[redacted]

[redacted]

Example 1:

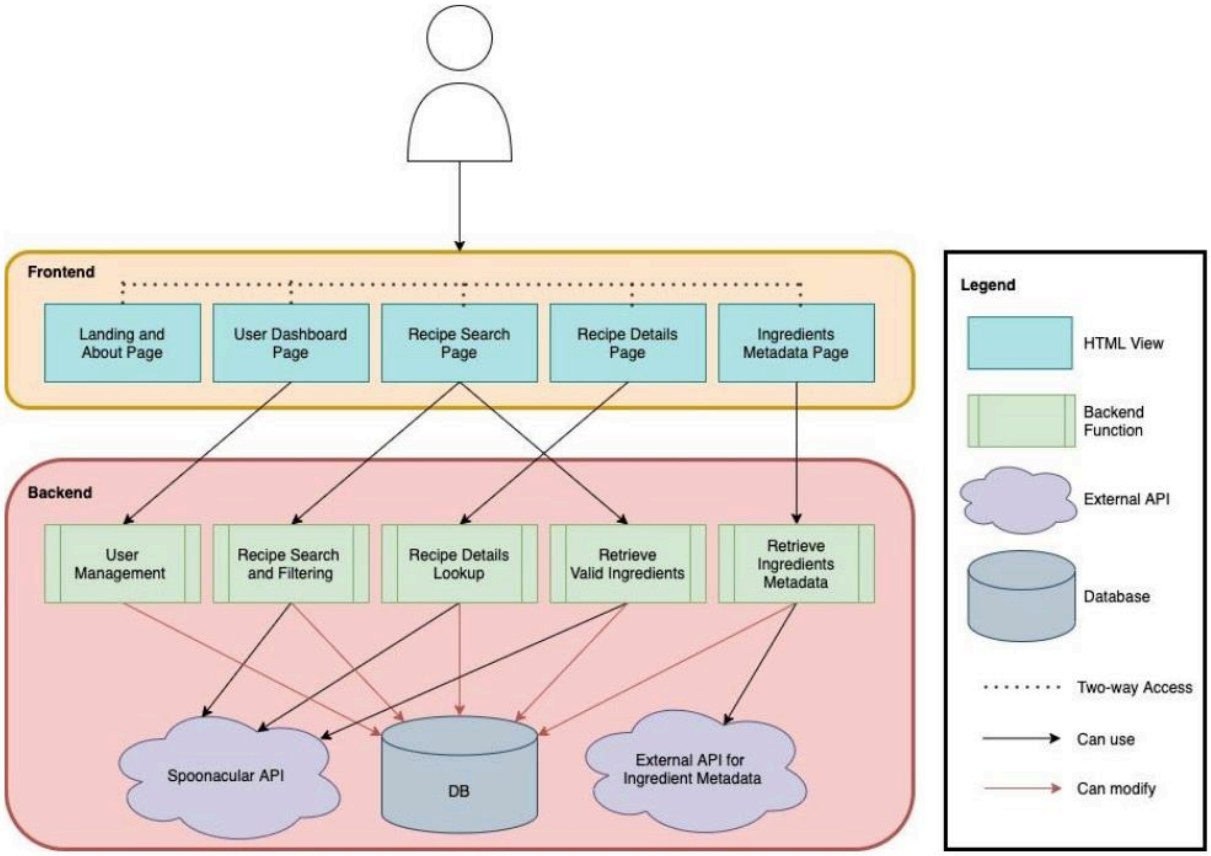

Figure 2: Client-Server view of Cheffu's architecture

The client-server view allows the application to be examined in detail. With this, the team is able to examine the flow of queries and data throughout the application in detail, such as enabling us to identify potential dataflow bottlenecks. Using this, the usability, accuracy and responsiveness of the design can be evaluated.

Alternate solution description

- For my alternate design, I picked a **Serverless Microservices Architecture** which heavily leverages amazon web services.
- The Frontend Interface would be in ReactJS and it would make calls to AWS's API Gateway service for different REST API calls to the serverless microservices
- The Microservices would be implemented using AWS's on demand Lambda functions which would serve the following purpose:
    - Get recipe related data from the Spoonacular API
    - Perform CRUD operation on information about recipes in AWS DynamoDB database
    - Apply filters to recipes
    - Other calculations and tasks

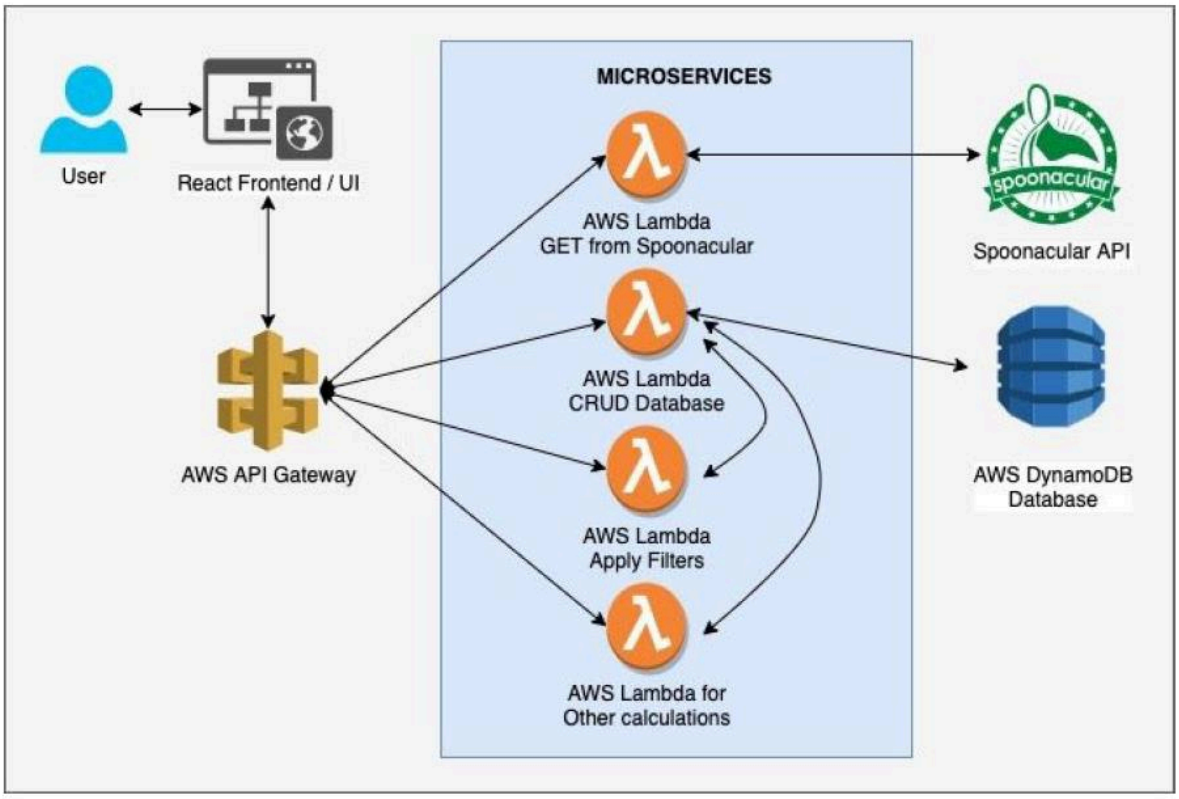

Figure 1: Alternate solution architecture

[redacted]

Example 2:

Microservices:

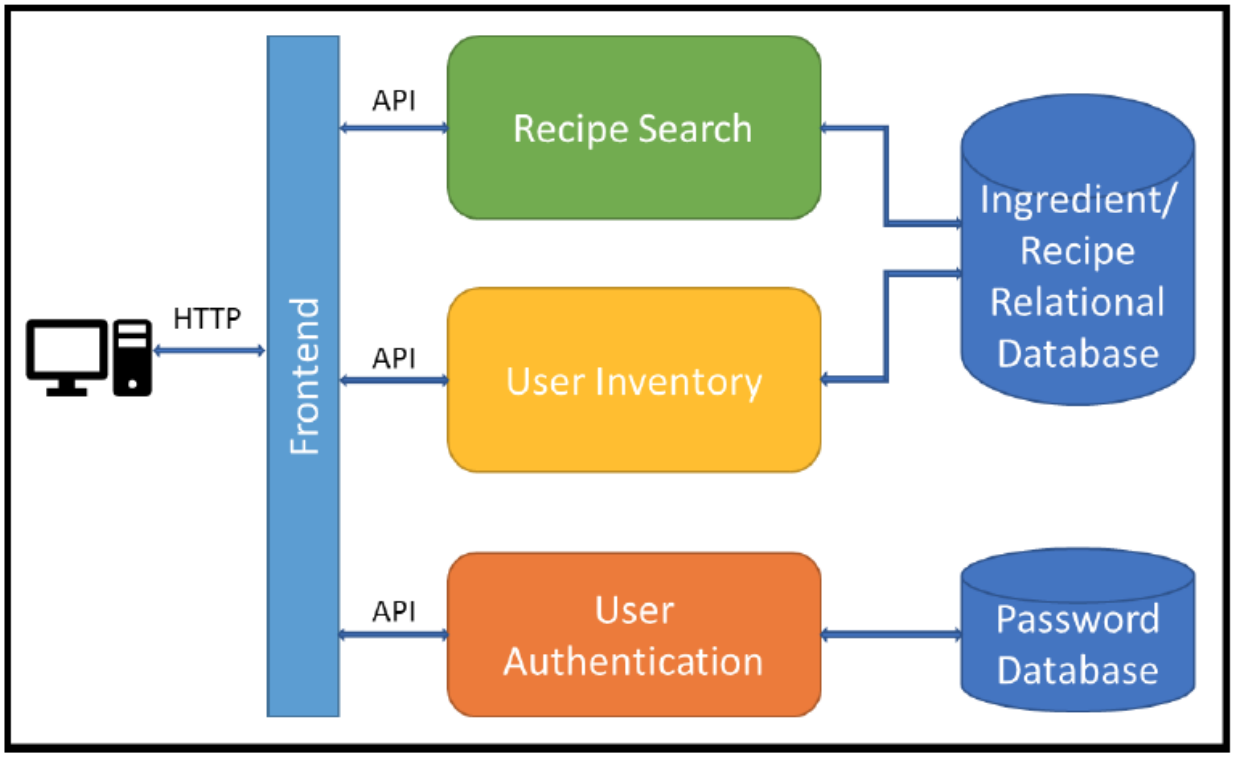

Figure 1: Decomposition of Monolithic Design into Microservice Architecture

Structure Description

The Frontend module serves as the point of interaction with clients, serving up the views and routing. It uses Rest APIs to communicate with the three other modules and HTTP to communicate with the client. The User Authentication module is used only for user's login credentials and has no other connections for security purposes. The User Inventory module is responsible for tracking which ingredients the user currently has and their quantities. This tracking role includes the removal of ingredients when they are used in a recipe. Each time a recipe is selected for cooking the Frontend will send a message to the User Inventory module with the quantities of ingredients used.

The Recipe Search module returns recipes based on the ingredients the user selected from their inventory and the filters they applied. These selections are made in the Frontend and sent to the Recipe Search module.

There are two distinct databases in this design (Figure 3): a relational database for inventories, ingredients, and recipes, and a non-relational database for user authentication. The User Authentication database is separate and access is limited to the User Authentication module in order to improve security.

Modifiability

Figure 1 exemplifies one on the modifiability properties of this new design. The microservice structure allows ongoing maintenance of microservices, bug fixes and security patches, as well as functionality

updates with very little impact on users. The replacement microservice can be brought online while the original is running, and traffic simply needs to be redirected. The highly modular nature of microservices allows complete replacement without needing to alter the other modules, as only the API needs to remain compatible.

Additionally, the compartmentalized form provided by microservices means that any new collaborator who would like to modify a feature only needs to understand the code in a given microservice rather than needing to understand the code as a whole.

Monolithic:

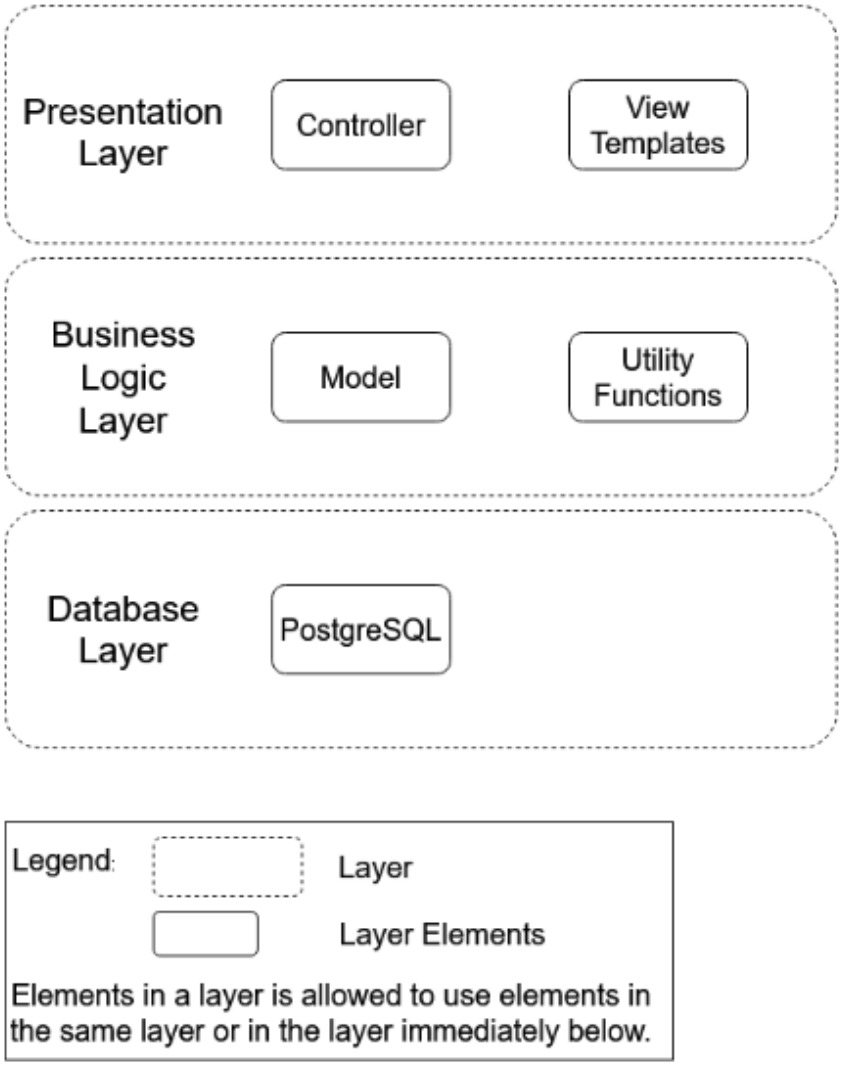

Figure 2: Static View Layer Structure

[redacted]

Example 3:

Microservices:

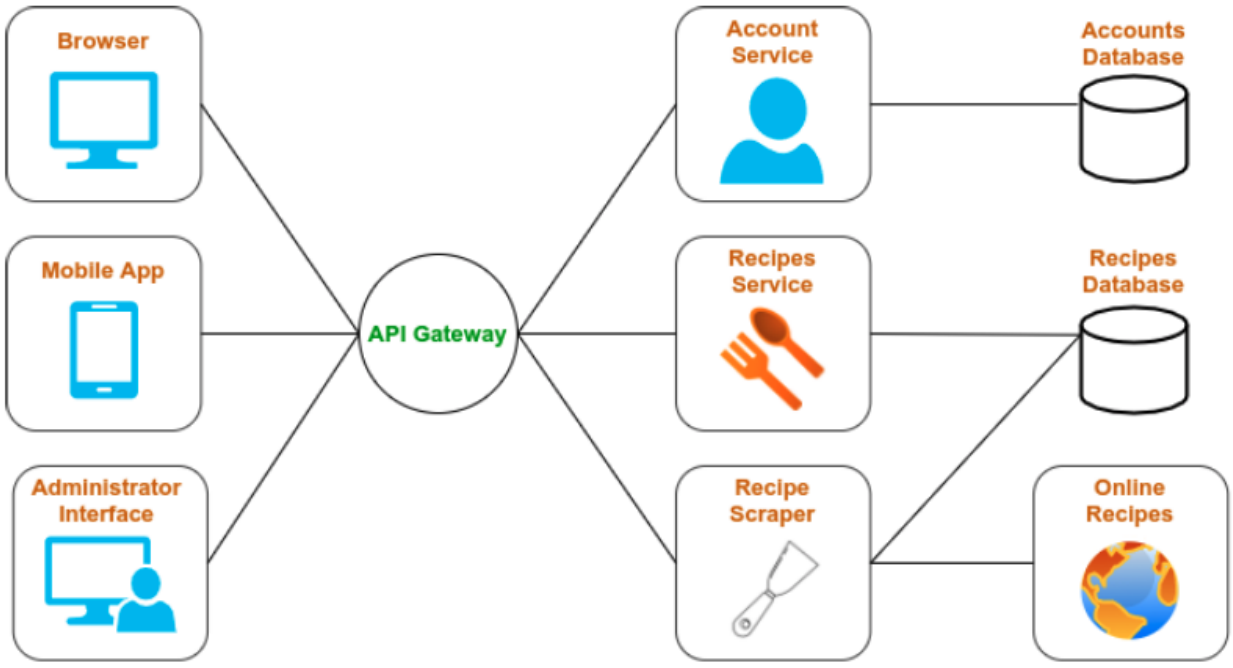

Figure 1: Decomposition of Devour into microservices

Figure 1 decomposes the key features of our recipe finder application, Devour, into a microservices architecture. The original application was developed as a monolith, using Django and deployed as a single Docker container. Under the microservices approach, each service can be deployed separately, or integrated from an external source (for example, a "Sign in with Google" service acting as the account service, with a translation layer). This diagram is a hybrid between a Dynamic and Allocation view, as it decomposes both components and deployment units. Our requirements do not include a mobile application, but it was included in the diagram as a demonstration of how new user views and services are easy to integrate with a microservices architecture.

There are 3 key services that compose the business logic of the application: the account service, the recipes service, and the recipe scraper. The account service is responsible for authenticating users. Unauthenticated users have the ability to search for recipes, but authenticated users gain the ability to save recipes to their account, and to add new recipes to the site via the recipe scraper. The account service would also be responsible for authenticating administrators, who have the ability to modify recipes and add ingredients via the administrator interface. The API gateway services as a unified interface to the services (which would be separated by RESTful resource endpoints), and can encompass a load balancer.

[redacted]

Monolithic:

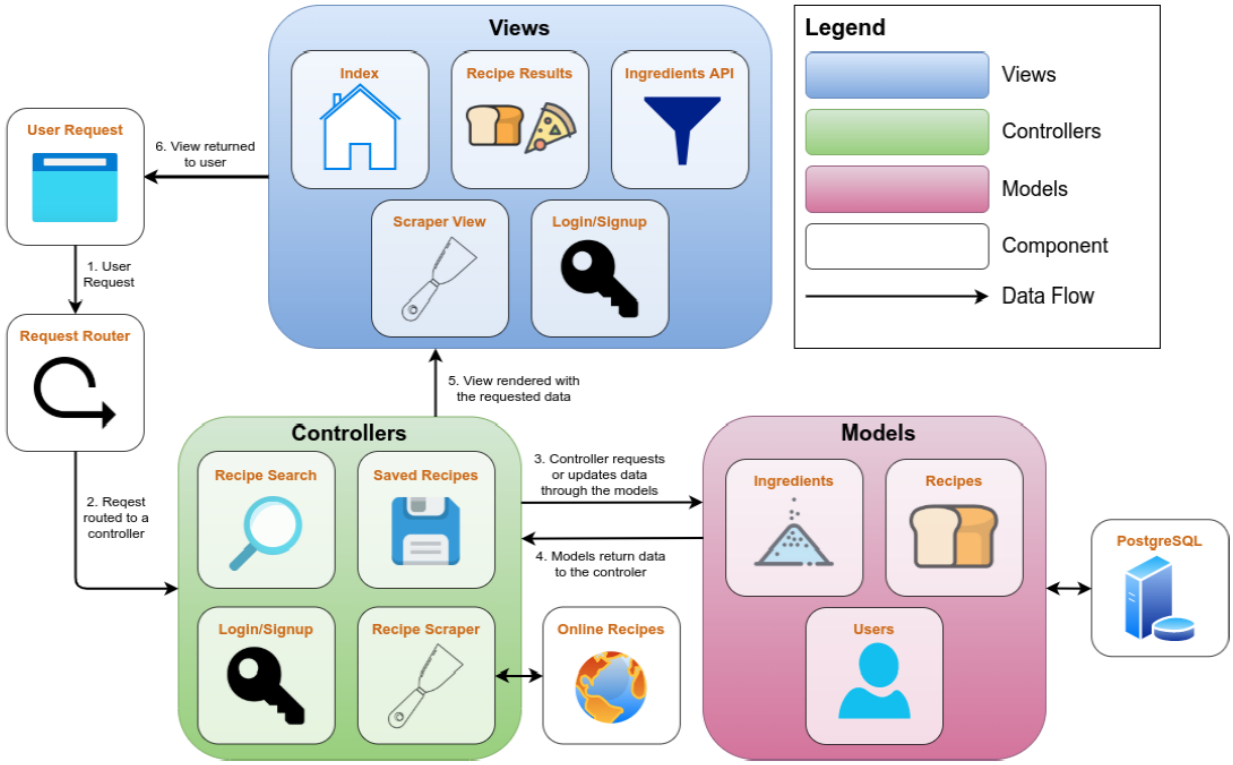

Figure 2: MVC View of Devour

Our application was implemented in Django which follows a similar design pattern to MVC (in Django parlance, “views” are called “templates” and “controllers” are called “views”). Figure 2 is the MVC view of our architecture. When a request is made by the user, it is sent to the appropriate controller by the request router. The controller processes the request, requesting or updating data from the database via the models. The controllers then render a response view, which is returned to the user. The Ingredients API view is implemented the same way but abstracted with Django Rest Framework. This view primarily lets us reason about the flow of data between functional units of the application, as they relate to the project requirements. Further, we get a good overview of how a monolithic application is typically developed, as each coloured section corresponds to separate code modules with disjoint functionality.

Reliability

Our focus on reliability is to ensure high availability while being fault-tolerant and recoverable. As Figures 1 and 2 show, our monolithic architecture is tightly integrated. Should a single part of the application at a lower layer in Figure 1 fail, it can affect the entire state of the application. We mitigate this risk to ensure availability by leveraging Django's built-in functionality as much as possible (such as for querying and user authentication), and hence our most critical functionality is battle-hardened by years of community development. It is unavoidable, however, that a single unlikely crash in a monolith can bring down the entire application. This is a tradeoff between, say, a microservices architecture which can have higher availability for parts of the application that didn't fail but is harder to maintain and develop for a smaller team.

Unlike microservices, however, monoliths can be relatively easy to re-deploy or scale, ensuring that our application is recoverable in the event of a failure. Application logs are maintained via the Docker Swarm cluster. Should a failure happen, the problem can be solved and the entire swarm stack indicated in Figure 1 quickly re-deployed. Our application is fault-tolerant as well, since PostgreSQL supports transactions – in the event of a failure during database writing within controllers or models, the transaction will be aborted and the integrity of the data maintained.