New York Institute of Technology
Department of Computer Science
CSCI 455
Senior Project Technical Report
Fall 2020 Semester
Prof. Maherukh Akhtar

Connect NY
Team Connect NY

Stefan Teau
steau@nyit.edu
+4 (075) 402 - 0088
ID: 1248951

Jennifer Ramdin
jramdi01@nyit.edu
+1 (347) 697 - 5084
ID:1197516

Samiha Gaffar
sgaffar@nyit.edu
+1 (516) 325 - 0057
ID: 1202753

Almir
akazafer@nyit.edu
+1 (347) 861 - 3650
ID: 0854440

Gabriel
gkushner@nyit.edu
+1 (718) 702 - 3517
ID: 105002

Table of Contents

Table of Figures

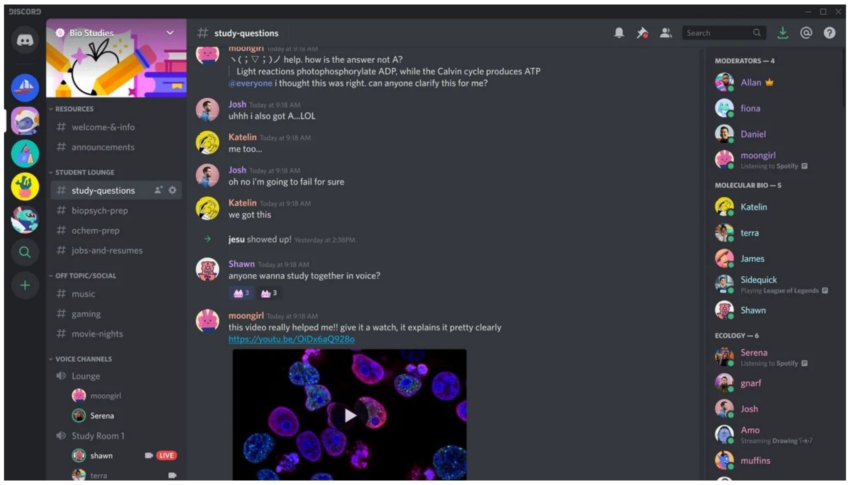
Abstract

In a never before seen pandemic (by our generations at least), remote work has become the norm where possible. Therefore, human-to-human interaction is limited, especially within organizations such as universities, businesses, and social groups. Our goal is to develop a social-networking and resource type application that helps people within affected organizations and social groups connect with each other, have access to resources that best benefit members of the organization. This application is intended to be used by both android and iOS users, using an email of their choice for authentication. The application is built using flutter for the structure and UI framework, using the Dart language, and Firebase as the primary software for database implementation and user authentication.

Keywords: Networking| Collaboration| Communication| Building Connections| Community

1.0 Introduction

1.1 Existing Systems



moongirl Today at 9:18 AM
ヽ(；▽；)ノ help. how is the answer not A?
Light reactions photophosphorylate ADP, while the Calvin cycle produces ATP
@everyone i thought this was right. can anyone clarify this for me?

Josh Today at 9:18 AM
uhhh i also got A...LOL

Katelin Today at 9:18 AM
me too...

Josh Today at 9:18 AM
oh no i'm going to fail for sure

Katelin Today at 9:18 AM
we got this

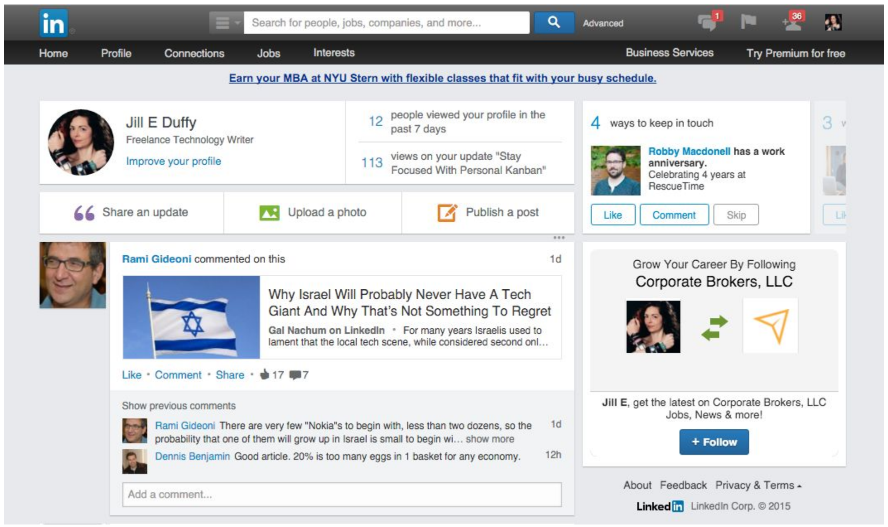
jesu showed up! Yesterday at 2:38PM

Shawn Today at 9:18 AM
anyone wanna study together in voice?
3 3

moongirl Today at 9:18 AM
this video really helped me!! give it a watch, it explains it pretty clearly
https://youtu.be/OiDx6aQ928o

Figure 1: Discord Page

Discord is a gaming-focused software/web/mobile application that lets users create their own server where they can invite other users to come and chat. Each server can be divided into separate channels each with their own chat rooms. For example, there is a public server for the game Minecraft that anyone can join, however Minecraft has many different components to it and has multiple communities that spawned from those different components that most servers are private and you would have to be invited in order to join. You cannot create your own channels unless you are an admin, so if you have an idea for a new themed channel for the server, you cannot create one yourself.



Earn your MBA at NYU Stern with flexible classes that fit with your busy schedule.

Jill E Duffy
Freelance Technology Writer
Improve your profile

12 people viewed your profile in the past 7 days

113 views on your update "Stay Focused With Personal Kanban"

4 ways to keep in touch

Robby Macdonell has a work anniversary.
Celebrating 4 years at RescueTime

Rami Gideoni commented on this 1d

Why Israel Will Probably Never Have A Tech Giant And Why That's Not Something To Regret
Gal Nachum on LinkedIn · For many years Israelis used to lament that the local tech scene, while considered second onl...

Rami Gideoni There are very few "Nokia"s to begin with, less than two dozens, so the probability that one of them will grow up in Israel is small to begin wi... show more 1d

Dennis Benjamin Good article. 20% is too many eggs in 1 basket for any economy. 12h

Grow Your Career By Following
Corporate Brokers, LLC

Jill E, get the latest on Corporate Brokers, LLC Jobs, News & more!

Figure 2: LinkedIn Page

LinkedIn is a software/web/mobile application that focuses on business networking. Users can create a profile that shows their professional experiences, skills and career interests. Then, they can add friends, companies, coworkers to your ‘network’ that acts similarly to a friends list. LinkedIn has multiple features for users to interact such as a job search/posting or creating groups for users to join. There is another feature called LinkedIn Events that lets users set up meetings, online workshops, seminars, etc. and has an event calendar that lets users keep track of enrolled events. One final feature is a recommendation list that shows other profiles that the user may find interesting enough to follow. While it has many interesting features for a general social app, LinkedIn is limited to business-oriented connections. Another issue is UX design, where many pages are a mess of links and buttons that it feels disorienting and overwhelming to use.

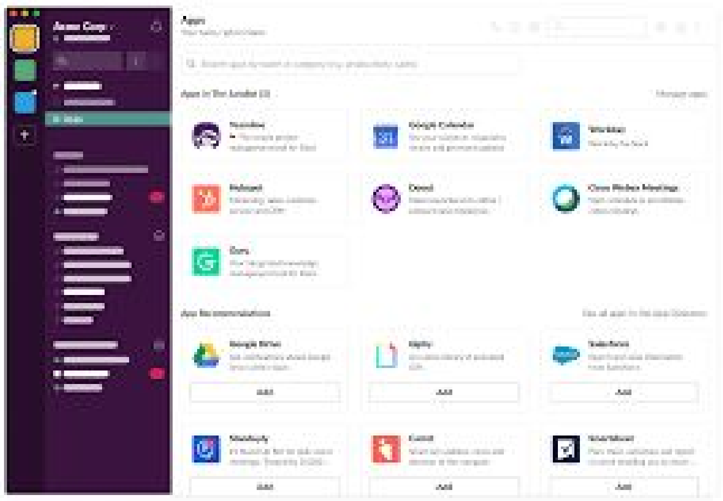


Figure 3: Slack Page

Slack is an app that is another social networking web/mobile app that has a very similar layout to Discord. This app highlights their ability to create channels for groups to communicate and create channels for different topics of conversation within the same group, or for whatever reason the user may see fit. Users are also able to do video-conferencing, make voice-calls, and share files. The app is currently being used by top companies such as TD Ameritrade, Uber, Oracle, TIME Magazine, and more familiar companies which reflects its success amongst the business industry. Slack currently advertises and has gained more traction with their focus on their app as a tool for business communication.

1.2 Proposed System

The system we're proposing is a networking and resource mobile application that is targeted to be used in universities and business organizations. We hope that it can connect members and acts as a tool to form a better sense of community within an organization and have the ability to be designed to cater the specific social and resource needs of an organization. For this project we've decided to focus on creating the app for our sample organization New York Institute of Technology. The app will consist of four focused features, a messaging feature for groups to form and connect within the networking community, a feature dedicated to posting and advertising social events on and off campus for members to attend and add to their personal calendars, and a feature dedicated to food delivery service for students to order from local restaurants, and a feature for user profile management.

Our application differentiates between leading competitors because unlike the existing apps on the market, ours is organization specific in which it is designed to represent the traits and needs of an organization. In addition to that, we intend to make finding and connecting with other members within an organization community easier by suggesting groups and contacts a user can join and have it clearly displayed for the user, compared to apps such as Discord, Slack, LinkedIn which typically require invitations to join groups, don't show existing groups that can be joined within a specific community/organization. In addition to that our app offers features such as a personal event page for the community so instead of organizational groups having to reach their community through personal invite links, attempt to spread the word about their events through multiple social media platforms where the community members are spread out amongst, groups can directly reach their audience with our app focused solely on that specific organizations member where they will have easy access to view postings. Also we offer the food delivery feature for university organizations, a tool that proves to be useful for students on campus so that they have ease of access to local restaurants and do not require having to search through multiple online food delivery services. Our app serves as a one-stop shop that makes students' lives easier.

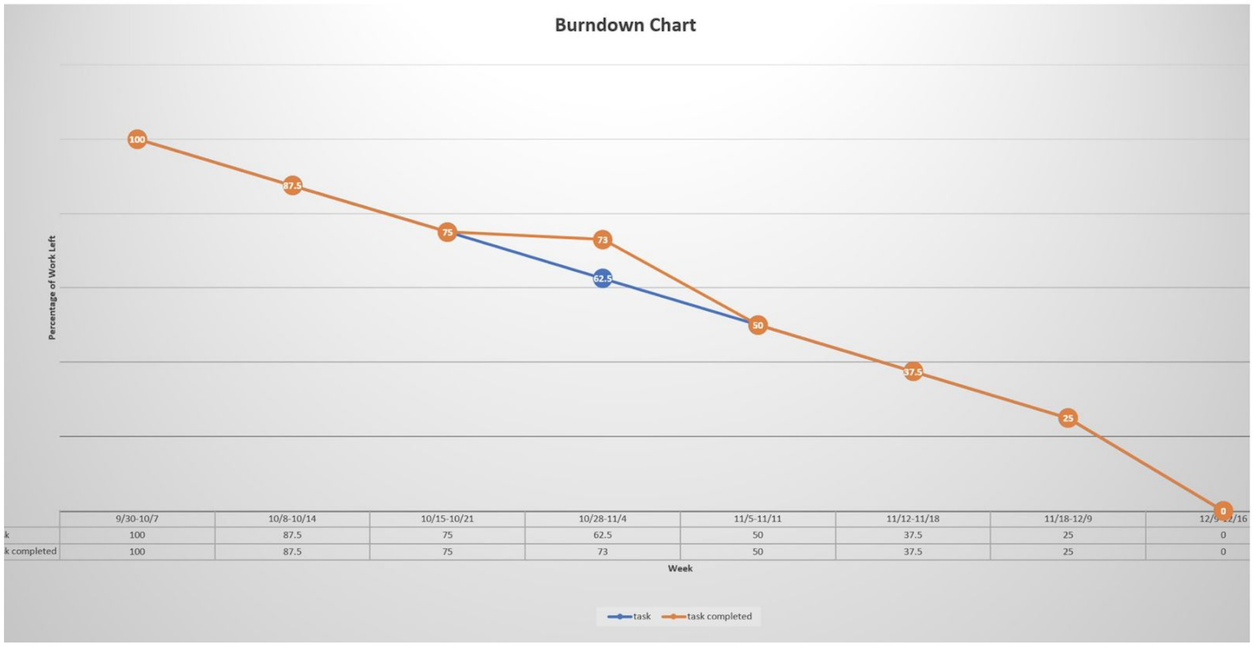
To implement the features our app offers we will be using Flutter, a UI/UX application developed by Google to create the app structure and framework. Therefore, we'll be using the Dart programming language, which was also developed by Google specifically for use with

Flutter to implement the framework and functionality of the app. The developers will be using Visual Studio Code for code editing and debugging.

The app is driven off of the connected databases using Firebase, an application also developed by Google, for user authentication and to implement the database functionality. In regards to authentication we intend for users to be able to sign up for an account to gain access to the applications tools using an email of their choice and allowing verification to be done by having an authentication email sent to the user's email account to verify their information in order to access their account.

1.3 Software Engineering Model and Scrum Process

Our team decided to follow the Incremental software development process, in order to give us the flexibility to go back and make changes throughout each phase of our software development cycle as necessary. In order to stay on track our team hosted JAD sessions to discuss weekly tasks, accomplishments,and progress made. In addition to that we used Atlassian's JIRA software to outline the tasks that needed to be completed for each feature of the project and write out "issues" (the main focuses of the project) and break the project down into sprints.



	9/30-10/7	10/8-10/14	10/15-10/21	10/28-11/4	11/5-11/11	11/12-11/18	11/18-12/9	12/9-12/16
k	100	87.5	75	62.5	50	37.5	25	0
k completed	100	87.5	75	73	50	37.5	25	0

Figure 4: Burndown Chart

1.4 Changes to Proposed System

When creating this application, there are a lot of changes we made to make it more user friendly. In the original designs there was not a navigation bar at the bottom which was later added so that users can easily switch between the different features of the application without having to go all the way back to the main menu. There were also extra pages that were created that made it more user friendly. There was an extra page that was added in the food ordering feature that takes the users payment information instead of having a tab pop up at the bottom of the ordering screen. The payment was originally at the bottom to make the application more consolidated but while developing the application we realized that it would be more user friendly to have the payment done on a different screen. The users also have the added option of possibly paying cash to the delivery person. There were also extra security features that we added that were not initially planned such as the email verification. When creating the application we realized there needed to be a way to confirm that the email that was used was valid which is why this feature was added.

1.5 Purpose

This application was developed as a tool for users within an organization or university to have easier access to organizational social information, events, in the case of a university sample have access to food delivery information and with the primary focus on encouraging regular communication between users that can be non-work/school related. We intend for users to have easier access to connecting with other users that they may or may not know initially and be able to use this application to network, develop and establish relationships that can be useful for their personal and professional success at a university or within an organization. When brainstorming ideas for this app, us as developers kept in mind our prior and current experiences as students at a university, agreeing upon the belief that a tool like this would be very useful for us to become more than just classmates that attend the same university but to build deeper connections, in which it would make it easier to form project groups, connect with others to find jobs, work on project ideas together, initiate recreational activities for each other to participate in, and more. We've come to realize that being able to engage in social activity with our peers not only makes our experience more enjoyable and meaningful at a university but also prepares us better for life moving forward, how to interact with others post-school, connects us to greater opportunities and is crucial to our personal and professional progress as individuals. Therefore we hope this app could achieve the goal of promoting more social interaction and enhancing the experience of students, co-workers, and teammates in general.

1.6 Objective and Scope

The objective of this project is to develop a mobile/web application that could be used within university and business organizations for social activity between organization members. Our focus was on the application of this app in a university setting where users would be able to communicate with other members with ease of access to discover new people, have access to social event listings within the organization, be able to post events as well and food delivery services near campus. Users must have access to either an up to date iOS or android device to use this application.

1.7 Tools and Technologies

The tools and technologies that will be used for our app development process will be the Flutter SDK, which uses the programming language Dart. Both of these tools were developed by Google to allow developers to easily create UI/UX frameworks for the applications with access to multiple plug-ins to create fully functional applications. We also used VisualStudio Code as the code editor and debugger. Firebase was used to user authentication and to create real-time databases to store information,which will be discussed further in the report. We also used Proto.io, a UI/UX design software to create the initial designs for the application.

1.8 Users

The users of this app will be universities and corporations. The app will be used by the students of the university and the employees of the corporation. Right now our sample audience is NYIT and their students but it can be extended to other universities and corporations as well. In a university setting, all types of students will be using this app. It would be appealing to all students because especially in times like this it can be very difficult to contact acquaintances from classes. It would appeal to freshmen and transfer students as they do not know a lot of people at school so it is a great way for them to connect with people as well as get to know about the events happening on campus. Also all the students would want to use the food delivery aspect of the app as everyone gets food ordered to campus. A way the employees of a company would use this app is that instead of being in groups based off the different teams they are on and the different projects they are working on. The calendar would show meetings, office events, conferences, etc. An employee would be able to use this feature the same as a university student would. A company employee would also be able to use the food delivery system the same as a university student. As of right now this app does not cater to people with disabilities (such as people with vision problems) but if time permitted it would have liked to add that feature.

1.9 Limitations

When creating this application there are many limitations and difficulties that have arised. One of the big things is that due to the current situation with the pandemic, nobody is able to see each other and everyone is in different places. Stefan, the group leader, has been in Romania for the entirety of the semester so when deciding when to meet to discuss and work on the project, time difference also has to be factored in. Another limitation is expenses. To make the application truly functional there are things that would be needed to purchase such as licenses but due to costs it is unable to be done. Another limitation that arose in the beginning is that not everybody knew each other so adapting to everyones' different work styles and strengths was an adjustment at first. Another limitation is time. There are other features we could have added but were unable to because of the time constraint.

2.0 Analysis of the Project

2.1 Context Diagram

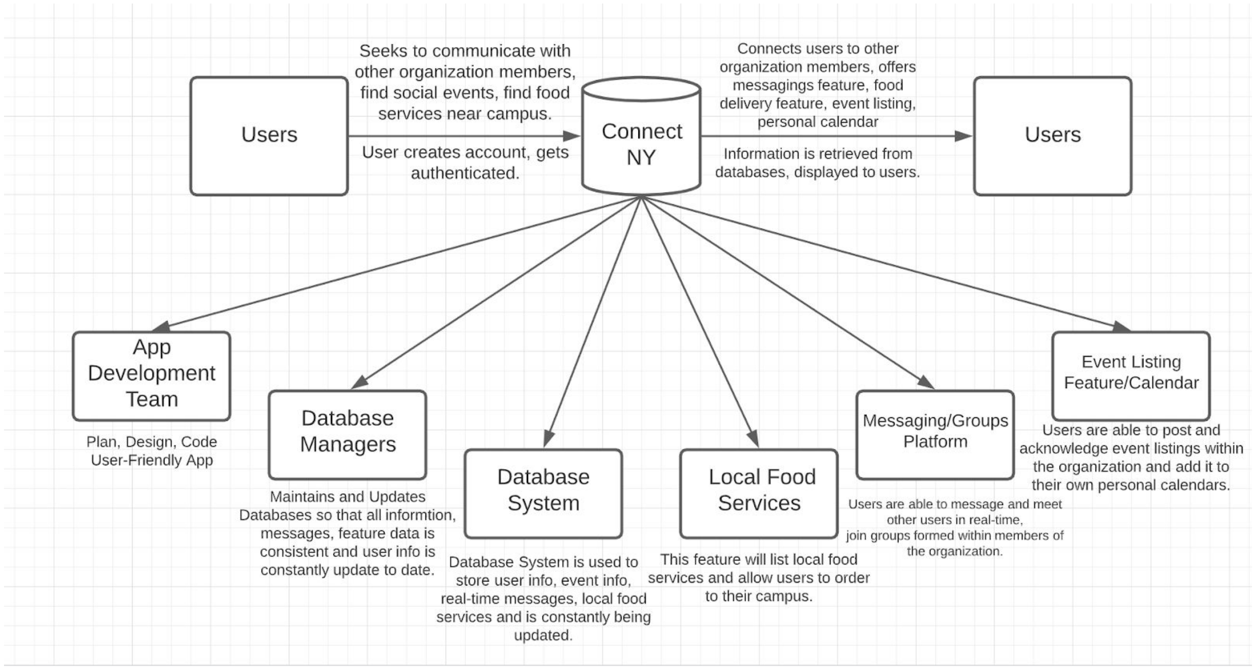

Users
Seeks to communicate with other organization members, find social events, find food services near campus.
User creates account, gets authenticated.
Connect NY
Connects users to other organization members, offers messagings feature, food delivery feature, event listing, personal calendar
Information is retrieved from databases, displayed to users.
Users
App Development Team
Plan, Design, Code User-Friendly App
Database Managers
Maintains and Updates Databases so that all informtion, messages, feature data is consistent and user info is constantly update to date.
Database System
Database System is used to store user info, event info, real-time messages, local food services and is constantly being updated.
Local Food Services
This feature will list local food services and allow users to order to their campus.
Messaging/Groups Platform
Users are able to message and meet other users in real-time, join groups formed within members of the organization.
Event Listing Feature/Calendar
Users are able to post and acknowledge event listings within the organization and add it to their own personal calendars.

Figure 5: Context Diagram

The context diagram above represents the components that are involved in the creation, organization, and functionality of our system. The main factors the application will consist of is the app development team used to plan, analyze, build, and test the application, the database managers that would be the administrators of the organizations customized app to ensure information is updated, stored and retrieved as it should, the database system which is implemented using firebase, the main features consisting of the food delivery system, the networking feature for messaging and group formation for members to use to connect to each other, and the event feature, and the profile management feature so that users can customize their profiles. The results of the application depend and begin with the user/organization, and are returned to the users within the organization.

2.2 Use Case Diagram

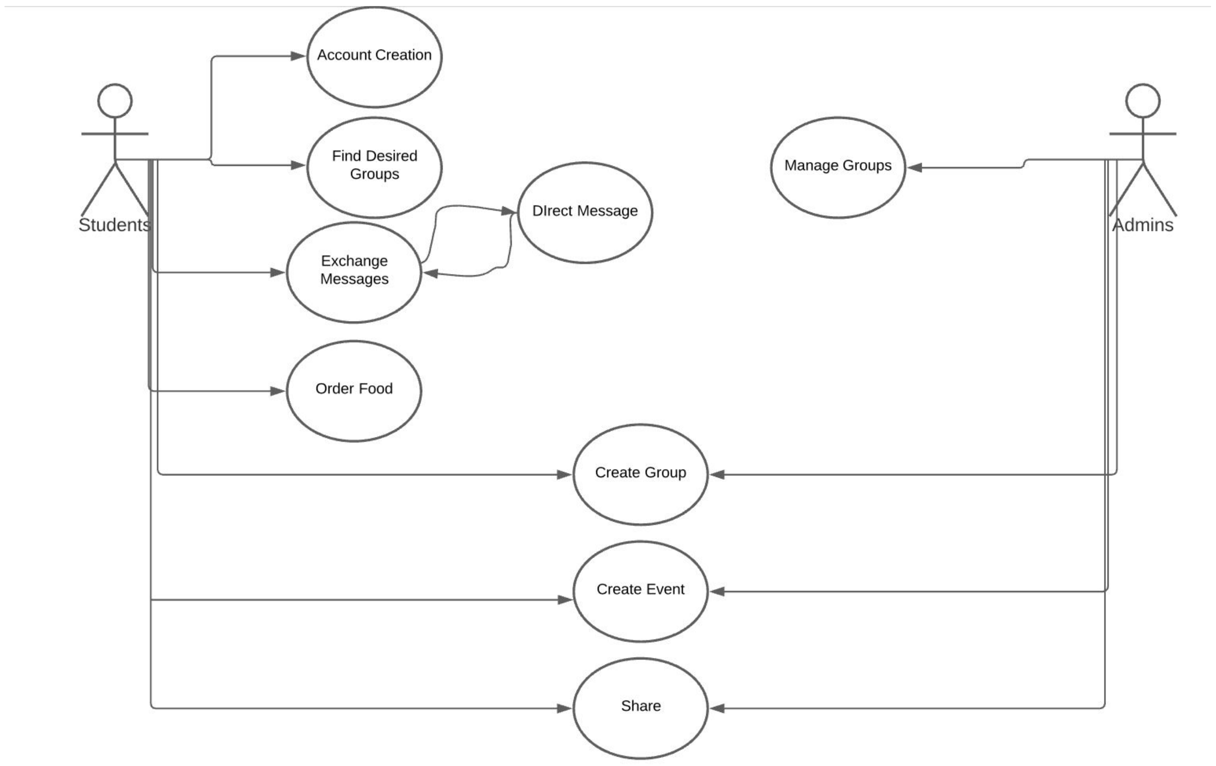


Figure 6: Use Case Diagram

According to the use case diagram above, students are able to create an account, find their desired groups, order food, and exchange messages with other users. The Admin would be able to manage and create the initial set of groups for each major, each class, and each organization (clubs). Students may create groups as well at their own discretion for any reason. The Students and Admin both also have the ability to create events and share any content they would like to.

2.3 Data-Flow-Diagram

The diagrams below were created to represent the flow of data within the app structure. The medium sized rectangles represent Entity's and Outputs. The smaller rectangles represent the databases used to connect and store information between each feature of the app. The cylinders represent main processes and sub-processes that occur within our application.

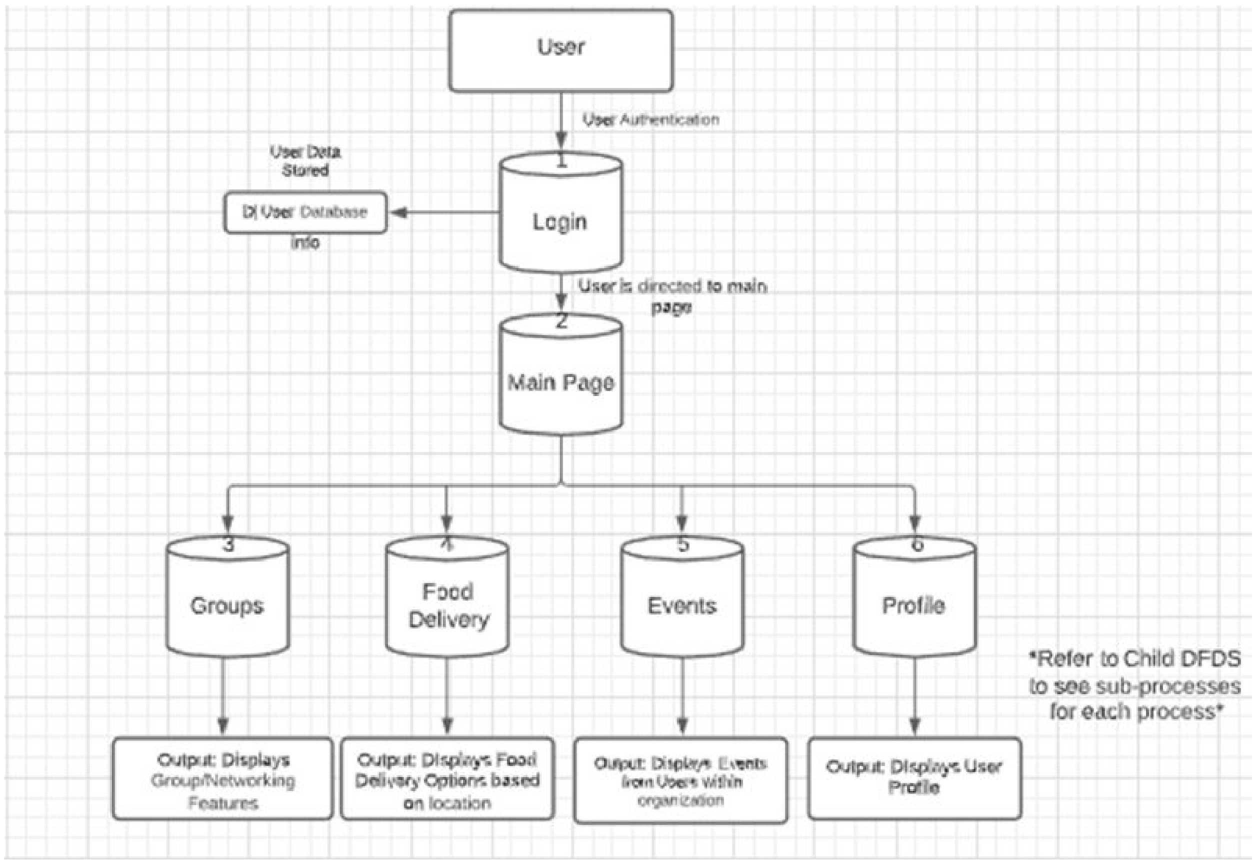

User
User Authentication
1
Login
User Data Stored
D| User Database info
User is directed to main page
2
Main Page
3
Groups
4
Food Delivery
5
Events
6
Profile
Refer to Child DFDS to see sub-processes for each process
Output: Displays Group/Networking Features
Output: Displays Food Delivery Options based on location
Output: Displays Events from Users within organization
Output: Displays User Profile

Figure 7: Main User DFD

The diagram above represents the main structure of the application. It begins with the main entity, the Administrators who have access and control over the application and features, databases..etc. Following the Admin, we have our second entity which would be the User's of the app. The admin's have administrative access to the user database to manage users on the platform, while the User's are also connected to the User database since their login credentials and profile information will be stored there. User authentication occurs when the user entity enters their credentials which are then verified which is included in process #1: Login. After a user logins they are directed to the second process, entering the main page which will display the following four main processes and features that are included within the application structure. The 3rd process, "Groups" represents our messaging/networking feature which will store messaging data exchanged between users and their messages will be displayed on the main screen as an output. The 4th process, "Food Delivery", will store information of local restaurants and food delivery options a user can choose and order from. The output will be a display of all the local food vendors on its' own individual page.. The 5th process, "Events", represents the feature where users will have all on/off campus events displayed on it's own individual page. The 6th process, "Profile", represents the feature where users can manage their profile data, this will also be outputted and displayed on its own individual page. Each of the main four sub-processes below the main page have their own sub-processes which are broken down into child DFD's below

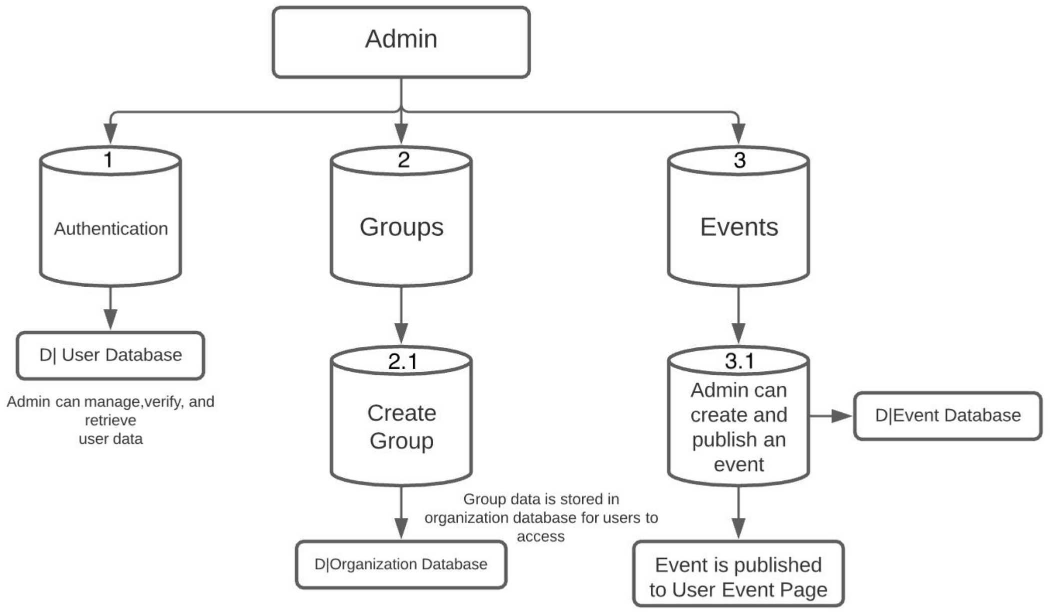

Admin
1
Authentication
2
Groups
3
Events
D| User Database
Admin can manage,verify, and retrieve user data
2.1
Create Group
3.1
Admin can create and publish an event
D|Event Database
Group data is stored in organization database for users to access
D|Organization Database
Event is published to User Event Page

Figure 8: Main Admin DFD

We also created a main DFD from the Admin's perspective since their privileges are different from the users. Here admins are able to verify and authenticate users and have access and control of the data stored in the user database. They are also able to control the organization database that stores information about the groups created, and create their own groups as well. Admin's can also manage and create their own events which would be published on the user screen. User's would be able to access both the groups and events the admin's create and the admin's would theoretically be able to view them as well from their Admin account when that feature is created.

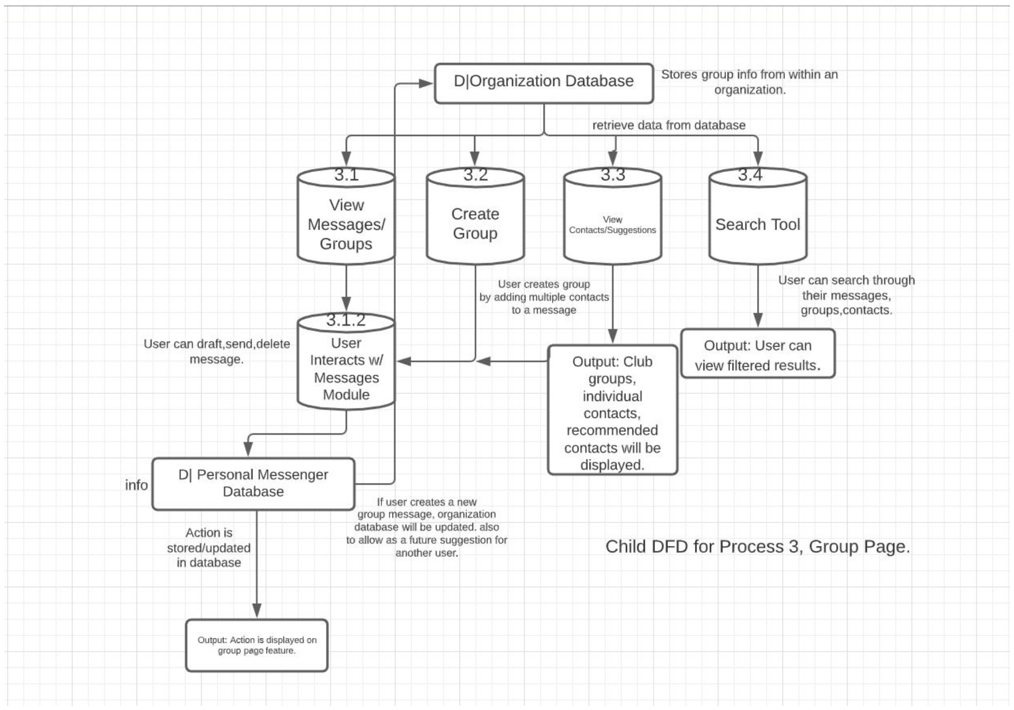

D|Organization Database
Stores group info from within an organization.
retrieve data from database
3.1
View Messages/ Groups
3.2
Create Group
3.3
View Contacts/Suggestions
3.4
Search Tool
User creates group by adding multiple contacts to a message
User can search through their messages, groups,contacts.
3.1.2
User Interacts w/ Messages Module
User can draft,send,delete message.
Output: Club groups, individual contacts, recommended contacts will be displayed.
Output: User can view filtered results.
info
D| Personal Messenger Database
If user creates a new group message, organization database will be updated. also to allow as a future suggestion for another user.
Action is stored/updated in database
Child DFD for Process 3, Group Page.
Output: Action is displayed on group page feature.

Figure 9: Groups Child DFD

The child DFD above represents the flow of data within process #3, the "Groups/Main" page. The data within this process is stored into the Organization database. In this process User's are able to view messages exchanged between individual contacts and groups. Their personal messages are connected to a personal messenger database which stores their messaging data so that their messages can be retrieved when needed. Hence, their messages are displayed on the main page and on their individual conversations page. User's are able to initiate the sub-process which allows them to create group conversations. After a group is formed the data goes back to the Messaging Module Process/Feature, which allows them to view that group conversation on the messaging feature/main page. User's are also able to view their contacts and contact suggestions which outputs a list of members and groups formed within the organization. This information is retrieved from the organization database. Users' are also able to search and filter their conversations using the search sub-process. This process yields a filtered list of members/groups a user can join.

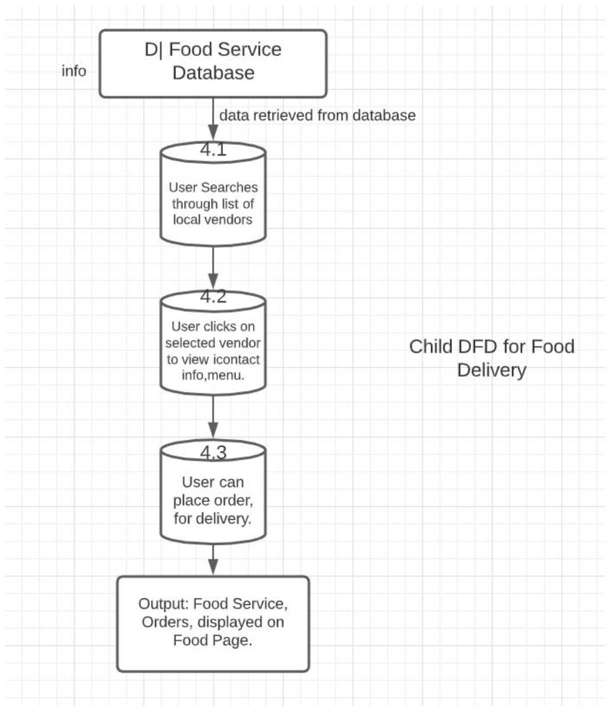

D| Food Service Database
info
data retrieved from database
4.1
User Searches through list of local vendors
4.2
User clicks on selected vendor to view icontact info,menu.
4.3
User can place order, for delivery.
Output: Food Service, Orders, displayed on Food Page.
Child DFD for Food Delivery

Figure 10: Food Delivery Child DFD

The diagram above is the child DFD for the 4th process labeled, “Food Delivery", in which this feature will connect to a database storing information of local food services that students can order food from. Users will be able to search for a local vendor and then click on a vendor that will display the vendors menu items in which they can then select items, enter their payment information, and place their order. All of the outputs for this process will be displayed on the food delivery page and subpages connected to the process.

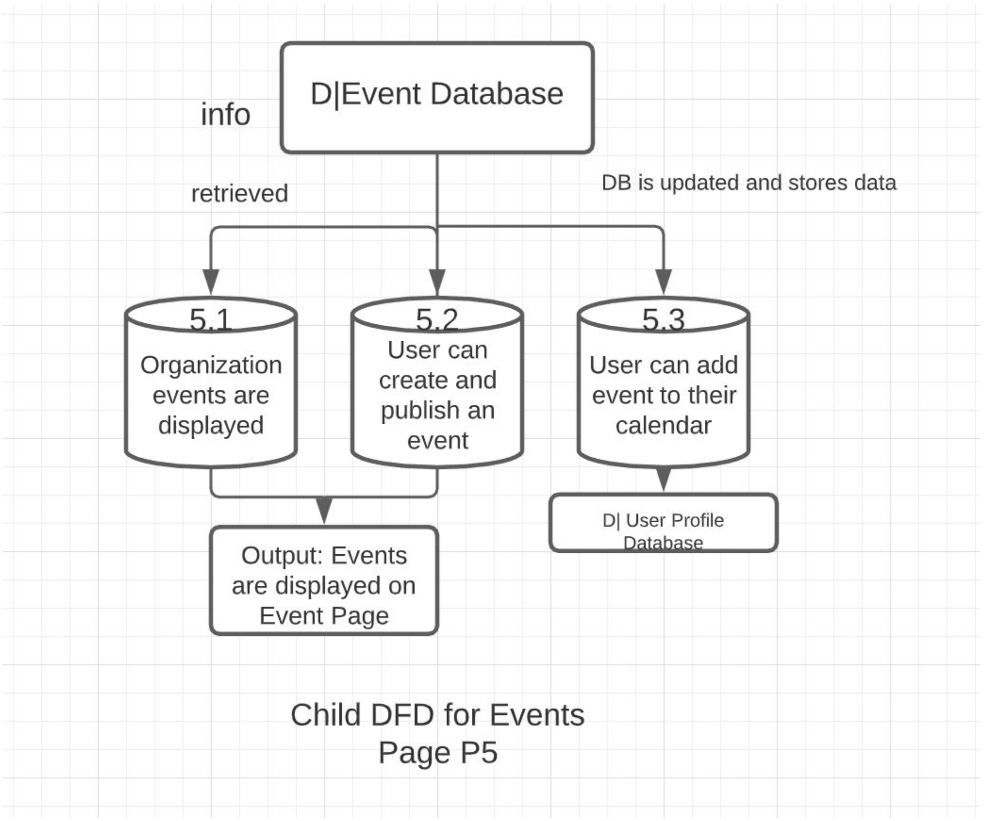

D|Event Database
info
retrieved
DB is updated and stores data
5.1
Organization events are displayed
5.2
User can create and publish an event
5.3
User can add event to their calendar
D| User Profile Database
Output: Events are displayed on Event Page
Child DFD for Events Page P5

Figure 11: Events Child DFD

This child data-flow-diagram represents the sub-processes for the Events feature. Information related to events are stored in the event database in which information from events posted by organization members, event listings can be stored and retrieved, thus outputting it's information onto the main event page. Also since users are able to add an event they are interested in going from the event page to their calendar, their information/interested events can be stored in the user profile database for their own personal access.

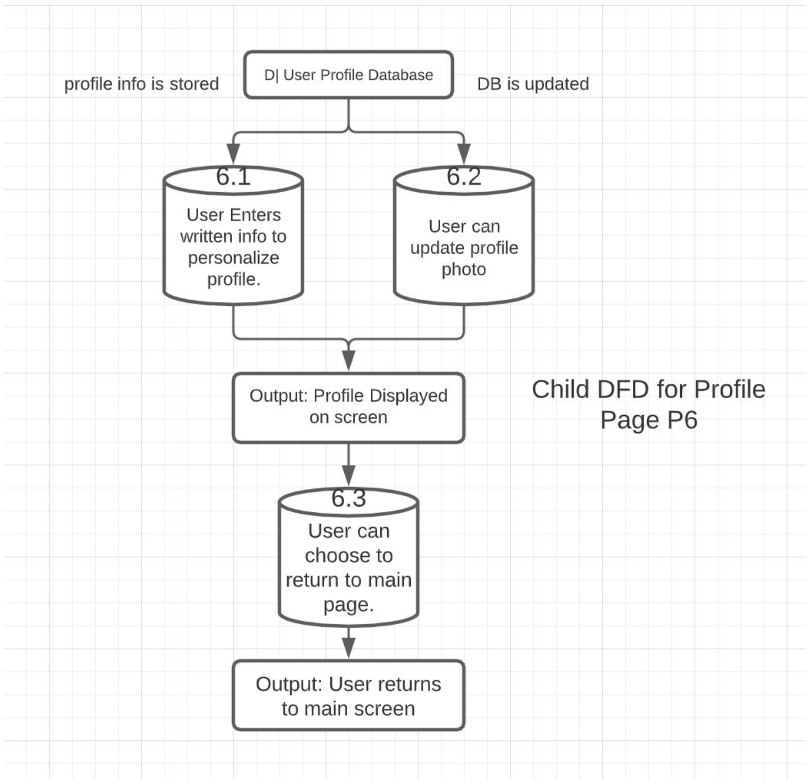

D| User Profile Database
profile info is stored
DB is updated
6.1
User Enters written info to personalize profile.
6.2
User can update profile photo
Output: Profile Displayed on screen
Child DFD for Profile Page P6
6.3
User can choose to return to main page.
Output: User returns to main screen

Figure 12: Profile Page Child DFD

Lastly, we have the child DFD for the Profile page. Here user data is stored and retrieved from the user profile database. Users are able to enter written text and upload a photo to personalize their profile page, which will then be uploaded and displayed on the profile page. Users are then able to return to the main page.

3.0 Explanation of the System

This section provides details of the database, server, and iOS application functionality and features.

3.1 Database Design

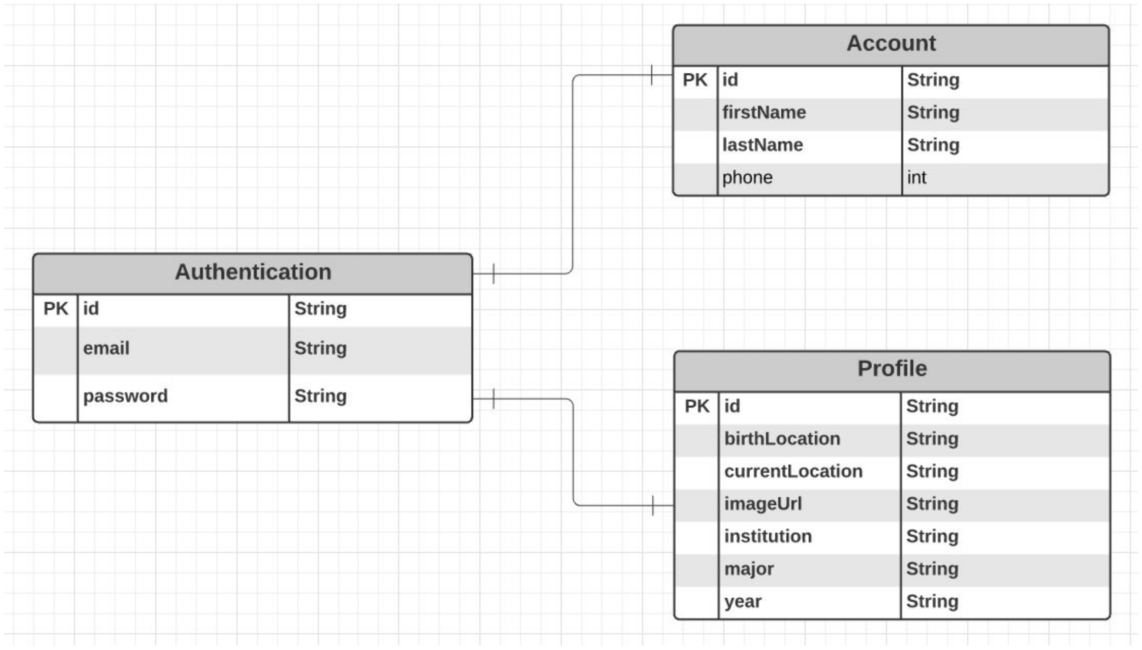


Figure 13: Database Design

The database was designed using firebase in order to authenticate the accounts and profile. The authentication has a one to one relationship with both account and profile to make sure each profile is unique to each person.

3.2 Database Schema

Below are all the tables used for the database and the relationships between them:

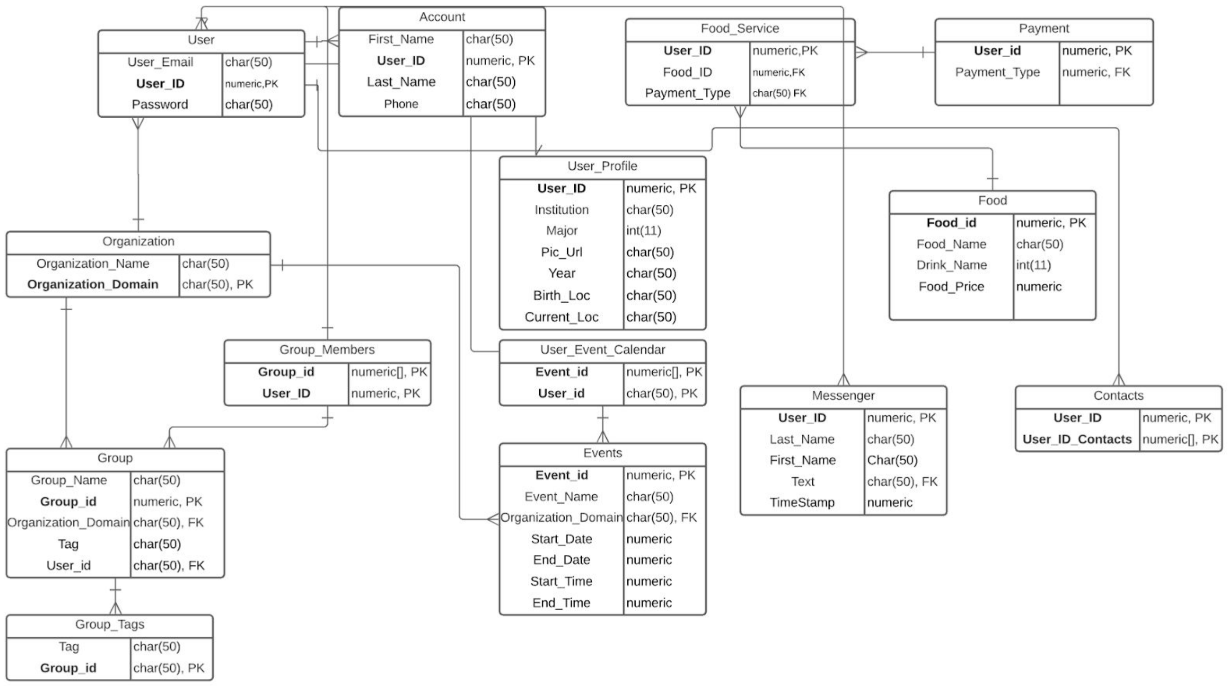
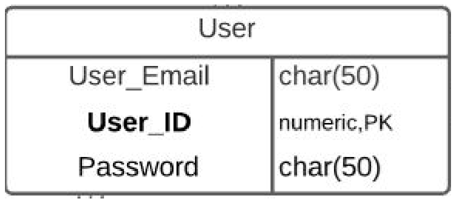


Figure 14: Database Schema

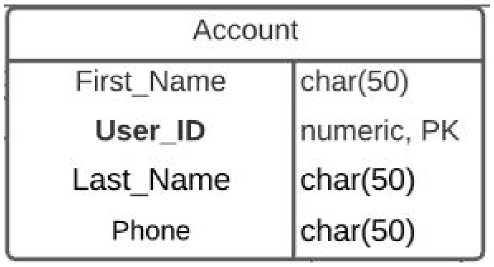
Below are all the individual tables along with their function:



User	
User_Email	char(50)
User_ID	numeric,PK
Password	char(50)

Figure 15: User Data Table

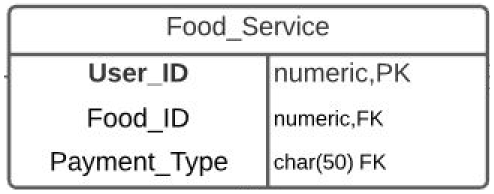
The "User" table contains all the important login information about each user. The primary key is the "User_ID" as that is how each user will be differentiated. Some information about users may overlap but each user has their own unique user ID making it the primary key for this table.



Account	
First_Name	char(50)
User_ID	numeric, PK
Last_Name	char(50)
Phone	char(50)

Figure 16: Account Data Table

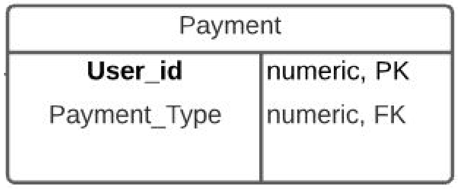
The “Account” table links the “User_ID” to the user’s name and phone number for their account information making “User_ID” the primary key.



Food_Service	
User_ID	numeric,PK
Food_ID	numeric,FK
Payment_Type	char(50) FK

Figure 17: Food Service Data Table

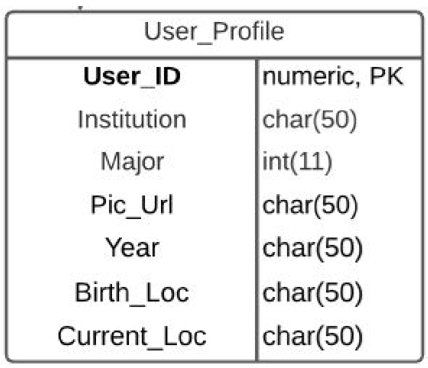
The “Food_Service” table connects the payment information and the food order to the user that ordered the food, making “User_ID” the primary key and making “Food_ID” and “Payment_Type” both foreign keys.



Payment	
User_id	numeric, PK
Payment_Type	numeric, FK

Figure 18: Payment Data Table

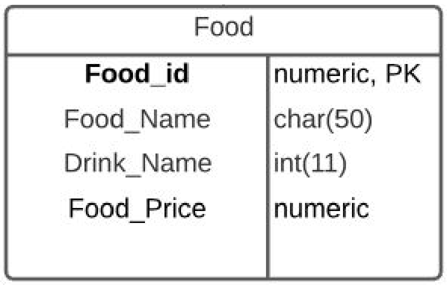
The “Payment” table links the payment to the unique user making “User_ID” the primary key and “Payment_Type” the foreign key.



User_Profile	
User_ID	numeric, PK
Institution	char(50)
Major	int(11)
Pic_Url	char(50)
Year	char(50)
Birth_Loc	char(50)
Current_Loc	char(50)

Figure 19: User Profile Data Table

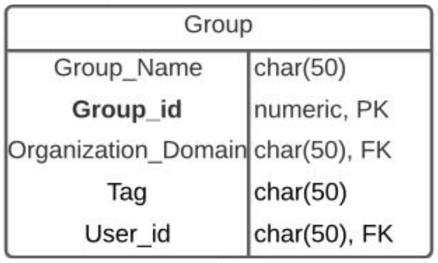
The “User_Profile” table consists of all the information that is linked to the User Profile page. The page will display the user’s institution, major, which year they are in, where they are from, where they currently reside, and a picture of them, which is linked to each user’s unique user ID. This makes “User_ID” the primary key that links all the users' information together.



Food	
Food_id	numeric, PK
Food_Name	char(50)
Drink_Name	int(11)
Food_Price	numeric

Figure 20: Food Data Table

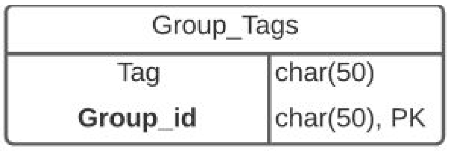
The “Food” table consists of all the information about the food order. Each food order is defined and uniquely identified by the “Food_id” which is the primary key. Under this all the details of the food order is stored.



Group	
Group_Name	char(50)
Group_id	numeric, PK
Organization_Domain	char(50), FK
Tag	char(50)
User_id	char(50), FK

Figure 21: Group Data Table

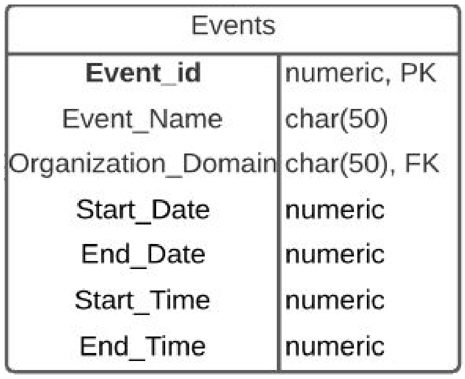
The “Group” table contains all the information about each individual group. The primary key is the “Group_id” because each group will have its own unique ID. Each group will then be associated with an organization and users making “Organizaton_Domain” and “User_ID” foreign keys.



Group_Tags	
Tag	char(50)
Group_id	char(50), PK

Figure 22: Group Tags Data Table

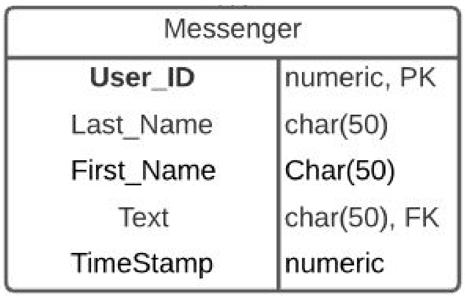
Each group will also have group tags. The “Group_Tags” table connects the tags to the primary key, “Group_ID”.



Events	
Event_id	numeric, PK
Event_Name	char(50)
Organization_Domain	char(50), FK
Start_Date	numeric
End_Date	numeric
Start_Time	numeric
End_Time	numeric

Figure 23: Events Data Table

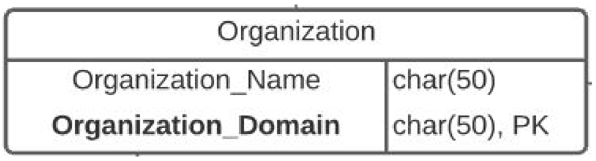
The "Events" table stores the information about each event that is created. The primary key is "Event_ID" as every event will have its own unique ID. Each event will be associated with an organization which is why "Organization_Domain" is the foreign key.



Messenger	
User_ID	numeric, PK
Last_Name	char(50)
First_Name	Char(50)
Text	char(50), FK
TimeStamp	numeric

Figure 24: Messenger Data Table

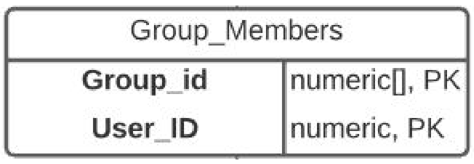
The "Messenger" table stores the conversations in group messaging. The primary key is "User_ID" of the user sending and receiving the message and the foreign key is the "Text" that is sent in the group.



Organization	
Organization_Name	char(50)
Organization_Domain	char(50), PK

Figure 25: Organization Data Table

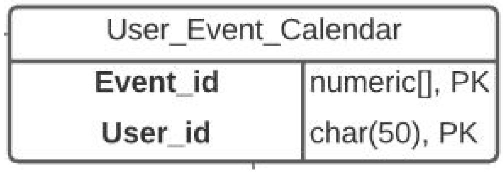
The "Organization" table links the "Organization_Name" to the primary key, "Organization_Domain".



Group_Members	
Group_id	numeric[], PK
User_ID	numeric, PK

Figure 26: Group Members Data Table

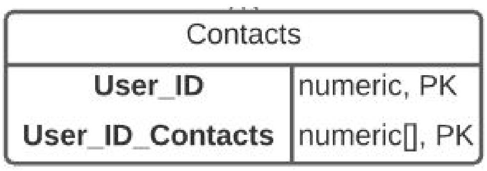
The “Group_Members” table links the group to the user which is why this table has two primart keys, “Group_ID” and “User_ID”.



User_Event_Calendar	
Event_id	numeric[], PK
User_id	char(50), PK

Figure 27: User Event Data Table

The “User_Event_Calendar” table links the events to the user which is why this table has two primart keys, “Event_ID” and “User_ID”.



Contacts	
User_ID	numeric, PK
User_ID_Contacts	numeric[], PK

Figure 28: Contacts Data Table

The “Contacts” table links contacts to the user which is why this table has two primary keys, “User_ID_Contacts” and “User_ID”.

3.3 Entity Relationship Diagram

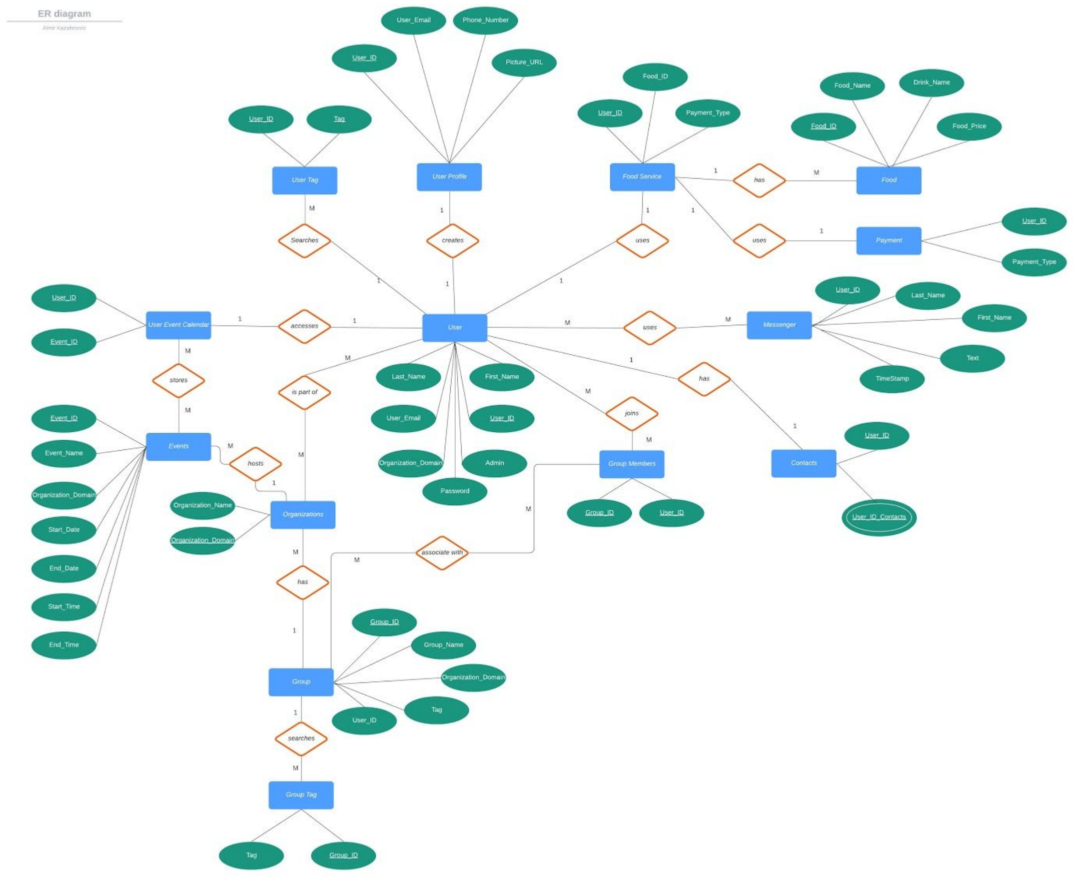


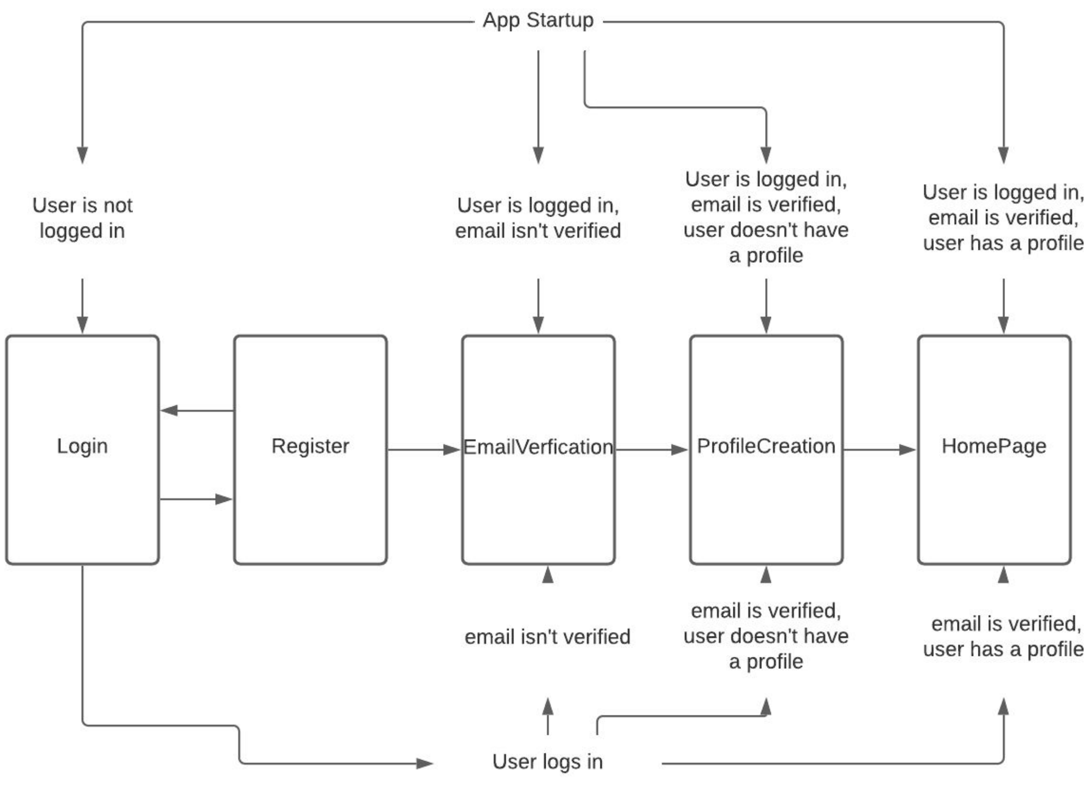
Figure 29: Entity Relationship Diagram

The main entity of this project is the “User”. The user creates a “user profile”, the user searches for multiple “user tags”, and the user also has “contacts”. Multiple users can also use “Messenger” where they would send and receive multiple messages. Users also join multiple “Group Members” that can be associated with multiple “Groups”. A group can be searched using multiple “Group Tags”. Many “Organizations” that the users are a part of each also have their own “Group”. An organization can also host many “Events” which get stored on multiple “User Event Calendars”, and each user has access to their individual user event calendar. The user also uses “Food Service” which has a variety of “Food” options and also uses “Payment”.

3.4 Functionality and Implementation

3.4.1 App StartUp and Login Navigation

When the user opens the app, there are different pages that the user can start in based on their account status. Below is a diagram that shows how the app determines what page to load when starting the app as well as the page navigation as the user is creating their account.



App Startup
User is not logged in
User is logged in, email isn't verified
User is logged in, email is verified, user doesn't have a profile
User is logged in, email is verified, user has a profile
Login
Register
EmailVerfication
ProfileCreation
HomePage
email isn't verified
email is verified, user doesn't have a profile
email is verified, user has a profile
User logs in

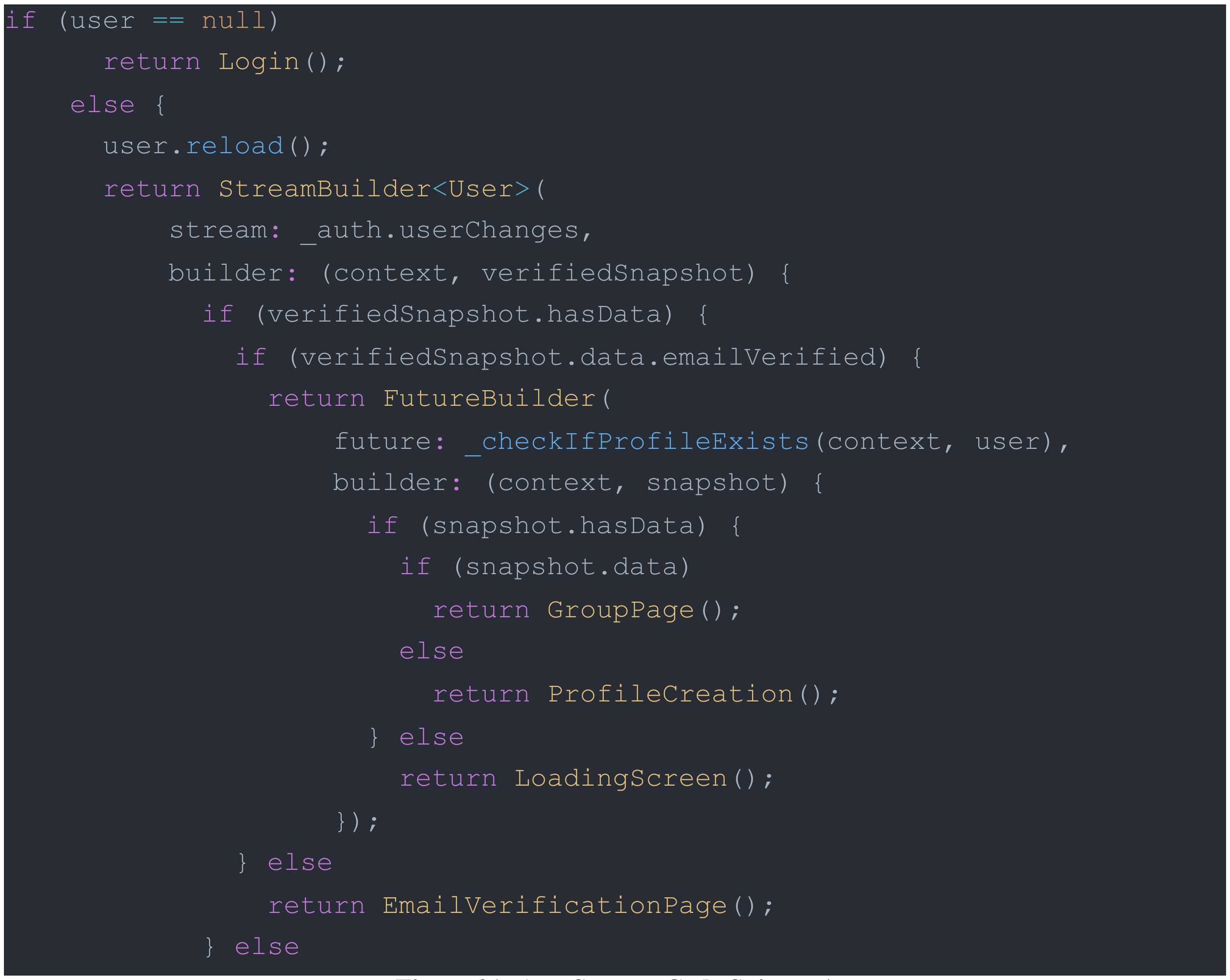
Figure 30: App Startup and Login Page Diagram

If the user opens the app for the first time, they will be taken to the Login page because they do not have an account yet. If the user logged out in a previous session, then they will also be taken to the Login page. The user's logged in status is stored locally so the app knows to automatically sign the user in if they closed and reopened the app without logging out.

If the user logs in, but hasn't verified their email, they will be taken to the EmailVerification page where the page will tell them to verify before continuing account creation.

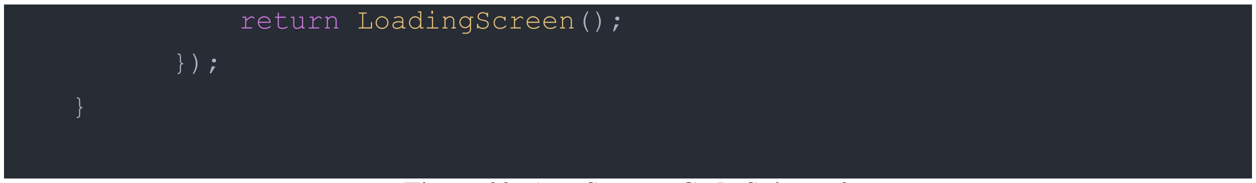
If the user is logged in, their email is verified, but they haven't created their profile yet, then they will be taken to the ProfileCreation page. Here the user has to create a profile before continuing.

If the user satisfies all above criteria (logged in previously, email verified, profile created, then the user will be taken to the homepage of the app, the Groups page, where they can now use the app. Another path that can be taken is if the user is signed out, but they verified their email and created a profile, then they will be taken to the Login page, however when they log in, they will be taken straight to the Groups page.



```
if (user == null)
      return Login();
    else {
      user.reload();
      return StreamBuilder<User>(
          stream: _auth.userChanges,
          builder: (context, verifiedSnapshot) {
            if (verifiedSnapshot.hasData) {
              if (verifiedSnapshot.data.emailVerified) {
                return FutureBuilder(
                    future: _checkIfProfileExists(context, user),
                    builder: (context, snapshot) {
                      if (snapshot.hasData) {
                        if (snapshot.data)
                          return GroupPage();
                        else
                          return ProfileCreation();
                      } else
                        return LoadingScreen();
                    });
              } else
                return EmailVerificationPage();
            } else
```

Figure 31: App Startup Code Snippet 1



```
              return LoadingScreen();
          });
    }
```

Figure 32: App Startup Code Snippet 2

Above is the code used to control where the user is taken on startup. When the app runs, the main method calls the widget, AuthProvider where the above code is it's build implementation.

The first thing AuthProvider checks is if a User was logged in in a previous session. The user variable contains data for a user that logged into the app, if the user variable has no data, then the user is not logged in, so they will be taken to the Login page.

The StreamBuilder listens in on a Stream to check the information of the user's account, including email verification. The emailVerified method returns a boolean on whether the user's email is verified or not.

The _checkIfProfileExists is a Future that returns a boolean on whether the user has an entry in the Profile table of Firebase. This method depends on another method from the ProfileService called checkIfProfileExists. This method is explained below in Section 4.4.1.3 DatabaseService.

3.4.2 AuthService (Firebase Authentication)

AuthService is a class that lets us access the Firebase Authentication database of the app. With this class we can add new accounts, log the user in and out, and get information on the user's account.

```
Stream<User> get user {
    return _auth.authStateChanges();
  }

  Stream<User> get userChanges {
    return _auth.userChanges();

  }
```

Figure 33: AuthService Code Snippet 1

The above two Streams lets us check the current status of the user's account. The 'user' steam keeps track of the user's sign in status (whether the user is signed in or out). The userChanges stream keeps track of user's account data and information. We use the user stream for app startup navigation and the userChanges stream is for checking email verification.

```
Future login(String email, String password) async {
    try {
      UserCredential userCredential = await FirebaseAuth.instance
          .signInWithEmailAndPassword(email: email, password: password);
      return userCredential;
    } on FirebaseAuthException catch (e) {
      if (e.code == 'user-not-found') {
        print('No user found for that email.');
      } else if (e.code == 'wrong-password') {
        print('Wrong password provided for that user.');
      }
    }
```

Figure 34: AuthService Code Snippet 2

The login method is used for logging the user into their account based on an email and password that the user provides. We call the method signInWithEmailAndPassword to connect with Firebase using the email and password arguments. If Firebase does not find the user's email in the system or their password is wrong, then this method will return an error. Otherwise, the user will be logged into Firebase.

```
Future register(String email, String password, String firstName,
      String lastName, int phone) async {
    try {
      UserCredential userCredential = await FirebaseAuth.instance
          .createUserWithEmailAndPassword(email: email, password:
password);

      User user = userCredential.user;

      DatabaseService(uid: user.uid)
          .updateAccountData(firstName, lastName, phone);
      //DatabaseService(uid: user.uid).updateProfileData("", "", "", "",
"", "");

      await user.sendEmailVerification();

      return userCredential;
    } on FirebaseAuthException catch (e) {
      if (e.code == 'weak-password') {
        print('The password provided is too weak.');
      } else if (e.code == 'email-already-in-use') {
        print('The account already exists for that email.');
      }
    } catch (e) {
      print(e);
    }
```
Figure 35: AuthService Code Snippet 3

The register method is used to register a new account into Firebase, create a new entry in the Account table of the database and send a verification email to the provided email. The email and password arguments are used to create a new account while the firstName, lastName, and phone arguments are stored in the Account table. The method, createUserWithEmailAndPassword creates a new account that uses an email and password system to log in. The updateAccountData method from DatabaseService creates the new Account entry using the remaining arguments. Finally, the sendEmailVerification method sends the user the verification email. If Firebase has any errors creating a new account then the register method will return an error and none of the other operations listed above will run.

```
 Future signOut() async {
    try {
      return await _auth.signOut();
    } catch (e) {
      print(e.toString());
      return null;
    }
  }
```
Figure 36: AuthService Code Snippet 4

The signOut method logs the user out of their account. It doesn't do anything else besides that, but just to have it as a convenient reference in the same class as the other two methods above.

```
String getEmail() {
    String email;
    email = _auth.currentUser.email;
    return email;
  }
```
Figure 37: AuthService Code Snippet 5

The getEmail method is used to get the email of the currently logged in user for display purposes.

3.4.3 DatabaseService (Cloud Firestore)

The DatabaseService class lets us access the Cloud Firestore database in order to add or get information from it. There are two tables in Cloud Firestore, Account and Profile (Section 3.2).

```
  final CollectionReference accountCollection =
      FirebaseFirestore.instance.collection('Account');
  //Profile
  final CollectionReference profileCollection =
      FirebaseFirestore.instance.collection('Profile');
```

Figure 38: Database Service Code Snippet 1

The above CollectionReferences are objects used in other FirebaseService methods to reference the Account and Profile tables respectively. We use these to add new entries into the tables or get data from the tables.

```
 //Account Table Update
  Future updateAccountData(String firstName, String lastName, int phone)
async {
    return await accountCollection.doc(uid).set({
      'firstName': firstName,
      'lastName': lastName,
      'phone': phone,
    });
  }

  //Profile Table Update
  Future updateProfileData(String imageUrl, String institution, String
major,
      String year, String birthLoc, String currentLoc) async {
    return await profileCollection.doc(uid).set({
      'imageUrl': imageUrl,
      'institution': institution,
      'major': major,
      'year': year,
      'birthLocation': birthLoc,
      'currentLocation': currentLoc,
    });
  }
```

Figure 39: Database Service Code Snippet 2

The updateAccountData and updateProfileData methods are used to update document entries in their respective tables. If the document does not exist then the methods will create a new entry instead.

```
  //Account
  Stream<Account> get account {
    return
accountCollection.doc(uid).snapshots().map(_accountListFromSnapshot);
  }

  //Profile
  Stream<Profile> get profile {
    return
profileCollection.doc(uid).snapshots().map(_profileListFromSnapshot);
  }
```

Figure 40: Database Service Code Snippet 3

These two streams are used to get data from the user's Account and Profile entries. The user's user id is the common key among these tables and is used to identify which document belongs to them.

```
  Future<bool> checkIfProfileExists() async {
    bool exists;
    try {
      await profileCollection.doc(uid).get().then((doc) {
        if (doc.exists)
          exists = true;
        else
          exists = false;
      });
      return exists;
    } catch (e) {
      return false;
    }
  }
```

Figure 41: Database Service Code Snippet 4

The checkIfProfileExists method returns a boolean value on whether the given user id has a document in the Profile table. This is used to check if the user created their profile yet.

3.4.4 Login Page

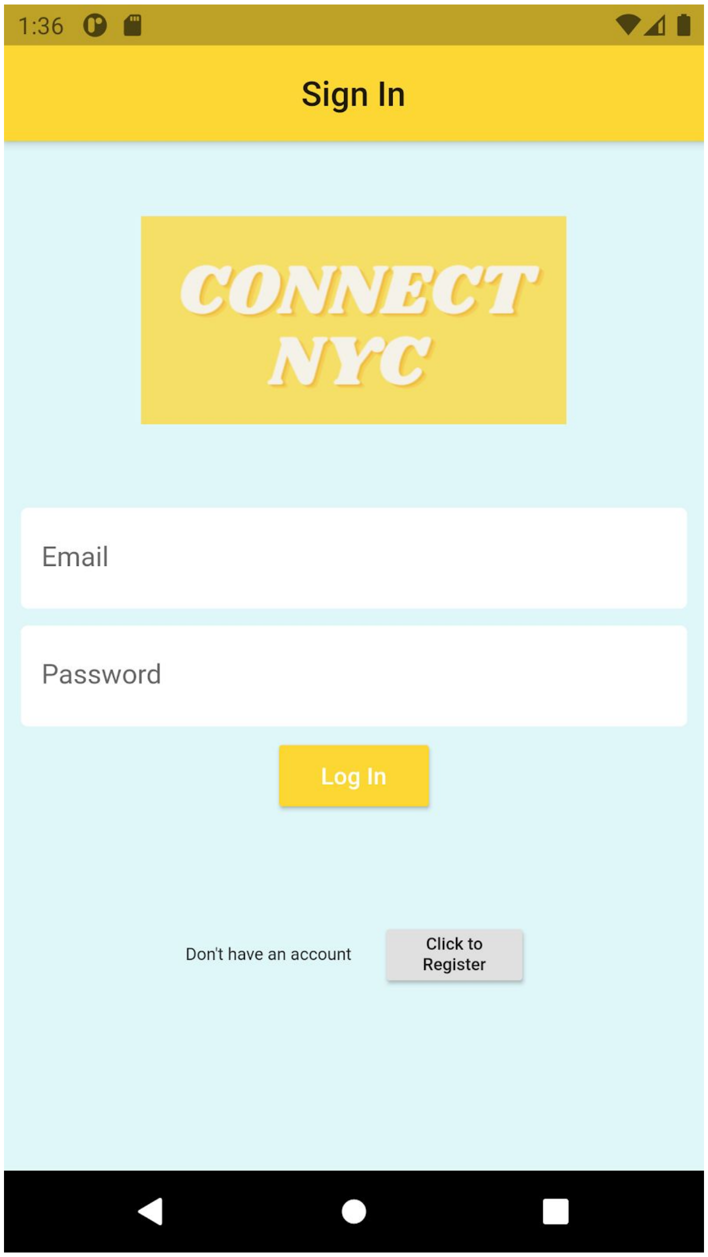


Figure 42: Login Page

When opening the application for the first time, the user is taken to the login screen shown above. The page features the app logo, text inputs for the user's email and password, a login button and a button that takes the user to the Register page.

If the user has an account already, they can input their username and password into the email and password text fields respectively and click the 'Login' button. This will take them to the Groups page. If the user does not have an account, they will click on the "Click to Register" button and this will redirect them to the Register page. If the user has already logged into the app in a previous session of using the app and hasn't signed out, then the app will automatically redirect the user to the Groups page.

The Login page is made up of the main Login page widget that sets the Scaffold for the page as shown below and the LoginForm widget that controls the functionality of the page.

```
return Scaffold(
      //backgroundColor: Colors.cyan[50],
      appBar: AppBar(
        // backgroundColor: Colors.yellow[600],
        title: Text('Sign In'),
        centerTitle: true,
      ),
      body: SingleChildScrollView(
        child: Column(
          children: [
            Container(
              margin: EdgeInsets.all(5),
              child: Center(child:
Image.asset('lib/assets/connect_logo.png')),
              width: 250,
              height: 200,
            ),
            LoginForm(),
```

Figure 43: Login Page Code Snippet 1

The Scaffold includes the app bar at the top of the page that displays 'Sign In', the app logo, and the Login form. The logo and login form are contained in a Column widget so that both can be displayed vertically.

When clicking on the email or password TextFormField, the keyboard that pops up will cover any widgets towards the bottom of the screen, causing an pixel overflow error. If the

TextFormField the user is trying to write in is towards the bottom of the screen, then the keyboard will cover the TextFormField and the user won't be able to see what they are typing in until they close the keyboard. The SingleChildScrollView widget prevents this issue by scrolling the screen up when the keyboard is opened so that the TextFormField being used is visible to the user while typing.

```
      loginPadding(TextFormField(
                     decoration: loginDecoration("Email"),
                     validator: (value) => value.isEmpty ? 'Enter email' :
null,
                     onChanged: (value) => email = value,
                   )),
                   loginPadding(TextFormField(
                     decoration: loginDecoration("Password"),
                     validator: (value) => value.isEmpty ? 'Enter password'
: null,
                     onChanged: (value) => password = value,
                     obscureText: true,
                   )),
```

Figure 44: Login Page Code Snippet 2

The LoginForm contains the TextFormFields and the buttons on the Login page as well as their functionality. The email and password TextFormFields, as shown above use, validators that check whether the fields are empty. If the fields are empty, then a message pops up under them stating to input something into the fields.

```
      RaisedButton(
                     onPressed: () async {
                       if (_formKey.currentState.validate()) {
                         dynamic result = await _auth.login(email,
password);
                         if (result == null) {
                           setState(() {
                             error = 'Email/password is incorrect';
                           });
                         } else {
                           //Navigator.pop(context);
                         }
                       }
```

```
                         },
```
Figure 45: Login Page Code Snippet 3

The above code controls the login button. When the user clicks on this button, the form key checks the email and password fields to check if there are any errors with validation (if they are empty). If there are issues, then the validator messages will pop up, but if validations are accepted then Firebase will attempt to log in the user through the login method with the inputs of the email and password TextFormFields as arguments. If the email is not in the database or the password is incorrect, then the login will fail. A message under the login button will display "Email/password is incorrect." if this happens. If login is successful, then the user is taken to the homepage.

3.4.5 Register Page

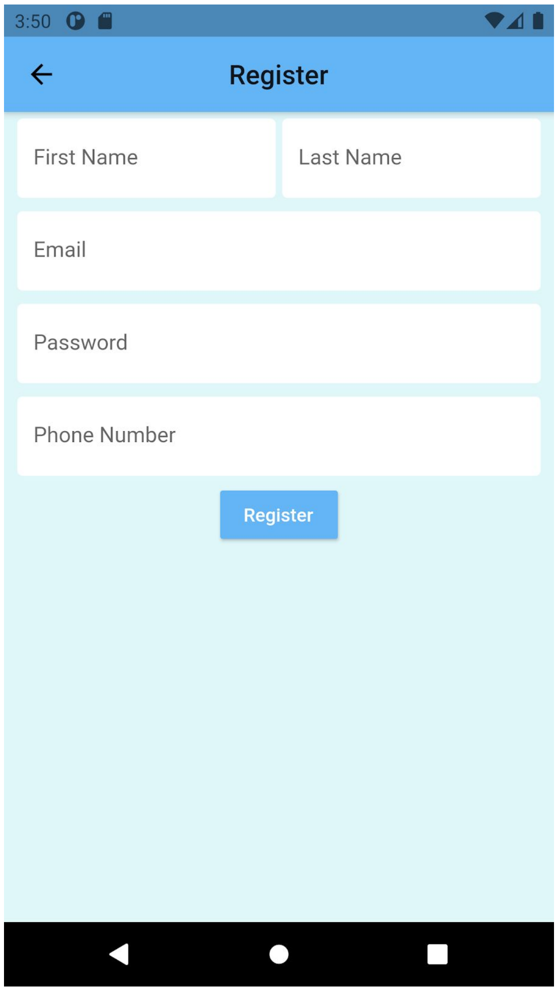
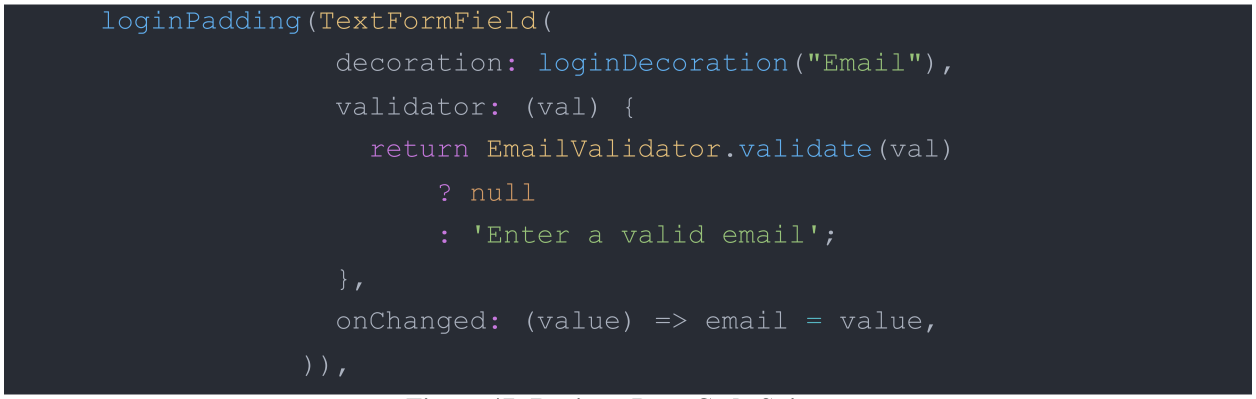


Figure 46: Register Page

The Register Page is where the user will create a new account if they do not have one. The page features TextFormFields for first name, last name, email, password and phone number. It also has a button used for registering and an arrow at the top-left of the page to go back to the login page. Like the Login Page, it has a RegisterForm that controls the functionality of the page.

Functionally, it works just like the LoginForm, each TextFormField uses a validator to check if the basic formatting is correct and the Register button uses the TextFieldForm inputs as arguments for the register method of the AuthService class, if there are no errors in the validators. For the first name and last name TextFormFields, the validators check whether or not the inputs are empty. The password validator checks if the inputted password is six characters or more. If it's less than six characters then it will return an error. The phone number validator checks if the String is 10 numbers long since phone numbers consist of 10 numbers. The keyboard for this TextFormField only contains numbers so you can’t put letters as your phone number for example.



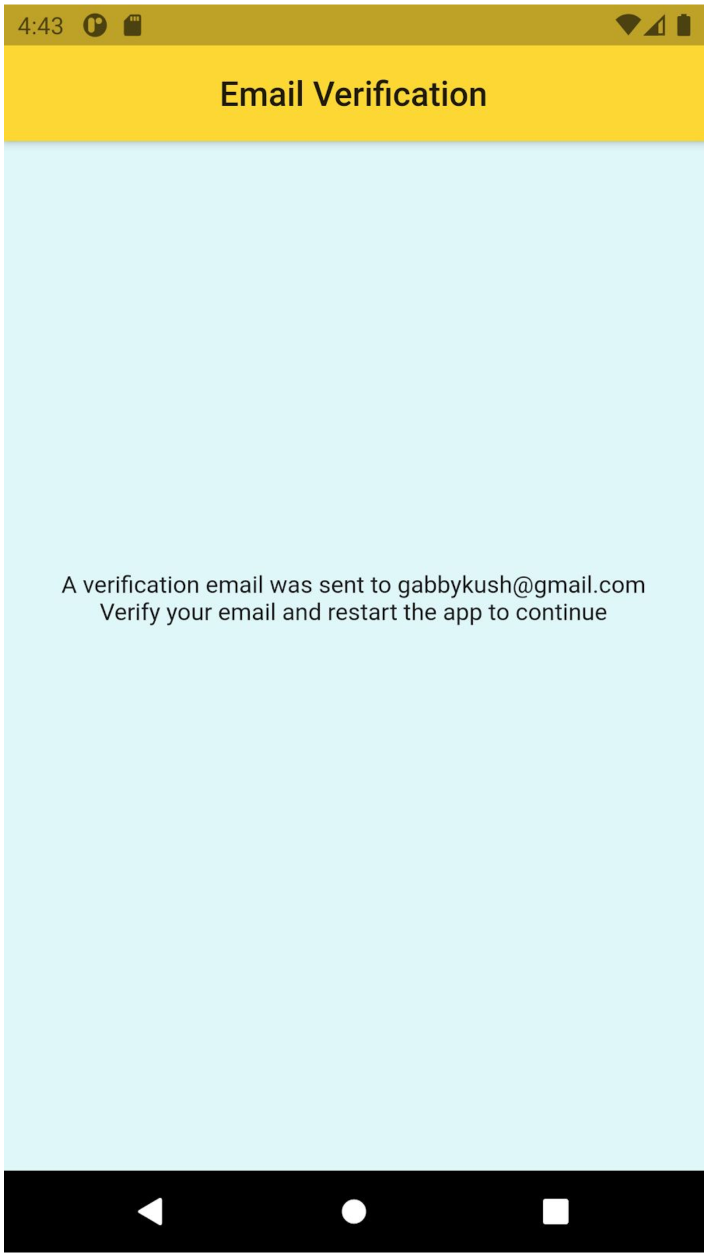
```
      loginPadding(TextFormField(
                    decoration: loginDecoration("Email"),
                    validator: (val) {
                      return EmailValidator.validate(val)
                          ? null
                          : 'Enter a valid email';
                    },
                    onChanged: (value) => email = value,
                  )),
```

Figure 47: Register Page Code Snippet

Finally, for the email TextFormField, we used a package called “email_validator” that has a function that checks whether the input String is formatted like an email (***@***.***) as shown above.

Once all the validators are satisfied and the user clicks on the Register button, the inputs are passed as arguments into the register method in the AuthService class. First, Firebase tries to create a new account using the provided email and password. If the email already exists, then an error message will pop up below the Register button stating that the email provided is already in the database. If Firebase successfully creates the new account, then it will use the other provided credentials to create a new document in the Account table. This is done through the updateAccountData method in the DatabaseService class. Finally, a verification email is sent to the provided email and the user is taken to the EmailVerificationPage.

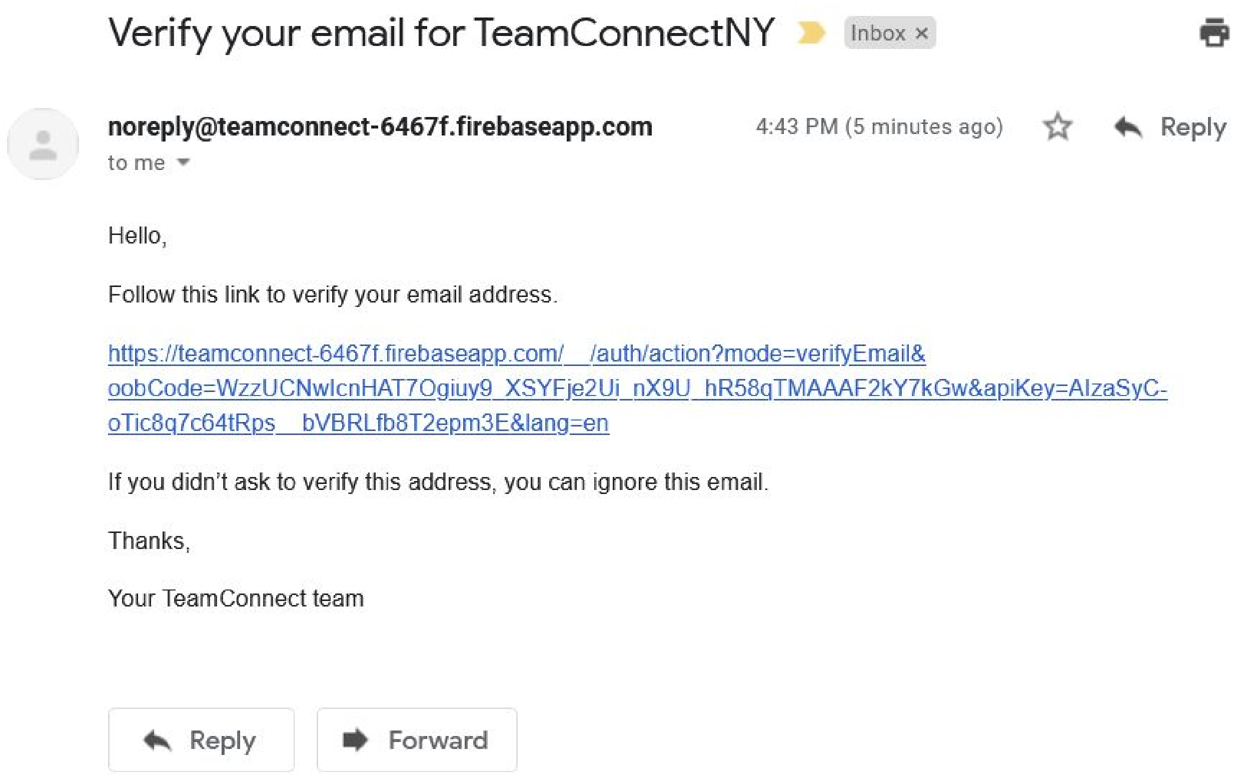
3.4.6 Email Verification Page



A verification email was sent to gabbykush@gmail.com
Verify your email and restart the app to continue

Figure 48: Email Verification Page

Once a user registers a new account, they are taken to this page above. The only thing on this page is the message that a verification email was sent to the user's provided email. Once the user clicks on the link in the email, then their email will be verified. Because of time constraints and lack of experience, we weren't able to get the verification to be autodetected while the app is still open. So for the user to continue, they have to close and reopen the app. When they do, the user is taken to the ProfileCreationPage.


Verify your email for TeamConnectNY

noreply@teamconnect-6467f.firebaseapp.com

to me

4:43 PM (5 minutes ago)

Hello,

Follow this link to verify your email address.

https://teamconnect-6467f.firebaseapp.com/__/auth/action?mode=verifyEmail&oobCode=WzzUCNwlcnHAT7Ogiuy9_XSYFje2Ui_nX9U_hR58qTMAAAF2kY7kGw&apiKey=AIzaSyC-oTic8q7c64tRps__bVBRLfb8T2epm3E&lang=en

If you didn't ask to verify this address, you can ignore this email.

Thanks,

Your TeamConnect team

Figure 49: Verification Email

Above is an example of the email that will be sent to the user's email for verification. The email is verified when the user clicks on the link.

3.4.7 Profile Creation Page

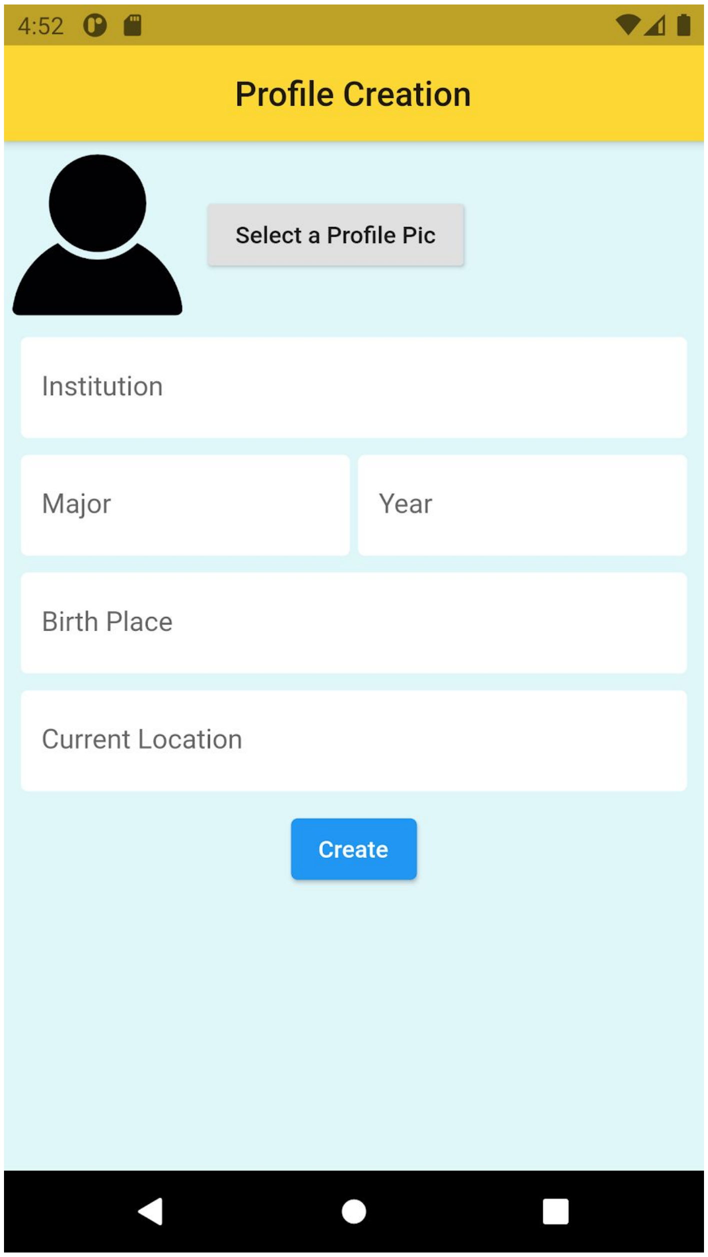


Figure 50: Profile Creation Page

The user is taken to the Profile Creation page once the user verifies their email. Here the user creates their profile to be displayed on the profile page. This page is composed of an image that will be used as the profile picture, a button to select a new profile picture (not working), TextFormFields to input the user's Institution, Major, current year, where they were born and where they currently live. Finally at the bottom is a "Create" button that will create the profile.

This page works just like the previously discussed pages, all functionality is in the ProfileCreationForm. Each TextFormField has a validator that checks whether the inputs are empty or not and the button uses the inputs to create a new document in the Profile table of Firebase.

```
child: RaisedButton(
                      onPressed: () async {
                        try {
                          profilePic = await ImagePicker()
                              .getImage(source: ImageSource.gallery)
                              .toString();
                        } catch (e) {
                          print(e);
                        }
                      },
```

Figure 51: Profile Creation Page Code Snippet 1

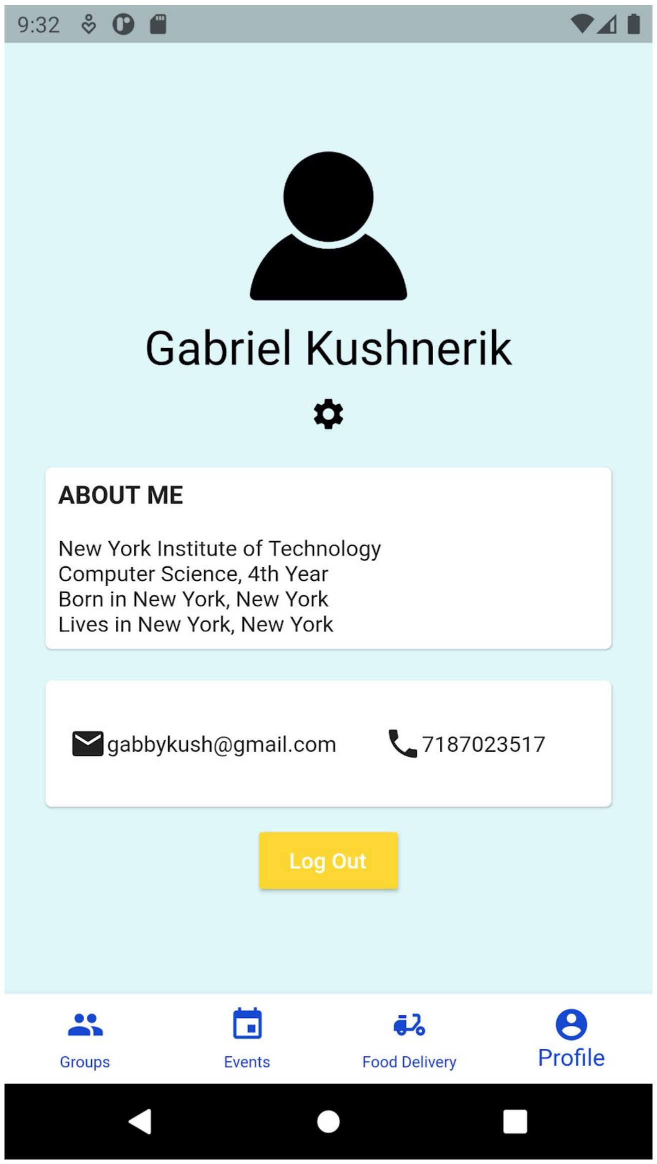
While the “Select a Profile Pic” doesn’t function, what it’s supposed to do is use a class from a package called “image_picker” in order to let the user pick a photo stored on their phone. The image would be displayed next to the button and would be saved into the database to show on the Profile page. However, an error comes up that a method that the above code uses to get pictures from the user has no implementation for a reason we cannot figure out.

```
              child: ElevatedButton(
                onPressed: () async {
                  if (_formKey.currentState.validate()) {
                    await DatabaseService(uid:
user.uid).updateProfileData(
                        profilePic,
                        institution,
                        major,
                        year,
                        birthLoc,
                        currentLoc);
                    Navigator.pushNamed(context, Routes.bottomnavbar);
                  } else
                    print('Not working');
                },
```

Figure 52: Profile Creation Page Code Snippet 2

Once all the fields are filled in and the “Create” button is pressed, the updateProfileData method of class DatabaseService is used to create a new document in the Profile table of the database and the user will be taken to the homepage of the app, the Groups Page. This would conclude the account creation process of the app and the user can now use the rest of the app.

3.4.8 Profile Page



New York Institute of Technology
Computer Science, 4th Year
Born in New York, New York
Lives in New York, New York

gabbykush@gmail.com 7187023517

Figure 53: Profile Page

The profile page is where users can check their information that was inputted during account creation and where the user can log out of their account. The user can also change their profile information by clicking on the gear icon under their name. This page is made up of the user's profile picture, their full name, a card that displays an "About Me" which contains their currently enrolled institution, their major and year, where they were born and where they currently live. The second card shows the user's email and phone number. The yellow button at the bottom while sign the user out when clicked. The user can navigate to this page by clicking on the profile button on the Bottom Navigation Bar.

All information shown above is displayed using the ProfileDisplay widget. This widget uses methods from the DatabaseService and AuthService classes in order to get the data.

```
class GetName extends StatefulWidget {
  @override
  _GetNameState createState() => _GetNameState();
}

class _GetNameState extends State<GetName> {
  @override
  Widget build(BuildContext context) {
    final user = Provider.of<User>(context);

    return StreamBuilder<Object>(
        stream: DatabaseService(uid: user.uid).account,
        builder: (context, snapshot) {
          if (snapshot.hasData) {
            Account userAccount = snapshot.data;
            return Text(userAccount.firstName + ' ' +
userAccount.lastName,
                style: TextStyle(fontSize: 30, color: Colors.black));
          }
          return Text("");
        });
  }
}
```

Figure 54: Profile Page Code Snippet 1

The above widget, GetName, gets the user's first and last name. The StreamBuilder gets data from the 'account' stream of type Account implemented in DatabaseService explained in Section 3.4.1.2.

```
class ProfileCard extends StatefulWidget {
  @override
  _ProfileCardState createState() => _ProfileCardState();
}

class _ProfileCardState extends State<ProfileCard> {
  @override
  Widget build(BuildContext context) {
    final user = Provider.of<User>(context);

    return StreamBuilder<Object>(
        stream: DatabaseService(uid: user.uid).profile,
        builder: (context, snapshot) {
          Profile profile;

          if (snapshot.hasData) {
            profile = snapshot.data;
```

Figure 55: Profile Page Code Snippet 2

The ProfileCard widget gets data from the user's Profile document and displays it in the "About Me" card shown in the Profile page screenshot above. The StreamBuilder gets data from the "profile" stream of type Profile implemented in DatabaseService explained in Section 3.4.1.3.

```
class AccountCard extends StatefulWidget {
  @override
  _AccountCardState createState() => _AccountCardState();
}

class _AccountCardState extends State<AccountCard> {
  final _auth = AuthService();
  String email;

  @override
  void initState() {
    // TODO: implement initState
    super.initState();
    email = _auth.getEmail();
  }
```

```
@override
Widget build(BuildContext context) {
  final user = Provider.of<User>(context);

  return StreamBuilder<Object>(
      stream: DatabaseService(uid: user.uid).account,
      builder: (context, snapshot) {
        if (snapshot.hasData) {
          Account account = snapshot.data;
```

Figure 56: Profile Page Code Snippet 3

The AccountCard widget gets the user's email and phone number from their Account document and displays them in a card. The StreamBuilder gets data from the 'account' stream of type Account implemented in DatabaseService explained in Section 3.4.1.2.

```
RaisedButton(
          onPressed: () async {
            LoadingScreen();
            Navigator.popUntil(context, (route) => route.isFirst);
            _auth.signOut();
          },
          color: Colors.yellow[600],
          child: Text(
            "Log Out",
            style: TextStyle(color: Colors.white),
          ),
        ),
```

Figure 57: Profile Page Code Snippet 4

The "Log Out" button uses the method signOut from class AuthService in order to log the user out of the app. When the user logs out, they are taken back to the Login Page where they would need to sign in again in order to use the app.

3.4.9 Profile Settings Page

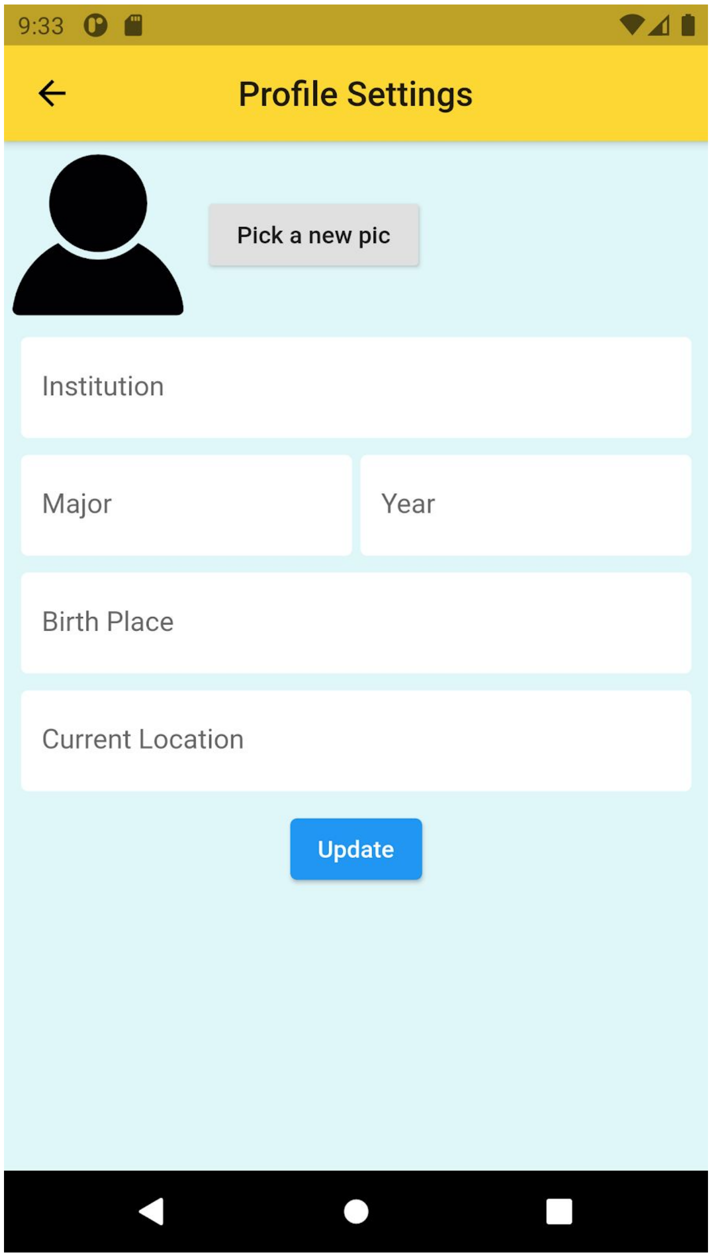
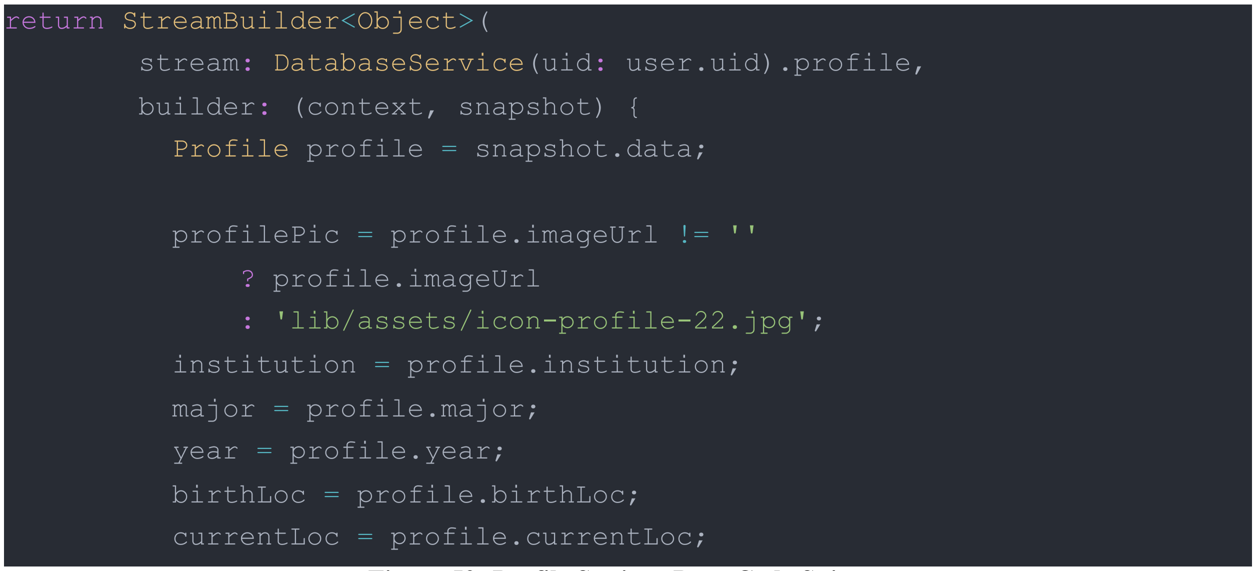


Figure 58: Profile Settings Page

When the user clicks on the gear icon in the Profile page, they are taken to this page where they can change the information displayed on their profile. This page looks almost exactly like the Profile Creation page and functions almost exactly the same. This page has an arrow button on the top left of the screen in order to go back to the profile page without changing anything. The button is also labeled “Update” instead of “Create”.



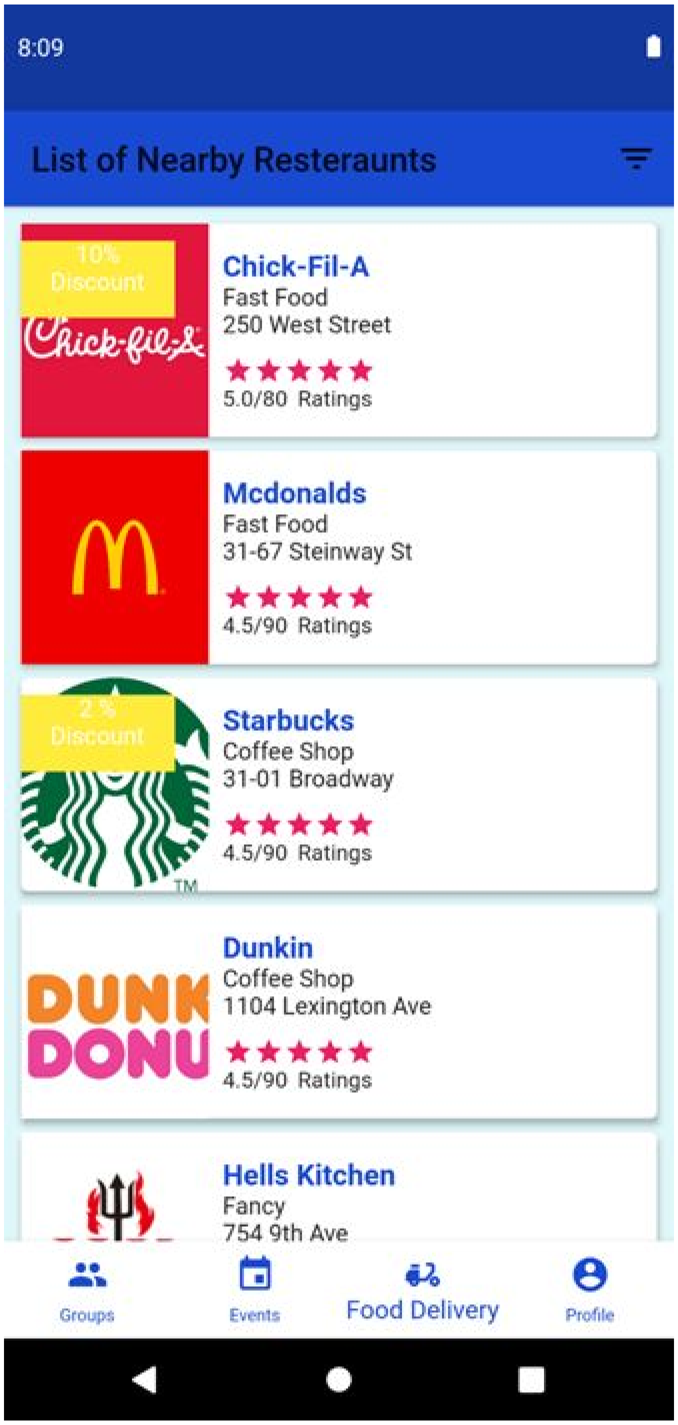
```
return StreamBuilder<Object>(
        stream: DatabaseService(uid: user.uid).profile,
        builder: (context, snapshot) {
          Profile profile = snapshot.data;

          profilePic = profile.imageUrl != ''
              ? profile.imageUrl
              : 'lib/assets/icon-profile-22.jpg';
          institution = profile.institution;
          major = profile.major;
          year = profile.year;
          birthLoc = profile.birthLoc;
          currentLoc = profile.currentLoc;
```

Figure 59: Profile Settings Page Code Snippet

The page uses the ProfileForm which is functionally the same as the ProfileCreationForm, with the only difference being that the user's profile picture is pulled from the database and is displayed on the page above.

Please refer to the **ProfileCreation Page** above to see the rest of the functionality of this page.

3.4.10. Nearby Restaurants Page


8:09

List of Nearby Resteraunts

10% Discount
Chick-Fil-A
Fast Food
250 West Street
★★★★★
5.0/80 Ratings

Mcdonalds
Fast Food
31-67 Steinway St
★★★★★
4.5/90 Ratings

2 % Discount
Starbucks
Coffee Shop
31-01 Broadway
★★★★★
4.5/90 Ratings

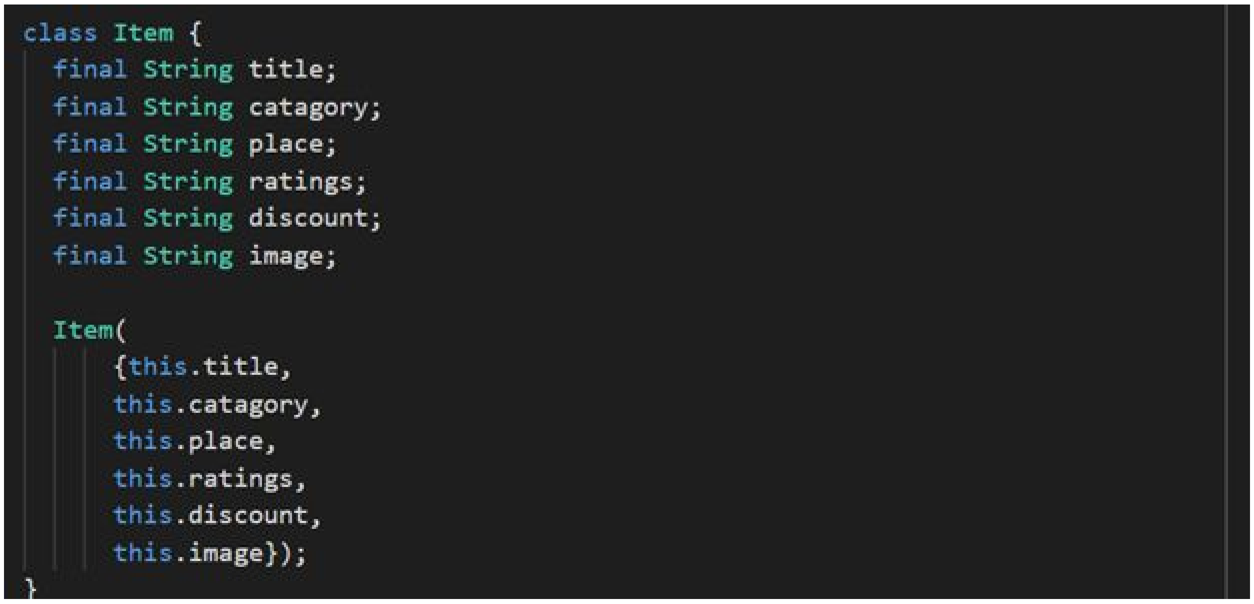
Dunkin
Coffee Shop
1104 Lexington Ave
★★★★★
4.5/90 Ratings

Hells Kitchen
Fancy
754 9th Ave

Figure 60: Nearby Restaurants Page

The nearby restaurants page is where the users can see a list of restaurants located near their location. The user can scroll up and down the list and click on their desired restaurant. The list consisted of cards which contained the restaurant logo, name, address, and rating based on user reviews. We also added a discount tag on certain cards to show the possibility of future collaboration of the school and the place to offer the students discounts.

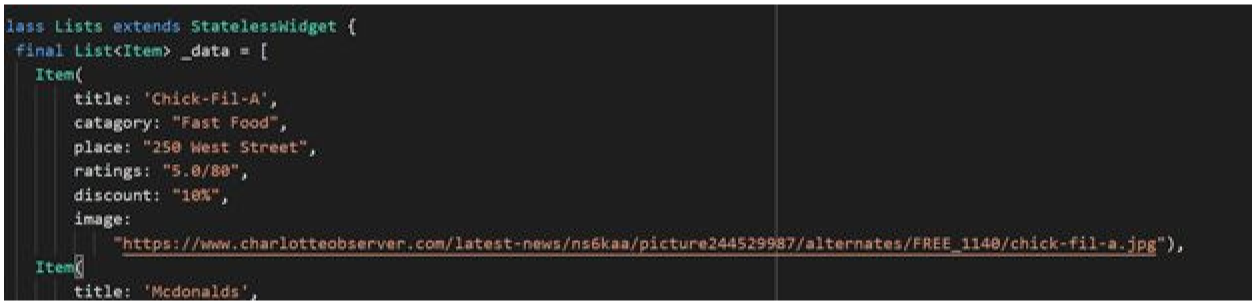
Below is a screenshot of the class called items with instances needed to create the card. This keyword was necessary to prevent confusion of variable names later.



```
class Item {
  final String title;
  final String catagory;
  final String place;
  final String ratings;
  final String discount;
  final String image;

  Item(
      {this.title,
      this.catagory,
      this.place,
      this.ratings,
      this.discount,
      this.image});
}
```

Figure 61: Nearby Restaurants Page Code Snippet 1

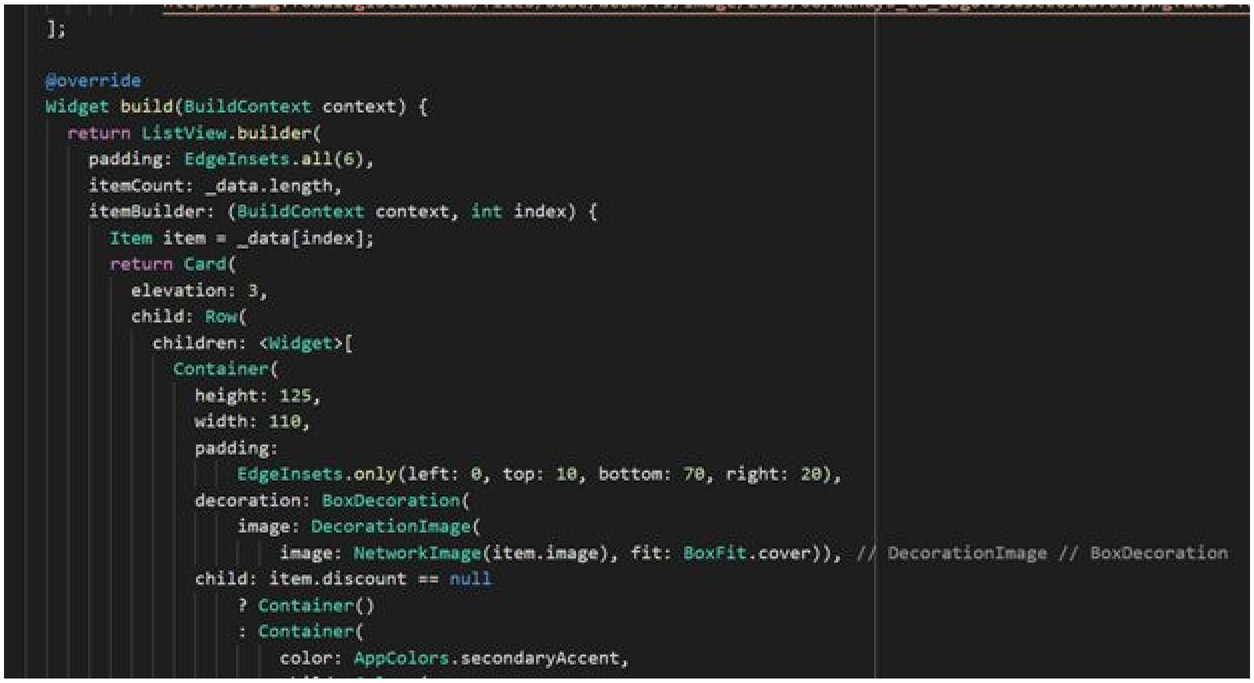
In the below screenshot, the Item class and its instances. For example purposes, we filled in the instances with text to visualize the concept. In the future, the code can be easily edited and each instance can be connected to our already established database.



```
lass Lists extends StatelessWidget {
 final List<Item> _data = [
   Item(
       title: 'Chick-Fil-A',
       catagory: "Fast Food",
       place: "250 West Street",
       ratings: "5.0/80",
       discount: "10%",
       image:
           "https://www.charlotteobserver.com/latest-news/ns6kaa/picture244529987/alternates/FREE_1140/chick-fil-a.jpg"),
   Item(
       title: 'Mcdonalds',
```

Figure 62: Nearby Restaurants Page Code Snippet 2

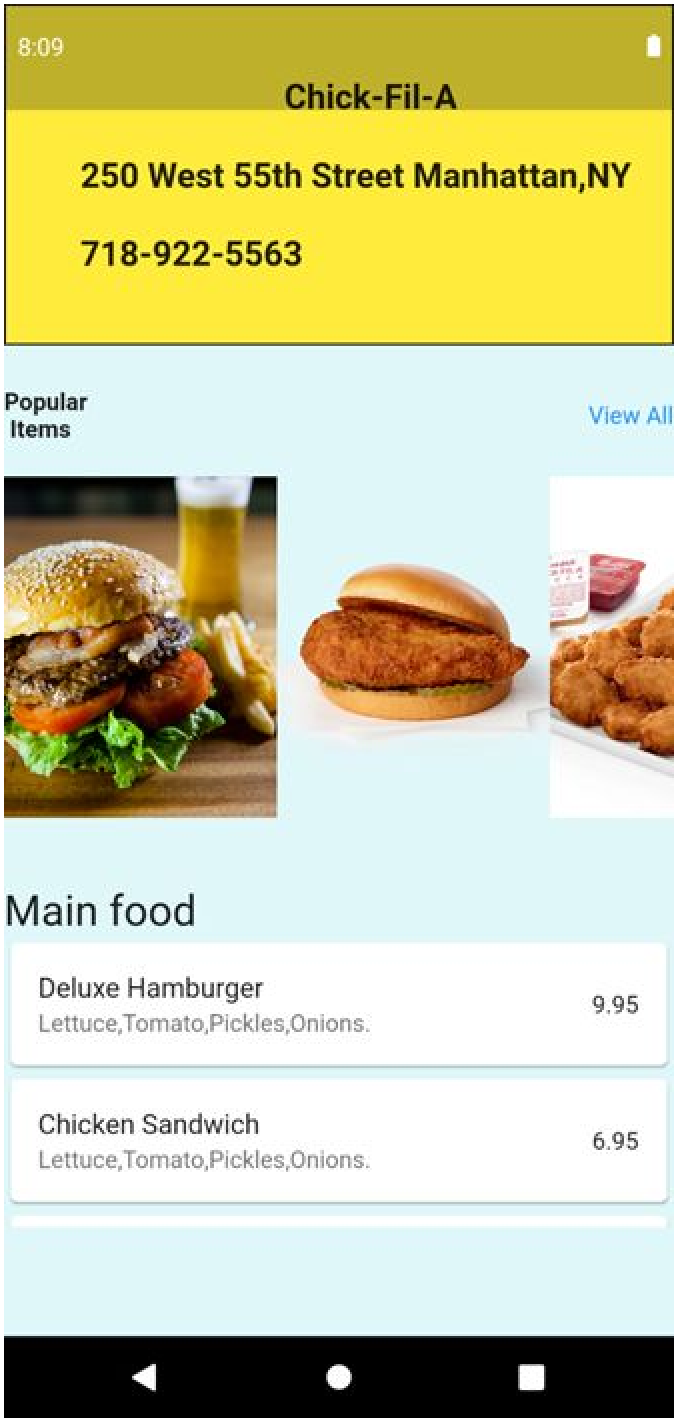
In the screenshot below the list of items is put into the Card widget which then returns several containers, creating the list UI design.



```
];

@override
Widget build(BuildContext context) {
  return ListView.builder(
    padding: EdgeInsets.all(6),
    itemCount: _data.length,
    itemBuilder: (BuildContext context, int index) {
      Item item = _data[index];
      return Card(
        elevation: 3,
        child: Row(
          children: <Widget>[
            Container(
              height: 125,
              width: 110,
              padding:
                  EdgeInsets.only(left: 0, top: 10, bottom: 70, right: 20),
              decoration: BoxDecoration(
                  image: DecorationImage(
                      image: NetworkImage(item.image), fit: BoxFit.cover)), // DecorationImage // BoxDecoration
              child: item.discount == null
                  ? Container()
                  : Container(
                      color: AppColors.secondaryAccent,
```

Figure 63: Nearby Restaurants Page Code Snippet 3

3.4.11 Restaurant Page


8:09

Chick-Fil-A

250 West 55th Street Manhattan,NY

718-922-5563

Popular Items

View All

Main food

Deluxe Hamburger
Lettuce,Tomato,Pickles,Onions.
9.95

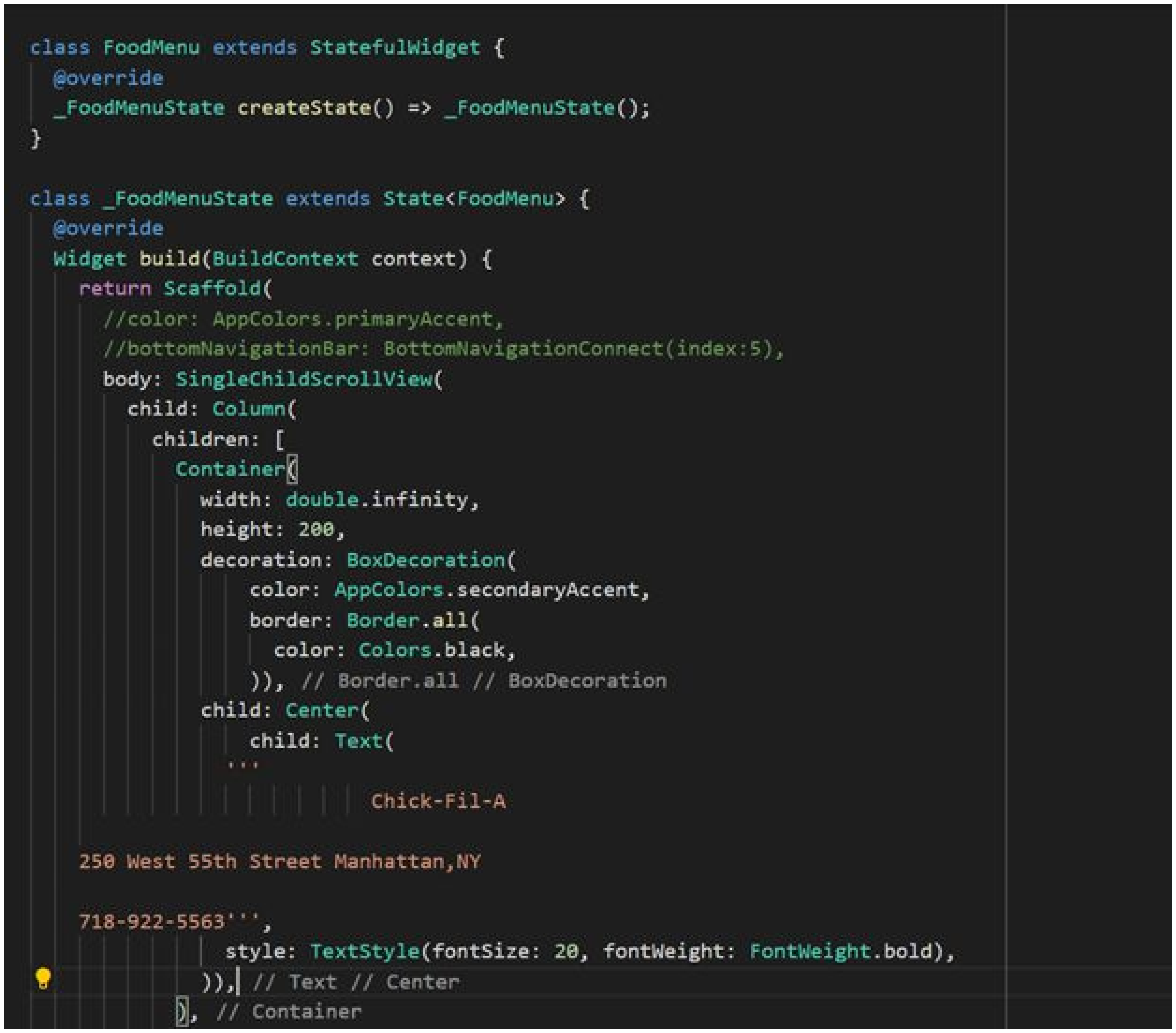
Chicken Sandwich
Lettuce,Tomato,Pickles,Onions.
6.95

Figure 64: Restaurant Page

This our restaurant page. This page is split into three sections. The top contains the name of the restaurant, address and phone number. The middle part of the section contains a horizontal scroll of popular food items from that restaurant with an option of selecting view all, which will pull the page down to our third section, the “Main Food”. In the main Food section, the user sees the entire food menu, including desserts, drinks and side dishes.

Once the user has decided on the food they choose, they simply click the card and the item will automatically be added to their checkout. The user can select the item several times to increase the quantity in case they are extra hungry.

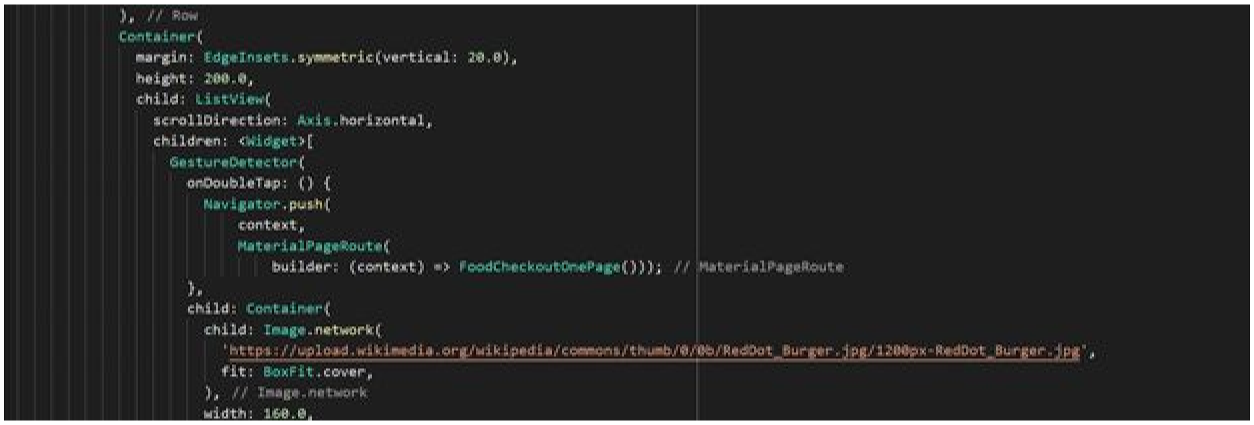
In the below screenshot, it is the code for the section on top which contains the restaurant’s name, address and phone number. For this section, we used a unique method of using three apostrophes, which allows us to use the white space to freely move the text. The text can then be positioned correctly left or right with the space bar.



```dart
class FoodMenu extends StatefulWidget {
  @override
  _FoodMenuState createState() => _FoodMenuState();
}

class _FoodMenuState extends State<FoodMenu> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      //color: AppColors.primaryAccent,
      //bottomNavigationBar: BottomNavigationConnect(index:5),
      body: SingleChildScrollView(
        child: Column(
          children: [
            Container(
              width: double.infinity,
              height: 200,
              decoration: BoxDecoration(
                  color: AppColors.secondaryAccent,
                  border: Border.all(
                    color: Colors.black,
                  )), // Border.all // BoxDecoration
              child: Center(
                  child: Text(
                '''
                          Chick-Fil-A

    250 West 55th Street Manhattan,NY

    718-922-5563''',
                style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
              )), // Text // Center
            ), // Container
```

Figure 65: Restaurant Page Code Snippet 1

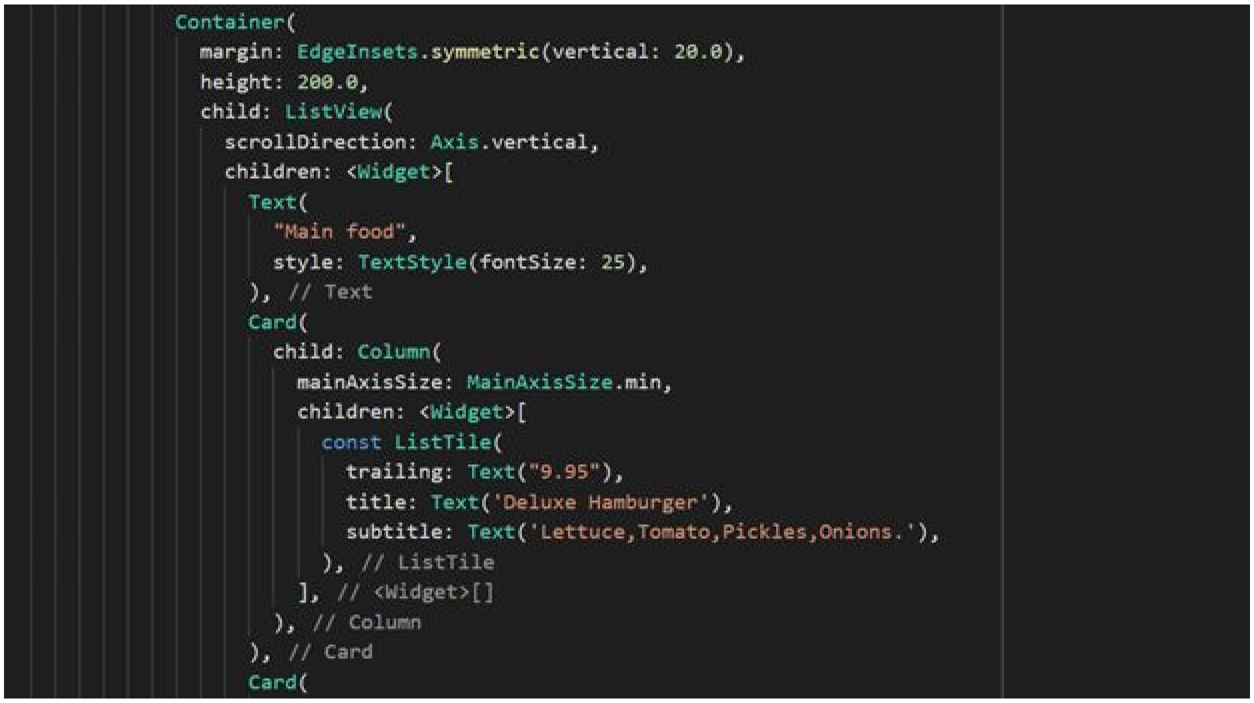
To create the horizontal scroll feature of the middle section of this place, the widget ListView was used. ListView has a unique property called scrollDirection, which we can change the direction of the scroll from going up and down to left and right. Each picture is linked out our food checkout page. So the user can select the desired item and have it be added to their checkout.



```
), // Row
Container(
  margin: EdgeInsets.symmetric(vertical: 20.0),
  height: 200.0,
  child: ListView(
    scrollDirection: Axis.horizontal,
    children: <Widget>[
      GestureDetector(
        onDoubleTap: () {
          Navigator.push(
              context,
              MaterialPageRoute(
                  builder: (context) => FoodCheckoutOnePage())); // MaterialPageRoute
        },
        child: Container(
          child: Image.network(
            'https://upload.wikimedia.org/wikipedia/commons/thumb/0/0b/RedDot_Burger.jpg/1200px-RedDot_Burger.jpg',
            fit: BoxFit.cover,
          ), // Image.network
          width: 160.0,
```

Figure 66: Restaurant Page Code Snippet 2

For the third section of the page, ListView was used again, except the scroll direction was changed from horizontal to vertical so the user can scroll the full menu up and down. For the menu items, a widget called ListTile was used. The listTile is a perfect widget for these situations as it allows a main text to be followed with a trailing text, which can be used for pricing. It also allows a subtitle for food description.



```
Container(
  margin: EdgeInsets.symmetric(vertical: 20.0),
  height: 200.0,
  child: ListView(
    scrollDirection: Axis.vertical,
    children: <Widget>[
      Text(
        "Main food",
        style: TextStyle(fontSize: 25),
      ), // Text
      Card(
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: <Widget>[
            const ListTile(
              trailing: Text("9.95"),
              title: Text('Deluxe Hamburger'),
              subtitle: Text('Lettuce,Tomato,Pickles,Onions.'),
            ), // ListTile
          ], // <Widget>[]
        ), // Column
      ), // Card
      Card(
```

Figure 67: Restaurant Page Code Snippet 3

3.4.12 Check-Out Page

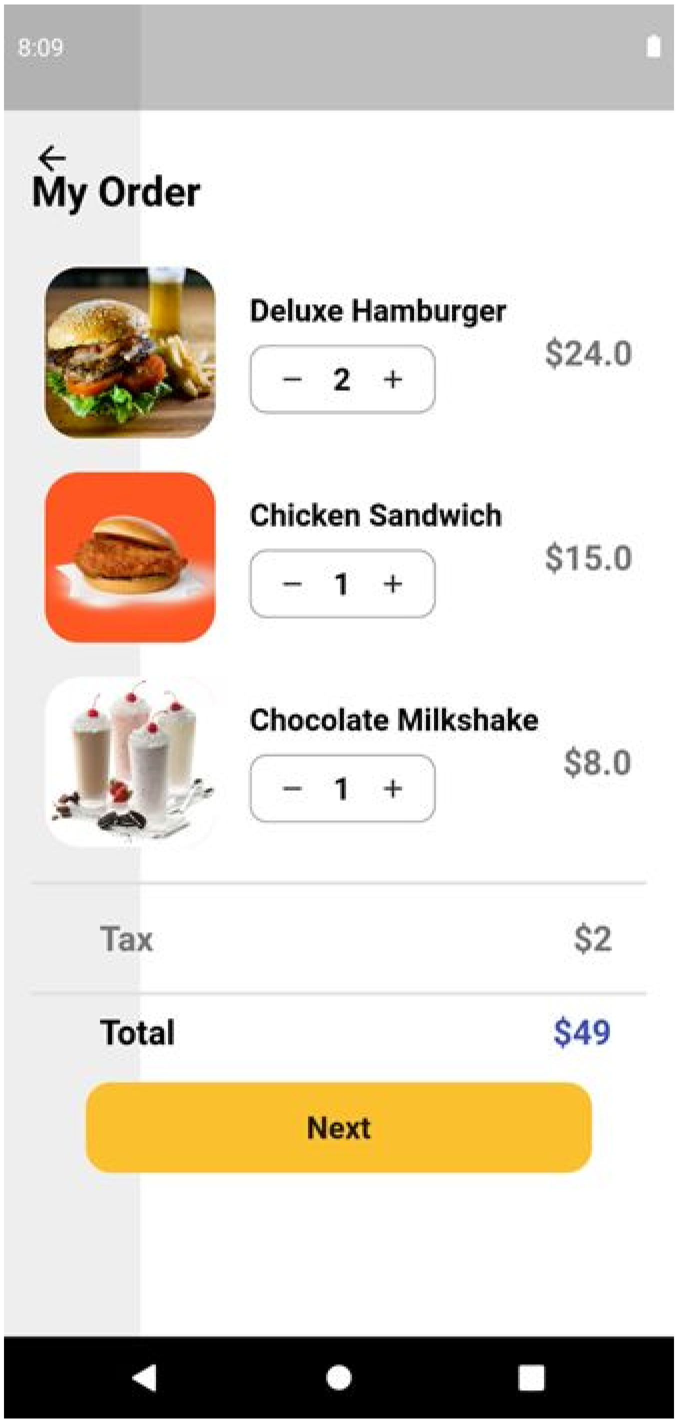
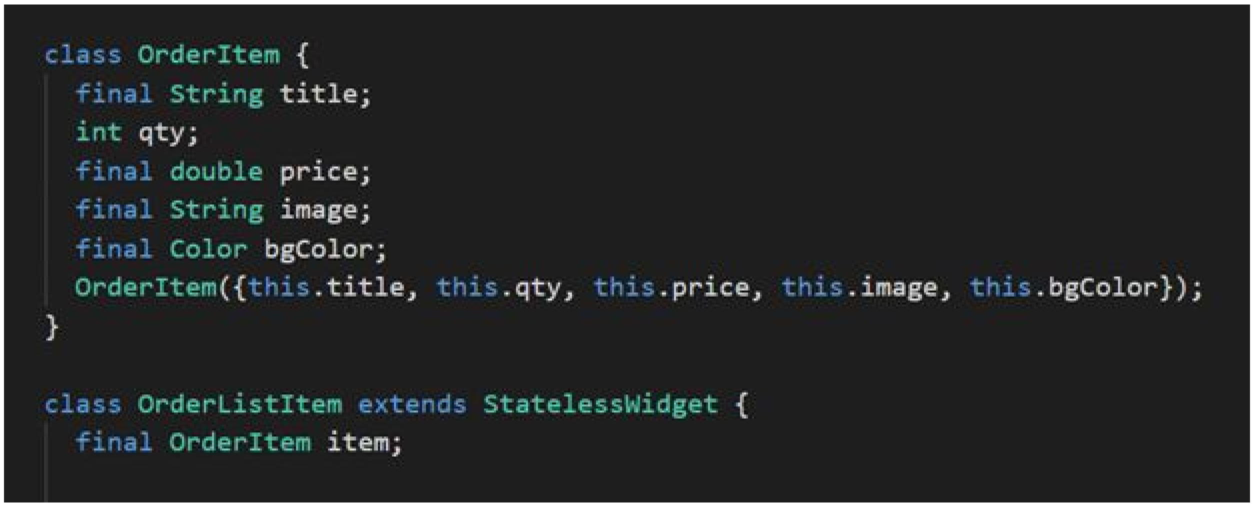


Figure 68: Check-Out Page

This is our food check out page. Here the user can see what they ordered. They can see a picture of each item with the name, quantity and total price of each item. The user can select the plus or minus buttons to increase the quantity of each item. When the user updates the quantity, the total price of each item, tax, and overall total price gets updated. The user then has the option of pressing the button named “Next” to go to the payment option page.

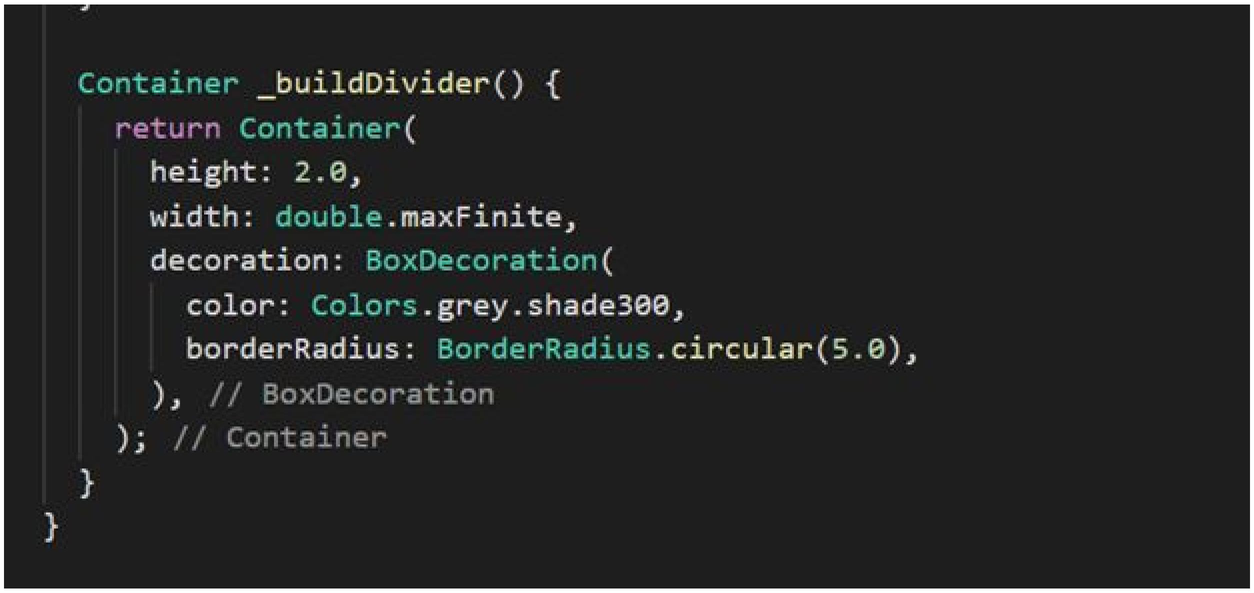
The below screenshot is the Class named OrderItem for this page. The instances, title, price, image are the same as the other classes so they can be used later to connect to the database. The instances qty which stands for quantity and bgColor are used specifically for this page due to its unique design.



```
class OrderItem {
  final String title;
  int qty;
  final double price;
  final String image;
  final Color bgColor;
  OrderItem({this.title, this.qty, this.price, this.image, this.bgColor});
}

class OrderListItem extends StatelessWidget {
  final OrderItem item;
```

Figure 69: Check-Out Page Code Snippet 1

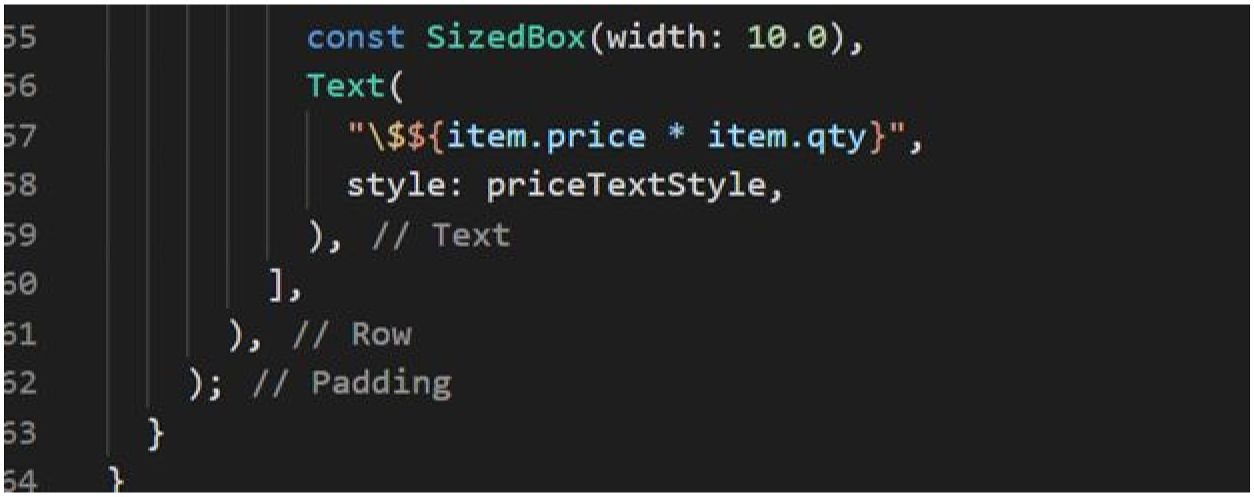
Here is a Container called _buildDivider(). Inside this container contains another container which creates a thin grey line to separate the tax and total.



```
  Container _buildDivider() {
    return Container(
      height: 2.0,
      width: double.maxFinite,
      decoration: BoxDecoration(
        color: Colors.grey.shade300,
        borderRadius: BorderRadius.circular(5.0),
      ), // BoxDecoration
    ); // Container
  }
}
```

Figure 70: Check-Out Page Code Snippet 2

In this screenshot, contains the bottom portion of the page where the total is calculated. Inside the text widget, contains a function that multiplies the price of the item with how many items there are in total.



```
                const SizedBox(width: 10.0),
                Text(
                  "\$${item.price * item.qty}",
                  style: priceTextStyle,
                ), // Text
              ],
            ), // Row
          ); // Padding
        }
      }
```

Figure 71: Check-Out Page Code Snippet 3

3.4.13 Payment Option Page

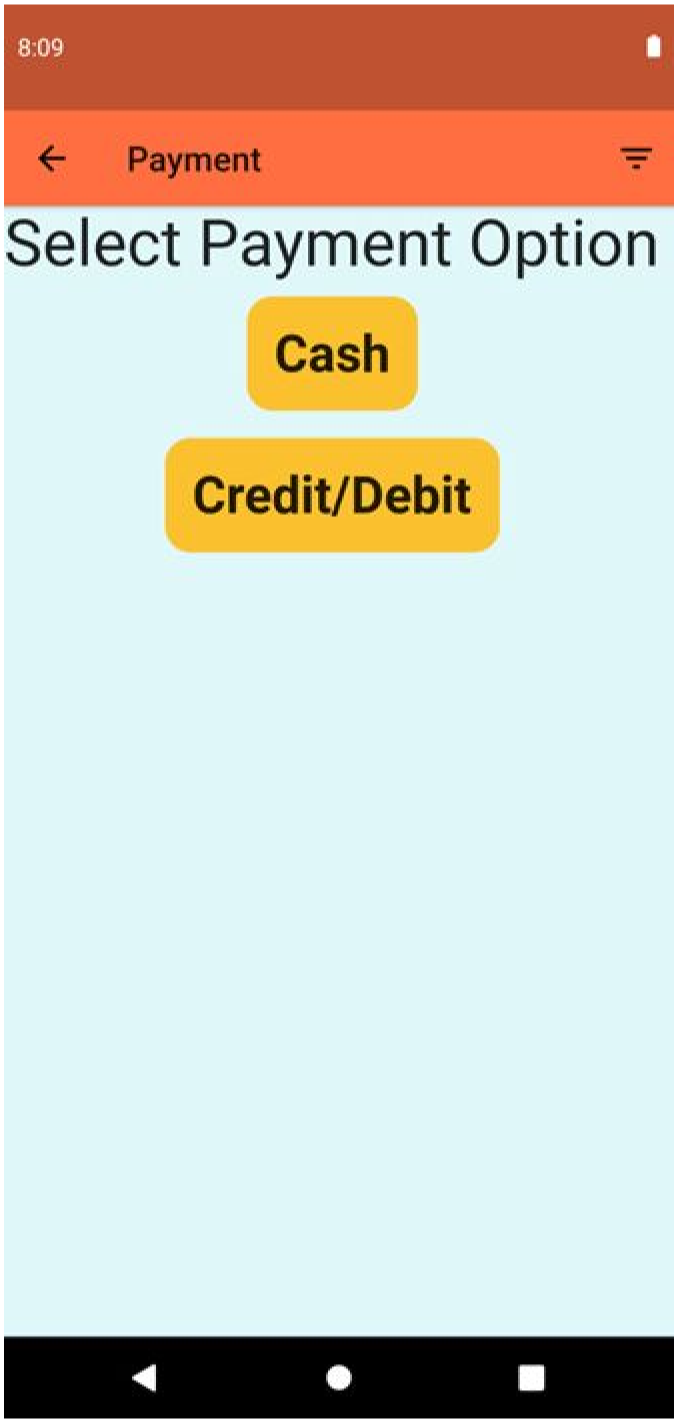
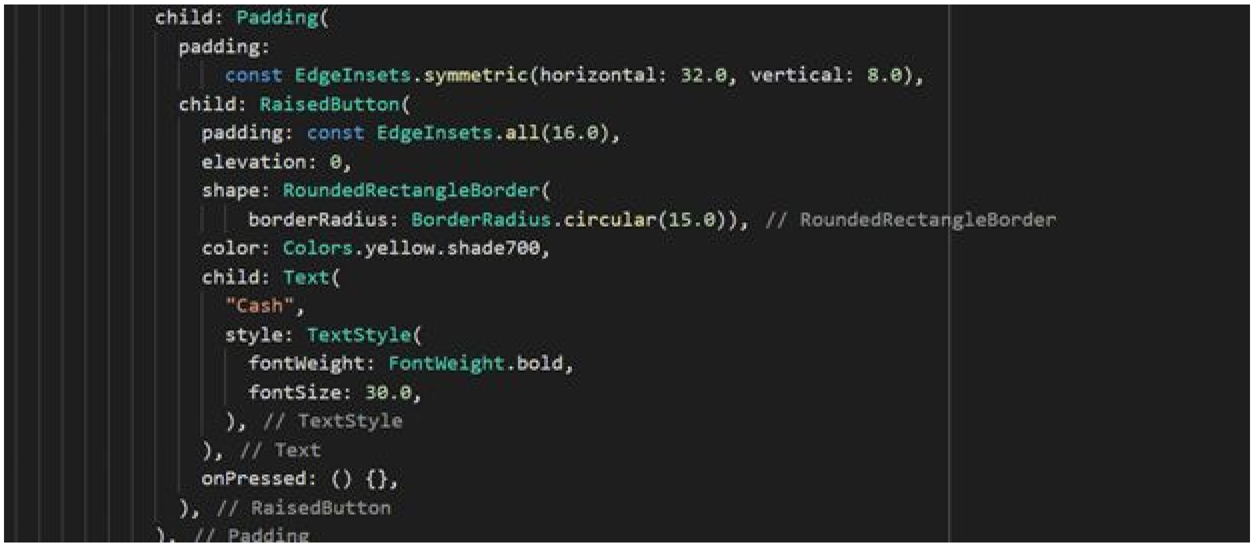


Figure 72: Payment Option Page

This is the payment selection page of the restaurant section of the page. The user can select one of two payment options. If the user selects the cash option, it will take them to a page with a message that confirms their purchase, and informs them to pay the delivery driver. If the user selects the Credit/Debit button, they will be redirected to the credit card payment page.

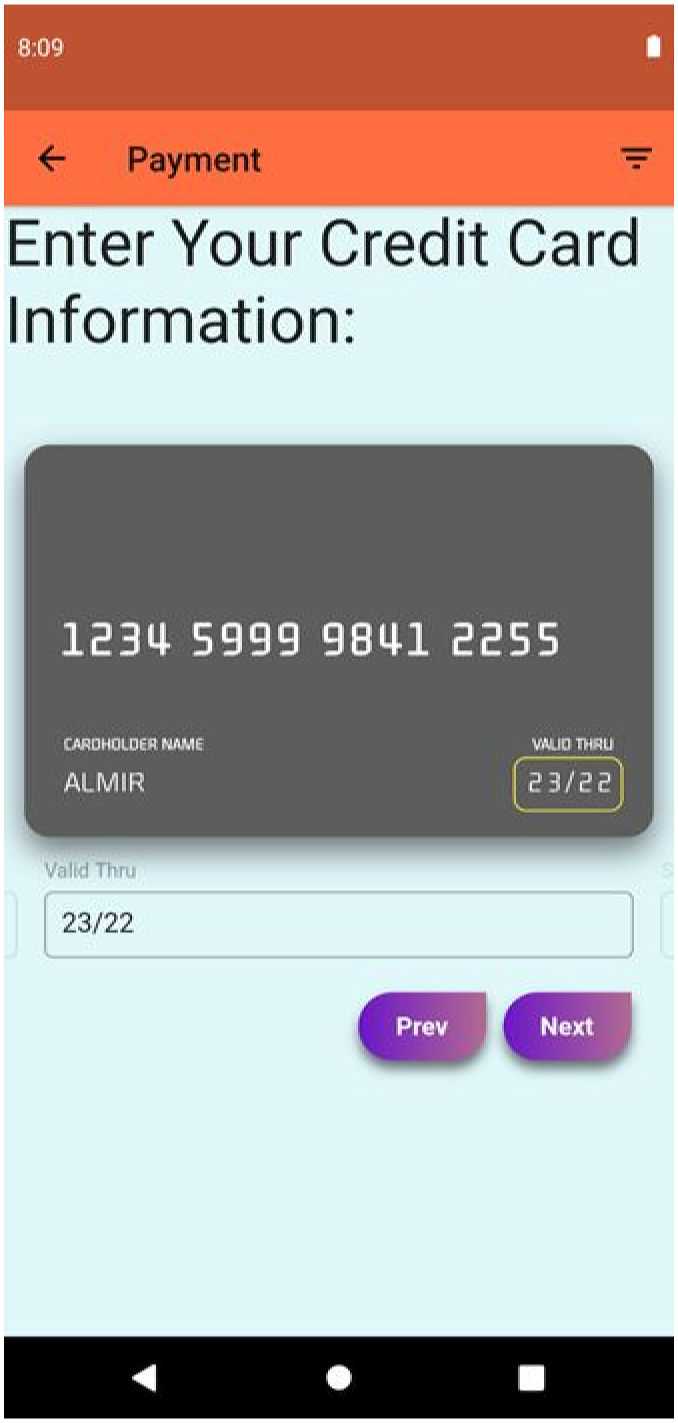
Below is a screenshot of the payment page. This page was rather simple so it only contained two buttons. The code below which is for the cash button was also used for the credit/debit button, with the only difference of the navigation of which page the user ends up on.



```
child: Padding(
  padding:
      const EdgeInsets.symmetric(horizontal: 32.0, vertical: 8.0),
  child: RaisedButton(
    padding: const EdgeInsets.all(16.0),
    elevation: 0,
    shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(15.0)), // RoundedRectangleBorder
    color: Colors.yellow.shade700,
    child: Text(
      "Cash",
      style: TextStyle(
        fontWeight: FontWeight.bold,
        fontSize: 30.0,
      ), // TextStyle
    ), // Text
    onPressed: () {},
  ), // RaisedButton
), // Padding
```

Figure 73: Payment Option Page Code Snippet

3.4.14 Credit Card Payment Page



Enter Your Credit Card Information:

1234 5999 9841 2255

CARDHOLDER NAME
ALMIR

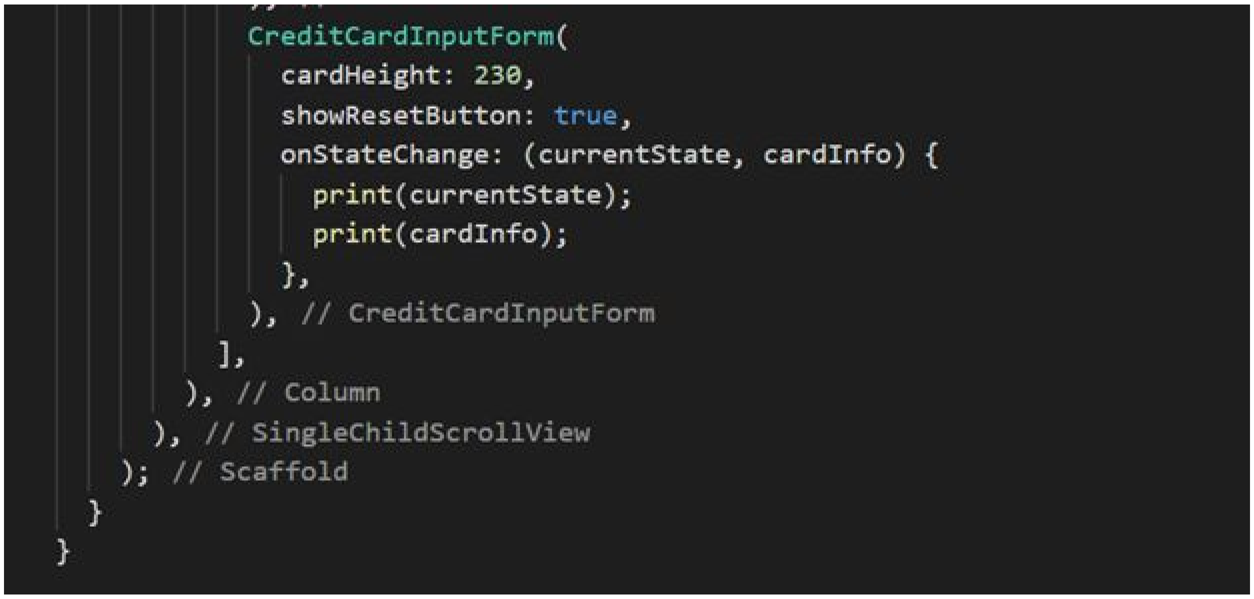
VALID THRU
23/22

Valid Thru
23/22

Figure 74: Credit Card Payment Page

This is the credit/debit card payment page. The user is met with a blank virtual credit card UI. The user then inputs their credit card number, cardholder name, and expiration date. Once the user is done, they can click next which the virtual card flips over to the back. The user then can put in their CVC number and click finish. Once they press the finish button, the payment and user information gets securely transferred to the restaurant.

For the entire functionality and UI of this page, we used a package which was called credit_card_input_form. This package contained all the necessary functions and UI we needed. In the below screenshot, we used the already created widget, CreditCardInputForm. We adjusted the height and removed the width because it wasn't necessary.



```
              CreditCardInputForm(
                cardHeight: 230,
                showResetButton: true,
                onStateChange: (currentState, cardInfo) {
                  print(currentState);
                  print(cardInfo);
                },
              ), // CreditCardInputForm
            ],
          ), // Column
        ), // SingleChildScrollView
      ); // Scaffold
    }
  }
```

Figure 75: Credit Card Payment Page Code Snippet

3.4.15 Events Page

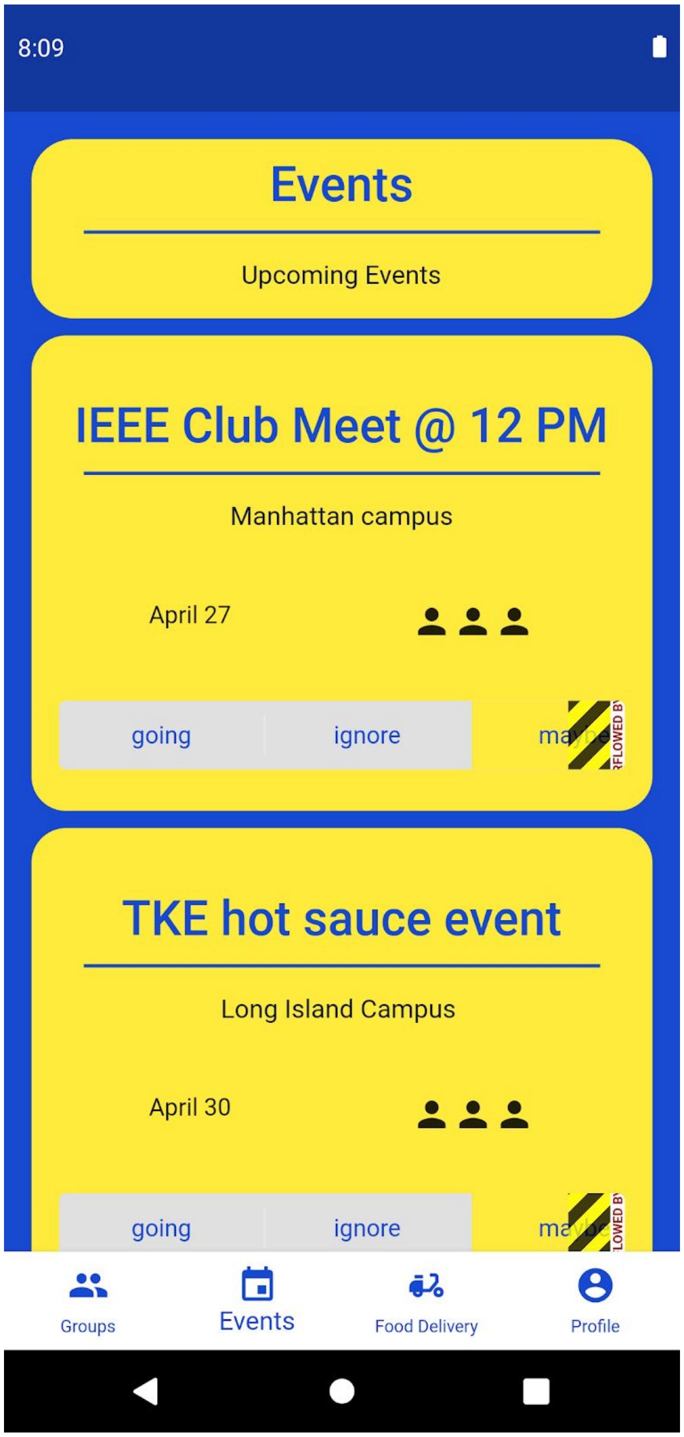
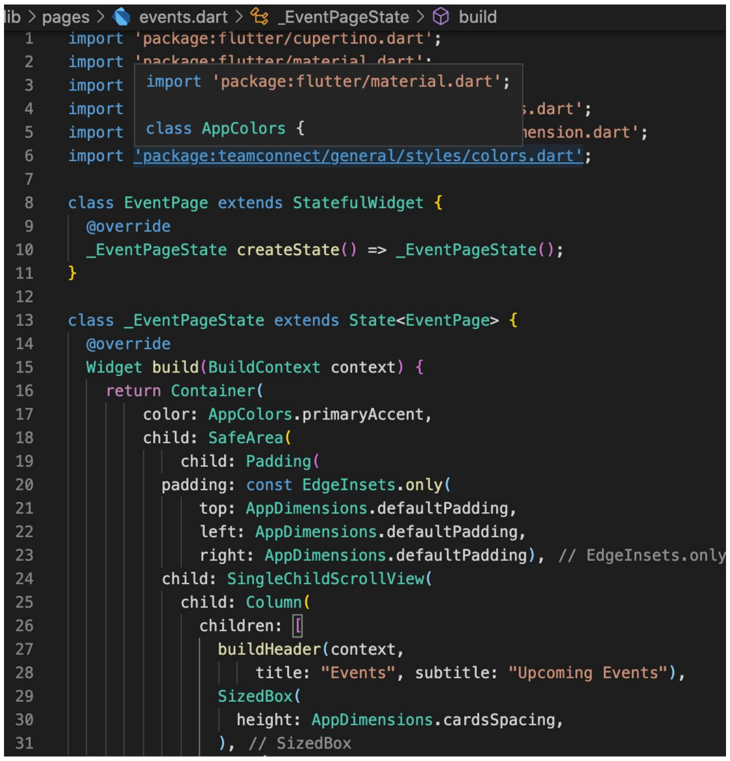


Figure 76: Events Page

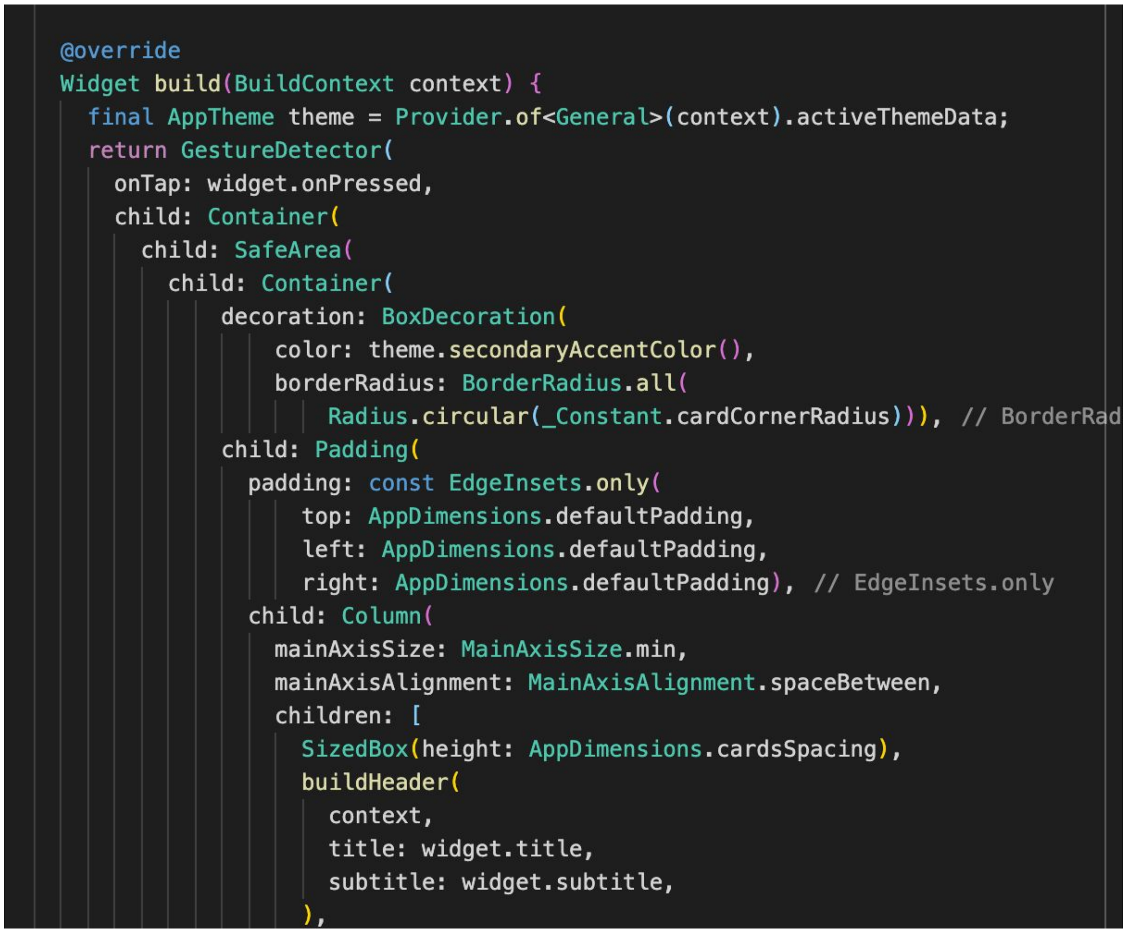
This is our Events page. Here the admins post events at their campus for the user to see. The user then can select one of three options; going, ignore, maybe. On the page, the user can see the name of the event, time, date and which campus it will be on.



```
lib > pages > events.dart > _EventPageState > build
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import  import 'package:flutter/material.dart';
import                                      .dart';
import  class AppColors {                   nension.dart';
import 'package:teamconnect/general/styles/colors.dart';

class EventPage extends StatefulWidget {
  @override
  _EventPageState createState() => _EventPageState();
}

class _EventPageState extends State<EventPage> {
  @override
  Widget build(BuildContext context) {
    return Container(
        color: AppColors.primaryAccent,
        child: SafeArea(
            child: Padding(
          padding: const EdgeInsets.only(
              top: AppDimensions.defaultPadding,
              left: AppDimensions.defaultPadding,
              right: AppDimensions.defaultPadding), // EdgeInsets.only
          child: SingleChildScrollView(
            child: Column(
              children: [
                buildHeader(context,
                    title: "Events", subtitle: "Upcoming Events"),
                SizedBox(
                  height: AppDimensions.cardsSpacing,
                ), // SizedBox
```

Figure 77: Events Page Code Snippet 1

Above we have the screen implementation for the Events page and below we have a screenshot of the widget created specifically for this page that can be reused anywhere in the application. Here the users are able to view a list of the events currently stored and retrieved from the database and they are able to navigate using the create event widget to create and post their own event.



```
@override
Widget build(BuildContext context) {
  final AppTheme theme = Provider.of<General>(context).activeThemeData;
  return GestureDetector(
    onTap: widget.onPressed,
    child: Container(
      child: SafeArea(
        child: Container(
            decoration: BoxDecoration(
                color: theme.secondaryAccentColor(),
                borderRadius: BorderRadius.all(
                    Radius.circular(_Constant.cardCornerRadius))), // BorderRad
            child: Padding(
              padding: const EdgeInsets.only(
                  top: AppDimensions.defaultPadding,
                  left: AppDimensions.defaultPadding,
                  right: AppDimensions.defaultPadding), // EdgeInsets.only
              child: Column(
                mainAxisSize: MainAxisSize.min,
                mainAxisAlignment: MainAxisAlignment.spaceBetween,
                children: [
                  SizedBox(height: AppDimensions.cardsSpacing),
                  buildHeader(
                    context,
                    title: widget.title,
                    subtitle: widget.subtitle,
                  ),
```

Figure 78: Events Page Code Snippet 2

3.4.16 Groups Page

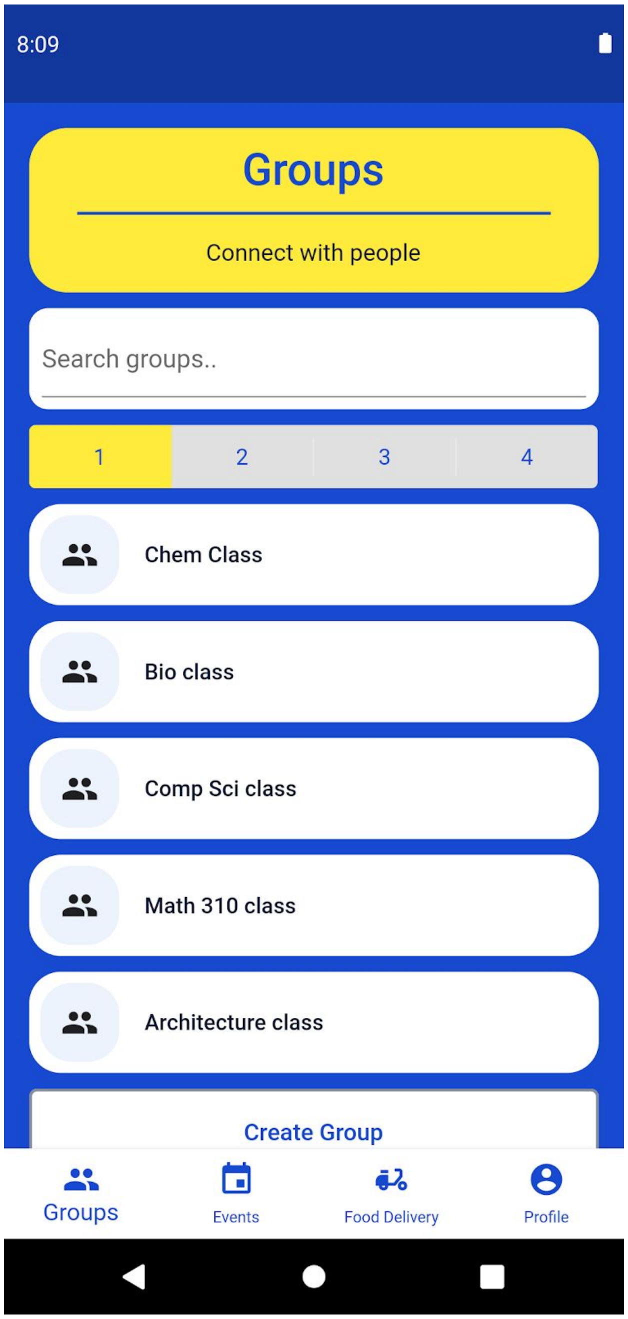
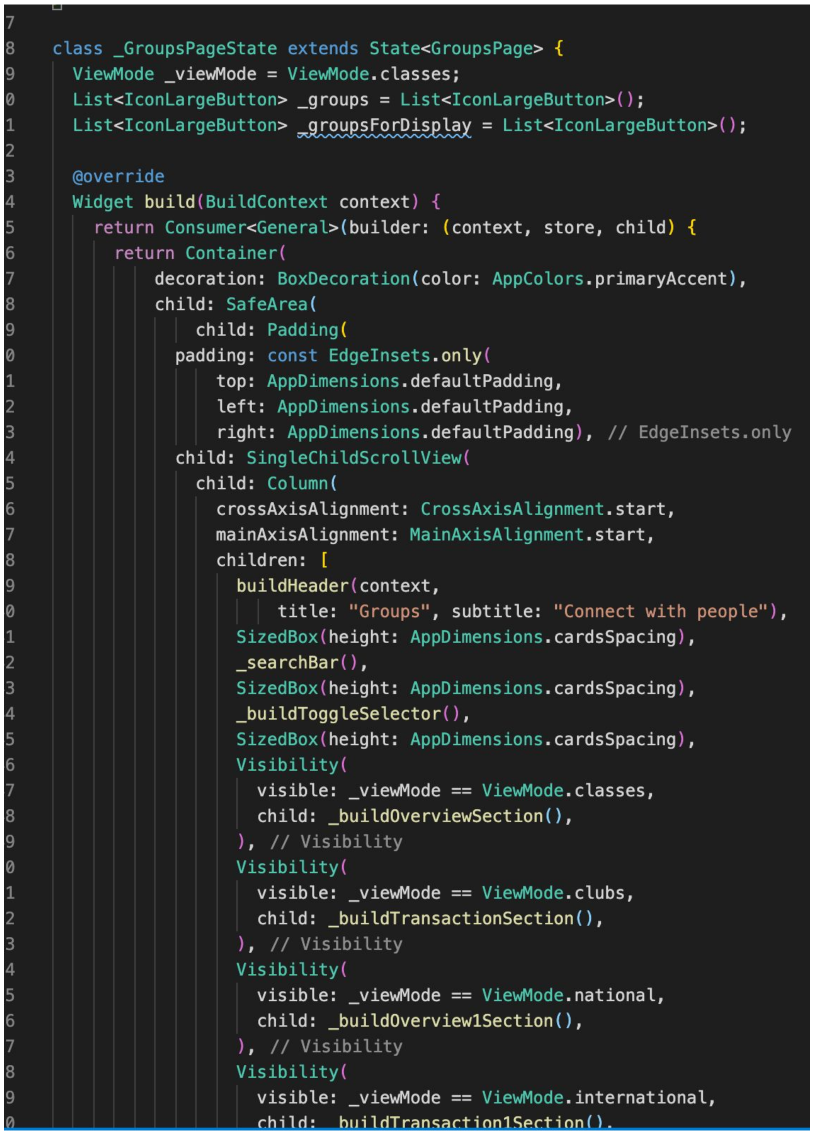


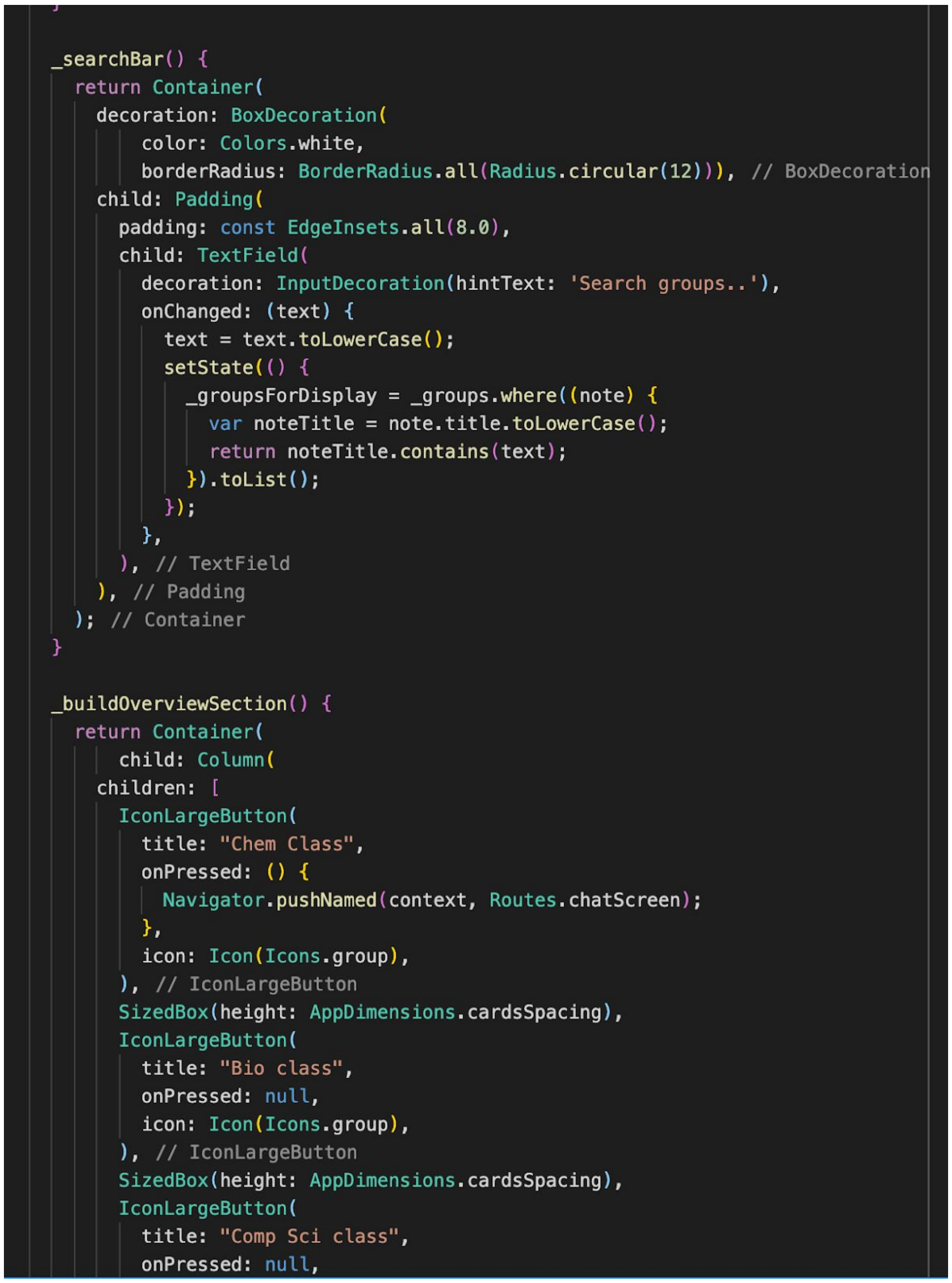
Figure 79: Groups Page

This is our groups page. Here the users can see groups, search, join and create a group. This is the main functionality of our application. The user can connect with other classmates from his or her school. They can talk about various topics such as asking for homework help.



```
class _GroupsPageState extends State<GroupsPage> {
  ViewMode _viewMode = ViewMode.classes;
  List<IconLargeButton> _groups = List<IconLargeButton>();
  List<IconLargeButton> _groupsForDisplay = List<IconLargeButton>();

  @override
  Widget build(BuildContext context) {
    return Consumer<General>(builder: (context, store, child) {
      return Container(
          decoration: BoxDecoration(color: AppColors.primaryAccent),
          child: SafeArea(
              child: Padding(
            padding: const EdgeInsets.only(
                top: AppDimensions.defaultPadding,
                left: AppDimensions.defaultPadding,
                right: AppDimensions.defaultPadding), // EdgeInsets.only
            child: SingleChildScrollView(
              child: Column(
                crossAxisAlignment: CrossAxisAlignment.start,
                mainAxisAlignment: MainAxisAlignment.start,
                children: [
                  buildHeader(context,
                      title: "Groups", subtitle: "Connect with people"),
                  SizedBox(height: AppDimensions.cardsSpacing),
                  _searchBar(),
                  SizedBox(height: AppDimensions.cardsSpacing),
                  _buildToggleSelector(),
                  SizedBox(height: AppDimensions.cardsSpacing),
                  Visibility(
                    visible: _viewMode == ViewMode.classes,
                    child: _buildOverviewSection(),
                  ), // Visibility
                  Visibility(
                    visible: _viewMode == ViewMode.clubs,
                    child: _buildTransactionSection(),
                  ), // Visibility
                  Visibility(
                    visible: _viewMode == ViewMode.national,
                    child: _buildOverview1Section(),
                  ), // Visibility
                  Visibility(
                    visible: _viewMode == ViewMode.international,
                    child: _buildTransaction1Section(),
```

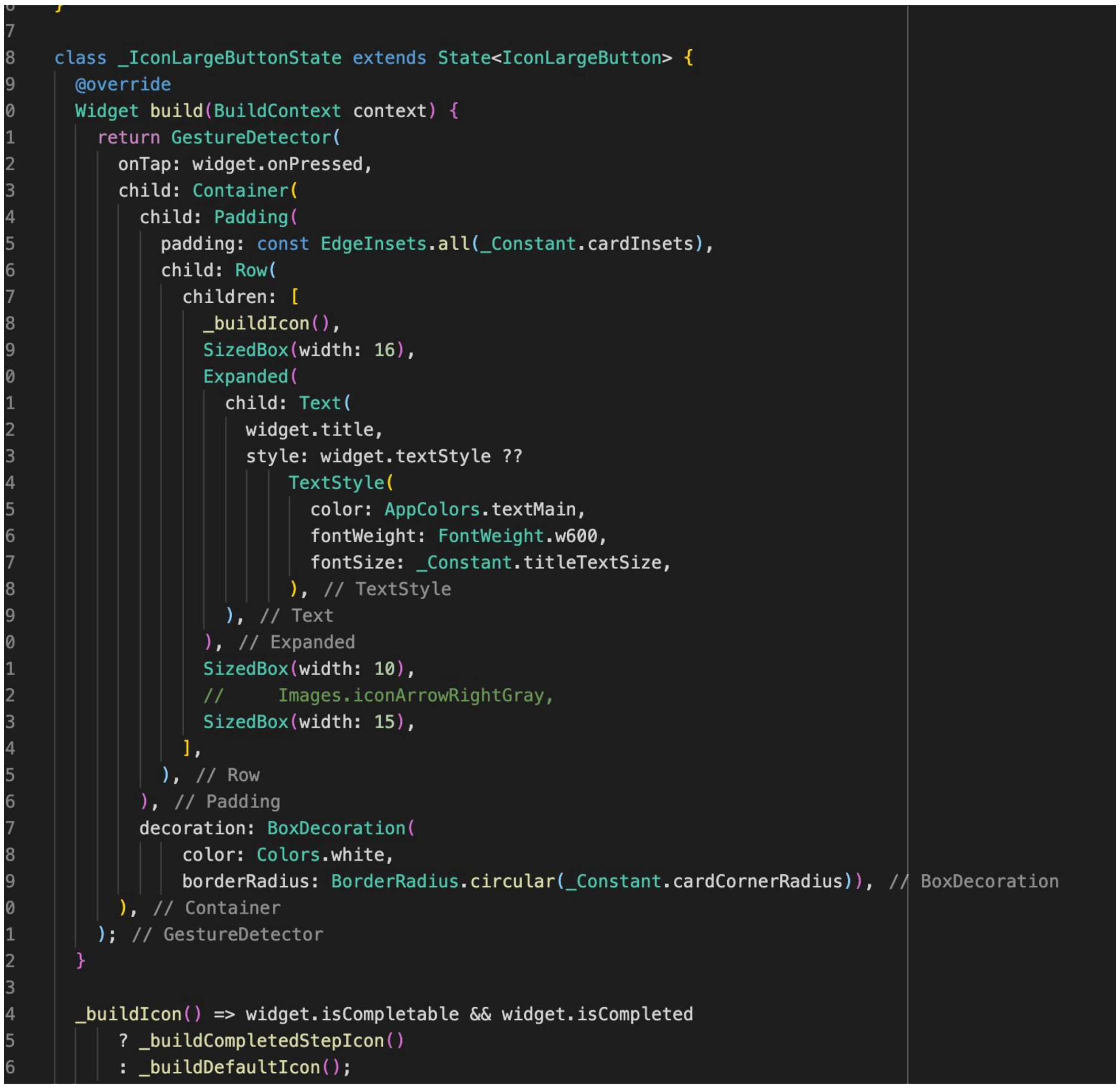
Figure 80: Groups Page Code Snippet 1



```
_searchBar() {
  return Container(
    decoration: BoxDecoration(
        color: Colors.white,
        borderRadius: BorderRadius.all(Radius.circular(12))), // BoxDecoration
    child: Padding(
      padding: const EdgeInsets.all(8.0),
      child: TextField(
        decoration: InputDecoration(hintText: 'Search groups..'),
        onChanged: (text) {
          text = text.toLowerCase();
          setState(() {
            _groupsForDisplay = _groups.where((note) {
              var noteTitle = note.title.toLowerCase();
              return noteTitle.contains(text);
            }).toList();
          });
        },
      ), // TextField
    ), // Padding
  ); // Container
}

_buildOverviewSection() {
  return Container(
      child: Column(
    children: [
      IconLargeButton(
        title: "Chem Class",
        onPressed: () {
          Navigator.pushNamed(context, Routes.chatScreen);
        },
        icon: Icon(Icons.group),
      ), // IconLargeButton
      SizedBox(height: AppDimensions.cardsSpacing),
      IconLargeButton(
        title: "Bio class",
        onPressed: null,
        icon: Icon(Icons.group),
      ), // IconLargeButton
      SizedBox(height: AppDimensions.cardsSpacing),
      IconLargeButton(
        title: "Comp Sci class",
        onPressed: null,
```

Figure 81: Groups Page Code Snippet 2

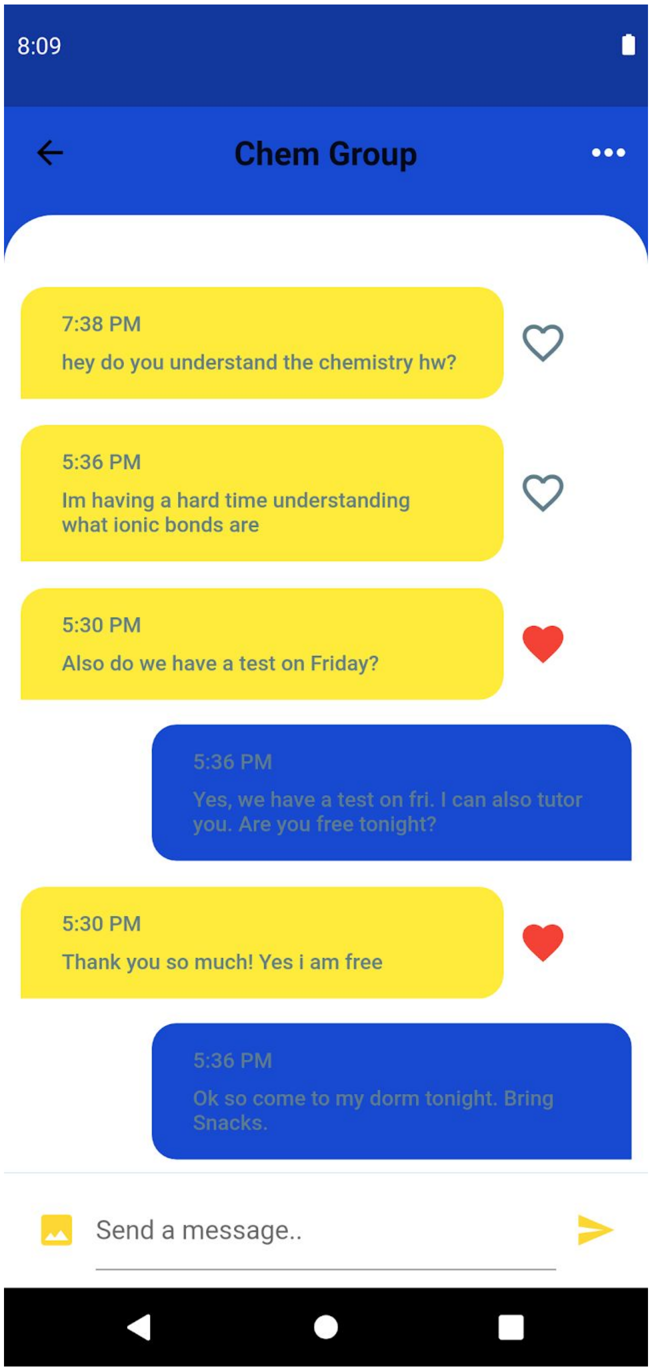
Above we have the screen implementation for the Groups page and below we have a screenshot of the Icon Large Button widget created specifically for this page that can be reused anywhere in the application.



```
class _IconLargeButtonState extends State<IconLargeButton> {
  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: widget.onPressed,
      child: Container(
        child: Padding(
          padding: const EdgeInsets.all(_Constant.cardInsets),
          child: Row(
            children: [
              _buildIcon(),
              SizedBox(width: 16),
              Expanded(
                child: Text(
                  widget.title,
                  style: widget.textStyle ??
                      TextStyle(
                        color: AppColors.textMain,
                        fontWeight: FontWeight.w600,
                        fontSize: _Constant.titleTextSize,
                      ), // TextStyle
                ), // Text
              ), // Expanded
              SizedBox(width: 10),
              //     Images.iconArrowRightGray,
              SizedBox(width: 15),
            ],
          ), // Row
        ), // Padding
        decoration: BoxDecoration(
            color: Colors.white,
            borderRadius: BorderRadius.circular(_Constant.cardCornerRadius)), // BoxDecoration
      ), // Container
    ); // GestureDetector
  }

  _buildIcon() => widget.isCompletable && widget.isCompleted
      ? _buildCompletedStepIcon()
      : _buildDefaultIcon();
```

Figure 82: Groups Page Code Snippet 3

3.4.17 Messenger Page



7:38 PM
hey do you understand the chemistry hw?

5:36 PM
Im having a hard time understanding what ionic bonds are

5:30 PM
Also do we have a test on Friday?

5:36 PM
Yes, we have a test on fri. I can also tutor you. Are you free tonight?

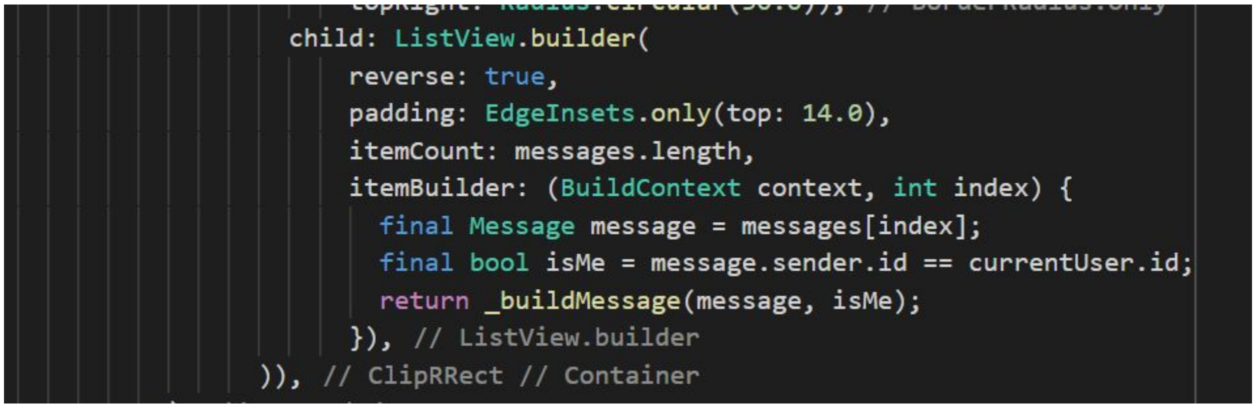
5:30 PM
Thank you so much! Yes i am free

5:36 PM
Ok so come to my dorm tonight. Bring Snacks.

Figure 83: Messenger Page

This is our messenger page. It is very similar to most popular messengers such as whatsapp and facebook messenger. The user can like messages, send photos and messages to the group they joined.

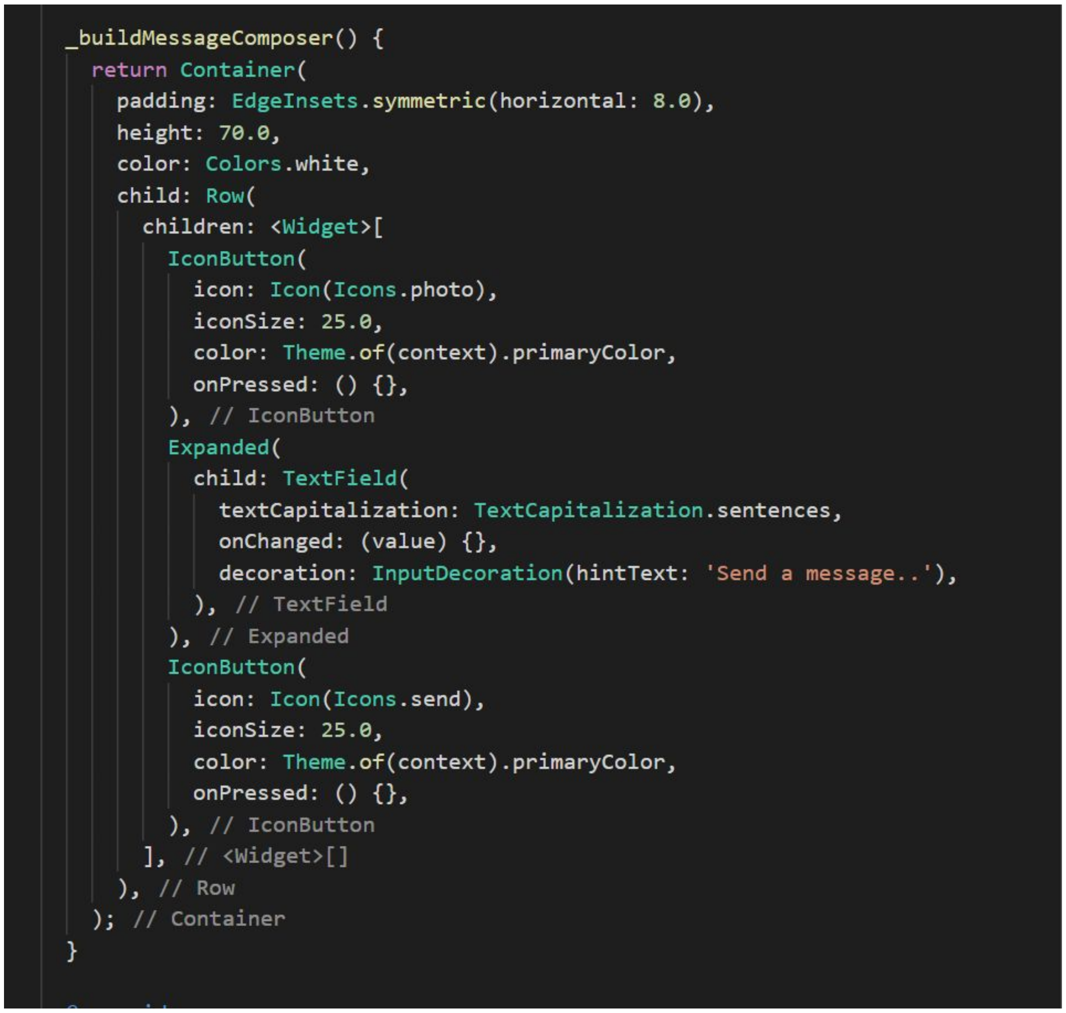
Below is a screenshot of the messenger app. This is the main functionality of the messenger where we pull the messages from the firebase database and display it to the receiving user.



```
          child: ListView.builder(
            reverse: true,
            padding: EdgeInsets.only(top: 14.0),
            itemCount: messages.length,
            itemBuilder: (BuildContext context, int index) {
              final Message message = messages[index];
              final bool isMe = message.sender.id == currentUser.id;
              return _buildMessage(message, isMe);
            }), // ListView.builder
        )), // ClipRRect // Container
```

Figure 84: Messenger Page Code Snippet 1

Below is the screenshot of the sending message feature. It is the UI that contains all the icons, suggestive text to send a message and the send button feature which uploads the text to the firebase database and can be pulled to the other user to see.



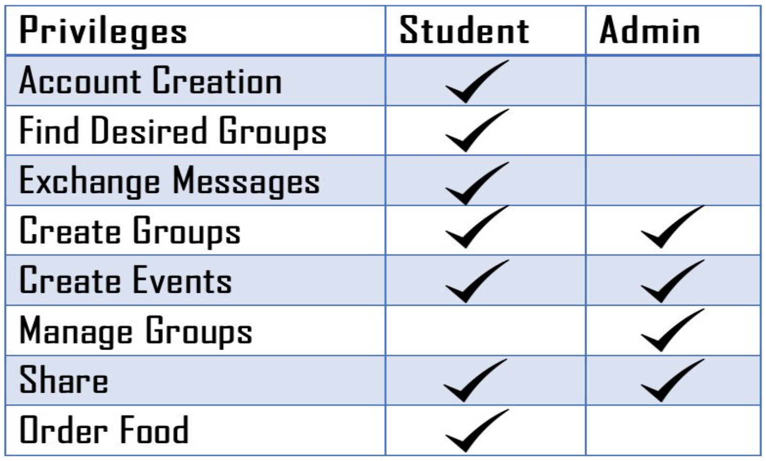
```
_buildMessageComposer() {
  return Container(
    padding: EdgeInsets.symmetric(horizontal: 8.0),
    height: 70.0,
    color: Colors.white,
    child: Row(
      children: <Widget>[
        IconButton(
          icon: Icon(Icons.photo),
          iconSize: 25.0,
          color: Theme.of(context).primaryColor,
          onPressed: () {},
        ), // IconButton
        Expanded(
          child: TextField(
            textCapitalization: TextCapitalization.sentences,
            onChanged: (value) {},
            decoration: InputDecoration(hintText: 'Send a message..'),
          ), // TextField
        ), // Expanded
        IconButton(
          icon: Icon(Icons.send),
          iconSize: 25.0,
          color: Theme.of(context).primaryColor,
          onPressed: () {},
        ), // IconButton
      ], // <Widget>[]
    ), // Row
  ); // Container
}
```

Figure 85: Messenger Page Code Snippet 2

3.4.18 Security

Our first method of security is the user authentication provided by our implementation of Firebase which allows us to verify users and have the users verify their email addresses to help us ensure that the users are from within the organization. Our second method of security involves the implementation of user and admin privileges which is elaborated on below.

3.4.18.1 Privilege Levels



Privileges	Student	Admin
Account Creation	✓	
Find Desired Groups	✓	
Exchange Messages	✓	
Create Groups	✓	✓
Create Events	✓	✓
Manage Groups		✓
Share	✓	✓
Order Food	✓	

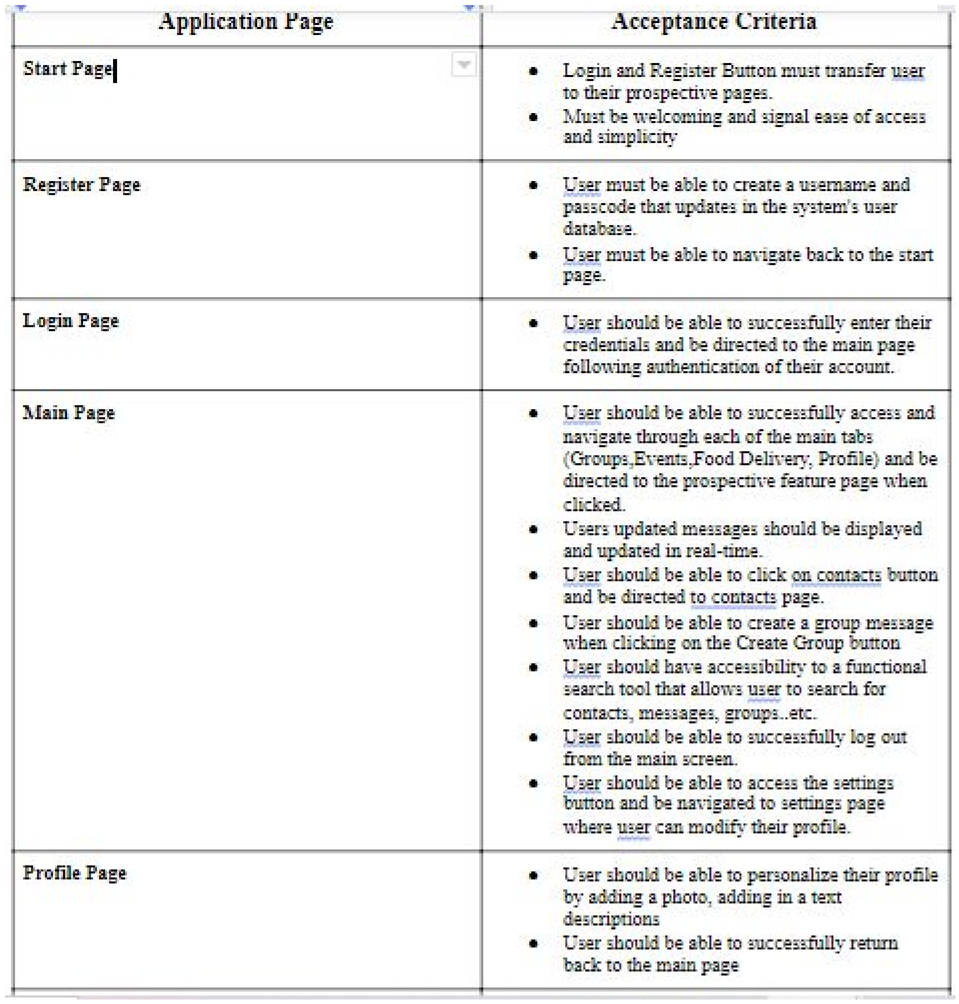
Figure 86: Privilege Levels Table

There are many different privileges to account for in this application. A lot of these privileges only the student/user have. They are the only one who can create accounts, find desired groups to join, exchange messages between other users, and order food to be delivered. There are some privileges that both the admin and users/students share such as create groups, create events, and share the application. When creating groups, the admin would create predetermined groups for each class, major, and organization such as clubs. When the user creates a group, it can be either a public group or private group, and for any reason. For creating events, the admin would create school wide events and when a user creates an event it would be public to everyone. Users can create an event for any reason such as creating an event on behalf of their organization. The privilege that only the admin has is managing groups. The admin can add or delete any group of their choice for a viable reason. This is done to protect the users from anything that could potentially happen with the creation of these groups.

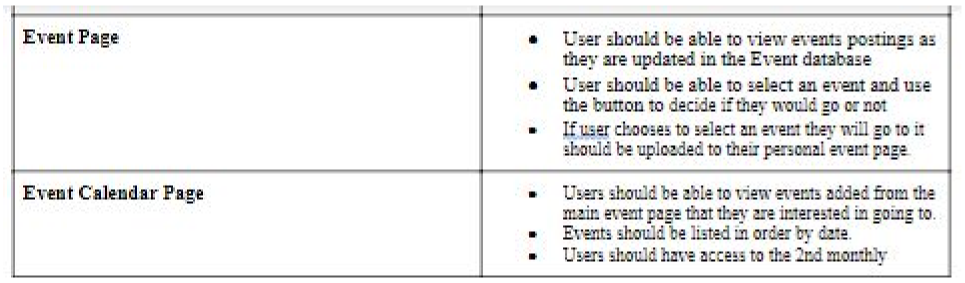
3.4.18.2 Testing the Application

There was a lot of testing that needed to be done to confirm that everything was running correctly. Following the black box testing method where we observed the results of testing the input and receiving an output. The first type of testing that was done was unit testing. The application was divided into four units, events, groups, food ordering, and profile. First it was determined that each individual unit was operating correctly. The next type of testing that was done was functional testing. Functional testing was done in order to confirm that each function, button, and toggle was operating correctly. The last type of testing that was done is system testing as the application was built for both iOS and android devices so it had to be tested in both environments.

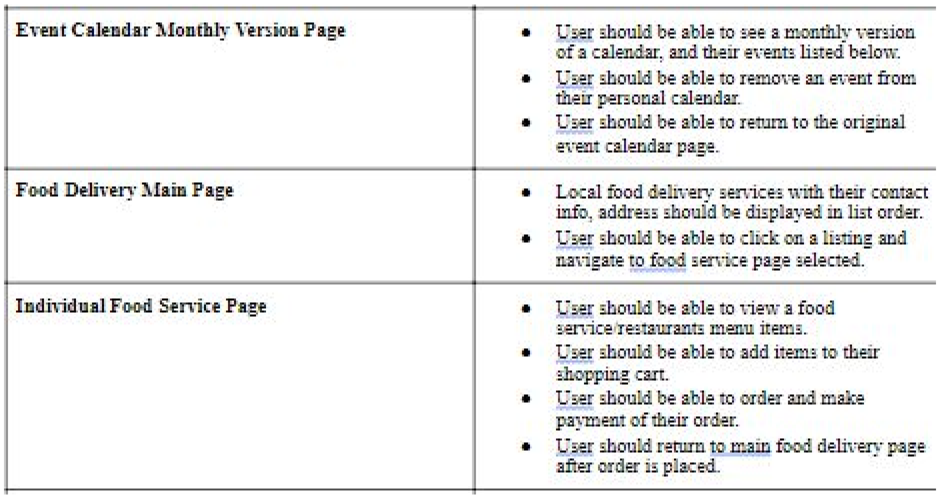
In addition to using the black box testing method to test functionality, we've also used the Acceptance Criteria to ensure that each feature and page reached our rubric of expectations. In this rubric we laid out all the things a user should be able to expect to do with each page, shown below:



Application Page	Acceptance Criteria
Start Page	• Login and Register Button must transfer user to their prospective pages. • Must be welcoming and signal ease of access and simplicity
Register Page	• User must be able to create a username and passcode that updates in the system's user database. • User must be able to navigate back to the start page.
Login Page	• User should be able to successfully enter their credentials and be directed to the main page following authentication of their account.
Main Page	• User should be able to successfully access and navigate through each of the main tabs (Groups,Events,Food Delivery, Profile) and be directed to the prospective feature page when clicked. • Users updated messages should be displayed and updated in real-time. • User should be able to click on contacts button and be directed to contacts page. • User should be able to create a group message when clicking on the Create Group button • User should have accessibility to a functional search tool that allows user to search for contacts, messages, groups..etc. • User should be able to successfully log out from the main screen. • User should be able to access the settings button and be navigated to settings page where user can modify their profile.
Profile Page	• User should be able to personalize their profile by adding a photo, adding in a text descriptions • User should be able to successfully return back to the main page



Event Page	• User should be able to view events postings as they are updated in the Event database • User should be able to select an event and use the button to decide if they would go or not • If user chooses to select an event they will go to it should be uploaded to their personal event page.
Event Calendar Page	• Users should be able to view events added from the main event page that they are interested in going to. • Events should be listed in order by date. • Users should have access to the 2nd monthly



Event Calendar Monthly Version Page	• User should be able to see a monthly version of a calendar, and their events listed below. • User should be able to remove an event from their personal calendar. • User should be able to return to the original event calendar page.
Food Delivery Main Page	• Local food delivery services with their contact info, address should be displayed in list order. • User should be able to click on a listing and navigate to food service page selected.
Individual Food Service Page	• User should be able to view a food service/restaurants menu items. • User should be able to add items to their shopping cart. • User should be able to order and make payment of their order. • User should return to main food delivery page after order is placed.

Figure 87: Acceptance Criteria

4.0 Conclusion

4.1 Testing Results

Our first tests failed because of merging issues, when the team tried merging all branches after a long time of working on separate branches. Fortunately, there were not that many conflicts, but it was difficult and required attention to detail and many tries. We managed, therefore , to make it work and link the app together. Testing initially was done separately, through AC's, on our own devices (the programming team- Almir, Gabriel and Stefan), and then again but with the whole app merged. Several devices were used and tests mostly succeeded.

4.2 Future Projected Improvements

In the future, we hope to expand this application so that it is not just used by NYIT students but other organizations as well. We hope that it can be used by other universities and companies as well as the app is very versatile. We also hope to add customizable features and options that the organizations can use based on their specific needs. We also hope to do more long term testing to determine any other changes that it may need in order to make it user friendly and easy to use. We would also like to link the group functionality to the database. Also in order to make the application more efficient and up to date we would like to develop a couple of algorithms, one for users to be able to find other users based on their interests and on to find free restaurant delivery. These algorithms will help users connect with each other as well as keep the delivery page up to date. We also hope to develop analytic and interface for admins in order to determine if the application is widely used in their organization and what customizations they would need to make in order to make it more efficient for their use.

4.3 Learning Experiences

4.3.1 Stefan

Connect NY has been an interesting project that I have invested a healthy amount of time and energy into, with a result that I am very content with. I have learned, throughout this experience, to better manage my time and communicate with colleagues across the ocean and in a different time zone. I learned to meet deadlines better and prepare for them in time, and to coordinate a team while trying to do so. I also learned to create an app architecture and work separately on it as a programming team, through Github. I learned to merge codes and how to correctly work on separate branches of the same project. It was an exciting experience that we will definitely continue to work on, as there is much room for improvement.

4.3.2 Jennifer

This group project has helped build on my teamwork skills which is crucial for real-world experiences when working within a company, since most projects are done with multiple people. I also got to learn a little more about the Flutter SDK and how it applies to the app development process and how it was used to bring our idea to life. I have been able to focus on improving my project preparation and analysis skills through diagramming which is a crucial step in the planning process of an application and found that planning and analysis is something that excites me and might possibly want to focus on in my future career in the tech field, maybe through the role of project management. I was also able to focus on documentation since one of my roles in this project was to focus on the organization and content of the project report and presentation. I’ve also realized more and more how hard it can be to balance big projects while trying to maintain my other academic and work obligations at the same time.

4.3.3 Samiha

This project helped me improve my team collaboration skills and taught me a lot about the app development process. Unlike the rest of my group, I am a student from the Old Westbury campus while the rest of them are from the Manhattan campus so I had never worked with any of them in the past. I learned to adapt to everyone's different work styles and collaboration methods in a short amount of time as well as learn about the app development process. I learned about all the different diagrams that needed to be made in order to thoroughly plan and design the application. I also learned how to properly document a large scale project such as this and how it is important to be consistent and balance it with all my other academic work.

4.3.4 Almir

I spent this summer learning flutter. and with this project I expanded my knowledge of flutter and dart. I learned how to work in a team using flutter and dart. During the course of this project I was introduced to Jira, which is a software management tool. With Jira came new terminologies and concepts to me such as stories, epics and tasks. For this project, I was responsible for the restaurant section of the app. In the middle of my coding, I encountered a problem I did not experience before. When I ran the android emulator, it would not load the gradle. After hours of searching on the internet for a solution, uninstalling and reinstalling VS Code, and flutter, I finally found the solution. My antivirus updated and did not allow the emulator to connect to the internet. It was such a simple solution to a complex problem. So I learned to check the obvious before tackling the more complex problem shooting options. During the Github merging process, I learned how to edit and integrate my code with my team. I feel like the github merging learning process was an important learning experience as it can help me in the future when I code with a team at a company. It was an overall interesting experience.

4.3.5 Gabriel

In my previous semester, I had to make an Android app for the first time using Kotlin and Java as my programming language, and SQLite as my database. It was very complicated and the code was messy and incohesive. For this project, I had to learn how to use Dart and Flutter as our API for the app and Google Firebase as our database. Due to how much easier and intuitive Flutter and Firebase are to implement and read, I spent more time actually designing and implementing the pages rather than trying to figure out how to do basic things like with Kotlin and SQLite. One of the challenges I have had to deal with is with my inexperience with not only the Github extension for VS Code, but with Github itself. I was not too familiar with it, and ran into a setback when pushing and pulling code. There was a problem where I couldn’t start the app and none of the solutions I could think of worked, so I decided to try and pull my old code from Github when the app was working and push my new code to a different branch. I figured out the problem was the emulator, not the code, but when I tried to pull the new code, I accidentally merged the branches where my old code overrid my new code and lost a lot of progress. Since then I became more proficient with using Github and the VS Code extension where this problem did not happen again.

5.0 References

Woodgate, R. (2019, July 16). What Is Slack, and Why Do People Love It? Retrieved December 28, 2020, from https://www.howtogeek.com/428046/what-is-slack-and-why-do-people-love-it/

Thomas, G. (2020, July 24). What is Flutter and Why You Should Learn it in 2020. Retrieved December 28, 2020, from https://www.freecodecamp.org/news/what-is-flutter-and-why-you-should-learn-it-in-2020/

Johnson, D. (2019, September 06). 'What is LinkedIn?': A beginner's guide to the popular professional networking and career development site. Retrieved December 28, 2020, from https://www.businessinsider.com/what-is-linkedin

Flutter & Dart - The Complete Guide [2021 Edition]. (2020, December 23). Retrieved December 28, 2020, from https://www.udemy.com/course/learn-flutter-dart-to-build-ios-android-apps/

Delfino, D. (2020, March 26). 'What is Discord?': Everything you need to know about the popular group-chatting platform. Retrieved December 28, 2020, from https://www.businessinsider.com/what-is-discord

Get Started on Web | Firebase. (n.d.). Retrieved December 28, 2020, from https://firebase.google.com/docs/storage/web/start