#2 MonoRepo in Spago

Saravanan M

Teams often grapple with managing multiple applications and internal libraries. Traditionally, each project might inhabit its own repository, necessitating separate Continuous Integration (CI) processes, review protocols, and versioning. This *polyrepo* approach, while modular, escalates technical debt over time, especially when updates to shared libraries necessitate cascading changes across dependent applications.

Photo by Martin Lostak on Unsplash

MonoRepo

Monorepo addresses this challenge by consolidating all(related) projects into a single repository. This ensures that, at any given time, there's only one version of any library or artifact in use across all projects. By doing so,

it guarantees that all projects are consistently up-to-date with their dependencies, significantly cutting down on technical debt.

Monorepo according to this website means,

> A monorepo is a single repository containing multiple distinct projects, with well-defined relationships.

Note the word, "*well-defined relationships*" that's what which separates monorepo from monolith.

You can read more about monorepo in this website and understand why mono repo is not a monolith here.

Spago Support for monorepo

Spago uses a concept called workspace and packages to organize the code,

To quote from the previous article:

1 Intro to Spago — The Modern Purescript Build tool

A introduction to Spago- a superfast purescript build tool.

- **workspace:** The place where you define your whole project wide configuration like which package set to use, overrides, project wide-build options etc
- **packages:** Since a single purescript project can have N number of packages(monorepo/polyrepo). Here, you define dependencies specific to individual packages, along with package-specific build options, bundling configurations, execution settings, linting etc.

Drawing analogy

- If workspace is a building, then packages are the rooms.
- **A single workspace can have multiple packages and a package can belong to only one *workspace***
- A workspace cannot contain another workspace, similar to how a building cannot be inside another building. If it were, it wouldn't truly be a building🤓
- Any project wise configuration you want to do, you do it on workspace section and any individual package wise configuration you do it on package section.

Now to convert the above rule in terms of `spago.yaml`,

- In a mono-repo, **only one `spago.yaml` should have the `workspace` section** (if you have another `spago.yaml` then it doesn't come under this "mono" repo)
- Packages which are a part of the mono-repo should only have the `package` section(no `workspace` section)

Some gotchas

- Even though we may have different `packages` inside a mono-repo, they all get compiled into the **same output folder** (so packages can't have the same module name)
- When a folder/sub folder has a `spago.yaml` with a workspace section, it doesn't get detected as a package and will be ignored(won't get compiled as a part of the monorepo)

A Peek Inside a Spago Monorepo

Let's look at a file structure which better explains how a mono repo would like,

```
. monorepo-project
│
├── lib1
│   ├── spago.yaml
│   └── src
│       └── Main.purs
├── lib2
│   ├── spago.yaml
│   ├── src
│   │   └── Main.purs
│   └── test
│       └── Main.purs
├── src
│   ├── Main.purs
│
├── test
│       └── Main.purs
└── spago.yaml
```

Where:

- we have two libraries `lib1, lib2`and a application code (written within the root `src` folder).
- The `lib1/spago.yaml` would look something like this:

```
package:
  name: lib1
  dependencies:
    - effect
    - console
    - prelude
```

- then, assuming `lib2` depends on `lib1`, `lib2/spago.yaml` might look like this:

```
package:
  name: lib2
  dependencies:
    - aff
    - console
    - prelude
    - lib1 # <------ Note the dependency here
```

- the top level `spago.yaml` could look like this:

```
workspace:
  package_set:
    registry: 41.2.0

package:
  name: app
  dependencies:
    - prelude
    - lib2 # And we have `lib2` as a dependency
```

You may have observed that “aff” is a dependency of “lib2”. However, the root spago.yaml, which utilizes “lib2” as a dependency, does not need to declare it in its dependency list. This is because dependencies are resolved transitively(so `aff` also becomes a dependency of `app` indirectly)

No need for extra_package section!

Moving libraries into a monorepo significantly reduces boilerplate code. How?

Get Saravanan M’s stories in your inbox

Join Medium for free to get updates from this writer.

Imagine if `lib1, lib2, app` were each in their separate repositories. In this scenario, we'd have to manage explicit **`extra_package`** section in spago.yaml.

The *app's spago.yaml* would look like,

```
workspace:
    registry: 41.2.0
  extra_packages:
    lib2:
        git: https://github.com/user/lib2.git
        ref : v1.0.0
```

and lib2 also has to specify the same for `lib1` in its spago.yaml.

However, with a monorepo, we might **not need to maintain this `extra_packages`** section at all. Dependencies within our monorepo are automatically included in our current package-set

More than one application?

The above configuration had one only root application, but what if we have more than one application? Still we can use monorepo, the structure would be

```
. monorepo-project
│
├── app1
│    ├── src
│    │   └── Main.purs
│    │
│    ├── test
│    │   └── Main.purs
│    └── spago.yaml
```

```
│
├── app2
│    ├── src
│    │   ├── Main.purs
│    │
│    ├── test
│    │   ├── Main.purs
│    └── spago.yaml
│
├── lib1
│   ├── spago.yaml
│   └── src
│       └── Main.purs
├── lib2
│   ├── spago.yaml
│   ├── src
│   │   └── Main.purs
│   └── test
│       └── Main.purs
├── src
│   ├── Main.purs
│
├── test
│       └── Main.purs
└── spago.yaml

/* src: spago docs */
```

- root spago.yaml now will only be having the `workspace` section

```
workspace:
  package_set:
    registry: 41.2.0
```

- App1 and App2's `spago.yaml` will have their own dependencies specified.

- In this way, we can have any number of ***distinct*** application in our monorepo ensuring they all utilize the same package-set(same dependencies, same purescript version) and always uses the up-to-date internal libraries.

How to run a package specifically?

With the introduction of multiple sub-projects within our project structure, executing commands like `spago build` ,`spago test` etc requires a slight adjustment. Specifically, we must now specify the target package along with the spago command. Here's how:

```
$ spago build -p app1

$ spago test --package app2 // other options can also be passed
```

> ***Note:*** *using the* `-p` *or* `--package` *we can target a specific package*

While the monorepo approach has gained popularity in recent years, it's important to recognize that it isn't a one-size-fits-all solution. Here's when to consider each:

Use it when:

- You want all projects to utilize up-to-date dependencies.
- Projects have well-defined relationships/dependencies.
- Enforcing a set of rules (e.g., warning censorship, specific package-set) across all projects is necessary.

It's better to go with polyrepo, if your projects don't have a well defined relationship or **can't always operate on the same version of dependencies** Example `app1` may want `v2` version of `aff` but `app2` may strictly operates on `v1` version of `aff`. This kind of requirements are not

possible in monorepo, since at any point in time only one version of a package can be present (since in a monorepo, package-set is fixed across the whole workspace)

The cool thing is, spago supports having multiple polyrepos inside a single project. If you are interested, you can read more about that here.

I hope you found this article informative! This is just the beginning of our exploration of Spago. In the next installment, we'll delve deeper into configuring Spago projects, covering topics like linting, configuring the bundle, build options etc. So stay tuned!