Department of Computer Engineering

Bilkent University

Senior Design Project

Foodster: Make maintaining your diet easier

Final Report

Khasmamad Shabanovi, Gledis Zeneli, Balaj Saleem, Ibrahim Elmas, Perman Atayev

Supervisor: Ozcan Ozturk
Jury Members: Dr. Çiğdem Gündüz Demir, Dr. Can Alkan
Innovation Expert: Haluk Altunel

1. Introduction

Food is the first of all fundamental human needs, yet how many times today have you, personally, consciously thought of what you have and will be eating today, what its nutritional value is and how it fits into your greater nutritional, fitness, and lifestyle goals? With the ease of falling into a routine and the abundance of staple food, keeping track of all these variables, and making sure that you plan your meals such that your priorities are heeded, becomes increasingly mundane, monotonous, and just unnecessary extra work.

With the fast-paced lifestyle of the 21st century, monotony and unnecessary work are the last things an individual needs in his/her busy life. Figuring out what to eat, to order or to cook, how to cook it, where to get the ingredients from, and how beneficial this meal would be for your body are questions that would take precious time and energy that could be better employed elsewhere.

Furthermore, the complications of searching for the nutritional data, planning a healthy diet according to one's needs, planning the budget for such a diet, and finding recipes to support this time is truly a cumbersome endeavor.

This is where Foodster comes in.

2. Requirements Details

This section discusses project requirements details in two sections: functional and nonfunctional requirements.

2.1. Functional Requirements

User is able to

- register using an email and a password
- verify her account via email verification
- login using her email and password
- enter her weight, height, age, and gender
- enter the undesired ingredients (allergies)
- enter her preferred calorie, protein content, fat content, and carbohydrate content range
- enter her preferred diet type (vegetarian, vega, paleo or keto)
- enter her preferred price range
- select several meals out of a given list of 10 meals initially to receive recommendations with similar meals
- update her personal information and preferences
- generate meal plans for a specified number of days, maximum 7 days, according to her preferences (calorie range, undesired ingredients, price range etc.)
- select the desired number of meals per each day for meal plan generation
- view the list of generated meals per day
- view the recipes of each meal, where a recipe includes
    - title
    - picture
    - serving size
    - list of ingredients together with their amounts and units of measurement
    - list of instructions
- see an estimate of the price of a meal
- view the nutrition information regarding a meal, which includes
    - calories
    - protein amount
    - carbohydrate amount
    - fat amount
- like a meal

2.2. Non-functional requirements

2.2.1. Accessibility

- The system will require Android Jelly Bean 4.1.x or newer and iOS 8 or newer because we will be using Flutter library to build applications for both Android and iOS. We will use Node.js for building the web-server of the application that will be easy to send requests to from Flutter applications.

2.2.2. Accuracy

- If the preferences of users are available for a user, they should be satisfied as much as possible. Especially the preferences that could affect the health condition of a user such as allergies must be considered.

2.2.3. Availability

- Our initial audience will be Turkey; therefore, any down-time for updating databases, fixing some critical bugs should be happening at night in Turkey, so that the least amount of users are affected by the down-time of the application that should not exceed more than 1 hour.

- Since the system is highly modular, the whole system should not be down due to an update to a particular submodule.

2.2.4. Backup and Recovery

- Firebase is going to take care of Backup and Recovery of data, since that is the database we are going to use.

- The preferences of users should be stored both locally and globally, so that no data is lost.

2.2.5. Capacity

- The server should have satisfactory computation power and storage for each user.

- The server should handle at least 10000 registered users from Turkey.

2.2.6. Compatibility

- Libraries used in Flutter should not conflict with any Android or iOS phones that support installation of the application.

- The users should have Android or iOS phone devices to support the application.

2.2.7. Concurrency

- We use Heroku to deploy our back end service which will be available 24/7. Heroku will support 5-10K requests per month, which will be more than enough to keep the response time of a user under 1s.

2.2.8. Configurability

- The user should be able to update any data that has to do with his / her payment methods, preferences, body characteristics such height, weight and any other information that user has access to.

2.2.9. Exception Handling

- In case of exception/error, error must be realized as much as possible. The error must be

clearly explained to the user clearly and also required steps must be explained, so that the user knows what to do and how to do.

- In case of unexpected errors, the users should be able to share this error with us via crash log or core dump.

2.2.10. Extensibility

- It must be easy to develop new features and add new functionality into the application for future business needs. So, logical separation of the application will be maintained so that application will be divided into different tiers(e.g. client, presentation, business logic, etc.)

2.2.11. Legal and Regulatory Requirements

- The users will be warned that they are responsible for law-violating actions (e.g. copying licensed content, etc.) that are taken by them.

2.2.12. Licensing

- Required licenses for the libraries, services and modules used during development will be arranged

2.2.13. Maintainability

- Subsystems will be loosely coupled by means of the logical separation we will maintain. So, a modification or integration to a subsystem/module will not affect others.

2.2.14. Performance

- Meal recommendations should not take more than 10 seconds.

2.2.15. Reliability

- The application should not crash at any time due to software domained error.
- The server must be running the whole time other than the maintenance time that is 1-hour once in a month during nights in Turkey.
- Crash logs will be stored in Firebase to be inspected and analyzed to avoid further crashes.

2.2.16. Scalability

- The system will be designed so that scaling the system will be easy by choosing the right technologies for web-server, database type, network etc.
- The database must be able to an annual growth rate of 20%, with no decrease in database performance
- The web-servers must be able to support an annual growth of 10% of new customers.

2.2.17. Security

- The users must log in with their private credentials.
- The web-server must be available and behave reliably even under DOS attacks..
- The application must provide the integrity of the customer’s private information.

2.2.18. Testing

- The web-server, database and mobile application will be tested regularly. Some of test types will be made:

- Reliability tests will be made for web-server to function without failure,
- Load tests will be made for the web-server to measure performance based on actual customer behavior.

2.2.19. Usability

- The GUI will be intuitive and user friendly such that it will not restrict any functionality and user interactions will be easy.
- The users will not need to spend more than 10 seconds to learn the functionalities of the screens.
- The users will be able to submit their feedback to developers.

3. Final Architecture and Design Details

3.1. Subsystem Decomposition

As we have mentioned in our previous reports we chose to go with Client Server Architecture for the Foodster. The whole decomposition of our system can be found below.

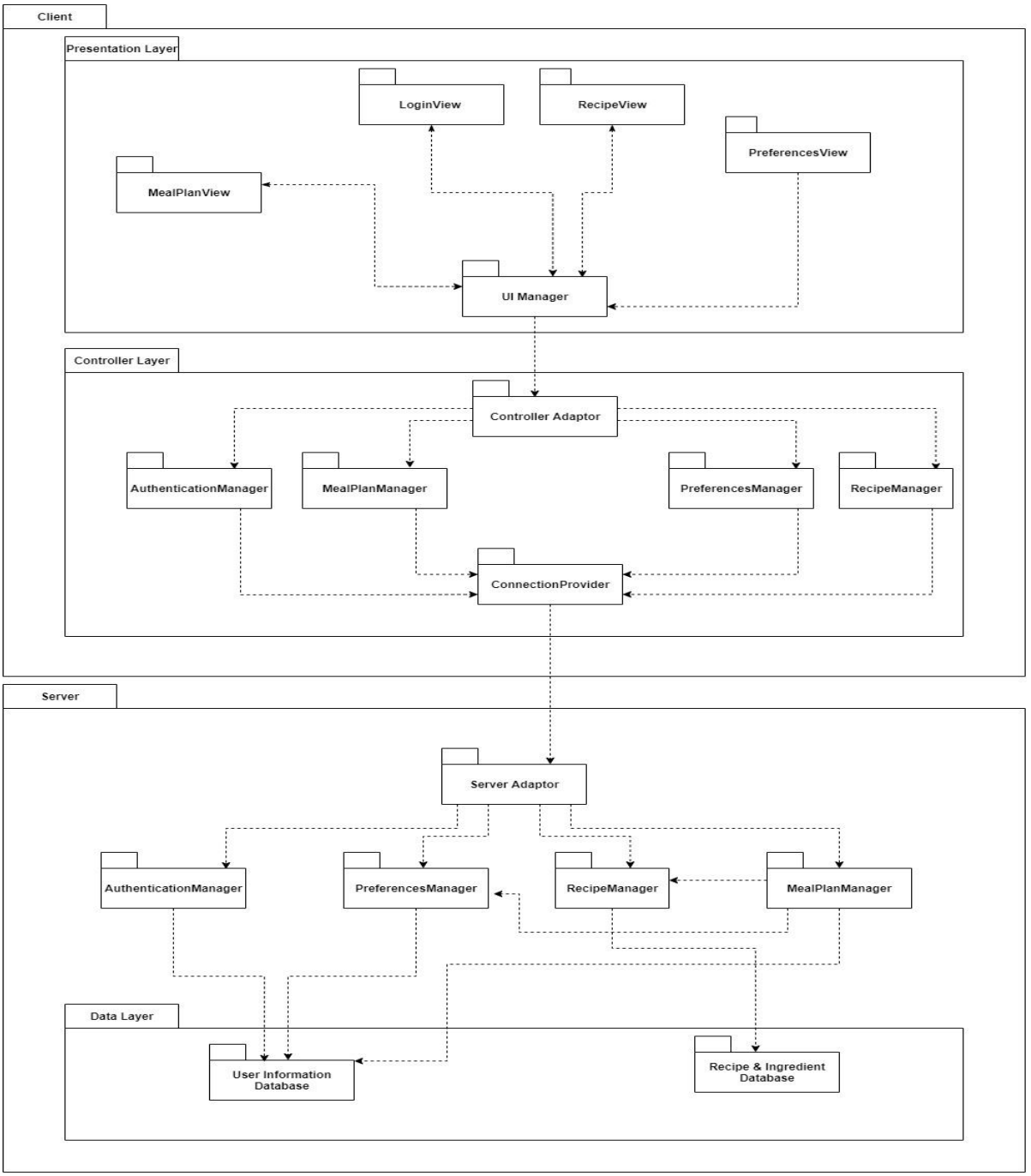


Figure 1: Subsystem decomposition

3.1.1. Client Side

We further decided to use MVC architecture on the Client side. You can find the details of each component of MVC below.

- **Model**

  The Model component of the client consists of all the classes that store information regarding the real life entities of the dieting process. You can find the details of the Model diagram below.

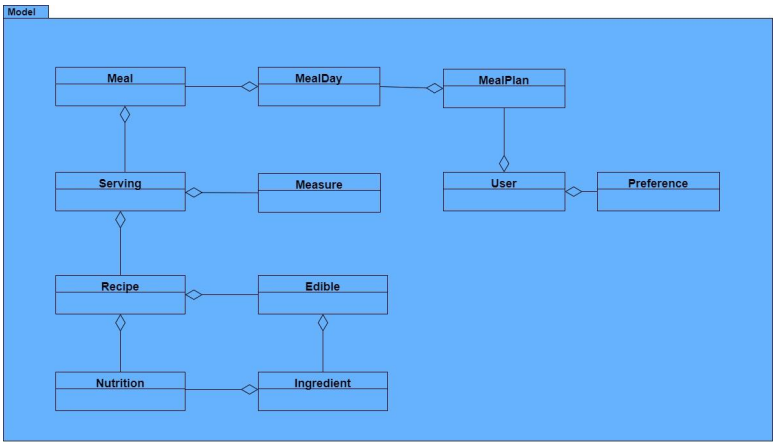


Figure 2: Model component

- **View**

  The View component of the client consists of all the classes which directly contribute to the UI.

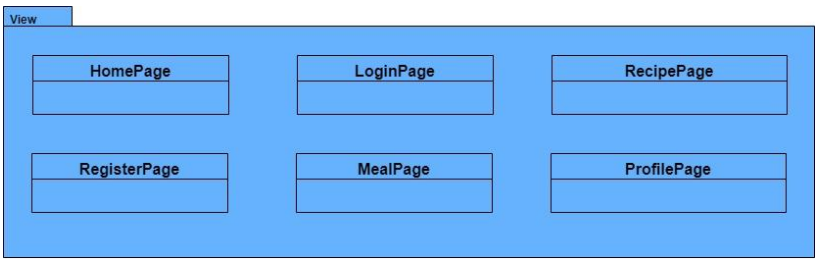


Figure 3: View component

- **Controller**

  The Controller component of the Client is composed of classes which regulate communication between the View and Model of the Client, but also between the Client and the Server.

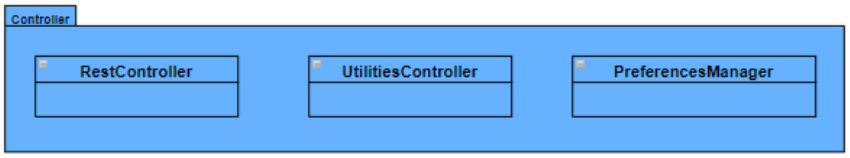


Figure 4: Controller component

3.1.2. Server Side

Server side of our application has three components: router layer, logic layer, and data layer. This section presents detailed information on these components. Technically we could also call

the data layer a data tier because it is on a separate machine, but since the logic layer and the router layer are located on the same machine, we just went with the layer convention.

- **Router Layer**

  Route layer serves as the router interface of the Server Side. All the requests that come from the client side have a designated component in the Logic Layer that will take care of the request and will provide the corresponding response. Route Layer will route requests to these components in the Logic Layer.

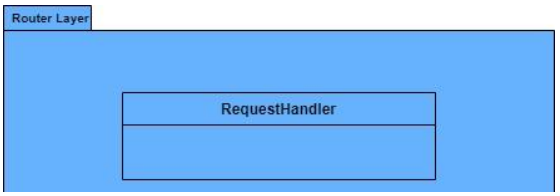


Figure 5: Route layer

- **Logic Layer**

  Logic Layer will take care of the request that was routed by the Router Layer. For example if the signup request came to the server, then it would take all the necessary actions for the signup and then communicate to the Data Layer to create certain models such as the User Model if the signup is requested.

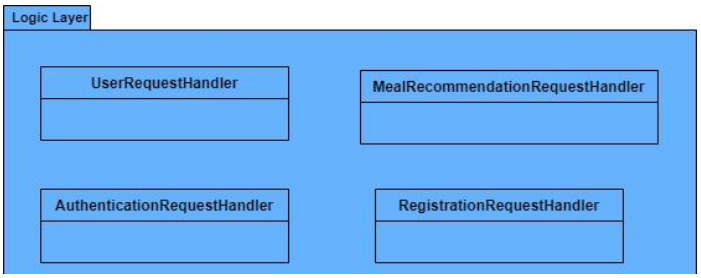


Figure 6: Logic layer

- **Data Layer**

  Data Layer takes care of mapping the information that comes from the user to models that would be stored in the database that is connected to the server. The data that comes to the user is propagated from the Logic Layer to the Data Layer. The Data Layer of the Server mirrors the Model of the Client Side.

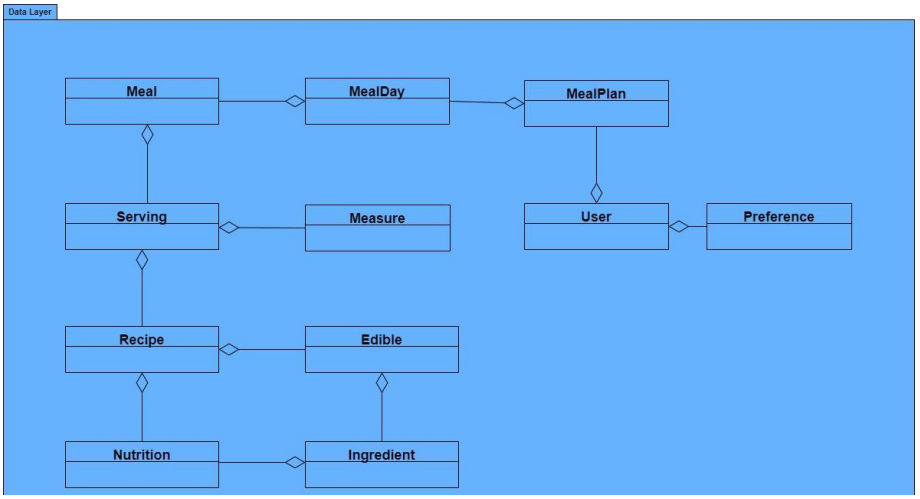


Figure 7: Logic layer

3.2. Class Interfaces

3.2.1. Client Side

- **View**

HomePage	
HomePage is class that contains the widget the displays the landing page for logged in user	
Attributes	
private int selectedNavIndex	
Methods	
public void initState()	Initializes the (state) variables of the class
public void build(BuildContext context)	Rebuilds the widgets / UI classes whenever the state is updated in the current context.
public void buildAppBar(BuildContext context))	Builds the app bar which contains the header
public void buildBottomNavigationBar(BuildContext context))	Builds the bottom navigation bar to move to different tabs
public void onNavItemTapped()	performs an action when the bottom navigation bar item is tapped

RegisterPage	
RegisterPage is class / widget the displays the sign up page for a user	
Attributes	
private String email	
private String password	
Methods	
public void initState()	Initializes the (state) variables of the

	class
public void build(BuildContext context)	Rebuilds the widgets / UI classes whenever the state is updated in the current context.
public void handleRegister(BuildContext context))	Validates and handles registration functionality.

LoginPage
Login is class / widget the displays the sign in page for a user
Attributes

private String email
private String password
private boolean isLoading

Methods

public void initState()	Initializes the (state) variables of the class
public void build(BuildContext context)	Rebuilds the widgets / UI classes whenever the state is updated in the current context.

MealPage
MealPage is class that contains the widget the displays the meal plan page to a logged in user
Attributes

private MealPlan mealPlan

Methods

public void initState()	Initializes the (state) variables of the class
public void build(BuildContext context)	Rebuilds the widgets / UI classes whenever the state is updated in the

	current context.
public void buildDatePicker(BuildContext context))	Builds the datepicker to select a date for meals
private void handleMealGeneration()	Handles the fetching of mealPlan

RecipePage
RecipePage is class that contains the widget the displays the meal plan page to a logged in user
Attributes
private Recipe recipe
Methods

public void initState()	Initializes the (state) variables of the class
public void build(BuildContext context)	Rebuilds the widgets / UI classes whenever the state is updated in the current context.
public void buildHeader()	Builds the header for recipe page
private void buildIngredients()	Builds the ingredients list for the recipe

ProfilePage
ProfilePage is class that contains the widget the displays the user profile to a logged in user
Attributes
private User user
private Preferences preferences
Methods

public void initState()	Initializes the (state) variables of the class
public void build(BuildContext context)	Rebuilds the widgets / UI classes whenever the state is updated in the current context.
public void buildHeader()	Builds the header for the user details
public void buildUserDetails()	Builds the user details section of the page
public void buildPreferences()	Builds the user preferences section of the page

- **Controller**

RestController
This controller is responsible for Restful API interactions of the mobile app
Attributes
private String baseUrl
Methods

public static String token signup(User user)	Handles the restful api call to signup
public static String token login(String email, String password)	Handles the restful api call to log in
public static MealPlan getMeals(String token)	Handles the restful api call to get meals for a user
public static User getUserDetails(String token)	Handles the restful api call to get all user details
public static Preferences getPreferences(String token)	Handles the restful api call to get user preferences
public static User updateUserDetails(String token, User user)	Handles the restful api call to update user details
public static Recipe getRecipe(String token)	Handles the restful api call to get a unique recipe
public static boolean logMeals(String token, Nutrition nutrition)	Handles the restful api call to log meals

UtilitiesController
This class contains basic warning / message utilities for the app
Attributes
Methods

public void showToast(BuildContext context)	Shows a toast for a specific UI context
public void showSnackbar(BuildContext context)	Shows a snackbar for a specific UI context
public String uppercaseText(String s)	Converts a string to uppercase

PreferencesManager
This class handles shared preferences for a session
Attributes

private String token

Methods

public String initialize()	Initializes the manager to store shared data
public boolean storeToken(String token)	Stores a token for the user in session
public String getTokent()	Gets the token for the user in session
public boolean removeToken()	Removes the token for the user in session

- **Model**

Recipe
This class is responsible for holding all the relevant information of a recipe.

Attributes

private String name
private int prepTime
private int cookTime
private String imgUrl
private Array<String> instructions
private Nutrition nutrition
private double estimatedPrice
private Array<Edible> ingredients
private Array<String> tags

Methods

public static Recipe fromJSON(String json)	Returns a Recipe object from a json string.
public static String toJSON(Recipe recipe)	Returns the json representation of a Recipe object
public Nutrition getScaledNutrition(Measure serving)	Returns a Nutrition object holding data for an amount of serving of that recipe.
public Array<Edible> getScaledIngredients(Measure serving)	Returns the ingredients required to cook an amount of servings for the recipe.
getters and setters	

Ingredient

Measures are responsible for holding information relevant to a real life cooking ingredient.

Attributes

private String name
private String imgUrl
private Nutrition nutrition
private double estimatedPrice

Methods	
public static Ingredient fromJSON(String json)	Returns an Ingredient object from a string json.
public static String toJSON(Ingredient ingr)	Returns the json representation of an Ingredient object.
public Nutrition getScaledNutrition(Measure measure)	Returns the nutritions of the Ingredient object for a measure as a Nutrition object.
getters and setters	

Measure
Measure is a class responsible for unit management. Different ingredients and food components are measured in different units and this class handles all the relevant conversion and information storage.

Attributes
private double magnitude
private String unit

Methods	
constructor(double mag, String unit)	Constructs a Measure object with given magnitude and unit.
public void convert(String newUnit)	Converts the objects unit to a new unit
getters and setters	

Serving
Serving is a class which holds information about a recipe and what amount of that recipe is cooked.

Attributes
private Recipe recipe
private Measure measure

Methods	
public static Serving fromJSON(String json)	Returns a Serving object from its json representation.
public static String toJSON(Serving serving)	Returns the json representation as a string of a Serving object.

Edible
Edible is a class which holds information about an ingredient and what amount of that ingredient is used.

Attributes
private Ingredient ingredient
private Measure measure
private String description

Methods	
public static Edible fromJSON(String json)	Returns an Edible object from its json representation.
public static String toJSON(Edible edible)	Returns the json representation as a string of an Edible object.

Meal
Meal is a collection of Serving objects which would represent a multi dish meal in real life.

Attributes
private String name
private Array<Serving> servings

Methods

public static Meal fromJSON(String)	Returns a Meal object from its json representation.
public static String toJSON(Meal meal)	Returns the json representation as a String of a Meal object.

MealDay
MealDay is a collection of all the meals planned for consumption on a particular day.
Attributes

private Date date
private Array<Meal> meals

Methods

public static MealDay fromJSON(String)	Returns a MealDay object from its json representation.
public static String toJSON(MealDay mealDay)	Returns the json representation as a String of a MealDay object.

MealPlan
MealPlan is a collection of all the daily meal plans, planned for consumption on a multi day time duration.
Attributes

private Date startDate
private Date endDate
private Array<MealDay> plan

Methods

public static MealPlan fromJSON(String)	Returns a MealPlan object from its json representation.

public static String toJSON(MealPlan mealPlan)	Returns the json representation as a String of a MealPlan object.
public MealDay getMealDay(Date date)	Returns the planned MealDay object for a certain date.
public void setMealDay(MealDay mealDay, Date date)	Set the planned MealDay object for a certain date.
getters and setters	

User

The user class holds relevant information needed to produce recommendations for a user, and some user account related information.

Attributes

private String username
private String email
private ENUM gender
private double height
private double weight
private String profileImage
private Array<String> allergies
private Preference preferences
private Array<Recipe> likedRecipes
private Array<Ingredient> likedIngredients

Methods

public static User fromJSON(String)	Returns a User object from its json representation.
public static String toJSON(User user)	Returns the json representation as a String of a User object.
public void addPreference(Preference newPreference)	Adds a new Preference to the list of user preferences
public void rmPreference(Preference preference)	Removes a Preferences from the list of user preferences.

public void addAllergy(Ingredient ingr)	Adds an ingredient to the list of user allergies.
public void rmAllergy(Ingredient ingr)	Removes an ingredient from the list of user allergies.
public void addLikeRecipe(Recipe likedRecipe)	Adds a recipe to the list of liked recipes.
public void rmLikedRecipe(Recipe recipe)	Removes a recipe from the list of liked recipes.
public void addDislikedRecipe(Recipe dislikedRecipe)	Adds a recipe to the list of disliked recipes.
public void rmDislikedRecipe(Recipe recipe)	Removes a recipe for the list of disliked recipes.
public void addLikedIngredient(Ingredient ingredient)	Adds an ingredient to the list of liked ingredients
public void rmLikedIngredient(Ingredient ingredient)	Removes an ingredient from the list of liked ingredients.
public void addDislikedIngredient(Ingredient ingredient)	Adds an ingredient in the list of disliked ingredients.
public void rmDislikedIngredient(Ingredient ingredient)	Removes an ingredient from the list of disliked ingredients.
getters and setters	

Nutrition
Nutrition holds macro and micro nutrient information.
Attributes

private Measure calories
private Measure carbs
private Measure proteins
private Measure fats
private Map<String, Measure> micros

Methods

public static Nutrition fromJSON(String)	Returns a Nutrition object from its json representation.
public static String toJSON(Nutrition nutrition)	Returns the json representation as a String of a Nutrition object.
getters and setters	

Preference
Preference is responsible for grouping filters that the user would typically apply while generating meals or meal plans.
Attributes

private int mealsPerDay
private int mealPlanDuration
private Pair<int, int> calRange
private Pair<int, int> fatRange
private Pair<int, int> carbRange
private Pair<int, int> protRange
private Pair<int, int> costRange
private Pair<int, int> cookingTime
private String dietType

Methods

public static Preference fromJSON(String)	Returns a Preference object from its json representation.
public static String toJSON(Preference preference)	Returns the json representation as a String of a Preference object.

3.2.2. Server Side

- Router Layer

RequestHandler	
This class is responsible for handling requests and rerouting them to a relevant request handler if possible.	
Methods	
public boolean handleRequest(String url, Request req)	Sends the request to a relevant request handler. If unsuccessful, which means that the request is not relevant to any request handler, then it returns false and sends back the corresponding response.

- Logic Layer

UserRequestHandler	
This class is responsible for handling user related requests. The handler methods of this class return true if successful, false otherwise. Request body that is provided to requests will contain all the relevant information that is needed by the method. Arguments to the methods (data) are passed inside the request body. For every request regarding a user a token is expected.	
Methods	
public boolean fetchUserInforHandler(String url, Request req	Fetches user related information.
public boolean updateUsernameHandler(String url, Request req)	Updates the username
public boolean updateHeightHandler(String url, Request req)	Updates the height of the user.
public boolean updateWeightHandler(String url, Request req)	Updates the weight of the user.
public boolean addAllergyHandler(String url, Request req)	Adds a new allergy type to the list of allergies of the user.
public boolean removeAllergyHandler(String url, Request req)	Removes an allergy type from the list of the allergies of the user.
public boolean updatePreferences(String url, Request req)	Updates the preferences of the user
public boolean likeRecipeHandler(String url, Request req)	Adds a recipe to the user's list of liked recipes

public boolean unlikeRecipeHandler(String url, Request req)	Removes a recipe from the user's list of liked recipes.
public boolean dislikeRecipeHandler(String url, Request req)	Adds a recipe to the user's list of disliked recipes.
public boolean undislikeRecipeHandler(String url, Request req)	Adds a recipe to the user's list of disliked recipes.
public boolean likeIngredientHandler(String url, Request req)	Adds an ingredient to the user's list of liked ingredients
public boolean unlikeIngredientHandler(String url, Request req)	Removes an ingredient from the user's list of liked ingredients.
public boolean dislikeIngredientHandler(String url, Request req)	Adds an ingredient to the user's list of disliked ingredients.
public boolean undislikeIngredientHandler(String url, Request req)	Adds an ingredient to the user's list of disliked ingredients.

AuthenticationRequestHandler
This class is responsible for the authentication of the related request. The handler methods of this class return true if successful, false otherwise. Arguments to the methods (data) are passed inside the request body.
Methods

public boolean authenticationHandler(String url, Request req)	This method will check whether a user exists in the system and if so it will return a token for a user so that he can use that token as an identity in the subsequent requests.

RegistrationRequestHandler
This class is going to handle all the requests to the server related to registration. Arguments to the methods (data) are passed inside the request body.
Methods

public boolean registration(String url, Request req)	This method is going to register a user if all mandatory fields such as email and password are provided. If the email is not a real email or if the password is not strong enough the registration request will be rejected and the method will return false. Otherwise the user will be successfully registered in the Foodster.

MealRecommendationRequestHandler
This class is responsible for recommending meals to users taking into consideration the history of meals that users liked and other preferences that they provided to the Foodster such as their diet types, amount of calories they want to take in, their budget et cetera. Arguments to the methods (data) are passed inside the request body.
Methods

public boolean recommendMealFromHistoryHandler(String url, Request req)	This conservative meal recommender method recommends meals to Users looking at the meals that they liked in the past, and how the meals that are planning to be recommended are similar to the meals they liked. The similarity of meals will be calculated using mathematics such as Euclidean distance of meals' ingredients and scalar product of ingredients of the meal.
public boolean recommendMealRandomHandler(String url, Request req)	This non conservative meal recommender method recommends meals to Users randomly from the pool of meals that are allowed to the User considering User's allergies and diet types. However, the history of meals will not be considered for this method, because we want a User to try something new that he might potentially like and at the same time that he would not try himself.

RecipeFetcher
The class that manages recipe operations. The request body for these methods assumed to have all relevant information for the method.
Methods:

public boolean getInstructions(String url, Request req)	Makes a database query to get the instruction details of the given recipe in the request.
public boolean getImagePath(String url, Request req)	Makes a database query to get the image path in the storage database of the recipe in the request.
public boolean getLikingUsers(String url, Request req)	Makes a database query to get the list of the users who liked the recipe in the request.
public boolean getRecipeWithNutritions(String url, Request req)	Finds a recipe with given nutrition constraints in the request.
public boolean getRecipe(String url, Request req)	Sets all attributes of the given recipe by making a database query with given recipe id in the request.

- Data Layer
  Data tier classes are the same with the client side model classes.

4. Development/Implementation Details

4.1. Server side

4.1.1. Technologies used

- Nodejs
  JavaScript runtime we used to be able to build a server that can run out of the browser. We wanted this server to run on the cloud whether it is Heroku or Amazon EC2.
- Express
  We used Express framework to be able to quickly setup the server functionalities on top of the Nodejs. Express facilitates creating endpoints on the server, parsing the incoming requests and comes with multiple builtin functionalities that make it faster to develop.
- Git / Github
  We used Git / GitHub to be able to collaborate on the project quickly. We had 2 branches for the project, development and master branches. Development branch was our default branch and we would use it until we got a stable running version of our application. Our master branch was connected in such a way, so that anything that is pushed to the branch would trigger the deployment of the project to the cloud.
- Heroku
  We mainly used Heroku for the deployment of our project, so that the front end team could easily access the API without having to run it on their local machines.
- Docker
  We set up a docker file in case we want to switch from Heroku to Amazon EC2 or Digital Ocean. We prepared a container for our application so that we can easily run it on any remote machine and start it up as fast as possible.
- Sendgrid
  We used Sendgrid mailing service to be able to send mails to people when they sign up, to make sure that no one spams the servers with emails that they don’t possess.

- Amazon S3
  We used Amazon S3 images as backup images for the Recipe images we have. Just in case something happens to the original image in AllRecipes or from wherever else we scrape the recipes from, we can easily switch to using Amazon S3 storage's version of the same image.
- Swagger
  We used Swagger to be able to create a nicely formatted documentation. This documentation was mainly created so that the Front End team can easily find all the endpoints, required input for those endpoints and what will the format of the output be from those endpoints.
- MongoDB
  MongoDB is a NoSQL database that we used to store all the user information, all their preferences, recipe information, recommended meals information and pretty much anything that needed to be persisted.
- Morgan
  Morgan is a logging library that we used to be able to debug the server crash in case it happened. Morgan logs crucial information of each request like when it was sent, what endpoint it called, what is the time it took for the response and the code of the response.
- Jest + Supertest
  Jest + Supertest are used to create unit tests and integration tests and also to automate running those tests, so that we don't have to run tests one by one.

4.1.2. Development details

- Server of the application was mainly developed by 2 people. They would meet every week to update each other on what kind of features need to be implemented and what features have problems if they have any.
- Generally Git and GitHub was used to keep the code up to date.
- GitHub's issues were used to track what are the problems that needed to be fixed and the progress that was done regarding those issues.
- Any code that was written by one teammate would be carefully studied for potential problems by the other teammate to improve the quality of the code.
- Brainstorming sessions would be done if teammates could not figure out the solution to a certain problem by research.

4.2. Client Side

4.2.1. Technologies used

- Flutter
- Git / GitHub

4.2.2. Development Details

- The development was mainly done by two members of the teams.
- Weekly Brainstorming sessions as well as the updating sessions took place to make sure that the schedule was being followed and no major problem stayed unsolved.
- Git / GitHub was used to keep the code up to date for any teammate that wants or needs to run the application.

4.3. Data Scraping

We scraped two different types of data for distinct purposes. The first is the data regarding the recipes. The second is the data from Migros that was used for price estimation. This section articulates the development and implementation details of scraping both types of data.

4.3.1. Recipe data

We scraped more than 2000 recipes from Allrecipes.com. This process was divided into 3 parts. In the first part we gathered a list of recipe urls. Then, in the second part, we processed the contents of the recipe webpages that were retrieved using the urls gathered in the first part. Lastly, we trained and ran a Named Entity Recognition (NER) model on extracted ingredient phrases to identify individual words in these phrases. The whole process was done in Python. The code can be accessed here.

- **Gathering the list of recipe urls**

  The first step was to gather the list of recipe urls from Allrecipes.com. The website provides multiple categories of recipes each including a myriad of recipes. The following table shows the information regarding what categories we gathered our recipes from and the number of recipes per category. In the subsequent steps, duplicate recipes and recipes with missing information were filtered out. The third column of the table shows the number of recipes after filtration.

  Table 1. Number of recipes per category

Category name	**# of recipes**	**# of recipes (post filtration)**
Breakfast and brunch	1000	181
Lunch	1000	225
Dinner	1000	275
Appetizer & Snack	1000	332
Main Dish	988	55
Salad	1000	93
Side dish	1000	420
Turkish	19	6
Vegetarian	1000	175
Keto	458	127
Paleo	1000	159
Vegan	1000	217

  Entering the main page of a specific category does not show the list of recipes. Instead, the main page of a category shows a limited number of recipes and more recipes can be loaded dynamically. After several trials and errors, we realized that the main page of a recipe category corresponds to only the first page of that category and the next pages have a neat list of recipes, which can be scraped easily as no dynamic loading is required. To access the next pages, we added a GET request parameter to the recipe category urls that indicates the page number. E.g. if a category url is "…/lunch/", we send a get request to "…/lunch/?page=i" where i is replaced with the page number. Then, we collected all the individual recipe page urls from the pages. We used Python's requests library to retrieve the html content of the pages.

- **Processing the recipe page content**
  After retrieving the html content of the pages, we used the BeautifulSoup to parse the content. The information stored regarding each recipe includes url, title, site name (e.g. allrecipes), image url, preparation time, cook time, serving size, ingredients, nutrients, instructions, rating, tags (e.g. salad, vegan etc.), and cuisines (e.g. turkish). Refer to this document for more detailed information.

- **NER**
  To be able to account for allergies or undesired ingredients of a user and do ingredient-based recipe similarity calculation, we need to be able to tell which ingredients are part of a recipe. However, in recipe websites, including allrecipes.com, the ingredients are given as a phrase such as "1 (5 ounce) can tuna, drained and flaked." What we need is to classify "1" and "5" as the quantities, "ounce" and "can" as the units of measurements, and "tuna" as the name of the ingredient. For this task, we trained a Named Entity Recognition (NER) model. An NER model does exactly what we want: it classifies the words in a given sentence into different classes that it was trained on. We used Python's spaCy library to train our models. The training data can be accessed here. Since the training data in its original format is not compatible with the requirements of spaCy, we had to preprocess the data using a Python script. Our model, trained on the preprocessed data, had an overall accuracy over 97%.

4.3.2. Migros data

About 5000 items from Migros were scraped with their titles, weights in grams, and prices. Brand names, units, magnitudes, special characters, uppercase letters were removed from the items' titles to normalize them for matching them with ingredients from our database. The ingredients from our database were cleaned from non-noun parts of text using a part-of-speech tagger, so they are normalized for matching. Later, string matching was done by fitting and transforming our database data on a tf-idf text vectorizer. Migros data was transformed using this vectorizer, and the closeness of the vectors was calculated using cosine difference. Whenever multiple migros ingredients were equally close in vectorized form to our recipe db ingredient, a heuristic was used to choose the migros ingredient with as close number of words as to our database ingredient.

Due to the data being bilingual, matching was done through both translating our ingredients to Turkish and doing the matching fully in Turkish, and translating the migros data to English and doing the matching in English as well. For every ingredient the match with the highest degree of confidence was chosen from both languages.

Then, all predictions with accuracy less than .5 were discarded (intuitively, .5 accuracy corresponds to about half of the phrase of the ingredient matching with the migros ingredient).

This technique matched ~17k total ingredients (with repetition) from our database. These resulted in ~2000 recipes with a relatively high confidence on price estimation and ~500 recipes from which we did not know the price of only 1 ingredient. Estimation techniques can be used to estimate the price of the lacking ingredient, such as the average price of other known ingredients. Guessing a single ingredient's price from 4-6 other ingredients is not always accurate (when the not known ingredient is expensive) but on many occasions (when the ingredient is relatively cheap), the estimation holds a relevant value.

5. Testing Details

5.1. Unit Tests

To make our development faster in the long term and also to improve the quality of code we decided to write unit tests as we developed our code. We used the Jest library as a framework to set up and run our unit tests. Although writing unit tests did take us time in the beginning, it made the development process faster in the long run because every time we changed some code or added some functionality, by running the Unit tests we had we could easily check whether anything that was already working got broken or worked properly. It also saved us from some bugs that we would not have noticed otherwise. On several instances before deploying the product to the cloud, we noticed some small bugs with Unit tests we wrote and thus could remove the bugs before they were in production. Also, in case any of our teammates that are testing the deployed product found a bug that we could not catch with our unit tests, then we would go back and update our unit tests to make sure that we don't miss that family of bugs in the future.

5.2. Integration Tests

To make sure that the modules we developed work well together, we also developed integration tests. Also, another reason why we developed integration tests is that some of the bugs that we encountered could not be really caught with unit tests. Because those bugs did not have to do with one functionality that we implemented, it had to do with one module using the other module's interface in an unexpected way. We would run our integration tests in two instances: after finishing development or update of a module and before deploying our server to the cloud. If we catch any bugs at either period of time, we would add the issue to GitHub and then before deployment we would make sure that the issue is fixed. Once the issue is fixed all tests would be run again and if no bugs were found the server would be deployed to the cloud. If after deployment, the front end team finds a bug that we did not know about and that bug can be covered with an integration test, we would add more integration tests for the family of the bug that was found.

5.3. API Testing with Postman

Architecture of our product is client / server architecture. Therefore, every now and then the feature that was implemented in the backend, could not be tested using our client application. Thus, we used an API testing tool called Postman. With Postman, any endpoint that we publish whether in the local environment or in the cloud can be tested quickly and efficiently. The reason why we wanted to test our API is to mainly check the time it takes for the request to go to the server and get a response back. Another reason is to make sure that in the response from the server that we return information with fields that are expected by the front end. Because if the fields of the response are not identical to what is expected by the front end, then they would not be able to properly parse the data we send them. We created different environments to test the deployment and local servers, and created variables for each environment to be able to quickly switch from one mode to another and also to be able to use the same request for both development and deployment servers by just changing the environment.

5.4. Database monitoring with MongoDB Compass

To make sure that whatever we write to the database is written properly, we used MongoDB Compass, which lets us quickly connect to our development and deployment databases. After connecting to databases we would look for any abnormalities in the data that we uploaded to the db and also we would make sure that for example all indexes that are set up in our code have been propagated to the configuration of the MongoDB. Also, we would check whether any

updates that we do not allow in the MongoDB configuration, such as an _id of an object should not be updated after its creation, are rejected by the database in the database logs and also by looking at the documents in the database.

6. Maintenance Plan and Details

6.1. Server and Client

Every week metrics from Heroku are going to be checked to make sure that the system is running without any major issues. If the system crashes, the email using Sendgrid is going to be sent to the team that is responsible for the server with the details of the crash that includes: the request that triggered the crash, whether the system could restart itself after the crash. If there are any issues / bugs reported by the users of the client application then the server and client applications would be fixed and the corresponding tests would be added, and everything would be redeployed to make sure that the same bug does not persist in the future.

6.2. Scraped data and NER model

Currently, the scraping framework supports only allrecipes. To enlarge the number and the variety of the recipes, we will have to onboard other recipe websites. Hence, the scraping framework will have to be adjusted to facilitate new websites. Additionally, as the variety of recipe data increases, we will have to compile a new training dataset for our NER model to ensure a high accuracy.

7. Other Project Elements

7.1. Consideration of Various Factors in Engineering Design

Our product revolves around optimising the convenience and health factors of cooking. Therefore, a big part of our design process was food and health related considerations ranging from more serious conditions like allergies to milder concerns like making sure the users receive meal plans according to their desired calorie ranges. These dictated how we configure our recommendation algorithms. Depending on how one looks at it, these may also be considered as safety concerns especially when certain foods might risk the well being of the users.

One of the features of our app is financial optimization of cooking. To this end, we do a price estimation for the meals recommended.

In order to make the experience most tailored to the user, we will design our recommendations as such to take into account the food culture of the region where the user lives and has grown up with. We believe this will have a great impact on the user experience with the app.

Table 2. Factors that can affect analysis and design.

	Effect level	Effect
Public health	10	Improving the health of the users is a primary concern. We have configured our recommendations so that they provide health benefits to our users.

Public safety	4	We have put reliable failsafes to protect users who can have adverse health effects from specific foods.
Public welfare	4	Designing our tool so we both collect optimal data and give proper recommendations to help our users make cheaper and qualitative choices.
Global factors	0	Our app is quite personal in the experience it provides and therefore its design is not really reliant on global factors.
Cultural factors	8	In order to provide the best experience possible our recommendations include a variety of recipes from different cultures.
Social factors	0	We did not find any social factors which may be critical to consider during our design process.

7.2. Ethics and Professional Responsibilities

We have an ethical and professional responsibility to safeguard the data of our users. Special care was given to ensuring that our meal recommendations are safe and do not risk the health of people who use our platform.

Professionally, we have a responsibility to provide a robust software platform which gives accurate tailored recommendations, in contrast to simply generating arbitrary recommendations which are loosely relevant to our users.

We relied on data scraping to gather data. Data scraping could be ethically gray depending on whether the scraped entities want their data to be scrapped. To avoid any form of legal retaliation or ethical concerns we limited ourselves to publicly available data, and not scrape from entities who would be harmed by having their data scraped.

7.3. Judgements and Impacts to Various Contexts

Table 3. Judgements and impacts to various contexts

Judgement Description		
	Impact Level	Impact Description
Impact in Global Context	N/A	The application is intended for the Turkish local context.
Impact in Economic Context	Mid	It was our secondary priority to help people make financially informed decisions regarding their meal plans by providing price estimations.
Impact in Environmental Context	N/A	Our application does not have any direct environmental implications.
Impact in Social Context	N/A	Our application does not operate in social contexts.

7.4. Teamwork Details

Overall distribution of work among team members was as follows:

- Backed: Ibrahim and Perman
- Data gathering and processing: Khasmamad and Gledis
- User interface: Balaj

The following subsections go into more details.

7.4.2. Contributing and functioning effectively on the team

We have determined our goals beforehand and made sure that all of us are aware of these goals, so that we can make decisions faster and spend more time on development. Also, we divide tasks into modules to ensure proper teamwork. These modules are assigned to 3 people at most. Each of these sub-team has a leader assigned. By means of modulation and structure of our sub-teams, we can specialize in that field and give proper support to each other. What is more important, we always include all team members in the planning processes for the team to function effectively. So that nobody feels excluded and all members will share their ideas and be a part of not only work, but also decision making throughout the project. We prefer short frequent weekly meetings instead of long discussion meetings, which we found to be unproductive. These meetings are used to inform each other on our progress on our assigned tasks.

7.4.3. Helping create a collaborative and inclusive environment

In all of our work, we always kept others up to date on our progress to track each other's work and not to miss any important information. Besides informing what we have done on our task, we also let our team know about bad situations like not being able to handle the task assigned to him/her. So that we can check our division again and make a better distribution of tasks to ensure that all members are contributing. What is more, before we submit our work, each one of the tasks was assigned to some other team-mate for review and feedback. So that we ensure the quality of the work and give each other support.

7.4.4. Taking lead role and sharing leadership on the team

We did not have a single leader throughout the project. We have decided that every member will be the leader of the organizational proceedings of the team on a weekly basis. The thinking behind this strategy is that everybody gets to lead the team sometimes and is able to dedicate themselves toward our project's success. Similarly, we also divided our project into work packages, and these packages also had assigned leaders who organize the work done and coordinate with other package leaders on the implementation.

7.4.5. Meeting objectives

The main objective of our project was to create a meal plan generator that recommends meals considering user preferences. We were able to create a meal plan generator engine that serves the users as planned. This engine relies on the data that was successfully gathered from the internet and curated to the specific needs of our application. We went a step further to do NER on top of our ingredient data to classify different words into categories like ingredient name, quantity, and unit of measure. This information was intended to be used in price estimation, ingredient-based recipe similarity calculation, grocery list compilation and ordering. However, these tasks proved to be more challenging than expected and were delayed.

7.5. New Knowledge Acquired and Applied

7.5.1. Backend

We have used MongoDB, Express framework with Nodejs for the backend side. Some of us developed their first big project with these technologies and learned them along the way. For example, some of us did not have any experience with NoSQL databases. We had problems while trying to configure our database like collection connections, creating schemas for our objects like Recipe, MealPlan and writing methods for these schemas. All these challenges were fun and

improving for all of us. Other than MongoDB, we have also used AWS S3 storage services for storing our images. We have configured our backend so that we could immigrate any image from our backend to our S3 bucket in AWS. This was new for all of us.

Our product is currently deployed on Heroku. We have chosen Heroku since it is free. While trying to deploy our product, we have learned to build pipelines, setting triggers on branches. We also learned and set up a docker file in case we want to move to a big platform like AWS services.

Other than writing functional code that gives write output when given input, we have applied some other concepts to improve backend quality. These technologies are caching, testing, API documentation, and some other tools that improve request-response flow.

While trying to apply these concepts, we had some challenges.

For testing, We have used Jest for testing framework and Supertest for mocking the server. One of the problems we have faced while writing unit tests was running unit tests in parallel. Jest framework could easily run them parallel for us but the main problem was database connection of these unit tests. Some of the unit tests were changing the database concurrently. The concurrent changes in the database were causing unit tests to affect each other. Our solution was forcing the tests to run in a band. Disadvantage of this way is that it takes more time to finish tests but we have decided this since accuracy of our tests were more important than time.

We have used a npm library for caching. Only problem we had with cache was the invalidation of cache. We have decided to invalidate the cache regularly (once in an hour).

We have learned Swagger for documenting our endpoints. Using Swagger was useful for the Frontend team mostly since Swagger provides a nice user interface for our endpoints. This was a really useful tool for us since the usage of these tools was very intuitive and not time consuming. We could document our endpoints by writing some comment in the format specified by Swagger-above our endpoints.

We have learned how to verify a user account with email. We needed this functionality since people with bad intent can harm our database by spamming random accounts. Creating a token for email verification, storing them, validating them when the user uses it, setting expiry dates were fun challenges for us. We have used SendGrid for sending mails. Although we have discussed whether we should have an email server or not, we do not have an email server currently since we decided to focus on our major functionalities.

The code for the backend of Foodster can be found in this Github repository.

7.5.2. Data scraping and Interpretation

Both data scraping and processing were done in Python. Although the data team were familiar with Python and basic scraping techniques, it was the first time that we were working on data of this scale.

We started with exploring off-the-shelf solutions for scraping recipe-related data. However, neither of the available solutions met our needs exhaustively. Similarly, there was no off-the-shelf NER model that was trained for our purposes. Hence, we decided to build our own scraper and train our own NER model.

To begin with, we decided to maintain a Github repository separate from other components of our system. This repository includes the codebase and the necessary data files for scraping and processing data.

To assist us in the process, we studied the codebase given in this repository. However, we applied the policy of "failing fast." This policy allowed us to design our work to the best of our knowledge, then test it and add new functionalities or modify the existing design only if necessary. The motivation behind this policy is to test our knowledge, understand the gaps in our understanding with real feedback (e.g. the code fails to work or does not deliver the expected results), and apply fixes for specific reasons only. This helped us learn deeply and save time by being fully conscious of the additions and updates made to our codebase.

Training the NER model was not a trivial task either. We quickly realized that discussions around the applications of the spaCy library in the discussion forms are very limited. It was especially hard to find information on the new version of the library, which was remarkably different from the older versions. Thus, we were not able to find quick answers to our problems. This led us to study the official documentation of the library. It was not easy to pinpoint answers to our specific questions in the documentation. Thus, we decided to do an exhaustive exploration of the documentation and do trials and errors to find out solutions.

7.5.3. User interface

Flutter worked as the backbone of the user interface and hence the core of that was to learn and implement various widgets and components using the dart language. After learning core dart we decided to move on to the out of the box widgets and tools provided by flutter, widgets such as scaffolds, scroll views e.t.c were very useful throughout the development of the app. Design patterns such as Model View View-Controller which were encouraged by flutter developers were adopted.

Android development and debugging environments were set up, using both virtual and physical devices to ensure a smooth running of the app on a variety of devices for this android studio was a crucial tool, although at times IDE's such as VSCode were also used.

An array of libraries were used to help with the functionality of the frontend many of which were utilities, one specific utility that allowed for great efficiency was the HTTP library of flutter which allowed us to seamlessly process the interaction with RESTful API. Another useful library was the preferences manager which allowed us to keep track of user preferences, tokens and sessions.

The code for the front end can be found here.

8. Conclusion and Future Work

8.1. Conclusion

Although we could not get the API from Migros to order groceries and Yemeksepeti to order meals, everything else that we promised to do we did. It was a great experience for us to work on this project, because we learned how to collaborate in a team, how to develop high quality and large scale systems. We learned how to manage and lead teams. Therefore, we are happy with the knowledge that we gained by working on Foodster.
Not only did we learn how to write proper tests, come up with an architecture for an application and design the code, we also learned how to make code better by refactoring it and the value of the peer review that we did for both front end and back end of our applications.

8.2. Future work

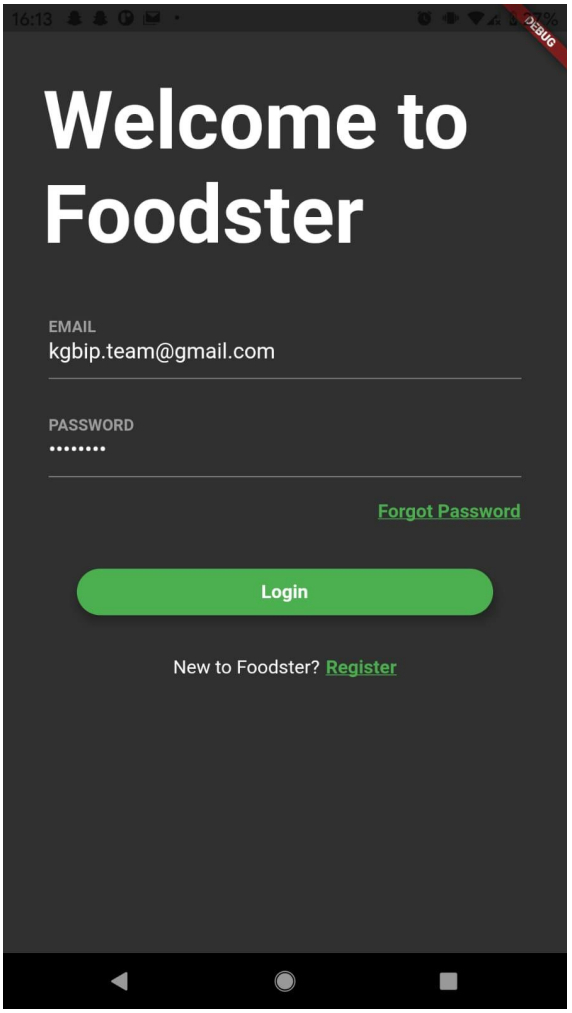
After gathering the feedback from other students and teachers, we are planning to keep working on the project. Also we are planning to publish the application to the Google Play and Apple Store to be able to get the feedback from real users as well and to further improve the quality of our application.
Since as a team we worked together for almost a year now, we might decide to work on projects together in the future as well.

9. User Manual

The user can download the app from the App Store/Playstore depending on their device and after installing will have the following set of pages, each of which will be delineated below.

Login Page

After installing the app the user must run the app from the launcher and will be greeted by the Login page. Here if the user possesses an account (with email and password details). He / She will enter these details to proceed to the Homepage of the app. If however this is not possible the user must proceed to register on the app from the Register Page (described below).



Register Page

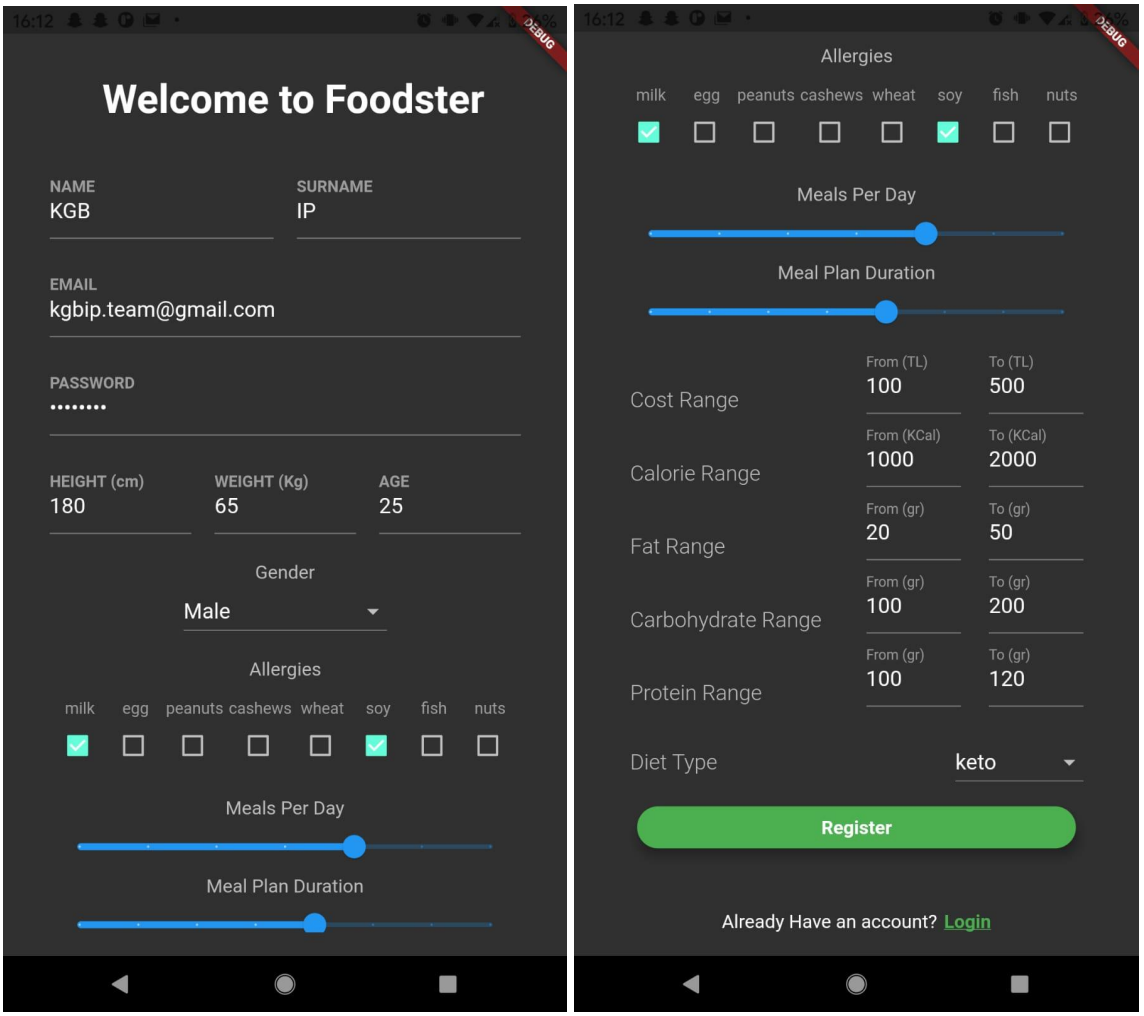
If during the first visit to the app the user does not possess an account he/she must sign up via the register page. Here the user enters a detailed amount of information which helps us generate a profile for the user and meal plans for the future. These details range from basic personal details:

- Name
- Surname
- Email (which the user will use to log in)
- Password (which the user will use to log in)
- Current Height
- Current Weight
- Current Age
- Gender

And some nutritional details

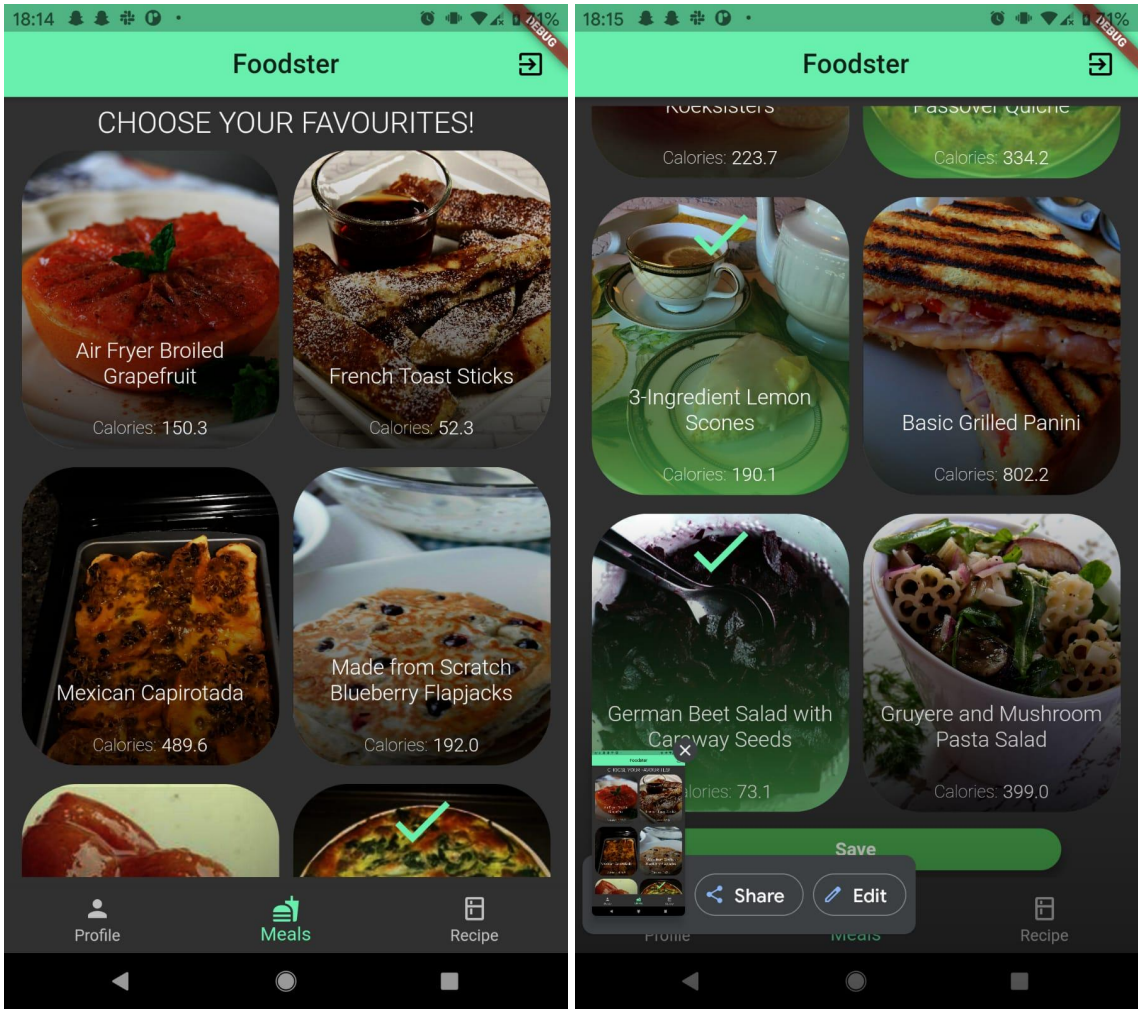
- Allergies
- Prefered number of Meals per day
- Duration of a generated meal plan
- Cost Range (for cooking in TL)
- Calorie Range (in the generated meal plan)
- Fat Rage (in the generated meal plan)
- Carbohydrate range (in the generated meal plan)
- Protein Range (in the generated meal plan)
- Diet Type (from popular diet types such as keto, paleo, vegetarian e.t.c)

After the user clicks the register button, if all details are valid the account is created and the user is directed back to the login page to use the entered details to login.



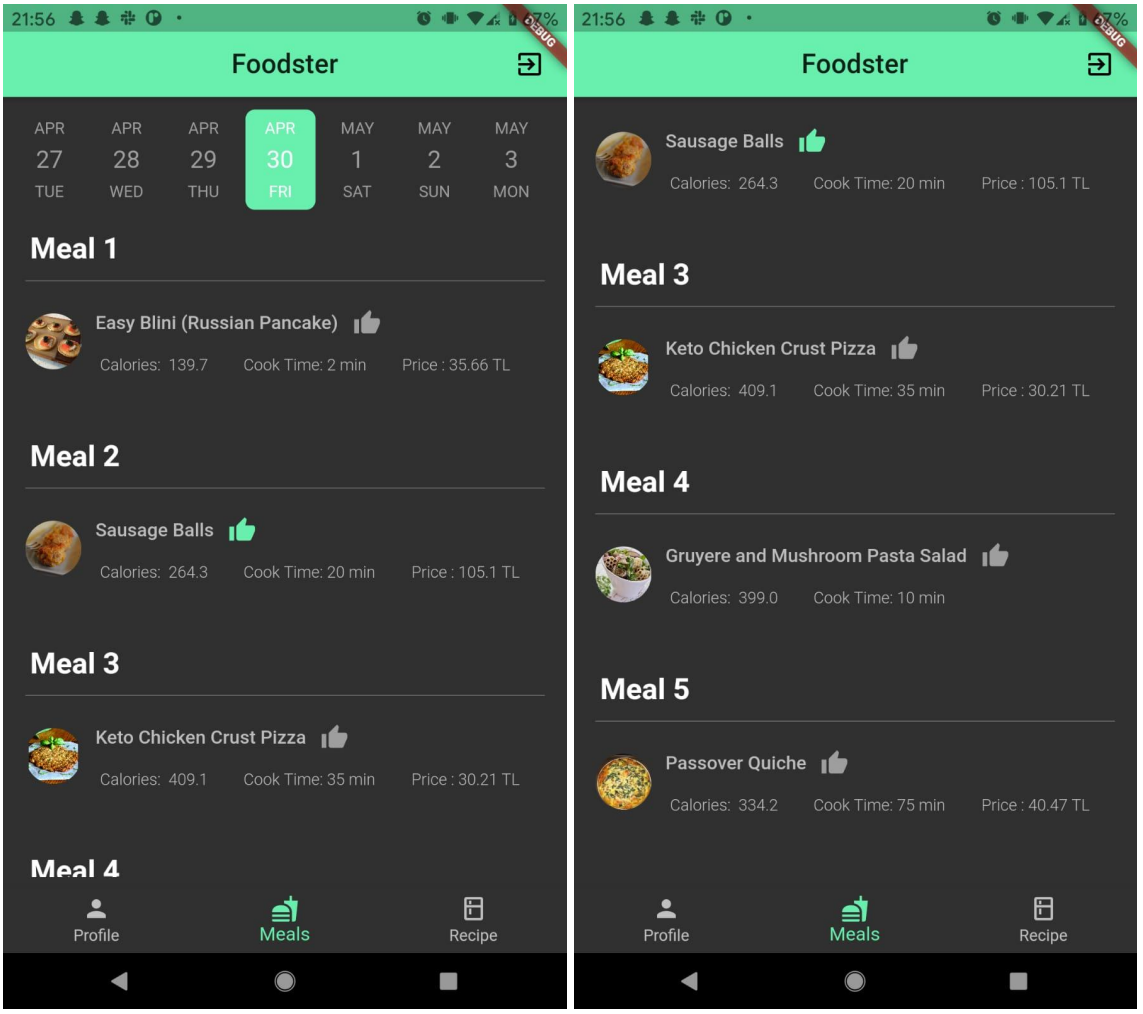
Recommendations Page

After the user logs in they are shown a number of the top recipes in the recommendation system, based on an array of factors upon which our recommendation system relies, here in a very convenient manner the user selects the recipe's that they find the most suitable to their preferences and clicks save. These recipes are used for generating further meal plans to the user in the future.



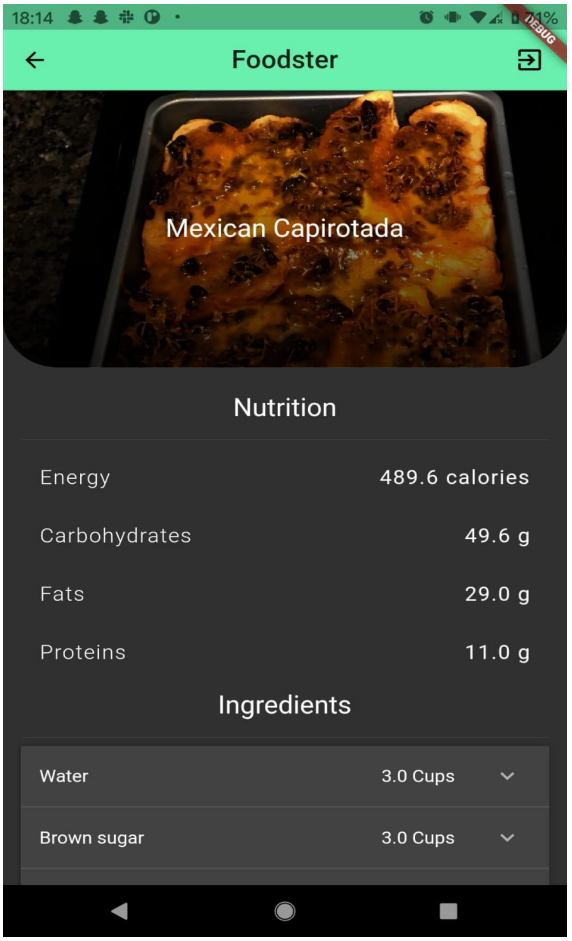
Meal Page

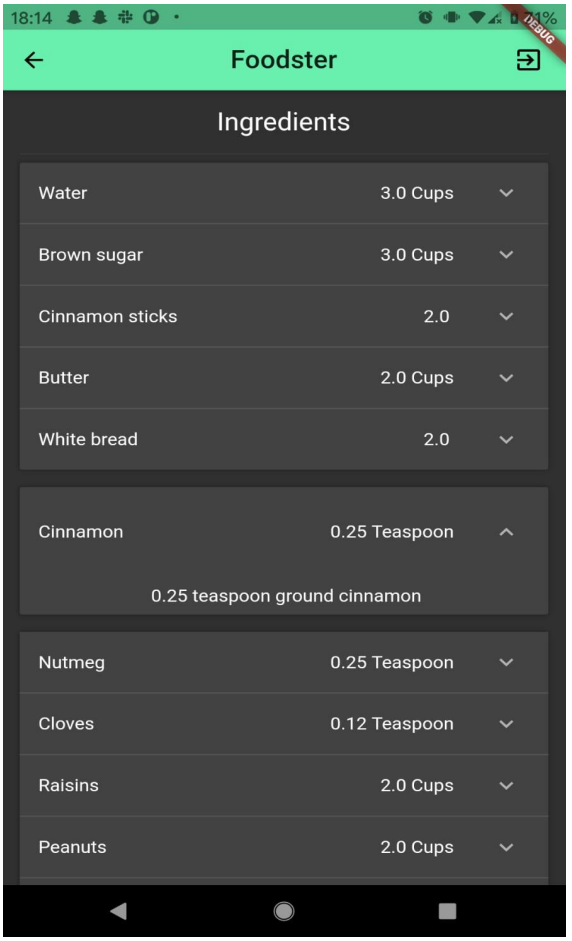
After choosing the favourites the user is taken to the Meal Page, which is the core page of the application, and can be navigated to by clicking the center icon on the bottom navigation bar. Here the user is given all his / her personalized and customized meals which consist of 1 or more recipes, their nutrition, the time to cook and the estimated price (fetched from local markets). Upon clicking any of the recipes the user is taken to the recipe details page where he/she can see the instructions, nutrition and all other details of the recipe. Clicking the top right logout button will lead the user back to the login page.

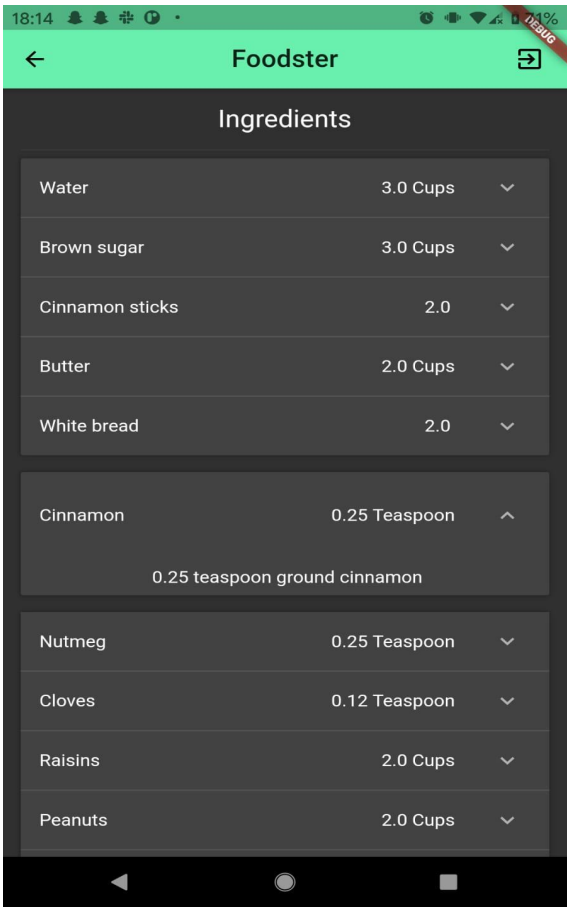


Recipe Details Page

If the user clicks on any of the recipes they are taken to the recipe details page, which contains an in depth information about nutrition of the recipe, including carbohydrate, fats, proteins and total calories. The Ingredients, their amounts and their details in words, and finally the preparation instructions for each recipe are provided.

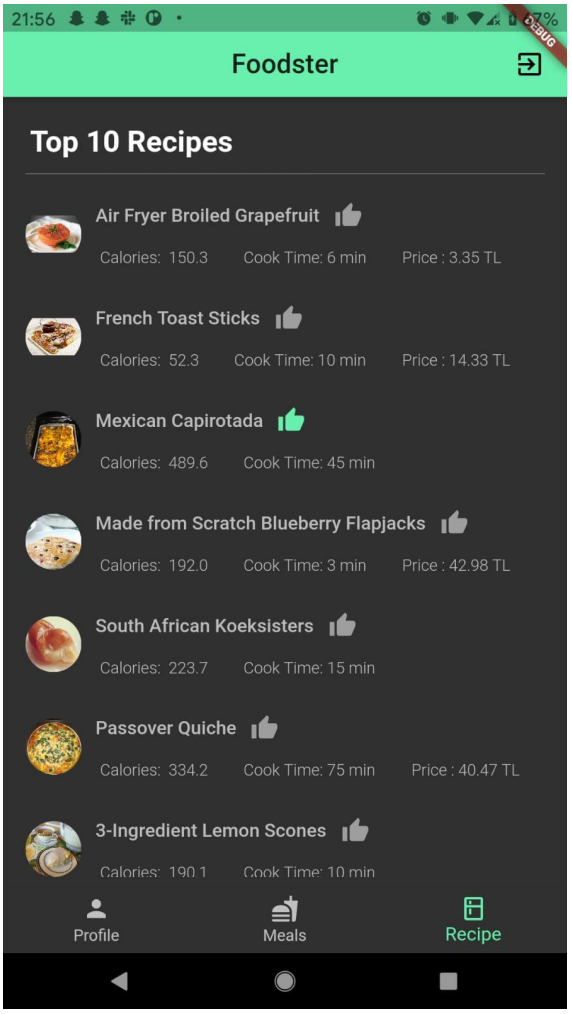

18:14
Foodster
Mexican Capirotada
Nutrition
Energy 489.6 calories
Carbohydrates 49.6 g
Fats 29.0 g
Proteins 11.0 g
Ingredients
Water 3.0 Cups
Brown sugar 3.0 Cups


18:14
Foodster
Ingredients
Water 3.0 Cups
Brown sugar 3.0 Cups
Cinnamon sticks 2.0
Butter 2.0 Cups
White bread 2.0
Cinnamon 0.25 Teaspoon
0.25 teaspoon ground cinnamon
Nutmeg 0.25 Teaspoon
Cloves 0.12 Teaspoon
Raisins 2.0 Cups
Peanuts 2.0 Cups


18:14
Foodster
Ingredients
Water 3.0 Cups
Brown sugar 3.0 Cups
Cinnamon sticks 2.0
Butter 2.0 Cups
White bread 2.0
Cinnamon 0.25 Teaspoon
0.25 teaspoon ground cinnamon
Nutmeg 0.25 Teaspoon
Cloves 0.12 Teaspoon
Raisins 2.0 Cups
Peanuts 2.0 Cups

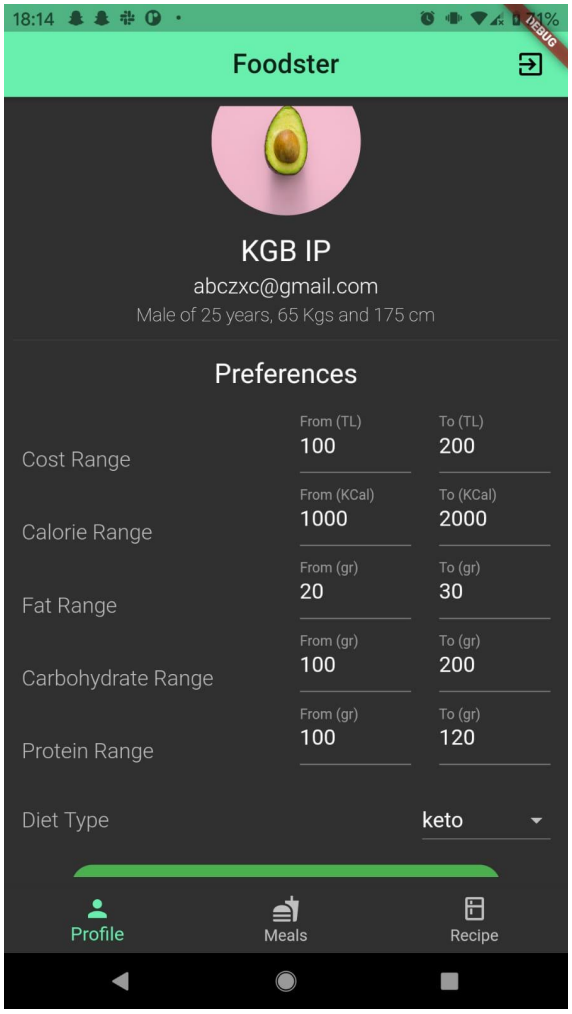
Top Recipes Page

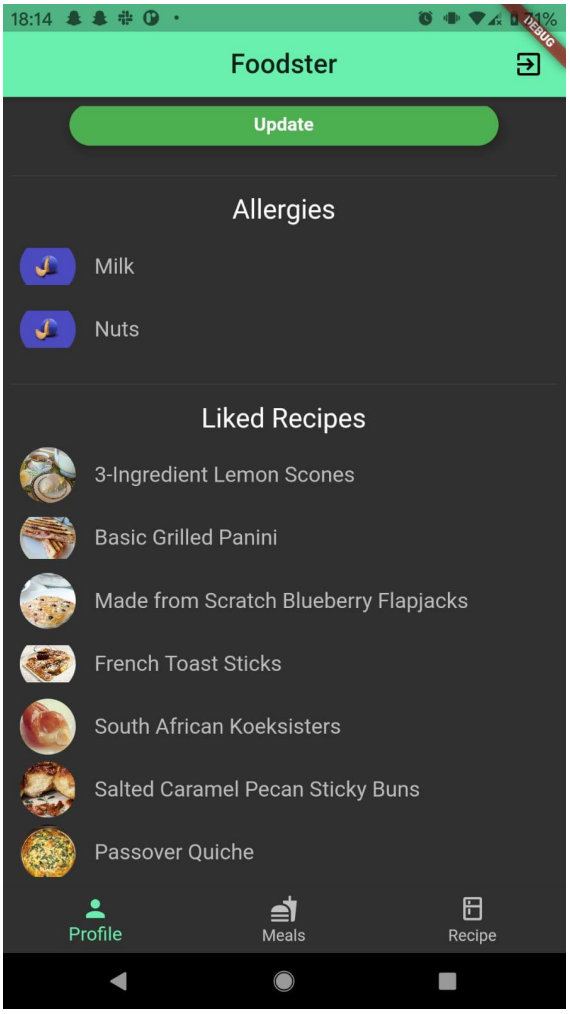
The top recipes page can be navigated to by clicking the leftmost icon on the bottom navigation bar and includes the best recipes available on the platform according to various users. Each of these recipes can be liked by the user so that the future recommendations are shaped accordingly, clicking on any of these recipes takes the user to its details



User Details Page

The user can also view the complete details of their profile by clicking the bottom left button on the navigation bar to go to the user details page. Here they can view all the preferences they had set when they signed up. A profile picture, their personal details. Upon scrolling down the user can view the allergies and the recipe's they had liked (upon which many of their recommendations are based). If the user wants to update their preferences they can simply click the update button after changing the respective fields and their choices will be updated and reflected.





10. Glossary

- Nodejs: asynchronous event-driven JavaScript runtime [1]
- Cloud: a remote computer that is available on demand
- Server: a computer that responses to requests of other computers
- Express: back end web application framework
- Docker: product that delivers software as a container, in other words an isolated OS [2]
- Migros: a supermarket chain
- Part-of-speech (POS) tagger: identifies the correct part of speech.
- Tf-idf: term frequency–inverse document frequency

11. References

1. "About NodeJS", OpenJS Foundation. https://nodejs.org/en/about/
2. "Docker Overview", Docker. https://docs.docker.com/get-started/overview/