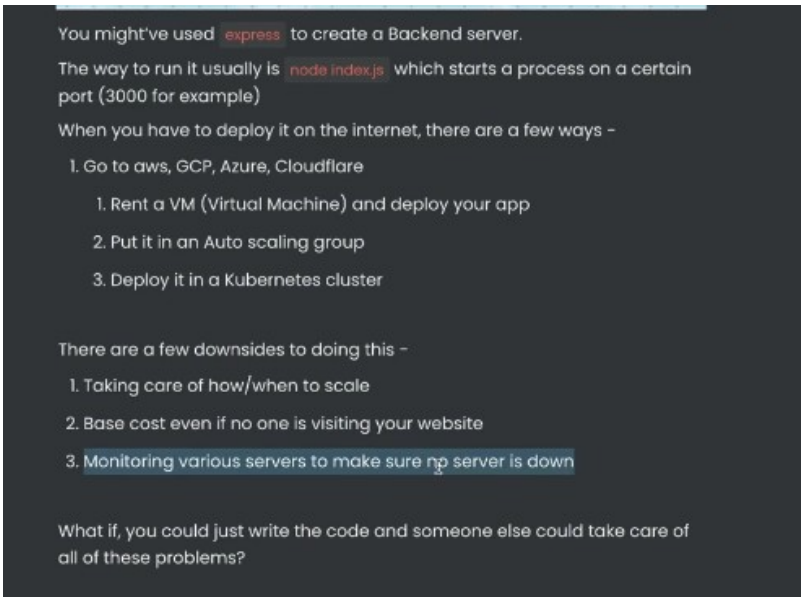
WHAT ARE BACKEND SERVERS?

They are like simple servers used in startups, hackathons etc initially its good but for scale its expensive so not used for scale

Vercel, Render are all serverless and good to use



WHAT IS SERVERLESS BACKENDS?

The better method to deploy things is to using asg, vm, kubernetes cluster and the other way is serverless
In this the cost is done based on the user request like you use this only when the number of users are less, and as a dev you don’t have to worry about the servers and all they manager on its own

very hard to migrate from serverless to vms

its very difficult to move from serverless to actual db as it require time and money

"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
Deploy
Autoscale
Charge you on a per request basis (rather than you paying for VMs)

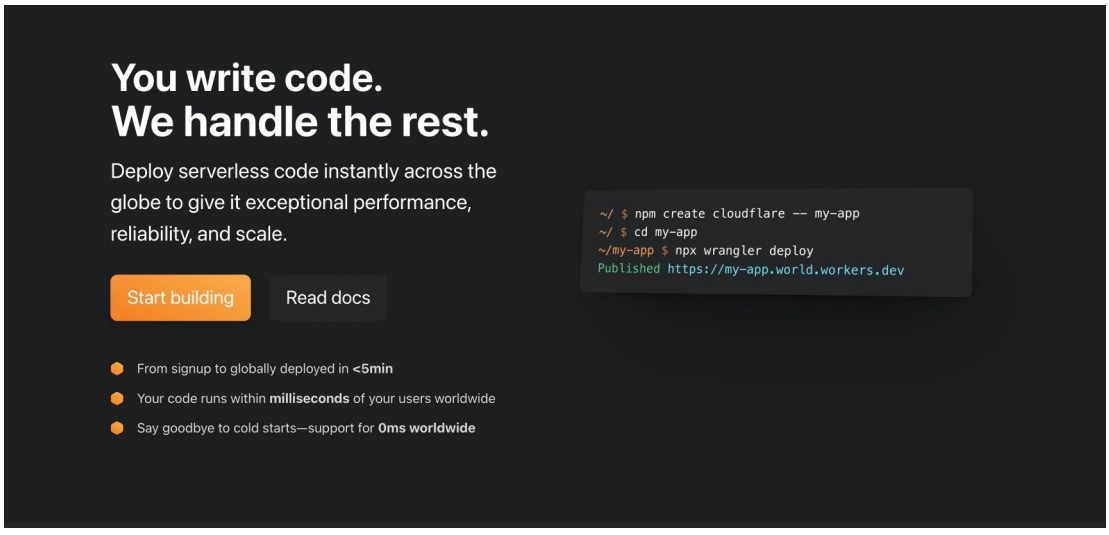
Problems with this approach

More expensive at scale
Cold start problem == when you are deploying on serverless
if imagine no one is hitting your be your backend will stop working and whenever imagine one user comes to your website then a container will start and then only your code will start to run which will take some time for the first user then for the other users it will be fast only

If no users than only the cold start problem
HOW TO SOLVE THIS?
Use a warm pool like using minimum 3-4 containers running but you have to pay money
Using a bot running somewhere , which hits the website every 5 seconds like that

Famous serverless providers

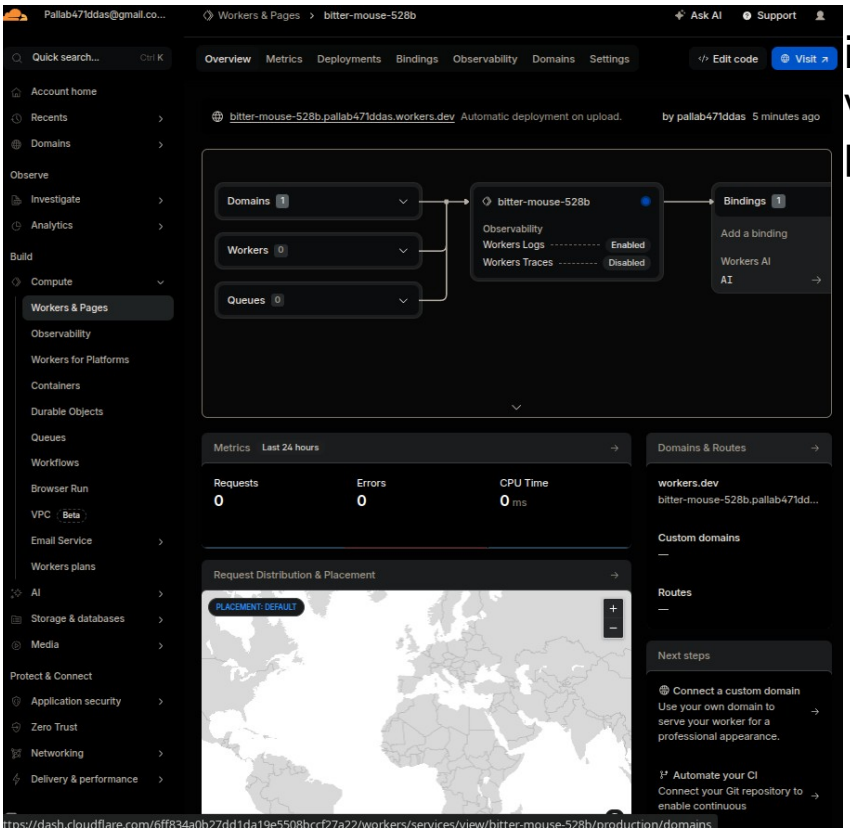
There are many famous backend serverless providers -
AWS LAMBDA
Google Cloud Functions
Cloudfare Workers



When should you use a serverless architecture?

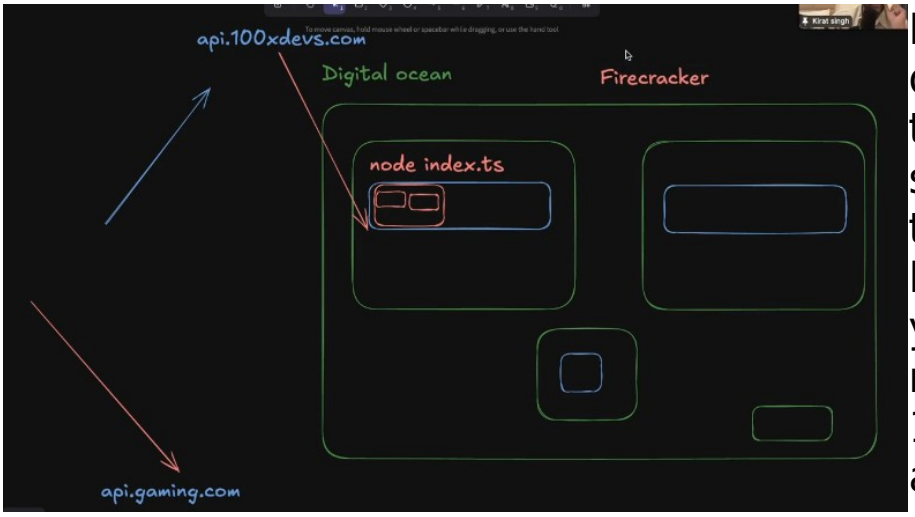
1. When you have to get off the ground fast and don’t want to worry about deployments
2. When you can’t anticipate the traffic and don’t want to worry about autoscaling
3. If you have very low traffic and want to optimise for costs

Now creating a cloudfare worker and using that
Dont buy url from the cloudfare they does like shady work if you like start making money from there url they send some random bills to pay



in this we can add env variable also in the settings part

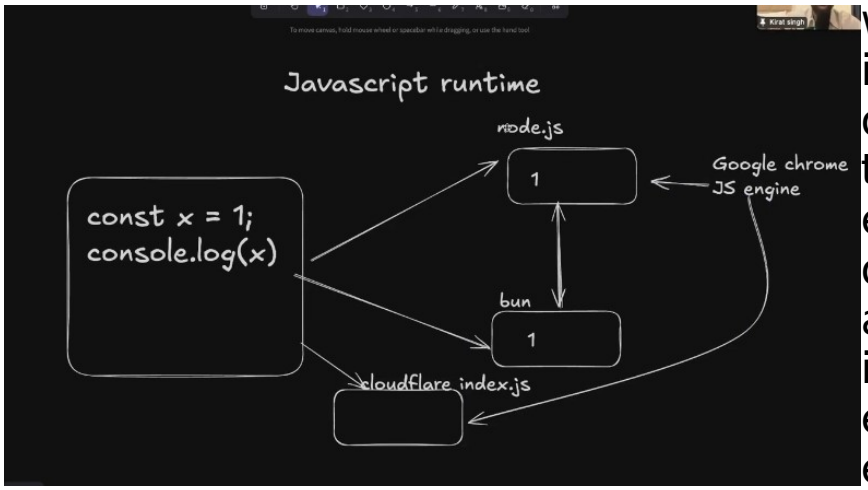
How cloudflare workers work?



Imagine you have digitalocean and you have to make a serverless server how you will do that?
It means like for a website you have to create machine like which can hit 1 user also and 100 users also , if you give like 1 machine to everyone its a waste of resources to fix that does this sounds like docker usage

only?? yes but the issue is docker also need some time to run and some computational power which is not feasible so this option is not that optimal so , so what is the approach?
AWS uses firecrackers to spin up these small machines but they use something very much related to node js which is taking a lot of time for them to involve new languages

You know what

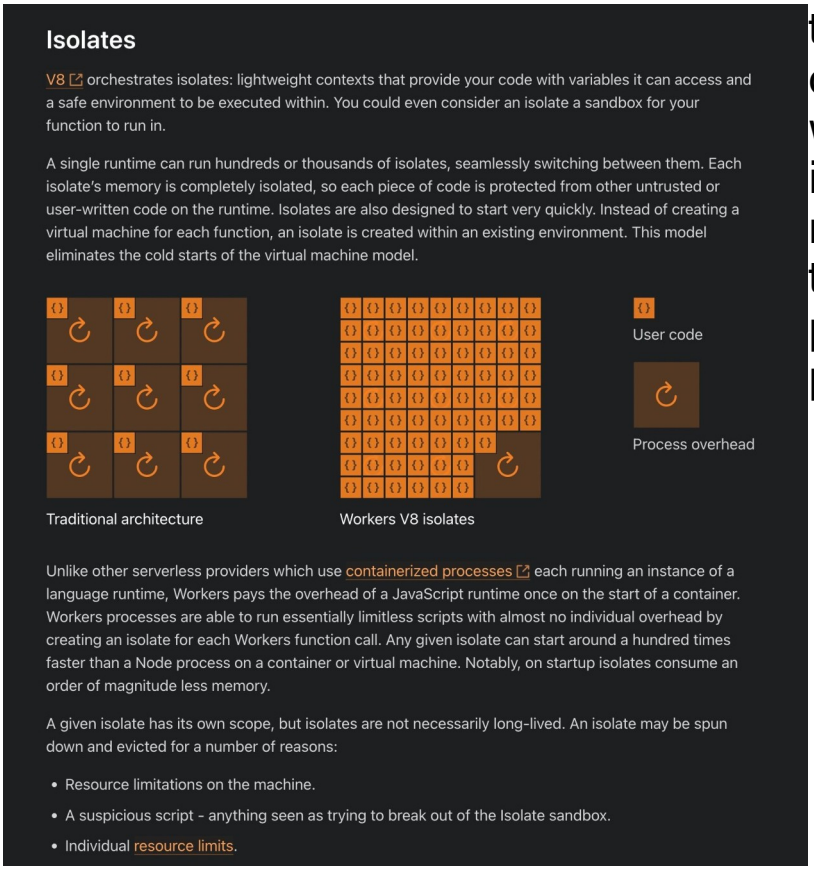


whenever we write a js code it does not give output on its own it is interpreted by things like node js , bun js etc which like gives the output but this cloudfare, aws created there own interpreter based on the v8 engine(Google chrome engine) which helps them to run the code in there machines and it also says it might not run all the code of the node js , its not fully compatiable with the node js and also in the past bun was not also compatiable but not its like 99%

there are many things which exist in browser which does not exist in node js but do exist in workers

so like our approach was we will create docker container inside a very big fat machine and then we will run all the codes but that is expensive but what we do in cloudfare is

we run a big cloudfare runtime which can run small small user code with the context in isolates



think isolate like a docker containter but much more light weight
its like they are running in one machine but the code cannot talk to each other so that the projects cannot call other projects api and all

NOW CREATING A WORKER FROM TERMINAL

npm create cloudflare-- todo-app-week-32

then select the project
then select the framework

then what tech
Workers, Assets(both fe and be), durable objects (storage very close to db but not actually db),
we selected worker, ts and then the project starts

in this cloudfare will handle everything like all the routes will be handled by this only
and the look of the code is little bit different

routing wagera hai nhi toh aisa ugly code likhna padtha hai yeh hai issue isme

to deploy on workers we have to use wrangler of cloudfare which is like its cli
do npx wrangler login

it will get the creds of wrangler
run npm run deploy (the code is deployed on the internet)
express does not work with the workers it worked with the node and earlier it used to not work with bun also but later it started working

for re deploy also it will not take time and if you just do npm run deploy it will work (helps to push the code fastly)

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

Initialize a worker
npm create cloudflare -- my-app

Select no for Do you want to deploy your application

Explore package.json dependencies
"wrangler": "^3.0.0"

Notice express is not a dependency there
Start the worker locally
npm run dev

How to return json?

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

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
});
```

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

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/)
notion image

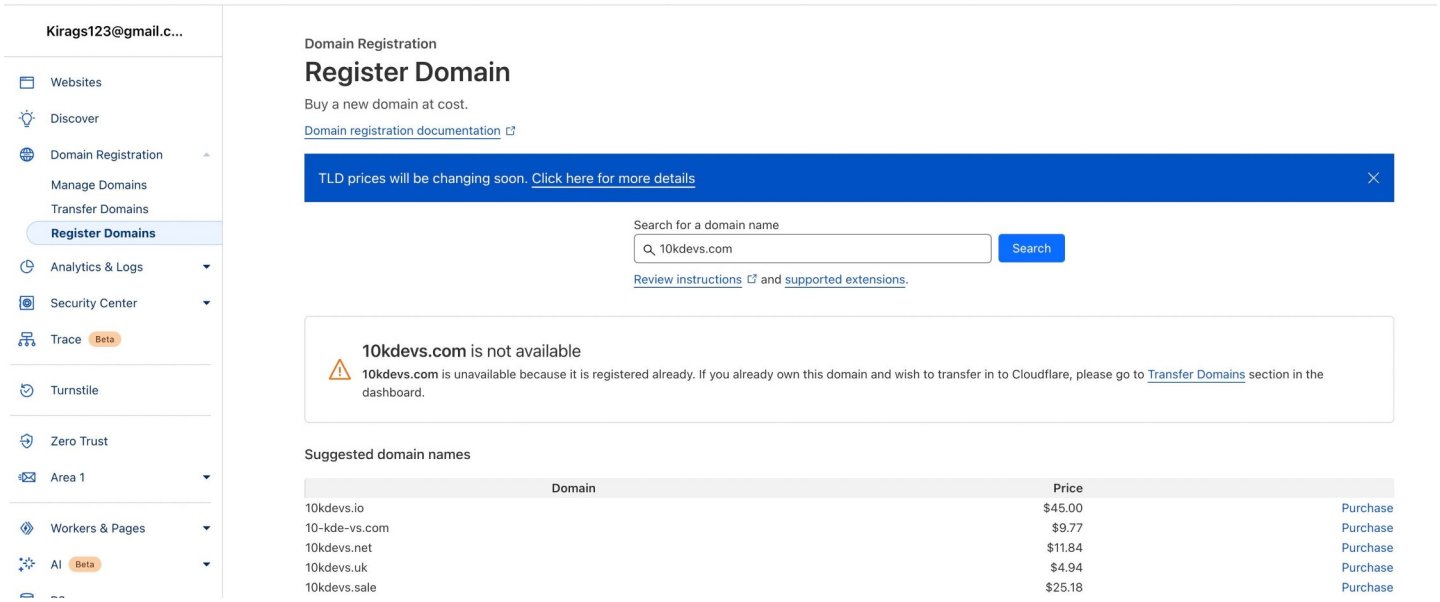
Step 1 - Login to cloudflare via the wrangler cli
Step 2 - Deploy your worker

Wrangler (command line)

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

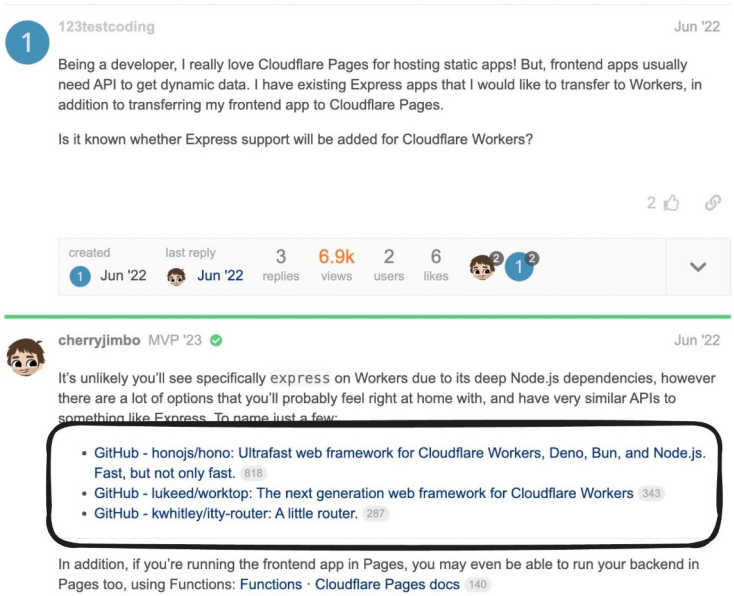
- Install/Update Wrangler: Get started by installing Wrangler, and update to newer versions by following this guide.
- API: An experimental API to programmatically manage your Cloudflare Workers.
- Bundling: Review Wrangler's default bundling.
- Commands: Create, develop, and deploy your Cloudflare Workers 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.

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



Adding express to it

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



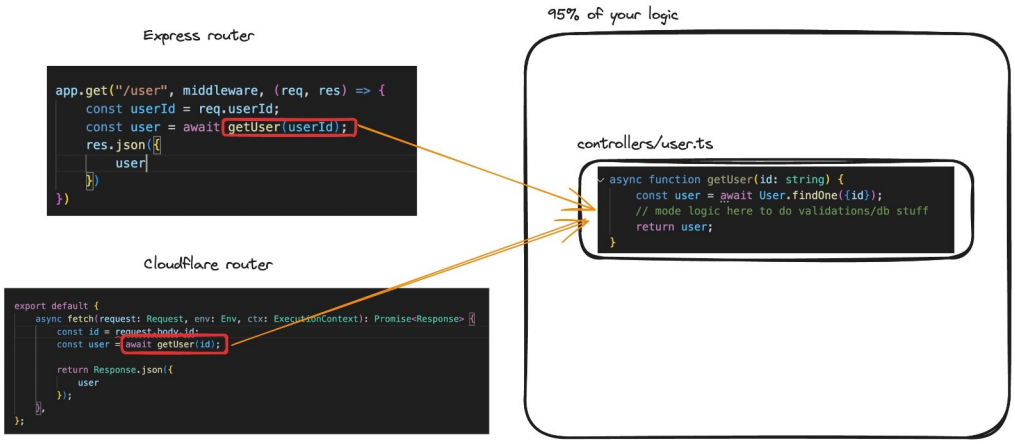
Reason 1 - Express heavily relies on Node.js

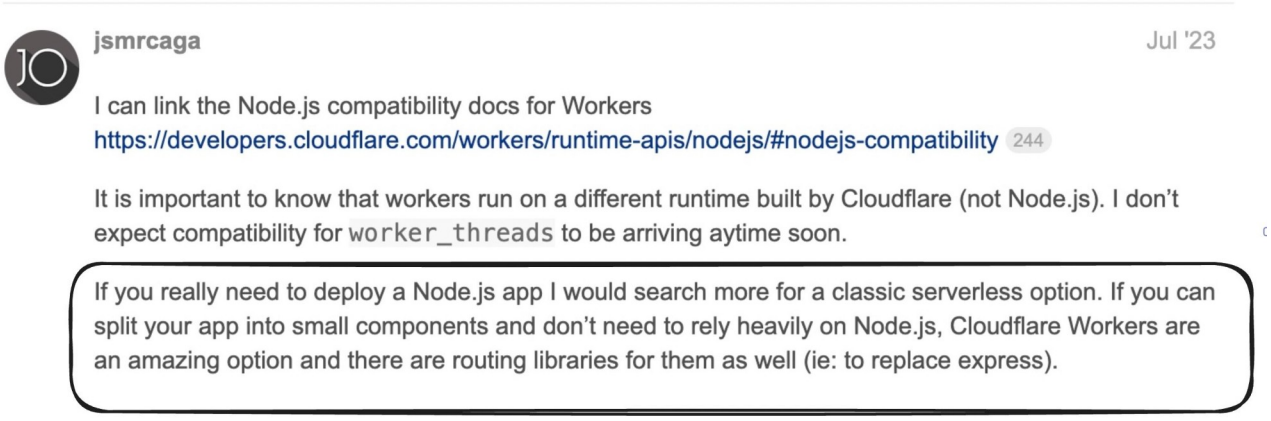
https://community.cloudflare.com/t/express-support-for-workers/390844

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`





TO FIX THAT WE USE HONO FOR THAT
SO HONO IS VERY HELPFULL AS ITS VERY SIMILAR TO EXPRESS AND IT DOES NOT HAVE APP.LISTEN(3000)

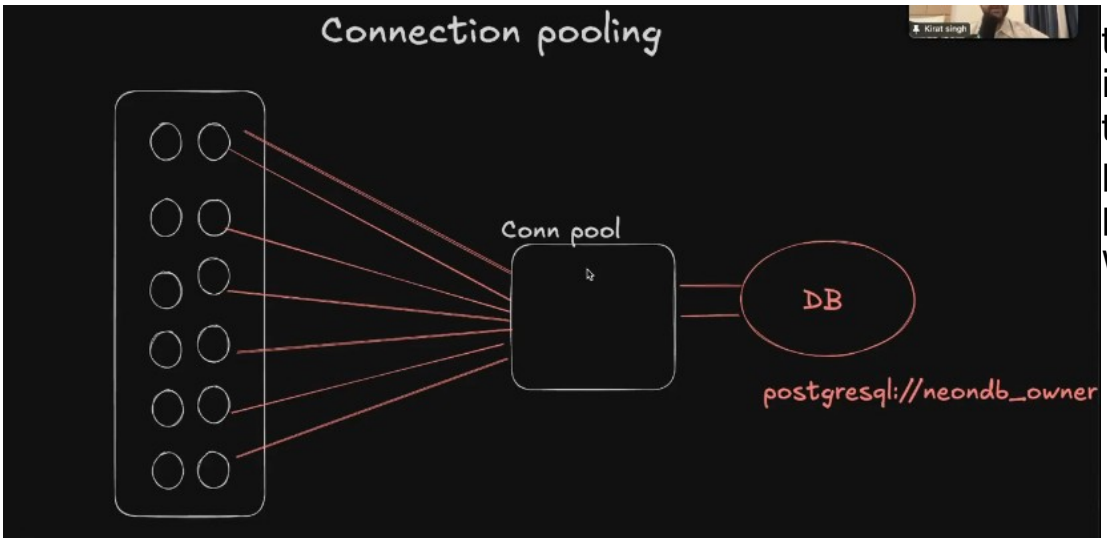
https://hono.dev/docs/

and in hono we get like single argument like in express we used to get req, res we get here only c
it supports everything, fe, be , react type code, middlewares etc... everything

basically a fullstack application

npm create hono@latest todo-app-latest

WHAT IS CONNECTION POLLING?
HOW DOES PRISMA MAKES MONEY THO?
BCOZ OF CONNECTION POLLING

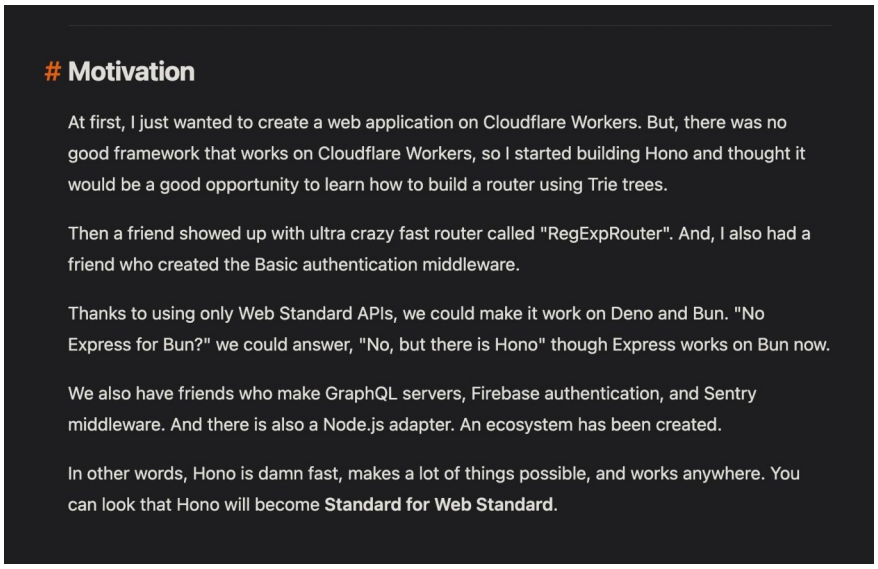


this option if you tick in connection pooling then its ok bcoz prisma used to use prisma acclerate which costs money

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.

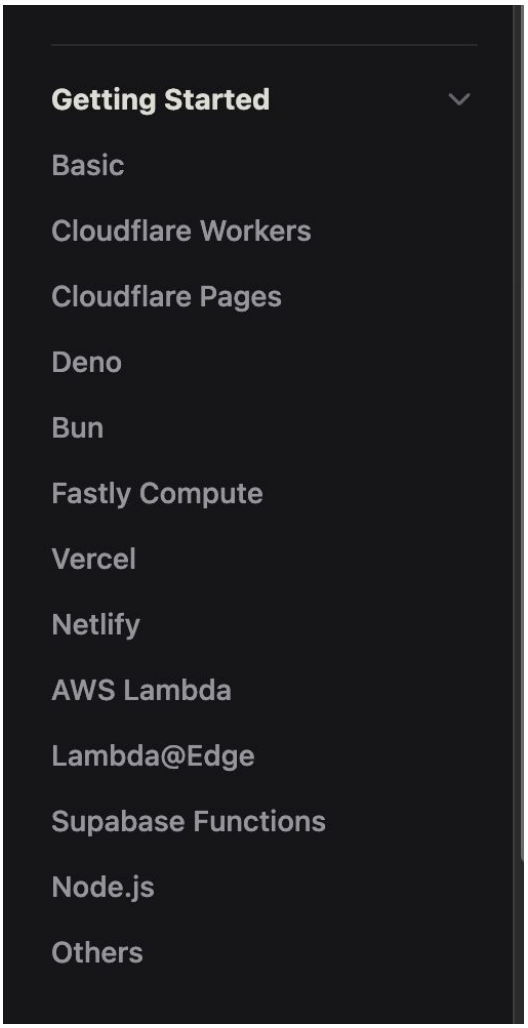
Thanks to using only Web Standard APIs, we could make it work on Deno and Bun. "No Express for Bun?" we could answer, "No, but there is Hono" though Express works on Bun now.

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?

Side photo



Working with cloudflare workers -

1. Initialize a new app

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

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

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

1. Hello World

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

app.get('/', (c) => c.text('Hello Cloudflare Workers!'))

export default app
```

Getting inputs from user

```
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("Authorization"));
  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
```

Middlewares

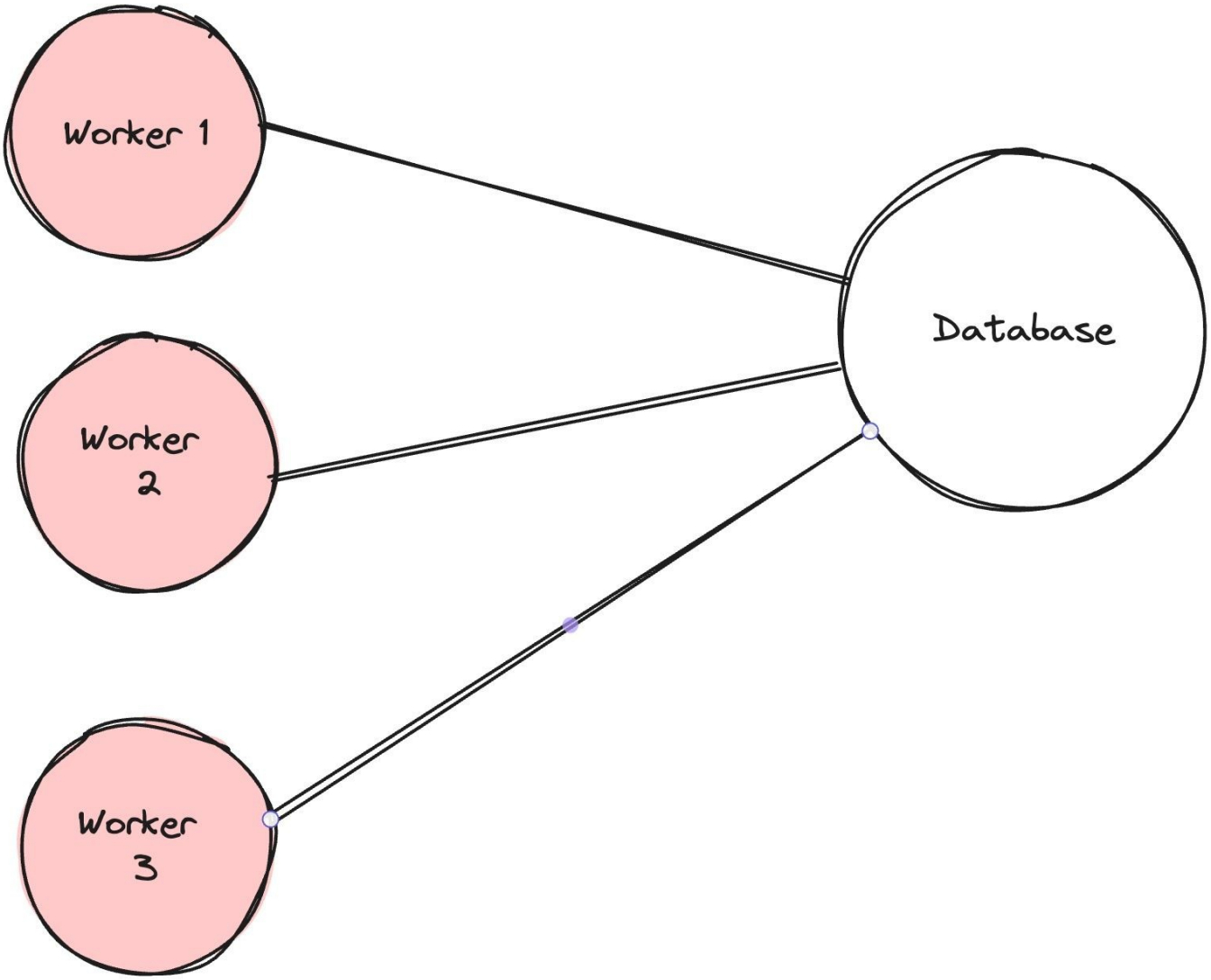
💡

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
```

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

```
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
}
```

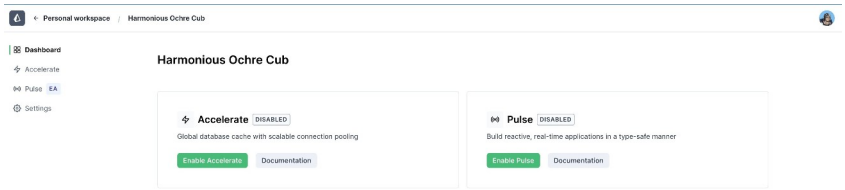
4. Create migrations

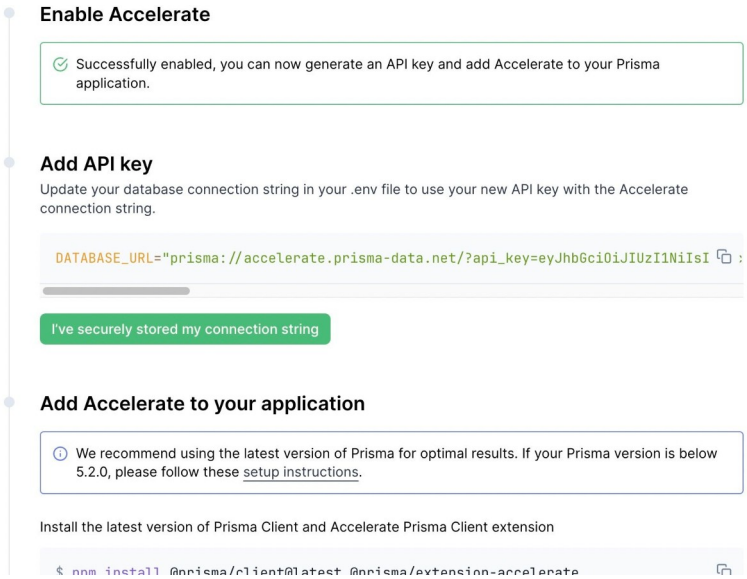
```
npx prisma migrate dev --name init
```

5. Signup to Prisma accelerate

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

Enable accelerate





Generate an API key

Replace it in .env

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

5. Add accelerate as a dependency

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

6. Generate the prisma client

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

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/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<{ 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
```