24
9
21

Shoubhit Dash
Posted on 12 thg 8

# Build a full stack app with create-t3-app

#webdev #nextjs #tutorial

Hey there! Today we'll be building an application with the T3 stack. We're going to building a Guestbook inspired by Lee Robinson's Guestbook. Let's get right into it!

## Getting Started

Let's set up a starter project with `create-t3-app`!

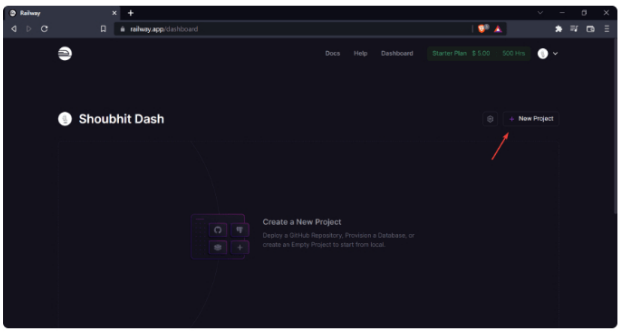
```
npx create-t3-app@latest
```

We're going to utilize all parts of the stack.

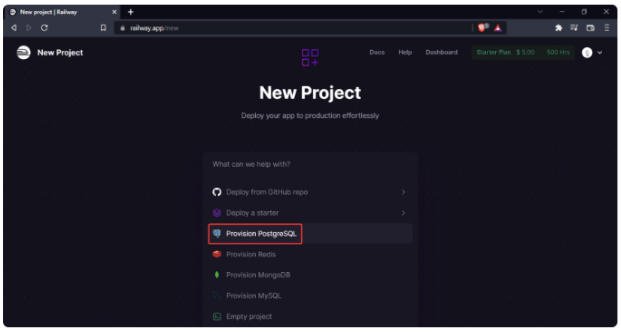
```
 ___ ___ ___   __ ____ ___   ____ ___    __   ___ ___
 / __| _ \ __| /  \_  _| __| |_  _|__ /  /  \ | _ \ _ \
| (__|   / _| / /\ \| | | _|   | | |_ \ / /\ \|  _/  _/
 \___|_|_\___|_/__\_\_| |___|  |_| |___/ /_/__\_\_| |_|

? What will your project be called? guestbook
? Will you be using JavaScript or TypeScript? TypeScript
Good choice! Using TypeScript!
? Which packages would you like to enable? nextAuth, prisma, tailwind, trpc
? Initialize a new git repository? Yes
Nice one! Initializing repository!
? Would you like us to run pnpm install? Yes
Alright. We'll install the dependencies for you!

Using: pnpm
```

Let's also set up a Postgres database on Railway. Railway makes it super simple to quickly set up a database.

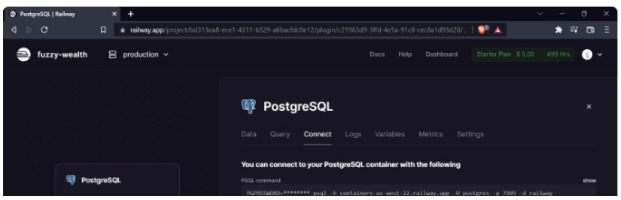
Go to Railway and log in with GitHub if you haven't already. Now click on `New Project`.




Now provision Postgres.




It's as simple as that. Copy the connection string from the `Connect` tab.




Shoubhit Dash
Follow
17 yo self-taught developer, trying to make the world a cooler place and stuff like that
JOINED
11 thg 3, 2022

More from Shoubhit Dash
t3 stack and my most popular open source project ever
#webdev #opensource #typescript
Build end-to-end typesafe APIs with tRPC
#typescript #nextjs #webdev #api
Make and deploy your own blog in under 30 mins
#react #tailwindcss #webdev #beginners

Let's start coding! Open the project in your favourite code editor.

There are a lot of folders but don't be overwhelmed. Here's a basic overview.

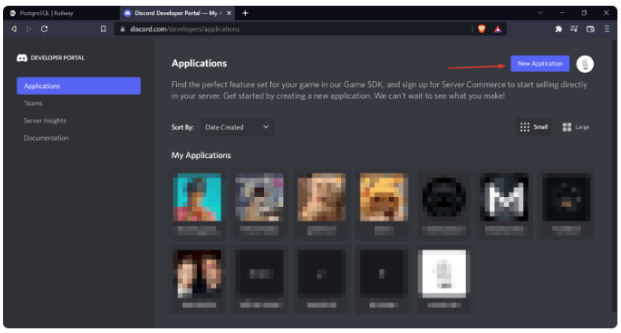
- `prisma/*` - The `prisma` schema.
- `public/*` - Static assets including fonts and images.
- `src/env/*` - Validation for environment variables.
- `src/pages/*` - All the pages of the website.
- `src/server/*` - The backend, which is a tRPC server.
- `src/styles/*` - Global CSS files, but we're going to be using Tailwind CSS for most of our styles.
- `src/types/*` - Next Auth type declarations.
- `src/utils/*` - Utility functions.

Open the `.env` file and paste the connection string in `DATABASE_URL`.

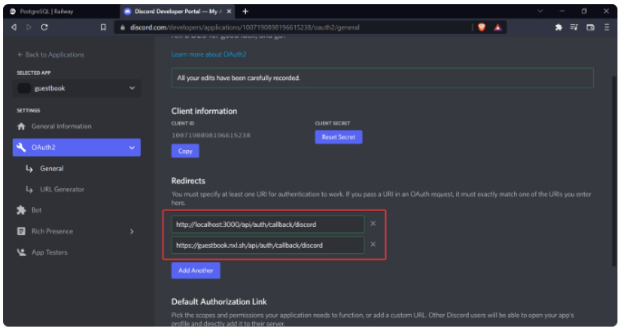
You'll notice we have Discord OAuth set up using `next-auth`, so we also need a `DISCORD_CLIENT_ID` and `DISCORD_CLIENT_SECRET`. Let's set that up.

## Setting up authentication

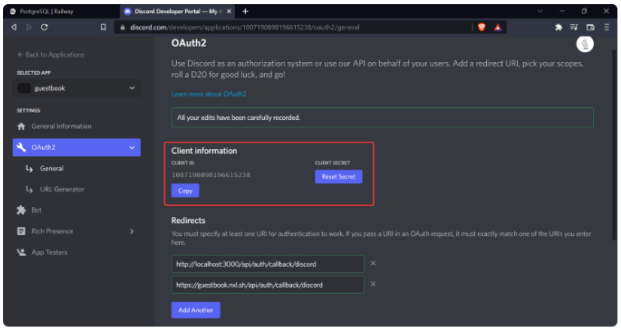
Go to the Discord Developers Portal and create a new application.




Go to `OAuth2/General` and add all of callback URLs to `Redirects`. For localhost the callback URL is `http://localhost:3000/api/auth/callback/discord`. I also added the production URL ahead of time.




Copy the client ID and secret and paste that into `.env`.




Set the `NEXTAUTH_SECRET` as some random string too. Now we have all of out environment variables configured.

Let's also change the database to `postgresql` and uncomment the `@db.Text` annotations in the `Account` model in `prisma/schema.prisma`. All the models

you see in the schema are necessary for Next Auth to work.

```
schema.prisma
datasource db {
  provider = "postgresql"
  // NOTE: When using postgresql, mysql or sqlserver, uncomment the @db.text annotations in model Account below
  // Further reading:
  // https://next-auth.js.org/adapters/prisma#create-the-prisma-schema
  // https://www.prisma.io/docs/reference/api-reference/prisma-schema-reference#string
  url      = env("DATABASE_URL")
}

model Example {
  id String @id @default(cuid())
}

// Necessary for Next auth
model Account {
  id                String  @id @default(cuid())
  userId            String
  type              String
  provider          String
  providerAccountId String
  refresh_token     String? @db.Text
  access_token      String? @db.Text
  expires_at        Int?
  token_type        String?
  scope             String?
  id_token          String? @db.Text
NORMAL  main                    schema.prisma                    [16/71]  22%
```

Let's push this schema to our Railway Postgres database. This command will push our schema to Railway and generate type definations for the Prisma client.

```
npx prisma db push
```

Now run the dev server.

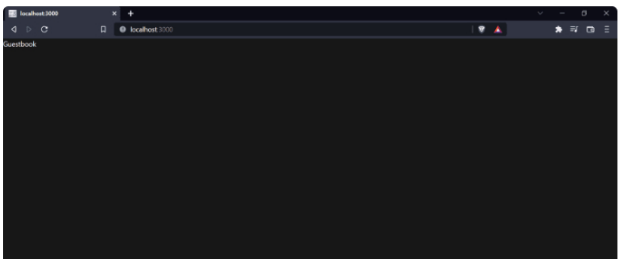
```
npm run dev
```

Go to the `src/pages/index.tsx` and delete all the code, let's just render a heading.

```
// src/pages/index.tsx

const Home = () => {
  return (
    <main>
      <h1>Guestbook</h1>
    </main>
  );
};

export default Home;
```

I can't look at light theme, so lets apply some global styles in `src/styles/globals.css` to make this app dark theme.

```
// src/styles/globals.css

@tailwind base;
@tailwind components;
@tailwind utilities;

body {
  @apply bg-neutral-900 text-neutral-100
}
```



Much better.

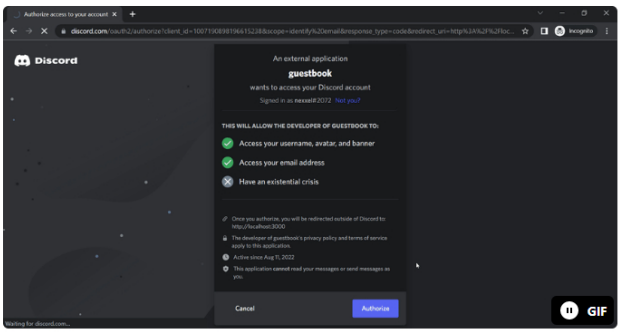
If you look at `src/pages/api/auth/[...nextauth].ts`, you can see we have Discord OAuth already set up using Next Auth. Here is where you can add more OAuth providers like Google, Twitter, etc.

Now let's create a button to let users login with Discord. We can use the `signIn()` function from Next Auth.

```
// src/pages/index.tsx

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

const Home = () => {
  return (
    <main>
      <h1>Guestbook</h1>

      <button onClick={() => signIn("discord")}>Login with Discord</butt
    </main>
  );
};

export default Home;
```

We can use the `useSession()` hook to get the session for the user. While we're at it, we can also use the `signOut()` function to implement log out functionality.

```
// src/pages/index.tsx

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

const Home = () => {
  const { data: session, status } = useSession();

  if (status === "loading") {
    return <main>Loading...</main>;
  }

  return (
    <main>
      <h1>Guestbook</h1>
      {session ? (
        <div>
          <p>
            hi {session.user?.name}
          </p>

          <button onClick={() => signOut()}>Logout</button>
        </div>
      ) : (
        <div>
          <button onClick={() => signIn("discord")}>Login with Discord</
        </div>
      )}
    </main>
  );
};

export default Home;
```




Great! We now have auth working. Next Auth really makes it stupidly simple.

## Backend

Let's work on the backend now. We'll be using tRPC for our API layer and Prisma for connecting and querying our database.

We're going to have to modify our prisma schema and add a `Guestbook` model. Each message in the guestbook will have a name, and a message. Here's how the model will look like.

```
// prisma/schema.prisma

model Guestbook {
    id        String   @id @default(cuid())
    createdAt DateTime @default(now())
    name      String
    message   String   @db.VarChar(100)
}
```

Let's push this modified to our Railway Postgres database.

```
npx prisma db push
```

Now let's get to the fun part - its tRPC time. Go ahead and delete the `protected-example-router.ts` and `protected-router.ts` and `example.ts` in `src/server/router`. First we're going to define a mutation to post messages to our database.

```
// src/server/router/guestbook.ts

import { z } from "zod";
import { createRouter } from "./context";

export const guestbookRouter = createRouter().mutation("postMessage", {
  input: z.object({
    name: z.string(),
    message: z.string(),
  }),
  async resolve({ ctx, input }) {
    try {
      await ctx.prisma.guestbook.create({
        data: {
          name: input.name,
          message: input.message,
        },
      });
    } catch (error) {
      console.log(error);
    }
  },
});
```

Here we have a tRPC mutation that uses zod to validate the input and has a resolve function that runs a simple prisma query to create a new row in the `Guestbook` table.

Working with prisma is an absolutely wonderful example. The autocomplete and typesafety is amazing.

We also want this mutation to be protected. Here we can use tRPC middlewares. If you take a look at the `src/server/context.ts`, we're using [`unstable_getServerSession`] from Next Auth that gives us access to the session on the server. We're passing that into our tRPC context. We can use this session to make our mutation protected.

```
// src/server/router/guestbook.ts

export const guestbookRouter = createRouter()
  .middleware(async ({ ctx, next }) => {
```

```
    // Any queries or mutations after this middleware will
    // raise an error unless there is a current session
    if (!ctx.session) {
      throw new TRPCError({ code: "UNAUTHORIZED" });
    }
    return next();
  })
  .mutation("postMessage", {
    // ...
```

Next, let's write a query to get all messages in the guestbook, this one will be pretty simple too.
This doesn't need to be protected so we can put it above the middleware.

```
// src/server/router/guestbook.ts

export const guestbookRouter = createRouter()
  .query("getAll", {
    async resolve({ ctx }) {
      try {
        return await ctx.prisma.guestbook.findMany({
          select: {
            name: true,
            message: true,
          },
          orderBy: {
            createdAt: "desc",
          },
        });
      } catch (error) {
        console.log("error", error);
      }
    },
  })
  .middleware(async ({ ctx, next }) => {
    // ...
```

Here getting just the name and message from all the rows from the `Guestbook` model and sorting it in descending order by the `createdAt` field.

Now merge this router in the main `appRouter`.

```
// src/server/router/index.ts

import superjson from "superjson";
import { createRouter } from "./context";
import { guestbookRouter } from "./guestbook";

export const appRouter = createRouter()
  .transformer(superjson)
  .merge("guestbook.", guestbookRouter);

// export type definition of API
export type AppRouter = typeof appRouter;
```

And we're pretty much done here on the backend part. Let's work on the UI now.

## Frontend

Let's first center everything.

```
// src/pages/index.tsx

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

const Home = () => {
  const { data: session, status } = useSession();

  if (status === "loading") {
    return <main className="flex flex-col items-center pt-4">Loading...<
  }

  return (
    <main className="flex flex-col items-center">
      <h1 className="text-3xl pt-4">Guestbook</h1>
      <p>
        Tutorial for <code>create-t3-app</code>
      </p>

      <div className="pt-10">
        {session ? (
          <div>
            <p>hi {session.user?.name}</p>

            <button onClick={() => signOut()}>Logout</button>
```

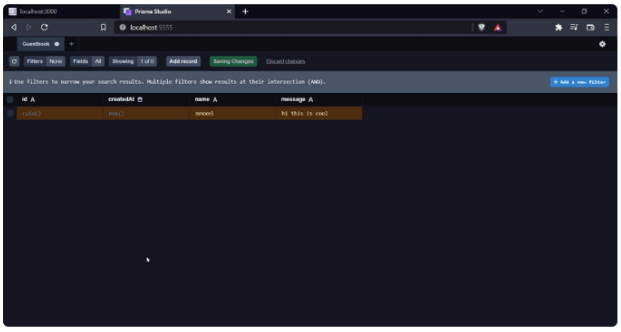
```
          </div>
        ) : (
          <div>
            <button onClick={() => signIn("discord")}>
              Login with Discord
            </button>
          </div>
        )}
      </div>
    </main>
  );
};

export default Home;
```

I also made the heading bigger and added some padding between the elements.

Let's use our tRPC query to get all the messages for the guestbook in the database. But we don't have any data right now. We can use [Prisma Studio](#) to some data manually.

```
npx prisma studio
```

It will automatically open on `http://localhost:5555`. Go to the `Guestbook` table and add a bunch of records like this.

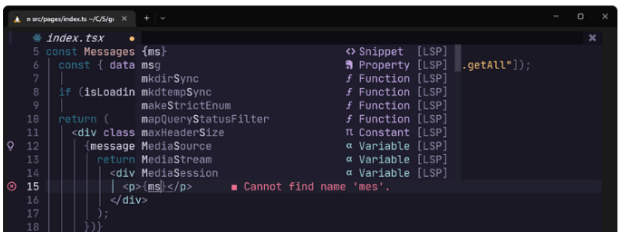


Now that we have data, we can use the query and display the data. For this we can use the tRPC react-query wrapper. Let's create a component for this in `src/pages/index.tsx`.

```
// src/pages/index.tsx

import { trpc } from "../utils/trpc";

const Messages = () => {
  const { data: messages, isLoading } = trpc.useQuery(["guestbook.getAll]

  if (isLoading) return <div>Fetching messages...</div>;

  return (
    <div className="flex flex-col gap-4">
      {messages?.map((msg, index) => {
        return (
          <div key={index}>
            <p>{msg.message}</p>
            <span>- {msg.name}</span>
          </div>
        );
      })}
    </div>
  );
};
```

Here we're using `useQuery()` and mapping over the array it returns.

Of course here too we have wonderful typesafety and autocomplete.



Now render this component in the `Home` component.

```
// src/pages/index.tsx

 <main className="flex flex-col items-center">
    <h1 className="text-3xl pt-4">Guestbook</h1>
    <p>
      Tutorial for <code>create-t3-app</code>
    </p>

    <div className="pt-10">
      {session ? (
        <div>
          <p>hi {session.user?.name}</p>

          <button onClick={() => signOut()}>Logout</button>

          <div className="pt-10">
            <Messages />
          </div>
        </div>
      ) : (
        <div>
          <button onClick={() => signIn("discord")}>
            Login with Discord
          </button>

          <div className="pt-10" />
          <Messages />
        </div>
      )}
    </div>
  </main>
```

Let's now create a form and use our tRPC mutation there.

```
// src/pages/index.tsx

const Home = () => {
  const { data: session, status } = useSession();
  const [message, setMessage] = useState("");
  const postMessage = trpc.useMutation("guestbook.postMessage");

  if (status === "loading") {
    return <main className="flex flex-col items-center pt-4">Loading...<
  }

  return (
    <main className="flex flex-col items-center">
      <h1 className="text-3xl pt-4">Guestbook</h1>
      <p>
        Tutorial for <code>create-t3-app</code>
      </p>

      <div className="pt-10">
        {session ? (
          <div>
            <p>hi {session.user?.name}</p>

            <button onClick={() => signOut()}>Logout</button>

            <div className="pt-6">
              <form
                className="flex gap-2"
                onSubmit={(event) => {
                  event.preventDefault();

                  postMessage.mutate({
                    name: session.user?.name as string,
                    message,
                  });

                  setMessage("");
                }}
              >
                <input
                  type="text"
                  value={message}
                  placeholder="Your message..."
                  maxLength={100}
                  onChange={(event) => setMessage(event.target.value)}
                  className="px-4 py-2 rounded-md border-2 border-zinc-8
                />
```

```
            <button
              type="submit"
              className="p-2 rounded-md border-2 border-zinc-800 foc
            >
              Submit
            </button>
          </form>
        </div>

        <div className="pt-10">
          <Messages />
        </div>
      </div>
    ) : (
      <div>
        <button onClick={() => signIn("discord")}>
          Login with Discord
        </button>

        <div className="pt-10" />
        <Messages />
      </div>
    )}
  </div>
</main>
);
};
```

Here we have a form and we're using `useMutation()` to post the data to the database. But you'll notice one problem here. When we click on the submit button, it does post the message to the database, but the user doesn't get any immediate feedback. Only on refreshing the page, the user can see the new message.

For this we can use optimistic UI updates! `react-query` makes this trivial to do. We just need to add some stuff to our `useMutation()` hook.

```
// src/pages/index.tsx

const ctx = trpc.useContext();
const postMessage = trpc.useMutation("guestbook.postMessage", {
  onMutate: () => {
    ctx.cancelQuery(["guestbook.getAll"]);

    let optimisticUpdate = ctx.getQueryData(["guestbook.getAll"]);
    if (optimisticUpdate) {
      ctx.setQueryData(["guestbook.getAll"], optimisticUpdate);
    }
  },
  onSettled: () => {
    ctx.invalidateQueries(["guestbook.getAll"]);
  },
});
```

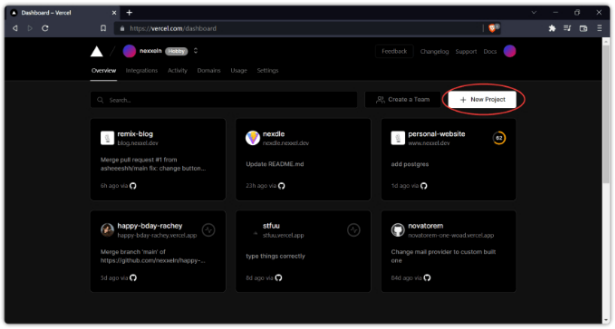
The code is pretty much self-explanatory. You can read more about optimistic updates with `react-query` here.

We're pretty much done with the coding part! That was pretty simple wasn't it. The T3 stack makes it super easy and quick to build full stack web apps. Let's now deploy our guestbook.

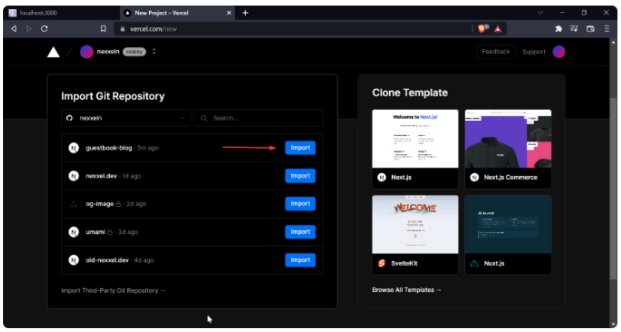
## Deployement

We're going to use Vercel to deploy. Vercel makes it really easy to deploy NextJS apps, they are the people who made NextJS.

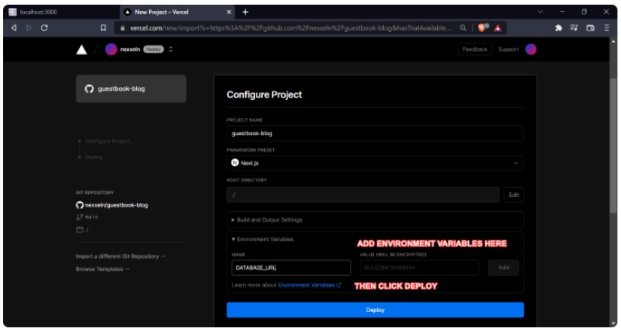
First, push your code to a GitHub repository. Now, go to Vercel and sign up with GitHub if you haven't already.




Then click on `New Project` and import your newly created repository.




Now we need to add environment variables, so copy paste all the environment variables to Vercel. After you've done that, click `Deploy`.




Add a custom domain if you have one and you're done! Congratulations!

All the code can be found here. You can visit the website at guestbook.nxl.sh.

## Credits

- Lee Robinson for the idea of a guestbook.
- JAR and Krish for proof reading.

## Discussion (0)

Subscribe

Add to the discussion

Code of Conduct • Report abuse

## Read next

**File Upload in React**
sharath mohan - Aug 11

**Deploying Modern Applications Using Kubernetes and Harness**
Pavan Belagatti - Aug 11

**Every javascript developer should learn Typescript 🔥 🔥 and here is why**
TURINUMUGISHA Souvenir - Aug 11

**Code Smell 157 - Balance at 0**
Maxi Contieri - Aug 10

DEV Community — A constructive and inclusive social network for software developers. With you every step of your journey.

Built on Forem — the open source software that powers DEV and other inclusive communities.
Made with love and Ruby on Rails. DEV Community © 2016 - 2022.