Recipe Generator Console App

Abstract

This report details the development of our recipe generator console app, including its design, implementation, and testing, as well as the integration of the Spoonacular API.

By Antonella, Evelina, Dominique and Hafsa

Contents

Introduction

This document provides a comprehensive overview of the development and functionality of a recipe generator console application. The primary objective of this project is to develop a functional, user-friendly, and efficient application.

The roadmap of the report is detailed below:

- **Background**: Provides a comprehensive explanation of the project’s context and how it works, including the specific requirements and features of the recipe generator application.
- **Specifications and Design:** Delivers an in-depth analysis of how the app's features were developed and integrated, including relevant code examples and diagrams where applicable. Detailing the functional and non-functional requirements of the project.
- **Implementation and Execution**: Outlines the development and implementation process and approach. Specifies each team member's roles and responsibilities and details the tools and libraries. As well as, any agile development elements used in the project.
- **Testing and Validation**: Discusses the testing strategies used to ensure the app’s reliability and performance.

Background

This recipe generator app was designed to help users discover recipes based on ingredients they already have in their homes. The primary objective of this application was to streamline the cooking process by assisting users in finding recipes that do not require them to purchase additional ingredients. Secondary to this, it minimises food waste and allows for a convenient and accessible way for people to prepare meals. In addition, more features were introduced to the app during the development process.

The final version of the app connects with the external API, Spoonacular, which requires an API key for activation. The app fetches data from various API endpoints and offers three query options: users can either input specific ingredients to find recipes that include them, get random recipes, or search for recipes based on predefined categories. Additionally, users have the option to save the recipes provided by the app and export them to an Excel file. It is important to note that users will require an API key to activate this API and it also uses libraries that need to be installed. For an easy user experience, we have created a README file which contains step-by-step instructions on how to run the application along with a requirements file.

Specifications and Design

Functional vs non-functional requirements

Functional requirements

The main functional requirement of the app is the possibility to search for recipes from the Spoonacular API according to set parameters. To achieve this the app utilises user interaction in the form of **input prompts** where the user can enter their choice or data.

We've included three options to get different recipes.

- **Search recipes by ingredient/s:** this allows users to input ingredients and get recipes from the API based on those ingredients.
- **Search random recipes:** users can get random recipes from the API that they can take inspiration from.
- **Search recipes by category:** this presents a category menu to users, allowing them to select from predefined categories such as vegan/vegetarian, quick snacks, desserts and various types of meat recipes.

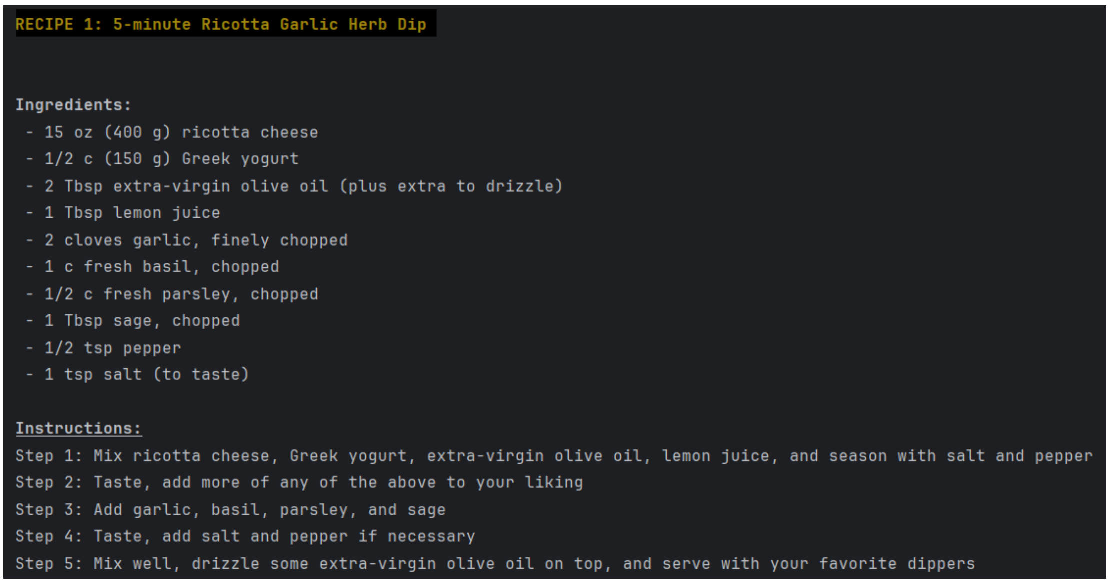
In addition to this functionality, we've included the possibility to save chosen recipes whilst the app is running. These can be viewed at any time by selecting the appropriate menu option. Users can then also export these saved recipes to an excel file for offline use. Furthermore, and across those functional requirements, the app includes error handling to manage potential issues with the external API and to manage user input errors, such as incorrect entries or menu options.

Non-functional requirements

Our app features a simple and easy to navigate interface with well-structured and concise menus. The user prompts, instructions and messages are clear and easy to understand. Whilst the API retrieves a lot of data from each query, we've carefully selected the data that are relevant to the objective of the app. In addition, we added a logging system that tracks function calls, outputs and errors to assist with debugging and performance monitoring.

Design and architecture

We have divided the code architecture into multiple files and classes according to the features of the app. For example, we have a main class that handles the core flow and runs the app from start to finish. The app starts by initialising all necessary components, such as the API connection, recipe finder, and display modules. The user is presented with the main menu and can choose to either search for recipes, display saved recipes, or export recipes. Based on the user input, the app will get recipe data from the API and display it. The user can choose to save recipes temporarily so they can continue browsing the available recipes, which can later be exported to an Excel file. The user can choose to exit the app at any time. Below is a picture of what the output looks like when showing the recipes:



```
RECIPE 1: 5-minute Ricotta Garlic Herb Dip


Ingredients:
 - 15 oz (400 g) ricotta cheese
 - 1/2 c (150 g) Greek yogurt
 - 2 Tbsp extra-virgin olive oil (plus extra to drizzle)
 - 1 Tbsp lemon juice
 - 2 cloves garlic, finely chopped
 - 1 c fresh basil, chopped
 - 1/2 c fresh parsley, chopped
 - 1 Tbsp sage, chopped
 - 1/2 tsp pepper
 - 1 tsp salt (to taste)

Instructions:
Step 1: Mix ricotta cheese, Greek yogurt, extra-virgin olive oil, lemon juice, and season with salt and pepper
Step 2: Taste, add more of any of the above to your liking
Step 3: Add garlic, basil, parsley, and sage
Step 4: Taste, add salt and pepper if necessary
Step 5: Mix well, drizzle some extra-virgin olive oil on top, and serve with your favorite dippers
```

Implementation and Evaluation

General development approach and team member roles

The development approach was agreed upon by all team members. We used agile methodology, which meant we allowed the project's objectives to change and improve during the process. We opted for flexible roles to ensure that contributions were as balanced as possible, taking into account our individual availability and strengths. In our first team meeting we discussed our project ideas and selected one that aligned with our interests and was feasible to achieve. We set up a shared calendar to track all our meetings and internal deadlines, ensuring that everyone was informed of the timelines.

Tools and libraries

We used **PyCharm** as our chosen IDE to develop the code and utilised **Git** and **Github** to manage it. We created a repository and added all group members as collaborators. Each member of the team created a separate branch to work on to not interfere with the main branch and compromise the current working version of the code. **Slack** was our main source of communication as well as weekly calls on **Google meet**. We also created a **Google Drive** shared folder to organise our project.

Our app utilised different Python libraries to execute the code:

requests - was used to make HTTP requests to the API

abc - was used to create abstract base classes

collections - utilised ***deque*** and ***defaultdict*** for queue operations and dictionaries with default values

functools - utilised ***wraps*** to preserve the original function's metadata when decorating it

unittest - was used for testing the functionality of the code. From this library we used the submodule ***unittest.mock*** to utilise ***Mock*** and **patch** which are used to replace real objects and functions with mock versions, allowing for controlled testing environments and isolation of dependencies. ***patch*** temporarily substitutes objects or methods, whereas ***Mock*** creates objects that simulate specific behaviours and interactions.

io - utilised ***StringIO*** to provide an in-memory stream for capturing text output, allowing for easy testing of functions that produce print statements or other text-based output. In our unit test, we used ***StringIO*** to redirect and capture the output generated by the application. This approach enabled us to effectively verify that our console application's output was correct and formatted as intended.

Implementation process

At the first iteration we agreed each team member would work on getting the main functions of the code working, i.e. getting recipes by ingredients and searching for random recipes. At our first review we agreed to merge one branch into main as it was the most complete code and thereafter we divided the work according to features. We defined the pending task, set deadlines and assigned a team member responsible for delivering it. We had regular check-ins on Slack and had calls at least once a week, sometimes more if required. These meetings were recorded so every member would be aware of the agreements. We tried to circle the work when necessary to make sure everyone contributed to every part.

At each weekly meeting we discussed our progress, dependencies and blockers. It was in these meetings decisions were made to alter our objective by adding new features to the code, this was features such as adding search by category, saving recipes and exporting recipes.

As the project grew, so did our code and we decided to separate the code into several files to distinguish the features and make it more readable. This also made it easier to avoid merging issues. We did continual code reviews and code refactoring which were communicated and agreed upon in Slack before merging.

Implementation challenges

Time Management: We set deadlines and communicated if we could achieve them or not and in that case to inform why we couldn't. In that way, we could figure out how to solve them as a group. It was a challenge and we managed to handle it in the best possible way.

Overlapping Work: Collaborating simultaneously was beneficial as it allowed us to observe each team member's approach and ideas but it was also challenging. We sometimes had to avoid interfering with each other's code, which complicated refactoring and code reviews.

External Factors: Most team members have full-time jobs, which not only consumed significant hours but also contributed to increased exhaustion.

Unit Testing: Unit testing was particularly challenging due to the project deadline and our development process. We began testing before the code was fully completed, which led to the unit tests being incorrect as the rest of the code developed.

Merging issues: At first, we questioned if everyone should test the code before merging new changes into the main branch. After failing and testing, we decided that the best approach was to merge if the code worked and to trust each other's work. We had merging issues that we solved individually or by handing it over to someone else to have a fresh perspective.

Testing and Evaluation

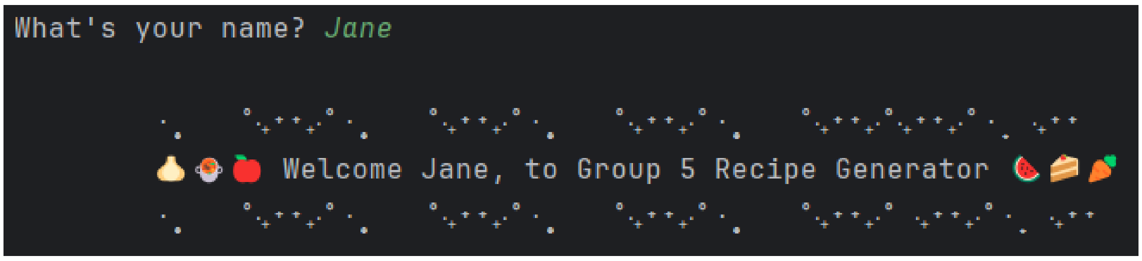
Testing strategy

The testing strategy that we adopted was designed to ensure optimum functionality and performability of the recipe generator console application. During the development stage, we used print statements and simplified versions of the code to validate intermediate results and to ensure that each component was working as expected. We enforced regularly tested error handling mechanisms to ensure that issues such as invalid inputs and API errors were managed effectively. Furthermore, we created comprehensive unit test files to evaluate different scenarios, covering the primary functionalities and mock what the user input would be.

Functional and user testing

We conducted tests that ensured all the functional requirements were met. This included: recipe searches by ingredients, random recipes, and category-based searches. Additionally, we made sure that we tested that the app handled incorrect inputs, HTTP and API errors to guarantee that there would be error messages shown to the user. This allows for a positive user experience.

Primarily, we repeatedly tested the requests to ensure the app responded as expected. For instance, we created a unit test file to mock our existing display file. One of the tests verified that the 'display_recipes' method of the 'RecipeDisplay' class correctly formats and prints the recipe details when provided with specific data. As well as this, it checks if the printed output contains the correct strings so that the recipe title is displayed correctly, used and missing ingredients are listed as expected. For instance, when the app runs as intended, some of the output will look like the below:



```
What's your name? Jane

          ·.   °·₊⁺⁺₊·° ·.   °·₊⁺⁺₊·° ·.   °·₊⁺⁺₊·° ·.   °·₊⁺⁺₊·°·₊⁺⁺₊·° ·.  ·₊⁺⁺
          🧄🍲🍎 Welcome Jane, to Group 5 Recipe Generator 🍉🍰🥕
          ·.   °·₊⁺⁺₊·° ·.   °·₊⁺⁺₊·° ·.   °·₊⁺⁺₊·° ·.   °·₊⁺⁺₊·° ·₊⁺⁺₊·° ·.  ·₊⁺⁺
```

System limitations

Although the application underwent extensive testing, there are some system limitations. In order for the app to function, it heavily depends on data provided by the external API. If there are any changes to the API's endpoints, data structure, or downtime, this could severely impact the app's performance and reliability. Additionally, the API requires a key with a limited number of queries. Despite the API having a plethora of advantageous characteristics, a major drawback is that the users must sign up to the Spoonacular in order to use the console app. To combat this issue, we have included a comprehensive guide in the README.md file that walks users through the process of setting up and running the console app.

Conclusion

To summarise, the development of our recipe generator console application has been rewarding and informative as it has taught us how to collaborate effectively to build a successful project. From our initial ideas to a functional, user-friendly product, we fulfilled all project requirements and objectives, serving as a testament to our teamwork and dedication. Our console application effectively meets its core objectives by providing functionalities to search for recipes by ingredients, randomly, or by category, and along with features such as temporary saving and exporting recipes to Excel. It handles errors using decorators and provides a user-friendly display, leveraging an external API and various libraries. Our robust testing strategy identified and resolved potential issues, resulting in a dependable and reliable final product. Despite assigning individual tasks, we found the time to communicate our progress to each other and complete tasks collaboratively. We utilised our README.md file, comments in the code, slack and meetings to maintain a flexible and coordinated approach. This project showcased our capacity to create a functional and efficient application, highlighting our strong teamwork and problem-solving skills, with the iterative approach, team collaboration, and adaptability being crucial in overcoming challenges and enhancing the application's features.