Enterprise Angular Monorepo Patterns

Book v0.1 | December 1, 2018

Nrwl

Enterprise Angular Monorepo Patterns

Authors: Victor Savkin and Nitin Vericherla
with contribution from Thomas Burleson

Nrwl

For more information visit nrwl.io

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

```
function() {
  console.log('hello world');
}
```

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

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).

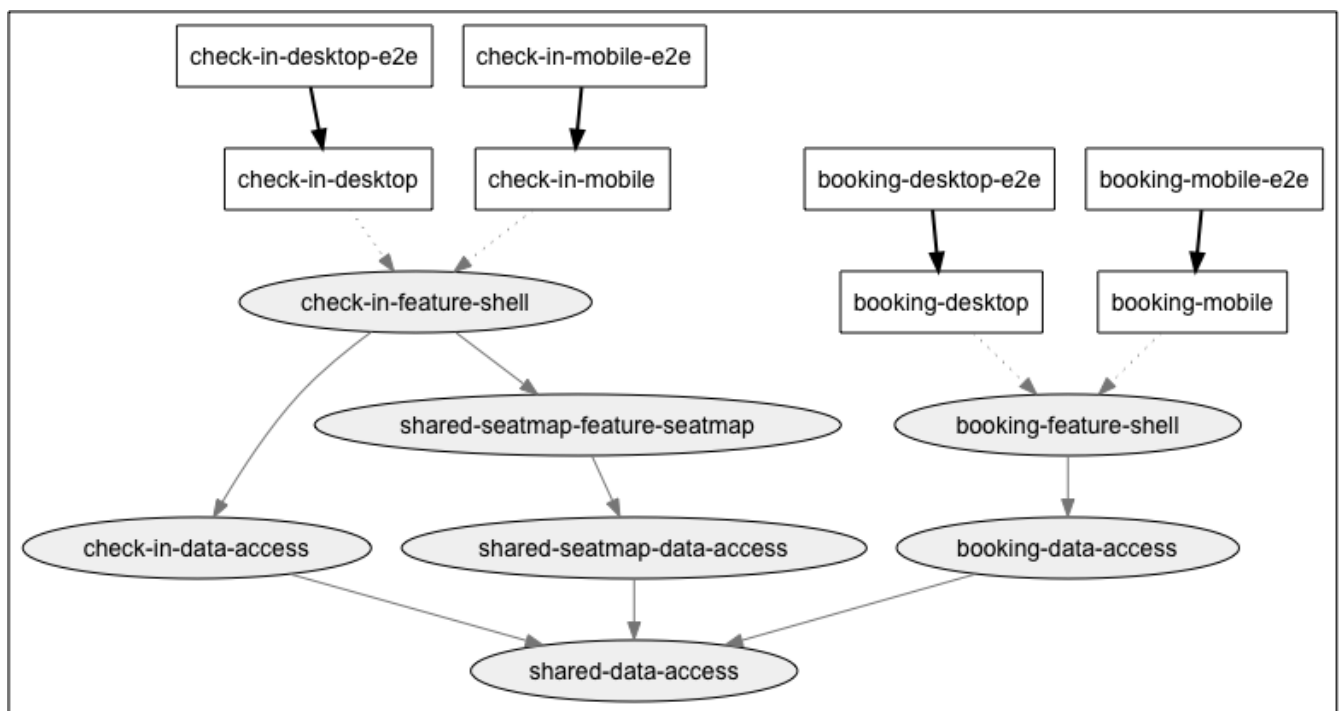Its dependency graph of the code looks like this:

Figure 1. The example repo structure and dependencies

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 these 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 Trunk-based Development 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. See Single Version Policy in Part 5 of the book.

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:

Folder tree of apps

```
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.

Libs have a well-defined public API in the `index.ts` file. They might also come with a README and have a designated code owner (see Part 5 for a discussion on code owners).

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
```

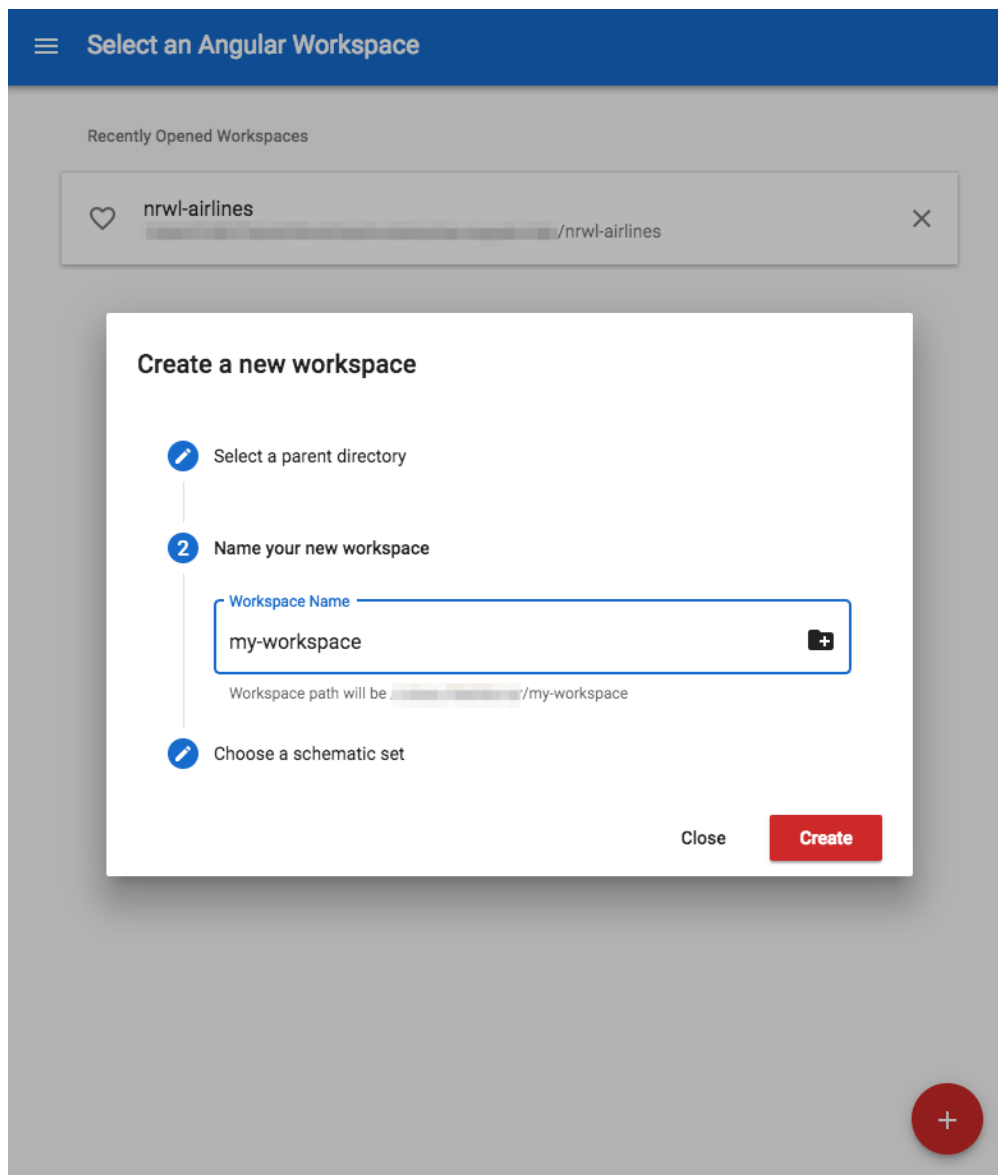Finally, you can create an Nx Workspace using Angular Console.

Figure 2. 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.

Default workspace contents

```
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:

Output when using the --help option

```
  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).

Figure 3. 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.

- We looked at some reasons for adopting a monorepo mindset: 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. We look at

monorepos in detail in Part 5 of the book.

- We covered the basic terminology and foundational concepts of Nx: a workspace, app and lib.

- We discovered how to create workspaces, apps and libs using the command line and Angular Console.

- We looked at two different ways to get help with the commands: the `--help` command-lne option, and opening the project in Angular Console to look at the visual descriptions for the options.

With the foundations in place we can take a deep-dive into libraries and examine how we can structure our code to be composable, reusable, and easy to navigate.

Part 2: Organizing Code With Libraries

In a Nutshell

Applications tend to grow in size and complexity over time and this is multiplied when there are multiple apps that need to share code. Large organizations and business units need to consider ways to break applications down in both size and complexity and to reuse parts of applications so that there is consistency in how they are implemented.

Libraries, which are simply a collection of related files that perform a certain task, help in developing in a modular way. They come with a well-defined public API in their `index.ts` file that dictates how they are to be used. Developers should also include a README to document the library's behaviour and have a designated code owner (see Code Ownership in Part 5 of the book for more information).

In an Nx workspace libs require classifiers to describe their contents and intended purpose. These classifiers help to organize the libraries and to provide a way to visually distinguish them. There are two basic classifiers: "scope" and "type". There can be additional classifiers in order to help with different scenarios in a particular organization for example, "platform".

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` as above, and have a prefix by its type. Refer to the section Enforce Restrictions In Library Dependencies in Part 3 for more information on how tags are used.

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. This allows a developer to work on related files without needing to traverse the folder tree often.

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 specific **business use cases** or pages in an application.
- **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. It also includes all the code related to State management.
- **Utility libraries**: A utility library contains **common utilities and services** used by many

libraries and applications.

Before we look at all of the library types, let's quickly cover some basic Flux-inspired fundamentals about the types of components so that we can understand Smart and Presentational components.

Aside: Smart Vs. Presentational Components

There are two types of components when we build applications via a uni-directional data flow: those that can communicate with the rest of the application by receiving and sending data (known as "smart" components), and those that only receive data (known as "presentational" or "dumb" components).

Smart Components

These components manage or delegate business logic and use DI to inject services. They are able to send out updates to the rest of the application by dispatching actions. They have child instances of presentational components: the smart components pipe data to the child presentational components and respond to events emitted by the child components to then handle as needed by the application.

Presentational Components

These components have very little or no business logic. They **only** rely on *Inputs* and *Outputs* to communicate with the outside world. Their only purpose is to render data and to accept user input - but not to process the input. Rather, these components emit events via *Outputs* to parent components which know how to handle them.

They are highly reusable and are the easiest to test and may be fully generic/domain-agnostic (e.g,. data-table) or have a domain-context (e.g, log-book-table).

Now that we are aware of the terminology for Smart and Presentational components, let's look at the types of libraries that contain them. Feature libraries contain smart components and `ui` libraries contain presentational components.

Feature Libraries

What is it?

A feature library contains a set of files that configure a business use case or a page in an application. These libraries contain an ngModule that specifies how this part of the application behaves (it may contain a slice of the Store, handle its own routing within a particular application section, and can be lazy loaded into an app).

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

Naming Convention

`feature` (if nested) or `feature-*` (e.g., `feature-shell`).

```
libs/
  booking/
    feature-shell/ ①
      src/
        index.ts
        lib/
          booking-feature-shell.module.ts
```

① `feature-shell` is the app-specific feature library (in this case, the "booking" app).

Example 1. Let's consider a use-case for a feature component

> In our example application, nrwl-airlines would like to start working on the UI for the booking application. We want to provide three screens to the user: flight search, passenger information, and a seatmap.

Let's consider first that we put the routing for these directly inside the `booking/desktop` application.

```
export routes = [
  {
    path: '',
    pathMatch: 'full',
    component: 'FlightSearchComponent'
  },
  {
    path: '/passenger',
    pathMatch: 'full',
    loadChildren: '@nrwl-airlines/booking/feature-passenger-
info#BookingFeaturePassengerInfoModule'
  },
  {
    path: '/seatmap',
    pathMatch: 'full',
    loadChildren: '@nrwl-airlines/shared/seatmap/feature-seat-
listing#SharedSeatmapFeatureSeatListingModule'
  }
]
```

If this routing were to be placed into `apps/booking/desktop`, we would need to duplicate this in `apps/booking/mobile`, and maintain the two to be in sync going forward. In order to keep this DRY, we would want to extract the routing out to a shared file.

We can move the routing logic out to a lib. This lib would contain the routing structure for `booking` and also carry out any initialization. The module in this lib can also be lazy-loaded into both parent applications `booking/desktop` and `booking/mobile`.

An appropriate naming convention is `BookingFeatureShellModule` located in `libs/booking/feature-shell` as shown above the example. We now have four feature libs: `booking/feature-shell`, `booking/feature-flight-search`, `booking/feature-passenger-info`, and `shared/seatmap/feature-seat-listing`. Each of them correspond to a screen in the application.

An example feature lib module

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

import { FlightSearchComponent } from './components/flight-search/flight-
search.component';

@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild([ ①
      {
        path: '',
        component: FlightSearchComponent,
        pathMatch: 'full'
      },
      {
        path: '/passenger',
        pathMatch: 'full',
        loadChildren: '@nrwl-airlines/booking/feature-passenger-
info#BookingFeaturePassengerInfoModule'
      },
      {
        path: '/seatmap',
        pathMatch: 'full',
        loadChildren: '@nrwl-airlines/shared/seatmap/feature-seat-
listing#SharedSeatmapFeatureSeatListingModule'
      }
    ])
  ],
  declarations: [FlightSearchComponent]

})
export class BookingFeatureShellModule {} ②
```

① Note the use of `.forChild`. We would never use a `.forRoot` within a feature lib - that is for apps only.

② The path to this module would be `libs/booking/feature-shell/src/booking-feature-shell.module.ts`

See the Command Line Options section below for discussion of command line options available for generating lazy-loading boilerplate for the feature libraries.

Now that we have seen where to place smart components, let's look at where to place the presentational components.

UI Libraries

What is it?

A UI library is a collection of related **presentational** components. There are generally no services injected into these components (all of the data they need should come from *Inputs*).

For a discussion of whether to make a ui library app-specific and shared, please refer to Appendix C ("where should I create my new lib?").

Naming Convention

ui (if nested) or ui-* (e.g., ui-buttons)

An example ui lib module

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

@NgModule({
  imports: [CommonModule],
  declarations: [ConfirmButtonComponent],
  exports: [ConfirmButtonComponent]
})
export class CommonUiButtonsModule {} ①
```

① The path to this module would be libs/common/ui-buttons/src/common-ui-buttons.module.ts

Outside of smart and dumb components, there are also libraries for data-access and utilities. Let's examine what they contain next.

Data-access Libraries

What is it?

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 (by convention, they can be grouped under a +state folder under src/lib).

Naming Convention

data-access (if nested) or data-access-* (e.g. data-access-seatmap)

An example data-access lib 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, { ①
      initialState: customersInitialState
    }),
    EffectsModule.forFeature([CustomersEffects]) ①
  ],
  providers: [CustomersEffects]
})
export class CustomersDataAccessModule {}
```

① Again, notice that we only use `.forFeature` in libs. Any `.forRoot` calls should be in apps.

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

> If a library is shared, document it! See Documenting Libraries for details.

Utility Libraries

What is it?

A utility contains common utilities/services used by many libraries. Often there is no ngModule and the library is simply a collection of utilities or pure functions.

Naming Convention

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

An example ui lib module

libs/shared/util-formatting

```
export { formatDate, formatTime } from './src/format-date-fns';
export { formatCurrency } from './src/format-currency';
```

Now that we've covered the different types of libraries, we can cover two important concepts: grouping libraries together in a nested hierarchy and encouraging code reuse with shared libraries.

Grouping Folders

What is it?

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.

The purpose of these folders is to help with organizing by scope. We recommend grouping libraries together which are (usually) updated together. It helps with minimizing the amount of time a developer spends navigating the folder tree to find the right file.

```
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
```

Sharing Libraries

One of the main advantages of using a monorepo is that there is more visibility into code that can be reused across many different applications. Shared libraries are a great way to save the developer time and effort by reusing a solution to a common problem.

Let's consider our reference monorepo. The `shared-data-access` library contains the code needed to communicate with the back-end (For example, the URL prefix). We know that this would be the same for all libs; therefore, we should place this in the shared lib and properly document it so that all projects can use it instead of writing their own versions.

```
libs/
  booking/
    data-access/            <---- app-specific library

  shared/
    data-access/            <---- shared library

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

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

We are now in a position to create libs in our workspace. Let's look at some important considerations when creating new libraries.

Notes On Using Libraries

The Barrel File

When we use Nx and @nrwl/schematics to create a library, there are a few steps that are executed. The most important feature is the creation of the barrel file.

Here are the files generated when creating a lib called `data-access` in `shared`:

```
jest.config.js ①
src/
  index.ts ②
  lib/
    shared-data-access.module.spec.ts
    shared-data-access.module.ts
  test-setup.ts
tsconfig.lib.json
tsconfig.spec.json
tslint.json
```

① `jest` was selected as the testing framework for this lib.

② This is the barrel file.

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.

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

Configuration of the path alias for our lib

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

The above allows us to write `import * from '@nrwl-airlines/shared/data-access';` anywhere in our workspace and instructs typescript to import from the barrel file (though we should import specific entities from the barrel instead of *).

Each app and lib that we create using the CLI 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.

Let's look at when we should use this alias in the workspace and when we should not.

Relative paths vs. using the workspace alias

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 violations to the above.

Once our lib is created, one of the main things that we need to do is to inform other developers about what this library is for, how to use it, and what restrictions there might be to using it.

Documenting Libraries

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

- code owner
- the library's usage visualization (dependency-graph)
- the dependency-constraints (which apps or libraries are authorized to USE this library)
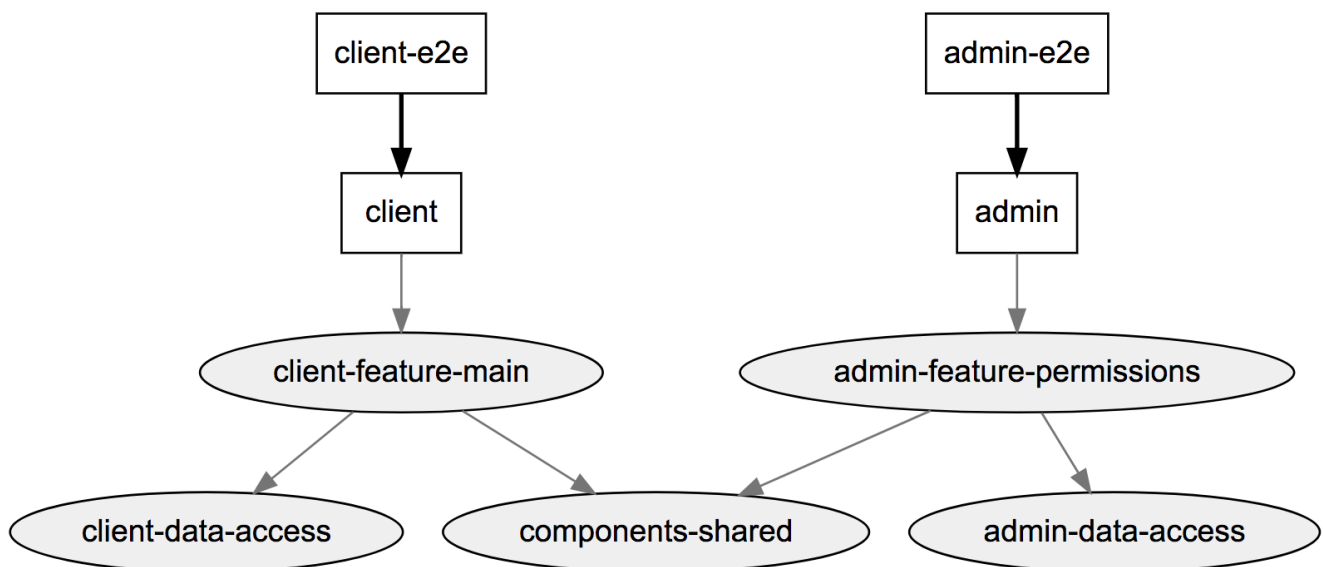

Figure 4. An example visualization

You may have noticed that the module name for the libs follow a particular pattern. Let's see why that is the case.

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.

Let's now look at some of the options that Nx and the CLI provide when creating a library.

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.

Summary

In this section we looked at libraries in depth.

- Libraries offer a way to modularize code and to make it easy to share code across applications in the workspace.

- Libraries can be classified with **scope**, **type**, **platform** and other classifiers to organize them.

- We denote **scope** with folder structure and **type** with prefixes. Each library contains all of the classifiers to aid in restricting their usage to only compatible libs.

- There are four libraries types in a typical Nx workspace: `feature`, `ui`, `data-access`, and `util`.

- Grouping Folders allow for a hierarchy of **scope**.

- The barrel file for a library defines its public API and is the most important aspect of the library. The public API must be clearly thought out, explicit, and only include what we want other libs to use.

- We covered some of the command-line options when creating libs: `directory`, `routing`, `parent-module`, `publishable`, and `tags`.

Part 3: Enforcing Quality And Consistency

Nx contains a few tools to help with maintaining consistency across the code-base and to ensure the code quality. Let's look in detail at the following tools:

- Enforce a **Single Version Policy**: A single package.json ensures that all apps and libs use the same dependency versions.

- Ensure **consistency in code formatting**: Build tools like `prettier` assist with ensuring consistency in code formatting (one immediate benefit is the removal of whitespace and formatting diffs in PRs).

- Enforce **restrictions in library dependencies**: It allows for configurable boundary enforcement between libraries e.g. a `util` library not being able to depend on a `feature` library (or even an app).

- Enforce **consistency in code generation**: Nx allows you to generate workspace schematics to handle code generation consistently.

- Enforce **consistency and reliability in network communications within applications**: Nx includes a DataPersistence library to help with managing interactions with the back-end in a consistent and reliable way.

Enforce a Single Version Policy

Google enforces a constraint that there is a single Angular version across all projects. The biggest benefit to them for doing this is that they are able to test new versions of Angular internally to iron out the issues; however, there are other benefits to be realized:

- It eases the sharing of code across the organization. Having the same version of dependencies removes the overhead of checking to see whether a pre-existing piece of code is going to work when integrated.

- It ensures that no security vulnerabilities remain unfixed.

- It helps developers move between projects.

There can be some downsides as well if we are also using trunk-based development (See Part 5 for a discussion of trunk-based development).

- It can make updating of dependency versions take more effort if there are many changes to be made. This can be mitigated a little by ensuring that the organization updates regularly (even for minor or patch versions). A large gap in versions would involve many more changes.

- It makes it impossible for a certain project to use a different version of a dependency. This makes prototyping using a newer dependency version difficult (the code can't be merged unless this change is organization-wide).

The single version policy applies to two facets of the workspace:

1. Single package.json: The dependencies listed in `package.json` apply across the entire workspace if there is only one `package.json`.

2. All the lib code in the same repo: The lib code is within the same repository and so there is no way to have previous versions of the lib code unless we are not up-to-date with our target branch.

There are alternatives to having a single version policy for the two facets described:

Single package.json

It is possible to use Lerna or yarn workspaces to have a separate package.json for each app and to manage them centrally. Due to the overhead of managing these, Nx doesn't support multiple package.json files and assumes that there is a single package.json file in the workspace.

All the lib code in the same repo

It is possible to use git submodules to simulate different versions of libs (with each lib in its own repo). Nx would work when using git submodules, but this is discouraged due to the complexity it introduces and the increased likelihood of mismatched SHAs for the submodules in the workspace.

Nx, being an opinionated tool, prescribes a single version policy by containing a single package.json and by the virtue of being only one repo (and not using git submodules).

Ensure Consistency In 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.

Formatting using automation has a very important benefit: diffs only contain actual code changes. A large diff that contains whitespace changes makes it difficult to discern the actual code changes from the formatting changes. This is worse if developers have different local settings and keep committing the same changes back-and-forth.

Enforce Restrictions In Library Dependencies

An Nx workspace is a large dependency graph. Apps depend on libs that in turn depend on other libs, and so on. When there are many teams and many libs it can lead to confusion about which libs should depend on others. Let's consider an example scenario:

> Alice wants to add an HTTP interceptor to a lib that is in `seatmap/util`. However, this interceptor needs a value from the Store. Should the `util` lib depend on the `data-access` lib under seatmap?

The answer to this is that according to how Nx defines a `util` library this would be incorrect - Alice should move the interceptor into the `data-access` folder so that it can refer to the value from the Store.

However, there is nothing in the workspace by default that prevents Alice from leaving the interceptor in `util` and importing from `data-access`. This issue might not come to light until a circular dependency is discovered.

Let's consider some of the ways that we can see or prevent this.

Analyzing And Visualizing the Dependency Graph

Nx uses advanced code analysis to build a dependency graph of all the apps and libs in the workspace and how they depend on each another.

You can visualize the entire workspace by running `npm run dep-graph`.
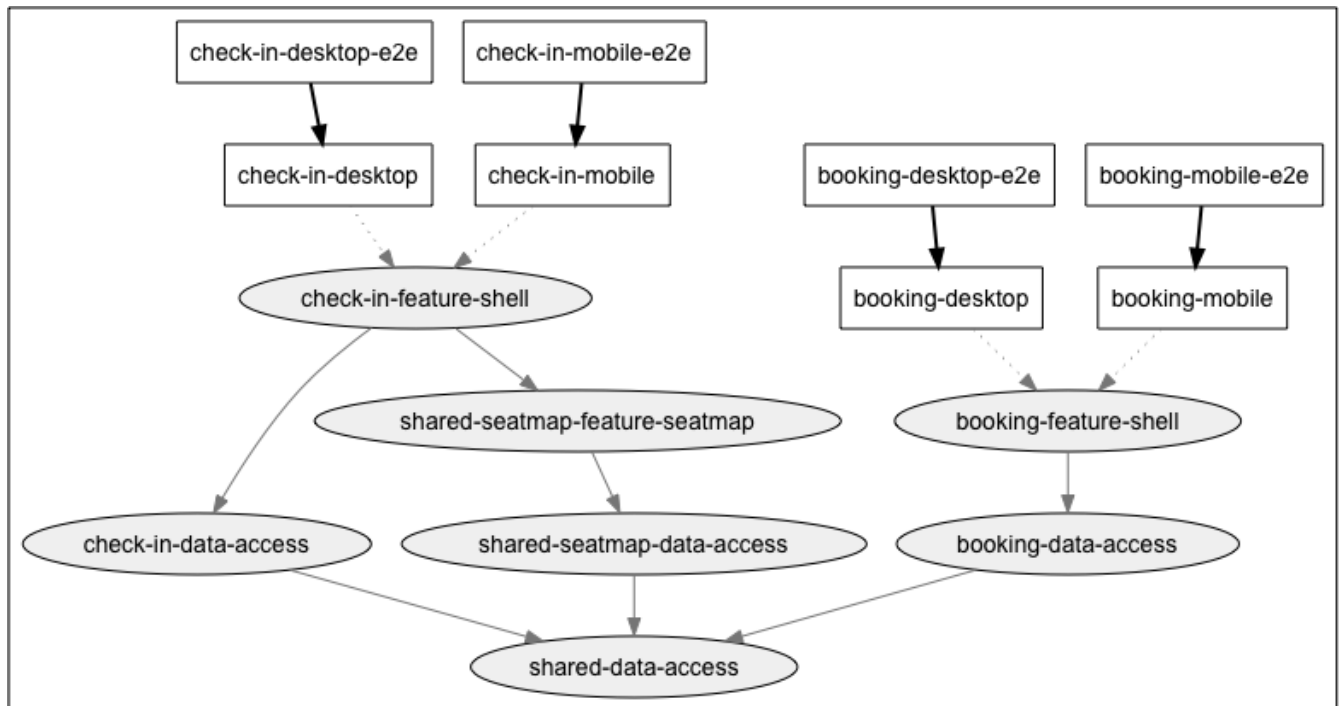
Figure 5. The dependency graph of the whole workspace

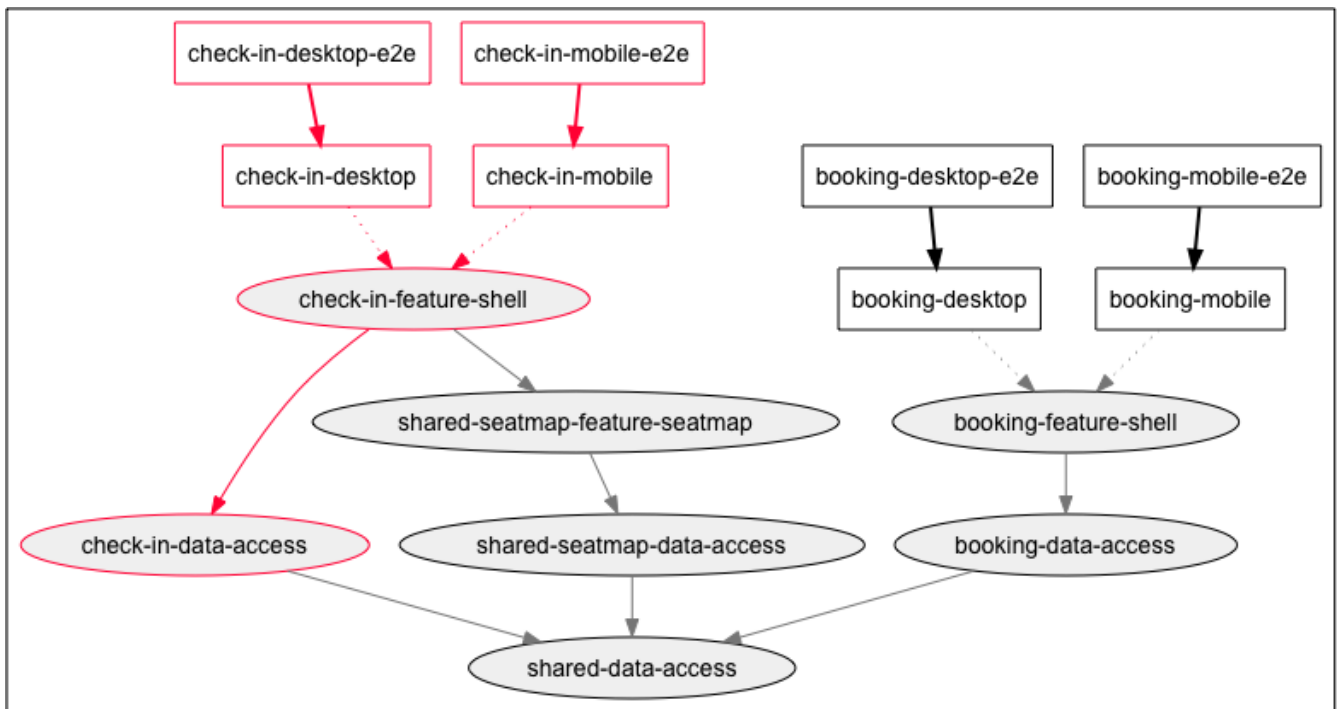You can also visualize what is affected by your change, by using the `affected:dep-graph` command.

Figure 6. The dependency graph when we make a change to check-in-data-access

[Part 4](#) of the book looks 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.

The dep-graph highlights the affected apps and libs and traces 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.

The Angular Console features a more interactive visualization: you can navigate specific apps/libs, choose to test affected or selected, and choose different branches.
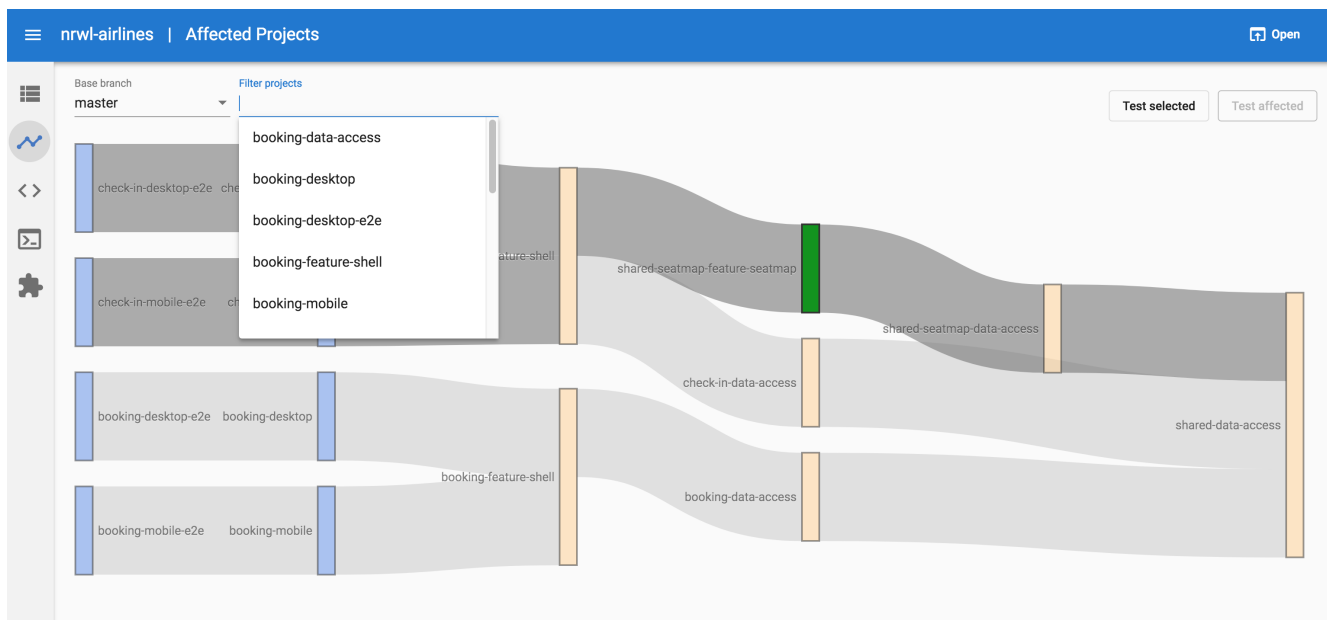
Figure 7. Angular Console features an interactive dependency graph

Now the we know how to visualize the dependencies, let's take a look at how to help Alice.

Enforcing Constraints In the Workspace

Nx uses code analysis 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.

Nx comes with some defaults that should apply to all workspaces, and allows for custom rules as needed by the organization.

Universal Constraints

The following invariants should hold true for all workspaces:

1. a **lib** cannot depend on an **app**
2. a project cannot have circular dependencies
3. 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.

Nx schematics also include this as a default

```
"nx-enforce-module-boundaries": [
  true, ①
  {
    "allow": [], ②
    "depConstraints": [
      { "sourceTag": "*", "onlyDependOnLibsWithTags": ["*"] } ③
    ]
  }
]
```

① This is a flag that enables the rule.

② This is a whitelist that bypasses the checks in `depConstraints` - see the "Exceptions" section below.

③ This is where we specify the explicit dependency rules based on the tags in our workspace libs.

Imposing Your Own Constraints On Library Dependencies

Some examples of recommended workspace rules are below:

1. An **app-specific** library cannot depend on a lib from another app (e.g., "booking/*" can only depend on libs from "booking/*" or shared libs).

2. A **shared library** cannot depend on an **app-specific** lib (e.g., "shared-data-access" cannot depend on "booking-data-access").

3. A **ui library** cannot depend on a feature or data-access library.

4. A **util library** can only depend on other `util` libraries.

5. A **data-access library** cannot depend on a feature or ui library.

These constraints are enforced in the following ways:

- IDEs and editors display an error if you are trying to violate these rules (if they are set up to recognize tslint rules)

- CI will fail if we use the lint rule in the pipeline

Constraints are defined using the rules in `tslint.json` and tags in `nx.json`:

```
ng g lib feature-destination --directory=booking --tags=scope:booking,type: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 been associated with each library, tsLint rules can be defined to configure constraints:

```
"nx-enforce-module-boundaries": [
  true,
  {
    "allow": [], ①
    "depConstraints": [
      {
        "sourceTag": "scope:shared", ②
        "onlyDependOnLibsWithTags": ["scope:shared"]
      },
      {
        "sourceTag": "scope:booking", ③
        "onlyDependOnLibsWithTags": ["scope:booking", "scope:shared"]
      },
      {
        "sourceTag": "type:util", ④
        "onlyDependOnLibsWithTags": ["type:util"]
      }
    ]
  }
]
```

① The `allow` parameter allows us to specify a whitelist that we can import from regardless of the other rules in `depConstraints`.

② A lib tagged `scope:shared` can only import from other libs with tag `scope:shared`.

③ A lib tagged `scope:booking` can only import from libs tagged with either `scope:booking` or `scope:shared`.

④ 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 a lib from `check-in` into `booking` (violates rule 2 above).
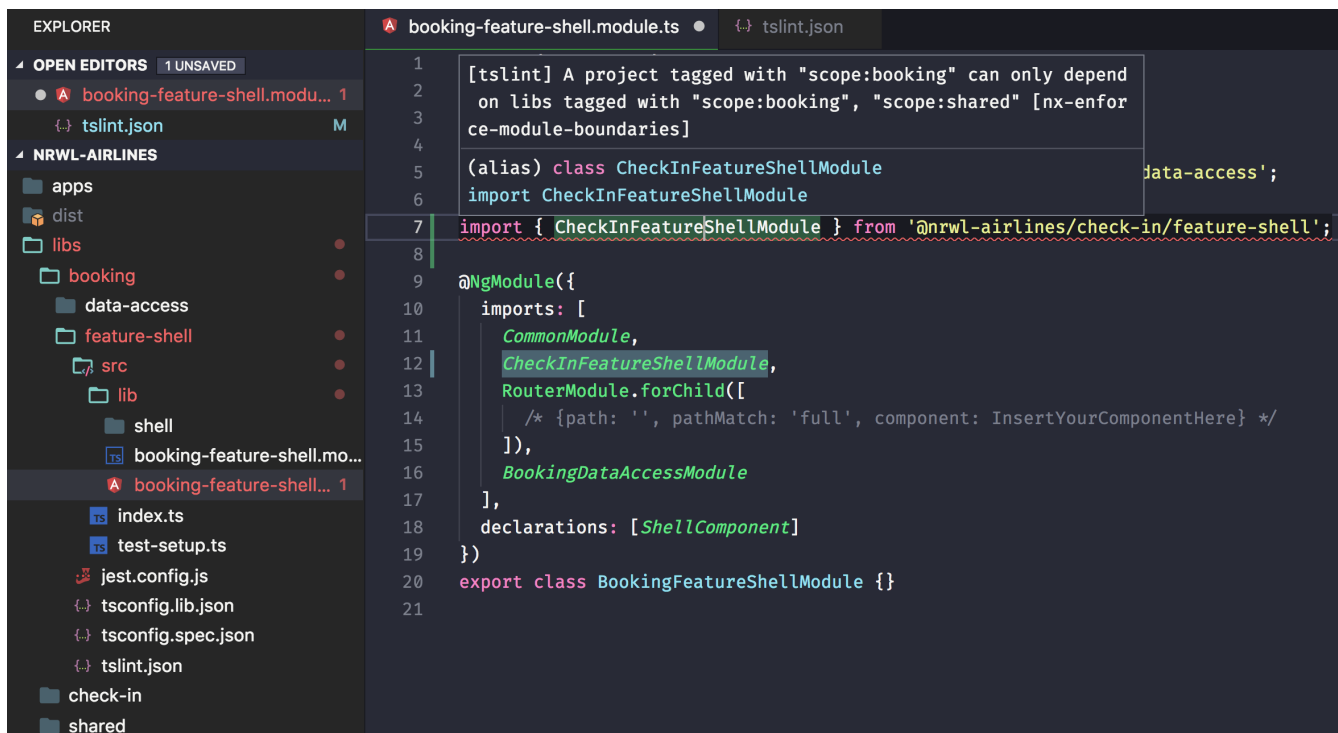
Figure 8. Error thrown when we try to add `CheckInFeatureShellModule` *into* `BookingFeatureShellModule`

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.

By default Nx adds the following rule, which allows any library to import from any other library:

```
"depConstraints": [
  { "sourceTag": "*", "onlyDependOnLibsWithTags": ["*"] }
]
```

We can disable this rule in order to force that all libs are explicit about their dependencies. Doing so throws the following error:
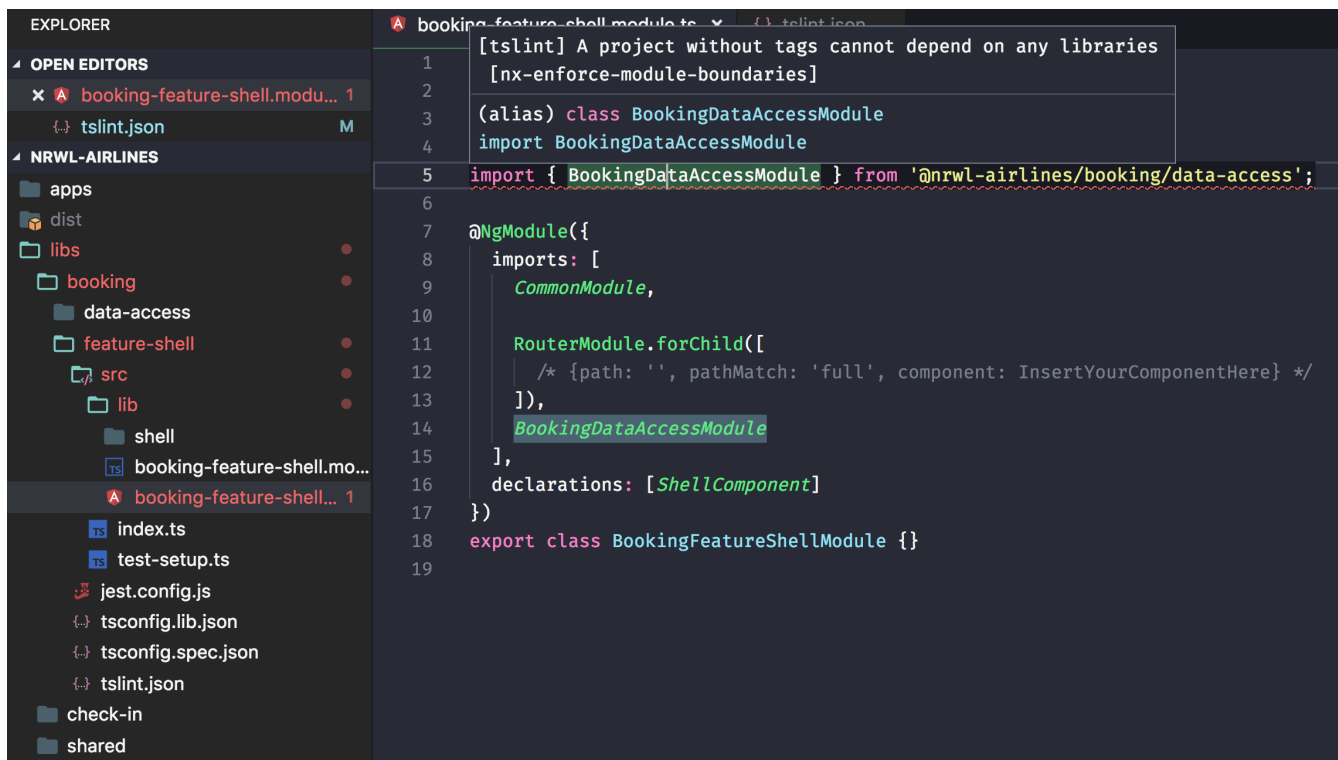
Figure 9. Error when we remove the wildcard rule

We've looked at how to enforce rules about depending on other apps and libs. Some files exist in the workspace and would not be able to be covered. Some examples are `angular.json`, `tsconfig.json`, etc. Let's look at how we can handle these as well.

Implicit Dependencies

Nx has support for implicit dependencies: it covers anything that is not in a lib or an app. These are defined in nx.json:

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

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.

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

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": [
    ]
  }
]
```

Using Workspace Schematics

Schematics offer a way to execute scripts against your workspace via the CLI. These run against the files and directories in your workspace and can add, modify, copy, move, or remove them as needed. They have many uses; some of which we outline below.

- Code scaffolding: We can create schematics to generate code for when we need a new type of lib, new ngrx entity, adding ngrx scaffolding to an existing lib, etc.

- Updating our projects: We can run scripts to update our workspace in specific ways as needed.

- Making automated changes across the entire workspace: For example, we might have a need to

perform a sequence of actions across all libs: create a folder called `+state` and move all of the state-related files (actions, reducers, selectors and facades) into that folder, updating the barrel file. Using a schematic would help here instead of having to do it by hand.

One of the great benefits of using a schematic is that it performs the changes to an in-memory representation rather than executing it directly against the file system. If everything looks good we can run the schematic without the `dry-run` to persist the changes.

Nx comes with some built-in schematics.

Built-in Nx schematics

The following schematics are found in the `@nrwl/schematics` collection.

- **ngrx**: This schematic scaffolds ngrx for a project and creates example reducers, actions, selectors, and an optional facade.

- **node**: This schematic scaffolds a choice of blank or Express node application and wires it up to be run alongside the Angular applications.

- **jest**: The collection contains two schematics for jest: adding jest support to the workspace, and adding jest as a test runner for a lib.

- **cypress**: This schematic sets up the workspace to use cypress for e2e testing.

- **downgrade module**: This schematic modifies your project to use the ngDowngradeModule.

- **new**: The collection contains schematics to create and bootstrap a new workspace, new app, or new lib.

- **add**: This schematic lets you add other schematics to your workspace.

- **upgrade**: This schematic is used to upgrade the versions of Nx and related peer dependencies (Angular, RxJs, etc). This is an internal schematic and is used by the `upgrade` npm script.

We might need specific schematics for our organization. Nx can help with those as well.

Creating a custom schematic

Let's take an example scenario: we want to promote a pattern of encapsulating NgRx-related code into data-access libraries going forward.

In other words, we want to do the following:

- Check that the lib name starts with `data-access-`.
- Invoke the `ngrx` schematic with this new lib name and pipe the other options to it.

Start by generating a new workspace schematic.

```
ng g workspace-schematic data-access-lib
```

This creates the following:

Output of creating a new schematic

```
tools/
  schematics/
    data-access-lib/
      index.ts ①
      schema.json ②
```

① This is the main file of the schematic.

② This is the file that contains a JSON schema for the arguments we receive from the CLI (if you want to use it for input validation in the schematic).

The `index.ts` file has the following contents

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

export default function(schema: any): Rule {
  return chain([
    externalSchematic('@nrwl/schematics', 'lib', { ①
      name: schema.name
    })
  ]);
}
```

① The provided example schematic calls the `lib` generator from `@nrwl/schematics`. This should be replaced with your actual implementation.

Let's provide our own implementation.

```typescript
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-')) { ①
    throw new Error(`Data-access lib names should start with 'data-access-'`);
  }

  const stateName = schema.name.replace('data-access-', '');

  return chain([ ②
    externalSchematic('@nrwl/schematics', 'lib', { ③
      name: schema.name,
      tags: 'type:data-access'
    }),
    externalSchematic('@nrwl/schematics', 'ngrx', { ④
      name: stateName, ⑤
      module: path.join(
        'libs',
        schema.name,
        'src',
        'lib',
        `${schema.name}.module.ts`
      )
    })
  ]);
}
```
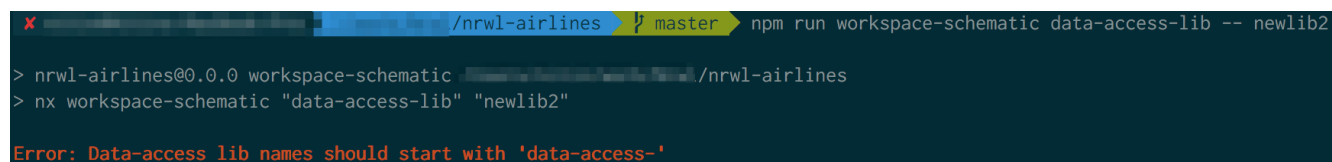
① Here is the check for the `data-access` prefix.

② `chain` allows us to provide multiple schematics to execute in order.

③ First we create a new lib using `@nrwl/schematics`.

④ Then we create the ngrx scaffolding again with `@nrwl/schematics`.

⑤ We use the lib name (without the `data-access` prefix) as the state name in ngrx.

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

```
npm run workspace-schematic data-access-lib -- data-access-newlib2
```

This command has multiple parts to it. `npm run workspace-schematic` accepts an argument for the name of the schematic. In our case this is `data-access-lib`. The rest of the arguments follow the `--` to bypass npm and be sent to the invoked command (`data-access-newlib2`, which is the `name` of our new lib).

Figure 10. Output of running our new schematic without specifying the prefix

```
                                            /nrwl-airlines   master    npm run workspace-schematic data-access-lib -- data-access-newlib2

> nrwl-airlines@0.0.0 workspace-schematic                    /nrwl-airlines
> nx workspace-schematic "data-access-lib" "data-access-newlib2"

create libs/data-access-newlib2/karma.conf.js (496 bytes)
create libs/data-access-newlib2/tsconfig.lib.json (752 bytes)
create libs/data-access-newlib2/tsconfig.spec.json (271 bytes)
create libs/data-access-newlib2/tslint.json (267 bytes)
create libs/data-access-newlib2/src/index.ts (161 bytes)
create libs/data-access-newlib2/src/test.ts (700 bytes)
create libs/data-access-newlib2/src/lib/data-access-newlib2.module.ts (696 bytes)
create libs/data-access-newlib2/src/lib/data-access-newlib2.module.spec.ts (483 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.actions.ts (805 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.effects.spec.ts (1171 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.effects.ts (871 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.reducer.spec.ts (1150 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.reducer.ts (1183 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.selectors.spec.ts (1619 bytes)
create libs/data-access-newlib2/src/lib/+state/newlib2.selectors.ts (1010 bytes)
update angular.json (23005 bytes)
update nx.json (1978 bytes)
update tsconfig.json (1292 bytes)
update package.json (2622 bytes)

> nrwl-airlines@0.0.0 format                        /nrwl-airlines
> nx format:write "--untracked" "--quiet"

libs/data-access-newlib2/src/index.ts 178ms
libs/data-access-newlib2/src/lib/+state/newlib2.actions.ts 25ms
libs/data-access-newlib2/src/lib/+state/newlib2.effects.spec.ts 28ms
libs/data-access-newlib2/src/lib/+state/newlib2.effects.ts 14ms
libs/data-access-newlib2/src/lib/+state/newlib2.reducer.spec.ts 36ms
libs/data-access-newlib2/src/lib/+state/newlib2.reducer.ts 27ms
libs/data-access-newlib2/src/lib/+state/newlib2.selectors.spec.ts 35ms
libs/data-access-newlib2/src/lib/+state/newlib2.selectors.ts 76ms
libs/data-access-newlib2/src/lib/data-access-newlib2.module.spec.ts 37ms
libs/data-access-newlib2/src/lib/data-access-newlib2.module.ts 25ms
libs/data-access-newlib2/src/test.ts 14ms

> nrwl-airlines@0.0.0 format                        /nrwl-airlines
> nx format:write "--untracked" "--quiet"

libs/data-access-newlib2/src/index.ts 192ms
libs/data-access-newlib2/src/lib/+state/newlib2.actions.ts 23ms
libs/data-access-newlib2/src/lib/+state/newlib2.effects.spec.ts 41ms
libs/data-access-newlib2/src/lib/+state/newlib2.effects.ts 13ms
libs/data-access-newlib2/src/lib/+state/newlib2.reducer.spec.ts 35ms
libs/data-access-newlib2/src/lib/+state/newlib2.reducer.ts 16ms
libs/data-access-newlib2/src/lib/+state/newlib2.selectors.spec.ts 33ms
libs/data-access-newlib2/src/lib/+state/newlib2.selectors.ts 25ms
libs/data-access-newlib2/src/lib/data-access-newlib2.module.spec.ts 15ms
libs/data-access-newlib2/src/lib/data-access-newlib2.module.ts 15ms
libs/data-access-newlib2/src/test.ts 14ms
npm WARN @ngrx/effects@6.1.0 requires a peer of @angular/core@^6.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @ngrx/router-store@6.1.0 requires a peer of @angular/common@^6.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @ngrx/router-store@6.1.0 requires a peer of @angular/core@^6.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @ngrx/router-store@6.1.0 requires a peer of @angular/router@^6.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @ngrx/store@6.1.0 requires a peer of @angular/core@^6.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @ngrx/entity@6.1.0 requires a peer of @angular/core@^6.0.0 but none is installed. You must install peer dependencies yourself.

audited 70587 packages in 7.683s
found 1 critical severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details
```

Figure 11. Output of running our new schematic successfully

We can now create schematics that can help with making our code consistent across all projects and to abstract away some of the boilerplate.

Let's take a look at how we can apply consistency within our Angular applications when making network calls.

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.

Nx DataPersistence is a set of helper functions that enables the developer to manage the synchronization of state with an explicit strategy and to handle error state.

Refer to Using NgRx 4 to Manage State in Angular Applications for a detailed look at the state problem DataPersistence is solving.

Why Use the DataPersistence Library?

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

1. It fetches the state from the Store to retrieve one or more values during the asynchronous operation

2. It forces us to choose between updating the application state pessimistically or optimistically.

3. It provides help with dealing with multiple calls and with sequencing them in a meaningful way.

4. It also helps with being explicit about error handling: it 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)

5. It helps with initiating asynchronous logic when a route change occurs to a particular component.

Let's take a look at some of the features that it provides:

- Optimistic and pessimistic updates
- fetching arbitrary data reliably
- initiating asynchronous workflows on route change (to a particular component)

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
  ) {}

}
```

The DataPersistence library is a useful tool to standardize the way our applications interact with back-ends, rather than hand-coding it each time leading to different approaches across many applications.

Summary

This section explored the ways that Nx can help to enforce quality and consistency across the workspace.

- Nx enforces a Single Version Policy fro dependencies by containing a single `package.json` file for the entire workspace, and by virtue of having a single repository (so that all the code available in the organization or business unit is available to be accessed directly without needing to be deployed).

- Nx ensures consistency in code formatting by making prettier available with the formatting npm scripts that can be run locally and in CI. This results in better diffs in PRs by eliminating formatting and whitespace diffs.

- Nx provides ways to enforce restrictions in library dependencies by configuring boundary enforcement between libraries e.g. a `util` library not being able to depend on a `feature` library. It includes built-in enforcement of rules that apply to all workspaces (e.g. libs cannot import from apps), and provides a way to bypass the restrictions for specific cases when it is necessary.

- Nx helps to enforce consistency in code generation by making it easy to generate workspace schematics.

- Nx provides the DataPersistence module which enforces consistency and reliability in network communications within applications.

Part 4: Helping With Builds And CI

This is a short section on how Nx includes ways to make CI and CD a little easier by reducing the amount of time needed to build the projects.

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 of 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`, `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 file list that we can use to determine which projects are affected by those changes and only process those.

  > This also includes untracked and uncommitted changes!

- **Explicit files**: Use `--files` to provide an explicit comma-delimited file list (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)

  > The following are private options: `--uncommitted` and `--untracked`. They should not be used directly.

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:
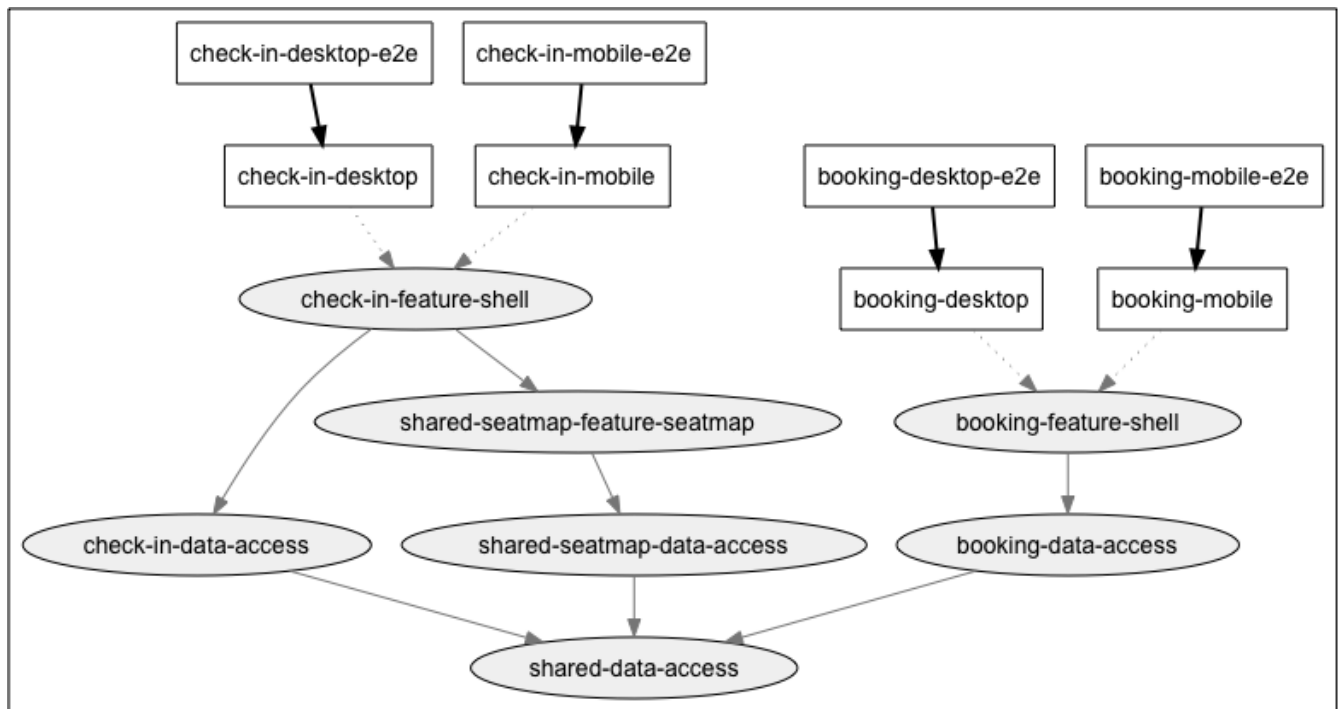

Figure 12. Original dependency graph

Scenario 1: Change In an 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`.
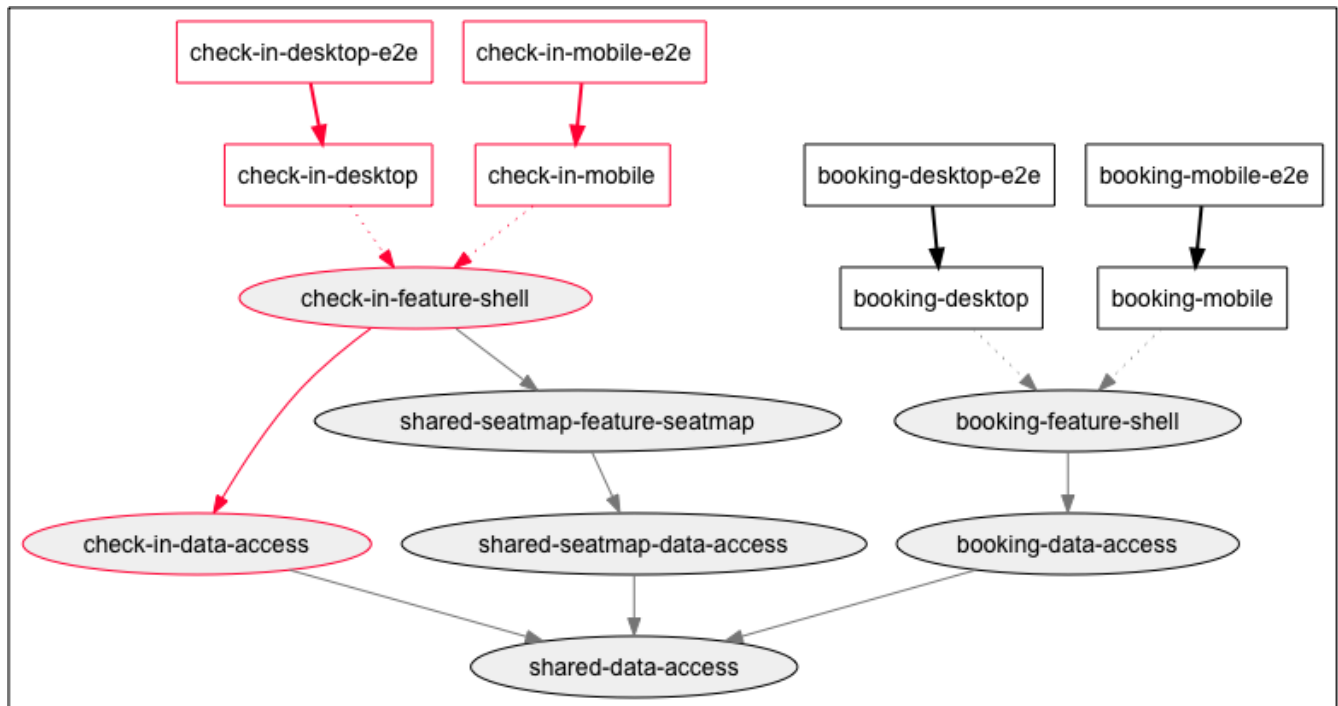

Figure 13. We don't have to build or test `booking` or `seatmap`!

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.
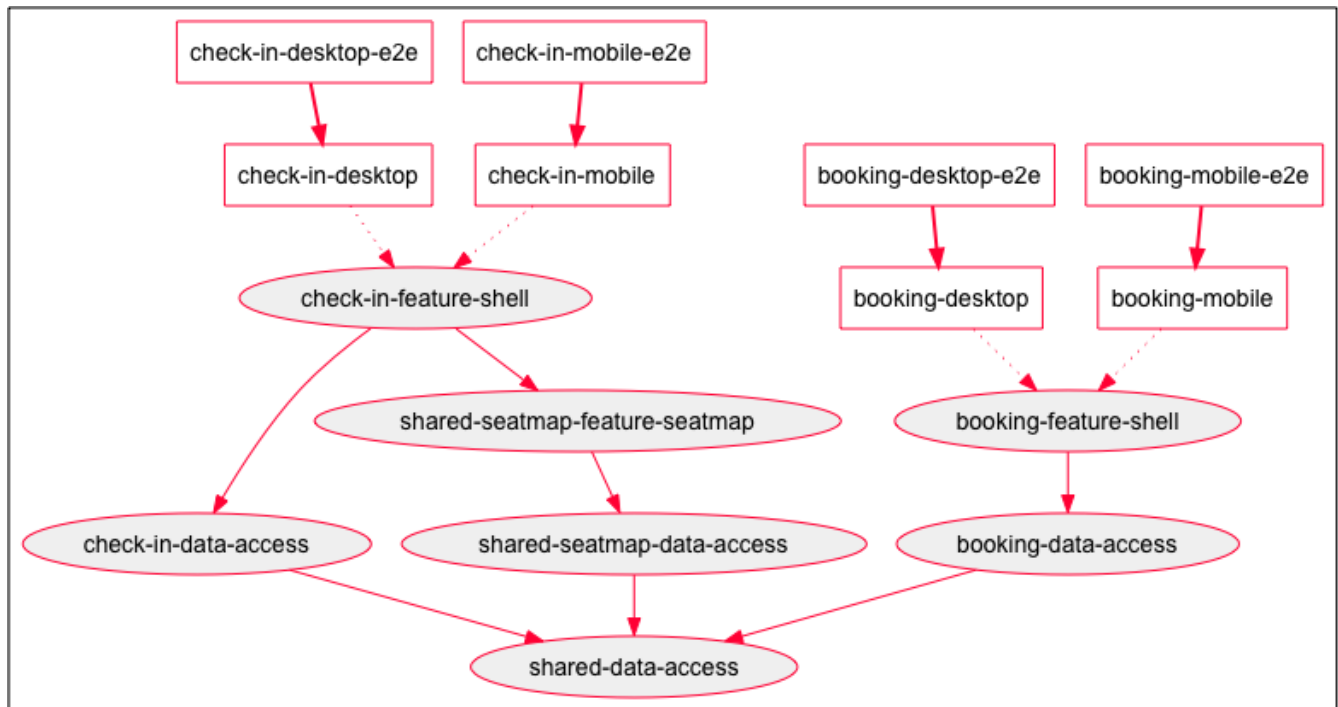
Figure 14. Changes made to the `shared-data-access` lib affect everything

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.

The other major advantage is that we don't need to run the tests for projects that weren't affected by our changes. This also has a considerable effect on the time it takes to get through CI.

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 limit the maximum number of parallel processes used.

```
npm run affected:build -- --base=master --parallel --maxParallel=4
```

Summary

This section focussed on the ways that Nx can help with making the build times much shorter within CI/CD pipelines.

- Only the apps and libs affected by code changes in a PR need to be **rebuilt** and **retested**. Nx provides this functionality and reduces build times inversely proportional to the number of affected projects (if only a few projects are affected, build times are drastically shorter). At its worst (the PR affects every project in the workspace) it takes exactly as long as when it is not used; so there is no down-side to using the `affected` commands.

- We can still force all projects to be rebuilt or retested by passing a flag (`--all`).

Part 5: Development Challenges In a Monorepo

We've spent the majority of the book talking about how Nx can provide tooling that helps with development within a monorepo but we have delayed talking about what challenges a monorepo sets out to solve, what its advantages and drawbacks are, and what type of unique challenges an organization would face after moving to developing within a monorepo.

Here are some common questions asked when migrating to 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 versioning of 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 creating too much technical debt in future 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.

How To Deal With Code Changes Between Teams

Let's consider the example repo. The dependency graph is below:
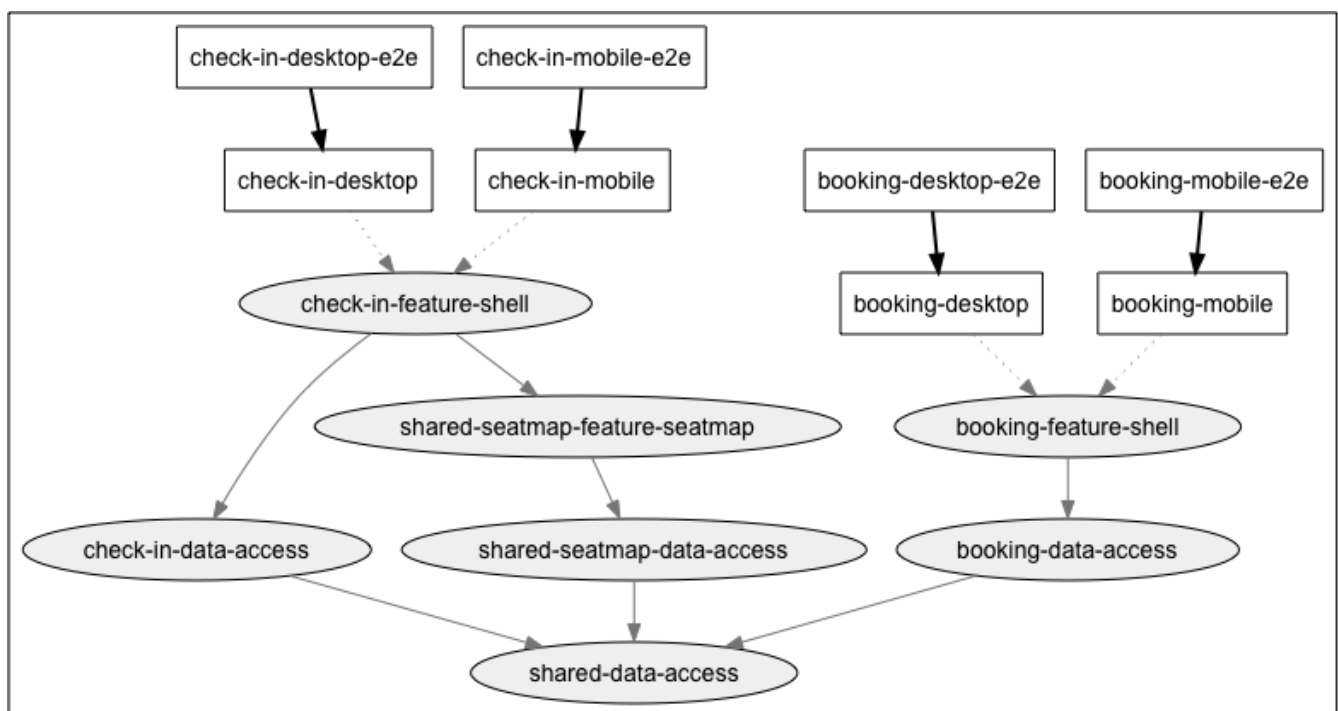
Figure 15. The example repo structure and dependencies

Our repo has four (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. The teams are confused about what branches they should maintain: How can we ensure that the trunk branch is deployable? Should we even adopt trunk-based development?

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:

   a. Create a new version of the code (method/class/library).

   b. Develop on this new version until it is ready

   c. Issue a deprecation warning for the old method/class/library with a set expiry time

   d. Work with the other teams to help them migrate to the new version (see Code Owners below)

   e. Remove the old version when the expiry time elapses

Let's take a closer look at two solutions presented above for the issues of managing code changes between teams: code owners and feature toggles (aka. feature flags).

Code Ownership

A good way to assign responsibility for a lib is to assign code owners for the libs in the Nx workspace. A code owner (usually a group rather than a single individual) is responsible for all changes to a lib, and would orchestrate the process 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.

Github allows for the specification of code owners in a repository and similar functionality might

be available in your organization's git or CI provider.

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. These are usually controlled via the URL.

Let's consider two examples.

Build-time toggle

Let's say that we are migrating the back-end from v1 to v2. We add the functionality to initialize the application environment by reading from environment variables during the deployment in CI/CD. However, since this happens during deployment, we aren't able to dynamically flip the version toggle at will after the application is loaded.

This is a build-time toggle - it is only meant to be set when the application builds or is deployed. It can't be changed at run-time.

Run-time toggle

Let's say that the Seatmap team is completely changing their UI so that the seats display a lot more information to the user: whether they have extra leg-room, whether they are in an exit row, etc. The Seatmap team would like the user to be able to click a button in the UI to try out the new UI.

This is a run-time feature toggle - the feature can be turned on and off using the UI, changing the URL, retrieving a value from the server, or any other method that retrieves the value to be used at run-time. We don't need to deploy a new version of the application.

These feature toggles allow teams to continue development on features that touch shared code without stepping on each other's toes. Features can be introduced into the code base and worked upon without being enabled for the end user.

They are especially useful when using trunk-based development. Let's take a brief look at what this means for teams.

Trunk-based Development

Trunk-based development revolves around the concept of having a single branch in the repository that acts as the "trunk" or the main branch that all of the teams use. This is the branch from which all feature branches originate and the one that all of them get merged into.

There are two fundamental concepts with trunk-based development:

- The existence of a single branch that all feature branches originate from and get merged into.
- The short-lived nature of feature branches: each feature branch only lasts a day or two and has a very specific purpose.

The main reason for working in this way is that long-lived branches cause problems when it is time to merge them back with code from other teams:

- The code is very much out-of-date and will require a very long time to resolve all of the merge conflicts.
- It requires and additional round of regression testing after resolving the conflicts before the code can be merged back in.
- It is very difficult to roll back and debug changes because of the number of branches involved.
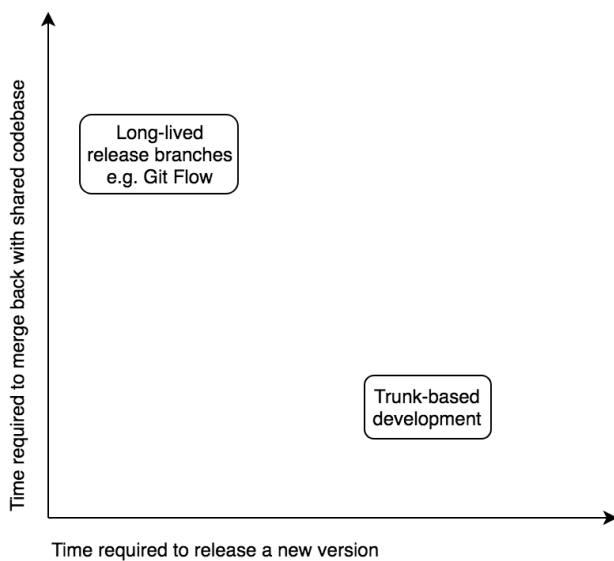
Figure 16. Git flow favours speed in releasing new artifacts over time to merge the code back

We generally encourage trunk-based development because we believe in the basic 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
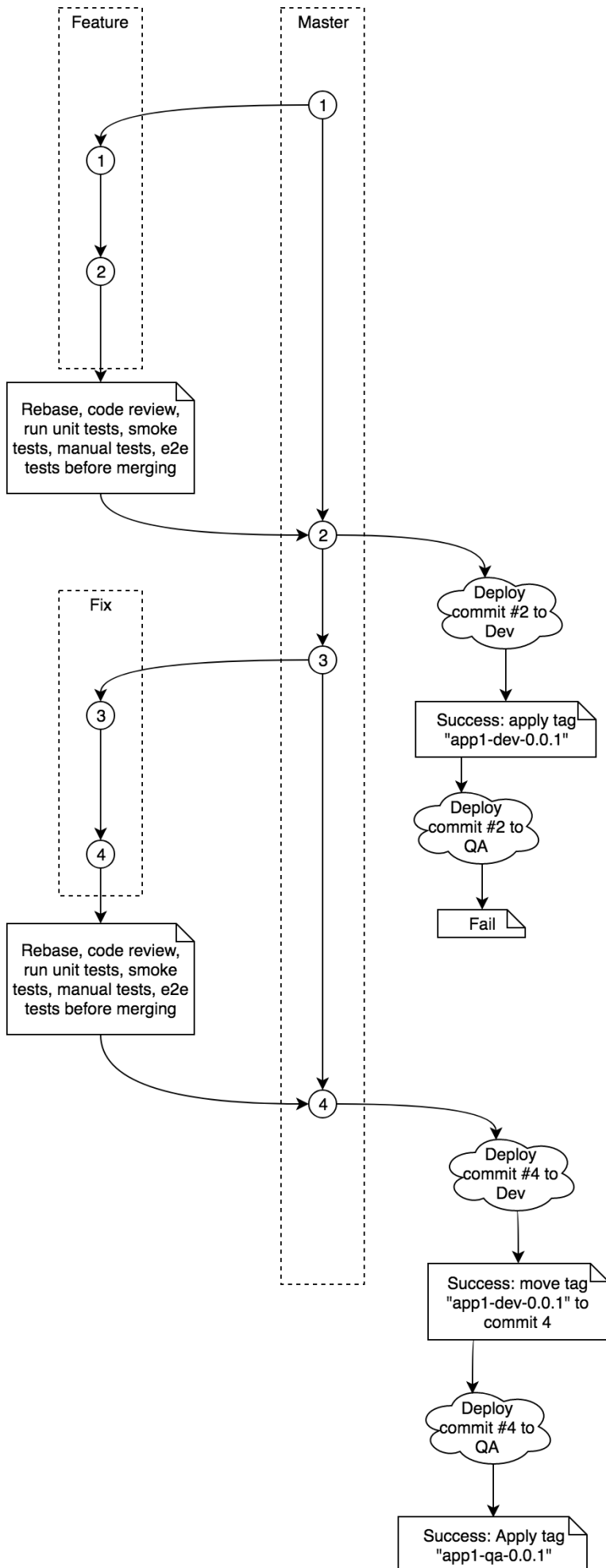
Figure 17. Trunk based Development

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 are smaller PRs, requiring that PRs ar eup-to-date with the target branch, and to use the three-step plan when making updates to shared code: creating a new version and using a feature-toggle when working on it, deprecating the old version and working with code owners to transition to the new version, and removing the old version.

- Setting up code owners for various libs assigns responsibility both for merging code into that library and also to promote the changes responsibly to the affected teams.

- Feature toggles allow teams to tests their code in environments without affecting or being affected by other teams. Build-time toggles are useful when dealing with settings that are required during initialization of the application, and run-time toggles can be changed after the application is running by the end-user.

- Trunk-based development is preferable when working in a monorepo in order to avoid the large amount of effort when it comes time to merge branches.

- 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.
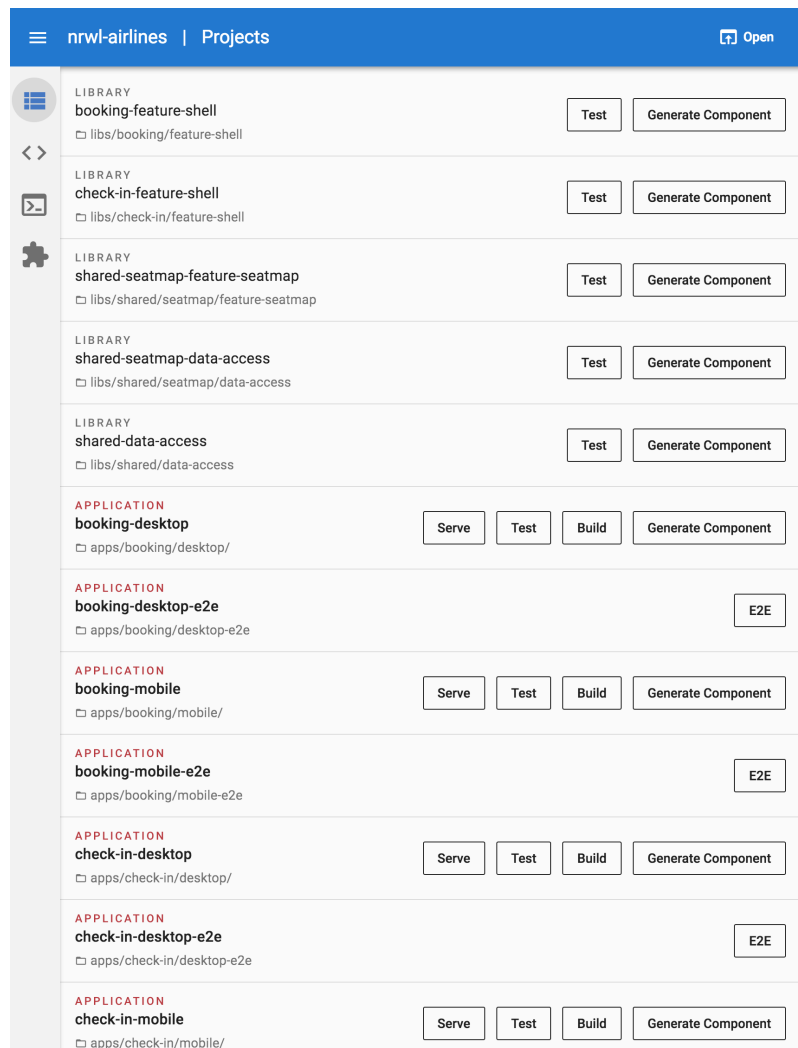
Figure 18. Angular Console

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.
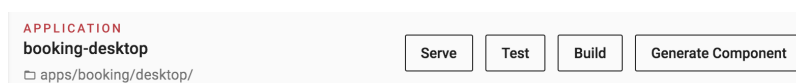
Figure 19. Application quick actions

Libs have quick actions for test and create component. Further actions would also be found

under `Run tasks` in the left menu.
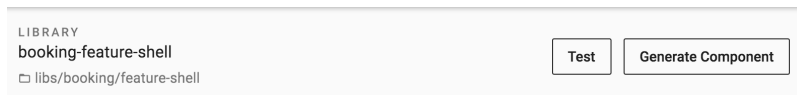
Figure 20. Library quick actions

Run npm scripts

Angular Console reads in all of the commands specified under `scripts` in `package.json` and allows their execution.
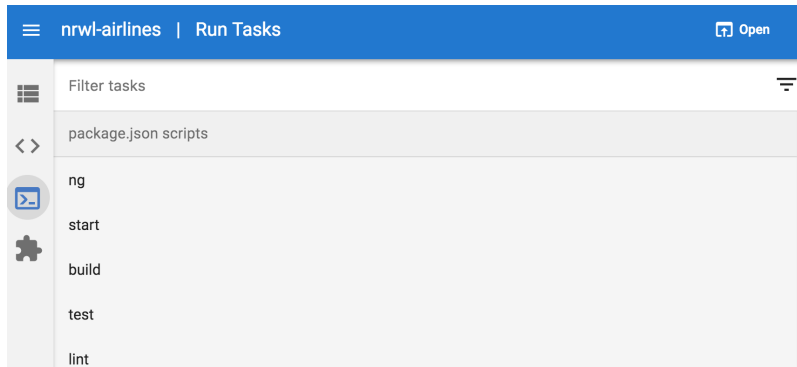
Figure 21. Run any task in package.json

Interactive dependency graph

See Part 3 of the book for a description of the dependency graph
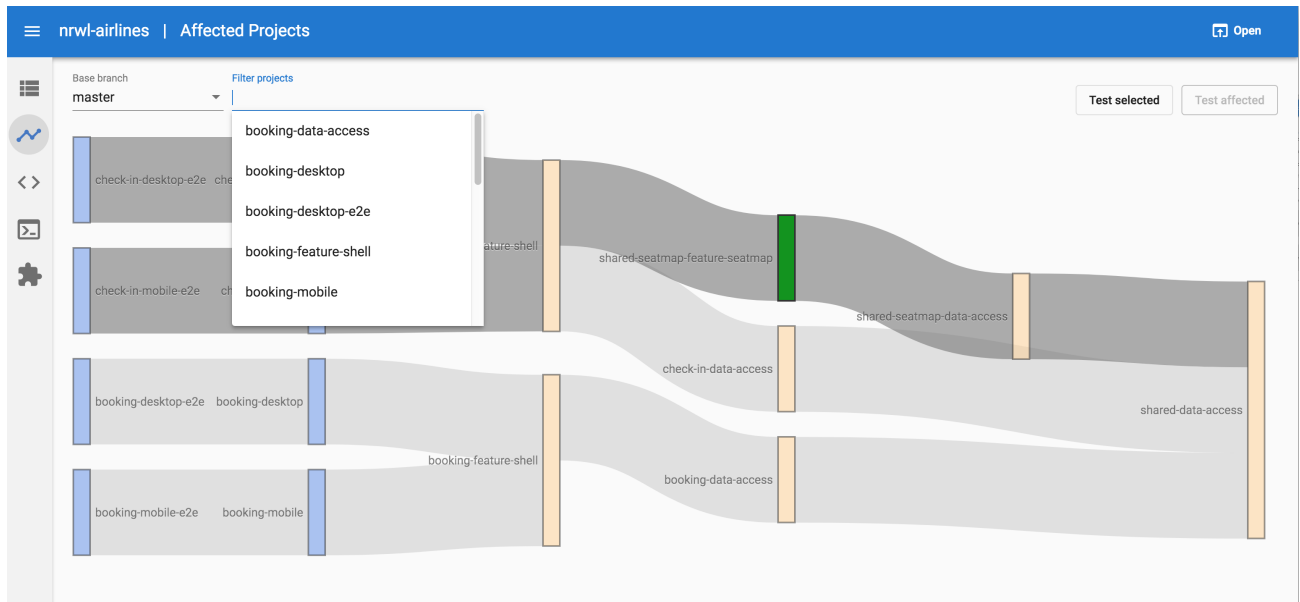
Generate code

We can generate the schematics detected in the workspace by picking from the grouped list.
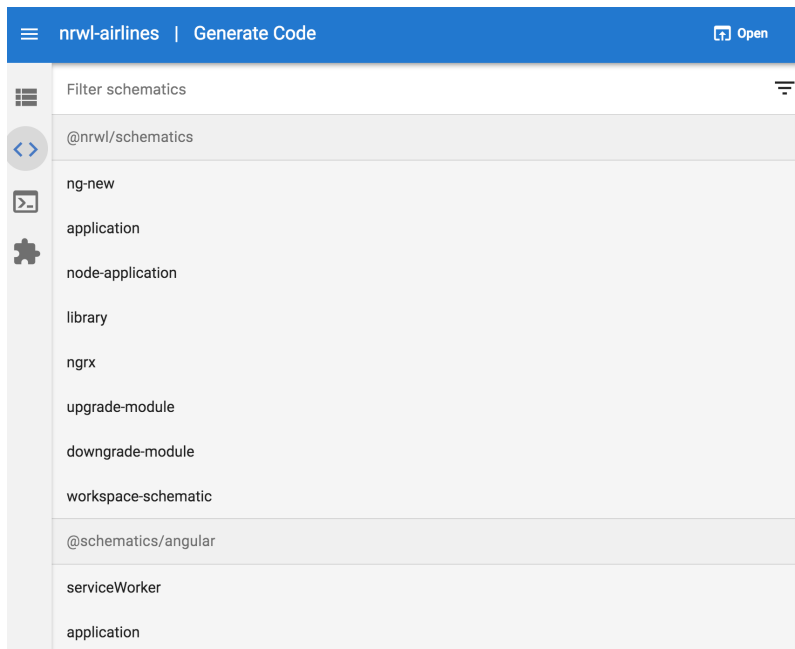
Figure 22. Code generation support

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.
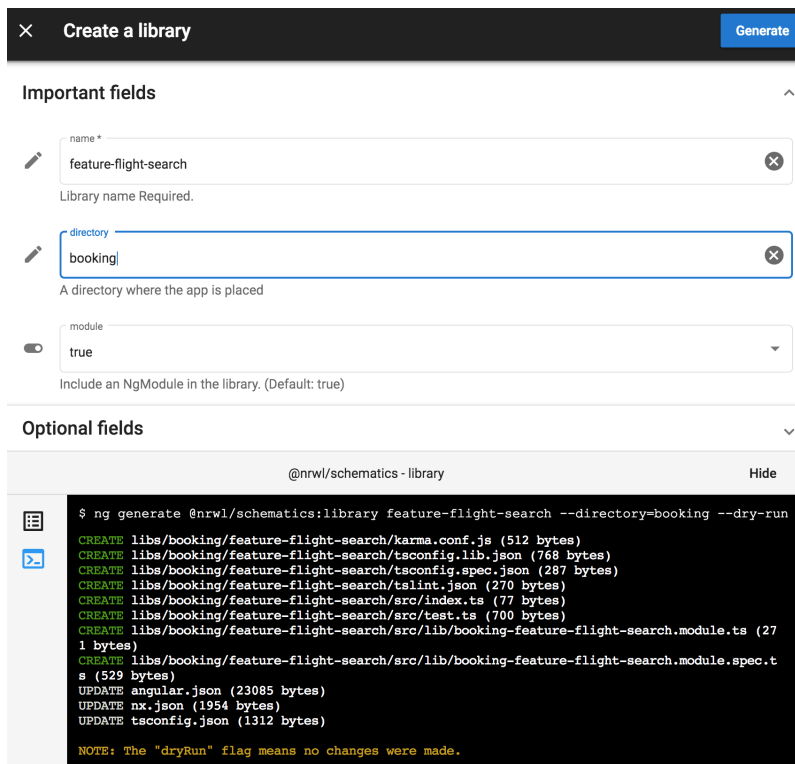
Figure 23. All options and expected output

Third-party extensions

Third-party extensions can be added from a curated list.
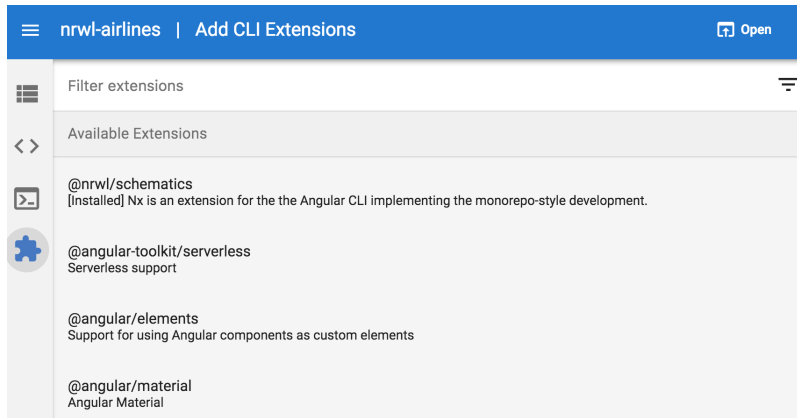
Figure 24. Add third-party extensions

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

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 the Code Formatting section in Part 3 of the book.

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 the Analyzing and Visualizing the Dependency Graph section in Part 3 of the book.

workspace-schematics

Nx allows you to generate workspace schematics that can assist with automating workflows and generating scaffolding.

`Affected` **Commands**

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

- affected

- 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)

- 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.

```
ng update
```

Where Should I Create My New Lib?
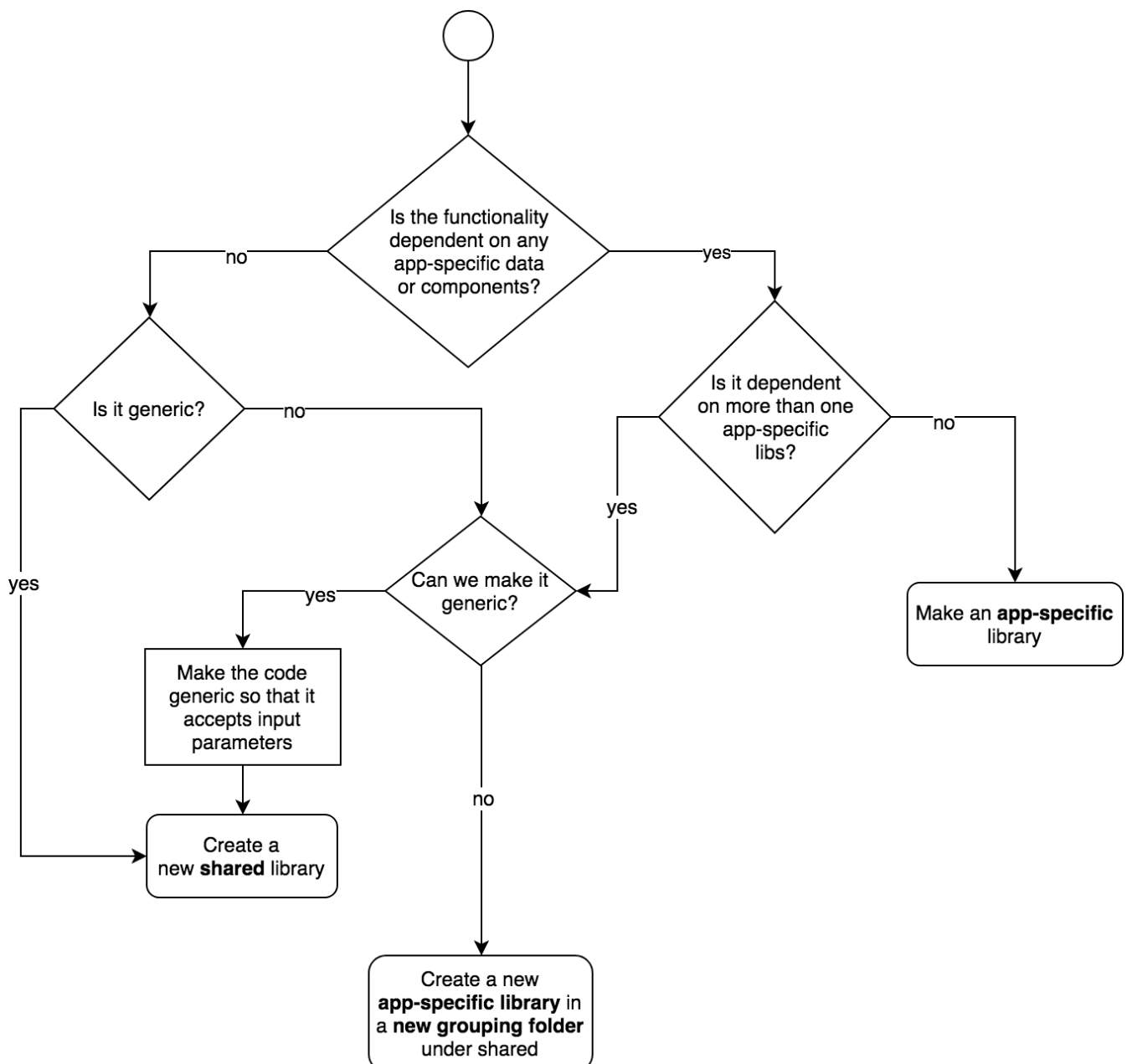
Figure 25. Shared vs. app-specific lib

In general there are two levels that we can share libs: across all or most applications in the workspace, and across a subset.

Domain-agnostic libs

These libraries can be used across most applications in the workspace without needing to reference anything in domain-specific libraries.

An example domain-agnostic library from our example workspace is `shared-data-access`: it doesn't need to know or reference anything in the `booking`, `check-in`, or `seatmap` domains.

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 one or a few applications. These go into grouping folders. (e.g., `booking`)

An example is the `feature-seatmap` library: it does need to reference other libraries in the `seatmap` domain (e.g. retrieving a value from the seatmap slice of the Store).

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.
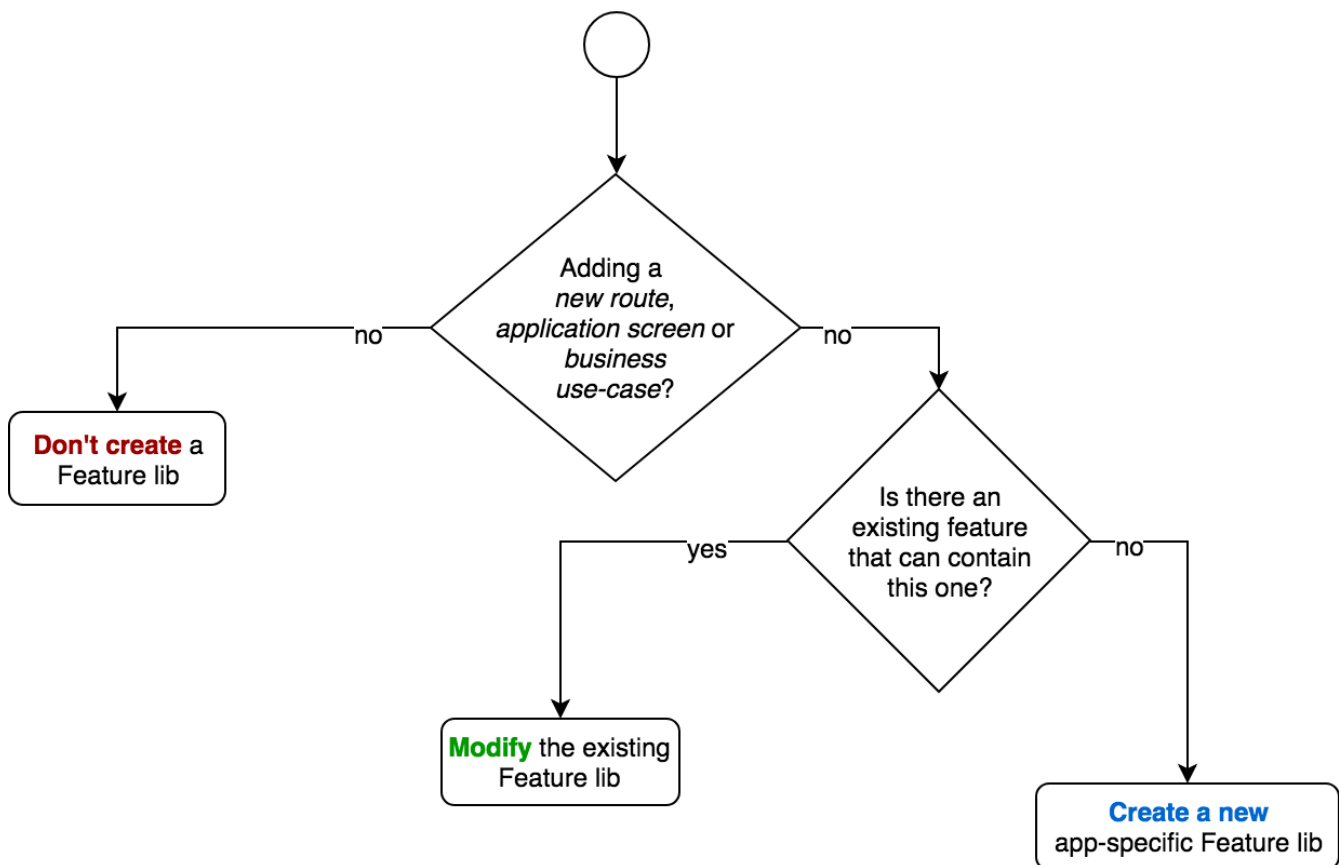
Figure 26. Should I create or modify a feature lib?

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.

Index

Y

Discover more resources from Nrwl

Read our blog: **blog.nrwl.io**

Use our open source products: **nrwl.io/products**

Find out how our team can help yours: **nrwl.io/services**

About Nrwl

For enterprise software teams and technology leaders, Nrwl is a software development consultancy and developer training company that provides experts, developer tools and best practices for collaboration so enterprises can build high-quality software that stands the test of time. Unlike other development firms Nrwl has unparalleled engineering talent and thought-leaders, who are closer to the source of truth for Angular. Nrwl has the world's highest concentration of Angular expertise outside of Google, and we use that depth to help our clients work through the biggest and most nuanced challenges of building Angular applications at scale.

Contact Us

hello@nrwl.io