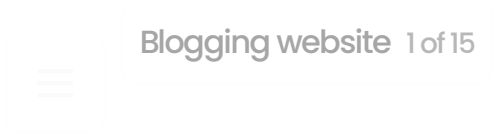


Step 1 - The stack

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

Blogging website 1 of 15

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/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

💡 https://hono.dev/api/routing

▼ Solution

```
import { Hono } from 'hono';

// Create the main Hono app
const app = new Hono();

app.post('/api/v1/signup', (c) => {
  return c.text('signup route')
})

app.post('/api/v1/signin', (c) => {
  return c.text('signin route')
```

```
c                                    d', (c) => {
  Blogging website 1 of 15           n('id')
    console.log(id);
    return c.text('get blog route')
})

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

    return c.text('signin route')
})

app.put('/api/v1/blog', (c) => {
    return c.text('signin route')
})

export default app;
```

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=eyJhbGciOiJIUzI1NiIsInR5cCI6I
```

Make s Blogging website 1 of 15 ckend folder

```
)m 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=eyJhbGciOi
```

💡 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
```

Blogging website 1 of 15

```
  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())
```

Blogging website 1 of 15

Step 5 - Create non auth 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

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

    return c.text('jwt here')
  } catch(e) {
    return c.status(403);
  }
})
```

💡 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: {
```

Blogging website 1 of 15

💡 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/

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

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

// Create the main Hono app
const app = new Hono<{
    Bindings: {
        DATABASE_URL: string,
        JWT_SECRET: string,
    }
}>();


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

```
    prisma.user.create({
      data: {
        email: body.email,
        password: body.password
      }
    });
    const jwt = await sign({ id: user.id }, c.env.JWT_SECRET);
    return c.json({ jwt });
  } catch(e) {
    c.status(403);
    return c.json({ error: "error while signing up" });
  }
})
```

3. Add a signin route

▼ Solution

```
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 });
})
```

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()
})
```

Write th the user id and passes it over to the main route.

Blogging website 1 of 15

to pass data from middleware to the route handler?

Using the context - https://hono.dev/api/context

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

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

▼ Solution

```
app.use('/api/v1/blog/*', async (c, next) => {
    const jwt = c.req.header('Authorization');
    if (!jwt) {
        c.status(401);
        return c.json({ error: "unauthorized" });
    }
    const token = jwt.split(' ')[1];
    const payload = await verify(token, c.env.JWT_SECRET);
    if (!payload) {
        c.status(401);
        return c.json({ error: "unauthorized" });
    }
    c.set('userId', payload.id);
    await next()
})
```

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

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

Callout

> 💡 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

Hono le together so you can have a cleaner file

Create two new files -

routes/user.ts

routes/blog.ts

and push the user routes to user.ts

▼ index.ts

```
import { Hono } from 'hono'
import { userRouter } from './routes/user';
import { bookRouter } from './routes/blog';

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

app.route('/api/v1/user', userRouter)
app.route('/api/v1/book', bookRouter)

export default app
```

▼ user.ts

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

export const userRouter = new Hono<{
   Bindings: {
      DATABASE_URL: string;
      JWT_SECRET: string;
   }
}>();

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

lerate());

Blogging website 1 of 15

```
  const body = await c.req.json();

  const user = await prisma.user.create({
    data: {
      email: body.email,
      password: body.password,
    },
  });

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

  return c.json({
    jwt: token
  })
})

userRouter.post('/signin', async (c) => {
  const prisma = new PrismaClient({
  //@ts-ignore
    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 });
})
```

Blogging website 1 of 15

1. Create the route to initialize a blog/post

▼ Solution

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

    const body = await c.req.json();
    const post = await prisma.post.create({
        data: {
            title: body.title,
            content: body.content,
            authorId: userId
        }
    });
    return c.json({
        id: post.id
    });
})
```

2. Create the route to update blog

▼ Solution

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

    const body = await c.req.json();
    prisma.post.update({
        where: {
            id: body.id,
            authorId: userId
```

```
        content: body.content
      }
    });

    return c.text('updated post');
});
```

3. Create the route to get a blog

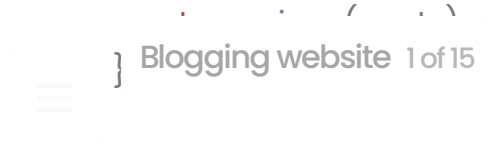
▼ Solution

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

    const post = await prisma.post.findUnique({
      where: {
        id
      }
    });

    return c.json(post);
})
```

4. Create the route to get all blogs

▼ Solution

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

                                          ({});
```



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 wan't to get and set values on the context of the request, you can use `c.get` and `c.set`

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

c

Blogging website 1 of 15

💡 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

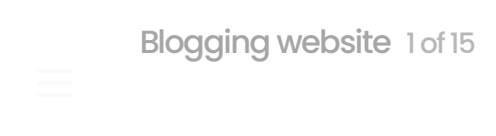
Step 9 - Deploy your app

```
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.

\ npm module for now.

Eventu monorepos make it easier to have multiple p g e 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`

Blogging website 1 of 15

```
{
  "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

```
import z from "zod";

export const signupInput = z.object({
    email: z.string().email(),
    password: z.string(),
    name: z.string().optional(),
});

export type SignupType = z.infer<typeof signupInput>;

export const signinInput = z.object({
    email: z.string().email(),
```

```
export type SigninType = z.infer<typeof signinInput>;

export const createPostInput = z.object({
    title: z.string(),
    content: z.string(),
});

export type CreatePostType = z.infer<typeof createPostInput>;

export const updatePostInput = z.object({
    title: z.string().optional(),
    content: z.string().optional(),
});

export type UpdatePostType = z.infer<typeof updatePostInput>;
```

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

Blogging website 1 of 15

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

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

// Create the main Hono app
const app = new Hono<{
    Bindings: {
        DATABASE_URL: string,
        JWT_SECRET: string,
    },
    Variables : {
        userId: string
    }
}>();

app.use('/api/v1/blog/*', async (c, next) => {
    const jwt = c.req.header('Authorization');
    if (!jwt) {
        c.status(401);
        return c.json({ error: "unauthorized" });
    }
    const token = jwt.split(' ')[1];
    const payload = await verify(token, c.env.JWT_SECRET);
    if (!payload) {
        c.status(401);
        return c.json({ error: "unauthorized" });
```

```
'.id);
})

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 { success } = signupInput.safeParse(body);
  if (!success) {
    c.status(400);
    return c.json({ error: "invalid input" });
  }
  try {
    const user = await prisma.user.create({
      data: {
        email: body.email,
        password: body.password
      }
    });
    const jwt = await sign({ id: user.id }, c.env.JWT_SECRET);
    return c.json({ jwt });
  } catch(e) {
    c.status(403);
    return c.json({ error: "error while signing up" });
  }
})

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 { success } = signinInput.safeParse(body);
  if (!success) {
    c.status(400);
    return c.json({ error: "invalid input" });
  }
  const user = await prisma.user.findUnique({
```

```
  });

  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 });
})

app.get('/api/v1/blog/:id', async (c) => {
  const id = c.req.param('id');
  const prisma = new PrismaClient({
    datasourceUrl: c.env?.DATABASE_URL  ,
  }).$extends(withAccelerate());

  const post = await prisma.post.findUnique({
    where: {
      id
    }
  });

  return c.json(post);
})

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

  const body = await c.req.json();
  const { success } = createPostInput.safeParse(body);
  if (!success) {
    c.status(400);
    return c.json({ error: "invalid input" });
  }

  const post = await prisma.post.create({
```

```
            authorId: userId
        }
    });
    return c.json({
        id: post.id
    });
})

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

    const body = await c.req.json();
    const { success } = updatePostInput.safeParse(body);
    if (!success) {
        c.status(400);
        return c.json({ error: "invalid input" });
    }

    prisma.post.update({
        where: {
            id: body.id,
            authorId: userId
        },
        data: {
            title: body.title,
            content: body.content
        }
    });

    return c.text('updated post');
});

export default app;
```

Blogging website 1 of 15

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 Fmp :all 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
```

1. 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 (
```

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

export default App
```

1. Make sure you can import `types` from `your_package`

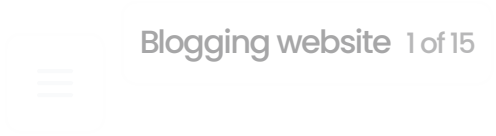
Step 15 - Creating the components

Designs generated from v0.dev - an AI service by vercel that lets you generate frontends

Signup page

Blogs page

Create blog page



Blogs page