenshots that accompany the explanation of Take-In's properties have been taken with one. With this in mind, we made it look like a mobile screen on your browser. The login page is fairly self-explanatory:

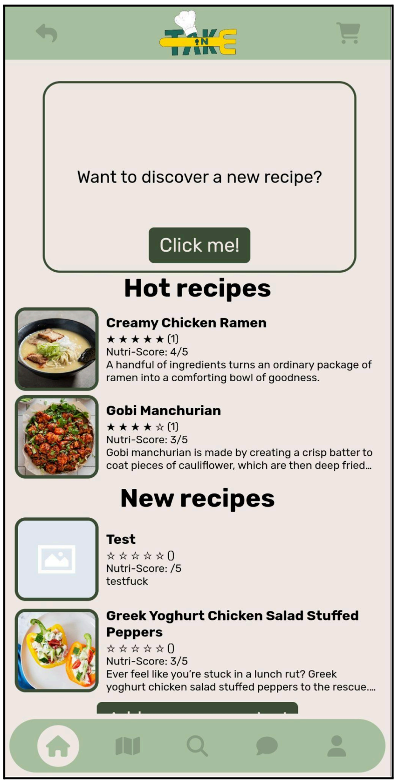


Email: test

Password: ····

There are two things we would like to draw attention to. First, as our application is designed to be a web application, it is accessed by entering the corresponding link in a browser. You may notice that the link is a Render link[2]; this is because, at the time of writing this section, our application was still in its beta testing phase. The other thing we would like to very briefly mention is the text boxes, which simply open up the keyboard to enter a combination of username and password when clicked on.

Next, we have the first page our user sees; the main menu. Since this section is meant to be a global overview, we will not go into too much detail here. There are a couple of things to be noticed. First, the top and bottom bars for every page on Take-In are kept consistent. The top bar has, of course, our logo, a back button to revisit the previous page the user was on and a shortcut to access their shopping cart. In the bottom bar, we have five buttons. From left to right, these are the home button, finding restaurants, searching for recipes, chats and user profile.



Want to discover a new recipe?

Hot recipes

Creamy Chicken Ramen
★★★★★ (1)
Nutri-Score: 4/5
A handful of ingredients turns an ordinary package of ramen into a comforting bowl of goodness.

Gobi Manchurian
★★★★☆ (1)
Nutri-Score: 3/5
Gobi manchurian is made by creating a crisp batter to coat pieces of cauliflower, which are then deep fried...

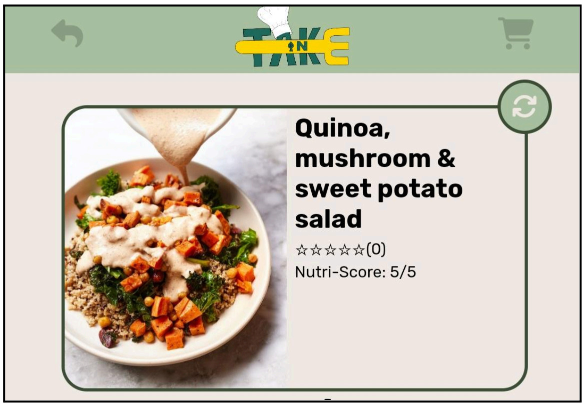
New recipes

Test
☆☆☆☆☆ ()
Nutri-Score: /5
testfuck

Greek Yoghurt Chicken Salad Stuffed Peppers
☆☆☆☆☆ ()
Nutri-Score: 3/5
Ever feel like you're stuck in a lunch rut? Greek yoghurt chicken salad stuffed peppers to the rescue....

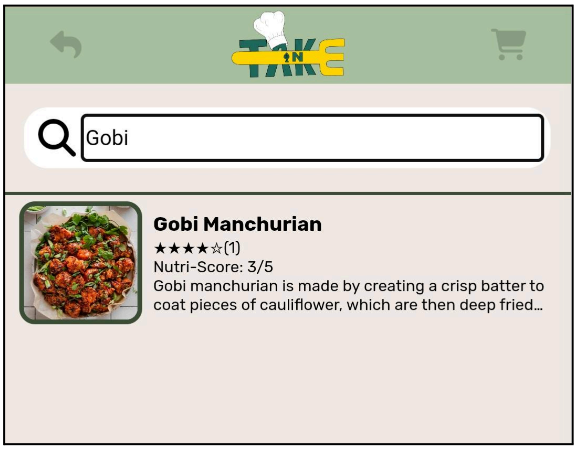
According to an article about good UI design[3], segmenting the information presented on a page helps users navigate it with much less confusion. Since our main menu is fairly important, we applied this principle here as well by using either borders, title text (bold and larger to serve as a clear indication of a new section) and dark buttons with clear descriptions of their functions provided on them.

If the user were to click on the button at the top of the main menu, shown in the previous screenshot, they would be provided a random recipe from our database. Initially, the plan for this feature was to generate a recipe for the day, but we decided to make it a random recipe generator to facilitate exploration of our recipes database.



Quinoa, mushroom & sweet potato salad
☆☆☆☆☆(0)
Nutri-Score: 5/5

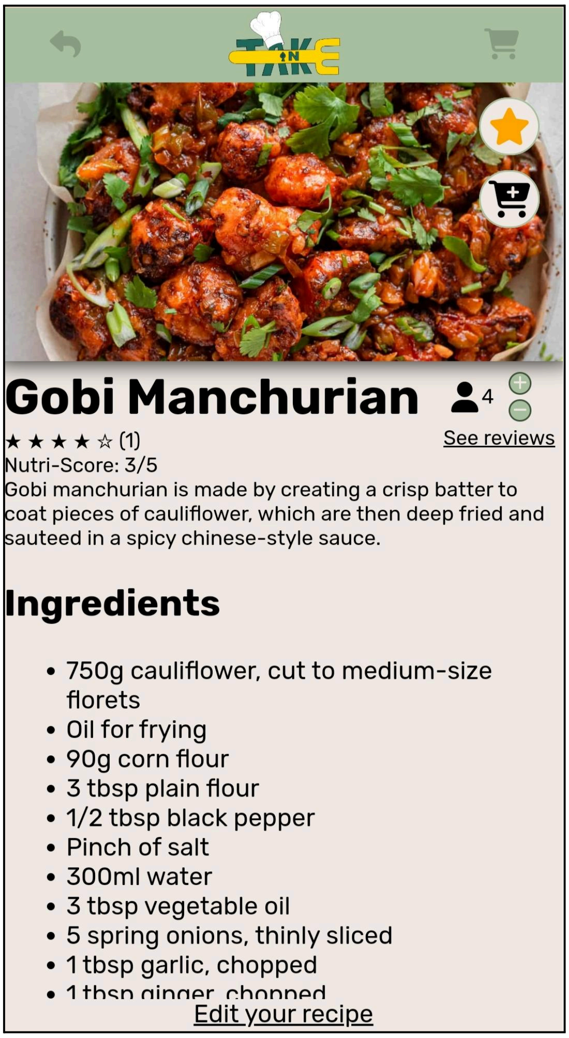
Shown underneath the picture of the random recipe provider is what the user would see in the search menu. Here, by entering the name of a recipe or a prompt into the search bar, they can be provided with a list of recipes that they could try. We provide the most elementary information in the results of these queries; namely, the recipe name, rating, nutri-score, picture and brief description. This is the most important information for our users, considering the goals of our application.



Gobi

Gobi Manchurian
★★★★☆(1)
Nutri-Score: 3/5
Gobi manchurian is made by creating a crisp batter to coat pieces of cauliflower, which are then deep fried...

By clicking on this recipe, the user would be taken to its viewing page. Here, quite a bit of information is put at their disposal, and we have (again) attempted to organise it in a comprehensible manner. We first provide the elementary details that the user would have seen in the search menu; this is to give them the important numbers about a recipe if they happened to access it from anywhere else. Underneath it follows the list of ingredients and then the recipe itself, for ease of reading; the user would just have to scroll through the page as they cook. The rest of the recipe viewing page is shown below. Near the name of the recipe, there is an option to select for how many people you are cooking. By doing so, you can click on the cart button to add the recipe and the amount of people to cook for to the shopping cart.



Gobi Manchurian 4

★★★★☆ (1) See reviews
Nutri-Score: 3/5
Gobi manchurian is made by creating a crisp batter to coat pieces of cauliflower, which are then deep fried and sauteed in a spicy chinese-style sauce.

Ingredients

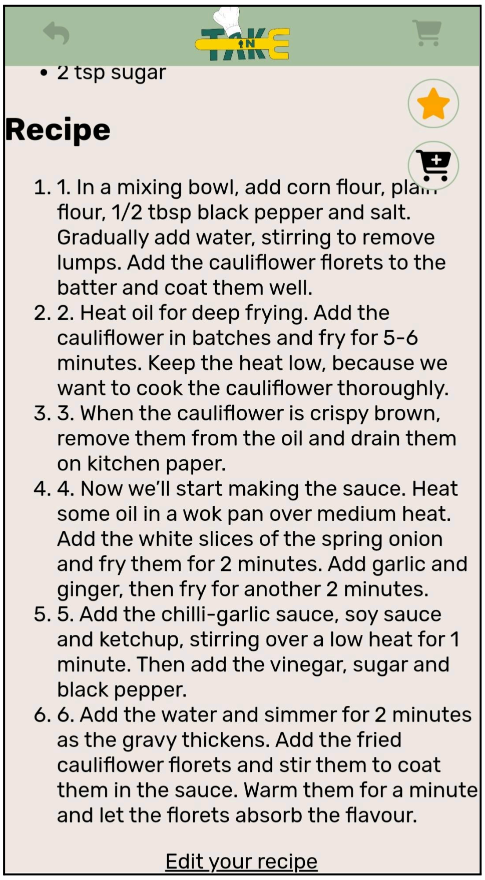
- 750g cauliflower, cut to medium-size florets
- Oil for frying
- 90g corn flour
- 3 tbsp plain flour
- 1/2 tbsp black pepper
- Pinch of salt
- 300ml water
- 3 tbsp vegetable oil
- 5 spring onions, thinly sliced
- 1 tbsp garlic, chopped
- 1 tbsp ginger, chopped

Edit your recipe

The last important feature here is the favourite button, the gold star in the top right; this creates a shortcut to the recipe from your profile page.

Shown alongside is a screenshot of the shopping cart after adding a recipe to it. We provide delete buttons in all lists (friends, shopping cart, favourites) for ease of removing elements of the list. This makes Take-In more convenient to use.

From here, we can move further down the bottom bar. Next, we have our application's social features, which are accessed from the chat menu. Here, users can see conversations they have had with other users, sorted so that the conversations with the most recent messages are placed at the top of the screen. By clicking on a conversation, users can view the chat history and send new messages.

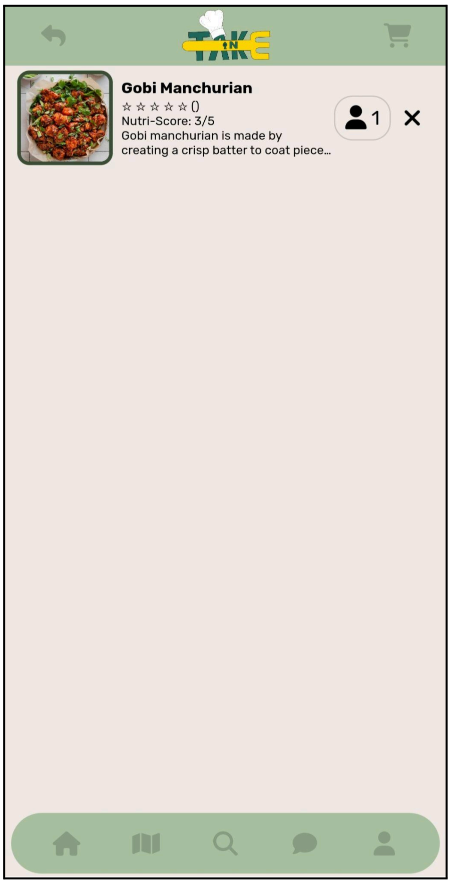


- 2 tsp sugar

Recipe

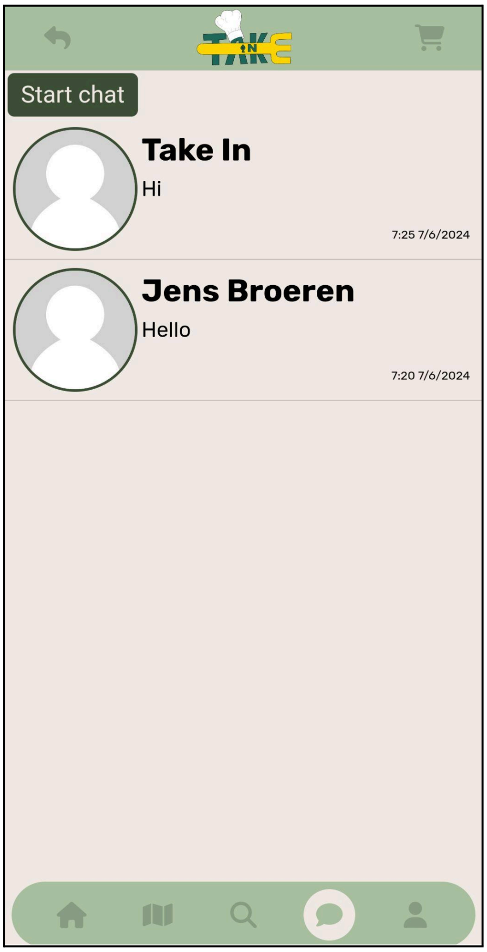
1. 1. In a mixing bowl, add corn flour, plain flour, 1/2 tbsp black pepper and salt. Gradually add water, stirring to remove lumps. Add the cauliflower florets to the batter and coat them well.
2. 2. Heat oil for deep frying. Add the cauliflower in batches and fry for 5-6 minutes. Keep the heat low, because we want to cook the cauliflower thoroughly.
3. 3. When the cauliflower is crispy brown, remove them from the oil and drain them on kitchen paper.
4. 4. Now we'll start making the sauce. Heat some oil in a wok pan over medium heat. Add the white slices of the spring onion and fry them for 2 minutes. Add garlic and ginger, then fry for another 2 minutes.
5. 5. Add the chilli-garlic sauce, soy sauce and ketchup, stirring over a low heat for 1 minute. Then add the vinegar, sugar and black pepper.
6. 6. Add the water and simmer for 2 minutes as the gravy thickens. Add the fried cauliflower florets and stir them to coat them in the sauce. Warm them for a minute and let the florets absorb the flavour.

Edit your recipe



Gobi Manchurian
☆☆☆☆☆ ()
Nutri-Score: 3/5
Gobi manchurian is made by creating a crisp batter to coat piece...

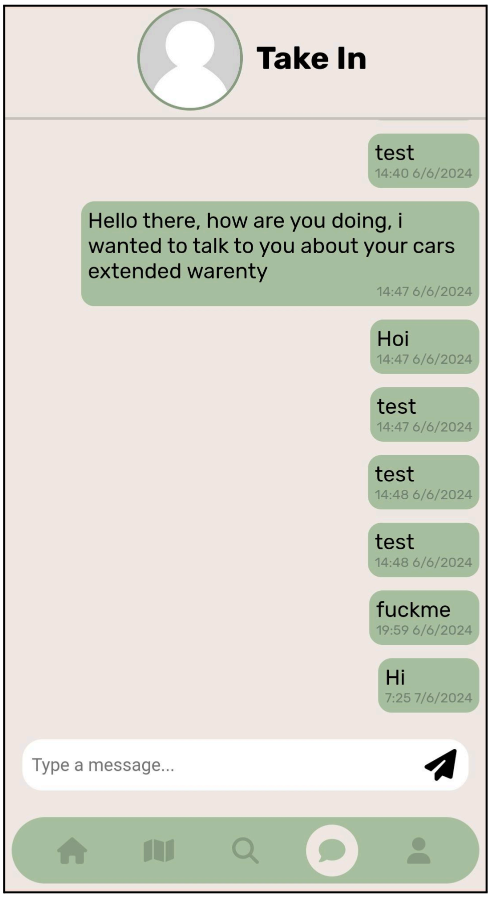
1 ×



Start chat

Take In
Hi
7:25 7/6/2024

Jens Broeren
Hello
7:20 7/6/2024



Take In

test
14:40 6/6/2024

Hello there, how are you doing, i wanted to talk to you about your cars extended warenty
14:47 6/6/2024

Hoi
14:47 6/6/2024

test
14:47 6/6/2024

test
14:48 6/6/2024

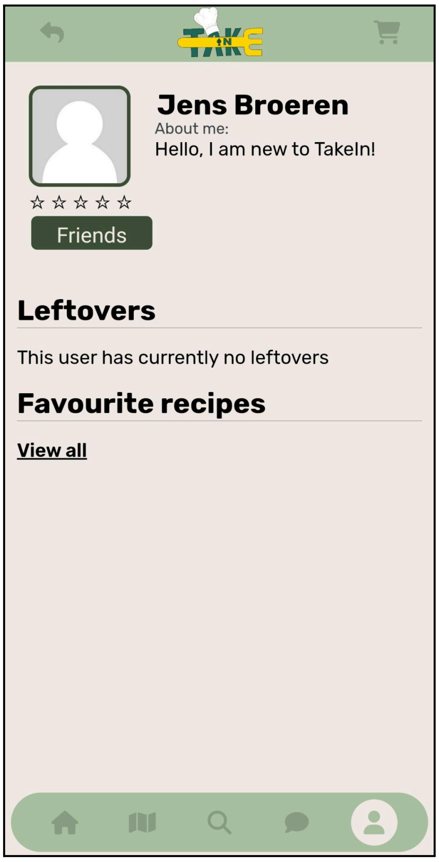
test
14:48 6/6/2024

fuckme
19:59 6/6/2024

Hi
7:25 7/6/2024

Type a message...

By either clicking on the profile picture of a user from the chat menu, or by clicking on the Friends button in your profile and then selecting a user from the list that appears, it is possible to view another user's profile. In their profile, you can view their personal description, friends that the two of you share, their rating (which is provided by other users), their leftovers and their favourite recipes.



Jens Broeren
About me:
Hello, I am new to TakeIn!
☆ ☆ ☆ ☆ ☆
Friends

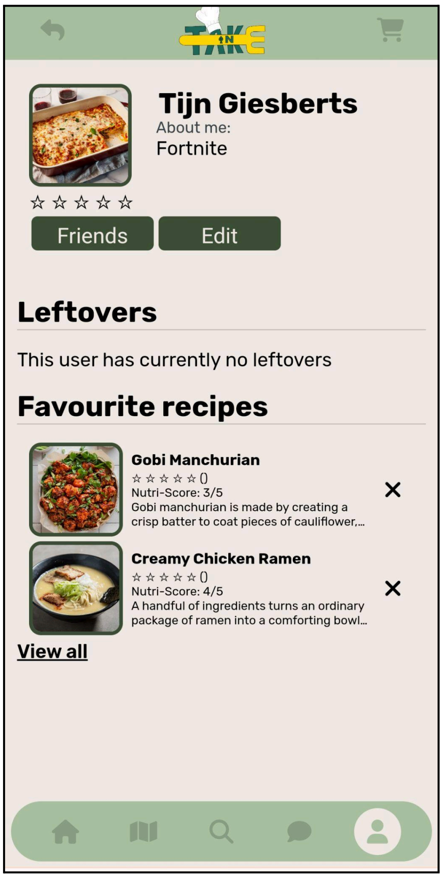
Leftovers

This user has currently no leftovers

Favourite recipes

View all

To explain the rating system, we would have to begin with explaining the leftovers system. On a user's profile, they can add leftovers for other users to view on the map page. After this, other users can see that you have leftovers and can send you a message to request them. After two users meet up in person and the hand-off of leftovers is completed, and the leftovers are removed from your user profile, you will receive a prompt to rate the user with whom you exchanged them. By doing this, our users can indicate who is a more agreeable part of the community and who is not, thus allowing us to build a network centred around helping out other users.



Tijn Giesberts
About me:
Fortnite
☆ ☆ ☆ ☆ ☆
Friends Edit

Leftovers

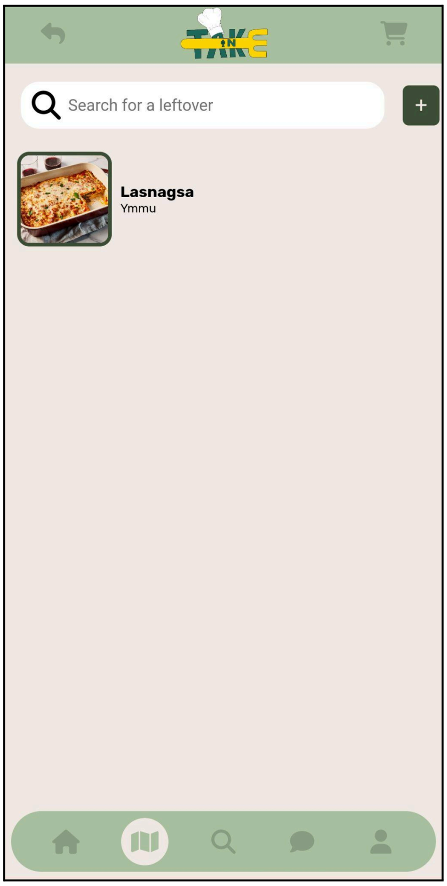
This user has currently no leftovers

Favourite recipes

Gobi Manchurian
☆ ☆ ☆ ☆ ☆ ()
Nutri-Score: 3/5
Gobi manchurian is made by creating a crisp batter to coat pieces of cauliflower,...
✕

Creamy Chicken Ramen
☆ ☆ ☆ ☆ ☆ ()
Nutri-Score: 4/5
A handful of ingredients turns an ordinary package of ramen into a comforting bowl...
✕

View all

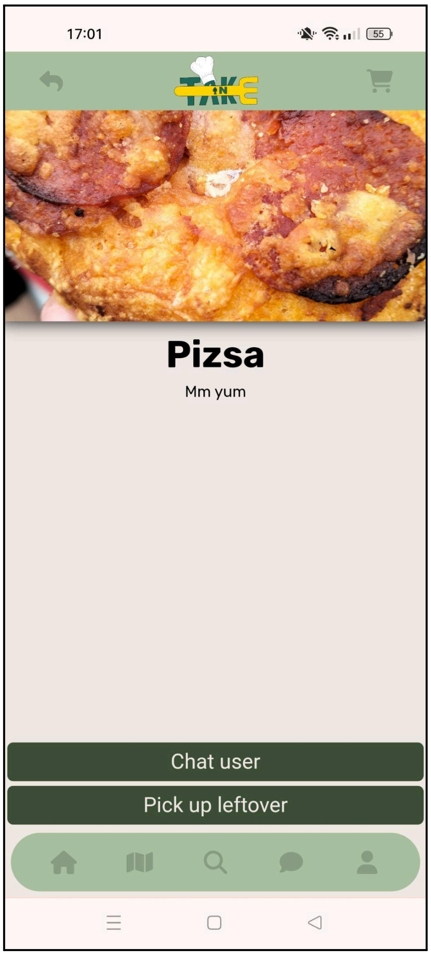


Search for a leftover
+

Lasnagsa
Ymmu

Next, we have, of course, your user profile. From here, users can see their own rating, their friends list, edit their profile (namely the About Me section) and leftovers, as well as view the recipes that they have marked as favourites. All of these features are fairly self-explanatory, considering prior explanations, so we will not be elaborating on them further here. By clicking on the friends button, you can see a menu where all of your friends are listed; clicking on their name takes you to their profile, where you have the option to unfollow them if you so desire by pressing a button.

Finally, we have the finding leftovers page, which is shown in the rightmost screenshot above. Here, users can see all the leftovers available on our platform (this is elaborated upon further in the report). Additionally, by clicking on the plus sign next to the search bar, users can add their own leftovers by selecting an image and writing a small description. Clicking on a leftover opens the page to the right. Here, you can either choose to message the user who put the leftover up for viewing or pick it up, effectively reserving it for you.



2.2 Product Justification

Take-In is not the only application that aims to promote a more sustainable lifestyle by expanding its users' culinary options. Provided below is a list of some of our platform's contenders, as well as some of their key features / selling points[1]:

- **Happy Cow:** An application that allows users to find vegan / vegetarian restaurants near them, with a decently-sized recipe section. It is very convenient to use.
- **How's Good:** An application that rates products at grocery stores based on 60 indicators of sustainability. Though its uses are limited, it excels in its specialty.
- **Good Guide:** An application that rates food based on how healthy it is, on a scale of 0 to 10. Ingredients, environmental impact and the company's attention to labour rights are factored into this rating. Like *How's Good*, it has limited usage but is a strong contender in its field.

It is worthwhile to build our system for a few reasons. First of all, while there are many other applications that allow users to discover new recipes and rate them, Take-In is unique in that it also incorporates social features more strongly than its counterparts. By using a messaging feature similar to WhatsApp, we facilitate communication between users. Our leftovers sharing system gives users a reason to connect while also reducing food waste. By letting users rating other users and recipes, we create a community that encourages positive user contribution and lessens the impact of users with a negative influence on our initiative.

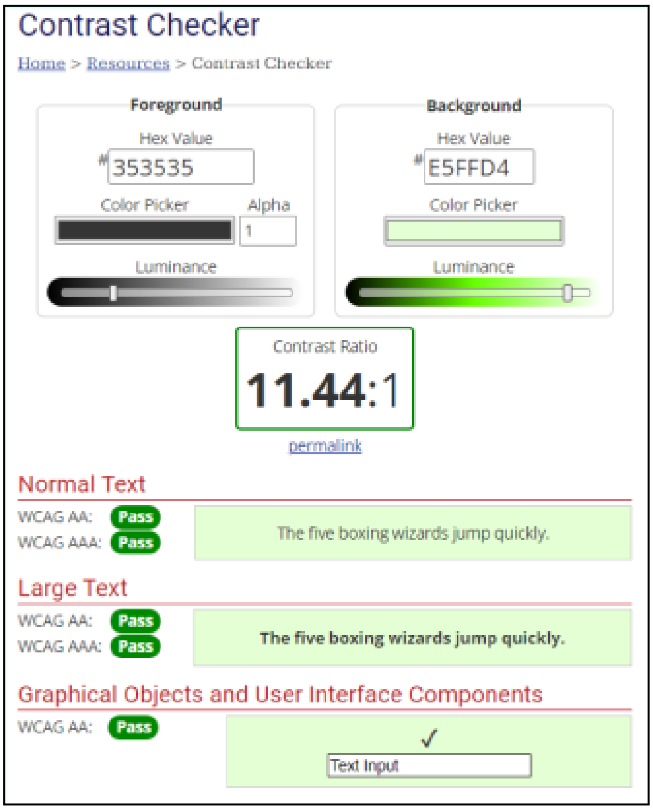
Many applications achieve one or two very specific goals, but they lack feature diversity. Take-In's biggest selling point is how it fulfils many purposes, thus greatly increasing the convenience it provides its users with by discouraging them from installing, say, four different sustainability apps and instead using just one.

2.3 Specifications

In this section of the report, we shall not only consider how our project deviated from our original plans but also some accessibility considerations that we discussed prior to beginning Take-In's development.

First, the accessibility considerations. To begin with, we used a colour contrast-checking tool[4] to see if our text would be easily viewable on our background. These tests were conducted using the Figma prototype of Take-In; this is why the pages shown in this section of the specifications may not be the same as in the screenshots provided in section 2.1.

Another important aspect of the properties of Take-In is the colour scheme; we tried to make buttons highly distinguishable from the background, which is kept plain and light on the eyes. Black text increases legibility and accounts for different types of colour-blindness, as we observed that it is the colour least affected by the variations of the condition. Shown below are the tests we conducted, using a colour blindness simulator:



Contrast Checker

Home > Resources > Contrast Checker

Foreground

Hex Value #353535

Alpha 1

Background

Hex Value #E5FFD4

Contrast Ratio

11.44:1

Normal Text

WCAG AA: Pass

WCAG AAA: Pass

The five boxing wizards jump quickly.

Large Text

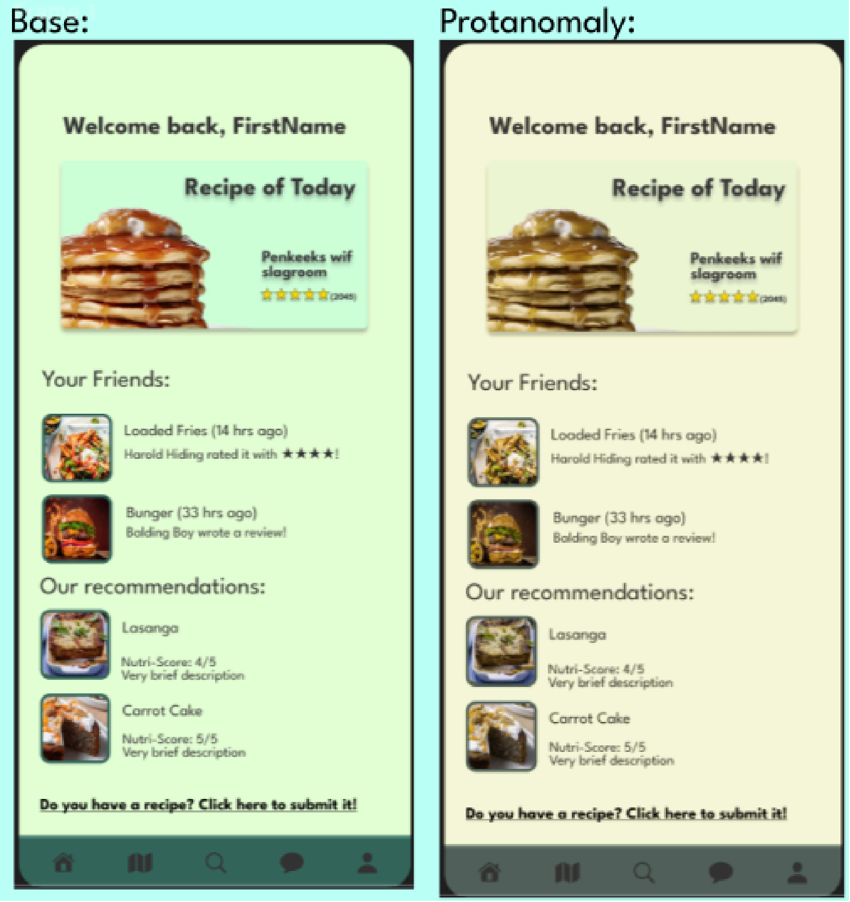
WCAG AA: Pass

WCAG AAA: Pass

The five boxing wizards jump quickly.

Graphical Objects and User Interface Components

WCAG AA: Pass



Base:

Welcome back, FirstName

Recipe of Today

Penkeeks wif slagroom

★★★★★(2045)

Your Friends:

Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!

Bunger (33 hrs ago)
Balding Boy wrote a review!

Our recommendations:

Lasanga
Nutri-Score: 4/5
Very brief description

Carrot Cake
Nutri-Score: 5/5
Very brief description

Do you have a recipe? Click here to submit it!

Protanomaly:

Welcome back, FirstName

Recipe of Today

Penkeeks wif slagroom

★★★★★(2045)

Your Friends:

Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!

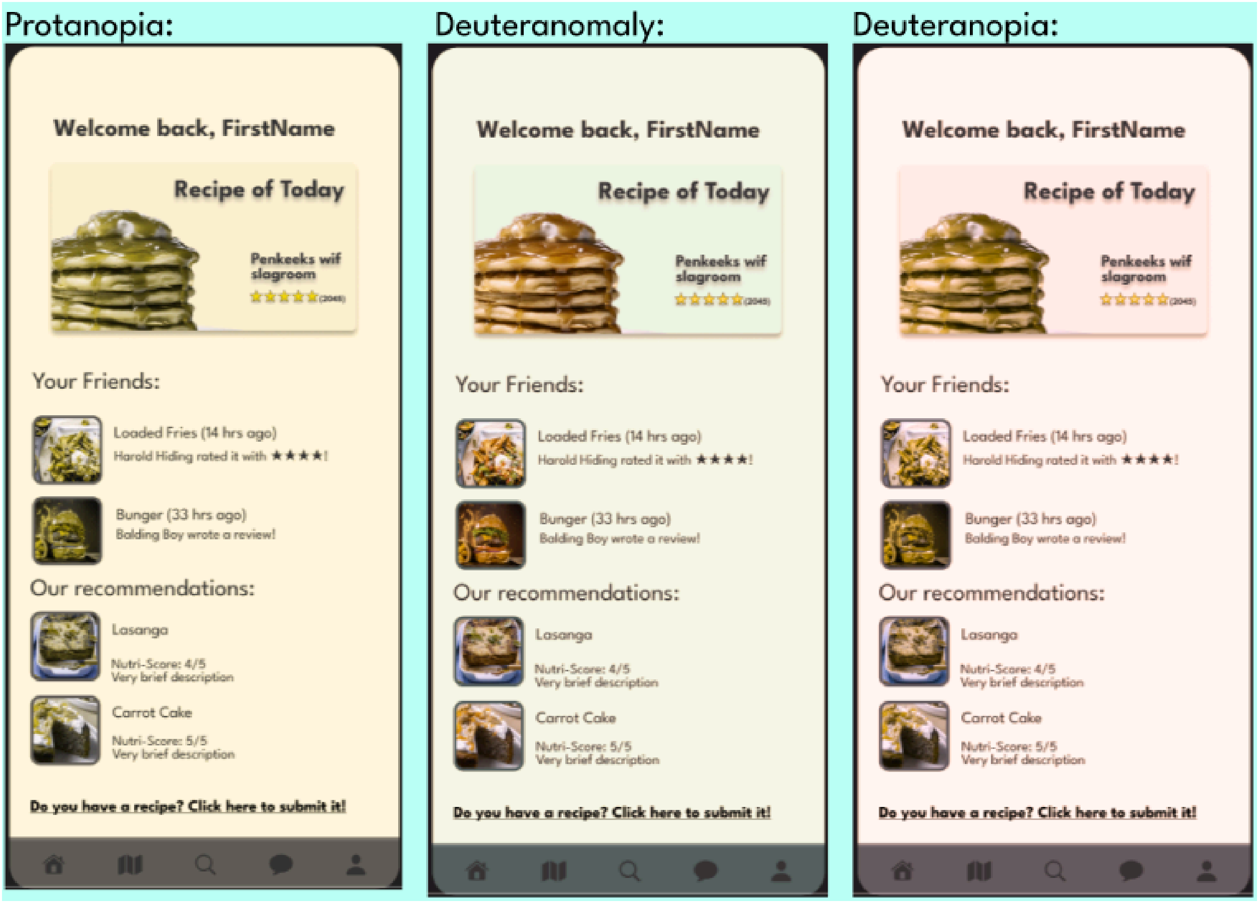
Bunger (33 hrs ago)
Balding Boy wrote a review!

Our recommendations:

Lasanga
Nutri-Score: 4/5
Very brief description

Carrot Cake
Nutri-Score: 5/5
Very brief description

Do you have a recipe? Click here to submit it!



Welcome back, FirstName

Recipe of Today

Penkeeks wif slagroom

★★★★★(2045)

Your Friends:

Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!

Bunger (33 hrs ago)
Balding Boy wrote a review!

Our recommendations:

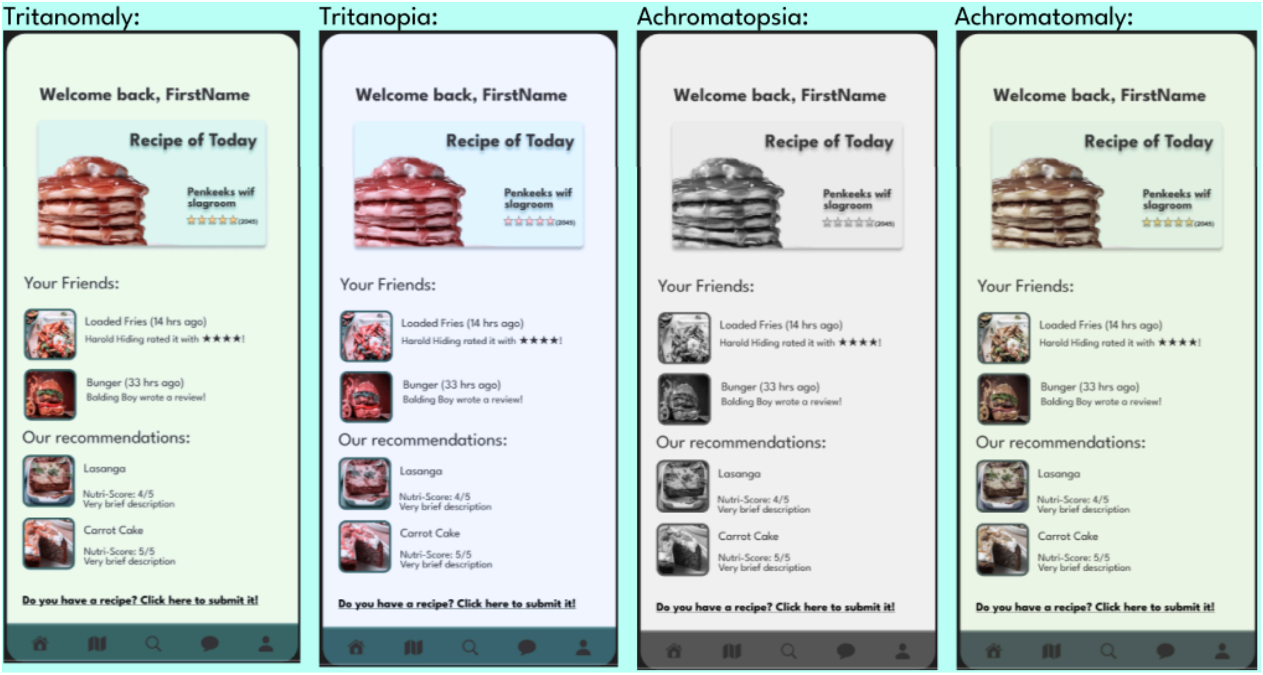
Lasanga
Nutri-Score: 4/5
Very brief description

Carrot Cake
Nutri-Score: 5/5
Very brief description

Do you have a recipe? Click here to submit it!

Welcome back, FirstName
Recipe of Today
Penkeeks wif slagroom
★★★★★(2045)
Your Friends:
Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!
Bunger (33 hrs ago)
Balding Boy wrote a review!
Our recommendations:
Lasanga
Nutri-Score: 4/5
Very brief description
Carrot Cake
Nutri-Score: 5/5
Very brief description
Do you have a recipe? Click here to submit it!

Welcome back, FirstName
Recipe of Today
Penkeeks wif slagroom
★★★★★(2045)
Your Friends:
Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!
Bunger (33 hrs ago)
Balding Boy wrote a review!
Our recommendations:
Lasanga
Nutri-Score: 4/5
Very brief description
Carrot Cake
Nutri-Score: 5/5
Very brief description
Do you have a recipe? Click here to submit it!



Welcome back, FirstName
Recipe of Today
Penkeeks wif slagroom
★★★★★(2045)
Your Friends:
Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!
Bunger (33 hrs ago)
Balding Boy wrote a review!
Our recommendations:
Lasanga
Nutri-Score: 4/5
Very brief description
Carrot Cake
Nutri-Score: 5/5
Very brief description
Do you have a recipe? Click here to submit it!

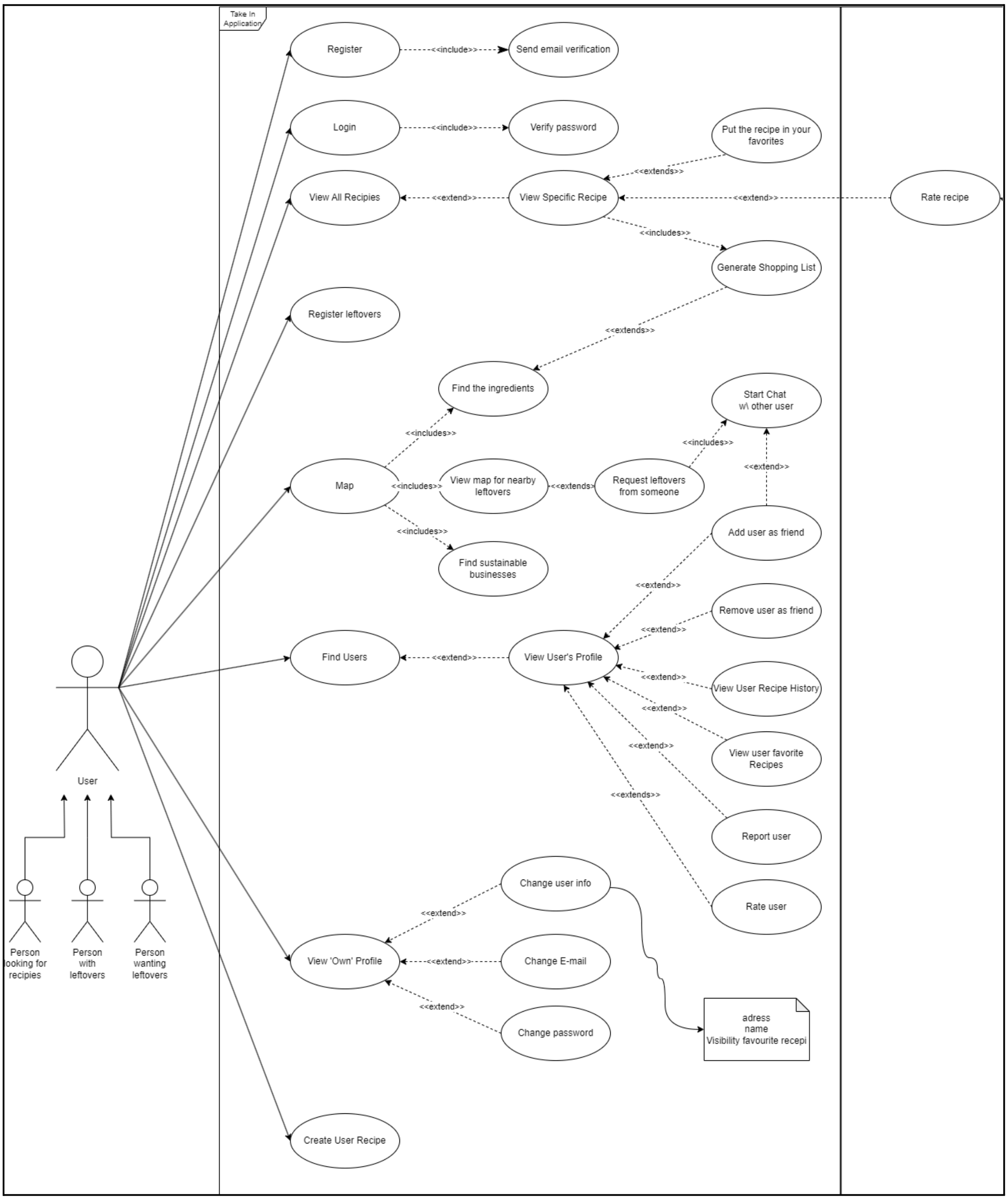
Welcome back, FirstName
Recipe of Today
Penkeeks wif slagroom
★★★★★(2045)
Your Friends:
Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!
Bunger (33 hrs ago)
Balding Boy wrote a review!
Our recommendations:
Lasanga
Nutri-Score: 4/5
Very brief description
Carrot Cake
Nutri-Score: 5/5
Very brief description
Do you have a recipe? Click here to submit it!

Welcome back, FirstName
Recipe of Today
Penkeeks wif slagroom
★★★★★(2045)
Your Friends:
Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!
Bunger (33 hrs ago)
Balding Boy wrote a review!
Our recommendations:
Lasanga
Nutri-Score: 4/5
Very brief description
Carrot Cake
Nutri-Score: 5/5
Very brief description
Do you have a recipe? Click here to submit it!

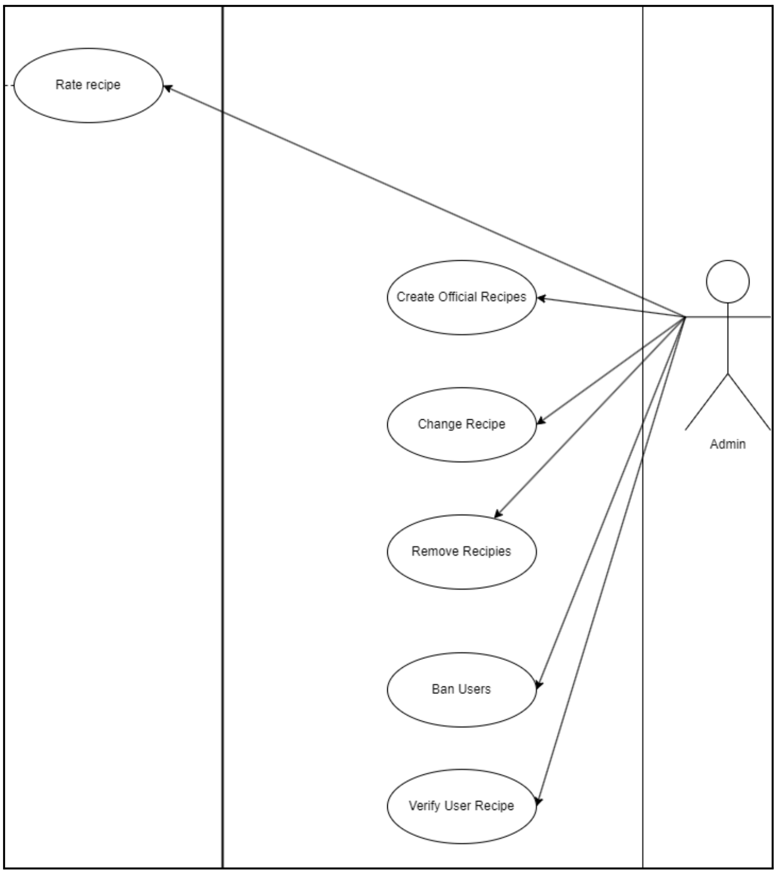
Welcome back, FirstName
Recipe of Today
Penkeeks wif slagroom
★★★★★(2045)
Your Friends:
Loaded Fries (14 hrs ago)
Harold Hiding rated it with ★★★★!
Bunger (33 hrs ago)
Balding Boy wrote a review!
Our recommendations:
Lasanga
Nutri-Score: 4/5
Very brief description
Carrot Cake
Nutri-Score: 5/5
Very brief description
Do you have a recipe? Click here to submit it!

By accounting for users that may not necessarily have the same ability to use our system as the "average" or "most common" user, we improve the accessibility of Take-In, allowing it to reach more people and giving it an edge over its competitors that fail to do so.

Having discussed accessibility, we can now look at the use case diagram of our application. What is shown below is a preliminary version from the early stages of our application's development, so it may not be entirely in line with the final product. We begin with the use case diagram of the users; it may require a little zooming in:



There are a couple of prominent details in this diagram that we would like to focus on, since all of the user interaction aspects of our application and UI have been explained in the Focus on Properties section. First, we tried to make the different parts of our program accessible from each other in some way; this makes using our application easier, frustrating our users less. One example would be being able to access your conversation with another user from both your friends list, the conversation menu and the map. Another such example would be how the shopping list enables you to view where to buy ingredients for recipes, along with being accessible from the page of a recipe that you are viewing.



Then, we have the use case diagram for admin accounts, shown alongside. Admins are capable of doing essentially everything that users are capable of doing, along with some other features. Their permissions include creating pre-verified recipes, changing recipes, removing recipes, banning users and verifying recipes. This is another aspect of Take-In that we are not entirely certain will be in the final product, but would be worth explaining as an extension to the points discussed in section 2.1. The verification system contributes to Take-In's "web of trust" design, along with user ratings. Basically, when a user submits a recipe, it is not verified just yet. It can still be viewed by other users and can be given comments, likes or reviews. However, it will be more difficult to find using our search tool and may not appear as frequently in our random recipe generator in the main menu. When an admin checks this recipe and verifies it, it not only gains a mark to signify this but also gains the privileges made implicit by the previous sentence.

3. Design

3.1 Global Design

We will detail our system's components and the connections between them, one at a time, starting with the index. The index.js file is our main file, serving as the heart of our program. From here, we load all environment variables with dotenv (which will not be elaborated upon in section 3.2, since it contains sensitive data), call all of our config files and start the server.

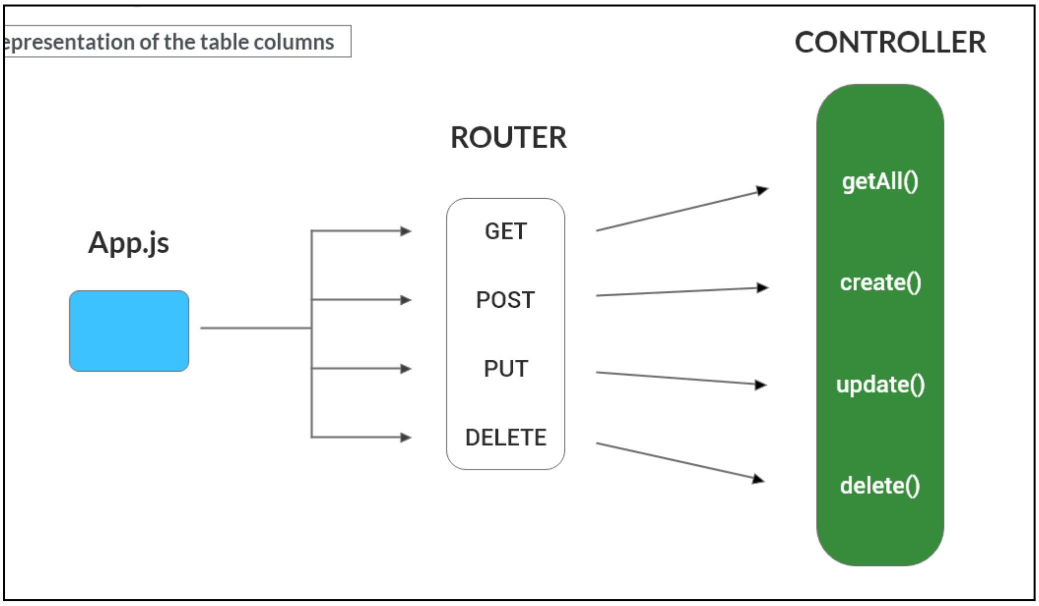
We have three config files; one for setting up expressJS, which functions as our web server. The second one is for mongoose, which is our connection to our database. Finally, we have passportJS, which allows us to authenticate users and log them in.

Following up on authentication, we use Middleware. Whenever the user makes a request, we use Middleware to check if the user is logged in; if they are not, we redirect them back to the login page. We implemented this to prevent users from starting a new session on a different tab or somehow bypassing the security of our application in another way.

We also have model files, which define the structure of our database. Unlike a standard SQL database, we do not use tables; instead, we use collections. Each model file represents one part of our application's databases; for example, we have a file for user data. See section 3.2 for further elaboration.

To connect our program, we use routers and routes. In the index file, we connect to the routers and this tells our server that, if a certain path is called, it is going to connect to a required router and go through it. This sends it through a function, which takes the user to the required page. This is further elaborated upon in section 3.2.

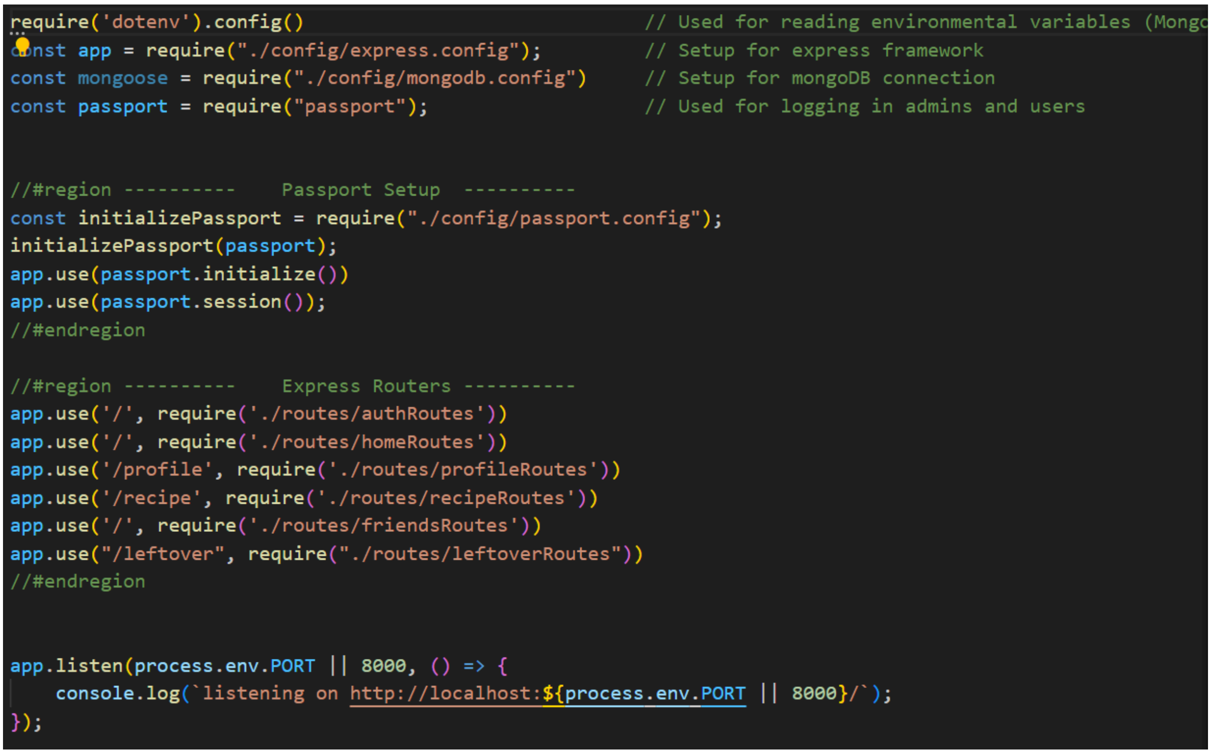
Our routers connect to a controller, which contains all of the functions and code that will be run whenever the user makes a request. See the diagram below[5] for a visualisation:



Finally, we have a "views" folder, which is basically our entire front-end. It is structured such that every route has its own folder, and in those folders are the HTML file for its corresponding pages. There are also folders for CSS files, javascript files, images and templates (pieces of code that are used very often). There is also a global CSS file and a global javascript file, which both contain code that is used frequently.

3.2 Detailed Design

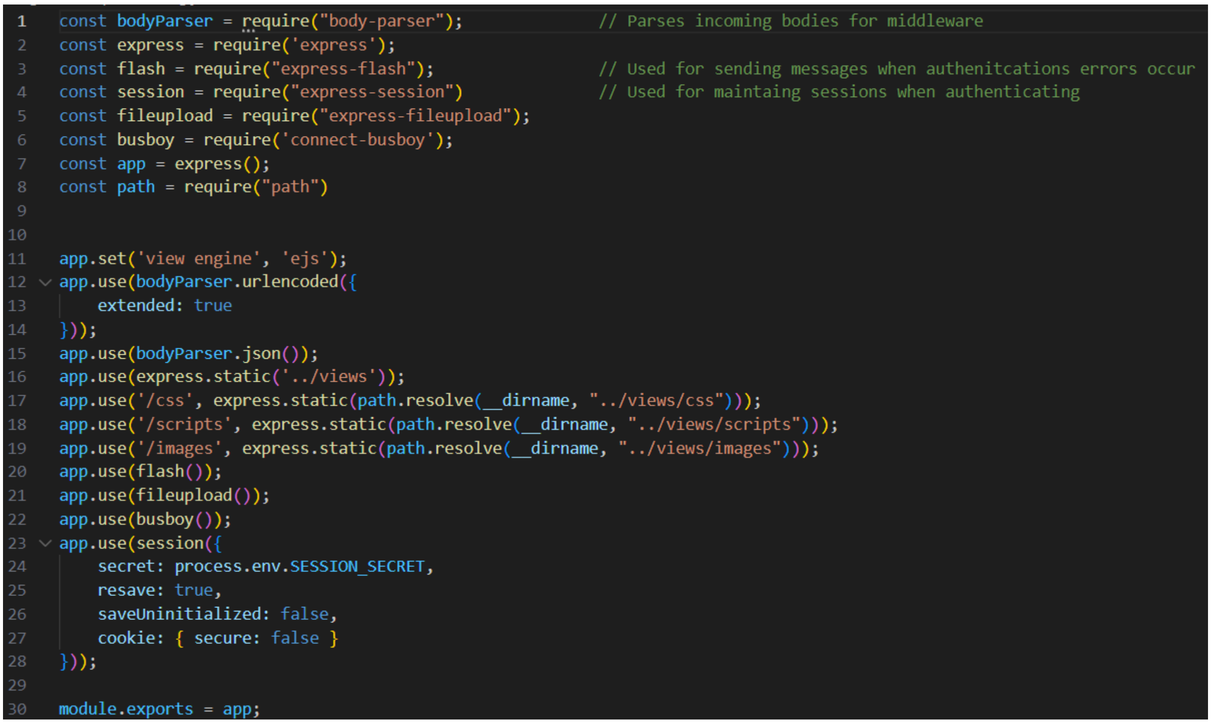
As our project is very expansive, we will not be going to every single controller and explaining every single route. However, we will elaborate upon a few of the key routes, controllers and files in the views folder. We start at the index:



```
require('dotenv').config()                              // Used for reading environmental variables (Mongo
const app = require("./config/express.config");         // Setup for express framework
const mongoose = require("./config/mongodb.config")     // Setup for mongoDB connection
const passport = require("passport");                   // Used for logging in admins and users


//#region ----------    Passport Setup  ----------
const initializePassport = require("./config/passport.config");
initializePassport(passport);
app.use(passport.initialize())
app.use(passport.session());
//#endregion

//#region ----------    Express Routers ----------
app.use('/', require('./routes/authRoutes'))
app.use('/', require('./routes/homeRoutes'))
app.use('/profile', require('./routes/profileRoutes'))
app.use('/recipe', require('./routes/recipeRoutes'))
app.use('/', require('./routes/friendsRoutes'))
app.use("/leftover", require("./routes/leftoverRoutes"))
//#endregion


app.listen(process.env.PORT || 8000, () => {
    console.log(`listening on http://localhost:${process.env.PORT || 8000}/`);
});
```

The first five lines of code call our modules and our config files. The “app” variable is the result of setting up expressJS, which results in our server. The app.use method allows us to use certain functions with our server. In the passport setup section, we integrate passport into our server, allowing for authentication and sessions. Following this, we connect all routers to our server. Finally, we call app.listen() to start our server and listen for connections.

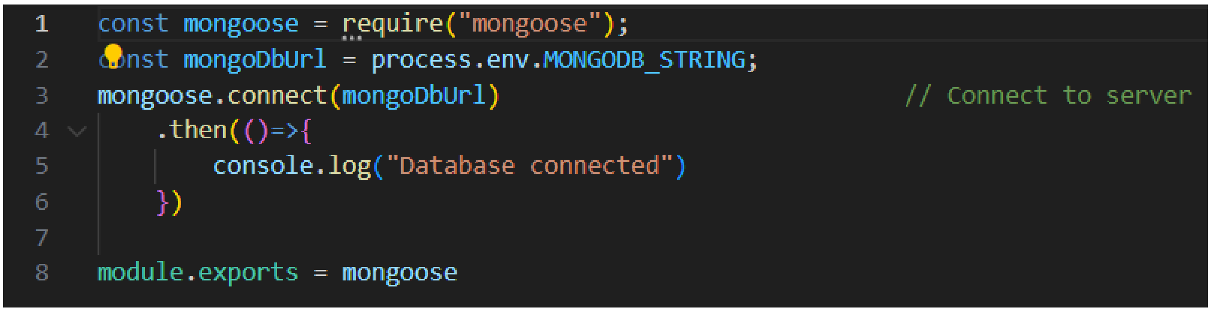
From here, we look at our express.config file:



```
const bodyParser = require("body-parser");                  // Parses incoming bodies for middleware
const express = require('express');
const flash = require("express-flash");                     // Used for sending messages when authenitcations errors occur
const session = require("express-session")                  // Used for maintaing sessions when authenticating
const fileupload = require("express-fileupload");
const busboy = require('connect-busboy');
const app = express();
const path = require("path")


app.set('view engine', 'ejs');
app.use(bodyParser.urlencoded({
    extended: true
}));
app.use(bodyParser.json());
app.use(express.static('../views'));
app.use('/css', express.static(path.resolve(__dirname, "../views/css")));
app.use('/scripts', express.static(path.resolve(__dirname, "../views/scripts")));
app.use('/images', express.static(path.resolve(__dirname, "../views/images")));
app.use(flash());
app.use(fileupload());
app.use(busboy());
app.use(session({
    secret: process.env.SESSION_SECRET,
    resave: true,
    saveUninitialized: false,
    cookie: { secure: false }
}));

module.exports = app;
```

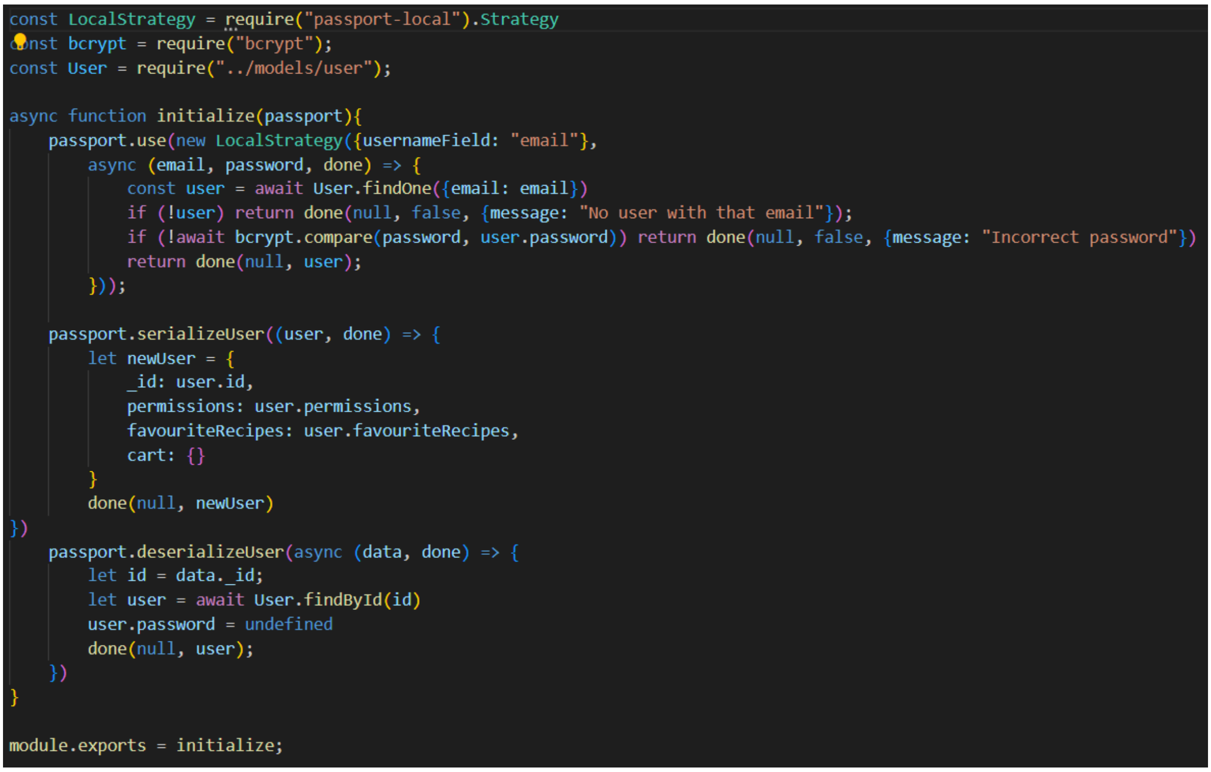
The first eight lines state the required modules that we will integrate into our server. To explain these briefly: body-parser allows us to receive json data through HTTP requests, flash is used to give error messages, session is for maintaining sessions for users so data is persisted through visits, fileupload allows for uploading files and busboy also formats incoming data. After that we mostly call the modules inside of app.use to integrate them into our server. We also set our view engine to ejs which is used to embed data into html. We also set some default directories for easy access for our frontent like '/css', '/scripts' and '/images'.

Now, let us take a look at the mongoose config file, where we set up mongoDB for our databases:



```
const mongoose = require("mongoose");
const mongoDbUrl = process.env.MONGODB_STRING;
mongoose.connect(mongoDbUrl)                       // Connect to server
    .then(()=>{
        console.log("Database connected")
    })

module.exports = mongoose
```

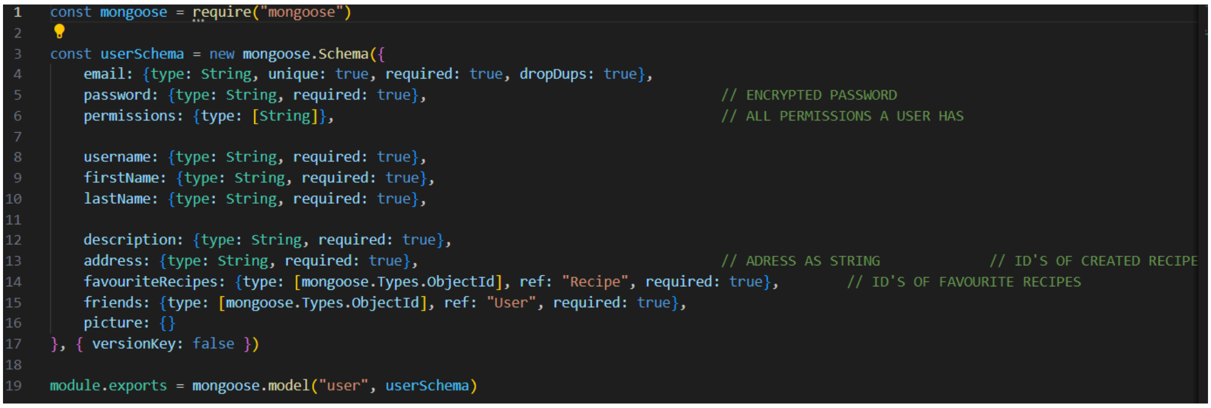
Then, we look at the last config file, which is for passport:



```
const LocalStrategy = require("passport-local").Strategy
const bcrypt = require("bcrypt");
const User = require("../models/user");

async function initialize(passport){
    passport.use(new LocalStrategy({usernameField: "email"},
        async (email, password, done) => {
            const user = await User.findOne({email: email})
            if (!user) return done(null, false, {message: "No user with that email"});
            if (!await bcrypt.compare(password, user.password)) return done(null, false, {message: "Incorrect password"})
            return done(null, user);
        }));

    passport.serializeUser((user, done) => {
        let newUser = {
            _id: user.id,
            permissions: user.permissions,
            favouriteRecipes: user.favouriteRecipes,
            cart: {}
        }
        done(null, newUser)
})
    passport.deserializeUser(async (data, done) => {
        let id = data._id;
        let user = await User.findById(id)
        user.password = undefined
        done(null, user);
    })
}

module.exports = initialize;
```

The first 3 lines are getting the necessary modules. After that we define our initialise function which is called every time a user starts a session and logs in. Inside there are three functions, the first one is run when a user wants to login. We check within the user collection if there is a user with that email, then we use bcrypt to check if the given password matches with the password in the database. If so we are going to serialise the user, which tells the server to create a session corresponding to the user which is used for authenticating the user throughout the program. The last function is for logging a user out, here we basically kill the session.

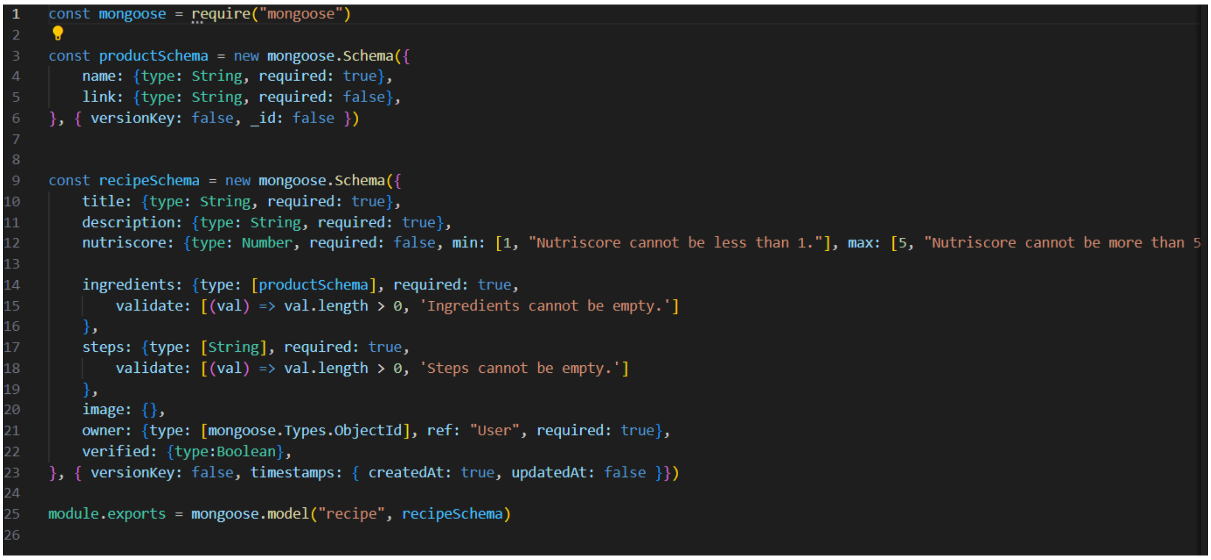
Continuing with the setup, let us now look at our models. There are five of them; we will start with the user model:



```
const mongoose = require("mongoose")

const userSchema = new mongoose.Schema({
    email: {type: String, unique: true, required: true, dropDups: true},
    password: {type: String, required: true},                                      // ENCRYPTED PASSWORD
    permissions: {type: [String]},                                                 // ALL PERMISSIONS A USER HAS

    username: {type: String, required: true},
    firstName: {type: String, required: true},
    lastName: {type: String, required: true},

    description: {type: String, required: true},
    address: {type: String, required: true},                                       // ADRESS AS STRING              // ID'S OF CREATED RECIPE
    favouriteRecipes: {type: [mongoose.Types.ObjectId], ref: "Recipe", required: true},        // ID'S OF FAVOURITE RECIPES
    friends: {type: [mongoose.Types.ObjectId], ref: "User", required: true},
    picture: {}
}, { versionKey: false })

module.exports = mongoose.model("user", userSchema)
```

Our models have fields (or properties, whatever you would prefer to call them), which have traits. For example, take a look at “email”: these fields are of type String, they are unique (as no two users can have the same email), they are required (as they serve as a username) and they get rid of duplicates for the same reason as why they are unique. Focusing on the less self-explanatory fields now: permissions is an array of Strings that basically dictates what a user can or cannot do; we use this mainly to distinguish between admins and regular users. When defining favoriteRecipes and friends, we made their types arrays of objectIDs (since they are lists of things) with references to the type of object they contain (the favourite recipes list contains recipes, for example). One of our “edge cases”, so to speak, would be the picture field for images; we could not give this field a type, as it crashed our program, but we do know that it is stored in binary in our databases and we can use it with that in mind.

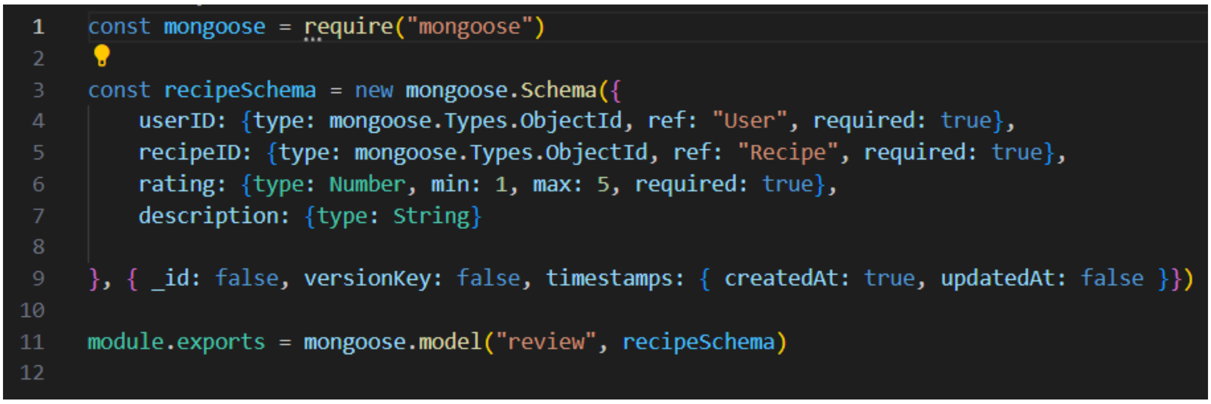
Then the recipe model:



```
const mongoose = require("mongoose")

const productSchema = new mongoose.Schema({
    name: {type: String, required: true},
    link: {type: String, required: false},
}, { versionKey: false, _id: false })


const recipeSchema = new mongoose.Schema({
    title: {type: String, required: true},
    description: {type: String, required: true},
    nutriscore: {type: Number, required: false, min: [1, "Nutriscore cannot be less than 1."], max: [5, "Nutriscore cannot be more than 5

    ingredients: {type: [productSchema], required: true,
        validate: [(val) => val.length > 0, 'Ingredients cannot be empty.']
    },
    steps: {type: [String], required: true,
        validate: [(val) => val.length > 0, 'Steps cannot be empty.']
    },
    image: {},
    owner: {type: [mongoose.Types.ObjectId], ref: "User", required: true},
    verified: {type:Boolean},
}, { versionKey: false, timestamps: { createdAt: true, updatedAt: false }})

module.exports = mongoose.model("recipe", recipeSchema)
```

Our recipe model is a bit more complicated as it exists out of two Schema’s. A recipe exists out of a Title and description which are simple strings and a nutriscore which is an integer between 1 and 5, a boolean verified which is true when it is approved by a moderator and an owner, which is an objectId which references a user. A recipe can also contain an image which is stored as binary in base64. The steps for a recipe is just an array of strings. For our ingredient we are going to use our second schema, which consists of a name of the ingredient and a link to the website where the ingredient can be bought. We also store the timestamp of when the recipe is created. For determining recent recipes on our homepage.

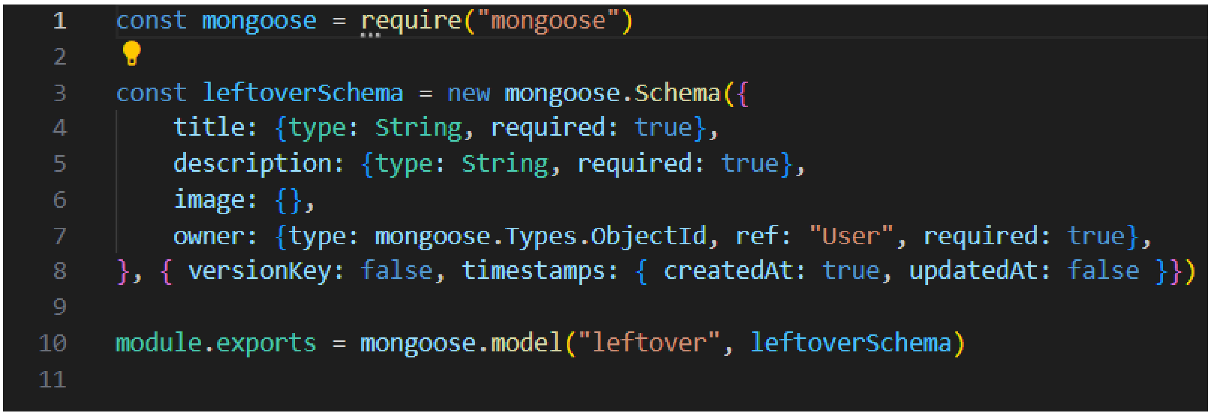
Then the review model:



```
const mongoose = require("mongoose")

const recipeSchema = new mongoose.Schema({
    userID: {type: mongoose.Types.ObjectId, ref: "User", required: true},
    recipeID: {type: mongoose.Types.ObjectId, ref: "Recipe", required: true},
    rating: {type: Number, min: 1, max: 5, required: true},
    description: {type: String}

}, { _id: false, versionKey: false, timestamps: { createdAt: true, updatedAt: false }})

module.exports = mongoose.model("review", recipeSchema)
```

A review consists of the ObjectId of the user who wrote it, the ObjectId of the recipe it was written about, a rating between 1 and 5 and an optional description about the recipe. We store the timestamp of when it was created to determine hot/popular recipes at the moment.

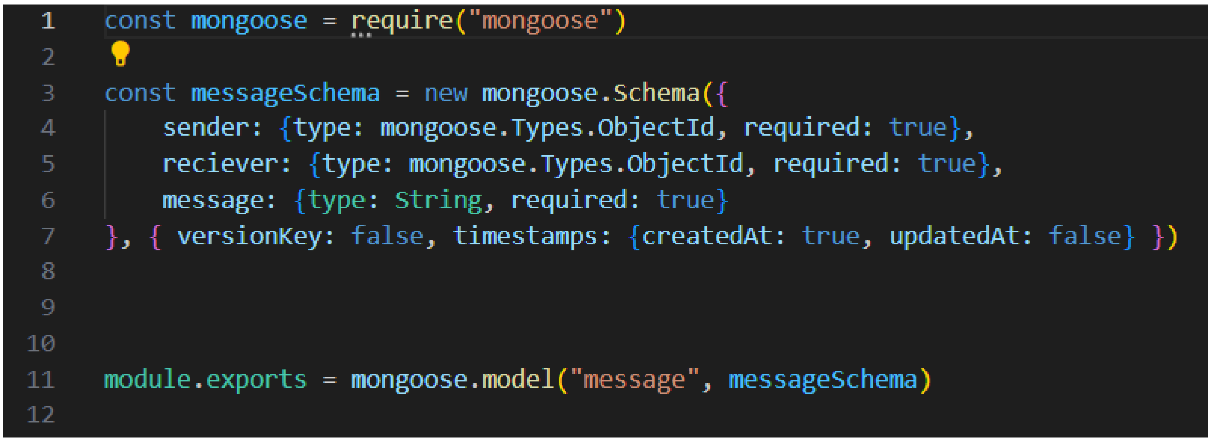
Then the leftovers model:



```
const mongoose = require("mongoose")

const leftoverSchema = new mongoose.Schema({
    title: {type: String, required: true},
    description: {type: String, required: true},
    image: {},
    owner: {type: mongoose.Types.ObjectId, ref: "User", required: true},
}, { versionKey: false, timestamps: { createdAt: true, updatedAt: false }})

module.exports = mongoose.model("leftover", leftoverSchema)
```

A leftover is a strippeddown version of a recipe, consisting of a title and description which are stored as strings, an optional image stored as binary in base64 and the ObjectId of the user who created the leftover.

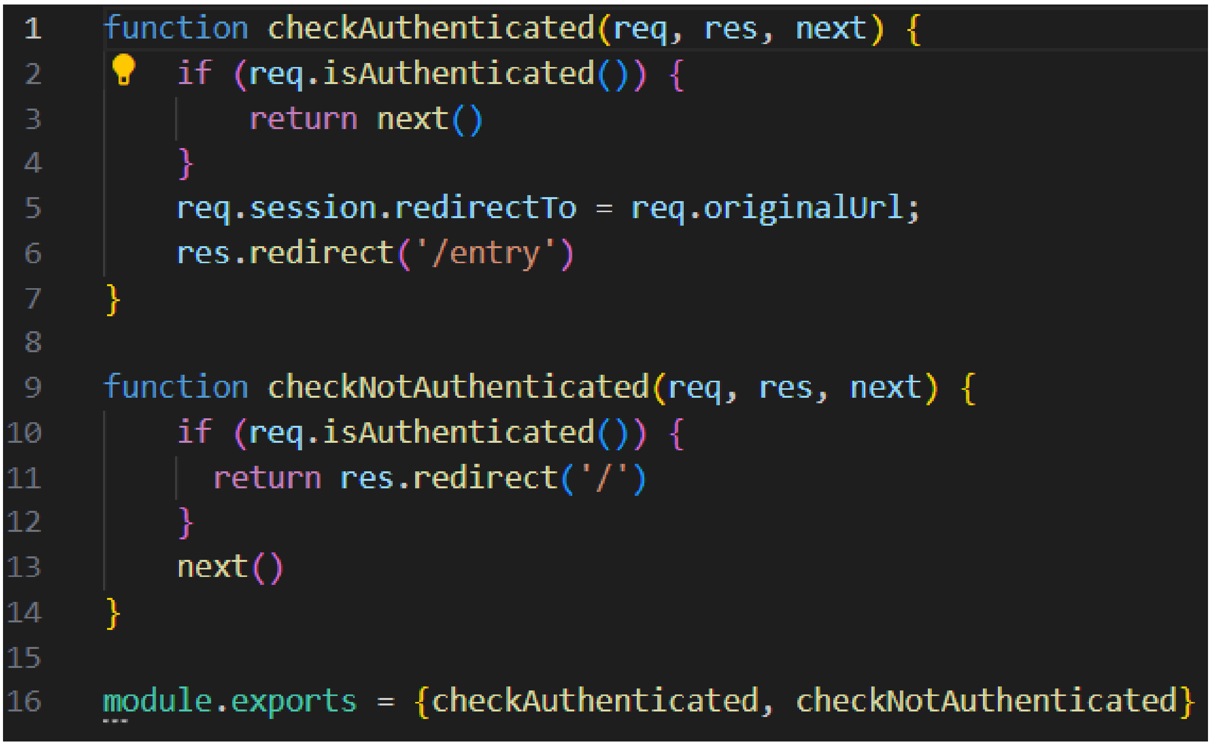
And finally, the message model:



```
const mongoose = require("mongoose")

const messageSchema = new mongoose.Schema({
    sender: {type: mongoose.Types.ObjectId, required: true},
    reciever: {type: mongoose.Types.ObjectId, required: true},
    message: {type: String, required: true}
}, { versionKey: false, timestamps: {createdAt: true, updatedAt: false} })



module.exports = mongoose.model("message", messageSchema)
```

The message schema is used for storing chat messages between users. It stores the two ObjectIds from the users of the conversation, with the sender storing the id of which user sent the message and receiver the id who it was sent to. We also store the timestamp of the time the message was sent. The message is stored as a simple String.

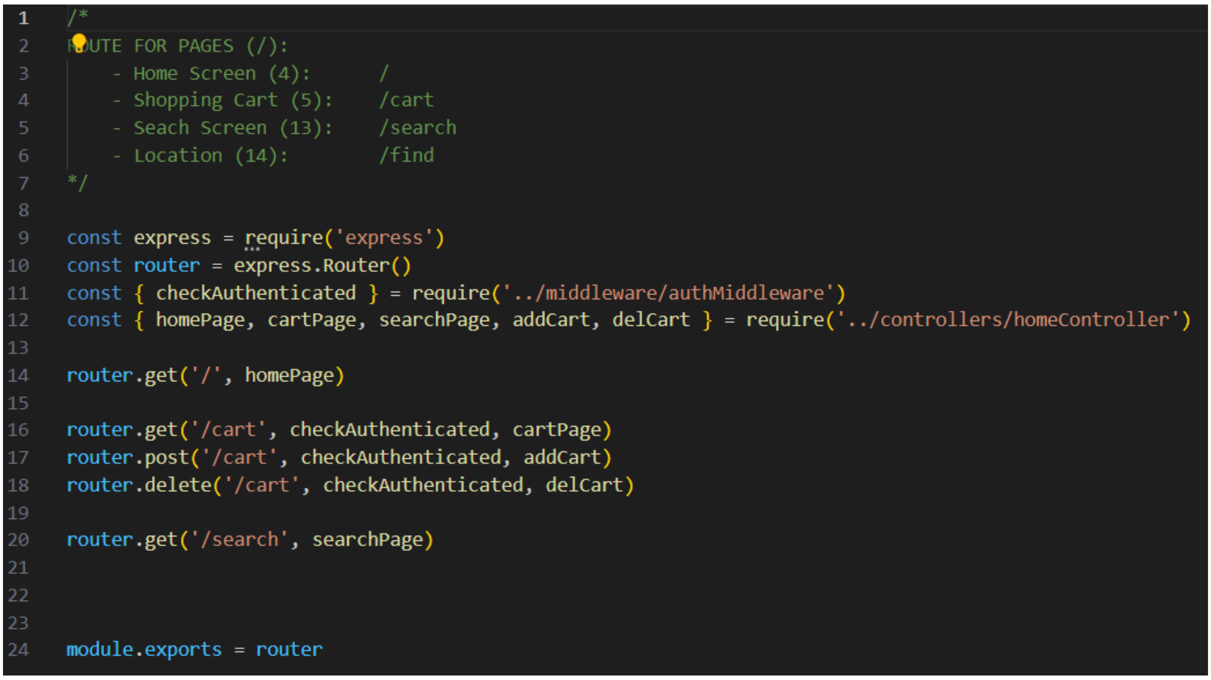
Let us look at our middleware file next:



```
function checkAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {
    return next()
  }
  req.session.redirectTo = req.originalUrl;
  res.redirect('/entry')
}

function checkNotAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {
   return res.redirect('/')
  }
  next()
}

module.exports = {checkAuthenticated, checkNotAuthenticated}
```

This file consists of two functions. checkAuthenticated is used for pages that users aren't allowed to visit when they are not logged in. CheckNotAuthenticated is used for the login, register and entry page so the user doesn't login twice.

As we have a lot of routes and we do not want to make our project report obscenely long, we instead chose to only follow one for the sake of the detailed design descriptions; the home router. However, this should not be a huge problem, because all of our routers are similar in functionality. That being said, this is our homeRouter file:



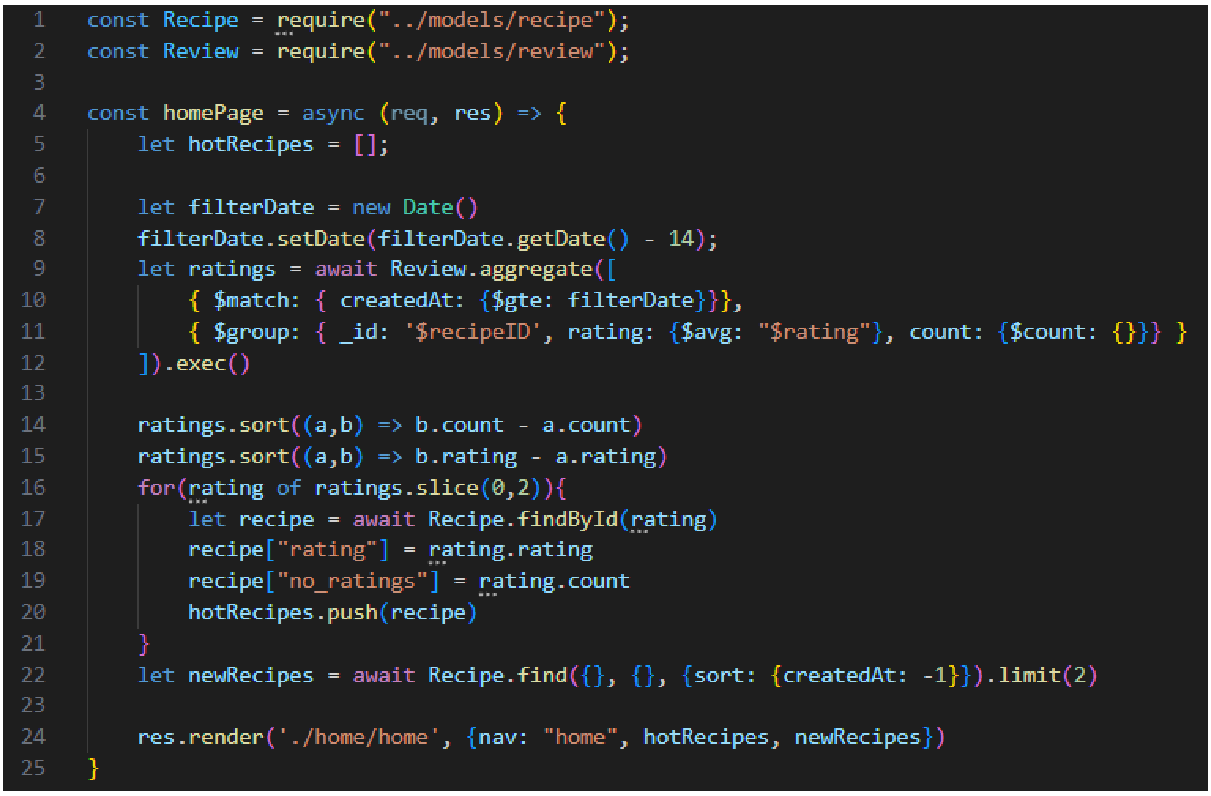
```
/*
ROUTE FOR PAGES (/):
    - Home Screen (4):      /
    - Shopping Cart (5):    /cart
    - Seach Screen (13):    /search
    - Location (14):        /find
*/

const express = require('express')
const router = express.Router()
const { checkAuthenticated } = require('../middleware/authMiddleware')
const { homePage, cartPage, searchPage, addCart, delCart } = require('../controllers/homeController')

router.get('/', homePage)

router.get('/cart', checkAuthenticated, cartPage)
router.post('/cart', checkAuthenticated, addCart)
router.delete('/cart', checkAuthenticated, delCart)

router.get('/search', searchPage)



module.exports = router
```

Every router starts with calling express and setting up a router, then calling middleware functions and the controller corresponding to the route. Then we use router.get(), router.post() or router.delete() to connect the right path with the right function; this function comes from inside of the controller. Then, we export the router. While this sounds fairly compact in description, it is slightly more complicated in application.

As our controllers make up the bulk of our code (consisting of over 600 lines), we would have far too much detail for this project report. Instead, we will show the most prominent functions in our controllers (namely, the home page and most of the recipe controller), and leave the rest for the examiners to see.

Before going into this, we wanted to briefly explain the general function of our controllers. When a user makes a request there are two objects passed through the route to the controller; the request object and the response object. We can use the request object to access data, such as query data, parameters, json data passed with post or delete requests, the current session of the user which also includes some user data, et cetera. The response object, which we call res, is used to reply to the user, this is mostly done with either res.render("/path/to/file"), which renders an ejs file and is used for get requests, or with res.status(ERROR_CODE).send("message"), which lets the user know if the request was successful or not according to the ERROR_CODE and can contain an additional message, used for post and delete requests. So every function in the controller gets some data out of the request object, and then the server responds using the response object. Every controller ends by exporting all of its functions so they can be used in the router.

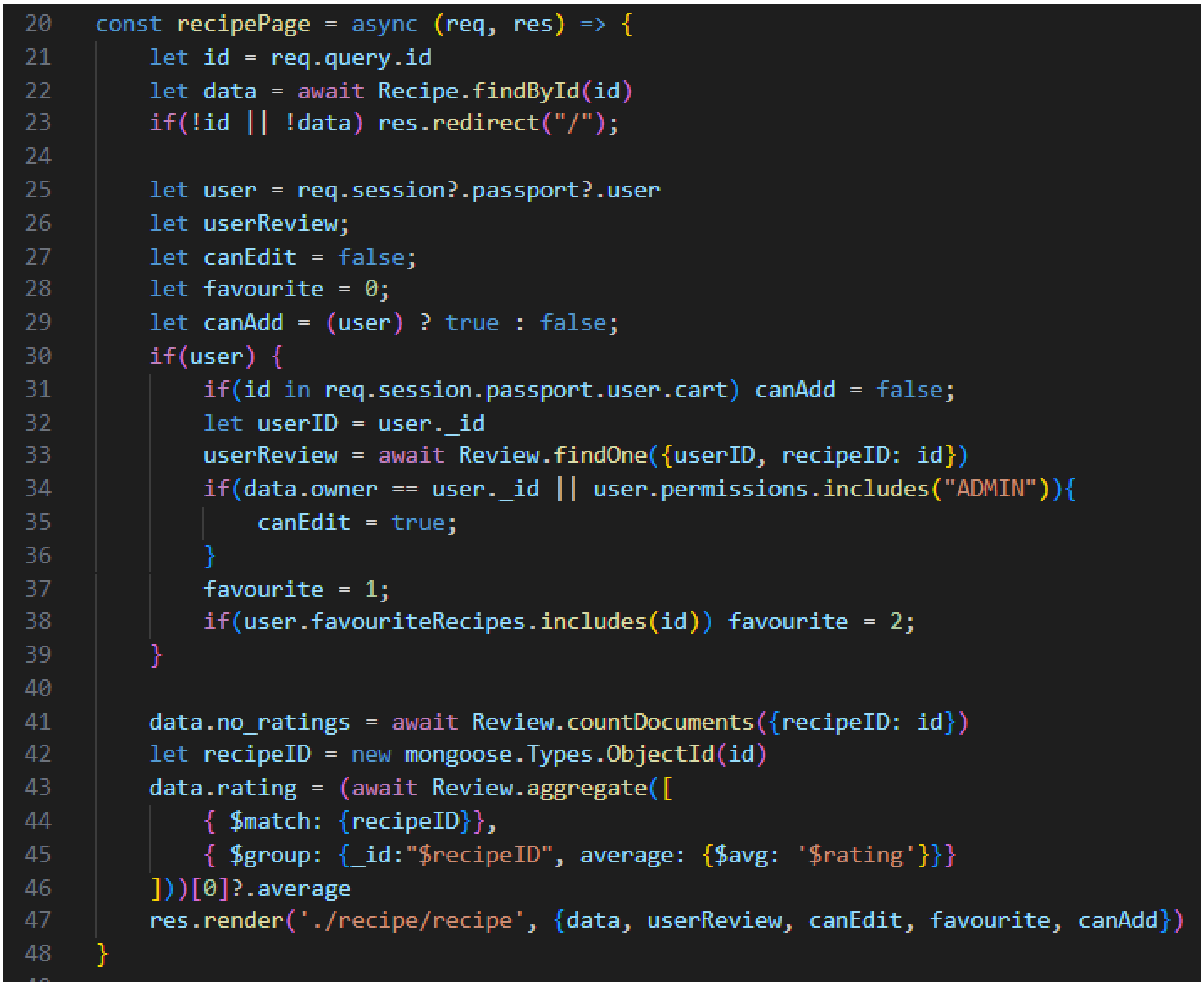
With that being said, let us begin by looking at a part of the home controller:



```
const Recipe = require("../models/recipe");
const Review = require("../models/review");

const homePage = async (req, res) => {
    let hotRecipes = [];

    let filterDate = new Date()
    filterDate.setDate(filterDate.getDate() - 14);
    let ratings = await Review.aggregate([
        { $match: { createdAt: {$gte: filterDate}}},
        { $group: { _id: '$recipeID', rating: {$avg: "$rating"}, count: {$count: {}}} }
    ]).exec()

    ratings.sort((a,b) => b.count - a.count)
    ratings.sort((a,b) => b.rating - a.rating)
    for(rating of ratings.slice(0,2)){
        let recipe = await Recipe.findById(rating)
        recipe["rating"] = rating.rating
        recipe["no_ratings"] = rating.count
        hotRecipes.push(recipe)
    }
    let newRecipes = await Recipe.find({}, {}, {sort: {createdAt: -1}}).limit(2)

    res.render('./home/home', {nav: "home", hotRecipes, newRecipes})
}
```

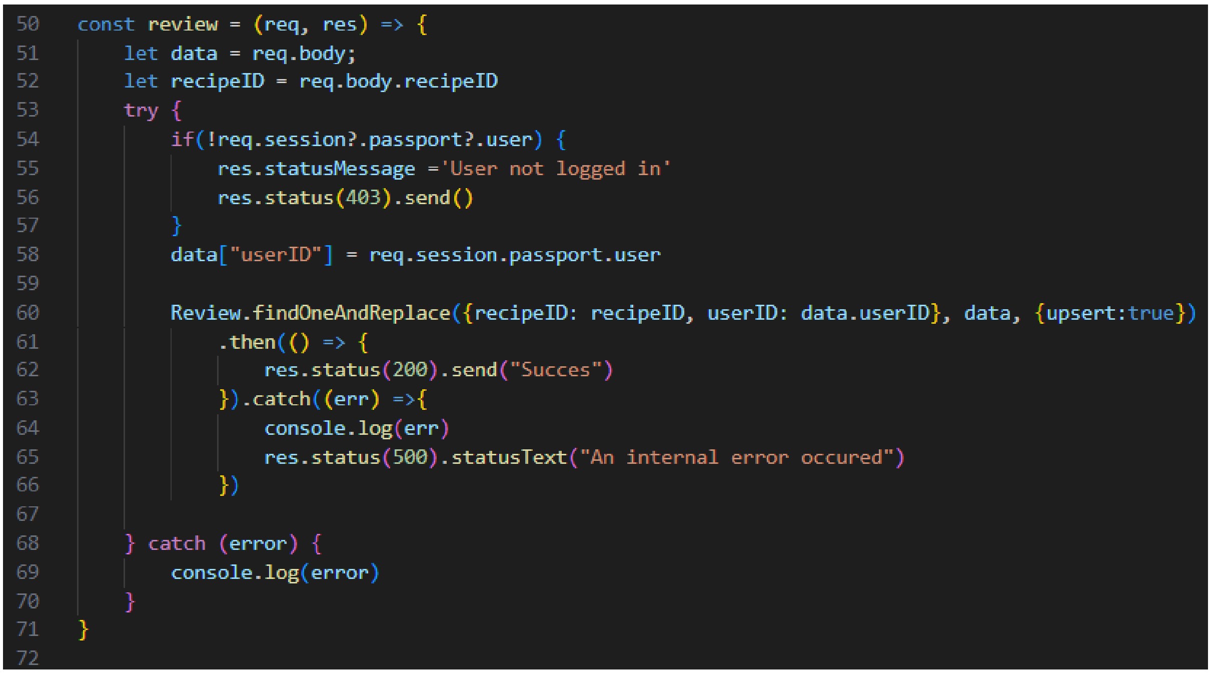
This function is for gathering and formatting the data for our homepage. The homepage consists of three sections, the random recipe generator, which works by making a request to '/recipe/random' which will be explained later, the hot recipes category and the new recipes category. Hot recipes shows two recipes which have the highest rating in the past two weeks and for the new recipes the two most recent. For the hot recipes we get the current date and subtract 14 days from it, then we use this to look up all reviews in this timeframe and calculate the average and number of reviews for every recipe. After that we sort the recipe first according to the number of reviews and then the average rating. This ensures that a recipe with the same rating but more reviews ends up on top. Then we pick the top two recipes, get the data from the database and then put them in the hotRecipes array. Getting the new recipes is a lot easier since we can just use mongoose to sort the recipes based on createdAt and then limit it to two results. We then use res.render to render the homepage and pass our data to the formatter.

Then, some segments of the recipe controller. Since this is our largest controller, we have split it up by function and will explain each one individually.



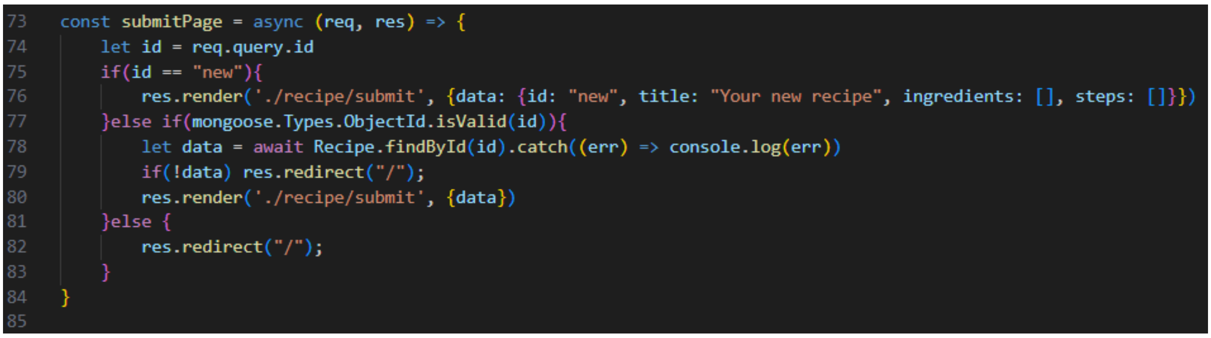
```
const recipePage = async (req, res) => {
    let id = req.query.id
    let data = await Recipe.findById(id)
    if(!id || !data) res.redirect("/");

    let user = req.session?.passport?.user
    let userReview;
    let canEdit = false;
    let favourite = 0;
    let canAdd = (user) ? true : false;
    if(user) {
        if(id in req.session.passport.user.cart) canAdd = false;
        let userID = user._id
        userReview = await Review.findOne({userID, recipeID: id})
        if(data.owner == user._id || user.permissions.includes("ADMIN")){
            canEdit = true;
        }
        favourite = 1;
        if(user.favouriteRecipes.includes(id)) favourite = 2;
    }

    data.no_ratings = await Review.countDocuments({recipeID: id})
    let recipeID = new mongoose.Types.ObjectId(id)
    data.rating = (await Review.aggregate([
        { $match: {recipeID}},
        { $group: {_id:"$recipeID", average: {$avg: '$rating'}}}
    ]))[0]?.average
    res.render('./recipe/recipe', {data, userReview, canEdit, favourite, canAdd})
}
```

This function is for rendering the recipe page, which shows a single recipe in more detail and allows for leaving reviews and editing if the user has the permissions. First we extract the id out of the url and using that url search for the recipe. If we don't find one we redirect the user to the homepage, otherwise we check if there is a user logged in and put it into user. We create some variables like userReview, canEdit, favourite and canAdd. Favourite is 0 when the user is not logged in, 1 when the user doesn't have the recipe as a favourite and 2 if it does. CanEdit means that the user can edit the recipe which is the case if it's the owner and canAdd means that the user is logged in and can add the recipe to his cart. UserReview contains the data about the review if the user has left one for this recipe. We change these values accordingly if there is a user logged in. After that we put the number of ratings into data.no_ratings and get the average ratings. Then we pass this along to res.render to render the page.



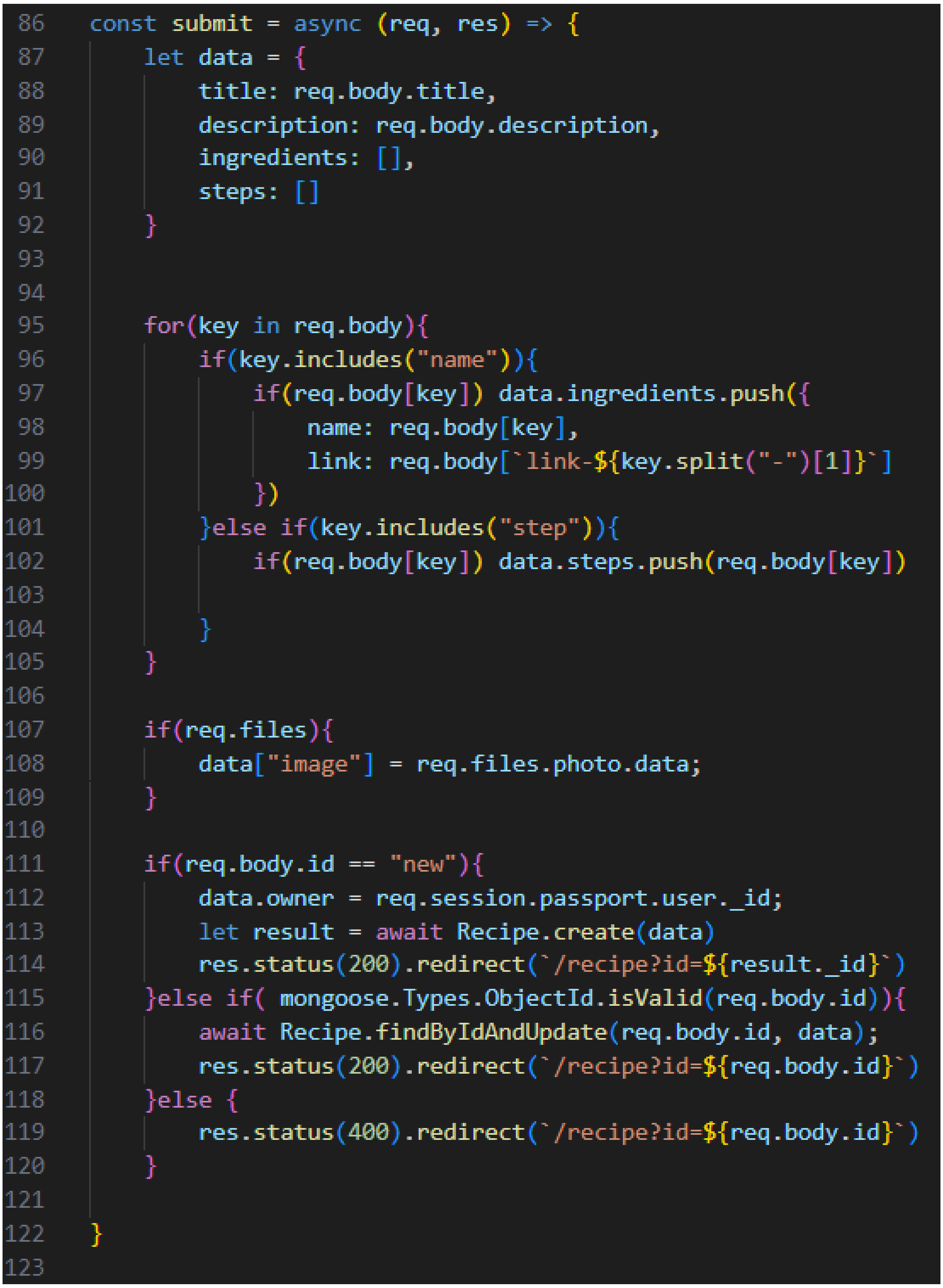
```
const review = (req, res) => {
    let data = req.body;
    let recipeID = req.body.recipeID
    try {
        if(!req.session?.passport?.user) {
            res.statusMessage ='User not logged in'
            res.status(403).send()
        }
        data["userID"] = req.session.passport.user

        Review.findOneAndReplace({recipeID: recipeID, userID: data.userID}, data, {upsert:true})
            .then(() => {
                res.status(200).send("Succes")
            }).catch((err) =>{
                console.log(err)
                res.status(500).statusText("An internal error occured")
            })

    } catch (error) {
        console.log(error)
    }
}
```

This is the code for creating a review. Data will consist of the recipeId, rating and message. We first check if the user is logged in by checking the active session. If the user is, we are going to search for a review with that userId and recipeId. If one isn't found we are going to upsert/create one, otherwise we replace it with the new one.



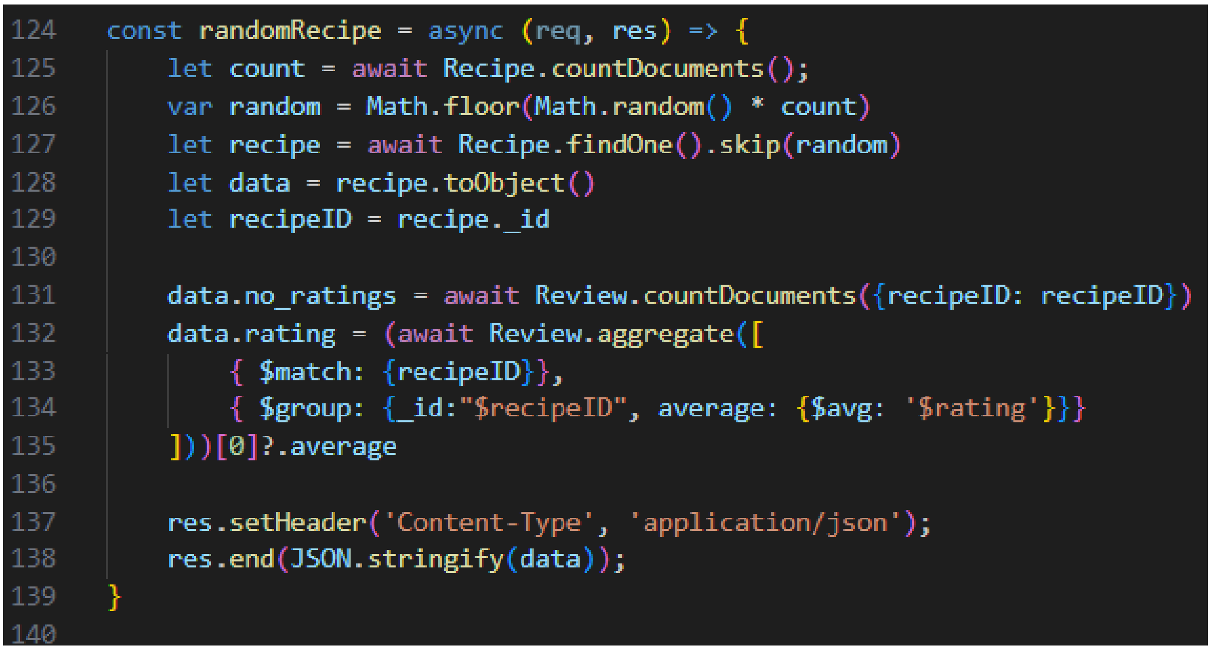
```
const submitPage = async (req, res) => {
    let id = req.query.id
    if(id == "new"){
        res.render('./recipe/submit', {data: {id: "new", title: "Your new recipe", ingredients: [], steps: []}})
    }else if(mongoose.Types.ObjectId.isValid(id)){
        let data = await Recipe.findById(id).catch((err) => console.log(err))
        if(!data) res.redirect("/");
        res.render('./recipe/submit', {data})
    }else {
        res.redirect("/");
    }
}
```

The function for rendering the recipe page is fairly simple. If the id is new it means that we are going to create a new recipe, we pass some default variables for data and render the page. If the id is valid we grab the data for the recipe and also render the page. Otherwise we redirect the user back to the home page.



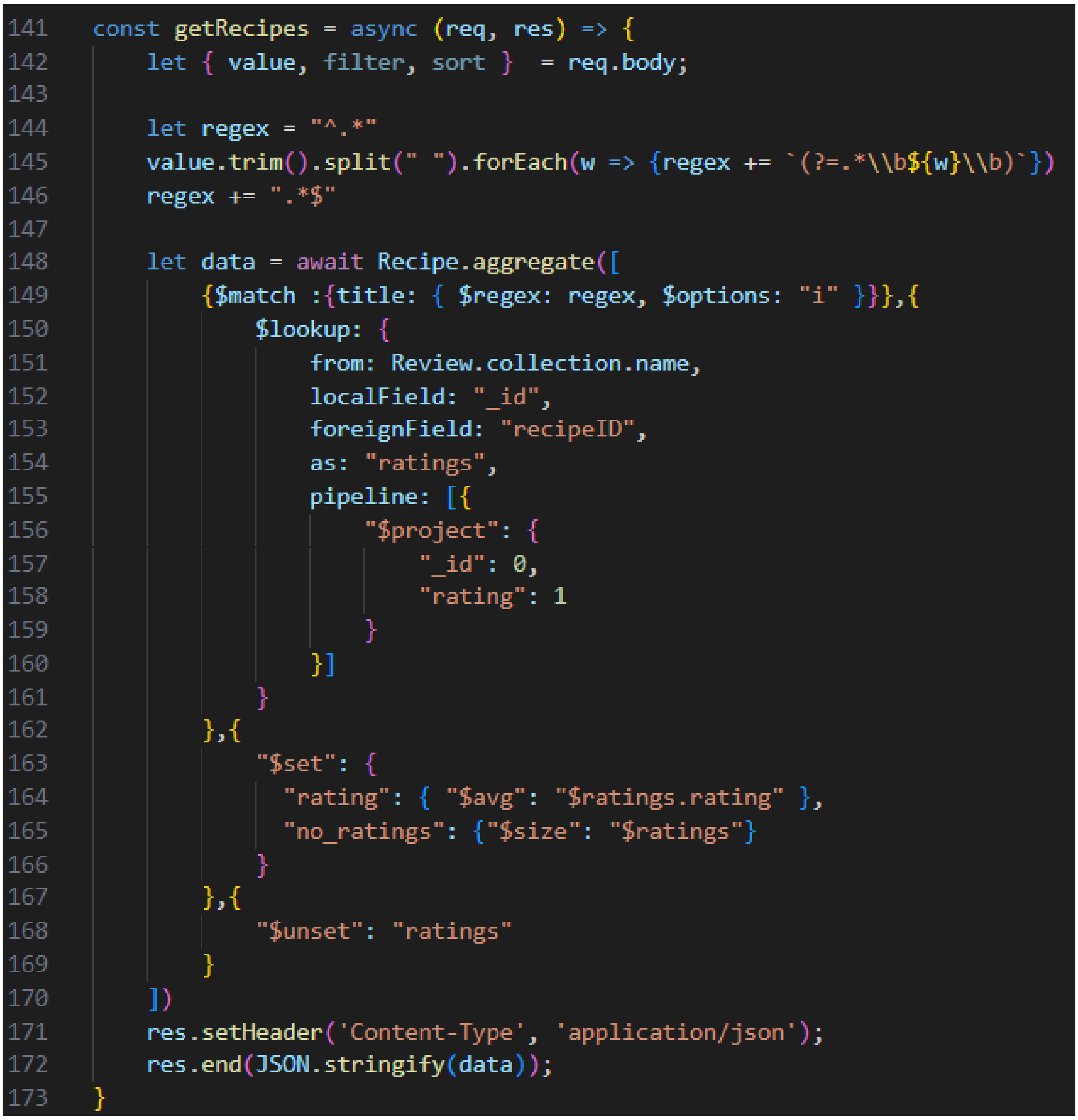
```
const submit = async (req, res) => {
    let data = {
        title: req.body.title,
        description: req.body.description,
        ingredients: [],
        steps: []
    }


    for(key in req.body){
        if(key.includes("name")){
            if(req.body[key]) data.ingredients.push({
                name: req.body[key],
                link: req.body[`link-${key.split("-")[1]}`]
            })
        }else if(key.includes("step")){
            if(req.body[key]) data.steps.push(req.body[key])

        }
    }

    if(req.files){
        data["image"] = req.files.photo.data;
    }

    if(req.body.id == "new"){
        data.owner = req.session.passport.user._id;
        let result = await Recipe.create(data)
        res.status(200).redirect(`/recipe?id=${result._id}`)
    }else if( mongoose.Types.ObjectId.isValid(req.body.id)){
        await Recipe.findByIdAndUpdate(req.body.id, data);
        res.status(200).redirect(`/recipe?id=${req.body.id}`)
    }else {
        res.status(400).redirect(`/recipe?id=${req.body.id}`)
    }

}

```

When a user submits a recipe, either new or an existing one, this function runs. We put the title and description from the body directly into data. After that we loop over all the keys and check if they contain either name or step. If it includes ‘name’ we know that there is a corresponding link key to it. We combine the name and link into one object and put it into the array of ingredients. For each step we do the same but then we put it into data.steps. After that we check if the request contains an image, we also put that into data. Then if the id equals new we create a new recipe. If it doesn’t we assume it's an existing recipe and update that recipe. If somehow the user sends the form with an invalid id we redirect them back to the recipe.



```
const randomRecipe = async (req, res) => {
    let count = await Recipe.countDocuments();
    var random = Math.floor(Math.random() * count)
    let recipe = await Recipe.findOne().skip(random)
    let data = recipe.toObject()
    let recipeID = recipe._id

    data.no_ratings = await Review.countDocuments({recipeID: recipeID})
    data.rating = (await Review.aggregate([
        { $match: {recipeID}},
        { $group: {_id:"$recipeID", average: {$avg: '$rating'}}}
    ]))[0]?.average

    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify(data));
}
```

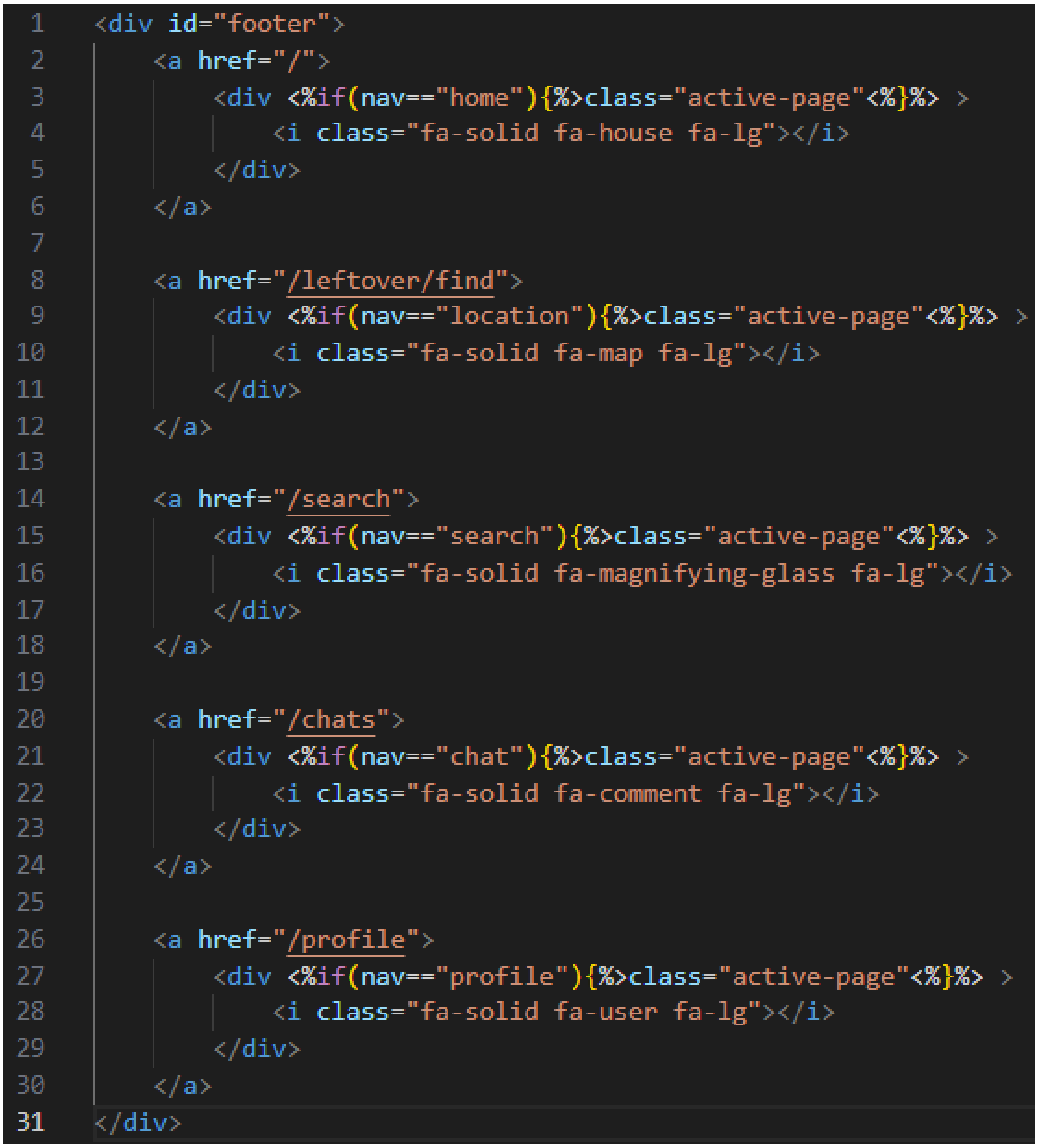
This is the function for grabbing a random recipe mentioned earlier. First we need to know how many recipes there are and grab a random number in between 1 and the count. We use findOne() and skip() to findOne document and skip a random amount of recipes to grab a random recipe instead of the first. Then we grab the number of ratings and the average for this recipe and return this data to the user.



```
const getRecipes = async (req, res) => {
    let { value, filter, sort }  = req.body;

    let regex = "^.*"
    value.trim().split(" ").forEach(w => {regex += `(?=.*\\b${w}\\b)`})
    regex += ".*$"

    let data = await Recipe.aggregate([
        {$match :{title: { $regex: regex, $options: "i" }}},{
            $lookup: {
                from: Review.collection.name,
                localField: "_id",
                foreignField: "recipeID",
                as: "ratings",
                pipeline: [{
                    "$project": {
                        "_id": 0,
                        "rating": 1
                    }
                }]
            }
        },{
            "$set": {
              "rating": { "$avg": "$ratings.rating" },
              "no_ratings": {"$size": "$ratings"}
            }
        },{
            "$unset": "ratings"
        }
    ])
    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify(data));
}
```

This is the function used for searching for recipes. Our search function allows you to find recipes that contain the same words as you searched for. For example: If the user searches for “Rice Chicken”, it returns every recipe that contains the words ‘Rice’ and ‘Chicken’, no matter what order. We use a regular expression for this. After that we look for all the reviews for every matching recipe and get the average rating and number of ratings from them respectively.

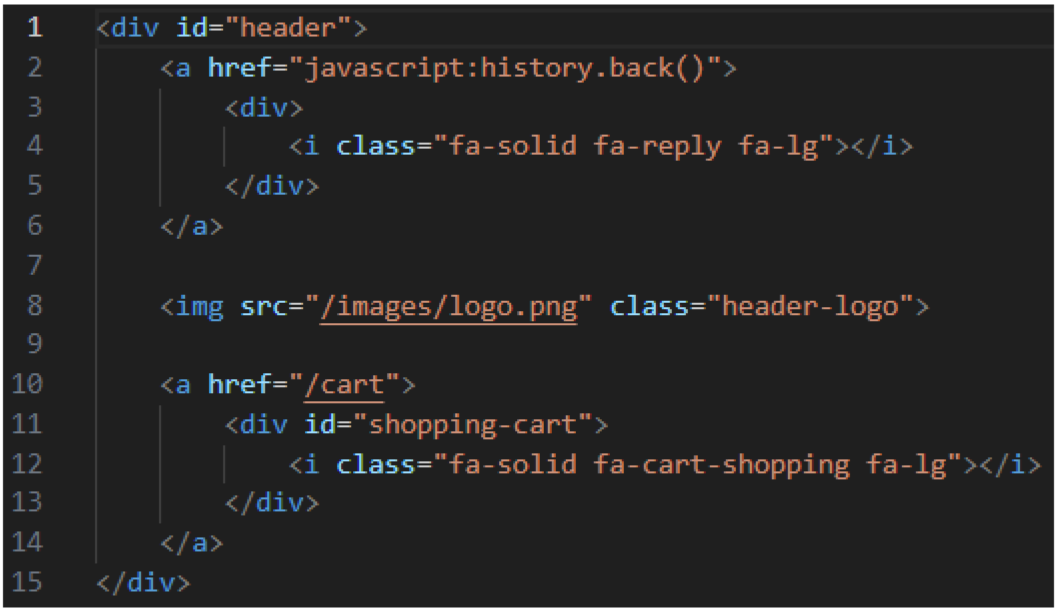
Then, after having looked at two of our five controllers, let us now move to the templates. These are very important parts of our project, because they define fragments of code that are used very often. We begin with the footer template, for our UI’s bottom bar:



```
<div id="footer">
    <a href="/">
        <div <%if(nav=="home"){%>class="active-page"<%}%> >
            <i class="fa-solid fa-house fa-lg"></i>
        </div>
    </a>

    <a href="/leftover/find">
        <div <%if(nav=="location"){%>class="active-page"<%}%> >
            <i class="fa-solid fa-map fa-lg"></i>
        </div>
    </a>

    <a href="/search">
        <div <%if(nav=="search"){%>class="active-page"<%}%> >
            <i class="fa-solid fa-magnifying-glass fa-lg"></i>
        </div>
    </a>

    <a href="/chats">
        <div <%if(nav=="chat"){%>class="active-page"<%}%> >
            <i class="fa-solid fa-comment fa-lg"></i>
        </div>
    </a>

    <a href="/profile">
        <div <%if(nav=="profile"){%>class="active-page"<%}%> >
            <i class="fa-solid fa-user fa-lg"></i>
        </div>
    </a>
</div>
```

The footer is active on every page, it shows five icons that each link to a different page. We pass the variable nav into the footer to determine on what page the user currently is and then give it the class 'active-page' which gives that icon a different style.

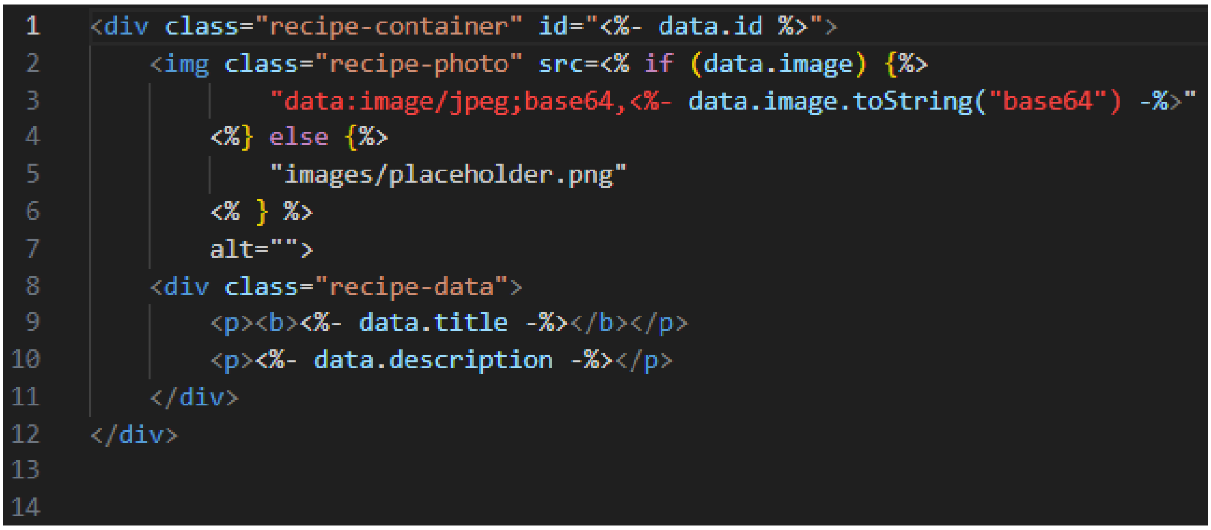
Then, the header template, for our UI's top bar:



```html
<div id="header">
    <a href="javascript:history.back()">
        <div>
            <i class="fa-solid fa-reply fa-lg"></i>
        </div>
    </a>

    <img src="/images/logo.png" class="header-logo">

    <a href="/cart">
        <div id="shopping-cart">
            <i class="fa-solid fa-cart-shopping fa-lg"></i>
        </div>
    </a>
</div>
```

The header is also on every page, it has the logo in the middle, a back button on the left which allows the user to go back one page and a shopping cart which links to the cart page.

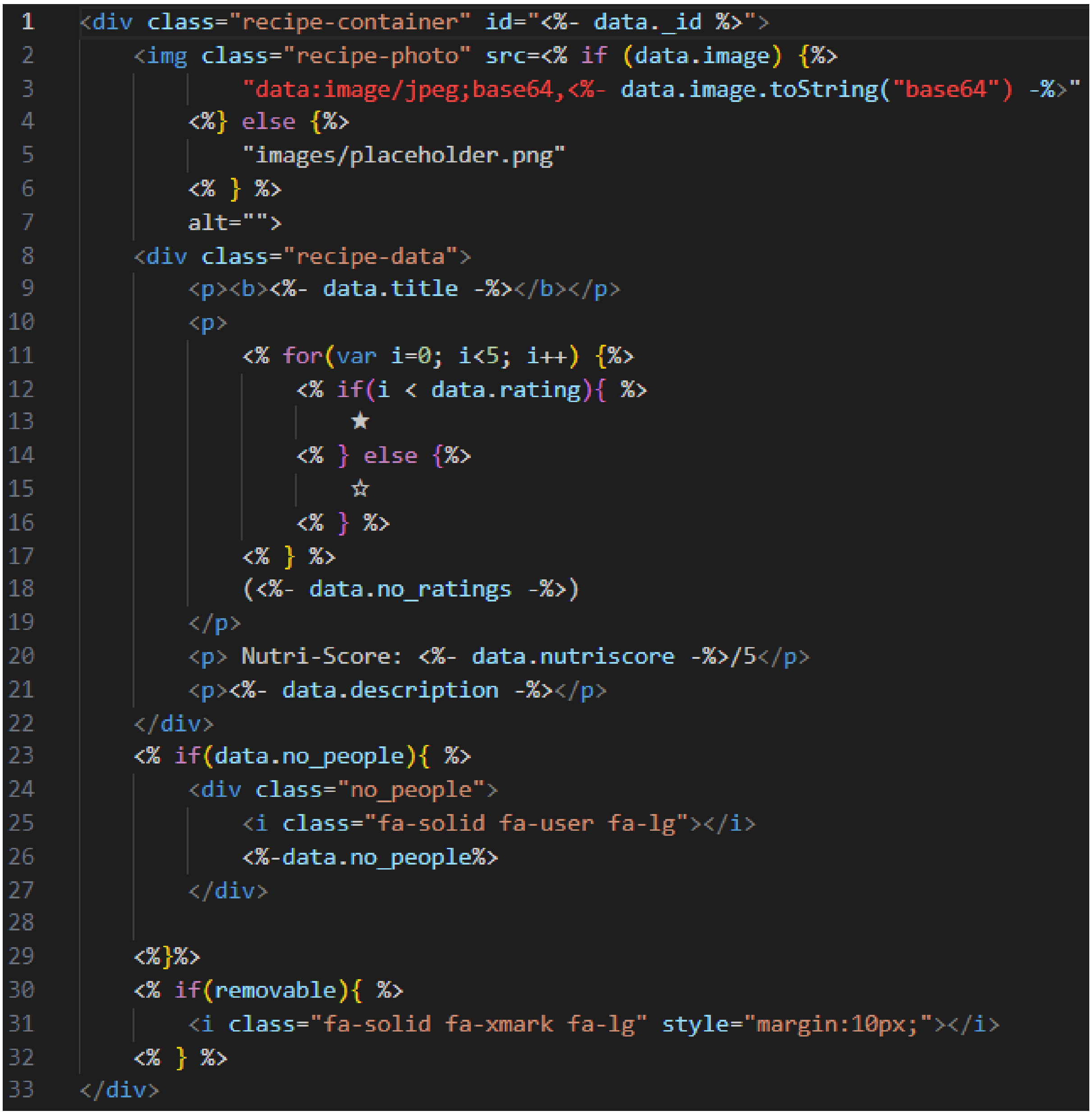
Then, the leftovers template:



```html
<div class="recipe-container" id="<%- data.id %>">
    <img class="recipe-photo" src=<% if (data.image) {%>
            "data:image/jpeg;base64,<%- data.image.toString("base64") -%>"
        <%} else {%>
            "images/placeholder.png"
        <% } %>
        alt="">
    <div class="recipe-data">
        <p><b><%- data.title -%></b></p>
        <p><%- data.description -%></p>
    </div>
</div>


```

This is a simple container for showing leftovers on the leftover finder page. It shows the image of the leftover if it exists and a placeholder image otherwise. With the title and description next to it.

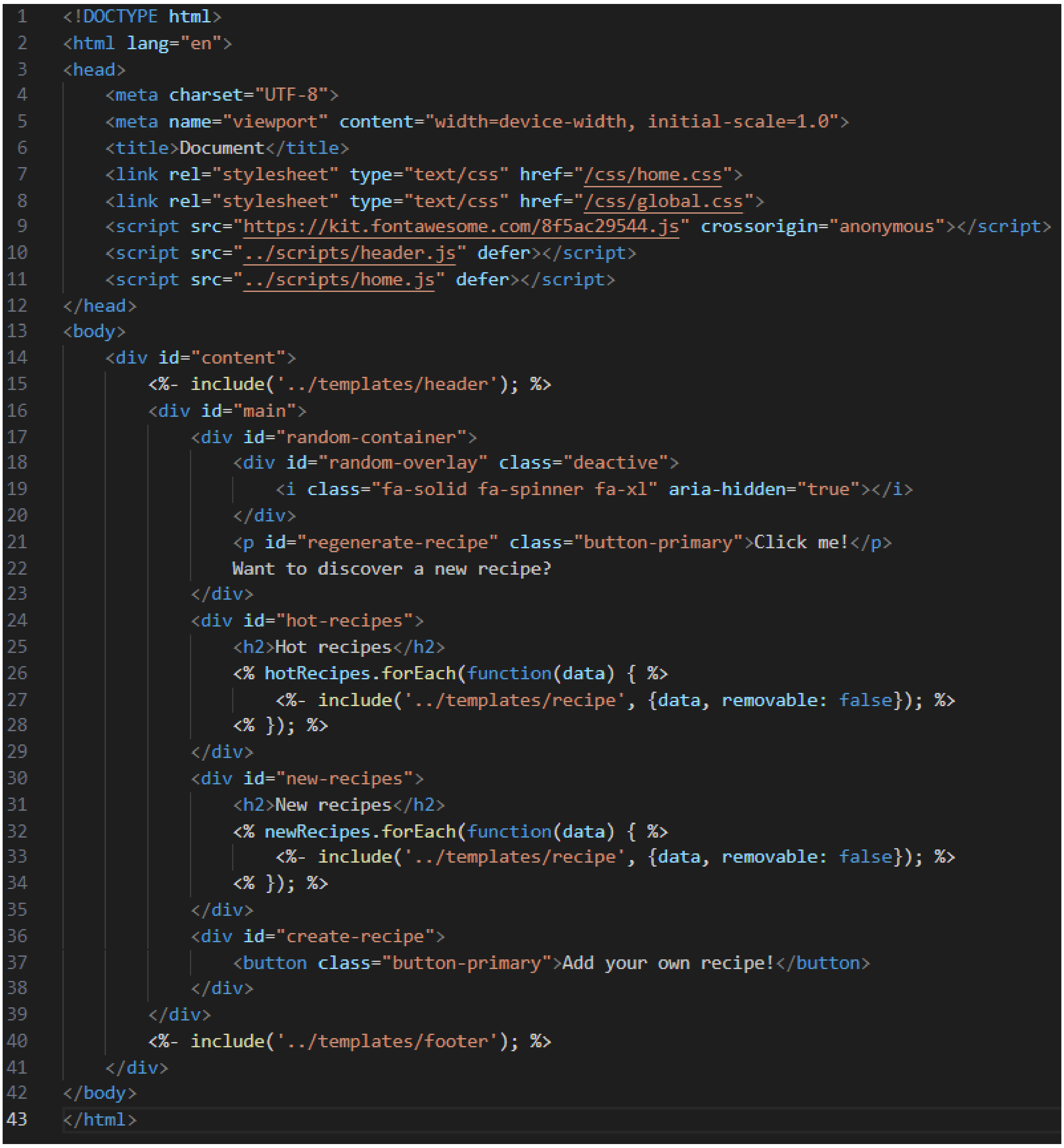
And now the last template, for recipes:



```
<div class="recipe-container" id="<%- data._id %>">
    <img class="recipe-photo" src=<% if (data.image) {%>
            "data:image/jpeg;base64,<%- data.image.toString("base64") -%>"
        <%} else {%>
            "images/placeholder.png"
        <% } %>
        alt="">
    <div class="recipe-data">
        <p><b><%- data.title -%></b></p>
        <p>
            <% for(var i=0; i<5; i++) {%>
                <% if(i < data.rating){ %>
                    ★
                <% } else {%>
                    ☆
                <% } %>
            <% } %>
            (<%- data.no_ratings -%>)
        </p>
        <p> Nutri-Score: <%- data.nutriscore -%>/5</p>
        <p><%- data.description -%></p>
    </div>
    <% if(data.no_people){ %>
        <div class="no_people">
            <i class="fa-solid fa-user fa-lg"></i>
            <%-data.no_people%>
        </div>

    <%}%>
    <% if(removable){ %>
        <i class="fa-solid fa-xmark fa-lg" style="margin:10px;"></i>
    <% } %>
</div>
```

This template does the same for the image, title and description but adds the nutriscore and the average review and number of reviews. For the review we loop five times and add a solid star if 'i' is smaller than the average rating and a hollow one otherwise. There are also two optional elements. If no_people is present in data, this is the case for the cart, we add that number with a person icon to the right of the container. If removable is true, this is the case inside profile where a user can delete it from their favourites, a cross is added. THe functionality of this cross is made through a separate script.

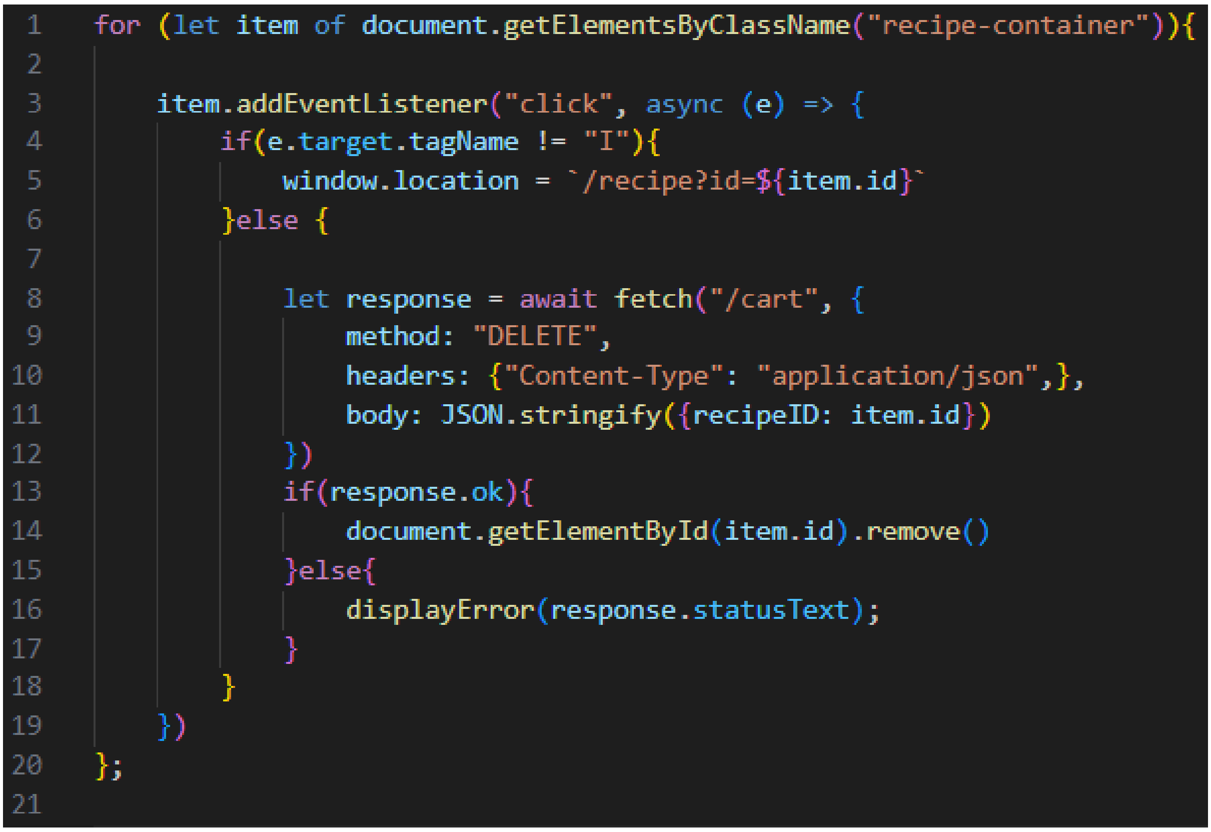
And finally, though we have a lot of HTML and javascript scripts, we will show one of each for the sake of avoiding this project report being obscenely long. For the HTML file, we will show our home file:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" type="text/css" href="/css/home.css">
    <link rel="stylesheet" type="text/css" href="/css/global.css">
    <script src="https://kit.fontawesome.com/8f5ac29544.js" crossorigin="anonymous"></script>
    <script src="../scripts/header.js" defer></script>
    <script src="../scripts/home.js" defer></script>
</head>
<body>
    <div id="content">
        <%- include('../templates/header'); %>
        <div id="main">
            <div id="random-container">
                <div id="random-overlay" class="deactive">
                    <i class="fa-solid fa-spinner fa-xl" aria-hidden="true"></i>
                </div>
                <p id="regenerate-recipe" class="button-primary">Click me!</p>
                Want to discover a new recipe?
            </div>
            <div id="hot-recipes">
                <h2>Hot recipes</h2>
                <% hotRecipes.forEach(function(data) { %>
                    <%- include('../templates/recipe', {data, removable: false}); %>
                <% }); %>
            </div>
            <div id="new-recipes">
                <h2>New recipes</h2>
                <% newRecipes.forEach(function(data) { %>
                    <%- include('../templates/recipe', {data, removable: false}); %>
                <% }); %>
            </div>
            <div id="create-recipe">
                <button class="button-primary">Add your own recipe!</button>
            </div>
        </div>
        <%- include('../templates/footer'); %>
    </div>
</body>
</html>
```

Every ejs file has the same head, where ‘home.css’ and ‘home.js’ are replaced by the correct css and js files. In the head we give the page a title and load the script for the icons we use. In the body we alway have a div with the id content and a div with the id main. Inside content before and after main we add the header and the footer using include. This is the same way we implement the template for recipes or leftovers. We can loop over the arrays for hotRecipes and newRecipes to render the recipes. This method is used throughout our project.

To finish off this section, we will explain our javascript script for the cart. We chose this particular script because it best demonstrates how we interact with our server from the front-end of the program. Here it is:



```
for (let item of document.getElementsByClassName("recipe-container")){

    item.addEventListener("click", async (e) => {
        if(e.target.tagName != "I"){
            window.location = `/recipe?id=${item.id}`
        }else {

            let response = await fetch("/cart", {
                method: "DELETE",
                headers: {"Content-Type": "application/json",},
                body: JSON.stringify({recipeID: item.id})
            })
            if(response.ok){
                document.getElementById(item.id).remove()
            }else{
                displayError(response.statusText);
            }
        }
    })
};

```

Here we give function to a recipe container for a recipe in our cart. We loop over all recipe containers and listen for a click event. We check if we press the cross or if we press something else by checking the tagName. If the tagName is ‘I’ it's the cross and we create a fetch request to our server. We do this by using the method “DELETE”, because we want to delete the recipe from our cart, set the headers so we can send data and add the recipeId to our body. If we get a positive response we delete the message. Otherwise we create a popup with the error. If we click on anything else from the recipe container we go to the actual recipe.

3.3 Design Justification

Our design is a good design because of several aspects of it; let us break it down. First, we have a lot of files, but it is very easy to understand where the code for every function of our program is contained with some background javascript knowledge. We placed an emphasis on organisation, especially because four people were working on the project concurrently; our views folder is, admittedly, somewhat chaotic, though. A point of improvement would be to separate it into one file for scripts and another for CSS; perhaps a different way of organising everything would make it even more easy to navigate. Overall, though, we are very content with the structure of our program; we have a lot of smaller, very connected files instead of huge, complicated and incomprehensible ones. Our use of routers and controllers was also largely for organisation's sake.

We used nodeJS for our back-end so the front-end and back-end use the same language. Not only does this make it easier for them to work together, but it makes coordinating our coding much more feasible. On this topic, we used plain javascript over reactJS (which is tailored to mobile applications) because of the simplicity and control regular javascript offers. We were also averse to using reactJS because it is component-based, meaning that we would have to make a lot of components for things we may only use once or twice. As an alternative, we used templates that are reused very frequently, effectively letting us get the best of both worlds.

The last design justification we would like to make is about mongoDB. Not only does it better suit javascript compared to SQL, but Tijn (the effective lead of this project group) is more familiar with it than with SQL. Additionally, it is faster than its alternative, making it viable for potentially upscaling our project if future programmers were to take it over.

4. Collaboration Platform and Repository Manager

The collaboration platforms we utilised were Gitlab and Google Drive. We used Gitlab to share our project files whereas Google Drive was used to collaborate on the project report, seeing as though it has a handy function where all of us could work on the same document simultaneously and see each other's changes in real time.

Gitlab enabled us to work on the project simultaneously, where everyone could edit a different file of code. After each session, we would submit our changes. This implemented everyone's changes in the shared code documents. Even though Git has a lot of functionality, we chose to specifically use it for managing our shared documents; this was so that everyone had access to the documents and could edit these files, all while updating the other group members on the changes in specific code files using the submission messages. Git helped us keep track of what still needed to be changed because of the submission messages. All group members could see what was currently being edited and improved, and which parts needed some attention.

5. Evaluation

Our group is reasonably satisfied with our product; if we had allocated a little more time towards the actual development of the product, and slightly less towards the planning of it, we could have improved upon a lot of smaller things and made quite a lot more of the quality of life changes we had initially planned to implement.

Our unsolved issues are less “issues”, and more limitations. For example, as we are very unfamiliar with working with live location data, the map page of Take-In ended up just displaying all leftovers in a list; this means that our application is not really localised yet, so we would only have to release our application in Nijmegen or any other one city for it to be used as intended. Another limitation that we are aware of is that conversations with other users are not updated live, meaning that users would have to refresh their page every time they wished to see a new message that was sent.

Regarding the development process, we believe that it would be better to adopt a less reliant team structure for future projects. As Tijn was the most familiar with the development platform and the tools we used to make Take-In, the rest of our group (myself, Jens and Ozgun) had to frequently consult him for advice. Granted, this gradually became less of an issue, but our early reliance on Tijn meant that the project would probably not have launched off if not for him.

This ties into the consequences of our organisational choices for future projects and how we would operate; we realised, after finishing the R&D project, that allotting time to learn an unfamiliar platform is extremely important and the amount of time required for this should never be underestimated. Everyone in our group has realised how much of a struggle having members who are unfamiliar with the development tools can be, and will very likely put more emphasis on the training process before even beginning to code in the future.

6. Appendix

6.1 Logbook

Do keep in mind that our logbook is not a perfectly accurate timeline of events, and that a lot of the work on the project was done more gradually over a period of time. Truthfully speaking, we often forgot to update it until about two weeks after finishing a pending task, which is why it is only a rough representation of our progress.

When	How long	Who	Achieved
		Week 1	
10-04-2024	2 hours	Jens, Aditya	Project plan has been started. The section 'Ideas' was finished and 'Technical plan' was started, we had a couple questions about this so we decided it would be better to leave it and continue the next TA session.
12-04-2024	2.5 hours	Tijn - Ozgun	Tijn has decided on the language we will use for the app and started helping Ozgun learn the language.
		Week 2	
15-04-2024	2 hours	Tijn	Ask the TA for advice about what the best way is to make the app. Start reviewing coding and other skills for making the app.
15-04-2024	2 hours	Jens Özgün	Finish up the project plan and finalise the software and language we will use.
15-04-2024	2 hours	Aditya	Make a start on the Project report, namely the Introduction and all sections of the Description. Using the information from the course RE&D.
19-04-2024			**Deadline project plan.**
		Week 3	
22-04-2024	2 hours	Tijn	Started on coding and making the app. Finished the setup for Git for the rest of the group to use it.

22-04-2024	2 hours	Ozgun - Aditya - Jens	Kept working on the Project report, asked Tijn for how they can be of any help on the coding part. Started reviewing various skills for making the app.
25-04-2024	1 hour	Ozgun - Jens	Review some HTML and CSS for the implementation of the project
Week 4 (Holiday)			
01-05-2024	4 hours	Jens - Ozgun	Started with learning HTML and CSS basics, got comfortable with those and continued on to a little bit more advanced topics.
03-05-2024	2 hours	Aditya - Tijn	Tijn kept working on the coding part with Aditya's help and Aditya submitted both his and Tijn's work on both Git and the Project Report.
04-05-2024	1 hour	Jens - Aditya	Revised the HTML and CSS topics and continued on implementing the project.
Week 5			
6-05-2024	3 hours	Jens - Ozgun - Aditya	Gained enough knowledge to help the rest of the group on coding. Finished a couple sections on the app.
8-05-2024	5 hours	Tijn	Focused on coding and managed to finish the ⅓ of the app already. Submitted his work on Git and Filled some sections in the Project Report.
8-05-2024	30 minutes	Jens - Aditya	Went through the project report and updated segments of section 2 to reflect the changes made by Tijn.
Week 6			
13-05-2024	3 hours	Jens - Ozgun	Tried to apply the gained HTML and CSS knowledge on the project, I managed to put up a picture and make sure it stays centred. Helped Adi and Tijn with minor adjustments.
17-05-2024 **And** **18-05-2024**	7 hours	Aditya - Tijn	Kept working on the implementation in JavaScript. Finished half of the project already. Submitted their progress on the coding part to Git.
Week 7			

27-05-2024	6 hours	Tijn	Finished approximately ⅔ of the coding, helped with the troubleshooting and implementation of minor adjustments. Really pulling his weight, honestly.
27-05-2024	4 hours	Jens - Aditya - Ozgun	Went through the project to do a lot of troubleshooting. Made some minor adjustments to eliminate most known bugs.
Week 8			
03-06-2024 And 05-06-2024	8 hours	Tijn - Ozgun	Managed to finish the implementation of the Project. Put what they did during the week in both Git and in the Project Report. We love Tijn.
06-06-2024	3 hours	Aditya - Jens	Nearly finished the Project Report. While Tijn and Ozgun were working on the implementation he also helped with minor/major adjustments and implementations.
07-06-2024	1.5 hours	Aditya - Jens	Last adjustments of the project report with the fully finished project.
07-06-2024	1.5 hours	Aditya - Tijn	Rewrote parts of section 3 of the project report, reflecting all changes and features adjusted in our code.
07-06-2024			**Final deadline**

6.2 Bibliography

1. Rowland, Michael Pellman. “5 Food Apps That Make It Easy to Eat Sustainably.” *Forbes*, Forbes Magazine, 14 Aug. 2017, www.forbes.com/sites/michaelpellmanrowland/2017/08/14/sustainable-food-apps/.
2. “Cloud Application Hosting for Developers.” *Render*, render.com/. Accessed 7 June 2024.
3. Soegaard, Mads. “The 10 Most Inspirational UI Examples in 2024.” *The Interaction Design Foundation*, Interaction Design Foundation, 7 June 2024, www.interaction-design.org/literature/article/ui-design-examples.
4. “Contrast Checker.” *WebAIM*, webaim.org/resources/contrastchecker/. Accessed 7 June 2024.
5. Santos, Soon Sam. “NodeJS API-Part 5 / Model/Router/Controller Structure.” *Medium*, Medium, 17 Jan. 2021,

soonsantos.medium.com/nodejs-api-part-5-model-router-controller-structure-c5b13c2660ae.