# *Munchmice*: The All-in-one Recipe Cookbook and Planner for Students

CS310 Project Final Report

**Katie Yang**

Supervisor: Dr Claire Rocks

Year of Study: 3

2022-2023

# Abstract

Home cooks are finding themselves looking for recipes online now more than ever before. A large part of this group will be younger people like students who are more accustomed with this method of looking up recipes, rather than hard copies of cookbooks. When cooking, students face a variety of challenges, one of which is budgeting - a problem that does not seem to be addressed particularly well by existing systems. To this end, Munchmice is a recipe management web application designed to extract and save recipes from the web, as well as providing an automatic cost estimation function for ingredients and an in-built shopping list, combined with providing other useful functionalities to improve the student cooking experience as a whole. The final recipe cost estimations utilise price data collected from Trolley [1], a grocery price lookup site, and produce results accurate to average price groceries in the UK. Munchmice brings with it promising recipe import and price estimation functions, as well as an intuitive and pleasant design, making it a valuable tool for aiding student cooking. As the market has few recipe applications exploring this area, Munchmice creates significant possibilities for future development.

**Keywords** Recipe management, Web Development, Javascript, Web scraping, Information extraction, Natural language processing

# Acknowledgements

I would like to extend my sincere thanks to Dr Claire Rocks, who provided guidance and support throughout the development of this project. I am also grateful to everyone who responded to the anonymous survey for Munchmice. Many thanks to Ewan Moran, who gave me support and patience throughout the duration of this project, as well as Anthony Yang for their faith and encouragement. Lastly, thanks should also go to my friends Raumaan Ahmed, Jacob Coates, and Manas Rawat for their helpful advice and support.

# Contents

# Chapter 1

# Introduction

As a low-to-no income group, as well as typically being inexperienced cooks, cooking as a student presents a variety of unique challenges. One of the main problems when trying to eat healthier and cooking at home is finding a meal plan that fits into budget. However, cost is not often readily accessible information, especially when it comes to finding new recipes. Cooking your own food has been shown to have positive effects on mental health [2], with the process encouraging creativity, a healthy lifestyle, and even social connection. Furthermore, cooking and trying a variety of new recipes requires stepping out of one's comfort zone and can take a great amount of effort to do on a regular basis, especially given student schedules and the easy accessibility of ready-meals and takeaways.

With the above in mind, this project aims to achieve two particular objectives: providing an easy-to-use system that alleviates some of the work needed for inexperienced cooks to make meals regularly, and to present a solution for better budgeting of meals that does not limit the user's options to any specific set of recipes. Ultimately, the purpose of this project is to not only help budgeting, but also to encourage students to explore cooking, find healthy recipes, as well as to gain new skill sets and build healthy living habits.

This report will first cover the background of current student cooking habits as well as research on existing solutions, then use the results to outline requirements for

the project itself. The management plan of this project is discussed in the next chapter. Then, we will move on to planning the design of the system, including its architecture and how to approach the implementation of individual functionalities. The next chapter, Implementation, follows on from this and describes the technical details surrounding set up and code implementation. At this point, the initial development of the project has been completed, and we move onto testing the system and collecting results, as well as showcasing the product. A thorough evaluation of the project, its fulfilment of requirements, as well as the development is carried out with the collected results and user feedback. Lastly, we will cover any legal, social, and ethical considerations involved, conclude the project and summarise its outcomes, as well as discuss future directions for Munchmice.

# Chapter 2

# Background

The background chapter aims to give a better insight into the present state of student cooking and the current challenges that come with it. We will discuss the results of a small survey polling university students designed as part of this research, with the purpose of gaining a deeper understanding of the reasoning behind the statistics. Lastly, we look at a few existing solutions for recipe applications and compare their features in order to understand what makes a useful recipe tool, and if there are any features they may be lacking.

## 2.1 Student Cooking at University

There are numerous benefits to home cooking, a term which is defined by the general UK public in terms of "preparing a meal from scratch, cooking with love and care, and nostalgia", with perceptions often focused on "social and emotional associations" [3]. In a large, population-based cohort, eating home cooked food is associated with better quality of diet and lower adiposity [4]. This may be associated with higher nutrient value of home prepared meals, as well as the fact that meals prepared at restaurants are "significantly higher in kilocalories, fat, and saturated fat" when compared to meals prepared in other locations [5].

In a study on dietary patterns published in 2018, about 55% of students from universities across the UK reported that they were able to cook a wide variety of meals

from raw ingredients, and 73% consumed these self-cooked meals "every" or "most" days [6]. One conclusion from this study is that the "convenience, red meat & alcohol" pattern was identified most consistently across universities, but also associated with higher weekly food spending, while the "vegetarian" and "health-conscious" patterns were more cost-efficient, as well as having greater nutrient density. These healthier patterns were also associated with students reporting greater cooking ability.

Nonetheless, student cooking comes with its own challenges, one of the most significant being budgeting. Compared to the UK average expenditure on food and non-alcoholic drinks of £39.44 per person (2018/19) [7], most UK students reported a food budget of £20-29 per week, with just less than one quarter spending under £20 and one in five spending over £40 (2018) [6].

## 2.2 Primary Research

To put the background research into perspective and to gain a more detailed understanding of student cooking, a survey polling 16 second and third-year university students is conducted. The majority of respondents are third-year students from the University of Warwick, and a smaller fraction of students from the University of Oxford, University of Exeter, Imperial College, as well as University College London. It must be recognised that the sample size is small and may not be representative of the overall UK student population, but nevertheless allowed better insight into student cooking habits.

The survey is split into two sections: the first section enquires about general dietary habits and the usage of cooking applications, and the second section gathers feedback related to the design of Munchmice. In the dietary habit section of the survey, 31.3% of students responded that they cook at least three lunches per week (Figure 2.1), and 56.3% cooked at least three dinners per week (Figure 2.2).

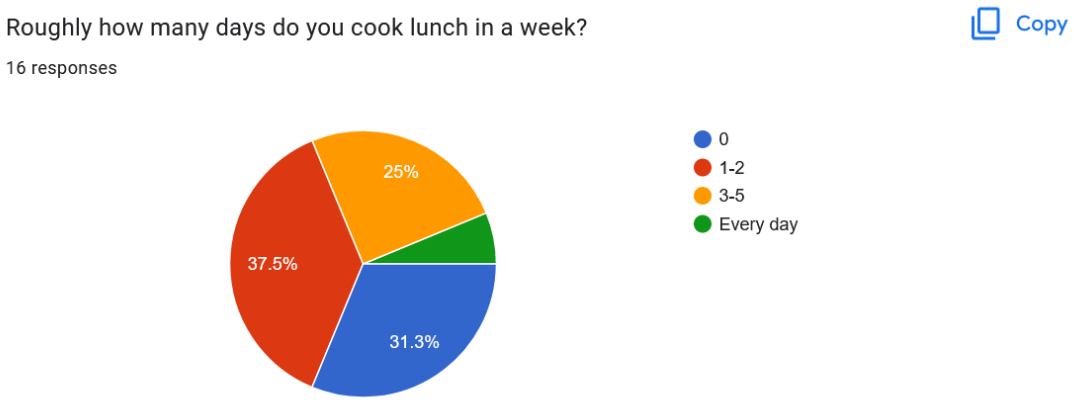


Figure 2.1: *Roughly how many days do you cook lunch in a week?*

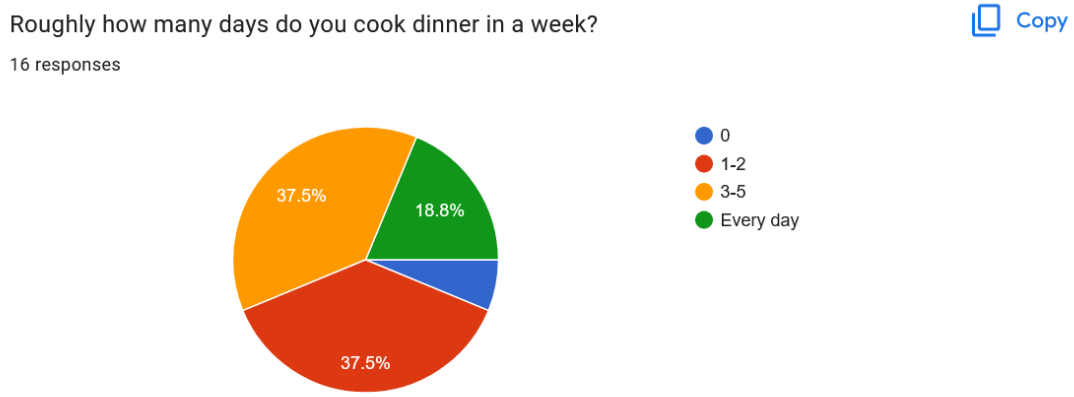


Figure 2.2: *Roughly how many days do you cook dinner in a week?*

When it comes to recipes, 87.5% of respondents said that they sometimes or often follow recipes when cooking or baking (Figure 2.3), demonstrating that a large majority of the target market has the basis to find use in a recipe application. Despite this fact, all but one respondent reported that they do not use a cooking or recipe application. This disparity is attributed to a number of reasons by participants: some have not found an existing application with recipes they like; some were not avid cooks; and some found that recipe applications added more work to meal planning, instead of the opposite. In terms of food costs, only 12.5% of respondents said that they know roughly how much they spend on each meal they cook, with 25% having no idea at all (Figure 2.4).

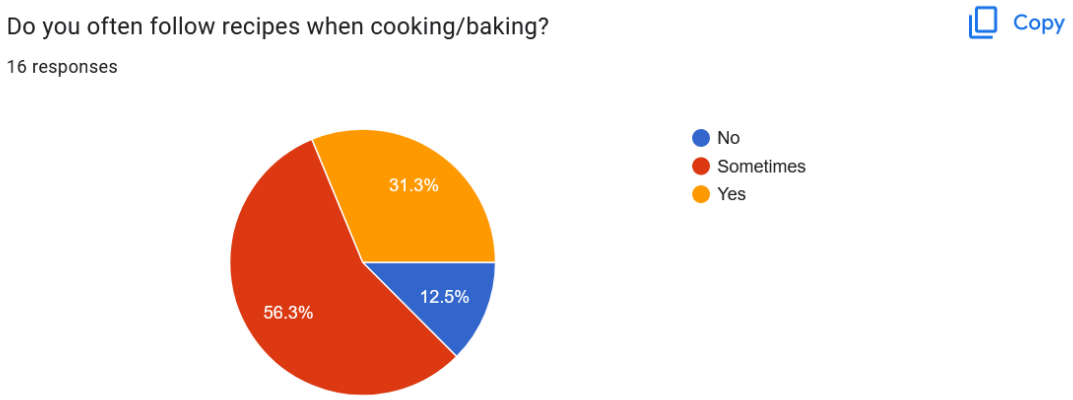


Figure 2.3: *Do you often follow recipes when cooking/baking?*

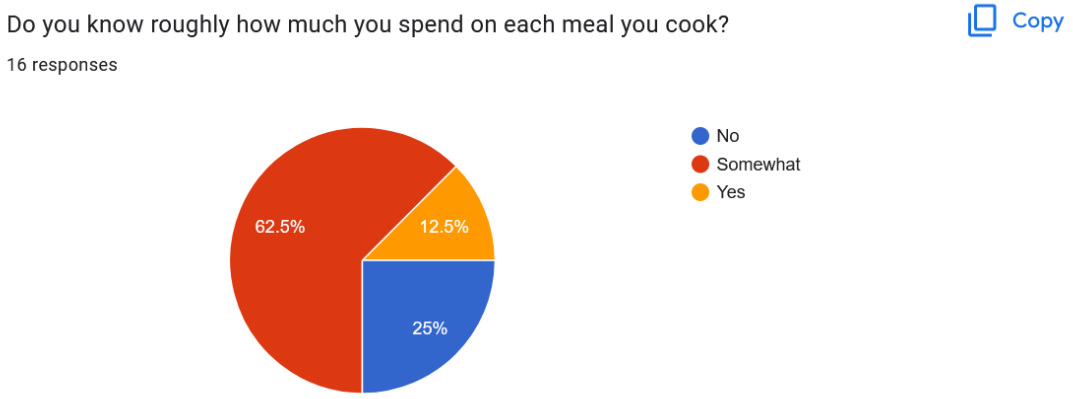


Figure 2.4: *Do you know roughly how much you spend on each meal you cook?*

Moving onto the second section, questions specific to the design of Munchmice are asked. 81.3% agreed it would be useful to have a functionality that estimates the cost of a recipe, and one respondent also suggested the inclusion of a cost per serving metric. When asked about potentially useful functions for a cooking application, many suggested a shopping list, as well as regional costs estimates and ingredient substitutes.

Further details of this survey can be found in Appendix A, and survey results about the interface design for Munchmice are discussed in the Design chapter. Drawing conclusions from both the background and primary research, it is believed that for less-skilled home cooks, cost is a limiting factor when it comes to preparing more

varied, nutritious, and tasty meals. To encourage student home cooking, it may be a worthy endeavour to introduce tools to assist in cooking to help simplify the process, as well as to help budgeting to allow a more varied meal plan.

## 2.3 Existing Solutions

With the background and target demographic in mind, it is imperative that an investigation into similar products is conducted. The idea of a recipe management application is not novel: an abundance of tools both online and offline already exists for this purpose. Though this project has a focus on price estimation as well as recipe management, it is in the interest of functionality comparison to assess features of currently popular recipe tools. As a result, it may help put into perspective the scope of the project, as well as helping to steer the initial requirement engineering. Before analysing specific examples, we define three broad categories of recipe applications relevant to our project, each focusing on one of three aspects: discovery, management, and price estimation.

**Recipe discovery** Discovery applications are the most well-known and widely used tools, with examples such as BBC Good Food [8], Yummly [9], and Allrecipes [10]. These applications typically have a large set of original recipes that users can browse through and take inspiration from.

**Recipe Management** These applications usually do not provide any original recipes. Instead, they focus on the organisation of recipes and user personalisation. These applications are typically used by those who have or are building a collection of recipes gathered from different sources, and at minimum want to keep track of those recipes, but imported recipes can also be customised. Examples of recipe management applications include Recipe Keeper [11], Saffron [12], and Paprika [13].

**Recipe Cost Estimation** Price estimation applications are much more uncommon than the aforementioned categories. The vast majority of recipe cost estimation tools that were found as part of the investigation are targeted towards small

businesses or restaurant owners, designed to estimate batch ingredient costs and to keep track of profit margins. One example of this is Recipe Cost Calculator [14] (shown in Figures 2.5 and 2.6), although other applications were found to be very similar in functionality. While these tools can be highly personalised to ensure excellent ingredient price estimation accuracy, they require the user to first know the exact price of their ingredients, as they are targeted for repeated calculation with the same ingredients. This is not appropriate for personal cooking, and not useful for recipe experimentation. In fact, there seems to be little in the way of personal recipe cost estimation tools, and no examples were found during research.


Figure 2.5: Recipe Cost Calculator cost breakdown interface.

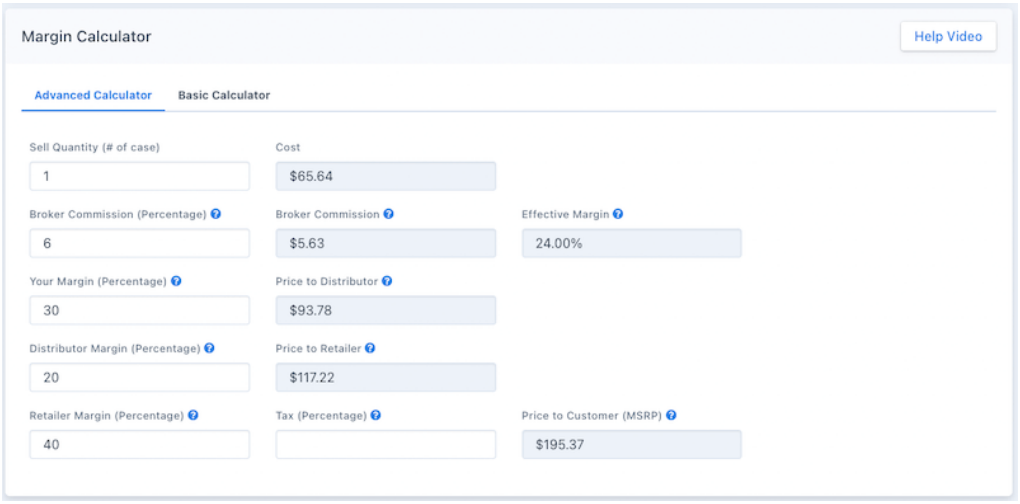


Figure 2.6: Recipe Cost Calculator margin calculator interface.

Two recipe applications are dissected further below. It should be noted that the applications discussed are non-exhaustive: there are a myriad of recipe applications on the market that all come with a range of functionalities. However, the examples selected were deemed to be the most popular and representative, as well as functionally similar to our project.

### 2.3.1 Yummly

Yummly [9] is a recipe and cooking platform that provides personalised recipe recommendations based on a range of information about the user. This includes but is not limited to: cooking skill, dietary preferences, and favourite cuisines. Hosting over 2 million recipes with a user count of 23 million, it is one of the most popular online recipe platforms currently available [15]. Shown in Figure 2.7, Yummly is interesting for the recipes it provides: not just limited to hosting original recipes, it partners with various blogs and external recipe sites to provide the user with a huge selection. This does somewhat come at the cost of a clear layout, however, as it is often required for the user to click through to the original site in order to read recipe directions.

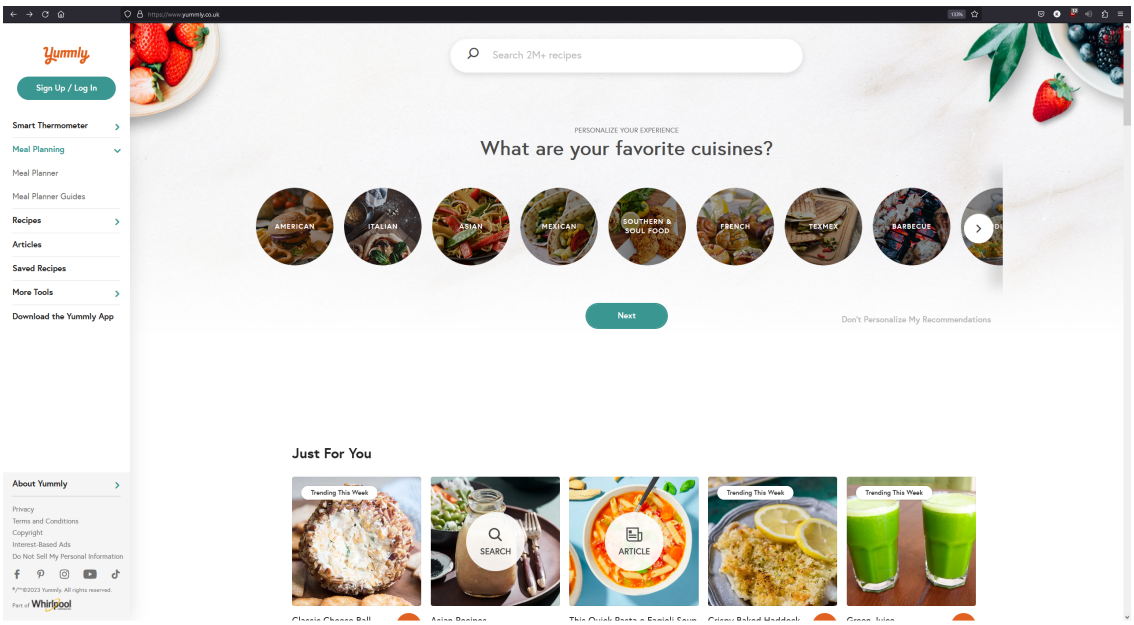


Figure 2.7: Yummly homepage.

Nevertheless, Yummly spans across both the recipe discovery and recipe management categories listed above, as it is packed with useful functionalities. On top of allowing users to search and save recipes, create shopping lists, and watch step-by-step instruction videos, it also provides a meal planning function. However, many of Yummly's useful features are locked behind a premium account that is available for a fee. This will not be the case for Munchmice, as even if it was to be released onto the market, its purpose is to help users budget and save money. Hence, it does not make sense to charge a fee to access features of the application.

The lack of an ingredient price estimation function in Yummly is not unreasonable. As a platform used by millions worldwide, it would need to have access to up-to-date grocery data spanning many countries. As a recipe site, it is a sizable endeavour to explore this possibility. Munchmice, on the other hand, is limited to just the UK, where it suffices to have one standardised set of data, though this might not be as accurate for some locations, such as London.

### 2.3.2 Paprika

Unlike Yummly, Paprika [13] is solely a recipe management application. Providing the functionality for users to import recipes from the web, as well as allowing the user to edit them, Paprika is similar to the planned recipe management aspect for Munchmice. The layout of its interface, shown in Figure 2.8, is simple and functional - an important design given the potential of a large number of recipes to be displayed. Similar to Yummly, it also provides a shopping list function, as well as a meal planning tool.

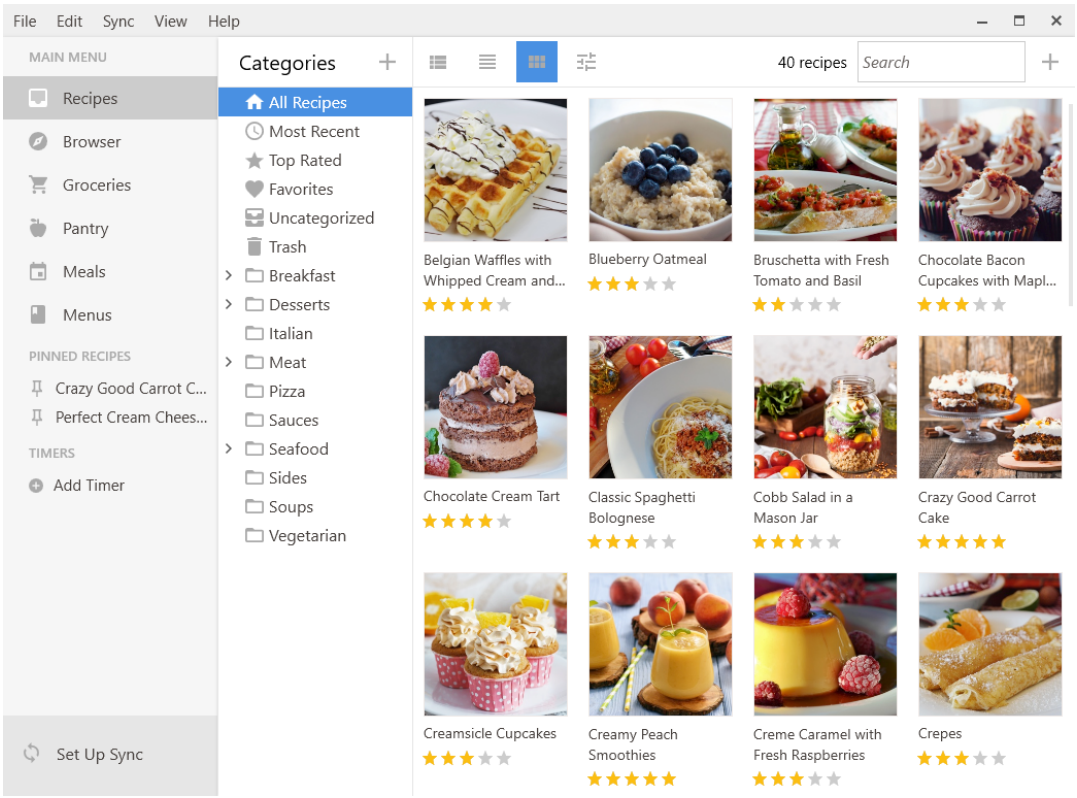


Figure 2.8: Paprika recipe page.

At the time of writing, Paprika is available on Windows for $29.99. While this may be suitable for experienced home cooks managing a large amount of recipes, it is not a realistic commitment for those who are just starting out cooking. In fact, it does appear that Paprika's target market are those who are already cooking enthusiasts that required a great recipe management tool. On top of this, Paprika is a software that needs to be separately downloaded to different devices, which adds more difficulty and resistance to the setup of the application.

### 2.3.3 Summary of Findings

Through market research of existing solutions, it is found that current tools for recipe management, though full of content and feature-heavy, do not provide budgeting features such as ingredient cost estimation. Yummly is focused on the discovery of recipes, and is highly personalised; Paprika is a great example of a recipe management app, and its design is worth noting. Together with another recipe management application Recipe Keeper and Recipe Cost Calculator, the table in

Figure 2.9 has been organised to better showcase the features present in current systems.

| | Original Recipes | Recipe Import | Meal Planning | Recipe Modification | Recipe Collections | Shopping List | Cost Estimation |
|---|---|---|---|---|---|---|---|
| Yummly | ✓ | ✓ | ✓ | | ✓ | ✓ | |
| Paprika | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Recipe Keeper | | ✓ | ✓ | ✓ | | ✓ | |
| Recipe Cost Calculator | | | | | | | ✓ |

Figure 2.9: Comparison of core functionalities between popular recipe applications.

From the table it is clear to see that there is no overlap between applications that tackle recipe management and cost estimation. Furthermore, it is evident that even when looking in the greater market outside of the context of Yummly and Paprika, cost estimation for personal recipes is not an area explored by many, especially cost estimation systems with in-built cost data that does not require users to manually set ingredient prices. Hence, the primary aim of Munchmice is to fill this gap by tackling automatic cost estimation of recipes, as well as providing a useful array of other recipe management tools, the most significant of which being recipe import and recipe edit.

# Chapter 3

# Requirements

We have previously outlined the main objectives of the project, but it is important to analyse the requirements in more detail. This will not only bridge the gap between system design and implementation, but also provide a basis to build a model of the system, as well as giving a broad development road map. For this project, we first discuss a set of core requirements, which later are developed and documented into a formal requirement specification.

The six initial core requirements were formulated keeping in mind the main aim of the project, the opinions gathered from the survey, as well as functionalities of the existing solutions that were found to be useful. These are largely functional requirements, as their purpose is to give an idea of what the final system will look like. Once this general framework is constructed, it can be built on, and specifics regarding each requirement can be detailed. These requirements are categorised in the requirement specification as the "Must" requirements.

**1. The application must be able to efficiently scrape recipe sites and correctly identify the ingredients, measurements, and cooking directions.**

- This is one of the most important recipe management functionalities, as users need to have a way of importing recipes in order to build their collection.

**2. The ingredient measurements and cooking directions must be fully modifiable, and the measurements scalable to serving size.**

- Allowing users to modify a recipe to their preference is a key feature, as it allows for personalisation which is not usually available on recipe sites. As well as this, users may want to simplify recipes, add notes, or change the amount of servings in a recipe.

**3. Ingredient measurements must be convertible between different metrics.**

- Many measurements come in imperial format, which could be difficult to reproduce without appropriate equipment. A conversion function is a simple tool that adds a small amount of convenience to cooking.

**4. The application must be able to accurately estimate the total cost, as well as the per serving cost of a recipe based on cost of its ingredients.**

- As highlighted in the background research, cost estimation is a central feature in the application, as it has a lot of perceived value to users cooking on a budget. Furthermore, automatic cost estimation is not a task tackled by many existing cooking applications, making it an interesting topic to explore.

**5. There must be a system for users to register, which will allow them, at minimum, to save recipes to collections.**

- Munchmice is designed to be a web application - a decision that will be further discussed in the Technologies section in the next chapter. This means that user data will be centrally stored, hence a user system is absolutely necessary.

**6. There must be a user-modifiable shopping list.**

- Added as a result of the user survey, where a shopping list was suggested by many respondents. This contributes to the all-in-one aspect of the system, as it eliminates the need for another application/notepad when shopping for ingredients.

On top of this, a set of ten further possible expansions, "additional features", of the system were proposed. Features 1-5 are features currently implemented in similar cooking applications that were thought to be useful but not essential for this project, while features 6-10 are some features selected from those suggested by survey respondents on what they believed to be useful.

1. Allowing users to create and add recipes to different cookbooks;
2. Options to sort recipes by a number of criteria such as rating, cooking time, or cost;
3. Allowing users to add ingredients of a recipe to their shopping list in a few clicks;
4. A favourite system to allow users to quickly get to their best recipes;
5. Allowing users to choose a photo of the recipe from the source website;
6. Categorising and filtering recipes by cuisine type (British, Italian, Chinese), or by diet type (vegetarian, vegan, gluten-free);
7. A way for users to create their own recipes from scratch;
8. Allowing users to choose which supermarket they usually shop at to get better estimation of prices;
9. A way to discover new recipes, either full recipes or inspirations on what type of dish to cook;

10. Giving cost-saving tips based on ingredients (e.g. Switching from chicken to tofu could save you £1.50 in this recipe!).

## 3.1 Functional Requirements

After finalisation, these general requirements were further broken down into detailed functional requirement specifications found in Table 3.1, each with a requirement ID assigned for future reference. They were also assigned a priority based on the MoSCoW system [16]. There are four levels of priority: (M)ust, (S)hould, (C)ould, and (W)on't. The M requirements mostly consist of the core requirements listed above. The additional features are assigned either S or C, with the deciding factor being their perceived value contrasted with their cost to implement. In the end, most are assigned C, the rationale being that development effort should primarily focus on the core scraping and pricing functions. After evaluation, additional feature 5 was assigned a priority of W, meaning its implementation will not go ahead, due to image copyright concerns; additional feature 9 was also assigned a priority of W, owing to the fact that there will likely not be an external source to provide recipes integrated with Munchmice, hence recipe suggestions cannot go ahead.

Table 3.1: Functional requirement specification.

| **Functional Requirements** | | | |
|---|---|---|---|
| Epic | RID | Requirement | Priority |
| Recipe Import and Creation | 1.1 | Correct recipe must be found from URL. | Must |
| | 1.2 | Correct recipe details such as ingredients, measurements, and directions must be identified. | Must |
| | 1.3 | Ingredient must be processed into meaningful attributes. | Must |
| | 1.4 | Recipe must be stored into the correct cookbook. | Must |
| | 1.5 | There could be an interface for users to write a recipe from scratch. | Could |
| | 1.6 | Users could choose a photo from the source website for their recipe. | Won't |

| | | | |
|---|---|---|---|
| Recipe Organisation | 2.1 | User should be able to sort recipes by various attributes. | Should |
| | 2.2 | Different cookbooks can be created for recipe categorisation. | Could |
| | 2.3 | Users could favourite recipes to access them quickly. | Could |
| | 2.4 | Users could filter recipes by cuisine or diet type. | Could |
| Recipe Editing | 3.1 | Recipe details must be fully modifiable. | Must |
| | 3.2 | Recipe ingredients and directions must be fully modifiable. | Must |
| | 3.3 | Ingredient measurements must be scalable to serving size. | Must |
| | 3.4 | Units such as weights or volumes must be convertible between different metrics. | Must |
| Cooking Suggestions | 4.1 | New recipes suggested based on user's recipes. | Won't |
| | 4.2 | Ingredient substitutes suggested to cut costs or for a healthier alternative. | Could |
| Ingredient Pricing | 5.1 | The application must be able to calculate a cost estimation for any recipe given its ingredients. | Must |
| | 5.2 | The cost estimation should ideally be accurate but must be at least reasonable. | Should |
| | 5.3 | There should be a display of cost breakdown based on each ingredient. | Should |
| | 5.4 | There must be a per-serving cost for each recipe. | Must |
| | 5.5 | Users could choose which supermarket they frequent for more accurate pricing. | Could |
| User Profile | 6.1 | Users must be able to access a personalised view of recipes and shopping. | Must |
| | 6.2 | Users could be able to access a profile page displaying their details. | Could |
| | 6.3 | Users could be able to change details such as their username. | Could |
| | 6.4 | Users could be able to delete their account. | Could |
| Authentication | 7.1 | Users must be able to register for an account. | Must |
| | 7.2 | Users must be able to log in with their email and password. | Must |
| | 7.3 | An unauthenticated user must not be able to access the main interfaces of the website (e.g. dashboards, recipe details). | Must |

| | | | |
|---|---|---|---|
| | 7.4 | An user must not be able to access any information stored by another user. | Must |
| Shopping List | 8.1 | Users must be able to add items into a shopping list. | Must |
| | 8.2 | Users must be able to delete items from shopping list. | Must |
| | 8.3 | Shopping list must display an up-to-date view of items. | Must |
| | 8.4 | Users could add ingredients into shopping list directly from recipe. | Could |
| | 8.5 | Users could create different shopping lists. | Could |

## 3.2 Non-Functional Requirements

The non-functional requirements are formatted similarly to the functional requirements. These intend to define the qualities of our system, rather than its capabilities and features [17]. The non-functional requirement analysis can be found in Table 3.2. The main reference of our usability requirements, and the general framework kept in mind when creating designs, is the Eight Golden Rules of Interface Design outlined by Ben Shneiderman et al. in the book *Designing the user interface: Strategies for effective human-computer interaction* [18]. These are reflected in R10.1 - 10.6.

Table 3.2: Non-functional requirement specification.

| **Non-Functional Requirements** | | | |
|---|---|---|---|
| Category | RID | Requirement | Priority |
| Platform | 9.1 | The product must be a web application. | Must |
| | 9.2 | The system should be hosted on DCS servers. | Should |
| Usability | 10.1 | The system should be intuitive enough for users to operate without tutorial. | Should |
| | 10.2 | Users should be able to understand the function of clickable components by their appearance. | Should |
| | 10.3 | The user interface must have appropriate colour contrast for accessibility. | Could |

| | | | |
|---|---|---|---|
| | 10.4 | The user interface should be pleasant to the eyes. | Could |
| | 10.5 | The system should be fault tolerant in that user actions can either be undone, or warned beforehand. | Could |
| | 10.6 | There could be a customisation feature for users to change the appearance of the system. | Could |
| Performance and Scalability | 11.1 | The system must be able to support storing at least 50 recipes per user. | Must |

# Chapter 4

# Project Management

This chapter documents the planning which took place before the start of technical development. It will cover the methodology used, as well as introduce the technologies used to build the system. Lastly, project risks are outlined in a risk assessment and discussed, in order to ensure risks are accounted for and that the project can be delivered without issue.

## 4.1 Technologies

Munchmice is designed to be a web application. The main reason for this is its accessibility: as a website is platform independent, it can be accessed from any device connected to the internet and has a browser; it also requires no installation, which simplifies both the development process and the users' joining process. However, it is worth keeping in mind that the interaction interface will be more limited than that of a mobile or desktop application.

An overview of the technologies chosen for this project can be found in Table 4.1. For a web-based system, the most crucial layers in the technology stack are the frontend and backend frameworks. To this end, React.js [19] and Flask [20] are chosen respectively. As a JavaScript library, React not only simplifies and streamlines JS development, but provides a wealth of tools to reduce time spent writing boilerplate code, as well as giving access to pre-built components which can be imported. Orig-

inally, Django [21] was chosen as the backend framework for its batteries-included philosophy and its ease of scalability. However after initial experimentation, it was replaced with Flask [20], another Python framework based on WSGI that is more explicit than Django, has less base code [22], and is generally easier to work with.

Table 4.1: Table of technologies.

| **Name** | **Purpose** |
|---|---|
| React.js | Front end development |
| Flask | Back end development |
| SQLAlchemy | Python database toolkit and ORM |
| MySQL | Database management system |
| Python | Web scraper development |
| Extruct | Library for extracting embedded metadata |
| GitHub | Version control |
| Asana | Kanban board for project management |
| Overleaf | LaTeX-based write-up tool |

Munchmice will be developed and tested locally. The rationale behind this is that it is a personal tool and does not require interaction between users at this stage. Setting up the project to be ran on servers provided by the Department of Computer Science at the University of Warwick would be ideal, as it allows for user testing on a larger scale, as well as allowing for performance tests, most of which cannot be carried out on a local server. Nevertheless, the process of setting up this server is an optional task after the local implementation is completed, as it does not have an major effect over the system functionalities, and only facilitates better testing.

## 4.2 Work Breakdown Structure

A work breakdown structure, shown in Figure 4.1, is created based on the functional requirement specification, using epics as level 1 elements. When it comes to scale, recipe import and ingredient pricing, though self-contained, have the greatest workload required. As a result, they can be found in their individual sections for the

Design and Implementation chapters, whereas other work packages will generally be discussed together or in smaller subsections.

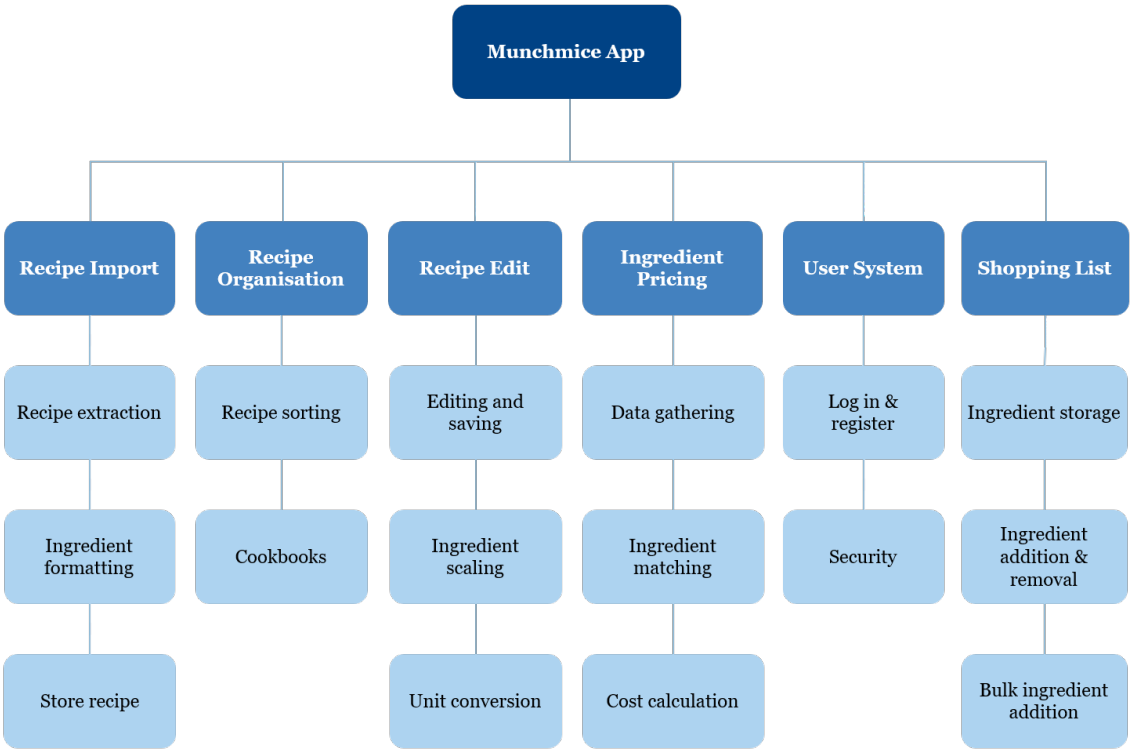


Figure 4.1: Work breakdown structure.

## 4.3 Methodology

The development methodology used is that of a combined approach between agile and waterfall. Web development naturally lends itself to iterative methodologies with features being added in each iteration, and a developer team of one meant that it is easier to accommodate changes during development compared to a larger team. After some consideration, it was decided that this project will not be utilising sprints. Though initially it appeared as a logical solution to break down the 6-month development period into 2-week long sprints, the actual time spent working each week varies wildly owing to holidays and deadlines. This changing availability, coupled with uncertainties associated with project development, makes it difficult to plan effective sprints that can be strictly adhered to.

Instead, development is planned around four longer iterations, where each iteration will yield a minimum viable product by the end. That is, a early version of the software that fulfils the minimum necessary requirements, though for our project it may have missing features compared to the final version. The four iterations planned are explained below, and these are planned in such a way that the project's core features would all be complete and fully functional before any major work on additional features are started. Not only does this map out a clear path of development, it also ensures that the project will be finished on time even if it runs behind schedule, at the cost of a few additional features.

**Iteration 1: Basic website creation and scraper development.** The first iteration has a large amount of groundwork planned, as the goal is to build a framework that can be worked on in later iterations. To this end, both the React and Flask application are set up and integrated. The wireframe for the website, designed on Figma [23], is also made in this iteration. Lastly, the prototype for the recipe extractor, or scraper, is built so that it can be tested and refined along development, but it is not yet integrated into the system at this point.

**Iteration 2: Cost estimation implementation and integration.** As a large part of the project, the research and implementation of the cost estimation system is planned in an individual iteration. This involves a deeper research into how cost estimation can be achieved, implementing and integrating the system, as well as integrating the scraper.

**Iteration 3: Finish implementation of core features.** By the end of this iteration, it is planned for all functional "Must" features to be implemented. This includes recipe editing, shopping list, and the user system. As this essentially marks the completion of the project, this iteration can overrun without much issue if necessary.

**Iteration 4: Additional features implementation and testing.** In the last iteration, additional features are implemented based on their priority, starting with "Should" features. The aim at the end of this iteration is to have all "Should" features completed, and to have done preliminary integration testing.

It was stated that the development approach also combined techniques from waterfall, a planned methodology. This is due to the use of a Gantt Chart to plan out the project. In the chart shown in Figure 4.2, each iteration, as well as the tasks within each iteration, are associated with a date range. It should be noted that this is the first version of the Gantt Chart, and it only includes three iterations. Later, the first iteration is split into two, following the above iteration plan. It is planned out this way initially to determine the feasibility of project completion within the given time, and during development its main purpose is to serve as a reference of how long each iteration should last, and for regular review to ensure the progress is inline with expectations.

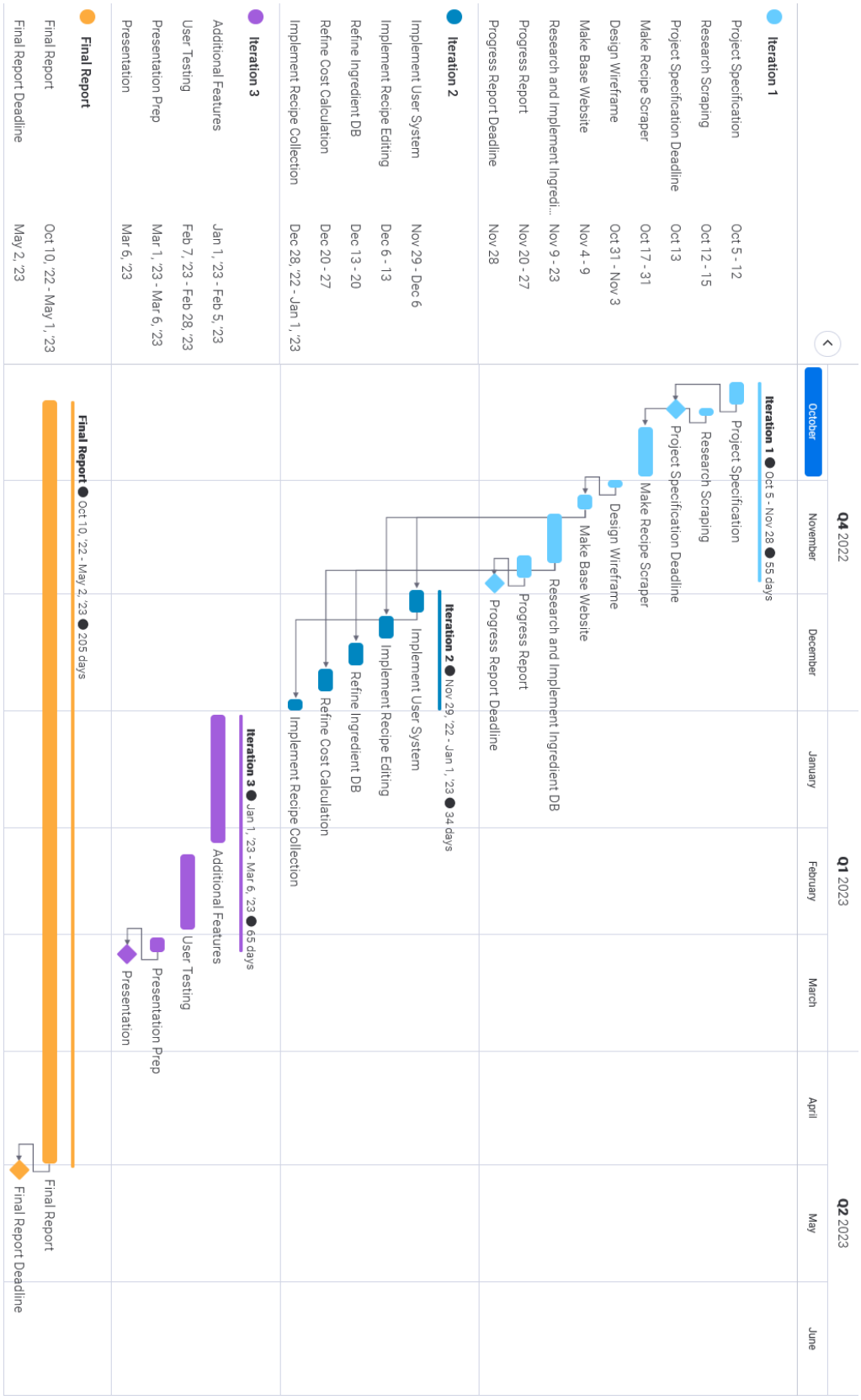


Figure 4.2: Gantt Chart at the start of project. The second iteration was later split into two iterations.

To assist in better management of tasks within iterations, a simple Kanban board in Asana [24] was employed to manage smaller work packages. As seen in Figure 4.3, the board is split into three columns: to do, in progress, and complete. This was not a formal or particularly regulated process, however by enforcing a limit of at most two tasks in the in progress section, it was able to help focus efforts onto the selected, most important tasks at the moment. Later, the tasks were also divided into [STORY] and [BACKLOG] categories, where story involves the implementation of new work packages, and backlog addresses issues currently found within the code. The purpose of this is to keep in check the time spent on debugging and polishing already written code, and ensure there is progress made with new features.

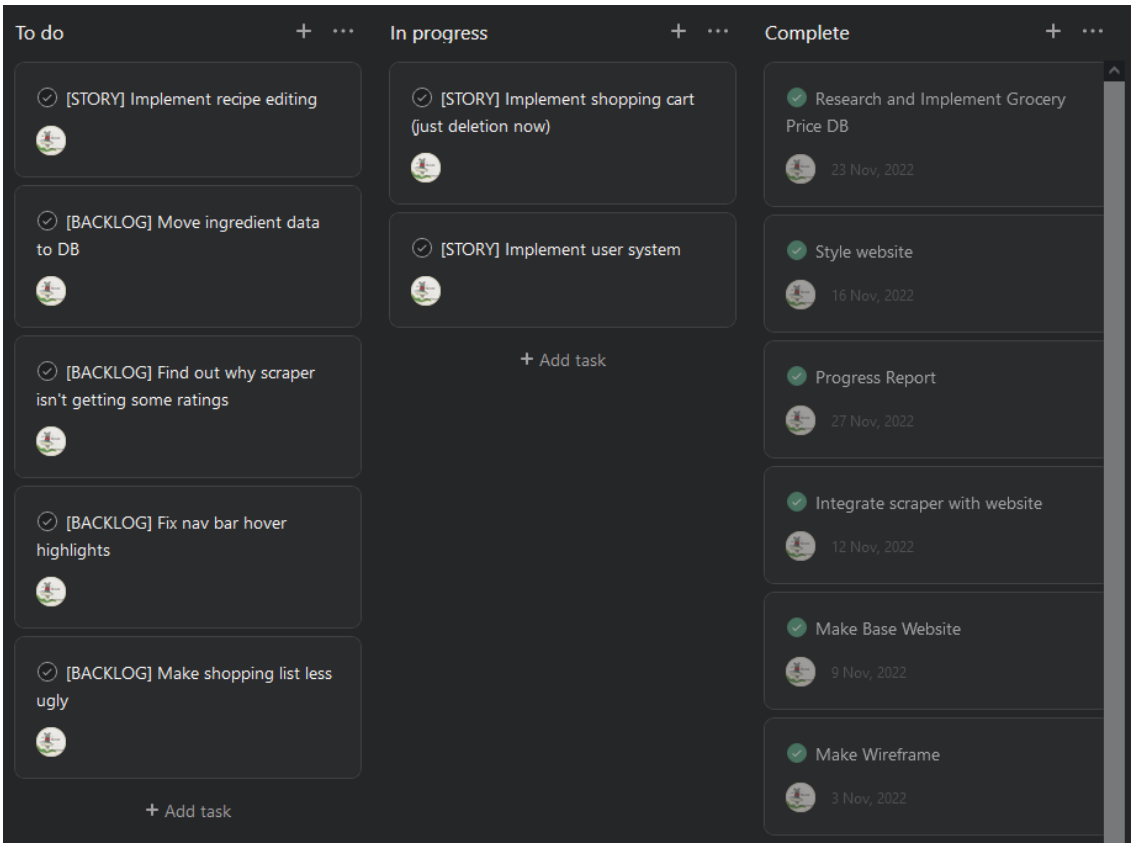


Figure 4.3: Kanban board layout used for the project.

## 4.4 Risk Management

Potential risks are outlined in Table 4.2. Their likelihood and negative effect on the project are assessed in combination, and as a result each risk is assigned an impact of low, medium, or high. Mitigation strategies are planned for each risk, and their impacts are re-assessed after mitigation. The purpose of this task is to help keep the project on track, should there be any unforeseen circumstances.

Table 4.2: Project risks and mitigation strategies.

| **Risk** | **Impact** | **Mitigation Strategy** | **Residual Impact** |
|---|---|---|---|
| Time crunch | High | Use of gantt chart and overestimating time needed for tasks | Medium |
| Scope creep | Medium | Defining clear requirements at the start, scope creep is also limited by the use of iterations | Low |
| Illness or low performance | Medium | Overestimating time required | Low |
| Change in development direction | High | Conduct thorough research before development | Medium |
| Insufficient technology | Low | Very unlikely and can be detected early on | Low |

# Chapter 5

# Design and Research

Prior to the start of development, the planning for the technical parts of development was also carried out. This chapter will discuss the designs made for each sub-system, as well as the user interface wireframe, and discuss the rationale behind them.

## 5.1 System Architecture

The Munchmice system has three primary subsystems, shown in Figure 5.1. The Flask backend handles most of the data processing, as well as communicating with the database. It also contains the two subsystems written in Python: the web scraper for recipe importing, and the pricer for cost estimation. The React frontend is responsible for user interaction and communicates with the backend using a JavaScript API, where it is able to request or send data. The MySQL database interacts with the backend using SQLAlchemy [25].

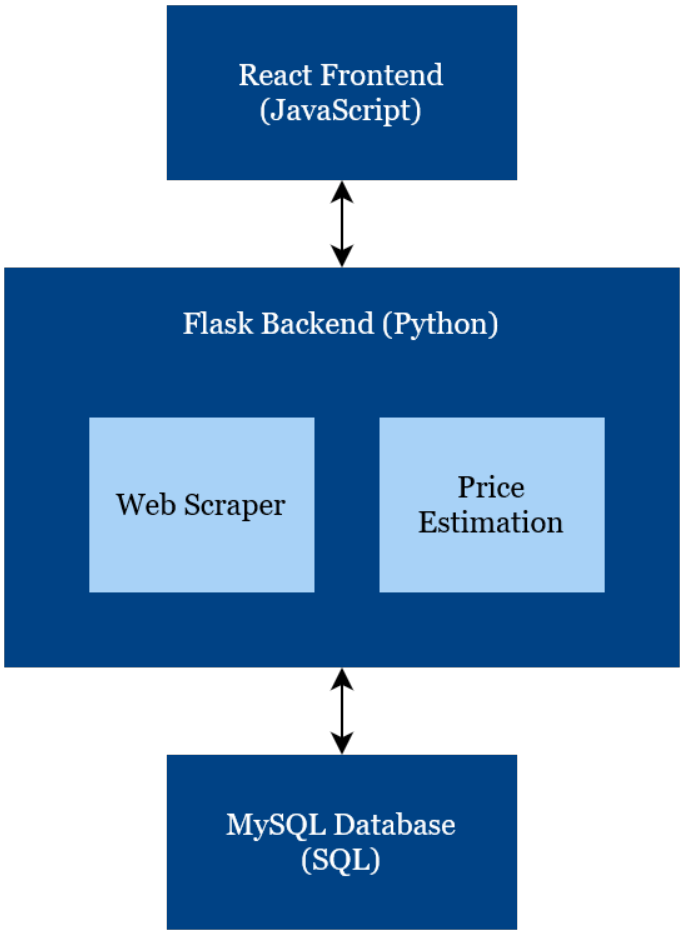


Figure 5.1: System architecture for Munchmice.

The two smaller subsystems within the Flask backend have been highlighted due to their distinct purpose. One of these is the scraper, the system designed to extract recipe information given a particular URL; the other is the pricer, which is designed to match a given list of ingredients to ingredients in the price database and return their price estimates. Both the scraper and the pricer are written in Python to keep the structure of the overall system simple.

## 5.2 MySQL Database

The relational database consists of seven tables, the relationships and details of which are illustrated in the relation schema shown in Figure 5.2. Though the storage of recipes is the key focus and the most important table in the database, the relation of the tables are in reality centered around the User table, as any cookbooks

and shopping lists must be associated with one user. From this point, any number of recipes can be associated with one cookbook, and any number of shopping list ingredients can be associated with one shopping list. Details for each table can be found in Table 5.1.

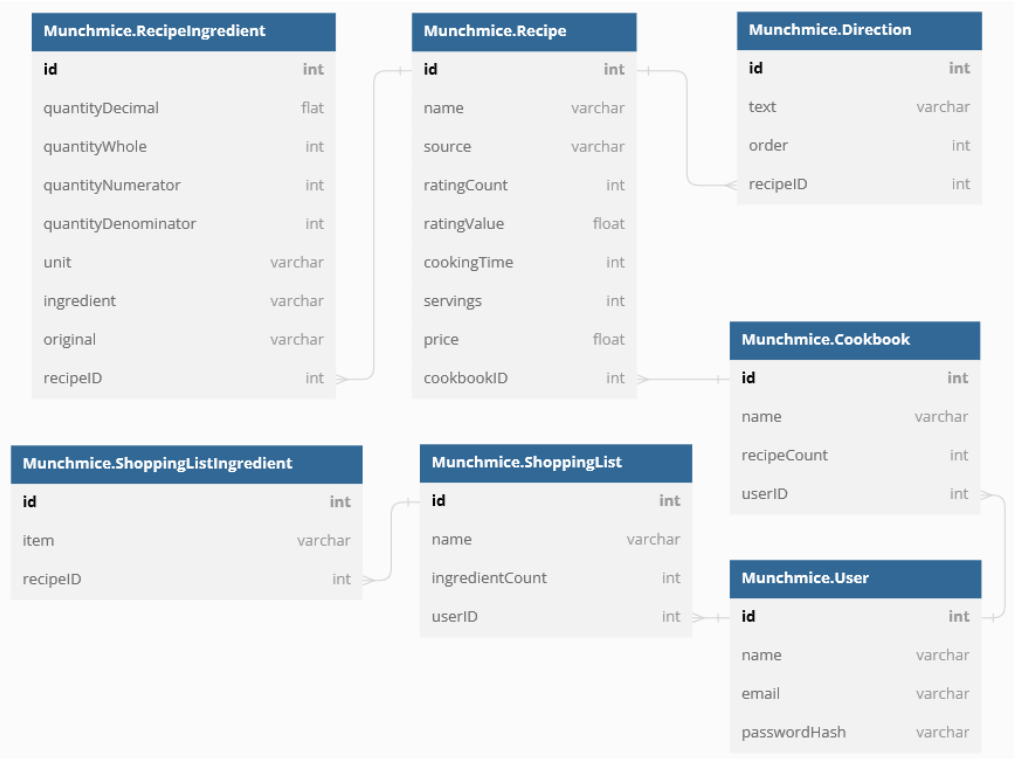


Figure 5.2: Database schema for Munchmice.

Table 5.1: Details of each database model.

| Model | Description |
|---|---|
| Recipe | Each row holds the description of one recipe, which has a unique ID referenced by ingredients and directions. Details of the recipe, such as its source, cooking time, and ratings, are stored here. Originally, the price was planned to be calculated and stored upon import, but later it was changed for the price to be dynamically calculated upon request and not stored in the database. Each recipe has a cookbook which it belongs to, indicated by the `cookbookID`. |

| | |
|---|---|
| RecipeIngredient | Each row holds one ingredient to one recipe, indicated by `recipeID`. The quantity of the ingredient is either stored as a decimal number in `quantityDecimal`, or stored as a fraction in `quantityWhole`, `quantityNumerator`, and `quantityDenominator`. The original ingredient string prior to processing is stored in `original` for reference. |
| Direction | Similar idea to RecipeIngredient, each direction belongs to one recipe referenced by `recipeID`. The direction text is stored in `text`, and the step number of the direction is kept track in `order` so that a list of directions can be sorted. |
| Cookbook | The cookbook is referenced by a number of recipes in it. It belongs to a single user, referenced by the foreign key `userID`. The field `recipeCount` was designed to save time for specific operations but did not end up being used. |
| User | A user can have multiple cookbooks and multiple shopping lists. Each user has a username, the email they registered with which is unique, and their password hash used to verify identity. |
| ShoppingList | Very similar to Cookbook. Each shopping list is associated with one user, and can be referenced by a number of items, or shopping list ingredients. |
| ShoppingList-Ingredient | Each row holds a single item added to a shopping list. The item is stored in plain text. |

As each recipe ingredient needed to be broken down into a number of components, it makes sense to represent them in a structured way via the use of an ingredient table. It is possible to include the directions as a field in the recipe table, however, storing each direction separately allows smart ordering of directions upon edit, and better display. This design of the database is similar to the one described in McKean's blog post [26], however, in their design the ingredient measurement unit, quantity, and description each have their own table, as can be seen in Figure 5.3. For this to be useful, there should be universal units and quantities in the system. As Munchmice uses web-imported recipes, consistency between recipes cannot be guaranteed, and it is safer to stay as close to the original format of ingredients as possible, hence all

information regarding recipe ingredients will be stored in the ingredient table.


Figure 5.3: Relational database design for a recipe, proposed by Alex McKean.

It is worth keeping in mind that this implementation will result in longer times taken to display recipes, due to the fact that the correct ingredients and directions with matching recipe IDs need to be searched for in the database. When thinking about scaling up, it may make more sense to change this storage approach, for example to store directions directly in the recipe table.

## 5.3 Flask Backend

As outlined in the system architecture diagram, the Flask backend communicates with both the React frontend and the MySQL database, and has two main sub-systems: the web scraper and the price estimator. In the following sections, we will focus on the design of these two primary sub-systems.

### 5.3.1 Web Scraper

A key feature of Munchmice is the ability to extract recipe details from an URL. To achieve this, there are two general approaches; they both require first sending a request for the page, but the response is treated differently:

**Method One** Parse the HTML body of the page arbitrarily. This requires manual identification of the required information. For example, to find the instructions of a

recipe, we first need to determine how likely some text is an instruction. Ben Awad, the creator of Saffron, discusses this in his dev blog [27], where he proposes some criteria for how likely this may be the case, such as: *Does it start with a capital letter? Does it end in punctuation? Does it contain words specific to instructions such as "Sprinkle", "Mix", or "Heat"?* These criteria are used to score each node in the HTML. After the highest scoring nodes for both ingredients and instructions are found, the lowest common ancestor between them can be obtained, and this is thought to be the recipe node containing all relevant information relating to the recipe. From this point, further processing is needed to distinguish between nodes containing different sections of the recipe.

**Method Two** Extract recipe details from the structural metadata contained in the page. Structural metadata contains a condensed representation of information present on a page, and is primarily used for search engine optimisation purposes [28]. For this reason, it is often found in recipe sites in order for Google to provide "Rich Results". Structural data for a recipe contains all its core information, such as its ingredients, instructions, and cooking time, which means we can make use of this information for recipe extraction.

For Munchmice, the structural metadata method will be used for to extract recipe information. It must be noted that exclusively using metadata means some sites would not be able to be parsed at all, as they may not contain this information. However, this constitutes of an increasingly small percentage of sites. Furthermore, the HTML traversal method has to cater to a large amount of different cases, and as Awad explained himself, a number of assumptions were made for his design, such as ingredients and recipes being in two contiguous blocks. With these factors in mind, as well as taking into consideration the time cost to implement a sufficiently accurate design, it was determined that the returns of implementing a HTML traversal algorithm is diminishing.

With this said, we must understand how to extract information from a page's em-

bedded metadata. Typically, recipe information can be found in a part of the script with the type `application/ld+json`, containing the data in a JSON block. This is the format known as JSON-LD, which is added to a page using a scripting language [29]. As JSON-LD is the format recommended and preferred by Google, most websites make use of it. However, as this format is relatively new, many sites have structural data instead represented as Microdata, which is similar to JSON-LD, but instead embedded in the HTML body of a page [30]. This can be found in sections with the `itemprop` tag. What is important is that both JSON-LD and Microdata use the markup vocabulary for recipes defined by Schema.org [31], under the Recipe type. Hence, once the JSON-LD or Microdata blocks are obtained, the details of the recipe can be essentially "looked up" using the schema provided, as seen in Figure 5.4.

| Property | Expected Type | Description |
|---|---|---|
| **Properties from Recipe** | | |
| cookTime | Duration | The time it takes to actually cook the dish, in ISO 8601 duration format. |
| cookingMethod | Text | The method of cooking, such as Frying, Steaming, ... |
| nutrition | NutritionInformation | Nutrition information about the recipe or menu item. |
| recipeCategory | Text | The category of the recipe—for example, appetizer, entree, etc. |
| recipeCuisine | Text | The cuisine of the recipe (for example, French or Ethiopian). |
| recipeIngredient | Text | A single ingredient used in the recipe, e.g. sugar, flour or garlic. Supersedes ingredients. |
| recipeInstructions | CreativeWork or ItemList or Text | A step in making the recipe, in the form of a single item (document, video, etc.) or an ordered list with HowToStep and/or HowToSection items. |
| recipeYield | QuantitativeValue or Text | The quantity produced by the recipe (for example, number of people served, number of servings, etc). |
| suitableForDiet | RestrictedDiet | Indicates a dietary restriction or guideline for which this recipe or menu item is suitable, e.g. diabetic, halal etc. |

Figure 5.4: Schema of a recipe block in structural metadata.

### Ingredient Formatting

As a part of recipe extraction and storage, ingredient formatting is one of the most complex challenges in the development of the scraper. This sub-section hopes to illustrate the problem, as well as discuss approaches that could be taken to overcome it.

Any ingredient string in a recipe can be broken down into four meaningful attributes, shown in Table 5.2 below. In particular, the quantity, unit, and name of

an ingredient are utilised by other parts of the system. On top of this, an ingredient may sometimes come with some additional text which give further details regarding its form or preparation, such as "finely chopped" for an onion.

Table 5.2: Attributes of an ingredient and their usages.

| Attributes | Examples | Usage |
|---|---|---|
| Quantity | 1, 2.5, 1/2 | Recipe scaling |
| Unit | tsp, tablespoons, oz | Unit conversion |
| Name | brown sugar, broccoli, paprika | Ingredient pricing |
| Additional Text | chopped, softened, room temperature | Additional instruction |

The issue arises with the inconsistency of formatting between ingredients. Illustrated in Figure 5.5, we can see that an ingredient may only have some of the attributes listed. One ingredient may be have anywhere between 1 and 4 attributes, with a minimal example being "lime wedge". The question, then, is how to extract meaningful information out of an ingredient with an unknown number of attributes.

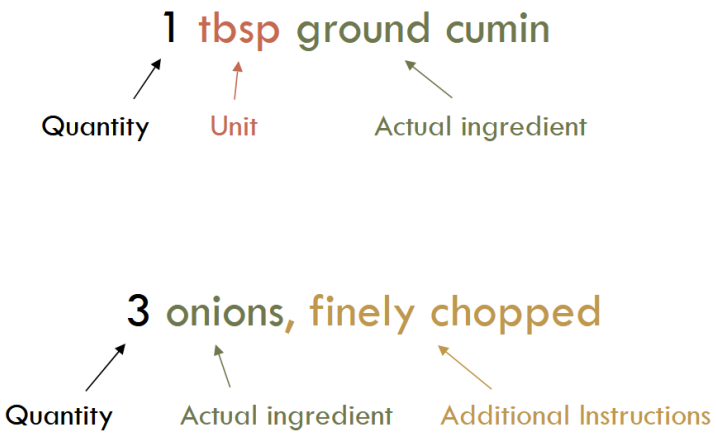


Figure 5.5: Example ingredients and their breakdowns.

The answer is not trivial. In fact, without access to a dataset of all possible instances of quantities, units, and ingredients, there is no method of parsing an ingredient string that guarantees complete accuracy. The most sophisticated approach, then, is machine learning. Shi et al. [32] proposes the use of a recurrent neural network, which utilises an attention mechanism, to output 8 attributes of a given ingredient. Through training their experimental setup over a set of 6612 ingredients and testing over 2188, they were able to attain an F1-score of 0.93. In comparison, the named entity recognition model proposed by Diwan et al. [33] had an F1-score of 0.61. Certainly, the attention-based recurrent model has its merits in ingredient parsing and would make an excellent model, but the time and scope of this project calls for a simpler approach.

Simson [34] uses a parser combinator based approach for ingredient parsing in his personal project. Essentially, an ingredient is broken down into a set of tokens, for example, `{amount} {unit} {ingredient}`. This is a promising approach, though Simson did not have an implementation. Building on this idea further, MichielMag [35] developed a solution in Python using regular expressions and predefined units. We will take a further look at this approach, as it appears to be a good fit in terms of complexity and accuracy. In a simplified view of their implementation, first the quantity is found using a series of regular expressions, such as `(\d{1,3}?\s\d\/\d{1,3})` for simple slash fractions. Then, the rest of the ingredient string is searched for any units listed in a predefined list. Lastly, the rest of the string can be named the ingredient.

MichielMag's implementation heavily relies on a large amount of regular expressions, and makes the assumption that the ingredient attributes are in the order of quantity, unit, ingredient. Using this same assumption and the same key ideas, we can develop a similar design for an ingredient parser that is simpler and uses many less regular expressions. On top of this, we make one further assumption that any additional instruction or information is at the end of a string, separated by a comma or an open bracket. In our design, there is a predefined list of units, but differs in that it

uses the units to split the ingredient string into three parts. Then, the part before the unit is deemed the quantity, and the part after the ingredient name.

### 5.3.2 Price Estimation

Another key feature of Munchmice is providing a price estimation for user-imported recipes. There are two possible approaches to price data acquisition: an online search and extraction could be conducted for each ingredient once the user requests a price estimation; alternatively, we could gather price data for a large set of grocery items and create a data set in advance, which the system can query upon a request. The second approach would require periodic updates to ensure the data provided reflects current prices.

Though it requires more maintenance, the second approach is chosen. Compared to sending multiple HTTP requests to an external website, querying the database is more consistent and can be done in a batch format. The most reliable and up to date data source for grocery pricing is supermarket websites, hence the initial direction of this part of development is to build a web scraper to extract price information from supermarket websites. This was possible through the use of the Python HTTP library, Requests [36], and processing the HTML body in the response. However, though conceptually sound, this approach was not feasible due to legal limitations surrounding grocery price scraping. Trolley [37], a free grocery price comparison site, was sent a legal letter [38] from a company digitising information for 98% of all UK groceries requesting them to remove all scraped information or pay a licensing fee. These information included both images and price data. Though this project only aims to develop Munchmice as a proof of concept rather than a ready-to-deploy system, the design itself should have ways to assure it is future-proof; and since Munchmice is designed to be a free-to-use application, there would not be sufficient funds to maintain a yearly licensing fee. Hence another direction needs to be considered for sourcing price data.

Another option for grocery price data could be free, public data sets - the De-

partment for Environment, Food and Rural Affairs, for example, conducts regular surveys of commodity prices in the UK [39]. One data set for wholesale prices of home-grown horticultural produce in England and Wales [40] is exceedingly comprehensive, and updated as recently as April of 2023 at the time of writing this report, a snippet of which can be seen in Figure 5.6. However, the problem arises with the type of data included in the collection; specifically, there are data sets on agricultural, milk, egg, and cattle price indices. Though comprehensive in their own right, many types of commonly used ingredients, as well as products produced outside of the UK, are not included. Many other similar data sets suffered from the same problem.

| category | item | variety | date | price | unit |
|---|---|---|---|---|---|
| fruit | apples | bramleys_seedling | 2023-04-14 | 1.05 | kg |
| fruit | apples | braeburn | 2023-04-14 | 0.99 | kg |
| fruit | apples | gala | 2023-04-14 | 1.03 | kg |
| fruit | apples | other_late_season | 2023-04-14 | 1.25 | kg |
| fruit | pears | conference | 2023-04-14 | 1.44 | kg |
| vegetable | asparagus | asparagus | 2023-04-14 | 12.72 | kg |
| vegetable | beetroot | beetroot | 2023-04-14 | 0.62 | kg |
| vegetable | pak_choi | pak_choi | 2023-04-14 | 3.7 | kg |
| vegetable | curly_kale | curly_kale | 2023-04-14 | 4.86 | kg |
| vegetable | cabbage | red | 2023-04-14 | 0.9 | kg |
| vegetable | cabbage | savoy | 2023-04-14 | 0.96 | head |
| vegetable | spring_greens | prepacked | 2023-04-14 | 1.9 | kg |

Figure 5.6: Wholesale prices of horticultural produce in England and Wales.

The lack of sufficient data presented a roadblock to the project, as the legal issues only came into view after the start of development. After much deliberation, it was decided that the pricing data for Munchmice will be gathered from Trolley itself. Trolley provides a Grocery Price Index [1], which contains up to date records of price estimates for a significant number of categories of grocery items, each calcu-

lated from the average cost of an item between multiple grocery stores. This was not an ideal solution, as it requires manual gathering of data due to trolley's policies regarding web scraping. Furthermore, this means Munchmice's price data would not be realistically maintainable or at least kept accurate, should it be launched for public use. Nevertheless, Trolley's data is by far the most up to date and comprehensive record of grocery prices in the UK, and the absence of another dataset that is remotely comparable makes it the best candidate for this project.

It is worth noting that Trolley's data does not include price by unit, and instead provides a singular price figure for one type of item. This effectively removes the possibility of cost estimation by weight or by unit, as most grocery items come in a range of sizes. Furthermore, as Trolley's data takes the average of all items in the category, the price estimation is also limited to returning the average price of an ingredient, as opposed to the minimum price of an ingredient, which may be more accurate for a student budget. Due to the lack of data, both of these issues are large challenges to overcome, and will not be tackled in this project. However, solutions have been considered should Munchmice be further developed, and a discussion on this topic can be found in the Future Work Section in the Conclusion Chapter.

## 5.4 React Frontend

The user interface design is a large part of the development of Munchmice. The project itself is focused on application of relatively standard techniques to bring about the development of a novel tool, hence it is important for this tool to provide a good user experience and aesthetic design. This requirement is also included in the Usability section of the non-functional requirements of the project, and reflected specifically in requirements 10.1 to 10.5.

The recipe details page has the highest importance, and hence was the first page to be designed. As well as being the page a user will spend most of their time looking at, it holds a lot of information and more functionalities than any other page.

Therefore, it is important to nail down the design of this page, as the rest of the designs will follow in a similar style. The layout of the page had inspirations taken from OneNote [41] - though OneNote is designed for note taking, the mechanisms of the two applications are in reality quite similar: they both allow creation of editable pages, organised into a notebook format. To tackle the organisation of notes and notebooks, OneNote uses a side bar to allow quick navigation of the collection, while the actual note content is displayed on the middle and right side of the page. On top of the page, there is a tools bar that helps to edit the current note. This very functional design inspired the layout of Munchmice.

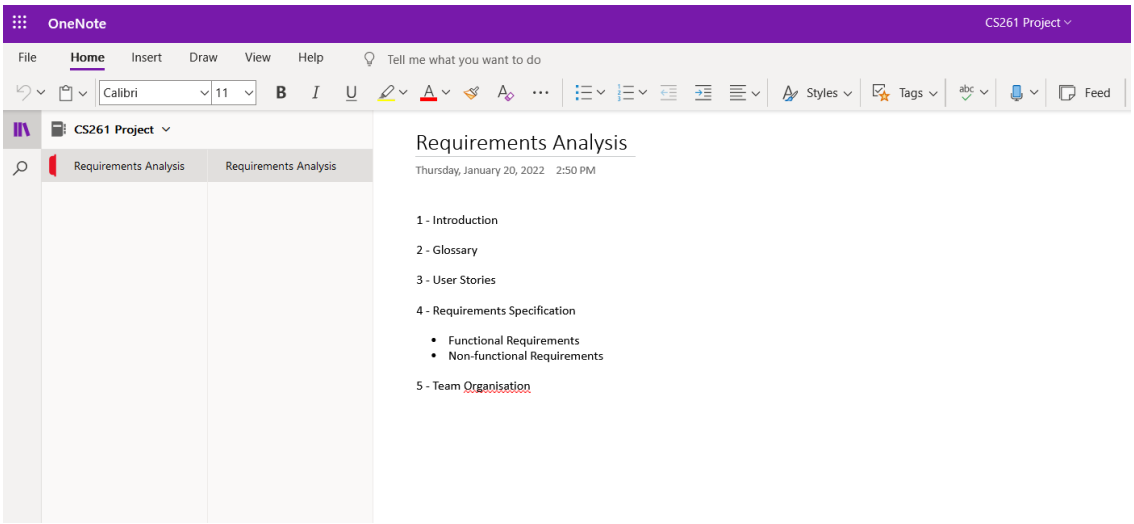


Figure 5.7: Interface for OneNote by Microsoft.

An initial design is first mapped out on Figma; shown in Figure 5.8, it establishes some initial ideas of the style. Flat block colours are used to convey a modern feel, and the bright yellow colour is introduced to make the design more interesting. The recipe section itself is designed to be reminiscent of a ring-bound notebook, which may inspire users to associate it with a real-life home cookbook.

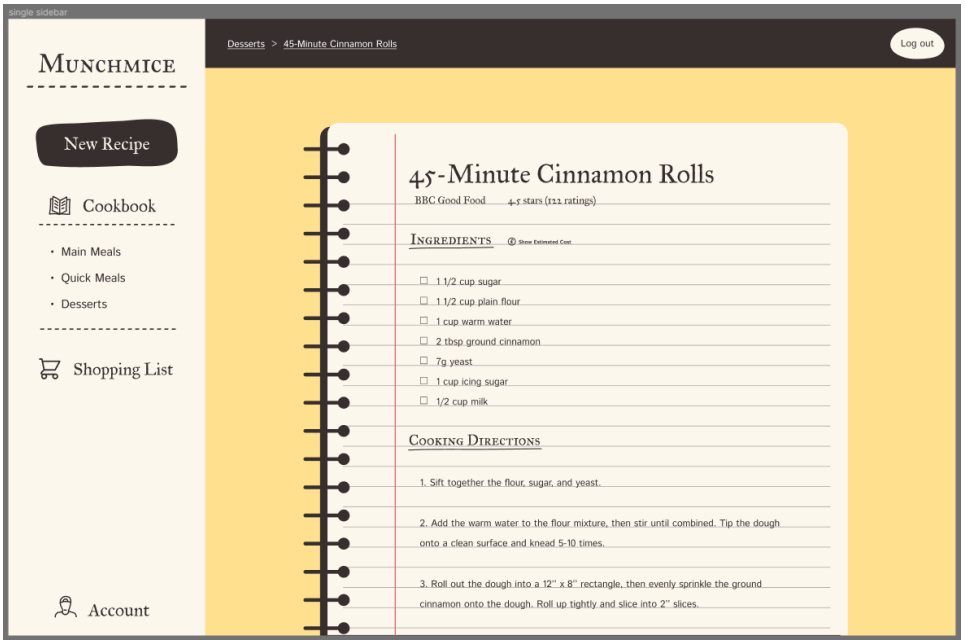


Figure 5.8: Initial design of the recipe page.

There are a few lackluster aspects of the initial design: the yellow background feels overwhelming and does not provide enough contrast with the cream colour to make the cookbook stand out, while having too much contrast with the dark brown bar at the top; the side bar is attractive but would become cluttered if there are too many recipes; and the cookbook itself is almost too similar to a notebook, with lines making the space feel cluttered. With more styling experimentation, the design eventually evolved into the one shown in Figure 5.9. The side bar has been split into two, with the outer, darker bar responsible for navigation between core interfaces, and the inner, lighter bar responsible for recipe navigation. A fourth colour, light brown, is introduced to bridge the sharp contrast there previously was between the dark brown and cream colours, establishing the final colour palette shown in Figure 5.10.

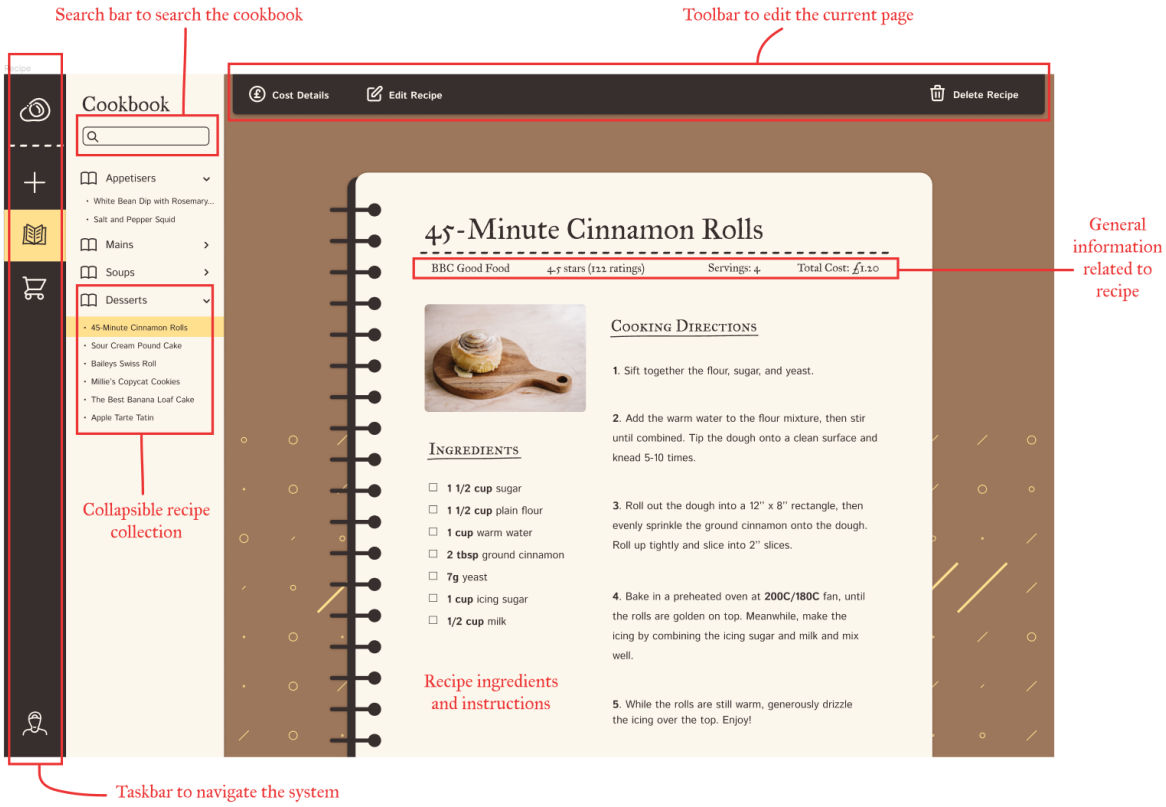


Figure 5.9: Final design of the recipe page.

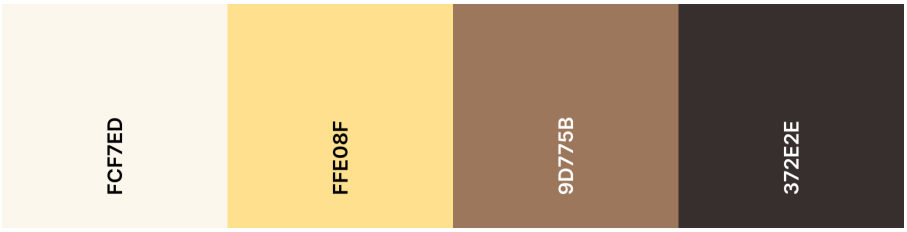


Figure 5.10: Colour palette for the default style of Munchmice.

The colour palette was chosen for its warmth and earthy tone. While this is not standard in modern sites, the bold stylistic choice was made in an effort to create the cozy atmosphere that home cooking can bring. Different colours were also experimented with for a better view of the structural design of the page, and whether the current colour is appropriate, which can be seen in Figure 5.11. These variants were also included in the aforementioned user survey (Figure 5.12), and the conclusion of this experiment was that the current colour is the best default choice, but the possibility of giving the user options to change the theme could be explored.


Figure 5.11: Alternative colour variants for Munchmice.

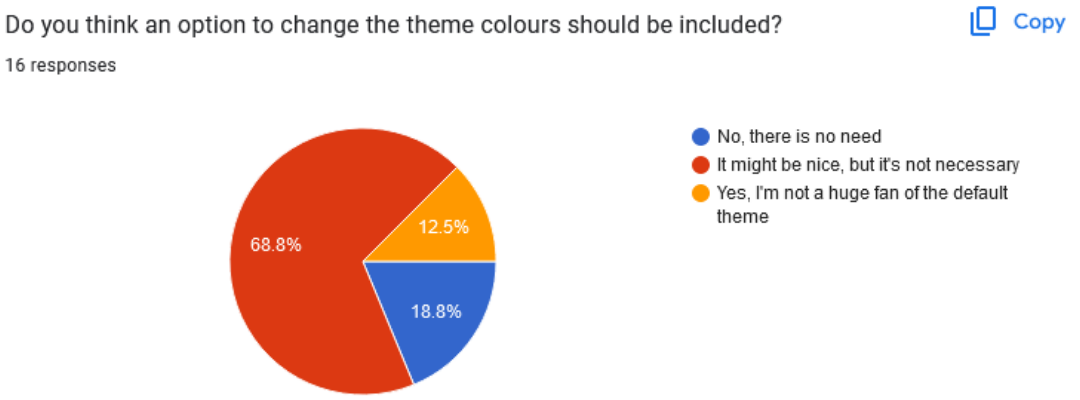


Figure 5.12: Alternative colour variants for Munchmice.

Once the design of the recipe page has been confirmed, other pages followed in a similar style. Consistency, Shneiderman's first interface design principle [18], is kept in mind throughout, with a consistent colour palette, as well as the cookbook silhouette and standardised fonts. The side and top bars remain the same in most pages, with the exception of the home page. The home page is where the user first lands after login, and its functionality is mostly that to provide an overview of the system, one example of which is a mini version of the shopping list. In the future, this page could also host other information, such as recipe suggestions.

The interfaces are linked via the side bars, with the aim of keeping users in control and allowing fast operations for experienced users. Most importantly, the pages are designed so that there are no surprises for users - each clickable item is designed to have a predictable result with obvious feedback, to both keep users in the loop and provide ways to backtrack to previous states and locations. One example of this is the recipe page at the bottom right of Figure 5.13, where the cost estimation window is overlaid on top of the recipe itself. This gives the user a sense of their location, as well as a clear way to exit the current state, indicated by the cross icon.


Figure 5.13: Overview of example page designs.

The last step of design is creating icons to replace all the placeholder icons used in the wireframes. Thought not strictly necessary, the vector graphics shown in Figure 5.14 are quick to create and give more originality to the design. Furthermore, the use of icons help reduce short-term memory load without the need of reading any text - their resemblance to real-world objects facilitates fast navigation with no memorisation. This is yet another principle outlined by Shneiderman [18]. A logo was also created through some experimentation, with the idea being a mouse icon that resembles the letter M. The design evolution for this process can be seen in Figure 5.15.

Figure 5.14: Icons designed for Munchmice.


Figure 5.15: Logo designs. The fifth icon is chosen for the final logo.

# Chapter 6

# Implementation

The implementation of the system and its features lasted throughout the first and second term, with some final additions made after this time. As there are a large number of moving parts and components to the system, the goal of this stage of development is to achieve a prototype for Munchmice that fulfils the requirements on a smaller scale.

In this chapter, we will discuss the construction of the website, as well as the management of source code. First, the React application was created, followed by setting up Flask. Then, the web scraper and price estimation sub-systems are implemented and integrated with the Flask application. The front and backend applications, as well as the MySQL server, are hosted using Apache in XAMPP [42], the interface of which can be seen in Figure 6.1.

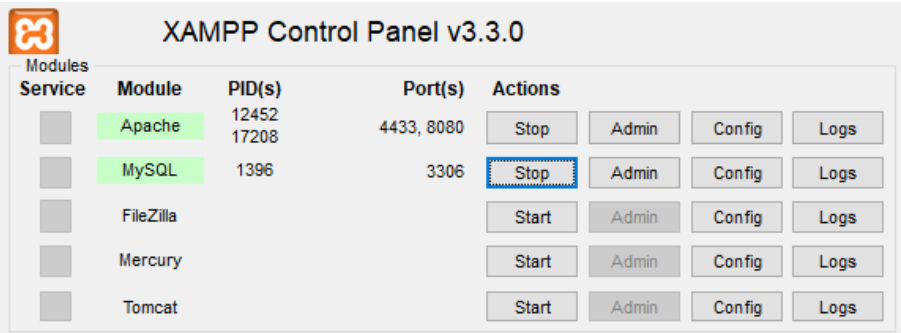


Figure 6.1: Interface of XAMPP. Apache and MySQL are used for the project.

## 6.1 React.js Frontend

The initial App.js file simply renders the title of the page with `useEffect`, a type of in-built React Hook [43] that is used throughout frontend development, a very simple example is shown in Figure 6.2. It is worth noting that the `useEffect` hook is typically used to connect and synchronise components with external systems.

```
useEffect(() => {
    document.title = 'Munchmice';
  }, []);
```

Figure 6.2: Example usage of `useEffect`.

Munchmice makes use of page routing using React Router [44]. As a component-based library, React does not come with in-built support for routing between multiple pages, but React Router makes this possible. In traditional websites, when the user requests for a new page, new content needs to be requested and downloaded, as well as rendered. This includes the CSS and JavaScript assets. React Router allows for client side routing, which lets the UI of the new page to be immediately rendered while contents of the page is requested, resulting in a smoother user experience.

```
const Main = () => (
    <Routes>
      <Route path='/' element={<Login/>}></Route>
      <Route path='/register' element={<Register/>}></Route>
      <Route path='/home' element={<Home/>}></Route>
      <Route path='/recipes' element={<Recipes/>}></Route>
```

Figure 6.3: Navigation to different pages using routes.

Though all component files are stored in the same directory, the logical structure of the React files roots in the `Main.js` file. As shown partially in Figure 6.3, the `Main` component is the parent of routes for each page of the website. Not all other components are pages - notable examples include `Navbar.js` and `Topbar.js`. These

components are children of the main pages, and are built as components for reusability. `Navbar.js`, for example, is present on all pages of the website apart from the login and register interfaces. Changes made to this file will be enacted on all pages which reference it, highlighting another benefit of React's components system. Furthermore, using React Router's `useLocation` hook, the navigation bar's appearance can be changed dynamically based on its current location, or route. This can be seen in Figure 6.4, where `var isHomePage = useLocation().pathname ==="/home"`.

```
<Link to="/home" className='logo'>
    <img className='navbar-icon' src={meece} alt="Munchmice Logo"/>
    {isHomePage &&
        <span>Munchmice</span>
    }
</Link>
```

Figure 6.4: The link displays "Munchmice" next to the icon if the current page is the homepage.

The styling of the frontend is done using SCSS [45], a CSS extension which provides more advanced syntax, allowing writing CSS to be less repetitive and hence saving time. The CSS nesting feature it provides proved to be very useful in development, given the nested nature of HTML hierarchy. By installing SCSS using `Node.js`, it can be set up to automatically convert `.scss` files to `.css` files at run time.

## 6.2 Flask Backend

Most of backend development surrounds `app.py`, where the Flask class object is created. The structure of the Flask system and much of its integration with React referenced the Flask-React setup by Forogh P. [46].

The main purpose of the Flask backend is to respond to API requests sent by the frontend, as well as establishing a connection with the database so that information can be stored and retrieved. To achieve this, we define a schema in `models.py`. Further explained in the next section, this schema is responsible for creating and

defining tables in the database. The configuration file is located in `config.py`, where the class `ApplicationConfig` is defined to hold configuration-specific parameters. The contents of the class can be seen in Figure 6.5.

```
class ApplicationConfig:
    SECRET_KEY = os.environ["SECRET_KEY"]

    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SQLALCHEMY_ECHO = False
    SQLALCHEMY_DATABASE_URI = r"mysql://root:@localhost/munchmice"

    CORS_HEADERS = r"application/json"

    SESSION_TYPE = "redis"
    SESSION_PERMANENT = False
    SESSION_USE_SIGNER = True
    SESSION_REDIS = redis.from_url("redis://127.0.0.1:6379")
```

Figure 6.5: Configuration class for Flask backend.

Security is another responsibility of the Flask setup. Bcrypt [47] is used to provide a de-optimised hash for user passwords. Flask-Session [48] is used to add support for server-side sessions, which ensures that data is not stored in cookies visible to the client.

```
bcrypt = Bcrypt(app)
server_session = Session(app)
```

Figure 6.6: Setting up bcrypt and server-side sessions.

### 6.2.1 Web Scraping

The implementation of the scraper can be found in `scraper.py`. Upon receiving a request from the API service, the `getRecipe()` function in `scraper.py` is called with the requested URL. Then, the contents of the requested page is obtained with the help of Requests [36] and Extruct [49]. Requests is a HTTP library for Python,

that allows HTTP requests to be sent in a very simple way. Once the contents of the page have been obtained and decoded via the use of `text()`, specific information can be extracted using `extruct.extract()`, resulting in a structure that only contains embedded metadata, which includes both JSON-LD and Microdata formats. The code for this process is shown in Figure 6.7.

```
r = requests.get(url)
base_url = get_base_url(r.text, r.url)
data = extruct.extract(r.text, base_url=base_url)
```

Figure 6.7: Using Requests and Extruct to obtain embedded metadata.

As the resulting data structure contains all types of metadata, the correct information needs to be identified. Hence, the exact block that contains recipe information needs to be located. A function named `findSchema()` is implemented for this purpose. When called with the metadata block, it first looks for the presence of either an entry with the JSON-LD type or the Microdata type. Once found, it searches for an entry with the `Recipe` type to return. If the data does not contain any JSON-LD, Microdata, or `Recipe` entries, it returns `None`. Once the `Recipe` entry is found, the attribute in Table 6.1 can be retrieved by simply specifying `recipe['attribute']`.

Table 6.1: Attributes of recipe data, their types, and their corresponding name defined by Schema.org.

| Attribute | Data Type | Schema Name |
|---|---|---|
| Name | String | `name` |
| Source | String | None (Input Parameter) |
| Average Rating | Decimal | `ratingValue` |
| Number of Ratings | Integer | `ratingCount` / `reviewCount` |
| Cooking Time (Mins) | Integer | `totalTime` |
| Recipe Yield | Integer | `recipeYield` |
| Ingredients | Array | `recipeIngredient` |
| Directions | Array | `recipeInstructions` |

Once the attributes are retrieved from the metadata, many of them require formatting or extra processing before they can be stored into the database. The scraper is also responsible for this process, which is described below for each attribute. A dictionary object named `recipe` is created so that formatted values can be assigned to it before being returned. (verify types of each attribute with schema.org)

**Name** The name of the recipe is straightforward: a check is needed to ensure that it is non-empty, then it can be assigned to the `recipe` object.

**Source** The source of the recipe does not require scraping, as it is simply the URL inputted by the user.

**Ratings** The rating of a recipe is a structure itself named `aggregateRating`, and contains more information than we need. Hence, once confirmed that this structure exists and is non-empty, `ratingValue` can be obtained from it, as well as the number of ratings, which may be named as either `ratingCount` or `reviewCount`.

**Cooking Time** The cooking time, according to Schema.org, is included in ISO 8601 duration format [50]. Each duration is represented as **P**dD**T**hHmMsS, where **P** designates duration and **T** designates time components. D, H, M, and S respectively designates Days, Hours, Minutes, and Seconds, whereas d, h, m, and s represents the number of each. To convert the duration string into a more readable format represented by only minutes, we implement a function named `convertIsoToMinsInt()`, which splits the ISO string on the aforementioned designators and converts each duration to minutes. This can then be stored as an integer value in the `recipe` object.

**Recipe Yield** The yield of a recipe comes in a few different formats, but we want to store it as an integer for consistency. Strings usually only require a type cast,

however, sometimes strings such as "Yields 20 cookies" can appear. In this type of situations, a simple RegEx search with `\d+` is done on the string to extract just the number.

**Ingredients** Formatting the ingredients is more challenging than the rest of the fields, as it requires understanding the meaning of parts of the ingredient string. This is discussed in detail in the next subsection, Ingredient Formatting.

**Directions** Directions typically appear as a structure of its own, containing each of its children as a sub element. As only the text part of each direction is required, it needs to be extracted from the structure using `direction['text']`.

Lastly, once the `recipe` dictionary object is returned by `scraper.py`, the information is formatted and added to a new `Recipe` class, of which the schema has been defined earlier. This recipe entry is then added to the database, but not yet committed. Next, for each ingredient and direction in the recipe, a corresponding `RecipeIngredient` or `Direction` object is created and assigned values, before also being added to the database. Only then the changes are committed, in order to prevent errors resulting in partial entries, such as a recipe with missing ingredients or directions.

#### 6.2.1.1 Ingredient Formatting

Based on the attributes defined in Table 5.2, we have made the assumption that an ingredient string is in the form of {`quantity`} {`unit`} {`ingredient`}. On top of this, we also assume that the optional additional instruction will be at the end of an ingredient, separated by a comma or an open bracket. For the implementation, the function `formatIngredients()` is written specifically to convert a list of ingredients to a dictionary of tokenised ingredients.

For each ingredient, `formatIngredients()` first standardises any units found in the ingredient string, as demonstrated in Figure 6.8. A reference table for the recog-

nised units and their alternative formats can be found in Table 6.2. Then, if any standardised unit is found, the string is split on that unit. The part before the found unit, if non-empty, is assigned as the quantity, and the part found after is assigned as the ingredient name. The quantity itself goes through another formatting process named `formatQuantity()`, which checks whether the quantity is a fraction and formats it accordingly.

```
# Regex to make sure we can find units stuck together with numbers, e.g. 20g, 5lb
m_regex = re.compile(r'([.\d]+)\s*(?:' + unit + ')\s')
measurement = m_regex.search(original)
```

Figure 6.8: Using regex to find standardised units.

Table 6.2: List of recognised units.

| Unit | Alternative formats |
|---|---|
| tsp | teaspoons, teaspoon, tsps |
| tbsp | tablespoons, tablespoon, tbsps |
| cup | cups |
| oz | ounces, ounce |
| kg | kilograms, kilogram |
| ml | millilitres, milliliters, millilitre, milliliter |
| lb | pounds, pound, lbs |
| fl oz | fluid ounces, fluid ounce |
| g | grams, gram |
| l | litres, liters, litre, liter |

Otherwise, we try to split out the quantity by simply matching the first number that appears in the string. Then, the unit is left as `None`, and the ingredient name becomes the string that follows the unit. If a number could not be found, it means the string is likely something like "icing sugar to finish", which will simply be stored as the ingredient name.

Both the quantity and unit are optional, but the ingredient name must be non-empty. This allows for a degree of error tolerance: should the function incorrectly assigns "pinch of salt" as ingredient name instead of correctly extracting "salt", there is no perceived sense of incorrectness when displayed to the user. A further step to ensure fault tolerance is taken by storing the original complete ingredient strings. This allows the program to back-reference and restore an ingredient, should there be an error in formatting or another fault.

### 6.2.2 Price Estimation

The pricing tool currently uses a CSV file with the records of 1013 ingredients and their respective prices. As aforementioned, this data was obtained from Trolley's Grocery Price Index. The pricing function, `getIngredientPrices()`, can be found in `pricer.py` and is called upon a request is received by the Flask application for a specific recipe. The function itself takes as parameter a list of ingredient names, and returns a nested list with the found ingredient match and its price.

The pricer first loads ingredient price data into a Pandas [51] dataframe for easier access. Then, for each ingredient in the requested list of ingredients, it selects the best match in the dataframe as follows.

**Finding Match Candidates** To avoid excessive traversals of the dataframe, just one search is done for each ingredient. In this search, we look for a list of potential matches for the ingredient string. The condition for a potential match is that the one word of the input string must be a substring of the ingredient in the data set. For example, for the ingredient "apple", one matching candidate could be "apple sauce". Figure 6.9 shows the code for this part, and it also accounts for the case of the ingredient being "apples", which it handles by removing the last letter in the word.

```
for index, row in ingredientsData.iterrows():
    for term in searchStrTerms:
        if (term in row["Name"].lower()) or (term[:-1] in row["Name"].lower()):
            resultCandidates[row["Name"]] = row["Price"]
```

Figure 6.9: Finding matching candidates for an input string.

**Finding the Best Match** Within the found match candidates, we conduct a round of searching for the best match. This is defined as the two full strings being an exact match. If this is not found, we proceed to another search for a still satisfactory match, which is where a candidate is the same as the last word in the input string. One instance of this for the input string "apple" would be that there are no exact matches, but there is the candidate "gala apple". This process is shown in Figure 6.10.

```
for i in resultCandidates:

    if searchStr.lower() == i.lower():
        topResult = i
        topPrice = resultCandidates[i]
        break

    if (searchStrTerms[-1] == i.lower()) or (searchStrTerms[-1] + "s" == i.lower()):
        topResult = i
        topPrice = resultCandidates[i]
```

Figure 6.10: Finding exact or preferred match for an input string.

**Finding Less Preferable Matches** In the case that the better matches cannot be found, the list of candidates is traversed again to look for slightly poorer candidates. A candidate qualifies for this match, if any words in the ingredient string is the exact same as the candidate. Shown in Figure 6.11, this might match an input of "apple" to the candidate "apple slices".

```
if not topResult:
    for i in resultCandidates:
        for term in searchStrTerms[:-1]:
            if (term == i.lower()):
                topResult = i
                topPrice = resultCandidates[i]
                break
```

Figure 6.11: Finding the third best match for an input string.

**Finding the Most Similar String** The last matching attempt is the "safety net" of the matching system. If there are still no matches found, we sort the candidates by their Levenshtein Distance [52] to the input string. The Levenshtein Distance or edit distance between two strings $a$ and $b$ is given by $lev(a, b)$, where

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \operatorname{lev}\big(\operatorname{tail}(a), b\big) \\ \operatorname{lev}\big(a, \operatorname{tail}(b)\big) \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) \end{cases} & \text{otherwise} \end{cases}$$

After sorting the candidates, the top result is taken to be the closest match, as shown in Figure 6.12. The only case where this does not return a match is if there are no matching candidates, in which case it will return "No match found", and the user will be alerted of this information.

```
resultSorted = dict(sorted(resultCandidates.items(), key=lambda item: distance(item[0], searchStr)))
topResult = next(iter(resultSorted))
topPrice = resultSorted[topResult]
```

Figure 6.12: Ranking candidates by the smallest Levenshtein Distance.

## 6.3 MySQL Database

Flask SQLAlchemy (source) is used to establish a connection to the database and add support for SQLAlchemy to Flask. SQLAlchemy is a Object Relation Mapper, which essentially maps object parameters found in Python to the structure of an RDBMS table, and allows the developer to easily perform operations on the database without directly using SQL. Flask requires an URI for the database, and since this project currently runs on localhost and uses MySQL, the URI can be set as `mysql://root:@localhost/munchmice`.

The schema of the database tables are defined according to the schema specified in the Design section. In Flask, this can be achieved by creating model classes for each table in `models.py`. In Figure 6.13, an example is shown for the direction table. Field names as well as data types are declared. For any foreign keys, `recipeID` in this case, a reference to a column in a different table is made. This relationship is also specified in the other table as seen in Figure 6.14, where the action to take on deletion is defined. In the direction table, `recipeID` is non-nullable, which reflects the cascading nature of deletion. That is, since a direction always belongs to a recipe, there is no way to remove it if it was not removed by cascading deletion. Hence, each direction must have a reference to a recipe. Similar logic follows for ingredient (recipe), recipe (cookbook), cookbook (user), shoppinglistIngredient (shoppingList), and shoppingList (user).

```python
class Direction(db.Model):
    __tablename__ = "directions"

    id = db.Column(db.Integer, primary_key=True)
    text = db.Column(db.Text)
    order = db.Column(db.Integer)
    recipeID = db.Column('recipeID', db.Integer, db.ForeignKey('recipes.id'),
    nullable=False)

    def __init__(self, text, order, recipeID):
        self.text = text
        self.order = order
        self.recipeID = recipeID
```

Figure 6.13: Creating a model class for recipe directions.

```python
directions = relationship("Direction", cascade="all,delete", backref="recipes")
```

Figure 6.14: Reference from the recipes table to the directions table.

Another integration layer, Marshmallow [53], is used to serialise the Python class data returned by SQLAlchemy. This is because our API works with JSON-formatted data, and so the more complex data structures returned by SQLAlchemy requires reformatting and converting into JSON data types that can be understood by the frontend scripts.

The database can be viewed and accessed using phpMyAdmin [54], shown in Figure 6.15. A popular administration tool, its purpose in this project was mostly that of testing and checking the data entered or retrieved is consistent with expectations.

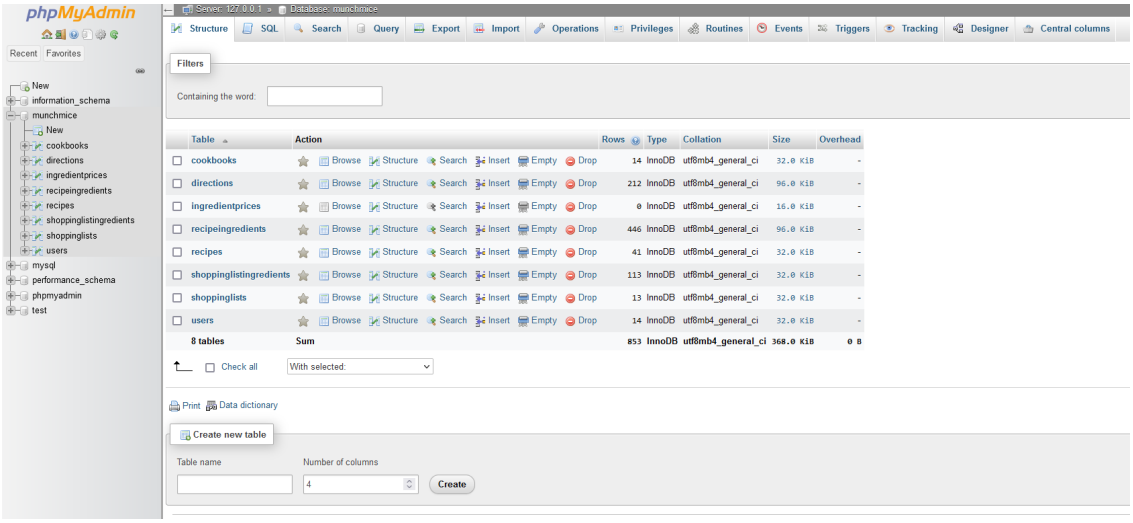


Figure 6.15: Interface for phpMyAdmin.

## 6.4 Integration and API

Now that the Flask application can communicate with the database, a method of communication between the React and Flask sides need to be established. This is achieved via the use of `APIService.js` which utilises `fetch` to request data from routes that are established in `app.py`. More specifically, the same retrieval method between `GET`, `PUT`, `POST`, and `DELETE` is selected for both sides, and a fetch request, shown in Figure 6.16, is sent. Depending on the request, additional information such as a particular recipe ID might be included as part of the route, whereas information such as URLs or recipe details are included as the body of the request. When the Flask application receives the request, it queries the corresponding table using a Python-like syntax (as opposed to SQL), then returns the results in a JSON format, as shown in Figure 6.17.

```
static GetRecipeIngredients(recipeID) {
    return fetch(`http://127.0.0.1:5000/getRecipeIngredients/${recipeID}}`, {
        method: 'GET',
        credentials: 'include',
        headers: {
            'Content-Type': 'application/json',
        },
    }).then((resp) => resp.json())
}
```

Figure 6.16: API service to query a specific route.

```
@app.route('/getRecipeIngredients/<id>', methods = ['GET'])
def get_recipe_ingredients(id):
    all_ingredients = RecipeIngredient.query.filter_by(recipeID=id)
    results = recipeIngredients_schema.dump(all_ingredients)

    return jsonify(results)
```

Figure 6.17: Corresponding route in backend.

## 6.5 Code Management

The front and backend source code are organised into separate directories within the home directory. Their respective file structures can be found in Figures 6.18 and 6.19.


Figure 6.18: File structure of frontend source code.

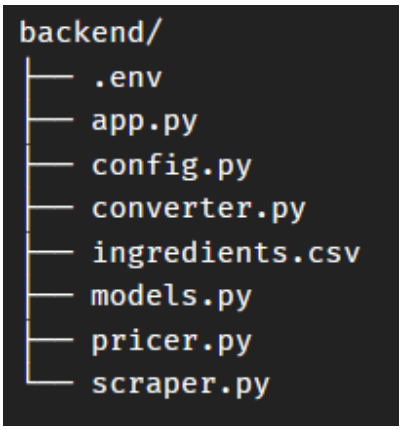


Figure 6.19: File structure of backend source code.

The use of GitHub also added structure to the code. First and foremost, it ensured that progress is tracked and hardware failures do not result in code loss. On top of this, GitHub also facilitated the possibility of working branches on top of the main branch, which works particularly well with the iterative development used in this project, as finished and debugged MVPs were able to be locked behind versions in the main branch.

# Chapter 7

# Testing and Results

At this point in the project, all major code implementations have been completed. This chapter will cover the three testing approaches taken for this project and present their results; this includes unit testing, integration/functional testing, and user/non-functional testing. Then, the final software product is showcased and discussed.

## 7.1 Unit Testing

Unit testing is typically understood as the isolated testing of individual parts of a system, where each part is the smallest unit the system can be broken down into. In the context of a web application, this is separated into testing React components in the frontend, and testing Flask and API routes in the backend.

Unit testing in React is done as each component is implemented. Though there are JavaScript testing frameworks available, such as Jest [55], most of the frontend features are hand-tested to ensure correctness. Should Munchmice be released to the public, automated tests will then become essential. Flask and the JavaScript API are tested with a more systematic approach: Postman [56] is utilised to ensure each route functions as expected, and examples of this can be seen below in Figures 7.1 and 7.2, as well as Figures 7.3 and 7.4.

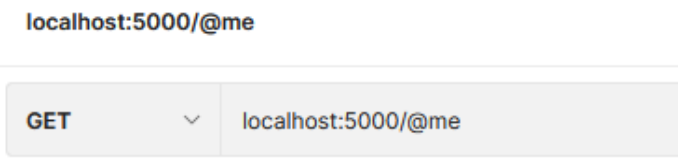


Figure 7.1: Requesting for user information as a logged-in user in an active session.

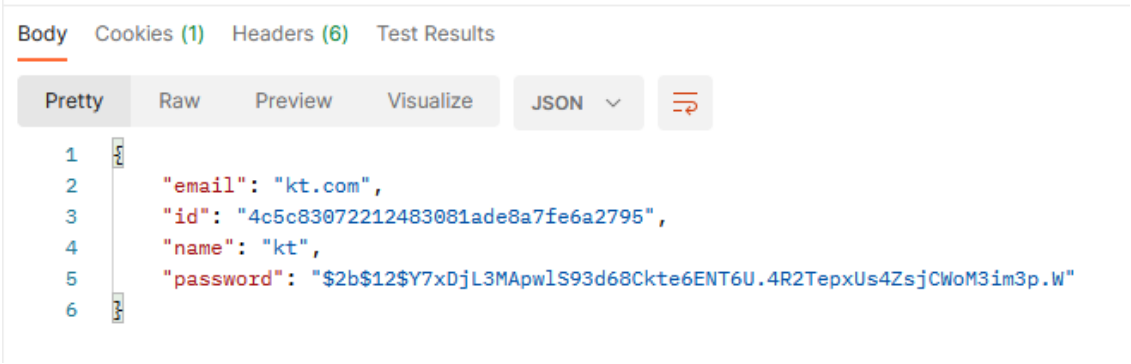


Figure 7.2: Response from API linked to corresponding Flask route.

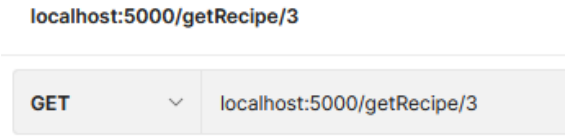


Figure 7.3: Requesting for recipe information, where recipe is owned by current user.


Figure 7.4: Response from API linked to corresponding Flask route.

Test summary of API routes are summarised in Table 7.1 below. For each route, edge cases as well as inputs which would result in errors are considered.

Table 7.1: API service test summary.

| Name | Method | Passed |
|---|---|---|
| `UpdateRecipe(id, body)` | PUT | Yes |
| `DeleteRecipe(id)` | DELETE | Yes |
| `ConvertUnits(id, body)` | PUT | Yes |
| `AddRecipeByURL(body)` | POST | Yes |
| `GetRecipe(id)` | GET | Yes |
| `GetRecipes(userID)` | GET | Yes |
| `GetIngredientPrices(id)` | GET | Yes |
| `GetShoppingListItems(userID)` | GET | Yes |
| `AddShoppingListItem(userID, body)` | POST | Yes |
| `AddShoppingListItems(userID, body)` | POST | Yes |
| `DeleteShoppingListItem(id)` | DELETE | Yes |
| `LoginUser(body)` | POST | Yes |
| `LogoutUser()` | POST | Yes |
| `RegisterUser(body)` | POST | Yes |
| `IsLoggedIn()` | GET | Yes |
| `GetRecipeIngredients(recipeID)` | GET | Yes |
| `GetRecipeDirections(recipeID)` | GET | Yes |

## 7.2 Integration and Functional Testing

As the system does not use a large software stack, there is typically little difference between testing the integration between two systems, and testing the behaviour of the entire system; it would be redundant to try to draw a line between the two approaches. Hence, they are both covered by the requirement-based functional tests, where test cases are created as per the requirement analysis. A brief outline of test cases, as well as their results, can be found in Table 7.2.

It should be noted that the test cases used do not correspond to individual requirements on a one-to-one basis; instead one test case often covers multiple requirements

within the same epic. Furthermore, test cases have only been designed for implemented features, as opposed to all requirements. Passing criteria for the two core functionalities of the system, the web scraper and the price estimator, are further expanded on in the sections below; any tests associated with these criteria have been annotated with the asterisk (*).

Table 7.2: Functional requirement tests.

| Epic | Test ID | Test Case | Status |
|---|---|---|---|
| Recipe Import and Creation | 1.1 | Import recipe from valid URL | Passed* |
| | 1.2 | Import recipe from invalid URL | Passed |
| | 1.3 | View recipe details | Passed |
| | 1.4 | View recipe details as unauthorised user | Passed |
| Recipe Organisation | 2.1 | View cookbook recipes | Passed |
| | 2.2 | View cookbook as unauthorised user | Passed |
| | 2.3 | Sort recipes by name | Passed |
| | 2.4 | Sort recipes by rating | Passed |
| | 2.5 | Sort recipes by cooking time | Passed |
| | 2.6 | Access recipe via cookbook page | Passed |
| | 2.7 | Access recipe via side bar | Passed |
| Recipe Editing | 3.1 | Switch to recipe edit view | Passed |
| | 3.2 | Modify recipe name | Passed |
| | 3.3 | Modify recipe cooking time | Passed |
| | 3.4 | Modify recipe ingredients | Passed |
| | 3.5 | Modify recipe directions | Passed |
| | 3.6 | Scale recipe servings | Passed |
| | 3.7 | Convert recipe measurement units | Passed |
| | 3.8 | Delete recipe | Passed |
| Ingredient Pricing | 5.1 | View recipe cost estimate | Passed* |
| | 5.2 | View cost estimate after ingredient edit | Passed* |
| User Profile | 6.1 | View dashboard | Passed |
| Authentication | 7.1 | Register for an account | Passed |
| | 7.2 | Log into the site | Passed |
| | 7.3 | Log in with incorrect credentials | Passed |
| Shopping List | 8.1 | View shopping list | Passed |
| | 8.2 | Add item to shopping list | Passed |
| | 8.3 | Add items from recipe to shopping list | Passed |
| | 8.4 | Remove an item from shopping list | Passed |
| | 8.5 | Remove multiple items from shopping list | Passed |

When considering long-term development and further iterations, automated testing instead of manual tests conducted in this instance is preferred, as each new feature

increases the number of tests required. Test suites such as Selenium [57], which provide automation for browsers, could be appropriate options for automated testing. However in this case, given the relatively low number of test cases contrasted with the learning curve of automated test suites, a manual approach was chosen.

### 7.2.1 Web Scraper

To test the accuracy of the web scraper, a data set containing 50 URLs linking to valid recipes hosted on various platforms was used. Each link was inputted into the `getRecipe()` function, and the results were inspected and compared against the original recipe to check for inconsistencies.

The results of scraper testing is shown in Table 7.3. In summary 41 out of 50, or 82% of the test samples, were correctly imported. Of the 9 unsuccessful samples, 8 failed due to incorrect ingredient formatting. For example, in one instance, the scraper did not know how to process an ingredient string with a range ("2-3 eggs"). This specific issue can be fixed by matching for similar numerical ranges using regular expressions, then taking an average of the two values; or better yet, modifying the database schema so that there is a structure to store ranges appropriately. The general takeaway from these 8 failed samples, however, is that there is no standardised format for ingredient strings, and hence there are many distinct cases to account for in the ingredient formatter. This is an issue that can be alleviated with repeated testing and improvement of code, but it is unlikely to ever be completely eliminated.

The specific requirement for ingredient processing, R1.3, specifies that ingredients must be processed into meaningful attributes. Though 16% of recipes tested encountered issues with ingredient processing, it is generally the case that each recipe had a single erroneous ingredient, hence the actual rate of ingredient errors is much lower. The accuracy of the scraper as a result becomes >90%.

Table 7.3: Web scraper test results on data set of size 50.

| Total | Success | Comment |
|---|---|---|
| 50 | 41 Successful | 41 - Recipe imported without issue |
| | 9 Unsuccessful | 8 - Recipe imported with ingredient formatting issue |
| | | 1 - No structural metadata found in response |

Within the 9 unsuccessful samples, one could not be imported at all. This is due to it being a WordPress [58] site, and the creator did not set up any type of metadata formatting. As discussed in the Design section, the web scraper cannot process sites without structural metadata. If Munchmice was to be truly comprehensive in processing recipe sites, a HTML traversal algorithm could be implemented; however, as defined at the start, this feature is out of scope for this project, and hence the test failure is expected. Furthermore, the appearance rate of sites without metadata in this test data set comes to 2% - naturally, this is not a sufficiently large sample size to draw conclusions from, but it is a promising indicator.

The results of the conducted tests have demonstrated satisfactory accuracy of the scraper. Hence, test case 1.1 is considered to be passed.

### 7.2.2 Price Estimation

The pricer is the subsystem responsible for recipe price estimation. In its testing, two aspects are considered: firstly, as ingredients from a recipe needs to be matched to entries in the ingredient price database, the accuracy of this matching process is evaluated. Then, given a recipe with correctly matched ingredients, the estimated price given by Munchmice is compared against prices found at the supermarkets Tesco and Aldi.

For ingredient matching of the pricer, ingredients from 30 imported recipes are used, yielding a total of 106 unique ingredients. Once again a manual process, each set of ingredients from recipes are inputted into the function `getIngredientPrices()` in `pricer.py`. The results are shown in Figure 7.5, where it can be seen that 81%

of ingredients are correctly matched, and 19% are incorrectly matched.

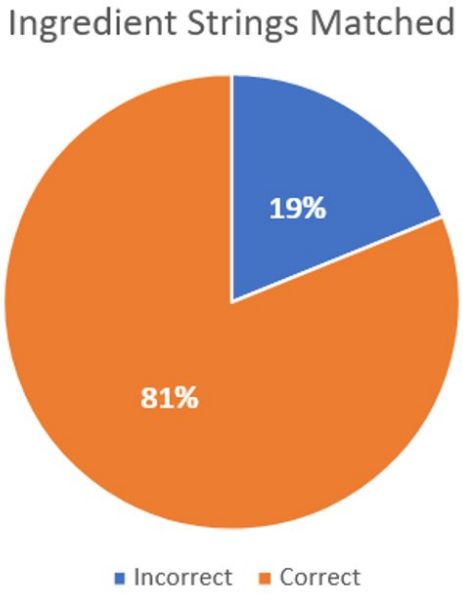


Figure 7.5: Percentage of correct ingredient matches.

The incorrect ingredient matches are attributed to two causes, which each account for 9.5% the errors:

1. **Mismatched substring**: When a string contains two ingredient words, the pricer prefers to match the latter word. This is because the descriptor of the ingredient can likely be interpreted as an ingredient itself. One example is "plum tomatoes", where the first word, "plum", is a type of the actual ingredient, "tomatoes". However, issues arise when this is not the case. "cloves of garlic" is often written as "garlic cloves", and as "cloves" is an ingredient itself, the ingredient formatter incorrectly interprets the ingredient as "cloves".

2. **Ingredient not found**: The ingredient itself does not exist in the database. This is an expected error for a small part of the ingredients, because our data set for ingredient prices is not entirely comprehensive. Hence more obscure ingredients, for example "yellow bean paste", cannot be matched accurately.

When there is a mismatched subtring, a general solution may be to filter out counters that are followed by "of". This involves building a list of commonly used

counter words. The only way to address the second issue, however, is to expand the ingredient price database. In this project, the price data is sourced from Trolley, but in future development, Munchmice could build its own database by crowd sourcing information - this will be discussed in the Future Works section in the conclusion of the report.

It should also be taken into account that many common ingredients are shared by similar recipes. For baking, this includes ingredients such as sugar, flour, and egg; for main meals, oil, salt, and onions are often present across recipes. Retrospectively, it might have been useful to also find the matching accuracy for total ingredients as well as unique ingredients, in order to better demonstrate this point. Nevertheless, the perceived matching accuracy of the pricer is higher than that of the unique ingredients shown in the test above.

A comparison between Munchmice's price estimation and actual costs of a recipe is shown below in Figure 7.6.

| Margherita Pizza | | | | |
|---|---|---|---|---|
| Ingredient | Cost (Munchmice) | Tesco (regular) | Tesco (cheapest) | Aldi |
| Olive oil | £ 5.80 | £ 3.70 | £ 2.20 | £ 1.85 |
| Plain flour | £ 1.46 | £ 0.80 | £ 0.58 | £ 0.58 |
| Yeast | £ 1.09 | £ 1.25 | £ 1.20 | n/a |
| Sugar | £ 1.89 | £ 2.25 | £ 0.99 | £ 0.99 |
| Passata | £ 1.74 | £ 1.20 | £ 0.45 | £ 0.45 |
| Mozzarella | £ 2.23 | £ 1.85 | £ 0.69 | £ 0.69 |
| Parmesan | £ 2.23 | £ 2.35 | £ 2.10 | £ 3.59 |
| Basil | £ 1.12 | £ 1.10 | £ 0.52 | £ 0.52 |
| Total | £ 17.56 | £ 14.50 | £ 8.73 | £ 8.67 |

Figure 7.6: Price comparison for the ingredients of a margherita pizza recipe.

It is evident that there is a large difference between Munchmice's estimate (second column) and the cheapest cost items from Tesco [59] (fourth column) and Aldi [60] (last column), with the estimate at about twice the cost of others. However, the

Tesco prices usually indicate those of the "basic" range, and when we move onto the regular, more mid-range items, the total becomes a lot closer to our estimate. The same pattern is exhibited in a number of recipes, another one of which is shown in Figure 7.7.

| Banana Bread | | | | |
|---|---|---|---|---|
| Ingredient | Cost (Munchmice) | Tesco (regular) | Tesco (cheapest) | Aldi |
| Butter | £ 3.26 | £ 1.99 | £ 1.99 | £ 1.99 |
| Caster sugar | £ 3.26 | £ 2.25 | £ 2.00 | £ 1.45 |
| Eggs | £ 2.45 | £ 2.50 | £ 2.50 | £ 1.29 |
| Self-raising flour | £ 1.80 | £ 0.80 | £ 0.58 | £ 0.58 |
| Baking powder | £ 1.36 | £ 1.39 | £ 0.52 | £ 1.39 |
| Bananas | £ 0.94 | £ 1.35 | £ 1.35 | £ 0.32 |
| Icing sugar | £ 1.92 | £ 2.30 | £ 2.30 | £ 0.79 |
| Total | £ 14.99 | £ 12.58 | £ 11.24 | £ 7.81 |

Figure 7.7: Price comparison for the ingredients of a banana bread recipe.

To explain this trend, we need to understand the makeup of the price data. Munchmice's price data is sourced from Trolley, and it is likely that Trolley uses an average of prices between multiple items between multiple supermarkets to calculate their estimate. This may include items from higher quality ranges, as well as items from more expensive supermarkets. The resulting price estimates, therefore, are accurate to the price of mid-range items, and the pricer is considered satisfactory. However, students' budgets should be considered in this aspect. Indeed, the average student may choose cheaper items over mid-range items when grocery shopping, and the possibility for different price tier selection could be explored as an extension to the current system.

## 7.3 Non-functional and User Testing

Most non-functional requirements cannot be covered using test cases. Instead, we opt to verbally discuss their fulfilment, referring back to the requirement specification as necessary. First, the platform requirements shown in Table 7.4. R9.1 is

fulfilled by nature of the software. However, R9.2 was never completed due to time crunch towards the end of the project and complexity of migrating the server.

Table 7.4: Platform requirement specification.

| Category | RID | Requirement | Priority |
|---|---|---|---|
| Platform | 9.1 | The product must be a web application. | Must |
| | 9.2 | The system should be hosted on DCS servers. | Should |

User acceptance testing is an important aspect of the project, as it validates that the final product aligns with users' needs. However, the non-fulfilment of R9.2 has in turn limited user testing possibilities, as the tester's physical presence is required to test the system locally; instead of the previously planned survey aiming to receive 10-20 responses, two users were invited for in-person feedback. The set up of user acceptance tests is as follows: individual testers are first asked to follow the user flow shown in Figure 7.8 to accomplish a set of tasks; then, they are free to experiment with the system as they like. Feedback is gathered in this process and organised in accordance with the usability requirements as listed in Table 7.5.

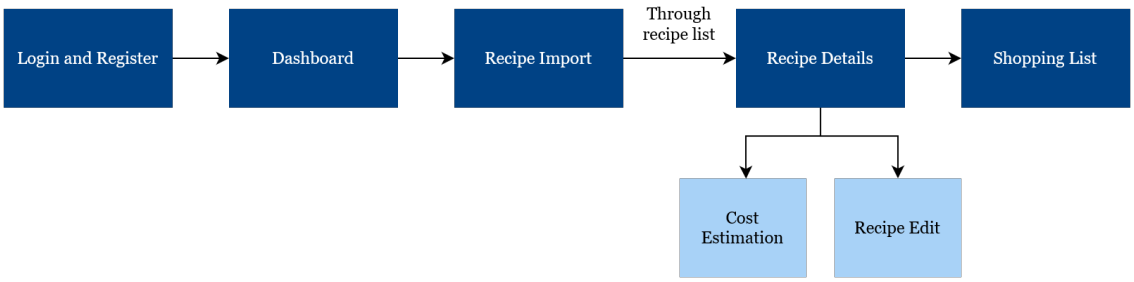


Figure 7.8: User flow used for acceptance testing.

Table 7.5: Usability requirement specification.

| Category | RID | Requirement | Priority |
|---|---|---|---|
| | 10.1 | The system should be intuitive enough for users to operate without tutorial. | Should |

| | | | |
|---|---|---|---|
| Usability | 10.2 | Users should be able to understand the function of clickable components by their appearance. | Should |
| | 10.3 | The user interface must have appropriate colour contrast for accessibility. | Could |
| | 10.4 | The user interface should be pleasant to the eyes. | Could |
| | 10.5 | The system should be fault tolerant in that user actions can either be undone, or warned beforehand. | Could |
| | 10.6 | There could be a customisation feature for users to change the appearance of the system. | Could |

**R10.1 & R10.2 - Fulfilled** The use of icons and their resemblance to real-life items was important for the overall intuitiveness of the system. Users were able to navigate the system and fulfil predefined tasks without confusion or requiring assistance. Affordances, such as scroll bars next to overflowing lists, and highlight/bolding of buttons when hovered (Figures 7.9 and 7.10) made it easy to interpret functionalities of the system, and results predictable.

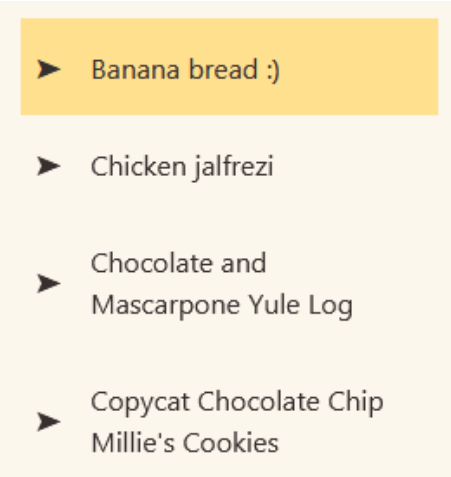


Figure 7.9: Highlight of recipe button when hovered over.

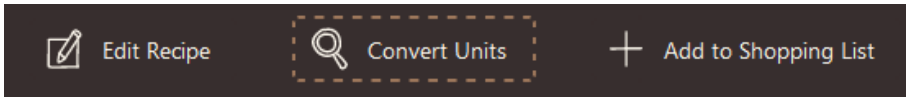


Figure 7.10: Highlight of tool bar button when hovered over.

**R10.3 - Fulfilled** The contrast is deemed appropriate, especially for the notebook contents shown in Figure 7.11 where the user will spend the majority of their time viewing. Other texts have similar contrast: white text is almost exclusively used on dark backgrounds, and vice versa.


Figure 7.11: Notebook containing recipe details.

**R10.4 - Fulfilled** A rather subjective requirement, however users have given feedback that they enjoyed the overall style of the website. In particular, the dashboard was pointed out to be pleasant and design-forward, and the minimalist approach with the general interface was also well received. It was thought that the spacing could be improved in areas such as the top bar. Overall, users rated the design an average of 8.5 out of 10.

**R10.5 & 10.6 - Not Fulfilled** Users felt that there could have been more warnings before certain actions go through. For example, the delete recipe button should bring up a confirmation window before any action is taken; the add recipe ingredient to shopping list function could use a similar warning as well. The originally planned customisation feature was not implemented, due to it being a Could have requirement and time crunch towards the end of the project.

On top of determining the outcome of each requirement, user feedback was also used to make improvements to the user interface. In particular, the serving adjustment interface has caused issues where users can manually set serving size to 0, which does not make sense in the context of the recipe. Hence, this was fixed in Figure 7.12, where the UI has been locked so that a user can only modify the serving size with arrow buttons. Furthermore, a limit between 1 and 100 has been enforced on serving sizes. Another improvement was the submit button for for edited recipes: this was initially at the bottom of the recipe and usually have to be reached by scrolling down. Users felt that this was confusing unclear, and hence the positioning of the button has been fixed to the bottom of the viewport, allowing it to be visible to the user at all times (Figure 7.13).

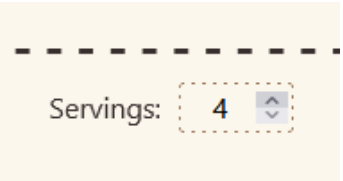


Figure 7.12: UI of scale button changed to only allow editing with arrows.

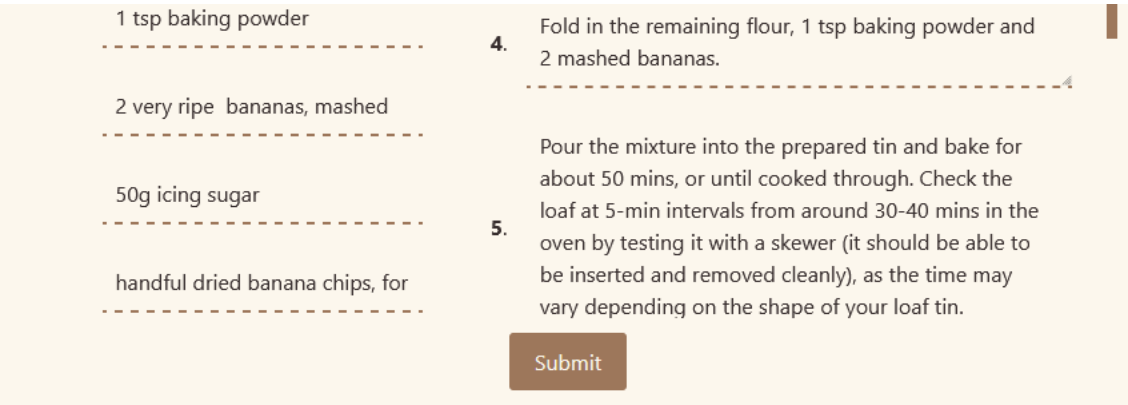


Figure 7.13: Submit button in recipe edit page changed to be fixed at bottom of viewing area.

Table 7.7: Performance and scalability requirement specification.

| Category | RID | Requirement | Priority |
|---|---|---|---|
| Performance and Scalability | 11.1 | The system must be able to support storing at least 50 recipes per user. | Must |

Lastly, the performance of the system is tested. Though it was not deployed onto a central server, local testing added 16 users, responsible for a total of 53 recipes, 252 recipe directions, and 490 recipe ingredient entries. The system did not show any perceivable difference in performance compared to when it was initially built, hosting only one user and a handful of recipes. Although the effects might start to be seen with a further increased magnitude of recipes, as well as simultaneous access by multiple users, it was not feasible to be tested at this stage.

## 7.4 Results

This section showcases the final product by stepping through a typical user flow that includes access to all parts of the system. This involves five main interfaces: login/register, dashboard, recipe import, recipe details, and shopping list. The pipeline seen before in Figure 7.8 outlines our example user flow between these interfaces, as well where other features are interacted with. In the following sections, we will look at each of the main interfaces in more detail.

### 7.4.1 Login and Register

Upon arriving at the site, the user is greeted with a simple sign in page shown in Figure 7.14. If they do not have an account, they can choose to navigate to the registration page shown in Figure 7.15 via the link provided at the bottom of the form.

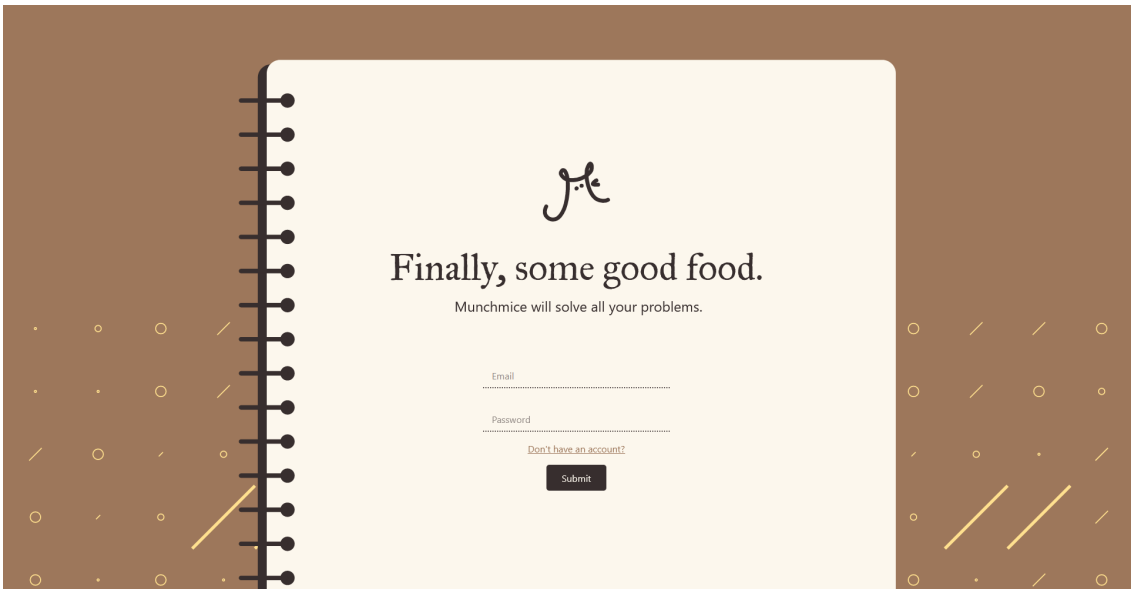


Figure 7.14: Login page.


Figure 7.15: Register page.

### 7.4.2 Dashboard

After signing in, the user arrives at their personal dashboard, shown in Figure 7.16. This interface greets them and provides a summary of their shopping list. In the future, this page could also be used for recipe recommendations. On this page, users can also see the navigation bar for the system, which includes recipe import, cookbook (list of recipes in collection), as well as their shopping list.

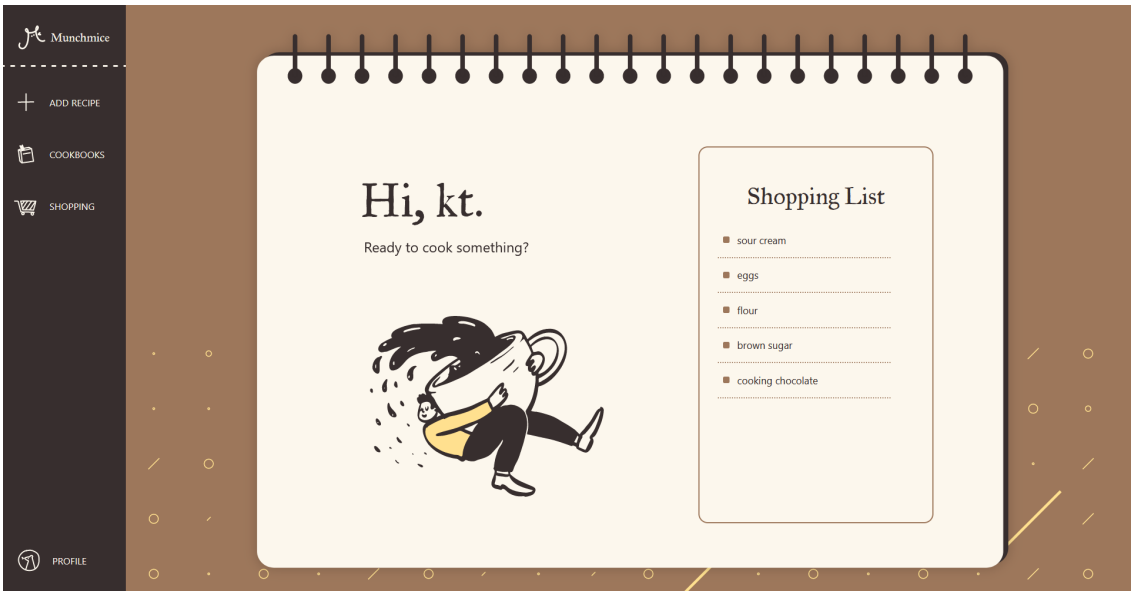


Figure 7.16: User dashboard.

### 7.4.3 Recipe Import

From the dashboard, the user goes to add a new recipe. To do this, they navigate to the "add recipe" interface using the navigation bar. On this page (Figure 7.17), they can paste in the URL of the recipe they would like to import. Note that on this page, and all pages other than the dashboard, the dark navigation bar retracts to leave only the icons, and an inner cookbook bar pops up. With this bar, users can quickly navigate to their recipes.

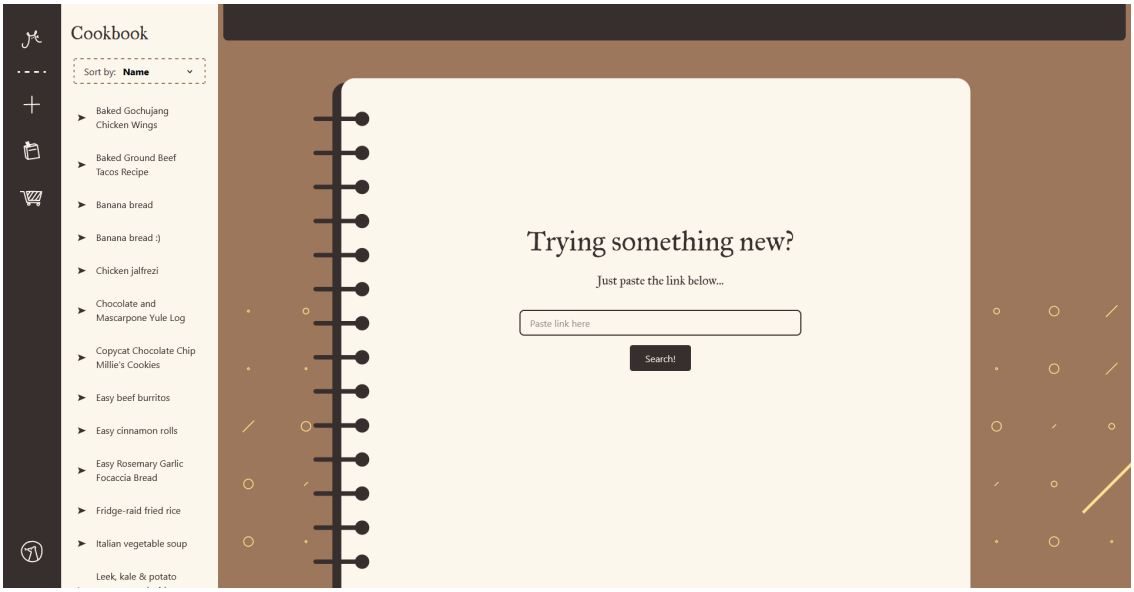


Figure 7.17: Recipe import interface.

### 7.4.4 Recipe Details

After a recipe is successfully imported, the user is automatically redirected to the recipe collections page, shown in Figure 7.18. This page shows a list of all recipes owned by the user, but the user can also choose to navigate using the light cookbook side bar.


Figure 7.18: List of recipes in the cookbook.

After selecting a recipe, the user is brought to the recipe details page in Figure 7.19. The notebook section in this page contains the recipe title, source, star rating, cooking time, and the serving size of the recipe, which can be adjusted by the user. On top of this, the ingredients and cooking directions are also listed here.

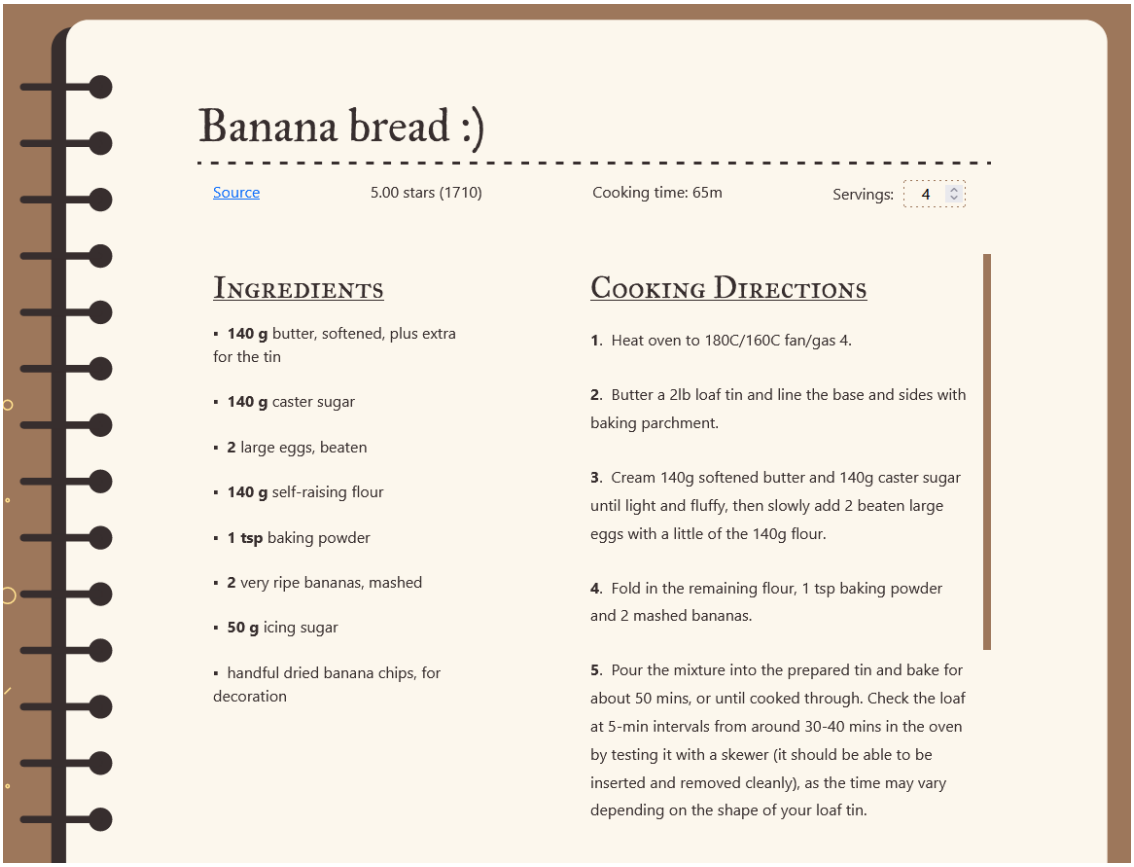


Figure 7.19: Recipe details.

We can see that there is a top bar present in the page with a number of functions, these are: recipe cost details, recipe edit, unit conversion, quick add ingredients to shopping list, and delete recipe. They are represented by icons as well as a description to minimise confusion.

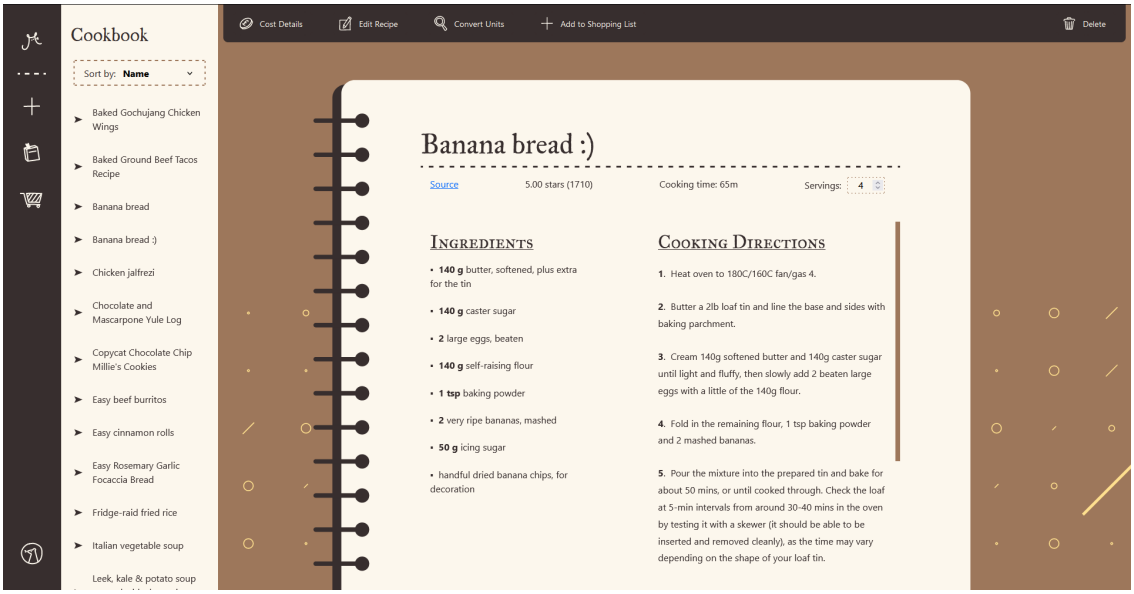


Figure 7.20: Recipe detail page.

By clicking on the cost estimation button, a window showing the cost breakdown of the recipe is brought up (Figure 7.21). Here, the user can see the total cost of their recipe as well as its breakdown by ingredient.

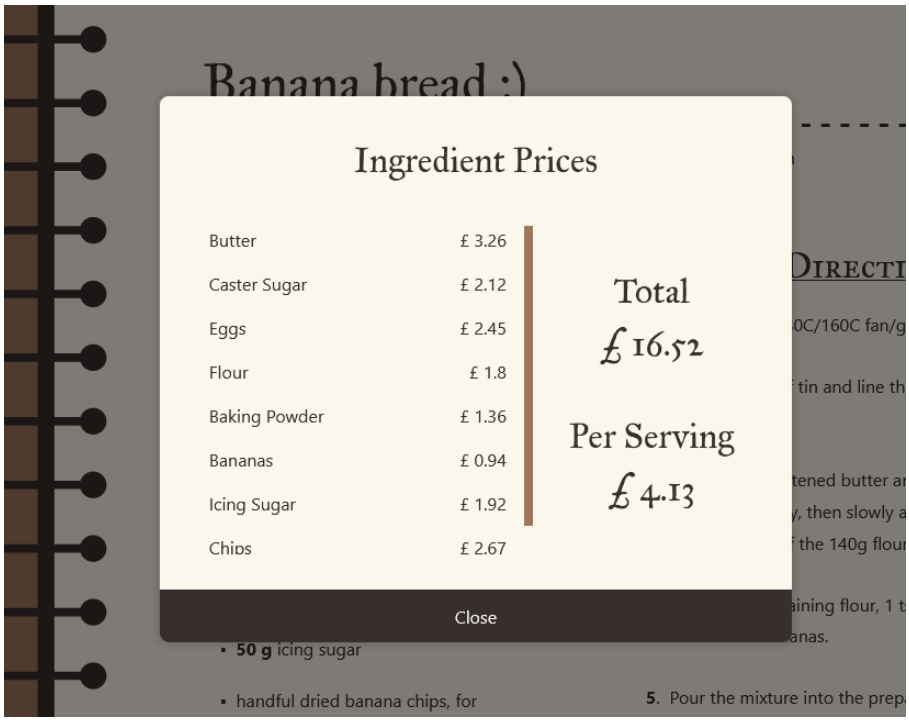


Figure 7.21: Price detail estimation window.

By clicking on the recipe edit button, the user can switch to an editing interface (Figure 7.22) that allows them to modify the recipe. This includes the title, cooking time, as well as the ingredients and directions of the recipe. When done, they can click the submit button to save the recipe.


Figure 7.22: Recipe editing interface.

### 7.4.5 Shopping List

The shopping list can be viewed by clicking on the trolley icon in the navigation bar. On this page, shown in Figure 7.23, users are free to add, delete, or view the items in their shopping list. To delete items from the shopping list, users first have to click on the items to be deleted, then click the update button in the top right corner of the page. This removes all selected items from the shopping list.

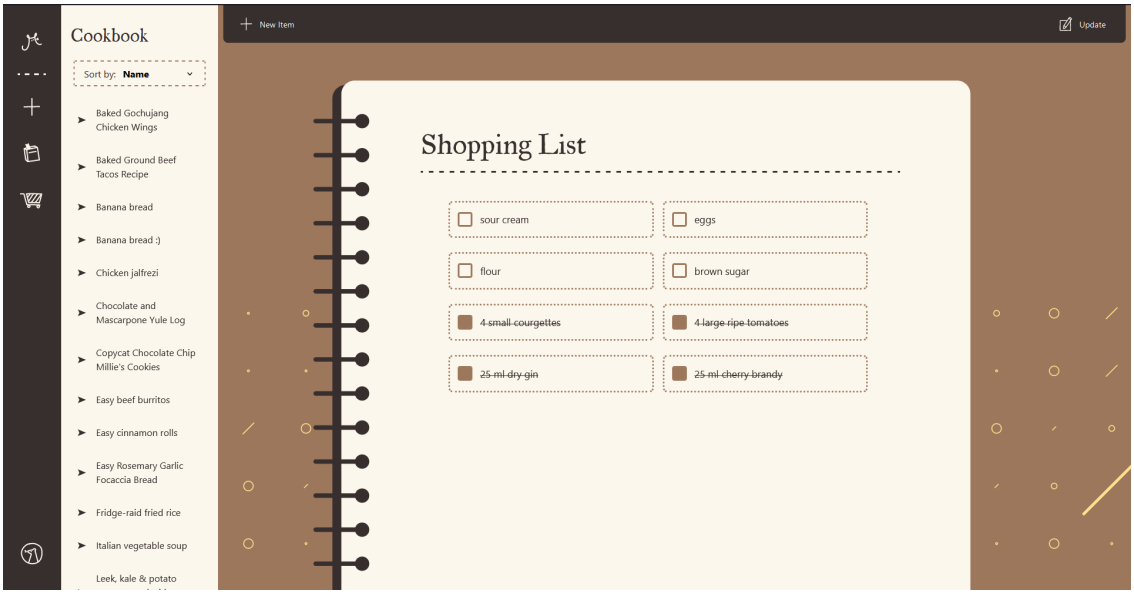


Figure 7.23: Shopping list page.

If a user wish to add an item into the shopping list, they first need to click on the new item button located on the top left of the page. Then, they can enter their item and press submit, which will add the new item into the shopping list. This is demonstrated in Figures 7.24 and 7.25.

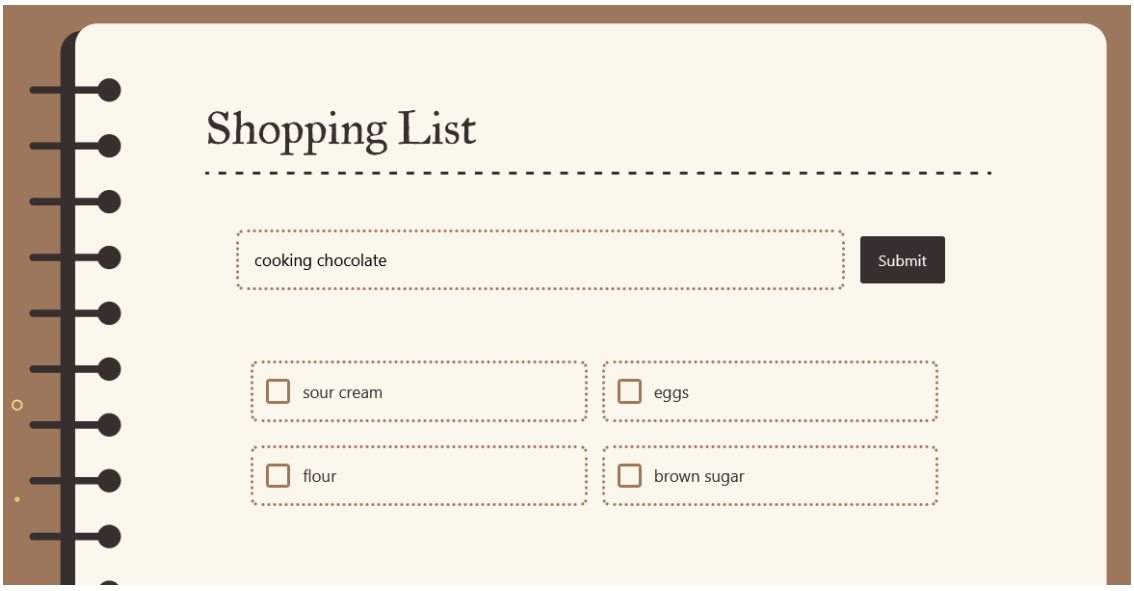


Figure 7.24: Adding a shopping list item.

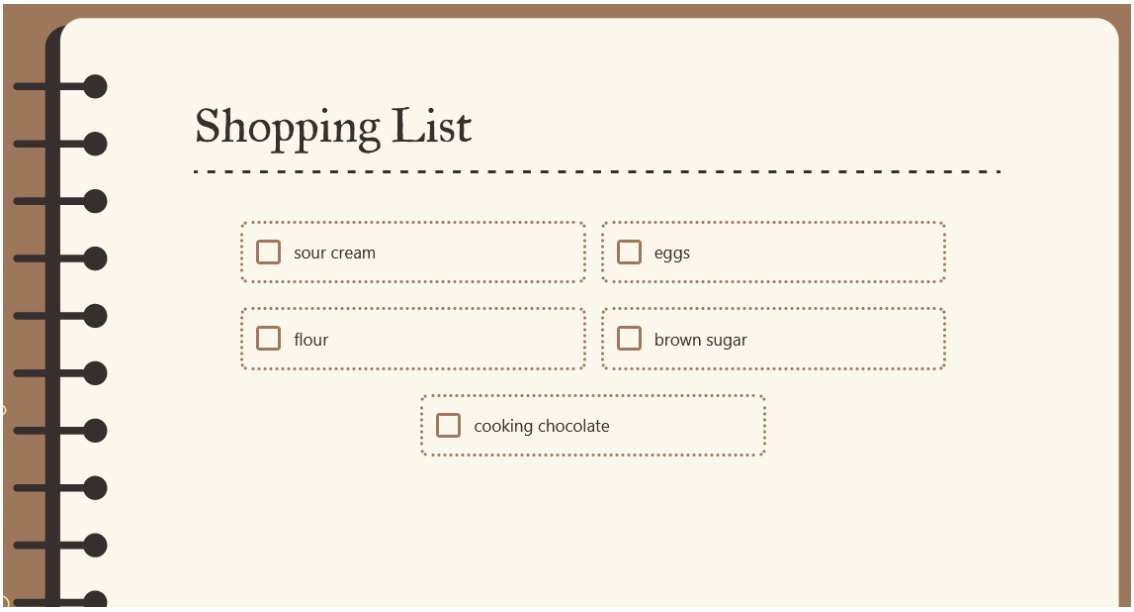


Figure 7.25: Results of adding an item.

# Chapter 8

# Evaluation

Following results from the Testing chapter, an evaluation process is carried out. The primary purpose of this is to measure the success of the project, which involves two parts: evaluation against the original requirement specification, and evaluation based on feedback by users who have tested the system. Additionally, this chapter hopes to evaluate the process of development, which includes the appropriateness of the methodology used, as well as the time management throughout development and any unforeseen circumstances met, in an effort to reflect upon the decisions taken and improve in subsequent projects.

## 8.1 Fulfilment of Requirements

An important document drafted at the start of development is the requirement analysis. As this project is heavily software development oriented, a key contributing factor to its success is the fulfilment status of the initial requirements. In Table 8.1, the completion of each requirement is summarised. The criterion of determining whether a requirement has been met is generally either that the software passed relevant tests in the testing phase, or that the presence of the functionality in the final product already fulfils the requirement. Comments on this aspect, as well as the reasons behind unmet requirements, can be found in the comment column.

Table 8.1: Functional requirement fulfilment status.

| **Functional Requirements Fulfilment** | | | | |
|---|---|---|---|---|
| Epic | RID | Priority | Met | Comments |
| Recipe Import and Creation | 1.1 | Must | Yes | Passed tests |
| | 1.2 | Must | Yes | Passed tests |
| | 1.3 | Must | Yes | Passed tests |
| | 1.4 | Must | Yes | Passed tests |
| | 1.5 | Could | No | Did not implement |
| | 1.6 | Won't | No | Assigned a priority of W |
| Recipe Organisation | 2.1 | Should | Yes | Passed tests |
| | 2.2 | Could | No | Did not implement |
| | 2.3 | Could | No | Did not implement |
| | 2.4 | Could | No | Did not implement |
| Recipe Editing | 3.1 | Must | Yes | Passed tests |
| | 3.2 | Must | Yes | Passed tests |
| | 3.3 | Must | Yes | Passed tests |
| | 3.4 | Must | Yes | Passed tests |
| Cooking Suggestions | 4.1 | Won't | No | Assigned a priority of W |
| | 4.2 | Could | No | Did not implement |
| Ingredient Pricing | 5.1 | Must | Yes | Passed tests |
| | 5.2 | Should | Yes | Passed tests |
| | 5.3 | Should | Yes | Passed tests |
| | 5.4 | Must | Yes | Passed tests |
| | 5.5 | Could | No | Could not implement |
| User Profile | 6.1 | Must | Yes | Passed tests |
| | 6.2 | Could | No | Did not implement |
| | 6.3 | Could | No | Did not implement |
| | 6.4 | Could | No | Did not implement |
| Authentication | 7.1 | Must | Yes | Passed tests |
| | 7.2 | Must | Yes | Passed tests |
| | 7.3 | Must | Yes | Passed tests |
| | 7.4 | Must | Yes | Passed tests |
| Shopping List | 8.1 | Must | Yes | Passed tests |
| | 8.2 | Must | Yes | Passed tests |
| | 8.3 | Must | Yes | Passed tests |
| | 8.4 | Could | Yes | Passed tests |
| | 8.5 | Could | No | Did not implement |

The functional requirement fulfilment status shows that all "Must" and "Should" functional requirements have been implemented, as well as one "Could" requirement, R8.4. This is the key criterion for the success of the project - we have accomplished all that were planned to be achieved within expectation.

It is at this point we should come back to the rationale behind the inception of the "Could" requirements. Owing to the uncertainty of the project, and its experimental nature, it was unclear if the amount of work allocated at the start resulted in an appropriate workload. On top of this, there were a lot of ideas for features that could be useful as part of the system. These factors combined resulted in the decision of adding all suitable non-essential features into the initial requirement analysis, and assigning them a priority of "Could". Therefore, it allows the developer to prioritise the core functions, while also having the freedom to schedule these additional requirements.

Though most of the "Could" requirements were not implemented by the end of the project, it helped when thinking about the overall function and purpose of Munchmice. It also put into perspective the multitude of aspects that need to be considered for a seemingly straight forward system, as well as a glimpse of the work that needs to be done. Additionally, thinking about these requirements gave an idea of what the future direction of Munchmice could look like.

Table 8.2: Non-functional requirement fulfilment status.

| **Non-Functional Requirements Fulfilment** | | | | |
|---|---|---|---|---|
| Category | RID | Priority | Met | Comments |
| Platform | 9.1 | Must | Yes | Implemented |
| | 9.2 | Should | No | Did not set up |
| Usability | 10.1 | Should | Yes | Passed user tests |
| | 10.2 | Should | Yes | Passed user tests |
| | 10.3 | Could | Yes | Passed user tests |
| | 10.4 | Could | Yes | Passed user tests |
| | 10.5 | Could | No | Did not implement |
| | 10.6 | Could | No | Did not implement |
| Performance and Scalability | 11.1 | Must | Yes | Passed tests |

As discussed in the Testing chapter, the system was not set up to be hosted in department servers. The main result of this is the inability to perform more com-

prehensive system performance tests, as well as conducting a wider-scale user acceptance test and survey. In future development, the migration of the system to be hosted on a public facing server is absolutely necessary.

## 8.2 Final Product Evaluation

This section sums up all results found in testing, user feedback, and concluded through evaluation, and produces a final evaluation of the entirety of our system, Munchmice.

The final system provides accurate scraping outputs as demonstrated by test cases. Price estimation of recipes is accurate to mid-range supermarket items, which, when taken into account the lack of a comprehensive price data set, is considered satisfactory. Users have reported that they enjoyed the interface and design of the website in both user acceptance testing and survey questions regarding the wireframe design of the site (Appendix A), and the end result is labelled intuitive and easy to use.

Nonetheless, we felt that the testing done for Munchmice could have been more comprehensive. Further testing should be conducted, with particular focus on the system's performance, as well as a long-term user testing spanning over at least one week of regular usage. This will be more conducive in providing evidence that Munchmice is having a positive impact on student cooking.

Finally, when comparing Munchmice's features to similar tools widely used today, we can see that it does in fact span across the two categories of recipe systems; though it is not as rich in functionality as select recipe management sites, it has successfully bridged the gap between recipe management applications and recipe cost estimation.

| | Original Recipes | Recipe Import | Meal Planning | Recipe Modification | Recipe Collections | Shopping List | Cost Estimation |
|---|---|---|---|---|---|---|---|
| Yummly | ✓ | ✓ | ✓ | | ✓ | ✓ | |
| Paprika | | ✓ | ✓ | ✓ | | ✓ | |
| Recipe Keeper | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Recipe Cost Calculator | | | | | | | ✓ |
| Munchmice | | ✓ | | ✓ | | ✓ | ✓ |

Figure 8.1: Comparison of core functionalities between popular recipe applications and Munchmice.

## 8.3 Methodology, Time Management, and Unforeseen Circumstances

Throughout development, time management remained a key skill carefully exercised by the developer, which ultimately led to the success of the project. First and foremost, all planned core functionalities have been implemented within the time limit. In this part, the Gantt Chart played a key role in ensuring tasks were completed before the deadline. There was a tendency towards perfectionism: regularly referencing iteration plans, and moving tasks down the Kanban board helped curb the compulsion to overspend time polishing features mid-development.

Despite this, the development process was not without complication. The project itself had fallen behind schedule at the point of the presentation, and some work on the code had not been finalised. The Easter holiday, a key time period planned for the majority of report writing and finishing code development, was spent abroad, where an unforeseen turn of events rendered us unable to access key resources such as Google, Wikipedia, and the like. Eventually, we were able to set up an overseas connection on a mobile phone, where we conducted a significant amount of research

for this final report. This resulted in a rush in completing the report in the last few weeks before the deadline, as a lot of catch up had to be done.

Overall, the project was successful due to careful time management which enabled the developer to implement planned functionalities. While the time crunch towards the end of the project was stressful, unexpected obstacles were overcome with hard work, perseverance, and adaptability in the face of challenges.

## 8.4 Personal Evaluation

It is believed that the developer was able to meet project goals with a strong work ethic and dedication throughout development. Regular meetings with the supervisor were organised, and feedback was valued and used to refine the approach to the project. In the process of engineering the project source code, close attention was paid to detail, ensuring its accuracy and completeness.

One shortcoming was the lack of consideration for developing a mobile application at the planning phase. Recipe apps are more suited to a mobile platform by nature, and this was reflected in user feedback, which was realised much too late into the process. Initially, as there was little experience in working with applications the idea was discarded in favour of a web app, which does not work as well on mobile. In hindsight, more in depth planning and research could have taken place at the start of the project, which would have given the development a clearer goal and an improved schedule. Nevertheless, it is believed that the project was overall well conceived, and good project management and execution skills were demonstrated.

# Chapter 9

# Legal, Social, and Ethical Considerations

No formal ethical consent process was required for testing, as the project is software development based and testers were acquaintances and had given verbal consent before participation.

The legality surrounding web scraping is a frequently debated topic, and should be clarified here. The web scraper in Munchmice:

- Can only access publicly available content;
- Does not crawl through websites;
- Tracks sources;
- And scrapes recipes for a particular user's personal use only.

These aspects reduce the likelihood of legal dispute over the action of web scraping. Regarding copyright laws in particular, there are two main points to note: the purpose of Munchmice is to benefit the public without commercial gains, and that recipes typically fall under factual information rather than fictional work. These two factors in combination means that Munchmice is very likely to be covered by the Fair Use doctrine [61]. Furthermore, the nature of Munchmice means that

users will usually first visit a site before saving it to their collection, hence there exists no significant impact on the potential market for the original work. The fulfilment of the above criteria, as well as the fact that Munchmice is not used for commercial purposes, minimises the legal risk associated with web scraping and copyright claims.

# Chapter 10

# Conclusion

This chapter brings the project to a close by summarising the project and its final outcome. Additionally, we will discuss recommendations for future work on Munchmice, also applicable for tools designed for similar purposes.

## 10.1 Project Outcome

Munchmice is a web development project devised with an aim to assist with student cooking. To achieve this purpose, it proposes the use of two key functionalities: the ability to import online recipes via the use of a scraper, and price estimation for recipes that relied on a central price database. The price estimation function is unique and innovative in particular, as the implementation of an automatic estimation tool sets it apart from existing recipe tools in the field. We faced a number of challenges during development, a major obstacle being the sourcing of ingredient price data; in the end, Trolley [1] was used for extracting a basic set of ingredient prices. However, this was likely the average price of items across multiple stores and ranges, and our pricer in turn was only able to give mid-range estimates that were not influenced by unit of ingredient used.

The methodology used was agile-oriented with a waterfall-like Gantt Chart to map out the general structure of development, and combined with the four iterations of the project, ensured development was completed within schedule; time crunch

towards the deadline was overcome owing to the agile approach allowing for rescheduling of tasks. Despite the fact that we were not able to test the project comprehensively, it has achieved all core requirements and produces accurate results, with the scraper successfully importing 82% of recipes and the pricer producing a true-to-life estimate for recipe ingredient prices. We believe its recipe import and price estimation functions, combined with its intuitiveness and pleasant design, shows a great deal of promise as a viable tool to aid in student cooking, which is not an often explored area despite the multitude of recipe applications available on the market. There are also significant possibilities presented for future development, which we will explore in the following section.

## 10.2 Future Work

This section presents final thoughts on the future direction for a tool like Munchmice, exploring ways its current functionalities can be improved, as well as additional features it could benefit from.

### 10.2.1 Natural Language Processing

Ultimately, Munchmice was a very functional, usability-oriented software development project, rather than a research-based project. Though this does not mean that it was without a variety of interesting technical challenges, its implementation could also benefit from next-generation AI techniques. One obvious aspect is the formatting of ingredient strings in a recipe: Natural Language Processing techniques such as Named Entity Recognition could be applied to improve the accuracy of ingredient identification - an aspect Munchmice's scraper-formatter was not able to perfect.

### 10.2.2 User-Contributed Recipe Pricing

The current price data set being used is not meeting the comprehensive requirements for its intended purpose. This has three implications on the estimated price:

firstly, it is limited to be an average of prices, where a cheaper band of items might be more similar to the grocery shopping habits of students; secondly, the price of an item does not have an unit associated with it (e.g. £3.50 per 100g), which reduces the accuracy of a recipe's cost estimation; lastly, the available ingredients is limited to those provided by Trolley which can result in rarer ingredients not being found in the database.

A proposed solution which addresses all three current issues is to source data from users. The current data set can be used as a starting point; the application could encourage users to input ingredient prices from their last shopping trip, which can then be used in future calculations for all users. With a large enough user base, ingredient pricing can be vastly comprehensive. Users could tag their cost estimations with a price band when submitting a price, indicating whether the item they bought is considered cheaper or more high-end. This feature could be integrated into either the shopping list or the recipe details page, where users can select ingredients in their recipes to input prices. There is no particular extrinsic motivation for this action, however, it is believed that as Munchmice is a free software, through the collaborative effort of its user base, everyone is able to benefit from individual contributions.

### 10.2.3 Unimplemented Requirements

The requirement analysis included a variety of "Could have" requirements. These features can enhance the usability of the system, while also expanding the application's functionality beyond its current proof of concept stage. Smaller changes and additions involve the ability to create multiple cookbooks and shopping lists, filtering recipes by diet or cuisine, ingredient substitution recommendations, as well as style and theme customisation to allow users to add a personal touch to their recipes. A potential addition that would require more time and effort is the implementation of an AI-based recommendation system, where new recipes are suggested to users based on their current recipe collection.

### 10.2.4 Mobile Application

Lastly, as we have discussed earlier in the report, a mobile application would be the logical next step after the successful launch of the web application. In the long run, most users would be accessing the tool from their mobile phones, as it is both portable and convenient. A mock-up design for the mobile dashboard and recipe details page is shown below, in Figure 10.1.


Figure 10.1: Proposed mobile app design for Munchmice.

Additionally, a mobile application can take advantage of device-specific features such as push notifications for the shopping list, or recommendations for new recipes. Lastly, it can improve the user experience by allowing smoother and faster navigation, as well as an option to download recipes to grant offline access to saved recipes.

# Bibliography


[1] Trolley. *Grocery Price Index*. URL: https://www.trolley.co.uk/grocery-price-index/. (accessed: 12.04.2023).

[2] Heyl. *Can cooking my own food help my mental health?* URL: https://www.verywellmind.com/mental-health-benefits-of-cooking-your-own-food-5248624. (accessed: 09.04.2023).

[3] Susanna D.H. Mills et al. "Perceptions of 'home cooking': A qualitative analysis from the United Kingdom and United States". In: *Nutrients* 12.1 (2020), p. 198. DOI: 10.3390/nu12010198.

[4] Susanna Mills et al. "Frequency of eating home cooked meals and potential benefits for diet and health: Cross-sectional analysis of a population-based cohort study". In: *International Journal of Behavioral Nutrition and Physical Activity* 14.1 (2017). DOI: 10.1186/s12966-017-0567-y.

[5] Lu Ann Soliah, Janelle Marshall Walter, and Sheila Ann Jones. "Benefits and barriers to healthful eating". In: *American Journal of Lifestyle Medicine* 6.2 (2011), pp. 152–158. DOI: 10.1177/1559827611426394.

[6] E. F. Sprake et al. "Dietary patterns of university students in the UK: A cross-sectional study". In: *Nutrition Journal* 17.1 (2018). DOI: 10.1186/s12937-018-0398-y.

[7] GOV.UK. *Family Food 2018/19*. URL: https://www.gov.uk/government/statistics/family-food-201819/family-food-201819. (accessed: 20.04.2023).

[8] BBC Good Food. *Welcome to good food*. URL: https://www.bbcgoodfood.com/. (accessed: 11.04.2023).

[9] Yummly. *Personalized recipe recommendations and search.* URL: https://www.yummly.co.uk. (accessed: 12.10.2022).

[10] Allrecipes Editorial Team. *Recipes, how-tos, videos and more.* URL: https://www.allrecipes.com/. (accessed: 22.04.2023).

[11] Recipe Keeper. *The easiest way to organize your recipes.* URL: https://recipekeeperonline.com/. (accessed: 12.10.2022).

[12] Saffron. *Saffron Cooking App.* URL: https://www.mysaffronapp.com/. (accessed: 12.10.2022).

[13] Paprika. *Paprika Recipe Manager.* URL: https://www.paprikaapp.com/. (accessed: 12.10.2022).

[14] Recipe Cost Calculator. *Premier recipe costing for food businesses of all sizes since 2013.* URL: https://recipecostcalculator.net/. (accessed: 12.10.2022).

[15] Craig Smith. *10 interesting Yummly Facts and Statistics.* URL: https://expandedramblings.com/index.php/yummly-facts-statistics/. (accessed: 21.04.2023).

[16] Agile Business Consortium. *Chapter 10: Moscow Prioritisation.* URL: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html. (accessed: 22.04.2023).

[17] Dean Leffingwell. *Nonfunctional requirements.* URL: https://scaledagileframework.com/nonfunctional-requirements/. (accessed: 22.04.2023).

[18] Ben Shneiderman et al. *Designing the user interface: Strategies for effective human-computer interaction.* Pearson, 2018.

[19] React. URL: https://react.dev/. (accessed: 22.04.2023).

[20] Flask. *Flask.* URL: https://flask.palletsprojects.com/. (accessed: 11.04.2023).

[21] Django Project. *Django.* URL: https://www.djangoproject.com/. (accessed: 22.04.2023).

[22] GeeksforGeeks. *Python — Introduction to Web development using Flask*. URL: https://www.geeksforgeeks.org/python-introduction-to-web-development-using-flask/. (accessed: 11.04.2023).

[23] Figma. *The Collaborative Interface Design Tool*. URL: https://www.figma.com/. (accessed: 25.04.2023).

[24] Asana. *Manage your team's work, projects, & tasks online • asana*. URL: https://asana.com/. (accessed: 30.04.2023).

[25] SQLAlchemy. *ORM Mapped Class Configuration¶ - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/orm/mapper_config.html. (accessed: 06.11.2022).

[26] Alex McKean. *Designing a Relational Database for a Cookbook*. URL: https://dev.to/amckean12/designing-a-relational-database-for-a-cookbook-4nj6. (accessed: 11.04.2023).

[27] Ben Awad. *Scraping Recipe Websites*. URL: https://www.benawad.com/scraping-recipe-websites/. (accessed: 19.10.2022).

[28] Google. *Recipe (Recipe, HowTo, ItemList) structured data*. URL: https://developers.google.com/search/docs/appearance/structured-data/recipe. (accessed: 20.11.2022).

[29] Wikipedia. *JSON-LD*. URL: https://en.wikipedia.org/wiki/JSON-LD. (accessed: 20.11.2022).

[30] Wikipedia. *Microdata*. URL: https://en.wikipedia.org/wiki/Microdata_(HTML). (accessed: 20.11.2022).

[31] Schema.org. *Recipe*. URL: https://schema.org/Recipe. (accessed: 19.10.2022).

[32] Zhengxiang Shi et al. *Attention-based ingredient phrase parser*. Tech. rep. Gower St, London, United Kingdom: University College London, Oct. 2022. URL: https://arxiv.org/abs/2210.02535.

[33] Nirav Diwan, Devansh Batra, and Ganesh Bagler. "A Named Entity Based Approach to Model Recipes". In: *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. 2020, pp. 88–93. DOI: 10.1109/ICDEW49219.2020.000-2.

[34] Jed Simson. *Parsing ingredients from online recipe articles*. URL: https://www.jedsimson.co.nz/blog/2020/06/04/parsing-ingredients-from-online-recipe-articles. (accessed: 11.04.2023).

[35] MichielMag. *parse-ingredients*. URL: https://github.com/MichielMag/parse-ingredients. (accessed: 11.04.2023).

[36] Kenneth Reitz. *Requests*. URL: https://pypi.org/project/requests/. (accessed: 19.10.2022).

[37] Trolley. *Compare supermarket prices*. URL: https://www.trolley.co.uk/. (accessed: 11.04.2023).

[38] Trolley. *Save Trolley*. URL: https://www.trolley.co.uk/save-trolley/. (accessed: 12.04.2023).

[39] Food Department for Environment and Rural Affairs. *Commodity prices*. July 2019. URL: https://www.gov.uk/government/collections/commodity-prices. (accessed: 11.04.2023).

[40] Food Department for Environment and Rural Affairs. *Wholesale fruit and vegetable prices*. Apr. 2023. URL: https://www.gov.uk/government/statistical-data-sets/wholesale-fruit-and-vegetable-prices-weekly-average. (accessed: 11.04.2023).

[41] Microsoft. *Microsoft OneNote Digital Note taking app: Microsoft 365*. URL: https://www.microsoft.com/en-us/microsoft-365/onenote/digital-note-taking-app. (accessed: 11.04.2023).

[42] Apache Friends RSS. *Apache Friends*. URL: https://www.apachefriends.org/. (accessed: 23.04.2023).

[43] React. *Built-in React Hooks*. URL: https://react.dev/reference/react. (accessed: 11.04.2023).

[44] React Router. *Home V6.10.0 — React Router*. URL: https://reactrouter.com/en/main. (accessed: 11.04.2023).

[45] Sass. *CSS with superpowers*. URL: https://sass-lang.com/. (accessed: 25.10.2022).

[46] Parwiz Forogh. *Python Flask & REACT.JS Full Stack (Python Back-end React Front-end)*. URL: https://www.youtube.com/watch?v=msEmUtYqVV0. (accessed: 02.11.2022).

[47] npm. *Bcrypt*. URL: https://www.npmjs.com/package/bcrypt. (accessed: 30.04.2023).

[48] Flask. *Flask-Session*. URL: https://flask-session.readthedocs.io/en/latest/. (accessed: 30.04.2023).

[49] Scrapinghub. *Extruct 0.14.0*. URL: https://pypi.org/project/extruct/. (accessed: 19.10.2022).

[50] Wikipedia. *ISO 8601*. URL: https://en.wikipedia.org/wiki/ISO_8601. (accessed: 11.04.2023).

[51] pandas. *Pandas*. URL: https://pandas.pydata.org/. (accessed: 20.04.2023).

[52] Wikipedia. *Levenshtein distance*. URL: https://en.wikipedia.org/wiki/Levenshtein_distance. (accessed: 20.04.2023).

[53] Flask-Marshmallow. *Flask-Marshmallow*. URL: https://flask-marshmallow.readthedocs.io/en/latest/. (accessed: 11.04.2023).

[54] phpMyAdmin. *PhpMyAdmin*. URL: https://www.phpmyadmin.net/. (accessed: 30.04.2023).

[55] Jest RSS. *Jest*. URL: https://jestjs.io/. (accessed: 28.04.2023).

[56] Postman. *Postman API Platform*. URL: https://www.postman.com/. (accessed: 28.04.2023).

[57] Selenium. *Selenium*. URL: https://www.selenium.dev/. (accessed: 29.04.2023).

[58] WordPress.com. *Build a site, Sell your stuff, start a blog & more*. URL: https://wordpress.com/. (accessed: 29.04.2023).

[59] Tesco. *Tesco - Supermarkets: Online Groceries, Clubcard & Recipes.* URL: https://www.tesco.com/. (accessed: 29.04.2023).

[60] ALDI. *Terms of use.* URL: https://www.aldi.co.uk/terms-and-conditions. (accessed: 11.04.2023).

[61] Lou Trotta. *Web scraping – legal considerations - copyright - UK.* Sept. 2020. URL: https://www.mondaq.com/uk/copyright/984108/web-scraping-legal-considerations. (accessed: 30.04.2023).

# Appendix A

# User Survey Results

The survey was anonymous and received 16 responses. It was shared with friends, acquaintances, and colleagues studying in universities across the UK. Below are the survey questions asked about student diet habits.

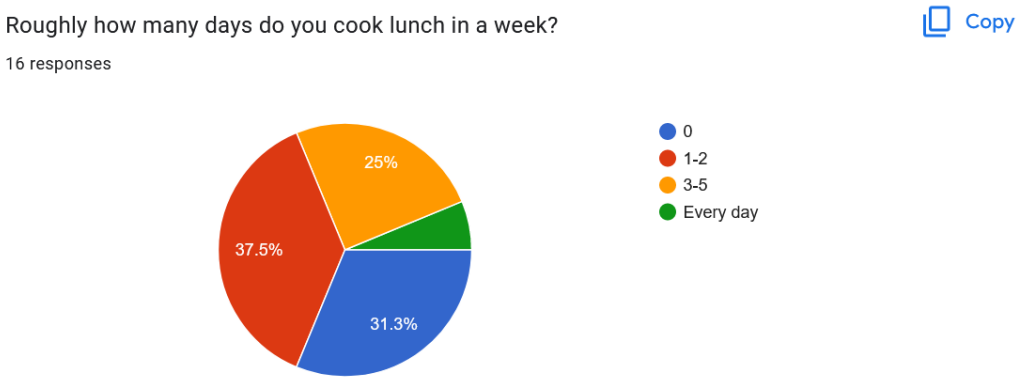


Figure A.1: *Roughly how many days do you cook lunch in a week?*

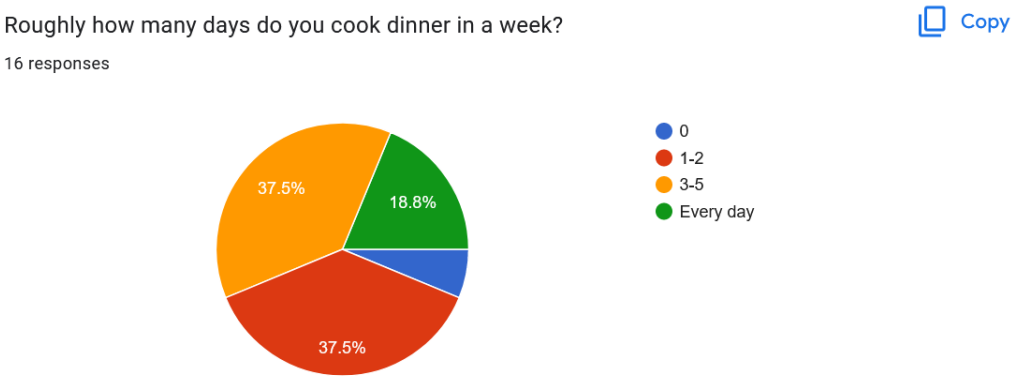


Figure A.2: *Roughly how many days do you cook dinner in a week?*

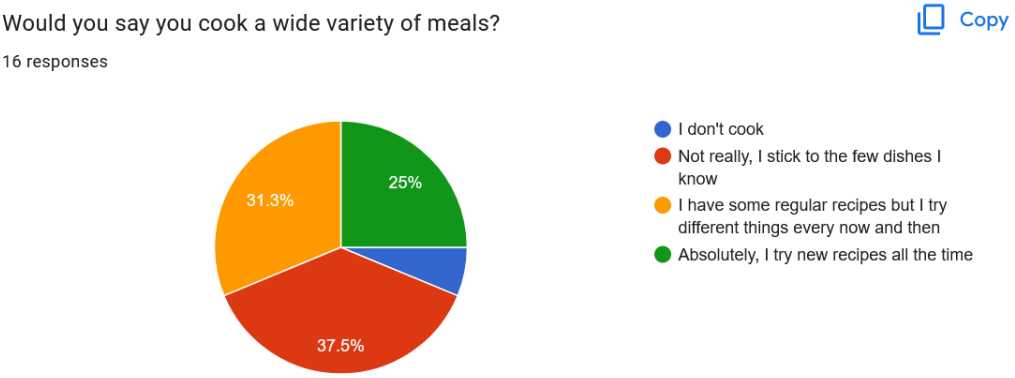


Figure A.3: *Would you say you cook a wide variety of meals?*

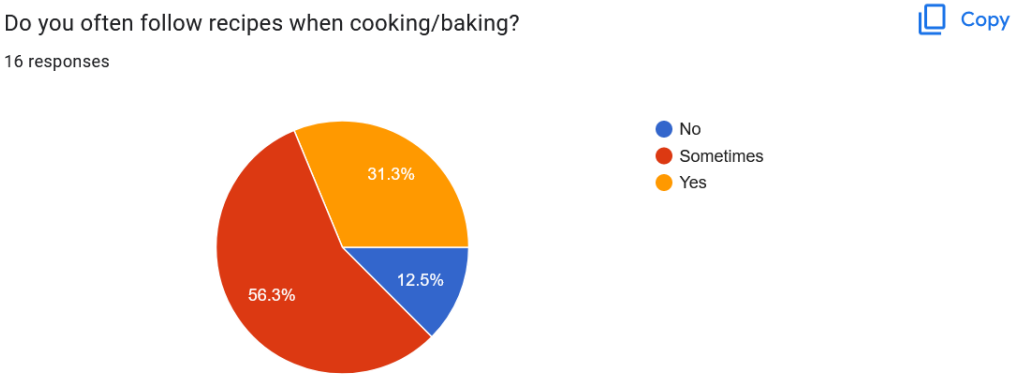


Figure A.4: *Do you often follow recipes when cooking/baking?*


Figure A.5: *Do you know roughly how much you spend on each meal you cook?*

Below are the specific questions asked about Munchmice.

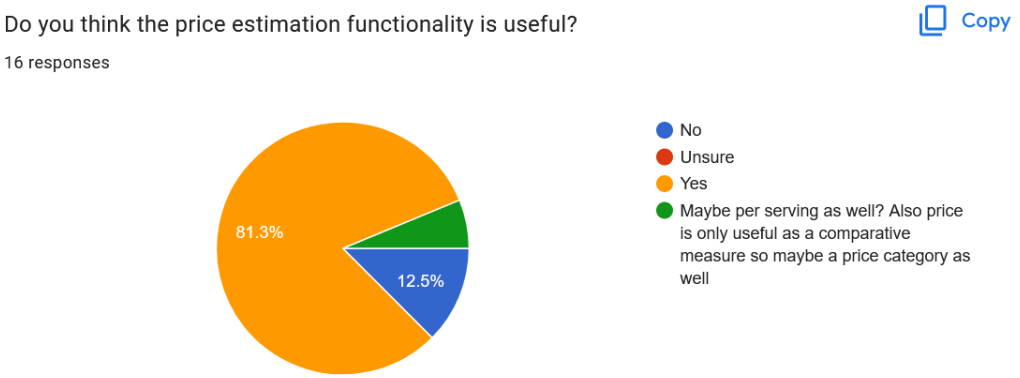


Figure A.6: *Do you think the price estimation functionality is useful?*

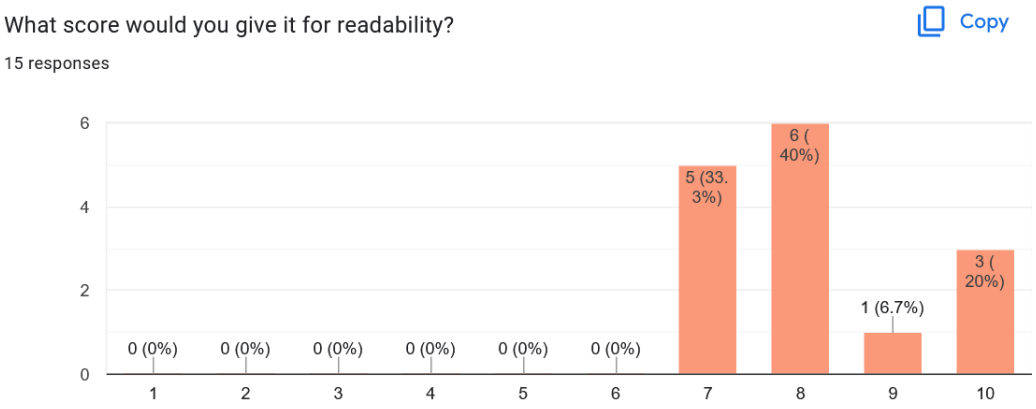


Figure A.7: *What score would you give it for readability?*

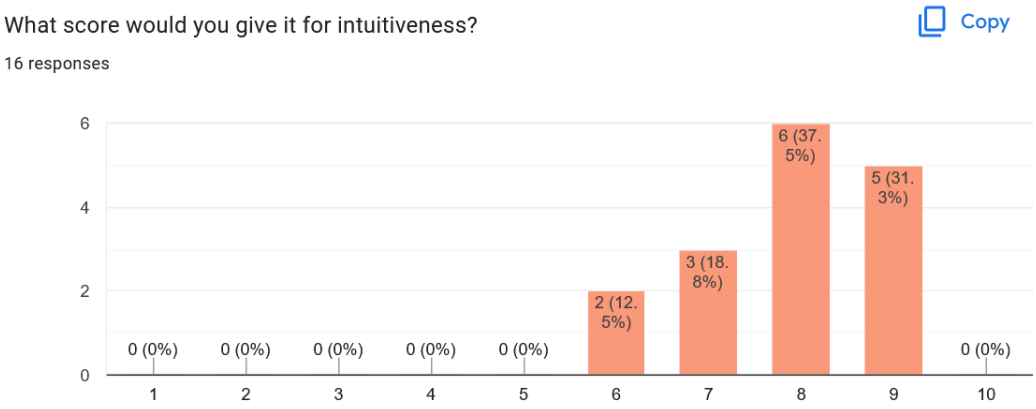


Figure A.8: *What score would you give it for intuitiveness?*

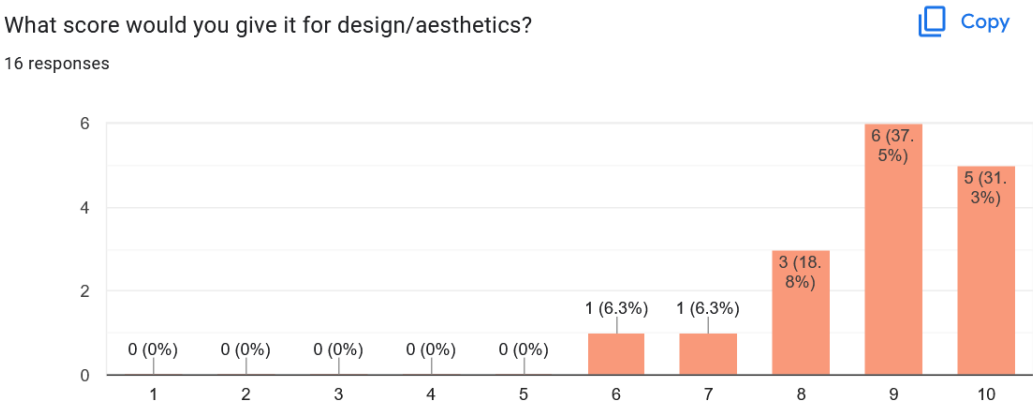


Figure A.9: *What score would you give it for design/aesthetics?*

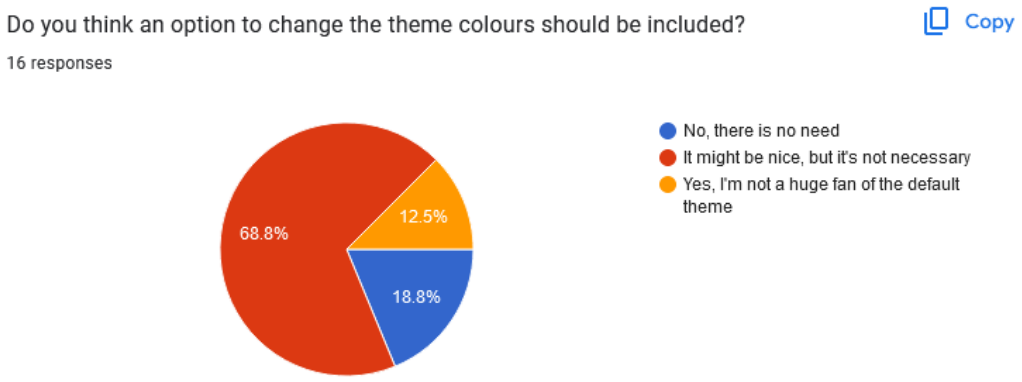


Figure A.10: *Do you think an option to change the theme colours should be included?*