Boston University

Department of Electrical & Computer Engineering

First Semester Report

EC463 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

December 12, 2021

Contents

0.1 Executive Summary ii

1 Introduction **1**
1.1 Problem Definition and Purpose 1
1.2 Approach and Key Features 1
1.2.1 High-Level Approach - Software Design Life Cycle (SDLC) and Organization 1
1.2.2 Product Highlights 1
1.2.3 Key Features 2

2 Concept Development **3**
2.1 Inception and Elaboration 3
2.2 Translation to Formal Requirements 4
2.3 Alternate Concepts 5

3 System Description **6**
3.1 Solution Overview 6
3.2 System Model and Architecture 6
3.2.1 Component Decisions 6
3.2.2 Architecture Overview 7
3.2.3 Software Overview 7

4 First Semester Progress **9**
4.1 Backend Development 9
4.1.1 API Development 9
4.1.2 Messaging System 10
4.2 Frontend Development 10
4.2.1 User Interface 10
4.2.2 Map 11
4.3 Prototype Testing Results 11

5 Technical Plan **14**
5.1 Merging - First Round 14
5.2 Rudimentary Testing 14
5.3 UI Beautification 14
5.4 Map Improvement 15
5.5 API Expansion 15
5.6 Messaging System Updates 15
5.7 Notifications 16
5.8 Re-Merge 17

5.9 Functional Testing 17

6 Budget Estimates **18**

7 Appendix **19**
7.1 Appendix 1 - Engineering Requirements 19
7.2 Appendix 2 - GANTT Chart 20

Bibliography **21**

0.1 Executive Summary

Team 7 - Amadeus

Amadeus is a mobile application designed for musicians to connect and self-promote. The application 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. 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 that registered users can add entries to.

Chapter 1

Introduction

We begin this report by providing a broad overview to Amadeus and its many facets. By detailing a summary of what the product will be, as well as the general need for the product in the market, we demonstrate the effective uses for Amadeus in a variety of capacities.

1.1 Problem Definition and Purpose

Through a series of client interviews that we have performed, we have identified three central problems that are pervasive throughout the musician community -

1. Lack of a **Person** to play with
2. Lack of a **Venue** to attend
3. Lack of a **Centralized Location** to acquire such information

To solve these three problems, we ultimately propose Amadeus. The proposed 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.

1.2 Approach and Key Features

1.2.1 High-Level Approach - Software Design Life Cycle (SDLC) and Organization

In order to have an organized process in which we could make rapid updates to the application, we strictly adhered to an agile methodology for the software development life cycle. This iterative approach allowed for requirements, plans, and results to be evaluated continuously so the team had a mechanism for responding to change quickly. Following such a methodology also helped in structuring our commits and merges, making them less taxing than a traditional waterfall approach.

1.2.2 Product Highlights

To best address the needs of an incredibly diverse market base, Amadeus will offer two primary services - connecting local musicians and displaying local shows.

In order to best connect such a large group of people, we will be keeping several key metrics of each user that can be used in filtering potential bandmates. Amongst these are skill-level, location, primary instruments, and genre. Users will also be able to showcase videos on their profile.

For displaying local shows, registered users will be granted the ability to specify the details of a local gig which will be broadcasted to all users in the area. Users can search for these shows via a map within the application

1.2.3 Key Features

A large part of developing this application centered around identified the primary facets necessary in solving the aforementioned problems. As a result, we want to implement five core features - each of which was/will be the core task of a team member.

- Profiles + Search Functionality
- User-to-user communication
- Messaging board
- Authentication
- Map data

Many of these features are nearing completion and are detailed in Chapter 4. Implementing these aspects of the application also require a comprehensive and complex database and backend structure, much of which was the primary focus of the first semester and is further detailed in Chapter 3.

Chapter 2

Concept Development

In this section we aim to elaborate upon the customers' problems and describe the translation into concrete requirements. This will enable us to robustly define the required components of the project.

2.1 Inception and Elaboration

In order 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 idea to be refined heavily and the problem narrowed to smaller key areas. The key takeaways from the interviews are as follows -

- Younger musicians (especially those without an established network) don't have an existing solution
- Finding potential bandmates is a tricky prospect, despite having large-scale social media platforms
- A "try before you buy" approach was highly sought after
- The "business aspects" of starting a band are very prohibitive for many musicians
- Established (older) musicians preferred the "Show Promotion" feature whereas new musicians (younger) preferred the "Connect Local Musicians" feature
- A significant number of musicians preferred to be solo

These insights 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 jamming 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.

2.2 Translation to Formal Requirements

Though the interviews gave many insights, there was still the need for us to translate the desires of our customer base into concrete requirements for the purposes of application development. From the aforementioned takeaways, we were able to form the core of Amadeus with the original mock up in Figure 2.1. Though the architecture has evolved immensely since this system was designed, it serves as the initial base-point for translating the requirements into tangible goals.

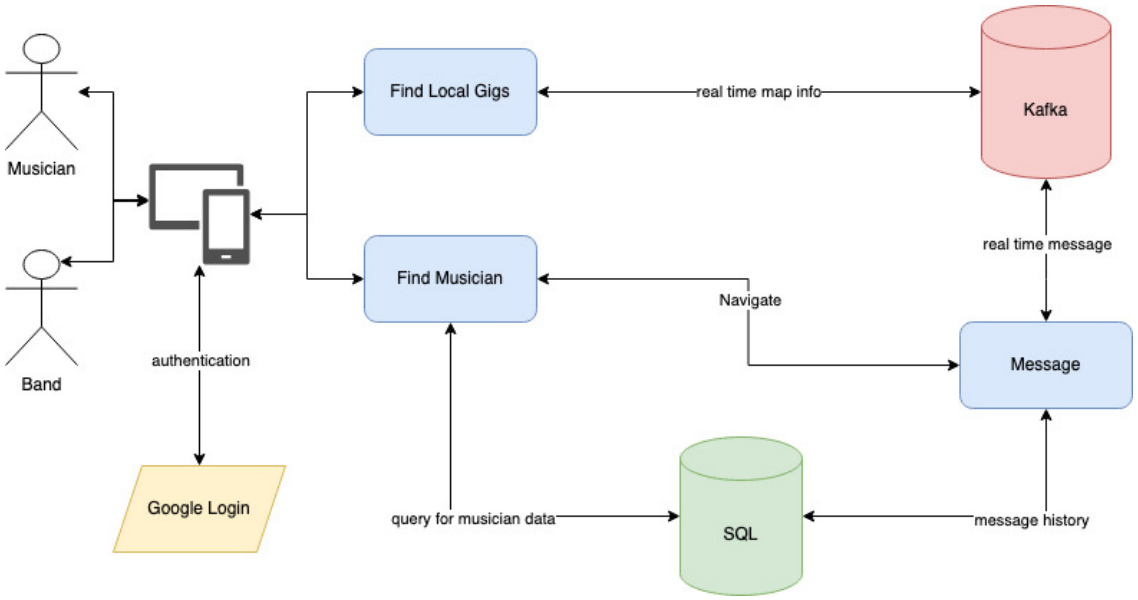

Figure 2.1: Initial Mockup of the Amadeus Architecture

The process of creating the figure was the catalyst for organizing the initial **feature** requirements for the application. From there, it became clear that our requirements would need to contribute to an efficient and fault-tolerant software system. Some of the most important requirements can be found below, with a complete list in Appendix 1.

- **Streamlined Design** - The app must be easily used by a wide range (>80%) of users. The general workflow must be transparent, and adhere to the Apple AppStore technical specifications
- **Reliable and Efficient Communication** - Messaging should happen with very infrequent erros (<1% Do Not Send), and deliver relatively quickly (¡ 30 seconds with strong network)
- **Accurate Location Data** - The location data must be able to pinpoint an exact location (address) on the generated map (+/- 0.5 miles)
- **Secure Backend** - Our application should securely store passwords and user data, subject to pen-tests

2.3 Alternate Concepts

Amadeus, as it stands currently, is the product of hours upon hours of iteration. Early conceptual models suggested that we split the two halves of the project entirely - creating a platform for connecting musicians (akin to a Tinder or LinkedIn) and a platform for finding local gigs (akin to a more focused google maps search).

These early models were scrapped rather quickly when looking at the competitive landscape of the music space. We noticed platforms like MUZE and Bandfriend were similar in concept to Amadeus, but failed to do anything after simply connecting musicians - a core reason as to why the platforms have a faltering userbase. To avoid this pitfall, we elected to merge the two sides of the Amadeus coin.

Additionally, several tools and technologies were considered before arriving at the current slate. These are elaborated upon in Chapter 3.

Chapter 3

System Description

Chapter 3 details the ultimate goal of the solution introduced in Chapter 1. Based on the requirements detailed in Appendix 1, we introduce an updated and more intricate system to deliver a solution to the aforementioned problems.

3.1 Solution Overview

At the macro scale, Amadeus is an application that will provide musicians the capability to do several things. What follows are the primary features.

- Establish a profile
- Tether that profile to a group
- View events on a map
- Add entries to a map

In order to facilitate the data transfer and stream processing required by this system while building out the UI, we require a decoupled architecture. A decoupled architecture is an architectural approach that allows each individual component of the application to exist and perform tasks independently of one another, unless otherwise directed [1]. This architectural approach can be seen in figure 3.1.

3.2 System Model and Architecture

3.2.1 Component Decisions

We chose React Native and Expo as our frontend framework and Python Flask as our backend framework. React Native was chosen for its low barrier to entry, as well as its extensive library of application plugins compatible with IOS environments. As for the backend, one of the team members had experience with building back-end servers using Python Flask and RESTful, which would be responsible for creating the endpoints.

Also in the backend, Kafka was chosen to be responsible for the real-time processing and notification system. After deliberating upon the best way to implement a chat room functionality, and we found Google Firebase to be the best API to develop the chat room for our purposes. PostgreSQL was chosen to be our main database, as we

required a relational database to handle the user's relations with their data including genre, live shows attended, or instrument they play. PSQL additionally supports non-relational data which increases the freedom with which we can iterate upon the existing application.

3.2.2 Architecture Overview

The significance of the decoupled architecture to the application becomes apparent when looking at Figure 3.1. The user creation screen will be the main entry point for a new user (i.e. a user that isn't already in the database). A tether to the user handler and login handler in the API ensure 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 messages page. The current status of these pages can be found in Chapter 4.

3.2.3 Software Overview

The Oauth Procedure is the following [2] -

- The application requests authorization to access service resources from the user
- If the user authorized the request, the application receives an authorization grant
- The application requests an access token from the authorization server (API) by presenting authentication of its own identity, and the authorization grant
- If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application. Authorization is complete.

That authorization token is what will be used throughout our application to verify the user, but only the user's email (typically tethered to the auth token) address will be stored in the database. The database querying process possesses some glaring similarities to the Authorization procedure.

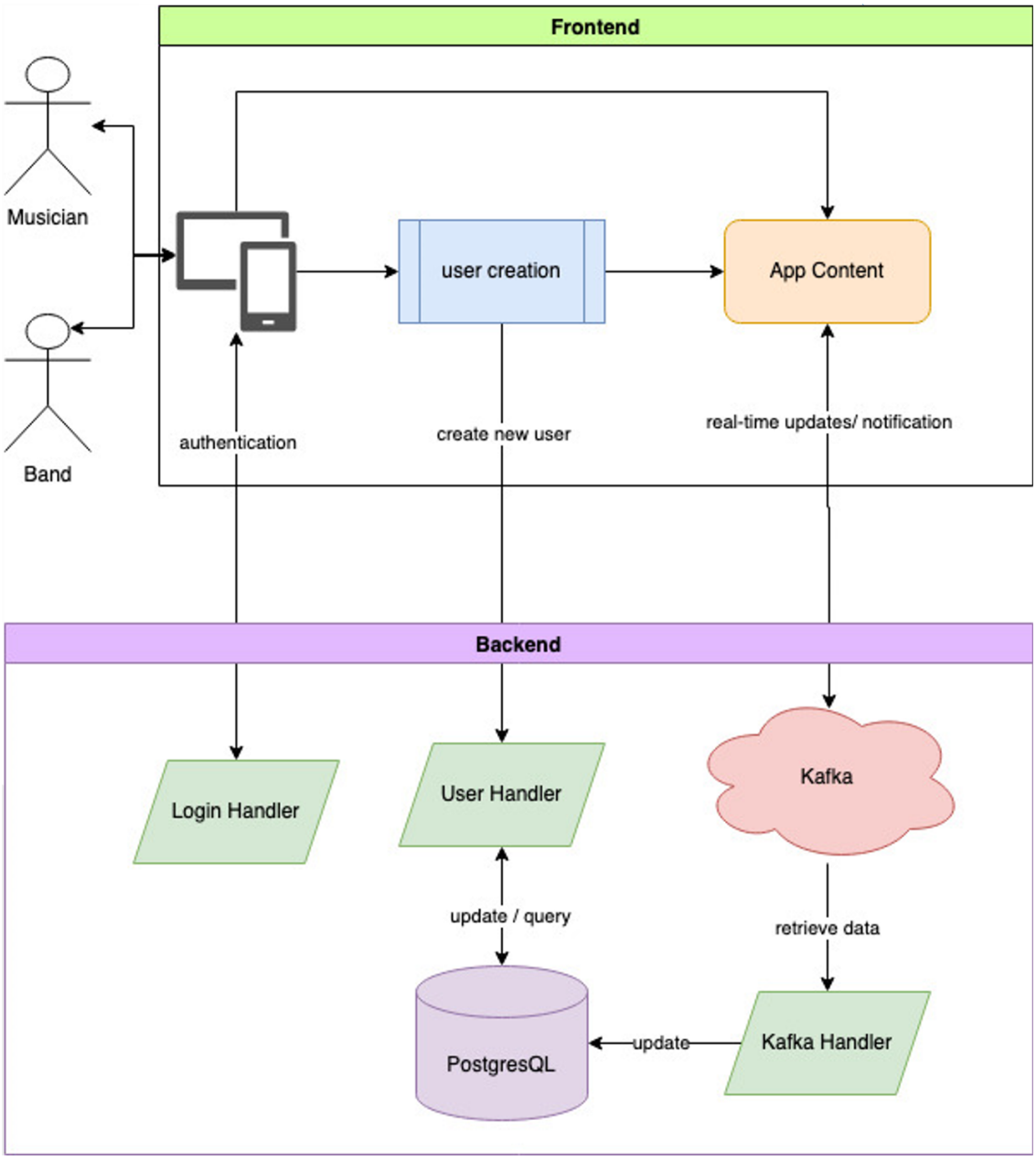

Figure 3.1: Amadeus' Software Architecture

Chapter 4

First Semester Progress

Chapter 4 details the cumulative progress of the team in working towards the final deliverable. Figure 4.1 demonstrates the established screen flow scheme.

4.1 Backend Development

4.1.1 API Development

The team has constructed the APIs in the backend server and linked those APIs with the PostgreSQL database using SQLAlchemy. Three endpoints have been developed at this point:

1. /api/v1/users
   This endpoint is responsible for handling GET, POST, and PUT requests and interacting with the database. The GET request will get all the user information, POST request will create a new user, and PUT request will update an existing user in the database.

2. /api/v1/user/<id>
   This endpoint gets GET requests, returning one user with the ID specified in the query string in the URL. It helps find a specific user from the database.

3. /login
   This endpoint handles login request sent in a form of POST request. It takes ID tokens of the user's Google account, then calls Google API to verify the user. Upon successful verification, it returns the user information to the client side (frontend side). If the verification fails, it returns a failure code so the client side will not proceed to the next step.

In order to link the APIs with the database, we created database models using SQLAlchemy for each table. An example may be found at the following link - *[Click Me]*.

Lastly, some members experienced issues with running PostgreSQL on their machines. To address the problem, we acquired an old laptop and configured it to run as an SQL server, ensuring that team members do not have to deal with the problem of running the database locally anymore.

4.1.2 Messaging System

We ultimately settled on using Google FireBase for the messaging functionality as it is well-documented, would be fairly 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 were 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. This was then connected to the FireStore database by storing individual messages as documents within the aforementioned 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.

4.2 Frontend Development

4.2.1 User Interface

Navigation

To allow for users to easily navigate the application, we elected to implement a rudimentary navigation bar. This bar would allow users to use various features of the application without needing to fully switch screens.

Through the process of integrating the navigation bar, we found that including the profile button with the find musicians screen can be confusing for users. As a result, we decided to add a profile tab on the navigation bar to ease the process for our potential clients. For the other two pages (messages and find gigs), sample screens were created to aid with the integration process.

Profile Screen

The profile system was designed with searchability in mind, so each profile was required to have a series of information displayed. This was mostly done by including extended versions of styles from the StyleSheet provided within React Native library. For instance, the profile image uses styles of width and height of 200 with borderRadius of 100 and hidden overflow to create the centered, circular profile image on the page.

Next, a more formal profile page was built to include the user's name, title, and musical background information. This was supplemented with the number of interested, interests, following, and followers so that other users can reference to learn more about the specific user. A recent activity section was subsequently added to showcase what other musicians the user is interested in.

Find Musicians Screen

Most recently we have been working on a "find musicians" screen which is designed to allow musicians to search for other musicians nearby. Much of the code for this screen currently appears to be redundant, but it serves a very important purpose - being a placeholder for later integration with the API.

Within the screen itself, we have been working on displaying other musician's profiles with their profile image as well as having their names as clickable links such that the user can tap into their respective profiles. Because the clickable link has more than one component (i,e. image and text), we made use of the TouchableOpacity module instead of buttons to create each user's profile.

Through more integration work, this was implanted into the navigation bar. Stack screens allow for a new page that is not navigated from the navigation bar with stacks, as touched upon in the Navigation Bar.

4.2.2 Map

We were able to successfully implant the google maps API within the application, alongside some of the features it offers. Currently a user is able to change the location of the map using coordinates, create a draggable cursor, and create a circle around the current position with a selected radius. With that said, it also contains a lot of extra map details that are not necessary for our app's purpose, and will need to be combed through before deployment to ensure a lack of bloat.

4.3 Prototype Testing Results

The central goal of testing was to ensure that the database could communicate with the frontend and backend of the application. The measurements taken revolved around that specifically - effectively yes/no based on if each step in Figure 4.2 occurred successfully. Success was primarily evaluated in the form of data being successfully being transferred between each screen and the database. More specifically:

1. After arriving at the home screen, the postgresQL database should have been populated with the following fields for a single user:
    - Name
    - Email
    - Date of Birth
    - Genre
    - Instrument
    - Picture
2. Every screen (besides the login screen) should display the user's name, pulled from the google authentication
3. The application should compile in the development server without errors
4. The pressing of the "continue" button in each screen should progressively upload the targeted field to the database and shift to the following screen

Following the pipeline in Figure 4.2 allowed for us to demonstrate that the API worked as intended, and could be used throughout the various parts of the application.

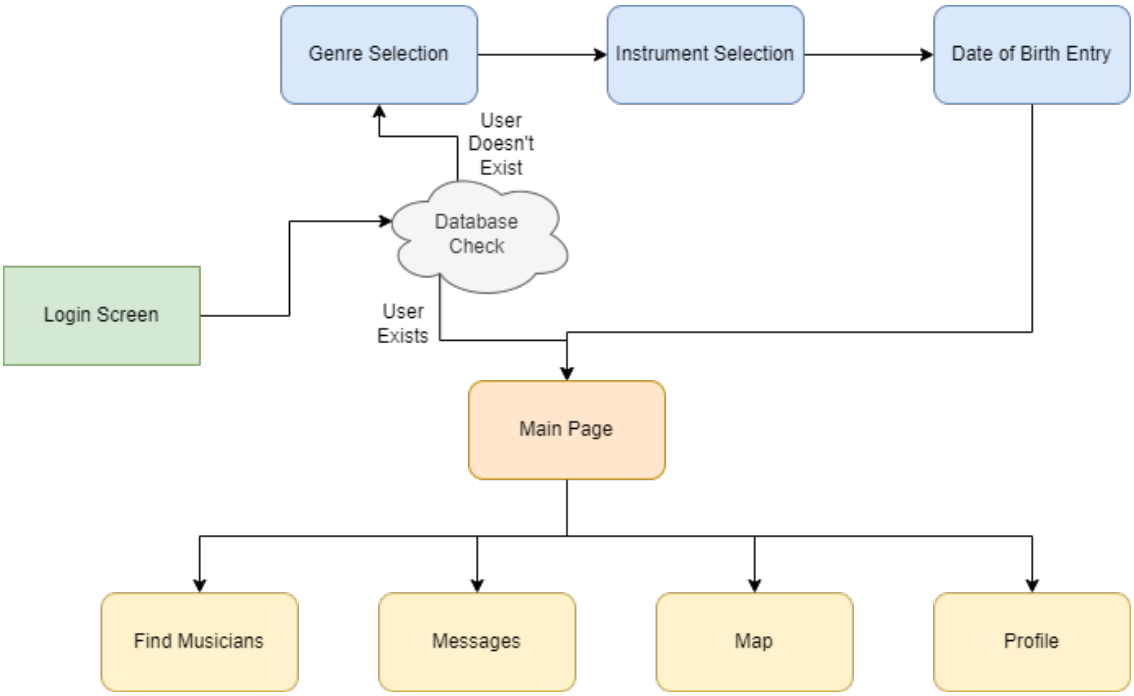

Figure 4.1: Screen Flow Schematic

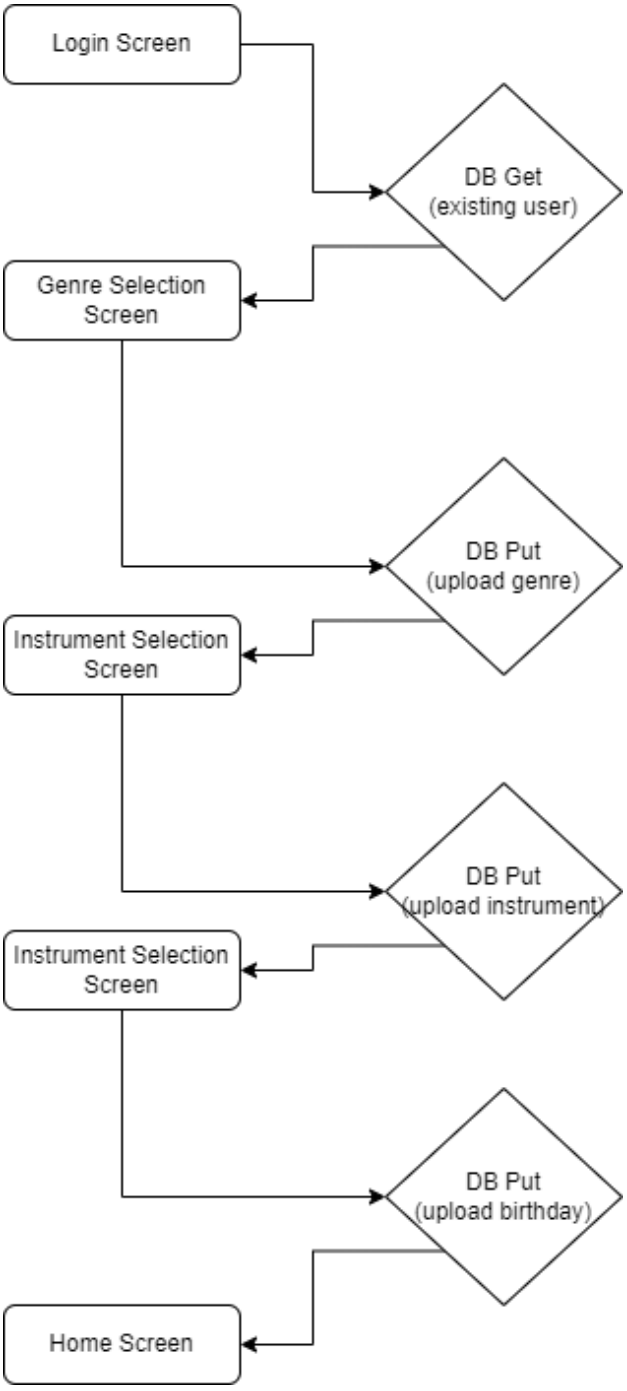

Figure 4.2: Prototype Demonstration Flow

Chapter 5

Technical Plan

This section will serve as an overview of the plan to achieve the final deliverable. These points are summarized in the GANTT Chart in Appendix 2.

5.1 Merging - First Round

Though we are currently in the midst of a large-scale integration effort, there still is much to be done. The Map and Messaging functionalities have not yet been merged into the main application, and will require special attention to do so.

Merging the messaging and map functionalities comes with its own set of challenges, primarily dealing with merge conflicts.

Lead: Ryuichi Ohhata
Assisting: Team

5.2 Rudimentary Testing

After merging the pieces of the application, we would like to perform an intense round of testing before we move to update the features. This unit testing will allow us to not only ensure that the application is being built to desired specifications, but it will also offer invaluable feedback on the modularity and design of the code being developed. [3]

This will be a major milestone for Amadeus as it will be the first time that all of the individual components have been integrated and made to work with each other. This will undoubtedly uncover unexpected problems in the existing codebase, ultimately resulting in a better finished product.

Lead: Dev Bhatia
Assisting: Team

5.3 UI Beautification

As it currently stands, the user-interface appears very simple. This was important for testing, as getting to specific portions of the application for testing became efficient. We would like to retain the efficiency, but beautify the application heavily.

This will entail creating a series of customized user interfaces for each screen, stemming from our knowledge of the users (including understanding their goals, skills, prefer-

ences, and tendencies). We seek to implement the following widely accepted guidelines for user interface design [4] -

- Keep the interface simple
- Create consistency and use common UI elements
- Use typography to create hierarchy and clarity
- Make sure that the system communicates what's happening

The last point is a particularly important one, as gracefully handling errors is not something that was a focus in development up to this point.

Lead: Samantha Au
Assisting: Dev Bhatia

5.4 Map Improvement

Expanding the capabilities of the current map will be imperative in realizing one of the original goals of Amadeus - allowing musicians to upload and search for local gigs.

A large portion of next semester will be spent in building the tools for musicians to detail the location of a gig and uploading it to the database. These details should be pervasive for every user of the application.

Lead: James Wasson
Assisting: Dev Bhatia

5.5 API Expansion

The API expansion will be critical in order to support more features as more database tables are created. The login API needs to not only return the user verification information but also enable the authentication session for the users. This will store the users' state information and require them to log in again when timed out. The API will need to support Facebook login as well as the standard login method that does not use third party's login APIs, such as Google and Facebook.

There will be new API endpoints for interacting with the gig information in the database. These endpoints will be called when the users are interacting with the map function to find gig sites. The map API will retrieve the gig information from the database based on the selected location, then send it to the client side to display them. The API will also be responsible for uploading information. For example, if a band wants to announce its gig information, the API will create a new entry in the database accordingly.

Lead: Ryuichi Ohhata
Assisting: Dev Bhatia

5.6 Messaging System Updates

After messaging functionality is merged with the rest of the frontend and configured so that chat rooms are shown only to users that belong to them, we plan on implementing

a local cache utilizing SQLite to eliminate the perceptible delay between a message retrieval request from the app and the population of the user's message list. We plan on later encrypting these local messages, but our initial implementation of this cache functionality will not contain encryption functionality. Once local caching is implemented, we seek the implementation of image sharing through this messaging system, giving musicians a wider array of communication abilities. We are currently considering setting up an additional database to store images and configuring our message documents in our Firestore database to have a field for these links. This additional database would allow for images to be stored in a more cost-efficient manner as Firestore data storage is more expensive than other services.

Lead: Brandon Desiata
Assisting: Ryuichi Ohhata

5.7 Notifications

When an user follows bands, it is essential to notify the user of the bands' activities. The notification system will utilize Kafka, a stream processing framework. Whenever there is a new update from a band, a Kafka Producer produces a message to a topic, which a Kafka Consumer will consume in the client side, delivering the notifications to the users. These messages in the topic may or may not be enriched or processed in real-time based on the needs. Kafka can also be used to track the user activity given its ability of real-time processing. Such Kafka pipeline will be most likely to be developed using the Spring Boot framework in Java for its abundant supports in Kafka Stream Libraries.

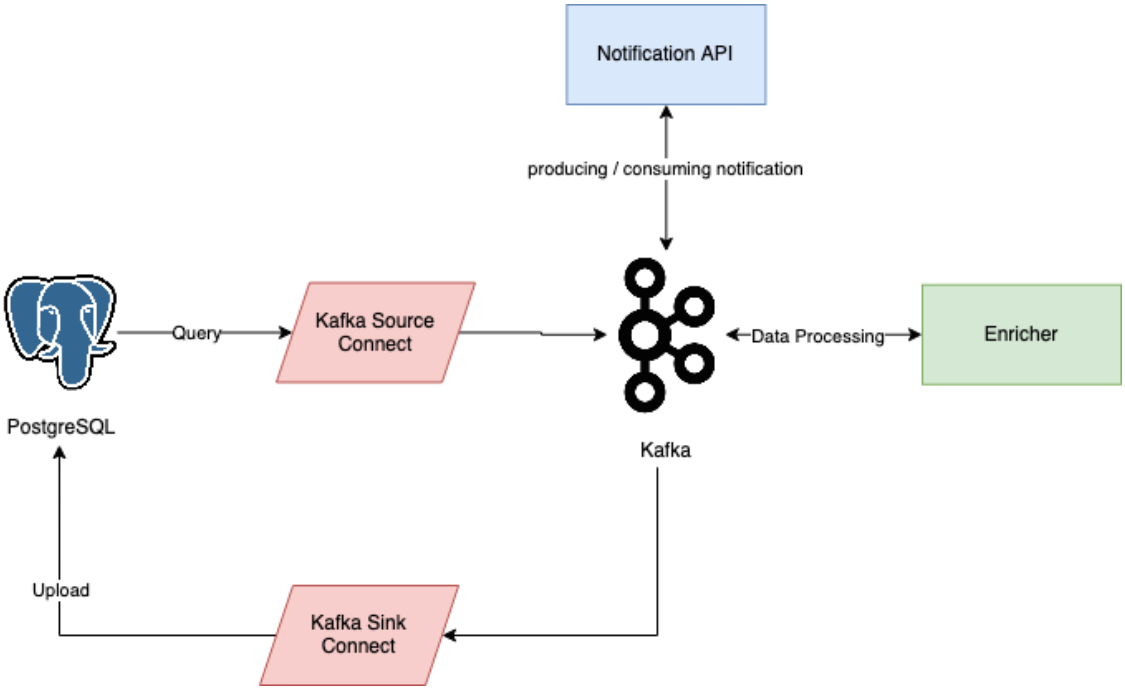

Figure 5.1: Notification System

Figure 5.1 shows the architecture of the notification system. Kafka Connect will directly interact with the PostgreSQL database for both querying and uploading data.

All the notification messages from the app will go to the notification API, which is responsible for both producing and consuming Kafka records. Those Kafka records will be sent to a Kafka topic, enriched if necessary, and stored in the database, changing the states. Once changes in the states are observed, Kafka Connect will stream the new states to a Kafka topic as records, then a Kafka Consumer in the Kafka API will consume the records and distribute them to the appropriate receivers.

Lead: Ryuichi Ohhata
Assisting: Brandon Desiata

5.8 Re-Merge

After tweaking each individual facet of the application, there will need to be another dedicated period of integration. This period of integration will likely have more issues that arise when compared to the first round, as a result o the increased complexity of each feature. By this point, each feature's frontend will also have a complex relationship with the backend (either the database or kafka) that must be preserved when integrating within the context of the wider service.

Lead: Ryuichi Ohhata
Assisting: Team

5.9 Functional Testing

After the final merging process we will commence another round of testing. This testing will be focused more on the user experience as a whole, and ensuring that each individual component accurately fits the original specification.

Lead: Dev Bhatia
Assisting: Team

Chapter 6

Budget Estimates

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 a small number of components that require monetary attention.

The estimate data that needs to be stored per user includes: name, email, date of birth, type of genre, instrument, pictures, videos, the number of followers, following, and interests. Additionally, our messaging platform as well as gigs information (i.e. location, band information, time & date) that will be stored along our map requires storage space. Lastly, the developer fee for publishing our mobile application to the Apple App Store and Google Play Store is 99 + 25 = 124 (USD).

This project is not exactly bound by the budget constraint as it is software based and can be ran locally, including both back-end and database. However, below is the cost estimation for supporting 1000 users for a month. An AWS EC2 instance will be used for the back-end and API services. 10GB of the storage is enough for our use case, as this instance does not store anything other than the back-end codes. For the PostgreSQL database, AWS RDS for PostgreSQL with 100GB will be used, so roughly each user is granted with 100MB for their account storing user information, pictures, videos, and gigs information. Additionally, the estimate cost for the messaging storage within Google Firebase including read/write cost with an estimate of 50K free read cost + 30K reads at 0.06/100K, 20K free writes cost, 20K free delete cost and storage/networking cost with an estimate of 3GB free egress, 1GB free storage with estimate of 3KB daily message storage/ DAU + 2KB storage/install which results 1.35GB monthly storage usage would be roughly 3.47 (USD) for supporting 1000 users per month.

In short, the estimate budget to support our initial estimate of 3000 users per month with the initial publishing fee is 531.37 (USD) for the first month and an estimate average of 135.79 (USD) per month after.

Budget List (per 1000 users, per month)		
Item	Description	Cost (USD)
1	AWS EC2 10GB	62.54
2	AWS RDS for PostgreSQL, db.m5.large, 18 utilization, On Demand, 100GB (DB)	69.78
3	Firebase for Read/Write, storage/networking costs for messaging	3.47
	Total Cost per month	135.79

Chapter 7

Appendix

7.1 Appendix 1 - Engineering Requirements

Requirement	Value, range, tolerance, units
Ease to use UI	$>$ 80% of Users
Server capacity	$>$ 50 simultaneous logins
Database Capacity	$>$ 3000 Users
Notification & Messaging Delays	$<$ 30 Seconds
Database Querying	$>$ 2 musicians per second
Location Pinpointing	$<$ 0.1 mile error
Map Data Load Time	$<$ 5 Seconds

7.2 Appendix 2 - GANTT Chart

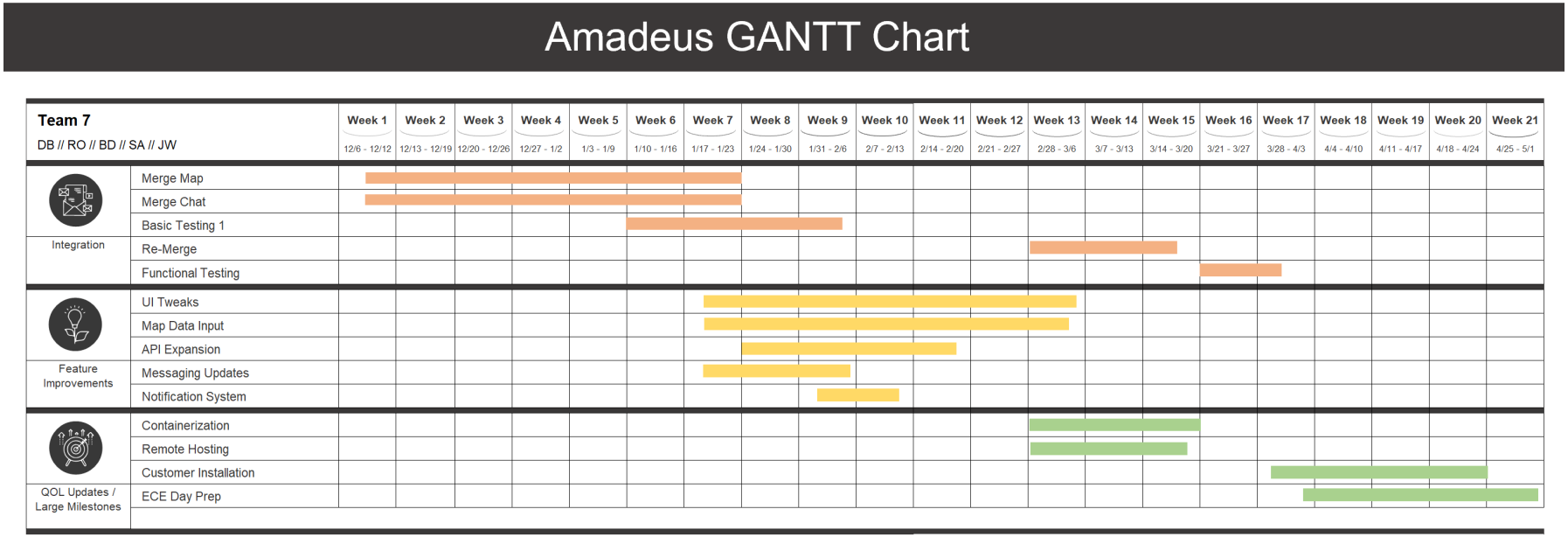

Figure 7.1: GANTT CHART

Bibliography

[1] Indicative, *What is decoupled architecture? data defined*, Sep. 2021. [Online]. Available: https://www.indicative.com/resource/decoupled-architecture/.

[2] DigitalOcean, *An introduction to oauth 2*, Nov. 2021. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2.

[3] TechTarget, *What is unit testing? definition from whatis.com*, Aug. 2019. [Online]. Available: https://searchsoftwarequality.techtarget.com/definition/unit-testing#:~:text=Unit%5C%20testing%5C%20is%5C%20a%5C%20software,developers%5C%20and%5C%20sometimes%5C%20QA%5C%20staff..

[4] Usability.gov, *User interface design basics*, May 2014. [Online]. Available: https://www.usability.gov/what-and-why/user-interface-design.html.