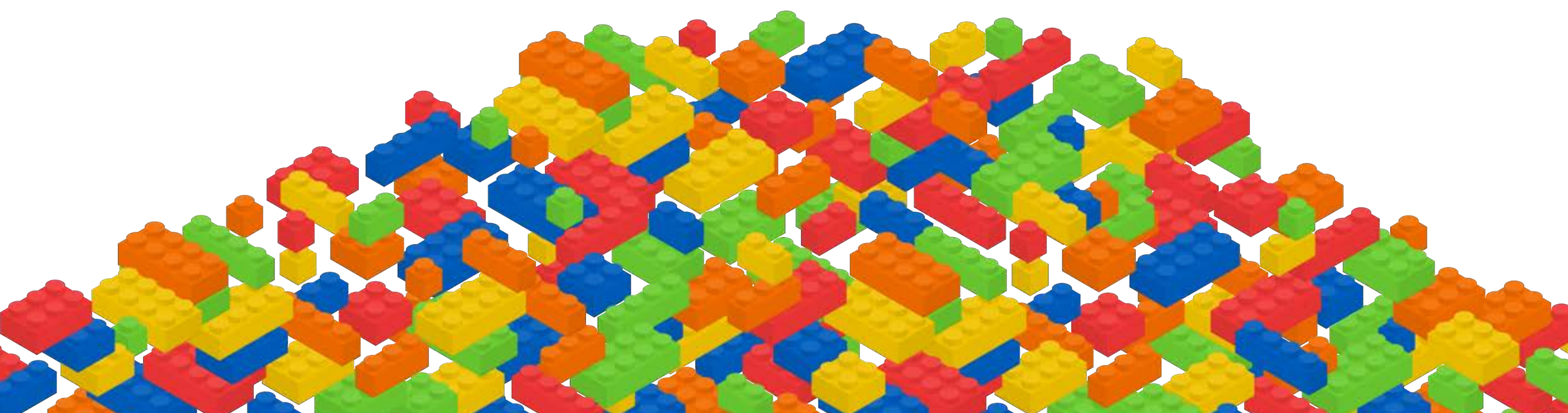
ENTERPRISE ANGULAR MONOREPO PATTERNS

Nitin Vericherla



Enterprise Angular Monorepo Patterns

Nrwl Technologies

Version v0.1, November 26, 2018

Table of Contents

Introduction

How this book is organized

In **Part 1** we begin by looking at development in a monorepo. We cover some basics about Nx and look at how to get started with Nx.

In **Part 2** we look at libraries in depth: how to organize them, name them, combine them and other techniques to aid in code reuse and modularization.

In **Part 3** we look at how to enforce quality and consistency across the monorepo with the tools built into Nx.

In **Part 4** we look at how we can help to make more intelligent builds and to use the Nx tools locally and in a CI pipeline.

In **Part 5** we look at some common development challenges when working in a monorepo with many teams with different release schedules.

In **Appendix A** we look at how to interact with Nx using other tools (Angular Console).

In **Appendix B** we look at all of the commands that you can use in Nx and refer to specific sections in the book where you can find more information on the command.

In **Appendix C** we look at some common how-tos and illustrate the decision trees for common questions.

Formatting

```
Code blocks are formatted like this.
```

> Asides are formatted like this and indicate information that might add some context to the topic being described.

ℹ	Informational call-outs highlight pertinent information that should be noted
🔥	Cautions indicate common gotchas
❗	Warnings indicate important information that can have a large effect

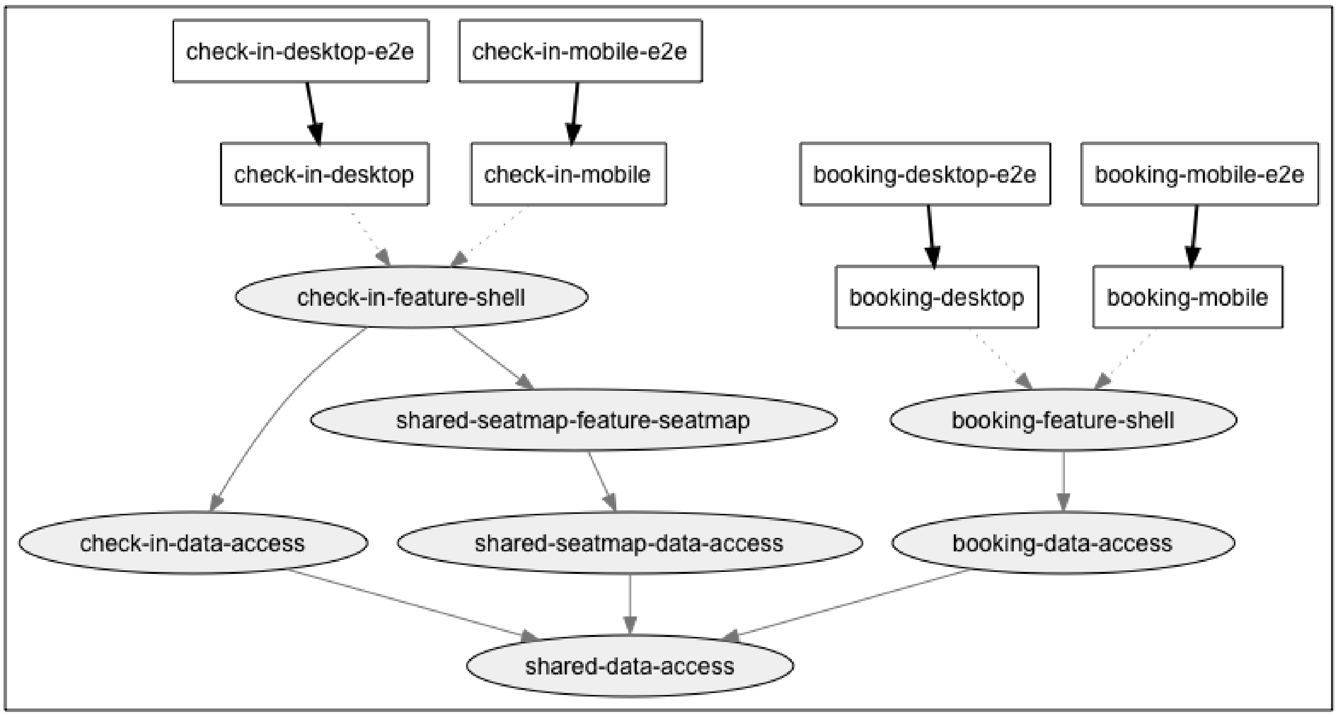
An example reference application

As a reference that we can use throughout the book, let's consider the fictional Nrwl Airways. There are three teams in this organization:

- **Booking:** The team works on allowing the user to book a flight to a destination.
- **Check-in:** The team works on allowing the user to use on-line check-in for a flight that they've booked.
- **Seatmap:** The team works on allowing the user to pick a seat on a flight graphically.

There are four (4) applications that are deployed separately: check-in (desktop and mobile) and booking (desktop and mobile). The end user is served one of these applications based on the URL they visit (`booking.nrwl-airlines.com` or `check-in.nrwl-airlines.com`) and the browser metadata that is sent with the request (to send them a desktop or mobile experience).

Its dependency graph of the code looks like this:



Part 1: Getting Started

Why a monorepo?

Many large organizations and business units within an organization are opting to develop all of their code (including numerous front-end applications as well as back-end applications) inside of a single repository. There are a few goals with this approach:

Increase visibility

Teams within an organization or business unit are often unaware of changes that other teams are making, and these have a large impact during integration. A lot of time can be saved if integration issues are discovered as soon as code is checked in. We discuss some strategies to deal with the changes in Part 5 of the book.

Furthermore, API contracts are easily accessible in a monorepo and can be used directly by both the front-end and the back-end. Types can be generated from the contracts and consumed by both the front-end and back-end to ensure that errors are caught at compile-time rather than the more expensive integration-time.

Reuse code directly

The traditional way to modularize and share code is to create a package, deploy it to a private npm repository, and to depend on it in by adding it to the project dependencies. There is a large amount of overhead when making changes to these because of the time it takes to package and deploy the dependency code. There is also an issue with versioning because we have to refer to the right version number.

Developers might alternatively use `npm link` or ways to simulate the dependency management for local development, but this is also cumbersome to set up and use: there might be a lot of dependencies that we need to set up in this way and we have to remember to remove the links when we are done.

Working in a monorepo allows you to refer to the dependency directly (using workspace-relative paths in the case of Nx). The code is also available to the developer to work on directly, and we discuss ways to integrate changes to shared code in Part 5 of the book.

Ensure a consistent version of dependencies

Version control in a monorepo becomes much easier - organizations or business units can choose to have a single version of dependencies across all projects or to have a "latest-minus-X" policy to ensure that all projects are kept up-to-date. This reduces the likelihood that deprecated dependencies and vulnerable versions are relied upon in their code.

All of these goals are achievable when using Nx.

Why Nx?

Large organizations encounter some issues that one might not find in smaller teams:

- While ten (10) developers can reach a consensus on best practices by chatting over lunch, five hundred (500) developers cannot. You have to establish best practices, team standards, and use tools to promote them.
- With three (3) projects developers know what needs to be retested after making a change. With thirty (30) projects, however, this is no longer a simple process. Informal team rules to manage change no longer work with large teams and multi-team, multi-project efforts. You have to rely on the automated CI process instead.

In other words, small organizations can often get by with informal ad-hoc processes, whereas large organizations cannot. Large organizations must rely on tooling to enable that, and Nx provides this tooling. It includes the following:

- It contains schematics to help with code generation in the projects to ensure consistency.
- It contains tools to help with enforcing linting rules and code formatting.
- It allows you to visualize dependencies between apps and libraries within the repository
- It allows commands to be executed against code changes: any uncommitted changes, comparison between two specific git commits, comparison against another branch, etc. These commands can test, lint, etc. only the `affected` files, saving us a lot of time by avoiding files that were not affected by our changes.
- It includes utilities to help with race conditions in Angular applications

Let's look at some of the basic building blocks of Nx: workspaces, apps and libs.

Nx Basics

What is a workspace?

A workspace is a folder created using Nx. The folder consists of a single git repository, with folders for `apps` (applications) and `libs` (libraries); along with some scaffolding to help with building, linting, and testing.

What is an app?

An app produces a binary. It contains the minimal amount of code required to package many libs to create an artifact that is deployed.

The app defines how to build the artifacts that are shipped to the user. If we have two separate targets (say desktop and mobile), we might have two separate apps.

In our reference example the four (4) applications are organized as below:

```
apps/
  booking/
    booking-desktop/  <--- application
    booking-mobile/   <--- application
  check-in/
    check-in-desktop/ <--- application
    check-in-mobile/  <--- application
```

Each has a corresponding e2e testing app folder generated as well.

Apps are meant only to organize other libs into a deployable artifact - there is not a lot of code present in the applications outside of the module file and maybe a some basic routing. All of the application's code is organized into libs.

What is a lib?

A lib is a set of files packaged together that is consumed by apps. Libs are similar to node modules or nuget packages. They can be published to NPM or bundled with a deployed application as-is.

The purpose of having libs is to partition your code into smaller units that are easier to maintain and promote code reuse. With Angular CLI 6.x, libraries can be used within applications in the `/apps` folder or even be bundled and deployed to NPM as a stand-alone package.

Some libraries are used only by a particular app (e.g. booking) and should go into the appropriate directory (e.g., `libs/booking`). We call them "app-specific". Even though such libraries can be used in more than one place, the goal of creating them is not code reuse, but factoring the application into well-defined modules to simplify the application's maintenance.

A typical Nx workspace contains only four (4) types of libs: *feature*, *data-access*, *ui*, and *util*. You can read about these types of libraries in detail in **Part 2** of the book.

Interacting with Nx

Nx is configured for a workspace with these configuration files:

- `nx.json` which is specific to Nx
- `package.json` which is provided by npm
- `angular.json` which is provided by the Angular CLI

Nx also provides commands to work with your workspace, apps and libs. These can be entered into a terminal or called from within the Angular Console, which is a graphical tool to interact with the Angular CLI and Nx. The Angular Console is discussed in Appendix A.

In this book we provide the terminal commands using `npm` in each section as they are covered. We provide instructions for the Angular Console where it is appropriate. All the commands are also listed in Appendix B of the book.

`yarn` vs. `npm`

In this book we provide example code using `npm`. All of these commands also work with `yarn`. There are some differences between the two:

1. npm commands can accept parameters for the command itself and for the underlying command. For example, `npm run task1 --watch — --param1=val1` passes the parameter `watch` to npm itself and `param1` to the underlying implementation of `task1`. The `--` by itself indicates the break after which all parameters are forwarded to the underlying task. `yarn` on the other hand does not do this and passes all parameters to the underlying task; hence it doesn't need the `--` separator.
2. `npm` commands are run with `npm run <command>` and `yarn` commands are run with `yarn <command>`.
3. `npm install` installs the packages and you need to specify `--save` or `--save-dev` to save the dependency; whereas `yarn` installs packages with `yarn add` and adds them to package.json by default.

In the book, you can convert an npm script to work with yarn by using the following method:

1. If the command starts with `npm run` you can replace `npm run` with `yarn`. Otherwise, if the command is `npm install` you can use `yarn add` instead (if it is `npm install -g` or `npm install --global`, use `yarn add global`). For most of the other occurrences you should be able to swap `npm` with `yarn` directly.
2. If the command contains a `--` by itself, remove this for `yarn`.

Installing and Setting Up a Workspace

Creating a workspace

You can install Nx in the following way:

```
npm install -g @angular/cli @nrwl/schematics
```

The `@nrwl/schematics` library contains a binary script that provides the command `create-nx-workspace`. That command can be used to create a new Nx Workspace:

```
 create-nx-workspace myworkspacename
```

This creates a new Nx Workspace using a sandboxed environment and running the Angular CLI's `ng new` command under the hood with the Nx schematics collection.

You can also add Nx capabilities to an existing CLI project by running:

```
 ng add @nrwl/schematics
```

Finally, you can create an Nx Workspace using Angular Console.

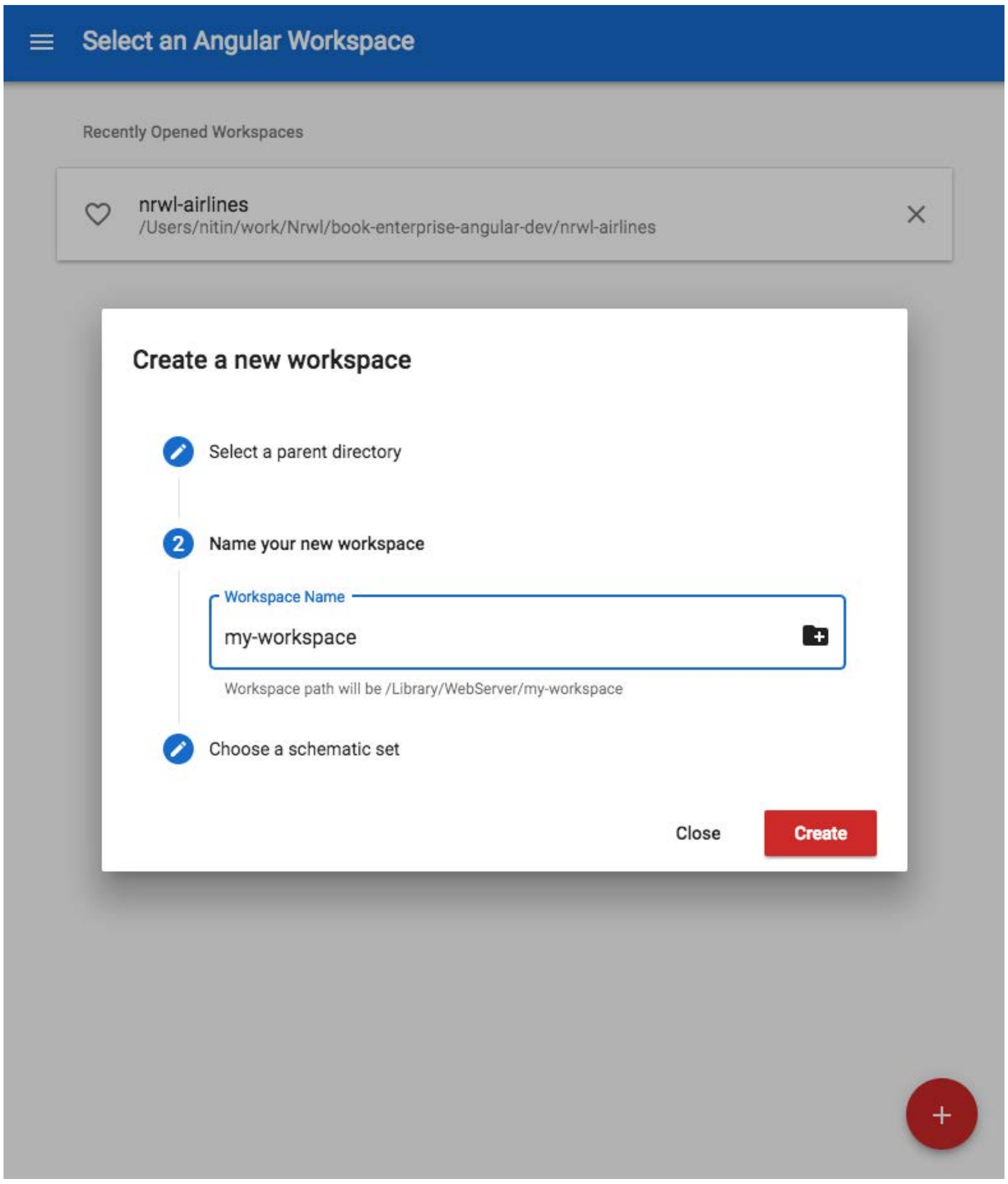

Select an Angular Workspace
Recently Opened Workspaces
nrwl-airlines
/Users/nitin/work/Nrwl/book-enterprise-angular-dev/nrwl-airlines
Create a new workspace
Select a parent directory
2 Name your new workspace
Workspace Name
my-workspace
Workspace path will be /Library/WebServer/my-workspace
Choose a schematic set
Close
Create

Figure 1. Creating a new workspace in Angular Console

Regardless of how you create a workspace, what you end up with is a **Nx Workspace** with the following files/folders created.

```
apps/
libs/
tools/
angular.json
nx.json
tslint.json
tsconfig.json
```

It's similar to the standard Angular CLI projects with a few changes:

- It has an `apps` dir where all applications are placed
- It has a `libs` dir where all custom library code is placed

A short description of each of them is below; we expand on each of them as we progress through the book.

- **apps/:** This is where all the apps and e2e folders reside
- **libs/:** This is where libraries are placed
- **tools/:** This is used for tooling e.g. workspace schematics
- **angular.json:** This is used by the Angular CLI to describe the projects (apps and libs), how to build and test them, etc.
- **nx.json:** This is used by Nx to provide metadata for projects e.g. tags
- **tslint.json:** This is the linter configuration file
- **tsconfig.json:** This is the workspace tsconfig. Nx adds path aliases for each lib here to allow for workspace-relative imports e.g.

  ```
  import { myLib } from '@myProject/shared/my-lib';
  ```

A new Nx Workspace does not set up an initial application, but adding one is simple.

Creating an app

Adding new apps to an Nx Workspace is done by using the Angular CLI generate command. Nx has a schematic named `app` that can be used to add a new app to our workspace:

```
ng generate app myapp
ng generate application myapp # same thing
```

This command executes a few actions:

1. It creates a new app, places it in the `apps` directory, and configures the `angular.json` and `nx.json` files to support the new app.
2. It configures the root NgModule to import the `NxModule` code so we can take advantage of things like `DataPersistence`.
3. It also provides an e2e sibling folder for this app that contains the e2e testing code.

Run `ng generate app --help` to see the list of available options.

Most of the options are identical to the ones supported by the default CLI application, but the following are new or different: `directory`, `routing`, and `tags`.

- `ng generate app myapp --directory=myteam` creates a new application in `apps/myteam/myapp`.
- `ng generate app myapp --routing` configures the root `NgModule` to wire up routing, as well as add a `<router-outlet>` to the `AppComponent` template to help get us started.
- `ng generate app myapp --tags=scope:shared,type:app` annotates the created app with the two tags, which can be used for advanced code analysis. Read more about this in the section "Constraints on libraries" in **Part 3.**

Once we’ve created an application, we can start creating libs that contain all of the components and logic that make up the application.

Creating a lib

Adding new libs to an Nx Workspace is done by using the Angular CLI generate command, just like adding a new app.

```
ng generate lib mylib
ng generate library mylib # same thing
```

This creates a new lib, places it in the `libs` directory, and configures the `angular.json` and `nx.json` files to support the new lib.

Refer to the section "Notes on using libraries" for further description of the options.

Getting help

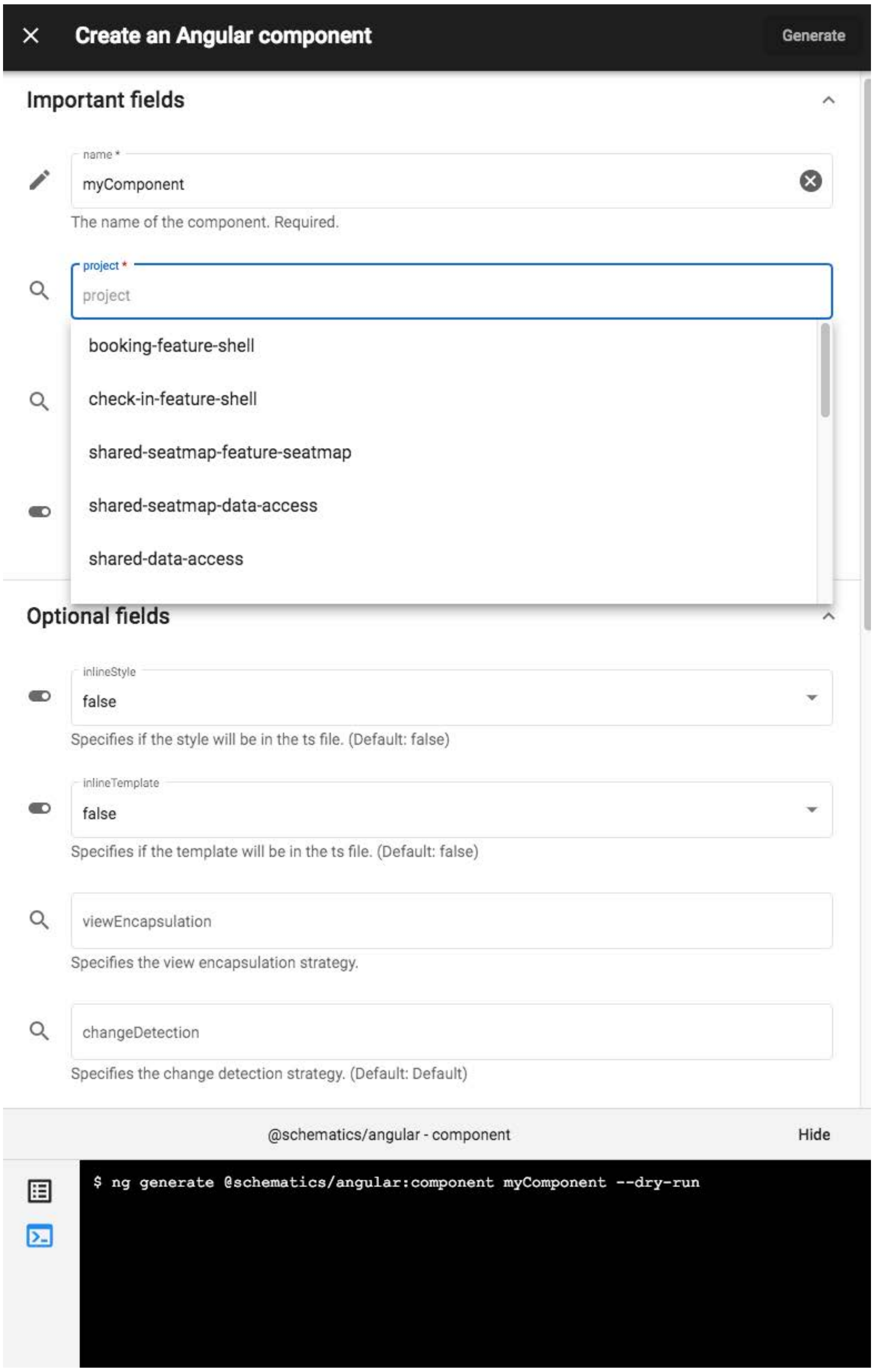
When using the terminal, all Nx commands offer the `--help` flag to display the available options and descriptions for each. The Angular Console displays help text visually and also provides the list of options grouped by whether they are required or not.

A sample output is below:

Run `ng generate lib --help` to see the list of available options:

```
  usage: ng generate lib <name> [options]
  options:
    --directory
      A directory where the app is placed
    --dryRun (-d)
      Run through without making any changes.
    --force (-f)
      Forces overwriting of files.
    --lazy
      Add RouterModule.forChild when set to true, and a simple array of routes when
set to false.
    --parent-module
      Update the router configuration of the parent module using loadChildren or
children, depending on what `lazy` is set to.
    --prefix (-p)
      The prefix to apply to generated selectors.
    --publishable
      Generate a simple TS library when set to true.
    --routing
      Add router configuration. See lazy for more information.
    --skip-format
      Skip formatting files
    --skip-package-json
      Do not add dependencies to package.json.
    --skip-ts-config
      Do not update tsconfig.json for development experience.
    --tags
      Add tags to the library (used for linting)
    --unit-test-runner
      Test runner to use for unit tests
```

Getting help is also possible using the Angular Console, which provides all the options for each command along with descriptions. It also separates the required options from the optional ones and auto-fills the values based on your workspace (allowing you to choose from a list instead of manually typing).



name *
myComponent
The name of the component. Required.

project *
project

booking-feature-shell
check-in-feature-shell
shared-seatmap-feature-seatmap
shared-seatmap-data-access
shared-data-access

inlineStyle
false
Specifies if the style will be in the ts file. (Default: false)

inlineTemplate
false
Specifies if the template will be in the ts file. (Default: false)

viewEncapsulation
Specifies the view encapsulation strategy.

changeDetection
Specifies the change detection strategy. (Default: Default)

@schematics/angular - component

```
$ ng generate @schematics/angular:component myComponent --dry-run
```

Figure 2. Getting help with Angular Console. Note that the options are separated into "important" and
"optional", and that the values are able to be selected in a drop down.

Summary

This section sets the stage for the rest of the book. It covered key concepts to lay the groundwork for the following:

- Some reasons for adopting a monorepo mindset were covered: increase visibility for all teams into the codebase, reuse code directly instead of publishing and consuming via package management, and ensure a consistent version of dependencies across all projects.

- The basic terminology and foundational concepts of Nx were presented (a workspace, application and library)
- A description of the command line and how to interact with Nx
- Getting set up with Nx and creating workspaces, apps and libs
- How to get help with Nx commands

Part 2: Organizing code with libraries

In a nutshell

Applications grow in size and complexity and in a large organization there are many ways to reuse code to maintain consistency and to lower the development effort. Developing in a modular way can be done using libraries, which are simply a collection of related files that perform a certain task. These are composed together to make up an application.

Libraries require classifiers to describe their contents and intended purpose. These classifiers help to organize the libraries and to provide a way to locate them.

Scope

Scope relates to a logical grouping, business use-case, or domain. Examples of scope from our sample application are `seatmap`, `booking`, `shared`, and `check-in`. They contain libraries that manage a sub-domain of application logic.

> We recommend using **folder structure to denote scope.**
>
> The following folder structure is an example scope hierarchy used to describe the seatmap feature:
>
> ```
> shared/
>     seatmap/
>         feature/
> ```
>
> Here, "shared" and "seatmap" are grouping folders, and `feature` is a library that is nested two levels deep. This offers a clear indication that this feature belongs to a domain of `seatmap` which is a sub-domain of `shared` items.
>
> The tag used in this library would be `scope:shared`, as this is the top-level scope.

Type

Type relates to the contents of the library and indicates its purpose and usage. Examples of types are `ui`, `data-access`, and `feature`. See the longer discussion below.

> We recommend using **prefixes and tags to denote type.** We recommend limiting the number of types to only the four described in the sections to follow.
>
> The folder name for this feature would be `feature-shell` so that it uses the prefix for its library type.
>
> The tag for the seatmap feature library as in the previous example would now be `scope:shared,type:feature`.

Platform

There can be other classifiers used to differentiate between similar libraries (e.g. between `server`, mobile, and `desktop`). **Platform** is one such classifier.

> We recommend using **tags to denote platform.**
>
> The final tag for the seatmap feature would be `scope:shared,type:feature,platform:desktop`.

Every library should be located in the folder tree by scope, have tags that are in the format `scope:SCOPE,type:TYPE,platform:PLATFORM`, and have a prefix by its type.

> *Don't Organize by file type!*
>
> We strongly discourage organization by file type e.g. `directives/`, `services/`, etc. The reason for this is that when we are looking to make a change, we often need to change a few related files at once. When organizing by file type we need to traverse the folder tree multiple times to locate the needed files.
>
> Rather we suggest that the code base be organized by **domain** and include all the related files together e.g. `airline` which includes state, ui components, etc. inside a single grouping folder.

Types of libraries

There are many different types of libraries in a workspace. In order to maintain a certain sense of order, we recommend having only the below four (4) types of libraries:

- **Feature libraries:** Developers should consider **feature** libraries as libraries that implement smart UI (with injected services) for **Business Use Cases** specific to that **feature.**
- **UI libraries:** A UI library contains only `presentational` components.
- **Data-access libraries:** A data-access library contains services and utilities for interacting with a back-end system.
- **Utility libraries:** A utility library contains **common utilities and services** used by many libraries and applications. It also includes all the code related to State management.

Feature libraries

Feature libraries contain router configurations for a particular application section.

Most of the UI components in such a library are smart components that interact with the NgRx Store. This type of library also contains most of the application logic, validation etc. **Feature libraries** are almost always app-specific and are often lazy-loaded.

> Developers should consider feature libraries as libraries that implement smart UI (with injected services) for Business Use Cases specific to that feature.

Naming Convention: feature or feature-* (e.g., feature-shell).

```
libs/
  booking/
    feature-shell/
      src/
        index.ts
        lib/
          booking-feature-shell.module.ts
```

> Example - Feature Library Module:

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
import { CommonModule } from '@angular/common';

import { DashboardComponent } from './components/dashboard/dashboard.component';

@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([
      {
        path: '',
        component: DashboardComponent,
        pathMatch: 'full'
      }
    ])
  ],
  declarations: [DashboardComponent]

})
export class BookingFeatureShellModule {}
```

See the **Notes on using Libraries** for discussion over command line options available for lazy-loading the feature libraries.

UI libraries

A UI component library contains only **presentational** components.

- An app-specific component library contains the components specific to a particular application (or a few very closely connected applications).
- A shared component library contains components used in multiple applications or across many business functions.

Naming Convention: ui or ui-* (e.g., components-buttons).

There are two types of components when we build applications:

- **Smart Components**
  - manage or delegate business logic and use DI to inject services.
  - have child instances of presentational components.
- **Presentational Components** (aka **dumb**)
  - no or very little business logic
  - **only** rely on Inputs and Outputs to communicate with the outside world.
  - have Smart component parents
  - announce user interactions using outputs
  - are highly reusable and are the easiest to test
  - render data in a presentational format (e.g., display a user ticket), and
  - may be fully generic/domain-agnostic (e.g,. data-table) or have a domain-context (e.g, log-book-table).

Consider the `OrderCreate` Smart component below:

> **order-create.component.ts**

```
@Component({
  selector: 'order-creator',
  styleUrls: ['./order-create.component.css'],
  template: `
      <div fxLayout fxLayoutAlign="center center">
        <order-card  [order]="order"
                      [users]="users$ | async" >
        </order-card>
      </div>
    `
})
export class OrderCreatorComponent {
  users$: Observable<User[]> = this.service.users$;
  order: Order = makeNewOrder();

  constructor(public service: OrdersFacade, public users: UsersFacade) {}
}
```

Here is its child presentational component `OrderCard`:

> order-card.component.ts

```
@Component({
  selector: 'order-card',
  styleUrls: ['./order-card.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
      <mat-card class="order-card" *ngIf="order">
        ...
      </mat-card>
    `
})
export class OrderCardComponent {
  @Input() users: User[];
  @Input() order: Order;

  @Output() save     = new EventEmitter<Order>();
  @Output() cancel   = new EventEmitter<void>();
  @Output() reassign = new EventEmitter<string>();
  @Output() complete = new EventEmitter<Order>();

  title(order): string {
    const status = order.completed ? 'Finished' : 'Pending';
    return !order.id ? 'Create New Order' : `${status} Order `;
  }
}
```

Example - Component (Presentational) Library Module:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { LoadingComponent } from './loading/loading.component';
import { ConfirmButtonComponent } from './confirm-button/confirm-button.component';

@NgModule({
  imports: [CommonModule],
  declarations: [LoadingComponent, ConfirmButtonComponent],
  exports: [LoadingComponent, ConfirmButtonComponent]
})
export class ComponentButtonsModule {}
```

The path to this module would be libs/common/ui-buttons/src/common-ui-buttons.module.ts

Data-access libraries

Data-access libraries contain REST or webSocket services that function as client-side delegate layers to server tier APIs. All files related to State management also reside in a data-access folder.

Name Conventions: data-access or data-access-*.

> Example Data-Access Library Module:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { StoreModule } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';
import { customersReducer } from './+state/state.reducer';
import { customersInitialState } from './+state/state.init';
import { CustomersEffects } from './+state/state.effects';

@NgModule({
  imports: [
    CommonModule,
    StoreModule.forFeature('customer', customersReducer, {
      initialState: customersInitialState
    }),
    EffectsModule.forFeature([CustomersEffects])
  ],
  providers: [CustomersEffects]
})
export class CustomersDataAccessModule {}
```

It's easy to **reuse data-access** libraries, so focus on creating more shared data-access libraries.

If a library is shared, document it!

Often building a shared data-access library requires you to get data from multiple endpoints. In this case build an api where you specify exactly what fields you depend on, to indicate which endpoints to hit (see the customers data access library).

When using NgRx facades, developers should consider three (3) levels of APIs:

- API for your HTTP REST services: these are simple, public methods invoked by views or facades. These methods hide the complexity of using an internal HTTPClient service.
- API for your NgRx facades: api that is used only at the view component levels to—publish query observables as properties: observables to NgRx store selectors—public methods that internally dispatch actions to the NgRx store.
- API for your NgRx Effects: define which effects 'run' for which NgRx action. These effects usually run asynchronous processes.

Utility libraries

A utility contains common utilities/services used by many libraries.

Naming Conventions: `util` (if nested), or `util-*` (e.g., util-testing)

> Example - Utility Library Index File (aka Barrel):

```
export { navigateToPage } from './src/navigate-to-page';
export { HttpMockBackend } from './src/http-mock-backend';
```

Grouping folders

In our reference structure, the folders `libs/booking`, `libs/check-in`, `libs/shared`, and `libs/shared/seatmap` are grouping folders. They do not contain anything except other library or grouping folders.

```
apps/
  booking/
  check-in/
libs/
  booking/                <---- grouping folder
    feature-shell/        <---- library

  check-in/
    feature-shell/

  shared/                 <---- grouping folder
    data-access/          <---- library

    seatmap/              <---- grouping folder
      data-access/        <---- library
      feature-seatmap/    <---- library
```

Shared libraries

In our reference example there are some shared libraries

```
libs/
  shared/
    data-access/          <---- shared library

    seatmap/
      data-access/        <---- shared library
      feature-seatmap/    <---- shared library
```

Domain agnostic: Libraries that are used across most applications. These go into grouping folders that describe the logical shared domain or into shared. (e.g., `shared/data-access`)

Domain-specific: Libraries that are only used in a few applications. These go into grouping folders. (e.g., `booking`)

Sometimes, it is not easy to see if a library should be shared or not. See the decision trees in the Reference section for some guidance.

Notes on using libraries

Command line options

When using the Nx schematics to create libraries, developers have several **power** options. Most of these options are identical to the ones supported by the default CLI library, but the following are new or different.

The `--lazy` flag is especially important to create lazy-loaded libraries.

Options when using `ng generate lib mylib`

Flags	Result
`--directory=myteam`	Create a new library with path `libs/myteam/mylib`
`--routing`	Configure the lib's NgModule to wire up routing to be loaded eagerly.
`--parent -module=apps/myapp/src/app/app.module.ts`	Configure the routes of `AppModule` at the given path to include a route to load this library. If used in conjunction with `--lazy`, the route is loaded using `loadChildren` to lazy-load the route.
`--publishable`	Generate a few extra files configuring for ng-packagr. You can then `ng build mylib` to create an npm package you can publish to a npm registry. This is very rarely needed when developing in a monorepo. In this case the clients of the library are in the same repository, so no packaging and publishing step is required.
`--tags=scope:shared,type:feature`	Create an entry in `nx.json` to associate the created lib with the two tags, which can be used for advanced code analysis.

When creating lazy-loaded libraries, you need to add an entry to the **`tsconfig.app.json`** file of the parent module app, so TypeScript knows to build it as well:

```
{
  "include": [
      "**/*.ts"
      /* add all lazy-loaded libraries here: "../../../libs/my-lib/index.ts" */

      , "../../../libs/mymodule/src/index.ts"
  ]
}
```

In most cases, Nx does it by default. Sometimes, you need to add this entry manually.

The barrel file

When we use Nx and @nrwl/schematics to create a library, there are a few steps that are executed.

Firstly the folder is created (in a nested structure if `--directory` was specified). Next, the basic files are generated for the library in this structure:

```
jest.config.js
src/
  index.ts
  lib/
    shared-data-access.module.spec.ts
    shared-data-access.module.ts
    shared-data-access.service.spec.ts
    shared-data-access.service.ts
  test-setup.ts
tsconfig.lib.json
tsconfig.spec.json
tslint.json
```

NB: `jest` was selected as the testing framework for this lib.

Note the `index.ts` file is in the `src/` folder. This is the **barrel file** for the lib. It contains the public API to interact with the library and we should ensure that any constants, enums, classes, functions, etc. that we want to expose are exported in this barrel file.

If it is nested inside `src/` folder though, how does the barrel get exposed when we call the library as follows: `import { SharedDataAccessModule } from '@nrwl-airlines/shared/data-access';`? The import would expect the index.ts file to be in the root folder of the lib and not inside `src/`.

Nx configures this in the root tsconfig.json, which would contain the following entry:

```
"paths": {
  ...
  "@nrwl-airlines/shared/data-access": [
    "libs/shared/data-access/src/index.ts"
  ]
  ...
}
```

Each app and lib that we create gets an entry in this file to help with the mapping using the `@` workspace-relative path syntax. You can create your own aliases in this way if it helps with development.

Import Paths

Components and classes **contained within** a library should be imported with relative paths only. Referring to them with the workspace-relative path leads to linting errors.

Components and services **imported from outside** the current library must use **npmScoped** imports (eg. `@myOrg/customers/`) instead of relative paths.

TsLint rules have been configured to display errors for such violations.

Module names

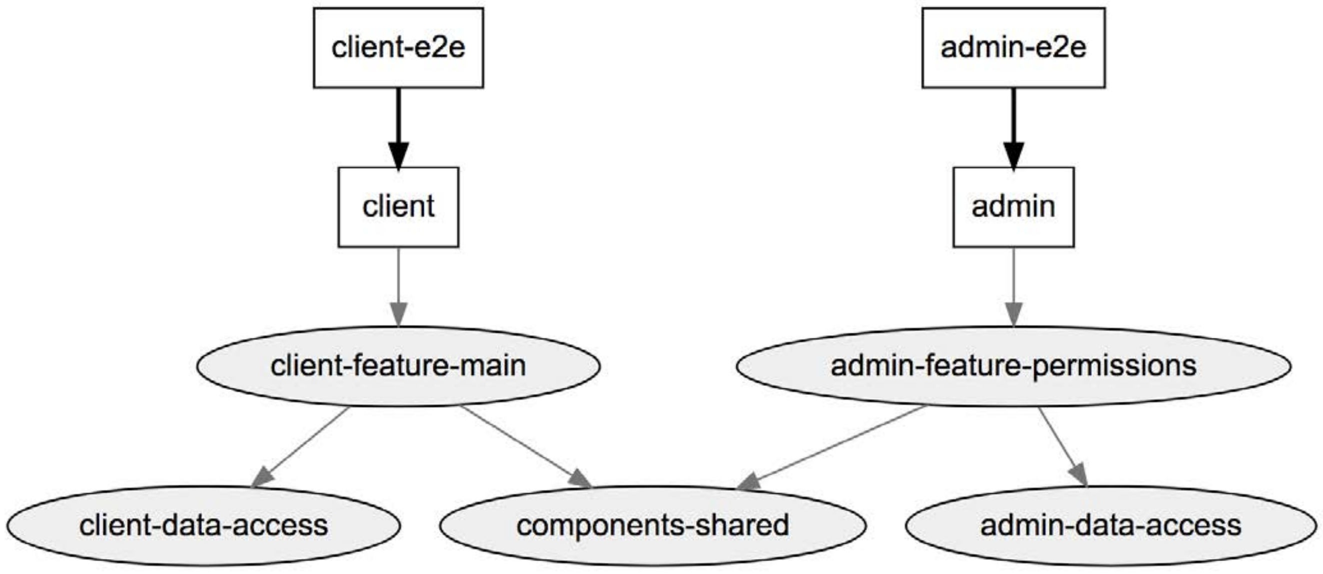
The main module for a library must contain the full path (relative to `libs`) within the filename. e.g. the main module for the library `libs/booking/feature-destination` would have the module filename as `booking-feature-destination.module.ts`.

This is the way that the CLI creates the modules and we are recommending that we keep this pattern.

Documenting libraries

The README should identify the purpose of the library and outline the public API for the library. The document may also include other details such as:

- code owner
- visualization of the libraries usages (dependency-graph)
- visualization of the dependency-constraints (which apps or libraries are authorized to USE this library).



Summary

In this section we looked at libraries in depth. We discussed the following:

- Why it is not a scalable strategy to group by file type
- Using **scope, type, platform** and other classifiers to organize libraries
- The four types of libraries: `ui`, `feature`, `data-access`, and `util`
- Using Grouping Folders to introduce hierarchy for **scope**
- Shared vs. domain-specific scopes

Part 3: Enforcing quality and consistency

Google enforces a constraint that there is a single version of Angular across all projects. This ensures that there are no legacy projects that make it difficult to refactor, and enables developers to move between projects as well.

Nx contains a few tools to help with maintaining consistency across the code-base and to ensure the code quality:

- A single package.json ensures that all apps and libs use the same dependency versions.
- Build tools like `prettier` assist with ensuring consistency in code formatting (one immediate benefit is the removal of whitespace and formatting diffs in PRs).
- It allows for configurable enforcement of boundaries between libraries e.g. a `util` library not being able to depend on a `feature` library (or even an app).
- Nx allows you to generate workspace schematics to handle code generation consistently.
- Nx includes a DataPersistence library to help with managing interactions with the back-end in a consistent and reliable way.

Code Formatting

At Google the general tenet is that anything that can be automated should be automated. One of those things is code formatting. That's why Nx comes with built-in support for Prettier.

There is absolutely nothing you need to configure. Just call `format:write` to format the affected files, and `format:check` in the CI to guarantee that everything is formatted consistently.

Run `npm run format:write — --help` to see the available options.

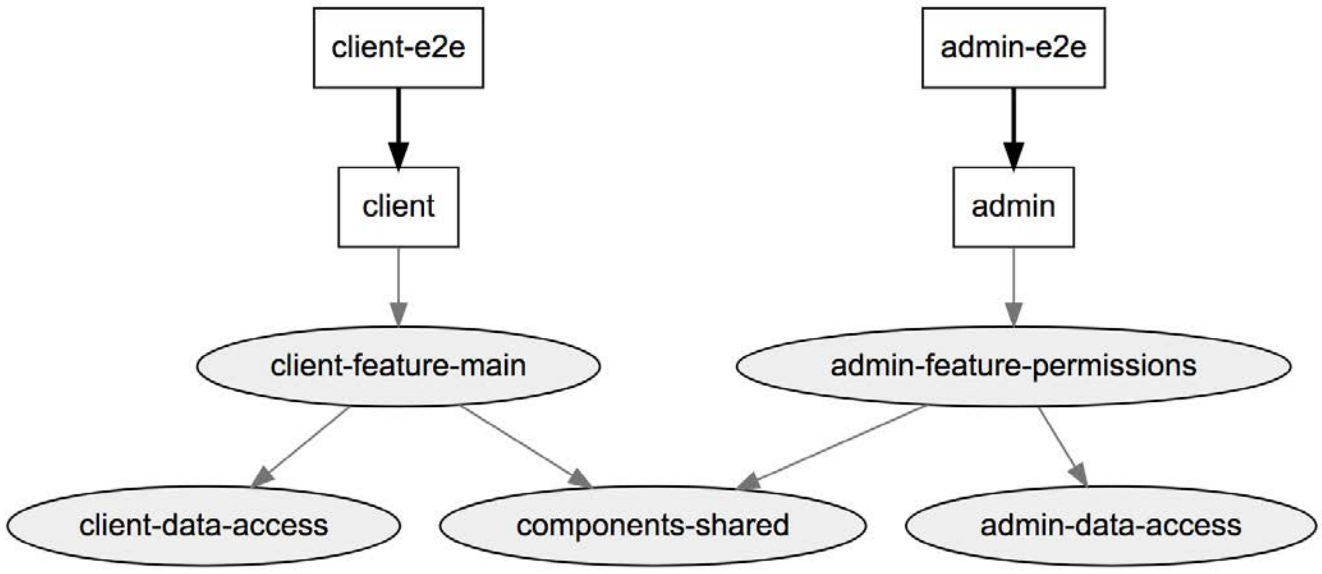
`npm run format:write — --base=[SHA1] --base=[SHA2]`. Nx calculates what changed between the two SHAs, and formats the changed files. For instance, `npm run format:write — --base=origin/master --base=HEAD` formats what is affected by a PR.

The `format:check` command accepts the same options, but instead of formatting files, it throws if any of the files aren't formatted properly. This is useful for CI/CD.

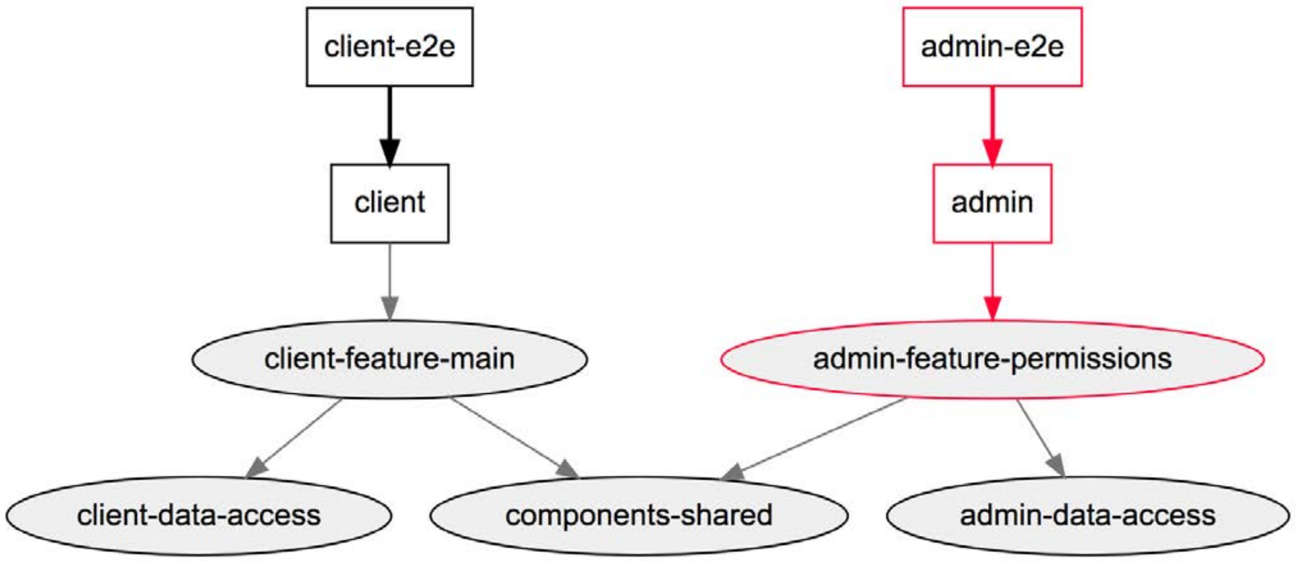
Analyzing and visualizing the dependency graph

To be able to support the monorepo-style development, the tools must know how different projects in your workspace depend on each other. Nx uses advanced code analysis to build this dependency graph.

You can visualize it by running `npm run dep-graph`.



You can also visualize what is affected by your change, by using the `affected:dep-graph` command.



We look at how to inform the `affected` commands which file changes to use to determine all the apps and libraries that are affected by those code changes in **Part 4** of the book. The dep-graph responds by highlighting the affected apps an libs and to visually depict the critical path.

By default, the `dep-graph` and `affected:dep-graph` commands open the browser to display the graph, but you can also output the graph into a file by running:

- `npm run dep-graph — --file=graph.json` emits a json file.
- `npm run dep-graph — --file=graph.dot` emits a dot file.
- `npm run dep-graph — --file=graph.svg` emits an svg file.

Built-in Constraints on Libraries

The following invariants should hold true:

1. a **lib** cannot depend on an **app**

2. an **app-specific** library cannot depend on a lib from another app (e.g., "safe/" **can only depend on libs from "safe/"** or shared libs)
3. a **shared library** cannot depend on on a app-specific lib (e.g., "common-ui/" **cannot depend on "safe/"**)
4. a **ui library** cannot depend on a feature library, or a data-access library.
5. a **utils library** cannot depend on a feature library, data-access library, or a component library.
6. a **data-access library** cannot depend on a feature library, or a component library.
7. a project cannot have circular dependencies
8. a project that lazy loads another project cannot import it directly.

Nx comes with a few predefined rules that apply to all workspaces:

- Libs cannot imports apps.
- A project loading a library via `loadChildren` cannot also import it using an ESM import.
- Circular dependencies aren't allowed.
- Libs cannot be imported using relative imports.

Imposing Your Own Constraints on Library Dependencies

To help with managing complexity, Nx uses code analyses to make sure projects can only depend on each other's well-defined public API. It also allows you to declaratively impose constraints on how projects can depend on each other.

These constraints are enforced statically:

- The IDEs and editors display an error if you are trying to violate these rules
- CI fails

To make constraints are defined and satisfied you need to provision tags when creating new libs:

```
ng g lib feature-destination --directory=booking --tags=booking,feature
```

Note, you can also modify the tags in `nx.json` after the fact, like this:

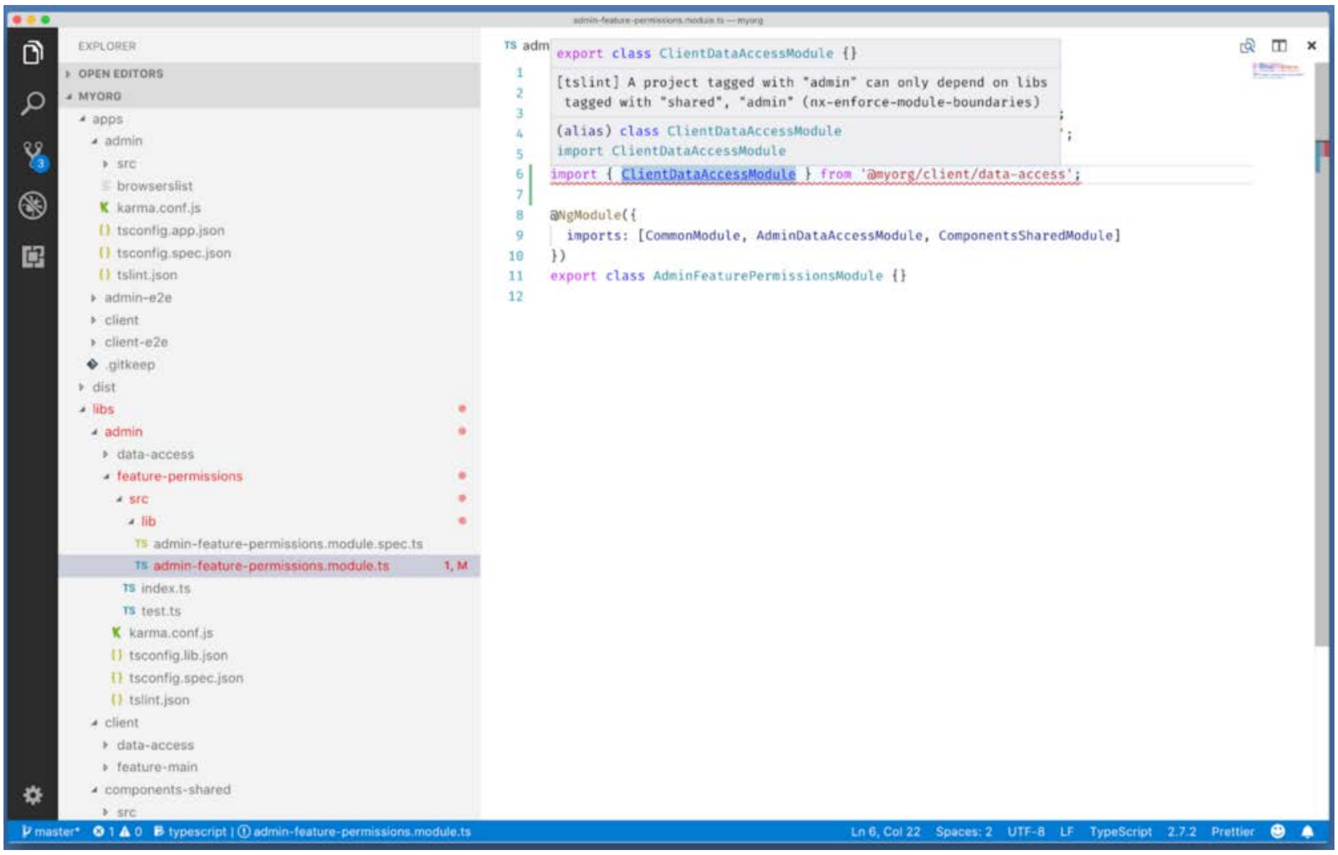
```
  "booking-feature-destination": {
    "tags": ["scope:booking", "type:feature"]
  },
  "booking-shell": {
    "tags": []
},
```

Once tags have associated with each library, tsLint rules can be defined to configure constraints:

```
  "nx-enforce-module-boundaries": [
    true,
    {
      "allow": [],
      "depConstraints": [
          {
            "sourceTag": "scope:shared", ①
            "onlyDependOnLibsWithTags": ["scope:shared"]
          },
          {
            "sourceTag": "scope:booking", ②
            "onlyDependOnLibsWithTags": ["scope:booking", "scope:shared"]
          },
          {
            "sourceTag": "type:util", ③
            "onlyDependOnLibsWithTags": ["type:util"]
          }
        ]
    }
  ]
```

- A lib tagged `scope:shared` can only import from other libs with tag `scope:shared`.
- A lib tagged `scope:booking` can only import from libs tagged with either `scope:booking` or `scope:shared`.
- A lib tagged `type:util` can only import from another lib that is tagged `type:util`.

With the example configuration above, we should see an error when we try to import private client code from the admin part of our repo.



```
export class ClientDataAccessModule {}

[tslint] A project tagged with "admin" can only depend on libs
 tagged with "shared", "admin" (nx-enforce-module-boundaries)

(alias) class ClientDataAccessModule
import ClientDataAccessModule
```

```
import { ClientDataAccessModule } from '@myorg/client/data-access';

@NgModule({
  imports: [CommonModule, AdminDataAccessModule, ComponentsSharedModule]
})
export class AdminFeaturePermissionsModule {}
```

With dependency constraints, another team won't create a dependency on your internal library. You can define which projects contain components, NgRx code, and features, so you, for instance, can disallow projects containing presentational UI components from depending on NgRx. You can define which projects are experimental and which are stable, so stable applications cannot depend on experimental projects etc.

Implicit Dependencies

Nx uses its built-in intelligence to create the dependency graph of the apps and libs, and that gets used to figure out what needs to be rebuilt and retested. There are certain files, however, that Nx cannot analyze. That's why Nx has support for implicit dependencies. They are defined in nx.json.

```
{
  "npmScope": "mycompany",
  "implicitDependencies": {
    "package.json": "*",
    "angular.json": "*",
    "tsconfig.json": "*",
    "tslint.json": "*",
    "nx.json": "*"
},
"projects": {}
}
```

The `"package.json": "*"` line tells Nx that a change to package.json affects every single project. Since the root-level README file isn't listed, changing it won't affect anything.

We can be more specific and list the projects that are affected by a particular file.

```
{
  "npmScope": "mycompany",
  "implicitDependencies": {
    "package.json": "*",
    "angular.json": "*",
    "tsconfig.json": "*",
    "tslint.json": "*",
    "nx.json": "*",

    "tools/scripts/app1-rename-bundles.js": ["app1"]
  },
  "projects": {}
}
```

In addition to being able to list implicit dependencies between files and projects, we can also add implicit dependencies between projects. For instance, an Nx workspace can contain the backend code for our Angular app. The back-end code implicitly depends on the Angular app, but it is not expressed in the code and so cannot be deduced by Nx. We can explicitly define it as follows:

```
{
  "npmScope": "mycompany",
  "implicitDependencies": {
    "package.json": "*",
    "angular.json": "*",
    "tsconfig.json": "*",
    "tslint.json": "*",
    "nx.json": "*",

    "tools/scripts/app1-rename-bundles.js": ["app1"]
  },
  "projects": {
    "app": {},
    "backend": {
      "implicitDependencies": ["app"]
    }
  }
}
```

Exceptions

As with everything there are exceptions, which can also add to your `tslint.json`.

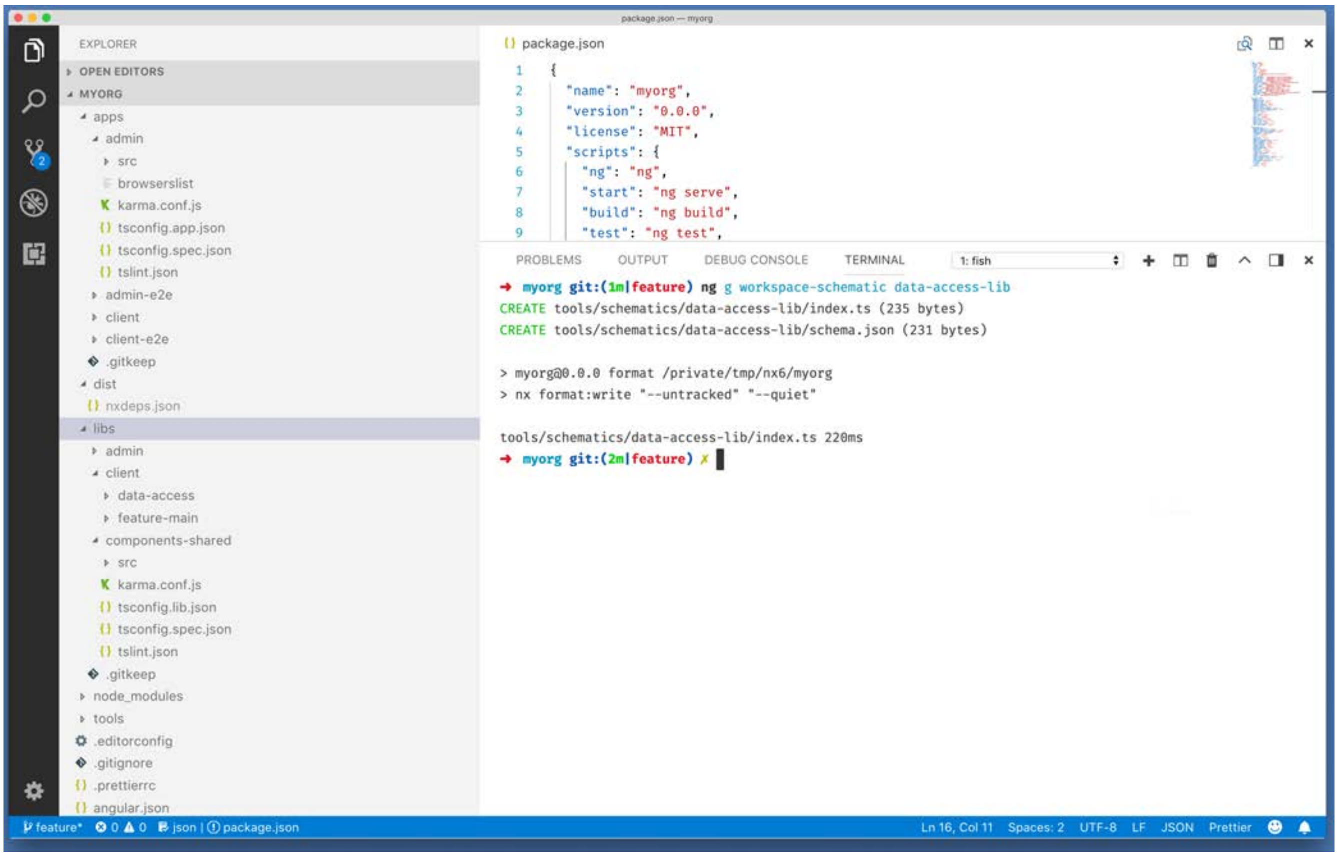
```
"nx-enforce-module-boundaries": [
  true,
  {
    "allow": ["@myworkspace/mylib"],
    "depConstraints": [
    ]
  }
]
```

Workspace Schematics

Some best practices are well-established in the Angular community. For instance, most non-trivial Angular projects use NgRx, so Nx comes with a set of built-in code generators and runtime libraries to make sure your applications' state management is consistent. Just run `ng g ngrx accounts --module=one/two/app.module.ts` to set everything up (read more [here](https://nrwl.io/nx/guide-setting-up-ngrx)).

Some, however, are specific to your organization. Nx can help with those as well. Say, for example, we want to promote a pattern of encapsulating NgRx-related code into data-access libraries. This is how easy it is to do it with Nx:

Start by generating a new workspace schematic.



package.json

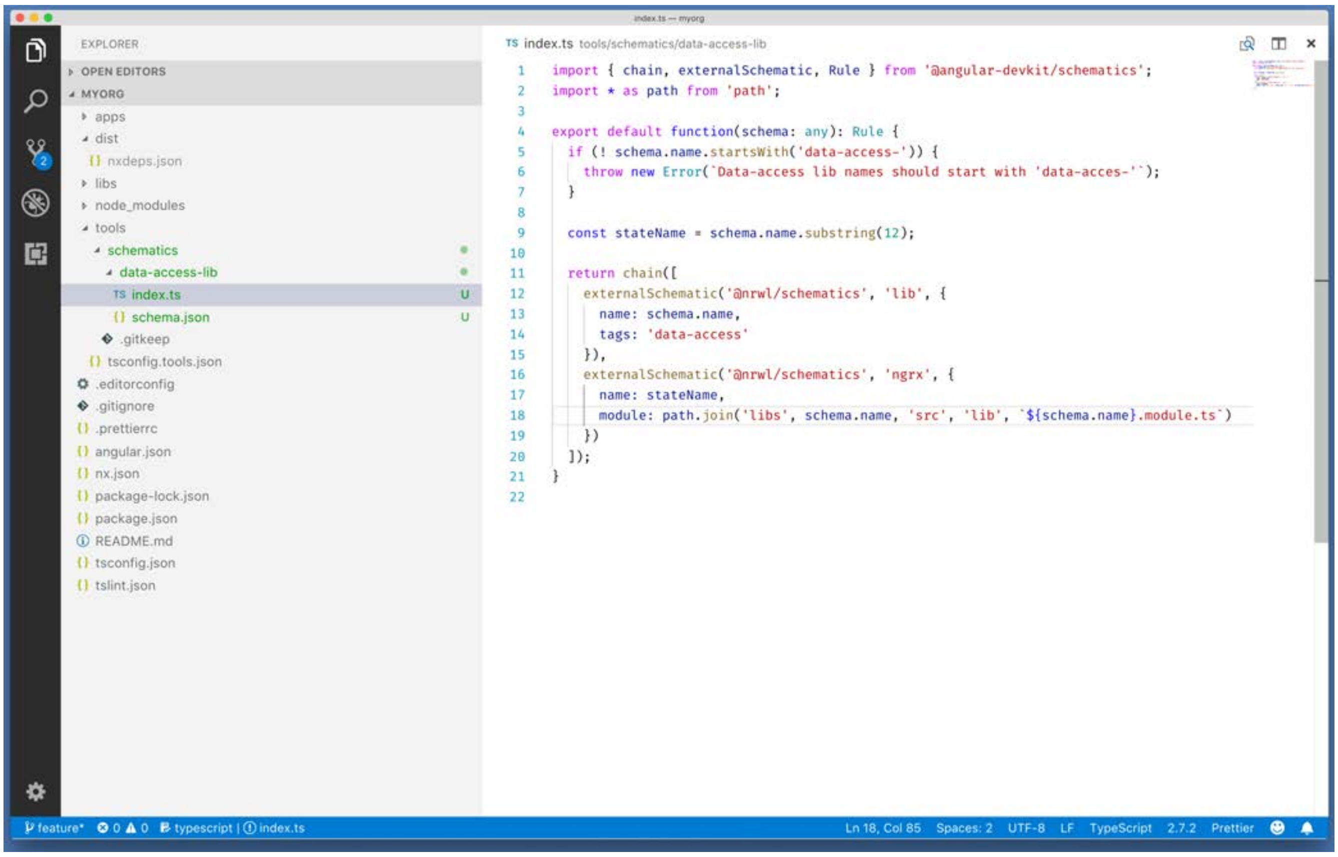
```
{
  "name": "myorg",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
```

```
➜ myorg git:(1m|feature) ng g workspace-schematic data-access-lib
CREATE tools/schematics/data-access-lib/index.ts (235 bytes)
CREATE tools/schematics/data-access-lib/schema.json (231 bytes)

> myorg@0.0.0 format /private/tmp/nx6/myorg
> nx format:write "--untracked" "--quiet"

tools/schematics/data-access-lib/index.ts 220ms
➜ myorg git:(2m|feature) ✗
```

Then, provide the implementation.



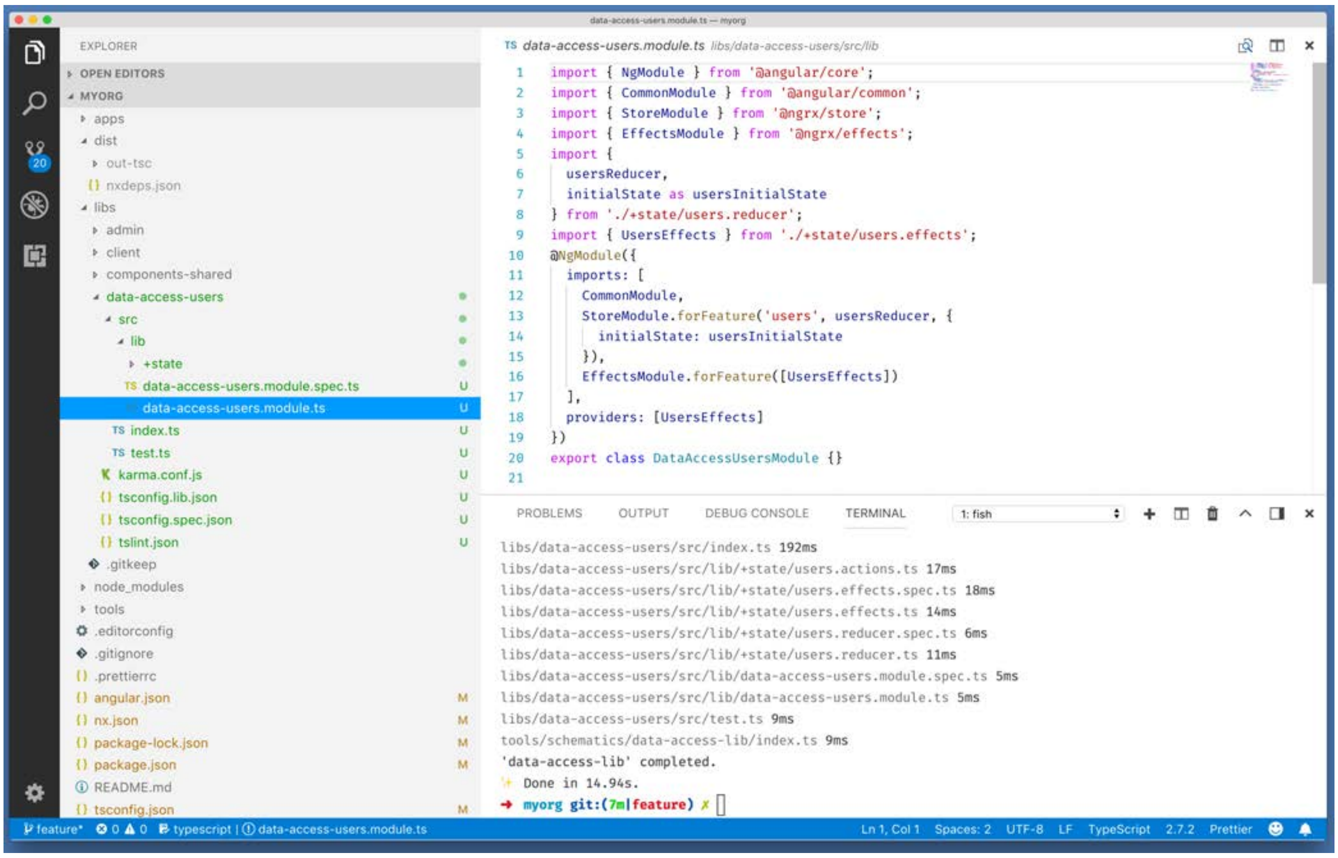
```
import { chain, externalSchematic, Rule } from '@angular-devkit/schematics';
import * as path from 'path';

export default function(schema: any): Rule {
  if (! schema.name.startsWith('data-access-')) {
    throw new Error(`Data-access lib names should start with 'data-acces-'`);
  }

  const stateName = schema.name.substring(12);

  return chain([
    externalSchematic('@nrwl/schematics', 'lib', {
      name: schema.name,
      tags: 'data-access'
    }),
    externalSchematic('@nrwl/schematics', 'ngrx', {
      name: stateName,
      module: path.join('libs', schema.name, 'src', 'lib', `${schema.name}.module.ts`)
    })
  ]);
}
```

Finally, invoke it to generate a new data-access lib.



```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { StoreModule } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';
import {
  usersReducer,
  initialState as usersInitialState
} from './+state/users.reducer';
import { UsersEffects } from './+state/users.effects';
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forFeature('users', usersReducer, {
      initialState: usersInitialState
    }),
    EffectsModule.forFeature([UsersEffects])
  ],
  providers: [UsersEffects]
})
export class DataAccessUsersModule {}
```

```
libs/data-access-users/src/index.ts 192ms
libs/data-access-users/src/lib/+state/users.actions.ts 17ms
libs/data-access-users/src/lib/+state/users.effects.spec.ts 18ms
libs/data-access-users/src/lib/+state/users.effects.ts 14ms
libs/data-access-users/src/lib/+state/users.reducer.spec.ts 6ms
libs/data-access-users/src/lib/+state/users.reducer.ts 11ms
libs/data-access-users/src/lib/data-access-users.module.spec.ts 5ms
libs/data-access-users/src/lib/data-access-users.module.ts 5ms
libs/data-access-users/src/test.ts 9ms
tools/schematics/data-access-lib/index.ts 9ms
'data-access-lib' completed.
✨ Done in 14.94s.
➜ myorg git:(feature) ✗
```

We generally recommend that schematics reside in the `/tools` folder.

Handling back-end calls with DataPersistence

Managing state is a hard problem. We need to coordinate multiple backends, web workers, and UI components, all of which update the state concurrently.

What should we store in memory vs. the URL? What about the local UI state? How do we synchronize the persistent state, the URL, and the state on the server? All these questions have to be answered when designing the state management of our applications. Nx Data Persistence is a set of helper functions that enables the developer to manage state with an intentional synchronization strategy and handle error state. Refer to Using NgRx 4 to Manage State in Angular Applications for more detailed example of the state problem Data Persistence is solving. [1: https://blog.nrwl.io/using-ngrx-4-to-manage-state-in-angular-applications-64e7a1f84b7b]

Why use the DataPersistence library?

The library was designed to abstract some of the logic that is common to Effects:

1. Fetching the state from the Store to retrieve one or more values during the asynchronous operation
2. Updating the UI pessimistically or optimistically
3. Multiple calls to the same Effect need to be handled in a reliable way
4. Error handling is often forgotten when using a `switchMap` operation, leading to the Effect completing when an error is encountered (and therefore not being able to respond to further actions that trigger it)

Optimistic Updates

For a better user experience, `optimisticUpdate` method updates the state on the client application immediately before updating the data on the server-side. While it addresses fetching data in order, removing the race conditions and handling error, it is optimistic about not failing to update the server. In case of a failure, when using `optimisticUpdate`, the local state on the client is already updated. The developer must provide an undo action to restore the previous state to keep it consistent with the server state. The error handling must be done in the callback, or by means of the undo action.

```
import { DataPersistence } from '@nrwl/nx';
...

class TodoEffects {
  @Effect() updateTodo = this.dataPersistence.optimisticUpdate('UPDATE_TODO', {
    // provides an action and the current state of the store
    run: (a: UpdateTodo, state: TodosState) => {
      return this.backend(state.user, a.payload);
    },

    undoAction: (a: UpdateTodo, e: any) => {
      // dispatch an undo action to undo the changes in the client state
      return ({
        type: 'UNDO_UPDATE_TODO',
        payload: a
      });
    }
  });

  constructor(
    private dataPersistence: DataPersistence<TodosState>,
    private backend: Backend
  ) {}
}
```

Pessimistic Updates

To achieve a more reliable data synchronization, `pessimisticUpdate` method updates the server data first before updating the UI. When the change is reflected in the server state, a change is made in the client state by dispatching an action. The `pessimisticUpdate` method enforces the order of the fetches and error handling.

```
import { DataPersistence } from '@nrwl/nx';
...

@Injectable()
class TodoEffects {
  @Effect() updateTodo = this.dataPersistence.pessimisticUpdate('UPDATE_TODO', {
    // provides an action and the current state of the store
    run: (a: UpdateTodo, state: TodosState) => {
      // update the backend first, and then dispatch an action that
      // updates the client side
      return this.backend(state.user, a.payload).map(updated => ({
        type: 'TODO_UPDATED',
        payload: updated
      }));
    },

    onError: (a: UpdateTodo, e: any) => {
      // we don't need to undo the changes on the client side.
      // we can dispatch an error, or simply log the error here and return `null`
      return null;
    }
  });

  constructor(
    private dataPersistence: DataPersistence<TodosState>,
    private backend: Backend
  ) {}
}
```

Data Fetching

DataPersistence's fetch method provides consistency when fetching data. If there are multiple requests scheduled for the same action it only runs the last one.

```
import { DataPersistence } from '@nrwl/nx';
...

@Injectable()
class TodoEffects {
  @Effect() loadTodos = this.dataPersistence.fetch('GET_TODOS', {
   // provides an action and the current state of the store
   run: (a: GetTodos, state: TodosState) => {
     return this.backend(state.user, a.payload).map(r => ({
       type: 'TODOS',
       payload: r
     }));
   },

   onError: (a: GetTodos, e: any) => {
     // dispatch an undo action to undo the changes in the client state
     return null;
   }
 });

  constructor(
    private dataPersistence: DataPersistence<TodosState>,
    private backend: Backend
  ) {}

}
```

This is correct, but we can improve the performance by supplying and id of the data by using an accessor function and adding concurrency to the fetch action for different ToDo's.

```
@Injectable()
class TodoEffects {
  @Effect() loadTodo = this.dataPersistence.fetch('GET_TODO', {
    id: (a: GetTodo, state: TodosState) => {
      return a.payload.id;
    },

    // provides an action and the current state of the store
    run: (a: GetTodo, state: TodosState) => {
      return this.backend(state.user, a.payload).map(r => ({
        type: 'TODO',
        payload: r
      }));
    },

    onError: (a: GetTodo, e: any) => {
      // dispatch an undo action to undo the changes in the client state
      return null;
    }
  });

  constructor(
    private dataPersistence: DataPersistence<TodosState>,
    private backend: Backend
  ) {}

}
```

With this setup, the requests for Todo run concurrently with the requests for Todo 2. Consecutive calls for Todo are queued behind the first.

Data Fetching On Router Navigation

Since the user can always interact with the URL directly, we should treat the router as the source of truth and the initiator of actions. In other words, the router should invoke the reducer, not the other way around.

When our state depends on navigation, we can not assume the route change happened when a new url is triggered but when we actually know the user was able to navigate to the url. DataPersistence `navigation` method checks if an activated router state contains the passed in component type, and, if it does, runs the `run` callback. It provides the activated snapshot associated with the component and the current state. And it only runs the last request.

```
import { DataPersistence } from '@nrwl/nx';
...

@Injectable()
class TodoEffects {
  @Effect() loadTodo = this.dataPersistence.navigation(TodoComponent, {
    run: (a: ActivatedRouteSnapshot, state: TodosState) => {
      return this.backend.fetchTodo(a.params['id']).map(todo => ({
        type: 'TODO_LOADED',
        payload: todo
      }));
    },

    onError: (a: ActivatedRouteSnapshot, e: any) => {
      // we can log and error here and return null
      // we can also navigate back
      return null;
    }
  });

  constructor(
    private dataPersistence: DataPersistence<TodosState>,
    private backend: Backend
  ) {}

}
```

Part 4: Helping with builds and CI

Rebuilding and retesting only affected apps and libs

When we consider large repositories that contain many dozens and hundreds of libs and many apps, we realize that it would be very difficult to have to test and build all of them whenever code new is merged. It doesn't scale. Neither does a manual trace fo the dependencies: it would be unreliable. It would be great if there was a reliable way for us to only test and build the affected libraries and apps.

Nx uses code analysis to determine what needs to be rebuild and retested. It provides this via the `affected` commands: `affected:build`, `affected:test`, and `affected:e2e`. These commands can be run with the following options to determine only those libraries and apps (aka. "projects") that are affected by your code changes.

Options to target specific projects

- **Compare changes between 2 git commits:** You can run `--base=SHA1 --head=SHA2`, where SHA1 is the one you want to compare with and SHA2 contains the changes. This generates a list of files that we can use to determine which projects are affected by those changes and only process those.
- **Uncommited changes:** Use the flag `--uncommited` to use all the uncommitted changes on the current branch to obtain the list of affected projects to process (files that were changed since the last commit: you can use `git status` to view them)
- **Untracked:** Use the flag `--untracked` to use the list of only untracked files (files that were created since the last commit) to determine the projects to process (useful during development)
- **Explicit files:** Use `--files` to provide an explicit comma-delimited list of files (useful during development)
- **All projects:** Use `--all` to force the command for all projects instead of only those affected by code changes
- **Only last failed:** Use `--only-failed` to isolate only those projects which previously failed (default: false)

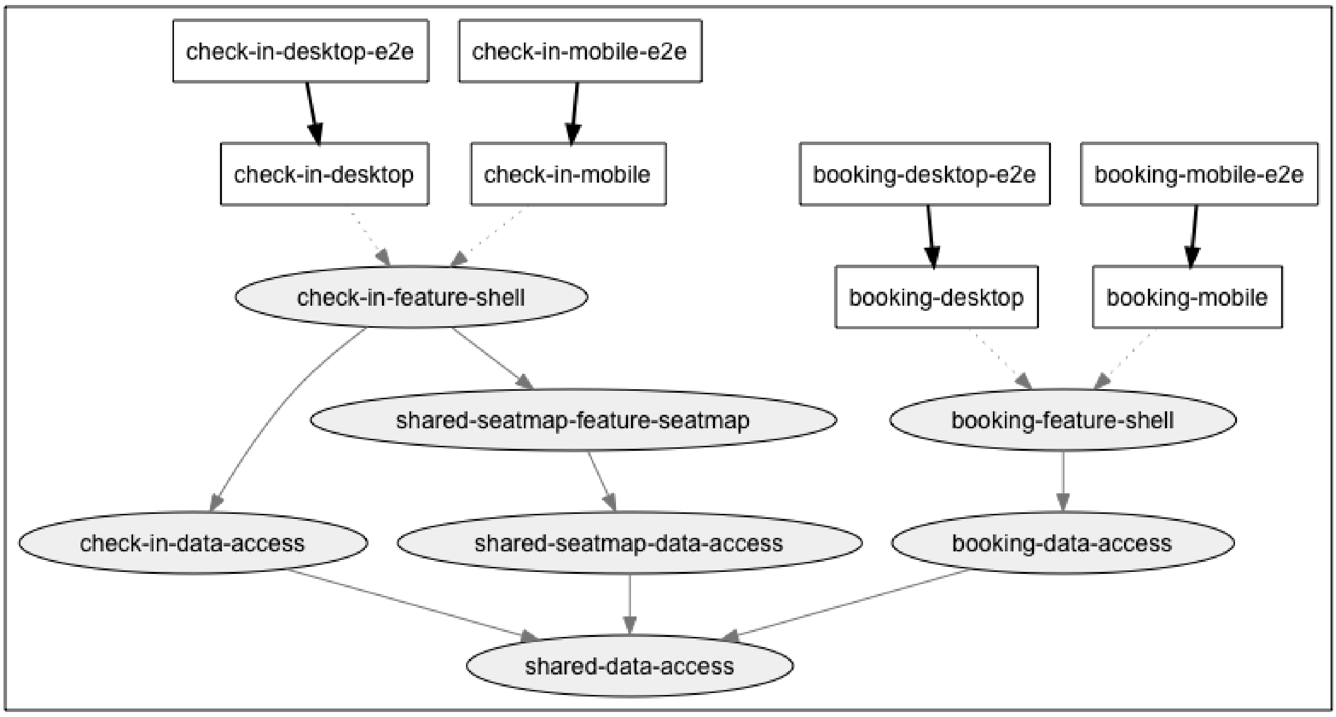
Additional options

- `--parallel`: Parallel-ize the command (default: false)
- `--maxParallel`: Set the maximum of parallel processes (default: 3)
- `--exclude`: Exclude certain projects from being processed (comma-delimited list)

> All other options are passed to the underlying CLI command.

A walkthrough

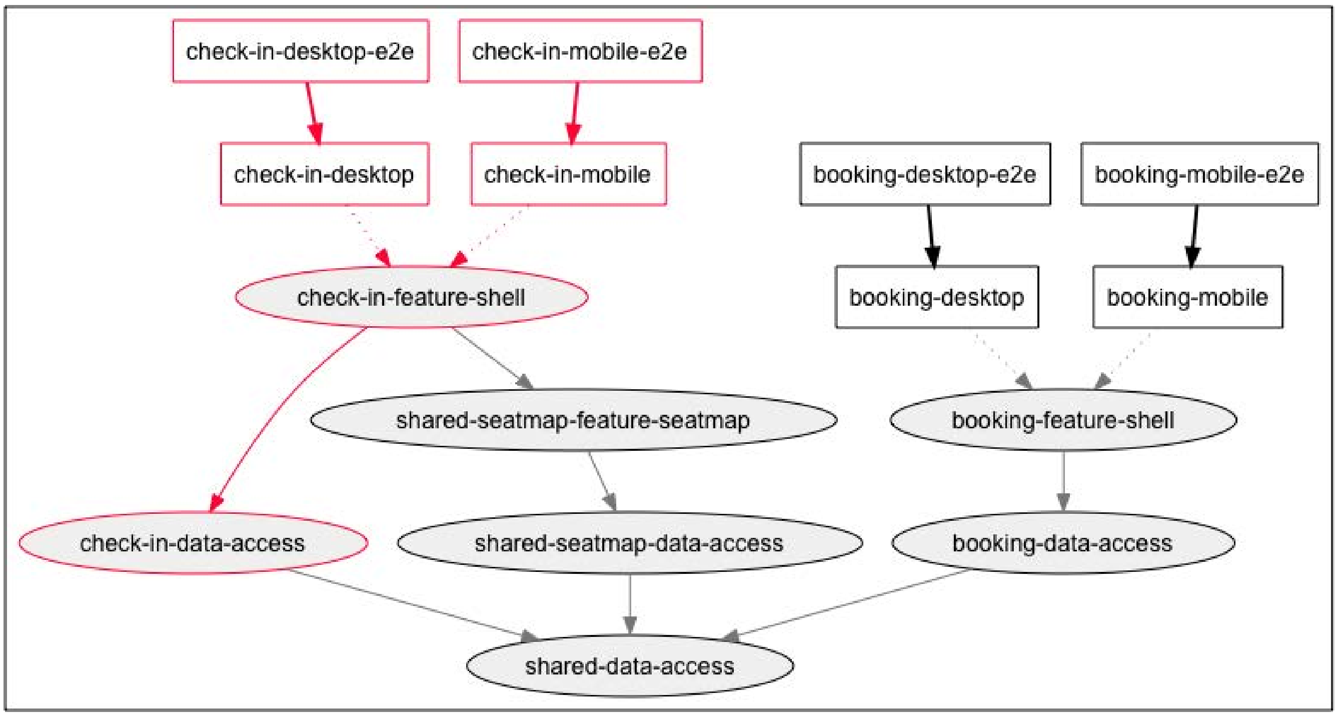
Let's take as an example our sample repo (nrwl-airlines). Its dependency graph is as below:


check-in-desktop-e2e
check-in-mobile-e2e
check-in-desktop
check-in-mobile
booking-desktop-e2e
booking-mobile-e2e
check-in-feature-shell
booking-desktop
booking-mobile
shared-seatmap-feature-seatmap
booking-feature-shell
check-in-data-access
shared-seatmap-data-access
booking-data-access
shared-data-access

Scenario 1: Change in a app-specific library

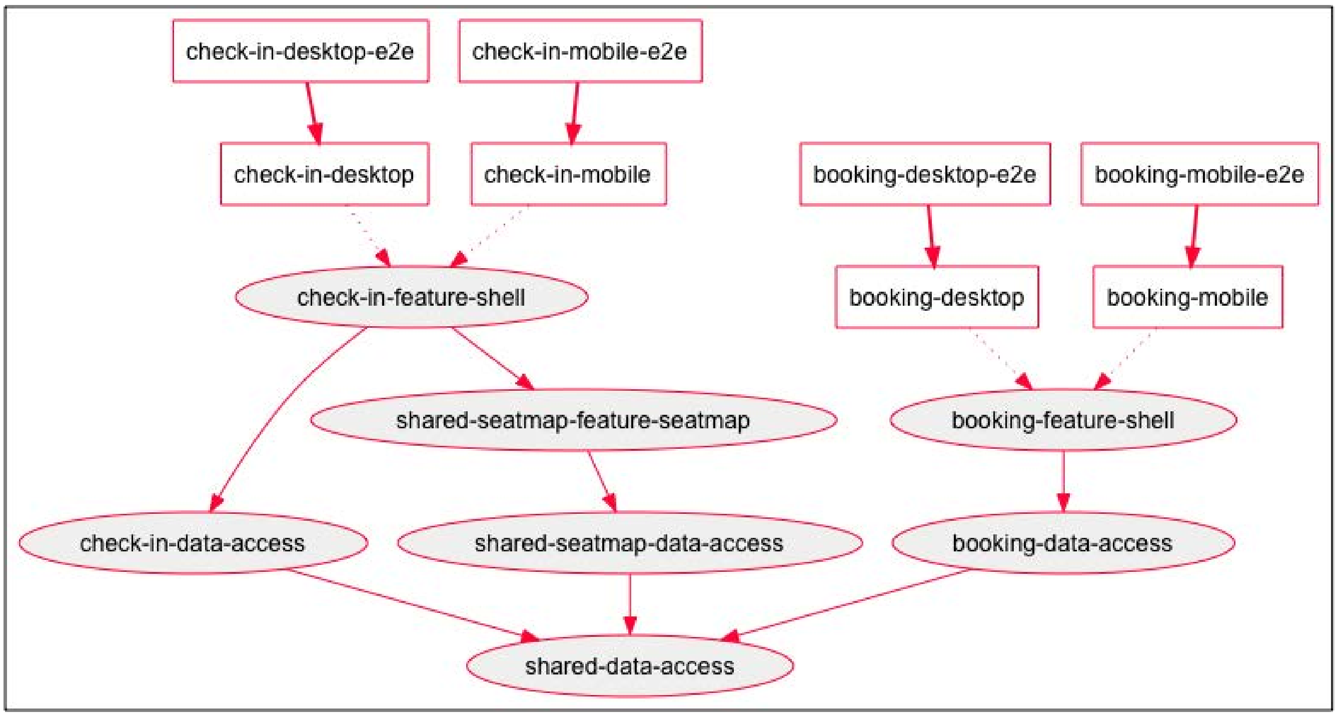
If we change a file in the `check-in-data-access` library, we can now see that it only affects `check-in-feature-shell`, `check-in-desktop`, and `check-in-mobile`.

> **We don't have to touch `booking` or `seatmap`!**



Scenario 2: Change in a shared library

If we now change a file in the `shared-data-access` library, we see that it affects all the projects in the workspace! The impact is large and so we know that we have to take extra care and to reach out to the other projects' owners to make sure that everyone is aware of the changes.



Advantages to using "affected" commands

The difference in Scenario 1 and Scenario 2 is pretty clear: rebuilding or retesting only the affected projects can have a huge impact on the amount of time that the builds and tests take. We also don't have to deploy changes to projects that haven't changed.

Running commands in parallel

When executing these commands, Nx topologically sorts the projects, and runs what it can in parallel. But we can also explicitly pass `--parallel` like so:

```
npm run affected:build -- --base=master --parallel
npm run affected:test -- --base=master --parallel
npm run affected:e2e -- --base=master --parallel
```

We can also pass `--maxParallel` to specify the number of parallel processes.

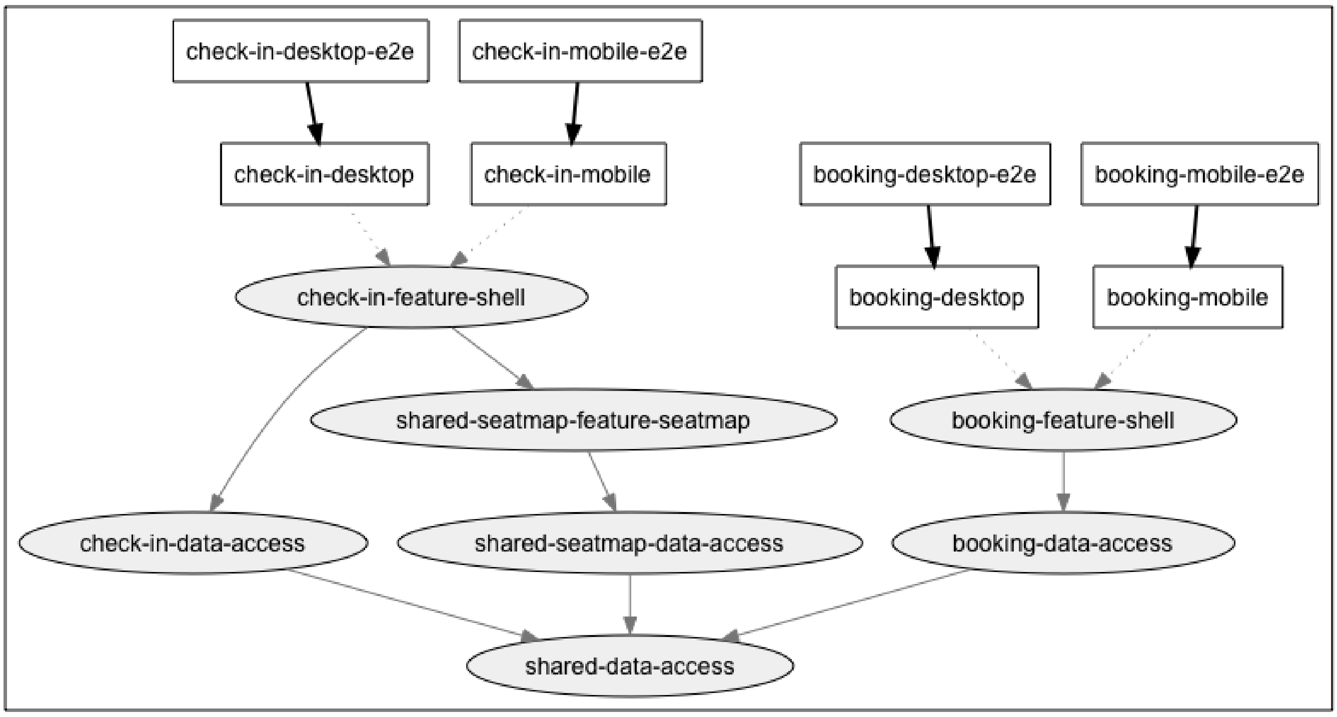
Part 5: Development Challenges in a Monorepo

Here are some of the common challenges with moving into a monorepo:

1. How do we manage releases to various environments when there are many teams and applications in the same repository?
2. How can we manage dependencies between projects? Many projects can be running simultaneously and we might need to reference older versions of some code.
3. How do we organize the shared code so that it is reusable without incurring too much technical debt and without needing constant maintenance?

We looked at Issue #3 in **Part 2** of the book (Organizing code with Libraries). The rest of this section focuses on how to allow teams to work together in a single repository and still have the flexibility to deploy different applications.

Let's consider the example repo. The dependency graph is below:



Our repo has 4 applications that need to be deployed: desktop and mobile versions of `booking` and `check-in`. All of the apps have an implicit dependency on a `shared-data-access` library.

Listed below are some of the common challenges that can present themselves:

1. The booking team makes changes to `shared-data-access`: how can they communicate the changes to the check-in team to ensure that nothing is broken?
2. The check-in team failed a QA check and need to make a code fix; however there is new code in the repo from the booking team.
3. The seatmap team discovers a bug in prod: how can we hotfix this and ensure that the fix is also

in our codebase?

4. The builds and tests take a long time in CI: how can we reduce the amount of time?
5. How can we ensure that our trunk branch is deployable? Should we even adopt trunk-based development?

How to deal with code changes from another team

There are a few ways to minimize the effects of code changes from other teams:

1. Ensure that the **repository settings disallow plain merges** and instead only allow rebased (or fast-forward) merges. This ensures that the developer has all of the latest changes (and has tested the code) from the shared branch before merging the PR.
2. Ensure that **PRs are very small in scope.** This makes the risk a lot lower and also allows for better review (through code review as well as manual testing).
3. **Use feature toggles** so that features that are still in development are not visible to the end user (See below section).
4. **Be aware (and make others aware) of changes to shared code** and minimize the risk in the following way:
    1. Create a new version of the code (method/class/library)
    2. Develop on this new version until it is ready
    3. Issue a deprecation warning for the old method/class/library with a set expiry time
    4. Work with the other teams to help them migrate to the new version
    5. Remove the old version when the expiry time elapses

Code owners

One of the other mitigation strategies is to assign code owners for the libs in the Nx workspace. A code owner (is usually a group rather than a single individual) is responsible for all changes to a lib, and would orchestrate the process outlined above to deprecate and migrate to newer versions of the code.

Working with a code owner for shared code removes some of the guesswork and helps to formulate a plan when it comes to modifying shared code. There are fewer surprises when it comes time for integration.

Feature toggles

There are two types of feature toggles: build-time and run-time. Build-time toggles are recommended for initialization settings (connection/URL settings, layout settings, etc.) which would be needed when the application loads, so that we are not waiting for a network request to complete before rendering the app.

Run-time feature-toggles are usually implemented via network calls to a settings file hosted on the server. These are meant for scenarios where we want to control the settings without rebuilding the application. It needs careful management for handling the caching behavior.

Trunk-based development

We generally encourage trunk-based development because we believe in the basics tenets:

- Branches should be short-lived and should be very specific to accomplishing a single piece of work
- We believe that all code should be as up-to-date as possible with other teams so that there is no lengthy integration process
- All code that is being merged into the trunk branch has already been tested at a basic level

There are some organizational challenges to implementing this: release deadlines, teams that need to work in isolation as part of a research or secret effort, picking up unexpected changes in shared code when trying to implement a separate feature, etc.

Below is our recommendation for working with a monorepo. It needs to be tailored to your organization, but can offer some guidance on what works in general for most organizations.

A recommended git strategy

We recommend the following:

- Always use Pull Requests when merging code. A PR has two purposes:
- To initiate a conversation and to get feedback on the implementation
- To initiate a build and to run tests and lint checks to ensure that we don't break the build
- Avoid long-running branches and don't merge branches locally
- Enforce a git merging strategy that ensures that feature branches are up-to-date before merging. This ensures that these branches are tested with the latest code before the merge.

The website trunkbaseddevelopment.com contains a lot of very helpful information on trunk-based development and is a great resource.

The following sections are the most pertinent:

- Feature flags
- Strategy for migrating code
- Feature branches

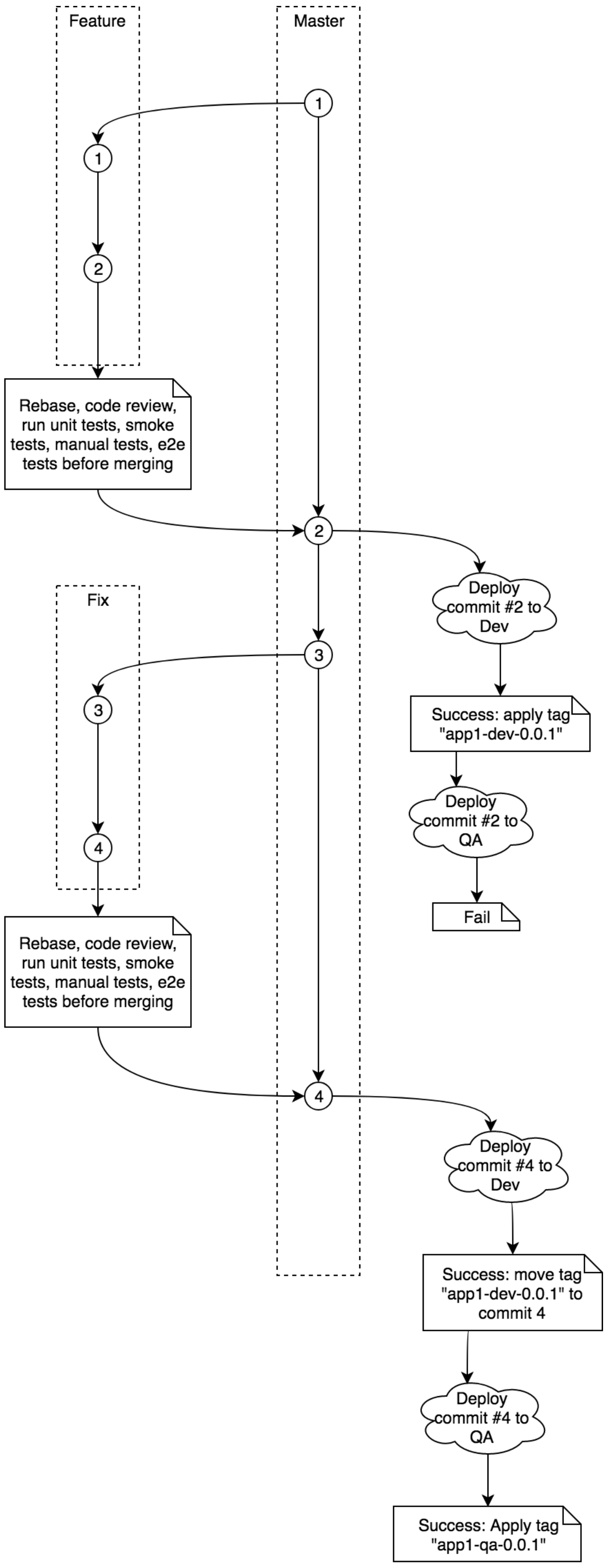

Feature
Master
1
1
2
Rebase, code review, run unit tests, smoke tests, manual tests, e2e tests before merging
2
Deploy commit #2 to Dev
Fix
3
3
Success: apply tag "app1-dev-0.0.1"
Deploy commit #2 to QA
4
Fail
Rebase, code review, run unit tests, smoke tests, manual tests, e2e tests before merging
4
Deploy commit #4 to Dev
Success: move tag "app1-dev-0.0.1" to commit 4
Deploy commit #4 to QA
Success: Apply tag "app1-qa-0.0.1"

Figure 3. Trunk based Development

Summary

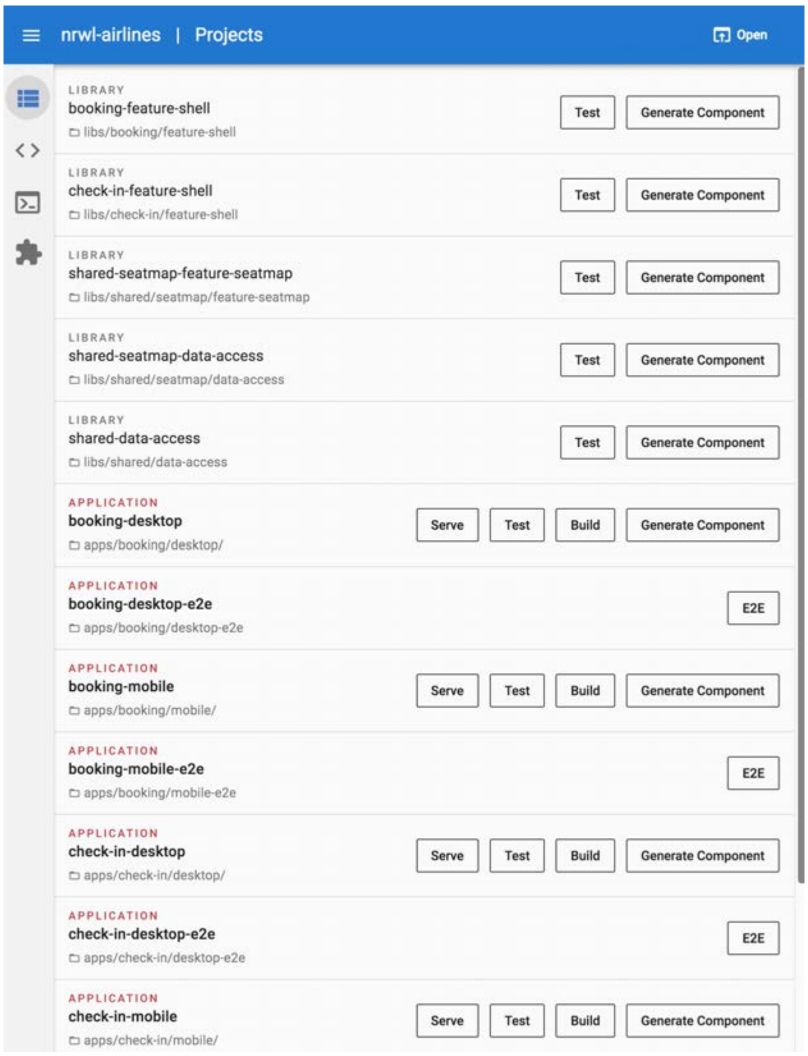
In this section we took a brief look at some of the common issues that teams run into when adopting a monorepo workflow.

- Some strategies to deal with overlapping code between teams: smaller PRs, requiring that PRs ar eup-to-date with the target branch, and a way to version the code so that there is a transition plan.
- Setting up code owners for various libs so that there is a responsibility both for merging code into that library and also to promote the changes responsibly to the affected teams.
- Feature flags allow teams to tests their code in environments without affecting or being affected by other teams. Run-time flags avs. build-time strategies were discussed.
- trunkbaseddevelopment.com is a valuable resource

Appendix A: Other environments

Users can choose to use a graphical UI for Nx and the Angular CLI. The Angular Console allows developers to interact with Nx in a visual way. All the CLI options are visible and interactive, and commands can be previewed and executed directly within the interface (the commands are executed using the Angular CLI under the hood).

The screenshot below is a glimpse of the sample workspace in Angular Console.



LIBRARY
booking-feature-shell
libs/booking/feature-shell
Test | Generate Component

LIBRARY
check-in-feature-shell
libs/check-in/feature-shell
Test | Generate Component

LIBRARY
shared-seatmap-feature-seatmap
libs/shared/seatmap/feature-seatmap
Test | Generate Component

LIBRARY
shared-seatmap-data-access
libs/shared/seatmap/data-access
Test | Generate Component

LIBRARY
shared-data-access
libs/shared/data-access
Test | Generate Component

APPLICATION
booking-desktop
apps/booking/desktop/
Serve | Test | Build | Generate Component

APPLICATION
booking-desktop-e2e
apps/booking/desktop-e2e
E2E

APPLICATION
booking-mobile
apps/booking/mobile/
Serve | Test | Build | Generate Component

APPLICATION
booking-mobile-e2e
apps/booking/mobile-e2e
E2E

APPLICATION
check-in-desktop
apps/check-in/desktop/
Serve | Test | Build | Generate Component

APPLICATION
check-in-desktop-e2e
apps/check-in/desktop-e2e
E2E

APPLICATION
check-in-mobile
apps/check-in/mobile/
Serve | Test | Build | Generate Component

Figure 4. Angular Console

Features

Quick actions

Apps listed in the workspace view have quick-access buttons to serve, build, test and create components. Further actions are found under `Run tasks` in the left menu.

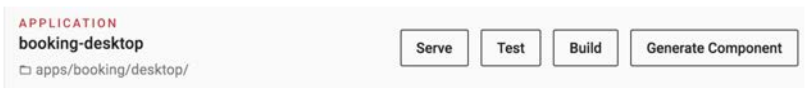


Figure 5. Application quick actions

Libs have quick actions for test and create component. Further actions would also be found under `Run tasks` in the left menu.

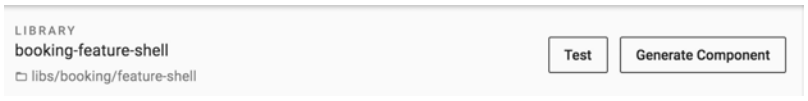


Figure 6. Library quick actions

Generate schematics

We can generate any of the schematics detected in the workspace (including custom ones) by picking from the grouped list.

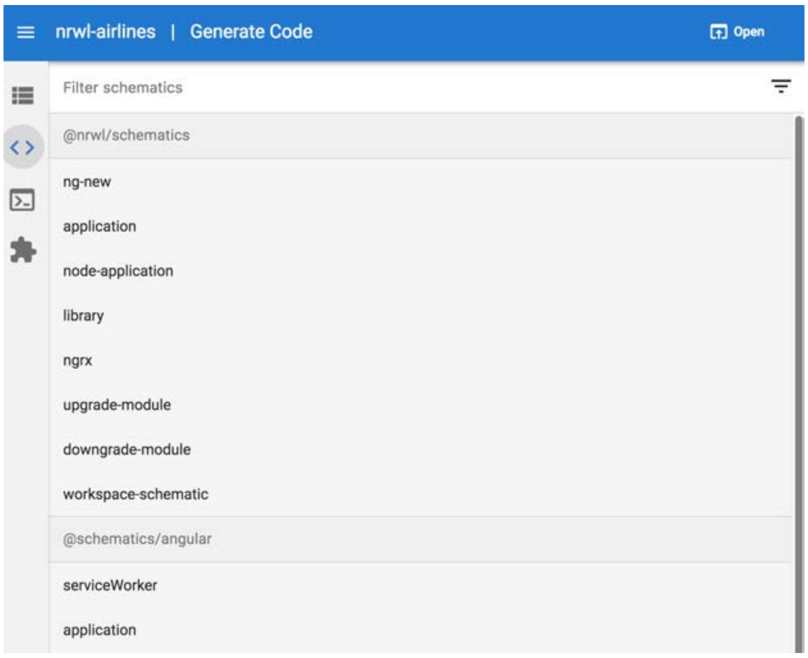


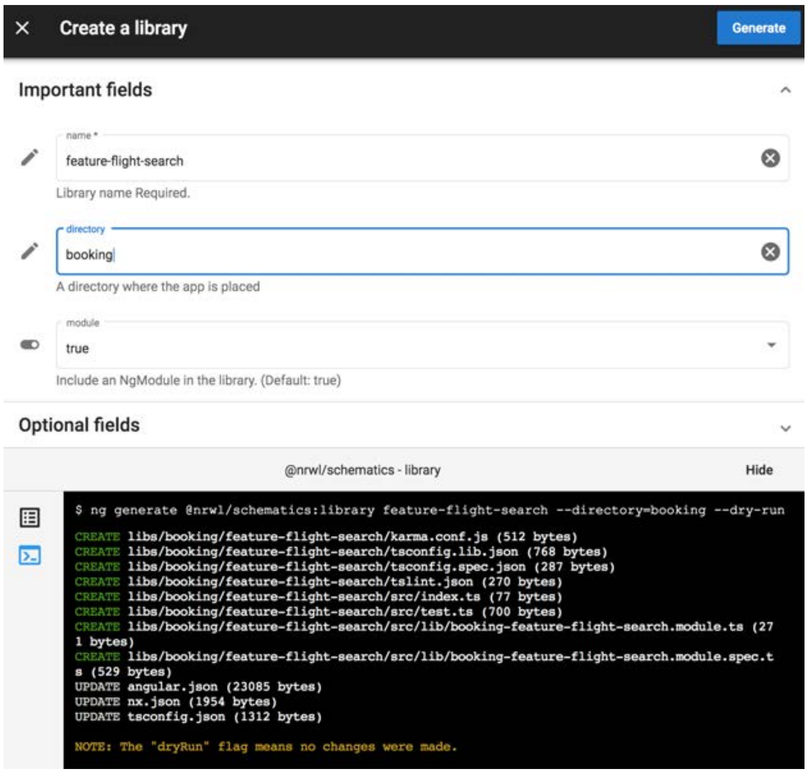
Figure 7. Code generation support

Option completion

There is the ability to pick from a list of choices instead of manual typing. This comes in handy when specifying a path to a module since you don't need to type in the path manually. Also, toggles are visual, allowing you to avoid having to decide between `true`, `false`, `--no-option`, etc.

View all available options

One can see all the command-line options available. These are grouped into "required" and "optional". There is also a running output displaying the expected result of running a command, so that corrections can be made before executing the command.



name *: feature-flight-search
Library name Required.

directory: booking
A directory where the app is placed

module: true
Include an NgModule in the library. (Default: true)

```
$ ng generate @nrwl/schematics:library feature-flight-search --directory=booking --dry-run

CREATE libs/booking/feature-flight-search/karma.conf.js (512 bytes)
CREATE libs/booking/feature-flight-search/tsconfig.lib.json (768 bytes)
CREATE libs/booking/feature-flight-search/tsconfig.spec.json (287 bytes)
CREATE libs/booking/feature-flight-search/tslint.json (270 bytes)
CREATE libs/booking/feature-flight-search/src/index.ts (77 bytes)
CREATE libs/booking/feature-flight-search/src/test.ts (700 bytes)
CREATE libs/booking/feature-flight-search/src/lib/booking-feature-flight-search.module.ts (27
1 bytes)
CREATE libs/booking/feature-flight-search/src/lib/booking-feature-flight-search.module.spec.t
s (529 bytes)
UPDATE angular.json (23085 bytes)
UPDATE nx.json (1954 bytes)
UPDATE tsconfig.json (1312 bytes)

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

Figure 8. All options and expected output

Run npm scripts

Angular Console reads in all of the commands specified under `scripts` in `package.json` and allows their execution.

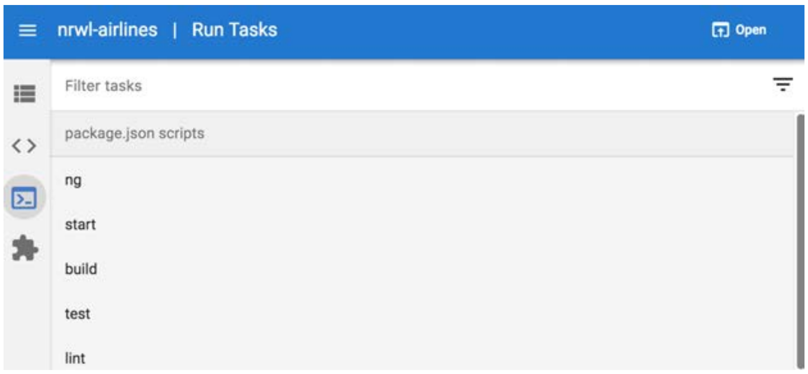
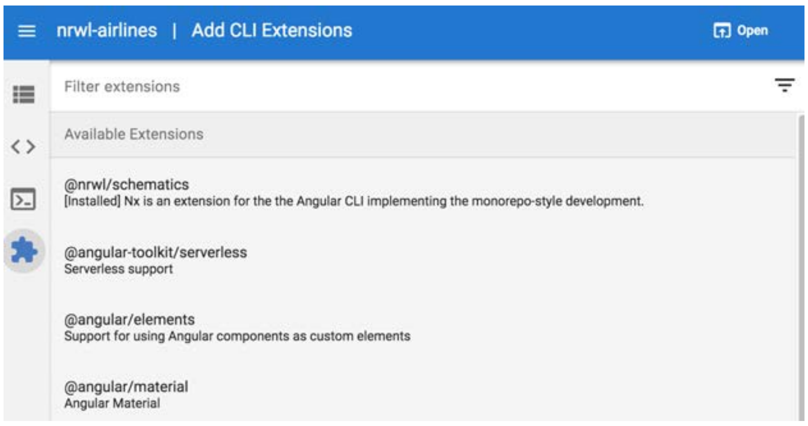


Figure 9. Run any task in package.json

Third-party extensions

Third-party extensions can be added from a curated list.



Available Extensions

@nrwl/schematics
[Installed] Nx is an extension for the the Angular CLI implementing the monorepo-style development.

@angular-toolkit/serverless
Serverless support

@angular/elements
Support for using Angular components as custom elements

@angular/material
Angular Material

Figure 10. Add third-party extensions

Appendix B: Commands

We have covered many commands in this book. We can group the commands into two larger groups: those provided by the CLI and those provided by Nx; and we can divide the latter even further by those that can target specific files using `affected` and those that can't or don't need to.

Scripts provided by the CLI

- build
- test
- lint
- e2e
- workspace-schematic

All of these commands can be run using `npm` e.g. `npm run lint`.

Scripts provided by Nx

format

Nx adds support for prettier and to format your files according to the prettier settings. There are two commands:

- `format:write` alters all the files
- `format:check` lists all the files that do not conform to the rules in prettier

These are discussed in **Part 3** of the book.

update

Nx allows you to update it with this command. `update:check` does a check only and informs you if an update is needed.

dep-graph

Nx displays a graphical view of the project dependencies between apps and libs. This is useful to understand which projects may be affected by changes that you make to a lib or app.

The dependency graph is discussed in **Part 3** of the book.

`Affected` commands

Affected commands contain a prefix of `affected:` and target specific files for a given action. Supported actions are below:

- affected:apps
- affected:libs
- affected:build

- affected:e2e
- affected:test
- affected:lint
- affected:dep-graph

We can target the files in the following ways:

- Comparing two git commits (using their SHAs or using branches)
- `--untracked`: all the untracked files on the current working branch
- `--uncommited`: all the files that have been modified since checking out this branch
- specific files: a comma-delimited list of files
- all files: if you want to explicitly use all the files
- `only-failed`: only use the files from the last failed job

These commands are discussed in **Part 4** of the book.

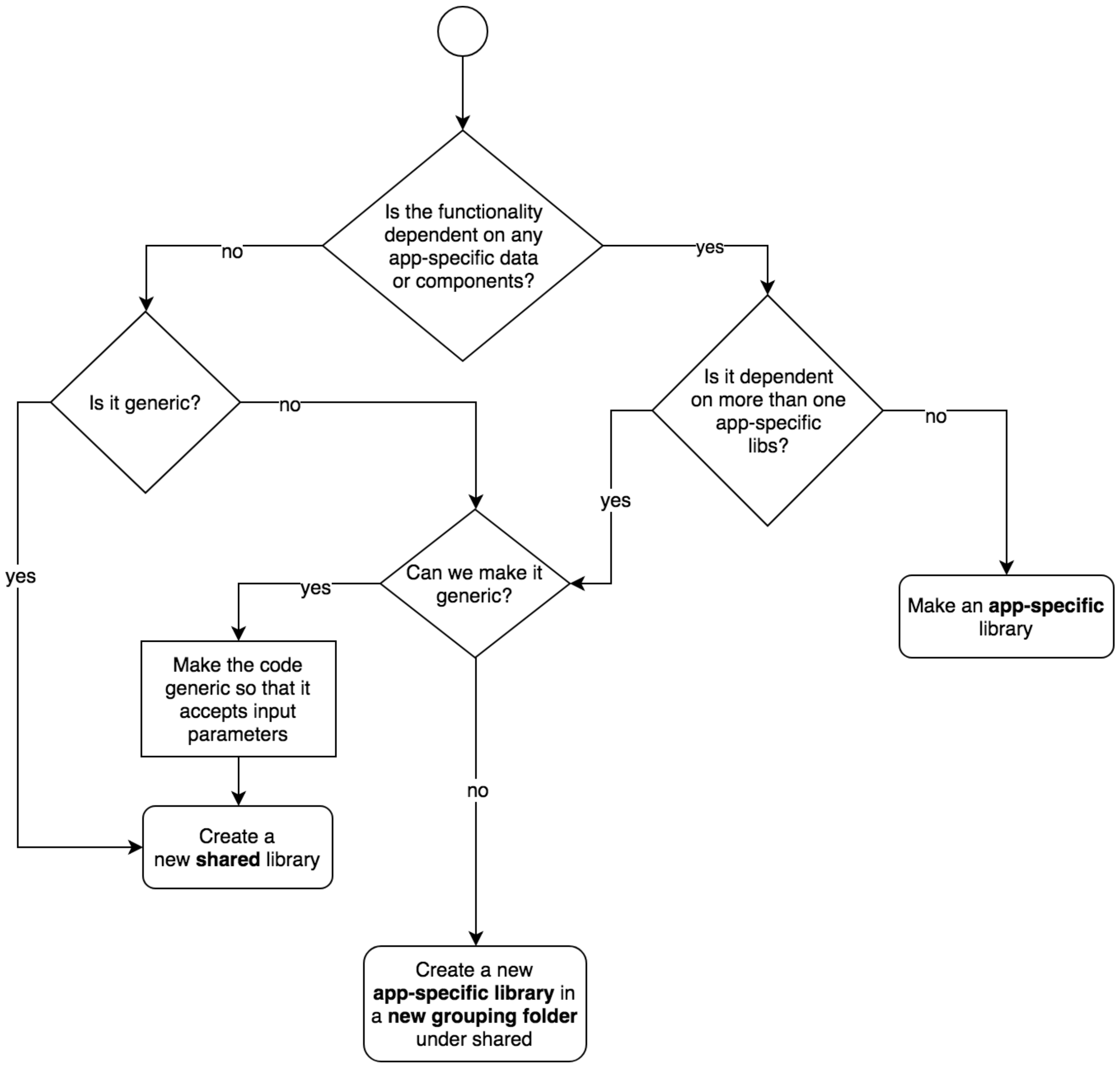
Appendix C: How-tos

Updating Nx

If you created an Nx Workspace using Nx 6.0.0 and then decided to upgrade the version of Nx, the Nx update command can handle modifying the configuration file and the source files as needed to get your workspace configuration to match the requirements of the new version of Nx.

```
npm run update
```

Where should I create my new lib?

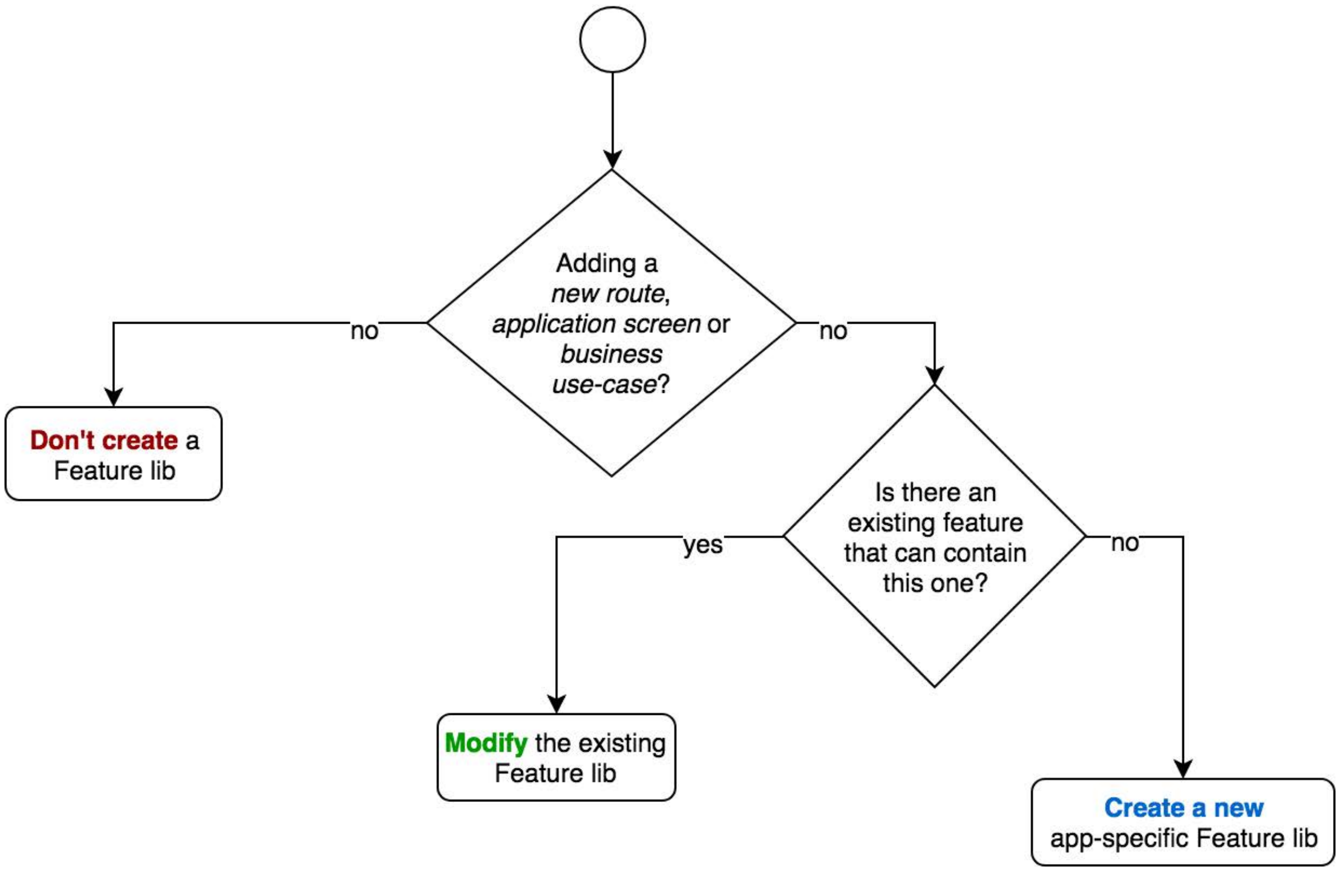

Is the functionality dependent on any app-specific data or components?
no
yes
Is it generic?
no
yes
Is it dependent on more than one app-specific libs?
yes
no
Can we make it generic?
yes
no
Make the code generic so that it accepts input parameters
Create a new **shared** library
Make an **app-specific** library
Create a new **app-specific library** in a **new grouping folder** under shared

Should I reuse or create a feature library?

To answer that question, ask yourself:

- Am I adding a new route?
- Am I adding a new "application screen"?
- Am I working on an app-specific use case?

Create a new feature library or modify an existing feature library. All (most) feature libraries are app-specific, so you need to select a application section directory to put it.


Adding a *new route*, *application screen* or *business use-case*?
no
no
Don't create a Feature lib
Is there an existing feature that can contain this one?
yes
no
Modify the existing Feature lib
Create a new app-specific Feature lib

How do I extract a feature lib from an app?

The process of extracting of an app-specific lib looks like this:

- Run `ng g lib lib-name --directory=app-name` (add `--routing`, `--lazy`, and `--parentModule` if needed)
- Copy the content of the app (everything except index.html, main.ts, polyfills.ts, app.module.ts, app.component.ts and other "global" files) into the lib.
- Update `app.module.ts` to remove all the imports and declarations that are no longer needed. Leave `forRoot` calls there, but we may have to update them (e.g., `StoreModule.forRoot({})`).
- Replace all `forRoot` calls with `forFeature` calls in the feature lib. We have to introduce a prefix for the store, so we have to update the tests and the production code to reflect that.
- Replace `BrowserModule` with `CommonModule` in the feature lib. BrowserModule can only be

imported once, and it should be done at the app level.

- Create an integration test that exercises the new module without requiring the containing app.

Custom Material Modules

When using Angular Material components within custom views, a best-practice encourages developers to create a custom Material module that manages the imports of the specific Angular Material component modules.

This is a special form of a library module; often managed under the path /libs/common/ui/custom-material/.

The example NgModule `custom-material.module.ts` below has been configured to load only the Material Button, Card, Icon, and ProgressBar components.

```
import { NgModule } from '@angular/core';
import {
  MatButtonModule,
  MatCardModule,
  MatIconModule,
  MatProgressBarModule
} from '@angular/material';

import { BidiModule } from '@angular/cdk/bidi';

@NgModule({
  exports: [
      MatButtonModule,
      MatCardModule,
      MatIconModule,
      MatProgressBarModule,
      BidiModule
  ]
})
export class CustomMaterialModule {}
```

Any library or app using these components [in their templates] simply imports the **CustomMaterialModule.** When additional components are needed, only this `CustomMaterialModule` needs to be modified to add the required imports and exports.

Get in touch to scale up your development capacity

Reimagine how you solve your toughest challenges and build an amazing application with the greatest development partner, best tools and latest practices. Build software for the future with Nrwl. Our modern open-source development tools and practices enable software teams to achieve better collaboration and build long-term team capabilities and success. Contact us, at hello@nrwl.io about your project or fill out the form at https://nrwl.io/services. We look forward to hearing from you!