Building large scale Angular Apps with Nx

OVERVIEW

This article explores issues when dealing with multi repos and outlines the scale of codebase, details mono-repository and the reasons the model was chosen.

PART 1

In “Implications of a multi-repo” we will discuss the result that can be drawn from multiple repository

PART 2

Describes what exactly mono repo is and the problems mono repo is addressing

PART 3

This part describes few checklists to get the maximum benefit of mono repository

PART 4

This section explains about Nx to get clear insight about it.

PART 5

A detailed view on how Nx workspace is different from Angular CLI workspace.

PART 6

Real benefits of mono repository are elaborated in this section.

CONCLUSION

Justifies the reason to go for mono repository.

REFERENCES

PART 1 IMPLICATIONS OF MONO - REPO

The fundamental law of repo topology is that you must not have cyclical dependencies between repos. On the surface, large and small organizations must care about robustness, maintainability and consistency between the repos when the code is kept scattered i.e. multi repo. What's different about large organizations is that they have hundreds of engineers building dozens of apps.

- It's very much essential to establish best practices, team standards, and use tools to promote them.
- Code ownership concepts become very important as the code size and complexity scales.
- Must explicitly define and automate the ownership model.
- Must rely on the automated CI process instead.

Implications of a multi-repo

Perhaps the main advantage of a multi-repo is that it avoids the tooling challenges of a mono-repo since existing tools are already designed to deal with the scale of project you're likely to put in a single repo. However, there are other issues, even if you believe that the multi-repo approach will get you productivity gains:

- You need to be very strict and intentional about managing versions and baking in the cost of updating code to use new versions of libraries it depends on. You may need to tool up around tracking dependencies globally to give people a proper understanding of what needs to be updated when.

- The way multi-repos force a certain loose coupling between projects becomes a drawback when reality dictates that things the used to be coupled should be decoupled or things that used to be decoupled need to be joined closer together. You will want tooling to make this not terrible.

- To mitigate the problems caused by having to work in multiple repos you'll probably want to provide some standardization of tooling, conventions, etc. Though this cuts against the basic tenet of the multi-repo philosophy.

- Finding code is harder when you first must find the right repo. At the very least you'll probably need to maintain catalogue of repos or strong naming convention that will allow someone looking at, say, a dependency in a build file, to figure out what repo they need to look in to find the code.

PART 2 Mono Repo and the problem it addresses

Mono - Repo:

- The repository contains more than one logical project. These projects are most likely unrelated, loosely connected or can be connected by other means.
- The repository is large in many ways: Number of commits, Number of branches and/or tags, Number of files tracked, Size of content tracked.

The anxieties that are there in multiple repositories are getting addressed in style when all your apps and its libraries are at one single place with proper version control. Working with and maintaining multiple applications and libraries is not an easy task.

Google to Facebook, Uber, Twitter and more, a good amount of large software companies handle this challenge by taking a mono-repo approach. It is an approach that is tested and proven to help companies be successful growing large code bases at scale, both with the code and with the teams that write and maintain it.

Google has shown the monolithic model of source code management can scale to a repository of one billion files, 35 million commits, and tens of thousands of developers. Benefits include unified versioning, extensive code sharing, simplified dependency management, atomic changes, large-scale refactoring, collaboration across teams, flexible code ownership, and code visibility. Google's codebase is shared by more than 25,000 Google software developers from dozens of offices in countries around the world.

Google repository statistics, January 2015.

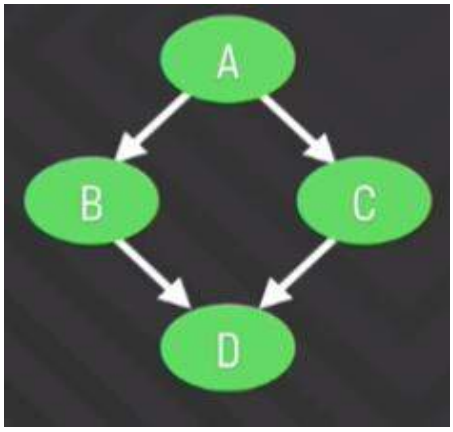
Total number of files	1 billion
Number of source files	9 million
Lines of source code	2 billion
Depth of history	35 million commits
Size of content	86TB
Commits per workday	40,000

On a typical workday, they commit 16,000 changes to the codebase, and another 24,000 changes are committed by automated systems. Each day the repository serves billions of files read requests, with approximately 800,000 queries per second during peak traffic and an average of approximately 500,000 queries per second each workday. The statistics shows there are over and above 40,000 commits and each commit triggers millions of testcase since every app resides in single repository.

What is the major problem 'mono-repo' is addressing??

To take a simple case: suppose you have code A and B that lives in one repo and code C that lives in another repo. If C depends on A and B depends on C even though there are no cycles in the code dependency graph, the cycle between the two repos means that updating A requires the following:

- Change A and publish a new version of A's artifact.
- Go to C's repo and change C to consume the new version of A.
- Publish a new artifact for C.
- Go back to the A/B repo and change B to consume the new C artifact.



This require at least three separate CI runs as well as probably some branching and merging since the changes to A are incompatible with B until it is updated to the new version of C.
Suppose both Libraries B and C depend on X and Application A depends on both B and C. This is the classic "**diamond dependency**" that has the consequence that A can only depend on versions of B and C which agree on what version of X they want. So, if A wants to upgrade C to the latest version but that version has moved ahead to a newer version of X than the latest B is on, A is stuck until B moves forward. Even a half diamond, where A depends directly on X and B which also depends on X, is sufficient to cause problems if A wants to be on a newer version of X than B is. Now imagine a more complex dependency graph and you can see that the coordination costs of upgrading of a low-level library are going to be non-trivial. *To conclude, it may be impossible to build A and it may be very difficult to release library D.*

The two main areas that can have the biggest impact on your organization are:

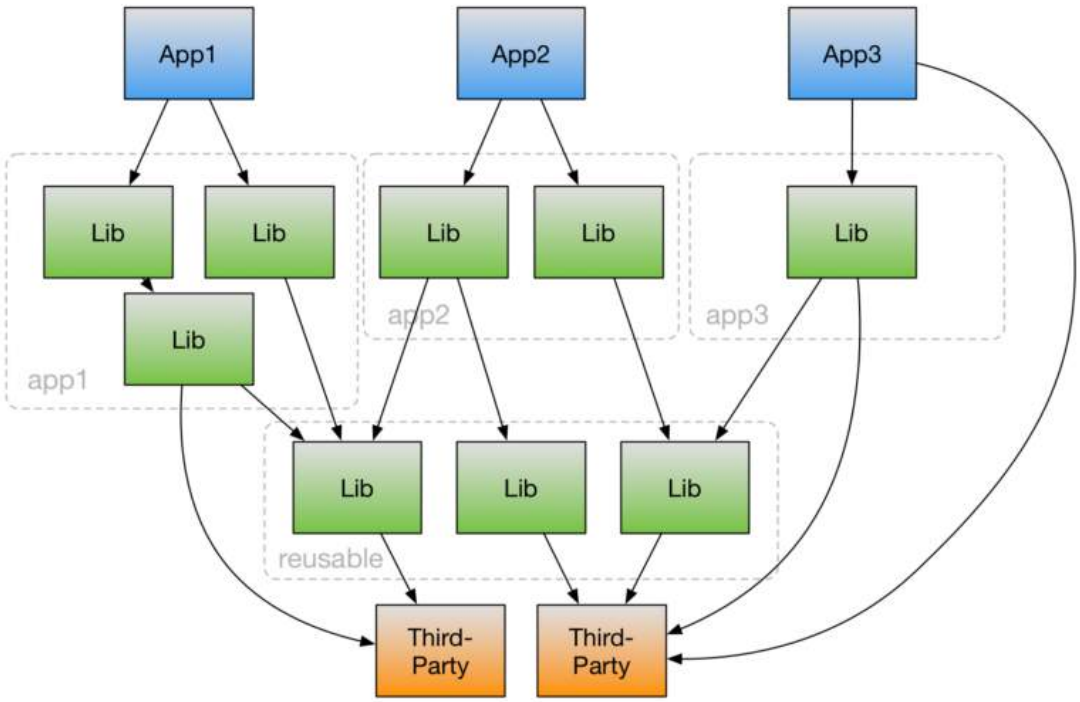
1) source code and dependency management,

2) promoting best practices and automation.

1) Source Code/Dependency Management

As I mentioned above, large organizations have a lot of source code and a lot of people make changes to source, packages, deployment, etc. So, figuring out how to store it, how to effectively make changes to it, how to set up dependencies between different projects, how to build and release them are the questions you need to answer early on. Once you start examining these questions, you will see that not all projects/modules are alike.
In a typical setup, there are four categories of projects/modules:

- **Apps**. These are your products, something you ship to the user.
- **App-specific libs.** These are apps' sections that can be developed and tested independently.
- **Reusable libs.** These are your components, services, utilities used in many different apps.
- **Third-party libs and tools.**



Credits/ Src : Nrwl

To be effective, developers should be able to:

- Create app-specific libs
- Extract reusable libs
- Verify that a code change to reusable lib does not break any apps and libs that depend on it
- Refactor multiple apps and libs together
- Determine who is responsible for a app or lib

The ease with which developers can do all of these has a big impact on your teams' velocity and your projects' code quality. If it takes a day to create a new reusable library (create a repo, set up CI, publish it to a local npm registry), few will do it. As a result, developers will either duplicate the code or put it in a place where it does not belong. When it is easy to scaffold and configure a reusable library within minutes (instead of days) only then will developers invest in building and maintaining reusable libraries. If multiple applications have dependencies on multiple libraries, regression testing can become very difficult when a library changes. If it is impossible to automatically verify that a change to a reusable library does not break any apps depending on it, developers will be afraid to make changes and will write defensive, brittle code.

2) Promoting Best Practices and Automation

Code reviews and the word-of-mouth are great ways to promote best practices in small organizations. The larger the organization gets, however, the more resource intensive, error prone, and inconsistent this process becomes. Use tools to remedy this problem.

Create Small Reusable Libraries

Often the simplest thing can have the biggest impact. Take, for instance, creating small reusable libraries. Even by extracting only 30 lines of code into a reusable library, you can make your teams much more productive and save many weeks of work.

For instance, every single application the team at Nrwl looks at has a lot of race conditions. Some of these are obvious and some are very subtle and hard to track origination. Even the most seasoned, senior Angular developers overlook subtle race conditions. After helping many clients resolve these types of issues, we were forced to ask ourselves: "Can we build a small tool to eliminate (or avoid) code manifesting race-conditions?". As a result, we implemented a 50-line service that integrates with NgRx Effects and eliminates many of the race conditions seen in Web applications. This critical feature is so small that you can read through it and understand what it does in minutes. We made it part of Nx. Encourage your developers to do the same. You can start by focusing on the three areas that tend to cause the most problems: testing, state management, and routing.

Create Schematics for Code Generation

Angular CLI uses the schematics library for code generation. Nrwl/Nx is built on top of the Angular CLI, and it passes a custom collection of schematics tailoring the CLI experience for the multi-app/multi-lib setup. Enterprise teams should take it one step further and create a collection of schematics used in your organization. For instance, if you use mock data in your integration tests, create a schematic to generate the fixture. Measure how long it takes for a developer to implement a simple source code generator. Only when it takes minutes, folks will do it.

Create Custom Lint Checks

TSLint is good at two things: making sure developers do not make costly mistakes and making the source code more consistent. TSLint can be run on the CI and integrates well with all major editors and IDEs.Embrace this tool.

Automate Everything (Use Formatters)

It's surprising how few organizations set up things like automatic code formatting. Largely, it is because many believe it is not the most important thing to focus on. And I would agree if it took weeks to get it right. But it often takes 30 minutes, and it has a surprisingly large effect: it makes the source code more consistent, and it eliminates a whole class of problems. The biggest benefit with formatters and team usage is commits and code reviews. Then only the actual changes show as deltas, instead of myriad formatting changes obfuscating a few critical code changes.

PART 3 Checklist to get best of Mono Repo

This is a list of things (a checklist) architects at large organizations can do to make their teams more productive with Nx Workspace(figure) and Angular CLI.

1. Define **the set of tools you will use use to build Angular apps (CLI, Nx, NgRx)**. Developers like to build custom tools, but using standard tools is almost always a better option, at least in the long run. Create an exemplary repo built with the tools.

2. Define **a process for creating/extracting a new library**. Measure how long it takes.

3. Define **a process for verifying that a code change to a reusable lib does not break any apps and libs that depend on it**. Treat the dev machine and the CI experiences separately. Measure how long it takes.

4. Define **a process for refactoring multiple apps and libs**.

5. Define **a process for assigning and checking ownership**. Without a good ownership model in place a chaos will ensue.

6. Define **a source control management strategy**: trunk-based development or feature-based development. Try to make it the same for all apps and libs.

7. Define **an explicit policy of managing third-party dependencies**.

8. Define **state management and side effect management best practices**. Automate as much as possible.

9. Define **testing best practices**. Automate as much as possible.

10. Create **an organization-specific tslint package** from the get go. It's a great way to promote best practices.

11. Create **an organization-specific schematics package** from the get go. It's a great way to promote best practices.

12. Automate everything that can be automated (e.g., **set up automatic formatting**).

Unsurprisingly Nrwl Nx helps with a lot of these. After all, that's why we created it. But as with most things, it is not about using a tool, but about choosing tools and processes carefully.

PART 4 What Nx gives you?

Credits/ Src : Nrwl

Productivity

Nx helps your team move quickly when setting up enterprise Angular applications by providing a default setup for complex projects. It adds the monorepo-style development to the Angular CLI, which enables fast project development, team collaboration, and code reuse. Nx also provides runtime libraries to simplify complex tasks with state management and testing, so developers can spend more time implementing features and less time wrestling with setup and testing. Finally, Nx provides an easy path for otherwise complex tasks, such as setting up AngularJS upgrade in hybrid mode for an iterative approach to upgrading applications.

Consistency

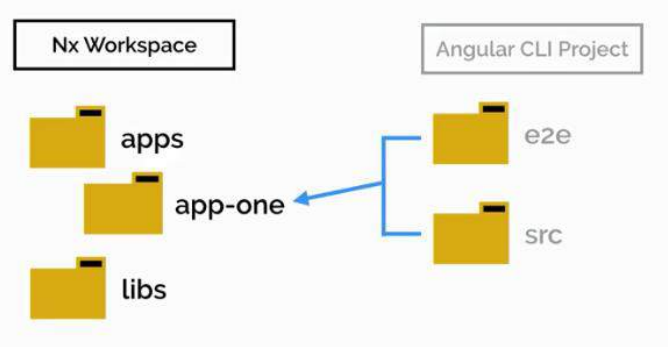
Out of the box, Angular CLI follows the Angular style guide and helps enforce patterns and practices within application folder/file structure, naming conventions, component organization and more, providing a good foundation for most applications. Nx builds on the foundation provided by Angular CLI, and provides a more opinionated architecture, which allows Nx to provide more advanced code generation and analysis. With stronger conventions around code organization and state management in place, teams are able to share more knowledge, code, and infrastructure.

Safety

The contributors to Nx have worked with many teams building massive Angular applications and have found common mistakes and antipatterns which can lead to race conditions, performance issues, or code maintainability challenges. Fortunately, many of these mistakes can be identified by static analysis at development time.

PART 5 Nx Workspace Vs CLI Workspace

Each folder inside libs will have its own package.json specifying it's dependencies and npm scripts if needed.



The dependencies within each package can be external or within the same repository.

Sharing code across multiple repositories

Once you start feeling the need to share code across multiple applications you may choose to start creating different repositories and publishing them as npm packages.

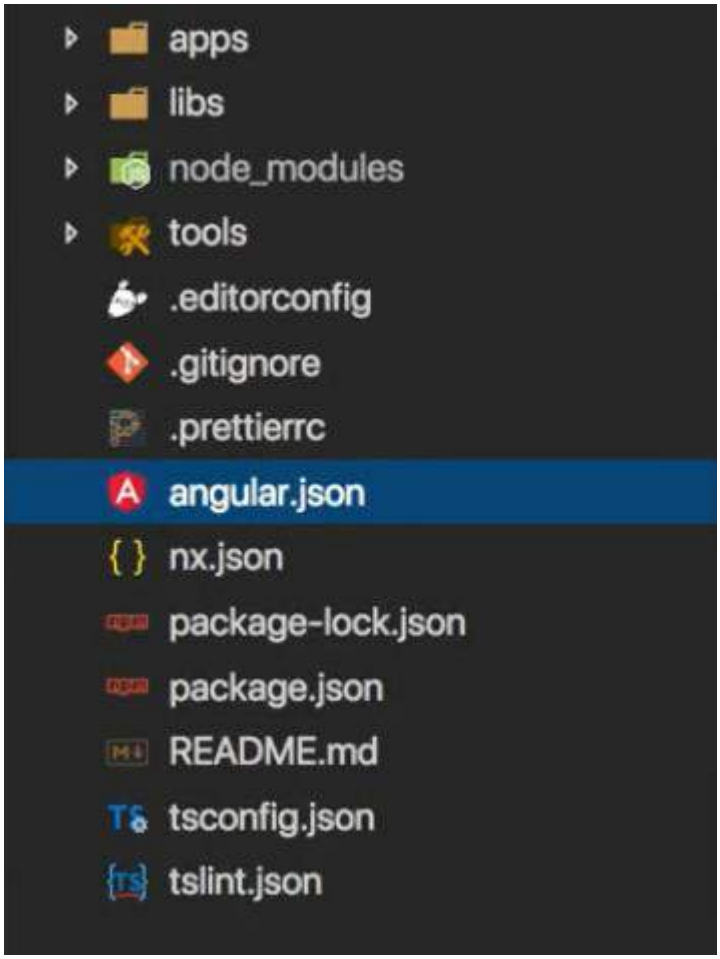
As the number of applications and shared components scales you start facing some issues:

Testing: To properly test a shared package you'll need to locally link several different projects, this can become a daunting task as the number of interdependent packages grows and npm linking can sometimes yield unexpected behaviours.

Publishing: Propagating a fix might require updating and publishing several different interdependent projects in a correct order.

Discoverability: Available reusable components may become hard to find.

Refactoring: Refactoring may lead to updates across multiple packages which then leads to the testing and publishing issues mentioned above.



PART 6 Real benefits of using mono repo

Below are some more specific advantages to going with a monorepo workspace.

- Unified versioning
    - Everything at that current commit works together
    - A label or branch can capture the same
- Promotes code sharing and reuse
    - Easy to split code into lib modules
    - Easy to consume/implement that code and the latest changes to it
- Easier dependency management
    - One node_modules for all code
    - One build setup (e.g., Angular CLI)
- Refactoring benefits
    - Code editors and IDEs are "workspace" aware
    - Can have a single commit for a refactor that spans applications in the domain
- Consistent developer experience
    - Ensures all necessary dependant code is available

CONCLUSION

The JavaScript ecosystem already seems to have all the pieces available to make these concepts work properly, they just aren't provided as single unified solution.

This seems to be a great model to allow the development of applications by multiple interdependent teams that can work on multiple packages and applications while keeping all the parts tightly integrated. This model makes sure your application does not have race conditions, that the code looks consistent, and that it is fast.

If you ask it to build and test using naive build system, it will build all the code in the repository and run all the tests. It does this each time you make a change.

However, the daily resource demand for naive CI is theoretically quadratic O(C × T):

C is the number of changes committed by all engineers per day
T is the cumulative resource requirement of all the tests

A monorepo increases these factors a lot. On your team, C is close to constant (you don't grow headcount that fast) but it's linear or exponential in a big company. T increases slowly on your project, (and you have an incentive to keep your tests fast) but across the whole company, you might be running many other teams' tests that are more resource-intensive.

But what you pay for with the slow-quadratic growth is totally worth it. Today you integrate changes infrequently, after you cut a release and most of your user's upgrade. Adopting a monorepo means every change can be integrated. If you believe in Continuous Integration (a requirement for Continuous Delivery) then you should seriously think about using a monorepo, and a build system that helps you keep up with the greater amount of testing.

REFERENCE

Nrwl Extensions for Angular
Mono repos at google
12 Things to Help Large Organizations Do Angular Right
Mono repos in the wild

Author Info:

Saleem Malik Raja is with HCL Technologies for past 1year and 6months in Experience Design and Engineering (EDGE) Practice. He has extensive experience of 4.5+ years in developing effortless experiences for Digital Products and loves anything that involves the creative process from concept to execution. He loves exploring new methods to manifest his ideas, from illustrations to digital media.