CSC 431

Apollo: A Music Dating and Social Network Application

System Architecture Specification (SAS)

Team 09

Samantha Kamath	Project Manager
Megan Page	Project Manager
Yunting Zhao	Prototyper

Version History

Version	Date	Author(s)	Change Comments
1	4/1/2021	Samantha Kamath, Megan Page, Yunting Zhao	

Table of Contents

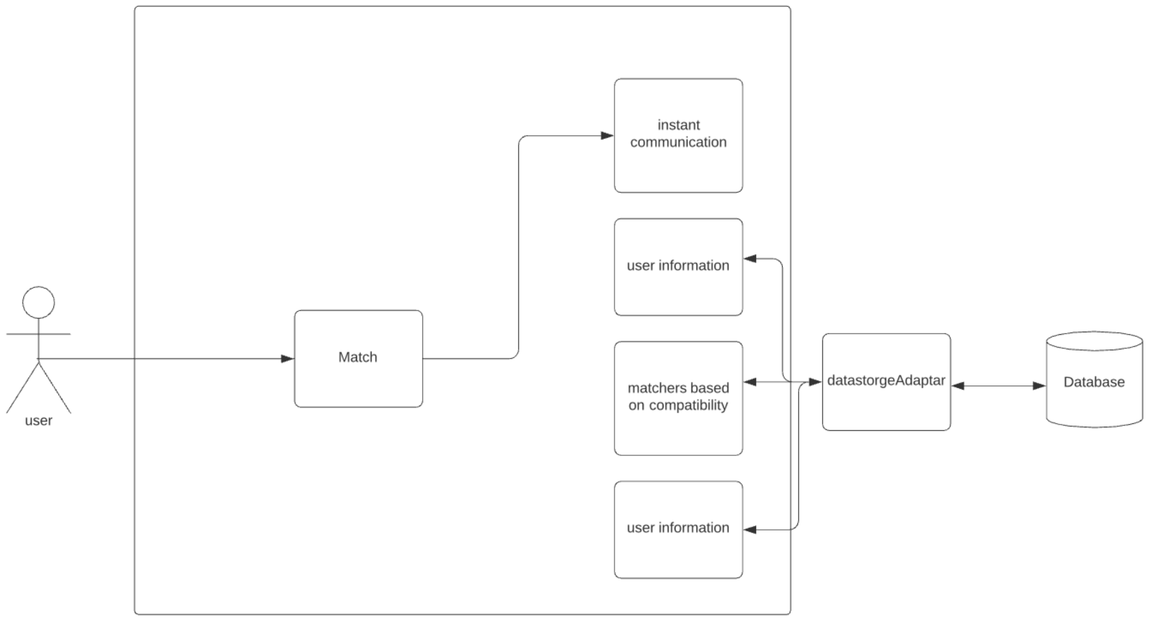
1. System Analysis

1.1 System Overview

This document describes the architecture design and specification of the Apollo Music Dating and Social Network Application. This system uses a three-tier architecture, including a client layer, a business layer, and a service layer. The client server is the IOS client, which accesses the back-end service domain through DNS and provides the user interface. The business layer includes the application logic, including the messaging feature, the upcoming concerts feature, and the compatibility matching. The service layer uses Firebase Database to read and store data; Firebase Database also provides backend services like authentication and security.

The system consists of three main parts. Firstly, the system uses the business logic server, which includes a Profile class object; this object stores the user's information, including each user's set of matches, potential matches, and user preferences. The Profile class is responsible for storing message history information for each of the user's matches. The system also includes the Adaptor class, which converts the information from the Profile class to the JSON data used to store information in Firebase Realtime Database; the Adaptor class helps carry out functions like compatibility matching and creating matches. Lastly, the system uses the service layer, which includes Firebase Realtime Database; this database provides functions such as registration, login, and authentication; the database also stores user information.

1.2 System Diagram



1.3 Actor Identification

Actors interacting with the system include:
- New Users: Users are individuals who have not yet created an account with Apollo.
- Registered Users: Users are individuals who have registered their accounts with Apollo.

- Platform Server: The platform, Firebase, includes a built-in cloud database and authentication services with third-party applications.

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 the preferences matcher and suggested upcoming concerts.
- Service Layer: This layer uses Firebase for services like authentication through third-party applications and database storage.

1.4.2 Design Pattern(s)

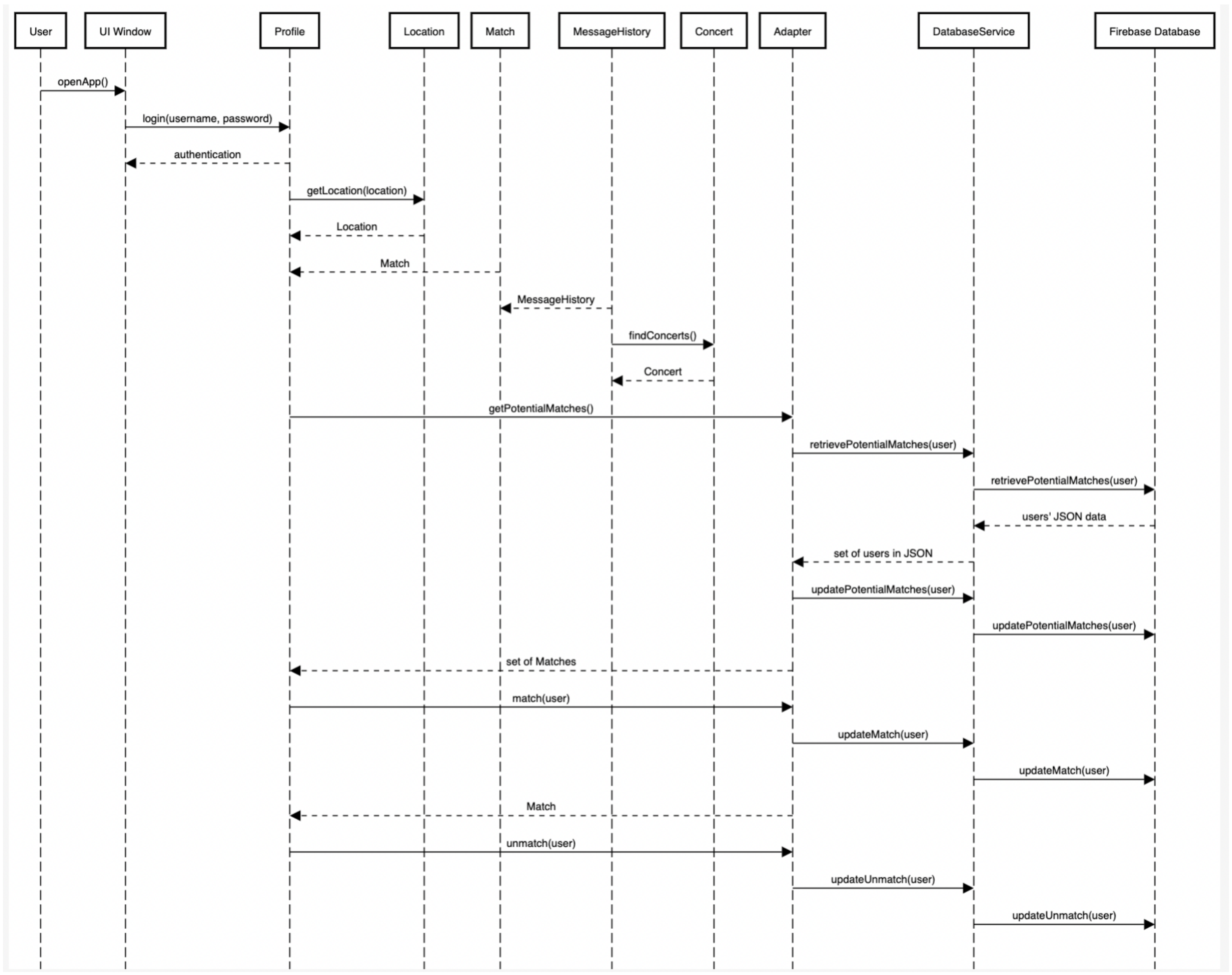
We predict that the Adaptor Design pattern will be primarily used in our project. Adaptor classes will be used to communicate between the business logic layer and service layer. Data from the database in the service layer will be converted to the Profile class in the business logic layer using an adaptor class; similarly, user profiles will be converted to JSON data to add to the database in the service layer using another adaptor class.

1.4.3 Framework

We will be using Flutter for the UI framework; Flutter is well-documented, and developers on our team have previous experience with the framework. However, our app also requires additional backend services, such as authentication, database storage, and business logic to implement compatibility algorithms and the upcoming concerts algorithm. Firebase is a Backend-as-a-Service that can implement authentication, cloud database storage, and security functions for our app with ease. For the other backend logic, such as the app's algorithms, the ExpressJS framework will be used. ExpressJS has access to all the NodeJS packages, and developers on our team have previous experience using JavaScript and ExpressJS.

2. Functional Design

2.1 Interacting with the App

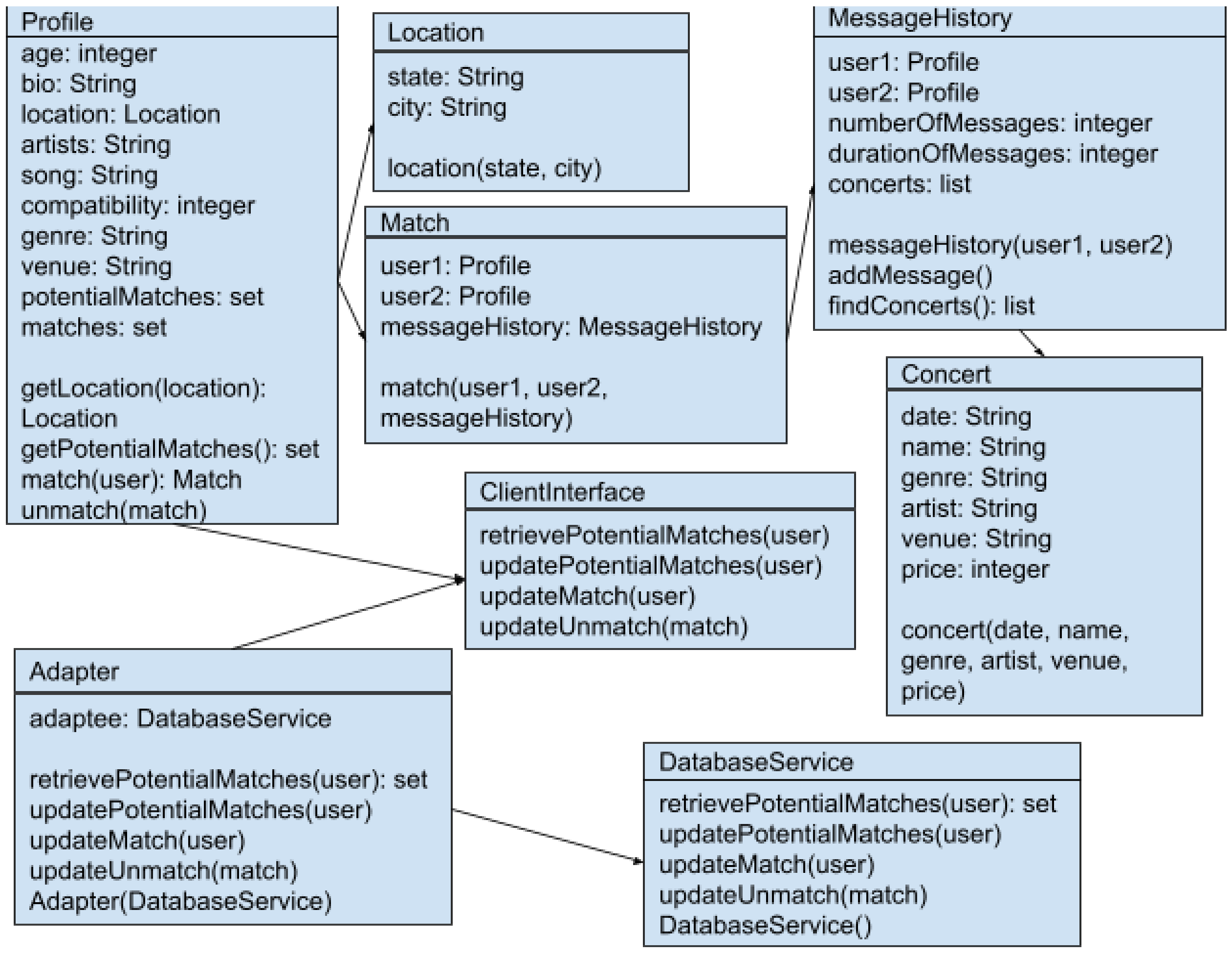


- When a user opens the app, they are taken to the UI window.
- At this point, they login with their username and password, and authentication is provided for them to access the rest of the app.
- The user has a Profile associated with them. Upon changing their location in their profile, the Profile class creates a Location data type for their location.
- When a user saves their profile, the Profile class calls getPotentialMatches(), which both retrieves the potential matches for the user and saves the potential matches as a set of Profiles to the user's Profile class. First, the Profile class calls getPotentialMatches(), which then uses the Adapter class to retrieve potential matches. This returns a set of users in JSON form. Then, the Adaptor class updates the user's data in the database to reflect

this new set of users as potential matches. This returns a set of Matches to the Profile class.

- When a user matches with another, the Profile class calls match(user), which creates a Match object and adds it to the set of matches in the user's Profile class. This also includes updating the user's data in the database via the Adapter class. Similarly, unmatch(user) removes a Match from the set of matches in the user's Profile class.

3. Structural Design



Profile
age: integer
bio: String
location: Location
artists: String
song: String
compatibility: integer
genre: String
venue: String
potentialMatches: set
matches: set
getLocation(location): Location
getPotentialMatches(): set
match(user): Match
unmatch(match)
Location
state: String
city: String
location(state, city)
MessageHistory
user1: Profile
user2: Profile
numberOfMessages: integer
durationOfMessages: integer
concerts: list
messageHistory(user1, user2)
addMessage()
findConcerts(): list
Match
user1: Profile
user2: Profile
messageHistory: MessageHistory
match(user1, user2, messageHistory)
Concert
date: String
name: String
genre: String
artist: String
venue: String
price: integer
concert(date, name, genre, artist, venue, price)
ClientInterface
retrievePotentialMatches(user)
updatePotentialMatches(user)
updateMatch(user)
updateUnmatch(match)
Adapter
adaptee: DatabaseService
retrievePotentialMatches(user): set
updatePotentialMatches(user)
updateMatch(user)
updateUnmatch(match)
Adapter(DatabaseService)
DatabaseService
retrievePotentialMatches(user): set
updatePotentialMatches(user)
updateMatch(user)
updateUnmatch(match)
DatabaseService()