Shaelyn Reno

Senior Project CS480

3 December 2025

Senior Project: Exchange Mart

The idea for my senior project comes from one of my favorite hobbies, blind box collecting. There are collectibles, either about an original character or a well-known character like Snoopy, with different themed series. In these series, you can buy a box that has a mystery figurine in it. It is a popular interest both online and within my friend group. I thought that creating a web application that allowed people to keep track of their collections and connect with others was missing from the blind box community. So, I sought out to create a website that I thought would bring together collectible lovers.

At a past internship with Northcross Group, I gained experience with the JavaScript framework, React. I had a hard time building my foundational skills and best practices with the language since it was my first time using JavaScript. When debating doing a senior project, I thought that it would be the perfect opportunity to deep dive into JavaScript to better my skills with the language. Additionally, I have experience with SQL Databases through my Database Management Systems and Server-Side Programming at Northern. I thought an obvious solution for how I would complete my senior project was just to use an SQL database since I knew how to create and manage a SQL database, React for my frontend, and Node.js as my backend since I had exposure to both at my internship. When I brought this up to one of the senior developers at Northcross Group, he recommended that I use Supabase and Next.js to build my project because Supabase helps to automate APIs and Next.js is a full-stack framework with similarities to React, but with an extended library. He explained that these technologies would help me speed up the

process of developing my project. Trusting his judgement, I decided to use these technologies for my project.

Supabase is an incredibly easy-to-understand and ideal for a small-scale application. It is a Postgres development platform and functions as a backend-as-a-service and is an open-source alternative to Firebase. It provides a large variety of authentication, storage, and database tools to make development easy. They enable social logins like Google, Facebook, and Microsoft, as well as open-source S3-compatible object storage. With access only to the tools available in the free version, I had no troubles or limitations with my project. To implement Supabase in my project, I went to go to their documentation on how to create a Next.js and Supabase and ran this command:

```
npx create-next-app -e with-supabase
```

This created an example project that had a tutorial for making your own project. To fully connect my project to my Supabase database, I had to add my Supabase URL and public key to my env.local file. These were provided through Supabase's Authentication and Project settings.

Before setting up my project, I had taken the time to plan my database design in Draw.io. This allowed me to immediately start implementing my design through Supabase's SQL editor. Per their recommendation, I enabled Row Level Security (RLS) on each of my tables. However, as I was working on aspects of my project that required writing to and deleting from the database, I would get weird errors. Despite the number of times I double-checked or fixed my code, I would get the same error. When I looked at my database, I realized I had never taken the time to configure the Row Level Security that I enabled on my tables properly. For the sake of time, and with disregard for security, I deleted all Row Level Security policies. This allowed the data from my site to be properly pushed through and update my database. I have not yet re-

enabled the RLS policies on my database, but I do intend to go back and configure rules to increase the security of my application.

Supabase makes authentication easy by including helpful tools such as popular authentication providers, user objects, and more. As I was creating my database in Supabase, I was having a hard time figuring out how to store my project's users. I had known that Supabase had a way to handle my authentication, but I did not understand how to connect it to my own database design. After spending more time in the application, I found that they have several built-in schemas of their own, one of which being auth. The auth schema is the basis for Supabase's authentication. When a user attempts to log in, Supabase's authentication server will receive the request and verify the credentials. If successful, a JSON Web Token is generated and added to that user's row in the sessions table. I did not have to configure the logic myself since I reused the authentication components from the example Next.js and Supabase project.

Another benefit of Supabase's authentication is the ability to send emails regarding user authentication actions. For my application, I was able to configure Supabase so that when a user is created, an email is sent to the user to confirm their email address. Once the link is clicked, confirming their email, they are automatically redirected to the home page of my application. The same process happens when a user would like to update their account password. An email is sent redirecting them to the update password page of my application. Initially, this feature gave me trouble when I first made my website public for my friends. After viewing the screenshot they provided of the error they were getting, I realized I had not yet reconfigured my URLs to match my production URL. I was able to adjust this by going into Supabase's authentication and to the URL configuration. I had to define my new site URL and a redirect URL for the update password URL.

Next.js takes a file-based approach to routing, meaning the way that you structure your program directly reflects how your web application's routing will be. A project can be structured by placing routes in a pages or app folder. These define the routing mechanisms, default component types, and data fetching. By putting all my website's pages within the app directory, it meant that I would need to use the Next.js App Router and all components to server components. This is their newer, server-centric system that allows for more configuration to be done by the developer. The way that routing is done is based on file structure and file naming conventions. The most common types of files in my application were layout, page, and route. The layout page is applied to the current directory and all children of that directory. I use this when creating my header and navigation bar at the top of my application. By placing my file in my root directory, each page in my application has the same header. The page file holds the page's HTML and UI for the route. A route page is specifically used for APIs, which are often used for client components. Next.js allows for dynamic routes by putting square brackets around the parameters that will be passed through the URL. This especially became useful in the checklist section of my application. This file-based approach works very well for me, since it is highly visual. However, because of the need to create so many folders to define the paths for the application, it can very easily become messy with the more paths that are needed.

Since Next.js is a full-stack framework, components are divided into server and client components. Server components are made to fetch data and render the user interface on the server and stream it to the client. Client components are made for interactivity and browser APIs. When I was initially working on my project, I would often mix these two components by adding a "use server/client" line at the top of a function or file. I would try to fetch data within a client component that I specifically had made for user interaction on the page, like the checklist, and I

would be met with an error. I did understand the concept of what a server or client component was and why I was getting these errors. Once I learned more about what each of these components are capable of, I was able to correct a lot of my work and fix my errors. I had to backtrack through the blog, checklist, and profile pages to break out my client and server components. I learned that server side is good for displaying data, so it works well for displaying blog posts, profile information, collectibles, and series. However, when a user wants to change a piece of data, a client component with an API call works much better than trying to manipulate a server component.

An example of this is when I modified blog posts on the blog page so that comments were a separate client component. This allowed me to use React use states to submit new comments and replies to existing comments. When a user submits a new comment, the content is sent to a handleSubmit function that will prepare all data needed for the API call's body, such as the comment content, blog post ID, and UUID for any potential parent comment. The API is called through a fetch with the file path to the API and an object holding the method type, header object, and the body (a stringified JSON object) as parameters. In the API, a Supabase client is created, user data is retrieved, and content from the JSON body is extracted. Finally, a new row can be inserted into the comment table in the database, and a Next Response can be returned signaling the end of the API call. The remaining code in the handleSubmit function uses the React use states to reset all the comment variables for the next comment made.

The same client and server components go for redirecting users based on their actions. Within server components, it is best to use the Next.js navigation and redirect users. However, in client components, you must use a React router to redirect users. When I began using the Next.js

navigation, I did not realize that it was limited to server components. I kept trying to use them in client components and got more component-based errors.

Next.js and Supabase work nicely together. The example project saved a lot of time with my project's authentication. Instead of setting up communications with Supabase myself, all the code had already been established. It just required me to add my own personal touches to the styling to make it match with the rest of my website, saving me lots of stress and time.

To do all the styling of my website, I used Tailwind CSS. It is a CSS framework that allows styling directly within HTML by using utility classes. Learning Tailwind CSS was very simple since the utility classes are reminiscent of typical CSS but abbreviated. Something I struggled with was when I had multiple, similar components across my website. It was tedious to ensure that every similar component had styling that matched one another. I turned to making my own styling classes within the global.css file for things like button styling to help ensure that the styling would remain consistent across all components. While it is easy to use, I do feel like it makes my code very messy as a result of the inline styling.

I faced many struggles when working on this project. Most notable was mixing up my client and server components. This came back to the fact that I did not understand the concept of client and server components and the ways they worked together to make a functioning application. Since I haven't dealt with a full-stack framework like this before, having both server and client components was something that I struggled to grasp. In hindsight, the concepts of the two components make a lot of sense, as the purpose of these components are in their names. However, a sense of stress and urgency to get my project completed pushed me to make poor decisions with my code.

Once, I had tried to modify the sign-up page to include a username input for the new user. I searched for where I had included an incorrect import into the auth components, but I could not find where it was. In my troubleshooting, a few sources online said to try updating my NPM package. After attempting to update my NPM package, when I tried to start my application again, it said my package had been corrupted. Incredibly frustrated, I decided to start my project from scratch with the newest version of NPM and bring my old application's components over one by one. So, I began the process of creating a new Next.js and Supabase project and brought over each part of my application. When I started it up for the first time, the newer NPM version gave me the same error but with the exact location where I had messed up. At the top of one of my auth components, I added a "use client" to try to get it to add data to my user profile table, but it needed to stay a server component. With my old project's NPM being corrupted, I decided to continue working on the newer project instead of troubleshooting my old project.

Despite a lot of things that went badly, some things went well. Since I had a lot of experience with database design, I felt confident going into designing the database for my project. Having to take the time to think through all the data that was going to be going in and out of my application helped me to grasp a better scope of what I wanted to do. I started by drawing up mockups of what each page was going to look like to understand what was going to need to be stored. I then broke down each page into its own separate tables. Then I transferred all my tables ideas into Supabase through their SQL editor.

Getting to dive into Tailwind CSS after spending hours writing all the base functionality of my components revitalized my energy for the project. Putting the final details on my project to make my project come to life quickly became a rewarding process. The CSS framework was easy to understand and helped to make all my components share similar visuals.

Once I understood how to make client and server components function together well, adding new components like deleting blog posts became much easier. Instead of putting multiple hours into one ability of my application, I was able to complete a new function within an hour. This also helped to clean up my code and make it more readable.

This is not the end of my blind box collecting website. I want to continue to rework my application to ensure that all my components are made as simple and efficient as possible. I am planning to add a follower system in the near future since I was unable to add it during the span of this semester. I believe this would be an easy feature to add since it would only require one table to be added that holds information about a user's accounts that they follow. On the blog page, a second view could be added that toggles between a view for all blog posts created and blog posts created by a user's followed list. As my friends use this website, I want to hear their thoughts and wants for this application to keep improving the user experience.

I have learned a lot about the Next.js framework, though I don't feel as confident as I could be. Supabase has quickly become one of my favorite applications and will be the first tool that I gravitate to when starting another project of this scale. Though I feel I am nowhere near the end of my blind box collecting website, I am very happy and proud of how far it has come this semester.