Medium

(https://projects.100xdevs.com/tracks/blog/blog-1)

(https://github.com/nthapa000/MediumBlog)

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

Initialize the backend

Hono is similar like express.

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

Initialize a hono based cloudflare worker app

npm create hono@latest

Target directory › backend

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

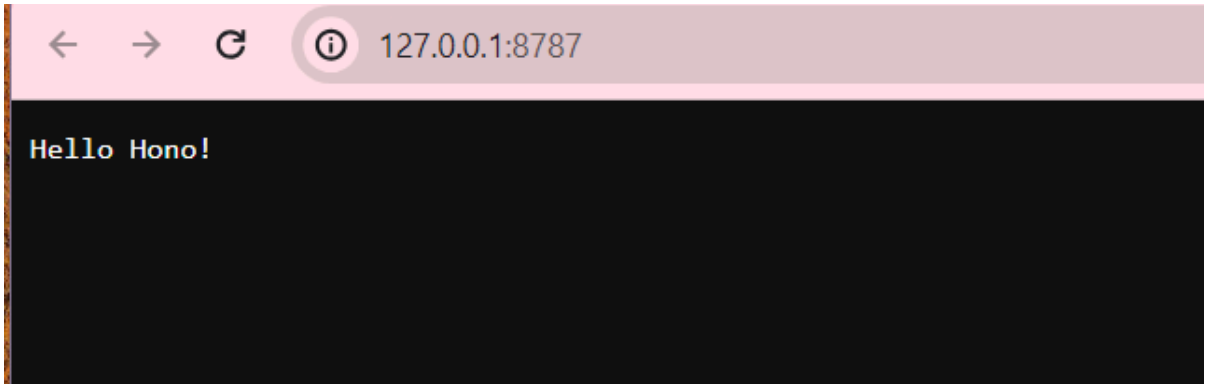
Do you want to install project dependencies? … yes

Which package manager do you want to use? › npm (or yarn or bun, doesnt matter)

```
C:\Users\NTC\Desktop\Dev\Week13\MediumBlog>npm create hono@latest
Need to install the following packages:
  create-hono@0.6.2
Ok to proceed? (y) y
create-hono version 0.6.2
? Target directory backend
? Which template do you want to use?
cloudflare-workers
✓ Cloning the template
? Do you want to install project dependencies?
yes
? Which package manager do you want to use? npm
✓ Installing project dependencies
🎉 Copied project files
Get started with: cd backend
```

Reference: https://hono.dev/top

```
C:\Users\NTC\Desktop\Dev\Week13\MediumBlog\backend>n
pm run dev

> dev
> wrangler dev src/index.ts

 ⛅️ wrangler 3.41.0
------------------
○ Starting local server...
[wrangler:inf] Ready on http://127.0.0.1:8787
[wrangler:inf] GET / 200 OK (38ms)
[wrangler:inf] GET /favicon.ico 404 Not Found (6ms)
┌──────────────┬──────────────┬──────────────┬──────────────┬──────────────┐
│ [ open a     │ [ open       │ [l turn off local │ [ clear      │ [x to        │
│ ]browser,    │ ]Devtools,   │ mode,        │ ]console,    │ exit         │
└──────────────┴──────────────┴──────────────┴──────────────┴──────────────┘
```



Initialize handlers

To begin with, our backend will have 4 routes

1. POST /api/v1/user/signup
2. POST /api/v1/user/signin
3. POST /api/v1/blog
4. PUT /api/v1/blog
5. GET /api/v1/blog/:id
6. GET /api/v1/blog/bulk

```
import { Hono } from 'hono'

const app = new Hono()

app.post('/api/v1/signup', (c) => {
  return c.text('Sign up!')
})
app.post('/api/v1/signin', (c) => {
  return c.text('Sign in!')
})
app.post('/api/v1/blog', (c) => {
  return c.text('post blog!')
})
app.put('/api/v1/blog', (c) => {
  return c.text('Hello Hono!')
```

```
})
app.get('/api/v1/blog/:id', (c) => {
  return c.text('Hello Hono!')
})
// dynamic parameter

export default app
```

Initialize DB (prisma)

1. Get your connection url from neon.db

```
postgresql://neondb_owner:xXcR4nuM3eKm@ep-soft-snow-a5i13jlo.us-east-2.aws.neon.tech/?sslmode=require
```

2. Get connection pool URL from Prisma Accelerate

There could be 100th of workers trying to connect to the database

Hence we connect via connection pool

```
DATABASE_URL="prisma://accelerate.prisma-data.net/?api_key=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhcGlfa2V5IjoiMDM1OWMxYTctYTFmZS00NWY5LWFkYjItZjVmNmRjMmU4ZDg0IiwidGVuYW50X2lkIjoiZGVlYTkyZDBkNDI5MGQzZjE2YjI4YTczY2E5MDAwMGI0YmIwNzM1Njg5Y2U5Njc1MTNjMGVhNzE2NmFlMDJhZiIsImludGVybmFsX3NlY3JldCI6ImQ4NTNlNDRmLTM2YTEtNDQyYy1hYTYzLWJhMWU2ZGEzMDVkMiJ9.y5gmGgqfTG85bDA5vk8T2zXjAaIjs9qcNnsaJan-PKk"
```

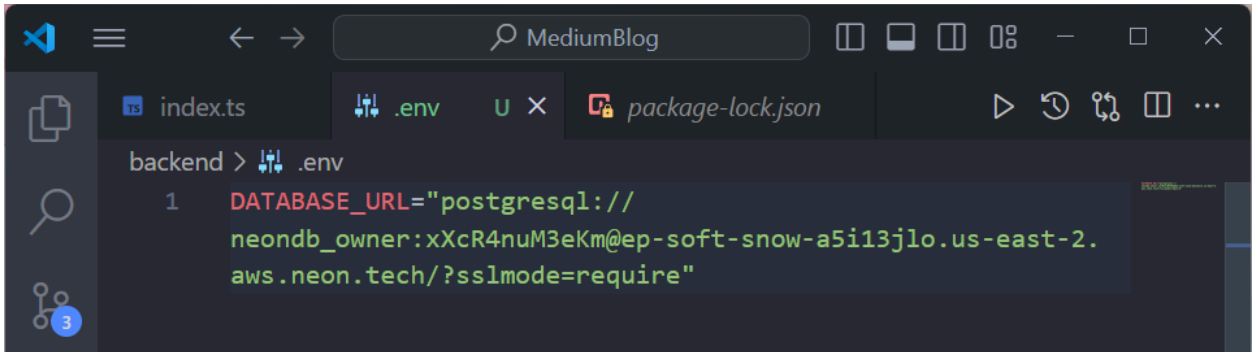
3. Initialize prisma in your project

npm i prisma

npx prisma init

Replace **DATABASE_URL** in **.env**

In .env file we put the real database url as it is connected to the cli and we want cli to directly connect to the database

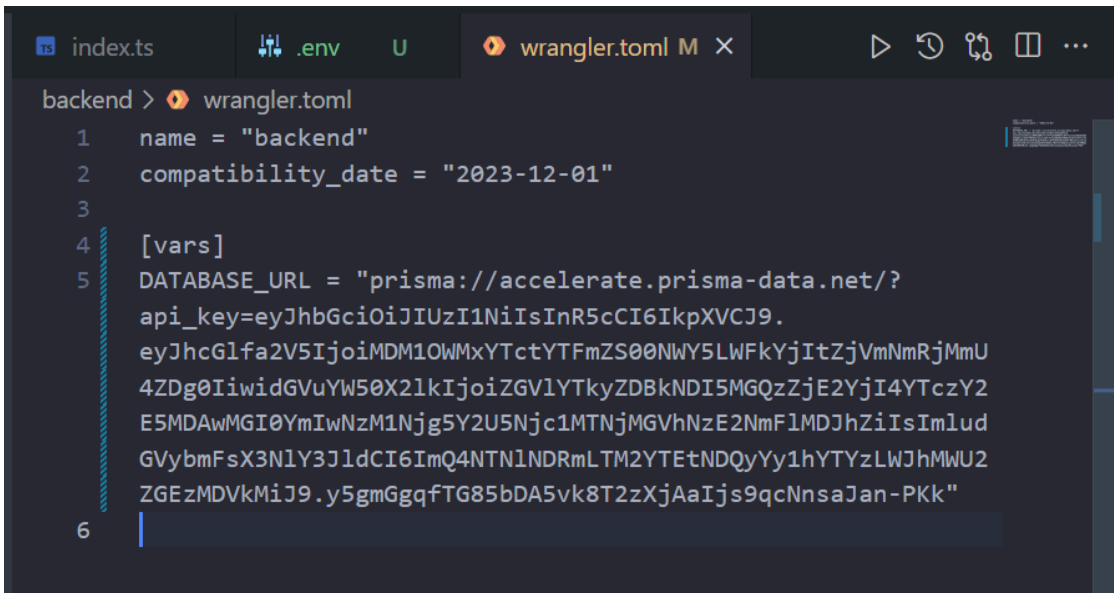


Add **DATABASE_URL** as the **connection pool** url in **wrangler.toml**

Whenever we do npm run dev environment variable will be picked from wrangler.toml

Backend application will pick env variable from it

env.DATABASE_URL from wrangle.toml



4. Initialize the schema

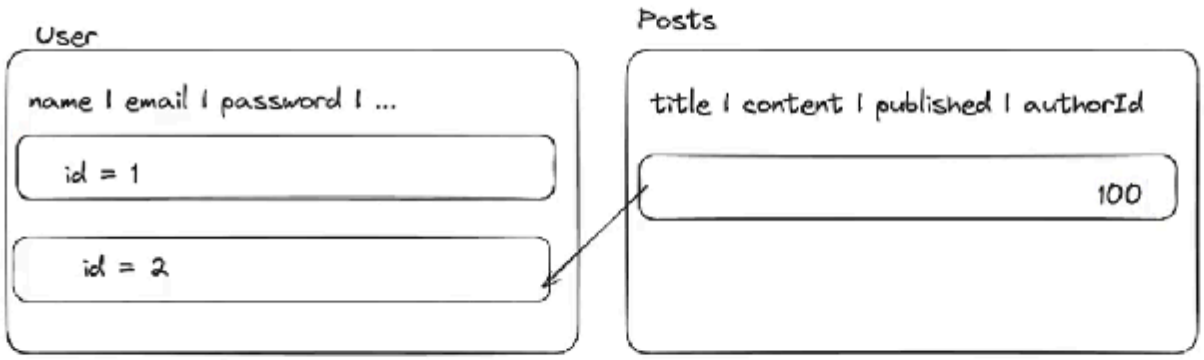
We will have two table in this DB User and Post and they will be connected somehow.

Very low probability for two uuid to be same

```
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[]
//array of 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
}
```

Enforcing a constraint , Hence this input is invalid since there is no author with id = 100 in User



5. Migrate your Database

npx prisma migrate dev --name init_schema

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())
```

We have to specify the types of environment variables

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

To get the right types on c.env, when initializing the Hono app, pass the types of env as 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/138077

https://developers.cloudflare.com/workers/runtime-apis/web-crypto/

We can understand these things by clicking on Hono and then seeing its syntax etc

```
const app = new Hono
```

——————

We can ignore typescript error by doing

@ts-ignore

We say typescript ignore the next line it consist some error

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

app.post('/api/v1/signup',async (c) => {
  // c is basically context has all our request and response and
environment variable etc
  // specify our env variable type
  // we can't give environment variable outside
  // we don't have global access in serverless
  const prisma = new PrismaClient({
    datasourceUrl: c.env.DATABASE_URL,
}).$extends(withAccelerate())
// .env  accessed from the wrangler.toml
// Cloudflare picks environment variable from wrangler.toml

// this is how we get the body in Hono
// we have to await when it is converted to json
  const body = await c.req.json();
  await prisma.user.create({
    data:{
      email:body.email,
      password:body.email
```

```
    },
  })

  return c.text('Sign up!')
})
```

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

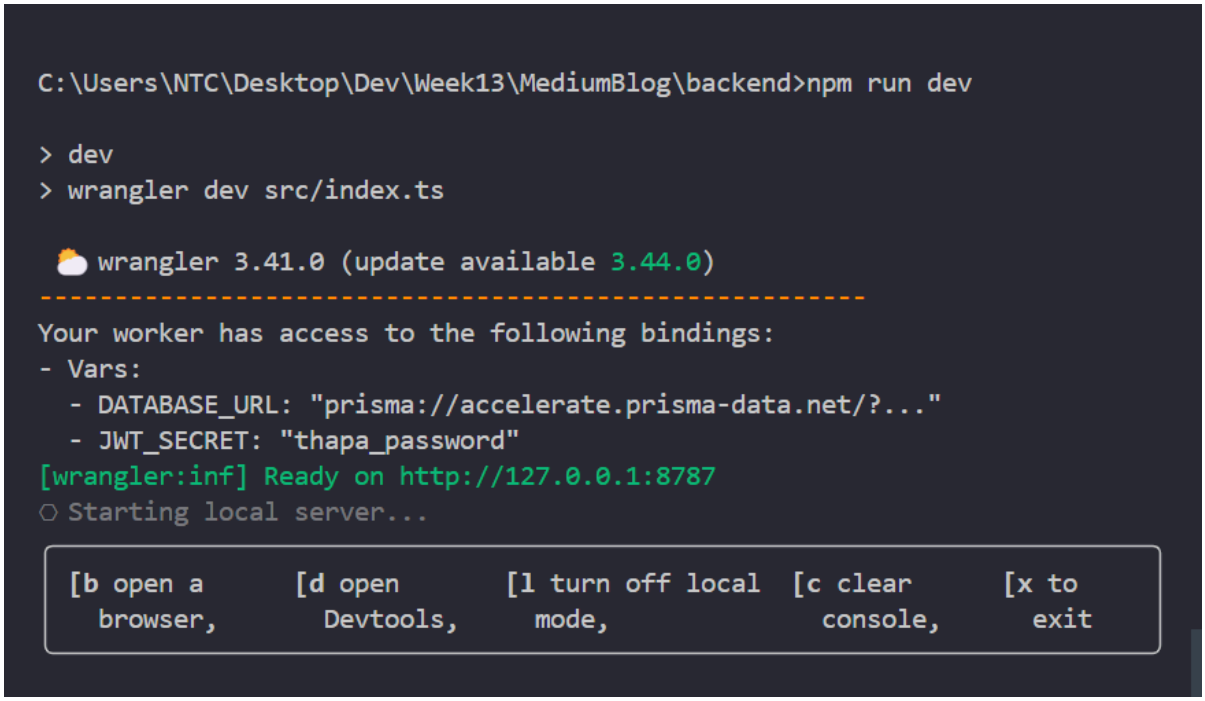
We have to use jwt provide by hono https://hono.dev/helpers/jwt as there is high chances that cloudflare doesn't support jsonwebtoken
import { decode, sign, verify } from 'hono/jwt'

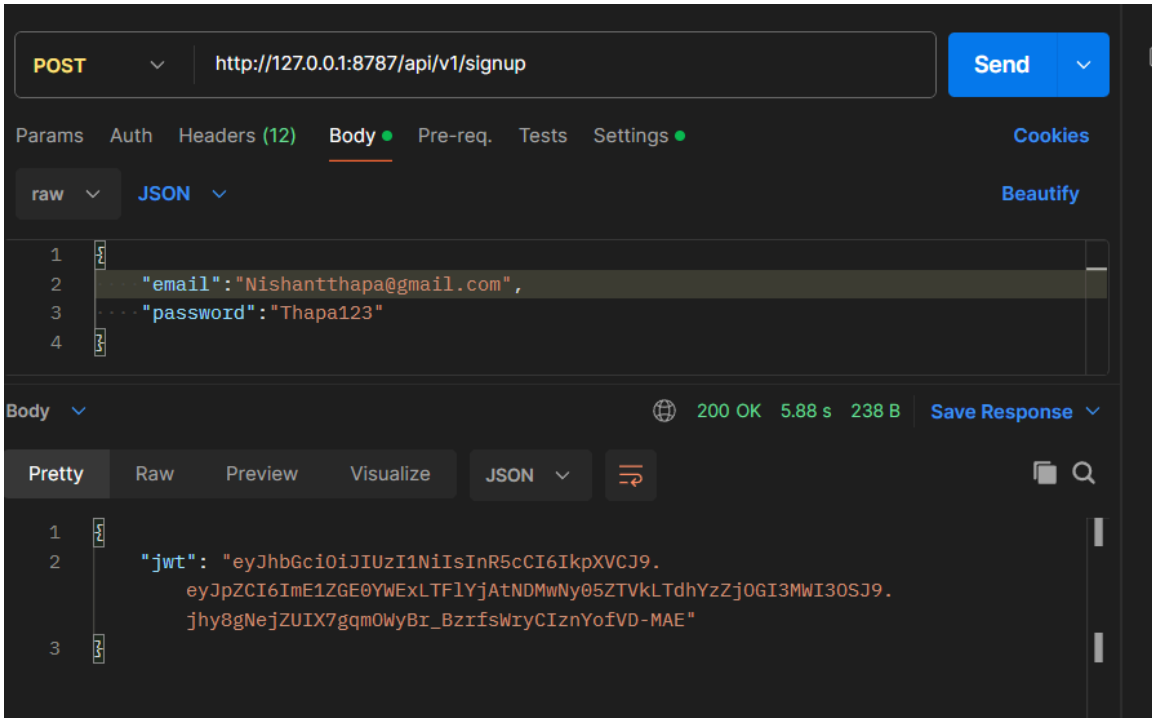
```
app.post('/api/v1/signup',async (c) => {
  const prisma = new PrismaClient({
    datasourceUrl: c.env.DATABASE_URL,
}).$extends(withAccelerate())
  const body = await c.req.json();
  // no need to check whether user already exist because we have mentioned
in schema.prisma , we have check at database level
  // we dont have to do
  // const userExist = prisma.user.find()
  try{
  const user = await prisma.user.create({
    data:{
      email:body.email,
      password:body.password,
      name:body.name
    },
  })
```

```
  // Authentications , whole point of signup need to do is entry in
database and second thing sign up route need to do is return the user a
JWT, user can send in local storage and eventually send in every request
  const token =await sign({id:user.id},c.env.JWT_SECRET)
  return c.json({jwt: token})
}catch(e){
  c.status(411)
  return c.text('Invalid')
}
})
```

Now lets run locally in POSTMAN

npm run dev





JWT verification

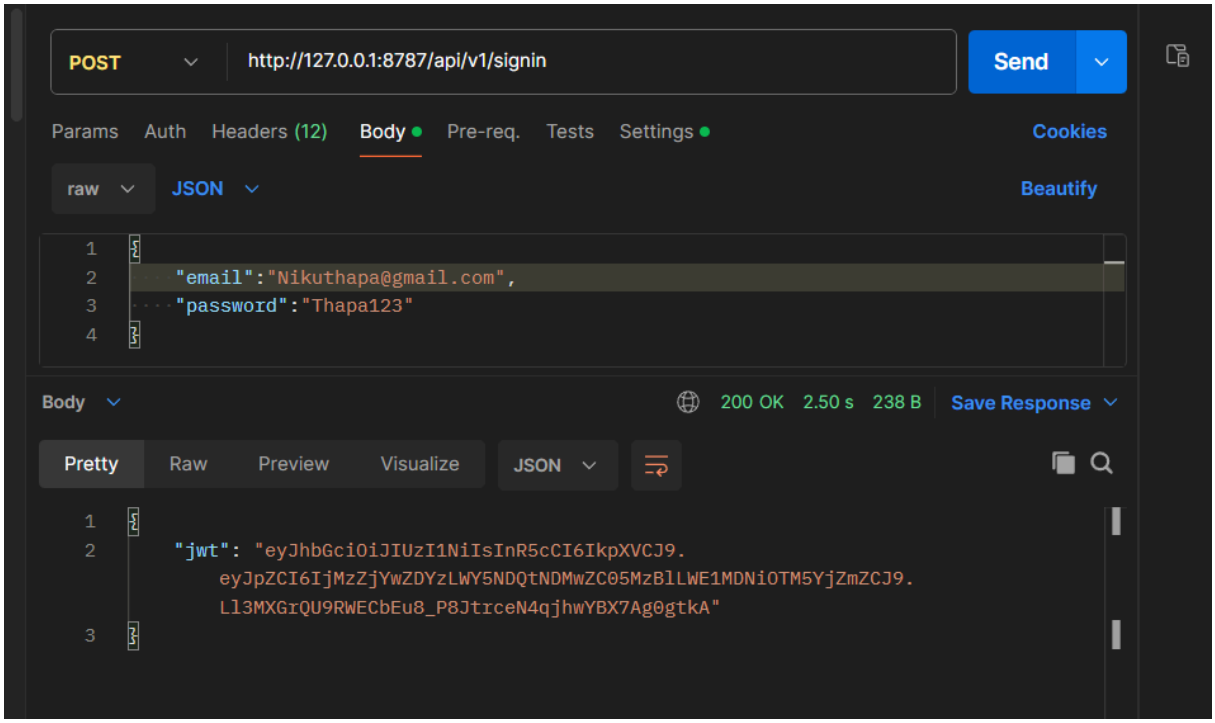
```
  const token =await sign({
    id:user.id
  },c.env.JWT_SECRET)
  return c.json({jwt: token})
```

3. Add a signin route

```
app.post('/api/v1/signin', async (c) => {
  const prisma = new PrismaClient({
    datasourceUrl: c.env?.DATABASE_URL   ,
  }).$extends(withAccelerate());

  const body = await c.req.json();
  const user = await prisma.user.findUnique({
    where: {
      email: body.email
    }
  });

  if (!user) {
    c.status(403);
```

```
    return c.json({ error: "user not found" });
  }

  const jwt = await sign({ id: user.id }, c.env.JWT_SECRET);
  return c.json({ jwt });
})
```

JWT is returned when we send correct email and password.



Middleware

Middleware in Hono https://hono.dev/guides/middleware

We can put more thing inside the context variable

set() and get() ,

Pass data from middleware to the routes

1. **Limiting the middleware**

To restrict the middleware to certain routes , we can use following

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

For posting a blog and updating a blog we need a middleware to extract the details of the user , does authentication check
Middleware is run before the request

So we can add a top-level middleware.

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

Blogs needed to be protected

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

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

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

Before control reaches

```
(c) => {}
```

We want an authentication middleware to do some authentication checks

```


app.use('/api/v1/blog/*', async (c, next) => {
  // get the header
  // verify the header
  // if the header is correct , we can proceed
  // if not, we return the user a 403 status code
  await next()
})
```

‘Backend pick url from wrangler.toml and CLI will pick from .env
When we run npx prisma migrate dev , it uses real postgres url , it doesn’t support accelerate url.

The Env variable is not accessible in index.ts globally hence we need to initialize prisma in each route to access the env variable

```
app.post('/api/v1/signup',async (c) => {
  const prisma = new PrismaClient({
    datasourceUrl: c.env.DATABASE_URL,
}).$extends(withAccelerate())
  const body = await c.req.json();
  const user = await prisma.user.create({
    data:{
      email:body.email,
      password:body.email
    },
  })
  const token =await sign({id:user.id},c.env.JWT_SECRET)
  return c.json({jwt: token})
})
```

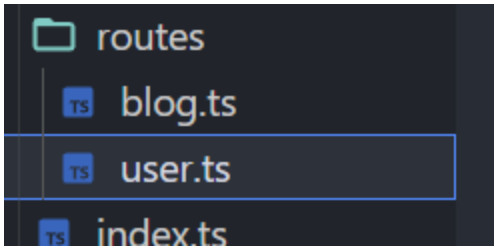
c variable give access to the env variable

Way to debug is doing logging

status (403) return when we have unauthorized access

Lets create a better route

Divide the index.js file into two folder



blog.ts

```ts
import { Hono } from "hono";
export const blogRouter = new Hono<{
    // passing the generics
      Bindings: {
        DATABASE_URL: string;
        JWT_SECRET:string;
      }
  }>();

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

  blogRouter.post('/api/v1/blog', (c) => {
    return c.text('post blog!')
  })
  blogRouter.put('/api/v1/blog', (c) => {
    return c.text('Hello Hono!')
  })
  blogRouter.get('/api/v1/blog/:id', (c) => {
    return c.text('Hello Hono!')
  })
```

user.ts

```ts
import { Hono } from "hono";
import { PrismaClient } from '@prisma/client/edge'
import { withAccelerate } from '@prisma/extension-accelerate'
import {sign, verify} from 'hono/jwt'

export const userRouter = new Hono<{
```

```
    // typescript doesn't understand wrangler.toml code
    // passing the generics
      Bindings: {
        DATABASE_URL: string;
        JWT_SECRET:string;
      }
  }>();


userRouter.post('/signup',async (c) => {
    const prisma = new PrismaClient({
      datasourceUrl: c.env.DATABASE_URL,
  }).$extends(withAccelerate())
    const body = await c.req.json();
    // no need to check whether user already exist because we have
mentioned in schema.prisma , we have check at database level
    // we dont have to do
    // const userExist = prisma.user.find()
    try{
    const user = await prisma.user.create({
      data:{
        email:body.email,
        password:body.password,
        name:body.name
      },
    })
    // Authentications , whole point of signup need to do is entry in
database and second thing sign up route need to do is return the user a
JWT, user can send in local storage and eventually send in every request
    const token =await sign({
      id:user.id
    },c.env.JWT_SECRET)
    return c.json({jwt: token})
  }catch(e){
    c.status(411)
    return c.text('Invalid')
  }
  })

  userRouter.post('/signin', async (c) => {
```

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

    const body = await c.req.json();
    const user = await prisma.user.findUnique({
        where: {
            email: body.email,
      password: body.password
        }
    });

    if (!user) {
        c.status(403);
        return c.json({ error: "user not found" });
    }

    const jwt = await sign({ id: user.id }, c.env.JWT_SECRET);
    return c.json({ jwt });
})
```

Now lets write the blog routes

Posting the blog

```
blogRouter.post('/',async (c) => {
    const body = await c.req.json();
    const prisma = new PrismaClient({
        datasourceUrl: c.env.DATABASE_URL,
    }).$extends(withAccelerate())


    const blog = await prisma.post.create({
        data:{
            title: body.title,
            content:body.content,
            authorId:"1"
            // the extraction will occur in the middleware
        }
```

```
  })

  return c.json({
      id:blog.id
  })
})
```

Put request to allow user to update title and content

```
blogRouter.put('/',async (c) => {
  const body = await c.req.json();
  const prisma = new PrismaClient({
      datasourceUrl: c.env.DATABASE_URL,
  }).$extends(withAccelerate())

  const blog = await prisma.post.update({
      where: {
          id:body.id
      },
      data:{
          title: body.title,
          content:body.content,
          // the extraction will occur in the middleware
      }
  })

  return c.json({
      id:blog.id
  })
})
```

Getting a blog

```
blogRouter.get('/:id', async (c) => {
  const body = await c.req.json();
```

```
    const prisma = new PrismaClient({
        datasourceUrl: c.env.DATABASE_URL,
    }).$extends(withAccelerate())
    try{
        // must be wrapped because this request can fail due to multiple
reasons
        const blog = await prisma.post.findFirst({
            where: {
                id:body.id
            }
        })
        return c.json({
            blog
        })
    }catch(e){
        c.status(411);
        return c.json({
            message:"Error while fetching blog post"
        })
    }


  })
```

Bulk getting the blog ,just there title

```
// Ideally we should add pagination meaning that we should not add all the
post , suppose we return only 10 and user ask for more when it scroll
blogRouter.get("/bulk", async (c) => {
  const prisma = new PrismaClient({
    datasourceUrl: c.env.DATABASE_URL,
  }).$extends(withAccelerate());
  const blogs = prisma.post.findMany();
  //getting all the blogs
  return c.json({
    blogs
  })
});
```

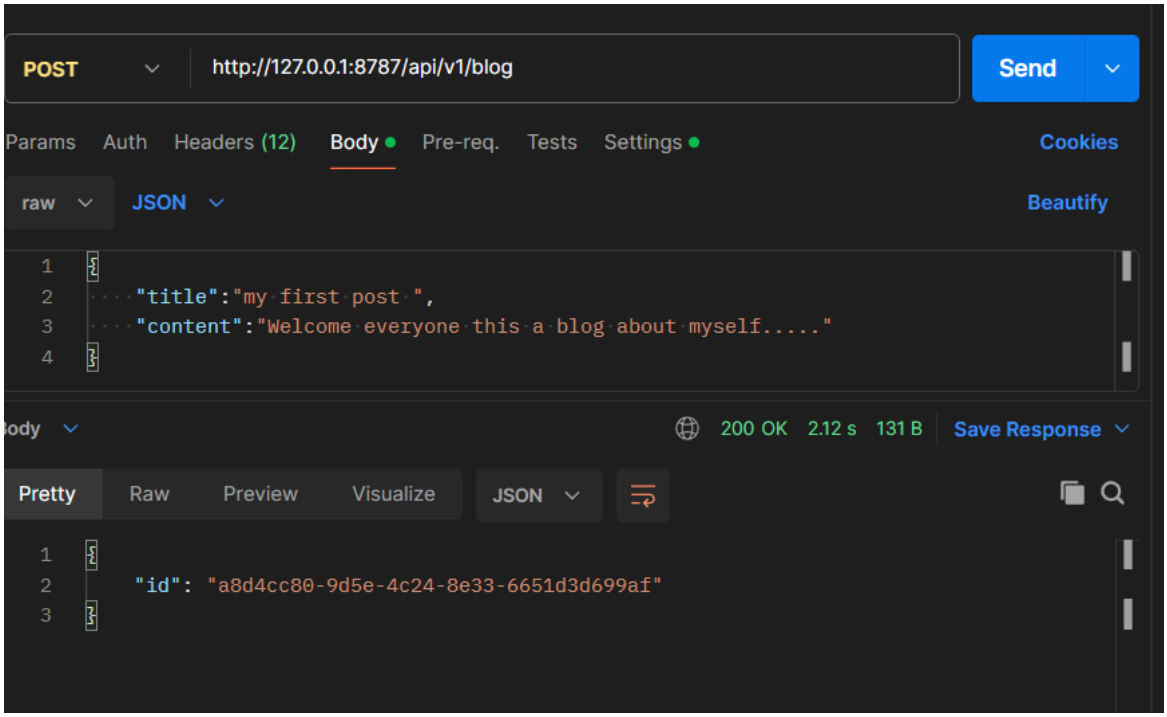
Middleware

```
export const blogRouter = new Hono<{
  // passing the generics
  Bindings: {
    DATABASE_URL: string;
    JWT_SECRET: string;
  },
  Variables:{
    userId: string;
  }
}>();


blogRouter.use('/*', async (c, next) => {
    // get the header
    // verify the header
    // if the header is correct , we can proceed
    // if not, we return the user a 403 status code
    const header = c.req.header("authorization") || "";
    //if this is undefined then we want to set the type of header as
string or typescript will raise an error

    // auth header
    // Bearer token(format of the token)
    // ["Bearer","token"]
    const token = header.split(" ")[1]

    // decode is similar to decode a jwt and give orginal jwt
    // but verification can be done only with the secret_password
    const user =await verify(token,c.env.JWT_SECRET)
    if(user){
        // we can also do bad solution @ts-ignore
        c.set("userId",user.id)
        // context or c doesn't have userId as a key , hence we again need
to modify the Bindings
      next()
```

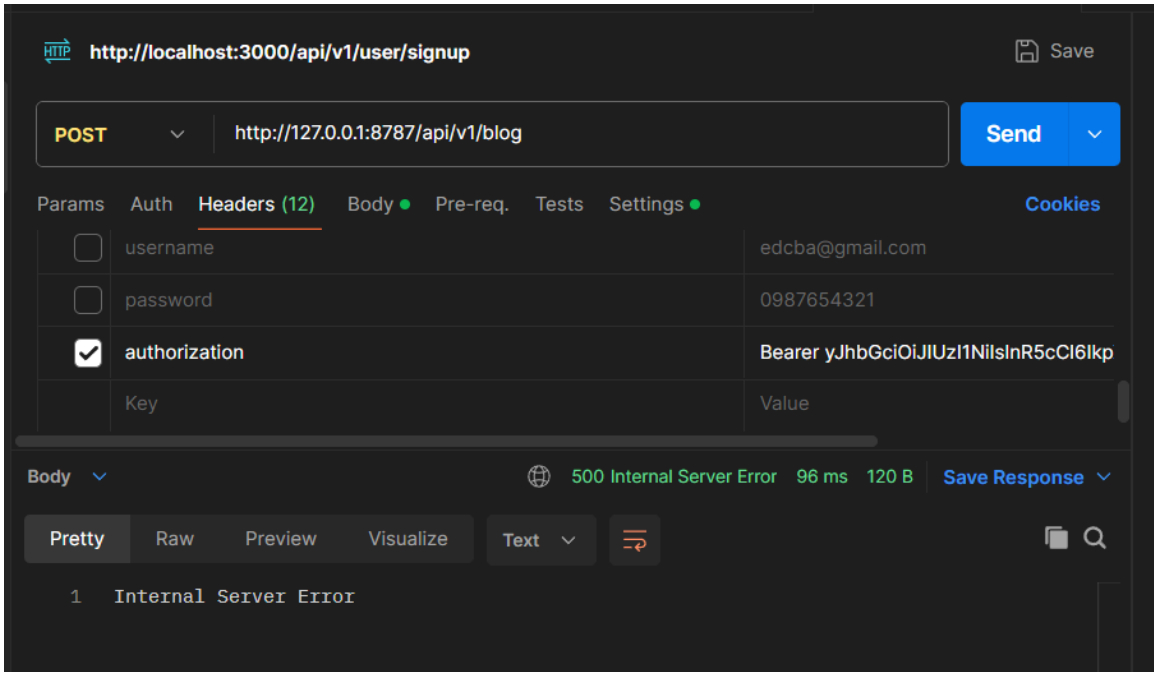
```
  }else{
    c.status(403)
    return c.json({error:"unauthorized"})
  }
})
```

Testing on the POSTMAN

If connection pool has not been hit it will take a little time at the beginning to warm up.

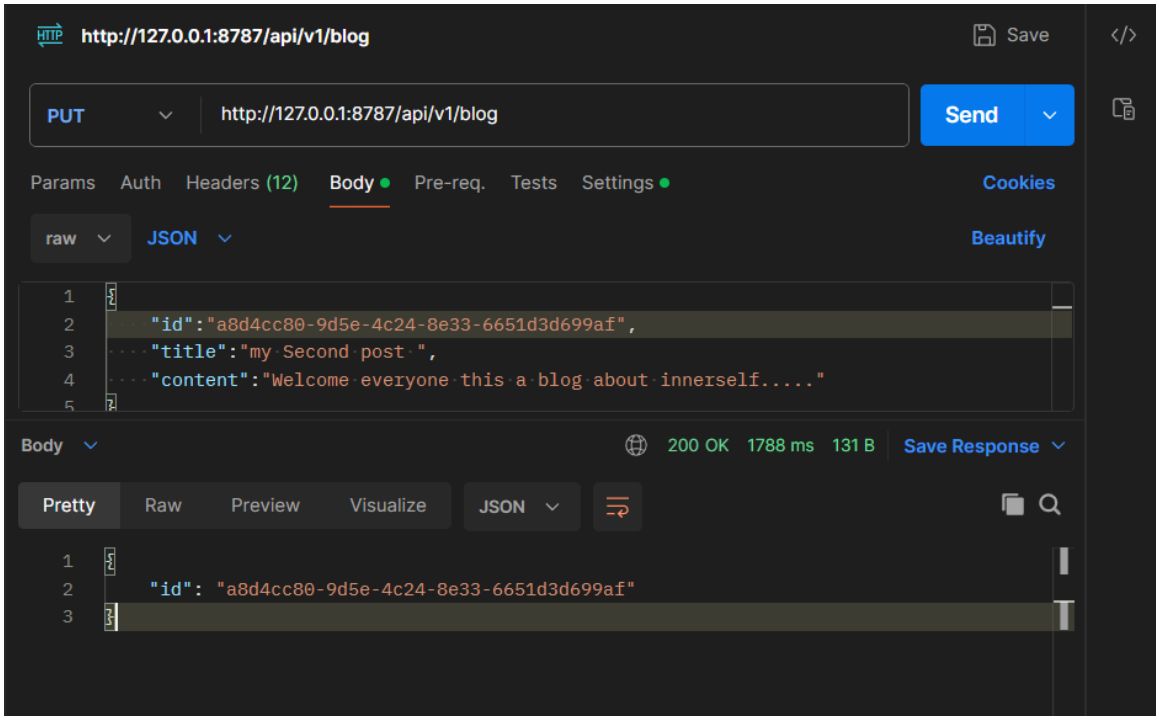


Now lets give wrong authorization token

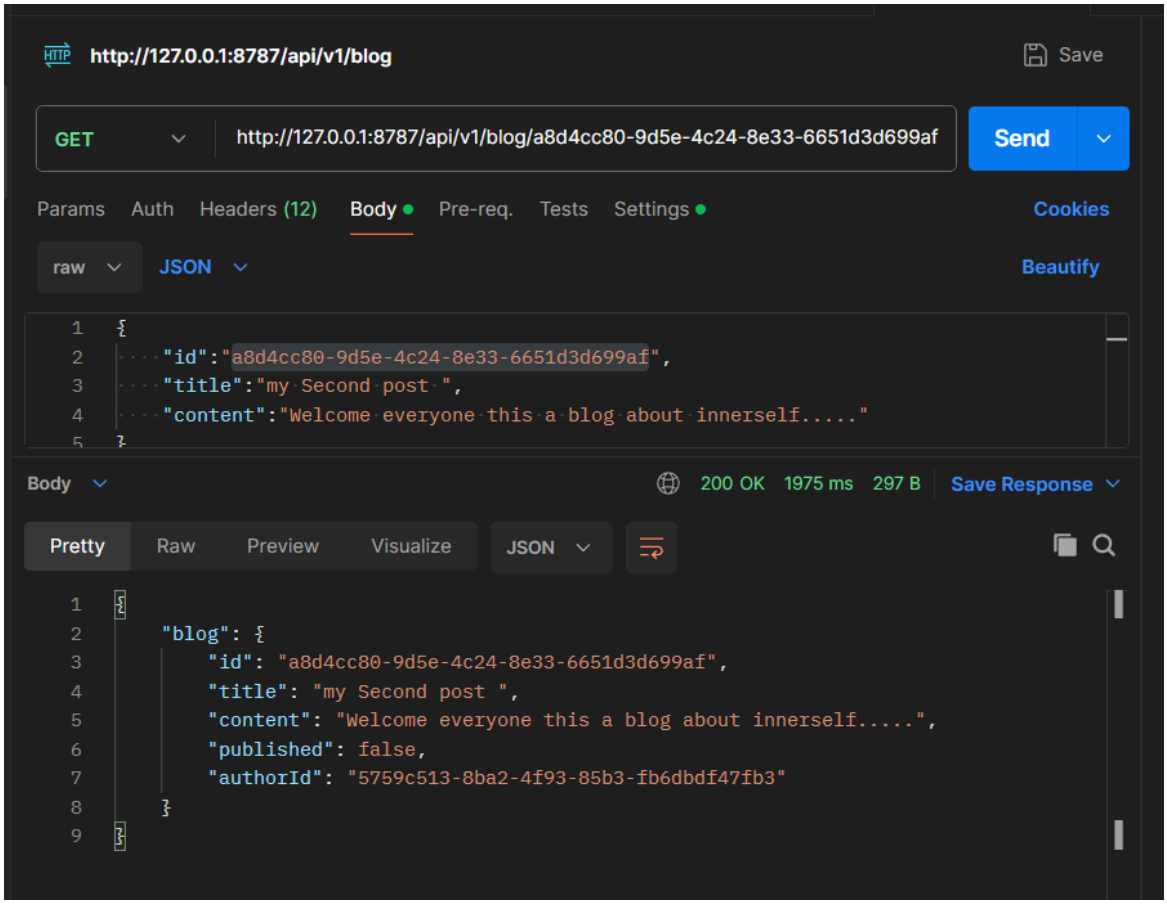


We get an error

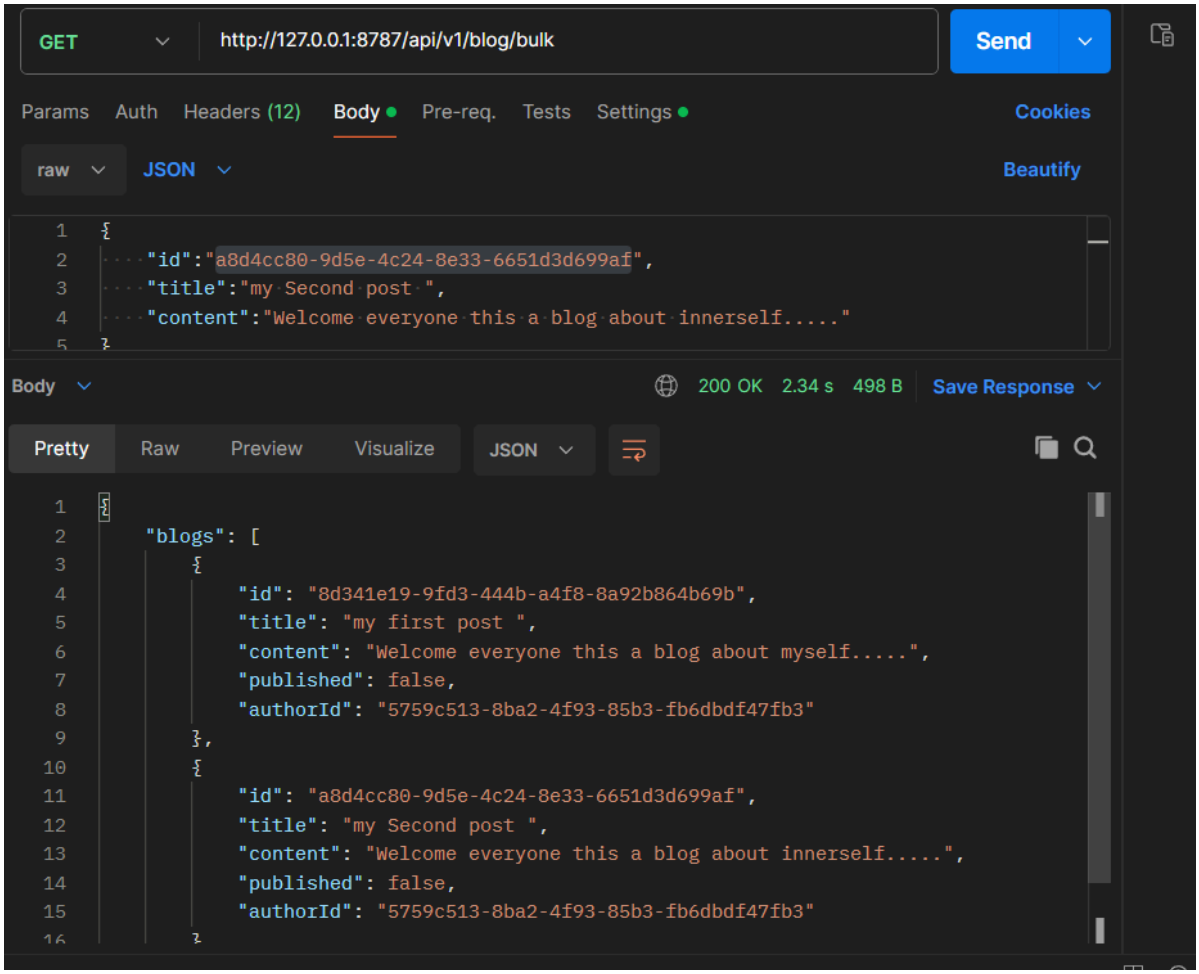
Put endpoint which is used to update



Get request to fetch data



Bulk endpoint to fetch all the blogs



Deploying our App in cloudflare

npm run deploy

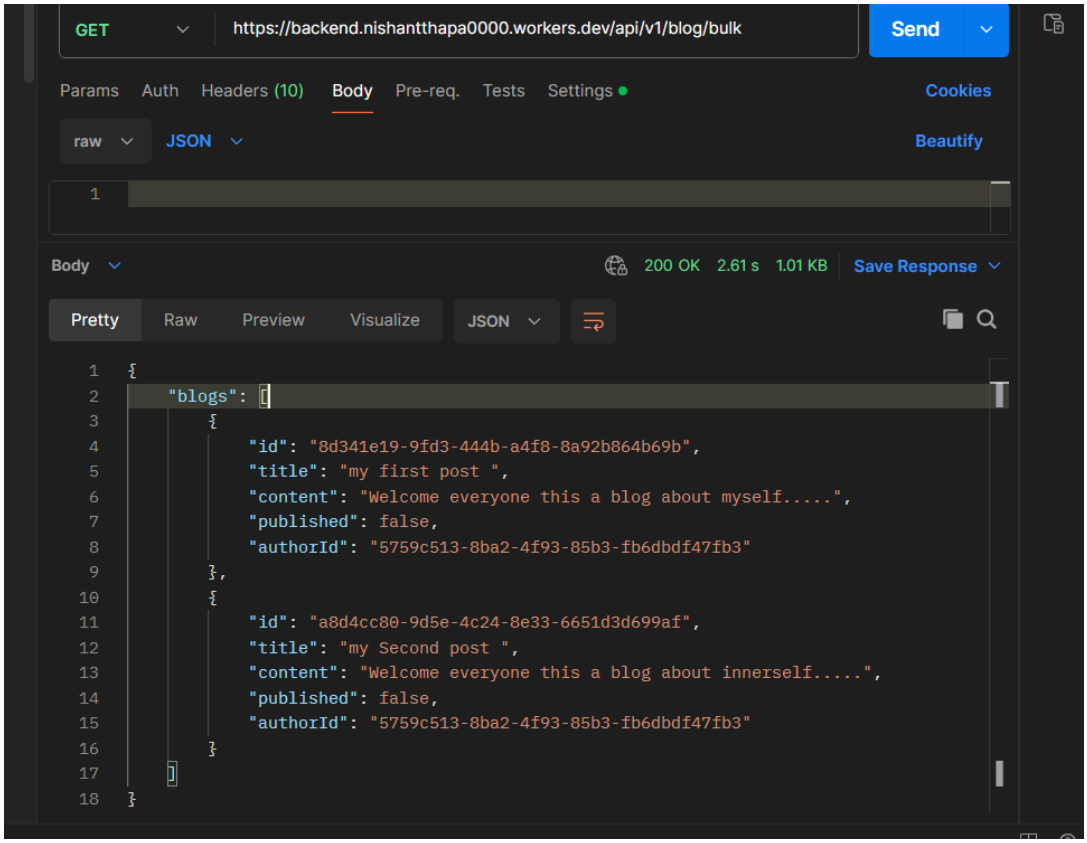
npx wrangler whoami

To check whether connected to cloudflare or not

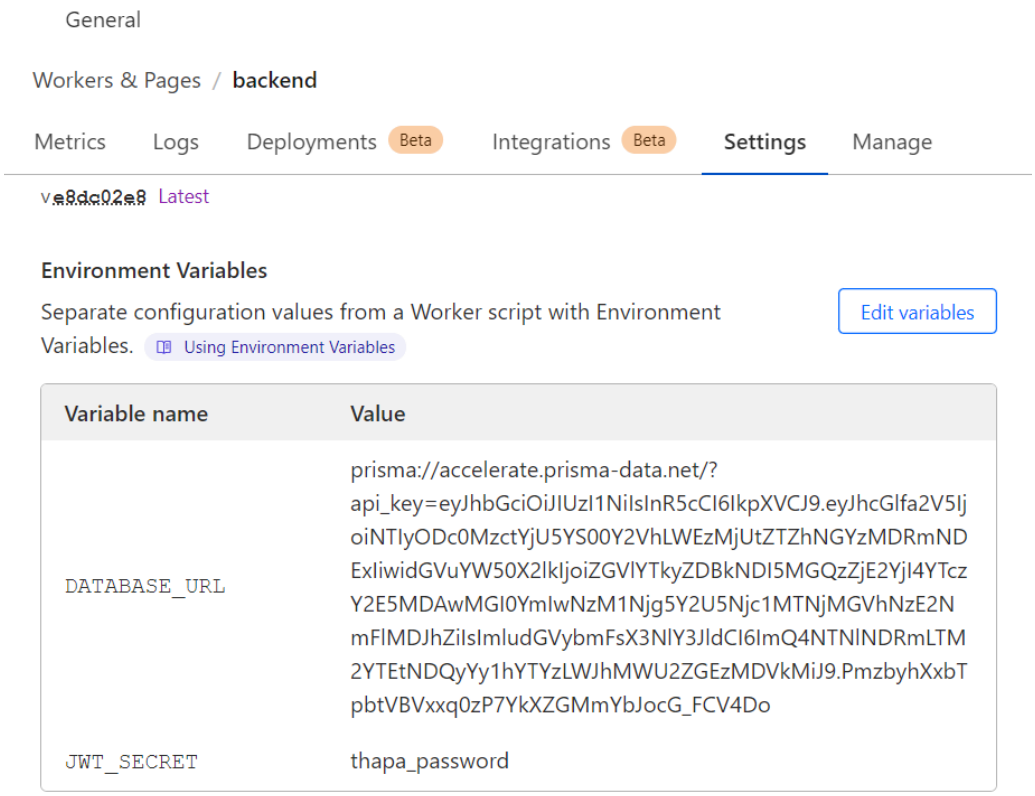
https://backend.nishantthapa0000.workers.dev

```
C:\Users\NTC\Desktop\Dev\Week13\MediumBlog\backend>npm run deploy

> deploy
> wrangler deploy --minify src/index.ts

 ⛅ wrangler 3.41.0 (update available 3.47.1)
-----------------------------------------------------
Your worker has access to the following bindings:
- Vars:
  - DATABASE_URL: "prisma://accelerate.prisma-data.net/?..."
  - JWT_SECRET: "thapa_password"
Total Upload: 189.71 KiB / gzip: 62.36 KiB
Uploaded backend (4.24 sec)
Published backend (9.27 sec)
  https://backend.nishantthapa0000.workers.dev
Current Deployment ID: e8dc02e8-ffa6-4473-9842-03054bd5edeb


NOTE: "Deployment ID" in this output will be changed to "Version ID" in a futu
re version of Wrangler. To learn more visit: https://developers.cloudflare.com
/workers/configuration/versions-and-deployments
```

Now we can replace our local url with this production ready url



We can update the environment variable in setting



Zod Validation

Type inference in zod

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

We need to check whether the body send by user is correct format or not,

One way is to do is

```
import z from 'zod';
const signupInput = z.object({
  email: z.string().email(),
  pasword:z.string().min(6),
  name:z.string().optional()
})
```

```
userRouter.post('/signup',async (c) => {
    const prisma = new PrismaClient({
      datasourceUrl: c.env.DATABASE_URL,
  }).$extends(withAccelerate())
    const body = await c.req.json();
    const {success} = signupInput.safeParse(body);
    if(!success){
      c.status(411);
      return c.json({
        message:"Inputs not correct "
      })
    }
    try{
    const user = await prisma.user.create({
      data:{
        email:body.email,
        password:body.password,
        name:body.name
      },
    })
    // Authentications , whole point of signup need to do is entry in
database and second thing sign up route need to do is return the user a
JWT, user can send in local storage and eventually send in every request
    const token =await sign({
      id:user.id
    },c.env.JWT_SECRET)
    return c.json({jwt: token})
  }catch(e){
    c.status(411)
    return c.text('Invalid')
  }
  })
```

The problem comes when we also need these in the frontend

type inference in zod

export type SignupInput = z.infer<typeof signupInput>

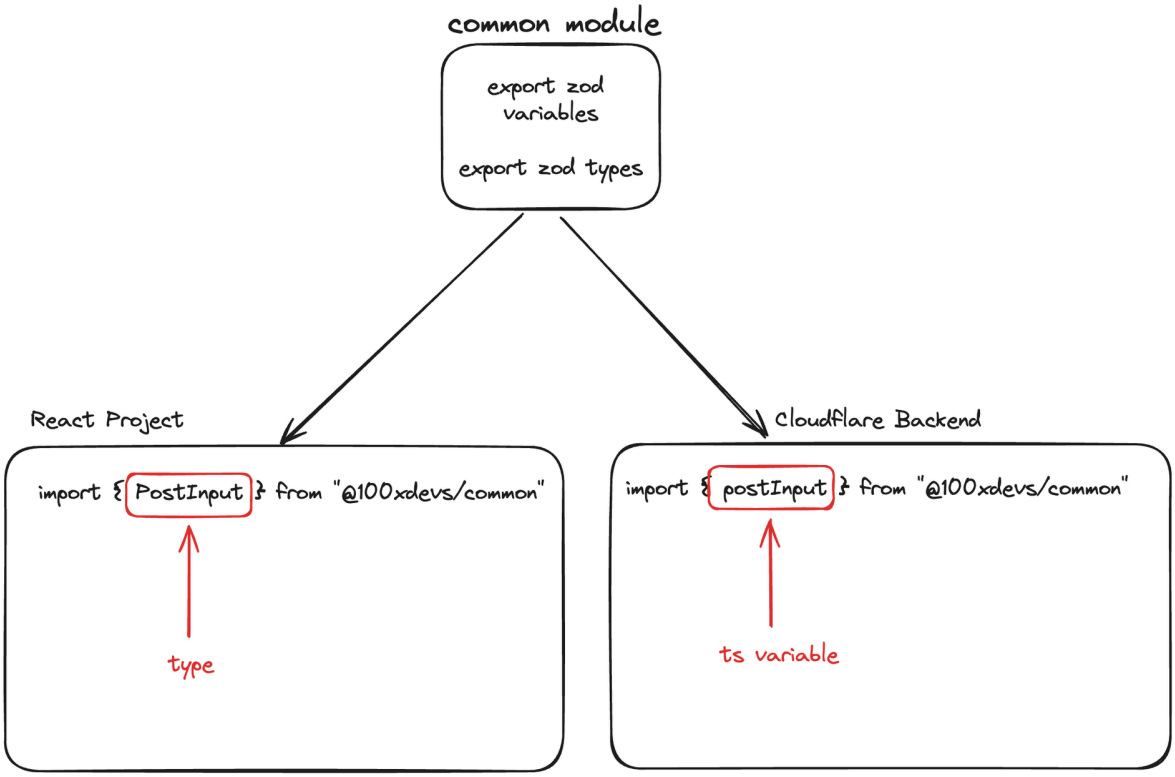
Basically wherever we have defined the zod variable it is better to also export its type.

We are exporting this from backend folder how will frontend will access it

Hence this is not right place to export zod type

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

We will now divide our project into two part



We will divide our project into 3 parts

Backend

Frontend

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

```
 "declaration": true,
```

In tsconfig.json

Zod module

```
import z from 'zod';
export const signupInput = z.object({
  email: z.string().email(),
  pasword:z.string().min(6),
  name:z.string().optional()
})
export const signinInput = z.object({
    email: z.string().email(),
    pasword:z.string().min(6),
  })
export const createBlogInput = z.object({
    title:z.string(),
    content:z.string()
})
export const updateBlogInput = z.object({
    title:z.string(),
    content:z.string(),
    id:z.string()
})

export type SignupInput= z.infer<typeof signupInput>;
export type SigninInput = z.infer<typeof signinInput>
export type CreateBlogInput = z.infer<typeof createBlogInput>
export type UpdateBlogInput = z.infer<typeof updateBlogInput>
```

How can backend access common folder??

And import it

Should we do “../../../common/src/index”

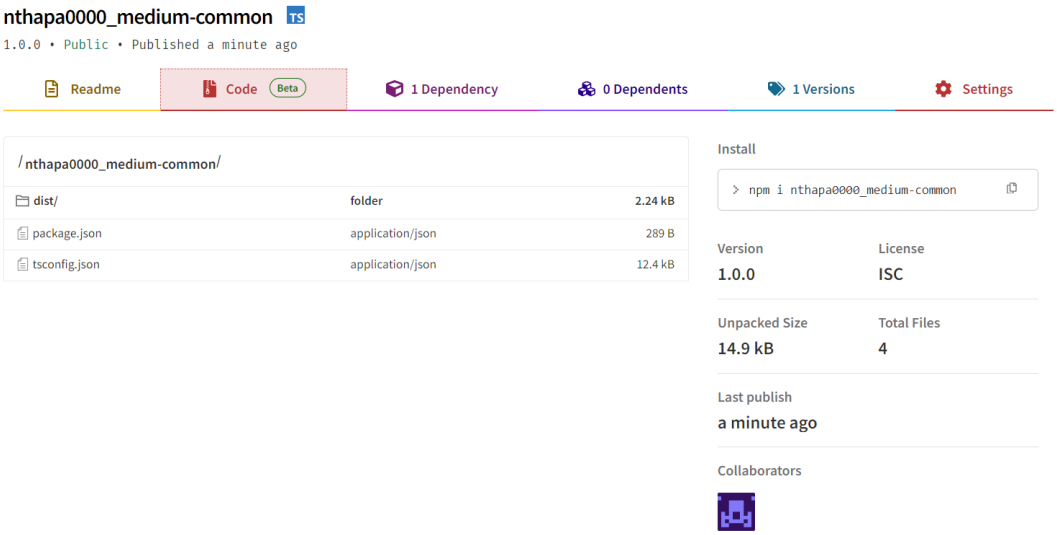
It can throw error as cloudflare doesn’t understand anything outside the backend folder

Ideally we should do monorepos

But we will publish common to npm

npm login

npm publish –access public

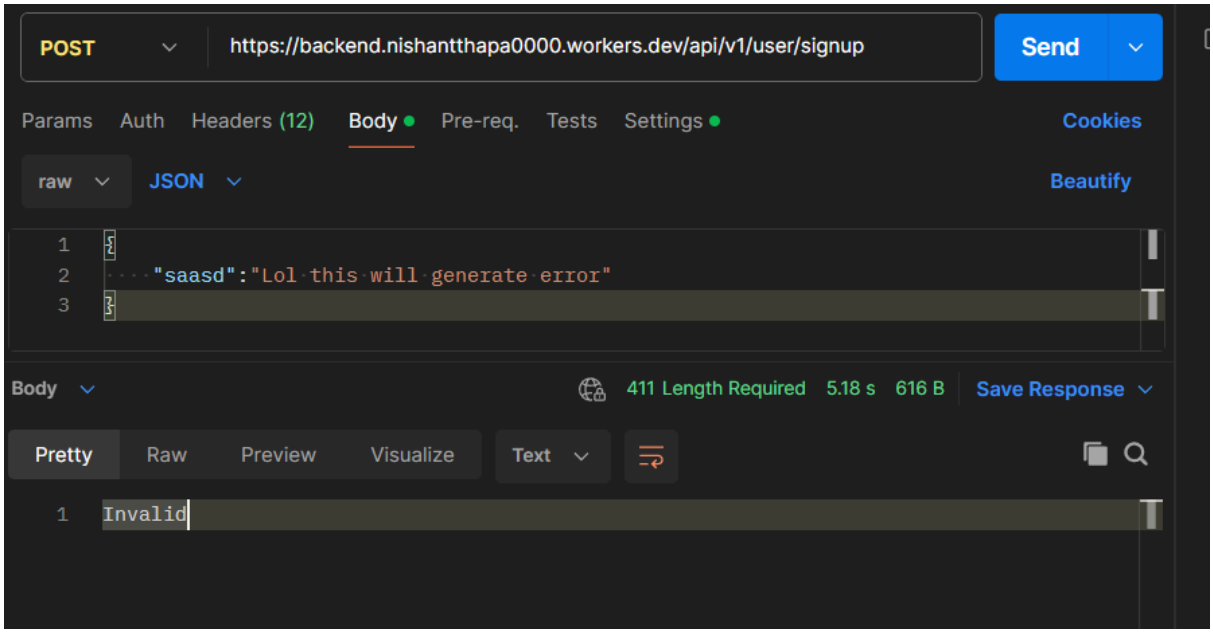


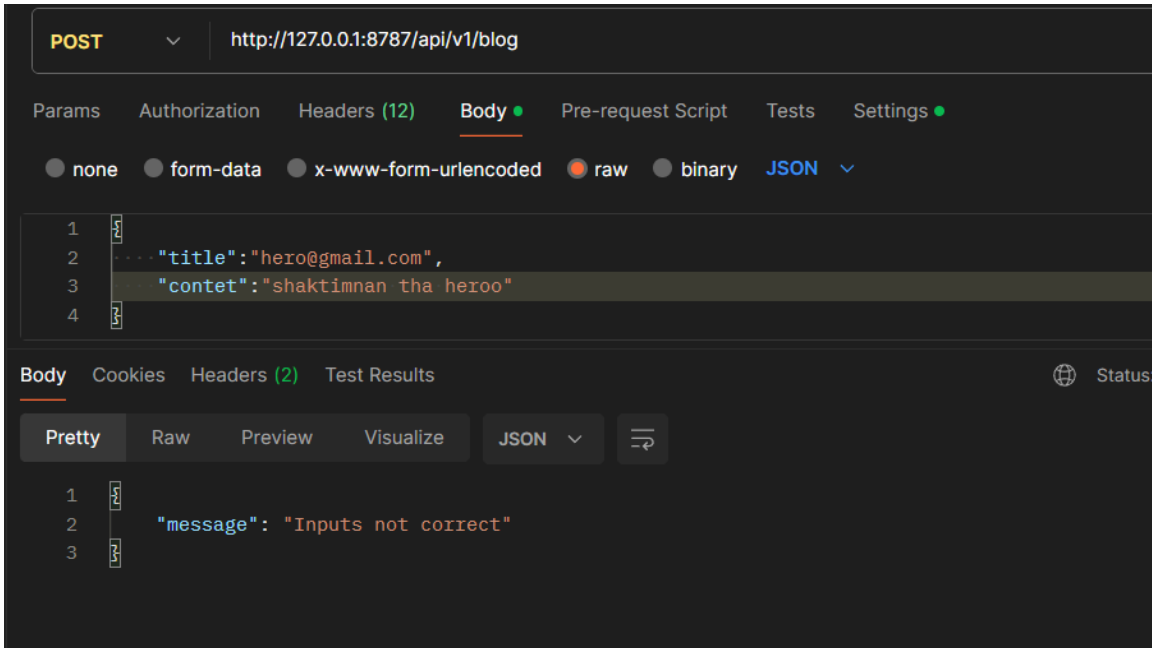
Now we will install it in our backend folder

```
C:\Users\NTC\Desktop\Dev\Week13\MediumBlog\backend>npm  install nthapa0000_med
ium-common

added 1 package, and audited 85 packages in 5s

11 packages are looking for funding
  run `npm fund` for details

1 low severity vulnerability

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
```

```
import  {signupInput}  from 'nthapa0000_medium-common'
```

Now lets hit signup with wrong zod input




POST
http://127.0.0.1:8787/api/v1/blog
Params
Authorization
Headers (12)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
JSON
{
"title":"hero@gmail.com",
"contet":"shaktimnan tha heroo"
}
Body
Cookies
Headers (2)
Test Results
Status
Pretty
Raw
Preview
Visualize
JSON
{
"message": "Inputs not correct"
}