Grocery Price Comparison App: Recipesaver

Group 3 Members: Freddy, Hila, Nikki, Rebecca, Mali, Leima

Introduction

Aims and objectives

Have you ever stood in your kitchen, stomach growling, wallet anxious, wondering which supermarket has the best deals today? That's exactly the problem our team set out to solve with *Recipesaver*.

We've created *Recipesaver,* a Python application that compares grocery prices across multiple supermarkets. This transforms a mundane, everyday activity into a simple, money-saving experience. We aimed to create something people genuinely want to use to improve their daily lives during challenging economic times.

Recipesaver gives you two options: discover where your entire shopping list would be cheapest (e.g., finding out Tesco beats Asda by £3.50 today), or enter ultimate savings mode with a mixed basket showing exactly which items to grab where (e.g., 'Milk's cheaper at Asda, but dash to Tesco for those eggs!').

Whether you're typing in ingredients on the fly or selecting from our database of commonly used ingredients (e.g., eggs, flour, or milk), our user-friendly application keeps things straightforward while our backend technology handles the complex calculations. We've balanced technical innovation with practical accessibility, creating a tool that turns anyone into a savvy shopper.

Contents

Conclusion	7	Project outcome evaluation, technological implementation assessment, user benefit analysis and strategic recommendations for future development.

Background

Prices for groceries can differ between stores, creating a time-consuming challenge for budget-conscious shoppers. RecipeSaver addressed this by automating price comparisons, enabling users to make informed purchasing decisions without manually checking multiple websites. By offering instant access to the cheapest options available, the tool helps users save both time and money.

Project logic

The application's logic revolved around three core processes: user input handling, price aggregation, and cost calculation. Users began by either entering a list of ingredients or selecting from our database of commonly used items, such as those in a pancake recipe (e.g., eggs, flour, sugar, milk). This triggered the backend to construct custom URLs for each supermarket. The app then utilised *ScraperAPI* to collect real-time prices from supermarket websites (Tesco, Morrisons and Aldi). These prices were stored in a structured dictionary format for efficient comparisons and quick retrieval.

The app supported two cost calculation modes: In the multi-store mode, it iterated through each item, identified the cheapest supermarket for each item, and produced a breakdown (e.g., "Buy bananas at Asda (£0.50)"). Whereas, in the single-store mode, it calculated the total cost of buying all the items from a single supermarket and suggested the most affordable choice overall. Savings were calculated by subtracting the user's total spend from the cost of the most expensive alternative, providing a clear metric such as "£1.50 saved".

Specifications and Design

Non-Technical Requirements

Users' experience, accessibility and general expectations:

1. **Ease of use:** The app is intuitive and beginner-friendly.
2. **A Cost-saving Focus:** Users want to spend less on shopping, the app shows how much they saved.
3. **Recipe Flexibility:** Users can enter ingredients manually or choose from a built-in list.
4. **Availability Assumptions:** The app assumes at least one shop stocks the item.

Technical Requirements

Practical specifications and considerations:

1. **Real-time Price Data:** Uses *ScraperAPI* or similar tools to find and compare real-time supermarket prices.

2. **Input Handling:** Accepts custom ingredient lists or selections from a predefined list.
3. **Price Aggregation:** Aggregates prices for all items across *Tesco, Morrisons, Aldi*,
4. **Cost Calculation:**
    a. The total cost if all items are bought from one supermarket.
    b. Optimised cost if items are split across multiple supermarkets.

Data Flow Example

User input (e.g. *'Milk, bread') → Backend search URLs → Scraper fetches prices → System calculates total cost → User sees the most affordable shop to find chosen ingredients.*

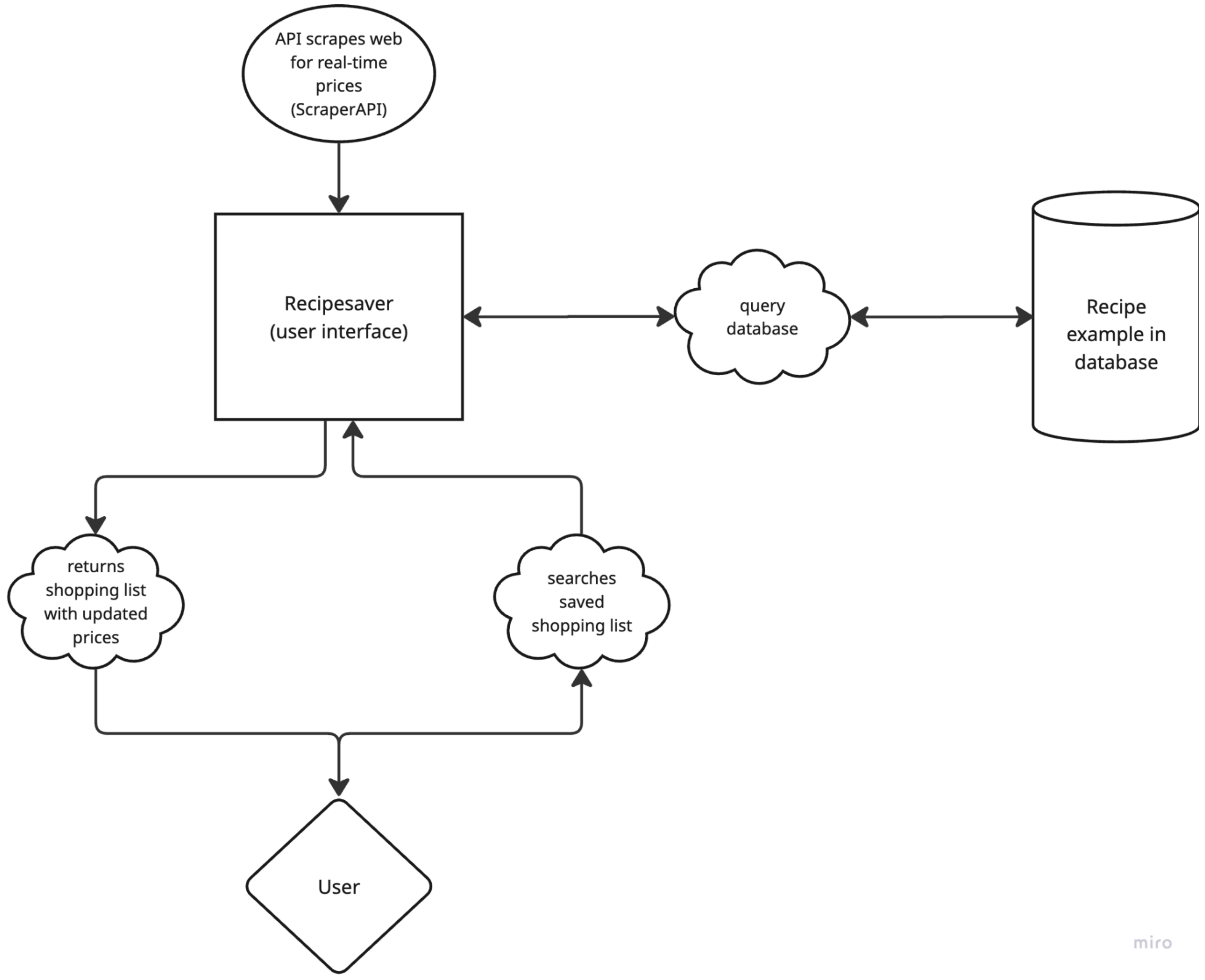
Design Goals

1. **Modular structure:** Split between front end, back end and scraping/API.
2. **Scalable:** Easy to add more supermarkets in future.
3. **Maintainable:** Use clear data flow and reusable functions.

Web Scraper/API Layer

1. *ScraperAPI* or another tool scrapes supermarket websites.
2. Data is formatted into a dictionary:
   {“milk”: {“Tesco”: 1.10, “Asda”: 1.20}}

Architecture Overview


API scrapes web for real-time prices (ScraperAPI)
Recipesaver (user interface)
query database
Recipe example in database
returns shopping list with updated prices
searches saved shopping list
User
miro

Implementation and Execution

Development Approach

For our project, we used Trello as our project management tool to ensure we adopted a collaborative approach. We tracked tasks, offered each other support and monitored progress transparently. This approach allowed us to maintain steady progress while adapting to challenges, ensuring all team members were aligned with project goals and timelines. We used GitHub for our code documentation and repository management. We worked on individual branches from the *main*, using *git fetch* before merging or pushing to stay updated without auto-merging. We reviewed each other's branches to ensure we are all supported.

Collaborative Development Process

Our development process followed a structured yet flexible approach that emphasised:

1. Each team member had well-defined responsibilities based on their strengths and interests.
2. Daily updates through messaging and weekly video meetings.
3. Frequent merging of code to avoid conflicts and ensure everyone worked with the current versions, with an appointed branch master.
4. Built features in small, testable portions rather than large implementations.
5. Created comprehensive documentation alongside code to maintain clarity.

Team Member	**Role**	**Tools + Libraries**	**Responsibilities**	**Deadlines**
Hila	Strategic vision & context	SQL + Python3 - database creation.	Project aims, SWOT analysis,	26/04/2025
Malika	Technical architecture	ScraperAPI + Python3 - API testing.	System design, technical/non-technical requirements	30/04/2024
Freddy	Technical implementation	ScraperAPI + Flask + Python3 + HTML - API implementation and hosting, frontend, backend Python code.	Tool selection, agile methodology implementation	29/04/2025
Nikki	Project management	SQL+ Python3 - backend research for database, backend Python code.	Development approach, team coordination, and challenge analysis	22/04/2025
Leima + Mali	Quality assurance	ScraperAPI + Python3 - API testing.	Testing strategy, evaluation of system limitations	30/04/2025
Rebecca	Project outcomes	Github + Python 3 - code documentation and repository management.	Results documentation/conclusion, future recommendations	02/05/2025

Technical Challenges
The project faced several technical challenges of varying priority levels. High-impact issues included supermarket website data scraping, which required implementing robust scraping algorithms with fallback options, and explored available API alternatives. Alongside this, product name matching was another high-impact issue, and we mitigated this by creating a standardised product dictionary. Medium-impact challenges included maintaining data consistency through scheduled data rules, validation rules and error flagging for anomalous price fluctuations. This optimised user experience through a minimalist UI strategy that prioritised critical user journeys, adding data security by minimising data collection in compliance with privacy standards.

Implementation and Execution
We used incremental programming to help avoid as many bugs as possible. We used branches for every small feature, and merged the branches often to allow everyone to work on up-to-date code. We set up branch protection on GitHub to avoid accidental pushes to *main* and ensured that another person had to accept merge pull requests. This ensured regular checks of our code and helped us to avoid merge conflicts.

We employed agile methodology strategies to help streamline our development, allowing us to maximise what we achieved within the deadline. This means we split our features into smaller tasks, allowing multiple members to work collaboratively on one feature, and enabling everyone to assist on sections they may have previously not worked on. This flexibility ensured all members contributed around other commitments such as work, revising for our final exam, children or appointments. We used Trello as a Kanban board, and during our catch-ups, we reviewed our tasks required for the day and worked accordingly.

Testing Strategy
We used a combination of manual and automated testing to validate the accuracy and functionality of the app's primary features. Throughout development, we recorded bug reports and documented inconsistencies. The report included steps to reproduce *expected* vs *actual* output.

Functional Testing
We performed tests on the following functions of the app:
1. **Ingredient input:** ensured users could submit single or multiple items without errors
2. **API integration and price matching:** Tested if ingredient names, e.g, *'carrot'*, was correctly matched with supermarket data, and that if the ingredient *'carrot'* was not available, a suitable message was displayed for unlisted items.
3. **Cost calculations:**
    - The total cost per supermarket was accurately calculated
    - The app could correctly identify which supermarket is the cheapest overall
    - A 'multi-supermarket' strategy was also shown correctly when users wanted the cheapest option of an item across multiple stores
    - Total savings display: the savings value (the difference between the most expensive and cheapest option) was calculated and displayed correctly

User Testing
We conducted user testing within our team. For each user, this involved:
- Entering your shopping list,
- Select from a predefined list,
- Reviewing the app's recommendations for the cheapest priced item (checking for clarity and how easy it is to use).

The feedback from this testing helped us identify areas for improvement, such as:
- Error handling for misspelt or unrecognised ingredients
- Adding more descriptive loading/error messages

- Enhancing user instructions for users

System Limitations & Ideas for Future Work

1. **Real-time API reliability:** real-time prices can be unavailable or inconsistent, as the system relies on third-party supermarket websites and API
2. **Item matching:** there can be variation between users' choice of product naming, for instance, *'banana'* vs *'bananas'*. This may lead to unmatched items unless improved
3. **No multi-currency or regional variation:** our prices are shown based on a fixed region.

Future scope:
To improve item matching, we could implement simple solutions such as converting all input to lowercase, trimming extra spaces and creating a small list of common variations (e.g mapping "bananas" to "banana"). In the future, the app could also expand to support different currencies and regions, and allow users to save preferences.

Conclusion

We identified a need for a user-friendly grocery comparison application and created a practical and scalable solution for users to identify the most cost-effective option for purchasing everyday grocery items specifically from Tesco, Morrisons, and Aldi.

We used the requests library within Python to access real-time price data through an external web-scraping API and stored it in a structured SQL database for efficient retrieval. We have also incorporated a user-friendly, front-end interface to create a simple and efficient user experience. By using this integrated approach, we have been able to ensure data accuracy, efficient querying and an ability to adapt dynamically as both retail prices and product availability fluctuate over time.

This tool is beneficial for users with a scalable projection. For future updates, there is scope to include user reviews, product ratings and location mapping for proximity access.

This project demonstrates our understanding of the real-world relevance of developing data-driven applications that provide user value, whilst communicating our budding expertise in Python, SQL and APIs.