Boston University
Electrical & Computer Engineering

amadeus

User Manual

EC464 Capstone Senior Design Project

Team 7:
Dev Bhatia - devb@bu.edu
Brandon DeSiata - desiata@bu.edu
Ryuichi Ohhata – ryu74@bu.edu
James Wasson - jwasson@bu.edu
Samantha Au - samau@bu.edu

Submitted to:
Osama Alshaykh, Ph.D.
Department of Electrical & Computer Engineering
Boston University
osama@bu.edu

Submitted: April 18, 2022

Amadeus User Manual

Table of Contents

Executive Summary

Amadeus is a mobile application designed for musicians to connect and self-promote. The application will allow users to create a profile to showcase themselves playing, tether those profiles to a musical group, and network with other musicians. These individuals and groups can communicate with others and promote events to their network. The ultimate result of this project will be a shippable mobile application with profiles, direct messaging, and a hyper-local map to which registered users can add entries to.

1 Introduction

To gain a comprehensive understanding of the problem to be solved and current market conditions, Amadeus began with a series of musician interviews. These interviews allowed for the concept to be refined and the problem narrowed to smaller key areas.

The insights gained from conversations with musicians were imperative in elucidating Amadeus' conceptual model. We gathered that though platforms exist for connecting people, their mass appeal often leave much to be desired in the form of finding a specific “brand” of musician to play with, often preventing many from starting bands or even just collaborating altogether. These musicians additionally had little in the form of promoting live shows, which further decreased their propensity towards creating a musical group. The illustrated needs are further exacerbated for certain groups of people - younger musicians - those without an established network of musicians are particularly susceptible to imposter syndrome, in which they feel as though the entire musical community is far above their level.

To solve these problems, we ultimately propose Amadeus. The created solution will unify the disjointed and unreliable aspects of starting a band. By aiding users through the many phases of starting a band and eventually gigging, we hope that our platform will become the de facto application for connecting with like-minded musicians.

2 System Overview and Installation

2.1 Overview block diagram

Amadeus relies on a decoupled architecture that enables components of the application to execute independently while still interfacing with each other through our API. The frontend was constructed using React Native and the backend using Python Flask.

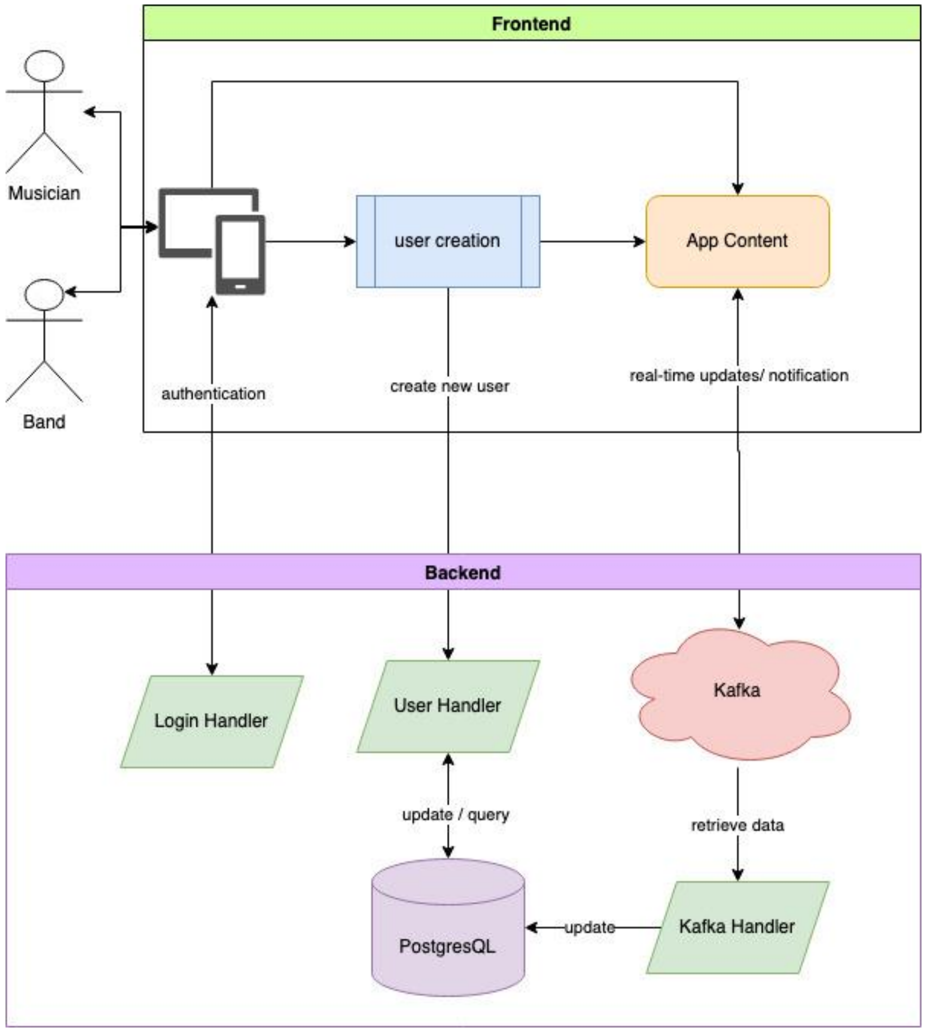
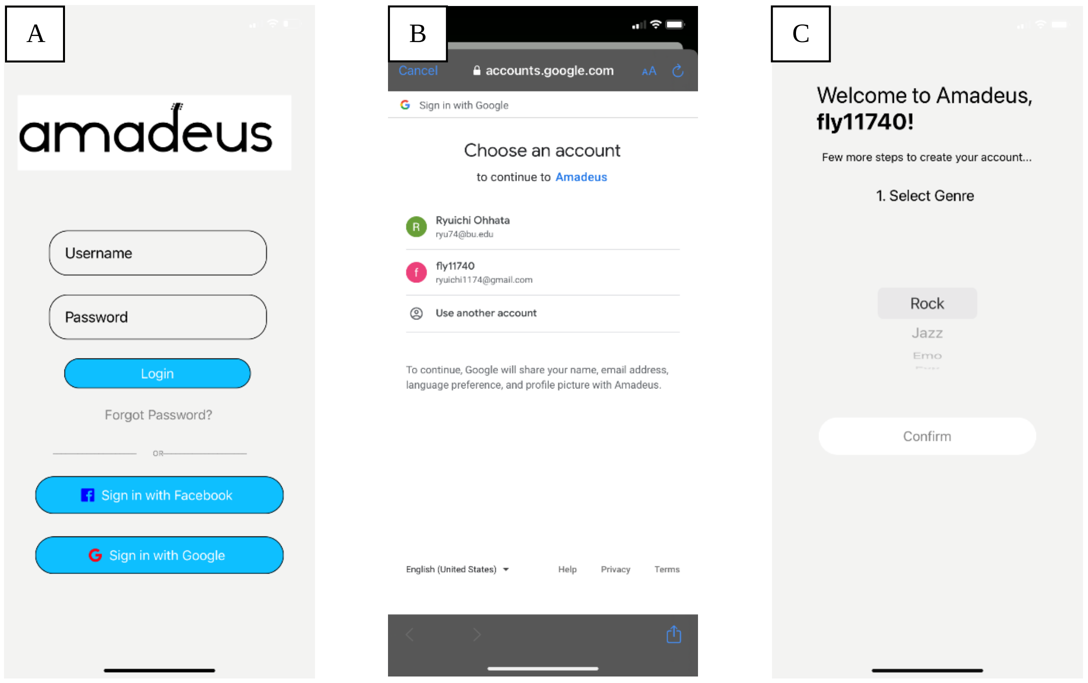


Figure 2.1 - An overview of Amadeus' underlying architecture.

2.2 User interface.

The user interface and general screen flow are the core of Amadeus. The overarching set of expected interactions with the application are shown in Figure 3.1. The screens used for initializing the user after a Google login are shown below.



To continue, Google will share your name, email address, language preference, and profile picture with Amadeus.

Figure 2.2 - Starting Screens of Amadeus. A) Upon launching the application, the user will be greeted with the home screen. B) The application will ask for Google verification if that option is selected. C) The first field that will be requested of the user is their genre of choice.

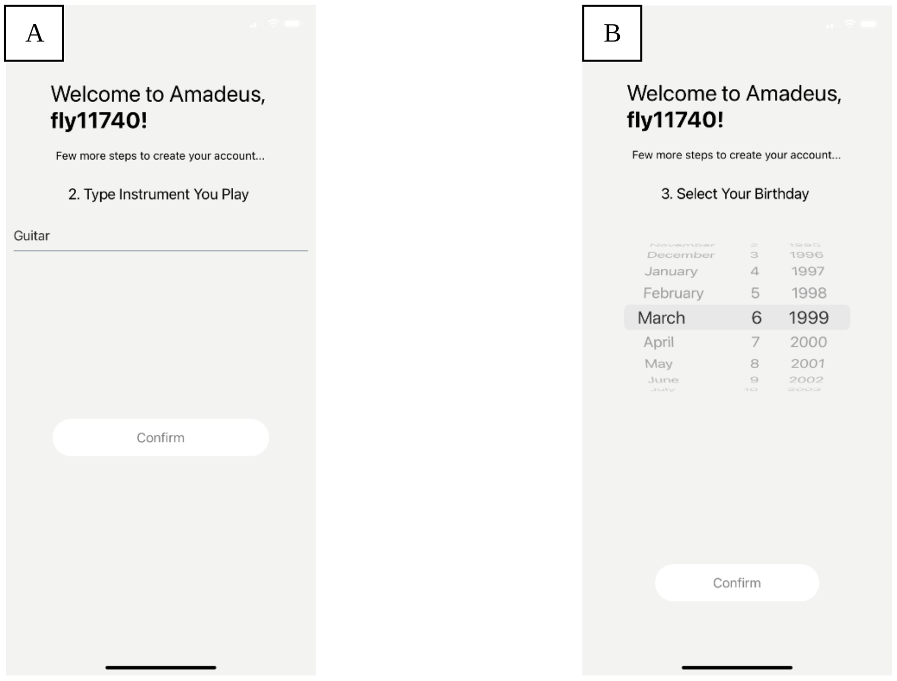
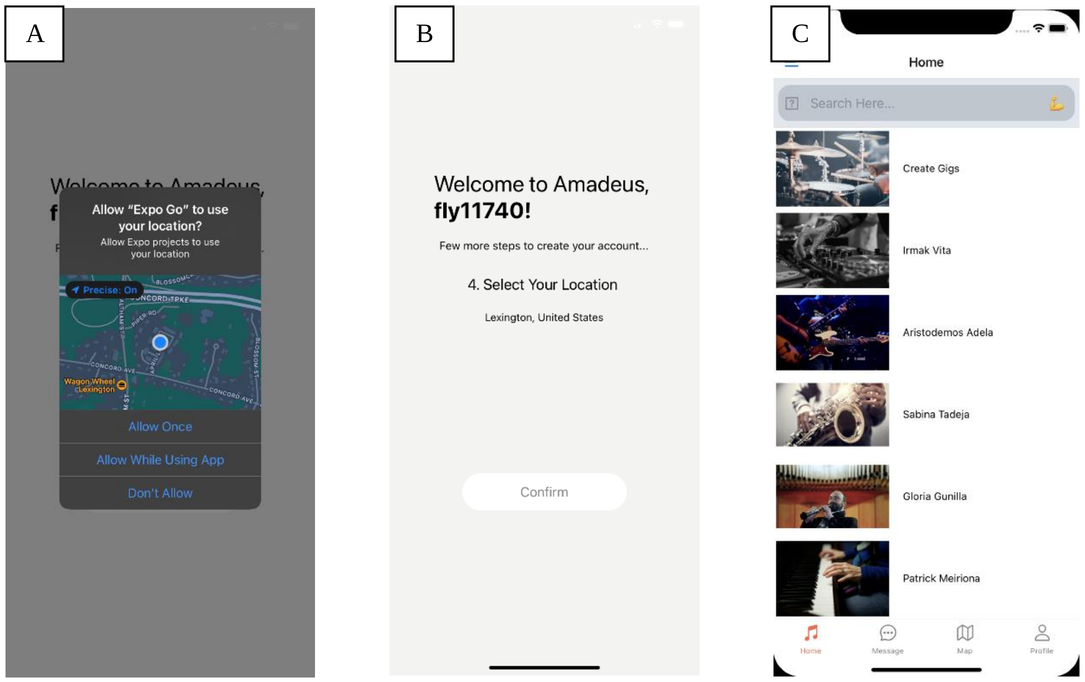


Figure 2.3 – Follow-up Information Requests. A) A text input field for the user's instrument of choice. B) A picker module for entering a birthdate



Allow "Expo Go" to use your location?
Allow Expo projects to use your location

Welcome to Amadeus, **fly11740!**

Few more steps to create your account...

4. Select Your Location

Lexington, United States

Figure 2.4 – Additional Information Request and Landing Screen. A) After clicking "confirm" on the birthday clicker, a popup asking for the location permission will appear. B) Upon receiving permission, the location will appear in the field shown. C) After confirming, the user will land at the default home screen.

2.3 Physical description.

Because Amadeus is entirely a software project, the physical description of the project will vary user-to-user depending entirely on the user's iPhone size. The application's user interface has been configured for a variety of screen sizes.

2.4 Installation, setup, and support

Setting up the application to run locally should be done in two steps. First, the backend and database system must be set up as described in our readme on Amadeus' Github page. The steps to do so are as follows:

- Create python env through
  `python3 -m venv env`
- Activate python env through
  `source env/bin/activate`
- Download dependencies through
  `pip install -r requirements.txt`
- Append .env file in the backend/api folder. It should have
  `DATABASE_URL=[your_postgres]`
- Run app.py

Once the backend is operational, the frontend can be run easily using `npm start`, or `expo start`. Assuming the backend has been initialized correctly, scanning the QR code that pops up on the screen should allow for the user to download the Expo App, and use our application.

3 Operation of the Project

3.1 Operating Mode 1: Normal Operation

The user creation screen will be the main entry point for a new user (i.e., a user that is not already in the database). A tether to the user handler and login handler in the API ensures that this process runs smoothly in the backend and reports the results back to the frontend. Once the user has been successfully verified and created, they are sent into the main contents of the application. Here, the user will be able to access the various features of the application, including a map, their own profile, a profile search page, and a browse message page. The expected control flow from the user's perspective is shown in Figure 3.1.

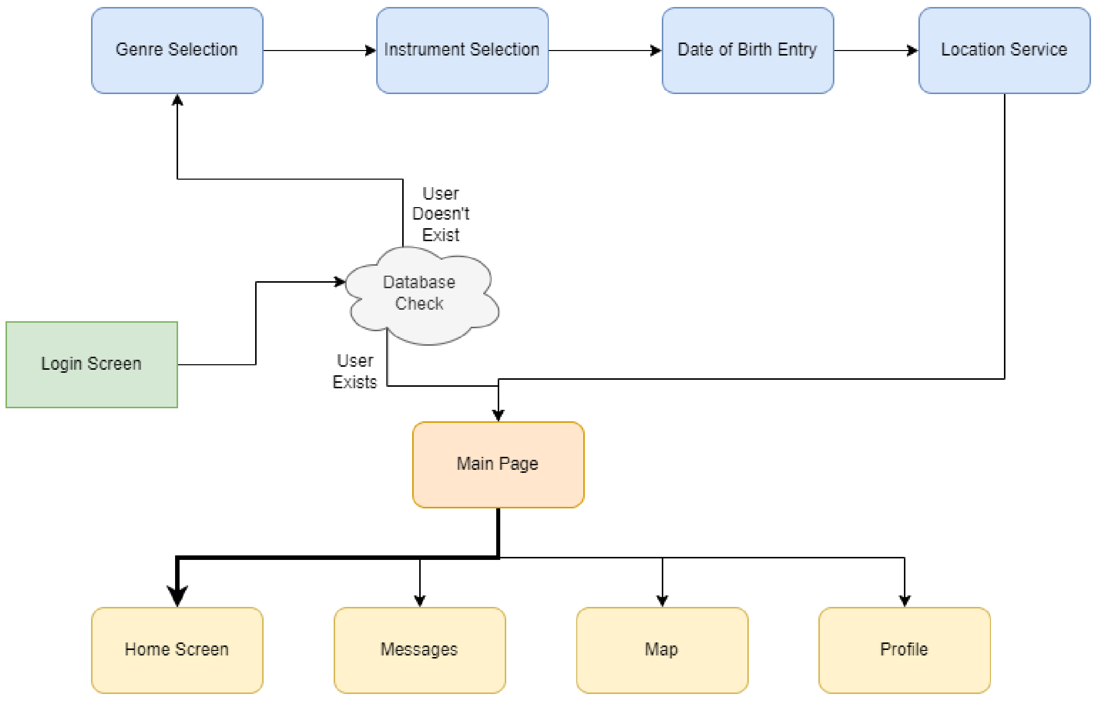


Figure 3.1 – User-Facing Screen Flow

After signing in and arriving at the home screen (refer to Section 2.2), the various features that the user can interact with are demonstrated in Figures 3.2 - 3.5.

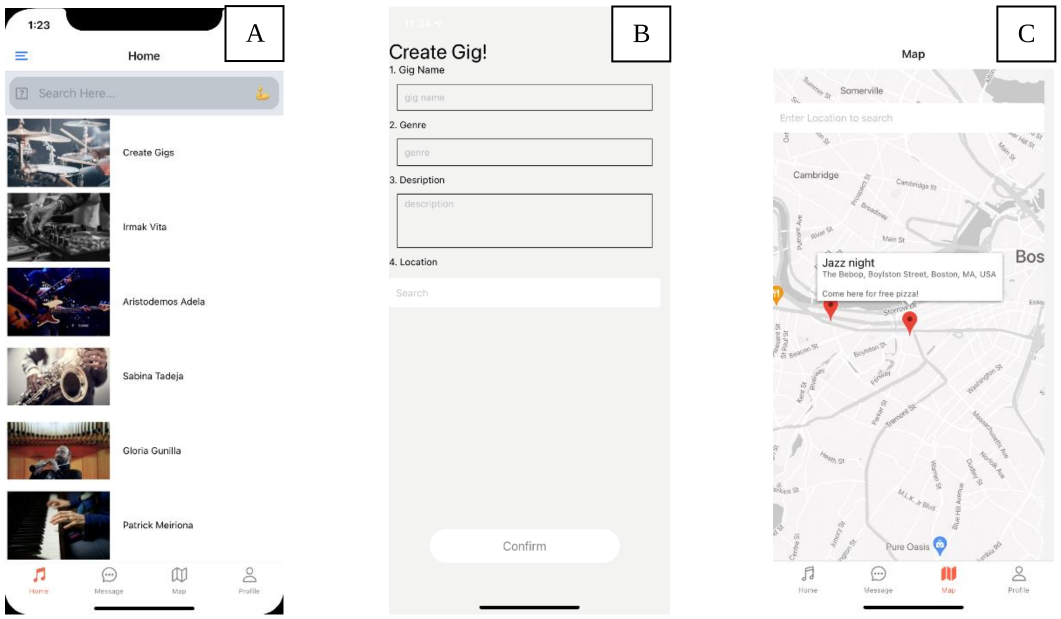


Figure 3.2 – The Gig Creation Flow. A) The home screen that the user arrives at after the user creation loop. B) The page to create a gig entry in the table. The "location" field relates to the Google Maps API and will recommend locations as the user types. C) The Maps screen with the created gigs presented as pins on the map.

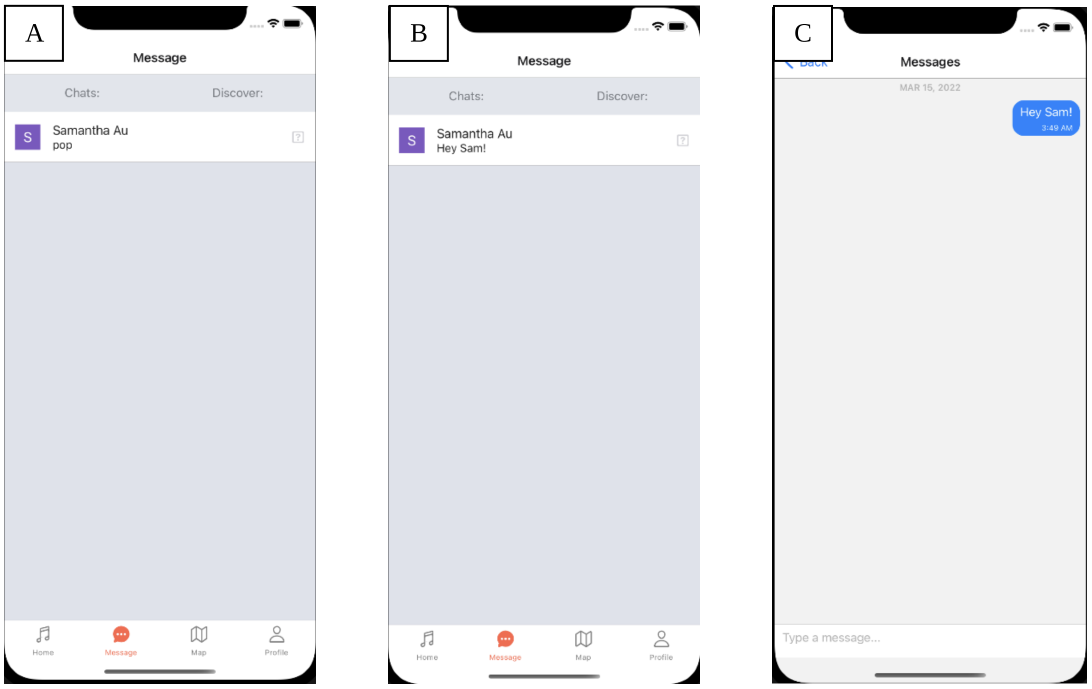
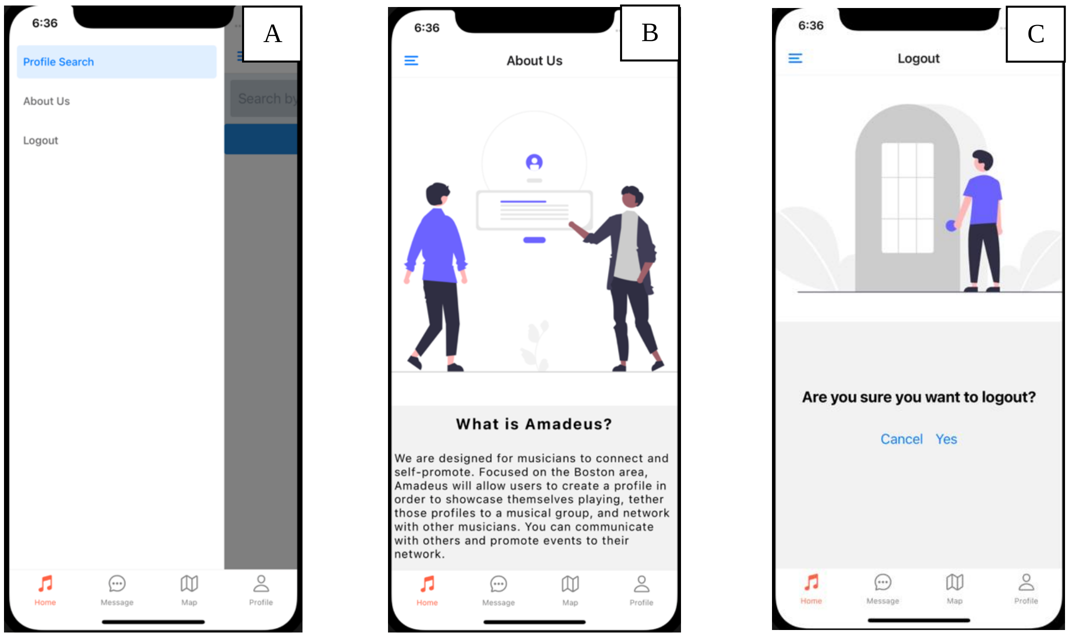


Figure 3.3 – The Messaging Flow. A) When the user arrives at the screen, they will have the option to open a chatroom with other registered users on the application, viewing their genre of choice before opening the room. B) Once the chatroom is open, the user can switch the "Chats" sub-tab to enter the newly created chat. C) The user can now send and receive messages with the other user

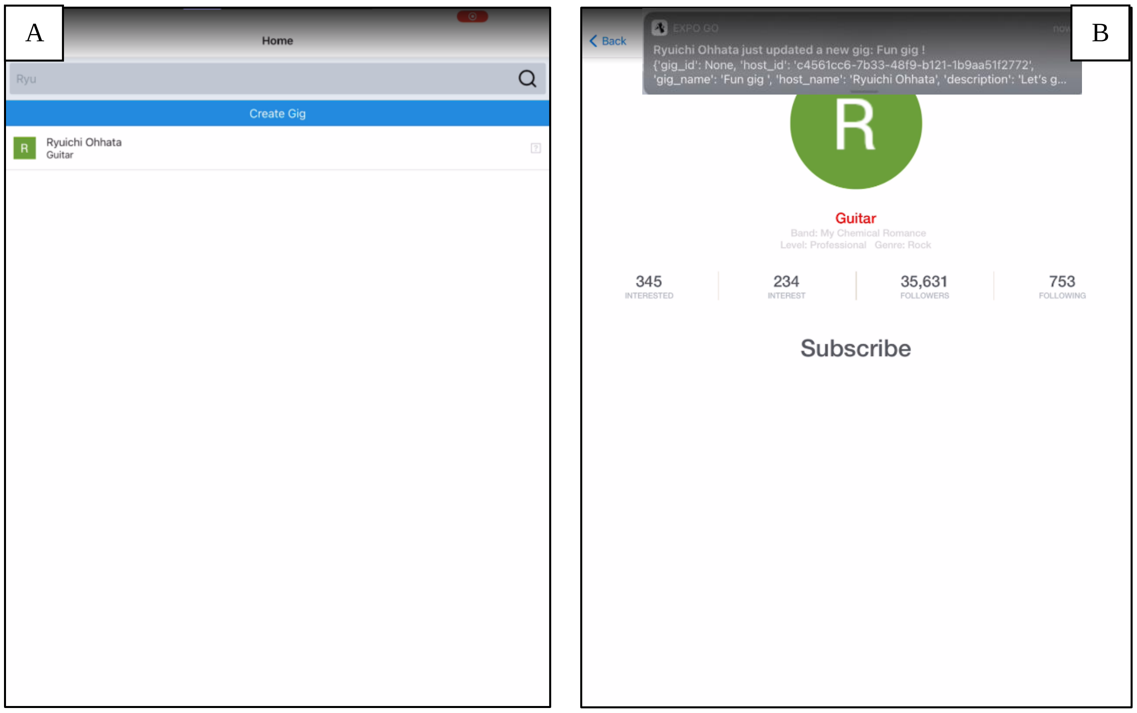


What is Amadeus?

We are designed for musicians to connect and self-promote. Focused on the Boston area, Amadeus will allow users to create a profile in order to showcase themselves playing, tether those profiles to a musical group, and network with other musicians. You can communicate with others and promote events to their network.

Are you sure you want to logout?

Figure 3.4 – The Navigation Flow. A) After clicking on the navigation side bar located at the top left corner, an option of three screens will be shown (i.e., Profile Search, About Us, Logout). B) The About Us screen provides the user a short introduction of the purpose of this application platform. C) The Logout screen allows user to log out of their account by clicking on the "Yes" button then it will redirect back to the login screen.



Ryuichi Ohhata just updated a new gig: Fun gig !
{'gig_id': None, 'host_id': 'c4561cc6-7b33-48f9-b121-1b9aa51f2772', 'gig_name': 'Fun gig ', 'host_name': 'Ryuichi Ohhata', 'description': 'Let's g...

Figure 3.5 – The User Subscription Flow (from a secondary account). A) From the initial homescreen, the user will be able to search for other registered users on the platform. B) Clicking on another registered user will allow for the user to "Subscribe" to the musician. This will allow for the user to receive push notifications when the musician posts new gigs.

3.2 Operating Mode 2: Abnormal Operations

Poor Internet connection will periodically cause certain images to not appear throughout the application. To protect against confusion that this lack of labeling would cause the user, we placed text labels next to nearly every single image used to aid navigation. Moving to an area with better Internet connection and restarting the application should rectify this issue.

Should our current database scheme reach a high capacity (nearly impossible at this level of deployment), the user may experience some slowdown of the application. This is rectified by the developers periodically cleaning the database of duplicate or dead users.

3.3 Safety Issues

As with any online ecosystem, the user must be wary of the data that they share with other users on Amadeus. We ensure that the user knows exactly what pieces of information will be shared with the other users on the platform during the user creation loop.

Furthermore, we encrypt certain pieces of critical user data that is stored within our database. The user should feel safe signing into our application through a third party.

4 Technical Background

4.1 Frontend

The frontend was developed in React Native, with package management done with the Node Package Manager (NPM). React was chosen because it provides a slick, smooth and responsive user interface, while significantly reducing load time. The conversion to a mobile target was handled by Expo, which provided a more consistent mobile target experience than using vanilla React Native on a mobile target.

4.2 Backend

Our backend consists of three main components – a login handler, a database handler, and Kafka. Both the login handler and the database handler are written in Python, but the Kafka component is written in Java due to the language's extensive support for the Kafka technology. Together, they provide APIs to make the frontend fully functional.

The login handler currently supports Google login. When a user logs in with their Google account, it sends the user information to the handler. The handler verifies the user by comparing the "aud" value against the Google Client ID in our backend configuration to authenticate.

The database handler serves as an intermediary between the frontend and the PostgreSQL database. The GET, POST, and DELETE API calls will retrieve, update, and delete rows from the database, specifically in Users, GigPlayers, and Gigs tables. The database handler endpoints are as follow:

- /users
  - GET returns all the users
  - PUT updates the user table
- /user/<id>
  - GET returns one user specified by the id
- /gigs
  - GET returns all the gigs
  - PUT updates both the gig and gigPlayer
  - DELETE delete both the gigPlayer

4.3 Messaging System

We ultimately settled on using Google Firebase for the messaging functionality as it is well-documented, would be simple to implement, and is scalable. A rudimentary UI was then created in React Native that allowed for the creation of a chat room, as well as a Firebase account and project. This interface was then connected to a FireStore database so that the creation of a chat room in the UI would create a chat room document in this database. Each of the chat room documents is configured to contain information regarding the most recent message (which is updated when messages are sent in the app) as well as a collection of documents that each represents an individual message in that chat room.

A reading functionality for the UI was then implemented so that after creating a chat room, the app retrieves a list of chat rooms from the documents in the main collection in the FireStore database. To implement messaging, we utilized the *react-native-gifted-chat* module to provide a user-friendly interface for sending and reading messages after joining a chat room through the associated discover tab. This tab was then connected to the FireStore database by storing individual messages as documents within the chat room collections. These message documents contain the actual text sent in the message as well as other metadata, including the ID of the user, the username of the user, and the creation time of the message. Messages are cached on the user's device once received using SQLite to reduce latency between the opening of a chat room and the displaying of messages. Caching the messages locally also allows for the future iterations of the app to allow for the user to brose messages while offline (a feature not available in the current version of the application). A full overview of this system is depicted in Figure 4.1.

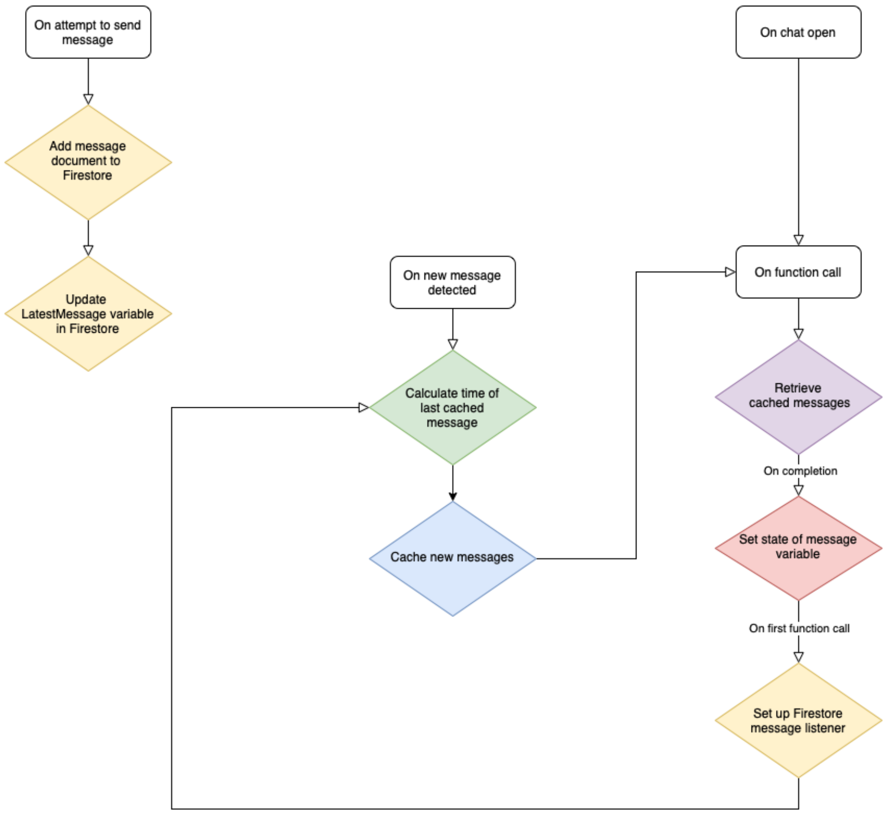

On attempt to send message
Add message document to Firestore
Update LatestMessage variable in Firestore
On new message detected
Calculate time of last cached message
Cache new messages
On chat open
On function call
Retrieve cached messages
On completion
Set state of message variable
On first function call
Set up Firestore message listener

Figure 4.1 – An Overview of the Messaging System

4.4 Gig Creation System

The gig creation system exists at the intersection of the database and the Google Maps API. Using the location autofill features provided by the Google API, we can help the user pinpoint exact locations that will get uploaded to our database. The exact fields that the application requests for a gig can be seen in Figure 3.2 (c). These fields are stored in our database and are tethered to the existing user.

Upon switching to the map screen, the existing gigs from both the active user and every other user in the application will be pinned on the map, along with the gig's name, description, and genre. The exact latitude and longitude can be pulled from the Google databases.

4.5 Stream Processing

Kafka Streams will be used to handle notifications in the application. These streams will grant users the ability to receive notifications about new activities from their subscribed musicians in real-time.

First, users subscribe to musicians which trigger the Kafka producer in the backend to send records. These records are key-value pairs under a topic dubbed the "user-subscription." In this case, the Musician ID will be the key and the User ID will be the value.

Second, subscribed musicians send updated activities which trigger the Kafka producer to send records under a different topic named "musician-updates." The musician ID will be the key and the activity will be the value, which is stored in JSON format.

Next, the Kafka enricher will match the key (musician ID) from the user-subscription topic and the musician-updates topic and make one record with the User ID as its key and information about the updated activity as its value, sending it to the Kafka cluster under a topic user-notification.

Finally, the Kafka consumer subscribed to the user-notification topic retrieves only the records with the corresponding user ID to get the notifications from their subscribed musicians. The cluster's retention period of the records is configurable so the user-subscription records will be periodically "fired" to the cluster by the accumulator. All the records will be simultaneously stored in the PostgreSQL via a connector.

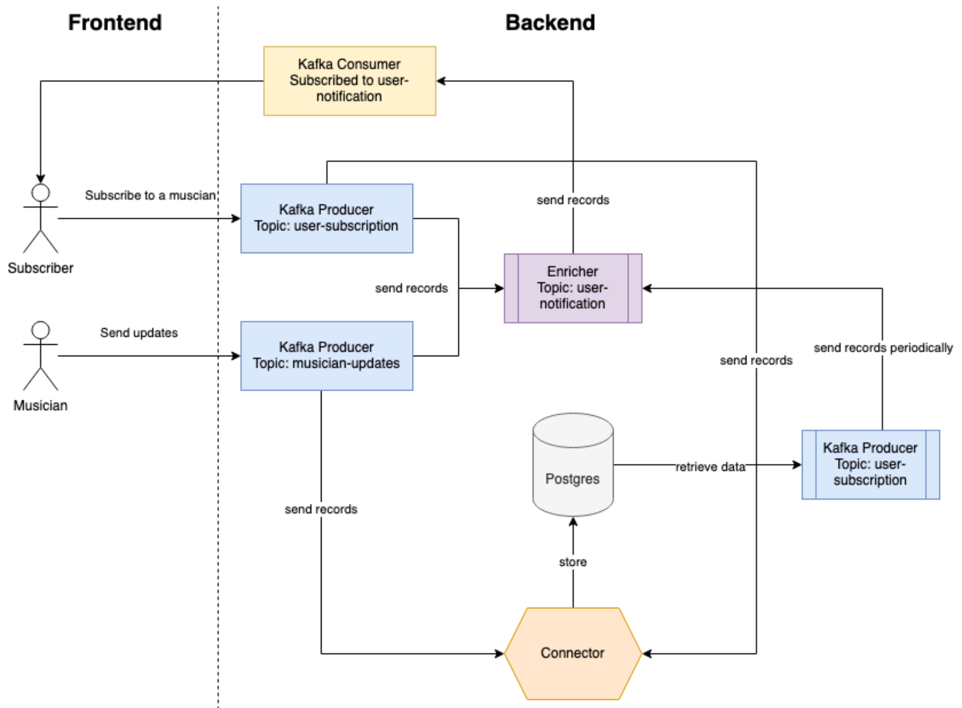

Frontend
Backend
Kafka Consumer Subscribed to user-notification
Subscribe to a muscian
Subscriber
Kafka Producer Topic: user-subscription
send records
Enricher Topic: user-notification
send records
Send updates
Musician
Kafka Producer Topic: musician-updates
send records
send records periodically
Postgres
retrieve data
Kafka Producer Topic: user-subscription
send records
store
Connector

Figure 4.2 - An Overview of the Notification System

5 Relevant Engineering Standards

Amadeus was developed in complete adherence with both the Apple Human Interface Guidelines, an overview of best practices when developing customer-facing applications. The primary aspects of the application design guidelines that we have implemented in the application are as follows -

1. Aesthetic Integrity
    a. “Aesthetic integrity represents how well an app’s appearance and behavior integrate with its function. For example, an app that helps people perform a specific task can keep them focused by using subtle, unobtrusive graphics, standard controls, and predictable behaviors.”
2. Clarity
    a. “Throughout the system, text is legible at every size, icons are precise and lucid, adornments are subtle and appropriate, and a sharpened focus on functionality motivates the design. Negative space, color, fonts, graphics, and interface elements subtly highlight important content and convey interactivity.”
3. User control
    a. “The best apps find the correct balance between enabling users and avoiding unwanted outcomes. An app can make people feel like they’re in control by keeping interactive elements familiar and predictable, confirming destructive actions, and making it easy to cancel operations, even when they’re already underway.”

Amadeus additionally pulls from the App Store Review Guidelines as we would ultimately like to submit our application to the App Store. Accounting for these guidelines while developing the application should make the App store review process go substantially smoother. We made special note of the following guidelines during the building of the application –

1. Safety
    a. “When people install an app from the App Store, they want to feel confident that it’s safe to do so - that the app doesn’t contain upsetting or offensive content, won’t damage their device, and isn’t likely to cause physical harm from its use.”
2. Design
    a. “Apple customers place a high value on products that are simple, refined, innovative, and easy to use, and that’s what we want to see on the App Store… And remember that even after your app has been approved, you should update your app to ensure it remains functional and engaging to new and existing customers. Apps that stop working or offer a degraded experience may be removed from the App Store at any time.”
3. Performance
    a. “Make sure your app has been tested on-device for bugs and stability before you submit it and include demo account info (and turn on your back-end service!) if your app includes a login.”

6 Cost Breakdown

Because our project is entirely software-based, we can avoid many of the costs that arise with building a more traditional project. With that said, there are still several components that require monetary attention.

The data that needs to be stored for each user includes the user's name, email, date of birth, type of genre, instrument, pictures, videos, the number of followers, the number of people the user follows, and interests. Additionally, our messaging platform and gig information (i.e., location, band information, time, and date) requires storage space. Lastly, the developer fee for publishing our mobile application to the Apple App Store is $124.

In short, the estimate budget to support our initial estimate of 1000 users per month with the initial publishing fee is $234.74 for the first month and an estimated average of $135.79 per month subsequently for continued use of AWS.

Project Costs for Production of Beta Version				
Item	Quantity	Description	Unit Cost ($)	Extended Cost ($)
1	1	App Store Publishing Fee	99.00	99.00
2	1 (Recurring)	AWS EC2 Instance – 10GB	62.54	62.54
3	1 (Recurring)	AWS RDS for PostgreSQL – 100GB	69.78	69.78
4	1 (Recurring)	Firebase Read/Write Messaging Storage	3.47	3.47
			Beta Version-Total Cost (First Month, Est. 1000 Users)	234.74
			Beta Version-Total Cost (Subsequent Months, Est. 1000 Users)	135.79

7 Appendices

7.1 *Appendix A – Specifications*

Requirement	Value / Range / Tolerance / Units
User Database Capacity	>3000 users
Messaging Success	< 1% Do not send
Location Accuracy	< 0.1-mile location error
Load Times (After initial install)	< 1 second with strong network

7.2 *Appendix B – Team Information*

Ryuichi Ohhata built the overarching architecture as well as the tools for managing data. He will be working for HPE in San Jose, CA as a Software Engineer after graduation.

Brandon DeSiata built the messaging functionality and associated architecture.
He will be working for John Hancock in Boston, MA as a Software Engineer after graduation.

Samantha Au built the user interface.
She will be working for Accenture in Los Angeles, CA as an Analyst after graduation.

James Wasson built the Map functionality.
He will be commissioning as an officer in the United States Army after graduation.

Dev Bhatia worked on team management, documentation, and DevOps.
He will be working for Raytheon in Marlborough, MA as a Software Engineer after graduation.

7.3 *Appendix C – Entrepreneurship*

Ultimately, we see this platform as a springboard for many musicians to grow confidence in their skills, collaborate with each other, and develop incredible music together. By implementing the above requirements, this project should be easily transitioned to the open market upon completion. Though a highly competitive market, Amadeus carves a comfortable niche in the musical community - one that we plan to continue to learn more about as we develop the service.

This entrepreneurial spirit was fostered by Innovation@BU and their Innovation Pathway (IP) – a four-step program to acquire funding from BU, as well as access to coaches, mentors, and peer-to-peer learning. We are currently applying to enter step three of the program (labeled the “Run Stage”), in which we will have verified our early validation research and begin to form Amadeus into a more viable venture idea with a compelling business model. Entering this stage will provide us with early funding for Amadeus in the form of $1000 to be used on the costs of server hosting, developer publishing fees, and further iterating on the application.