Week 13.1

Building Medium

Up until now, our discussions have primarily revolved around theoretical concepts. In this lecture, Harkirat takes a practical approach by guiding us through the hands-on process of building a Medium like application

We'll be applying the knowledge we've gained so far, specifically focusing on implementing the frontend using React and the backend using Cloudflare Workers — creating a modern fullstack application.

While there are no specific notes provided for this section, a mini guide is outlined below to assist you in navigating through the process of building the application. Therefore, it is strongly advised to actively follow along during the lecture for a hands-on learning experience.

Step 1 — The stack

We'll be building medium in the following stack

1. React in the frontend

2. Cloudflare workers in the backend
3. zod as the validation library, type inference for the frontend types
4. Typescript as the language
5. Prisma as the ORM, with connection pooling
6. Postgres as the database
7. jwt for authentication (Cookies approach explained in the end as well)

Step 2 - Initialize the backend

Whenever you're building a project, usually the first thing you should do is initialise the project's backend.

Create a new folder called medium

mkdir medium

cd medium

Initialize a hono based cloudflare worker app

npm create hono@latest

Target directory › backend

Which template do you want to use? - cloudflare-workers

Do you want to install project dependencies? … yes Which package manager do you want to use? › npm (or yarn or bun, doesnt matter)

Reference https://hono.dev/top

Step 3 - Initialize handlers

To begin with, our backend will have 4 routes

1. POST /api/v1/signup
2. POST /api/v1/signin

3. POST /api/v1/blog
4. PUT /api/v1/blog
5. GET /api/v1/blog/:id

💡

https://hono.dev/api/routing

Solution

Step 4 - Initialize DB (prisma)

1. Get your connection url from neon.db or aieven.tech

postgres://avnadmin:password@host/db

2. Get connection pool URL from Prisma accelerate

https://www.prisma.io/data-platform/accelerate

prisma://accelerate.prisma-data.net/?api_key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhcGlfa2V5IjoiNTM2M2U5ZjEtNmNjMS00MWNkLWJiZTctN2U4NzFmMGFhZjJmIiwidGVuYW50X2lkIjoiY2I5OTE2NDk0MzFkNWZmZWRmNmFiYzViMGFlOTIwYzFhZDRjMGY5MTg1ZjZiNDY0OTc3MzgyN2IyMzY2OWIwMiIsImludGVybmFsX3NlY3JldCI6Ijc0NjE4YWY2LTA4NmItNDM0OC04MzIxLWMyMmY2NDEwOTExNyJ9.HXnE3vZjf8YH71uOollsvrV-TSe41770FPG_O8IaVgs

3. Initialize prisma in your project

Make sure you are in the backend folder

npm i prisma

npx prisma init

Replace DATABASE_URL in .env

DATABASE_URL="postgres://avnadmin:password@host/db"

Add DATABASE_URL as the connection pool url in wrangler.toml

name = "backend"

compatibility_date = "2023-12-01"

[vars]

DATABASE_URL = "prisma://accelerate.prisma-data.net/?api_key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhcGlfa2V5IjoiNTM2M2U5ZjEtNmNjMS00MWNkLWJiZTctN2U4NzFmMGFhZjJmIiwidGVuYW50X2lkIjoiY2I5OTE2NDk0MzFkNWZmZWRmNmFiYzViMGFlOTIwYzFhZDRjMGY5MTg1ZjZiNDY0OTc3MzgyN2IyMzY2OWIwMiIsImludGVybmFsX3NlY3JldCI6Ijc0NjE4YWY2LTA4NmItNDM0OC04MzIxLWMyMmY2NDEwOTExNyJ9.HXnE3vZjf8YH71uOollsvrV-TSe41770FPG_O8IaVgs"

You should not have your prod URL committed either in .env or in wrangler.toml to github wranger.toml should have a dev/local DB url .env should be in .gitignore

4. Initialize the schema

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id       String   @id @default(uuid())
  email    String   @unique
  name     String?
  password String
  posts    Post[]
}

model Post {
  id       String   @id @default(uuid())
  title    String
```

```
  content   String
  published Boolean  @default(false)
  author    User     @relation(fields: [authorId], references: [id])
  authorId  String
}
```

5. Migrate your database

```
npx prisma migrate dev --name init_schema
```

You might face issues here, try changing your wifi if that happens

6. Generate the prisma client

```
npx prisma generate --no-engine
```

7. Add the accelerate extension

```
npm install @prisma/extension-accelerate
```

8. Initialize the prisma client

```
import { PrismaClient } from '@prisma/client/edge'
import { withAccelerate } from '@prisma/extension-accelerate'


const prisma = new PrismaClient({
   datasourceUrl: env.DATABASE_URL,
}).$extends(withAccelerate())
```

Step 5 - Create routes

1. Simple Signup route

Add the logic to insert data to the DB, and if an error is thrown, tell the user about it

Solution

To get the right types on c.env, when initializing the Hono app, pass the types of env as a generic

```
const app = new Hono<{
        Bindings: {
                DATABASE_URL: string
        }
}>();
```

Ideally you shouldn't store passwords in plaintext. You should hash before storing them. More details on how you can do that - https://community.cloudflare.com/t/options-for-password-hashing/138077https://developers.cloudflare.com/workers/runtime-apis/web-crypto/

2. Add JWT to signup route

Also add the logic to return the user a jwt when their user id encoded. This would also involve adding a new env variable JWT_SECRET to wrangler.toml

Use jwt provided by hono - https://hono.dev/helpers/jwt

Solution

3. Add a signin route

Solution

Step 6 - Middlewares

Creating a middleware in hono is well documented - https://hono.dev/guides/middleware

1. Limiting the middleware

To restrict a middleware to certain routes, you can use the following -

```
app.use('/message/*', async (c, next) => {
  await next()
})
```

In our case, the following routes need to be protected -

```
app.get('/api/v1/blog/:id', (c) => {})

app.post('/api/v1/blog', (c) => {})

app.put('/api/v1/blog', (c) => {})
```

So we can add a top level middleware

```
app.use('/api/v1/blog/*', async (c, next) => {
  await next()
})
```

2. Writing the middleware

Write the logic that extracts the user id and passes it over to the main route.

How to pass data from middleware to the route handler?

How to make sure the types of variables that are being passed is correct?

Solution

3. Confirm that the user is able to access authenticated routes

app.post('/api/v1/blog', (c) => {

console.log(c.get('userId'));

return c.text('signin route')

})

Send the Header from Postman and ensure that the user id gets logged on the server

💡

If you want, you can extract the prisma variable in a global middleware that set's it on the context variable

```
app.use("*", (c) => {
        const prisma = new PrismaClient({
    datasourceUrl: c.env.DATABASE_URL,
 }).$extends(withAccelerate());
 c.set("prisma", prisma);
})
```

Ref https://stackoverflow.com/questions/75554786/use-cloudflare-worker-env-outside-fetch-scope

Step 7 - Blog routes and better routing

Better routing

https://hono.dev/api/routing#grouping

Hono let's you group routes together so you can have a cleaner file structure.

Create two new files -

routes/user.ts

routes/blog.ts and push the user routes to user.ts

index.ts

user.ts

Blog routes

1. Create the route to initialize a blog/post

Solution

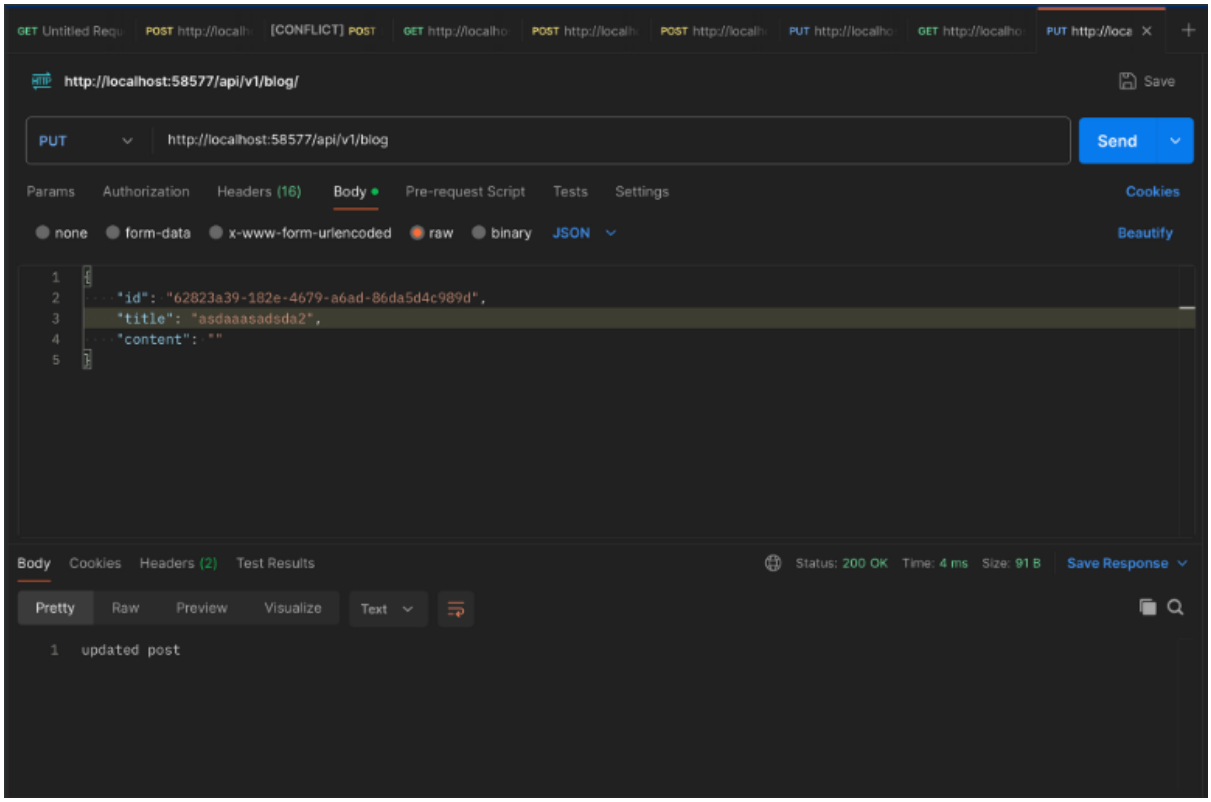
2. Create the route to update blog

Solution

3. Create the route to get a blog

Solution

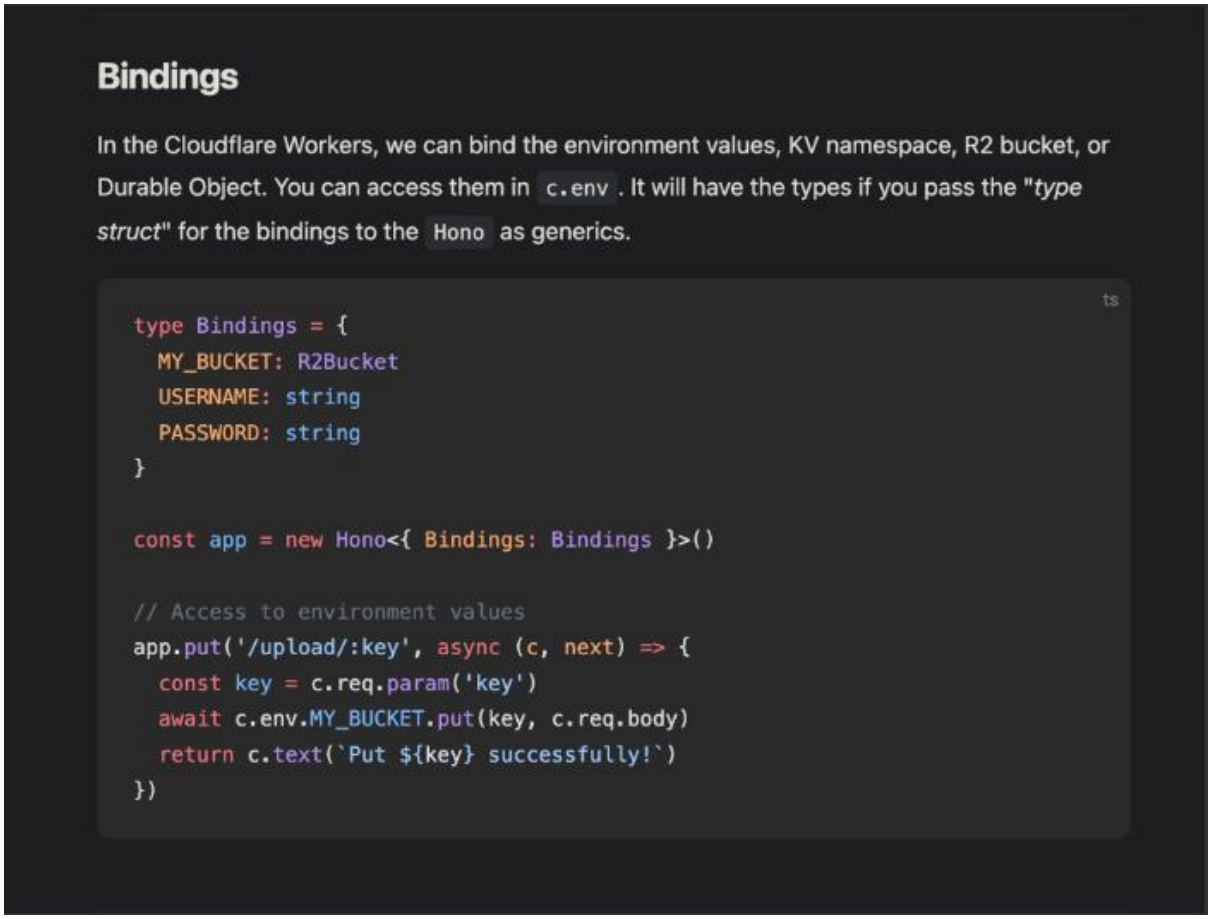
Try to hit the routes via POSTMAN and ensure they work as expected



Step 8 - Understanding the types

Bindings

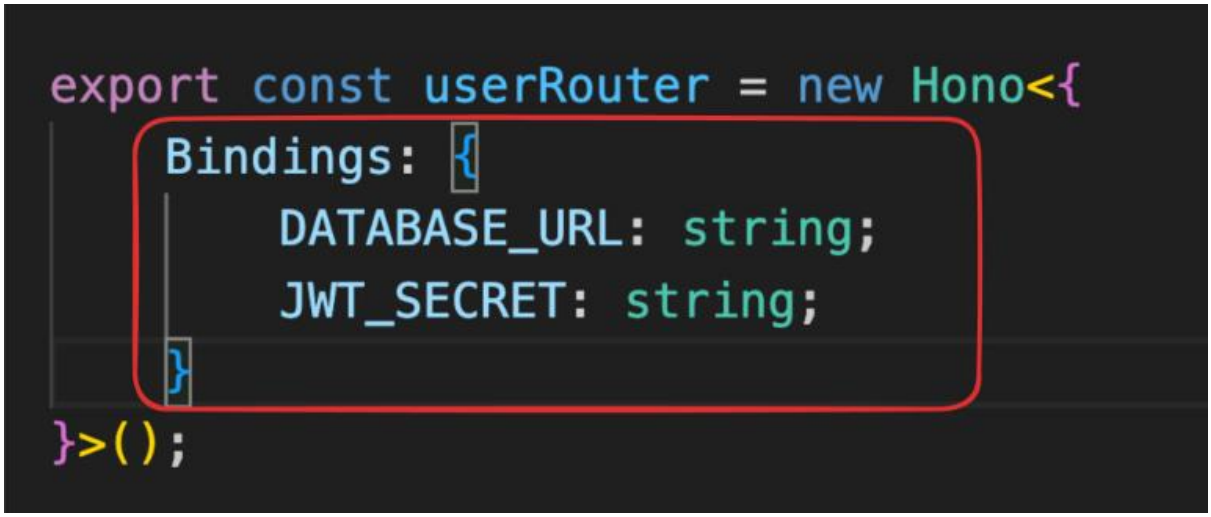
https://hono.dev/getting-started/cloudflare-workers#bindings



In our case, we need 2 env variables -

JWT_SECRET

DATABASE_URL



Variables

https://hono.dev/api/context#var

If you want to get and set values on the context of the request, you can use c.get and c.set

```
bookRouter.use(async (c, next) => {
    // check if the jwt is value
    c.set('userId', "jwt");
    await next()
});
```

You need to make typescript aware of the variables that you will be setting on the context.

```
export const bookRouter = new Hono<{
    Bindings: {
        DATABASE_URL: string;
        JWT_SECRET: string;
    },
    Variables: {
        userId: string
    }
}>();
```

💡

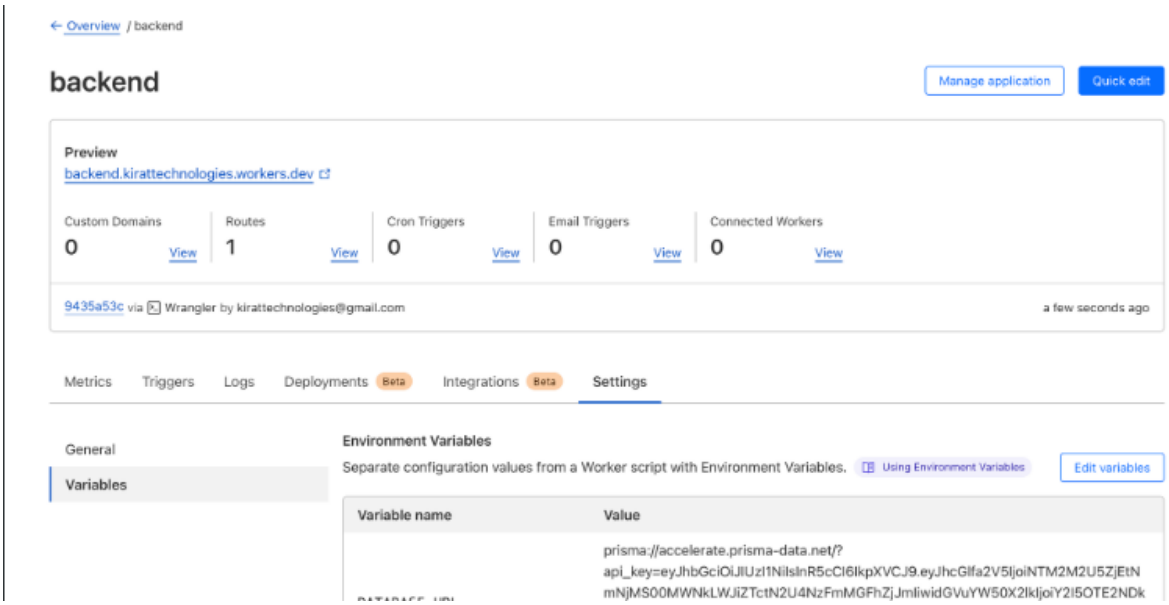
You can also create a middleware that sets prisma in the context so you don't need to initialise it in the function body again and again

Step 9 - Deploy your app

npm run deploy

Make sure you have logged in the cloudflare cli using npx wrangler login

Update the env variables from cloudflare dashboard



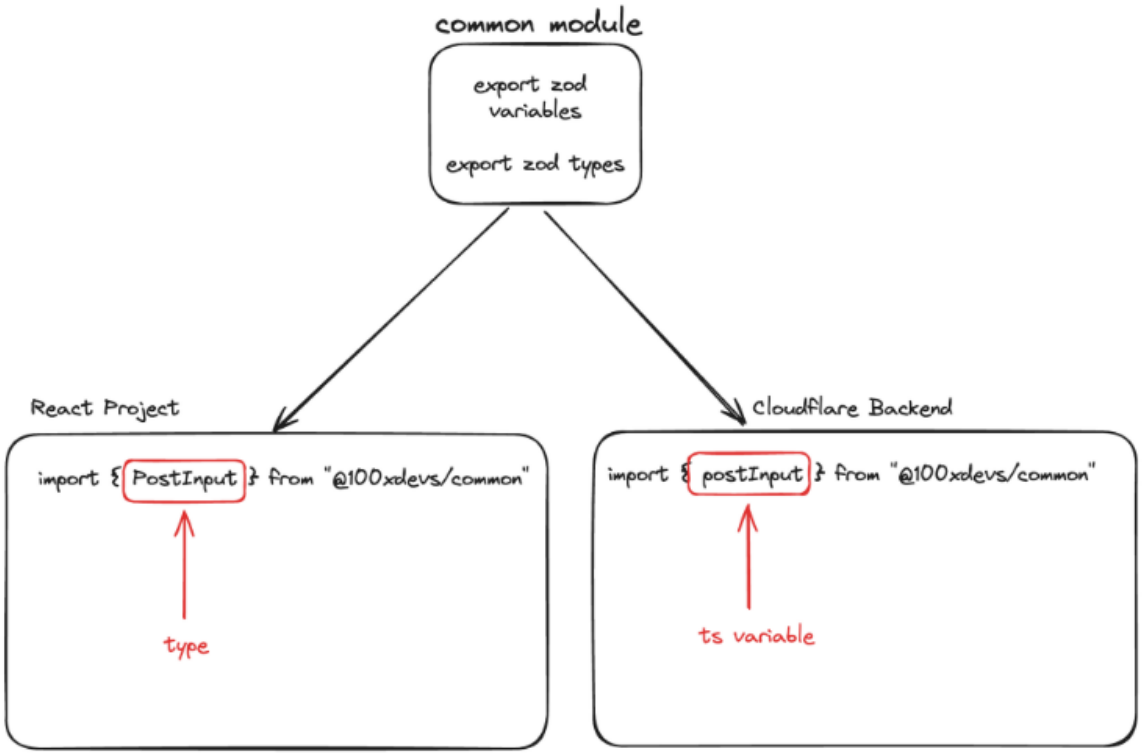
Test your production URL in postman, make sure it works

Step 10 - Zod validation

If you've gone through the video Cohort 1 - Deploying npm packages, Intro to Monorepos, you'll notice we introduced type inference in Zod

https://zod.dev/?id=type-inference

This let's you get types from runtime zod variables that you can use on your frontend



We will divide our project into 3 parts

1. Backend
2. Frontend
3. common

common will contain all the things that frontend and backend want to share. We will make common an independent npm module for now. Eventually, we will see how monorepos make it easier to have multiple packages sharing code in the same repo

Step 11 - Initialise common

1. Create a new folder called common and initialize an empty ts project in it

mkdir common

cd common

npm init -y

npx tsc --init

1. Update tsconfig.json

"rootDir": "./src",

"outDir": "./dist",

"declaration": true,

1. Sign up/login to npmjs.org
2. Run npm login
3. Update the name in package.json to be in your own npm namespace, Update main to be dist/index.js

```
{
 "name": "@100xdevs/common-app",
 "version": "1.0.0",
 "description": "",
        "main": "dist/index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [],
 "author": "",
 "license": "ISC"
}
```

1. Add src to .npmignore
2. Install zod

npm i zod

1. Put all types in src/index.ts
    1. signupInput / SignupInput
    2. signinInput / SigninInput
    3. createPostInput / CreatePostInput
    4. updatePostInput / UpdatePostInput

Solution

1. tsc -b to generate the output
2. Publish to npm

npm publish --access public

1. Explore your package on npmjs

Step 12 - Import zod in backend

1. Go to the backend folder cd backend 1. Install the package you published to npm npm i your_package_name 1. Explore the package cd node_modules/your_package_name 1. Update the routes to do zod validation on them

Solution

Step 13 - Init the FE project

1. Initialise a react app

npm create vite@latest

1. Initialise tailwind https://tailwindcss.com/docs/guides/vite

npm install -D tailwindcss postcss autoprefixer

npx tailwindcss init -p

1. Update tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

1. Update index.css

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

1. Empty up App.css
2. Install your package

```
npm i your_package
```

1. Run the project locally

```
npm run dev
```

Step 14 - Add react-router-dom

1. Add react-router-dom npm i react-router-dom

2. Add routing (ensure you create the Signup, Signin and Blog components)

```
import { BrowserRouter, Route, Routes } from 'react-router-dom'
import { Signup } from './pages/Signup'
import { Signin } from './pages/Signin'
import { Blog } from './pages/Blog'

function App() {

  return (
    <>
      <BrowserRouter>
        <Routes>
          <Route path="/signup" element={<Signup />} />
          <Route path="/signin" element={<Signin />} />
          <Route path="/blog/:id" element={<Blog />} />
        </Routes>
```

```
      </BrowserRouter>
    </>
  )
}


export default App
```

3. Make sure you can import types from your_package