MealWize

Department of Computer Science, Virginia Tech

CS 5934: Capstone Project

Instructor: Sara Hooshangi

Authors: Seren Episcopo, Connor Wyman, Kara Probasco, Anant Sharma, Akanksha Singh

December 11, 2024

Table of Contents

Demo Link

https://code.vt.edu/cs-5934-fall24/mealwize/cs-5934-mealwize/-/wikis/Final-Demo

Product Description

MealWize is a web application designed to help users explore new recipes, track their dietary goals, and maintain a healthy lifestyle - all in one place. What sets our platform apart is its ability to combine recipe discovery with personalized diet tracking while allowing users to upload and share their own unique or culturally significant recipes, fostering a community of inclusivity and diversity. Users can search, favorite, and manage recipes while seamlessly tracking calories and macronutrients to stay aligned with their health objectives. With features like tailored recipe recommendations, an integrated shopping list, and the ability to log custom meals, our app simplifies meal planning and empowers users to maintain a balanced lifestyle. It's perfect for anyone who values a unique, user-friendly solution that blends health tracking with recipe sharing.

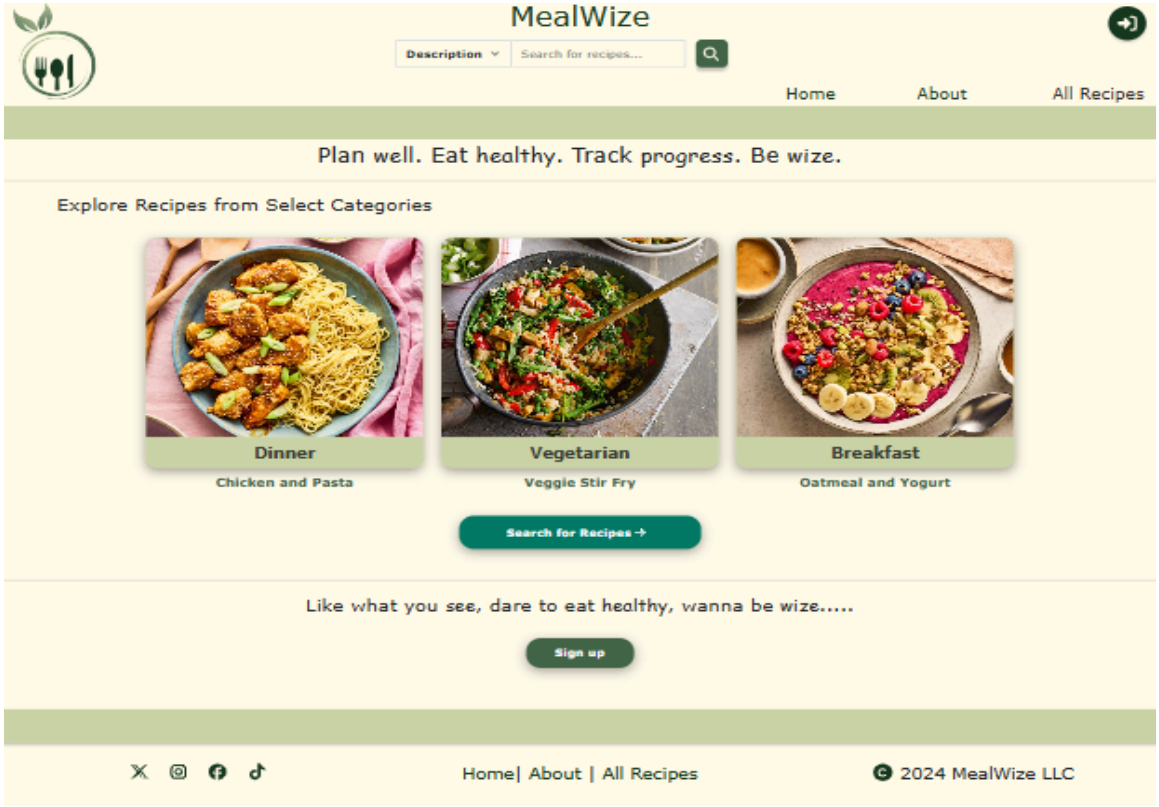
Product Functionality

For product functionality, there were seven main features that were completed for our project. within three sprints.

The Home and About pages were created with the purpose of serving as the storefront for the application and informing users of the purpose of the application.

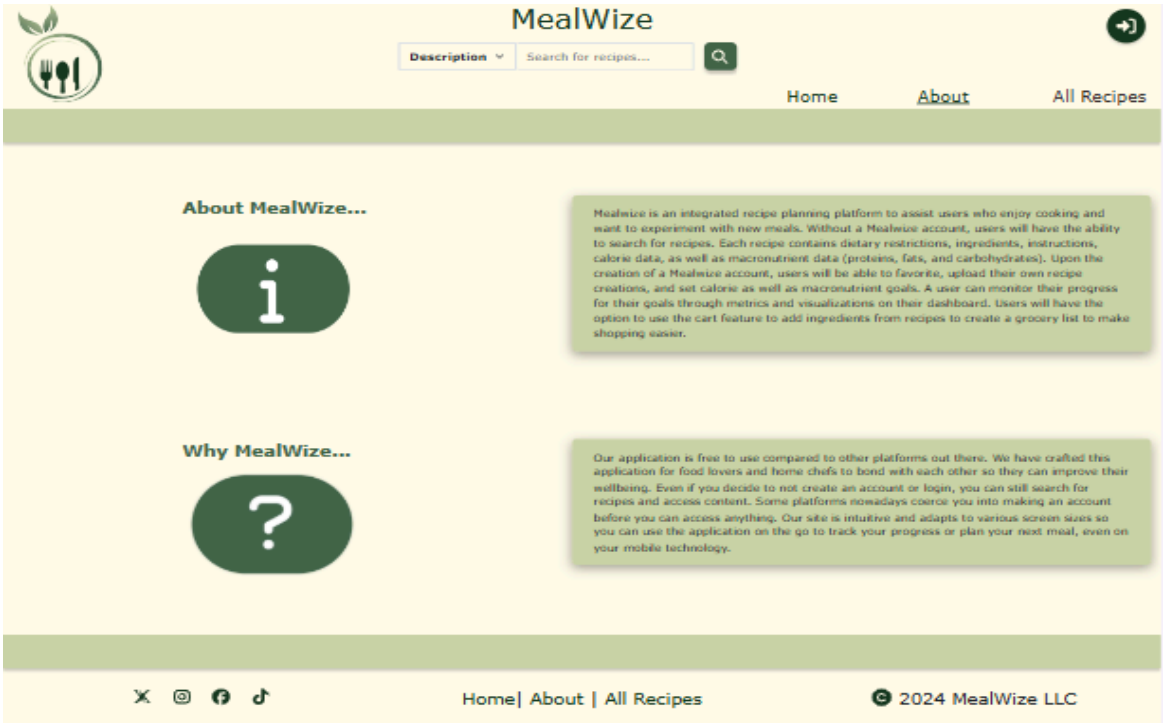
Home Page

The Home page is where users can use the search bar at the top to search for recipes in four different categories: Description, Source, Title, and Ingredients. On the Home page, users can locate the Sign up button to create an account or click the Login icon button at the top right of the page to login.



About Page

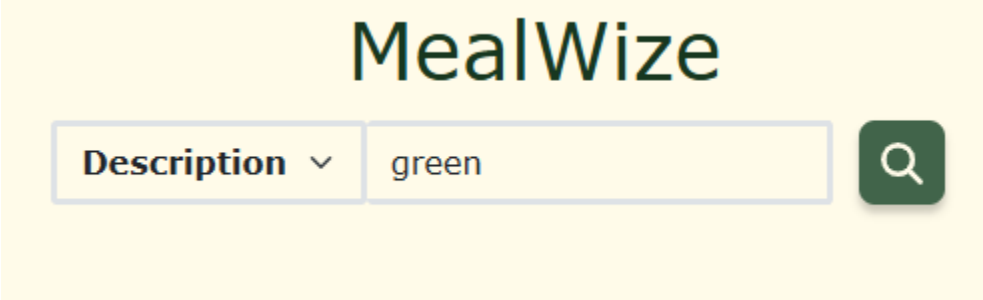
The About page is where users can learn more about the features of the MealWize application.



Search/View Recipes Feature

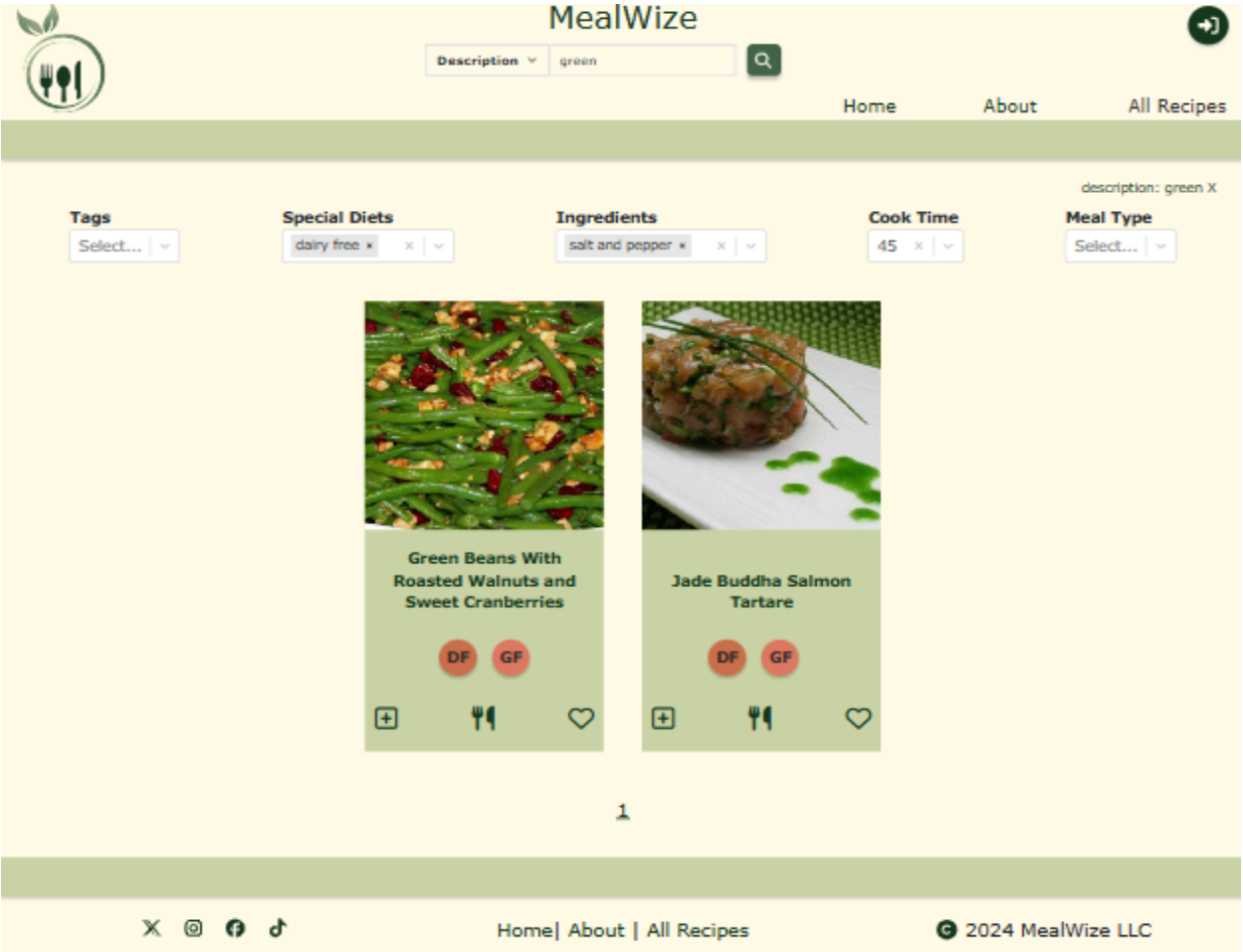
Search Bar

Users can search for recipes using the search bar in the header of each page on the site. A user can search for and view recipes without a MealWize account. After clicking the Search icon button, the user will be redirected to the All Recipes page.



All Recipes Page

On the All Recipes page, users can view recipe cards that show the recipe title, dietary restriction labels, and recipe image. Users can filter the recipes using the five multi select boxes: Tags, Special Diets, Ingredients, Cook Time, and Meal Type.



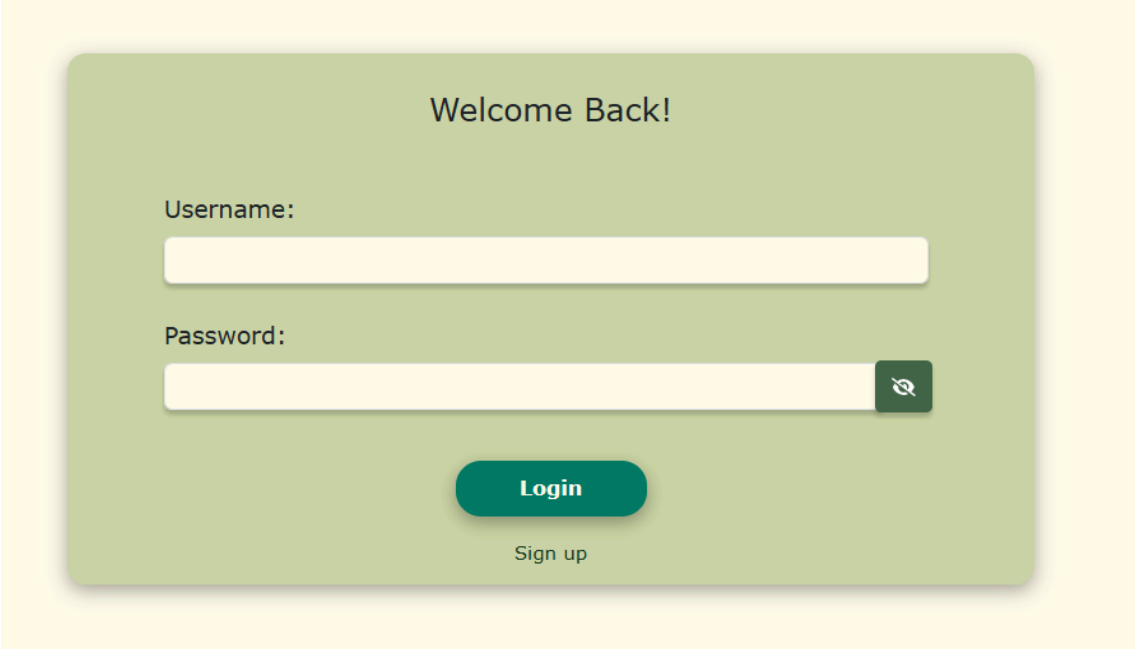
View Recipe Page

Users can click on a recipe card to view a recipe. When viewing a recipe, a user can see the recipe title, description, dietary restrictions, macronutrients, ingredients, and instructions.

User Account Feature

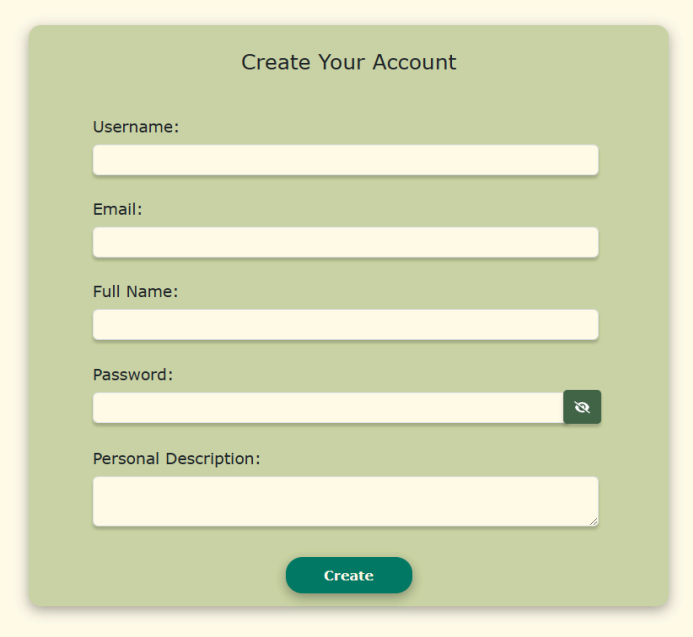
Login Page

If a user has a MealWize account, the user can sign into the account with a username and password. There is a Login icon button at the top right corner of the site to direct users to the Login page.



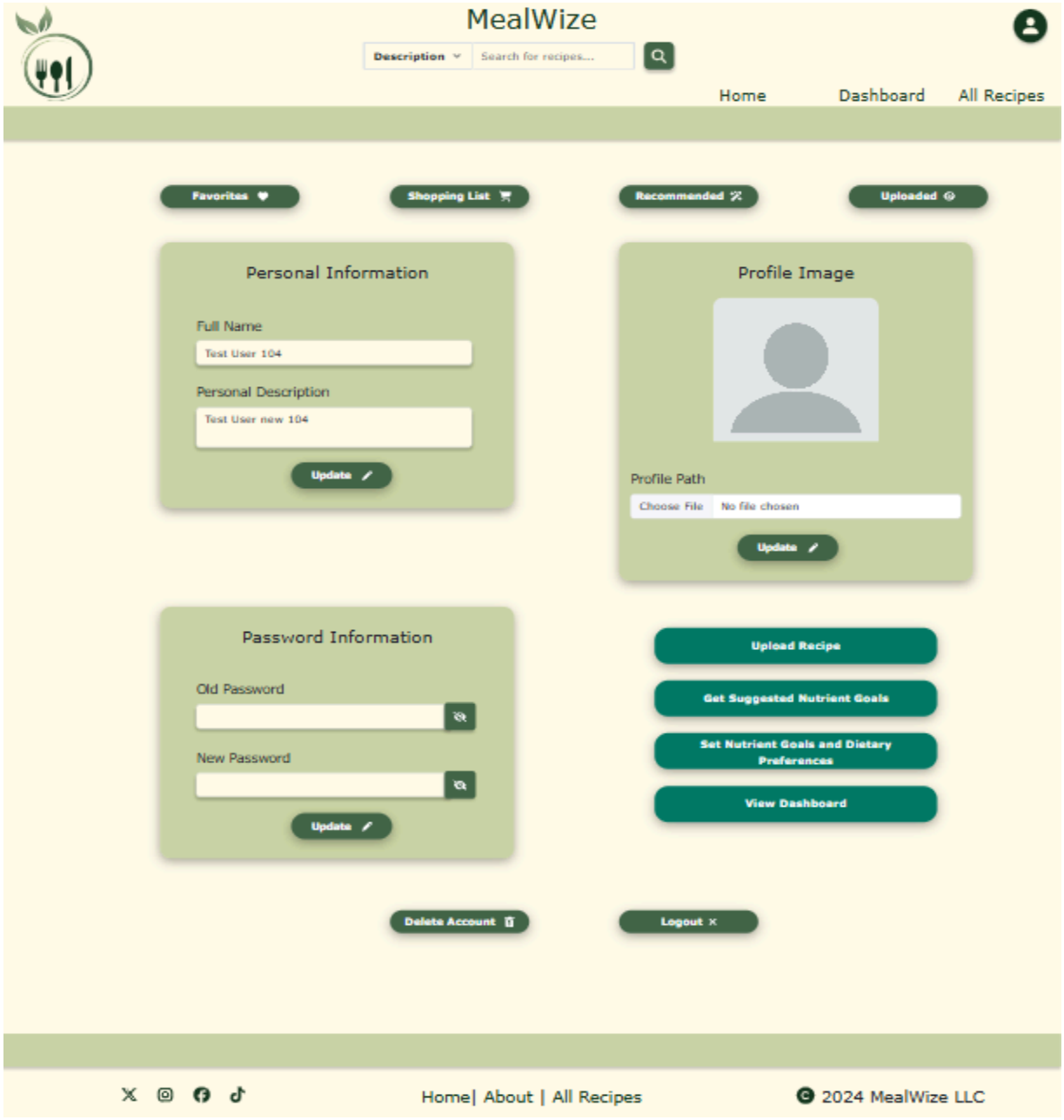
Create Your Account Page

If a user has no MealWize account, then the user can click the Sign up link on the Login page to create an account. The user has to fill in the username, email, full name, password, and personal description fields for the account to be created before hitting the Create button.



Profile Page

When a user successfully creates an account or logs into the account, the user will be directed to the Profile page. On the Profile page, they can update profile information like personal details, password, and profile image. A user will have access to additional features when logged into the application.



A user can select the Delete Account button on the profile page to delete an account. Once an account is deleted, the account will not be able to be recovered. A user can select the Logout button to logout of the account.

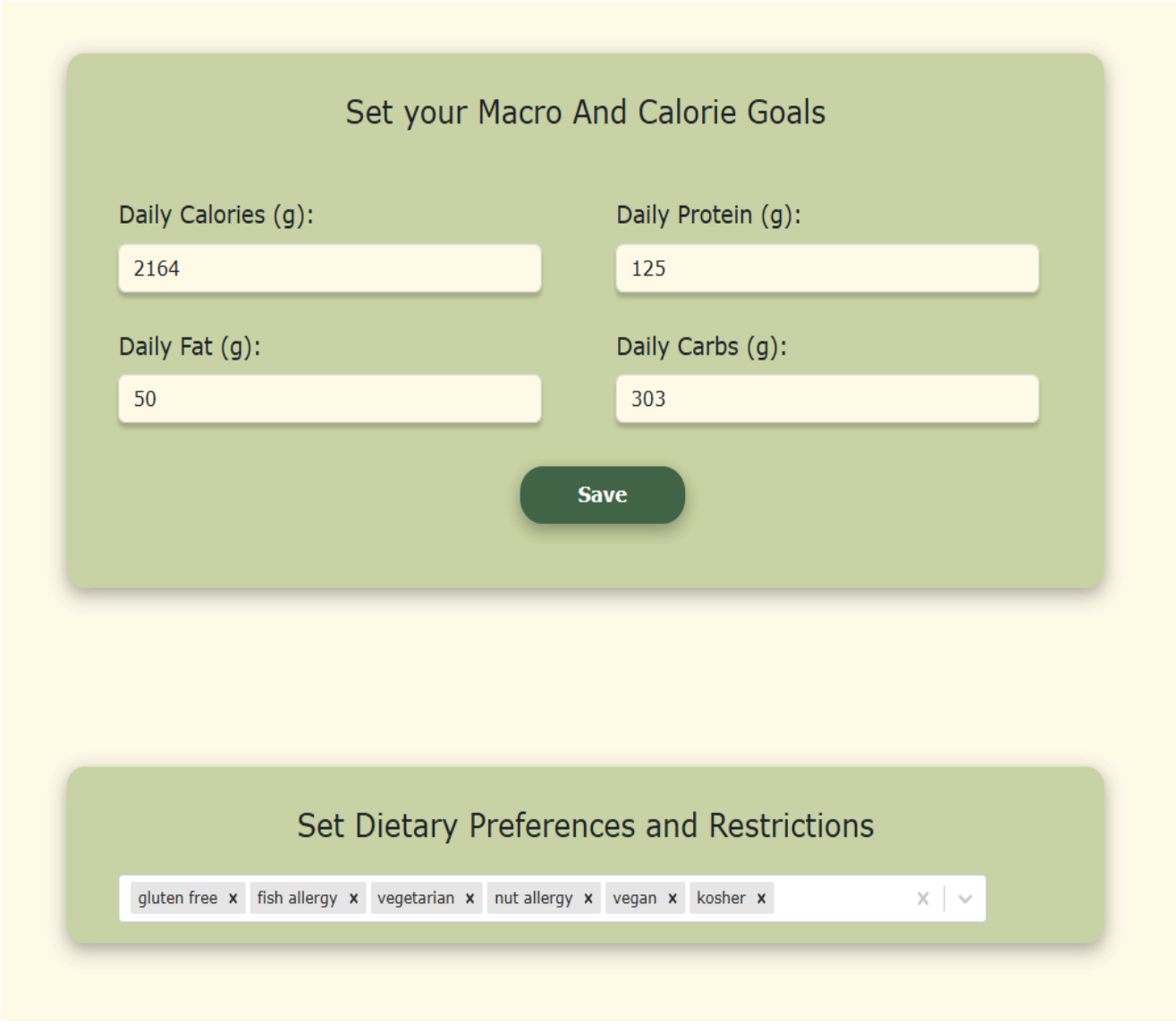
Calorie and Macronutrient Tracking Feature

Suggested Macro and Calorie Goals Page

From clicking the Get Suggested Nutrient Goals button on Profile or Dashboard pages, a user can get suggested macronutrient and calorie goals. A user will have to enter height, weight, age, activity level, overall goal, and gender to view output for suggested goals at the bottom of the form.

Set Nutrient Goals and Dietary Preferences Page

From clicking the Set Nutrient Goals and Dietary Preferences button on Profile or Dashboard page, a user can set macronutrient and calorie goals. A user will have to enter calories, protein, fat, and carbs. On the bottom of the page is an option to set dietary preferences and restrictions. To add a preference, a user must select the preference from the dropdown for the preference to appear in the multi-select field. To remove a preference, a user has to click the x icon on the Preference button.

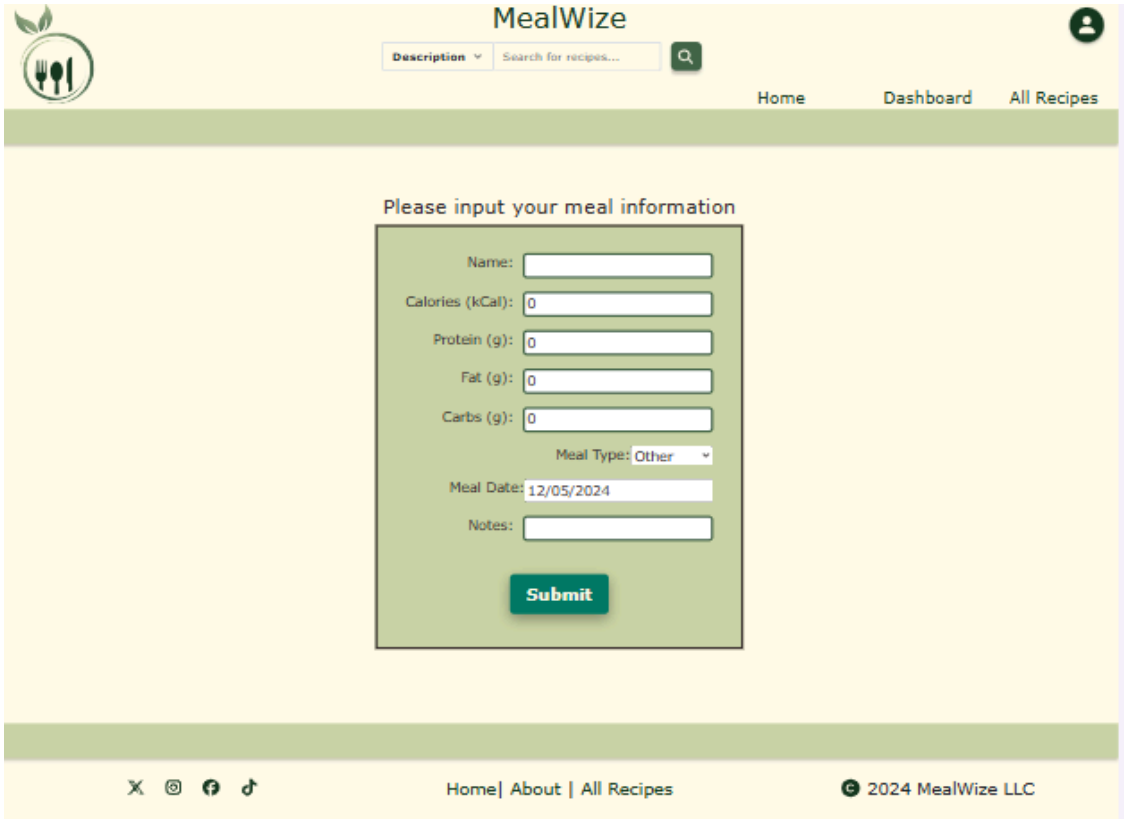


Dashboard page

Users can track their progress for their macronutrient and calorie goals on the dashboard. There is a Weekly Calorie Tracker for users to track their progress for their daily total calories consumed over the course of one week. A user can visualize their progress for fats, carbs, and proteins within the Weekly Calorie Tracker as well. In the Daily Macro Goal progress circular charts, users can see their daily progress for each of their macronutrient goals. On the bottom of the dashboard is a table which shows the user's logged meals. Within each row of the table, a user can delete or edit a logged meal.

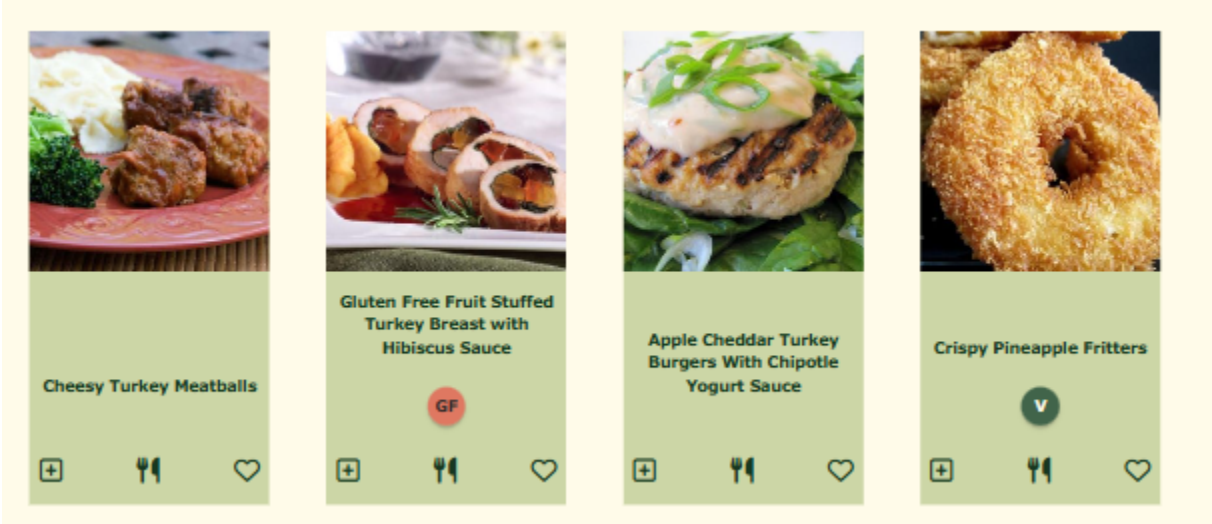
Add a Meal Page

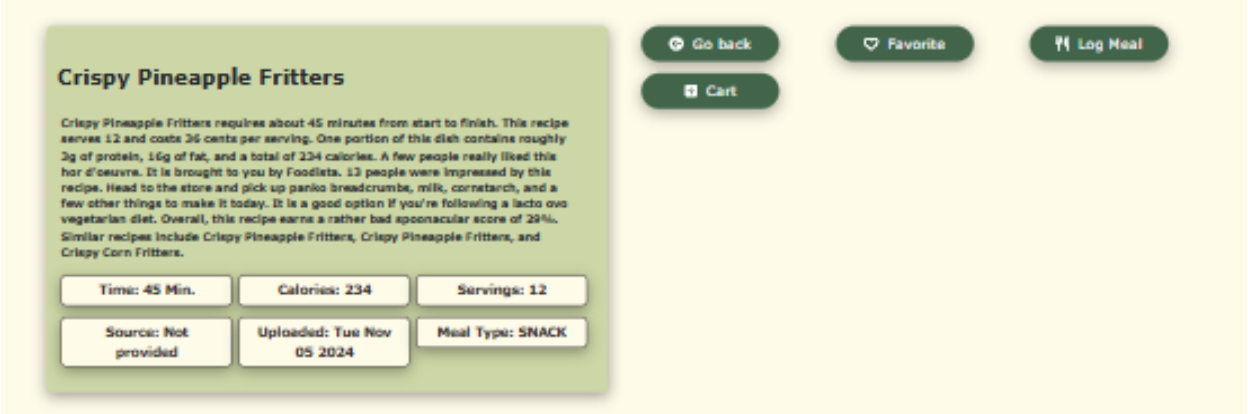
To add a meal, a user can click on the Add Meal button on the Dashboard page. The Add a Meal page is for tracking nutrients and calories for meals or recipes not within the application. After adding the meal, the Dashboard page will update accordingly with the meal.



Log Meal Page

To log a recipe as a meal, a user can click on the Dining Utensil icon located on the recipe card or the Log Meal button on the View Recipe page.





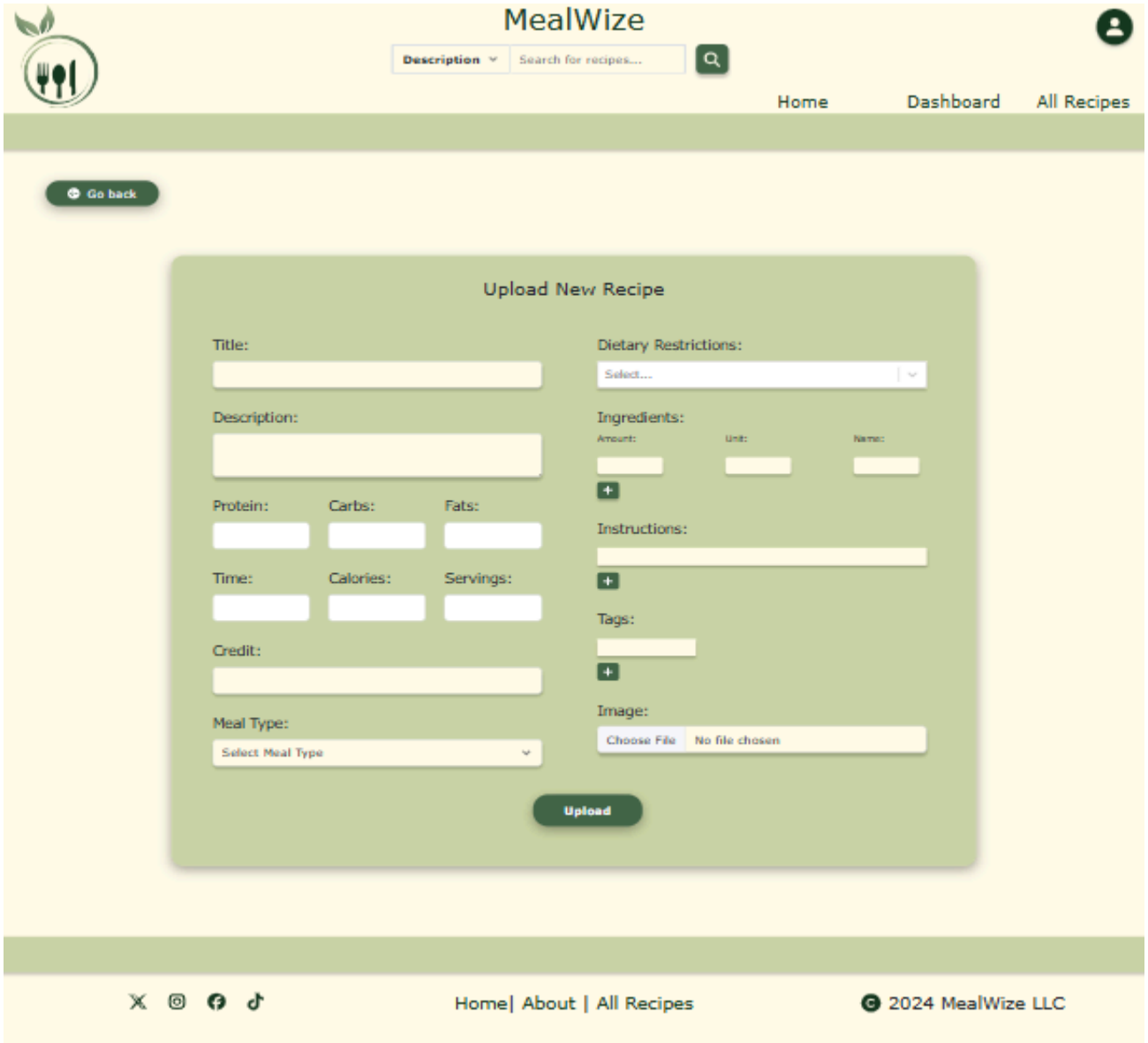
After clicking on those icons, a user will be directed to the Log Meal page. The user has the option to enter the amount of servings, date, and any notes associated with the recipe before hitting the Submit button.

After the user hits submit, the Dashboard page will update accordingly with the newly added meal.

Sharing Recipes Feature

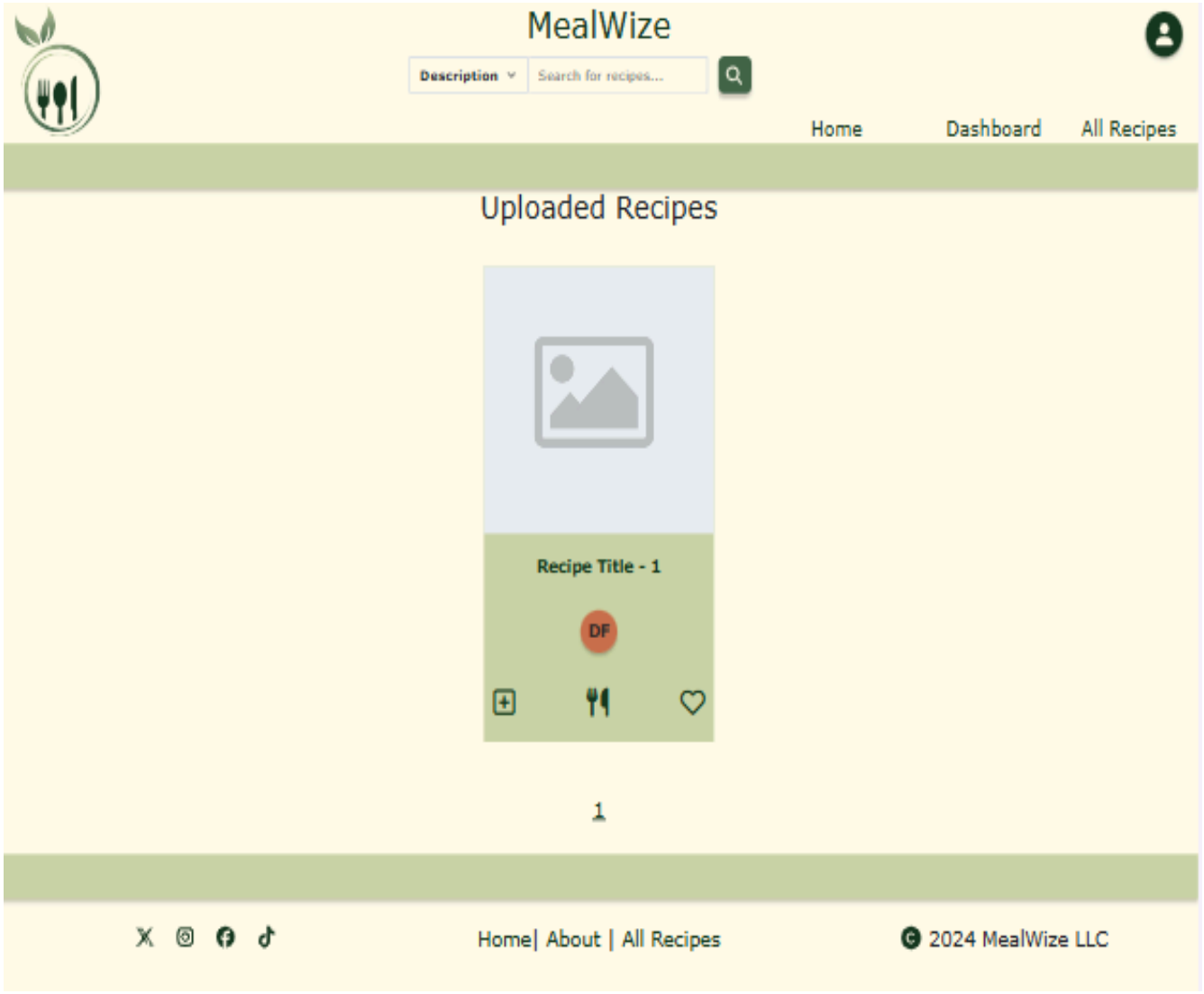
Upload Recipe Page

Users can share recipes by clicking on the Upload Recipe button on the Profile page. A user will then be prompted to enter details like title, meal type, dietary restrictions, ingredients, instructions, and tags. The user has to hit the Upload button to share recipes with others as the last step to share the recipe.



View Uploaded Recipes Page

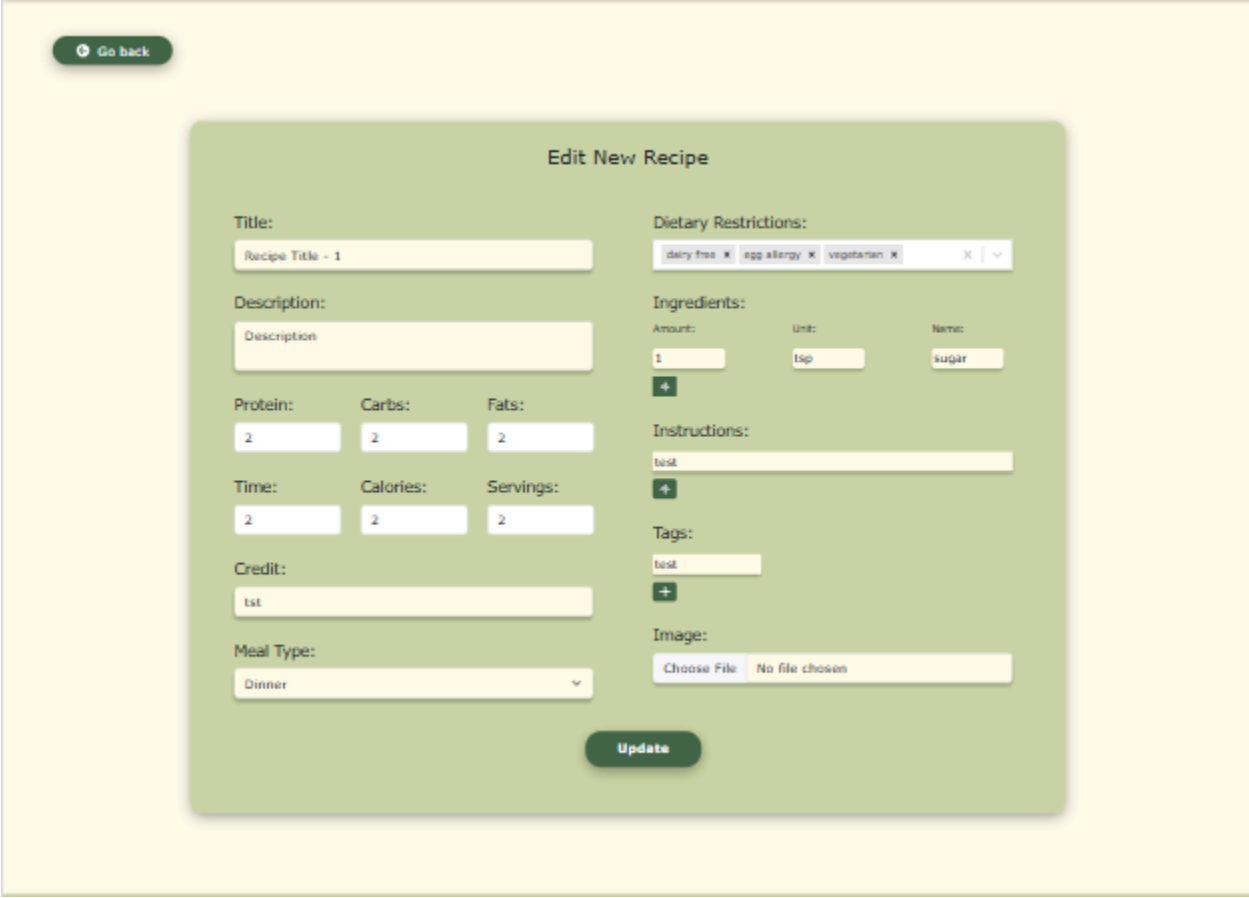
A user can view uploaded recipes by going to the Profile page and clicking on the Uploaded Recipe button. For a user to view an uploaded recipe or edit the uploaded recipe, the user has to click on the recipe card.



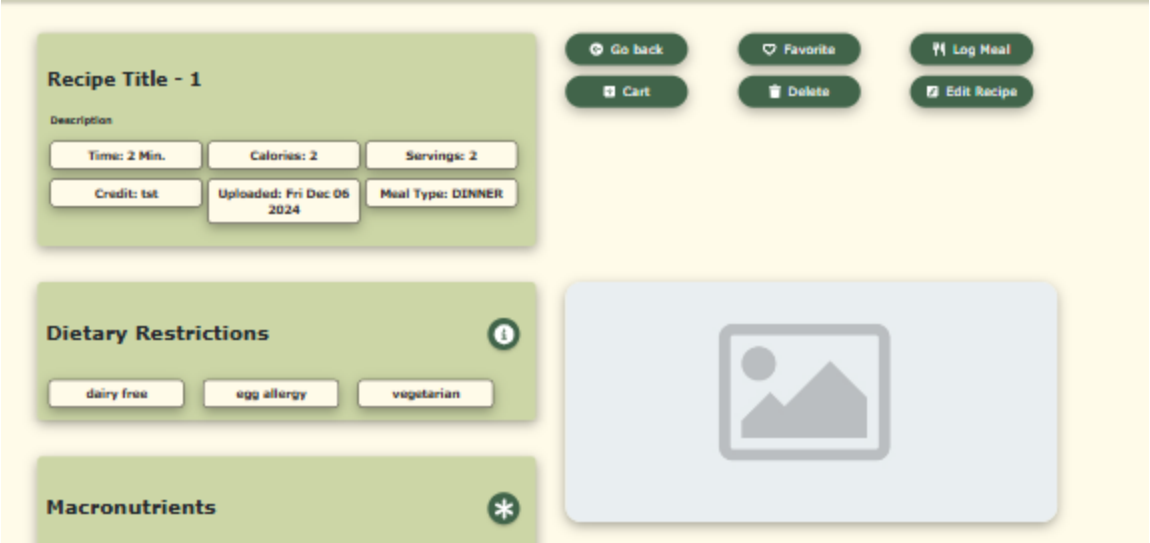
Edit Recipe Page

To edit a recipe, the user has to first click on the Edit Recipe button when viewing the recipe.

The user will then be able to edit the recipe. The user will have to click the Update button when editing the recipe is completed.



After editing the recipe, a user will return to viewing the recipe. In addition to editing a recipe, a user will also be able to delete the recipe.



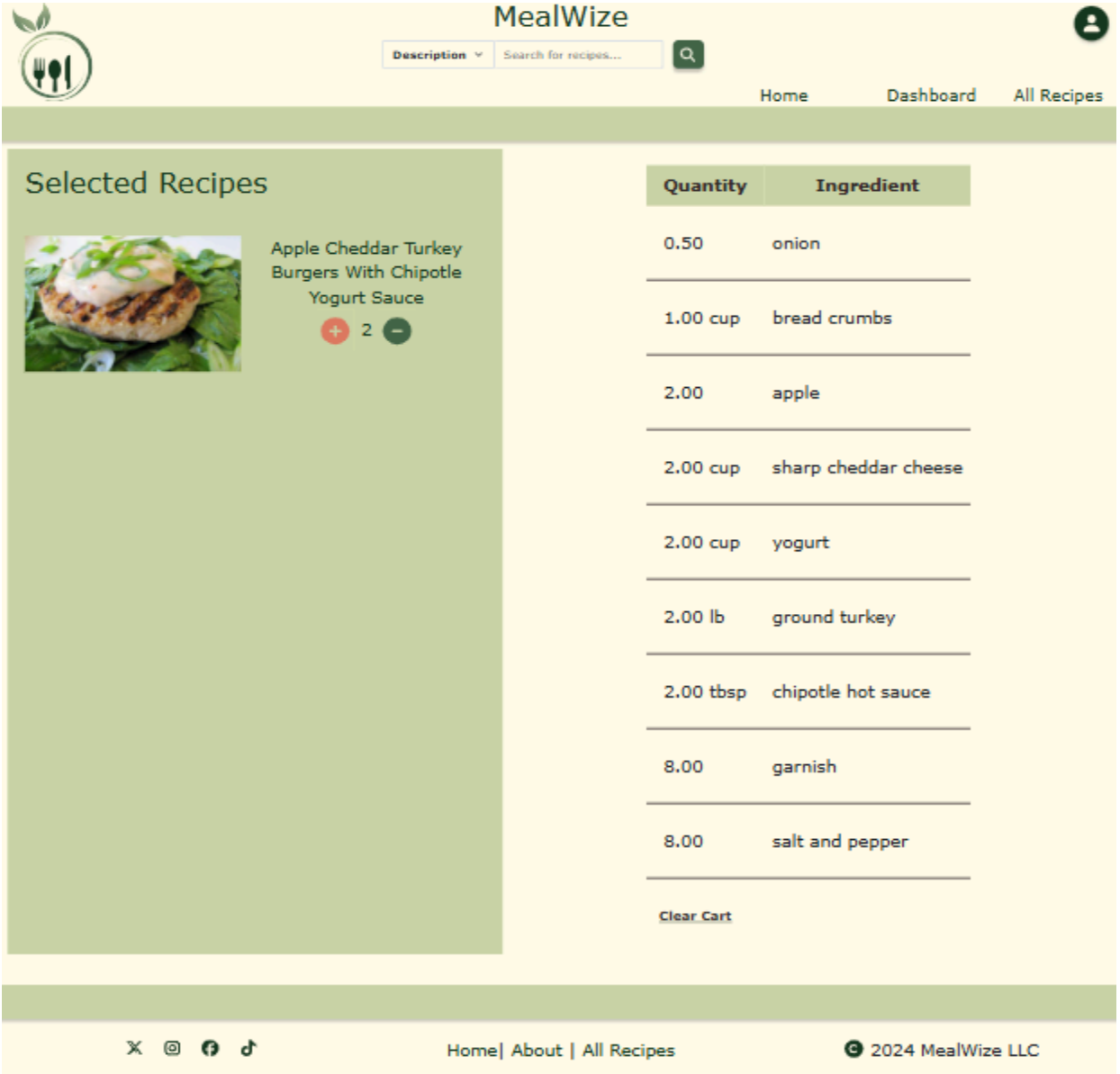
When clicking on the Delete button, the user will be prompted to confirm the action to delete the recipe.

After deleting a recipe, the recipe will no longer appear in the Uploaded Recipes page and be shared with other users.

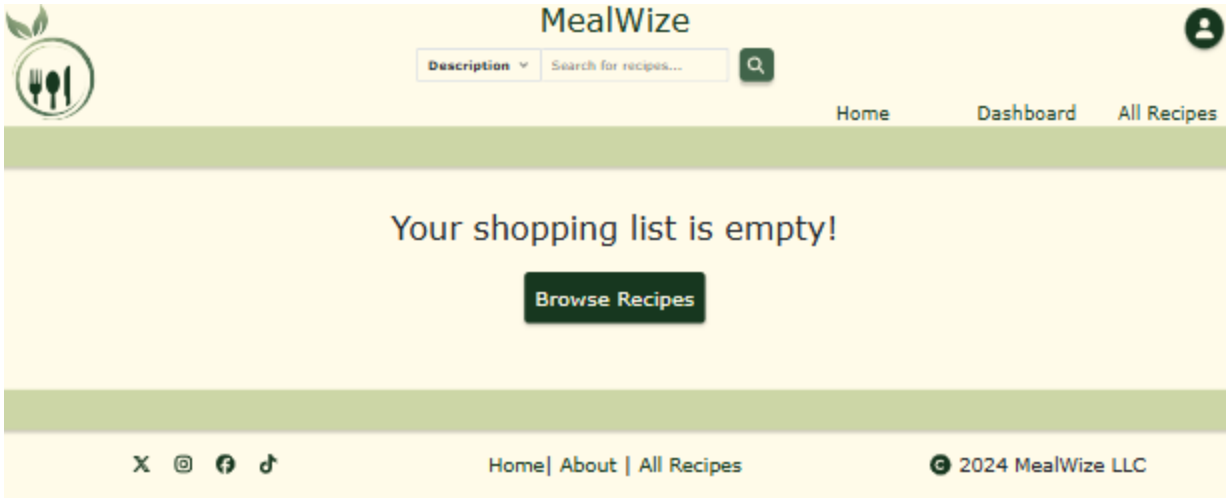
Shopping List Feature

Shopping List Page

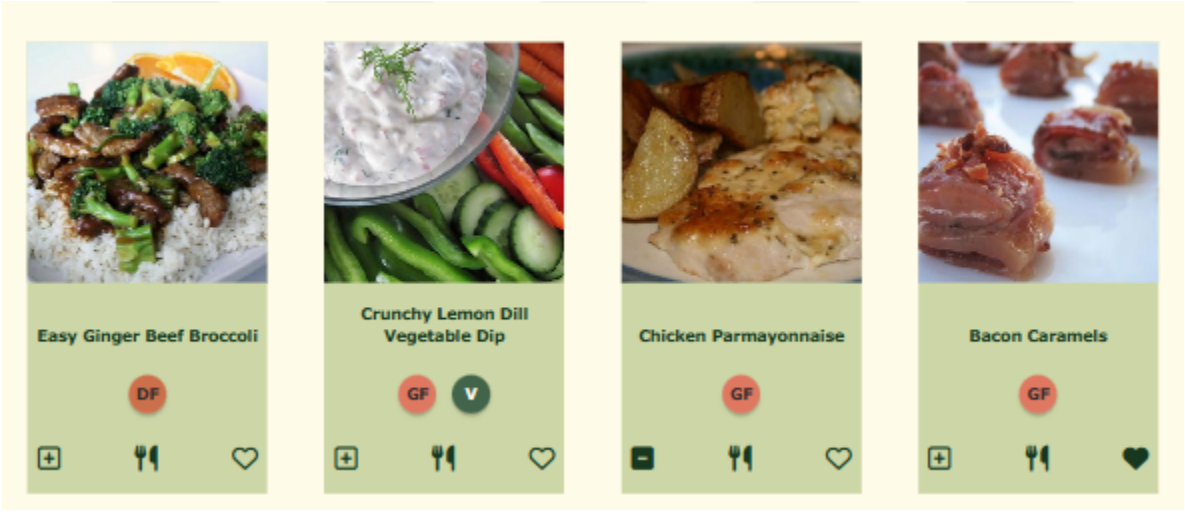
A user can view a shopping list by clicking on the Shopping List button which is located on both the Profile and Dashboard pages. There is an option to increment or decrement the amount of recipe batches on the left side of the page. On the right side of the page, is a table to view the quantities and ingredients for the recipes.



To clear the shopping list, a user can click on the Clear Cart link. The user will then be prompted to browse for recipes after clearing the cart.



When browsing for recipes, users can click on the Plus icon to add the recipe to the shopping list or Minus icon to remove the recipe from the shopping list.



When viewing a single recipe, a user can click on the Cart button to add the recipe to the shopping list and re-click the Cart button to remove the recipe from the list.

Recommended Recipes Feature

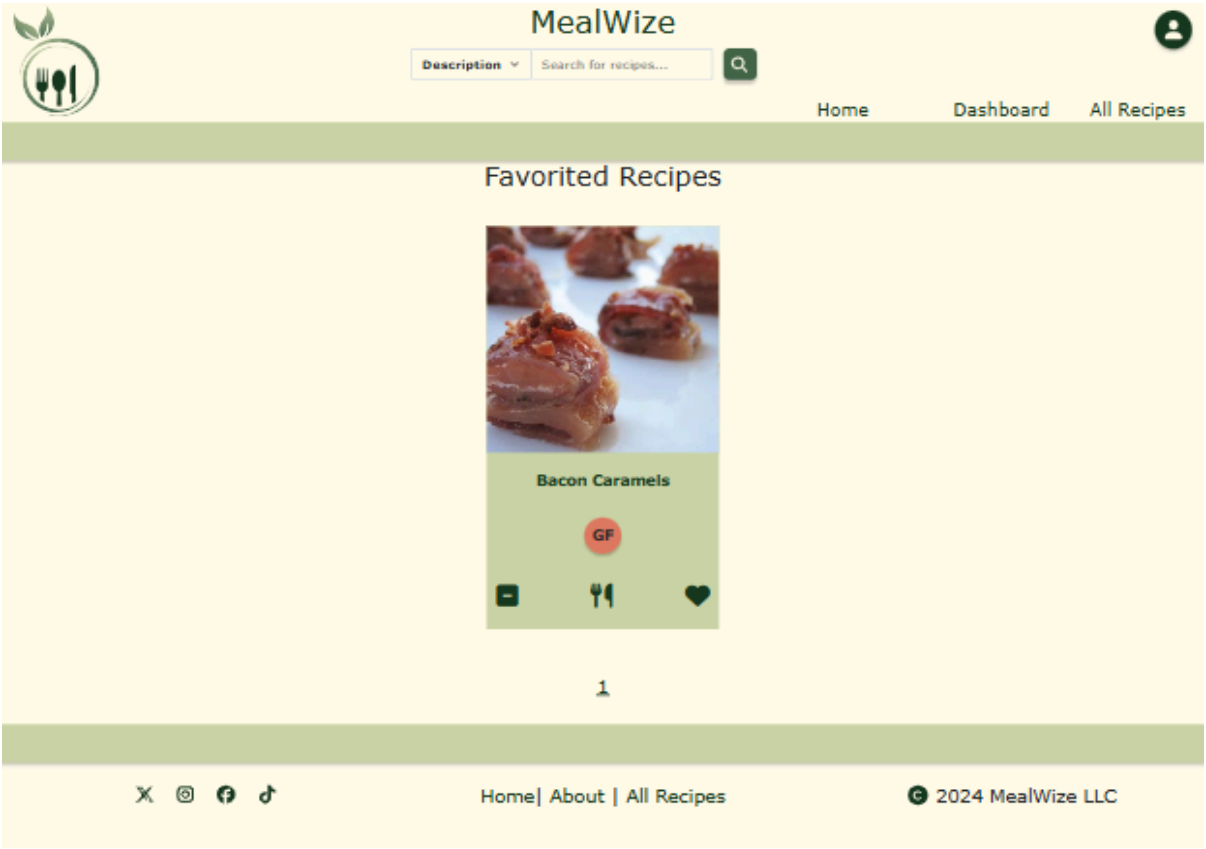
Recommended Recipes Page

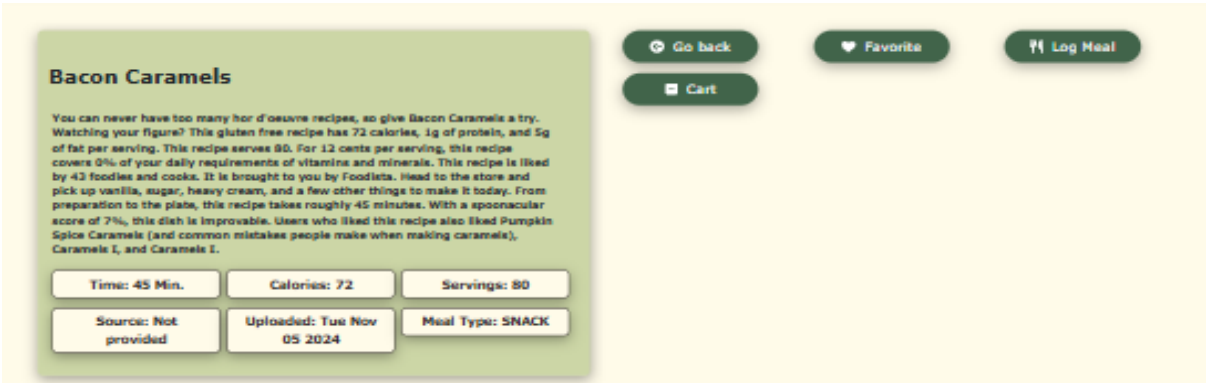
Users can view recommended recipes by clicking the Recommended button on either the Profile or Dashboard pages. Please see the Recommendation System section of the report.

Favorited Recipes Feature

Favorited Recipes Page

Users can find recipes they have favorited by clicking on the Favorites button either on the Profile or Dashboard pages. To favorite a recipe, a user can click on the Heart icon located on the recipe card when searching for recipes or the Favorite button on the View Recipe page. To unfavorite a recipe, a user can just re-click the Heart icon or Favorite button.

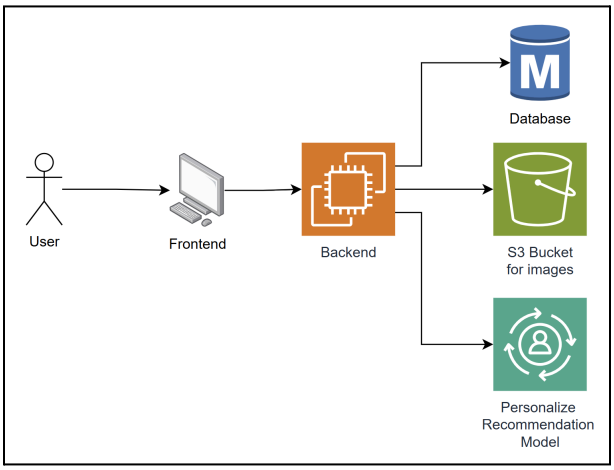




Design

Architecture

MealWize uses a common website architecture consisting of a frontend, backend, and database. The frontend uses TypeScript and React, the backend is written in Java with Spring Boot, and the database runs MySQL. The backend connects to two additional AWS endpoints: an S3 bucket stores user-uploaded images and our Personalize campaign provides personalized recipe recommendations.



MealWize high-level architecture.

We chose this architecture because MealWize is a traditional web application. It is a tried and true design and we don't require any novel architecture design. The languages and tools we used were inspired by the team's previous experience. Some of us took the Web Application Development course over the summer, which taught using React and Java. Spring Boot was brought in from our experience in the workforce and jobs that use it.

AWS was the best fit for hosting our group's project for several reasons. First of all, several members of the team were already familiar with it from other classes or their jobs. Virginia Tech offering to provide us the AWS services we needed was also a significant factor. Finally, AWS provides all the services we need to run the app from one cloud provider.

EC2 is AWS' traditional server offering in the cloud. This is where we hosted our backend and from where we served our frontend. We hosted our MySQL database on AWS RDS. S3 stores files, which we used for storing profile and recipe pictures. Personalize made it easy to create our own custom recommendation system to suggest new recipes to our users.

Frontend Design

The MealWize website was built in Typescript with React and Bootstrap. By developing the website in Typescript, we ensured data passed between components and received through the backend APIs were typed correctly. This reduced the risk of unnoticed type-related errors remaining in the final website. React was chosen as our development framework since it has high reusability between components and our team has prior experience using it. Bootstrap was used for certain pages because it allows for a responsive user interface.

In this section, we will highlight a series of design decisions within the MealWize frontend that are vital to the application's functionality. Since we did not use a template to build MealWize, these decisions set MealWize apart from the vanilla templates that currently exist.

Authentication and Route Protection

MealWize requires an account to access many of the pages and features it offers. To make sure unauthorized users did not access such pages, we created a protected route component, PrivateRoute, that routes the user to their desired location if they are logged in, or directs them to the login page if they are not logged in. PrivateRoute also relies on an authentication context to check the status of the user's token.

MealWize receives user tokens from the server when a user successfully logs in or creates a new account. This logic should be consolidated and accessible to all pages of the website due to many API calls requiring access to a valid user token. This ensures consistent usage and reduces the risk of security errors caused by improperly implemented authentication logic. Because of this, the login, logout, user account creation, and token validation logic was combined into a context. This context, called AuthContext, saves the user token to local storage and reads and deletes the user token from local storage. AuthContext also contains the user's token and various error statuses within stateful values that are also provided to all of the MealWize components through the context provider.

Recipe Card Lists and Component Reuse

As mentioned previously, component reuse was vital to our development process and design. An example of this is the recipe card list component that was used across the Recommended Recipes, the Search Recipes, and the Favorited Recipes pages. The recipe card list takes in a list of recipes returned from various API endpoints and generates a list of recipe cards to display to the user.

The recipe card also has buttons to add the recipe to the cart, add the recipe as a meal, or favorite the recipe. The icons for favoriting and adding a recipe to the cart also toggle between selected and not selected appearances, which requires multiple state variables and the use of a context that handles adjusting the recipes and ingredients in the shopping cart. Overall, the recipe card handles multiple user interactions and API calls.

Since the recipe card is used across multiple pages, reusing the same component instead of duplicating the code in multiple files decreases the chances of errors and overall development time.

Forms and Security

Forms are used to get user input and validate the fields before performing some action, such as making an API request. For MealWize, Bootstrap Forms and Formik with Yup are used to create dynamic forms that provide user feedback when certain conditions are not met. For pages that use Formik, including the pages for creating custom and recipe meals, Yup is used to easily validate field values and print error messages. By validating user input, we prevent various types of cyberattacks, such as cross-site scripting and command injection attacks. Field validation was applied to bootstrap forms by making fields within the forms required, checking password inputs, and notifying users of errors when logging in or when an invalid password is entered.

MealWize Layout and Appearance

When planning the design and aesthetics of MealWize, we focused on how to evoke the ideas of health and calmness. After looking at various color palettes, we decided green monotone encapsulated those two ideas perfectly. Paired with a light beige background and accents of orange and blueish-green, the color scheme for MealWize was finalized. The layout for MealWize also went through multiple iterations. We wanted to ensure users could easily navigate throughout the website and locate the many features we included.

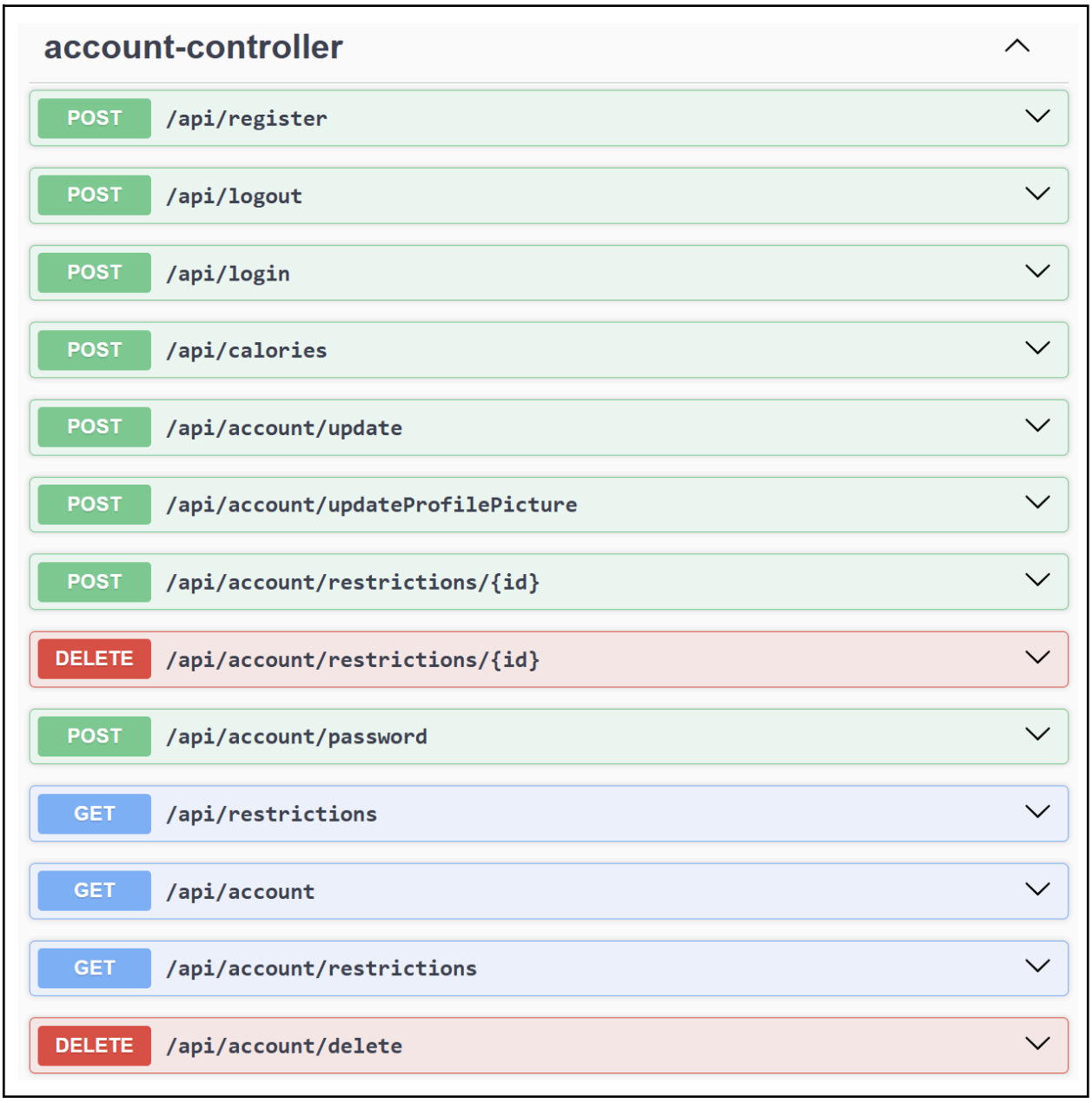
When designing the homepage, we wanted to direct users to both sign up for accounts and search for recipes, since these are the only features available to users who are not logged in. The buttons to navigate to those pages are very prominent and draw the user's eye. This design choice continued throughout the website, especially with the submit buttons. Although MealWize has a monotone color scheme, the size and color of buttons and other components was selected to enhance the user experience and overall usability of the website.

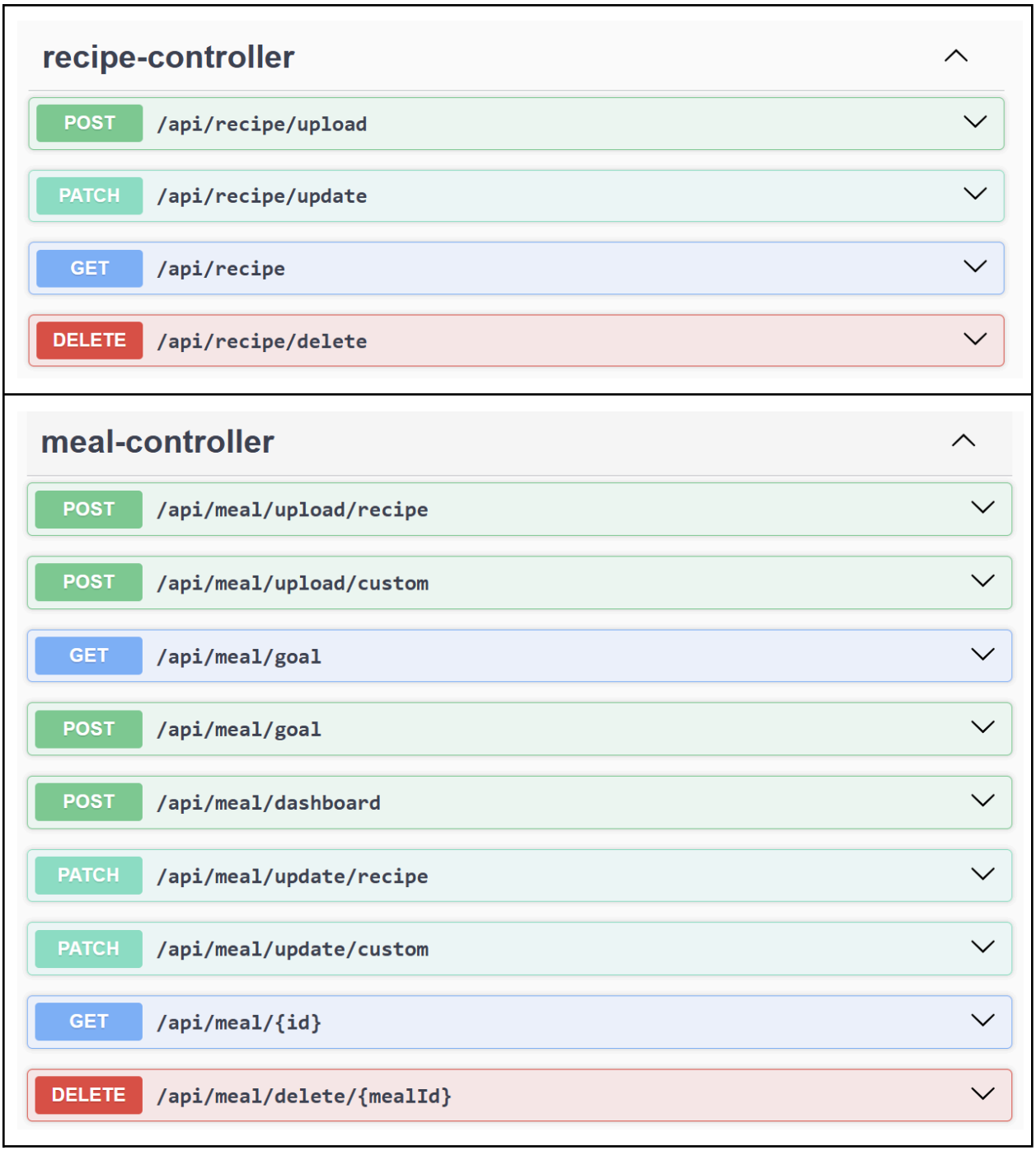
Backend Design

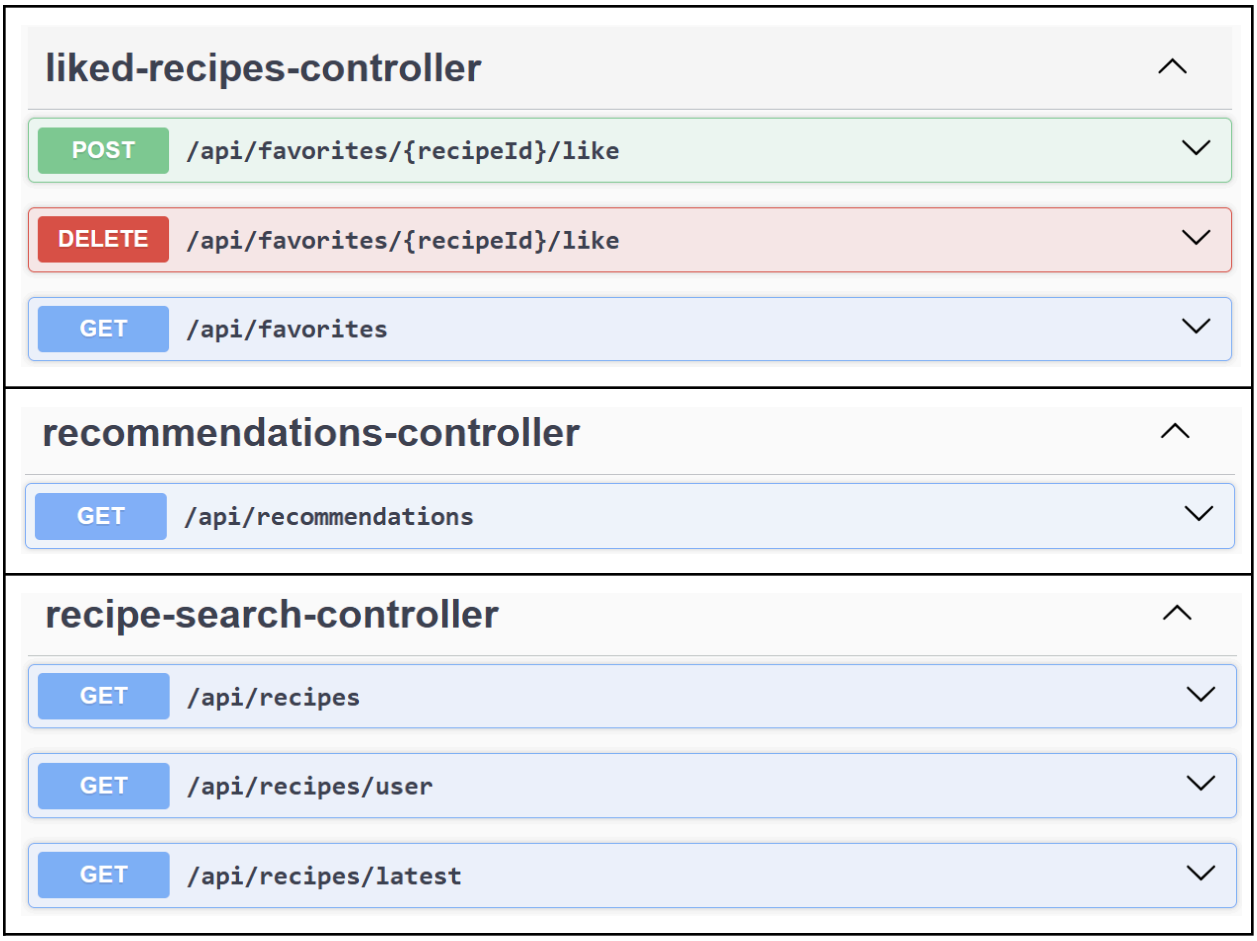
The MealWize backend is a REST API written in Java using the Spring Boot framework. The backend interfaces between the client frontend, database, and other AWS services. The primary responsibilities of the backend are data transformation, data validation, and enforcing user access control and authentication where necessary.

We chose Java and Spring Boot for our backend due to some of our team members' recent experience using them for other projects. Java is a popular language for creating APIs since it enforces typing and has a large ecosystem of libraries. It also has reasonable

performance and it is cross-platform since it runs on the JVM instead of natively. Spring Boot is a Java framework that provides support for many different features in different libraries. We used libraries that support web application development, security, and database interaction. To manage our dependencies, we used Maven. We also used libraries outside of the Spring Boot ecosystem to support authentication and to connect with our AWS services.




recipe-controller
POST /api/recipe/upload
PATCH /api/recipe/update
GET /api/recipe
DELETE /api/recipe/delete
meal-controller
POST /api/meal/upload/recipe
POST /api/meal/upload/custom
GET /api/meal/goal
POST /api/meal/goal
POST /api/meal/dashboard
PATCH /api/meal/update/recipe
PATCH /api/meal/update/custom
GET /api/meal/{id}
DELETE /api/meal/delete/{mealId}



List of backend API endpoints divided by class.

Class Structure

Our backend architecture is divided into six sets of classes. Some of these are focused on business logic, others are for data formatting, others are for interacting with the Spring Boot framework, and others are for calling external AWS services.

The first set of classes are the Controllers. These interface with the Spring Boot framework to receive incoming requests to the API after they have been authenticated. They automatically convert the provided HTTP request into Java objects, which are then passed into the function as parameters. Depending on the case, we parse the request path, URL query parameters, and the request body. These classes coordinate subsequent operations at the highest level, including data validation, external calls, and error handling. These methods also control the value we return to the client, the HTTP status code, and any error messages.

Next are the DAO (Data Access Object) classes. These use the eponymous pattern, wrapping external calls with a class to abstract away the implementation details. These classes are responsible for querying the database and processing any associated errors.

These operate at the lowest level, directly calling the JDBC library we use to access the database.

The largest set of classes are our Record classes. These store data that our application uses. Many of these are used to properly format JSON data we receive from and return to the frontend.

We have two Config classes providing global configuration for the backend. One configures clients to connect to the database, S3, and Personalize. These clients are accessed by other classes using Spring Boot's dependency injection. The second configuration class configures Spring Boot's built-in web security for our application. Here we specify which paths require authentication, how users should be authenticated, and CORS restrictions. These classes read configuration values provided in the Spring Boot profile, which is specified at runtime. We specify default configuration values for the application in the application.properties file, and we can overwrite these values with different profiles. We configured a "cloud" profile, which uses values from application-cloud.properties to overwrite the default values and call our AWS database instead of a local database.

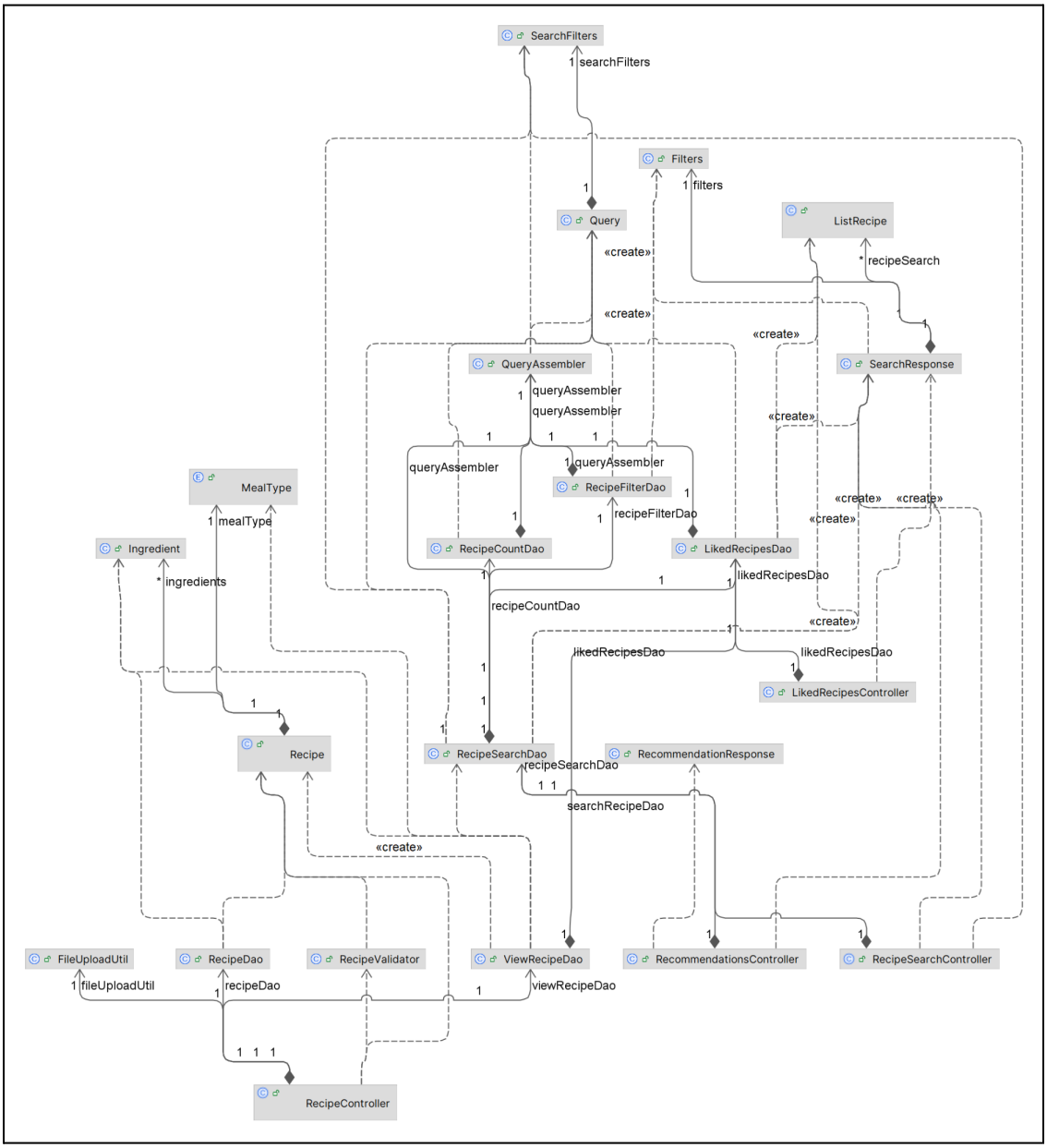
In addition to the security configuration class, we have three more classes dedicated to user authentication. Their responsibilities are interfacing with the Spring Boot infrastructure, verifying user credentials, generating session tokens, and enforcing authentication against incoming requests.

Finally, we have miscellaneous utility classes grouped together. These provide functionality that we didn't categorize with any other classes. Some of the functions provided by these classes include uploading files to S3, data transformation and validation, and assembling SQL queries.

Recipe Searching

One of our biggest features of MealWize is recipe searching. This is supported on the backend by SQL queries we developed to query the database depending on the information provided by the user. Depending on which field the user searches by (Description, Title, Ingredients, etc.) and which filters they have selected (Dietary Restrictions, Ingredients, Cook Time, etc.), we generate an SQL query that contains the specific fields the user is searching for. When the user searches for text from the search bar, we use MySQL's text matching algorithm's Natural Language Mode, which ranks results by their similarity to the search term. We then apply the user's desired filters to filter out results that don't fit their search.

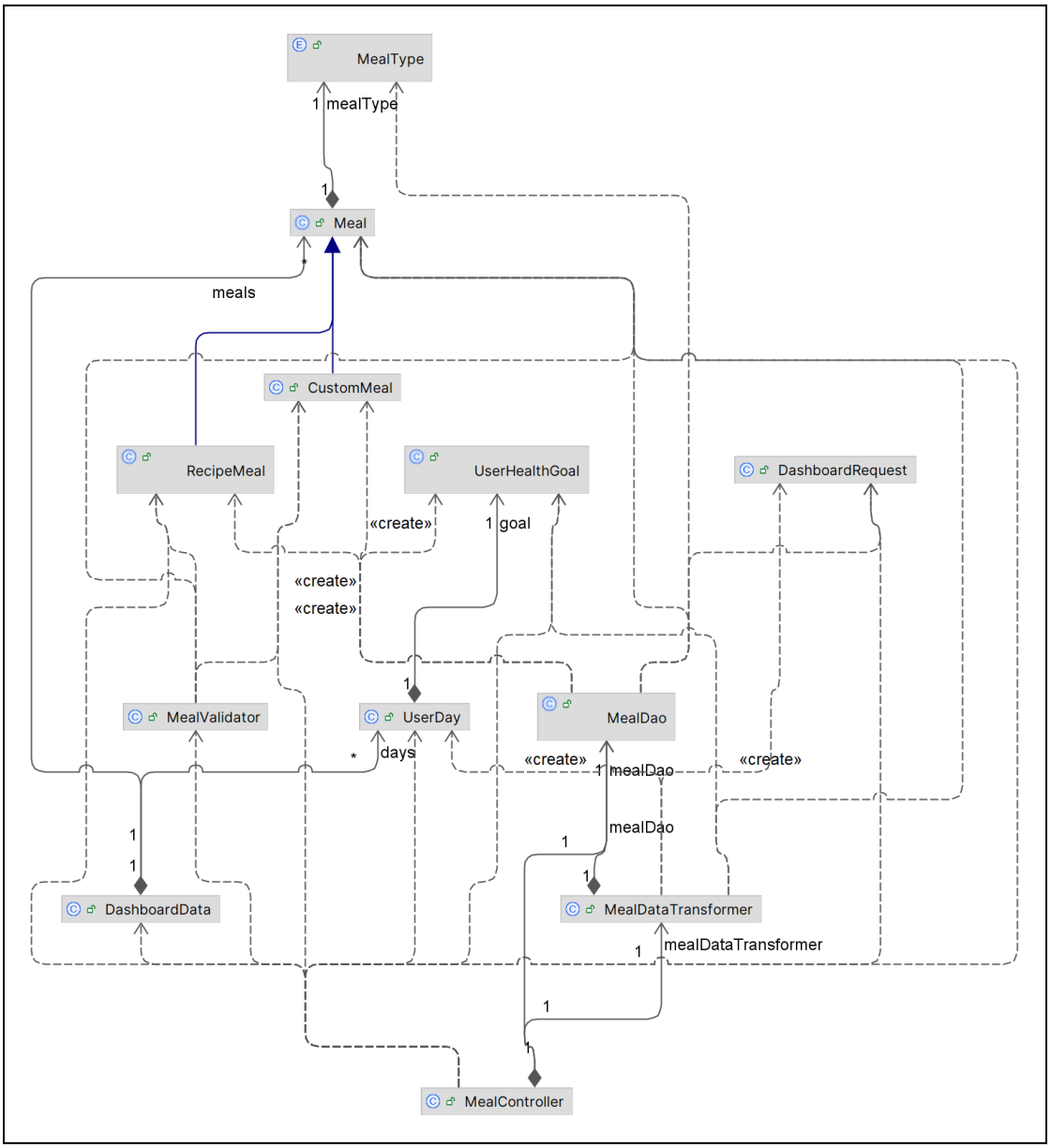
The SQL queries are assembled from hard-coded query fragments in our code, and the search terms are populated using syntax provided by JDBC. This keeps us safe against SQL injection attacks, since we aren't concatenating any potentially unsafe strings provided by the user into the query.



Backend class UML diagram for getting and viewing recipes.

User Meal Tracking

Our user meal tracking feature records users' meals and health goals and allows them to look at their history to see if they are keeping up with their goals. We store meals by day, and the user can get the sum of their calories and macronutrients for each day on their dashboard. When a user changes their calorie or macronutrient goals, we store the date they changed the goal and retain their old goals. This way, users can go back to see their past meal data accurately compared to their goal at the time it was recorded.



Backend class UML diagram for getting and viewing meals.

Authentication

We use JSON Web Tokens (JWTs) for authenticating users. We chose this method of authentication because it is largely stateless on the server side, so after the token is issued keeping track of it is the responsibility of the frontend. It also provides support for expiration time to keep user accounts secure.

When a user makes an API call to the MealWize backend, we first check if they are calling an endpoint that requires them to be logged in. If they are, we look for an

Authorization header in their HTTP request. This is where the JWT is sent by the frontend. These tokens have three sections:

- Header containing cryptographic information about the token.
- Payload, containing information about who issued the token, the user identified by the token, and the expiration timestamp.
- Signature, which verifies that the token was not tampered with.

When verifying whether a user is logged in, we validate the data in the token. The signature confirms that the token hasn't been tampered, and the payload tells us which user is logging in and whether the token has expired. If the user passes the signature and expiration time checks, we perform a third check to verify the token hasn't been logged out. We have a table in our database that stores invalid user tokens that users have logged out from. This last step fails authentication if the token is present in the database, which means the user previously logged out while using this token. This protects users from attacks that steal their token to gain access to their account. Even if the user's token is stolen, once they log out, the token becomes invalid and can no longer be used to authenticate. Since we don't store a list of valid tokens on the server side, this is the only way to invalidate tokens to provide a secure "log out" feature on the server side before a token expires. This is also why our implementation is "mostly" stateless. Without this logout feature, our authentication would be completely stateless, but less secure. We have a job running on the database that automatically clears tokens from this table after they expire, since they will now be denied for being expired. If a user passes all of these checks, they will be authenticated and their request will be served. If they fail any of them, the server responds with a 401 Unauthorized message.

We used Spring Boot's builtin security features that come provided with the spring-boot-starter-security and spring-security-crypto dependencies. These provide a framework that we built off of in our WebSecurityConfig and other classes to automatically determine which API endpoints require authentication and to automatically authenticate users when they attempt to connect to those endpoints. This prevents us from having to hardcode an authentication check for each endpoint we want to require authentication for. Additionally, this approach makes it easy to authenticate users wherever they provide a valid token, even if the token isn't required. This simplifies implementing optional features for logged in users, such as recipe favorites, on pages that don't require authentication to view.

Users log into MealWize using their username and password which are created with their account. The usernames are stored in our database along with a secure hash of their password, to prevent the password from being leaked in the event of a data breach. We used the Argon2id password hashing algorithm with secure parameters as suggested by OWASP[1]. When the user logs in we take the hash of the provided

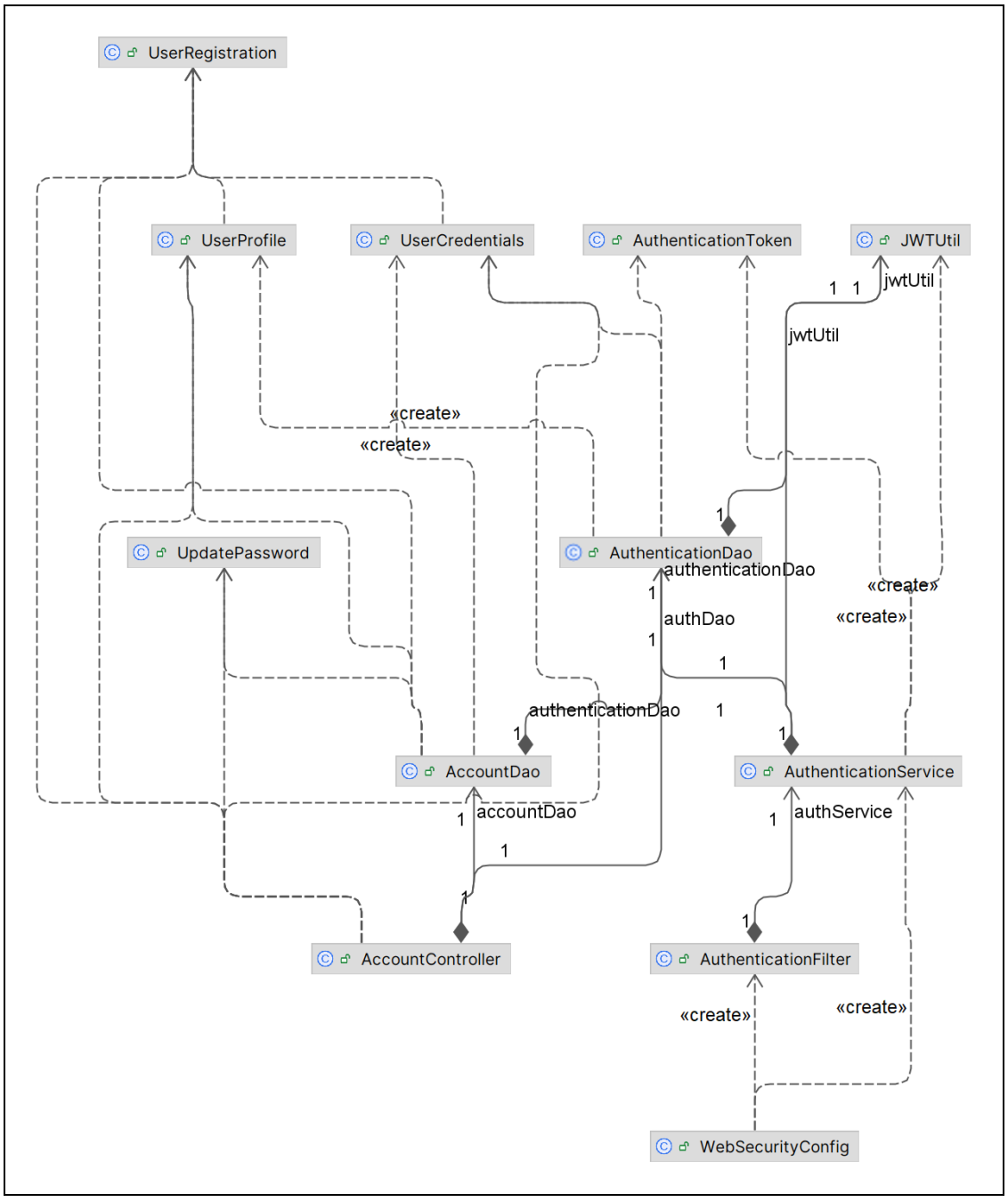
[1] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#argon2id
We use the following parameters as suggested: m=12288 (12 MiB), t=3, p=1.

password and compare it against the original password hash. If the user's credentials match, we provide them with a JWT to use for future requests. This is faster and more secure than validating their username and password with each request. The JWTs issued by the server are valid for 24 hours before the user must log in again. This duration provides a balance of convenience and security.

One of our goals when creating MealWize was to address an annoyance we found with existing recipe websites: many of them lock functionality behind an account. We understand the necessity of requiring user accounts for features that involve user-specific data, however we sought to provide as much functionality as possible to users who don't wish to make an account. In our authentication config, we require our users to have an account to access the follow endpoints (Note: ** matches any nested path recursively):

- /api/account
- /api/account/**
- /api/logout
- /api/favorites
- /api/favorites/**
- /api/recipe/upload
- /api/recipe/update
- /api/recipe/delete
- /api/meal/**

We determined that this subset of features is the bare minimum which must be gated behind a user account. Each of these features require a user account as an integral part of the action being performed. The account page is an obvious example. We also require an account to favorite recipes, because favorites are tied to the user's account. However, every other feature, including searching for recipes, is left open for users without accounts.



Backend class UML diagram for user accounts.

Database

We selected MySQL for our database. We chose MySQL over other options such as PostgreSQL because it is simple to set up and provides all the functionality we need. While PostgreSQL has more features, these were unnecessary for our project. We briefly considered a NoSQL database such as MongoDB or AWS' DynamoDB since we could store images in the database, removing our dependency on S3, however we decided against it because our data is structured enough to make a relational database

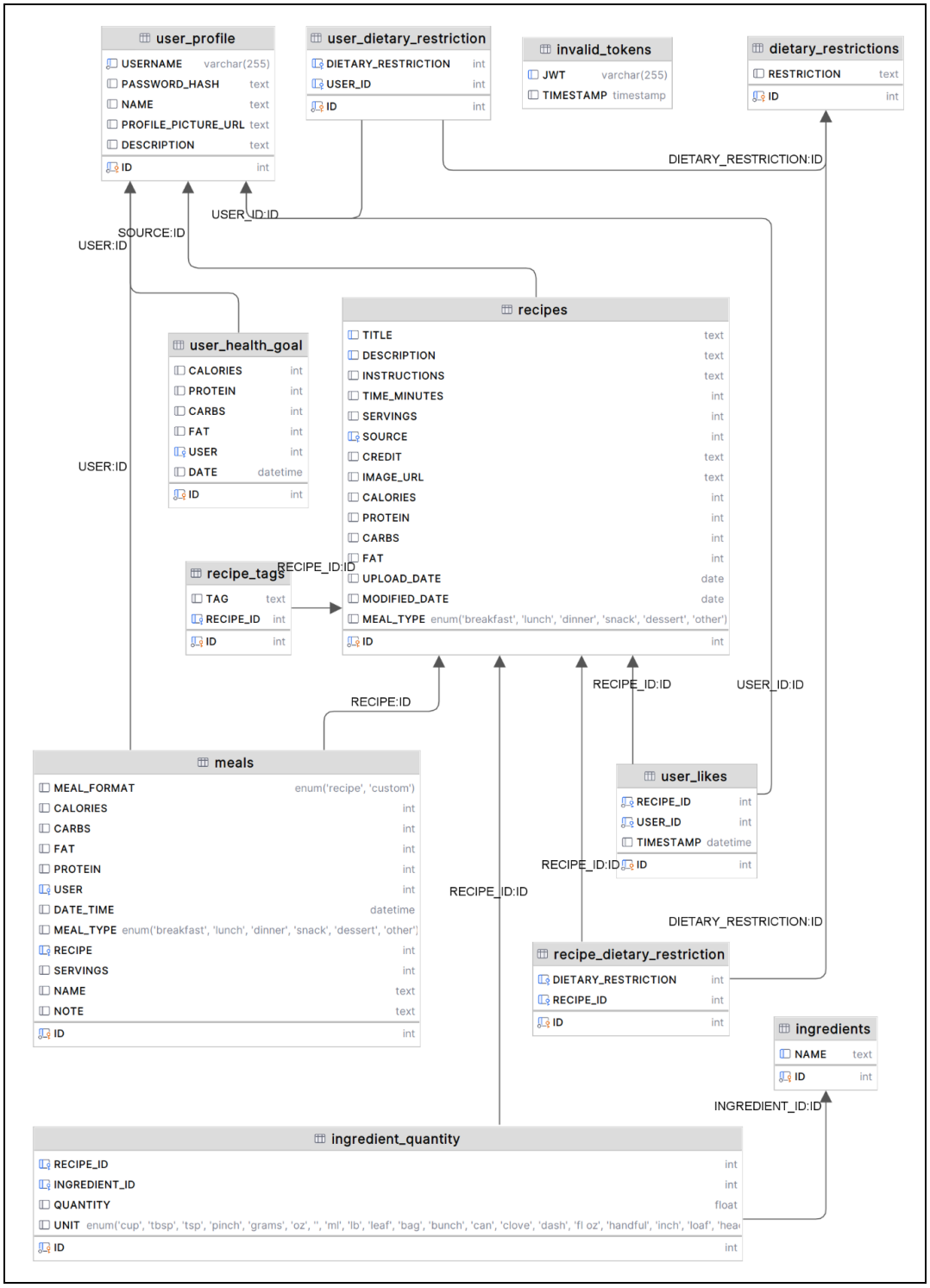
a good fit for the rest of our data. Additionally, as the total size of the uploaded images grows, S3 will be much more economical than a database.

Our database has the following 12 tables:

- dietary_restrictions
- ingredient_quantity
- ingredients
- invalid_tokens
- meals
- recipe_dietary_restriction
- recipe_tags
- recipes
- user_dietary_restriction
- user_health_goal
- user_likes
- user_profile

Many of the table names are self-explanatory, such as user_profile, recipes, and meals. Some of our data have one-to-many or many-to-many relationships, which is the purpose of the dietary_restrictions, ingredient_quantity, and ingredients tables, among others. These tables hold values and map them to recipes, such that each recipe can have multiple ingredients, dietary restrictions, and tags. The backend uses several SQL JOIN statements to collect the full data for each recipe across tables.

One of the first things our group discussed during our planning meetings, before the first sprint started, was how to populate our database with sample recipe data for testing and demonstrations. We investigated a few public recipe APIs, eventually deciding on the Spoonacular API. This API provided enough data for each recipe to populate each of our table's columns, letting us fully test each feature during development. We wrote a Python script to automatically upload the data from the API to our database. Later on, we also generated synthetic sample data containing user accounts and favorites, which was used to train our AWS Personalize model to provide helpful suggestions to users.



MealWize database schema.

Image File Storage

We use Amazon S3 to store image files for user profiles and recipes. Each user can upload a profile picture that will be displayed on their profile, and each recipe can also

have a corresponding image. We have one S3 bucket for our project, with one folder for recipe images and another for user profile images. We access the images by URL from the frontend, and the images are uploaded by the backend to S3 using the AWS Java SDK.

AWS Personalize Recommendations
The backend connects to the AWS Personalize campaign via a Python intermediate layer. The Python layer connects directly to the Personalize campaign using AWS' boto3 library and exposes the recommendations as an API endpoint using FastAPI. We also have Python scripts for automating the setup of our Personalize Campaign which are described in more detail below.

Testing
Most of the backend testing was done manually. During the first sprint we wrote unit tests for some of our initial work, but we didn't keep this up for the remainder of the project. Our manual testing consisted of making requests to the backend using IntelliJ's built-in HTTP client with the expected request from the frontend and verifying the correct response. These requests are stored in the test.http file within the repo, and were shared between team members. We also used IntelliJ's built-in database client to view the contents of the database during testing and to make manual changes to the database as necessary. Finally, during the last week of each sprint, after most of the development work was done for the frontend and backend individually, we did integration testing to make sure that the two sides of the application were communicating properly.

Recommendation System

Our recommendation system initially used a custom-built approach leveraging Collaborative Filtering and Content-Based Filtering to deliver recipe suggestions tailored to each user's historical interactions, dietary preferences, and nutritional goals. The system accounted for parameters such as the recipes a user liked, their dietary restrictions, and the calorie content of the recipes.

While this system provided good results, we later tested AWS Personalize with User-Personalization and User-Personalization-v2 recipes and decided to integrate Personalize's User-Personalization-v2 recipe after gaining access to AWS services. It provided us with significantly improved relevance and accuracy of meal suggestions, prompting us to adopt it as our primary recommendation engine moving forward.

The transformer-based model processes sequences of user interactions to learn contextual relationships and patterns. This enables the generation of personalized recommendations that align with individual user preferences. The model emphasizes recent user interactions to capture shifts in user behavior, ensuring that recommendations remain timely and pertinent. We

also tried tuning the hyperparameters- such as hidden layer sizes, learning rate, and exploration weights, but ultimately the difference was negligible.

Using Personalize provides our recommendations with the following features:

- **Dietary-Aware:** Users with specific restrictions (e.g., gluten-free, vegetarian) receive recipes that respect their dietary needs.
- **Nutrient-Focused:** Suggestions take into account calorie and macronutrient targets, helping users maintain or achieve their health objectives.
- **Community-Informed:** By learning from patterns in other users' interactions, the system can surface recipes that users with similar tastes and health goals have enjoyed.
- **Continuously Improving:** As users engage with recipes—by favoriting, skipping, or logging them—the recommendation engine refines its predictions. Over time, it becomes more accurate, adapting to evolving user preferences and leveraging data from like-minded individuals.

  Automation with Pipelines
  A key part of this system is the automated pipeline, written in Python, that streamlines the entire lifecycle—from data extraction to recommendation deployment. With a single command, the pipeline:

  1. **Extracts Data:** Retrieves recipe and user interaction data from the RDS MySQL database and prepares it in the required format.
  2. **Uploads to S3:** Moves the processed data into an S3 bucket for secure and scalable storage.
  3. **Configures Amazon Personalize:**
     - Creates and updates datasets and dataset groups.
     - Initiates dataset import jobs.
     - Generates solution versions and deploys campaigns without manual intervention.
  4. **Deploys Recommendations:** Once campaigns are active, the system is ready to serve personalized recommendations through the REST API.

  In addition to these automated steps, we have scheduled the pipeline to run at regular intervals using a cron job. This ensures that the entire process—from data extraction to recommendation deployment—occurs automatically and on a timely schedule, without the need for any manual triggers, keeping the recommendations up-to-date.

  By automating these steps, we eliminate repetitive manual tasks, reduce the risk of human error, and ensure that the recommendation engine stays current with the latest user data. This integrated process allows for rapid iteration and continuous improvement, delivering fresh, relevant suggestions that align with users' evolving tastes and health goals.

Retrospective

Sprint 1:

The first sprint was focused on setting up the foundation for our project by dividing tasks into three subteams: frontend, backend, and AI. This structure allowed us to allocate user stories efficiently based on expertise. Most of the planned user stories were completed successfully, with the exception of the AI model story, which was allocated as one large story and had to be deferred to Sprint 2.

Key learnings included:

- **Task Planning:** Dividing the sprint into two weeks of development followed by one week of integration and bug fixing helped keep us on track, even if not strictly adhered to.
- **Dynamic Adjustments:** As the project progressed, we identified additional features, such as advanced filtering for recipe searches, which required coordination between frontend and backend teams. These stories were successfully added and completed within Sprint 1.
- **Communication:** Challenges in group availability were mitigated by forming subteams and communicating asynchronously via Discord.

Sprint 2:

In Sprint 2, we expanded upon the foundation built in Sprint 1 by implementing key functionalities including the cart, favorites, and account management. However, we faced several challenges:

- **AWS Setup:** The delay in gaining access to AWS resources caused some planned stories to carry over from Sprint 1. Prioritizing AWS setup in this sprint enabled us to make significant progress, despite earlier setbacks.
- **Technical Hurdles:** Issues such as data formatting for the Spoonacular API for sample data and confusion around backend endpoint logic were addressed through clearer documentation.
- **Story Management:** Discrepancies in JIRA task tracking, such as stories being marked incorrectly or missing assignees, created confusion but highlighted the importance of accurate story management.

What worked well:

- **Team Coordination:** Regular communication ensured seamless integration between frontend and backend teams. Prompt issue resolution during testing helped avoid bottlenecks.
- **Proactive Adaptation:** New stories, like addressing display issues and integrating additional endpoints, were tackled successfully.

Sprint 3:

The final sprint focused on diet tracking features, completing stories carried over from earlier sprints, and addressing newly identified needs. With AWS access secured, we successfully deployed our application and completed overdue stories, including the Personalize recommendation model. However, we encountered unexpected costs with AWS services, prompting cautious use of the model during the final demos.

Key achievements:

- **Diet Tracking Features:** Implementing user calorie and macronutrient tracking was a significant milestone, leveraging the foundational work from earlier sprints.
- **AWS Integration:** Hosting the website on AWS and creating functionalities like image upload/download via S3 marked critical deliverables for the sprint.
- **Proactive Planning:** Creating dedicated stories for AWS hosting and calorie/macronutrient goal calculations ensured timely completion of objectives.

Learnings for the future:

- **Budget Awareness:** AWS cost overruns highlighted the importance of understanding service billing models to avoid unexpected expenses.
- **Story Management Improvements:** Ensuring accurate status updates and proper assignment in JIRA will help avoid confusion and improve clarity in team workflows.

Future Recommendations

For the future, some important “notes-to-self” would be:

- Proactive Planning:
    - Allocate extra time for complex dependencies like AWS setup or external integrations, as delays in access can derail timelines.
    - Plan and test budget-related constraints (e.g., AWS costs) in advance to avoid unexpected overruns.
- Communication:
    - Use subteams and asynchronous communication effectively when full-team availability is limited.
    - Maintain regular check-ins between frontend and backend teams, especially during testing and integration phases, to promptly address issues.
- Realistic Expectations:
    - Accept that some tasks may need to be pushed to future sprints if resource availability or technical issues arise, but document the reasons and learning points.
- Documentation and Maintenance:

- Formalize API documentation early in the process to streamline integration and onboarding for all stakeholders.

As we reflect on the progress made during this project, we recognize several opportunities to enhance and expand the application in future iterations. These enhancements focus on enriching the user experience, improving functionality, and ensuring the platform meets real-world demands effectively.

One of the key priorities for future development would be introducing community features, such as enabling users to comment on recipes and create blog posts. This will foster a sense of engagement and interaction among users, encouraging them to share insights, tips, and personal experiences. Additionally, implementing a recipe rating system will help users quickly identify popular and highly recommended recipes, making the platform more intuitive and user-friendly.

To broaden the application's accessibility, developing a mobile application is another crucial step. A dedicated mobile app will allow users to access features seamlessly on the go, catering to a broader audience. Moreover, integrating OAuth2 authentication will streamline the sign-in process by allowing users to log in using popular platforms like Google and Facebook, improving convenience and security.

On the technical front, an integral objective would be to improve the application's performance to ensure scalability and a smoother user experience. Integrating with additional sources of recipes will diversify the recipe database, offering users a richer and more comprehensive selection. We would also seek to address the moderation of user-uploaded content, either through AI-driven moderation or manual oversight, to maintain content quality and appropriateness.