Prerequisites:

Please make sure to go through the setup document provided to you on github so that your system is ready for the workshop. The following prerequisites should be set up for use.

- Node JS installed
- NPM/NPX/PNPM installed
- Started up a basic project with create t3-app@latest
- Discord API Keys setup and ready to use
- **As a reminder please do not mix using different package managers for the same project as it may cause dependency issues**
- **Add the environment variables AUTH_DISCORD_ID, AUTH_DISCORD_SECRET to the .env file.**

First Steps:

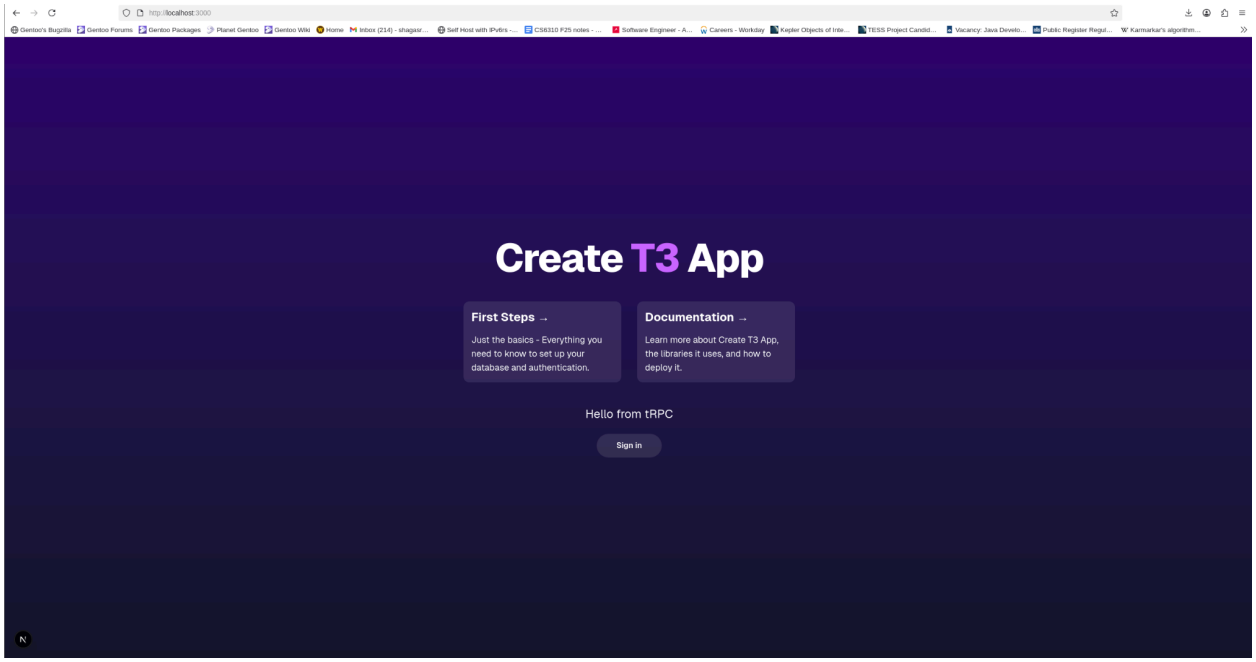
Welcome to the Web Community Workshop where we will be making a full stack application Notes app with next js, next auth login with discord, webhook events, trpc and more. Currently you should be all set up with the starter project. You can use the following commands to run the starter app that was just built for you.

npm run dev

(or)

pnpm dev

Please choose command as per the package manager you have installed and you should see something like the image on next page



Building the Base Application:

We first start by creating the components **app/_components/HomeComponent.tsx**, use the following code snippet.

This will create the base UI for our notes application.

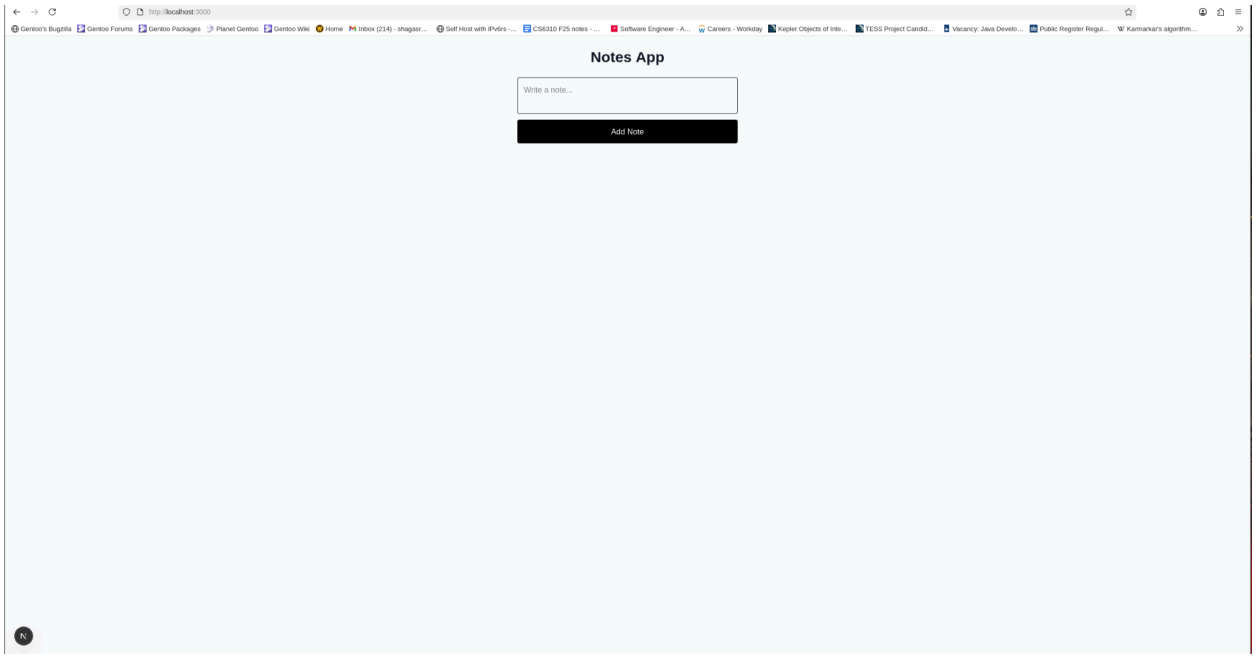
```
"use client";

import { useState } from "react";

export default function HomeComponent() {
  const [note, setNote] = useState("");

  const handleAddNote = (e: React.FormEvent) => {
    e.preventDefault();
    console.log("New note:", note);
    setNote("");
  };

  return (
```

```
      <main className="flex min-h-screen flex-col items-center justify-start p-6">
        <h1 className="text-3xl font-bold mb-6">Notes App</h1>

        <form
          onSubmit={handleAddNote}
          className="w-full max-w-md flex flex-col gap-3 mb-6"
        >
          <textarea
            className="rounded border p-3 outline-none resize-none"
            placeholder="Write a note..."
            value={note}
            onChange={(e) => setNote(e.target.value)}
          />
          <button
            type="submit"
            className="rounded bg-black p-3 text-white hover:bg-gray-800 transition"
          >
            Add Note
          </button>
        </form>
      </main>
    );
  }
```

Now we modify the page.tsx to the following which will set it to use this component to render instead of the Welcome code we coded earlier.

```
import HomeComponent from "src/app/_components/HomeComponent";

export default function Home() {
  return <HomeComponent />;
}
```

You should now be able to see your UI similar to the following.



Lets now add Note Storage and Display:

Open **src/app/_components/HomeComponent.tsx** and make the following changes:

1. Add another state variable right below the first **useState**:

   const [notes, setNotes] = useState<string[]>([]);

2. Update the **handleAddNote** function and remove the console.log line and replace it with:

   e.preventDefault();
   if (!note.trim()) return;
   setNotes((prev) => [note.trim(), ...prev]);
   setNote("");

3. Add a delete function below **handleAddNote**:

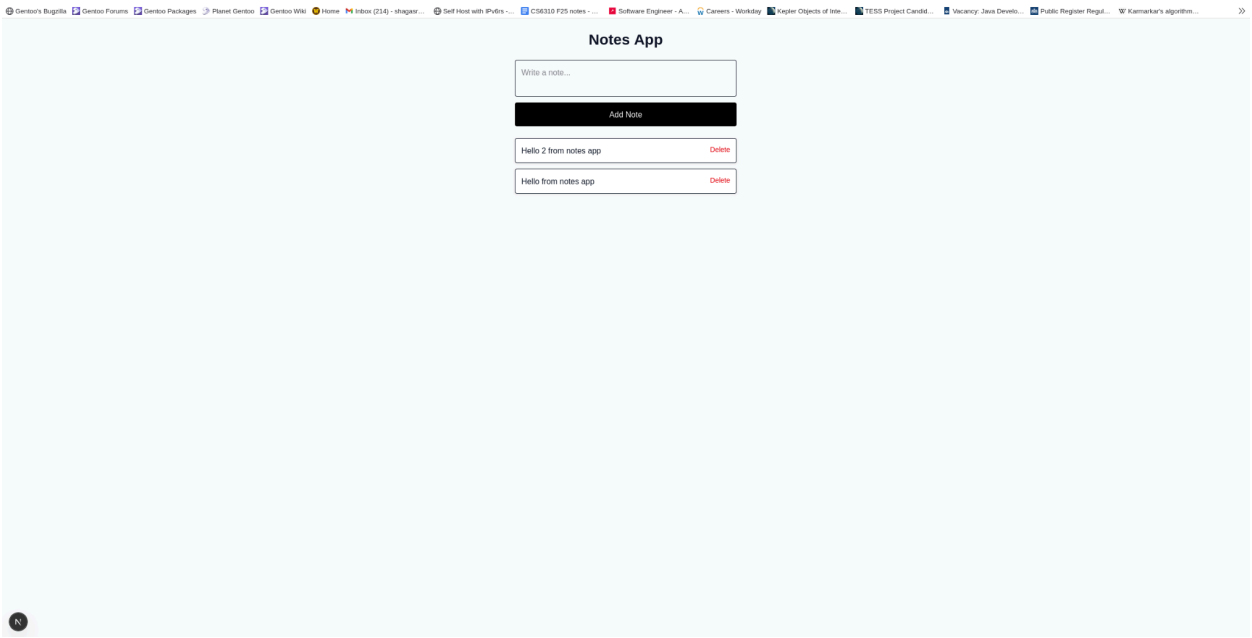
```
const handleDelete = (index: number) => {
        setNotes((prev) => prev.filter((_, i) => i !== index));
};
```

4. Add a section after the form to display all notes:

```
<section className="w-full max-w-md flex flex-col gap-3">
 {notes.length === 0 ? (
  <p className="text-gray-500 italic text-center">
   No notes yet. Write one above.
  </p>
 ) : (
  notes.map((item, index) => (
   <div
    key={index}
          className="flex items-start justify-between rounded
border bg-white p-3 shadow-sm"
   >
    <p className="whitespace-pre-wrap">{item}</p>
    <button
     onClick={() => handleDelete(index)}
     className="ml-4 text-sm text-red-600 hover:text-red-800"
    >
     Delete
    </button>
   </div>
  ))
 )}
</section>
```

Now you can re run **npm run dev or pnpm dev** if you have stopped it earlier if not you can see the changes hot reloaded. From here you can

go around and add the notes using the button and you will be able to see them in the section below. You should have something similar to the image below.



If you try refreshing your page you will notice all the notes are gone. This is because currently we are not utilizing the database and are storing all the notes in the application memory. Lets now work with prisma files and try to make the notes persistent by saving it to the db we created earlier using **Trpc**.

1. Add the following model to your prisma schema. You can find it under the prisma directory outside of src scope. **‘prisma/schema.prisma’**

```
model Note {
  id        String   @id @default(cuid())
  content   String
  title          String
  createdAt DateTime @default(now())
  updatedOn DateTime
}
```

2. Apply the changes to your db using the **‘migrate’** functionality of prisma.

npx prisma migrate dev --name notesAdd
(or)
pnpm dlx prisma migrate dev --name notesAdd

3. Once the db is synced you should see a new folder appear named **‘migrations’** which upon inspection has the migration with the name you specified earlier. By default migrations are named by the unix time index but by adding the name to a migration it becomes easy to identify what changes were done. Also if you see the generated sql file you should see it is creating a table.

Now, let's write the CRUD operations which will be responsible for all the database read/write/delete/etc operations. Create a new file **‘src/server/api/routers/note.ts’** and use the following snippet.

```
import { z } from "zod";
import { createTRPCRouter, publicProcedure } from "src/server/api/trpc";

export const noteRouter = createTRPCRouter({
 findAllNotes: publicProcedure.query(async ({ ctx }) => {
  return ctx.db.note.findMany({ orderBy: { createdAt: "desc" } });
 }),

 addNote: publicProcedure
  .input(z.string().min(1, "Note cannot be empty"))
  .mutation(async ({ ctx, input }) => {
   return ctx.db.note.create({ data: { content: input } });
  }),

 deleteNote: publicProcedure
  .input(z.string())
  .mutation(async ({ ctx, input }) => {
   await ctx.db.note.delete({ where: { id: input } });
   return true;
  }),
```

```
});
```

Now we need to register the router in the **'root.ts'** file

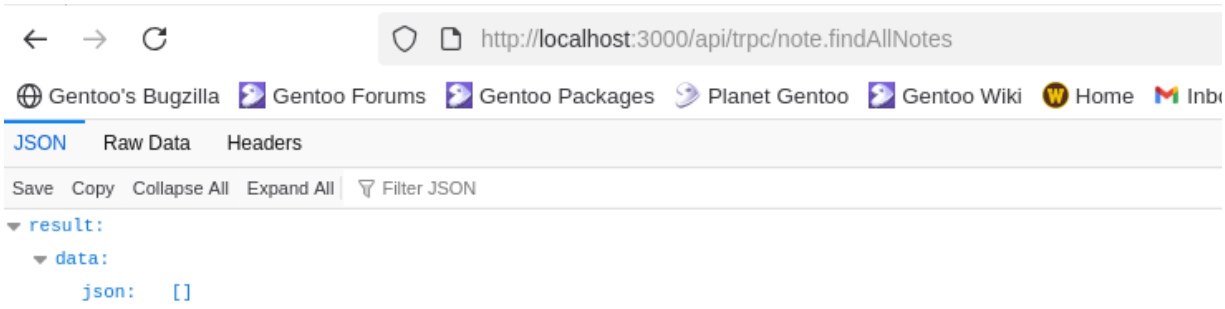
1. Add the following import

   ```
   import { noteRouter } from "src/server/api/routers/note";
   ```

2. Register the router in AppRouter

   ```
   export const appRouter = createTRPCRouter({
     post: postRouter,
     note: noteRouter,
   });
   ```

You can now visit **'http://localhost:3000/api/trpc/note.findAllNotes'** on your browser or postman and you will see an empty list which means the api endpoints are functional.



Now that our trpc setup is complete, lets update our frontend to communicate with the backend.

1. Update the page.tsx to wrap the home component in the trpc react provider

```
"use client";

import HomeComponent from
"src/app/_components/HomeComponent";
import { TRPCReactProvider } from "src/trpc/react";

export default function Page() {
  return (
    <TRPCReactProvider>
      <HomeComponent />
    </TRPCReactProvider>
  );
}
```

2. Update the Home Component to shift from the variables we created and instead use the new trpc api calls.
    - Import tRPC client
      import { api } from "src/trpc/react";

    - Replace local variable storage with api calls
      Replace
      ```
      const [notes, setNotes] = useState<string[]>([]);
      ```
      With
      ```
      const utils = api.useUtils();
      const { data: notes, isLoading, isError } =
      api.note.findAllNotes.useQuery();
      ```

    - Replace the add note logic with mutate from api

      Replace
      ```
      setNotes((prev) => [note.trim(), ...prev]);
      ```
      With

```
addNote.mutate(note.trim());
```

- Add these TRPC mutation hooks near the top of the component:

```
const addNote = api.note.addNote.useMutation({
  onSuccess: () => {
    setNote("");
    void utils.note.findAllNotes.invalidate();
  },
});

const deleteNote = api.note.deleteNote.useMutation({
  onSuccess: () => {
    void utils.note.findAllNotes.invalidate();
  },
});
```

- Replace delete logic

  Replace

```
const handleDelete = (index: number) => {
  setNotes((prev) => prev.filter((_, i) => i !== index));
};
```

  With

```
const handleDelete = (id: string) => {
  deleteNote.mutate(id);
};
```

- Update rendering for the notes by updating the section part of the code

```
<section className="flex w-full max-w-md flex-col gap-3">
```

```
 {isLoading ? (
   <p className="text-center text-gray-500 italic">Loading
notes...</p>
 ) : isError ? (
   <p className="text-center text-red-500 italic">Failed to load
notes.</p>
 ) : !notes || notes.length === 0 ? (
   <p className="text-center text-gray-500 italic">
     No notes yet. Write one above.
   </p>
 ) : (
   notes.map((n) => (
     <div
       key={n.id}
       className="flex items-start justify-between rounded border
bg-white p-3 shadow-sm"
     >
       <p className="whitespace-pre-wrap">{n.content}</p>
       <button
         onClick={() => handleDelete(n.id)}
         className="ml-4 text-sm text-red-600 hover:text-red-800"
       >
         Delete
       </button>
     </div>
   ))
 )}
</section>
```

Now, you can see the changes you made in the browser. You will be able to add and delete notes and the changes are written to the database to be persistent.

As a next exercise lets add the update functionality. For adding this functionality as all the other process, we need to add a mutation and trpc endpoint that handles updates, the old values need to be held before being committed and the rendering needs to be updated to include this new field.

1. Add the new updateNote endpoint to note.ts

```
updateNote: publicProcedure
  .input(z.object({ id: z.string(), content: z.string().min(1) }))
  .mutation(async ({ ctx, input }) => {
    return ctx.db.note.update({
      where: { id: input.id },
      data: { content: input.content },
    });
  }),
```

2. Add **updateNote** mutation below your existing mutations (addNote and deleteNote), add the following snippet.

```
const updateNote = api.note.updateNote.useMutation({
  onSuccess: () => {
    void utils.note.findAllNotes.invalidate();
  },
});
```

3. Add local state for editing and tracking the old value

```
const [editingId, setEditingId] = useState<string | null>(null);
const [editValue, setEditValue] = useState("");
```

4. Replace your current note loop by adding the new edit button which has save and cancel buttons in the new edit box.

```
<section className="flex w-full max-w-md flex-col gap-3">
  {isLoading ? (
    <p className="text-center text-gray-500 italic">Loading notes...</p>
  ) : isError ? (
```

```
    <p className="text-center text-red-600">
      Could not load notes. Try again.
    </p>
  ) : !notes || notes.length === 0 ? (
    <p className="text-center text-gray-500 italic">
      No notes yet. Write one above.
    </p>
  ) : (
    notes.map((n) => (
      <div
        key={n.id}
        className="flex flex-col rounded border bg-white p-3 shadow-sm"
      >
        {editingId === n.id ? (
          <>
            <textarea
              className="mb-2 resize-none rounded border p-2 outline-none"
              value={editValue}
              onChange={(e) => setEditValue(e.target.value)}
            />
            <div className="flex gap-2">
              <button
                onClick={() => {
                  if (!editValue.trim()) return;
                  updateNote.mutate({
                    id: n.id,
                    content: editValue.trim(),
                  });
                  setEditingId(null);
                }}
                className="rounded bg-green-600 px-3 py-1 text-white transition
hover:bg-green-700"
                disabled={updateNote.isPending}
              >
                Save
              </button>
              <button
                onClick={() => setEditingId(null)}
                className="rounded bg-gray-300 px-3 py-1 text-black transition
hover:bg-gray-400"
```

```
            >
              Cancel
            </button>
          </div>
        </>
      ) : (
        <div className="flex items-start justify-between">
          <p className="whitespace-pre-wrap">{n.content}</p>
          <div className="flex gap-2">
            <button
              onClick={() => {
                setEditingId(n.id);
                setEditValue(n.content);
              }}
              className="text-sm text-blue-600 transition hover:text-blue-800"
            >
              Edit
            </button>
            <button
              onClick={() => handleDelete(n.id)}
              className="text-sm text-red-600 transition hover:text-red-800"
              disabled={deleteNote.isPending}
            >
              Delete
            </button>
          </div>
        </div>
      )}
    </div>
  ))
 )}
</section>
```

Now when you try it, you should also be able to edit each note and all changes are persistent. From here the next step would be to add and update the notes router which handles api routes to include the usage of other fields we created

```
findAllNotes: publicProcedure.query(async ({ ctx }) => {
  const foundNotes = await ctx.db.note.findMany({
```

```
    orderBy: { createdOn: "desc" },
  });
  return foundNotes ?? null;
}),

addNote: publicProcedure
  .input(z.string().min(1, "Note cannot be empty"))
  .mutation(async ({ ctx, input }) => {
    const note = await ctx.db.note.create({
      data: {
        content: input,
        title: "Some title",
        updatedOn: new Date(),
      },
    });
    return note;
  }),

updateNote: publicProcedure
  .input(z.object({ id: z.string(), content: z.string().min(1) }))
  .mutation(async ({ ctx, input }) => {
    const note = await ctx.db.note.update({
      where: { id: input.id },
      data: { content: input.content },
    });
    return !!note;
  }),

deleteNote: publicProcedure
  .input(z.string())
  .mutation(async ({ ctx, input }) => {
    const note = await ctx.db.note.delete({
      where: { id: input },
    });
    return !!note;
  }),
```

With this we are ready to integrate the discord login with the front end. Before we proceed there are two ways to approach making the login page. Either we can do it all in the page.tsx and layout and wrap the home component in it which works but isnt the best approach. The best practices for any application lies in its modularity and its ability to reuse parts of the code in different places. So, now we are going to create a new component for login then wrap the home component in it. The login component will be acting as the bouncer not allowing any person who isn't already logged in. We will also be using sessions to temporarily save the logged in user details so that they don't need to keep logging in every time they refresh or reload the application. To do this we will also be making use of next-auth react **useSessions().**

Create a new component named **'SignInComponent.tsx'** in the _components directory where we previously created the **HomeComponent.**

```
"use client";

import { signIn } from "next-auth/react";

export default function SignInComponent() {
  const handleSignIn = async () => {
    await signIn();
  };

  return (
    <button
      onClick={handleSignIn}
          className="rounded-lg bg-blue-500 px-6 py-2 text-white transition duration-200 hover:bg-blue-600 focus:ring-2 focus:ring-blue-500 focus:ring-offset-2 focus:outline-none"
    >
      Sign In
    </button>
  );
}
```

We also need to make a few changes to the **layout.tsx** as we need to handle sessions and to do that we will be wrapping children in a session wrapper. You can replace the existing code with the snippet below.

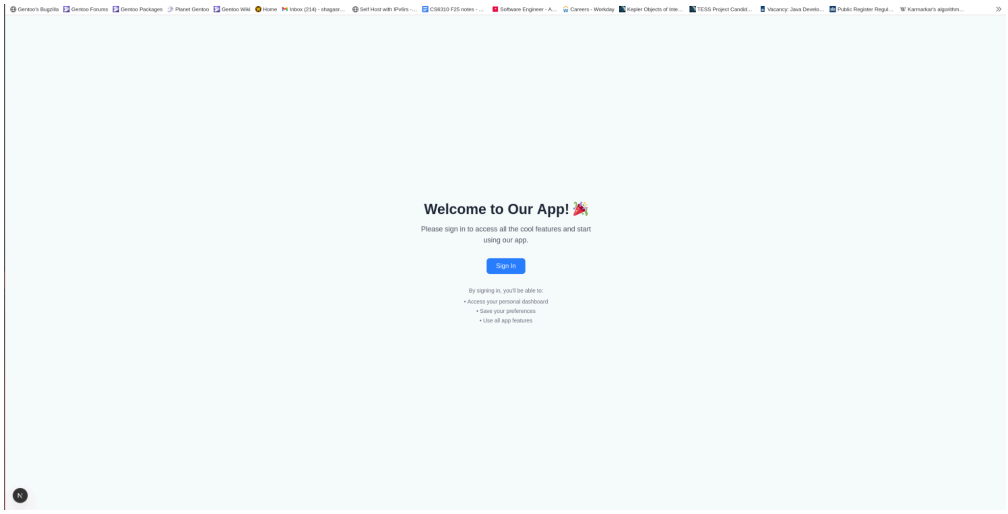
```
import "src/styles/globals.css";

import { type Metadata } from "next";
import { Geist } from "next/font/google";

import { TRPCReactProvider } from "src/trpc/react";
import { SessionProvider } from "next-auth/react";

import { auth } from "src/server/auth/index";

export const metadata: Metadata = {
  title: "Create T3 App",
  description: "Generated by create-t3-app",
  icons: [{ rel: "icon", url: "/favicon.ico" }],
};

const geist = Geist({
  subsets: ["latin"],
  variable: "--font-geist-sans",
});

export default async function RootLayout({
  children,
}: Readonly<{ children: React.ReactNode }>) {
  const session = await auth();
  return (
    <html lang="en" className={`${geist.variable}`}>
      <body>
        <TRPCReactProvider>
          <SessionProvider session={session}>{children}</SessionProvider>
        </TRPCReactProvider>
      </body>
    </html>
  );
}
```

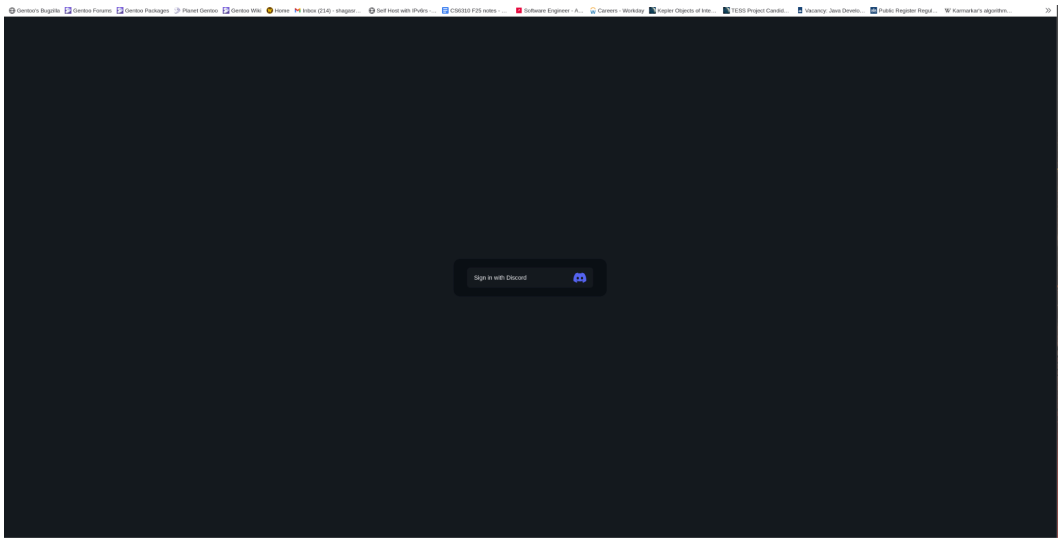
This will make it such that in order to access any child components they need to be logged in first. From here lets add the Signin component to **page.tsx** and style it.

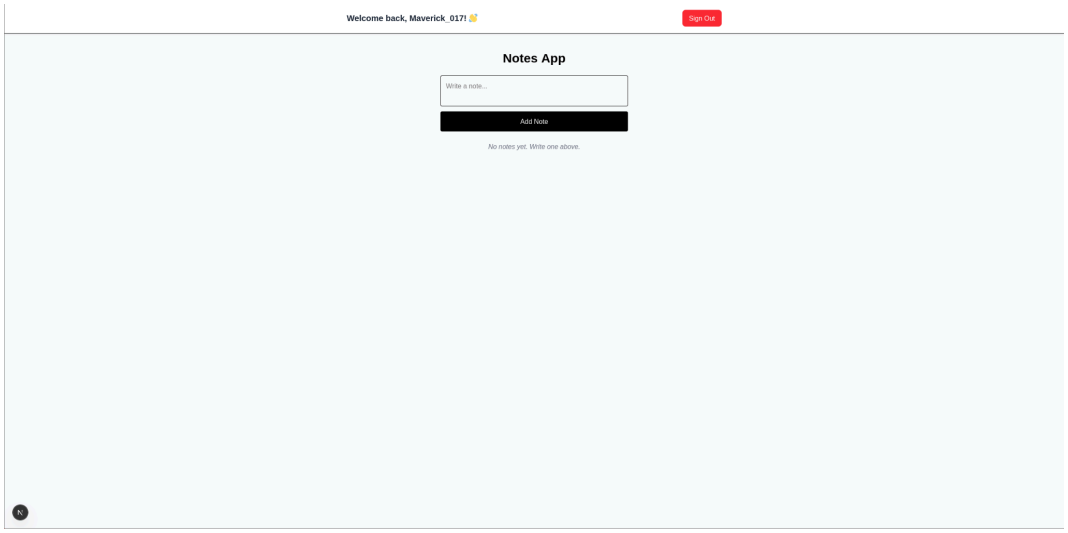
```
"use client";

import { signOut, useSession } from "next-auth/react"; // Tools for checking
if someone is signed in
import HomeComponent from "src/app/_components/HomeComponent"; //
The main app that signed-in users see
import SignInComponent from "src/app/_components/SignInComponent";
// Our simple sign-in button

export default function Home() {
  const { data: session } = useSession();

  const handleSignOut = async () => {
    await signOut(); // This logs the person out and clears their "ticket"
  };

  return (
    <div className="min-h-screen bg-gray-50">
      {session ? (
        <div>
          <div className="border-b bg-white p-4 shadow-sm">
            <div className="mx-auto flex max-w-4xl items-center
justify-between">
              <h1 className="text-xl font-semibold text-gray-800">
                Welcome back, {session.user?.name ?? "Friend"}! 👋
              </h1>

              <button
                onClick={handleSignOut}
                className="rounded-lg bg-red-500 px-4 py-2 text-white transition
duration-200 hover:bg-red-600 focus:ring-2 focus:ring-red-500
focus:ring-offset-2 focus:outline-none"
              >
                Sign Out
              </button>
            </div>
```

```
          </div>

          <div className="mx-auto max-w-4xl p-4">
            <HomeComponent />
          </div>
        </div>
      ) : (
        <div className="flex min-h-screen items-center justify-center">
          <div className="text-center">
            <h1 className="mb-4 text-4xl font-bold text-gray-800">
              Welcome to Our App! 🎉
            </h1>

            <p className="mb-8 max-w-md text-lg text-gray-600">
              Please sign in to access all the cool features and start using our
              app.
            </p>

            <SignInComponent />

            <div className="mt-8 text-sm text-gray-500">
              <p>By signing in, you&apos;ll be able to:</p>
              <ul className="mt-2 space-y-1">
                <li>• Access your personal dashboard</li>
                <li>• Save your preferences</li>
                <li>• Use all app features</li>
              </ul>
            </div>
          </div>
        </div>
      )}
    </div>
  );
}
```

Now we have a good looking signin page integrated. You can go ahead and test it. Once you click on click on sign in you should be able to see the discord login and on successful auth we will see the notes ui we created.


Welcome to Our App! 🎉
Please sign in to access all the cool features and start using our app.
Sign In
By signing in, you'll be able to:
• Access your personal dashboard
• Save your preferences
• Use all app features


Sign in with Discord


Welcome back, Maverick_017! 👋
Sign Out
Notes App
Write a note...
Add Note
No notes yet. Write one above.

Webhooks:

Webhooks are a simple way for one application to automatically send information to another when a specific event occurs. Instead of constantly checking for updates, a webhook works like a notification system that pushes data in real time. For example, when you create or edit a note in an app, the system can immediately send a small message to another server with details about what just happened. This helps different systems stay synchronized without the need for manual refreshes or repeated requests.

In practice, a webhook is just an HTTP request sent to a URL that you define. That URL belongs to another service or application that listens for incoming data. When the webhook is triggered, the app sends a short JSON payload describing the event. This makes webhooks very useful for integrating separate systems, such as sending notifications, updating databases, or triggering automation workflows whenever something changes in your main application.

For this workshop we will utilize a third party online server which can receive and show us webhook events. It can be accessed below using the following link

https://webhook.site

We will utilize this later once we have webhooks integrated and we can set several events for testing this. First we will need to create more models and a change to the User model to have webhook events linked to their account.

To the **schema.prisma,** add the following snippet which adds the models WebhookInfo, WebhookEvent, EventType.

```
// This setup allows for flexibility but we'll scope it to
// just one host for this demo

// Each hostURL gets many events
model WebhookInfo {
  ID          String        @id @default(cuid())
  userID      String        @unique
  hostURL     String // The target of all webhooks
  webhookEvents WebhookEvent[] // Can listen to many events
  user        User          @relation(fields: [userID], references: [id])
}

model WebhookEvent {
  ID          String     @id @default(cuid())
  event       EventType // The corresponding event type
  webhookInfoID String // The ID of the parent Webhook
      webhookInfo          WebhookInfo   @relation(fields:   [webhookInfoID],
references: [ID])
  enabled     Boolean    @default(false)
  createdOn DateTime @default(now())
  @@unique([webhookInfoID, event], name: "webhookInfoID_event")
}

enum EventType {
  NOTE_CREATE
  NOTE_READ
  NOTE_UPDATE
  NOTE_DELETE
}
```

Now, we again apply the changes to the database using the following command.

npx prisma migrate dev --name webhookSetup

(or)

pnpm dlx prisma migrate dev --name webhookSetup

Now lets prep the API routes by adding two new routers. Up until this point we only had one router so now we will have 3 in total and we will see how we can register these routers as we registered the note router. Create the files **webhookInfo.ts** and **webhookEvent.ts** in the **routerw** directory.

Use the following code for **webhookInfo.ts**.

```
import z from "zod";
import { createTRPCRouter, protectedProcedure } from "../trpc";

export const webhookInfoRouter = createTRPCRouter({
  setWebhookURL: protectedProcedure
    .input(z.string().url())
    .mutation(async ({ ctx, input }) => {
      const userID = ctx.session.user.id;

      const record = await ctx.db.webhookInfo.upsert({
        where: { userID }, // userID is unique
        update: { hostURL: input }, // update if exists
        create: { userID, hostURL: input }, // create if missing
        include: { webhookEvents: true },
      });

      return record;
    }),

  findWebhookInto: protectedProcedure.query(async ({ ctx }) => {
    return ctx.db.webhookInfo.findUnique({
      where: {
        userID: ctx.session.user.id,
      },
    });
  }),
});
```

This router handles how we save and retrieve each user's webhook details. Since it uses protectedProcedure, only logged in users can access it. The setWebhookURL endpoint takes a valid URL and uses an upsert operation to either create a new webhook record or update the existing one for that user. The findWebhookInfo endpoint then fetches that stored data whenever needed. In short, this router lets every authenticated user securely manage their own webhook URL inside the system.

Use the following code for **webhookEvent.ts**.

```
import z from "zod";
import { createTRPCRouter, protectedProcedure } from "../trpc";
import { EventType } from "@prisma/client";
import { TRPCError } from "@trpc/server";

export const webhookEventRouter = createTRPCRouter({
  setEvent: protectedProcedure
    .input(
      z.object({
        event: z.nativeEnum(EventType),
        enabled: z.boolean(),
      }),
    )
    .mutation(async ({ ctx, input }) => {
      const userId = ctx.session.user.id;

      // require a configured webhook host for this user
      const info = await ctx.db.webhookInfo.findUnique({
        where: { userID: userId },
        select: { ID: true },
      });
      if (!info)
        throw new TRPCError({
          code: "BAD_REQUEST",
          message: "Webhook host not configured",
        });

      const row = await ctx.db.webhookEvent.upsert({
        where: {
```

```
                webhookInfoID_event: { webhookInfoID: info.ID, event:
input.event },
      },
      update: { enabled: input.enabled },
      create: {
        webhookInfoID: info.ID,
        event: input.event,
        enabled: input.enabled,
      },
    });

    return row;
  }),

 getEvents: protectedProcedure.query(async ({ ctx }) => {
  const userId = ctx.session.user.id;
  const info = await ctx.db.webhookInfo.findUnique({
    where: { userID: userId },
    include: { webhookEvents: true },
  });
  if (!info) return [];
  return info.webhookEvents;
 }),
});
```

This router manages which webhook events a user wants to enable or disable. Like before, it uses protectedProcedure, so only authenticated users can access it. The setEvent endpoint lets a user turn specific event types on or off, but first checks if they have a webhook host configured and if not, it throws an error. It then uses an upsert to update the existing event or create a new one as needed. The getEvents endpoint retrieves all webhook events linked to the user's webhook setup. Now lets register these routers in the root.ts file.

Add the following imports first

```
import { webhookInfoRouter } from "./routers/webhookInfo";
import { webhookEventRouter } from "./routers/webhookEvent";
```

Then add the imported routers into the AppRouter.

```
export const appRouter = createTRPCRouter({
  post: postRouter,
  note: noteRouter,
  webhookInfo: webhookInfoRouter,
  webhookEvent: webhookEventRouter,
});
```

We will also need a way to call the webhook endpoint whenever the event is triggered. It calls the webhook service provided by the user i.e., webhook.site. Lets create a service that performs a post request onto that endpoint with the event details. Create a **webhookService.ts** file under **server/** and paste the following code into it.

```
import type { EventType } from "@prisma/client";

export class WebhookService {
  static async updateWebhook<T>(url: string, type: EventType, data: T) {
    const urlObj = new URL(url);

    try {
      const res = await fetch(urlObj.toString(), {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ type, timestamp: Date.now(), data }),
      });
      if (!res.ok) {
        console.error(
          `Webhook request failed: ${res.status} ${res.statusText}`,
        );
      }
    } catch (err) {
```

```
      console.error("Webhook request error:", err);
    }
  }
}
```

Add the following line in the trpc.ts file.

```
export type TRPCContext = Awaited<ReturnType<typeof
createTRPCContext>>;
```

This line defines a shared type for the tRPC context so every router knows what data and helpers are available, such as the database or session.

Now, we need to update the Note api endpoints to prepare them for working with webhooks so that the events can be triggered and the subsequent ping is sent to the webhook.

```
import { WebhookService } from "src/server/webhookService";
import z from "zod";
import {
  createTRPCRouter,
  protectedProcedure,
  type TRPCContext,
} from "../trpc";
import type { EventType } from "@prisma/client";

export const noteRouter = createTRPCRouter({
  /** Create */
  addNote: protectedProcedure
    .input(z.string().trim().min(1))
    .mutation(async ({ ctx, input }) => {
      const note = await ctx.db.note.create({
        data: {
          content: input,
          title: "Some title",
          updatedOn: new Date(),
        },
```

```
    });
    await updateHost(ctx, "NOTE_CREATE", note);
    return note;
  }),

/* UPDATE */
updateNote: protectedProcedure
  .input(
    z.object({
      id: z.string().trim(),
      content: z.string().trim().min(2),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    let success = false;
    const note = await ctx.db.note.update({
      where: {
        id: input.id,
      },
      data: {
        content: input.content,
      },
    });
    await updateHost(ctx, "NOTE_UPDATE", note);
    if (note) success = true;
    return success;
  }),

// READ
findAllNotes: protectedProcedure.query(async ({ ctx }) => {
  const foundNotes = (await ctx.db.note.findMany()) ?? null;
  await updateHost(ctx, "NOTE_READ", foundNotes);
  return foundNotes;
}),

// DELETE
deleteNote: protectedProcedure
  .input(z.string().trim().min(1))
  .mutation(async ({ ctx, input }) => {
    let success = false;
```

```
      const note = await ctx.db.note.delete({
        where: {
          id: input,
        },
      });
      if (note) success = true;

      await updateHost(ctx, "NOTE_DELETE", note);
      return success;
    }),
});

async function updateHost<T>(ctx: TRPCContext, type: EventType, data: T)
{
  if (!ctx.session) return;
  const webhookInfo = await ctx.db.webhookInfo.findFirst({
    where: {
      userID: ctx.session.user.id,
    },
    include: {
      webhookEvents: true,
    },
  });

  if (!webhookInfo) return;

  for (const eventInfo of webhookInfo.webhookEvents) {
    if (eventInfo.event === type && eventInfo.enabled) {
          await  WebhookService.updateWebhook(webhookInfo.hostURL,  type,
data);
      return;
    }
  }
}
```

With this we have completed all the setup for the backend, now we need to update the frontend to make the api calls, get and set the webhook urls, set specific event listeners, etc. First we add in the placeholders. Please replace your code with the code below and later we will discuss what changes we have made.

Please use the code snippets from the following Link/QRCode

https://drive.google.com/drive/folders/1RSMSXNzi280eqwReNFZsnpn7WagLVFeR?usp=sharing

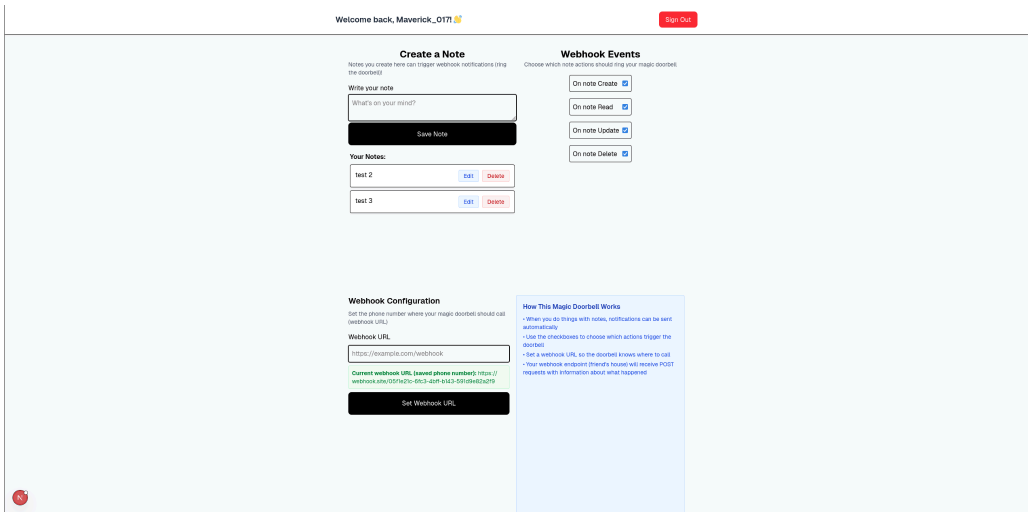
Here in Part 1 please replace the code with the snippet from **HomeComponent_P1.txt.** Here we replaced the plain text edit and delete options with styled buttons that make the actions clearer and the layout more polished. The page is now divided into organized sections, with the top area for managing notes, a right panel prepared for webhook event selections, and a bottom section for the webhook configuration form. Next we will implement the webhook handlers to listen for the events, set the webhook url and set what kind of events we wanna send notifications to.

For part 2, please replace the code with the snippet from **HomeComponent_P2.txt.** Here we connected the webhook sections to the backend and made them functional. The webhook URL form now saves and retrieves the configured endpoint for each user, and the event toggles update which note actions trigger webhooks. We also introduced backend

communication through tRPC mutations and queries to synchronize data between the client and the database. The app now supports real webhook configuration and event control.

For part 3, please replace the code with the snippet from **HomeComponent_P3.txt.** The webhook configuration now shows the saved URL, and the event toggles are fully connected to the backend, letting users decide which actions should send webhook notifications. We also added an informational box at the bottom explaining how the webhook system works, helping users understand the logic behind each part of the interface.

For testing these lets goto the website **https://webhook.site.** Here you will obtain a public webhook url which you can use to test the functionality. Now once you are logged in with discord and at the home page you can tick the checkboxes for the events you wanna listen for and paste the webhook url in the input box for it and then click the button which saves the webhook url to db and this webhook url will associated to your user profile in the db.



From here you can perform the notes actions and you can see the actions triggered on the webhook site.