DSpace 8 Angular : library-based architecture proposal

Goal

This proposal has the goal to try to provide a solution in order to be able to modularize the DSpace Angular architecture. This need was already expressed during one of the last dspace visioning meeting group :

DSpace Product Visioning Group 2021.

So the main porpouse is to define and to provide a new architcture allowing, for example, that the dspace's core functionalities could be used by an external plugin which is not part of the dspace's code repository.

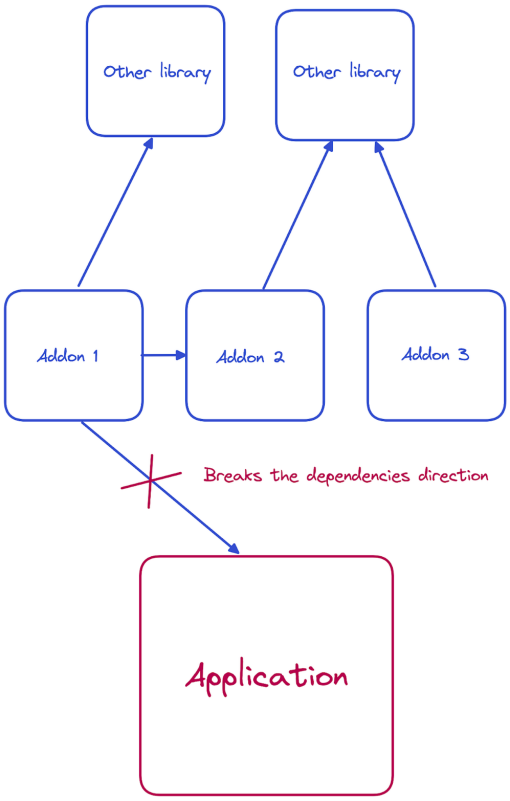
How?

Our application has been built using a **monolithic architecture**, where all the components and functionality are tightly coupled within a single codebase. While this approach has served us well till now, it has also introduced challenges in terms of the ability to introduce new features and updates independently from the codebase.

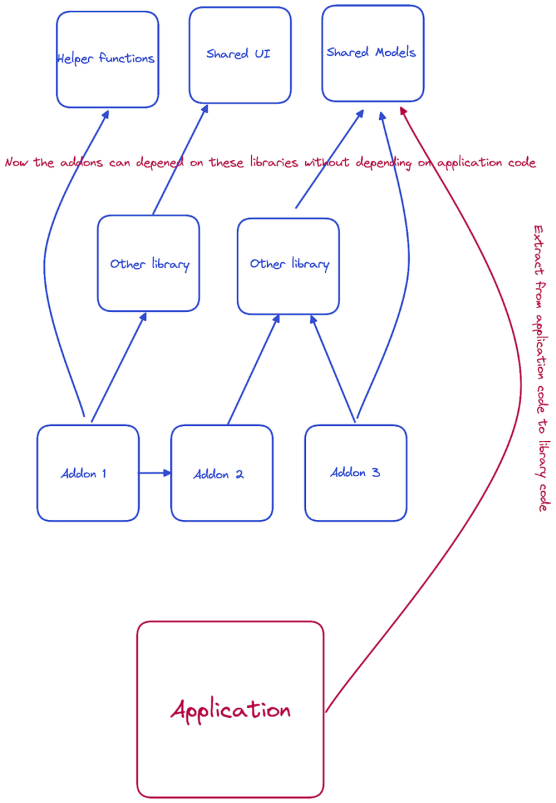
In order to move to a more modular architecture, we propose to migrate from a **monolithic** approach of building apps to a **library-based** architecture here each feature is **built in isolation**.

Why?

With the current monolithic architecture if any DSpace adopter needs to implement an additional DSpace feature/addon they **can't depends on the application code.** Because, if the addon depends on core functionalities, helpers functions, or shared components in the application, it means it depends on application code and this breaks the libraries dependencies direction.



Moving to a library-based architecture allows to break down our Angular application into smaller, self-contained libraries or modules. These libraries can be reused across multiple projects or addons. The application can depend on these libraries just like it did before, but now in a more structured way.



With a library-based architecture, we can separate different concerns and responsibilities into individual libraries. This separation enables better organization and maintainability of our codebase. For example, we can have separate libraries for core functionality, UI components, data services, authentication, and more, making it easier to modify specific parts of the application.

Implementation Plan

It will require **massive code changes** and refactorings and probably will make **breaking changes,** so that's why it's recommended to start it with DSpace 8.

We can incrementally introduce these changes, by firstly analyzing our existing Angular application and identifing functionalities and components that can be separated into individual libraries. We should consider to break down the application based on different concerns such as core functionality, shared UI components, data services, authentication, or any other logical divisions that make sense for us.

How to deal with it?

Moving all things to libraries will create a lot of files, and they will need to be managed. Also, we will need to fix the build rules because we can't build the application before building the libraries it depends on. **So, we will need a build orchestration tool,** and that's where the NX comes in. With NX, all these can be done in a predictable and standardized way.

quote

Nx is a powerful open-source build system that provides tools and techniques for enhancing developer productivity, optimizing CI performance, and maintaining code quality. Learn more about how Nx works.

You can use Nx to quickly scaffold a new standalone project or even an entire monorepo. It can be incrementally adopted and will grow with your project as it scales.

How does all these help?

- Faster builds (build caching)
- Better code reuse (specific libraries for each usecase)
- Faster dev time because of provided schematics
- Better code visibility (project deps graph)
- Dependency contraints
- Easy to introduce new addons without caring about build time or bundle size

Helper links:

- https://nx.dev/more-concepts/library-types
- https://nx.dev/more-concepts/dependency-management
- https://nx.dev/more-concepts/code-sharing
- https://nx.dev/more-concepts/creating-libraries
- https://nx.dev/more-concepts/applications-and-libraries
- https://nx.dev/more-concepts/how-project-graph-is-built
- https://nx.dev/recipes/environment-variables/define-environment-variables

POC Analysis Overview

POC BRANCH IS AVAILABLE HERE

What we've done

- migrate to standalone components
- add NX to the project
- create examples of how the library can be created https://github.com/4Science/dspace-angular/tree/f/ej/nx-poc/libs

Standalone migration

https://github.com/DSpace/dspace-angular/issues/2370

By migrating to standalone using the schematics provided by Angular, we solve the issue of not knowing component dependencies. Because the migration now puts the components deps directly in the imports array of the component, we can easily understand what depends on what and move these components around without breaking anything.

Also, by understanding the components deps, later on we can move these components to libraries in a non-breaking way because everything will be explicit, while doing that while we still use modules will be hard and error-prone, because of implicit dependencies.

Benefits

- Easier future refactorings
    - Moving to other folders
    - Moving to libraries (nx)
- Smoother learning curve for new devs
    - Junior devs don't have to understand or see what SharedModule has imported and exported
    - Easier to reason about what the components depends on
- Easier lazy loading
    - Before standalone we couldn't safely lazy load a component without also loading it's module (missing providers), while now we can lazy load them and be sure that it won't break at runtime
    - Easier routing lazy loading per component using **loadComponent**
    - No need for {FEATURE}RoutingModule to do lazy loading of routes
- Better compilation time
    - Because the angular compilation scope will have to compile everything that's changed, when using a SharedModule all the components will be affected if we change a component, while when using standalone the compilation is optimized because only that component and its dependants will have to be compiled.

Migrating to NX

Angular CLI is great, until the project becomes too big and compilation time reaches **15 minutes** and **ng serve** takes **4 minutes**. That's not Angular's fault, but just current tools being used by Angular CLI to compile our apps. The solution to performance for almost all things in programming is caching. Angular CLI does implement caching in cold builds (filesystem caching), but it's not enough to fix our issue. Here comes NX, NX in the beginning was just Angular CLI on steroids, but it has grown to be much more than that. With NX we can create libraries that have their own compilation scope, and that need to be compiled only when changed, and their compilation result will be re-used until it's changed again.

Benefits:

- If we chunk our app into smaller **buildable libraries** we can shrink down that compilation time from **15 minutes** to **2-3 minute** in best case scenarios.

- Parallel builds
- Build graph tools
- Task orchestration
- Tooling to manage a big repo of libs and more

Issues with the migration and possible roadmap:

- We have a big **SharedModule** and **CoreModule** that depend on each other, so migrating these to libraries is not easily doable without the standalone migration.
- We have to start with components or util function that don't depend on anything else except themselves (finding root files) and they are not so easy to find in this big project
- We have to define the folder architecture we want to follow based on our needs
- It will move a lot of files around in folders, so maybe this is an issue for clients that have extended Dspace in their own branches (syncing repos becomes harder)
- It is hard to chunk services as they have a lot of hidden deps because of module providers
- We will have to update how some parts of theming feature works because if we move components to libs they cannot depend on application code, so we will have to make the theming configurable from app part.

Things to keep in mind:

Misconception

> *Developers new to Nx are initially often hesitant to move their logic into libraries, because they assume it implies that those libraries need to be general purpose and shareable across applications.*

This is a common misconception, moving code into libraries can be done from a pure code organization perspective.

Ease of re-use might emerge as a positive side effect of refactoring code into libraries by applying an *"API thinking"* approach. It is not the main driver though.

In fact when organizing libraries you should think about your business domains.

Most often teams are aligned with those domains and thus a similar organization of the libraries in the `libs/` folder might be most appropriate. Nx allows to nest libraries into sub-folders which makes it easy to reflect such structuring.