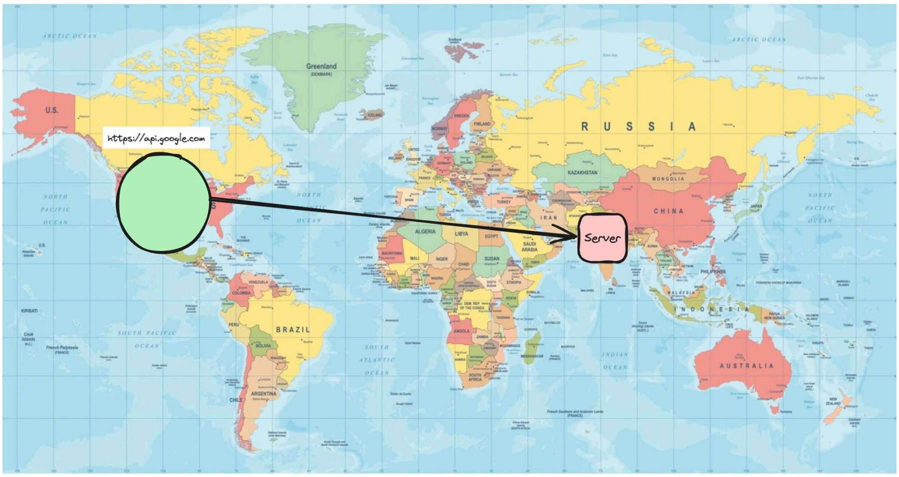
# What are backends servers?



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

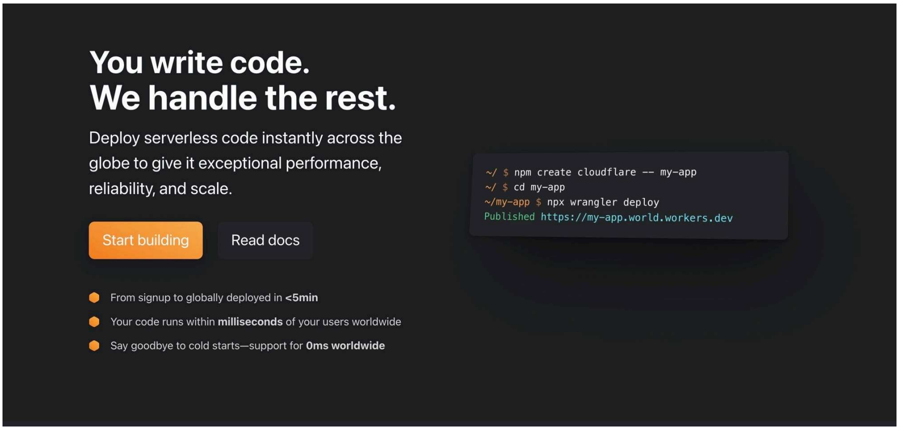
[https://aws.amazon.com/pm/lambda/?trk=5cc83e4b-8a6e-4976-92ff-7a6198f2fe76&sc_channel=ps&ef_id=CjwKCAiAt5euBhB9EiwAdkXWO-i-th4J3onX9ji-tPt_JmsBAQJLWYN4hzTF0Zxb084EkUBxSCK5vhoC-1wQAvD_BwE:G:s&s_kwcid=AL!4422!3!651612776783!e!!g!!aws lambda!19828229697!143940519541]

▼ 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.

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

- 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**

## When should you use a serverless

## architecture?

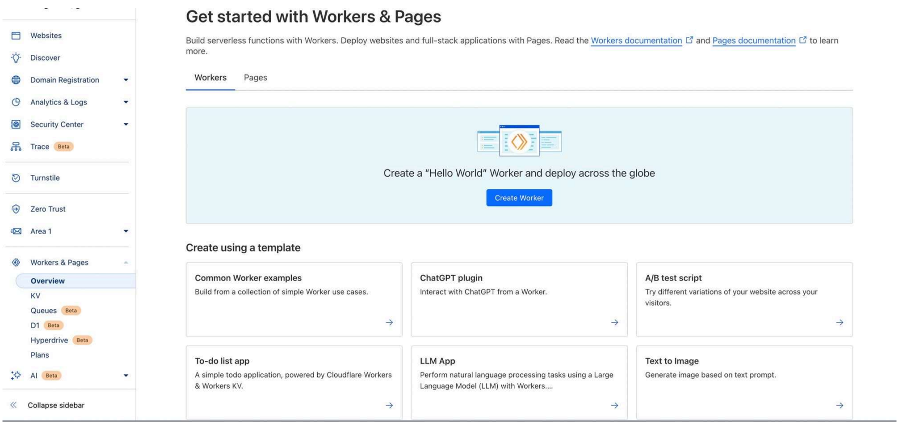
1. When you have to get off the ground fast and don’t want to worry about deployments
2. When you can’t anticipate the traffic and don’t want to worry about autoscaling
3. If you have very low traffic and want to optimise for costs

## 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/



Get started with Workers & Pages

Build serverless functions with Workers. Deploy websites and full-stack applications with Pages. Read the Workers documentation and Pages documentation to learn more.

Create a “Hello World” Worker and deploy across the globe

Create using a template

Common Worker examples
Build from a collection of simple Worker use cases.

ChatGPT plugin
Interact with ChatGPT from a Worker.

A/B test script
Try different variations of your website across your visitors.

To-do list app
A simple todo application, powered by Cloudflare Workers & Workers KV.

LLM App
Perform natural language processing tasks using a Large Language Model (LLM) with Workers....

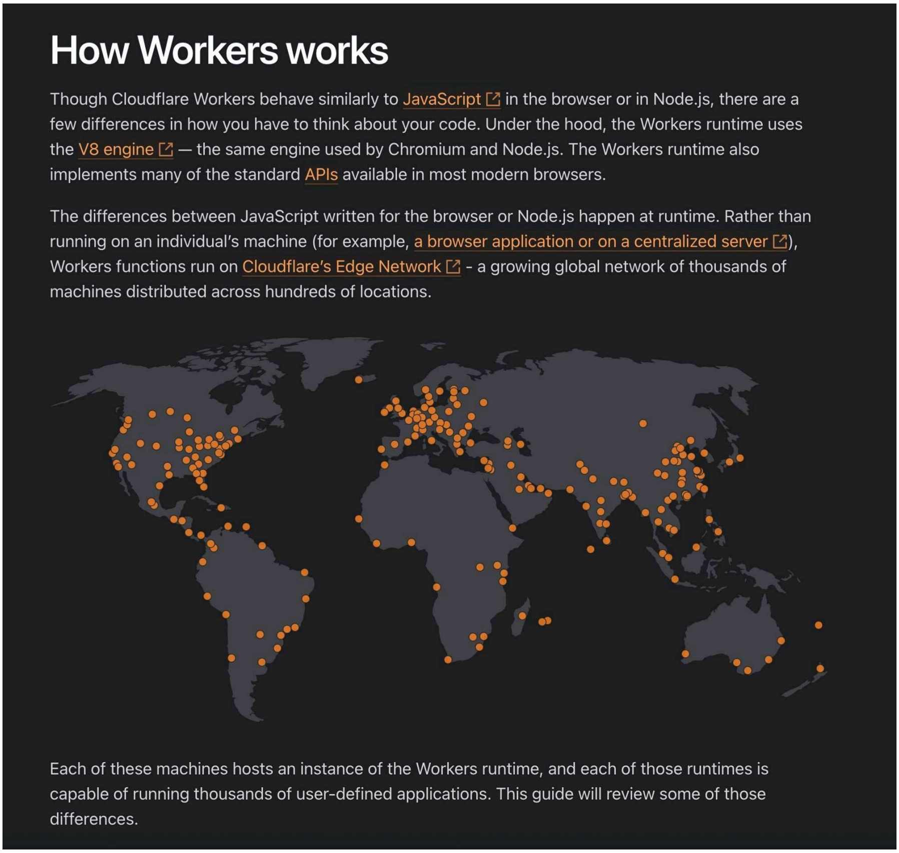
Text to Image
Generate image based on text prompt.

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.

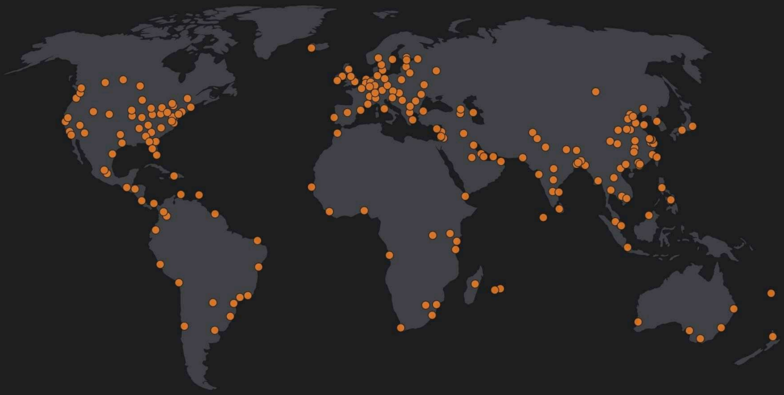
💡 Cloudflare workers DONT use the Node.js runtime. They have created their own runtime. There are a lot of things that Node.js has



## How Workers works

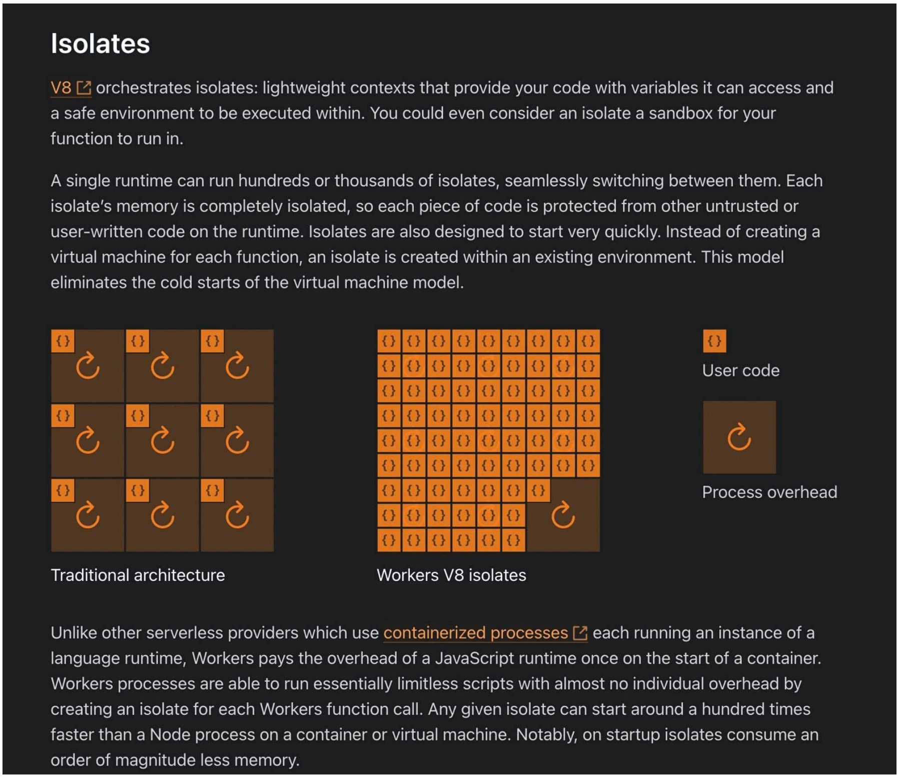
Though Cloudflare Workers behave similarly to JavaScript in the browser or in Node.js, there are a few differences in how you have to think about your code. Under the hood, the Workers runtime uses the V8 engine — the same engine used by Chromium and Node.js. The Workers runtime also implements many 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.

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.

# 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:
        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 -

Copy

```
import express from "express"
const app = express();

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

How can you do the same in the Cloudflare environment?

Copy

```
export default {
    async fetch(request: Request, env: Er
        console.log(request.body);
        console.log(request.headers);

        if (request.method === "GET") {
            return Response.json({
                message: "you sent a get
            });
        } else {
            return Response.json({
                message: "you did not ser
            });
        }
    },
};
```

💡 How to get query params - https://community.cloudflare.com/t/parseurl-query-strings-with-cloudflare-workers/90286

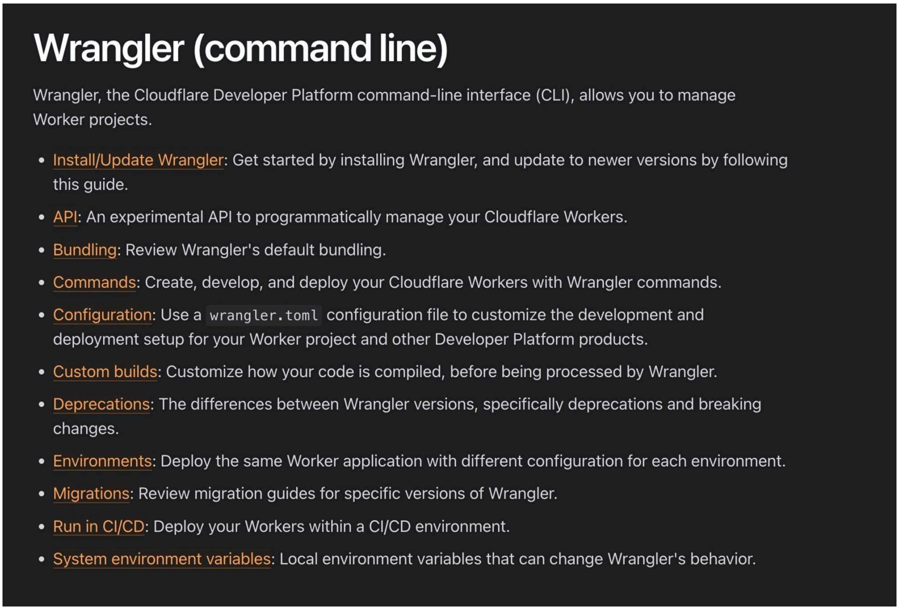
Cloudflare does not expect a routing library/http server out of the box. You can write a full application with just the constructs available above.

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/)



Wrangler (command line)

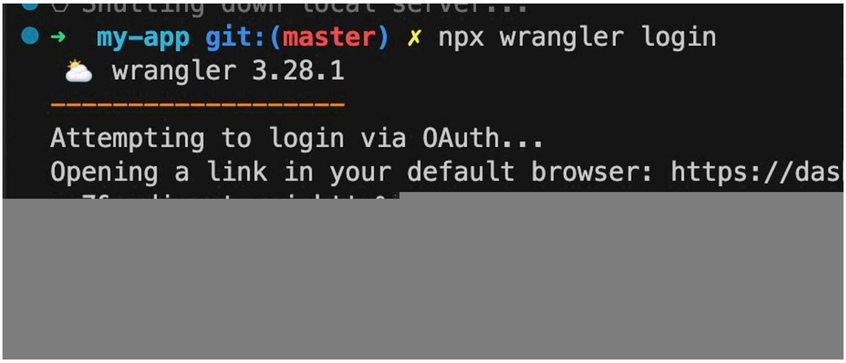
Wrangler, the Cloudflare Developer Platform command-line interface (CLI), allows you to manage Worker projects.

- Install/Update Wrangler: Get started by installing Wrangler, and update to newer versions by following this guide.
- API: An experimental API to programmatically manage your Cloudflare Workers.
- Bundling: Review Wrangler's default bundling.
- Commands: Create, develop, and deploy your Cloudflare Workers with Wrangler commands.
- Configuration: Use a `wrangler.toml` configuration file to customize the development and deployment setup for your Worker project and other Developer Platform products.
- Custom builds: Customize how your code is compiled, before being processed by Wrangler.
- Deprecations: The differences between Wrangler versions, specifically deprecations and breaking changes.
- Environments: Deploy the same Worker application with different configuration for each environment.
- Migrations: Review migration guides for specific versions of Wrangler.
- Run in CI/CD: Deploy your Workers within a CI/CD environment.
- System environment variables: Local environment variables that can change Wrangler's behavior.

## ▼ Step 1 - Login to cloudflare via the `wrangler cli`

Copy

```
npx wrangler login
```



```
Shutting down local server...
→  my-app git:(master) ✗ npx wrangler login
 ⛅️ wrangler 3.28.1
-------------------
Attempting to login via OAuth...
Opening a link in your default browser: https://das
```

[redacted]

▼ Step 2 - Deploy your worker

```
npm run deploy
```

If all goes well, you should see the app up and running

## Assigning a custom domain

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

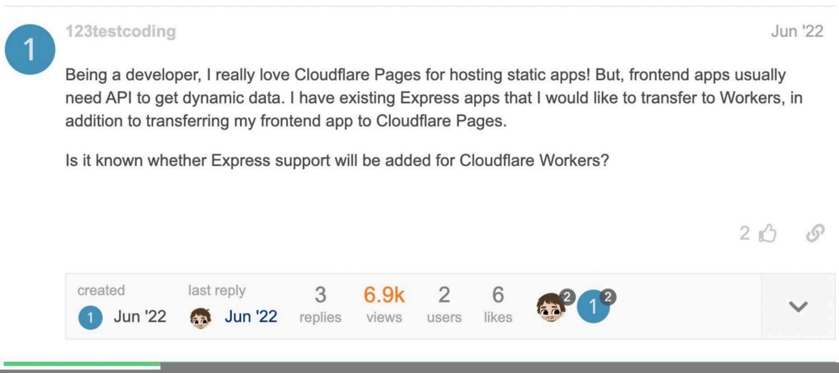
Kirags123@gmail.c
[redacted]

# 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


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 |
|---|---|---|---|---|---|
| Jun '22 | Jun '22 | replies | views | users | likes |

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`

# Using hono

## What is Hono

https://hono.dev/concepts/motivation

## What runtimes does it support?

## Working with cloudflare workers

-

1. Initialize a new app

Copy

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

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

Copy

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

1. Hello World

Copy

```
import { Hono } from 'hono'
const app = new Hono()

app.get('/', (c) => c.text('Hello Cloudf

export default app
```

## Getting inputs from user

Copy

```
import { Hono } from 'hono'

const app = new Hono()

app.get('/', async (c) => {
  const body = await c.req.json()
  console.log(body);
  console.log(c.req.header("Authorizatio
  console.log(c.req.query("param"));

  return c.text('Hello Hono!')
})

export default app
```

💡 More detail - https://hono.dev/getting-started/cloudflare-workers

## Deploying

Make sure you're logged into cloudflare ( `wrangler login` )

```
npm run deploy
```

Copy

## Middlewares

💡 https://hono.dev/guides/middleware

### Creating a simple auth middleware

Copy

```
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("Authorizatio
  console.log(c.req.query("param"));

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

export default app
```

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

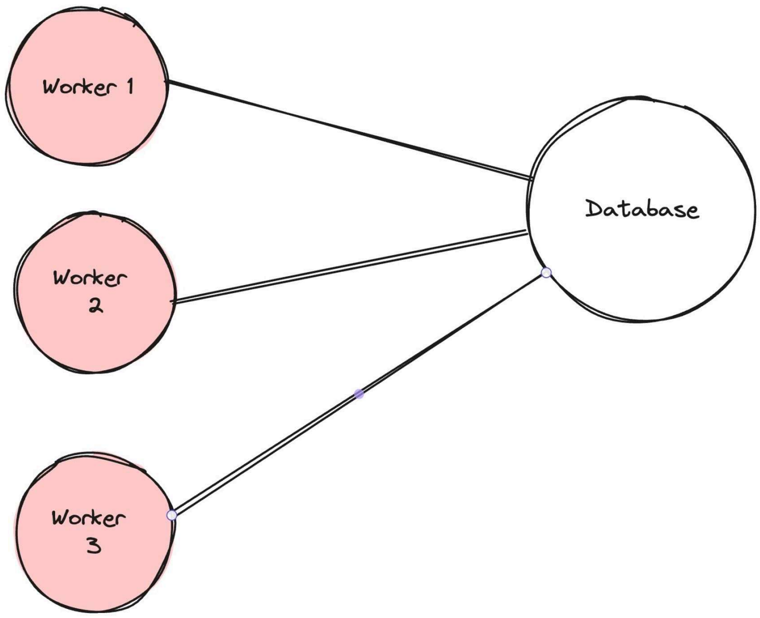
## Connecting to DB

💡 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
```

## 2. Init Prisma

```
npx prisma init
```

## 3. Create a basic schema

Copy

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

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

model User {
  id       Int    @id @default(autoincre
  name     String
  email    String
    password String
}
```

## 4. Create migrations

Copy

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

## 5. Signup to Prisma accelerate

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

Enable accelerate

Generate an API key

Replace it in .env

```
Copy
DATABASE_URL="prisma://accelerate.prisma-
```

## 5. Add accelerate as a dependency

```
npm install @prisma/extension-accel
```
Copy

## 6. Generate the prisma client

```
npx prisma generate --no-eng
```
Copy

## 7. Setup your code

Copy

```
import { Hono, Next } from 'hono'
import { PrismaClient } from '@prisma/cl
import { withAccelerate } from '@prisma/e
import { env } from 'hono/adapter'

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<{ DATABAS

  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
```