NELSON MANDELA UNIVERSITY

School of Information Technology

IT PROJECT 3 (ITPV302)

Bachelor of Information Technology (BIT)

18 October 2024

Implementation Document for Thyme To Cook Recipe Application

Compiled by

Lecturer	Student Name	Student Number	Date Submitted
Prof Bertram Haskins	Nathan Rogers Zanele Mndaweni Max Naidoo	221412581 225097524 225227053	18 October 2024

Table of Contents

Table of Figures

3. Introduction

This document outlines the implementation of our Thyme To Cook recipe application and website, detailing all the tools and technologies that were used during development, complex code excerpts, and all the challenges encountered during development. The main purpose of this document is to explain the choices made during development, demonstrate the complexity of the code that drives the app's unique features, and reflect on the technical problems faced and how they were resolved.

This document has 4 key sections. The Choice of tools section describes the specific tools, frameworks and APIs used in developing the app and why they were chosen. The extracts of code section displays code snippets of complex code that caused significant challenges during the development of the app and provides detailed explanations for each code snippet. The source code reference section provides references for any external code or resources we used or adapted for our project. The Problems encountered section provides a reflection on the technical difficulties we faced while developing the app, such as specific coding errors or technical limitations and how we eventually resolved them.

This document serves as the comprehensive overview of the technical implementation of our project.

3.1 Choice of Tools

3.1.1 Visual Studio 2022

We chose Visual Studio Code (VS code) 2022 as our primary Integrated Development Environment (IDE) because it is very lightweight, highly customizable and provides various useful extensions for Dart and Flutter. VS code integrates Git for version control which provides a useful way of working in a group. VS Code also allows for the projects to easily developed within isolated environments away from external packages on the system which made it the best choice for our app development.

3.1.2 Firebase

We used Firebase to store all recipe data, user details and images in real-time. Firebase was selected because of its built-in security features and smooth integration with flutter. Firebase handles authentication for user login and registration and ensures the data is handled securely. It also supports real-time updates through Firestore which allows for data synchronization when internet is available. Firebase has a console that has monitoring tools allowing us to track app usage and performance.

3.1.3 Kaggle

To build our initial recipe dataset, we used data from Kaggle, a platform with various datasets, we chose the "Food Ingredients and Recipes Dataset with Images" dataset (Sashi Goel, 2019) with license CC BY-SA 3.0. This allowed us to quickly get a collection of recipes which were later enhanced using the Edamam API for nutritional information.

3.1.4 Edamam API

We used Edamam API to get accurate and detailed nutritional information for our recipe data which will be needed for fihltering and search functionality. The API provided various nutritional information such as diet labels, calories and macronutrients.

3.1.5 Python and Java

We used Java and Python to preprocess the recipe dataset before sending it to Firebase. Python was used to clean ad format the data into an appropriate format, so all our data entries were consistent. Java was used for automating the uploading process for images and recipes data to the Firebase and Firestore which saved a lot of time.

3.1.6 Dart and Flutter

We chose flutter and Dart to develop our application because of its ability to create native like applications for various platforms from a single codebase. Flutter also has a rich set of widgets and customizable components which provide a lot of flexibility when it comes to building visually appealing user interface.

3.1.7 Figma

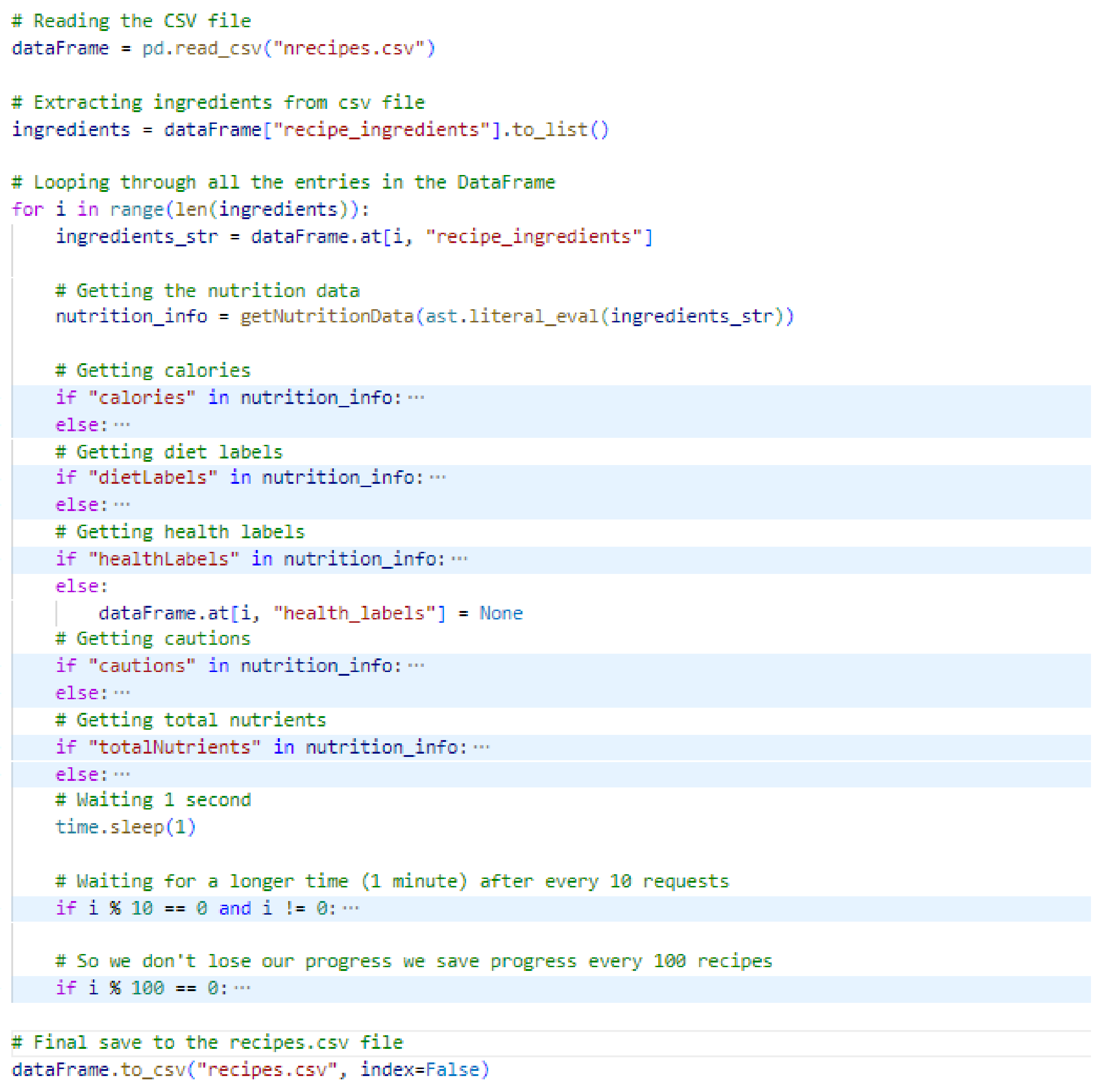
Figma was used to create the interactive prototypes for the app. The designs were shared and used to guide our development.

3.1.8 ChatGPT

ChatGPT was used to create the logo, assist with general programming questions and colour palette queries.

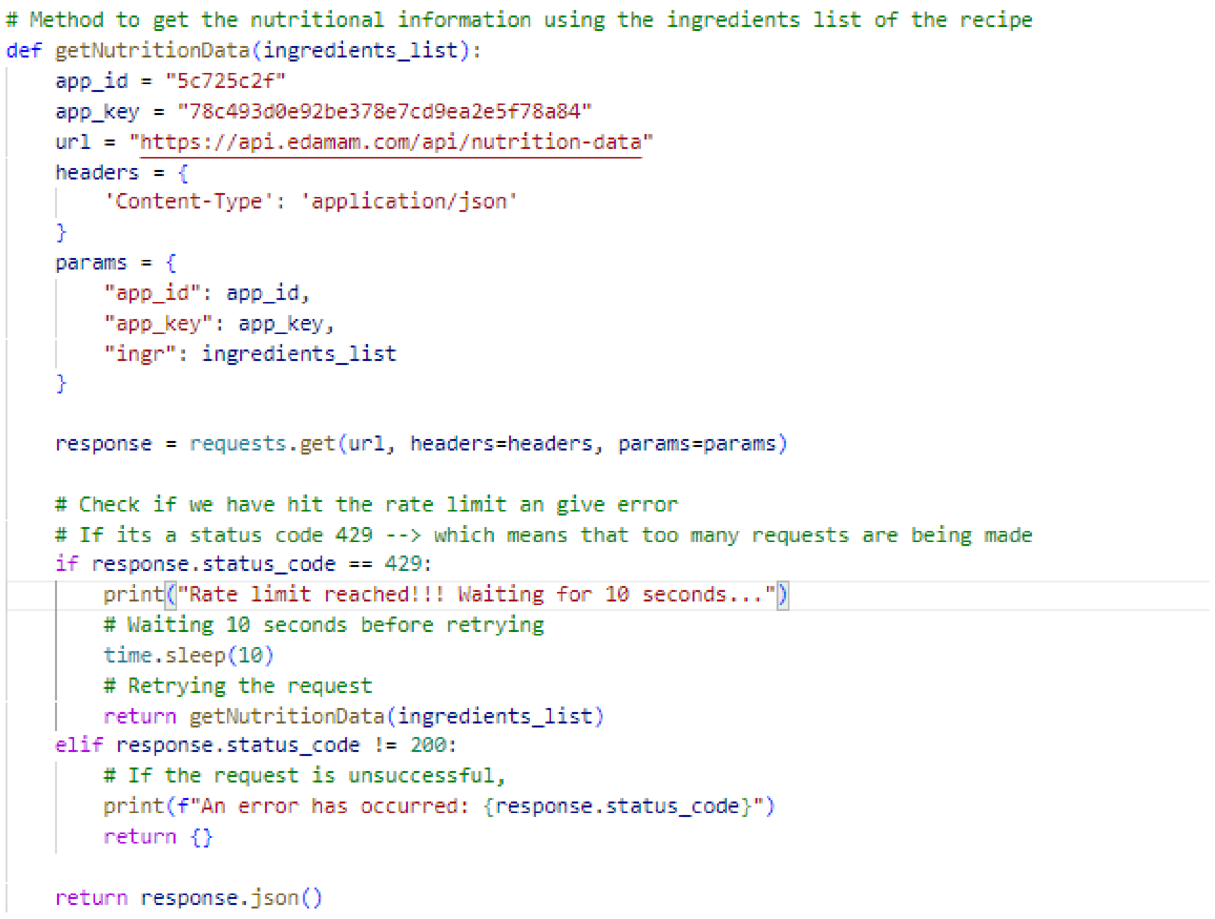
3.2 Extracts of complex code

Figure 3.1 shows a loop that goes through all entries in the dataframe (csv file) and for every recipe ingredient list, the method "GetNutritionData" shown in Figure 3.2 is called which makes a request to the Edamam API and passes through the ingredient list (first converted to a literal string) and gets the nutritional information of the recipe. The API provides the calories, diet labels, health labels and total nutrients. To not hit the rate limit set by Edamam, the "GetNutritionData" method waits 10 seconds before retrying a request. For further protection the program also waits an additional second after every request and 1 minute after every 10 requests. The omitted if statements were each adding the specific nutritional information (calories, dietLabels, etc) to their respective column in the dataframe. Since there were cases where the program would fail halfway through, the dataframe gets saved to a separate csv file after every 100 requests to the API. The method ends off by doing a final save of all the recipe data to the recipes.csv file.



```python
# Reading the CSV file
dataFrame = pd.read_csv("nrecipes.csv")

# Extracting ingredients from csv file
ingredients = dataFrame["recipe_ingredients"].to_list()

# Looping through all the entries in the DataFrame
for i in range(len(ingredients)):
    ingredients_str = dataFrame.at[i, "recipe_ingredients"]

    # Getting the nutrition data
    nutrition_info = getNutritionData(ast.literal_eval(ingredients_str))

    # Getting calories
    if "calories" in nutrition_info:···
    else:···
    # Getting diet labels
    if "dietLabels" in nutrition_info:···
    else:···
    # Getting health labels
    if "healthLabels" in nutrition_info:···
    else:
        dataFrame.at[i, "health_labels"] = None
    # Getting cautions
    if "cautions" in nutrition_info:···
    else:···
    # Getting total nutrients
    if "totalNutrients" in nutrition_info:···
    else:···
    # Waiting 1 second
    time.sleep(1)

    # Waiting for a longer time (1 minute) after every 10 requests
    if i % 10 == 0 and i != 0:···

    # So we don't lose our progress we save progress every 100 recipes
    if i % 100 == 0:···

# Final save to the recipes.csv file
dataFrame.to_csv("recipes.csv", index=False)
```

Figure 3. 1: Code that saves relevant nutritional data to a csv file



```python
# Method to get the nutritional information using the ingredients list of the recipe
def getNutritionData(ingredients_list):
    app_id = "5c725c2f"
    app_key = "78c493d0e92be378e7cd9ea2e5f78a84"
    url = "https://api.edamam.com/api/nutrition-data"
    headers = {
        'Content-Type': 'application/json'
    }
    params = {
        "app_id": app_id,
        "app_key": app_key,
        "ingr": ingredients_list
    }

    response = requests.get(url, headers=headers, params=params)

    # Check if we have hit the rate limit an give error
    # If its a status code 429 --> which means that too many requests are being made
    if response.status_code == 429:
        print("Rate limit reached!!! Waiting for 10 seconds...")
        # Waiting 10 seconds before retrying
        time.sleep(10)
        # Retrying the request
        return getNutritionData(ingredients_list)
    elif response.status_code != 200:
        # If the request is unsuccessful,
        print(f"An error has occurred: {response.status_code}")
        return {}

    return response.json()
```

Figure 3. 2: A method that requests nutritional information from Edamam API

Figure 3.3 shows the formatIngredient method. The formatIngredient method is used to extract key details from an ingredient string from a recipe. This method finds the quantities, units, and the ingredient name from text strings that come in various formats, such as “1/2 cup sugar “or “200g flour”. This method makes use of two regex expressions (shown in Figure 3.4 and Figure 3.5) to match the ingredients to known quantity and unit patterns

The method starts by removing any extra inverted commas and spaces from the ingredient string. If the ingredient includes both a quantity and a unit (“2 cups” or “½ teaspoon”), this pattern is caught by the “quantity_unit_pattern” regular expression. Any fractions are then converted into proper numbers by the replaceFraction method for easier storage and later calculations.

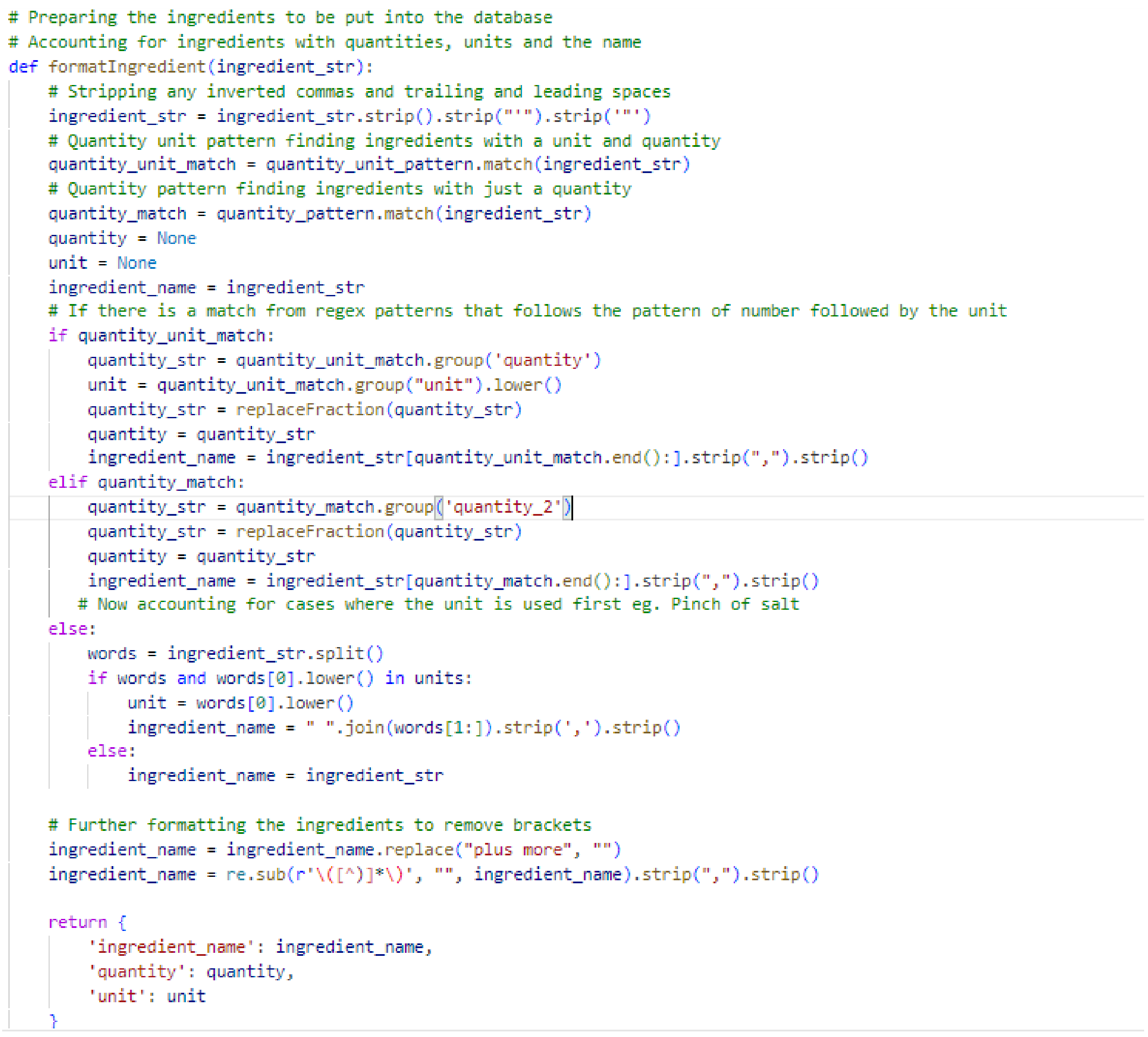
If a unit is not found, the method searches for just a quantity using the “quantity_pattern” which finds ingredient strings like “3 eggs”.

If there is no quantity and unit present in cases such as “pinch of salt”, the method checks if the first word in the string is a known unit such as “pinch” or “dash” and then extracts the ingredient name from the remaining words in the string disregarding the text before like “of”.

Finally, the method removes any extra information such as text in brackets like “¼ cup (60 g) mayonnaise” and replaces cases of “plus more” with empty space, leaving only important ingredient data which will be added to the database.

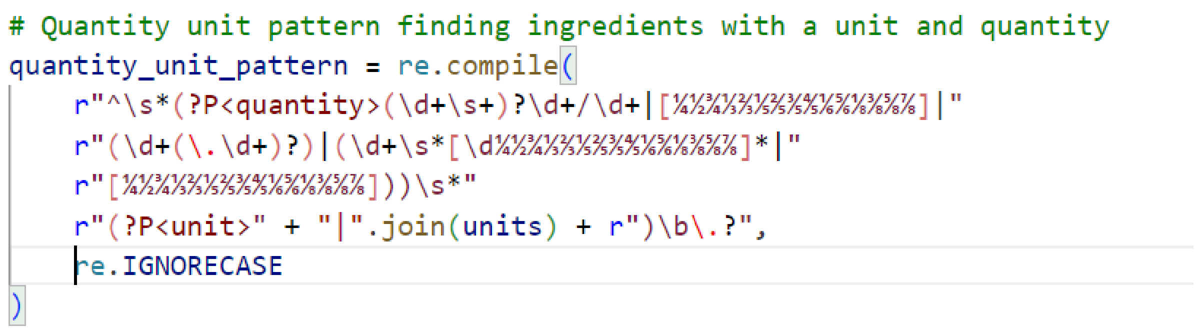
Figure 3.4 shows the “quantity_unit_pattern” which is responsible for identifying ingredient strings that contain both a quantity and a unit. It is flexible enough to match common formats like "1/2 cup," "100 g," or even fractions like "¾ teaspoon." The pattern captures number quantities and fractions and matches units from a predefined list ("cup," "teaspoon," "gram", etc.).

Figure 3.5 shows the “quantity_pattern” which is used for cases where the ingredient has a quantity but no unit ("2 eggs"). This pattern finds fractions and number quantities, ensuring that even ingredients like "1/4" or "½" are correctly extracted.



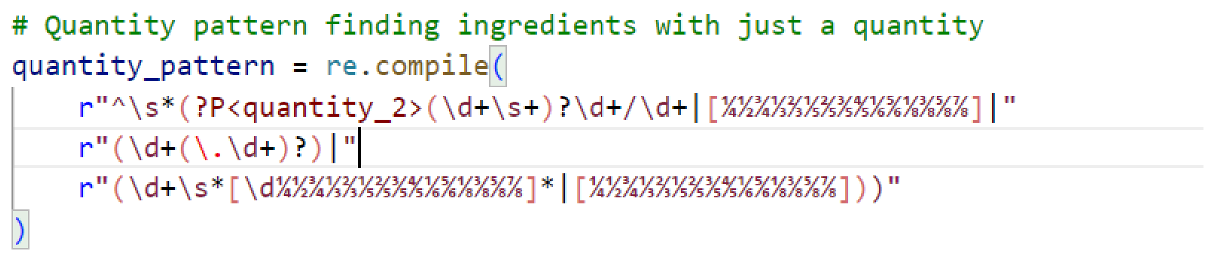
```python
# Preparing the ingredients to be put into the database
# Accounting for ingredients with quantities, units and the name
def formatIngredient(ingredient_str):
    # Stripping any inverted commas and trailing and leading spaces
    ingredient_str = ingredient_str.strip().strip("'").strip('"')
    # Quantity unit pattern finding ingredients with a unit and quantity
    quantity_unit_match = quantity_unit_pattern.match(ingredient_str)
    # Quantity pattern finding ingredients with just a quantity
    quantity_match = quantity_pattern.match(ingredient_str)
    quantity = None
    unit = None
    ingredient_name = ingredient_str
    # If there is a match from regex patterns that follows the pattern of number followed by the unit
    if quantity_unit_match:
        quantity_str = quantity_unit_match.group('quantity')
        unit = quantity_unit_match.group("unit").lower()
        quantity_str = replaceFraction(quantity_str)
        quantity = quantity_str
        ingredient_name = ingredient_str[quantity_unit_match.end():].strip(",").strip()
    elif quantity_match:
        quantity_str = quantity_match.group('quantity_2')
        quantity_str = replaceFraction(quantity_str)
        quantity = quantity_str
        ingredient_name = ingredient_str[quantity_match.end():].strip(",").strip()
       # Now accounting for cases where the unit is used first eg. Pinch of salt
    else:
        words = ingredient_str.split()
        if words and words[0].lower() in units:
            unit = words[0].lower()
            ingredient_name = " ".join(words[1:]).strip(',').strip()
        else:
            ingredient_name = ingredient_str

    # Further formatting the ingredients to remove brackets
    ingredient_name = ingredient_name.replace("plus more", "")
    ingredient_name = re.sub(r'\([^)]*\)', "", ingredient_name).strip(",").strip()

    return {
        'ingredient_name': ingredient_name,
        'quantity': quantity,
        'unit': unit
    }
```

Figure 3. 3: A method that formats ingredients from recipes



```python
# Quantity unit pattern finding ingredients with a unit and quantity
quantity_unit_pattern = re.compile(
    r"^\s*(?P<quantity>(\d+\s+)?\d+/\d+|[¼½¾⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞]|"
    r"(\d+(\.\d+)?)|(\d+\s*[\d¼½¾⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞]*|"
    r"[¼½¾⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞]))\s*"
    r"(?P<unit>" + "|".join(units) + r")\b\.?",
    re.IGNORECASE
)
```

Figure 3. 4: Regex pattern that matches ingredients with quantities that are followed by units



```python
# Quantity pattern finding ingredients with just a quantity
quantity_pattern = re.compile(
    r"^\s*(?P<quantity_2>(\d+\s+)?\d+/\d+|[¼½¾⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞]|"
    r"(\d+(\.\d+)?)|"
    r"(\d+\s*[\d¼½¾⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞]*|[¼½¾⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞]))"
)
```

Figure 3. 5: Regex pattern that matches ingredients with just a quantity and no unit

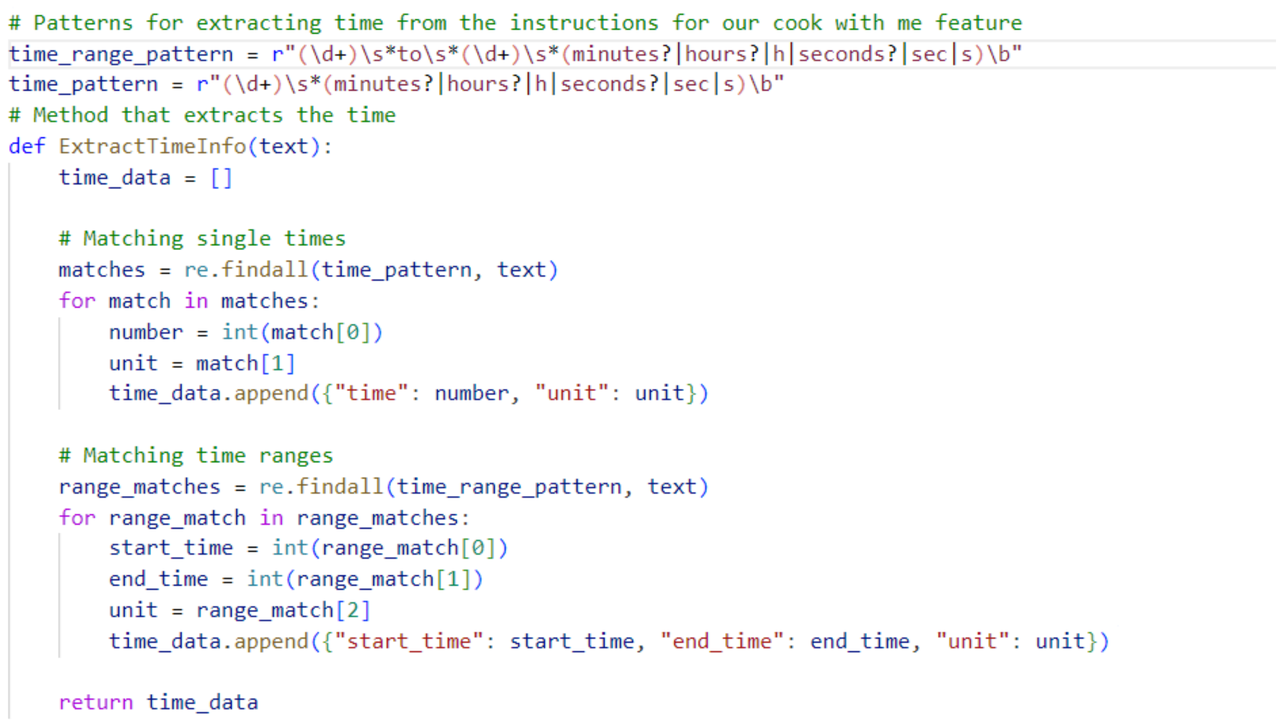
Figure 3.6 shows the ExtractTimeInfo method which is used to extract time-related data from the recipe instructions. This is done for the “Cook with me” feature of our app, which will help users follow recipes with built in timers.

The method uses two regex expressions which are also shown in Figure 3.7 and Figure 3.8 to identify both single times and time ranges ("bake for 20 minutes" or "simmer for 5 to 10 minutes").

First, the method looks for individual time mentions using the “time_pattern”, which matches time units such as "minutes," "hours," or "seconds" along with their corresponding numeric values ("20 minutes" or "1 hour"). Each time match is converted into an integer and stored in a dictionary, which contains the time value and the unit, and this is added to the “time_data” list.

Next, the method looks for ranges of time using the “time_range_pattern”, which captures expressions like "5 to 10 minutes" or "1 to 2 hours." Both the start and end times are extracted and converted into integers, with the unit identified. This data is then added to the time_data list as well.

The method returns a list of dictionaries, where each dictionary contains either a single time or a time range with its corresponding unit, allowing the app to create timers based on the recipe’s instructions.

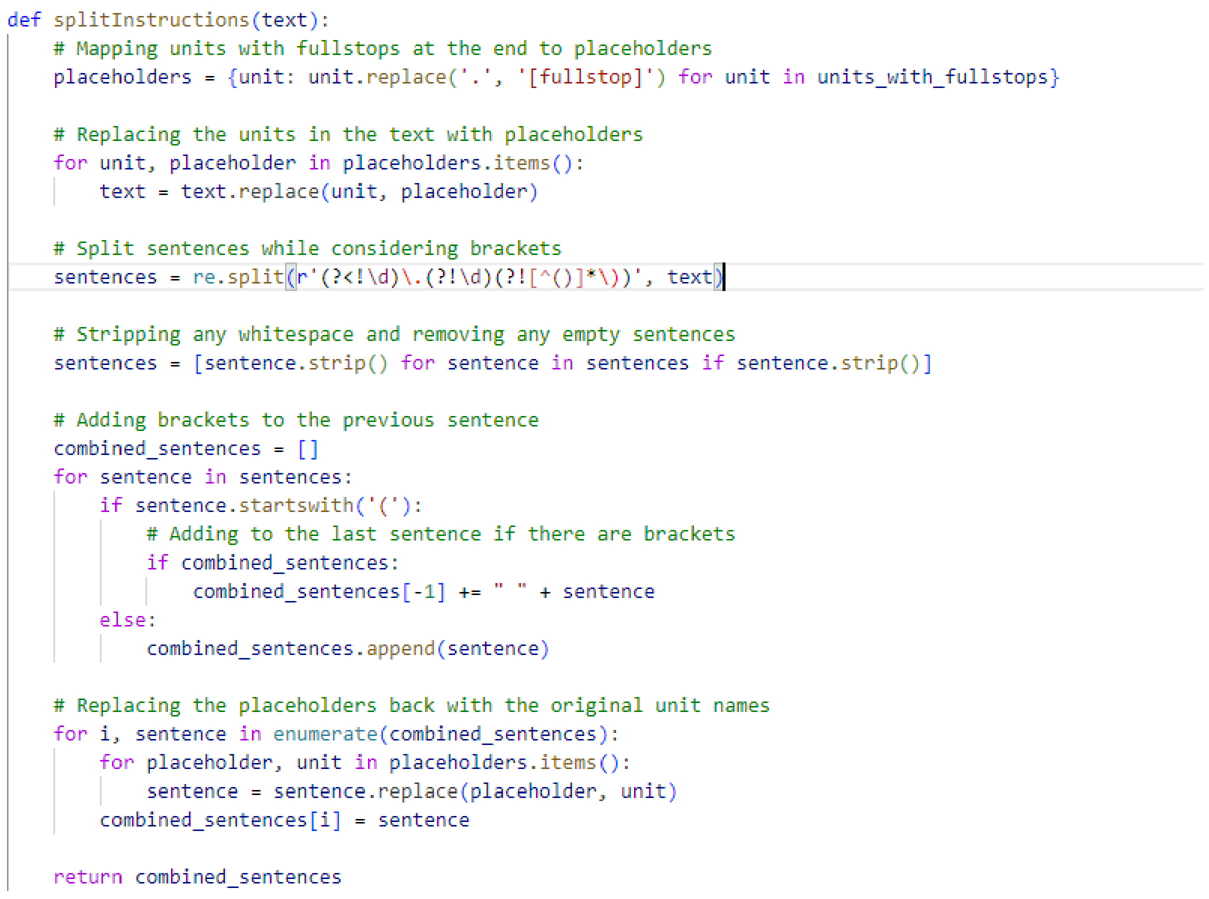


```python
# Patterns for extracting time from the instructions for our cook with me feature
time_range_pattern = r"(\d+)\s*to\s*(\d+)\s*(minutes?|hours?|h|seconds?|sec|s)\b"
time_pattern = r"(\d+)\s*(minutes?|hours?|h|seconds?|sec|s)\b"
# Method that extracts the time
def ExtractTimeInfo(text):
    time_data = []

    # Matching single times
    matches = re.findall(time_pattern, text)
    for match in matches:
        number = int(match[0])
        unit = match[1]
        time_data.append({"time": number, "unit": unit})

    # Matching time ranges
    range_matches = re.findall(time_range_pattern, text)
    for range_match in range_matches:
        start_time = int(range_match[0])
        end_time = int(range_match[1])
        unit = range_match[2]
        time_data.append({"start_time": start_time, "end_time": end_time, "unit": unit})

    return time_data
```

Figure 3. 6: A method that extracts the time from the instructions of a recipe

Figure 3.7 shows the SplitInstructions method which is used to separate the recipe instructions into sentences while also ensuring that the units and brackets are handled correctly. The method starts by using a predefined list of unit (grams, litres,etc) called “units_with_fullstops” to create a placeholder for those type of units, this is done to prevent them from being seen as a sentence break.

The method then checks for any text that has such units with fullstops. The regex expression is then used to divide the text into sentences correctly. The method then handles any cases where the sentences start with bracket. This ensures instructions like “Mix ingredients (including the spices)” stay together. The method then replaces all the placeholders with the original unit names then returns the modified sentences as a list.



```python
def splitInstructions(text):
    # Mapping units with fullstops at the end to placeholders
    placeholders = {unit: unit.replace('.', '[fullstop]') for unit in units_with_fullstops}

    # Replacing the units in the text with placeholders
    for unit, placeholder in placeholders.items():
        text = text.replace(unit, placeholder)

    # Split sentences while considering brackets
    sentences = re.split(r'(?<!\d)\.(?!\d)(?![^()]*\))', text)

    # Stripping any whitespace and removing any empty sentences
    sentences = [sentence.strip() for sentence in sentences if sentence.strip()]

    # Adding brackets to the previous sentence
    combined_sentences = []
    for sentence in sentences:
        if sentence.startswith('('):
            # Adding to the last sentence if there are brackets
            if combined_sentences:
                combined_sentences[-1] += " " + sentence
        else:
            combined_sentences.append(sentence)

    # Replacing the placeholders back with the original unit names
    for i, sentence in enumerate(combined_sentences):
        for placeholder, unit in placeholders.items():
            sentence = sentence.replace(placeholder, unit)
        combined_sentences[i] = sentence

    return combined_sentences
```

Figure 3. 7: A method that splits instructions into list of strings

3.3 Source code references

3.3.1 Flutter and Firebase Tutorial

Description

To learn more about the dart programming language, flutter and the fundamentals of firebase we followed a course created by Vandad Nahavandipoor.

Usage

- Flutter setup
- Login View
- Email Verification
- Firebase backend setup
- Auth service
- Migration to Firestore Service
- Bloc for routing and dialogs

Reference

freeCodeCamp.org (Director). (2022, February 24). *Flutter Course for Beginners – 37-hour Cross Platform App Development Tutorial* [Video recording]. https://www.youtube.com/watch?v=VPvVD8t02U8

3.3.2 Implementing a grid view UI

Description

A grid view was implemented to display and organize the UI elements efficiently. The GridView.builder was used to structure the layout dynamically.

Usage

The grid view is used to display the saved recipes on the Saved Recipes screen.

Reference

Mitch Koko (Director). (2022, November 4). *Donut App UI • Flutter Tutorial* [Video recording]. https://www.youtube.com/watch?v=OmYL-VK75-o

3.3.3 Implementing Bottom Navigation Bar

Description

To be able to navigate to different screens in the app, a navigation bar was needed that followed the Material Design 3 guidelines.

Usage

Used for the main navigation of the app

Reference

BottomNavigationBar class—Material library—Dart API. (n.d.). Retrieved 18 October 2024, from https://api.flutter.dev/flutter/material/BottomNavigationBar-class.html

3.3.5 Uploading recipe data to Firebase Firestore

To upload formatted recipe data from a CSV file to Firebase Firestore, the recipe ingredients, instructions, and nutritional information had to be extracted and formated to ensure that the data structure follows the Firebase's NoSQL schema. The code includes error handling to avoid data corruption during parsing, and it formats the data correctly before saving it to Firestore.

Usage

To upload recipe data (ingredients, instructions, calories etc) to the Firestore database.

Reference

ChatGPT (2024). Code assistance for formatting and uploading recipe data to Firebase Firestore. Retrieved October 17, 2024, from https://chatgpt.com.

3.4 Problems encountered

3.4.1 Creating a Flutter App

To create a cross-platform app we had to find a framework that would allow us to do so efficiently so we chose Flutter. We had never used Flutter and dart before, and the learning curve was quite steep

However, we found a 37-hour long video on YouTube that really helped us to gain insight on how to use Flutter and code an app using dart. The tutorial allowed us to learn and use that knowledge to start developing the app.

3.4.2 Using Firebase

Having only worked with relational databases before, it was quite an adjustment to work with Firebase and to understand how it is structured as well as the difference in formats. Watching some YouTube videos and playing around on Firebase helped us understand collections and documents better.

3.4.3 Key Down Event error

There was an issue with how keyboard events were being handled in the search field. This occurred because there was a conflict with the listener and input handling. Online forums suggested upgrading flutter since it is a common issue with how flutter handles physical keystrokes, but it seemed too risky. It was solved by having a check on the search controller text. Also, the error would loop because the text editing controller was being created every time the widget was built, so the search controller that was declared was called instead to fix it.

3.4.4 Bottom Overflow Error

Occasionally we encountered an error that would pop up whenever a text field was pressed and brought up the keyboard which was the bottom overflow by x pixels.

This error comes about when the screen is made to be static and then an element such as the keyboard will push the content below it off the screen. To fix this we wrapped the widget that is shown under a Single Child Scroll View which allows the screen to be scrollable.

3.4.5 Right Overflow Error

When designing the UI elements for the saved recipe screen there was a right overflow error as seen in Figure 3.8. This occurred when using the GridView.builder widget to allow two items per row. Causing an overflow due to the items overlapping.

To fix this error we needed to ensure the width of the grid items do not overflow by using a combination of crossAxisSpacing and padding. To avoid a bottom overflow error for the grid items we used mainAxisSpacing as well. The aspect ratio of the grid items was also changed so that all contents could be fitted inside.

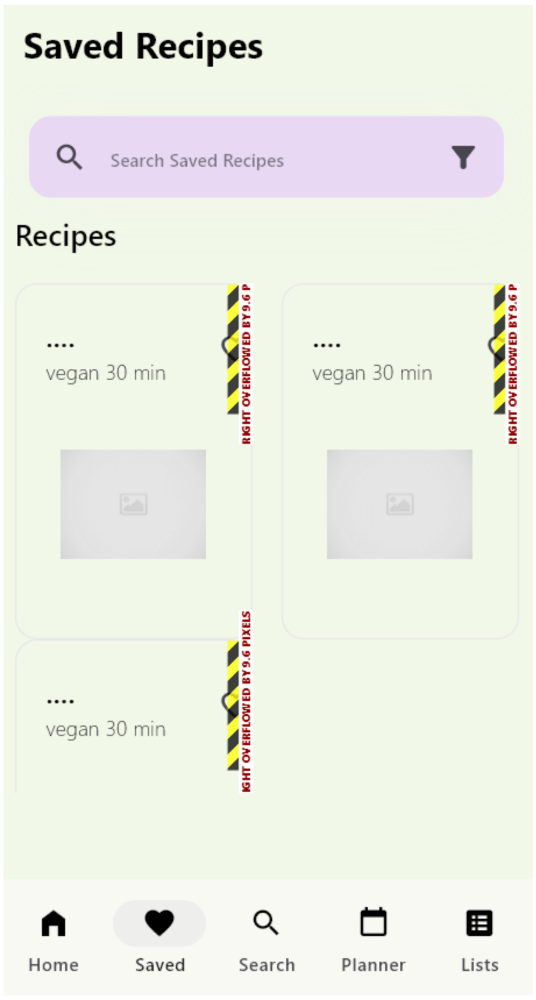


Figure 3. 8: A screenshot that demonstrates the right overflow error

3.4.6 Rate limit and requests limit

When working on getting nutritional information from the Edamam API, we encountered an issue which was causing the requests to fail. Because we had so many recipes in our original csv file, Edamam could not handle so many requests per second. To fix this we had to limit the number of requests to 10 then make the process sleep for 1 – 10 seconds. We encountered a similar problem when uploading the recipes data and images to Firebase Firestore which was fixed in a similar way. Edamam also had a limit on the number of requests we could make to the API per month, so we had to cut our original dataset of 13000 recipes to only 3000 recipes.

3.4.7 Stack Overflow Error

On the search screen where we were testing a placeholder model for displaying data, we got a stack overflow error shown in Figure 3.9. There were no errors highlighted in the code, but the debug console showed it was occurring because of the model class.

The problem occurred because it was calling the getRecipe() method, which caused an infinite loop of the data being displayed. To fix the issue the recipe list itself was returned.

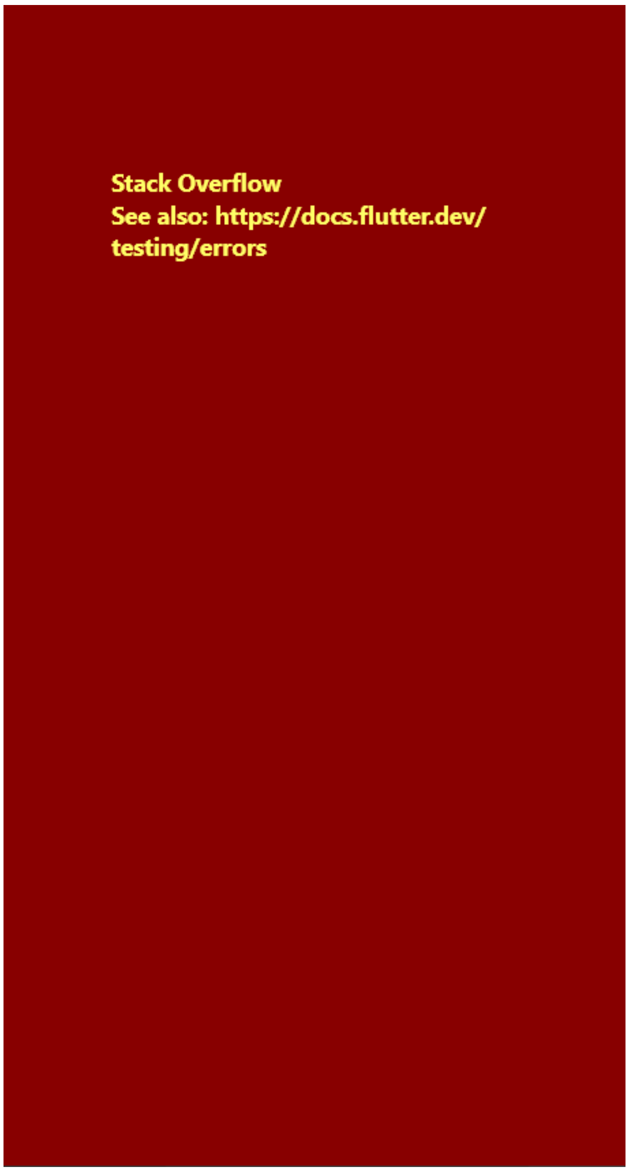

```
Stack Overflow
See also: https://docs.flutter.dev/
testing/errors
```

Figure 3. 9: A screenshot of the stack Overflow error on the search screen

3.4.8 Formatting Recipe ingredients and instructions

The recipe ingredients and instructions were in format that was not very useful for our objectives such as the ingredients didn't have the quantities, unit and the name of the ingredient laid out very nicely and the instructions did not have times specified which is needed for our "cook with me" feature. We sorted this out by implementing a formatting script that would go through all the recipes and extract such information before being put into the database.

3.4.9 Regex Implementation

Since we had never worked with regex before it was quite challenging to get a handle on how it works. After some digging, a regex cheat sheet really came in handy when it came to figuring out how to capture certain data (*Regex Cheat Sheet*, n.d.).

References

Regex Cheat Sheet. (n.d.). Retrieved 18 October 2024, from https://www.rexegg.com/regex-quickstart.php

Sashi Goel, A. D. (2019, February 19). *Food Ingredients and Recipes Dataset with Images*. Retrieved from Kaggle: https://www.kaggle.com/datasets/pes12017000148/food-ingredients-and-recipe-dataset-with-images