CSC 431

UHUNGRY?

System Architecture Specification (SAS)

Team 10

Matthew Maya	Scrum Master and Software Developer
David Mills	Software Developer
Nicholas Sosnivka	Software Developer

Version History

Version	Date	Author(s)	Change Comments
1.1	03/30/2022	Matthew Maya David Mills Nicholas Sosnivka	Original SAS due date

Table of Contents

Table of Figures

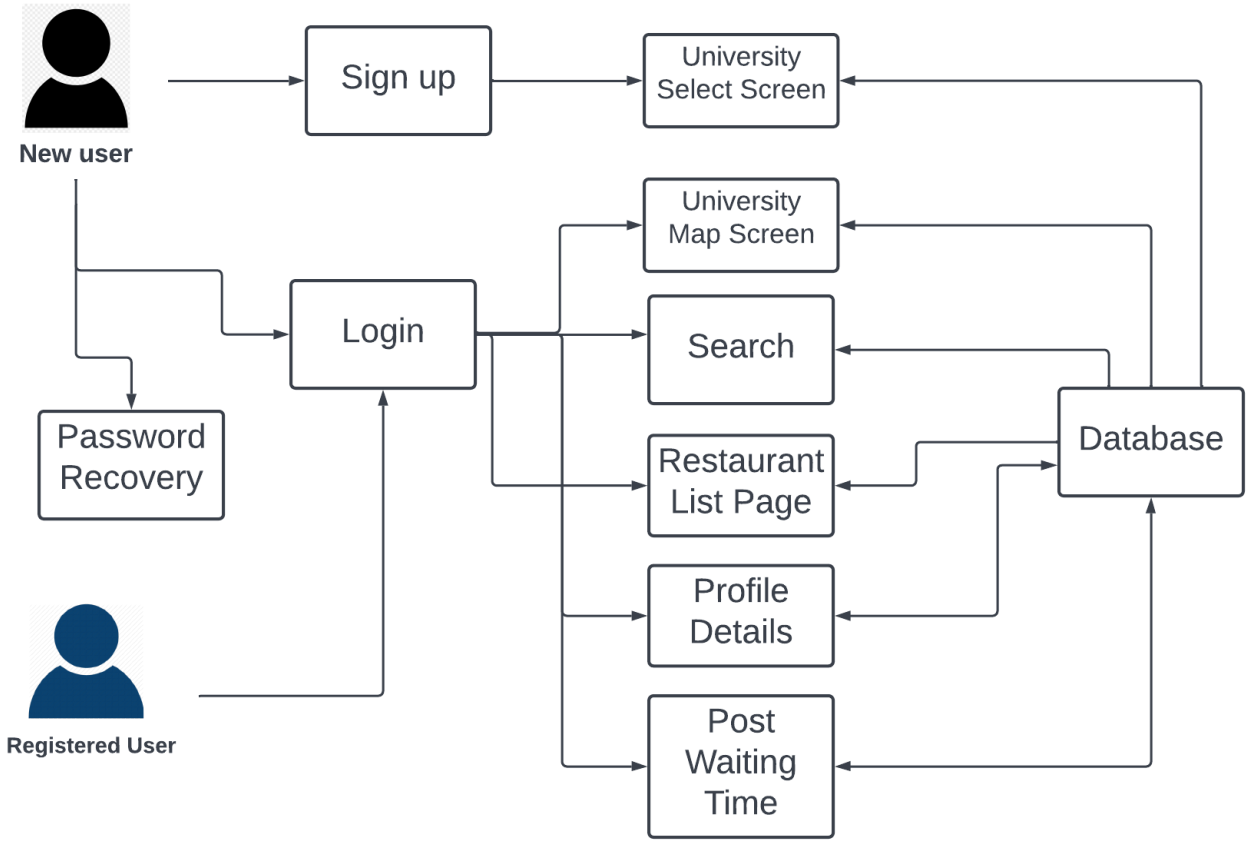
1. System Analysis

1.1 System Overview

This document describes the architecture design and specification of the UHUNGRY? food service application. The system uses a three-tier architecture which includes a client layer, a business layer, and a service layer. The client server is the IOS/Android client, which accesses the back-end service domain through GraphQL and provides the user interface. The business layer is where the application logic occurs, such as our algorithm for the waiting time, food service location card feature, and the filtering feature. The service layer uses the Amplify package to read and store data with GraphQL and handle authentication and security with AWS Cognito.

The system is made up of three principal parts: UI layer, Business logic layer, and service layer. The UI layer is connected to MySQL within the business logic layer in order for the user to be able to login. The business logic layer is composed of creating user profiles, selecting a university, updating waiting time based on algorithm, displaying dining cards, etc. Ultimately, the service layer includes the Amplify package to handle functions such as registration, login, authentication, as well as storing the user information in the database.

1.2 System Diagram



1.3 Actor Identification

New User: Users are individuals who have yet to create an account with UHUNGRY?
Registered User: Users are individuals who have previously registered their accounts with UHUNGRY? Registered Users have the ability to view the plethora of food services offered by the selected university. These users are the driving force of the application as they are the ones who will be self-reporting wait times in order to keep the application running as efficiently as possible.
Administrator: User is a higher-level individual behind the scenes of the application who is making sure that integrity rules aren't broken. For example, if the wait line at Panda Express is 15 minutes long and someone suggest a wait time of 2 hours then the administrator will flag that result and throw it away.

1.4 Design Rationale

1.4.1 Architectural Style

The app will utilize 3-tier architecture in the form of

- UI Layer: This layer uses Flutter to create UI widgets.
- Business Logic Layer: This layer includes the logic for calculating the approximate wait times of the selected food service at the selected University using our algorithm.
- Service Layer: This layer uses Amplify and GraphQL for services like authentication through third-party applications and database storage.

1.4.2 Design Pattern(s)

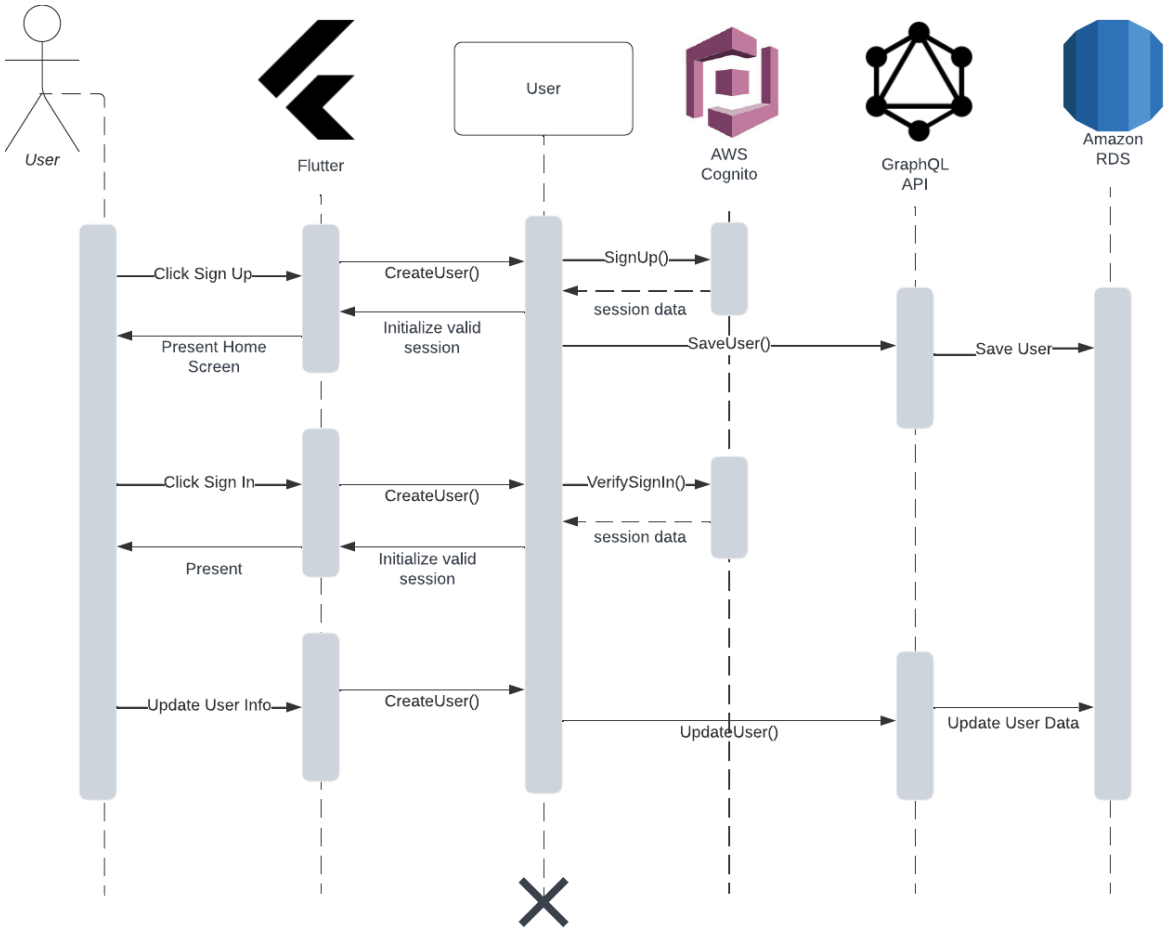
Our application's intended UI simplicity enables us to use the Façade design pattern as it would allow for the most ease when the user is navigating the application, while limiting the user's accessibility to all the components, and still ensuring functionality with more complex underlying code.

1.4.3 Framework

Our system will use the Flutter framework for the front-end and the AWS amplify package for the backend, specifically AWS Cognito, GraphQL API, and Amazon RDS. Using flutter in the front end will allow us to develop the application for IOS and Android simultaneously, shortening the development time significantly. The AWS amplify framework allows us to integrate data storage, authentication, and encryption seamlessly as they are all packaged on the same interface. AWS Amplify is also highly compatible with Flutter, and the steps for integrating the two frameworks is very well documented online. Finally, AWS amplify allows for integration of many more AWS API's that we may want to use for business purposes or increased functionality in the future.

2. Functional Design

2.1 User Actions Sequence Diagram



For user Registration

- When a user enters info for creating an account (email, username, password), and presses the sign-up button, the front-end Flutter API creates a User object with the registration information.
- That User object then runs a method that passes the registration information to AWS Cognito via the Auth Cognito package in Flutter.
- If the authentication in Cognito is successful, the user class fetches the session data from Cognito. This valid session is then initialized in the Flutter API
- If the valid session is initialized, the Futter API then renders the widget tree for the user's home page on the screen
- The user's information is also passed to the GraphQL API in amplify, and runs the GraphQL code which saves the users data into the MySQL database running via Amazon RDS.

- The User class's lifeline is destroyed after completion

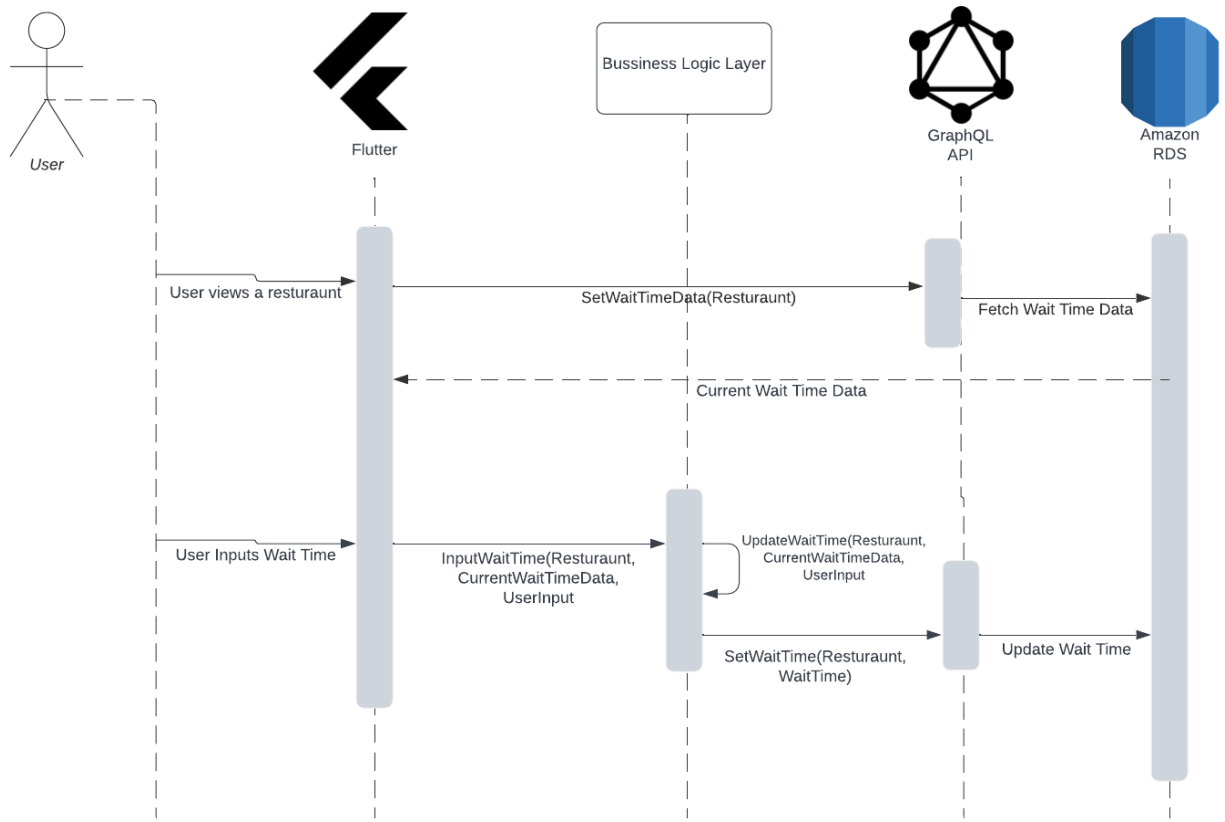
For user sign in

- When a user enters their account information (email, password) and presses the sign in button, the front-end Flutter API creates a User object with the sign in information
- That User object then runs a method which verifies the sign in information in AWS Cognito via the auth Cognito package in Flutter
- If AWS Cognito verifies that a user exists with that sign in information, the user class fetches the session data from Cognito. This valid session is then initialized in the Flutter API
- If the valid session is initialized, the Flutter API then renders the widget tree for the user's home page on the screen.
- The User class's lifeline is destroyed after completion

For updating user info

- When the user changes their information in the settings tab and confirms, the front-end Flutter API creates a User object with the updated information
- The user object then runs a method which executes a block of GraphQL code in the GraphQL API in amplify. This block of code updates the user's info in the MySQL database running via Amazon RDS.
- The User class's lifeline is destroyed after completion

2.2 Wait Times Sequence Diagram



For rendering Wait Times

- When the User decided to view a restaurant, The flutter API sends the restaurant information to the GraphQL API.
- The GraphQL API then runs a block of code which fetches the current wait time data for the restaurant in question and returns it to the flutter API.
- This data is stored locally on the device while the resturaunt is still selected.

For Inputting Wait Times

- When the User inputs the wait time for a restaurant, information on the restaurant in question, the current wait time, and the user's input are sent to the business layer. The current wait time data is still stored locally on the machine, so it doesn't need to be fetched again
- The business layer executes a script which runs an algorithm to update the wait time.
- The new wait time is then passed to the GraphQL API which updates the wait time for that restaurant in the Database.

3. Structural Design