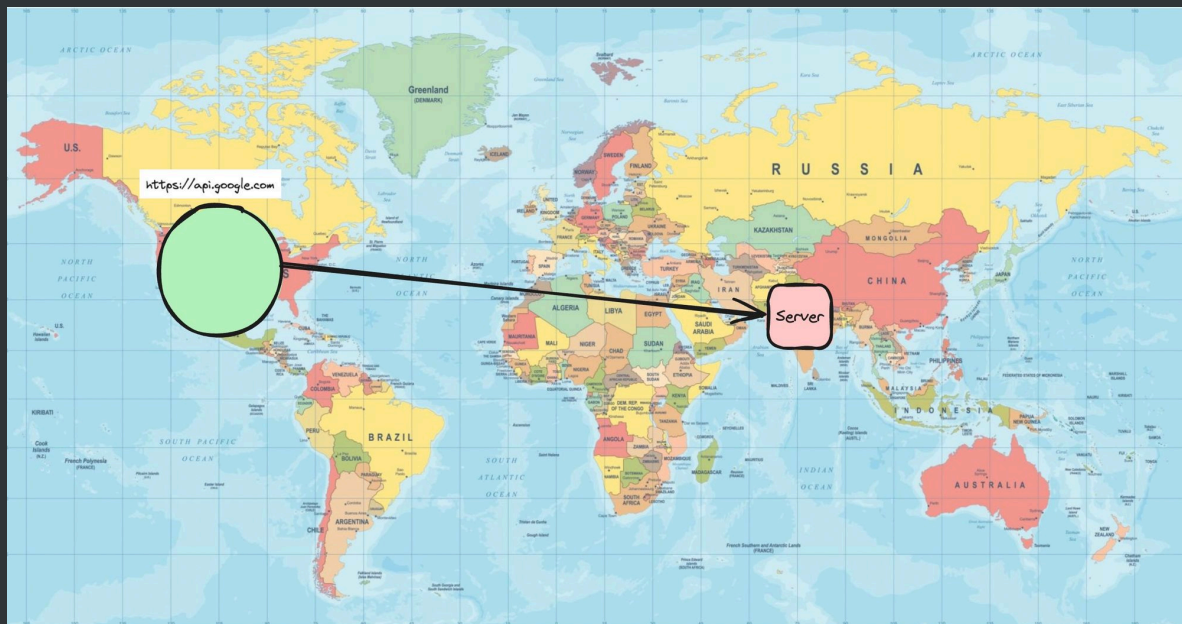


# What are backends servers?



You might've used `express` to create a Backend server.

The way to run it usually is `node index.js` which starts a process on a certain port (3000 for example)

When you have to deploy it on the internet, there are a few ways -

1. Go to aws, GCP, Azure, Cloudflare
1. Rent a VM (Virtual Machine) and deploy your app
2. Put it in an Auto scaling group
3. Deploy it in a Kubernetes cluster

There are a few downsides to doing this -

1. Taking care of how/when to scale
2. Base cost even if no one is visiting your website
3. Monitoring various servers to make sure no server is down

What if, you could just write the code and someone else could take care of all of these problems?

# What are **serverless** Backends



"Serverless" is a backend deployment in which the **cloud provider** dynamically manages the allocation and provisioning of servers. The term "serverless" doesn't mean there are no servers involved. Instead, it means that developers and operators do not have to worry about the servers.

## Easier defination

What if you could just write your **express routes** and run a command. The app would automatically

1. Deploy
2. Autoscale
3. Charge you on a **per request** basis (rather than you paying for VMs)

## Problems with this approach

1. More expensive at scale
2. Cold start problem

# Famous serverless providers

There are many famous backend serverless providers -

## ▼ AWS Lambda

[https://aws.amazon.com/pm/lambda/?trk=5cc83e4b-8a6e-4976-92ff-7a6198f2fe76&sc\\_channel=ps&ef\\_id=CjwKCAiAt5euBhB9EiwAdkXWO-i-th4J3onX9ji-tPt\\_JmsBAQJLWYN4hzTF0Zxb084EkUBxSCK5vhoC-1wQAvD\\_BwE:G:s&s\\_kwcid=AL!4422!3!651612776783!e!!g!!aws-lambda!19828229697!143940519541](https://aws.amazon.com/pm/lambda/?trk=5cc83e4b-8a6e-4976-92ff-7a6198f2fe76&sc_channel=ps&ef_id=CjwKCAiAt5euBhB9EiwAdkXWO-i-th4J3onX9ji-tPt_JmsBAQJLWYN4hzTF0Zxb084EkUBxSCK5vhoC-1wQAvD_BwE:G:s&s_kwcid=AL!4422!3!651612776783!e!!g!!aws-lambda!19828229697!143940519541)

## ▼ Google Cloud Functions

<https://firebase.google.com/docs/functions>

## ▼ Cloudflare Workers

<https://workers.cloudflare.com/>

## You write code. We handle the rest.

Deploy serverless code instantly across the globe to give it exceptional performance, reliability, and scale.

Start building

Read docs

- From signup to globally deployed in <5min
- Your code runs within **milliseconds** of your users worldwide
- Say goodbye to cold starts—support for **0ms worldwide**

```
~/ $ npm create cloudflare -- my-app
~/ $ cd my-app
~/my-app $ npx wrangler deploy
Published https://my-app.world.workers.dev
```

# When should you use a serverless architecture?

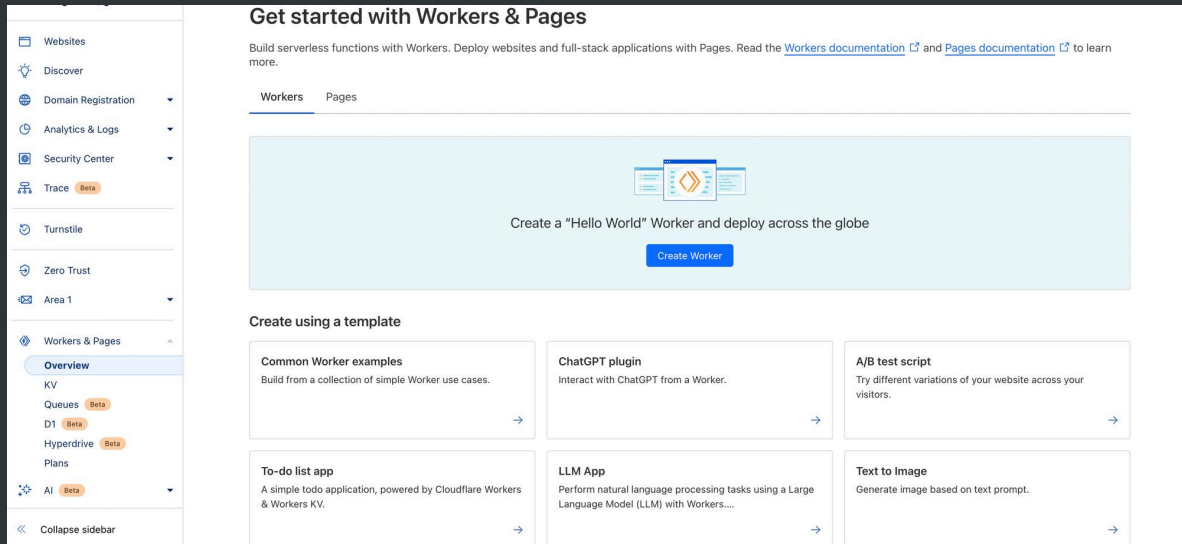
1. When you have to get off the ground fast and don't want to worry about deployments
2. When you can't anticipate the traffic and don't want to worry about autoscaling
3. If you have very low traffic and want to optimise for costs

# Cloudflare workers setup

We'll be understanding cloudflare workers today.

Reason - No credit card required to deploy one

Please sign up on <https://cloudflare.com/>



Try creating a test worker from the UI (Common worker examples) and try hitting the URL at which it is deployed

# How cloudflare workers work?

Detailed blog post - <https://developers.cloudflare.com/workers/reference/how-workers-works/#:~:text=Though Cloudflare Workers behave similarly,used by Chromium and Node.>

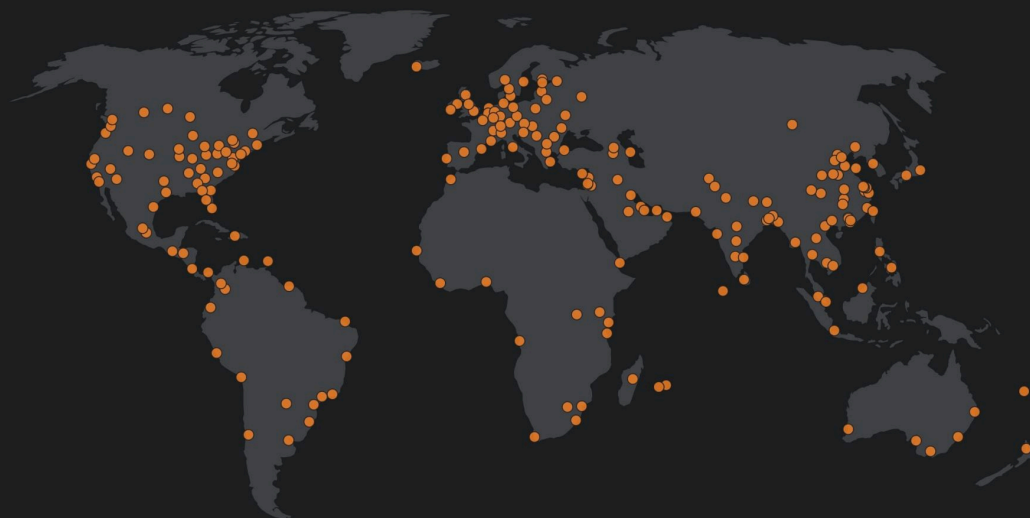


Cloudflare workers DONT use the Node.js runtime. They have created their own runtime. There are a lot of things that Node.js has

## How Workers works

Though Cloudflare Workers behave similarly to [JavaScript](#) in the browser or in Node.js, there are a few differences in how you have to think about your code. Under the hood, the Workers runtime uses the [V8 engine](#) — the same engine used by Chromium and Node.js. The Workers runtime also implements many of the standard [APIs](#) available in most modern browsers.


The differences between JavaScript written for the browser or Node.js happen at runtime. Rather than running on an individual's machine (for example, [a browser application or on a centralized server](#)), Workers functions run on [Cloudflare's Edge Network](#) - a growing global network of thousands of machines distributed across hundreds of locations.



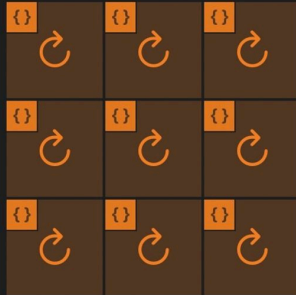
Each of these machines hosts an instance of the Workers runtime, and each of those runtimes is capable of running thousands of user-defined applications. This guide will review some of those differences.

## Isolates vs containers

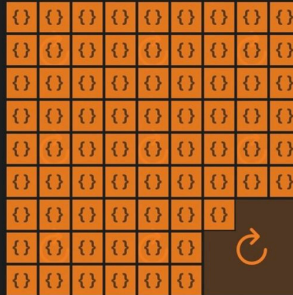
## Isolates

[V8](#)  orchestrates isolates: lightweight contexts that provide your code with variables it can access and a safe environment to be executed within. You could even consider an isolate a sandbox for your function to run in.

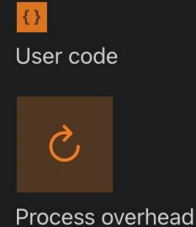
A single runtime can run hundreds or thousands of isolates, seamlessly switching between them. Each isolate's memory is completely isolated, so each piece of code is protected from other untrusted or user-written code on the runtime. Isolates are also designed to start very quickly. Instead of creating a virtual machine for each function, an isolate is created within an existing environment. This model eliminates the cold starts of the virtual machine model.




Traditional architecture



Workers V8 isolates



Unlike other serverless providers which use [containerized processes](#)  each running an instance of a language runtime, Workers pays the overhead of a JavaScript runtime once on the start of a container. Workers processes are able to run essentially limitless scripts with almost no individual overhead by creating an isolate for each Workers function call. Any given isolate can start around a hundred times faster than a Node process on a container or virtual machine. Notably, on startup isolates consume an order of magnitude less memory.

A given isolate has its own scope, but isolates are not necessarily long-lived. An isolate may be spun down and evicted for a number of reasons:

- Resource limitations on the machine.
- A suspicious script - anything seen as trying to break out of the Isolate sandbox.
- Individual [resource limits](#).

## Initializing a worker

To create and deploy your application, you can take the following steps -

### ▼ Initialize a worker

```
npm create cloudflare -- my-app
```

Copy

Select **no** for **Do you want to deploy your application**

### ▼ Explore package.json dependencies



```
"wrangler": "^3.0.0"
```

Copy

Notice `express` is not a dependency there

▼ Start the worker locally

```
npm run dev
```

Copy

▼ How to return json?

```
export default {  
  async fetch(request: Request, env: Env, ctx: ExecutionContext):  
    return Response.json({  
      message: "hi"  
    });  
},  
};
```

Copy

## Question - Where is the express code? HTTP Server?

Cloudflare expects you to just write the logic to handle a request.

Creating an HTTP server on top is handled by cloudflare

## Question - How can I do `routing` ?

In express, routing is done as follows -

```
import express from "express"  
const app = express();  
  
app.get("/route", (req, res) => {  
  // handles a get request to /route  
});
```

Copy

How can you do the same in the Cloudflare environment?

```
export default {  
  async fetch(request: Request, env: Env, ctx: ExecutionContext): Promise<Response>  
    console.log(request.body);  
    console.log(request.headers);  
  
    if (request.method === "GET") {  
      return Response.json({  
        message: "you sent a get request"  
      });  
    } else {
```

Cop



```
        return Response.json({
            message: "you did not send a get request"
        });
    },
};
```



How to get query params - <https://community.cloudflare.com/t/parse-url-query-strings-with-cloudflare-workers/90286>

Cloudflare does not expect a routing library/http server out of the box. You can write a full application with just the constructs available above.

We will eventually see how you can use other HTTP frameworks (like express) in cloudflare workers.

## Deploying a worker

Now that you have written a basic HTTP server, let's get to the most interesting bit — **Deploying it on the internet**

We use **wrangler** for this (Ref <https://developers.cloudflare.com/workers/wrangler/>)



You have to buy a plan to be able to do this

You also need to buy the domain on cloudflare/transfer the domain to cloudflare



# Adding express to it

Why can't we use express? Why does it cloudflare doesn't start off with a simple express boiler plate?

## Reason 1 - Express heavily relies on Node.js

<https://community.cloudflare.com/t/express-support-for-workers/390844>

1

123testcoding

Jun '22

Being a developer, I really love Cloudflare Pages for hosting static apps! But, frontend apps usually need API to get dynamic data. I have existing Express apps that I would like to transfer to Workers, in addition to transferring my frontend app to Cloudflare Pages.

Is it known whether Express support will be added for Cloudflare Workers?

2

👍

🔗

created

last reply

3

6.9k

2

6

👤

1

2

1

Jun '22

👤

Jun '22

replies

views

users

likes

👤

1

2

---

👤

cherryjimbo MVP '23

Jun '22

It's unlikely you'll see specifically `express` on Workers due to its deep Node.js dependencies, however there are a lot of options that you'll probably feel right at home with, and have very similar APIs to something like Express. To name just a few:

- [GitHub - honojs/hono: Ultrafast web framework for Cloudflare Workers, Deno, Bun, and Node.js.](#) Fast, but not only fast. 818
- [GitHub - lukeed/worktop: The next generation web framework for Cloudflare Workers](#) 343
- [GitHub - kwhitley/itty-router: A little router.](#) 287

In addition, if you're running the frontend app in Pages, you may even be able to run your backend in Pages too, using Functions: [Functions · Cloudflare Pages docs](#) 140

<https://github.com/honojs/hono>

## You can split all your handlers in a file

Create a generic `handler` that you can forward requests to from either `express` or `hono` or `native cloudflare handler`



jsmrcaga

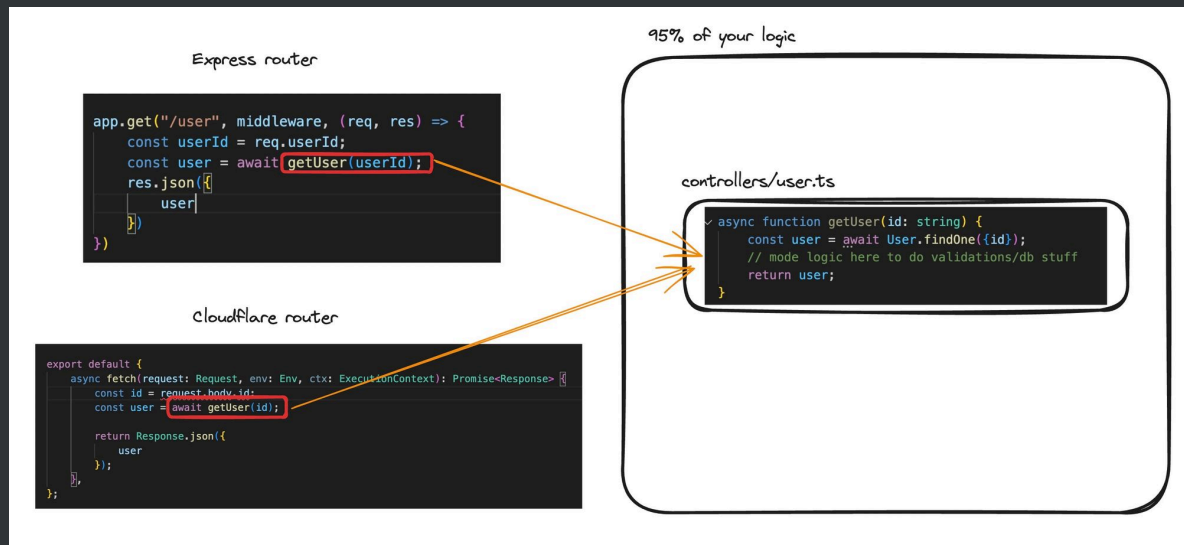
Jul '23

I can link the Node.js compatibility docs for Workers

<https://developers.cloudflare.com/workers/runtime-apis/nodejs/#nodejs-compatibility> 244

It is important to know that workers run on a different runtime built by Cloudflare (not Node.js). I don't expect compatibility for worker\_threads to be arriving anytime soon.

If you really need to deploy a Node.js app I would search more for a classic serverless option. If you can split your app into small components and don't need to rely heavily on Node.js, Cloudflare Workers are an amazing option and there are routing libraries for them as well (ie: to replace express).



# Using hono

## What is Hono

<https://hono.dev/concepts/motivation>

## # Motivation

At first, I just wanted to create a web application on Cloudflare Workers. But, there was no good framework that works on Cloudflare Workers, so I started building Hono and thought it would be a good opportunity to learn how to build a router using Trie trees.

Then a friend showed up with ultra crazy fast router called "RegExpRouter". And, I also had a friend who created the Basic authentication middleware.

Thanks to using only Web Standard APIs, we could make it work on Deno and Bun. "No Express for Bun?" we could answer, "No, but there is Hono" though Express works on Bun now.

We also have friends who make GraphQL servers, Firebase authentication, and Sentry middleware. And there is also a Node.js adapter. An ecosystem has been created.

In other words, Hono is damn fast, makes a lot of things possible, and works anywhere. You can look that Hono will become **Standard for Web Standard**.

## What runtimes does it support?

## Getting Started



Basic

Cloudflare Workers

Cloudflare Pages

Deno

Bun

Fastly Compute

Vercel

Netlify

AWS Lambda

Lambda@Edge

Supabase Functions

Node.js

Others

## Working with cloudflare workers -

1. Initialize a new app

```
npm create hono@latest my-app
```

Copy

1. Move to `my-app` and install the dependencies.

```
cd my-app  
npm i
```

Copy

1. Hello World

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

Copy

# Middlewares



<https://hono.dev/guides/middleware>

## Creating a simple auth middleware

```
import { Hono, Next } from 'hono'  
import { Context } from 'hono/jsx';  
  
const app = new Hono()  
  
app.use(async (c, next) => {  
  if (c.req.header("Authorization")) {  
    // Do validation  
    await next()  
  } else {  
    return c.text("You dont have acces");  
  }  
})  
  
app.get('/', async (c) => {  
  const body = await c.req.parseBody()  
  console.log(body);  
  console.log(c.req.header("Authorization"));  
  console.log(c.req.query("param"));  
  
  return c.json({msg: "as"})  
})  
  
export default app
```

Copy





Notice you have to return the `c.text` value

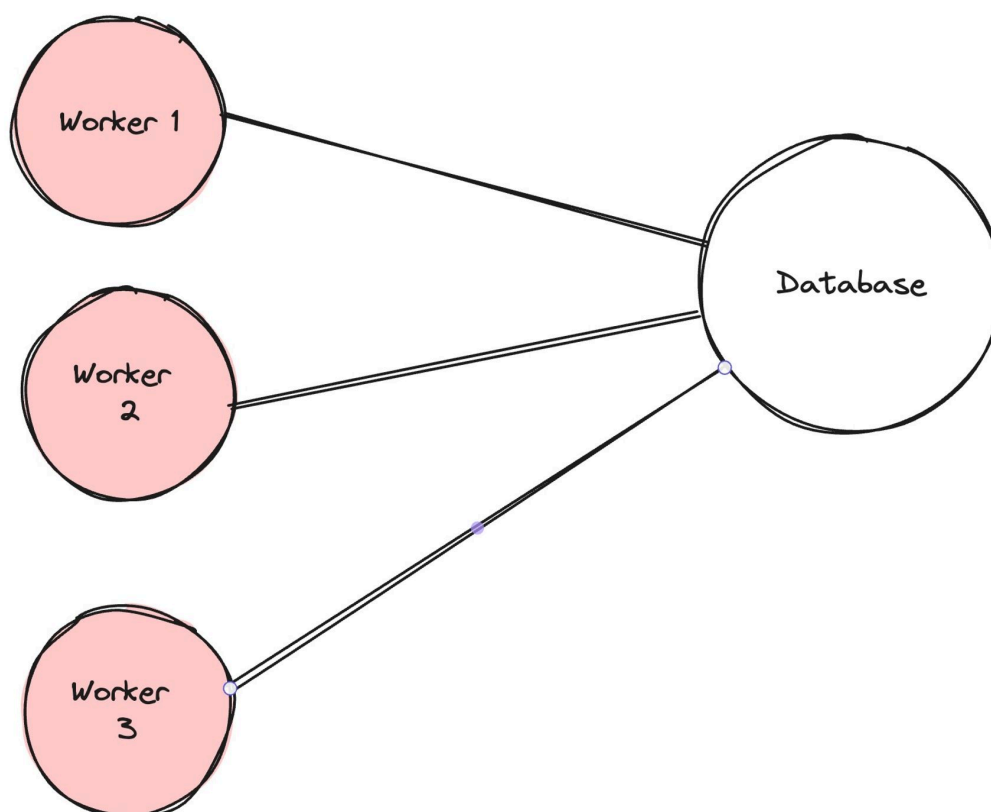
## Connecting to DB



<https://www.prisma.io/docs/orm/prisma-client/deployment/edge/deploy-to-cloudflare-workers>

Serverless environments have one big problem when dealing with databases.

1. There can be many connections open to the DB since there can be multiple workers open in various regions



1. **Prisma** the library has dependencies that the **cloudflare runtime** doesn't understand.

## Connection pooling in prisma for serverless env



<https://www.prisma.io/docs/accelerate>

<https://www.prisma.io/docs/orm/prisma-client/deployment/edge/deploy-to-cloudflare-workers>

### 1. Install prisma in your project

```
npm install --save-dev prisma
```

Copy

### 2. Init Prisma

```
npx prisma init
```

Copy

### 3. Create a basic schema

```
generator client {  
  provider = "prisma-client-js"  
}  
  
datasource db {  
  provider = "postgresql"  
  url      = env("DATABASE_URL")  
}  
  
model User {  
  id      Int    @id @default(autoincrement())  
  name    String  
  email   String  
  password String  
}
```

Copy

### 4. Create migrations

```
npx prisma migrate dev --name init
```

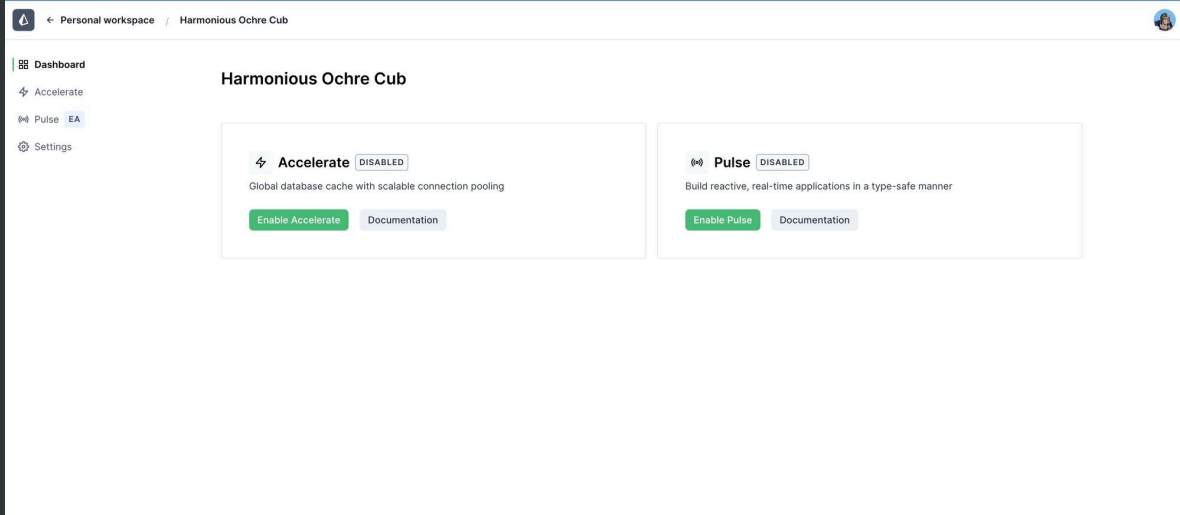
Copy

## 5. Signup to Prisma accelerate

```
https://console.prisma.io/login
```

Copy

### Enable accelerate



### Generate an API key

#### Enable Accelerate

✔ Successfully enabled, you can now generate an API key and add Accelerate to your Prisma application.

#### Add API key

Update your database connection string in your .env file to use your new API key with the Accelerate connection string.

```
DATABASE_URL="prisma://accelerate.prisma-data.net/?api_key=eyJhbGciOiJIUzI1NiIsI
```

I've securely stored my connection string

#### Add Accelerate to your application

ⓘ We recommend using the latest version of Prisma for optimal results. If your Prisma version is below 5.2.0, please follow these [setup instructions](#).

Install the latest version of Prisma Client and Accelerate Prisma Client extension

```
$ npm install @prisma/client@latest @prisma/extension-accelerate
```

### Replace it in .env

```
DATABASE_URL="prisma://accelerate.prisma-data.net/?api_key=your_key"
```

Copy

## 5. Add accelerate as a dependency

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

Copy

## 6. Generate the prisma client

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

Copy

## 7. Setup your code

```
import { Hono, Next } from 'hono'
import { PrismaClient } from '@prisma/client/edge'
import { withAccelerate } from '@prisma/extension-accelerate'
import { env } from 'hono/adapters'

const app = new Hono()

app.post('/', async (c) => {
  // Todo add zod validation here
  const body: {
    name: string;
    email: string;
    password: string
  } = await c.req.json()
  const { DATABASE_URL } = env<{ DATABASE_URL: string }>(c)

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

  console.log(body)

  await prisma.user.create({
    data: {
      name: body.name,
      email: body.email,
      password: body.password
    }
  })

  return c.json({msg: "as"})
})

export default app
```

Copy