

Additional costs apply for AMIs with pre-installed software

Key pair (login) Info

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - required
pallab-key-pair [Create new key pair](#)

Network settings Info [Edit](#)

Network Info
vpc-0ea7d70ae4483754f

Subnet Info
No preference (Default subnet in any availability zone)

Auto-assign public IP Info
Enable

Firewall (security groups) Info
A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

Create security group Select existing security group

We'll create a new security group called 'launch-wizard-1' with the following rules:

- Allow SSH traffic from
Helps you connect to your instance. Anywhere (0.0.0.0/0)
- Allow HTTPS traffic from the internet
To set up an endpoint, for example when creating a web server.
- Allow HTTP traffic from the internet
To set up an endpoint, for example when creating a web server.

⚠ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only. ✕

Configure storage Info [Advanced](#)

1x 8 GiB gp3 Root volume, 3000 IOPS, Not encrypted

[Add new volume](#)

The selected AMI contains instance store volumes, however the instance does not allow any instance store volumes. None of the instance store volumes from the AMI will be accessible from the instance.

🔄 Click refresh to view backup information
The tags that you assign determine whether the instance will be backed up by any Data Lifecycle Manager policies.

0 x File systems [Edit](#)

Summary

Number of instances | Info
1

Software Image (AMI)
Canonical, Ubuntu, 24.04, amd64...read more
ami-073130f74f5ffb161

Virtual server type (instance type)
t3.micro

Firewall (security group)
New security group

Storage (volumes)
1 volume(s) - 8 GiB

[Cancel](#) [Launch instance](#) [Preview code](#)

© 2025, Amazon Web Services, Inc. or its affiliates. [Privacy](#) [Terms](#) [Cookie preferences](#)

TO TAKE ACCESS:

we first check the name of the user using
whoami

```
icacls "D:\Data\Save\Keys\pallab-key-pair.pem" /grant  
desktop-abc123\pallab:F
```

```
icacls "D:\Data\Save\Keys\pallab-key-pari.pem"  
/remove "BUILTIN\Users"  
icacls "D:\Data\Save\Keys\pallab-key-pari.pem"
```

output: desktop-abc123\pallab:(F)

```
ssh -i pdkey.pem ubuntu@13.48.10.208
```

```
ssh -i D:\Data\Save\Keys\pallab-key-pari.pem ec2-  
user@your-ip  
done
```

```
then make a new folder and add  
run sudo apt install nodejs  
sudo apt install npm  
npm init -y  
npm i express
```

```
lsof = what is running on what port  
lsof -i :3000 = to know what all processes running on  
the port 3000  
kill 300 to know the port at which the node js file  
is running
```

```
sudo npm I -g pm2
```

```
pm2 start app.js is the file we are running
```

```
now when we again try to find the 3000 and try to  
kill  
it will still be online
```

```
and curl localhost:3000 will give me the output  
curl command == this is like a post command from the  
postman
```

```
pm2 stop 0
```

```
pm2 ls
```

```
pm2 examples == shows all the commands
```

Continuous Integration

Continuous Integration (CI) is a development practice where developers frequently integrate their code changes into a shared repository, preferably several times a day. Each integration is automatically verified by

1. Building the project and
2. Running automated tests.

This process allows teams to detect problems early, improve software quality, and reduce the time it takes to validate and release new software updates.

ITS SIMPLY LIKE PASSING ALL THE CHECKS IF YES THEN THE DEVLOPER WILL LOOK INTO YOUR PR AND TRY TO SOLVE IT AND IF NOT IT WILL SHOW ERROR AFTER THAT

ITS LIKE WHENEVER SOMEONE PUSHES CODE TO MAIN BRANCH THEN IT SHOULD BE DEPLOYED TO THE SERVER

```
Continuous-Integration:

name: Performs linting, formatting on the application
runs-on: ubuntu-latest
steps:
  - name: Checkout the Repository
    uses: actions/checkout@v3

  - name: Setup pnpm
    run: npm install -g pnpm

  - name: Install Dependencies
    run: pnpm install

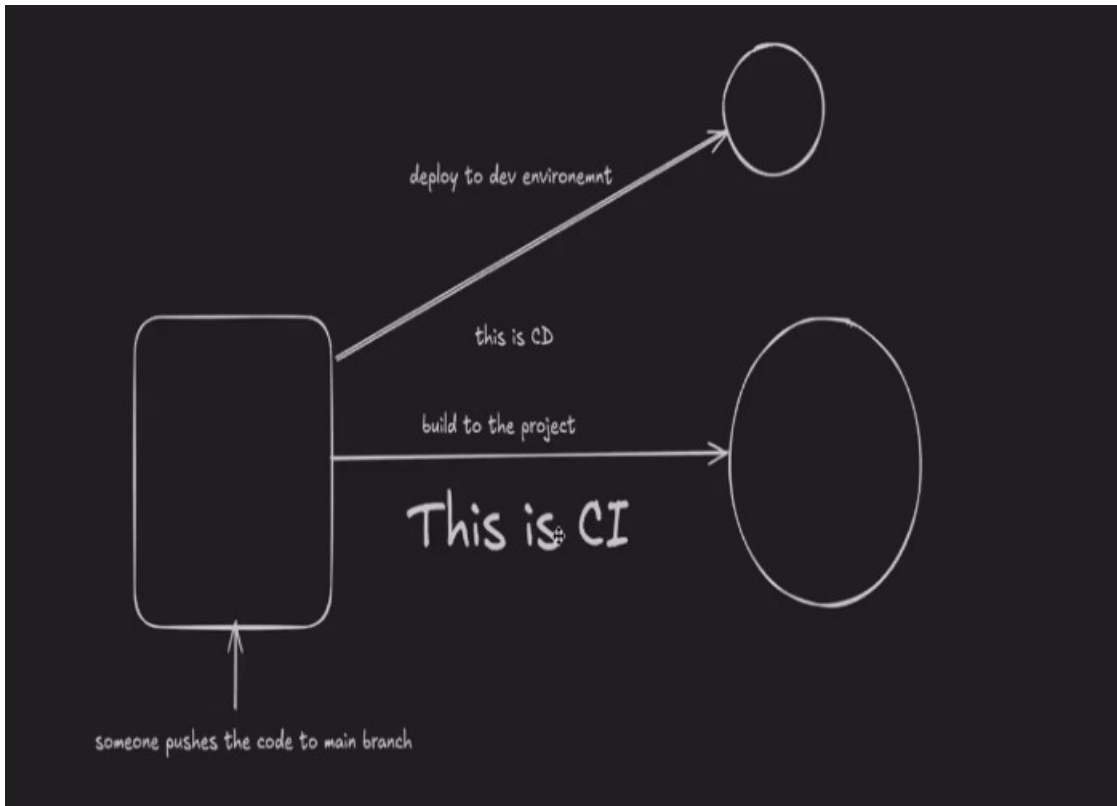
  - name: Run linting check
    run: pnpm run lint:check

  - name: Check formatting
    run: pnpm run format:check
```

Continuous Deployment

As the name suggests, deploying your code continuously to various environments (dev/stage/prod)

THIS HAS LIKE THE STAGES DEV AND PROD
THE WORKING PART IS FIRST DONE IN DEV REPO AND IS
HIGHLY UNSTABLE



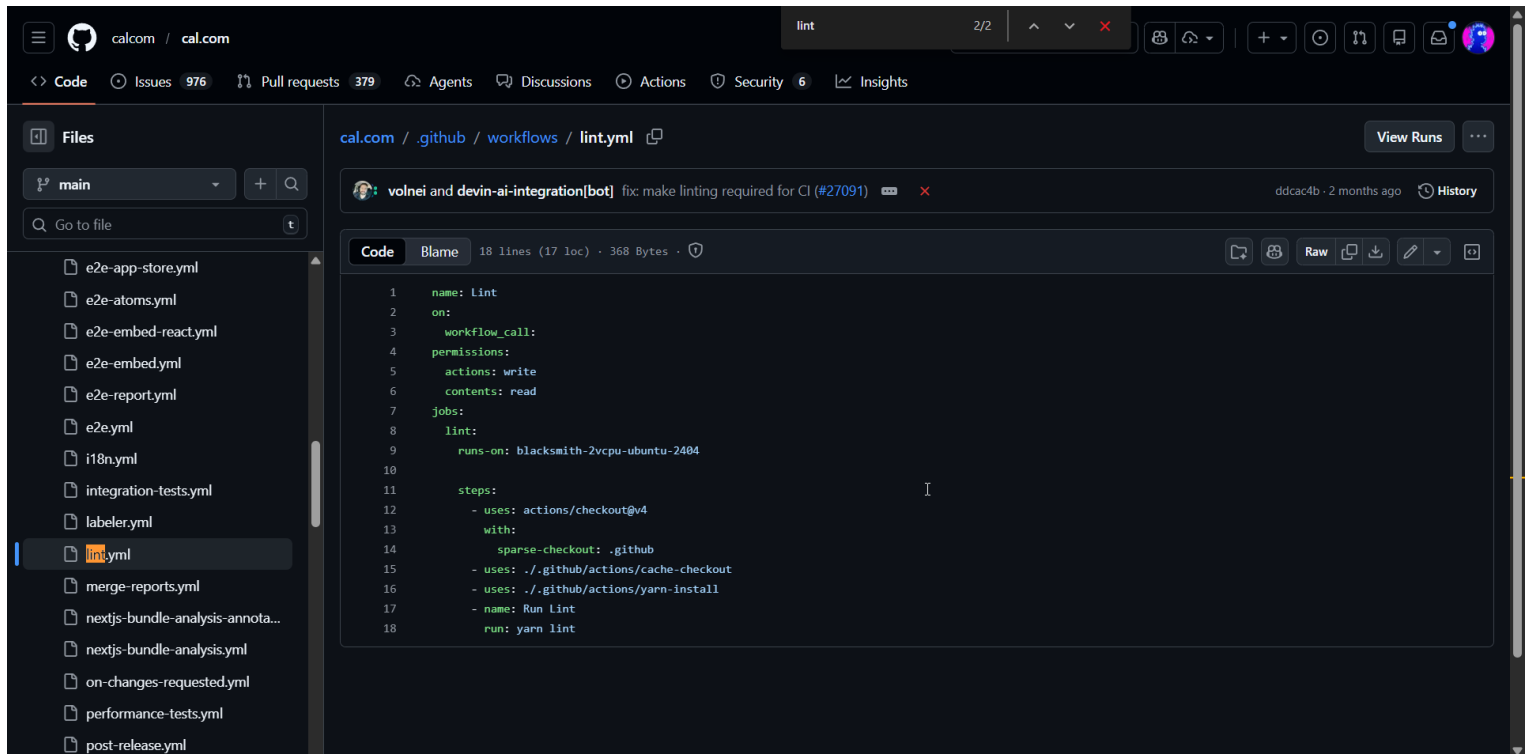
LINTING MEANING :::

This means when we add some checks to the code like the callbacks should follow this method, the async function should be used like this, its like defining a format and then everyone should follow it
eg: eslint

Some special files in github

In a codebase if a file is inside of `.github/workflows`

It is considered a CI file which means github will run the contents of the files whenever the conditions are met.



The screenshot shows a GitHub repository for 'cal.com'. The file path is `.github/workflows/lint.yml`. The file content is as follows:

```
1 name: Lint
2 on:
3   workflow_call:
4   permissions:
5     actions: write
6     contents: read
7 jobs:
8   lint:
9     runs-on: blacksmith-2vcpu-ubuntu-2404
10
11 steps:
12   - uses: actions/checkout@v4
13     with:
14       sparse-checkout: .github
15   - uses: ./github/actions/cache-checkout
16   - uses: ./github/actions/yarn-install
17   - name: Run Lint
18     run: yarn lint
```


this CD is like checks whether the code follows the things listed and if not it will not add the code and if yes then the code will be added

this can also like run py files, convert ts to js and also in this github runs its own ec2 machines to verify the code

MAIN USECASE:

1. whenever a pr is raised , we have to check whether the code inside the pr is correct or not, then we build that project
2. then deploy it to check if functional or not

FOR WHAT CI/CD IS USED FOR?

- to test the code if passing or failing the test cases
- to deploy the code to somewhere
- to test linting and formatting errors

to add the file in the ubuntu

use git clone

npm i

<https://github.com/appleboy/ssh-action>

add the code from the above to the yml file

```
name: Execute remote SSH commands using password
uses: appleboy/ssh-action@v1
with:
  host: ${ secrets.HOST }}
  username: ${ secrets.USERNAME }}
  password: ${ secrets.PASSWORD }}
  port: ${ secrets.PORT }}
  script: whoami
```

to need the secrets and all we need to access that from the website for ssh key in the ec2 machine

```
npm install -g pnpm
```

1. Deploying a monorepo (http, ws, prisma, postgres, next)
2. env variables
3. dev vs prod environments, periodic releases
4. Testing in CI pipelines
5. Cert management
6. CD pipeline to refresh certs every month

created the http, ws-server, setting up prisma in package as it will be used by everyone

making the folder and then initializing npm
`init -y`

make changes in the package.json file
change the name
remove the script

```
npx tsc --init
```

create this and then just change the json

The screenshot shows a VS Code editor with a `package.json` file open. The file content is as follows:

```
1 {
2   "name": "@repo/db",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "keywords": [],
7   "author": "",
8   "devDependencies": {
9     "@repo/typescript-config": "workspace:*"
10  },
11  "license": "ISC",
12  "type": "commonjs"
13 }
14
```

The Explorer sidebar on the right shows a project structure with folders like `apps`, `packages`, and `prisma`. The `prisma` folder contains `package.json` and `tsconfig.json`.

The terminal window at the bottom shows the command `npx tsc --init` being executed, with the output: `Created a new tsconfig.json`. Below the terminal, there is a link: `You can learn more at https://aka.ms/tsconfig`.

we have to extend from the typescript folder base.json file (takki yeh code hame khud na likhna pade)

```
{
  "extends" : "@repo/typescript-config/base.json"
}
```

this we have to write in the tsx file

in the above image of package.json when we use node we will add:

* in the dev dependencies otherwise for pnpm workspace:* for node

pnpm add prisma (prisma is a orm as we know, easier to connect to db)

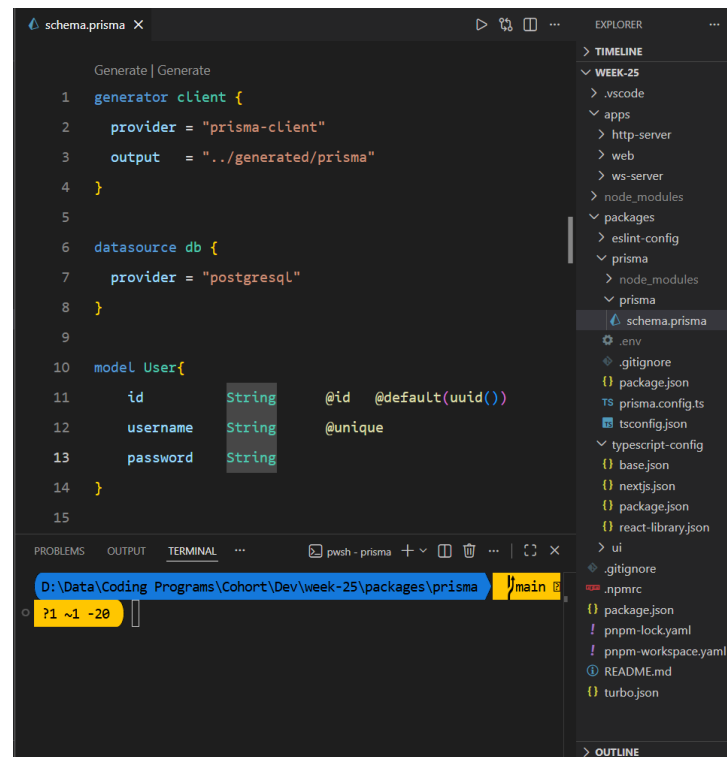
npx prisma init

then making changes to the schema.prisma

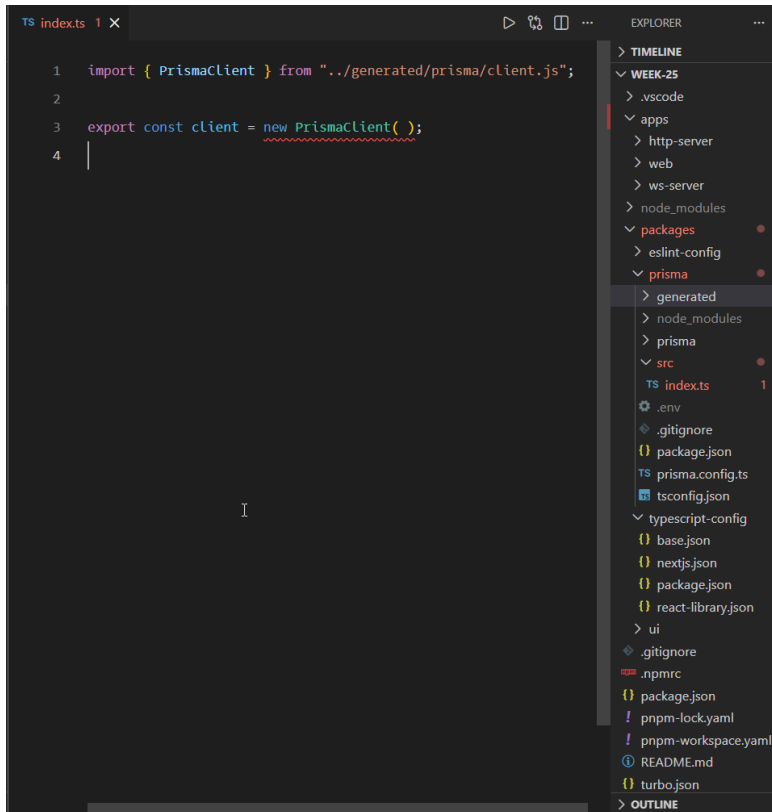
then get a db from neon.tech

then add the db url in the env file

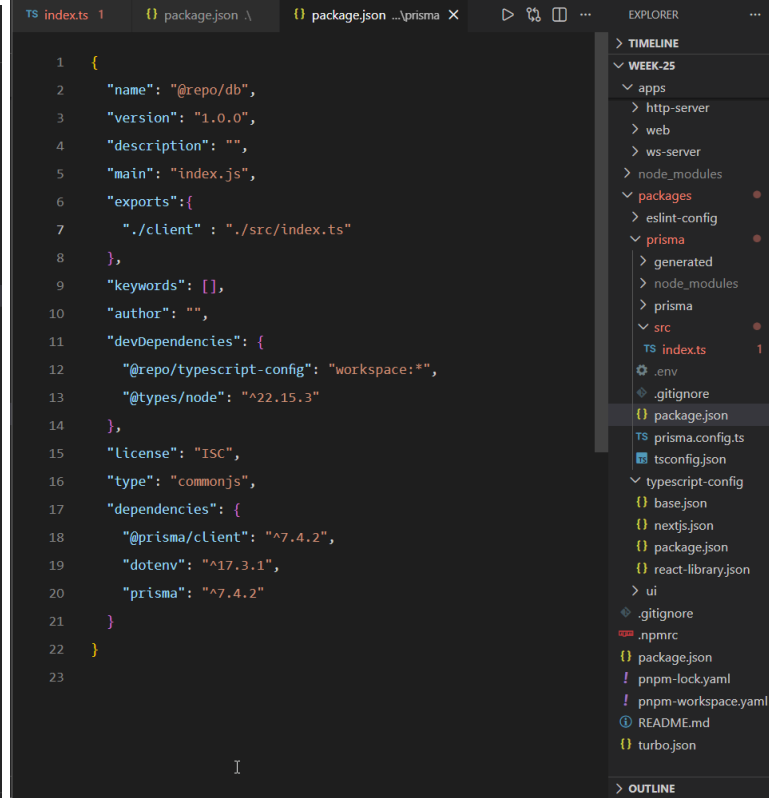
use azure aws is shit for neon tech
then do npx prisma migrate dev
enter a name for the migration
npx prisma generate npm install dotenv



now to export this primsa folder you have to create it a package
then create a src folder inside the prisma folder



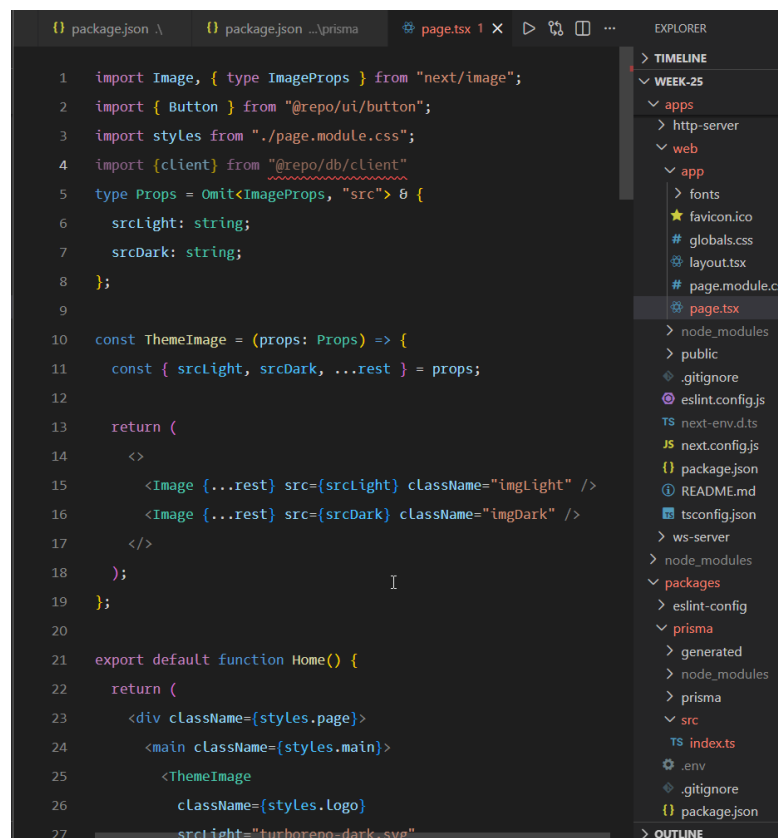
```
1 import { PrismaClient } from "../generated/prisma/client.js";
2
3 export const client = new PrismaClient();
4
```



```
1 {
2   "name": "@repo/db",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "exports": {
7     "./client": "./src/index.ts"
8   },
9   "keywords": [],
10  "author": "",
11  "devDependencies": {
12    "@repo/typescript-config": "workspace:*",
13    "@types/node": "^22.15.3"
14  },
15  "license": "ISC",
16  "type": "commonjs",
17  "dependencies": {
18    "@prisma/client": "^7.4.2",
19    "dotenv": "^17.3.1",
20    "prisma": "^7.4.2"
21  }
22 }
23
```

add the index.ts code and also add a exports section in the src package.json file

now just import it in the page.tsx folder using the command
and also
dub.sh follows the same method



```
1 import Image, { type ImageProps } from "next/image";
2 import { Button } from "@repo/ui/button";
3 import styles from "../page.module.css";
4 import { client } from "@repo/db/client"
5 type Props = Omit<ImageProps, "src"> & {
6   srcLight: string;
7   srcDark: string;
8 };
9
10 const ThemeImage = (props: Props) => {
11   const { srcLight, srcDark, ...rest } = props;
12
13   return (
14     <>
15       <Image {...rest} src={srcLight} className="imgLight" />
16       <Image {...rest} src={srcDark} className="imgDark" />
17     </>
18   );
19 };
20
21 export default function Home() {
22   return (
23     <div className={styles.page}>
24       <main className={styles.main}>
25         <ThemeImage
26           className={styles.logo}
27           srcLight="turborepo-dark.svg"
28         />
29       </main>
30     </div>
31   );
32 }
```

```
{ } package.json ...\prisma  page.tsx 1  {} package.json ...\web  X  EXPLORER  ...
4  "type": "module",
5  "private": true,
  > Debug
6  "scripts": {
7    "dev": "next dev --port 3000",
8    "build": "next build",
9    "start": "next start",
10   "lint": "eslint --max-warnings 0",
11   "check-types": "next typegen && tsc --noEmit"
12 },
13 "dependencies": {
14   "@repo/ui": "workspace:*",
15   "next": "16.1.5",
16   "react": "^19.2.0",
17   "react-dom": "^19.2.0"
18 },
19 "devDependencies": {
20   "@repo/eslint-config": "workspace:*",
21   "@repo/typescript-config": "workspace:*",
22   "@repo/db": "workspace:*",
23   "@types/node": "^22.15.3",
24   "@types/react": "19.2.2",
25   "@types/react-dom": "19.2.2",
26   "eslint": "^9.39.1",
27   "typescript": "5.9.2"
28 }
29 }
30
```

add the repo/db line in the dependency to import that also

and to get all the libraries you have to do
pnpm install
using this the red squiggle of the repo will go away
as it will download the things we require

now the prisma can be used in the web project and we
have to do in both the projects

http-server

ws-server

```
for http server
npm init -y
npx tsc -init
add this in package.json also
```

```
"scripts": {
  "build": "tsc-b",
  "dev" : "npm run build && npm run start",
  "start": "node dist/index.js"
},
"devDependencies": {
  "@repo/db": "workspace:*",
  "@repo/typescript-config": "workspace:*"
},
```

we have to extend from the typescript folder
base.json file (takki yeh code hame khud na likhna
pade)

```
{
  "extends" : "@repo/typescript-config/base.json",
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

for the tsconfig file

we have to do same for both http and the ws-server

setup is done time to write the code

First for http server

add

```
pnpm add express @types/express
```

you can also reload the ts server

for ws server

```
pnpm add ws @types/ws
```

client is used to just connect to db

restart ts server

```
npx install -g pnpm
npx create-turbo@latest
```

now removing the dev folder and adding the http, ws-server and then creating prisma folder in the packages

in prisma folder initialize

```
npm init -y
```

in the package.json file changing the name to "name": "@repo/db", and remove the script section

```
"dependencies": {
  "@repo/typescript-config": "workspace:*"
},
```

and this workspace:* is for pnpm
for npm its only *

```
npx tsc -init
```

then remove everything and add this

```
{
  "extends": "@repo/typescript-config/base.json"
}
```

to reduce the copy thing we just using the existing base.json file

then do pnpm add prisma (ORM for the day - lets to connect to the db, gives objects to conencting to the db)

then do `npx prisma init`(this will initialize `prisma.schema`)

add the schema in the schema file

then setup `neon.tech`(use azure it connects man)

then add the db link in the env file

```
npx prisma migrate dev
```

```
npx prisma generate
```

now to export we have to do smthg

now in the `packages/prisma/src/index.ts` create this and then add the export thingy in this

in the `index.ts` add this

```
import { PrismaPg } from '@prisma/adapter-pg'
import { PrismaClient } from
'../generated/prisma/client.js'
```

```
const adapter = new PrismaPg({ connectionString:
process.env.DATABASE_URL })
```

```
export const client = new PrismaClient({ adapter })
```

and then install

```
pnpm add prisma --save-dev
```

```
pnpm add @prisma/client @prisma/adapter-pg pg dotenv
```

```
pnpm i --save-dev @types/node
```

now in the `prisma package.json` add this

```
"exports":{
  "./client":"./src/index.ts"
},
```

the above line helps us to use the prisma file in the codebase to use that we have write this in the project

for the next js project:

```
import {client} from "@repo/db/client";
```

and in the package.json add this

```
"@repo/db": "workspace:*
```

adding the package is just not enough you have to go to root folder and do pnpm install to make it work properly

now doing the same things in the http folder

```
npm init -y
```

```
npx tsc --init
```

add this in the package.json folder

```
"scripts": {
  "build": "tsc -b",
  "dev": "npm run build && npm run start",
  "start": "node dist/index.js"
},
"devDependencies": {
  "@repo/db": "workspace:*",
  "@repo/typescript-config": "workspace:*"
},
```

tsconfig file for the http server

```
{
  "extends": "@repo/typescript-config/base.json",
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

in the http folder create src folder with index.ts

same for the ws-server

```
npm init -y
```

```
npx tsc --init
```

add this in the package.json folder

```
  "scripts": {
    "build": "tsc -b",
    "dev": "npm run build && npm run start",
    "start": "node dist/index.js"
  },
  "devDependencies": {
    "@repo/db": "workspace:*",
    "@repo/typescript-config": "workspace:*"
  },
```

tsconfig file for the http server

```
{
  "extends": "@repo/typescript-config/base.json",
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

in the http folder create src folder with index.ts

NOW SETTING UP THE HTTP SERVER

```
pnpm add express @types/express
```

to fix the ts error use this

```
import * as express from "express";
```

to remove the red squiggle we have to restart the ts server

and the ts server restart only comes in the when we are in a ts file

do pnpm install in the root dir if any issues

NOW SETTING UP THE WS SERVER

```
pnpm add ws @types/ws
```

now to remove the error of the prisma for nextjs projects

make changes in this

```
generator client {  
  provider      = "prisma-client-js"  
  output        = "../generated/prisma"  
  previewFeatures = ["driverAdapters"]  
}
```

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

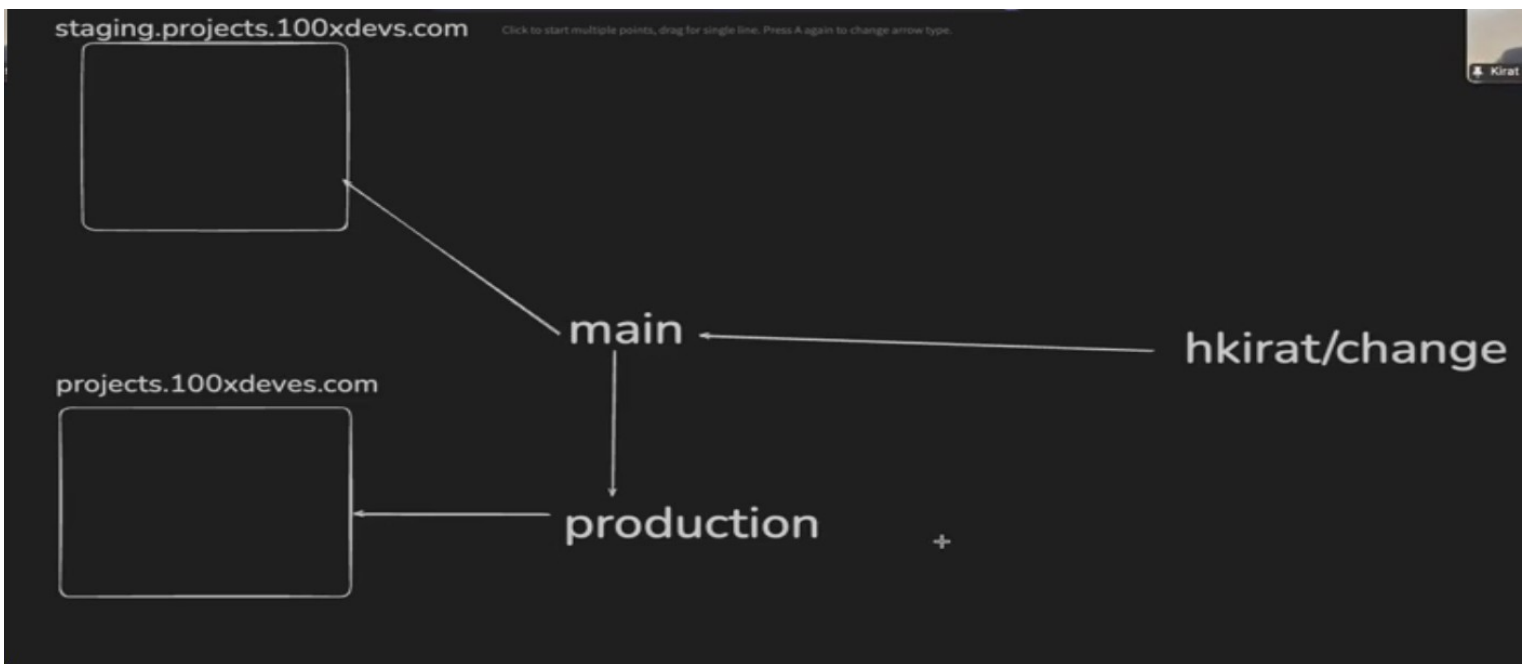
```
model User {  
  id      Int      @id @default(autoincrement())
```

```
  username String
  password String
}
```

```
pnpm remove prisma @prisma/client
pnpm add -D prisma@6.6.0
pnpm add @prisma/client@6.6.0
```

bss yahi sahab karke thik hoga

```
add this to nextjs ki configuration
const nextConfig = {
  transpilePackages: ["@repo/db/client"],
};
```



all the devs when pushes to main it shows the work on the staging website and once a week the work is pushed to the production repo and then the original websites sees the changes

creating another branch

`git checkout -b production`

now the main branch will push to the main website

production branch will push to production website

DEPLOYING THE MONOREPO SERVER

1. Create 2 servers
2. Add node, nginx to both the servers
3. Clone the monorepo to both the servers
4. Start 3 processes (next, ws, http)
5. Point our Domain names to the respective servers
week-25-http.100xdevs.com
week-25-ws.100xdevs.com
week-25-fe.100xdevs.com

staging.week-25-http.100xdevs.com
staging.week-25-ws.100xdevs.com
staging.week-25-fe.100xdevs.com
6. Refresh nginx config
7. Test that everything is working

setup two machines and add node and nginx

one more step in this install
pnpm

STEP 1

To install node

<https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04>

use from step 3 of this

curl -o-

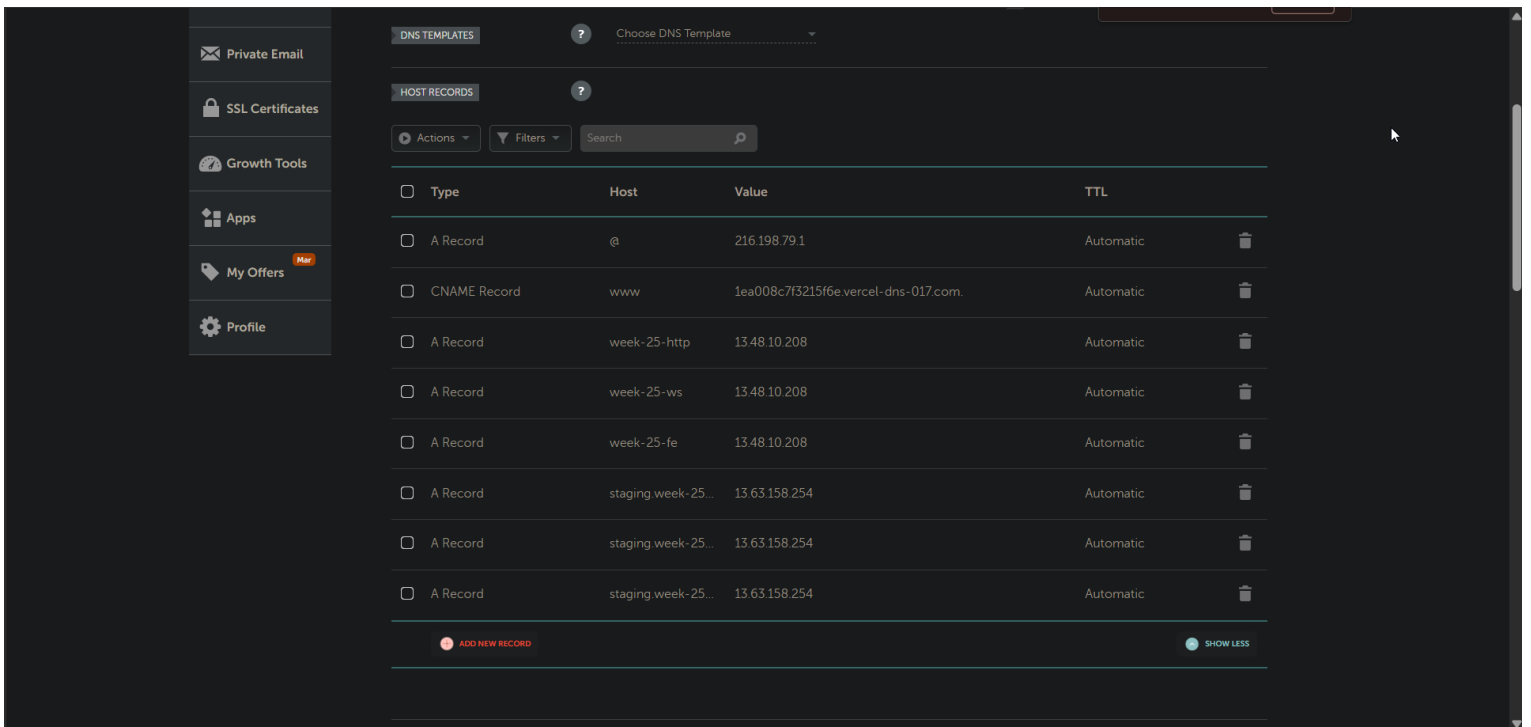
<https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh> | bash

1. Deploying a monorepo (http, ws, prisma, postgres, next)
2. env variables
3. dev vs prod environments, periodic releases
4. Testing in CI pipelines
5. Cert management
6. CD pipeline to refresh certs every month
7. CD pipeline to copy the prod DB to dev every day

```
source ~/.bashrc
nvm install -lts
```

```
to check type node
const x = 1
console.log(x)
```

```
STEP 2 nginx
sudo apt update
sudo apt install nginx
```



added these ips for prod and dev

```
npm install -g npm@11.11.0
```

STEP 3

clone the repo

STEP 4

now to start the fe, ws, and http servers
do pnpm install

```
sudo fallocate -l 2G /swapfile  
sudo chmod 600 /swapfile  
sudo mkswap /swapfile  
sudo swapon /swapfile
```

```
free -h
```

```
pnpm install --frozen-lockfile
```

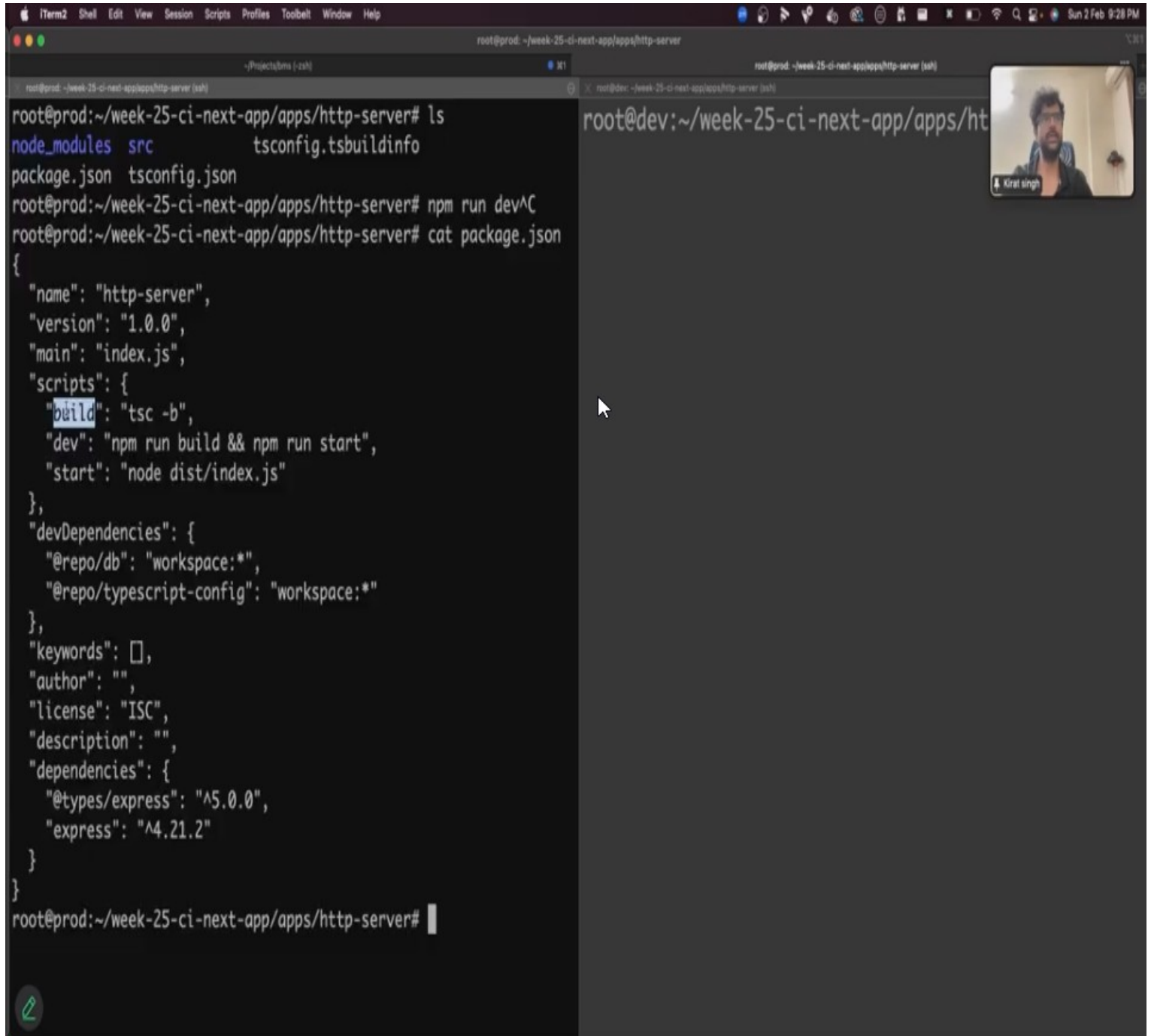
also need to create two db for dev and production

then add the db url to the .env file by going to
package/primsa/.env file and adding the urls

and then in both the machines do
npx prisma migrate dev

now after this go to http folder and should I run npm run dev?

No the start command is for production use that but before that run build



```
root@prod: ~/week-25-ci-next-app/apps/http-server
root@prod: ~/week-25-ci-next-app/apps/http-server# ls
node_modules  src          tsconfig.tsbuildinfo
package.json  tsconfig.json
root@prod: ~/week-25-ci-next-app/apps/http-server# npm run dev^C
root@prod: ~/week-25-ci-next-app/apps/http-server# cat package.json
{
  "name": "http-server",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "build": "tsc -b",
    "dev": "npm run build && npm run start",
    "start": "node dist/index.js"
  },
  "devDependencies": {
    "@repo/db": "workspace:*",
    "@repo/typescript-config": "workspace:*"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "@types/express": "^5.0.0",
    "express": "^4.21.2"
  }
}
root@prod: ~/week-25-ci-next-app/apps/http-server#
```

pnpm run build

this check passes

iske baad test karne kae liye teeno process thodi chalengye iske liye use hoga process management

pm2 huhuhu

```
npm install -g pm2
```

```
pm2 start npm --name "http-server" -- start
```

```
pm2 start npm --name "ws-server" -- start
```

```
pm2 start npm --name "fe-server" -- start
```

this helps in keep the server running

pm2 list

do this same in dev server

to make this work need to open these ports also

The screenshot shows the AWS Management Console interface. The top navigation bar includes the AWS logo, search bar, and user information (PDGamerSG). The left sidebar shows the navigation menu with categories like EC2, Images, Elastic Block Store, Network & Security, and Load Balancing. The main content area displays the 'Instances (1/2)' page, showing a table of EC2 instances. Two instances are listed: 'prod' (Instance ID: i-0ea1f65e1135f7559) and 'dev' (Instance ID: i-062a2d77a5ef66bb3). Both are in a 'Running' state. Below the table, the details for the 'prod' instance are shown, including the 'Security' tab. The 'Security details' section shows the IAM role, Owner ID, and Launch time. The 'Inbound rules' section shows a table of security group rules.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP	IPv6 IPs
prod	i-0ea1f65e1135f7559	Running	t3.micro	3/3 checks passed	View alarms +	eu-north-1b	ec2-13-48-10-208.eu-n...	13.48.10.208	-	-
dev	i-062a2d77a5ef66bb3	Running	t3.micro	3/3 checks passed	View alarms +	eu-north-1a	ec2-13-63-158-254.eu-...	13.63.158.254	-	-

Name	Security group rule ID	Port range	Protocol	Source	Security groups	Description
-	sgr-080e29a2b233edc22	443	TCP	0.0.0/0	launch-wizard-5	-
-	sgr-07e9a9be65e157654	22	TCP	0.0.0/0	launch-wizard-5	-
-	sgr-0ba068f094be8a500	80	TCP	0.0.0/0	launch-wizard-5	-

BY CLICKING ON THE LAUNCH WIZARD YOU HAVE TO ADD THE 3000, 3001, 3002 PORT TO MAKE IT WORK ON THE URL OTHERWISE THE MACHINE URL+ THE PORT WILL NOT WORK you can check dns using check dns propagation

<http://staging.week-25-http.pallabdas.me:3000,3001,3002>

our application is deployed now but write now the nginx is not set up that's why we have to use the port and if we change the starting port that does not matter that much

to configure that write

```
sudo vi /etc/nginx/nginx.conf
```

to remove everything is this use :%d

```
events {
    # Event directives ...
}

http {
    server {
        listen 80;
        server_name staging.week-25-ws.pallabdas.me;
        location / {
            proxy_pass http://localhost:3001;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
    server {
        listen 80;
        server_name staging.week-25-http.pallabdas.me;
        location / {
            proxy_pass http://localhost:3002;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
    server {
        listen 80;
        server_name staging.week-25-fe.pallabdas.me;
        location / {
            proxy_pass http://localhost:3000;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
}
```

after adding the nginx code like this just use this
sudo nginx -s reload

now all the routes will start working

<http://staging.week-25-http.pallabdass.me/>

<http://week-25-http.pallabdass.me/>

just like this replace the http with ws, fe etc
DAMN BRO

NOW THE DEPLOYING OF THE MONOREPO IS DONE LOL JUST THE FIRST STEP

signed commit = this is like in github like with a name just that person can commit not anyone else

**now we have to add rules to production branch so that no one else can commit to production branch
THIS IS CALLED AS BRANCH PROTECTION RULES**

go to github with production branch open settings
go to branches
go to ruleset name
add a target branch by clicking on the include by pattern and then add the production branch

tick restrict deletion

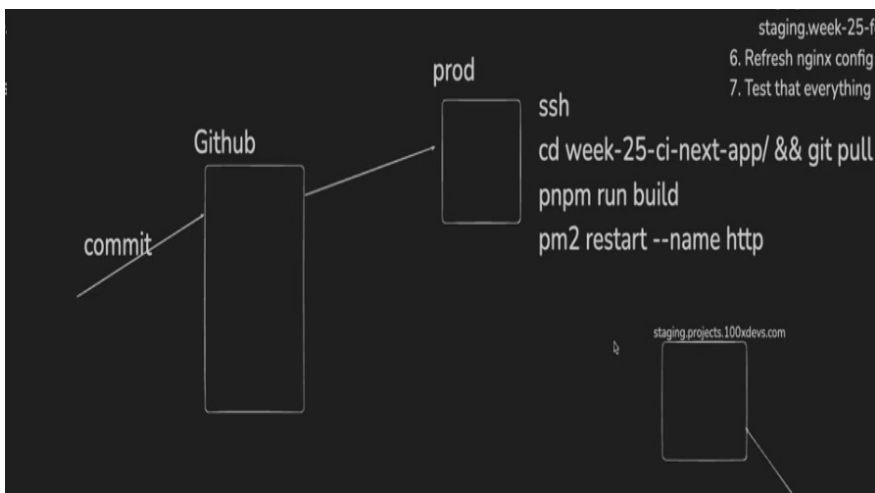
tick signed commits (only pallab can commit as pallab)

Require a pull request before merging (we can add like after 2 people has approved it can be added)

MANUALLY DEPLOYMENT IS ALL DONE

Now if you want to create a continuous deployment pipeline created a `.github/workflows/cd_prod.yml`, `cd_staging.yml`

now like imagine you made some changes in the repo and then you want to deploy that to production server so first you will do is `git pull`



then run `pnpm run build` and then restart the `pm2` server for the changes to see

`pm2 restart fe-server`

if already running then

if doing again then you have to change the ip in the a records and also start the `pm2` again if once the instance is closed

TO RUN THE PNPM COMMAND USE THIS FIRST
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
then pnpm turbo run build --concurrency=1

you are busy person you want to automate this you can do this using ci/cd pipeline

for this you will write the yml file

like its like called as **github actions/CI CD Actions** in which the above process is done by github only we manually don't have to do that

go to setting
secret and variables
actions
then add a secret(the key ec2 gave)

```
! cd_staging.yml X
1 name: Deploy to staging
2
3 on:
4   push:
5     branches: [ main ]
6
7 jobs:
8   redeploy_everything:
9     runs-on: ubuntu-latest
10    name: Deploying everything to the staging cluster
11    steps:
12      - run: |
13        echo "${{ secrets.SSH_PRIVATE_KEY }}" > ~/ssh_key chmod 600 ~/ssh_key
14        ssh -o StrictHostKeyChecking=no -i ~/ssh_key ubuntu@13.49.18.127 <<
15        'EOF'|
16        cd ~/turborepo-project && git pull
17        pnpm install
18        pnpm run build
19        pm2 restart http-server
20        pm2 restart fe-server
21        pm2 restart ws-server
```

in this the echo lines means that we have added a secret in the github actions it access that secret and that create a ssh key file and then ssh into the machine using that key file and run all the below commands
ez as f

we can validate the yml file using the yml validator online

there is problem in this like when we try to access the ec2 using the github actions it shows no access bcoz when we login it asks for yes or no and by default it chooses noo thats the problem

this script fucking god

```

name: Deploy to staging
on:
  push:
    branches: [ main ]

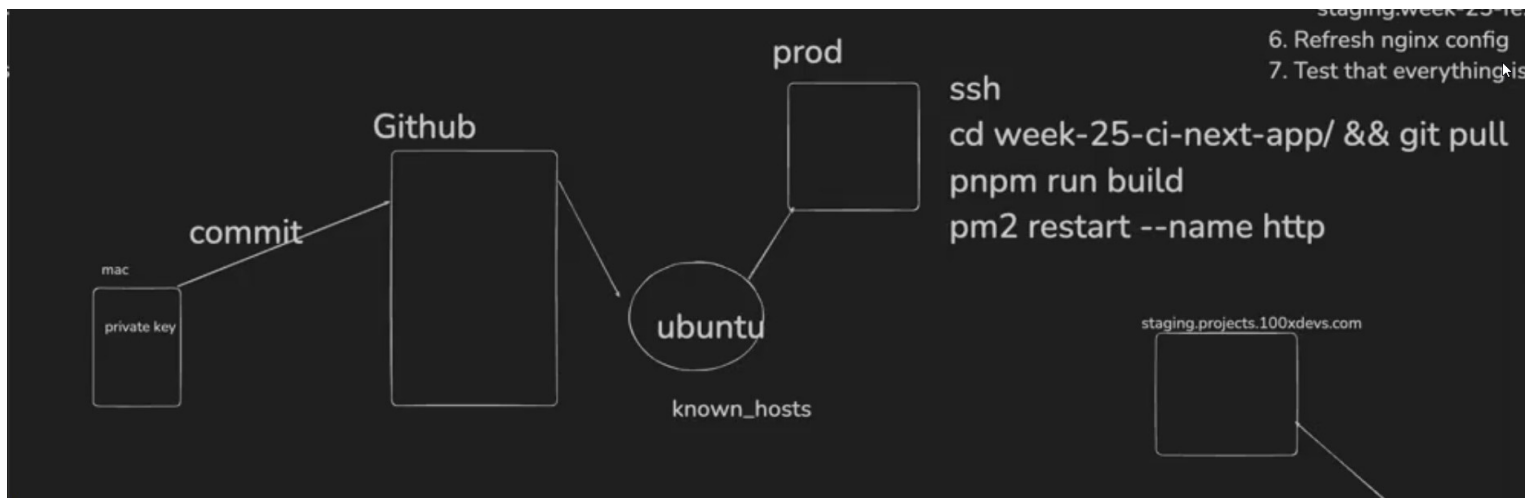
jobs:
  redeploy_everything:
    runs-on: ubuntu-latest
    name: Deploying everything to the staging cluster
    steps:
      - run: |
          echo "${{ secrets.SSH_PRIVATE_KEY }}" &> ~/ssh_key
          mkdir -p /home/runner/.ssh
          ls /home/runner/.ssh
          touch /home/runner/.ssh/known_hosts
          echo "${{ secrets.KNOWN_HOSTS }}" &> /home/runner/.ssh/known_hosts
          chmod 700 /home/runner/ssh_key
          ssh -o StrictHostKeyChecking=no -i ~/ssh_key ubuntu@13.49.18.127 -t "cd turborepo-
project/ && git pull origin main && export
PATH=/home/ubuntu/.nvm/versions/node/v24.14.0/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin && npm install -g pnpm && pnpm install &&
pnpm run build && pm2 restart fe-server && pm2 restart http-server && pm2 restart ws-server"

```

IN THIS THE PATH MUST BE TAKEN FROM THE SYSTEM USING THE COMMAND

echo \$PATH fucking node path and check the name of ws-server also, change ip also

now its working



same yml for production top par name change to production