Contents

Analysis

Introduction

I aim to create a website that will make it easier for people to decide what to cook based on the ingredients they have available. Given how busy people are it can often be difficult to decide what to cook, which can lead to unhealthy eating habits. The intention of the website is to simplify the process of choosing a meal to cook without requiring users to purchase additional ingredients by recommending recipes that contain the ingredients that users have available. The website will also provide an API to enable developers to use the service in their own applications. The project will take the form of a website that accesses a database of standardised recipes assembled using a web scraper. This will involve the creation of an ETL – Extraction, Transformation and Loading – pipeline to collect, process and display data about recipes to users. This will require a variety of data, collection, processing, storage, and information retrieval techniques.

Clients

Target clients

The target clients for this project are people who struggle to decide what to cook with the ingredients they have. The project is not specifically targeted towards people who want to learn to how cook, but it could be beneficial to that demographic by encouraging people to try recipes they otherwise would not have.

Client survey

To gather information, I have sent a survey to potential clients amongst my friends and family.

Questions

1. How do you normally go about deciding what to cook?
2. How often are you unsure of what food to cook?
3. Where do you look for recipes?
4. What is your experience of using recipe websites?
5. How would a website that provided recipe recommendations based on a given list of ingredients make cooking easier?

Responses

- **Ben Davis**
    - Q1: What ingredients need using up, what do people want to eat, what haven't we eaten recently, inspiration from recipe books or magazines
    - Q2: Most of the time
    - Q3: Internet search -> saved recipes from internet -> recipe books -> magazines
    - Q4: Generally ok, but done typically via a search for a specific term or ingredients set. Some recipe sites tend to be more reliable, at least by reputation, but normally recipe choice is driven by experience
    - Q5: That would help a great deal. It's a situation that I'm faced with almost daily.
- **Iain Walker**
    - Q1: I pick from a small set of common recipes, or spend half of Saturday pouring over recipe books and websites.
    - Q2: Every night except Thursday and Friday, as Thursday is pasta night and Friday is home made chips.
    - Q3: Recipe books such as River Cottage, or the internet for recipes involving X ingredient that we might have in.

    - Q4: A lot of the time they're bloated with ads and unnecessary information.
    - Q5: It would take the uncertainty out of knowing if I can make anything with what we have in the fridge
- **George Mack**
    - Q1: Whatever's available - parents do the shopping, so I rarely plan anything big. I'll also be somewhat influenced by seasonality, as we grow veg.
    - Q2: Very often - well over half of my cooking is improvised.
    - Q3: Recipes from parents, Common sense, Delia Smith's Complete Cookery Course, Online
    - Q4: I've never really used them for regular cooking; I only resort to the interweb if I need something unusual, like a Christmas recipe.
    - Q5: It would be far more relevant to everyday cooking than most recipe websites. It might also enrich the cooking (and eating) experience by suggesting new recipes. Tailoring recipes based on what's available could reduce food waste - I wonder if there could be a feature to prioritise ingredients which need using more urgently?
- **Matthew**
    - Q1: Whatever comes to mind at the time
    - Q2: Rarely, I tend to know roughly what I want to make once I have seen what is available.
    - Q3: Online
    - Q4: Relatively good
    - Q5: It would make it a lot easier as I tend to be limited on resources because I play lots of strategy games that force me to make do with limited resources. So I would like a tool that assists with utilising the foods I have to make something I would like.

Evaluation

- Question 1:
    - The responses to this question indicate that clients don't often plan meals in advance, and instead often cook with what is available. This supports the idea that the project will be beneficial to clients
    - Additionally, that people make decisions based on what is readily available
    - But also, that people do seek out variation and new ideas
- Question 2:
    - Responses to this suggest that people are often unsure of what to cook and improvise
- Question 3:
    - Whilst this does suggest that online sites are sometimes used, the responses do also point to a pattern of using recipe books and magazines as well.
- Question 4:
    - The somewhat negative responses here could explain the above pattern, suggesting that people dislike using current recipe websites
- Question 5:
    - The positives responses here reaffirms that the project will be beneficial to clients
    - Additionally, the mention of prioritisation of ingredients is something to explore with regards to ordering the response to a recipe search – and something I will need to take in to account when developing the algorithm to search the database.

Objectives

0. The project must allow users to find recipes based on a series of inputted requirements
1. The project must have a web interface(website)
   - 1.1. The website must have a navigation bar at the top of screen that provides access to all key pages
   - 1.2. The website must have an information page
     - 1.2.1. This page should explain how to use the different functions of the website
   - 1.3. The website must have a home page
     - 1.3.1. On this page there must be a series of fields for the user to enter the information that is used to find recipes
       - 1.3.1.1. This information could include:
         - 1.3.1.1.1. The ingredients to be used
         - 1.3.1.1.2. The maximum time taken to cook the meal
         - 1.3.1.1.3. The 'type' of meal to be cooked, i.e., dinner, breakfast, etc
       - 1.3.1.2. Only the ingredients field should be mandatory
         - 1.3.1.2.1. If no ingredients are provided, the website should reject the search and inform the user that information must be provided
       - 1.3.1.3. If any input is invalid then the search should be rejected with an appropriate message
     - 1.3.2. This home page must provide a button to begin the search for recipes once information is provided
   - 1.4. The website should have a page to display the results of the search
     - 1.4.1. The results must be generated through the search algorithm described in 2.
     - 1.4.2. The results should be displayed in a standardised format
       - 1.4.2.1. Each result should display:
         - 1.4.2.1.1. The name of the recipe
         - 1.4.2.1.2. The time to cook
         - 1.4.2.1.3. The ingredients the recipe requires
           - 1.4.2.1.3.1. The ingredients that are part of the user's search should be distinguished from the others
         - 1.4.2.1.4. A button to 'save' a recipe if a profile is logged in
           - 1.4.2.1.4.1. The recipe should then be added to a stored list associated with the profile in the database
           - 1.4.2.1.4.2. If not logged in, the button should redirect the user to a login page
       - 1.4.2.2. Clicking on a result should open the page the recipe is from
     - 1.4.3. The results should be sortable by:
       - 1.4.3.1. Relevance
         - 1.4.3.1.1. The relevance of a recipe is determined by how well it fits the requirements given by the user
       - 1.4.3.2. Time to cook
     - 1.4.4. The user should be able to limit the number of results returned

1.5. The website must allow a user to have a profile
1.5.1. The website must provide a profile page
1.5.1.1. If a profile is not logged in, the user should be prompted to either:
1.5.1.1.1. Create an account
1.5.1.1.1.1. This should be done by setting a username and password
1.5.1.1.1.1.1. Text entered in the password field should be obscured and asked for twice to confirm it has been typed correctly
1.5.1.1.1.2. Once a profile has been created, the username and hash of the password should be stored in the database, as well as a unique API key
1.5.1.1.2. Or log in
1.5.1.1.2.1. This user should be prompted to enter a username and password
1.5.1.1.2.2. The relevant database entry for the username should be found and the hash of the given password checked against the stored hash
1.5.1.1.2.3. If a relevant entry is found and the hashes match, then the profile should be logged in and the user redirected to the profile page
1.5.1.1.2.4. Otherwise, the data should be rejected, and the user informed that the login failed
1.5.1.2. If the user is logged in, the profile page should display:
1.5.1.2.1. The name of the user
1.5.1.2.2. Their 'saved' recipes
1.5.1.2.2.1. These should be displayed as a list like the results page
1.5.1.2.3. Fields to change their username and password
1.5.1.2.3.1. If a change is made the database should be updated and the user logged out
1.5.1.2.4. A button to log out
1.6. The project must have an API
1.6.1. The API should have documentation detailing every function
1.6.1.1. This documentation should be on a page of the website
1.6.2. Requests should use URL parameters
1.6.3. Responses should use the JSON format
1.6.4. "Bad" requests should be rejected with an appropriate message
1.6.5. The API should provide the recipe search functionality of the main page
1.6.5.1. The algorithm described in 2. should be used to find recipes
1.6.5.2. The found results should be returned
1.6.5.2.1. The results should be sorted by the user's specified method if one exists
1.6.5.2.2. The number of results should be limited to the user's specified value if one exists
2. The project must have an algorithm to search for recipes

2.1. The algorithm should return results where at least one of the ingredients given in the search is included in the ingredients of the recipe

2.1.1. It should generate a 'relevance' for the result based upon how many of the given ingredients it includes

2.2. If a maximum cooking time is given, then the algorithm should return only results that do not exceed this cooking time

2.3. If a type of meal is given, then the algorithm should only return recipes that are of this type

3. The project must have a database that stores the following objects and attributes:

3.1. Recipe

3.1.1. The ID

3.1.2. The name

3.1.3. The list of ingredients

3.1.4. The time the recipe takes to cook

3.1.5. The 'type' of meal

3.1.6. A link to the original recipe

3.2. Profiles

3.2.1. The ID

3.2.2. The username

3.2.3. The hash of the password

3.2.4. The list of saved recipes

3.3. Ingredients

3.3.1. The ID

3.3.2. The name

3.3.3. The recipes it is used in

4. The data for recipes in the database should be collected using a web scraper

4.1. The web scraper should be able to collect information from multiple cooking websites

4.2. Information from websites should be stored in JSON

4.2.1. The information should follow the structure presented in 3.1.

4.3. Information collected should be transferred to the database

4.4. Any information that the web scraper fails to collect or incorrectly collects could be manually corrected

Research

Frameworks and libraries

Flask

Flask is a Python framework for creating web servers. I have opted to use Flask to create my website. This is partially due to my prior experience with using it, but also due to its lightweight nature, flexibility, and integration with other systems such as SQL databases. This will enable me to focus more on the data processing elements of the project and still create a responsive and easy to use website. I have purchased a book[1] on Flask to help me learn the more complex aspects of the framework.

Selecting a database system

I will need a database system to store the information described in my objectives. As I was aware that many different competing systems existed, I researched which database system would be best suited for my project. I found an informative article[2] that I used to create the below notes.

- SQLite
  - Advantages
    - Serverless
      - Read/write from database directly
      - Simple setup
      - No config needed
    - Lightweight
    - Self-contained
    - Easy to use
    - Stored in a single file
  - Disadvantages
    - Only one process can change the database at any given time
      - Will be a problem if multiple users are modifying the database – i.e.: creating accounts – at the same time
- MySQL
  - Advantages
    - Popular – lots of support
    - Speed
    - Reliability
    - Accessed through separated daemon
  - Disadvantages
    - Doesn't stick to standard SQL
      - I.e.: doesn't have FULL JOIN
      - Could cause problems in writing statements – might have to learn new syntax?

[1] Grinberg, M. (2018). Flask web development: Developing web applications with Python (2nd edition). O'Reilly.

[2] Drake, M. & ostezer. (2014, February 21). SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems

MySQL appears to be the best option for my project, since its speed and reliability makes it very suitable for use in a web application that could potentially need to handle a large number of database queries over a short period of time. Further, SQLite is unsuitable as it is unable to have multiple processes modifying the database at one time. This missing functionality is a necessity for a database used in a web application as multiple users could be modifying the database at one time, with each request handled in a different process.

Interfacing with the database in python - Flask-MySQLdb and Flask-SQLAlchemy

Although I originally intended to use Flask-SQLAlchemy to provide database functionality to my website, I have chosen to use Flask-MySQLdb instead because it provides a lower-level interface with the database that will enable me to write SQL queries myself and have them executed by a database connector. SQLAlchemy uses object-relational mapping, where objects are used in a program to represent aspects of the database. Whilst this does provide a simpler method of interacting with a database, it does not allow me to write SQL queries directly. Additionally, Flask-MySQLDB provides a connection that properly handles memory and creating and closing MySQL connections in the context of a Flask application.

Requests and BeautifulSoup

Requests is a Python library for making HTTP requests, and BeautifulSoup is a library for processing HTML. I have chosen to use these libraries in my web scraper because they provide a simple interface for making requests to websites and subsequently extracting relevant elements from the returned HTML.

NLTK

NLTK is a Python library for performing natural language processing. I have chosen to use this library to perform analysis on the lists of ingredients I extract from recipe pages. This is due to its ability to perform part-of-speech tagging, allowing me to identify sequences of relevant nouns and adjectives[3]. Additionally, it can lemmatize strings, removing the risk of misidentifying an ingredient due to the use of a plural.

Version control and IDE

In order to keep track of the development of my project and backup the code I will need to use some form of version control. I have chosen to use git and the website GitHub to store my code due to my familiarity with them and their integration with IDEs such as PyCharm – the IDE I have chosen to use for my project. The code will be stored in a private repository.

Background research

The Flavour Thesaurus[4]

The Flavour Thesaurus is a book that groups ingredients by flavour type and suggests pairings with other ingredients. No information from the book itself will be used in this project, but as I have regularly referenced it when cooking it is a significant source of inspiration for this project – particularly the idea of a tool to help people explore new recipes using only what they have available.

Bags of words and ingredients

In considering the processing of ingredients, I have a chosen to use a “bag-of-words” model. This approach involves treating text – in this case the lists of ingredients in both recipes and user

[3] Bird, S., Klein, E., & Loper, E. (2009). Natural language processing with Python. O’Reilly Media Inc. https://www.nltk.org/book/

[4] Segnit, N. (2010). The flavour thesaurus: Pairings, recipes and ideas for the creative cook. Bloomsbury.

searches – as a sets of space-separated "tokens". This enables me to perform mathematical operations on these sets, as described below.

Ordering recipes with ranking models

I will need to order the recipes returned to the user. This could be done using the total time, but that is unlikely to be very useful to the user as the order in which recipes would be shown would have no relation to the relevance of the results. Neither would be sorting the results alphabetically – the results will need to be sorted by a relevancy score.

This will involve the implementation of some form of ranking model to calculate the relevancy score for each recipe.

One option to do this is to sort the recipes by the number of matches they contain, in other words the number of ingredients of the recipe that match an ingredient given in the input string. I.e.: for the set of input terms *terms* and the set of recipe ingredients *ingredients*:

$$relevancy = |terms \bigcap ingredients|$$

However, this raises the problem of identifying unique ingredients from the input string. More importantly, a recipe that contains fewer ingredients but fully matches some of the input would incorrectly be given a lower relevancy than a second recipe that contains more ingredients – and more matching ingredients – even if the ingredients that do match in the second recipe are a smaller fraction of its ingredients. This means that results that are technically less relevant could be given a higher relevancy score. For example:

$$terms = \{\text{egg, tomato, pasta, cheese, onion}\}$$
$$ingredients_1 = \{\text{tomato, pasta}\} \rightarrow relevancy_1 = 2$$
$$ingredients_2 = \{\text{egg, cheese, pepper, chicken, pastry, onion}\} \rightarrow relevancy_2 = 3$$

In this case, the user could be intending to create a pasta dish, but instead has been given a pastry dish. Whilst the second recipe technically uses more of the user's requested ingredients, it is unlikely to have been as relevant to their situation as the first recipe. Whilst the second recipe is still somewhat relevant in this example, on a larger scale this issue could cause recipes that are technically more relevant to be ignored.

This led me to consider calculating the matching ingredients as a proportion of the overall set of ingredients. I could calculate the relevancy as the fraction of input terms that match recipe ingredients divided by the number of recipe ingredients.

$$relevancy = \frac{|terms \cap ingredients|}{|ingredients|}$$

However, this would give an equal weighting to all ingredients in calculating the relevancy: the presence of common matching ingredients like "salt" would impact the relevancy of a result as much as any other ingredient.

$$terms = \{\text{salt, egg, rice}\}$$
$$ingredients_1 = \{\text{salt, cucumber}\} \rightarrow relevancy_1 = 0.5$$
$$ingredients_2 = \{\text{egg, salt, rice, milk, raisins, sugar, cinnamon, vanilla}\} \rightarrow relevancy_2 = 0.375$$

This would result in a relevancy score that does not accurately reflect the frequency at which particular ingredients appear in the recipe dataset, and as such could distort the relevancy rankings by giving higher relevancy to recipes that contain very common matching ingredients.

Another approach would to be to use a vector space model[5] to calculate a relevancy with regards to the frequency of terms. This would involve vectorising the input ingredients and every recipe's ingredients, and then computing the angle between the vectors of the input and each recipe to obtain a relevancy score. The smaller the calculated angle, the greater the relevancy. Whilst this would likely give the most reliable and accurate relevancy scores, it is also the most complex of the options I have considered and would likely be the slowest.

In conclusion, I will either implement the second or third of the approaches presented here - depending on which I find to be the fastest and most reliable.

Lemmatisation and pre-processing

Before analysing the data – particularly ingredients - I scrape from webpages I will need to perform pre-processing (cleaning). This is the process of transforming a piece of text into a form that can then be further processed – reducing the risk of incorrectly analysing the text and removing unnecessary data.[6] For my project the pre-processing I will perform on ingredients will be:

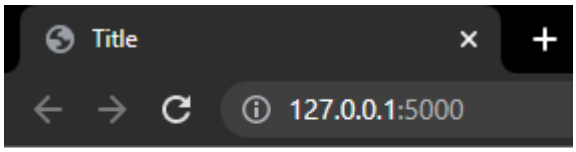
1. Make all characters in the string lowercase
2. Split the string into tokens
3. Lemmatise each token
4. Extract the most relevant tokens
5. Remove common, irrelevant words not removed by the previous step

Lemmatisation is the process of turning a word into its *lemma*, the root or dictionary form of the word.[7] For example, "tomatoes" would become "tomato". This is relevant to my project because I will need to recognise the ingredient string of "5 Potatoes" and the ingredient string of "a potato" as referring to the same ingredient: potato. Without this I will risk misidentifying the ingredients of a recipe and treating the plural and singular forms of nouns as different ingredients.

Prototyping

Website Form Prototype

I created a prototype of flask web server with a form to demonstrate how the web interface could receive data from a user store that data. I began by creating a simple flask website, seen below.



Home page

I then began adding a form to this page. After some investigation of the best methods for adding a form to a flask app, I chose to use the Flask-WTF extension. I created a form object for the recommender form:

---

[5] Vector space model. (2022). In Wikipedia. https://en.wikipedia.org/w/index.php?title=Vector_space_model&oldid=1125140229

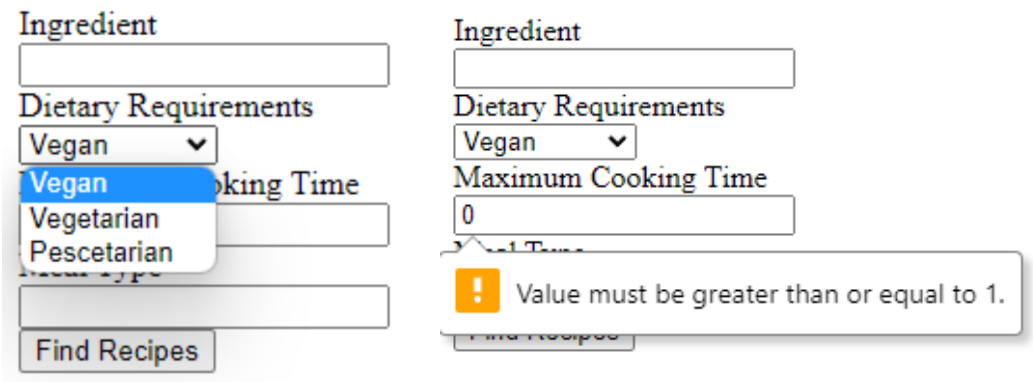
[6] Cheng, R. (2020, June 29). Text Preprocessing With NLTK. Medium. https://towardsdatascience.com/nlp-preprocessing-with-nltk-3c04ee00edc0

[7] Lemma (morphology). (2022). In Wikipedia. https://en.wikipedia.org/w/index.php?title=Lemma_(morphology)&oldid=1127152553

```
from flask_wtf import FlaskForm
from wtforms import StringField, SelectField, IntegerField, SubmitField
from wtforms.validators import NumberRange

class RecommenderForm(FlaskForm):
    ingredients = (StringField("Ingredient"))
    diet = SelectField("Dietary Requirements", choices=[
        ('vgn', 'Vegan'),
        ('veg', 'Vegetarian'),
        ('psc', 'Pescetarian')
    ])
    max_time = IntegerField("Maximum Cooking Time", validators=[NumberRange(min=1)])
    meal_type = StringField("Meal Type")
    submit = SubmitField("Find Recipes")
```

I then added this form to the HTML template of the home page.

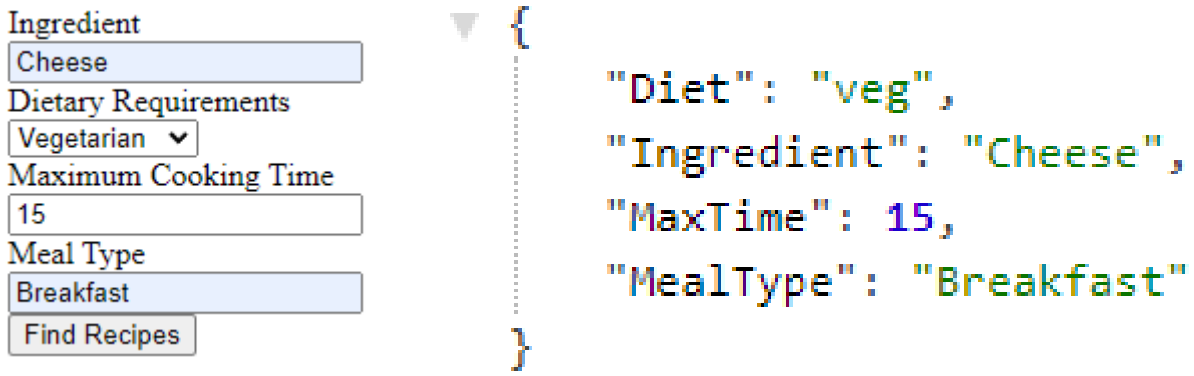
The form provides a dropdown menu for dietary requirements and a minimum cooking time of 1 – but none of the fields are required to be completed. In the final project at least one of the fields will be required to be completed.



To handle the data received from this form, I used the inbuilt Flask-WTF form.validate_on_submit function. The data from the form is then stored in a dictionary that is stored in the flask session. This client-side session dictionary will be used to store a recipe search, but for now it is simply displayed.

```
@app.route('/', methods=["GET", "POST"])
def index():
  form = RecommenderForm()
  if form.validate_on_submit():
    data = {
      "Ingredient": form.ingredients.data,
      "Diet": form.diet.data,
      "MaxTime": form.max_time.data,
      "MealType": form.meal_type.data
    }
    session["recommenderFormData"] = data
    return redirect("/recommend")
  return render_template("index.html", form=form)
```

For example, the following input to the form produces the following dictionary.



Web scraper Prototype

I also created a prototype web scraper for the website allrecipes.com.

I began by scraping the raw html of the page and passing it into a Beautiful Soup object.

```
page = r.get(url).text
soup = BeautifulSoup(page, "html.parser")
```

Then I inspected the HTML of the website to find the id used to identify the title of the recipe page and used it to extract the title text from the HTML.

```
# Find title
title = soup.find(id="article-heading_2-0").text.strip()
```

To extract the ingredients, I used a list comprehension to get the text of every element tagged as an ingredient into a list.

```
ingredients = [i.text.strip() for i in soup.select('span[data-ingredient-name="true"]')]
```

Then I encountered the question of how I would go about extracting the relevant parts of the ingredient text for my database, but I decided to focus on creating a minimum-viable product of a web scraper.

To extract the cooking time of a recipe I used regular expressions, as the time could be written in a variety of formats – i.e.: 1h 2m, 15 minutes, etc. Using a regular expression enables the web scraper to find the relevant part of text – the total time to make a recipe - regardless of its exact format. I began by extracting the relevant text from the HTML.

```
time_text = soup.find(id="recipe-details_1-0").text
```

Then, I wrote a regular expression to find the "total time".

Total Time:

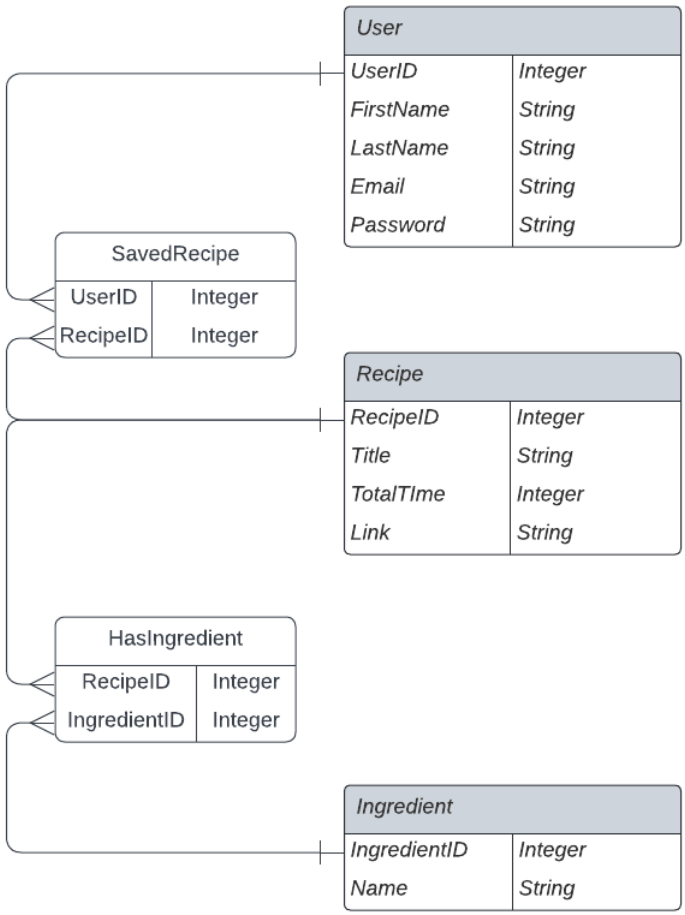
35 mins

```
TOTAL_TIME_REGEX = r"(?<=Total Time:\n)( *\d+ (hr(s)?|min(s)?) *)+"
```

For the above example this expression would match "35 mins".

Modelling

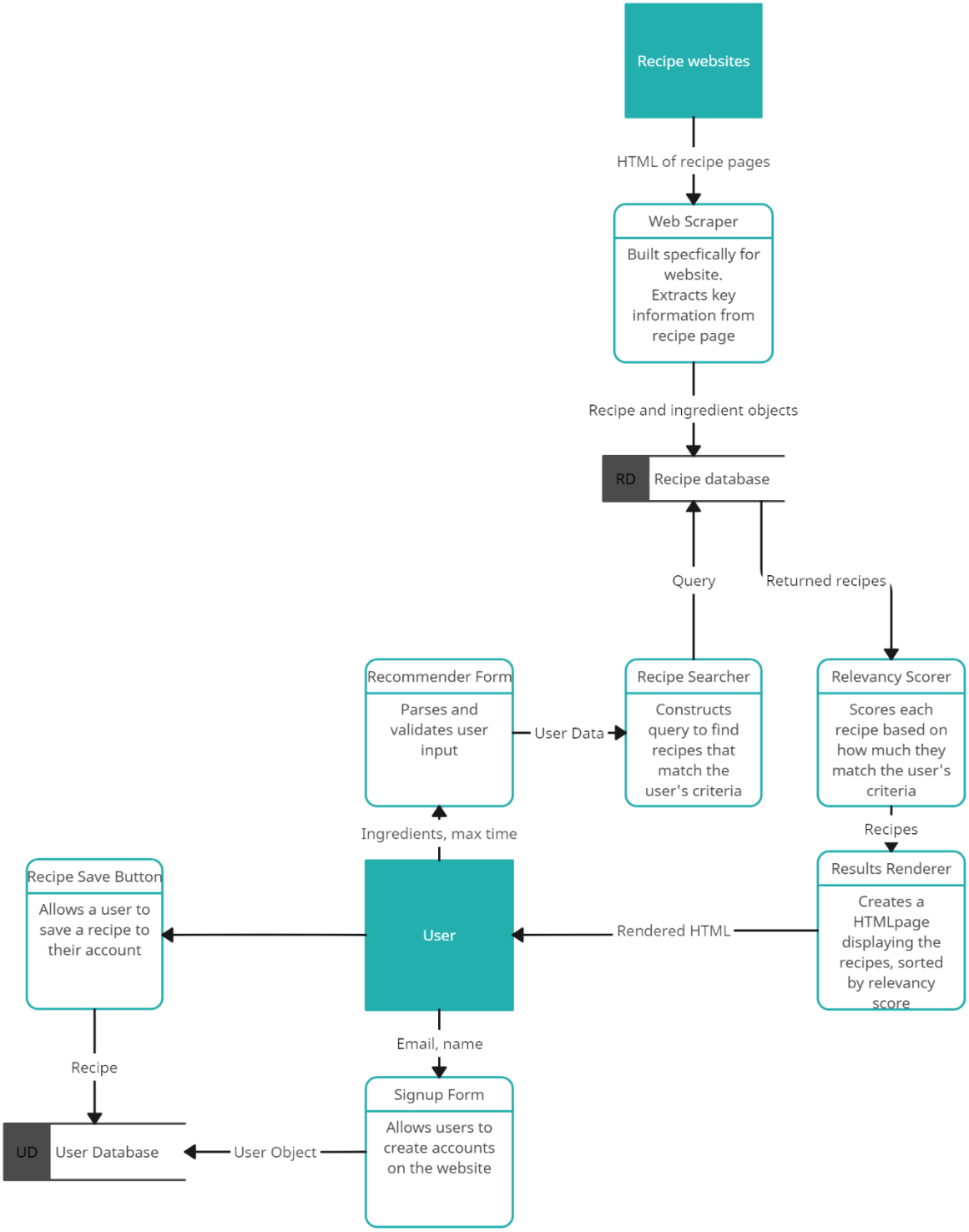
Entity relationship diagram



This diagram shows how I plan to organise my database. The **Ingredient** table will store unique ingredients and their names. The **Recipe** table will store unique recipes. As many different ingredients are used in many different recipes – in other words the relationship is many-to-many – it is necessary to include a linking table. This turns a many-to-many relationship into two one-to-many relationships and ensures that the database is normalised – and can be queried correctly. The **User** table will store data about the users of the website. This will allow users to "save" recipes, making it easier for them to find recipes again. This again requires the use of a linking table, as many recipes can be saved by many users.

Design

Data Flow Diagram

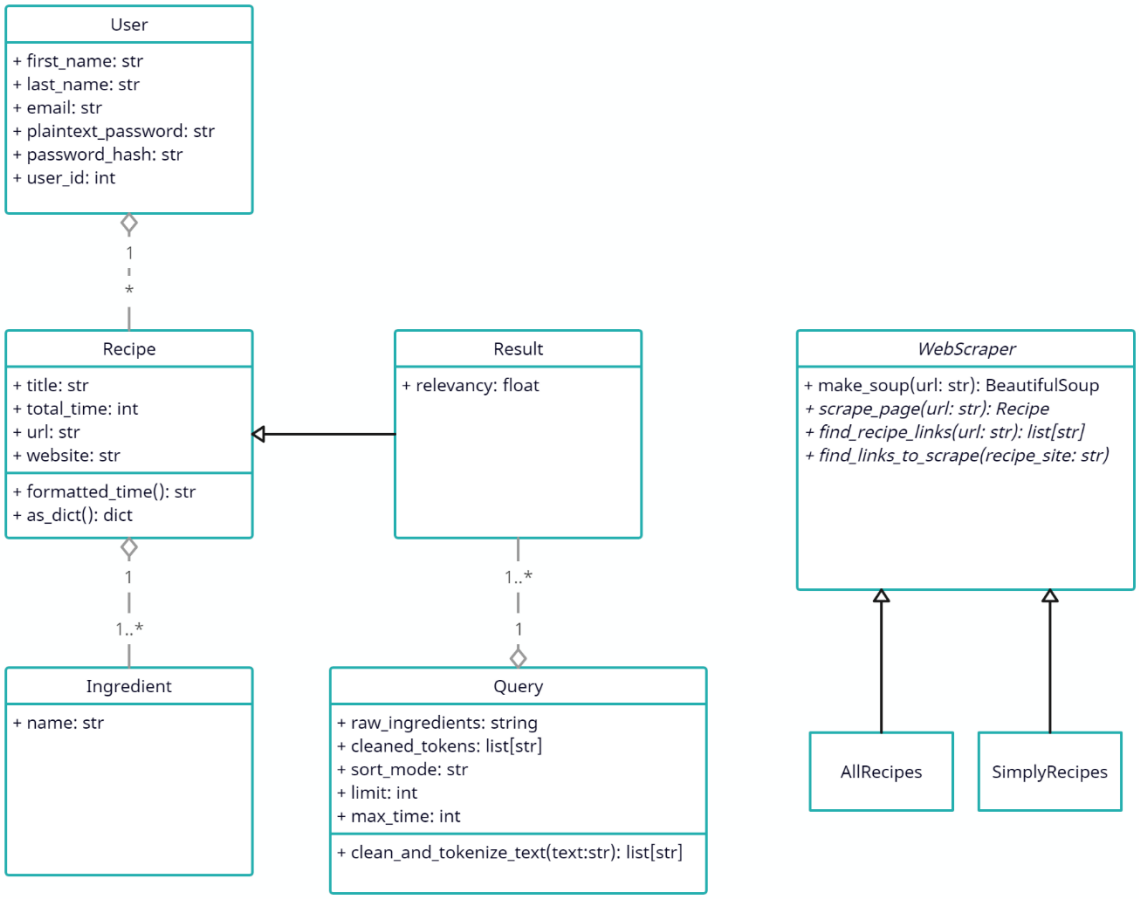


This diagram shows how I plan for data to move around the system. Recipe data will be collected from recipe websites, processed, and stored in a database. The user will be able to search this data, and the results will be ranked by relevancy before being returned. The user will also be able to register an account with the site and use this account to save recipes that can then be retrieved

without using the recommender again. Although they are represented as separate data stores, the UD and RD will be part of a single database – as shown in the ERD.

Data structures

Unified Modeling Language class diagram

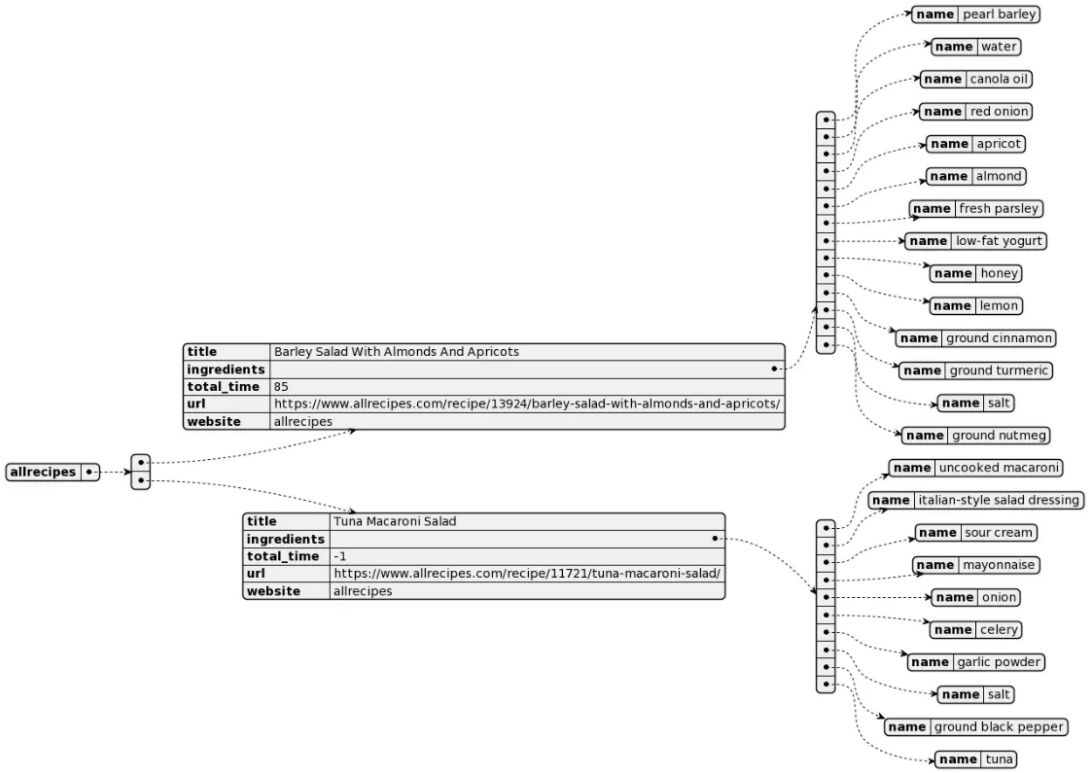


This diagram shows the organization of classes in my project. Ingredient objects are aggregated in Recipe objects to represent the ingredients that are required for a recipe. Recipe objects are aggregated in User objects to represent the recipes that a user has saved. Result objects inherit from Recipe objects and add an attribute used to store the recipe's relevancy to a query. These queries are represented by Query objects - which aggregate the Result objects made when the search is performed based on the Query object's attributes.

All web scrapers are classes that inherit from a WebScraper abstract base class. Each one implements overrides of the abstract methods - shown in italics – specific to their relevant website. The method make_soup is a static, non-abstract method used to create a beautiful soup object from HTML data and as such does not need overriding.

Web scraper results JSON

This example diagram shows how recipe data is stored in JSON files after being scraped and before being inserted into the database. This is a tree data structure and is represented in Python by a dictionary.


allrecipes
title
Barley Salad With Almonds And Apricots
ingredients
total_time
85
url
https://www.allrecipes.com/recipe/13924/barley-salad-with-almonds-and-apricots/
website
allrecipes
name pearl barley
name water
name canola oil
name red onion
name apricot
name almond
name fresh parsley
name low-fat yogurt
name honey
name lemon
name ground cinnamon
name ground turmeric
name salt
name ground nutmeg
title
Tuna Macaroni Salad
ingredients
total_time
-1
url
https://www.allrecipes.com/recipe/11721/tuna-macaroni-salad/
website
allrecipes
name uncooked macaroni
name italian-style salad dressing
name sour cream
name mayonnaise
name onion
name celery
name garlic powder
name salt
name ground black pepper
name tuna

The diagram was created from the below JSON (shown with most ingredients removed and replaced with “…”):


```
{
  "allrecipes":[
    {
      "title":"Barley Salad With Almonds And Apricots",
      "ingredients":[
        {
          "name":"pearl barley",
        }, ...
      ],
      "total_time":85,
      "url":"https://www.allrecipes.com/recipe/13924/barley-salad-with-almonds-and-apricots/",
      "website":"allrecipes"
    },
    {
      "title":"Tuna Macaroni Salad",
      "ingredients":[
        {
          "name":"uncooked macaroni",
        }, ...
      ],
      "total_time":-1,
      "url":"https://www.allrecipes.com/recipe/11721/tuna-macaroni-salad/",
      "website":"allrecipes"
    }
  ]
}
```

Modules

I created a variety of modules to separate code with similar purposes into particular files.

Web scraping

Module	Purpose
database_insertion.py	Inserting recipe data scraped from websites into the database
mass_scraping.py	Using the webscrapers to scrape a large number of recipes and store them in JSON
natural_language_processing.py	Identifying the relevant part of ingredient strings
scrapers.py	Web scrapers created for specific websites

Webserver App

Module	Purpose
__init__.py	Runs when the webserver starts and handles configuration
database.py	Handles database operations that are run by the server whilst it is running
forms.py	Flask-WTF forms
models.py	Models for various parts of the program, such as Users and Recipes
reccomnder.py	Searching for and sorting recipes
routes.py	View functions for different routes of the website

Scraping recipes

Selecting websites

I considered several recipe websites to scrape from. These included the following: BBC Good Food, Simply Recipes, AllRecipes and Yummly. My criteria in selecting these websites were: 1) that they contained many directly accessible recipe pages, 2) that each recipe should clearly list the ingredients and 3) that each recipe should include the total time to make it. These properties would hopefully make scraping from them simpler and provide me with a wide range of recipes.

Abstract scrapers

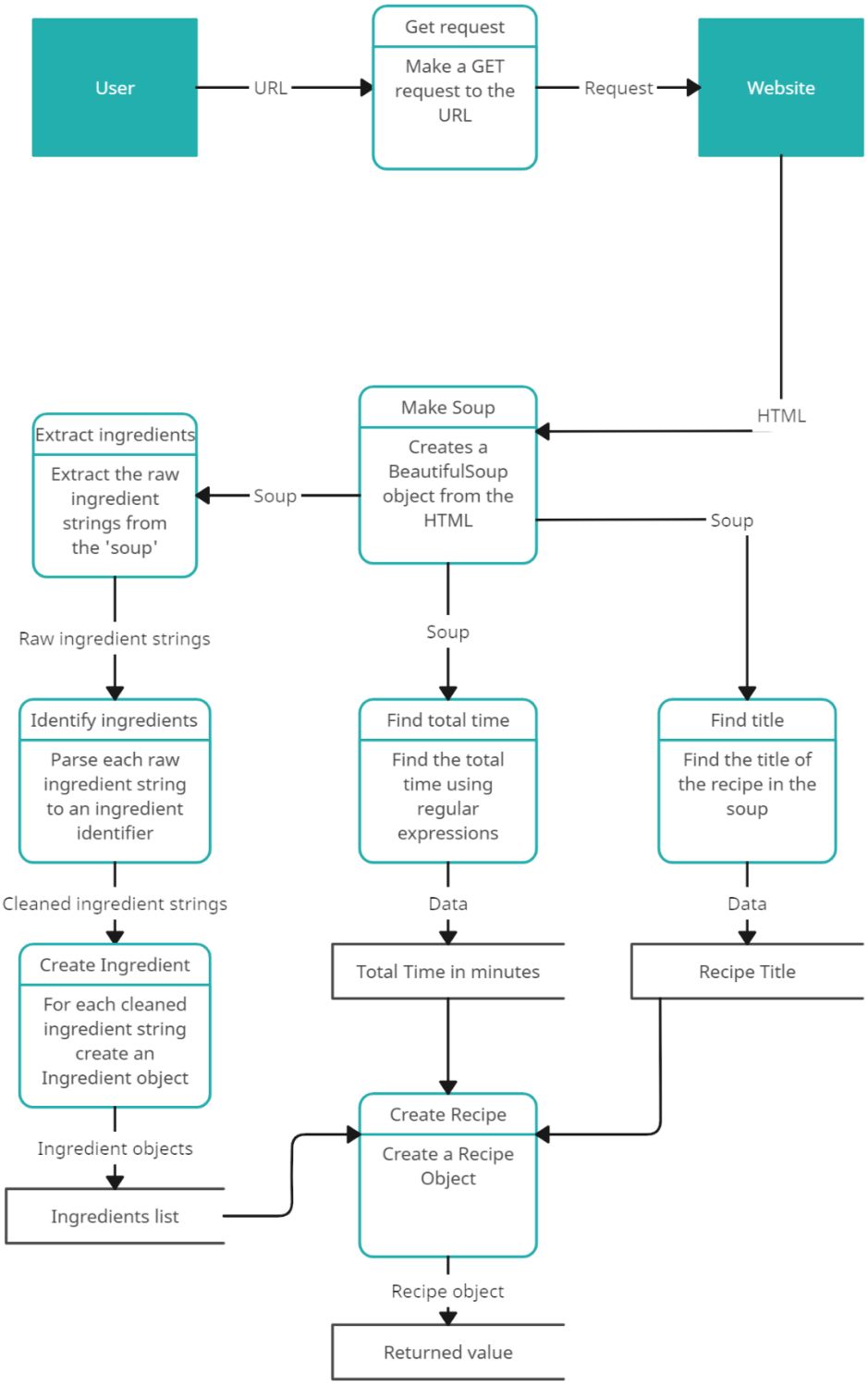
I chose to implement each web scraper as a child class of a Web Scraper abstract base class. Because I would be using different web scraper objects in the same scripts when scraping recipes, I wanted to avoid repeating code and take a more object-oriented approach. As such, I created the following abstract base class:

```
class WebScraper(ABC):
    @staticmethod
    def make_soup(url: str) -> BeautifulSoup:
        """Creates a BeautifulSoup Soup from a given url"""
        page = requests.get(url).text
        soup = BeautifulSoup(page, "html.parser")
        return soup

    @staticmethod
    @abstractmethod
    def scrape_page(url: str) -> Recipe:
        """Scrape a specific page and return a recipe object"""
        pass

    @staticmethod
    @abstractmethod
    def find_recipe_links(url: str) -> list[str]:
        """Find links to recipe pages on a given page"""
        pass

    @staticmethod
    @abstractmethod
    def find_links_to_scrape(recipe_site: str):
        """Create a text file of links to recipe pages"""
        pass
```

The “abstractmethod” decorator designates a method as abstract – allowing it to be overridden by children of the class. For each website that I scrape from I have created a child class that inherits from this abstract base class and overrides the abstract methods with behavior specific to scraping from that website.

Scraping a recipe

Scraping a recipe is performed by the WebScraper.scrape_page method. This process involves extracting the title, ingredients, and total cooking time from a recipe page. The below data flow diagram illustrates this process. All scraper classes inherit from the WebScraper class, and follow this process, although the implementations are specific to the website in question. The code for this can be found in the Technical Solution section.

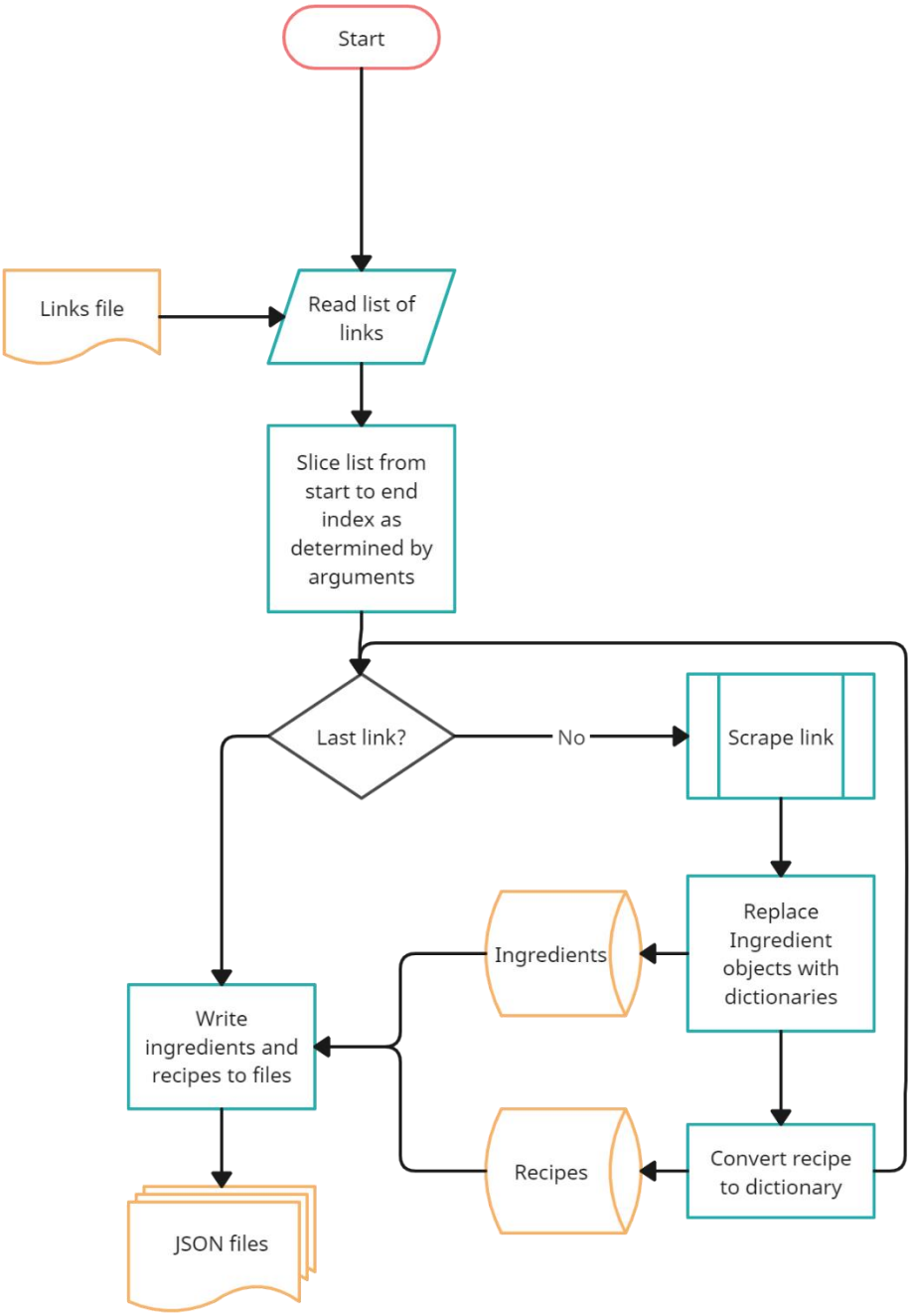


Regular expressions

Pattern	Purpose
(?<=Total Time:\n)(*\d+ (hr(s)?|min(s)?) *)+	This is used to find the 'total time' on a recipe page. It will match 1 or more instances of 0 or more whitespaces, then 1 or more digits followed by "hr", "hrs", "min or "mins" – all preceded by a non-capturing pattern of the string "Total Time:\n".
\d+(?= hr(s)?)	This will find the numerical hour component of the total time. It will match 1 or more digits followed by a non-capturing pattern of "hr" or "hrs".
\d+(?= min(s)?)	This will find the numerical minutes component of the total time. It will match 1 or more digits followed by a non-capturing pattern of "min" or "mins".

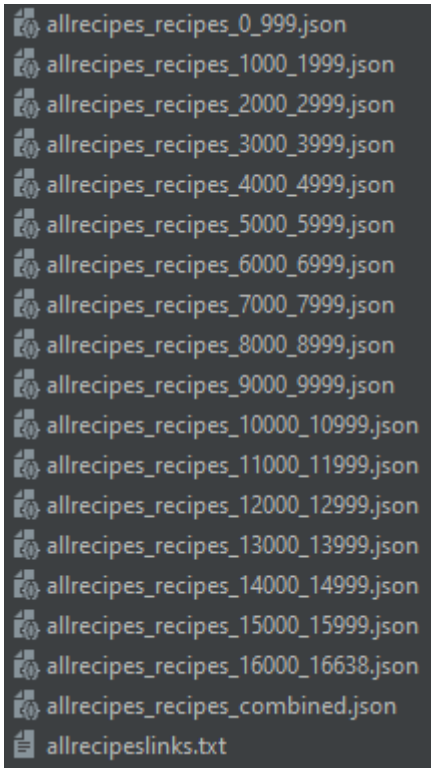
Mass scraping

Before scraping a large number of recipes, I first had to create a list of recipe URLs to scrape. For example, with the Allrecipes website I searched every page linked on the navigation bar, on the A-Z of recipes and on the A-Z of ingredients for potential recipe page links. Once I had done this, I stored all the links in a text file in a folder specifically for data scraped from Allrecipes. I then realized that scraping all 16,639 links would not only be very slow but would also run the risk of losing all the scraped data if an error was raised and the program stopped. As such, I chose to implement a 'batch' scraping system, where a particular slice of the list of links was scraped and stored in a JSON file. This process was repeated for the SimplyRecipes website, the other website I scraped recipes from.


Start
Links file
Read list of links
Slice list from start to end index as determined by arguments
Last link?
No
Scrape link
Replace Ingredient objects with dictionaries
Ingredients
Write ingredients and recipes to files
Recipes
Convert recipe to dictionary
JSON files

Once all the links had been scraped and turned into Recipe objects, I combined the JSON files into a single file.

```
def combine_json(recipe_site: str, ingredients_or_recipes: str):
    """Combine all json files into one, ingredients_or_recipes should be one of _ingredients_ or
_recipes_"""
    data_path = f"{recipe_site}_data\\"
    out_path = data_path + f"{recipe_site}{ingredients_or_recipes}combined.json"
    file_paths = [file for file in os.listdir(data_path) if ingredients_or_recipes in file and
file.endswith("json")]
    data_list = []
    for path in file_paths:
        with open(data_path + path, "r") as file:
            data = json.load(file)
            data_list += data[recipe_site]
    data = {recipe_site: data_list}
    with open(out_path, "w+") as file:
        json.dump(data, file)
```



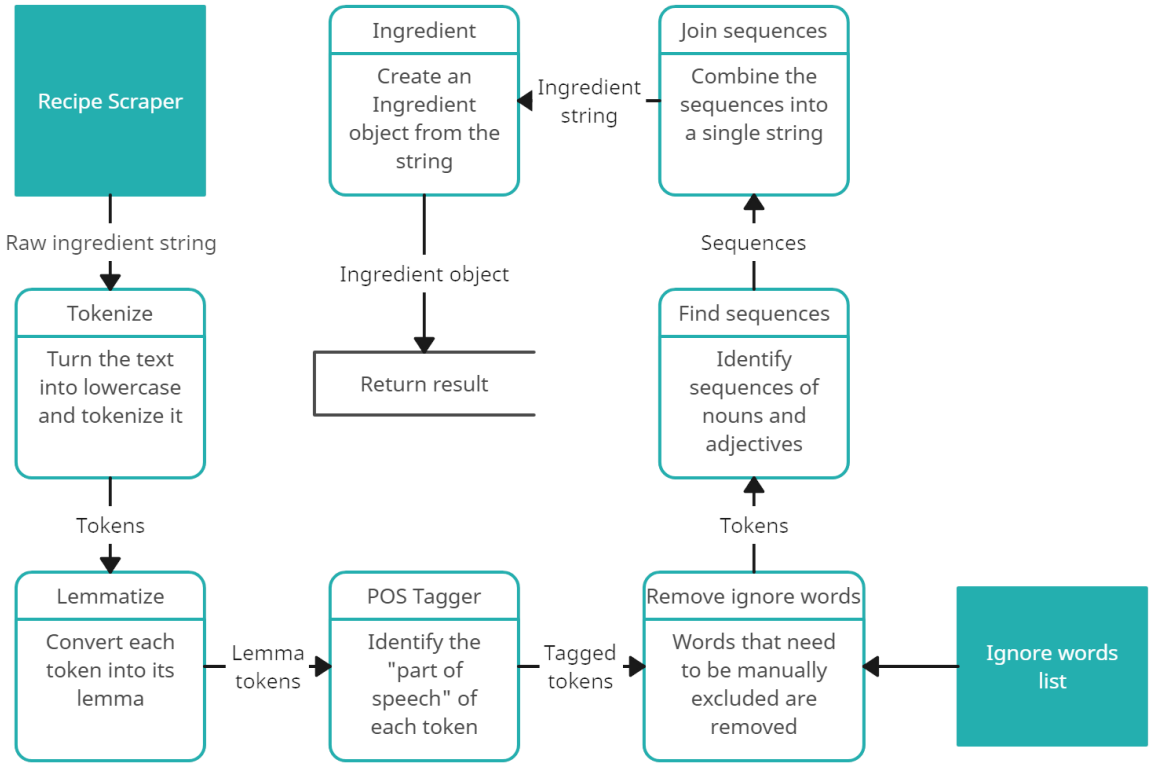
This image shows some batch JSON files - each created from 1000 links - and the combined file containing all the scraped recipe informtion. When implementing this function I used the abstract

base class “WebScraper” rather than any specfic scraper. This avoids repeating code for each scraper and allows me to simply change the scraper object parsed to the batch scrape function to change which website is being scraped. This can be seen in the defintion of the function where the type of scraper is the abstract scraper class, but a specfic scraper is created to be passed to the function.

```
def batch_scrape_pages(recipe_site: str, scraper: scrapers.WebScraper, start: int, end: int):
    """Scrape recipe links from the relevant recipe links file from a start index to the end index and
store the result
    in a json file"""

    # code


all_recipes_scraper = scrapers.AllRecipes()
all_recipes_scraper.find_links_to_scrape("allrecipes")
batch_scrape_pages("allrecipes", all_recipes_scraper, 8000, 9000)
```

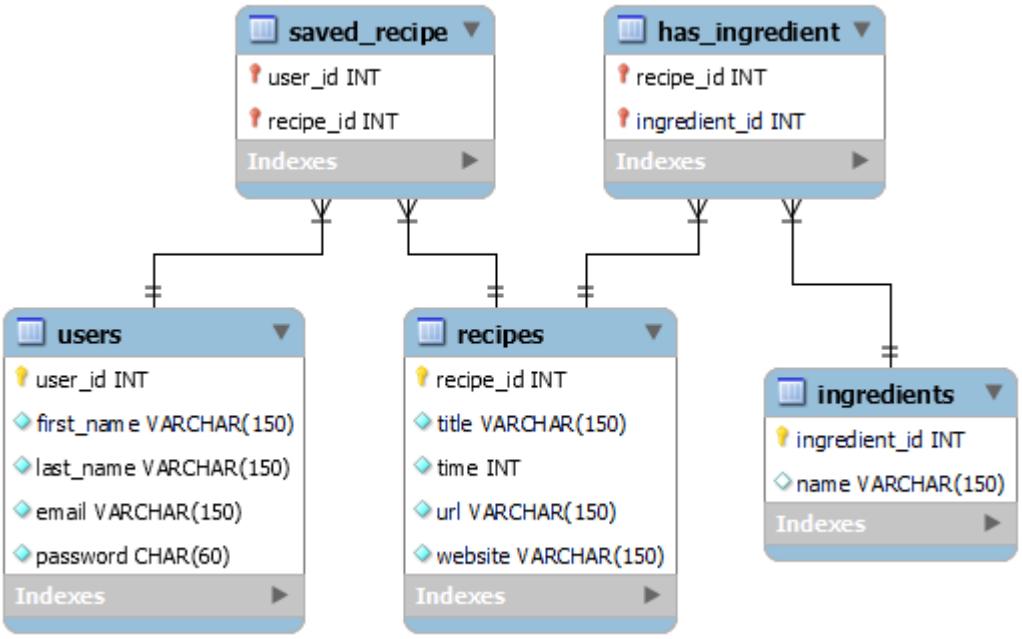
Natural Language Processing and ingredient identifcation

Ingredients as listed on recipe pages often contain a lot of superfluous information about the ingredient, such as instructions on how to prepare it (sifted, diced, etc) or its quantity. To remove this extra information, I utilised natural language processing, specifically the NLTK library. I created an algorithm that, given a raw ingredient string will turn it to lowercase, remove words that are included in a list of common words to remove, tokenize it and lemmatize each token. This produces a list of ‘cleaned’ ingredient strings. I then extract sequences of consecutive nouns and adjectives from the cleaned ingredient strings and join these sequences to produce a final ‘clean’ string containing the relevant part of the raw ingredient string. The process of identifying sequences is described in the Key Algorithms section.



Database design

Enhanced entity-relationship model



This diagram, created using the MySQL workbench, shows in detail the structure of my database. The tables "saved_recipe" and "has_ingredient" are linking tables that create relations between the users and recipes tables, and the recipes and ingredients tables, respectively.

Creation of database and tables

The following queries were executed directly in the MySQL CLI.

SQL query	Purpose
CREATE DATABASE nea_project;	Creates the database
CREATE TABLE ingredients (ingredient_id INT unsigned NOT NULL AUTO_INCREMENT, name VARCHAR(150) NOT NULL, PRIMARY KEY (ingredient_id), UNIQUE(name));	Creates the ingredients table. (ingredient_id) is an automatically generated, positive, unique primary key identifier for each record. (name) is the name of the ingredient and has a unique constraint – this avoids duplicate ingredient records.
CREATE TABLE recipes (recipe_id INT unsigned NOT NULL AUTO_INCREMENT, title VARCHAR(150) NOT NULL, time INT NOT NULL, url VARCHAR(150), website VARCHAR(150), PRIMARY KEY (recipe_id),);	Creates the recipes table. (recipe_id) is an automatically generated, positive, unique primary key identifier for each record. (title) is the title of the recipe. (time) is the total time a recipe takes to make. Recipe pages with no recorded total time are recorded as having a total time of -1, so the field is signed.

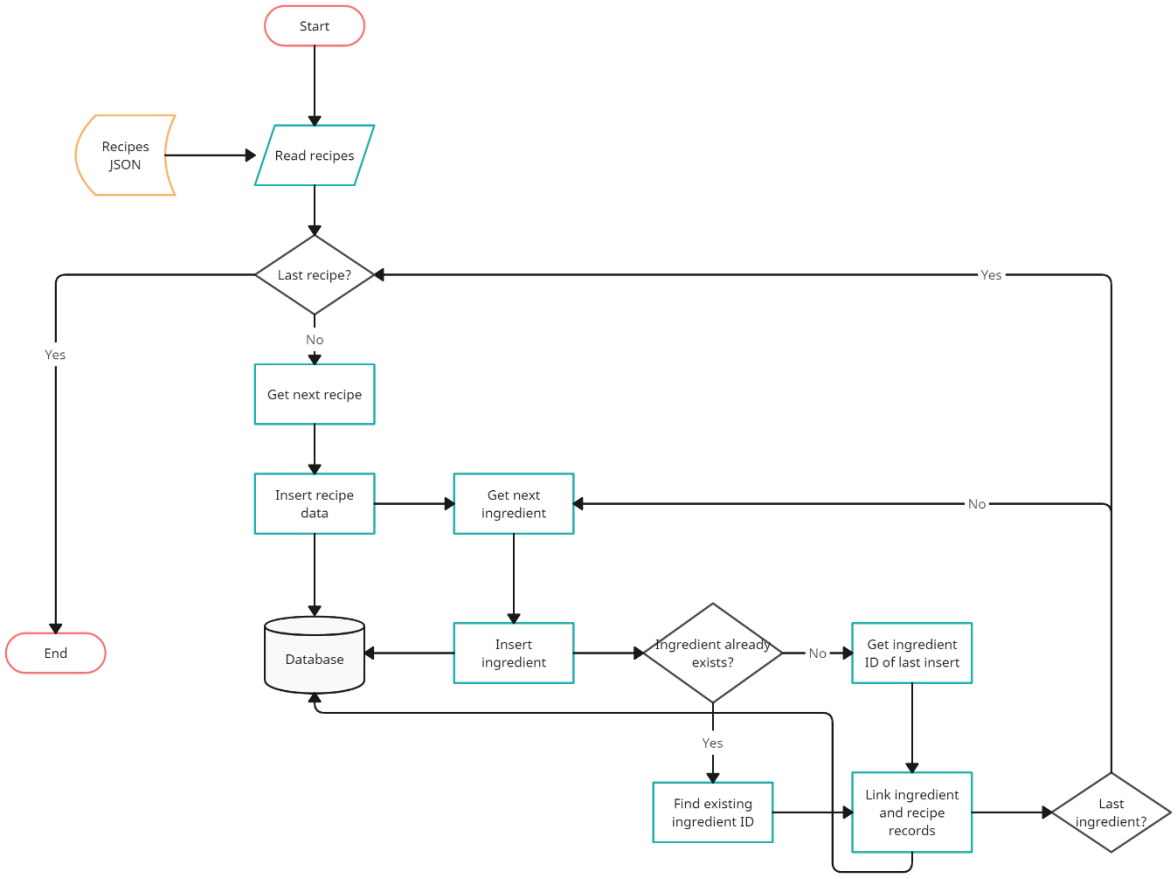
	(url) is the URL of the webpage the recipe was scraped from. (website) is the name of the website the recipes was scraped from.
CREATE TABLE has_ingredient (recipe_id INT unsigned NOT NULL, ingredient_id INT unsigned NOT NULL, PRIMARY KEY (recipe_id, ingredient_id), FOREIGN KEY (recipe_id) references recipes (recipe_id), FOREIGN KEY (ingredient_id) references ingredients (ingredient_id));	Creates the linking table between recipes and ingredients. (recipe_id) and (ingredient_id) are foreign keys that form the composite primary key of the records. They reference primary keys in their respective tables.
CREATE TABLE users (user_id INT unsigned NOT NULL AUTO_INCREMENT, first_name VARCHAR(150) NOT NULL, last_name VARCHAR(150) NOT NULL, email VARCHAR(150) NOT NULL, password CHAR(60) NOT NULL, PRIMARY KEY (user_id), UNIQUE(email));	Creates the users table. (user_id) is an automatically generated, positive, unique primary key identifier for each record. (first_name) and (last_name) are the name of the user. (email) is the user's email address, and is unique (password) is the hash of the user's password.
CREATE TABLE saved_recipe (user_id INT unsigned NOT NULL, recipe_id INT unsigned NOT NULL, PRIMARY KEY (user_id, recipe_id), FOREIGN KEY (user_id) references users (user_id), FOREIGN KEY (recipe_id) references recipes (recipe_id));	Creates the saved recipes linking table (recipe_id) and (user_id) are foreign keys that form the composite primary key of the records. They reference primary keys in their respective tables.

Inserting data

Inserting scraped recipe data from JSON files

To insert recipe data into the database from the JSON files it was stored in, I chose to write a Python script using the MySQL connector for Python.[8] Similar to my approach to scraping the recipes, I created a function that would read a specified slice of from a list of recipe stored in JSON and insert those recipes and their ingredients into the database. Additionally, I created a linking record between each ingredient and each recipe it is used in. The diagram below illustrates this process, and the table shows the queries used to insert data.

[8] MySQL Connector/Python Developer Guide. (2022, December 22). https://dev.mysql.com/doc/connector-python/en/



Query	Purpose
INSERT INTO recipes (title, time, url, website) VALUES (%s, %s, %s, %s)	This query inserts recipe data into the recipes table. Each “%s” parameter is replaced with the recipe title, time, url and website respectively when the query is executed by the connector.
INSERT IGNORE INTO ingredients (name) VALUES ("{ingredient}")	This inserts ingredient data into the ingredients data. {ingredient} is replaced by the python script with the name of the ingredient before the query is executed by the connector. The IGNORE statement means that the query ignores duplicates - if an ingredient already exists with that name, then no insert is performed and no error is raised.
SELECT * FROM ingredients WHERE name="{ingredient}”	This finds ingredient records with a given name. This is used to find an existing ingredient record id when the above insert ignores a duplicate name.
INSERT IGNORE INTO has_ingredient (recipe_id, ingredient_id) VALUES (%s, %s)	This creates the linking record between an ingredient and the recipe it is used in. The “%s” parameters are replaced with the recipe id and ingredient id, respectively.

Adding a user profile to the database

```
def add_profile_to_database(profile: User) -> bool:
    cursor = db.connection.cursor()
    values = (profile.first_name, profile.last_name, profile.email, profile.password_hash)
    sql = "INSERT INTO users (first_name, last_name, email, password) VALUES (%s, %s, %s, %s);"
    try:
        cursor.execute(sql, values)
    except MySQLdb.IntegrityError:
        # Throws when trying to create a profile with an already used email
        return False
    db.connection.commit()
    return True
```

This function is used when registering a new user account. It takes a User object as an argument which contains the data of the new account. These values are then interpolated into an SQL statement which is then executed by the database connector object. A try-catch statement is used to handle integrity errors, which are raised when a user is being inserted into the database with an email address that is already in use - the email field of the user table has the “unique” property. If this is error is raised, then the function returns False to indicate a failure. Otherwise, it returns True.

Query	Purpose
INSERT INTO users (first_name, last_name, email, password) VALUES (%s, %s, %s, %s);	Inserts a user record into the database. The four “%s” parameters are replaced with the User object’s first name, last name, email, and password attributes

Saving a recipe to a user's account

```
def user_save_recipe(recipe_title: str, user: User):
    # Get recipe id by title
    cursor = db.connection.cursor()
    recipe_select = f"SELECT recipe_id FROM recipes WHERE title='{recipe_title}';"
    cursor.execute(recipe_select)
    recipe_id = cursor.fetchone()[0]

    insert_sql = f"INSERT INTO saved_recipe (user_id, recipe_id) VALUES (%s, %s);"
    values = (user.user_id, recipe_id)
    cursor.execute(insert_sql, values)
    db.connection.commit()
```

This function is used when a user saves a recipe result. The function takes a recipe title and a User object as arguments. The relevant recipe record is found with a SELECT statement and the recipe id is stored. The recipe id and user id are then inserted into the saved recipe linking table. This creates a relation between the user record and the recipe record.

Query	Purpose
SELECT recipe_id FROM recipes WHERE title='{recipe_title}';	Selects recipe records with the specified title
INSERT INTO saved_recipe (user_id, recipe_id) VALUES (%s, %s);	Inserts the user ID and recipe ID into the linking table and creates a relation between the user record and the recipe record

Altering Data

Changing a user's email address

```
def change_user_email(user: User, new_email: str) -> bool:
    if find_user_by_email(new_email) is not None:
        return False
    cursor = db.connection.cursor()
    sql = f"UPDATE users SET email='{new_email}' WHERE user_id={user.user_id};"
    cursor.execute(sql)
    db.connection.commit()
    return True
```

From the profile page users can change the email address associated with their account. The function that performs this takes as arguments a User object representing the current user, and a

new email address string. A check is performed to make sure that the email address is not already in use. If it is, then the function returns False and an appropriate error message is displayed. Otherwise, a SQL query is executed by the database connector that updates the user's record and the function returns True to indicate a success.

Query	Purpose
UPDATE users SET email='{new_email}' WHERE user_id={user.user_id};	Changing the email address of a user – specified by the user ID – to a value specified by the new_email argument.

Changing a user's password

```
def change_user_password(user: User, new_password_hash: str):

  cursor = db.connection.cursor()

  sql = f"UPDATE users SET password='{new_password_hash}' WHERE user_id={user.user_id};"

  cursor.execute(sql)

  db.connection.commit()
```

Users can also change their passwords. This function takes as arguments a User object representing the current user, and a new password hash string to be stored in the database. The hash of the password is calculated outside this function, so this function simply executes the following SQL query.

Query	Purpose
UPDATE users SET password='{new_password_hash}' WHERE user_id={user.user_id};	Changing the stored password hash of a user – specified by the user ID – to a value specified by the new_password_hash argument.

Correcting mistakes in the database

After examining my database, I realized that some of the "name" fields of ingredient records contained a "®" character. Because this could have prevented matches being found when searching the database, I removed all these characters with the following query.

Query	Purpose
UPDATE ingredients SET name = REPLACE(name, "®", "")	Removing the ® from all name fields in the ingredients table

Selecting Data

Selecting recipes using a query

```
def select_recipes_with_query(query: Query) -> list[Result]:
    """Find all recipes with at least one ingredient that matches a token in the query"""
    # open database connection
    cursor = db.connection.cursor()
    find_recipes_sql = create_recipe_select_sql(query)


    cursor.execute(find_recipes_sql)
    results = cursor.fetchall()
    recipes = []
    for result in results:
        ingredients = []
        for ingredient in result[5].split(","):
            ingredients.append(Ingredient(ingredient))
        recipes.append(Result(result[1], ingredients, result[2], result[3], result[4]))
    return recipes
```

A key operation in my project is finding any recipe that matches a user's input. This is performed by a function that takes a Query object as an argument and returns a list of unsorted Result objects. The function opens a connection to the database and executes a SQL query - created using the below function - from the Query object that selects any recipe that contains at least one ingredient that matches an ingredient in the Query object. The returned rows are then iterated over and the data for each row is passed into a Result object and appended to a list.

Creating a data frame using a query

```
def recipe_dataframe_from_query(query: Query) -> pd.DataFrame:
    """Pandas dataframe from results of query"""
    sql = create_recipe_select_sql(query)
    connection = db.connect
    dataframe = pd.read_sql(sql, connection).set_index("recipe_id")
    return dataframe
```

When calculating relevancy scores from the results, the above function can be sidestepped by using the Pandas inbuilt read_sql method. This is more time efficient than creating a list of Results and transforming the list into a data frame as the function described above runs in linear time, and as such is quite slow for larger numbers of recipes. The function to create a data frame takes a Query

object as an argument and returns a data frame object. It uses the same below function to generate the SQL query as the above function. The SQL query is passed to the read_sql function with the database connection object, which returns the data frame object. The index of the data frame is set be the database IDs of the recipes.

Creating the SQL to select recipes

This function is used to create the SQL query string that selects any recipes that match the given Query object. A regular expression is created by joining the query's cleaned tokens – individual words – with "|" characters. This means that the expression will match any text that is equal to any of the tokens. Depending on whether the Query object has a set maximum time that a recipe can take to prepare, one of the two below statements is used with the generated regular expression to produce a string that is then returned.

Query	Purpose
SELECT recipes.*, GROUP_CONCAT(ingredients.name SEPARATOR ',') AS ingredients FROM recipes, ingredients, has_ingredient WHERE recipes.recipe_id=has_ingredient.recipe_id AND ingredients.ingredient_id=has_ingredient.ingredient_id AND recipes.recipe_id IN (SELECT has_ingredient.recipe_id FROM ingredients, has_ingredient WHERE has_ingredient.ingredient_id=ingredients.ingredient_id AND ingredients.ingredient_id = ANY(SELECT ingredient_id FROM ingredients WHERE name REGEXP "{ingredients_regex}")) GROUP BY recipes.recipe_id, recipes.title;	This query will return all data about recipes and a comma-separated list of their ingredients where the recipes contain at least one ingredient whose name matches a given regular expression (ingredients_regex). This is used to find recipes that are relevant to user's inputted list of ingredients.
SELECT recipes.*, GROUP_CONCAT(ingredients.name SEPARATOR ',') AS ingredients FROM recipes, ingredients, has_ingredient WHERE recipes.recipe_id=has_ingredient.recipe_id AND ingredients.ingredient_id=has_ingredient.ingredient_id AND recipes.recipe_id IN (SELECT has_ingredient.recipe_id FROM ingredients, has_ingredient WHERE has_ingredient.ingredient_id=ingredients.ingredient_id AND ingredients.ingredient_id = ANY(SELECT ingredient_id FROM ingredients WHERE name REGEXP "{ingredients_regex}")) AND recipes.time <= {max_time} GROUP BY recipes.recipe_id, recipes.title;	As above but limits the returned recipes to ones with a preparation time below or equal to a given value (max_time).

Finding a user record using an email address

```
def find_user_by_email(email: str) -> Optional[User]:
    cursor = db.connection.cursor()
    sql = f'SELECT * FROM users WHERE email="{email}";'
    cursor.execute(sql)
    if cursor.rowcount == 0:
        # no results found
        return None
    else:
        result = cursor.fetchone()
        user = User(user_id=result[0], first_name=result[1], last_name=result[2], email=result[3],
                    password_hash=result[4])
        return user
```

When logging a user in, registering a user, or modifying a user's data it is necessary to retrieve their record from the database, or to check that no user already exists in the database with that email address. The email field of the user table is unique, so no two user records can have the same email value. The below SQL query is executed by the database connector. If no results are found the function returns None. Otherwise, the function builds a User object using the selected record and returns it.

Query	Purpose
SELECT * FROM users WHERE email="{email}";	Selecting all records from the users table where the value of the email address is a given value (email). As the email field is unique this will only ever return 0 or 1 records.

Finding a user's saved recipes

```
def find_user_saved_recipes(user: User) -> list[Recipe]:

    cursor = db.connection.cursor()

    sql = f"SELECT * FROM recipes JOIN saved_recipe ON recipes.recipe_id=saved_recipe.recipe_id
WHERE" \

        f" user_id={user.user_id};"

    cursor.execute(sql)

    results = cursor.fetchall()

    recipes = []

    for result in results:

        recipes.append(Recipe(title=result[1], ingredients=[], total_time=result[2], url=result[3],
website=result[4]))

    return recipes
```

When loading the profile page, it is necessary to find all the recipes that a user has saved. This function takes a User object as an argument and returns a list of Recipe objects. A SQL select statement is executed to find all recipes linked to the user ID of the User object – using a JOIN statement to connect the recipes table and the saved recipes table. Each result is then used to build a Recipe object and appended to a list.

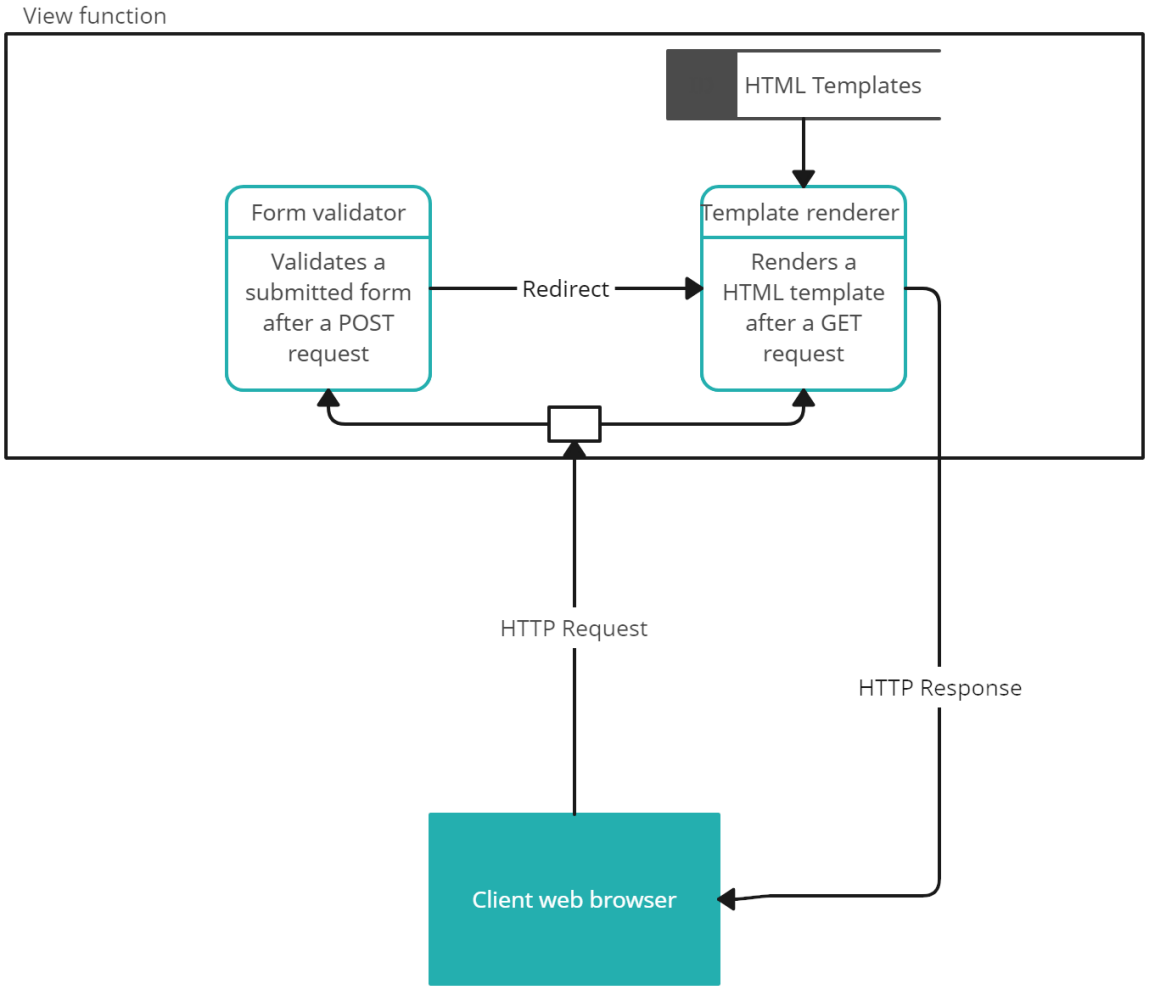
Query	Purpose
SELECT * FROM recipes JOIN saved_recipe ON recipes.recipe_id=saved_recipe.recipe_id WHERE user_id={user.user_id};	Selecting all recipes that have been saved by a user with a specified ID

Design of the website

My project uses a website to provide a user-friendly interface for searching recipes, as well as a number of other functions. Users make requests to a Flask web server which then renders HTML templates with dynamic Jinja variables and logic and returns the page to the user. This section will focus on the design of the website itself.

View functions

A view function is the Python function that runs when a request is made to a path of the web site. These functions are located in the routes.py file and are described in detail in the Key Functionalities section.



This diagram shows how view functions operate. Upon receiving a request to a registered resource path, the web server will call the relevant view function. If the request is a POST request – i.e.: is submitting data through a form – then the request data will be validated, and an operation will be performed. This operation ranges from performing a search for recipes to registering an account. If the request is a GET request, then the relevant HTML file will be rendered.

Sessions

Flask provides a session object. This is effectively dictionary that is stored server-side, and stores information about every client that makes a request to the server. This dictionary is encrypted with the secret key defined in the configuration. I use the session object to store data about a client as they use the website, such as what account is logged in and what their last recipe search was. A key piece of data that is stored is the email address of the currently logged in account. This is used to handle user authentication and display relevant information based on the account currently logged in.

Configuration

The configuration for the Flask app is defined in a config.py file. This file is hidden from git as it is included in the gitignore. It contains the secret key (a random sequence generated using the inbuilt uuid module) used by Flask to sign the encrypted session and by Flask-WTF to prevent CSRF attacks. Additionally, it contains the password to the MySQL database, as well as the database name, username, and host. Data from the config file is loaded by the server when it initializes.

app/__init.py__

```
from flask import Flask, session

import config

from flask_wtf.csrf import CSRFProtect

from flask_mysqldb import MySQL


app = Flask(__name__)

csrf = CSRFProtect(app)

db = MySQL(app)

app.config["SECRET_KEY"] = config.SECRET_KEY

app.config["MYSQL_DB"] = config.DATABASE_NAME

app.config["MYSQL_HOST"] = config.DATABASE_HOST

app.config["MYSQL_PASSWORD"] = config.DATABASE_PASSWORD

app.config["MYSQL_USER"] = config.DATABASE_USERNAME

session.permanent = True

from app import routes
```

config.py

```
SECRET_KEY = “key”

DATABASE_NAME = "database"

DATABASE_HOST = "localhost"

DATABASE_PASSWORD = "password"

DATABASE_USERNAME = "username"
```

Jinja2 templates

Jinja is an engine for rendering HTML “templates.” Blocks of python-like code in the HTML contained within {} or {{}} brackets are evaluated by the rendering engine when the flask render_template function is called. This allows for static HTML to be populated with information using python code.

Rendering Forms

A key use of this is rendering forms. To simplify the process of including forms in my HTML code, I first create Flask-WTF Form objects to represent each form. These are then included in the HTNL using the blocks described above and passed to the render_template function in the view function. The label attribute of each form field is used to included text, and the field is called to create the input field HTML:

```
        <div id="current-email-field" class="form-field email-field">

            {{change_email_form.current_email.label}}

            <br>

            {{change_email_form.current_email()}}

         </div>
```

Inheritance and the navbar

Another functionality of Jinja is template inheritance. Templates can inherit from one another template. The HTML of the inheriting template will be included in the “base” HTML of the inherited template. This allows for the construction of a “base” template that all the other templates will share code with. In practice this means that elements that should be included on all pages – such as the navigation bar that has links to all key pages – can be included in the base template and then do not need repeating in all inheriting templates. In the “base.html” template the content of each page that is replaced by the inheriting template is represented by:

```
<div id="content">

  {% block content %}{% endblock %}

</div>
```

And in the inheriting templates as:

```
{% extends "base.html" %}

{% block title %}Info{% endblock %}

{% block content%}

<!-- content -->

{% endblock%}
```

Iteration and logic

Jinja also allows for iteration to be used to generate HTML. I utilize this when displaying results, as the number of recipes that needs to be displayed is variable – but all need to follow the same structure. The HTML inside the loop is added to the document for each iteration.

I also use logic to check if any of the words in each ingredient name match any of the ingredient words given in the query. If they do, then the ingredient name is made bold.

```
{% for recipe in results %}
<div class="recipe_result" id="recipe-{{ loop.index }}" style="border: solid black">
  <a href="{{ recipe.url }}">
    <div class="recipe-title">
      <h3>{{ recipe.title }}</h3>
    </div>
    <div class="recipe-time">
      <p>Total time: {{ recipe.formatted_time }}</p>
    </div>
    <div class="recipe-ingredients">
      <ul>
        {% for ingredient in recipe.ingredients %}
        {% if ingredient.name.split()|select("in", query_ingredients)|first %}
          <li><b>{{ ingredient.name }}</b></li>
        {% else %}
          <li>{{ ingredient.name }}</li>
        {% endif %}
        {% endfor %}
      </ul>
    </div>
  </a>
  <div class="recipe-save-button">
    <form method="POST" action='{{url_for("save_recipe")}}'>
      {{form.hidden_tag()}}
      <input type="submit" value="Save recipe" name="save_button">
      <input type="hidden" value="{{recipe.title}}" name="recipe_title">
    </form>
  </div>
</div>
{% endfor %}
```

CSS

I also included some simple CSS to set a background colour, format the navigation bar and to make links look like normal text. I also added simple borders to my tables

```
body
{
   background-color: lightgrey;
}
#navbar
{
   overflow: hidden;
   float: center;
}

a {
color: inherit;
text-decoration: none;
}

table, td, th {
   border: 1px solid black;
   border-collapse: collapse;
}
```

Preventing Cross Site Request Forgery

A risk created by having forms on a website is that of a CSRF attack, where a malicious actor exploits a user's authentication to submit harmful requests to the web server from a trusted client.[9] Fortunately, the Flask-WTF library provides a method for protecting against these attacks.[10]

```
from flask_wtf.csrf import CSRFProtect

app = Flask(__name__)
csrf = CSRFProtect(app)
```

[9] KirstenS. (n.d.). Cross Site Request Forgery (CSRF). OWASP. Retrieved 24 January 2023, from https://owasp.org/www-community/attacks/csrf

[10] CSRF Protection—Flask-WTF Documentation (0.15.x). (n.d.). Retrieved 24 January 2023, from https://flask-wtf.readthedocs.io/en/0.15.x/csrf/

This works by creating a CSRF token when a user sends an authentication request. This token is included, along with a session cookie, in the POST response to the user. The token is included in HTML of a form with the following method added to the HTML using Jinja.

```
{{ form.hidden_tag() }}
```

When the user submits the form the token and session data are sent along with the form data to be validated by the flask web server.[11]

Forms

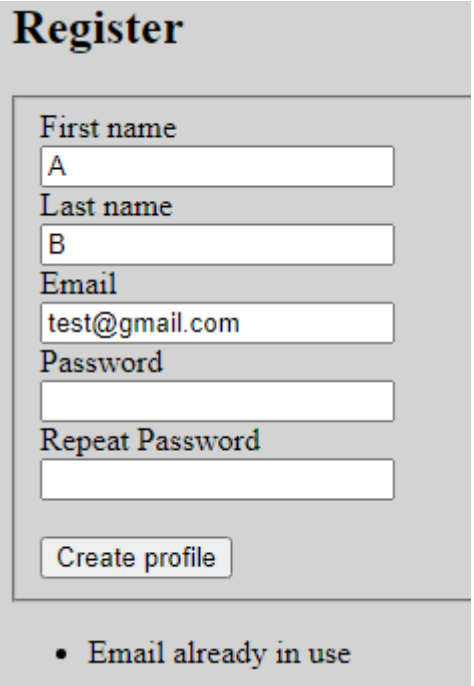
I used several HTML forms in my project to create a user interface where users could submit data. To simplify the process of creating forms I used the Flask-WTF extension. Form objects are passed to the template rendering function and are then included in the HTML of the page that is returned. Each form is defined as a class that inherits from the FlaskForm base class. Each field is defined as an attribute of the class. If data is submitted that does not meet the requirements of the validators, then a message is displayed using the alert system.

Alerts

Alerts are used to give temporary feedback messages to the user on the webpage. When invalid data is submitted to a form, the inbuilt Flask flash function is called in the view function of the form's page. When the page is reloaded after the form is submitted, a message will be displayed. When another request is made the alert log is cleared. For example, the python code:

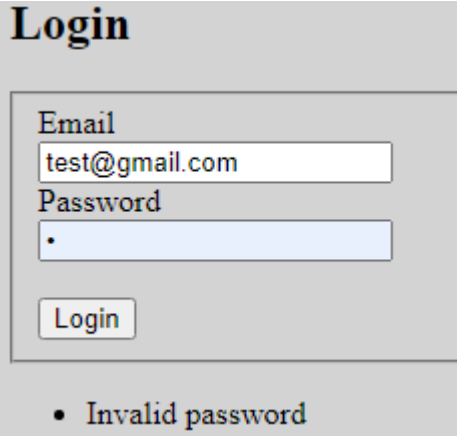
```
if not profile_added_successfully:
    flash("Email already in use", "form")
```

Will flash the message "Email already in use" when a user attempts to register an account with an email address that is already associated with another account. This is then displayed underneath the form.



[11] Shaji, A. (2022, October 14). CSRF Protection in Flask. https://testdriven.io/blog/csrf-flask/

Alerts are used to provide a variety of input validation messages to users that is not handled by the forms themselves, such as checking if a password is correct.



Alerts are added dynamically to the HTML using a Jinja code block that is included in the static HTML

```
{% with alerts = get_flashed_messages(with_categories=true) %}
    {% if alerts %}
        <div id="alerts">
            <ul>
                {% for type, content in alerts %}
                    {% if type=="form" or type=="info"%}
                        <li>
                            {{ content }}
                        </li>
                    {% endif %}
                {% endfor %}
            </ul>
        </div>
    {% endif %}
{% endwith %}
```

When the HTML is rendered, any alerts will be included in the HTML as unordered list items. The above if statement shows how alerts can be given specific “types.” This is used to control which alerts are shown in different areas of the webpage.

Key functionalities of the website

Searching for recipes

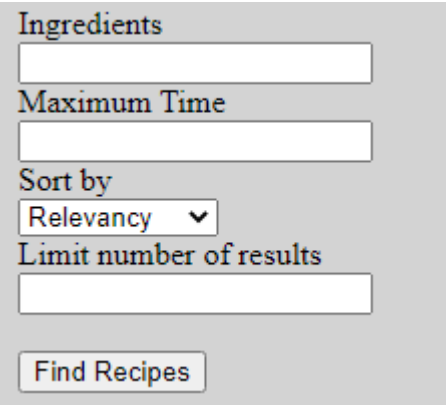
The form to search for recipes

```
class RecommenderForm(FlaskForm):

    ingredients = StringField("Ingredients", validators=[DataRequired()])

    max_time = IntegerField("Maximum Time", validators=[Optional(), NumberRange(min=1)])

    sort_mode = SelectField("Sort by",

                        choices=[('relevancy', 'Relevancy'), ('title', 'Title'), ('total_time', 'Time to cook')])

    limit = IntegerField("Limit number of results", validators=[Optional(), NumberRange(min=1)])

    submit = SubmitField("Find Recipes")
```

This form is used by users to search for recipes. It requires uses to submit a list of ingredients as a string, and allows users to set a maximum preparation time, specify how to sort the recipes and limit the number of results returned.

Field name	Field Type	Validators	Purpose
ingredients	String	Data Required	Specifying the ingredients to be used in the search
max_time	Integer	Optional Must be greater than or equal to 1	Setting a maximum preparation time for the recipes returned
sort_mode	Select		Selecting how the recipes should be sorted from the following options: Relevancy Title Total time
limit	Integer	Optional Must be greater than or equal to 1	Sets the number of recipes that will be returned
submit	Submit		Submitting the form

```
<div id="recipe-form-container" class="form">
  <form method="POST" id="recommender-form">
    <fieldset>
    {{ form.hidden_tag() }}
    <div>
      {{form.ingredients.label}}
      <br>
      {{form.ingredients()}}
    </div>
    <div id="max-time-field" class="form-field number-field">
      {{form.max_time.label}}
      <br>
      {{form.max_time()}}
    </div>
    <div id="sort-mode-field" class="form-field select-field">
      {{form.sort_mode.label}}
      <br>
      {{form.sort_mode()}}
    </div>
    <div id="limit-field" class="form-field number-field">
      {{form.limit.label}}
      <br>
      {{form.limit()}}
    </div>
    <br>
    <div id="form-submit-button" class=" form-field form-button">
      {{form.submit()}}
    </div>
  </fieldset>
  </form>
</div>
```



The index view function

The index view function handles requests made to the "home page" of the website. It provides an interface for searching for recipes. It is important to note that this function does not actually perform the search, it only handles the user input and then redirects the client to the view where the search is actually performed – the "recommend" view. When the function is called with a GET request it creates a ReccomenderForm object and renders the index.html file with that form. When the user submits the form the view function validates it and creates a Query object from the user's input. This object is then transformed into a dictionary and stored in the session so that it can be accessed by the "recommend" view function, which the user is then redirected to. The Query object itself cannot be stored in the session as only objects that are JSON serializable can be stored in the session.

```
@app.route('/', methods=["GET", "POST"])

def index():

    """Home page of the site, displaying the recommender form"""

    form = RecommenderForm()

    if form.validate_on_submit():

        query = Query(raw_ingredients=form.ingredients.data, sort_mode=form.sort_mode.data,
max_time=form.max_time.data, limit=form.limit.data)

        session["query"] = query.__dict__

        return redirect(url_for("recommend"))

    return render_template("index.html", form=form)
```

Recommending recipes

This view function performs a search from and displays the results of the query stored in the session to the user. If no query data is stored in the session, then the user is redirected to the home page. Otherwise, a SaveRecipeForm object is created to be included in the HTML template and a Query object is created from the data stored in the session. The find_results function described in the Key Algorithms section is then called to find results based on the Query object. The returned list of Result objects is then sliced to limit the number of results displayed to the value specified by the user. The "recommend.html" template is then rendered with the results. The SaveRecipeForm is passed to be included in the HTML as a button that users can click to save a recipe. The list of

ingredients in the query is also passed so that matching recipes can be highlighted using Jinja. Each of the recipes is listed with its title, time to cook and ingredients – with ingredients that match the query in bold. Clicking on the recipe will open the page the recipe was scraped from.

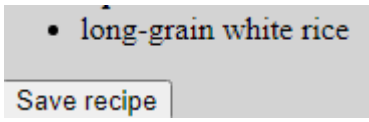
```
@app.route("/recommend", methods=["GET", "POST"])

def recommend():

    if "query" in session:

        form = SaveRecipeForm()

        query_data = session["query"]

        query = Query(raw_ingredients=query_data["raw_ingredients"],
sort_mode=query_data["sort_mode"],

                      max_time=query_data["max_time"], limit=query_data["limit"])

        query.results = recommender.find_results(query)

        # Limit number returned

        query.results = query.results[:query.limit]

        return render_template("recommend.html", results=query.results,
query_ingredients=query.cleaned_tokens,

                               form=form)

    else:

        return redirect("/")
```

Saving recipes

The form to save a recipe

```
class SaveRecipeForm(FlaskForm):

    # Provides a csrf token to the save recipe button

    pass
```

This form is used to save recipes, and although it takes no user-facing input the title of the recipe the button is associated with is included in a hidden field. This means that when the form is submitted, the recipe that is to be saved can be identified from the form data.



```
<div class="recipe-save-button">
    <form method="POST" action='{{url_for("save_recipe")}}'>
      {{form.hidden_tag()}}
      <input type="submit" value="Save recipe" name="save_button">
      <input type="hidden" value="{{recipe.title}}" name="recipe_title">
    </form>
  </div>
```

The save recipe view function

When recipes are shown on the recommendation page, each one has a button that the user can click to save that recipe. When the user clicks this button, it submits a POST request with the form data to the save recipe view function. If no user is logged in, then the user is redirected to the login page. Otherwise, the database operation to save the recipe is called and the user is redirected to their profile page, where they can see their saved recipes.

```
@app.route("/save_recipe", methods=["POST"])
def save_recipe():
  if "active_user_email" in session:
    if request.method == "POST":
      user = find_user_by_email(session["active_user_email"])
      user_save_recipe(request.form["recipe_title"], user)
    return redirect("/profile")
  else:
    flash("Please log in to save recipes", "info")
    return redirect("login")
```

The info page

The view function for the page that gives information on how to use the website.

```
@app.route('/info')
def info():
  # Route for info page
  return render_template("info.html")
```

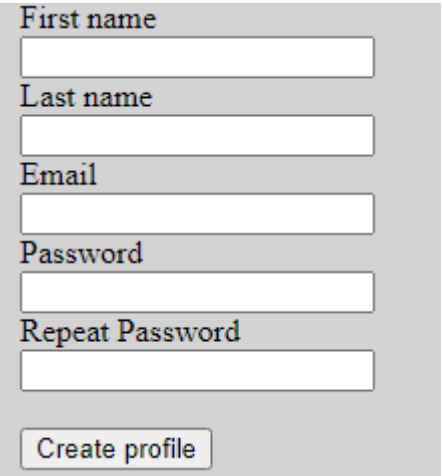
Registering an account

Form to register an account

```
class CreateProfileForm(FlaskForm):

    first_name = StringField("First name", validators=[DataRequired()])

    last_name = StringField("Last name", validators=[DataRequired()])

    email = EmailField("Email", validators=[DataRequired()])

    password = PasswordField("Password", validators=[DataRequired()])

    repeat_password = PasswordField("Repeat Password", validators=[DataRequired()])

    submit = SubmitField("Create profile")
```

This form is used to register an account. Users specify their name, email, and password. They are asked to repeat their password to make sure they typed it correctly. However, this check is performed in view function for the register page and not by the form itself.

Field name	Field Type	Validators	Purpose
first_name	String	Data Required	The user's first name
last_name	String	Data Required	The user's last name
email	Email	Data Required Valid email address	The user's email address. The input is checked to ensure that it is a valid email address
password	Password	Data Required	The user's password. Any value entered in this field is obscured whilst it is being typed
repeat_password	Password	Data Required	The user's password repeated to ensure that it is typed correctly.
submit	Submit		Submitting the form



```
<div id="create-profile-form-container" class="form">
  <form method="POST" id="create-profile-form">
    <fieldset>
      {{form.hidden_tag()}}
      <div id="first-name-field" class="form-field string-field">
        {{form.first_name.label}}
        <br>
        {{form.first_name()}}
      </div>
      <div id="last-name-field" class="form-field string-field">
        {{form.last_name.label}}
        <br>
        {{form.last_name()}}
      </div>
      <div id="email-field" class="form-field string-field">
        {{form.email.label}}
        <br>
        {{form.email()}}
      </div>
      <div id="password-field" class="form-field password-field">
        {{form.password.label}}
        <br>
        {{form.password()}}
      </div>
      <div id="repeat-password-field" class="form-field password-field">
        {{form.repeat_password.label}}
        <br>
        {{form.repeat_password()}}
      </div>
      <br>
      <div id="form-submit-button" class=" form-field form-button">
```

The register view function

This view enables users to create an account. A CreateProfileForm object is created and used in rendering the template. When the form is submitted the two password fields are checked for equivalency. If they do not match, the form is rejected. Otherwise, a User object is created from the form data and a password hash is calculated using the algorithm described in the Key Algorithms section. The database operation to insert a new user account is performed. If it fails due to the email already being in use, then the form is rejected. Otherwise, the account is stored in the database and the user is redirected to the login screen.

```
@app.route('/register', methods=["GET", "POST"])

def register():

  # Route for register page

  form = CreateProfileForm()

  if form.validate_on_submit():

    if form.password.data != form.repeat_password.data:

      flash("Passwords do not match", "form")

    # Create profile

    else:

      user = User(form.first_name.data, form.last_name.data, form.email.data,
form.password.data)

      user.password_hash = user.calculate_password_hash(user.plaintext_password)

      profile_added_successfully = add_profile_to_database(user)

      if not profile_added_successfully:

        flash("Email already in use", "form")

      else:

        return redirect("/login")

  return render_template("register.html", form=form)
```

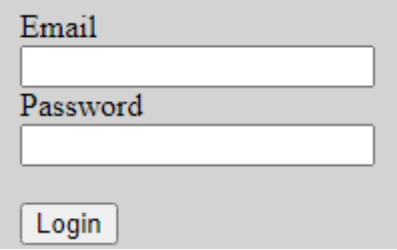
Logging in

Form to log in

```
class LoginForm(FlaskForm):

  email = EmailField("Email", validators=[DataRequired()])

  password = PasswordField("Password", validators=[DataRequired()])

  submit = SubmitField("Login")
```

This form is used to log in. The user submits a username and password, and then the view function for the login page will determine if that data can be used to log the user in.

Field name	Field Type	Validators	Purpose
email	Email	Data Required Valid email address	The user's email address. The input is checked to ensure that it is a valid email address
password	Password	Data Required	The user's password. Any value entered in this field is obscured
submit	Submit		Submitting the form



```
<div id="login-form-container" class="form">
  <form method="POST" id="login-form">
    <fieldset>
      {{form.hidden_tag()}}
      <div id="email-field" class="form-field string-field">
        {{form.email.label}}
        <br>
        {{form.email()}}
      </div>
      <div id="password-field" class="form-field password-field">
        {{form.password.label}}
        <br>
        {{form.password()}}
      </div>
      <br>
      <div id="form-submit-button" class=" form-field form-button">
      {{form.submit()}}
      </div>
    </fieldset>
  </form>
  {% with alerts = get_flashed_messages(with_categories=true) %}
    {% if alerts %}
      <div id="alerts">
        <ul>
          {% for type, content in alerts %}
            {% if type=="form" or type=="info"%}
              <li>
                {{ content }}
              </li>
            {% endif %}
          {% endfor %}
```

The login view function

This view function provides the login screen. A LoginForm object is created and passed to the template rendering function to be included in the returned HTML. When the form is submitted, a search is made for an account with the email address given in the form. If no account is found, then the form is rejected with an alert. Otherwise, a new User object is created from the information given in the form and the given password is hashed. The hashed form password is then compared to the stored hash. If they match, then the user's email is stored in the session and the client is redirected to the profile page. Otherwise, the form is rejected with an alert informing the user that their password is incorrect.

```
@app.route("/login", methods=["GET", "POST"])

def login():

  form = LoginForm()

  if form.validate_on_submit():

    # Find profile with given email

    email = form.email.data

    password = form.password.data

    stored_user = find_user_by_email(email)

    if not stored_user:

      flash("Invalid email", "form")

    else:

      login_user = User(first_name=stored_user.first_name, last_name=stored_user.last_name,
email=email,

                  plaintext_password=password)

      login_user.password_hash =
login_user.calculate_password_hash(login_user.plaintext_password)

      if login_user.password_hash == stored_user.password_hash:

        session["active_user_email"] = email

        return redirect("/profile")

      else:

        flash("Invalid password", "form")

  return render_template("login.html", form=form)
```

Logging out

The log out view function

This view function allows users to log out, although it doesn't have a HTML file associated with it. A link is included on the profile page to this page so that users can click that to log out. When a user

follows the link, the email address stored in the session that represents the currently logged in user is deleted. The client is then redirected back to the login page.

```
@app.route("/logout")

def logout():

  # Logout user

  if "active_user_email" in session:

    del session["active_user_email"]

  return redirect("/login")
```

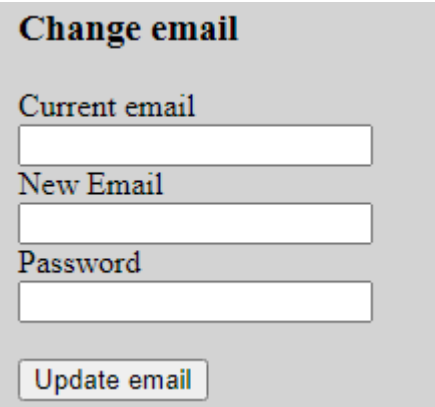
Displaying profiles

Form to change an accounts email address

```
class ChangeEmailForm(FlaskForm):

  current_email = EmailField("Current email", validators=[DataRequired()])

  new_email = EmailField("New Email", validators=[DataRequired()])

  password = PasswordField("Password", validators=[DataRequired()])

  submit = SubmitField("Update email")
```

This form enables users to submit a new email address. The user submits their current email and password, as well as a new email address. This data is validated in the view function for the profile page.

Field name	Field Type	Validators	Purpose
current_email	Email	Data Required Valid email address	The user's current email address. The input is checked to ensure that it is a valid email address
new_email	Email	Data Required Valid email address	The user's new email address. The input is checked to ensure that it is a valid email address
password	Password	Data Required	The user's password. This is checked to ensure that it is the correct password Any value entered in this field is obscured
submit	Submit		Submitting the form

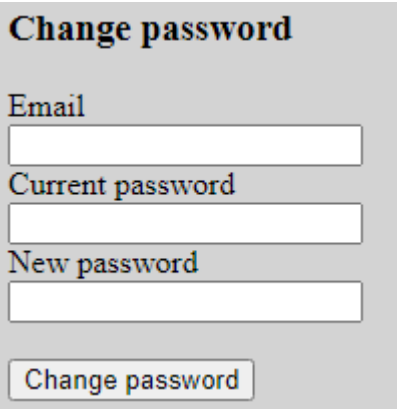


Form to change an account's password

```
class ChangePasswordForm(FlaskForm):

    email = EmailField("Email", validators=[DataRequired()])

    current_password = PasswordField("Current password", validators=[DataRequired()])

    new_password = PasswordField("New password", validators=[DataRequired()])

    submit = SubmitField("Change password")
```

This form enables users to change the password they use to log into their account. The user submits their email, their current password, and their new password. The behavior for this form is handled in the profile page view function.

Field name	Field Type	Validators	Purpose
email	Email	Data Required Valid email address	The user's email address. The input is checked to ensure that it is a valid email address
current_password	Password	Data Required	The user's current password. This is checked to ensure that it is the correct password Any value entered in this field is obscured
new_password	Password	Data Required	The user's new password. Any value entered in this field is obscured
submit	Submit		Submitting the form



The profile view function

When a user is logged in, this view function will return a page that displays information about their profile and their saved recipes, as well as options to change their email and password. If a user is not logged then the client is redirected to the login page.

Profile options

Two forms are used on this page, ChangeEmailForm and ChangePasswordForm. This causes problems due to the way that flask-wtf handles form validation. Each form object has a method called validate_on_submit. When a form is submitted and a POST request is made to the view function, this method returns true. However, this method returns true for all forms regardless of which form was submitted. As such, it is necessary to also check if each form object has values stored in its "data" attribute. This attribute is a dictionary, and if it is empty then Python will evaluate it as False when included in a logical statement. Although both validate_on_submit methods will return true when any form is submitted, only the submitted form will have a data attribute that contains values. As such, I use this attribute in the if statements to check which forms has been submitted.

Both forms follow a similar structure of checking whether a valid email and password has been used, and if so, then altering the user's record in the database and redirecting them to the logout page. The ChangeEmailForm validation does also check if the email the user is trying to change theirs to is already in use and rejects the form if it is.

Displaying recipes

To display recipes, the database module's find_user_saved_recipes function is used. This returns a list of recipe objects that is then passed to the template rendering function to be iterated over and included in the HTML.

```
@app.route("/profile", methods=["GET", "POST"])
def profile():
    change_email_form = ChangeEmailForm()
    change_password_form = ChangePasswordForm()
    if change_email_form.validate_on_submit() and change_email_form.data:
        user = find_user_by_email(change_email_form.current_email.data)
        form_password_hash = User.calculate_password_hash(change_email_form.password.data)
        if not user:
            flash("Invalid email", "change-email-form")
        else:
            if form_password_hash == user.password_hash:
                has_changed_email = change_user_email(user, new_email=change_email_form.new_email.data)
                if has_changed_email:
                    return redirect("/logout")
                elif not has_changed_email:
                    flash("Email already in use", "change-email-form")
            else:
                flash("Incorrect password", "change-email-form")
    elif change_password_form.validate_on_submit() and change_password_form.data:
        user = find_user_by_email(change_password_form.email.data)
        current_password_hash = User.calculate_password_hash(change_password_form.current_password.data)
        new_password_hash = User.calculate_password_hash(change_password_form.new_password.data)
        if not user:
            flash("Invalid email", "change-password-form")
        else:
            if current_password_hash == user.password_hash:
                change_user_password(user, new_password_hash)
                return redirect("/logout")
            else:
                flash("Incorrect password", "change-password-form")
    if "active_user_email" in session:
        # find currently logged in account
        user = find_user_by_email(session["active_user_email"])
        saved_recipes = find_user_saved_recipes(user)
        return render_template("profile.html", user=user, recipes=saved_recipes, change_email_form=change_email_form,
                               change_password_form=change_password_form)
    else:
        return redirect("login")
```

API

API info

A simple view function that displays info on how to use the API.

```
@app.route('/api')

def api():

  # Route for api info page

  return render_template("api.html")
```

API search

The API search view function provides a way for users to search for recipes without using the user interface. It uses the same functions as the search done through the web page but takes inputs as URL parameters and returns JSON data.

/api/search			
Parameter name	Type	Required	Purpose
ingredients	String	Yes	A space-separated list of ingredients
max_time	Int	No	The maximum total preparation time that returned recipes can have
sort_mode	String	No	How to sort the returned recipes Defaults to relevancy
limit	Int	No	The number of results to return after they have been sorted

If no ingredients value is given then the request is rejected with a 400 status code, indicating a bad request. Otherwise, the optional values are set to the given value or None if no value is given. A Query object is the constructed with these values and used to perform a search. The number of returned results is restricted to the limit if one is given, and then all the results objects are transformed into a dictionary of dictionaries. This dictionary then returned by the view function as JSON.

For example, the request:

```
/api/search?ingredients=”cheese potato leek”&limit=3
```

Will return the top 3 results sorted by relevancy as JSON:.

{"results": [{"ingredients": [{"name": "leek"},{"name": "salt ground black pepper"},{"name": "unsalted butter"}],"relevancy": 0.0761111692806109,"title": "Easy Sauteed Leeks","total_time": 26,"url": "https://www.allrecipes.com/recipe/8472762/easy-sauteed-leeks/","website": "allrecipes"},{"ingredients": [{"name": "chicken broth"},{"name": "cheddar cheese"},{"name": "salt ground black pepper"},{"name": "half-and-half"},{"name": "onion"},{"name": "garlic powder"},{"name": "fresh parsley"},{"name": "leek"},{"name": "potato"},{"name": "butter"}],"relevancy": 0.07571648837962841,"title": "Winter Leek and Potato Soup","total_time": 65,"url": "https://www.allrecipes.com/recipe/220779/winter-leek-and-potato-soup/","website": "allrecipes"},{"ingredients": [{"name": "salt pepper"},{"name": "chicken broth"},{"name": "butter"},{"name": "milk"},{"name": "leek"},{"name": "red new potato"}],"relevancy": 0.07129737431020067,"title": "Real Potato Leek Soup","total_time": 90,"url": "https://www.allrecipes.com/recipe/22932/real-potato-leek-soup/","website": "allrecipes"}]}

Key Algorithms

Password hashing

I created a hashing algorithm to hash passwords before storing them in the database, as storing plaintext passwords is a significant security risk. The function takes a string as an argument, then calculates the sum of the integer representations of each character raised to the power of each character's position in the string (starting at 1). Then it calculates the product of the integer representations of every character in the string. These two values are multiplied together and then divided the largest value that can be stored in a MySQL integer field. The remainder of this division (found using the modulo operator) is then returned as a string. This ensures that the output of the function will not be too large to store in the database.

The output of this hashing algorithm is dependent on the entire input and only the input, which makes it a reliable hashing algorithm as the same input will always produce the same output. Additionally, it is very difficult to calculate the input from only the output because of the modulo operation.

```
def calculate_password_hash(password: str = None) -> str:
    total = 0
    for i, character in enumerate(password):
        total += ord(character) ** (i + 1)
    product = 1
    for character in password:
        product *= ord(character)
    value = total * product
    hash = str(value % 2147483647)
    return hash
```

Relevancy scorer

The algorithm I chose to use in calculating a relevancy score for results uses TF-IDF (term frequency – inverse document frequency) vectorization to turn lists of ingredients into vectors and then calculates the size of the angle between the query vector and each result vector to produce a relevancy score. I chose this because it is the most reliable and accurate scorer of relevancy compared to the other algorithms I considered. The time-complexity of the algorithm does not a significant impact on the speed of the website

```
def score_recipes_by_relevancy_from_query(query: Query) -> list[Result]:

    """Returns an unsorted list of recipes with relevancy scores"""
```

Retrieving relevant recipes from the database

The function is passed a Query object to determine the search to perform on the database. The algorithm then selects any recipe that contains at least one of the ingredients that match an ingredient in the Query object's list of ingredients. The selected recipe records, each one combined with a comma-joined string of its ingredients, is returned as a Pandas Data Frame – a complex 2-dimensional array capable of performing very fast operations on its data. I will refer to the data frames as tables. The speed of data frames, as well as a wide variety of methods, makes them well

suited for data processing programs. The selection method is described in the Database Design section.

```
# Read the recipe data from the database

recipe_matrix = database.recipe_dataframe_from_query(query)
```

For this explanation, I will use the following simplified example data – in actuality more data would be used to make the search and would be returned in the data frame of recipes, such as the time they take to prepare and the website the recipe is from. However, the purpose of this example is to demonstrate the comparison of sets of ingredients and as such that data is not relevant.

Recipe ID	Ingredients
1	Salt, cucumber
2	Egg, salt, rice, milk, sugar
3	Egg, milk, salt, flour, vegetable oil
4	Sugar, rice, egg, milk, salt
5	Flour, rice, sugar, butter, salt

Query ID	Ingredients
6	Salt egg rice

Creating term frequency tables

The commas in the ingredients field in each recipe in the recipes table are replaced by spaces, and then the string is tokenized by the NLTK word tokenizer. This produces a list of tokens – individual words – from the names of all the ingredients in each recipe.

```
 # Copy the dataframe and turn the "ingredients" column into a list of tokens

 recipe_matrix_split_ingredients = recipe_matrix.copy()

 recipe_matrix_split_ingredients["ingredients"] =
recipe_matrix_split_ingredients["ingredients"].apply(

    lambda x: x.replace(",", " ")).apply(nltk.word_tokenize)
```

Recipe ID	Tokens
1	[“salt”, “cucumber”]
2	[“egg”, “salt”, “rice”, “milk”, “sugar”]
3	[“egg”, “milk”, “salt”, “flour”, “vegetable”, “oil”]
4	[“sugar”, “rice”, “egg”, “milk”, “salt”]

5	[“flour”, “rice”, “sugar”, “butter”, “salt”]

A collection of unique tokens is then created by creating a list of all the token and converting the list into a python set – a collection that can only contain unique values. This is used to create the columns of the query term frequency table.

```
# Create a list of all unique ingredients

all_ingredients = []

for recipe_ingredients in recipe_matrix_split_ingredients["ingredients"].tolist():

    all_ingredients += recipe_ingredients

unique_ingredients = list(set(all_ingredients))
```

In this case the collection would contain the following:

(“salt”, “cucumber”, “egg”, “rice”, “milk”, “sugar”, “flour”, “vegetable”, “oil”, “butter”)

The tokens field of each recipe is converted into a dictionary of the names of tokens and the number of times they appear in each recipe.

```
# Turn each list of tokens in the dataframe into a dictionary of tokens and how many times they appear in the list

recipe_ingredient_counts = recipe_matrix_split_ingredients["ingredients"].apply(

    lambda x: {i: x.count(i) for i in set(x)})
```

Recipe ID	Tokens
1	{“salt”: 1, “cucumber: 1”}
Etc…	

Each dictionary field is then transformed into a Pandas series – a 1 dimensional array where each column represents a token, and the values are the number of occurrences of that token in the recipe. This is applied across the recipe table and produces a term frequency table.

```
# Turn each of the dictionaries in the dataframe into a series

# Each recipe row has a column for every token, with the value being the count of that token in the recipe

recipes_term_frequency_matrix = recipe_ingredient_counts.apply(pd.Series)

recipes_term_frequency_matrix.fillna(0, inplace=True)  # Replace NaNs with 0
```

Recipe ID	Salt	Cucumber	Egg	Rice	Milk	Sugar	Flour	Vegetable	Oil	Butter
1	1	1	0	0	0	0	0	0	0	0
2	1	0	1	1	1	1	0	0	0	0
3	1	0	1	0	1	0	1	1	1	0
4	1	0	1	1	1	1	0	0	0	0
5	1	0	0	1	0	1	1	0	0	1

The same process is applied to the query table.

```
# Count each of the terms in the query text

query_counts = {i: query.cleaned_tokens.count(i) for i in set(query.cleaned_tokens)}

# Create a term frequency table using the term counts of the query and the set of terms that appear in the recipes

query_term_frequency = pd.DataFrame(query_counts, columns=unique_ingredients, index=[0])

query_term_frequency.fillna(0, inplace=True)
```

Query ID	Salt	Cucumber	Egg	Rice	Milk	Sugar	Flour	Vegetable	Oil	Butter
6	1	0	1	1	0	0	0	0	0	0

Vectorizing the texts

Before comparing sets of ingredients, I first had to turn them into vectors where each component of the vector represented an ingredient, and the magnitude of each component was the TF-IDF score of that ingredient. The TF-IDF score of an ingredient is the product of that ingredients term frequency (TF) and inverse document frequency (IDF). The term frequency of an ingredient is the number of times that ingredient appears in a recipe – as is calculated above, and the document frequency is the number of times that ingredient appears in a recipe (in the set of recipes retrieved from the database described above). The inverse document frequency represents the relevancy weighting of an ingredient, and is calculated using the below formula:[12]

$$idf = \log_2 \frac{n}{df}$$

Where n is the total number of recipes and df is the document frequency of the term.

The inverse document frequency is higher for terms that appear rarely, and lower for terms that appear often. If a term appears in all documents(recipes) then it will have an IDF score of $\log_2 \frac{n}{n} = 0$. This resolves the issue of common terms having equal weighting in calculating relevancy described in my analysis.

Applying this formula to every column, excluding the ID column, of the data frame – in other words every unique token – produces a dictionary of tokens and their IDF scores – shown below in a table to 3 decimal places.

[12] Ramadhan, L. (2021, February 4). TF-IDF Simplified. Medium. https://towardsdatascience.com/tf-idf-simplified-aba19d5f5530

```
# Calculate the inverse document frequency for each term i
number_of_recipes = len(recipe_matrix.index)
inverse_document_frequency = {}
for i in recipes_term_frequency_matrix:
    # for every term column
    # Find the document frequency of the term i
    # df = the length of the series of the term column where the cell > 0
    document_frequency_i =
len(recipes_term_frequency_matrix[recipes_term_frequency_matrix[i] > 0])
    # Calculate the idf for i
    inverse_document_frequency_i = math.log(number_of_recipes / document_frequency_i, 2)
    inverse_document_frequency[i] = inverse_document_frequency_i
```

Salt	Cucumber	Egg	Rice	Milk	Sugar	Flour	Vegetable	Oil	Butter
0	2.322	0.737	0.737	0.737	0.737	1.322	2.322	2.322	2.322

Each token's value in each row of the recipe term frequency table is then multiplied by its IDF score to produce the recipe TF-IDF table.

```
# Calculate the tf-idf matrix by multiplying each row of the term frequency matrix by the idf values
# This produces the dataframe of recipe vectors
recipe_tf_idf_matrix = recipes_term_frequency_matrix.mul(inverse_document_frequency)
```

Recipe ID	Salt	Cucumber	Egg	Rice	Milk	Sugar	Flour	Vegetable	Oil	Butter
1	0	2.322	0	0	0	0	0	0	0	0
2	0	0	0.737	0.737	0.737	0.737	0	0	0	0
3	0	0	0.737	0	0.737	0	1.322	2.322	2.322	0
4	0	0	0.737	0.737	0.737	0.737	0	0	0	0
5	0	0	0	0.737	0	0.737	1.322	0	0	2.322

The same IDF scores are applied to the query term frequency table using the above method.

```
# Create the query tf-idf matrix by multiplying the query term frequency matrix by the idf values created earlier

# This is the vectorized query

query_tf_idf = query_term_frequency.mul(inverse_document_frequency)

query_tf_idf_series = query_tf_idf.iloc[0]  # The query tf-idf needs to be stored as a series for the dot product
```

Query ID	Salt	Cucumber	Egg	Rice	Milk	Sugar	Flour	Vegetable	Oil	Butter
6	0	0	0.737	0.737	0	0	0	0	0	0

The TF-IDF tables of the recipes and the query are vectorized representations of the original lists of ingredients, where each component represents a token and the value of that component is its TF-IDF value.

Comparing vectors

To find the similarity between the query vector and a given recipe vector I used the cosine similarity formula to find the size of the angles between two vectors.

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \times |\vec{B}|}$$

I used a modified version to calculate the relevancy of a recipe vector B to the query vector A.

$$relevancy = 1 - \frac{\cos^{-1} \frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \times |\vec{B}|}}{0.5\pi}$$

This expresses the inverse-size of the angle between the vectors as a value in the range of 0 to 1. The value of the angle is divided by half pi and subtracted from 1 to bound the results between 0 and 1, and as such as the size of the angle increases, the relevancy score decreases - a larger angle corresponds to a less relevant result.

To calculate the dot-product of the vectors the inbuilt pandas DataFrame.dot method is used, shown in the example to 3 decimal places. I used this inbuilt method due to its speed, as the tables often have several thousand columns and rows.

```
# Calculate the dot products of each recipe vector and the query vector

dot_products = recipes_term_frequency_matrix.dot(query_tf_idf_series)
```

Recipe ID	Dot product with query vector
1	0
2	1.086
3	0.543
4	1.086
5	0.543

To calculate the magnitudes of the vectors a custom algorithm is used.

```
def magnitude(vector: list[float]) -> float:
    """Calculate the magnitude of a vector, ie: [x, y] -> sqrt(x^2 + y^2)"""
    # List comprehensions are faster than for loops, necessary given how slow this runs
    return math.sqrt(sum([math.pow(i, 2) for i in vector]))
```

A list comprehension is used to produce a list of the squares of each component value of the vector. This is then summed and square rooted. List comprehensions are slightly faster than for loops in python, and as the magnitude functions is being run on thousands of vectors with thousands of components the speed difference is significant.

ID	Magnitude
1	2.322
2	1.474
3	3.690
4	1.474
5	2.868
6 (Query)	1.042

The magnitudes of each recipe are then multiplied by the magnitude of the query to produce the denominator for each recipe's relevancy equation.

```
    # Multiply the magnitudes of each recipe vector by the query vector to produce the denominator
of the angle equation
    equation_denominator = recipe_magnitudes * query_magnitude
```

The dot products for each recipe are then divided by the denominator for each recipe to give the cosine of the angle between the vectors.

```
    # Divide the dot products by the product of the magnitudes to find the cosine of the angle
between the vectors
    vector_angles = dot_products.divide(equation_denominator)
```

The final relevancy scores for the example are below and are calculated with the math library.

```
    # Calculate the angle between the vectors and bound it within positive space (0 to 1)
    vector_similarities = vector_angles.apply(lambda x: 1 - (math.acos(x) / (0.5*math.pi)))
    vector_similarities.fillna(0, inplace=True)
```

Recipe ID	Relevancy
1	0
2	0.527
3	0.094
4	0.527

5	0.349

As can be seen, the most relevant recipes are ID 2 and 4. Recipe 1 had no relevancy at all, as the only intersecting token was “salt”, which had an IDF score of 0 as it appeared in all recipes. Terms such as “oil” had a term frequency of 0 in the query vector, so had no bearing at all on the relevancy score.

The original recipes from the database selected earlier are then used to create Result objects, as well as each recipe’s relevancy score. These are returned as an unsorted list of Result objects.

```
recipes = []
for i in range(number_of_recipes):
    relevancy = vector_similarities.iloc[i].item()
    recipe_data = recipe_matrix.loc[vector_similarities.index[i]]
    ingredients = [Ingredient(ingredient) for ingredient in recipe_data["ingredients"].split(",")]
    recipe = Result(recipe_data["title"],
                    ingredients,
                    recipe_data["time"].item(),
                    recipe_data["url"],
                    recipe_data["website"],
                    relevancy
                    )
    recipes.append(recipe)
return recipes
```

The method .item() is used to obtain the relevancy score and the time as floats because these values are stored in the dataframe using the Numpy float32 datatype. Whilst this behaves largely like a native Python float, it cannot be used directly in the Result object as the Result must be JSON-serializable and the Numpy float32 objects are not, whereas the native Python float is.

Finding results

This function will return a list of Result objects that match a given query. If the results are to be sorted by relevancy, then the Relevancy Scorer algorithm is used and the results are then sorted by descending relevancy score. If a different sort mode is used, a different selection function is used. The relevancy score function is relatively slow, and as the recipes are not being sorted by relevancy there is no need to calculate the relevancy score. Instead, the recipes are selected from the database directly and returned as a list of Result objects. The list can then be sorted by the given sort mode.

```
def find_results(query: Query) -> list[Result]:
    if query.sort_mode == "relevancy":
        results = score_recipes_by_relevancy_from_query(query)
        results = quick_sort(results, query.sort_mode)
        results.reverse()
    else:
        results = database.select_recipes_with_query(query)
        results = quick_sort(results, query.sort_mode)
    return results
```

Sorting results

To sort the results of a search I implemented a quick sort algorithm. The function takes as arguments a list of Result objects and the sorting mode.

To determine what attribute to sort the objects by an embedded function is used. This function takes as an argument a Result object. It uses a switch statement to determine what attribute of the object to return – this is the attribute that is used to compare and sort the objects. The expression used in the switch is the sort mode, accessed from the larger scope of the enclosing function. The default case will raise a ValueError, as it indicates that an invalid sort mode has been included in the query.

I used this structure of an embedded function to allow me to avoid repeating my code when comparing objects. The embedded function can access the sort_mode variable without it being an argument because the function is within the local scope of the sorting function. Overall it enables me to sort Result objects by a variety of attributes without repeating code.

```
def quick_sort(results: list[Result], sort_mode: str) -> list[Result]:


  def get_sorting_value(item: Result):
    match sort_mode:
      case "relevancy":
        return item.relevancy
      case "title":
        return item.title
      case "total_time":
        return item.total_time
      case _:
        raise ValueError(f"Invalid sort mode: {sort_mode}")
```

The quick sort function is a recursive function. As such, I began by identifying the base cases – the inputs that will not result in a recursive call. If the input list only contains 1 or 0 items, then it is already sorted and can just be returned as is.

```
  length = len(results)


  # Base case
  if length < 2:
    return results
```

If the list contains two items, then it may or may not be sorted. This can easily be checked by comparing the two items and checking if they are in order. If they are, then the list is returned. Otherwise, the items are swapped and then the list is returned. The values are compared using the function described above.

```
  # Simplest case
  if length == 2:
    if get_sorting_value(results[0]) > get_sorting_value(results[1]):
      results[0], results[1] = results[1], results[0]
    return results
```

If the list has 3 or more items, then that is the recursive case. A randomly selected “pivot” value is selected, and all values that are less than or equal to the pivot are placed on the “left” of the pivot. All the other values are placed on the “right”. The left and right lists are then passed to the quick sort function to be sorted, and are then combined with the pivot to produce the sorted list.

```
for i in range(length):
    if i != pivot:
        r = get_sorting_value(results[i])
        if r > pivot_value:
            right.append(results[i])
        else:
            left.append(results[i])


return quick_sort(left, sort_mode) + [results[pivot]] + quick_sort(right, sort_mode)
```

Finding sequences of nouns and adjectives

When processing a recipe's ingredients, I needed to identify the relevant information from the webpage. As part of this, I constructed a function to find sequences of nouns and adjectives. Once the tokens have been lemmatised and tagged, they are passed to this function. The function removes any token contained within a list of "ignore words" – words that need to be manually removed to improve the effectiveness of the processing, such as "teaspoon". Then, it iterates over every token-tag tuple in the list of tagged tokens. If the tag of a token means that it is a noun, proper noun or adjective then the tuple is appended to a temporary list. When the iteration encounters a token with a tuple that is not a noun, proper noun or adjective then the tokens in the temporary list are joined with spaces and added to a list of sequences. The temporary list is then cleared. Once the iteration is complete a final check is performed to see if there are still values in the temporary list. This resolves the issue of the last token in a list being added to the temporary list and subsequently the sequence never being added to the list of sequences.

```
def find_phrases(tagged_tokens: list) -> list:
    """Find consecutive sequences of nouns and adjectives"""
    noun_adjective_tags = ["NN", "NNP", "JJ"]  # NLTK part-of-speech tags: Noun, Proper Noun,
Adjective
    phrases = []
    current_phrase_tokens = []

    # Clean tokens of commonly mistaken words
    with open("ignore_words.txt", "r") as file:
        ignore_words = file.read().splitlines()
    tagged_tokens = [token for token in tagged_tokens if token[0] not in ignore_words]

    for tagged_token in tagged_tokens:
        if tagged_token[1] in noun_adjective_tags:
            current_phrase_tokens.append(tagged_token)
        elif current_phrase_tokens:
            phrases.append(" ".join([token[0] for token in current_phrase_tokens]))
            current_phrase_tokens.clear()
    else:
        # In case the last token is a noun or adjective, then the phrase needs to be added
        if current_phrase_tokens:
            phrases.append(" ".join([token[0] for token in current_phrase_tokens]))

    return phrases
```

Technical Solution

File structure

I divided my code into two key folders, as well as some files in the top directory. Files within the "app" folder form the web server and are run from the command line with Flask as part of the server. Files within the "web_scraping" folder are used in scraping recipes and are run directly through a Python interpreter.

Note – I have not included the data files created by web scraping as they are too large to print.

Areas of interest

Function/Class	File	Page	Skills demonstrated
create_recipe_select_sql	app/database.py	81	Cross-table parameterised SQL Aggregate SQL functions
select_recipes_with_query	app/database.py	80	Complex OOP – aggregation
user_save_recipe	app/database.py	82	Parameterised SQL and linking tables
find_user_saved_recipes	app/database.py	82	Cross-table parameterised SQL
Recipe, Result	app/models.py	84, 85	Complex OOP – inheritance Dictionaries
User.calculate_password_hash	app/models.py	85	Hashing
WebScraper	app/models.py	86	OOP – Abstract class
score_recipes_by_relevancy_from_query	app/reccomender.py	86	Advanced matrix operations Complex algorithm – relevancy calculation
quick_sort	app/reccomender.py	88	Quick sort Recursion
get_sorting_value	app/reccomender.py	88	Switch statement
recommend	app/routes.py	80	Server-side scripting
register	app/routes.py	80	POST request handling
api_search	app/routes.py	92	Server-side scripting

			Dynamic generation of objects Parameterised API returning JSON
batch_insert_recipes	web_scraping/database_insertion.py	101	Reading from files Cross-table parameterised SQL Simple algorithm
batch_scrape_pages	web_scraping/mass_scraping.py	102	Writing to files Simple algorithm
find_phrases, identify_ingredient	web_scraping/natural_language_processing.py	104	Natural language processing List operations
AllRecipes, SimplyRecipes	web_scraping/scrapers.py	105-6	Web scraping HTML processing Pattern matching

Files

config.py

```
SECRET_KEY = 'key'
DATABASE_NAME = "database"
DATABASE_HOST = "localhost"
DATABASE_PASSWORD = "password"
DATABASE_USERNAME = "root"
```

NEAProject.py

```
from app import app
```

requirements.txt

```
beautifulsoup4==4.11.1
bs4==0.0.1
certifi==2022.12.7
charset-normalizer==2.1.1
click==8.1.3
colorama==0.4.6
Flask==2.2.2
Flask-MySQLdb==1.0.1
Flask-WTF==1.0.1
idna==3.4
importlib-metadata==6.0.0
itsdangerous==2.1.2
Jinja2==3.1.2
joblib==1.2.0
MarkupSafe==2.1.1
mysql-connector-python==8.0.32
mysqlclient==2.1.1
nltk==3.8.1
numpy==1.24.1
pandas==1.5.3
protobuf==3.20.3
python-dateutil==2.8.2
pytz==2022.7.1
regex==2022.10.31
requests==2.28.1
six==1.16.0
soupsieve==2.3.2.post1
tqdm==4.64.1
urllib3==1.26.14
Werkzeug==2.2.2
WTForms==3.0.1
zipp==3.11.0
```

app/database.py

```
from flask import Flask
from app import db
from app.models import Query, Result, Ingredient, User, Recipe
import pandas as pd
import MySQLdb
from typing import Optional


def select_recipes_with_query(query: Query) -> list[Result]:
    """Find all recipes with at least one ingredient that matches a token
```

```
in the query"""
    # open database connection
    cursor = db.connection.cursor()
    find_recipes_sql = create_recipe_select_sql(query)

    cursor.execute(find_recipes_sql)
    results = cursor.fetchall()
    recipes = []
    for result in results:
        ingredients = []
        for ingredient in result[5].split(","):
            ingredients.append(Ingredient(ingredient))
        recipes.append(Result(result[1], ingredients, result[2], result[3],
result[4]))
    return recipes


def recipe_dataframe_from_query(query: Query) -> pd.DataFrame:
    """Pandas dataframe from results of query"""
    sql = create_recipe_select_sql(query)
    connection = db.connect
    dataframe = pd.read_sql(sql, connection).set_index("recipe_id")
    return dataframe


def create_recipe_select_sql(query: Query) -> str:
    """Creates a sql query to select recipes that contain any ingredient
that matches any given in the user query"""
    # Create pattern to match ingredient names. Join by | (or operator)
allows any ingredient token to match
    # Two different queries for limiting by total time and not
    ingredients_regex = "|".join(query.cleaned_tokens)

    if query.max_time is None:
        find_recipes_sql = f"SELECT recipes.*,
GROUP_CONCAT(ingredients.name SEPARATOR ',') AS ingredients FROM recipes,
ingredients, has_ingredient WHERE
recipes.recipe_id=has_ingredient.recipe_id AND
ingredients.ingredient_id=has_ingredient.ingredient_id AND
recipes.recipe_id IN (SELECT has_ingredient.recipe_id FROM ingredients,
has_ingredient WHERE has_ingredient.ingredient_id=ingredients.ingredient_id
AND ingredients.ingredient_id = ANY("SELECT ingredient_id FROM ingredients
WHERE name REGEXP \"{ingredients_regex}\")) GROUP BY recipes.recipe_id,
recipes.title;"
    else:
        find_recipes_sql = f"SELECT recipes.*,
GROUP_CONCAT(ingredients.name SEPARATOR ',') AS ingredients FROM recipes,
ingredients, has_ingredient WHERE
recipes.recipe_id=has_ingredient.recipe_id AND
ingredients.ingredient_id=has_ingredient.ingredient_id AND
recipes.recipe_id IN (SELECT has_ingredient.recipe_id FROM ingredients,
has_ingredient WHERE has_ingredient.ingredient_id=ingredients.ingredient_id
AND ingredients.ingredient_id = ANY(SELECT ingredient_id FROM ingredients
WHERE name REGEXP \"{ingredients_regex}\")) AND recipes.time <=
query.max_time} GROUP BY recipes.recipe_id, recipes.title;"
    return find_recipes_sql


def add_profile_to_database(profile: User) -> bool:
    cursor = db.connection.cursor()
    values = (profile.first_name, profile.last_name, profile.email,
```

```
profile.password_hash)
    sql = "INSERT INTO users (first_name, last_name, email, password)
VALUES (%s, %s, %s, %s);"
    try:
        cursor.execute(sql, values)
    except MySQLdb.IntegrityError:
        # Throws when trying to create a profile with an already used email
        return False
    db.connection.commit()
    return True


def find_user_by_email(email: str) -> Optional[User]:
    cursor = db.connection.cursor()
    sql = f'SELECT * FROM users WHERE email="{email}";'
    cursor.execute(sql)
    if cursor.rowcount == 0:
        # no results found
        return None
    else:
        result = cursor.fetchone()
        user = User(user_id=result[0], first_name=result[1],
last_name=result[2], email=result[3],
                    password_hash=result[4])
        return user


def user_save_recipe(recipe_title: str, user: User):
    # Get recipe id by title
    cursor = db.connection.cursor()
    recipe_select = f"SELECT recipe_id FROM recipes WHERE
title='{recipe_title}';"
    cursor.execute(recipe_select)
    recipe_id = cursor.fetchone()[0]

    insert_sql = f"INSERT INTO saved_recipe (user_id, recipe_id) VALUES
(%s, %s);"
    values = (user.user_id, recipe_id)
    cursor.execute(insert_sql, values)
    db.connection.commit()


def find_user_saved_recipes(user: User) -> list[Recipe]:
    cursor = db.connection.cursor()
    sql = f"SELECT * FROM recipes JOIN saved_recipe ON
recipes.recipe_id=saved_recipe.recipe_id WHERE" \
          f" user_id={user.user_id};"
    cursor.execute(sql)
    results = cursor.fetchall()
    recipes = []
    for result in results:
        recipes.append(Recipe(title=result[1], ingredients=[],
total_time=result[2], url=result[3], website=result[4]))
    return recipes


def change_user_email(user: User, new_email: str) -> bool:
    if find_user_by_email(new_email) is not None:
        # If email already in use
        return False
    cursor = db.connection.cursor()
```

```
    sql = f"UPDATE users SET email='{new_email}' WHERE
user_id={user.user_id};"
    cursor.execute(sql)
    db.connection.commit()
    return True


def change_user_password(user: User, new_password_hash: str):
    cursor = db.connection.cursor()
    sql = f"UPDATE users SET password='{new_password_hash}' WHERE
user_id={user.user_id};"
    cursor.execute(sql)
    db.connection.commit()
```

app/forms.py

```
from flask_wtf import FlaskForm
from wtforms import StringField, SelectField, IntegerField, SubmitField,
EmailField, PasswordField
from wtforms.validators import NumberRange, DataRequired, Optional
from wtforms import ValidationError


class RecommenderForm(FlaskForm):
    ingredients = StringField("Ingredients", validators=[DataRequired()])
    max_time = IntegerField("Maximum Time", validators=[Optional(),
NumberRange(min=1)])
    sort_mode = SelectField("Sort by",
                            choices=[('relevancy', 'Relevancy'), ('title',
'Title'), ('total_time', 'Time to cook')])
    limit = IntegerField("Limit number of results", validators=[Optional(),
NumberRange(min=1)])
    submit = SubmitField("Find Recipes")


class CreateProfileForm(FlaskForm):
    first_name = StringField("First name", validators=[DataRequired()])
    last_name = StringField("Last name", validators=[DataRequired()])
    email = EmailField("Email", validators=[DataRequired()])
    password = PasswordField("Password", validators=[DataRequired()])
    repeat_password = PasswordField("Repeat Password",
validators=[DataRequired()])
    submit = SubmitField("Create profile")



class LoginForm(FlaskForm):
    email = EmailField("Email", validators=[DataRequired()])
    password = PasswordField("Password", validators=[DataRequired()])
    submit = SubmitField("Login")


class SaveRecipeForm(FlaskForm):
    # Provides a csrf token to the save recipe button
    pass


class ChangeEmailForm(FlaskForm):
    current_email = EmailField("Current email",
validators=[DataRequired()])
    new_email = EmailField("New Email", validators=[DataRequired()])
```

```
    password = PasswordField("Password", validators=[DataRequired()])
    submit = SubmitField("Update email")


class ChangePasswordForm(FlaskForm):
    email = EmailField("Email", validators=[DataRequired()])
    current_password = PasswordField("Current password",
validators=[DataRequired()])
    new_password = PasswordField("New password",
validators=[DataRequired()])
    submit = SubmitField("Change password")
```

app/models.py

```
from abc import ABC, abstractmethod
import requests
from bs4 import BeautifulSoup
from nltk.stem import WordNetLemmatizer
from nltk import word_tokenize, pos_tag


class Ingredient:
    def __init__(self, name: str):
        self.name = name

    def as_dict(self):
        return {"name": self.name}


class Recipe:
    def __init__(self, title: str, ingredients: list[Ingredient],
total_time: int, url: str, website: str):
        self.title = title
        self.ingredients = ingredients
        self.total_time = total_time
        self.url = url
        self.website = website

    def as_dict(self):
        return {
            "title": self.title,
            "ingredients": [i.as_dict() for i in self.ingredients],
            "total_time": self.total_time,
            "url": self.url,
            "website": self.website
        }

    @property
    def formatted_time(self) -> str:
        """Returns the recipe's total time formatted in hours and
minutes"""
        if self.total_time == -1:
            return "N/A"
        else:
            hours = self.total_time // 60
            minutes = self.total_time % 60

            if hours == 0:
                return f"{minutes} minutes"
            elif hours == 1:
                return f"1 hr {minutes} mins"
```

```
            else:
                return f"{hours} hrs {minutes} mins"


class Result(Recipe):
    def __init__(self, title: str, ingredients: list[Ingredient],
total_time: int, url: str, website: str,
                 relevancy: float = None):
        super().__init__(title, ingredients, total_time, url, website)
        self.relevancy = relevancy

    def as_dict(self):
        return {
            "title": self.title,
            "ingredients": [i.as_dict() for i in self.ingredients],
            "total_time": self.total_time,
            "url": self.url,
            "website": self.website,
            "relevancy": self.relevancy
        }


class User:
    def __init__(self, first_name: str, last_name: str, email: str,
plaintext_password: str = None,
                 password_hash: str = None, user_id: int = None):
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.plaintext_password = plaintext_password
        self.password_hash = password_hash
        self.user_id = user_id

    @staticmethod
    def calculate_password_hash(password: str = None) -> str:
        total = 0
        for i, character in enumerate(password):
        # Sum of the numerical representation of each character raised to
the power of its position in the string
            total += ord(character) ** (i + 1)
        product = 1
        for character in password:
            product *= ord(character)
        value = total * product
        hash = str(value % 2147483647) # Limit values to between 0 and max
value that can be stored in signed 32 bit integer - also a prime
        return hash


class Query:
    def __init__(self, raw_ingredients: str, sort_mode: str = "relevancy",
max_time: int = None, limit: int = None):
        self.raw_ingredients = raw_ingredients
        self.cleaned_tokens = self.clean_and_tokenize_text(raw_ingredients)
        self.sort_mode = sort_mode
        self.max_time = max_time
        self.results: list[Result] = []
        self.limit = limit

    @staticmethod
    def clean_and_tokenize_text(text: str) -> list[str]:
```

```
        """Return a list of cleaned tokens"""
        wnl = WordNetLemmatizer()
        text = text.lower()
        tokens = word_tokenize(text)
        lemmatized_tokens = [wnl.lemmatize(token) for token in tokens]
        tagged_tokens = pos_tag(lemmatized_tokens)
        clean_tokens = [token[0] for token in tagged_tokens if token[1] in
["NN", "NNP", "JJ"]]
        return clean_tokens


class WebScraper(ABC):
    @staticmethod
    def make_soup(url: str) -> BeautifulSoup:
        """Creates a BeautifulSoup Soup from a given url"""
        page = requests.get(url).text
        soup = BeautifulSoup(page, "html.parser")
        return soup

    @staticmethod
    @abstractmethod
    def scrape_page(url: str) -> Recipe:
        """Scrape a specific page and return a recipe object"""
        pass

    @staticmethod
    @abstractmethod
    def find_recipe_links(url: str) -> list[str]:
        """Find links to recipe pages on a given page"""
        pass

    @staticmethod
    @abstractmethod
    def find_links_to_scrape(recipe_site: str):
        """Create a text file of links to recipe pages"""
        pass
```

app/reccomender.py

```
import pandas as pd
import nltk
import math
import random
from app import database
from app.models import Query, Result, Ingredient

pd.set_option("display.max_columns", None)
pd.set_option("display.max_rows", None)


def magnitude(vector: list[float]) -> float:
    """Calculate the magnitude of a vector, ie: [x, y] -> sqrt(x^2 +
y^2)"""
    # List comprehensions are faster than for loops, necessary given how
slow this runs
    return math.sqrt(sum([math.pow(i, 2) for i in vector]))


def score_recipes_by_relevancy_from_query(query: Query) -> list[Result]:
    """Returns an unsorted list of recipes with relevancy scores"""

    # Read the recipe data from the database
```

```
    recipe_matrix = database.recipe_dataframe_from_query(query)

    # Copy the dataframe and turn the "ingredients" column into a list of
tokens
    recipe_matrix_split_ingredients = recipe_matrix.copy()
    recipe_matrix_split_ingredients["ingredients"] =
recipe_matrix_split_ingredients["ingredients"].apply(
        lambda x: x.replace(",", " ")).apply(nltk.word_tokenize)

    # Create a list of all unique ingredients
    all_ingredients = []
    for recipe_ingredients in
recipe_matrix_split_ingredients["ingredients"].tolist():
        all_ingredients += recipe_ingredients
    unique_ingredients = list(set(all_ingredients))

    # Turn each list of tokens in the dataframe into a dictionary of tokens
and how many times they appear in the list
    recipe_ingredient_counts =
recipe_matrix_split_ingredients["ingredients"].apply(
        lambda x: {i: x.count(i) for i in set(x)})

    # Turn each of the dictionaries in the dataframe into a series
    # Each recipe row has a column for every token, with the value being
the count of that token in the recipe
    recipes_term_frequency_matrix =
recipe_ingredient_counts.apply(pd.Series)
    recipes_term_frequency_matrix.fillna(0, inplace=True)  # Replace NaNs
with 0

    # Calculate the inverse document frequency for each term i
    number_of_recipes = len(recipe_matrix.index)
    inverse_document_frequency = {}
    for i in recipes_term_frequency_matrix:
        # for every term column
        # Find the document frequency of the term i
        # df = the length of the series of the term column where the cell >
0
        document_frequency_i =
len(recipes_term_frequency_matrix[recipes_term_frequency_matrix[i] > 0])
        # Calculate the idf for i
        inverse_document_frequency_i = math.log(number_of_recipes /
document_frequency_i, 2)
        inverse_document_frequency[i] = inverse_document_frequency_i
    # Calculate the tf-idf matrix by multiplying each row of the term
frequency matrix by the idf values
    # This produces the dataframe of recipe vectors
    recipe_tf_idf_matrix =
recipes_term_frequency_matrix.mul(inverse_document_frequency)

    # Count each of the terms in the query text
    query_counts = {i: query.cleaned_tokens.count(i) for i in
set(query.cleaned_tokens)}
    # Create a term frequency table using the term counts of the query and
the set of terms that appear in the recipes
    query_term_frequency = pd.DataFrame(query_counts,
columns=unique_ingredients, index=[0])
    query_term_frequency.fillna(0, inplace=True)

    # Create the query tf-idf matrix by multiplying the query term
frequency matrix by the idf values created earlier
```

```
    # This is the vectorized query
    query_tf_idf = query_term_frequency.mul(inverse_document_frequency)
    query_tf_idf_series = query_tf_idf.iloc[0]  # The query tf-idf needs to
be stored as a series for the dot product

    # Calculate the magnitude of each recipe vector
    recipe_magnitudes = recipe_tf_idf_matrix.apply(lambda x: magnitude(x),
axis=1)
    # Calculate the magnitude of the query vector
    query_magnitude = magnitude(query_tf_idf_series)
    # Multiply the magnitudes of each recipe vector by the query vector to
produce the denominator of the angle equation
    equation_denominator = recipe_magnitudes * query_magnitude

    # Calculate the dot products of each recipe vector and the query vector
    dot_products = recipes_term_frequency_matrix.dot(query_tf_idf_series)
    # Divide the dot products by the product of the magnitudes to find the
cosine of the angle between the vectors
    vector_angles = dot_products.divide(equation_denominator)

    # Calculate the angle between the vectors and bound it within positive
space (0 to 1)
    vector_similarities = vector_angles.apply(lambda x: 1 - (math.acos(x) /
(0.5*math.pi)))
    vector_similarities.fillna(0, inplace=True)

    recipes = []
    for i in range(number_of_recipes):
        relevancy = vector_similarities.iloc[i].item()
        recipe_data = recipe_matrix.loc[vector_similarities.index[i]]
        ingredients = [Ingredient(ingredient) for ingredient in
recipe_data["ingredients"].split(",")]
        recipe = Result(recipe_data["title"],
                        ingredients,
                        recipe_data["time"].item(),
                        recipe_data["url"],
                        recipe_data["website"],
                        relevancy
                        )
        recipes.append(recipe)
    return recipes


def quick_sort(results: list[Result], sort_mode: str) -> list[Result]:

    def get_sorting_value(item: Result):
        match sort_mode:
            case "relevancy":
                return item.relevancy
            case "title":
                return item.title
            case "total_time":
                return item.total_time
            case _:
                raise ValueError(f"Invalid sort mode: {sort_mode}")

    length = len(results)

    # Base case
    if length < 2:
        return results
```

```
    # Simplest case
    if length == 2:
        if get_sorting_value(results[0]) > get_sorting_value(results[1]):
            results[0], results[1] = results[1], results[0]
        return results

    # Recursive case
    pivot = random.randint(0, length - 1)
    pivot_value = get_sorting_value(results[pivot])

    left = []
    right = []

    for i in range(length):
        if i != pivot:
            r = get_sorting_value(results[i])
            if r > pivot_value:
                right.append(results[i])
            else:
                left.append(results[i])

    return quick_sort(left, sort_mode) + [results[pivot]] +
quick_sort(right, sort_mode)


def find_results(query: Query) -> list[Result]:
    if query.sort_mode == "relevancy":
        results = score_recipes_by_relevancy_from_query(query)
        results = quick_sort(results, query.sort_mode)
        results.reverse()
    else:
        results = database.select_recipes_with_query(query)
        results = quick_sort(results, query.sort_mode)
    return results
```

app/routes.py

```
from app import app, recommender
from app.forms import RecommenderForm, CreateProfileForm, LoginForm,
SaveRecipeForm, ChangeEmailForm, ChangePasswordForm
from flask import render_template, session, redirect, url_for, request,
flash
from app.models import Query, User
from app.database import add_profile_to_database, find_user_by_email,
user_save_recipe, find_user_saved_recipes, \
    change_user_email, change_user_password


@app.route('/', methods=["GET", "POST"])
def index():
    """Home page of the site, displaying the recommender form"""
    form = RecommenderForm()
    if form.validate_on_submit():
        query = Query(raw_ingredients=form.ingredients.data,
sort_mode=form.sort_mode.data, max_time=form.max_time.data,
                      limit=form.limit.data)

        session["query"] = query.__dict__
        return redirect(url_for("recommend"))
    return render_template("index.html", form=form)
```

```
@app.route("/recommend", methods=["GET", "POST"])
def recommend():
    if "query" in session:
        form = SaveRecipeForm()
        query_data = session["query"]
        query = Query(raw_ingredients=query_data["raw_ingredients"],
sort_mode=query_data["sort_mode"],
                      max_time=query_data["max_time"],
limit=query_data["limit"])
        query.results = recommender.find_results(query)
        # Limit number returned
        query.results = query.results[:query.limit]
        return render_template("recommend.html", results=query.results,
query_ingredients=query.cleaned_tokens,
                               form=form)
    else:
        return redirect("/")


@app.route("/save_recipe", methods=["POST"])
def save_recipe():
    if "active_user_email" in session:
        if request.method == "POST":
            user = find_user_by_email(session["active_user_email"])
            user_save_recipe(request.form["recipe_title"], user)
        return redirect("/profile")
    else:
        flash("Please log in to save recipes", "info")
        return redirect("login")


@app.route('/info')
def info():
    # Route for info page
    return render_template("info.html")


@app.route('/register', methods=["GET", "POST"])
def register():
    # Route for register page
    form = CreateProfileForm()
    if form.validate_on_submit():
        if form.password.data != form.repeat_password.data:
            flash("Passwords do not match", "form")
        # Create profile
        else:
            user = User(form.first_name.data, form.last_name.data,
form.email.data, form.password.data)
            user.password_hash =
user.calculate_password_hash(user.plaintext_password)
            profile_added_successfully = add_profile_to_database(user)
            if not profile_added_successfully:
                flash("Email already in use", "form")
            else:
                return redirect("/login")
    return render_template("register.html", form=form)


@app.route("/login", methods=["GET", "POST"])
```

```
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # Find profile with given email
        email = form.email.data
        password = form.password.data
        stored_user = find_user_by_email(email)
        if not stored_user:
            flash("Invalid email", "form")
        else:
            login_user = User(first_name=stored_user.first_name,
last_name=stored_user.last_name, email=email,
                              plaintext_password=password)
            login_user.password_hash =
login_user.calculate_password_hash(login_user.plaintext_password)
            if login_user.password_hash == stored_user.password_hash:
                session["active_user_email"] = email
                return redirect("/profile")
            else:
                flash("Invalid password", "form")
    return render_template("login.html", form=form)


@app.route("/logout")
def logout():
    # Logout user
    if "active_user_email" in session:
        del session["active_user_email"]
    return redirect("/login")


@app.route("/profile", methods=["GET", "POST"])
def profile():
    change_email_form = ChangeEmailForm()
    change_password_form = ChangePasswordForm()
    if change_email_form.validate_on_submit() and change_email_form.data:
        user = find_user_by_email(change_email_form.current_email.data)
        form_password_hash =
User.calculate_password_hash(change_email_form.password.data)
        if not user:
            # invalid email
            flash("Invalid email", "change-email-form")
        else:
            if form_password_hash == user.password_hash:
                has_changed_email = change_user_email(user,
new_email=change_email_form.new_email.data)
                if has_changed_email:
                    return redirect("/logout")
                elif not has_changed_email:
                    flash("Email already in use", "change-email-form")
            else:
                flash("Incorrect password", "change-email-form")
    elif change_password_form.validate_on_submit() and
change_password_form.data:
        user = find_user_by_email(change_password_form.email.data)
        current_password_hash =
User.calculate_password_hash(change_password_form.current_password.data)
        new_password_hash =
User.calculate_password_hash(change_password_form.new_password.data)
        if not user:
            flash("Invalid email", "change-password-form")
```

```
        else:
            if current_password_hash == user.password_hash:
                change_user_password(user, new_password_hash)
                return redirect("/logout")
            else:
                flash("Incorrect password", "change-password-form")
    if "active_user_email" in session:
        # find currently logged in account
        user = find_user_by_email(session["active_user_email"])
        saved_recipes = find_user_saved_recipes(user)
        return render_template("profile.html", user=user,
recipes=saved_recipes, change_email_form=change_email_form,
                                change_password_form=change_password_form)
    else:
        return redirect("login")


@app.route('/api')
def api():
    # Route for api info page
    return render_template("api.html")


@app.route("/api/search")
def api_search():
    # API search function
    if "ingredients" not in request.args:
        return "400: Ingredients string not found", 400
    raw_ingredients = request.args["ingredients"]

    if "max_time" in request.args:
        max_time = request.args["max_time"]
    else:
        max_time = None
    if "sort_mode" in request.args:
        sort_mode = request.args["sort_mode"]
    else:
        sort_mode = "relevancy"

    query = Query(raw_ingredients, sort_mode, max_time)
    if "limit" in request.args:
        query.limit = int(request.args["limit"])

    try:
        query.results = recommender.find_results(query)
    except ValueError:
        #     Not a recognised sort mode
        return "400: Not a valid sort mode", 400
    if query.limit:
        query.results = query.results[:query.limit]
    data = {"results": [r.as_dict() for r in query.results]}
    return data
```

app/__init__.py

```
from flask import Flask, session
import config
from flask_wtf.csrf import CSRFProtect
from flask_mysqldb import MySQL

app = Flask(__name__)
```

```
csrf = CSRFProtect(app)
db = MySQL(app)
app.config["SECRET_KEY"] = config.SECRET_KEY
app.config["MYSQL_DB"] = config.DATABASE_NAME
app.config["MYSQL_HOST"] = config.DATABASE_HOST
app.config["MYSQL_PASSWORD"] = config.DATABASE_PASSWORD
app.config["MYSQL_USER"] = config.DATABASE_USERNAME
from app import routes
```

app/static/css/style.css

```
body
{
    background-color: lightgrey;
}
#navbar
{
    overflow: hidden;
    float: center;
}

a {
color: inherit;
text-decoration: none;
}

table, td, th {
    border: 1px solid black;
    border-collapse: collapse;
}
```

app/templates/api.html

```
{% extends "base.html" %}
{% block title %}API{% endblock %}
{% block content%}
<h1>API</h1>
<p>This website provides an API alternative to make searches for
recipes.</p>
<p>It uses the same functions as the search done through the web page but
takes inputs as URL parameters and returns JSON data.</p>
<h2><b>/api/search</b></h2>
<table id="api-table">
    <tr class="table-header-row">
        <th>Parameter</th>
        <th>Type</th>
        <th>Required</th>
        <th>Purpose</th>
    </tr>
    <tr>
        <td>ingredients</td>
        <td>String</td>
        <td>Yes</td>
        <td>A space-separated list of ingredients</td>
    </tr>
    <tr>
        <td>max_time</td>
        <td>Int</td>
        <td>No</td>
        <td>The maximum total preparation time that returned recipes can
```

```
have</td>
    </tr>
    <tr>
        <td>sort_mode</td>
        <td>String</td>
        <td>No</td>
        <td>How to sort the returned recipes</td>
    </tr>
    <tr>
        <td>limit</td>
        <td>Int</td>
        <td>No</td>
        <td>The number of results to return after they have been
sorted</td>
    </tr>
</table>
<br>
<table id="sort-modes-table">
    <tr class="table-header-row">
        <th>Sort mode</th>
        <th>Description</th>
    </tr>
    <tr>
        <td>
            Relevancy
        </td>
        <td>
            Sorts the recipes by how relevant they are to the query
        </td>
    </tr>
    <tr>
        <td>
            title
        </td>
        <td>
            Sorts the recipes alphabetically by title
        </td>
    </tr>
    <tr>
        <td>
            total_time
        </td>
        <td>
            Sorts the recipes by the time they take to make
        </td>
    </tr>
</table>
<br>
<h2>Example returned JSON</h2>
This shows one returned recipe, although more would likely be in the
"results" list.
<pre>
<code id="json-code-block">
{
    "results": [
        {
            "ingredients":[
                {
                   "name":"leek"
                },
                {
```

```
                    "name":"salt ground black pepper"
                },
                {
                    "name":"unsalted butter"
                }
            ],
            "relevancy":0.0761111692806109,
            "title":"Easy Sauteed Leeks",
            "total_time":26,
            "url":"https://www.allrecipes.com/recipe/8472762/easy-sauteed-
leeks/",
            "website":"allrecipes"
        }
    ]
}
</code>
</pre>
{% endblock%}
```


app/templates/base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet" href="{{ url_for('static',
filename='css/style.css') }}">
</head>
<body>
<div id="navbar">
    <a class="navbar-link" href="{{url_for('index')}}">Home</a>
    <a class="navbar-link" href="{{url_for('info')}}">Info</a>
    <a class="navbar-link" href="{{url_for('profile')}}">Profile</a>
    <a class="navbar-link" href="{{url_for('api')}}">API</a>
</div>
<br>
<div id="content">
    {% block content %}{% endblock %}
</div>
</body>
</html>
```

app/templates/index.html

```
{% extends "base.html" %}
{% block title %}Home Page{% endblock %}
{% block content%}
<div id="page-title" class="title text"><legend><h2>Recommend me
recipes</h2></legend></div>
<div id="recipe-form-container" class="form">
    <form method="POST" id="recommender-form">
        <fieldset>
        {{ form.hidden_tag() }}
        <div>
            {{form.ingredients.label}}
            <br>
            {{form.ingredients()}}
        </div>
        <div id="max-time-field" class="form-field number-field">
```

```
            {{form.max_time.label}}
            <br>
            {{form.max_time()}}
        </div>
        <div id="sort-mode-field" class="form-field select-field">
            {{form.sort_mode.label}}
            <br>
            {{form.sort_mode()}}
        </div>
        <div id="limit-field" class="form-field number-field">
            {{form.limit.label}}
            <br>
            {{form.limit()}}
        </div>
        <br>
        <div id="form-submit-button" class=" form-field form-button">
            {{form.submit()}}
        </div>
    </fieldset>
    </form>
</div>
{% endblock%}
```

app/templates/info.html

```
{% extends "base.html" %}
{% block title %}Info{% endblock %}
{% block content%}
<h1>Info</h1>
<p>
    To use this search, simpy enter a list of space-separated ingredients
in the "ingredients" field of the form.
    Do not include any other kind of seperator in the field, only spaces.
</p>
<h2>Searching</h2>
<p>
    You can optionally include a maximum time that recipes can take to
prepare, this must be an integer greater than 0.
</p>
<p>
    You can also determine what the results will be sorted by, and limit
the number of returned recipes.
</p>
<h2>Results</h2>
<p>
    After you have made a search you will be shown the results. Ingredients
that match your search will be in bold.
    Clicking on any of the recipes will take you to the webpage for it.
</p>
<h2>Accounts</h2>
<p>
    You can also <a href='{{url_for("register")}}'>create an account
here.</a>
    This account will enable you to save recipes.
</p>
<h2>Saving recipes</h2>
One you have crated an account simply click the "save recipe" button on any
recipe after making a search to save it to your account.
{% endblock%}
```

app/templates/login.html

```
{% extends "base.html" %}
{% block title %}Login{% endblock %}
{% block content%}
<div id="page-title" class="title
text"><legend><h2>Login</h2></legend></div>
<div id="login-form-container" class="form">
    <form method="POST" id="login-form">
        <fieldset>
            {{form.hidden_tag()}}
            <div id="email-field" class="form-field string-field">
                {{form.email.label}}
                <br>
                {{form.email()}}
            </div>
            <div id="password-field" class="form-field password-field">
                {{form.password.label}}
                <br>
                {{form.password()}}
            </div>
            <br>
            <div id="form-submit-button" class=" form-field form-button">
            {{form.submit()}}
            </div>
        </fieldset>
    </form>
    {% with alerts = get_flashed_messages(with_categories=true) %}
        {% if alerts %}
            <div id="alerts">
                <ul>
                    {% for type, content in alerts %}
                        {% if type=="form" or type=="info"%}
                            <li>
                                {{ content }}
                            </li>
                        {% endif %}
                    {% endfor %}
                </ul>
            </div>
        {% endif %}
    {% endwith %}
</div>
<div id="register-link-container">
    <a class = "intext-link" href="{{url_for('register')}}"><p>Don't have
an account? Register here</p></a>
</div>
{% endblock%}
```

app/templates/profile.html

```
{% extends "base.html" %}
{% block title %}Profile{% endblock %}
{% block content%}
<div id="page-title" class="title text"><h2>Profile</h2></div>
<div id="user name"><h2>{{user.first_name}} {{user.last_name}}</h2></div>
<br>
<div id="logout-button" class="button">
    <a href='{{url_for("logout")}}'>
        <button>
            Logout
```

```
        </button>
    </a>
</div>
<br>
<div id="profile-options">
    <div id="change-email-form-container" class="form">
        <form method="POST" id="change-email-form">
            <fieldset>
                <h3>Change email</h3>
                {{ change_email_form.hidden_tag() }}
                <div id="current-email-field" class="form-field email-
field">
                    {{change_email_form.current_email.label}}
                    <br>
                    {{change_email_form.current_email()}}
                </div>
                <div id="new-email-field" class="form-field email-field">
                    {{change_email_form.new_email.label}}
                    <br>
                    {{change_email_form.new_email()}}
                </div>
                <div id="change-email-password-field" class="form-field
password-field">
                    {{change_email_form.password.label}}
                    <br>
                    {{change_email_form.password()}}
                </div>
                <br>
                <div id="change-email-submit-button" class="form-field
form-button">
                    {{change_email_form.submit()}}
                </div>
            </fieldset>
        </form>
        {% with alerts = get_flashed_messages(with_categories=true) %}
        {% if alerts %}
            <div id="alerts">
                <ul>
                    {% for type, content in alerts %}
                        {% if type=="change-email-form"%}
                            <li>
                                {{ content }}
                            </li>
                        {% endif %}
                    {% endfor %}
                </ul>
            </div>
        {% endif %}
    {% endwith %}
    </div>
    <div id="change-password-form-container" class="form">
        <form method="POST" id="change-password-form">
            <fieldset>
                <h3>Change password</h3>
                {{ change_password_form.hidden_tag() }}
                <div id="email-field" class="form-field email-field">
                    {{change_password_form.email.label}}
                    <br>
                    {{change_password_form.email()}}
                </div>
                <div id="change-password-current-password-field"
```

```
class="form-field password-field">
                    {{change_password_form.current_password.label}}
                    <br>
                    {{change_password_form.current_password()}}
                </div>
                <div id="change-password-new-password-field" class="form-
field password-field">
                    {{change_password_form.new_password.label}}
                    <br>
                    {{change_password_form.new_password()}}
                </div>
                <br>
                <div id="change-password-submit-button" class="form-field
form-button">
                    {{change_password_form.submit()}}
                </div>
            </fieldset>
        </form>
        {% with alerts = get_flashed_messages(with_categories=true) %}
        {% if alerts %}
            <div id="change-password-alerts">
                <ul>
                    {% for type, content in alerts %}
                        {% if type=="change-password-form"%}
                            <li>
                                {{ content }}
                            </li>
                        {% endif %}
                    {% endfor %}
                </ul>
            </div>
        {% endif %}
    {% endwith %}
    </div>
</div>
<div id="saved-recipes">
    <h3>Saved recipes</h3>
    {% for recipe in recipes %}
    <div class="saved-recipe" id="recipe-{{ loop.index }}" style="border:
solid black">
        <a href="{{ recipe.url }}">
            <div class="recipe-title">
                <h3>{{ recipe.title }}</h3>
            </div>
        </a>
    </div>
    {% endfor %}
</div>
{% endblock%}
```

app/templates/recommend.html

```
{% extends "base.html" %}
{% block title %}Recommend{% endblock %}
{% block content%}
<p>Recommend</p>
<br>

{% for recipe in results %}
<div class="recipe_result" id="recipe-{{ loop.index }}" style="border:
solid black">
```

```
    <a href="{{ recipe.url }}" target="_blank">
        <div class="recipe-title">
            <h3>{{ recipe.title }}</h3>
        </div>
        <div class="recipe-time">
            <p>Total time: {{ recipe.formatted_time }}</p>
        </div>
        <div class="recipe-ingredients">
            <ul>
                {% for ingredient in recipe.ingredients %}
                {% if ingredient.name.split()|select("in",
query_ingredients)|first %}
                    <li><b>{{ ingredient.name }}</b></li>
                {% else %}
                    <li>{{ ingredient.name }}</li>
                {% endif %}
                {% endfor %}
            </ul>
        </div>
    </a>
    <div class="recipe-save-button">
        <form method="POST" action='{{url_for("save_recipe")}}'>
            {{form.hidden_tag()}}
            <input type="submit" value="Save recipe" name="save_button">
            <input type="hidden" value="{{recipe.title}}"
name="recipe_title">
        </form>
    </div>
</div>
{% endfor %}

{% endblock%}
```

app/templates/register.html

```
{% extends "base.html" %}
{% block title %}Register{% endblock %}
{% block content%}
<div id="page-title" class="title
text"><legend><h2>Register</h2></legend></div>
<div id="create-profile-form-container" class="form">
    <form method="POST" id="create-profile-form">
        <fieldset>
            {{form.hidden_tag()}}
            <div id="first-name-field" class="form-field string-field">
                {{form.first_name.label}}
                <br>
                {{form.first_name()}}
            </div>
            <div id="last-name-field" class="form-field string-field">
                {{form.last_name.label}}
                <br>
                {{form.last_name()}}
            </div>
            <div id="email-field" class="form-field string-field">
                {{form.email.label}}
                <br>
                {{form.email()}}
            </div>
            <div id="password-field" class="form-field password-field">
                {{form.password.label}}
```

```
                <br>
                {{form.password()}}
            </div>
            <div id="repeat-password-field" class="form-field password-
field">
                {{form.repeat_password.label}}
                <br>
                {{form.repeat_password()}}
            </div>
            <br>
            <div id="form-submit-button" class=" form-field form-button">
            {{form.submit()}}
        </div>
        </fieldset>
    </form>
    {% with alerts = get_flashed_messages(with_categories=true) %}
        {% if alerts %}
            <div id="alerts">
                <ul>
                    {% for type, content in alerts %}
                        {% if type=="form" %}
                            <li>
                                {{ content }}
                            </li>
                        {% endif %}
                    {% endfor %}
                </ul>
            </div>
        {% endif %}
    {% endwith %}
</div>
<div id="login-link-container">
    <a class = "intext-link" href="{{url_for('login')}}"><p>Already have an
account? Login here</p></a>
</div>
{% endblock%}
```

web_scraping/database_insertion.py

```
import mysql.connector
import json
import config


def batch_insert_recipes(recipe_site: str, start: int, end: int):
    """Insert recipes from json (sliced from start to end) into the
database"""
    json_path = f"{recipe_site}_data\\{recipe_site}_recipes_combined.json"
    with open(json_path) as file:
        recipe_data = json.load(file)[recipe_site][start: end]
    db = mysql.connector.connect(host=config.DATABASE_HOST,
                                 username=config.DATABASE_USERNAME,
                                 password=config.DATABASE_PASSWORD,
                                 database=config.DATABASE_NAME)
    for recipe in recipe_data:
        cursor = db.cursor()
        # Insert the recipe
        recipe_values = (recipe["title"], recipe["total_time"],
recipe["url"], recipe["website"])
        recipe_insert_sql = "INSERT INTO recipes (title, time, url,
website) VALUES (%s, %s, %s, %s)"
```

```
        cursor.execute(recipe_insert_sql, recipe_values)
        recipe_id = cursor.lastrowid

        ingredient_names = [i["name"] for i in recipe["ingredients"]]
        for ingredient in ingredient_names:
            ingredient_sql = f"INSERT IGNORE INTO ingredients (name) VALUES
(\"{ingredient}\")"
            cursor.execute(ingredient_sql)

            # If ingredient already exists, find its ID
            # Otherwise get id of last insert
            if cursor.lastrowid == 0:
                ingredient_select = f"SELECT * FROM ingredients WHERE
name=\"{ingredient}\""
                cursor.execute(ingredient_select)
                result = cursor.fetchone()
                ingredient_id = result[0]
            else:
                ingredient_id = cursor.lastrowid

            link_insert_sql = "INSERT IGNORE INTO has_ingredient
(recipe_id, ingredient_id) VALUES (%s, %s)"
            link_values = (recipe_id, ingredient_id)
            cursor.execute(link_insert_sql, link_values)

    db.commit()
```

web_scraping/ignore_words.txt

```
tablespoon
slice
cup
pan
dish
lb
tsp
ml
tbsp
min
optional
teaspoon
pound
```

web_scraping/mass_scraping.py

```
import json
import scrapers
import time
import os


def batch_scrape_pages(recipe_site: str, scraper: scrapers.WebScraper,
start: int, end: int):
    """Scrape recipe links from the relevant recipe links file from a start
index to the end index and store the result
    in a json file"""
    recipe_file_path =
f"{recipe_site}_data\\{recipe_site}_recipes_{start}_{end - 1}.json"
    ingredients_file_path =
f"{recipe_site}_data\\{recipe_site}_ingredients_{start}_{end - 1}.json"
```

```
    links_file_path = f"{recipe_site}_data\\{recipe_site}links.txt"

    with open(links_file_path, "r") as file:
        links = file.read().split("\n")[start:end]

    recipes = []
    ingredients_data = []

    for link in links:
        try:
            time.sleep(0.05)  # Reduces risk of requests being blocked and
raising a Connection Error
            recipe = scraper.scrape_page(link)
            # Get dictionary representations of Ingredient objects from
Recipe object
            recipe_ingredient_data = [ingredient.__dict__ for ingredient in
recipe.ingredients]
            ingredients_data += recipe_ingredient_data
            # Replace ingredient objects in recipe with dictionary
representations
            # So that ingredients are stored as dictionaries inside the
dictionary representation of the recipe object
            recipe.ingredients = recipe_ingredient_data

            recipes.append(scraper.scrape_page(link))
        except AttributeError as e:
            # Handle invalid page
            print("Invalid page " + link)
            print(e)
        except Exception as e:
            # General error catching, mainly Connection Errors caused by
sending too many requests
            print(e, link)

    recipe_data = [recipe.__dict__ for recipe in recipes]

    with open(recipe_file_path, "w+") as file:
        json.dump({recipe_site: recipe_data}, file)
    with open(ingredients_file_path, "w+") as file:
        json.dump({recipe_site: ingredients_data}, file)


def combine_json(recipe_site: str, ingredients_or_recipes: str):
    """Combine all json files into one, ingredients_or_recipes should be
one of _ingredients_ or _recipes_"""
    data_path = f"{recipe_site}_data\\"
    out_path = data_path +
f"{recipe_site}{ingredients_or_recipes}combined.json"
    file_paths = [file for file in os.listdir(data_path) if
ingredients_or_recipes in file and file.endswith("json")]
    data_list = []
    for path in file_paths:
        with open(data_path + path, "r") as file:
            data = json.load(file)
            data_list += data[recipe_site]
    data = {recipe_site: data_list}
    with open(out_path, "w+") as file:
        json.dump(data, file)
```

web_scraping/natural_language_processing.py

```
import nltk
from nltk import word_tokenize, pos_tag
from nltk.stem import WordNetLemmatizer
from app.models import Ingredient
from typing import Optional

# Download nltk packages and create the Lemmatizer model
# Outside of functions because these are very slow calls, so instead run
when the module is imported
nltk.download("popular")
wnl = WordNetLemmatizer()


def find_phrases(tagged_tokens: list) -> list:
    """Find consecutive sequences of nouns and adjectives"""
    noun_adjective_tags = ["NN", "NNP", "JJ"]  # NLTK part-of-speech tags:
Noun, Proper Noun, Adjective
    phrases = []
    current_phrase_tokens = []

    # Clean tokens of commonly mistaken words
    with open("ignore_words.txt", "r") as file:
        ignore_words = file.read().splitlines()
    tagged_tokens = [token for token in tagged_tokens if token[0] not in
ignore_words]

    for tagged_token in tagged_tokens:
        if tagged_token[1] in noun_adjective_tags:
            current_phrase_tokens.append(tagged_token)
        elif current_phrase_tokens:
            phrases.append(" ".join([token[0] for token in
current_phrase_tokens]))
            current_phrase_tokens.clear()
    else:
        # In case the last token is a noun or adjective, then the phrase
needs to be added
        if current_phrase_tokens:
            phrases.append(" ".join([token[0] for token in
current_phrase_tokens]))

    return phrases


def identify_ingredient(text: str) -> Optional[Ingredient]:
    """Identify the key components of a given ingredient string and return
an Ingredient object"""
    text = text.lower()
    tokens = word_tokenize(text)
    # Turn every token into its lemma form, ie: tomatoes -> tomato
    # All inflections of the word are turned into the root lemma, reducing
the risk of misclassifying an ingredient
    lemmatized_tokens = [wnl.lemmatize(t) for t in tokens]
    # Tag tokens by part of speech, ie: noun, adjective, verb
    tagged_tokens = pos_tag(lemmatized_tokens)

    # Identify sequences of nouns and adjectives
    phrases = find_phrases(tagged_tokens)
    # If no phrases are found then None is returned
    ingredient = Ingredient(" ".join(phrases)) if phrases else None
    return ingredient
```

web_scraping/scrapers.py

```
from app.models import WebScraper, Ingredient, Recipe
from natural_language_processing import identify_ingredient
import re


class AllRecipes(WebScraper):
    def scrape_page(self, url: str) -> Recipe:
        soup = self.make_soup(url)

        # Find title
        title = soup.find(id="article-heading_2-0").text.strip()

        # Find ingredients
        raw_ingredients = [i.text.strip() for i in soup.select('span[data-
ingredient-name="true"]')]
        identified_ingredients = [identify_ingredient(i) for i in
raw_ingredients]
        ingredients = [i for i in identified_ingredients if i]  # Remove
None values returned by identify_ingredient

        # Find total time:
        total_time_regex = r"(?<=Total Time:\n)( *\d+ (hr(s)?|min(s)?) *)+"
        hours_regex = r"\d+(?= hr(s)?)"
        minutes_regex = r"\d+(?= min(s)?)"

        time_text = soup.find(id="recipe-details_1-0").text

        prep_time_results = re.search(total_time_regex, time_text)
        prep_time = prep_time_results[0] if prep_time_results else None
        if prep_time:
            prep_minutes_results = re.search(minutes_regex, prep_time)
            prep_minutes = int(prep_minutes_results[0]) if
prep_minutes_results else 0

            prep_hours_results = re.search(hours_regex, prep_time)
            prep_hours_in_minutes = int(re.search(hours_regex,
prep_time)[0]) * 60 if prep_hours_results else 0

            total_time = prep_minutes + prep_hours_in_minutes
        else:
            # Default value if no total time is defined in the recipe, so
will always be included regardless of max time
            total_time = -1

        # Create recipe object
        recipe = Recipe(title, ingredients, total_time, url, "allrecipes")

        return recipe

    def find_recipe_links(self, url: str) -> list[str]:
        soup = self.make_soup(url)
        links = [element.get("href") for element in soup.findAll("a",
class_="mntl-card-list-items")]
        return links

    def find_links_to_scrape(self, recipe_site: str):
        links_file_path = f"{recipe_site}_data\\{recipe_site}links.txt"
```

```
        # Searching for recipes on the navbar, the page of a-z of recipes
and the a-z of ingredients
        pages_to_search = [["https://www.allrecipes.com/", "header-nav_1-
0"],
                           ["https://www.allrecipes.com/recipes-a-z-
6735880", "alphabetical-list_1-0"],
                           ["https://www.allrecipes.com/ingredients-a-z-
6740416", "alphabetical-list_1-0"]]

        links = []
        for page, element_id in pages_to_search:
            soup = self.make_soup(page)
            links += [link.get("href") for link in
soup.find(id=element_id).findAll("a")]

        links_to_scrape = []
        for link in links:
            try:
                recipe_links = self.find_recipe_links(link)
                for recipe_link in recipe_links:
                    links_to_scrape.append(recipe_link)
            except Exception as e:
                # Handle invalid links
                print(e)
                print(f"Invalid link: {link}")

        unique_links = set(links_to_scrape)
        with open(links_file_path, "w+") as file:
            file.write("\n".join(unique_links))
            print("File Written")


class SimplyRecipes(WebScraper):
    def scrape_page(self, url: str) -> Recipe:
        soup = self.make_soup(url)

        # find title
        title = soup.find("h1", class_="heading__title").text
        # find ingredients
        raw_ingredients = [i.text.strip() for i in soup.findAll("li",
class_="ingredient")]
        identified_ingredients = [identify_ingredient(i) for i in
raw_ingredients]
        ingredients = [i for i in identified_ingredients if i]  # Remove
None values returned by identify_ingredient

        # find total time
        time_text = soup.find("span", id="meta-text_1-0").text
        total_time_regex = r"(?<=Total Time\n)( *\d+ (hr(s)?|min(s)?) *)+"
        hours_regex = r"\d+(?= hr(s)?)"
        minutes_regex = r"\d+(?= min(s)?)"
        prep_time_results = re.search(total_time_regex, time_text)
        prep_time = prep_time_results[0] if prep_time_results else None
        if prep_time:
            prep_minutes_results = re.search(minutes_regex, prep_time)
            prep_minutes = int(prep_minutes_results[0]) if
prep_minutes_results else 0

            prep_hours_results = re.search(hours_regex, prep_time)
            prep_hours_in_minutes = int(re.search(hours_regex,
prep_time)[0]) * 60 if prep_hours_results else 0
```

```
            total_time = prep_minutes + prep_hours_in_minutes
        else:
            # Default value if no total time is defined in the recipe, so
will always be included regardless of max time
            total_time = -1


        # create recipe object
        recipe = Recipe(title, ingredients, total_time, url, "Simply
Recipes")
        return recipe

    def find_recipe_links(self, url: str) -> list[str]:
        soup = self.make_soup(url)
        links = [element.get("href") for element in soup.findAll("a",
class_="mntl-card-list-items")]
        return links

    def find_links_to_scrape(self, recipe_site: str):
        links_file_path = f"{recipe_site}_data\\{recipe_site}links.txt"

        # Searching for recipes on the pages linked on the navbar
        pages_to_search = [["https://www.simplyrecipes.com/", "global-
nav_1-0"]]

        links = []
        for page, element_id in pages_to_search:
            soup = self.make_soup(page)
            links += [link.get("href") for link in
soup.find(id=element_id).findAll("a")]

        links_to_scrape = []
        for link in links:
            try:
                recipe_links = self.find_recipe_links(link)
                for recipe_link in recipe_links:
                    links_to_scrape.append(recipe_link)
            except Exception as e:
                # Handle invalid links
                print(e)
                print(f"Invalid link: {link}")

        unique_links = set(links_to_scrape)
        with open(links_file_path, "w+") as file:
            file.write("\n".join(unique_links))
            print("File Written")
        return unique_links
```

Testing

A video recording of the website testing can be found at https://youtu.be/anZk34o99No and at https://tinyurl.com/22c6dre7

Website testing

Test 1.1.1 – Recommend form – Missing required data

Description

Not entering values into the ingredients field - a required field

Input

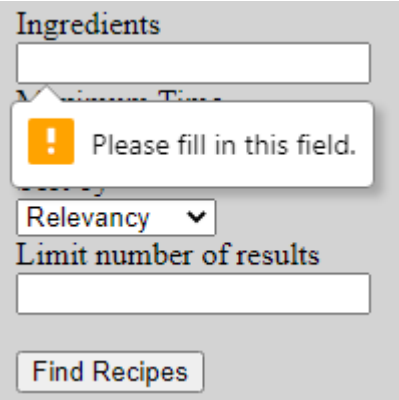
None

Expected result

The website should reject the submission and inform the user that the ingredients field must be filled out

Actual result

Success

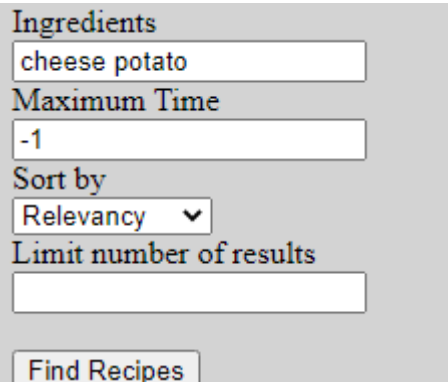


Test 1.1.2 – Recommend form – Invalid maximum time

Description

Giving an invalid time in the “Maximum Time” field

Input

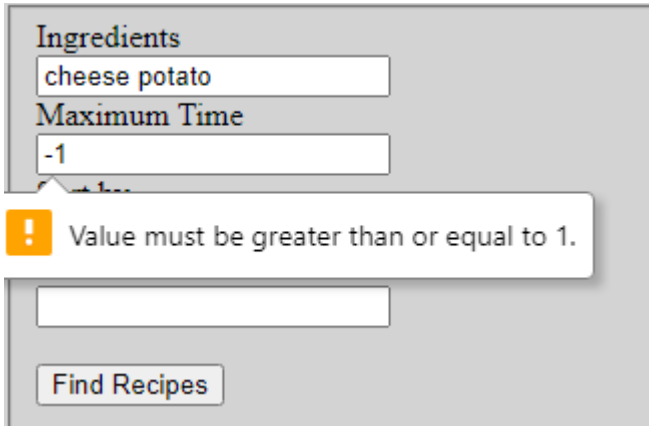


Expected result

The website will reject the submission as the maximum time must be greater than or equal to 1

Actual result

Success

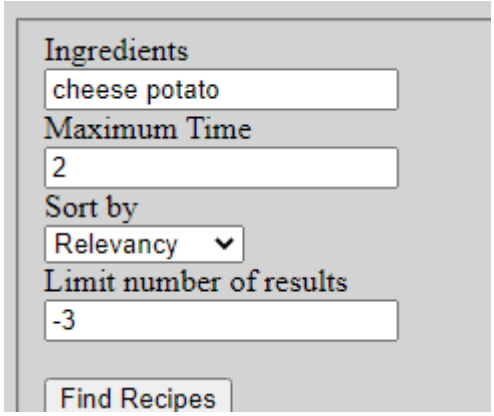


Test 1.1.3 – Recommend form – Invalid limit

Description

Giving an invalid limit on the number of results

Input

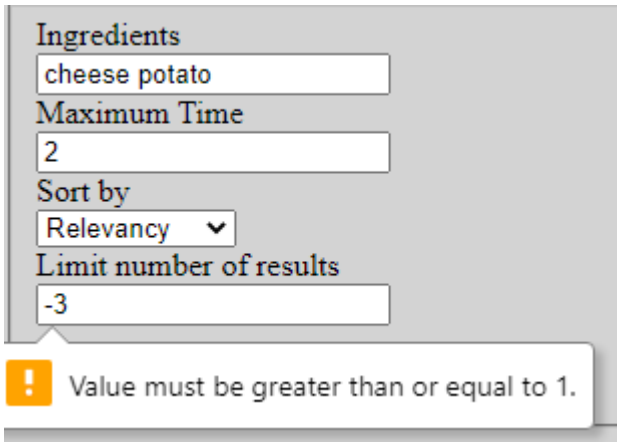


Expected result

The website will reject the submission as the limit must be greater than or equal to 1

Actual result

Success



Test 1.1.4 – Recommend form – Valid search with relevancy sort

Description

Performing a valid search for recipes

Input

Ingredients

cheese potato

Maximum Time

30

Sort by

Relevancy

Limit number of results

2

Find Recipes

Expected result

2 results that contain “cheese” and/or “potato” as ingredients, have a maximum time of less than 30 minutes and are sorted by relevancy.

Actual result

Success

Recommend

Potato Pancakes II

Total time: 15 minutes

- salt
- **potato**
- egg
- butter
- **cheddar cheese**

Save recipe

Homemade Hashbrowns

Total time: N/A

- salt
- onion
- ground black pepper
- **potato**
- egg
- olive oil

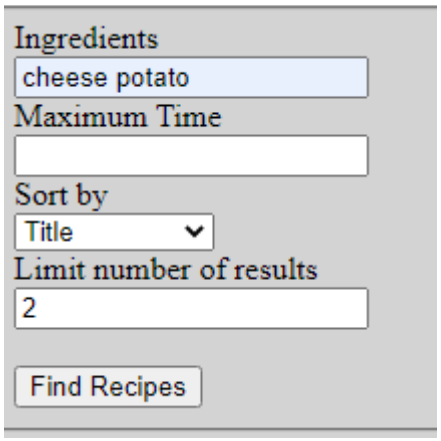
Save recipe

Test 1.1.5 – Recommend form – Valid search with title sort

Description

Searching for recipes and sorting the results alphabetically by title

Input

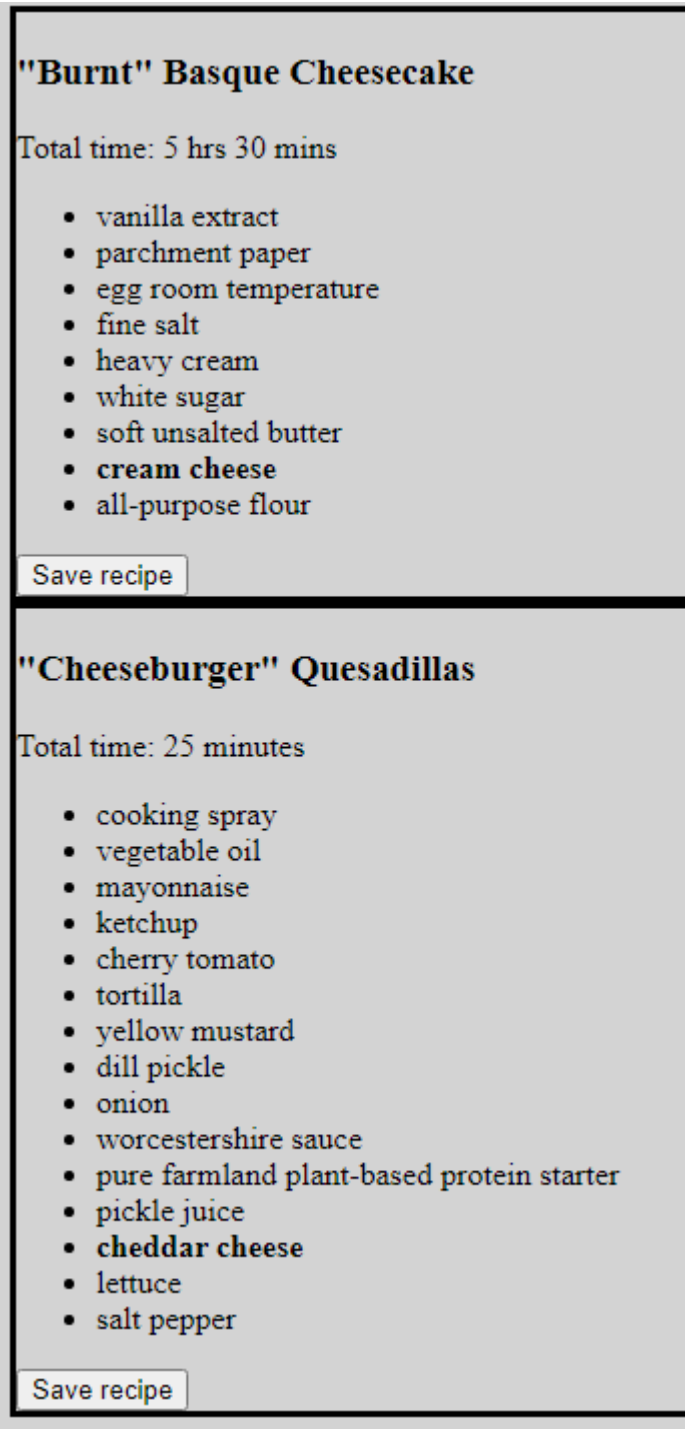


Expected result

2 results that contain “cheese” and/or “potato” as ingredients and are sorted alphabetically by title.

Actual result

Partial success – although the results are technically sorted alphabetically the “ character in the title has placed these recipes at the top of the list of results, which may not be the behaviour expects.

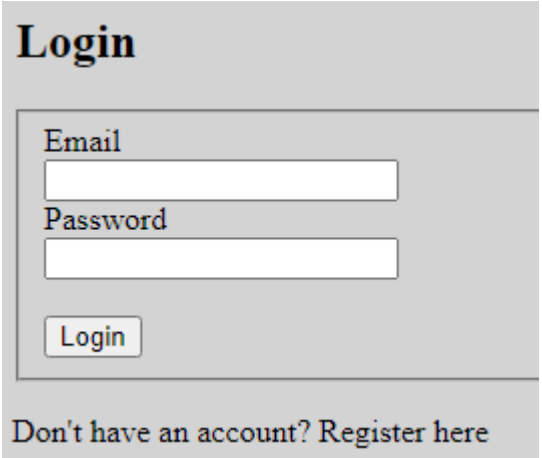


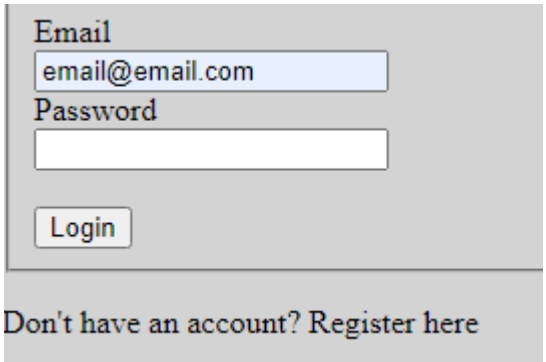
Test 1.2.1 – Login form – Missing required data

Description

Not giving an email or password value when logging in

Input



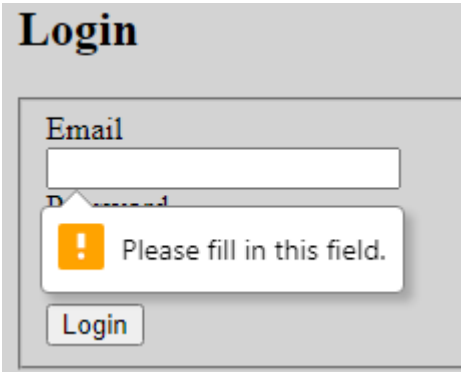


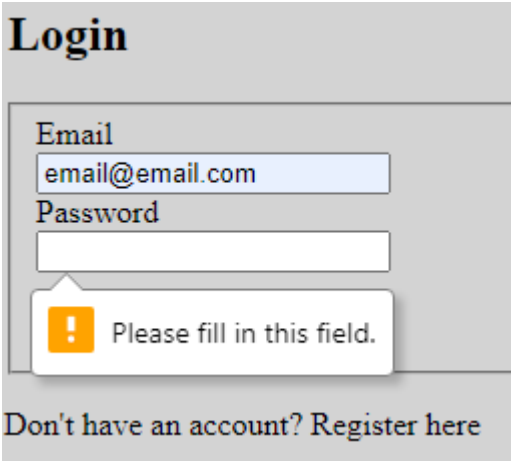
Expected result

Submission rejected and user informed of required fields

Actual result

Success



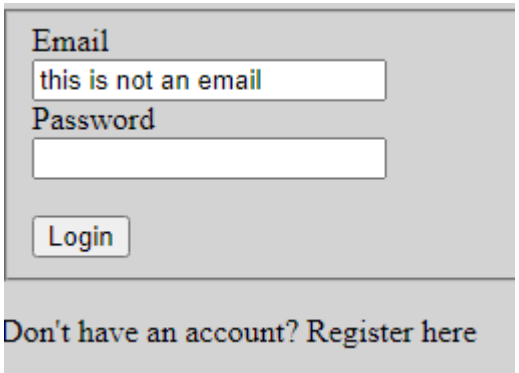


Test 1.2.2 – Login form – Invalid email

Description

Entering a non-email string in the email field

Input



Expected result

The submission is rejected, and the user is informed

Actual result

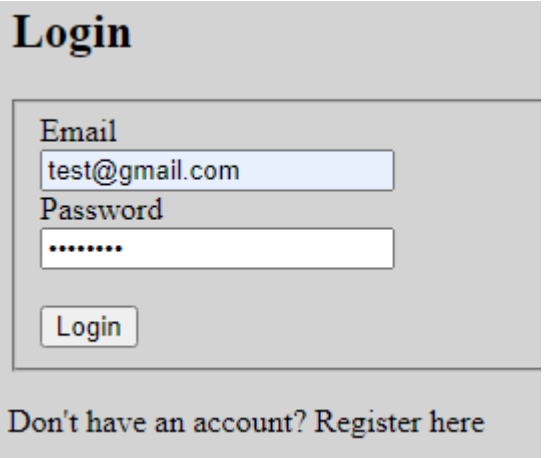
Success

Test 1.2.3 – Login form – Incorrect password

Description

Entering a correct email but with the wrong password for that account

Input



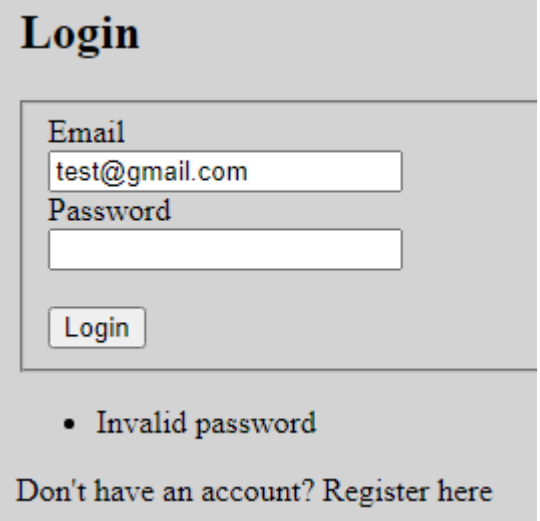
Note – Password input is “Password”

Expected result

Rejected due to incorrect password

Actual result

Success

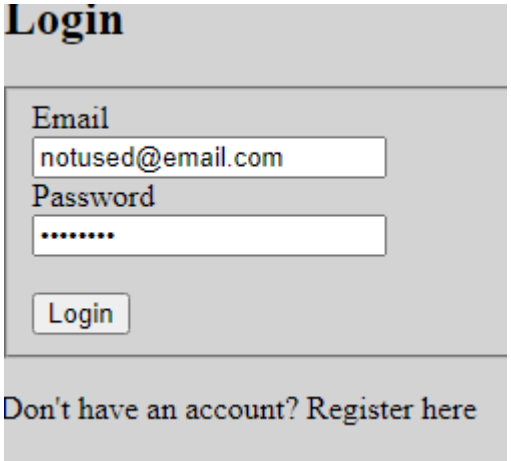


Test 1.2.4 – Login form – Email not associated with an account

Description

Entering a valid email address, but one that isn’t used by any existing account

Input



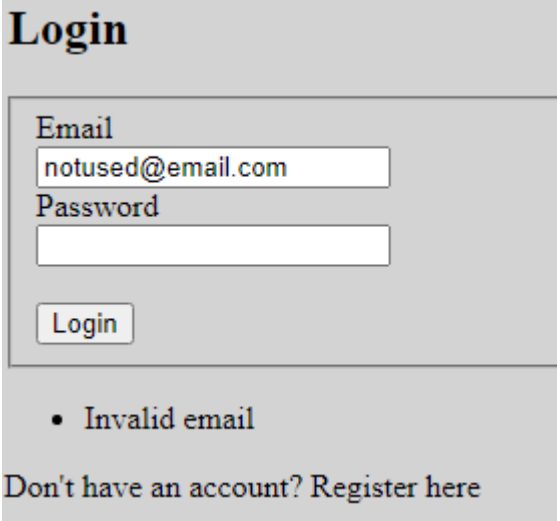
Note – password input is “Password”

Expected result

Rejected as email not linked to an account

Actual result

Success

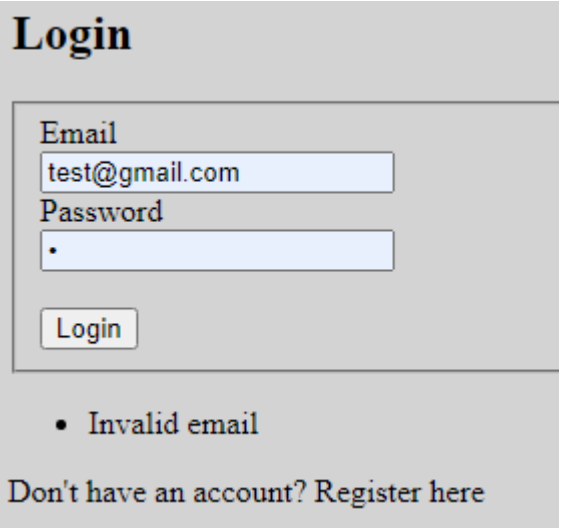


Test 1.2.5 – Login form – Valid login

Description

Attempting to log in with valid inputs

Input



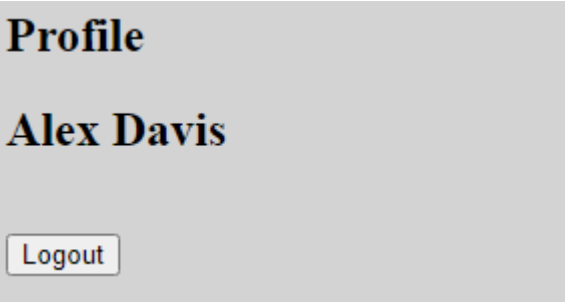
Note – password input is “a”, which is the correct password for this account

Expected result

The client will be logged in and redirected to the profile page for “Alex Davis”

Actual result

Success

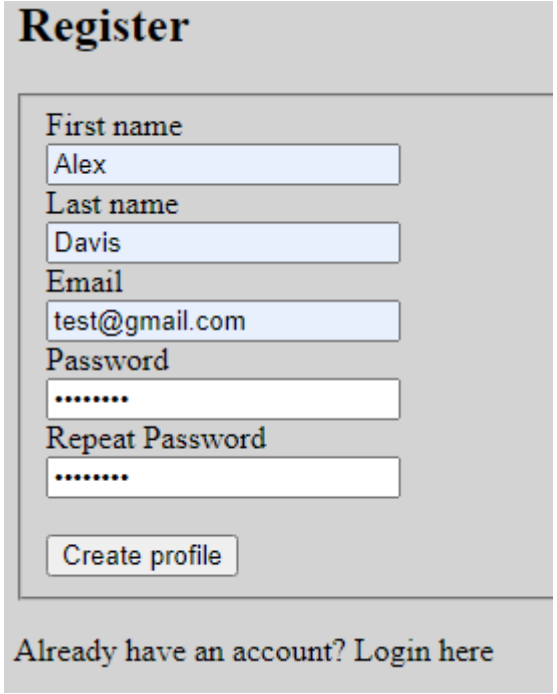


Test 1.3.1 – Registration form – Email already in use

Description

Attempting to create an account with an email address already associated with another account

Input



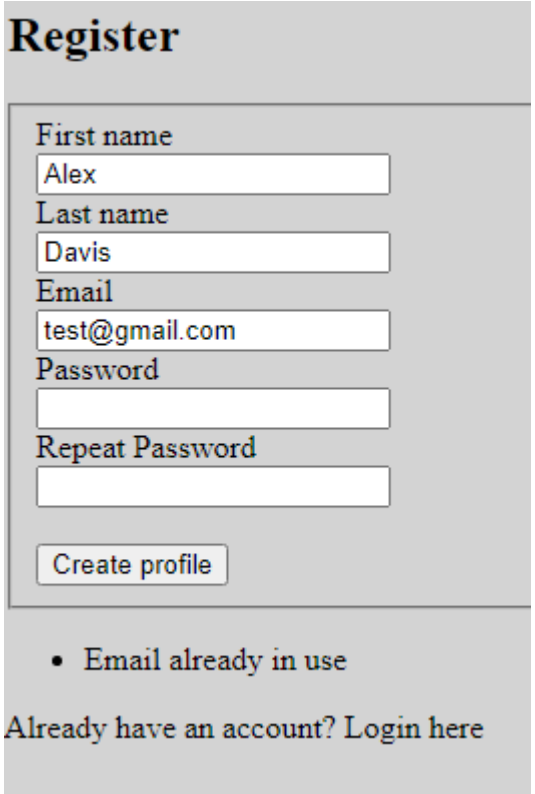
Note – both password field inputs are “Password”

Expected result

Rejected as the email is already in use

Actual result

Success

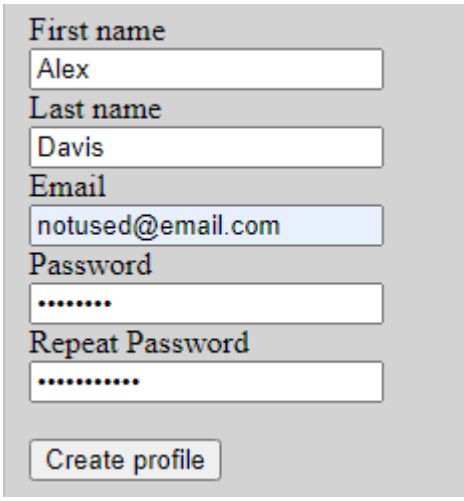


Test 1.3.2 – Registration form – Passwords do not match

Description

Passwords are entered twice when creating an account to ensure that it is typed correctly. I will test what happens if the password fields do not match

Input



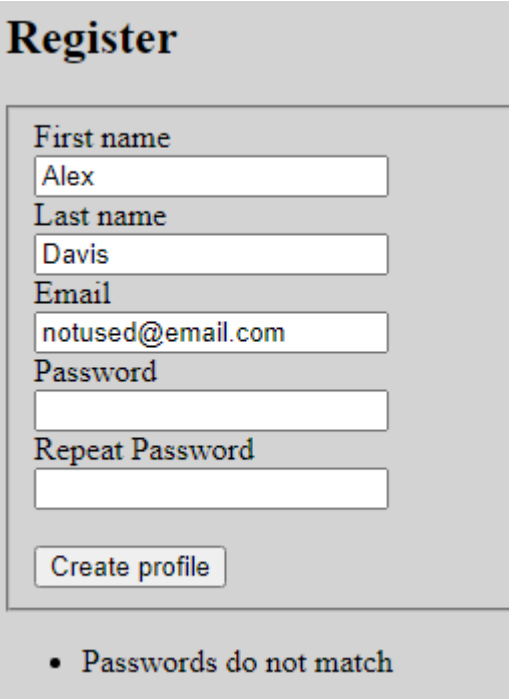
Note – Password field input is “Password” and Repeat Password field input is “NotPassword”

Expected result

Rejected as passwords do not match

Actual result

Success

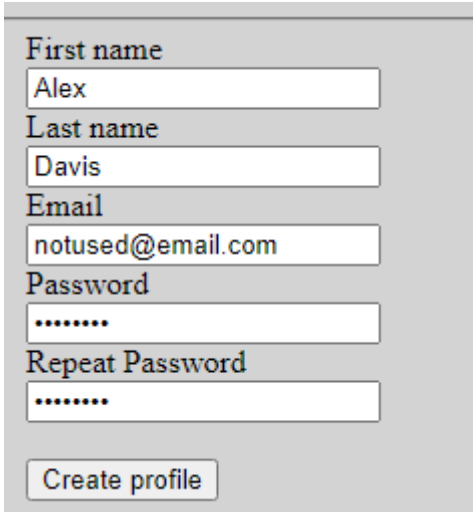


Test 1.3.3 – Registration form – Valid registration

Description

Creating an account with valid inputs

Input



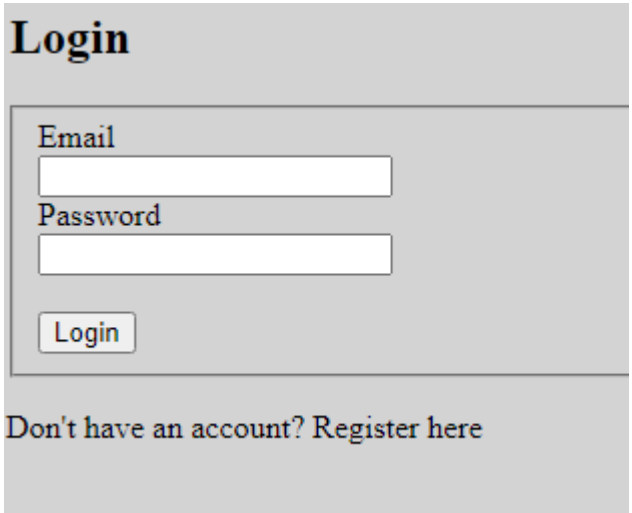
Note – both password fields have the input “Password”

Expected result

The client will be redirected to the login page and an entry will be added to the database table Users

Actual result

Success




```
mysql> SELECT * FROM users WHERE email="notused@email.com";
+---------+------------+-----------+-------------------+------------+
| user_id | first_name | last_name | email             | password   |
+---------+------------+-----------+-------------------+------------+
|      41 | Alex       | Davis     | notused@email.com | 1441103081 |
+---------+------------+-----------+-------------------+------------+
```

Test 1.4.1 – Saving a recipe – Not logged in

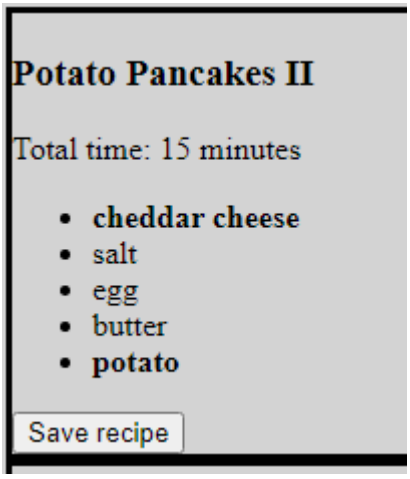
Description

Saving a recipe should only work if an account is logged. I will make a search for recipes and then attempt to save one whilst not logged in.

Input

User – not logged in

Clicking the button in the below screenshot

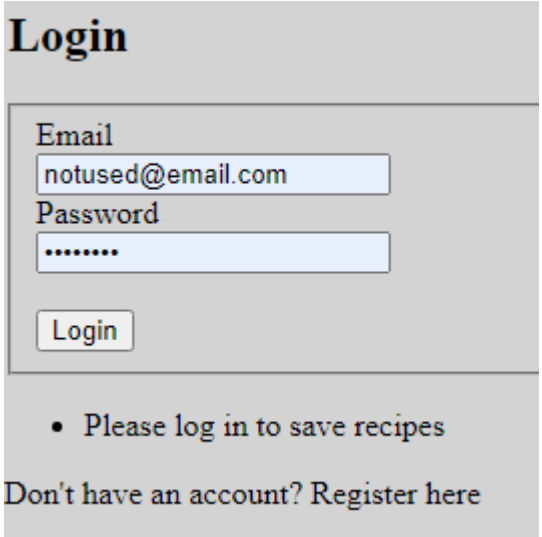


Expected result

Redirected to the login page with an alert saying that users need to be logged in to save recipes

Actual result

Success



Test 1.4.2 – Saving a recipe – Logged in

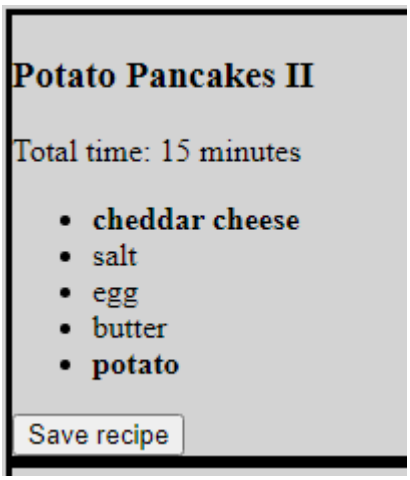
Description

Saving a recipe whilst logged in should save it to that account

Input

User – logged in

Clicking the button in the below screenshot

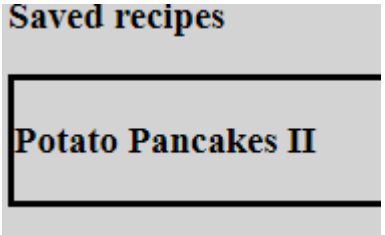


Expected result

The client will be redirected to the profile page, where the recipe will be saved, and a database entry for the saved recipe will be added to the "saved_recipe" linking table.

Actual result

Success




```
mysql> SELECT users.email, recipes.title FROM users, recipes, saved_recipe WHERE users.user_id=saved_recipe.user_id AND
recipes.recipe_id = saved_recipe.recipe_id AND users.email="notused@email.com";
+-------------------+--------------------+
| email             | title              |
+-------------------+--------------------+
| notused@email.com | Potato Pancakes II |
+-------------------+--------------------+
```

Test 1.5.1 – API – Missing ingredients string

Description

Making a request to the API without an ingredient parameter

Input

Request made to /api/search

Expected result

Returned 400 status code

Actual result

Success

400: Ingredients string not found

Test 1.5.2 – API – Invalid sort mode

Description

Submitting an invalid sort mode to the API

Input

Request made to /api/search?ingredients=”salmon”&sort_mode=”invalid”

Expected result

Returned 400 status code

Actual result

Success

400: Not a valid sort mode

Test 1.5.3 – API – Valid search with relevancy sort

Description

Making a valid search request to the API and sorting the results by relevancy

Input

Request made to /api/search?ingredients=”cinnamon sugar apples”&limit=2&sort_mode=relevancy

Expected result

Returned JSON containing two recipes, both containing at least one of the given ingredients and sorted by relevancy

Actual result

Success

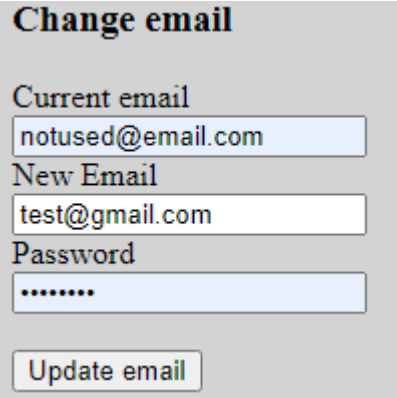
{"results":[{"ingredients":[{"name":"apple"},{"name":"ground cinnamon"},{"name":"white sugar"},{"name":"water"}],"relevancy":0.20484223250543898,"title":"Sarah's Homemade Applesauce","total_time":25,"url":"https://www.allrecipes.com/recipe/51301/sarahs-applesauce/","website":"allrecipes"},{"ingredients":[{"name":"all-purpose flour"},{"name":"ground cinnamon"},{"name":"butter"},{"name":"white sugar"},{"name":"salt"},{"name":"apple"}],"relevancy":0.19387593057118446,"title":"Apple Crisp - Perfect and Easy","total_time":60,"url":"https://www.allrecipes.com/recipe/229274/apple-crisp-perfect-and-easy/","website":"allrecipes"}]}

Test 1.6.1 – Change email form – Email already in use

Description

Trying to change an account’s associated email to an email already in use

Input

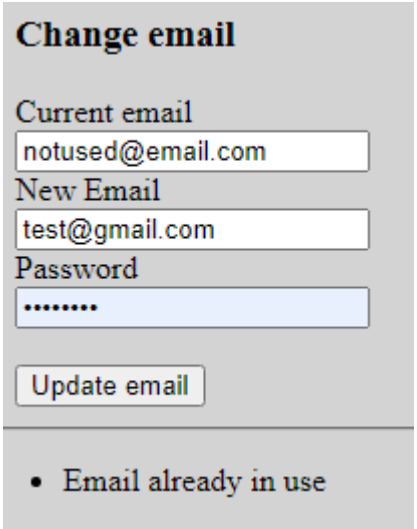


Expected result

Rejected as new email is already in use

Actual result

Success

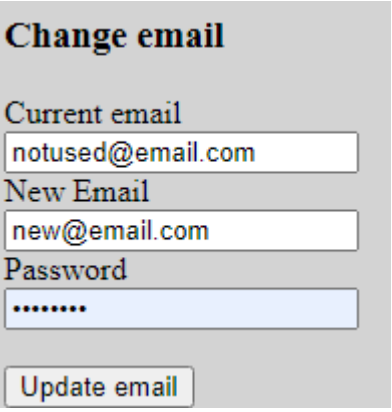


Test 1.6.2 – Change email form – Incorrect password

Description

Trying to change an account's associated email with an incorrect password

Input



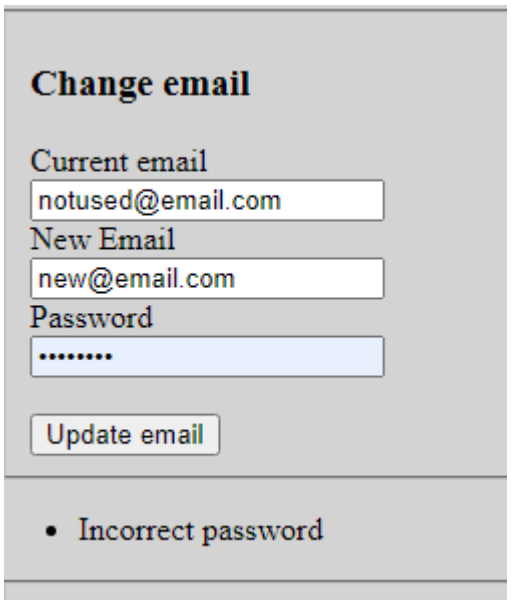
Note – Password field input is “Incorrect”

Expected result

Rejected due to incorrect password

Actual result

Success

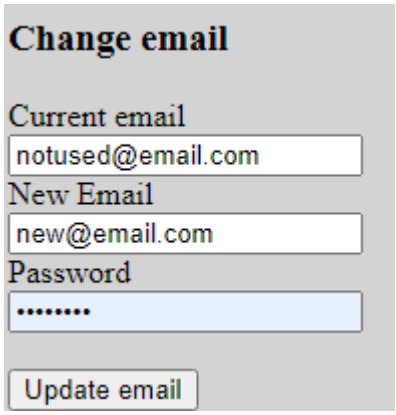


Test 1.6.3 – Change email form – Valid input

Description

Trying to change an account’s associated email with valid inputs

Input



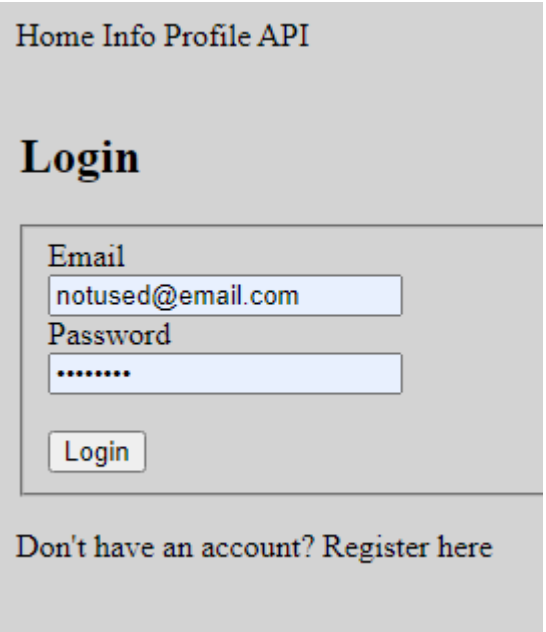
Note – Password field input is “Password”, the correct password

Expected result

The user will be logged out and redirected to the log in page. The associated with the account will be changed in the database.

Actual result

Success



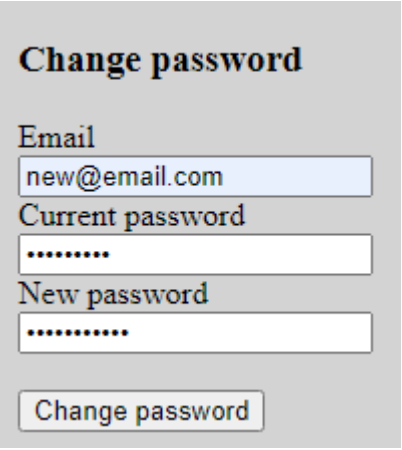

```
mysql> select * from users where email="new@email.com";
+---------+------------+-----------+---------------+------------+
| user_id | first_name | last_name | email         | password   |
+---------+------------+-----------+---------------+------------+
|      45 | Alex       | Davis     | new@email.com | 1441103081 |
+---------+------------+-----------+---------------+------------+
1 row in set (0.00 sec)
```

Test 1.7.1 – Change password form – Incorrect password

Description

Trying to change a user's password but giving the incorrect current password.

Input



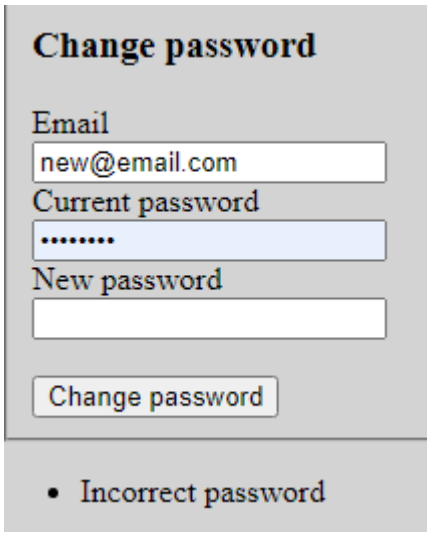
Note - Current password field input is "Incorrect" and the New password field input is "NewPassword"

Expected result

Rejected due to incorrect password

Actual result

Success

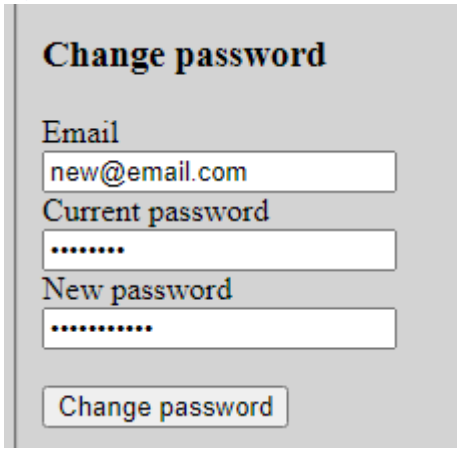


Test 1.7.1 – Change password form – Valid inputs

Description

Changing a user's password with valid inputs

Input



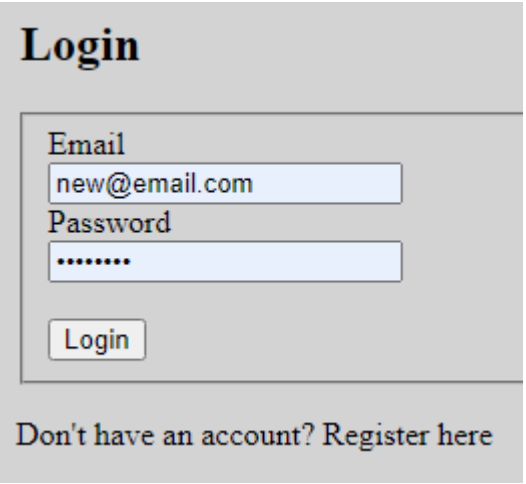
Note – Current Password field value is "Password", the correct password. New Password field value is "NewPassword"

Expected result

The user will be logged out and redirected to the login page. The hash of the password will be updated in the database.

Actual result

Success




```
mysql> select * from users where email="new@email.com";
+---------+------------+-----------+---------------+------------+
| user_id | first_name | last_name | email         | password   |
+---------+------------+-----------+---------------+------------+
|      45 | Alex       | Davis     | new@email.com | 1578600235 |
+---------+------------+-----------+---------------+------------+
1 row in set (0.00 sec)
```

Scraper testing

Test 2.1.1 – Allrecipes Scraper – Invalid recipe page

Description

Attempting to scrape a page that isn't a recipe

Input

```
from scrapers import AllRecipes

url = "https://www.allrecipes.com/could-an-onion-be-the-cure-for-your-next-cold-7252890"

scraper = AllRecipes()
result = scraper.scrape_page(url)
print(result)
```

Expected result

An error will be raised that would be handled in the mass scraper module

Actual result

Success

```
AttributeError: 'NoneType' object has no attribute 'text'
```

Test 2.1.2 – Allrecipes Scraper – Scraping a recipe page

Description

I will attempt to scrape information about a recipe and then print it is a dictionary.

Input

```
from scrapers import AllRecipes

url = "https://www.allrecipes.com/recipe/270713/burnt-basque-cheesecake/"

scraper = AllRecipes()
result = scraper.scrape_page(url)
print(result.as_dict())
```

Expected result

A dictionary containing information about the recipe, such as the total time to cook being 5 hours and 30 minutes

Actual result

{'title': '"Burnt" Basque Cheesecake', 'ingredients': [{'name': 'soft unsalted butter'}, {'name': 'parchment paper'}, {'name': 'cream cheese'}, {'name': 'white sugar'}, {'name': 'all-purpose flour'}, {'name': 'fine salt'}, {'name': 'vanilla extract'}, {'name': 'egg room temperature'}, {'name': 'heavy cream'}], 'total_time': 330, 'url': 'https://www.allrecipes.com/recipe/270713/burnt-basque-cheesecake/', 'website': 'allrecipes'}

Test 2.2.1 – SimplyRecipes Scraper – Invalid recipe page

Description

Attempting to scrape a page that isn't a recipe

Input

```
from scrapers import SimplyRecipes

url = "https://www.simplyrecipes.com/best-microwaveable-foods-from-costco-
under-usd10-7095746"

scraper = SimplyRecipes()
result = scraper.scrape_page(url)
print(result.as_dict())
```

Expected result

An error will be raised that would be handled in the mass scraper module

Actual result

Success

```
AttributeError: 'NoneType' object has no attribute 'text'
```

Test 2.2.2 – SimplyRecipes Scraper – Scraping a recipe page

Description

I will attempt to scrape information about a recipe and then print it is a dictionary.

Input

```
from scrapers import SimplyRecipes

url = "https://www.simplyrecipes.com/recipes/tuna_patties/"

scraper = SimplyRecipes()
result = scraper.scrape_page(url)
print(result.as_dict())
```

Expected result

A dictionary containing information about the recipe, such as the total time to cook being 18 minutes

Actual result

{'title': 'Tuna Patties', 'ingredients': [{'name': '6-ounce'}, {'name': 'dijon mustard'}, {'name': 'white bread small piece'}, {'name': 'lemon zest'}, {'name': 'lemon juice'}, {'name': 'water liquid tuna'}, {'name': 'fresh parsley'}, {'name': 'fresh chive green onion shallot'}, {'name': 'salt ground black pepper'}, {'name': 'crystal tabasco hot sauce'}, {'name': 'raw egg'}, {'name': 'extra virgin olive oil'}, {'name': 'butter'}, {'name': 'lemon wedge'}], 'total_time': 18, 'url': 'https://www.simplyrecipes.com/recipes/tuna_patties/', 'website': 'Simply Recipes'}

Evaluation

Introduction

Overall, I was largely successful in achieving my goal of creating website that could recommend users recipes based on a given set of ingredients. Scraping recipes proved to be quite complex, but I was successful in creating web scrapers for two different websites and extracting key information. I was able to store the recipes and ingredients in a structured and well-organised database, and query that to retrieve information quickly and easily. The algorithm to recommend recipes was the most theoretically complex element of the project. It also fully fulfils the requirements, and accurately calculates a relevancy score for recipes. The website functions well, is user-friendly and fast and fulfils the requirements.

Objectives

Key:

Completed

Partially completed

Not completed

0. The project must allow users to find recipes based on a series of inputted requirements

1. The project must have a web interface(website)

1.1. The website must have a navigation bar at the top of screen that provides access to all key pages

1.2. The website must have an information page

1.2.1. This page should explain how to use the different functions of the website

1.3. The website must have a home page

1.3.1. On this page there must be a series of fields for the user to enter the information that is used to find recipes

1.3.1.1. This information could include:

1.3.1.1.1. The ingredients to be used

1.3.1.1.2. The maximum time taken to cook the meal

1.3.1.1.3. The 'type' of meal to be cooked, i.e., dinner, breakfast, etc

1.3.1.2. Only the ingredients field should be mandatory

1.3.1.2.1. If no ingredients are provided, the website should reject the search and inform the user that information must be provided

1.3.1.3. If any input is invalid then the search should be rejected with an appropriate message

1.3.2. This home page must provide a button to begin the search for recipes once information is provided

1.4. The website should have a page to display the results of the search

1.4.1. The results must be generated through the search algorithm described in 2.

1.4.2. The results should be displayed in a standardised format

1.4.2.1. Each result should display:

1.4.2.1.1. The name of the recipe

1.4.2.1.2. The time to cook

1.4.2.1.3. The ingredients the recipe requires

1.4.2.1.3.1. The ingredients that are part of the user's search should be distinguished from the others

1.4.2.1.4. A button to 'save' a recipe if a profile is logged in

1.4.2.1.4.1. The recipe should then be added to a stored list associated with the profile in the database

1.4.2.1.4.2. If not logged in, the button should redirect the user to a login page

1.4.2.2. Clicking on a result should open the page the recipe is from

1.4.3. The results should be sortable by:

1.4.3.1. Relevance

1.4.3.1.1. The relevance of a recipe is determined by how well it fits the requirements given by the user

1.4.3.2. Time to cook

1.4.4. The user should be able to limit the number of results returned

1.5. The website must allow a user to have a profile

1.5.1. The website must provide a profile page

1.5.1.1. If a profile is not logged in, the user should be prompted to either:

1.5.1.1.1. Create an account

1.5.1.1.1.1. This should be done by setting a username and password

1.5.1.1.1.1.1. Text entered in the password field should be obscured and asked for twice to confirm it has been typed correctly

1.5.1.1.1.2. Once a profile has been created, the username and hash of the password should be stored in the database, as well as a unique API key

1.5.1.1.2. Or log in

1.5.1.1.2.1. This user should be prompted to enter a username and password

1.5.1.1.2.2. The relevant database entry for the username should be found and the hash of the given password checked against the stored hash

1.5.1.1.2.3. If a relevant entry is found and the hashes match, then the profile should be logged in and the user redirected to the profile page

1.5.1.1.2.4. Otherwise, the data should be rejected, and the user informed that the login failed

1.5.1.2. If the user is logged in, the profile page should display:

1.5.1.2.1. The name of the user

1.5.1.2.2. Their 'saved' recipes

1.5.1.2.2.1. These should be displayed as a list like the results page

1.5.1.2.3. Fields to change their username and password

1.5.1.2.3.1. If a change is made the database should be updated and the user logged out

1.5.1.2.4. A button to log out

1.6. The project must have an API

1.6.1. The API should have documentation detailing every function

1.6.1.1. This documentation should be on a page of the website

1.6.2. Requests should use URL parameters

1.6.3. Responses should use the JSON format

1.6.4. "Bad" requests should be rejected with an appropriate message

1.6.5. The API should provide the recipe search functionality of the main page

1.6.5.1. The algorithm described in 2. should be used to find recipes

1.6.5.2. The found results should be returned

1.6.5.2.1. The results should be sorted by the user's specified method if one exists

1.6.5.2.2. The number of results should be limited to the user's specified value if one exists

2. The project must have an algorithm to search for recipes

2.1. The algorithm should return results where at least one of the ingredients given in the search is included in the ingredients of the recipe

2.1.1. It should generate a 'relevance' for the result based upon how many of the given ingredients it includes

2.2. If a maximum cooking time is given, then the algorithm should return only results that do not exceed this cooking time

2.3. If a type of meal is given, then the algorithm should only return recipes that are of this type

3. The project must have a database that stores the following objects and attributes:

3.1. Recipe

3.1.1. The ID

3.1.2. The name

3.1.3. The list of ingredients

3.1.4. The time the recipe takes to cook

3.1.5. The 'type' of meal

3.1.6. A link to the original recipe

3.2. Profiles

3.2.1. The ID

3.2.2. The username

3.2.3. The hash of the password

3.2.4. The list of saved recipes

3.3. Ingredients

3.3.1. The ID

3.3.2. The name

3.3.3. The recipes it is used in

4. The data for recipes in the database should be collected using a web scraper

4.1. The web scraper should be able to collect information from multiple cooking websites

4.2. Information from websites should be stored in JSON

4.2.1. The information should follow the structure presented in 3.1.

4.3. Information collected should be transferred to the database

4.4. Any information that the web scraper fails to collect or incorrectly collects could be manually corrected

Objective	Comments
1.3.1.3	This was not fulfilled, as I found it very difficult to identify the "type" of a recipe from the recipe web pages. Ultimately I decided that it didn't add much to the search
1.5.1.1.1.2	I did implement password hashing, but not a system of API keys as there was not a need for them.
1.5.1.2.2.1	The recipes are displayed, but only their titles are shown. I chose to do this so that the recipes fit better on the profile page
2.3	As above, I did not implement a system to account for the "type of meal".
4.1	I had planned to create scrapers for 3 websites, but only found time to create 2. As such, this objective is partially completed.

4.4	I did not perform manual correction on the JSON data because the number of recipes was so large that it would not have been a productive use of time.

Feedback

I created a survey to gather feedback on my project from my clients.

Questions

1. Do you find the website easy to use?
2. Would you use this website to find recipes?
3. Do the recipes returned match what you searched for?
4. Were you able to create an account and use it to save recipes?
5. What aspects of the interface would you change?

Responses

- **Iain Walker**
  - 1. Yes the search prompts are clear and the headings are self explanatory
  - 2. Yes definitely (although there are features I'd want available for daily use)
  - 3. While results were limited, they were within parameters for what I'd searched - this is where I'd want a feature that lets me add my own recipes to the cache
  - 4. Yes, that was very easy
  - 5. I'd possibly just make the fields a little bigger, and put a bit more space between the headings for people with impaired vision. plus there's plenty of screen real-estate not currently being used
- **George Mack**
  - 1. I found the website easy to use, once the initial issues were resolved. It is a neat, understandable premise, and I found it simple to find relevant and useful recipes.
  - 2. Genuinely think this could be a really good tool for finding recipes.
  - 3. The recipes were useful, looked good, and provided a good amount of variety. (To be honest, I'm not sure if I'd ever want to filter results by anything other than relevance or length - I just ended up with seemingly random recipes which matched only one ingredient.)
  - 4. Yes, I was able to create an account without any difficulty. I very much appreciated that I wasn't required to use numbers or special characters for the password. Saving recipes was simple - although I did wonder if there could be a 'search saved recipes' option.
  - 5. I might be tempted to change the colours/graphics - but I recognise that that wasn't the primary aim of the project. Other than that, my only issue was that, when inputting the time, I was unsure if the units were minutes or hours - but it seemed to work.
- **Matthew**
  - 1. Mostly, the navigational tools were not obvious at a first glance.
  - 2. Probably.
  - 3. Yes, definitely.
  - 4. Yes
  - 5. Just make the navigational tools clearer
- **Ben Davis**

    - 1. Yes, although information about the search format (space separated list) would be useful to have close to the search box
    - 2. Yes, it's actually very useful and rapidly identifies some relevant recipes, including some I wouldn't have thought of
    - 3. Yes, and the use of bold to highlight the searched ingredients is very useful
    - 4. Yes, although: a clear "login/register" page rather than profile might be slightly simpler to use and a forgotten password / password reset would be useful, since existing email addresses cannot be re-used (which is good, but tricky if you've forgotten the password). Saving the recipes is a useful function and works well.
    - 5. Visually it is rather plain and more use of colour (white background, coloured search terms and so on) would liven the appearance. This is a pretty minor point though.

Discussion of feedback

- Question 1
    - The response to this is overall positive and indicates that I have successfully met requirements.
    - Developing the layout would definitely be an area of improvement I would look into further with more time.
- Question 2
    - Responses again indicate that the project has met requirements.
    - But with areas of improvement that could make the website easier to use.
- Question 3
    - Again, indicates that I have met requirements and that the search algorithm functions well.
    - Allowing users to upload their own recipes is an interesting feature that I would explore if I were to take the project further.
- Question 4
    - Responses here suggest that the creation of an account is simple to do.
    - A suggestion I would act on is making the “log in” and “register” pages more visible, possibly by placing them on the navigation bar.
    - Adding a system to reset passwords was beyond the scope of this project, as it would require an automated email account to send the reset links.
- Question 5
    - The clear response here is that my interface could have been larger and visually interesting.
    - This is definitely an area of improvement I would investigate in future.
    - Not including the units in the time was definitely an oversight.

Areas of improvement

Areas to improve

If I were to start my project again, I would focus more on creating a visually appealing and understandable user interface. I would also improve my ingredient identification algorithm to be more complex and reliable. Additionally, I would have created web scrapers for more websites to widen the variety of recipes in the database.

Additional features

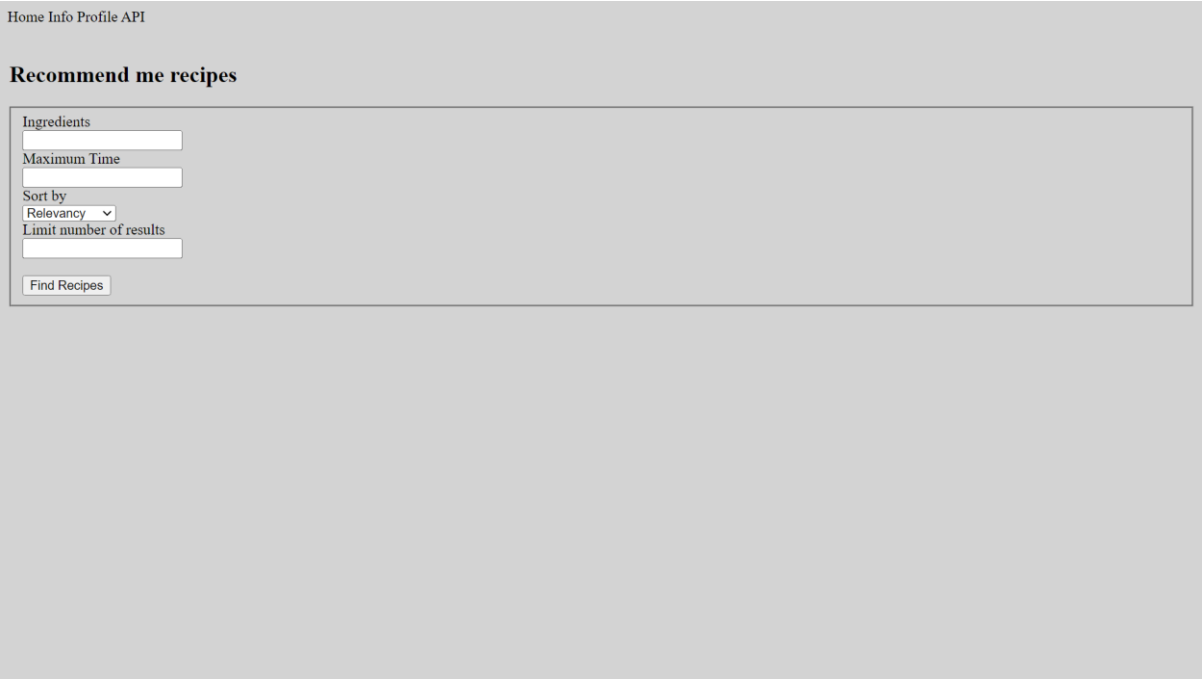
I would have liked to add a “types of recipe” option to the search, and this is something I might look in to further were to continue with the project. I would also add a system to search saved recipes, as suggested in the feedback. Additionally, I would add more functions to the API, such as getting a user’s saved recipes and saving recipes. A difficult addition would be my initial idea of a dietary requirements field in the search, which I was unable to include. A definite additional feature would be a system to identify ingredients which occur commonly together, which would involve the creation of a weighted graph where the nodes are ingredients and the weights of the edges are the number of recipes they appear in together.

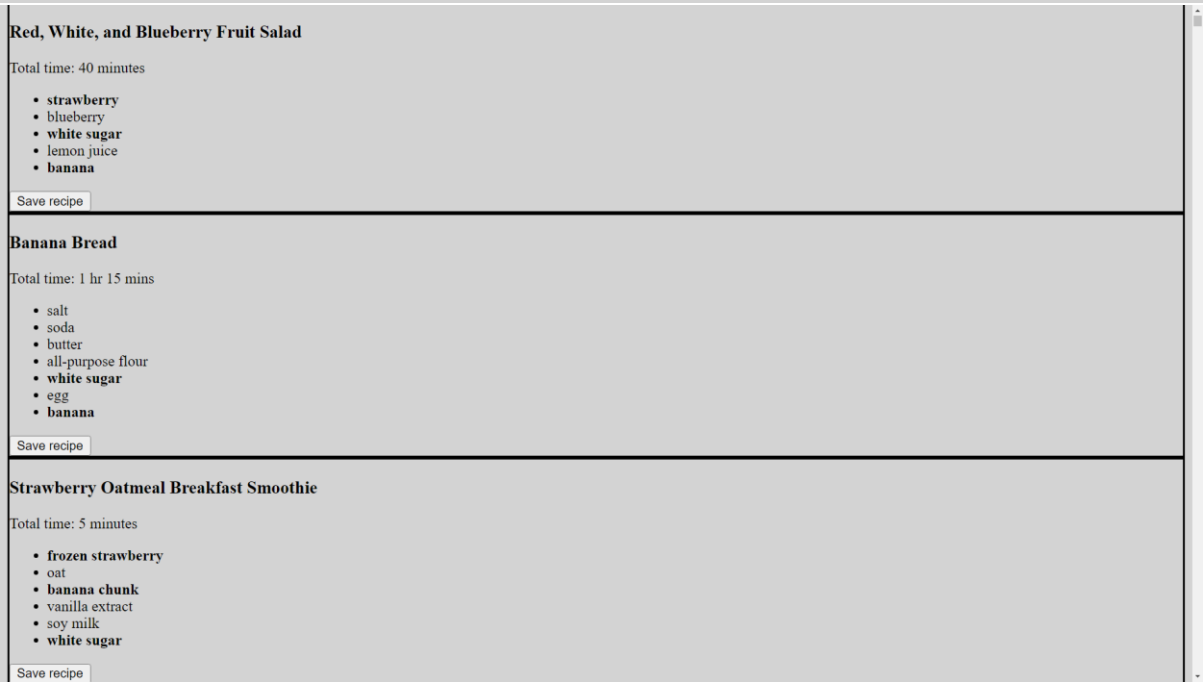
Conclusion

In conclusion, my project successfully fulfils the objectives laid out in my analysis. It is a website that can recommend a variety of relevant recipes given a list of ingredients, and as such meets the needs of my clients. The API and the versatility of web scraping present clear opportunities to develop the project further, but the project has met the requirements.

Appendix – screenshots

A few screenshots showing various parts of the website and the results of some searches





Home Info Profile API

Info

To use this search, simpy enter a list of space-separated ingredients in the "ingredients" field of the form. Do not include any other kind of seperator in the field, only spaces.

Searching

You can optionally include a maximum time that recipes can take to prepare, this must be an integer greater than 0.

You can also determine what the results will be sorted by, and limit the number of returned recipes.

Results

After you have made a search you will be shown the results. Ingredients that match your search will be in bold. Clicking on any of the recipes will take you to the webpage for it.

Accounts

You can also create an account here. This account will enable you to save recipes.

Saving recipes

One you have crated an account simply click the "save recipe" button on any recipe after making a search to save it to your account.

Home Info Profile API

API

This website provides an API alternative to make searches for recipes.

It uses the same functions as the search done through the web page but takes inputs as URL parameters and returns JSON data.

/api/search

Parameter	Type	Required	Purpose
ingredients	String	Yes	A space-separated list of ingredients
max_time	Int	No	The maximum total preparation time that returned recipes can have
sort_mode	String	No	How to sort the returned recipes
limit	Int	No	The number of results to return after they have been sorted

Sort mode	Description
Relevancy	Sorts the recipes by how relevant they are to the query
title	Sorts the recipes alphabetically by title
total_time	Sorts the recipes by the time they take to make

Example returned JSON

This shows one returned recipe, although more would likely be in the "results" list.


```
{
    "results": [
        {
            "ingredients":[
                {
                    "name":"leek"
                },
                {
                    "name":"salt ground black pepper"
```

Parameter	Type	Required	Purpose
ingredients	String	Yes	A space-separated list of ingredients
max_time	Int	No	The maximum total preparation time that returned recipes can have
sort_mode	String	No	How to sort the returned recipes
limit	Int	No	The number of results to return after they have been sorted

Sort mode	Description
Relevancy	Sorts the recipes by how relevant they are to the query
title	Sorts the recipes alphabetically by title
total_time	Sorts the recipes by the time they take to make

Example returned JSON

This shows one returned recipe, although more would likely be in the "results" list.


```
{
    "results": [
        {
            "ingredients":[
                {
                    "name":"leek"
                },
                {
                    "name":"salt ground black pepper"
                },
                {
                    "name":"unsalted butter"
                }
            ],
            "relevancy":0.0761111692806109,
            "title":"Easy Sauteed Leeks",
            "total_time":26,
            "url":"https://www.allrecipes.com/recipe/8472762/easy-sauteed-leeks/",
            "website":"allrecipes"
        }
    ]
}
```


Alex Davis

Logout

Change email

Current email

New Email

Password

Update email

Change password

Email

Current password

New password

Change password

Saved recipes

Paella II

Save recipe

Chipped Beef Cheese Ball

Total time: 10 minutes

- **onion**
- **celery**
- **cream cheese**
- **celery salt**
- beef
- paprika desired

Save recipe

Sautéed Leeks in Butter and White Wine

Total time: 17 minutes

- salt
- unsalted butter
- dry white wine
- **leek**
- fresh parsley garnish

Save recipe

Ian's Potato-Vegetable Soup

Total time: N/A

- water
- salt
- **onion**
- **celery**
- ground black pepper
- potato
- olive oil