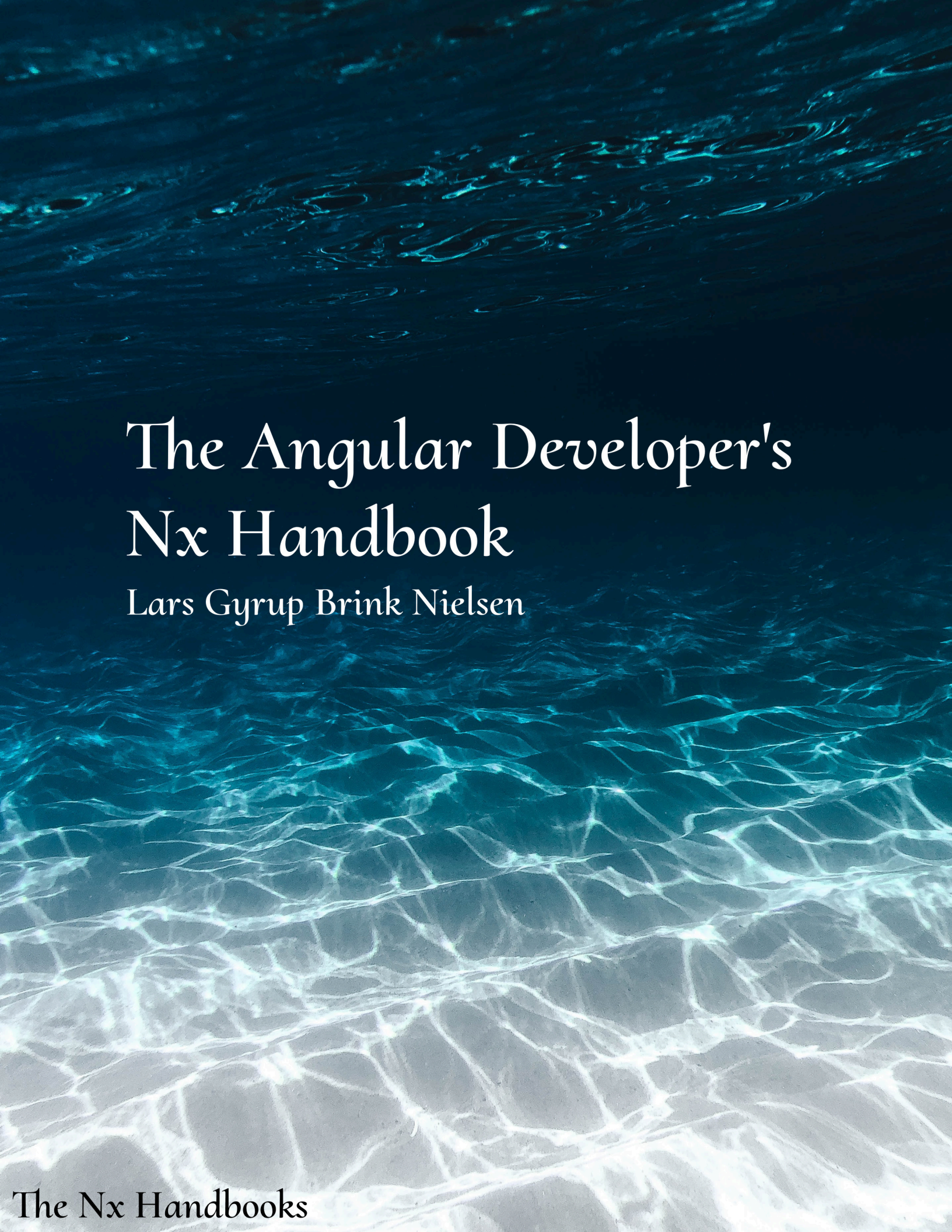


The Angular Developer's Nx Handbook

Lars Gyrup Brink Nielsen

The Nx Handbooks

The Angular Developer's Nx Handbook

Lars Gyrup Brink Nielsen

This book is for sale at http://leanpub.com/the-angular-developers-nx-handbook

This version was published on 2022-07-16

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Lars Gyrup Brink Nielsen by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm looking forward to read "The Angular Developer's Nx Handbook" by @LayZeeDK #TheNxHandbooks

The suggested hashtag for this book is #TheNxHandbooks.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#TheNxHandbooks

Contents

Introduction

This free book about the Nx build framework targets Angular developers. It assumes familiarity with the Angular CLI to use common concepts and compare them to equivalents or replacements included in the Nx CLI.

The Angular Developer's Nx Handbook is a curated guide to essential Nx concepts, principles, and features. We learn to use Nx to build features for a real-world Angular application.

We start our Nx Angular journey by generating a workspace from scratch. After exploring the contents of a freshly generated Nx workspace, we go through the Nx CLI commands that are similar to or replacements for Angular CLI commands. Next, we discuss the unique commands offered by the Nx CLI.

Following Nx CLI commands, we learn about the important Nx library concepts *library traits* and *library types*.

With an understanding of Nx library concepts in place, we start work on Gitropolis, a GitHub clone using GitHub's Primer design system, the GitHub REST API via Octokit, RxAngular for state management, an improved developer experience, and optimized rendering.

Creating an Nx workspace

In this chapter, we learn how to create an Nx workspace. In particular, we cover these topics:

- Choosing a package manager
- Installing and using the create-nx-workspace command
- Using the Angular workspace preset
- Specifying workspace options
- Creating an Angular CLI workspace for comparison

Time to start our Nx Angular adventure!

Using the create-nx-workspace command

To create an Nx workspace, instead of using the `ng new` command, we use the `create-nx-workspace` package.

First, we choose a package manager. We run the `create-nx-workspace` command in different ways depending on our package manager.

```
# npm
npm init nx-workspace <workpace-directory-name> --package-manager=npm

# Yarn
yarn create nx-workspace <workpace-directory-name> --package-manager=yarn

# pnpm
pnpm init nx-workspace <workpace-directory-name> --package-manager=pnpm
```

We use the following command to create an Nx workspace using Yarn as our package manager:

```
# Create the Gitropolis workspace using Yarn
yarn create nx-workspace nx-angular-gitropolis --package-manager=yarn \
  --preset=angular --app-name=gitropolis-app --npm-scope=gitropolis \
  --style=scss --no-nx-cloud
```

This creates an Nx workspace in the `nx-angular-gitropolis` directory using the `angular` workspace preset. The `create-nx-workspace` command generates an Angular application named `gitropolis-app` and a matching Cypress end-to-end testing project. The TypeScript import alias namespace used for libraries is `@gitropolis` and the application uses `scss` syntax for stylesheets. For this book, we opt out of Nx Cloud using the `--no-nx-cloud` parameter.

Creating an Angular CLI workspace for comparison

To create a similar workspace using the Angular CLI, the following commands can be used. Feel free to try it out yourself to compare the toolchains:

```
# Create an Angular CLI workspace
npx @angular/cli new angular-gitropolis --package-manager=yarn \
  --no-create-application
# Enter the workspace
cd angular-gitropolis
# Generate an application project in the projects directory
npx ng generate application gitropolis-app --no-routing --style=css
```

In the chapter *Exploring a fresh Nx Angular workspace*, we discuss the differences between Nx Angular and Angular CLI workspaces.

Conclusion

You now have an Nx workspace in a directory named `nx-angular-gitropolis` which we generated using the `create-nx-workspace` command. Which package manager did you choose? In this book, I demonstrate how to use Yarn Classic–that is version 1.x–but feel free to pick your preference and figure out any differences in terminal commands on your own.

We used the Angular workspace preset by passing the argument `angular` to the `--preset` parameter. In the following chapter, we explore all the generated directories and files, then we discuss how they differ from a workspace generated by the Angular CLI.

We specified other workspace options using the parameters `--app-name`, `--npm-scope`, and `--style`. We also opted out of Nx Cloud by specifying the `--no-nx-cloud` parameter.

> **Note:** If you want the full Nx experience, you should use the Nx Cloud. It comes with 500 free hours of compute time per month. While Nx Cloud is a significant benefit of using Nx, it is not covered in this book.

Finally, you had the choice to generate a workspace using the Angular CLI to compare the workspaces side-by-side. Don't worry, you can proceed to the next chapter without generating an Angular CLI workspace but it's interesting to inspect and compare on your own while following along.

Exploring a fresh Nx Angular workspace

In this chapter, we explore the generated files and directories in our Nx workspace on a high level.

First, we compare top-level files, including configurations for:

- EditorConfig
- Git
- TypeScript
- Prettier
- ESLint
- Jest
- Nx

To improve our productivity and keep a consistent style, we configure code generation defaults in the Nx CLI configuration.

Next, we explore top-level directories in our Nx workspace and compare them to what we know from an Angular CLI workspace. This includes locations for:

- Visual Studio Code configurations
- Application projects
- End-to-end testing projects
- Library projects
- Custom Nx plugins
- Custom scripts

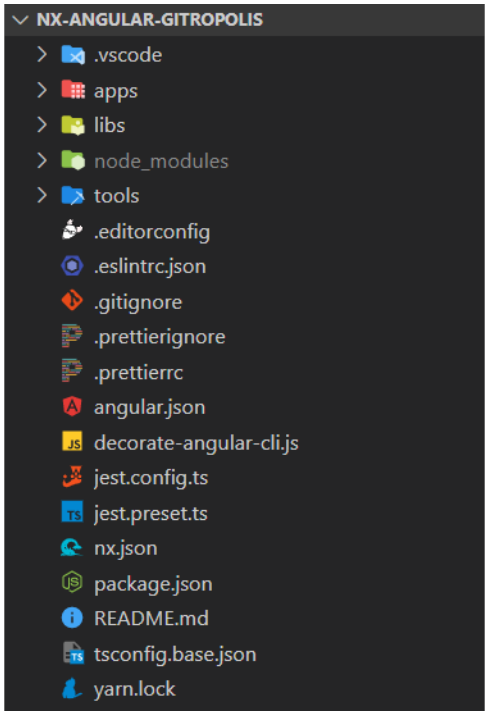
We also keep a consistent style by configuring automatic organization of import statements when source code is saved.

Finally, we introduce solution-style TypeScript configurations and their benefits since this configuration style is used by the Nx CLI but not the Angular CLI.

Comparing top-level files

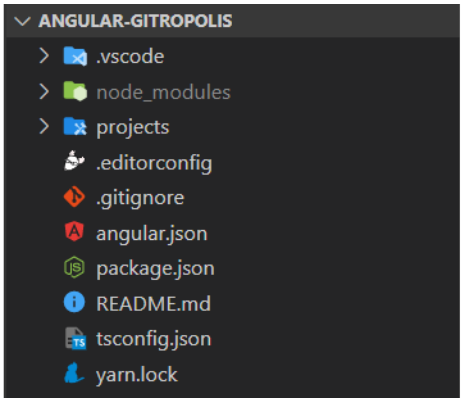
Let us compare a freshly generated Nx Angular workspace to a freshly generated Angular CLI workspace with one application.

A freshly generated Nx Angular workspace looks like this image:



Freshly generated Nx Angular workspace

In comparison, a freshly generated Angular CLI workspace looks like the following image:



Freshly generated Angular CLI workspace

Familiar files

Considering the root workspace files first, we see that they have the following files in common:

- `.editorconfig`: Polyeditor configuration file
- `.gitignore`: Ignore patterns for Git
- `angular.json`: Workspace configuration
- `package.json`: Package file with scripts, dependencies, and various metadata
- `README.md`: The workspace readme file
- `yarn.lock`: The package manager lockfile

The name and content of the lockfile (`yarn.lock`) vary depending on the package manager we are using.

TypeScript configurations

The Nx Angular workspace has a `tsconfig.base.json` file while the Angular CLI workspace has a `tsconfig.json` file. As you might know, they are configuration files for the TypeScript compiler.

Compared to the Angular CLI's TypeScript compiler configuration, Nx's configuration has fewer TypeScript compiler settings and at first glance does not have settings for the Angular compiler, namely the well-known `angularCompilerOptions` object.

The Angular-specific settings, as well as strict settings, are not added to the base TypeScript configuration by Nx's Angular workspace preset. As it turns out, they are added to project-specific TypeScript configurations for applications and libraries.

Nx uses solution-style TypeScript configurations which were introduced to Angular CLI 10.0 with TypeScript 3.9 but rolled back in Angular CLI 10.1. Learn more about this TypeScript configuration style in the section *Introducing solution-style TypeScript configurations* in this chapter.

Additional files

Most of the other top-level files are related to tools that Nx adds by default:

- Prettier for formatting: `.prettierignore` and `.prettierrc`
- ESLint and Angular ESLint for linting: `.eslintrc.json`
- Jest for unit tests: `jest.config.ts` and `jest.preset.ts`

Prettier

Prettier is an open-source tool that normalizes file formatting. Prettier formats files in our workspace based on settings in the `.editorconfig` and `.prettierrc` configuration files. In Nx workspaces, it is used through the `nx format` commands. Read the chapter *Increasing productivity with the Nx CLI's unique commands* to learn how to use these commands.

ESLint

The open-source ESLint plugin Angular ESLint adds Angular-specific lint rules to ESLint. Nx's Angular workspace preset preinstalls ESLint, TypeScript ESLint, and Angular ESLint with recommended plugins and linting presets.

> **Note:** As you might know, Angular CLI doesn't come with a linter preinstalled since support for the end-of-life tool TSLint was fully removed. However, if we try to use the `ng lint` command in an Angular CLI workspace, we are prompted to install Angular ESLint.

Jest

While Nx has built-in support for using Google's Karma test runner and implicitly the Jasmine test framework for unit tests, the open-source test runner and test framework Jest governed by The OpenJS Foundation is the default choice for Nx projects.

ng command delegation

Our Nx workspace contains a file with the interesting name `decorate-angular-cli.js`. This file runs as part of the `postinstall` hook listed in `package.json`. It delegates all supported `ng` Angular CLI commands to the Nx CLI executable, `nx`. When your muscle memory makes you write `ng generate component` and other common commands, the Nx CLI will be called instead of the Angular CLI.

> **Note:** The `decorate-angular-cli.js` file is a Node.js script. You do not have to read or understand it but it includes a description of its features with instructions for disabling the patch entirely.

Workspace configuration

Before looking at `nx.json`, the final file added to the workspace root by Nx, let us open `angular.json` in our Nx Angular workspace:

```
{
  "version": 2,
  "projects": {
    "gitropolis-app": "apps/gitropolis-app",
    "gitropolis-app-e2e": "apps/gitropolis-app-e2e"
  }
}
```

Wow! Being familiar with the Angular CLI, I guess that's not what you expected. Where are all the project and CLI settings?

The `"version": 2` setting signifies the workspace schema. The Angular CLI currently uses the workspace version 1 schema, while the Nx CLI uses version 2. The supported settings are similar but by default, Nx uses *standalone project configurations*. This means, that what we see in the `projects` object is a collection of project names mapped to their paths in the workspace.

The project-specific settings are located in `project.json` files at the root of each project folder. For example, the `project.json` file in the `apps/gittropolis-app` directory has content in the following format which should mostly look familiar:

```
{
  "projectType": "application",
  "root": "apps/gitropolis-app",
  "sourceRoot": "apps/gitropolis-app/src",
  "prefix": "gitropolis",
  "targets": {
    "build": {
      "//": "(...)"
    },
    "serve": {
      "//": "(...)"
    },
    "extract-i18n": {
      "//": "(...)"
    },
    "lint": {
      "//": "(...)"
    },
    "test": {
      "//": "(...)"
    }
  },
  "tags": []
}
```

Later in this book, we discuss the differences between these settings and the settings we know from an Angular CLI workspace configuration.

Nx CLI configuration

Now, you might be wondering: Where are the CLI settings? In Angular CLI workspaces, they are in the `angular.json` file. In Nx workspaces, they are in `nx.json`:

```
{
  "npmScope": "gitropolis",
  "affected": {
    "defaultBase": "main"
  },
  "cli": {
    "defaultCollection": "@nrwl/angular",
    "packageManager": "yarn"
  },
  "implicitDependencies": {
    "//": "(...)"
  },
  "tasksRunnerOptions": {
    "//": "(...)"
  },
  "targetDependencies": {
    "//": "(...)"
  },
  "generators": {
    "@nrwl/angular:application": {
      "style": "css",
      "linter": "eslint",
      "unitTestRunner": "jest",
      "e2eTestRunner": "cypress"
    },
    "@nrwl/angular:library": {
      "linter": "eslint",
      "unitTestRunner": "jest"
    },
    "@nrwl/angular:component": {
      "style": "css"
    }
  }
}
```

The `cli` settings might look familiar. The `generators` object we know as `schematics` in Angular CLI workspace configurations. Notice that Nx comes with its own set of Angular *generators* which is what *schematics* are called in Nx workspaces. The other settings are specific to Nx. We discuss them later in this book.

In this workspace, we create single-file standalone Angular components that use the `OnPush` change detection strategy. Let us configure our generator defaults so that we only have to specify parameters that are unique to a component when generating them.

As this workspace only has a single application project, let us make it the default project for Nx CLI commands by adding the following setting to the `nx.json` workspace configuration file:

```
{
  "defaultProject": "gitropolis-app"
}
```

To prefer Nx Angular generators over the Angular CLI's generators, we configure the `cli.defaultCollection` setting in `nx.json`:

```
{
  "cli": {
    "defaultCollection": "@nrwl/angular"
  }
}
```

Add the following settings to the `generators["@nrwl/angular:component"]` configuration:

```
{
  "generators": {
    "@nrwl/angular:component": {
      "style": "scss",
      "changeDetection": "OnPush",
      "displayBlock": true,
      "flat": true,
      "inlineStyle": true,
      "inlineTemplate": true,
      "skipTests": true,
      "standalone": true
    }
  }
}
```

In the chapter *Creating an Nx workspace*, we specified the `--style=scss` argument when creating the workspace so the `"style": "scss"` setting should already be present.

Now, let us also configure code generation defaults for generating standalone Angular directives and pipes by adding the following settings to the `nx.json` file:

```
{
  "generators": {
    "@schematics/angular:directive": {
      "skipTests": true,
      "standalone": true
    },
    "@schematics/angular:pipe": {
      "skipTests": true,
      "standalone": true
    }
  }
}
```

In addition to making standalone Angular declarables, we skip test generation since we will only be covering a few classes and features with tests throughout this book.

We also skip test generation for services by adding these settings:

```
{
  "generators": {
    "@schematics/angular:service": {
      "skipTests": true
    }
  }
}
```

Finally, we configure similar settings for application projects for the sake of consistency:

```
{
  "generators": {
    "@nrwl/angular:application": {
      "style": "scss",
      "inlineStyle": true,
      "inlineTemplate": true,
      "skipTests": true
    }
  }
}
```

Comparing top-level directories

Now that we have taken a brief look at all the files in the root of our Nx Angular workspace, let us discuss the generated top-level directories.

Visual Studio Code

The `.vscode` directory contains Visual Studio Code (VS Code) settings. This directory is also part of Angular CLI-generated workspaces although The Angular workspace preset for Nx comes with a few more recommended VS Code extensions, most importantly *Nx Console* with the extension ID `nrwl.angular-console` since early versions of Nx only supported Angular projects.

> **Note:** In addition to Nx Console, make sure to install the recommended Prettier and ESLint extensions for VS Code to get the best developer experience.

To configure VS Code to prefer the TypeScript version installed in the workspace, we create a `settings.json` file in the `.vscode` directory with the following contents:

```
{
  "typescript.tsdk": "./node_modules/typescript/lib"
}
```

This increases the accuracy of TypeScript suggestions and syntax error detection when editing source code.

Additionally, we add the following settings to the `setting.json` file to organize imports on save:

```
{
  "editor.formatOnSave": true,
  "[typescript]": {
    "editor.codeActionsOnSave": {
      "source.organizeImports": true
    }
  }
}
```

This increases consistency in our codebase and decreases the number of changes and merge conflicts because of different editor settings across developer machines. ESLint plugins to manage this are available but more complicated to configure so we configure VS Code in this book instead.

Custom Nx plugins and scripts

The `tools` directory is where we locally develop and use workspace-specific generators and *executors* which we know as *builders* in Angular CLI workspaces. Nx has generators for generating local generators and executors. Say that again five times, fast!

The `tools` directory is also a decent location to place various Node.js scripts that support our workspace.

Library projects

By now, you must be wondering what is in the `libs` directory, that is if you have not yet peeked and found it to be empty. This is where we generate library projects which contain most of our code. Unlike libraries generated by the Angular CLI, these libraries are configured by default to be directly referenced from application projects through TypeScript path mappings rather than having to be built and output to the `dist` directory before usage.

> **Note:** Nx library projects do not have to be packaged, published, reused, or shared. Making this assumption is a common mistake when learning Nx since the Angular CLI only has native support for generating publishable library projects to distribute in package registries such as npm.

Application projects and end-to-end-tests

When we open the `apps` directory, we get a happy throwback to early versions of Angular: End-to-end tests are back! By default, Nx generates a Cypress end-to-end testing project next to every Angular application project.

> **Note:** Even Protractor is still supported despite it being unmaintained and removed from official Angular CLI generators and executors. Enjoy it while it lasts! Or not.

In our case, the Cypress project is generated in the `gitropolis-app-e2e` directory next to the `gitropolis-app` application project directory. This is different from when previous versions of the Angular CLI generated end-to-end tests inside the application project directory that they covered.

Introducing solution-style TypeScript configurations

Remember that we mentioned solution-style TypeScript configurations earlier in this chapter? When we open up our first Angular application project, we see the following TypeScript configuration files:

- `tsconfig.app.json`: Configures compilation of application source files.
- `tsconfig.editor.json`: Configures our editor such as VS Code to have dependency type and auxiliary global type support in TypeScript source files that are not referenced by other source files. This is especially important when creating files, otherwise, auto-import extensions do not work.
- `tsconfig.json`: The base TypeScript configuration of the application project including Angular compilation settings. Extends `tsconfig.base.json` from the workspace root directory and references the other TypeScript configuration files in this project.
- `tsconfig.spec.json`: Configures compilation of test suites running Jest in Node.js.

One of the benefits of this setup is that our editor will understand that Node.js and testing APIs are unavailable in application source files. Notice, however, that in the end-to-end testing project, only a single TypeScript configuration file is generated, namely `tsconfig.json`. All files in end-to-end testing projects share the same compilation settings since only test suites and their supporting files are included, with no application source code.

Because Nx supports a range of technologies other than Angular, Angular compilation settings are added to each Angular-specific project rather than to `tsconfig.base.json`.

Conclusion

In a nutshell, when comparing a fresh workspace generated by the Angular CLI, the following list describes a fresh workspace generated by Nx's Angular workspace preset:

- The root workspace directory contains shared or base configurations for EditorConfig, ESLint, Git, Prettier, Jest, the Nx CLI, and the TypeScript compiler
- Instead of a single `projects` directory, our projects are split into the `apps` and `libs` directories
- End-to-end testing projects are generated next to the applications they cover, in the `apps` directory
- Every project has a standalone `project.json` configuration instead of it being part of `angular.json`
- The `tools` directory contains local executors and generators but is also a good location for scripts used to support our workspace
- Solution-style TypeScript configurations are used in application projects

To keep a consistent style of creating single-file standalone Angular components, directives, and pipes, we configured code generation defaults by adding settings to the `nx.json` file. Additionally, we configured VS Code to prefer the TypeScript version installed in the workspace and to organize imports on file save.

Do you think Nx uses solution-style TypeScript configurations for library projects? The answer is revealed later in this book.

Gear up for the next chapter where we learn Nx commands that are familiar, coming from the Angular CLI.

Feeling right at home with familiar Nx CLI commands

We all know and love the Angular CLI for all of its useful commands. With a few exceptions and differences, the Nx CLI supports the same commands and then some. In this chapter, we compare Angular CLI commands to Nx CLI commands.

We get familiar with the following Nx CLI commands, most of which sound familiar to Angular developers like us:

- `create-nx-workspace`
- `nx build`
- `nx serve`
- `nx generate`
- `nx lint`
- `nx test`
- `nx e2e`
- `nx run`
- `nx deploy`
- `nx extract-i18n`
- `nx migrate`
- `nx report`
- `nx analytics`
- `nx doc`
- `nx --help`

The Nx CLI comes with a few more important commands which are covered in later chapters.

create-nx-workspace is like ng new

In previous chapters, we used the standalone `create-nx-workspace` package and executable. This is similar to the `ng new` command that we as Angular developers are familiar with for generating a new Angular workspace. The `create-nx-workspace` executable is best used with the Angular preset for Nx Angular workspaces, for example:

```
# Creat an Nx Angular workspace using Yarn
yarn create nx-workspace nx-angular-gitropolis --package-manager=yarn \
  --preset=angular
```

The `create-nx-workspace` executable has other built-in presets and even supports third-party and custom presets, none of which are covered in this book though.

The `--npm-scope` parameter indicates the import alias used to reference projects in the same workspace or for publishable libraries, that is packages distributed through package registries such as npm. An example is `--npm-scope=gitropolis` in which case we could generate and reference a library such as `@gitropolis/repositories/data-access-github`.

The `--app-name` parameter specifies the project name of the generated application, for example `--app-name=gitropolis-app` (the `app` suffix is optional). An end-to-end testing project is generated using the same name concatenated with an `-e2e` suffix, for example `gitropolis-app-e2e`. Both of these projects are generated in the `apps` directory.

The `--package-manager` parameter supports the values `npm`, `yarn`, and `pnpm`. I recommend against using pnpm as package manager as long as your workspace or its dependencies rely on the Angular Compatibility Compiler (`ngcc`) as this compiler does not support modern module layouts such as those used by npm (7+) Arborist, Yarn (2+) Plug'n'Play, or the pnpm virtual store.

If you use a legacy default branch name, that is a branch name other than `main`, make sure to pass it to the `--default-base` parameter, for example `--default-base=master`.

nx build to build projects

It should come as no surprise to you that the `nx build` command supports all the parameters that `ng build` does, for example:

```
nx build gitropolis-app --configuration=production
# is the same as
nx run gitropolis-app:build:production
```

As seen in the previous example, `nx build` is a shorthand to run the `build` target of the specified project, same as `ng build`.

nx serve to start a development server

The `nx serve` command does exactly what you expect. It starts a development server hosting with live reload that hosts your application for development purposes. Similar to Angular CLI's `ng serve` command, `nx serve` is a shorthand to run the `serve` target of the specified project, for example:

```
nx serve gitropolis-app
# is the same as
nx run gitropolis-app:serve
```

Add the `--hmr` parameter to enable Webpack Hot Module Replacement (HMR), for example:

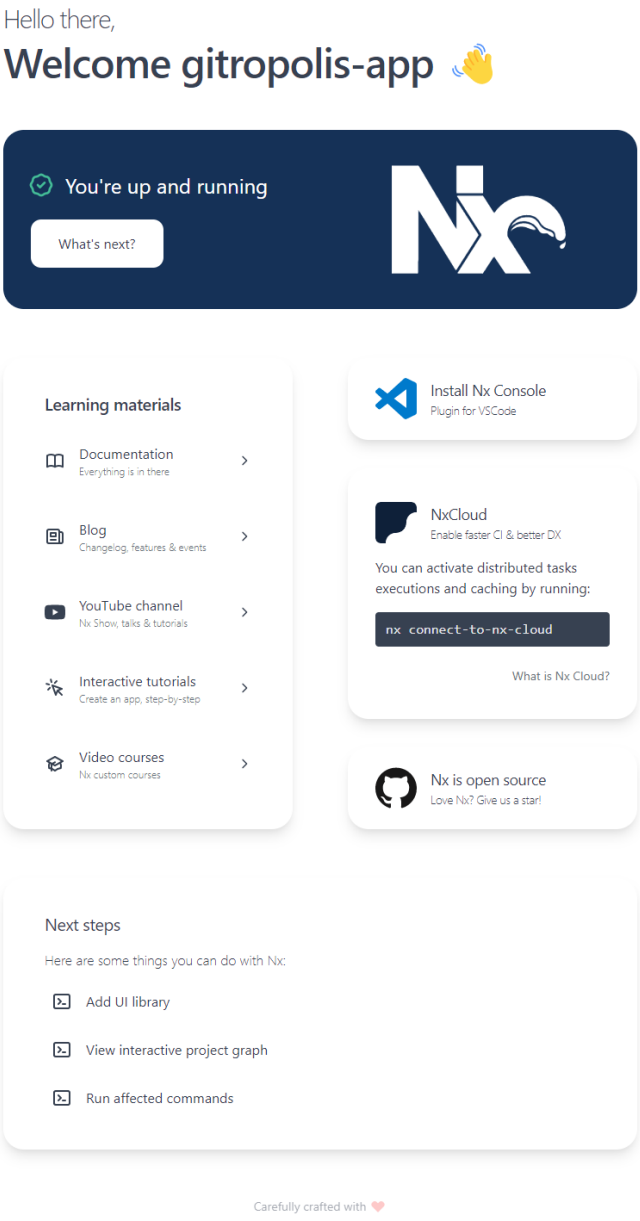
```
nx serve gitropolis-app --hmr
```

> **NOTE:** When using Hot Module Replacement, dependencies provided in environment injectors should have an `OnDestroy` hook.

Try out the `nx serve` command with the `--open` parameter to see the Gitropolis application running:

```
nx serve gitropolis-app --serve
```

This opens your browser with the address `https://localhost:4200` and you should see the Nx welcome component:



The Nx welcome component

nx generate to generate code

The `nx generate` command is exactly what we as Angular developers expect. It is used to generate code but in addition to invoking Angular CLI *schematics*, it can invoke Nx *generators* which are a framework-agnostic alternative that serve the same purpose.

The `nx generate` command accepts the name and optionally the namespace of a generator or schematic followed by generator-specific parameters and their values.

As an example, the following commands generate an Angular application project named `gitropolis-app`:

```
nx generate @nrwl/angular:application gitropolis-app --prefix=gitropolis
# is the same as
nx g app gitropolis-app -p=gitropolis
```

Because the value of the `cli.defaultCollection` setting in `nx.json` is `@nrwl/angular`, the generators that come with Nx' Angular preset are the default rather than those available in the `@schematics/angular` package.

The following common Nx Angular generators are available from the `@nrwl/angular` package:

- `application`
- `library`
- `component`
- `scam`
- `scam-directive`
- `scam-pipe`
- `move`

The `move` schematic is used to rename an Angular application or library project and/or to move it to a different directory.

The `@nrwl/workspace` package has the commonly used `remove` Nx generator which deletes a project and all references to it in shared workspace configuration files.

More generators are available, use the `nx list` command to list them all:

```
# Display information about generators and executors included in
# the Nx Angular package
nx list @nrwl/angular
```

> **Note:** The interactive *Nx Console* extension for VS Code is the best tool for exploring the generators and targets that are available in your Nx Angular workspace.

Nx' `application` and `library` generators for Angular projects have many options that are not included with the Angular CLI's `application` and `library` generators. We explore these in later chapters of this book.

nx generate instead of ng add

Nx does not have a feature that fully matches that of the `ng add` command. For example, the equivalent of running `ng add @angular/material` are the following commands when using Yarn as the package manager:

```
# Install the Angular Material and Angular CDK packages
yarn add @angular/material @angular/cdk
# Run Angular Material's `ng add` generator
nx generate @angular/material:ng-add
```

You might not know this but Angular CLI looks for an `ng-add` schematic in a package when the `ng add` command is invoked. Unfortunately, we have to do this manually when using Nx. Additionally, the default name of an equivalent Nx generator is `init`, not `ng-add`. For example, to add support for Storybook to your Nx Angular workspace, use the following commands:

```
# Add the Storybook capability to your Nx Angular workspace
yarn add @nrwl/storybook
# Initialize the Storybook capability in your Nx Angular workspace
nx generate @nrwl/storybook:init
```

If we struggle to remember these initialization instructions, we can use the `nx list <plugin-package-name>` command, see examples in the chapter *Increasing productivity with the Nx CLI's unique commands.*

nx lint to lint source files

This one is straightforward. The `nx lint` command replaces `ng lint` which means that it is a shorthand for running the `lint` target of the specified project, for example:

```
nx lint gitropolis-app
# is the same as
nx run gitropolis-app:lint
```

An Nx Angular workspace is generated with ESLint and Angular ESLint configured with Nrwl's recommended lint presets and Nx-specific lint rules.

Add the `--fix` parameter to run lint fixers, for example:

```
nx lint gitropolis-app --fix
```

nx test to run unit tests

The `nx test` command is a shorthand for running the `test` target of the specified project same as `ng test` in an Angular CLI workspace, for example:

```
nx test gitropolis-app
# is the same as
nx run gitropolis-app:test
```

An Nx workspace is generated with Jest configured for unit testing applications and library projects. While not the default option, Karma also has first-class support.

Adding the `--watch` parameter, starts Jest or Karma in file watch mode rather than exiting after running all of a project's test suites, for example:

```
nx test gitropolis-app --watch
```

If this is your first time using Jest, press the `A` key to watch all test suites in the project or explore the other options listed in the interactive prompt. Alternatively, add the `--watch-all` parameter when using Jest, for example:

```
nx test gitropolis-app --watch-all
```

When using Jest, adding the `--code-coverage` parameter, outputs an HTML test coverage report to a sub-directory of the `coverage` directory in the workspace root, for example run the following command:

```
nx test gitropolis-app --code-coverage
```

The previous command outputs a test coverage report to the following directory:

```
coverage/apps/gitropolis-app
```

Open `index.html` to start browsing the interactive test coverage report.

Alternatively, try the `text` and/or `text-summary` coverage reporters which generate test coverage reports to the standard output:

```
nx test gitropolis-app --code-coverage --coverage-reporters=text,text-summary
```

Other `--coverage-reporters` values include `html` (the default), `json`, `lcov`, and `cobertura`.

Similarly, third-party test execution reporters can be specified using the `--reporters` parameter, for example:

```
yarn add --dev jest-junit
nx test gitropolis-app --reporters=jest-junit
```

nx e2e to run end-to-end tests

The `nx e2e` command is a shorthand for running the `e2e` target of the specified project same as `ng e2e` in an Angular CLI workspace, for example:

```
nx e2e gitropolis-app-e2e
# is the same as
nx run gitropolis-app-e2e:e2e
```

An Nx workspace is generated with Cypress configured for end-to-end testing applications and library projects. While not the default option, Protractor also has first-class support.

Adding the `--watch` parameter, starts Cypress or Protractor in file watch mode rather than exiting after running all of an application's test suites, for example:

```
nx e2e gitropolis-app-e2e --watch
```

If this is your first time using Cypress, file watch mode will open the Cypress application. Click the filename of the test suite you want to run or click the label saying *Run <x> integration specs* to the right of the collapsible section saying *INTEGRATION TESTS*. The tests will be rerun when you change and save a test suite or cause the application to be rebuilt.

Running the `e2e` target of a Cypress project runs all of its test suites once in the Cypress application, then closes the Cypress application and reports the result in the console output. For CI environments such as hosted GitHub Actions runners, set the `headed` parameter to `false`, for example:

```
nx e2e gitropolis-app-e2e --headed=false
```

This will run all tests once in a headless browser without opening the Cypress application.

Test execution reporters for the Mocha testing framework can be specified using the `--reporter` parameter, for example:

```
nx e2e gitropolis-app-e2e --reporter=junit
```

This command generates a `test-results.xml` test report in the end-to-end testing project directory.

Mocha has many built-in test reporters and Cypress includes the `teamcity` and `junit` Mocha reporters, all of which are available out-of-the-box when using Cypress.

nx run to run any target in a project

The `nx run` command is similar to Angular CLI's `ng run` command. It accepts the names of a project, a target, and optionally a configuration, separated by colons, for example:

```
nx run gitropolis-app:build:production
```

Alternatively, the name of a configuration can be passed to the `--configuration` parameter, for example:

```
nx run gitropolis-app:build --configuration=production
```

If no configuration argument is specified, the default configuration will be used if one is defined in the project configuration.

nx deploy to deploy a project

While the Nx CLI recognizes common targets including `build`, `e2e`, `lint`, `serve`, and `test`, the Angular CLI recognizes other common targets, namely `deploy` and `extract-i18n`. While we can run these targets by executing the `nx run` command, the Nx CLI falls back to the Angular CLI for certain command shorthands when we use the Angular workspace preset, for example:

```
# Deploy `gitropolis-app` if it has a `deploy` target
nx deploy gitropolis-app
# is the same as
nx run gitropolis-app:deploy
```

> **Note:** A `deploy target` is not automatically generated but if we run the `nx deploy` command, the Angular CLI prompts us to install one of several common deployment packages for Angular projects.

nx extract-i18n to extract internationalization texts

As mentioned in the section *nx deploy to deploy a project*, the Nx CLI falls back to the Angular CLI for certain command shorthands, for example:

```
# Extract i18n messages from the `gitropolis-app` application
nx extract-i18n gitropolis-app
# is the same as
nx run gitropolis-app:extract-i18n
```

If we are using Angular's built-in internationalization APIs, this is an important command. If not, feel free to delete the `extract-i18n` target in an Angular application's `project.json` file.

nx migrate instead of ng update

Many Angular developers are familiar with the `ng update` command which finds a compatible version of the package with the specified name, installs it using the used package manager CLI and runs Angular-specific migrations if they are available, for example `ng update @angular/material`.

> **Note:** It is possible to migrate using the Angular CLI by setting the environment variable `FORCE_NG_UPDATE` to `true`, then running the `nx update` command, for example:
>
> ```
> nx update @angular/material
> ```

The `nx migrate` is also used for updating dependencies and running migrations but is used in a slightly different way. If no package is specified, the `nx migrate` command assumes that we mean the `@nrwl/workspace` package, for example:

```
nx migrate latest
# is the same as
nx migrate @nrwl/workspace@latest
```

This updates all packages that are managed by Nx, for example:

- `@nrwl` packages
- `@angular` packages
- ESLint, Angular ESLint, TypeScript ESLint, and other ESLint plugins managed by Nx
- Jest
- Cypress
- TypeScript
- Prettier

After running the `nx migrate` command, a `migrations.json` file is generated which contains a list of available migrations. The next step is to update the affected packages to compatible versions, for example:

```
yarn install
```

After installing the packages, we review the `migrations.json` file and remove any migrations that we don't want to run before passing the `--run-migrations` parameter:

```
nx migrate --run-migrations
```

If we want to run migrations in multiple batches, we can split them into separate files, run them one at a time, delete the migration file, then committing the changes, for example:

```
nx migrate --run-migrations=migrations-nx.json
rm migrations-nx.json
git add .
git commit --message "chore: migrate Nx packages"
nx migrate --run-migrations=migrations-angular.json
git add .
git commit --message "chore: migrate Angular packages"
```

The `nx migrate` command supports the `--from` parameter to set a range of migrations to run for specific packages, for example:

```
nx migrate @nrwl/workspace@14.x --from="@nrwl/workspace@13.0.0,@nrwl/angular@13.0.0"
```

to migrate to the latest Nx 14 version while running all Nx workspace and Nx Angular migrations since version 13.0.0, ignoring the locally installed versions of the specified packages.

Similarly, the `--to` parameter is supported, for example:

```
nx migrate @nrwl/workspace@14.x --to=@nrwl/angular@14.0.0
```

to migrate to the latest Nx packages while staying on version 14.0.0 of the `@nrwl/angular` package, preventing migrations for versions newer than that from running.

nx report instead of ng version

Similar to the `ng version` command, the `nx report` command lists essential information about an Nx environment, for example:

```
nx report

> [NX] Report complete - copy this into the issue template
  Node : 16.15.1
  OS   : win32 x64
  yarn : 1.22.19

  nx : 14.2.4
  @nrwl/angular : 14.2.4
  @nrwl/cypress : 14.2.4
  @nrwl/detox : Not Found
  @nrwl/devkit : 14.2.4
  @nrwl/eslint-plugin-nx : 14.2.4
  @nrwl/express : Not Found
  @nrwl/jest : 14.2.4
  @nrwl/js : Not Found
  @nrwl/linter : 14.2.4
  @nrwl/nest : Not Found
  @nrwl/next : Not Found
  @nrwl/node : Not Found
  @nrwl/nx-cloud : Not Found
  @nrwl/nx-plugin : Not Found
  @nrwl/react : Not Found
  @nrwl/react-native : Not Found
  @nrwl/schematics : Not Found
  @nrwl/storybook : 14.2.4
  @nrwl/web : Not Found
  @nrwl/workspace : 14.2.4
  typescript : 4.7.3
  ---------------------------------------
  Community plugins:
        nx-stylelint: 13.4.0
```

When submitting a bug report to the `nrwl/nx` GitHub repository, make sure to generate and include an Nx report such as the sample above. When asking for support in GitHub Discussions for the `nrwl/nx` GitHub repository or in the `@nrwl/community` Slack workspace, also generate and include an Nx report. This makes it easier for Nrwlians and Nx community members to help you out.

> **Note:** Unlike `ng version`, the `nx report` command also lists third-party packages that are registered as Nx community plugins.

Another use case for the `nx report` command is to verify our assumptions about a local or remote environment, for example a colleague's local development environment or a GitHub-hosted runner

for our GitHub workflows.

nx analytics to configure usage metrics

The `nx analytics` command falls back to the Angular CLI to configure whether the Angular CLI collects usage metrics.

By default, Angular CLI usage metrics are disabled. To opt-in to send usage metrics to Google, use the `enable` or `on` setting:

```
# Opt-in to send Angular CLI usage metrics to Google
nx analytics enable
# is the same as
nx analytics on
```

To stop sending Angular CLI usage metrics to Google, use the `disable` or `off` setting:

```
# Do not send Angular CLI usage metrics to Google
nx analytics disable
# is the same as
nx analytics off
```

To read our current settings for Angular CLI usage metrics, we pass the `info` value:

```
# Display settings for sending Angular CLI usage metrics to Google
nx analytics info
```

For an interactive prompt to configure settings for Angular CLI usage metrics, pass the `prompt` value:

```
# Open interactive prompt to configure settings for sending Angular CLI metrics to G\
oogle
nx analytics prompt
```

nx doc to search the Angular documentation

The Nx CLI falls back to the Angular CLI when running the `nx doc` command. For example, to search for *testing* in the Angular documentation, we use the following command:

```
# Search the Angular documentation for "testing"
nx doc testing
```

nx - -help to list parameters and examples for commands

The `nx --help <command>` command works similar to the `ng --help <command>` command, for example:

```
nx run --help
```

Adding the `--help` parameter to an Nx command outputs supported parameters and options for the specified Nx command.

Conclusion

In this chapter we learned about the Nx CLI commands that correspond to Angular CLI commands, namely these familiar commands:

- `create-nx-workspace`: Create an Nx workspace, similar to `ng new`
- `nx build`: Build a project, same as `ng build`
- `nx serve`: Start a development server, same as `ng serve`
- `nx generate`: Generate code, same as `ng generate`
- `nx lint`: Lint source files, same as `ng lint`
- `nx test`: Run unit tests, same as `ng test`
- `nx e2e`: Run end-to-end tests, same as `ng e2e`
- `nx run`: Run any target in a project, same as `ng run`
- `nx deploy`: Deploy a project if a deployment executor is configured, falls back to `ng deploy`
- `nx extract-i18n`: Extract internationalization messages from an Angular application, falls back to `ng extract-i18n`
- `nx report`: Display versions that affect your Nx environment, similar to `ng version`
- `nx analytics`: Configure Angular CLI usage metrics, falls back to `ng analytics`
- `nx --help`: Display information about the specified Nx command, same as `ng --help`

Instead of the `ng deploy` and `ng extract-i18n` commands, we use the `nx run` command to execute these targets.

We also learned about Nx alternatives to certain Angular CLI commands:

- `nx generate <package-name>:init`: Initialize a first-party or third-party package, similar to `ng add`

  > **Note**: Angular libraries might have an `ng-add` generator to initialize it instead of an `init` generator.

- `ng migrate`: Migrate to a newer version of a first-party or third-party package, similar to `ng update`

The following Angular CLI commands are incompatible with the Nx CLI as of Nx version 14.2:

- `ng cache`: Unfortunately, there is no way access the Angular CLI's `cache` command when using the Nx CLI. However, we can configure the Angular CLI's cache settings through the `cli.cache` property of `nx.json`, for example:

```
{
  "cli": {
    "cache": {
      "environment": "local",
      "path": ".angular/cache"
    }
  }
}
```

  The Angular CLI's cache is output to the `.angular/cache` directory by default. Since we cannot clean it through the Angular CLI's `cache` command, we delete it manually instead.
- `ng completion`: There are no terminal completions for Nx as of Nx version 14.2. Instead, it is recommended to use the interactive Nx Console extension for VS Code.
- `ng config`: To configure Angular CLI-specific settings, add them to the `cli` property in the `nx.json` file as described for the Angular cache settings above.

In this chapter, we have learned that–with a few exceptions–the Nx CLI is familiar to Angular developers. When we compare Nx CLI commands to their Angular CLI equivalents, we know about these improvements:

- More built-in generators, for example the `move` and `remove` generators
- `nx lint` comes with preconfigured linters and lint rules
- `nx e2e` comes with Cypress end-to-end tests
- A more customizable migration experience with the `nx migrate` command when compared to `ng update`

Generators have more options than their Angular CLI equivalents, especially the `library` generator which we will learn about in upcoming chapters.

Some of the primary benefits of using Nx are the Nx CLI commands that have no equivalents in the Angular CLI. We learn about the `format`, `list`, `run-many`, `affected`, and `graph` commands in later chapters.

Cleaning up the application project

In this chapter, we clean up the Gitropolis application project to prepare our workspace for implementing our first application features.

We learn how to:

- Convert the root component to a standalone component with a router outlet
- Convert the application to a standalone Angular application
- Add a top-level routes file
- Set up GitHub's Primer design system

Deleting the Nx welcome component

Now that we start implementing our application, we don't need the Nx welcome component, so we delete the `nx-welcome.component.ts` file from the `apps/gitropolis-app/src/app` directory.

This causes the application to break but is fixed in the next section.

Converting the root component to a standalone component

We want our root component, the `AppComponent`, to be a single-file standalone Angular component same as the rest of our application. To do this, we use these instructions in the `apps/gitropolis-app/src/app` directory:

1. Delete the `app.component.html` file.
2. Delete the `app.component.scss` file.
3. Delete the `app.component.spec.ts` file.
4. Change the contents of the `app.component.ts` file to match the following:

```
import { ChangeDetectionStrategy, Component } from "@angular/core";
import { RouterModule } from "@angular/router";

@Component({
  selector: "gitropolis-root",
  standalone: true,
  imports: [RouterModule],
  template: `
    <div class="container-lg p-responsive">
      <h1>Gitropolis</h1>
    </div>
    <router-outlet></router-outlet>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class AppComponent {}
```

The classes on the `<div>` element are part of Primer which we will install in *Steps 8-9*.

5. Delete the `app.module.ts` file.
6. Change the contents of the `main.ts` file in the `apps/gitropolis-app/src` directory to match the following:

```
import { enableProdMode, importProvidersFrom } from "@angular/core";
import { bootstrapApplication } from "@angular/platform-browser";
import { RouterModule } from "@angular/router";

import { AppComponent } from "./app/app.component";
import { appRoutes } from "./app/app.routes";
import { environment } from "./environments/environment";

if (environment.production) {
  enableProdMode();
}

bootstrapApplication(AppComponent, {
  providers: [importProvidersFrom(RouterModule.forRoot(appRoutes))],
}).catch((err) => console.error(err));
```

7. Create an `app.routes.ts` file with the following contents, adding no top-level routes for now:

```
import { Routes } from "@angular/router";

export const appRoutes: Routes = [];
```

8. Run `yarn add @primer/css` to install Primer[1] CSS.
9. Add Primer to the global styles of Gitropolis by adding the following to `apps/gitropolis-app/src/styles.scss`:

```
@use '@primer/css/index';
```

Make sure that the application works by running the `nx serve --open` command. You should see the text *Gitropolis* and there should be no errors in your browser console.

Conclusion

In this short chapter, we cleaned up our application by deleting the Nx welcome component, converting the Gitropolis application to a standalone Angular application currently consisting of a standalone root component with the text *Gitropolis* and a router outlet. We modified the `main.ts` file to bootstrap our standalone application and created the `app.routes.ts` file where we place top-level routes in later chapters.

Finally, we set up GitHub's Primer design system so that Gitropolis has a style similar to GitHub.

Before we start implementing the first Gitropolis application feature, we are going to learn important Nx library concepts. The next chapter covers Nx library traits and their characteristics.

[1] https://primer.style

Choosing library traits

Before we create our first Nx Angular library, we should know about *library traits* and *library types*. In this chapter covering library traits, we:

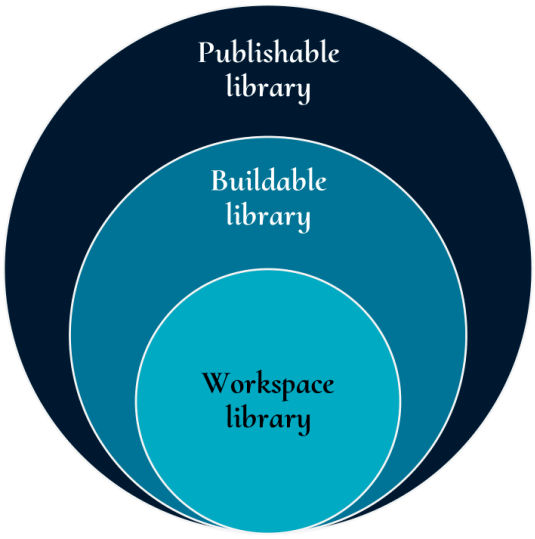
- Discuss the 3 Nx library traits and their characteristics
- Discover constraints imposed by library traits
- Learn about library management features handled by Nx
- Explore the contents of freshly generated library projects

An Nx library can have one or more of 3 different library traits:

- Workspace library
- Buildable library
- Publishable library

> **Note:** Nx has generators for non-Angular-specific libraries whose traits share most of the characteristics explained in this chapter, for example the `@nrwl/workspace:library` generator.

As illustrated in the following diagram, a publishable library *is* a buildable library and a buildable library *is* a workspace library:



Nx library traits

Workspace libraries

A *workspace* library:

- Is declared as an Nx library project by having a project configuration and an entry in the workspace configuration
- Has a primary entry point exporting its publicly available classes and other software artifacts
- Has project-specific TypeScript configuration files
- Has a path mapping entry in the workspace's base TypeScript configuration
- Has a `lint` target with a project-specific ESLint configuration
- Has a `test` target with a project-specific test runner configuration and test setup file
- **Does not** have a `build` target
- **Can** depend on workspace libraries whether they are buildable or not
- **Does not** support incremental builds

This supports running lint checks and unit tests for a single library instead of the whole workspace. It also enables us to import this library into other workspace projects to reference and use its publicly available components, services, and other software artifacts.

Workspace libraries are built as part of one or more applications.

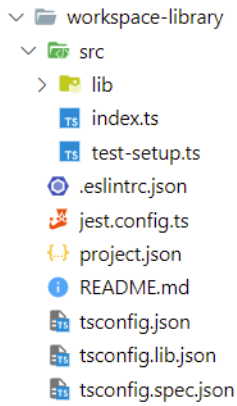
> **Note:** Nx has a lint rule which prevents other projects from making *deep imports*, that is importing software artifacts that are not exposed through a library's entry point(s).

For demonstration purposes, we generate a library called `workspace-library` using the following command:

```
1 nx generate library workspace-library
```

The name does not have to be `workspace-library`. The name of a library should not include its library trait(s).

The following file tree shows the generated files for this Nx Angular workspace library:



A freshly generated Nx Angular workspace library

The following files are generated in the root directory of a workspace library:

- `.eslintrc.json`: A project-specific ESLint configuration extending the base ESLint configuration in the workspace root directory. It adds Angular-specific lint rules.
- `jest.config.ts`: A project-specific Jest configuration extending the Jest preset in the workspace root directory. It configures `ts-jest`, `jest-preset-angular` and references a project-specific test setup file.
- `project.json`: The project configuration declaring `lint` and `test`project targets, for example:

```
{
  "projectType": "library",
  "sourceRoot": "libs/workspace-library/src",
  "prefix": "org",
  "targets": {
    "test": {
      "executor": "@nrwl/jest:jest",
      "outputs": ["coverage/libs/workspace-library"],
      "options": {
        "jestConfig": "libs/workspace-library/jest.config.ts",
        "passWithNoTests": true
      }
    },
    "lint": {
      "executor": "@nrwl/linter:eslint",
      "options": {
        "lintFilePatterns": [
          "libs/workspace-library/**/*.ts",
          "libs/workspace-library/**/*.html"
        ]
      }
    }
```

```
    },
    "tags": []
}
```

- `README.md`: A project readme file listing the Nx command to run the project's unit test suites.
- `tsconfig.json`: The base TypeScript configuration of the library project including Angular compilation settings. Extends `tsconfig.base.json` from the workspace root directory and references the other TypeScript configuration files in this project.
- `tsconfig.lib.json`: Configures compilation of library source files.
- `tsconfig.spec.json`: Configures compilation of test suites running Jest in Node.js.

Next, we have the `src` (source) directory. It contains the following files:

- `index.ts`: The primary entry point. This *barrel file* is the public API of the library in which exported software artifacts are exposed to other projects in the workspace.
- `test-setup.ts`: This test setup file is executed by Jest before any test suites are loaded. It sets up `jest-preset-angular`.

Finally, the `lib` (library) directory inside the `src` directory is where we place the source code of our library. To expose software artifacts to the rest of the workspace, we re-export them in the primary entry point, that is `index.ts`.

When an Nx library is generated, a project entry is added to the `angular.json` workspace configuration file, for example:

```
{
  "projects": {
    "buildable-library": "libs/buildable-library"
  }
}
```

The key `buildable-library` is the Nx project name used for configurations and commands, for example when running the `nx build` command:

```
nx build buildable-library
```

Additionally, an import path mapping is added to the `tsconfig.base.json` base TypeScript workspace configuration file, for example:

```
{
  "compilerOptions": {
    "paths": {
      "@workspace/buildable-library": ["libs/buildable-library/src/index.ts"]
    }
  }
}
```

This enables our editor to recognize when the library is imported into other projects in the workspace. In case of buildable libraries, without having built the library which is useful for development.

Buildable libraries

A *buildable* library:

- Shares all workspace library characteristics
- **Has** a `build` target
- **Cannot** have secondary entry points
- Uses the `@nrwl/angular:ng-packagr-lite` executor with a project-specific ng-packagr configuration
- Uses the *full* Angular Ivy compilation mode **by default** but **supports** *partial* compilation mode
- Has a project-specific package description declaring peer dependencies
- Can **only** depend on buildable libraries and by extension publishable libraries
- Supports incremental builds

Buildable libraries are built independently from other projects. They are used to support Nx' incremental builds feature. The `ng-packagr-lite` executor only produces a bundle in the latest ECMAScript Module (ES Module) version, for example the `esm2020` format.

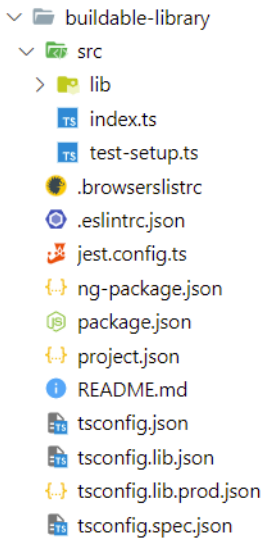
> **Note:** Applications that depend on buildable libraries should use an executor that supports incremental serve for its `serve` target, for example the `@nrwl/web:file-server` executor.

For demonstration purposes, we generate a library called `buildable-library` using the following command:

```
nx generate library buildable-library --buildable
```

The name does not have to be `buildable-library`. The name of a library should not include its library trait(s).

The following file tree shows the generated files for this buildable Nx Angular library:



A freshly generated buildable Nx Angular library

In addition to the files generated for a workspace library, the following files are generated in the root directory of a buildable library:

- `.browserslistrc`: A project-specific Browserslist configuration.
- `ng-package.json`: A ng-packagr-compatible configuration. For buildable libraries, it is used by the ng-packagr-lite executor. It references the primary entry point and configures the relative path to the output directory for the built library bundle.
- `package.json`: A package description listing the library's peer dependencies which are automatically updated in the output `package.json` file when the library is built based on the import statements in the library's source files.
- `tsconfig.lib.prod.json`: The TypeScript production build configuration extending `tsconfig.lib.json`. It disables sourcemaps for `d.ts` files.

In addition to `lint` and `test` targets, the `project.json` project configuration file declares a `build` target, for example:

```
{
  "targets": {
    "build": {
      "executor": "@nrwl/angular:ng-packagr-lite",
      "outputs": ["dist/libs/buildable-library"],
      "options": {
        "project": "libs/buildable-library/ng-package.json"
      },
      "configurations": {
        "production": {
```

```
            "tsConfig": "libs/buildable-library/tsconfig.lib.prod.json"
          },
          "development": {
            "tsConfig": "libs/buildable-library/tsconfig.lib.json"
          }
        },
        "defaultConfiguration": "production"
      }
    }
}
```

Publishable libraries

A *publishable* library:

- Shares all buildable library characteristics
- **Can** have secondary entry points
- Uses the `@nrwl/angular:package` executor with a project-specific ng-packagr configuration
- Is **only** built using the *partial* Angular Ivy compilation mode
- Is published to a private or public package registry such as npm or GitHub Packages

A publishable library is a package meant to be published on a private or public package registry. If you have ever generated a library using the Angular CLI, you created a publishable library. The `package` executor used by Nx is a modified version of Angular CLI's ng-packagr tool that produces all formats specified by the current version of the Angular Package Format (APM).

> **Note**: When we do not have a specific reason to pick other library traits, we generate workspace libraries.

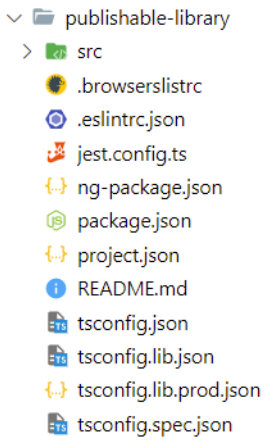
For demonstration purposes, we generate a library called `publishable-library` using the following command:

```
nx generate library publishable-library --publishable --import-path=@gitropolis/publ\
ishable-library
```

The name does not have to be `publishable-library`. The name of a library should not include its library trait(s).

> **Note**: We must manually specify the *import path* of a publishable library, for example `@gitropolis/publishable-library`.

The following file tree shows the generated files for this publishable Nx Angular library:



A freshly generated publishable Nx Angular library

The number and type of files generated in the root directory of a publishable library are the same as for a buildable library. The differences lie in 2 files:

- `project.json`: The `build` target uses a fully Angular Package Format (APM)-compatible executor, for example:

```
{
  "targets": {
    "build": {
      "executor": "@nrwl/angular:package",
      "outputs": ["dist/libs/publishable-library"],
      "options": {
        "project": "libs/publishable-library/ng-package.json"
      },
      "configurations": {
        "production": {
          "tsConfig": "libs/publishable-library/tsconfig.lib.prod.json"
        },
        "development": {
          "tsConfig": "libs/publishable-library/tsconfig.lib.json"
        }
      },
      "defaultConfiguration": "production"
    }
  }
}
```

- `tsconfig.lib.prod.json`: The *partial* Angular Ivy compilation mode is used, that is:

```
{
  "angularCompilerOptions": {
    "compilationMode": "partial"
  }
}
```

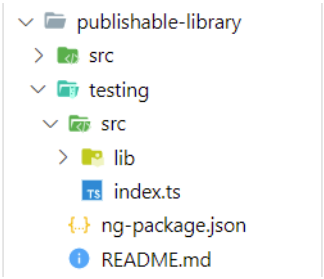
Secondary entry points

To add one or more secondary entry points, we use the `@nrwl/angular:library-secondary-entry-point` generator, for example by running the following command:

```
nx generate library-secondary-entry-point testing --library=publishable-library
```

This adds a secondary entry point to the `publishable-library` library with the sub-package name `testing`.

The following file tree shows a freshly generated secondary entry point named `testing` for a publishable Nx Angular library:



A freshly generated secondary entry point for a publishable library

A secondary entry point directory is created as a sibling directory to the primary entry point's `src` directory. In the previous screenshot, this directory is called `testing` because of the implicit `name` argument we specified when running the `nx generate` command.

In the root secondary entry point directory, 2 files are generated:

- `ng-package.json`: A ng-packagr-compatible configuration. It references the secondary entry point, for example:

```
{
  "lib": {
    "entryFile": "src/index.ts"
  }
}
```

- `README.md`: A project readme file listing the import path for the secondary entry point.

Other than these files, a `src` directory is created, containing the secondary entry point barrel file, that is `index.ts`, which exposes software artifacts to consumers of the package bundle, including other projects in the workspace.

Finally, a `lib` directory is created in which we place library source code.

> **Note**: A secondary entry point cannot have a separate package description, that is a `package.json` file. Instead, consider making its unique peer dependencies optional through configuration.

The secondary entry point belongs to the publishable library project so no additional entry is created in the workspace configuration file. However, a path mapping is added to the base TypeScript configuration, for example:

```
{
  "compilerOptions": {
    "paths": {
      "@workspace/publishable-library": [
        "libs/publishable-library/src/index.ts"
      ],
      "@workspace/publishable-library/testing": [
        "libs/publishable-library/testing/src/index.ts"
      ]
    }
  }
}
```

Conclusion

In this chapter, we learned about the characteristics of the 3 different Nx library traits, namely *workspace*, *buildable*, and *publishable* libraries. As a rule of thumb, we generate workspace libraries unless we have a reason to choose another library trait. If we want to use incremental build and incremental serve, both features offered by Nx, we generate buildable libraries. However, these build optimization features require more than just buildable libraries and are not covered by this book. To

create a package that we want to publish in a private or public package registry, we generate a publishable library.

Workspace libraries have `lint` and `test` targets while buildable libraries add a `build` target that uses the ng-packagr-lite executor which only outputs one bundle format in order to optimize build performance. Publishable libraries use a fully Angular Package Format (APM)-compatible executor for their `build` target.

These library management features are handled by Nx:

- When a library is built, peer dependencies declared in its package description are automatically updated based on import statements in its source code.
- All libraries add at least one path mapping to the `tsconfig.base.json` base TypeScript configuration file, allowing us to reference them without having built them. This is useful for development.

A few constraints are worth pointing out:

- Buildable libraries support both the *full* and *partial* Angular Ivy compilation modes while publishable libraries must be built using the *partial* Angular Ivy compilation mode.
- Publishable libraries support secondary entry points while workspace libraries and buildable libraries only support a primary entry point.

In the next chapter, we learn how to pick a conventional Nx library type.

Picking a library type

The Nx philosophy for applications is that *application projects* should contain very little source code. Almost all of our source code is contained in library projects. While you may think of *a library* as a reusable package distributed through a package registry, as we learned in the chapter *Choosing library traits*, Nx libraries and their content does not have to be reused or shared. One-off very specific libraries are perfectly valid. In fact, they are more common than shared reusable libraries.

> **Note**: If you are familiar with developing .NET systems, think of an Nx workspace as a .NET *solution*. Similarly, we can think of applications and libraries as .NET *projects*. They might be published packages but they might also be logical groupings in our system, referenced by other projects in the solution.

When creating a library intended to be used by applications, we must carefully pick a library type depending on the concerns we intend it to address. There are 4 conventional Nx library types:

- Data access libraries
- UI libraries
- Feature libraries
- Utility libraries

While an Nx library can only have one library type, we can define as many or as few library types as we want in our workspace. For example, this chapter introduces the following library types for testing purposes:

- Test utility libraries
- End-to-end test utility libraries

Additionally, the *domain library* type is introduced.

As your system grows you may find reasons to introduce more library types. Make sure their purpose is well-defined, for example in your workspace's `README.md` or `CONTRIBUTING.md` file.

> **Note**: Feel free to redefine the library types introduced in this chapter in the context of your workspace, including the conventional Nx library types.

In this chapter, we will:

- Discuss application concerns

- Define the 4 conventional library types
- Introduce 3 additional library types

In particular, we discuss which application concerns a library types addresses, which software artifact types it contains, the library types it is allowed to depend on, and naming conventions.

Application concerns

Angular applications address many concerns:

- Business logic, for example application-specific logic, domain logic, and validation rules
- Persistence, for example WebStorage, IndexedDB, WebSQL, HTTP, WebSocket, GraphQL, Firebase, and Meteor
- Messaging, for example WebRTC, WebSocket, Push API, and Server-Sent Events
- I/O, for example Web Bluetooth, WebUSB, NFC, camera, microphone, proximity sensor, and ambient light sensor
- Presentation, for example DOM manipulation, event listeners, and formatting
- User interaction, for example UI behavior and form validation
- State management, for example application state management and application-specific events

Data access libraries

A data access library deals with the following concerns:

- Persistence
- Messaging
- I/O
- State management, that is server state, persistent state, client state, transient client state, URL state, and router state

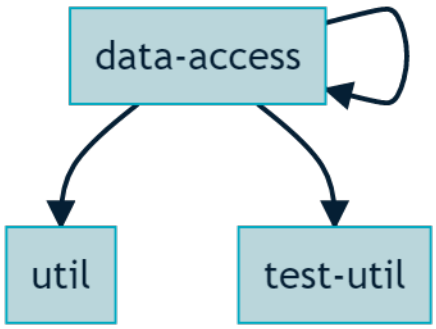
A data access library can contain:

- Data services
- HTTP interceptors
- Global or shared state management

Data access libraries can depend on the following library types:

- Data access libraries

- Utility libraries
- Test utility libraries



Allowed dependencies of data access libraries

The name of a data access library is prefixed with `data-access-`, for example `data-access-github`, or the full name can be `data-access` when the library is located in a grouping directory, for example `libs/repositories/data-access`.

UI libraries

A UI library is either a collection of reusable presentational declarables and services, or it is a collection of application-, feature- or domain-specific presentational declarables and services.

> **Note**: A publishable library is most often a UI library.

A UI library deals with the following concerns:

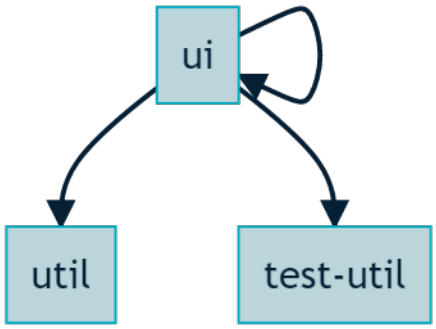
- Presentation
- User interaction
- State management, that is local UI state

A UI library can contain:

- Presentational components
- Directives
- Pipes
- Presentational services, for example presenters

UI libraries can depend on the following library types:

- UI libraries
- Utility libraries
- Test utility libraries



Allowed dependencies of UI libraries

The name of a UI library is prefixed with `ui-`, for example `ui-design-system`, or the full name can be `ui` when the library is located in a grouping directory, for example `libs/repositories/ui`.

Feature libraries

A feature library can represent a page, a screen, a use case, or any other logical grouping. It contains one or more components that may or may not be routed. A feature library orchestrates data and interactions between our application and our user. Feature libraries are used in one or more applications. The components contained by a feature library are included in an application through elements in component templates, eagerly loaded routes, lazy-loaded routes, or dynamic rendering.

We can choose to make a feature library address the same concerns as UI libraries or we can leave those concerns to UI libraries. For example, we might have multiple container components in one or more feature libraries that use a presentational component contained by a UI library.

Other than concerns shared with UI libraries, a feature library deals with the following concerns:

- Integrating presentational concerns and business logic with application state management

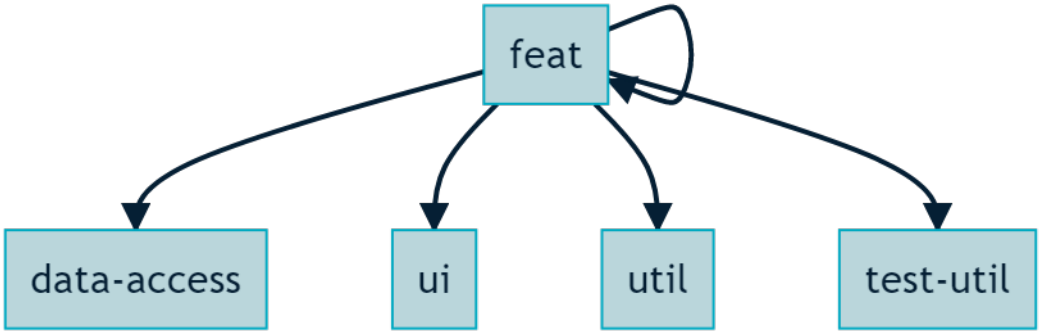
A feature library can contain:

- Smart components, for example routed components, page components, container components, and mixed components
- Routing, that is routes, route paths, route guards, route resolvers, route reuse strategies, and preloading strategies

- Local state management

Feature libraries can depend on the following library types:

- Data access libraries
- UI libraries
- Feature libraries
- Utility libraries
- Test utility libraries



Allowed dependencies of feature libraries

The name of a feature library is prefixed with `feat-`, for example `feat-repository-list`, or the full name can be `feat` when the library is located in a grouping directory, for example `libs/repositories/feat`.

Utility libraries

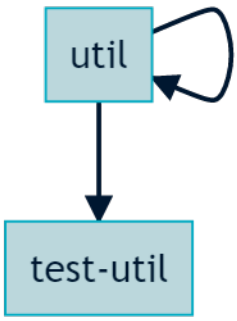
A utility library is a collection of reusable logic and values.

A utility library can contain:

- Stateless services
- Pure functions
- Constants
- Utility types

Utility libraries can depend on the following library types:

- Utility libraries
- Test utility libraries



Allowed dependencies of utility libraries

The name of a utility library is prefixed with `util-`, for example `util-date-times`, or the full name can be `util` when the library is located in a grouping directory, for example `libs/repositories/util`.

Test utility libraries

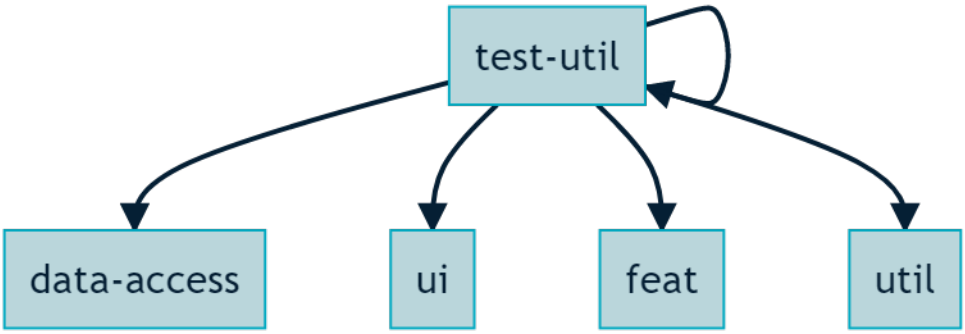
A test utility library is a collection of reusable software artifacts used for unit testing.

A test utility library can contain:

- Unit test lifecycle hooks
- Unit test configuration values
- Unit test doubles
- Unit test matchers

Test utility libraries can depend on the following library types:

- Data access libraries
- UI libraries
- Feature libraries
- Utility libraries
- Test utility libraries



Allowed dependencies of test uitlity libraries

The name of a test utility library is prefixed with `test-util-`, for example `test-util-browser`, or the full name can be `test-util` when the library is located in a grouping directory, for example `libs/repositories/test-util`.

End-to-end test utility libraries

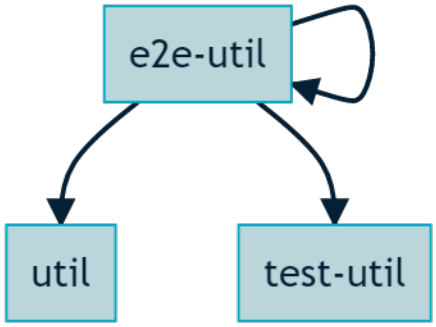
An end-to-end test utility library is a collection of reusable software artifacts used for end-to-end testing.

An end-to-end test utility library can contain:

- End-to-end test lifecycle hooks
- End-to-end test configuration values
- End-to-end test doubles
- End-to-end test queries
- End-to-end test matchers

End-to-end test utility libraries can depend on the following library types:

- Utility libraries
- Test utility libraries
- End-to-end test utility libraries



Allowed dependencies of end-to-end test utility libraries

The name of an end-to-end test utility library is prefixed with `e2e-util-`, for example `e2e-util-dom-matchers`, or the full name can be `e2e-util` when the library is located in a grouping directory, for example `libs/shared/e2e-util`.

Domain libraries

A domain library deals with the following concerns:

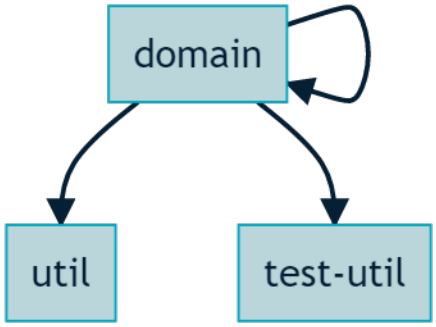
- Business logic

A domain library can contain the following software artifact types as long as they are related to a specific business domain:

- Domain types
- Domain object classes
- Domain services
- Domain constants
- Domain functions

Domain libraries can depend on the following library types:

- Utility libraries
- Test utility libraries
- Domain libraries



Allowed dependencies of domain libraries

The name of a domain library is prefixed with `domain-`, for example `domain-user`, or the full name can be `domain` when the library is located in a grouping directory, for example `libs/repositories/domain`.

Conclusion

Application projects should almost exclusively depend on feature libraries. An application must not depend directly on data access libraries, UI libraries, or domain libraries.

Feature libraries are a good starting point. We could choose to start by having all logic for an entire page or screen in a feature library but as our implementation grows, we split it into multiple feature libraries and/or start extracting concerns to more appropriate library types.

We defined these library types which we use throughout this book:

- Data access libraries
- UI libraries
- Feature libraries
- Utility libraries
- Test utility libraries
- End-to-end test utility libraries
- Domain libraries

In the next chapter, we create our first libraries to make room for our first application feature.

Creating a feature library

It's time to create our first Nx Angular library. In this chapter, we:

- Create a feature library
- Create a routed feature component
- List a single repository with static data

Creating a feature library

To generate the *repositories* feature library for the *organizations* domain, use the following generator command:

```
nx generate library feat-repositories --directory=organizations \
  --prefix=gitropolis-orgs
```

The name passed to the `nx generate library` command follows the format `<library-type>[-<library-name>]`, in this case `feat-repositories` where *feat* is shorthand for *feature library* and *repositories* is the library name.

We choose to identify the *domain* by the top-level directory name by passing `organizations` to the `--directory` parameter. The `--prefix` parameter specifies the default prefix for components and other software artifacts generated in this library, in this case the prefix is `gitropolis-orgs`.

Delete the generated `organizations-feature-repositories.module.ts` file in the library's `src/lib` directory containing an Angular module.

Creating a feature component

Generate a standalone *repositories* feature component using the following command:

```
nx generate component organizations-feat-repositories --export \
  --project=organizations-feat-repositories \
  --selector=gitropolis-orgs-repositories
```

We specify a name parameter using the format `<domain>-feat-<feature-name>`, in this case `organizations-feat-repositories`, because a feature component is special.

> A feature component is special because it will either be routed or included in the template of a component in another feature library. It is the entry point of a feature library.

The `--export` parameter flag exports the feature component from our feature library's primary entry point, the barrel file, that is `src/index.ts`. This component is now part of our feature library's public API surface, allowing other projects to import it.

We pass the Nx project name of our new library to the `--project` parameter. Notice how it follows the format `<domain>-<library-type>-<library-name>`, in this case `organizations-feat-repositories`.

Finally, we pass an element name in the format `<app-prefix>-<domain-prefix>-<feature-name>` to the `--selector` parameter, in this case `gitropolis-orgs-repositories`.

Adding the feature route

To try the repositories feature library, we add a route to its feature component. Open `apps/gitropolis-app/src/app/app.routes.ts` and add the following route to the `appRoutes` array:

```
{
  path: 'orgs/:organization/repositories',
  loadComponent: () =>
    import('@gitropolis/organizations/feat-repositories').then(
      (m) => m.OrganizationsFeatRepositoriesComponent
    ),
},
```

The route path has an `organization` route parameter. The lazy-loaded route resolves the feature component.

Notice that we refer to our feature library using the import path alias `@gitropolis/organizations/feat-repositori` which follows the format `@<workspace-npm-scope>-<domain>-<library-type>-<library-name>`.
The import path alias closely matches the library path which is `libs/organizations/feat-repositories`.

Let us try out the feature, shall we?

Run `nx serve --open` to start a development server and https://localhost:4200 is automatically opened in your default browser when the server is ready. You should see the application title *Gitropolis*.

Now, enter the following URL and press your *Enter* key:

```
http://localhost:4200/orgs/nrwl/repositories
```

The small text *organizations-feat-repositories works!* should be visible under the application title.

Congratulations, you created our first routed feature!

Listing organization repositories

We proceed by creating a repository list component in the repositories feature library. Run the following command to generate the component:

```
nx generate component repository-list --project=organizations-feat-repositories
```

The Nx CLI generates a component with the selector `gitropolis-orgs-repository-list` because of our generator defaults.

> **Note**: With the Nx Console extension, we can right-click a directory and press the `Nx generate...` menu item in the context menu to automatically specify values for the `--project` and `--path` parameters.

Add a *Repositories* heading and the repository list component to the repositories feature component's template. The contents of `organizations-feat-repositories.component.ts` should be as follows:

```
import { ChangeDetectionStrategy, Component } from '@angular/core';
import { RepositoryListComponent } from './repository-list.component';

@Component({
  selector: 'gitropolis-orgs-repositories',
  standalone: true,
  imports: [RepositoryListComponent],
  template: `
    <div class="container-lg p-responsive">
      <gitropolis-orgs-repository-list></gitropolis-orgs-repository-list>
    </div>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class OrganizationsFeatRepositoriesComponent {}
```

We should see *repository-list works!* in our Gitropolis browser tab.

As you might have noticed, we do not export the repository list component from the feature library's entry point by re-exporting it in the `index.ts` barrel file. Other projects in our Nx Angular workspace should have no interest in referring to it.

Now, we generate a *public source* component that presents an overview of a public repository owned by an organization:

```
nx generate component public-source --project=organizations-feat-repositories
```

Add the public source component to the repository list component's template and wrap it in a Primer Box Row like so:

```
import { ChangeDetectionStrategy, Component } from '@angular/core';
import { PublicSourceComponent } from './public-source.component';

@Component({
  selector: 'gitropolis-orgs-repository-list',
  standalone: true,
  imports: [PublicSourceComponent],
  template: `
    <h2>Repositories</h2>
    <div class="Box">
      <ul>
        <li class="Box-row">
          <gitropolis-orgs-public-source></gitropolis-orgs-public-source>
        </li>
      </ul>
    </div>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class RepositoryListComponent {}
```

Inside the *Box*, we should see the text *pubilc-source works!* in our browser.

Adding repository name and description

Our initial implementation of the public source component contains static data. Later, we load data dynamically from the GitHub REST API.

Change the file contents of the public source component to the following:

```
import { CommonModule } from '@angular/common';
import { ChangeDetectionStrategy, Component } from '@angular/core';

@Component({
  selector: 'gitropolis-orgs-public-source',
  standalone: true,
  imports: [CommonModule],
  template: `
    <h3>
      {{ name }}
      <span class="Label Label--secondary ml-1">Public</span>
    </h3>
    <p>{{ description }}</p>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class PublicSourceComponent {
  description = 'Smart, Fast and Extensible Build System';
  name = 'nx';
}
```

In the browser, we see the following:

Adding repository topics

To add a static list of repository topics, we first generate a *topics* component:

```
nx generate component topics --project=organizations-feat-repositories
```

Add `TopicsComponent` to the `imports` array of the public source component's template, then add the topics component to the end of its template:

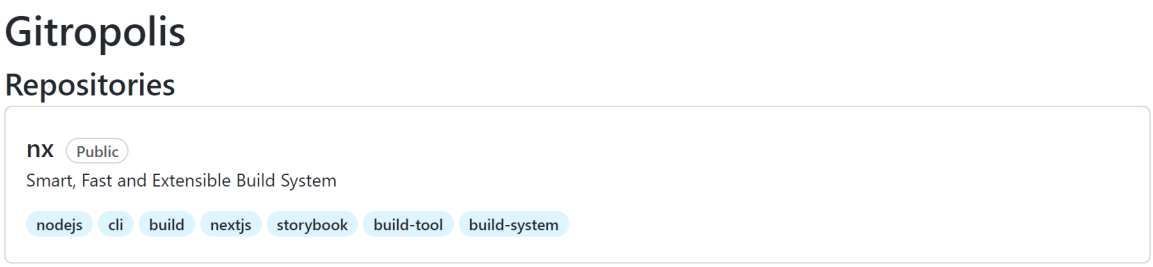
```
<gitropolis-orgs-topics></gitropolis-orgs-topics>
```

Change the contents of the `topics.component.ts` file to the following to present a static list of topics:

```
import { CommonModule } from '@angular/common';
import { ChangeDetectionStrategy, Component } from '@angular/core';

@Component({
  selector: 'gitropolis-orgs-topics',
  standalone: true,
  imports: [CommonModule],
  template: `
    <div class="d-inline-flex flex-wrap flex-items-center my-1">
      <span
        *ngFor="let topic of topics"
        class="IssueLabel color-bg-accent color-fg mr-1"
        >{{ topic }}</span
      >
    </div>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class TopicsComponent {
  topics = [
    'nodejs',
    'cli',
    'build',
    'nextjs',
    'storybook',
    'build-tool',
    'build-system',
  ];
}
```

Conclusion

In our browser, we now see the following:



The repository list with repository topics

If your browser shows something else, go through the steps in this chapter again before moving on to the next chapter.

In this chapter, we learned to create a feature library by generating the *repositories* feature library. We created and exported the *repositories* feature component and added a top-level feature route to the Gitropolis application.

As a first step, we list the `nrwl/nx` repository using static data in the *public source* component which itself uses the *topics* component. We list a repository's name, visibility (*Public*), description, and topics. The *repository list* is wrapped in a Primer *Box* and the public source component is wrapped in a *Box row*. The *Public* visibility uses a secondary Primer *Label* while the topics use *Issue labels*.

That is a good first step. In the next chapter, we create a data access library in order to load the data for a single repository dynamically.

Creating a data access library

Having prepared our *repositories* feature with static data for a single repository, we create a data access library in this chapter to support loading data dynamically.

In this chapter, we:

- Generate a data access library
- Introduce the abortable RxJS operator to make GitHub REST API requests cancelable
- Implement the repository API service wrapping the GitHub REST API
- Create the repository domain model
- Implement the repository state service which loads server state using the repository API service

Creating a data access library

To generate the data access library for the *organizations* domain, use the following generator command:

```
nx generate library data-access --directory=organizations
```

Similar to generating a feature library–or any other library type–the name passed to the `nx generate library` command follows the format `<library-type>[-<library-name>]`, in this case `data-access-api`.

Delete the generated Angular module and remove its re-export in the data access library's entry point. Temporarily create an empty entry point by replacing the file contents of `index.ts` with the following:

```
export {};
```

Adding Octokit REST

Run the following command to install the Octokit REST API client:

```
yarn add @octokit/rest
```

> **Note:** Refer to the Octokit documentation at https://octokit.github.io/rest.js/ and the GitHub REST API documentation at https://docs.github.com/en/rest for details on Octokit and the REST API it sends HTTP requests to.

We create an Octokit singleton service by providing a factory to a dependency injection token. Create an `octokit.token.ts` file with the following contents:

```
import { InjectionToken } from '@angular/core';
import { Octokit } from '@octokit/rest';

export const octokitToken = new InjectionToken<Octokit>('octokitToken', {
  providedIn: 'root',
  factory: () => new Octokit(),
});
```

Introducing the abortable operator

Before creating a data service, we need a custom *abortable* RxJS operator. The Octokit REST API client returns promises but supports passing an `AbortSignal` to abort an HTTP request. Wrapping requests in RxJS observables in this way makes GitHub API requests abortable, meaning they are canceled when no subscribers remain.

Create an `abortable.operator.ts` file with the following contents:

```
import { Observable } from 'rxjs';

export function abortable<TValue>(
  promiseFactory: (signal: AbortSignal) => Promise<TValue>
): Observable<TValue> {
  return new Observable<TValue>((subscriber) => {
    const abortController = new AbortController();

    promiseFactory(abortController.signal)
      .then((value) => {
        if (subscriber.closed) {
          return;
        }

```

```
        subscriber.next(value);
        subscriber.complete();
      })
      .catch((error: unknown) => {
        if (subscriber.closed) {
            return;
        }

        subscriber.error(error);
      });

    return () => {
      abortController.abort();
    };
  });
}
```

Do not worry too much about this implementation. We will use the operator and discuss its API shortly.

Implementing the repository API service

Now, we create a repository data service with a method for requesting a repository list with the dynamic data we need for our repositories feature. Generate a service using the following command:

```
nx generate service repository-api --project=organizations-data-access
```

Add an `byOrganization` method by replacing the file contents of `repository-api.service` with the following:

```
import { inject, Injectable } from '@angular/core';
import { abortable } from './abortable.operator';
import { octokitToken } from './octokit.token';

@Injectable({
  providedIn: 'root',
})
export class RepositoryApiService {
  #octokit = inject(octokitToken);

  byOrganization(organization: string) {
```

```
12      return abortable((signal) =>
13        this.#octokit.rest.repos
14          .listForOrg({
15            org: organization,
16            sort: 'updated',
17            request: {
18              signal,
19            },
20          })
21          .then((response) => response.data)
22      );
23    }
24  }
```

We pass `octokitToken` to the `inject` function to resolve the singleton instance of the `Octokit` client.

The `byOrganization` method passes a promise factory to our `abortable` operator to defer the creation of the promise until an observer subscribes. Our promise factory receives an `AbortSignal` which we name `signal` and forward to the `request.signal` option for the `Octokit#rest.repos.listForOrg` method.

We pass the name of an organization that we want to list repositories for. Except for sorting repositories by the date a commit was last pushed, we use default options meaning that we receive a single page of results with a maximum of 30 results. To avoid introducing additional complexity, we only list the first page rather than supporting pagination.

Creating a repository domain model

When we created the repository list feature in the chapter *Creating a feature library*, we learned which repository properties we need to display the list.

We want to load a repository list dynamically. To do that, we create a repository domain model with the properties needed.

Create a new file named `repository.ts` in the `libs/organizations/data-access/src/lib` directory and add the following contents:

```
export interface Repository {
  readonly description: string;
  readonly name: string;
  readonly organization: string;
  readonly topics: readonly string[];
}

export type Repositories = readonly Repository[];
```

Our `Repository` domain model contains the repository description, name, and topics as well as the name of the organization it belongs to. We choose to declare `readonly` access to all properties to help enforce immutability.

The `Repositories` type is a shorthand for the `ReadonlyArray<Repository>` collection type which is also expressed as `readonly Repository[]`.

Re-export the repository domain model and collection type in our data access library's entry point. Replace the file contents of `index.ts` with the following:

```
export * from './lib/repository';
```

This enables other libraries to reference our domain model.

Adding RxAngular State

We are going to use the RxAngular State package for state management. In the next section, we use this package to create a state service. Install it using the following command:

```
yarn add @rx-angular/state
```

RxAngular State is now ready to use. It does not require any global configuration.

Implementing the repository state service

Next, we create a repository state service which loads public repositories when an organization is selected. Afterwards, we will connect this service to the repository list in our repositories feature. Generate the service using the following command:

```
nx generate service repository-state --project=organizations-data-access
```

Open the generated file and start by adding the `RepositoryState` interface which describes the internal state of the service:

```
import { Injectable } from '@angular/core';
import { RxState } from '@rx-angular/state';
import { Repositories } from './repository';

interface RepositoryState {
  readonly organization: string;
  readonly repositories: Repositories;
}

@Injectable({
  providedIn: 'root',
})
export class RepositoryStateService extends RxState<RepositoryState> {}
```

The state service contains `organization` and `repositories` properties internally. Our application will pass the organization name when selected by a user.

> **Note:** We keep the `{ providedIn: 'root' }` option for the `Injectable` decorator so that repositories are cached locally in memory when we return to the repository list after having navigated to a different route. Another options is totie this service to the lifecycle of a component by adding it to a `Component.providers` metadata option, for example to the repository list component. This would clear loaded repositories when navigating away from the component's route.

Now, we *connect* a data source to the `repositories` state property which loads repositories when a new `organization` state property is emitted:

```
import { inject, Injectable } from '@angular/core';
import { RxState } from '@rx-angular/state';
import { map, switchMap } from 'rxjs';
import { Repositories, Repository } from './repository';
import { RepositoryApiService } from './repository-api.service';

interface RepositoryState {
  readonly organization: string;
  readonly repositories: Repositories;
}

@Injectable()
export class RepositoryStateService extends RxState<RepositoryState> {
  #api = inject(RepositoryApiService);

```

```
  constructor() {
    super();

    this.connect(
      'repositories',
      this.select('organization').pipe(
        switchMap((organization) =>
          this.#api.byOrganization(organization).pipe(
            map((response) =>
              response.map(
                ({ description, name, topics }): Repository => ({
                  description: description ?? '',
                  name,
                  organization,
                  topics: topics ?? [],
                })
              )
            )
          )
        )
      )
    );
  }
}
```

We use the repository API service to load the public repositories of the specified organization and map the response to our repository domain model, stored in the `repositories` property state and emitted to subscribers of that property. A `null` description is converted to an empty string while a `null` list of topics is converted to an empty array.

We use the `switchMap` operator to cancel any in-flight requests when a new organization is selected. This is possible thanks to the abortable RxJS operator we added to the repository API service earlier in the *Implementing the repository API service* section.

Now, re-export the repository state service in the entry point of our organizations data access library to make it consumable from other libraries in our Nx workspace. Remember that this is done in the library's `index.ts` barrel file as seen in the following code listing:

```
export * from './lib/repository';
export * from './lib/repository-state.service';
```

This is the public API of the organizations data access library. The repository API service is intentionally hidden from other libraries.

Conclusion

In this chapter, we created our first data access library located in the *organizations* domain.

We installed the Octokit client to use the GitHub REST API. Octokit uses promises. One of the benefits of using RxJS is that we can cancel side effects such as HTTP requests as needed. Unlike promises, RxJS observables are lazily evaluated. To enable both of these features in our data access library, we created the `abortable` RxJS operator which internally sets up an `AbortController` and exposes its `AbortSignal` to support setting up promises by wrapping them in a factory function. We generated the repository API service and added the `byOrganization` method which uses the abortable operator to pass an abort signal to the Octokit REST client.

We created a `Repository` domain model and exported it with the collection type `Repositories`, a shorthand for a read-only array of repositories.

In addition to the domain model, we re-export the repository state service which we implemented using the RxAngular State package. The service has a built-in side effect that loads public repositories using the repository API service every time an organization is selected. Consumers are able to subscribe to repositories but we are going to need to connect a data source for organizations for this to work.

In the next chapter, we select an organization based on the URL a user navigates to and we pass the dynamically loaded repository list to our *repositories* feature, replacing the static repository listing we used in the chapter *Creating a feature library*.

Loading repositories dynamically

In the chapter *Creating a feature library*, we created the *repositories* feature. In the previous chapter, we created the *organizations* data access library that exports the repository state service. With that in place, we are ready to dynamically load repositories and display them in our *repositories* feature.

In this chapter, we:

- Select an organization based on the URL a user navigates to which in turn loads the organization's repositories
- Replace the static repository listing by passing the dynamically loaded repositories to our repository list component and its child components

We change the implementations of the components in our *repositories* feature to accept repository domain models, starting from the repository list component and ending at the topics component.

Connecting the organization route parameter to the repository state

Open `libs/organizations/feat-repositories/src/lib/repository-list.component.ts` and connect the organization route parameter to the repository state by adding the following property initializers and method call:

```
import { ChangeDetectionStrategy, Component, inject } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { RepositoryStateService } from '@gitropolis/organizations-data-access';
import { filter, map, Observable } from 'rxjs';

@Component({
  // (...)
})
export class RepositoryListComponent {
  #route = inject(ActivatedRoute);
  #store = inject(RepositoryStateService);
  #organizationRouteParameter$ = this.#route.paramMap.pipe(
    map((params) => params.get('organization')),
    filter((organization) => organization !== null)
  ) as Observable<string>;
```

```
16
17    constructor() {
18      this.#store.connect('organization', this.#organizationRouteParameter$);
19    }
20  }
```

We inject the `ActivatedRoute` service to access its `ParamMap`. The route path for the *repositories* feature is `orgs/:organization/repositories`. We access the `organization` route parameter from the `ParamMap` and filter out null values to prevent them from entering our repository state.

In the repository list component's constructor, we call `RepositoryStateService#connect`, passing `'organization'` for the state property we want to target followed by the organization route parameter observable that we stored in the `#organizationRouteParameter$` property.

Passing an organization's repositories from the repository state

Our next step is to expose the `'repositories'` state property to the template of the repository list component. We do this by initializing the `repositories$` property to `this.#store.select('repositories')` which returns `Observable<Repositories>`:

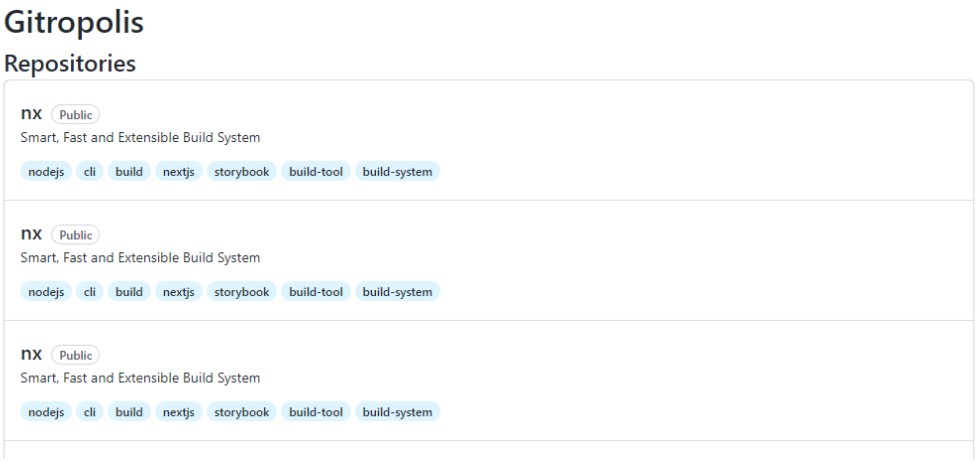
```
1  // (...)
2  export class RepositoryListComponent {
3    #store = inject(RepositoryStateService);
4    // (...)
5    repositories$ = this.#store.select('repositories');
6  }
```

With the selected organization's repositories wrapped in an observable, we use Angular's `AsyncPipe` to loop over each `Repository` domain object and create a public source component for them:

```
1  @Component({
2    selector: 'gitropolis-orgs-repository-list',
3    standalone: true,
4    imports: [CommonModule, PublicSourceComponent],
5    template: `
6      <h2>Repositories</h2>
7      <div class="Box">
8        <ul>
9          <li *ngFor="let repository of repositories$ | async" class="Box-row">
10           <gitropolis-orgs-public-source></gitropolis-orgs-public-source>
```

```
        </li>
      </ul>
    </div>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class RepositoryListComponent {
  // (...)
}
```

To use the `AsyncPipe` and the `NgForOf structural directive`, we add back the `CommonModule` to the `imports` option for the `Component` decorator.



The repository loop repeating static data.

The previous screenshot demonstrates that the repository loops repeats the static data as many times as the number of dynamically loaded repositories.

The next section passes a repository to each public source component through a yet-to-be-defined input property.

Accepting a repository domain object

Now, we open `public-source.component.ts` and add the `repository` input property which accepts a `Repository` domain object. It must also accept a null value to support passing value through the `AsyncPipe` as it initially emits a null value when subscribing to an observable:

```
import { Repository } from '@gitropolis/organizations-data-access';
// (...)
export class PublicSourceComponent {
  @Input()
  repository: Repository | null = null;
}
```

Make sure to remove the static `description` and `name` properties. The application breaks temporarily but that is okay.

To pass a repository to each public source component, change the template of the repository list component to the following:

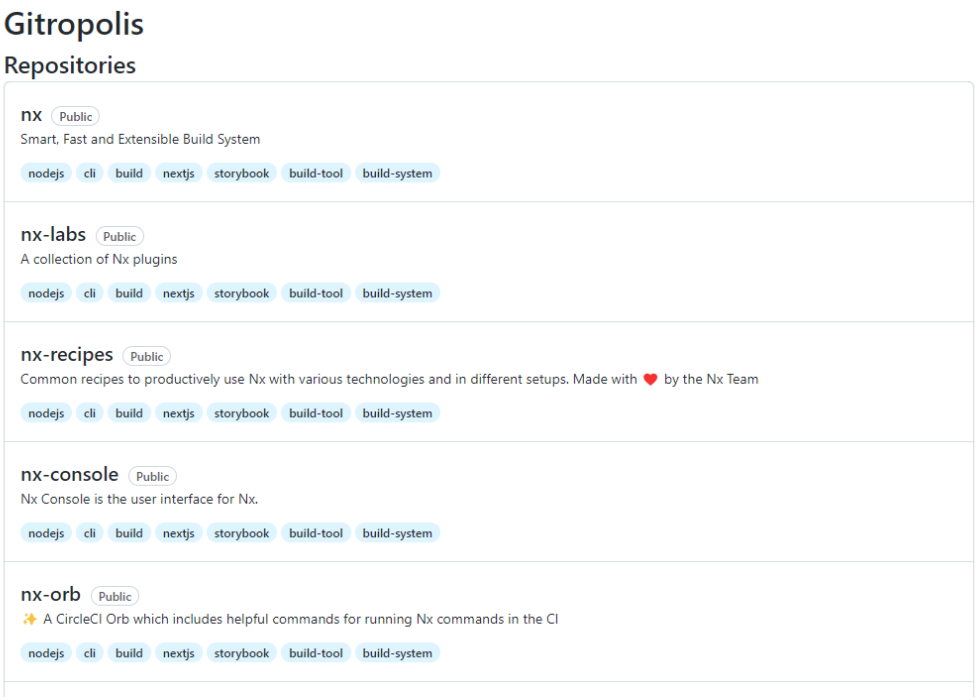
```
<!-- (...) -->
<li *ngFor="let repository of repositories$ | async" class="Box-row">
  <gitropolis-orgs-public-source
    [repository]="repository"
  ></gitropolis-orgs-public-source>
</li>
```

Next, we change the public source component so that it supports a null input value and hides the paragraph element when a repository has no description:

```
import { CommonModule } from '@angular/common';
import { ChangeDetectionStrategy, Component, Input } from '@angular/core';
import { Repository } from '@gitropolis/organizations-data-access';
import { TopicsComponent } from './topics.component';

@Component({
  selector: 'gitropolis-orgs-public-source',
  standalone: true,
  imports: [CommonModule, TopicsComponent],
  template: `
    <h3>
      {{ repository?.name }}
      <span class="Label Label--secondary ml-1">Public</span>
    </h3>
    <p *ngIf="hasDescription">{{ repository?.description }}</p>
    <gitropolis-orgs-topics></gitropolis-orgs-topics>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
```

```
export class PublicSourceComponent {
  @Input()
  repository: Repository | null = null;

  get hasDescription(): boolean {
    return (this.repository?.description ?? '') !== '';
  }
}
```

Null values are supported by using the safe navigation template operator (`?.`). The paragraph element is hidden based on the `hasDescription` property using the `NgIf` structural directive. Make sure to add back the `CommonModule` to the `imports` option of the `Component` decorator to support using `NgIf`.



Dynamically loaded repositories with static repository topics.

In the previous screenshot, we see that the list of repositories is dynamically loaded while the static repository topics are still repeated.

Our final state change to the *repositories* feature is to accept a dynamic list of repository topics in the topics component. This is implemented in the next section.

Passing a dynamic list of repository topics

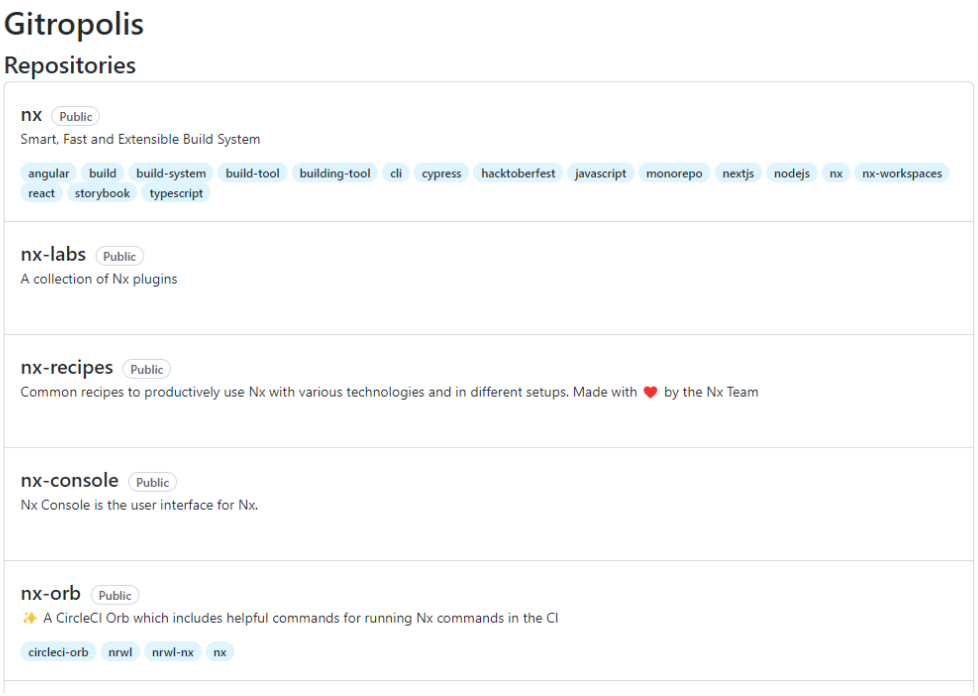
Open `topics.component.ts` and replace the static `topics` property with an input property accepting a read-only array of strings:

```
import { CommonModule } from '@angular/common';
import { ChangeDetectionStrategy, Component, Input } from '@angular/core';

@Component({
  selector: 'gitropolis-orgs-topics',
  standalone: true,
  imports: [CommonModule],
  template: `
    <div class="d-inline-flex flex-wrap flex-items-center my-1">
      <span
        *ngFor="let topic of topics"
        class="IssueLabel color-bg-accent color-fg mr-1"
        >{{ topic }}</span
      >
    </div>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class TopicsComponent {
  @Input()
  topics: readonly string[] = [];
}
```

Go back to the public source component and modify its template so that it passes repository topics to the topics component through its `topics` input property:

```
<!-- (...) -->
<gitropolis-orgs-topics
  [topics]="repository?.topics ?? []"
></gitropolis-orgs-topics>
```

The safe navigation (`?.`) and nullish coalescing (`??`) template operators are used to support a null value passed to the public source component by the repository list component which uses `AsyncPipe`. When `null` is passed to the `repository` input property, we pass an empty array to the topics component's `topics` input property.



Dynamically loaded repository topics.

The previous screenshot shows dynamically loaded repository topics. If you are used to browsing repositories on GitHub, you might be surprised to see the long list of repository topics for the `nx` repository. Additionally, a repository without topics takes up a blank line in the list. We address these issues in the next section.

Optimizing the topics list visually

First, we hide the `<div>` element wrapping the topic list when the `topics` array is empty:

```
import { CommonModule } from '@angular/common';
import { ChangeDetectionStrategy, Component, Input } from '@angular/core';

@Component({
  selector: 'gitropolis-orgs-topics',
  standalone: true,
  imports: [CommonModule],
  template: `
    <div
      *ngIf="hasTopics"
      class="d-inline-flex flex-wrap flex-items-center my-1"
    >
      <span
```

```
        *ngFor="let topic of topics"
        class="IssueLabel color-bg-accent color-fg mr-1"
        >{{ topic }}</span
      >
    </div>
  `,
  styles: [],
  changeDetection: ChangeDetectionStrategy.OnPush,
})
export class TopicsComponent {
  @Input()
  topics: readonly string[] = [];

  get hasTopics(): boolean {
    return this.topics.length > 0;
  }
}
```

This is done to prevent the topics component from creating a blank line in the repository list component.

Next, we remove all but the first 7 topics from the list to prevent visual overflow.

> **Note:** GitHub selects 7 topics based on criteria such as global topic popularity. Gitropolis keeps it simple.

To do this, we create an array comparison function. Create the file `libs/organizations/feat-repositories/src/li` and add the following file contents:

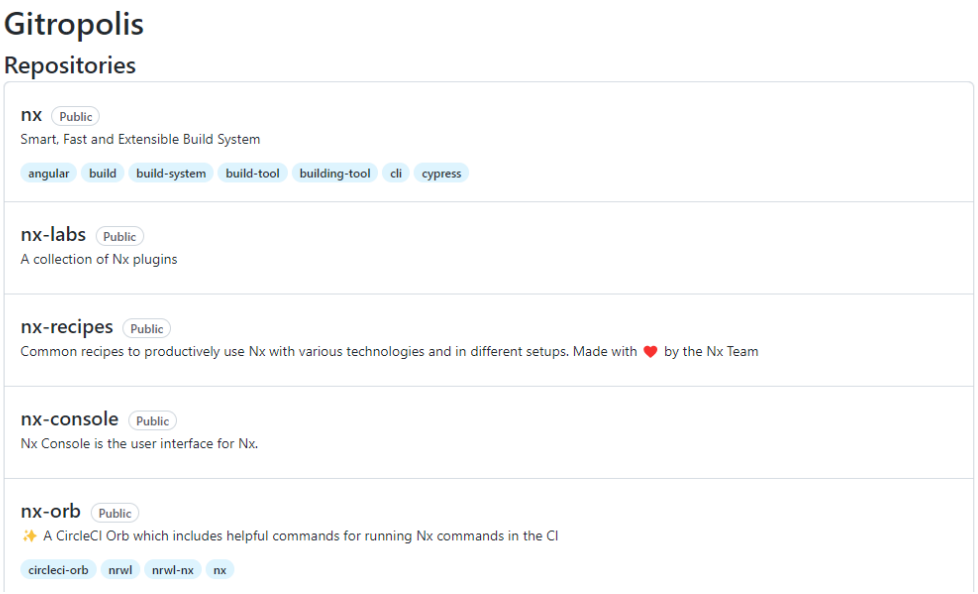
```
export const arrayEquals = <TValue>(
  xs: readonly TValue[],
  ys: readonly TValue[]
): boolean =>
  xs.length === ys.length && !xs.some((x, index) => x !== ys[index]);
```

`arrayEquals` compares the elements of two arrays *by reference* to determine whether the two arrays can be considered *equal*.

Now, we replace the `topics` input property with the following properties:

```
import { arrayEquals } from './array-equals';
// (...)
export class TopicsComponent {
  #topics: readonly string[] = [];

  @Input()
  set topics(topics: readonly string[]) {
    // We only display 7 topics to prevent an overflowing topic list
    topics = topics.slice(0, 7);

    if (arrayEquals(topics, this.#topics)) {
      return;
    }

    this.#topics = topics;
  }
  get topics(): readonly string[] {
    return this.#topics;
  }

  get hasTopics(): boolean {
    return this.topics.length > 0;
  }
}
```

Topics are now stored in the `#topics` backing field. Every time an array of strings is passed to the `topics` input property, we select the first `7` topics, compare them to the topics in the backing field and store them if they are different from the previously-stored topics.



Visually optimized repository topics.

The previous screenshot shows the repository list visually optimized concerning repository topics. At most, 7 topics are displayed. No blank line is displayed when a repository has no topics.

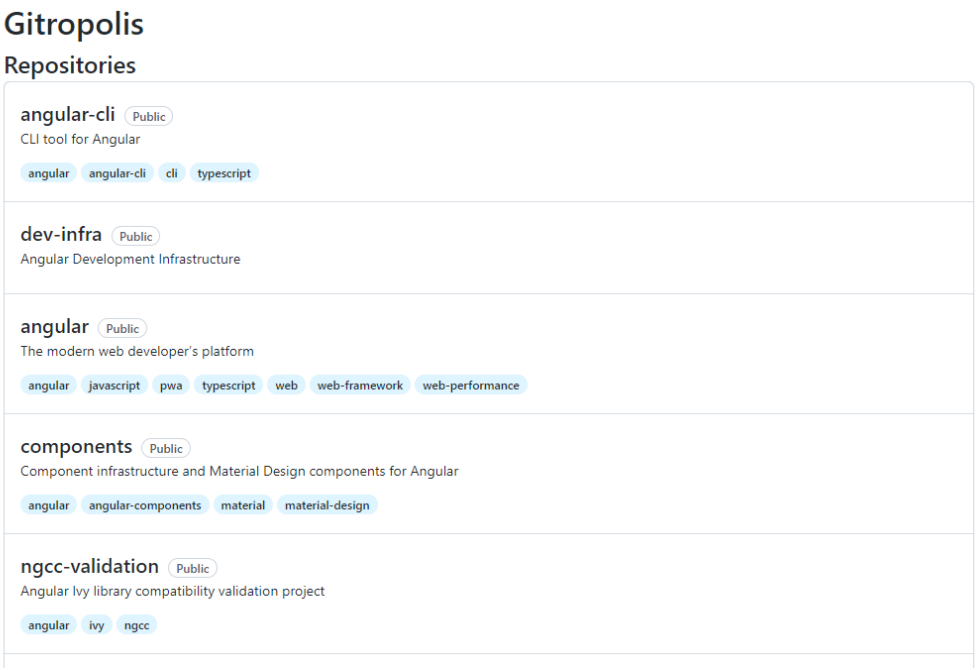
Conclusion

In this chapter, we connected the `organization` route parameter to the repository state which triggers the dynamic loading of repositories when a user navigates to the feature route.

We passed the dynamically loaded repositories to the repository list component which loops over them and forwards each repository domain object to the public source component. We made visual improvements by removing blank space for repositories without a description.

The public source component passes the repository topics to the topics component which loops over them. The topics component now has visual optimizations to prevent it from taking up a blank line for repositories without topics. Additionally, it displays a maximum of 7 topics to prevent visual overflow in the repository list.

We have been using the `nrwl` GitHub organization as an example by navigating to http://localhost:4200/orgs/nrwl/repositories but other valid GitHub organizations also work, for example `angular` at http://localhost:4200/orgs/nrwl/repositories as seen in the following screenshot:



The repository list for the `angular` GitHub organization.