CSC 431

Quick Eats

System Architecture Specification (SAS)

Team 8

Kalpit Mody	Scrum Master
Jack St. Hilaire	Developer
Leah Harper	Developer

Version History

Version	Date	Author(s)	Change Comments
1.0	03/29/22	Kalpit Mody, Jack St. Hilaire Leah Harper	First Draft
1.1	04/29/22	Kalpit Mody Jack St. Hilaire Leah Harper	Updates to Diagrams

Table of Contents

Table of Figures

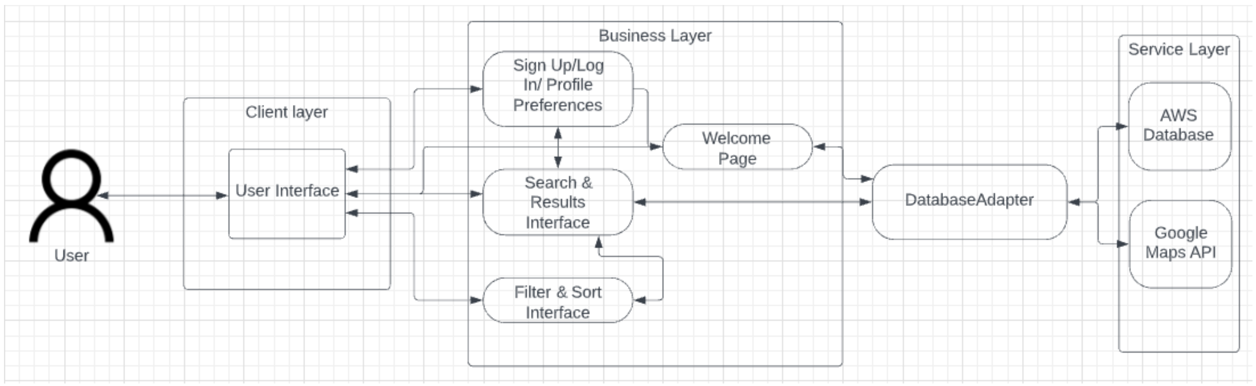
1. System Analysis

1.1. System Overview

The system for Quick Eats will leverage the three-tier architecture which includes a client, business, and service layer. The client server consists of the iOS and Android clients which will access the back-end service to display relevant features to the user interface. The business layer is where the magic happens. This is where the application will take user input such as location to query existing APIs and output the results to the client side. This is also where we will maintain the user preferences such as favorited restaurants. The service layer will host and maintain databases and API queries for the rest of our application through AWS.

The UI layer will host the user interface which will connect to the business layer through the DBConnectionInterface. The business layer will also include components to view the welcome page, sign up/log in, search, view results, filter and sort results, and view the estimated wait time. We will use API calls in the service layer to communicate with the database to store user information and preferences and output user search results for fast food locations in the provided area.

1.2. System Diagram



1.3. Actor Identification

The actors in our system include

- New Users: These actors are those users who have not created an account.
- Registered Users: These actors are those who have created an account.
- Server: This actor includes AWS which hosts our database and holds user data.

1.4. Design Rationale

1.4.1. Architectural Style

We will utilize a three-tier architecture in order to structure the project in an effective and efficient way. This architecture will allow us to use the React framework alongside AWS to create a seamless user experience. The layers will be:

- Client Layer: User interface for iOS and Android, allowing users to interact with our data easily without worrying about what is going on behind the scenes
- Business Logic Layer: Still using the React framework, this layer will be where we determine what information the user needs, calculate wait times based on user selections, and collect/analyze user submitted data
- Service Layer: This layer will deal with our databases and provide both security and authentication, utilizing AWS and relational databases

1.4.2. Design Patterns

1. Adapter

We will be using multiple different APIs, and creating many of our own components and services. Therefore, it will be very important for us to allow for interaction between all of these different moving parts. The adapter design pattern will be valuable because it will allow us to make incompatible components compatible with each other, allowing us to create a seamless three-tier system.

2. Facade

There will be a good amount of complexity and back-end processes that will be operating within our Business Logic layer, but the users do not need to be aware of a lot of this. They will find the application more usable and efficient if the UI is simple and straightforward, providing them with only what they actually need.

3. Strategy

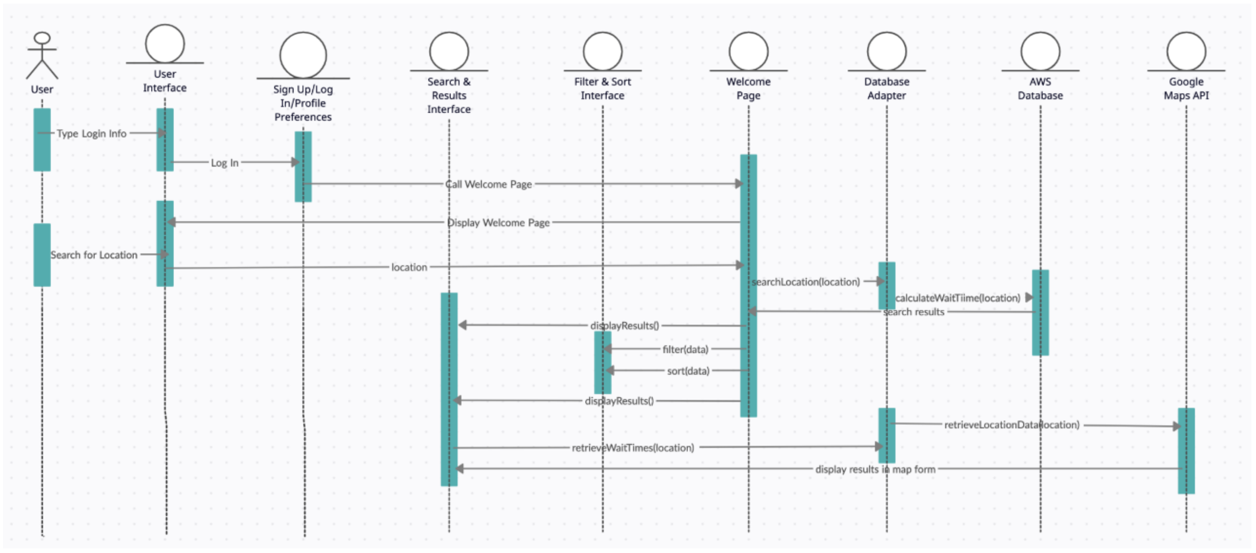
We want to do a lot of analysis, prediction, and data manipulation within the back-end of the application. In order to achieve this, it will be very important to be able to utilize various algorithms in an interchangeable way. The strategy design pattern will provide us with that ability.

1.4.3. Framework

The framework we will be using is React. Not only is this a well known and documented framework, but it is also one that provides us with a lot of flexibility. Most importantly, however, is that it will allow us to develop for multiple different platforms easily. The idea of the app running smoothly on both iOS and Android will be essential to the success of our project in the short and long term. In addition to cross-platform design capabilities and documentation, the React framework will also allow us to interact with APIs and services such as AWS, which is imperative to the functionality of our product.

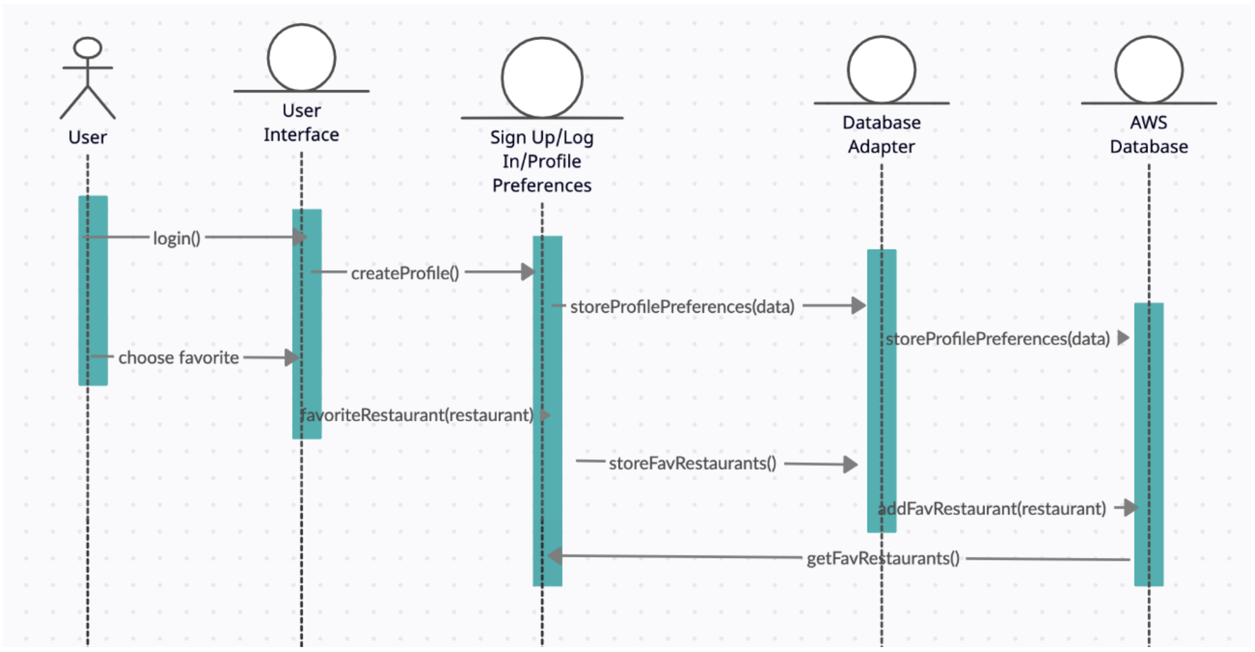
2. Functional Design

2.1. Finding Drive Through Wait Times



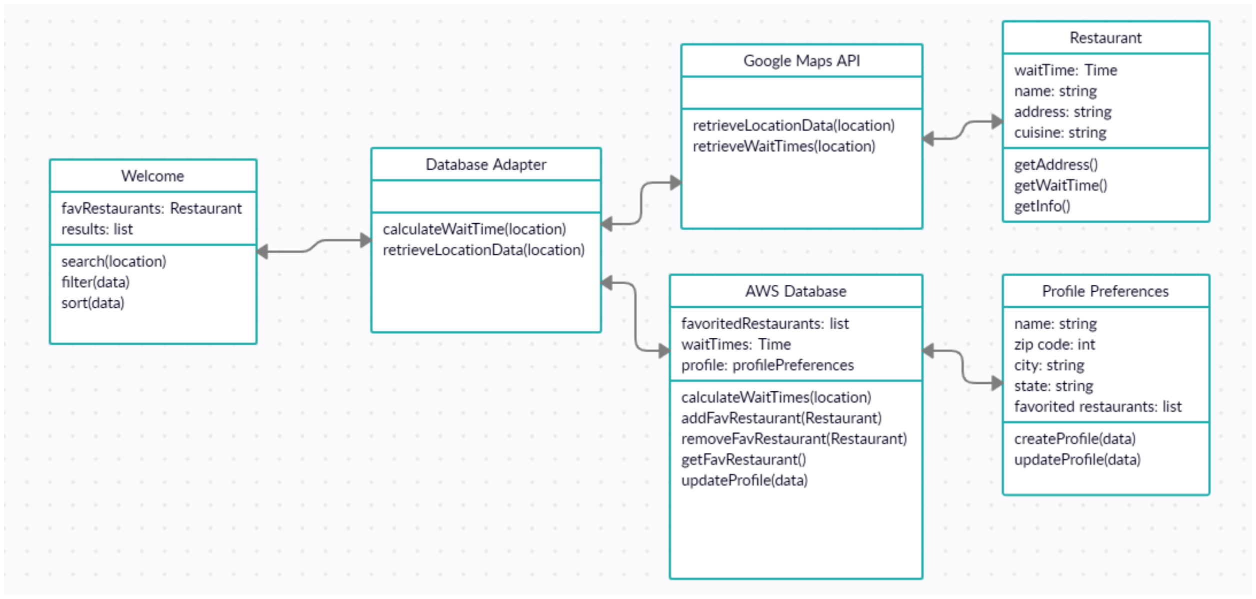
- When a user logs into the app they automatically go to the Welcome page.
- From the Welcome page they can see their favorited restaurants and search based on their location
- Users also can go to their profile page where they can edit their profile and favorited restaurants
- When a user searches based on their location they can choose to receive the results in list or map form
- They then can view the wait times for various restaurants in that location

2.2. Updating User Preferences


User
User Interface
Sign Up/Log In/Profile Preferences
Database Adapter
AWS Database
login()
createProfile()
storeProfilePreferences(data)
storeProfilePreferences(data)
choose favorite
favoriteRestaurant(restaurant)
storeFavRestaurants()
addFavRestaurant(restaurant)
getFavRestaurants()

- To update user preferences users must first log in and go to the welcome page
- Then, users can choose to go to their profile page
- From the profile page users can update their profile information (name, location, etc.)
- They can also see and modify their favorited restaurants. This data is stored in the AWS Database along with wait time information

3. Structural Design


Welcome
favRestaurants: Restaurant
results: list
search(location)
filter(data)
sort(data)
Database Adapter
calculateWaitTime(location)
retrieveLocationData(location)
Google Maps API
retrieveLocationData(location)
retrieveWaitTimes(location)
Restaurant
waitTime: Time
name: string
address: string
cuisine: string
getAddress()
getWaitTime()
getInfo()
AWS Database
favoritedRestaurants: list
waitTimes: Time
profile: profilePreferences
calculateWaitTimes(location)
addFavRestaurant(Restaurant)
removeFavRestaurant(Restaurant)
getFavRestaurant()
updateProfile(data)
Profile Preferences
name: string
zip code: int
city: string
state: string
favorited restaurants: list
createProfile(data)
updateProfile(data)