Component Specification

By: Corina Geier, Jeffrey Lai, Edward Lou, Sai Muktevi & Andrew Zhou

RecipEat

Table of Contents

Overview 2
API 2
Software components 3
Interactions to accomplish use cases 4
Preliminary plan 5
Testing 6
References 7
Appendix 8

Overview

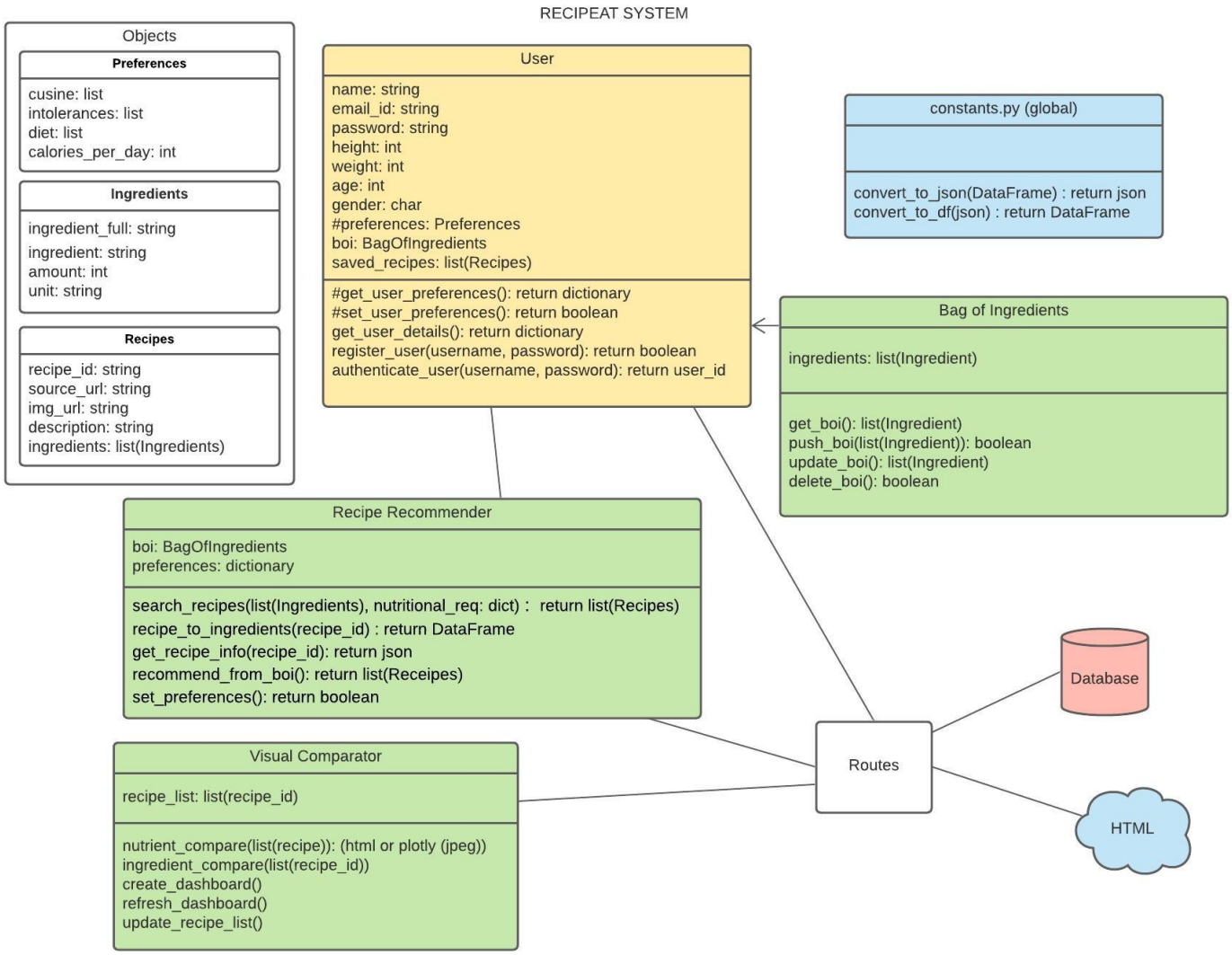
The RecipEat project offers a web application for users to incorporate a balanced diet based on nutrient targets and receive recommended recipes based on ingredients.

The tech stack of the project include Python (programming language), Flask (Framework), Firebase (Database), HTML, and CSS. Python provides easy-to-use packages for developers to retrieve data from Spoonacular through APIs, perform data transformations, build modules, and create visualizations. Flask offers an extensible web microframework for building our web application. The Firebase platform serves the website to implement an authentication flow. HTML and CSS, the foundations of the site, describe the structure and presentation of our web pages.

API

Spoonacular is a Recipe-Food-Nutrition API providing access to over 365,000 recipes through ingredient- and nutrient-based queries. The user can also include keywords or phrases, such as ‘lactose intolerance’, to search for recipes that accommodate any special dietary restrictions or preferences. Additionally, the user can analyze nutritional information for each recipe by utilizing the visualization widgets.

Software components



Our software is split into two major components: frontend and backend. Our frontend component is a web page interface that allows users to register an account, add ingredients, find recipes, and compare recipes. These interfaces are made up of several html pages (*index.html, login.html, register.html, ingredients.html, recommender.html*). All these html components and our backend are tied together by *routes.py*, a Flask web framework in python.

Our backend component is made up of several modules to recommend recipes to a user.

user.py contains the user class and stores user data such as username, name, email, height, weight, age, and gender, and BagOfIngredients. This class has methods to

register and authenticate a user. There are also methods to return user details such as height, weight, gender, and age.

bag_of_ingredients.py contains the BagOfIngredients class. This class is used to store a list of ingredients and the amount of each ingredient. The class has methods to return the ingredient list, update (delete and insert) ingredients into the list, and delete the bag of ingredients (set the ingredient list empty).

recipe_recommender.py is a class that has methods to search for recipes with given ingredients and nutritional requirements. It can also get recipe information with a given recipe.

visual_comparater.py is a class that is used to make a visual comparison of two or more recipes. Calling functions in the visualComparater will return visual comparisons of the nutritional information of the given recipes.

Interactions to accomplish use cases

The software components are split into 3 different *base* layers:

1. Front-end - User Interface (HTML, CSS, Bootstrap, Flask)
2. Back-end - Python Modules and Spoonacular API
3. Database - Firebase (using Pyrebase python wrapper library for Firebase)

Each layer is composed of modules that work together to carry out the different possible use case scenarios as mentioned in the Functional Specifications document. The following are the various interactions between these components and their modules to execute the necessary steps to implement the required functionality for each use case.

1) A student wants to make quick healthy meals with ingredients he/she has at hand.

The User in this scenario will be the student. The student will use the User Interface to register or login. All the templates being rendered for the UI and requests being generated from the application will be handled in the *routes.py* folder using Flask. The authentication system is integrated with Firebase which stores user credentials. Once authenticated the student will have access to their “Bag of Ingredients” where the user can add ingredients they have available into their bag, set user preferences regarding type of food, and input any intolerances. The user is also provided a separate field to enter their nutritional requirements. The class defined in the *bag_of_ingredients.py* component will be responsible for operating on the user’s bag. A button will be provided to initiate the recommendation system which will interact with the Spoonacular recommendation API and provide results accordingly. This will be handled by the

recipe_recommender.py component. The user is then given an option to choose from a set of recommended recipes that are based on the “Bag of Ingredients” and preferences. The user can then quickly choose a recipe of their liking and confirm the ingredients being used in the recipe will be deleted from their bag. The selection process will be taken care of in the *routes.py* component which will update the Firebase Database in the back-end. The user can then repeat the process as required by continuously adding more ingredients to the bag when available.

2) A Nutrition Coach that does Meal Planning and decides to prepare a menu to fulfill dietary requirements for a client.

The user in this case will be the Nutrition Coach or anyone looking to plan future healthy meals for their diet. The user will build a menu for each day choosing from the recipes recommended by the application. Similar to the previous use case, the user will register or login to the application and look at recipes according to their preferences and nutritional requirements without having to fill out a “bag of ingredients” this time. The *recipe_recommender.py* component will recommend recipes accordingly and show the ingredients required, which the user can plan to add to their bag later. To make sure that the macro nutrient requirements involved in the diet plan for their client are being fulfilled the user can utilize the visual comparator to help streamline the process of choosing the best recipe. The comparator is handled by the *visual_comparator.py* component which works with the Spoonacular API to obtain recipe data.

Preliminary plan

A list of tasks in priority order.

1. Bag of ingredients
2. Nutrition Constraints
3. Recipe Recommender
4. User Registration and Login
5. Visual Comparator
6. User Preferences

Our bare minimum functionality is to have a user get recommended recipes from their provided list of ingredients and nutritional constraints. That’s why our number one priority is to set up functionality for a user to add ingredients and set up their nutritional constraints. After having these functionalities built in, our next most important feature is the recipe recommender. By having 1, 2, and 3 implemented, we have achieved the bare minimum recipe recommender system.

Our next priority is to add a user registration and login system, so users can have a running ingredient list and save nutritional preferences. This will allow better and more

tailored recipe recommendations. We would also like to implement a visual recipe comparator, allowing the user to graphically compare the details of multiple recipes of their choice, since people absorb information the best visually. This is an optional feature to include if we have time. Last in our priority list is to save user preferences. We can ask for nutritional constraints everytime the user wants to search recipes, and therefore saving user preferences is not a crucial feature to implement.

Testing

Testing the Back-End Components

Testing will be performed incrementally throughout the various components of our application.

Most of our code is based on working with the Spoonacular API which is already well documented and maintained. We will be feeding the API with parameters based on user and application requirements and transforming the data obtained from the APIs. At this basic level we will implement **unit testing** based on Test-Driven Development to check the input and outputs of each method using the APIs as per the requirements.

The next level of testing includes unit tests on the various transformations of the data obtained from the APIs. Transformations will be needed to move data between modules and make it feasible to work with for displaying on a UI. An example of this would be changing the API output of a Pandas DataFrame to a JSON object for displaying on an HTML page using Flask. These tests ensure proper change in data format.

Finally we move on to testing the functionalities tied with our 3 main components i.e., the Bag of Ingredients, the Recipe Recommender, and the Visual Comparator. Each of the functional specifications of these components are thoroughly unit tested with the necessary inputs and for the expected outputs.

At the use case level, we test the interaction between these components using **integration testing** to ensure the correct flow of data across components and ensure the required transformations. Each use case has a different flow of control according to the system design of our application.

Testing the Database

Data consistency is also very important in a web application. We have unit tests for:

- Checking for data integrity and errors in editing, deleting, modifying the forms or doing any DB related functionality.
- Checking if all the database queries are executed correctly, that data is retrieved, and also updated correctly.

Testing the User Interface

We have concerned user profiles testing the application with different use cases to ensure that we have the right amount of abstraction and simplification of the UI to enable ease of interactivity with the application. We may consider looking into working with Selenium or a more automated tool like LambdaTest. The main testing will be **usability** and **functionality tests** focussing on:

- Cross-browser compatibility,
- Server response tests,
- Rendering HTML templates with proper styling,
- Interactive and responsive design,
- Forms,
- Navigation, Internal and external links

We are looking into handling continuous integration, automated testing and deployment using Travis CI. Due to this being an end-to-end application, a more pragmatic target is to implement complete test coverage of at least our main component specifications based on the current project timeline. We shouldn't have too many performance issues due to usage of a cloud storage system and a well developed readily available API. Security of the database will also be managed by the Firebase authentication system.

References

Firebase Documentation. (2020). Firebase. https://firebase.google.com/docs

Flask Documentation (1.1.x). (2010). Flask.

https://flask.palletsprojects.com/en/1.1.x/

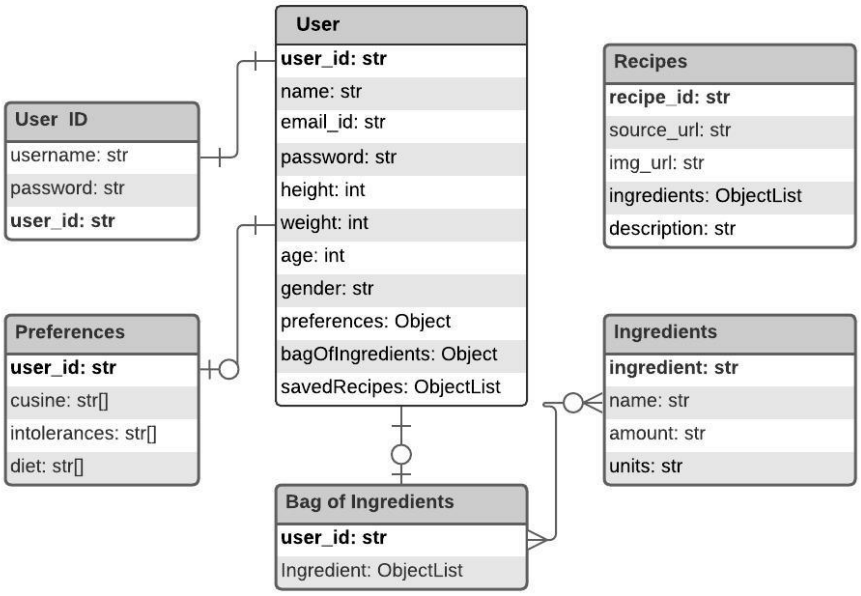
Spoonacular recipe and food API. (2020). Spoonacular. https://spoonacular.com/food-api/docs

Appendix

DATABASE SCHEMA (NoSQL)

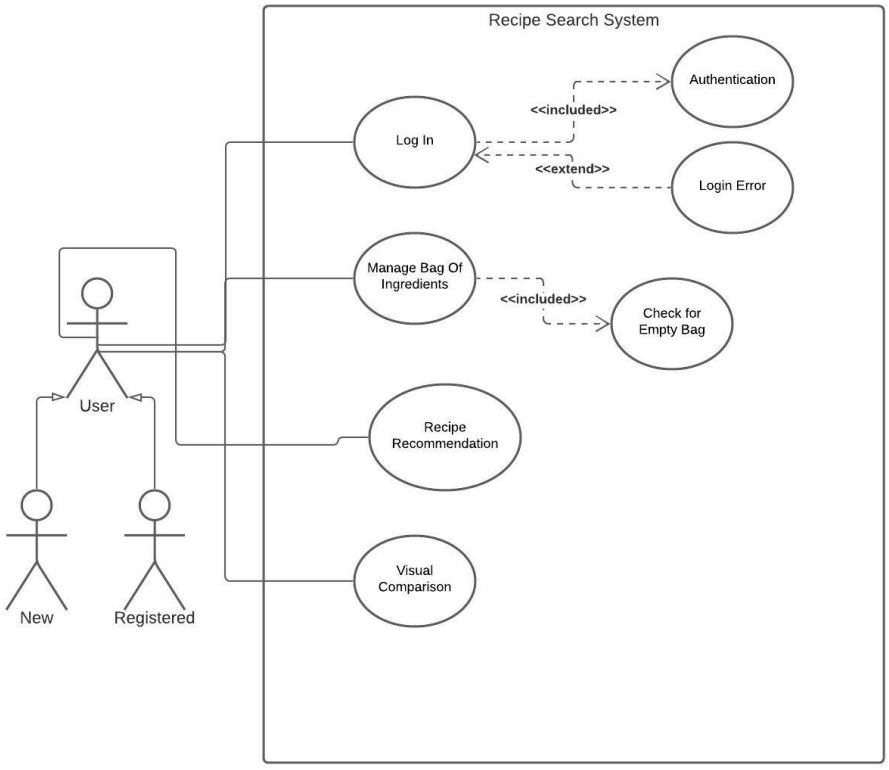
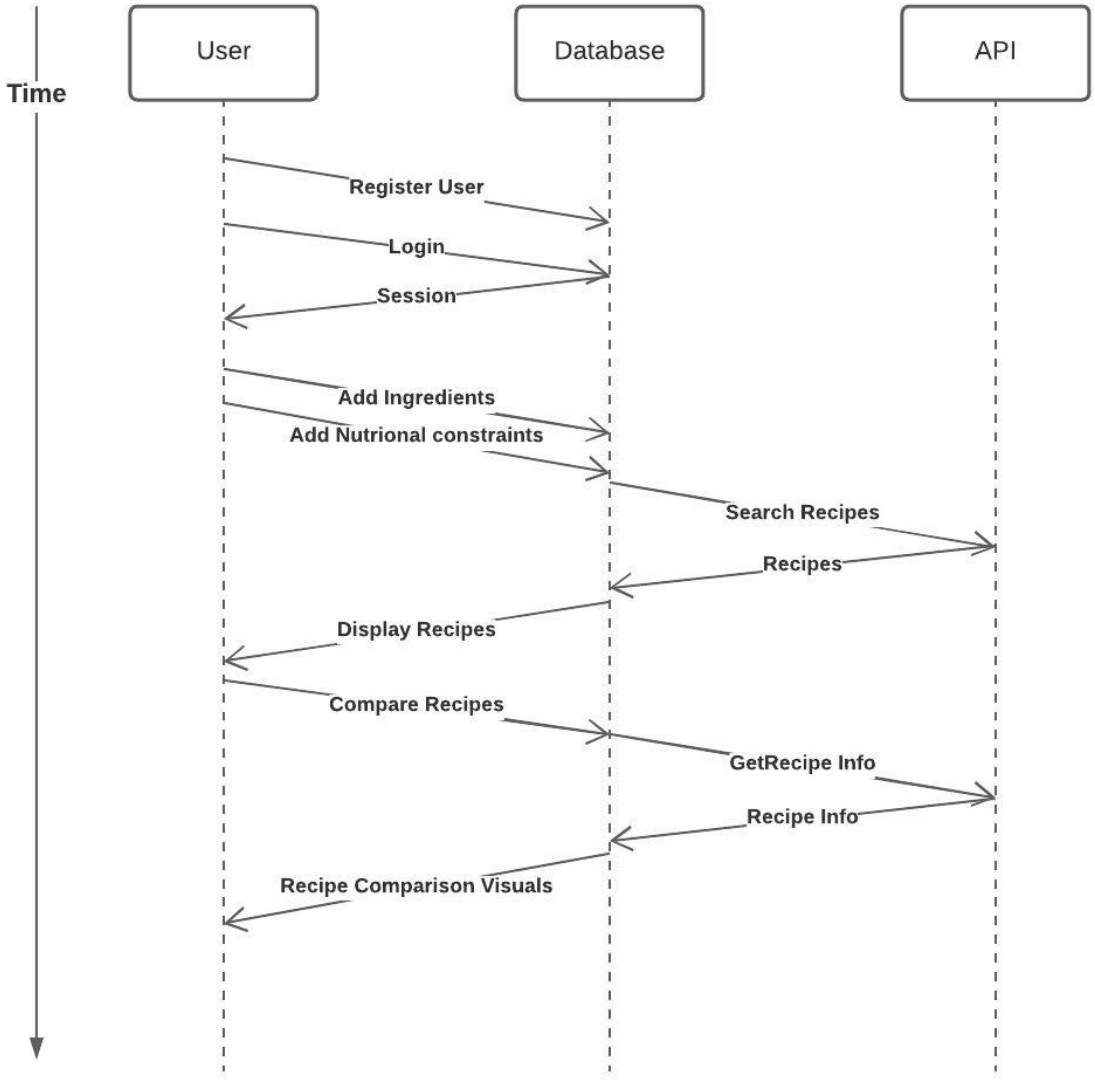
Recipes can store the recommended recipes that were used and keep track

primary key is bold



USE CASE DIAGRAM

included : happens every time
extend : happens sometimes



SEQUENCE DIAGRAM