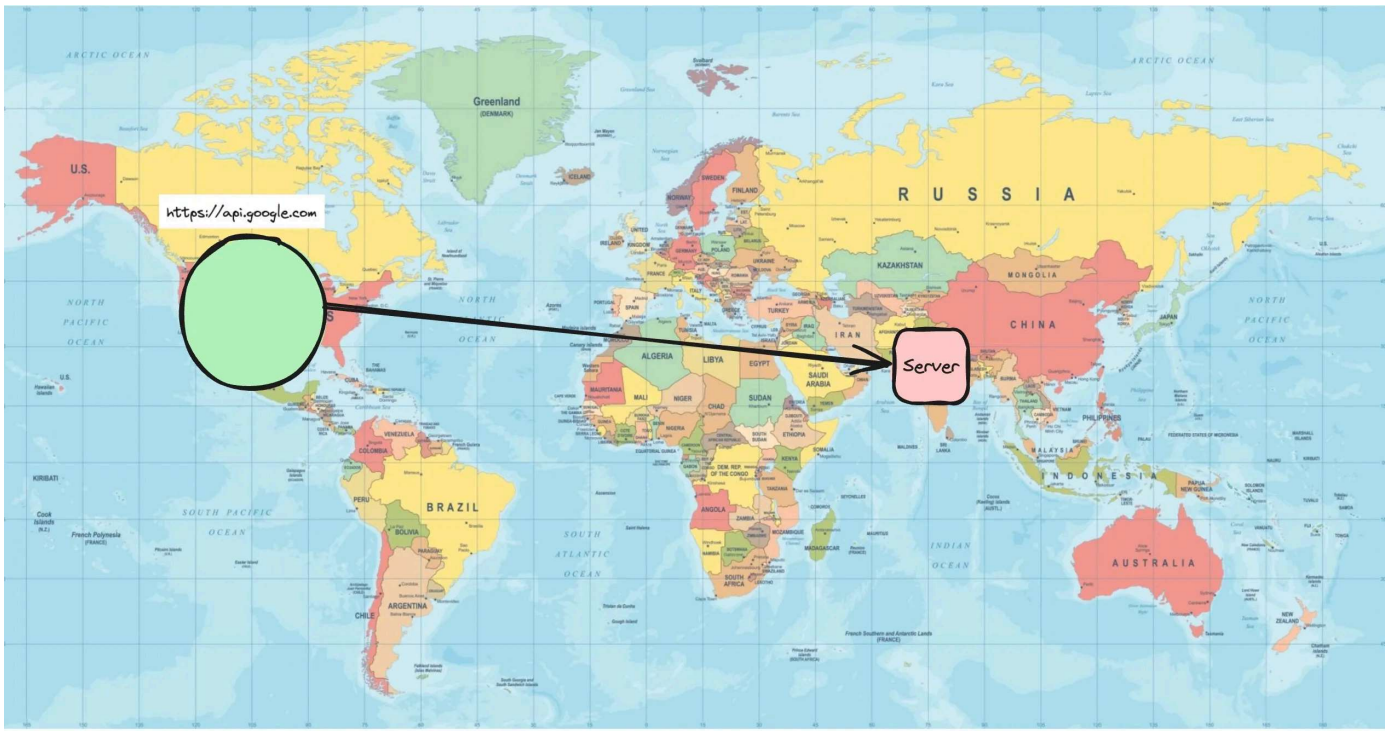
What are backends servers?



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
- Google Cloud Functions
- Cloudflare 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

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

Isolates vs containers

Initializing a worker

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

- Initialize a worker
- Explore package.json dependencies
- Start the worker locally
- How to return json?

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<Resp
    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"
          });
        }
      },
    };
```

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/)

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

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

Using hono

What is Hono

https://hono.dev/concepts/motivation

What runtimes does it support?

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
```