Genius Documentation

1. Description

'Genius' is an AI-driven chat application that integrates a variety of AI functionalities into a user-friendly platform. It makes use of a suite of modern technologies including TypeScript, Tailwind, Next.js, Prisma, MySQL, and Clerk. The application offers interactive AI tools such as OpenAI-powered conversations, image generation, music generation, code generation and video generation. This documentation outlines the application's structure, its AI integration points, and the user flow, providing insights into the technical and design decisions behind the project.

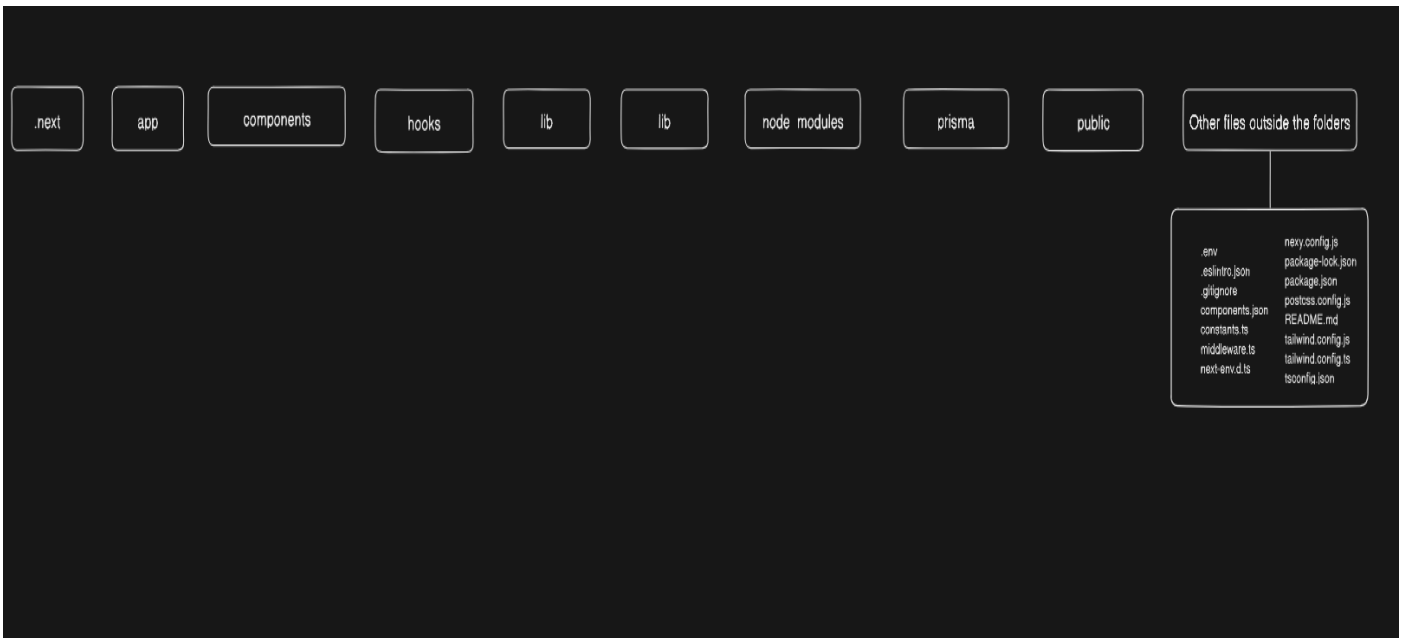
2. Objectives

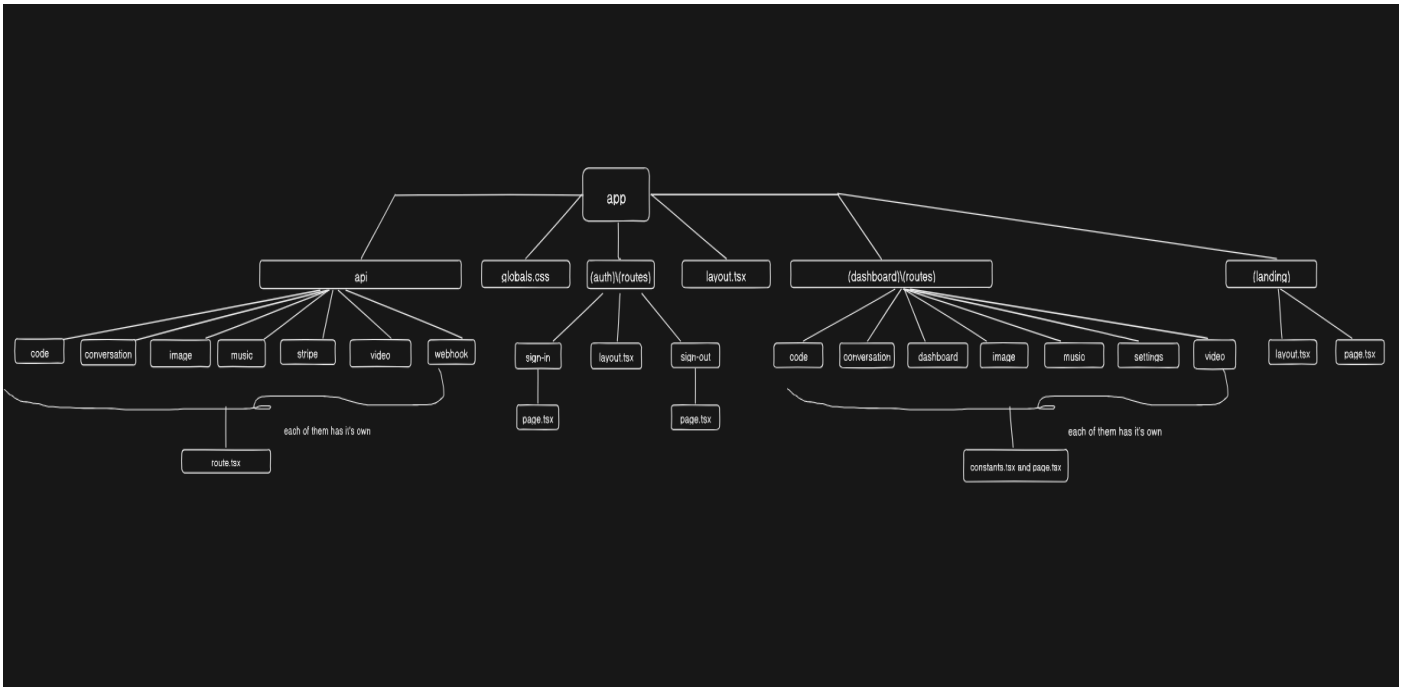
The goal of this project is to showcase the power of modern tools in an elegant matter. The main objective of this project are as follows:

- **Deliver an Engaging User Interface**: To create a visually appealing and intuitive experience for all users.
- **Harness the Power of Advanced AI**: Integrating sophisticated tools to explore the vast possibilities in AI-driven applications.
- **Showcase Versatile Functionality**: Demonstrate the range of features, from chat interactions to media generation, highlighting the application's diverse nature.
- **Ensure Sustainable Usage**: Implement a tiered access system to balance user accessibility with the sustainable use of AI resources.
- **Provide Support**: Offer robust customer support integration, enhancing user engagement and satisfaction.

3. Design and Code

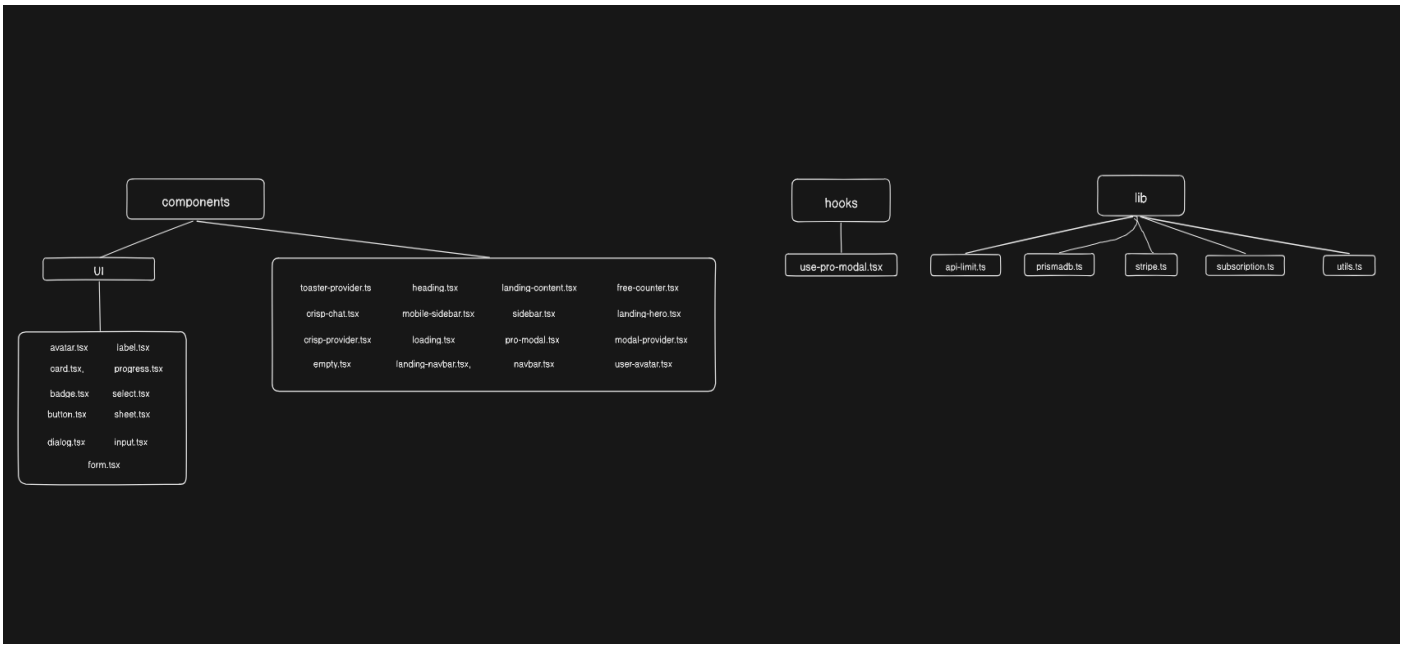
Genius is developed in Visual Studio Code, featuring a structured Next.js framework that enhances maintainability and scalability. The project is organized into several key directories, each serving a distinct purpose.





The ‘app’ folder is the heart of the application, containing subdirectories like ‘auth’ for authentication processes, and various modules such as ‘conversation’, ‘code’, ‘image’, ‘music’, ‘settings’, and ‘video’, each dedicated to a specific feature of the platform.

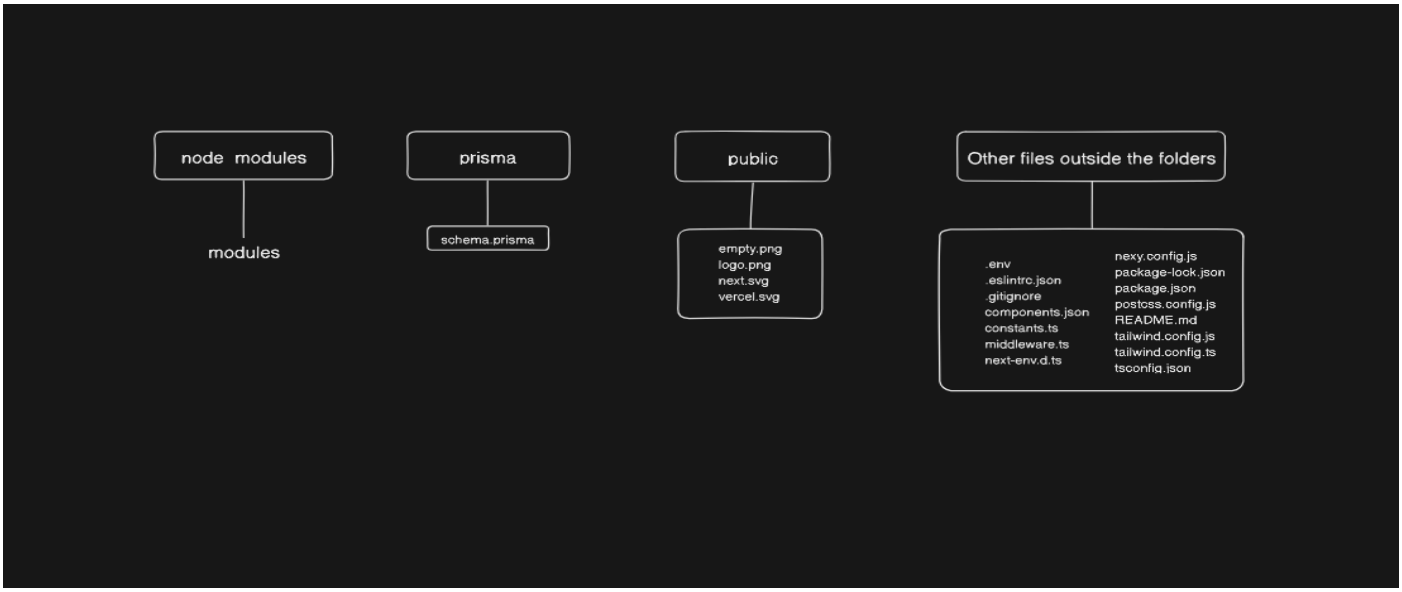
Up next we have the ‘components’, ‘hooks’ and ‘lib’:



Within ‘components’, you'll find reusable UI elements that ensure a consistent look and feel across the app, whereas ‘hooks ‘ contain custom React hooks that abstract component logic for cleaner code.

The ‘lib’ directory holds utility functions and libraries that are central to the application's operations, providing a repository of shared logic.

Lastly, we will take a look at the following folders and files:

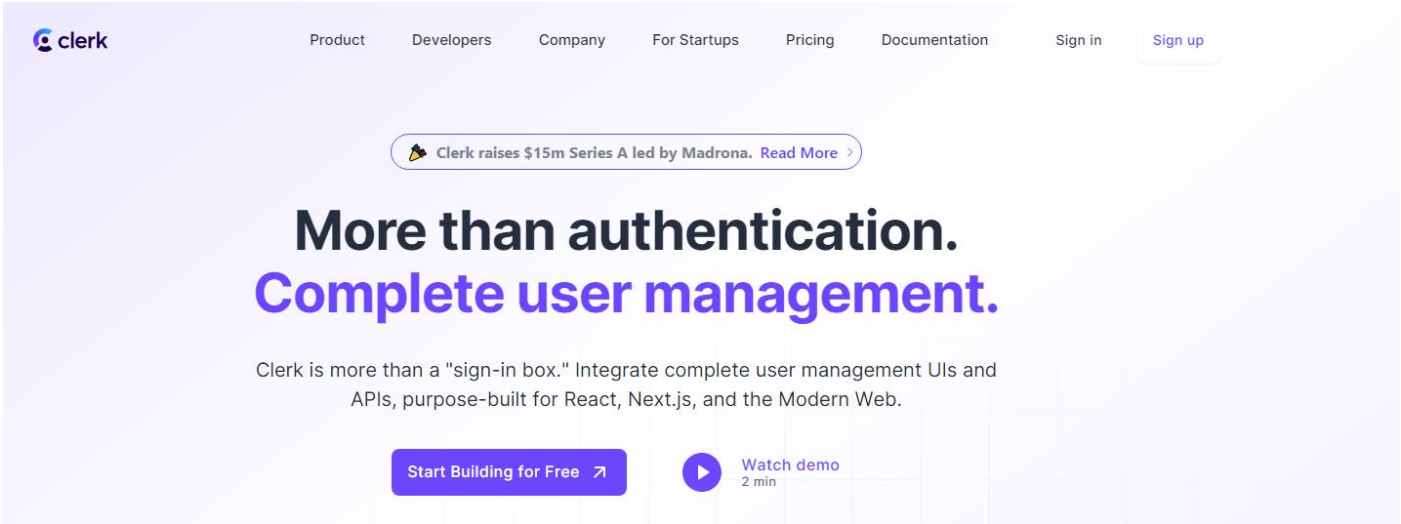


'node_modules' is where all the dependencies of the project are located, 'prisma' contains the ORM schema for database interactions, and public is for static assets like images and icons.

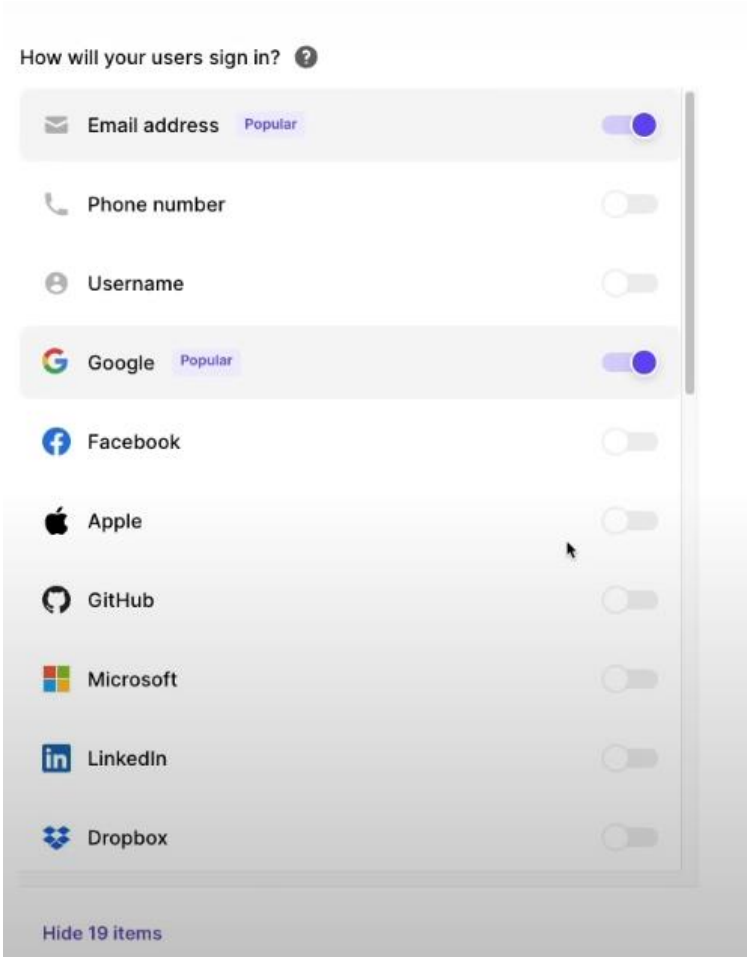
Finally, the root level includes configuration files like 'next.config.js' and 'package.json' , essential for customizing the Next.js setup and managing project dependencies.

For this application, shadcn/ui was used for styling, the documentation is here https://ui.shadcn.com/docs/installation/next . The app router from next.js 13.4 is used to make directories in the app folder. By doing so every folder in the app folder will be a route if the page inside of it will be called "page.tsx".
For example, if inside my app folder I will have the folder "dashboard" -> "page.tsx" which returns a div saying "Hello World", to be able to see that in your project you would have to access https://www.localhost:3000/dashboard. There is a way to use folders name just as an organizational tool and you can bypass the app router by writing the folder name in " () ". You can read more here https://nextjs.org/docs/app/building-your-application/routing/route-groups

The first thing I did was adding authentication for the project using clerk. After you make an account on their page https://clerk.com/

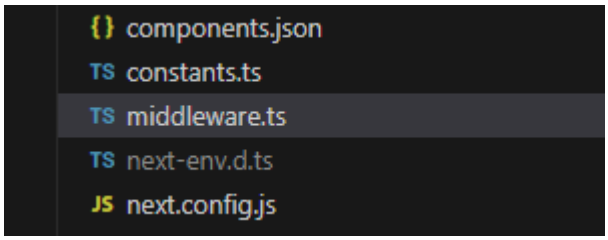


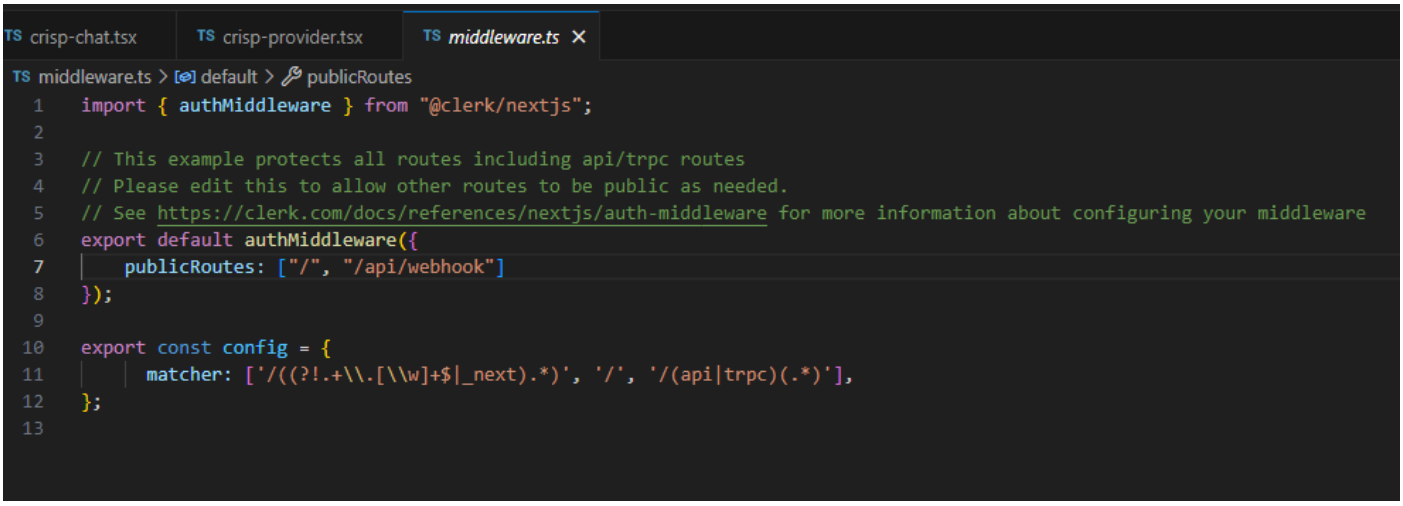
You can build your sign-in and customize it by choosing sign in methods. For this project only 'google' and the email address were used



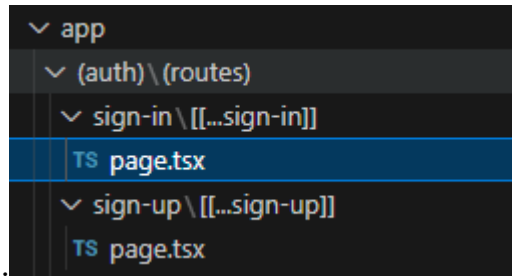
After this step you got your public clerk publishable key and the secret key. These keys will be held in the .env file of the project and the keys will be used by clerk to enable authentication.

You will have to install clerk's package into your project first, more details here: https://clerk.com/docs/quickstarts/nextjs and add the <ClerkProvider> wrapper inside the app/layout.tsx like in the documentation. Inisde your middleware.ts you can decide which pages are public or protected by the authentication



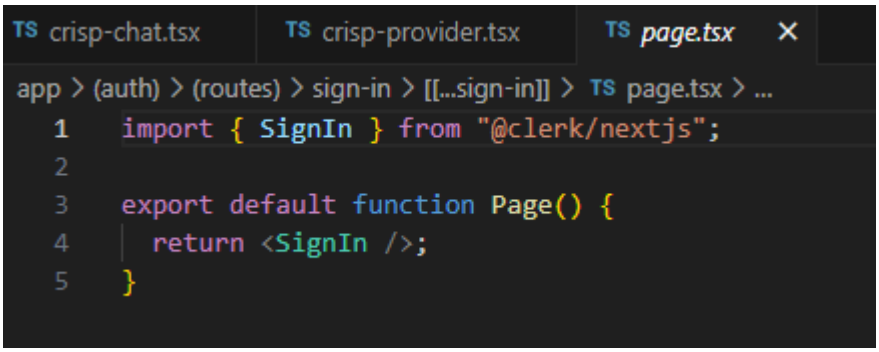


The last step to do is to create your sign-in and sign-out components, in my case they are in the route group (auth)/(routes)/sign-up/[[...sign-up]]/page.tsx, the same thing for the sign-out

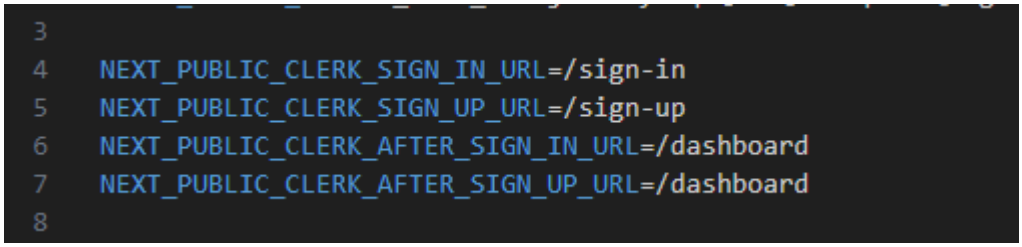


The [[...sign-up]] is a convention used In next 13 which allows clerk to use all the necessary routes for authentication. It is called “Optional Catch-all Segments” https://nextjs.org/docs/app/building-your-application/routing/dynamic-routes

The sign-in, and sign-out components look like this:

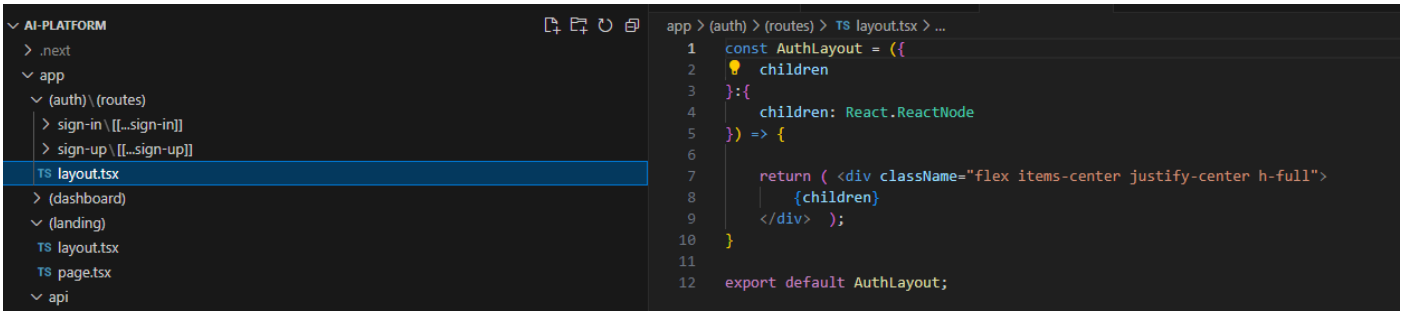


Now in your env file you can choose where to be redirected after signing up.

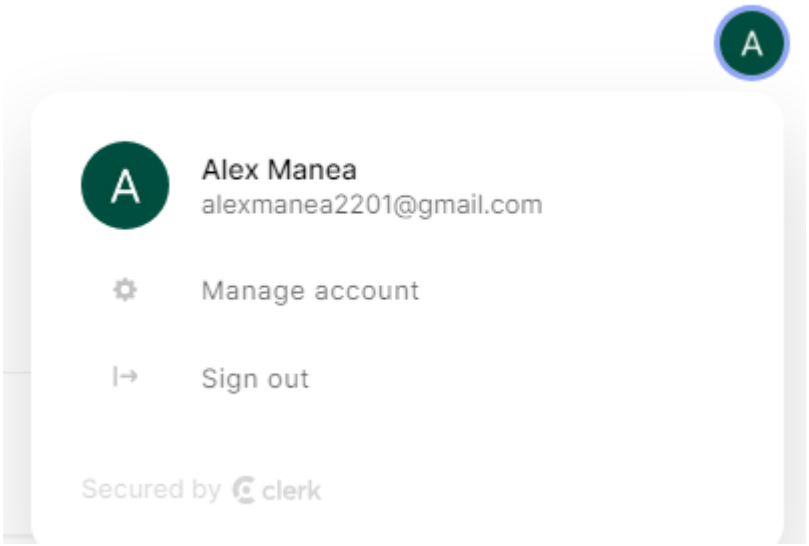


If you take a look at the folder structure inside the app router in the photo below, you will notice that the user will first land on (landing)/page.tsx. The page can’t be accessed because it’s not protected by our middleware as it can be seen above.

To style and place the sign-in and sign-out a layout file is added into the auth route group which will control the placement of the clerk authentication component.



Now from the landing page buttons can be added to redirect the user to signing in or signing up and by importing {UserButton} from “@clerk/nextjs” you can sign out



You can even choose where the user to redirected after

```
<UserButton afterSignOutUrl="/"/>
```

Up next we can take a look at the dashboardlayout which is contained inside (dashboard)/(routes):

TS crisp-chat.tsx | TS crisp-provider.tsx | TS layout.tsx

app > (dashboard) > (routes) > TS layout.tsx > DashboardLayout

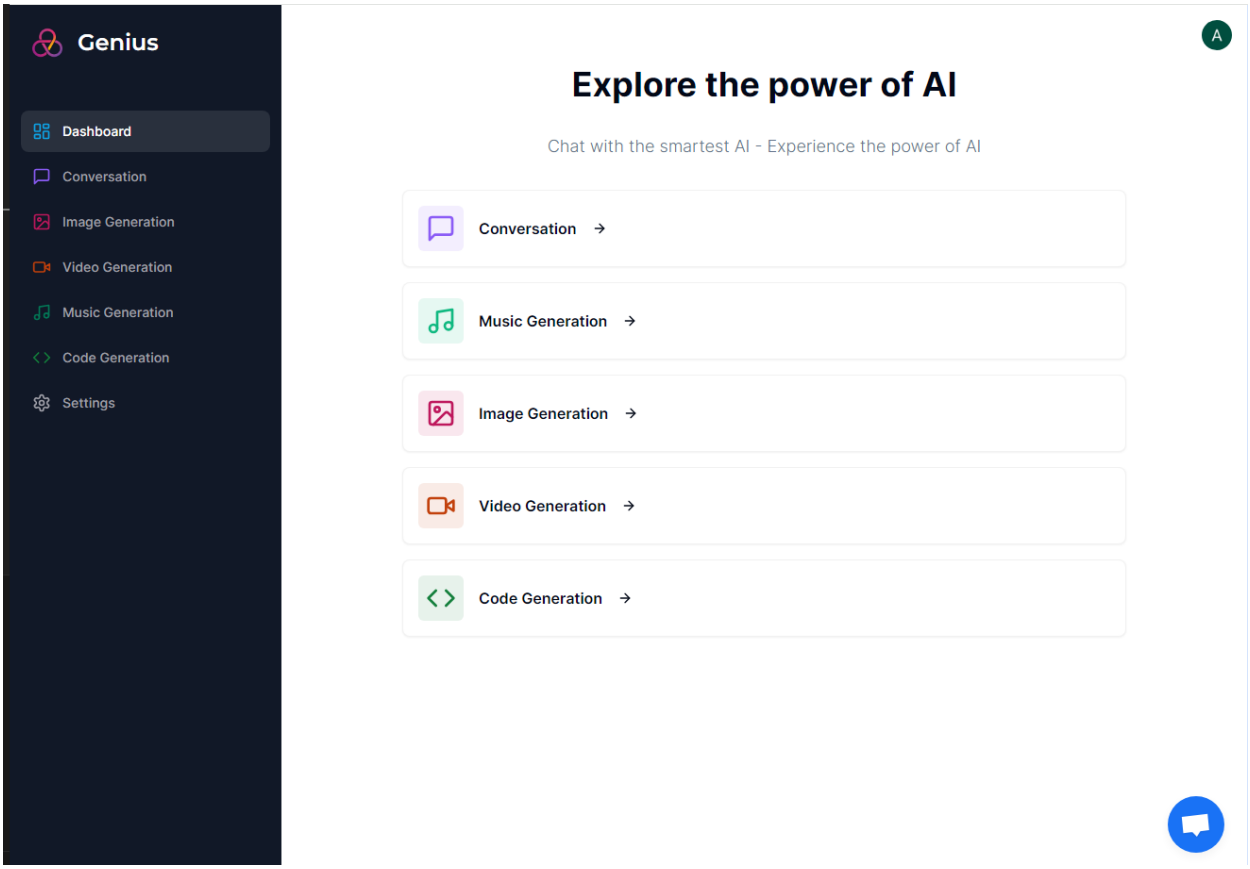
```tsx
import Navbar from "@/components/navbar";
import Sidebar from "@/components/sidebar";
import { getApiLimitCount } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

const DashboardLayout = async({
    children
} :{
    children: React.ReactNode
 }) => {

    const apiLimitCount = await getApiLimitCount()
    const isPro = await checkSubscription()

    return (
        <div className="h-full relative">
            <div className="hidden h-full md:flex
            md:w-72 md:flex-col md:fixed md:inset-y-0
               bg-gray-900">
              <Sidebar apiLimitCount={apiLimitCount} isPro={isPro}/>
             </div>
                <main className="md:pl-72">
                        <Navbar/>
                        {children}
                </main>


        </div>
     );
}

export default DashboardLayout;
```

The ‘DashboardLayout ‘component is important for the application, acting as the structural framework for the dashboard interface. It uses components such as ‘Navbar’ and ‘Sidebar’ , which are imported from the components directory to provide navigation and user interface in the dashboard. On mobile the ‘Sidebar’ will be hidden.

The layout uses two important functions, ‘getApiLimitCount’ and ‘checkSubscription’ from the lib directory, to fetch the current API usage count and determine if the user has a pro subscription. Structured with responsive design, it has a flexible layout that adapts to various screen sizes, ensuring a seamless user experience.

The ‘main’ tag uses dynamic content (children), allowing for different components to be rendered as users interact with the dashboard, and it’s content will not overlap with the sidebar since it has a padding higher than the width of the sidebar.

You can see below the sidebar which is placed on the left side:

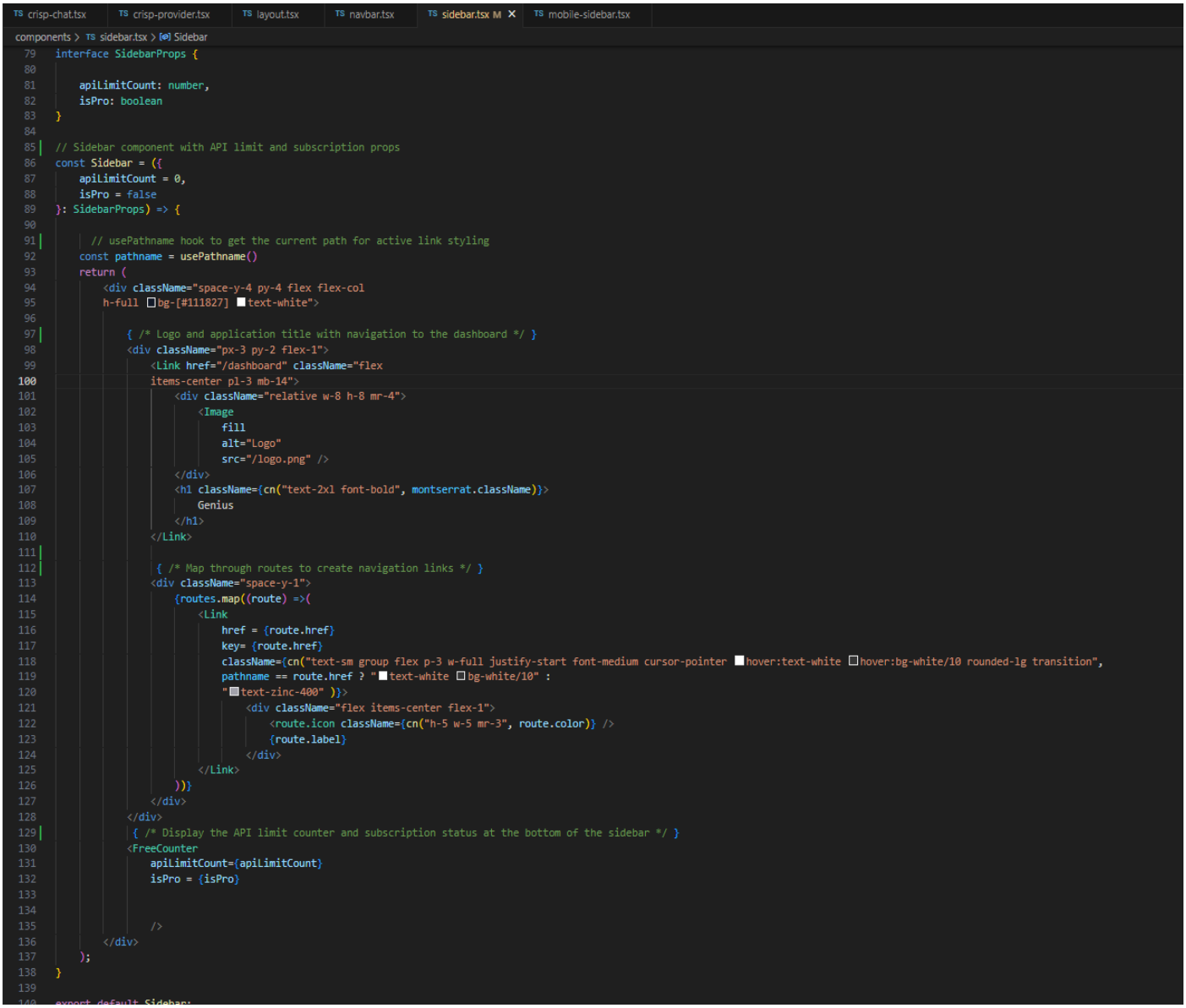


And here we can explore the code:

TS crisp-chat.tsx | TS crisp-provider.tsx | TS layout.tsx | TS navbar.tsx | TS sidebar.tsx M | TS mobile-sidebar.tsx

components > TS sidebar.tsx > Sidebar

```tsx
import { usePathname } from "next/navigation";

import { cn } from "@/lib/utils";
import {
    LayoutDashboard,
    MessageSquare,
    ImageIcon,
    VideoIcon,
    Music,
    Code,
    Settings
} from "lucide-react"
import { FreeCounter } from "@/components/free-counter";

// Define Montserrat font with specific weight for a consistent look
const montserrat = Montserrat({
weight: "600",
subsets:["latin"]})

// Define the navigation routes with their labels, icons, paths, and icon colors
const routes = [
    {
        label: "Dashboard",
        icon: LayoutDashboard,
        href: "/dashboard",
        color: "text-sky-500"

    },
    {
        label: "Conversation",
        icon: MessageSquare,
        href: "/conversation",
        color: "text-violet-500"

    },
    {
        label: "Image Generation",
        icon: ImageIcon,
        href: "/image",
        color: "text-pink-700"

    },
    {
        label: "Video Generation",
        icon: VideoIcon,
        href: "/video",
        color: "text-orange-700"

    },
    {
        label: "Music Generation",
        icon: Music,
        href: "/music",
        color: "text-emerald-700"

    },
    {
        label: "Code Generation",
        icon: Code,
        href: "/code",
        color: "text-green-700"

    },
    {
        label: "Settings",
        icon: Settings,
        href: "/settings",


    },

]
```



The component is the principal navigation mechanism within the dashboard. It is a client-side component and it’s styled with Tailwind CSS for a responsive and consistent user interface.

It uses Next.js's ‘Link’ and ‘usePathname’ for navigation between the app's features, wrapped in the user interface. The Sidebar's appearance has a dark background color set by a Tailwind CSS class and white text for clarity. The Layout and spacing are achieved using Flexbox.

Navigation through the app is defined by an array of routes as you can see above, each represented by a label for the display text, an icon component from ‘lucide-react’, a href attribute for the navigation path, and a Tailwind CSS class for the icon's color. The Sidebar differentiates the active route by comparing the current pathname with the route paths and applies different styling to the active route to enhance user orientation.

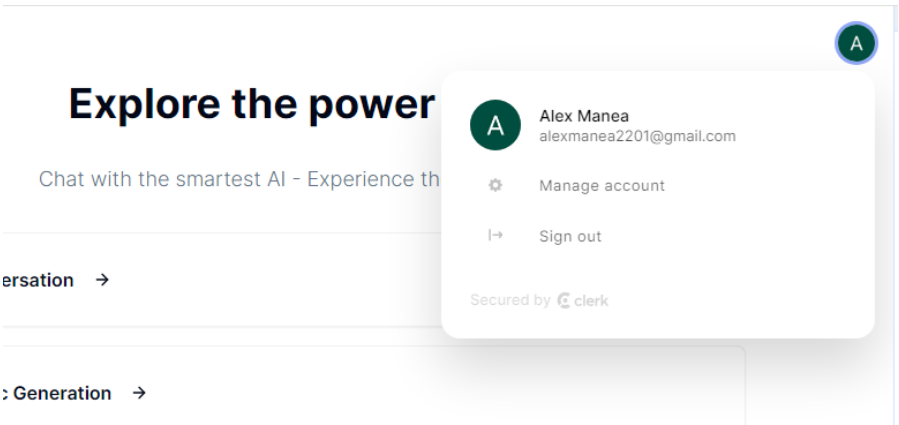
Further styling is done with the Montserrat font, particularly in the sidebar heading, designed to be consistent throughout the app.

The ‘FreeCounter’ component is integrated to provide a visual indicator of the API call limit, keeping the user informed of their usage. Additionally, the component checks the user's subscription status and adjusts the displayed content accordingly.

Progressing further, we can take a look at the ‘Navbar’ , if you remember from the layout.tsx, the sidebar will be visible only on medium-sized screens and bigger. We can take a look now at how the Navigation was implemented.

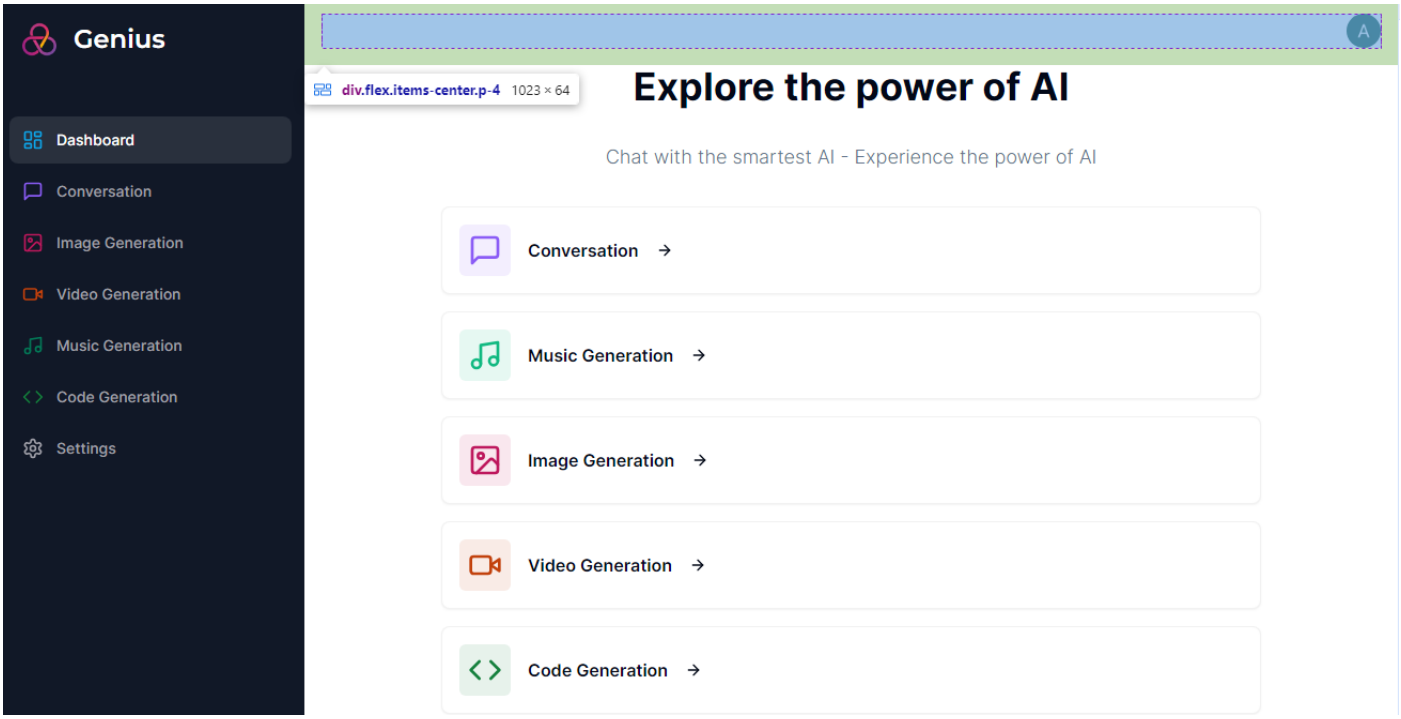
```tsx
import { Button } from "@/components/ui/button";
import { UserButton } from "@clerk/nextjs";

import MobileSidebar from "./mobile-sidebar";
import { getApiLimitCount } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

const Navbar = async () => {

    // Fetch API limit and subscription status
    const apiLimitCount = await getApiLimitCount()
    const isPro = await checkSubscription()

    // Render the Navbar with responsive design
    return ( <div className="flex items-center p-4">

                <MobileSidebar isPro={isPro} apiLimitCount={apiLimitCount}/>
                  <div className="flex w-full justify-end">

                    <UserButton afterSignOutUrl="/"/>
                  </div>
                </div>
     );
}

export default Navbar
```

The component serves as the primary navigation header. It implements the 'UserButton' from Clerk for user authentication and account management. This allows users to access their profile settings and handle sign-out directly from the navbar as you can see below.



Additionally, the component integrates the 'MobileSidebar' , ensuring that navigation remains functional and user-friendly, on mobile devices.

Styling with Tailwind CSS is applied to the Navbar for a consistent look and responsive design. It utilizes a flexbox layout for alignment and spacing, ensuring that the items are centrally aligned and appropriately spaced out for a clean and accessible user interface.



Now let's continue with the mobile sidebar inside the navigation:

crisp-chat.tsx | crisp-provider.tsx | layout.tsx | navbar.tsx M | sidebar.tsx M | mobile-sidebar.tsx M

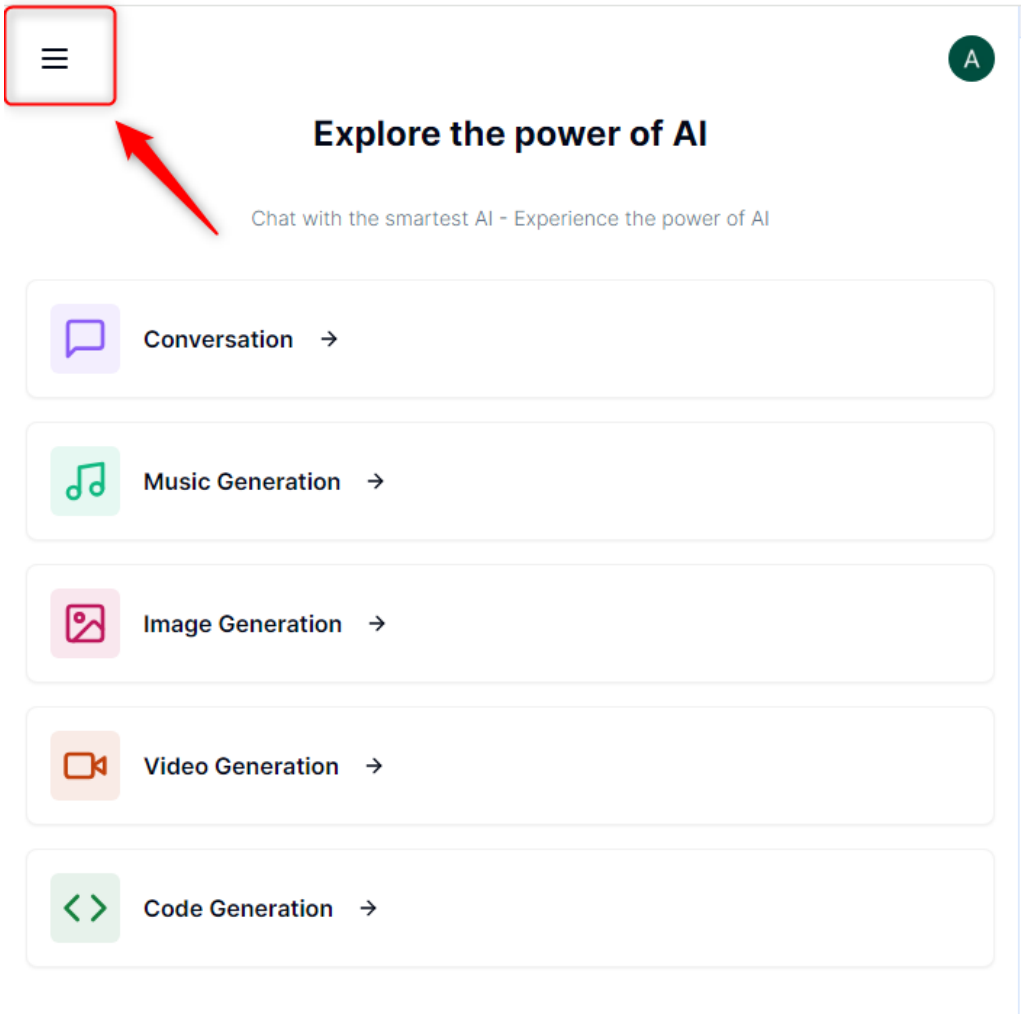
components > mobile-sidebar.tsx > MobileSidebar

```tsx
"use client"

import { Button } from "@/components/ui/button"

import { Menu } from "lucide-react"
import { Sheet, SheetContent, SheetTrigger } from "@/components/ui/sheet";
import Sidebar from "@/components/sidebar";
import { useEffect, useState } from "react";

interface MobileSidebarProps  {
    apiLimitCount: number
    isPro: boolean
}

const MobileSidebar = ({
    apiLimitCount = 0,
    isPro = false
}: MobileSidebarProps) => {
    const [isMounted, setIsMounted] = useState(false)

    // Conditional rendering based on mounting state
    useEffect(() =>{
        setIsMounted(true)
    },[])

    if(!isMounted)
        return null

    // Sidebar content with custom styling for mobile view
    return (
        <Sheet>
            <SheetTrigger>
                <Button variant ="ghost" size="icon"
                        className="md:hidden">
                            <Menu />
                </Button>
            </SheetTrigger>
            <SheetContent side="left" className="p-0">
                <Sidebar apiLimitCount={apiLimitCount} isPro={isPro}/>

            </SheetContent>
        </Sheet>
    );
}

export default MobileSidebar;
```

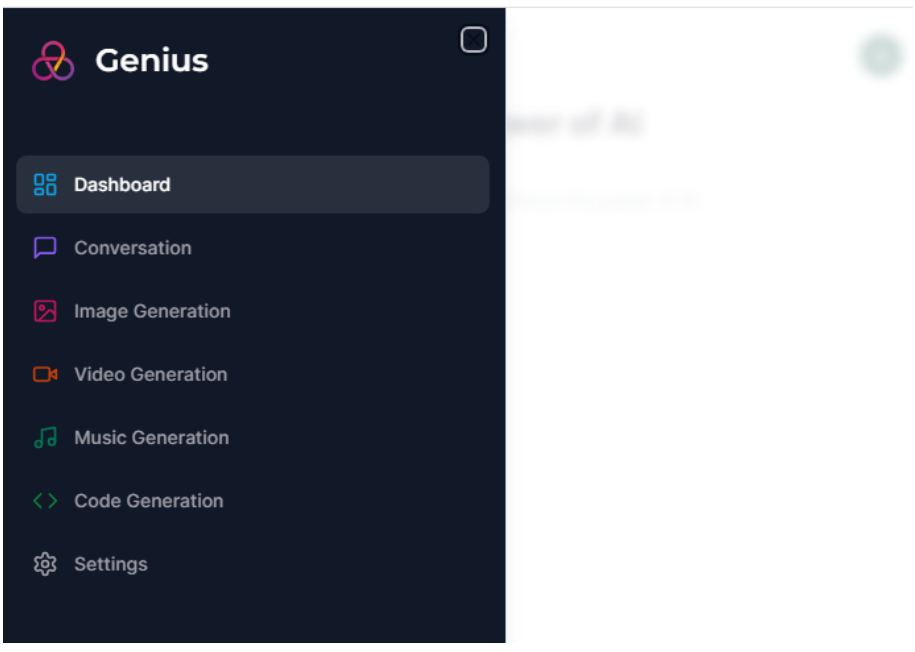
The ‘MobileSidebar’ component is important for improving the mobile user experience within the app. It provides a collapsible sidebar navigation menu and allows users to access the navigation options from mobile devices. The component uses the Sheet component from shadcn/ui (https://ui.shadcn.com/docs/components/sheet) to create a sliding drawer effect, which is activated by an interactive menu icon from lucide-react.

The layout and behavior are controlled using ‘SheetTrigger’ and ‘SheetContent’ components. These manage the opening and content display of the sidebar. React's ‘useState’ and ‘useEffec’t hooks are utilized for managing the component's mounting state.

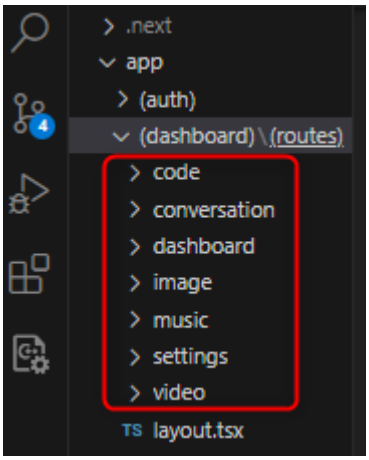
When the screen is small enough, we will see the menu button

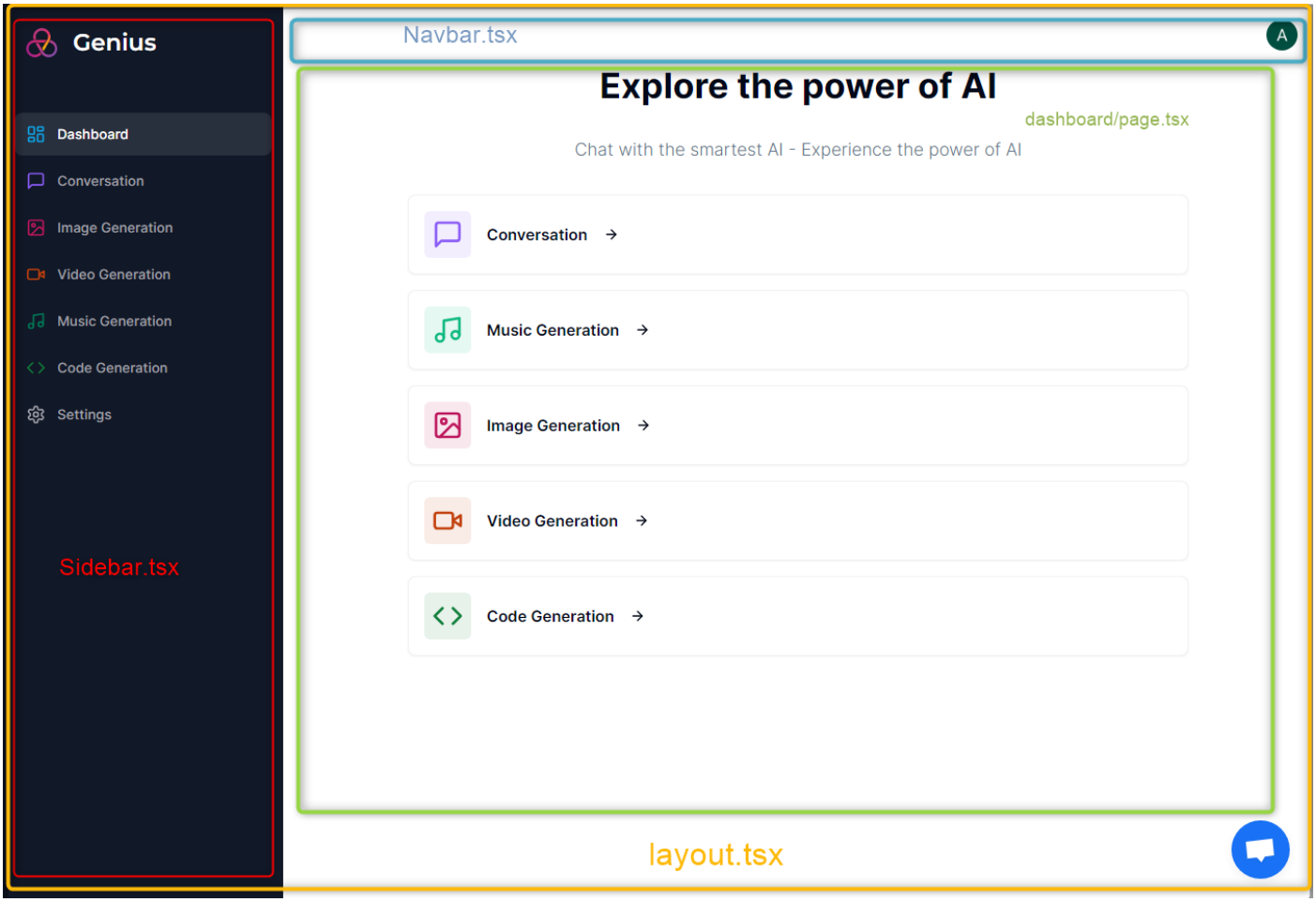


Upon clicking it will open the sidebar

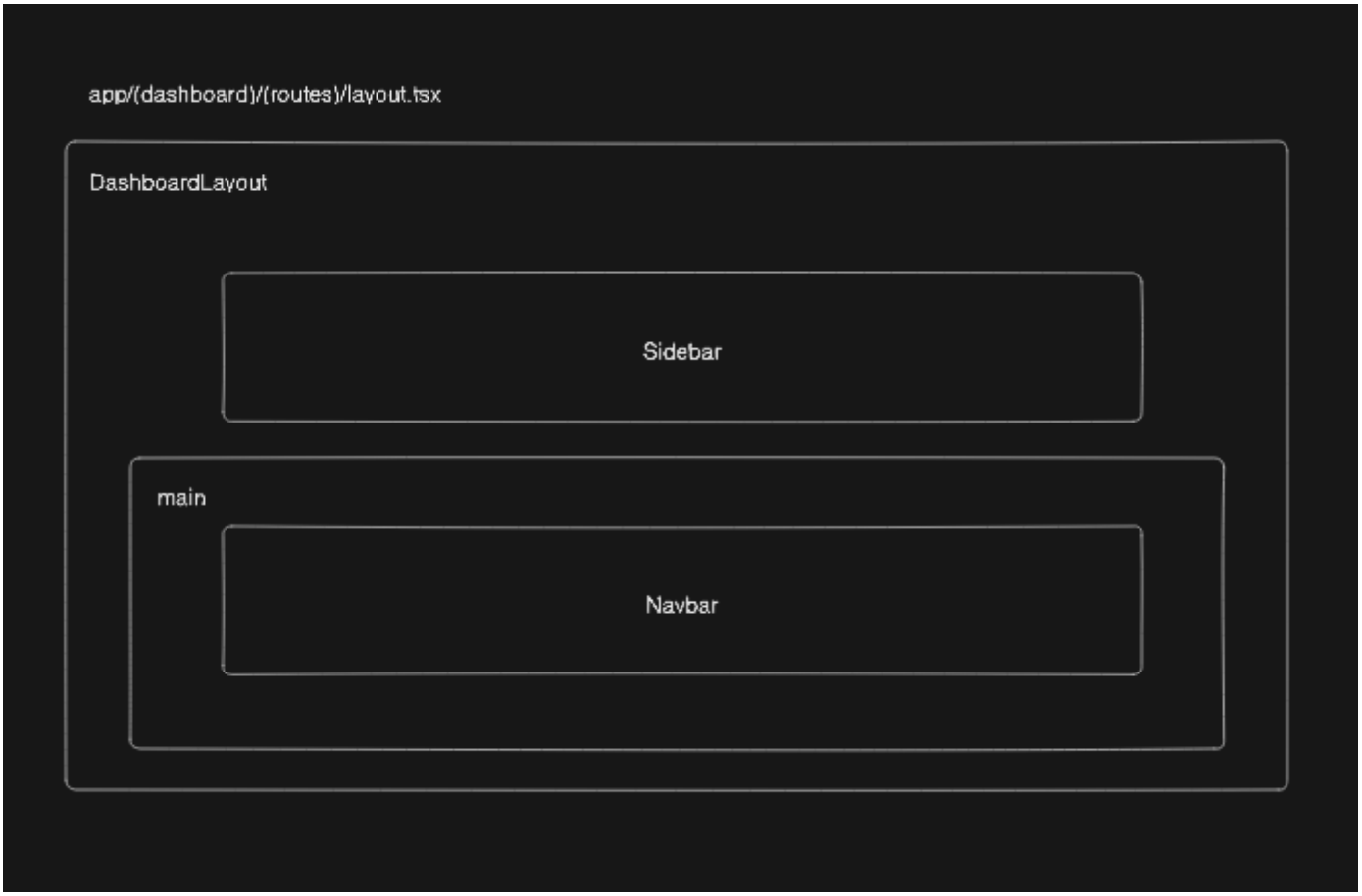


Alright, so now we covered the creation of the the layout.tsx file inside the app/(dashboard)/(routes) which will act as the wrapper for these folders:

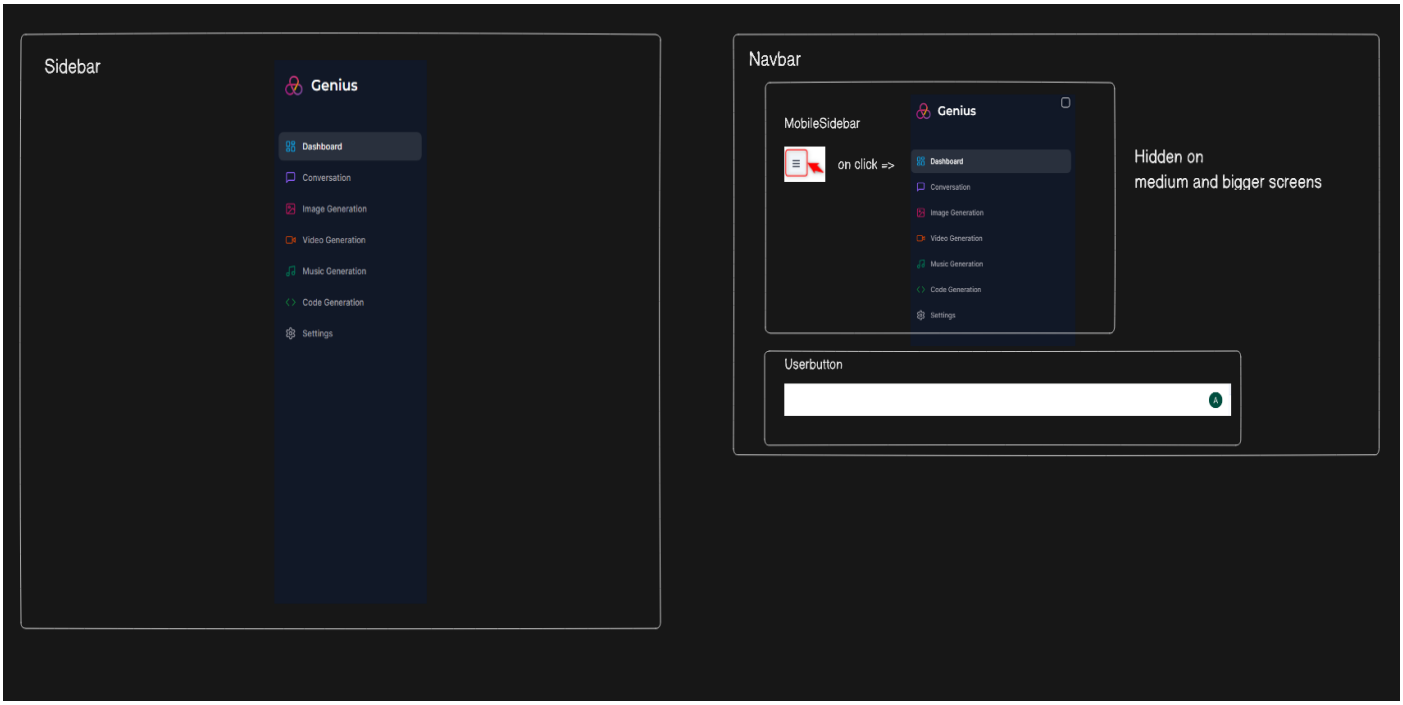




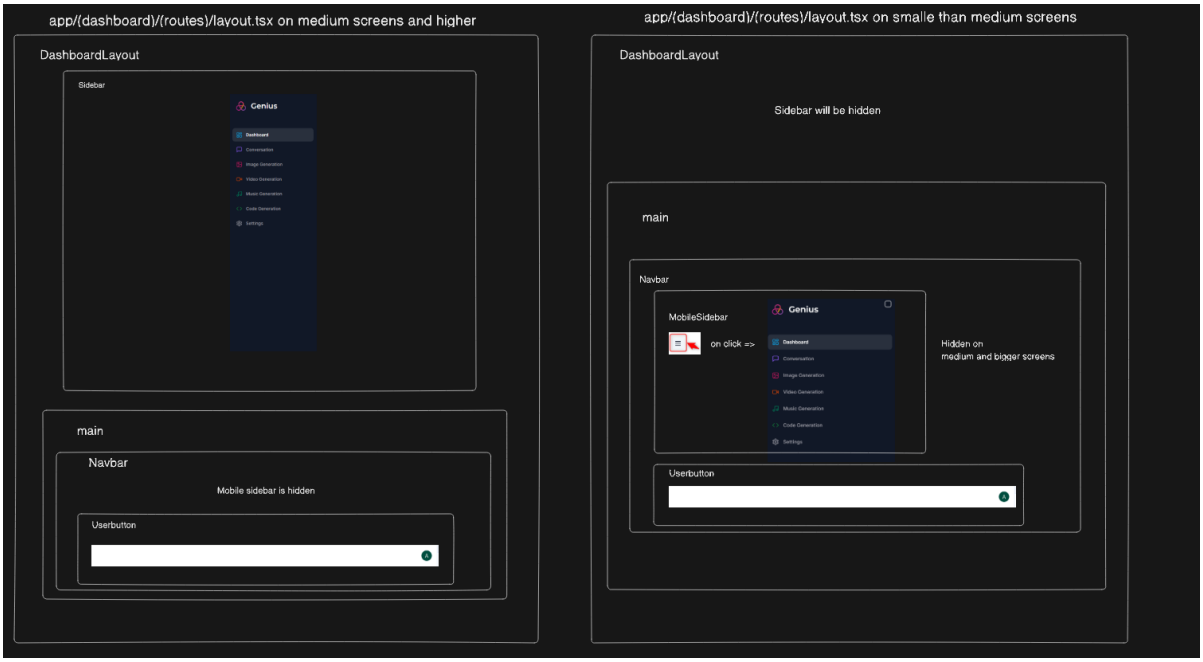
Another way of looking at what we covered is using some diagrams. First, we'll have the 'DashboardLayout':



And the contents of ‘Navbar’ and ‘Sidebar’



There will be two use cases, one when the screen size is bigger than medium size and the one where its smaller represented as follows



Another thing to remember is that inside the main component will be rendered the contents of the code, conversation, image, music, settings, video folders inside (dashboard)/(routes). This will be done through the children props.

Up next it can be observed through the interface that the application is structured around seven key elements. Each is designed to offer a unique functionality and improve the user experience. These components are important in defining the application's capabilities and user interactions. They are as follows: Dashboard, Conversation, Code Generation, Image Generation, Video Generation, Music Generation and the Settings Page.

Let’s start by taking a look at the first one.

Dashboard

```tsx
import { Card } from "@/components/ui/card"
import {
  MessageSquare,
  ArrowRight,
  Music,
  ImageIcon,
  VideoIcon,
  Code} from "lucide-react"

import { cn } from "@/lib/utils"
import { useRouter } from "next/navigation"

// Array of tools with  details used for rendering cards
const tools = [
  {
    label: "Conversation",
    icon: MessageSquare,
    color: "text-violet-500",
    bgColor: "bg-violet-500/10",
    href: "/conversation"
  },
  {
    label: "Music Generation",
    icon: Music,
    color: "text-emerald-500",
    bgColor: "bg-emerald-500/10",
    href: "/music"
  },
  {
    label: "Image Generation",
    icon: ImageIcon,
    color: "text-pink-700",
    bgColor: "bg-pink-700/10",
    href: "/image"
  },
  {
    label: "Video Generation",
    icon: VideoIcon,
    color: "text-orange-700",
    bgColor: "bg-orange-700/10",
    href: "/video"
  },
  {
    label: "Code Generation",
    icon: Code,
    color: "text-green-700",
    bgColor: "bg-green-700/10",
    href: "/code"
  },

]
```

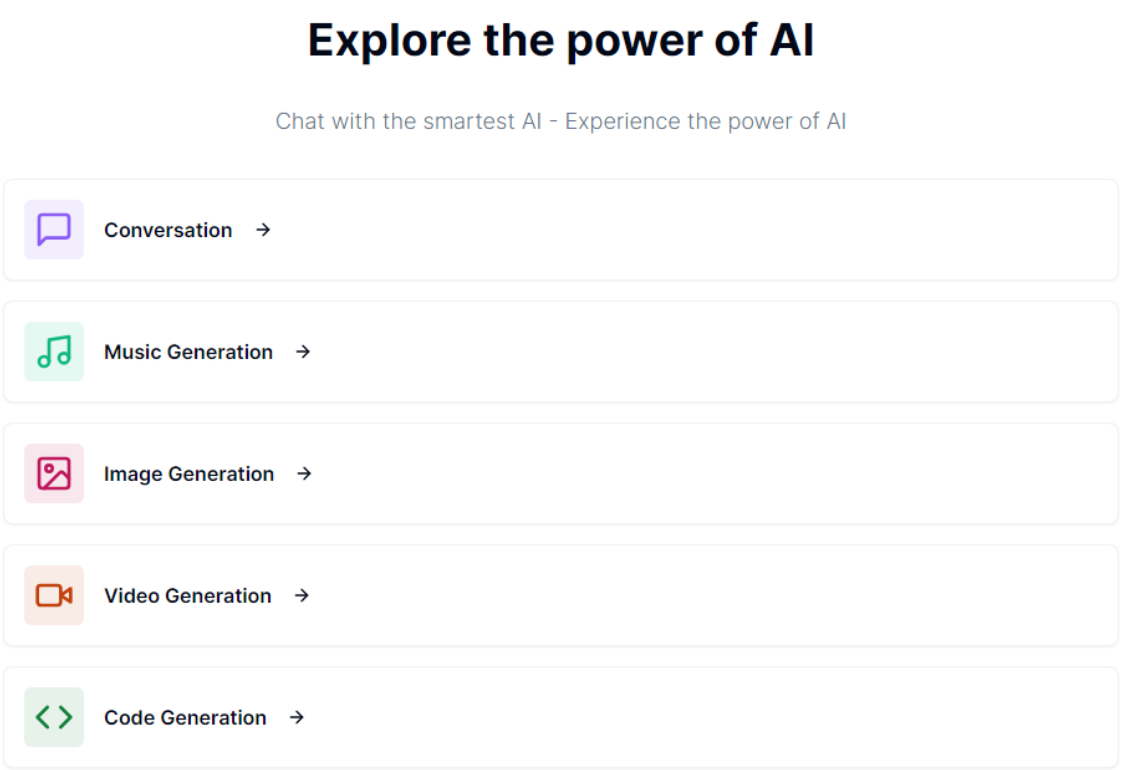
```tsx
]

const DashboardPage = () => {

  // useRouter hook for navigation on card click
  const router = useRouter()

    // Render the Dashboard page
  return (
    <div>
      <div>
        <div className="mb-8 space-y-8">
          <h2 className="text-2xl md:text-4xl font-bold text-center">
            Explore the power of AI
          </h2>
           <p className="text-muted-foreground font-light text-sm
          md:text-lg text-center">
            Chat with the smartest AI - Experience the power of AI
          </p>
        </div>
        <div className="px-4 md:px-20 lg:px-32 space-y-4">
          {tools.map((tool) =>(
                <Card
                onClick={() => router.push(tool.href)

              }
                key={tool.href}
                className="p-4 border-black/5 flex items-center
                justify-between hover:shadow-md transition cursorr-pointer">

                  <div className="flex items-center gap-x-4">
                    <div className={cn("p-2 w-fit rounded-md", tool.bgColor)}>
                      <tool.icon className={cn("w-8 h-8", tool.color)} />
                    </div>
                    <div className="font-semibold">
                      {tool.label}
                    </div>

                    <ArrowRight className="w-4 h-5"/>

                  </div>

                </Card>
            ))}
        </div>
      </div>
    )
}

export default DashboardPage
```

The ‘DashboardPage ‘ component is the main interface for users to interact with the various AI tools available. This page dynamically displays a selection of tools, such as "Conversation", "Music Generation", and "Image Generation", each represented by a card with a photo that users can interact with.

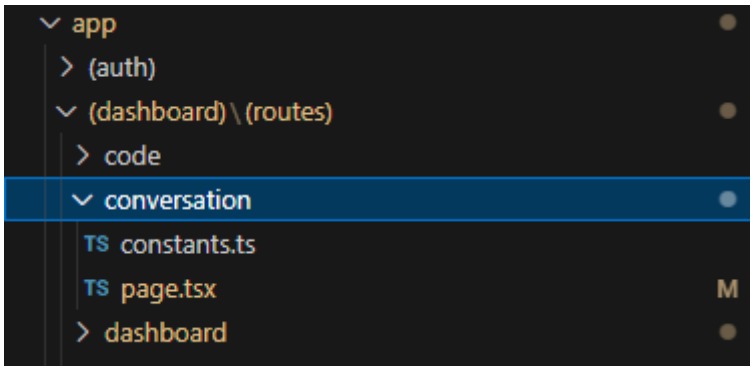
The navigation works like this: when a user clicks on one of the Card components, the’ onClick’ event handler is triggered. The *router.push(tool.href)* inside the ‘onClick’ handler tells the Next.js router to navigate to the route specified by *tool.href*.

For example, if a user clicks on the ‘Conversation’ card, *router.push* navigates to /conversation. This is similar to the user clicking a link that navigates to a new page.

The interface looks like below:



Now let’s examine each individual tool, starting with the conversation one. First things first is creating a folder called ‘conversation’ on the same level as the dashboard one.



Inisde the ‘conversation’ folder we will have 2 files, *page.tsx* and the *constants.ts* . Page.tsx will facilitate the AI conversation experience and it contains a form where users can input questions or statements. Through open’s AI model, responses will be generated. In the constants file will be the form schema used to validate the form

Now let’s dig deeper, and start with the page.tsx.

TS heading.tsx | TS constants.ts | TS page.tsx M X | TS navbar.tsx M

app > (dashboard) > (routes) > conversation > TS page.tsx > ...

```tsx
// Used for making HTTP requests to external APIs
import axios from "axios";

// zod: A schema validation library to validate  data structures
import * as z from "zod";

import { MessageSquare } from "lucide-react";

// Custom UI components: Heading for page titles, UserAvatar and BotAvatar for user and bot representations in chats,
// Empty for empty state messages, and Loading for indicating loading states.
import Heading from "@/components/heading"
import { UserAvatar } from "@/components/user-avatar";
import { BotAvatar } from "@/components/bot-avatar";
import { Empty } from "@/components/empty";
import { Loading } from "@/components/loading";

// Schema for form validation using zod
import { formSchema } from "./constants";

import { useRouter } from "next/navigation"

// Using zod to create our schema and validation
import { zodResolver } from "@hookform/resolvers/zod";

//Shadcn components and packages
import { Form, FormControl, FormField, FormItem } from "@/components/ui/form";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";
import { useForm } from "react-hook-form"

import { useState } from "react";

// Interface from the OpenAI package used to define the structure of chat messages
import  {ChatCompletionRequestMessage}  from "openai"
import { cn } from "@/lib/utils";

// custom hook for managing the modal related to the pro features of the app.
import { useProModal } from "@/hooks/use-pro-modal";

// used for user feedback like success or error messages.
import toast from "react-hot-toast";


const ConversationPage = () => {

    // Hook for modal and routing
    const proModal = useProModal()
    const router = useRouter()

    // State for managing messages in the conversation
    const [messages, setMessages] = useState<ChatCompletionRequestMessage[]>([])

    // Setup form with react-hook-form and zod for validation
    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
```

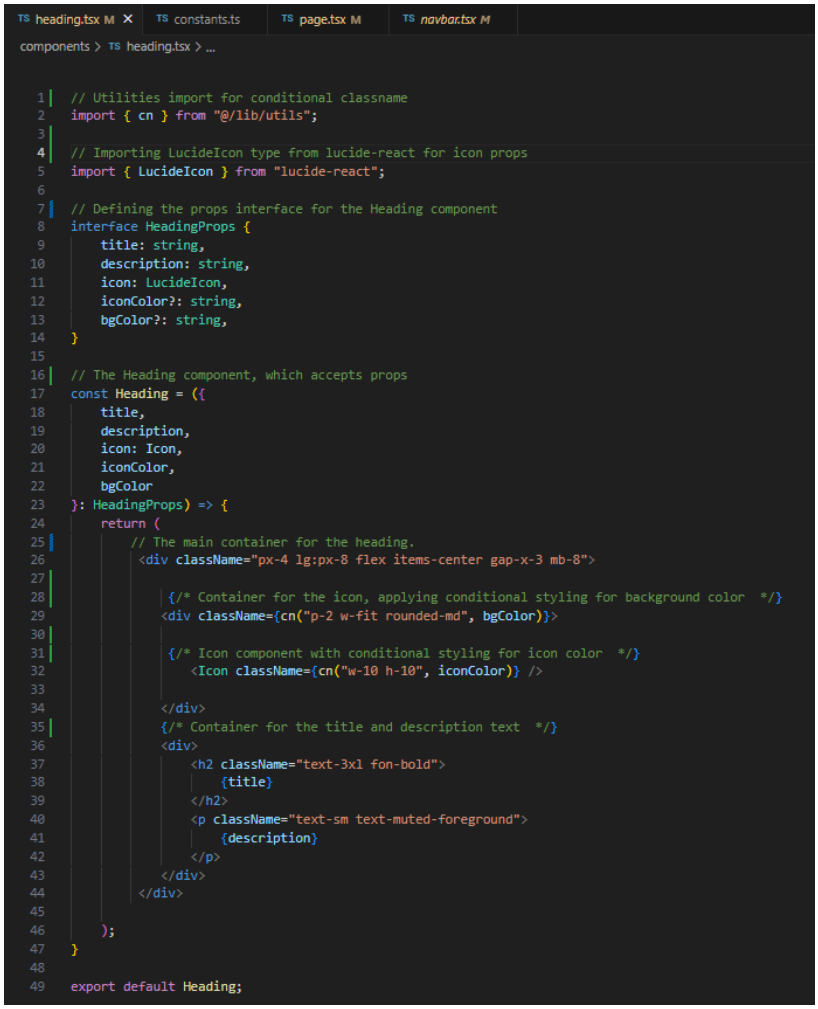
TS heading.tsx | TS constants.ts | TS page.tsx M X | TS navbar.tsx M

app > (dashboard) > (routes) > conversation > TS page.tsx > (e) ConversationPage > (e) onSubmit

```tsx
    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            prompt: ""
        }
    })

    // Flag for submission state
    const isLoading = form.formState.isSubmitting

    // Handling form submission
    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
        try{
            const userMessage: ChatCompletionRequestMessage = { role: "user", content: values.prompt };

            const newMessages = [...messages, userMessage]

            const response = await axios.post("/api/conversation", {
             messages: newMessages
            })

            setMessages((current) => [...current, userMessage, response.data])

            form.reset()

        } catch (error: any){
            if(error?.response?.status === 403){
                proModal.onOpen()
            } else {
             toast.error("Something went wrong")
            }
        } finally{
            router.refresh()
        }
    }
    // Rendering the conversation page
    return (
        <div>
            <Heading
            title = "Conversation"
            description = "Our most advanced conversation model."
            icon = { MessageSquare}
            iconColor = "text-violet-500"
            bgColor = "bg-violet-500/10 "
            />
            <div className="px-4 lg:px-8">

                <div>
                    <Form {...form}>
                        <form onSubmit={form.handleSubmit(onSubmit)}
                            className="rounded-lg border w-full p-4 px-3 md:px-6 focus-within:shadow-sm grid grid-cols-12 gap-2">

                            <FormField
                                name = "prompt"
                                render={({field}) =>(
                                    <FormItem className="col-span-12 lg:col-span-10">
                                        <FormControl className="m-0 p-0">
                                            <Input
                                            className="border-0 outline-none
                                            focus-visible:ring-0
                                            focus-visible:ring-transparent"
                                            disabled ={isLoading}
                                            placeholder="How do I calculate the radius of a circle"
                                            {...field}
                                            />
                                        </FormControl>
                                    </FormItem>
```

```tsx
                                )}/>
                                <Button className = "col-span-12 lg:col-span-2 w-full"
                                    disabled={isLoading}>
                                    Generate
                                    </Button>
                            </form>
                        </Form>
                    </div>
                    <div className="space-y-4 mt-4">
                        {/* Display a loading indicator when a request is in progress */}
                        {isLoading && (
                            <div className="p-8 rounded-lg w-full flex items-center
                                justify-center bg-muted">
                                    <Loading/>
                            </div>
                        )}

                        {/* Display a message if no conversation has started yet */}
                        {messages.length === 0 && !isLoading && (
                            <Empty label=" No conversation started"/>
                        )}

                        {/* Render the conversation messages */}
                        <div className="flex flex-col-reverse gap-y-4">
                            {messages.map((message)=>(
                                <div key = {message.content}
                                 className={cn(
                                    "p-8 w-full flex items-start gap-x-8 rounded-lg",
                                    message.role === "user" ? "bg-white border border-black/10" : "bg-muted"
                                )}
                                >
                                    {/* Display user or bot avatar depending on the message role */}
                                    {message.role === "user" ? <UserAvatar/> :
                                    <BotAvatar /> }

                                    <p className="text-sm">

                                        {message.content}
                                    </p>
                                </div>

                            ))}
                        </div>
                    </div>
                </div>
            </div>
        );
    }

    export default ConversationPage;
```

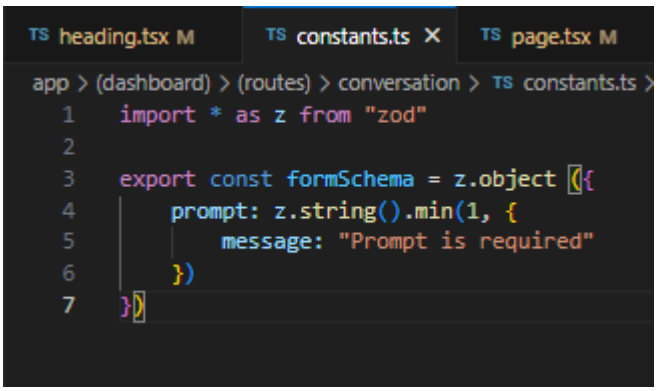
The development of ‘ConversationPage’ began with the implementation of the ‘Heading’ component.



This component serves as a dynamic header for the page, displaying the title "Conversation", a brief description of the page's functionality, and an associated icon. The Heading component is designed to be reusable, enhancing consistency across different parts of the application.

Another significant part of the page is dedicated to a form where users can input their questions or statements. To ensure data handling and validation, the react-hook-form library was used, providing an efficient way to manage form states and validations.

Alongside, a schema for the form was defined using ‘zod’ . The ‘formSchema’ specifies the structure and validation rules for the user input, ensuring that the data submitted is correctly formatted. In this case, it validates that the user's prompt is not empty.



If we break it down even further, this is how it works:

‘z.object(...)’ T is a zod method to define a schema for an object. It's used to create a schema for the form data. Inside z.object, the prompt field is defined as a string with a minimum length of 1. If the prompt field does not meet this criterion (if it's empty), zod will return an error with the message "Prompt is required".

This is how the form schema is used in the creation of the form inside page.tsx:

```
55      // Setup form with react-hook-form and zod for validation
56      const form = useForm<z.infer<typeof formSchema>>({
57          resolver: zodResolver(formSchema),
58          defaultValues: {
59              prompt: ""
60          }
61      })
62
```

- 'useForm' is a hook from 'react-hook-form' that manages the form state and handles form submission.
- <z.infer<typeof formSchema>>: This syntax is used for TypeScript (enhances type safety). It infers the type of the form's data based on the 'formSchema' defined using 'zod'. It ensures that the form data matches the structure and types specified in 'formSchema'.
- resolver: 'zodResolver(formSchema)' : This integrates 'zod' with 'react-hook-form'. It uses 'zod' to validate the form data against 'formSchema' . The 'zodResolver' converts 'zod' validation rules into a format that react-hook-form can use.

Then there is the function handling the form submission after its validation, let's break it down

```
66      // Handling form submission
67      const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
68          try{
69              const userMessage: ChatCompletionRequestMessage = { role: "user", content: values.prompt };
70
71              const newMessages = [...messages, userMessage]
72
73              const response = await axios.post("/api/conversation", {
74               messages: newMessages
75              })
76
77              setMessages((current) => [...current, userMessage, response.data])
78
79              form.reset()
80
81          } catch (error: any){
82              if(error?.response?.status === 403){
83                  proModal.onOpen()
84              } else {
85                  toast.error("Something went wrong")
86              }
87          } finally{
88              router.refresh()
89          }
90      }
```

The 'onSubmit' function is asynchronous, allowing it to handle API requests. When a user submits the form, 'onSubmit' constructs a 'userMessage' object. This object represents the user's input, labeled with the role 'user' and containing the content the user typed. Besides this, the type 'ChatCompletionRequestMessage' is used to ensure that the data sent to the AI service uses the expected format. The 'userMessage 'is then added to an array of existing messages . This array will now represent the entire conversation up to this point, including the latest user input.

The updated array of messages is sent to the server-side route (/api/conversation) via an axios.post request. This route, defined in 'route.ts' , processes the request, interacts with the OpenAI API, and generates a response based on the user's message. Then the response from the OpenAI API (handled in 'route.ts') is returned to the 'onSubmit' function. This response typically contains the AI response to the user's input.

After this, the 'setMessages' function updates the application's state with both the user's message and the OpenAI's response. This ensures that the conversation displayed to the user includes both parts: what the user said and how the bot responded.

Finally, the function ensures that the page's state is refreshed using router.refresh() basically it refreshes the data on the page without needing a full page reload, maintaining the accuracy of the displayed conversation.

The last part of the 'page.tsx' is what resides in the return statement:

```
// Rendering the conversation page
return (
    <div>
        <Heading
        title = "Conversation"
        description = "Our most advanced conversation model."
        icon = { MessageSquare}
        iconColor = "text-violet-500"
        bgColor = "bg-violet-500/10 "
        />
        <div className="px-4 lg:px-8">

            <div>
                <Form {...form}>
                    <form onSubmit={form.handleSubmit(onSubmit)}
                        className="rounded-lg border w-full p-4 px-3 md:px-6 focus-within:shadow-sm grid grid-cols-12 gap-2">

                        <FormField
                            name = "prompt"
                            render={({field}) =>(
                                <FormItem className="col-span-12 lg:col-span-10">
                                    <FormControl className="m-0 p-0">
                                      <Input
                                        className="border-0 outline-none
                                        focus-visible:ring-0
                                        focus-visible:ring-transparent"
                                        disabled ={isLoading}
                                        placeholder="How do I calculate the radius of a circle"
                                        {...field}
                                        />
                                    </FormControl>
                                </FormItem>
                            )}/>
                            <Button className = "col-span-12 lg:col-span-2 w-full"
                              disabled={isLoading}>
                              Generate
                              </Button>
                    </form>
                </Form>
            </div>
            <div className="space-y-4 mt-4">
                {/* Display a loading indicator when a request is in progress */}
                {isLoading && (
                    <div className="p-8 rounded-lg w-full flex items-center
                        justify-center bg-muted">
                            <Loading/>
                    </div>
                )}

                {/* Display a message if no conversation has started yet */}
                {messages.length === 0 && !isLoading && (
                    <Empty label=" No conversation started"/>
                )}

                {/* Render the conversation messages */}
                <div className="flex flex-col-reverse gap-y-4">
                    {messages.map((message)=>(
                        <div key = {message.content}
                        className={cn(
                            "p-8 w-full flex items-start gap-x-8 rounded-lg",
                            message.role === "user" ? "□bg-white border □border-black/10" : "bg-muted"
                        )}
                        >
                            {/* Display user or bot avatar depending on the message role */}
                            {message.role === "user" ? <UserAvatar/> :
                            <BotAvatar /> }

                            <p className="text-sm">

                                {message.content}
                            </p>
                        </div>
```

Starting with the ‘Heading ‘ component which is used to display the title and description of the page at the top. It includes an the icon ‘MessageSquare’ to visually represent the conversation feature.

Form Component Setup:

The form starts with a custom Form component from @/components/ui/form. This component is designed to work seamlessly with react-hook-form.

The {...form} spread syntax is used to pass down the ‘useForm’ hook's methods and properties to the ‘Form’ component. This approach is efficient because it avoids the need to manually pass each prop.

Native HTML Form Element:

Inside the custom ‘Form’ component, a native HTML form element is used. The ‘onSubmit’ prop of this native form is linked to ‘form.handleSubmit(onSubmit)’, which connects it to the react-hook-form's submission handler.

Here’s what this means:

- ‘handleSubmit’ is a method provided by ‘react-hook-form’ , accessed via the form object. It takes care of processing the form's submission.
- When you submit the form ‘handleSubmit’ is invoked.
- ‘handleSubmit’ internally performs validation checks as defined by ‘react-hook-form’ and the validation schema.
- If the validation is successful, ‘handleSubmit’ then calls the ‘onSubmit’ function, passing the form data to it.
- ‘onSubmit’ function is where is defined what happens with the form data, like sending it to a server or updating the application state.

FormField Component for User Input:

The ‘FormField’ component represents an individual form field. It's given a name prop ("prompt"), which links it to the corresponding field in the ‘useForm’ state.

The render prop of FormField is a function that takes a field object and returns JSX. This function provides a way to define the UI for the input field.

FormItem and FormControl Components:

Inside the ‘FormField’, the ‘FormItem’ and ‘FormControl’ components are used. These components are part of the UI library and provide consistent styling and behavior for form elements.

‘FormItem’ is responsible for the overall layout of the form item, while ‘FormControl’ handles the specific control like an input field.

Input Component for Data Entry:

The Input component is where the user types their message. It receives the field object via spread syntax, connecting it to react-hook-form. This connection ensures that the input's value and change events are managed by react-hook-form.

Additional class names and the disabled prop are applied to the Input component for styling and to disable it when the form is submitting (isLoading).

Submit Button:

A Button component is added to submit the form. It's styled to be full-width and is also disabled during form submission.

Overall, the ‘FormField’ component, along with ‘FormItem’ ,’ FormControl’ , and Input, create a structured, controlled form field that is fully integrated with react-hook-form's features.

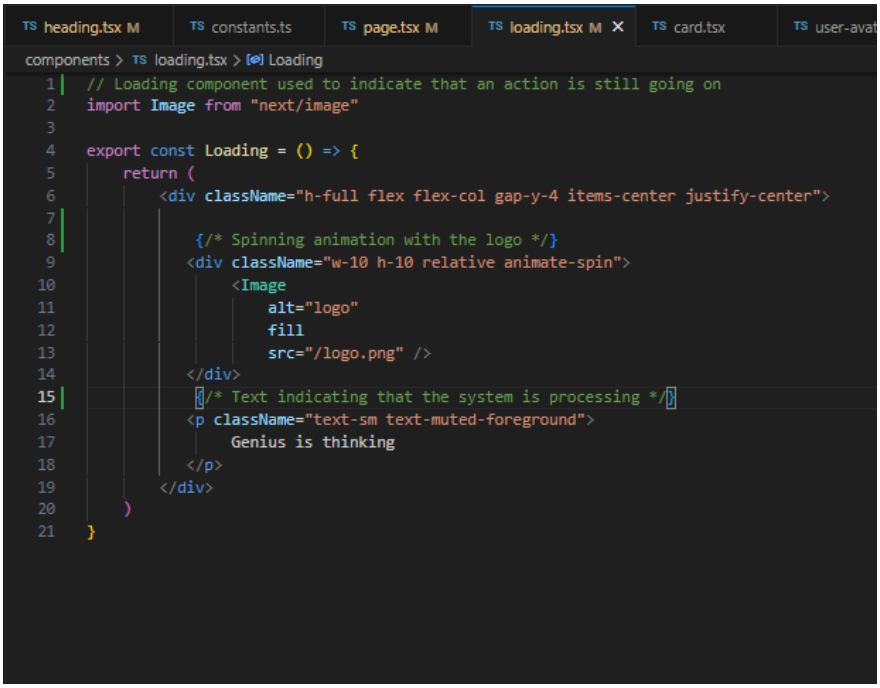
The use of these components, instead of plain HTML elements, provides a more consistent and customizable user interface while leveraging react-hook-form. Here is the official documentation
https://ui.shadcn.com/docs/components/form

The second part of the return statement handles the display and rendering of messages in the conversation interface:

```
            </div>
            <div className="space-y-4 mt-4">
                {/* Display a loading indicator when a request is in progress */}
                {isLoading && (
                    <div className="p-8 rounded-lg w-full flex items-center
                        justify-center bg-muted">
                            <Loading/>
                    </div>
                )}

                {/* Display a message if no conversation has started yet */}
                {messages.length === 0 && !isLoading && (
                    <Empty label=" No conversation started"/>
                )}

                {/* Render the conversation messages */}
                <div className="flex flex-col-reverse gap-y-4">
                    {messages.map((message)=>(
                        <div key = {message.content}
                        className={cn(
                            "p-8 w-full flex items-start gap-x-8 rounded-lg",
                            message.role === "user" ? "bg-white border border-black/10" : "bg-muted"
                        )}
                        >
                            {/* Display user or bot avatar depending on the message role */}
                            {message.role === "user" ? <UserAvatar/> :
                            <BotAvatar /> }

                            <p className="text-sm">

                                {message.content}
                            </p>
                        </div>

                    ))}
                </div>
            </div>
        </div>
    </div>
    );
}

export default ConversationPage;
```

We have the ‘Loading’ Indicator:

- isLoading && (...): This checks if the ‘isLoading’ state is true, which typically indicates that a request is in progress.
- When ‘isLoading’ is true, a loading indicator , <Loading/> is shown. This informs the user that the application is processing something, here is the component:



Showing a Placeholder for Empty Conversations:

- messages.length === 0 && !isLoading && (...): This condition checks if there are no messages and the application is not currently loading
- If true, it displays an <Empty> component with the label "No conversation started". This is a user-friendly way to indicate that the conversation area is empty, encouraging the user to start a conversation. Here it’s how the ‘Empty’ component looks like:

components > TS empty.tsx > ...

```tsx
// Empty component displays a placeholder when there are no items
import Image from "next/image";

// defining a orop for the message to be displayed
interface EmptyProps {
    label:string
}

export const Empty = ({
    label
}: EmptyProps) => {
    return (
        <div className="h-full p-20 flex flex-col items-center
            justify-center">

            {/* Placeholder image */}
            <div className="relative h-72 w-72">
                <Image
                    alt= "Empty"
                    fill
                    src = "/empty.png"
                />
            </div>
            {/* Custom message displayed below the image */}
            <p className="text-muted-foreground text-sm text-center">
                {label}
            </p>
        </div>
    );
}
```

Rendering Conversation Messages:

- messages.map((message) => (...): iterates over each message in the messages array. Each message object represents a single message in the conversation.
- For each message, a div is created to display the message content. The key prop uses 'message.content' for React's list rendering .
- The className uses a conditional (cn(...)) to apply different styles depending on the message's role (user or bot). For instance, user messages and bot messages might have different background colors or borders for distinction.
- Inside each message div, an avatar is shown (<UserAvatar/> or <BotAvatar/>) based on whether the message is from the user or the bot.
- The actual text of the message ({message.content}) is displayed in a paragraph (<p>).
  Here is how the components look like(the avatar component on top of which 'UserAvatar' and 'BotAvatar' are created was installed using the shadcn CLI here https://ui.shadcn.com/docs/components/avatar) :

components > TS user-avatar.tsx > UserAvatar

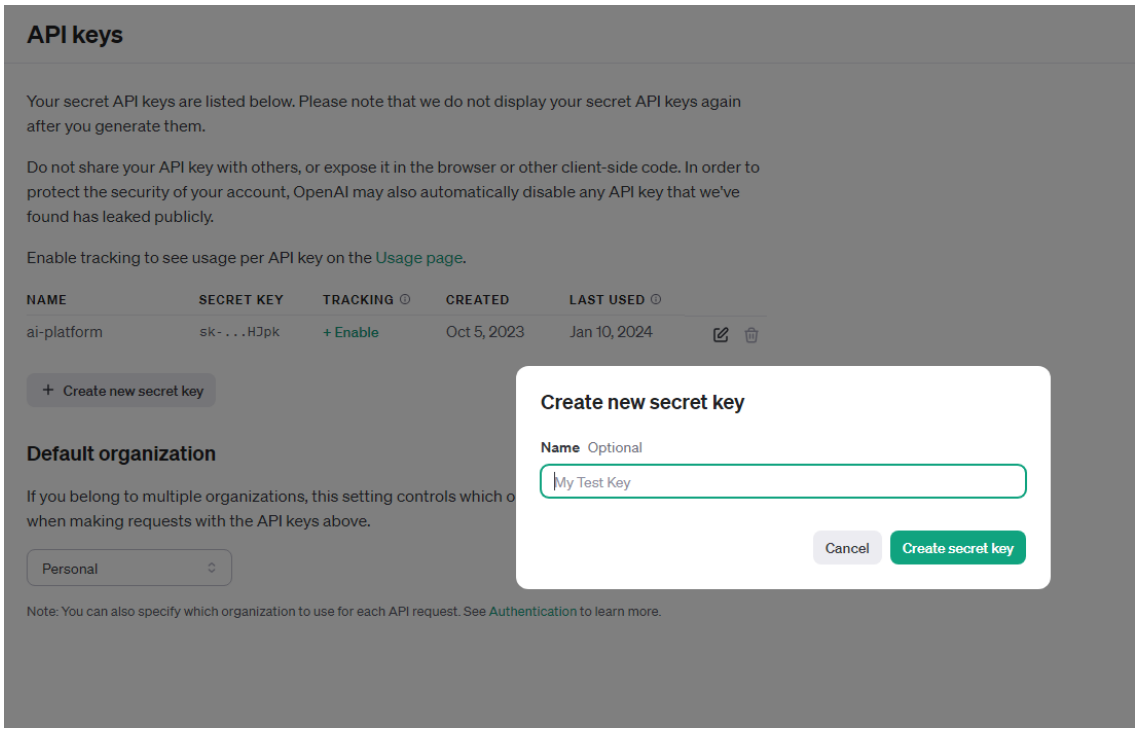
```tsx
// UserAvatar component displays the avatar of the logged user
import { useUser } from "@clerk/nextjs"
import { Avatar, AvatarImage } from "@/components/ui/avatar"
import { AvatarFallback } from "@radix-ui/react-avatar"

export const UserAvatar = () => {
    // Hook to access the user's information
    const { user } = useUser()

    return (
        <Avatar className="h-8 w-8">
            {/* Display user's profile image */}
            <AvatarImage src={user?.profileImageUrl} />

            {/* The case if no image is available */}
            <AvatarFallback>
                {user?.firstName?.charAt(0)}
                {user?.lastName?.charAt(0)}
            </AvatarFallback>
        </Avatar>

    )
}
```

components > TS bot-avatar.tsx > ...

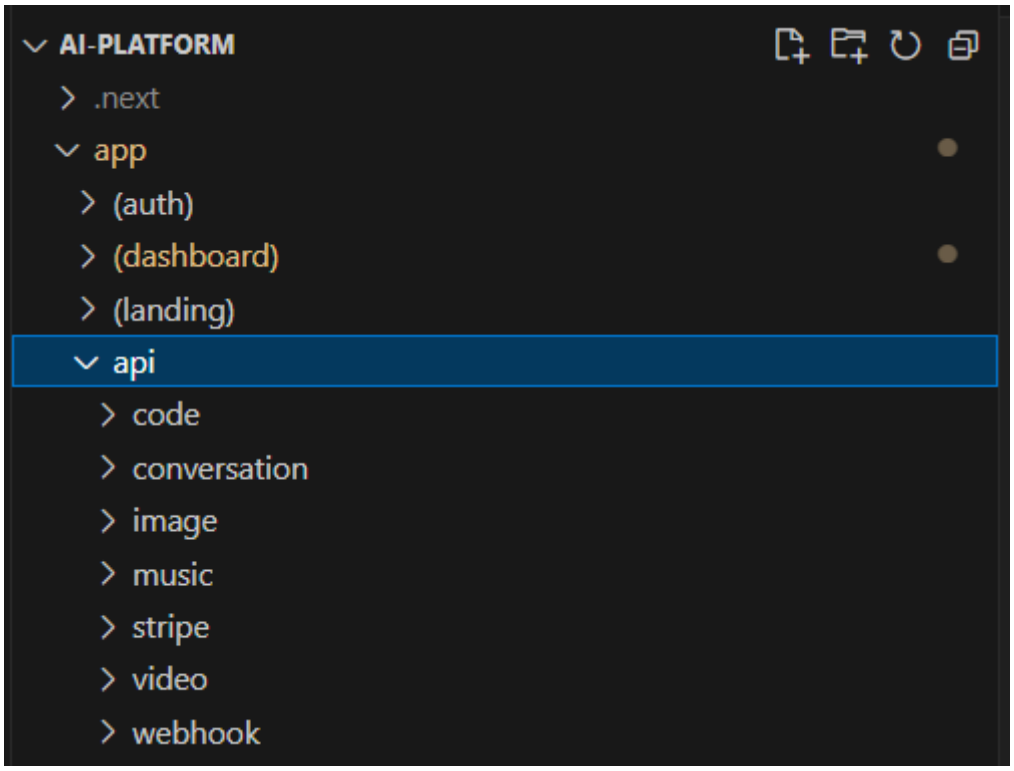
```tsx
// BotAvatar component for displaying a default avatar for the bot
import { Avatar, AvatarImage } from "@radix-ui/react-avatar"

export const BotAvatar = () => {
    return (
        <Avatar className="h-8 w-8">

            {/* Static image used for the bot avatar */}
            <AvatarImage className="p-1" src = "/logo.png" />
        </Avatar>
    )
}
```

Up next, we can take a look at the API route which is used to connect to Open's AI API. You can check out their API by signing up on their page https://platform.openai.com/ . You can still use the free tier if it's your first time registering and you will have 5$ to use on their API for free, but that's only in the first three months.

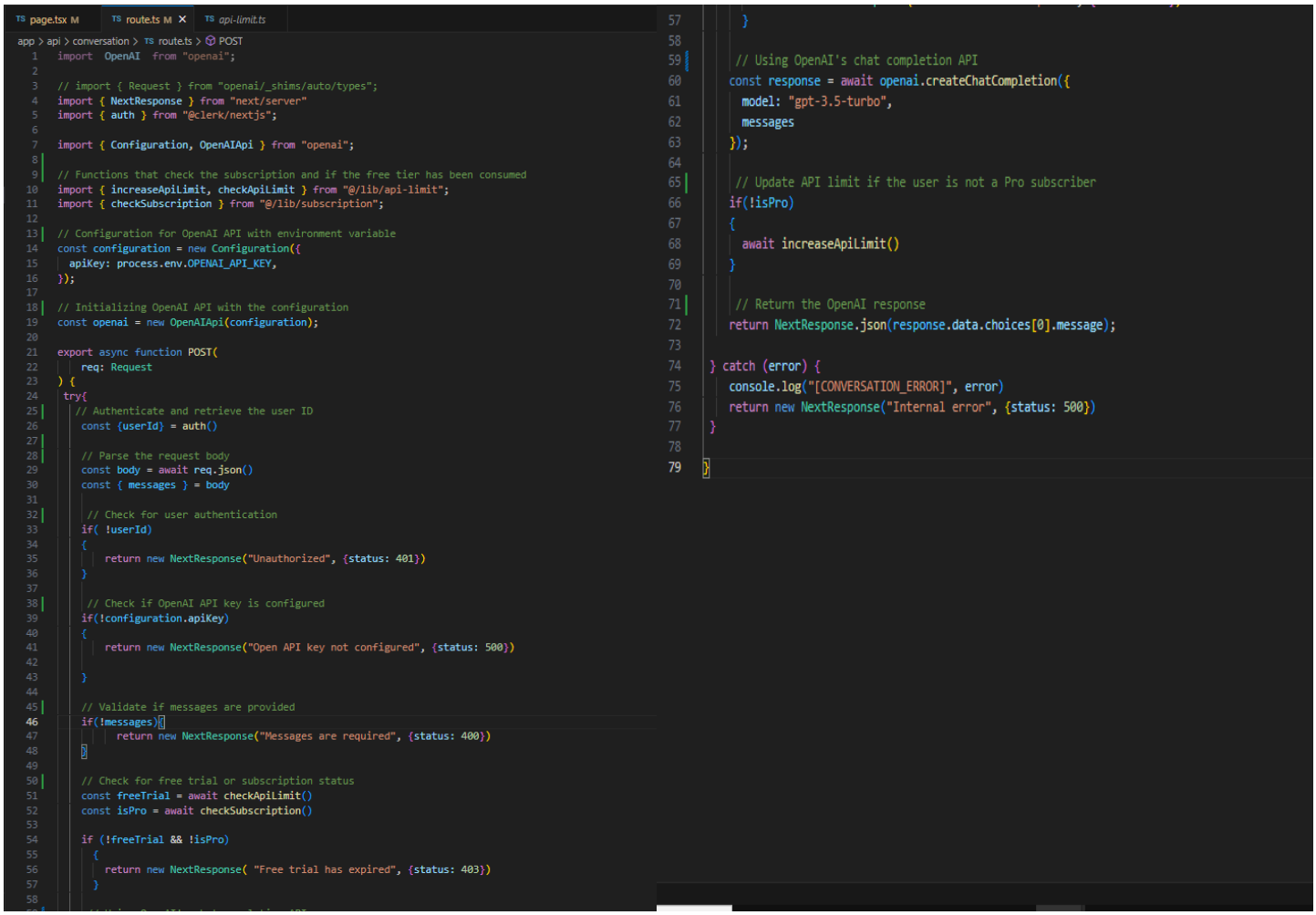
You can go to the API keys section, create an API key and then add it to your environment variables, this will be used to interact with the API.



Inside the app folder now the routes for the api's will be added at the following location



Before taking a look at the conversation folder which contains a file called route.ts, first I had to install the openai package using ***npm i openai.*** Now let's see the inside of 'route.ts':



```ts
import OpenAI from "openai";

// import { Request } from "openai/_shims/auto/types";
import { NextResponse } from "next/server"
import { auth } from "@clerk/nextjs";

import { Configuration, OpenAIApi } from "openai";

// Functions that check the subscription and if the free tier has been consumed
import { increaseApiLimit, checkApiLimit } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

// Configuration for OpenAI API with environment variable
const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});

// Initializing OpenAI API with the configuration
const openai = new OpenAIApi(configuration);

export async function POST(
    req: Request
) {
 try{
    // Authenticate and retrieve the user ID
    const {userId} = auth()

    // Parse the request body
    const body = await req.json()
    const { messages } = body

    // Check for user authentication
    if( !userId)
    {
        return new NextResponse("Unauthorized", {status: 401})
    }

    // Check if OpenAI API key is configured
    if(!configuration.apiKey)
    {
        return new NextResponse("Open API key not configured", {status: 500})

    }

    // Validate if messages are provided
    if(!messages){
        return new NextResponse("Messages are required", {status: 400})
    }

    // Check for free trial or subscription status
    const freeTrial = await checkApiLimit()
    const isPro = await checkSubscription()

    if (!freeTrial && !isPro)
     {
        return new NextResponse( "Free trial has expired", {status: 403})
     }
    }

    // Using OpenAI's chat completion API
    const response = await openai.createChatCompletion({
      model: "gpt-3.5-turbo",
      messages
    });

    // Update API limit if the user is not a Pro subscriber
    if(!isPro)
    {
      await increaseApiLimit()
    }

    // Return the OpenAI response
    return NextResponse.json(response.data.choices[0].message);

  } catch (error) {
    console.log("[CONVERSATION_ERROR]", error)
    return new NextResponse("Internal error", {status: 500})
  }

}
```

The ‘route.ts’ file is an implementation of a server-side route handler for the Next.js application, particularly for handling POST requests associated with the OpenAI Chat API. Let's discuss its functionality.

The script begins by importing necessary modules and functions. OpenAI from the ‘openai’ package is imported to interact with OpenAI's API. ‘NextResponse’ from ‘next/server’ is used for server-side responses in Next.js. ‘auth’ from ‘@clerk/nextjs’ is used for authentication purposes. Additional imports include configuration utilities for the OpenAI API and functions for API limit and subscription checks. Let’s divide everything into sections, starting with:

Configuration for OpenAI API:

- A new configuration object is created for the OpenAI API using the API key stored in environment variables. This configuration is essential for authenticating and making requests to OpenAI's services.

POST Function:

- The POST function is an asynchronous function that handles POST requests. It begins by extracting the user ID using the auth function and reading the request body. If there's no user ID, it returns a 401 Unauthorized response. If the OpenAI API key is not configured, it returns a 500 Internal Server Error.

Request Validation and API Limit Check:

- The function checks if the messages field is present in the request body. If not, it returns a 400 Bad Request. It then checks the user's API usage limit and subscription status. If the user has exceeded the free trial limit and is not a Pro user, it returns a 403 Forbidden response.

OpenAI Chat Completion Request:

- If all checks pass, the script makes a request to OpenAI's ‘createChatCompletion’ API, passing the model and messages. The model specified here is "gpt-3.5-turbo".

Response Handling and API Limit Update:

- After receiving the response from OpenAI, if the user is not a Pro subscriber, the script updates the API usage limit. Finally, it returns the response from the OpenAI API to the client.

Error Handling:

- If any error occurs during the process, the function catches it and logs the error. It then returns a 500 Internal Server Error response.

The 'onSubmit' function inside the conversation/page.tsx is connected with the route.ts

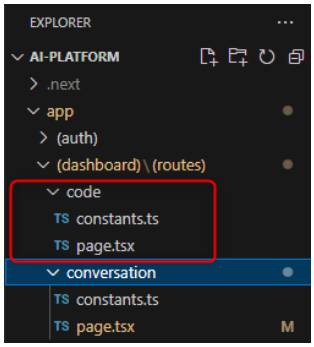
```
66    // Handling form submission
67    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
68        try{
69           const userMessage: ChatCompletionRequestMessage = { role: "user", content: values.prompt };
70
71           const newMessages = [...messages, userMessage]
72
73           const response = await axios.post("/api/conversation", {
74            messages: newMessages
75          })
76
77          setMessages((current) => [...current, userMessage, response.data])
78
79          form.reset()
80
81        } catch (error: any){
82          if(error?.response?.status === 403){
83              proModal.onOpen()
84          } else {
85            toast.error("Something went wrong")
86          }
87        } finally{
88           router.refresh()
89        }
90    }
```

It handles the user's input, sends it to the server-side 'route.ts'(which we talked about now) , which in turn communicates with the OpenAI API.

The server-side logic in 'route.ts' adds a layer of control, as it validates requests and manages API keys and limits. This workflow ensures that the conversation with the OpenAI model is smoothly managed and updated on the client-side.

Code Generation

The next AI tool we're going to take a look at the code generation, which is an evolution of the Conversation component, redesigned to interact with OpenAI's API for generating code snippets. The adaptation process was straightforward because of the similarities in functionality and structure between the two components. To start off, since the code is similar to the one from the conversation folder, we are just going to duplicate that and place it in the (dashboard)/(routes).



After that, the code was modified to fit my needs, and the changes will be discussed below

```tsx
// Used for making HTTP requests to the API
import axios from "axios";

// zod: A schema validation library to validate  data structures
import * as z from "zod"
import { Code, Loader } from "lucide-react";
import { useForm } from "react-hook-form"

// Custom UI components: Heading for page titles, UserAvatar and BotAvatar for user and bot representations in chats,
// Empty for empty state messages, and Loading for indicating loading states.
import Heading from "@/components/heading";
import { Empty } from "@/components/empty";
import { Loading } from "@/components/loading";
import { UserAvatar } from "@/components/user-avatar";
import { BotAvatar } from "@/components/bot-avatar";

// Schema for form validation using zod
import { formSchema } from "./constants";

import { useRouter } from "next/navigation"

// Using zod to create the schema and validation
import { zodResolver } from "@hookform/resolvers/zod"
//Shadcn components and packages;
import { Form, FormControl, FormField, FormItem } from "@/components/ui/form";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";

import { useState } from "react";
import { cn } from "@/lib/utils";

// interface from the OpenAI package used to define the structure of chat messages
import  {ChatCompletionRequestMessage}  from "openai"

// Component used to display the code generated by the AI in an elegant manner
import ReactMarkdown from "react-markdown"

// custom hook for managing the modal related to the pro features of the app.
import { useProModal } from "@/hooks/use-pro-modal";

// used for user feedback like success or error messages.
import toast from "react-hot-toast";

const CodePage = () => {
    // Hook for modal and routing
    const proModal = useProModal()
    const router = useRouter()

    // State for managing messages in the conversation
    const [messages, setMessages] = useState<ChatCompletionRequestMessage[]>([])

    // Setup form with react-hook-form and zod for validation
    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            prompt: ""
        }
    })
```

```tsx
    // Flag for submission state
    const isLoading = form.formState.isSubmitting

    // Handling form submission
    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
        try{


            const userMessage: ChatCompletionRequestMessage = { role: "user", content: values.prompt };

            const newMessages = [...messages, userMessage]

            const response = await axios.post("/api/code", {
              messages: newMessages
            })

            setMessages((current) => [...current, userMessage, response.data])

            form.reset()


        } catch (error: any){
            if(error?.response?.status === 403){
              proModal.onOpen()
            }else {
              toast.error("Something went wrong")
            }
            console.log(error)
        } finally{
            router.refresh()
        }

    }
    // Rendering the code page
    return (
        <div>
            <Heading
            title = "Code Generation"
            description = "Generate code using descriptive text"
            icon = { Code}
            iconColor = "text-green-700"
            bgColor = "bg-green-700/10 "
            />
            <div className="px-4 lg:px-8">

                <div>
                    <Form {...form}>
                        <form onSubmit={form.handleSubmit(onSubmit)}
                            className="rounded-lg border w-full p-4 px-3 md:px-6 focus-within:shadow-sm grid grid-cols-12 gap-2">

                            <FormField
                                name = "prompt"
                                render={({field}) =>(
                                    <FormItem className="col-span-12 lg:col-span-10">
                                        <FormControl className="m-0 p-0">
                                            <Input
                                            className="border-0 outline-none
                                            focus-visible:ring-0
                                            focus-visible:ring-transparent"
```

```tsx
                                            disabled ={isLoading}
                                            placeholder="Simple toggle button using react hooks"
                                            {...field}
                                            />
                                        </FormControl>
                                    </FormItem>
                                )}/>
                                <Button className = "col-span-12 lg:col-span-2 w-full"
                                    disabled={isLoading}>
                                    Generate
                                    </Button>
                        </form>
                    </Form>
                </div>
                <div className="space-y-4 mt-4">
                    {/* Display a loading indicator when a request is in progress */}
                    {isLoading && (
                        <div className="p-8 rounded-lg w-full flex items-center
                            justify-center bg-muted">
                                <Loading/>

                        </div>
                    )}
                     {/* Display a message if no conversation has started yet */}
                    {messages.length === 0 && !isLoading && (
                        <Empty label=" No conversation started"/>
                    )}
                    {/* Render the conversation messages */}
                    <div className="flex flex-col-reverse gap-y-4">
                        {messages.map((message)=>(
                            <div key = {message.content}
                            className={cn(
                                "p-8 w-full flex items-start gap-x-8 rounded-lg",
                                message.role === "user" ? "bg-white border border-black/10" : "bg-muted"
                            )}
                            >
                                {/* Display user or bot avatar depending on the message role */}
                                {message.role === "user" ? <UserAvatar/> :
                                <BotAvatar /> }

                                {/* Display the code written by the bot in an unique manner */}
                            <ReactMarkdown
                                components={{
                                    // Customizing the <pre> tag for code blocks
                                    pre: ({node, ...props}) => (
                                        <div className="overflow-auto w-full my-2 bg-black/10 p-2 rounded-lg ">
                                                <pre {...props} />
                                        </div>
                                    ),
                                    // Customizing the <code> tag for inline code
                                    code: ({node, ...props}) => (
                                        <code className="bg-black/10 rounded-lg p-1" {...props} />
                                    )
                                }}

                                className="text-sm overflow-hidden leading-7"
                            >
                                {message.content || ""}
                            </ReactMarkdown>
                            </div>

                        ))}
                    </div>
                </div>
            </div>
        </div>
    );
}
export default CodePage;
```

The Heading props were changed to match the current description (things like color, description, title, icon) and ‘ReactMarkdown’ was added to enhance the user experience by styling the code generated by the bot.

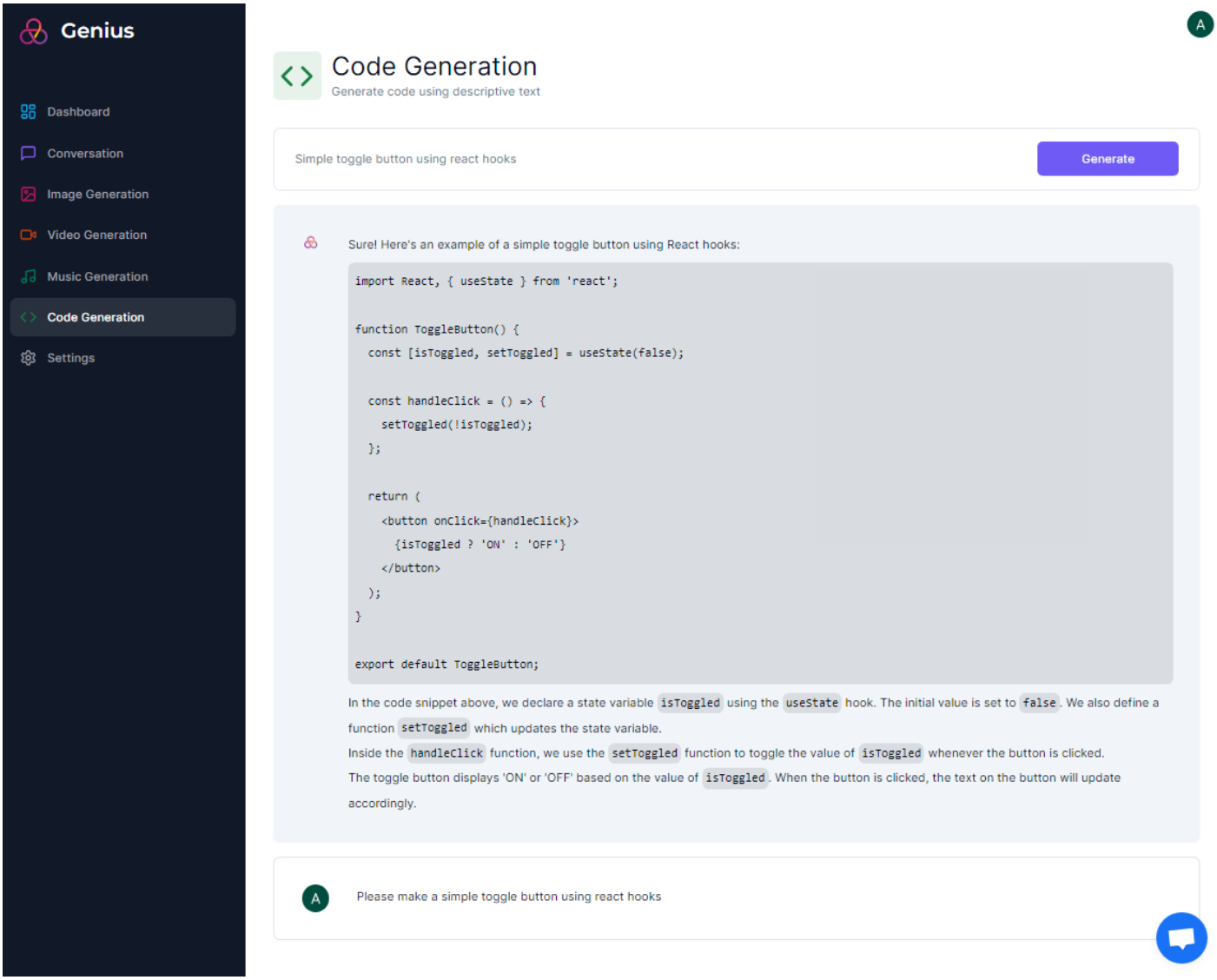
Let’s talk more about the styling of the code. First the ‘ReactMarkdown’ was installed using ***npm i react-markdown*.**

Inside the <ReactMarkdown /> we have the ‘components’ prop which allows you to override the default rendering of markdown elements. The <pre> tag is used for displaying preformatted text in HTML.

In the context of Markdown, it's typically used to display large bits of code. It is wrapped in a div to enhance the appearance of the code.

The <code> tag is used for inline code in HTML, for small snippets of code within a line of text. In this case, the inline code snippets make them different from other text. The customization includes adding a background color and padding.

As for node, it refers to the markdown element being processed. It represents the content in the <pre> or <code> tags and ‘...props’ is used to pass all the received properties of the markdown element to the custom component. This includes standard HTML attributes and any additional properties that ‘ReactMarkdown’ might provide. Below you can observe how these modifications affect the styling of the text.



Besides this, other modifications from the conversation page include the color theme which was altered to green to differentiate the ‘Code Generation’ tool from the ’Conversation’ component and the input placeholder was updated with a different prompt.

Lastly, the API endpoint in the ‘onSubmit’ function was switched from /api/conversation to /api/code to correctly route the code generation.

Now let’s glance over the modifications that were added to the new route (api/code/route.ts)

```tsx
// page.tsx M
// route.ts M
// app > api > code > route.ts > POST
import  OpenAI, { ChatCompletionRequestMessage }  from "openai";

// import { Request } from "openai/_shims/auto/types";
import { NextResponse } from "next/server"
import { auth } from "@clerk/nextjs";

// Configuration for OpenAI API with environment variable
import { Configuration, OpenAIApi } from "openai";

// Functions that check the subscription and if the free tier has been consumed
import { increaseApiLimit, checkApiLimit } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

// Configuration for OpenAI API with environment variable
const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});

// Initializing OpenAI API with the configuration
const openai = new OpenAIApi(configuration);

const instructionMessage: ChatCompletionRequestMessage = {
  role: "system",
  content: " You are a code generator. You must answer only in mardown code snippets. Use code comments for explanations"
}

export async function POST(
    req: Request
) {
 try{
  // Authenticate and retrieve the user ID
    const {userId} = auth()

    // Parse the request body
    const body = await req.json()
    const { messages } = body

    // Check for user authentication
    if( !userId)
    {
        return new NextResponse("Unauthorized", {status: 401})
    }

    // Check if OpenAI API key is configured
    if(!configuration.apiKey)
    {
        return new NextResponse("Open API key not configured", {status: 500})

    }

    // Validate if messages are provided
    if(!messages){
        return new NextResponse("Messages are required", {status: 400})
    }

    // Check for free trial or subscription status
    const freeTrial = await checkApiLimit()
    const isPro = await checkSubscription()

    // Check for free trial or subscription status
    const freeTrial = await checkApiLimit()
    const isPro = await checkSubscription()

    if (!freeTrial && !isPro)
      {
        return new NextResponse( "Free trial has expired", {status: 403})
      }

    // Using OpenAI's chat completion API
    const response = await openai.createChatCompletion({
      model: "gpt-3.5-turbo",
      messages: [instructionMessage, ...messages]
    });

    // Update API limit if the user is not a Pro subscriber
    if(!isPro)
    {
      await increaseApiLimit()
    }
    // Return the OpenAI response
    return NextResponse.json(response.data.choices[0].message);

  } catch (error) {
    console.log("[CODE_ERROR]", error)
    return new NextResponse("Internal error", {status: 500})
  }
}
```

Similar to the front-end component, the server-side route for the 'Code Generation' tool was created by copying the 'Conversation' route. A significant modification was the addition of an instruction message. This message, predefined as 'instructionMessage' , serves as an initial message to the OpenAI model, specifically telling it to behave as a code generator. Moreover, it is instructed to only generate markdown code snippets and use comments for explanations.

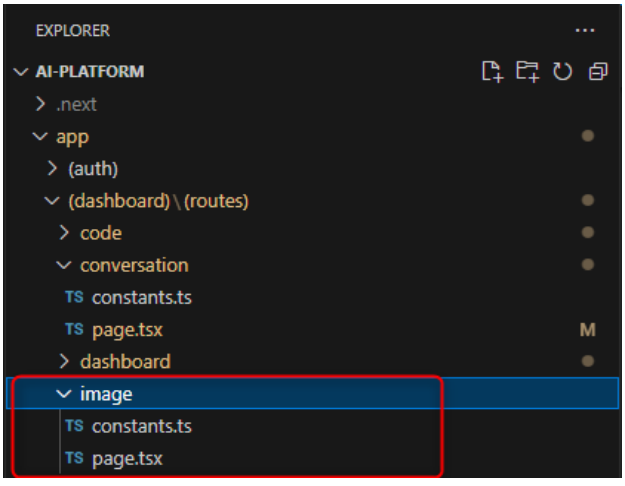
Also, this 'instructionMessage' is placed as the first message in the sequence sent to the OpenAI API. This ensures that every code generation request starts with the instruction message

The error logging was modified specifically for this route as well. By labeling errors as "code errors," it becomes easier to identify and debug. The rest remains the same as for the conversation route.

Following the Code Generation, we have the Image Generation component

Image Generation

The same procedure as for the code generation is applied here as well, the code from the conversation is copied and renamed.



```tsx
// Used for making HTTP requests to the API
import axios from "axios";

// zod: A schema validation library to validate  data structures
import * as z from "zod"
import { Download, ImageIcon } from "lucide-react";
import { useForm } from "react-hook-form"

// Custom UI components: Heading for page titles, UserAvatar and BotAvatar for user and bot representations in chats,
// Empty for empty state messages, and Loading for indicating loading states.
import Heading from "@/components/heading";
import { Empty } from "@/components/empty";
import { Loading } from "@/components/loading";

// Schema for form validation using zod as well as the array of objects used in the user interface
import { amountOptions, formSchema, resolutionOptions } from "./constants";
import { useRouter } from "next/navigation"

// Using zod to create the schema and validation
import { zodResolver } from "@hookform/resolvers/zod";

//Shadcn components and packages:
import { Form, FormControl, FormField, FormItem } from "@/components/ui/form";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";
import { Card, CardFooter } from "@/components/ui/card";
import { Select, SelectItem, SelectTrigger, SelectValue, SelectContent } from "@/components/ui/select";

import { useState } from "react";
import Image from "next/image";

// custom hook for managing the modal related to the pro features of the app
import { useProModal } from "@/hooks/use-pro-modal";

// used for user feedback like success or error messages
import toast from "react-hot-toast";

const ImagePage = () => {

    // Hook for modal and routing
    const proModal = useProModal()
    const router = useRouter()

    // State for managing the images in the conversation
    const [images, setImages] =useState<string[]>([])

    // Setup form with react-hook-form and zod for validation
    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            prompt: "",
            amount: "1",
            resolution: "512x512"
        }
    })

    // Flag for submission state
    const isLoading = form.formState.isSubmitting
```

```tsx
    // Handling form submission
    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
        try{

            // Clearing existing images before making a new request
            setImages([])

            // Sending a POST request to the server with form values
            const response = await axios.post("/api/image", values)

            // Extracting image URLs from the response and updating state
            const urls = response.data.map((image: {url: string}) => image.url)

            setImages(urls)

             // Resetting the form to its default values after submission
            form.reset()

            // Handling specific error scenarios
        } catch (error: any){
            if(error?.response?.status === 403){
              proModal.onOpen()
            }else {
              toast.error("Something went wrong")
            }
             console.log(error)
        }
        finally{
            router.refresh()
        }

    }

    // Rendering the image page
    return (
        <div>
            <Heading
             title = "Image Generation"
             description = "Turn your prompt into an image."
             icon = {ImageIcon}
             iconColor = "text-pink-700"
             bgColor = "bg-pink-700/10 "
             />
             <div className="px-4 lg:px-8">

                <div>
                    <Form {...form}>
                        <form onSubmit={form.handleSubmit(onSubmit)}
                            className="rounded-lg border w-full p-4 px-3 md:px-6 focus-within:shadow-sm grid grid-cols-12 gap-2">
                            <FormField
                                name = "prompt"
                                render={({field}) =>(
                                    <FormItem className="col-span-12 lg:col-span-6">
                                        <FormControl className="m-0 p-0">
                                            <Input
                                              className="border-0 outline-none
                                              focus-visible:ring-0
                                              focus-visible:ring-transparent"
                                              disabled ={isLoading}
```

```tsx
                                              disabled ={isLoading}
                                              placeholder="A picture of a horse in Swiss alps"
                                              {...field}
                                              />
                                        </FormControl>
                                    </FormItem>
                                )}/>

                                {/* SELECT AMOUNT FIELD */}
                                <FormField
                                    name = "amount"
                                    control = {form.control}
                                    render = {({field}) => (
                                        <FormItem className=" col-span-12 lg:col-span-2">
                                            <Select
                                                disabled={isLoading}
                                                onValueChange={field.onChange}
                                                value = {field.value}
                                                defaultValue={field.value} >
                                                <FormControl>
                                                    <SelectTrigger>
                                                        <SelectValue
                                                            defaultValue={field.value}
                                                            />
                                                    </SelectTrigger>
                                                </FormControl>
                                                <SelectContent>
                                                    {amountOptions.map((option)=>(
                                                        <SelectItem
                                                            key={option.value}
                                                            value={option.value}>

                                                                {option.label}
                                                        </SelectItem>
                                                    ))}
                                                </SelectContent>

                                            </Select>
                                        </FormItem>
                                    )}
                                />

                                {/* SELECT RESOLUTION FIELD */}
                                <FormField
                                    name = "resolution"
                                    control = {form.control}
                                    render = {({field}) => (
                                        <FormItem className=" col-span-12 lg:col-span-2">
                                            <Select
                                                disabled={isLoading}
                                                onValueChange={field.onChange}
                                                value = {field.value}
                                                defaultValue={field.value} >
                                                <FormControl>
                                                    <SelectTrigger>
                                                        <SelectValue
                                                            defaultValue={field.value}
                                                            />
                                                    </SelectTrigger>
```

```tsx
                                                </SelectTrigger>
                                            </FormControl>
                                            <SelectContent>
                                                {resolutionOptions.map((option)=>(
                                                    <SelectItem
                                                        key={option.value}
                                                        value={option.value}>

                                                            {option.value}
                                                    </SelectItem>
                                                ))}
                                            </SelectContent>

                                        </Select>
                                    </FormItem>
                                )}
                            />
                            <Button className = "col-span-12 lg:col-span-2 w-full"
                              disabled={isLoading}>
                              Generate
                              </Button>
                        </form>
                    </Form>
                </div>
                <div className="space-y-4 mt-4">
                    {/* Display a loading indicator when a request is in progress */}
                    {isLoading && (
                        <div className="p-20">
                                <Loading/>

                        </div>
                    )}
                    {/* Display a message if no conversation has started yet */}
                    {images.length === 0 && !isLoading && (
                        <Empty label=" No images generated"/>
                    )}
                    {/* Render the  images */}
                    <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4 gap-4 mt-8">
                        {images.map((src) =>(
                            <Card
                              key={src}
                              className="rounded-lg overflow-hidden">
                                <div className="relative aspect-square">
                                    <Image
                                        alt="Image"
                                        fill
                                        src={src} />
                                </div>
                                <CardFooter className="p-2">
                                    <Button
                                      onClick = {() =>
                                      window.open(src)
                                    }
                                      variant = "secondary"
                                      className="w-full">
                                        <Download className="h-4 w-4 mr-2 "/>
                                            Download
                                    </Button>
                                </CardFooter>

                            </Card>
                        ))}
                    </div>

                </div>
            </div>
        </div>
    );
}

export default ImagePage;
```

Once again, the Heading component's title, description, and icon were updated to reflect the image generation context. For instance, the title was changed to "Image Generation" and the icon as well. The default values in the form were also adjusted to include specific fields relevant to image generation, like amount and resolution.

```
// Setup form with react-hook-form and zod for validation
const form = useForm<z.infer<typeof formSchema>>({
    resolver: zodResolver(formSchema),
    defaultValues: {
        prompt: "",
        amount: "1",
        resolution: "512x512"
    }
})
```

Unlike conversation and code generation, for image generation, the form's schema was modified in the constants file to fit the requirements. Two new arrays: ‘amountOptions’ and ‘resolutionOptions’ were made.

These arrays provide predefined options for users to select the number of images and the resolution for image generation. Below is an image with the contents of the constants file located here ‘(dashboard)/(routes)/image/constants.ts’

Further changes to the code generations page are as follows:

Form Fields for Image Generation

In the form, new fields were introduced to select the number of images and their resolution. These fields use the Select component from Shadcn's UI library, providing a user-friendly dropdown selection interface. Let's observe the component which is in charge of the amount field first.

```
{/* SELECT AMOUNT FIELD */}
<FormField
   name = "amount"
   control = {form.control}
   render = {({field}) => (
      <FormItem className=" col-span-12 lg:col-span-2">
         <Select
            disabled={isLoading}
            onValueChange={field.onChange}
            value = {field.value}
            defaultValue={field.value} >
            <FormControl>
               <SelectTrigger>
                  <SelectValue
                     defaultValue={field.value}
                  />
               </SelectTrigger>
            </FormControl>
            <SelectContent>
               {amountOptions.map((option)=>(
                  <SelectItem
                     key={option.value}
                     value={option.value}>

                        {option.label}
                  </SelectItem>
               ))}
            </SelectContent>

         </Select>
      </FormItem>
   )}
/>
```

The' FormField' for the amount field is used to create a dropdown selector. It uses 'form.control' which is an object containing methods for registering an input in the react-hook-form system to manage the field state. The 'control' prop provides 'FormField' with the necessary context to control the select dropdown below.

Next, the render prop is a function that returns JSX. It receives an object, deconstructed to { field }, which contains methods and properties for the input field, like 'onChange' ,' onBlur', 'value'.
This makes it easy to connect the input components like Select to the form's state and handlers provided by react-hook-form.

The Select component is utilized to provide a dropdown list, which is dynamically populated using 'amountOptions'. Users can select the number of images they wish to generate, with the selected value managed by react-hook-form.

Similarly, the same thing happens with the resolution field, except that it uses 'resolutionOptions' to provide different resolutions. More about the 'Select' component can be found here:
https://ui.shadcn.com/docs/components/select

```
{/* SELECT RESOLUTION FIELD */}
<FormField
    name = "resolution"
    control = {form.control}
    render = {({field}) => (
        <FormItem className=" col-span-12 lg:col-span-2">
            <Select
                disabled={isLoading}
                onValueChange={field.onChange}
                value = {field.value}
                defaultValue={field.value} >
                <FormControl>
                    <SelectTrigger>
                        <SelectValue
                            defaultValue={field.value}
                        />
                    </SelectTrigger>
                </FormControl>
                <SelectContent>
                    {resolutionOptions.map((option)=>(
                        <SelectItem
                            key={option.value}
                            value={option.value}>

                              {option.value}
                        </SelectItem>
                    ))}
                </SelectContent>

            </Select>
        </FormItem>
    )}
/>
```

Upcoming, this is how the images are rendered:

```
{/* Render the  images */}
<div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4 gap-4 mt-8">
   {images.map((src) =>(
    <Card
      key={src}
      className="rounded-lg overflow-hidden">
        <div className="relative aspect-square">
            <Image
                alt="Image"
                fill
                src={src} />
        </div>
        <CardFooter className="p-2">
            <Button
              onClick = {() =>
              window.open(src)
            }
              variant = "secondary"
              className="w-full">
                <Download className="h-4 w-4 mr-2 "/>
                    Download
            </Button>
        </CardFooter>

    </Card>
   ))}
</div>

</div>
```

It dynamically renders a grid of images generated based on user inputs. Each Card component displays an individual image using the Next.js Image component. The ‘CardFooter’ contains a button that allows users to

download the respective image while the images array holds URLs of generated images, and each image is rendered as a separate card in the grid.

API Route Modification

The API route for handling image generation requests (/api/image) was made by duplicating and modifying the /api/conversation route. The new route was designed to handle requests specific to generating images, including processing the amount and resolution parameters.

Submission Logic

The 'onSubmit' function was adapted to handle image generation requests. After a successful API call, the returned image URLs are stored in the images state variable, which is then used to render the images on the page.

```
// Handling form submission
const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
    try{

      // Clearing existing images before making a new request
      setImages([])

      // Sending a POST request to the server with form values
      const response = await axios.post("/api/image", values)

      // Extracting image URLs from the response and updating state
      const urls = response.data.map((image: {url: string}) => image.url)

      setImages(urls)

       // Resetting the form to its default values after submission
      form.reset()

      // Handling specific error scenarios
    } catch (error: any){
          if(error?.response?.status === 403){
            proModal.onOpen()
       }else {
         toast.error("Something went wrong")
       }
        console.log(error)
    }
    finally{
        router.refresh()
    }

}
```

The 'route.ts' for the image generation had some modifications as well:

```ts
app > api > image > route.ts > POST
import  OpenAI  from "openai";

// import { Request } from "openai/_shims/auto/types";
import { NextResponse } from "next/server"
import { auth } from "@clerk/nextjs";

// Configuration for OpenAI API with environment variable
import { Configuration, OpenAIApi } from "openai";

// Functions that check the subscription and if the free tier has been consumed
import { increaseApiLimit, checkApiLimit } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

// Configuration for OpenAI API with environment variable
const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});

// Initializing OpenAI API with the configuration
const openai = new OpenAIApi(configuration);

export async function POST(
    req: Request
) {
 try{
    // Authenticate and retrieve the user ID
    const {userId} = auth()
    const body = await req.json()

    // Destructuring prompt, amount, and resolution from the request body
    const { prompt, amount = 1, resolution = "512x512" } = body

    // Check for user authentication
    if( !userId)
    {
        return new NextResponse("Unauthorized", {status: 401})
    }

    // Check if OpenAI API key is configured
    if(!configuration.apiKey)
    {
        return new NextResponse("Open API key not configured", {status: 500})

    }

    // Validate if the prompt, amount and resolution are provided
    if(!prompt){
        return new NextResponse("Prompt is required", {status: 400})
    }

    if(!amount){
        return new NextResponse("Amount is required", {status: 400})
    }
    if(!resolution){
        return new NextResponse("Resolution is required", {status: 400})
    }
```

```ts
page.tsx M   route.ts M
app > api > image > route.ts > POST
    }

    // Check for free trial or subscription status
    const freeTrial = await checkApiLimit()
    const isPro = await checkSubscription()

    if (!freeTrial && !isPro)
    {
        return new NextResponse( "Free trial has expired", {status: 403})
    }


    // Using OpenAI's image generation API
    const response = await openai.createImage({
     prompt,
     n: parseInt(amount, 10),
     size: resolution
    });

    // Update API limit if the user is not a Pro subscriber
    if(!isPro)
    {
      await increaseApiLimit()

    }

    // Return the OpenAI response
    return NextResponse.json(response.data.data);

 } catch (error) {
    console.log("[IMAGE_ERROR]", error)
    return new NextResponse("Internal error", {status: 500})
 }
}
```

Different Request Body Parameters:

The image generation route looks for prompt, amount, and resolution in the request body. These parameters are distinct from those in the conversation route and are tailored to the image generation process.

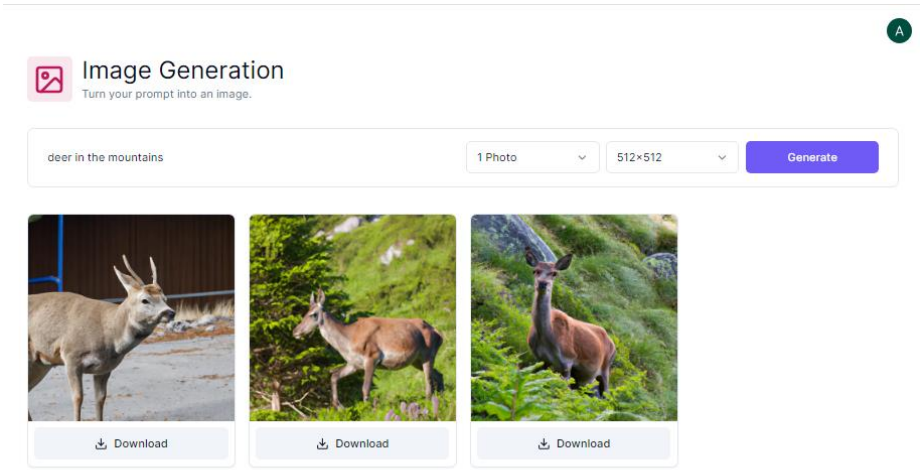
Validation of Additional Parameters:

Unlike the conversation route, this route includes validation for amount and resolution parameters. These validations ensure that the required data for image generation is correctly formatted.

Use of OpenAI's Image Generation API:

The main difference is the utilization of ‘openai.createImage()’ instead of ‘openai.createChatCompletion()’ . This function is specific to generating images and requires parameters like n (number of images) and size (resolution). Also, the response from the OpenAI API is handled differently. It uses return ‘NextResponse.json(response.data.data)’. This is because the structure of the response from the ‘openai.createImage()’ method is different from that of the ‘openai.createChatCompletion()’

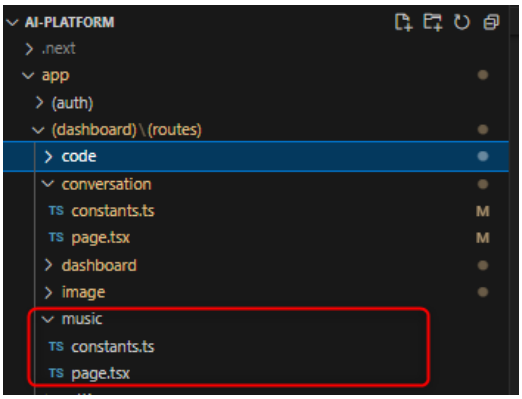
This is how a response from the image generation tool would look like:



The next tool is going to be the music generation.

Music Generation

Once again, the same procedure as before applies here as well, copy the contents of the ‘conversation’ into a new folder called ‘music’ and from there the customization begins.



Nothing changed in the ‘constants.ts’ file and below you can see the ‘page.tsx’ following to discuss the modifications.

```tsx
// Used for making HTTP requests to the API
import axios from "axios";
// zod: A schema validation library to validate  data structures
import * as z from "zod"

import {  Music } from "lucide-react";
import { useForm } from "react-hook-form"

// Custom UI components: Heading for page titles, UserAvatar and BotAvatar for user and bot representations in chats,
// Empty for empty state messages, and Loading for indicating loading states.
import Heading from "@/components/heading";
import { Empty } from "@/components/empty";
import { Loading } from "@/components/loading";

// Schema for form validation using zod
import { formSchema } from "./constants";
import { useRouter } from "next/navigation"

// Using zod to create the schema and validation
import { zodResolver } from "@hookform/resolvers/zod";
//Shadcn components and packages;
import { Form, FormControl, FormField, FormItem } from "@/components/ui/form";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";
import { useState } from "react";
// custom hook for managing the modal related to the pro features of the app
import { useProModal } from "@/hooks/use-pro-modal";
// used for user feedback like success or error messages
import toast from "react-hot-toast";



const MusicPage = () => {

    // Hook for modal and routing
    const proModal = useProModal()
    const router = useRouter()

    // State for managing the audio
    const [music, setMusic] = useState<string>()

    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            prompt: ""
        }
    })

    // Flag for submission state
    const isLoading = form.formState.isSubmitting

    // Handling form submission
    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
        try{
          // Clearing existing audio before making a new request
          setMusic(undefined)

           // Sending a POST request to the server with form values
          const response = await axios.post("/api/music", values)

          setMusic(response.data.audio)

          // Resetting the form to its default values after submission
          form.reset()

        // Handling specific error scenarios
        } catch (error: any){
            if(error?.response?.status === 403){
              proModal.onOpen()
            }else {
```

```tsx
                toast.error("Something went wrong")
            }

        } finally{
            router.refresh()
        }

    }

    // Rendering the music page
    return (
        <div>
            <Heading
            title = "Music Generation"
            description = "Turn your prompt into music"
            icon = { Music}
            iconColor = "text-emerald-500"
            bgColor = "bg-emerald-500/10 "
             />
            <div className="px-4 lg:px-8">

                <div>
                    <Form {...form}>
                        <form onSubmit={form.handleSubmit(onSubmit)}
                            className="rounded-lg border w-full p-4 px-3 md:px-6 focus-within:shadow-sm grid grid-cols-12 gap-2">

                            <FormField
                                name = "prompt"
                                render={({field}) =>(
                                    <FormItem className="col-span-12 lg:col-span-10">
                                        <FormControl className="m-0 p-0">
                                            <Input
                                              className="border-0 outline-none
                                              focus-visible:ring-0
                                              focus-visible:ring-transparent"
                                              disabled ={isLoading}
                                              placeholder="Piano solo"
                                              {...field}
                                              />
                                        </FormControl>
                                    </FormItem>
                                )}/>
                                <Button className = "col-span-12 lg:col-span-2 w-full"
                                  disabled={isLoading}>
                                  Generate
                                  </Button>
                        </form>
                    </Form>
                </div>
                <div className="space-y-4 mt-4">
                        {/* Display a loading indicator when a request is in progress */}
                    {isLoading && (
                        <div className="p-8 rounded-lg w-full flex items-center
                            justify-center bg-muted">
                                <Loading/>

                        </div>
                    )}
                      {/* Display a message if no conversation has started yet */}
                    {!music && !isLoading && (
                        <Empty label=" No music generated"/>
                    )}
                      {/* Render the audio file */}
                    {music && !isLoading && (
                    <audio controls className="w-full mt-8">
                        <source src = {music} />
                    </audio>
                    )}
                </div>
            </div>
        </div>
    );
}

export default MusicPage;
```

The Heading component was updated to reflect the new functionality of the page. The title was changed to "Music Generation" the description to "Turn your prompt into music", and the icon as well to represent the focus on music.

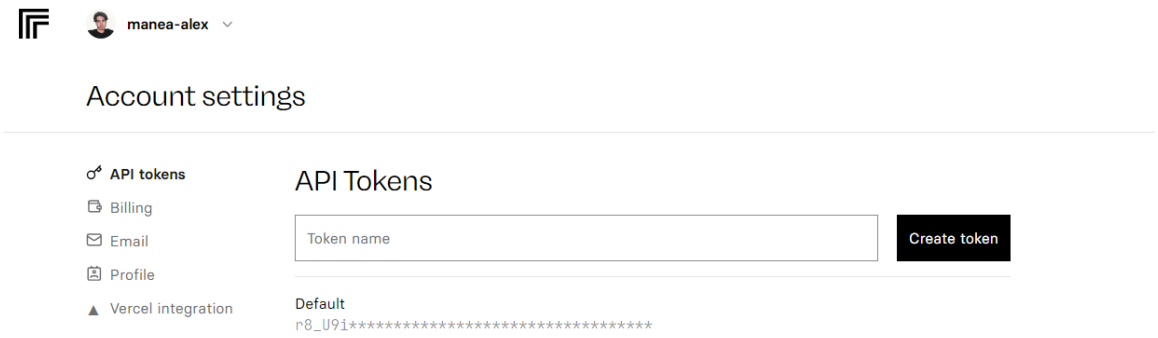
Unlike the conversation tool, which handled an array of messages, the music tool now+ manages a single piece of state, music, which is a string. This state holds the URL of the generated music.

The form utilizes the same structure as before with a 'FormField' for the prompt. However, the placeholder text within the input field was adapted to suggest a musical context.

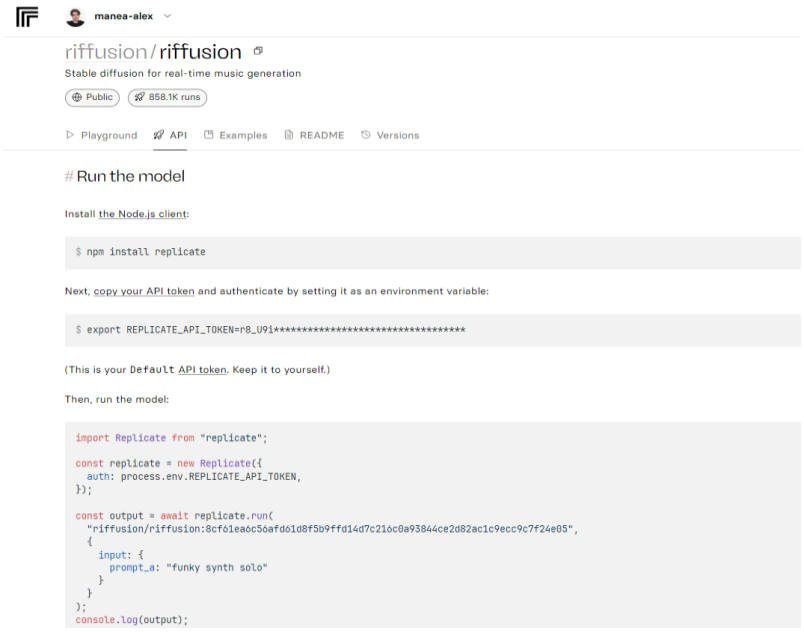
The 'onSubmit' function was modified to handle the music generation request. Upon submission, it makes an HTTP request to the new backend route (/api/music) which will be discussed shortly, with the prompt provided by the user. After receiving the response, the music state is updated with the audio URL returned from the server.

The music generated by the user's prompt is rendered using an HTML audio element. This element is conditionally displayed only if the music state contains a URL and an Empty component is displayed if no music has been generated yet.

Up next, we are going to explore the new api route used for the music page. Unlike the other tools, for the audio generation and video generation we are going to use Replicate AI. You can explore more here https://replicate.com/ Replicate has a free tier so you can test it out. Creating a token is really easy, you just navigate to your profile after logging in and click on 'API Tokens'. From here you can just generate a key and then add it to your .env



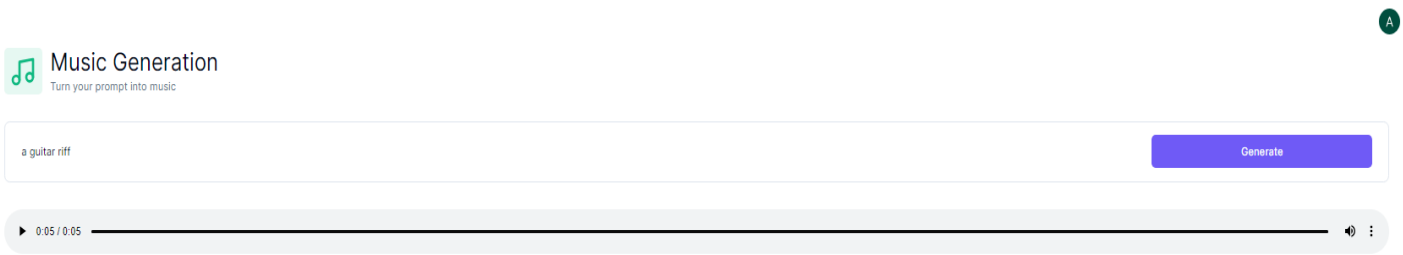
After generating the key, you have to install replicate by running the following command in your terminal: ***npm I replicate***
For the audio generation, the model used will be: riffusion. On their webpage you have information you need in order to use the API (https://replicate.com/riffusion/riffusion/api?tab=nodejs)



After making the necessary adjustments the music route will look like this:

```ts
import { NextResponse } from "next/server"
import { auth } from "@clerk/nextjs";

import Replicate from "replicate"

// Functions that check the subscription and if the free tier has been consumed
import { increaseApiLimit, checkApiLimit } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

// Configuration for Replicate AI with environment variable
const replicate = new Replicate({
  auth: process.env.REPLICATE_API_TOKEN!
})

export async function POST(
    req: Request
) {
 try{

    // Authenticate and retrieve the user ID
    const {userId} = auth()

    // Parse the request body
    const body = await req.json()
    const { prompt } = body

    // Check for user authentication
    if( !userId)
    {
        return new NextResponse("Unauthorized", {status: 401})
    }


    // Validate if the prompt is provided
    if(!prompt){
        return new NextResponse("Prompt is required", {status: 400})
    }

    // Check for free trial or subscription status
    const freeTrial = await checkApiLimit()
    const isPro = await checkSubscription()

    if (!freeTrial && !isPro)
      {
        return new NextResponse( "Free trial has expired", {status: 403})
      }


  // Using the Replicate AI audio generation API to get a response
   const response = await replicate.run(
    "riffusion/riffusion:8cf61ea6c56afd61d8f5b9ffd14d7c216c0a93844ce2d82ac1c9ecc9c7f24e05",
    {
      input: {
        prompt_a: prompt
      }
    }
  );

  // Update API limit if the user is not a Pro subscriber
  if(!isPro)
  {
    await increaseApiLimit()

  }
    // Return the Replicate AI response
    return NextResponse.json(response);

 } catch (error) {
    console.log("[MUSIC_ERROR]", error)
    return new NextResponse("Internal error", {status: 500})
 }
```

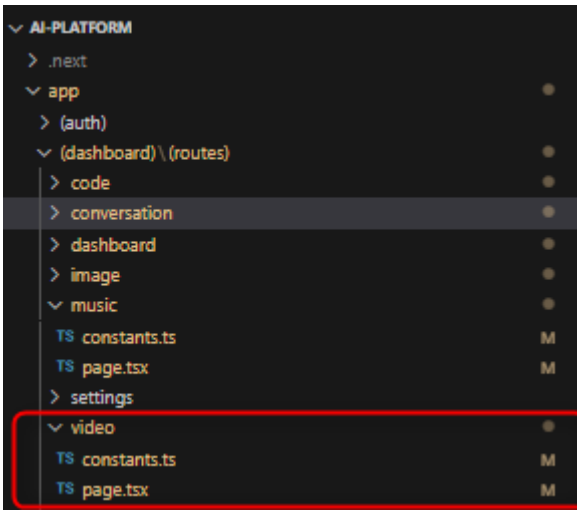
Here is a how a response from the audio model will look like displayed on the ‘page.tsx.



The last AI model on the page will be video generation.

Video Generation

This time we’ll copy the music folder(because it’s more similar) , instead of the conversation and then make the changes needed. Let’s take a look at the ‘page.tsx’ (‘constants.ts’) remained the same.




```tsx
// Used for making HTTP requests to external APIs
import axios from "axios";
// zod: A schema validation library to validate  data structures
import * as z from "zod"
import { VideoIcon } from "lucide-react";
import { useForm } from "react-hook-form"

// Custom UI components: Heading for page titles, UserAvatar and BotAvatar for user and bot representations in chats,
// Empty for empty state messages, and Loading for indicating loading states.
import Heading from "@/components/heading";
import { Empty } from "@/components/empty";
import { Loading } from "@/components/loading";

// Schema for form validation using zod
import { formSchema } from "./constants";
import { useRouter } from "next/navigation"

// Using zod to create our schema and validation
import { zodResolver } from "@hookform/resolvers/zod";

//Shadcn components and packages
import { Form, FormControl, FormField, FormItem } from "@/components/ui/form";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";
import { useState } from "react";

// custom hook for managing the modal related to the pro features of the app.
import { useProModal } from "@/hooks/use-pro-modal";

💡 used for user feedback like success or error messages.
import toast from "react-hot-toast";

const VideoPage = () => {
    // Hook for modal and routing
    const proModal = useProModal()
    const router = useRouter()

    // State for managing the videos in the conversation
    const [video, setVideo] = useState<string>()

    // Setup form with react-hook-form and zod for validation
    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            prompt: ""
        }
    })

     // Flag for submission state
    const isLoading = form.formState.isSubmitting

    // Handling form submission
    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
        try{
          setVideo(undefined)

          const response = await axios.post("/api/video", values)

          setVideo(response.data[0])

          form.reset()


        } catch (error: any){
            if(error?.response?.status === 403){
                proModal.onOpen()
            }else {
              toast.error("Something went wrong")
            }
        } finally{
            router.refresh()
        }
```

```tsx
    }

    // Rendering the video page
    return (
        <div>
            <Heading
            title = "Video Generation"
            description = "Turn your prompt into video"
            icon = { VideoIcon}
            iconColor = "text-orange-700"
            bgColor = "bg-orange-700/10 "
             />
            <div className="px-4 lg:px-8">

                <div>
                    <Form {...form}>
                        <form onSubmit={form.handleSubmit(onSubmit)}
                            className="rounded-lg border w-full p-4 px-3 md:px-6 focus-within:shadow-sm grid grid-cols-12 gap-2">

                            <FormField
                                name = "prompt"
                                render={({field}) =>(
                                    <FormItem className="col-span-12 lg:col-span-10">
                                        <FormControl className="m-0 p-0">
                                            <Input
                                              className="border-0 outline-none
                                              focus-visible:ring-0
                                              focus-visible:ring-transparent"
                                              disabled ={isLoading}
                                              placeholder="Clown fish swimming around a coral reef"
                                              {...field}
                                              />
                                        </FormControl>
                                    </FormItem>
                                )}/>
                                <Button className = "col-span-12 lg:col-span-2 w-full"
                                  disabled={isLoading}>
                                  Generate
                                  </Button>
                        </form>
                    </Form>
                </div>
                <div className="space-y-4 mt-4">
                    {/* Display a loading indicator when a request is in progress */}
                    {isLoading && (
                        <div className="p-8 rounded-lg w-full flex items-center
                            justify-center bg-muted">
                                <Loading/>

                        </div>
                    )}
                      {/* Display a message if no conversation has started yet */}
                    {!video && !isLoading && (
                        <Empty label=" No video generated"/>
                    )}
                      {/* Render the videos */}
                    {video && !isLoading && (
                        <video className="w-full aspect-video mt-8 rounded-lg border bg-black" controls>
                            <source src={video} />
                        </video>
                    )}
                </div>
            </div>
        </div>
    );
}

export default VideoPage;
```

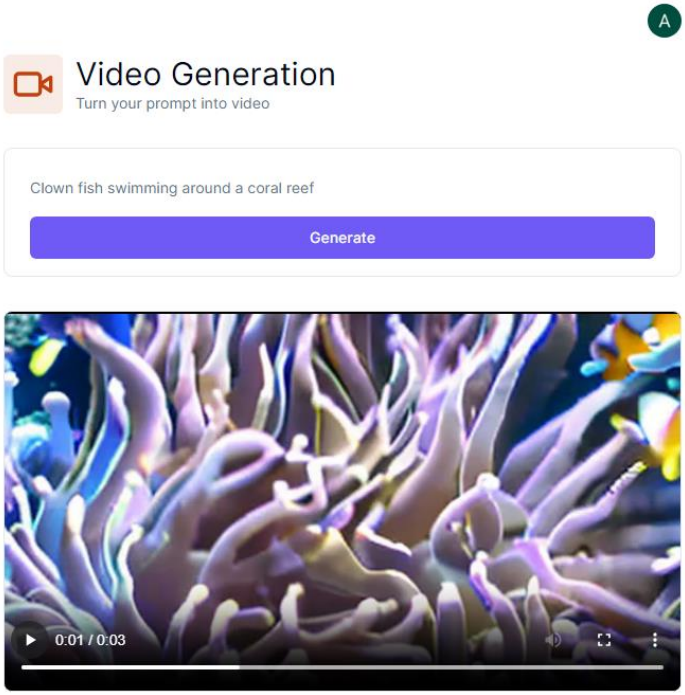
The ‘Heading’ component's title and description were changed to "Video Generation" and "Turn your prompt into video," respectively, to accurately describe the tool's functionality as well as the color scheme and the icon used.

The ‘onSubmit‘ function was modified to handle video generation requests. Upon submission, it sends a request to the /api/video endpoint with the user's input values.

The response handling was adjusted for video content. Instead of audio files, the response is expected to contain video data. The state ‘setVideo‘ is updated with the first item from the response array.

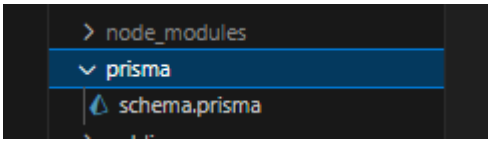
Now for the ‘route.ts’ inside the api folder, a new model was used entitled: zeroscope-v2-xl. Here’s some info on how to use it: https://replicate.com/anotherjesse/zeroscope-v2-xl/api?tab=nodejs. After following the instructions in the link provided, the changes for the video generation route were made.

This is how a response from the video generation page will look like:

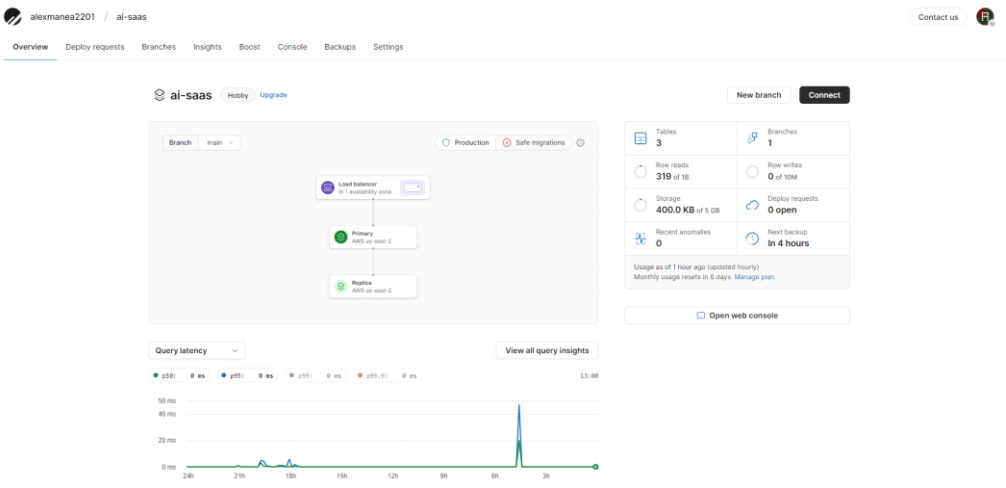


What we're going to go into now is going over the subscription and the API limits. The users will be able to use up to 5 free generations, after that they will have to pay. To do this prisma is used. Prisma is a next-generation ORM(object-relational mapping) that can be used to access a database in Node.js and TypeScript applications. It is used as an alternative to writing plain SQL, or using another database access tool such as SQL query builders.

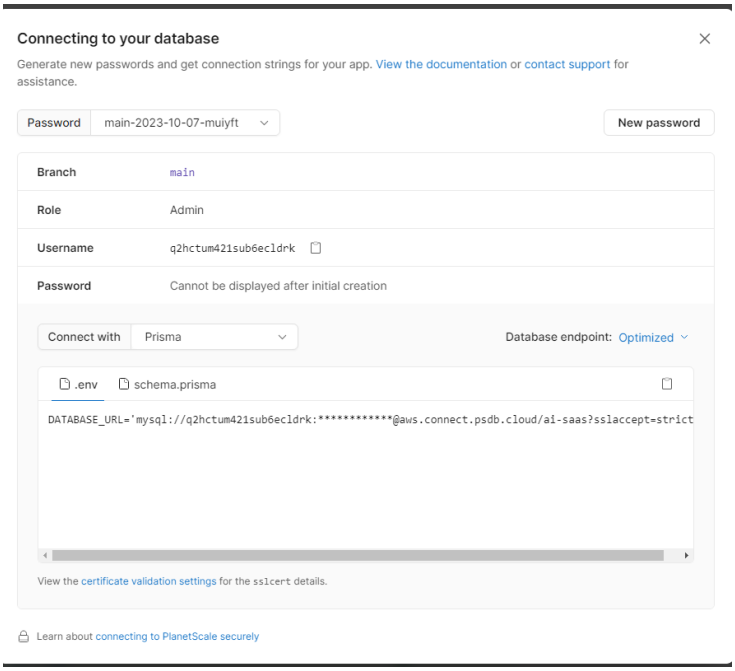
First things first, prisma should be installed using the following command: ***npm i -D prisma*.** To create the prisma file ***npx prisma init*** will be run. After executing this a new folder should be created containing the schema.prisma inside.



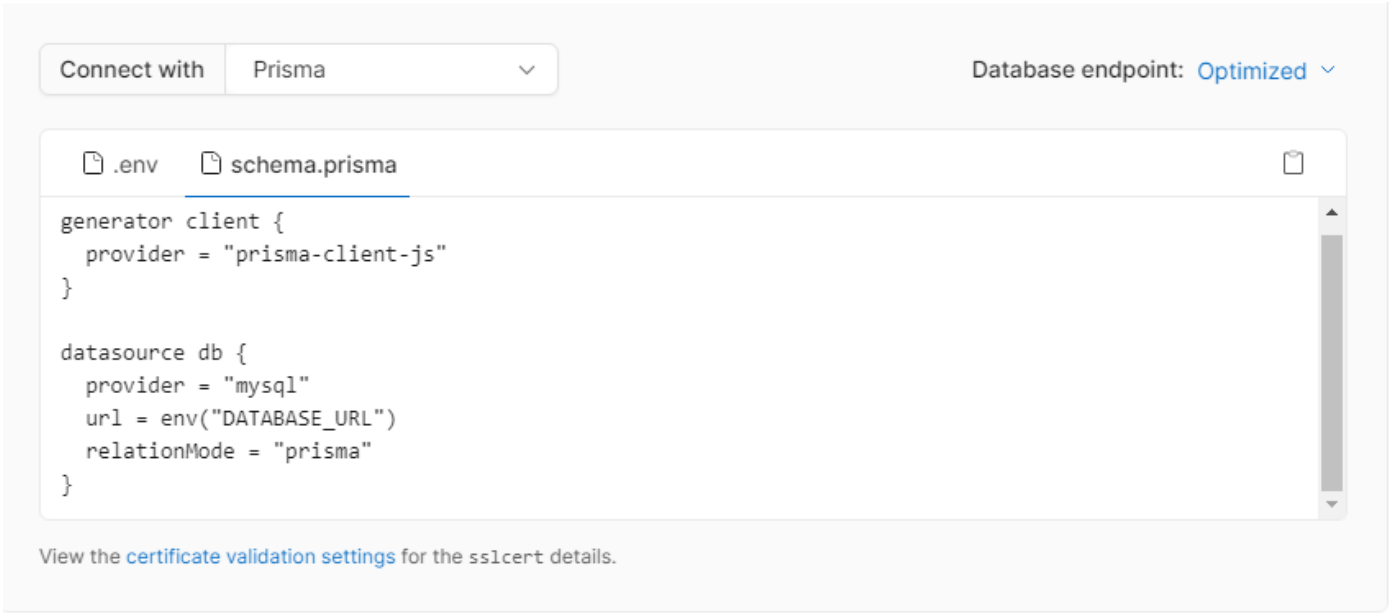
After this I created an account on planetscale and created a database https://app.planetscale.com/ selecting the free tier. It should look like this after it's created.



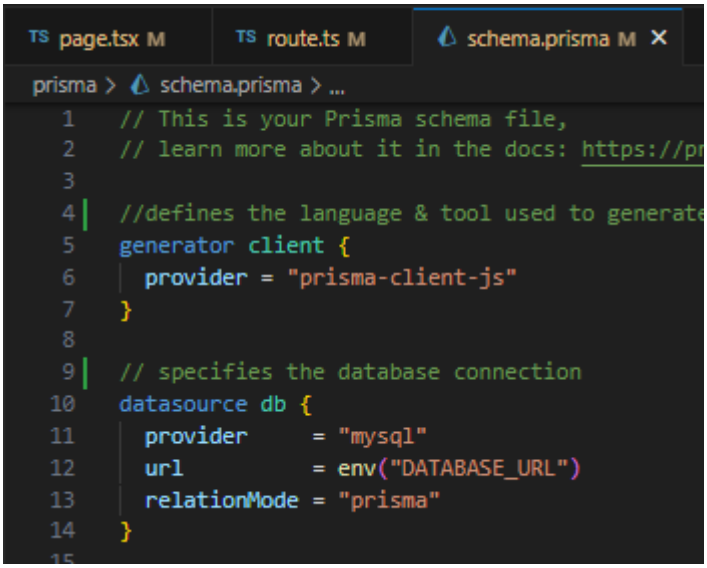
Then you can choose how to connect to your database, and insert the 'DATABASE_URL' in the .env file



After that I went to ‘schema.prisma’ copied the contents and pasted it inside the code editor:



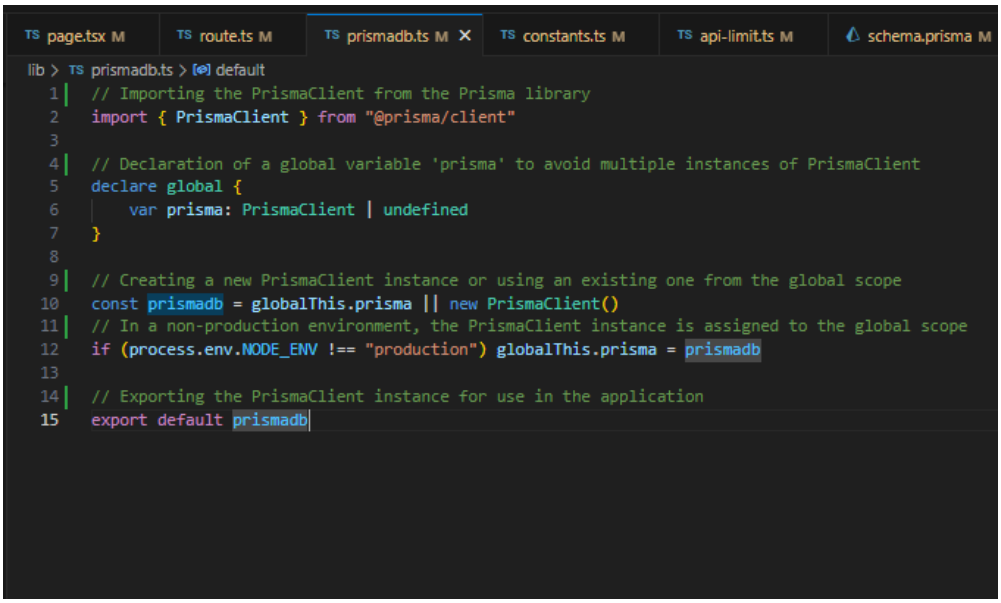
Learn about connecting to PlanetScale securely



After this, the package is added***: npm i @prisma/client***. The package @prisma/client is the Prisma Client library. It is an auto-generated, type-safe query builder used to programmatically read and write data in the database.

When you run the command, it installs the Prisma Client library in the Node.js project, enabling you to interact with your database using simple JavaScript or TypeScript commands**.**

Now inside the lib folder, 'prismadb.ts' file will be created



The 'prismadb.ts' file initializes and exports a single instance of Prisma Client. This instance is used throughout the application to interact with the database.

To avoid creating multiple instances of Prisma Client during development, which can lead to excessive memory use and performance issues, the code checks if an instance already exists in the global scope (globalThis.prisma). If it exists, it uses the existing instance, otherwise, it creates a new one.

The environment check (process.env.NODE_ENV !== "production") is used to assign the Prisma Client instance to the global scope only during development (development environment). In production, it's not advisable to attach instances to the global scope as it can lead to unexpected behaviors and security issues.

Hot Reload in Development: In development environments, particularly in Node.js applications, hot reloading is a common feature. It automatically restarts or refreshes the application when file changes are detected. This feature speeds up development by reflecting changes without manual restarts.

Each time the application restarts due to hot reloading, a new instance of Prisma Client might be created. If these instances accumulate, they can consume unnecessary resources and potentially lead to issues. To solve this, the 'prismadb.ts' file checks if a Prisma Client instance already exists in the global scope before creating a new one. This way, even if the application restarts multiple times due to hot reloading, it reuses the same Prisma Client instance, preventing the creation of multiple instances and conserving resources.

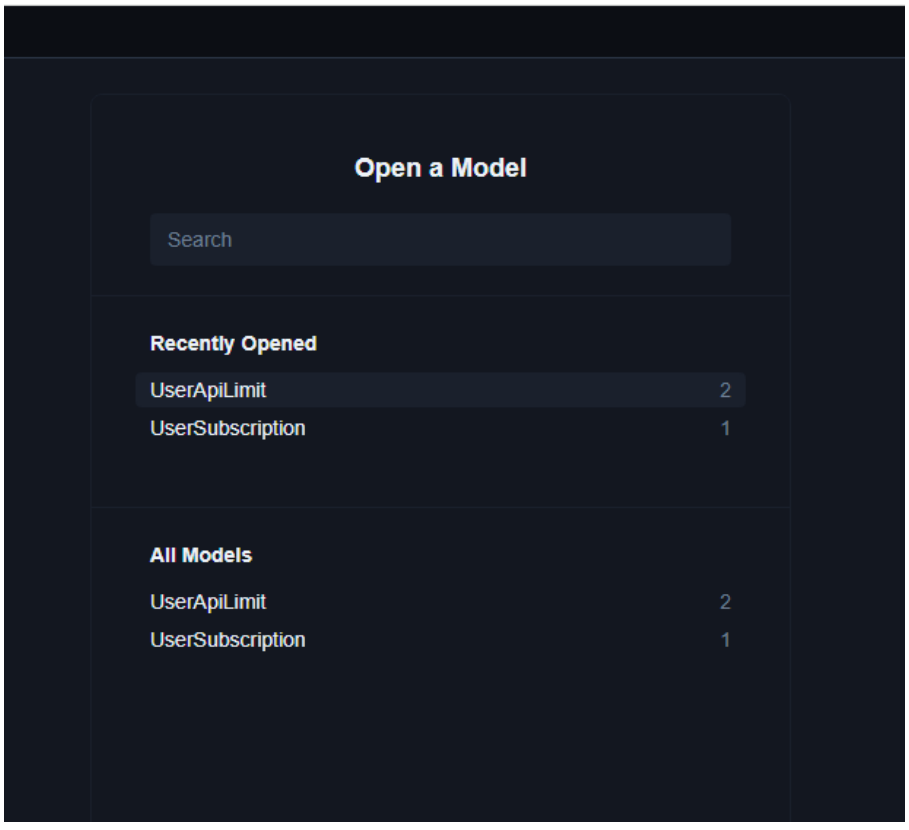
After this the model is created in 'schema.prisma':

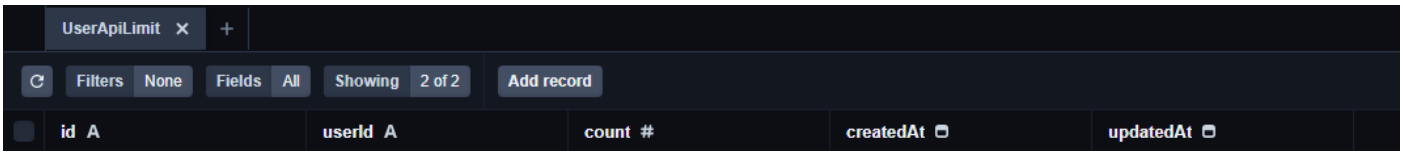
```prisma
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

//defines the language & tool used to generate the client
generator client {
  provider = "prisma-client-js"
}

// specifies the database connection
datasource db {
  provider     = "mysql"
  url          = env("DATABASE_URL")
  relationMode = "prisma"
}

// Model for UserApiLimit: Tracks API usage limits for each user
model UserApiLimit {
  id        String   @id @default(cuid())
  userId    String   @unique
  count     Int      @default(0)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}
```

To push the model to the database and to add the model to the node modules so then you have the types and no errors while developing, the following commands are run in the terminal: ***npx prisma generate*** and ***npx prisma db push***

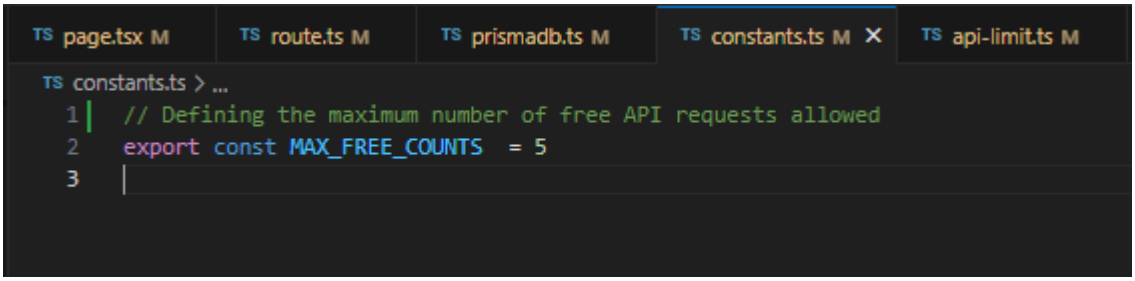
To visualize the data you can run ***npx prisma studio.***

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Alex\Desktop\ai-platform> npx prisma studio
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Prisma Studio is up on http://localhost:5555
```





The focus will now be on implementing the a way to check the amount of generations the user has used. First, in the ‘constants.ts’ file in the folder the free generations amount will be defined as 5.



In the lib folder, the file ‘api-limit.ts’ will have the functions responsible for checking the and updating the number of generations.

Import Statements:

auth from @clerk/nextjs is used for authentication, particularly for identifying the current user.
prismadb from ./prismadb is the Prisma Client instance for interacting with your database.
MAX_FREE_COUNTS from @/constants is as we discussed, the constant that defines the maximum number of API calls a free tier user can make.

increaseApiLimit Function:
Increments the API usage count for a user each time they make an API call. It first retrieves the user ID from the authentication context. Checks if there's an existing record for the user in the 'UserApiLimit' model. If a record exists, it updates the count. If no record exists (first-time user or new user), it creates a new entry with a count of 1.

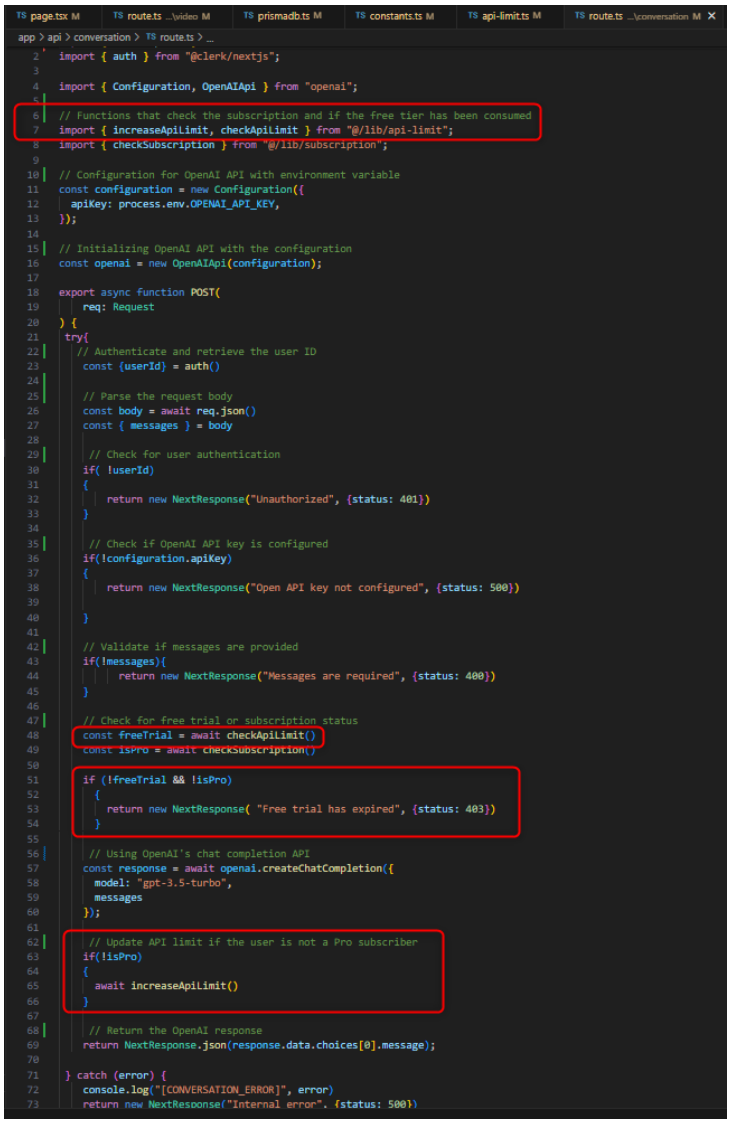
checkApiLimit Function:

Checks if a user has exceeded their API usage limit. Similar to increaseApiLimit, it first identifies the user and then retrieves the user's API limit record from the database. Compares the current count with the MAX_FREE_COUNTS and returns true if the user is within the limit (hasn't exceeded) and false otherwise.

getApiLimitCount Function:

Retrieves the current API usage count for a user. First it, identifies the user and fetches their UserApiLimit record and then returns the current count or 0 if no record exists.

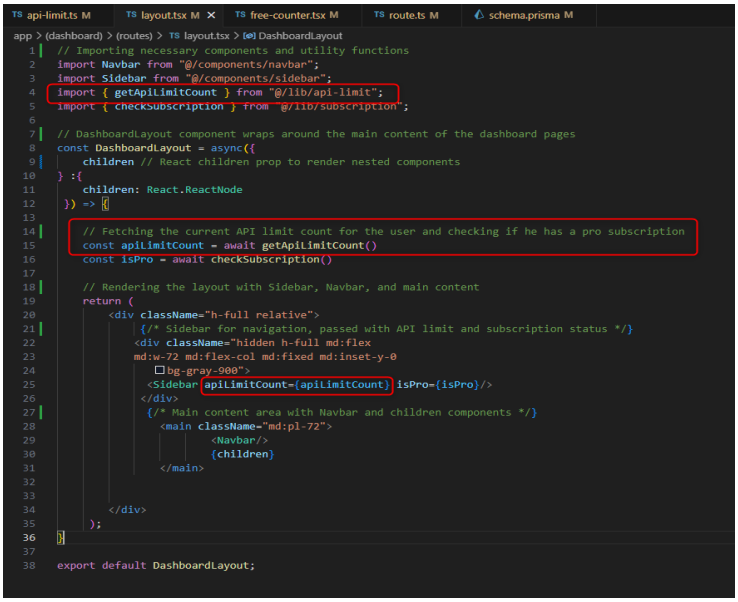
These functions will be used in all of the api routes, protecting them for excessive usage, an example would be this, in the conversation route.



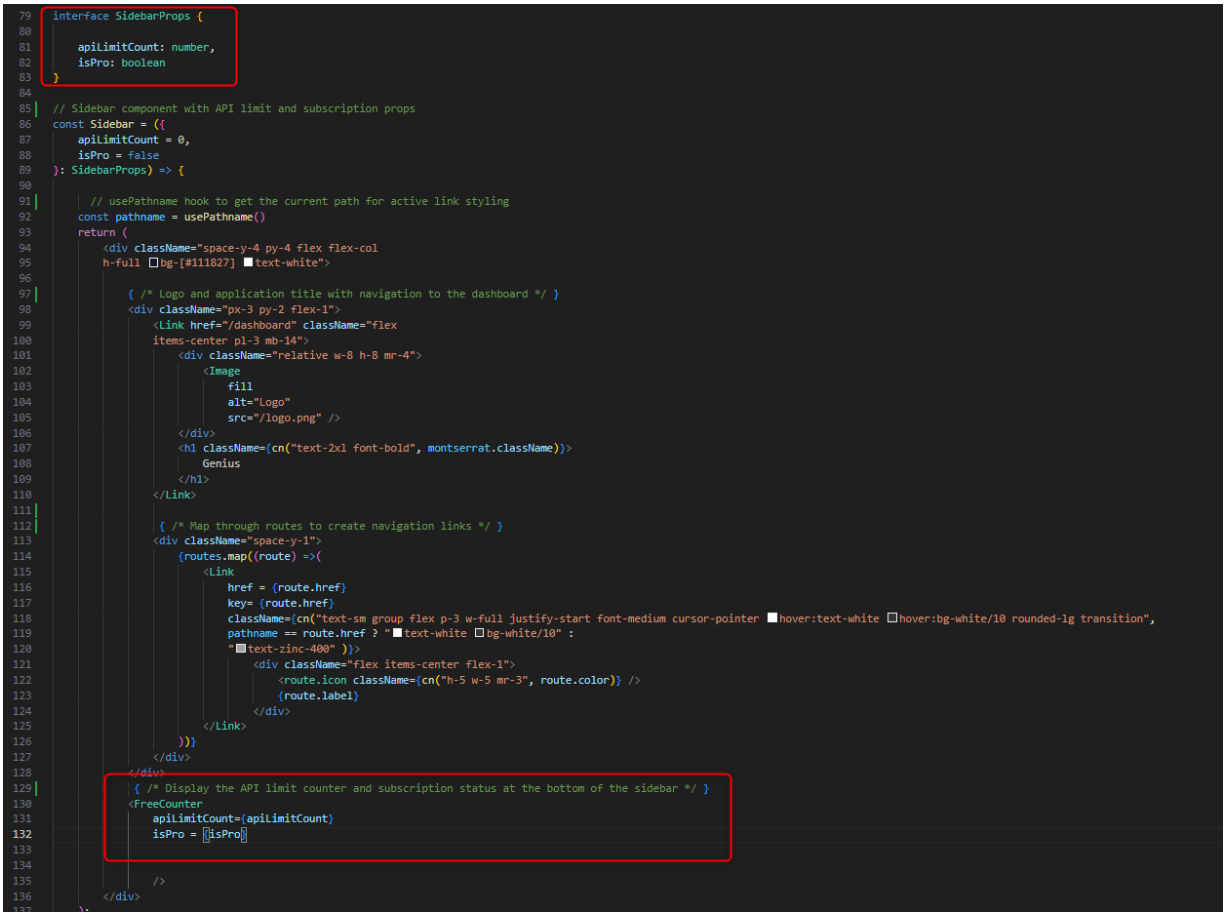
In summary:

- schema.prisma is the database structure blueprint, dictating how data is organized in the PlanetScale database.
- prismadb.ts creates and manages the instance of Prisma Client, enabling the application to communicate effectively with the database.
- api-limit.ts leverages the Prisma Client to enforce API usage limits, ensuring fair and sustainable access to the application's features.

Now, the API limit counter will be showcased. In 'DashboardLayout' ((dashboard)/(routes)/layout.tsx), 'getApiLimitCount' is invoked asynchronously to fetch the current API limit count.



This count, is passed as props to the Sidebar as it was seen when discussed about this component, that's why an interface is defined, to accommodate the props.



From the Sidebar, the counter is passed once more to a custom component called “FreeCounter” which will be in charge of visually representing the free tier.

```tsx
// Importing necessary hooks and components
import { useEffect, useState } from "react"
import { MAX_FREE_COUNTS } from "@/constants"
import { Card, CardContent } from "@/components/ui/card"
import { Progress } from "@/components/ui/progress"
import { Button } from "@/components/ui/button"
import { Zap } from "lucide-react"
import { useProModal } from "@/hooks/use-pro-modal"

// Interface for the FreeCounter props
interface FreeCounterProps{

    apiLimitCount: number
    isPro: boolean

}

// FreeCounter component displays the user's API usage and upgrade button
export const FreeCounter = ({
    apiLimitCount = 0,
    isPro = false
}: FreeCounterProps) => {
    const proModal = useProModal()

    // State to manage component mounting and setting the state when the component is mounted with useEffect
    const [mounted, setMounted] = useState(false)

    useEffect(() =>{
        setMounted(true)
    }, [])

    // Return nothing if not mounted or user is Pro subscriber
    if(!mounted)
    {
        return null
    }

    if(isPro){
        return null
    }

    // Component content for free tier users
    return (
        <div className="px-3">
            <Card className="bg-white/10 border-0">
                <CardContent className="py-6">
                    {/* Displaying API usage and progress bar */}
                    <div className="text-center text-sm text-white mb-4
                    space-y-2">
                        <p>
                            {apiLimitCount} / {MAX_FREE_COUNTS} Free Generations
                        </p>
                        <Progress
                            className="h-3"
                            value={(apiLimitCount / MAX_FREE_COUNTS) * 100} />

                    </div>
                    {/* Button to trigger Pro subscription modal */}
                    <Button onClick={proModal.onOpen} variant="premium" className="w-full">
                        Upgrade
                        <Zap className="w-4 h-4 ml-2 fill-white" />
                    </Button>
                </CardContent>
            </Card>
        </div>
    )
}
```

FreeCounter Component Explanation

Imports and Props Definition:

Necessary hooks and components are imported (useEffect, useState, MAX_FREE_COUNTS, etc.).
An interface called ‘FreeCounterProps’ is defined to type-check the props (‘apiLimitCount’ and ‘isPro’) that the component receives.

Component Function:
‘FreeCounter’ is a component which accepts the ‘apiLimitCount’. Default values are set for the props

Mount State and useEffect:

A state variable mounted is initialized with false. This variable is used to manage the rendering of the component, ensuring it only renders on the client side after mounting.
The ‘useEffect’ hook updates mounted to true after the component mounts. This approach prevents any server-side rendering of the component, which is necessary to avoid hydration errors in Next.js.

Conditional Rendering:

If ‘mounted’ is false or ‘isPro’ is true, the component returns null and nothing is rendered. This means the component does not show up for Pro users or before it is properly mounted on the client side.

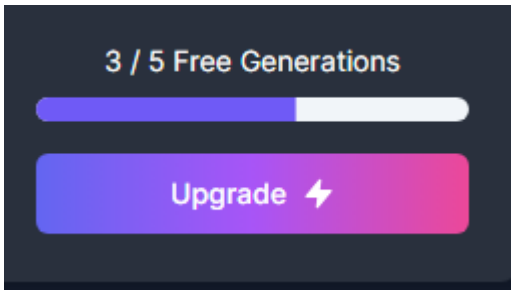
Component Content:

The component returns a Card component with a progress bar installed via shadcn ui (https://ui.shadcn.com/docs/components/progress) and a button.
The progress bar shows the proportion of the free API usage to the maximum free counts and the button is styled for upgrading to a Pro subscription.

To summarize, in the application, the 'getApiLimitCount' function is utilized within the 'DashboardLayout', which acts as a server component. This enables it to access the server-side functionality needed to retrieve API limit counts.
The count is then passed as a prop to the 'Sidebar' component, and again to the 'FreeCounter' component. The updating of the API limit count occurs due to the router.refresh() (example below In the image page) method in each AI tool's form submission. This refresh action rehydrates server components, ensuring they fetch the latest data from the database.

```
// Handling form submission
const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
    try{

      // Clearing existing images before making a new request
      setImages([])

      // Sending a POST request to the server with form values
      const response = await axios.post("/api/image", values)

      // Extracting image URLs from the response and updating state
      const urls = response.data.map((image: {url: string}) => image.url)

      setImages(urls)

       // Resetting the form to its default values after submission
      form.reset()

    // Handling specific error scenarios
    } catch (error: any){
        if(error?.response?.status === 403){
          proModal.onOpen()
       }else {
         toast.error("Something went wrong")
       }
        console.log(error)
    }
    finally{
        router.refresh()
    }

}
```

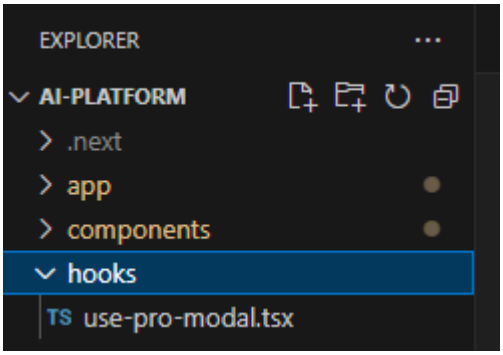
The Freecounter looks like this:

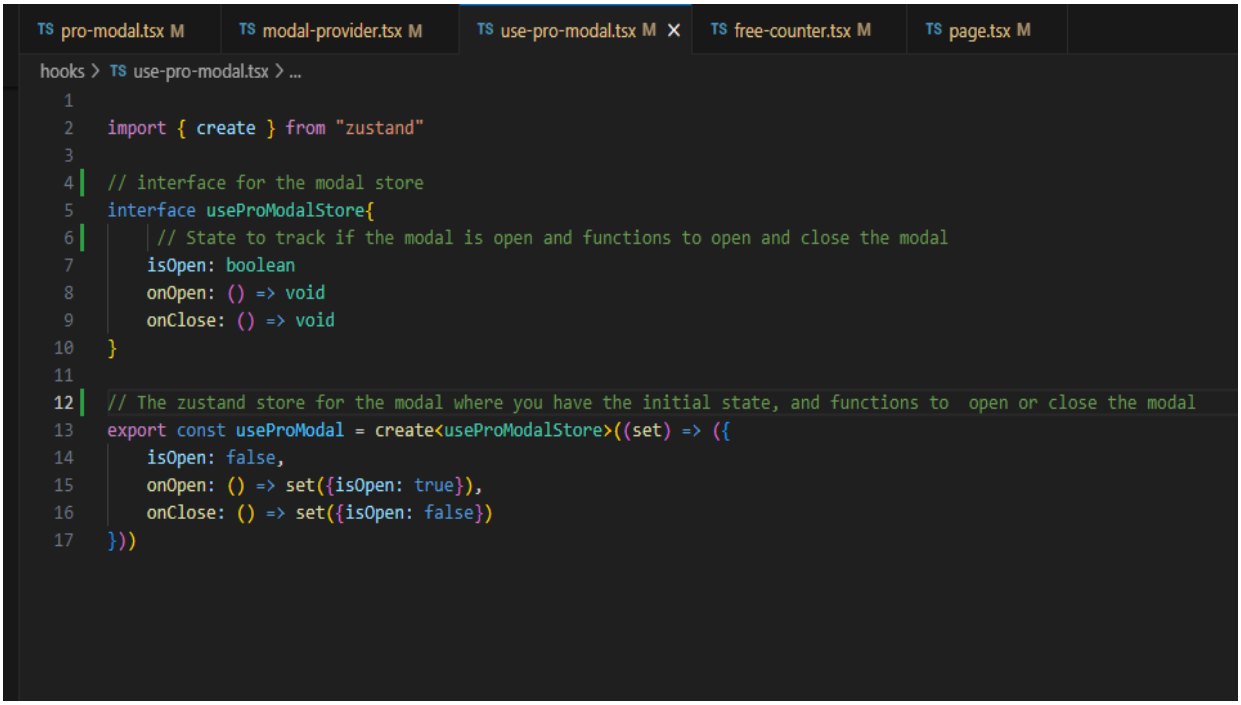


The next component is the pro modal that is connected to stripe and opens when the free generations are used or when the 'Upgrade' button is clicked.

For this the package called 'zustand' is needed. To install it you can run the following command in the terminal
npm i zustand

In the root folder of the application, the 'hooks' folder is added alongside the 'use-pro-modal.tsx'





Zustand is a lightweight state management solution. This package provides a way to manage global state in React applications.

The ‘use-pro-modal.tsx’ file defines the global state and functions to manage the modal’s visibility.

After creating the ‘use-pro-modal’ , the ‘modal-provider’ (inside the components folder) is added for managing the rendering of the modal on the client side. It ensures that the modal is not rendered on the server side, preventing hydration errors in Next.js.

Let’s discuss a bit about the differences between SSR and CSR.

Server-side Rendering (SSR) in Next.js

When a Next.js page is requested, the server prepares the HTML and CSS for the page. This includes executing the initial render logic of React components. But React methods (like useEffect) and browser functionalities (like accessing window or document) do not execute on the server. This is because these features require a browser, which is not available on the server.

Client-side Rendering (CSR) in Next.js

Once the SSR-generated HTML and CSS are sent to the client React takes over and the process is known as hydration. React "hydrates" the pre-rendered content, attaching event listeners and executing methods like ‘useEffect’ essentially turning the static content into a dynamic web application.

A hydration error occurs when the content generated by React on the client-side does not match the one sent from the server. This can happen if the React components behave differently on the server compared to the client.

Modals can rely on browser features and dynamically generated content (like showing or hiding based on user interaction). If a modal is rendered on the server and its content or state differs when ‘React’ tries to hydrate it on the client, it can lead to differences, causing hydration errors.

To avoid these errors, modals are mostly rendered only on the client-side. This is done by using techniques like checking if the component has mounted on the and rendering the modal only if it's on the client-side. This way, the server sends a version of the page without the modal, and the modal only gets rendered once React takes over in the browser, avoiding hydration errors.

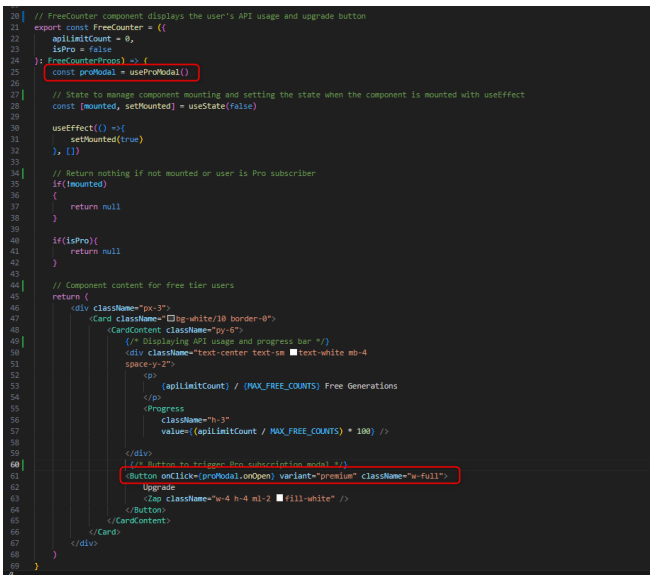
An example of a hydration error would be this. Imagine a component that displays the current time. On the server, it renders the time at which the server-side rendering happened. However, when the client-side React code takes over it might render the current time according to the client’s local time. If there’s a big time difference between the server and the client, React notices sees this and throws a hydration error.

Next we have the ‘pro-modal’ component looks like this:

And is made out of the following:

- Dialog Components: Used to structure the modal. ‘DialogHeader’, ‘DialogTitle’, ‘DialogDescription’, and ‘DialogFooter’ organize the content and layout and are shadcn components (https://ui.shadcn.com/docs/components/dialog)
- tools Array: Defines the tools available for upgrade. Renders a list of tools in the modal.
- onSubscribe Function: Handles the subscription process. On clicking the upgrade button, it initiates a request to the server and redirects the user to the payment page, in this case stripe which will be discussed about a bit further
- Loading State: Manages the interface state during the subscription process, disabling the button while the request is in progress.

The modal should open when the ‘Upgrade’ button in the ‘Sidebar’ is clicked. That is being done by adding the pro-modal inside the ‘free-counter.tsx’



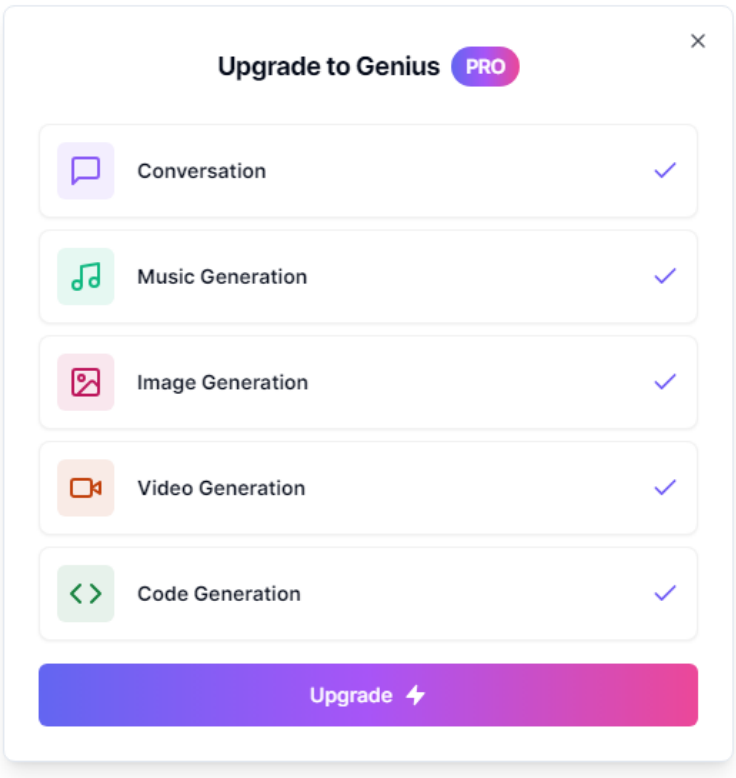
Another use case for the pro-modal opening is when we encounter a 403 error which was defined in the API routes and it represents that the access to the requested resource is forbidden.

```
if (!freeTrial && !isPro)
    {
      return new NextResponse( "Free trial has expired", {status: 403})
    }
```

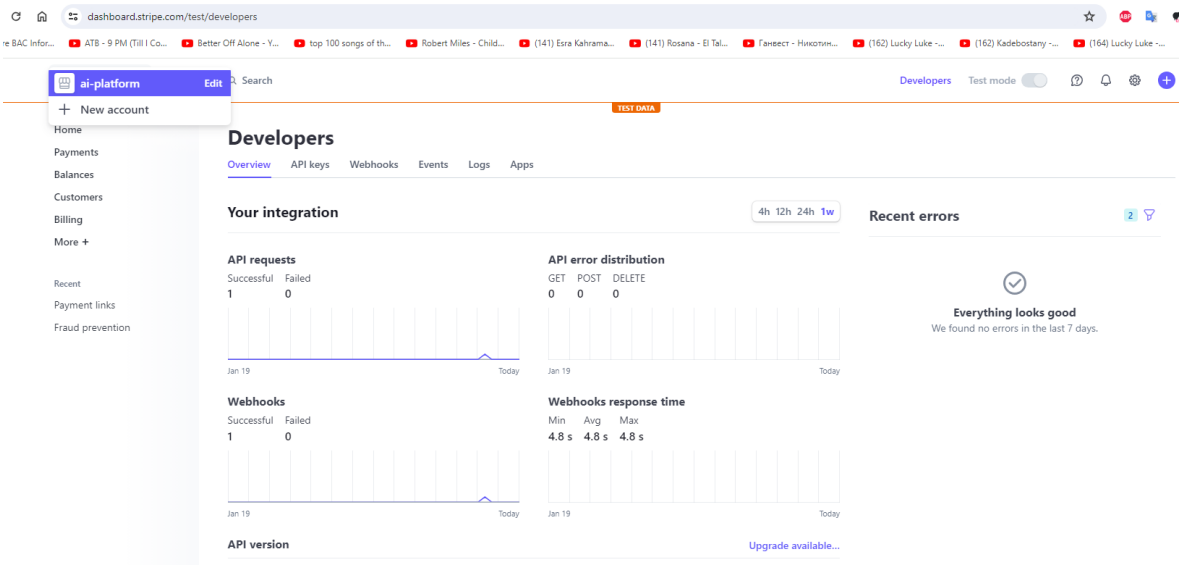
To open the modal when this status code is received is pretty straightforward. This can be done by modifying each tool, upon making the POST request and receiving the error code, and it can be observed below in the code page tool.

```
// Handling form submission
const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
    try{


      const userMessage: ChatCompletionRequestMessage = { role: "user", content: values.prompt };

      const newMessages = [...messages, userMessage]

      const response = await axios.post("/api/code", {
        messages: newMessages
      })

      setMessages((current) => [...current, userMessage, response.data])

      form.reset()


    } catch (error: any){
          if(error?.response?.status === 403){
            proModal.onOpen()
        }else {
          toast.error("Something went wrong")
        }
          console.log(error)
    } finally{
        router.refresh()
    }

}
```

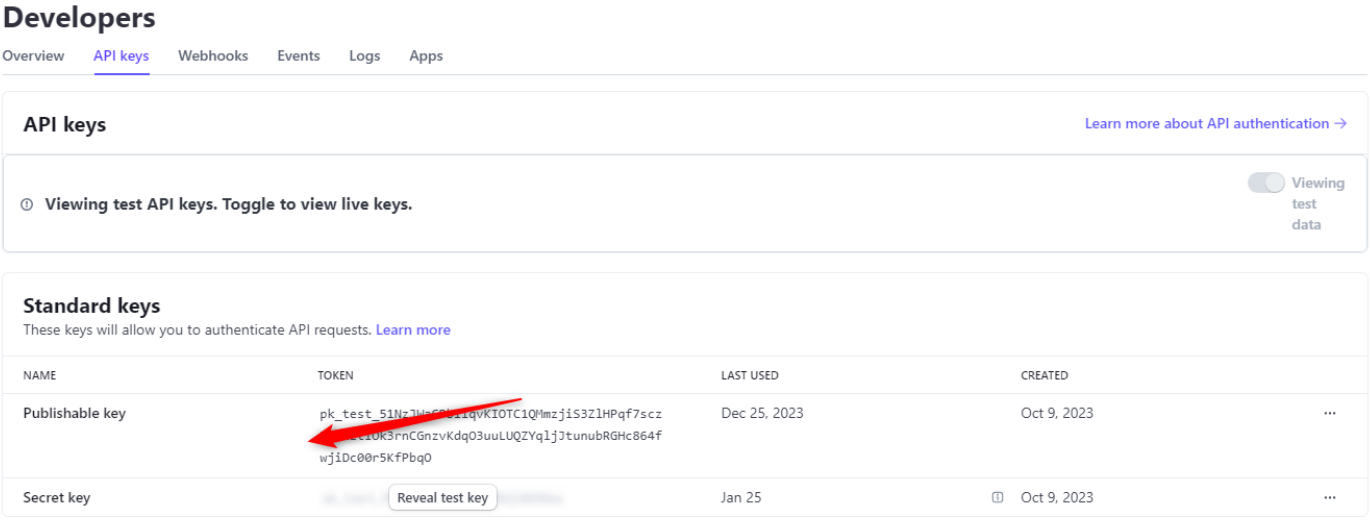
The pro modal has the following interface:



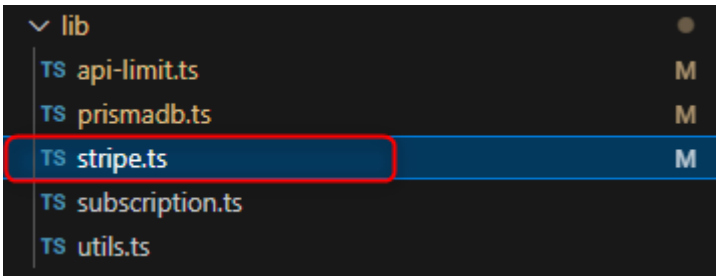
Next is integrating stripe in the application. First is making the stripe account https://dashboard.stripe.com/test/developers and then make a new account.



After that the API key can be taken from the API key tab and inserted in the .env file.

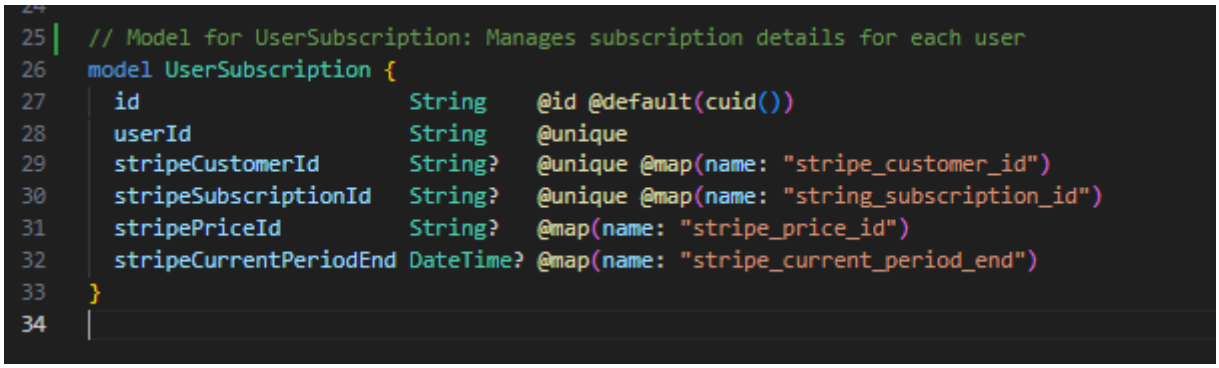


Inside the terminal, a package for stripe will be installed by running ***npm i stripe***
Inside the lib folder, a library for stripe will be made.



And a stripe instance will be created:

Now in the schema.prisma, a new model will be created for the subscription.



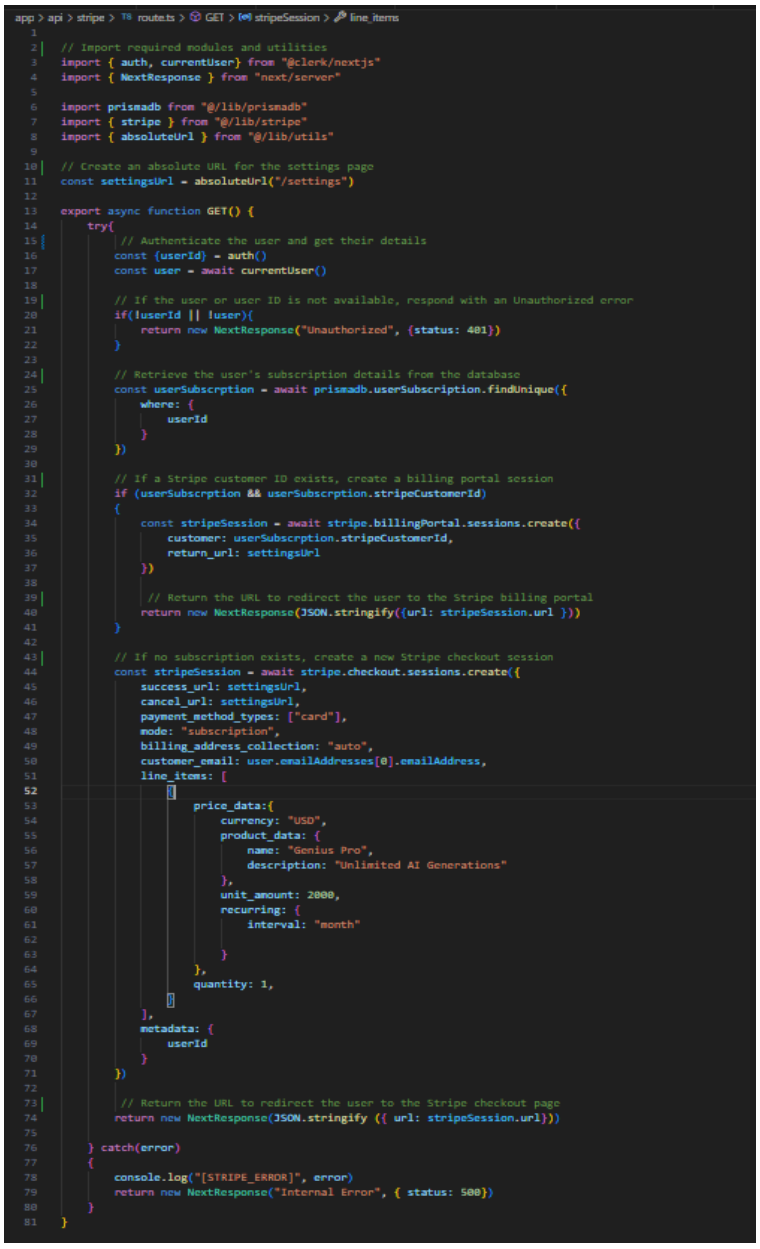
The ‘UserSubscription’ model is designed to store and manage subscription information for each user, particularly focusing on integrating with Stripe's subscription features. The model uses fields to keep track of Stripe's customer ID, subscription ID, price ID, and the current period end date. The @map is used to align the database column names with Stripe's API field names

After this ***npx prisma generate*** and ***npx prisma db push*** are ran to push the table to the database and update the node modules with the model.

Next is the stripe api route which is created inside the api folder under the stripe folder, but before doing so its necessary to establish a utility function that ensures Stripe can reliably interact with the application's URLs. This necessity comes from the requirement to provide absolute URLs to Stripe for callbacks and redirections.

The base URL of the application is stored in an environment variable named NEXT_PUBLIC_APP_URL. This variable will be set to the root of the application, in this case (http://localhost:3000). The ‘absoluteUrl’ function takes a path as an argument and returns the complete absolute URL by appending the path to the base URL.

After this the stripe route was developed



The Stripe route is designed to handle user subscriptions for paid features. It interacts with Stripe's API to manage billing and subscription processes.

Key Components of the Stripe Route:

User Authentication: The route first authenticates the user using Clerk's 'auth' and 'currentUser' methods. This makes sure that only authorized users can access Stripe.

Subscription Retrieval: It then checks if the user has an existing subscription by querying the prismadb

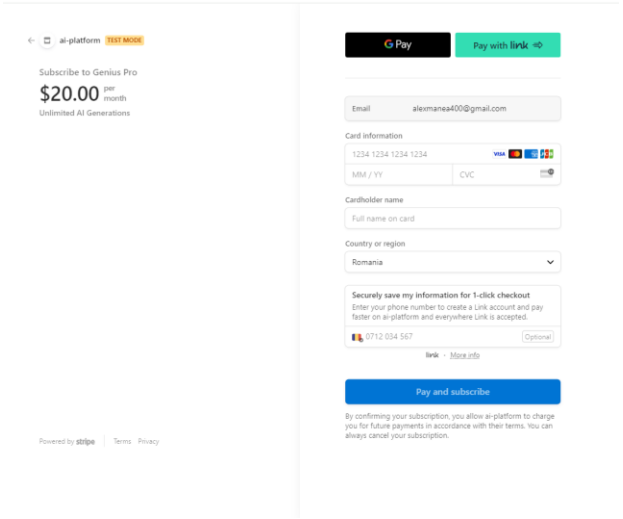
Billing Portal Session: If the user already has a Stripe customer ID (basically an active subscription), the route creates a session for Stripe's billing portal, allowing users to manage their subscription.

Stripe Checkout Session: For new users or users without an active subscription, the route creates a Stripe checkout session. This session is configured for subscription payments, with details like price, product description, and customer email.

Metadata for User Association: The checkout session includes metadata containing the user's ID. This metadata is important for associating the successful payment with the correct user account in the following webhook processing.

Response Handling: Depending on the user's subscription status, the route responds with a URL redirecting the user to either the Stripe billing portal or the Stripe checkout page.

This is how the stripe portal looks like:



After this, the stripe webhook is created:

pro-modal.tsx M | sidebar.tsx M | user-avatar.tsx M | modal-provider.tsx M | use-pro-modal.tsx M | api-limit.ts M

app > api > webhook > route.ts > POST > data > stripeCurrentPeriodEnd

```ts
// Import necessary modules
import Stripe from "stripe"
import { headers } from "next/headers"
import { NextResponse } from "next/server"
import prismadb from "@/lib/prismadb"
import { stripe } from "@/lib/stripe"

export async function POST(req: Request) {

    // Retrieve the raw body and Stripe signature from the request
    const body = await req.text()
    const signature = headers().get("Stripe-Signature") as string

    let event: Stripe.Event

    try{
        // Construct the event using Stripe library to validate the webhook
        event = stripe.webhooks.constructEvent(
            body,
            signature,
            process.env.STRIPE_WEBHOOK_SECRET!

        )
    } catch (error: any){
        // Return a 400 response for any webhook errors
        return new NextResponse(`Webhooks Error: ${error.message}`, {status: 400})
    }

    // Extract the session information from the event data
    const session = event.data.object as Stripe.Checkout.Session

    // Handle 'checkout.session.completed'
    if(event.type === "checkout.session.completed"){
        // Retrieve the subscription details from Stripe
        const subscription = await stripe.subscriptions.retrieve(
            session.subscription as string
        )
    // Check if user ID metadata is present
    if (!session?.metadata?.userId){
        return new NextResponse("User id is required", {status:400})
    }

    // Create a new subscription record in the database
    await prismadb.userSubscription.create({
        data: {
            userId: session?.metadata?.userId,
            stripeSubscriptionId: subscription.id,
            stripeCustomerId: subscription.customer as string,
            stripePriceId: subscription.items.data[0].price.id,
            stripeCurrentPeriodEnd: new Date(
                subscription.current_period_end * 1000
            )

        }
    })

    }
    // Handle 'invoice.payment_succeeded' event type
    if (event.type === "invoice.payment_succeeded"){
       const subscription = await stripe.subscriptions.retrieve(
            session.subscription as string
        )

        // Update the subscription in the database
        await prismadb.userSubscription.update({
            where: {
                stripeSubscriptionId: subscription.id,
            },
            data: {
                stripePriceId: subscription.items.data[0].price.id,
                stripeCurrentPeriodEnd: new Date(
                    subscription.current_period_end * 1000
                )
            }

        })
    }

    // Return a 200 response indicating successful handling of the webhook
    return new NextResponse(null, {status: 200})

}
```

A webhook is an automated message sent from apps when something happens. They usually have a message and are sent to a unique URL. When using Stripe, webhooks are used to receive notifications about events that happen in a Stripe account, like successful payments, subscription creation.

When Stripe sends a webhook event, it includes a body (event data) and a signature. The body contains details about the event, like what type of event it is and it's data. The signature is used to verify that the event is from Stripe. The 'stripe.webhooks.constructEvent' method takes the body, signature, and the Stripe webhook secret to construct and validate the event. If the event is valid, it means it's from Stripe.

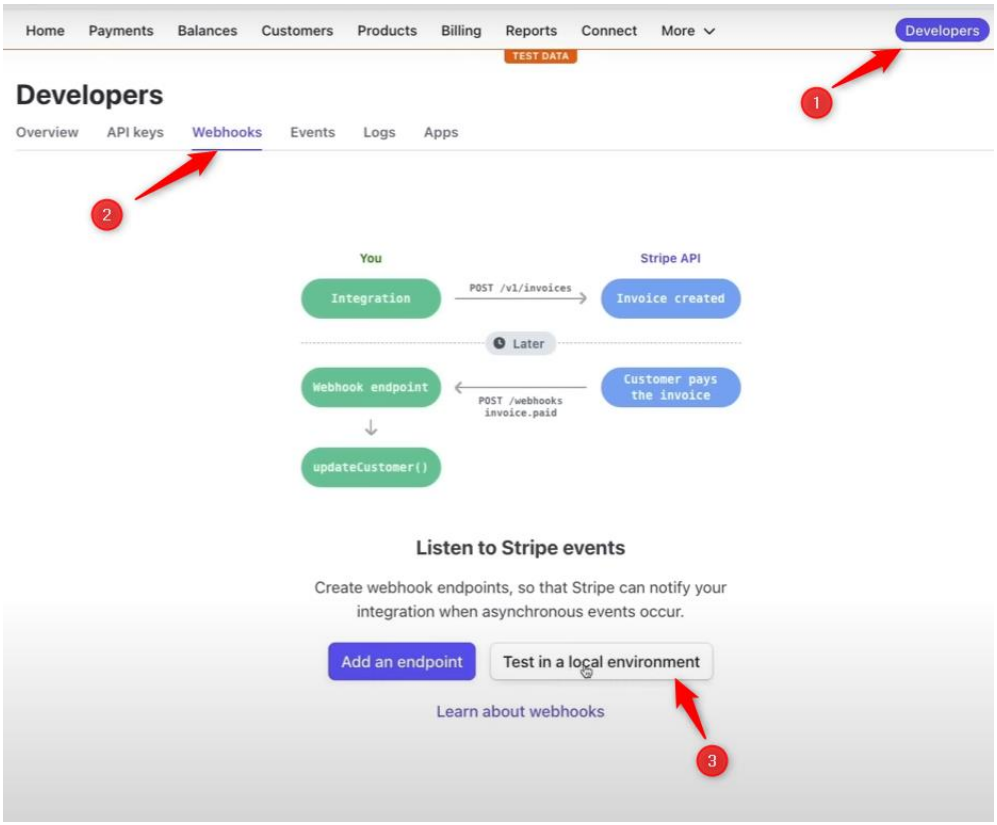
The webhook is then set up to handle two specific Stripe event types:

- 'checkout.session.completed': Occurs when a new subscription is successfully created.
- 'invoice.payment_succeeded': Occurs when a subscription payment is successfully processed.

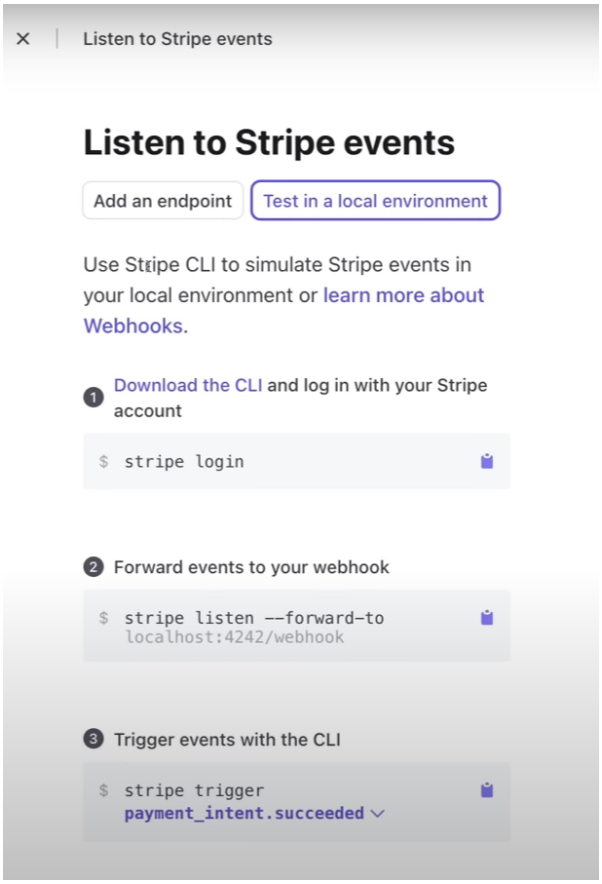
For 'checkout.session.completed': The webhook creates a new record in the database with the subscription details. This includes the user ID, subscription ID, customer ID, price ID, and the period end date.
For 'invoice.payment_succeeded': It updates the existing subscription record with the new price ID and period end date.

The webhook checks if the 'userId' metadata is present in the session. If not, it responds with an error, as this information is crucial for linking the Stripe event to the correct user in the database. After processing the events, the webhook responds with a status of 200 to represent the success in handling the event.

The webhook also relies on environment variables like STRIPE_WEBHOOK_SECRET for security checks. This secret is used to validate the Stripe event signature. To create the STRIPE_WEBHOOK_SECRET, you need to go to the stripe page, click on 'Developers' and then 'Webhooks'

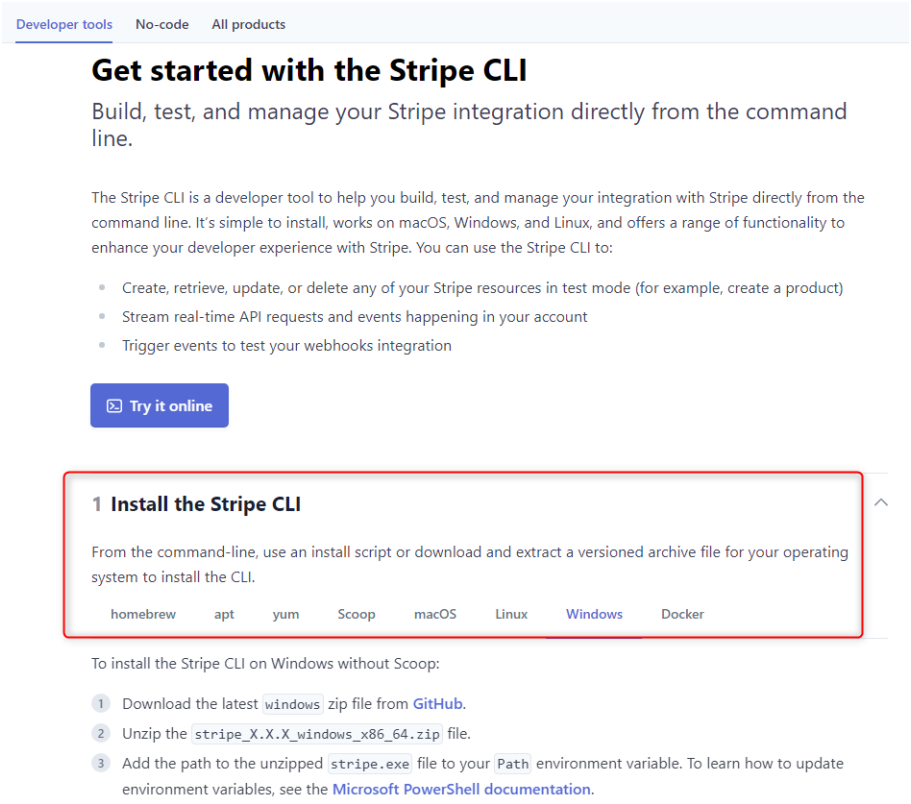


After this, these steps need to be followed:

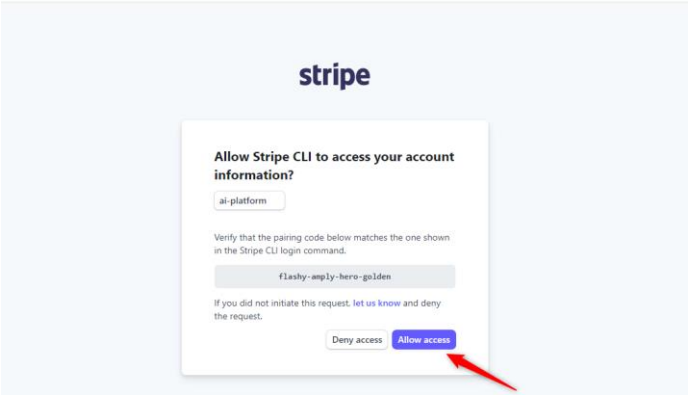


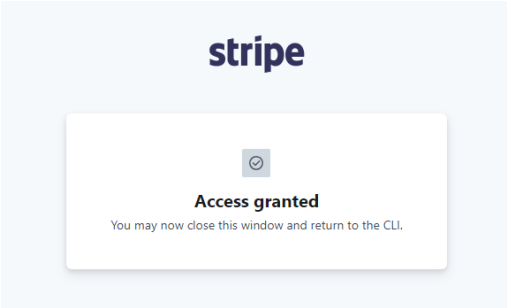
Below is a demonstration of the steps described:

1. Install Stripe and then log in from your terminal https://stripe.com/docs/stripe-cli



Go to the link and grant access:






```
PS C:\Users\Alex\Desktop\ai-platform> stripe login
Your pairing code is: flashy-amply-hero-golden
This pairing code verifies your authentication with Stripe.
Press Enter to open the browser or visit https://dashboard.stripe.com/stripecli/confirm_auth?t=oE499dUAxxuaxnkNvecZtyRQrUDDTmod (^C to quit)
> Done! The Stripe CLI is configured for ai-platform with account id acct_1NzJWaCPbIiqvKIO

Please note: this key will expire after 90 days, at which point you'll need to re-authenticate.
PS C:\Users\Alex\Desktop\ai-platform>
```

2. Listen to the webhook and get the STRIPE_WEBHOOK_SECRET

```
Please note: this key will expire after 90 days, at which point you'll need to re-authenticate.
PS C:\Users\Alex\Desktop\ai-platform> stripe listen --forward-to localhost:3000/api/webhook
A newer version of the Stripe CLI is available, please update to: v1.19.2
> Ready! You are using Stripe API Version [2023-08-16]. Your webhook signing secret is whsec_[redacted] (^C to quit)
```

After this you need to keep this terminal page open when you're developing so stripe listens to your webhook.

Flow of Stripe Subscription and Webhook:

- User Initiates Subscription via GET Route (Stripe Route) : This happens when a user application decides to subscribe. They are directed to the Stripe checkout page via the GET route set up.
- User Completes Subscription on Stripe: The user enters their payment details on the Stripe checkout page and completes the subscription process.
- Stripe Processes the Subscription: After the user submits their payment information, Stripe processes the subscription and payment.
- Stripe Triggers the Webhook: Once Stripe successfully processes the subscription, it sends out an event notification (like 'checkout.session.completed') to the webhook URL.
- Webhook Receives the Event: Server receives this event notification. The webhook's job is to listen for these Stripe events and perform actions in response, such as updating the database to show the user's new subscription status.

The webhook is triggered as a result of the actions taken on the Stripe checkout page, which is linked to the GET route (Stripe Route). It's a sequential process in that the webhook is called after the user completes the subscription process, but it's important to note that the webhook is not called by the GET route directly. Instead, it's triggered by Stripe once it completes its internal processing of the subscription or update of subscription.

Basically, the stripe route is the starting point of the subscription process. It guides the user to Stripe's checkout page to enter payment details and confirm their subscription. After the user completes the subscription, the webhook plays a crucial role in synchronizing the subscription status with the application's database. It ensures that the application is aware of the user's subscription status as soon as Stripe processes it and allows for real-time updates in the system.

Now in the Pro Modal, the 'onSubscribe' function was added and it is triggered by the 'Upgrade' button. This will take the user to Stripe's checkout page and upon subscription, Stripe will trigger the webhook.

```
export const ProModal = () => {

    const proModal = useProModal()

    const [loading, setLoading] = useState(false)

    // Function to handle the subscription process
    const onSubscribe = async() =>{
      try{
          setLoading(true)
          const response = axios.get("/api/stripe") // Request subscription URL
          window.location.href = (await response).data.url // Redirect to payment page

      } catch (error)
      {
        toast.error("Something went wrong")
      }finally{
        setLoading(false)
      }

    }
```

Before creating the Settings page from where the user can access his subscription, a new util is created to check if the user has a subscription or not and if it's not expired.

```
import { auth } from "@clerk/nextjs";

import prismadb from "./prismadb";

// Define a constant for a day in milliseconds
const DAY_IN_MS = 86_400_000

export const checkSubscription = async()  =>{

    // Retrieve the current user's ID
    const {userId} = auth()

    if(!userId){
        return false
    }

    // Retrieve the user's subscription details from the database
    const userSubscription = await prismadb.userSubscription.findUnique({
        where: {
            userId: userId
        },

        // Select specific fields related to the subscription
        select: {
            stripeSubscriptionId: true,
            stripeCurrentPeriodEnd: true,
            stripeCustomerId: true,
            stripePriceId: true
        }
    })

    // Return false if no subscription is found for the user
    if(!userSubscription){
        return false
    }

    // Check if the subscription is valid and not expired
    const isValid = userSubscription.stripePriceId && userSubscription.stripeCurrentPeriodEnd?.getTime()! + DAY_IN_MS > Date.now()

    return !!isValid
}
```

The function first checks if a user is logged in by retrieving the 'userId' . If no user is logged in the function returns false. Then it fetches the user's subscription details from the database using Prisma.
The functions checks if the subscription is valid. It ensures that:

- The subscription has a 'stripePriceId' , indicating an active subscription.

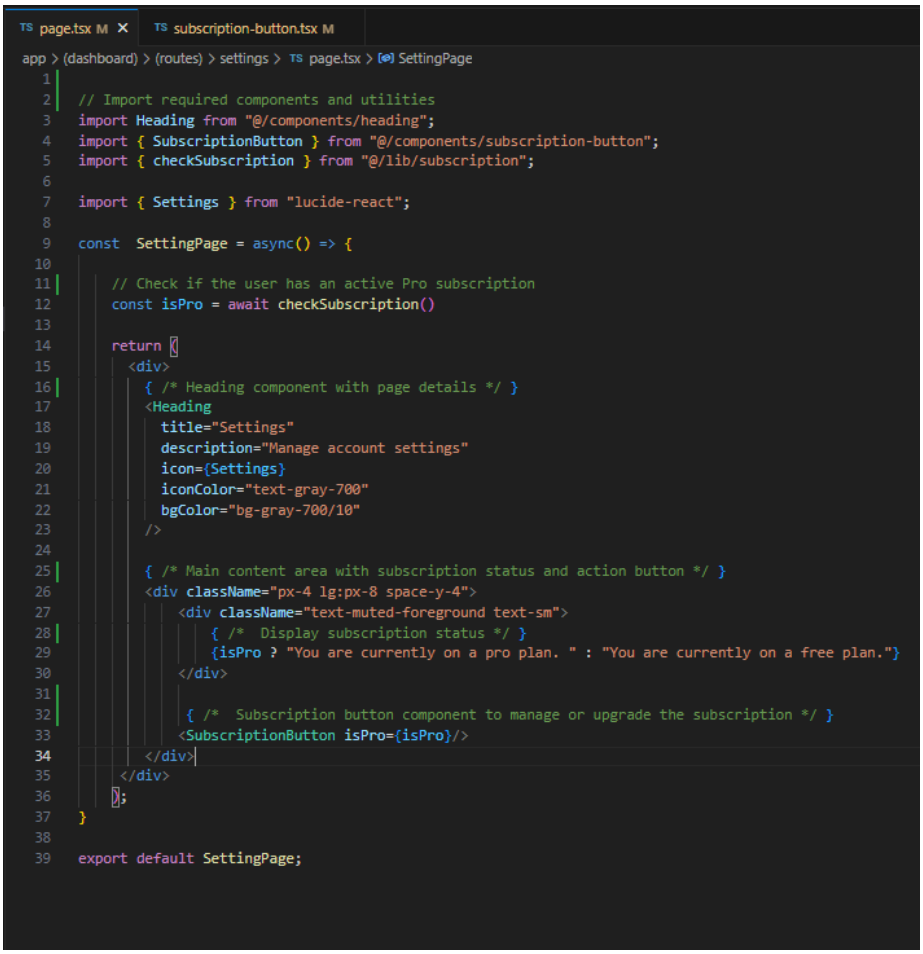
- The subscription hasn't expired. This is determined by adding one day (DAY_IN_MS) to the ‘stripeCurrentPeriodEnd’ and comparing it with the current date.

Exclamation Marks:

- getTime()! The exclamation mark here is a TypeScript non-null assertion operator, it ‘says’ that ‘stripeCurrentPeriodEnd’ is not null or undefined.
- !!isValid: The double exclamation marks convert the result to a Boolean value. This ensures that ‘checkSubscription’ always returns a Boolean. In some cases, especially in JavaScript, a value can be ‘truthy’ or ‘falsy’ without being strictly a Boolean type. For example, null, undefined, 0, "" (empty string), and NaN are ‘falsy’, but they are not strictly the Boolean value false. The first exclamation mark ! negates the value. For a ‘truthy’ isValid, !isValid becomes false, and for a ‘falsy’ isValid, !isValid becomes true. The second exclamation mark negates it again, turning it back to its original truthiness but as a strict Boolean value. So, a ‘truthy’ isValid becomes true, and a ‘falsy’ isValid becomes false.

The utility function is crucial for determining whether a user has an active, non-expired subscription. It's used to control access to subscription-only features.

Now inside the (dashboard)/(routes)/settings the settings page will be created.



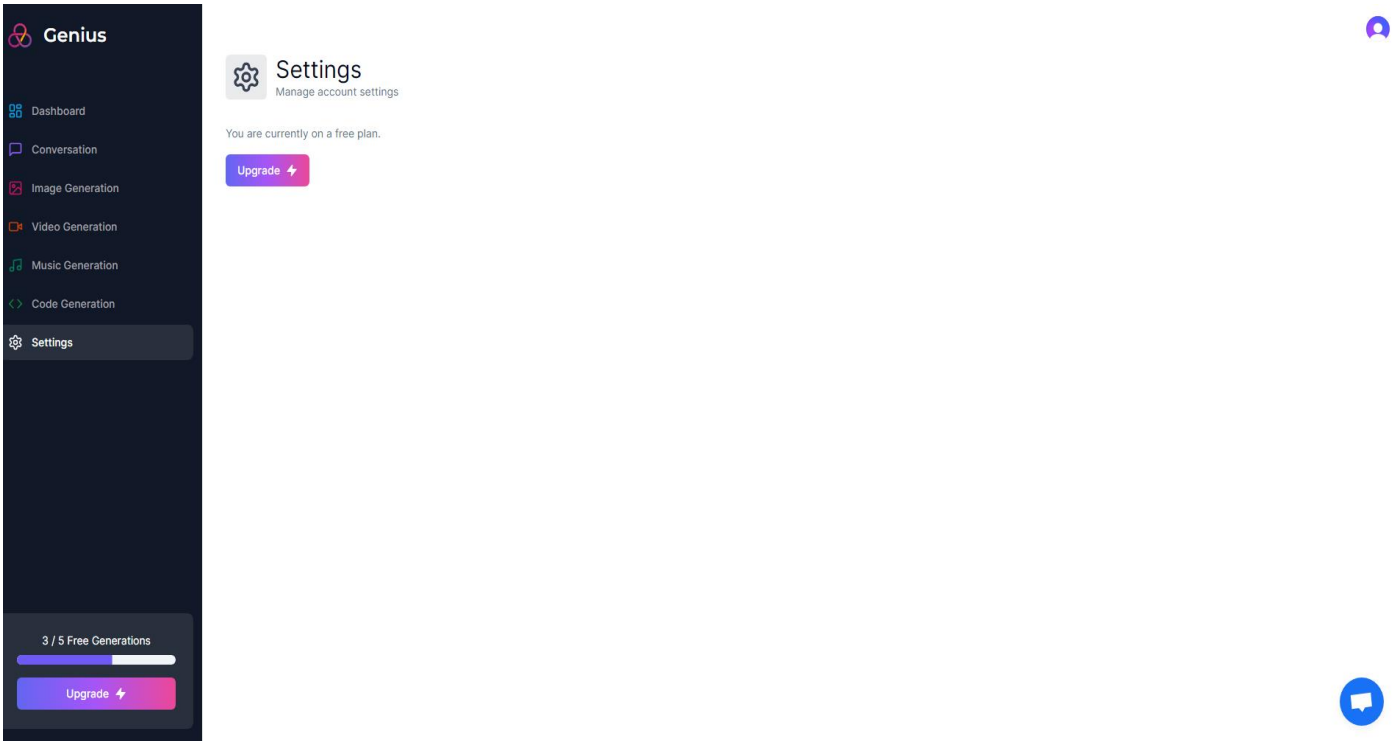
When a user navigates to the settings page, it first checks the user's subscription status using the ‘checkSubscription’ utility. Depending on whether the user is a Pro subscriber or not, it displays the relevant message.

The ‘SubscriptionButton’ is then rendered with the ‘isPro’ status passed as a prop.

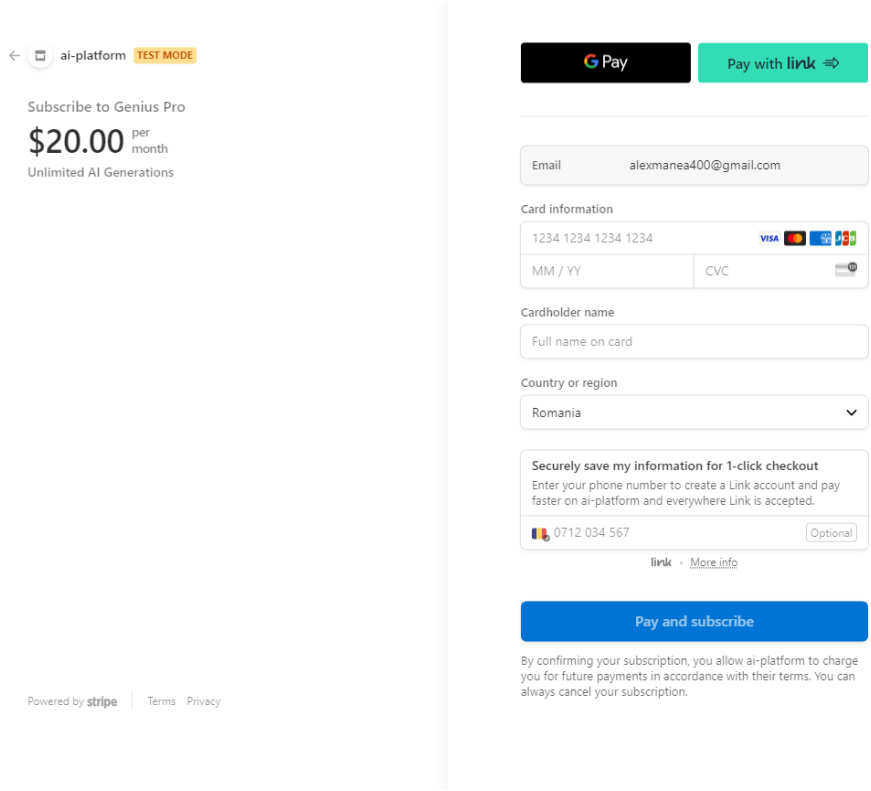
```tsx
"use client"

// Import necessary modules and components
import axios from "axios"
import { Zap } from "lucide-react"

import { Button } from "@/components/ui/button"
import { useState } from "react"
import toast from "react-hot-toast"


// Interface for props passed to the component
interface SubscriptionButtonProps {
    isPro: boolean
}

export const SubscriptionButton = ({
    isPro = false
}: SubscriptionButtonProps) =>{

    const [loading, setLoading] = useState(false)

    // Function to handle button click
    const onClick = async () => {


        try{
            setLoading(true)

            // Request to Stripe API route
            const response = await axios.get("/api/stripe")

            // Redirect to the URL received from the Stripe API response
            window.location.href = response.data.url

        // Handle error message
        } catch (error) {
            toast.error("Something went wrong")
            console.log("BILLING_ERROR", error)
        } finally{
            setLoading(false)
        }

    }

    // Return the button component with dynamic text and action based on the subscription status
    return (
        <Button disabled = {loading} variant={isPro ? "default" : "premium"} onClick={onClick}>
            {isPro ? "Manage Subscription" : "Upgrade"}
            {!isPro && <Zap className="w-4 h-4 ml-2 fill-white" />}
        </Button>

    )


}
```

It displays a button with the text ‘Manage Subscription’ for Pro users and ‘Upgrade’ for free users.
On clicking the button, it triggers the ‘onClick’ function. ‘onClick’ makes a GET request to the /api/stripe endpoint. Then the endpoint determines whether to direct the user to the billing portal or the checkout page based on their subscription status. In case of any errors during this process, a toast message is displayed.

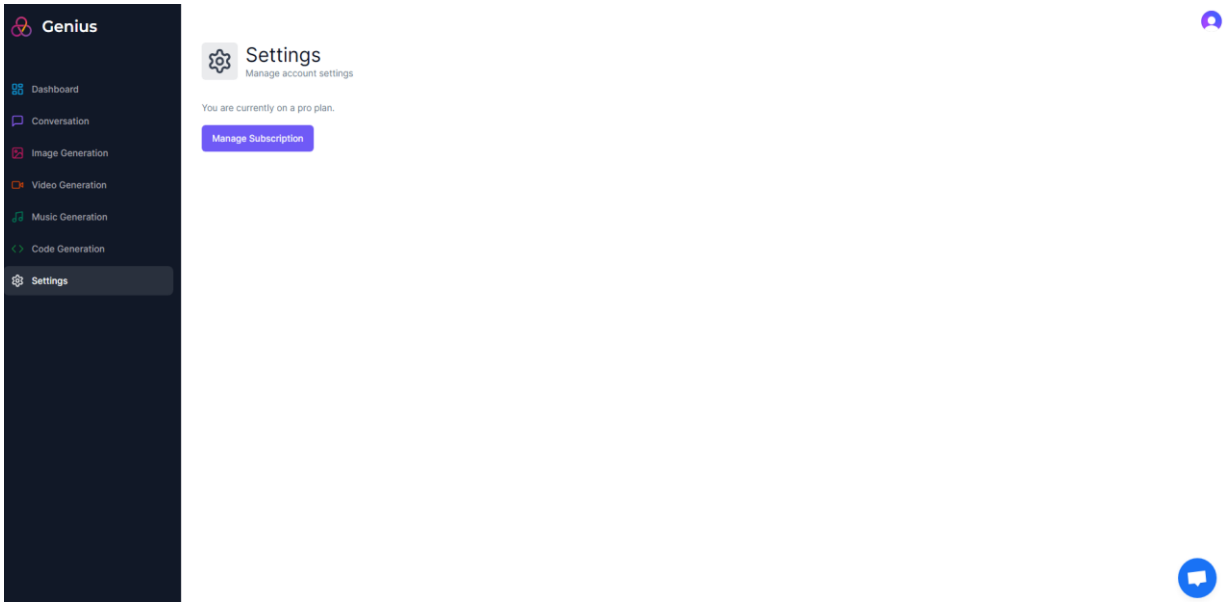
The ‘Settings’ page looks like this for a user on a free plan



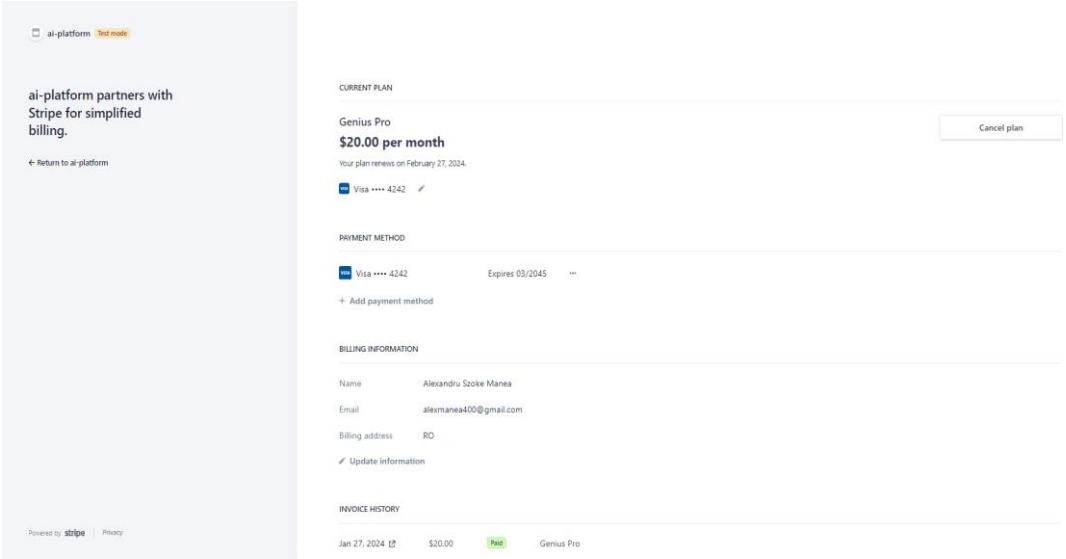
Upon clicking the ‘Upgrade’ button the stripe checkout portal will look like this:



And on the paid plan the ‘Settings’ page will look like this:



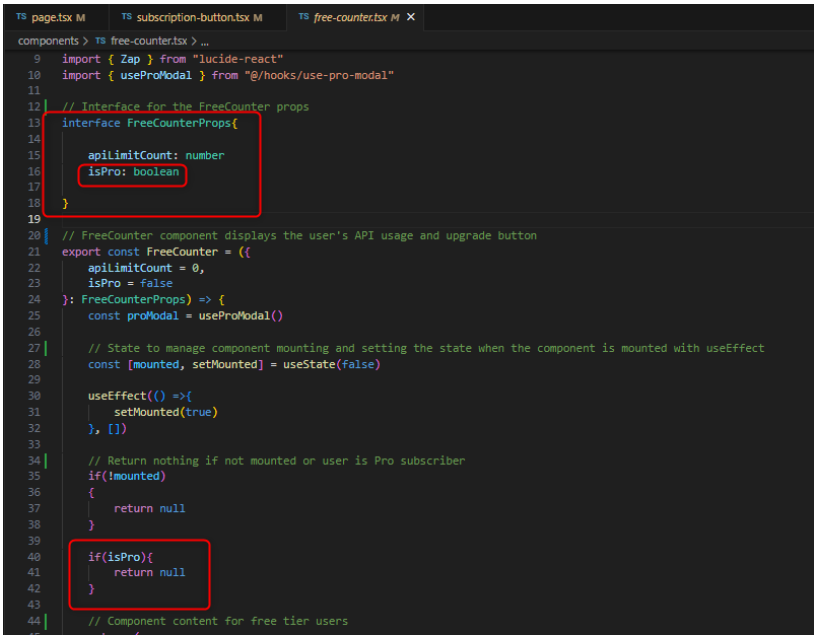
While the billing portal will be like this:



Now, the UI components and API routes need to accommodate the new logic that differentiates between Pro and free users. This means adjusting the UI components and API route logic based on the user's subscription status

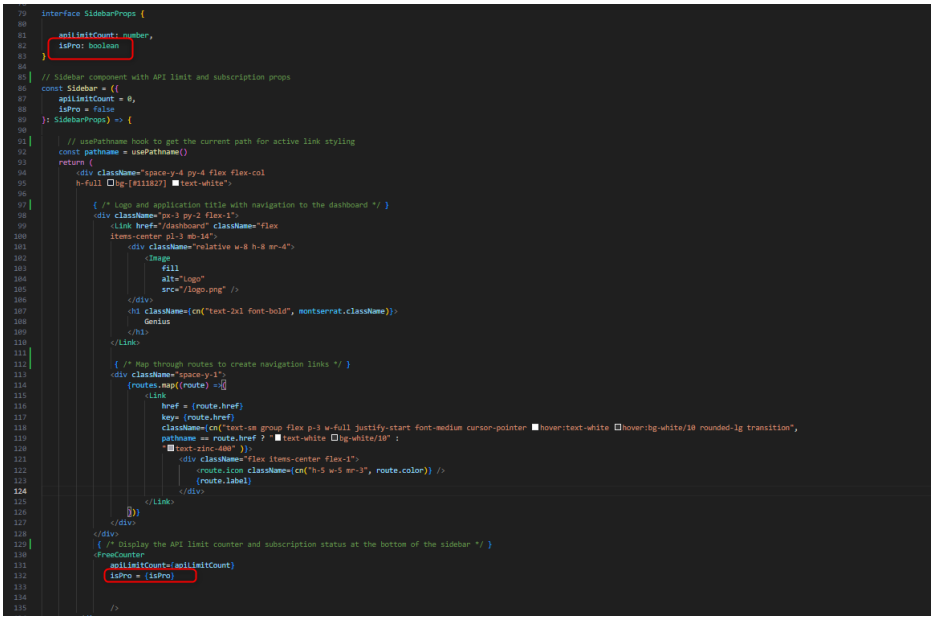
Modifying the Free Counter Component

Adding ‘isPro’ prop means that the ‘FreeCounter’ component accepts an additional prop. This prop indicates whether the user is a Pro subscriber.
If ‘isPro’ is true, the ‘FreeCounter’ component will return null, effectively hiding the free generation counter for Pro users.



Updating the Sidebar Component

Modifying the Sidebar component to accept the ‘isPro’ prop and then passing the ‘isPro’ prop to the ‘FreeCounter’ component within the Sidebar.



Adjusting ‘DashboardLayout’

In ‘DashboardLayout’ , the ‘checkSubscription’ utility is used to determine if the user is a Pro subscriber and then the ‘isPro’ value is provided as a prop to the Sidebar component.

```tsx
// Importing necessary components and utility functions
import Navbar from "@/components/navbar";
import Sidebar from "@/components/sidebar";
import { getApiLimitCount } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

// DashboardLayout component wraps around the main content of the dashboard pages
const DashboardLayout = async({
    children // React children prop to render nested components
} :{
    children: React.ReactNode
}) => {

    // Fetching the current API limit count for the user and checking if he has a pro subscription
    const apiLimitCount = await getApiLimitCount()
    const isPro = await checkSubscription()

    // Rendering the layout with Sidebar, Navbar, and main content
    return (
        <div className="h-full relative">
            {/* Sidebar for navigation, passed with API limit and subscription status */}
            <div className="hidden h-full md:flex
            md:w-72 md:flex-col md:fixed md:inset-y-0
                bg-gray-900">
              <Sidebar apiLimitCount={apiLimitCount} isPro={isPro}/>
            </div>
            {/* Main content area with Navbar and children components */}
              <main className="md:pl-72">
                    <Navbar/>
                    {children}
              </main>


        </div>
    );
}

export default DashboardLayout;
```

Similar to the ‘DashboardLayout’ , the ‘isPro’ propped is passed from the Navbar component to the Mobile sidebar.

```tsx
import { Button } from "@/components/ui/button";
import { UserButton } from "@clerk/nextjs";

import MobileSidebar from "./mobile-sidebar";
import { getApiLimitCount } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

const Navbar = async () => {

    // Fetch API limit and subscription status
    const apiLimitCount = await getApiLimitCount()
    const isPro = await checkSubscription()

    // Render the Navbar with responsive design
    return ( <div className="flex items-center p-4">

                <MobileSidebar isPro={isPro} apiLimitCount={apiLimitCount}/>
                  <div className="flex w-full justify-end">

                    <UserButton afterSignOutUrl="/"/>
                  </div>
                </div>
    );
}

export default Navbar
```

Which accepts the props and then passes it again from there to the Sidebar.

```tsx
"use client"

import { Button } from "@/components/ui/button"

import { Menu } from "lucide-react"
import { Sheet, SheetContent, SheetTrigger } from "@/components/ui/sheet";
import Sidebar from "@/components/sidebar";
import { useEffect, useState } from "react";

interface MobileSidebarProps  {
    apiLimitCount: number
    isPro: boolean
}

const MobileSidebar = ({
    apiLimitCount = 0,
    isPro = false
}: MobileSidebarProps) => {
    const [isMounted, setIsMounted] = useState(false)

    // Conditional rendering based on mounting state
    useEffect(() =>{
        setIsMounted(true)
    },[])

    if(!isMounted)
        return null

    // Sidebar content with custom styling for mobile view
    return (
        <Sheet>
            <SheetTrigger>
                <Button variant ="ghost" size="icon"
                        className="md:hidden">
                            <Menu />
                </Button>
            </SheetTrigger>
            <SheetContent side="left" className="p-0">
                <Sidebar apiLimitCount={apiLimitCount} isPro={isPro}/>

            </SheetContent>
        </Sheet>
    );
}

export default MobileSidebar;
```

Flow of ‘isPro’ Prop

DashboardLayout Component (Full view)
In the full view, the ‘DashboardLayout’ component checks the user's subscription status using ‘checkSubscription’ and it then passes the ‘isPro’ prop to the Sidebar component.

Sidebar Component:
The Sidebar receives the ‘isPro’ prop from ‘DashboardLayout’ and it then passes this prop down to the ‘FreeCounter’ component. (The Sidebar component is used in both desktop and mobile views via ‘MobileSidebar’).

FreeCounter Component:
The ‘FreeCounter’ accepts the ‘isPro’ prop and depending on the value of ‘isPro’, it decides to render the free generation counter or not.

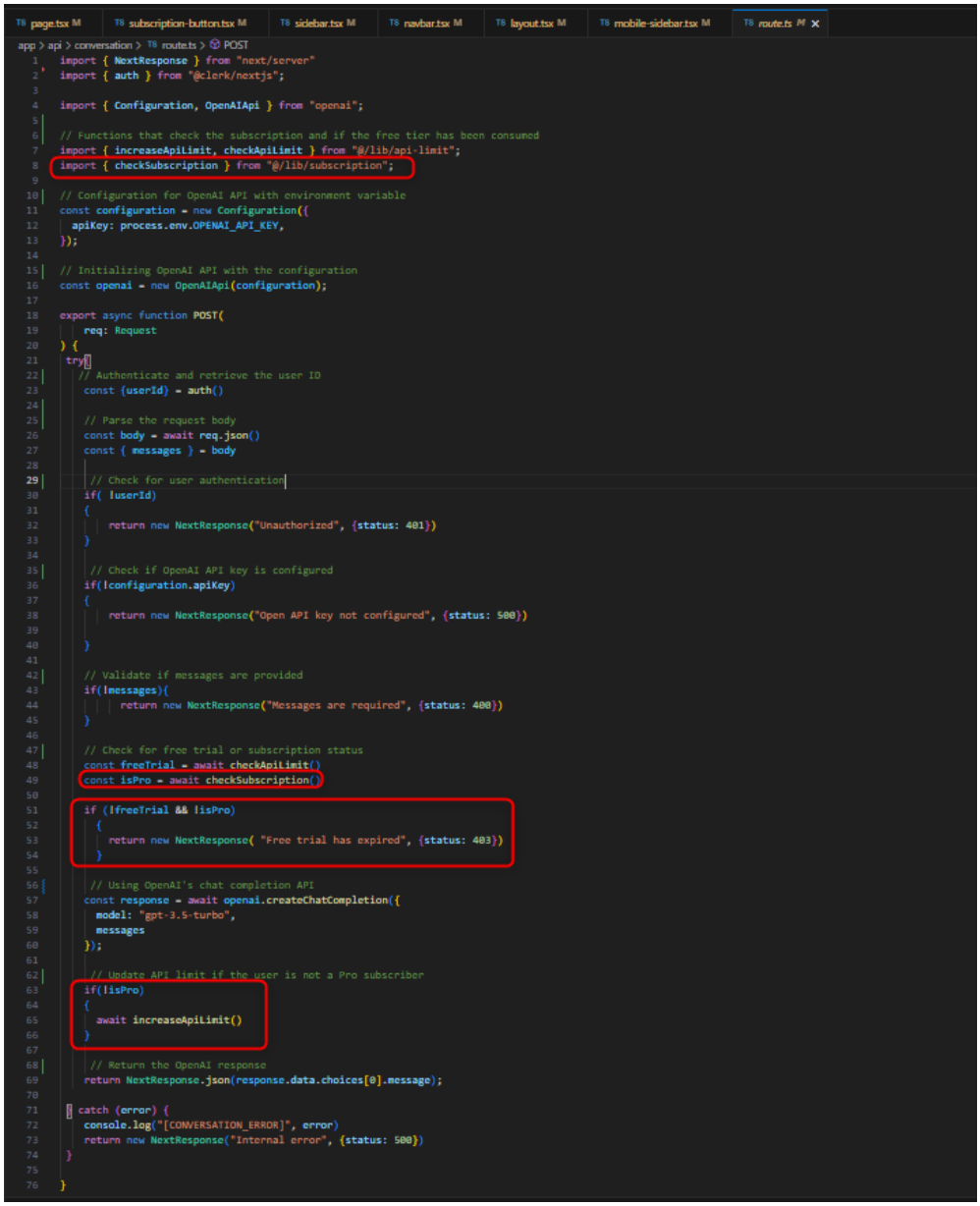
Navbar Component (Mobile View):
In the mobile view, the Navbar component uses ‘checkSubscription’ to determine the user's subscription status and it passes the ‘isPro’ prop to the ‘MobileSidebar’ component.

MobileSidebar Component:
The ‘MobileSidebar’ receives the ‘isPro’ prop from the Navbar and it then uses the standard Sidebar component to render the sidebar content in a mobile-friendly way. The prop is passed to this instance of the Sidebar, which then passes it to the ‘FreeCounter’.

To accommodate the new logic that differentiates between Pro and free users, the API routes for the: Image, Conversation, Video, Code and Music generation will be updated as follows:

- Each API route now includes the ‘checkSubscription’, which determines if the user is a Pro subscriber.
- If the user is a Pro subscriber, certain limitations, such as API usage limits, are bypassed.
- For non-Pro users, the routes continue to check and enforce the free tier usage limits through ‘checkApiLimit’.
- If the user is not on a Pro plan, the API limit is updated using ‘increaseApiLimit’ after each API call.

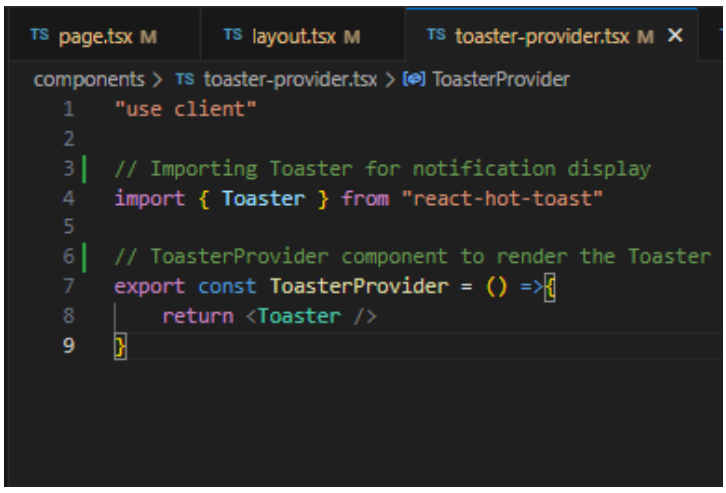
These changes ensure that Pro users have uninterrupted access to services, while free tier users have usage limits.



```
app > api > conversation > route.ts > POST
import { NextResponse } from "next/server"
import { auth } from "@clerk/nextjs";

import { Configuration, OpenAIApi } from "openai";

// Functions that check the subscription and if the free tier has been consumed
import { increaseApiLimit, checkApiLimit } from "@/lib/api-limit";
import { checkSubscription } from "@/lib/subscription";

// Configuration for OpenAI API with environment variable
const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});

// Initializing OpenAI API with the configuration
const openai = new OpenAIApi(configuration);

export async function POST(
    req: Request
) {
 try{
   // Authenticate and retrieve the user ID
    const {userId} = auth()

    // Parse the request body
    const body = await req.json()
    const { messages } = body

    // Check for user authentication
    if( !userId)
    {
        return new NextResponse("Unauthorized", {status: 401})
    }

    // Check if OpenAI API key is configured
    if(!configuration.apiKey)
    {
        return new NextResponse("Open API key not configured", {status: 500})

    }

    // Validate if messages are provided
    if(!messages){
        return new NextResponse("Messages are required", {status: 400})
    }

    // Check for free trial or subscription status
    const freeTrial = await checkApiLimit()
    const isPro = await checkSubscription()

    if (!freeTrial && !isPro)
      {
        return new NextResponse( "Free trial has expired", {status: 403})
      }

    // Using OpenAI's chat completion API
    const response = await openai.createChatCompletion({
      model: "gpt-3.5-turbo",
      messages
    });

    // Update API limit if the user is not a Pro subscriber
    if(!isPro)
    {
      await increaseApiLimit()
    }

    // Return the OpenAI response
    return NextResponse.json(response.data.choices[0].message);

 } catch (error) {
    console.log("[CONVERSATION_ERROR]", error)
    return new NextResponse("Internal error", {status: 500})
 }
}
```

Next step is improving error handling and user feedback in the application. Right now, if something goes wrong, you don't get a notification about it. For this toast will be installed by running ***npm i react-hot-toast.*** This package provides an easy way for displaying toast notifications.

In the components folder, a 'ToasterProvider' component will be created to display the toast notifications.



```
components > toaster-provider.tsx > ToasterProvider
"use client"

// Importing Toaster for notification display
import { Toaster } from "react-hot-toast"

// ToasterProvider component to render the Toaster
export const ToasterProvider = () =>{
    return <Toaster />
}
```

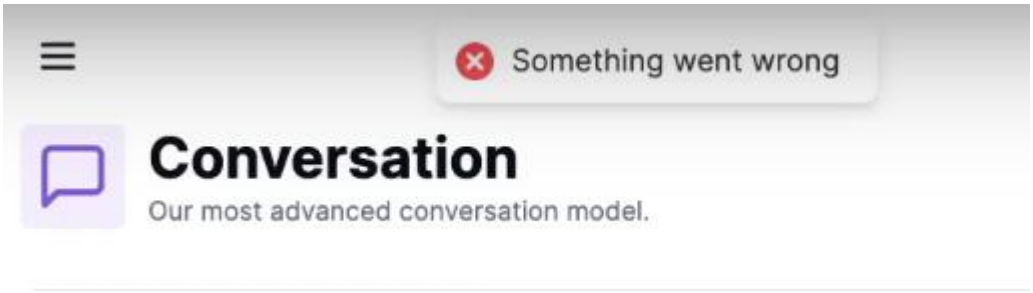
This component will be integrated into the application root layout to make sure that the notifications are available throughout the application.

```tsx
// Importing necessary components and providers
import './globals.css'
import type { Metadata } from 'next'
import { Inter } from 'next/font/google'
import { ClerkProvider } from '@clerk/nextjs'

import { ModalProvider } from '@/components/modal-provider'
import { ToasterProvider } from '@/components/toaster-provider'
import { CrispProvider } from '@/components/crisp-provider'

// Set up font
const inter = Inter({ subsets: ['latin'] })

export const metadata: Metadata = {
  title: 'Genius',
  description: 'AI Platform',
}

// RootLayout component wrapping the entire application
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <ClerkProvider>
      <html lang="en">
        <CrispProvider />
        <body className={inter.className}>
          <ModalProvider />
          <ToasterProvider />
          {children}

          </body>
      </html>
    </ClerkProvider>
  )
}
```

The toast notifications will be implemented in each generation's tool 'page.tsx' , 'pro-modal.tsx' and 'subscription-button.tsx' to provide feedback, particularly for errors.

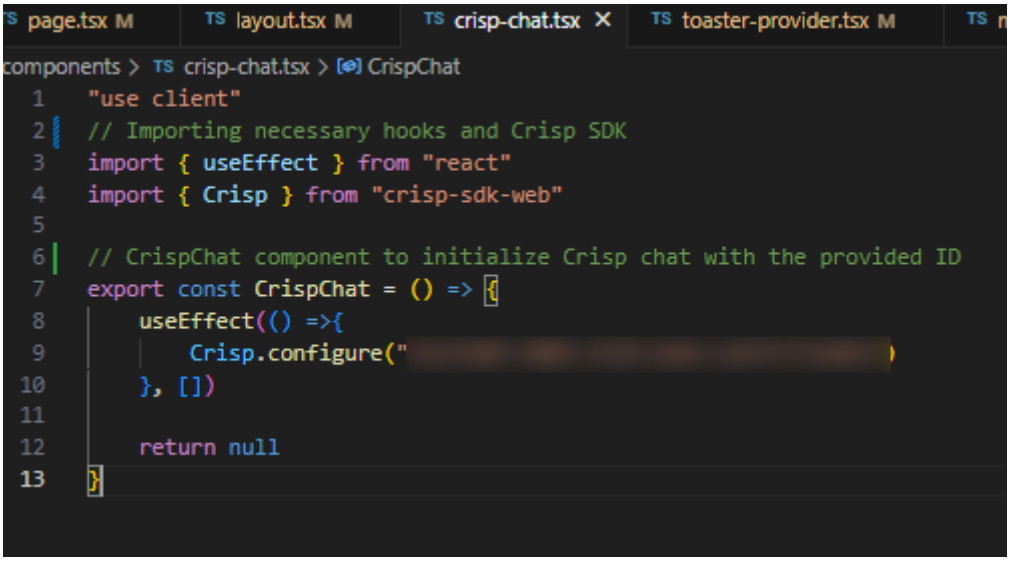
```tsx
const ConversationPage = () => {

    // Hook for modal and routing
    const proModal = useProModal()
    const router = useRouter()

    // State for managing messages in the conversation
    const [messages, setMessages] = useState<ChatCompletionRequestMessage[]>([])

    // Setup form with react-hook-form and zod for validation
    const form = useForm<z.infer<typeof formSchema>>({
        resolver: zodResolver(formSchema),
        defaultValues: {
            prompt: ""
        }
    })

    // Flag for submission state
    const isLoading = form.formState.isSubmitting

    // Handling form submission
    const onSubmit = async ( values:z.infer<typeof formSchema>) =>{
        try{
          const userMessage: ChatCompletionRequestMessage = { role: "user", content: values.prompt };

          const newMessages = [...messages, userMessage]

          const response = await axios.post("/api/conversation", {
           messages: newMessages
          })

          setMessages((current) => [...current, userMessage, response.data])

          form.reset()

        } catch (error: any){
          if(error?.response?.status === 403){
              proModal.onOpen()
          } else {
            toast.error("Something went wrong")
          }
        } finally{
          router.refresh()
        }
    }
```

A toast error looks like this:

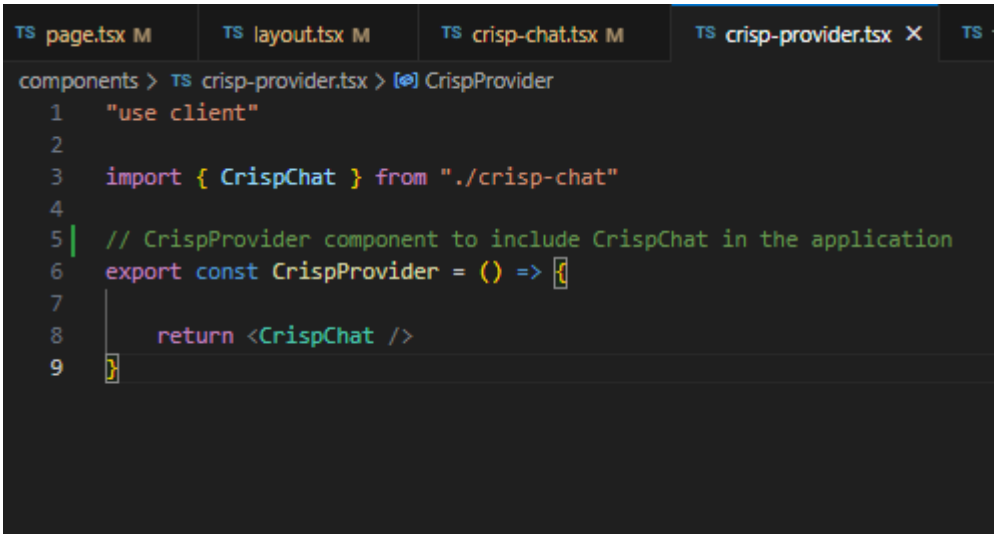


For customer support, Crisp was set up. First an account is needed on their page https://crisp.chat/en/ . After creating an account and following the steps there, you will get a success message and a 'script' tag with an id. That id is needed in order to run crisp. In the terminal run ***npm i crisp-sdk-web*** to install crisp. This package facilitates the integration of Crisp chat functionality into React.

The 'CrispChat' is created in the components folder, and it contains the id that you got during the creation of the account.



Then you a provider is made to wrap and mange the 'CrispChat' integration.

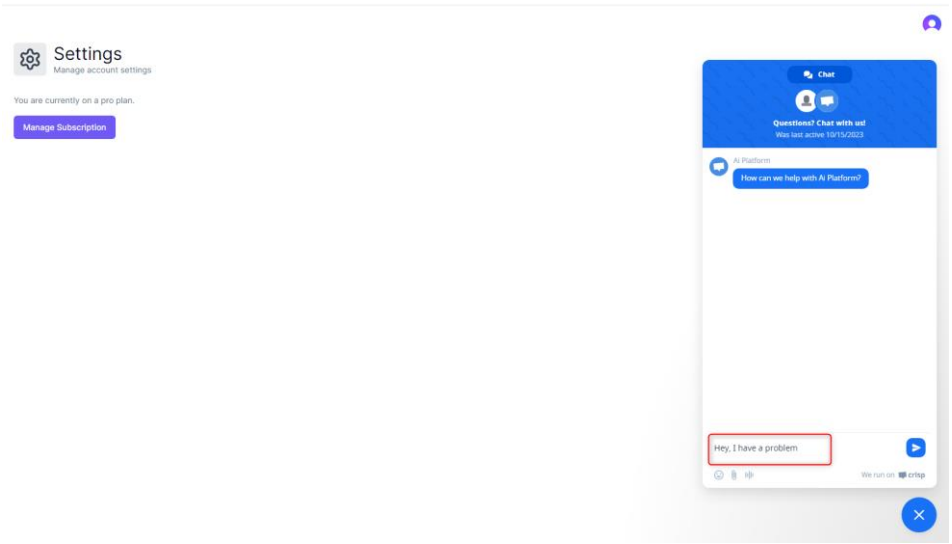


The last step is adding it to the application root layout like the toast notifications.

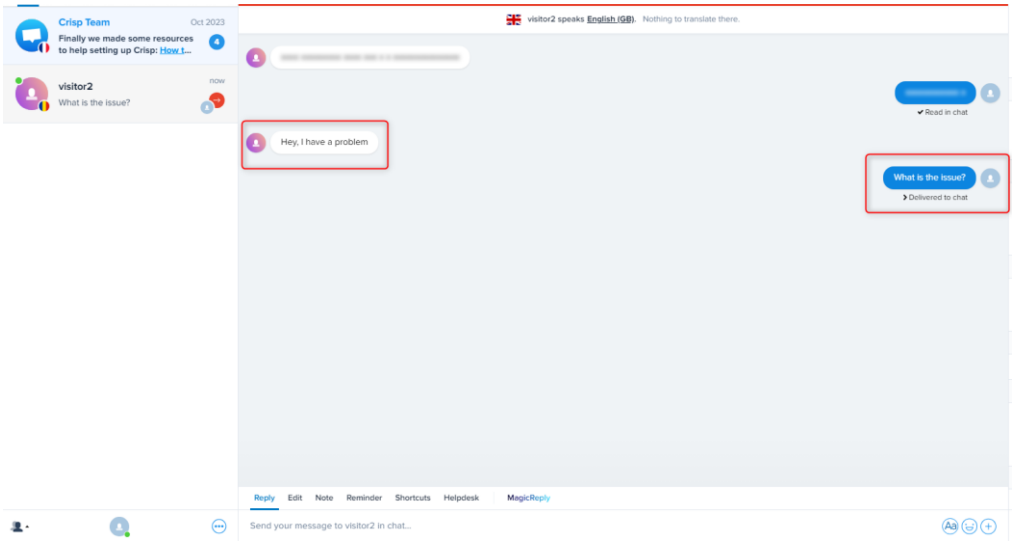
```tsx
app > TS layout.tsx > RootLayout
// Importing necessary components and providers
import './globals.css'
import type { Metadata } from 'next'
import { Inter } from 'next/font/google'
import { ClerkProvider } from '@clerk/nextjs'

import { ModalProvider } from '@/components/modal-provider'
import { ToasterProvider } from '@/components/toaster-provider'
import { CrispProvider } from '@/components/crisp-provider'

// Set up font
const inter = Inter({ subsets: ['latin'] })

export const metadata: Metadata = {
  title: 'Genius',
  description: 'AI Platform',
}

// RootLayout component wrapping the entire application
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <ClerkProvider>
      <html lang="en">
        <CrispProvider />
        <body className={inter.className}>
          <ModalProvider />
          <ToasterProvider />
          {children}

        </body>
      </html>
    </ClerkProvider>
  )
}
```

Crisp integration allows for real-time communication with users, enhancing customer support.

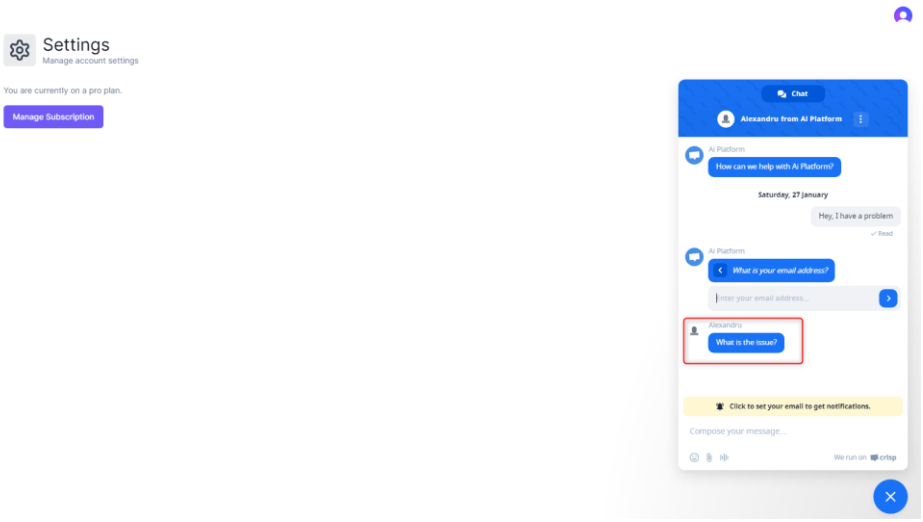
Now users can send messages, and you can respond to them from the crisp dashboard. This is demonstrated below



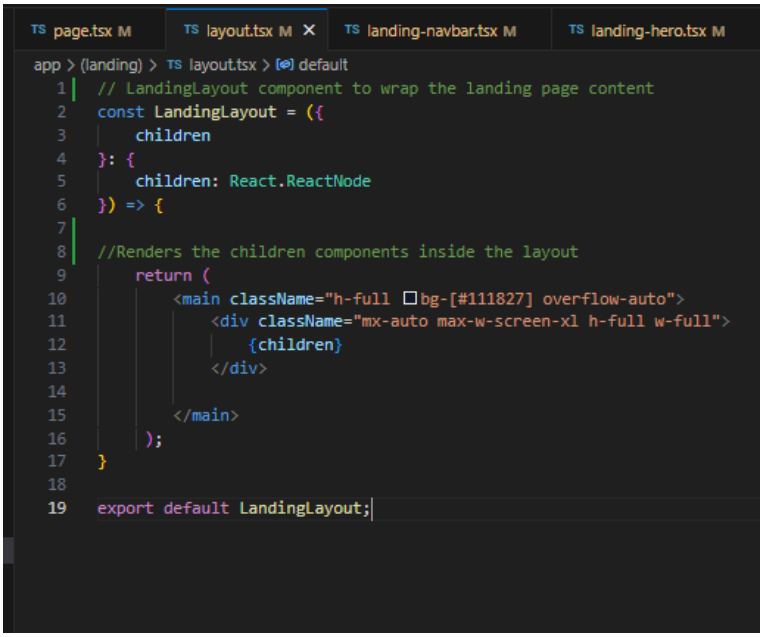
From the crisp dashboard, you can reply to the user's message



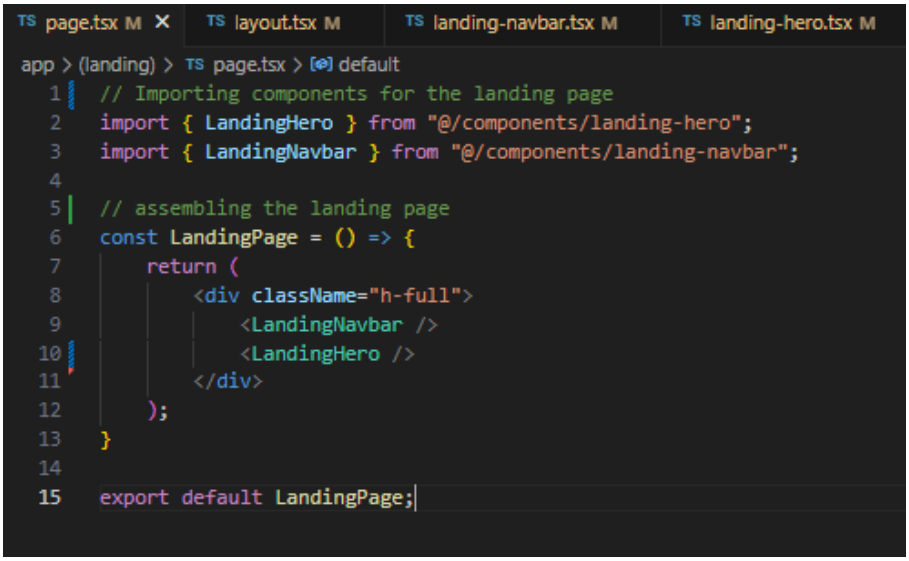
Then the user will see your message in real time.



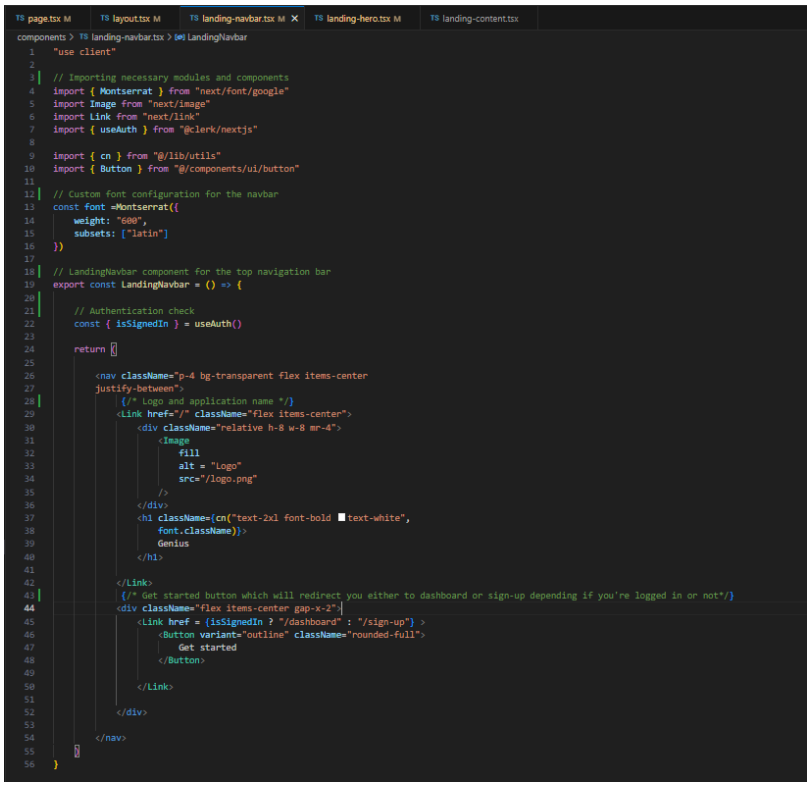
The last part of this project is the landing page which serves as an introduction for the users. First, the landing layout forms located in (dashboard)/(routes)/(landing)/layout.tsx serves as the backbone of the landing page. It's designed to provide a consistent for the page.



Following the layout, the landing page itself, created in ‘page.tsx’, acts as a hub, bringing together all the individual components. This page integrates the ‘LandingNavbar’ and ‘LandingHero’ components, providing a seamless user experience.



As for the individual components, the ‘LandingNavbar’ is crucial for navigation. It not only displays the logo and name but also alters its functionality based on the user’s authentication status. By offering a dynamic link that changes between 'Get Started' for new users and a direct link to the dashboard for signed-in users, it enhances the user experience.



```tsx
"use client"

// Importing necessary modules and components
import { Montserrat } from "next/font/google"
import Image from "next/image"
import Link from "next/link"
import { useAuth } from "@clerk/nextjs"

import { cn } from "@/lib/utils"
import { Button } from "@/components/ui/button"

// Custom font configuration for the navbar
const font =Montserrat({
    weight: "600",
    subsets: ["latin"]
})

// LandingNavbar component for the top navigation bar
export const LandingNavbar = () => {

    // Authentication check
    const { isSignedIn } = useAuth()

    return (

        <nav className="p-4 bg-transparent flex items-center
        justify-between">
            {/* Logo and application name */}
            <Link href="/" className="flex items-center">
                <div className="relative h-8 w-8 mr-4">
                    <Image
                        fill
                        alt = "Logo"
                        src="/logo.png"
                    />
                </div>
                <h1 className={cn("text-2xl font-bold text-white",
                    font.className)}>
                    Genius
                </h1>

            </Link>
            {/* Get started button which will redirect you either to dashboard or sign-up depending if you're logged in or not*/}
            <div className="flex items-center gap-x-2">
                <Link href = {isSignedIn ? "/dashboard" : "/sign-up"} >
                    <Button variant="outline" className="rounded-full">
                        Get started
                    </Button>

                </Link>

            </div>

        </nav>
    )
}
```

The hero section is where the main features of the 'Genius' platform are highlighted. Using a typewriter effect, it dynamically displays the different AI tools available. Also, it encourages new users to explore the platform further, while existing users are redirected to their dashboard.

```tsx
"use client"

// Importing necessary modules and components
import { useAuth } from "@clerk/nextjs"
import TypeWriterComponent from "typewriter-effect"
import Link from "next/link"
import { Button } from "./ui/button"

// LandingHero component for the main showcase area
export const LandingHero = () => {

    const {isSignedIn} = useAuth()
    return ( <div className="text-white font-bold py-36 text-center space-y-5">
                <div className="text-4xl sm:text-5xl md:text-6xl
                lg:text-7xl space-y-5 font-extrabold">
                    <h1>
                        The Best AI tool for
                    </h1>
                    {/* Animated text  */}
                    <div className="text-transparent bg-clip-text
                      bg-gradient-to-r from-purple-400 to-pink-600">
                            <TypeWriterComponent
                                options = {{
                                    strings: [

                                        "Conversation.",
                                        "Photo Generation.",
                                        "Music Generation.",
                                        "Code Generation.",
                                        "Video Generation.",

                                    ],
                                    autoStart: true,
                                    loop: true
                                }}

                            />
                    </div>

                </div>
                <div className="text-sm md:text-xl font-medium text-zinc-400">
                    Create content using AI 10x faster

                </div>
                <div>
                    {/* Button that will take you to the dashboard or sign-up */}
                    <Link href = {isSignedIn ? "/dashboard" : "/sign-up"}>
                        <Button variant = "premium" className="md:text-lg p-4 md:p-6
                          rounded-full font-semibold">
                            Start Generating for free
                        </Button>
                    </Link>
                </div>
                <div className="text-zinc-400 text-xs md:text-sm font-normal">
                    No credit card required
                </div>
            </div>
    )
}
```

All assembled, the landing page will look like this: