# RESUME

* Used interfaces for props.
* Established a set of distinct features for server and client components.
* Made a lot of reusable components.
* Used React.js with TypeScript with Tailwind CSS to code the frontend of the application.
* We created .env.local file to store the API keys from the supabase.
* Made environment variables that store next\_public\_supabase\_url, and more.
* Created tables using the table editor of the Supabase with all the required entries determining the primary and foreign keys.
* Create RLS policies that provide policies for read access, insert access, delete access etc.
* Created public storage buckets to upload and store our songs.
* Made personal access tokens with Supabase CLI to ensure permissions.
* Generated types for typescript using Supabase CLI so that we can generate a file that includes all the database in typescript language.
* Incorporated abstraction by creating ‘MyUserContextProvider’ separately that manages the UserContext.Provider and supplies user-related data such as ‘accessToken’, ‘user’, ‘userDetails’, ‘isLoading’, and ‘subscription’ to all the descendants of the ‘UserProvider’ component.

# Installs

* npm install react-icons
* npm install tailwind-merge
* npm install [supabase@”>=1.8.1](mailto:supabase@)” –save-dev
* npx supabase login

# sidebar component

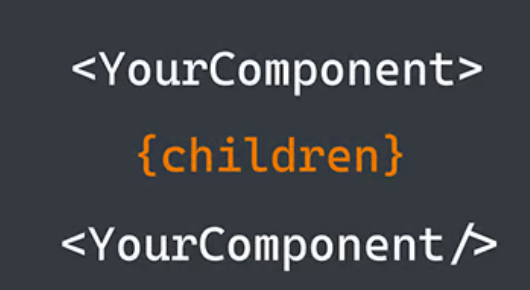
* Created a folder called “**components**”.
* Create a file called “Sidebar.tsx” and made stateless function inside. *(Code snippet: sfc)*

## ReactNode

* **React.ReactNode** represents an array of react elements, Boolean, number, or a string.
  + It defines the type of data a variable can hold inside.
* Most of the time a JSX component is closed.

< Component />

* + But we use the {children} prop when the component is not closed.



## Interace Sidebarprops

* “**interface** **SidebarProps**” basically is type checking mechanism. In TypeScript, interfaces are used to describe the structure of objects or classes. They define a set of properties and their types that an object must have in order to be considered compatible with the interface.

## React.fc

* **React.FC** is used to define the functional component. It is not mandatory and there are other ways to define a functional component.
  + Basically, it provides additional type checking.
  + So, the props that are coming to the component must be of type {children} which is of type ReactNode.
  + It’s not universally used as it provides additional constraints and non-standard props cannot be passed.

## Server Components

* **Server Components** get rendered on the server itself and are static.
  + They don’t need interaction or don’t need any state and don’t change are server components.
  + They cannot use useEffect, useStates etc. inside of a server component.
  + All the API calls, database calls happen on the server side, and it makes the site faster.
  + Environment is kept a secret on the server.
* **“use client”** is used as a convention. It declares a boundary between a server and a client component. Basically, we use “use client” where we need to use react hooks. The files using “use-client” are considered to be client-side components.
  + It is the proper way to pass server components inside of a client component.

## use memo

* This hook is used to memoize the result of a computation and use it for future purpose.
* Since the dependency array is empty, “[ ]”, the routes are only computed once and are reused in the subsequent renders optimizing performance.
* Basically, the routes are not rendered again and again because they don’t change throughout the application.

# Header Component

# Button component

## forward ref

* ‘forwardRef’ function is used to forward a ref to a child component, allowing parent components to interact with DOM elements or components defined within the child component.
* The button component uses a forwardRef which allows the ref to be forwarded from the parent component to the underlying <button> element.

# supabase

* We created .env.local file to store the API keys from the supabase.
* Made environment variables that store next\_public\_supabase\_url, and more.
* Supabase is excellent because it provides us a variety of features like table editor, sql editor, database viewer and has a separate tab for authentication.
* We can use supabase for resetting the password of the user, removing the user, email templates, providers etc.
* Stripe Subscription Template provided by Supabase was used to maintain a database for the user, our songs etc.
* This template is used to create table that are handy for managing Stripe payments and user database.
* There are row level securities that ensure that only the user can view, update, delete their data and no one else. This is done with the help of auth.id().
* Created tables using the table editor of the Supabase with all the required entries determining the primary and foreign keys.
* Created RLS policies that provide policies for read access, insert access, delete access etc.
* Created public storage buckets to upload and store our songs.
* Made personal access tokens with Supabase CLI to ensure permissions.
* Generated types for typescript using Supabase CLI so that we can generate a file that includes all the database in typescript language.
* createClientComponent is used to create a Supabase Client Instance for authentication purposes in Next.js.
* SessionContextProvider is used to provide Supabase Client instance to the session context, to access and utilize authentication related functionalities.

# hooks

* Using the interfaces created in types.ts, different user hooks are created.

# stripe

* Created interfaces for Stripe URLs and useUser hooks that represent different data structures related to user details, products, prices, and subscriptions in the context of a payment system integrated with the Stripe API.

# hooks

## useuser

* UserContext created using ‘createContext’ will hold user-related data.
  1. The initial value is set to undefined.
  2. UserContextType defines the structure of the data to be stored in the userContext.
  3. When userContext is created using useContext (react), two components are provided.
     1. Provider Component wraps the React Component Tree and makes userContext available to all the components inside the tree.
     2. Consumer Component allows the components to access the context values within the rendering tree.
* Inside the MyUserContextProvider component, it uses the useSessionContext and useSupaUser hooks from the @supabase/auth-helpers-react library to access the user session and user information.
  1. MyUserContextProvider acts as the provider for UserContext. So, all the values inside the MyUserContextProvider will be available to all the components wrapped inside it.
* Two functions, **getUserDetails** and **getSubscription**, are defined to fetch user details and subscription information from a Supabase database.
* The useEffect hook is used to handle the side effects when the user or loading status changes. It triggers the fetching of user details and subscription if a user exists and the necessary data is not already loaded. If the user is not logged in, it resets the user details and subscription data.
* The value object is created to hold the user context values, including the access token, user object, user details, loading status, and subscription information.
* Finally, the MyUserContextProvider component wraps its children with the UserContext.Provider component, passing the value object as the context value.
* **Promise.allSettled** function is used to execute two promises concurrently: getUserDetails(), getSubscription(). These promises are responsible for fetching user details and subscription information form a supabase database, respectively.
* So, when the promises are fulfilled then the data of the user and their subscription status is stored in userDetailsPromise and subscriptionPromise which are then used to update the state variables userDetails and subscription.
* **Promise**.**allSettled** function is used to fetch details and subscription information from a Supabase database using promise. It updates the respective state variables with the retrieved data and manages the loading status.
* Basically, MyUserContextProvider is responsible for creating and providing the user context and supplies the user-related data such as ‘access token’, ‘user’, ‘userDetails’ etc. to the components that are descendants of the ‘UserProvider’ component. These components can consume the user context data using the ‘useContext’ hook provided by React.

# providers

## UserProvider

* MyUserContext provider is used as a separate component because it helps us isolate the logic and functionality related to managing the user context in one place.
* UserProvider component abstracts the details of the ‘MyUserContextProvider’ implementation which makes the code readable and only exposes a higher-level interface for utilizing the user context.

## MODalprovider

* We never want to render a modal if we are in server-side rendering.
* useEffect only runs once because of the empty dependency provided and works like componentDidMount in class components.
* If our apps use modals or other components (dropdowns, tabs, accordions etc.) that require interactivity, you may want to ensure that they are rendered and functional only on the client side.