Guide

ANGULAR WORKSPACE

The Ultimate Guide

Learn how to leverage the capabilities of the Angular Workspace to elevate your developer workflow.

Techniques and strategies to minimize effort and maximize results.

AngularArchitecture.com

build something amazing

Vers. 6, 7, 8 & 9

MATT VAUGHN

Angular Workspace - The Ultimate Guide

Matt Vaughn

This book is for sale at http://leanpub.com/angular-workspace-the-ultimate-guide

This version was published on 2020-09-23

Contents

What is an Angular Workspace? **1**
Workspace: Where You Work Matters 1
May the 4th Be With You 2
What Is So Special About Angular version 6? 2
Monorepo with Angular Workspace 3
Angular Workspace Improves Developer Efficiency 3

Workspace: A Container for Projects **5**
Angular.json 5
Applications 6
Libraries 13
Schematics 19

Anatomy of an Angular Application **20**

Share and Reuse Code **21**
Why Share Code? 21
How Can We Do It? 21

JavaScript Packages **22**
Not Effective? 23

Nx Dev Tools **24**
Upgrade Existing Angular Workspace 24
Getting Started - New Workspace 24
Create a New Workspace 25

Code Organization **28**
Architectural Patterns 28

Library Project Types **29**

Benefits of a Monorepo Workspace **30**

Resources **31**

What is an Angular Workspace?

- multiple projects in a single development environment
  - application
  - library
- other monorepo examples
  - .NET and Java –> solutions of different projects
  - JavaScript: Lerna
  - Nrwl.io Nx Dev Tool

Workspace: Where You Work Matters

Most everyone agrees that a nice workspace is not only enjoyable but essential to being effective. Angular gave us an impressive gift with version 6. This chapter provides an overview of the game-changing work environment for developing web applications. For years, other development environments (i.e., .NET, Java, etc.) have had rich IDEs that provide environments to create solutions with multiple projects of different types.

Programmers require the ability to organize code and projects related to each other. Today, the requirements for web applications require the same capabilities.

Web developers typically have a folder and the entire code base for the project is in that folder. There is no concept of sharing and reusing libraries in a solution (or workspace). If you want to create a JavaScript library, it is typically developed as a separate project, published to a package manager, and then consumed by other JavaScript applications. This is a workflow that has many steps - not a very efficient or effective workflow for rapid development. I know, because my team did this for Angular 2 applications up through version 5 for about 11 or 12 projects. Some of our packages had dependencies on other custom libraries. A small change in one library would require us to build and deploy several other libraries due to our dependencies. This was very time consuming.

So, what if there was a work environment that enabled you to manage multiple web application projects, create libraries, and share/reuse those libraries with any of the web applications and/or the other libraries without the overhead of publishing and consuming packages via npm? It was quite a surprise when Angular 6 was released with support for a Workspace environment that does all of these things.

The Angular Workspace at first appearance may not seem like much. The ability to add multiple projects is significant. However, the new supported project type for library projects may provide the biggest impact. It allows us to build modern web applications with new architectural patterns that simplify and make our code more powerful.

> This book will focus on the changes the new Angular Workspace provides to implement strong enterprise-level architectures.

The Angular Workspace simplifies the management of multi-project environments. All projects now share the same package dependency configuration, the same Angular version, a single configuration for building, serving, linting, and testing projects. We got all of this with Version 6 of Angular!

> We can have great web application architecture that is clean without the Angular Workspace. However, the additional CLI tooling along with the ability to share and reuse code via libraries is the biggest game-changer to web development workflows and work environments. Individuals and teams that leverage these capabilities will certainly be more efficient and effective - this means a lot in a competitive software industry.

May the 4th Be With You

On, May 4th, 2018, Angular 6 was released. The Angular.io[1] site actually says May 3rd. However, it appears that the press releases were pushed the following day. And I like the sound and reference to using the force with the date that most recognize as the public release date.

What Is So Special About Angular version 6?

The Angular CLI command to create a new project was dramatically upgraded. This command no longer creates a project. It creates a **Workspace**.

```
ng new <my-workspace-name-here>
```

Most developers didn't notice much. The command above still worked - there were no breaking changes. I think there should have been, because the change was so significant - I think the new capability flew under the radar for awhile. This command created a new Workspace with a default application project where both had the same name. In fact, the default application project was in the same location as before. There was no indication that anything different was going on. Yes, there was a new angular.json file.

If you inspected this file, you noticed that there was a projects node with the default project containing a project type of application. Interesting.

```
"projectType": "application"
```

> Note: Angular version 7, allows for an empty workspace with the `ng new` command. You would use the `ng generate application <name>` to add a new application to the workspace.

[1]https://angular.io/guide/releases

Monorepo with Angular Workspace

We just got a monorepo. A single environment where we can now develop multiple projects. Not just single-page web applications, but also library projects (more on this later).

Is this a good thing? It is for several reasons. Think about all of the time that could be saved without the overhead of configuring development environments. A monorepo allows a team to continually improve the workspace environment - and everyone benefits. It is like a community garden that continually gets better and always produces fine results.

One of the main goals of Angular is to enable companies, teams, and individual developers to be as effective as possible. Effectiveness also includes efficiency and building the right things at the right time. Therefore, to enable and provide the necessary environment to develop effective Angular solutions, we require the implementation of a single Workspace for all things Angular. This includes:

- all Angular applications
- all Angular and Typescript libraries (cross-cutting concerns, domain libraries, core and foundational libraries)
- all tooling (Schematics)
- documentation
- linting
- testing
- build

This environment allows a team to establish, monitor and improve patterns and practices as a collective. This does not mean the removal of specialization and/or creativity within the team or individual developers - however, it provides increased transparency, knowledge sharing, workflow and collaboration for the many in a single work environment.

A good example for my current team is code formatting. This may seem like a trivial thing. However, HTML formatting became very important in regard to internationalization (i18n). We were able to use tools like Prettier - with a single configuration that applies to all applications and libraries in our Workspace. We also added a pre-commit hook to format our Typescript files before commits to our repository. This level of consistency and maintainability is very impressive.

Angular Workspace Improves Developer Efficiency

When you are working on any project whether it is software, home, automobile, or other hobbies; the setup and breakdown of your work environment takes a lot of time.

- getting the tools together in a single location
- clearing out a space for work

- retrieving and putting things away
- cleaning up after the work

Although important, the setup and breakdown of a project is considered waste or non-productive time, because you are not building features. To solve this problem, a monorepo allows you to focus on a specific project - not setting up an environment for each new project.

There is a lot of overhead and maintenance of a project beyond the actual domain-specific implementation. Before Angular 6, if you had 3, 5, or 10 Angular applications, you had as many Angular projects. Depending on when they were created, they may vary with different versions of Angular, 3rd-party packages, or Typescript. Keeping all of your projects in-sync with version updates is/was a time consuming process. I have seen older projects just left alone without any upgrades.

The new Angular Workspace solves this problem. The workspace is a single work environment where all projects share the same runtime and development dependencies (packages). They all use the same version of dependencies and they all share the same package.json version. When you upgrade the workspace, all projects are updated at the same time. If there are any problems or issues, you have early notification during the build and serve of your applications.

> No project left behind.

Instead of setting up and managing each project independently, you have a single-source of truth and place to manage all of your Angular projects.

If you develop custom Angular libraries and publish them as packages, you doing a very noble thing. Sharing and reusing code is very effective. However, the process to build, publish, and consume library updates is not a very efficient process, right? There are many steps. There are even more steps if you have libraries that have dependencies on other custom libraries. The Angular Workspace solves this problem with the new project type: **library**.

For more information on this topic, please read my blog post: Monorepo + Angular Packaged Libs :: You Can Have Your Cake and Eat it Too!![2].

[2]https://medium.com/@angularlicious/monorepo-angular-packaged-libs-you-can-have-your-cake-and-eat-it-too-8c5687c4ffe9

Workspace: A Container for Projects

The Angular Workspace is an environment to manage multiple projects. The `angular.json` file contains the configuration for each project in the `projects` node of the file. There are currently (2) supported projects types.

1. application
2. library

Our reference workspace for this book contains a project named `lms` (learning management system). Using the CLI command creates the application project and files in the Workspace. The CLI updates the `angular.json` by adding the project definitions to the `projects` node.

```
ng generate application lms
```

Angular.json

Configuration for any application is a big topic. If you had 10 applications and 10 different teams, each application would most likely have a different configuration and quite possibly many variations or deviations depending on your perspective. What if there was a way to have a conventional and reliable configuration for your projects where there was no deviation between projects, teams, or even companies?

The `angular.json` configuration and CLI provides the answer. We do not need to spend time coming up with configuration for building, testing, linting, serving, and even localizing (l10n/i18n) our applications. It is already done for us - focus on building something amazing.

The Angular.io[3] website contains a nice overview of the Angular Workspace Configuration. The following properties, at the top level of the file, configure the workspace.

- version: The configuration-file version.
- newProjectRoot: Path where new projects are created. Absolute or relative to the workspace folder.
- defaultProject: Default project name to use in commands, where not provided as an argument. When you use ng new to create a new app in a new workspace, that app is the default project for the workspace until you change it here.

[3] https://angular.io

- schematics : A set of schematics that customize the ng generate sub-command option defaults for this workspace. See Generation schematics below.
- projects : Contains a subsection for each project (library or application) in the workspace, with the per-project configuration options.

There is another project type that doesn't fit into the category of either application or library. There is a **Schematic** project type that is used by the CLI and provide utility-like features to generate, update, and remove items from specific projects and Workspace environments.

Applications

Application projects are where most developers spend their time. These projects require a runtime element, are hosted, and typically run in a browser. The runtime elements of an Angular application are HTML, JavaScript, and CSS. How we get there is another story. There are (2) levels of configuration for an application project.

1. angular.json: provides Workspace-level configuration for running builders, linters, i18n, and testers for the application. You can also define specific configurations that match to target environments.
2. root project folder: The application root folder contains configuration files to provide instructions for TypeScript compilation, Linting, Testing, and browser list targets.

Each Angular application has a conventional file structure.

> Learn about the application file structure and configuration files at:
> https://angular.io/guide/file-structure[4]

Here is a partial sample of a web application configured in an `angular.json` file. Using JSON allows for a schema definition - a reliable configuration definition.

> Configure at will, but stay within the lines of the schema.

[4]https://angular.io/guide/file-structure

```
  "projects": {
    "lms": {
      "projectType": "application",
      "schematics": {
        "@nrwl/workspace:component": {
          "style": "scss"
        }
      },
      "root": "apps/lms",
      "sourceRoot": "apps/lms/src",
      "prefix": "lms",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/apps/lms",
            "index": "apps/lms/src/index.html",
            "main": "apps/lms/src/main.ts",
            "polyfills": "apps/lms/src/polyfills.ts",
            "tsConfig": "apps/lms/tsconfig.app.json",
            "assets": ["apps/lms/src/favicon.ico", "apps/lms/src/assets"],
            "styles": [
              "apps/lms/src/styles.scss",
              "apps/lms/src/assets/demo/demo.css"
            ],
            "scripts": [
              "apps/lms/src/assets/js/core/popper.min.js",
              "apps/lms/src/assets/js/core/bootstrap.min.js",
              "apps/lms/src/assets/js/blk-design-system.min.js"
            ]
          },
          "configurations": {
            "production": {
              "fileReplacements": [
                {
                  "replace": "apps/lms/src/environments/environment.ts",
                  "with": "apps/lms/src/environments/environment.prod.ts"
                }
              ],
              "optimization": true,
              "outputHashing": "all",
              "sourceMap": false,
              "extractCss": true,
              "namedChunks": false,
              "aot": true,
              "extractLicenses": true,
              "vendorChunk": false,
              "buildOptimizer": true,
              "budgets": [
                {
                  "type": "initial",
                  "maximumWarning": "2mb",
                  "maximumError": "5mb"
                }
              ]
            }
          }
        },
        "serve": {...},
        "extract-i18n": {...},
        "lint": {...},
        "test": {...}
    }
  }
```

Application Configuration (angular.json)

Angular application Workspace configuration is managed in the `angular.json` file. The following application assets in the `architect|build|options` node of the project configuration.

- assets: use to set the application `*.ico` and the target `assets` folder
- styles: use to define styles for the application
- scripts: use to define any required JavaScript files used by other packages

Application Module (AppModule)

An Angular application is based on modules - the application is bootstrapped using a default `AppModule`. This module references other required modules along with a set of routes. In the sample application files listed below, there are (4) modules in addition to the AppModule.

- CoursesModule (feature)
- CoreModule (core)
- SharedModule (shared)
- SiteModule (site)

```
app
 □ features
 □ □ courses
 □ □ □ latest-courses
 □ □ □ □ latest-courses.component.css
 □ □ □ □ latest-courses.component.html
 □ □ □ □ latest-courses.component.spec.ts
 □ □ □ □ latest-courses.component.ts
 □ □ □ courses-routing.module.ts
 □ □ □ courses.module.ts
 □ modules
 □ □ core
 □ □ □ core.module.ts
 □ □ shared
 □ □ □ shared.module.ts
 □ site
 □ □ footer
 □ □ □ footer.component.css
 □ □ □ footer.component.html
 □ □ □ footer.component.spec.ts
 □ □ □ footer.component.ts
 □ □ navbar
 □ □ □ navbar.component.css
 □ □ □ navbar.component.html
 □ □ □ navbar.component.spec.ts
 □ □ □ navbar.component.ts
 □ □ sidebar
 □ □ □ sidebar.component.css
 □ □ □ sidebar.component.html
 □ □ □ sidebar.component.spec.ts
 □ □ □ sidebar.component.ts
 □ □ site.module.ts
 □ app-routing.module.ts
 □ app.component.html
 □ app.component.scss
 □ app.component.spec.ts
 □ app.component.ts
 □ app.module.ts
```

Application Module

Each Angular application has an `AppModule` responsible for bootstrapping the application. A module is a container to group related items together. It is an organizational structure. The concern for the application module is the entry-point for the application. It is responsible for:

- bootstrapping the application
- loading other modules
- providing global (application-level) services

```
import { BrowserModule } from "@angular/platform-browser"
import { NgModule } from "@angular/core"

import { AppComponent } from "./app.component"
import { SiteModule } from "./site/site.module"
import { AppRoutingModule } from "./app-routing.module"
import { SharedModule } from "./modules/shared/shared.module"

@NgModule({
  declarations: [AppComponent],
  imports: [AppRoutingModule, BrowserModule, SharedModule, SiteModule],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

> Learn more about Angular modules at: https://angular.io/guide/ngmodules

Shared Module

We can use the Angular CLI to generate additional modules in our applications. The concern of the `SharedModule` is to manage Angular and other 3rd-party modules for the application.

> Creating shared modules[5] allows you to organize and streamline your code. You can put commonly used directives, pipes, and components into one module and then import just that module wherever you need it in other parts of your app.

[5]https://angular.io/guide/sharing-ngmodules#sharing-modules

```
import { NgModule } from "@angular/core"
import { CommonModule } from "@angular/common"
import { RouterModule, Router } from "@angular/router"

@NgModule({
  declarations: [],
  imports: [CommonModule, RouterModule],
  exports: [RouterModule],
})
export class SharedModule {}
```

Core Module

The `CoreModule` is a module used to coordinate modules related to the domain of the application. There should only be a single instance of this module for the application. Use teh `parentModule: CoreModule` to check for another instance of the module in the constructor.

```
import { NgModule, Optional, SkipSelf } from "@angular/core"
import { CommonModule } from "@angular/common"
import { SiteModule } from "../../site/site.module"

@NgModule({
  declarations: [],
  imports: [CommonModule, SiteModule],
  exports: [SiteModule],
})
export class CoreModule {
  /**
   * Use the check to determine if the [CoreModule] has been loaded in the parentMod\
ule (AppModule root).
   */
  constructor(
    @Optional()
    @SkipSelf()
    parentModule: CoreModule
  ) {
    if (parentModule) {
      throw new Error(`CoreModule is already loaded. Import it in the AppModule only\
.`)
    }
  }
}
```

Site Module

In most applications there are elements that are not specific to a feature. They are site-level. Therefore, I create a SiteModule to contain things that are specific to the site:

- navigation
- sidebar
- header
- footer

```
import { NgModule } from "@angular/core"
import { CommonModule } from "@angular/common"
import { NavbarComponent } from "./navbar/navbar.component"
import { SidebarComponent } from "./sidebar/sidebar.component"
import { FooterComponent } from "./footer/footer.component"
import { SharedModule } from "../modules/shared/shared.module"

@NgModule({
  declarations: [NavbarComponent, SidebarComponent, FooterComponent],
  exports: [NavbarComponent, SidebarComponent, FooterComponent],
  imports: [CommonModule, SharedModule],
})
export class SiteModule {}
```

Feature Modules

Feature modules are containers for all related items of a specific feature. You can encapsulate the components, routes, services for business logic, and more within the feature module. Feature modules can be configured for lazy-loading to increase application performance.

Consider a feature module to be a mini-application within your Angular application. Use feature modules to organize your code - create as many as you need for each of the domain features of your application. The example below is the `CoursesModule`, which is responsible for displaying course information to the users of the application.

□courses
□ □latest-courses
□ □ □latest-courses.component.css
□ □ □latest-courses.component.html
□ □ □latest-courses.component.spec.ts
□ □ □latest-courses.component.ts
□ □courses-routing.module.ts
□ □courses.module.ts

```
import { NgModule } from "@angular/core"
import { CommonModule } from "@angular/common"

import { CoursesRoutingModule } from "./courses-routing.module"
import { LatestCoursesComponent } from "./latest-courses/latest-courses.component"

@NgModule({
  declarations: [LatestCoursesComponent],
  exports: [LatestCoursesComponent],
  imports: [CommonModule, CoursesRoutingModule],
})
export class CoursesModule {}
```

Routes

Components

Services

Models

Application Routes

Libraries

Most modern programming languages give you the ability to create reusable libraries. The question is how we do this with modern web applications? We need the ability to share and reuse code is if we're using compiled libraries. Modern web applications use JavaScript packages. These packages provide a library that can be used by installing the package in our application.

In order to consume these packages, JavaScript web applications need to consume these modules using a specific module loader. Depending on the type of application, different module loaders are used to initialize and use the specific JavaScript package. If you are an Angular developer and would like to consume custom libraries you need to develop separate JavaScript projects and compile them to specific package formats.

There are different package formats. The Angular team has created an Angular Package Format[6] guide that helps us to create packages using typescript and angular that can be shared and reused by other applications. The reason for this is because the angular CLI did not have a way to compile and publish libraries to a package manager such as npm. Therefore you would need to implement a script that would create a package that could be consumed as a module by different web applications. Later this process was automated by a new package call ng-packagr[7].

[6]https://docs.google.com/document/d/1CZC2rcpxffTDfRDs6p1cfbmKNLA6x5O-NtkJglDaBVs/preview#heading=h.k0mh3o8u5hx
[7]https://github.com/ng-packagr/ng-packagr

Now the current version of the angular CLI has the capability to build libraries using this the ng-packagr package. when you add this new capability to the new angular workspace, it now provides us with the capability of creating libraries and sharing and reusing those libraries among other angular applications as well as other angular libraries. These capabilities have been available since angular version 6. You can now create a library project in your workspace using a CLI command. There are other options to provide if you would like the library to be publishable to a package manager like npm. Without this new tooling, the creation and management of publishing packages and consuming them within an angular application was very time-consuming and not a very efficient process.

> Note: our current angular development environment with the workspace allows us to be very effective when it comes to sharing and reusing code AS libraries.

At a recent developer conference, I spoke to a member of a team that was using Python and React for their web development environment. However, they were also using Angular libraries with TypeScript to create reusable libraries for their React applications. This means that libraries are not just for angular applications. These Angular libraries packaged using ng-packagr may be consumed by any JavaScript application client-side or server-side.

Libraries are essential to effective architecture. They allow applications to be built using libraries for specific concerns throughout an application. We can share and reuse code in the different layers of our application architecture. Using these libraries allows us to keep our code simple without copying and pasting code in different parts of the application. We do not repeat ourselves with this type of code, but we reuse one single source of truth contained in a single library.

There are different types of Library projects that we can create to make our Angular architecture more effective. here is a list of different types of libraries that we can use to simplify our architecture:

- Cross-cutting concerns
- Custom Frameworks
- Feature Libraries
- Component Libraries
- Common

Cross-Cutting Concerns

Cross-cutting concerns are elements within an application that are not specifically associated with a domain feature of that application. The responsibility of a cross-cutting library is to provide specific functionality that crosses many or all of the domains within the application - thus the name cross-cutting concern. If we use the same process to identify features of our application we can also group and categorize cross-cutting concerns for our workspace. For example, most applications require logging and error handling. These are two cross-cutting concerns that can be implemented each as a library. Implementing these concerns as libraries, allow us to reuse and share this code with other applications and library projects.

There are several benefits to implementing cross-cutting concerns as libraries. If we do this, we do not have to have logging or error handling code scattered throughout our applications and library projects. We now have a single source of truth for each of these cross-cutting concerns. Therefore any new logging features or changes to the logging functionality can be done within the logging library project.

> Note: Creating cross-cutting concerns as libraries is really a code organization technique that simplifies our application workspace.

Custom Framework Libraries

We can also create custom framework libraries for our applications and Library projects. A framework is really a set of features and functionality that provide some utilities and/or features to the consumer of that framework. You may find that there is a specific utility or feature in the domain of your business that could be shared and used by many other application projects. Instead of copying this code from project-to-project, create a library project and implement the features and utilities in a single Library. This way any changes to that library are available to all of the consumers of that specific Library.

As a .NET/C# developer, I developed two frameworks to implement highly scalable business logic. Two of my first TypeScript projects were to create the same libraries as reusable TypeScript libraries for Angular applications. Let's now look at each of these as examples of custom framework libraries.

- Business actions
- Rule engine

Business Actions

My main focus on developing web applications is on the business layer. I have specialized in implementing very complex business logic that includes the evaluation of simple and composite business rules along with data validation. I have built applications with very strong business layers that are 100% unit testable. Business logic is one of the most important elements of your application. Therefore, it is essential that it is consistent, testable, maintainable and extensible.

Rule Engine

The Angularlicious Rules Engine is a JavaScript/Typescript based rule engine that allows applications to implement simple or sophisticated business rules as well as data validation. It contains a set of common rules ready for use; as well as a framework and set of classes for you to create any custom rule you need for your application.

- Provides a consistent way to implement business and validation rules and provide a consistent mechanism to retrieve the results.

- You can use the existing library of rules already implemented.
  - AreEqual
  - AreNotEqual
  - IsFalse
  - IsTrue
  - IsNullOrUndefined
  - IsNotNullOrUndefined
  - Max
  - Min
  - Range
  - StringIsNotNullEmptyRange
- You can create a reusable library of rules and use them in one or more applications.
- Combine default and one or more custom rules to create a `CompositeRule` - a rule that contains other rules (rule set).
- Each rule has a `Priority` property to execute rule sets in a specified sequence.
- Take advantage of Typescript classes to quickly create `simple` or `composite` (nested) rules using the API that is part of the framework.
- Use the `ValidationContext` to simply add, execute, and retrieve rule results.
- Code faster using Fluent API style syntax - be more productive.
- Using the `CompositeRule` base class, you can create a rule that contains other rules of either `simple` or `composite` types. The rule execution algorithm manages the complexity - now you can create rules and reuse rules to match your business logic.

Two core principles of good software design are Separation of Concerns (SoC)[8] and Single Responsibility[9]. Business rules and validation are an integral part of most business applications. There are rules and validations that must occur during processing of business logic. Most applications combine the business logic with rules and data validation - when this happens, testing and maintaining applications becomes more difficult.

A business rule engine allows the application to have a good Separation of Concerns (SOR). The Angular Rules Engine allows you to:

- Quickly start using out-of-the-box rules that are already implemented.
- Create custom rules that are either simple or composite.
- Create rules that can be reused throughout the application. Code reuse eliminates copy/paste of common rules.
- Use a single ValidationContext to add rules, execute rules, and evaluate the rule results.
- Use a consistent pattern and mechanism to implement your business rules and data validation.

[8]https://en.wikipedia.org/wiki/Separation_of_concerns
[9]https://en.wikipedia.org/wiki/Single_responsibility_principle

Feature Libraries

You can also create feature libraries for your application. A feature library is specific to a domain within your application. For example, you may have a security feature library, or a library that is concerned with account management, or shopping carts, or anything else related to an e-commerce application.

A feature library candidate, is a feature that many applications can share. For example, you may have a set of security feature that is shared by a suite of applications for your business. Another example, could be a shopping cart feature library that includes the components and services for all of your applications.

Component Libraries

You might find it useful to create a library for a specific set of components. This Library may have components that could be used by several applications. For example, you may have a notification component or a set of notification components that could be used by all of your applications. This lLibrary would include different kinds of components that are not specifically tied to a domain but could be used generically by different applications.

Here are some more examples.

- navigational or menus
- notifications
- popups
- cards
- tabs
- calendar
- form controls (inputs, dropdowns, checkboxes, radio, buttons)

This type of component library could use a 3rd-party package to abstract the implementation details of a specific control suite, such as:

- PrimeNg
- Material Design

Common

Another library project type could be for common elements. Common elements would include anything that is shared by most, if not all of your library or application projects. For example, you may have a specific model for your API responses. You will want all of your applications to conform to this schema provided by the API model. Therefore you can add an item like this to your common Library, and this Library could be made available to all other libraries. Typically a common Library has no or very few dependencies on other libraries. This is to avoid recursive references.

- models or classes commonly used throughout all layers of applications
  - API response
  - service messages with severity
  - error messages
- items shared by all project types

API Response Model

A good practice is to have a consistent API response for all calls to the back end of your application. A consistent response is well-known and becomes familiar to the developers that need to work with the responses. Typically, a response is mostly successful and contains a payload of data. Otherwise, it is failed and should contain information or messages related to the failure.

In our example below, we have an absctract class that defines a response using a generic type. This means that we can have a single definition - but it can be specialized by specifying the data type of the response using the `<T>` generic. All API responses would then have a:

- specific response type
- `IsSuccess` indicator
- `Message` from the API
- `StatusCode` representing the HTTP Response code
- `Timestamp` that indicates the datetime the response was created

Notice that the `ApiResponse<T>` is an abstract class. It cannot be instantiated, but must be extended by sub-classes. The abstract class provides the default definition of a response model.

```
export abstract class ApiResponse<T> {
  IsSuccess: boolean
  Message: string
  StatusCode: number
  Timestamp: Date
}
```

> Note: Research and determine what data elements and format your team could use as a reliable API response model. It is an interesting discussion.

A **success** response (`SuccessApiResponse<T>`) would extend this abstract base class and provide an implementation for the `Data` payload of type `<T>` as indicated.

```
import { ApiResponse } from "./api-response"

/**
 * Use to define a successful API response. A successful response will
 * most likely include a payload of data (i.e., use the Data property).
 */
export class SuccessApiResponse<T> extends ApiResponse<T> {
  Data: T
}
```

An **error** response (`ErrorApiResponse<T>`) would extend the same abstract base class and provide an implementation for a set of `ApiErrorMessage` items returned in the `Errors` property.

```
import { ApiResponse } from "./api-response"
import { ApiErrorMessage } from "./api-error-message"

/**
 * Use to provide error information from an API. You can also
 * use this class to create a response with errors while doing
 * error handling.
 *
 * Errors: is a list om [ApiErrorMessage] items that contain specific
 * errors for the specified request.
 */
export class ErrorApiResponse<T> extends ApiResponse<T> {
  Errors: ApiErrorMessage[] = []
}
```

Schematics

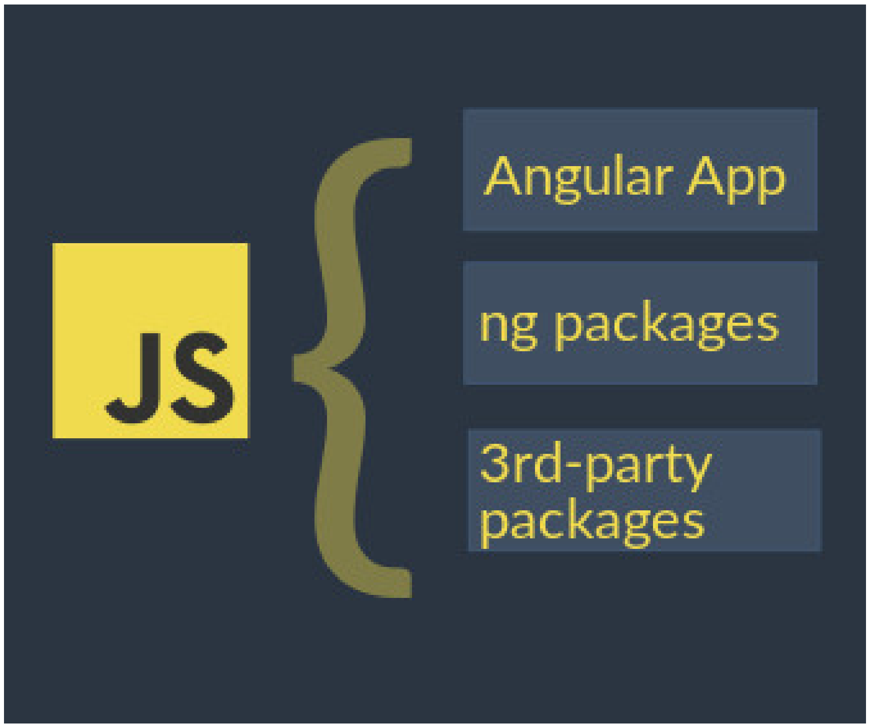
Component Services

Business Logic Services

Business Actions

Rules

Anatomy of an Angular Application



Architecture Layers

- Single Page Application
  - (1) index.html
  - JavaScript bundles
- Bundles
  - JavaScript
  - CSS/SCSS
  - Assets: images, .json, *.ico, etc.
  - 3rd-party JavaScript packages

Share and Reuse Code

- saves time and resources
- efficient development process

Why Share Code?

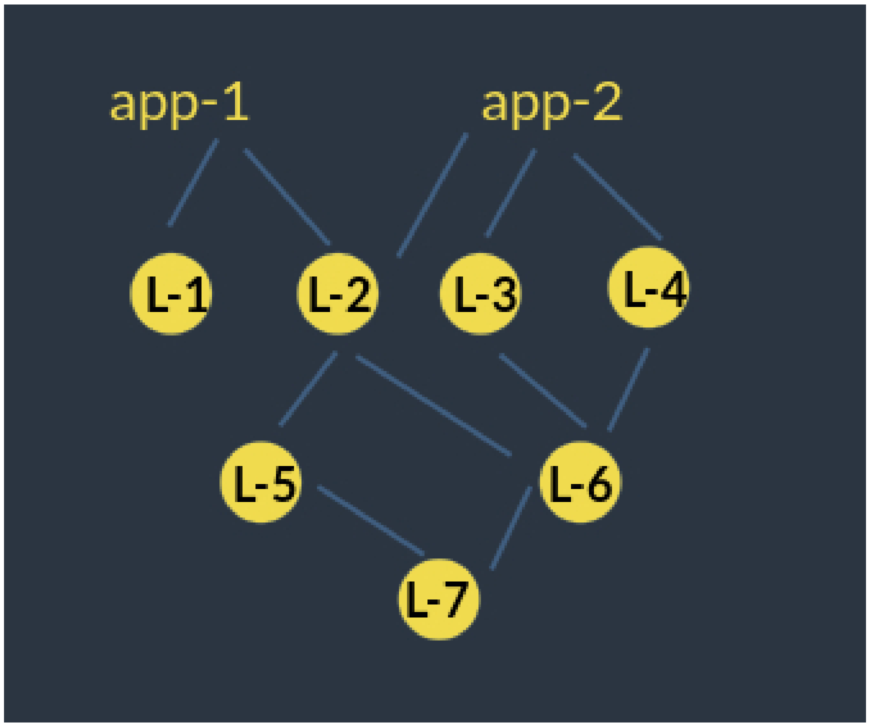
- single source-of-truth
- no copy/paste programming
- smaller code footprint
- reduces maintenance - and cost of ownership

How Can We Do It?

- create reusable components
- create reusable libraries/packages
- provide mechanism to share

JavaScript Packages

We need the ability to reuse and share JavaScript packages within our Angular applications and feature modules.



Architecture Layers

- Workflow
  - develop
  - test
  - build and version
  - publish
  - manage npm repository (private)
- Consume
  - manage versions
  - install

Not Effective?

> Using libraries to share and reuse code is the primary goal - not the maintenance and overhead of publishing libraries.

- Doesn't scale well with multiple libraries with dependencies.
- Inefficient workflow for enterprise development

Nx Dev Tools

- Library Projects - no build, no publish, no installation
- Supports Angular Schematic Projects
  - generate code using CLI
  - scaffold application elements
  - template-based code generation
- supports custom scope (i.e., @acme, @herff-jones)
- advanced library dependency management

Upgrade Existing Angular Workspace

Use the CLI command to upgrade an existing Workspace (Angular version 6 or greater) to use Nx Dev Tools (works when there is a single application).

```
ng add @nrwl/workspace
```

Getting Started - New Workspace

> Getting started: https://nx.dev/angular/getting-started/why-nx[10]

Use the Terminal window (Ctrl + J).

> Pro Tip: Get in the practice of using the `--dry-run` option. Allows you to run the command without making any changes to your files. This works on most CLI commands.

```
npx create-nx-workspace@latest --dry-run
```

[10]https://nx.dev/angular/getting-started/why-nx

```
npx create-nx-workspace@latest --dry-run
npx: installed 198 in 54.923s
? Workspace name (e.g., org name)     workspace
? What to create in the new workspace empty             [an empty workspace]
? CLI to power the Nx workspace       Angular CLI  [Extensible CLI for Angular appli\
cations. Recommended for Angular projects.]
Creating a sandbox with Nx...
warning @angular/cli > universal-analytics > request@2.88.2: request has been deprec\
ated, see https://github.com/request/request/issues/3142
[###################################################################################\
##############################] 357/358
```

Create a New Workspace

```
npx create-nx-workspace@latest
```

1. Workspace name: workspace
2. What to create in the new workspace empty
3. CLI to power the Nx workspace Angular CLI [Extensible CLI for Angular applications. Recommended for Angular projects.]

Output:

```
Creating a sandbox with Nx...
warning @angular/cli > universal-analytics > request@2.88.2: request has been deprec\
ated, see https://github.com/request/request/issues/3142
warning Your current version of Yarn is out of date. The latest version is "1.22.4",\
 while you're on "1.22.1".
new workspace "--dry-run" --preset="empty" --interactive=false --collection=@nrwl/wo\
rkspace
CREATE workspace/nx.json (462 bytes)
CREATE workspace/tsconfig.json (509 bytes)
CREATE workspace/package.json (1154 bytes)
CREATE workspace/README.md (2694 bytes)
CREATE workspace/.editorconfig (245 bytes)
CREATE workspace/.gitignore (503 bytes)
CREATE workspace/.prettierignore (74 bytes)
CREATE workspace/.prettierrc (26 bytes)
CREATE workspace/angular.json (96 bytes)
CREATE workspace/apps/.gitkeep (1 bytes)
CREATE workspace/libs/.gitkeep (0 bytes)
CREATE workspace/tools/tsconfig.tools.json (218 bytes)
CREATE workspace/tools/schematics/.gitkeep (0 bytes)
CREATE workspace/.vscode/extensions.json (164 bytes)

NOTE: The "dryRun" flag means no changes were made.

_______________________________________________



>  NX   NOTE  Nx CLI is not installed globally.

  This means that you might have to use "yarn nx" or "npm nx" to execute commands in\
 the workspace.
  Run "yarn global add @nrwl/cli" or "npm install -g @nrwl/cli" to be able to execut\
e command directly.
```

Update our new workspace to use the Nx CLI.

```
yarn global add @nrwl/cli
```

Verify the installation.

```
yarn global list
```

Output:

```
yarn global list
yarn global v1.22.1
info "@angular/cli@9.0.0" has binaries:
   - ng
info "@nrwl/cli@9.3.0" has binaries:
   - nx
info "create-nx-workspace@9.0.4" has binaries:
   - create-nx-workspace
info "http-server@0.12.1" has binaries:
   - http-server
   - hs
Done in 1.31s.
```

New workspace.

New Angular Workspace

Run the command below in the terminal to open the workspace folder in VSC.

```
cd .\workspace\
code .
```

```
git add .
git commit -am "add workspace"
git push
```

Code Organization

- Single-Responsibility
- Separation of Concerns
- Less Code
- Extensibility
- Easier Maintenance
- Testability

Architectural Patterns

- shared components
- shared libraries
- layered architecture –> CLEAN Architecture
- n-tier architecture
  - user interface (UI)
    - feature modules, components, UI Services
    - display and collect information only
    - no HTTP client or Web API code
    - pure UI logic
  - business logic (BL)
    - domain libraries
    - domain services
    - business logic
    - business rules processing
  - data access layer (DAL)
    - HTTP Client repositories
    - data access via Web API data operations

Library Project Types

There are several different types of library projects. Application and Library projects may consume other libraries.

Library	Type
configuration	cross-cutting
logging	cross-cutting
error-handling	cross-cutting
http-service	cross-cutting
foundation	common/shared
common	common/shared
security	common/shared
notifications	common/shared
business actions	framework
rule engine	framework
feature library	domain service
micro-app	domain

Benefits of a Monorepo Workspace

- developer effectiveness and efficiency
  - increased development velociy
- higher quality - easier to test (SR, SoC)
- longer shelf-life of application
- lower cost of maintenance

Resources

- Awesome Monorepo - Github[11]
- Code Reuse - Wikipedia[12]

[11]https://github.com/korfuri/awesome-monorepo
[12]https://en.wikipedia.org/wiki/Code_reuse#:~:text=Code%20reuse%20aims%20to%20save,the%20software%20product%20development%20process.&text=Code%20reuse%20may%20imply%20the,version%20of%20the%20reusable%20assets.