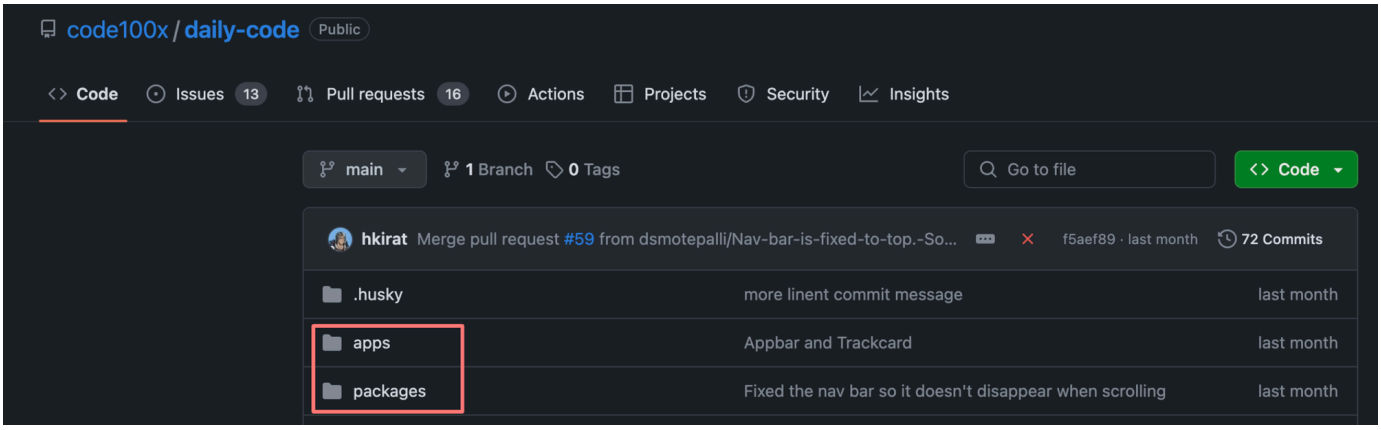
What are monorepos

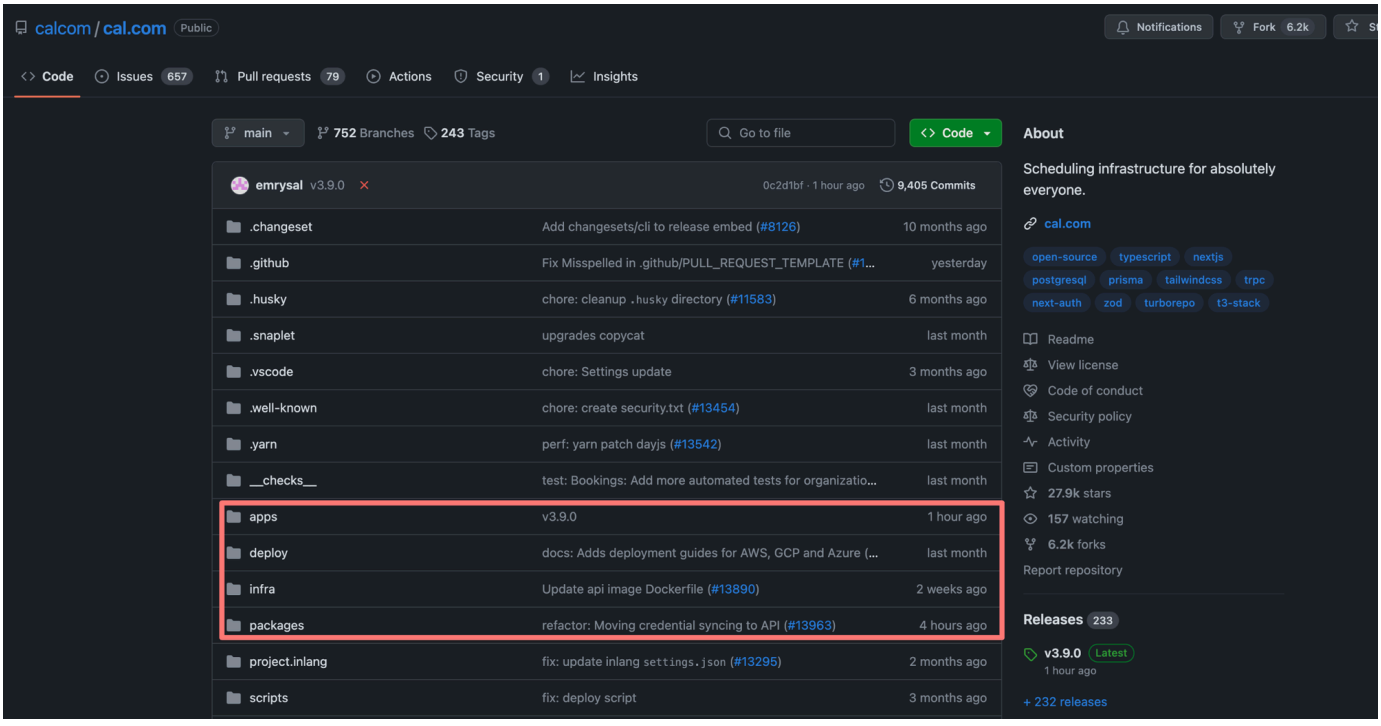
As the name suggests, a single repository (on github lets say) that holds all your frontend, backend, devops code.

Few repos that use monorepos are -

1. https://github.com/code100x/daily-code



1. https://github.com/calcom/cal.com



Do you need to know them very well as a full stack engineer

Not exactly. Most of the times they are setup in the project already by the `dev tools` guy and you just need to follow the right practises

Good to know how to set one up from scratch though

Why Monorepos?

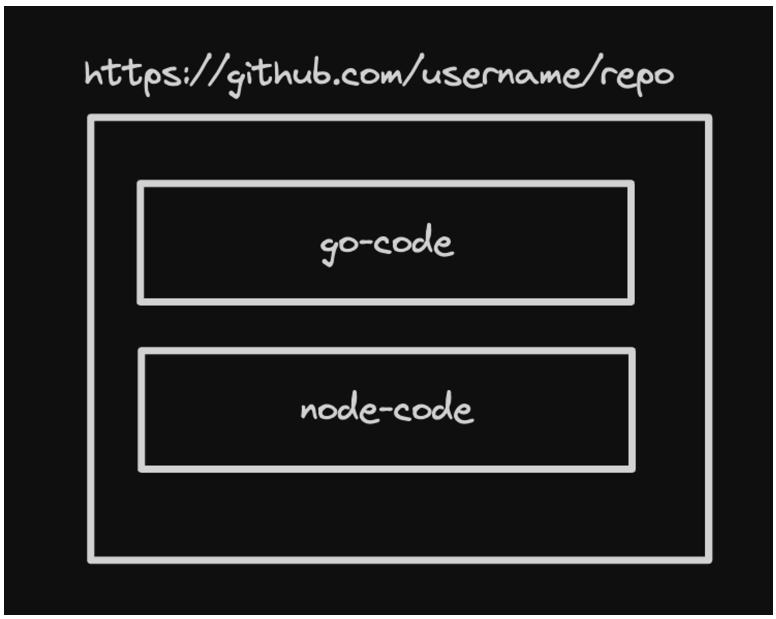
Why not Simple folders?

Why cant I just store services (backend, frontend etc) in various top level folders?

You can, and you should if your

1. Services are highly decoupled (dont share any code)
2. Services don't depend on each other.

For eg - A codebase which has a Golang service and a JS service



Why monorepos?

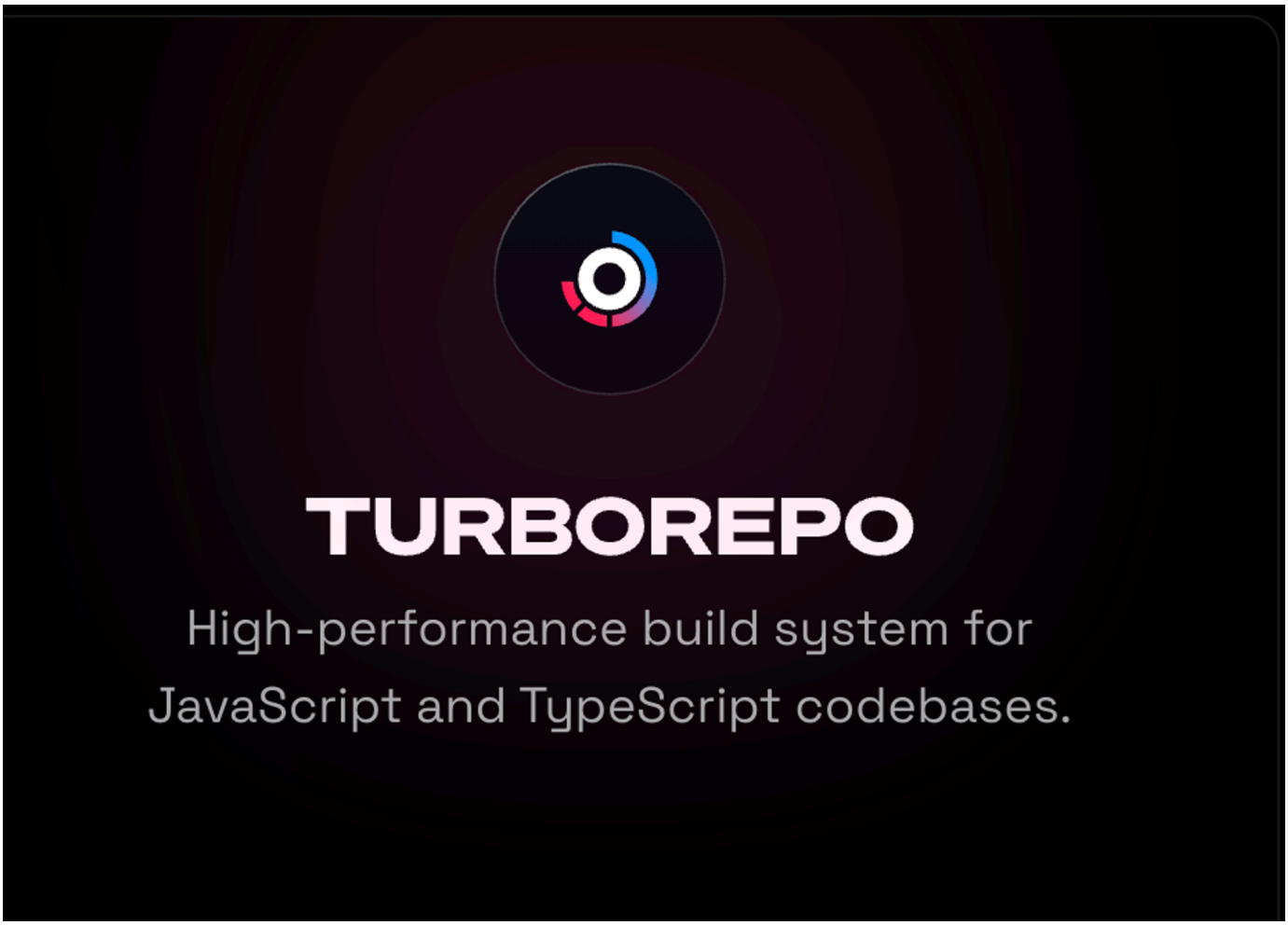
1. **Shared Code Reuse**
2. **Enhanced Collaboration**
3. **Optimized Builds and CI/CD**: Tools like TurboRepo offer smart caching and task execution strategies that can significantly reduce build and testing times.

4. **Centralized Tooling and Configuration**: Managing build tools, linters, formatters, and other configurations is simpler in a monorepo because you can have a single set of tools for the entire project.

Common monorepo framework in Node.js

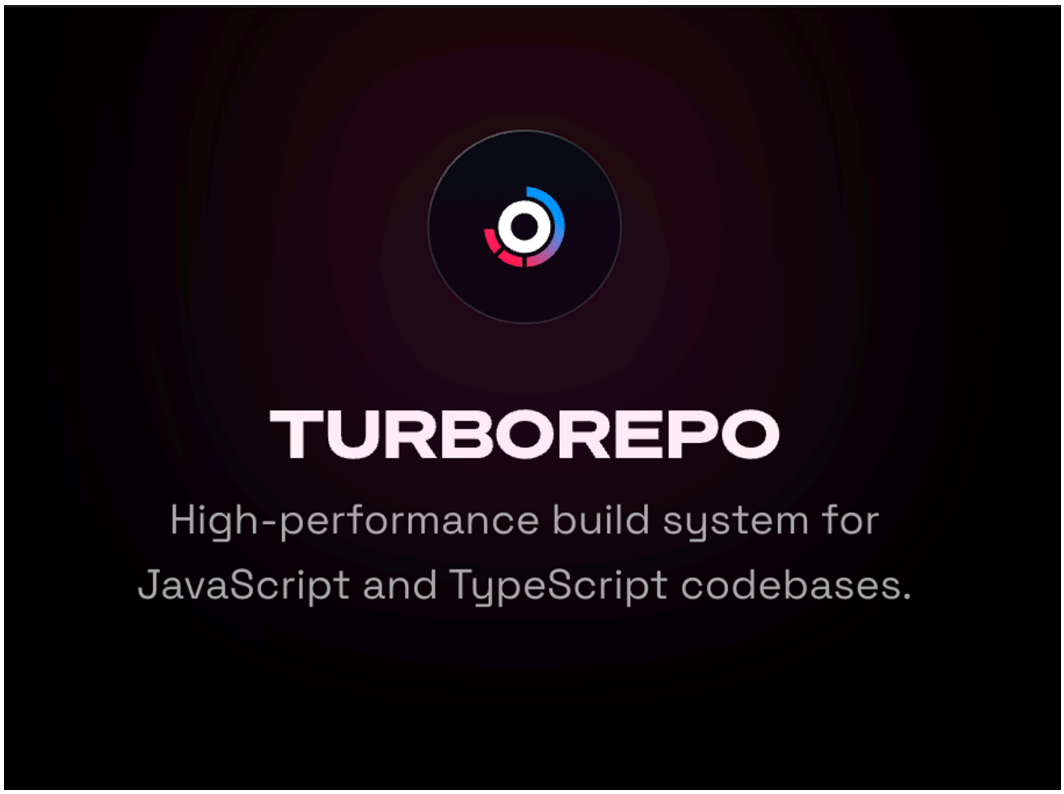
1. Lerna - https://lerna.js.org/
2. nx - https://github.com/nrwl/nx
3. Turborepo - https://turbo.build/ — Not exactly a monorepo framework
4. Yarn/npm workspaces - https://classic.yarnpkg.com/lang/en/docs/workspaces/

We'll be going through turborepo since it's the most relevant one today and provides more things (like build optimisations) that others don't

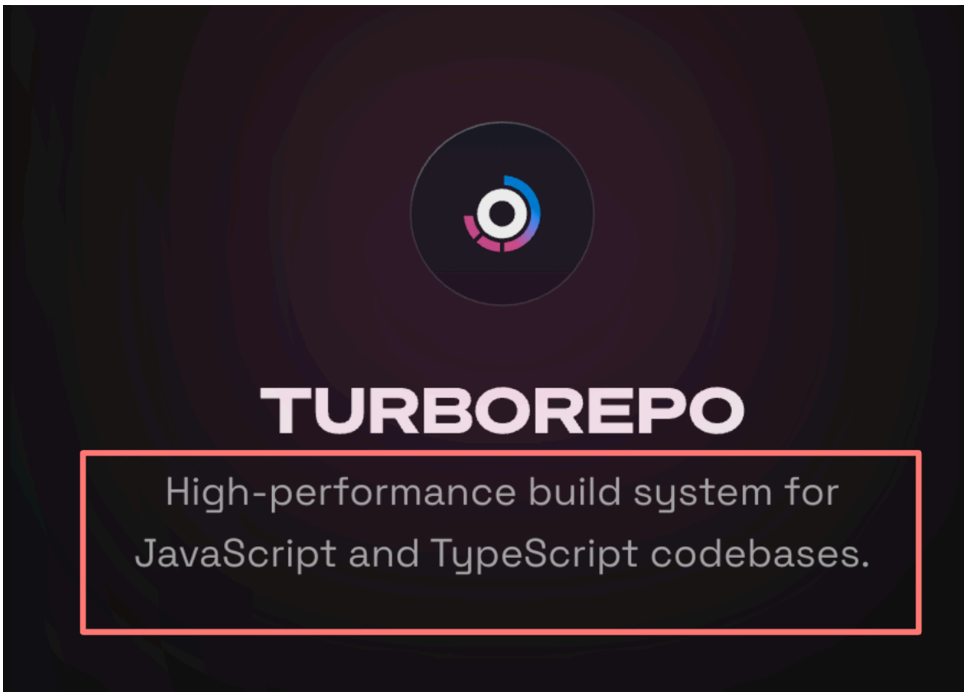


History of Turborepo

1. Created by `Jared Palmer`
2. In December 2021 Acquired/aqui-hired by `Vercel`
3. Mild speculation/came from a random source - Pretty hefty dealp
4. They've built a bunch of products, Turborepo is the most used one



Build system vs Build system orchestrator vs Monorepo framework



Build System

A build system automates the process of transforming source code written by developers into binary code that can be executed by a computer. For JavaScript and TypeScript projects, this process can include transpilation (converting TS to JS), bundling (combining multiple files into fewer files), minification (reducing file size), and more. A build system might also handle running tests, linting, and deploying applications.

Build System Orchestrator

TurboRepo acts more like a build system orchestrator rather than a direct build system itself. It doesn't directly perform tasks like transpilation, bundling, minification, or running tests. Instead, TurboRepo allows you to define tasks in your monorepo that call other tools (which are the actual build systems) to perform these actions.

These tools can include anything from tsc, vite etc

Monorepo Framework

A monorepo framework provides tools and conventions for managing projects that contain multiple packages or applications within a single repository (monorepo). This includes dependency management between packages, workspace configuration

Turborepo as a build system orchestrator

Turborepo is a `build system orchestrator` .

The key feature of TurboRepo is its ability to manage and optimize the execution of these tasks across your monorepo. It does this through:

1. **Caching**: TurboRepo caches the outputs of tasks, so if you run a task and then run it again without changing any of the inputs (source files, dependencies, configuration), TurboRepo can skip the actual execution and provide the output from the cache. This can significantly speed up build times, especially in continuous integration environments.
2. **Parallelization**: It can run independent tasks in parallel, making efficient use of your machine's resources. This reduces the overall time needed to complete all tasks in your project.
3. **Dependency Graph Awareness**: TurboRepo understands the dependency graph of your monorepo. This means it knows which packages depend on each other and can ensure tasks are run in the correct order.

Let’s initialize a simple Turborepo

Ref https://turbo.build/repo/docs

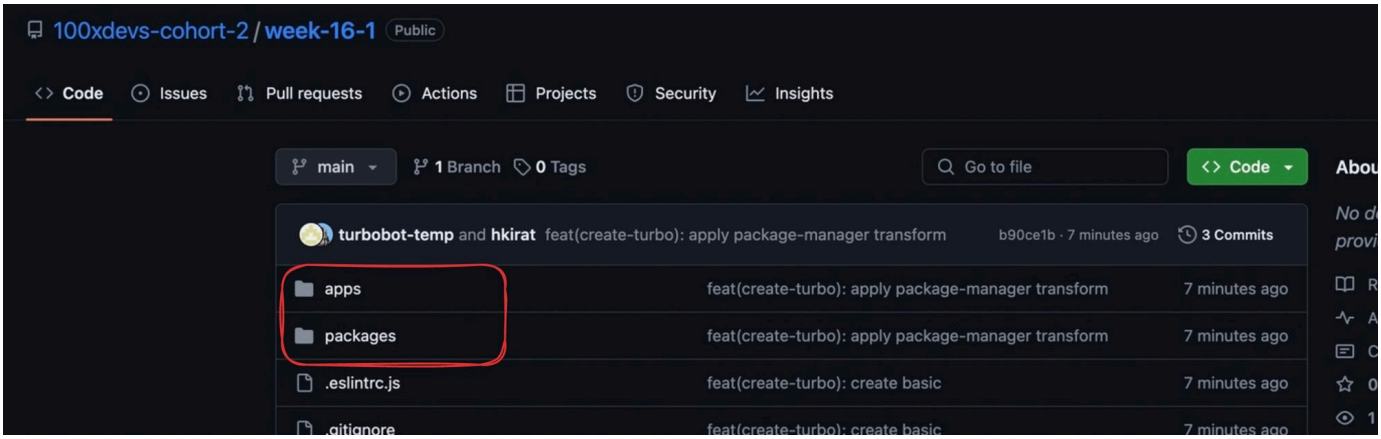
1. Initialize a Turborepo

```
npx create-turbo@latest
```
Copy

1. Select `npm workspaces` as the monorepo framework

> 💡 If it is taking a long time for you, you can close this starter from https://github.com/100xdevs-cohort-2/week-16-1 and run `npm install` inside the root folder

By the end, you will notice a folder structure that looks like this -



Explore the folder structure

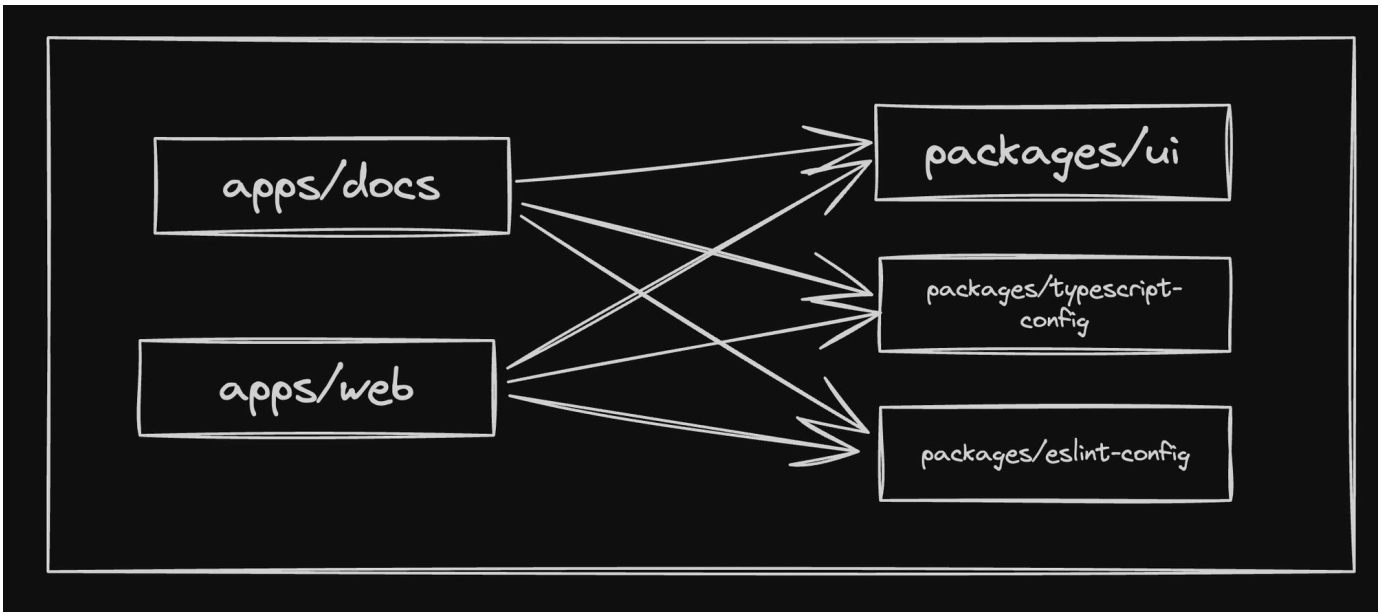
There are 5 modules in our project

End user apps (websites/core backend)

1. `apps/web` - A Next.js website
2. `apps/docs` - A Docs website that has all the documentation related to your project

Helper packages

1. `packages/ui` - UI packages
2. `packages/typescript-config` - Shareable TS configuration
3. `packages/eslint-config` - Shareable ESLine configuration



Let's try to run the project

In the root folder, run

```
npm run dev
```
Copy

💡 You might have to upgrade your node.js version

You will notice two websites running on

1. localhost:3000
2. localhost:3001

This means we have a single `repo` which has multiple `projects` which share code from `packages/ui`

Exploring root package.json

scripts

This represents what command runs when you run

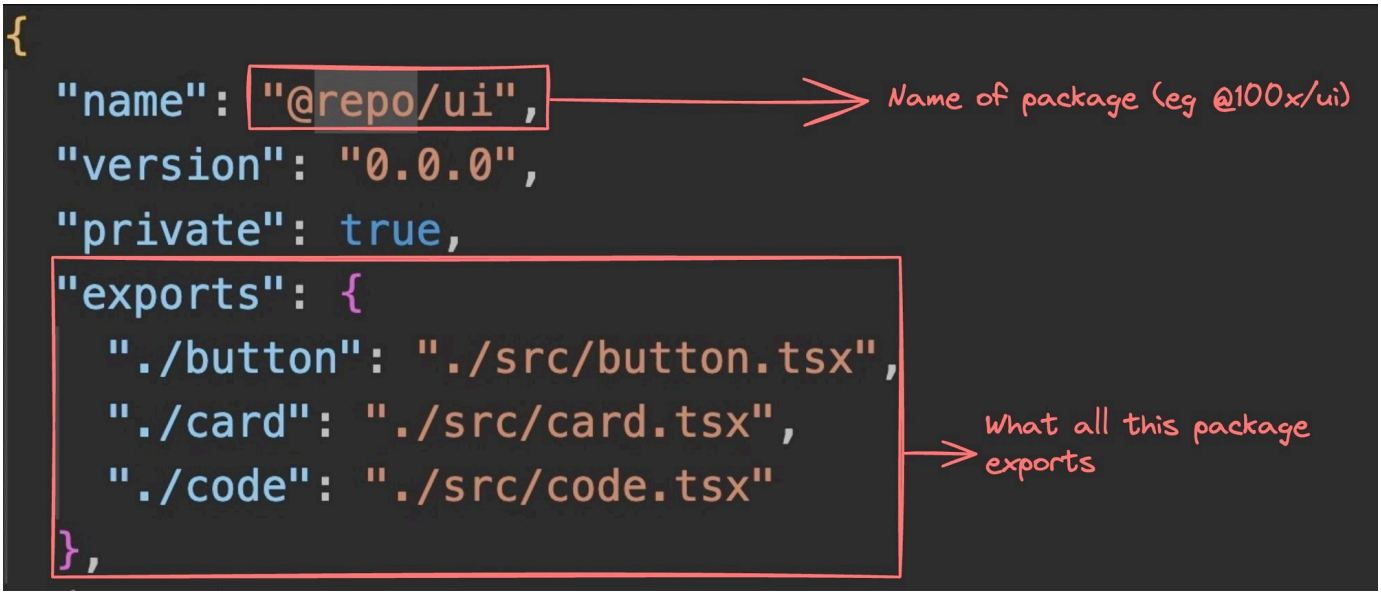
1. npm run build
2. npm run dev
3. npm run lint

`turbo build` goes into all packages and apps and runs `npm run build` inside them (provided they have it)

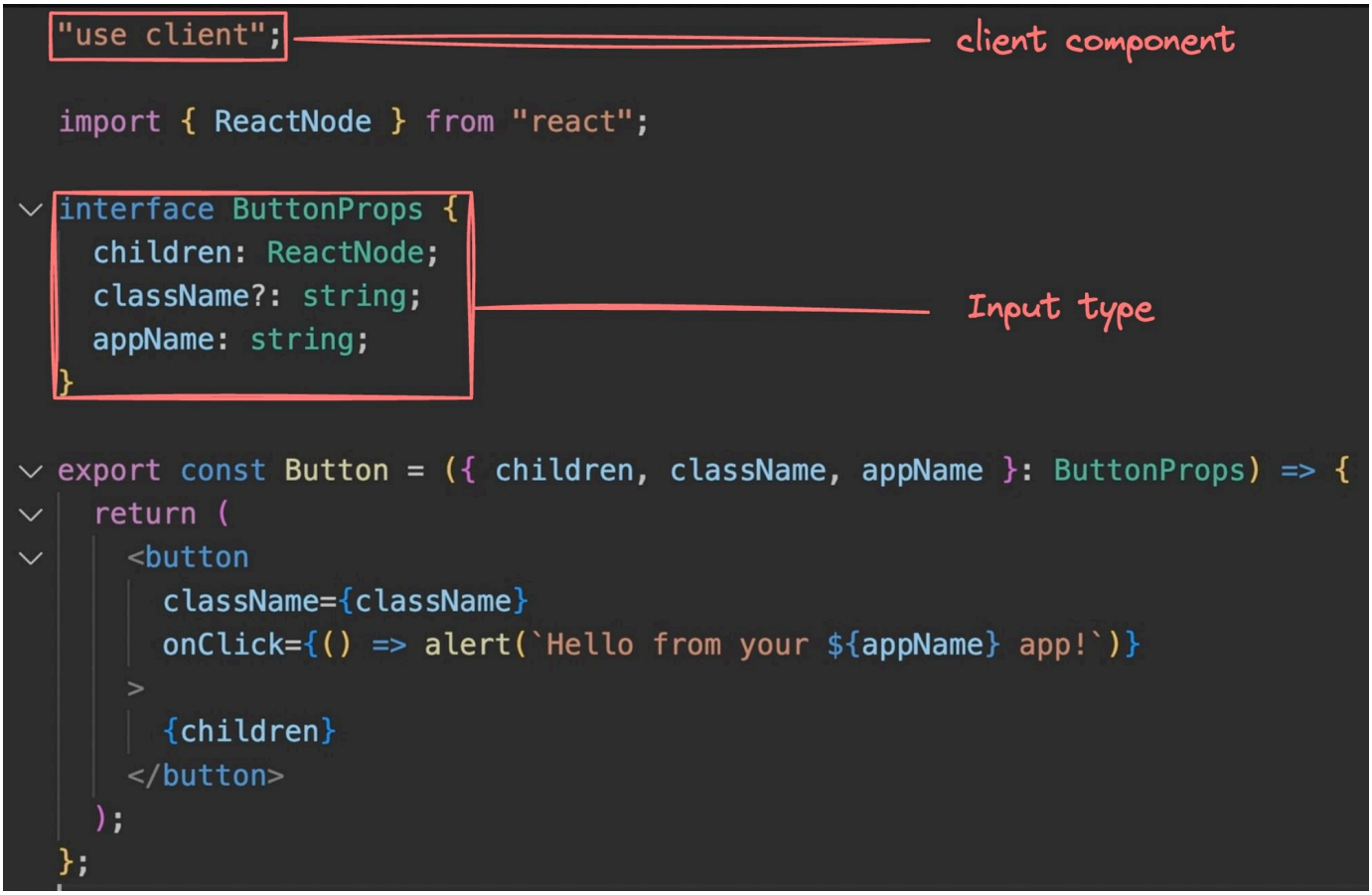
Same for `dev` and `lint`

Exploring packages/ui

1. package.json



2. src/button.tsx



3. `turbo` folder

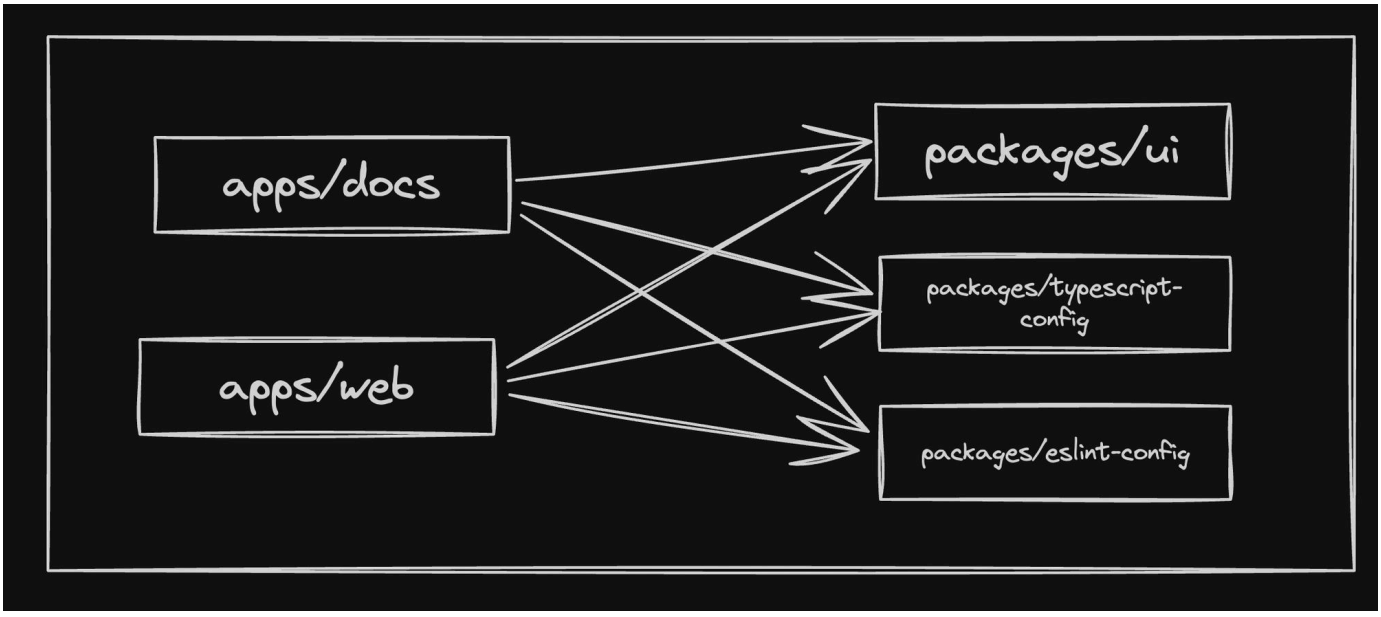
This is an interesting folder that was introduced recently. More details here - https://turbo.build/repo/docs/core-concepts/monorepos/code-generation

We'll come back to this after a few slides

Exploring `apps/web`

1. Dependencies

It is a simple next.js app. But it uses some `UI components` from the `packages/ui` module



2. Exploring package.json

If you explore package.json of `apps/web` , you will notice `@repo/ui` as a dependency

```
"dependencies": {
  "@repo/ui": "*",
  "next": "^14.1.1",
  "react": "^18.2.0",
  "react-dom": "^18.2.0"
},
```

3. Exploring page.tsx

This is a very big page, let's try to see the import and usage of the `Button` component

```
import Image from "next/image";
import { Card } from "@repo/ui/card";
import { Code } from "@repo/ui/code";
import styles from "./page.module.css";
import { Button } from "@repo/ui/button";
```

Import from packages module

```
<Button appName="web" className={styles.button}>
  Click me!
</Button>
```

Let's add a new page

Try adding a new page to `/admin` to the `apps/web` next.js website.

It should use a simple `Admin` component from `packages/ui`

Steps to follow -

- Create a new file `admin.tsx` inside `packages/ui/src`
- Export a simple React component

▼ Solution

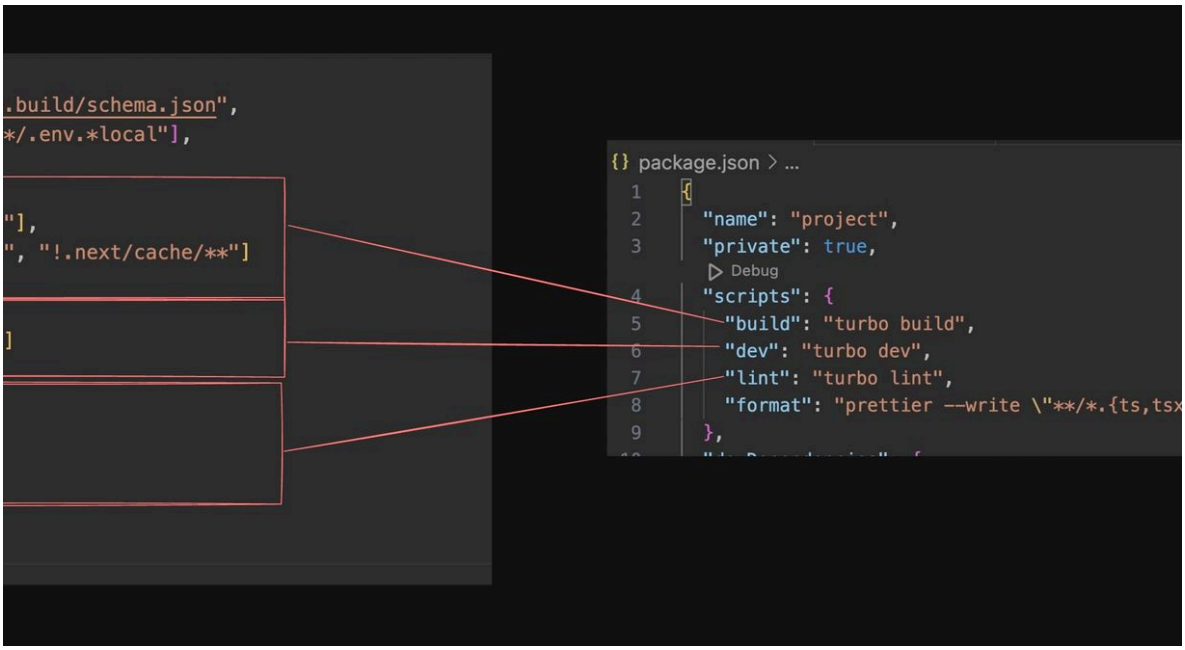
```
"use client";

export const Admin = () => {
  return (
    <h1>
        hi from admin component
    </h1>
  );
};
```

Copy

- Add the component to exports in `packages/ui/package.json`
- Create `apps/web/app/admin/page.tsx`
- Export a default component that uses the `@repo/ui/admin` component
- Run npm run dev (either in root or in `apps/web`) and try to see the website
- Go to http://localhost:3000/admin

> 💡 You can also use the `packages/ui/turbo/generators` to quickly bootstrap a new component
> Try running `npx gen react-component` and notice it'll do step 1, 2, 3 for you in one cli call

Exploring turbo.json



Ref - https://turbo.build/repo/docs/getting-started/create-new#3-understanding-turbojson

References - https://turbo.build/repo/docs/reference/configuration#globaldependencies

Adding React projects

1. Go to the apps folder

```
cd apps
```
Copy

2. Create a fresh vite app

```
npm create vite@latest
```
Copy

1. Update package.json to include `@repo/ui` as a dependency

```
"@repo/ui": "*",
```
Copy

1. Run npm install in the root folder

```
cd ..
npm install
```
Copy

1. Run npm run dev

```
npm run dev
```
Copy

1. Try importing something from the `ui` package and rendering it
2. Add a `turbo.json` to the react folder to override the `outputs` object of this module.

Ref https://turbo.build/repo/docs/core-concepts/monorepos/configuring-workspaces

```
{
  "extends": ["//"],
  "pipeline": {
    "build": {
      "outputs": ["dist/**"]
    }
  }
}
```
Copy

Caching in Turborepo

Ref - https://turbo.build/repo/docs/getting-started/create-new#using-the-cache

One of the big things that make turborepo fast and efficient is caching

It watches your files across builds and returns the cached response of builds if no files have changed.

Try running `npm run build` more than once and you'll see the second times it happens extremely fast.

You can also explore the `node_modules/.cache/turbo` folder to see the zipped cache files and unzip them using

```
tar --use-compress-program=unzstd -xvf name.tar.zst
```

Copy

Adding a Node.js app

Everything else remains the same (Create a new project, add typescript, add express...)

The only thing that's different is that `tsc` doesn't perform great with turborepo

You can use either `tsup` or `esbuild` for building your backend application

1. Create `apps/backend`
2. Initialize empty ts repo

```
npm init -y
npx tsc --init
```

1. Use base tsconfig (Ref - https://github.com/vercel/turbo/blob/main/examples/kitchen-sink/apps/api/tsconfig.json)

```
{
  "extends": "@repo/typescript-config/base.json",
  "compilerOptions": {
    "lib": ["ES2015"],
    "module": "CommonJS",
    "outDir": "./dist",
  },
  "exclude": ["node_modules"],
  "include": ["."]
}
```

1. Add dependencies

```
npm i express @types/express
```

1. Add `src/index.ts`

```
import express from "express";

const app = express()

app.get("/", (req, res) => {
    res.json({
        message: "hello world"
    });
})
```

1. Update turbo.json

```
Copy
{
  "extends": ["//"],
  "pipeline": {
    "build": {
      "outputs": ["dist/**"]
    }
  }
}
```

1. Install esbuild

```
Copy
npm install esbuild
```

1. Add build script to package.json

```
"build": "esbuild src/index.ts --platform=node --bundle --outdir=dist"
Copy
```

Adding a `common` module

A lot of times you need a module that can be shared by both frontend and backend apps

1. Initialize a `packages/common` module

```
Copy
cd packages
mkdir common
```

1. Initialize an empty node.js project

```
npm init -y
npx tsc --init
```

Copy

1. Change the name to `@repo/common`
2. Export a few things from `src/index.ts`

```
export const NUMBER = 1;
```

Copy

1. Add it to the `package.json` of various apps (next app/react app/node app)

```
"@repo/common": "*",
```

Copy

1. Import it in there and try to use it
2. Run npm install in root folder and see if it works as expected