Enterprise Angular Monorepo Patterns

Enterprise Angular Monorepo Patterns

Authors: Nx Core Team

Updated March 2025

For more information visit nx.dev

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 (workspace, apps & libs) 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 to make intelligent builds with Nx and use its set of tools locally as well as how to integrate the tools into 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 (Nx 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 Nx 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.nx-airlines.com` or `check-in.nx-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:

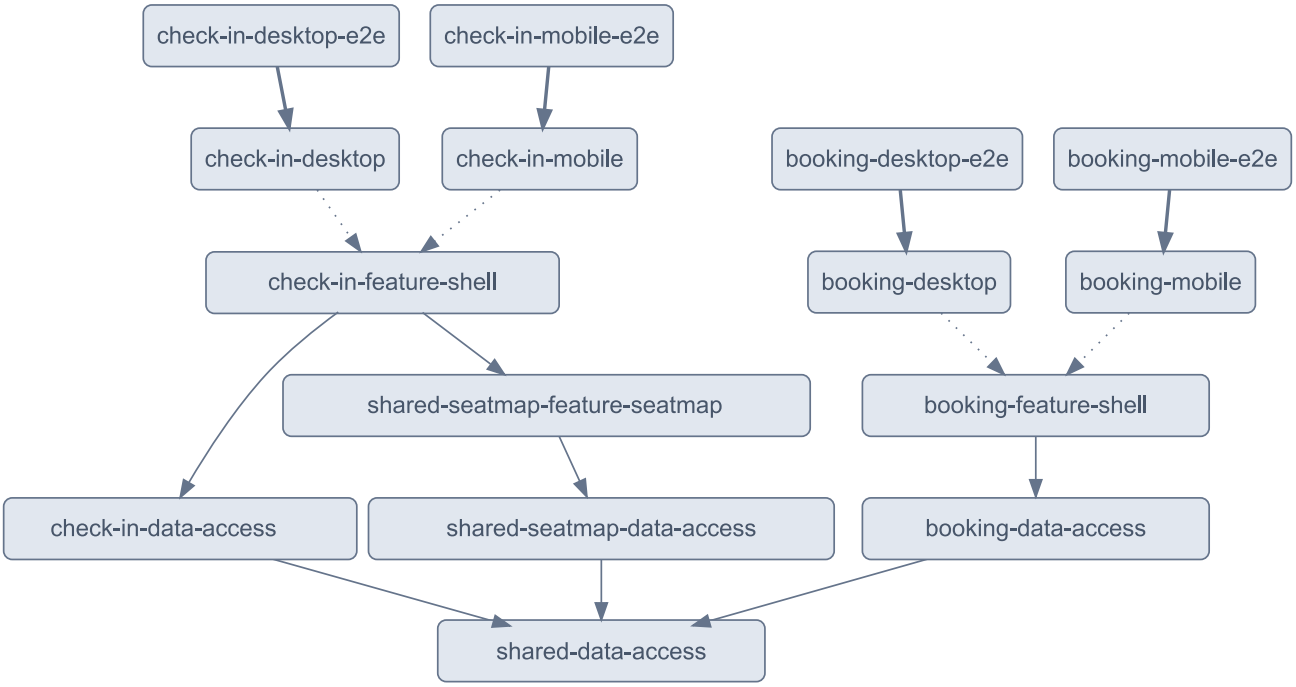


Figure 1. The example repo structure and dependencies

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 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:

- Generators to help with code generation in the projects to ensure consistency.
- Tools to help with enforcing linting rules and code formatting.
- Visualization of dependencies between apps and libraries within the repository
- 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.

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 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 Nx, 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 file 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` for workspace and plugin configuration
- `project.json` for per-project configuration

Nx also provides commands to work with your workspace, apps and libs. These can be entered into

a terminal or called from within the Nx Console, which is an editor extension to interact with the Nx CLI. The Nx 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 Nx 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 create a new Nx Workspace with the following command

```
npm create nx-workspace
```

This creates a new Nx Workspace and will prompt you to pick your framework of choice, as well as a few other tools (bundler, linter, E2E testing).

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

```
npx nx@latest init
```

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/
.editorconfig
.gitignore
.prettierignore
.prettierrc
eslint.config.mjs
jest.config.ts
jest.preset.js
nx.json
package.json
tsconfig.base.json
```

It's similar to the standard Angular projects, but instead of a single `src` directory, there is an `apps` directory where all applications are 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
- **nx.json**: This is used by Nx to provide metadata for projects e.g. tags
- **eslint.config.mjs**: This is the linter configuration file
- **tsconfig.base.json**: This is the workspace's main tsconfig. Nx adds path aliases for each lib here to allow for workspace-relative imports e.g.

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

A new Nx Workspace will set up an initial application, and adding more is simple.

Creating an App

Add a new app to an Nx Workspace using the Nx generate command. Nx has a generator named `app` that can be used to add a new app to our workspace:

```
nx g @nx/angular:app apps/myapp
nx generate @nx/angular:application apps/myapp # same thing
```

This command will scaffold out a new Angular app and place it in the `apps` directory.

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

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.

```
nx g @nx/angular:lib libs/mylib
nx generate @nx/angular:library libs/mylib # same thing
```

This creates a new lib, places it in the `libs` directory, and configures the `nx.json` files to support the new lib. This will also add an additional path alias to `tsconfig.base.json` to simplify the import statement in your apps.

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 Nx 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 @nx/angular:lib --help` to see the list of available options:

Output when using the --help option

```
 NX   generate @nx/angular:library [directory] [options,...]

From:  @nx/angular (v20.4.5)
Name:  library (aliases: lib)

  Creates an Angular library.

Options:
    --directory                A directory where the                [string]
                               library is placed.
    --buildable                Generate a buildable                [boolean]
                               library.
    --lazy                     Add                                 [boolean]
                               `RouterModule.forChild`
                               when set to true, and a
                               simple array of routes
                               when set to false.
    --name                     The name of the                      [string]
                               library.
    --parent                   Path to the parent                   [string]
                               route configuration
```

```
                            using `loadChildren` or
                            `children`, depending
                            on what `lazy` is set
                            to.
--publishable               Generate a publishable              [boolean]
                            library.
--routing                   Add router                          [boolean]
                            configuration. See
                            `lazy` for more
                            information.
--addModuleSpec             Add a module spec file.             [boolean]
--addTailwind               Whether to configure                [boolean]
                            Tailwind CSS for the
                            application. It can
                            only be used with
                            buildable and
                            publishable libraries.
                            Non-buildable libraries
                            will use the
                            application's Tailwind
                            configuration.
--changeDetection, -c       The change detection       [string] [choices:
                            strategy to use in the    "Default", "OnPush"]
                            new component.           [default: "Default"]
                            Disclaimer: This option
                            is only valid when
                            `--standalone` is set
                            to `true`.
--compilationMode           Specifies the              [string] [choices:
                            compilation mode to        "full", "partial"]
                            use. If not specified,
                            it will default to
                            `partial` for
                            publishable libraries
                            and to `full` for
                            buildable libraries.
                            The `full` value can
                            not be used for
                            publishable libraries.
--displayBlock, -b          Specifies if the                    [boolean]
                            component generated
                            style will contain
                            `:host { display:
                            block; }`. Disclaimer:
                            This option is only
                            valid when
                            `--standalone` is set
                            to `true`.
--flat                      Ensure the generated                [boolean]
                            standalone component is
                            not placed in a
```

```
                              subdirectory.
                              Disclaimer: This option
                              is only valid when
                              `--standalone` is set
                              to `true`.
  --importPath                The library name used                [string]
                              to import it, like `@my
                              org/my-awesome-lib`.
                              Must be a valid npm
                              name.
  --inlineStyle, -s           Include styles inline                [boolean]
                              in the component.ts
                              file. Only CSS styles
                              can be included inline.
                              By default, an external
                              styles file is created
                              and referenced in the
                              component.ts file.
                              Disclaimer: This option
                              is only valid when
                              `--standalone` is set
                              to `true`.
  --inlineTemplate, -t        Include template inline              [boolean]
                              in the component.ts
                              file. By default, an
                              external template file
                              is created and
                              referenced in the
                              component.ts file.
                              Disclaimer: This option
                              is only valid when
                              `--standalone` is set
                              to `true`.
  --linter                    The tool to use for         [string] [choices:
                              running lint checks.         "eslint", "none"]
                                                         [default: "eslint"]
  --prefix, -p                The prefix to apply to               [string]
                              generated selectors.
  --selector                  The HTML selector to                 [string]
                              use for this component.
                              Disclaimer: This option
                              is only valid when
                              `--standalone` is set
                              to `true`.
  --setParserOptionsProject   Whether or not to                    [boolean]
                              configure the ESLint
                              `parserOptions.project`
                              option. We do not do
                              this by default for
                              lint performance
                              reasons.
```

```
--simpleName                Don't include the                  [boolean]
                            directory in the name
                            of the module or
                            standalone component
                            entry of the library.
--skipModule                Whether to skip the                [boolean]
                            creation of a default
                            module when generating
                            the library.
--skipSelector              Specifies if the                   [boolean]
                            component should have a
                            selector or not.
                            Disclaimer: This option
                            is only valid when
                            `--standalone` is set
                            to `true`.
--skipTests                 Do not create `spec.ts`            [boolean]
                            test files for the new
                            component. Disclaimer:
                            This option is only
                            valid when
                            `--standalone` is set
                            to `true`.
--skipTsConfig              Do not update                      [boolean]
                            `tsconfig.json` for
                            development experience.
--standalone                Generate a library that    [boolean] [default:
                            uses a standalone                      true]
                            component instead of a
                            module as the entry
                            point.
--strict                    Create a library with      [boolean] [default:
                            stricter type checking                 true]
                            and build optimization
                            options.
--style                     The file extension or       [string] [choices:
                            preprocessor to use for "css", "scss", "sass",
                            style files, or `none`        "less", "none"]
                            to skip generating the       [default: "css"]
                            style file. Disclaimer:
                            This option is only
                            valid when
                            `--standalone` is set
                            to `true`.
--tags                      Add tags to the library             [string]
                            (used for linting).
--unitTestRunner            Test runner to use for     [string] [choices:
                            unit tests.                "jest", "vitest",
                                                       "none"] [default:
                                                                 "jest"]
--viewEncapsulation, -v     The view encapsulation     [string] [choices:
```

```
                              strategy to use in the     "Emulated", "None",
                              new component.                    "ShadowDom"]
                              Disclaimer: This option
                              is only valid when
                              `--standalone` is set
                              to `true`.
    --skipFormat              Skip formatting files.              [boolean]
    --skipPackageJson         Do not add dependencies             [boolean]
                              to `package.json`.
    --standaloneConfig        Split the project          [boolean] [default:
                              configuration into                     true]
                              `<projectRoot>/project.
                              json` rather than
                              including it inside
                              `workspace.json`.



Find more information and examples at: https://nx.dev/nx-
api/angular/generators/library
```

Getting help is also possible using the Nx 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).

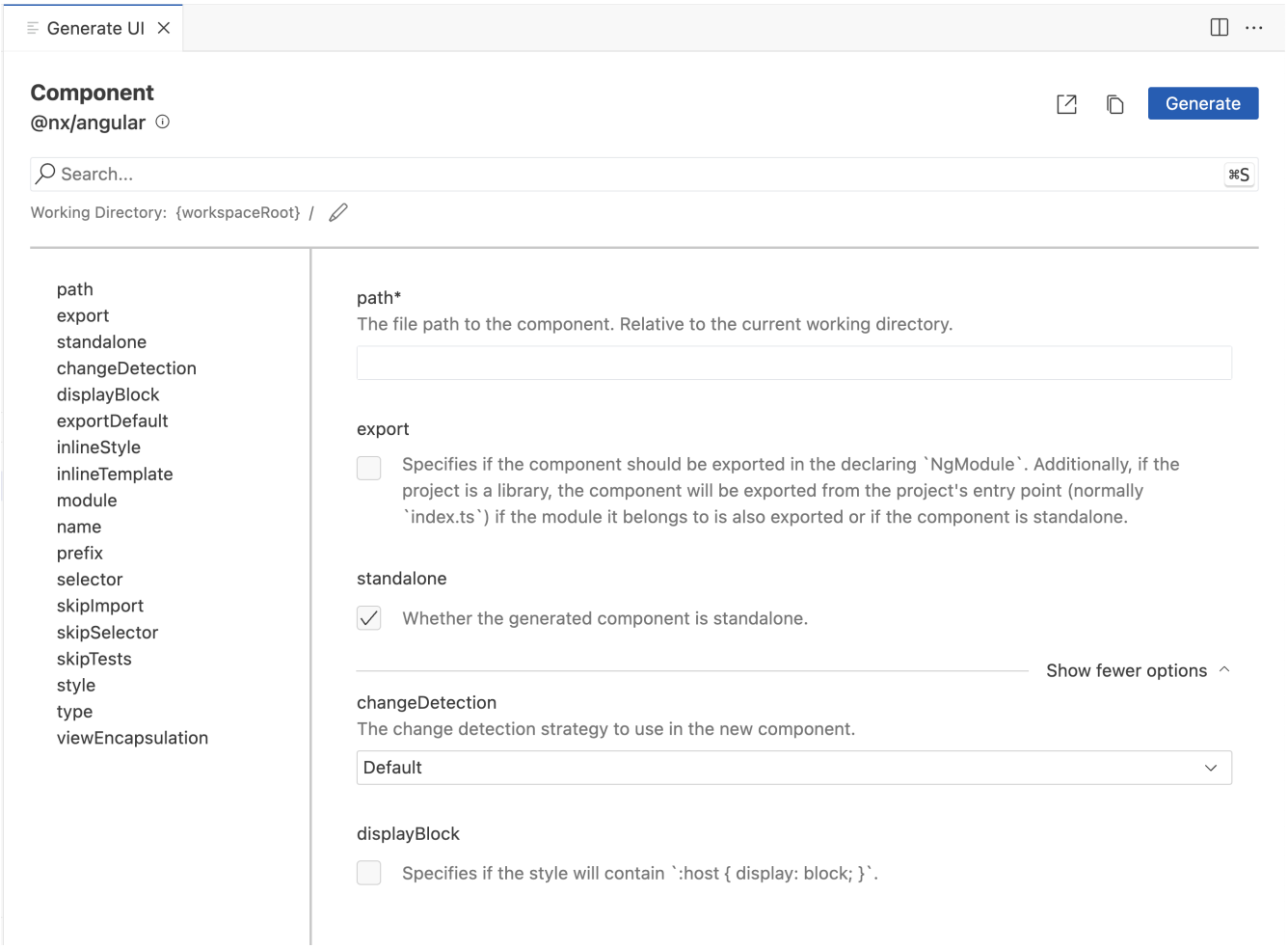


Figure 2. Getting help with Nx Console. Note that the options are separated into "important" and "optional", and that the values are able to be selected in a drop down.

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 Nx Console.
- We looked at two different ways to get help with the commands: the `--help` command line option, and opening the project in Nx 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 file 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 should have a classifiers to describe their contents and intended purpose. These classifiers help to organize the libraries and to provide a way to distinguish their intent. 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-<intent>/
```

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 container UI (with injected services) for specific **business use cases** or pages in an application.
- **UI libraries**: A UI library contains only `presentational` components (also called "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 Container and Presentational components.

Aside: Container 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 "container" components), and those that only receive data (known as "presentational" components).

Container Components

These components manage or delegate business logic and use DI (Dependency Injection) 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 container 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 Container and Presentational components, let's look at the types of libraries that contain them. Feature libraries contain container 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 container 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/
          .. ②
```

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

② Any component that is part of the `feature-shell` will be placed in the `lib` direcotry

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

> In our example application, nx-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 const routes = [
  {
    path: '',
    pathMatch: 'full',
    component: 'FlightSearchComponent',
  },
  {
    path: '/passenger',
    pathMatch: 'full',
    loadChildren: () =>
      import('@nx-airlines/passenger-info')
      .then(m => m.passengerInfoRoutes)
  },
  {
    path: '/seatmap',
    pathMatch: 'full',
    loadChildren: () =>
      import('@nx-airlines/seat-listing')
      .then(m => m.seatListingRoutes)
  },
];
```

If this routing were to be placed into `apps/booking/desktop`, we would need to duplicate this in `apps/booking/mobile` (assuming that both applications behave the same way and have the same routes)> We'd also need to maintain the two to be in sync going forward. In order to keep this DRY, we want to extract the routing out to a shared location.

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 routes in this lib can also be lazy-loaded into both parent applications `booking/desktop` and `booking/mobile`.

An appropriate naming convention is `<feature>-routes` located in `libs/booking/feature-shell` as shown above the example. We now have four feature libs:

- `booking/feature-shell`: sets up initialization and the routing
- `booking/feature-flight-search`: displays the flight search for booking
- `booking/feature-passenger-info`: displays the passenger info for booking
- `shared/seatmap/feature-seat-listing`: displays the seat listing that is shared between booking and check-in

An example feature lib

```
  import { Routes } from '@angular/router';
  import { FlightSearchComponent } from './components/flight-search/flight-
search.component';

  export const routes: Routes = [ ①
    {
      path: '',
      component: FlightSearchComponent,
      pathMatch: 'full'
    },
    {
      path: '/passenger',
      pathMatch: 'full',
      loadChildren: () =>
        import('@nx-airlines/passenger-info')
        .then(m => m.passengerInfoRoutes)
    },
    {
      path: '/seatmap',
      pathMatch: 'full',
      loadChildren: () =>
        import('@nx-airlines/seat-listing')
        .then(m => m.seatListingRoutes)
    }

  ];
```

① Here we're just exporting a regular const that we can import in our main routes config.

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 container 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 entry point

```
export * from './lib/confirm-button/confirm-button.component'; ①
```

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

Outside of container and presentational 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 Web Socket 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 library

```
import { inject, Injectable } from "@angular/core";
@Injectable({
  providedIn: 'root',
})
export class CustomerDataAcces {
  private backend = inject(BackendService);
  async getCustomerData(id: number) {
    await this.backend.getCustomer(id);
  }
  async updateCustomerData(id: number) {
    await this.backend.updateCustomer(id);
  }
  // And more methods
}
```

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

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 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.ts ①
src/
  index.ts ②
  lib/
    shared-data-access.spec.ts
    shared-data-access.ts
tsconfig.json
```

```
tsconfig.lib.json
tsconfig.spec.json
eslint.config.mjs
```

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.base.json` file, which would contain the following entry:

tsconfig.base.json

```
"paths": {
  ...
  "@nx-airlines/data-access-seatmap": [ ①
    "libs/shared/data-access-seatmap/src/index.ts"
  ]
  ...
}
```

① Configuration of the path alias for our lib

The above allows us to write `import {} from '@nx-airlines/data-access-seatmap';` anywhere in our workspace and instructs typescript to import the feature we've chosen the export.

Each app and lib that we create using the Nx 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. `@my-org/customers/`) instead of relative paths.

Eslint 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)

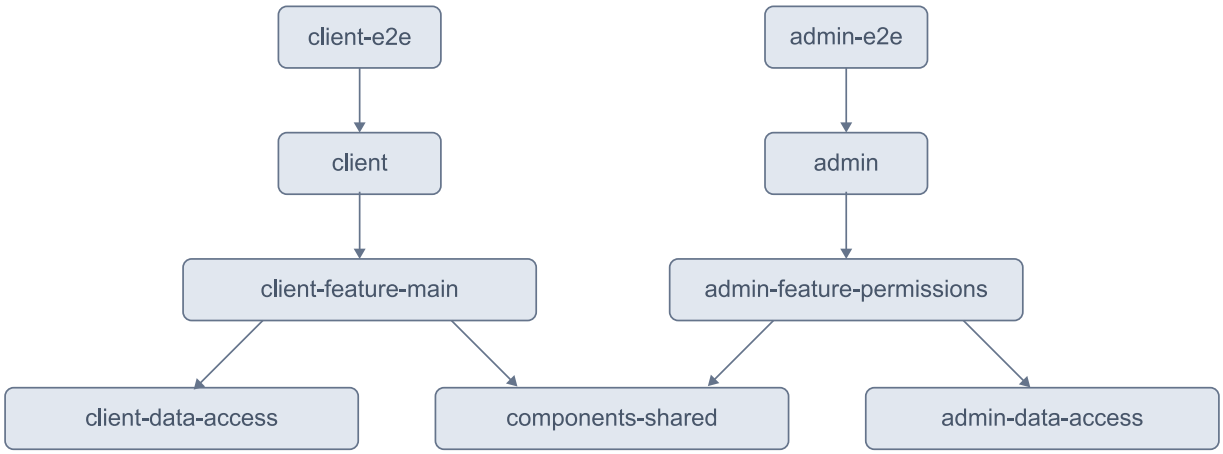


Figure 3. An example visualization

Command Line Options

When using the Nx generators to create libraries, developers have several **power** options. Most of these options are similar to the ones supported by the Angular CLI, but the following worth highlighting.

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

Options when using `nx generate @nx/angular:lib mylib`

Flags	Result
`--directory=myteam`	Create a new library with path `libs/myteam/mylib`
`--routing`	Configure the lib's router config to wire up routing to be loaded eagerly.
`--parent=apps/myapp/src/app/app.routes.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.

Flags	Result
`--publishable`	Generate a few extra configuration files for ng-packagr. You can then `nx 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.

In most cases, Nx does it by default for you. 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 (4) 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`, `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 generators to handle code generation consistently.

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 recommends a single version policy by containing a single package.json and by the virtue of being only one repo.

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 builtin support for Prettier.

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

Run `npx nx format --help` to see the available options.

`npx nx format --base=[SHA1] --base=[SHA2]`. Nx calculates what changed between the two SHAs, and formats the changed files. For instance, `npx nx format --base=origin/master --head=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 `npx nx graph`.

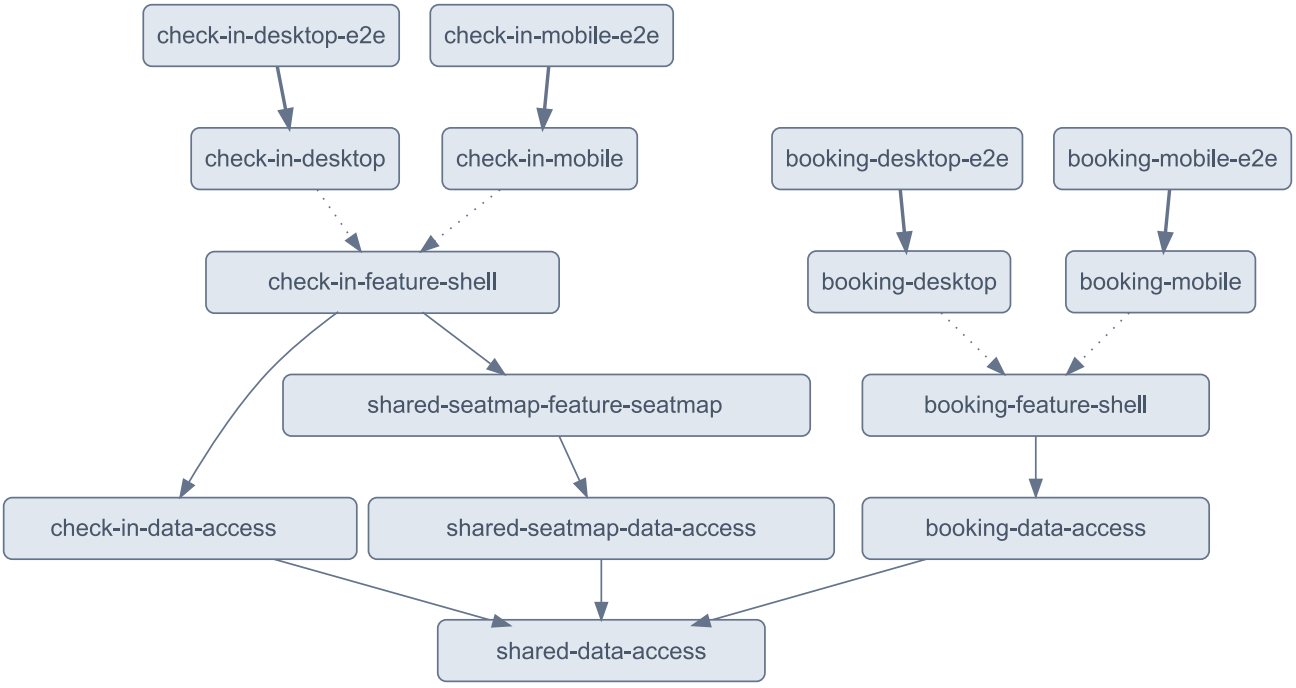


Figure 4. The dependency graph of the whole workspace

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

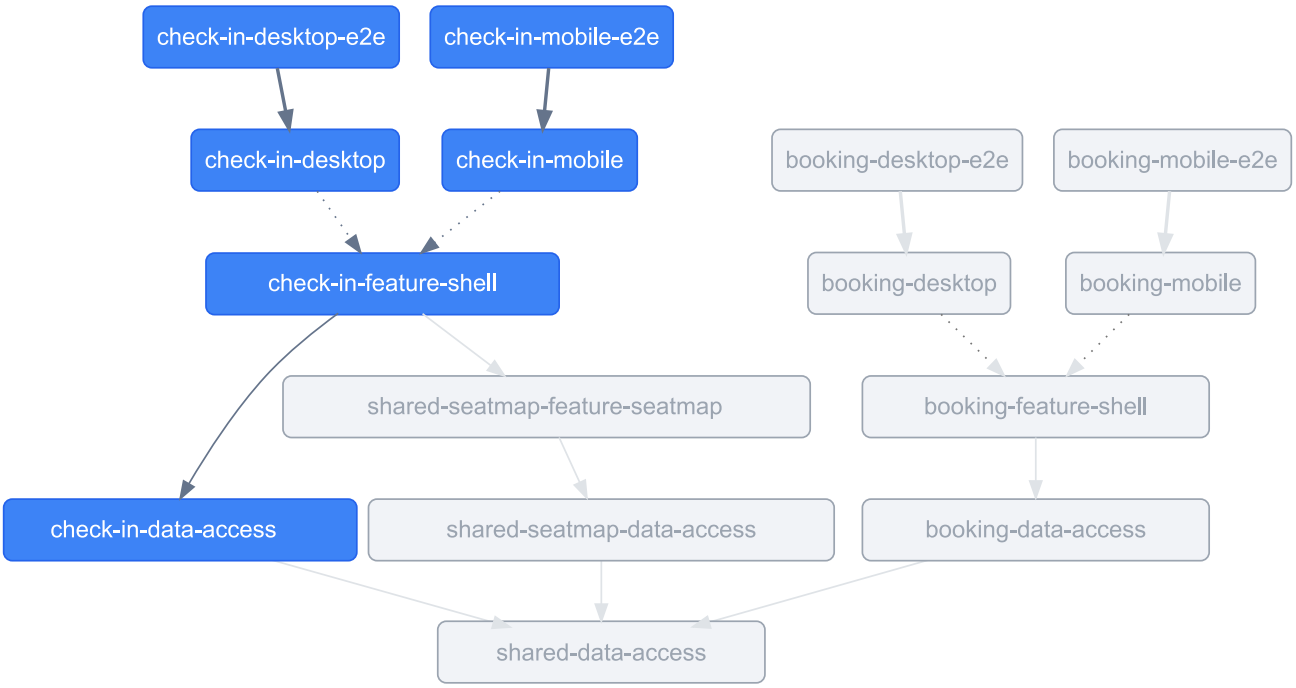


Figure 5. The dependency graph when we make a change to check-in-data-access

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 `graph` highlights the affected apps and libs and traces the critical path.

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

- `npx nx graph --file=graph.json` emits a json file.

- `npx nx graph --file=graph.html` emits a html file.

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.
- Circular dependencies aren't allowed.
- Libs cannot be imported using relative imports.

`eslint.config.mjs`

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

① This is a flag that enables the rule to be an error.

② 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 eslint rules)
- CI will fail if we use the lint rule in the pipeline

Constraints are defined using the rules in `eslint.config.mjs` and tags in `project.json`:

```
  npx nx g @nx/angular:lib feature-destination --directory=libs/booking --tags
=scope:booking,type:feature
```

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

`eslint.config.mjs`

```
  "@nx/enforce-module-boundaries": [
    "error",
    {
      "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).

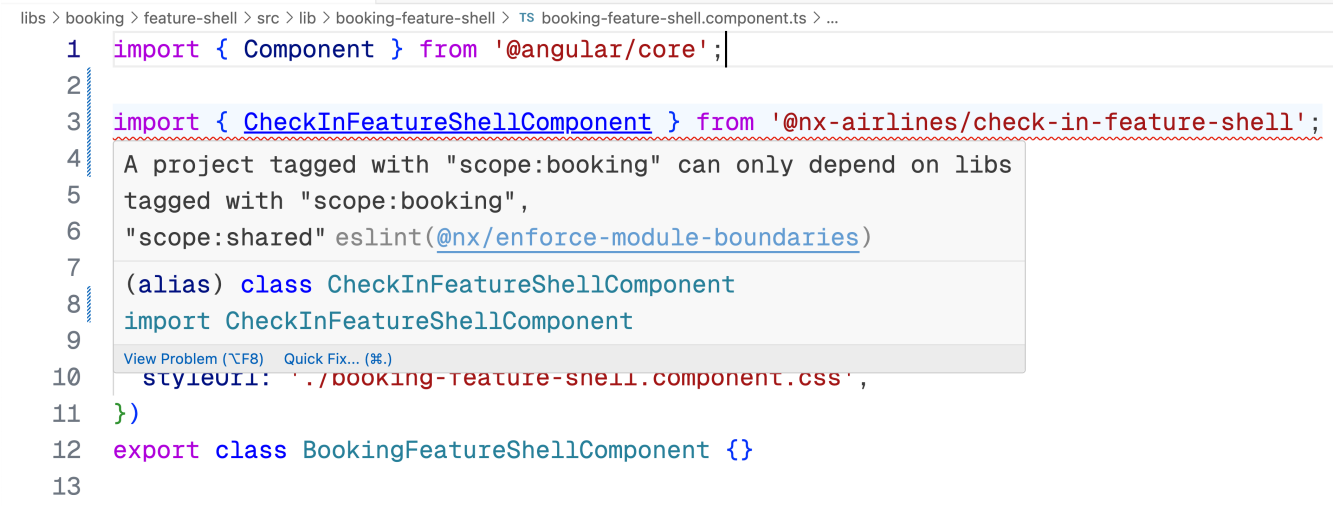


Figure 6. Error thrown when we try to add `CheckInFeatureShell` into `BookingFeatureShell`

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:

```
.`eslint.config.mjs`
"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:

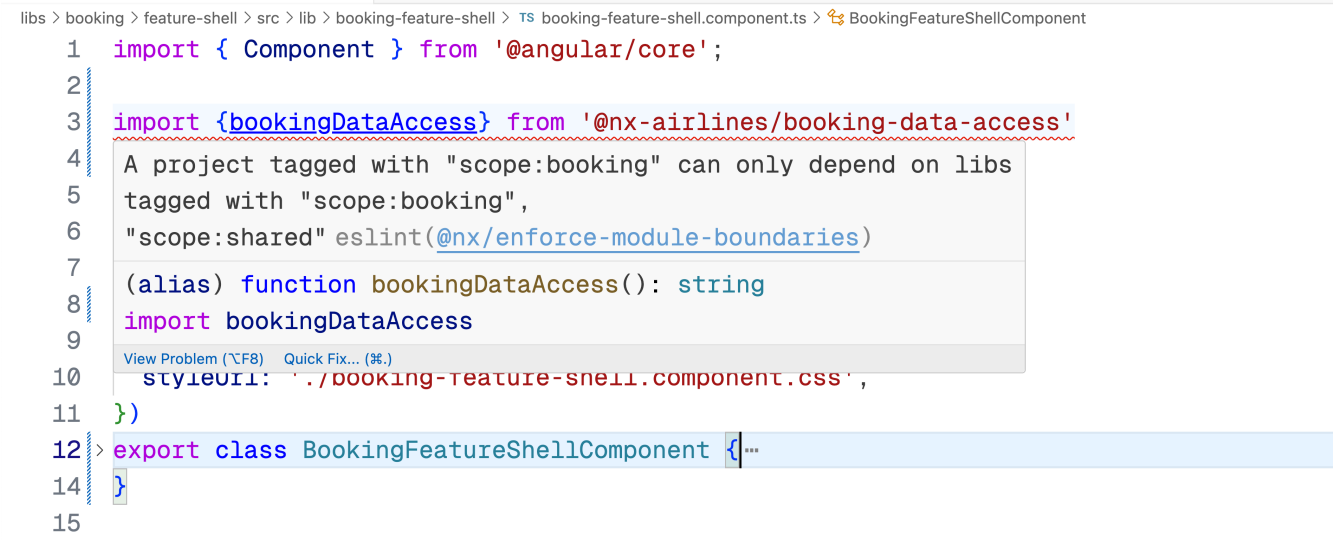


Figure 7. Error when we remove the wildcard rule

Using Workspace Generators

Generators 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 generators 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 generator would help here instead of having to do it by hand.

One of the great benefits of using a generator 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 generator without the `dry-run` to persist the changes.

Nx comes with some built-in generators.

Built-in Nx generators

The following generators are found in the `@nx/angular` collection.

- **application**: Creates a new Angular application.
- **component**: Creates a new Angular Component.
- **component-test**: Creates a cypress component test file for a component.
- **convert-to-application-executor**: Converts projects to use the `@nx/angular:application` executor or the `@angular-devkit/build-angular:application` builder.
- **directive**: Creates a new Angular directive.
- **federate-module**: Create a federated module, which is exposed by a remote and can be

subsequently loaded by a host.

- **library**: Creates a new Angular library.
- **library-secondary-entry-point**: Creates a secondary entry point for an Angular publishable library.
- **remote**: Generate a Remote Angular Module Federation Application.
- **host**: Generate a Host Angular Module Federation Application.
- **ngrx-feature-store**: Adds an NgRx Feature Store to an application or library.
- **ngrx-root-store**: Adds an NgRx Root Store to an application.
- **pipe**: Generate a new Angular Pipe.
- **scam-to-standalone**: Convert an existing Single Component Angular Module (SCAM) to a Standalone Component.
- **scam**: Generate a component with an accompanying Single Component Angular Module (SCAM).
- **scam-directive**: Generate a directive with an accompanying Single Component Angular Module (SCAM).
- **scam-pipe**: Generate a pipe with an accompanying Single Component Angular Module (SCAM).
- **setup-mf**: Generate a Module Federation configuration for a given Angular application.
- **setup-ssr**: Generate Angular Universal (SSR) setup for an Angular application.
- **setup-tailwind**: Configures Tailwind CSS for an application or a buildable/publishable library.
- **stories**: Creates stories/specs for all components declared in a project.
- **storybook-configuration**: Adds Storybook configuration to a project.
- **cypress-component-configuration**: Setup Cypress component testing for a project.
- **web-worker**: Creates a Web Worker.

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

Creating a custom generator

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` generator with this new lib name and pipe the other options to it.

Start by generating a new workspace generator.

```
npm install @nx/plugin
npx nx g @nx/plugin:plugin tools/generators/data-access-lib
npx nx g @nx/plugin:generator tools/generators/data-access-lib/src/new
```

This creates the following:

Output of creating a new generator

```
tools/
  generators/
    data-access-lib/
      src/
        files/
        new.ts ①
        schema.d.ts
        schema.json ②
```

① This is the main file of the generator.

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 generator).

The `new.ts` file has the following contents

```
import { addProjectConfiguration, formatFiles, generateFiles, Tree, } from
'@nx/devkit';
import * as path from 'path';
import { IndexGeneratorSchema } from './schema';

export async function indexGenerator( tree: Tree, options: IndexGeneratorSchema) {
  const projectRoot = `libs/${options.name}`;
  addProjectConfiguration(tree, options.name, {
    root: projectRoot,
    projectType: 'library',
    sourceRoot: `${projectRoot}/src`,
    targets: {},
  });
  generateFiles(tree, path.join(__dirname, 'files'), projectRoot, options); ①
  await formatFiles(tree);
}

export default indexGenerator;
```

① The provided example generator will create a new sample project based on the template in `files`

Let's provide our own implementation.

```
import { join } from 'node:path';
import { formatFiles, Tree } from '@nx/devkit';
import { libraryGenerator, ngrxFeatureStoreGenerator, } from '@nx/angular/generators';

import { IndexGeneratorSchema } from './schema';

export async function indexGenerator( tree: Tree, options: IndexGeneratorSchema,) {
  if (!options.name.startsWith('data-access-')) { ①
    throw new Error(`Data-access lib names should start with 'data-access-'`);
  }

  const libName = options.name.replace('data-access-', '');
  const projectRoot = `libs/${libName}`;

  await libraryGenerator(tree, { ②
    name: libName,
    directory: projectRoot,
    routing: true,
  });
  await ngrxFeatureStoreGenerator(tree, { ③
    name: options.name,
    minimal: false,
    directory: '+state',
    parent: join('libs', libName, 'src', 'lib', 'lib.routes.ts'),
  });

  await formatFiles(tree);
}

export default indexGenerator;
```

① Here is the check for the `data-access` prefix.

② First we create a new lib using `@nx/angular`.

③ Then we create the ngrx feature state scaffolding again with `@nx/angular`.

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

```
npx nx generate @nx-airlines/data-access-lib:new data-access-newlib2
```

This command is like other generators in Nx, we call the generator with our workspace npm scope, passing the name we'd like to use for our new lib, in this case `data-access-newlib2`.

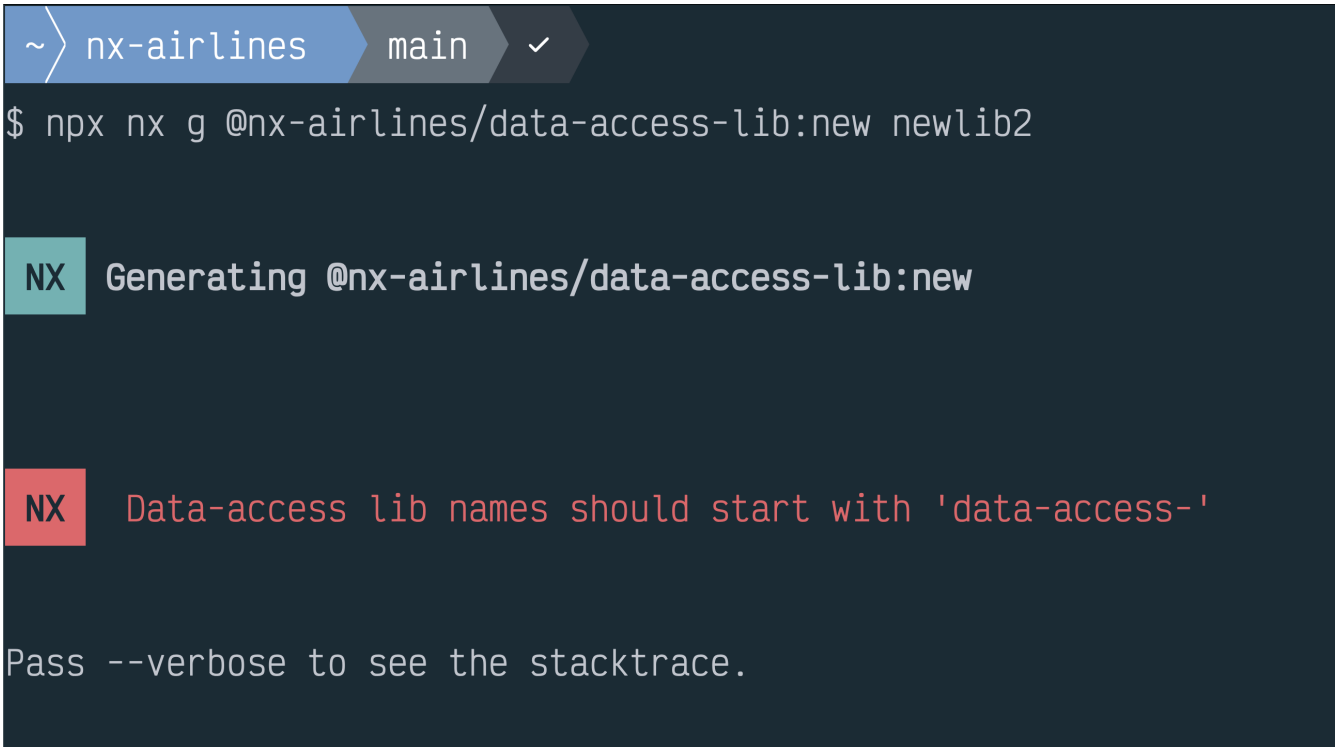


Figure 8. Output of running our new generator without specifying the prefix

```
~ nx-airlines main ✓
$ npx nx g @nx-airlines/data-access-lib:new data-access-newlib2

NX  Generating @nx-airlines/data-access-lib:new

CREATE libs/newlib2/project.json
CREATE libs/newlib2/README.md
CREATE libs/newlib2/tsconfig.json
CREATE libs/newlib2/tsconfig.lib.json
CREATE libs/newlib2/src/index.ts
CREATE libs/newlib2/src/lib/lib.routes.ts
CREATE libs/newlib2/jest.config.ts
CREATE libs/newlib2/src/test-setup.ts
CREATE libs/newlib2/tsconfig.spec.json
CREATE libs/newlib2/src/lib/newlib2/newlib2.component.css
CREATE libs/newlib2/src/lib/newlib2/newlib2.component.html
CREATE libs/newlib2/src/lib/newlib2/newlib2.component.spec.ts
CREATE libs/newlib2/src/lib/newlib2/newlib2.component.ts
CREATE libs/newlib2/eslint.config.mjs
UPDATE tsconfig.base.json
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.actions.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.effects.spec.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.effects.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.models.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.reducer.spec.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.reducer.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.selectors.spec.ts
CREATE libs/newlib2/src/lib/+state/data-access-newlib2.selectors.ts
```

Figure 9. Output of running our new generator successfully

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

Summary

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

- Nx enforces a Single Version Policy for 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 generators.

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 new code 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` command: `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)

Additional Options

- `--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 (nx-airlines). Its dependency graph is as below:

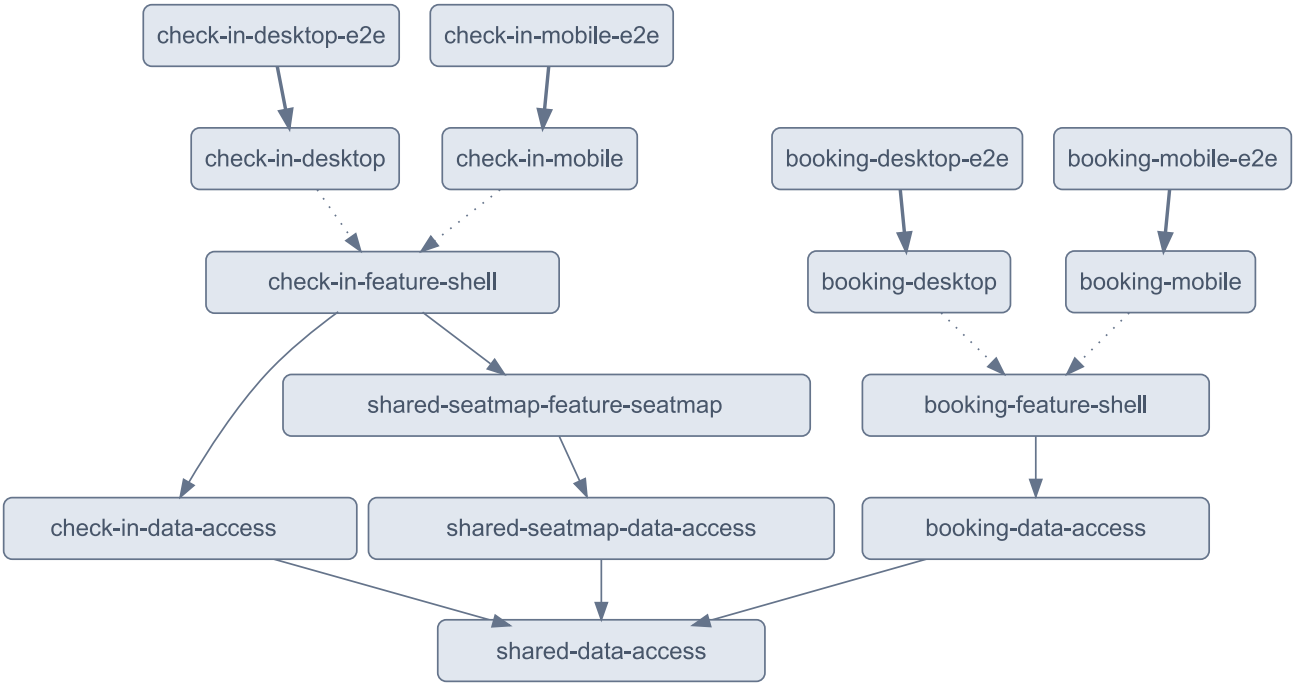


Figure 10. Original dependency graph

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`.

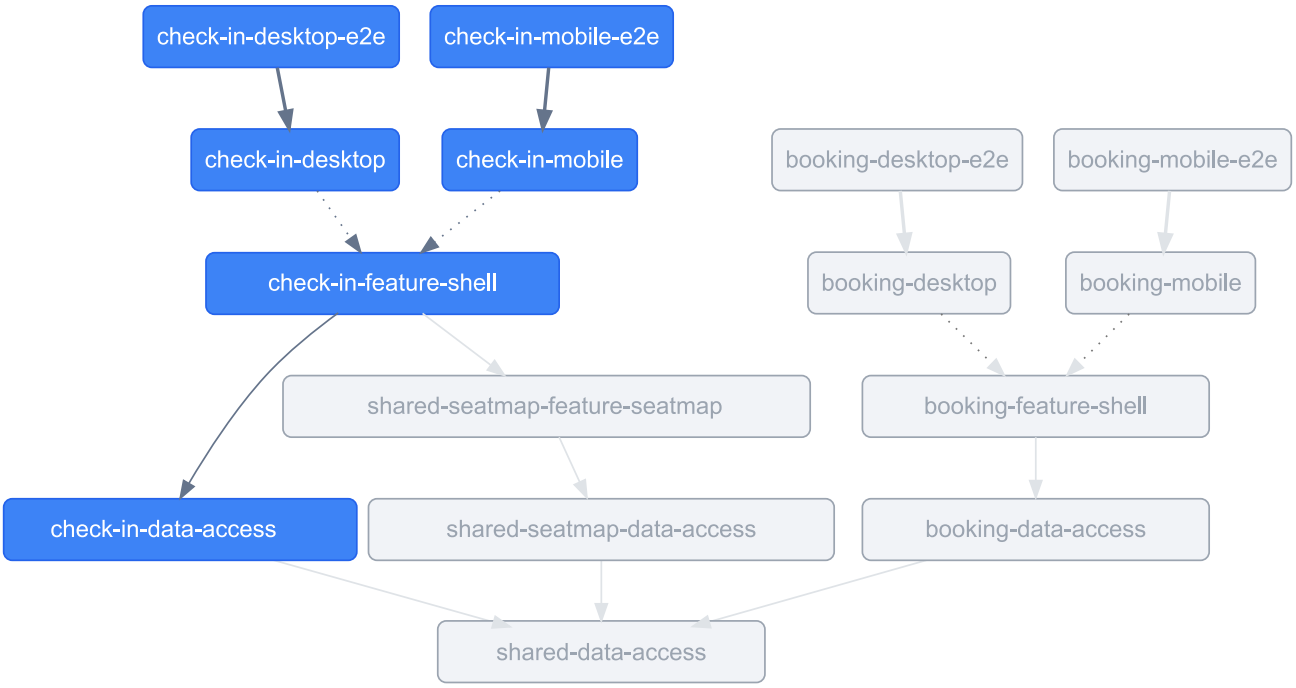


Figure 11. We don't have to build or test `booking` or `seatmap`!

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.

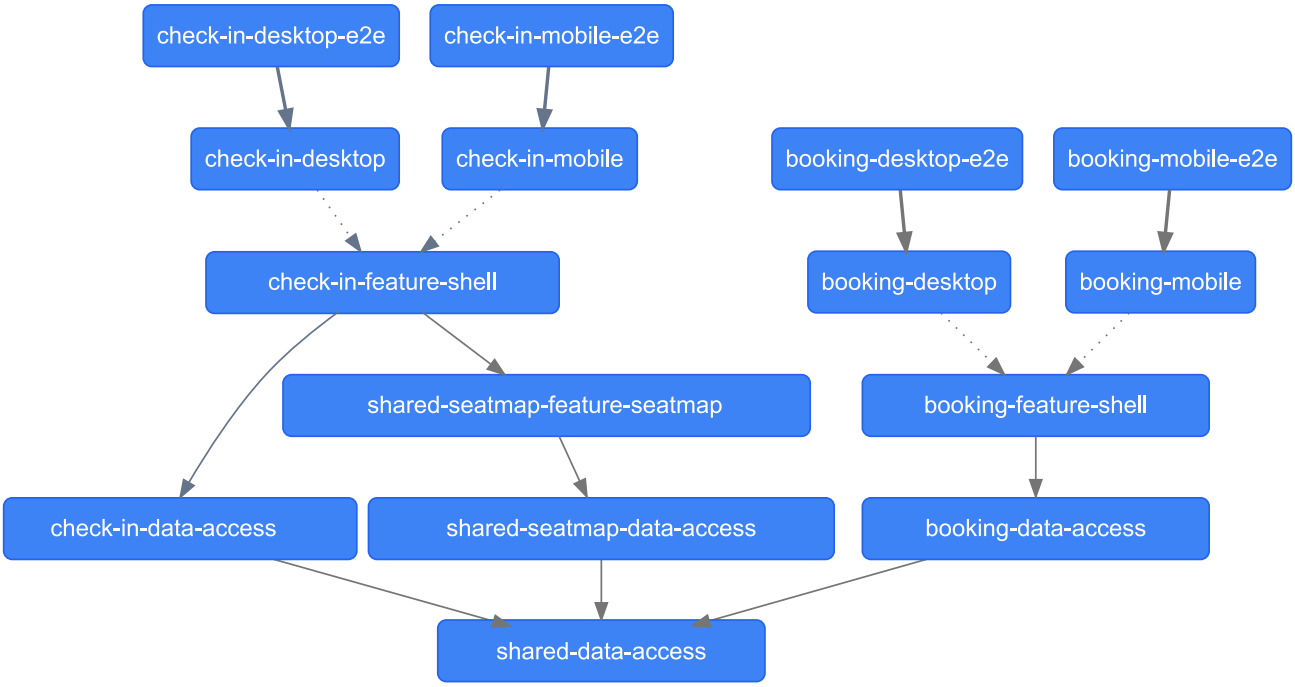


Figure 12. Changes made to the `shared-data-access` lib affect everything

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.

```
npx nx affected:build --base=master
npx nx affected:test  --base=master
npx nx affected:e2e   --base=master
```

We can also pass `--maxParallel` to limit the maximum number of parallel processes used.

```
npx nx affected:build -- --base=master --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:

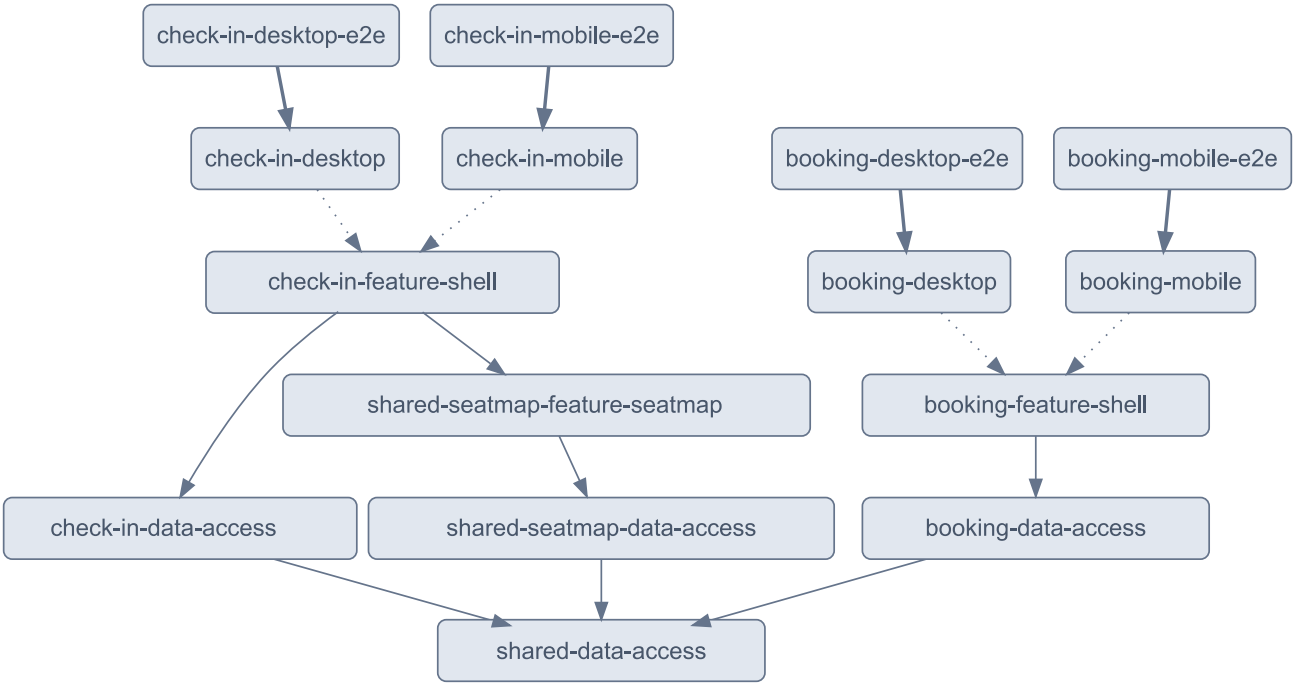


Figure 13. The example repo structure and dependencies

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.

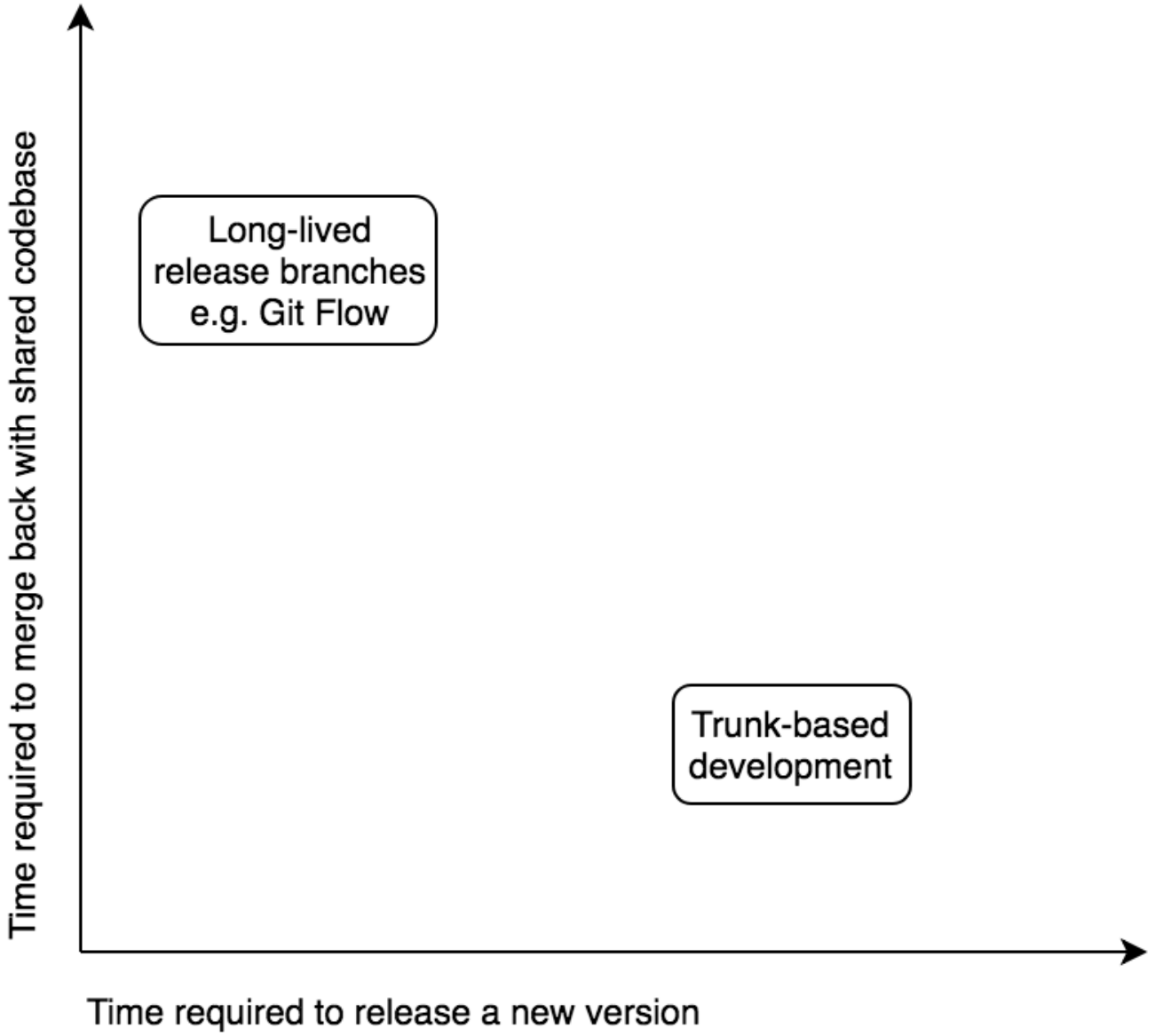


Figure 14. Git flow favours speed in releasing new artifacts over time to merge the code back

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

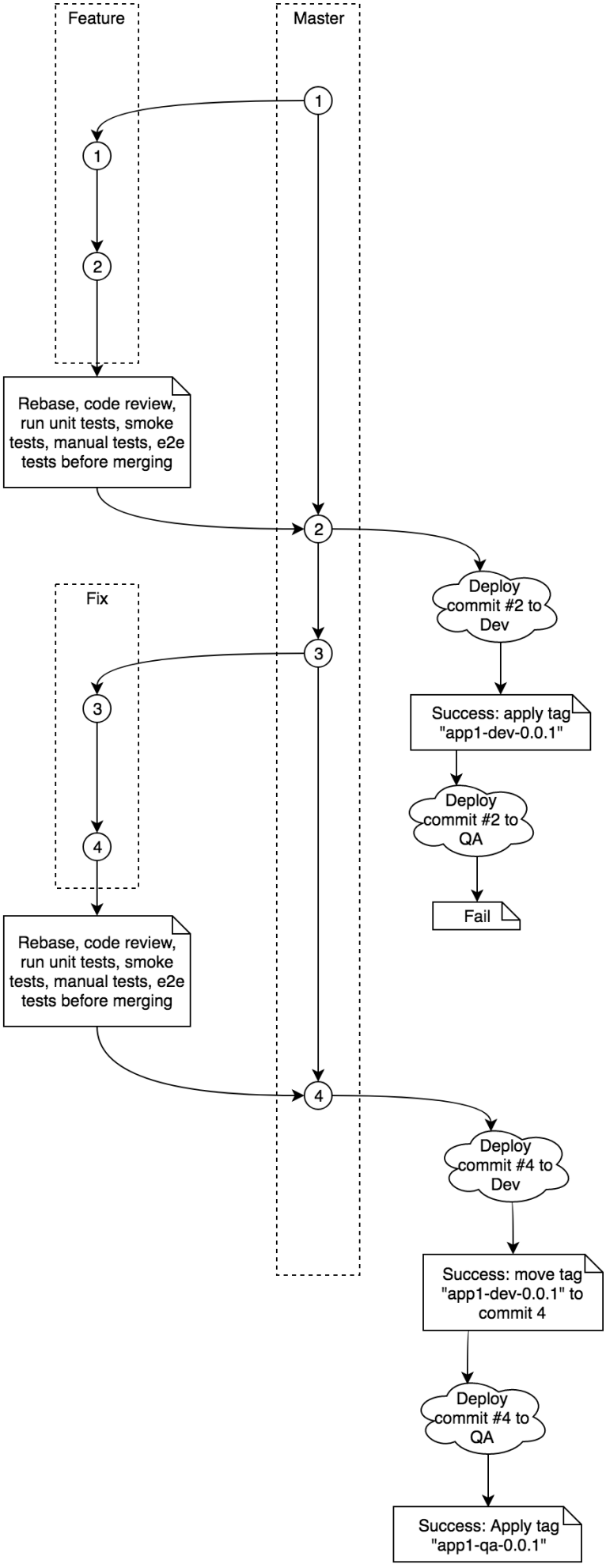


Figure 15. Trunk based Development

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 are up-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. The Nx Console allows developers to interact with Nx in a visual way. All the CLI options are visible and interactive, and commands can run in dry-run mode to preview what the results will be (the commands are executed using the Nx CLI under the hood).

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

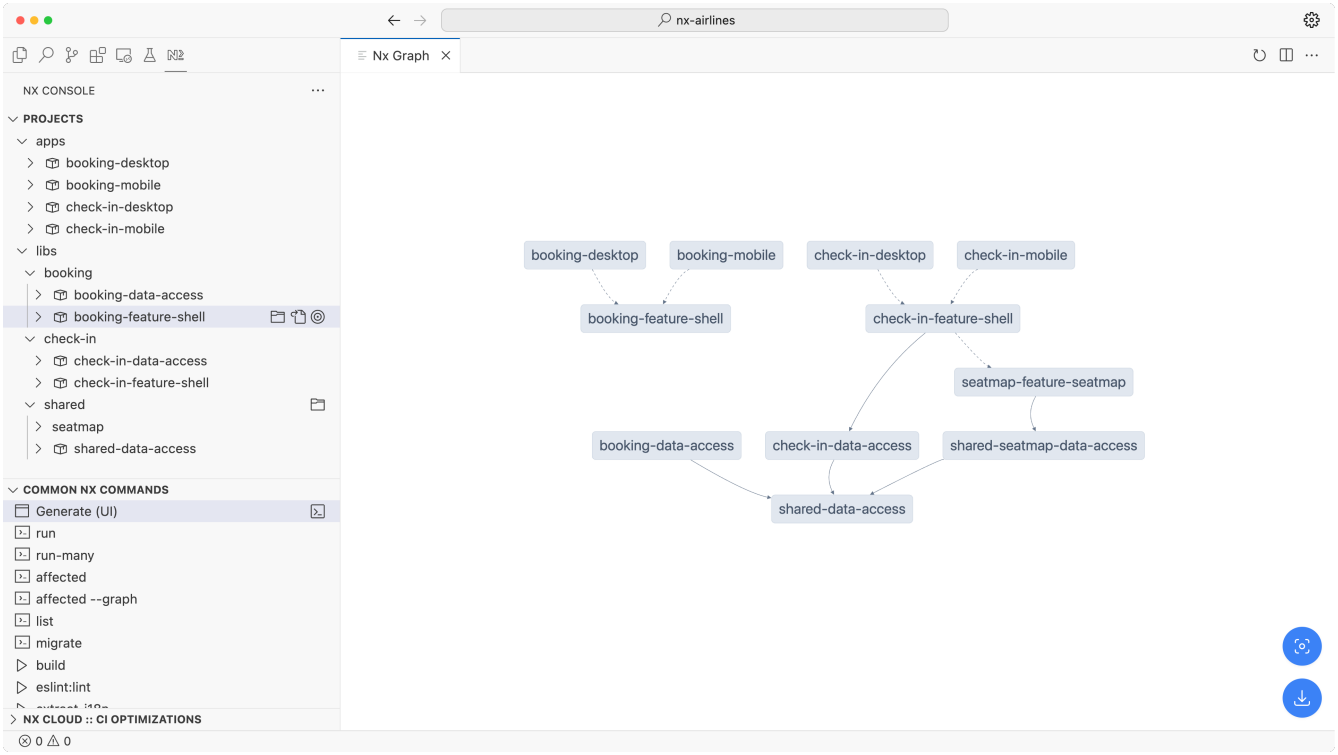


Figure 16. Nx Console

Features

Quick actions

Apps listed in the workspace view have quick-access buttons to serve, build, test or any other tasks defined in that app. Further actions are avaiable when you right click on the app.

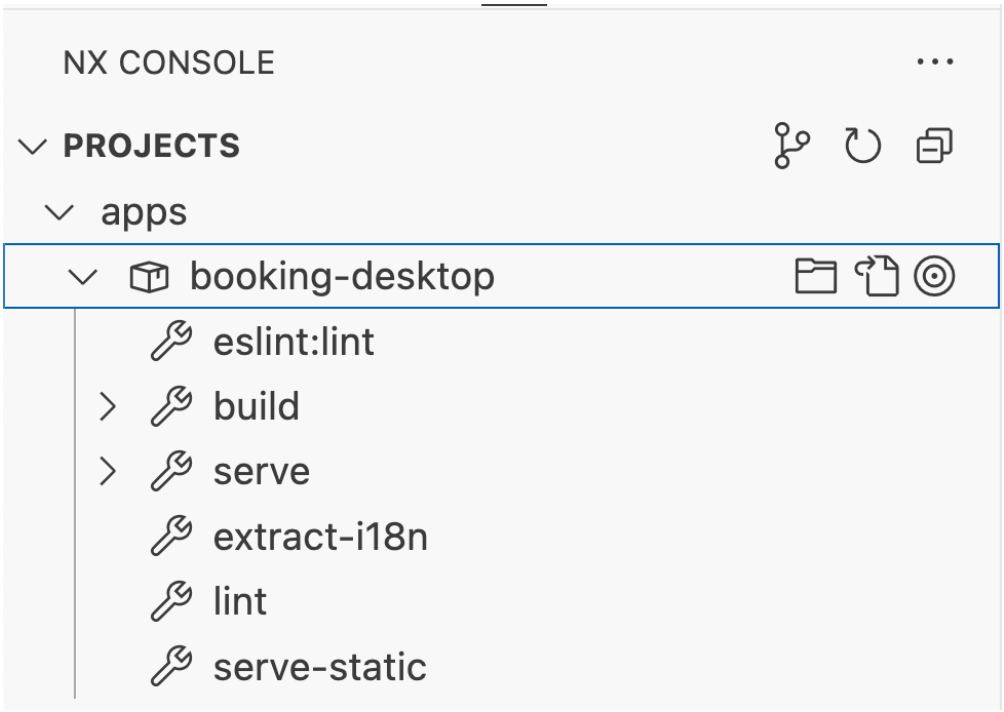


Figure 17. Application quick actions

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

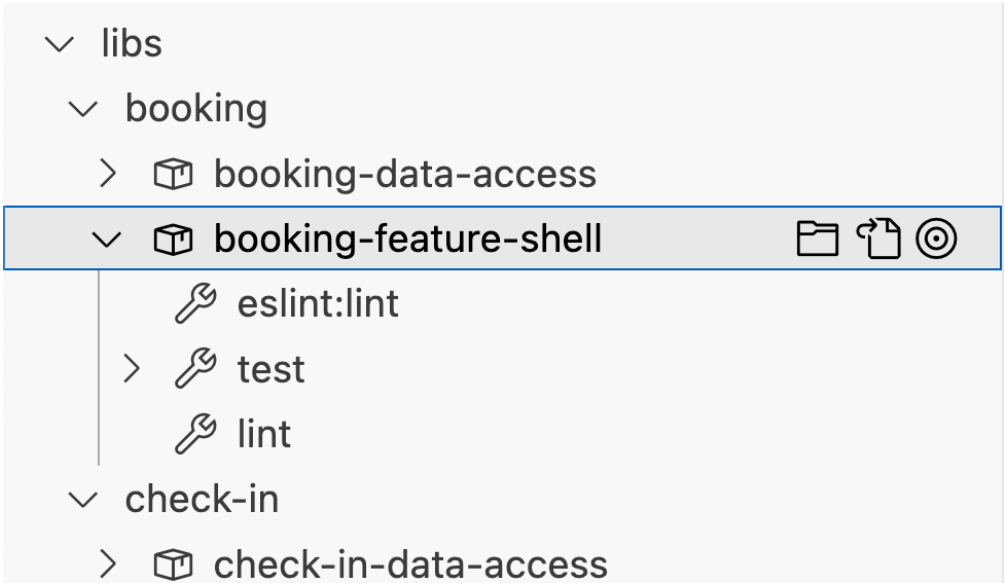


Figure 18. Library quick actions

Run commands

Nx Console can run any commands for projects in your workspace.

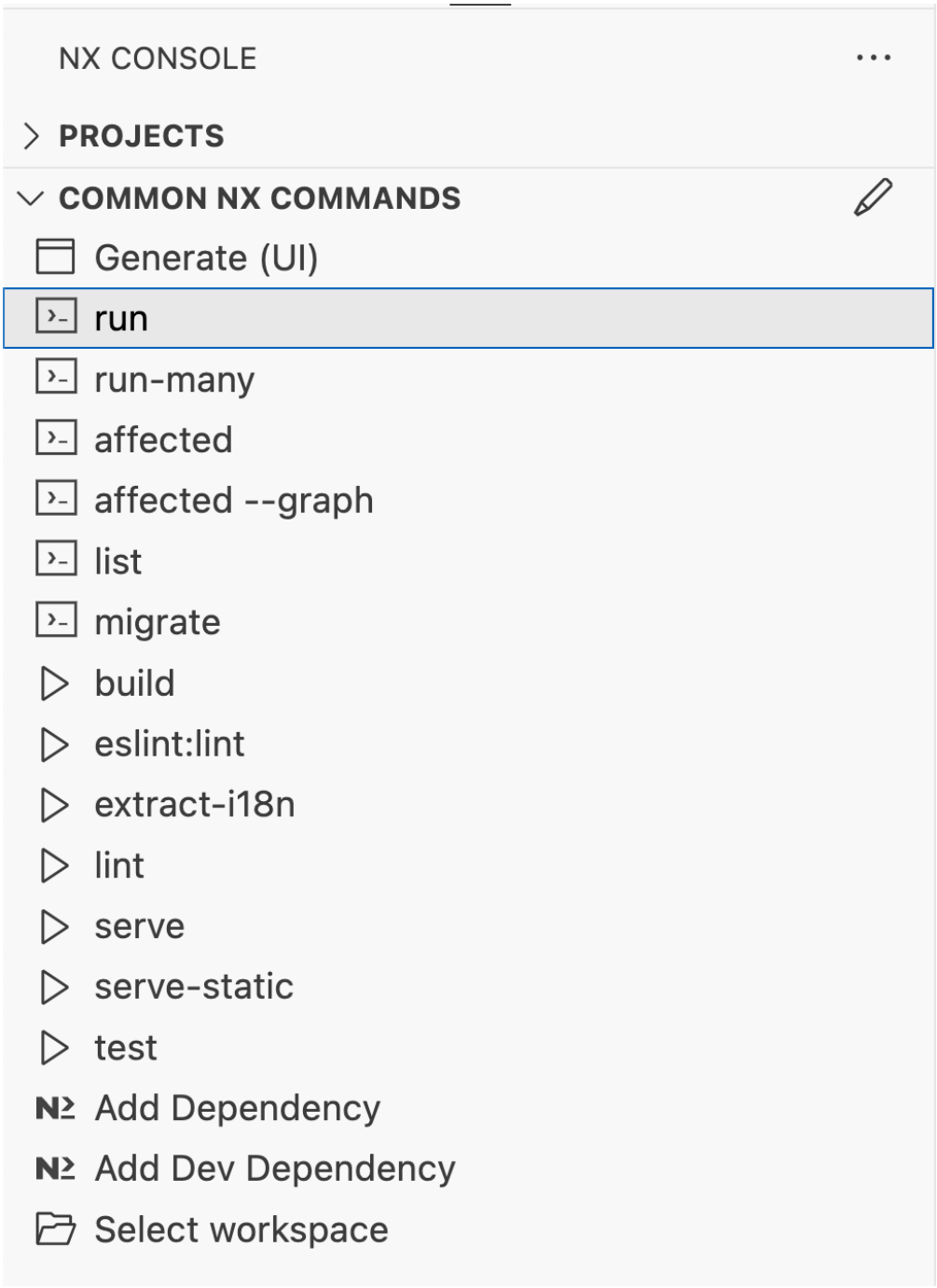
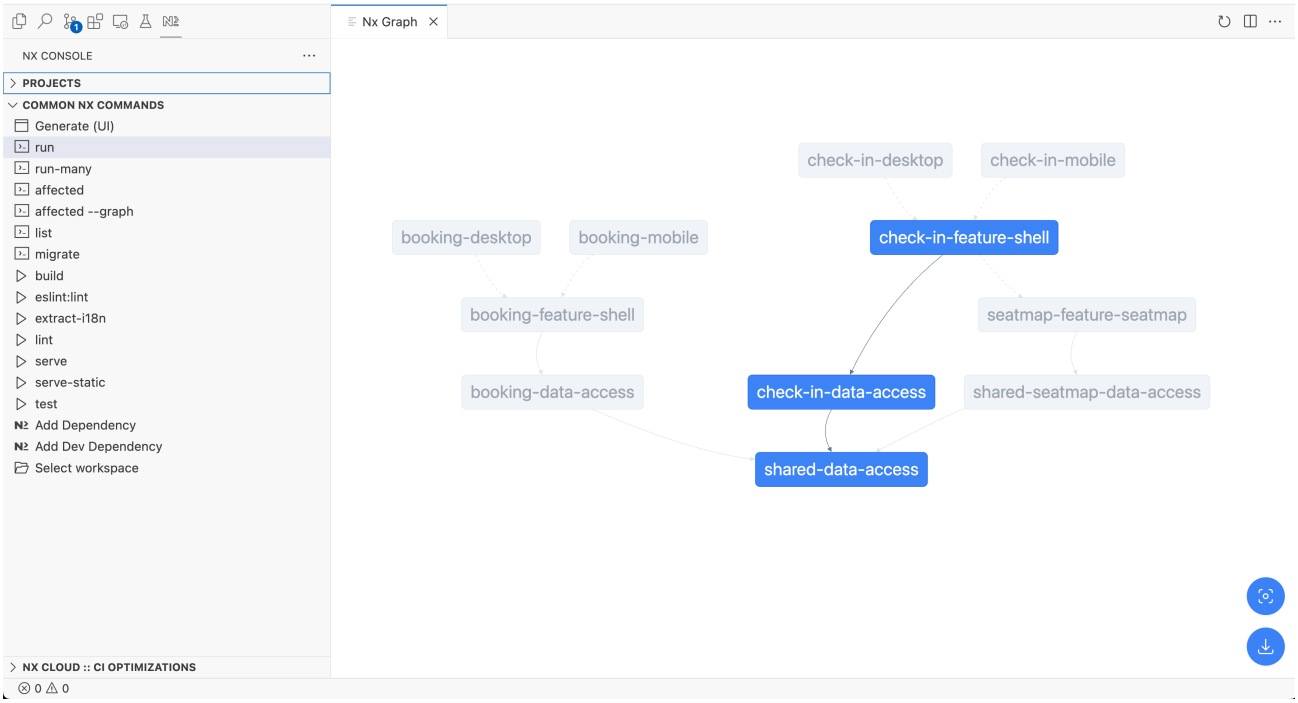


Figure 19. Run any task in package.json

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.

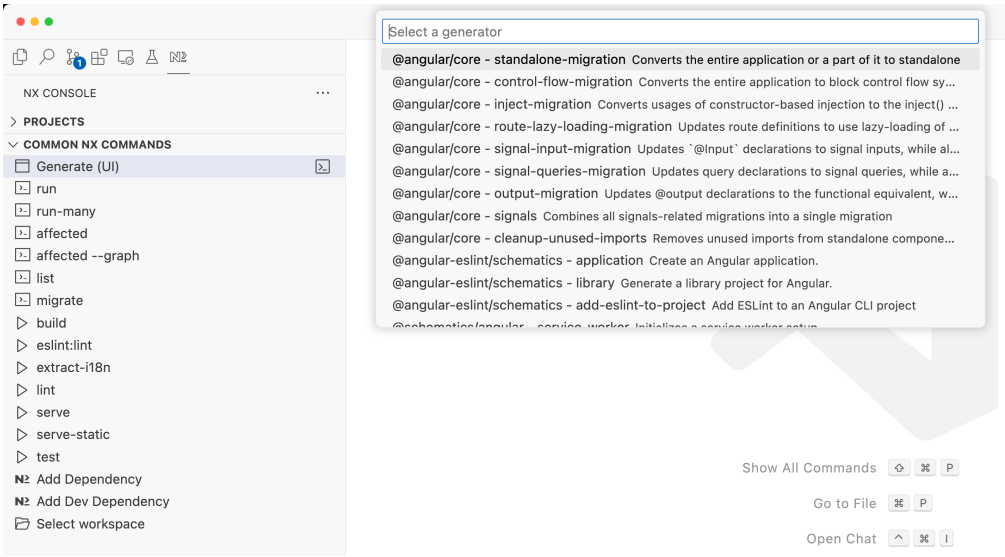


Figure 20. Code generation support

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.

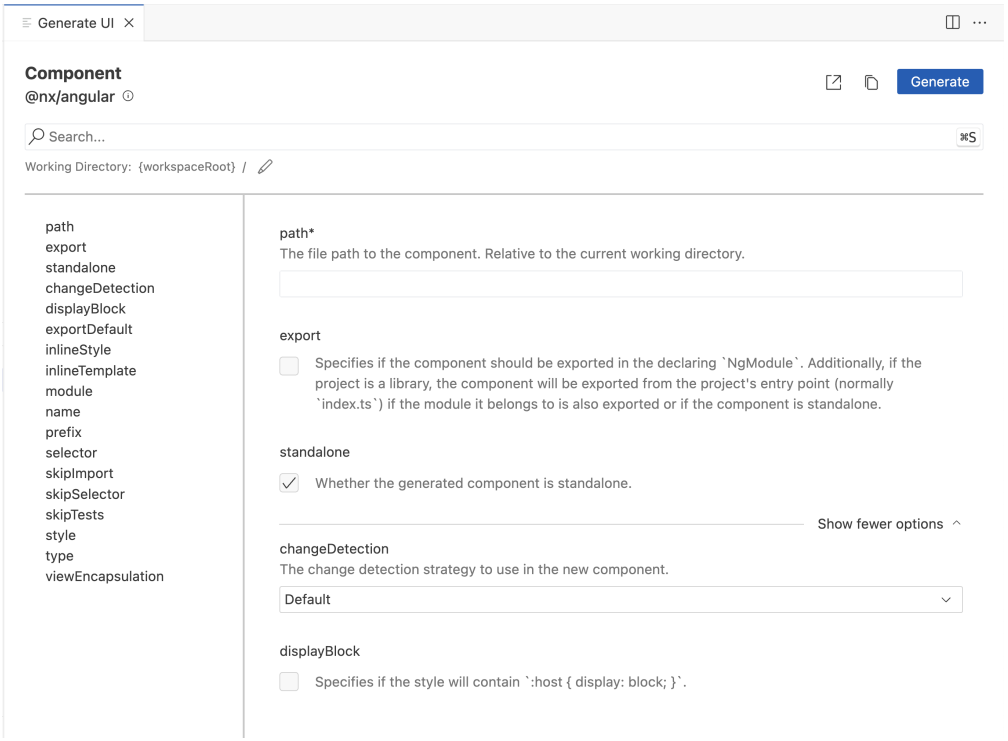


Figure 21. All options and expected output

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 Angular CLI

- build
- test
- lint
- e2e

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

Commands Provided By Nx

run

Nx can run various tasks that are defined in each project. These could be a build, lint, or other custom executors that are in your workspace. You can define what gets run by using:

```
npx nx run [project][:target][:configuration]
```

- `project` is the name of the project
- `target` is the name of the task defined in your `project.json`
- `configuration` is the configuration to use. Typically this could be production or development, but could be anything defined by the user.

run-many

Nx can perform run the same task across multiple projects in a workspace

```
npx nx run-many --target=build ①
npx nx run-many --target=test -p proj1 proj2 ②
npx nx run-many --target=test -p proj1 proj2 --parallel=false ③
```

① Run build for all the projects

② Run test for two projects in parallel

③ Run test for two projects in series

exec

Nx can run any scripts defined the `package.json` of a project or arbitrary scripts.

format

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

- `format` 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.

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.

`Affected` Commands

Affected commands accept a `--target` and the action you wish to run. Supported actions are below:

- affected --target=build
- affected --target=e2e
- affected --target=test
- affected --target=lint
- affected --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 20.0.0 and then decided to upgrade the version of Nx, the Nx migrate 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.

```
npx nx migrate latest
```

Where Should I Create My New Lib?

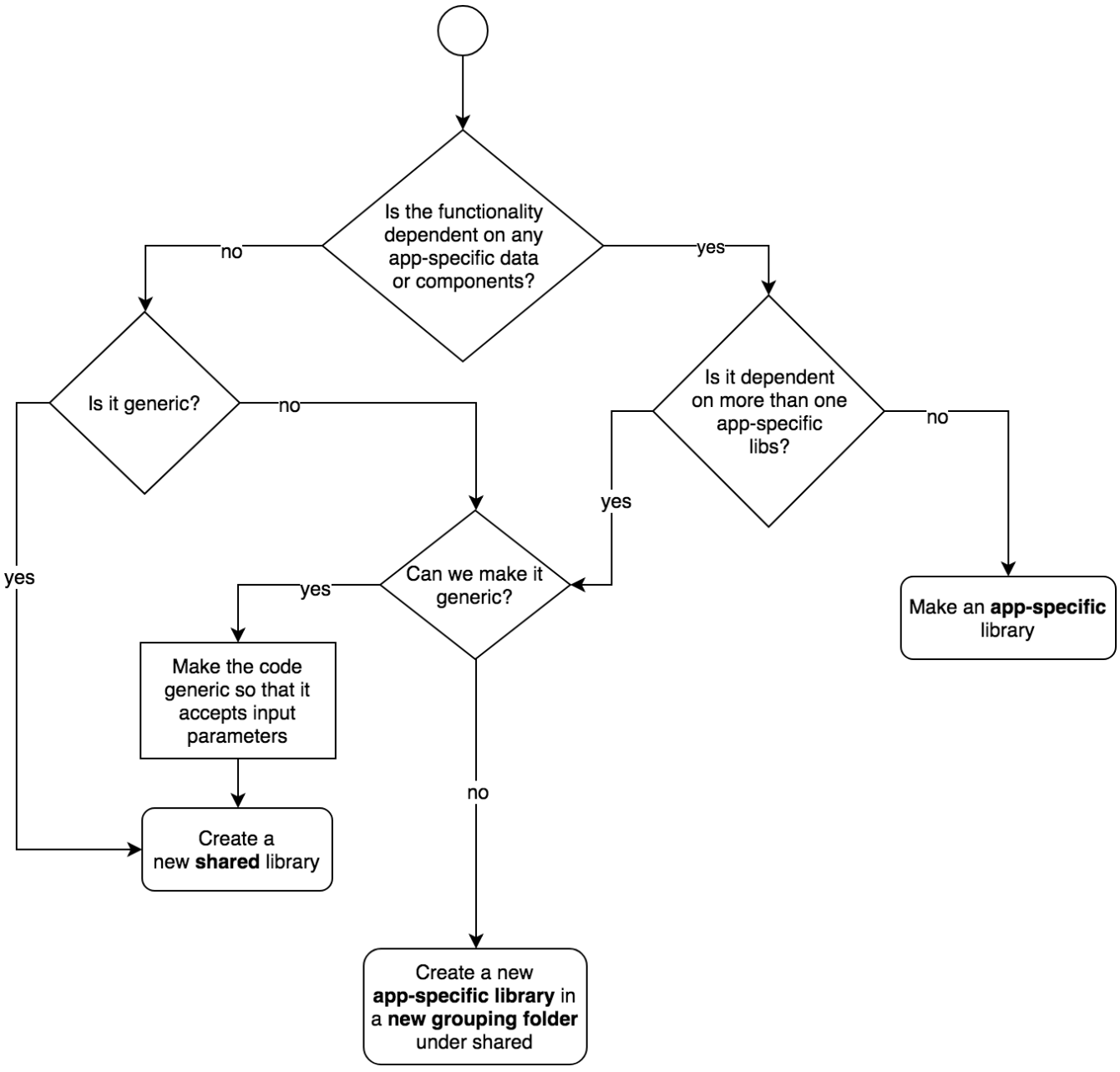


Figure 22. Shared vs. app-specific lib

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.

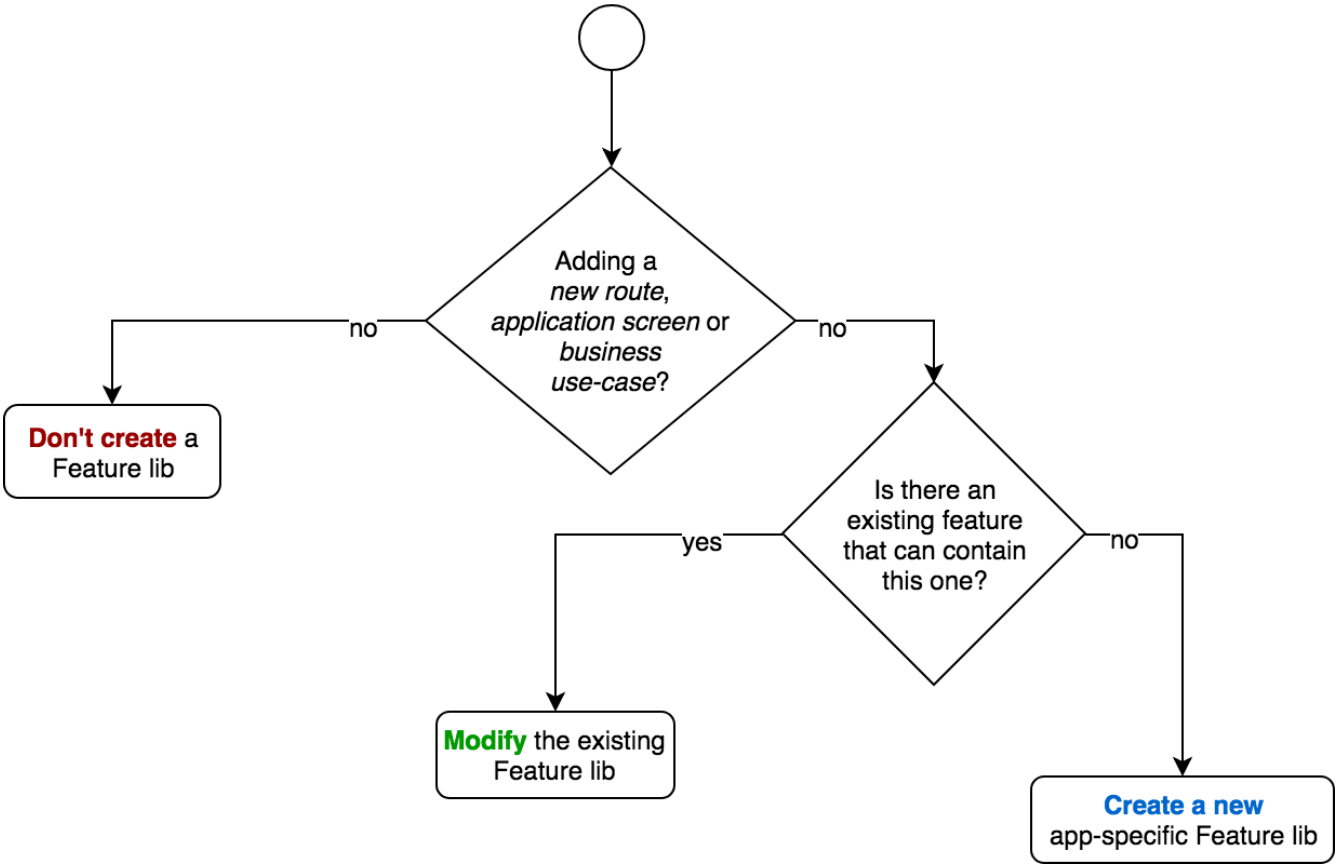


Figure 23. Should I create or modify a feature lib?

How Do I Extract a Feature Lib From an App?

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

- Run `npx nx g @nx/angular:lib libs/lib-name` (add `--routing`, `--lazy`, and `--parent` if needed)
- Copy the content of the app (everything except index.html, main.ts, app.config.ts, app.component.ts and other "global" files) into the lib.
- Update `app.component.ts` to remove all the imports that are no longer needed.
- Create an integration test that exercises the new module without requiring the containing app.

Index

Discover more resources from Nx

Read our blog: **nx.dev/blog** Use our open source products: **nx.dev/docs**. Use Nx Cloud for a fast, monorepo friendly CI platform **nx.dev/nx-cloud**

About Nx

Nx is a smart, fast, and extensible open source build system with first class monorepo support and powerful integrations. It was created and is maintained by a growing team of software development experts with extensive experience helping startups and Fortune 500's alike build better software through better developer experience and smarter devops.

Contact Us

marketing@nrwl.io