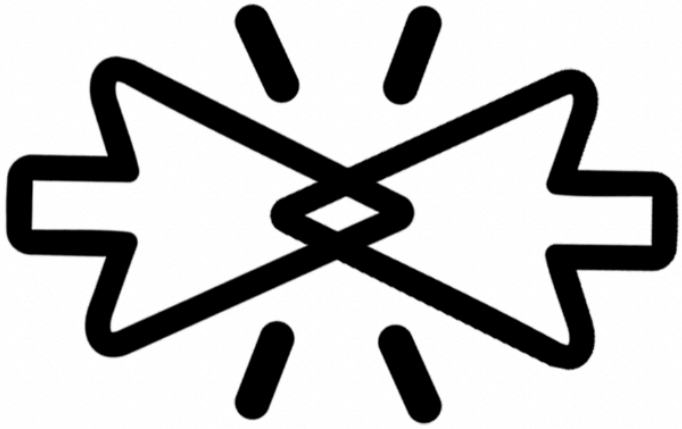


Cliq - Senior Project Report

A new way to make platonic connections

Chris Valente
21/01/2024

Cliq - Senior Project

21.01.2024

Chris Valente

cvalent1@ramapo.edu

R#: R00745638

Introduction - an overview of Cliq

Cliq is a matchmaking web application that serves as an easy way for users to make *platonic matches* with other users on the platform. Based on research studies done by the CDC, and Cigna, on average, people are lonelier than ever; especially in the younger age groups, and especially after COVID. *~60%* of adults in the US have reported feeling lonely on a regular basis, and *49%* of Americans have reported having fewer than three close friends. The concept of Cliq arose with an attempt to take the tried and true concept of dating apps such as Tinder, to create an app that focuses less on the intimate aspect of these apps, and more on the connection aspect. In other words, Cliq seeks out to use the logic of other proven methods of matching successful online encounters, to redirect efforts to create another way for users to make *platonic connections*. On Cliq, users are encouraged to create and customize their own profile, which culminates into a *card,* which is tailored accordingly to one's individual likes and interests, and displays it to other users

in a condensed form factor for them to observe in a digestible way. The core functionality of the site exists on the dashboard page, which highlights the *"swipe queue"*, where users are able to view a selection of cards belonging to other users, one at a time, (within a user specified radius), and can swipe *right* (if they are intrigued by the other users profile) or *left* (if they aren't interested in the other users profile) with the hopes of making matches with those who have mutually caught each other's eye. If a match is made, the two users will be allowed to connect and communicate within the app's chat page, where hopefully these matches will cultivate friendships; or in other words, hopefully, they *click*.

Goal of the Project

As mentioned above, the idea behind Cliq is to (theoretically) provide another avenue for making new connections with others. To understand the idea of Cliq, it is prerequisite to understand apps such as *Tinder*. The photos on this page can give a solid idea of what a *Tinder* card looks like *(see bottom image)* - at first glance, a card is a photo that has basic information about the user overlaid on top of it (name, age). If you tap on these cards, it opens them up to a different view *(see the image to the left)*, where you are able to see some more basic information about the

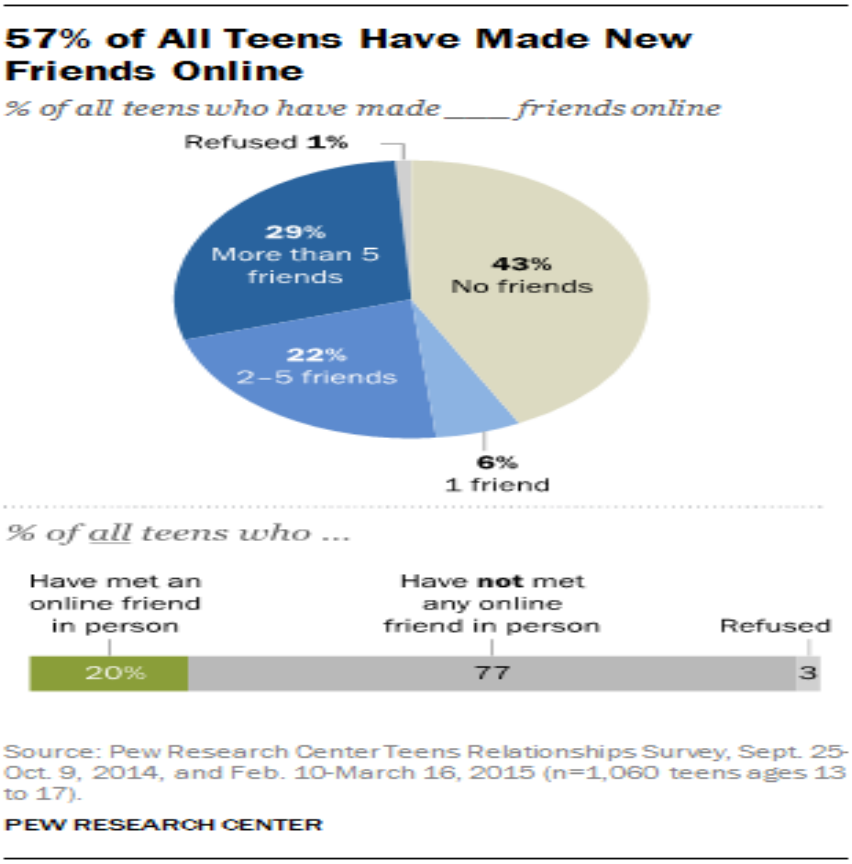
user - including job/school, distance away, and bio. These are important features to include, and Cliq has decided to use many of these tried and true attributes to profiles as well. The main focus here, as seen in the image below, is the picture. The way Tinder works, users submit a gallery of photos of themselves, which can be interacted with within the expanded profile view, and based on how they look, users will choose to engage on attractive looking users, and disengage those who don't fit the standards they're looking for. And since Tinder is a dating app, this is a totally common way of using the platform. In Cliq cards, which will be displayed later, looks/attraction is not the end goal here. Users are allowed a profile picture, which gets displayed, but the focus is *not* on images, but rather the content of the profile - after all, one doesn't necessarily choose friends based on their attractiveness. A lesson to take from Tinder, though, is that realistically, a profile card has a very short amount of time to win someone over. With that being said, decisions were made to try to condense information, and try to give users as much content as possible on one card, without the need for expanding views into images, or anything of that nature. Everything basic that anyone would want to know about a user, could theoretically fit into one Cliq card.

From a *personal perspective* - this project was very interesting to me, and the idea came very quick. On the internet today, social media runs the show. Everyone always talks about *the next big thing*, and a stereotypical dialogue that often gets joked about, is that Computer Science students always get approached by others with *app ideas*, and *"could you make this?"* - it made me curious about what it actually took to create a platform like this at a reasonable scale, so I knew I wanted to make some sort of social app. I've also said previously that creating a large web application was a goal of mine before I graduated, so this was quite literally the perfect time to do so. Additionally, in a previous semester for my *Data Science minor*, I've created a final project about *Tinder*, and how users interact with the platform, from a data point of view.

I wrote a scraper that leveraged the *Tinder API* to get select public data about a mixed assortment of profile data of both male and female profiles nearby, and wanted to use the data to observe how users as an aggregate actually interacted with these platforms, and if it differed at all by gender, with the goal of hopefully finding some interesting key takeaways. This project ended up being one of my favorites that I've done at Ramapo, and it actually ended up being key in helping me obtain my current internship. All of that to say, since working on that project, I was quite familiar with the type of data that these profiles contain, and I became heavily intrigued in the concept of designing a similar platform, since I have already done quite a bit of research on *Tinder* for this other project. I was curious if I could implement something like this by myself. Thus, Cliq.

Target Audience

Naturally, it's easy to imagine a platform like this being received in a more positive way the younger you look, towards those who are comfortable with the idea of making "internet friends". In a study that was done ~9 years ago, it was observed that 57% of surveyed teenagers have made new friends via online mediums; 29% of these have made more than 5 friends. With the way things have evolved since then, it's easy to imagine that in the time that has passed, these numbers have increased quite a bit. The gist of these online friendships is that they usually occur through video games, or social media, where odds are, your new friend could be thousands of miles away. It is for this reason, why one could read between the lines and understand why only 20% of these friendships have transferred to an "in person" medium. It's quite easy to make friends, but it isn't quite so easy to take flights, or drive extremely long distances to spend time with each other. Cliq offers users the ability to find friends wherever they want; by allowing users to search



based on a given radius, there is a greater likelihood for an increased *in-person friendship conversion rate*, so to speak, where users can find potential friends from as close to them as the next house over, that maybe they've never had the chance to meet before. These are the encounters that Cliq would love to capitalize on - removing the *chance* that is required to encounter others in the right spot. With all this being said, the target audience is definitely for a younger audience in mind, but can truly be used by *anyone* - there should be no limit on who could use the platform, and the UI is simple enough for anyone of any age to pick up easily.

Tools Used - stack and key features

Cliq has an interesting stack of tools holding it together. Web development has a vast amount of resources that one can use for projects, and deciding between certain tools was difficult; there were often times in development where one tool would finally be selected, just to be replaced the next day - lots of hours were wasted by integrating tools that weren't used in the final product. By the end, though, with this final version of the Cliq proof of concept, the following are tools that were used to create and maintain the application.

- ➔ ***Typescript/Javascript***
  - ◆ The language of choice - *Typescript*. For the uninitiated, *Typescript* is a superset of *Javascript* which allows for static typing, classes, and interfaces. These additional features add a sort of robustness to *Javascript*, making it perhaps easier to follow along in larger projects if utilized correctly. This means it is backwards compatible with *Javascript*, as it is essentially compiled

into such. There are instances of both *Typescript* and *Javascript* being used in the project, but most code is written in *Typescript* due to these reasons (and desire to learn it, of course)

➔ ***TailwindCSS***

- As a rule of thumb, if there is a way to make things easier along the way, it will be attempted, at the very least. As said before, web development is *vast*, and the tools are more or less infinite, so why not take advantage of everything it has? *TailwindCSS* is an alternative to regular CSS, that works by allowing CSS to render within custom class names, as opposed to being explicitly written in a stylesheet. Alternatively, *Tailwind* provides the functionality of several lines of CSS and abstracts it to just one class name, that contains different bits of styling instructions separated by spaces. As someone with little CSS experience I felt that referring to *Tailwind* was a better use of time vs. writing potentially thousands of lines of CSS - and it was definitely the right idea.

➔ ***Next.JS***

- There are a *lot* of web frameworks to choose from - *Django, Angular, React, Vue, etc.* The list can go on forever. One of the frameworks I've heard the most about going into the project was *React*. I was pretty keen on using this throughout production, as it was always something I wanted to try. In doing my research, I discovered that it was very popular at the moment to use

Next.js, which is built on top of *React*, which provides additional functionality without having to utilize external libraries/dependencies, and operates slightly differently. For example, *Next.js* uses **server-side rendering** - this is when applications load on the server side, after being generated in advance, as opposed to rendering dynamically on the browser. This theoretically leads to quicker load times, but makes things slightly more involved from a development front.

Similar to how people say *everything in Python is an object*, everything in React is a *component*. *Components* are reusable bits of code that return custom HTML, as opposed to other types of data (ex. integers, strings). This allows you to create reusable HTML, like a very specifically styled button, for example, that can be used anywhere in the codebase by just adding `<CustomButton/>` into the page HTML. Components also utilize *props*, as a function argument, which are used to be passed between components to provide information in a streamlined way (I learned about these properly way too late). To put everything together, due to the way *Next.js* renders information, we wouldn't be able to call a server function on the client side (ex. database read to get user information). So, we would make this call on the server side (in a routes `page.tsx`), and pass our read information to the client component, so we can use this information on the client side.

➔ ***Xata***

- *Xata* was an interesting choice - I came across *Xata* during the research stage, after looking at many different database solutions. Admittedly, at this point, I'd have gone through maybe two or three different platforms, and I responded well to the logic of integrating *Xata* within the codebase, so I kept it. *Xata* is a *serverless* database platform powered by PostgreSQL. In this context, serverless essentially means that the demand for managing the database (configuration, scaling) is erased, as *Xata* handles all of these things automatically. The core mission that *Xata* has is to *simplify the way developers work with data* - and again, that message will always resonate with me.

➔ ***Clerk***

- In the scope of this project, *Clerk* was a lifesaver. From the beginning, one of the *must-have* features was acquiring a means for third-party authentication. *Clerk* allows for the *Sign-In* and *Sign-Up* components to work, by allowing users to sign in/up with either third party authentication (as mentioned above), or with a classic user/password combination. This database filled with users is separate from the *Xata* database as described in the previous section, but they are linked together using the *Clerk* ***UserId*** as a foreign key in our *Xata* table - which will be demonstrated in a future section.

  Going into the project, I was prepared to figure out how to implement this myself, but after searching how to integrate this feature when I was in the research phase of developing Cliq, I quickly discovered that there were

several different tools that can assist with the process; I didn't want to reinvent the wheel where unnecessary, so this was one of the first features to be fully implemented in the project, and arguably the most important. Without *Clerk*, things would have been much harder.

➔ ***Heroku***

- *Heroku*, which is a PaaS tool (platform as a service). This means that Heroku is a *complete development and deployment environment in the cloud*, which is perfect for the current needs of the project. By hosting on Heroku, the application can also auto deploy changes after each new push to Github, so long as the newest code passes the build test on Heroku's backend. This is done by adding a *Procfile* to the root directory of the project, which contains a simple line of code, which is essentially the line that starts up the project (in this case: `web: npm run start`), we can tell *Heroku* how to automate the project once it receives confirmation that the most recent build was successful. We give *Heroku* additional information, like our `.env` keys, so they are protected from users, and can configure domain settings as well. This is all just the tip of the iceberg - *Heroku* is highly customizable with the use of extensions, that can provide support to many different features/applications that a user may want to integrate to their project/platform, and tracks all sorts of project metrics; average response time, memory usage, throughput, server load, etc.

➔ ***OpenCage***

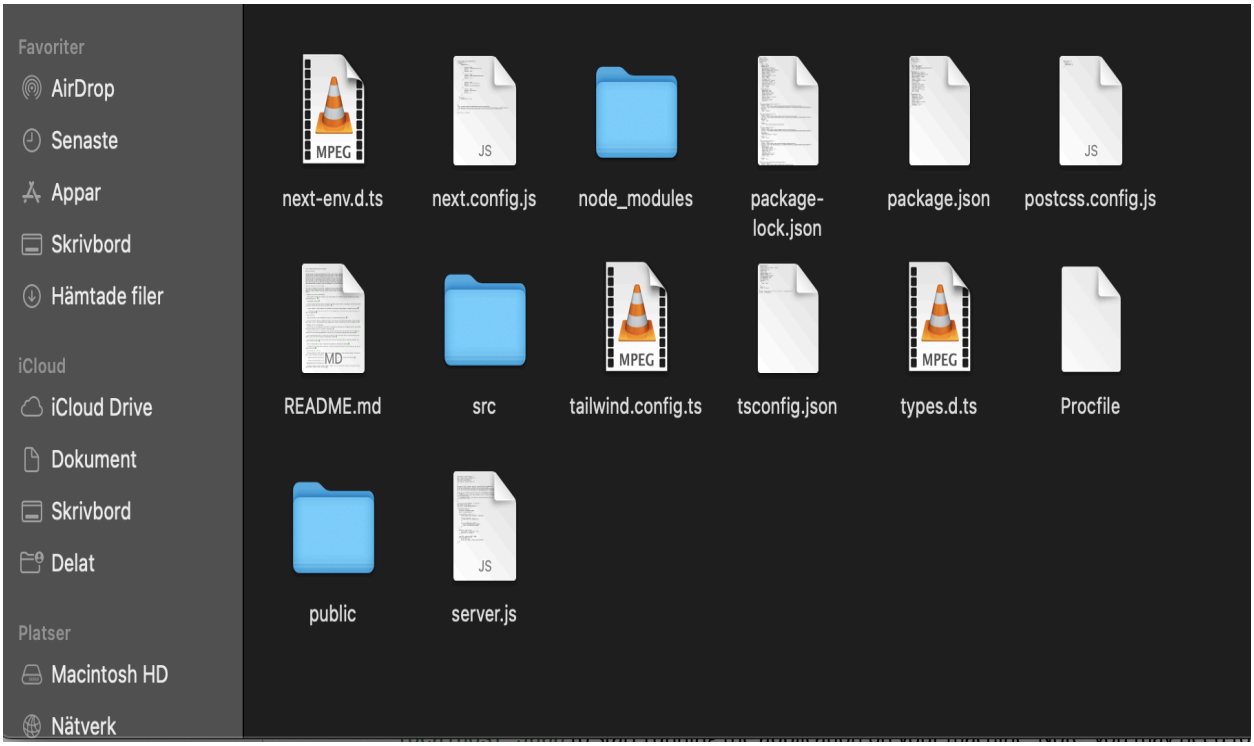
- *OpenCage* is not necessarily a part of the *tool stack*, but deserves acknowledgement in this section for providing a core feature with its services. By specifically leveraging the OpenCage *geocaching* API, Cliq is able to calculate the distance between two users given locations, and provide feedback to the user on their cards, letting them know how far a given user is away from them in miles.

➔ ***HeroIcons***

- *HeroIcons* is *also* not necessarily a part of the *tool stack*, but this platform deserves recognition *as well* for providing core support to the application. The creators of *TailwindCSS* have also created this large library, called *HeroIcons*, which contain several different variations of sleek badges/icons that can be imported into use into projects. It was very easy to set up - just a line of code to import these icons, and then another to link the imported icon component into an HTML tag to be rendered. *HeroIcons* have provided all of the badges, icons, etc. that were used in Cliq. As a resource, the documentation (and the ability to search through for certain icons) for the library can be found here: https://heroicons.com/

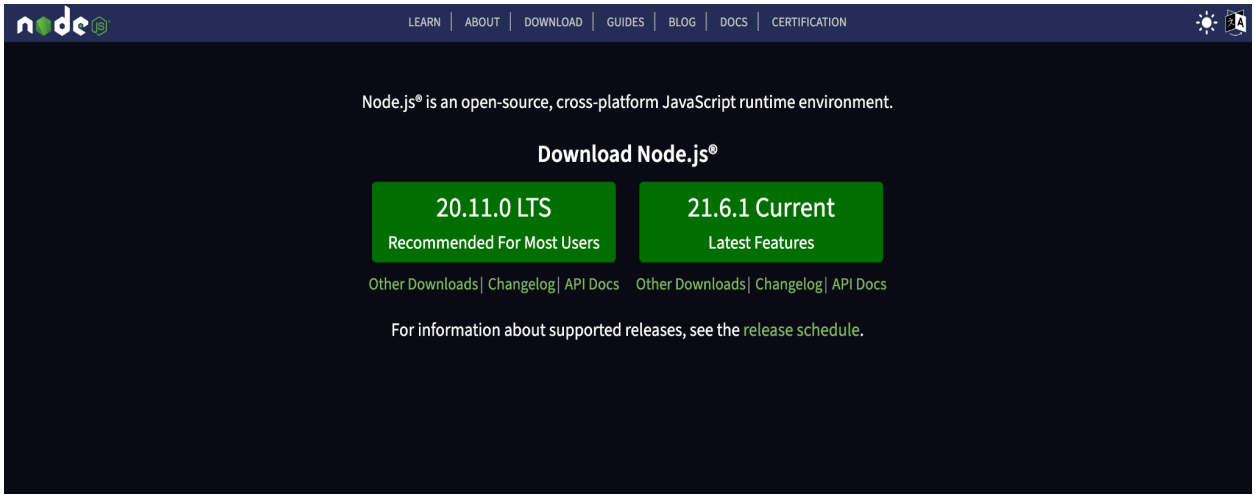
Installation Requirements

Since Cliq is a web application, there is no installation required for users to access this platform, per se. All one needs to do is visit http://www.cliq.fi/, and begin setting up an account! Cliq is currently being hosted by *Heroku*, which, as mentioned before, is a PaaS tool (platform as a service). Because of the dynamic nature of *Heroku*, and the flexibility one has when choosing to deploy with it, *Heroku* is perfect for the current needs of the project. If one is inclined to run this project locally, they would need to download the latest source code on my Github (link is currently private, as instructed, but can be made public on request) at the following link: https://github.com/ccvale/cliq/. Assuming the project is downloaded, you should open up the project folder, and have a terminal window open in the project directory. For reference, the directory should look very similar to the following:



To run this code, ***NodeJS*** must first be installed on your device - you will need to have this downloaded as a prerequisite in order to run `npm` commands.

If you do *not* have ***NodeJS*** installed, you should follow the following steps:



1. Go to the following link: https://nodejs.org/en/
2. Click *download* on the latest recommended version of ***NodeJS*** *(for reference, in January 2024, this is version* ***20.11.0 LTS***)
3. Follow the installation guide, and continue through all the steps until installation is complete
4. Open a terminal window and enter the command `node -v` to verify installation

5. If the correct version build pops up on the screen, the install was successful, and you can return to the beginning to start running Cliq locally!

From here, you should type in the command `npm start` or `npm run dev` to start running the application (depending on what you want to do with it), and visit `localhost:3000` to start running the application on your machine. Note: you may need to run the `npm install` command to download any dependencies if need be prior to starting the application (which is likely).

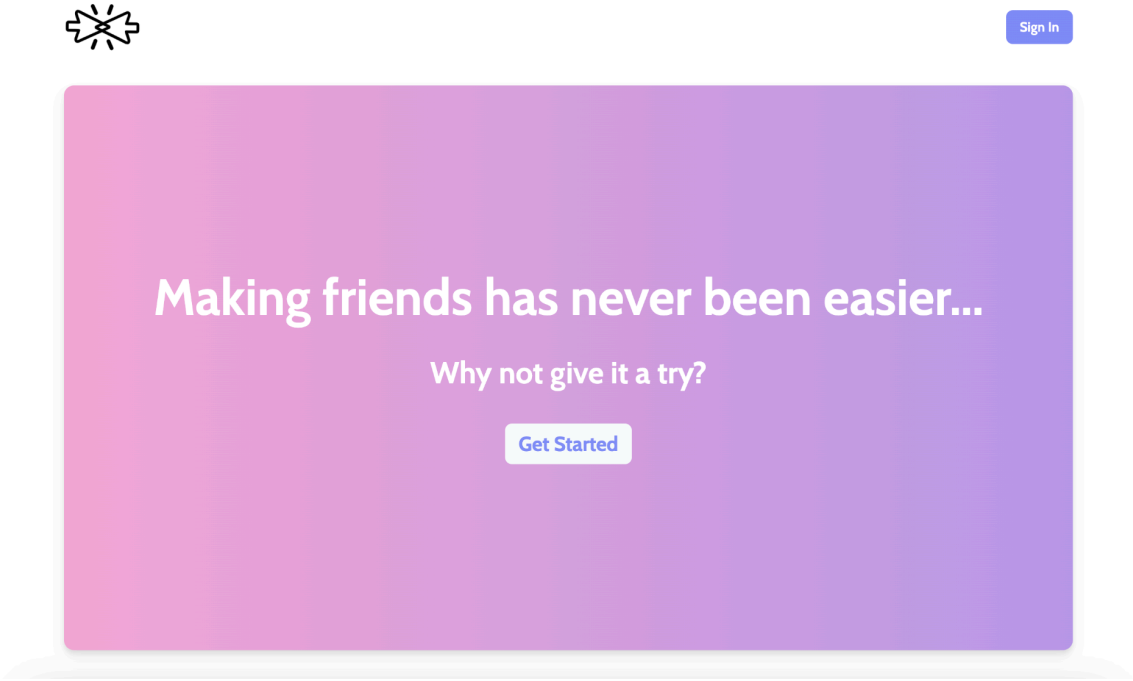
Note: you will need to also create a `.env.local` file in the root, and obtain your own API keys for the following, in order to use the platform on your own. The file should look like this: **(Professor, if for some reason, you want to run this locally using *my* API keys for consistency/access to the same data I'm using, and it isn't already on the physical copy, please email me, and I will provide you)**

```
NEXT_PUBLIC_CLERK_PUBLISHABLE_KEY= your_api_key
CLERK_SECRET_KEY= your_api_key
NEXT_PUBLIC_CLERK_SIGN_IN_URL= /sign-in
NEXT_PUBLIC_CLERK_SIGN_UP_URL= /sign-up
NEXT_PUBLIC_CLERK_AFTER_SIGN_IN_URL= /
NEXT_PUBLIC_CLERK_AFTER_SIGN_UP_URL= /
XATA_BRANCH= main
XATA_API_KEY= your_api_key
NEXT_PUBLIC_OPEN_CAGE_API_KEY= your_api_key
```

Users Manual

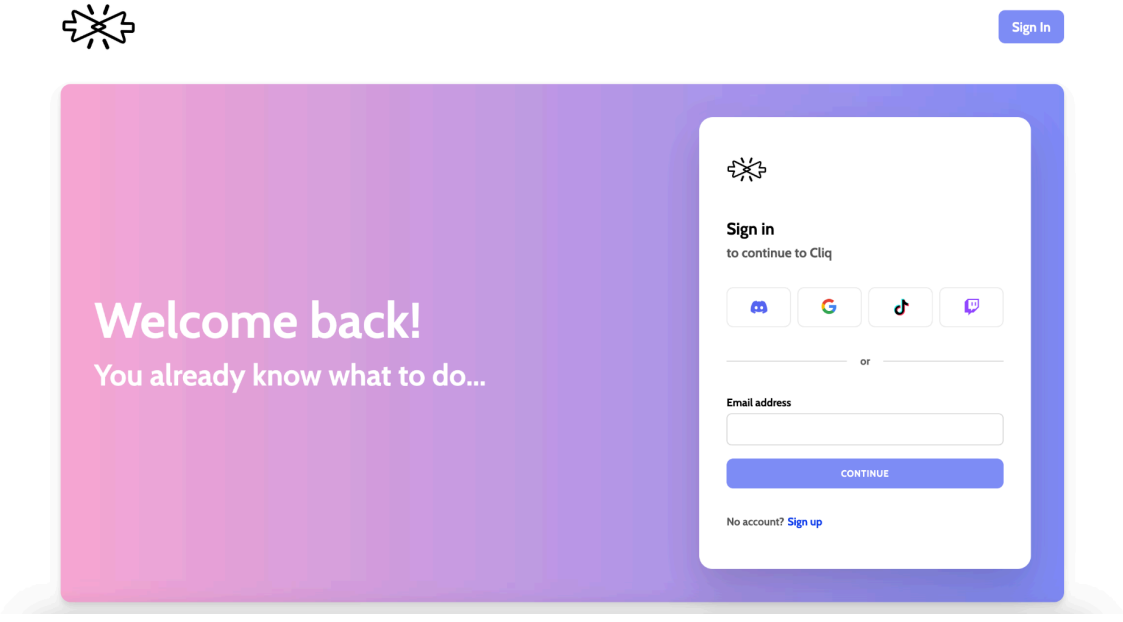
Cliq is very simple to use by design, but there are many different aspects of the user experience that can be explained to simplify the platform for a new user. For starters, the core components of the web application should be introduced and detailed; the main pages/sections of the application that the user will be interacting with, will typically be the

home page *(below)*, the sign-in/sign-up pages, the dashboard, the settings page, and the chat page. These are the main functional frontend interfaces for a user to interact with, and



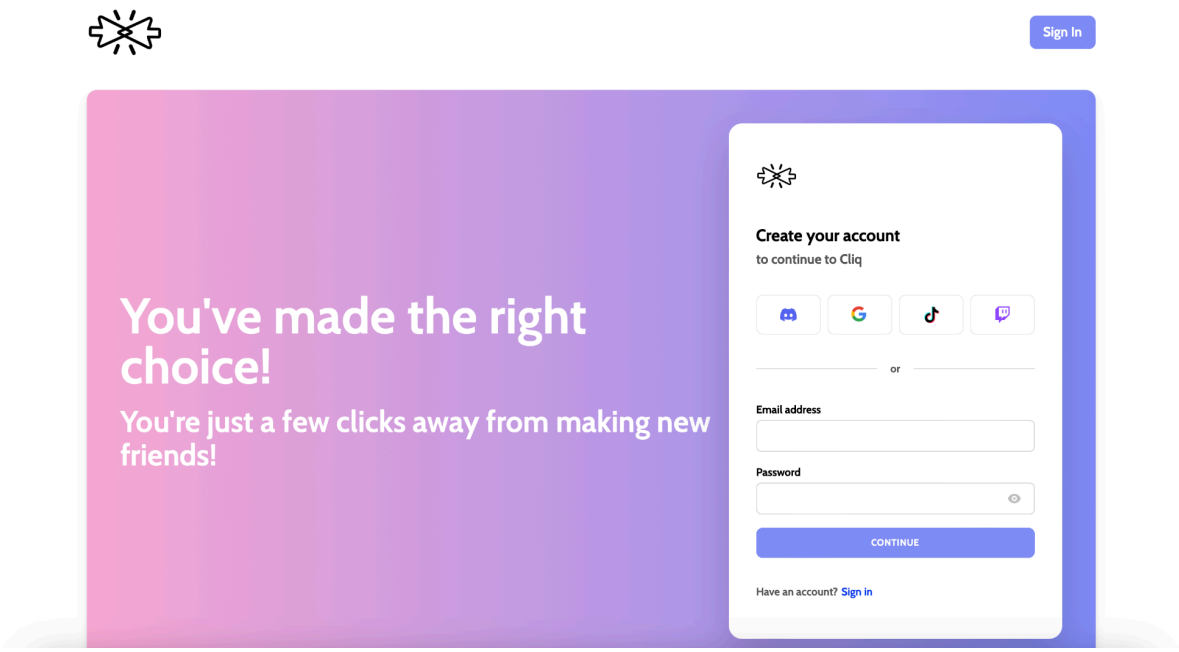
by using screenshots, and other instructions, following along with this section will make one's experience on Cliq a piece of cake!

- ***Signing In***



- If you have already created your account on Cliq, you may notice that signing in is very similar to signing up. Instead of clicking the *Get Started* button, you can now click *Sign In*. In familiar fashion, you can select your method of authentication, and get yourself back inside the web application very quickly. Hopefully many new friendships await you!

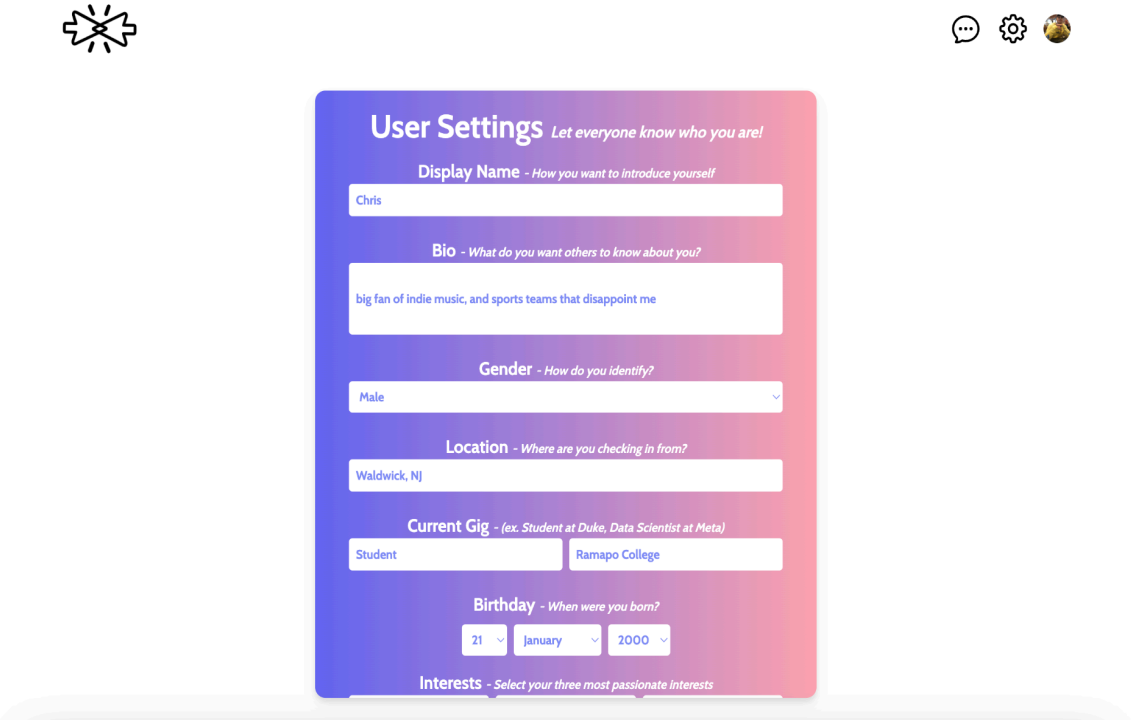
- ***Signing Up***



- For a new user, you will want to visit the website for the first time - www.cliq.fi/. From here, you will be greeted with a message on the homepage, and you will be able to *Sign In* (we aren't quite there yet), or *Get Started* (which is what we're looking for!) After clicking this button, you will be met with several options; if you have any of the following accounts (*Google*,

Discord, *Tiktok*, *Twitch*), you might want to consider clicking one of these options. This will allow you to authenticate yourself on Cliq by logging into one of your pre-existing accounts on another platform, which makes things very simple. If you don't have one, or if you would like to create a separate account, you can enter an email + password to get your account created. *Note*: *you will be able to link your account with third party platforms at any time, so these options will always be available to you.* Assuming all steps have been followed, you have officially signed up for a Cliq account! From here, your next step is to make sure that all of your settings are tailored to you, and fully to your liking - after all, other users will be seeing these!

- ***Settings***



- This is the most important part of the user experience; after all, this is where you get to decide how you want to represent yourself to other users! Though many of these options are rather straight forward, each aspect of the user settings page will be listed below:

    - ➔ *Display Name*
        - ◆ The name that will be at the top of your card - in other words, the name others will associate you with. In many instances, I would recommend using your first name, or maybe a nickname that you go by.
    - ➔ *Bio*
        - ◆ A key aspect to the profile; this is your chance to speak directly to the other users - give them a strong insight into your personality!
    - ➔ *Gender*
        - ◆ Given the option to choose between Male, Female, and Other, this also allows us to help find optimal matches based on what you're looking for!
    - ➔ *Location*
        - ◆ Helps other users understand where you are; helps you figure out how far other users are away from you. Is used for some calculations, and filters as well.

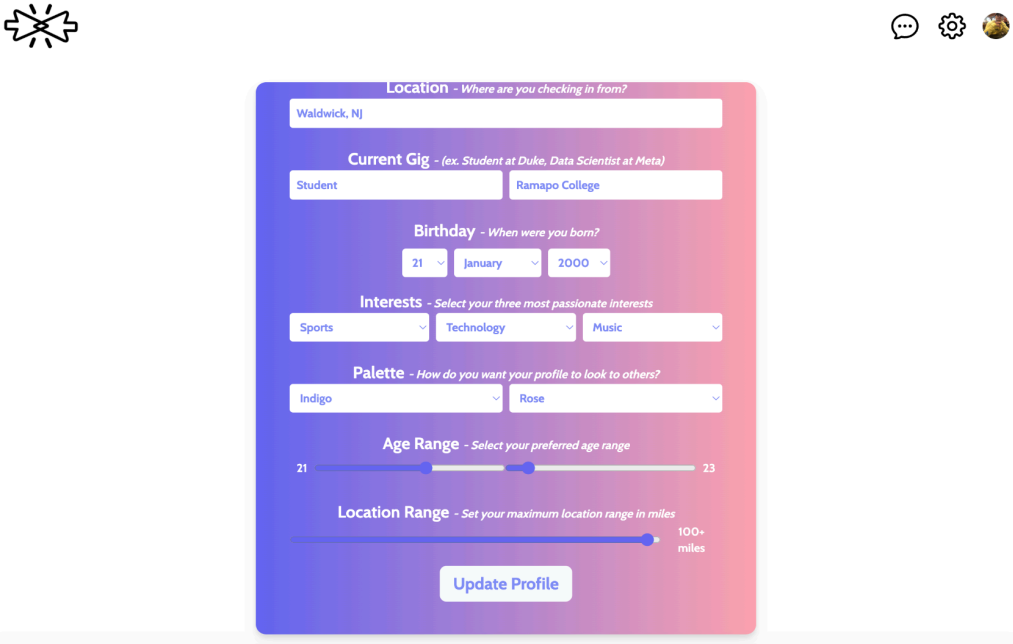
- *Current Gig*
    - Lets the user provide some information on their ‘day to day’ - do they work? Are they in school? Maybe users will discover that they go to the same school, or work in the same industry.
- *Birthday*
    - This allows us to filter users for you based on age range; it also protects users under 18 from encountering users above the age of 18, and similar logic for the other way around. This helps everyone!
- *Interests*
    - A very relevant part to a profile - a profile isn’t truly personalized without sharing these kinds of details. Let other people know the things you enjoy the most...you might find that many others have the same interests as you!
- *Palette*
    - Cliq’s answer to a unique layer of user customization. In dating apps (i.e. Tinder, Hinge), user images are a key component to the user cards. Since Cliq isn’t about dating, or theoretically not really about looks at all, per se, images are not as relevant here. Instead, by selecting two colors in a selection of ~a dozen shades, apply your own custom gradient that will be applied to your user card for others to see, as well as many aspects of your own user experience on Cliq (ex. settings page, pop up notifications, chatting experience) .

➔ *Age Range*

- Allows users to define a range of user ages that they would like to see - for 18+ users, the range is from 18-100, and for users 17 and under, the range is 13-17. Users outside this range are automatically filtered out.

➔ *Location Range*

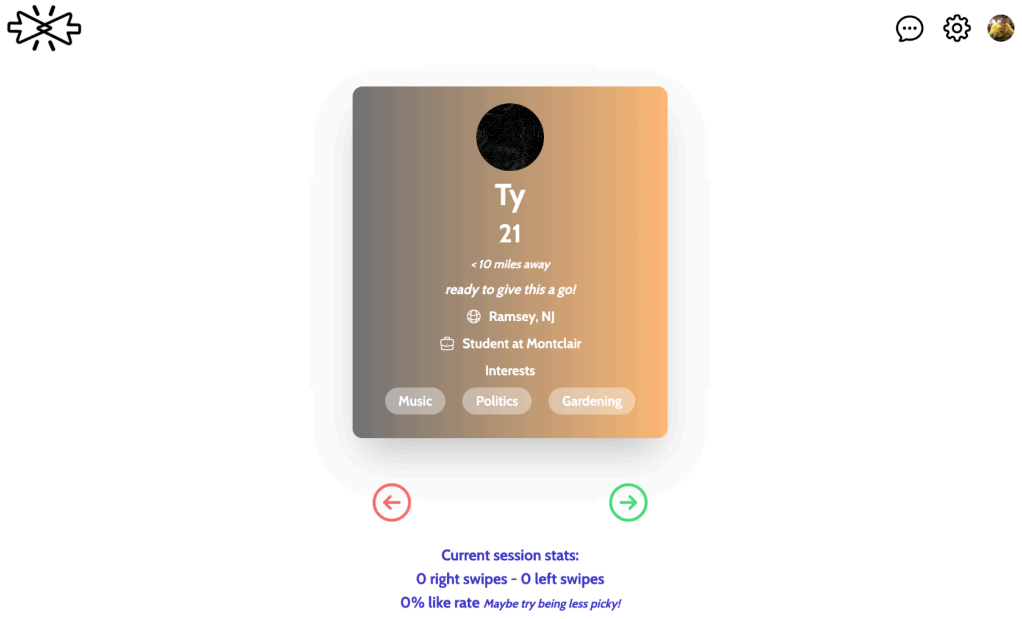
- Similar logic as for the age range. With location range, there is no minimum bound that is set - this will be 0 by default. The max location range slider goes from 0-100+. This is something that could potentially be modified down the line, as 100 miles isn't really all that far away, but for now, it does the job.



---

* These settings will be updated, so long as you make the changes on the page form, and will adjust both your user card, and the cards you encounter

- ***Dashboard***
    - The dashboard is *'where the magic happens'* - this is the core of the entire website. Here, the swipe queue commands control of the center of the page; user cards are stacked on top of one another, after being filtered to meet your preferences, as well as being sorted by a custom algorithm that attempts to give priority to the users that best match up with your profile. Using your mouse, you can click and drag a profile to the right if you would like to *match* with them. Inversely, you can drag them to the left if you wouldn't like to talk with them. You can also click on the red arrow button to simulate a left swipe, or the green button to simulate a right swipe, to make things easier for those who are tired of dragging their mouse around.

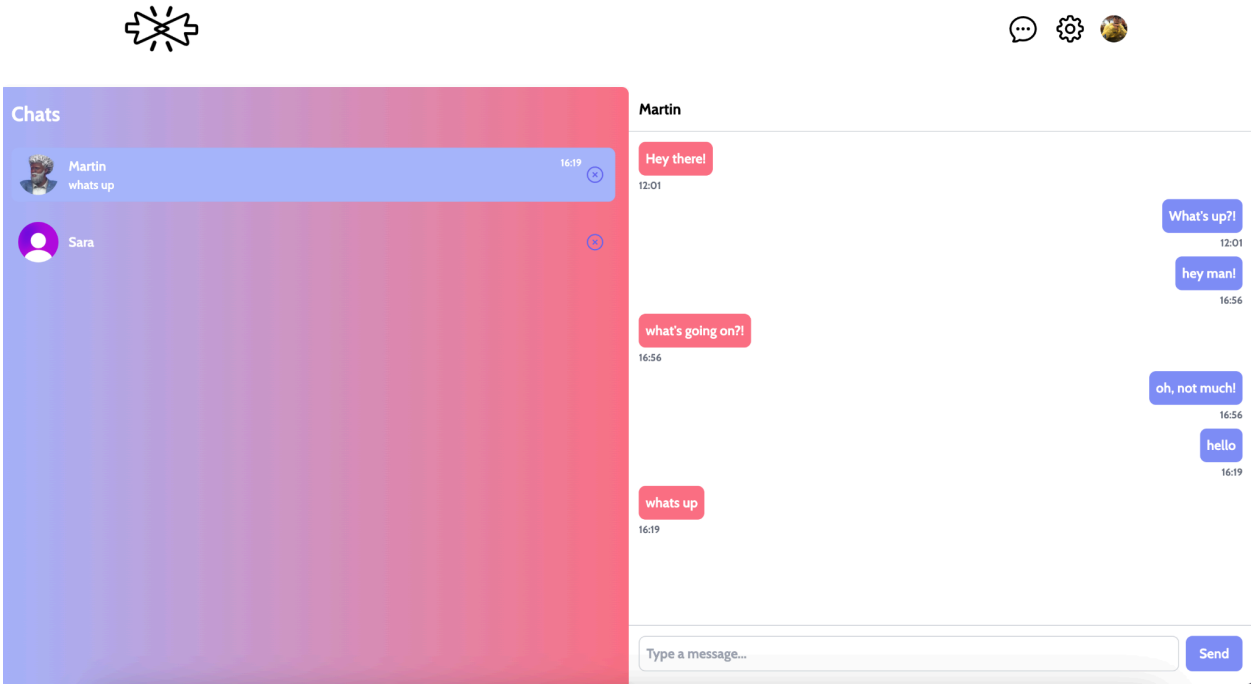


On the bottom of the page, there is a small section that outlines some swiping statistics for your current session. *(A session is defined as simply the current iteration of your dashboard visit. If you refreshed the page, or went to*

another page, this would reset the session statistics.) This includes information about left/right swipes, as well as your current session right swipe percentage, along with a message describing your current swipe habits. Ultimately, if you like a user, we check to see if the user has already liked you. If this is the case, it is already a match! If not, the other user will likely see your card soon, so the chance of matching is increased. On matches, users will be able to chat in real time with each other.

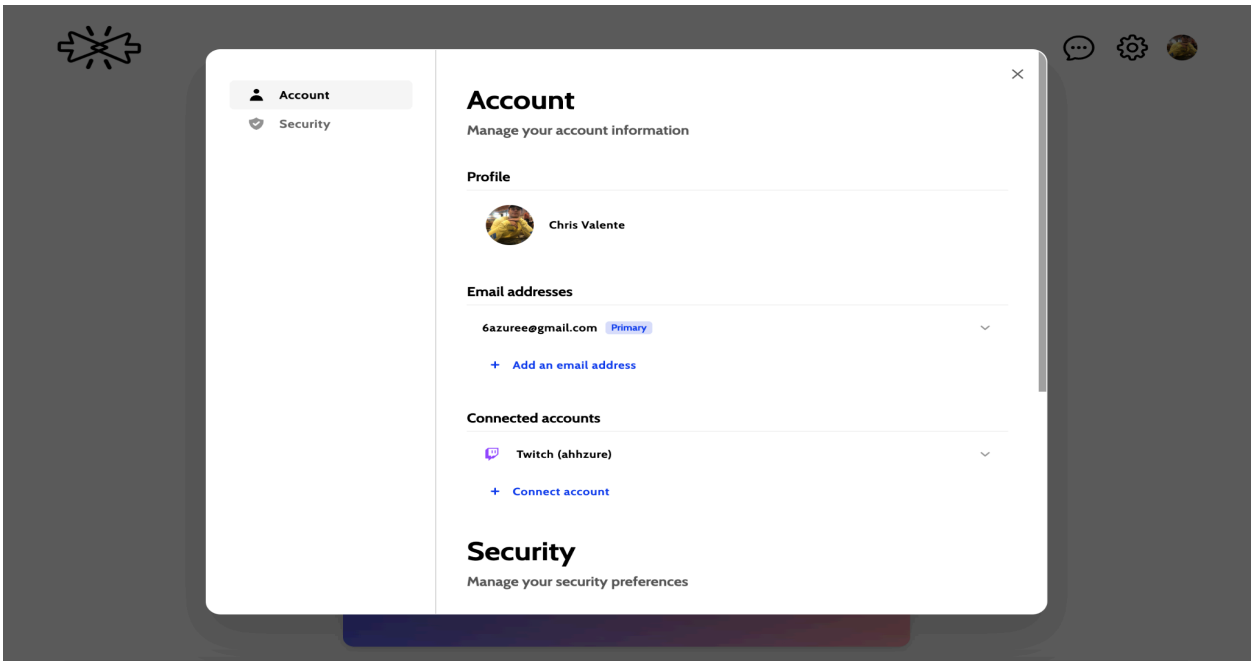
Note: if you are a new user, you will likely be prompted with a message telling you there are *'no more users left to swipe on'* - this is because you have to go to the settings page (cog icon) and fill out your profile from default settings. *This is done to encourage users to actually customize their experience before they begin swiping/using the platform.*

- ***Chat***



- This brings us to the chat page *(Note: design features as seen above are entirely based on the user palette; nearly all design aspects of the platform are dynamic, and based purely on user choices)* - if you visit without having any matches, you should expect to see a message telling you that you have no chats. After all, without any matches, the chat page would be kind of lonely anyway, right? If you come back to this page with any amount of matches, you'll see the screen split in two, into the following:
  - ***Left hand side of the screen*** - an organized list of all matches - both new, and ongoing conversations. Each given section for a match contains the name of the matched user, and their profile picture displayed. If any messages have been shared back and forth with the user, it will also display the last message sent, and the timestamp of the message. There is also a small 'X' button on the right side - this is the *unmatch* button, and if pressed, will prompt you to ask if you would like to unmatch the user. There could be countless reasons why one would want to do this, but it is an important feature, regardless of the reason.
  - **Right hand side of the screen** - closed on default, but once you click on a conversation, the chat will fill up the right side of the screen. The conversation opens up to the most recent messages shared between the two users, so you can respond quickly, and move on.

- ***Account Settings***

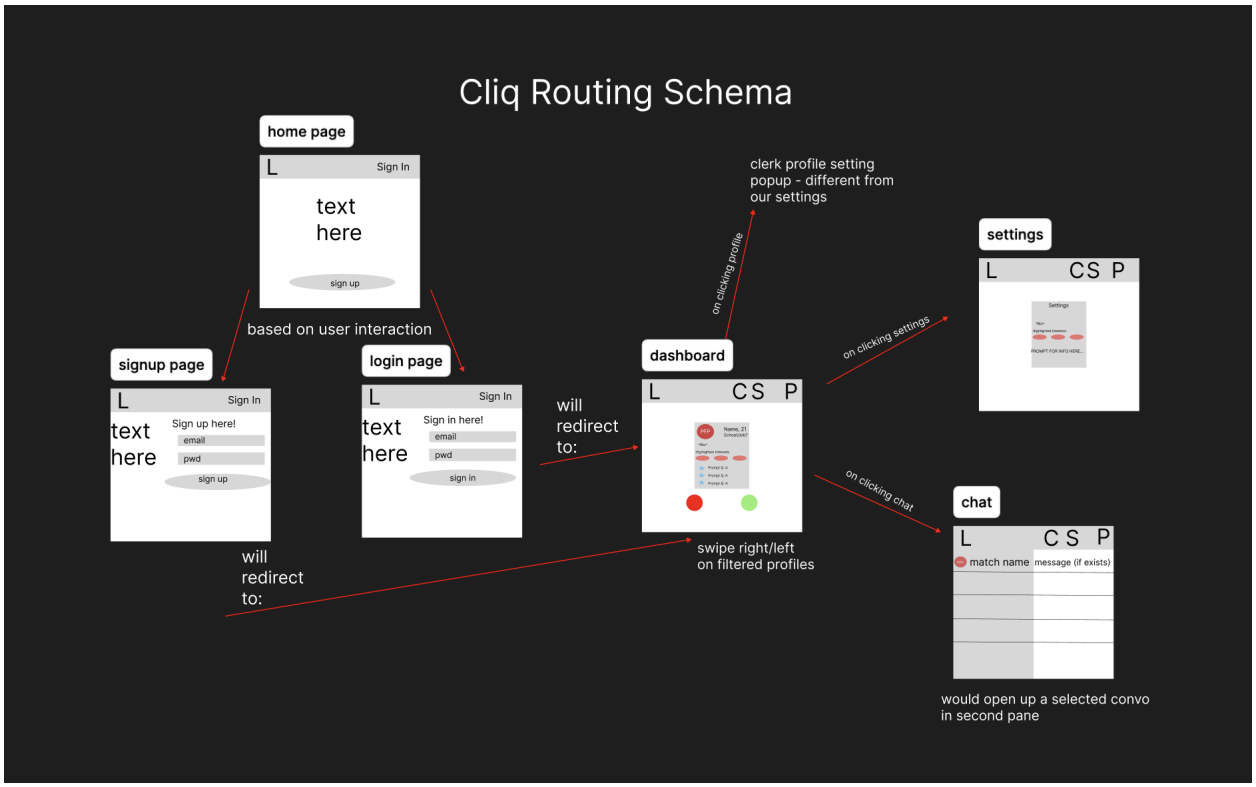


- This may be a bit confusing; there are two different types of settings that are utilized on Cliq. The settings page, ***which is identified by the cog icon*** in the navbar. There is also the *account settings* tab, ***which is identified by the circular profile picture*** in the top right corner. This particular instance of settings is powered by *Clerk*, and relates to the backend facing features. It allows you to do some of the more *account based* things - change profile picture, modify your third party connections, email settings, etc. These are two separate pages with two separate purposes, but they are both very useful.

Design - Documentation

When designing the outline for the project, there were two things I focused on at first; the mapping out of the page structure, with a very rough idea of how I wanted pages to be

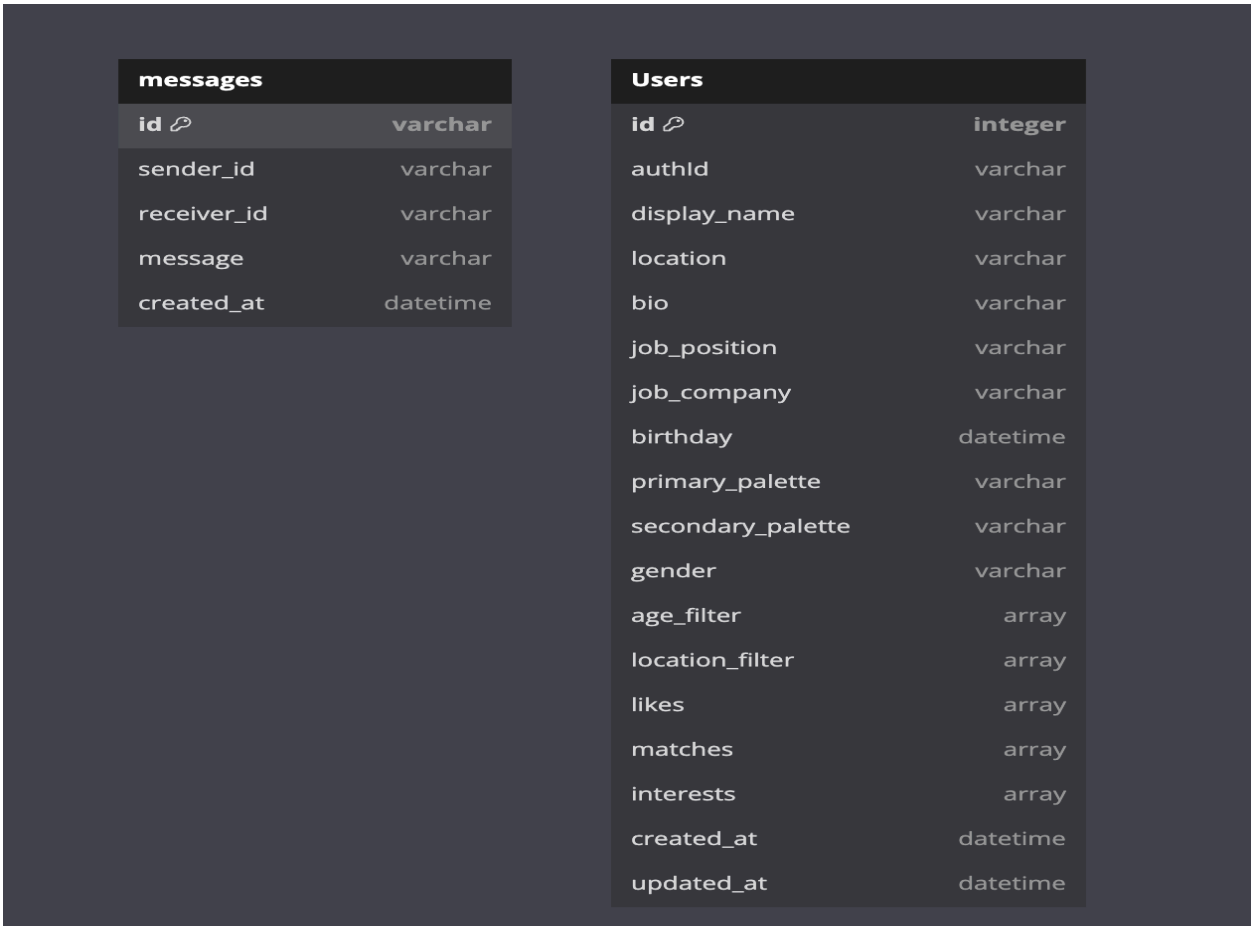
organized, and how I wanted the flow of the application to go. The illustration below was the design diagram of sorts, for the layout. As you can see, all the pages were accounted for, even from the beginning. In this sense, I always had a vague general idea, and because of this planning, things were able to stay very close to the conceptual outline. I was even able to mock the design of each page, which actually served as a reference/guide, more or less, throughout development, as I tried to turn this very simplified sketch of these page designs into pages that matched the aesthetic I had cultivated in my head.



It's important to note that this design process for the page routing/layout took place *before* development even began - since I have never worked on a larger web application project before, I thought it was prudent to get a foundational idea of how this website would ultimately be structured. I was satisfied to see that much of my original concept was able to exist how I thought it would, and am happy that even the very sketched versions of the

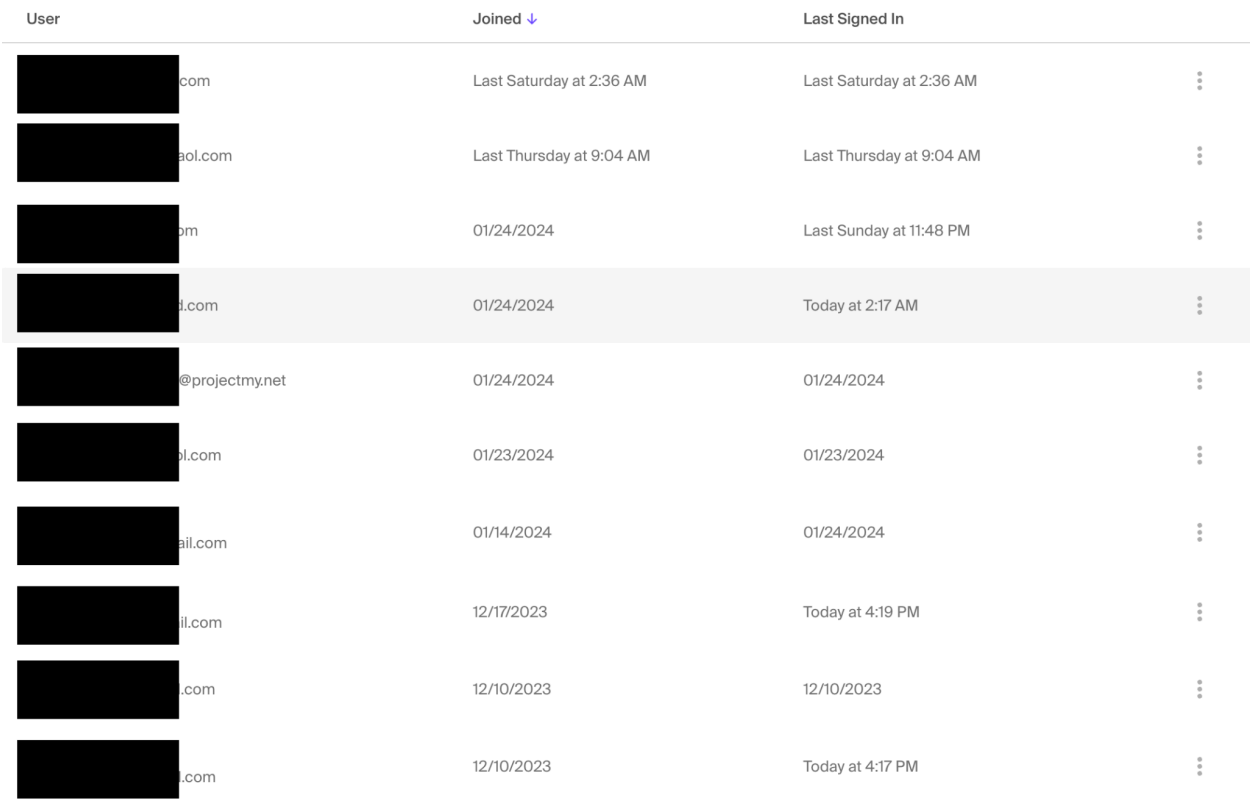
actual pages were able to help guide the design process to the point where many of the page layouts are very similar to reality.

Another thing here that was key was the database planning. I knew that there were two separate tables that we wanted to involve - one table for our users, and another for messages. As features were added, and removed, and as the project went on, things were modified quite a bit from the original database diagram. But as of completion, the current schema for both of the tables are as follows.

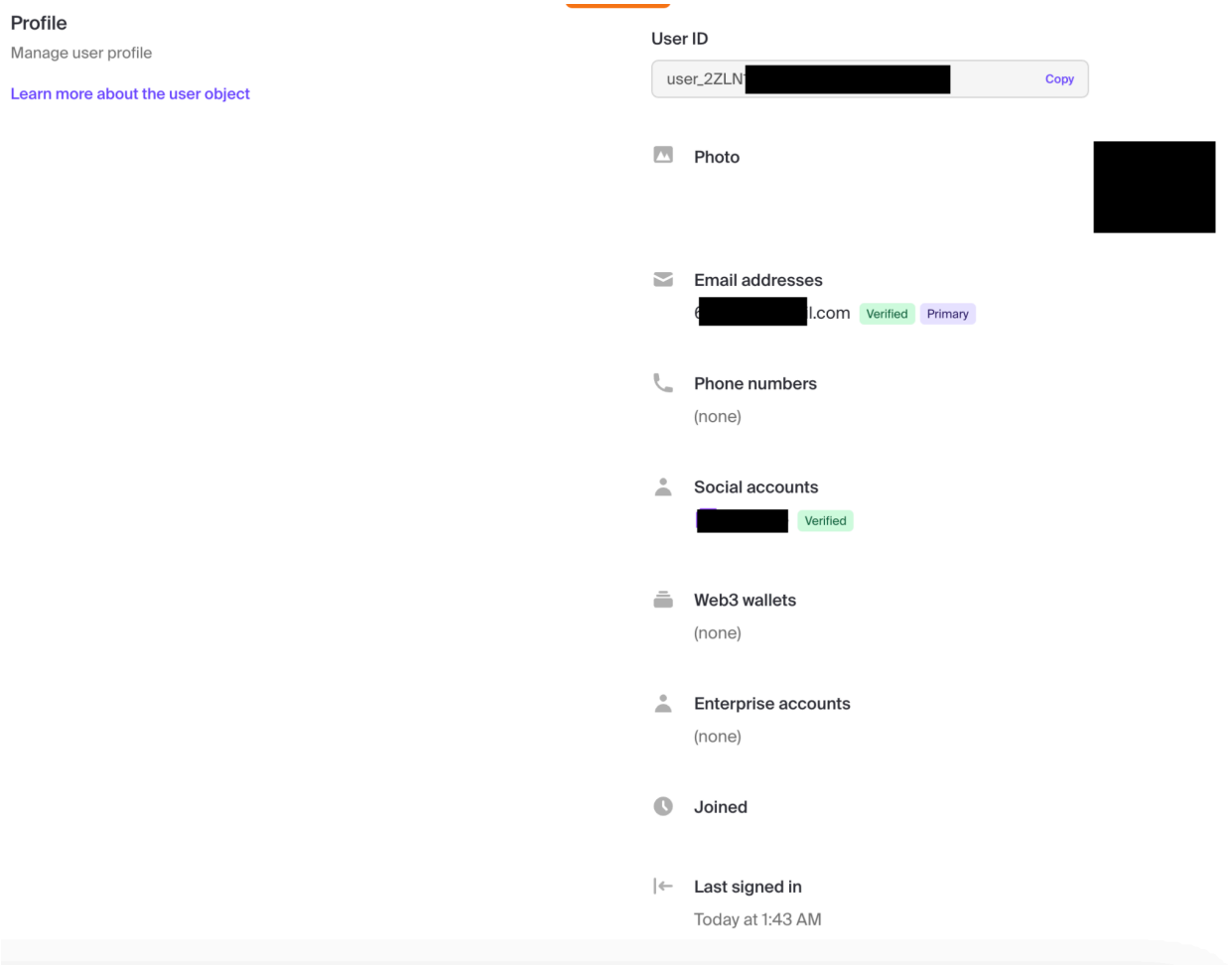


This diagram didn't let me indicate foreign keys, but the implementation of this design behind the scenes relied heavily on the balance between *Xata* **id**, and *Clerk* **userId *(fk)*** to

obtain, and update user information. Similarly, **sender_id** and **receiver_id** were both used as foreign keys to get matching results based on the pair - these messages would indicate the existence of a conversation, and were formatted based on the other metadata of the message that is provided in the table (ex. who the sender/receiver was).



The above is an example of the *Clerk* dashboard. Just previously, I described the need to link *Xata* and *Clerk* tables together, as *Clerk* was for authentication, and *Xata* was for just about everything else (user data, message data). From this view, we can see how users are represented on *Clerk* - more specifically, the way that they display user information to the developer. For some registered accounts, we can see attributes such as date joined, and last sign in, which allow us to observe our level of engagement at a quick glance.

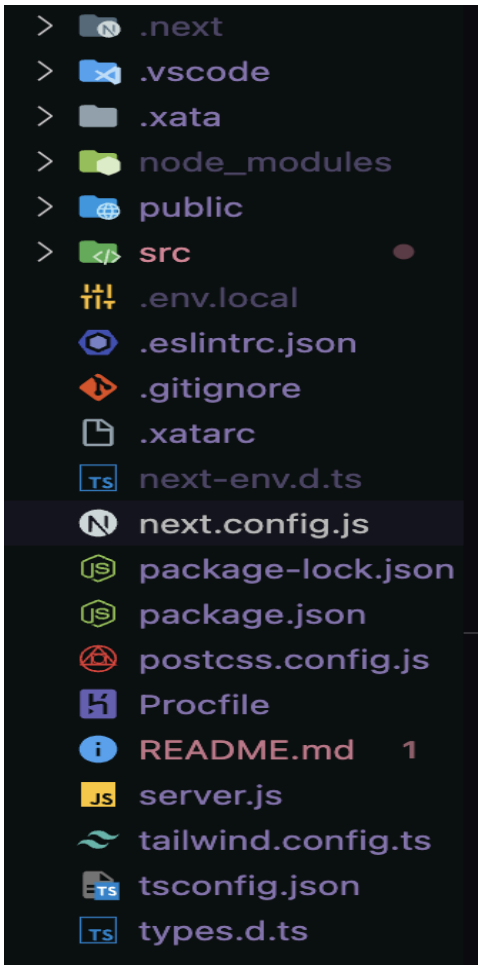


This view, however, correlates fully with the settings that the user can modify within the *Clerk* settings component. Here, for an arbitrary profile, we see the *user id*, which is used to link together this data with our main database, and other information, such as email, photo, and linked social account(s). By linking the two databases together, we are able to easily access *all* of this information for a given user with a quick query.

Design - File Structure

The file structure for the project from the root directory looks like this *(image on the right)*. Each folder/file that

contains ***relevant substance*** that is worth discussing will be explained, so it is clear what the function of each major aspect of this structure exactly is.



- ➔ *.next* (folder)
  - ◆ Contains default build settings, imports, files, that is required from *Next.js* - these were placed into the folder automatically on project initialization

- ➔ *.xata* (folder)
  - ◆ Contains initialization metadata for our *Xata* tables and structures

- ➔ *public* (folder)
  - ◆ Contains images; logos, favicons, etc.

- ➔ *src* (folder)
  - ◆ This folder contains almost all of the user written code that relates to the actual functionality of the application (components, routes, pages, library functions, API calls, etc.) - this will be discussed in detail in the next section

- ➔ *.env.local*
  - ◆ Contains all of our *secret keys* for the project - these are not carried over to Github, so they are kept on *Heroku* during hosting. Can be provided if needed for reference

- ➔ *server.js*
  - ◆ Provides critical functionality, specifically to the *chat* page - this file creates an *express* server, and a *socket.io* server, which allows the application to be configured to listen for connections/disconnections, and specifically for the *sendMessage* event and emit the *receiveMessage* events, which act as follows:
  - ◆ *sendMessage*: When a client sends a message, the server receives the message and emits a receiveMessage event to the user on the other end.

- *receiveMessage*: When a client receives a message from another user, the client displays the message in the chat window as soon as it's sent.

➔ *tailwind.config.ts*

- This file has actually proven to be very useful throughout the project. Due to the way *Tailwind* works, it doesn't like the idea of rendering *dynamic CSS*. For example, passing a given class:

```
w-2/3 bg-gradient-to-r
from-${sessionUser.primary_palette.toString().toLowerCase()}
-300
to-${sessionUser.secondary_palette.toString().toLowerCase()}
-400 p-4 overflow-y-auto shadow-lg rounded-lg
```

This doesn't work by default. The idea here is to dynamically adjust the color scheme of some arbitrary object to match the primary and secondary palettes of the *session user*. Since we are passing through these variables, this is *dynamic*, which unless specified, *Tailwind* cannot render these classes like this. However, if we make some modifications to the *tailwind.config.ts* file, we can explicitly tell *Tailwind* that there are some dynamic classes that we want to include, and the real classes are added to a *safelist*, where they are essentially rendered beforehand, so that when the dynamic class name from above is read, it will now interpret it as, for example:

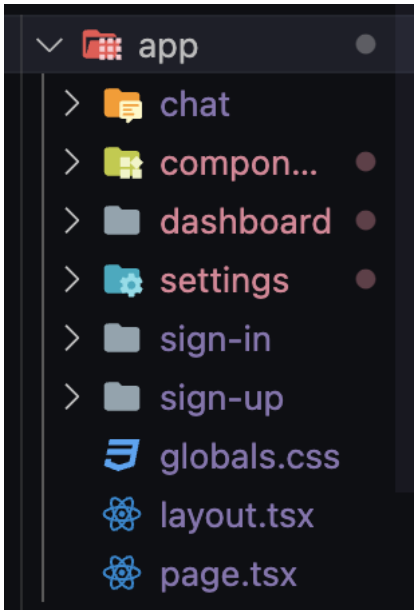
```
w-2/3 bg-gradient-to-r from-indigo-300 to-violet-400 p-4
overflow-y-auto shadow-lg rounded-lg
```

which are now rendered classes that are included within the safelist, so they can be used. This is essentially the logic behind how the dynamic user palette UI is rendered.

Design - `src/app/components`

The contents of the *src* folder ultimately contains two major folders - the *app* folder, which contains the structure of our pages/their components, and the *lib* folder, which contains a

lot of the reusable functions, API calls, etc. In this section, the *app* folder will get explained, and clarified, as the goal is to explain as much of the core components as possible in the most simple way. To understand the layout of this folder, one needs to understand the way routes work in *Next.js*. For each folder represented within the app folder (ex. sign-in, chat, etc.), they must contain a *page.tsx* file inside. This file, essentially, is the route, and is thus, the server side component to the page. Because of this, the majority of the time, this *page.tsx* file will perform much of the "server side" requirements for a particular file. For example, at the *chat -> page.tsx* route, the server side aspect obtains the current user, checks to see if the user has any matches, and renders different components depending on this - if the user *does* have matches, we will make a further call to obtain all messages sent to/from this user and pass details about the user, the users matches, and any user messages to our *components -> ChatComponent.tsx* file. Alternatively, our *page.tsx* file will, in most situations, obtain the aspects that we need on the client side to properly render the things we need to render, and we pass it to a client side component, which is found in the *src -> app -> components* path. In this *components* folder, we have several different files, which all provide the core functionality for each different page. These are:



- ➔ *Chat*
  - ◆ Receives details about the current session user, and formatted information about their matches, and message history with these users as parameters from the corresponding *page.tsx*
  - ◆ The chat page handles many state situations; if a user has no matches, and if a user *does* have matches, first and foremost. If a user does have matches, we render out a common interface, where chat previews are on the left hand side, and, if clicked on, the conversation on the right.
  - ◆ On user sending message (or on user choosing to unmatch), we communicate with our database using our API functions to update chat history, or to remove users from their matches
  - ◆ We also have our socket listening here for incoming/outgoing messages, and works to format incoming messages appropriately, to allow for real time communication between users

- ➔ *SwipeQueue*

- Receives details about the current session user, and a formatted+filtered object containing profiles that match the filters selected in the session user's settings preferences, and are sorted in order of *desirability* according to custom algorithm as parameters from the corresponding *page.tsx*
- Initializes information about the cards that will make up the *swipe queue* - calculations such as distance apart, age, etc. have to be made, and are done here
- Logic for handling swipes is also done here; minimal work on a left swipe, but on right swipe, sending a like to the corresponding user in the backend, checking to see if a match exists, and updating both users accordingly if so
- This also handles mapping to the bottom half of the page (logic for the buttons, and the *session stats* section, as well)

➔ *UserCard*

- Receives details about the user which corresponds to the card that is to be rendered, and a distance tag that needs to be awaited to calculate as parameters from the corresponding *page.tsx*
- Takes the calculations from the *SwipeQueue*, and the user profile data to generate the card, and render it when it comes up in the queue - very simple, as all the heavy lifting is done in the *SwipeQueue* component

➔ *Navbar*

- Is a dynamic, core attribute to the application; based on page state and authentication, can carry out many different functions, and exists to provide easy interface to users for page routing
- On logged in state, logo press is mapped to dashboard, and chat, settings, and profile icons are displayed
- On logged out state, logo press is mapped to home page, and the sign in button replaces our logged in actions

➔ *Settings*

- Receives details about the current session user as parameters from the corresponding *page.tsx*
- Consists of a large form - uses state to read and update data entered into the forms for each field; on submit, these changes are updated on the backend, and reflected in both the user card, and on the user interface (on palette change

Testing

The way this application was *tested*, doesn't necessarily show up in a detailed way here in the source code, with not exactly lots of assertions or test cases written. After all, this is a dynamic application that takes place over the web - lot's of different things can go right/wrong in different environments, to different users, etc. This doesn't mean that testing didn't happen, though. For starters, a heavy amount of time developing was spent using *test user data*. Much of the remnants of code used to generate these users still survives in the project, but this was crucial to testing many aspects of the project - from testing the way data can (and can't) be unpacked from objects, to unpacking and formatting this data onto a card, to handling states when a queue of users are in line. Until every necessary situation was able to be implemented successfully, test users kept pressure off the database, and made developing these features much quicker. Something else that was tested and tweaked quite a bit was the *algorithm* used to compare similarity between users. I'm sure that current algorithms exist, but I wanted to have some fun with this, and try to do things my own way. To do this, I again, brought in some test users, and filled them with data (varying in age, interests, location, job, etc.). The goal was to try to ascertain what attributes that *I personally* would care about, if I was a user on the platform. I would tweak the *scoring.py* testing file accordingly, and would rerun the scores until I found a fair balance that correlates to the order that I would have expected to see, had I needed to rank these profiles in order of interest.

Though I could have looked into ways to implement automated testing, I decided to do much of the actual user testing myself, manually. After the implementation of real accounts, I created a few accounts to test some of the core platform features - this included handling likes/matches, real time messaging, and the rendering of elements based on user palette. A majority of this testing was focused on ensuring proper updates on the database, and that these updates were being reflected in the front end. As one can expect, in testing, the expected outcome, and the reality are often two separate outcomes entirely. There were many instances where likes/matches/messages didn't register, or colors didn't render, for example. This led to the further bolstering of the code, where certain edge cases were able to be identified, and improved upon. Since a lot of functionality was really implemented on a page exclusive basis, there weren't many instances where something like *regression testing* was necessary, as most occurrences of bugs, etc, were rather isolated to their own section. However, one instance can be used as an example.

When it came time to register likes/matches, there were some design decisions to make in the backend. Originally, when a like occurred, the `user_id` of this user would be registered and kept track of within the session user's *likes*. Continuing down the line, this logic was the same for when a match was made, as well. However, when we need to access data about our matches, we wouldn't be able to, as we are just being passed the `user_id` in our structure, from session user's *matches*, and don't want to make another database call for each of our matches. This was detected during implementation of another feature, and I realized that we would need more than just the `user_id`. So as more features were added, I realized that quirks of my previous implementation have caused issues further down the pipeline. This is essentially the very definition of *regression testing*. This was resolved by adding more attributes individually as needed, which needed to be revisited and modified a few times. *(Now, I would probably just append the entire user object to begin with, but it's perhaps too late to modify this entire structure. But the point of learning is that, if I was to begin on a project like this again, I would have already learned this lesson, and wouldn't have to learn it again.)*

Note: see separately labeled section in binder to view test plan!

Summary

Overall, this project was extraordinarily insightful, and resulted in me learning a lot. Coming into the project, I had little to no experience with web development, as I was always slightly overwhelmed by all the different aspects, and never had a good enough reason to dive in. I've definitely had a lingering desire to do so, and combined with my renewed interest in *social apps* due to the curiosity that I described in the introduction, I believed this was a worthy project to pursue. There were absolutely points where I felt completely lost and confused, and wondered if I was starting to regret jumping into something I had no experience with. But, like with many times in programming (and in life, really), at some point, all it takes is for one concept to click for everything else to come into place. Though it took me a bit later than expected to get there, I'm glad I kept going, and have ended up being very satisfied with the current state of the proof of concept social platform that I have ended up with today, where every new line of code written was a lesson learned. How sockets interact with web applications, how users indirectly interact with databases, how different aspects of web apps interact with *each other*.

I can say that there has become a proficiency with Javascript/Typescript that definitely didn't exist before; repeated use of certain syntax has allowed me to get much more

comfortable with the languages, and many of their quirks. Considering the fact that this was one of my *potential roadblocks* in my introductory presentation, I think a good job was done in regard to gaining a better grasp on this. With regard to *Next.js*, I am aware that this definitely wasn't the cleanest web application ever written. The fact of the matter, though, which I mentioned in a previous section, is that *many* things that were learned, perhaps were learned too late into development. For example, the use of *props*, I understood vaguely at the beginning of the project. But the closer I got towards the end, it increasingly made a *lot* more sense, and I started to realize all the ways I could restructure the project to better implement them. This is just one example, but there are frustratingly multiple instances of similar sentiment. The bright side, which I am choosing to see here, is that, (as I said before), if I was to do it all again, it would be much better; and I can't help but feel that this was the purpose of the course in the first place. I was afforded the time to research and learn about an offshoot of development which I had always wanted to look into; I was able to plan, and develop a project to finish within that very field. It was so interesting to see how all of these different technologies in a stack come together to provide so much functionality to an application. From a personal standpoint, the completion of this project is a particular milestone for me, and has perhaps pulled back the veil of mystery behind all of these major apps, and platforms.

Works Cited

Articles

https://www.rootsofloneliness.com/covid-19-loneliness-survey

https://newsroom.cigna.com/loneliness-epidemic-persists-post-pandemic-look

https://xata.io/about

https://xata.io/docs/getting-started/cli

https://www.mongodb.com/databases/serverless-database

https://www.freecodecamp.org/news/next-vs-react/

https://nextjs.org/docs

https://clerk.com/docs/references/nextjs/auth-middleware

https://nextjs.org/learn/ (super helpful)

https://nextjs.org/learn/react-foundations/server-and-client-components (this chapter helped a lot with a breakthrough)

Videos

https://www.youtube.com/watch?v=tWNrM-7mebk (Video where I discovered Xata)

https://www.youtube.com/watch?v=aQN4bJ1yc-k (Video where I discovered Clerk)

https://www.youtube.com/watch?v=843nec-IvW0 (Nextjs tutorial - followed along with this until I felt like I could get started, absolutely recommend him)