Week 16.1

In this lecture, Harkirat introduces `monorepos` and the `Turborepo framework`, exploring its features like `caching`, `parallelization`, and `dependency management`. He guides through initializing a Turborepo project, running applications, adding different types of apps (Next.js, React, Node.js), and creating `shared common modules` across the monorepo.

What are Monorepos

A monorepo, short for "monolithic repository," is a single repository that contains all the code for multiple projects or components of a larger application. Instead of having separate repositories for frontend, backend, and DevOps code, a monorepo consolidates everything into a single repository.

In a monorepo structure, you'll typically find directories for different parts of the application, such as:

```
monorepo/
  ├── frontend/
  │   ├── web-app/
  │   └── mobile-app/
  ├── backend/
  │   ├── api-server/
  │   └── database/
  ├── devops/
  │   ├── ci-cd/
```

```
│   └── infrastructure/
└── shared/
    ├── utils/
    └── components/
```

This structure allows for a centralized location where all the code related to a project or organization can be stored, versioned, and managed together.

Examples of Monorepos

Several popular open-source projects and companies adopt the monorepo approach. Here are a couple of examples:

1. Daily Code (https://github.com/code100x/daily-code):
   This repository contains code for a daily coding challenge website. It includes both the frontend and backend code in a single repository.
2. Cal.com (https://github.com/calcom/cal.com):

   Cal.com is an open-source scheduling platform. Their monorepo contains the code for the web application, backend services, and various integrations.

Importance for Full Stack Engineers

As a full stack engineer, it's not essential to have in-depth knowledge of setting up and managing monorepos from scratch. In most cases, the monorepo structure is already established by the development tools team or the project's initial setup.

However, it's beneficial to understand the best practices and conventions used within a monorepo. This includes:

- Understanding the directory structure and where to locate specific components or modules.
- Following the established coding standards and guidelines for the project.
- Utilizing the shared libraries and utilities effectively to maintain consistency across the codebase.
- Collaborating with other team members and working within the monorepo workflow.

While setting up a monorepo from scratch is not a common task for most full stack engineers, it's still valuable to have a general understanding of how monorepos work and their benefits.

Setting Up a Monorepo

If you do need to set up a monorepo from scratch, there are several tools and frameworks available to help you. Some popular choices include:

- Lerna (https://lerna.js.org/): A tool for managing JavaScript projects with multiple packages.
- Nx (https://nx.dev/): A powerful suite of tools for monorepo development, particularly suited for Angular and React projects.
- Bazel (https://bazel.build/): A build and test tool that supports monorepo structures and enables fast and scalable builds.

These tools provide features like shared code management, dependency resolution, and efficient building and testing processes. While setting up a monorepo from scratch can be complex, the tools and frameworks available make it easier to manage and scale large codebases effectively.

> Monorepos provide a centralized approach to managing code for multiple projects or components within a single repository. They offer benefits such as code sharing, simplified dependency management, and easier collaboration among team members.

Why Monorepos

When to Use Monorepos

1. **Shared Code Reuse**:
   One of the primary benefits of using a monorepo is the ability to easily share code between different services or projects. In a monorepo, you can

have a dedicated directory for shared libraries, utilities, or components that can be used across multiple services.

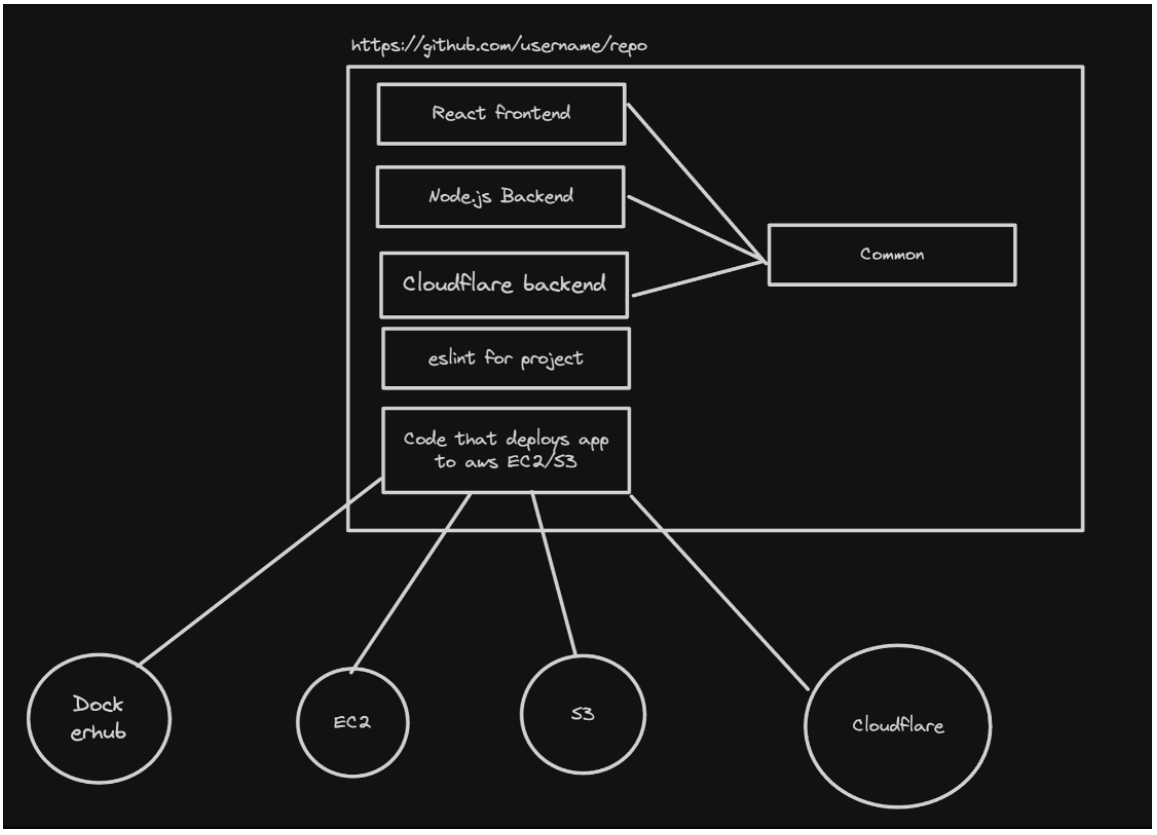
For example, consider a monorepo structure like this:

```
monorepo/
  ├── shared/
  │   ├── utils/
  │   └── components/
  ├── frontend/
  │   └── web-app/
  └── backend/
      └── api-server/
```

In this structure, the `shared` directory contains code that can be reused by both the `frontend` and `backend` services. This promotes code reuse, reduces duplication, and makes it easier to maintain and update shared code.

2. **Enhanced Collaboration**:
   Monorepos foster collaboration among team members by providing a centralized location for all the code. Developers can easily navigate and contribute to different parts of the project without the need to switch between multiple repositories.

   With a monorepo, developers have visibility into the entire codebase, making it easier to understand the dependencies and relationships between different services. This promotes cross-team collaboration and knowledge sharing.



3. **Optimized Builds and CI/CD**:
   Monorepos enable the use of specialized tools like TurboRepo, which offer smart caching and task execution strategies. These tools can significantly reduce build and testing times by leveraging the shared nature of the codebase.

   For example, TurboRepo can intelligently cache build artifacts and only rebuild the parts of the project that have changed. This means that if you modify a specific service, only that service and its dependencies will be rebuilt, rather than rebuilding the entire project from scratch.

   Here's an example of how you can configure TurboRepo in your monorepo:

```
// turbo.json
{
  "$schema": "<https://turbo.build/schema.json>",
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**"]
```

```
    },
    "test": {
      "dependsOn": ["build"],
      "outputs": []
    }
  }
}
```

In this configuration, the `build` task depends on the `build` tasks of its dependencies, and the `test` task depends on the `build` task. TurboRepo will optimize the execution of these tasks based on the defined dependencies and cache the outputs for faster subsequent builds.

4. **Centralized Tooling and Configuration**:
   Managing build tools, linters, formatters, and other configurations becomes simpler in a monorepo. Instead of duplicating configurations across multiple repositories, you can have a single set of tools and configurations that apply to the entire project.

   For example, you can have a single `package.json` file at the root of your monorepo that defines the common dependencies and scripts for all the services:

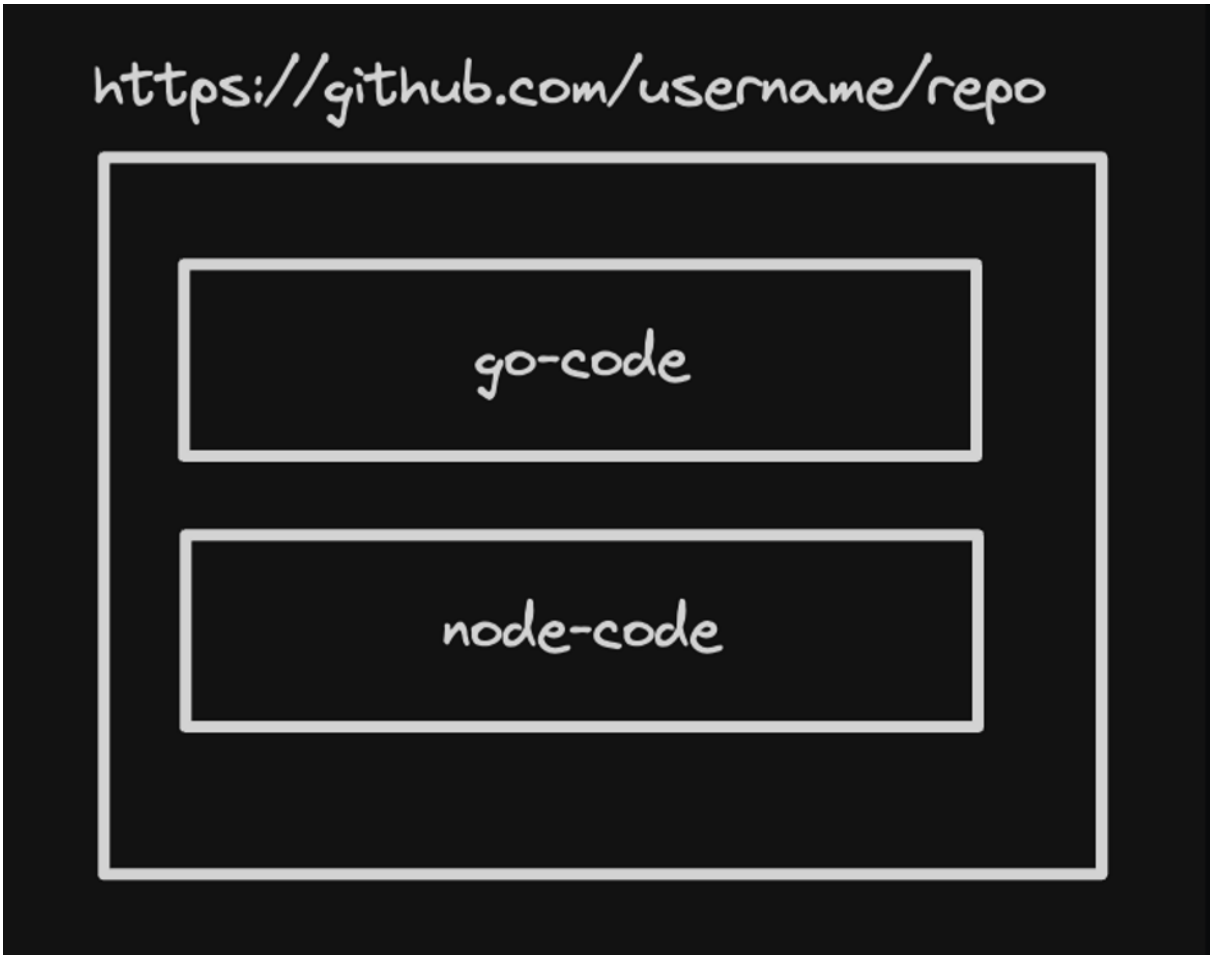
```
// package.json
{
  "name": "monorepo",
  "scripts": {
    "build": "turbo run build",
    "test": "turbo run test",
    "lint": "eslint .",
    "format": "prettier --write ."
  },
  "devDependencies": {
    "eslint": "^8.0.0",
    "prettier": "^2.0.0",
    "turbo": "^1.0.0"
  }
}
```

With this centralized configuration, you can run commands like `npm run build`, `npm run test`, `npm run lint`, or `npm run format` at the root level, and they will be applied consistently across all the services in your monorepo.

When to Use Simple Folders

While monorepos offer several benefits, there are scenarios where using simple folders might be sufficient:

1. **Highly Decoupled Services**: If your services are highly decoupled and don't share any code, using separate folders or even separate repositories might be a simpler approach. This is especially true if the services are written in different languages or have vastly different requirements.
2. **Independent Services**: If your services don't depend on each other and can be developed, tested, and deployed independently, using separate folders or repositories can provide more flexibility and autonomy to individual teams.



For example, if you have a codebase with a Golang service and a JavaScript service that don't share any code or dependencies, you can structure them in separate folders:

```
project/
  ├── golang-service/
  └── js-service/
```

In this case, each service can have its own build process, dependencies, and deployment pipeline, without the need for a monorepo setup.

Conclusion

Monorepos offer several advantages, such as shared code reuse, enhanced collaboration, optimized builds and CI/CD, and centralized tooling and configuration. They are particularly beneficial when you have multiple services or projects that share code and dependencies.

However, if your services are highly decoupled or don't depend on each other, using simple folders or separate repositories might be a more straightforward approach.

Ultimately, the choice between a monorepo and separate folders/repositories depends on the specific needs and characteristics of your project. Consider factors such as code sharing, collaboration requirements, build and deployment processes, and team structure when making the decision.

Monorepo Frameworks

Common Monorepo Frameworks in Node.js

1. **Lerna** (https://lerna.js.org/):
   Lerna is a popular tool for managing JavaScript monorepos. It provides features like package management, versioning, and publishing. With Lerna, you can easily manage dependencies, run scripts across multiple packages, and publish packages to npm.

   Example `lerna.json` configuration:

   ```
   {
     "packages": ["packages/*"],
     "version": "independent",
     "npmClient": "yarn",
     "useWorkspaces": true
   }
   ```

2. **Nx** (https://github.com/nrwl/nx):
   Nx is a powerful monorepo tool and build system developed by Nrwl. It offers a wide range of features, including code generation, dependency management, and advanced build optimization. Nx supports various frontend and backend frameworks, making it a versatile choice for monorepo development.

   Example `nx.json` configuration:

   ```
   {
     "npmScope": "myorg",
   ```

```
  "affected": {
    "defaultBase": "main"
  },
  "tasksRunnerOptions": {
    "default": {
      "runner": "@nrwl/workspace/tasks-runners/default",
      "options": {
        "cacheableOperations": ["build", "lint", "test",
"e2e"]
      }
    }
  }
}
```

3. **Turborepo** (https://turbo.build/):
   Turborepo is a high-performance build system for JavaScript and TypeScript codebases. While not strictly a monorepo framework, it provides powerful features for managing and optimizing monorepo builds. Turborepo focuses on fast incremental builds, caching, and efficient task execution.

   Example `turbo.json` configuration:

```
{
  "$schema": "<https://turborepo.org/schema.json>",
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**"]
    },
    "test": {
      "dependsOn": ["build"],
      "outputs": []
    }
  }
}
```

4. **Yarn/npm Workspaces** (https://classic.yarnpkg.com/lang/en/docs/workspaces/):

Yarn and npm both provide built-in support for workspaces, which allow you to manage multiple packages within a single repository. Workspaces enable you to share dependencies across packages and link them together for development purposes.

Example `package.json` configuration with Yarn workspaces:

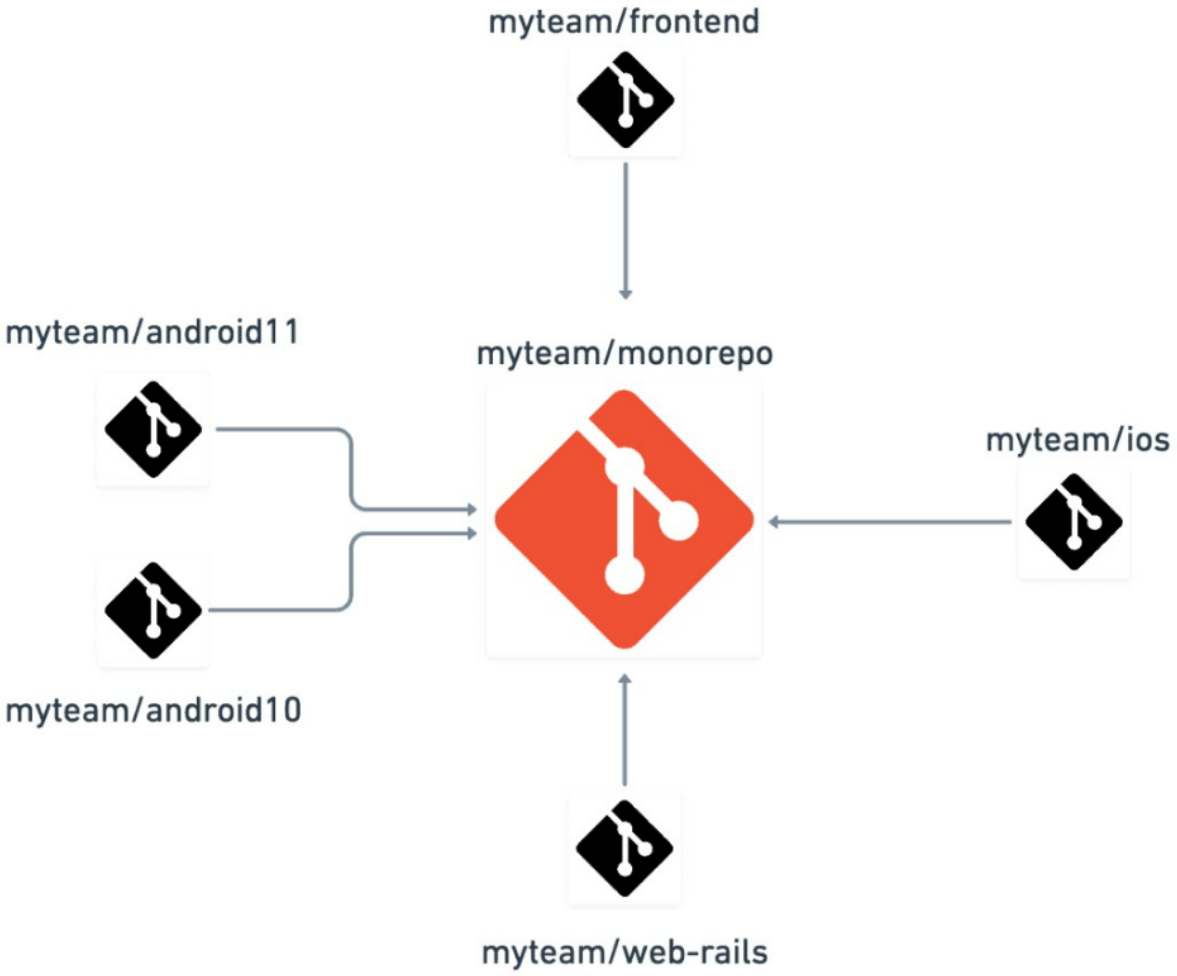
```
{
  "private": true,
  "workspaces": ["packages/*"]
}
```

Monorepos vs. Turborepo

Monorepos and Turborepo are related but distinct concepts. Let's clarify the differences:

1] Monorepos:

- A monorepo is an architectural approach where multiple projects or packages are stored and managed within a single repository.
- Monorepos provide benefits such as code sharing, simplified dependency management, and unified versioning across projects.
- Monorepo tools like Lerna, Nx, and Yarn/npm workspaces help manage and orchestrate the monorepo structure, including package management, script execution, and publishing.
- Monorepos can be used with various build systems and tools, not limited to Turborepo.



2] Turborepo:

- Turborepo is a specific build system and task runner designed for high-performance builds in monorepo setups.
- It focuses on optimizing build times, caching, and efficient execution of tasks across multiple packages.
- Turborepo leverages a smart scheduling algorithm to determine the optimal order of task execution based on package dependencies.
- It provides features like remote caching, incremental builds, and parallel execution to speed up the development workflow.
- While Turborepo is commonly used in monorepo setups, it can also be used in multi-repo setups where fast and efficient builds are desired.

> In summary, monorepos are an architectural pattern for managing multiple projects within a single repository, while Turborepo is a specific build system that can be used within a monorepo (or multi-repo) setup to optimize and accelerate the build process. Turborepo complements monorepo tools by providing advanced build optimization features, making it a popular choice for monorepo development in the Node.js ecosystem. However, it's important to note that Turborepo is not limited to monorepos and can be used in various project setups where fast and efficient builds are required.

Short Interesting History of Turborepo

Turborepo was created by Jared Palmer to address the challenges of building and managing monorepos in the JavaScript ecosystem. The tool was initially written in Go, inspired by the success of the JavaScript bundler esbuild, which is also written in Go.

The main objectives behind Turborepo were:

1. Utilize advanced build techniques used by big tech companies in their monorepo tools.
2. Provide an easy-to-adopt tool that doesn't require significant changes to the existing codebase.
3. Meet developers where they are, integrating with their current tools and workflows.

Turborepo achieves faster build times through smart caching and parallelization. It uses content-addressable storage to cache build artifacts and constructs a dependency graph to determine which tasks can be run in parallel.

In December 2021, Vercel acquired Turborepo through an acqui-hire of Jared Palmer and his team. This acquisition brought additional resources and support to grow and evolve Turborepo.

Although Turborepo has gained popularity and praise for its capabilities, it is still a relatively new tool (the latest version as of the search results is 1.1.16). Some users have reported inconsistencies and the need for further polishing for production usage.

Despite these early challenges, Turborepo has shown significant potential in improving build times and simplifying monorepo management. The company highlights a case study where Turborepo saved 67 hours of continuous integration time for a team of only four developers.

> Overall, Turborepo has emerged as a promising solution for JavaScript monorepos, leveraging advanced build techniques and offering an easy-to-adopt approach for developers.

Important Concepts

1] Build System

A build system is a tool that automates the process of transforming source code written by developers into executable code that can be run on a computer. In the context of JavaScript and TypeScript projects, a build system performs tasks such as:

- Transpilation: Converting TypeScript code to JavaScript code.
- Bundling: Combining multiple JavaScript files into a single file or a smaller set of files to optimize loading and execution.
- Minification: Reducing the size of JavaScript files by removing unnecessary characters and optimizing the code.
- Testing: Running automated tests to ensure the correctness and reliability of the code.
- Linting: Analyzing the code for potential errors, style inconsistencies, and adherence to coding standards.
- Deployment: Preparing the built code for deployment to a production environment.

Examples of popular build systems in the JavaScript ecosystem include Webpack, Rollup, Parcel, and Vite

2] Build System Orchestrator

A build system orchestrator, like Turborepo, sits on top of the actual build systems and coordinates the execution of tasks across multiple packages or projects within a monorepo. Instead of directly performing tasks like transpilation or bundling, a build system orchestrator focuses on:

- Task Definition: Allowing developers to define tasks in a configuration file (e.g., `turbo.json`) that specify the commands to run for each task.

- Task Orchestration: Determining the order in which tasks should be executed based on their dependencies and optimizing the execution process.
- Caching: Intelligently caching the results of previous builds to speed up subsequent builds and avoid redundant work.
- Parallel Execution: Leveraging available system resources to run tasks in parallel, improving overall build performance.

Here's an example of how you might define tasks in a `turbo.json` file:

```
{
  "$schema": "<https://turborepo.org/schema.json>",
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**"]
    },
    "test": {
      "dependsOn": ["build"],
      "outputs": []
    },
    "lint": {
      "outputs": []
    }
  }
}
```

In this example, the `build` task depends on the `build` tasks of its dependencies (indicated by `^build`), the `test` task depends on the `build` task, and the `lint` task has no dependencies. Turborepo will optimize the execution of these tasks based on their dependencies and cache the outputs for faster subsequent builds.

3] Monorepo Framework

A monorepo framework provides a set of tools and conventions for managing projects that contain multiple packages or applications within a single repository. It focuses on:

- Dependency Management: Handling dependencies between packages within the monorepo, ensuring that packages can reference and use each other correctly.
- Workspace Configuration: Providing a way to define and configure workspaces, which are separate packages or projects within the monorepo.
- Shared Code: Facilitating the sharing of common code, utilities, and configurations across packages within the monorepo.
- Versioning: Managing the versioning of packages within the monorepo, often using a unified versioning scheme.

Examples of monorepo frameworks include Lerna, Nx, and Rush.

Here's an example of a monorepo structure using Lerna:

```
monorepo/
  ├── packages/
  │   ├── package-a/
  │   │   ├── src/
  │   │   ├── package.json
  │   │   └── tsconfig.json
  │   └── package-b/
  │       ├── src/
  │       ├── package.json
  │       └── tsconfig.json
  ├── package.json
  └── lerna.json
```

In this structure, the `packages` directory contains individual packages (`package-a` and `package-b`), each with its own `package.json` and source code. The root `package.json` and `lerna.json` files configure the monorepo and define the workspaces.

Relationship between all 3

These three concepts are related but serve different purposes:

- A build system focuses on the actual transformation and processing of source code into executable code.

- A build system orchestrator sits on top of build systems and coordinates the execution of tasks across multiple packages or projects within a monorepo.
- A monorepo framework provides the structure, conventions, and tools for managing multiple packages or projects within a single repository.

> In a typical monorepo setup, you might use a monorepo framework like Lerna to manage the overall structure and dependencies, a build system orchestrator like Turborepo to optimize and coordinate the execution of tasks, and individual build systems like Webpack or Vite to perform the actual code transformation and bundling.

Turborepo Features

Turborepo is a build system orchestrator designed to manage and optimize the execution of tasks across a monorepo. It sits on top of existing build systems and provides a layer of coordination and optimization.

Instead of directly performing build tasks like transpilation, bundling, or testing, Turborepo focuses on managing the execution of these tasks across the packages in your monorepo. It leverages intelligent caching, parallelization, and dependency graph awareness to speed up the overall build process.

Here are the key features of Turborepo that make it an effective build system orchestrator:

1. **Caching**:
   Turborepo implements a caching mechanism to avoid unnecessary work and speed up subsequent builds. It caches the outputs of tasks based on their inputs (source files, dependencies, and configuration). If a task is executed again without any changes to its inputs, Turborepo can retrieve the cached output instead of re-executing the task.

   For example, consider a `build` task defined in your `turbo.json` file:

```
{
  "$schema": "<https://turborepo.org/schema.json>",
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**"]
    }
  }
}
```

In this configuration, the `build` task depends on the `build` tasks of its dependencies (indicated by `^build`). Turborepo will cache the output of the `build` task in the `dist` directory. If the task is run again without any changes to its inputs, Turborepo will retrieve the cached output from the `dist` directory instead of re-executing the task.

2. **Parallelization**:
   Turborepo can identify independent tasks and run them in parallel, making efficient use of available system resources. By leveraging parallelization, Turborepo can significantly reduce the overall build time.

   For example, consider the following `turbo.json` configuration:

```
{
  "$schema": "<https://turborepo.org/schema.json>",
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**"]
    },
    "test": {
      "dependsOn": ["build"],
      "outputs": []
    },
    "lint": {
      "outputs": []
    }
```

```
    }
  }
```

In this example, the `test` task depends on the `build` task, but the `lint` task has no dependencies. Turborepo will recognize that the `lint` task can be run in parallel with the `build` task, optimizing the execution process.

3. **Dependency Graph Awareness**:
   Turborepo understands the dependency graph of your monorepo. It knows which packages depend on each other and ensures that tasks are run in the correct order based on those dependencies.

   For example, consider a monorepo with two packages: `package-a` and `package-b`. If `package-b` depends on `package-a`, Turborepo will ensure that the tasks for `package-a` are executed before the tasks for `package-b`.

   This dependency graph awareness allows Turborepo to build packages efficiently and avoid unnecessary rebuilds of downstream packages.

By leveraging caching, parallelization, and dependency graph awareness, Turborepo can significantly speed up the build process in a monorepo. It minimizes redundant work, optimizes resource utilization, and ensures that tasks are executed in the correct order based on package dependencies.

Turborepo integrates seamlessly with existing build systems and tools, allowing you to define tasks using familiar commands and scripts. It acts as an orchestration layer, coordinating the execution of these tasks across the packages in your monorepo. To get started with Turborepo, you typically create a `turbo.json` file at the root of your monorepo, defining the tasks and their dependencies. Turborepo then uses this configuration to optimize the execution of tasks based on the defined pipeline.

> By adopting Turborepo as a build system orchestrator, you can significantly improve the efficiency and speed of your

> monorepo builds, especially in large-scale projects with multiple packages and complex dependencies.

Initializing a Simple Turborepo

Let's dive into initializing a simple Turborepo and explore the generated folder structure.

Initializing a Turborepo

To initialize a new Turborepo, you can use the `create-turbo` CLI. Run the following command in your terminal:

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

This command will install the `create-turbo` package and run it to create a new Turborepo.

During the initialization process, you'll be prompted to select a monorepo framework. In this case, choose "npm workspaces" as the monorepo framework.

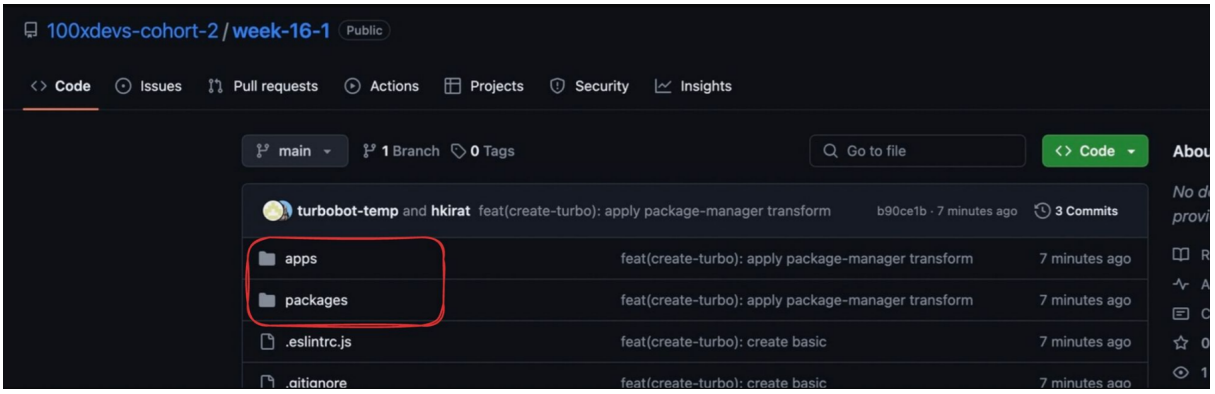
Cloning the Starter Repository

If the initialization process is taking a long time, you have the option to clone a starter repository instead. The starter repository provides a pre-configured Turborepo setup that you can use as a starting point.

To clone the starter repository, run the following command:

```
git clone <https://github.com/100xdevs-cohort-2/week-16-1.git>
```

After cloning the repository, navigate to the root folder of the cloned project and run `npm install` to install the necessary dependencies.



Turborepo Folder Structure

Once the initialization process is complete or you have cloned the starter repository, you will notice a folder structure similar to the following:

```
my-turborepo/
  ├── apps/
  │   ├── docs/
  │   └── web/
  ├── packages/
  │   ├── ui/
  │   ├── eslint-config/
  │   └── typescript-config/
  ├── package.json
  ├── turbo.json
  └── pnpm-workspace.yaml
```

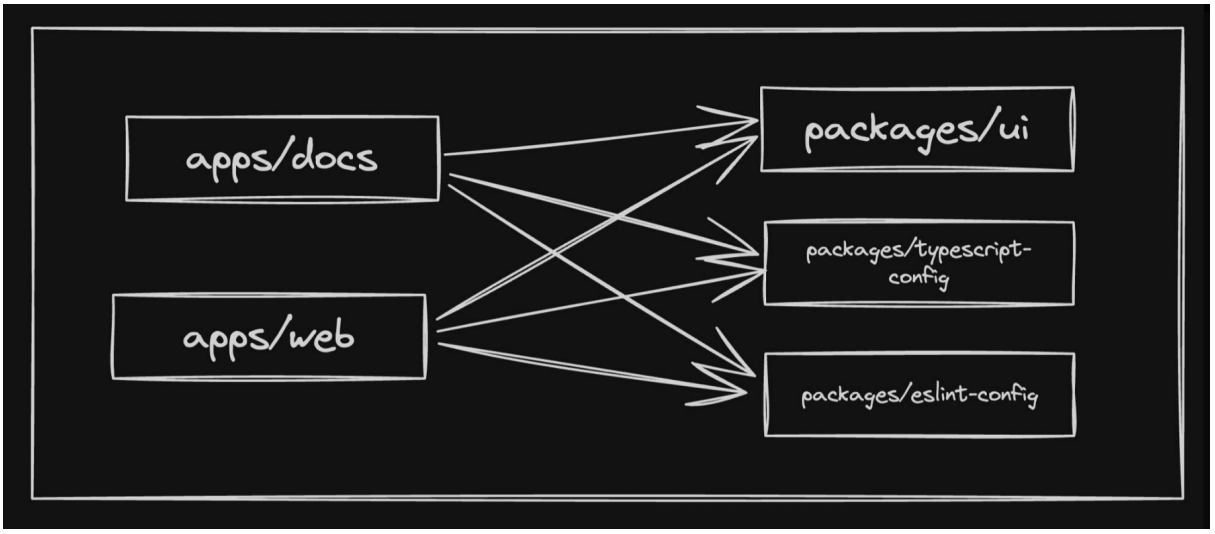
Let's explore the different modules and their purposes in this folder structure:

End User Apps

1. `apps/web`: This directory contains a Next.js website, which serves as the main user-facing application of your project.
2. `apps/docs`: This directory contains a documentation website built with Next.js. It is intended to host all the documentation related to your project.

Helper Packages

1. `packages/ui`: This directory contains UI packages that can be shared across different applications in your project. It may include reusable components, styles, or UI-related utilities.
2. `packages/typescript-config`: This directory contains a shareable TypeScript configuration. It allows you to maintain a consistent TypeScript configuration across all the packages and applications in your monorepo.
3. `packages/eslint-config`: This directory contains a shareable ESLint configuration. It helps enforce consistent coding standards and best practices across your project.



The `package.json` file in the root directory serves as the main configuration file for your monorepo. It includes dependencies, scripts, and other metadata relevant to the entire project.

The `turbo.json` file is the configuration file for Turborepo. It specifies the build pipeline, task dependencies, and other Turborepo-specific settings.

The `pnpm-workspace.yaml` file is a configuration file for pnpm workspaces. It defines the workspace structure and shared dependencies within your monorepo.

By having this modular structure, you can easily manage and develop different parts of your project independently while still maintaining a cohesive and organized codebase.

To start developing with your Turborepo, you can navigate to the root directory and run the following command:

```
pnpm dev
```

This command will start the development server for all the applications and packages in your monorepo, allowing you to work on them simultaneously.

Remember to refer to the Turborepo documentation (https://turbo.build/repo/docs) for more detailed information on configuring and working with Turborepo.

Running the Project

To run the project, navigate to the root folder of your Turborepo and execute the following command:

```
npm run dev
```

This command will start the development server for all the applications and packages in your monorepo.

Node.js Version

If you encounter any issues related to your Node.js version, make sure you have a compatible version installed. Turborepo requires a relatively recent version of Node.js. You can check the required version in the project's documentation or the `package.json` file.

To upgrade your Node.js version, you can use a version manager like nvm (Node Version Manager) or download and install the latest version from the official Node.js website (https://nodejs.org).

Running Applications

After running `npm run dev`, you will notice that two websites are running on different ports:

1. `localhost:3000`: This is the main user-facing application (`apps/web`) built with Next.js.
2. `localhost:3001`: This is the documentation website (`apps/docs`) also built with Next.js.

You can access these applications by opening your web browser and navigating to the respective URLs.

Shared Code

One of the key benefits of using a monorepo with Turborepo is the ability to share code across different projects. In this case, the `packages/ui` directory contains reusable UI components that can be shared between the `apps/web` and `apps/docs` applications.

For example, let's say you have a custom button component defined in `packages/ui/Button.tsx`:

```
// packages/ui/Button.tsx
import React from 'react';

interface ButtonProps {
  children: React.ReactNode;
  onClick?: () => void;
}

export const Button: React.FC<ButtonProps> = ({ children, o
nClick }) => {
  return (
    <button className="bg-blue-500 text-white px-4 py-2 rou
nded" onClick={onClick}>
      {children}
    </button>
  );
};
```

You can then import and use this button component in both the `apps/web` and `apps/docs` applications:

```tsx
// apps/web/pages/index.tsx
import { Button } from 'ui';

export default function Home() {
  return (
    <div>
      <h1>Welcome to the Main App</h1>
      <Button onClick={() => console.log('Button clicked')}
>Click Me</Button>
    </div>
  );
}
```

```tsx
// apps/docs/pages/index.tsx
import { Button } from 'ui';

export default function Docs() {
  return (
    <div>
      <h1>Documentation</h1>
      <Button onClick={() => console.log('Button clicked')}
>Click Me</Button>
    </div>
  );
}
```

By sharing code through the `packages/ui` directory, you can maintain a consistent UI across your applications and avoid duplication of code.

Turborepo Cache

When you run `npm run dev`, Turborepo will automatically cache the build artifacts and dependencies of your packages and applications. This caching mechanism helps speed up subsequent builds and improves the overall development experience.

Turborepo intelligently determines which packages and applications need to be rebuilt based on the changes made to the codebase. It leverages the dependency graph to minimize unnecessary rebuilds and optimize the build process.

With this setup, you can efficiently develop and manage multiple applications and packages within a single repository using Turborepo. The shared code in the `packages/ui` directory allows for code reuse and consistency across your projects.

Exploring Root `package.json`

The root `package.json` file is located at the top level of your Turborepo project and serves as the central configuration file for your monorepo. It contains scripts and dependencies that are shared across all the packages and applications within the monorepo.

Let's break down each script:

1. `build` **Script**:
    - When you run `npm run build` in the root of your Turborepo project, it executes the command `turbo run build`.
    - This command tells Turborepo to run the `build` script defined in each package and application within the monorepo.
    - Turborepo goes into all the `packages/*` and `apps/*` directories and runs `npm run build` inside them, provided they have a `build` script defined in their respective `package.json` files.
    - The `build` script is typically used to compile, transpile, or bundle the code in each package and application.
2. `dev` **Script**:

- Running `npm run dev` in the root of your Turborepo project executes the command `turbo run dev --parallel`.
- This command tells Turborepo to run the `dev` script in each package and application within the monorepo, in parallel.
- Turborepo goes into all the `packages/*` and `apps/*` directories and runs `npm run dev` inside them, provided they have a `dev` script defined in their respective `package.json` files.
- The `dev` script is typically used to start the development server or watch for changes in each package and application.

3. `lint` **Script**:
   - When you run `npm run lint` in the root of your Turborepo project, it executes the command `turbo run lint`.
   - This command tells Turborepo to run the `lint` script defined in each package and application within the monorepo.
   - Turborepo goes into all the `packages/*` and `apps/*` directories and runs `npm run lint` inside them, provided they have a `lint` script defined in their respective `package.json` files.
   - The `lint` script is typically used to perform static code analysis and check for potential errors or style violations in each package and application.

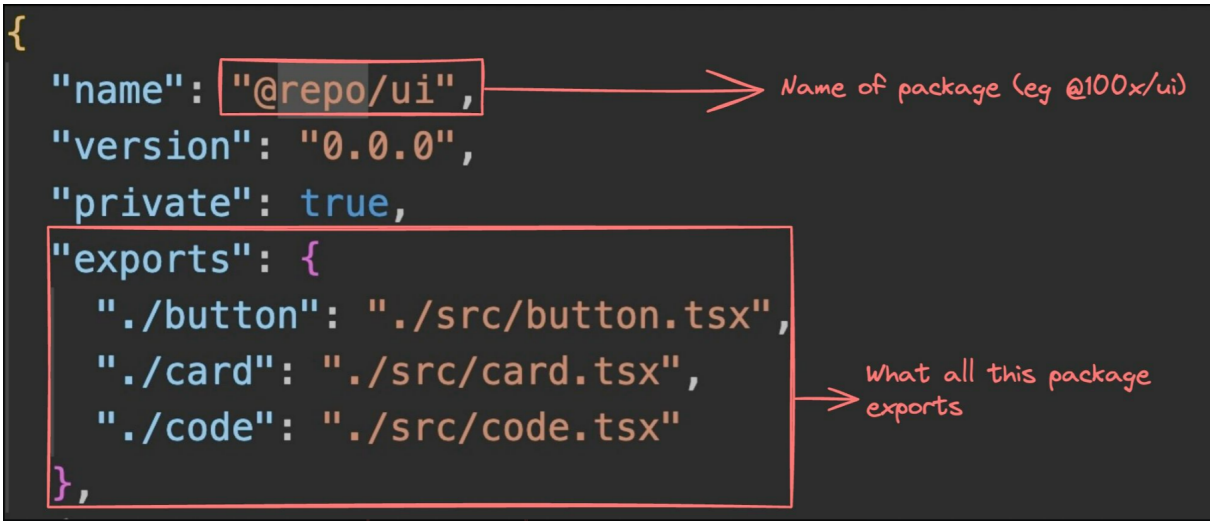
> By defining these scripts in the root `package.json` file, you can easily run the corresponding tasks across all the packages and applications in your monorepo using a single command.

Exploring `packages/ui`

Let's explore the `packages/ui` directory in a Turborepo project and understand its structure and purpose.

1. package.json

The `package.json` file in the `packages/ui` directory is used to configure and manage the UI package within the monorepo. It contains metadata, dependencies, and scripts specific to the UI package.



One important aspect of the `package.json` file in the UI package is the `exports` field. In the example you provided, the `exports` field is defined as follows:

```
{
  "exports": {
    "./button": "src/button.tsx",
    "./card": "src/card.tsx",
    "./code": "src/code.tsx"
  }
}
```

The `exports` field specifies the public API of the UI package. It defines the entry points for other packages or applications to import and use the components from the UI package.

In this case, the UI package exposes three components:

- `./button`: Mapped to the `src/button.tsx` file.
- `./card`: Mapped to the `src/card.tsx` file.

- `./code`: Mapped to the `src/code.tsx` file.

Other packages or applications can import these components using the specified paths, for example:

```
import { Button } from 'ui/button';
import { Card } from 'ui/card';
import { Code } from 'ui/code';
```

2. src/button.tsx

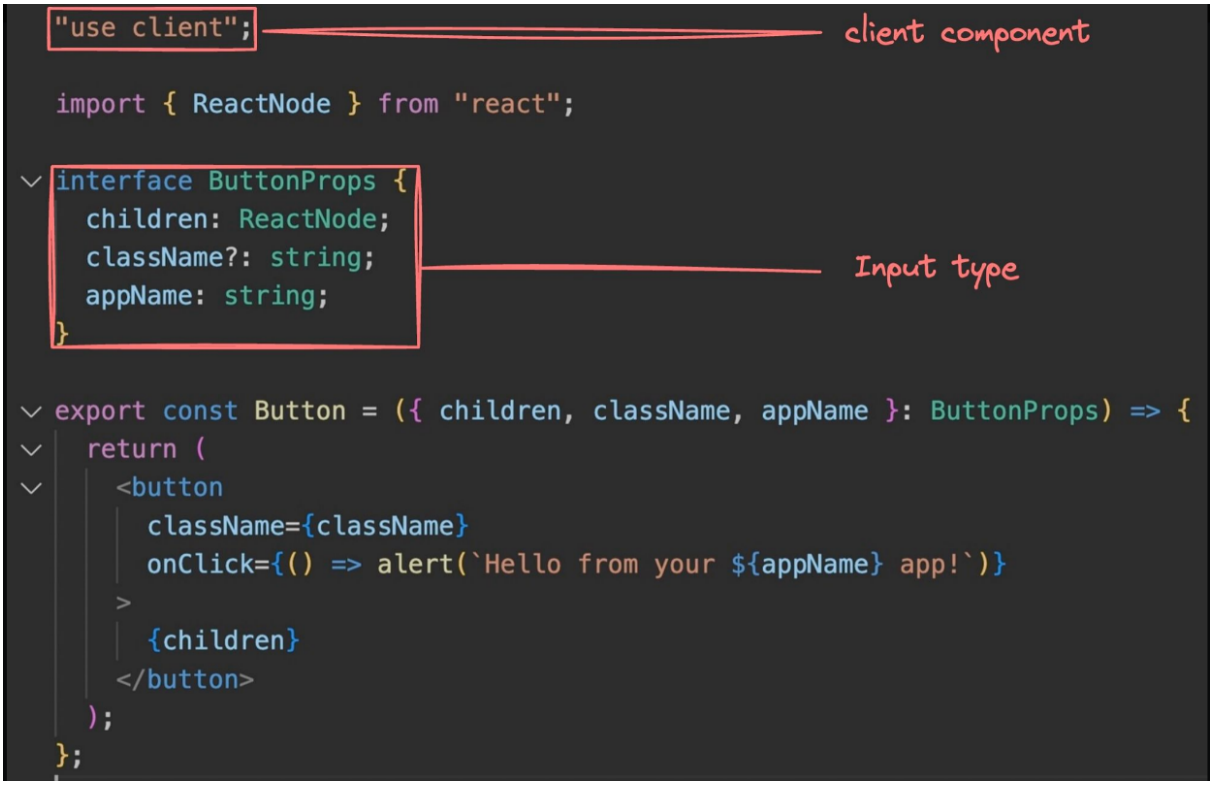
The `src/button.tsx` file contains the implementation of the Button component. Let's break down the code snippet you provided:

```
"use client"

interface ButtonProps {
  children: React.ReactNode;
  className?: string;
  appName: string;
}

export const Button: React.FC<ButtonProps> = ({ children, c
lassName, appName }) => {
  return (
    <button
      className={className}
      onClick={() => alert(`Hello from your ${appName} app
`)}
    >
      {children}
    </button>
  );
};
```

- The `"use client"` directive at the top of the file indicates that this component is a client-side component in a Next.js application.
- The `ButtonProps` interface defines the props that the Button component expects, including `children`, `className`, and `appName`.

- The `Button` component is exported as a functional component that accepts the props defined in `ButtonProps`.
- Inside the component, a `<button>` element is rendered with the provided `className` and an `onClick` event handler.
- The `onClick` event handler displays an alert with a message that includes the `appName` prop.
- The `children` prop is rendered inside the `<button>` element, allowing the content of the button to be customized.

This Button component can be imported and used in other packages or applications within the monorepo.



3. turbo folder

The `turbo` folder is a special folder introduced by Turborepo for code generation purposes. It is used to store configuration files and templates related to code generation within the monorepo.

Code generation in Turborepo allows you to automatically generate boilerplate code, such as components, pages, or API routes, based on predefined templates. This can help maintain consistency and speed up development by reducing manual repetitive tasks.

The `turbo` folder typically contains configuration files that define the code generation templates and their associated settings. These configuration files are written in a specific format (e.g., YAML or JSON) and specify the structure and content of the generated code.

For more details on code generation in Turborepo, you can refer to the official documentation: https://turbo.build/repo/docs/core-concepts/monorepos/code-generation

> By leveraging the `packages/ui` directory and its components, you can create a reusable UI library that can be shared across multiple applications within your Turborepo monorepo. The `turbo` folder and code generation capabilities further enhance the development experience by automating repetitive tasks and maintaining consistency throughout the monorepo.