

# NextAuth Credentials — easy signup & login with email & password (Next.js 14 App router and Zod resolver)



PiotrDev · [Follow](#)

9 min read · Feb 3, 2024



132



3



## NEXTAUTH & ZOD INTRO

NextAuth.js is a robust, open-source authentication solution tailor-made for Next.js applications, seamlessly blending with Next.js and Serverless environments. It accommodates many popular sign-in options, including email and passwordless sign-ins, making it a versatile choice for developers. Although it demands a bit more effort to configure compared to solutions like Clerk, it rewards users with unmatched control over data management and unparalleled flexibility in customization without incurring extra costs.

In addition to its comprehensive authentication capabilities, which are compatible with any backend system (e.g., Active Directory, LDAP), NextAuth.js is compatible with JSON Web Tokens and database sessions.

A notable feature that enhances its form-handling capabilities is the integration with Zod, a TypeScript-first schema declaration and validation

**library.** By utilizing Zod alongside the Zod resolver with `react-hook-form`, developers can enforce strong typing and validation for form data seamlessly. This combination streamlines the development process and significantly improves data integrity and user experience by leveraging Zod's schema validation to catch errors early and ensure that only valid data is processed.

Let me walk you through the configuration of NextAuth.js with the modern Full Stack Next.js App (using app router, powered by TailwindCSS & shadcn)

See the repository [here](#).

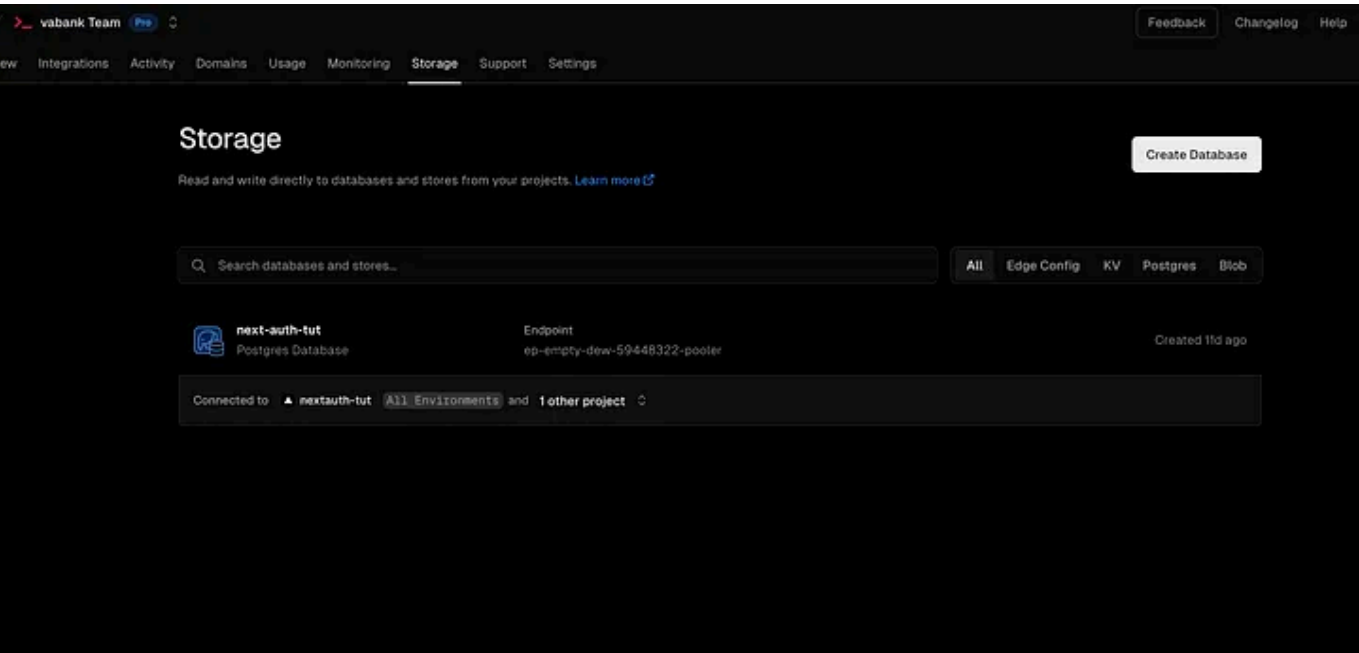


Photo by [Towfiq barbhuiya](#) on [Unsplash](#)

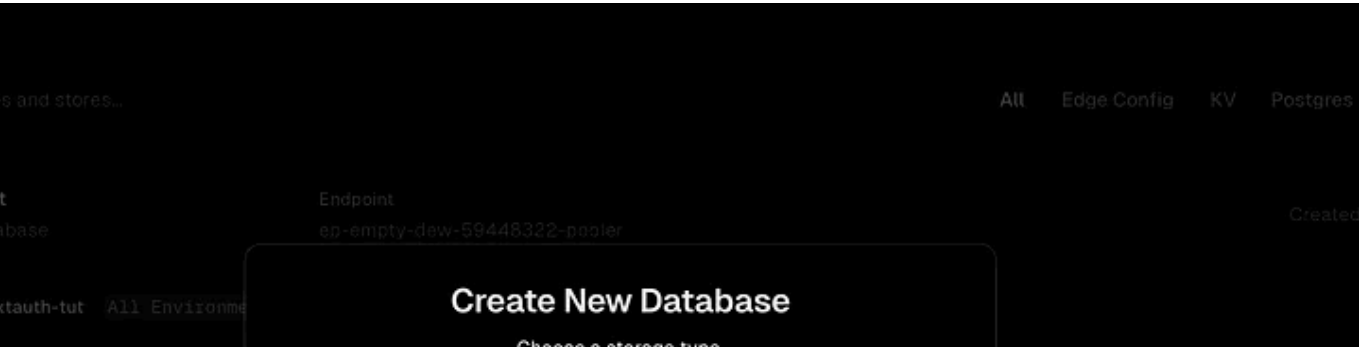
## INTRODUCTION

Assuming you have a Next.js project already initialized and pushed to GitHub, the following steps will guide you through setting up a PostgreSQL database directly on Vercel for your project.

1 — You click on **Storage** and then **Create Database** button.



2- You can then select Postgres Serverless SQL



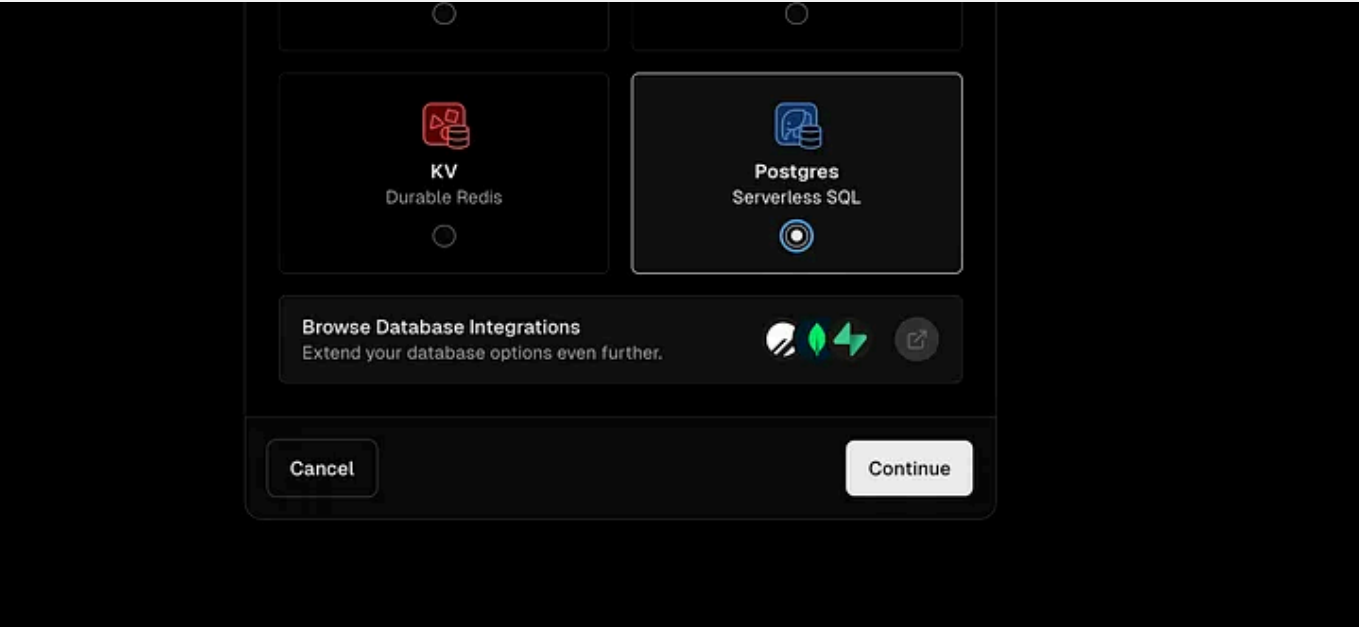
Open in app ↗

[Sign up](#) [Sign in](#)

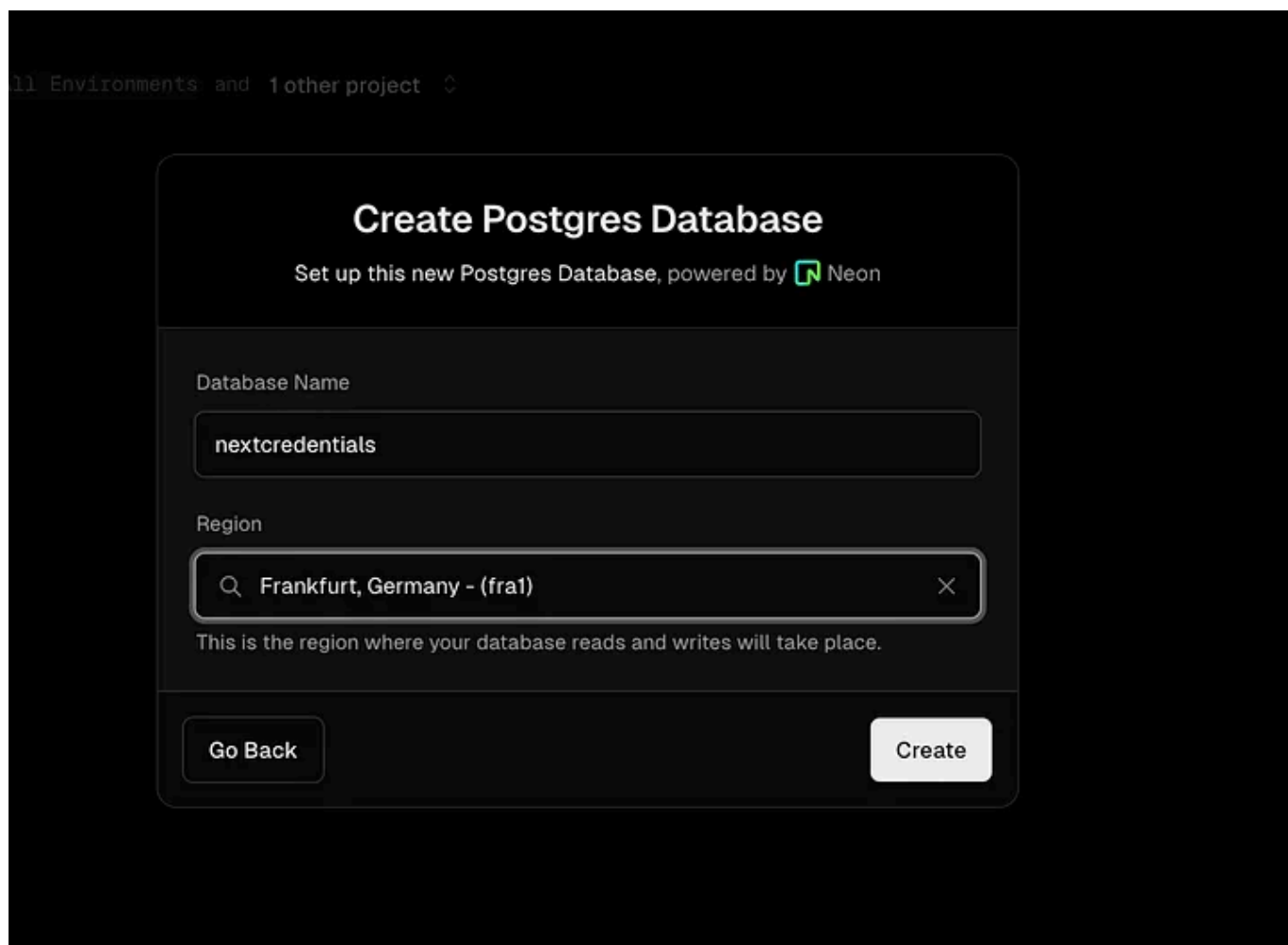
 Medium

 Search

 Write 



3 — add a name for your database and click **Create**



The screenshot shows the 'Create Postgres Database' interface. At the top, it says 'Set up this new Postgres Database, powered by Neon'. Below this, there are two input fields: 'Database Name' with the value 'nextcredentials' and 'Region' with the value 'Frankfurt, Germany - (fra1)'. A note below the region field states: 'This is the region where your database reads and writes will take place.' At the bottom, there are two buttons: 'Go Back' and 'Create'.

11 Environments and 1 other project

## Create Postgres Database

Set up this new Postgres Database, powered by Neon

Database Name

nextcredentials

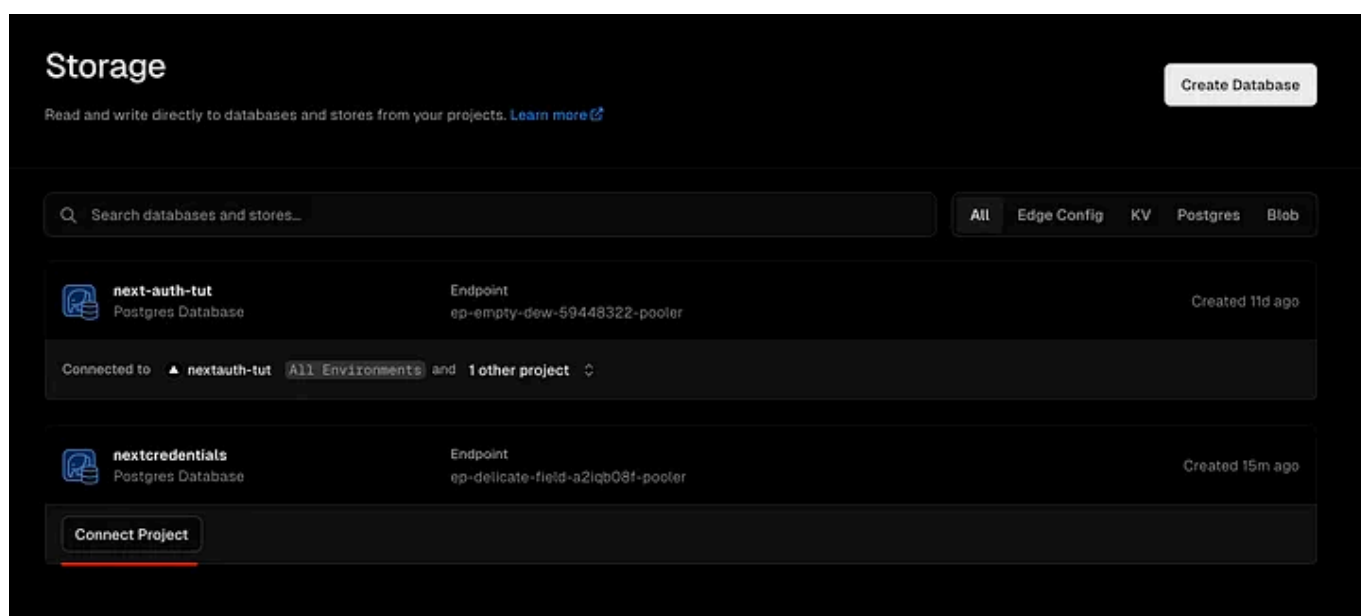
Region

Frankfurt, Germany - (fra1)

This is the region where your database reads and writes will take place.

Go Back Create

4 — and then connect the newly created database to your project



The screenshot shows the 'Storage' page in Vercel. It has a search bar and tabs for 'All', 'Edge Config', 'KV', 'Postgres', and 'Blob'. There are two database entries listed: 'next-auth-tut' and 'nextcredentials'. Each entry shows its name, type (Postgres Database), endpoint, and creation time. A 'Connect Project' button is at the bottom.

## Storage

Read and write directly to databases and stores from your projects. [Learn more](#)

Create Database

Search databases and stores...

All Edge Config KV Postgres Blob

next-auth-tut Postgres Database Endpoint ep-empty-dew-59448322-pooler Created 11d ago

Connected to nextauth-tut All Environments and 1 other project

nextcredentials Postgres Database Endpoint ep-delicate-field-a2lqb03f-pooler Created 15m ago

Connect Project

Make sure you have installed the **vercel CLI**:

```
npm i -g vercel
```

5 — then you link your project with the Postgres database on your environment (you will be asked 3 questions):

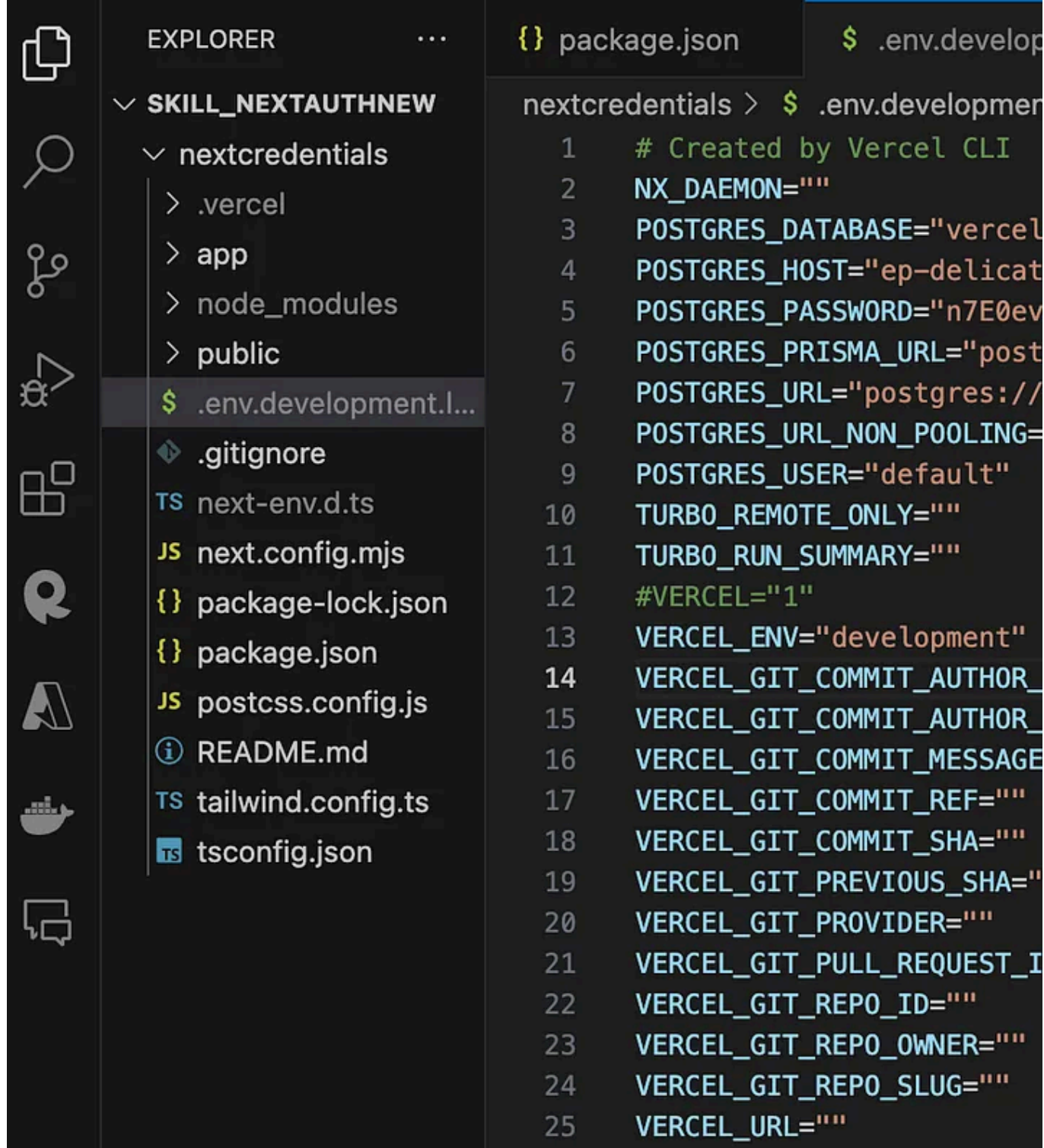
```
vercel link
```

6 — now you want to clone all the environment variables created while setting up Postgres on Vercel

```
vercel env pull .env.development.local
```

You can probably see that `.env.development.local` has been created with all the necessary credentials.

**I suggest commenting on “VERCEL=”1” ” as per the screenshot as it may force https while we are at our local host.**



## SETTING THE NECESSARY MODULES

I am using **Shadcn** to speed up the components' build.

```
npm i bcrypt next-auth
npm i --save-dev @types/bcrypt
npm i @vercel/postgres
npx shadcn-ui@latest init
npx shadcn-ui@latest add form
```

## SETTING UP OUR APP DIRECTORY AND THE LOGIC

Let's prepare the register and the login page

```
>app
  >login
    form.tsx
    page.tsx
  >register
    form.tsx
    page.tsx
```

and the API folder structure for NextAuth

```
>app
  >api
    >auth
      >[...nextauth]
        route.ts
```

## CLOSER LOOK AT NEXT AUTH DOCUMENTATION

The Credentials provider allows you to handle signing in with arbitrary credentials, such as a username and password, domain, or two-factor authentication or hardware device (e.g. Yubikey U2F/FIDO)

1 — Let's prepare **the route.ts within api/auth/[...nextauth]** according to the documentation. I have removed the authorize logic for now.

```
// app>api>auth>[...nextauth]>route.ts

import NextAuth from "next-auth/next";
import CredentialsProvider from "next-auth/providers/credentials";
import { sql } from "@vercel/postgres";
import { compare } from "bcrypt";
```



```

const handler = NextAuth({
  session: {
    strategy: "jwt",
  },

  pages: {
    signIn: "/login",
  },

  providers: [
    CredentialsProvider({
      // The name to display on the sign in form (e.g. 'Sign in with...')
      name: "Credentials",
      // The credentials is used to generate a suitable form on the sign in page
      // You can specify whatever fields you are expecting to be submitted.
      // e.g. domain, username, password, 2FA token, etc.
      // You can pass any HTML attribute to the <input> tag through the object.
      credentials: {
        email: {},
        password: {},
      },
      async authorize(credentials, req) {
        return null;
      },
    }),
  ],
});

export { handler as GET, handler as POST };

```

2 — also, let's prepare the **api/auth/register/route.ts** to see if we are correctly passing the values from the **Zod-validated form**

```

// app>api>auth>register>route.ts

import { NextResponse } from "next/server";

export async function POST(request: Request) {
  try {
    const { email, password } = await request.json();
    // YOU MAY WANT TO ADD SOME VALIDATION HERE

    console.log({ email, password });
  } catch (e) {
    console.log({ e });
  }
}

```



```
    return NextResponse.json({ message: "success" });
  }
}
```

### 3 — Let's prepare the register frontend:

```
import { getServerSession } from "next-auth";
import { redirect } from "next/navigation";

import FormPage from "./form";

export default async function RegisterPage() {
  const session = await getServerSession();

  if (session) {
    redirect("/");
  }

  return (
    <section className="bg-black h-screen flex items-center justify-center">
      <div className="w-[600px]">
        <FormPage />
      </div>
    </section>
  );
}
```

### 4 — and the FormPage itself (using react-hook-form and zod validation)

```
"use client";

import { zodResolver } from "@hookform/resolvers/zod";
import { useForm } from "react-hook-form";
import * as z from "zod";

import { Button } from "@components/ui/button";
import {
  Form,
  FormControl,
  FormDescription,
  FormField,
  FormItem,
  FormLabel,
  FormMessage,

```

```

} from "@components/ui/form";
import { Input } from "@components/ui/input";
import { toast } from "@components/ui/use-toast";

const FormSchema = z.object({
  username: z.string().min(2, {
    message: "Username must be at least 2 characters.",
  }),
  password: z.string().min(6, {
    message: "Password must be at least 6 characters.",
  }),
});

type FormData = z.infer<typeof FormSchema>;

export default function FormPage() {
  const form = useForm({
    resolver: zodResolver(FormSchema),
    defaultValues: {
      username: "",
      password: "",
    },
  });

  const onSubmit = async (data: FormData) => {
    console.log("Submitting form", data);

    const { username: email, password } = data;

    try {
      const response = await fetch("/api/auth/register", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ email, password }),
      });
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      // Process response here
      console.log("Registration Successful", response);
      toast({ title: "Registration Successful" });
    } catch (error: any) {
      console.error("Registration Failed:", error);
      toast({ title: "Registration Failed", description: error.message });
    }
  };

  return (
    <Form {...form} className="w-2/3 space-y-6">
      <form onSubmit={form.handleSubmit(onSubmit)}>
        <FormField
          control={form.control}
          name="username"
          render={({ field }) => (
            <FormItem>

```

```

        <FormLabel>Username</FormLabel>
        <FormControl>
          <Input placeholder="Username" {...field} />
        </FormControl>
        <FormDescription>
          This is your public display name.
        </FormDescription>
      </FormItem>
    )}
  />
  <FormField
    control={form.control}
    name="password"
    render={({ field }) => (
      <FormItem>
        <FormLabel>Password</FormLabel>
        <FormControl>
          <Input placeholder="Password" {...field} type="password" />
        </FormControl>
      </FormItem>
    )}
  />
  <Button type="submit">Submit</Button>
</form>
</Form>
);
}

```

The FormPage's interactivity forces it to be a client component, hence the **“use client”** directive above.

Please look closely into the onSubmit async function where we pass the form data (email, password) to **/api/auth/register**.

```

const onSubmit = async (data: FormData) => {
  console.log("Submitting form", data);

  const { username: email, password } = data;

  try {
    const response = await fetch("/api/auth/register", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ email, password }),
    });
    if (!response.ok) {

```

```

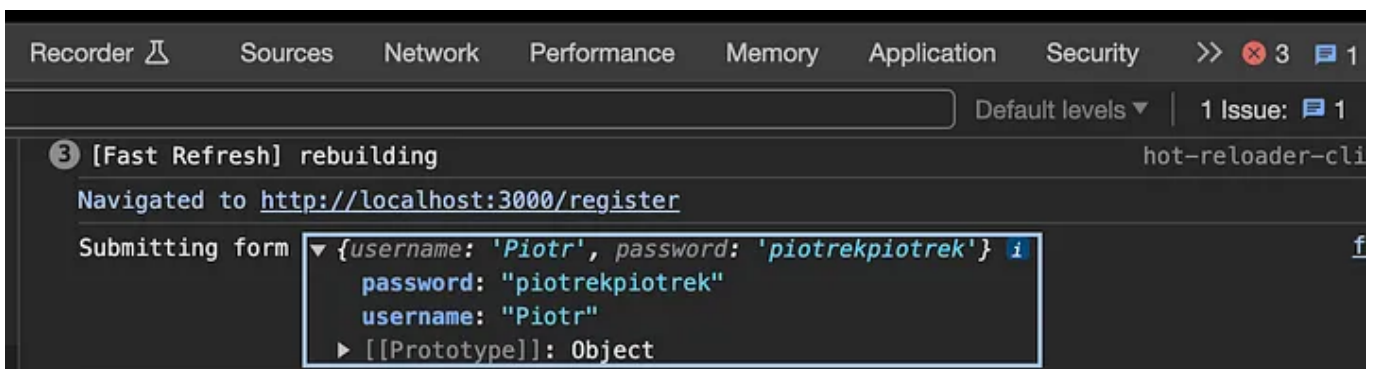
        throw new Error("Network response was not ok");
    }
    // Process response here
    console.log("Registration Successful", response);
    toast({ title: "Registration Successful" });
} catch (error: any) {
    console.error("Registration Failed:", error);
    toast({ title: "Registration Failed", description: error.message });
}
};

```

So at this stage, when visiting our /register page we should see the following:

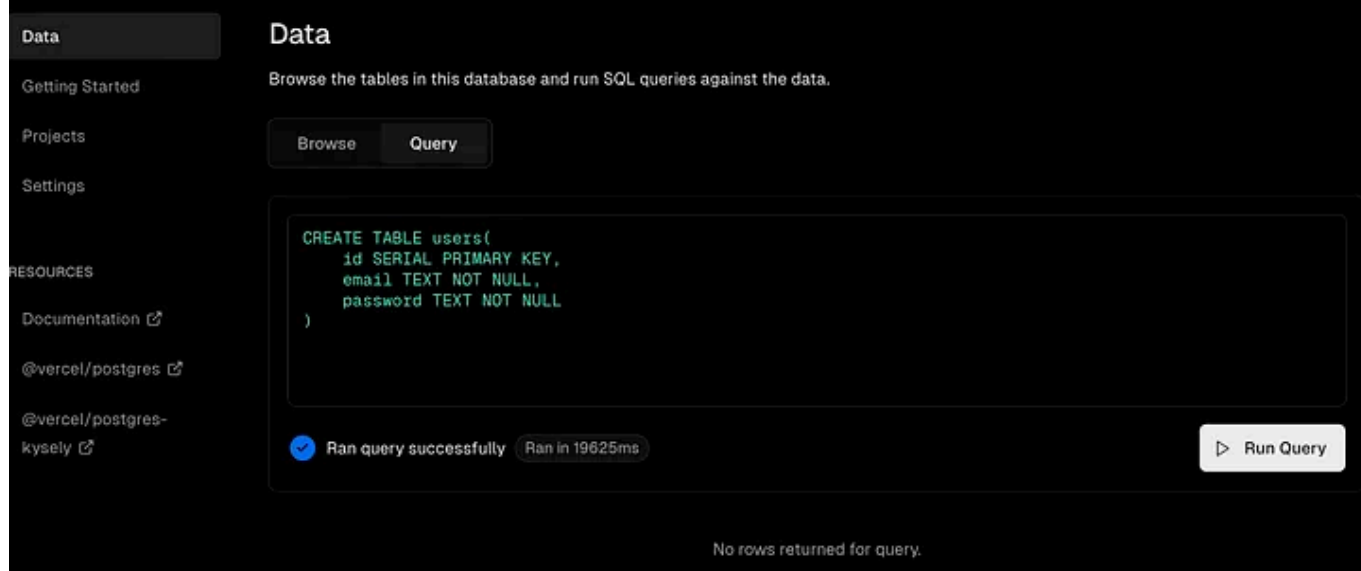


and, we should be able to see the inserted data in the console:



## DATABASE PREPARATION

We need a users' table in our database at this stage.



Now, let's incorporate the registration logic, which includes password hashing, within `api/auth/register/route.ts`:

```
import { NextResponse } from "next/server";
import { hash } from "bcrypt";
import { sql } from "@vercel/postgres";

export async function POST(request: Request) {
  try {
    const { email, password } = await request.json();
    // YOU MAY WANT TO ADD SOME VALIDATION HERE

    console.log({ email, password });

    const hashedPassword = await hash(password, 10);

    const response =
      await sql`INSERT INTO users (email, password) VALUES (${email}, ${hashedPa
    } catch (e) {
      console.log({ e });
    }

    return NextResponse.json({ message: "success" });
  }
}
```

now, after tackling the registration, we should see our registered customer in the Postgres db:

Data

Getting Started

Projects

Settings

Data

Browse the tables in this database and run SQL queries against the data.

Browse

Query

users

RESOURCES

Documentation

@vercel/postgres

@vercel/postgres-kysely

Id	Email	Password
1	piotr@dev.com	\$2b\$10\$YSsJDXfY4lYdldqdQrzlo.qbAe6AYm4tanHOxL/9CVC6314S27m

Ran in 228ms

**BIG MILESTONE — REGISTRATION TO POSTGRES DB FROM OUR NEXT.JS FRONTEND IS WORKING AS EXPECTED :)**

**LOGIN FUNCTIONALITY NOW**

Here is my code for the login page:

```
// >app>login>page.tsx

import { getServerSession } from "next-auth";
import { redirect } from "next/navigation";
import LoginForm from "../form";

export default async function LoginPage() {
  const session = await getServerSession();
  console.log({ session });

  if (session) {
    redirect("/");
  }

  return (
    <section className="bg-black h-screen flex items-center justify-center">
      <div className="w-[600px]">
        <LoginForm />;
      </div>
    </section>
  );
}
```

and, the LoginForm component:

```
"use client";

import { signIn } from "next-auth/react";
import { useRouter } from "next/navigation";
import { zodResolver } from "@hookform/resolvers/zod";
import { useForm } from "react-hook-form";
import * as z from "zod";

import { Button } from "@components/ui/button";
import {
  Form,
  FormControl,
  FormDescription,
  FormField,
  FormItem,
  FormLabel,
  FormMessage,
} from "@components/ui/form";
import { Input } from "@components/ui/input";
import { toast } from "@components/ui/use-toast";

const FormSchema = z.object({
  email: z.string().email({
    message: "Invalid email address.",
  }),
  password: z.string().min(6, {
    message: "Password must be at least 6 characters.",
  }),
});

type FormData = z.infer<typeof FormSchema>;

export default function LoginForm() {
  const router = useRouter();

  const form = useForm({
    resolver: zodResolver(FormSchema),
    defaultValues: {
      email: "",
      password: "",
    },
  });

  const onSubmit = async (data: FormData) => {
    console.log("Submitting form", data);

    const { email, password } = data;

    try {
      const response: any = await signIn("credentials", {
        email,
```



```

        password,
        redirect: false,
    });
    console.log({ response });
    if (!response?.error) {
        router.push("/");
        router.refresh();
    }

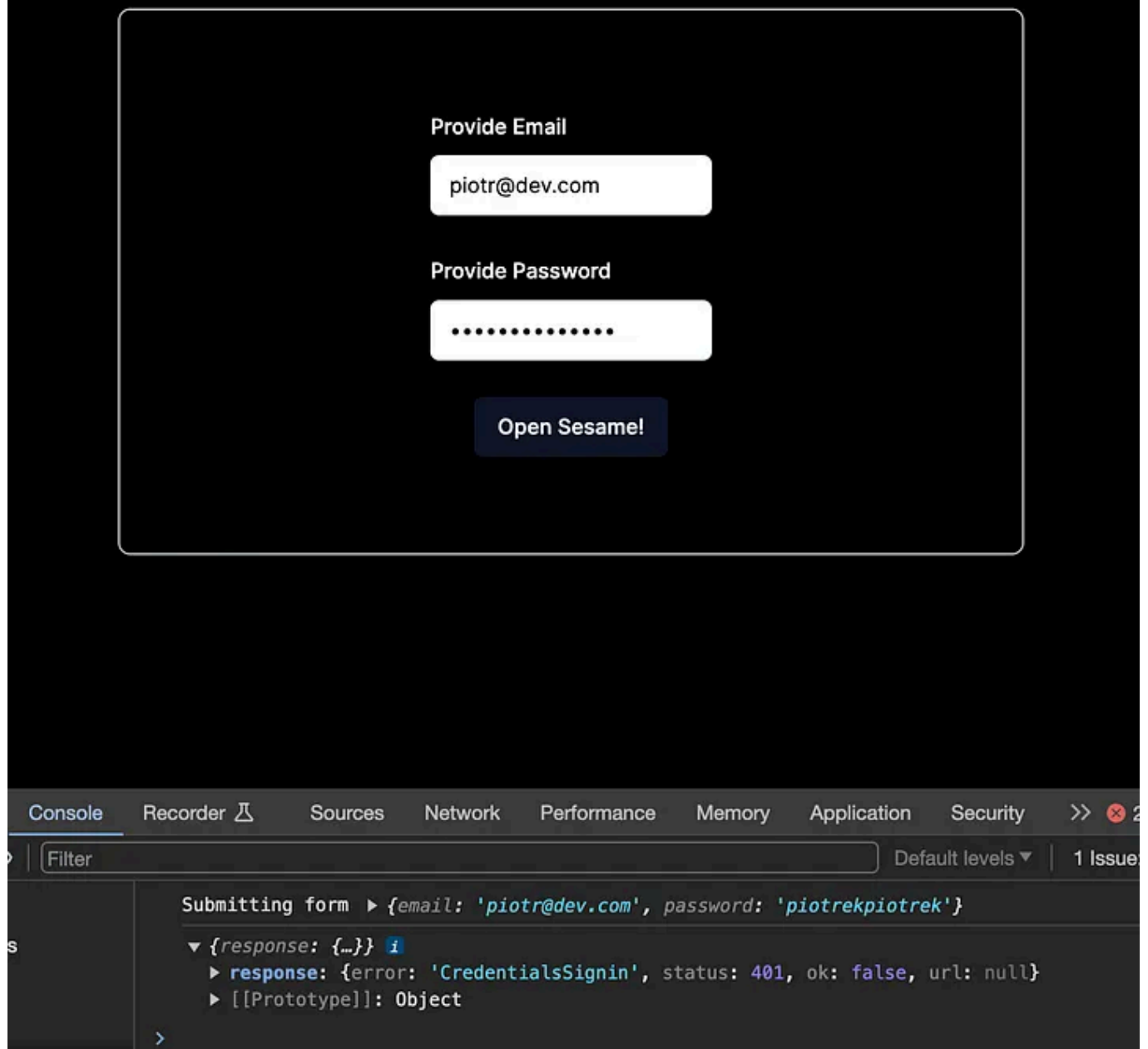
    if (!response.ok) {
        throw new Error("Network response was not ok");
    }
    // Process response here
    console.log("Login Successful", response);
    toast({ title: "Login Successful" });
} catch (error: any) {
    console.error("Login Failed:", error);
    toast({ title: "Login Failed", description: error.message });
}
};

return (
    <Form {...form} className="w-2/3 space-y-6">
        <form
            onSubmit={form.handleSubmit(onSubmit)}
            className="text-white p-4 md:p-16 border-[1.5px] rounded-lg border-gray-
        >
            <FormField
                control={form.control}
                name="email"
                render={({ field }) => (
                    <FormItem>
                        <FormLabel>Provide Email</FormLabel>
                        <FormControl>
                            <Input
                                className="text-black"
                                placeholder="Provide Email"
                                {...field}
                                type="text"
                            />
                        </FormControl>
                    </FormItem>
                )}
            />
            <FormField
                control={form.control}
                name="password"
                render={({ field }) => (
                    <FormItem>
                        <FormLabel>Provide Password</FormLabel>
                        <FormControl>
                            <Input
                                className="text-black"
                                placeholder="Has1o"
                                {...field}
                                type="password"
                            />

```

```
        </FormControl>
      </FormItem>
    )}
  />
  <Button
    type="submit"
    className="hover:scale-110 hover:bg-cyan-700"
    disabled={form.formState.isSubmitting}
  >
    {form.formState.isSubmitting ? "Opening..." : "Open Sesame!"}
  </Button>
</form>
</Form>
);
}
```

This should accurately display the entered email and password in the console:



We can now proceed to implement the authorization logic, which involves retrieving the user from the database, **dehashing** their password, and comparing it with the credentials provided. *For the purposes of this tutorial, I'll be using plain SQL queries, although I acknowledge the various advantages and disadvantages associated with this approach.*

```
import NextAuth from "next-auth/next";
import CredentialsProvider from "next-auth/providers/credentials";
import { sql } from "@vercel/postgres";
import { compare } from "bcrypt";

const handler = NextAuth({
  session: {
    strategy: "jwt",
```

```

    },

    pages: {
      signIn: "/login",
    },

    providers: [
      CredentialsProvider({
        // The name to display on the sign in form (e.g. 'Sign in with...')
        name: "Credentials",
        credentials: {
          email: {},
          password: {},
        },
        async authorize(credentials, req) {
          const response = await sql`
            SELECT * FROM users WHERE email=${credentials?.email}
          `;
          const user = response.rows[0];

          const passwordCorrect = await compare(
            credentials?.password || "",
            user.password
          );

          if (passwordCorrect) {
            return {
              id: user.id,
              email: user.email,
            };
          }

          console.log("credentials", credentials);
          return null;
        },
      }),
    ],
  });

  export { handler as GET, handler as POST };

```

Additionally, ensure to include `NEXTAUTH_URL` and `NEXTAUTH_SECRET` in your `.env` file. You can generate a secure `NEXTAUTH_SECRET` value by executing `openssl rand -base64 32` in your terminal.

```

NEXTAUTH_URL="http://localhost:3000"
NEXTAUTH_SECRET=password

```

## CONFIGURING RESTRICTED PAGES

To enforce access control on your pages, create a `middleware.ts` file in the root directory of your Next.js project:

```
export { default } from "next-auth/middleware";

export const config = {
  // specify the route you want to protect
  matcher: ["/"],
};
```

With this setup, your application is now configured to restrict access to specified pages, working seamlessly with the JWT strategy and redirection policy defined in your `[...nextauth].ts` configuration.

Happy coding :)

Piotr

Nextauth

Nextjs

React

Authentication

Full Stack



Written by PiotrDev

Follow

