Independent Study

Introduction

Dining out is a very simple and effective way to socilize. However, it is not always easy to choose a restaurant that suits everyone's needs, especially when you are in a group with different dietary restrictions, preferences, and a plethora of unfamiliar choices. This is where my independent study comes in - it addresses the problem of choosing a restaurant in a group by making the process of selecting a restaurant more efficient and enjoyable.

My inspiration for creating this app came from a frustrating experience I had while visiting Charlotte with a group of 5. We spent 20 minutes searching and calling around for a nearby restaurant that had gluten-free and vegetarian options, was open, and had a wait time of less than half an hour. We ended up settling on a chain restaurant, feeling unsatisfied with the choice we made. This experience made me realize how frustrating this problem can be, and motivated me to try to create a simple solution.

By providing a solution to this problem, the app not enhances the dining experience and saves time. Furthermore, it's important to solve this problem because it could lead to a very popular app that people recommend to their friends. Everyone in the group needs to have the app, so it has the potential to spread quickly. By creating an app that allows users to vote on a list of restaurants, my hope is to provide a fun, easy, and effective solution to this problem.

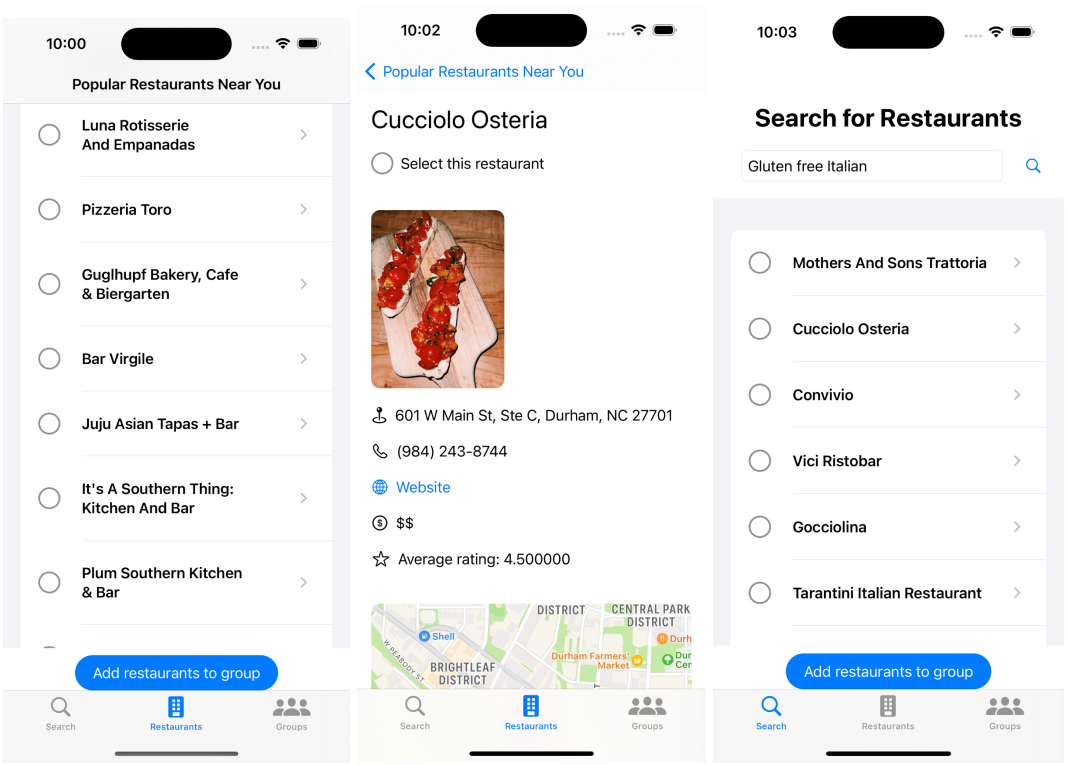
The app differs from existing restaurant-related apps in a few key ways. Restaurant Roulette, for example, allows users to spin a roulette with a list of restaurants but does not allow for multiple users to participate in the selection process. Meanwhile, Google Maps, Yelp, and

Apple Maps do not provide a way for users to vote on a list of restaurants. Dischcovery has a similar design to my wireframe (swiping to select restaurants), but it lacks a group feature.

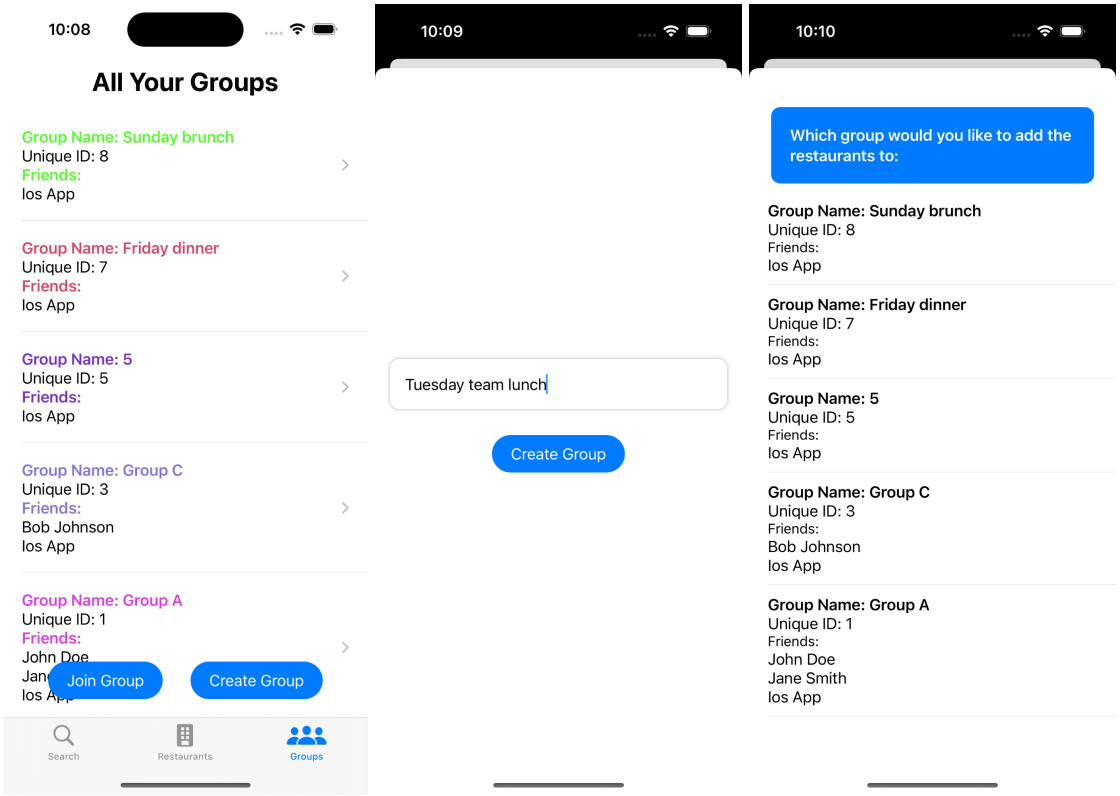
The main goal of this independent study is to create a functional prototype of the app so that it can be tested by potential users. This testing will allow me to determine if the concept is viable and if it solves the problem of choosing where to go to eat when in a group.

State of the App

The current state of the app's functionality is that users can search for restaurants or browse popular restaurants near them. The user can create a group or join an existing group using a unique code. Once they find restaurants they like, they can add them to a group. Each member of the group can then vote on the restaurants that are part of the group, and the total results are displayed.

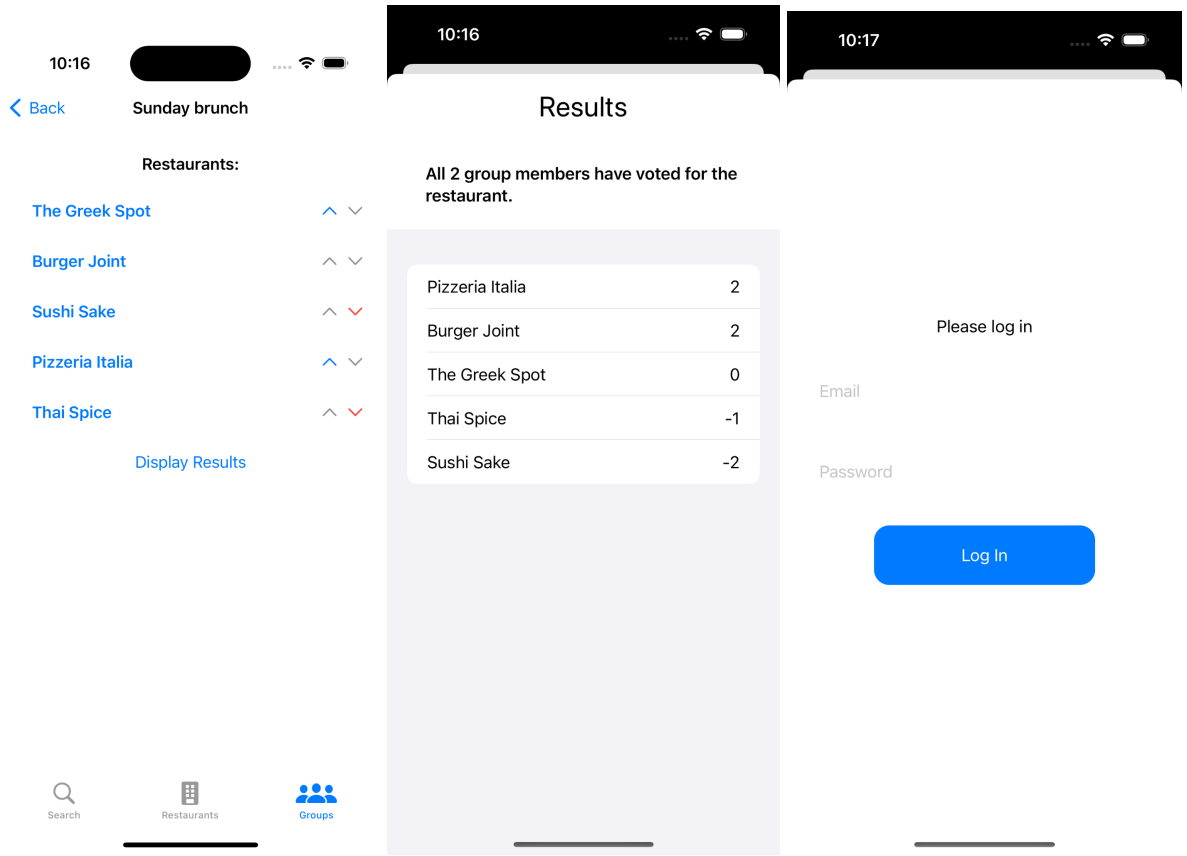


On the left screenshot, you can see the "Popular Near You" page, which displays a list of nearby restaurants. In the middle screenshot, you can see what it looks like when a user clicks on a restaurant. The app shows a picture of the restaurant, a link to its website, and a map with a pin on the location. Finally, the third screenshot shows the search page, where users can enter their search criteria and find specific restaurants that meet their needs.



On the left screenshot, you can see the "All Your Groups" page which displays some information about each group such as the name, group ID, and other group members. The page also features two buttons that allow the user to either join an existing group or create a new one. Upon clicking either button, a sheet appears prompting the user to enter the relevant information such as the group ID to join or the name of the group they want to create. In the middle screenshot, you can see an example of a user creating a new group. Finally, the right

screenshot shows how a user can select restaurants and add them to a group by choosing from a list of existing groups.



The screenshot on the left displays the list of restaurants associated with a group when the user clicks on the group. Additionally, it illustrates the voting system with an upvote (yes, +1) and downvote (no, -1) button along with an option to remain neutral (indifferent, 0). In the middle screenshot, the result display shows a message indicating whether everyone in the group has finished voting or lists the names of the individuals who still need to vote. It also ranks the most popular restaurant and displays their names. Finally, the screenshot on the right shows the login page of the application.

There are still some known bugs in the app. For instance, users cannot access the groups if they are not logged in, as it throws an error. If a user tries to join a group that they are already part of, the app shows a generic server error, but they can retry to join. The up or down vote on the restaurants don’t necessarily reflect the most up to date data from the server.

There are also some limitations. At present, the app lacks a registration system for users. There is no way to remove people from a group that you created. Furthermore, there is no way to remove restaurants from a group either. Prior to conducting a more extensive testing with a group of users, it would be necessary to incorporate this feature into the app.

Assumptions were made about how users will interact with the app. One such assumption is that users have a way outside the app to communicate, as there is no in-app messaging system. Also, the way to join a group is by typing the ID, and there is currently no way to send it. This could be implemented in the future, such as sending a joining link or ID over messages or even a QR code.

The original vision for the app's functionality was to allow users to swipe left and right on each restaurant. This could be helpful to users who are more visually inclined and want to see more pictures of the restaurant or other details. I thought this would be an easy way for several people to go through a list of restaurants and see some information about each. This may be implemented in future.

Based on feedback from testers, the app's functionality has been refined and improved. For instance, some users suggested that the app could benefit from more visual appeal, leading to the implementation of a color-coded system for the groups page. This feature displays each

group name and its members in distinct colors to enhance the visual appeal and make it easier to differentiate between different groups. It is now more fun.

App Design

The aim of the independent study was to develop a functional prototype that could be tested on potential users to gauge their interest in the app. Thus, the focus was on achieving a balance between creating a working prototype and writing code that could be used in the final version of the app, should it be deployed.

The primary goal of the architecture design was to create a scalable app that could handle multiple users voting on restaurants in real-time. To achieve this, I structured the app's code to have all the data on the server-side, including restaurants, groups, user data, reviews, ratings, and results. The app makes GET and POST requests to the server to retrieve and send data that the user has inputted or changed. I separated the client/server logic in this way to ensure that multiple users could vote on restaurants at the same time, without any conflicts.

To develop the backend of the app, I used Ruby on Rails, as it made spinning up a prototype very easy. Another benefit was that the ORM managed a lot of the SQL queries. For the frontend, I used SwiftUI for the iPhone app, as this is what Apple recommends for developing iOS apps.

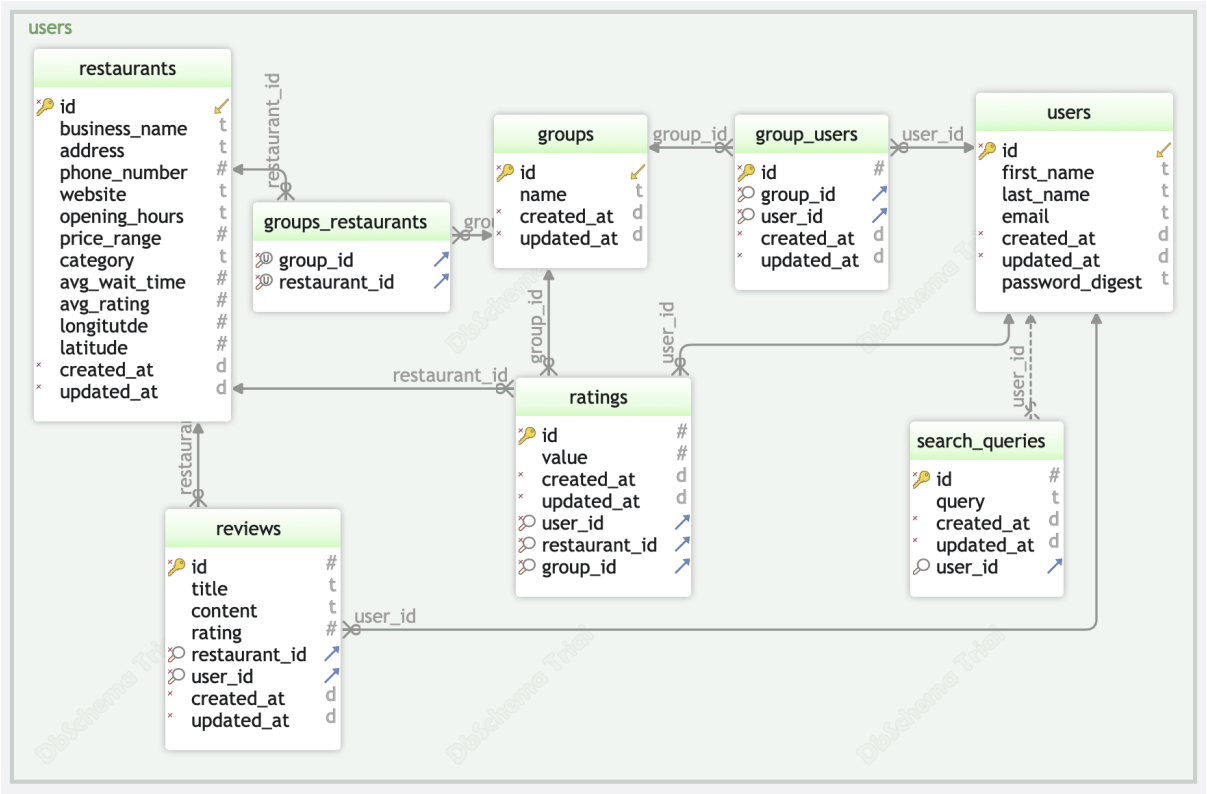
In terms of design patterns, I used the Model-View-Controller (MVC) pattern to structure the code. Each model/table had its own controller, and some of the join tables also had a controller to facilitate actions such as a user joining a group. The view consisted of either

sending json to the Swift app or sending HTML to the browser. The browser allowed me to test that Rails works before trying to implement it in Swift.

The Swift app utilized a view-loader-apiclient design pattern with a separate loader, API client, and model for each GET or POST request. The models were used to pass data between the different components ensuring consistency throughout the app. The loader managed the data displayed in the view, handled user input, and sent the appropriate data to the API client, while the API client performed the necessary API calls, decoded the response JSON, and made it accessible to the loader. This modular approach allowed for efficient management of the app's features and functionalities.

To test and debug the app's code, I used the Xcode debugger and version control systems like Git and Gitlab for tracking changes and rolling back. I used the built-in Rails testing functionality to ensure that the backend worked. Lastly, I utilized curl commands to verify proper network communication between the app and the backend.

Database Schema



The main tables in the database are the restaurants and users table and these were the two I started with. The groups table is also fundamental to the main function of the app. Groups create a central reference from which join tables construct the ability for users to vote on restaurants.

There are also several join tables in the schema which connect the different models together. For example, the group_restaurants table is used to establish a many-to-many relationship between groups and restaurants, allowing users to vote on restaurants within a particular group. The user_groups table is another join table which connects users and groups, allowing users to join and leave groups as they desire.

The primary goals when designing the database schema were to create a structure that would allow for efficient data retrieval and manipulation. Challenges encountered during the design

process included determining when to use a join table versus when to use many-to-many relationships for some of the models. In general, I used join tables when additional attributes needed to be stored for the relationship between two models or when a controller was needed to create a specific resource, such as when creating a group_user resource to allow a user to join an existing group. On the other hand, I used many-to-many relationships when no additional attributes were required.

Learning

1. MVC and nesting resources

One of the most important technical lessons I learned during the development of my app was the Model-View-Controller (MVC) design pattern used in Ruby on Rails. I started by creating the restaurant scaffold with all the appropriate attributes by following the Code Academy tutorial.

One challenge I faced early on was implementing a review model where users could leave restaurant ratings. To achieve this, I needed to create a nested resource for both users and restaurants. I followed a Digital Ocean tutorial that provided a step-by-step guide on how to create nested resources for a Rails application. This tutorial was very helpful and provided me with a solid understanding of how to create nested resources for related models. This concept was particularly helpful during the implementation of groups and the associated actions that the model needed to handle. For instance, it assisted with adding restaurants or users to a group.

2. How to do a GET request

During my independent study, I learned the importance of having a solid foundation for API client development. This was particularly evident during the first sprint involving SwiftUI, where the goal was to pull restaurant data from the server. To achieve this, I first had to learn how to use curl to make a GET request to the localhost. I then had to configure rails to listen to any request from the network by binding the rails server to my laptop's local IP address.

Once the server was set up to receive requests, I had to learn how to make a GET request in Swift. I reached out to Assistant Professor Jon Phillips, who was teaching CS 207 - iOS development, and gained access to his Sakai to look at the lectures where he taught how to do GET requests. I followed his lectures and implemented the code accordingly. I created custom models to parse the JSON data sent by the server for restaurant information. In addition, I learned how to implement an API client to send the appropriate GET requests and store the session in a variable. The loader was responsible for calling the API client and managing the returned data. It made sure that the data from the GET request was decoded into a restaurant model and then returned to the view that was waiting on the loader to complete the request.

3. How to send a POST request

During the sprint following my successful implementation of a functional GET request, my goal was creating a functional POST request in Swift. Specifically, I aimed to implement a search feature where users could send in a search query and Rails would search over restaurant names for similar matches.

The course lectures did not cover all of the details needed to implement a POST request. First, I had to create a variable that would hold the user's search query from the search bar. Then, I had to figure out how to send this query along with the request once the user pressed the search button. This required passing information from the view to the API client via the loader. In particular, the loader passed the search query to the API client, which then had to convert the string data to JSON and send it along as data to the request using an appropriate decodable model.

4. How to use bindings in Swift views

During one of the sprints the aim was to allow users to add restaurants to a group and then vote on them. I had to learn what bindings in swift were to implement this. Bindings in Swift allow two-way communication between a view and its underlying data, which is essential for implementing features like adding restaurants to a group. In this case, I needed to pass the list of selected restaurant IDs from the list view to the detail view so that the user could add them to a group. Using bindings, I was able to create a state variable that would hold the list of selected restaurant IDs and pass it as a binding to the detail view. This way, any changes made to the list in the detail view would be reflected in the list view as well.

5. Reload group page after user joins or creates a group

During testing, I observed that many users were confused when they created a group but it was not presented after dismissing the sheet. After some investigation, I discovered that the bug was caused by the loader of the group not resetting once the sheet was removed. The loader was still running in the background, preventing the newly created group from being displayed. To fix this issue, I had to learn about Swift modifiers such as onDisappear and onChange. These modifiers allowed me to track the states of the sheet and other variables,

and know when a group was successfully created. Once I had this information, I could fetch the relevant data from the server and present it to the user.

6. Seeing who hasn't voted yet in display results page

Another valuable piece of feedback I received was that users wanted to know who in their group had not yet voted, so they could remind them to vote. Implementing this feature required changes to the Rails backend, as I needed to create a function that could retrieve users who had already voted and then compute the set difference with the total number of users in the group to determine who had not voted yet. I was able to accomplish this by retrieving the names of potential users who had not yet voted and displaying them in the app.

Future Work

To make the app a viable product, there are several steps that need to be taken. One of the most important ones is to include a user registration system. This will allow users to create an account, save their preferences and access additional features such as the ability to bookmark their favorite restaurants. Another key aspect is to provide restaurants with access to the app so they can manage their listings, update their menus, and provide more detailed information about their offerings. This will help increase engagement and improve the quality of the data.

In terms of feedback from users and beta testers, there are several areas that need improvement. One area that was highlighted was the restaurant detail view. While it works well for a prototype, it needs to be more fleshed out and provide more information such as photos, opening hours, and menus. The search feature was also mentioned as an area for improvement. One potential solution to make search more user-friendly would be to leverage advances in natural language processing to allow users to search using natural language and

match them with relevant restaurants. Additionally, beta testers have suggested adding a deals feature that could help prevent food waste by partnering with restaurants that are about to close and offering heavy discounts to users.

The independent study successfully achieved its goal of creating a prototype that users could test and assessing interest in the product.