

# Monorepo And Turborepo

*What is a monorepo?*

*-> Monolithic repositories or monorepos are code bases where multiple projects are stored in a single place. Code can be shared between multiple projects and versioned separately.*

*Monorepos are not monoliths:-*

*A monorepo is a code base where multiple projects or components are stored in a single repository. It's a way of organizing code that allows multiple, potentially unrelated projects to coexist in the same repository.*

*On the other hand, monolith is a type of software architecture where the entire application is built as a single, tightly integrated unit. All components, modules, and features are part of a single code base and run within a single process.*

*Workspace:-*

*Workspaces are a way of sharing packages inside a monorepo. A workspace contains multiple smaller packages, each having their own package.json file. Each of these packages can be shared across the workspace.*

Package managers like npm, yarn and pnpm have support for workspaces. For this article we will use pnpm as our package manager. The procedure is the same for any other package manager.

Creating a workspace:-

To create a workspace you need to have a pnpm-workspace.yml file at the root. There you can specify the packages you want to include in your workspace.

```
# make docs and everything inside
# packages and apps part of the workspace.
docs
packages/*
apps/*
```

In the above example, we made the docs folder and everything inside the packages and the apps folder part of our workspace.

Each package or app in your workspace must have a name mentioned in its package.json. The name is necessary as package managers use the package name to figure out where to install a package.

Say, we have three packages foo, bar and web (packages/foo, packages/bar and apps/web).

### *Installing a package inside a workspace project*

*Installing a package is same as how you would install normally, except you need to also mention where you want to install the package. For example, if you want to install react, in app/web, you need to do run the following command. In our case, the package name is same as the name of the folder.*

```
# pnpm add react --filter package-name  
pnpm add react --filter web
```

### *Sharing packages:-*

*To use a workspace package inside another package, you need to specify it as a dependency with the workspace: suffix in it's version. This tells the package manager to use a local version instead of downloading it from the package registry.*

```
pnpm i foo@workspace:* --filter web
```

```
{
  "dependencies": {
    "shared-package": "workspace:*"
  }
}
```

---

*Here, \* means the latest version, which saves us from needing to bump the versions of our dependency if the versions of our packages change. You can use a specific version too.*

---

## Turborepo

---

*Turborepo:-*

---

*Turborepo is a high-performance build system for JavaScript and Typescript code bases. It makes working with monorepos a lot simpler.*

---

*Tasks:-*

---

*Tasks are basically a batch of processes you can run. For example, if you run turbo run dev, it will run the dev scripts from each of your packages. You can also filter which scripts to run using the filter flag. Running turbo run dev --filter docs,foo will run the dev script inside docs and foo.*

---

---

You can configure the behaviour of these tasks. First create a `turbo.json`. Inside this file, you can list all the task and modify their default configuration. For example, turbo caches every build, you can change that by changing the cache property.

---

```
{ "pipeline": { "dev": { "cache": false } }}
```

---

If you want intellisense in your `turbo.json` file, add a `$schema` property and set it to `https://turbo.build/schema.json`.

---

If a task depends on other tasks, we can mention it.

---

```
{
  "pipeline": {
    "lint": { "cache": false },
    "publish": {
      "dependsOn": ["lint"],
      "cache": true
    }
  }
}
```

## Caching:-

Turborepo tries to cache results whenever possible. Basically, it keeps track of the inputs (your code) and outputs (your build outputs and terminal outputs). If the inputs changes, then only it will re-run a task, otherwise, it will just use the cached results.

For every task, you can set cache property to true or false. By default, turbo will cache a task. However, for the caching to work properly, we need to specify the build outputs, otherwise it will only cache terminal outputs.

The outputs options takes an array of glob paths that are generated as a result of building an application. These outputs will not be rebuild if the cache is valid. For example, in a next.js application, you would want to include the .next folder, as that is where the build output is generated.

Similar to outputs, you can also mention the inputs. By default, turbo will consider everything (except your outputs) to be your inputs.

```
{
  "pipeline": {
    "build": {
      "outputs": [".next/**", "!next/cache/**"],
      "inputs": ["src/**"]
    }
  }
}
```

---

*If you use the no-cache flag while running a task, turbo will not write to the cache (but it will use the cache if possible).*

---

*If you use the force flag, it will not use the cache (but it will write to the cache if possible).*

---

---

*Remote Caching:-*

---

*->Remote caching allows downloading and reusing cached results from a remote server.*

---

*It is particularly useful in CI/CD pipelines to speed up builds.*

---

*Cached results from previous builds are stored remotely and can be reused across multiple machines.*

---

### *Benefits of Remote Caching:-*

- a. Faster Builds – No need to recompute results.*
- b. Efficient Resource Usage – Saves CPU and memory.*
- c. Cross-Machine Consistency – Works across different environments.*
- d. Reduces Network Load – Only downloads what's needed.*

### *How Remote Caching Works:-*

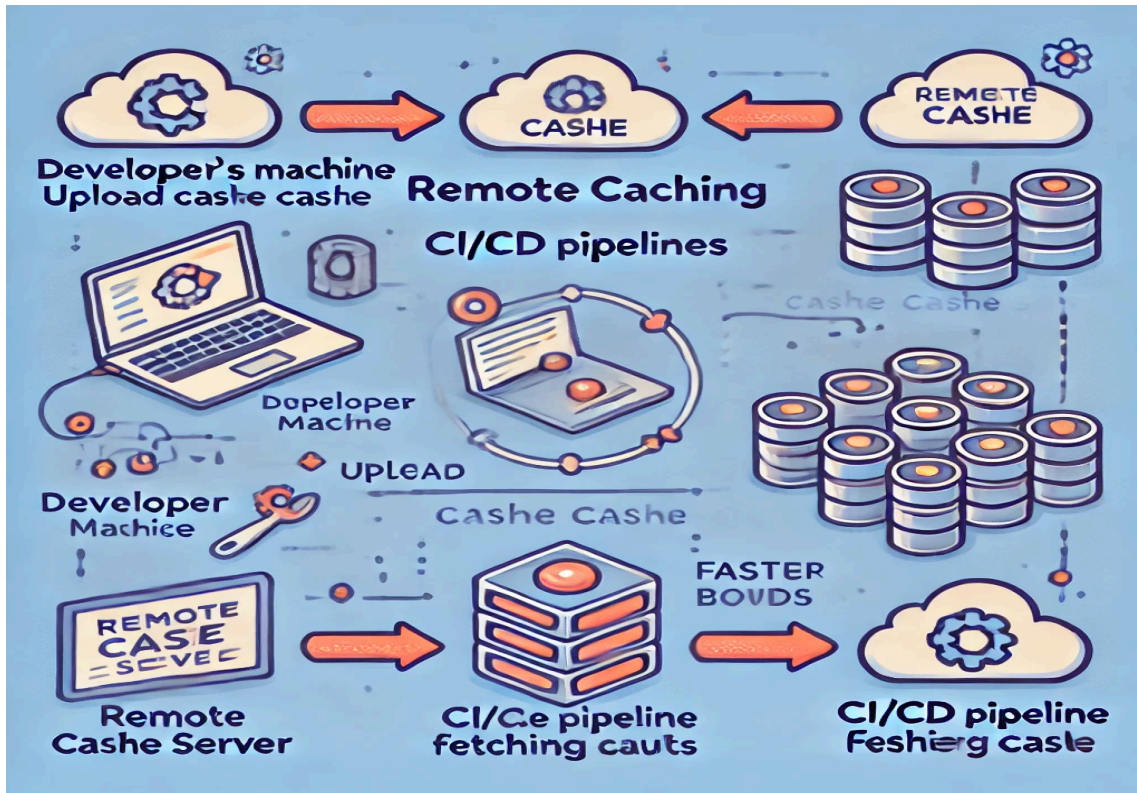
- a. Run a build – Outputs are cached.*
- b. Store results remotely – Cache is uploaded to a remote server.*
- c. Retrieve cache – Future builds download the cache instead of recomputing.*
- d. Reuse cache – CI/CD pipeline speeds up!*

## **Example Code for Remote Caching (Gradle Example)**

```
tasks.withType(JavaCompile).configureEach {
    outputs.cacheIf { true }
}

buildCache {
    remote(HttpBuildCache) {
        url = 'https://cache.example.com'
        push = true // Enable pushing to remote cache
    }
}
```

# Remote Caching Workflow



*This is the Remote Caching diagram showing the process of how caching works in CI/CD pipelines. Let me know if you need any modifications!*