Contents

1 Introduction

The current landscape of UI development is in a very interesting place, for many reasons. For starters, the mere capacity of web to do many tasks previously unavailable(Add examples here), is growing by the year. In addition, frameworks which allow for scalability, DRY development, and consistency is ever going. In your average web application, it generally includes type checking, unit testing, integration testing, as well as state management, and observables/effects. These are all things that 3 years ago were not common place in the enterprise, and it is only becoming a greater landscape as time goes on.

Angular in my opinion, having worked with other frameworks such as Elm, Vue, React, and Cycle, is in a very unique place. I will admit, there are many reasons as to why to use other front end frameworks (, or if you prefer to not call them frameworks, I understand that as well). For instance, Vue, is very simple to use. React, is always on the cusp of cutting edge. Reason in particular as of this writing is simply incredible. Cycle, in it's approach towards functional programming, is refreshing. However, Angular specializes in consistency, therefore productivity, as well as safety.

It does not surprise me that the most robust command line, by a long shot, is with Angular. Everything, from how to style a component, folder structure, routing, observables, state management, type checking, is agreed upon by the entire Angular community. It is a very safe bet when building out an enterprise application. It makes architectural decisions very easy, and it does have the ability to layer on newer technologies, if need be, albeit harder than with some frameworks such as React. It does not surprise me, that the first full gamet book will be written on Angular. It towers over the rest in consistency, and that deserves a head nod.

The issue with many technical books, is that as a reader I will walk away from it feeling like I scratched the surface. That is, I am not confident I have covered the full gamut of that topic. So that if I plan on building it myself, I still have to do research on my own. In addition, when I do plan on building an app, I do not have an example app to work against. Essentially making the book useless. Like seriously, it has me wondering the benefit of many of the technical books I've reading lately!

As of now, it pretty much helps for discovery, and for maintaining new material. However, learning it does not help. This book on the other hand will help you, really one of the first of it's kind and sort of revolutionary. By offering a QR code, with the latest commit, linked to a Stackblitz, you will simultaneously be able to follow along while reading the book, seamlessly. In addition, when you have read the entire book, and need to reference

bits here, and there, you can look at a specific commit for a certain part of the book, and remind yourself how to do it once again.

In this the full gamut series, we will build a sample application. The application will be a pixel illustrator. With the ability to draw a pixel on an artboard, and plot the coordinates on the left side. In addition, it will have the ability to change colors on the right side.

The intent of the book, is that it will be built in the most cutting edge fashion. The reader should be walking away knowing, that what they have learnt from this book is best practices, as well the Full Gamut of Angular. Being confident, that they are aware of the full scope of the Angular ecosystem at this time.

In addition, conventions will be casually sprinkled from time to time. Conventions are on wide spectrum of impact. Some conventions beckon being chosen, while others are chosen without the thought of doing so. From an architectural perspective, they are equally as important, because choosing them ahead of time, will save days of unnecessary re-factoring. This book as a result will also consider conventions as part of it's architecture, and will put them in a blue box.

In addition, this book acknowledges that there are three parts to learning:

1. Discovery

2. Maintaining

3. Learning

Some parts of this book are discovery, and some parts are learning. This are parts that every book has, but this one would like to also add maintenance to the mix. As we build out our app throughout this book, we will also repeat steps when possible, that we have already learnt.

Part of the value of this book as well, is if this is something that you would like to implement, the time can be minimized by using the content mentioned in this book.

In addition, for the core part of this book. I tried, to make the core documentation as similar to that, that can me found in the documentation, to make it as least confusing as possible for the reader.

Some might consider this book opinion. However, after working at 5 different companies, I can say with confidence that they all could have used the same architecture, and component library. That is, if you have web application, that is data heavy, trying to allow user to interact with said data, creating, removing, updating, and deleting, this architecture will work for you. It is therefore my opinion that this is the singular best architecture for Angular, and I feel 100% comfortable calling it the Definitive Guide.

Also, the idea of a good architecture, isn't necessarily so that it goes through everything. The idea of good architecture, is so that if something new comes up, you will be able to drastically change your codebase.

This book is going to be app agnostic. However, it is going to work against the app in order to show how technology should be integrated in real time. In addition, there are going to be commits made in the repo alongside the book, so that it can give a better idea of how this architecture will work in real time.

I had to learn all of this for the first time too, so I am extremely empathetic towards explaining things in a concise and clear manner. In addition, Angular is a ginormous ecosystem. Between all the nuances with RxJS, Angular it's self, `@ngrx/store` it is so easy to forgot without constant repitition. I therefore tried to find the happy medium between the three potential scenarios:

1. Expert, looking to organize thoughts on architecture, and learn prior unknowns.

2. Expert who needs to refresh due to time elapsed.

3. First time learner.

Whenever the opportunity arises, this book uses number listing, e.g.:

1. item one

2. item two

We at Razroo feel it is much easer to think in numbers, versus having to quantify for one's self the content in a paragraph.

Each chapter is meant to be looked through thoroughly before one goes ahead and one actually does actual work on them. For instance, for the chapter on state, it goes through the architecture for all chapters on the ngrx/store. Only once it does so, will it go through the folder/file architecture for state.

1.1 Enterprise examples in Core

One other thing that this book does, is that contrary to many other books of this nature, it will include enterprise examples in the core chapter. This is because we are assuming that those who are reading this book mean business, and at the very least, want to be on a level where they can develop enterprise applications.

1.2 We Have Best Practices

This book is developed around the idea that we have best practices for how to develop an enterprise level UI application within Angular. However, some of these best practices are beyond the UI Engineer's capabilities, such as choosing GraphQL for data querying, or E2E testing. For those scenarios, we give a head nod towards those technologies.

2 Dependency Graph

A dependency graph is a very simple way of seeing which components are dependent on which components. It goes hand in hand with a parent and child component architecture. By looking a dependency graph, one can see that a singular component is reliant on another. The following are the benefits of using a dependency graph:

1. Make sure app is following Parent/Child Component architecture.

2. Allows us to see what components are dependent on the one we are working on, so that we may run linting, or unit testing only, based on these components.

3. Visualize all components in use across app.

2.1 Create a Dependency Graph

My personal favorite when it comes to using a dependency graph is compodoc. However, when I am in development mode, and not demo mode, I will tend to use the dependcy graph provided by nrwl, granted you are using nx.

```
npm run affected:dep-graph;
npm run dep-graph;
```

This will show you something like: (Image of depedency graph goes here)

2.2 Using Dependency Graph

3 Angular CLI

None of the code being generated in this part of the chapter will be used in the actual app we are using. Hold off on coding along(if you are, until the next chapter. This will be the first and last time this will happen)

The purpose of the Angular CLI(Command Line Interface) is: "The CLI is responsible for automating away many of the challenges and headaches that come with being a developer in 2017. By starting with a configuration that works out of the box, baking in best practices that have been discovered by the community over time, and by helping you build more maintainable code, the CLI exposes some of the most powerful capabilities of Angular in a way that is easier to get started, and easier to get moving quickly when compared to rolling everything together yourself." [1]

The following are a couple of things which you are able to do with the Angular CLI:

1. Create an application out of the box
2. Generate a module
3. Generate a component
4. Generate a route
5. Generate a service
6. Serve application
7. Test application
8. Lint application

Just to get an idea for yourself, of what the Angular CLI can do, and how to generate files, let's run through the CLI high level . Let's install Angular CLI:

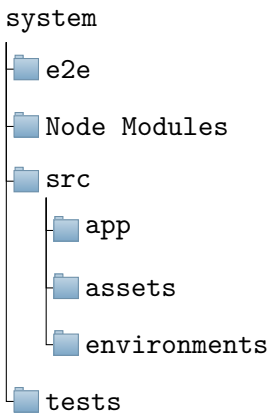
```
npm install -g @angular/cli
```

[1] https://blog.angular.io/the-past-present-and-future-of-the-angular-cli-13cf55e455f8

Next, inside the directory that you would like to genrate your application, run:

```
ng new pixelIllustrator --service-worker
```

[2] This will create a pixelIllustrator folder [3], with all of our new angular code inside of it. The directory structure looks as follows:



3.1 Ng New Notable Mentions

End to End Integration Created for us is an e2e folder. We will be using this for integration tests.

Linting A tslint.json has been created. We can add linting rules that we so choose here, and run linting using ng lint [4].

App folder Inside of the src folder, a folder has been created for styling, components, modules, and unit testing. We will add onto this file directory, moving forward.

Assets Folder Any images, fonts, or otherwise go in this folder. When the angular cli command for pushing to prod is made, it will transfer over the assets folder to the prod folder. This will make it so that filepaths, as long as relative, will still work as expected.

[2] We will be using service workers, and using the above will save us time later on.
[3] Angular CLI version as of this writing is: 1.6.1
[4] Add note with regards to having a watcher set up

3.2 Hold on - It's not going to be that easy

The Angular CLI is fantastic, and has removed a tremendous amount of effort from getting our project started. However, the idea as mentioned in the Angular CLI, is that the Angular CLI is meant to expose the most powerful parts of Angular, but it isn't the full package. This issue arises twofold. It doesn't impart to us how to use some of the more complicated parts of the created application, in particular Typescript, Observables, routing, and styling. In addition, there are things which we will want to add to our application, which the Angular CLI has not provided:

1. State Management
2. Sass(The CLI will give us the option to do so)
3. Workspace(Creating a mono repo using Nrwl Nx)
4. Library(Also going to be created using Nrwl Nx)

That being said, let's come back to the Angular CLI shortly when we are ready to be build our application. First, however, let's take one step back. Let's talk about Nrwl Nx and turn our app into a workspace.

4 Introducing Nrwl Nx

4.1 Arguably The Greatest Strength of Angular

Arguably one of the greatest strengths of the Angular(2+) ecosystem is consistency(a point we've discussed before). As a result, Angular has the most advanced Front End Framework CLI.

4.2 Angular CLI Shortcoming with Regards to State Management

However, the Angular CLI does not deal with state management. State management in Angular2, for the most part should be done using NGRX. State management in it's self, however, can be unwieldy. For every new reducer, a new constant file, as well as action, effect, and proper unit testing files are needed. In addition, the Angular Framework is not built around state management. It is easy for it to fall through the cracks, and for an app to be done without it in the first place. Introducing the Nrwl nx CLI.

4.3 NX CLI

NX is built by a team called Nrwl. They are perhaps thought leaders in the Angular space, as they well should be, being that a large part of them came from the core Angular team. They decided to build a cli around state management. However, using their cli comes the following pre-conditions, include the NX Workspace.

4.4 Introducing the NX Workspace

One of the things that the Nrwk team really tries to push, that isn't mainstream yet, is the concept of a workspace. Perhaps you will remember an article floating around a while back, about Google's Mono Repo. [Worth noting, this idea has been popularized at Facebook as

well]. The idea is that there is a singular repository for everything. The benefits of such are (Taken from Nrwl site):

1. Unified versioning
    a) Everything at that current commit works together
    b) A label or branch can capture the same
2. Promotes code sharing and reuse
    a) Easy to split code into lib modules
    b) Easy to consume implement that code and the latest changes to it
3. Easier dependency management
    a) One node_modules for all code
    b) One build setup (like the AngularCLI)
4. Refactoring benefits
    a) Code editors and IDEs are "workspace" aware
    b) Can have a single commit for a refactor that spans applications in the domain
5. Consistent developer experience
    a) Ensures all necessary dependent code is available

Some of the biggest disadvantages include:

1. Taking time to limit access to part of workspace.
2. One upgrade in a lib, changes all areas.
3. Make it overkill to work on a small feature.

4.5 Why We Love the Iea of a Workspace

Code Re-use In any large application, code re-use is key. Any corporation not using a workspace, will have to create a a separate npm repo, for libraries and code shared. A shared

workspace makes it easy to share code, and have it available for all to see. A separate app can be created for users to see code created if need be.

It Emphasizes Smaller Modules Being that in a workspace environment, things can become unwieldy by using the tree directory, there is more of an emphasis on keeping all of the code, specific to that feature, service, or component, in the aforementioned folder. It makes it so everything is in the same place, and that is a perfect place to move on the to the next point.

4.6 Let's Begin to Build Our Application

First we are going to want and make sure that we install both the Nrwl Schematics and the Angular/Cli. The Nrwl Schematics are an extension to the Angular CLI. There are soem differences in the code structure between what Angular CLI expects and what the workspace creates. The schematics work to patch those differences so we can still use the Angular CLI, without noticing a difference.

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

The nrwl/schematics package comes with a binary for creating a workspace. In the directory of your choice [1]. Run the following command:

```
  create-nx-workspace angularPixelIllustrator
```

This will create a folder called angularPixelIllustrator which will contain the following files/-folders:

```
apps/
libs/
.angular-cli.json
tslint.json
test.js
tsconfig.json
tsconfig.spec.json
```

[1] Personally, I have a directory called GithubProjects.

4.7 Nrwl nx notable mentions

4.7.1 apps folder

All application that we will be creating will go into the apps folder.

4.7.2 libs folder

All code intended to be shared by any one of our apps, will be created in the libs folder. It is a secret weapon.

4.7.3 tsconfig, test, and linting

Universal configuratiomns for Typescript, unit testing, and linting. It will be used across all applications apps folder, as well as all libraries in the lib folder.

5 Compodoc

Having a documentation tool within your app is incredibly important. The reason it's important, is that it increases visibility of your application in two areas in particular:

1. Overview
2. Dependencies
3. Various types of Modules, Components, Services etc.
4. Routes
5. Documentation coverage

Let's go through how to install compodoc, and the areas we have analyzed it is useful in, regarding documentation.

5.1 Install Compodoc

npm install -g @compodoc/compodoc –save-dev

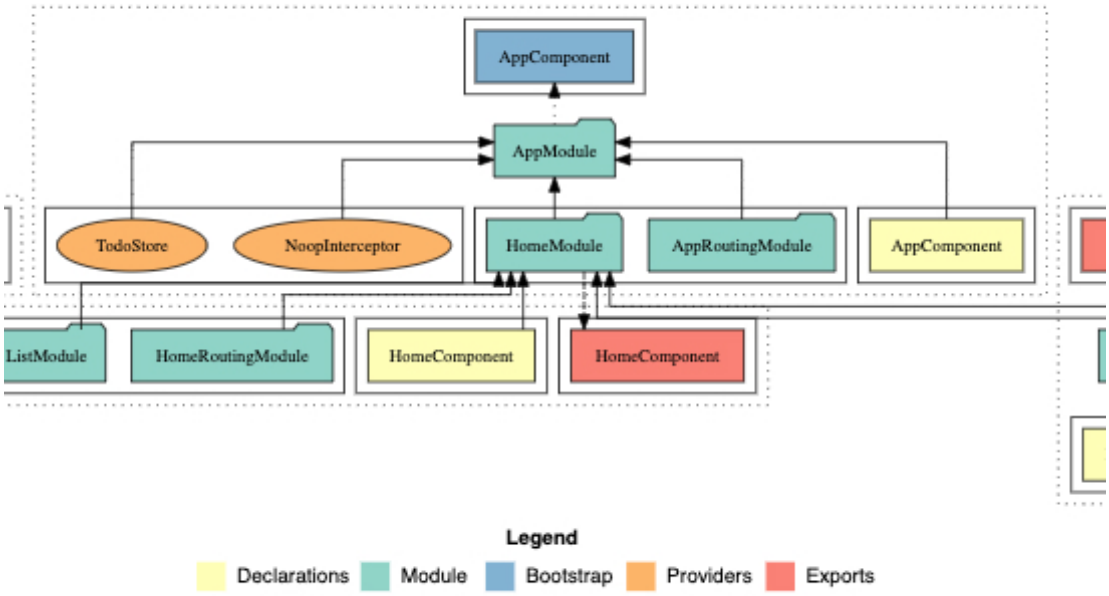
5.2 Add a package.json script

```
"compodoc": "compodoc -p tsconfig.json",
"compodoc-serve": "compodoc -s tsconfig.json",
```

Now if we want to generate documentation, and we are in development mode, we can simply run:

```
npm run compodoc-serve
```

Figure 5.1: Compodoc overview



5.3 Overview - Using Compodoc to Remove Dead Code

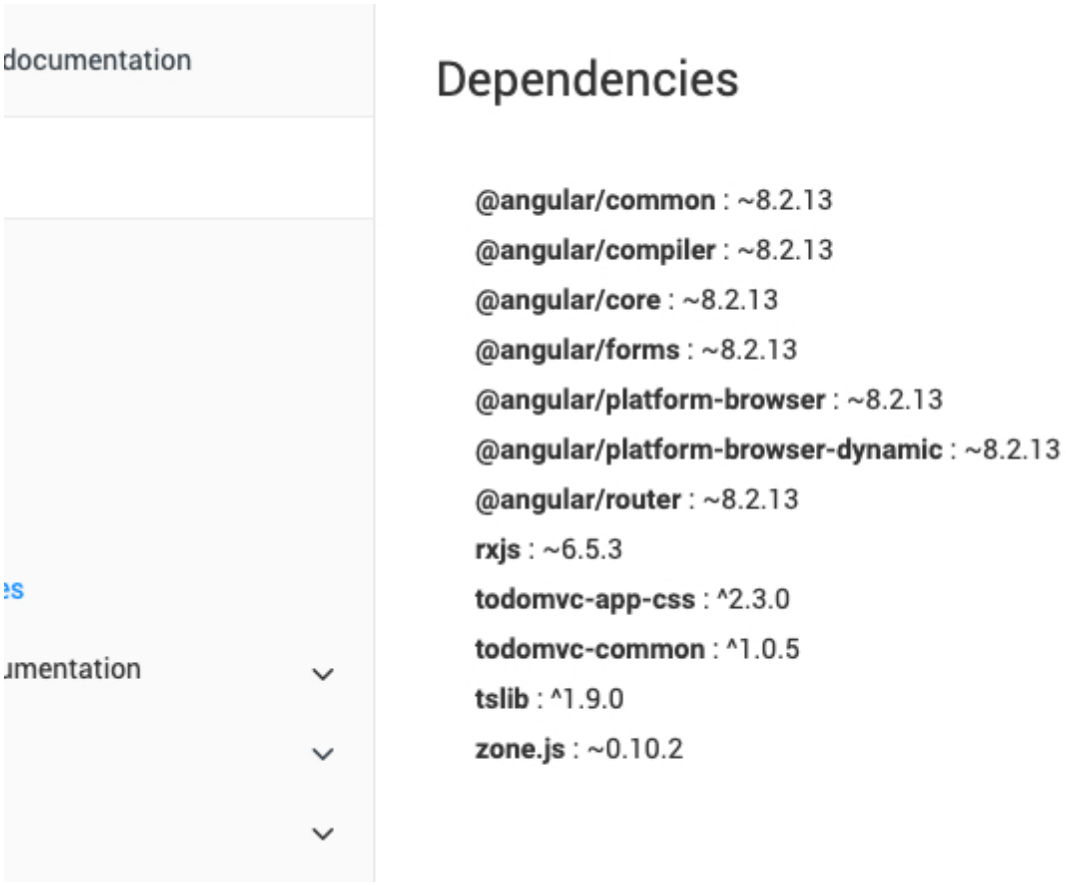
Compodoc has a section called overview. It allows you to see all of your modules, components, and services. By doing so, you can see how they all feed into each other.

In the above image, we can see how Compodoc works. We can very clearly see what components are feeding into which module. If too many lines are randomly feeding into each other, then we know our app needs to be better organized. Also, if we have any components that do not feed into any modules, then we can clearly say that it is dead code, and should be deleted.

5.4 Dependencies

Seeing the depedencies within your application is usually as simple as going into your package.json. However, I've personally found that the package.json usually does not receive the attention it needs, respective to it's importance. Using Compodoc, it sticks out like a sore thumb, and does make you keep it in mind. So if this was the only feature, it wouldn't have that much value. However, I do like the fact that they did include it.

Figure 5.2: Compodoc overview



So we can look through the above and immediately say, "oh ok", this is what version of Angular we are on. I really like it for that reason, because, once again, the package.json has a way of creeping up on a project. This makes sure that it is front and center.

5.5 Various Types of Modules, Components etc.

Compodoc also has offers dropdown of all modules, components, classes, services, interceptors, guards, interfaces, and miscellaneous within app. I like this for two reasons:

1. It lays everything out there, so I know everything that's going on within my app.

2. Also defines all the things that we should see in our app. In particular, I like the sections for interceptors, guards, and interfaces. It can leave an impact on the team, that these are things that should be written.

5.6 Routes

Another one of the favorites is the ability to visualize all the routes that are in your application, and which components are attached.

It also gives you the ability to click through the routes and see which components are using a particular route.

5.7 Documentation Coverage

Documentation coverage is another cool item with regards to compodoc. It's something that already exists in a number of different frameworks. However, it is nice that compodoc includes it, so all documentation is included in one place. In particular, the flavor of documentation it uses is JsDoc. For instance, if we have a method within our component, and it looks some like this:

```
1  /**
2    * Display only completed todos
3    */
4  displayCompleted() {
5    this.currentFilter = 'completed';
6    EmitterService.get(this.id).emit('displayCompleted');
7  }
8  /**
9    * Display all todos
10   */
```

Figure 5.3: Compodoc navigation of Modules, Components, etc.

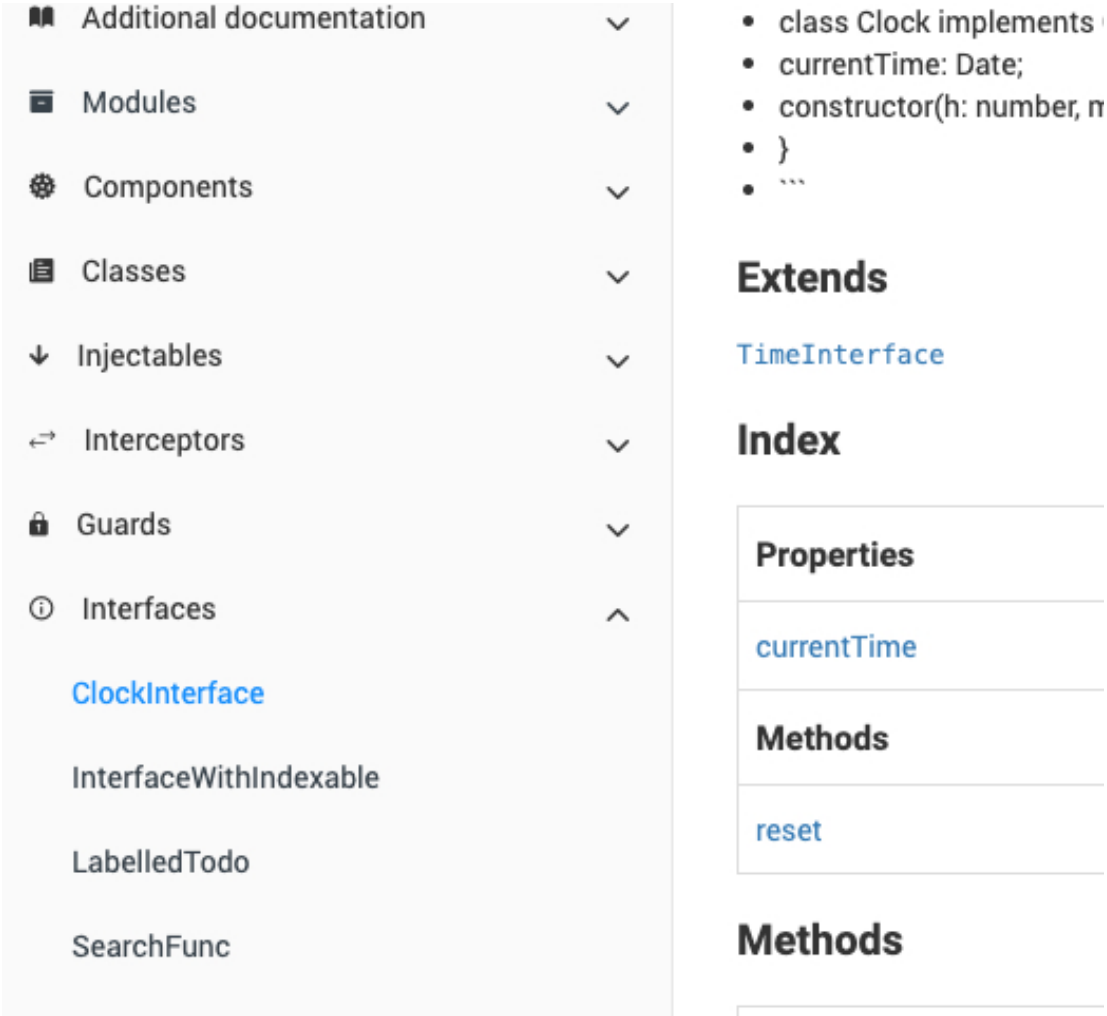
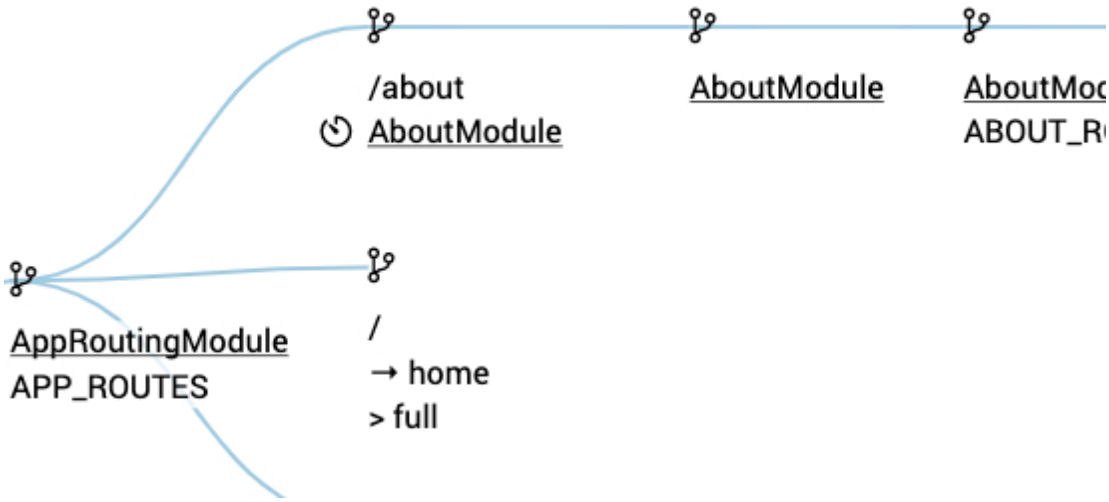


Figure 5.4: Compodoc Routes Example




```
displayAll() {
  this.currentFilter = 'all';
  EmitterService.get(this.id).emit('displayAll');
}
```

Compodoc will take the JsDoc documentation and generate it in a way that is very, very easy to navigate.

5.8 Pushing Documentation to Staging Area

As we did initially, is that we created a two npm scripts. One of them for compodoc development, and the other compodoc production. By running:

```
"compodoc": "compodoc -p tsconfig.json",
```

This will generate documentation in your default output folder. You can then push this bundled output to a link, that everyone can go to, to see existing documentation for your app.

5.9 Final Thoughts

There are many people out there that have incredible things to say about Compodoc. Nonetheless, I want to make sure, that anyone I advise on Angular, this is the one of the

Figure 5.5: Compodoc Documentation Example

Methods

displayAll

`displayAll()`

Defined in src/app/footer/footer.component.ts:81

Display all todos

Returns : `void`

displayCompleted

`displayCompleted()`

Defined in src/app/footer/footer.component.ts:65

Display only completed todos

Returns : `void`

things I inform them about. In addition, more importantly I want to advise them on a way that they can go ahead and do this, with the push of a button. Compodoc is the documentation tool that Razroo recommend's time and time again.

6 Using Angular CLI in an Nx Workspace

Now that we have created an nx workspace, let's create our app. Run

```
ng g app angularPixelIllustrator --routing
```

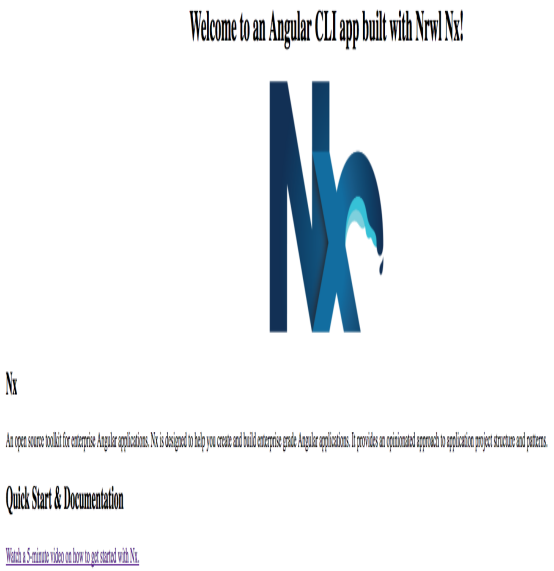
This will create an app called angular-pixel-illustrator[1] with routing capabilities using the Angular CLI [2].

We can now serve[3] our app, by running:

```
ng serve
```

4

[1] that's right angular cli will automatically convert camel case to dash case
[2] If you will recall, we discussed the Angular CLI folder/file directory in the Angular CLI Chapter
[3] I.e. run on a server for development reasons
[4] It's important to note, that ng serve will open up angularPixelIllustrator by default. As we begin to add more apps, it will make more sense to specify specific app being opened.



At this time, if you were to go to localhost:8080 you will see our app, is ready to go.

Let's now create our first component. For our Pixel Illustrator, we want a form. We will name the component chooseSize.

6.1 Wait a Minute!

Before we go ahead and create our component, we are going to want to tidy up our folder architecture. The architecture we are introducing in this book is heavily influenced by two projects. One, Nrwl, and by extension Nx. The other is the example app, introduced in the ngrx/platform repo [5].

6.1.1 Sidebar

At the time of thiwriting there is one main area of conflict with regards to ngrx/store v. Nx. Even though we have not experienced it yet, it makes sense to talk about it before moving on from the cli/nx workspace chapters. Nx is very opiniated with regards to it's folder structure. It believes everything should be turned into it's own module, and all files related to that module should be encapsulated inside of it. This includes (and if you are not

[5] It can be seen here: https://github.com/ngrx/platform/tree/master/example-app

familiar, do not worry, we will get to it in later chaptes) pipes, services, interfaces, guards, and enums.

In the ngrx/example-app project, these will be split into separate folders, and the appropriate file, will be put into that specific folder. I would imagine that many on the ngrx/platform team agrees with nrwl/nx. I most certainly do, and especially with state management, it makes sense for all others items to be encapsulated into their appropiate folder. If it is something that should be shared across app, then it should be put into it's own library. Something that we will discuss moving forward.

6.2 Phew, sidebar over, moving on

The above being said, whenever we create a component, we are going to want to encapsulate it, into a local module. That way we can add state, pipes, services, you name it, and it will all be encapsulated in that component folder.

In order to create our module we run the following angular cli command:

```
ng g module choose-size
```

[6]

Then, in order to create our component:

```
ng g component choose-size --exports
```

[7] The following five files have been created (using git diff –cached)

```
new file:   apps/angular-pixel-illustrator/src/app/choose-size/choose-size.component.css
new file:   apps/angular-pixel-illustrator/src/app/choose-size/choose-size.component.html
new file:   apps/angular-pixel-illustrator/src/app/choose-size/choose-size.component.spec.ts
new file:   apps/angular-pixel-illustrator/src/app/choose-size/choose-size.component.ts
new file:   apps/angular-pixel-illustrator/src/app/choose-size/choose-size.module.ts
```

[6] Once again we have the liberty with not having to specify the app name
[7] exports allows us to use the ¡choose-size¿ component by default

7 Creating Code Owners

As we have discussed previously, Github is our preferred client for pull requests. As the team grows larger it becomes imperative, that default code reviewers are setup. Creating a CODEOWNERS file is the easiest way to do this.

7.1 What is a CODEOWNERS file?

A CODEOWNERS file is a config file, used with Github, that will allow you to specify which github users are considered as CODEOWNERS for a specific project. In addition, it will allow you to target CODEOWNERS for a specific file type. So just to go a bit more in detail, let's say you have a dynamic team, and a dynamic codebase. Part of the team develops in Python, and part of the team develops in Javascript. You would be able to set CODEOWNERS specifically for javascript, so that if only js files have been affected, only these specific CODEOWNERS should be brought up.

7.2 How to create a CODEOWNERS file?

There are two places wherein you can create a CODEOWNERS file. One would be in the root of your app. The other would be in a .github folder in the root of your repo. We recommend the .github folder, as we will also be creating webhooks within our repo. So:

```
mkdir .github; cd .github; touch \codeowners{}
```

When you make a pull request within your github app, github will automatically pick up on this file.

7.3 Creating CODEOWNERS

As we mentioned before, there are two parts to a CODEOWNERS repo. One would be specifying specific github users who would be considered as CODEOWNERS. The other

would be, if the app is more complex, who is considered as CODEOWNERS if a certain file is edited.

```
# Lines starting with '#' are comments.
# Each line is a file pattern followed by one or more owners.

# These owners will be the default owners for everything in the repo.
*        @CharlieGreenman
```

7.4 Creating CODEOWNERS Based on File Type

```
.js @CharlieGreenman
```

Now Charlie Greenman will be a code owner whenever a js file is a part of a pull request.

7.5 Final Note

When using a CODEOWNERS file for the first time, it can be a bit confusing. The CODEOWNERS will only be created once an actual pr is made. However, during the pr process, the CODEOWNERS will not be made. There is no fix to this to the best of my knowledge.

8 Github Wiki

Using the wiki for Github, is really intuitive and there is no reason why a chapter should have to be dedicated as to how to use it. However, knowing what to use the wiki for within an interprise app, can be something that does need explaining. The following are great uses for the github wiki.

8.1 Using Github Wiki for endpoints

If you are using GraphQL, you will still have to setup data per each endpoint. Graphiql client works by setting up data within the url, giving the option to share a url. Once a developer will click on that link, they will be able to test with the data for that GraphQL query without having to try it for themselves.

Creating a wiki for github endpoints is similar. You will create a master page that will specify this is where wiki pages go. You will then have child pages per each endpoint that you can use for putting down links per each graphql query, or mutation. These queries and mutations will include the full data possible for these queries. This will help greatly alleviate overhead of team.

9 Github Board

The github board, for open source projects is used in a task management capacity. In an enterprise setting, it can be used to track the one thing that can't be tracked through JIRA, app considerations.

For instance, let's say you would like to migrate your app to the latest version of Angular. You would create a board called "Tech Concerns". You would then go ahead and create a widget for Upgrade Angular. Everyone on your team would then know that you have certain things that you would like to progress in tech. This can be used for other items as well, such as change directory structure to use mono repo, or integrate E2E tests within app.

10 Format all the things

In a Nrwl setting, we have a format npm script that is available to us by default. It is called:

```
nx format write
```

This will call, a .prettier file that has been created by the nx workspace command. Prettier is similar to to the CLI to the extent that there is a lot that is happening behind the scenes. In short, it will automatically format files for you, to it's liking. It's like a pro-active linter, that will format code for you.

Two architectural talking points, with regards to prettier:

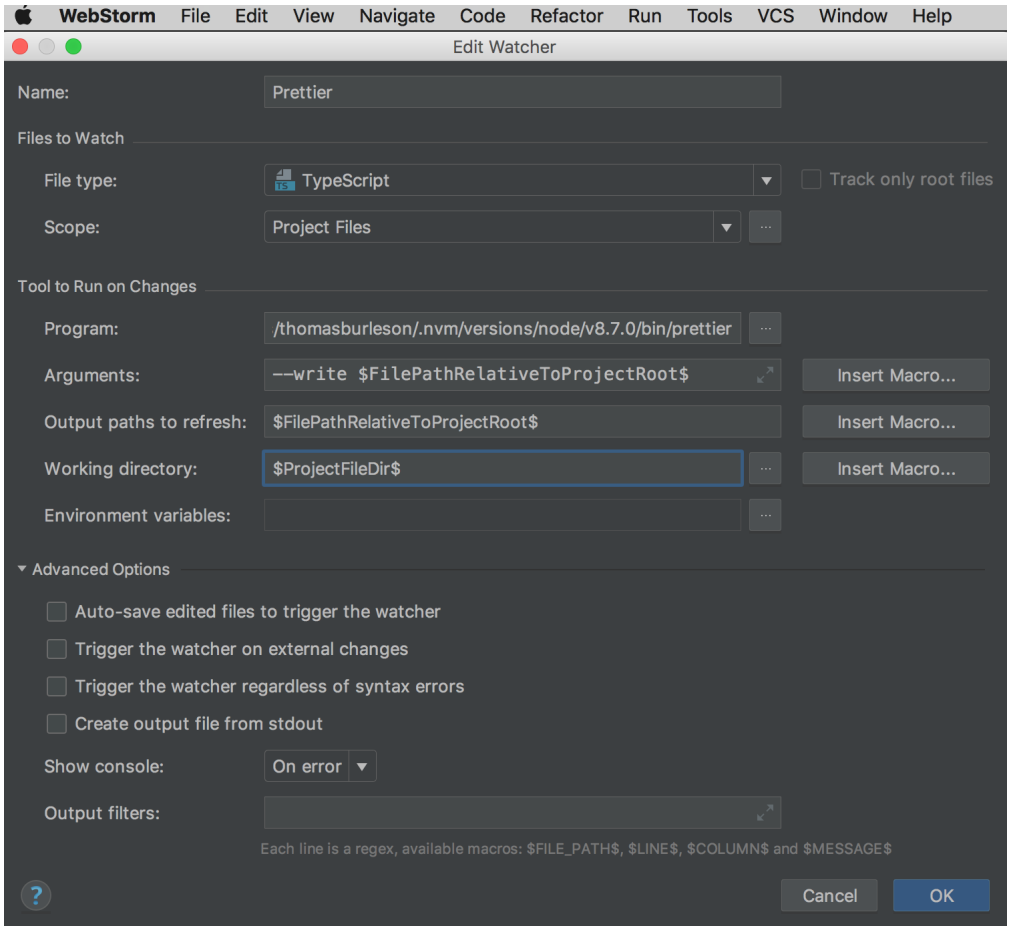
1. You will have to format your tslint, so that it does not compete with prettier.

2. You are going to want to hook up prettier with your IDE, so prettier can go to work without having to run it in your terminal.

Note: We are going to have to create a way that we can update the prettier file automatically.

10.1 How to add prettier to Webstorm

As we mentioned in a previous chapter, Webstorm is our IDE of choice. VIM users and Atom users, I completely respect your decision, and feel free to code in that capacity. However, my experience working with large teams, is that Webstorm has a lot to offer outside of the box. For the non power user, it will offer all of those benefits. Ok, great, the following is how to add prettier to Webstorm with a screenshot!


WebStorm File Edit View Navigate Code Refactor Run Tools VCS Window Help
Edit Watcher
Name:
Prettier
Files to Watch
File type:
TypeScript
Track only root files
Scope:
Project Files
Tool to Run on Changes
Program:
/thomasburleson/.nvm/versions/node/v8.7.0/bin/prettier
Arguments:
--write $FilePathRelativeToProjectRoot$
Insert Macro...
Output paths to refresh:
$FilePathRelativeToProjectRoot$
Insert Macro...
Working directory:
$ProjectFileDir$
Insert Macro...
Environment variables:
Advanced Options
Auto-save edited files to trigger the watcher
Trigger the watcher on external changes
Trigger the watcher regardless of syntax errors
Create output file from stdout
Show console:
On error
Output filters:
Each line is a regex, available macros: $FILE_PATH$, $LINE$, $COLUMN$ and $MESSAGE$
Cancel
OK

11 Lint all the Things

11.1 Lint all the Things

Linting is incredibly important. Think of it this way. Someone's favorite food item might be brocolli, another person's favorite food item, might be steak. Now the two might work well together(on a number of levels, mmm hungry), but forcing one person to adopt the other food item as their favorite, is unjust!

Ok, bad example, but the concept is the same. Things such as how many spaces per indentation, double quotes vs. single quotes, trailing commas, empty interfaces, these are all important points. To be honest, I've only found my peers arguing on more minute points, such as double vs. single quotes, and how many spaces per indentation. Therefore, linting allows:

1. Agreement on code formatting rules.

2. Automated way of keeping track of things that need to be changed.

3. Self documentation through cli, on what needs to be changed.

Another important point, is that we would ideally like a formatter, that automatically formats these things for us. So, from an architectural perspective when we are looking for a linter, we are also looking for a formatter to go along with it.

11.2 What are we trying to lint?

We are trying to lint HTML, SASS, and Typescipt. Angular CLI, offers Tslint out of the box. A Tslint has been created out of the box for the Angular CLI. However, the tslint.json that comes out of the box, will not work well with the prettier file produced by the Nrwl NX CLI. The best method as of this time, is to work through with the prettier CLI and work through differences. There is also a gist which you can copy for your convenience, put here.

Ideally, we would like to create a linter that is agnostic to IDE, so that we can allow the

team to use whatever it is available.

That being said, the Angular CLI only offers a linter outside of the box for Typescript. However, it should, in my humble opinion it should also offer a linter for HTML and SASS. We will therefore take the default linting task, and add three npm scripts on top of it:

```
1   "lint-html": "htmlhint --rulesdir './rules/' '{apps,libs}/**/*.html'",
2   "lint-scss": "sass-lint -v -q",
3   "lint-ts": "ng lint --format=stylish",
```

We will get to the specifics of the above three scripts right now.

11.3 Linting Typescript

As we mentioned Typescript will offer an ng lint command out of the box for Typescript. This should be moved over to it's own lint-ts command to make room for linting for html and scss. In addition, the tslint.json should be updated accordingly to work with prettier.

11.4 Linting Sass

For linting SASS, we are going to be using the sass-lint package.

```
1 npm install sass-lint --save-dev
```

We are also going to want to create a .sass-lint.yml file. Make sure to put the dot before the .sass-lint.yml file, so that the sass-lint linter will pick up on the file by default.

In addition, there is now prettier that it now available for sass. We will have to follow the same formula that we did for Typescript. Worst comes to worst, we will have to modify the yml according to what the linter tells us we can, and cannot do. [1]

Note: Put a link to the appropriate gist here.

11.5 Linting HTML

For HTML, we are going to be using HTML Hint. It can be argued that it hasn't been maintained for a while. Two things: One we are banking on the fact that prettier is going have formatting for HTML soon. Second, how much has html really changed structurally

[1] The appropriate commands for prettier will be added in the following chapter

over the last couple of years. We expect this solution to change over the next couple of years, but for now, we are using ts-lint.

```
1 npm install htmlhint --save-dev
```

We are also going to create a .htmlhintrc file.

There is no prettier yet, so don't worry about this one yet. We'll update the book and let you know when it happens.

12 Linting HTML

Linting html is not as mature as it is with regards to Sass and Javscript in my opinion. Howvever, that makes sense as html is essentially strucutred xml that is used within the context of a web app.

That being said, once again, there is no linter that is offered out of the box through the Angular CLI, or Nrwl Nx either. It is, however, beneficial.

12.1 Side Bar

Why not use something like Pug(once called Jade) for html templating. The main benefit of something like Pug I have found is that it tells me where the html begins and ends. In a complex html element, where there are many levels of nesting this can be beneficial. However, in a templating engines, there tends to be many quirks, in particular when it comes to using various frameworks. I have rarely been on a team where after selling Pug, engineers have been enthusiastic in using it.

12.2 Why We Chose HTML Hint

The honest truth is that the landscape for html hasn't drastically changed at it's core level over the past couple of years. Of course, if there was a more mature linter, the linter would complain that the engineer should use certain html element as opposed to another. That being said, we have found html hint to be the most robust html linter, even though development has been lack luster over the past couple of years.

12.3 Installing HTML Hint

```
npm install htmlhint --save-dev
```

12.4 Create an .htmlhintrc config file

In the root of your app, create an .htmlhintrc file. The .htmlhintrc is set up to be the default config name for html hint. A sample config for html hint will just be a simple JSON object containing key values. For instance:

```
{
  "attr-value-double-quotes": true,
  "src-not-empty": true,
  "alt-require": true,
}
```

12.5 Adding an NPM Script in your package.json

```
  "lint-html": "htmlhint --rulesdir './rules/' '{apps,libs}/**/*.html'",
```

12.6 Adding a Rules Directory for html hint

You will notice that inside of our html hint command, we have created a rules directory. We are doing this so that we can potentially create our own sample html hint rule.

12.7 What a Sample Rule Looks Like

In the root of your directory, create a rules directory. Inside of that directory, let's create some sample logic:

```
module.exports = function(HTMLHint) {
  HTMLHint.addRule({
    id: 'attr-space',
    description: 'Attributes cannot have useless whitespace between "=" and attribute name or attribute value.',
    init: function(parser, reporter) {
      var self = this;

      function handleTagStart(event) {
        var col = event.col + event.tagName.length + 1;

        event.attrs
          .filter(function(attr) {
            return attr.value;
          })
          .forEach(function(attr) {
            var rawAttr = attr.raw;
            var indexOfEqualSign = rawAttr.indexOf('=');

```

```
19              if (rawAttr.charAt(indexOfEqualSign - 1) === ' ') {
20                reporter.warn('Space between attribute name and "="',
                      event.line, col + attr.index + indexOfEqualSign - 1,
                      self, attr.raw);
21              }
22
23              if (rawAttr.charAt(indexOfEqualSign + 1) === ' ') {
24                reporter.warn('Space between "=" and attribute value',
                      event.line, col + attr.index + indexOfEqualSign + 1,
                      self, attr.raw);
25              }
26            });
27        }
28
29        parser.addListener('tagstart', handleTagStart);
30      }
31    });
32  };
```

The above allows for a robust html lint archticture, with the ability to add more rules if need be.

13 Linting Sass

As we mentioned in the previous chapter, there is a greate matter of importance when it comes to linting. In particular when it comes to sass, the Angular CLI, nor the Nrwl Nx cli will offer sass linting out of the box. In particular, we will be choosing the package sass-lint.

13.1 Installing Sass Lint

```
npm install sass-lint --save-dev
```

13.2 Adding a Lint Config File

For sass-lint, it will hook into by default a file that is in the root of folder called .sass-lint.yml. It's quite long, and you can see the rest of the file in the actual github repo. However, you would create a sass-lint.yml file.

```
touch sass-lint.yml
```

Inside of the sass-lint.yml file, it will look something like this:

```
1  options:
2    formatter: stylish
3  files:
4    include:
5      - '{apps,libs}/**/*.scss'
6    ignore:
7      - 'libs/font-awesome/**/*.scss'
8  rules:
9    # Extends
10   extends-before-mixins: 1
11   extends-before-declarations: 1
12   placeholder-in-extend: 1
13
14   # Mixins
15   mixins-before-declarations: 1
```

```
16
17  # Line Spacing
18  one-declaration-per-line: 1
19  empty-line-between-blocks: 1
20  single-line-per-selector: 1
21
22  # Disallows
23  no-attribute-selectors: 0
24  no-color-hex: 0
25  no-color-keywords: 1
26  no-color-literals: 1
27  no-combinators: 0
28  no-css-comments: 1
29  no-debug: 1
```

The list for what sass-lint disallows goes on and on. Some of the linting rules I really like is no color words, empty line between blocks, bem-depth, for starters. From an architects perspective, and from a developers perspective, it has made the code review process much easier. When this linting process is combined with the functional sass paradigms we will mention, it becomes self managing architecture for styling. Let me say that again, SELF MANAGING. I thought that might be worth repeating.

13.3 Adding an NPM Script in your package.json

```
"lint-scss": "sass-lint -v -q",
```

13.4 The Final Touch

Adding the sass-lint npm script as part of your CI/CD architecture is what really brings it all together. This is twofold of course. One so that when you make a pr for your Github repo, it will check to make sure there are no Sass linting errors. In addition, when pr is actually merged, pipeline runs as well, to make sure there are no errors.

14 Accessibility with Codelyzer

Codelyzer if not familiar already, is a static code analyzer built on top of Tslint. It's a layer over Tslint with pre-made rules. Some of which include:

1. Component selectors are kebab-case
2. Directive selectors are camelCased
3. no-host-metadata-property - Disallows use of `host` within components

14.1 Why Codelyzer is Amazing

Codelyzer is amazing, because it's a great example of a situation where one should think harder, instead of work harder. In this instance, by thinking ahead, and finding the right tool, it can make all the difference in your application. True, you could implement these `a11y` compliant linting rules later on. However, having these Codelyzer `a11y` compliant rules available ahead of time, will make you cognizant of everything to begin with. Front loading things ahead of time, while developing new features, will always make things easier long term. Ok, so in practical terms, how does one implement Codelyzer?

14.2 Angular CLI and Codelyzer

Surprise if not already aware, Angular CLI will actually include Codelyzer out of the box. However, Codelyzer will not include `a11y` rules by default, as these are all "experimental". AKA not enough people care about enough to make it a default. [1] So, if you want to go ahead and include these into your application, you will have to physically add them to the tslint.json file.

[1] Which is a shame, because once again, empathy, but I digress.

14.2.1 Install Codelyzer If Not Using CLI

We are assuming you have codelyzer installed already due to the fact that Angular CLI/NX by default installs Codelyzer. If you do not have it installed, feel free to run the following:

```
npm i codelyzer --save-dev;
```

14.2.2 Add rules to tslint.json

The following are the rules that suggested by the Codelyzer team as of now to add accessibility to your application:

```
{
  "rulesDirectory": [
    "codelyzer"
  ],
  "rules": {
    ...,
    "template-accessibility-alt-text": true,
    "template-accessibility-elements-content": true,
    "template-accessibility-label-for": true,
    "template-accessibility-tabindex-no-positive": true,
    "template-accessibility-table-scope": true,
    "template-accessibility-valid-aria": true,
    "template-click-events-have-key-events": true,
    "template-mouse-events-have-key-events": true,
    "template-no-autofocus": true,
    "template-no-distracting-elements": true
  }
}
```

Granted you have tslint enabled within your code editor, there will be two scenarios in which you can take advantage of these rules.

1. From within your Typescript files.

2. Whenever you run `ng lint` over entire project. You can tie ng lint into your CI/CD process, of course, to make sure project is thrown back if these ng lint rules do not pass.

14.3 Accessibility tools in general

It is important to note there are other tools on accessibility that Razroo reccomends as well. We will get to those soon. However, within the context of linting, codelyzer is the only tool you will need in your dev tool chain.

15 Lighthouse

Lighthouse is a framework agnostic piece of technology for auditing web pages. While framework agnostic, it is an integral part of the angular ecosystem. It is the singular most powerful tool for auditing websites. In particular, it is especially powerful for auditing the following:

1. SEO

2. Perforance.

Here is an example of what lighthouse looks like:

15.1 My Concern with Lighthouse

While Lighthouse is a fantastic tool, it is more so marketed as a one off tool, whereing you can check periodically you website. This is great, and fantastic for the web developer-journeyman. However, for the average software engineer working on an enterprise level application, it can seem like a bit frustrating. Lighthouse is such a powerful tool, yet included as CI does not seem to push heavily. That being said, we recommend that you use Lighthouse. In addition, we want to make the CI approach more approachabe, so that this becomes the default approach. I for one am not a fan of how TravisCI is the only CI tool introduced alongside Lighthouse CI. From what my devops friends tell me, whether, or not they are right, is that

15.2 Lighthouse Features

Before we jump into the nitty gritty of setting up Lighthouse CI within our application, I would like to do a deep dive of Lighthouse features. Speaking for myself, when I definitively know that a tool is incredibly useful, I tend to be more motivated to use it. I believe Lighthouse is incredibly useful, and I would liek to show you the same.

15.3 Using Lighthouse CI

The Lighthouse CI is split into two different sections:

1. Lighthouse Node CLI - "Runs Lighthouse, asserts results, and uploads reports."

2. Lighthouse Node Server - "stores results, compares reports, and displays historical results with a dashboard UI."

15.4 Using the Lighthouse Node CLI

15.4.1 Install the Lighthouse Node CLI

```
npm install -g @lhci/cli
```

16 Setting up Schematics Using Angular CLI

In any project, one is ineviatbley going to have their architecture. It would be extremely beneficial for one to create their own schematics. Primarily so, that with a singular command line input, one would be able to generate all of the files they need.

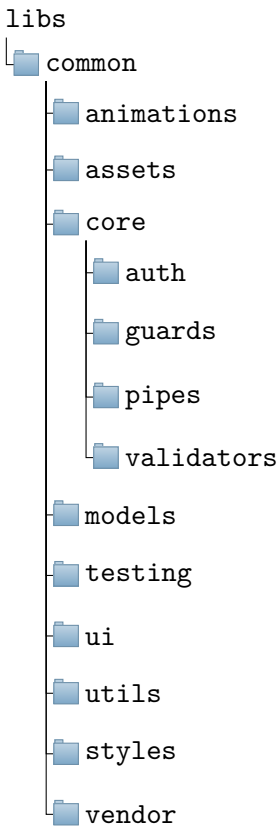
Schematics can be used to easily introduce and enforce project wide conventions. This will easily ramp up time for new developers, as well as diminish time spent on project for current developers.

16.1 Download Schematics Globally

```
1   npm install -g @angular-devkit/schematics-cli
```

16.2 Create a file-directory Schematics

While the application we are building specifically for this book is a pixel illustrator, in practice the conventions we set up for this app, will be used across the app. In a previous chapter we mentioned we mentioned the following folder/file structure:



If we do not enforce this using schematics, it can very difficult for other teams to follow this folder structure. In addition, it will take quite a bit of time to solve at high level.

```
1  schematics blank --name=px-schematics
```

16.3 Understanding Rules and Trees

A Tree is a data structure that contains a base [1] and a staging area [2].

A Rule is a function that takes a Tree and returns another Tree. A RuleFactory are functions that create a Rule.

The blank RuleFactory that we have so far:

```
1  import { Rule, SchematicContext, Tree } from '@angular-devkit/
       schematics';
2
3  // You don't have to export the function as default. You can also have
       more
```

[1] A set of files that already exists
[2] A list of changes to be applied to the base

```
4   // than one rule factory per file.
5
6   export function pxSchematics(options: any): Rule {
7     return (tree: Tree, _context: SchematicContext) => {
8       tree.create(options.name || 'hello', 'world');
9       return tree;
10    };
11  }
```

The options argument is an object that can be seen as the input of the factory. From the CLI, it is the command line arguments the user passed. From another schematic, itâĂŹs the options that were passed in by that schematic.

16.3.1 Tree Deepdive

There are four methods that directly create a change in a Tree:

1. create
2. delete
3. rename
4. overwrite

Similar to how we used create as an example above, to create a file, we also have the option to use the schematic to delete, rename, or overwrite a file.

16.3.2 Generating a Folder using Schematics

We are able to create a series of folders using schematics. However, it is important to keep in mind two things. In any git setting a folder is only committed if it contains a file. In addition, within the Angular schematics it follows the same general guideline. Wherein, by choosing a file to be in a particular file path. If no folder currently exists that is name as such, it will be created nonetheless. I really think the value of schematics cannot be overrated. Simply for it's ability to cause everyone on the team to conform out of mere simplicity. Let's deep dive in another chapter.

17 Schematics Deep Dive

In the previous chapter we have created a px-schematics schematics. The first thing that we are going to want to modify is the collection.json file for px-schematics. Let's add an alias for 'px', as well as create a description.

17.1 Creating an Alias and Description

```
1   "px-schematics": {
2     "description": "Schematic for generating app folder structure",
3     "aliases": ["app"],
4     "factory": "./px-schematics/index#pxSchematics"
5   }
```

17.2 Using NPM Link for development

Being that we do not have an npm module yet, there is a super easy way to hook up our custom schematic, to the actual Angular Schematics. Go into the root of your px-schematics project. For instance, for me that would be:

```
cd /Users/charlie/angularPixelillustrator/libs/px-schematics
npm link;
```

Now go back into your app root

```
cd /Users/charlie/angularPixelillustrator/libs/px-schematics;
npm link px-schematics;
```

17.2.1 What NPM Link Actually Does?

When we ran npm link px-schematics, it automatically targeted our px-schematics folder, and

```
/Users/charlie/angularpixelillustrator/node_modules/px-schematics ->
/Users/charlie/.npm-global/lib/node_modules/px-schematics ->
/Users/charlie/angularpixelillustrator/libs/px-schematics
```

17.3 Creating a schema.json for px-schematics

The schema.json works like a regular schema, telling the CLI(command line interface) what options can be used. In order to add a schema that is specific for a collection create a schema.json file in you collection root. E.g.:

`cd libs/px-schematics/src/px-schematics; touch schema.json;`

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "px-schematics",
  "title": "Add px-schematics support to a app directory",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "Name of the folder to contain files",
      "$default": {
        "$source": "argv",
        "index": 0
      }
    }
  }
}
```

If we look at our schema.json file, we will notice, that it contains a couple of important points. One is that a type field, string in our instance. Second, that it has the familiar description key/value. In addition, we have created name as a default. Therefore if we were to pass our schema an option, without giving it a flag [1] then it will execute it by default.

17.4 Change Folder Architecture

We are going to want to brace for future collections. So let's create a collections folder to put our schematics into.

[1] For instance:

```
px angularPixelillustrator
```

```
mdkir collections; mv px-schematics collections
```

We are also going to modify our collections.json config to let schematics know that we have our own schematics file. Our collections.json for px-schematics should look like the following:

```
{
  "$schema": "../node_modules/@angular-devkit/schematics/collection-
      schema.json",
  "schematics": {
    "px-schematics": {
      "description": "Schematic for generating app folder structure",
      "aliases": ["px"],
      "schema": "./collections/px-schematics/schema.json",
      "factory": "./collections/px-schematics"
    }
  }
}
```

If we were to run

```
ng g px-schematics:px world
```

It will output test as is appropriate.

18 px schematics

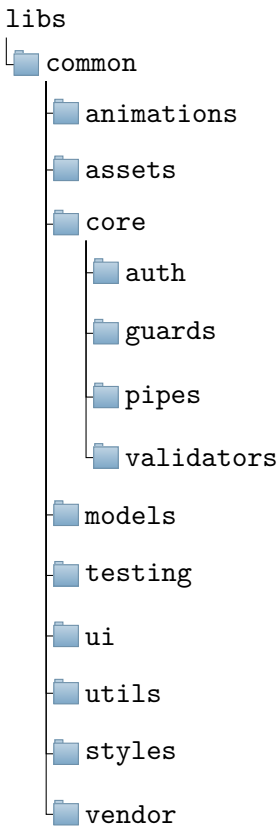
Now that we have deep dived into creating a schematic. We have the particulars with regards to our application. We would like to architect a folder structure that might include some repeat files, such as our app logo.

18.1 A Word to the Wise

First off, there is a need to run npm run build within the repo, everytime that you go ahead and create a schematic. Otherwise, it will not work as expected.

18.2 Analyzing a File Directory

Within our px-schematics, let's create a files directory:



Due to how Angular Schematics works, we can go ahead and create a folder directory as is, and supplant it within our app. We are going to create the following folder directory with placeholder files for our app.

18.3 Creating a Files Directory

We are going to follow a really simply approach when it comes to creating our files directory and follow a functional approach. We are going to create a .gitkeep file for each of our directories so that they can be committed to the app and kept there. It looks something like this:

```
1  import {
2    apply, branchAndMerge, chain, mergeWith,
3    Rule,
4    SchematicContext, template,
5    Tree, url,
6  } from '@angular-devkit/schematics';
7  import { strings } from '@angular-devkit/core';
8  import { libVersions } from "@nrwl/schematics/src/lib-versions";
9  import { DEFAULT_NRWL_PRETTIER_CONFIG } from "@nrwl/schematics/src/utils
       /common";
10
```

```
11 // You don't have to export the function as default. You can also have
       more than one rule factory
12 // per file.
13 export function pxSchematics(options: any): Rule {
14   return (host: Tree, context: SchematicContext) => {
15     const templateSource = applyTemplateSource(options);
16     return chain([branchAndMerge(chain([mergeWith(templateSource)]))])(
17       host,
18       context
19     );
20   };
21 }
22
23 let applyTemplateSource = (options: any) => {
24   const npmScope = options.npmScope ? options.npmScope : options.name;
25
26   return apply(url('./files'), [
27     template({
28       utils: strings,
29       dot: '.',
30       tmpl: '',
31       ...libVersions,
32       ...(options as object),
33       npmScope,
34       defaultNrwlPrettierConfig: JSON.stringify(
35         DEFAULT_NRWL_PRETTIER_CONFIG,
36         null,
37         2
38       )
39     })
40   ]);
41 };
```

18.4 Briefly Discussing Code Base

What we have done in our code, is that we have created a virtual file system, which has been read from our ./files directory. It is then combined with our actual file system. This function is called branchAndMerge. This is a really simple rinse and repeat for generating files within your app.

The documenation for schematics is ever evolving. The project might be specifically under the angular-cli. However, there is no reason as to why this should be limited to one framework in particular. I highly reccomend using it for other frameworks as well.

19 Nrwl Schematics

We have gone into a deep dive with regards to creating custom schematics. However, being that we are working with Nrwl schematics, there is a slightly easier way for us to set up custom schematics within our app. It does tighly couple our schematics with nrwl schematics, and for that reason I didn't like it at first. However, after thinking about it, re-factoring the schematics to be custom schematics wouldn't be the worst thing to happen.

In the previous chapters we created a schematic called the px-schematics. All it did was generate files. For this task, we decided to go ahead and create our own custom angular schematics. However, let's say we wanted to create our own custom schematics, that made use of one of schematics nrwl already has available for us, then it would definitely make sense to use the Workspace Specific Schematics.

19.1 Workspace Specific Schematics

19.1.1 Generate a workspace specific schematic

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

Go to /tools/schematics/data-access/index.ts, where we will add in our custom code.

19.1.2 Adding in External Schematics

```
externalSchematic('@nrwl/schematics', 'lib', {
  name: name,
  directory: schema.directory,
  tags: schema.directory ? `state, ${schema.directory}` : 'state, aero',
}),
externalSchematic('@nrwl/schematics', 'ngrx', {
  name: name,
  module,
  directory: '+state',
  facade: true,
}),
```

```
12 externalSchematic('@schematics/angular', 'service', {
13   name: name,
14   path: 'data-access',
15   sourceDir: normalize(sourceDir),
16   directory: schema.directory,
17   app: schema.name,
18 }),
19 externalSchematic('@schematics/angular', 'interface', {
20   name: name,
21   path: 'data-access',
22   sourceDir: normalize(sourceDir),
23   directory: schema.directory,
24   app: schema.name,
25 }),
```

First we include lib, so that we can choose which directory our schematics should go in. Next we include ngrx, so that state is automatically generated when we create a data-access. As part of our data access, we would like to create a service as well, that will act as our liason between our GraphQL requests, and our actual app. In addition, we will be creating an interface file that will be used across all parts of our app. [1]

19.1.3 Adding GraphQL Files

One of the wonderful things about the Angular ecosystem, that schematics, follows, is the amount of cookie cutter code. Creating your own schematics at first can be daunting, however, the more we deep dive, we will find that we have learnt the bulk of scheamtics. That is file generation and templating.

[1] Feel free to refer back to the chapter on interfaces.

20 Storybook

Storybook is an incredibly useful tool within your Angular arsenal. It was initially created for React, and I pined for the day something similar would be available for Angular. Well that time quickly arrived, and storybook is now available for Angular.

20.1 What is Storybook

Storybook is really good at two things:

1. Enables developers to create components independently.

2. Showcase components interactively in an isolated development environment.

It does this by creating an interactive showcase for each component.

20.2 Using storybook within Nrwl Nx

Nrwl Nx as usual has made things really easy, by offering storybook out of the box.

20.2.1 Generatng configuration for Nrwl/nx

This an be done by running the following command:

```
nx g @nrwl/angular:storybook-configuration project-name
```

This will generate a storybook folder at the root of your workspace if none has been previously created. In addition, this will also generate a project specific folder for your application.

20.2.2 Serve storybook using Nrwl Nx

```
nx run project-name:storybook
```

20.2.3 Cool Features Offered Out of the Box

Some cool features that are offered out of the box for Nrwl/Nx include the following:

21 Data Access Interface

In the previous chapter we discussed having a data access folder to manage the various parts of our data. One of the primary benefits of such an architecture, is that it can be enforced that there is to be one interface. This interace can be used for service, component, state, and respective files.

21.1 The Beauty of a Singular Data Access Interface

By having a singular data access interface, we can ensure that the data is being formatted in the same way across the service, component, and state. In addition, and perhaps more valuable, it allows us to mock all of specs. By having a singular interface used across the app, it forces the mocked functioanlity to be in sync with our specs.

21.2 Singleton Interface - An Example

Just for clarity sake, let's run through singleton interface.

```
export interface User {
  id: string;
  name: string;
  location: string;
}

//user.service.ts file
getAllUsers(sort: object): Observable<AllBuyers> {
    const query = AllUsersQuery;
    const variables = {
      projectId: this.projectFacade.projectId,
      sort,
    };

    const allUserss$ = this.apollo
      .query<any>({ query, variables })
      .pipe(pluck('data'), pluck('allBuyers'));

    return combineLatest(allBuyers$, (allUsers: User[]) => allUsers);
  }

```

```
//user.reducer.ts
  case UserTypes.UserLoaded: {
      const user = <User>action.payload;

      return {
        ...state,
        user,
      };
    }

// user.effect.ts
@Effect()
loadUser$ = this.dataPersistence.fetch(UserTypes.LoadUser, {
  run: (action: LoadUser, state: UserStateModelState) => {
    const userId: string = action.payload;
    return this.service
      .getAllUsers(userId)
      .pipe(map((users: User[]) => new UserLoaded(user)));
  },

  onError: (action: LoadUser, error) => {
    console.error('Error', error);
  },
});

// user-page.component.ts
loadUser() {
  this.userSevice.allUsers();
}

// user-page.component.spec.ts
generateMockUsers(data): User[] {
  return {
    ...data
  }
}

mockUserService() {
  allUsers() {
    return users;
  }
}
```

We have gone through all the files which will touch data with type annotations in the data access folder, for clarity sake. As we can see, having the same interface for all of them will ensure we are using the same data. This is a no brainer as it is what interfaces are, and what they do. However, there are two things that we are doing that which isn't neccesarily considered as intuitive:

1. We have one singular interface for the entire project

2. We are making sure that we use the same interface for our specs.

21.2.1 (

Why this Seems so Flawless) If you were to read the above you might think that this is something which is obvious. However, only by grouping together all of our data-access elements together in the same folder, and keeping with one interface is it obvious. In many other repos, they will put them all in different folers, and maintain different interfaces, causing the entire project to not be in sync.

22 Data Models

A data model is an abstract model, that organizes different sorts of data, and how they relate to each other. Interfaces in Typescript, very much so cater to this cause. Coupling various functionality together is arguably Typescript's greatest strength. Imagine 40 functions using the same data model, which has required properties. If we add a single required property, or change/update a single required property, the compiler will complain on the 40 functions that now to be updated. This is incredibly valuable. There are many more benefits which are beyond the scope of this chapter. However, introducing data-models does introduce an interesting dilemma.

22.1 Dilemma with interfaces in Typescript

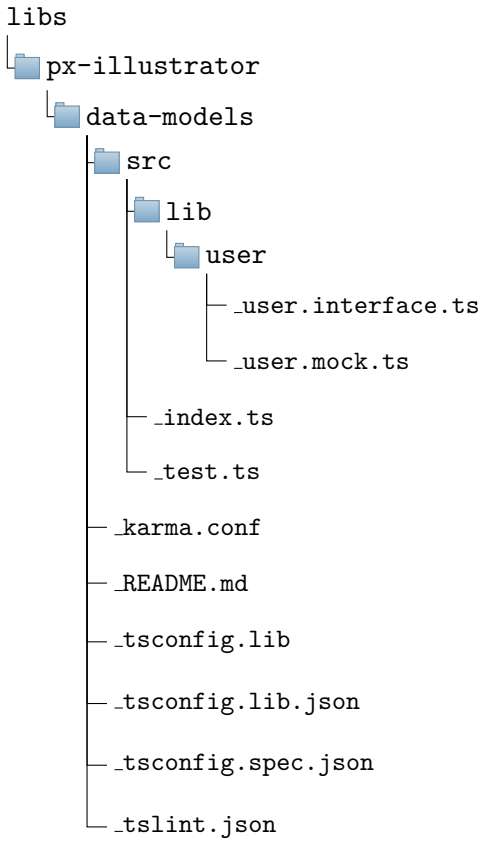
In any enterprise data heavy app, the majority of interfaces will be created to be used in unison with data being retrieved from the backend. In an Angular setting, this means that we will be using a service to make the request. In addition, we will be feeding the data through the entire ngrx/store pipiline, for http requests. Effect ¿ Action ¿ Action ¿ Reducer. In addition, inside of the component that will be consuming this data, we will also need to use the interface. Not to mention, if we want to tightly couple our unit tests to our interface. We want to make sure that there aren't any use cases that we do not test properly. Data types can be unique. We might be missing as a result of data type(array, dictionary, object), some particular use case. Integrating data type with component as well, is important. Keeping the following scenarios in mind:

1. Data Services
2. ngrx/store pipeline (Data Access)
3. Component (Consuming business logic)
4. Unit Tests

What would be the ideal place for us to place our interfaces?

22.2 Data Models Directory Structure

Data Models, are unique, by the fact that they tie everything together. Data Models also seems like an appropriate name for the folder containing these interfaces. It is important to note that not all interfaces will go into this folder, but rather interfaces related to data being pulled in from backend. In addition, interfaces created as a result of interacting with aforementioned data.



22.3 Data Mocks

You will notice that we have also coupled mocks with each interface. These mocks help with unit tests, and by keeping them with the interface, it eases the ability to update when an interface is updated as well. It is obviously of less importance that these be kept to date, as long as general integrity of data is represented closely. However, as a rule of thumb, actually hooking mocks directly into interfaces, so that they closely represent interface, is the easiest way to make sure unit tests cover all use cases.

23 Interfaces and Partials

As we discussed in the previous chapter, interfaces are a large part of data-access architecture. They are the glue that makes sure that all parts of data-access are using the same data schema. However, within the interface, we have the option to use either an optional, real value. The question is, do we want to turn the interface into a real representation of the data in the real world, or do we want to turn all parts of the data as neccesary? The value in having all parts of the interface as neccesary, is that it will force the mock to include all values. However, on the flip side, if we test all services within out graphql application, it will complain the fragments do not match. However, having the interfaces directly complain as a result of the data of the mock not matching is ideal.

23.1 Solution to the Above Dillema

In Typescript there is the option to use:

```
<Partial>
```

Partial will treat an interface that will require every data type, as if all values are optional.

23.2 Where We Might Want a Partial

So that being said, within our data-access architecture we have:

1. GraphQL
2. Models
3. Service/Facade
4. State

Our actual services, facades, and state can take full interface without an issue. They are retrieving data from the back end. In addition, in our mocks.ts file, we want to include the interface fully, to make sure that all potential data is included in mock. The mock for the most part is going to be used across the spec for services and effects.

23.3 Word to the Wise

Ideally interfaces should be a direct one to one correlation between GraphQL queries within app. If any of these items are used, it should be updated, and appropriately updated in the interface. It makes considerable sense to create a tool around this.

TODO asses situations in which a partial would need to be created, if ever.

24 Tsconfig

24.1 What is the Tsconfig?

The tsconfig file corresponds to the configuration of the Typescript compiler.

24.2 What the Default Tsconfig Looks Like.

Angular CLI will generate a tsconfig out the box:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2017",
      "esnext.asynciterable",
      "dom"
    ],
    "baseUrl": ".",
    "paths": {
      "@angularPixelIllustrator/*": [
        "libs/*"
      ]
    }
  },
  "exclude": [
    "node_modules",
    "tmp"
  ]
}
```

24.3 Notable Mention

24.3.1 sourceMap

Source map true, will help debugging while using the console in the browser. However, when debugging unit tests, it will cause somewhat of an issue. Switching this to false at times can be helpful. Alternatively, one can hijack the npm script for npm run test using.

```
--source-maps=false
```

24.4 Using paths

Being that we are using a mono repo, if we were to pull components from within the lib folder, we would have to use a very long relative path. However, tsconfig has an option to specify a specific path. If you look above, in the tsconfig, you will notice that there is a defauly path for libs, called the angularPixelIllustrator.

In most cases the default name will suffice. However, in our scenario, let's shorten the lib path to just ill.

```
1 +       "@ill/*": [
2 -       "@angularPixelIllustrator/*": [
```

Now, whenever we would like to import a module/component from the lib folder from within our app all we have to do is the following:

We are using a barrel file, will get to that in a bit

```
1 import {module} from '@ill/color-picker';
```

25 Component Inheritance

Anyone who has worked with Javascript before frameworks became popular, is well versed with prototypes in javascript. Arguably the most complicated part of the language. In short, it allowed for objects(containing whatever you can think of) , to be inherited by other prototypes. Thereby, allowing one to use classes in Javascript without explicitly using classes.

When it comes to inheritance, JavaScript only has one construct: objects. Each object has an internal link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. null, by definition, has no prototype, and acts as the final link in this prototype chain.

25.1 Extending Classes

In Typescript land, we have the ability to use a class to extend a parent class. A classic example of this is let's say we have the following parent component:

```
@Component({
    template: ''
})
export class BaseComponent {

    constructor(protected utilitiesService: UtilitiesService,
        protected loggingService: LoggingService) {
        this.logNavigation();
    }

    protected logError(errorMessage: string) { . . .}
    private logNavigation() { . . .}
}
```

If we were to try and inherit it, using the following child component:

```
@Component({ . . . })

    export class ChildComponent extends BaseComponent {

    constructor(private childDataService: ChildDataService,

    utilitiesService: UtilitiesService,
```

```
8
9                     loggingService: LoggingService) {
10
11    super(utilitiesService, loggingService);
12
13      }
14
15    }
```

We would unfortunately have to pass all parent providers into the child component to extend it.

25.2 Creating a Class to Store Injector

What we can do, is create a class to store our store injector. For instance:

```
1  import { Injector } from '@angular/core';
2
3  export class AppInjector {
4    private static injector: Injector;
5
6    static setInjector(injector: Injector) {
7      AppInjector.injector = injector;
8    }
9
10   static getInjector(): Injector {
11     return AppInjector.injector;
12   }
13 }
```

We are then able to inject this into the AppInjector:

```
1  platformBrowserDynamic().bootstrapModule(AppModule).then((moduleRef) =>
       {
2     AppInjector.setInjector(moduleRef.injector);
3  });
```

25.3 New and Improved Base Component

Our base component now uses the injector service in order to retrieve all dependencies.

```
1  @Component({
2    template: ''
3  })
4  export class BaseComponent {
5      protected utilitiesService: UtilitiesService;
6      protected loggingService: LoggingService;
```

```
7
8  constructor() {
9          // Manually retrieve the dependencies from the injector
10         // so that constructor has no dependencies that must be passed
               in from child
11         const injector = AppInjector.getInjector();
12         this.utilitiesService = injector.get(UtilitiesService);
13         this.loggingService = injector.get(LoggingService);
14         this.logNavigation();
15     }
16
17     protected logError(errorMessage: string) { . . . }
18     private logNavigation() { . . . }
19 }
```

Now in our child component, we can simply do the following:

```
1  @Component({ . . . })
2  export class ChildComponent extends BaseComponent {
3    constructor(private childDataService: ChildDataService) {
4       super();
5    }
6  }
```

25.4 Where Component Inheritance Really Shines

Component Inheritance is of course, very valuable. However, having a dumb and smart component seems to suffice in many scenarios. However, what is very valuable is when using forms. Inheritance really shines in those scenarios.

6

26 Barrel File

26.1 What is a Barrel File?

A barrel File is a way to rollup exports from several modules into a signle convenience module. The barrel itself is a module file that re-exports selected exports of other modules.

26.2 Barrel File In Practice

In the previous chapter we discussed doing something like the following:

```
1 import {module} from '@ill/color-picker';
```

We are able to do this, because the nrwl nx layer on top Angular CLI, will generate an index.tx file which will contain all imports. Anything that is within the component, that should be exposed outside the lib, should be put in the index.ts(barrel file).

```
1 export { IllColorPickerModule } from './src/ill-color-picker.module';
```

Listing 26.1: index.ts

26.3 Enforcing Barrel File With Tslint

In addition, Nrwl nx has a tslint add on called nx-enforce-module-bounderies.

```
1 // tslint.json
2 "nx-enforce-module-boundaries": [
3       true,
```

```
4       {
5         "allow": [],
6         "depConstraints": [
7           {
8             "sourceTag": "*",
9             "onlyDependOnLibsWithTags": [
10              "*"
11            ]
12          }
13        ]
14      }
15    ],
```

Adding true, will make the tslint complain whenever we are not using the barrel import when accesing a lib file.

27 Typescript - Getting and Setting

Creating a getter and setter is a common occurance in OOP paradigm. When using a class, Typescript offers an internal getter and setter. It is important to note that within the Typescript documenation, a large part of the reason why they create setters and getters, is to give a way of intercepting the setting of a particular element. [1] For instance, let's say that you want to give people the ability for it to error out if it doesn't actually contain one, or two strings. Doing something like that with a setter is great.

27.1 Typescript - Why create a getter and setter?

Another important point, is the syntactic sugar for two reasons.

1. Getting and setting is a integral part of OOP programming. Before redux came around, it was literally a part of every component without a doubt. Having a functiin pegged with a get, or set directly before the function, helps specify the intent of the getter, or setter.

2. While yes it is true, that one would be able to specifiy a function that will act as a setter or getter, it is syntactically, a bit awkward within an OOP setting. If I may:

```
getDinerText()


Diner.text
```

So yes, can you set and get, without using the internal setter and getter that Typescript has to offer like everything else, yes. However, the syntactic sugar will make it seem like you are natively setting and getting. In addition, putting emphasis on whenever an item is getted, or setted.

[1] typescriptlang.orgdocshandbookclasseshtml

27.2 Valuable getting and setting within an ngrx/store setting

So the question then becomes, if you are using ngrx/store within your app, for what reason would you hvae value for getting and setting. For the most part, the values that you have will either be set through @Input's within Angular, or accessed through your @ngrx/store. There is one in particular very valuable reason. That would be if within your Typescript component you have deep nested data. Having a getter and setter within a component would be very useful. For instance:

```
get numberOfDocuments(): number {
  try {
    return this.userLog.user.documents.length;
  } catch (e) {
    return 0;
  }
}
```

Now we have a getter with logic, that says that if it doesn't exist, then it will return something 0. We can then do something like the following:

```
<span> Number of Documents: {{ numberOfDocuments }} </span>
```

in your html.

28 Typescript - Immutability

Immutability is one of the core concepts when it comes to data. I first learnt about it when I was introduced to Redux. However, as time progressed it was something that I learn to integreate with all of my projects. With regards to Typescript, it allows type annotations in the way of Immutability:

```
export interface User {
  readonly firstName: string;
  readonly lastName: string;
}

let user: User = {
  firstName: 'Larry',
  lastName: 'Snow'
};

// This will result in a compile time error
person.firstName = 'Pam';
```

Instead this promotes using immutability. So if someone would like to turn this data into something else, they would do something like the following:

```
export interface NewUser extends User {
  location: string;
}
let newUser: NewUser = {
  ...user,
  location: 'New York'
};
```

29 Declaration Files

Declaration files have an integral part of the Angular/Typescript ecosystem. However, if you haven't worked with declaration files, it's because most major libraries have them bundled with library already. In addition, there are 5,000+ typings created by the open source library DefinitelyTyped. Nonetheless, granted this is the reality of present day Angular development, it is nonetheless integral to understand how it works. Every application usually has a one-off use case that makes it special. When that time arrives, knowing how a declaration file works is going to come in handy. Most likely because you will need to create one yourself.

Declaration files, however, happen to be very involved. They need understanding of numerous prior Typescript concepts. Let's go through that now, as efficiently we can.

29.1 Type Annotation

First, there is what we quite frequently see within a Typescript application called a type annotation. That looks something like this:

```
const userName: string;
```

In the above, we are inform the Typescript compiler, that the type of this constant is a string. This is for a single value. Let's move onto the next step in the ladder, type annotating an object.

29.2 Typescript Interface

A Typescript interface can be used to describe an entire object, such as the following `User` interface:

```
1 export interface User {
2   password: string;
3   userName: string;
4   email: string;
```

```
}
```

Listing 29.1: Typescript User Interface

Now, in our application, if we plan on having user data, we can (type) annotate that object with our interface. A common enterprise example of this:

```
import { User } from '@razroo/data-models/user';

export interface UserState {
  list: User; // list of Users; analogous to a sql normalized table
  selectedId?: string | number; // which User record has been selected
  loaded: boolean; // has the User list been loaded
  error?: any; // last none error (if any)
}
```

Listing 29.2: user.reducer.ts

In the above `UserState` interface, our `list`, which is where our reducer is going to place `user` data, has the type annotation for the `User` interface.

29.2.1 The Multiple Interface Dilemma

Ok, great, so now we know that we can create an interface, and use that to type annotate our object. However, what if wanted to use 10, or so interfaces from the `user.models.ts` file(the file where our user interfaces are located), is there are a more efficient way to import them all at once, instead of doing something like this:?

```
import { User, UserTable, UserSettings, UserForm, UserProject,
    UserCorporate, UserConsumer } from '@razroo/data-models/user';

// let's pretend that all imports are being used in this file, for the
    sake of
// brevity..
export interface UserState {
  list: User; // list of Users; analogous to a sql normalized table
  //..
}
//..
```

Listing 29.3: user.reducer.ts

Typescript's answer to this came in form of the next two core concepts we shall discuss:

1. Modules

2. Namespaces

29.3 Modules in Typescript

29.3.1 What is a Module

A module in Typescript is any file containing:

1. Values

```
// e.g.
export const person: Person;
```

2. Functions

```
// e.g.
export function square(n: number) {
  return n * n;
}
```

3. Classes

```
// e.g.
@Component({
  selector: 'razroo-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  constructor() {}
}
```

29.3.2 Example of Module in Typescript

A great example of using a module in Typescript is the classic `import`. It's so commonplace that you've undoubtedly come across it, and when you read this code example (probably) will say, "Ooooh, that's a module, I know what that is!".

```
1 import { add } from "./math";
```

t̂he math file is a module in Typescript.

29.4 Namespaces in Typescript

You most likely have come across a namespace in Typescript as well. A local namespace looks something like this:

```
1 import * as math from "./math";
```

wherein math is the namespace for the `math` module. Now that we have our namespace, we can tap into any method within the namespace, using classic dot notation:

```
math.add(2, 3);
// 5
```

A namespace put simply, is a way of grouping all Typescript interfaces, classes, functions, and variables under one single name. Similar to what we did above for the `math` namespace. The benefits of this are two-fold:

1. Simplify process of import.

2. Create a respective Typescript interface overlay → for a non-Typescript Javascript library(something we will get to momentarily).

29.5 Global in Typescript - The Last Piece

There is just one last missing foundational piece in order to understanding a declaration file is.Âă

In Typescript, there is the ability to create global variables, functions, and namespaces. For instance, if we want to create a variable called `razrooAssetsBaseURL`and have it be available across our entire app, we can use something really cool called `declare`:

```
declare let razrooAssetsBaseURL = 'assets/logo';
```

Now that we have this global variable within our app, we can use it anywhere we want.

```
1 console.log('razrooAssetsBaseURL');
2 console.log(razrooAssetsBaseURL);
```

Listing 29.4: useless.component.ts

29.5.1 Creating a Global Namespace

We can also create a global namespace, without the need of using an import/export. We would do this by coding:

```
declare namespace razrooLib {
  function makeGreeting(s: string): string;
  let numberOfGreetings: number;
}
```

Now we are all set to finally jump into what a declaration file is.

29.6 What is a Declaration File?

Defined concisely:

> "A declaration file in Typescript is simply a way of transferring over a Javascript library to Typescript"

There is a bit to unpack in this definition, as it's not immediately apparent why a Javascript library would need to be converted over to Typescript? In addition, how exactly would a declaration file covert a Javascript library to Typescript?

29.6.1 Fantastic Moment.js Example

The `Moment.js` library, is an extremely popular library used for dates. (For me personally, in the past 10 years working on applications, it is the only library to consistently be used in every application.) The actual library is written in Javascript. However, in order for the Typescript compiler to understand the Moment library, it is necessary to create a declaration file.

Lucky for us, the Moment.js core contributors have created their own typescript definition files. These definition files are bundled with the moment npm package. Lets look at the moment definition file:

```
1 declare namespace moment {
2   //..
3   interface Moment extends Object {
4     format(format?: string): string;
5
6     startOf(unitOfTime: unitOfTime.StartOf): Moment;
```

```
7       endOf(unitOfTime: unitOfTime.StartOf): Moment;
8       //..
9     }
10    //..
11  }
```

Listing 29.5: moment.d.ts currently at the time of writing 736 lines long chockfull with interfaces for the Moment library

[1].

There are three things that have been done here, in order for this declaration file to take hold:

1. We created a global namespace called moment. Whenever Typescript imports the Javascript `moment` library, it immediately taps into the types for Moment, contained within the global `moment` namespace.

2. We create a type annotation for all of our methods. Here we are showing one of the more commonly used one's - `format`(along with `startof` and `endOf`). Wherein the library specifies that it can optionally take in string parameter, and returns a string.

3. The actual file has the suffix `.d.ts`. When a file has a suffix of `.d.ts`, the Typescript compiler will not immediately know of it's existence. Instead you will have to use a reference path similar to:

   ```
   ///<reference path="path/to/file.d.ts" />
   ```

   The current practice is to place all reference paths in an `index.d.ts` file, and then feed that one `index.d.ts` file into your application. This is fed into main application, by putting this type into the "typings"/"types" field in your package.json

```
1  {
2    "name": "moment",
3    ...
4    "typings": "./moment.d.ts",
5    ...
6  }
```

Listing 29.6: typings field as used by the moment.js library

[1] github.com/moment/moment/blob/0aae7249928ae0dacad94de30d68434cabc03844/moment.d.ts

30 Custom Declaration Files

Earlier we through all the steps neccesary to fully understand what a declaration file is. We brought up an example of the `moment.js` library, and how they created their own declaration file. However, let's say we needed to create our own custom declaration file, how would we go around doing something like that?

30.1 Example Scenario - Taking a Step Back

Ok, so let's take a step back. In what situation would it arise that a declaration file is not readily available, and we would have to create one ourselves? Well, for starters, in another chapter, we reccomend using the native Network Information API for performance boosts.

It is experimental(at the time of this writing). As is the case with all experiemental Javascript technology, the Typescript team has not included core type definitions for it. (Which for clarity sake, as opposed to non-experimental Javascript technology, which they do include core type definitions for). So, we need to go ahead, and create our own custom type definitions.

In addition, there will be times within your Angular application, wherein due to the requirements of your organization, you might need to create your own custom type definition. Primarily for security concerns. For instance, sometimes "typings" will be available for a particular package, however, it might be produced by someone who you cannot trust for your company based on security concerns.

The following is the cookie cutter process for creating a custom type definition within Typescript in Angular.

30.2 Create a Custom Declaration File

Just to re-iterate(as explained in depth in previous chapter)

> "A declaration file in Typescript is simply a way of transferring over a Javascript library to Typescript"

It is important to note that it is common practice to add the `*.d.ts` suffix to a Typescript Declaration file. The `*.d.ts` makes it apparent to anyone perusing through your files, that this is a decleration file. The folder/file structure that we will be using, will drive that point even more so.

So let's go ahead and create our typescript file. In addition, let's be aware of our folder/file directory. We are going to create a lib module called typings. Using `@nrwl/nx`

```
ng g lib typings;
"In which directory should the library be generated?": common
"What stylesheet format would you like to use?": SASS(.scss)
```

In addition, within our `common/typings` folder let's create a `network-information` folder, and a `network-information.d.ts` file.

```
mkdir network-information;
touch network-information.d.ts;
```

30.3 Hooking in our Declaration File to Typescript

It is also important to note that as of Typescript 2.* and greater, the tsconfig.json has two properties available:

1. `typeRoots`: Specifies the folder in which the Typescript transpiler should look for type definitions.

2. `types`: Will target a specific file within your application.

30.4 Example Code as is in Nrwl Workspace

In the root `tsconfig.json` file, the Angular CLI/Nrwl Nx has automatically specify the root `typeRoots` config for use, which is `node_modules/@types`. If we wanted, we could create our own package, and specify another `typeRoots` to be used within Typescript.

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "typeRoots": ["node_modules/@types"],
    "types": [],
    ...
  }
```

```
8   "exclude": ["node_modules", "tmp"]
9 }
```

Listing 30.1: tsconfig.json

and then inside of our tsconfig.json file, Nrwl Nx will automatically generate for us the types that we will be using within our lib.

```
1 {
2   "extends": "../../../tsconfig.json",
3   "compilerOptions": {
4     "types": ["node", "jest"]
5   },
6   "include": ["**/*.ts"]
7 }
```

Listing 30.2: libs/common/services/tsconfig.json

30.5 Adding Our Own Custom Type NPM Package

There are very few scenarios within Angular, wherein we would add our own custom types. I am of the opinion, that if you do find yourself having to add a custom type, let's make it available to everyone. I.e. let's open source our package, and make it something everyone can use. So let's run through the steps of creating our own NPM package.

30.5.1 Creating a Github Repo

We have created a github repo entitled `network-information-types`. We checked the box for initializing with a README, a .gitignore file for Node, and a MIT license. We clone it locally by running:

```
git clone git@github.com:razroo/network-information-types.git
```

30.5.2 Running npm init

Next, we navigate to our newly cloned repo, and run:

```
npm init -y
```

The `-y` tells the `package.json` that we want to use all default options.

```
{
  "name": "project-name",
  "version": "0.0.1",
  "description": "Project Description",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "the repositories url"
  },
  "author": "your name",
  "license": "N/A"
}
```

Listing 30.3: package.json

30.5.3 Install Typescript and modify tsconfig.json

First, let's go and install Typescript as a dev dependency:

```
npm i typescript -D
```

Next, let's create a `tsconfig.json` file and make it look like this:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "declaration": true,
    "outDir": "./dist",
    "strict": true
  }
}
```

30.6 Adding a index.d.ts File

When creating a custom type definition file, generally we will follow a four step process:

1. Create Private Types
2. Create Private Interfaces
3. Create Public Interface

4. Extend Public Interface, to be Used/Consumed by Typescript

In our scenario, we are creating an interface around navigator. So:

```
// W3C Spec Draft http://wicg.github.io/netinfo/
// Edition: Draft Community Group Report 20 February 2019

// http://wicg.github.io/netinfo/#navigatornetworkinformation-interface
declare interface Navigator extends NavigatorNetworkInformation {}
declare interface WorkerNavigator extends NavigatorNetworkInformation {}

// http://wicg.github.io/netinfo/#navigatornetworkinformation-interface
declare interface NavigatorNetworkInformation {
  readonly connection?: NetworkInformation;
}

// http://wicg.github.io/netinfo/#connection-types
type ConnectionType =
  | 'bluetooth'
  | 'cellular'
  | 'ethernet'
  | 'mixed'
  | 'none'
  | 'other'
  | 'unknown'
  | 'wifi'
  | 'wimax';

// http://wicg.github.io/netinfo/#effectiveconnectiontype-enum
type EffectiveConnectionType = '2g' | '3g' | '4g' | 'slow-2g';

// http://wicg.github.io/netinfo/#dom-megabit
type Megabit = number;
// http://wicg.github.io/netinfo/#dom-millisecond
type Millisecond = number;

// http://wicg.github.io/netinfo/#networkinformation-interface
interface NetworkInformation extends EventTarget {
  // http://wicg.github.io/netinfo/#type-attribute
  readonly type?: ConnectionType;
  // http://wicg.github.io/netinfo/#effectivetype-attribute
  readonly effectiveType?: EffectiveConnectionType;
  // http://wicg.github.io/netinfo/#downlinkmax-attribute
  readonly downlinkMax?: Megabit;
  // http://wicg.github.io/netinfo/#downlink-attribute
  readonly downlink?: Megabit;
  // http://wicg.github.io/netinfo/#rtt-attribute
  readonly rtt?: Millisecond;
  // http://wicg.github.io/netinfo/#savedata-attribute
  readonly saveData?: boolean;
  // http://wicg.github.io/netinfo/#handling-changes-to-the-underlying-
      connection
  onchange?: EventListener;
}
```

Listing 30.4: index.d.ts

[index.d.ts]

30.7 Creating our own NPM Package (Continued...)

30.7.1 Adding Ability to Build Typescript

We are going to be using Pika/pack within our application. As of now, I have found it to be the easiest tool to use to build an npm library. In particular, it greatly eases out of the box the need for building a package.json file.

We are going to tap into Typescript to build our application. Let's add a build script to our `package.json` file.

```
"build": "tsc"
```

Next, we are going to add some NPM scripts to our app, so that we can go ahead and build/test our type definitions.

More content needs to be written here:

30.8 Publishing Our NPM Package

31 Design Language System

Creating a design language system is imperative to any architecture. Within Angular the Full Gamut, we are going to assume that you are using Material Design as your DLS. Reason as we discussed before, is that it is the most robust library for creating Angular components. However, there are particulars of Angular that one is going to want to modify. This is where having a light design language system coming in can be very important.

31.1 Identifying Key Points of DLS

The following are the 10 points that are a part of DLS:

1. Colors
2. Styles
3. Icons
4. Grid and Spacing
5. Typography
6. Buttons
7. Form Controls
8. Navigation
9. Cards and Portlets
10. Data Tables

These are arguably the 10 parts of any material application that will be used the most.

31.2 Identifying Proper Architecture

With regards to overriding material design, is the part where architecture kicks in. This is a very important part of the application and I will go through one by one, the parts of the application that have similar architecture with regards to overrides.

31.2.1 Colors

Material design has the ability to be overriden in a sass file. It is important to note, that the material theme allows for overrides using Sass Variables. So, one would do something like this:

```
@import 'src/styles/themes/blue-orange';
@import 'src/styles/material-overrides/material-overrides';
```

Doing something like the following:

```
$gray-50: #fafafa;
$gray-200: #dbe1ea;
$gray-300: #e0e0e0;
$gray-400: #cccccc;
$gray-500: #bdbdbd;
$gray-600: #9b9b9b;
$gray-700: #757575;
$gray-800: #444444;
$gray-900: #212121;
```

Now the colors you have are specific to your app.

31.2.2 Grid and Spacing

This one etc.

32 Material Design

I was debating writing this chapter. The reason primarily being, that depending on the size of your company, you might up end writing your own design system. I completely understand that, and it makes sense if you are a B2C [1] application, or a B2B application. If you are a Business to Business application, then using an out of the box design system probably makes sense. If you are building a cosumer application, I can see how you would want your experience to be unique to that of other websites(granted not working on an MVP).

However, I truly do not understand why a company using Angular, would not want to use material design. It is the most robust design framework that exists within open source. In addition, the documentation for Angular components is next to none. I personally have been in companies where they had a business to business applications and they decided not to use material design. It was an absolute mess! I'll never forget the conversation we had 6 months in, wherein I asked if it was possible for us to get design closer to the internal design system we agreed on, so we can create the custom component! The designers response is a classic! "I thought the developers were doing that on their own! ". Avoiding a scenario like this, is very difficult, and in my opinion, not worth it for many team.

Companies choosing to use Material Design could have saved loads of resources not having to design and implement their own components. It is out of the vast amount of use cases that I see Material Design being valuable, that I have decided to go ahead and write about it.

32.1 Material Design - Talking to UX/UI

This chapter right here, is perhaps why I like Material Design the most. Material Design has documentation for how the UX[2] should work. It also has an Angular Component Library with demos, that I can show off to UX and show them, this is how it works by default. It addition, theming for Material Design, is very easy.

[1] Business to Consumer
[2] User Experience

32.1.1 Theming your Material Design

Putting your own company specific theme on it is generally very easy. In addition, it can help alleviate any concerns those might have of using Angular Material Components, due to it being possible to move over into a different library. From professional experience, I have found the following to be the cornerstone of what your team can expect to customize:

1. Colors
2. Font
3. Spacing(Margin + Padding)
4. Icons(not that this is anything particular)
5. Buttons

The above would be it for starters. As your designs go on, you will have components that you will end up overriding. These will go in a partial sass file, something that we will go into more detail as time goes on.

32.2 Material Design - Create your own Confluence Doc

It is important when working with UX/UI to document discrepencies. For inspiraiton look at the material design docs. The idea is to have a central place where UX can document the differences they have made from the general Material DLS. Something like a Confluence doc(if you are familiar with Atlassian), is a bit excessive. I have found that it's too difficult for developers who spend the majority of their time in code tools to document on confluence. In addition, for designers to spend their time outside of the design tools(e.g. Sketch and Invision).

From a matter of ownership, engineering has a stronger discipline of documentation and organization, due to code being very abstract at times. Engineers should look to take ownership of the confluence doc. However, an Invision doc, seems to be more efficient. Design should look to create an Invision doc, that spans maybe 5 - 15 pages, on the DLS deviations they have from actual Material Design.

32.3 Material Design - Use Invision

It's interesting, because someone might not think of tooling as something which is a part of engineering architecture. However, with regards to finding discrepencies in DLS(Design Language System), Invision is integral. It will make creating comments on particular components as something which will be fluid.

32.4 Material Design - Push Back

The following will be worth alot of time for many different people within your organization. Make sure that your component does not deviate from Material Design. In addition, look into whether, or not it is pre-described for you to go ahead, and create your own components. However, I can assure you designers, product/business, and engineers will all be happy when you go with the default components when possible. When building a product, unless it is beyond the MVP go with what is available for you by default.

32.5 Material Design - Architecture Corner

In a Material Design setting, there will be discrepencies in the design, which we have mentioned above, two ways in order to address, and make sure that engineering is in sync with Design.

However, how does Engineering make sure, that all engineers are adhering to the principles layed out in the DLS. There are two methods which will help to a great extent:

1. Sass functions, with error reporting.

2. Automating UI layer. I.e. overrides for material components across the app.

33 Customize Angular Material Design

Razroo best practices, is that the easiest way to get your app up and running, is to use Material Design within your Angular application. Generally, an organization will want to roll it's own theme into Angular Material. When that happens, the developer will have to go ahead and customize the build for material design. Let's talk about how we can go ahead and do that, and how relatively simple it is.

33.1 Understanding Colors in Material

First and foremost, it is important to understand something called a color palette. I know some of you might be aware of what it is, but personally, I was not aware. A color palette in the digital world, refers to the full range of colors that can be displayed on a device. Within a Material Design application, it refers to the range of colors that can be used within the application. Material's design language makes use of two main colors for it's color palette. That would be primary and seconday values.

33.1.1 Primary and Secondary Values

1. Primary - Color displayed most frequently across your app's screens and components.

2. Secondary - "Provides more ways to accent and distinguish your product." This includes:

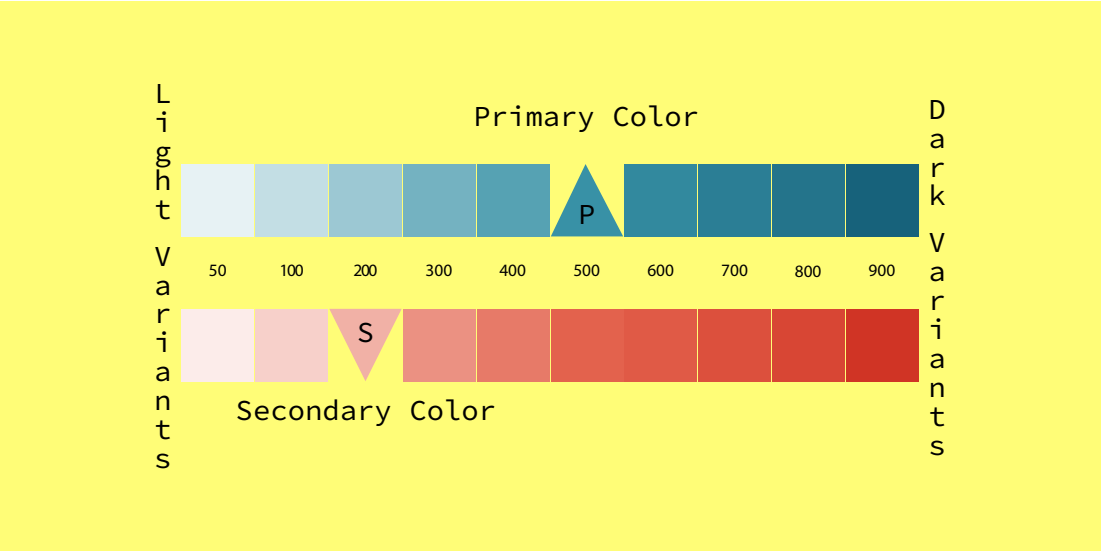
    a) Floating Action Button(Literally buttons that float over main content)

    b) Selection controls(sliders, switches etc.)

    c) Highlighting selected text

    d) Progress bars

    e) Links and headlines



33.1.2 Material Color Maps

Based on a primary value, Material will create a color map to used within different components across it's library. It will then create a series of light and dark variants based on the primary and secondary values. The primary and second values must be a map of colors going from lightest(50) to darkest(900).

The Material team has already created a series of 16 pre-defined color maps, for used with it's design system. An example of material green color map, for instance, will look something like this:

```
$mat-green: (
  50: #e8f5e9,
  100: #c8e6c9,
  200: #a5d6a7,
  300: #81c784,
  400: #66bb6a,
  500: #4caf50,
  600: #43a047,
  700: #388e3c,
  800: #2e7d32,
  900: #1b5e20,
  A100: #b9f6ca,
  A200: #69f0ae,
  A400: #00e676,
  A700: #00c853,
  contrast: (
    50: $dark-primary-text,
    100: $dark-primary-text,
    200: $dark-primary-text,
    300: $dark-primary-text,
    400: $dark-primary-text,
```

```
22      500: $dark-primary-text,
23      600: $light-primary-text,
24      700: $light-primary-text,
25      800: $light-primary-text,
26      900: $light-primary-text,
27      A100: $dark-primary-text,
28      A200: $dark-primary-text,
29      A400: $dark-primary-text,
30      A700: $dark-primary-text,
31    )
32  );
```

For primary values, the most used value will start be 500. For secondary values, the most used value will is 200. The significance of these values is that Material Design will follow a Hierarchical system. The darker the color is, the more of an emphasis we are placing on that button. The lighter it is, the less emphasis we are placing on that element.

You might be wondering about two things. For starters, why is it that the values progress by 100's? Second, what is up with the values that have an "A" attached to the left side? Values progress by 100's is merely a convention used by Material Design. Other design frameworks progress by 10's(IBM Design) instead of 100's, for instance, or even by 1's(Open Color). It is merely a convention used to show that values are progressing.
Values attached with an A on the left side, are Accent colors. Accent colors are colors used for emphasis within a color scheme. In particular, within the setting of Angular, these are used within"

1. Text fields and cursors
2. Text Selection
3. Progress Bars
4. Selection controls, buttons, and sliders links

The contrast color map is used by Angular to display the appropriate text color based on respective non-contrast color map

33.2 Material Design and Sass

In Angular, Sass tends to be the de-facto CSS pre-processor that is used across multiple libraries in Angular. Material Design unexpectedly offers Sass functionality out of the box, and makes it incredibly easy to customize your environment based on sass overrides. Let's install Angular Material, so that we can use Material's Sass Library, and customize the theme as we see appropriate.

33.3 Npm Install Material Theme

First and foremost, let's make sure that we have properly installed and Angular Material in our Angular application.

```
npm install --save @angular/material @angular/cdk @angular/animations
```

You package.json will now include packages needed to use Angular Material within the application in general. In addition, the package (`@angular/material`) to make the Sass changes we so dearly need.

33.4 Import Material Design and Call Core Styles

The next step, is for us to go ahead and import Material Design in our `styles.scss` file. The `styles.scss` file can be found in the root Angular application `src` folder.

```
@import '~@angular/material/theming';
// always include only once per project
@include mat-core();
```

Listing 33.1: styles.scss

You will notice that we are adding a tilda~ next to the node module folder, containing the sass file we need. This tell the sass that file we would like to import is located inside of the `node_modules` folder.

What the above does is import the `theming.scss` file that contains all of the theming variables for material design. We are also calling the `mat-core()` function, which is a,

> "Mixin that renders all of the core styles that are not theme-dependent."

[1]

33.5 Using The Material Light + Dark Theme

Angular offers out of the box in the `_theming.scss` file a light and dark theme function. The function looks as follows:

```
@function mat-light-theme($primary, $accent, $warn: mat-palette($mat-red
    )) {
```

[1] This quote can be found in the `_theming.scss` file

```
  @return (-
    primary: $primary,
    accent: $accent,
    warn: $warn,
    is-dark: false,
    foreground: $mat-light-theme-foreground,
    background: $mat-light-theme-background,
  );
}
```

It takes in two required parameters:

1. Primary - Primary color

2. Accent - Accent color

and one optional parameter called warn, which by default will be red. So, let's say we wanted to create a custom theme based on some of the values that Angular provides. We can use the functionality the Angular Material team provides out of the box. In particular, the `mat-light` theme.

```
$px-app-primary: mat-palette($mat-green);
$px-app-accent: mat-palette($mat-yellow);

$px-theme: mat-light-theme($px-app-primary, $px-app-accent);

@include angular-material-theme($px-theme);
```

Now all of our Angular Material components, will be using our unique theme. (Granted this is based on pre-built colors Angular has provided, but you get the point.)

33.6 Creating Our Own Custom Theme

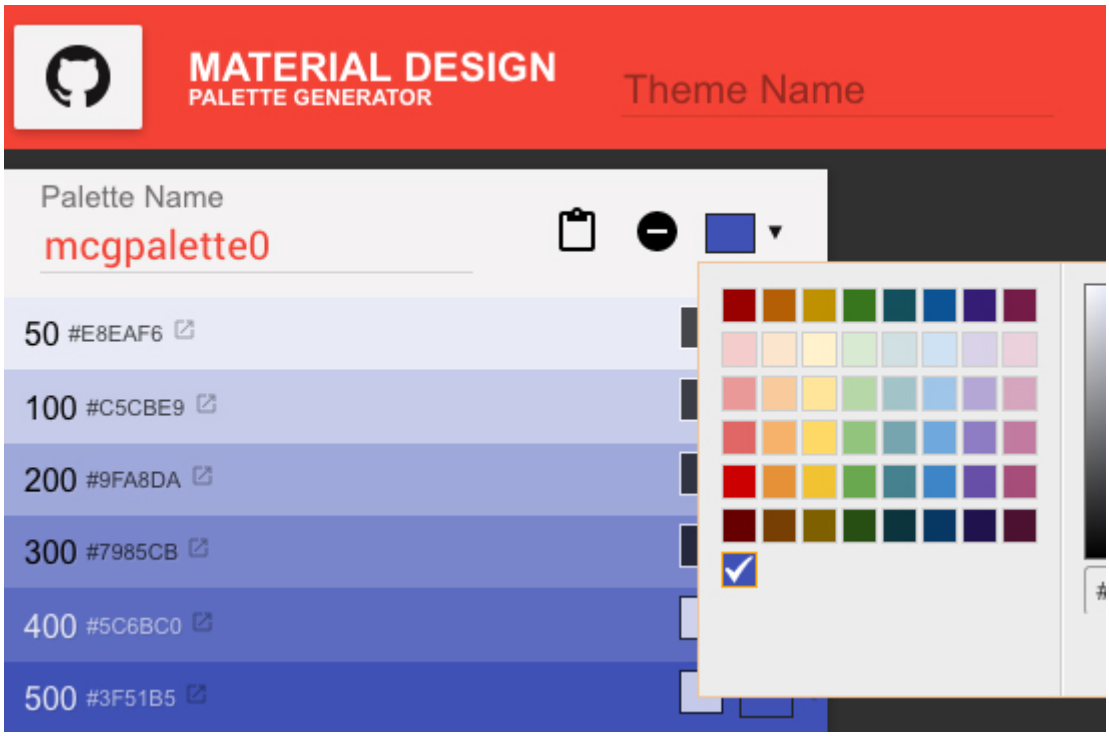
Quite common is that your organization will want to layer their own custom theme outside of the 16 colors that Angular provides. This might manifest it's self in two scenarioes:

1. A new primary and secondary color

2. In addition to new primary and secondary color, a new background and foreground color as well.

Designers have a tool that allows them to automatically generate the appropriate color in their design by supplying a singular color. Angular developers have that luxury as well. There are tools that will do that for you. My personal favorite is the Material Design Palette Generator. You will then have the ability to click on the clipboard icon, click on

the dropdown for Angular JS 2 (Material 2), and copy the scss variable map. It's really as simple as that.



33.6.1 Create a _themes.scss file

Being that we are creating our own themes, the cleanest thing for us to do would be to place it in it's own _themes.scss file. In addition, assuming that the organization is going to build out more applications, giving it the ability to plug and play the companies theme, will really speed up development for other parts of the company. That being said, Razroo best practices is to create a lib folder for styles.

```
libs
└ common
  └ styles
    └ _themes.scss
```

and inside of our styles folder, we are going to create a `_themes.scss` file. Our generated themes using our primary, or seconday colors might look something like this:

```
1 $razroo-primary-blue: (
2   50 : #e7f2f4,
3   100 : #c3dee4,
4   200 : #9cc8d3,
```

```
    300 : #74b2c1,
    400 : #56a2b3,
    500 : #3891a6,
    600 : #32899e,
    700 : #2b7e95,
    800 : #24748b,
    900 : #17627b,
    A100 : #b3eaff,
    A200 : #80dcff,
    A400 : #4dcfff,
    A700 : #33c8ff,
    contrast: (
      50 : #000000,
      100 : #000000,
      200 : #000000,
      300 : #000000,
      400 : #000000,
      500 : #ffffff,
      600 : #ffffff,
      700 : #ffffff,
      800 : #ffffff,
      900 : #ffffff,
      A100 : #000000,
      A200 : #000000,
      A400 : #000000,
      A700 : #000000,
    )
);
```

It's quite a bit of code, but I just wanted to visualize that all of this is created by using the Material Design Palette Generator. I.e. what you can expect when you do the same.

33.7 Using Libs _themes.scss file

Inside of our `styles.scss` file, we can import our `_themes.scss` file. Assuming we are just changing the primary color and secondary color, we can do the following:

```
@import '~@angular/material/theming';
@import 'libs/common/styles/_themes';

$razroo-theme: mat-light-theme(mat-palette($razroo-primary-blue), mat-
    palette($razroo-secondary-red));
@include angular-material-theme($razroo-theme);
```

We now have custom themes that we have created. With the architecture we setup, they are available globally to be used by other applications/teams. In addition , using the Angular mat-light-theme function(mat-dark-theme also an option), or app is now using our exclusive theme.

33.8 Background + Foreground

It is important to mention that per the Material Design guidelines, background and foreground are not meant to represent brand. They are more so used to convey the energy of the application. For that reason the Material Team does not offer a way of the box to change it. For Razroo's Pixel Illustrator(the world's most complete open source enterprise application that Razroo will be open sourcing soon), we wanted to create a very vibrant application. This meant that we wanted to use our own background and foreground colors. Doing something like this requires a bit more of effort probably from the Material team expecting you to do it less. There are four steps required to change the background to what you want.

1. Generate color theme maps, using the Material Design Palette Generator
2. Create a backround theme, and foreground theme for our application.
3. Create our own custom theme function.
4. Creating a default background for html + body, being that this will only work as an override for material design components.

```
1  // Background palette for light themes.
2  $razroo-theme-background: (
3    status-bar: map_get($razroo-background-yellow, 300),
4    app-bar:    map_get($razroo-background-yellow, 100),
5    background: map_get($razroo-background-yellow, 50),
6    hover:      rgba(map_get($razroo-background-yellow, 500), 0.04), //
         TODO(kara): check style with Material Design UX
7    card:       map_get($razroo-background-yellow, 500),
8    dialog:     map_get($razroo-background-yellow, 500),
9    disabled-button: rgba(map_get($razroo-background-yellow, 500), 0.12),
10   raised-button: map_get($razroo-background-yellow, 500),
11   focused-button: $dark-focused,
12   selected-button: map_get($razroo-background-yellow, 300),
13   selected-disabled-button: map_get($razroo-background-yellow, 400),
14   disabled-button-toggle: map_get($razroo-background-yellow, 200),
15   unselected-chip: map_get($razroo-background-yellow, 300),
16   disabled-list-option: map_get($razroo-background-yellow, 200),
17 );
```

Listing 33.2: Example of what a custom background theme looks like.

Your team should have a designer who figures out what the opposite color of your background is, in order to create foreground. However, you can use a tool such as such as Color Tool's Opposite Color Tool. Granted, that you do not have a designer as a resource.

Use the background + foreground color by designer, and follow up with the Material Design

Color Palette tool, and create the appropriate color map.

```
@function razroo-theme($primary, $accent, $warn: mat-palette($mat-red))
    {
  @return (
    primary: $primary,
    accent: $accent,
    warn: $warn,
    is-dark: false,
    foreground: $mat-light-theme-foreground,
    background: $razroo-theme-background,
  );
}
```

Listing 33.3: What custom theme function would look like

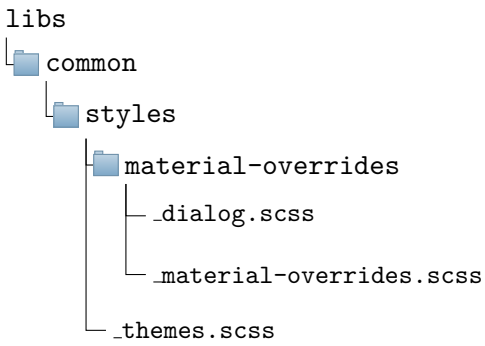
```
$razroo-theme: razroo-theme(mat-palette($razroo-primary-blue), mat-
    palette($razroo-secondary-red));

// Inject $razroo-theme across angular-material-theme
@include angular-material-theme($razroo-theme);

html, body {
  width: 100%;
  height: 100%;
  background-color: map_get($razroo-background-yellow, 50);
}
```

Listing 33.4: html and body override

Angular Material will not change the html and body background. You will have to go ahead, and do that yourself.

33.9 Overriding Components

After we have overridden our theme in general across the app, there will be times wherein we will need to override specific styles for the component. There really isn't any way to modify the styling ahead of time. The only way is to target the specific material component's class, and to modify it when appropriate at run time. However, what we can do, is consolidate all of our overridden components in a singular place, so that it is well organized. For instance, let's say that we have a dialog that we would like to remove the default padding for in some scenarios, but keep it in others. We would put a dialog file inside of our material overrides folder. Our folder/file structure will look like the following:



We import every material override into the main `_material-override.scss` file. Then, import the material-override file in our main `styles.scss` file.

33.10 Overriding Overlay Components

Overlay components are unique from an Angular Material persperctive. You will actually have the ability to target a specific class. If you wish to override an overlay component, overlay components have a `panelClass` property that can be used to target the overlay pane. For instance, with regards to dialogs, we have the ability to add a class doing the following:

```
1 this.dialog.open(PxDialogComponent, {panelClass: 'razroo-no-padding-
    dialog'})
```

Now, we have the option to apply a css theme specifically to the razroo-no-padding-dialog class.

```
1 .myapp-no-padding-dialog .mat-dialog-container {
2   padding: 0;
3 }
```

This gives us freedom moving forward, so that we can have multiple appearances for our material components.

33.11 Overriding Non Overlay Components

Non overlay components, do not have the option to add a `panelClass` to be target using CSS. The only solution which truly makes sense, and is sustainable, is to create a global override on the component itself. So, let's say that we wanted to change the appearance of our material card components. Perhaps make the padding a little less pronounced. We would do the following:

```
1  .mat-card {
2    // this will be 8px
3    padding: rz-space-multiplier(1);
4  }
```

Listing 33.5: _mat-card.scss override

33.12 Wrapping up

Material is, and will for a very long time, be the premier design language for Angular. It is incredibly simple to set up, and a breeze to modify. I am incredibly thankful to the Angular team for all they have done to integrate Material out of the box. In my humble opinion, it is one of the three greatest architectural decisions a team can make for Angular(State management, and extensive unit testing being the others). In my humble opinion, I believe it can knock off 2 months of development time, from a team of 3 over the course of 2 years.

34 Angular Material Typography

Typography at a design level, is a way of presenting text in an attractive fashion. In particular, this is in three areas:

1. Decipher difference between letters.

2. Make blocks of elements, such as paragraphs, or headers, easy to distinguish between each other.

3. Have text that draws you in, or speaks to you as a reader. It should ideally be unique to your brand.

Typography, at a functionality level is a paradox. A it's core, it is not that complex. There maybe are 10-20 different html elements to keep in mind. However, because those 10-20 different elements get used literally everywhere, it makes typography the single most used item in your site. Typography, while it by nature design, and therefore involves less functionality, it should still be treated with the same gravity as UI architecture.

This book, as well as the creators (i.e. Razroo), are strong believers of using Material Design as part of your MVP. After that, moving onto some sort of other design language system as soon as your product is validated. So we recommend the use of Material Design with Angular, and do indeed use it throughout the book. Understanding how to override the Material typography, so that you yourself can do it, and let your message, and brand bleed through the text is important!

34.1 Understanding Different Levels of Typography in Angular Material

First and foremost, let us begin at ground zero. Let's discuss the different levels of Angular Material Typography. The easiest way do this, is to dissect the core sass function for the Angular Material typography config. Razroo has also added comments, to make it appropriate for the context of this book.

```
1 // Represents a collection of typography levels.
2 // Defaults come from https://material.io/guidelines/style/typography.
    html
```

```
3  // Note: The spec doesn't mention letter spacing. The values here come
       from
4  // eyeballing it until it looked exactly like the spec examples.
5  @function mat-typography-config(
6    $font-family:   'Roboto, "Helvetica Neue", sans-serif',
7    // Large, one-off header, usually at the top of the page (e.g. a hero
         header).
8    $display-4:     mat-typography-level(112px, 112px, 300, $letter-
         spacing: -0.05em),
9    // Large, one-off header, usually at the top of the page (e.g. a hero
         header).
10   $display-3:     mat-typography-level(56px, 56px, 400, $letter-spacing:
          -0.02em),
11   // Large, one-off header, usually at the top of the page (e.g. a hero
         header).
12   $display-2:     mat-typography-level(45px, 48px, 400, $letter-spacing:
          -0.005em),
13   // Large, one-off header, usually at the top of the page (e.g. a hero
         header).
14   $display-1:     mat-typography-level(34px, 40px, 400),
15   // Section heading corresponding to the <h1> tag.
16   $headline:      mat-typography-level(24px, 32px, 400),
17   // Section heading corresponding to the <h2> tag.
18   $title:         mat-typography-level(20px, 32px, 500),
19   // Section heading corresponding to the <h3> tag.
20   $subheading-2:  mat-typography-level(16px, 28px, 400),
21   // Section heading corresponding to the <h4> tag.
22   $subheading-1:  mat-typography-level(15px, 24px, 400),
23   // Bolder body text.
24   $body-2:        mat-typography-level(14px, 24px, 500),
25   // Base body text.
26   $body-1:        mat-typography-level(14px, 20px, 400),
27   // Smaller body and hint text.
28   $caption:       mat-typography-level(12px, 20px, 400),
29   // Buttons and anchors.
30   $button:        mat-typography-level(14px, 14px, 500),
31   // Line-height must be unit-less fraction of the font-size.
32   // Form input fields.
33   $input:         mat-typography-level(inherit, 1.125, 400)
34 ) {
35   // ..rest of function goes here
36 }
```

Listing 34.1: @angular/material/_theming.scss

There are a total of 13 typography items, which we have the ability to override. To understand how these feed into general components throughout the site, let's take two a look at two examples. This is so that we can get an intuitive sense as to how we can create our own typography based on Angular Material Design.

34.1.1 Angular Material Cards

Let's dissect the typography Sass mixin the Angular team uses for the material card component.

```
@mixin mat-card-typography($config) {
  .mat-card {
    font-family: mat-font-family($config);
  }

  .mat-card-title {
    font: {
      size: mat-font-size($config, headline);
      weight: mat-font-weight($config, title);
    }
  }

  .mat-card-header .mat-card-title {
    font-size: mat-font-size($config, title);
  }

  .mat-card-subtitle,
  .mat-card-content {
    font-size: mat-font-size($config, body-1);
  }
}
```

Listing 34.2: @angular/material/_theming.scss

1. Title - Uses the config relating to `h2` for size (`$headline`), and weight of equivalent config for `h3`.

2. Title within Header - Uses the config equivalent for (`h3`) through out.

3. Subtitle and Content - Uses smaller body config (`$body-1`).

This gives us a bit of an idea. The config, as expected, directly correlates to the purpose of the mat-card to give extra importance. It also brings home, that mat-card isn't meant to directly encompass main content of the page. (However, once again, it's ultimately you who get's decide how you want to use something.)

34.2 Dynamics of Applying Angular Material Typography Globally

It is important to note, that the Angular Material theme will not by default change native global HTML elements. HTML Elements such as headers (`<h1>`, `<h2>`, `<h3>`), list items(`<`

li>), and <p> tags will not be styled by default. However, Angular Material Design design does have it's own internal typography system, that can be seen here:

```
@mixin mat-base-typography($config, $selector: '.mat-typography') {
  .mat-h1, .mat-headline, #{$selector} h1 {
    @include mat-typography-level-to-styles($config, headline);
    margin: 0 0 16px;
  }

  .mat-h2, .mat-title, #{$selector} h2 {
    @include mat-typography-level-to-styles($config, title);
    margin: 0 0 16px;
  }

  .mat-h3, .mat-subheading-2, #{$selector} h3 {
    @include mat-typography-level-to-styles($config, subheading-2);
    margin: 0 0 16px;
  }

  .mat-h4, .mat-subheading-1, #{$selector} h4 {
    @include mat-typography-level-to-styles($config, subheading-1);
    margin: 0 0 16px;
  }
  // ...
}
```

Listing 34.3: @angular/material/_theming.scss

In the above, we can see that Angular Material stays true to the Material spec, and applies the respective style. I.e.

1. .mat-h1 - headline
2. .mat-h2 - title
3. .mat-h3 - subheading-2
4. .mat-h4 - subheading-1

This is true for all of other global typography HTML elements as well.

34.2.1 Code Example

There is a class that one can use called `.mat-typography` that the Angular Material library provides. To apply it globally, you can apply it on the div wrapper for your content:

```
<div class="page-wrap">
  <razroo-header></razroo-header>
  <div class="content mat-typography">
    <router-outlet></router-outlet>
```

```
5     </div>
6     <razroo-footer></razroo-footer>
7   </div>
```

Listing 34.4: app.component.html

34.3 Realizing That Material Specs Do Not Cover Everything

It is important to realize, that the Material specs will not cover every use case. In particular, this is exemplified by code inside of the `\_theming.scss` file.

```
1  // Note: the spec doesn't have anything that would correspond to h5 and
       h6, but we add these for
2  // consistency. The font sizes come from the Chrome user agent styles
       which have h5 at 0.83em
3  // and h6 at 0.67em.
4  .mat-h5, #{$selector} h5 {
5    @include mat-typography-font-shorthand(
6        // calc is used here to support css variables
7      calc(#{mat-font-size($config, body-1)} * 0.83),
8      mat-font-weight($config, body-1),
9      mat-line-height($config, body-1),
10     mat-font-family($config, body-1)
11   );
12
13   margin: 0 0 12px;
14 }
15
16 .mat-h6, #{$selector} h6 {
17   @include mat-typography-font-shorthand(
18       // calc is used here to support css variables
19     calc(#{mat-font-size($config, body-1)} * 0.67),
20     mat-font-weight($config, body-1),
21     mat-line-height($config, body-1),
22     mat-font-family($config, body-1)
23   );
24
25   margin: 0 0 12px;
26 }
```

As you can see in the above example, core Angular Engineers have had to comment inside of their core code base, that the spec doesn't have anything that correlates, but they have created styling for consistency sake. In the same vein, it is important to realize that Material Design will not cover all use cases that you need. You will have to use your general sense of what Material Design covers in order to fill in the gaps where there is nothing mentioned specifically in the specs.

34.4 Mat Typography Customization

Within Angular Material we have the ability to customize internally the material typography config:

```
@import '~@angular/material/theming';

// Define a custom typography config that overrides the font-family as well as the
// `headlines` and `body-1` levels.
$custom-typography: mat-typography-config(
  $font-family: 'Roboto, monospace',
  $headline: mat-typography-level(32px, 48px, 700),
  $body-1: mat-typography-level(16px, 24px, 500)
);
```

It should be noted, that we have the option to override any of the mat-typography-config variables. To see them, navigate to the Angular Material theming Sass file, and search for mat-typography-config.

Once we have created our config we can use it in one of four ways:

1. Override Typography CSS classes(e.g. `mat-h1, mat-display-1, etc.`)

   ```
   @include mat-base-typography($custom-typography);
   ```

2. Override typography for a specific Angular Material component.

   ```
   @include mat-checkbox-typography($custom-typography);
   ```

3. Override typography for all Angular Material

   ```
   @include angular-material-typography($custom-typography);
   ```

4. Override typography in the core CSS

   ```
   @include mat-core($custom-typography);
   ```

My personal favorite is to use `mat-core`, just because it anyways has to be included atleast once per project.

35 UI Skeleton

Data, very rarely will be immediately available. Even if your app has a very quick backend service, bandwidth due to other web applications the user might be using, can affect the time it takes to load. It can be a very uncomfortable experience for the user if they are unaware of what is happening. I know personally, that I've thought of leaving Spotify, due to their web application having no indicator of when something is loading. Let's do everyone a favor and not do what Spotify does.

Within a design language system, a UI Skeleton, or Ghost Elements, as it should probably be called, is a good idea. Most commonly, it is gray box depiction of a UI component awaiting action, that will be available in the future. It is a very simple way of easing anxiety of the user, and allowing them to be aware of all times, that something is loading.

35.1 One True Way of Implementing Ghost Views

There are three options of implementing Ghost Elements.

1. Creating an Overlay Ghost Element
2. Creating an Inline Ghost Element
3. Inline Ghosts with Async Loads

35.2 Why not to use an Overlay Ghost Element

I would personally not recommend using an Overlay Ghost Element. It requires that developers determine when Ghost it removed. In addition, it requires for layouts to match the "Real" DOM layouts. However, DOM layouts are changing all the time, and this solution is not a good long term solution. Most definitely not an enterprise solution.

35.3 Why not to use an Inline Ghost Element

There is an option to have a css only option. That is, that the css class changes based on whether, or not the data is available. The only real issue with this solution, is that it will be more of an off/on switch with regards to transitioning between a ghost element and an inline element.

35.4 Why use Inline Ghosts with Async Loads

This is a very sophisticated solution, which I first saw from a one Thomas Burleson. The idea, is that we create a queryState function:

```
/**
 * Wrapper function to easily determine async state
 */
export function queryState<T>(item:AsyncItem<T>) {
  return {
    isPolling : (item.state === AsyncItemState.POLLING),
    isLoading : (item.state === AsyncItemState.LOADING),
    isLoaded  : (item.state === AsyncItemState.LOADED)
  };
}
```

Then, inside of our component, we can go ahead and call the queryState within out component.

```
export class UserListComponent {
  state = queryState;
  user$ = this.facade.users$;
}
```

Here, we can create numerous states including polling. This is a more robust solution than using plain css.

35.5 Ghost Elements Always?

There are scenarios wherein a ghost element might not be the ideal scenario.

35.6 Key Take Aways

1. Angular Animations can be implemented as re-usable recipes.

2. AsyncItem is a general pattern used to decorate server entity items with 'client side data state'

3. Each Ghost component is a custom component crafted for that specific component. The odds of it being re-usable is next to none.

4. Ghost grades and annimations are re-usable.

Ghosts might simply be css.

36 Icons

I would like to dedicate a quick chapter to icons, simply because I have never come across an application that has not used icons. In addition, every app has to make the decision, what icons they are going to use. It's interesting, because I had less of a hesitation to write this chapter, than I did for Material Design. With regards to icons, the choice is really ubiqitious at this point. If you are going to use icons, it should be Font Awesome!

36.1 Font Awesome

In short, font-awesome is the largest icon library currently available, and the pro version is relatively cheap. In addition, there are many particulars with icons that they solve, making sure that they look good on different devices. So, in short I would like to reccomend font-awesome. It is the best architectural decision you can make when it comes to fonts. Of course, unless your team has the need to use their own icons. However, before proceeding to do so, let's discuss some of the benefits of using Font Awesome:

1. Pixel Perfect, Consistent Look.
2. 5,000 icons and growing.
3. Ability to play with Size on Web.
4. Accesibility Minded
5. Desktop Friendly.
6. Tried and Tested, by the whole community.
7. Multiple ways to use. CDN, Download yourself, install via NPM.

Font Awesome truly is ubiqitious on the web. In the past 10 years that I have been using it, it is the only consistent tool that has been used across companies.

37 Sass Error Reporting

I have decided to include the chapter on Sass Error Reporting in the chapter for Design Language System. The main reason for this, is that any core style is going to be considered as part of the core Design Language System. However, it can be very difficult to maintain a core design, without some safe guards in place, to make sure it is consistent across the app.

37.1 When to use Sass Error Reporting

One should use Sass Error reporting if it is a core style. It is a core style if it is used in more than one page, as a foundational piece of styling, non unique to specific component.

37.2 What We Are Looking For With Using Sass Functions

1. No values other than these are used
2. When a Pr comes our way, and we say to use the above, we have a function which is self documenting.
3. Have a UI of sorts that also trains developers on how the internal of the DLS works, so that they should be aware if anything is wrong.

The following is a great example of what a core design function would look like.

```
1  // Gutter variables, for padding + margin
2  // function to take in multiplier(8), which must emit of one of values
       within ill-space-amounts
3  @function ill-space-multiplier($n) {
4    $ill-space-amounts: (0, 4, 8, 16, 24, 32, 40, 48, 56, 64);
5    $ill-space-multiplier: 8;
6
7    @if(index($ill-space-amounts, ($n * $ill-space-multiplier))) {
8      @return #{$n * $ill-space-multiplier}px;
9    }
10   @else {
```

```
11      @error "Must contain one of the following numbers: #{$ill-space-
            amounts}.";
12    }
13  }
```

If we have any component that is going to go ahead and use this function, we can simply go ahead and use it:

```
1  @import 'src/styles/variables';
2
3  :host {
4    padding: ill-space-multiplier(2);
5  }
```

In this particular situation this helps, so that if the input passed to the multiplier is not a number, it will complain. In addition, if the result is not one of the multipliers, it will complain as well. So for instance, if the number passed in, is 1.5, it will cause the function to error out, being that there is no number 12, that is one of the ill space amounts.

37.3 Applying Architecture to Design Language System as a whole?

The truth is that this pattern only applicable to padding, and spacing. For instance, we technically could create a function for breakpoints:

```
1  // breakpoints to be used in conjunction with media queries across app
2  @function ill-breakpoint($breakpoint) {
3    $breakpoints: (
4      'small': 400,
5      'medium': 720,
6      'large': 1024,
7      'extra-large': 1424
8    );
9    @if (map-get($breakpoints, $breakpoint)) {
10     @return #{map-get($breakpoints, $breakpoint)}px;
11   } @else {
12     @error 'Must contain one of the following strings: #{$breakpoints}.'
           ;
13   }
14 }
```

However, we reccomend the use of flex-layout. Flex layout currently does not have the ability to use this pattern. Nonetheless, for padding, and spacing alone, this singular piece of sass functionality will probably be used on a daily basis, and is very much so worth it.

38 Introduction to RxJS - The RxJS Airplane

38.1 What is Reactive Programming?

RxJS is a library based on the concept of Reactive programming. To re-iterate what we mentioned in the chapter on ngrx. A great founding paper discussing reactive programming for concurrent programming can be found here. [1]

It discusses the benefit of Real Time Progamming a.k.a., reactive programming. In it, it discusses the two main benefits of Reactive Programming:

1. Asynchronous

2. Deterministic

That would be reactive programming in a nutshell. It makes sure, that one function happens after another. In addition, by it's definition of being a set of pre-made functions, it gives stability around the way we are transforming our data in Angular. I always did, and still love what is in my opinion the most equally profound and succinct quote on reactive programming by Andre Statlz:

> "Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic..."

[2]

I.e. RxJS allows for cookie cutter code, so that after mastering reactive progamming, you can apply those same operators time and time again.

[1] http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf

[2] This quote can be found in his excellent article on why you should consider adopting Reactive programming principles in your app https://gist.github.com/staltz/868e7e9bc2a7b8c1f754#why-should-i-consider-adopting-rp

38.2 How Reactive Programming Allows for Cookie Cutter Code

Naturally, the next question becomes, well how does reactive programming accomplish cookie cutter code. For the sake of brevity, let's jump into a synopsis of RxJS. More effectively, let's talk about RxJS within the context of Angular.

38.3 RxJS's Importance in Angular

RxJS has a pretty big place in Angular. I would say that in your average app, I would consider it as one of the big three, alongside Typescript, and Ngrx. Even though these are independent libraries, when working on enterprise Angular applications, I personally come across them on a day to day basis. So, while they are independent entities to Angular, I very much so consider them as part of the Angular framework.

38.4 The RxJS Observable

The core of RxJS, is the ability to program reactively using observables. The easiest way to understand an observable, is that it's like a promise that can emit multiple values, over a series of time. (A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred)). RxJS offers the ability to create observables, as well as manipulate them. In particular, there are four scenarios to keep in mind, when it comes to creating an observable:

1. promise
2. counter
3. event
4. AJAX request

38.4.1 Promise

Assuming we were using Apollo client to retrieve data from our GraphQL requests, RxJS gives us the ability to transfer data we have retrieved from our backend into an observable:

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

// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe((data) => {
  console.log(data);
});
```

In the above we are using the RxJS `from` method to convert the data we have into an observable. This way, it can be accessed using subscribe. You might be wondering what may be the benefit of using an observable over JSON data and a promise in this scenario. Well, the benefit would be that if we do plan on mutating this data in the near future, then it will be beneficial to have data available as an observable. Primarily, when data is mutated, your front end component will register the change. In addition, being that other parts of our application are using observables(for instance, through the use of subject and next, which we will get to soon), being able to have them all follow the same cookie cutter logic, is beneficial to our application.

38.4.2 Counter

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

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

38.4.3 Event

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

const el = document.getElementById('my-element');

// Create an Observable that will publish mouse movements
const mouseMoves = fromEvent(el, 'mousemove');

// Subscribe to start listening for mouse-move events
const subscription = mouseMoves.subscribe((evt: MouseEvent) => {
  // Log coords of mouse movements
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

  // When the mouse is over the upper-left of the screen,
  // unsubscribe to stop listening for mouse movements
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
```

38.4.4 AJAX Request

```
import { ajax } from 'rxjs/ajax';

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

38.5 Operators

38.5.1 Sophisticated Manipulation

Once an observable has been created, RxJS provides operators to manipulate the data contained within an observable. The following is a great example. Let's say that we have an observable. In this observable, we have data for user settings. Specifically, there are settings for currency preferences that we would like. Within our official schema, it looks something like this:

```
{
  settings {
    currency: {
      ...
    }
  }
}
```

We would like to make sure that when we subscribe to our data store across the application, we pull in a specific subset of data. We can map our data within our `ngrx/store` data selector.

```
const getPostsCollection = createSelector(
  getAllPostEntities,
  getAllPostIds,
  (entities: any, ids: any) => ids.map(id => entities[id])
);
```

Listing 38.1: settings.selector.ts

Without having to specify what data that is every time, we can map our data to a specific data field. Here we are using the `ngrx/entity` library, as well as the `map` method to create a collection of entities. Very sophisticated, and allows us to pull in all the data we need within the observable. This keeps logic outside of individual component, and therefore re-usable.

38.5.2 Link Operators Together

RxJS also offers the ability to link operators together. For instance, let's say within our entity we want to filter out all numbers that are odd. Then we want to square that number:

```
1  import { filter, map } from 'rxjs/operators';
2
3  const nums = of(1, 2, 3, 4, 5);
4
5  // Create a function that accepts an Observable.
6  const squareOddVals = pipe(
7    filter((n: number) => n % 2 !== 0),
8    map(n => n * n)
9  );
10
11 // Create an Observable that will run the filter and map functions
12 const squareOdd = squareOddVals(nums);
13
14 // Subscribe to run the combined functions
15 squareOdd.subscribe(x => console.log(x));
```

You might be wondering why RxJS calls this a pipe?! Well if not familiar with already, the concept of a pipe originally comes form Unix. It is the ability to pass information from one process to another. For instance, if using a the terminal in Unix, something such as the following ps aux — grep root — pbcopy; would display all processes, grep specifically for those tied to root, and then copy that to the clipboard.

38.5.3 No subscribe, No describe

In case you didn't get that title, it's a unit testing joke. But for real, if you do not call a subscribe on your pipe, it will never be called. Consider the pipe as the function(which under the hood it is), and subscribe effectively calling the pipe(i.e. calling function). So, if pipe not being called, that might be why.

38.5.4 Common Operators

One of the biggest complaints that many have about RxJS, is that it's an overly bloated library. A while back, a man by the name of Andre Saltz who is very much so responsible for popularizing observables within Javascript, created a library called XStream, to focus just on the operators one needs to use on a day to day basis. Primarily, because some people were complaining of the vast amount of operators they happened to learn. (One can perhaps complain on the bloat that the RxJS library adds, however, RxJS introduced tree shaking in verson 5.5)

Ironically, what this did do, is start to create transparency around what would be considered a common operator. The RxJS then went ahead and started creating what would be considered common operators. The following are common operators listed by the Angular documentation:

Area	Operators
Creation	`from`,`fromEvent`, `of`
Combination	`combineLatest`, `concat`, `merge`, `startWith` , `withLatestFrom`, `zip`
Filtering	`debounceTime`, `distinctUntilChanged`, `filter`, `take`, `takeUntil`
Transformation	`bufferTime`, `concatMap`, `map`, `mergeMap`, `scan`, `switchMap`
Utility	`tap`
Multicasting	`share`

I like to call these operators the RxJS airplane.

We will get into these at a very high level at a later time within this book. It's important to have this table in mind, so that you know the RxJS operators to keep an eye out for.

38.6 Subjects

It's also a worth noting the use of subjects within RxJS, as they get used quite often. A subject, in addition to being an observable, is also an observer. This means that we can use `next` to add a value to an observable. Depending on type of subject, the relationship with next will change. Three subjects come up quite common, and are worth mentioning simply to keep them in mind.

Before we get to the different types of subjects, let's first lightly graze what an observer is(,in addition to the observable which we are already familiar with). An observer is simply a set of callback function attached to an observable:

```
const observer = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};
```

Based on the type of callback called, that is the value the observable will have. So a subject is this observer type, in addition to the observable type already familiar with. With this is mind:

```
// RxJS v6+
import { Subject } from 'rxjs';

const sub = new Subject();

sub.next(1);
sub.subscribe(console.log);
sub.next(2); // OUTPUT => 2
sub.subscribe(console.log);
sub.next(3); // OUTPUT => 3,3 (logged from both subscribers)
```

Listing 38.2: using next on our subject

In the above code, calling next will call all previous instantiated observables. Note the final next, which is logged from both subscribers, whereas the 2nd next, having only one subscriber instantiated, only logs one amount.

38.6.1 BehaviorSubject

`BehaviorSubject` As opposed to a subject, allows you to supply an initial value. It can be useful when creating something such as a re-usable component, and want to supply an initial value. In addition, unit testing, and want to circumvent the extra code need to use next, when mocking values.

```
// RxJS v6+
import { BehaviorSubject } from 'rxjs';

const subject = new BehaviorSubject(123);

// two new subscribers will get initial value => output: 123, 123
subject.subscribe(console.log);
subject.subscribe(console.log);

// two subscribers will get new value => output: 456, 456
subject.next(456);

// new subscriber will get latest value (456) => output: 456
subject.subscribe(console.log);

// all three subscribers will get new value => output: 789, 789, 789
subject.next(789);

// output: 123, 123, 456, 456, 456, 789, 789, 789
```

38.6.2 ReplaySubject

`ReplaySubject`, in addition to allowing an initial value, will also store the old values.

```
// RxJS v6+
import { ReplaySubject } from 'rxjs';

const sub = new ReplaySubject(3);

sub.next(1);
sub.next(2);
sub.subscribe(console.log); // OUTPUT => 1,2
sub.next(3); // OUTPUT => 3
sub.next(4); // OUTPUT => 4
sub.subscribe(console.log); // OUTPUT => 2,3,4 (log of last 3 values
    from new subscriber)
sub.next(5); // OUTPUT => 5,5 (log from both subscribers)
```

In the above code, we have given our `ReplaySubject` a buffer size of 3. Therefore, every next we have, before instantiating another subscribe, will introduce buffer size of 3. This can be useful for instance, when we are using something such as a timeline, or a table of contents, need to access to an observable, and want to keep track of all previous selections of user.

38.7 Naming Conventions for Observables

Hands down, my favorite naming convention in an Angular application is the use of a dollar sign `$` as an indicator of an observable. Observables started becoming popular before the use of Typescript. While Typescript/RxJS does have the type annotation of `<Observable>` to indicate an observable, I still approve of the trailing `$` to indicate an observable.

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html'
})
export class StopwatchComponent {
  stopwatchValue: number;
  stopwatchValue$: Observable<number>;

  start() {
    this.stopwatchValue$.subscribe(num =>
      this.stopwatchValue = num
    );
  }
}
```

Listing 38.3: Observable Naming Convention

38.7.1 Why Dollar sign was chosen as a convention

You will notice in the above code, for the Observable `stopwatchValue$`, we add a `$` to the end. This is to signify that this value is a stream. In many languages such as Java, there is a concept called a stream. While similar, there are some distinct difference such as:

1. Observables are asynchronous as opposed to Streams, which aren't.

2. Observables can be subscribed to multiple times.

3. Observables are push based, streams are pull based.

Nonetheless, being that streams existed before, and have this sort of naming convention, it bled it's way to it's cousin the observable.

38.7.2 Benefits of Naming Convention

The benefits of this naming convention are two-fold:

1. When scanning through code, and looking for observable values

2. If you want a non-observable value to store observable value, it can simply have no dollar value. This can then be used without a dollar sign, and makes for really transparent code.

That pretty much wraps the introduction to RxJS, as I would have liked to have been introduced to RxJS. I hope you like it, because I really wanted to help on this one.

39 Debugging Rxjs

One of the more difficult things to master when learning RxJS, is learning how to debug a stream. Within regular Javascript/Typescript, debugging is relatively easy. I find myself going to two differet methods when following this approach:

1. console.log

2. source when dev tool

39.1 Using Console.log

In a regular Typescript/Javascript setting a `console.log` is a relatively simple concept. We show(,or log out), whatever the current status is of our code at a given time. So for instance, let's say that we wanted to create a piece of code that takes the first and last name, and combines(proper software term is concatenates) them together. Something like this:

```
1 combinedName(firstName: string, lastName: string): string {
2   return firstName + lastName;
3 }
```

Listing 39.1: console in action

So the developer, due to the low level of the logic of this function, mistakenly forgets to add a white space in between. He decides to console out the function, before applying the logic directly to the application, to make sure it works as expected. So the developer does something like this(assuming we are talking about a method within a service):

```
1 console.log(this.combinedName('James', 'Harden));
```

The developer is expecting it to look like 'James Harden', but instead the developer finds out that it is 'JamesHarden'. The developer immediately realizes that there is a missing white space in between the two words, and edits the function, so that it works as expected:

```
1 combinedName(firstName: string, lastName: string): string {
2   return firstName + ' ' + lastName;
3 }
```

He then goes back, and refreshes the page where the `console.log` was,and finds out that it now looks like:

```
'James Harden'
```

In practice, something like this should ideally be tested using unit tests. However, many enterprise applications unfortunately are not built in this optimal fashion for various reasons.

39.2 Using Source Within Developer Tool

The only other tool that I usually have within my tool belt to dissect an error immediately is the developer tools within chrome. Any time that there is an error, there is a unique signature that your code will throw out. In a modern day application, this will happen even if it isn't unique to your application, due to the number of libraries/frameworks we use. However, it can still be incredibly useful for hard to find bugs. This article will not discuss this technique in depth. However, it will be the top two tools in your debugging tool belt. Feel free to checkout this video on egghead.io for more.

39.3 Debugging within RxJS - The Dilemma

The immediate issue with debugging RxJS from a traditional setting, is that it's only the final result that can be debugged. So you might have a stream of numerous chained events:

```
this.companies$ = searchBy$.pipe(
  debounceTime(300),
  distinctUntilChanged(),
  startWith(''),
  switchMap((criteria:string) => {
    const request$ = this.companyService.searchCompanies(criteria);
    return !criteria.length ? of([]) : request$
  })
);
```

In the above, only in the switchMap, or in a subscribe will we be able to tap into the result that we are looking for. For instance, let's go through the potential bugs that might happen with the above stream:

1. The result might not appear because it is within the `debounceTime`
2. It might not have changed from the previous result i.e. `distinctUntilChanged` and therefore is not triggered.

3. The `companyService.searchCompanies` request might be erroring out, or there might be some internal logic to the `companyService.searchCompanies` request making something error out.

So of course we need a similar way to console out our results along the line of the stream, to figure out where our error is happening.

39.4 Using Tap

`tap` is RxJS's way of taking the current value and outputting it. It is commonly used in real life code, by taking the value returned by a service, and emitting it. However, tap can also be used a debugger. For instance, in the above code, let's say we aren't getting the result we wanted. We can do the following:

```
this.companies$ = searchBy$.pipe(
  debounceTime(300),
  tap(result => console.log(result)),
  distinctUntilChanged(),
  startWith(''),
  switchMap((criteria:string) => {
    const request$ = this.companyService.searchCompanies(criteria);
    return !criteria.length ? of([]) : request$
  })
);
```

In the above `tap`, a result is coming back so we know that it is not erroring out before `debounceTime`. Ok, so let's move the tap one more down:

```
this.companies$ = searchBy$.pipe(
  debounceTime(300),
  distinctUntilChanged(),
  tap(result => console.log(result)),
  startWith(''),
  switchMap((criteria:string) => {
    const request$ = this.companyService.searchCompanies(criteria);
    return !criteria.length ? of([]) : request$
  })
);
```

Here we are no longer receiving the console.log that we want. This is telling us that the reason we might not be recieving our result, is that it is not being registered as anything distinct.

I've seen one other article in particular claim use of a debug utility functionality that is only turned on in development mode. I personally am against such debugging patterns, and feel it is an anti-pattern. I strongly believe that unit tests should be in place to prevent errors. Debugging should happen in one off scenarios, and then reinforced with

unit tests and integration tests to make sure error does not happen again.

40 Cold v Hot Observables

One of the foundational concepts of RxJS is understanding what the difference between cold and hot observables is. At it's core this difference is:

> "When the data is created inside the Observable itself, it is a cold Observable. When the data is created outside the Observable, it is a hot Observable."

There is a lot to unpack here, so let's dissect.

40.1 Cold Observables

To re-iterate, an observable is called "cold", when the data is produced inside of the observable. An observable is only instantiated, once the subscribe is called. So, we can have something like this:

```
1 import { Subject } from 'rxjs';
2
3 const randomVal$ = new Subject().next(Math.random());}
```

Listing 40.1: observable without subscribe

That will actually create an observable, but it will not be called until we add a `subscribe` to the code.

```
1 import { Subject } from 'rxjs';
2
3 const randomVals$ = new Subject().next(Math.random());
4
5 // Subscribe to run the combined functions
6 randomVals$.subscribe(x => console.log(x));
```

Listing 40.2: observable with subscribe

For that reason, it's possible to have it subscribe mutliple times to a singular observable, yet get different results each time. If in the above code, we created two subscribes, so that our code now looks like this:

```
1 import { Subject } from 'rxjs';
2
3 const randomVals$ = new Subject().next(Math.random());
```

```
4
5  // Subscribe to run the combined functions
6  randomVals$.subscribe(x => console.log(x));
7  // 0.8771441274333971 (random number)
8  randomVals$.subscribe(x => console.log(x));
9  // 0.09728173083079916 (random number)
```

Listing 40.3: observable with two subscribes

As we can see in the comments above, each subscribe will give us a new value.

40.1.1 Maintaining Same Value for Both Subscribers - Venture into the Land of Hot

However, let's say that we wanted to maintain the same random value for both subscribers. What we would do, is create a constant outside of the subject logic, so that it remains the same value for both.

```
1  import { Subject } from 'rxjs';
2
3  const random = Math.random();
4  const randomVal$ = new Subject().next(random);
5
6  // Subscribe to run the combined functions
7  randomVals$.subscribe(x => console.log(x));
8  // 0.8771441274333971 (random number)
9  randomVals$.subscribe(x => console.log(x));
10 // 0.8771441274333971 (random number)
```

Listing 40.4: Turning Cold into Hot

Here, we have moved the data producer(i.e. `Math.random()`) out of observable. So, to go back to our definition of a hot, vs cold observable. Being that this data was produced outside of the observable, our observable now just went from being a cold observable to a hot observable. However, as time goes on, you will have to apply this (hot v cold) logic to every scenario to truly understand where the data is being produced.

40.2 Hot Observables

So, while we have produced a good example for what a cold vs hot observable is, we have not produced an enterprise example, that really drives it home. The truth is, that actually `@ngrx/store` is based on hot observables. In particular when using `@ngrx/effects`.

```
1  @Effect() loadPosts$ = this.dataPersistence.fetch(
2    PostsActionTypes.LoadPosts,
3    {
4      run: (action: LoadPosts, state: PostsPartialState) => {
```

```
      return this.postsService.getPosts().pipe(
        map((posts: Post[]) => new PostsLoaded(posts))
      );
    },
    onError: (action: LoadPosts, error) => {
      console.error('Error', error);
      return new PostsLoadError(error);
    }
  }
);
```

Listing 40.5: posts.effects.ts

As we can see, we are passing in an action that already exists. In fact, we are using nrwl's utility function `dataPersistence` to make sure our data persists. If we dig deep, we will see that it extends this core functionality of `ngrx/effects`:

```
/**
* @whatItDoes Provides convenience methods for implementing common
    operations of persisting data.
*/
export declare class DataPersistence<T> {
  store: Store<T>;
  actions: Actions;
  constructor(store: Store<T>, actions: Actions);
```

Listing 40.6: data-persistence.d.ts file from nrwl library

```
export declare class Actions<V = Action> extends Observable<V> {
  constructor(source?: Observable<V>);
  lift<R>(operator: Operator<V, R>): Observable<R>;
  //..
}
```

Listing 40.7: actions.d.ts file from ngrx/effects library

As, we can see, the actions, are extending the Observable type annotation. (This is immediately apparent if using the ngrx/effects 's createEffect method. However, on principal, the Angular: The Full Gamut series uses enterprise examples, even when trying to explain core concepts

We are accepting data from an action, turn all actions within our app as an observable, thereby making the data produced outside of our effects. This is a classic scenario that happens time and time again, that is in truth a hot observable. Knowing this truth is very valuable, especially when it comes to unit testing your code.

40.3 Why Make an Observable Hot?

To re-iterate, an Observable by default is cold. So, why would one want to make an observable hot? This has value in two particular situations:

1. Have multiple subscribers get the same data

2. If you would have to create a new value time and time again within your observables subscribe(websocket, action etc.).

3. Separation of concern. Within @ngrx/store we have files, and pieces of our state management that do different things. By creating a hot observable, it allows us to pass through the data, use a hot observable, and keep with separation of concerns.

41 RxJS Common Creation Operators

Just to re-iterate, one of the more novel parts of this book, is that when we introduce a new concept we tie that into the actual enterprise scenario, wherein this is repeated. Allowing for one to use code directly in one'e application, without having to think about it too much how it is used.

The common RXJS creation operators are:

1. from
2. fromEvent
3. of

41.1 from

A common scenario when using from, is that you would like to return an http request, and turn that into an observable. Razroo has a very opinionated architecture, and we follow through with that architecture throughout the book. We realize that every scenario may not be able to follow the ideal architecture. However, let's assume that you are using three very integral libraries:

1. GraphQL
2. Apollo Client
3. ngrx/store

`from` will be used in a scenario wherein you are trying to convert an apollo query into an observable,

```
1 getPosts(): Observable<Post[]> {
2   const posts$ = this.apollo.query({ query: GetPosts });
3
4   return from(posts$).pipe(pluck('data', 'posts'));
5 }
```

Listing 41.1: posts.service.ts

so that it can be passed into your ngrx/store:

```
@Effect() loadPosts$ = this.dataPersistence.fetch(
  PostsActionTypes.LoadPosts,
  {
    run: (action: LoadPosts, state: PostsPartialState) => {
      return this.postsService.getPosts().pipe(
        map((posts: Post[]) => new PostsLoaded(posts))
      );
    },
    onError: (action: LoadPosts, error) => {
      console.error('Error', error);
      return new PostsLoadError(error);
    }
  }
);
```

Listing 41.2: posts.effects.ts

41.2 fromEvent

Whole `fromEvent` is a common operator in RxJS, I would argue that within an Angular setting, it is not a common operator. There is truly no scenario within Angular wherein `fromEvent` makes sense. Instead use the core Angular even handlers within your template:

```
<button (click)="submitForm()">
```

Listing 41.3: user-form.input.ts

```

submitForm(userData: User) {
  this.userFormFacade.submitForm(userData);
}
```

Listing 41.4: user-form.component.ts

41.3 of

`of` while similar in purpose to of, will convert a value to an observable. However, the main difference between the two, can be seen here in this snipper of code:

```
// the subscribe here, will emit all of the data at once
// [1, 2, 3]
Observable.of([1, 2, 3]).subscribe(x => console.log(x));

// the subscribe here, will iterate through data one at a time
Observable.from([1, 2, 3]).subscribe(x => console.log(x));
// 1
// 2
// 3
```

`from` for the most part is preferabble, as data will usually be contained in an array, and `from` sidesteps the need of tapping into the array, doing something like this:

```
console.log(data[0]);
```

From my personal experience, `of` becomes invaluable when it comes to mocking data within your unit tests. For instance, doing something such as the following:

```
const postMock = {
  id: '123',
  title: 'test title',
  featureImage: 'sample feature image',
  created_at: '2019-08-05T07:49:11.405Z'
}

const postMock$ = of(postMock);
```

This is the most efficient solution is this scenario for mocking data, and comes up quite frequently within your app, and something to be aware of.

42 Combination

42.1 combineLatest

Combine latest allows us to combine multiple observables into one, and have the subscribe emit when either of them are called. A common enterprise example, is let's say that we have a form. The form gives us a search bar, but we would like to have our search bar, search against a drop down as well.

```
export class AppComponent implements OnInit, OnDestroy {
  private destroyed$ = new Subject<void>();

  get searchText() {
    return this.form.get('searchText');
  }

  get searchFor() {
    return this.form.get('searchFor');
  }

  form: FormGroup;

  constructor(formBuilder: FormBuilder) {
    this.form = formBuilder.group({
      searchText: null,
      searchFor: null,
    });
  }

  ngOnInit() {
    const searchText$ = this.searchText.valueChanges;
    const searchFor$ = this.searchFor.valueChanges;

    // combineLatest emits when any of the source observables
    // emits (provided that they have all emitted once to
    // begin with). that's why the console.logs start appearing
    // after values are selected for both form controls
    combineLatest(searchText$, searchFor$)
      .pipe(takeUntil(this.destroyed$))
      .subscribe(([searchText, searchFor]) => {
        console.log('searchText', searchText);
        console.log('searchFor', searchFor);

        // make a call to an API whenever either observable
        // emits
```

```
37          // this.apiService.loadData(searchText, searchFor)
38        })
39    }
40
41    ngOnDestroy() {
42      this.destroyed$.next();
43      this.destroyed$.complete();
44    }
45  }
```

Listing 42.1: search-form.component.ts

42.2 concat

One very useful use of concat within an enterprise setting, is that if you have all of the data you need available. However, you would like to make sure that they emit in proper order of each other. Using the following:

```
1  import { concat } from 'rxjs/operators';
2
3  concat(printLog1, printLog2, printLog3);
4  // printLog1 emits, then printLog2, and then printLog3
```

Listing 42.2: concat-example.ts

In a realistic scenario, this will come up in a situation such as we would like to

```
1  export class AppComponent {
2    searchStream$ = new BehaviorSubject('');
3
4    constructor(private productsService: ProductsService) {}
5
6    obs$ = this.searchStream$.pipe(
7      debounceTime(200),
8      distinctUntilChanged(),
9      switchMap((query) =>
10       concat(
11         of({ type: 'start'}),
12         this.productsService.getByFilter(query).pipe(map(value => ({
               type: 'finish', value })))
13       ))
14   );
15 }
```

Listing 42.3: search-bar.component.ts

42.3 merge

Merge allows us to "merge" multiple observables into one singular observable. A great re-occuring real world example, is to group multiple events into single observable. For instance, if we are trying to create a music player:

```
1  const start$ = fromEvent(this.startButton.nativeElement, 'click').pipe(
       mapTo(true));
2  const pause$ = fromEvent(this.pauseButton.nativeElement, 'click').pipe(
       mapTo(false));
3  const reset$ = fromEvent(this.resetButton.nativeElement, 'click').pipe(
       mapTo(null));
4
5  this.playerButtons$ = merge(start$, pause$, reset$, stateChange$).pipe(
6    switchMap(val => {
7      if(val === null) // reset
8      else if(val) // start
9      else // stop
10   }
11 );
```

Listing 42.4: player-buttons.component.ts

Here, by merging all of our events within a single observable, it tidies up our code base, and allows us to deal with all of our events in a singular fashion.

42.4 startWith

There is a really old tweet by Andre Saltz, that is no longer available for some reason. Regardless, the tweet goes something along these lines:

> "You rarely want `combineLatest()`, unless mixed with `startWith()`"

`startWith`, as the name implies, is a way to tell your observable to start with a paticular value. With the advent of `behaviorSubject` which allows you to supply the initial value, or the alternative scenario of data being retrieved from the backend, with the use of Angular's `async` pipe, wherein the initial value is superfluous.

However, there is one scenario wherein the `startWith` pipe does get used alot within an Angular setting. That is in conjuction with `combineLatest`. However, even in this scenario, I would like to argue that `behaviorSubject` get's used more. [1]

[1] To see why not to use startWith and go with behaviorSubject, check the addendum

42.5 withLatestFrom

`withLatestFrom` will combine two observables with each other. However, only when the value changes for the primary observable, will the subscribe be triggered. If the secondary observable value changes, the subscribe will not be triggered.

Let's use a common example that will occur within the enterprise architecture set up by Razroo. Let's say that you are using `ngrx/store`, and that you would like to determine whether, or not a blog post exists already within our store. If the blog post already exists within our store, then do make another HTTP request, as we have the data we need already.

```
pipe(
  ofType(LoadPost),
  withLatestFrom(
    this.store.pipe(select(postsQuery.getCurrentPost))
  )(this.store),
  filter([post] => !post),
  mergeMap([a] => s.getPost(a.id))
)
```

Listing 42.5: data-access-post.ts

In the, we are looking to see if current post already exists within our blog store. If it does, then we do not make the request. If it does not, then we go ahead and make the request once again.

42.6 zip

```
const yin   = Rx.Observable.of('peanut butter', 'wine','rainbows');
const yang  = Rx.Observable.of('jelly', 'cheese', 'unicorns');

const combo = Rx.Observable.zip(yin, yang);

combo.subscribe( arr => console.log(arr));
// peanut butter, jelly
// wine, cheese
// rainbows, unicorns
```

Listing 42.6: yin-yang.component.ts

42.7 Expontential Backoff

`zip` can be particularly useful using something called exponential backoff. Exponential backoff is a technique wherein you retry an API after failure. However, instead of retrying let's say every 5 seconds, instead we increase the values exponentially after each consecutive

failure, with a maximum amount of retries. This can potentially be really complex, but zip makes something like this alot easier:

```
import { pipe, range, timer, zip } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { retryWhen, map, mergeMap } from 'rxjs/operators';

function backoff(maxTries, ms) {
 return pipe(
   retryWhen(attempts => zip(range(1, maxTries), attempts)
     .pipe(
       map(([i]) => i * i),
       mergeMap(i =>  timer(i * ms))
     )
   )
 );
}

ajax('/api/endpoint')
  .pipe(backoff(3, 250))
  .subscribe(data => handleData(data));

function handleData(data) {
  // ...
}
```

In my opinion, the above code, is a bit like ninja code, so let's go ahead and unpack everything that is being done here.

1. First we pass maximum amount of tires, and multiplier to increase exponentially.

2. In our rxjs pipe, we first use `retryWhen`. This is an `rxjs` operator that takes in an observable, and then retries that for x amounts of time.

3. within our `retryWhen` we pass a zip, so that we create x x amount of observable streams equal to the amount of `maxTries`

4. We then pass the `retryWhen` index, and increase that exponentially.

5. We then use the new value, and pipe that over to our timer, which is passed as the new `retryWhen` amount.

43 Filtering

43.1 debounceTime

`debounceTime` means that if during the buffer time passed to `debounceTime` method, another event is fired, it will cancel previous stream. This is a very common use case that is used within search. It can also be used it regular event handlers, to prevent from clicking a submit button multiple times.(However, Angular's internal event handler is usually pretty good at this.)

> It's important to note, that adding a debounce method, while will greatly help backend with amount of requests needed to be made, will also help with the users CPU.

```
1 @Effect() searchImage$ = this.actions$.pipe(
2   ofType(fromBlog.SEARCH_BLOGS),
3   map((action: fromBlogActions.SearchBlog) => action.query),
4   debounceTime(300),
5   switchMap(query: string) => this.blogService.getBlogBySearching(query)
      )
```

Listing 43.1: blog.effects.ts

The above is a common re-occuring pattern within the `ngrx/effects` library. We will be debouncing our search, so that if a user searches again within a given time, then it will cancel the previous observable.

43.2 distinctUntilChanged

Adding a `distinctUntilChanged()` operator to your pipe, will make it so that an observable is not changed if current value emitted, is the same as the prior value. A great use case for this, is within the effect we already used within our app.

```
1   @Effect() searchImage$ = this.actions$.pipe(
2     ofType(fromBlog.SEARCH_BLOGS),
3     map((action: fromBlogActions.SearchBlog) => action.query),
4     debounceTime(300),
5     distinctUntilChanged(),
```

```
6     switchMap(query: string) => this.blogService.getBlogBySearching(
          query))
```

Listing 43.2: blog.effects.ts

The reason we are adding this, is that there is one use case we have not handled yet within our stream. Let's say that a user types part of their name `char`, proceeds to write `charlee`, and then shortly thereafter deletes `lee`, because they decide for this site they would like keep with their nickname `char`. The letters l, e, and e were typed within the 500 `debounceTime` time limit. However, the deletion of the final letter of `l` after the deletion of ee did happen after the `debounceTime`. Adding `distinctUntilChanged` into the mix, makes it so that in this one particular use case, the event is not fired again.

43.3 filter

`filter` in RxJS will very similarly to the method in Javascript, only return those values which pass the condition. We can re-visit our code snippet from the `withLatestFrom` example.

```
1 pipe(
2   ofType(LoadPost),
3   withLatestFrom(
4     this.store.pipe(select(postsQuery.getCurrentPost))
5   )(this.store),
6   filter([post] => !post),
7   mergeMap([a] => s.getPost(a.id))
8 )
```

Listing 43.3: data-access-post.ts

Here with are taking the currentPost. If it already exists, the filter returns null, which makes the observable cancel itself, as the internal engine of RxJS works.

43.4 take

`take` will specify how many times a value should be used. A very common example within an enterprise setting, is to use `take(1)` to unsubscribe from an observable.

```
1 // RxJS v6+
2 import { fromEvent } from 'rxjs';
3 import { take, tap } from 'rxjs/operators';
4
5 const oneClickEvent = fromEvent(document, 'click').pipe(
6   take(1),
7   tap(v => `'${v.screenX}:${v.screenY}`')
8 );
9
```

```
10 const subscribe = oneClickEvent.subscribe(clickLocation => {
11   this.analyticsFacade.locationAnalytics(clickLocation);
12 });
```

In the above code, we have a specific internal analytics service set up. We want to see the first location that the user clicks on. After that point in time, we no longer want to collect the user data. We have these analytics set up because we want to know on average what is the most attention grabbing part of our site.

43.5 takeUntil

`takeUntil` operator will unsubscribe from the observable once it has satisfied a certain condition. `takeUntil` in my opinion is the cleanest way of unsubscribing from an observable.

```
1  private unsubscribe$ = new Subject();
2
3  this.postFacade.blogPosts$.pipe(
4    takeUntil(this.unsubscribe$);
5  );
6
7  ngOnDestroy(): void {
8    this.unsubscribe$.next();
9    this.unsubscribe$.complete();
10 };
```

Listing 43.4: blog.component.ts

44 Transformation

Transforming in RxJS is the process of modifying a value. It is quite a common occurence within RxJS.

44.1 bufferTime

`bufferTime` will collect a value until the provided time elapses. Once time has elapsed it will emit values collected during that time as an array. (It will not unsubscribe from the observable). A good example of when we would want to use bufferTime, is if we want to give the user notifications within their application. However, we don't want to flood them with notifications. In addition, the updates within our application are quite frequent, and are updated quite frequently. Think something along the lines of a stock market application.

```
this.notificationService.updates$.pipe(
  bufferTime(5000)
).subscribe(batch => {
  this.notifications = notifications;
});
```

To be honest, within an enterprise Angular application, I haven't used `bufferTime` very often.

44.2 RxJs Higher Order Mapping

Before we dive into these three values:

1. `concatMap`
2. `mergeMap`
3. `switchMap`

it's important to note that they are all similar in one regard. It will subscribe to an inner observable. This means that it allows for access to the value, and will create an inner

observable. This inner observable can then be modified. Why create an inner observable? Well this get's into something called higher order observables. Think about the way classic promises work. Let's say we wanted to modify the data nested twice within our promise:

```
this.postService.getPosts().then(data => {
  const blogIds = data.map(data => data.id);
  this.analyticsService.analytics(blogIds).then(analyticsData => {
    this.analyticsData = analyticsData;;
  });
});
```

In the above code, we have one promise inside of another promise. RxJS allows us to do all of these within the same stream. It allows for the observable stream to be more manageable and concise(i.e. avoiding nested subscribes). These are what would be called higher order observables.

44.3 concatMap

`concatMap` will map values to the inner observable, subscribe and emit in order. Emphasis is emit in order, that is what `concatMap` specializes in. A great way to visualize this, is let's say we wanted to create a cascading effect within our application.

```
getItems(ids: number[]): Observable<Item> {
  return from(ids).pipe(
     concatMap(id => <Observable<Item>> this.httpClient.get(`item/${id
         }`))
  );
}
```

Listing 44.1: data-table.component.ts

In the above, the previous observable will emit first, causing a visual waterfall effect within our app.

44.4 mergeMap

`mergeMap` (similar to `switchMap` and `concatMap`), will create an inner observable. The main difference, is that `mergeMap`, will merge all observables into one(, as opposed to `switchMap`, which will cancel all prior observables).

```
  @Effect()
  loadAllBlogPosts$: Observable<any> = this.actions$.pipe(
    ofType(PokemonActions.loadPokemon),
    mergeMap(() =>
      this.postsService.getAll().pipe(
        map(posts => PokemonActions.loadPokemonSuccess({ posts })),
```

```
7           catchError(message => of(PokemonActions.loadPostsFailed({
                message })))
8         )
9       )
10    );
```

Listing 44.2: mergeMap example

You might notice that we are using mergeMap for what might be otherwise thought of using a switchMap for. The reason I like using mergeMap is that switchMap doesn't actually do anything in this scenario. Even if we were to cancel the inner observable the action and http request has still made it's way through. I feel like using mergeMap makes this scenario more transparent and is therefore the right thing to use in this scenario.

44.5 switchMap

`switchMap` (similar to `concatMap` and `mergeMap`), will create an inner observable. The main difference is that it will complete the previous inner observable, so that only the latest observable is re-used.

```
1  @Effect()
2  findAddresses: Observable<any> = this.actions.pipe(
3    ofType(LocationActionTypes.FindAddresses),
4    map(action => action.partialAddress),
5    debounceTime(400),
6    distinctUntilChanged(),
7    switchMap(partialAddress => this.backend
8      .findAddresses(partialAddress)
9      .pipe(
10       map(results => new FindAddressesFulfilled(results)),
11       catchError(error => of(new FindAddressesRejected(error)))
12     )
13   )
14 );
```

Listing 44.3: search-bar.component.ts

In the example above, we have set up our effect to handle search. It has a

1. `debounceTime` so that if user types multiple times within a 400 milliseconds, it will only trigger once

2. `distinctUntilChanged` to handle the use case wherein user deletes letters after typing, but returns to same word after deleting

3. ...and then the magic! `switchMap` is used within our app, because once a new search is made, prior observables are no longer needed and can be removed.

It's interesting. While this has value, an alternative option could have been used, i.e. `concatMap`. Due to the finicky nature of maps, and complexity behind it, maps can also be considered as a way of documentation. In my opinion, on documentation alone, it is useful to use the most appropriate higher order observable. However, thinking about it again now, there are always use cases that get overlooked. So, choosing the most appropriate observable, on top of unit tests, is the most bonafide way to sidestep potential bugs.

When choosing a flattening operator for an effect/epic, if subsequent actions of the same type will render pending results stale, switchMap is unequivocally the best choice.

44.6 scan

`scan` actually works exactly like `reduce` does in regular Javascript for arrays, but for observables. So, why did the RxJS team call it `scan` instead of `reduce`? Well, it turns out there is one little difference. `reduce` actually is an operator in RxJS as well. However, `scan` will emit the value for every iteration, whereas `reduce` will emit only the final value.

`scan` therefore as a combinator has more use than reduce does, because it allows us to tap into the history of our state. (In fact, while we won't go into that here, we can create a really low level state management using `scan`)

So in an enterprise setting, where we already have use of `ngrx/store`, why would scan be considered as a common operator? The truth is, that within an enterprise Angular application, using `ngrx/store` there are some really one of cases that use it. It is not an operator that I would keep in mind, unless you are trying to introduce state to legacy Angular application, that does not have state.

44.7 map

`map` will apply a projection to each value in source. For instance:

```
1  // RxJS v6+
2  import { from } from 'rxjs';
3  import { map } from 'rxjs/operators';
4
5  //emit (1,2,3,4,5)
6  const source = from([1, 2, 3, 4, 5]);
7  //add 10 to each value
8  const example = source.pipe(map(val => val + 10));
9  //output: 11,12,13,14,15
10 const subscribe = example.subscribe(val => console.log(val));
```

Listing 44.4: rxjs map example

A common occurrence within actual applications is to use map within a `switchMap`, when using an `ngrx/effects`. That way, it returns the `map` to the `switchMap`, and kills any other observables from happening.

```
@Effect()
loadAllBlogPosts$: Observable<any> = this.actions$.pipe(
  ofType(PokemonActions.loadPokemon),
  mergeMap(() =>
    this.postsService.getAll().pipe(
      map(posts => PokemonActions.loadPokemonSuccess({ posts })),
      catchError(message => of(PokemonActions.loadPostsFailed({ message
          })))
    )
  )
);
```

Listing 44.5: map example

Here we are mapping and returning an action observable to our effect. This allows us to hook in a backend service to our general state ecosystem.

45 Utility

Utility functions in RxJS as are utility methods in programming, are often re-used operators, which are helpful for accomplushing routine RxJS tasks. However, the only common used utility on a day to day is `tap`.

45.1 tap

`tap` can be used to perform an action, or a side effect. A great example, and one used in an enterprise setting, is to navigate user to a particular place based on particular action. For instance, let's say the user log's out.

```
@Effect({ dispatch: false })
logOut = this.actions.pipe(
  ofType(LoginTypes.LogOut),
  tap([payload, username] => {
    this.router.navigate(['/']);
  })
)
```

Listing 45.1: login.effects.ts

In the above, we are tapping into the action that specifies user has logged out. In addition to our reducer logic wiping clean all the data that was in the user's store, we are also navigating the user to the default rout. This a perfect scenario for using `tap`, as we want to create a side effect of navigating to a route. However, we do want to return an observable/action (i.e. do not want to use `map`).

46 Multicasting

I've seend Netanel Basel mention this, and he is most definitely right. Of the entire RxJS bunch, multicase operators are the most complicated topic to understand. That is primarily because it requires an understanding of the fundementals of RxJS. This book contains a fantastic chapter on the difference between cold and hot observables. If unfamiliar with, feel free to reference first.

46.1 share

Share will create a new observable, that shares the observable that is was called on. A great example of this, is:

```
// RxJS v6+
import { timer } from 'rxjs';
import { tap, mapTo, share } from 'rxjs/operators';

//emit value in 1s
const source = timer(1000);
//log side effect, emit result
const example = source.pipe(
  tap(() => console.log('***SIDE EFFECT***')),
  mapTo('***RESULT***')
);

/*
  ***NOT SHARED, SIDE EFFECT WILL BE EXECUTED TWICE***
  output:
  "***SIDE EFFECT***"
  "***RESULT***"
  "***SIDE EFFECT***"
  "***RESULT***"
*/
const subscribe = example.subscribe(val => console.log(val));
const subscribeTwo = example.subscribe(val => console.log(val));

//share observable among subscribers
const sharedExample = example.pipe(share());
/*
  ***SHARED, SIDE EFFECT EXECUTED ONCE***
  output:
  "***SIDE EFFECT***"
  "***RESULT***"
```

```
31    "***RESULT***"
32  */
33  const subscribeThree = sharedExample.subscribe(val => console.log(val));
34  const subscribeFour = sharedExample.subscribe(val => console.log(val));
```

Within an Angular setting, this method really isn't used. The Angular framework will internally subscribe and unsubscribe using the async pipe. However, I personally haven't noticed any performance issues as a result. Nonetheless something to be aware of, so when the possibility of using `share` arises, you have something to work with.

47 RxJS Pitfalls

RxJS is a powerful library to use. With that power, however, comes a lot of complexity. With that complexity comes a lot of ways to get into trouble. In this article, We will go through the most common pitfalls to look out for when using RxJS. In addition, tips on how to avoid them, as well as linking to external resources when I can.

47.1 Mishandling Subscriptions

Alain Chautard calls this âĂIJSubscribing too earlyâĂİ within his RxJS pitfalls article. Essentially, this arises when youâĂŹre writing a function whose asynchronous behavior is of interest to a piece of calling code. The problem arises when, within the implementing function, you subscribe and handle side effects within an Observable that would have been of interest to the caller.

In his article, Chautard shows us a LoginService class that has a `login()` method which subscribes to the Observable returned by `put()`. It looks something like this:

```
1  /**
2  * As seen within https://blog.angulartraining.com/5-rxjs-angular-
       pitfalls-to-be-aware-of-160adfd402d8
3  */
4  @Injectable()
5  export class LoginService {
6    constructor(private http: HttpClient) {}
7
8    login(username: string, password: string) {
9      this.http
10       .put('http://localhost:8000/login', {username, password})
11       .subscribe(data => {
12           this.currentUser = username;
13           this.authToken = data['token'];
14       });
15   }
16 }
```

Notice how any caller using `login()` would have no way of knowing when the login was successfully completed. Furthermore, all error handling would have to be handled by the LoginService code, which leads to a bad separation of concerns. This happens because the

code subscribes to the observable at the call site, and does not give the client any way of being notified of changes.

One thing we could do to fix this is return the Observable outright, passing all data along to the client.

```
@Injectable()
export class LoginService {
  constructor(private http: HttpClient) {}

  login(username: string, password: string): Observable<{currentUser:
      string, authToken: string}> {
    this.http
      .put('http://localhost:8000/login', {username, password})
      .pipe(
        map(data => ({
          currentUser: username,
          authToken: data['token'],
        }))
      );
  }
}
```

Notice here that, rather than subscribing to the returned Observable from `put()`,we use the map operator to return both the username and the auth token to the caller, rather than having to store anything ourselves.

While this works, itâĂŹs often the case that you want the service itself to store the returned data. For example, you may want to access loginService.currentUser in multiple different components or contexts. Therefore, it makes sense to store that information within the login service itself. In the original article, Chautard uses tap() to save information about the user. This is definitely a viable strategy, however it has the implication that any component code would have to rely on AngularâĂŹs change detection mechanisms to determine any changes. We can get a boost in performance by taking AngularâĂŹs change detection out of the equation.

It turns out that we actually can subscribe inside of this method, so long as we allow our client code to be notified of changes in auth state. We can achieve this by leveraging ReplaySubjects to store `currentUser` and `authToken` in an observable that clients could subscribe to in order to get login info, as well as providing an additional subject for error handling.

```
@Injectable()
export class LoginService {
  private currentUserSubject$ = new ReplaySubject<string>(1);
  readonly currentUser$ = this.currentUserSubject$.asObservable();

  private authTokenSubject$ = new ReplaySubject<string>(1);
  readonly authToken$ = this.authTokenSubject$.asObservable();

  private authErrorSubject$ = new ReplaySubject<any>(1);
  readonly authError$ = this.authErrorSubject$.asObservable();

```

```
12    constructor(private http: HttpClient) {}
13
14    login(username: string, password: string) {
15      this.http
16        .put('http://localhost:8000/login', {username, password})
17        .subscribe({
18          next(data) {
19            this.currentUserSubject$.next(username);
20            this.authTokenSubject$.next(data['authToken']);
21          },
22          error(err) {
23            this.authError$.next(err);
24          }
25        });
26    }
27  }
```

This allows our components to use the async$ pipe to subscribe to changes to the logged in user. Components therefore donâĂŹt have to rely on AngularâĂŹs built-in change detection, and can use push-based change detection for optimal performance.

A few additional things to notice in this example:

1. We still return the observable, so that the client has direct access to the information, as well as the ability to handle any errors from the observable should they arise.

2. WeâĂŹve passed in an argument of 1 to our ReplaySubjects. This argument is the âĂIJbufferâĂİ argument, which tells the subject only to retain the latest value it was given.

3. WeâĂŹve kept our ReplaySubjects private here, and exposed them as readonly properties using asObservable(). WeâĂŹll come back to that later in this article.

47.1.1 What to watch out for

Subscribing to an observable whose asynchronous outcome is of interest to calling code, without allowing client code to observe the results of the outcome.

47.1.2 What to do instead

Ensure the observable is returned, or make use of subjects and update their state as part of a subscription to an observable.

47.2 Forgetting to Unsubscribe

Have you ever seen code like this?

```
@Component({...})
export class MyComponent implements OnInit {
  constructor(
    private route: ActivatedRoute,
    private itemService: ItemService
  ) {}

  ngOnInit() {
    this.route.params.subscribe(params => {
      this.itemService.loadItemDetails(params['itemId']);
    });
  }
}
```

At first glance, this looks innocent enough, but thereâĂŹs a problem: within `ngOnInit()` we subscribe to route params, but we never unsubscribe. This is problematic because even when the component is no longer being used, the fact that the activated route is a service that exists outside the scope of the component causes the component to be retained in memory! This causes memory leaks and can easily degrade performance if left unchecked. This gets even worse as your component grows in complexity and the number of subscriptions increase.

Instead, you should always be sure to unsubscribe to any subscriptions you manually subscribe to. A good rule of thumb is: subscribe within `ngOnInit()`, unsubscribe within `ngOnDestroy()`.

HereâĂŹs how we could change the above code in order to handle this:

```
@Component({...})
export class MyComponent implements OnInit, OnDestroy {
  private routeSubscription: Subscription;

  constructor(
    private route: ActivatedRoute,
    private itemService: ItemService
  ) {}

  ngOnInit() {
    this.routeSubscription = this.route.params.subscribe(params => {
      this.itemService.loadItemDetails(params['itemId']);
    });
  }

  ngOnDestroy() {
    this.routeSubscription.unsubscribe();
  }
}
```

We can also write a unit test to verify that all subscriptions are cleaned up.

```
describe('MyComponent', () => {
  let fixture: ComponentFixture<MyComponent>;
  let itemServiceSpy: jasmine.SpyObj<ItemService>;
  let mockActivatedRoute: MockActivatedRoute;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [MyComponent],
      providers: [
        {provide: ActivatedRoute, useClass: MockActivatedRoute},
        {provide: ItemService, useValue: jasmine.createSpyObj('
            itemService', ['loadItemDetails'])}
      ]
    });

    itemServiceSpy = TestBed.get(ItemService);
    mockActivatedRoute = TestBed.get(ActivatedRoute);
    fixture = TestBed.createComponent(MyComponent);
    fixture.detectChanges();
  });

  // Verify the core behavior works
  it('calls loadItemDetails on param change', () => {
    mockActivatedRoute.paramsSubject$.next({itemId: 'itemIdParam'});
    expect(itemServiceSpy.loadItemDetails).toHaveBeenCalledWith('
        itemIdParam');
  });

  // Verify unsubscribe
  it('stops listening to param changes when destroyed', () => {
    fixture.destroy();
    mockActivatedRoute.paramsSubject$.next({itemId: 'itemIdParam'});
    expect(itemServiceSpy.loadItemDetails).not.toHaveBeenCalled();
  });
});

class MockActivatedRoute {
  /** Used within the test code to trigger a new param change */
  readonly paramsSubject$ = new Subject<Params>();
  /** Used by the component to subscribe to params */
  readonly params = this.paramsSubject$.asObservable();
}
```

Note that this is only important to do for Observables that donâĂŹt complete, such as route params, and store subscriptions, etc. For things like HTTP observables, you generally donâĂŹt have to worry about unsubscribing from them, since they always complete.

While the above practice works for a single subscription, this will become unwieldy for multiple subscriptions. A clever solution to this is to use a single parent subscription to manage all child subscriptions, and then only call `unsubscribe()` on the parent subscriptions.

HereâĂŹs how we could modify the above code to accomplish it:

```
@Component({...})
export class MyComponent implements OnInit, OnDestroy {
  private subscriptions: Subscription;

  constructor(
    private route: ActivatedRoute,
    private itemService: ItemService
  ) {}

  ngOnInit() {
    this.subscriptions.add(
      this.route.params.subscribe(params => {
        this.itemService.loadItemDetails(params['itemId']);
      })
    );
    // Any other subscriptions could be added this way as well.
  }

  ngOnDestroy() {
    this.subscriptions.unsubscribe();
  }
}
```

Note that the unit tests will remain the same.

Alan Chautard again has an excellent post on this topic from his blog.

If you donâĂŹt like the idea of putting subscriptions with `.add()` blocks, you could also use `takeUntil()` in combination with a destroy subject to handle this.

```
@Component({...})
export class MyComponent implements OnInit, OnDestroy {
  private destroy$ = new Subject<void>();

  constructor(
    private route: ActivatedRoute,
    private itemService: ItemService
  ) {}

  ngOnInit() {
    this.route.params.pipe(
      takeUntil(this.destroy$),
    ).subscribe(params => {
      this.itemService.loadItemDetails(params['itemId']);
    })
  }

  ngOnDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
  }
}
```

In most cases, you should try to use the `aync\$` pipe when possible.

47.3 Not dealing with errors

Error handling in RxJS can be tricky, because thereâĂŹs not only lots of ways to handle it, but there are many ways it could go wrong. Understanding how to deal with errors effectively is a key part of building robust Observable APIs. If you havenâĂŹt yet, IâĂŹd highly recommend reading Angular UniversityâĂŹs RxJS error handling guide. It is the most comprehensive guide I have seen on the topic.

47.3.1 What to watch out for

Dangling subscriptions.

47.3.2 What to do instead

Add all subscriptions to a parent subscription, then unsubscribe() from that parent subscription on destroy.

47.4 Reinventing the wheel

While itâĂŹs true that you can recreate redux with a single line of RxJS code, it doesnâĂŹt necessarily mean you should. When you start to find yourself managing a large amount of stateful data using Observables, you should consider using a pre-built library such as NgRx, NGXS, or Akita. These libraries will not only give you a lot of functionality for free, but theyâĂŹll save you from having to maintain your own bespoke solution, and are already used and understood by developers outside of your team, making it easy to onboard new developers.

If you currently have an app that you need to migrate over to one of these state management libraries, I recommend starting off by implementing a view facade to abstract your current logic, and then once you have the facade in place, introduce your state management library of choice. This will ensure your componentâĂŹs interaction with the underlying state management system wonâĂŹt change at all, no matter what that solution is, giving you a clean separation of concerns along the way!

47.4.1 What to watch out for

Replicating too much of what other state management libraries out there are doing.

47.4.2 What to do instead

Use an existing state management library. If dealing with existing code that uses bespoke state management, start by refactoring out the view facade and then introduce the state management library behind the facade.

47.5 Exposing subjects as part of a read-only API

Many times when writing services, youâĂŹll want to use a subject to be able to write reactive state. However, we have to be careful not to expose the subject itself to the outside world. This would allow any client using the code to update the state of the service, which would break the principle of data encapsulation.

Instead, you should make your subjects private and use `asObservable()` to expose a readonly version of that subject for use by clients. LetâĂŹs take another look at that UserService we used to talk about mishandling subscriptions:

```
@Injectable()
export class LoginService {
  private currentUserSubject$ = new ReplaySubject<string>(1);
  readonly currentUser$ = this.currentUserSubject$.asObservable();

  private authTokenSubject$ = new ReplaySubject<string>(1);
  readonly authToken$ = this.authTokenSubject$.asObservable();

  private authErrorSubject$ = new ReplaySubject<any>(1);
  readonly authError$ = this.authErrorSubject$.asObservable();

  constructor(private http: HttpClient) {}

  login(username: string, password: string) {
    this.http
      .put('http://localhost:8000/login', {username, password})
      .subscribe({
        next(data) {
          this.currentUserSubject$.next(username);
          this.authTokenSubject$.next(data['authToken']);
        },
        error(err) {
          this.authError$.next(err);
        }
      });
```

```
  }
}
```

Notice that for all of the subjects we used, we exposed them using `asObservable()`. Within our `login()` method, users simply call that method and the observables update themselves:

```
@Component({
  selector: 'app-login-example',
  template: `
    <ng-container *ngIf="loginService.currentUser$ | async as user; else loginForm">
      <h1>Welcome, {{user}}!</h1>
    </ng-container>
    <ng-template #loginForm>
      <form [formGroup]="form">
        <label>Username: <input formControlName="username" type="text" /></label>
        <label>Password: <input formControlName="password" type="password" /></label>
        <button (click)="loginService.login(form.get('username').value, form.get('password').value)"
                [disabled]="form.invalid">
          Log In
        </button>
      </form>
    </ng-template>
  `
})
export class LoginExampleComponent {
  readonly form = this.fb.group({
    username: this.fb.control('', Validators.required),
    password: this.fb.control('', Validators.required),
  });

  constructor(readonly loginService: LoginService,
              private fb: FormBuilder) {}
}
```

Notice how the observable attributes can only be read from, never written to. This prevents clients from being able to modify the internal state of the service, ensuring clean encapsulation.

47.5.1 What to watch out for

Exposing subjects as part of an API.

47.5.2 What to do instead

Make the subject private and use asObservable() to expose it as a public property.

47.6 Nesting subscriptions

This is one that IâĂŹve seen a lot of people do when theyâĂŹre first starting out with RxJS. ItâĂŹs very similar to how people nest .then() calls in Promises when theyâĂŹre first starting out using them. TheyâĂŹll have some asynchronous pieces of code that need to be executed in sequence, and theyâĂŹll accomplish this by nesting one .subscribe() inside of another.

For example, imagine that you were building a metrics dashboard application. On load, it fetches a configuration detailing which data it needs to load for a user. It then proceeds to load the data based on the configuration. These are two different asynchronous operations. You want them to be asynchronous so that you can begin to progressively render parts of the page based on the config. Using nested subscribes, that code might look like this:

```
@Component({
  selector: 'app-dashboard-page',
  ...
})
export class DashboardPageComponent implements OnInit {
  config: Config = null;
  data: Data = null;

  constructor(
    private configService: ConfigService,
    private dataService: DataService) {}

  ngOnInit() {
    this.configService.getUserConfig().subscribe(config => {
      this.config = config;
      this.dataService.loadData(config).subscribe(data => {
        this.data = data;
        // ...
      });
    });
  }
}
```

Notice how, even with two levels of nesting, we are slipping into pyramid of doom territory. What's worse, we can't take advantage of our async pipes because we have to store the returned data as properties on the component instance!

Instead, you can use higher-order mapping functions in order to flatten nested Observables into a single Observable. For example, the previous code could be written as:

```
@Component({
  selector: 'app-dashboard-page',
  ...
})
export class DashboardPageComponent implements OnInit {
  config: Config = null;
  data: Data = null;

  constructor(
    private configService: ConfigService,
    private dataService: DataService) {}

  ngOnInit() {
    this.configService.getUserConfig().pipe(
      tap(config => {
        this.config = config;
      }),
      mergeMap(config => this.dataService.loadData(config))
    }).subscribe(data => {
      this.data = data;
      // ...
    });
  }
}
```

Here, we use the mergeMap operator to take the data returned from the config service, and use it to produce a new observable using the data service. That Observable is then passed down the rest of the operator chain, and allows you to subscribe to it. This is better, but itâĂŹs still not quite ideal. We have to subscribe to the data in order to store it on our component, and we also have to use tap() to add a side-effect of storing the config object before itâĂŹs replaced in the operator chain via the mergeMap call.

A better strategy here is to not subscribe at all, and instead store config and data as Observables.

```
@Component({
  selector: 'app-dashboard-page',
  ...
})
export class DashboardPageComponent {
  config$ = this.configService.getUserConfig().pipe(
    shareReplay(1)
  );
  data$ = config$.pipe(
    mergeMap(config => this.dataService.loadData(config)),
    shareReplay(1)
  );

  constructor(
    private configService: ConfigService,
    private dataService: DataService) {}
}
```

Note that we use shareReplay(1) above to safe-guard against multiple subscriptions firing multiple HTTP requests. LetâĂŹs look at what weâĂŹve accomplished here:

1. We've completely removed the need for OnInit
2. We've reduced the amount of code weâĂŹve had to write
3. We've removed all subscribing and side-effect code completely from the component, making it fully reactive! We could easily add push-based change detection to this component now, and take advantage of the async$ pipe in order to manage Observable subscriptions.

Becoming proficient with higher-order mapping functions, and reactive programming in general, will take some practice, but once you do your code will become a lot cleaner. If youâĂŹd like to dive deep into how higher-order mapping functions, and operators in general, work, I wrote an article on that recently :)

47.6.1 What to watch out for

Nested subscriptions.

47.6.2 What to do instead

Use higher-order mapping operators.

47.7 Conclusion

If you can eliminate most of these common mistakes from your code, you wisll be well on your way to writing elegant async code with RxJS.

48 RxJS and Facades

48.1 The point of this Chapter

This entire chapter, is going to discuss how you can create your own "light" internal state, using RxJS and Facades. However, the WHOLE point of this, is to make you strongly consider, using Ngrx/store in the first place, without using Facades + RxJS. In addition, proving to the developer, create an internal service for a component, that interacts with an http service, likewise should use ngrx/store in the first place.

I think this is important to read through, because there is going to be a time in your career already, where you have decided, /lstinlinengrx/store might be too much for what you are working on. Likewise the point of this article is to push back on that urge, and how ngrx/store will actually the decrease the time it takes to build the application long term.

48.2 Recap on the Facade Pattern

I want this piece of information to be as authentic as possible. Therefore, I would like to present the facade pattern in the way that I first learnt how to do so. There is a fundemental book on computer science called "Design Patterns: Elements of Re-usable Object-Oriented Software" otherwise known as the GoF(Gang of Four) book due to it's four authors. It's a bit of classic in software, and I would say analogous to, "How to Win Friends and Influence People" by Dale Carnegie, if you are familiar. In the GoF book, it discusses the idea of the facade pattern. Paraphrasing it:

1. Implements a singular interface that contains multiple interfaces. Those interfaces work through the singular interface.

    a) This helps readability by making the name more straight to the point. For instance, in our scenario, `UpdateTodo`, vs. `this.store.dispatch(new TodoUpdated(TodoPayload))`.

    b) Usability by removing need of dependencies, and truncated form of functionality. For instance, in our scenario, we are using `UpdateTodo`, there is no need to include

dependency for store. In addition, developer can now just type in `UpdateTodo` instead

2. Provide context specific interface. This is non relevant within an ngrx setting, as front end services by default, without state, is generally very, very specific.

3. Serve as a launching point for a broader re-factor, or a tightly coupled system, in favor a more loosely coupled code. For instance, in our scenario, by tying all of our state underneath a singular facade, if at a later date we want to swap out the tech needed to manage state, we can do that by simple changing the logic within a particular location.

48.3 Recap: Ngrx/store + Facades

Just to re-iterate, let's say that we have a todo app, and we create a `todo.facade.ts` file:

```
@Injectable({})
class TodoFacade {
  constructor(private store: Store<any>) {}

  todos$: Observable<Todo[]> = this.store.pipe(select(getTodos));
  idOfTodos$: Observable<Todo[]> = this.store.pipe(select(getTodosIds));
  loaded$: Observable<boolean> = this.store.pipe(select(getTodosLoading)
      );

  UpdateTodo(TodoPayload): void {
    this.store.dispatch(new TodoUpdated(TodoPayload));
  }
}
```

Listing 48.1: todo.facade.ts

Our facade within an ngrx/store setting is doing three unique things.

1. Handling the store constructor for us. (With regards to unit testing this makes life alot easier.)

2. Allows us to have to only put selector in one location(our facade), and trickle it down to all other areas.

3. Create a simpler to use interface for our actions, that can be re-used time and time again. Especially if we decide to change the logic, or dynamics of how action works, we only have to update in one particular area.

48.4 Progressing Idea of Facades over to RxJS

Similarly, the idea of facades is incredibly valuable within RxJs. Let's imagine the most heavily used RxJS use case(atleast the most heavily used one I've come across). Perhaps, because there isn't a single application that doesn't use it, "Search".

Some of the business use cases with our particular company search includes:

1. Search by typing text into search bar(let's say here searching for companies)
2. Route between search, company, or company details view
3. Company details

Our particular RxJS code

```
@Component({
  selector: 'search-companies',
  templateUrl: './search-companies.component.html',
  styleUrls: ['./search-companies.component.scss']
})
export class SearchCompaniesComponent implements OnInit {
  companies$: Observable<Company[]>;
  searchCriteria = new FormControl();

  constructor(private companyService: CompanyService) {}

  ngOnInit() {
    // Observable stream to input control values
    const searchBy$ = this.searchTerm.valueChanges;

    this.companies$ = searchBy$.pipe(
      debounceTime(300),
      distinctUntilChanged(),
      startWith(''),
      switchMap((criteria:string) => {
        const request$ = this.companyService.searchCompanies(criteria);
        return !criteria.length ? of([]) : request$
      })
    );
  }
}
```

Listing 48.2: search-companies.component.ts

Let's dissect our RxJS code:

1. `debounceTime(300)` - Only after 300 milliseconds, will the request go through. If the user decides to type once again, within 300 milliseconds, the 300 millisecond count time, restarts again.

2. `distinctUntilChanged()` - Let's say the string the user inputs is "Apple". Then they decide to delete the e "Appl", but then add it again, "Apple", this will not trigger the http request. This is because, this is the exact same word that it was initially.

3. `startWith('')` - Not completely necessary, but will make sure that is an empty string everytime we initialize this page again.

4. `switchMap` - Powerful, because it has an internal project function, which allows us to break from the observables and use our "transformed" value. Here we are passing it to the service, which calls our compnayService, based on search criteria user passed in.

48.5 Addressing Architectural Concerns in Above Code

48.5.1 Concern One - Business Logic in View Layer

Obviously the logic here is contained within the view layer of the actual component. Best practices dictate, logic pertaining to view itself should be contained within component. Business logic, such as here, pertaining to when the service should fire should be moved to a different service.

48.5.2 Concern Two - State

Primarily around caching. In addition, updating the search criteria. For instance, if we want to search our application by companies, use one service. If we want to search by CEO's, then another function for CEOs. If we want to use another search criteria, let's say a particular category, then we will need another facade for controlling that. This ends up having alot of internal logic, and it greatly simplifies our app, by having the state be external. Especially, if would like to separate search from the data-table. Having the logic be re-usable, and be able to be used in multiple places, is useful.

48.6 Creating State Using RxJS

I would like to jump straight to this one. I don't want to jot down here the interface, rather the internal logic that we can create withing our SearchFacade. It would look something like this:

```
1 export enum CompanySearchActionTypes {
2   UPDATE_CRITERIA = 'Update Search Criteria',
3   UPDATE_RESULTS = 'Update Search Results'
```

```
4  }
5
6  export interface SearchAction {
7    readonly type: SearchActionTypes;
8    readonly payload: SearchCriteria | SearchResult[];
9  }
10
11 export class CompanySearchFacade {
12   searchResults$: Observable<SearchResult[]> = this.dispatch
13     .asObserable()
14     .pipe(map(state => state.companies), startWith([] as SearchResult[])
           );
15
16   searchCriteria$ = Observable<SearchCriteria> = this.dispatch.
         asObservable().pipe(map(state => state.criteria)
17   )
18
19   constructor(private companyService: CompanyService, private
         userService: UserService)
20
21   searchCompanies(companyName: string): Observable<SearchResult[]> {
22     // more code potentially goes here
23     pendingCompanies$.subscribe(companies => {
24       connst type = SearchActionTypes.UPDATE_RERULTS;
25       const action = { payload: tickets, type};
26       this.dispatch.next(
27         (this.state = reduce(this.state, action))
28       );
29     });
30   }
31
32   updateCompanyCriteria(companyName: string, categoryName: string) {
33     const type = SearchActionTypes.UPDATE_CRITERIA;
34     const payload = { user, ticket };
35     const state = reduce(this,state, {type, payload});
36
37     this.dispatch.next((this.state = state));
38   }
39 }
40
41 function reduce(state: SearchState, action: SearchAction): SearchState {
42   switch (action.type) {
43     case SearchActionTypes.UPDATE_CRITERIA: return {
44       ...state,
45       criteria: action.payload as SearchCriteria
46     };
47     case SearchActionTypes.UPDATE_RESULTS: return {
48       ...state,
49       companies: action.payload as SearchResult[]
50     }
51   }
52 }
```

Listing 48.3: company-search.facade.ts

Don't think too much into the above code. However, what we've proven is that we can

maintain a relatively simple store within our Facade simply using RxJS. That being said, I would like to make the following case for not doing the above and sticking with ngrx/store.

48.7 Why you should stick to ngrx/store

The assumption with doing the above, is that the code is light enough, to where we don't need to use ngrx/store. However, let's say within our code, we want to create a general company. Here we sort of move companies over to a cart of sorts, where we can then move these companies over an analytics platform. Here, we would be able to combine search, and our product cart under one store. Using the above facade, it would fracture our store between many different things. We might:

1. Want to create an ngrx/effect, so that whenever a user searches, it resets the product cart(who knows, maybe product wants it).

2. We want to modify our data to use ngrx/entity.

3. Maybe we want a history of all searches, to be used for internal analytics periodically.

4. Our reducers and actions bloat out of control, wherein it makes sense to put them into two separate files. While we are at it, we might as well be using state.

5. We have similar state logic elsewhere(let's say for our influencer search), and want to use that our company logic as well. It might be easier to keep them all in the same sort of structure.

6. It might make our code more brittle. Who's to say now that we have ngrx/store and rxjs/facades, that the team won't break up into more patterns. For instance, a one-off internal service?

For this reason, and many others, once you are building an enterprise application, it makes sense to use ngrx/store.

I've worked with many startups, and I've seen them work really quickly with ngrx/store as well. When done correctly, I would argue that ngrx/store speeds up team development, due to it's cookie cutter style of front end architecture

49 Creating a Config

Note: Creating your own config for the most part is not ideal. Ideally, configs should be altered in the backend and pulled in as an api. It should be noted, that any config has the potential of being exposed on the front end, therefore making front end config unsafe. However, many apps will experience iteration, even in larger enterprises. This chapter therefore deals with the importance of having configs available within app. In addition, the importance of having a config within app.

49.1 Use Case for Creating a Config

Just for clarity sake, I would like to present why one would want to create their own config. We are, of course, all aware of what the environemnt config that is naturally baked into every Angular project [1]. That is a great example of a great config file, which allows us to determine which values to use on the front end, based on environment. Another couple of situations where a config file might make sense is:

- Feature Flags
- Api Server Rules
- App Insights Key
- OAuth

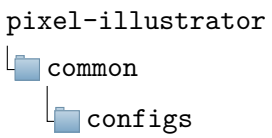
Technically, these can all be broken into a singular config. It would be a giant environment file. However, it would be cleaner to break these into smaller pieces, that give different values based on the central environment file.

49.2 How to Setup a Config File for App.

With the mono repo architecture we are using, let's assume that our code is going into the pixel-illustrator folder. It will be in the common folder, wherein there will be a configs

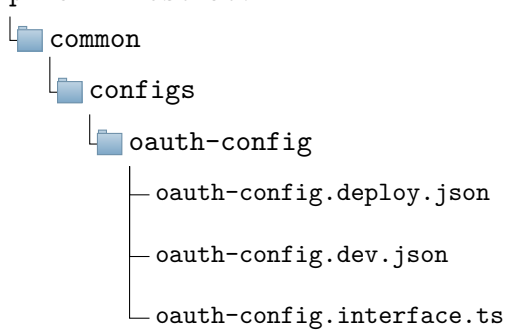
[1] If not, feel free to look here [Link to chapter on environemnt files goes here]

folder. Something like this:



49.2.1 How to Setup a Config File for App.

As config files, are giant data globs, we are going to want to create an interface for them. The actual config files will go in our config file. So let's say we want to create a config specifically for OAuth, we would create a config folder called oauth-config, with the following files:



49.2.2 Creating a Config Service

It is then important at this point to create a service, that will take it the appropriate config file, based on the appropriate envrionment.

```
1  import { Injectable } from '@angular/coreâĂŹ;
2  import { Http, Response } from '@angular/http';
3  import { environment } from '../environments/environment';
4  import { IAppConfig } from './models/app-config.model';
5
6  @Injectable()
7  export class AppConfig {
8
9      static settings: IAppConfig;
10
11     constructor(private http: Http) {}
12
13     load() {
14         const jsonFile = `assets/config/config.${environment.name}.json
               `;
15         return new Promise<void>((resolve, reject) => {
16             this.http.get(jsonFile).toPromise().then((response :
                   Response) => {
```

```
17                AppConfig.settings = <IAppConfig>response.json();
18                resolve();
19            }).catch((response: any) => {
20                reject(`Could not load file '${jsonFile}': ${JSON.
                      stringify(response)}`);
21            });
22        });
23    }
24 }
```

49.2.3 Load Files Prior to App Creation

Angular provides a token called APP_INITIALIZER, which will allow for our application to execute once application is finished.

```
1  import { APP_INITIALIZER } from '@angular/core';
2  import { AppConfig } from './app.config';
3
4  export function initializeApp(appConfig: AppConfig) {
5    return () => appConfig.load();
6  }
7  @NgModule({
8      imports: [ , , , ],
9      declarations: [ . . . ],
10     providers: [
11        AppConfig,
12        { provide: APP_INITIALIZER,
13          useFactory: initializeApp,
14          deps: [AppConfig], multi: true }
15     ],
16     bootstrap: [
17       AppComponent
18     ]
19 })
20 export class AppModule { }
```

We now have the option to use this config anywhere we want throughout the app.

50 Creating Feature Flags

Feature flags are an important part of Angular Architecture. What it means is, that a feature can be hidden, disabled, change code flow [1] or have routing being prevented from going to component.

50.1 Why are Feature Flags Important?

Feature flags are important, because they can allow the UI Engineering team to work independently from the back end and Automation/QA engineering. Some sample situations to illustrate this point:

Product asks for a new cc + bcc field to be added to the email client. We do not have the backend available yet, but would like to integrate it with the backend at a later date it is ready. Sure, we can put the UI on hold until back end is ready. However, what we can do, is test and hide ahead of time from the UI side of things. We can then hide the feature and integrate with backend when the time is right.

Another great example might be if this is a feature we only want to introduce to certain members. In addition, we might have admins within our app. These users when they log in, are given the option to go to the admin interface. Those who are not admins, are not allowed to see it. A feature flag might be something else that would be great.

Another use case of when a feature flag might be useful is if we have a new feature that we only want to show to our closest clients and see what they think about it, and give them the chance to sample it first. A feature flag would something incredibly useful in this situation as well.

Let's say there are features that haven't been QA'd yet, and they are holding back new features that are ready from making their way to stage. This feature can have a flag put on it, so that it only works for dev and not stage. That way, QA can test it, and when it is ready to go, we can go ahead and remove the feature flag. This would stop it from

[1] Changing code flow means having an if statement turned off for instance

inhibiting stage from having the push being made.

50.2 Creating a config using an API and server

Ideally a feature flag system is created through the backend. There would be some sort of management system, where a product owner can go in, and turn off features specific to a certain part of the application. There would then be a report that would go out monthly, let's say, that would tell everyone within the company which features have been turned off, to make sure there isn't dead code laying around for features that aren't used. I will not go into detail here, but this is ideally how this should be built out. [TODO discuss this option more in detail].

50.3 Creating a config

If your app currently does not have an api server that can be used in order to pull in the config, then you can create your own config. You can refer to the chapter on creating a config, on how someone would solve this. With regards to using the config with app, a service would need to be created as follows:

```
1  import { Injectable } from '@angular/core';
2  import { AppConfig } from '../../app.config';
3
4  @Injectable()
5  export class FeatureFlagService {
6
7    featureOff(featureName: string) {
8      // Read the value from the config service
9      if (AppConfig.settings.features.hasOwnProperty(featureName)) {
10         return !AppConfig.settings.features[featureName];
11     }
12     return true; // if feature not found, default to turned off
13   }
14
15   featureOn(featureName: string) {
16     return !this.featureOff(featureName);
17   }
18 }
```

51 Environment

When getting ready for production, it is important that you have different environment files for dev and prod. Luckily the CLI will offer two different environment files out the box. So, it should be immediately inherit to any developer experienced, or otherwise, that it is something that should be payed attention to. However, the full scope(gamut if you will) might not be immediately inherit, wherein certain elements migth not be looked into appropriately.

52 Environment Architecture - Deep Dive

52.1 API URL's

The most popular use of a environment file is to set up a url for prod and qa. QA will most likely be used by dev as well. Therefore, you will have something like the following in your UI:

```
1 export const environment = {
2   production: false,
3   envName: 'qa'
4 };
```

Doing something as simple as the above, will tag on 'qa' to your url. Being that in the reccomended architecture we are using GraphQL, we will have just one place where we have to specify the url we are using.

52.2 Cookie Names

Something I've seen in some projects as well, is wherein the cookie name will be different on dev than it will be on prod. This is a use for the environment file as well.

52.3 Miscellaneous

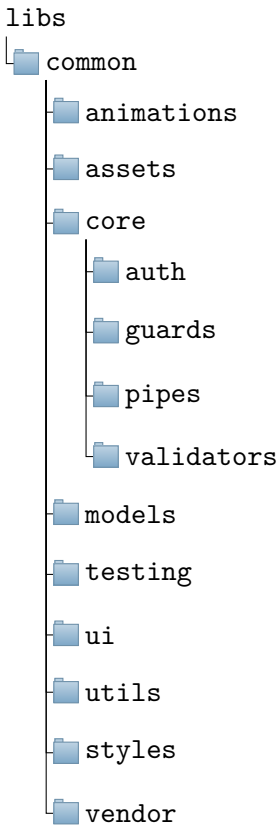
There might also be some other smidegts of use cases here and there, that one might want to use environment variables for.

52.4 Unit Testing

The real reason I am writing this chapter is for unit testing. I've seen alot of unit testing, where the environment variables get thrown in, and they hope for the best. In a proper development environment, we can implement unit testing.

53 Lib File Structure

When working in a monospace repo, file archticture is very important, as different parts of an application tend to be abstracted. First let's outline the different potential parts of an Angular application.



53.1 Lib File Structure in Detail

53.1.1 Animations

Animations are where any common animations might go. Things such as ripple effects, ghost elements, etc. Animations are really a whole different science when it comes to

development, and therefore makes sense for them to have their own folder.

53.1.2 Assets

This is where commonly re-used assets, such as icons, or logos are used.

53.1.3 Core

Any piece of functionality that is re-used the app is used here.

53.1.4 Models

This is where interfaces used across the app are used. This will simply type annotations across the app. In addition, when using a place to reference the full capacity of data requests within app, this model will be a life saver.

53.1.5 Testing

All data mocks that are common, can go here as well. Data tends to be re-used many different times within different parts of apps. So ideally, all mocks should go here.

53.1.6 UI

All presententational components go here. Some examples of might go into an example app is a header, footer, loading spinner, or charts.

53.1.7 Utils

Shared services used across different apps. For instance, dates.

53.1.8 Styles

Any shared styles used across app, for instance spacing, media queries etc.

53.1.9 Vendor

The vendor folder is used to customized 3rd party libraries. Some libraries customized might include Angular Materal.

54 Setting Up Lib Folder Structure

In the previous chapter we discussed what an ideal lib folder structure should look like. Setting up an architecture that looks like this can be a bit cumbersome. In addition, it is repeated time and time again.

We should create a schematic for this. In fact, now would be a good time to running through creating a schematic.

55 Data Access

When working within a GraphQL app, having a manageable way of accessing data is important.

55.1 Data Access Folder/File Structure

```
buyers
  state
    src
      lib
        data-access
          buyer.interface.ts
          buyers.service.spec.ts
          buyers.service.ts
          buyers.fragments.ts
          buyers.mutations.ts
          buyers.queries.ts
        +state
          buyers.actions.ts
          buyers.effects.spec.ts
          buyers.effects.ts
          buyers.facade.spec.ts
          buyers.facade.ts
          buyers.reducer.spec.ts
          buyers.reducer.ts
          buyers.selectors.spec.ts
          buyers.selectors.ts
        buyers-state.module.ts
    index.ts
```

55.2 Data Access Deep Dive

There are really four parts in one with regards to this folder/file structure. They are

1. GraphQL
2. Models
3. Service/Facade
4. State

55.3 The Data Access Life-Cycle

First, one is to specify fragments and queries for the GraphQL request. At this time, one is to specify the the interface, which is a UI carbon copy of the fragment. Next, one is to create a service that will be used for the GraphQL request. It will supply the proper params for the GraphQL request. Then depending on the type of compoennt that the data will be used for. Creating an effect that will be used in conjuction with the facade for providing data, that can be used across the app is appropriate.

This is a cookie cutter process with regards to loading data, that allows for repeating the process time and time across the app. In addition, it tightly couples the elements of your app together. For instance, it will use a singular interface across the app, which will then be able to be used in unit tests, and actual app alike. Therefore, one can continue mocking one's services, and by using interfaces make sure the data passed through across different parts of app is the same.

56 Nx Lib Conventions

56.1 Why is an NX Workspace different than an NPM Repo?!

One of the difficult things when starting with an Nx workspace for the first time, is comparing it to an NPM repo. One may ask, why is it different than an actual NPM repo?

1. All code in the lib can be modified by any one person in the ogranization, as opposed to an NPM repo. It is therefore cheaper, and faster.

2. There is no need to upgrade dependencies. Editing and updating a lib, will automatically affect all apps.

3. It is cheaper to create a lib, as opposed to creating a new NPM package. No need to set up a CI/CD, or otherwise.

56.2 Put Everything into Libs

Everything should be put into Libs.

Note: Todo - The following are the different types of libs that should be put into app.

57 Data Services - Directory Structure

In an @ngrx/store setting, a very large part of an app's services will be directed towards handling data directly. For instance, let's say you have a GraphQL user query that get's a current user's data. The service would look something like this:

```
1  import { Injectable } from '@angular/core';
2
3  import { Observable, from } from 'rxjs';
4  import { pluck } from 'rxjs/operators';
5  import { Apollo } from 'apollo-angular';
6
7  @Injectable({ providedIn: 'root' })
8  export class UserService {
9    getUser(): Observable<User> {
10     const user$ = this.apollo.query({ query: GetCurrentUser });
11
12     return from(user$).pipe(pluck('data', 'getCurrentUser'));
13   }
14   constructor(private apollo: Apollo) {}
15 }
```

It's a rough guess, but odds are that 80% of services within any Angular app, will be built to directly work with data. These services will be used by a number of effects. In addition, they will contain their own respective unit tests.

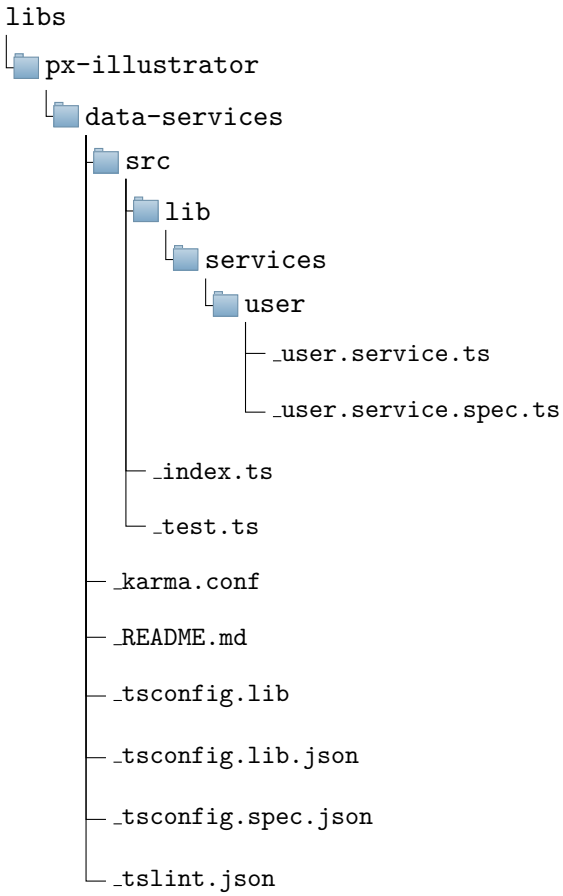
57.1 Setting the Landscape

As of Angular 6, Angular offers a metadata option called providedIn. In short, if providedIn is set to 'root', it allows us to bypass the need of using a module, and creates a tree-shakable version [1].

Therefore, the bundler will be able to tell if this service is being used in a particular module. If it is not, service will not be bundled with particular module. This allows us to be very liberal with our services and put them in a central location. (Of course we would be able to do this with a module, but it make's it more architecturally appealing without the need of using a module)

[1] https://angular.io/guide/dependency-injection-providers#creating-tree-shakable-providers

57.2 Data Services Folder/File Structure



Just as simple as that. We have a central location called data-services where we put bulk of services that deal with retrieving data, and returning it as an observable. If you want to be really technical about it, you can create every data-service as it's own lib. However, I think this is a bit much, as services tend to not touch the DOM and run really fast as unit tests.

58 Dialogs

Dialogs are an essential part of any app. For that reason, having a consistent architecture for dialog architecture will greatly increase maintainability of one's app. However, the technique that is used for dialogs, is something which can be used across many different components across one's app.

59 Lazy Loading Modules

One of the more initally overlooked pieces of UI Architecture, is with regards to lazy loading. Lazy loading, if not already familiar, is the concept of loading something when it is required, rather than all at once on page load. Similar to how lazy people only do things when it is required of them, lazy loading will only load when it is required for the page it is navigating to. With regards to Angular, lazy loading is a routing/module architecture, and heavily tied to routing.

The main benefit of lazy loading, is so that on initial load of the web page, we drastically decrease the bundle size. This improves user experience. Thankfully, Angular makes it relatively easy to include a lazy loaded module into the app. The Angular CLI even has a command, for easily setting up a lazy loaded route. However, before we go ahead and show the command, that automatically scaffolds lazy loading for us, let's discuss how to add a lazy loaded route if we were to do that process manually.

59.1 Adding a Lazy Loaded Module Using Angular CLI

```
ng g lib about --routing --lazy --directory=razroo
```

This command will automatically add a module to our lib. In addition, will modify the route within the about.module.ts, so that it can be used as a lazy loaded route.

An important note is that all the additional –routing –lazy flags will add is this line: RouterModule.forChild([/* path: '', pathMatch: 'full', component: InsertYourComponentHere */])
If you are upgrading an existing module to use lazy loading, the following directions will work for you as well.

59.2 What We Should Edit Post Generation

Now that we have generated a route for our "about" page, let's make the two edits required post CLI generation.

59.2.1 Editing app.module.routing.ts File

Edit one, will be in our main `app.module.routing.ts` file:

```
{
  path: 'about',
  loadChildren: () =>
    import('@razroo/razroo/about').then(
      module => module.RazrooAboutModule
    )
},
```

1

You will notice two things in the above code:

1. A `path` key, standard for Angular routing, to specify what module should be loaded when navigating to a specific route.

2. A `loadChildren` key, which calls a function followed by the standard syntax for importing a module.

In addition, being that we are using Nrwl Nx (which this book is littered with) the import path is using our Nx workspaces shortened path. Here that would be the `razroo-workspace/razroo-lib/lib-name`.

The second edit for us to make, will be in the actual module for our about page:

```
@NgModule({
  imports: [
    //...
    RouterModule.forChild([
      {path: '', pathMatch: 'full', component: AboutComponent}
    ])
  ],
  declarations: [AboutComponent]
})
export class RazrooAboutModule {}
```

> You will notice that in the above code we are actually using an AboutComponent. This component will need to be generated in addition to module we've already created, and can be done simply by navigating to our libs folder, and runnning `ng g component about`.

The above is the cookie cutter process involved with creating a lazy loaded module within an Angular application. Thankfully, due to Angular, it is relatively painless process for what it is accomplishing. It is architecture that is worth implementing early on in the app. In particular, it might save you from circular dependency nightmares later on.

[1] Just in case you are familiar with a different syntax, this is the latest syntax for Angular 8+.

60 Transloco

60.1 What is Transloco

Transloco is a relatively new library for translating in Angular.

61 Lazy Loading Images

Even though we discussed lazy loading modules, we might also want to lazy load the content inside of the lazily loaded modules. There is a very popular article from Google that is generally spread around with regards to performance for webpages. In short, it presents the following very persuasive set of data:

As page load time increases	
Seconds	Probability of Bounce
1s to 3s	Increases by 32%
1s to 5s	Increases by 90%
1s to 6s	Increases by 106%
1s to 10s	Increases by 123%

As we can see with the data presented above, performance of our webpages are very important. More so it presents the data very clearly that the faster a webpage is, the higher probability there is to retain our user base.

61.1 The Idea of Lazy Loading Images

There are, of course, many different ways to increase performance. The intent of this chapter, however, is just to discuss the one performance boost that is gained by lazy loading images. The idea of lazy loading images, similar to lazy loading modules in general, is to load an image only when a user get's to that image.

61.1.1 Side bar - User Experience and Lazy Loaded Images

It is important to note, that we do not want to lazy load all images that are present on our page. For instance, imagine that we were creating a blog for our website. On each single blog page, we have a feature image that shows up first for our blog. In addition, we have more images that show up throughout the remainder of the blog. It can be strongly argued that lazy loading should not be applied to the feature image. Because, it would potentially

cause an awkward experience for the user, to load the page and then wait an additional second to see what the feature image looks like.

Therefore, when creating lazy loaded images, it is important to not create a blanket rule that will apply to all images. Rather, those images which are not primary to the page, those should be lazy loaded.

61.2 Real Talk - Implementing Lazy Loading

Within Angular, there would be a way to implement lazy loading from scratch. The simplest solution I've seen is wrapping an Angular Directive around the `Intersection Observer` api. It will allow you to determine when an element is in the viewport. This approach , works great, however, there are readily available npm plugins that do this for Angular. While the author of the aforementioned article did create their own plugin there is another that is much more mature. It implements the intersection observer as well, and is called `ng-lazyload-image`.

61.3 ng-lazyload-images

Razroo's preferred package is ng-lazyload-image. It is the most mature of all packages in the Angular eco-system, has the most stars, and fantastic documentation.

61.3.1 Install

```
npm install ng-lazyload-image --save
```

61.3.2 Setup

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { LazyLoadImageModule } from 'ng-lazyload-image'; // <-- import
    it
import { AppComponent } from './app.component';

@NgModule({
    declarations: [ AppComponent ],
    imports: [ BrowserModule, LazyLoadImageModule ], // <-- and include
        it
    bootstrap: [ AppComponent ]
})
export class MyAppModule {}
```

61.3.3 If Using IE

If want to use IE, will need to use a polyfill.

```
  import { NgModule } from '@angular/core';
  import { BrowserModule } from '@angular/platform-browser';
  import { LazyLoadImageModule, intersectionObserverPreset } from 'ng-
      lazyload-image'; // <-- include intersectionObserverPreset
  import { AppComponent } from './app.component';

  @NgModule({
      declarations: [ AppComponent ],
      imports: [
        BrowserModule,
        LazyLoadImageModule.forRoot({
          preset: intersectionObserverPreset // <-- tell LazyLoadImage
              that you want to use IntersectionObserver
        })
      ],
      bootstrap: [ AppComponent ]
  })
  export class MyAppModule {}
```

61.4 Example Use Case

The following is an example use case, of how to create a lazy loaded image with ng-lazyload:

```
import { Component } from '@angular/core';

@Component({
  selector: 'image',
  template: `
    <img [defaultImage]="defaultImage" [lazyLoad]="image">
  `
})
class ImageComponent {
  defaultImage = 'https://www.placecage.com/1000/1000';
  image = 'https://images.unsplash.com/photo-1443890923422-7819ed4101c0?
      fm=jpg';
}
```

The package also has examples for multiple other use cases such as background images, responsive images, async images(i.e. `| async`) etc.

61.5 Transitioning Photos

Something that you might want to do is add a transition to your photo, so that it swaps out the defaultImage with the lazyLoaded image. Doing something like this would be relatively straightforward. For instance:

```
1 img.ng-lazyloaded {
2   animation: fadein .5s;
3 }
4 @keyframes fadein {
5   from { opacity: 0; }
6   to   { opacity: 1; }
7 }
```

That is all it would take to put a transition effect on your photos.

61.6 Hooking Up Lazy Loading To Our Back End

I am going to assume that the reader is intelligent enough to figure out how to hook up the backend to their component, without the need of this book. Therefore, I just wanted to mention the strategy really quick. In short, you would get all image urls from the actual GraphQL query. You would put them underneath the `lazyload` directive. Just like that you have lazy loaded configured.

62 Network Aware Predictive Pre-Loading

Lazy loading modules in Angular, allows for javascript loading to be optimized. That is, so browser only loads the page the user is navigating to. This helps to decrease the initial load time. However, depending on how expensive including a module is, pre-loading can save you time. Pre-loading is a strategy that allows for modules in Angular to be loaded as soon as possible. Modules can be pre-loaded either all at the same time, or a select few, or when a custom event happens. You can check yourself how long it takes for a module to load, and the potential value of pre-loading. Open up your developer console tool(I'm using Chrome), navigate to the js section, of the network tab, and then load the page you want to test.

62.1 Being Aware of How Much Time Pre-Loading Saves

You might be curious as to how much time is actually saved with regards to pre-loading? I was curious as well. I tried personally(as shared in the screenshot above), and the amount of time saved, to be honest, is negligible. However, I also realize that the app I am working on has a minimal amount of modules. I can see that for another app, wherein there are multiple modules that are loaded. Therefore, let's throw out an arbitrary number. If you have a module that is going to use more than 20 imports inside of it's module, then worry about a pre-loading strategy. Regardless, it is something to be aware of, and here is how to go around implementing pre-loading.

62.2 Pre-Load Everything

While this strategy will rarely work for any real-world application, there is an option to pre-load every module in Angular. To do so, you would use `PreloadAllModules` as your preloading strategy:

```
1 import { RouterModule, PreloadAllModules } from '@angular/router';
2
3 @NgModule({
4   imports: [
5     RouterModule.forRoot(routes, {
6       preloadingStrategy: PreloadAllModules,
7     }),
```

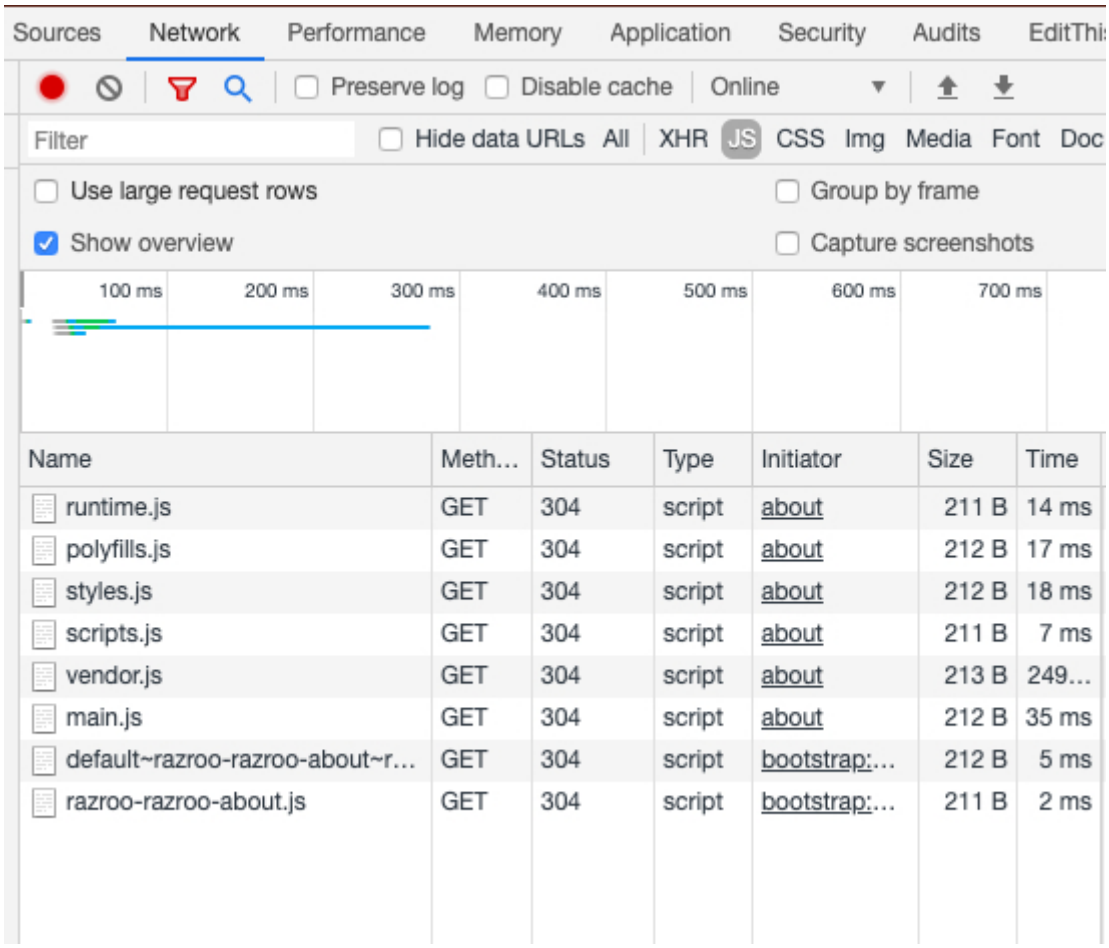


Figure 62.1: As we can see, it took 2ms to download the Razroo About page.

```
  ],
})
class AppRoutingModule {}
```

62.3 Custom Pre-Loading

What does make more sense in an enterprise setting, is custom pre-loading modules. That is, pre-load the more expensive modules, and do not pre-load those that are less expensive. In addition, make the pre-loading happen at a time more convenient for the app. Let's dive into what that means and how we can do that.

Angular offers the ability to pre-load specific modules(as opposed to all of them at the same time, as we showed before). It offers a `preload` method that takes two arguments:

1. route - Route object to tap into, for the load function.

2. load - Function that when run, triggers the module being loaded

62.3.1 General Strategy

If we wanted to pre-load some modules, and did not want to pre-load others, we would follow the following strategy:

1. Give our route unique data(i.e. `preload: true`) to be used within our custom pre-loading function.

2. Create a custom pre-loading function, that makes use of our unique data.

3. Pass in custom pre-loading as a provider to the `preloadingStrategy` key.

62.3.2 Strategy Exemplified in Code

Give Route Unique Data

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

@NgModule({
  imports: [
    RouterModule.forRoot(
      [
```

```
        {
          path: 'books',
          loadChildren: () =>
            import('@razroo/razroo/books').then(
              module => module.RazrooBooksModule
            ),
          data: { preload: true }
        },
        {
          path: 'consulting',
          loadChildren: () =>
            import('@razroo/razroo/consulting').then(
              module => module.RazrooConsultingModule
            )
        },
      ],
      {
        initialNavigation: 'enabled',
        relativeLinkResolution: 'corrected'
      }
    )
  ],
  exports: [RouterModule]
})
export class RazrooAppRoutingModule {}
```

Listing 62.1: app.routing.module.ts

Custom Function For Pre-Loading

```
export class CustomPreloadingService implements PreloadingStrategy {
  preload(route: Route, load: Function): Observable<any> {
    return route.data && route.data.preload ? load() : of(null);
  }
}
```

Listing 62.2: custom-preloading.ts

It is worthwhile to note that Razroo put's the `custom-preloading.ts` util file in the `libs/common/services` folder.

Pass in custom pre-loading as a Provider

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
import { CustomPreloadingService } from '@razroo/common/ui/services';

@NgModule({
  imports: [
    RouterModule.forRoot(
      [
        // ...routes go here
      ],
      {
```

```
12          preloadingStrategy: CustomPreloadingService
13          //...
14        }
15      )
16    ],
17    exports: [RouterModule]
18  })
19  export class RazrooAppRoutingModule {}
```

Listing 62.3: app.routing.module.ts

62.4 Enabling Module Pre-loading on a Custom Event

We can extend the pre-loading architecture one step further. We can tie in custom events into already existing custom pre-loading. In particular, we will implement a strategy, that when a user hovers over a navigation menu item, we can pre-load a module.

62.4.1 General Strategy For Event Driven Preloading

The general strategy will look somewhat similar to custom pre-loading, with some modified/added steps.

1. Give our route some unique data(i.e. preload: true) to be used within our custom-preloading function.

2. Create a separate service that will be used to trigger a next on the observable contained in the custom-preloading function.

3. Create custom pre-loading function, that makes use of our unique data. In addition, give it access to a `Subject`, so it can be triggerred, by an outside service.

4. Pass in custom pre-loading service as a provider to the `preloadingStrategy` key.

5. Use a mouseover function, that can trigger the service.

62.4.2 Strategy Exemplified in Code

We will be giving our route the same unique data for event driven module pre-loading as we did for custom module pre-loading:

Give Route Unique Data

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

@NgModule({
  imports: [
    RouterModule.forRoot(
      [
        {
          path: 'books',
          loadChildren: () =>
            import('@razroo/razroo/books').then(
              module => module.RazrooBooksModule
            ),
          data: { preload: true }
        },
        {
          path: 'consulting',
          loadChildren: () =>
            import('@razroo/razroo/consulting').then(
              module => module.RazrooConsultingModule
            )
        },
      ],
      {
        initialNavigation: 'enabled',
        relativeLinkResolution: 'corrected'
      }
    )
  ],
  exports: [RouterModule]
})
export class RazrooAppRoutingModule {}
```

Listing 62.4: app.routing.module.ts

Create a Separate Service to Trigger Pre-Loading

We will be creating a separate service, that will be used within our `CustomPreloadingService` to trigger preloading:

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

export class OnDemandPreloadOptions {
  constructor(public routePath: string, public preload = true) {}
}

@Injectable({
  providedIn: 'root'
})
export class OnDemandPreloadService {
  private subject = new Subject<OnDemandPreloadOptions>();
  state = this.subject.asObservable();

  startPreload(routePath: string) {
```

```
16      const message = new OnDemandPreloadOptions(routePath, true);
17      this.subject.next(message);
18    }
19  }
```

Listing 62.5: on-demand-preloading.service.ts

Custom Pre-Loading Service

Now we will be integrating our `OnDemandPreloadService` with our `CustomPreloadingService`

```
1  import { Injectable } from '@angular/core';
2  import { PreloadingStrategy, Route } from '@angular/router';
3  import { EMPTY, Observable, of } from 'rxjs';
4  import { mergeMap } from 'rxjs/operators';
5  import { OnDemandPreloadOptions, OnDemandPreloadService } from './on-
       demand-preload.service';
6
7  @Injectable({ providedIn: 'root', deps: [OnDemandPreloadService] })
8  export class CustomPreloadingService implements PreloadingStrategy {
9    private preloadOnDemand$: Observable<OnDemandPreloadOptions>;
10
11   constructor(private preloadOnDemandService: OnDemandPreloadService) {
12     this.preloadOnDemand$ = this.preloadOnDemandService.state;
13   }
14
15   preload(route: Route, load: () => Observable<any>): Observable<any> {
16     return this.preloadOnDemand$.pipe(
17       mergeMap(preloadOptions => {
18         const shouldPreload = this.preloadCheck(route, preloadOptions);
19         return shouldPreload ? load() : EMPTY;
20       })
21     );
22   }
23
24   private preloadCheck(route: Route, preloadOptions:
         OnDemandPreloadOptions) {
25     return (
26       route.data &&
27       route.data['preload'] &&
28       [route.path, '*'].includes(preloadOptions.routePath) &&
29       preloadOptions.preload
30     );
31   }
32 }
```

Listing 62.6: custom-preloading.service.ts

The most important piece with the above code, is that we are passing in the preloadOnDemandService as an observable, to the preload function. Therefore, we can tap into the internal Angular preload strategy and re-load it whenever we call our `OnDemandPreloadService` .

Pass in Custom Pre-loading Service as a Provider

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
import { CustomPreloadingService } from '@razroo/common/services';

@NgModule({
  imports: [
    RouterModule.forRoot(
      [
        // ...routes go here
      ],
      {
        preloadingStrategy: CustomPreloadingService
        //...
      }
    )
  ],
  exports: [RouterModule]
})
export class RazrooAppRoutingModule {}
```

Listing 62.7: app.routing.module.ts

Here there is nothing particularly novel about what we are doing. We simply inject our `CustomPreloadingService` into `preloadingStrategy`, to tell Angular to use it as our preloading strategy.

Trigger Service

In our particular scenario, as would make sense for alot of applications, is trigger module pre-loading on mouseover. For instance, when a user hovers over a menu item trigger pre-loading. We can therefore do something such as the following:

```
<a
  [routerLink]="item.link"
  class="nav-link"
  (mouseover)="preloadBundle('books')"
  >heroes</a
>
```

```
preloadBundle(routePath) {
  this.preloadOnDemandService.startPreload(routePath);
}
```

62.5 Creating a Directive

Using this method, we can also create a directive to allow the logic for preloading to be re-usable.

```
import { Directive, ElementRef, HostListener } from '@angular/core';
import { OnDemandPreloadService } from '@razroo/common/services';

@Directive({
  selector: '[razrooPreload]'
})
export class PreloadDirective {

  constructor(private elementRef : ElementRef,
              private onDemandPreloadService: OnDemandPreloadService) {}

  @HostListener('mouseenter')
  onMouseEnter() {
    const pathName = this.elementRef.nativeElement.attributes.routerlink
        .value;
    this.onDemandPreloadService.startPreload(pathName);
  }
}
```

Listing 62.8: preload.directive.ts

In the above code for the directive we are assuming that there is a routerlink, on the a tag we are looking for. [If there isn't, the directive will return an error.] In addition, we are tapping into the native element, so that we can get the pathname we need. Should add, that for the `routerlink` directive on actual element, not using a forward slash. If you are using a forward slash, you will have to add in logic to remove forward slash. This allows us to now simply add the following:

```
  <a routerLink="books" razrooPreload></a>
```

When the user mouses over, we will now be able to see in our chrome console, that the appropriate module has been pre-loaded.

62.6 Network Aware Pre-Loading

We've created an on-demand pre-loading strategy. When a user hovers over a menu item, before they click, and navigate to route, it will already begin to pre-load respective module for the route. It's a very clever strategy, that uses the user's own intent to maximize the app's performance. However, let's say in our pre-loading strategy, we have three pages.

1. About
2. Product
3. Application Page

The application page opens up a very heavy js bundle, that take's about .25 seconds to load

on a fast network. However, on a slower network it might take around an entire second. In such a slow network, if the user hovers over the larger Applications menu item first, and then the much smaller About page, our pre-loading strategy might have the reverse effect in such a slow network(the user will not have to wait longer for the about page to load). If we want an air tight strategy that takes care of all scenarios, it makes sense for us to go ahead and take network connection into account.

62.6.1 Network Information API

I have seen some blogs in this regards suggesting use of the network information api, for figuring out the speed of the network. It's actually interesting, W3C specs have included talks for the Network Information API since 2011. There have been many discussions since then, landing on the latest version solidified in 2014.

First and foremost, before discussing how to use this API, it probably makes sense to go ahead and discuss that for the API we plan on using, current usage is at about 70%. It's worth noting that the lions share of the percentage is with regards to IOS Safari(a whopping 10%), which does not support this feature. In addition, Firefox currently does not support this feature. However, Chrome which is used by the majority of users between Android and Desktop is at about 60% on it's own.

Still the API has ways to go, so understandably, this is not a fool proof solution. However, seeing as support is somewhat decent, and this is relatively easy to implement, let's go ahead and do so.

62.6.2 Strategy Exemplified in Code

Since the Network Information API, is still experiemental technology, Typescript will not support it as part of it's core library yet. So we will simply declare a var for navigator to make it go away. Let's add a has `GoodConnection` function to our `custom-preloading.ts` file, so that we can tell whether, or not a user has a good connection.

```
1  export declare var navigator;
2  hasGoodConnection(): boolean {
3    const conn = navigator.connection;
4    if (conn) {
5      if (conn.saveData) {
6        return false; // save data mode is enabled, so dont preload
7      }
8      const avoidTheseConnections = ['slow-2g', '2g'];
9      // 'slow-2g', '2g', '3g', or '4g'
10     const effectiveType = conn.effectiveType || '';
11     if (avoidTheseConnections.includes(effectiveType)) {
12       return false;
13     }
```

```
  }
  return true;
}
```

Listing 62.9: custom-preloading.ts

In the above function, we are using the navigator.connection effective types, which tells us the connection that the user is currently experiencing. If it is slow-2g, or 2g, we return false in this function. We can now hook it up into the pre-emptive loading code we had before.

Because the network information api is experimental technology, let's just add a custom type declaration in our file, to knock out any type errors that we receive. Putting it all together, our file will something like the following:

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { EMPTY, Observable, of } from 'rxjs';
import { mergeMap } from 'rxjs/operators';
import { OnDemandPreloadOptions, OnDemandPreloadService } from './on-
    demand-preload.service';
export declare var navigator;

@Injectable({ providedIn: 'root', deps: [OnDemandPreloadService] })
export class CustomPreloadingService implements PreloadingStrategy {
  private preloadOnDemand$: Observable<OnDemandPreloadOptions>;

  constructor(private preloadOnDemandService: OnDemandPreloadService) {
    this.preloadOnDemand$ = this.preloadOnDemandService.state;
  }

  preload(route: Route, load: () => Observable<any>): Observable<any> {
    return this.preloadOnDemand$.pipe(
      mergeMap(preloadOptions => {
        const shouldPreload = this.preloadCheck(route, preloadOptions);
        const hasGoodConnection = this.hasGoodConnection();
        if(shouldPreload && hasGoodConnection) {
          return load();
        }
        else {
          return EMPTY;
        }
      })
    );
  }

  private preloadCheck(route: Route, preloadOptions:
      OnDemandPreloadOptions) {
    return (
      route.data &&
      route.data['preload'] &&
      [route.path, '*'].includes(preloadOptions.routePath) &&
      preloadOptions.preload
    );
  }

```

```
40    private hasGoodConnection(): boolean {
41      const conn = navigator.connection;
42      if (conn) {
43        if (conn.saveData) {
44          return false; // save data mode is enabled, so dont preload
45        }
46        const avoidTheseConnections = ['slow-2g', '2g'];
47        // 'slow-2g', '2g', '3g', or '4g'
48        const effectiveType = conn.effectiveType || '';
49        if (avoidTheseConnections.includes(effectiveType)) {
50          return false;
51        }
52      }
53      return true;
54    }
55  }
```

Listing 62.10: custom-preloading.service.ts

You will notice, that we have added an additional condition for preload function. It now requires that it has a good connection, in addition to `shouldPreload` being true.

If you would like to try it yourself, throttle the network speed using the console and see that it doesn't download.

Congratulations, you are now an expert on knowing how to preload modules on your own. This is a very interesting strategy, and if you were to take if further, we can even begin to preload data, ahead of loading pages as well. Perhaps something we will discuss in another article.

63 Shared Modules

Within an Angular architecture, we have modules. Modules very commonly use the same sort of imports time and time again. Coming up with a sort of shared modules architecture, can help in two regards:

1. Prevent circular dependency issues.

2. Allow for easier imports within app.

63.1 Shared Modules in Practice

So in thoery, a shared module is simple. If there are a series of modules that need to be used time and time again, these are put into the shared module. However, in practice, it can be a very confusing thing, because what goes into a shared module. In addition, it can be concerning, because one has to know how it might affect performance.

63.1.1 Performance Impact of Unused Modules

As we mentioned earlier, the one potential issue with shared modules, is that they might affect performance. So the question is, that if we implement shared modules, and we include mutliple modules, that are not actually used by the component, will there be any performance issues to be concerned off. In that regard, let's dissect the code base.

As a test, I imported the `MatButtonModule` into the razroo website page module. The following code get's added to the general bundle as a result:

```
1 _angular_material__WEBPACK_IMPORTED_MODULE_11__["MatButtonModule"],
```

Listing 63.1: Extra code for module

If using AOT, the above will look a bit different. However, the resulting code is the same. It will call

Which will then call:

```
_angular_material__WEBPACK_IMPORTED_MODULE_11__ =
__webpack_require__(/*! @angular/material */ "../../node_modules/
    @angular/material/__ivy_ngcc__/esm2015/material.js");
```

```
// bundle.js
/******/ (function(modules) { // webpackBootstrap
/******/         // The module cache
/******/         var installedModules = {};
/******/
/******/         // The require function
/******/         function __webpack_require__(moduleId) {
/******/
/******/                 // Check if module is in cache
/******/                 if(installedModules[moduleId]) {
/******/                         return installedModules[moduleId].
    exports;
/******/                 }
/******/                 // Create a new module (and put it into the
    cache)
/******/                 var module = installedModules[moduleId] = {
/******/                         i: moduleId,
/******/                         l: false,
/******/                         exports: {}
/******/                 };
/******/
/******/                 // Execute the module function
/******/                 modules[moduleId].call(module.exports, module
    , module.exports, __webpack_require__);
/******/
/******/                 // Flag the module as loaded
/******/                 module.l = true;
/******/
/******/                 // Return the exports of the module
/******/                 return module.exports;
/******/         }
/******/
/******/
/******/         // expose the modules object (__webpack_modules__)
/******/         __webpack_require__.m = modules;
/******/
/******/         // expose the module cache
/******/         __webpack_require__.c = installedModules;
/******/
/******/         // define getter function for harmony exports
/******/         __webpack_require__.d = function(exports, name,
    getter) {
/******/                 if(!__webpack_require__.o(exports, name)) {
/******/                         Object.defineProperty(exports, name,
    {
/******/                                 configurable: false,
/******/                                 enumerable: true,
/******/                                 get: getter
/******/                         });
/******/                 }
/******/         };
```

```
47  /******/
48  /******/         // getDefaultExport function for compatibility with
        non-harmony modules
49  /******/         __webpack_require__.n = function(module) {
50  /******/                 var getter = module && module.__esModule ?
51  /******/                         function getDefault() { return module
        ['default']; } :
52  /******/                         function getModuleExports() { return
        module; };
53  /******/                 __webpack_require__.d(getter, 'a', getter);
54  /******/                 return getter;
55  /******/         };
56  /******/
57  /******/         // Object.prototype.hasOwnProperty.call
58  /******/         __webpack_require__.o = function(object, property) {
        return Object.prototype.hasOwnProperty.call(object, property); };
59  /******/
60  /******/         // __webpack_public_path__
61  /******/         __webpack_require__.p = "";
62  /******/
63  /******/         // Load entry module and return exports
64  /******/         return __webpack_require__(__webpack_require__.s = 0)
        ;
65  /******/ })
66  /*
        ************************************************************************
        */
67  /******/ ([
68  /* 0 */
69  /***/ (function(module, exports) {
70  /***/ })
71  /******/ ]);
```

Listing 63.2: webpack require source code

Webpack require takes something like less than a millisecond to perform. However, too many of them there in your application, and this can add a 1ms here, or there. Shared modules can potentially cause the following issues:

1. Performance issues. Shared modules only work within the context of services wherein one has the ability to use `providedIn: root`. However, for modules, they will be bundled regardless. So using shared modules within your app, can cause bloat in your application when it is not needed.

2. Maintanability. Modules can be difficult to know what exactly is contained within. The only module that consistently get's used time and time again, is `CommonModules`. `CommonModules` is used across all modules. However, if it's an arbitrary module, not used consistently across the app, it will end up being confusing for all team members.

For the above reasons Razroo feels that the general concept of Shared Modules does not make sense. It is something to be perhaps be re-factored at a later time if unique to your app.

64 Form Validation

One of main reasons why one would choose to use the internal Angular Form Group architecture, is the way it eases validation. Angular has a series of built in validators, that can be used for the fields in your form. For the sake of clarity, and brevity, I would like to jot down the name of current Angular Validators here, without going into detail:

1. min
2. max
3. required
4. requiredTrue
5. email
6. minlength
7. maxlength
8. pattern
9. nullValidator
10. compose
11. composeAsync

I think for the most part, the above validators speak for themselves as to what they do, with the exception of `compose`. `compose` will allow you to combine multiple validators, and return an error map for them. As you can see, the more popular errors, such as min, and max(for use with passwords/usernames) and emails, and patterns are what Angular's internal validators are there for. There is the ability to make custom validators, and most likely any app you work on, us going to need it's own custom validators, but before we go ahead and do that, let's see how we can integrate this with out application.

64.1 Integrating Form Validators within Component

Let's imagine that we have a newsletter component within our application. We want to ensure the user uses an email pattern, and it is also required. We write the following:

```
ngOnInit() {
  this.newsletterForm = this.fb.group({
    email: ['', [Validators.required, Validators.email]
    ],
  });

  get email() {
    return this.newsletterForm.get('email');
  }
}
```

As we can see using the above we have added two native angular validators to our formControl field email. This is the process wherein we would add Angular form validators to our application. It should be noted that in Angular, when we want to use multiple validators, we place them in a sub-array, within the array for the `fb.group`.

64.2 Integrating Form Validators within HTML Template

If we would like to integrate our form validators within our app, so that when we click on the button, they get triggered, we would do the following:

```
<ng-container *ngIf="email.invalid && (email.dirty || email.touched)">
  <mat-error *ngIf="email.errors.required">Name is required.</mat-error>
  <mat-error *ngIf="email.errors.email">Email is invalid</mat-error>
</ng-container>
```

Listing 64.1: Integrate Form Validation with HTML

In the above, we are creating a way of displaying the error if it appears. It should only appear if a user has actually touched the email field.

64.3 Custom Validators

As mentioned before, odds are your application, will also need it's own set of custom validators. Custom validators require their own function that returns true, based on value pass through. That function is then be hooked into the validators array. However, Razroo believes it is more scalable if we create a directive that can be used to automatically create validation for our form. In addition, if we create a separate function for the directive, it allows us to re-use the functionality for the directive without directly using the directive.

64.3.1 Creating Custom Directive Validator

Let's create a custom directive validator for numbers. We are using the NX workspace. In addition, we are using the Razroo reccommended folder structure. A custom directive validator, will go in the common folder for directives.

Generate The Directive

```
ng g lib directive --directory=common
```

Inside of newly created `CommonDirectivesModule`, we will create a folder

```
cd libs/common/directives/src/lib/;
mdkir number;
```

So now let's navigate to our number folder:

```
cd number;
```

and run the appropriate Angular CLI command for generating a directive, and exporting it within our `CommonDirectivesModule`

```
ng g directive number --export
```

This will generate the following output inside of the terminal:

```
CREATE libs/common/directives/src/lib/number/number.directive.spec.ts (224 bytes)
CREATE libs/common/directives/src/lib/number/number.directive.ts (144 bytes)
UPDATE libs/common/directives/src/lib/common-directives.module.ts (493 bytes)
```

Create The Function for Directive

Inside of the folder for our number directive, let's also create a number validator file.

```
touch validator.ts;
touch validator.spec.ts
```

Inside of our `validate.ts` file, we will go ahead and create logic for numbers.

```
import { AbstractControl, Validators, ValidatorFn } from '@angular/forms';

function isPresent(obj: any): boolean {
  return obj !== undefined && obj !== null;
}

export const number: ValidatorFn = (control: AbstractControl): {[key: string]: boolean} => {
  if (isPresent(Validators.required(control))) return null;

  let v: string = control.value;
  return /^(?:-?\d+|-?\d{1,3}(?:,\d{3})+)?(?:\.\d+)?$/.test(v) ? null : {'number': true};
};
```

Listing 64.2: number-validator.ts

Hook In Validator Function to Directive

We will now go ahead and integrate the number function into our directive.

```
import { Directive, forwardRef } from '@angular/core';
import { AbstractControl, NG_VALIDATORS, Validator } from '@angular/forms';
import { number } from './validator';

const NUMBER_VALIDATOR: any = {
  provide: NG_VALIDATORS,
  useExisting: forwardRef(() => NumberDirective),
  multi: true
};

@Directive({
  selector: '[razrooNumber]',
  providers: [NUMBER_VALIDATOR]
})
export class NumberDirective implements Validator {
  validate(c: AbstractControl): {[key: string]: any} {
    return number(c);
  }
}
```

Listing 64.3: number.directive.ts

We now have the ability to use this as a directive within our application. It should be noted that we are doing two unique things within our directive.

1. `Validator` - Our class is implementing `Validator` which is the same internal function used for validating a formControl.

2. We are providing a value called `NUMBER_VALIDATOR`. `NUMBER_VALIDATOR` will cause the `NG_VALIDATOR` `injectionToken`, which is the Angular provided token for custom providers, to use the value of `NumberDirective`. The internal Angular `forwardRef` value is there to make sure that it doesn't error out if no value exists. (More on `forwardRef` in another chapter.)

Hook In Directive to Component Template

Now all we need to do, is hook the directive into the template for our component.

```
1 <mat-form-field [formGroup]="newsletterForm" class="email-field" >
2   <input matInput razrooNumber formControlName="email" placeholder="Your
        E-mail" required>
3 </mat-form-field>
4
5 <ng-container *ngIf="email.invalid && (email.dirty || email.touched)">
6   <mat-error *ngIf="email.errors.number">Not a number</mat-error>
7 </ng-container>
```

Listing 64.4: newsletter.component.html

As you can see, all we had to add to our input is the directive for `razrooNumber`. We can then the appropriate `mat-error` within our application in order to hook into the respective error created for number. The reason it is number, is that within our number validator, we are returning a boolean for number.

64.3.2 A Word on This Approach

If you were to take a look at the documentation, it offers two approaches.

1. Template Driven Custom Validators

2. Reactive Custom Validators

Razroo reccommends this approach for using directives, which is a bit contrary to our general suggestion of using Reactive Forms. It is our understanding, that reactive forms are extremely valuable, because they group the form as one. Making it easily accessible from the Typescript side of things. However, in this particular aspect, of re-using custom validators, re-using functions as a utility function, we feel makes the app more brittle than directives. Primarily, because directives are more explicit.

That being said, with our approach, we are separating the logic of validation from the actual directive. Moving forward if your team, or other teams within your organization would like to use different approaches, they do have the flexibility to do so.

64.4 Cross Field Validation

When I first came across the term "cross field validation", I was a bit confused. I thought of it as a way for different field's validation to be dependent on each other. After reading more, I found that "cross field" validation is exactly that. We are validating our field based on two field's correlation to each other.

For instance, let's say we are working on a finance application. We only want the "Business Loan Amount" input field to show, if yearly gross income minus expenses, exceeds $100,000. We would be able to set up the internal Angular form validators, so that the loan-amount is invalid, if this cross field reference(grosss) is not valid.

THe one point to keep in mind, is that cross field validation is distinct from single form validation in two regards:

1. The validator will be used on two, or more form validators. This means, that we need access to the parent level formControl.

2. The validator will need to be aware of the specific field that it is operating on. Therefore usability will be limited to the app, and most likely will not make sense to be re-usable.

It should be noted, that based on the above, being that cross field validation tends to be component specific, contrary to our reccomendation above of using a directive for a single field validation, we would reccomend using a function. However, as teams are wont to do, they are going to want to have the flexibility to choose how they want to integrate with their app. So we still will be following the separate directive and validate function approach.

64.4.1 Sample Validator Logic

For the sake of brevity, we will not go through the steps we did earlier, re: proper folder structure, directive generation, and respective function. Just one note that I would like to make. We mentioned earlier, that due to cross field validators usually being unique to specific business logic, we prefer the use reactive form validator functions. Therefore, as opposed to prior single field validators going in the `common/directives` folder, multiple form validators will go in the app specific folder (e.g. `razroo/common/directives`).

However, we will go into the validator logic required for multiple `formControl` values.

Creating the Service

```
import { Injectable } from '@angular/core';
import { FormGroup, ValidationErrors, ValidatorFn } from '@angular/forms';

@Injectable({
  providedIn: 'root'
})
export class LoanAmountValidatorService {

  constructor() { }

  identityRevealedValidator: ValidatorFn = (formGroup: FormGroup): ValidationErrors | null => {
    const income = formGroup.get('income');
    const expenses = formGroup.get('expenses');

    return income && expenses && income.value - expenses.value > 100000 ? { 'loanAmount': true } : null;
  };
}
```

Listing 64.5: loan-amount-validator.ts

As we can see in the above code, our logic is now tapping into the entire `formGroup` control. We are:

1. Targeting every field that we need.

2. Creating logic, based on those two fields.

Cross form validation logic integration with our directive, is straightforward and very similar to how single form validation works:

```
import { Directive } from '@angular/core';
import { AbstractControl, NG_VALIDATORS, ValidationErrors, Validator } from '@angular/forms';
import { LoanAmountValidatorService } from './loan-amount-validator.service';

@Directive({
  selector: '[razrooLoanAmount]',
  providers: [{ provide: NG_VALIDATORS, useExisting: LoanAmountDirective, multi: true }]
})
export class LoanAmountDirective implements Validator {
  constructor(private loanAmountValidatorService: LoanAmountValidatorService) {}

  validate(control: AbstractControl): ValidationErrors {
    return this.loanAmountValidatorService.loanAmountValidator(control)
  }

```

```
16 }
```

Listing 64.6: loan-amount.directive.ts

Creating the Directive

Here we are just pulling in the service, and tucking it into our validate function. So, now using our prefferred approach of reactive form validators, for cross field validation, integrating it, would be as simple as this:

```
1  constructor(private fb: FormBuilder,
2              private loanAmountValidatorService:
                   LoanAmountValidatorService) { }
3
4  ngOnInit() {
5    this.newsletterForm = this.fb.group({
6      email: ['', [Validators.required, Validators.email]],
7    }, {validators: this.loanAmountValidatorService.loanAmountValidator});
8  }
```

Listing 64.7: finance-calculator.component.ts

Creating the error

Integrating the error within our app, is exactly as we would have done for a singular field.

```
1  <ng-container *ngIf="finance.invalid && (finance.dirty || finance.
       touched)">
2    <mat-error *ngIf="finance.errors.loanAmount">Calculator</mat-error>
3  </ng-container>
```

Listing 64.8: finance-calculator.component.html

64.5 Async Validation

An async validator, is a validator which returns an observable(promises work too, but we do not reccomend using a promise within an Angular setting). One other important note, is that the Observable must be finite i.e. end. So adding something like `first` to the async directive more than works.

A classic example with async validation is to see if a username is taken already. In order to give the user instant feedback, we can make an http request everytime the user types in the input. A validator is a one to one relationship, and is not meant to have a lifecycle beyond that of the initial validation. Using state for something like this would be overkill. However,

we do use GraphQL within our application. So, we are going to return an observable, and call first on it.

64.5.1 Integrating Service with Component for Async Validation

The nature of async validators is they need to use the backend to operate. Therefore using a directive is out of the question. It would be too brittle, and a hack to make work. I am not going to discuss the code behind service actually making request. Reason is, that use cases wherein we would use asynchronous validation, is usually in an enterprise setting, and it's one team building it for the entire organization. I feel like the use case for something like this so rare, albeit useful when the time arises. I want to discuss it more conceptually, so you know when to use something like this.

So that being said, let's assume that we have a service that returns a finite observable. It makes a request and determines whether, or not the user email is already taken.

```
ngOnInit() {
  this.myForm = this.fb.group({
    name: ['', Validators.required],
    email: [
      '',
      [Validators.required, Validators.email],
      ValidateEmailNotTaken.createValidator(this.signupService)
    ]
  });
}
```

Listing 64.9: user-signup.component.ts

Within our template, we would do the following:

```
<form [formGroup]="myForm">
<input type="text" formControlName="name">
<input type="email" formControlName="email">

<!-- Other related errors go here-->

<div *ngIf=" =myForm.get('email').errors && myForm.get('email').errors.
    emailUnavailable">
  This email is already taken.
</div>
</form>
```

Listing 64.10: user-signup.component.html

64.6 Performance Concerns

All validators are run after every form value change. As in, when an input value is changed, the validator will be run after every letter/number is added. Sychronous validators, which are not dependent on the backend, generally do not suffer from performance issues in this regard. However, when doing something like making an http request after every time a letter is clicked, it can be expensive. There is a general reccomended approach of making validators run on `blur`, or `submit` instead. So, let's say in our app, we want the http request to only be made if the user clicks off of the input, we would do something like the following:

```
ngOnInit() {
  this.myForm = this.fb.group({
    name: ['', Validators.required],
    email: [
      '',
      [Validators.required, Validators.email],
      ValidateEmailNotTaken.createValidator(this.signupService),
      {updateOn: 'blur'}
    ]
  });
}
```

Listing 64.11: Updated user-signup.component.ts

64.7 A Final Note on Form Validators

Form validation is an integral part of any application. The wonderful way about how Angular does things really shine with the way they approach form validation. Form validation is particularly difficult, because due to it's repetitive nature, and existing outside of component logic. By following the `common/directive` architecture for single field validation, and the `<app>/common/service/validators` architecture for cross field validation, your organization and app, will be in a very good place to scale.

65 Angular Elements - Introduction

Angular elements is a feature that's not talked about enough and have been around since version 6. It's been more than a year since the original release and works perfectly as a method to modularize your application or add independent components to a different app without the need to compile an Angular application.

In a way, Angular Elements is an ode to the portability factor that harks back to the original days of Angular.js. What elements essentially lets you do is build stand-alone components and allow you to export it into a single JavaScript file.

It's a fantastic feature -âĂŁespecially for micro-frontends and transitional applications that are working with multiple and possibly legacy technologies.

65.1 So how does it work?

Angular Elements is an Angular feature that lets you create new DOM elements. As we all know, JavaScript hooks onto the DOM to make visual changes and instigate change based on our configurations and settings. Angular is a framework that lets us do this quickly and easily.

However, there are a finite number of preconfigured DOM elements available, with the extension of custom elements coming in with HTML5. This means that your tags aren't limited to the h1 and p tags we're all used to seeing. Custom elements allow you to create your own HTML tags, based on your application's needs.

Angular leverages this feature through Elements by allowing us to create our own custom elements and hook into it via our compiled Angular script. So at the end of your coding session, your code could potentially look something like this:

```
<script src="angular-cart.js" />
<custom-cart></custom-cart>
```

This little code snippet lives independently from the rest of your application, making it highly portable and reusable. You could drop it into a static HTML page and or a pre-existing codebase that may be using a different or older framework or library.

65.2 So how do you go about creating an Angular Element?

1. install Angular elements via the CLI using the following command:

   ```
   npm i @angular/elements --save
   ```

2. Install the custom elements polyfill. A polyfill is essentially a piece of code that allows you to access APIs that the browsers are expected to provide. However, sometimes these features and functionality arenâĂŹt readily available, so a polyfill fills in the missing content and ensures that your application continues to work as expected.

   This means that your application becomes backward compatible, to a certain degree, for older browsers.

   ```
   npm i @webcomponents/custom-elements --save
   ```

   Once youâĂŹve done this, go into the poyfills.ts file, located in the src folder of your Angular application and import it into the file.

   ```
   import '@webcomponents/custom-elements/custom-elements.min';
   ```

3. Build your component as per usual and then convert it into an Angular custom element by importing `Injector` from `@angular/core` and `createCustomElement` from `@angular/elements`

   Because this component is not declared or used by the router, you need to add your component to an `entryComponents` array under `@NgModule`.

   Then go down to your `AppModule` class and add the injector into the constructor and then bootstrap manually via `ngDoBootstrap()`

   Pass your Angular component you want to turn into an Angular Element to the `createCustomElement()` method. This method will help create a bridge that will convert your Angular code into native JavaScript code that will work with DOM APIs.

   The last thing you need to do is register it via

   ```
   customElements.define('your-custom-element-name-here',
   theCustomElementYouJustCreatedInThePreviousStep)
   ```

At the end of it all, your code should look something like this:

```
...
import {NgModule, Injector } from '@angular/core';
import {createCustomElement} from '@angular/elements';
...
@NgModule({
...
entryComponents: [ CustomCartComponent ]
})
...
export class AppModule {
constructor(private injector: Injector){}
ngDoBoostrap(){
    const yourElement = createCustomElement( CustomCartComponent,
     { injector: this.injector});
    customElements.define('custom-cart', yourElement);
  }
}
```

You can now use this custom-cart component by writing `<custom-cart></custom-cart>` in your HTML. If you npm run it, you should be able to run the component independently.

Now all you have to do is run `ng build` to export it as a single JavaScript file.

65.3 Why would you use Angular Elements in the first place?

Unless you're working on a brand new project, possibly for a startup, you're going to encounter legacy code. There's no escape from this reality.

Many businesses often opt for a transitional approach towards upgrades and new features, meaning that you're going to need a way for your front end code to fit in nicely with the rest of the application.

Angular elements give developers this opportunity and present a clean solution to a very common problem. Your feature code becomes extremely modular and containerized in a way that truly isolates it from the other code.

This creates a level of independence in your application building process and modularizes your team's workflow.

Angular elements are also particularly suitable for delivering dynamic applications that are made up of many complex componentsâĂŁ-âĂŁsuch as a dashboard that may require independent deployments for each part. When you architecture your shell page and fill it with custom Angular elements, it gives you the ability to create deployments that are separate from another, reducing the potential impact and isolating issues if something went wrong.

In a good way, Angular Elements can help you transition your legacy app seamlessly into the future within creating contingent effects on your current code. If you have an Angular.js app that needs to be upgraded, Angular Elements is also a good way to go without creating conflicts in your current deployed code.

Using Angular Elements can also move your data methods away from being stored in the frontend and into a space that has more permanence through APIs that connect to a backend.

65.4 How to get rid of zone.js

zone.js is how Angular makes change detection possible. It's an external dependency that creates execution contexts, especially for async tasks. It's also the thing that makes binding possible, along with any UI changes we visually see when something changes.

However, zone.js has its own issues such as the occasional break in *ngFor loops. To give yourself more power and control over how and when your code responds to change, you can turn off zone.js manually at the global level in the main.ts file.

```
platformBorwserDynamic().bootstrapModule(AppModule, { ngZone: 'noop'});
```

Or if you just want to deactivate it for a particular component only, you can do so inside the `@Component` decorator.

```
@Component({
  ...
  changeDetection: ChangeDectionStrategy.OnPush
})
```

Now that you have zones turned off, you will need to manually tell Angular when to render data. Import `ChangeDetectorRef` from `@angular/core` and be sure to declare it inside your constructor. Now all you have to do is call the `detectChanges()` method in order to tell your application that a change has occurred.

65.5 Parting words

Angular Elements is one of those things that aren't talked about enough - but should be. It's a tool that increases Angular's ability to adapt to different environments in a succinctly effective manner.

Elements also allow for a lightweight way to use Angular code without the need to export an entire application, making effective micro-front ends and transitional applications possible, especially in a space where there is talk about Angular being too bloated to work with.

But it's not and we've come a long way since the original days of Angular 2's original release. Elements let you create independent features with the scaffold of Angular's structures and allow you to export in a manner that is highly modular.

66 Using React with Angular

One of the singular most fustrating parts of front end development, is the fact that frameworks change quite often as time goes on. In addition, different frameworks have different usages. Two of the currently most profitable front end "ecosystems" are Angular + React. Sometimes, having something only available in Angular can be cumbersome. Being able to break into the React ecosystem, for instance, can be extremely valuable.

Microsoft came across the use case of needing to have React within their Angular applications. They created the open source library called angular-react. It allows for the ability to use React in Angular. Let's discuss that.

66.1 Reasons to use React in Angular?

1. Internally one team built a very large component in React. Your Angular team would now like to use this React component, with limited overhead.

2. Industry trends push your team to migrate towards React. For instance, one time I was approached by a CTO of a company who mentioned he is coming across more React developers in the market. They might not architecting their React applications as they need, but that might be the benefit of React, for a startup. Wherein it doesn't require as much overhead. For their business, a lesser architected application might be beneficial, because their getting to the market and earning a profit is integral given the funding they have.

3. A very large component library is only availabe in React, and not in Angular. For instance, in the case of Microsoft, who actually created the open source library `angular-react`, they needed to use Office UI Fabric, which was only available in React.

66.1.1 Sidebar - When Not to Use React in Angular

Intuitively, one can look at the ability to use React in Angular, and think great! I'll use the best of both frameworks!!! Ok, well hold on their Cowboy, or Cowgirl. It might not be the best idea to create a Frankenstein of an Angular + React application. The two ecosystems can clash with each other. Between the way data is passed around, rendering, libraries used

for state management etc. However, if you find yourself gravitating towards one of the three reasons above, using React in Angular, can be very beneficial.

66.2 Performance Concerns of React in Angular

Obviously rendering React elements inside of Angular seems like a little bit too go to be true, right? Well this is a partially correct assumption. Let's dissect the logic within the `angular-react`. It creates a layer around React methods, so that they are understandable by Angular. It then will call React within Angular. The main fault with this approach, is there will be two separate rendering engines at play. So, if you attempt to have them both render at the same time on the same component, from the browser's side of things, it can cause them to trip over each other. The `angular-react` team has created a demo proving this point here.

Another performance concern, as exemplified here, is that there is one main difference between the two. With regards to React within Angular, the rendering time, as well as paiting is significantly higher. It will be visibilty noticable that it will take more time to show the components when using React inside of Angular.

66.3 Nature of the Library

I always try to do my background research on a library before I go ahead and use it within my application. In this one, there are two red flags:

1. They created this library to use Office UI Fabric. I.e. rendering of singular components within an Angular setting.

2. It is a product team on Microsoft, that are the main maintainers for this library. Given the limited use case of this library, expect it to not be optimized for maximum performance, or when multiple components are needed. Honest, a React Elements approach makes more sense in this scenario.

In most applications I've worked on, the use case for `angular-react` is very limited. Proceed to the next chapter on custom web components for a better long lasting solution.

67 Custom Web Components

Really, what makes a framework, a framework, is it's ability to tie together the different parts of the application into something larger. This includes services for data requests, files for interfaces/type checking, state management, rendering engine, and the ability to pass inputs and outputs to each other.

One very intelligent piece of architecture that will be getting more press as time goes on, is creating custom web components within you application. This allows for your dumb components to be independent of framework, and to be re-used wherever.

I would promote this practice by default to use in any application that you work on. All dumb component should be created as a custom web component, so any team can modify it.

67.1 Before We Get To Nx

Before we get to Nx, I would like to discuss one thing that really at this point bothers me. Nx controls too much of my code. If the team at Nrwl decides to take the direction of paid code, or documentation is lacking, I am screwed over. It really seems like a large price to pay, for tooling that can be created by my own team.

So while I am going to reccomend using Nx, and I know the people at Nrwl personally. I actually love them to death. Jeff and Viktor are some of my biggest personal inspirations. I really want them to succeed, and that their company balloons into a giant. However, I can't help but feel it irresponsible so that I don't through the rabbit hole too much. I remember sitting at a company with a very large engineering team, and the fact they weren't educating their own team members on Webpack, Bazel, and creating transparent schematics, almost seemed like a travesty. You are pretty much gaurenteeing that your company will have to use them as consultants to keep you app ticking, if you come across some quirk.

For me, the eureka moment really came when I started using Web Components. I was like at this point, ok, fine. I will use your app to scaffold ngrx, and generate the initial workspace. However, when I have to use you to actually build my app, that is when it becomes problematic.

So, suffice to say, I am going to attempt to create this solution on my own as well, for this one.

67.2 Using Nx

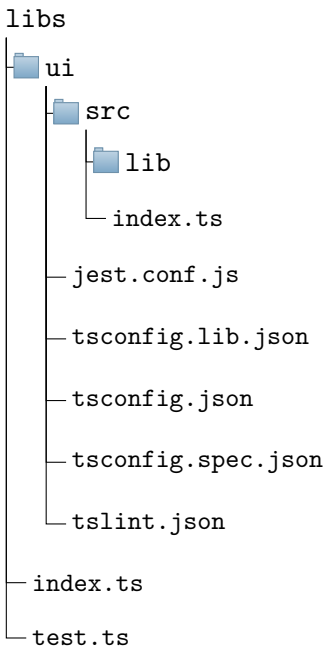
Granted you are working within the Nx workspace, which once again, I highly reccomend, but please do use with caution.

67.2.1 Scaffolding Web Components Lib

Within Nrwl Nx there is the ability to create a none framework library, which will include all of our framework agnostic code.

```
1 ng g lib ui --framework=none
```

This will generate a `ui` directory structure, that will something like the folliwing, within your application:



67.2.2 Create An elements.ts File and Export

Inside of your lib folder, let's create a data-table component.

```
cd libs/ui/src/lib;
mkdir data-table;
```

Inside of that data table, let's go ahead and create a custom web component.

```
export class GreetingElement extends HTMLElement {
  public static observedAttributes = ['title'];

  attributeChangedCallback() {
    this.innerHTML = `<h1>Welcome to ${this.title}!</h1>`;
  }
}

customElements.define('happynrwl-greeting', GreetingElement);
```

Listing 67.1: custom web component

A custom web component somewhat interacts like a regular Javascript component and styling is very similar in that regard. Here you can see us using the inner.html code, and other components are similar in this regard.

67.2.3 Reexport in index.ts file

You will want to go ahead and re-export it in your web-element component. This will simplify that way things work, and allow you to re-use within your app.

67.3 General Architecture of Inserting Web Element

67.3.1 Changing Target

Because custom web elements are a relativley new technology, it will require the target to be changed to ES2015.

67.3.2 Importing library

Your app will require importing the library into your web application.

67.3.3 Tell Framework To Take Chill Pill

Every framework will have a specific schema towards it's web application. Custom Web Elements are outside of the regular schema of every framework. So you will have to use the framework's preffered method of "taking a chill pill". I.e. saying the schema for custom web elements is cool, and should not be errored out.

67.3.4 Actually Using Element

Actually going ahead and using element similar to how you would use a component in any regular application. Now that we have specified these four steps, let's actually bring them to our application.

67.4 Inserting component in Angular App

67.4.1 Change Target

In our `tsconfig.json` we will go ahead and change the output target to `es2015`. Custom web elements are a relatively new technology, and require es2015 by default to be bundled as part of the application.

67.4.2 Change Target

Within our Angular application we will tell the schema that it should allow for custom elements i.e. `CUSTOM_ELEMENTS_SCHEMA`. This is primarily so it will allow for custom elements to be used within the app.

67.4.3 Insert Component in Angualr App

Inserting a web component within an Angular app, is done in exactly the same fashion that a regular Angular component would be inserted.

67.5 Inserting component in React App

67.5.1 Update Target

Change the target in target in React to `es2015`.

67.5.2 Importing Library

We go ahead and import library within our application.

67.5.3 Adding Intrisic Types

This step is somewhat different within an Angular application. Instead of adding to the schema, we will go ahead and create an `intrisic.d.ts` file.

67.5.4 Actually Using ELement

We then go ahead include this element within our application.

68 Micro Frontends

Micro frontends is definitely a "buzzword". I consider it most synonymous with the buzzword "the cloud". Simply it means breaking up the application into smaller more manageable parts. So, you might be asking yourself, what makes this different than simply architecting an application well? The answer to that,is that it's not so much folder/file directory, and how you dissect your components, but rather flexibility in bundling and deployment. By allowing for flexible bundling and deployment within app. This accomplishes three things:

1. Increases testability

2. Increases re-usability

3. Ability to select multiple frameworks

68.1 Micro Frontends are actually already in the wild!

The truth is that micro-frontends have been around for a really long time. Doing something like this is possible in vanilla Javacript is possible, by simply 1separating micro applications into different JS bundles. When the page goes to a specific route, it will call this unique JS bundle. Incredible simple in practice. I've also seen application get somewhat fancy, wherein routing was generated by the backend, via devops. The js bundle would be given a different id everytime, and therefore made sure prior js version would be removed. It was a relatively painless way for the JS team to create micro front ends.

However, I'm not sure if anyone on the team has ever coined this as "micro frontends". In addition, doing something like this was a real pain, and the flexibility was really only there on a single route.

So, why is there is now a push for micro frontends architecture, and why is it now coming into the forefront? For this, we have to understand a bit more of the history of web applications.

68.2 History of Web Applications

Let's dive into that really quickly.

68.2.1 Monolith Applications

In the earlier days, when front ends were relatively new, applications were generally built in a monolith fashion. Applications would be built with a single Java WAR file. Your backend code, and and front end code, were contained in a single file.

68.2.2 Separating Backend and Frontend

As time went on, backend and frontend would be separated. Html files, and javascript would be separated. The appropriate data would be inserted using php, or perl tokens, or the like. Every single api was in the php application, and it would be used, by simply inserting the php data needed for that site. However, the whole of the backend would be in a single php application.

68.2.3 Creating Micoservices

As time went on, and mobile devices became more popular(i.e. apps), it became more cost efficient for companies to create micro services. Primarily, so that multiple teams could work on it. It's important to note that a micro service is not a silver bullet, and won't neccesarily increase the efficiency, and maintainability of your application. However, it has it's place, and greatly resolves scaling issues. So mutliple backend applications would be built by either different teams, or singular teams. These would then be able to be deployed in a REST Service fashion, allowing different clients, such as mobile, or desktop, to consule.

68.2.4 Applying Microservice Architecture to Microfrontends

However, as front ends are growing very large in size, and it is becoming increasinly easy to do so, due to the frameworks readily available, large teams are required to work on a single application. It's much easier to break up those teams, give them a singular component library to work off of, and then have them work in their own clusters.

However, in my personal opinion, micro-frontends are not ready yet for prime time. There is no open source framework that I can point to, that would allow for this to be easily done.

Or rather, this can be done, but having a development tool that would allow for this to be done,

Within frameworks yes, and that is a large step. However, it can be a bit awkward to try and shove on bundle of a particular framework into another. In particular, because state management, should ideally be global. Doing this across multiple frameworks, is very difficult at this point in time.

68.3 Illustration in Code

A simple way of illustrating how we can create micoservices within Angular, would be as follows. Let's say that we have an application. One of them is a pdf viewer application. The other is the application for analytics. We want our teams to be able to build applications independent of each other.

68.3.1 Adding The Tools We Need

```
ng add @angular/elements
ng add ngx-build-plus
```

This adds Angular Elements to our app. In addition, we are going to be adding a tool called `ngx-build-plus` which allows us to extend the default Angular CLI without injecting. That way, we can continue adding default Angular CLI builds to our app.

68.3.2 General Strategy

The general strategy is like this. We are going to create miniature using Angular Elements. We are going to then insert these bundled applications into our main Angular application.

We then will go ahead and route to those specific applications within our application. Ideally using the Angular router. Particularly, because we want it to be robust enough, so that we can go ahead and create router-outlets.

68.4 Difference Of Approach Using Custom Web Components

The difference between the approach we are using here, and how we approached using custom web components is as follows. When creating micro-frontends, we don't just need

access to our custom web components, however, we actually want to build them separately. Once again we do this so that:

1. Increase testability
2. Increase re-usability
3. Ability to select multiple frameworks for these specific apps

68.5 Router Dillema

The main dillema within our Angular applications, is the fact that angular routers consist of two parts:

1. forRoot
2. forChild

Within our web components, or our Angular Elements rather, our forRoot will be reserved for our root component specifically. If we want to use web components within our app, forChild won't work either, because these are all micro front ends. In fact using something like Angular Router for our application, is a bit of an anti-pattern, because the whole point is to create a framework agnostic architecture. So, what should we use within our Angular application.

68.6 Micro Architecture - Key Design Principles

I personally feel that micro architecture has officially made it's way over to front end architecture. This means that it is now important for a UI Engineer, to be aware of what constitutes micro architecture. Martin Fowler, one of my favorites, has an excellent article on what the characteristics of micro-service architecture are.

[I will admit that I'm a bit new to this micro-service architecture, as I'm starting to break more into backend architecture. However, when I did a deep dive, I started to look for something similar within front end. That is sort what perked my interest, and why I am now writing about it.]

I would like to go through the points mentioned in this article and others I've found as well. However, for the sake of efficiency, only apply those concepts within the confines of micro frontend architecture.

68.7 Single Responsibility Principle

Robert C. Martin, otherwise known as "Uncle Bob", is best known creating the Agile Manifesto. In addition, he is well known for creating/promoting the SOLID principles. One of the SOLID principles, in fact the first letter stands for the single responsibility principle. The single responsibility principle states:

> "Gather together those things that change for the same reason, and separate those things that change for different reasons."

Micro front end architecture is a natural extension of this concept. It takes front ends which serve different purposes, and makes it so that they are developed, deployed, and maintained independently. However, these services can still communicate with each other, thereby gathering those things that change for the same reason.

68.8 Micro Frontend Architecture Design Principles

> "An architectural style where independently deliverable frontend applications are composed into a greater whole"

[1]

[1] https://martinfowler.com/articles/micro-frontends.html

Micro Frontend technology is still being developed, and I think there is still a lot of confusion around the benefits of such. I would imagine there is going to be a happy medium moving forward. However, right now I would like to present design principles as I view them, with the best tech currently available.

68.8.1 Business Centric

Each front end address a particular business need of you application. For instance, let's say that we have an online book store, and that we would like:

1. Shopping Cart
2. Single Book Page
3. Multiple Book Page
4. Header
5. Footer

Each one of these elements, serves an all around different business purpose within our application. The shopping cart, to give people an option to see everything they've bought so far. The multiple book page, to give the user the ability to see all books we are going to sell. The single book page, to get in depth detail of the book they want to buy. The header, to allow people to properly navigate through our site. The footer to give people an abstract of our information.

So, based on the fact that each one of these elements contains a different business use, we would build them into their own application(,or once again it's own library, as we will do). [2]

68.9 Autonomous Features

An autonomous feature (in software) means something that can control itself independently. So an autonomous feature here means a front end that can:

1. Change, test and deploy independently
2. Resiliency - If one area of application faults, allows us to degrade just one section of the application.

[2] martinfowler.com/articles/microservices.html#OrganizedAroundBusinessCapabilities

3. Observable - Centralized logging, and monitoring, we can see what our individual micro front ends are doing.

without breaking other front ends. The immediate obvious benefit to this, is allowing teams to work independently of each other.

68.10 Front End Framework Agnostic

In practice, autonomous features requires this one distinct feature, which is that applications are front end framework agnostic. From a practical perspective, this means:

1. Code isolation - All micro frontends are built as a separate library. We use single-spa to accomplish mounting separate front end frameworks. In addition, using Nrwl Nx to keep libs separate, and build separately.

2. Base app - All of our micro frontends are going to need a base app to hook all of our frontends into each other. However, ideally being able to use the browser directly instead of a base app, is ideal.

3. Front End API - Within the context of an enterprise Angular front end application. This means allowing our micro front ends to communicate with eacth other through:

    a) The router

    b) Browser Events(we reccomend using eev)

68.11 Super Resilient

Being that we are going to be hooking in potentially multiple frameworks, that can have an impact on the initial load time of our site. Being able to have a visible site, without data, or Javascript being loaded is key. Using something like Ghost Elements, is crucial, even more so than a single page application. Integrating universal rendering, or static site generation is also a good idea, just to make sure the ability of it loading is faster, granted the cohesive, performance heavy nature of bundling.

68.12 Team Ownership

This part can be a bit controversial, and I can definitely see some companies having opposition to this, being that the ecosystem isn't yet fully ready for something like this. However, the idea is, is that now we have a micro frontend, we can actually tie an entire "slice" of our product, end to end, by one team. This means that a single member, will be expected to contribute to front end, backend and the database. Arguably, this is beneficial for four reasons:

1. Simplifies communication. No need to talk to database, or dev-ops, because they are the dev-ops.

2. Simplifies co-ordination. No reason to talk to backend, or database specialist, because they are the specialist.

3. Changes happen quicker - Due to expeditious communication.

4. Improved performance - The developer get's to see how optimization they made on database, or back end, affects the performance on the front end.

68.13 Final Thoughts on Micro Frontends

The most humorous and simultaneously enlightening piece I found while putting it all together, is a Tweet by Dan Abramov. It's a very brave tweet, something along the lines of, "I don't understand micro-frontends". Two of my favorite open sourcers on this topic, Joel Denning + Michael Greers, respond something like the following. It allows teams to be:

1. Autonomous(work on their own without co-ordination of other teams)

2. Full stack

The above two are the single greatest reasons given by both and them, and are synonymous with the reasons for micro-services. However, as I started to think about it more, not quite.

The difference for me, in my very humble opinion between the two(micro-services vs micro-frontends), is that the tech + tools available within a singular framework(such as Angular) is arguably easier to work with. The micro front end ecosystem does not have a state management that can be used across all frameworks, or dev tools such as Nrwl Nx. It's going to require a very large overhead from the team's side of things. For me, it doesn't make sense at this point in time to implement. Especially if this does indeed start to become an issue, micro frontends can start to be built out, at this point in time.

Once the tool ecosystem becomes more sophisticated for micro frontends, this is something I would start reccomending. Right now, I don't think it's fair to have teams work full stack, and learn multiple frameworks, unless the company is willing to invest in something like that.

69 Static Site Generation

70 Smart Vs Dumb Components

In any UI framework, following a smart component, and dumb component architecture, is going to be a good idea with regards to re-usability. It will completely change the way(if it hasn't already), the way you go ahead and build your components. It is one of those design patterns that you will ineviabtly learn how to do on your own, but why take the hard way(i.e. reading this chapter) when you have chance to just read this chapter. If you are already familiar, feel free to keep reading, I worked hard to make sure there is something to learn as well.

70.1 A Dumb Component - Defined

A dumb component many times can be thought of as a child component. Think of it as this component is going to be re-used is many different places. How can we make it's logic as generic enough so that can happen? Simply put, the dumb component will receive it's data from it's parent component. Wherein the parent component will be responsible for retreiving the data. The question then becomes framework specific, how can we pass events up from the dumb component, to it's parent smart component. In addition, how can we pass down data from the parent component to the child component. So the only two concerns when creating a dumb component is:

1. Event Binding

2. Propery Binding

70.2 A Smart Component - Defined

A smart component inversely, will be responsible for creating the data, to be passed down to the dumb component. In addition, it will hook into the events for the specific dumb component, and make sure to make data calls at that time as well.

70.3 Where does State Come In?

One of the grayer areas when it comes to dumb vs. smart components, is state management. I personally have found putting a re-usable state in a dumb component to be extremely useful. State is to be made re-usable by keeping all unique sets of data nested one level deep. This is done by making the storeSelectName dynamic. We will discuss this in detail in another chapter.

70.4 Creating a Dumb Component in Practice

In an Angular setting there are two decorator functions that the framework offers out of the box that will help in this regard. They are @Input and @Output.

70.5 @Input in Detail

@Input if you are not already familiar, is Angular's way of passing down data from parent to child. So just a quick example of how data would be passed down from a smart component to a dumb component would be as follows:

```
export class DataTable {
  @Input() dataSource: DataTableData;
}
```

```
<div *ngFor="data in DataCollection">
  <div class="DataTable__row">{{data}}</div>
</div>
```

In our parent component's html, we would do something as the following:

```
<data-table [dataSource]="UserData"></data-table>
```

70.5.1 Setting and Getting an @Input

An important piece of architecture, is when getting back data it is possible to manipulate the data before actually being put within the component itself. I will not go into specifics here, but code can be seen by folliwing code specified in repo.

70.6 @Output in Detail

@Output is an observable property. It is usually always coupled with an event emitter. It's intended use to pick up on an even that happens within a dumb component, and have actual logic happen within the smart, parent component. An example would be as follows:

```
visible: boolean = true;
@Output() open: EventEmitter<any> = new EventEmitter();
@Output() close: EventEmitter<any> = new EventEmitter();

toggle() {
  this.visible = !this.visible;
  if (this.visible) {
    this.open.emit(null);
  } else {
    this.close.emit(null);
  }
}
```

Then in your html, it will look something like the following:

```
<data-table (open)="handleOpen()" (close)="handleClose()"></data-table>
```

70.7 Wrapping Up

That would be it. The above will allow you to architect creating a smart vs dumb component.

71 Error Handling

This chapter is a bit different then the rest of the chapters in the book. Many technologies by simply knowing about them, you are heads and shoulders above the rest of them, simply by knowing about them. For instance, if you were to know ngrx/store, ngrx/entities, apollo/graphql, that is more than enough to prod you in the right direction with regards to fantatic architecture. However, one thing that I've noticed by working in many different apps, that tends to fly under the radar, is error handling.

71.1 No Error Handling is Not Disatrious

Angular is a very robust ecosystem. By that, I mean that many of the proper technologies to use within one's app already have error handling baked into it. So, if one does not use error handling, then it is not completely disastrious. Many of the technologies that you are using will have error handling.

71.2 The Benefits of Error Handling

1. Send User Errors to Server

2. Allow Errors to be more specific

71.3 Server Specific Angular Errors

In any Angular app that is data centric, a large part of your errors are due to server side errors. Many times the server will return the proper error. However, the argument can be made for server side error reporting to be more explicit.

71.4 Client Side Error Reporting

Go into depth as to how we can potentially do error reporting client side.

72 Http Interceptors

An http interceptor, is a way to intercept an HTTP request, before passing them along. Some of the more popular reasons of doing something like this, include Authentication, and adding default Headers to requests. I would like to add scenarios for an Angular dev to be aware of, wherein they should apply interceptors. In addition, I would like to reccomend the use of using Apollo Client middleware over Angular's Http Interceptors, as that is Razroo's reccomendation. Third, I would like to bring in code examples, so that it will be easier for the dev reading this to get up and running.

72.1 Dissecting Two Ways of Creating Interceptors

Within our architecture we assume that your backend team will be using GraphQL. However, this is something that is beyond you as a UI Engineer. As a result, we give a head nod towards there being multiple apis, including those using a regular RestSvc. This translates to:

1. Angular's native `HttpInterceptor`

2. Apollo Client / Apollo Link

72.2 Should We Even Use Apollo Link At All?

One of the popular questions people ask, is should even be using apollo link at all. HTTP Interceptor is used under the hood for Apollo Client. The truth is, this is a fundemental question that can be applied in many similar scenarios. We are using a framework. One piece of it that it offers, is not required by our application. Should we use it, or should we not?

1. Are other teams using our framework, and if not will they have a chance to use our tech?

2. Do we ever expect to use any framework beyond the one we are using(in our scenario Apollo Client)?

3. If unexpectedly, which it always is. If unexpectedly, we end up changing over our framework, how difficult will it be to create this code once again?

4. Ease of use for the framework, vs using native code.

    a) Documentation better for one, over the other?

    b) Integration for this particular aspect of framework easier granted already using other pieces of framework?

Using the above back and forth, it would make sense to use Apollo Client's native functionality for middleware. To use HTTP Interceptors, and then move over to Apollo Client for everything else, such as cache, makes maintenance more cumbersome. For that reason Razroo reccomends using Apollo Link for middleware.

72.3 Understanding Interceptors In General

An Interceptor will generally take in the current outgoing request, and pass in the next interceptor. Alternatively, it can transform the response stream it's self.

72.4 Example of Interceptor using HttpInterceptor

The following is a great example of how interceptors work. Let's say that we want to console out an error whenever it happens. In addition, we want to set up different errors based on how they happened. Something simple like the above can setup error handling notifications across the site for all users.

```
1  return next.handle(req).pipe(
2    tap((event: HttpEvent<any>) => {
3    if (event instanceof HttpResponse && event.status === 400) {
4      this.dialog.error("There was an error trying to make your request.
5      If this continues to persists, please e-mail 401@razroo.com");
6    }
7  })
8  );
```

72.5 Example of Interceptor using Apollo Client

Apollo Client out of the box offers out of the box middleware to intercept an http request. Generally that this means, that there will be a series of apollo functions in your lib folder.

```
// authContext to set Authorization token for every request sent from client
const authContext = setContext(async (request, previousContext) => {
  // Getting the token from the session service
  const token = await this.session.getToken();

  // return {} if token is not set yet
  if(!token) {
    return {}
  }

  // Set Authorization headers with token
  return {
    headers: {Authorization: `Bearer ${token}`}
  }
});
```

Using apolloLink, we will combine all of the middleware we have created throughout the `module.ts` file.

```
// creating the conditional link for http and ws requests
const link = split(({query}) => {
  const { kind, operation } = getMainDefinition(query);
  return kind === 'OperationDefinition' && operation === 'subscription';
}, ws, ApolloLink.from([authContext, error, afterwareLink, http]));
```

While in our code, for purposes of demonstration, we have only included the chapter that includes the link. Other links are bundled together. We can then include this general link in our apollo `app.module.ts` file.

```
// creating the final Apollo client link with all the parameters
apollo.create({
  link: link,
  cache: new InMemoryCache(),
  defaultOptions: {
    query: {
      fetchPolicy: 'network-only'
    }
  }
});
```

In the above, we are including our link in the apollo create. This makes it, so that all our middleware intercepts our apollo requests before they happen.

72.6 Folder/File Structure for HTTP Interceptors

Regardless, if using Apollo, or HttpInterceptors within your application, it would make sense to create a special lib folder for apollo within common. Within the special folder for apollo, we can create a series of link functions and import them within the `app.module.ts` file.

72.6.1 Authentication

Starting from what will be time, and time again, the most single most important piece of your application. Authentication, if not already aware, is the process of identifying an individual usually based on a username and a password. There are certain situations wherein intercepting the http request before it is made, is required for authentication:

1. Add Bearer Token - The word "Bearer" in "Bearer Token" is to be understood as "give access to the bearer of this token". It is a token generated by the server on initial login, and is to be used for every protected request within the app.

2. Refresh Token - Refresh tokens are used when the original Bearer Token expires, and new ones are to be issued. [1]

3. Redirect To Login Page - This will be needed if the Bearer Token expired, and we need the user to go back to the login page to retrieve a brand new bearer token.

An old co-worker of mine, Sam Severance, once told a funny story, how we was once working on the couch and his wife stopped by and asked him what he was working on. He told her, "Nothing too much, just authentication". She said, "oh what's that?". He told her, "Oh, it's just something that makes programmers run around like chickens with their heads cut off". I always thought it was a funny one.

I guess the point behind that story, is that authentication is such an important piece of an application, and end's up being used in every request. It therefore becomes:

1. Relatively difficult to manage.

2. High cost to application if software is mis-managed.

In addition, every time authentication is worked on for the first time in an application, it tends to be specific to framework, and technology within framework you are working on.

72.7 Other Interceptors to Be Aware Of

There are many other interceptors to be aware of. It is important to note, that Authentication is the main one. It is also possible to set up global errors across the site based on different errors codes.

[1] Look into this, we might want Refresh tokens removed

73 Angular Router Guards

73.1 Router Guards - A Primer

A Router Guard is a simply a way to guard someone from going to a page if they aren't allowed to go there(Also known as authentication).

Without going into detail, it is important to recognize that there is such a thing as Route Guards. Most likely, you will be using them within you app for authentication reasons. In particular, your back end will provide you a JWT token. The following are the find fundementals of Angular Route Guards:

1. CanActivate
2. CanActivateChild
3. CanDeactivate
4. CanLoad
5. Resolve

73.1.1 CanActivate

It is used to determine if a certain route can be activated. An example would be as follows:

```
1  export class userGuard implements CanActivate {
2    constructor(
3      private router: Router,
4      private userService: UserService,
5      private userFacade: UserFacade,
6      private projectFacade: ProjectFacade
7    ) {}
8
9    canActivate(
10     route: ActivatedRouteSnapshot,
11     state: RouterStateSnapshot
12   ): Observable<boolean> {
13     const userId = route.paramMap.get('userId');
```

```
14
15      return this.projectFacade.projectId$.pipe(
16        switchMap(projectId =>
17          this.userService.getuser(userId, projectId).pipe(
18            tap(user => {
19              if (user && user.id) {
20                this.userFacade.userLoaded(user);
21              }
22            }),
23            switchMap(user => {
24              return user && user.id
25                ? of(true)
26                : _throw('Unable to retrieve user');
27            }),
28            catchError(error => {
29              this.router.navigateByUrl(this.getPrimaryOutletUrl(state.url
                    ));
30              return of(false);
31            })
32          )
33        )
34      );
35    }
36
37  }
```

Without going into detail the above snippet is a great example of how to tap into the power of canActivate. Here we have two things going on. One if the user actually has an id, and is actually a legitimate user, we will pass along the userFacade which will populate our store, thus populating the page with actual data.

73.1.2 CanActivateChild

CanActivateChild is very similar to canActivate only it is for the childRoute. Will not go into detail on this one, use documentation.

73.1.3 CanDeactivate

TODO

73.1.4 CanLoad

TODO

73.1.5 Resolve

TODO

74 Pre-loading with Route Guards

In Angular, a RouteGuard is an interface that can be implemented to determine if a given route request should be fulfilled or not. The core purpose of a RouteGuard is, as the name implies, to protect a route by applying authorization to it. However, we can use a Route Guard for another purpose: pre-loading data for a view.

74.1 Motiviation

The reason for doing this is to change where in the request process the loading of data happens. Instead of determining the route, rendering view, and then loading data, we find the route, load the data we need, and then render the view with the data already in hand. (Question as to why we want to do this though?)

For clarity sake, we would change the order from A to B.

We want to change the data loading order from Figrue 1, to Figure 2.

74.2 How It Works

In addition to providing hooks for determining authorization, RouteGuards provide a means for pre-fetching and caching data in the store. This is an effect of the place that Route Guards occupy in the processing of requests. Here's a look at a very simple Route Guard:

The canActivate() method is called by Angular to determine if the route in question is allowed, based on the boolean return value. If we were really using it for authorization, we could call out to a AuthService to check a token or similar.

This simple version always allows the route to be activated.

```
1 // Listing 1
2 import { Injectable } from '@angular/core';
3 import { Router, CanActivate } from '@angular/router';
4
5 @Injectable()
6 export class StoreLoadingGuardService implements CanActivate {
```

```
7   constructor(public auth: AuthService, public router: Router) {}
8
9   canActivate(): boolean {
10    return true;
11  }
12
13 }
```

If we were concerned with authorization here, the .canActivate() method would reach out to an authentication service to make it's determination. For our purposes, though, let's use this code in Listing 1 to see how to plug the Route Guard into our app architecture.

```
1   // Listing 2
2  import { Routes, CanActivate } from '@angular/router';
3  import { ExampleComponent } from './example/example.component';
4  import {
5    StoreLoadingGuardService as LoadingGuard
6  } from './auth/loading-guard.service';
7
8  export const ROUTES: Routes = [
9    //...
10   {
11     path: 'example',
12     component: ExampleComponent,
13     canActivate: [LoadingGuard]
14   }
15   //...
16 ];
```

What Listing 2 says, is: when the route example is called, invoke the LoadingGuard.canActivate() method we defined before. Right now, all that will do is allow the route with a default return value of true. However, we can do something more interesting by pre-loading our store.

74.3 The Action

Pre-loading data depends on the store being a central and persistent object that holds application state. When modifying this state, we use ngrx Actions, a la Redux. Below in Listing 3 is a simple Action for loading data. This simple action allows for a load action and a load success action for a Song data type. (Yes, that is correct, we are pre-tending that we are building a music application, right now.)

```
1  export const LOAD_ALL_SONGS = '[Song] Load All Songs';
2  export const ALL_SONGS_LOADED = '[Song] All Songs Loaded';
3
4  export class LoadAllSongs implements Action {
5    readonly type = LOAD_ALL_SONGS;
6    constructor(public payload?: any) { }
7  }
8  export class AllSongsLoaded implements Action {
9    readonly type = ALL_SONGS_LOADED;
```

```
10    constructor(public payload: string[]) { }
11  }
```

74.4 The Store

Our central state might look like the following:

```
1   export interface State {
2   songs: string[];
3   }
4
5   export const initialState: State = {
6   songs: [];
7   };
8
9   export function reducer(state = initialState, action: song.Actions) {
10    switch (action.type) {
11      case song.LOAD_ALL_SUCCESS: {
12        return Object.assign({}, state, {
13          songs: action.payload
14        });
15      }
16
17      // ...
18    }
19  }
```

This reducer simply applies the loaded songs to the state upon a successful load. We will rely on this reducer to merge the data returned by the action into the state.

74.5 The Effect

In the ngrx/store style pattern, we use Effects to handle async calls:

```
1   @Effect()
2   loadAll$: Observable = this.actions$
3   .ofType(song.LOAD_ALL)
4   .switchMap(() => {
5     return this.service.getAll()
6     .map(songs => new song.LoadAllAction(songs))
7     .catch(() => of(new song.LoadAllFailAction()));
8   });
```

This is a simple effect that relies on a service (that has been injected) to retrieve the set of songs, or invoke the LoadAllFailAction action if an error is thrown.

Our store is in place. We can now focus on our new updated route guard.

74.6 Modified CanActivate

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
  const loadedSongs = this.store.select(fromRoot.getSongs)
    .map(songs => songs.length > 0);

  loadedSongs
    .take(1)
    .filter(loaded => !loaded)
    .map(() => new song.LoadAllAction())
    .subscribe(this.store);

  return loadedSongs
    .take(1);
}
```

This determines if the desired data is already present in the store. If it's not, it loads that data, and then allows the route to proceed where the view will have access to the data loaded into the central state.

74.7 Unpacking

To begin, we use a store selector to pull the songs that are already present in this.store.select(fromRoot.getSong and of non-zero length. This we save in the loadedSongs const.

Next, we use take(1) to grab the first item in the dataset, and then check if it's falsey with filter(loaded =¿ !loaded) - the net result being to run the .map() call on an empty dataset if the source contains nothing. The net result is to skip loading the data in the next call if there is already data present.

If the dataset is empty, then we map a call to the song loading service, and subscribe the store to it's result, thereby loading the data into the store. Finally, we do a take(1) to unsubscribe from the source.

75 Containers, Routing + Ngrx/router

We now have a choose-size component, as well as a choose-size module. The dynamics of our app, is that there will only be two parent pages. One will be the choose-size page. The other will be the draw page. When a user goes to the page for the first time, they will see the choose-size page. Therefore we are going to add two routes in our app.

In our app.module.ts, we will use the existing RouterModule that has been created by Nx, and include it in our RouterModule:

```
// Inside imports add
 RouterModule.forRoot([
  {
    path: '',
    redirectTo: 'choose-size',
    pathMatch: 'full'
  },
  {
    path: 'choose-size',
    component: ChooseSizeComponent
  }
```

Now that we have redirected the default homepage to re-direct to the choose-size page, let's try it out. Open up http://localhost:4200, and your page should navigate to the choose-size page, with the text, "Choose Size Works", towards the bottom of the page.

75.1 ngrx/router-store

Before we move any further with regards to angular routes, let's discuss ngrx/router. In short, ngrx/router exists so that the route can be in the store as well. By default ngrx/router will dispatch a ROUTER_NAVIGATION action [1], when a navigation route get's called. This will enable time traveling with regards to routes.

75.2 Why use ngrx/router-store

Ngrx/router-store can be advantageous in two regards. One, it adheres to the principle of there being a single state of truth [2]. Two, it helps applications with regards to breadcrumbs, and adding filters to the url. If one wanted to follow an architecture, where the route contains the current state of the application(filters, inputs etc.) router-store definitely make's it very easy to do so.

75.3 Adding ngrx/router-store to our app

In order to add ngrx/router-store to our app, run:

```
npm install @ngrx/router-store --save
```

[1] We'll discuss this a bit more when we get to the chapter on ngrx/store
[2] https://redux.js.org/docs/introduction/ThreePrinciples.html#single-source-of-truth

In addition, we will be adding the route to our store for the first time, so we will be needing ngrx/store. Please run [3]:

```
npm install @ngrx/store --save
```

In our app.module.ts we will be adding the following:

```
+ import { StoreModule } from '@ngrx/store';
+ import { StoreRouterConnectingModule, routerReducer } from '@ngrx/
    router-store';
+ import { StoreDevtoolsModule } from '@ngrx/store-devtools';
// inside of imports
+ StoreModule.forRoot({
+    router: routerReducer
+ }),
+ StoreRouterConnectingModule.forRoot({
+   stateKey: 'router'
+ });
+ StoreDevtoolsModule.instrument({
+    maxAge: 5
+ }),
```

Listing 75.1: My Javascript Example

75.3.1 Side note when using devtools with ngrx/router-store

One final note when using ngrx/store router-store with devtools. The RouteStateSnapshot is a very large object, containing large amounts of data. I've run across performance issues, it is therefore reccomend setting up a custom router state serializer. The idea is that the user provides the specific items one wants from the route. This allows for maybe 3 key/values to appear, instead of a 1000+. For the sake of brevity, we will not include the technical steps here. However, the code for doing so can be found in the github branch equivalent.

Now that we have our first route set up, as well as ngrx/router-store, let's proceed to building a component.

[3] This will also install ngrx/store-devtools + ngrx/effects

76 Output

Decorator - A way of wraping once piece of code with another. Similar to how we have the ability of wrapping a function with a another, a decorator is a syntax friendly way of wrapping one function with another.

```
1 code example goes here
```

Interpolation -

77 Life Cycle Hooks

Any framework these days is going to have a lifecycle hook. All of them are actually very similar in many ways. There's a component has instantiated, and in between someplace, as in about to change, and is undergoing change, or what not. There is also, always a part in the lifecycle where the component is being destroyed/is destroyed. Angular is no different. I would like to go through the different part's of Angular's life cycle hooks, as it definitely is very important when it comes to development. I would also like to discuss the top 3 most important life cycle hooks when it comes to Angular development.

77.1 Lifecycle Example

Before we get into what the entire lifecycle is, it might be helpful to visualize a lifecycle hook:

```
export class PeekABoo implements OnInit {
  constructor(private logger: LoggerService) { }

  // implement OnInit's `ngOnInit` method
  ngOnInit() { this.logIt(`OnInit`); }

  logIt(msg: string) {
    this.logger.log(`#${nextId++} ${msg}`);
  }
}
```

The above code is hooking into the onInit lifecycle hook. (Hook is exactly what it sounds like. Angular will hook into that particular part of the lifecycle, and implement a certain piece of code). When the component is initialized, it will log out a certain message. OnInit from personal experience is the most used lifecycle hook, so something to keep in mind.

77.2 Angular Lifecycle

At this time, Angular has eight lifecycle hooks, in this order, more, or less:

1. ngOnChanges() - Triggered whenever Angular sets, or resets the data-bound input properties. It is called before ngOnInit, and whenever one, or more data-bound input properties change.

2. ngOnInit() - This one was already featured in the code above! It get's called after Angular display data-bound properties, and set's the directive, or component's input properties.

3. ngDoCheck() - This is called after an ngOnChanges, or ngOnInit. This was created, so that Angular can check on updates it won't check on it's own. [1]

4. ngAfterContentInit() - Triggered after html is populated. It is called once after the first ngDoCheck().

5. ngAfterContentChecked() - After content in html is checked by Angular, this will be called. Called after ngAfterContentInit() and every ngDoCheck() thereafter.

6. ngAfterViewInit() - Triggered after not only view for component is initialized, but child view is initalized as well. For a directive, will trigger once view it is in, will initialize.

7. `ngAfterViewChecked()` - Responds after Angular checks the component's views and child views and the view that a directive is in. Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked().

8. ngOnDestroy() - Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.

77.3 Three Lifecycles Used Most Often

The three lifecycles that are used most often are:

1. ngOnChanges()

2. ngOnInit()

3. ngOnDestroy()

I would like to explain why. When a component initializes, usually we subscribe to some data that we have(if not familiar with subscriptions no worries, will get to that soon.). Sometimes, if we are working with a graphical compoent, for instance, like a chart, we would like to update the component whenever we get new data passed into our input. In

[1] Example of what that would look like should go here.

addition, subscriptions that we pass in from the outside, mainly using state, will still stay around, and soak up our web application's memory. So, it is also quite a common occurence to use `ngOnDestroy()` to manually destroy subscriptions.

Angular documentation presets an example on what lifecycle hooks look like in real time. It is low key incredible, and you should check it out here.

There is more detail to go into with regards to these lifecycles. However, I strongly believe that a fundemental perspective, by reading this you know everything. This book will discuss important points of Angular architecture regarding these hooks, and they can be see here: // Places to put data for hooks can be seen here.

78 Dependency Injection

Dependency Injection from an Angular perspective, pragmatically only solves one real issue. That one issue is Unit Testing. That is, because in an Angular/Typescript setting a developer will have the ability to import and export a file. So there already is a way of decoupling different services/ classes from each other.

```
import { PostsFacade } from '@razroo/razroo/data-access/posts';

@Component({
  selector: 'razroo-blog',
  templateUrl: './blog.component.html',
  styleUrls: ['./blog.component.scss']
})
export class BlogComponent implements OnInit {
  posts: any[];
  allPosts$: Observable<Post[]> = new PostsFacade().allPosts$;

  constructor() {}

  ngOnInit() {}

}
```

Listing 78.1: passing in without dependency injection

In the above, our class doesn't control how this value in injected. It goes straight from the import into our class. However, if we were to do something like this:

```
import { PostsFacade } from '@razroo/razroo/data-access/posts';

@Component({
  selector: 'razroo-blog',
  templateUrl: './blog.component.html',
  styleUrls: ['./blog.component.scss']
})
export class BlogComponent implements OnInit {
  posts: any[];
  allPosts$: Observable<Post[]> = this.postFacade.allPosts$;

  constructor(private postsFacade: PostsFacade) {}

  ngOnInit() {}

}
```

Listing 78.2: passing in with dependency injection

Now it is our `BlogComponent` class that is controlling how the `postsFacade` is getting passed through. This makes things particularly easy when it comes to unit testing. It allows us to override the value in our unit test, and create mocks for all services. *We will discuss more on this in the chapters in the chapters involving unit testing, but I just wanted to bring up the main reason behind unit testing here.*

Less obvious, and specifically if the `providedIn` is used, is dependency injection also helps keep bundle sizes compact. This is done by tree shaking, which refers to the compiler removing code from final app, if it is not actually referenced by the app. If we do not use `providedIn`, tree shaking will not be done.

In addition, as we discussed above, it allows us to keep our configurations separate. While the ability to export/import is an option within Typescript, dependency injection, allows the framework to be completely aware of everything being used within the framework. So, besides unit testing, this architecture can be useful for using tokens which contain a particular value, and then being overridden depending on environment of application(e.g. development vs. production).

78.1 Real World Example

78.1.1 Creating Injectable Service

The DI(Dependency Injection) Framework, allows for an injectable service class to be passed to a component. For instance, let's say that we want to create an injectable service, we would do the following:

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root',
5 })
6 export class PxCodeService {
7   constructor() { }
8 }
```

This code right here is doing two things:

1. It is saying that this service is an Injectable.

2. It is saying that this injectable should be provided in the root(aka the AppModule).

Now would be a good time to discuss what the `providedIn` property does. It accomplishes two things:

1. Angular creates a single, shared instance of the service and injects it into any class that asks for it. So, there is no need to insert it as a provider for your module, and you can simply pull it into your class whenever you want. [a]

2. It also allows Angular to optimize an app, by removing the service from the compiled app, if it isn't used. [b]

[a] Don't worry if you are not familiar with how to use a service in your class, we will get to that soon.
[b] Taken from documentation for Angular.

78.1.2 Including Injectable Service in Component

Now if we would like to include this service in our component, we would do the following:

```
// code-box.component.html
<div *ngFor="let codeBox of codeBoxes">
  {{codeBox.data}}
</div>
```

```
// code-box.component.ts
import { Component }   from '@angular/core';
import { CodeBox }        from './code-box.interfaces';
import { PxCodeFacade } from './px-code.facade';

@Component({
  selector: 'px-code-box',
  template: './code-box.component.html',
  styles: ['./code-box.component.scss'],
})
export class HeroListComponent {
  codeBoxes: CodeBox[];

  constructor(pxCodeFacade: PxCodeFacade) {
    this.codeBoxes = pxCodeFacade.getCodeBoxes();
  }
}
```

Listing 78.3: Include Injectable Service in Component

Now would technically be a good time on discussing how to test these mocked dependencies, however it get's a little bit complicated. How to mock dependencies, will be discussed in the section on unit testing.

78.2 Services that need other services

Services can have their own dependencies. If we wanted to inject a service, into our service, it would be as simple as doing the following:

```
1  import { Injectable } from '@angular/core';
2  import { Logger } from '../logger.service';
3
4  @Injectable({
5    providedIn: 'root',
6  })
7  export class PxCodeService {
8    constructor(private logger: Logger) { }
9
10   getLog() {
11     this.logger.log('getting codeboxes');
12   }
13 }
```

As we can see in the above, our injected service, is taking another injected service. By simply passing it into the constructor, similar to how we do for our components, we can use it within our application.

78.3 Dependency Injection Token

Internally Angular uses dependency injection tokens for injectable services, to reference what injectable it is using. As an Angular developer, we also have the option to use these tokens directly within our app. There are two different ways of providing tokens in Angular:

1. Strings

```
1  {provide: 'EmailService', useClass: MandrillService}
```

2. Type Tokens

```
1  {provide: EmailService, useClass: MandrillService}
```

3. Injection Tokens

```
1  new InjectionToken{provide: EmailService, useClass:}
```

   Benefit of this approach, is that it avoids name clashes.

Tokens can be a useful way for providing a default value to be used across the app, hijacking the value, and providing something else, in the scenarios where it is not of use. An example of such, is within the Angular Material Components. By default, the `MAT_DATE_LOCALE` will

use the existing `LOCALE_ID`. However, if you would like to override the date locale across the app, all that needs to be done, is to overrride the `MAT_DATE_LOCALE` token.

```
@NgModule({
  providers: [
    {provide: MAT_DATE_LOCALE, useValue: 'en-GB'},
  ],
})
export class MyApp {}
```

This allows for numerous components across the app, using the same token to be overridden.

78.4 Wrapping Up

The intent of this chapter, is to introduce the concept of dependency injection in Angular. In addition, present the scenarios in which it is used in a regular enterprise application. Dependency injection as a pattern can be very complex, and I feel looking at the documentation, the majority of use cases tend not to be used. So I wanted to keep this simple. Unit testing, the other widely used part of dependency injection, will be discussed in a separate chapter on unit testing.

79 Input

The Input decorator is one of two integral Angular Decorators, instrumental for passing into a component, and out of a component into a another one. It's name sounds like what it does. Similar to an html input, wherein a user inputs a value, and that's the value that the form now has. Similarly, for an Angular component, you can put a name on what you would like your input is. Use that anticipated input value/name within your component. Now, you have the ability to re-use your component in multiple places, and pass in the data of your choosing based on the Input value you used.

It is important to realize that @Input and creating a re-usable component go hand in hand. If you would like to create a re-usable component, 9/10 times (rough estimate), you will need to use an @Input decorator. Ok, so let's get down to how to use it.

79.1 Input - An Example

Here is a great example. Let's say we have a bank account component, that we would like to create. This bank account component will be used on multiple pages for our web application. It will be displayed in multiple places. Sometimes in the header, sometimes in a modal, or to display the bank used by someone else, within your network.

```
// bank account component - bank-acount.component.ts
@Component({
  selector: 'bank-account',
  template: './bank-account.component.html',
})
class BankAccount {
  // This property is bound using its original name.
  @Input() bankName: string;
  // this property value is bound to a different property name
  // when this component is instantiated in a template.
  @Input() id: string;
}
```

In the html of your bank-account component:

```
<div> Bank Name: {{ bankName }} </div>
<div> Account Id: {{ id }} </div>
```

As we can see in the above code, our @Input's are considered as if they are a value contained directly on our components.

Just for the sake of clarity, our re-usable component is going to be put into it's own module. This module is then going to be imported by the module, containing the component we are going to use. This code is not included at this time.

79.1.1 Including Component in another Component

Now we have the ability to include this component with an Input in another component:

```
<bank-account bankName="RBC" id="4747"></bank-account>
```

This would be @Input() in a nutshell.

79.2 bindPropertyName

Input() does allow for an optional bindingPropertyName. This would mean that the re-usable component would internally refer to the Input value as one way, and the consuming component would refer to it, in another.

```
// bank account component - bank-acount.component.ts
@Component({
  selector: 'bank-account',
  template: './bank-account.component.html',
})
class BankAccount {
  // This property is bound using its original name.
  @Input() bankName: string;
  // this property value is bound to a different property name
  // when this component is instantiated in a template.
  @Input('bank-account') id: string;
}
```

In the html of your bank-account component:

```
<div> Bank Name: {{ bankName }} </div>
<div> Account Id: {{ id }} </div>
```

In the html of the component consuming the bank-account component.

```
<bank-account bankName="RBC" bank-account="4747"></bank-account>
```

80 @Output

Output is another decorator native to Angular. On it's own it doesn't do anything. However, it does allow for the three following things to happen:

1. Marks a class field as an output property.

2. Supplies configuration Metadata (we will get around to what that means soon)

3. DOM property bound to the output property is automatically updated during change detection.

The Output decorator is always used in tandem with an event listener. What this means, is that we can have a re-usable component, with a button for instance. By using Output, we can have a function within the dumb component, that whenever it get's called, it triggers the parent function.

80.1 Example of @Output

Here is a great example of what an Output is, and what it accomplishes. In our scenario, we want to build a re-usable pxl-color-changer component:

```
// pxl-code-changer.component.html

import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'pxl-color-changer',
  template: './pxl-code-changer.component.html',
  styleUrls: ['./pxl-code-changer.component.scss'],
})
export class PxlCodeChanger implements OnInit {
  @Output() colorChanged = new EventEmitter<any>();
  constructor() {}

  changeColor(data: string) {
    this.colorChanged.emit(data);
  }
}
```

```
<div>
<form>
<-- Update code when we get it -->
</form>
</div>
```

and in the parent component consuming our component:

```
<div>
<pxl-color-changer></pxl-color-changer>
</div>
```

80.2 bindPropertyName

Similar to Input, Output allows for binding an optional property name. This would mean that the re-usable component would internally refer to the Input value as one way, and the consuming component would refer to it, in another.

```
// pxl-code-changer.component.html

import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'pxl-color-changer',
  template: './pxl-code-changer.component.html',
  styleUrls: ['./pxl-code-changer.component.scss'],
})
export class PxlCodeChanger implements OnInit {
  @Output('changeColor') colorChanged = new EventEmitter<any>();
  constructor() {}

  changeColor(data: string) {
    this.colorChanged.emit(data);
  }
}
```

81 Internationalization and Localization

If not familiar with already, internationalization is the process of making sure your app can be used by a worldwide audience. Localization, not to be confused with internationalization, is the process of changing app's text, to represent locale of user.

Internationalization is one of those things that is generally done farther down the life cycle of an app. This makes complete sense, as when building a product, a team will generally want to build the infrastructure first, based on a particular language. Then, it will want to make it so that the app can be based on numerous different demographics. However, being aware of the following ahead of time will make internationalization more of a seamless process. This chapter goes through the entire process of internalization and localization.

81.1 General Concerns of Internationalization

1. What type of dates, numbers, percentages and currencies should the user get.

2. Setting aside text in components, so they can be swapped out to the appropriate language.

3. Plural words are different in different languages. It's important to have something baked within the framework that does this.

4. Alternate text based on scenario(e.g. if person in New York display such and such event, if person in San Francisco display message such and such.)

81.2 General Concerns of Localization

There are three concerns with regards to localization:

1. Creating multiple language versions of app(will get to how that is done)

2. Extracting Localizable text(We will get to this, but in short create an enterprise grade source file, that only deals with the text to be translated)

3. Building and servimg an app for a given locale.

81.3 Default Locale

Angular's internal framework uses the BCP47 norm. It's very important to note that if you plan on building on top of Angular's localization framework, that you use this as your standard within the framework. Specifications for BCP47 change over time. However, this might not make a difference within your application, as this only happens to subset.

Angular by default will use `en-us`. Therefore the default locale is American English.

81.4 i18n pipes

Angular includes within the framework, the ability to use something called i18n pipes. i18n is an abbreviation of word "internationalization". It's perhaps the world's most clever abbreviation standing for "i plus eighteen letters, plus the letter n". I thought I'd share a little bit of the interesting history behind `i18n`.

> "A DEC employee named Jan Scherpenhuizen was given an email account of S12n by a system administrator, since his name was too long to be an account name. This approach to abbreviating long names was intended to be humorous and became generalized at DEC. The convention was applied to "internationalization" at DEC which was using the numeronym by 1985. Use of the term spread. Searching the net, we found uses on-line as early as 1989. It was being used on /usr/group, which evolved into UniForum. The X Window standards community was also using the abbreviation by 1989. Looking in printed texts, the earliest reference I could find was in the book Soft Landing in Japan, published by American Electronic Association, 1992.
> The extension of this naming convention to the terms Localization (l10n), Europeanization (e13n), Japanization (j10n), Globalization (g11n), seemed to come somewhat after the invention of "i18n". The terms Canonicalization and Normalization, defined more recently, also have numeronym forms (c14n and n11n)." [a]
>
> [a] http://www.i18nguy.com/origini18n.html

(Graphic goes here for internationalization)

There is no actual pipe actually called the "i18n" pipe. However, these four pipes:

1. `DatePipe`

2. `CurrencyPipe`

3. `DecimalPipe`

4. `PercentPipe`

do use the `i18n` internal logic, and will automatically modify themselves based on the locale supplied within the app. As mentioned earlier, Angular will only by default import `en-US`. If you want to use other languages within the app.

81.4.1 Import Locale Data for Other Languages

If you would like to import data for other languages, import them locally within your `app.module.ts` file.

```
1 import { registerLocaleData } from '@angular/common';
2 import localeFr from '@angular/common/locales/fr';
3
4 // the second parameter 'fr' is optional
5 registerLocaleData(localeFr, 'fr');
```

Listing 81.1: app.module.ts

81.5 Understanding Translation Process

The Angular translation process definitely seems foreign, and even after going through it a couple of times, it can be difficult to grasp the whole picture. So, I just want to re-iterate four step process to Angular Translation here:

1. It determines what is static text(as opposed to dynamic text retreived back from the backend), so that it can be translated.

2. Uses the internal Angular CLI `i18n` command to transfer over determined translatable text to an industry standard translation file. (Need more information as to what this looks like.)

3. Translates extracted text, into the target language. The dynamics of how this works, is that it edits the already existing file, to the targeted language.

4. It then merges the translated file into the app, by replacing the original untranslated text, with the new translated text.

81.6 How to Make Content Translatable

You might be wondering how to make content translatable. Angular internally allows for content to be marked with the `i18n` tag like such:

```
1   <h1 i18n>i18n example</h1>
```

It is important to note the i18n tag is not an Angular directive. It is a custom tag recognized by the Angular compiler, and will be removed once the Angular compiler does it's magic.

81.6.1 Add a Description and a Meaning

Angular also offers the ability to add a description and a meaning to the `i18n` tag.

```
1 <h1 i18n="i18n tutorial text for internationalization chapter">i18n
    example</h1>
```

Listing 81.2: i18n description

We can also add a meaning to the `i18n` tag, in additon, to our already added description. The interface looks like this `<meaning | description>`.

```
1 <h1 i18n="tutorial text|i18n tutorial text for internationalization
    chapter">i18n example</h1>
```

Listing 81.3: i18n `<meaning | description>` Example

You might be wondering how offering a meaning and description will make a difference with regards to the translation of the application. The Angular extraction tool preserves both meaning and description in the translation source file. The translater, then translating the source file, can use the meaning and description to properly determine how file should be translated. (Likewise, when using only a meaning, or a description, the compiler will preserve the meaning, or description with the compiled text.)

Combination of Meaning and Description

However, the combination of meaning and description within your app, will produce a specific id of a translation. With regards to the combination of meaning and description, meaning is the main identifier in this process. Only if the meaning is different, will the Angular compiler produce a different id. If, for instance, the meaning is the same, but the description is different, the compiler will still produce the same id.

81.7 Set a Custom ID for Persistence and Maintenance

For each file that the Angular translator produces, it will by default attach an id to that file as well. It will look something like the following:

```
1 <trans-unit id="ba01234d3d68bf669f97b8d96a4c5d8d9559aa3" datatype="html"
    >
```

Listing 81.4: example.fr.xlf.html

When the text is changed, the extractor tool will generate new text. You must then go ahead and manually change the id. [1]

81.7.1 Setting up Custom ID

Angular will also allow you to set up a custom id for your translatable text:

```
1 <h1 i18n="@@sampleText">i18n Sample</h1>
```

The custom id persists even when text changes. This allows for maintenance and scalability. Perhaps even the ability to hook an id into a content management system, and allow for an admin to control translation of different parts of the application.

81.7.2 Using a Custom ID with a Description and/or Meaning

There is also the option within the app to use a custom id with the description.

```
1 <h1 i18n="i18n tutorial text for internationalization
    chapter@@sampleText">i18n example</h1>
```

Listing 81.5: custom id with description

In addition, you can add meaning along with description to your custom id:

```
1 <h1 i18n="tutorial text|i18n tutorial text for internationalization
    chapter@@sampleText">
2   i18n example
3 </h1>
```

Listing 81.6: custom id with meaning and description

Make sure to use unique custom ids for translation. If two of the same ids are used for translation, Angular will only use the one which is meant to be used within your app.

[1] Not sure if this means that the text must be manually updated, or if compiler will do that for you.

81.7.3 Translating attributes

Text can be contained inside of an attribute such as `placeholder`, or `title`. Due to the dynamics of how the `i18n` compiler works, if you would like to translate the text within an attribute, you will have to use the `i18n` flavored attribute. It follows the syntax of `i18n-x` wherein `x` is the name of the attribute you want to translate.

```
<img [src]="logo" i18n-title title="Razroo logo" />
```

81.8 Regular Expressions - Pluralization — Selections

Different languages treat the plural form of a word different ways. Angular offers a plural pipe out of the box. This is combination with `i18n` allows for a very sophisticated method with regards to internationalization.

```
<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago}
    other {{{minutes}} minutes ago}}</span>
```

The plural pipe, allows for the specification of the following categories:

1. =0(or any other number)
2. zero
3. one
4. two
5. few
6. many
7. other

In the above code, we are specifying:

1. Just now if the numbers of minutes is 0
2. "one minute ago" if the number of minutes is 1
3. and for every other amount we are specifying "x minutes ago"

By tagging in the `i18n` attribute into our span element, we are translating the appropriate text based on the appropriate scenario.

81.8.1 Alternative Text Messages

In addition to the scenario of plural text, another scenario to keep in mind, is when we want to select a certain text based on the variable provided.

```
<span i18n>The author is {gender, select, male {male} female {female}
    other {other}}</span>
```

In the above text, based on whether, or not the author is a female, or male, or other, we will allow for a specific type of text to be displayed. In addition, being that we are supplying the `i18n` pipe to our app, this text will appropriately be changed to the correct text based on a certain localization.

81.8.2 Alternative + Plural Combined

There is also the option is combine plural and alternative text message capabilities together.

```
<span i18n>Updated: {minutes, plural,
=0 {just now}
=1 {one minute ago}
other {{{minutes}} minutes ago by {gender, select, male {male} female {
    female} other {other}}}}
</span>
```

As you can see in the above code, the other section, is combined with code for select. Absolutely wonderful.

81.9 Translation Source File

81.9.1 Understanding the Angular Translation Process

First and foremost, before describing how to create a translation source file, let's discuss the purpose of a translation source file. The idea is, is that we have tagged all files that we would like to translate, with the `i18n` attribute (which really isn't an attribute, and is more a compiler specific indicator, but I digress). After the source file is created, we have an xml specific file. It will look something like this:

```
<trans-unit id="introductionHeader" datatype="html">
  <source>Hello i18n!</source>
  <note priority="1" from="description">An introduction header for this
      sample</note>
  <note priority="1" from="meaning">User welcome</note>
```

```
5 </trans-unit>
```

Listing 81.7: src/locale/messages.xlf

Now if we would like, for instance, to create a french equivalent file, we would create a source file with the `fr` suffix.

```
1 <trans-unit id="introductionHeader" datatype="html">
2   <source>Hello i18n!</source>
3   <note priority="1" from="description">An introduction header for this
        sample</note>
4   <note priority="1" from="meaning">User welcome</note>
5 </trans-unit>
```

Listing 81.8: src/locale/messages.fr.xlf

81.9.2 XLIFF Editor

This in an enterprise setting, the xlf file is usually sent over to a french translater, who will then convert the english inside of these french files, to french. They will use something called an XLIFF Editor, which stands for textitXML Localization Interchange File Format. The xlf file suffix stands for *XML Localization Format*.

81.9.3 Create a Translation Source File

The Angular CLI has an internal mechanism for generating a translation source file. Simply open a terminal window at the root of the app project and run the `CLI` command `xi18n`.

```
ng xi18n
```

Personally, I like to create files inside of the app's `src/locale` folder path. For that let's change the output path using the CLI:

```
ng xi18n --output-path src/locale
```

Generating Different Translation Formats

The Angular CLI also offers the ability to alter the format of the outputted file.

1. XLIFF 1.2 (default)

2. XLIFF 2

3. XML Message Bundle (XMB)

Respectively, that would be:

```
ng xi18n  --i18n-format=xlf
ng xi18n  --i18n-format=xlf2
ng xi18n  --i18n-format=xmb
```

Other Notable CLI options

The CLI also offers the ability to change the name of the translation source file using the `--outFile` command option

```
ng xi18n --out-file source.xlf
```

There is also the option to change the base locale of your app, from the default `US-EN` using the `--i18n-locale` command option.

```
ng xi18n --i18n-locale fr
```

81.9.4 Translate the source text

As mentioned earlier, when we were trying to understand the process behind creating a source file, we mentioned that we will be creating translation source files. We will then be going ahead and sending those translation source files over to a translater. As standard practice Razroo recommends a singular folder that can be used for the localization process. We do this by using the `output-path` option mentioned before:

```
ng xi18n --output-path src/locale
```

and the file that will be generated will be the `messages.xlf` file:

```
1 <trans-unit id="introductionHeader" datatype="html">
2   <source>Hello i18n!</source>
3   <note priority="1" from="description">An introduction header for this
        sample</note>
4   <note priority="1" from="meaning">User welcome</note>
5 </trans-unit>
```

Listing 81.9: src/locale/messages.xlf

and now after translation, and creating the respective french file, it will look something like the following:

```
<trans-unit id="introductionHeader" datatype="html">
  <source>Hello i18n!</source>
  <target>Bonjour i18n !</target>
  <note priority="1" from="description">An introduction header for this
      sample</note>
  <note priority="1" from="meaning">User welcome</note>
</trans-unit>
```

Listing 81.10: src/locale/messages.fr.xlf(after translation)

In the above you will notice that we create a target equivalent XML tag. This is used by the framework internally to say what the source should be translated over to.

81.10 Translation Source File that includes plural and select expressions

81.10.1 Translating Plural

To translate a `plural` we only translate the text related values.

```
<trans-unit id="5a134dee893586d02bffc9611056b9cadf9abfad" datatype="html
    ">
  <source>{VAR_PLURAL, plural, =0 {just now} =1 {one minute ago} other
      {<x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago} }</
      source>
  <target>{VAR_PLURAL, plural, =0 {Ãã l'instant} =1 {il y a une minute}
      other {il y a <x id="INTERPOLATION" equiv-text="{{minutes}}"/>
      minutes} }</target>
</trans-unit>
```

Listing 81.11: src/locale/messages.fr.xlf

81.10.2 Translating Select

Similarly for select:

```
<span i18n>The author is {gender, select, male {male} female {female}
    other {other}}</span>
```

The Angular extraction tool, will break the above into two separate parts:

```
<trans-unit id="f99f34ac9bd4606345071bd813858dec29f3b7d1" datatype="html
    ">
```

```
2     <source>The author is <x id="ICU" equiv-text="{gender, select, male
          {...} female {...} other {...}}"/></source>
3     <target>L'auteur est <x id="ICU" equiv-text="{gender, select, male
          {...} female {...} other {...}}"/></target>
4   </trans-unit>
5   <trans-unit id="eff74b75ab7364b6fa888f1cbfae901aaaf02295" datatype="html
       ">
6     <source>{VAR_SELECT, select, male {male} female {female} other {other}
          }</source>
7     <target>{VAR_SELECT, select, male {un homme} female {une femme} other
          {autre} }</target>
8   </trans-unit>
```

Listing 81.12: src/locale/messages.fr.xlf

The first `<trans-unit>` will contain the text outside of `select`. The second `<trans-unit>` will contain the text inside of the `select` message. The reason that the extraction tool will separate it into two separate sections, is because each expression unit is separate. (For our `plural` expression it was all on it's own. If it had text outside of the `plural` expression, it would indeed be separated into two parts)

81.10.3 Translating a Nested Expression

So let's say that we have mixed and matched plural and select as follows:

```
1 <span i18n>Updated: {minutes, plural,
2 =0 {just now}
3 =1 {one minute ago}
4 other {{{minutes}} minutes ago by {gender, select, male {male} female {
      female} other {other}}}}
5 </span>
```

The result will be very similar to before. The extraction tool will separate this into two different parts:

```
1   <trans-unit id="972cb0cf3e442f7b1c00d7dab168ac08d6bdf20c" datatype="html
       ">
2     <source>Updated: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1
          {...} other {...}}"/></source>
3     <target>Mis Ãã jour: <x id="ICU" equiv-text="{minutes, plural, =0
          {...} =1 {...} other {...}}"/></target>
4   </trans-unit>
5   <trans-unit id="7151c2e67748b726f0864fc443861d45df21d706" datatype="html
       ">
6     <source>{VAR_PLURAL, plural, =0 {just now} =1 {one minute ago} other
          {<x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago by {
          VAR_SELECT, select, male {male} female {female} other {other} }}
          }</source>
7     <target>{VAR_PLURAL, plural, =0 {Ãã l'instant} =1 {il y a une minute}
          other {il y a <x id="INTERPOLATION" equiv-text="{{minutes}}"/>
          minutes par {VAR_SELECT, select, male {un homme} female {une femme}
           other {autre} }} }</target>
```

```
8 </trans-unit>
```

Listing 81.13: src/locale/messages.fr.xlf

81.11 Merge Translated Files I.E. How to Use With App

So now that we have our completed translated files, we will need to provide the Angular compiler with three pieces of information:

1. The translation file(e.g. `messages.fr.xlf`)
2. The translation file format(e.g. `xlf`)
3. The locale(e.g. `fr`)

81.11.1 AOT v. JIT

This all depends on how you are compiling your app. Just to re-iterate, there are two ways of doing this:

1. AOT(Which compiles at build time)
2. JIT(Which compiles at run time)

Depending on how you are running your app, this depends. If you are running using AOT, then being that the compilation is being done at build time, you will need to specify configurations. If you are using JIT, you will have to specify providers with your app, so that it is aware of the translation files.

81.11.2 Merge Files with AOT Compiler

With AOT there will be four configurations:

1. `i18nFile`: the path to the translation file.
2. `i18nFormat`: the format of the translation file.
3. `i18nLocale`: the locale id.
4. `outputPath`: folder to distribute build.

That you will want to add to your `angular.json` file.

81.11.3 Configuration for Development Environment

```
"build": {
  ...
  "configurations": {
    ...
    "fr": {
      "aot": true,
      "outputPath": "dist/my-project-fr/",
      "i18nFile": "src/locale/messages.fr.xlf",
      "i18nFormat": "xlf",
      "i18nLocale": "fr",
      ...
    }
  }
},
"serve": {
  ...
  "configurations": {
    ...
    "fr": {
      "browserTarget": "*project-name*:build:fr"
    }
  }
}
```

Listing 81.14: angular.json

You can pass configuration to the `ng-serve`, or `ng-build` commans by doing the following:

```
ng serve --configuration=fr
```

81.11.4 Configuration for Production Environment

For prod, we specify a separate `production-fr` build configuration in the CLI configuration file:

```
...
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": { ... },
    "configurations": {
      "fr": {
        "aot": true,
        "outputPath": "dist/my-project-fr/",
        "i18nFile": "src/locale/messages.fr.xlf",
        "i18nFormat": "xlf",
```

```
12          "i18nLocale": "fr",
13          "i18nMissingTranslation": "error",
14        }
15      }
16    },
17    ...
18    "serve": {
19      "builder": "@angular-devkit/build-angular:dev-server",
20      "options": {
21        "browserTarget": "my-project:build"
22      },
23      "configurations": {
24        "production": {
25          "browserTarget": "my-project:build:production"
26        },
27        "fr": {
28          "browserTarget": "my-project:build:fr"
29        }
30      }
31    }
32  }
```

Listing 81.15: angular.json

81.11.5 Merge Files with JIT Compiler

JIT is an entirely different beast than working with AOT.

In truth, I prefer AOT because it is much more straightforward. I was debating in this chapter at all to mention configurations for JIT. However, as I have been in a similar scenario, there will be an app that will be built out with JIT, and moving over to AOT, immediately is something which will be very difficult, and might prevent the app from being able to move forward. For that reason, I am jotting down configurations for JIT as well. However, Razroo recommended architecture, is that your app uses `AOT` exclusively from the beginning.

Knowledge of Three Things

The JIT compiler requires the knowledge of three things to support translation:

1. Import language translation file as a string constant
2. Create corresponding translation provider
3. Bootstrap app with aforementioned providers

Three Providers To Be Aware Of

There are three providers within an Angular JIT translation setting to be aware of:

1. `TRANSLATIONS` - String containing content of translation file.
2. `TRANSLATIONS_FORMAT` - Format of file i.e. `xlf, xlf2, or xtb`
3. `LOCALE_ID` - locale of the target language

81.11.6 Merge Files with JIT Compiler In practice

Let's take a look at how this might work in practice.

First and foremost, in our main.ts let's provide:

1. `TRANSLATIONS` (We will be using the webpack raw-loader to return the content as a string)
2. `TRANSLATIONS_FORMAT`

```
1  import { enableProdMode, TRANSLATIONS, TRANSLATIONS_FORMAT } from '
       @angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-
       dynamic';
3
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 // use the require method provided by webpack
12 declare const require;
13 // we use the webpack raw-loader to return the content as a string
14 const translations = require('raw-loader!./locale/messages.fr.xlf').
       default;
15
16 platformBrowserDynamic().bootstrapModule(AppModule, {
17   providers: [
18     {provide: TRANSLATIONS, useValue: translations},
19     {provide: TRANSLATIONS_FORMAT, useValue: 'xlf'}
20   ]
21 });
```

Listing 81.16: src/main.ts

Next, let's provide the `LOCALED_ID` in our main module:

```
import { LOCALE_ID, NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from '../src/app/app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  providers: [ { provide: LOCALE_ID, useValue: 'fr' } ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Listing 81.17: app.module.ts

81.12 Know When a Translation is Missing

When a translation is missing the build will still succeed. However, it will still generate an error. As to what level that is, that is something that you can control. There are three levels:

1. Error - Throw an error. If AOT, build will fail. If JIT, app will fail to load.
2. Warning - Shows a warning("Missing translation") in the console, or shell.
3. Ignore - Do nothing yo.

81.12.1 Specify Warning in Practice

For the AOT Compiler we specify the warning in the actual `angular.json` file.

```
"configurations": {
  ...
  "fr": {
    ...
    "i18nMissingTranslation": "error"
  }
}
```

Listing 81.18: angular.json

Notice that in the above, the option for `i18nMissingTranslation` is located within the key/-value for `fr`.

For the JIT Compiler, the configuration is set within the `main.ts` file:

```
import { MissingTranslationStrategy } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-
    dynamic';
import { AppModule } from './app/app.module';
// ...

platformBrowserDynamic().bootstrapModule(AppModule, {
  missingTranslation: MissingTranslationStrategy.Error,
  providers: [
    // ...
  ]
});
```

Listing 81.19: src/main.ts

Notice the added `missingTranslation: MissingTranslationStrategy.Error,`

81.13 Tips and Tricks

81.13.1 Using ng-container with i18n

While something that might seem obvious for the experience Angular developer, it is still a notable mention. You can use `ng-container` to remove the requirement of having an html element be created when translating text. E.g.

```
<ng-container i18n>Translate this into multiple languages</ng-container>
```

82 Content Projection

Any component can technically be re-usable. However, what makes content projection so great, is that it allows the content inside of a component to change based on the need of the application. In addition, it allows us to separate concerns. We can build a component for display, and another component built for handling user actions.

82.1 Single Slot Projection

If we wanted to create a component wherein we can use content projection, it is as simple as adding ng-content inside of our component:

```
<!-- inside reusable component -->
<ng-content></ng-content>

<!-- inside component, consuming the re-usable component -->
<reusable-component> <p>Content goes here</p> </reusable-component>
```

As we can see, using our ¡reusable-component¿, we have the ability to put it whereever we want, and change the content based on the parent component consuming it. However, what if we have two separate places withing our component that we would like to inject content. For instance, let's say we have a card component, and we want there to be different content inside of the header and main body of the component?

82.2 Multiple Slot Projection

This is my preferred method of mutliple content projection, by creating binding content projection to class. In particular, because I feel it's a great of making sure content projection is transparent across the entire lifecycle. We can now do the following:

```
<div class="header">
<ng-content select=".header"></ng-content>
</header>
<div class="body">
<ng-content select=".body"></ng-content>
</div>
```

In our parent component consuming the re-usable component:

```
<reusable-component>
<div class="header">CSS</div>
<div class="body">{{css-data}}</div>
</reusable-component>
```

Just like that, we can have our content project in multiple places, into the re-usable component.

82.3 Styling Projected Content

One of the scenarios that comes up alot with regards to projected content, is attempting to style it. For instance, you might want the content in one component to have a top border, and in others for the text color to be of a different style. So how would we do this, being that we are projecting the content into a separate component?

> It should be mentioned that styling in this matter isn't the right approach. However, because there are times wherein styling in this fashion is neccesary for the particular use case, this is most definitely useful mentioning.

```
:host ::ng-deep .header {
  color: blue;
}

:host ::ng-deep .body {
  margin-top: pxl-space-multiplier(1);
}
```

Just like that, we are able to style the content within our project.

82.4 Interacting with Projected Content

One more scenario to take into consideration when working with projected content is to interact with it. For instance, let's say that we want to project an input field into our projected content. In addition, we would like to determine when that input field has been clicked on? The following is the best stategy. We will create a directive, that focuses on event handling. [1]

[1] Refer back to the chapter on directives. Generally speaking directives help solve two things:

82.4.1 An Example

This is a simple directive, that targets the focus and blur of host element using the @HostListener element.

```
@Directive({
  selector: '[inputRef]'
})
export class InputRefDirective {
  focus = false;

  @HostListener("focus")
  onFocus() {
    this.focus = true;
  }

  @HostListener("blur")
  onBlur() {
    this.focus = false;
  }
}
```

Not we can pass this inputRef directive onto our projected content:

```
<h1>FA Input</h1>
<fa-input icon="envelope">
  <input inputRef type="email" placeholder="Email">
</fa-input>
```

Now within our re-usable component, we can use the `@ContentChild` decorator to inject the `inputRefDirective` within our component. Then, we can use the `@HostBinding` decorator to change the class on our re-usable component, based on the status of the input ref.

```
@Component({
  selector: 'fa-input',
  template: `
    <i class="fa" [ngClass]="classes"></i>
    <ng-content></ng-content>
  `,
  styleUrls: ['./fa-input.component.css']
})
export class FaInputComponent {

  @Input() icon: string;

  @ContentChild(InputRefDirective)
  input: InputRefDirective;

```

1. Event handling
2. Passing in Values

These two tend to work in tandem with each other. Similarly here, we will be using directives to pass in a value, and have it work in tandem with event handling.

```
16    @HostBinding("class.focus")
17    get focus() {
18      return this.input ? this.input.focus : false;
19    }
20
21    get  classes() {
22      const cssClasses = {
23        fa: true
24      };
25      cssClasses['fa-' + this.icon] = true;
26      return cssClasses;
27    }
28  }
```

82.5 Wrapping Up

As we have seen, content projection is a very powerful way of re-using content within our component. We have also covered the two stereo-typical uses cases, which we will have to solve in an enterprise app from time to time. I.e. event handling for our project content, as well as controlling the styling for our projected content.

83 Displaying Data

In Angular, one of the nicer things about the migration from AngularJS to Angular, was that many html bindings felt at home. If you are someone who like me, had the ability to experience Angular, after working with AngularJS, then templates feel entirely intuitive. In fact, I actually had a difficult time writing this chapter, because I almost overlooked the fact, that to many, displaying data in an Angular setting can be counter intuitive.

However, I still think that it rings true, that templates in Angular, are one of the easier topics to grasp in Angular. Granted, and this is a very important point, granted that the syntax is properly explained. Sometimes, when learning syntax for the first time, there's so much there, that it's easy to overlook reasons behind syntax. Perhaps we assume the reason there are different symbols, letters, etc. for the different ways of operating within a framework, is because we just need to represent the different ways of doing a particular action within a framework. I think it goes without saying, that it would be valuable as a matter of documentation to mention how component and Templates work within Angular. However, this book will also make sure to go into the syntax, and offer why it looks like the way it does.

83.1 Interpolation

Interpolation in the dictionary, means inserting something of a different nature into something else. In the context of Angular, interpolation means being able to place a javascript expression, in your html. This is signified by the double curly brace.

```
1 This is 2 + 2
2 <!-- This is 4 -->
```

Listing 83.1: interpolation-example.component.html

83.2 Angular Components

In an Angular setting, based on best practices, and those set into place by the Angular CLI, the generation of the component, will consist of four files:

1. *.component.ts
2. *.component.spec.ts
3. *.component.html
4. *.component.scss

In the above files, html and component files will tend to interact with each other the most. For any component file, there is atleast going to be one, or more properties placed in the respective component file.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'pxl-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent implements OnInit {
  @Input() title: string;
  constructor() { }

  ngOnInit() {}
}
```

Listing 83.2: header.component.ts

```
<h1>{{title}}</h1>
```

Listing 83.3: header.component.html

Here, we are interpolating the `title` property into our html, by using double curly braces around the value. This is classic syntax in an Angular setting.

The first usage of curly braces in programming languages was by a language called BCPL. It was created by Martin Richards. Instead of the use of `BEGIN` and END implemented by other languages, he used `$( $)`. It was a natural extension of parenthesis, which is all that keyboards offered at that time. In 1967, curly braces, started making their way into mainstream keyboards.
The B Language developed by Ken Thompson, and later adopted the C language, was officially made avialable in 1969. Brackets were already used for arrays. Curly braces was a natural extension to `$( $)`, as it was easier to write, and now available in keyboards. So C, which was a natural extension of the B language, and the gateway to most modern programming languages, used curly braces to group statements(i.e. actions to carried out by programming language).
Angular, in a very similar way uses curly braces to represent statement in it's html. Anything in the double curly braces `{{}}`, can be regular Javascript code. The reason that angular uses a double curly brace, is because:

1. A singular curly brace in Javascript represents an object
2. We would come across the awkward situation, wherein a curly brace in regular html as a string, would require the need for code to be escaped. (Something I've experienced with other programming languages, and it is indeed really awkward.)

83.3 Displaying an Array within an HTML Template

Quite a common occurence within any application, is thay the data, will return an array of objects(AKA collection), to be consumed by the app. Many times there will be a need to iterate over the array with the html, so that the data objects can be accessed.

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'px-code-box',
5    templateUrl: './code-box.component.html',
6    styleUrls: ['./code-box.component.scss']
7  })
8  export class CodeBoxComponent implements OnInit {
9    @Input() cssCode: string[];
10   constructor() { }
11
12   ngOnInit() {}
13 }
```

Listing 83.4: px-code-box.component.ts

```
1 <div *ngFor="let css of cssCode">
2 {{css}}
3 </div>
```

Listing 83.5: px-code-box.component.html

The `*ngFor` works similar to a regular Javascript `for` loop, iterating through every value within the array.

The angular called a for loop in Angular, and ngFor, because it doesn't exactly act like a regular for loop within Angular. Instead, it iterates over every value contained within the array. It's an Angular flavored for loop. `*ngFor` is therefore a perfect name for the angular flavored for loop which iterates through all values.
You might also be wondering what the `*` is doing next to `ngFor` within Angular. The asterisk symbol `*` has a strong history within programming of meaning everything. So what is the everything that the asterisk here is referring to you might ask? Without going into too much detail, the `*ngFor` is actually a micro syntax. It is telling Angular wrap an `<ng-template>` (the syntax angular uses to determine something is a template) around all elements, and populate it with elements within the `<ng-template>`. So the asterisk

* is actually saying, target all templates, and insert this unique Angular directive (i.e. *ngFor).
a b

[a]https://angular.io/guide/structural-directives#asterisk
[b]https://angular.io/guide/template-syntax#ngFor

83.3.1 Accessing Object Data Within Arrays

It is quite common that a situation will arise, that a developer will need to access data within the object. Doing so, is exactly the same way as you would expect within regular Javascript. Let's update our code-box component code.

```
1  import { Component, OnInit } from '@angular/core';
2  import { CssCode } from 'css-code.interface.ts';
3
4  @Component({
5    selector: 'px-code-box',
6    templateUrl: './code-box.component.html',
7    styleUrls: ['./code-box.component.scss']
8  })
9  export class CodeBoxComponent implements OnInit {
10   @Input() cssCode: CssCode[];
11   constructor() { }
12
13   ngOnInit() {}
14 }
```

Listing 83.6: px-code-box.component.ts

```
1  <div *ngFor="let css of cssCode">
2  {{css.file}}
3  </div>
```

Listing 83.7: px-code-box.component.html

As you can see, we are accessing the file property within our css object the same way we would any regular Javascript object.

83.4 Conditionally Display HTML

Within Angular there is the ability to conditionally display html. Similar to an `if` statement within Javascript, the html will only be displayed, if it matches the expected statement.

```
1  <div *ngFor="let css of cssCode">
2  <div *ngIf="css.file as file" class="name">
3  {{file}}
4  </div>
```

In the above code, file will only be shown if it actually exists on the css object.

Note: We are using a syntax called `as`, which allows for us to use a short name for our `*ngIf` block. Will discuss this more as time goes on.

84 Template Syntax

84.1 Difference between Property and Attribute in HTML

First, and foremost, it is important to understand the difference between HTML attributes and properties. I know myself, when writing this chapter, I realized that I did not fully understand the difference between an HTML attribute, and an HTML property. So I thought, why no re-iterate here, and hopefully it will make this subject a bit easier to understand. First let's dive into what the computer science definition of a property, and an attribute would be.

Property - Something that can be read and written. Within a typescript setting, this would be something that would be translated into a `get` and `set` within Typescript.

Attribute - More correctly should be considered as a metadata. Something that is a property of a property, describing what the parent property is doing.

In HTML, the above definitions are a bit obscured. For instance, let's say that we are defining the type of input field, as well as it's value.

```
1 <input type="text" value="Name:">
```

The `type` and `value` are attributes, as they are explaining what the `input` property is doing (aka metadata). However, once the browser parses the code, it will turn it into an `HTMLInputElement` object. This contains dozens of properties, like `className`, `clientHeight`, and methods, such as `click()`. The browser will create a new sort of property based on the type of native html element it is creating.

`<script>` tags in Angular, unlike modern html are forbidden! It will result in an error. This is done to prevent from script injections.

84.2 Property Binding (In Angular)

In Angular, property binding is a way to set properties of a particular element. In addition, it is a way to set `@Input()` decorators set on the actual directive.

```
<img [src]="itemImageUrl">
```

You might be wondering why this is called property binding instead of attribute binding. `src` is actually an attribute. The reason behind1 this, is that Angular's engine will actually first initialize the component, and then change the property set by the browser. In addition, Angular will compile the component, as an object, and therefore internally is setting objects, i.e. properties. I just thought it was a fun fact, and is actually what led me to research the above re: attributes and properties. Another great proof of this, is that if we were to modify the column span for a table by 2 it would be:

```
<tr><td [colSpan]="2">Span 2 columns</td></tr>
```

You will notice that the above syntax for `colSpan` is camel cased as oppoed to the lowercased `colspan`. This is because we are using Angular's internal property binding engine to set the value. It is instead of using the native lowercased attribute `colspan`.

> You might be wondering why Angular decided to choose brackets for property binding. Without delving too much, in Javascript, wrapping an array around a value, signifies that it is property. Angular is just borrowing from that syntax.

84.2.1 Remembering Brackets

If you omit the brackets `[]`, then Angular will treat the value provided as a constant. Which means for the most part, you should be using brackets. There are threee situations wherein omitting brackets make sense.

1. Target property actually uses a string. For instance, the text for the header title.

2. Value of initialization never changes.

84.3 Attribute Binding

As we discussed earlier, there is a very distinct difference between attributes and properties. Setting a property is always preferable simply because the syntax is more straight forward. However, there are many Javascript properties that simply arent't available as properties. A great example of this is SVG's and ARIA labels. In a scenario like this , we can bind to an attribute by doing something like the following:

```
<!-- create and set an aria attribute for assistive technology -->
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

So we can bind to attributes similarly to how we bind to properties. Depending on the nature of your application, it may not, or not happen on a frequent basis.

84.4 Event Binding

Event binding, is an event listening mechanism built into Angular. Whenever something such a keyboard button pressed(keystrokes), mouse moves, clicks etc. it will be able to trigger the appropriate function at that time. The syntax for doing something like this is as follows:

```
1 <button (click)="onSave($event)">Save</button>
```

In the above code, we are telling Angular, when a user clicks on the button emit the method contained within our respective component called `onSave`.

> You might be wondering why Angular chose to use parenthesis for handling event handlers. To me of all the different sorts of syntaxes, it always made the most sense, because parenthesis is remniscient of functions. Event handling always results in functions. So that is what the syntax is telling us. Which function would you like to trigger based on the handling of this event.

84.5 $event

In addition, we are supplying it with `$event` which is Angular's way of internally passing data for that particular component. Depending on the target event, it will change what is returned by `$event`. For instance, if it is a native DOM element event, then `$event` will be a DOM event object. If it is not a native DOM element, and instead something such as an event emitter, then event will contain the value passed on through the event emitter.

84.6 NgClass

`NgClass` is a special way to add, or remove multiple CSS classes at the same time. More importantly, it gives you the ability to add a class conditionally based on whether, or not a value is true.

```
1 <!-- toggle the "special" class on/off with a property -->
2 <button (click)="!isOpen">Toggle</button>
3 <div [ngClass]="isOpen ? 'open' : ''">This div is special</div>
```

In the above code, we are toggling the class active based on the status of the toggle. So when the toggle button is clicked on, `isOpen` will now be true, and as a result, "open" class will now be active. `isActive`.

> The question mark(?) and colon (:), is the standard syntax for ternary operator in Javscript. Meaning if value is true, perform operation after `?` i.e. `'open'`. If value is not true, then perform operation after `:`

i.e. `''`.

84.7 NgStyle

This one is debatable if I want to include in the book. Let's discuss at a later period of time.

84.8 NgModel

Let's look into and see if this is something that we want, and decide if we want to get back to this.

84.9 NgSwitch

`NgSwitch` is Angular's template version of a Javascript `switch` statement. Like switch statements in Javascript, there are very few use cases wherein it makes sense. In addition, in data heavy applications, in general, using something like GraphQL, you will be able to choose what data should come back based on a certain response. However, there are scenarios, where keeping lstinlineNgSwitch in your arsenal will be useful. `NgSwitch` consists of three directives:

1. `NgSwitch` - The switch in which to compare case statements against.
2. `NgSwitchCase` - The case statement, in which to display value, if match is truthy.
3. `NgSwitchDefault` - The default value, if none of the other case values are matched.

```
1 <div [ngSwitch]="code.type">
2   <px-css-item     *ngSwitchCase="'css'"     [item]="currentItem"></px-
      stout-item>
```

```
3   <px-scss-item    *ngSwitchCase="'scss'"      [item]="currentItem"></px-
        device-item>
4   <px-javascript-item      *ngSwitchCase="'javascript'"  [item]="
        currentItem"></px-javascript-item>
5   <!-- . . . -->
6   <px-unknown-item  *ngSwitchDefault           [item]="currentItem"></px
        -unknown-item>
7 </div>
```

84.10 Template Reference Variables

Template reference variables are incredibly useful within an application, and are commonly used within Angular. The reason for this, is that it is referencing the:

1. DOM element within the template
2. Directive/Component
3. `TemplateRef`
4. Web Component

In order to reference a partcular template, use the hash symbol, and a name of your choosing.

```
1 <input #row placeholder="code" />
2
3 <button (click)="initiateGrid(row.value)">Initiate Grid</button>
```

In the above, using our `ref` value, we are able to retrieve the text entered into our input DOM element(i.e. row.value). Based on the scenario, ref values have their use cases.

84.11 Safe Navigation Operator

Many times when retrieving data, it might come back as null, or undefined. In scenario like this, doing the following:

```
1 <p>Hello {{user.name}}! How can we help you today?</p>
```

In the above, if user returns as null, or undefined, the browser will return an error:

```
1 TypeError: Cannot read property 'name' of null.
```

However, if we use the safe navigation operator:

```
1   <p>Hello {{user?.name}}! How can we help you today?</p>
```

(notice the addition of the question mark, appended to our parent variable), Angular will stop evaluating as soon as it hits the first `null` value. It will therefore render without any erorrs.

This also works with longer property paths. For instance:

```
1 <p>Hello {{user?.admin?.name}}! How can we help you today?</p>
```

In any data heavy enterprise app, you can expect to use the safe navigation operator quite often. If you want, you can call it the elvis operator.

85 Modules

Modules are an integral part of Angular. It's interesting, because in some other languages, modules are a part of the language(OCaml comes to mind). A module in a language like OCaml, is any code contained within a file. In Typescript similarly, i.e. the language used with Angular every file is a module. The module exposes objects meant to be public by using the export keyword. In Angular, a module, is completely unrelated to the system of modules usd by Typescript, albeit complementary. It is simply a decorator attached to a class that ultimately get's bundled together into a single class, for use with the app.

85.1 The Module Four, and the Fifth Wheel

Angular provides five key/values to be used with a module:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

1. imports - This takes in other modules, whose exported classes, are needed by this component. This can include, providers, declarations, exports, and imports, contained within the other module.

2. providers - This is used to put all services used by components within this module. [1]

3. declarations - Components, directives, and pipes that belong to the NgModule are put here.

4. exports - The components that should be able to be used in templates outside of this module, when this module is imported by other modules.

[1] It should be noted, that as of Angular 6, the option to use `forRoot` was introduced. It greatly decreases the need for using providers, but it very much so still has it's place in Angular. Especially for unit tests.

5. bootstrap - The main application, i.e, the root component, which hosts all other app views. Only the root NgModule sets the bootstrap property, which is usually handled by the CLI/Nx. However, just to be conscious of it, it is put here. I consider bootstrap as the fifth wheel when it comes to modules for bootstrap.

2

[2] Note this is not put in alphabetical order. Rather it is put in the order that the CLI tends to order them. It is this way, because it is the order in which these items are used within the NgModule declaration.

86 Services

First and foremost what is a service in Angular? In Angular, a service is a way to define business logic in a seperate file, and choose it in our appropriate component when it makes sense to do so. Services, even though they are a part of the core Angular framework, are by definition created in order to alleviate the maintainability and scalability of an application. The bulk of discussion around services will be discussed in later chapters. However, it is still a part of core Angular, and it is important to be aware of couple things.

86.1 Creating a Service and ProvidedIn: Root

In order to create a service, navigate to the folder you would like to create your service, and run:

```
ng g service code-box
```

This will create a service that by default includes `providedIn: 'root'`, It is important to keep in mind, that Angular provides for services only, the ability to add to the constructor something called `providedIn: 'root'`,. In short, it allows for the service to injected, without the need to include it in the respective module. It has some performance boosts, besides level of convenience. We will get more into that in later chapters.

86.2 What Generally Goes in a Service

A service generally deals with data. This means making data requests, and passing that data into our component. Razroo reccomends using Apollo Client instead of Angular's internal http client, due to the fact that we strongly suggest using GraphQL for our application. This is what a typical service would look like:

```
import { Injectable } from '@angular/core';

import { Observable, from } from 'rxjs';
import { pluck } from 'rxjs/operators';
import { Apollo } from 'apollo-angular';

@Injectable({ providedIn: 'root' })
```

```
export class UserService {
  getUser(): Observable<User> {
    const user$ = this.apollo.query({ query: GetCurrentUser });

    return from(user$).pipe(pluck('data', 'getCurrentUser'));
  }
  constructor(private apollo: Apollo) {}
}
```

In the above code, you will notice that we have created a `getUser()` method for our `UserService`. By doing this, we have separated logic from our component, and allow for our code to be more re-usabled. For instance, if we want get the user data, we can simply include the service in the appropriate component, and use the data as needed.

86.3 Including Service In Our Component

If we would like to use this service in our component, we can simply inject the appropriate service in our constructor.

```
// code-box.component.ts
import { Component }   from '@angular/core';
import { User }        from './code-box.interfaces';
import { UserFacade } from './user.facade';

@Component({
  selector: 'px-code-box',
  template: './code-box.component.html',
  styles: ['./code-box.component.scss'],
})
export class HeroListComponent {
  user: User;

  constructor(userFacade: UserFacade) {
    this.user = userFacade.getUser();
  }
}
```

You will notice in the above we included something called the UserFacade. We will get to this in later chapters. Per our architecture, services should always be fed through the facade file. Regardless, for now, what should be kept in mind, is that including service in the contructor, is all that is needed to consume the service.

86.4 Ending Off

I would just like to end off saying that actually creating a service is actually quite simple. It's a piece of architecture, so arguably from an architectural perspective, it is one of the

more complex pieces of Angular. However, to create and start using, is a relatively seamless process.

87 Routing

Routing is an integral part of any single page application:

> "It allows a user to navigate from one view to the next, as a user performs application tasks." [1]

The idea is that routing is that it is it's own internal state machine. There are two things that are unique to state with regards to routing:

1. Data to be pulled in based on page.
2. UI to be shown based on page.

87.1 Base Href

In any Angular application, there is going to be an initial point of entry for routing. First and foremost, in your src/index.html you will need to add an `<base href="/">`. This is added by the CLI by default, and not something you have to worry about.

87.2 RouterModule

Routes in Angular, are singleton instance.

```
1  import { NgModule } from '@angular/core';
2  import { RouterModule } from '@angular/router';
3  import { AppComponent } from './app.component';
4
5  @NgModule({
6    imports: [
7      RouterModule.forRoot( [
8        {
9          path: '',
10         component: AppComponent
11       },
```

[1] Angular Documentation - Routing & Navigation https://angular.io/guide/router

```
12        {
13          path: 'draw',
14          component: PxGridComponent
15        },
16        {
17          path: '**',
18          component: PageNotFoundComponent
19        }
20      ], { initialNavigation: 'enabled' })
21    ],
22  })
23  export class PixelAngstAppRoutingModule {}
```

Listing 87.1: app.module.ts file

As we can see in the above, we are supplying all routes within the app routing module.

87.3 RouterModule Options

Let's break down all the possible options that can be passed to the router:

1. url

```
1  {
2    path: 'draw',
3    component: GridComponent
4  },
```

   Here we are using the url for draw. So, for instance, let's say the url of our application is razroo.com, then razroo.com/draw, will display the grid component.

2. id Many times within our backend, we are going to retrieve data, based on the user's id. Alternatively, it might also be the id for a specific api. Being able to tie in the id for that particular api, into the route is very powerful. Angular routing allows for this to happen:

```
1  {
2    path: 'hero/:id',
3    component: HeroDetailComponent
4  },
```

   The syntax of colon, following by text(does not have to be id), means that if we were to navigate to `razroo.com/draw/123`, it would register within our app, that we want to call the id of '123', within the draw route.

   Within our app, we are going to use this, so that we can use the custom pixel illustrator settings, that our user opted into.

3. data - Allows us to place static data, that we can retrieve that is specific to the route. E.g. page titles, breadcrumb text, other read-only static data.

```
{
  path: 'css',
  component: cssComponent.
  data: { title: 'CSS' }
},
```

4. empty path - An empty path is our default route. I.e. when the app loads for the first time.

```
{
  path: '',
  component: HeroDetailComponent
},
```

5. ** path Two asterisks means that the route is a wildcard. It is particularly advantageous for error reporting:

```
{
  path: '**',
  component: PageNotFoundComponent
}
```

The order of the routes in the configuration matters. This is intentional so that specific routes can be matched first. More generic routes, such as the wild card (**) can therefore be matched, without obstructing other routes.

87.4 Router Outlet

> "Router outlet is a directive from the router library that acts like a component. It marks the spot in the template where the router should display the components for that outlet. " [2]

```
<router-outlet></router-outlet>
<!-- Routed components go here -->
```

Let's say now we were to go to razroo/com/draw, the component for the `draw` route will be placed as a sibling component i.e.

```
<router-outlet></router-outlet>
<px-grid></px-grid>
```

[2] Angular Documentation - Routing & Navigation https://angular.io/guide/router

87.5 Router Links

In order to actually navigate from one route to the next, you will need to use the Angular equivalent of href. However, instead of the classic functionality of href, routerLink, will instead reload the component, based on the new url.

```
<h1>Px Illustrator</h1>
<nav>
  <a routerLink="/draw">Draw</a>
</nav>
<router-outlet></router-outlet>
```

87.5.1 Active Router Links

In addition, Angular offers a way for determining what is the current active link. Something that is very valuable from a UX perspective when the app needs to show to the user, what menu item is currently selected.

```
<h1>Px Illustrator</h1>
<nav>
  <a routerLink="/draw" routerLinkActive="active">Draw</a>
</nav>
<router-outlet></router-outlet>
```

Now if the draw route is triggered, the class active will be added to the ¡a¿ tag. The `.active` class can obviously be styled.

Multiple active router link classes can be added for a particular active route as well:

```
<h1>Px Illustrator</h1>
<nav>
  <a routerLink="/draw" routerLinkActive="'active '">Draw</a>
</nav>
<router-outlet></router-outlet>
```

87.6 Router State

After each successful navigation lifecycle, Angular's internal system updates what's called the `ActivateRoute` object. This can be accessed by using the Router service. Inside of the router service, by accessing the routerState property, we can get to the plethora of properties.

As this is a chapter on fundamentals, we will not delve into all the details now. However, it is important to know that it does exist.

87.7 Router Events

When a route get's triggered, Angular will internally trigger a series of events, from when the navigation starts to where it ends. Angular also exposes these series of events by using the `Router.events` property. Once again, there is no need to go into all of the events, but at this time, they total 17. We will discuss in future chapters situations where the need for accessing the routing events can be beneficial.

88 Forms

Forms to this day, I think is still one of the most complicated pieces of UI, you will ever come across. Each input in a form, has a unique piece of functionality to it. It can be one of the most fustrating pieces of UI, because there is no way to make it truly re-usable as a result. In addition, it can be very fustrating to explain to business as to why it is taking so long, as a form seems truly simple to make. Over the course of this book we will discuss why it is that, that is the case, as well. In addition, we will discuss ways that we can alleviate the pain of forms. In the meantime let's discuss the fundementals of forms.

88.1 Let's Get Something Out of the Way

Angular offers two ways of handling forms:

1. Reactive Forms

2. Template-driven Forms

Whether, or not someone should choose reactive forms, or template driven forms is a large area of debate in an Angular setting. Personally, I believe and this is probably the most debatable point in this book. I strongly believe that if you have chosen Angular as your framework, then you are most likely a medium, sized, or large sized corporation. In which case, you need your forms to be scalable, with the ability to test well. In addition, less prone to bugs. There is a fantastic article by a one Netanel Basel, where he claims that we need to say goodbye to Angular Template-Driven Forms [1]. There is an equally amazing comment left by a one Ward Bell. It sums in totally the following, which I must credit to Deborah Kurata:

88.1.1 Template Driven

1. Easy to use

2. Similar to Angular 1

[1] Why it's Time to Say Goodbye to Angular Template-Driven Forms

3. Two-way data binding -¿ Minimal component code

4. Automatically tracks form and input element state

88.1.2 Reactive

1. More flexible → more complex scenarios

2. Immutable Data Model

3. Easier to perform an action on a value change

4. Reactive transformations → DebounceTime or DistinctUntilChanged

5. Easily add input elements dynamically

6. Easier Unit Testing

Personally, the easier unit testing would be enough for me to go with Reactive Forms. However, there is also the additional reason that Netanel Basel mentions, that very much so resonates with me, and that is readability. The ability to look at a data object with the ts file to see all of the form related items within the app, is simply fanastic.

For that reason, I am simply going to mention Reactive forms within this book. The truth is, is that the Angular documentation it's self sort of says something along the same way. So if you are working on an enterprise app, then I would highly reccomend you do stick with using reactive forms. If you do want to explore the subject of template driven forms on your own, I invite you to go ahead and read through the Angular documenation here. Now that we have established that we will be using Reactive forms, let's get into what they are all about!

88.2 Common Foundation of Forms

1. `FormControl` - Tracks the value and validation state of an individual form control.

2. `FormGroup` - Tracks the same values and status for a collection of form controls.

```
1 import { Component } from '@angular/core';
2 import { FormGroup, FormControl } from '@angular/forms';
3
4 @Component({
5   ...
6 })
```

```
export class CodeEditorComponent {
  codeForm = new FormGroup({
    row: new FormControl(''),
    column: new FormControl(''),
  });
}
```

The codeForm property, now has access to all of the formControls within the app.

3. `FormArray` - Tracks the same values and status for an array of form controls.

4. `ControlValueAccessor` - Creates a bridge between Angular FormControl instances, and native DOM elements.

88.3 Reactive Forms

First and foremost, it's important to get a really get handle on what Reactive means. Reactive means that your code is:

1. Asynchronous - An action happens, and something happens right after. Even though this can happen in regular javascript code, ideally if something is asyncrhonous, it is baked into the framework, making it more of a default.

2. Deterministic - Given action a happens, always action b, will happen. Reactive forms do this, wherein a user can be sure a certain thing will happen granted an input is affected.

88.3.1 Reactive Form Example

Reactive forms are an embodiment of the above, and I would like to show off an example that portays this:

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'px-color-picker',
  template: `
    Background Color: <input type="text" [formControl]="backgroundColor">
  `
})
export class pxColorPickerComponent {
  backgroundColor = new FormControl('');
}
```

In the above code, the data for the form is controlled by FormControl. Here we have attached it to a `backgroundColor` value accessor. Whenever we want to retrieve the value of `backgroundColor`, or modify it, we can tap into the value for `backgroundColor`.

88.4 Reactive Forms - Data Flow

88.4.1 View To Model

In Reactive forms, the data flow is pretty straight forward. As we have already mentioned, that view is directly related to the component's instance of `FormControl` in the component.

1. User types into input
2. Form input emits event with latest value of input
3. Control value accessor [2] listening for events on the form relays value to `FormControl` instance.
4. `FormControl.valueChanges` observable emits new value
5. Subscribers to the value changes observable receive new value.

88.4.2 Model To View

1. User calls setValue method on FormControl(e.g. `backgroundColor`)
2. `FormControl.valueChanges` emits new value
3. subscribers to FormControl.valueChanges recieve new value
4. The control value accessor on the form input element, updates the element with the new value.

88.5 Form Validation

Form validation is something that seems like it would belond as the same category as unit testing. For instance, let's make sure that nothing wrong happens with regards to user

[2] Control value accessor is an internal value within the FormControl

submission. However, form validation, i.e. making sure the right content is being inputted, before user submits, has a very large reason beyond correcting user error. It is to prevent abust my malicious users. Of the ways forms can be abused is:

1. Header injection attacks can be used to send email spam from your web server
2. Cross-site scripting may allow an attacker to post any data to your site
3. SQL injection may corrupt your database backend

88.6 Reactive Form Validation

Angular offers form validation out of the box. In addition, it gives the ability to create custom validators. There are two types of validators:

1. Sync Validators
2. Async Validators

For performance reasons you will only want to use async validators.

88.6.1 Built In Validators

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the
          custom validator.
    ]),
    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });

}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

Let's dissect the above code real quick:

1. We use `Validators.required` and `Validators.minLength(4)` and one custom validator `forbiddenNameValidator`.

2. All these validators are all sync validators, you pass them in as the second argument.

3. You can use multiple validators, by passing in functions as a part of the array

4. We use getters to access values. Even though we can access values by

In addition, we need to add the respective errors within the template, so that we can view the errors if they so arise.

```
<input id="name" class="form-control"
  formControlName="name" required >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
  class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>
</div>
```

One note to just keep in mind, is that we are using the name getter in the components class, to access errors.

88.6.2 Custom Validators

The built in validators, as with anything that is built in, might not always match the exact use case of your application. Within reactive forms, custom validators are simply expressed as functions. For instance:

```
/** A hero's name can't match the given regular expression */
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {'forbiddenName': {value: control.value}} : null;
  };
}
```

This is the function used in the above template. Custom validators are relatively straight forward, and easy to use.

88.7 Testing Reactive Forms

Testing reactive forms is relatively straight forward:

88.7.1 Testing View To Model

```
it('should update the value of the input field', () => {
  const input = fixture.nativeElement.querySelector('input');
  const event = createNewEvent('input');

  input.value = 'Red';
  input.dispatchEvent(event);

  expect(fixture.componentInstance.favoriteColorControl.value).toEqual('
      Red');
});
```

Listing 88.1: Testing View to Model

In the above code we are doing the following:

1. Query view for form input element, and create custom input event for the test.
2. Set the value for the inout, and dispatch input event.
3. Asser that component's new value, based on dispatched event, matches the value from the input.

88.7.2 Testing Model To View

```
it('should update the value in the control', () => {
  component.favoriteColorControl.setValue('Blue');

  const input = fixture.nativeElement.querySelector('input');

  expect(input.value).toBe('Blue');
});
```

In the above code we are doing the following:

1. Set value on the Form Control instance, to set the new value
2. Assign the view for the form input element to a constant
3. Tap into that constant's value, and assert that it is the new value

88.8 Wrapping Up

This would be a general chapter on forms, that dabbled only on reactive programming. In the next chapter, we are going to deep dive into the fundementals on Reactive Programming.

89 Reactive Forms

Reactive forms are extremely under-rated. As we have discussed before, there are many reasons as to why forms are very complicated. Reactive forms are no exception to that rule. However, we will discuss them all now. In additionm we are going to run through all the steps to get from point a to point b, so that you can constantly reference this, as you create a new reactive form component.

89.1 Registering Reactive Forms

Importing a reactive module, is no different than your regular module, however, this is the module to use to when importing reactive forms.

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

89.2 Generating a component, and adding FormControl

No different than any other scenario:

```
ng generate component grid-form
```

In your component simply add a new `FormControl`:

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'px-grid-form',
```

```
  templateUrl: './grid-form.component.html',
  styleUrls: ['./grid-form.component.css']
})
export class NameEditorComponent {
  size = new FormControl('');
}
```

89.3 Registering Control in Template

```
<label>
  Name:
  <input type="text" [formControl]="name">
</label>
```

As mentioned in the previous chapter, FormControl, will allow you to access value of form within component, and view. Most importantly, update in view or component, and have it affect the other.

89.4 Displaying Component

We can now include the component in any other component. E.g.

```
<px-grid-form></px-grid-form>
```

Listing 89.1: app.component.html

89.5 Grouping Form Controls

A FormControl on it's own has value. Primarily being able set a value, and accesing it within the template. However, a `formControl` is incomplete without a `formGroup`. A `formGroup`, will give us access to all of the formControl values, so we can use them all, when submitting a form.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'px-grid-form',
  templateUrl: './grid-form.component.html',
  styleUrls: ['./grid-form.component.css']
})
export class GridFormComponent {
  gridForm = new FormGroup({
    row: new FormControl(''),
    column: new FormControl(''),
```

```
13    });
14  }
```

89.6 Connecting FromGroup model and view

```
1  <form [formGroup]="gridForm">
2    <label>
3      First Name:
4      <input type="text" formControlName="row">
5    </label>
6
7    <label>
8      Last Name:
9      <input type="text" formControlName="column">
10   </label>
11 </form>
```

Within our template, we attach the `formGroup` directive supplied by the `ReactiveFormsModule`, to our component's `new FormGroup`.

89.7 Saving Form Data

The formGroup directive internally has an `(ngSubmit)` method, that can be used to call whenever you are ready to save data for the entire form, and pass it along to the backend.

```
1  <form [formGroup]="gridForm" (ngSubmit)="onSubmit()">
```

Listing 89.2: code-form.component.html

```
1  onSubmit() {
2    // this is where data for gridForm is exposed
3    console.log(this.gridForm.Value);
4  }
```

Listing 89.3: grid-form.component.ts

89.8 Nested Form Groups

In Angular, there is the ability to create nested form groups:

```
1  import { Component } from '@angular/core';
2  import { FormGroup, FormControl } from '@angular/forms';
3
4  @Component({
5    selector: 'px-code-form',
6    templateUrl: './code-form.component.html',
```

```
  styleUrls: ['./code-form.component.css']
})
export class CodeFormComponent {
  profileForm = new FormGroup({
    row: new FormControl(''),
    column: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

Listing 89.4: px-code-form.component.ts

A great way to think of nested form groups, is that it is exactly like creating a nested data object. The only different is that any parent, contains formGroup, whereas children use formControl.

89.9 Grouping nested form in the template

Now that we have created a nested FormGroup property within a class, lets' go ahead and show how we would access these values within our template.

```
<div formGroupName="address">
  <h3>Address</h3>

  <label>
    Street:
    <input type="text" formControlName="street">
  </label>

  <label>
    City:
    <input type="text" formControlName="city">
  </label>

  <label>
    State:
    <input type="text" formControlName="state">
  </label>

  <label>
    Zip Code:
    <input type="text" formControlName="zip">
  </label>
</div>
```

Most notably you will notice that we are using the `formGroupName` directive. The `formGroupName` directive, will sync a `formGroup` to a template. That allows us to access any children of that `formGroup` by simply accessing name, using `formControlName`.

89.10 Partial Model Updates

We have already discussed in the previous chapter, the ability to update a specific `formControl` by using the `setValue()` option is definitely a viable option. However, what if we would like to update multiple values at the same time within a `formGroup()` Angular provides the method called `patchValue()`. So for instance, let's say we wanted to update the `firstName`, and street address:

```
updateProfile() {
  this.profileForm.patchValue({
    firstName: 'Nancy',
    address: {
      street: '123 Drew Street'
    }
  });
}
```

89.11 Generating form controls with FormBuilder

The Angular team realized how excessive it is to constantly include `FormControl`'s and `FormGroup`'s everytime one wants to build out a form. Instead they created something called the `FormBuilder`.

```
  import { Component } from '@angular/core';
  import { FormBuilder } from '@angular/forms';

  @Component({
    selector: 'app-profile-editor',
    templateUrl: './profile-editor.component.html',
    styleUrls: ['./profile-editor.component.css']
  })
  export class ProfileEditorComponent {
    profileForm = this.fb.group({
      firstName: [''],
      lastName: [''],
      address: this.fb.group({
        street: [''],
        city: [''],
        state: [''],
        zip: ['']
      }),
    });

```

```
21      constructor(private fb: FormBuilder) { }
22    }
```

As we can see in the above, instead of having to attach `FormGroup` to the parents, and attaching `FormControl` to every child, we can just use `fb.group` on every parent. So, so nice!

90 Attribute Directives

An attribute directive changes the appearance, or behavior of a DOM Element. It is tagged on of an html element to change the way it works. For instance, while it is probably better to use CSS in this situation, let's create a really low level directive, to introduce how it works:

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Now we have the ability to apply this directive to out html element:

```
<p appHighlight>Highlight me!</p>
```

As a result of the directive we have applied on this p element, the background for this p element will now yellow.

It, of course, can be incredible useful.

> A @Directive is not to be confused with a @Component. A @Component requires html, wherein a @Directive does not use html. A @Directive is meant to be tagged onto html, and modify it's behavior.

90.1 A Great Example

A great example of this, is adding drag and drop functionality to a component. That in it's self is not component worthy. However, adding a dropzone directive for instance, to the element, would make any potential component drag and droppable worthy. A

A great example of a component, would be a data table. A large part of it's functionality is strickly tied to actual UI element.

A side note with regards to our architecture, directives will be very used. If you find yourself writing, with what feels like too many components, don't worry, it's to be expected.

90.2 Passing Values into the Directive

A directive has the ability to pass a value in. For instance, going back to our highlight example, let's create an @Input() (Angulars way of passing in values) for our highlight directive.

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() highlightColor: string;

  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = this.highlightColor';
  }
}
```

Now if we were to go back to our template, we have the option to insert the color we want within the template:

```
<p appHighlight='orange'>Highlight me!</p>
```

The background of this component is going to be orange!

90.2.1 Passing in Multiple Values

Passing in multiple values is as simple as adding a second @Input value to our Angular Directive:

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() highlightColor: string;
  @Input() defaultColor: string;

  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = this.highlightColor';
```

```
  }
}
```

Just like that we can now pass multiple values to our html element:

```
<p appHighlight="orange" defaultColor="blue" >Highlight me!</p>
```

Angular knows once the appHighlight directive has been exposed, that it has the input of defaultColor, or any other Input you might add for that matter.

90.3 Modify Values Based on Events

Directives also give the option to modify based on an event. For instance, we could add logic based on mouseenter.

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() highlightColor: string;

  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = this.highlightColor';
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor);
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Now let's say we add this directive to our ¡p¿ tag.

```
<p appHighlight highlightColor="yellow">This will be Highlighted in
   Yellow on
mouseenter. Good times!</p>
```

90.4 Examples of Directives

I think at this point, after just getting used to directives, it can be a bit difficult to internalize what an example of a directive looks like. I would therefore just like to jot down real quick, what an example of some custom directives would look like:

1. trim-whitespace.directive.ts - A directive for trimming extra whitespace from input fields. It would include an onChange event for whenever value is changed. It would also include an onTouched event, so that whenever user clicks on input, it will trim any of the text.

2. infinite-scroll.directive.ts - An infinite scroll directive. It would attach its self to a container, and allow for it to make Graphql reqeusts, whenever user scrolls beyond height of the container.

3. copy-to-clipboard.directive.ts - Allow for the ability of automatically copying text to the clipboard.

4. go-back.directive.ts - Allows for the ability to attach the capability to click on any button and go back to the previous page.

91 Pipes

Angular offers the ability to use Pipes out of the box. The idea behind a pipe is to get data, transform it, and show new transformed data to users. Pipes are something that I am going to assume the reader is already familiar with. Just as a primer, using the native angular date pipe, a pipe would do something such as the following:

```
1 The chained hero's birthday is
2 {{ birthday | date | uppercase}}
```

This would display

```
FRIDAY, APRIL 15, 1988
```

The pipe here is taking in the timestamp of

```
577065600
```

and converting it to the proper date. Having such dates in one's application obviously makes it very easy to go ahead and transform data throughout one's app. Here we are also chaining pipes, so that the transformed data that comes back in addition to being transformed, is also capitalized.

91.1 Performance Considerations

It is important off the cuff to be aware of some performance concerns when it comes to pipes.

91.1.1 Understanding Angulars Change Detection

This is a good time to interject and get into the nitty gritty of angular's change detection. As we discussed in the earlier chapter on change-detection, change detection in Angular works from the top down. That is, if a specific set of data changes within a component,

then the entire component will update as a result of new data. Angular pipes, however, change how that it is done, by changing content directly on the object, and only updating that one specific part. By using pipes, it allows one's component to be more performant.

91.2 When to Use Pipes

Pipes cover alot of ground. Within our architecture, one of the pipes that will be used more so that others in the async pipe. However, now is not the place to put that here. What is important, is that pipes can be used transform data. Should they always be used whenever one is transforming data? Personally, I like to think of pipes as unique to html. They have a way of dealing of performance when working with html templates. However, within a component itself it is questionable. To be honest this can go either way. However, from a maintainability perspective, whenever data is being transformed, it should be turned into a pipe, instead of a service. That would be the simple rule.

In addition, there are impure pipes that one can potentially do, which ususally has little to do with transforming data. Once again the async pipe is something that would register along these lines. However, I have not seen the need to create an impure pipe with the architecture given.

92 Observables

Observables are an integral part of any Angular application. There is no way around not using observables if using state management within your application. In addition, if your backend is using GraphQL + Apollo, which is the reccomended way of pulling in data, your application will have to implement observables. The reason observables are so popular and it has made it's way into mainstream Angular, is because it is so intuitive to use. An observable will consist of two things:

1. Publisher

2. Subscriber

Think of a publisher as a way of saying here is the data, I want you to watch. When something does indeed change, pull the data from the subscriber. This allows for the publisher to take place in one piece of your code, pass that through a facade of sorts, and then consume it in another file.

92.1 Rxjs - An Observable Example!

We will be using `rxjs` as our library to create Observables. It is used internally by many Angular libraries, and for good reason. For starters It is increcibly intuitive to use. Functions are imported as are any other Javascript/Typescript library. Second, rxjs, is a part of the larger ReactiveX ecosytem. Knowing how to use rxjs, you will be able to transfer your knowledge to other frameworks, such as Java, Swift, Python and more!

```
1 import { from } from 'rxjs';
2
3 // Create an Observable out of a promise
4 const data = from(fetch('/api/endpoint'));
5 // Subscribe to begin listening for async result
6 data.subscribe((data) => {
7   // emit data returned from endpoint
8  console.log(data);
9 });
```

from - Turn an array, promise or iterable into an observable

In the above code, we are using the native rxjs `from` function to turn our code into an observable(this is what we defined earlier as a publisher). We then subscribe to the publisher/observable, so that when the data request completes, and the data is pulled in, the subscriber will now emit the data, and make it available for the app to consume.

92.2 Operators and Pipes

One of the most powerful features of rxjs, is that it offers the ability to combine operators to allow for sophisticated manipulation of functions. While this is not a real world example, it definitely helps in order to better understand how pipeable operators work in the real world:

```
1  import { filter, map } from 'rxjs/operators';
2
3  const nums = of(1, 2, 3, 4, 5);
4
5  // Create a function that accepts an Observable.
6  const squareOddVals = pipe(
7    filter((n: number) => n % 2 !== 0),
8    map(n => n * n)
9  );
10
11 // Create an Observable that will run the filter and map functions
12 const squareOdd = squareOddVals(nums);
13
14 // Subscribe to run the combined functions
15 squareOdd.subscribe(x => console.log(x));
```

of - Similar to `from`, only `of` takes values only, instead of being passed an array. filter - Similar to filter in javascript. Will return all values that pass the provided condition. map - Similar to map in Javascript. Will apply projection with each value in source.

In the above code, we are using two separate operators, and chaining them within our pipe. First we pull in odd values using the filter, then we multiple files using the map. This can be an extremely powerful way of using chained operators within rxjs. In particular, in scenarios like search rxjs can be extremely powerful.

92.3 Common Operators

Rxjs is full of numerous operators. However, there really are only a certain amount of operators that are used.

Area	Operators
Creation	from, fromEvent, of
Combination	combineLatest, concat, merge, startWith , withLatestFrom, zip
Filtering	debounceTime, distinctUntilChanged, filter, take, takeUntil
Transformation	bufferTime, concatMap, map, mergeMap, scan, switchMap
Utility	tap
Multicasting	share

92.4 Naming Conventions for Observables

A very popular convention for writiing observables within Angular, and really any setting is add a trailing `$` to the end of the variable.

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
  selector: 'app-stopwatch',
  templateUrl: './stopwatch.component.html'
})
export class StopwatchComponent {

  stopwatchValue: number;
  stopwatchValue$: Observable<number>;

  start() {
    this.stopwatchValue$.subscribe(num =>
      this.stopwatchValue = num
    );
  }
}
```

In the above component, `stopwatchValue$` is visibly an observable, because it has the dollar sign. Why a dollar sign you ask? It's a clever way of appending an "S" to the end of a variable, and signifying that it is special. The s stands for stream - which is a sequence of values over time.

93 Angular Observables

It's important to point out, that there are observables that are unique to Angular. It is important to be aware of the fact that internally they are using an observable.

93.1 Event Emitter

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

```
@Component({
  selector: 'zippy',
  template: `
  <div class="zippy">
    <div (click)="toggle()">Toggle</div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div>
  </div>`})

export class ZippyComponent {
  visible = true;
  @Output() open = new EventEmitter<any>();
  @Output() close = new EventEmitter<any>();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

93.2 Async Pipe

The async pipe, which we will discuss more due to it's high usage within the app.

```
@Component({
  selector: 'async-observable-pipe',
  template: `<div><code>observable|async</code>:
       Time: {{ time | async }}</div>`
```

```
})
export class AsyncObservablePipeComponent {
  time = new Observable(observer =>
    setInterval(() => observer.next(new Date().toString()), 1000)
  );
}
```

The async pipe will subscribe to an observable(,or promise) and returns the latest value it has emitted. When new value has been emitted, the pipe marks the component to be checked for changes.

93.3 Router

93.3.1 Events

Router events are supplied as an observable. So let's say we want to listen in, into when a router event has reached a certain point we would be able to do that.

```
import { Router, NavigationStart } from '@angular/router';
import { filter } from 'rxjs/operators';

@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable1Component implements OnInit {

  navStart: Observable<NavigationStart>;

  constructor(private router: Router) {
    // Create a new Observable that publishes only the NavigationStart
       event
    this.navStart = router.events.pipe(
      filter(evt => evt instanceof NavigationStart)
    ) as Observable<NavigationStart>;
  }

  ngOnInit() {
    this.navStart.subscribe(evt => console.log('Navigation Started!'));
  }
}
```

93.3.2 ActivatedRoute

`ActivatedRoute` "contains the information about a route associated with a component loaded in an outlet." Specifically, one of the pieces of information that the `ActivatedRoute` injected

router service provides is `ActivatedRoute.url` which is provided as an observable.

```
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-routable',
  templateUrl: './routable.component.html',
  styleUrls: ['./routable.component.css']
})
export class Routable2Component implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) {}

  ngOnInit() {
    this.activatedRoute.url
      .subscribe(url => console.log('The URL changed to: ' + url));
  }
}
```

Using the above observable, we are able to determine what the url is at any given time.

93.4 Reactive Forms

Reactive forms, is another core Angular library that makes use of Observables. In particular, the `FormControl valueChanges` property contains an observable that determines whenever an event occured.

```
import { FormGroup } from '@angular/forms';

@Component({
  selector: 'my-component',
  template: 'MyComponent Template'
})
export class MyComponent implements OnInit {
  pxForm: FormGroup;

  ngOnInit() {
    this.logNameChange();
  }
  logNameChange() {
    const rowControl = this.pxForm.get('name');
    rowControl.valueChanges.subscribe(data => {
      console.log('data');
      console.log(data);
    });
  }
}
```

Using valueChanges in the context of formControl can be incredibly useful. There might be special effects, or some sort of hurdle you need to come across when building out this form, and it helps very much so in this regard.

Important to note, that http requests within Angular also support Observables. However, Razroo best practices, is that we support the use of Apollo Client over Angular's native http request. For that reason, it has been left out of this chapter.

93.5 What is the Point of this???

You might be asking yourself, what is the point of knowing specifically that there are observables that are baked into the framework? It would seem one would be able to just read up on the documentation, and be able to get done what you need in that fashion. What value is there in organizing all of the Angular observables into one location?

I think this chapter makes alot of sense. In particular, because it lays out all the parts of the Angular framework, wherein observables make sense. Very smart, and well thought out individuals have built, and continue to build the Angular framework. Therefore, it would stand to reason, that one should be particularly aware of these four scenarios:

1. Event Emitter
2. Async Pipe
3. Router
4. Forms

In all of them, being aware of the value at any given time, is incredibly important. If something has an observable it means:

1. Level of complexity is higher
2. Subject to change, and needs higher level of architecture
3. Interacts with data heavily outside of component instantiation

So keep an eye out for the above, next time you work on your Angular application.

94 Animations

Animations have come a long way. It's one of the most jaw dropping effects that any application can have. Angular has made it, so that animations can be baked into the framework. I would like to run through how one would go ahead and implement a simple animation within your component, so that you can get an idea of how you can do it.

94.1 Include Animations Module

Angular has a `BrowserAnimationsModule`, which is based off of the `BrowserModule`.

```
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { BrowserAnimationsModule } from '@angular/platform-browser/
       animations';
4
5  @NgModule({
6    imports: [
7      BrowserModule,
8      BrowserAnimationsModule
9    ],
10   declarations: [ ],
11   bootstrap: [ ]
12 })
13 export class AppModule { }
```

Listing 94.1: app.module.ts

94.2 Importing animation functions into component files

Just something to be aware of. Angular has many animation functions that can be used out of the box. We are not going to go into detail on them all now. However, I just wanted to bring some up here, for you to be aware of:

```
1  import { Component, HostBinding } from '@angular/core';
2  import {
3    trigger,
4    state,
5    style,
```

```
6     animate,
7     transition,
8     // ...
9   } from '@angular/animations';
```

Listing 94.2: app.component.ts

```
1
```

94.3 Add the animation metadata property to component

The next step in the adding an animation to a component process, would be adding an animations metadata [2] property.

```
1  @Component({
2    selector: 'app-root',
3    templateUrl: 'app.component.html',
4    styleUrls: ['app.component.css'],
5    animations: [
6      // animation triggers go here
7    ]
8  })
```

94.4 Animation State, Styles, and Transitions

With regards to animations, there are three main functions to keep in mind,

1. `state()` - Function to define different states to call at the end of each transition. It takes two arguments:

   a) A unique name like `open`, or `closed`

   b) A `style()` function

```
1  state('open', style({
2    height: '200px',
3    opacity: 1,
4    backgroundColor: 'yellow'
5  })),
```

2. `style()` - A function used to assign a set of styles for a given state name. Style attributes must be camelCase.

[1] For a summary of available animation functions, feel free to navigate here
[2] Clarify what a metadata property is

```
style({
  height: '200px',
  opacity: 1,
  backgroundColor: 'yellow'
})),
```

3. `transition` - Used to specify the change that occurs between one state and another over a period of time. Accepts two arguments:

   a) An expression that defines the direction between the two transition states.

   b) An `animate` function

```
transition('open => closed', [
  animate('1s')
]),
```

94.5 Triggering the animation

Every animation requires a `trigger`, so that it knows when to start. The `trigger()` function collects state, style, and transitions, and gives it a name, so that it can be triggered in the html template.

```
@Component({
  selector: 'px-open-close',
  animations: [
    trigger('openClose', [
      // state, style, and transition goes here
    ]),
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }
}
```

Listing 94.3: open-close.component.ts

We can now add this transition in our html, following the general syntax:

```
<div [@triggerName]="expression">...</div>;
```

Listing 94.4: *.html

94.6 Putting it all together

If we were to combine our animations, with our combined `state`, `style`, and `transitions`, it would look something like the following:

```
@Component({
  selector: 'px-open-close',
  animations: [
    trigger('openClose', [
      // ...
      state('open', style({
        height: '200px',
        opacity: 1,
        backgroundColor: 'yellow'
      })),
      state('closed', style({
        height: '100px',
        opacity: 0.5,
        backgroundColor: 'green'
      })),
      transition('open => closed', [
        animate('1s')
      ]),
      transition('closed => open', [
        animate('0.5s')
      ]),
    ]),
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }
}
```

Listing 94.5: open-close.component.ts

```
<div [@openClose]="isOpen ? 'open' : 'closed'" class="open-close-
    container">
  <p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>
</div>
```

Listing 94.6: open-close.component.html

95 Transitions and Triggers

It is worth noting that there are a number of transition states. We discussed from open to close, and from close to open. There are, however, other states that are equally as important to discuss.

95.1 Wildcard Matching

Using an asterisk in a transition state, will represent any sort of situation.

For instance, let's say were to use:

```
open => closed
open => *
* => closed
* => *
```

All of the above four, will match when an element's state changes from open to anything else. A good rule of thumb, is that similar to routing, wherein the asterisk will be a wild card meant to match any remaining use cases, animation transitions also follow in the same vein.

95.2 Situations where a Wildcard can be Used

95.2.1 Using Wildcard with Styles

```
transition ('* => open', [
  animate ('1s',
    style ({ opacity: '*' }),
  ),
]),
```

In the above, the transition of animate of 1s, will match whatever it is that the current stule is.

95.2.2 Combining Wildcard and Void States

A void state, is a way of causing an animation to occur whenever an element is entering, or leaving a page. The proper syntax for letting Angular knowing that an element should be animated when leaving, or entering would be to do the following:

```
animations: [
  trigger('flyInOut', [
    //...
    transition('void => *', [
      //...
    ]),
    transition('* => void', [
      //...
    ])
  ])
]
```

95.2.3 :enter and :leave aliases

However, Angular allows for the following alias, called `:enter` and `:leave`.

```
animations: [
  trigger('flyInOut', [
    state('in', style({ transform: 'translateX(0)' })),
    transition(':enter', [
      style({ transform: 'translateX(-100%)' }),
      animate(100)
    ]),
    transition(':leave', [
      animate(100, style({ transform: 'translateX(100%)' }))
    ])
  ])
]
```

In the above code, when an `HTML` element isn't attached to the view, we apply a transition. When entering the page, the element will fly in. When leaving the page, the element will fly out.

It is important to note that `:enter` and `:leave` will only run, if the element is removed, or added. Therefore, it is required to leave an `*ngIf` on the div. An example, would be something such as the following:

```
<div @myInsertRemoveTrigger *ngIf="isShown" class="insert-remove-
    container">
  <p>The box is inserted</p>
```

```
</div>
```

Listing 95.1: insert-remove.component.html

```
trigger('myInsertRemoveTrigger', [
  transition(':enter', [
    style({ opacity: 0 }),
    animate('5s', style({ opacity: 1 })),
  ]),
  transition(':leave', [
    animate('5s', style({ opacity: 0 }))
  ])
]),
```

Listing 95.2: insert-remove.component.ts

We do not need to use `state`, due to the fact that these transitions are expected to happen in all scenarios.

95.3 When a Value Increases, or Decreases

Angular has a built in transition for when a value increases, or decreases.

```
  trigger('filterAnimation', [
    transition(':enter, * => 0, * => -1', []),
    transition(':increment', [
      query(':enter', [
        style({ opacity: 0, width: '0px' }),
        stagger(50, [
          animate('300ms ease-out', style({ opacity: 1, width: '*' })),
        ]),
      ], { optional: true })
    ]),
    transition(':decrement', [
      query(':leave', [
        stagger(50, [
          animate('300ms ease-out', style({ opacity: 0, width: '0px' }))
              ,
        ]),
      ])
    ]),
  ]),
```

If the above, when the value is incremented, it will cause a transition to happen.

You might be wondering what would be the value of something like this within an Angular application setting. The value is that if there is something like a slider, you are able to bake transitions, into the counter of the slider. Now, whenever the slider goes to the

next slide, it can cause a transition to happen.

95.4 Transitions for Boolean Values

In many situations, we have the need for, should something be opened, or closed. Should it be shown, or hidden? For situations like this, Angular offers the ability to create transitions based on booleans:

```
<div [@openClose]="isOpen ? true : false" class="open-close-container">
</div>
```

Listing 95.3: open-close.component.html

and

```
animations: [
  trigger('openClose', [
    state('true', style({ height: '*' })),
    state('false', style({ height: '0px' })),
    transition('false <=> true', animate(500))
  ])
],
```

Listing 95.4: open-close.component.ts

95.5 Multiple Animation Triggers

In an Angular setting, each time an animation is triggered, the parent always get's priority, and cuts off the ability for child animation to run. In order for a child animation to run, a parent element must trigger an `animateChild` function placed on the child element.

```
<div [@.disabled]="isDisabled">
<div [@childAnimation]="isOpen ? 'open' : 'closed'"
  class="open-close-container">
  <p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>
</div>
</div>
```

Listing 95.5: open-close.component.html

```
@Component({
  animations: [
    trigger('childAnimation', [
      // ...
    ]),
  ],
})
```

```
export class OpenCloseChildComponent {
  isDisabled = false;
  isOpen = false;
}
```

Listing 95.6: open-close.component.ts

In the above html we are using something called `@.disabled`. When it is applied on the parent div, all animations on the element, as well as nested animations, are disabled. Likewise, to disable all animations for an Angular app, a hostBinding can be applied to the top most Angular component.

95.6 Animation Callbacks

There are times wherein one might want to tap into a particular time period of an animation. For instance, let's say there is a slow API request, you might want the download button to have some sort of pulsating animation, if it takes too long. When API request completes, the pulsating can stop. An icon can work as well, however, there is some psychology behind if the actual item clicked on is the one that animates, as opposed to bringing in an outside icon. So let's imagine we are triggering an animation, we can do something such as the following:

```
@Component({
  selector: 'app-open-close',
  animations: [
    trigger('openClose', [
      // ...
    ]),
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  onAnimationEvent ( event: AnimationEvent ) {
  }
}
```

Listing 95.7: open-close.component.ts

```
<div [@openClose]="isOpen ? 'open' : 'closed'"
  (@openClose.start)="onAnimationEvent($event)"
  (@openClose.done)="onAnimationEvent($event)"
  class="open-close-container">
</div>
```

Listing 95.8: open-close.component.html

95.6.1 Debugging Animations using Callbacks

Let's imagine in the above `animationEvent`, if we were instead to use cram our animationEvent with the following:

```
export class OpenCloseComponent {
  onAnimationEvent ( event: AnimationEvent ) {
    // openClose is trigger name in this example
    console.warn(`Animation Trigger: ${event.triggerName}`);

    // phaseName is start or done
    console.warn(`Phase: ${event.phaseName}`);

    // in our example, totalTime is 1000 or 1 second
    console.warn(`Total time: ${event.totalTime}`);

    // in our example, fromState is either open or closed
    console.warn(`From: ${event.fromState}`);

    // in our example, toState either open or closed
    console.warn(`To: ${event.toState}`);

    // the HTML element itself, the button in this case
    console.warn(`Element: ${event.element}`);
  }
}
```

Listing 95.9: open-close.component.ts

In the above, we can now view exactly what is happening within our animation and can use this to determine animation if anything is awry.

95.7 Keyframes

Many animations happen to have a simple two step solution. For instance, show and hide component, or maximize height. However, there might be an animation that might be more than two steps. For animations of this scale, we will want to use Angular's `keyframe()` function, which is very similar to keyframes in CSS [1].

If using default keyframes, it will automatically split the different times, across the time frame.

```
transition('* => active', [
  animate('2s', keyframes([
    style({ backgroundColor: 'blue' }),
    style({ backgroundColor: 'red' }),
    style({ backgroundColor: 'orange' })
  ]))
```

[1] Keyframes in CSS can be seen here

In the above code, initially the `backgroundColor` will be blue. From 0s to 1s, the `backgroundColor` will be red. From 1s to 2s the `backgroundColor` will be orange.

95.7.1 Offset

Angular also gives the option to define at which point in the keyframe the animation should occur. Let's take the previous default animation and supply it with offsets:

```
transition('* => active', [
  animate('2s', keyframes([
    style({ backgroundColor: 'blue', offset: 0}),
    style({ backgroundColor: 'red', offset: 0.8}),
    style({ backgroundColor: 'orange', offset: 1.0})
  ])),
]),
transition('* => inactive', [
  animate('2s', keyframes([
    style({ backgroundColor: 'orange', offset: 0}),
    style({ backgroundColor: 'red', offset: 0.2}),
    style({ backgroundColor: 'blue', offset: 1.0})
  ]))
]),
```

It is important to note, that any animatable property that can be animated using CSS transitions, can be animated using Angular.

95.8 Automatic property calculation with wildcards

Many times, we are not 100% aware of what the height will be of the component that we would like to animate. Angular allows us to apply a wild card styling to the element, so that the height is determined at run time. Therefore, if we want, we can apply something like the following, so that we can determine the height at runtime, and then shrink the element when appropriate.

```
animations: [
  trigger('shrinkOut', [
    state('in', style({ height: '*' })),
    transition('* => void', [
      style({ height: '*' }),
      animate(250, style({ height: 0 }))
    ])
  ])
]
```

96 Track By

In Angular, there are performance enhancements that are valuable here and there. It can be easy to miss them, because they are performance ehancements. So they are not integral towards the core documentation. However, for anyone building an application, these are things that one should be aware of.

In particular, when it comes to for loops for Angular, the way that change detection works, is that if any part of the array is changed, Angular will agressively change the entire DOM. This is of course can lead to performance issues, if all we need is a particular piece of data to be changed. The Angular framework has an internal `trackBy` function to combat these performance leaks.

96.1 Using Track By in Angular

Using `*ngFors` in Angular is actually quite common. I cannot think of one application that doesn't use a data table of some sort. Suffice to say,an ngFor that parses through alot of data, as well as optimizing it for performance reasons, is very important. Even it's an application that has pagination through the backend, and we only allow a maximum of 50 rows at a time.

96.1.1 So What is trackBy?

What trackBy does is check the set as a whole for order changes (sorting, insertion, deletion). When the order changes, instead of removing all elements from the DOM and creating new ones, the trackBy function is used to identify which elements do not need to be removed from the DOM. This reduces the number of DOM calls, and also reduces the number of angular digest cycles.

Under the hood `trackBy` tracks an order change. It does this by taking in index of each item, and a unique identifier of that item. If item does not exists, from prior id's and indexes, then it knows to insert that one particular item into the DOM.

96.1.2 Track By Within Data Tables

Specifically, within our architecture, we call for using Material. By default, if a trackBy function is not given, Material Table will deeply compare the elements in the set. So, a trackBy function is really used to reduce the amount of checks necessary to compare elements in a set. Instead of a deep copy, you can check for a single unique property.

96.2 Track By in Practice

```
1   <mat-table
2   [trackBy]="trackByBuyerId">
3   <mat-table/>
```

Listing 96.1: data-table.component.html

```
1 //.. this code is inside of our component class
2 trackByBuyerId(index, item) {
3   return item.id; // unique id corresponding to the item
4 }
```

Listing 96.2: data-table.component.ts

By doing the above, track by will in a material setting make sure that objects in the data table are not deeply compared, comparing objects directly to each other.

> This, of course, is equally as effective in a non Angular Material setting. However, I just wanted to bring up an enterprise setting, and this is most definitely one.

97 Bundle Size

Bundle sizes ironically, are one of those things that tend to get overlooked. It's ironic, because it can make such a large difference to a user, using the app for the first time, while simaltaneously being one of the easier things to tackle. I think for that reason it tends to not get the attention it needs, because it doesn't exactly fit into the box of computer science.

97.1 Being Aware of Bundle Size

When you run `ng build --prod`, Angular will generate 4 files:

1. runtime.*.js
2. main.*.js
3. polyfills.*.js
4. styles.*.css

Of the above, the largest will be main.js files. If we dissect our bundle size, it's important to keep in mind that this bundle size will be contributed to by a number of different things. This chapter won't go into how to decrease the bundle size before it is bundled, but rather after it has been bundles.

97.2 Gzip

gzipping for those that are not familiar, is the process of taking a chunk of data and making it smaller. The original data can be restored by un-zipping the compressed file. Within the context of HTTP protocol, it has the ability to unzip a file. There is a bit of a cost from the side of the browser to unzip a file. However, generally the benefit of a lower bundle size, outweighs the fact that the browser will have to unzip the file. All that is needed, is for your devops person, to set the gzip setting on your server.

As a general rule of thumb, gzipped files are about 20% the size of the original file, which of course, will drastically decrease the initial load time of your app. If you would like to check and see whether, or not your files are gzipeed, you can simply open up your console, and check the "Content-Encoding" under the Response Headers. If it says "Content-Encoding: "gzip", then you are in good hands, otherwise, you might be in trouble.

97.2.1 How to Gzip

Gzip'ing is something that should be controlled by your CDN, which will also deal with a slew of other things to make sure your files are served. However, your backend team will be able to deal with this one relatively quick, if they are not using a CDN system, due to many packages out of the box dealing with this.

97.3 Analyze Your Angular Bundle

Webpack has a built in tool that one can use to analyze build. For starters, it has an incredible visualization of what your entire build looks like. [1] In addition, it will tell you about things such as:

1. You forgot to remove some packages that you aren't using anymore.
2. Some packages are larger than expected, and can be replaced with another.
3. You have improperly imported libraries.

In order to get the above data:

1. `npm install -g webpack-bundle-analyzer`

   Then, in you Angular app, run:

2. `ng build --stats-json`

   We are going to be running stats on the original non minified build, as we want to make sure we do not muddy the results coming back from out stats.

3. Finally we run:

   ```
   webpack-bundle-analyzer path/to/your/stats.json
   ```

[1] Insert photo here on snapshot.

and navigate to `localhost:8888` wherein we will see our results.

97.4 Monitoring Bundle Size

In Angular 7, and later, angular adds a config in your `angular.json` file. It has a property called "budgets", and it looks like this:

```
"budgets": [
  {
    "type": "initial",
    "maximumWarning": "2mb",
    "maximumError": "5mb"
  }
]
```

This will throw an error if bundle size exceeds 2mb, and throw an error if bundle size exceeds 5mb. It is very much so possible to use this feature in your CI/CD pipeline. Feel free to go over to your devops person, and ask them to integrate it.

98 Image Performance in Angular

Image optimization is a concept which is ubiquitous to all of web development. Nonetheless, it is a bit unique of how to work it into an Angular setting. Primarily, because of Angular's use of routing for lazy loading, file bundling, and modules. It is also worth noting, in many enterprise applications, the types that are better suited for Angular, they are more data heavy, and less image heavy. Image optimization isn't something I've come across frequently in Angular applications, albeit something still very important to discuss.

In addition, the dynamics of image optimization, is that this is something that can be discussed at a later date. Having this conversation might only be neccesary when you are finally ready to deploy application to prod. However, this is something that can be done ahead of time, with minimal effort. It only requires the following two steps:

1. Aware of it.

2. Implementing it is made easy to implement in less than 3 steps

That is what we are going to attempt to do.

98.1 Lazy Loading Images

Lazy loading, if not already familiar, is the idea of loading an image only when neccesary. It is powerful primarily, because, non-critical images are loaded when needed, allowing for a quicker load of content on page load. It is also useful for the additional following reasons:

1. It reduces data consumption, i.e. less work for the server side.

2. Less workload for the browser.

3. Improve webpage loading time.

great way to see what images might need optimization, is to use Lighthouse in Chrome DevTools to see what might need changing.

98.2 Using lazysizes for Loading Images

`lazysizes`, in my opinion is the most robust library for making use of lazy loading libraries. Some of the benefits of using it as a library include:

1. Requires no configuration.
2. test
3. Fantastic Performance.
4. Optional integration with Intersection Observer. [1]
5. Supports plugins, which include those such as the object-fit extension, effect plugin, and respimg polyfill plugin.

98.3 Angular Directive with Lazysizes

98.3.1 Adding Lazysizes

[1] It's a relatively modern api that provides a way to asynchronously observe changes in the intersection of a target element.

99 Modern Script Loading

100 Ahead Of Time Compilation

Angular as a framework uses Typescript. In addition, it uses directives, components, `@Input()`'s and `@Output`'s. So, when one thinks about it, you start to realize, "hey! Is Angular compiling it to browser ready HTML and Javascript for me"? The answer to this question, is yes. How Angular does that, cannot be entirely controlled, as that is handled by the internals of the framework. However, the real question that we should be asking ourselves, is when does this happen.

100.1 Exploring Ahead of Time Compilation

Without using the ahead-of-time compilation flag `--aot` i.e.

```
ng serve
```

Angular will convert the code from Angular code, to browser readable code at run time. By using the `--aot` compiler i.e.

```
ng serve --aot
```

The Angular compiler will compile the code at build time. The benefits of compiling the code at build time are as follows:

1. Faster rendering - Browser doesn't have to compile code first
2. Fewer Requests - The compiler will inline external html and css.
3. Smaller bundle sizes - Angular Compiler not included in bundle size.
4. Template errors - They happen inside of compiler, instead of at runtime.
5. Better security - HTML and Components(css included), are compiled into Javascript before the webpage loads. Preventing against many different types of injections.

The internals are somewhat complex, and the Angular documentation does a really good job at discussing what that is. From a practical perspective, let's discuss the expression limitations. I will admit, at this time, why there are these limitations is beyond me.

100.2 Expression Syntax Limitations

The AOT collector only understands a subset of Javascript. It's quite a long list. Therefore, when using the `--aot` flag, and the AOT compiler comes across something it doesn't understand, it will throw the error into the `.metadata.json`. Later on, if it needs that piece of code to generate the application code, the compiler will complain. (However, I have personally found that there are scenarios wherein the AOT compiler will say that things are working as expected, but it won't actually work).

The `ng serve` will go through as usual, but it will fail silently in the form of it not appearing on the site. The easiest way to be aware of the limitations is to change the config in your `angularCompilerOptions`.

```
"angularCompilerOptions": {
  ...
  "strictMetadataEmit" : true
}
```

By changing the `strictMetadataEmit` to `true`, it will emit an error to the console directly, when using the `--aot` flag.

> It should be noted that in this book we reccomend the use of Nrwl Nx. Nrwl Nx when creating a library, will by default add the `"strictMetadataEmit": true`

The best approach towards AOT with regards to limitations is to be aware that there are those. Sometimes the compiler will throw code that it cannot read into an AST.

101 The Angular Service Worker - Implementing in App

For those of you unaware, a service worker is a a script that runs in the web browser that manages caching for an application. So let's say you are offline and you are making a query in your app that you have already made before, then the service workers will make it so that request can go through even without a network request. In short, having a servie worker, can decrease dependency on a network, and will greatly increase user experience.

101.1 Design Goals

1. Caching an application is like installing a native application. The application is cached as one unit, and all files update together.

2. A running application continues to run with the same version of all files. It does not suddenly start receiving cached files from a newer version, which are likely incompatible.

3. When users refresh the application, they see the latest fully cached version. New tabs load the latest cached code.

4. Updates happen in the background, relatively quickly after changes are published. The previous version of the application is served until an update is installed and ready.

5. The service worker conserves bandwidth when possible. Resources are only downloaded if they've changed.

101.2 Manifest File

To support the above design goals, Angular loads a manifest file. The manifest describes the resources to cache and includes hashes of every file's contents [1].

[1] taken from https://angular.io/guide/service-worker-intro

101.3 Using Angular CLI to Enable Service Workers

In the chapter where we used ng new for the first time, we set it up with a flag for service workers. [For practical purposes, if you did not use the flag for creating service workers, use the link here] and follow through on the steps in the link. I believe in you! You can do this!

For academic purposes, here is what the service worker flag does:

1. Adds the @angular/service-worker package
2. Sets the Angular Cli serviceWorker option to true, so that it generates a manifest for every build
3. Imports the ServiceWorkerModule, and registers the ngsw-worker.js file, which is the name of pre-build service worker script
4. Creates a ngsw-config.json file, which configures defaults for service worker

101.4 Simulating a Network Issue

1. Go to Chrome dev tools [2]
2. Go to the Network tab
3. Check the Offline box

If you service worker is properly being used, then the page will load normally, as opposed to the page displaying, "There is no internet connection".

For further reading, by all means read through the documentation on Service Workers, on the Angular.io. I agree, reading their documentation can be a bit bland at times, but it is really thorough and more than get's the job done. Good job Angular documentation person, or persons!

[2] write something here if person does not know how to do so

101.5 Some other architectural decisions

101.6 Available and Activated Updates

There is an SwUpdate service available within app, after importing the ServiceWorkerModule. It can be used to notify users, for instance, to update their page(s), when the code they are running is out of date:

```
    updates.activated.subscribe(event => {
      console.log('old version was', event.previous);
      console.log('new version is', event.current);
    });
```

Listing 101.1: app.routing.module.ts file

101.7 Checking for Updates

Within the same SwUpdate service, we can also check for updates, and set up a subscriber of sorts, for instance:

```
  import { interval } from 'rxjs/observable/interval';

  @Injectable()
  export class CheckForUpdateService {

    constructor(updates: SwUpdate) {
      interval(6 * 60 * 60).subscribe(() => updates.checkForUpdate());
    }
  }
```

Listing 101.2: app.routing.module.ts file

102 Understanding Rendering

Rendering within Angular is a really important thing to understand. Primarily, because of it's nature as a framework, the default way of doing things, can be a bit slower than needed. As an engineer, as you get to understand the framework better, you will find that there are performance boosts baked into the framework, to make your app more performant. Rendering within Angular is definitely one of those topics. Therefore, let's dive into the different ways of rendering on the web, so we can bring that over to Angular.

102.1 Terminology

Rendering and Performance Terminology	
Rendering	Performance
SSR	TTFB
CSR	FP
Rehydration	FCP
Prerendering	TTI

102.1.1 Rendering

1. SSR: Server-Side Rendering - Rendering a client-side, or universal app to HTML on the server

2. CSR: Client-Side Rendering - Rendering an app in a browser, generally using the DOM.

3. Rehydration: "Booting up" Javascript views on the client such that they re-use the server-rendered HTML's DOM tree and data

4. Pre-rendering: Rinning a client-side application at build time, to capture it's initial state as static HTML.

102.1.2 Performance

1. TTFB: Time to First Byte - seen as the time between clicking a link and the first bit of content coming in.

2. FP: First Paint - The first time any pixel becomes visible to the user.

3. FCP: First Contentful Paint - The time when request content (body, header etc.), becomes visible.

4. TTI: Time to Ineractive - The time at which a page becomes interactive (events wired up, etc.)

102.2 Setting Proper Frame of Mind - When Rendering Happens

For me personally, when trying to understand how rendering worked, I realized that one of the things holding me back was that I wasn't quite fully aware when I happened. When I was able to put into perspective that it happens in two scenarios:

1. Navigation

2. Events

I was then able to continue on with my understanding of rendering. So I just wanted to part this tidbit of information to you the reader.

102.3 Server Rendering

Server rendering will generate all of the html for page based on navigation. It's important to keep in mind the two scenarios above, and that server side rendering only happens for the navigation side of things. Events will still have the client side render changes.

This will produce a fast first paint, and a fast first contentful paint. In addition, it allows for a fast time to interactive. The one draw back with regards to this approach, is the Time to First Byte. Being that we have to generate all of the content in the server first, it takes longer than usual to generate the page. Within the Angular context, it has something called Universal, which is a bit of a mix and much between server and client. Something that we will discuss momentarily.

102.4 Static Rendering

Static rendering, is where we generate the full set of html content ahead of time. This checks all boxes with regards to performance i.e.

1. Fast TTFB, fast FP, fast FCP, fast TTI.
2. Fast FP
3. Fast FCP
4. Fast TTI

You might be familiar with tools like Jekyl, Gatsby, or Hugo that accomplish this. The main drawback with this approach, is that if we have dynamic content that we don't know ahead of time what the html page is going to look like, static sites are not able to accomplish this task.

> A really good rule of thumb to find out if your site can be used as a static site, go ahead and disable Javascript. If you site is still able to operate, then you site can be turned into a static site.

102.5 Server vs. Static Rendering

So let's say that you are coming across a situation wherein you cannot use static rendering. You would like to use static rendering. The only issue in this regard, is that you have a page that is dynamic. Even in this instance, server side rendering is not an end all be all solution. There will need to be some sort of caching solution offered in this regard, for server side rendering to be truly effective.

102.6 Client Side Rendering

Client side rendering is what usually happens by default in an Angular applciation.

1. logic
2. Data fetching
3. Templating

4. Routing

Will all happen on the client side. There are a couple of downsides when it comes to rendering client side. One of the more notable downsides is as follows:

1. Performance concerns for mobile

Irrelevant within an Angular setting

102.7 Universal Rendering - Server + Client Side

Universal rendering is the happy medium between server side and client side rendering. Navigation requests(full page loads + reloads) are accomplished by the server side. The actual HTML is delivered by the server. The Javascript and data needed is embedded into the document. The one downside to SSR, is that it might look like it is fully loaded, but a user will not be able to interact with until the Javascript is fully embedded. On devices such as mobile, this can take a couple of seconds. The one outlier with regards to SSR, is that if you can find a use case wherein the content will be highly cacheable, then SSR might be a good candidate.

102.8 Up and Coming Technologies

There are some technologies that are on the horizon that are useful to keep an eye on. There are two in particular, streaming server rendering + progressive rehydration.

102.8.1 Streaming Server Rendering

This allows for you to send HTML in chunks, so that the browser can progressively render as it's recieved.

102.8.2 Progressive Rehydration

With this approach, individual pieces of an application are booted up over time as opposed to an entire application. This will help reduce the amount of Javascript required for an application as it is aware of what it is that the user specifically needs. This also helps prevent one of the major concerns of SSR rehydration, wherein a server rendered DOM tree rebuilds itself.

102.8.3 Notable Mention

The idea behind this approach is for the browser to understand what parts of an actual application are being used. If some part of the application is not being used, then the browser will turn it into an inert HTML/decorative Javascript.

103 Angular Universal

103.1 What is Server Side Rendering?

> "Angular Universal is Angular's way of rendering something server side."

By default, Angular will render HTML on the page using Javascript within the browser. Server side rendering, however, will produce all of the HTML off the browser and within the server. Normally, people come across using Angular Universal for two reasons:

1. Mobile app - Javscript engines on mobile phones, while much stronger, are still lacking. In addition, while having the server generate the HTML instead of the client, can potentially boost battery performance of application.

2. SEO reasons - Angular Universal will make your site static.

Another added benefit of Angular Universal, is that it has a very fast FCP(first contentful paint). When you use Angular Universal for the first time, this will immediately be apparent. It should be noted however, the TTI(Time to interactive), wherein user will be able to interact with the page, will still be loading in the background.

103.2 Angular Universal Actually Requires A Sever

One thing that people using Angular Universal will be surprised, when using it for the first time, is that it actually requires a server! Looking back it will make sense, because obviously it is server side rendering. However, the name "Angular Universal" doesn't exactly lend to that assumption. Anywho, I digress, any web server will work with Angular Universal.

103.3 A Couple of Points to Keep in Mind

103.3.1 How Angular Universal Works

1. Angular Universal uses the `platform-server` package under the hood. This provides low level features that don't rely on a browser(`XMLHttpRequest`, etc.)

2. Server passes the client request (for application pages) to the `ngExpressEngine` which calls the Universal `renderModuleFactory`. This function inputs:

    a) Template HTML page.

    b) Angular `module` containing components.

    c) `route` determining which components to display.

3. The `renderModuleFactory()` function, renders view within the `<app>` tag of template.

4. The server will then return rendered page to the client.

103.3.2 Dynamic of Browser APIs with Angular Universal

A universal app works on the server side. Because of that, it needs to create abstractions over classic APIs such as `window`, or `location`. There some bugs that might show as a result, if your app does anything fancy.

Another really important point, is that a Universal app cannot interact with a mouse, or keyboard event. Without some hack/workaround, it is important to make your entire app routable.

103.3.3 Using Absolute URLs - Serving on Browser vs. Serving on Server

There is an interesting dynamic between serving on the browser, and serving on the server. When serving on the browser, the paths for a url are relative. When serving on a server, the paths must be absolute. So why is it that this is indeed the dynamic?

Well, think of it this way. When generation is happening within the browser, it is fully aware of everything going on from the client side of things. The actual url is irrelevant. There could potentially be a way of approaching this problem. However, after looking into the github issues, for the angular github, one will find that this is actually a surprisingly complicated issue. Given the way the package works out.

From a server side, it does not have access to the client side, so using relative paths is not an option. You will need to create an interceptor to pass a full url to your server, based on your client data. (It is possible to do this on the server's side of things, but arguably less overhead to do this on the client side.)

103.3.4 Universal Template Engine

Within your `server.ts` file, you will write something similar to the following:

```
app.engine('html', ngExpressEngine({
  bootstrap: AppServerModuleNgFactory,
  providers: [
    provideModuleMap(LAZY_MODULE_MAP)
  ]
}));
```

Listing 103.1: server.ts

There are two notable items in the code above:

1. AppServerModule - This is the bridge between the client and server.

2. extraProviders - This one is optional, and is something that

The `ngExpressEngine` function returns a callback promise to the client.

103.3.5 Security + Static Files

In order to ensure that all static files are delivered properly to clients. Put all client side files in the `dist` folder. Only honor requests for files coming from the `dist` folder.

104 Angular Elements Load TIme

Angular elements will include by default the Angular framework. As a result the bundle size will be somewhere around 60kb. This is one of the common complaints of Angular-elements. First and foremost, it is important to realize why it is that Angualr Elements is this size.

1. This size is large as a result of Angular being bundled within the general custom web elements.

2. With the way that Angular Ivy deals with bundling, this will be greatly diminished as time goes on.

3. When defering script, even though it ends up getting loaded, with a proper load stategy, it does not affect performance. In fact, if built right, with a proper Angular Elements strategy, it can be just as effective as lazy loaded elements.

   ```
   <script defer src="elements.js">
   ```

104.1 Lazy Loading Angular Elements

105 Change Detection

Change detection is one of the found principles behind Angular. It is a large part of what makes the frmaework what it is. Of course, understanding the minute details of Angular's change detection, can very such help us understand all other parts of the framework. In addition, help us increase boosts when it comes to performance.

105.1 Understanding Change Detection as a Concept

Without a framework in place, we would change a particular piece of text using Javascript. For instance, let's say we have a promise which returns data, with that new data, we would do something like the following:

```
yellowBoxText = newData;
```

In a framework, doing something like this isn't neccesary and one has the option to go lean into the framework to this for you. As you are mostly likely a UI Engineer familiar with Angular, I won't bore you with the details. However, here is a quick primer:

```
<!-- In our html file -->
{{ newData.name}}
// In our Typescript file
this.facade.user$.subscribe((userData) => {
    this.newData = userData;
  });
});
```

Very simply, using the above, any time that our newData changes, all of the relevant data inside of out html file will be updated.

105.2 Understanding Change Detection Performance

First and foremost, every component in Angular has it's own change detection. This means that if data changes for the component, only that particular component will be updated. In addition, as is probably intuitive at this point, change detection in Angular is top down. If a parent component is changed, then all child components will be re-rendered.

In addition, in angular, there is the ability to emit data. Even if a component is a parent component, it will still be updated accordingly.

105.2.1 Sibling Components under Same Parent Component

Sibling components under the same parent component will not update unless they both feed from the same component. Keeping change detection at this level from an architectural perspective is really all you need to know. It is important to know about zone.js the internal workings of how change detection works in Angular.

106 Integrating ngrx/store with Apollo Client

In many architectures, it is most likely going to make sense that it is microservice based. That is, your data will be served over a slew of different apis. In which case the intent is that in a regular application, you will be using GraphQL. It is reccomended that you use Apollo Client as well. We will cover real quick, how to integrate GraphQL with Apollo Client.

106.1 What is GraphQL

GraphQL is a backend data query language. It will allow you to use a query to make a request, as opposed to having to supply, url, endpoint, and type of request.

106.2 The Benefit of Apollo

Apollo is a GraphQL client, used to help ease use of GraphQL http requests within app. In particular:

1. Allows the application developer to easily execute GraphQL queries, and configure transport-specific features like headers.

2. Ensure that all GraphQL results currently being displayed in an app, are consistent with one another.

3. Provide flexible ways to update the cache with results from the server when using mutations, pagination, subscriptions, and more.

106.3 Dilemma When Using Apollo Client with @ngrx/store

Apollo will create itâĂŹs own inMemoryCache, without dependency on Redux. However, this creates two separate stores within the app when using @ngrx/store. One for ngrx/store, and another for the Apollo client. It would be much easier, obviously, if we had a singular cache/store, for the app.

106.4 Enter apollo-angular-cache-ngrx

apollo-angular is a series of packages for the integration of the Apollo client with Angular. apollo-angular-cache-ngrx is a package officially a part of one of the apollo-angular packages. It solves this exact problem, and allows one to use @ngrx/store as oneâĂŹs Apollo Cache. (The following can be seen in the github README.md for the apollo-angular-cache-ngrx repo, but going to put here for convenience reasons).

106.5 Installation

We will be wanting to install the entire apollo suite at this time.

```
npm install apollo-angular apollo-angular-link-http apollo-link apollo-
    client graphql-tag graphql --save
```

As well as apollo-angular-cache-ngrx

```
npm install apollo-angular-cache-ngrx âĂŤ-save
```

106.6 Usage

```
import {StoreModule} from '@ngrx/store';
import {
  NgrxCacheModule,
  NgrxCache,
  apolloReducer,
} from 'apollo-angular-cache-ngrx';

@NgModule({
  imports: [
    StoreModule.forRoot({
      apollo: apolloReducer,
    }),
    NgrxCacheModule,
  ],
```

```
})
class AppModule {
  constructor(ngrxCache: NgrxCache) {
    const cache = ngrxCache.create({});
  }
}
```

If one were to now make an Apollo GraphQL query in your Typescript component, your current ngrx/store will be populated with the appropriate Apollo data. You will be able to subscribe to it as usual. For instance:

```
  constructor(store: Store<any>, private apollo: Apollo) {
    this.store = store;
      apollo
        .query({
          query: gql`
            {
              users {
                status
                id
                name
              }
            }
          `
        })
        .subscribe((initialData: any) => {
          console.log(this.initialData);
        });
  }
```

At this time however, we are not using GraphQL within the above fashion. Nonetheless, your average app, will be using this sort of architecture, and it is very good to be aware of it, as well as how to use it.

106.7 Bonus - Using Fetch Policy with Apollo

In apollo, there is the option to specify the fetch policy with regards to cache. For instance, in your

```
  const fetchPolicy = 'network-only';
  const userActivities$ = this.apollo
    .query<any>({ query, variables, fetchPolicy })
```

Doing the above, will not cache your query. Therefore, if you have data, which will need to re-loaded once a request is made after initial load, using a fetchPolicy with network only, is the reccomended approach.

107 Sockets

When using caching withing an angular application, the first and only dillema that will come to play is how update that cache. For most applications, especially the nature of Angular, is that it will the data will need to be updated frequently.

107.1 Understanding Internals of Sockets

Sockets are that way of updating a cache. It sets up a hook to the server, so that whenever the server is ready to respond it does. You might be familiar with this, if you've had a network request that's taken a while to return it's request. Sockets are like that. When ever the cache has been updated, another socket goes out, to get it's data when needed. With every request that is sent an id(requestId) is sent over, so that server knows exactly where to send the data back, and it allows for a bi-directional flow between the server and client.

107.2 Handshake Protocol

When creating a websocket request,

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

the http header îs simply given an upgrade HEADER request. That will then go to the server. If the server can accept the websocket request, the request will be made. /footnotehttps://medium.com/@SunCerberus/setup-apollo-client-2-0-with-websocket-example-a879ca81aa83

108 Apollo Caching with Sockets

I would like to re-iterate, because I like you the reader, that the value of being aware of architecture is twofold. One is that one has the option of preventing future catastrophies. In addition, one has the ability to quickly put together the correct solution, and have in available in a time intensive setting. Caching with Apollo is exactly that scenario. One might include caching, but realize that sockets is something that we would like included with the app. As a result, the development team might decide to throw away caching, so that data can be updated. I would like to say that it is very much possible. I would also like to lay out, what in a couple of paragraphs that best strategy to do so within Apollo.

108.1 Apollo Caching with Sockets

> One side note, is that it will generally be a requirement that will come from front end first, that they will need to use something like sockets. However, backend will also need to setup websockets, in order to get everything to work as expected.

It should be noted that GraphQL offers three types of data queries:

- Query
- Mutation
- Subscriptions

Subscriptions in particular is very similar to websockets.

The package at this point which makes the most sense is subscriptions-transport-ws

109 Responsive Design

109.1 Choosing a Framwork

With regards to responsive design, there are a number of frameworks, one can choose with regards to creating a web app. The following are quite popular:

1. Foundation
2. Bootstrap
3. Semantic UI

However, the above for a grid system tend to be overkill in my opinion. Specifically, the direction many UI web app tend to head, is that it will only be used on Desktop. For mobile and tablet, there will be a separate Android and IOS app created. In addition, due to the nature of angulars component architecture, the use of ready made components, containers for apps, are the only thing which will actually have specific media queries.

It is strongly suggested that your own super lightweight grid is created, or used. I think it is important to keep in mind that most grid systems can be limited to 500 lines of code, or less. I personally prefer to use Skeleton [1]. However, I am in the process of creating my own grid system using css-grid. (need to get back to this one).

Alternatively, creating our own grid system is also advantageous. That is the direction that will make sense in any enterprise app. Generally, in any business setting, from the business side they will decide on having a unique look and feel. Setting up something for the app that works.

109.2 Breakpoints

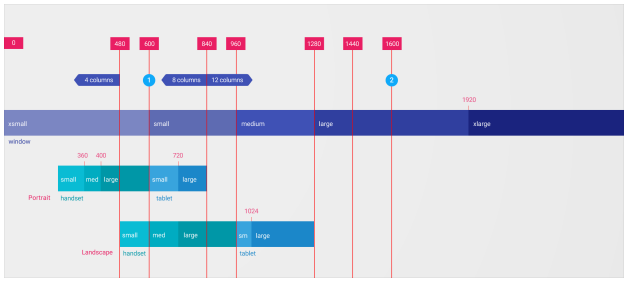
The reccomended architecture for a design language system is material design. First, it really is the largest design system. Second, think about it this way. Which design language would you rather choose, the one created by the company that uses the same front end

[1] http://getskeleton.com/#grid

framework that you are using, or the one created by the company, not using the same front end framework you are using? I 100realized that this logic can be flawed. However, the company that we are talking about is Google, so... yeah.

Now you might be thinking, that a design language system, is front end framework agnostic. However, I would like to make the following point. A design language system, is then built into a component library. If a design language is not built with a particular front end framework in mind, the odds of it being built into a robust component library, are slim. Angular Material Components is so robust, is almost turns not using Material Design as your design language as a sin.

109.2.1 Official Material Design Breakpoints



109.2.2 Choosing Four Specific Breakpoints

Even within an enterprise setting, using all the breakpoints specified within the Material Design specification can be quite a bit. We recommend using the following breakpoints:

1. 400px[Mobile Large - Portrait]
2. 720px[Tablet Large - Portrait + Mobile Large - Landscape]
3. 1024px[Tablet Large - Landscape]
4. 1424px[Desktop Large](16p below 1440 to accommodate for gutter space)

109.2.3 Of Four Breakpoints, Which Designs are Required?

2 Designs are required, Mobile Large(400px), and Tablet Large(1024px).

109.2.4 Does UX/UI Need to Follow these Breakpoints?

UX/UI does not need to use actual engineering breakpoints. For instance, it might be easier to design with a laptop in consideration, instead of a large tablet.

109.2.5 Build Media Query Function

Per conventions, it is reccomended that all items relating to the DLS be specified with regards to functions. That way, we have a way of making sure that nothing deviates from the design language system.

We will be creating an ill-ui folder and ill.scss file) in our lib folder, to hold all all of our app's .scss files. In addition, we will be creating an _ill-breakpoints.scss file, to be imported in our ill-ui.scss file. We will be creating the following in our app:

```
1   @function ill-breakpoint($breakpoint) {
2   $breakpoints: ('small': 400, 'medium': 720, 'large': 1024,
3   'extra-large': 1424);
4
5   @if(index($breakpoints, $breakpoint)) {
6     @return #{map-get($breakpoints, breakpoint)}px;
7   }
8   @else {
9     @error "Must contain one of the following strings: #{$breakpoints}."
          ;
10  }
11  }
```

Let's imagine you are now building a media query for a specific component that you are using, you can do something like the following.

110 PWA Toolset - Physical Devices

There is currently a formula with which devices to use. Latest regular sized Iphone, and Iphone Plus. Latest Google Pixel non plus [1]. Latest Ipad + Ipad Mini. That is it. These are mostly used as a way to see web in real time, as you are developing your application.

Therefore, the reccomended Physical mobile devices are as follows:

1. Iphone 8
2. Iphone 8 Plus
3. Google Pixel 2
4. Ipad (2018)
5. Ipad Mini 4

110.1 Browser Dependencies

The following is expected browser dependencies on Desktop:

1. latest two Chrome releases
2. latest two Firefox releases
3. latest Safari
4. latest Internet Explorer
5. latest Internet Explorer
6. latest Microsoft Edge
7. Windows 10

[1] That's right, skip the Samsung.

8. Windows 8

9. macOS Sierra

110.2 Testing Local Server on Physical Device

Now that we have our physical devices that we would like to work on, let's set up a way that we can test on these mobile devices. Ideally the following three criteria should be solved:

1. Url that remains the same for dev - to be used on mobile device

2. When edit is made, it should update all mobile devides simultaneously

3. Have all mobile devices in a central location, so that we can visibly see all changes that are being made

4. synchronized interactions [2]

110.3 Ghost Labs

First, our winner for responsive testing is Ghost Labs. Ghost labs fulfills all of the above criteria mentioned above. Going into short why we chose it over all other contendors:

1. Very easy to setup, and therefore removes overhead for initial setup

2. There isn't anything required to install on different devices. It is simply a url that is used, and shared across device.

3. Screenshots on remote mobile devices

4. Ghostlab has a built in inspector for debugging

5. One click workspace, in order to start up all devices once again.

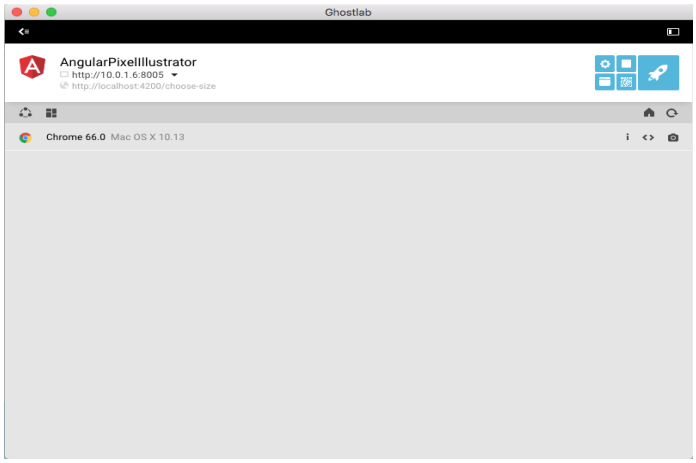
6. Presentation mode, allowing users to present web app.

[2] Clicking on a button in one place change it in all other places.

110.3.1 Setting up Ghost Labs

First and foremost, buy the Ghost Lab Device Lab Selector. I can assure you, it is an architectural decision. The whole point behind developing on a physical mobile/tablet devices, is to improve developer workflow. So that any change that happens, can be viewed immediatly. The device Lab selector serves that purpose.

Setting up ghost labs is as simple as running it in the mac application, and being able to open on numerous devices. The following is a screenshot of what you might see in your Ghostlab application:



Simply drag and drop the url of your browser into application. Click on the play button. It will open up the above screenshot. You then have the option of opening up the Ghost Labs generated url in any mobile device, by simply scanning the qr code.

110.3.2 Notable Mentions of Ghost Labs

1. Synchronized browsing between different devices.
2. Compile and refresh happens as if working with native CLI.
3. Remote inspection of mobile devices.
4. Remote screenshots.

110.3.3 Final Words on Ghost Labs

With regards to setting up responsive development workflow, Ghost Labs is next to none. The $49 dollar licensing fee is chump change. The amount of time saved on infrastructure, is worth the investment. The device lab in addition to buying actual current devices might go up to something in the ballpark of $1,000 to $2,500.

However, this is arguably a fantastic decision. The engineers on your team have 8 hours a day to develop. Many times taken up by meetings, architectural decisons, corporate events etc. Allowing them to seamlessly develop, without having to re-configure their browser is a time saver. Perhaps somewhere around 1 hour a week. In addition, it helps promote developer happiness, by making their life easier. At the end of the day, perhaps for this reason alone, it is worth it.

111 PWA Toolset - Sauce Labs

111.1 The Value of a Continuous Testing Cloud?

As we have mentioned in the chapter for physical devices, we have quite a bit of different platforms to work on. Ideally, we want an environment that we can set up with E2E tests, as well as integration tests, and then run on all environments. This is on top of the physical devices we already have. To clarify[1]

The idea of physical devices is as follows: to have a real time update of all edits being done in your local environment, so that you can go ahead and have a continuous local development environment.

The idea of a continuous testing cloud, is to be able to check that all devices and browsers are being properly tested on. This will work strictly with one's e2e tests.

111.2 Bring it to the Table - Why We Chose Sauce Labs

At this point in time, there are really two main competitors, when it comes down to continuous cloud computing:

1. Sauce Labs

2. BrowserStack

Before we go into the above, endtest which is a fantastic up and comer, is unfortunately a victom of it's own business model. While allowing users to create a very simple version of tests, it also locks in users to it's platform. There is no way of exporting these tests, and it makes it a very uncomfortable place for many enterprises. This is precisely the type of application we are trying to focus on, in this book, so we shall move on. [2]

[1] i'm saying this a completely friendly way

[2] Even if I were working on a small app, I would still not use endtest, for fear of scale

111.3 Sauce Labs

vs.

The competition for BrowserStack vs. Sauce Labs is pretty stiff. To be honest, at this point, they should go for different marketplaces, or one of them should go under. They have many of the following similar features:

1. Platform
2. Languages Support
3. Framework
4. Support
5. Browser Release Support
6. Browser Support

However, the following are the reasons why we believe Sauce Labs is the better choice. Sauce Labs has quite the roster of large clients, and it's fantstic documentation shows. In addition, because Sauce Labs is widely used, we are able to find better community support. Channels such as StackOverflow for a specific quirk.

111.4 How to use Sauce Labs

112 Mobile First - Building a Progressive Web App

112.1 Why Build a Progressive Web App?

When building a an enterprise application, think about building a Progressive Web App. It will allow your web experience to be built to feel as if it is a native app experience. Not only will it make it progressive, but it will make your users feel as if they are a part of an experience that is all encompassing. It will give them the overencompassing feeling that they are getting the best experience possible [1]. Swipe right on Progressive Web Apps [2].

112.2 The Technical Benefits of a PWA

1. **Progressive** - Work for every user, regardless of browser choice because theyâĂŹre built with progressive enhancement as a core tenet.

2. **Responsive** - Fit any form factor, desktop, mobile, tablet, or whatever is next.

3. **Connectivity** independent - Enhanced with service workers to work offline or on low quality networks.

4. **App-like** - Use the app-shell model to provide app-style navigations and interactions.

5. **Fresh** - Always up-to-date thanks to the service worker update process.

6. **Safe** - Served via TLS to prevent snooping and ensure content hasnâĂŹt been tampered with.

7. **Discoverable** - Are identifiable as âĂIJapplicationsâĂİ thanks to W3C manifests and service worker registration scope allowing search engines to find them.

8. **Re-engageable** - Make re-engagement easy through features like push notifications.

[1] We will discuss moving the app over to a native app soon using NativeScript.
[2] That is a millenial joke, but also a darn good PWA pun.

9. **Installable** - Allow users to âĂIJkeepâĂİ apps they find most useful on their home screen without the hassle of an app store.

10. **Linkable** - Easily share via URL and not require complex installation.

Kudos to Addy Somani for this List [3]

We will go into detail in the following chapters, into detail

112.3 Developing a PWA - The Toolset - An Overview

When Developing Mobile First, there are three tools, which will be particularly advantageous:

1. Physical Mobile Devices [4]

2. Sauce Labs [5]

3. Chrome Dev Tools [6]

112.4 Developing a PWA - Software - An Overview

When developing a PWA, it is important for us to keep in mind, that there will be specific pieces of software to develop. Of course, there is an official PWA checklist [7] , but specific technologies, would be as follows:

1. Service Worker

2. Manifest

3. Lighthouse

[3] https://addyosmani.com/blog/getting-started-with-progressive-web-apps/
[4] A moment on which one's it is, that you should work on
[5] Section on why we chose Sauce Labs, over Browser Stack
[6] Section on why we chose Chrome Dev Tools, over Firefox
[7] https://developers.google.com/web/progressive-web-apps/checklist

113 Flex Layout

113.1 What is Flex Layout

Flex layout provies an HTML UI layout for Angular applications; using Flexbox and a Responsive API.

113.2 Understanding the Issue With CSS Based Flexbox

- CSS specifity [1] issues rapidly become problemtic, and continue to be so.
- CSS footprint size becomes excessively large(¿250k for flexbox CSS) whereas with JS, the module is loaded within every specific module
- Changes in layout direction required changes to child flexbox stylings
- No built in support for customized media query breakpoints (need to specify them yourself)

113.3 Why Use Flex Layout?

- Flex Layout is indexpendent of Angular material
- No external Css requirements
- Support for Handeset/Tablet and Orientation breakpoints
- Support for any layour injector value
- Support for raw values, or interpolated values

[1] CSS speicifity, is the means by which browsers decide which CSS property values are the most relevant to an element and, therefore, will be applied

- Support for raw, percentage, or px-suffix values. Change detection for Layout injector values.
- Use provider to supply custom breakpoints
- Notifications for breakpoints changes
- MediaQuery Activation detection

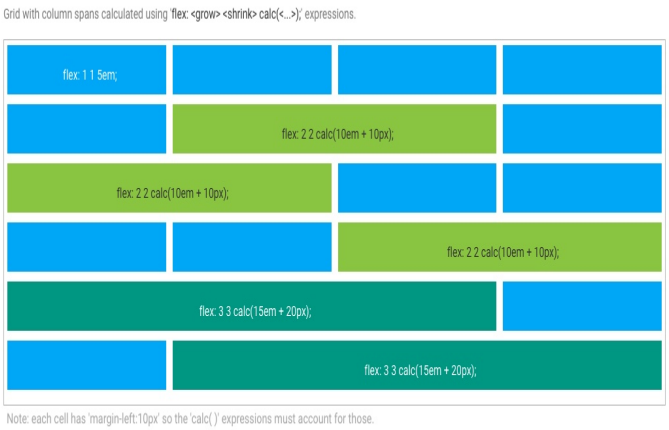
113.4 A Great Example of Flex Layout

One of the greatest selling points with regards to flex layout, is that it makes responsive design very easy to do. In particular, it's ability to apply a gap between elements, and easily change them between different devices. However, I would like to show an example of the piece of code that sold me on flex layout. I was playing around with a feature called fxLayoutGap. It particular, it uses margin-right used when the parent container `flex-direction == "row"` and margin-bottom is used when the parent container `flex-direction == "column"`

So for instance, let's say we have a div will be using flex. We would like there to be two divs inside, with a margin-left of 16px. This is how we would do it if we weren't using fxLayoutGap.

```
.icon-text {
  margin-left: mc-space-multiplier(1);
}
.chapter {
  margin-top: mc-space-multiplier(4);
}
```

The following is a great example, of what can be done with flex-layout.



The following is the code which

produces the above:

```
<div class="containerX" >
  <div class="container">
    <div>flex: 1 1 5em;</div>
    <div></div>
    <div></div>
    <div></div>
  </div>
  <div class="container" >
    <div></div>
    <div [fx-flex]="calc2Cols"> flex: 2 2 calc(10em + 10px); </div>
    <div></div>
  </div>
  <div class="container" >
    <div [fx-flex]="calc2Cols"> flex: 2 2 calc(10em + 10px); </div>
    <div></div>
    <div></div>
  </div>
  <div class="container">
    <div></div>
    <div></div>
    <div [fx-flex]="calc2Cols"> flex: 2 2 calc(10em + 10px); </div>
  </div>
  <div class="container">
    <div [fx-flex]="calc3Cols" class="col3">  flex: 3 3 calc(15em + 20px); </div>
    <div></div>
  </div>
  <div class="container">
    <div></div>
    <div [fx-flex]="calc3Cols" class="col3">  flex: 3 3 calc(15em + 20px); </div>
  </div>
</div>
```

114 Styling a Component

Styling a component, of course, is a very complex topic. With styling, as an architect in an Angular setting, there are 4 things that you will have to keep in mind:

1. Pre-processor of choice(Scss, Less, PostCss, etc.)
2. Design system
    a) Material Design(Google)
    b) Fluent Design(Microsoft)
    c) Flat Design(Apple)
3. Responsive design(even if you have a mobile/tablet app)
4. Naming convention of CSS classes

114.1 Pre-processor of choice

For our preprocessor, we have chosen Sass. [1]

114.2 Naming Convention

For our naming convention, we will go with BEM. It is an extremely easy way of setting a part a specific component from an html and css side of things. A quick primer on BEM. Block is a component. We will be using pascal casing for ours [2] Element is a child of block. It uses an underscore. For instance:

```
<div class = 'ChooseSize__input'></div>
```

[1] Incude link for a discussion of why that is
[2] Link to airbnb style guide

M stands for modifier. A modifier is an element, which modifies an already existing element.

114.3 Design System

In an Angular setting, the component library which seems to make most sense is Material Components [3]. For starters, it is a complete design system. All component's design will be synonymous with each other. In addition, it is in the process of creating a cdk, which makes all of these components customizable. In addition, it is a really nice design, and feels native to the way Angular works. I have used it versus other libraries and I can really say the documentation is just fantastic. I have used it in more complex settings(e.g. the data-table), and adding on new functionality has been just a joy.

114.4 Adding Material Design to Our App

First install Angular Material components and Angular Animations to our app.

```
npm install --save @angular/material @angular/cdk
npm install --save @angular/animations
```

In addition, we will need to add default styling to our app, in order for styling to be applied to our Angular Material component. Inside of our styles.scss file, import the following.

```
1 @import '~@angular/material/prebuilt-themes/deeppurple-amber.css';
```

114.5 Our first component

In our app, we are going to create our first component. It is essentially a form with three fields:

- Columns
- Rows

[3] https://material.angular.io/components/categories

- Pixel Size

In addition, there will be a button which will say, 'Create Grid'. We are also going to wrap our component, with the ¡mat-card¿ component, add a width of 300, margin-top and center.

114.5.1 Notable Mention - @HostBinding

In an angular app, many times, we will want to add a specific class to our parent container. In our situation, we will be using BEM, and creating a ChooseSize class. It will implement flex, and use justify content, in order to center the ¡mat-card¿ component.

```
import { Component, HostBinding, OnInit } from '@angular/core';

@Component({
  selector: 'app-choose-size',
  templateUrl: './choose-size.component.html',
  styleUrls: ['./choose-size.component.scss']
})
export class ChooseSizeComponent implements OnInit {
  @HostBinding('class') class = 'ChooseSize';
  constructor() {}

  ngOnInit() {}
}
```

Listing 114.1: My Javascript Example

By putting @HostBinding as a decorator [4] within our app, it causes the host class to have the ChooseSize class. We are then able to target our host element our scss:

```
:host.ChooseSize {
  display: flex;
  justify-content: center;
}
```

114.6 CSS Naming Convention

In your modern day front end framework, such as Angular, generally, we do not have to worry about clashing namespaces. [5]. Many other issues with css at scale, have been solved as well, have been solved by the general ecosystem, scss included.

[4] If not familiar with decorator , it is a function that is run when particular class is called
[5] Historal footnote, the turning point for me was with this article here

However, reccomended architecture is that one still use something like BEM. I would like to argue for using BEM in an Angular setting:

1. It allows for easy grep in code base, when inspecting element first within chrome.

2. It documents the type of element that it is.

3. It will give structure to html, without need of using pug, or some other html pre-processor.

4. Ease's creation of classes for integration testing[6]

It should be noted, that within our app, the form has been made a particular width, which will work on all screen sizes, without the need of adjusting width. As we move along in our app, we will have the option to look into sitations wherein we can use actual media queries.

[6] Use a modifer for BEM

115 Scully: Static Site Generation for Angular

Traditionally, static sites are just that - static. They didn't change and what you coded is essentially what you'd get. However, with the growth of APIs, the idea of JAMstack - that is, JavaScript, APIs, and Markup - grew in popularity.

By 2017, enterprise-level JAMstack projects started to make an appearance, with the first JAMstack conference commencing in 2018.

The idea behind JAMstack is that you can run entire websites without the need for server side code. This leaves you with the ability to focus on your front end experiences and reduce server response times for pages significantly. Your data needs are covered by APIs, with JavaScript dealing with the necessary connections between APIs and what users get to experience.

Scully sits in the sweet spot of being Angular's first static site generator that is helping fuel the growth of Angular based JAMstack sites.

Under normal Angular circumstances, we'd lazy load each component as they're needed. When it comes to JAMstack, pre-generated static pages are required. This means that if JavaScript is disabled for whatever reason, the site will continue to work.

When using Scully, you're essentially adding an additional step to the build process that compiles and builds the required static pages, in addition to the ability to generate the required code with the help of âĂŸplugins' to get started.

The fun part of Scully is that the code generated can be unit tested. This means that Scully can be implemented and integrated into your projects with end to end testing capabilities.

In part, this is because the code generated fits neatly into the Angular code seamlessly.

115.1 Getting started with Scully - the static part

Scully can sit on top of any existing Angular project and can easily be added via the CLI.

To add Scully to a project, simply use the following commands while in the root folder of your project.

```
ng add @scullyio/init
```

To build and run with Scully, use the following commands:

```
ng build
npm run scully
```

Once you've done this, you'd find static files inside your dist folder. A folder called static will appear alongside your application folder. Viola! You've just converted your existing Angular app into a JAMstack, static (but still data dynamic capable) site.

115.2 Getting started with Scully - the generator part

The perk with Scully is that it allows us to generate a blog using Angular's generation schematics. What this means is that you can use ng generate to create Scully templates.

To generate a simple blog, use the following command:

```
ng g @scullyio/init:blog
```

This command will create a blog module with the required routes. You'll also get a blog folder with some markdown files. Every time Scully builds, the markdown files will be rendered into HTML.

To create your first post, use the following command:

```
ng g @scullyio/init:post --name=âĂİthe-first-postâĂİ
```

If you look at your blog folder, you'd find a file called the-first-post.md with some markdown content inside.

When you want to run the blog, you'll need to build it because Angular can't read markdown and the contents of the post will need to be compiled into HTML.

Every time you build, the markdown will be converted into the necessary code and cut down on server response time because it's serving from generated static files.

To serve your Scully site, run the following command:

```
npm run scully serve
```

It's time we talked about Scully services Creating a Scully service is similar to creating a typical Angular service. What a Scully service enables your app with the ability to tap into the pre-rendered pages and display the content as needed.

To do this, you need to import ScullyRoute and ScullyRoutesService into your component and then initialize it as an Observable. ScullyRouteService is a pre-written service by Scully and comes packaged with the library.

```
import{ ScullyRoute, ScullyRoutesService } from âĂŸ@scullyio/ng-lib'
âĂę
âĂę
export class AppComponent implements OnInit{
   posts$: Observable<ScullyRoute[]>;
   constructor(private srs: ScullyRoutesService){ }
}

To connect and display it in your view:

<ul>
   <li *ngFor=âĂİlet post o posts$ | asyncâĂİ>
      <a [routerLink]=âĂİpost.routeâĂİ>{{post.title}}</a>
   </li>
</ul>
```

115.3 What about Angular Universal?

Angular Universal has been around for a while and is the natural solution to single-page apps SEO related rendering woes.

The similarities between Angular Universal and Scully lies in their pre-rendering aspect. However, Angular Universal and Scully diverge on how pre-rendering is done, thus marking the end of their main comparison point.

When it comes to Angular Universal, there is server-side rendering involved. This means that pages are built on the fly and then sent over to the client-side. On a technicality, Angular Universal is not JAMstack because it involves server-side rendering and is not a truly statically generated site.

Scully however, comes pre-rendered at deployment. This works well if the content deployed is not expected to change over time - making it perfect for blogs and blog posts.

While Angular Universal is pretty simple in approach - generate a pre-render on the server-side to give to client-side - Scully's methodology runs at two levels.

The first level is that the pre-rendered page is given to the user when requested. The second level is the actual âĂŸreal' Angular app that sits in the background on top of the pre-rendered view. So what you're essentially getting is the pre-render for a particular view (if available) and the actual app - rather than the app and then the page.

This cuts down on the initial delay and the need to call your database for data. As a result, Scully can help reduce the read loads on databases in the long term, allowing for more cost-effective ways to scale. This is because reading static files is much lower in cost than reading from a database.

115.4 Final thoughts

It may not seem like a big deal but static site generation is a rendering concept that reduces overall server loads and increases the potential to scale in a cost-efficient manner. Scully is an exciting community development because it automates the pre-rendering process and contributes greatly to the JAMstack ecosystem.

Another perk of pre-rendered single-paged apps that are not widely discussed enough is how it impacts on SEO and the ability to rank effectively on search engines.

The issue that many new developers face with SPAs is that their app may be working but fails to get picked up by major search engines because crawlers and bots are simple in design and work best on content that is not dynamically generated.

This problem is fixed when you use Scully as all pages are pre-rendered and load data only when you navigate away and into a space that requires dynamic content.

Integrating Scully into your existing development workflow to leverage the ability to generate static content is something that can be easily done. It is just a matter of importing Scully as a module, building the project and then running it. This entire process can be scripted inside your package.json scripts section.

As Scully grows, the ecosystem surrounding Scully is also expected to grow in the same way it did with Angular Universal. While there is debate if Scully will replace Angular Universal completely, it's still too early to make any dramatic conclusions.

Scully, after all, is still quite new.

Scully does, however, provide a methodology that's highly needed but unsupported effectively for blog related activities. The Internet is built on content and Scully is a tool that helps Angular developers get a leg up on supporting parts of the app that's traditionally linked with content that's unlikely to change often.

116 When to Use @ngrx/store

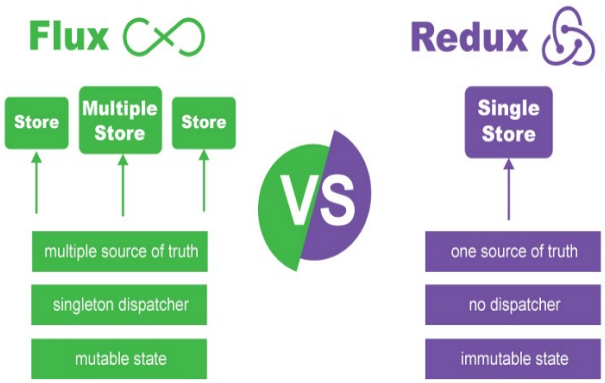
When to use @ngrx/store can be a very opiniated item to consider. State as we know it, is a single data object, that can be accessed anywhere within the app. How this happens, changes from one state management framework to another, but the core is the same. However, the difficulty, is that state, being that it exists outside of the component, can contain quite a bit of bloat.

It is interesting to note, that the original implementation of Flux was created by Facebook back in 2014[a]. It was because they had a bug with regards to the chat counter, that just kept on coming back. It would mention that a user did not read a message, but that wasn't true. Flux was the architecture they came up with in order to solve this bug. It's important to understand the history of where Flux came from, when trying to understand how to move forward using it. It's important to also realize, that the issue, wasn't they weren't able to solve it, it is that the bug kept coming back. Flux solved it, due to it's straight forward way of solving multiple components interacting with each other.
"It is important to remember, that a store is about creating an in-memory client side database, which is a user-specific slice of the database, and use that data to derivce View Models from it on the client."

[a]This video contains this fact

116.1 Redux as evolution on Flux

Redux solved the same problem Flux did. However, it simplified the process. Without going into too much detail and instead showing a photo, Redux simplified the problem it solved, and made it easier to implement:



It also more importantly, standardized the Flux architecture as a library.

116.2 @ngrx/store - Integrated Reactive Programming with the Store

@ngrx/store took redux one step further, and integrated observables with the store. There is a fantastic founding paper on the benefits of Real Time: Programming

In it, it discusses two main benefits of Reactive Programming:

1. Asynchronous

2. Detemrinistic

Just to throw out what a quote from who I consider a founder in Reactive programmin. Andre Staltz when discussing why one should consider Reactive programming really touches on the core of it all with the following, "Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic. . . ". [1]

That being said, it immediately becomes obvious as to why one would want a reactive approach towards state. State is generally changed with events, and being able to abstract the level of state makes complete sense. I cannot stress how many times, I have been able to pull in an observable using @ngrx/store, and modify it using rxjs. It greatly abstracts, and simplifies the process.

[1] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754#why-should-i-consider-adopting-rp

As a side note, ngrx was actually created, to solve performance issues with change detection in Angular2. It from that point onwards has morphed into a good idea, allowing the store to be automatically hooked into state: https://github.com/ngrx/store/issues/16#issuecomment-172027797

116.3 An Example of an @ngrx/store

For instance, let's say that we have a user-settings data-access feature. Our folder file structure might look like the following:

This, of course, would be in an enterprise setting, which honestly this book is geared towards. However, it definitely strikes home a very good point. There is a ton of bloat to creating a store. Even once a developer get's over the hill of creating state, and get's used to it, it's still a lot to manage, and one just has to ask, does it always make sense?

116.4 Understanding the Reality of State

I think that looking at the above folder/file strucutre, it is very important to realize that state has gone so far beyond what it was originally intended to do. You will notice effects, a facade pattern. In addition, quietly sitting inside of our reducer, is @ngrx/entity. The whole suite of @ngrx is so robust that it deals with the entire life cycle, of pulling in data, and integrating it with the client side. In fact, if it wasn't for the bloat that it introduced, it would be able to solve every single situation, wherein we are trying to carry over data into the client side, in a scalable, and maintainable fashion. Ok, so let's deep dive into it.

116.5 Addressing the Problems State Alleviates

First and foremost, I think it would be most effective if we were to directly jump into the problems that state solves:

1. Avoiding Multiple Actors

2. Avoid Extraneous @Inputs

    a) Pass down multiple levels to child, and send change back to parent

    b) Siblings in a tree, might have to pass up and over

3. Stops Event Bussing /marginparLook more into this one

4. Decouples component interaction. Component does not know what changed, only knows what changed it.

5. Allows for Component interaction via the Observable Pattern

6. Client Side Cache if needed.

7. Place to Put Temporary UI State

These would be in my humble opinion, the 7 things that state has to offer. If a component were to be able to interact with another component on that page, by altering it's data, it should have state.

116.5.1 Addressing Additional two Offered by @ngrx/store

Added into your classic @nrwl/nx ecosystem, is:

1. @ngrx/entity

2. Facade pattern for single

3. Effects for asynchronous programming

116.6 The Mesmerism of @ngrx/store

So, as mentioned before, the @ngrx/store ecosytem has grown into a beast. In fact, an Angular developer working on an enterprise, data heavy application, might spend 60% to 70%(rough estimate) of their time working within @ngrx/store. I was a little bit curious as to how this repition might psychologically affect the way that we as software engineers develop. In particular, if we can introduce services that accomplish what @ngrx/store does, albeit less bloat, is it worth it for the team?

I found an interesting book, called, "On Repeat: How Music Plays the Mind". It discusses a very interesting topic, of how we as humans interact with music. In it, it discusses something known as "The Exposure Effect". It's something that we have all experienced with regards to music. For instance, you will hear some music on the radio that you are not particularly fond of. However, you will here it again the grocery store, movie theater, and then the store corner once again! By this time, you are hooked! More importantly, as the book discusses, as you are used to the repition of the song, you are already expecting the next piece of

song, by the time you working on the beginning of it. Repition, allows for us as the book argues, to look at a passage as a whole.

Arguably, the point can be made for software as well. Within the application you are working on, the more you use that pattern the more accustomed you are to it. In addition, and I can personally attest to this, by the time, I am working on the facade, I am already thinking of the effect that is going to tie into our service, and how I am going to tie it into the component. Jumping out of this repitition can counter-productive, even if there is less code involved by creating a service.

116.7 Attachments Service - Story Time

There are some very unique cases, where perhaps state shouldn't be used on an objective level. I for one always felt that state has a bit of bloat(, albeit my opinion is beggining to change). However, after discussing with team members, I have begun to see the way they think. This particular situation was about a feature for attachments that the app needed. In particular, multiple components on the same page, would have attachments that would be uploaded to a server. Then, when the api using the attachment would make a request, it would pull attachment from the server, using it's id. If a service or @ngrx/store was not used, there would be alot of repitition across the app. This led to the dillema, should a service, or @ngrx/store be used.

116.8 Business Requirements

One, is that multiple components on the same page were to use this attachments service. Second, is that we want to show the user when a certain attachment is loading, and when it is no longer loading.

116.9 Argument for using a Service

In truth, the attachments were meant to be self contained within a singular component. This would have made the service as very simple. However, based on the business requirements, we would have had to create a double nested correlation id. This means that a double nested correlation id pattern had to be created. One, to make sure that different components do not affect each other. Two, the id produced on the front end, is different than the id produced on the backend. We need to create an id that can be used by both. This is done by creating a second Uuid, that is nested within the first one. I.e. a dictionary inside of a dictionary.

So, really in this situation, there is arguably nothing that the store has to offer. However, what has happened, using "The Exposure Effect" is that we are now comfortable with the API for @ngrx/entity. Comfortable isn't accurate actually. We are able to predict, @ngrx/store as a whole. Any developer who worked and will work within the app, can pick up on the code base easier than a service, beign that it is predictable. In addition, we have a single place where we expect all of our data to be. Creating a service like this, might make sense for the person spending the time thinking through the problem. However, for any other person, it will be an incredibly uncomfortable experience reading through your code. @ngrx/store therefore at this point takes on it's new life as a way to make sure code is consistent, even if it isn't the right choice for your app.

116.10 What Comes Out From Our Back and Forth

Truly, any enterprise situation, wherein a data request is made from the back end, it should be handled using @ngrx/store and not services. This is simply because, in any enterprise setting, the majority of situations is more properly handled using @ngrx/store.

1. It allows for a cookie cutter api, that is used time and time again. The code bloat it creates, is alleviated by use of Nrwl Nx.

2. Your application's performance will not affected as a result.

3. From personal experience, business requirment change quite a bit. The odds of your service now being needed to re-written to accomodate it being used in multiple components, is also quite high.

116.11 Final Note

If you are building an enterprise app from the beginning and are building a team, put this as part of the conventions right away. It will make your life easier. If you have a team, and you haven't fully agreed on this one, send them this article and start a conversation. Razroo Cares.

117 Primer - Actions

At this is a definitive guide, and covers all aspects of the framework, one would be remiss without going through the entire lifecycle of the store, and mentioning it here.

An action is a function which contains two very important pieces of information:

1. What the name(type) of the action is. [1]

2. The payload of the action. Which is a single object, containing all the data passed into the store.

117.1 An example of an Action in an Angular setting

An Action is pretty generic across different frameworks. [2]. However, within an Angular setting, it means that we are using @ngrx/store for controlling state across our app and Typescript. Within an Angular setting, an action will look like the following:

```
export class LoadCodeBox extends Action {
  readonly type = CodeBoxTypes.LoadCode;

  constructor(public payload: {id: string}) { }
}

export class CodeBoxLoaded extends Action {
  readonly type = CodeBoxTypes.LoadCode;

  constructor(public payload: CodeBox) { }
}
```

In the above code, is a perfect representation of your classic action. It contains the type of action. So that in you reducer, or effect(which we will go into momentarily if not familiar), we can trigger reduce(combine) data passed in, with data already present.

> In an Angular/Typescript setting, you will commonly find a general export type, for the entire app: export type Actions = LoadCodeBox — CodeBoxLoaded;

[1] Why this is important we will get to soon

[2] For certain frameworks, this is less relevant, such as React + Vue.

This is usually used within the reducer, and is called a union interface type. It allows us to combine the type for numerous actions within a single type. We can then pass this as a type to a reducer, and make sure that we do not introduce any action beyond that for the particular feature state. It's a great way to make the develop think twice, if they introduce an action outside of the feature state into the app.

117.2 An example of using an Action in an Angular setting

As a matter of philosophy, this book will only introduce how to use something as it is in the real world. Generally, an action will only be used in a variety of situations such as an effect, facade, or guard. [3]. As a simple example, let's use it in a facade [4].

```
1   loadCodeBox(id) {
2     new LoadCodeBox({id})
3   }
```

Here we are calling the class by using new. It will call the action whenever the facade is called.

This would be as simple as a primer can get for actions. It might seem like there are more questions that came up as a result (Go through potential questions one might have).

[3] If not familiar, no need to worry, we will go through this as time goes on.
[4] Something we will discuss, a dedicate an entire chapter

118 Primer - Reducers

Reducers are pure functions that take in data [1], and return a new data object. Immutability is ideal when using reducers, so that data is not affected by mistake, when being passed down the pipeline. Reducers tend to be universal across all frameworks, and usually consist of:

1. One exported function

2. A switch:case statement based on action.type

118.1 Example of Reducer

```
export function reducer(state = initialState, action: book.Actions
| collection.Actions): State {
  switch (action.type) {
    case: CodeBoxActionTypes.CodeBoxLoaded {
      return {
        ids: [ ...codeBoxIds ],
        entities: Object.assign({}, codeBoxEntities),
        selectedCodeBoxId: state.selectedCodeBoxId
      }
    }
    default: {
      return state;
    }
  }
}
```

This reducer will be called whenever any action is called. However, only if the action matches the case statement, will the reducer actually run.

[1] Should take in data only and for the sake of this primer, we are going to keep it this way.

119 History of State Management

I wanted to write this, because having a history of state management put's into perspective why we need state management. It also put's into perspective how much so things change, and how important having a foundation in software is. In particular, to be aware of alternatives, and to help with learning new concepts. Jumping right in, Jquery was created a very long time ago, already back in 2006. Show, hide, remove, add, as well as element selectors, were already present in Version 1. Javascript had this capability as well if need be. However, no one really thought of it as state management.

119.1 State Management with jQuery

A classic component, I remember that was always created with Jquery, would be image sliders. In the more elequent apps, they would use singleton classes, perhaps prototypes if they knew what they were really doing. Variables would be cached by initializing once. Functions would be kept small, and everything including css, would have very unique nomenclature(BEMCSS for instance). Folder/file structure was important, but there wasn't really anything like state management. Ironically, many smaller websites at this time were more performant in many ways. Why? Because, many intentionally kept them small, in order to do more. 2015-2016 was a great year of performance, due to a growth spurt in browser capabilities. The change log for 2015, is the last time you will see google chrome mentioning performance in it's logs.

119.1.1 Jquery and Javascript example

The following is a great example of how Jquery and Javascript "state management" would work(updated to use es6). A file which would contain values, would be used to create/add/delete/update across the app:

```
// _elem.js file
storeValues: [],
storeColors: [],
sassColorVariables: [],
lessColorVariables: []

// _grid.js file
updateGridColor: () => {
```

```
  for(let x = 0; x < elem.s.columnCount; x++) {
    for(let y = 0; y < elem.s.rowCount; y++) {
      ctx.strokeStyle = `${elem.el.backgroundRed.value + 44}. ${elem.el.
          backgroundGreen.value + 44}. ${elem.el.backgroundBlue.value +
          44}`;
      ctx.strokeRect(x * elem.s.pixSize, y * elem.s.pixSize, elem.s.
          pixSize, elem.s.pixSize);
      ctx.fillStyle = elem.el.backgroundHexColor.value;
      ctx.fillRect(x * elem.s.pixSize + 1, y * elem.s.pixSize + 1, elem.
          s.pixSize - 2, elem.s.pixSize - 2);
    }
  }

  for(let x = 0; x < elem.s.storeValues.length; x++){
    ctx.fillStyle = elem.s.storeValues[x][2];
    ctx.fillRect(parseFloat(elem.s.storeValues[x][0]) + 1, parseFloat(
        elem.s.storeValues[x][1]) + 1, elem.s.pixSize - 2, elem.s.pixSize
         - 2);
  }
}
```

Above code taken from the https://github.com/CharlieGreenman/codeIllustrator repo.

119.2 State Management with Backbone

Backbone applications to me were so funny, and still are. It literally looked like a well architected Jquery app minus routing. Which now that I think about it, isn't funny. Backbone was a big step up. Routing was a very nice touch that offered something like that out of the box. Ultimately, there really was no concept of state management with backbone either. However, I remember apps being performant, and unmanageable in many cases due to the bad architecture. A step up, of course from badly engineered Jquery applications. So, no state management at this point yet, still! However, using model, there was somewhat a way to do this, that was baked into best practices:

```
  // note_model.js
"use strict";
APP.NoteModel = Backbone.Model.extend({
  // you can set any defaults you would like here
  defaults: {
    title: "",
    description: "",
    author: "",
    // just setting random number for id would set as primary key from
        server
    id: _.random(0, 10000)
  },
  //...
  // note_edit.js
  save: function (event) {
      event.stopPropagation();
```

```
      event.preventDefault();

      // update our model with values from the form
      this.model.set({
        title: this.$el.find('input[name=title]').val(),
        author: this.$el.find('input[name=author]').val(),
        description: this.$el.find('textarea[name=description]').val()
      });
  //...
```

This model would global, or per each component, and could be updated using the above syntax.

119.3 State Management with AngularJS

AngularJS was fantastic because it offered two way binding out of the box. Alot of web applications need that. It also came hand in hand with Jasmine unit testing, and event handling. Completely irrelevant to state management. However, because it introduced services, it really was the first framework to start boxing applications into, "this is what front end architecture should look like", paving the way for state management.

Services, while mainly used for data, were also used different parts of the applciation to interact with each other. Being that Angular applications were Single Page Applications by default, this worked. State was synonymous with services. If you wanted different components to know about the data of service, you would have a setter and getter for that service. The issue with this approach, is that there would be 4, or 5 services that would interact with each other, and it would cause serious issues. In addition, in many applications, old code/bad practices would use $scope in the code base, causing some serious perfomance issues. The following is what a sample servic using a factory would look like:

```
var myApp = angular.module('myApp',[]);
myApp.factory('myService', function() {
    var test = 5;
    var obj = {
        test : 5
    }

    return{
      setTestVal: function(val){
        test = val;
        obj.test = val;
      },
      getTestVal: function(){
        return test;
      },
      data : obj
    }


```

```
  });

  function MyCtrl($scope, myService) {
      $scope.test = myService.getTestVal();
      $scope.data = myService.data;
  }

  function SetCtrl($scope, myService){
      $scope.newTestVal = '';
      $scope.setTestVal = function(val){
        myService.setTestVal(val)
      }
  }
```

119.4 State Management with React

React came around, and interested me atleast for two reasons. It offered flexibility being a library and not a framework. Second, it was fast in comparison to AngularJS. Flux came out, and was my first introduction to a state management system. Redux came out 6 months after Flux already, so admittedly, I only had a chance to work with Flux for a month, before we already started moving to Redux. Flux was a bit difficult, and during that month time, I remember my code reviews being rampant, with don't do this, do that etc. Shortly after Flux, Redux came around, and for the first time it felt like a mature state management system came around.

```
// pixel-color-picker.js component
handlePixelColorChange(e){
    const {dispatch} = this.props;
    this.setState({pixelHex: e.target.value}, function(){
        dispatch(PixelColor(this.state.pixelHex));
        dispatch(PixelColorRGB(hexToRgb(this.state.pixelHex).r,
         hexToRgb(this.state.pixelHex).g, hexToRgb(this.state.pixelHex).
            b));
    });
// control-panel.js actions
export function PixelColor(color){
  return{
    type: types.PIXEL\_COLOR,
    pixelHex: color
  }
}

// colorPicker.js reducer
case types.PIXEL\_COLOR:
  return Object.assign({}, state, {
    pixelHex: action.pixelHex || state.pixelHex
  });
```

1

119.5 Reactive State Management with React and Angular

Around this time @ngrx/store came out, reactive programming became more popular. Within the context of state, this meant redux-observable for React, and @ngrx/store for Angular. For Angular, this meant that state is now cookie cutter. For React and Angular, it meant that users have the ability to tie state into the rest of their application.

```
Observable.merge
  // Create observable map for  when background hex changes, and use
      that
  // value to update store for backgroundColor
  this.changePixelColor$.map((value: any) => (
    PixelColor(value)
  )),
  this.changePixelColorRGB$.map((value: any) => (
    PixelColorRGB(value.pixelRed, value.pixelGreen,
      value.pixelBlue)
  ))
)
.subscribe((action)=>{
  store.dispatch(action)
})
```

// code take from here

119.6 Hooks and Context with React + Vue

Where we are at currently, is that new waves are being made with regards to state management. State is being baked into frameworks in ways that make it more lightweight, and easier to deal with. Vue and React now have a feature called hooks, and context. These allow an app to have state out of the box. Redux + Redux Observable still have they're place. There are times where state is neccesary to allow components on different pages interact with each other. Other times, it can be a way of managing the data, to make sure the app is maintanable. If you see your app heading in the direction of the latter. Redux + Redux Observable is still reccomended.

```
// theme-context.js

// Make sure the shape of the default value passed to
// createContext matches the shape that the consumers expect!
export const ThemeContext = React.createContext({
  theme: themes.dark,
```

[1] code take from here

```
  toggleTheme: () => {},
});

// theme-toggler-button.js

import {ThemeContext} from './theme-context';

function ThemeTogglerButton() {
  // The Theme Toggler Button receives not only the theme
  // but also a toggleTheme function from the context
  return (
    <ThemeContext.Consumer>
      {({theme, toggleTheme}) => (
        <button
          onClick={toggleTheme}
          style={{backgroundColor: theme.background}}>
          Toggle Theme
        </button>
      )}
    </ThemeContext.Consumer>
  );
}

export default ThemeTogglerButton;
```

code take from here

119.7 Final Words on State Management

One point I would like to end off on. I remember 5 years when all of the framworks were coming out, there was a developer who told me that if you know what you are doing, really none of the frameworks are neccesary. That being said, no one in their right mind, is going to create their own framework when they have it readily availalble. That is unless the company has the agenda to make one. However, what is important, is to understand the internals, so that you can that much more valuable when it comes to performance, and mainatanability of your project. I think that is obvious, but just wanted to bring it up.

120 Introduction to @ngrx/store

As discussed in the chapter on the History of State Management, there has been quite a series of progression with regards to state management. Redux, is the mature concept of state management including actions, reducers, and a single application store. [1]. Angular's @ngrx/store, which is not front and center, is where redux meets Rxjs. Rxjs is the Javascript library for using observables [2].

120.1 What Makes @ngrx/store Different than Redux?

When considering this question, it is more important to consider what is reactive programming. Reactive programming in particular introduces two concepts [3]:

1. Asynchronous [4]

2. Deterministic [5]

120.1.1 Asynchronous

With regards to asynchronous programming, observables are not neccesarily reactive in the strictest sense. This is because the client is working separate from the server. Events with observables are most definitely reactive , and help, but many applications are data heavy, and the immediate value of observables are cut short. In addition, actual UI events from a browser perspective are put in a call stack, and are asynchronous by nature. What does help, is that baked into the framework is effects. Which does allow the UI to be asynchronous, but it's not like it's anything crazy. This is something which could easily be done with promises. However, @ngrx/store does tie it nicely into the store as a whole, allowing client state management, without going into detail.

[1] This is a footnote for more information on state management

[2] More on Observables can be read here

[3] http://www-sop.inria.fr/members/Gerard.Berry/Papers/Berry-IFIP-89.pdf

[4] Meaning one event fires after the previous one is complete, unlike synchronous which means they all fire at the same time, and might complete out of order.

[5] Always produces the same results. As a side effect of this, code becomes very much so cookie cutter, which is great in an enterprise setting, as it allows for greater re-use.

120.1.2 Deterministic

With regards to determinstic, this generally means in computer science, that with one particular input, you will alwyas have the same output. However, when the term is used loosely, it generally means that the code is cookie cutter. That is, that it can be re-used time and time again.

120.2 Wrapping Up

This is really what @ngrx/store tries to produce over other frameworks. It offers the ability to re-use patterns time and time again. In addition, by hooking it into the Rxjs lifecycle by using observables it allows patterns and for code to be cookie cutter. This is really the beauty of @ngrx/store, is that it offers a end to end solution to for state management. In particular, in the form of effects, and observables.

121 Ngrx CLI

Of one of the better reccomendations I can give with regards to using Angular within you app, is using the Nrwl Nx cli. If you are using Nrwl Nx already, which you should, and have a mono repo setup within your organization, then it should be readily available. If you haven't already, please refer to the chapter on Nrwl Nx setting it up.

121.1 Why Use a CLI?

One of the benefits of using a CLI, is that it subtly enforces the entire team to use a particular convention. With the Nx CLI for ngrx, this is doubly true, as it strongly enforces conventions to be used as to how ngrx works. In addition, within the ngrx arena, it strongly enforces how certain files should be built.

121.2 Why use a CLI for ngrx

Without a doubt the most fustrating thing about Angular before the CLI came around, is the amount of boilerplate that would be required in order to work with Angular [1]. The Nrwl Nx for the most part solves this. In addition, there is much more that can be done on top of this in the future. Think of automatically generating code based on instance, but I digress.

121.3 The two stages of Nx ngrx cli

There are going to be two stages with regards to using the Nx ngrx cli. One of them is to create root state. This is going to be empty, and it is just there to hold the feature reducers of the rest of the app. The 2nd stage, which will be repeated time and time again, is creating a feature reducer. (Creating a root reducer is something that you would only have to worry about if you are the one responsible for creating the project for the first time).

[1] As an example, having to create a scss + html + action + reducer + effect and respective spec file for each next component created

I think it would beneficial to running through the steps of how you would create state with the @ngrx/cli, if you are not familiar with it already.

121.4 Creating a Root State

At this point, the Definitive Guide assumes that you have already setup your space as an Nx Workspace. If you haven't please do. It will not negatively impact your app in any which way, and will only positively improve the way your codebase works and your day to day happiness.

The whole idea of state in ngrx/store, is that there is a single object, and all subsequent pieces of state are sub piece of state. Therefore all pieces of state will be contained in a single object. In order to have this done in your code, you will need to set up state for your root, so that subsequent peices of state can be added as child object(feature), to the root state. This will only have to be done once.

In order to create a root reducer:

```
1   ng generate ngrx app --module=apps/<app-name>/src/app/app.module.ts --
        onlyEmptyRoot
```

Note, we have passed in the flag for onlyEmptyRoot, so that none of the files for actions, reducers, and effects are created. We simply want a module generated that we can use to import other modules for state.

This will produce the following files /marginparRe-visit when we get back to the app

121.5 Creating Feature State

We will be going into detail, into when one should create feature state management, versus using a service. Or, when it might be considered overkill, or suprisingly the right thing to do. Please refer to the chapter on data access architecture, to learn more about when would be the proper time to generate state for your component.

1. libs/¡libname¿/src/+state/products.actions.ts
2. libs/¡libname¿/src/+state/products.effects.ts
3. libs/¡libname¿/src/+state/products.effects.spec.ts

4. libs/¡libname¿/src/+state/products.reducer.ts

5. libs/¡libname¿/src/+state/products.reducer.spec.ts

There is also the option to add a facade, which is highly reccomended. In addition, creating a seperate selector file is extremely valuable.

122 State Management - @ngrx/store

Ngrx/store is a layer on top of Redux. It is a state management tool that was originally created, in order to solve two way binding performance issues within Angular. [1]. It then extended as a way to bring redux natively to Angular, with the use of Observables.

Let's dive into integrating @ngrx/store into our app.

> This particular component has been written in the fashion of TDD. However, another chapter will be dedicated to TDD/BDD in order to specify this point specifically.

122.1 Using nx ngrx to Generate State

122.1.1 Create root state using nx ngrx

First we are going to generate an empty root, for our StoreModule, as well as our EffectsModule. Our StoreModule is responsible as a singular store object, which will be holding all of store data. Our EffectsModule is a singular effects object, which will be holding all of our effects. [2]

```
1 ng generate ngrx app --module=apps/angular-pixel-illustrator/src/app/app
    .module.ts --onlyEmptyRoot
```

122.1.2 Create component state using nx ngrx

Next, we are going to create state for our choose-size component. This is done with ease using nx ngrx [3]

Run the following command:

```
1 ng generate ngrx choose-size --module=apps/angular-pixel-illustrator/src
    /app/components/choose-size/choose-size.module.ts
```

[1] Need to further bring source for this one

[2] We will discuss effects in more detail later

[3] Trust me, I've been in situations where I was not using a CLI. It is not good news

This will generate the following files:

```
1 create apps/angular-pixel-illustrator/src/app/components/choose-size/+
    state/choose-size.actions.ts (684 bytes)
2 create apps/angular-pixel-illustrator/src/app/components/choose-size/+
    state/choose-size.reducer.ts (869 bytes)
3 create apps/angular-pixel-illustrator/src/app/components/choose-size/+
    state/choose-size.effects.ts (859 bytes)
4 create apps/angular-pixel-illustrator/src/app/components/choose-size/+
    state/choose-size.effects.spec.ts (1070 bytes)
5 create apps/angular-pixel-illustrator/src/app/components/choose-size/+
    state/choose-size.reducer.spec.ts (364 bytes)
```

And update the choose-size module,

```
1 update apps/angular-pixel-illustrator/src/app/components/choose-size/
    choose-size.module.ts
```

122.1.3 High level overview of nx ngrx

So, you might be wondering, what do those files that nx ngrx generated actually do? It will generate three files:

1. Action
2. Reducer
3. Effect

In addition, nx will add Typescript enums for the action types. It will also add a respective spec file(unit testing) for the action + reducer file.

Unit testing Actions?

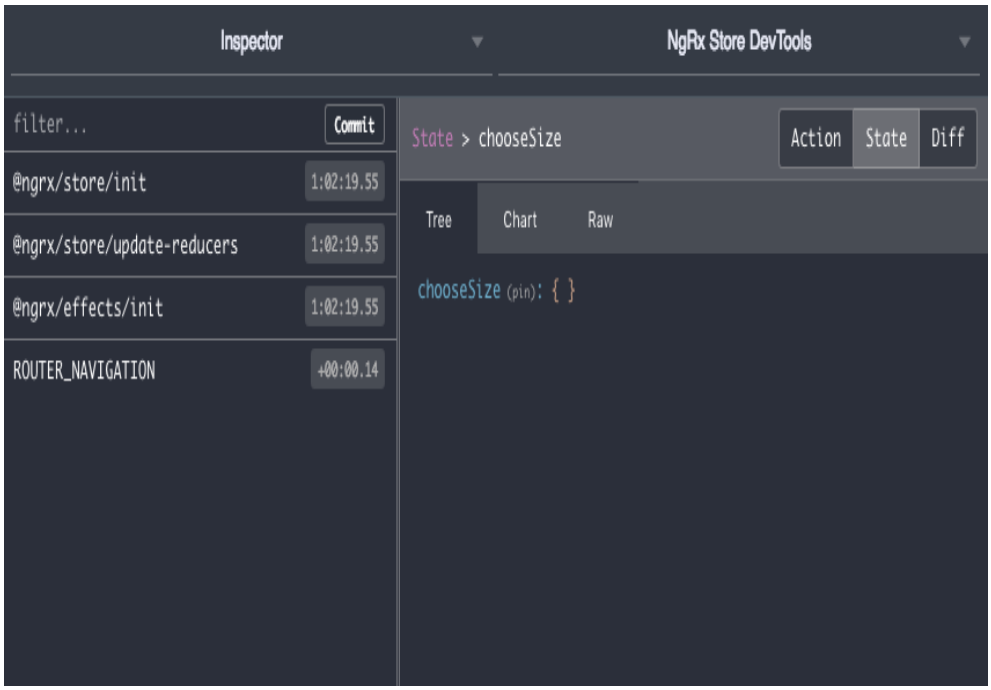
Unit testing an action, would simply say, when an action is dispatched, expect it to be of a certain type. However, enums, as well as type checking, fulfills that obligation. Therefore, if one is using Typescript along with enums, there should be no reason for writing unit tests.

122.1.4 Installing Redux Dev Tools

A state environment is incomplete without proper devtools. In particular, being able to see an action fired, as well as the complete state of any given time, is invaluable.

Google, "redux Devtools"[4]. It is offered by remotedev.io.

With the chooseSize ngrx nx command, we just made, you should see something like this:



[4]In a book format, in my humble opinion, more valuable than a link

123 ngrx/router-store

Ngrx/store as discussed previously is an Angular flavored Redux styled state management library, that leverages Rxjs. It deals with the management of data streams and propagation of change. It is time bound, meaning that data tracking and histories are stored for referencing and tracing.

Under the traditional model of routing, data is decentralized and sits on each routing state. This means that if the route changes, history of that data is lost. There is no past and future states, only the present and navigating away discards any memory of such occurrences. This can become quite a challenge to keep track of everything in medium to larger sized applications, especially when navigation is expected to occur in high frequency.

Ngrx solves this issue by creating a central storage space for routing. This official library is called `ngrx/router-store` and is pre-defined, to automatically hook into your ngrx/store. It keeps all your current application's data in one space - turning parts of your browser's memory into a storage bucket for all your data where all mutations occur only through explicit dispatch actions known as reducers and becomes the application's single source of truth. It allows for your application's events to exist in a unified manner rather than decentralized across different parts, children, siblings, partials, factories and routes.

`ngrx/router-store`, specifically, is the portion of NgRx module that allows for listeners to be used for routing actions, meaning that data is allowed to be stored, shared, consumed and mutated based on the routing status from a single source. `@ngrx/router-store`, in a way, is like an in-memory database for your application's route related data.

123.1 Why do we need it?

When data is decentralized and exists on the fly, it becomes prone to errors due to a lack of history tracking and mutations can occur from different directions. Duplications can accidentally happen as we try to replicate certain data in different states and parts of the application.

When relying on Angular's routing system, we rely on data persistence through params from navigation/router state. If a child or sibling component requires that data, it becomes coupled with the parent and data needs to be presented again in order to be consumed.

While factory patterns may solve this issue, it can quickly get messy if external entry is granted without explicit knowledge.

In larger teams, factory patterns may not be enough to control the flow and history of data and human error may introduce inconsistencies in the code.

ngRx solves this, along with the reduction of time and code overheads needed to create factory patterns and singular storage spaces. The library comes ready to be plugged into any Angular application with its own set of Redux inspired approach to centralized state storage. Each cycle in a router-store captures a snapshot of the route's state and its associated data. When data is decoupled from routing, it allows your application to become more agile and less dependent on data states through route params.

123.2 How to install router store

Once you have your Angular app, you can use npm to install router-store by using the following commands:

```
npm install @ngrx/router-store âĂŞsave
```

If you're using yarn:

```
yarn add @ngrx/router-store
```

If you're project is created with Angular CLI version 6+, you can use the following command:

```
ng add @ngrx/router-store
```

To check your Angular CLI version, use the following command:

```
ng --version
```

To use inside your application, you'll need to import the `StoreRouterConectingModule` and `routerReducer` from `@ngrx/router-store` like so:

```
1 import { StoreRouterConnectingModule, routerReducer } from '@ngrx/router
    -store';
```

123.3 Router actions and why they might be useful

An action is anything that you can do to your application. A router action is an event that can occur against a specific route on your Angular app. When it comes to routing, are 5 specific actions that can occur and they are the request, the action of navigation, the aftermath (also known as navigated), cancellation and navigation error.

Being able to access and track these actions allows you to control the flow of route state storage management, access to data and the life cycle process. When used in conjunction with route guards - a feature that allows you to protect your views from rendering when there isn't enough information or the right access permissions - router actions can help with the resolutions of states and its consumption.

A request action always kickstart the process which then runs the navigation action that determines if the dispatch should occur or not. If guards are valid, a successful navigation will occur and result in a router navigated action. If something went wrong due to exceptions or lack of user permissions, the router action will return a `ROUTER\_CANCEL` action and nullify any attempts to access the route.

A `ROUTER\_ERROR` action may occur during the navigation life cycle and returns the stored state before navigation occurred. This is particularly useful as it allows the application to back track its action and restore its former data âĂŞ a sort of back button without the need for extra configuration or call to the router state bucket.

123.4 How to use a custom serializer

A custom serializer prevents the mutation of snapshot data during the dispatch process. As data during the navigation cycle is prone to mutability, a custom serializer returns only what you need to be added to the payload and store. So in essence, it tracks the difference and change of a particular state without modifying the entire stored state snapshot.

A custom serializer can be implemented through the abstract class RouterStateSerializer. It is, in a way, a middleman class that processes the difference between what the current state is, what is to be changed and updates only what is necessary.

To create a custom serializer, you will need to import Params and RouterStateSnapshot from @angular/router, along with RouterStateSerializer from @ngrx/router-store

```
import { Params, RouterStateSnapshot } from '@angular/router';
import { RouterStateSerializer } from '@ngrx/router-store';
```

To create a custom serializer, export a class that implements RouterStateSerializer with an interface to ensure object uniformity.

Using the serialize() method to convert the state object into a unified format that conforms to your application's requirements. This often comes in the form of mapping router state values to a predefined interface that may look something like this:

```
//to be used by the serialization process
export interface RouterStateUrl {
  url: string;
  params: Params;
  queryParams: Params;
}
```

The CustomSerializer class that implements the imported RouterStateSerializer using the RouterStateUrl interface created.

```
export class CustomSerializer implements RouterStateSerializer<
    RouterStateUrl> {
   //serialization code here
}
```

Set up the serialization method to return a uniformed set of parameters based on the template set in the RouterStateUrl interface.

```
serialize(routerState: RouterStateSnapshot): RouterStateUrl {
    let route = routerState.root;

    while (route.firstChild) {
      route = route.firstChild;
    }

    const {
      url,
      root: { queryParams },
    } = routerState;
    const { params } = route;

    // returning the object based on the RouterStateUrl interface
    return { url, params, queryParams };
  }
```

To use the custom serializer, implement it inside your @NgModule and call your exported CustomSerializer class inside the StoreRouterConnectingModule.

```
@NgModule({
  imports: [
    StoreModule.forRoot(
      { router: routerReducer },
    ),
    RouterModule.forRoot([
      // routes
    ]),
```

```
    StoreRouterConnectingModule.forRoot({
      serializer: CustomSerializer,
    }),
  ],
})
```

123.5 Benefits of using router-store with ngrx/store-freeze

`ngrx/store-freeze` is a dev tool that can be used during the development phase of an Angular application to prevent state mutation when using `router-store`. It sits on top of `router-store` as meta data that acts as an insurance against changes in state data during the process of transfer to state storage.

As `router-store` provides snapshots of the `RouterState` during the navigation life cycle, it is vital that snapshots passed do not change during the process of dispatch as this will result the store cycle's truthiness breaking due to inaccurate snapshot data.

While serialization already prevents this, `store-freeze` acts as an additional safeguard with exceptions thrown when mutations do occur at runtime. It automatically *'deep freezes'* the entire store state object and dispatch actions, resulting in a read only effect before it gets passed to the serializer. This allows errors to be caught before it gets dispatched, serialized and passed into storage.

124 Store Selectors

Selectors are pure functions /footnotepure function in case you are not aware of already, are functions which always return the same result, given a certain parameter that take slices of state as arguments and return some state data that we can pass to our components.

Selectors can become nitty gritty, especially as we start to get into the finer detail of pure functions. The following covers about 90% of all use cases.

124.1 Object That We Will be Working With

```
{
  settings: {
    column: '20',
    row: '20',
    pixel: '20',
    draw: true,
    false: false,
  },
  colorPicker: {
    backgroundHex: '#191919',
    backgroundRed: '25',
    backgroundGreen: '25',
    backgroundBlue: '25',
    pixelHex: '#000000',
    red: '25',
    green: '25',
    blue: '25'
  },
  codeBox: {
    css: [
    {'123': {color: blue, x: 20, y: 17}},
    {'246': {color: orange, x: 20, y: 18}},
    {'246': {color: orange, x: 20, y: 19}},
    ],
    sass: false,
```

```
    less: false,
    js: false
  }
}
```

124.2 Basics of Select using ngrx/store

The simplest method is to grab state by the using store select method:

```
export const settings$ = this.store.select('settings');
```

This will grab the store data for settings and return an observable. So for instance, let's say inside of our component, doing:

```
  this.gridFacade.settings$.subscribe((data) => {
   console.log(data)
  });
```

will produce JSON for:

```
{
  column: '20',
  row: '20',
  pixel: '20',
  draw: true,
  false: false,
}
```

124.3 Feature State in NGRX

Previously we discussed being able to select a certain slice of state being using store.select. However, there are many scenarios wherein it is not as simple as selecting the featureState and passing it to the component. For instance:

1. When using ngrx/entity, creating a dictionary of values, and an array of ids. In order to grab ids, entities, ids and entities, a specific entity, it's important to be able to have the a base feature selector to work off of.

2. When having numerous nested peices of data, under one specific feature state slice. We can keep our code DRY [1], by want be able to build off of nested data on a number of levels.

[1] Don't Repeat Yourself

It's important to note, that in any enterprise setting, where the data in the store is being populated using backend(GraphQL preferably), we strongly recommend the use of ngrx/entity. It will simplify any given situation if you do anything beyond pulling in data as is. Therefore the following createFeatureSelector will always be used, and in the enterprise applications that I work on, they are consistently used for every data-access feature I create.

124.4 Feature State in NGRX - An Example using Ngrx/entity

Ngrx/entity if not familiar already, will use what is called a normalized database. It is a way of having the way of accessing something independent of the actual data. Therefore, it easily allows for an item to be created, removed, updated, or deleted. Ngrx/entity is ngrx/store's solution to creating a noramlized database. Using feature state with ngrx/entity is imperative in order to streamline all the various ways of selecting ngrx/entity data. For instance, let's say we had entities for our codebox, and we wanted to select data in a number of ways:

1. Select all entities
2. Select all ids
3. Select the current selectedId by using an Entity

In order to streamline this process, we would do the following:

```
const { selectAll, selectEntities, selectIds } = codeBoxAdapter.
    getSelectors();
const getCodeBoxState = createFeatureSelector<CodeBoxState>(
  'codeBoxStateModel'
);

const getCodeBoxEntities = createSelector(
  getCodeBoxState,
  selectEntities
);

const getCodeBoxIds = createSelector(
  getCodeBoxState,
  selectIds
);

const selectedBuyerId = createSelector(
  getCodeBoxState,
  state => state.selectedBuyerId
);
```

```
20
21 const getSelectCodeboxId = createSelector(
22   getCodeBoxEntities,
23   selectedBuyerId,
24   (codeBoxEntities, id) => codeBoxEntities[selectedBuyerId]
25 );
```

124.5 How this would look like in a real world application

Just to douse curiousity of someone who might be wondering how this would work in the real world. Normally one would wire this into a facade file. The facade line of code would look something like the following:
codeBoxEntities$ = this.store.select(getCodeBoxEntities);
This code would then be brought over to the component in the following fashion:
this.codeBoxData = this.codeBoxFacade.codeBoxEntities$

124.6 Introducing createSelector

You will have noticed that in the above code we have also introduced the idea of a createSelector. What a createSelector does, is take the value from a createFeatureSelector base, and build on it. What we have done in our app, for instance, is create a createFeatureSelector for codeBox, a createSelector for codeBoxEntities, and then proceeded to pull from the data nested data within those entities, by using the getSelectCodeboxId selector. In this fashion we are able to streamline the process of getting the data we need, pass it to our facade. Pass it to the component.ts file we are using, and then use it directly within the component.html.

124.7 Final Notes

There is more to be discussed on this topic, however, this covers about 80% of situations wherein selectors are used. By following this formula it allows for selecting data to be cookie cutter. It is highly recoemmended that ngrx/entity be used in addition, to make this cookie cutter process easier.

124.7.1 UI Architecture Notes

With regards to making data re-usable. As an app get's larger, the same sort of data formulation repeats it's self. For instance, in cases that involve pagination, sorting, and

filtering, these are data sets that are going to repeat themselves time, and time again. Please refer to the chapter on re-using reducer logic, for a practical solution to this problem.

125 Aggregation Pattern

Many times within any web setting, numerous apis will be feeding into a singular request. An example would be in an e-commerce setting wherein data for a particular item, might come from numerous locations. The data for the pants might come from one api, the analytics api might update, and then a third api along the lines of data persistence will be called at that time as well.

In a backend setting, usually we try and keep a singular api request for the business logic of a particular use case. However, there are many times wherein this will be beyond the power of the developer. In can be in antiquated apis, or in unique use cases.

125.1 The Unique Challenge with Ngrx/effects

In an ngrx/effects use case, when a user is hitting a single service, and returning a single action, it is relatively straight forward. This will look something like the following:

```
@Effect()
getProductInformation$ = this.dataPersistence.fetch(
  ProductTypes.getProductInformation, {
  run: (action: GetProductInformation, state: ProductModelState) => {
    const { userId, productId } = action.payload;

    return this.service
      .getProductInformation(userId, productId)
      .pipe(
        map((product: Product) => new ProductLoaded(product))
      );
  },

  onError: (action: GetProductInformation, error) => {
    console.error('Error', error);
  },
});
```

In this example we fire off a service, and return the data from that one service. We might even have the option to turn the map into a switchMap, that can be used for numerous actions, originating from a singular service. However, it immediatly becomes a problem once we start to have a singular effect call numerous services, wherein one action is expected to have all of it's data.

125.2 Using the Aggregator Pattern

Before we introduce the aggregator pattern, it is important to make one note. In any situation with numerous actions, we are going to want to include a correlation id within our app. It is important to do this, so that as we have numerous actions coming in at a time, the request is being done for the proper item at that time.

125.2.1 Defining Types for Actions

```
export type AggregatableAction = Action & { correlationParams?:
    CorrelationParams };

export type FailActionForAggregation = Action & { error?: Error,
    correlationParams?: CorrelationParams };
```

First we define the types for our different actions. Both will be a standard action with the addition of correlationParams, which we will use to make sure that we are indded calling the correct action.

125.2.2 Passing in Params for Functions

```
export function aggregate<T extends AggregatableAction,
    TAction1 extends AggregatableAction,
    TAction2 extends AggregatableAction,
    TFailAction extends FailActionForAggregation>
(
    action1$: Observable<TAction1>,
    action2$: Observable<TAction2>,
    failAction$: Observable<TFailAction>
): OperatorFunction<T, [TAction1, TAction2]> {
```

Now we go ahead and specify the params that we need to use for the aggregator pattern. We are going to assume that there will never be a situation wherein we will ever use more than two actions.

125.2.3 Creating a Filter Action

```
    const filterAction = (sourceAction: AggregatableAction, t:
        AggregatableAction) =>
        t.correlationParams && sourceAction.correlationParams &&
        t.correlationParams.correlationId === sourceAction.
            correlationParams.correlationId &&
        t.correlationParams.parentActionType === sourceAction.type;
```

First:

1. We confirm that the function passed in correlation params, and that our source action has correlation params.

2. We check to make sure that the correlationId of the correlation params matches that of the correlationId of the correlationParams of the soruceAction

3. The correlationParams action type, is equal to that of the sourceAction's type.

125.2.4 Creating Aggregated Actions

```
1
2       const getAggregatedActions = (sourceAction: AggregatableAction):
            Observable<[TAction1, TAction2]> => {
3           let a1$ = action1$
4               .pipe(
5                   filter(a => {
6                       return filterAction(sourceAction, a);
7                   }),
8                   first()
9               );
10          let a2$ = action2$
11              .pipe(
12                  filter(a => {
13                      return filterAction(sourceAction, a);
14                  }),
15                  first()
16              );
17
18          let f$ = failAction$
19              .pipe(
20                  filter(a => {
21                      return filterAction(sourceAction, a);
22                  }),
23                  first(),
24                  switchMap(b => {
25                      return Observable.throw(b.error);
26                  })
27              );
28
29          return race(forkJoin([a1$, a2$]), f$);
30      };
```

Here we use a forkJoin, to make sure we have the latest from all values. In addition, we create a race condition, so that if the error emits, it will return the error instead.

```
1       return (source: Observable<AggregatableAction>) => source.pipe(
2           switchMap(sourceAction => getAggregatedActions(sourceAction))
3       );
4   }
```

and then we simply return the result of the aggregated code.

126 Re-using Reducer Logic

In any state management setting, many times an individual will come across business logic that is repeated many times throughout one's feature state. For instance, let's say that we hava fileUpload service that is going to be used in our app. The purpose of this service, is to upload a file, and return a FileMetadataId to be used in another api.

The logic for this state manangement workflow will be the same for actions, facade, reducer, and effects. The question is how can we make our state re-usable so that it we can use it in multiple reducers without having to re-write logic?

126.1 Strategy

The strategy for how we would create a re-usable state is as follows. We would have a facade/action that would pass in an actionTypeName along with the regular payload. The effect would be passed this unique actionTypeNamem which would then call the appropriate facade/action for when data is uploaded, using re-usable service. We would then pass in uploaded data from the action.payload into the reducer. However, to make sure that reducer logic can be re-used, we pass a special param for the reducer function called actionTypeName. We then compare the actionNameIdentifier passed in from the action.paylod to the actionTypeName passed in from the reducer. If they are equal to each other, then we allow for the reducer logic to happen. Let's deep dive on this approach.

126.2 Creating a re-usable Facade and Action

Our UploadFile action will pass the following:

- actionNameIdentifier - For allowing us to re-use reducer [1]
- entityId - Primarily for allowing us to use the ngrx/entity api [2]

[1] Will Discuss this in more detail
[2] Will discuss this in more detail

- file - Using the name html input api, File is what will be passed in through from the input.

```
export class UploadFile implements Action {
  readonly type = UploadActionTypes.UploadAttachment;
  constructor(public payload: { actionNameIdentifier: string, entityId:
      string;
     file: File; name: string}) {}
}
```

3

In our Facade, would have the following:

```
uploadFile(actionType: string, file: File, entityId: string): void {
    this.store.dispatch(
      new UploadFile({ actionNameIdentifier, entityId, file })
    );
  }
```

Most notable, we pass in a parameter called actionNameIdentifier. This will be compared against created by the reducer to make sure we are targeting the featureState that we would like to work with.

126.3 Higher Order Reducers

A higher order reducer borrows it's name from a higher order function.

> A higher order function is: Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

A higher order reducer is very similar. It allows us to create a reducer, that can be passed into another reducer, to create single combined reducer.

```
export function attachmentReducer(
  state: UploadState = initialState,
  action: UploadAction,
  actionTypeName: string,
): UploadState {
  const { actionNameIdentifier } = action;
  if (actionNameIdentifier !== actionTypeName) return state

  switch (action.type) {
    case DraftActionTypes.UploadAttachments: {
      return attachmentsAdapter.addAll(action.payload, {
```

[3] In a real world application, normally I would include a param for name of the file. However, for the simplicity of this chapter, I decided to leave it out.

```
12          ...state,
13          loaded: true,
14        });
15      }
16    }
17
18    return state;
19  }
```

You will notice that in the above code, we have created a param for actionTypeName in our reducer. In our action, we are passing in a similar unique identifier called actionNameIdentifier. In our reducer we compare the actionTypeName created in the reducer, against the actionTypeName in the action, in order to re-use reducer logic, for any feature state we would like to use it.

126.4 Combining Higher Order Reducer with Feature State Reducer

Higher order reducers are fantastic. However, many times, the higher order reducer only solves one specific problem of state with regards to a particular component. So, without us being able to combine multiple higher order reducers within the app, a higher order reducer would be worthless. That being said, we would be able combine reducers by doing the following:

```
1  import { reducers } from './reducers';
2  @NgModule({
3    imports: [
4      ...
5      StoreModule.forFeature('userModule', {
6        userReducer,
7        attachments: attachmentReducer('userReducer'),
8
9      })
10   ],
11   ...
12 })
13 export default class UserModule { }
```

Now, any action that is made that matches the name for feature state we are working on, will update the state for the respective reducer.

126.5 Re-Using Effects

Effects are really a bridge from an action, to service, back to an action again. We are therefore able to re-use effect by simply passing in the actionNameIdentifier and making sure that it makes it's way back to reducer once again.

```
@Effect()
  uploadAttachment$ = this.dataPersistence.fetch(
    DraftActionTypes.UploadAttachment,
    {
      run: (action: UploadAttachment, state: UploadState) => {
        const { actionNameIdentifier } = action;
        return this.filesService
          .uploadFile(action.payload, FileLocation.ATTACHMENTS)
          .pipe(
            map((file: FileMetadata) => {
              return new AttachmentUploaded({
                id: action.payload.id,
                file,
                actionNameIdentifier,
              });
            })
          );
      },

      onError: (action: UploadDraftAttachment, error) => {
        console.error('Error', error);
      },
    }
  );
```

126.6 Ending Notes

There is some discussion along the lines of whether, or not this pattern makes sense. Arguably some scenarios where one might be tempted to use a pattern like this, might not be ideal. Therefore, it is worth noting to re-assess the reason for which you would like to re-use state. Perfectly valid reasons for which to use a re-usable reducer includes pagination, sorting, and potentially filtering.

127 Ngrx Effects

127.1 Ngrx Effects - A Primer

Ngrx effects can be one of the more ambigious parts of the ngrx stack. They are by definitition something that is supposed to happen when something else has happened. It will listen for a particular action, and true to Ngrx, return an observable. In this observable one will have the option to do whatever they want as well as publish(return) to the action stream.

The line, however, can be blurred, however, as to what the difference is between an ngrx/-effect and a ngrx/store. It is therefore important to distinguish for architectural reasons. In addition, it can be difficult to determine the different use cases wherein someone would use an effect. It can indeed be a slippery slope wherein when to use an effect.

127.1.1 Code Example

```
@Effect({ dispatch: false })
 userDeleted$ = this.dataPersistence.fetch(
   UserActivitiesTypes.UserDeleted,
   {
     run: (action: UserDeleted, state: UserStateModelState) => {
       this.snackBar.open('User Deleted', 'Ok', {
         duration: 2000,
         verticalPosition: 'top',
       });

       return null;
     },

     onError: (action: ActivityDeleted, error) => {
       console.error('Error', error);
     },
   }
 );
```

This code example, is a great example as to when someone might use an effect. As we can see here, we have an action that is being triggered for when a user is deleted. We then

have an effect who's sole purpose to have a snack bar open when action is called.

127.1.2 The Three Pillars of an Effect

As we discussed earlier, knowing when to use an effect can be a tricky thing to decipher. Think of it as having the ability to do the following:

1. Hook into State.

2. Ability to do whatever when action is called.

3. Publish an action back into the state management cycle.

In our scenario, for deleting a user we had two effects:

We called a GraphQL service to delete a user. We then retrieve the result returned by the GraphQL service, and trigger another effect, which is our snackbar effect. Yes, this logic can potentially be handled by our view layer within our component. In addition, we can use the service directly. However, having all of this logic encapsulated in our effect makes everything very clean.

127.1.3 Further Reading

While this is out of the scope for this book, I would like to suggest further reading. Naturally, they would be articles that I would discuss on:

- Use cases for using Effects.

- Use cases for NOT using Effects.

The article for use cases with regards to using effects is https://medium.com/@tanya/understanding-ngrx-effects-and-the-action-stream-1a74996a0c1c. The article with regards to use cases in which NOT to use effects is https://medium.com/@m3po22/stop-using-ngrx-effects-for-that-a6ccfe186399 These two articles along with the information from this chapter. You should be well along your way for architecting solid ngrx/effects. I personally do not have patience for the article on, "Stop Using Ngrx effects for That". However, it is nonetheless the best article on the topic, if you can swallow it.

128 The Case for Using Ngrx/Entity by Default

128.1 A Synopsis of Normalization

The idea of normalization with regards to relational databases(a database structured to recognise relations among stored items of information) is to eliminate data redundancy, and anomalies with regards to Insertion, Delete, and updating. Within a front end setting, it is mostly the backend that deals with a large portion of eliminating data redudancy, and CRUD anomalies. However, there are many use cases still with regards to Insertion, Delete, and Updating anomolies, that can be solved by using normalization within ngrx/platform.

128.2 Why Ngrx/entity is Exciting

@ngrx/entity for me personally, is the most exciting part of ngrx/platform for two reasons:

1. It is a library based on computer science fundamentals of building relational databases.

2. It actually introduces something novel to the Redux ecosystem (something I found that the Context and Hooks API didn't even give that sort of freshness).

128.3 Dis-secting a Real World Example of ngrx/entity

128.3.1 CodeBox Interface

Let's imagine that we are building a Pixel Illustrator application. Something similar to the following. One of our interfaces for the state contained within the codeBox, would look something like this:

```
1   interface CodeBox {
2     id: string;
3     color: string;
4     xPosition: number;
```

```
5     yPosition: number;
6   }
```

Let's imagine we have an array of CodeBox interface(AKA CodeBox[]) that we would like to use. If we want to select a specific slice of data within our database table, things can be a little bit tricky. We would need to loop through all of the data, and then find the appropriate set of data with the id/value that we want.

Instead, what would make our data table more efficient, is if we have an array of ids that corresponds directly to our dictionary of data. In our store we can add some extra key values such as selectedId. This way, we can simply say the data that we would like is based on one, or two parameters passed to a function. This can be re-used in other stores, as data is structured the same way, and selected in the same fashion.

128.4 ngrx/entity follows this pattern

Ngrx/entity is a library which follows this pattern, so that data used within ngrx/store is normalized. This means, within the context of using JSON data, that we have a dictionary, combined with an array of ids. The index of the ids directly correlates with the key/value in the dictionary. This is very powerful, and I will prove it by bringing up a couple of situations.

128.4.1 The Power of Ngrx/Entity Deep Dive

Scenario One: Updating one dictionary data set. Let's imagine that we were to load our Pixel Illustrator a set of coordinates to populate the pixel illustrator. However, let's say we wanted to update one particular piece of that data set, by only a smaller portion of data, when loading entire data set, and then a larger portion when expanding details for particular data set. Ngrx/entity allows us to maintain the same store, but simply update the once piece of data in the dictionary.

Scenario Two: Let's say we have two independent components pulling data from the same store, but we do not want them to interact with each other. We would need to create a unique id for both that would allow us to follow this pattern. ngrx/entity naturally allows us to pass in a unique id, that allows us to accomplish this correlation id pattern. This is such a powerful pattern.

Scenario Three: Let's say we have a data table, and we would like to know which set's of data we have selected. We have the ability to pass in id, into another chapter of data store, called selectedIds. So we can have a checkbox, select multiple checkboxes, and know which data sets have been selected.

Scenario Four: Let's say we want to delete, update, or upsert a certain subset of data. Being able to pass in the id only, makes this process infinitely easier, and the api more manipilating data, more re-usable.

These are just four real worlds scenarios, and the list goes on.

128.5 Choosing Ngrx/entity as the Default

This leads to the point this chapter is trying to make. In any situation with data, odds are that you are going to update, delete, upsert, or pull in more data for a specific subset of data. This means, that anytime that you are going to use more data, using ngrx/entity as the default makes sense. The Nrwl CLI for ngrx/store follows this pattern, and it is highly reccomended.

129 Ngrx Entity

The Ngrx repo until recently had many similar functionalities to your regular redux app. It included actions, reducers, selectors. However, there has been efforts to go ahead and create libraries for aspects of ngrx that can perhaps be re-usable. One of these is @ngrx/entity.

129.1 Ngrx Entity at a High Level

At it's core, ngrx entity is an API for manipulating and querying entity collections. In particular:

1. Reduce boilerplate for creating reducers that manage a collection of models.

2. Providing performant CRUD operations for managing entity collections.

3. Extensible type-safe adapters for selecting entity information.

This architecture works really well when creating data as a single source of truth. For instance, let's say in your application, you have a data table on every page that pulls in data. Throughout every page, you have a way of manipulating this data. Using ngrx/entity will allow for this architecture to be fluid, and have all manipulation of data be within a singular area.

129.2 Example of Ngrx Entity

Within our app we the ability to illustrate a pixelated charachter using pixels. Every time that a pixel within the grid is selected, we are going to add it to our store. This store is going to be used to display the code version of the app. In addition, we are going to have to remove the pixel when clicked on within our store. In addition, if we have selected a new color, and we select a new pixel with that color, that pixel should be updated with the proper color. What we have just described is a perfect CRUD app.

129.3 Installing ngrx/enity

```
npm install @ngrx/entity --save
```

129.4 ngrx/entity - A Step Back

Let's step back for the time being and look into what ngrx/entity actually does. Ngrx/entity will create a list of ids and a dictionary of entities. Let's brush up on entity, list, and dictionary:

entity: In relation to a database , an entity is a single person, place, or thing about which data can be stored.[a]
List: AKA an array.
Dictionary: Collection which is unordered, changeable and indexed.

[a] https://whatis.techtarget.com/definition/entity

That being said, a sample ngrx/entity data structure will look like this:

```
ids: [
  '3QOZBAAAQBAJ',
  'y4nmOe0-WDOC',
  'lS5SAQAAIAAJ',
],
entitites: {
  '3QOZBAAAQBAJ': {
    name: 'Lebron',
    id: '3QOZBAAAQBAJ'
  },
  'y4nmOe0-WDOC': {
    name: 'Kyle',
    id: 'y4nmOe0-WDOC'
  },
  'lS5SAQAAIAAJ': {
    name: 'Sarah',
    id: 'lS5SAQAAIAAJ'
  }
}
```

129.5 Adapter Pattern - A Primer

Before we go ahead and discuss what an ngrx/entity adapter is, let's go through a quick primer on the adapter pattern in general. Per the GoF book [1] an adapter pattern:

"is a software design pattern (also known as Wrapper, an alternative naming shared with the Decorator pattern) that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code."

129.6 Introducing Ngrx/entity adapter

The ngrx/entity adapter, similarly, will take in data, and wrap it inside of ids and entities. So the adapter can be considered as something that will modify the data. The default adapter that comes with ngrx/entity, takes two default values:

- selectId: A method for selecting the primary id for the collection.
- sortComparer: A compare function used to sort the collection. The comparer function is only needed if the collection needs to be sorted before being displayed. Set to false to leave the collection unsorted, which is more performant during CRUD operations.

The selectId is the more important default value. This will be the default UUID that will be used within the app. The general idea is that some sort of id will be returned by the database for that particular item. One will then be able to use that id for all crud operations. In addition, most likely pass in that id for your Rest Service, or GraphQL query.

129.7 Ngrx/entity Adapter Example

Creating an example adapter, might look something like the following:

```
1   export const adapter: EntityAdapter<any> = createEntityAdapter<any>({
2   selectId: (emailStore: any) => emailStore.id,
3   sortComparer: false,
4 });
```

There will then be a series of adapter methods returned by ngrx/entity. Without going into them in detail, here they are:

- addOne: Add one entity to the collection

[1] A.K.A. Design Patterns: Elements of Reusable Object-Oriented Software

- addMany: Add multiple entities to the collection
- addAll: Replace current collection with provided collection
- removeOne: Remove one entity from the collection
- removeMany: Remove multiple entities from the collection
- removeAll: Clear entity collection
- updateOne: Update one entity in the collection
- updateMany: Update multiple entities in the collection
- upsertOne: Add or Update one entity in the collection
- upsertMany: Add or Update multiple entities in the collection

129.8 addOne example

In our app we will be using a series of different ngrx/entity methods. However, we will be using addMany as for the most part many of the methods are very similar.

Let's focus on a specific reducer chapter within our app.

```
case gridTypes.added {
  return {
    adapter.addOne(action.payload, state)
  }
}
```

That would really be it ^. It will insert a unique id for that specific pixel. In addition, it will go ahead and new entity within the entities object.

129.9 Identifying Different Entity Selectors

So far ngrx/entity has given us an adapter, which allows us to choose the id we would like to use for our entity dictionary, as well as our id list. However, what if we wanted to retrieved all of our ids, or all of our entities? It can be a bit cumbersome. So thankfully enough, you saw it coming, ngrx/entity offers entity selectors out of the box.

```
// get the selectors
const { selectIds, selectEntities, selectAll, selectTotal } = adapter.
    getSelectors();

// select the array of user ids
export const selectUserIds = selectIds;

// select the dictionary of user entities
export const selectUserEntities = selectEntities;

// select the array of users
export const selectAllUsers = selectAll;

// select the total user count
export const selectUserTotal = selectTotal;
```

It is important to recognize that these selectors will not actually produce state on their own. What they do is return a function when used in conjuction with the createSelector function, will return the appropriate entity.

129.10 How to use getSelectors

These selectors are then meant to be used with the createSelector function. The following is an example:

```
export const selectUserIds = createSelector(
  selectUserState,
  fromUser.selectUserIds
);
```

Now one will have a state that specifically returns ids for a specific list.

129.11 Using updateOne

Just to show how convenient ngrx/entity is. Let's say in your app you wanted to update a specific field. For instance, in our app it is going to be the color for a specific pixel.] All you would need to do is the following:

```
  case gridTypes.updated {
    return {
      adapter.updateOne(action.payload.id, state)
```

```
    }
  }
```

The only difference between the signature for addOne and updateOne, is that with updateOne, you are just supplying the id to be updated.

129.12 Wrapping Up

Suffice to say that using ngrx/entity will greatly increase the efficiency of your app. Being able to use a CRUD app in this fashion, will simplify the architcture across the app, wherein this state can be used in numerous places.

130 State Management - Properly Unsubscribing

In Angular, when using ngrx when trying to pull in data, using the async pipe is the preffered approach. It will handle both the subscribe and the unsubscribe from the observable pipe for you.

```
<div>{{ observableStream$ | async }}</div>
```

130.1 What to do When Async Pipe is Not an Option

There are times When the process of subscribing and unsubscribing must be managed manually. In particular in situation where there is data manipilation that must happen within the component. In this case the recommended approach is to create an observable that emits when the component is destroyed, and use the rxjs takeUntil operator to handle the act of unsubscribing for you.

130.2 Using takUntil Example

```
import { Subject, Observable, pipe } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

import { MyService } from './my-service';

@Component({
  selector: 'my-component',
  template: `
    <div>
      Count: {{ count }}
    </div>
  `,
})
export class MyComponent {
  private destroy$ = new Subject();
  count: number;

  constructor(private myService: MyService) { }

```

```
20    ngOnInit() {
21      this.myService.observableStream$
22        .pipe(takeUntil(this.destroy$))
23        .subscribe(count => this.count = count);
24    }
25
26    ngOnDestroy() {
27      this.destroy$.next();
28      this.destroy$.complete();
29    }
30  }
```

130.3 takeUntil in Depth

Let's go back in depth to takeUntil to see what we are doing:

```
private destroy$ = new Subject();
```

We are create a subject which, of course, acts as both an observer and an observable. It allows us to call next and complete. We can then pass the value of true to next:

```
ngOnDestroy() {
  this.destroy$.next(true);
  this.destroy$.complete();
}
```

Which is called when the component is destroyed. We then run takeUntil, which will unsubscribe as soon as it is passed a true value.

```
.pipe(takeUntil(this.destroy$))
```

131 Re-Usable State - An Anti-Pattern

Creating re-usable state in an Angular application might be one of the most singular important architectural decisions you might also make. In addition, it will probably be the longest lasting architectural decision you might make, as redux is something which is pretty agnostic across many different frameworks.

131.1 Why Create Re-usable State?

If there is a singular component that is going to be used across a different page, having re-usable state, will greatly simplify the architecture. The logic for reducers can be created once. That logic can then be re-used numerous times within your app. However, I would like to express how having re-usable state in @ngrx/store is an anti-pattern.

131.2 Re-usable State - An Anti-Pattern

Let's imagine that you have a re-usable data-table, that you would like to use on numerous pages. There are certain pieces of logic that you want to use with your state. For instance, you want to create a reducer to determine which rows have been selected, and if all have been selected. If all has been selected, then it is moved over to the selected key/value. This logic you have decided should be mover over to state, so that it can re-used within the data table, so that it can be passed around and re-used within the app time and time again.

The only issue with re-usable state, is that as soon as you are using re-usable state, you are recognizing that the component has to be re-usable. As soon as you are saying the component is going to be re-usable, you are recongnizing that there is need for there to be a dumb component and a smart component. As soon as you are saying there is going to be a dumb component, then any logic relating to the interface should remain within the components logic it's self. Therefore, the only state that you will be needing, is the data that is loaded. That part of state is simplified to a great extent, to where it makes sense to have state unique per each page.

132 Facade Pattern

132.1 What is the Facade Pattern?

The facade pattern is a classic. Anyone who has read the GoF book [1] knows that it is a mainstay of computer science. Quoting from the GoF book:

"A facade is an object that provides a simplified interface to a larger body of code, such as a class library."

132.2 A Look at your Typical Non Facade State Pattern

This pattern is particularly advantageous when it comes to ngrx actions. If I may, let's imagine we have the following action:

```
  // choose-size.actions.ts
export class LoadChooseSize implements Action {
  readonly type = ChooseSizeActionTypes.LoadChooseSize;
  constructor(public payload: any) {}
}
```

Now any time that we have to call an action we have to do two things:

1. Have a store select within the component.

2. Call a dispatch.

```
  chooseSize: Observable<any>;
  // choose-size.component.ts
  import { Store } from '@ngrx/store';
  constructor(private store: Store<any>) {
      this.chooseSize = store.select('chooseSize');
  //..
  merge(
    this.updateSize\$.pipe(
      map((value: any) => new ChooseSizeUpdated(value))
    )
  ).subscribe(action => {
```

[1] which if you haven't you should probably take a look.

```
    store.dispatch(action);
  });
```

Obviously, this is quite a bit of overhead. Using the facade pattern let's see if we can simplify this process

132.3 Create the Facade Service

With a facade pattern, we have the ability to take the following two items:

1. Store select
2. Call a dispatch.

and put them into the into our facade.

The facade should be treated as a service, and we will create a service folder for our facade to go into.

The facade pattern looks like the following:

```
export class ChooseSizeFacade {
 constructor(private store: Store<any>) {}

 chooseSize$ = this.store.select('chooseSize');

 UpdateChooseSize(ChooseSizeFormPayload): void {
   this.store.dispatch(new ChooseSizeUpdated(ChooseSizeFormPayload));
 }
}
```

132.4 Hooking Facade Into Component

Now to call our action, all all we have to do is call the ChooseSizeFacade service, and appropriate method.

132.5 Why Even Bother Creating a Facade?

It might seem like a bit of overhead working towards creating an additional file as a facade for our actions. However, it keeps our component clean, and removes the need of keeping overhead of the different stores that our component is using.

133 State Directory Structure

When one begins an Angular project for the first time, it can be increasingly difficult to manage ngrx/store. State, while it should ideally be tied to a feature, as the app moves on, might not neccesarily be tied to a specific feature, or page. In addition, +state by nature, as a single giant object, is global by nature. Also, it takes up a large portion of any app. It makes sense to put all state in a single repository, so that state within the app can be transparent(go into this a bit more). Finally, for testing purposes, state is a very large chunk of business logic for app, and deserves it's own module for bundling, and testing purposes.

133.1 Data Access Folder/File Structure

State being a way of accessing data, an appropriate name for the folder/file tree for state makes sense to be called data-access. It will look something like the following:

Most notably, all of the state related code is contained within a single folder. By doing so, it solves all of the above three issues:

1. State is global, and therefore can now be used by multiple features

2. We have the ability to run ng test –project=px-illustrator-data-access-code-box and it will run code specifically for this data-access feature state

3. By globalizing naming convention, we can streamline naming convention of all global files intended towards working towards the same purpose. Namely, our data-services, and features.

4. In addition, it alleviates the potential issue of circular dependencies. If, for instance, we have feature folder A, and feature folder B. B might need state from A, and A might need feature state from B. By keeping all of our state global, it helps us circumvent this circular dependency problem.

134 Correlation ID Service

We have addressed in a previous chapter whether, or not when to use state. One of the more peculiar situations within an Angular application is file upload. Generally, it is for the following three reasons:

1. There is a before, and after state. What does the file look like before the upload, and what does it look like before the download.

2. Depending on scenario, we might have multiple components on the page, and therefore need to make sure, that the state of one, does not affect the state of the other.

3. State is contained within a single component, albeit there might be a number of different components on the same page. It would seem state is superfluous in this scenario.

4. Following dumb/smart component architecture, in order to keep application dry, we are going to need to inroduce our state from some outside source, using event emitters.

The scenario that commonly applies to file upload can happen in many other situations as well. While @ngrx/store is something which is a good idea for the majority of any enterprise application, it is not neccesarily the right choice in this scenario.

134.1 Identifying Bloat of @ngrx/store

@ngrx/store bloat is real. To put into perspective, it requires an action, a reducer, a facade, in better architectured solutions, ngrx/entity, and selectors. In addition, state isn't naturally re-usable, as it is an object. Lastly, and most definitely, not least, special nomenclature must be put in place to make it, so state can be re-used in the future.

134.2 Architectural Danger of Using a Service

Naturally using a service in this scenario makes the most sense. Services can have a direct 1:1 relationship for as what they need to do. In addition, it can help circumvent circular

dependencies in numerous scenarios, by keeping heavy logic from the store. However, services by default break away from the cookie cutter api's brought to you by the @ngrx life cycle. It is important to understand that if services are created, that one needs to add proper due dilligence to make sure services are properly used.

1. Documentation for each method.

2. Functions kept small.

3. Readme.md in root of service lib, specifying methods used.

4. Heavy Unit Testing

That would be it. Ideally these services are created sparingly, but they have their advantages, wherein @ngrx/store is obvious bloat.

Many times in a given project, there is the potential for scope creep. That is, product might present the data needed in a given use case, as only being used in a certain fashion, for the entire lifecycle of the app. As time goes on, however, this might not be true. If it is important for the developer to map out potential areas wherein this might not be true. If you see the potential of the business logic to expand beyond what you are working on, take that into consideration. If not, then seriously consider using a service.

135 Integrating a Component with @ngrx/store

Another chapter has been dedicated solely to integrating a component with @ngrx/store. This is because it is a cookie cutter process. Observables are notoriously known for abstracting events, and therefore allow the same code to be repeated time, and time again. The following is what can be expected to be repeated time and time again in your application, after initially generating files.

Just for redundancy sake, there are six steps, that go into setting up state with any given component, that are handled by the nx ngrx cli:

1. Store
2. Action
3. Reducer
4. Initial State/Enums[1]
5. Effects

All that is left, is for us to now to do three things component side:

1. Set ups actions in view layer (i.e. HTML) + action type.
2. Select store, so that it can be used in component.
3. Setup subject in component, so that it can be used with actions in view layer.
4. Setup reducer in component, so that it can be used with
5. Set up subscribe in component(to transfer model to controller) [2]

This is pretty standard, and this will be done in every standard application. It is this simple,

[1] In some other framework it might be constants
[2] A subscriber is rarely setup in the smae component setting up the store to begin with

and will follow this formula, and yes, you should be in shock in how seamless integrating state management technology is at this point, because I am.

135.1 Re-iterating purpose of book

I would just like to re-iterate, that the point of this book, is to go through the architecture of the entire angular ecosystem. So that someone can read this book and have the confidence that they are building their Angular app the proper way, and that they do not have to look elsewhere. That being said, I will not be going into the whole code base of what is happening. However, I would like to show an example of along the lines of what will end up happening.

135.2 Set up action

For our action, we are going to create an action type, and simple action, that is taking in a payload. Your action should not be doing anything fancy, and if it does, then you are not doing it right.

```
ChooseSizeUpdated = '[ChooseSize] Data Updated'

export class ChooseSizeUpdated implements Action {
  readonly type = ChooseSizeActionTypes.ChooseSizeUpdated;
  constructor(public payload: any) {}
}
```

135.3 Set up store

In any given component, we must setup a select for our store, in order to tell @ngrx/store, how we plan on interacting with it. The following is cookie cutter code, like the rest of this chapter, that will be repeated throughout any @ngrx/store process.

```
this.chooseSize = store.select('chooseSize');
```

135.4 Creating a subject

To re-iterate, what is a subject? It is both an observable and an observer?

1. ObserverâĂŁâĂŤâĂŁIt has the next, error, and complete methods.

2. ObservableâĂŁâĂŤâĂŁIt has all the Observable operators, and you can subscribe to him.

Therefore, in an Angular setting, using @ngrx/store, subjects are our friends. It allows us to have a singular event handler, to be used by all html event handlers within component. In addition, it gives us a subscribe. The general pattern in an Angular app will be as follows:

1. Create subject in component
2. Setup ElementRefs in Component HTML
3. Merge subjects into singular subscribe
4. Setup Reducer for action

135.5 Creating a Subject - Code Dive

135.5.1 Creating Subject - In Component

```
1  import { Subject } from 'rxjs/Subject';
2
3  updateSize$ = new Subject();
```

135.5.2 Setup ElementRefs in Component HTML

```
1  <input matInput placeholder="Columns" #columns>
2  <input matInput placeholder="Rows" #rows>
3  <input matInput placeholder="Pixel Size" #pixelSize>
4  <button (click)="updateSize$.next({columns: columns.value, rows: rows.
       value,
5    pixelSize: pixelSize.value})"
```

We are now going to feed our updateSize subject into an rxjs merge, and map, to make it future proof.

135.5.3 Merge Subjects into Singular Subscribe

```
1  merge(
2    this.updateSize$.pipe(
3      map((value: any) => new ChooseSizeUpdated(value))
4        )
5      ).subscribe(action => {
6        store.dispatch(action);
7    });
```

135.6 Setting up a Reducer for our App

```
1   case ChooseSizeActionTypes.ChooseSizeUpdated: {
2     return { ...state, ...action.payload };
3   }
```

135.7 Wrapping up

We have gone through the full gamut, of what it is going to look like adding new elements of state into your app. This is the one part with regards to state that will have to be done manually. Who knows, maybe down the line, we will have more sophisticated technology, that will allow us to have a command line interface similar to what we have now using the Angular CLI.

Next let's talk unit testing.

136 Charts

Having charts in one's application might not seem like an architectural decision. In particular, there is a very popular library library out there called d3. This might not be the right place to bring it up as it is Angular agnostic. However, data can be something that can be quite abstract. It might be tempting to tag it all into a singular service, and load it in the appropriate component. However, using the power on Angular, it would allow for the compartementalization of graphics, and makes Angular a very powerful tool for the job.

136.1 Install D3

```
npm install --save d3
npm install --save-dev @types/d3
```

136.2 Interfacing D3

When using any framework in general with a graphics library, the proper approach is to interface through that component. So, for instance, instead of keeping the framework really abstract, we can tie it to the DOM of the component. When we add an arc component, it adds an arc to the component. When we add a tooltip component, to the app, it adds a donut chart. I would just like to touch lightly on how this would be done. It is also important to note, that much of what will be discussed in this chapter, such as models, or visuals are native to Angular. However, being that we are bringing over something which is not in an Angular setting, I will be discussing everything, to remove any confusion.

136.3 Simplifying an Interface in the Context of Angular

We've discussed previously the concept of dumb and smart components. That is, having a component purely for visual purposes, and progating the need for retrieving data, and handling events to the smart component. This is a very similar phenomenon, with d3. Only the smart componet is going to be the donut-chart, and child compoents, will be arcs, and legends. The second cornerstone to understanding interfacing d3, is to create a service,

that will handle any non-feature specific d3 logic. For instance, let's say we would like to click on an arc, or hover over, and have a specific function, we can turn that into a core service. Therefore, moving forward, we would be able to attach this logic over to another component if need be.

136.4 Re-building Pixel Grid, interfacing d3 using Angular

Let's create a d3 rect component. Instead, we are going to call it the pixel component. Inside of our pixel component, we are going to interface it with width and height.

136.5 ApplyClickableBehavior Service

In addition, we are going to want to create a service that can be used on our specific element, as well as on other elements. It will look something like the following:

136.6 Hooking up Services to Directives

Now that we have our services, we are going to hook up services directly to our directives. The directives are going to by default hook into the native element of

137 Benefits of Unit Testing, TDD, and BDD

It truly makes sense for a definitive guide to deep dive into what the benefits of unit testing would be. From a developer perspective:

1. Give insight as to what unit testing should accomplish, so that a developer, can intuitively decide when appropriate to write a unit test.

2. Give confidence to developer as to why they should write unit test.

From a management perspective, to help introduce unit testing, for those that might be less inclined towards unit testing.

137.1 What is Unit Testing?

A unit in a best case scenario, is a function that always gives you the same result for a given input(pure function) Testing that unit, is to make sure that the expected result happens when running that function.

137.2 Benefits of Unit Testing

1. Refactoring. Change code once, and see everything else is working.

2. Focus(See TDD item #2)

3. Helps understand design of code working with(See TDD item #3)

4. Instant visual feedback that code works as expected.

5. Documentation (See TDD #4)

6. Helps with code-reuse. Ability to re-use code and tests. Tweak tests accordingly.

7. Testable code
    a) Modular
    b) Maintainable
    c) Readable

137.3 What is TDD (Test Driven Development)

1. Start by writing a test.
2. Run the test, and any other tests. At this point, your newly added test should fail. If it doesn't fail here, it might not be testing the right thing, and has a bug in it.
3. Write the minimum amount of code required to make the test pass.
4. Run the test to check the new test passes.
5. Optionally re-factor your code.
6. Repeat from 1.

137.4 What is BDD?

Typcially when unit testing, a particular function at a later date can change it's impementation. For instance, if we have a counter function, the counter can be changed to start at 5 instead of 0, breaking the expect statement of 1. In BDD we focus on the intended behavior, instead of the expected result. The following is a great explanation:

137.5 The Benefits of TDD

1. Higher Test Coverage
2. Focus
    a) Focus one part of an issue at a time.

b) Allows one to realize when to stop coding.

3. Interfaces. Allows you to think more organically about what should be put into interface. Allows for interface to be written bottom up(implementation, behavior) instead of top down(behavior, implementation).

137.6 What is BDD - Code Example

```
// Non BDD
describe('Counter', () => {
  it('should increase count to 1', () => {
    const counter = new Counter();

    counter.tick();

    expect(counter.count).toEqual(1);
  });
})

// BDD
describe('Counter', () => {
  it('should increase count by 1 after calling tick', () => {
    const counter = new Counter();
    const expectedCount = counter.count + 1;

    counter.tick();

    expect(counter.count).toEqual(expectedCount);
  });
})
```

137.7 The Benefits of BDD?

If at a later time, the counter(as seen above), for instance, has to change based on requirements(starting at 5, instead of 1), it will not affect the unit test.

137.8 What, When, and How

Unit testing, is the what. TDD is the when, and BDD is the how.

Convincing the skeptic might be difficult. The following are some steps to convince the skeptic.

1. Too Simple to Fail? If this is used regularly in our application, doesn't it make

sense to spend 5 - 30 minutes testing it?

2. Too difficult to Unit Test, won't it be time consuming? Good indication that it needs to be re-factored, proving point of benefit of unit test.
3. Too Time Consuming
    a) App will produce more bugs
    b) Lower Quality Code
    c) Lack of confidence. Perhaps bugs under the radar.
    d) Lower design.

138 Unit Testing Performance

Unit Testing isn't neccesarily one of those things that we tend to equate with performance. There are tools that allow one to a unit test by a specific folder. In addition, there is parallel unit testing, for the pipeline, so that unit tests can be sped up. However, as apps get larger, so do their unit tests. From a developer perspective, it is valuable to run all unit tests, to make sure that when larger impacting edits are made, that none of the unit tests are failing. Unit tests will be run when a pr is being made. In addition, they are going to be run when deploying. So there is enough going on, to say that being conscious of performance boost is something which is important.

138.1 Component and Integration Testing

It is quite common that many enterprise apps will take advantage of integration testing within their unit tests. However, creating the component within the unit test is intuitively, and is the most expensive task one can do within a unit test. Avoiding creating a component unless needed, is the ideal. For instance, the ideal scenario when creating a component, looks something like this:

```
describe('BannerComponent', () => {
  let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeDefined();
  });
})
```

Here we are setting a TestBed, configuring the test module, and then compiling the component. While, in practice, there is nothing wrong with this, but for every run, this will re-compile our components. So, if there is a way we can get around re-compiling our component, this would obviously go a bit quicker. There are a couple of approaches. One of them involves using something called ng-bullet. I am bit skeptical of using something like this, as I can see it causing some issues amongst the build. The second approach, is to only compile the component if you need it. It would be a good to suggest Jest, which is a faster way of running unit tests. In addition, for current karma/jasmine unit tests to be mindful of how you are writing unit tests.

138.2 Component Testing

There are three different ways of unit testing a component:

1. Isolated Unit tests
2. Shallow Unit Tests
3. Integrated Unit Tests

Of the three, Isolated Unit Tests, are the most performant. It will not cause the component to re-render itself. However, by using an isolated unit tests, we are simply testing the logic of the component, without testing how it interacts without actual html. This usually only works in certain scenarios.

In most scenarios, there will be a requirement of running the TestBed, which can cause performance issues when running a large amount of tests, and can be dealt with.

138.3 Running tests in Parallel

The next item in the checklist, is to separate your tests into separate modules, so that they can be run separately. You will then be able to run the tests in parallel. This will allow for the tests to run quicker.

138.4 Karma Parallel

Karma Parallel is an npm package that be used to run unit tests in parallel. It would require for the karma config to be updated accordingly. The one downfall of the karma confi. Tjose

138.5 Ng-Bullet

There is a fantastic library that has been written to accomodate for some performance boosts with regards to Angular Unit Tests. In short, there has been discussion around increasing the performance of Angular Unit Tests for quite some time. The way that the new Angular compiler will work, will greatly decrease time it will take for Karma/Jasmine unit tests regardless. I would imagine this is the reason why those working on Angular, have decided not to comment. However, Ng-bullet is a possibility to be used for those that seriously need the performance boost in their Angular App.

138.6 Style Cleanup

The biggest improvement with regards to performance boosts comes with improving memory leaks. The largest memory leak is caused by CSS. If you do not clear your css in your unit tests, your karma tests will consistently append hundreds if not thousands of ¡style¿ tages to your body. This will incredibly slow down how fast your unit tests run. This can be alleviated by adding an afterAll to your unit tests:

```
export function cleanStylesFromDOM(): void {
  const head: HTMLHeadElement = document.getElementByTagName('head')[0];
  const styles: HTMLCollectionOf<HTMLStyleElement>
  | [] = head.getElementsByTageName('style');

  for (let i: number = 0; i < styles.length; i++) {
    head.removeChild(styles[i]);
  }
}
afterAll(() => {
  clearStylesFromDOM();
});
```

The above functionality will improve performance of unit tests by five fold.

139 Fixture Vs. Debug

There are use cases, wherein accessing the dom as we've mentioned makes sense from a unit testing perspective. We want to make sure that the unit test that we've called, is actually called in the DOM as well. In Angular, there is the ability to tap into the native element, which is a good old fashioned DOM Element. Alternatively, there is the option to tap into the debug element. The debug element offers a wrapper on top of nativeElement.

There are a couple of methods that a debugElement offers over a nativeElement.

1. componentInstance
2. debugElement
3. query(By directive)

140 Sass Unit Testing

140.1 When Does Sass Unit Testing Make Sense?

One of the concerns with any architecture, is over engineering. With regards to unit testing Sass, to what extent should one unit test? Should it be for every class, for every function, any core class used within a framework?

After much back and forth the answer should be, a core class, which is used across the app. However, the actual class that is re-used should not be unit testing, because it can be re-used in the wrong context. In addition, the concept of combining two classes is closer to OOP than it is to functional programming.

The ideal approach would be to unit test functional scss that is used as a core style. The convention should be that when using a core style, such as padding, or a breakpoint.

140.2 The Benefits of using Functional Sass as a Convention

What would be the benefit of using Functional Sass as a convention? For starters within an app, skimming through css, it all looks the same. Even within our architecture using BEM, it can be hard to determine the difference between a core class, a component specific style, a media query. In addition, the importance, as well as degree of impact with regards to functional progamming.

While this book will not offer a complete paradigm, for a structured design pattern within scss, in order to design discernable sass. We do have a way to discern what is a core style, and what is not. This is by using functional Sass. The convention, is that for a specific app, all core style use functional sass. Similr to when we use sass error reporting. If a pr goes out, and it is not function used for styling, then comment is made that, "Per convention, this is a core style, and should be using the appropriate function".

140.3 Within a Design Language System, Choosing Core Functions

With a core framework, specifically built on top of a design language system, the following is what a functional DLS core framework would look like:

1. Buttons
2. Chips
3. Colors
4. Data Table
5. Font
6. Grid, Padding, and Border
7. Icons
8. Input, Date Picker, Checks, Toggle, Radio & Tabs
9. Line Style, Elevation, Element Styles
10. Side Nav
11. Toolbar

140.4 Unit Testing Within Our Specific App

Being that we are going to be creating sass functions for our core theming, it would make sense to unit test them as well. If they are going to be use in 10, or more places per each app, then we would like to make sure, that they are indeed working in the fashion that they should be.

140.5 Using Sass True

Sass True, is a set of Sass unit tests, written in Sass, so that they can mimic the usual describe, it, assert, and expects, one can expect from a usual unit test. The following is an example of a unit test one might make for a typical mixin.

```
// Test CSS output from mixins
@include it('Outputs a font size and line height based on keyword') {
  @include assert {
    @include output {
      @include font-size('large');
    }

    @include expect {
      font-size: 2rem;
      line-height: 3rem;
    }
  }
}
```

It should all be very familiar with your classic Mocha test.

140.6 Installing Sass True

```
npm install sass-true --save-dev
```

140.7 Setting up a scss.spec.ts

We are going to set up our own jasmine sass-test runner, that will pick up on all sass unit tests within directory. It will look like the following:

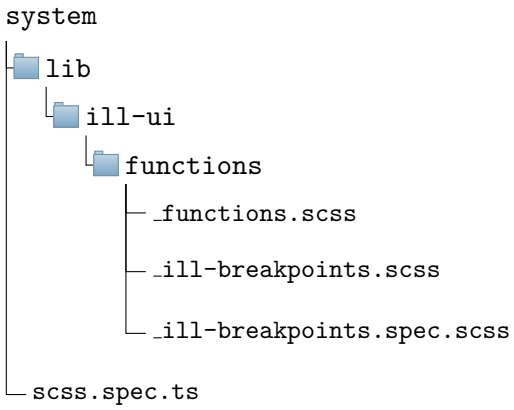
```
const path = require('path')
const sassTrue = require('sass-true')
const glob = require('glob')

describe('Sass', () => {
  // Find all of the Sass files that end in `*.spec.scss` in any
     directory in this project.
  // I use path.resolve because True requires absolute paths to compile
     test files.
  const sassTestFiles = glob.sync(path.resolve(__dirname, '**/*.spec.
     scss'))

  // Run True on every file found with the describe and it methods
     provided
  sassTestFiles.forEach(file =>
    sassTrue.runSass({ file }, describe, it)
  )
})
```

140.8 Run Sass True Directly, without using CLI

As of this time Sass True uses node-sass, which does not play well with the CLI. Therefore, we will need to run the sass spec runner directly. As a result, let's set up a specific npm script for running sass-true.

```
"sass-test": "jasmine libs/scss.spec.ts",
```

140.9 Wrapping it up - Reccomended Folder Structure



141 Spectator for Unit Testing

There are many quirks that surround angular unit testing. For instance, let's assume we have a very basic unit test:

```
import { TestBed, async, ComponentFixture} from `@angular/core/testing';
import { ButtonComponent } from `./button.component';
import { Component, DebugElement } from `@angular/core';
import { By } from `@angular/platform-browser';

describe('ButtonComponent', () => {
  let fixture: ComponentFixture<ButtonComponent>;
  let instance: ButtonComponent;
  let debugElement: DebugElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [
        ButtonComponent
      ]
    }).compileComponents();

    fixture = TestBed.createComponent(ButtonComponent);
    debugElement = fixture.debugElement;
    instance = fixture.componentInstance;
  });

  it('should set the class name according to the [className] input', ()
      => {
    instance.className = 'danger';
    fixture.detectChanges();
    const button = debugElement.query(By.css('button')).nativeElement as
        HTMLButtonElement;
    expect(button.classList.contains('danger')).toBeTruthy();
    expect(button.classList.contains('success')).toBeTruthy();
  });
});
```

This sample unit test has a quirk that is specific to Angular. Each time we have to go ahead and create a TestBed, as well as a fixture, and include the component.

141.1 Eliminating

If we were to use spectator, we can do the following:

```
import { ButtonComponent } from `./button.component';
import { Specator, createTestComponentFactory } from `@angular/core';

describe('ButtonComponent', () => {
  let spectator: Spectator<ButtonComponent;
  const createComponent = createTestComponentFactory(ButtonComponent);

  it('should set the class name according to the [className] input', ()
      => {
    spectator = createComponent;
    spectator.setInput('className', 'danger');
    expect(specator.query('button')).toHaveClass('danger');
    expect(specator.query('button')).not.toHaveClass('success');
  });
});
```

Using Spectator, we no longer have to create a Testbed and a fixture. Behind the scenes, spectator will handle all of these things behind the scenes. In addition, we are able to use the toHaveClass method, to check whether, or not a class exists.

141.2 Triggering Events

```
describe('Highlight Directive', () => {
  let host: SpectatorWithHost<HighlightDirective>;

  const createHost = createHostComponentFactory(HighlightDirective);

  it('should change the background color', () => {
    host = createHost(`<div highlight>Testing Hightlight Directive</div
        >`);

    host.dispatchMouseEvent(host.element, 'mouseover');

    expect(host.element).toHaveStyle({
      backgroundColor: '#fafafa'
    });

    host.dispatchMoustEvent(host.element, 'mouseout');

    expect(host.element).toHaveStyle({
      backgroundColor: '#ffffff'
    });
  });
});
```

Spectator greatly simplifies how we handle event handling within Angular.

141.3 Testing Services

Spectator greatly simplifies the way we can go ahead and test our services as well. Instead of having to create a TestBed that includes the service, we can simply do the following:

```
describe("AuthService", () => {
  const spectator = createService({
    service: AuthService,
    mocks: [DateService]
  });

  it("should not be logged in", () => {
    let dateService = spectator.get<DateService>(DateService);
    dateService.isExpired.and.returnValue(true);
    expect(spectator.service.isLoggedIn()).toBeFalsy();
  });
});
```

Using spectator, we are instead able to able to simply aplly mocks, the mocks key. We can then reference the mock service, and return a value, as Spectator treats all values as if they are spies.

141.4 Testing Components With Custom Host

Spectator also has the ability of greatly decreasing code needed to create a host element

```
@Component({ selector: 'custom-host', template: '' })
class CustomHostComponent {
  title = 'Custom HostComponent';
}

describe('With Custom Host Component', function () {
  let host: SpectatorWithHost<ZippyComponent, CustomHostComponent>;

  const createHost = createHostComponentFactory({
    component: ZippyComponent,
    host: CustomHostComponent
  });

  it('should display the host component title', () => {
    host = createHost(`<zippy [title]="title"></zippy>`);

    expect(host.query('.zippy__title')).toHaveText('Custom HostComponent
        ');
  });
});
```

By using Spectator, we have the ability to create a custom host component on the fly. Without using spectator, this process is very difficult to do so.

142 Unit Testing

Unit testing has always been a hot topic. It's because it really isn't so well understood amongst many software engineers. In addition, I know of many managers who look as unit testing as icing on the cake. Good unit testing, is incredibly difficult, and requires incredible amounts of due dilligence. Many books have been written in this regard, and will not go into detail here.

However, what unit testing allows us to do, is assure the code we are implementing is correct. Just as important, it sets testing in place, so that if another developer changes something, such as the ordering of function parameters, a class name, or a result of a function it will break. Unit testing can also be set in place to make sure a certain function is called, when a button is clicked, or in what order functions are correct. In can be set in place as automatic self correcting code.

Unit testing, can also make our code self documenting. In addition, it can give us the confidence moving forward knowing that this will work the way it needs.

Unit testing really is a beast, and is really hard to manage. It's because it isn't directly related to quality assurance, and can easily go under the radar, as the app will go on without it. However, everytime that I have communicated with management the bottom line. Research has been done on unit testing, and that the application will be harder to manage long term, without it, they understand. Bring the above point up to them, and hopefully you can have unit testing as a part of your application.

At this point in our application, the only thing that we have created that deserves to be unit tested, are the reducers. Let's go ahead and unit test those:

```
1  describe('Functionality for the ChooseSizeUpdated reducer', () => {
2    const chooseSizeData = {
3      columns: 20,
4      rows: 20,
5      pixelSize: 20
6    }
7    it('should update the chooseSize store as is approprate', () => {
8      const action: ChooseSizeUpdated = new ChooseSizeUpdated(
           chooseSizeData);
9      const actual = chooseSizeReducer(initialState, action);
10     expect(actual).toEqual(chooseSizeData);
11   });
12 });
```

The above is an example of what a sample unit test for a reducer would look like in the chooseSizeUpdated reducer.

Note: Add in chapter going over best conventions with regards to unit testing.

142.1 Unit Testing as a Discipline

Unit testing is difficult, because it is a different discipline. In particular, the above unit testing that we made, is a great example of poor unit testing. We simply tested to make sure that all fields properly made their way over.

However, there are a number of considerations to keep in mind:

1. What happens if we insert a string, instead of a number?

2. Is there any limit on the number of rows, or columns?

3. Should there be a limit on pixel size?

4. Should there be a 1:1 ratio between rows and columns, or vice versa?

142.2 The Irony of a Product Engineer

One might think that the above requirments, are to be pro-offered by the Product and to be tested by QA. However, at the end of the day, if these issues exist, it will mean that the software you created will be lackluster. [This indeed extends to other parts of the app, however, we are currently focusing on unit testing]. Therefore, try to take ownership as an engineer. There are different levels of intelligence, but amongst many other things, software engineers are a truly analytical bunch.

143 Understanding Different Types of Unit Tests

Just to re-iterate, the purpose of this book is to be a definitive guide.

As this is a chapter on unit tests, let's go ahead and dissect the various unique types of unit tests within our application.

144 Jest

144.1 A Primer.

Jest is a test runner created by Facebook to allow for the "Delightful Javascript Testing".

144.2 The Benefits of Jest Vs. Karma

1. Fast and sandboxed
2. Built-in code coverage reports

144.2.1 Fast and sandboxed

Jest parallelizes test runs across workers to maximize performance. Console messages are buffered and printed together with test results. Sandboxed test files and automatic global state resets for every test so no two tests conflict with each other.

144.2.2 Built-in code coverage reports

One has the ability to create code coverage reports using –coverage. No additional setup, or libraries are needed. Jest can collect code coverage information from entire projects, including untested files. The way it moves focus away from something like istanbul is fantastic. The UI, in my humble opinion, for what it's worth, is not the greatest.

144.2.3 Does not require starting a Browser

This one in particular helps with regards to CI/CD. Not requiring a browser, does not require a browser to be built into the CI/CD. In addition, a large part of performance issues with regards to unit tests, is having to start the browser every time.

144.3 Using Jest within an Nx Setting

```
ng generate jest
```


```
ng generate jest
```


After running the above generator, one can now run jest within your app. When generating a lib, one can now do:

```
ng generate lib libname --unit-test-runner jest
```

[1]

144.4 Primer on Jest Syntax

Real quick, I would like to go through a couple of things that Jest offers over Karma.

144.5 Switching over from Karma to Jest

At this point, this is the main selling point, of why Netanel Basel's library, Spectator, is so valuable, is that it allows for tests to be converted over to Jest automatically, by simply switching the imports. For instance, let's say that we are unit testing a service:

```
import { createService } from '@netbasal/spectator';
import { AuthService } from './auth.service';
import { DateService } from './date.service';

const spectator = createService({
  service: AuthService,
  mocks: [DateService]
});

it('should be logged in', () => {
  const dateService = spectator.get<DateService>(DateService);
  dateService.isExpired.mockReturnValue(false);
  expect(spectator.service.isLoggedIn()).toBeTruthy();
});
```

[1] If you are coming from an existing nrwl workspace, please use the following blog post to find out how to upgrade to Jest https://blog.nrwl.io/nrwl-nx-6-3-faster-testing-with-jest-20a8ddb5064

The above test is currently using karma. However, if we wanted to switch it over to use Jest, all we would need to do is change the import path:

```
- import { createService } from '@netbasal/spectator';
+ import { createService } from '@netbasal/spectator/jest';
```

Just like that magic, we can have our entire app using Jest.

145 Visual Unit Tests with Cypress

Unit testing is all great until something changes. The current issue with writing traditional unit tests is that you miss out on the point of front end developmentâĂŁ-âĂŁto create experiences for the end user. While unit tests may do very well in determining if an input produces an expected output, it only captures moment in time rather than a complete flow of actual events. This is where Cypress excels, especially in the end-to-end space.

e2e is easier to validate a user's experience and their interaction with an application. This makes it a good supplement to existing unit tests. While larger enterprises may have their own QA teams, e2e tests at the developer level makes coding driven by design rather than an additional checkpoint at the end.

Performance wise, it is easy to implement and has the potential to run tests in parallelâĂŁ-âĂŁmeaning, that multiple scenarios and outcomes can be tested simultaneously with auto reloading and snapshots for developers to debug if such an event should arise.

Cypress has solid documentation with event driven language for their syntax, making the task of writing tests more akin to actual user flows and potential interactions than just testing x and y. The visual nature of Cypress differentiates it from other unit testing and e2e suites currently available, giving the developer the ability to visually check the user experience with DOM state snapshots and historical contexts against different executions.

This makes Cypress a powerfully easy tool to use and lowers the bar of entry significantly for developers wanting to drive their development efforts with tests, catching bugs before they morph into something too big.

145.1 How to use Cypress with Nx

Before we proceed, Nx stands for Nrwl Extensions for Angular and is built by a team of ex-Google employees who were also part of the Angular core team. Nx isn't a replacement of the Angular CLI and instead extends it.

Nx sits on top of the CLI and works to give your application access to features and functionality currently not available in the CLI. It has the ability to create work spaces, along with applications and librariesâĂŁ-âĂŁthus expanding the CLI's capabilities.

So how do we use Nx with Cypress? and why should we?

Using Cypress with Angular CLI is possible but there is a lot of manual set up required. Nrwl Nx solves this problem by creating streamlined experience that pre-configures all you need for Cypress to work.

To use, Nrwl Nx, you'll need to install it using the following command in your console:

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

After this has installed, you'll need to create a workspace using create-nx-workspace command:

```
create-nx-workspace example-app --e2e-test-runner=cypress
```

This will give you a series of questions before creating your workspace. If you select Angular in the question that asks 'what to create in the new workspace', it will begin to set everything up for you. Nx will create a folder called example-platform and inside this folder, there will be an apps folder.

In this apps folder, you will find an empty Angular project and a Cypress ready e2e test unit.

Nx also supports React, so React developers are not left out from this out of the box set up.

For Angular, to run the testing suit, be sure to navigate into the workspace folder and use the command below:

```
ng e2e --watch
```

The –watch flag lets you write you tests in the background and Cypress automatically detects any changes in the test or application code and runs itself against the changes.

145.2 Example usage and cases

145.2.1 Accessing store

Cypress is able to test a multitude of front end frameworks and libraries, Redux being one of them. By using if (window.Cypress)window.store = store inside your src/index.js, you

are exposing the store when the application is run inside Cypress' browser. Now inside your test file, you can test the store's state through cy.window(). Here's an example of accessing a list store and testing if it has a specific item.

```
it('has expected state on load', () => {
   cy.visit('/')
   cy.window()
     .its('store')
     .invoke('getState')
     .should(
       'deep.equal',
       { list: [ { text: 'by apples', } ] }
      )
})
```

145.2.2 Button Clicked

Attaching a data-cy attribute to a button exposes the item for testing by Cypress. While this is not necessary as it can be accessed through classes, it is not recommended as it can result in the wrong object being targeted. data-cy therefore makes certain for Cypress that it is testing the right thing.

For example, your HTML button code may look something like this:

```
<button id="main" class="btn" data-cy="submit">
    Submit Me!
</button>
```

To create a test to check if the button is clicked, your Cypress test code may look something like this:

```
cy.get('[data-cy=submit]').click()
// OR
cy.contains('Submit Me!').click()
```

145.2.3 modal should appear when button is clicked

We all get given business and design rules to help us build our applications. It may read something like this: Modal should appear when button clicked, and should have drop down data, but should not have field available, if x data is not available.

To translate this into a Cypress test unit, your code may look something like this:

```
cy.get('[data-cy=submit]').click()
  cy.window()
    .its('store')
    .invoke('getState')
    .should(
      'deep.equal',
      { list: [ { text: 'by apples', } ] }
    )
cy.get('[data-cy=user-modal-dropdown]').click().contain([{text: 'by
    apples']);
```

145.3 Potential issues when using Cypress

The concept of code coverage refers to the percentage of code that is covered by my automated tests. Purity of test types may get muddied and percentage of test coverage may reduce over time when teams begin to mix and math e2e with unit tests.

The coverage therefore gets thinned over time and may leave pockets untested as a result. Certain pipelines may only pass a project to the next stage if it passes a percentage of testsâĂŁâĂŁtests that may may not account for e2e results or are exclusively e2e.

Another potential issue is that test hooks have the potential to muddy up the final production code. The idea of hooks is to keep the main code state untouched and independent from actions that are trying to observe events, inputs, outputs and outcomes.

145.4 Extended Features

Cypress' test runner is MIT open sourced and free. However, it does have a commercial side to it and offers a Dashboard service allows you to centralize your tests and enable continuous integration at a team level. This differs from the test runner which only runs on your local machine.

There is a free Seed tier that currently allows for up to 3 users and 500 test recordings. As you move up through the plans, the level of support, users and number of test recordings increases.

145.5 Final words

Cypress as a testing tool differs from all other testing tools, frameworks and assertion libraries because it offers front end developers a way to test their applications that is driven

by user experience.

Creating unit tests can be a dry process but Cypress brings the fun and beauty of creating visuals without having to manually click the buttons yourself every time to test if something worked. The DOM snapshots makes for fantastic debugging and the concise, yet information rich, documentation also helps in the learning process.

Setting up is quick and it doesn't take long at all to get started on Cypress.

146 Testing Cypress Locally With Authentication

In any real world application, there is going to be the need to add authentication to your application. In real world application, this translates to a login process, to verify you are, who you say you are.

The difficulty with this, is that all enterprise organizations re-direct the user to an alternate link to log in. After user log's in with username and password, this individual will then be re-directed to appropriate app. This re-direct will trigger a CORS issue. The error usually looks something like the following:

```
CypressError: Cypress detected a cross origin error happened on page load:

Blocked a frame with origin "url" from accessing a cross-origin frame.
```

If you want to test locally, and see your updates in real time, you will need to get around this CORS issue.

146.1 Steps to Solve Cors Issue

1. Add the following to your `plugins/index.js` file

```
module.exports = (on, config) => {
  on('before:browser:launch', (browser = {}, args) => {
    console.log(config, browser, args);
    if (browser.name === 'chrome') {
      args.push("--disable-features=
          CrossSiteDocumentBlockingIfIsolating,
          CrossSiteDocumentBlockingAlways,IsolateOrigins,site-per-
          process");
      args.push("--load-extension=cypress/extensions/Ignore-X-Frame-
          headers_v1.1");
    }
    return args;
  });
};
```

2. Add the following to your `cypress.json` file `chromeWebSecurity: false`

3. Change the appropriate environment url to use localhost (might be different in your development environment) in the cypress.json file. `test_url: "http://localhost:4200",`

4. Download the Ignore X-Frame Headers plugin within Cypress launched browser. Will remain downloaded, even when you end your Cypress session.

146.2 Friendly Reminder

These four steps are all you will need in order to run Cypress locally. Ideally your organization should have special urls to run your apis in a test environment. If that is the case, do not forget to change them in your environment.ts file.

147 Unit Testing State

Testing state potentially can be a grueling process. You will have to create a module for state. You will then have to inject appropriate data that you expect from the store. Within our reccomended Facade architecture, this problem is already alleviated. Why? Because, being that the store is located within our facade, we can simply provide it with an empty object. When we end up using the facade within our components, the injected store never even makes it's way through, because it is mocked out. Therefore, it never becomes more complicated than this.

```
describe('UserFacade', () => {
  let facade: UserFacade;
  let store: Store<any>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [StoreModule.forRoot({})],
      providers: [
        UserFacade,
      ],
    });

    facade = TestBed.get(UserFacade);

    store = TestBed.get(Store);
    spyOn(store, 'dispatch');
  });
```

For this reason, and many others, using the facade pattern within Angular, just makes so much sense. Feel free to check out the chapter on facades, for the full rundown on why this pattern just makes so much sense within an Angular setting.

148 Unit Testing Architecture

In any given UI application, the following should be considered as appropriate architecture.

1. Testing structure
2. Assertion functions
3. Generate, display and watch test results
4. Generate and compare snapshots of component and data structures to make sure changes from previous runs are intended
5. Provide mocks, spies, and stubs
6. Generate code coverage reports
7. Provide a browser or browser-like environment with a control on their scenarios execution

In the previous chapter we discussed using Jest for the large part of our application. Jest should ideally be used across the app. The truth is that Jest was built for a mono-repo. In particular, when it comes to running parallel unit tests. In addition, the way the view shows only the test of the component that has been updated. Lastly, the last thing that I notice, is it's use of snapshot testing.

149 Interfaces and Unit Testing

In unit testing it can be very difficult to keep in sync the mocked data you are using, with actual data used within app's actual live UI. The easiest, and most efficient way of doing this is creating interfaces. The part where it becomes tricky, is that generally data is used in multiple places. For instance, we might have state that is contained in a separate component, wherein the data originates from can be somewhere else. In addition, the data might also be used in some other component as well as in some other service.

149.1 In Sync Data - Interface Architecture

An interface at it's core is responsible for making sure that data follows a pre-described schema. The part, however, that is unintuitive in an Angular setting, is what happens when you have services, components, state, and spec files all vying for the same data. Do we use one interface for all of them, or different one's for each file group? If we do end up using one interface, what sort of data structure is it that we will use for all of these files.

149.1.1 Interface Architecture - The Dillema

As we discussed in the previous paragraph, in an Angular application, there will be services, components, state and spec files vying for the same data. The dillema when it comes to interfaces, however, is that they need data in different ways. I would like to layout this data in detail.

Let's imagine that we have a data-table that we need to specify data:

- Service - Used to determine status of checkbox logic. It only needs to know length of data, and actual data is irrelevant. Respective spec, only needs to be aware of similar data.

- Component - Needs to know actual data, so that it can pass along observable stream component html. Respective spec needs to be aware of data.

- State - Depnding on the reducer, or effect, it might need all of the data, or none of it. Respective spec will need to familiar of similar data.

149.2 In Sync Data - The Solution

As one can see, it is actually counter-intuitive to create a singular interface for one's service, state, component(s), and their respective specs. However, if one does not use the same singular spec for all of them, one runs the risk of them getting out of sync with each other. The data inside of the object might make all the difference when it coems to causing unit tests to fail.

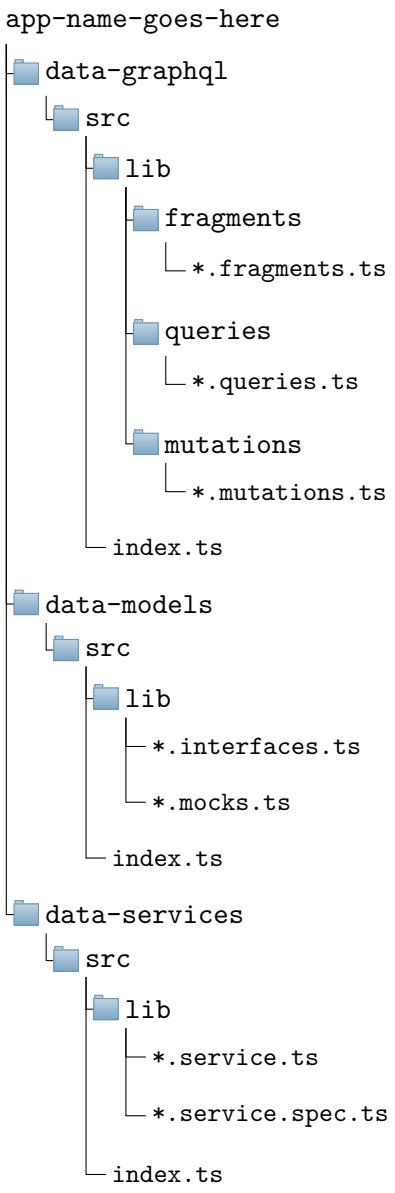
The solution is as follows. There should be a singular interface that exists for the root of the following files:

1. Services[Includes spec files]
2. State[includes spec files]
3. GraphQL(Interfaces not used, but influenced by, and therefore important to be in the same directory)
4. Component(Not in same folder as above)

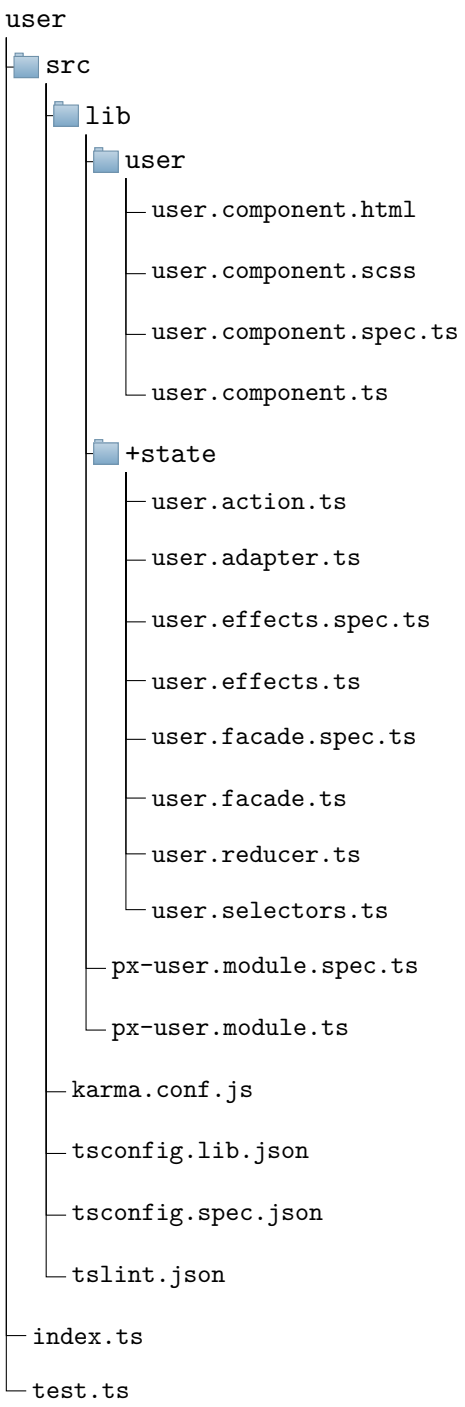
This, however, requires that all the files be tightly coupled together. In order to do this, we must create a well thought out folder/file structure. This is what we will be calling Data Access architecture.

149.3 Data Access - Folder Structure

Let's say we are creating an grid-form component, that will make use of backend data. The data structure will look as follows:



The expectation is that there will also be a sister folder/file structure for these three in particular(data-graphql, data-models, data-services). In the appropriate feature, it will have a state feature, which will have the same name as the respective data-access architecture.



Using this folder/file structure, we can enforce there being a single interface used across all files within our app that are using data. The idea is that the nomenclature will remain the same across

149.4 Example of How Interface Might Be Used in Real Time

Now that we have our folder/file structure in place, we can prove how having a singular interface can be used to make sure all of our files are in sync.

```
export interface GridForm {
  column: string;
  row: string;
  pixelSize: string;
}
```

149.5 Example of How Data Mock Will Look Like

In addition to our interface, we will have mocked data that will be used in a number of different instances. It makes sense to have data in a mock.ts file so that it can be re-used throughout app, without having to re-create.

```
cost GridForm = {
  column: '20';
  row: '20';
  pixelSize: '20';
}
```

149.5.1 Service for Pulling in Pre-Populated Grid Form

In our service, we will use the interface to determine what sort of data we expect to be pulled in.

```
getGridForm(projectId: string): Observable<GridForm> {
  const query = GridFormQuery;
  const variables = {
    projectId
  };

  const form$ = this.apollo.query<GridForm>({ query, variables });

  return from(buyers$).pipe(pluck('data'));
}
```

149.5.2 Service Spec for Pulling in Pre-Populated Grid Form

In the spec for our service, we will have one consistent piece of data that will be used throughout the service spec:

```
const gridForm: GridForm = {
  column: "20";
  row: "20";
  pixelSize: "20";
};
```

Here we are using the interface to make sure that the mocked data used with the service stays true to the data that is expected to be returned. If the data ever changes in actual app, the interface will be there to make sure our specs data mocks are up to data as well.

149.5.3 Reducer for populating state with appropriate Grid

For simplicity sake within our app, we are going to take the data as is, and pass it directly into our store to be used within app:

```
export function gridFormReducer(
  state: GridForm,
  action: BuyerAction
): GridForm {
  switch (action.type) {
    case BuyerTypes.FormLoaded: {
      return {
        ...state
      };
    }
  }
}
```

Here we have the interface telling our app that all data within our reducer must consist of the three items we have specified in our GridForm interface.

149.5.4 Reducer Spec for populating state with appropriate Grid

```
  describe('gridFormLoaded action', () => {
    it('should populate the buyer entities and ids', () => {
      const action = new FormLoaded(gridForm);
      const state = gridFormReducer(initialState, action);

      expect(state).toEqual(gridForm);
    });
  });
```

Here we have a very simply unit test that once again has the same data that is used across the app. By having an interface and using it for the data within the interface we can make sure that the data for the reducer spec is up to date with actual app.

149.5.5 Effect for populating state with appropriate Grid

```
@Effect()
loadGridForm$ = this.dataPersistence.fetch(BuyerTypes.LoadGridForm, {
  run: (action: LoadGridForm, state: GridForm) => {
    const projectId = this.projectFacade.getProjectIdFromState(state);

    return this.service
      .getGridForm(action.payload, projectId)
      .pipe(map((gridForm: GridForm) => new GridFormLoaded(gridForm)));
  },

  onError: (action: LoadGridForm, error) => {
    console.error('Error', error);
  },
});
```

In our effect, using the facade pattern [1]. In our param for state, the most data heavy part of our effect, we are once again using the same interface.

149.5.6 Effect Spec for populating state with appropriate Grid

```
const projectId = '123';

describe('loadBuyers$', () => {
  beforeEach(() => {
    spyOn(service, 'getGridForm').and.returnValue(of(gridForm));
  });

  it('should work', () => {};
    const action = new LoadBuyers();
    const completion = new BuyersLoaded(gridForm);

    actions$ = hot('-a', { a: action });
    const expected$ = cold('-c', { c: completion });

    expect(effects.loadGridForm$).toBeObservable(expected$);
    expect(service.getGridForm).toHaveBeenCalledWith(projectId);
  });
});
```

Here we are once again attaching an interface for gridForm, our data, so that it is consistent across our application.

[1] it is calling the effect, pulling in data, and then having the appropriate action fired off, for saying data is loaded

150 Mocking Data

An interesting phenomenon that happens within many enterprises, is with regards to unit tests and maintanability. Many times the data mocks are repititions of themselves. In addition, utilities with regards to mocking data. I've done this myself, maybe you have as well. This chapter is simply there to put this in as convention, to go ahead and edit moving forward.

150.1 Function Composition

With regards to creating re-usable data, there are two options. One can use a function to generate a mock. Alternatively, one can use a spread operator. For the most part, I prefer a function to generate a mock. It gives the option to specify parameters to be passed through. In addition, it enforces immutability. However, I have worked on teams, where functions seam out of place.

```
// this interface is in our users.interface.ts file
export interface User {
  id: string;
  name: string;
  location: string;
}
// this interface is in our users.interface.ts file
export interface Users {
  users: User[]
}

let generateMockUserData: Dictionary<User> = (user: User) -> {
  data[id]: {
    data
  }
}

let generateMockUsersData: Users = (data: Dictionary<User>) -> {
  users: {
    ...data
  }
}
```

By using the simple above function, and proper type annotation we have the option to generate mocked data that is unique to our interface. In addition, we have the ability to

pass in data as it is unique to our specific use case.

150.2 Core Constants

In addition to having function composition tied to our interfaces, odds are that the data being passed in might have the same signature time and time again.

Enter the three constant rule:

```
const genericMockUsers: Users = {
  generateMockUsersData(
    generateMockUserData({
      id: '123',
      name: 'Charles',
      location: 'New York'
    }),
    generateMockUserData({
      id: '246',
      name: 'Lisa',
      location: 'London'
    }),
    generateMockUserData(
      {
        id: '468',
        name: 'Yoda',
        location: 'Canary Islands'
      }
    ),
  )
}

const genericMockUser: User = {
  generateMockUserData({
    id: '123',
    name: 'Charles',
    location: 'New York'
  }),
}
```

Moving forward, if any of this data has a unique signature, then you can use the above functions to make sure you have the correct data.

151 Spies

Spy's are an integral part of any unit testing suite. For those familiar, you might think of it as something that doesn't neccesarily deserve it's own chapter. However, spies help simplify the unit test suite to such a great extent, that not discussing the finer detais of how it should work, almost seems like a crime.

151.1 A Primer

First and foremost, spies are actually one of the finer points of unit testing. Personally, when I started writing them professionally for Verizon, spies were one of the more difficult things for me to understand. I found the name particularly confusing. A spy?! You mean I am spying on a function that I already know about? Wait, so you mean I am giving my unit tests the ability to tell if function has run, or what it has been called with? Also, it gives me the ability to hijack it with a different function, or call through as is? Wow, I can't think of any name to call that? Ok, now I understand why it was called a spy.

151.1.1 Spy User Example

```
describe('', () => {
  const userId = '123';
  it('should call the userFacade.getUsers function, when getusers' +
  'is called', () => {
    spyOn(userFacade, 'getUsers');
    component.getUsers(userId);
    expect(userFacade.getUsers).toHaveBeenCalledWith(userId);
  });
});
```

The above is a classic example of what a real world spy would do. Here, we just want to just make sure that the facade is indeed being called when the appropriate component function is run.

In a real world scenario, it would also be responsible of us to test, and make sure that the component function was indeed called. There are two ways that something like this can be done. One is that we can targer the dom within our unit test. We can also choose to make this part of the unit test within Cypress. I like the idea of seperating any integration testing into Cypress. It would allow us to keep our component unit test shallow, without the need of having any component integration involved.

151.2 Two Methods of Declaring Spies

It would seem that there are two schools of thought when it comes to how one should place the spy. Some place the spy within the actual beforeEach statement. This simplifies the use of doing so. The other approach, is to apply the spy inside of every it block that has use of it. While I do not think it is a huge deal to have one approach over, the other, I do see the benefit of placing the spy in every it block that uses it. Why? Generally, in an enterprise Angular application, there will be a very small amount of spy's that will be set out throughout the application. So, adopting a always put a spy in a beforeEach, can be:

1. Irresponsible for performance reasons (from a code perspective. I would still gladly call you my friend).

2. Totilitarian in nature of adoption. However, due to the fact that not all tests require spies, it can be confusing.

So, it would make sense that putting spies in individual it blocks will make the most sense.

151.3 Strategy for Using Spies

Spies work particularly well with function composition. The attempt of this book is not to overlap with common unit testing principles, such as functions should be kept small. It is to introduce to those that might be less familiar with how to use spies, how it should be used. In particular, let's imagine we had a large function. This function applies sorting by alphebetical order, converts the text to camel case, and then converts it into a dictionary. From an abstract level, we can apply each of these into their own function.

```
1  alphabetize(users: users[]) {
2    return sort(users);
3  }
4
5  camelCase(userName: string) {
6    return camelCase(userName);
7  }
8
9  convertToDictionary(userId, user) {
10   return {
11     [userId]: user
12   }
13 }
14
15 convertUser(users: users[]) {
```

```
  const users = this.alphabetize(users);
  users.map((user: user[]) => {
    convertToDictionary(camelCase(user));
  })
}
```

This chapter, needs work, and it is something for use to work on

We can then run a unit test for each of the above functions, and then run a spy on them, for our global function. It would look something like the following:

```
describe('alphabetize', () => {
  it('should alphabetize users', () => {
    // appropriate unit test goes here
  });
});

describe('camelCase', () => {
  it('should convert userNames to camelCase', () => {
    // appropriate unit test goes here
  });
});

describe('convertToDictionary', () => {
  it('should convert users to dictionary', () => {
   // appropriate unit test goes here
  });
});

describe('convertUser', () => {
  const users = [...usersMock];
  it('should convertUser to appropriate name', () => {
    spyOn(component, 'alphabetize');
    spyOn(component, 'camelCase');
    spyOn(component, 'convertToDictionary');
    component.converUsers(users);
    expect(component.alphabetize).toHaveBeenCalledWith(users);
    expect(component.camelCase).toHaveBeenCalled();
    expect(component.convertToDictionary).toHaveBeenCalled();
  });
});
```

Here we can see that all of our unit tests are tested individually. When it comes to composing them all into a giant function, from a unit testing perspective, all we need to do is make sure that they were indeed called with the correct parameters.

151.4 A Final Note

Spies are a relatively simple concept. However, there are finer points that make spies a little bit of a confusing topic to understand. This chapter is an attempt to make that a bit easier.

152 Debugging

Debugging in any development environment is a neccesity. When it comes to unit testing, sometimes a unit test will not pass, and will be a bit difficult to decipher why it is that it isn't passing. I would like to present the following strategy to greatly expedite the process of debugging, when you find yourself going against a unit test that is taking longer than expected.

152.1 A winning Combo

There are two pieces to debugging Unit Tests, that will allow you to debug unit tests with ease. One of them, is to create smaller modules whenever possible. This will allow you to unit test specific modules using:

```
ng test --project user --watch
```

Now you will have a window that is open, with the ability to debug, that only runs unit tests for the specific module.

152.2 Opening Source

The simplest solution to start debugging, would be to open up the inspector. Click on source, add a debugger for the particular function that you are looking for, and re-run the page. This will cause the page to then pause on the function you are trying to test.

152.3 Turning Off Source Map

The specifics with regards to how Angular works, it can be advantageous to turn off the source map. The following is the syntax on how to turn off the source map for angular:

Output inside regular karma browser window will now give enough information, and the window will give enough information. Zone.js, Karma, and unit tests, the special reason as to why something might not work, might be obfusicated by zone. To alleviate this, we have the ability to turn souce map off.

```
ng test --project=px-illustrator --source-map=false
```

Source map might seem like a bad idea, but by keeping if off in general, while developing, it would allow us to make sure we see all the errors that we need to see.

153 Coverage Reporting

Coverage reporting for an Angular application using the Angular CLI is objectively easy to implement in this day of age. Albeit easy to generate coverage reporting, it's interesting, because as a software engineer, who develops exclusively using TDD/BDD, coverage reporting becomes of less importance. I find myself spontaneously checking up on coverage reporting from time to time (usually every week), to make sure the team is hitting the numbers we want.

153.1 Testivus On Test Coverage

There is a funny, yet honestly, largely impactful founding post by a one Alberto Savioa[1]. It is entitled, "Testivus On Test Coverage". I would like to post the entire entry here, because it is pervasive in unit test coverage:

> Early one morning, a programmer asked the great master:
> âĂIJI am ready to write some unit tests. What code coverage should I aim for?âĂİ The great master replied:
> âĂIJDonâĂŹt worry about coverage, just write some good tests.âĂİ The programmer smiled, bowed, and left.
> ...
> Later that day, a second programmer asked the same question.
> The great master pointed at a pot of boiling water and said:
> âĂIJHow many grains of rice should put in that pot?âĂİ The programmer, looking puzzled, replied:
> âĂIJHow can I possibly tell you? It depends on how many people you need to feed, how hungry they are, what other food you are serving, how much rice you have available, and so on.âĂİ âĂIJExactly,âĂİ said the great master.
> The second programmer smiled, bowed, and left.
> ...
> Toward the end of the day, a third programmer came and asked the same question about code coverage.
> âĂIJEighty percent and no less!âĂİ Replied the master in a stern voice, pounding his fist on the table. The third programmer smiled, bowed, and left.

[1] https://www.artima.com/weblogs/viewpost.jsp?thread=204677

...
After this last reply, a young apprentice approached the great master:
âĂIJGreat master, today I overheard you answer the same question about code coverage with three different answers. Why?âĂİ The great master stood up from his chair:
âĂIJCome get some fresh tea with me and letâĂŹs talk about it.âĂİ After they filled their cups with smoking hot green tea, the great master began to answer:
âĂIJThe first programmer is new and just getting started with testing. Right now he has a lot of code and no tests. He has a long way to go; focusing on code coverage at this time would be depressing and quite useless. HeâĂŹs better off just getting used to writing and running some tests. He can worry about coverage later.âĂİ
âĂIJThe second programmer, on the other hand, is quite experience both at programming and testing. When I replied by asking her how many grains of rice I should put in a pot, I helped her realize that the amount of testing necessary depends on a number of factors, and she knows those factors better than I do âĂŞ itâĂŹs her code after all. There is no single, simple, answer, and sheâĂŹs smart enough to handle the truth and work with that.âĂİ
âĂIJI see,âĂİ said the young apprentice, âĂIJbut if there is no single simple answer, then why did you answer the third programmer âĂŸEighty percent and no lessâĂŹ?âĂİ
The great master laughed so hard and loud that his belly, evidence that he drank more than just green tea, flopped up and down.
âĂIJThe third programmer wants only simple answers âĂŞ even when there are no simple answers âĂę and then does not follow them anyway.âĂİ The young apprentice and the grizzled great master finished drinking their tea in contemplative silence.

153.2 The Impact of Testivus

Testivus's impact on unit testing cannot be overrated. At most organizations that I have worked at, 80% is indeed the golden number. The question is, what is the validity of the number 80%? First, and foremost, I think that culturally 80% is intuitively considered above average. Particularly, because 70% from an academic perspective is considered as passing by many.

However, I would also like to assess the number objectively. 80% seems to coincide with the Pareto principle. The Alfred Pereto principle, if not already familiar, claims that 80 percent of consequences, come from 20 percent of the causes. In a super interesting article, Microsoft learned that, "...that 80 percent of the errors and crashes in Windows and Office are caused by 20 percent of the entire pool of bugs detected".

So, you might ask why don't we keep the rule of 20% to the most important features. Well this might just be an impossible task. For a software engineer to be aware of potential bugs before they happen, is once again, impossible. Instead what we should do, is focus on 80%. This means that we have a 4/5 chance of catching a 20% bug.

153.3 The phenomenon of the Pareto Rule

It is also important to realize, that within the Pareto rule, there is a 20% within the 80%. That is, the difficulty of completing a meaningful test(i.e. one that solves 80% of uses cases)increases.

The interesting thing about the Pareto rule, is that one can make the argument, that the 20% features that make an 80% impact, are those that are hardest to test. So arguably, those tend to overlooked. However, similar to what Testivus said, if we want a blanket rule, let's apply the Pareto Priority Index(PPI): [2]

$$PPI = \frac{\text{savings} \times \text{probability of success}}{\text{cost} \times \text{time of completion}}$$

The Pareto Priority Index, is really just fancy way of saying, if you find this unit test is taking you too long, and it's not worth it, granted the unknown of what this unit test is, move on. So this number(80%) is debatable, but given the above, 80% seems like a very respectable amount. I personally keep it at 80, and even the Angular docs reccomend 80.

153.4 Different Types of Coverage Reporting

The following are the four different types of coverage reporting:

1. Statements - Has each statement in the program been executed?
2. Branches - Has each branch of each control structure been executed? For instance, have all case statement, or if/else statements been called?
3. Functions - Has each function (or subroutine) in the program been called?
4. Lines - has each executable line in the source file been executed?

153.4.1 Distinguish Between Statements + Branches

I generally find it difficult to distinguish between a statement, and a branch. The following is a great example. Let's say we have the following scenario:

```
1 determineStatement(a: any, b: any): boolean {
2   if(a){
```

[2] https://www.sixsigmadaily.com/the-six-sigma-approach-to-project-selection/

```
3       if(b){
4         statement2 = true;
5       }
6       statement1 = true
7     }
8   }
```

If we test a = true, and that b = true, that will give us 100% statement coverage, as we have tested all statements. However, we have not tested the hidden else statements. Therefore while the statement coverage within our function is 100%, the branch coverage is only at 50%.

153.4.2 Understanding Letters on Side of Code Coverage

1. 'E' stand for 'else path not taken'. Meaning that if path has been test, but not the else path.

2. 'I' stand for 'if path not taken'.

3. 'xN' in left column, is the amount of times the line has been executed.

153.4.3 Understanding Colors

1. Orange: Functions not covered.

2. Pink: Statements not covered.

3. Red: Non executed lines, or pieces of code.

4. Yellow: Branches not covered.

153.5 Code Coverage Enforcement

In line with our decided 80% coverage, we can go into our karma.conf.js file and add the following in the coverageIstanbulReport: key.

```
1 coverageIstanbulReporter: {
2   reports: [ 'html', 'lcovonly' ],
3   fixWebpackSourcePaths: true,
4   thresholds: {
5     statements: 80,
6     lines: 80,
```

```
    branches: 80,
    functions: 80
  }
}
```

Now we have made it that our unit testing will fail whenever we reach a threshold below 80%. This is useful for development, and useful for devops, as we can keep a consistent way of keeping our unit tests at a certain level.

153.6 Running a Coverage Report in Angular

My preferred method, is to add a package.json script for running a –code-coverage report. It would look something like the following:

```
{
  ...,
  test-coverage: ng test --code-coverage
}
```

and then in your terminal run:

```
npm run test test-coverage
```

Walla! Just like that, the CLI will take care of emitting code-coverage for you.

153.7 Opening up Coverage Reporting

In order to open up a coverage report, after a coverage report has been generated, simply navigate to your coverage/src folder, and click on the index.html file.

I've found that a script that's available to automatically open up the coverage reporting index.html has made me more prone to paying attention to coverage reporting. However, as time has gone on, and I realize the value of TDD, I think that coverage reporting is less relevant. At this point in time, I feel it is more relevant as a devops tool, and a way of making sure that unit testing stays at a certain caliber.

153.8 Trap of Coverage Reporting

In a perfect world, all functions would be pure, and they would solve one use case. However, that is frequently not the case. In addition, even if it was the case, a unit test should still be tested against numerous use cases. So Coverage reporting while on paper makes seems like it is 80% it might be 10% to 20% less than that.

154 Unit Testing the DOM

The DOM is the one variable with front end, that complicates the unit testing process. While it is in the process of becoming more stable, it can be difficult to unit test, and can make TDD more difficult. It is very much important to do and can also be expensive from a unit testing perspective. A unit test will run much slower if it has to interact with the DOM. So, this is something to take into consideration as well. [1]

DOM unit testing refers to a couple of scenarios:

1. Event Handling.

2. Element is visible, or hidden.

3. Element contains certain text, or that text properly transformed.

These three are the major types of DOM unit testing that occurs. There is, of course, much more to unit test from an E2E perspective. Determining whether, or not an element is visible, or hidden, and whether it contains text, is better handled by Cypress. For this please do reference the chapter on visual unit testing. The one piece, however, that is handled better by unit testing, is testing that a function did indeed run, when an element has been clicked. There might also be other unique situations, including the above, if your team does not have the capacity to use Cypress.

154.1 Selecting element

In an Angular setting, the most important of all testing utilities, is the Angular TestBed. The TestBed creates a dynamically-constructed Angular test module, that emulates an NgModule [2]. In laymans terms, it allows you to swap out any piece that was included in the component, for testing purposes, and then to reference that swapped out piece.

Using the testBed, we are also able to create a fixture, which we can reference to target the nativeElement. We can then use the querySelector on the nativeElement, to target our element. Let's go back to the one use case we would like to target. That is, when an event

[1] Look into chapter on Cypress for visual unit testing.
[2] https://angular.io/guide/testing#testing-services-with-the-testbed

is triggered, we would like to make sure a particular function is called. In each scenario, we would like to make sure that once a filter is clicked on, the appropriate filtering function is called.

```
it('should call the appopriate function when filterToggle element' +
`is clicked on', () => {
  spyOn(component, 'filterUsers')
  const filterToggle = fixture.nativeElement.querySelector(
    '.filter-toggle'
  );
  filterToggle.click();
  expect(component.filterUsers).toHaveBeenCalledWith(component.id);
});
```

First, we are spying on the filterUsers method for our component. Next we are using the querySelector to target the .filter-toggle html class (assuming there is only one on the page). Moving on, when clicked on again, we want to make sure that the appropriate function is called.

We have now completely through the power of unit testing, determined whether, or not an element is going to show up.

154.2 Unit Testing - Determining Text

With regards to text, let's say that we want to test the entire component at a specific time period, and want to make sure it contains three different words:

```
it('should show buyer company names', () => {
  expect(fixture.nativeElement.innerText).toContain('Apple');
  expect(fixture.nativeElement.innerText).toContain('Microsoft');
  expect(fixture.nativeElement.innerText).toContain('Google');
});
```

Text might seem intuitive. However, there is the option to target text at different areas of time, and to make sure what one is looking is the correct format at a given time. Doing something like this takes experience to get it right. However, assuming you didn't know beforehand, you now know that you have the option to target text at a specific time.

As we mentioned, text and whether, or not an element is hidden, might be better handled by Cypress. However, just in case you want to see it for yourself, the above is a great example.

155 Unit Testing - Mocking Providers

Another important part of Angular Architecture, is that one has an option to import a service as is, or to mock it out. This will apply, as per our architecture to facades as well. Not doing so, will leave your unit tests at the mercy of your services, and might have you unit testing your service as well as your specific component. In addition, and perhaps, more dangerous, that if you are not careful, you might end having your data being retrieved from the back end while you are unit testing.

155.1 Re-iterating Previous Point

We have already discussed a previous point with regards to unit testing and interfaces. The point is that we can use a singular data mock to keep all of unit tests in sync. This is indeed a very important point that works in tandem with mocking providers, and we will touch on this in this chapter.

155.2 When to Mock Providers within App

The point of mocking service dependencies is in order to test the component in an isolated environment. This should includes pipes, services, and in our architecture especially facades. If it is a module, which is completely focuses UI, then there is no need to worry about an isolated environment, because there is no logic wherein to afffect the component.

155.3 Mocking Providers - Setting the Landscape

Just setting the landscape for what an example situation might be like with regards to mocking providers. In our Angular - The Full Gamut architecture, we have a facade which is always going to be responsible for bridging the data retrieved by our service, with our component. It is going to be service that will ultimately be used in our component to retrieve data.

```
1 gridForm$: Observable<GridForm[]> = this.store.pipe(select(getGridForm))
      ;
```

The above is an example snippet of our getGridForm.facade.ts file, that will be responsible for pulling data from our store, using the getGridForm selector already specified else where. If we were to pull in this facade as is, it would end up actually pulling data from the server while doing unit tests! That would be a cardinal sin. I

```
<form>
<div>{{ (gridForm | async).row }}</div>
<div>{{ (gridForm | async).column }}</div>
<div>{{ (gridForm | async).size }}</div>
</form
```

155.4 Mocking Providers Within Unit Test - A Primer

In our unit test in order to mock the above Observable gridForm$ stream, we can very simply use the data mock we have specified in our mocks.ts file.

```
providers: [
  {
    provide: GridFormFacade,
    useValue: {
      gridForm$: of(generateMockGridForm()),
    },
  },
],
```

In this very simple scenario, we have just made it so that the data returned within our component for our unit test, is using the central mock being used elsewhere in our data-access architecture. This gives us more control over what we want to do. We can make the data more complex, to test different use cases per it block that we want to use.

155.5 A Final Note

The architecture with providers, given the already written chapter on interfaces and mocking data, is really just a skip and a jump away. The reason I decided to write this chapter, is because while simple in practice, it tends to be overlooked. This chapter properly informs a software engineer when that situation might be appropriate and the proper way to do so.

156 Unit Testing Modules

I think this is an important chapter, because unit testing modules are a good little piece of architecture in Angular. Simply put, it allows you to put a number of providers into the module for unit testing. Then you can just go ahead and import the module. In fact, even if there is only provider being added to the module, it is still keeping your app DRY'er(AKA Don't Repeat Yourself).

156.1 Comparison of Using a Module Vs a Provider

I would just like to illustrate the point of how simple it is to use a module vs a provider. The reccomended use of a unit testing module, is for a facade. If you will recall, our use of a facade, for our data-access architecture. This means, that there is only one file that is required for us to provide in our unit tests. Nonetheless, let's say that we are attempting to mock our UserFacade that we use for user-settings. Our file would look like the following:

```
mport { NgModule } from '@angular/core';

import { UserFacade } from './+state/user.facade';
import { MockUserFacade } from './+state/user.facade.mock';

@NgModule({
  providers: [
    {
      provide: UserFacade,
      useClass: MockUserFacade,
    },
  ],
})
export class PxDataAccessUserTestingModule {}
```

Now, if we want to mock the data that we are using within our app,

```
describe('UserSettingsComponent', () => {
  let facade: UserFacade;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [PxDataAccessUserTestingModule],
    });

    facade = TestBed.get(UserFacade);
```

```
10    });
```

As components are wont to do, we will most likely be using the UserFacade in a number of different locations.

156.2 Pitfall of Using a Module

A module is a bit of a blackbox. We don't neccesarily know everything that is contained within the module. So for any developer other than the one who created it, it might be a bit confusing when looking at the file for the first time.

156.3 Moving past the Pitfall of using a Module

It is important to put in place a convention for all places wherein a module should be used. When running through our SMAG architecture, the only place that deserves a module, is for our data-access folder. Primarily, because our data-services are usually only going to be used within our data-access folder. So really a testing module only makes sense for our facade. So as a convention, your team should always use a testing module for facades within the context for your data-access folder.

157 Marble Unit Testing

Unit testing observables, heck working with obserables is never really easy. Primarily, it's not the logic one will come across in other parts of one's application. As opposed to regular logic, where one has the option to simply console out logic and see what is there, then and there, one does not have that option wih observables. Observables are a stream and in the context of javascript it is an object that will a series of functions. In addition, an Observable as a stream contains more than one snippet of data in it's entire lifecycle.

157.1 Marble Unit Testing - A Primer

Marble unit testing is a very efficient way of unit testing observables. It keeps in mind the following:

1. Recognizing that an observable stream can emit any number of values after a specific set of time.

2. An observable is always observing until actually complete

3. It can emit a number of different things at the same time, or at completely different times.

157.2 Great Example

Let's imagine that we get back a specific set of data as an observable. However, within that set of data, we only want ids. Backend is tied, and they do not have the capacity to give us a pre-poulated set of data for id. Our code will look something like this:

```
userIds$: Observable<string[]> = this.users$.pipe(
  map(users => {
    return users
      .map(user => {
        return user.id;
      })
  })
);
```

157.2.1 Creating a unit test

In our unit test, we would like to make sure that when we pass a set of data, id's are indeed being extracted and returning a new array. We can do something as follows using marble tests:

```
const usersMock: User[] = [
  {
    id: '123',
    name: 'Charlie',
  },
  {
    id: '246',
    name: 'Lisa',
  },
  {
    id: '369',
    name: 'Harley',
  },
];

it('should return buyer data for tier', () => {
  const expected$ = hot('(c|)', { c: ['123', '246', '369'] });

  expect(component.buyerTiers$).toBeObservable(expected$);
});
```

In this unit test, we expect our function to return.

```
['123', '246', '369']
```

As we are unit testing against an observable, we can potentially use subscribe, to emit the value of the component function we are testing. However, this can get a bit cumbersome, as it's a bit of code, and throwing subscribes into a unit test can go curious places. In our marble unit test, we can simply say that we have a hot observable that contains x amount of values.

158 Unit Testing Subscriptions

One of the more complicated things within an Angular architecture, is unit testing subscriptions. There are many scenarios within a UI application, wherein an observable cannot be passed directly to the html, for use with the async pipe. We have mentioned in a previous chapter the need of using the takeUntil operator, and that is not the intent here to discuss. What is the intent here, is to discuss how to unit test scenarios where functionality is happening within subscribe block.

158.1 Scenarios

Just for clarity sake, I would like to bring up some scenarios where unit testing is important within an Angular setting.

1. We are using subscribe data for a form. So that if data does exist, we can use the native Angular api for forms, and reset the form with data given by backend. However, we have the data by backend, has nested data when it is present. When it is not present, it is null. There are therefore some safeguards that need to be done by the front end inside of the subscribe block for those outlier use cases.

2. Backend does not intend to filter for a certain scenario, as it is a one off dashboard, for some clients. It is up to front end to filter based on limited backend data.

3. We have multiple projects for a user that they can use in their application. Data for the page will need to change when user selects a different project id. As a result, we will need to wrap data within the projectId we are using.

158.2 Mocking a Facade

Within our facade architecture[1], we will be injecting the facade into our Angular component/class. This allows us to mock the facade within our unit test, and control how we can interact with it within the unit test. In particular, we will always be passing in a observable,

[1]Which we have discussed thoroughly through out book

when is able to be modified. However, what we can do, is change the dynamics of how it becomes an observable.

Let's imagine that within our component we have a facade for the user.

```
constructor(private userFacade: UserFacade) {}

ngOnInit() {
  this.userFacade.user$.pipe(
    takeUntil(this.destroyed$);
  )
  .subscribe(user => {
    this.user = user;
    this.form.reset(user);
  });
}
```

Here, we are mimicking the 1st scenario we spoke about, wherein we need the data for the form reset we are using within the app. Let's say that we want to test, that the data is indeed being passed over the user, and form reset. In addition, want to set a safe guard, so that if a developer in the future accidently deletes any of the lines of code:

```
this.user = user;
this.form.reset(user);
```

The unit tests will complain. So let's move over to the unit test.

158.3 Unit Test

```
@Injectable()
class UserFacadeMock {
  userSubject$ = new BehaviorSubject({...userMock});
  user$ = userSubject$.asObservable();
}

describe('ComponentClass', () => {
  let component: ComponentClass;
  let fixture: ComponentFixture<ComponentClass>;
  let userFacade: UserFacadeMock;

  beforeEach(async(() => {
    Testbed.configureTestingModule({
      declaration: [ComponentClass],
      provides: [
       {
         provide: UserFacade,
         userClass: UserFacadeMock,
       }
      ]
```

```
    }).compileComponents();
  }));

  beforeEach( async( () => {
    fixture = TestBed.createComponent(ComponentClass);
    component = fixture.componentInstance;
    fixture.detectChanges();

    userFacade = TestBed.get(UserFacade);
  });
});
```

As we can see in the above we have layed out the structure for our unit tests, so that we can now control the value of user, by triggering the subject. Let's write that unit test we meant to get around to:

```
it('should properly pass in values from the subscribe block, to the' +
'component.user and component.form.reset', () => {
  const mockUserData: User = {...userMock};
  spyOn(component.form, `reset');
  userFacade.userSubject$.next({...mockUserData});
  expect(component.user).toEqual({...mockUserData});
  expect(component.form.reset).toHaveBeenCalledWith({...mockUserData});
});
```

158.4 Dis-secting What We've Done

Looking back at what we've done, by hijacking the UserFacade injected into our component, and supplying it with a subject, we have given our unit tests full freedom to test our subscribe blocks.

You might question, as to why it is that we do not directly change the observable to be a subject? This will cause type annotation errors, which is easier solved by simply creating a subject, and returning it as observable for the respective facade method.

159 Unit Testing TDD - First Princples Discovery

Unit testing as a discipline is something which is very hard to write. Writing a good unit test is as much as a discipline as writing good software. In addition, if someone is following TDD standards, then knowing of all the unit tests ahead of time can be very difficult. Leading many developers to leave the TDD environment. If they do write unit tests, it will be towards the end. The following is a great way, and as far as I am concerned the best way to discover unit tests.

159.1 What is the First Princples Thinking?

In Physics[1], there is a concept called, first principles thinking. This means boiling down a priciple to it's essnetial truth and then building up from there. With regards to Unit Testing, and specifically test driven development, this principle will help to create fantastic unit tests.

159.2 First Princples Thinking - Rubber Ducking - An example

```
1  Q: What would we like to get out of the choose size form?
2  A: We would like to specify number of rows, number of columns, and
       pixel size.
3
4  Q: Being that these are numbers, do we have any way of preventing
5  the user from entering in any value other than an number?
6  A: The input will only allow numbers.
7  Q: Ok, do you see any value in setting up logic, so that if it is not
       a
8  number, then it will throw an error?
9  A: No, because there will never be a situation wherein they can not
       put in a
10 number.
```

[1] Not a physicist, but I heard of this concept the first time from Elon Musk. I am, however, a Talmudist, and in talmud we have a similar concept called Pilpul

```
11  Q: In that case, should we take a snapshot and make sure input fields
        are
12  indeed of the type number. If not then it should error.
13
14  Q: Now that we have established that these are always numbers, is
        there any
15  limit on the size of this grid.
16  A: Not really, I can see it as just being what the window size.
17  Q: Hmm, I find that interesting, so you are saying that it can be any
        size.
18  A: Yes for this iteration.
19  Q: Okay, but it will be specific to window size.
20  A: Yes.
21  Q: Ok, so let's go ahead and create unit test, that it should have an
22  error, that based on window size, if value is greater, it should
        automatically
23  have it go to the height window size. In addition, there should be
        test that
24  specifies as such.
25
26  Q: Now that we have established that these we constrained window, is
        there any
27  limit on the ratio between column size, and row size?
28  A: No.
29  Q: Is there any way that we can create a unit test, to just say that
        it
30  passes, if there is a difference in ratio, between, then should not
        error out?
31  A: Yes, need to figure this out though. Not sure how to do off hand.
32
33  Q: Now that we have established that there are no constraints, what
        else is
34  there for us to test. We have covered window size, ratio, input type,
        what
35  about a max pixel Size?
36  A: Hmm, why would we care about that?
37  Q: Not sure, good point.
38
39  Q: Hmm, that being said, should we only allow pixel size that is
        within the
40  frame of the row and column size? For instance, let's say we have 20
        rows, and
41  20 columns, should the pixel size be something that perfectly divides
        into
42  400?
43  A: Not sure, it would be a lot of overhead at this point.
44  Q: Almost feel like we should automatically control pixel size based
        on column
45  and row size.
46  A: Ok, out of scope, let's not.
47
48  Q: Anything else?
49  A: Not that I can think of.
50
51  //End of Scene - dev takes a bow
```

To re-iterate in this book, we will not go into specifics of what is going on at a particular time. However, these unit tests can be found in the Angular Pixel Illustrator repo.

160 E2E Testing in a TDD/BDD Setting

One of the tricky things with regards to E2E tests, is how it fits into a TDD/BDD environment. Writing unit tests before we can see anything in our UI already takes quite a bit of discipline. Adding in an E2E test to the workflow seems like a bit much? Ok, so let's get into the thick of it. I think we'll all have a good time!

160.1 Where does E2E Testing fit in a TDD/BDD Setting?

160.1.1 Write an E2E test and Watch it Fail

Begin with E2E test, and watch it fail.

160.1.2 Write Unit tests and watch them fail

Write a unit test, that works towards satisfying the goal of your E2E test, and then watch it fail. One note, is that spectrum of what you will write, will be wider, being that a single E2E will have a wider scope. This is alright.

160.1.3 Code Until All Unit Tests are Satisfied

Implement code to satisfy your unit tests.

160.1.4 Optional - Tuck in any untucked corners

There might be some unit tests that will need to be written, that might not be directly correlated with the Protractor tests written for the E2E test. Writing these additional tests at this time, in between the next E2E test, would make sense.

160.1.5 Check to see if E2E Test Passes

When all unit tests pass, you should now run the command for running E2E commands.

160.1.6 Repeat the Process

Now that you have run your E2E tests, and unit tests, and have now code the appropriate HTML to run unit tests, you should now repeat process with appropriate unit tests.

Note: We will not be writing any E2E tests yet, as we do not have any two components to integrate yet

161 Automation Engineering

In an ideal QA environment, there will be a QA team layered on top of the Backend and Frontend Team. Their QA tests, however, will not be integrated into the environment of the aforementioned teams. Instead they will be in their own environment, and most likely running their tests in two environments: Dev, and productiong.

In particular, they should follow something called smoke testing.

161.1 Smoke Testing

Smoke Testing is preliminary testing to reveal simple failures severe enough to, for example, reject a prospective software release. Smoke tests are a subset of test cases that cover the most important functionality of a component or system, used to aid assessment if main functions of the software appear to work correctly. When used to determine if a computer program should be subjected to further, more fine-grained testing, a smoke test may be called an intake test

161.2 Automation Engineering - The Cross Over

Part of the question then becomes if automation engineering is creating their own integration tests for your app, it is essentially a bunch of E2E tests. So, why should UI go ahead and create their own tests, if QA is already creating them? Here are a couple of answers:

1. The earliest point in the lifecycle that QA will be able to test is in Dev. If UI Engineering does not create their own E2E tests, then failures in Dev, will then be sent back to UI, severly halting the workflow.

2. Redundancy. There might be different parts of the app that engineering is uniquely acute to testing. In addition, there might be other parts of the app wherin QA might be uniquely equiped to test. A great real life example of this is the following. We were integrating uploading files in the app for the first time, as well as sending emails. Being the developer who worked with both, I was acutely aware of odd file size of some of the documents being uploaded was 0. So I did some testing to make sure none of the files has a file size of 0, and lo and behold, our app was riddled with

them. That was something that QA would have never really thought of doing, simply because I had more access to the data.

3. Discipline. I think that it is important for UI engineering to get into the discipline of how their work will affect the actual user experience. Having UI Engineering actually write some of those tests is important.

That being said, it depends on your budget. If this is an app that is of high impact, then I would say do it. However, if it is something which your developers do not have time for, then I would say it is understandable. That being said, the following would be the limited crossover workflow in order for this budget friendly process to work.

161.3 Limited Crossover Workflow

161.3.1 Step #1

First and foremost, you are going to need a way to hook up automation engineering into your actual dev environment, and run those tests as a part of your actual dev environment. Keep in mind, this will make build times expensive, however, once again, it is for budget reasons.

161.3.2 Step #2

Make sure to loop QA into any concerns with the app you have. For instance, things that you've noticed the current QA tests do not cover, that you have noticed is an actual issue.

161.3.3 Step #3

Make sure to integrate automation tests within the CI/CD pipeline that you have. This means it will not fail when any of the E2E tests fail.

Regarding this last step, it is important that you optimize your tests so that they run in parallel. In addition, it will have to be a watered down version of integration tests. For instance, using Sauce Labs, for screenshots, as well as putting on multiple devices might be a bit cumbersome.

162 Writing E2E Tests

In addition to our Unit Testing, another very important element of our app is End to End Testing. We have discussed in the previous chapter how automation engineering is an integral part of this chapter. In addition, we discussed how when automation engineers are brought into the picture, it might be superfluous to have end to end testing. Nonetheless, I am a fan of having UI Engineers write their own E2E tests within the app.

163 Angular CLI

The Angular CLI will create out of the box a e2e folder that can be used for creating e2e tests. It can be run by

```
ng g e2e
```

. If you have an npm script within your app, running

```
npm run e2e
```

will work as well. Follow the steps through. Usually, the only thing that will be needed from your end, is downloading the chrome-webdriver needed for running the e2e tests. Suprisingly, there aren't actually any ways of generating an e2e test from within your angular cli.

164 Unit Testing Component using Apollo

Unit testing an app which has a DOM, is always a very difficult process. For instance, one can be familiar with how a certain technology should be stubbed. However, how one goes ahead and does so, is unique to that specific technology. With regards to UI, frameworks currently tend to change very quickly. In addition, thrre are many different frameworks, many of which use different testing frameworks. It goes without saying, if I have a chance to learn something and document it with regards to unit testing, it would be my honor to do so.

In particular, with regards to the Angular Apollo Client. Until recently (pre v1.1.0), using Apollo in an Angular App, was a very cumbersome ordeal. In particular, there was no testing module. One would have to spin it up themselves. However, now that we are dealing with v1.1.0, there is an official testing suite set up.

164.1 Re-visiting Component using Apollo

First let's look at your standard Angular component generated by the Angular CLI.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'ill-color-picker',
  templateUrl: './ill-color-picker.component.html',
  styleUrls: ['./ill-color-picker.component.scss']
})
export class IllColorPickerComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

164.2 Integrate Apollo with Component

```
+ import { Apollo } from 'apollo-angular';

+  ngOnInit() {
+    this.apollo
+      .query({
+        query: gql`
+          colors
+        `,
+      })
+      .subscribe((initialData: any) => {
+      });
+  }
```

164.3 Unit testing Apollo

First let's analyze what your generic Angular CLI generated spec will look like:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing'
    ;

import { IllColorPickerComponent } from './ill-color-picker.component';

describe('IllColorPickerComponent', () => {
  let component: IllColorPickerComponent;
  let fixture: ComponentFixture<IllColorPickerComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ IllColorPickerComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(IllColorPickerComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

164.4 Adding Apollo Testing Module

```
import {ApolloTestingModule} from 'apollo-angular/testing';
//..
+ ApolloTestingModule,
//..
```

The Apollo Testing Module will hijack the actual Apollo Module, and make it so that there is no error in particular with regards to creating an apollo client.

164.5 Hijacking the Component's GraphQL Request

After we have imported our ApolloTestingModule within the app, we now need to make it so that our GraphQL requests go straight to our fake backend request. If we do not, of course, the colors GraphQL request will attempt to go the default apollo link, making unit testing dependent on our GraphQL server.

164.5.1 Initializing the ApolloTestingBackend

```
1 + import {ApolloTestingBackend} from 'apollo-angular/testing/backend';
2
3 + const mock = new ApolloTestingBackend();
```

ApolloTestingBackend works by keeping a list of all open operations. As operations come in, they're added to the list. Users can assert that specific operations were made and then flush them. In the end, a verify() method asserts that no unexpected operations were made. [1]

164.5.2 Initializing the Apollo Link

```
1 + import {ApolloLink} from 'apollo-link';
2
3 + const link = new ApolloLink(op => mock.handle(op));
```

An Apollo Link as the documentation mentions, is a function that takes an operation and returns an observable. We are simply trying to repeat the process. In particular for the execute function.

164.5.3 Initializing the Operation

```
1  + const buildOperationForLink = (
2  +    document: DocumentNode,
3  +    variables: any,
4  +  ) => {
5  +    return {
6  +      query: document,
7  +      variables,
8  +      operationName: getOperationName(document) || undefined,
9  +      context: {},
10 +    };
```

[1] This particular description can be found in: https://github.com/apollographql/apollo-angular/blob/master/packages/apollo-angular/testing/src/backend.ts

```
11 +  };
12
13 + const operation = buildOperationForLink(query, {});
```

164.5.4 Running the Execute Function

```
1 + import {ApolloLink, execute} from 'apollo-link';
2
3 + execute(link, operation as any).subscribe(result => (response = result
    ));
```

The execute function will take link which is a mock link, as well as use our operation(i.e. GraphQL query) and return an observable that we can now use to get back our result.

165 Apollo Client Middleware

Middleware in general is a concept in many different plugins. In general, it means a service layer one can put on top of application in question. In particular, for an Apollo application, it will allow us to intercept the current request, and put in whatever we want.

165.1 Middleware as Architecture

Middleware can be considered architecture, because some plugins may not support middleware. In addition, even if they do support middleware, it might not be so apparent from documentation how one might go ahead and do so. So much so to the extent, that it will considerably slow down development, and a conversation will have to be had if research is worth it.

165.2 Middleware in Apollo

Apollo has a number of quirks with regards to it's middleware. For now, it adds typename to every query request. This can cause issues when it comes to unit testing. Let's create middleware to remove typename from the app:

```
const stripTypenameMiddleware = new ApolloLink((operation, forward) => {
  if (operation.variables) {
    operation.variables = omitDeep(operation.variables, '__typename');
  }

  return forward(operation);
});
```

Here we are stripping typename from the application.

165.3 Adding projectId to Requests

Using middleware, we have the option to add a projectId to our requests. Let's say that we need the data coming back to be specific to a certain project. Instead of having to

inject that on a request per request basis, we would like to go ahead and have apollo inject projectId as one of the query requests.

```
const attachProjectIdentifiers = new ApolloLink((operation, forward) =>
    {
  combineLatest(
    this.projectFacade.projectId$
  )
    .pipe(first())
    .subscribe(([projectId]) => {
      operation.variables = {
        ...operation.variables,
        projectId: projectId,
      };
    });

  return forward(operation);
});
```

In this fashion, one can eliminate clutter from apollo requests. In addition, one can alter apollo to have specific items in the response if need be.

166 Interfaces and Unions

In GraphQL, and more in particular, because we are working on things from the UI side of things, a Union type gives us the ability for one field to contain more than one field. The best way to really think of it, is in terms of Typescript. A union type would be:

```
type Book {
  title: String
}

type Author {
  name: String
}

type Result = Book | Author;

type Query {
  search: [Result]
}
```

As we can see, it is telling us that it can be one, of the other with regards to data type. So instead of returning irrelevant data, we only return the data we need. This is beneficial for a number of reasons.

1. It lightens payloads

2. Allows for succint sorting, on front end. I.e. do not have to sort from the backend.

166.1 Using Unions with Apollo Client

This is where at this time, things start to get really interesting. Using Apollo with unions/interfaces starts to become tricky. In particular, the client has no idea what is going on from the server side of things. Once we have different sorts of data for a singular query coming back based on the type of data avaialable, Apollo Client will just assume it is a certain type if it returns all data for a specific type.

Apollo made the decision to use something called the IntrospectionFragmentMatcher. It will look something like this:

```
const fragmentMatcher = new IntrospectionFragmentMatcher({
```

```
  introspectionQueryResultData: {
    __schema: {
      types: [
        {
          kind: 'INTERFACE',
          name: 'User',
          possibleTypes: [
            { name: 'User' },
            { name: 'UserWithReason' },
            { name: 'UserWithRound' },
            { name: 'UserWithBid' },
          ],
        },
      ],
    },
  },
});
```

It tells Apollo Client for Angular that it expects the above types in the interface for User. Where it starts to get really interesting is that Apollo Client requires for IntrospectionFragmentMatcher to be used regardless. So, in their documentation they reccomend that a script be used at build time, that creates a JSON file. This JSON file should then be used in the IntrospectionFragmentMatcher. This is very cumbersome, because if backend decides to throw something in for an already existing interface, it will cause the backend to break for that query.

I would like to repeat this once again, if you do not have an IntrospectionFragmentMatcher set up in your app, and backend adds union, or interface types, it will cause that data to break, and it will return an empty object.

It should be noted, that one might immediately think, why not go ahead and pull in the data for possibleTypes at runtime, and then use that for the IntrospectionFragmentMatcher? The difficult part about it, is that the way apollo works for the IntrospectionFragmentMatcher, is that it requires it to be passed into the cache. The cache is created on page load, and then having to update after cache is loaded, well you already missed the boat. So within the context of Apollo Client for Angular, the only way to load in possible types is to go along with the way documentation reccomends it.

166.2 What to Know Ahead of Time

What would be really helpful to know ahead of time, is that this is an issue that can only be solved by DevOps. It requires that back end and front end are built at the same time. That way, they both get pushed only once the new script is there. In this sitution, there really is no way about it. Having a mono repo architecture across your company will greatly

alleviate this process. [1]

[1] I have created this issue to bring awareness to this issue, but nothing so far

167 Data GraphQL

Assuming your app is using /hrefhttps://graphql.org/GraphQL, you will have to choose a client to use with GraphQL. Without going to into detail, we will be using Apollo Client as we truly believe it is best client for GraphQL. Within Apollo, there are numerous different files that one can create to interact with GraphQL.

167.1 Four Types of Apollo Client Files

Most notably there are four different types:

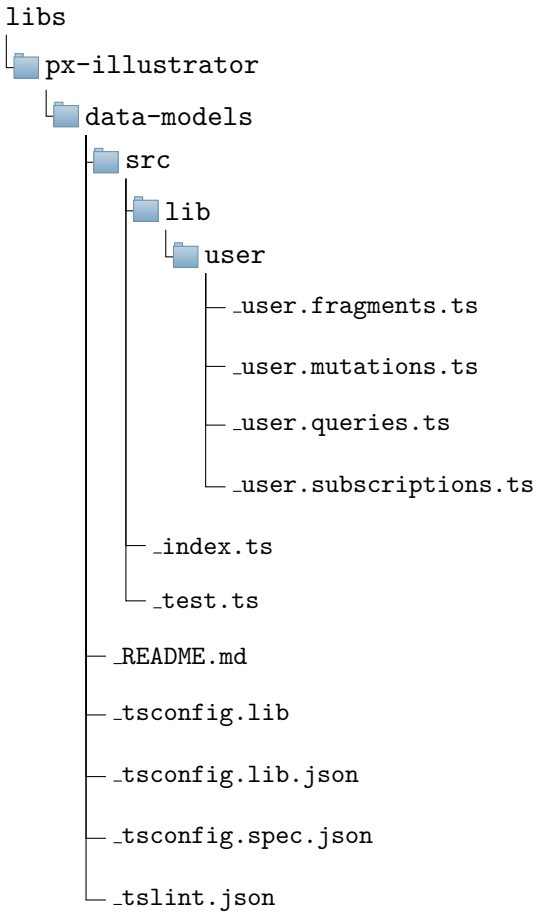
1. fragments
2. queries
3. mutations
4. subscriptions

This chapter is not a time for us to go into detail for each. However, the question is, where within our Angular Directory Structure should we go ahead and put it within our app?

167.2 Dissecting the Purpose of Apollo Client Files

It's important to understand that Apollo Client queries, mutations, and subscriptions will not only be used with their respective data-service. It is quite possible that multiple services will be using the same query, for instance, in numerous ways. In addition, the fragments used for a particular graphql query, mutation, or subscription, will be used within multiple apollo client files. It therefore make sense for the all apollo client files to be placed in their own disticnt folder within the libs folder(assuming we are using a mono repo), for the particular app. So now that we have decided it should warrant it's own folder let's take a quick look at how the Directory Structure might look like.

167.3 Data GraphQL Data Structure



As we proposed all files related to graphql are exclusively put into a single data-graphql folder. This alleviates the potential issues we mentioned above.

168 Versioning

A thank you, to a one Chris Bautista and Bobbie Barker for helping me formulate this strategy. In addition, thank you to a highly respected colleague Allan Goncalves. May you all be in good health.

168.1 Git

It goes without saying that you should be using Git for version control. Just to re-iterate the benefits of Git, are as follows:

1. Islolated Environments - I.e. every developer on your team works on the code base without affected others.

2. Pull Requests - Allows other developers to approve your code before officially being merged in.

3. Faster Release Cycle - Stability, distribution, pull requests, and community is streamlined.

Git should play as a foundational piece in your application. In addition, such a thing as Git architecture exists, especially within the context of an Angular application. Let's dive into that now.

168.2 Integration with JIRA

In any JIRA ticket, if one has the proper webhooks setup with Github, or if one is using BitBucket, then it will automatically hook up the commit into the ticket as a comment. All that is neccesary, is for the the commit message to include the ticket name. Will into this at another time. While this might not seem like architecture, in the sense of coding, this is actually the point of Git. Allow an immediate hook from your agile workflow into your versin control, by using the concept of feature branches. So first and foremost, make sure you have this integration set up.

168.3 Branching Name Convention

Understanding the different types of branches is equally just as important. In addition, what a branch name can accomplish. More specifically a Git branch can:

1. Inform the developer about the type of branch. These are:
    a) feature
    b) hotfix
    c) bugfix
    d) refactor
    e) cleanup
    f) e2e
2. Inform the developer of ticket being worked on.
3. Inform the developer of abstract of ticket.

Let's imagine our project is called "Pixe". We have a ticket called PIXE-113, which is responsible for adding a pixel color picker to our Pixel Illustrator app. We would create our branch as following:

```
feature/PIXE-113-details-and-actions
```

1. "feature" is issue type, all lowercase, followed by forward slash.
2. "PIXE-113" is the name of ticket in all caps, followed by a dash
3. details-and-actions is abstract of ticket.

Now, anyone looking through the current branch, can immediately find out the intent of the branch. It's a relatively low levle way of

168.4 Git Client

When it comes to using an IDE, I strongly believe that every team member should have the freedom to use whatever they want. It is very important, in general, that all team members feel free to use whatever they want. In this same vein they should feel comfortable in using a client if they would like. However, I personally prefer using the terminal. With regards to Git, I truly do feel it is what will ultimately let you work most efficiently.

168.5 Fork and Pull Workflow

There are numerous different Git workflows, however, the one I think is ideal, is a Fork and Pull Workflow. Your workflow will evolve on time, but this core will not change, and therefore I can confidently suggest it within this book. The workflow I am going to suggest is the fork and pull workflow. I am going to lightly describe what it is, it's benefits, and then delve more specifically into what it is.

168.5.1 What is a Fork and Pull Workflow

A traditional git workflow, is that each developer shares the same repository, it's just that they have different branches. In a fork and pull workflow, developers in addition to having their own branch, also have their own repository.

168.5.2 Benefits of a Fork and Pull Workflow

1. Allows for a single maintainer to accept commits from any developer without giving them write access to the official codebase.

2. No need to clean up git branches, as it is on each user's local repository.

3. Promotes developers to contribute to open source, being that this pattern is the defacto for open source projects.

4. A hidden one that I found myself doing, is doing side projects without worrying about it cluttering the main app.

5. If using a Mono Repo architecture, a fork and pull workflow is absolutely neccesary. At Razroo, we think a Mono Repo architecture is ideal, and have seen first hand, how it alleviates many things.

At Razroo, we use a Fork and Pull workflow, even when using private repositories. It allows for a cleaner workflow, and works better especially within a mono repo setting. Something we aspire to one day.

168.5.3 Setting up a Fork and Pull Workflow with Github

With Github, setting up a Fork and Pull workflow is as simple as:

1. Clicking on the fork button.
2. Clone your forked repo.
3. Sync repo with upstream

   ```
   git remote add upstream {forked from repo goes here}
   ```

   Now whenever you would like to pull from upstream dev branch can simply do:

   ```
   git pull upstream dev
   ```
4. push to your branch as usual
5. Pull branch from forked repo, while in original repo

168.6 Trunk Based Development

There are many version control strategies within Git. For this one, I've actually spoken in person to a number of highly respected software enginers. Trunk based development is what we all agreed on, is the ideal approach, granted you have an automated QA environment.

168.6.1 What is Trunk Based Development?

Trunk based development is that as opposed to the alternative of multiple branches for a repository, everyone commits to a core master branch.

168.7 Squash and Merge

Github, is our preferred client. However, git itself, as well as all other major 3rd party vendors, give the option to squash all other commits before merged. Some will offer the option to re-base as well. (If you are in a very Fortune 50 setting, and are working within a company wide Mono Repo, most likely you will want to re-base.) However, our reccomended aproach is to squash and merge, so you can maintain the integrity of the commits that happened along the way.

168.8 Ending Off

So there you go, the next time your team is wondering about what standards they should follow for Git, show them this. In the next chapter, we are going to dicusss, a proper Git workflow. I.e. master, dev, and test. We will get to that soon.

169 NG Container Hack for Structural Directives

In Angular there are two quirks with regards to structural directives:

```
<div *ngIf="someVariable">
  <p>This is text.</p>
  <p>This is more text.</p>
</div>
```

Listing 169.1: Quirk #1 - Requiring Extra Element to Implement Structural Directive

```
<ul>
  <li *ngFor="let box of boxes" *ngIf="box.item === 'food'">{{ box.name
      }}</li>
</ul>
```

Listing 169.2: Quirk #2 - Inability to apply more than two structural directives on the same element

The quirk probably naturally arises, because for loops, and if statements weren't meant to be used within a template. So if we think about it that way, wherein we can use some sort of Angular functionality, to bring it out of the DOM, it brings us to ng-container.

169.1 Understanding ng-container

In the Angular documentation, ng `ng-container` is specified as a way to group elements without introducing a new html element. Some of the examples mentioned include a `<span>` element that might introduce some accidental styling, or attempting to put a `<span>` inside of a `select` element. `ng-container` will allow for to side step those issues, by not introducing a new element to the actual DOM. We can use the `ng-container` to solve the quirks we mentioned earlier:

```
<ng-container *ngIf="someVariable">
  <p>The show goes on.</p>
  <p>and on and on and on.</p>
</ng-container>
```

Listing 169.3: Solution to Quirk #1 - Requiring Extra Element to Implement Structural Directive

Here, by introducing an ng-container, we no longer have to introduce a new div, if we want the content to show conditionally. Likewise, to solve the issue we had before of being able to use two structural directives, we can do the following:

```
<ul>
  <ng-container *ngFor="let box of boxes">
    <ng-container *ngIf="box.item === 'food'">
      <li>{{ box.name }}</li>
    </ng-container>
  </ng-container>
</ul>
```

Listing 169.4: Solution to Quirk #2 - Inability to apply more than two structural directives on the same element

We could technically apply the `*ngIf` on the `li` element itself, for consistency sake. If I had a team member that preferred otherwise, I would be more than happy with that, just my preference.

Thank you to Austin Spivey for being the person where I saw this approach from.

170 Npm Vs. Yarn

One of the inevitable conversations that comes into play when starting to use an app for the first time, is whether, or not to use yarn. Once upon a time, yarn introduced a yarn.lock file. This was fantastic, and really did increase install time. In addition, the syntax was a bit more intuitive, such as yarn add, vs npm install. However, as time progressed, NPM received it's own lock file. Performance time between the two became negligable. In addition, yarn always came with it's own set of problems. I remember a package giving me an error once that really took me sometime to solve.

170.1 Reasons to stick with NPM

1. There are no noticable differences between Yarn and NPM
2. NPM has been borrowing features from Yarn.
3. NPM has it's own package lock.
4. NPM Ci, will tell differences between package.json and package.json.lock.

That's really all the argument there is need to be had. At this point in time Yarn really does not offer anything valuable in ways of NPM. I am thankful for the contributions they have made to ecosytem. However, having the entire team use Yarn, when it doesn't come at a particular advantage and at times at a dis-advantage does not make complete sense.

So, if your team is curious which package manager to go with, unless they have a personal preference as a collective unit, feel free to tell them NPM.

171 Weekly Meetings

Weekly Dev Meetings is something that can be extremely beneficial to the team at large. A large app that is managed without a planned, methodic method of approaching the code base, might end up getting tangled in places here and there. In addition, it is a great forum for software engineers to be able to show off what they've seen. Some items that have been discussed in meetings, include the data-access architecture, organizing of interfaces. This helps to tidy corners in places wherein help may not have been an option beforehand.

172 Creating a component

For re-iteration purposes, the definition of a component is something consituting of a larger whole. Ideally anything we can turn into a component in an Angular environment, will help us. In addition, anything which we can re-use across the app, is beneficial as well.

When creating components, Angular also makes use of modules. Once again, just to re-iterate, a module is an independent unit, which is used to construct a larger interallated construct.

Angular stays true to these two definitions. A component can only be declared by one component. If it is used by two, or more Angular will complain, saying that it is already used by another component. Which by definition only constitues a larger whole. A module on the other hand, is simply an independent unit. If we ever want to use our component with two components, we will need to include it as part of a module.

Therefore, it is reccomended as general good practice, whenever creating a component (unless that particular component has children), to always create it with a module. For other reasons as well, it is smart idea. We will get into those later [1].

We have already created a component as needed for our router, but for redundancy sake here are the steps again.

Also, because we will be using sass, let's make sure that our cli is using sass. Open up the .angular-cli.json file, and change two areas. One:

```
ng set defaults.styleExt scss
```

This will make it, so that whenever we set up our components using the cli, again it will be in sass. Second, change your existing styles.css file to styles.scss.

Let's use the cli to create our first module called choose size.

```
ng g module choose-size
ng g component choose-size --exports
```

[1] If you can't wait, and want the full list now, go here to find it

(The file at this time is included in our app as a route. Let's remove the default nrwl text from app, so that all we have is choose-size works.)

172.1 Architecture time

Before, we haphazardly created a component in order to introduce routers. Now that we are going to work on our actual component, let's set aside to specific items with regards to architecture.

Whenever we want to create a page for our application that will be used as a route, it is a container. Something which is simply there to "contain" all of our components. In the root of our app directory we are going to create a container folder. [2]

```
mkdir containers
```

We will also be needing to mention, that we will be moving our choose-size directory to a newly created components folder.

cd into your containers folder, and create a choose-size-page module/component:

```
ng g module choose-size-page
ng g component choose-size-page
```

In this choose-size-page component, we will be adding our choose-size component.

In order to do so, we will need to import the choose-size component in our Angular app, and add it to our choose-size-page module like so:

```
1 import { ChooseSizeModule } from '../../components/choose-size/choose-
    size.module';
2
3 @NgModule({
4   imports: [
5     CommonModule,
6     ChooseSizeModule
7   ],
8   declarations: [ChooseSizePageComponent]
9 })
```

Listing 172.1: Importing the choose-size module

[2] We are once again borrowing from the example-app project in ngrx/store

In addition, we are going to want to make sure to add an exports key/value to our choose-size module, so that by importing it, we have the respective component available as well.

```
1  @NgModule({
2   imports: [CommonModule],
3   declarations: [ChooseSizeComponent],
4   exports: [ChooseSizeComponent]
5  })
```

Listing 172.2: Adding choose-size component as export

With the component module properly imported, we can now use the component in our choose-size-page html file:

```
// choose-size-page.component.html
<app-choose-size></app-choose-size>
```

173 Creating a Second Component

Creating a second component within your app is a monumentous occasion. It lays down the groundwork for how you are going to integrate numerous components together within your app. Let's lay down at a high level what this means with regards to a Mono Repo, in an enterprise Angular setting:

1. Lib Folder
2. Angular CLI Command
3. Ngrx/nx ngrx considerations
4. Routing Considerations
5. Service Considerations
6. Pipe Considerations
7. Responsive Considerations

173.1 Angular Pixel Illustrator - Example Use Case

Let's take the above considerations into our app. The next component we are going to create is a color picker. It will contain two sub components, a background color picker, and a pixel color picker, re-used by a singular component.

It will most definitely be going to into the lib folder. We will be using the angular cli. We will be creating state, called color. With regards to a route, we do want to create a generic route, that will switch out from the pixel grid chooser, over to a a color picker view. We are also going to want a consideration for mobile as this is a PWA. There aren't any pipes we will be using.

173.2 Dissecting Business Requirements for Color Picker

With regards to the color picker. There are three unique aspects of the business logic:

1. Color picker + Background Color Picker - Shared Logic
    a) RGB
    b) HEX
    c) Convert RGB to HEX and vice versa
    d) Have color bar below pixel picker, change based on color value.
2. Background Color picker
    a) Change grid background, based on background color picker.
3. Color picker
    a) Change pixel color, based on pixel color picker.

173.3 Creating a Second Component - Putting it all Together

As we mentioned in the beginning in this chapter, this process is going to be created each time we create a new component. Therefore, it would make sense to sum up this process so we can repeat the process in every app. In addition, for redundancy sake, we will go back to this checklist, and repeat the process.

Color Picker	Need	Do Not Need
@ngrx/store	✓	
Route	✓	
Service	✓	
Pipe		✓
Responsive	✓	

Regarding Business requirements:

```
Scenario: When Using the Color Picker
  Given I enter Hex
  Given I enter RGB
  Then I should see state affected appropriately.
```

You will notice for this particular component, the Then of our Aceptance criteria, is oddly focusing on State. This is because due to the dynamics of our system, this is what we are trying to focus on. However, if a product person were to create it, one can expect it to be more hollistic, and to focus on updating the grid as appropriate instead.

174 Creating a Dumb Component

174.1 Outline

1. Create a UI dumb component in Lib folder
    a) Use CII to generate module
    b) Use CII to generate component
2. Import module into appropriate page
3. Add component to page html
4. Add proper styling to the illColorPicker

174.2 Create a UI Dumb Component in Lib Folder

This is a dumb component, it should be created in the UI folder. [1]

```
ng g lib color-picker --routing --directory="ui"
ng g component color-picker -a=color-picker --export
```

174.3 Add Color Picker to Pixel Grid Page Component

Let's import our ill color picker into our Pixel Grid page.

```
// pixel-grid-page.module.ts
+ import { IllColorPickerModule } from "@ill/ill-color-picker";
// in imports array
+ IllColorPickerModule
```

In addition, let's add the ill-color-picker component to our html.

[1] Please reference the chapter discussing UI architecture for why that is.

```
+ <ill-color-picker></ill-color-picker>
```

174.4 Add Styling Element to Create Basic

Let's add a basic width and height for out element.

```
.IllColorPicker {
  display: flex;
  flex-direction: column;
  width: 200px;
  height: 100%;
}
```

175 Technical Design Notes

When creating a ticket, it is important that technical design notes be written as a part of actual JIRA ticket.

175.1 What are Technical Design Notes?

Technical Design Notes are a way to abstract the decisions one will make towards architecting an app.

175.2 Benefits of Creating Technical Design Notes

It allows the app to be thought through before app is built. Saving time on re-factoring code, ensure code quality, and retain confidence that tickets will be done the way they should be done.

175.3 When to Create Technical Design Notes

Technical Design Notes can be cumbersome, and writing them does not make sense in all instances. In one of two situations, technical design notes should be created, when one, or more of the following is true:

1. When Unit Testing is involved. For instance, let's say we have a filtering component, and we need to test what will happen if a user inputs a word with a space in it, or with a uppercase character.

2. When strategy architecture is involved. For instance, we need to create a strategy for routing, or how we will end up pulling in data. Sometimes, it is something which will be an unknown, and saying this is what I am trying to figure out, and it is a unknown is more than perfect.

175.4 What Goes into Technical Design Notes

It should mention at a very high level, what should go into the component. For instance, if I am building a filter, it should mention:

1. That I plan on using ngrx/store in order to store filters.

    a) Specifically as strings

2. Will be using ¡md-input¿ material design component for filters.

3. Will be writing integration test for filters individually, and how they will interact with each other.

4. Will be creating filters for the following test scenarios.

    a) Camel case

    b) Space in filter

    c) Pure text

    d) Dates

176 Acceptance Criteria

Acceptance Criteria should generally not be in the court of the software engineer. However, as is quite common in software engineering, product will need a bit of prodding from engineering, in order to discuss what it is that they are looking for.

176.1 Real Quick - What are Acceptance Criteria?

They are the conditions that a software product must satisfy to be accepted by a user, customer, or in the case of system level functionality, the consuming system.

176.2 Ideal Syntax for Acceptance Criteria?

After being in a number of settings, the ideal way to create acceptance criteria is to use Cucumber/Gherkin syntax.

176.3 What Gherkin Syntax?

Gherkin is a syntax which supports BDD. It is aimed at making executable specifications written in plain language[1]:

```
Scenario: eat 5 out of 12
Given there are 12 cucumbers
When I eat 5 cucumbers
Then I should have 7 cucumbers
```

[1] We will get to how we will integrate this with our QA efforts in a moment

176.4 Why is it important that we use Gherkin Syntax?

Gherkin syntax is designed to be succint, and easily understandable which it is. In addition, it is a syntax that the entire company can rally around, being that it will be used by Automation Engineers as well[2]. It is also in my experience, the only way to convince product to write acceptance criteria that actually stays the same from ticket from ticket, but don't tell them I said that!

176.5 Sample Ticket Creation for Choose Size Form.

```
Scenario: When Using the Choose Size picker
  Given I enter pixel size
  Given I enter column size
  Given I enter row size
  When I  click on the Create Pixel Grid button
  Then I should see grid created
```

Note that this is the ticket for creating a component for the choose size picker. However, we still do now know what the design will look. This we leave for the next chapter.

[2]Surpise! They will be using Gherkin as well, unless you knew that one already. In which case it is not a suprise.

177 Ticket Creation - Component Design

We have discussed the two initial steps with regards to creating a ticket. Technical Design Notes, and acceptance criteria. The third and final step with regards to any good ticket is design. The chapter regarding talking to UI/UX is not here, but you should be using JIRA as your ticketing system.

Within a JIRA setting, two things should happen so you have a clear idea of how a component should be designed within a PWA setting:

1. Description

2. Invision Link within JIRA

177.1 Component Design Quirk

When creating a ticket in a PWA environment, there is a need to create a specify the specific functionality around mobile, and desktop. Many times, the functionality is not the same. In addition, while we develop from a mobile first perpective, it is not the case for business and product. For engineers, there is an understanding that whatever is not used for mobile, is used for desktop. This is not the case for business. They look at the two as two separate entities.

In addition, understandably for design, they also look at mobile and desktop as two different entities.

It is therefore reccomended that you will have to create two separate tickets.

177.2 Development Corner

Having two tickets for mobile and development will cause conflict with regards to pull requests. In order to solve this concern, make git commit's against the web ticket. In addition, in your JIRA ticket, make mobile dependent on the web ticket.

178 Code Reviews

Code reviews are a very intuitive process. It can potentially be looked at as something that I would do. If the pull request isn't looking at the code the same way I would, then I should comment. However, it's the part of commenting and accepting criticism, that makes this is entire process very tricky.

178.1 Code Reviews - The Golden Rule

There are multiple ways of doing something. If the code reviewer leaves a comment for doing something in an alternate way, and the person recieving the code resists, then the code reviewer has no right to insist on her/his way. It is then important, however, from that point onwards, that the team agrees on a convention.

178.2 Story Time

The following is a great example of how this golden rule can manifest in real life. Once upon a time, my team was working on a component, wherein every tab was to be in uppercase. The person submitting the code felt that explicitly typing out every word in uppercase: NAME, STATUS, TIME; Made more sense. I expressed that adding a css class with text-transform: uppercase; would make more sense. The pr submitter expressed that they felt explicitly typing out everything made more sense. I mentioned that ok, I can see your point of view, and I will remove my comment.
The truth is that if this was a B2C[a] application, then I would have been adament about my approach. However, this was a B2B[b] application, and allowing this person to code the way they feel comfortable and be happy, because life really is too short, is ultimately what is important.

[a] Business to Consumer
[b] Business to Business

178.3 Setting Conventions

Another important part of the code review process is to set conventions. In my humble opinion, the reason conflict happens with regards to code, is due to uncertainty. When there are conventions set up before work on code happens it is to point to code guidelines and say this is what we do. In addition, somone has the ability to challenge code guidelines, and it is more challenging code guidelines. This allows those involved in discussion to save face.

178.4 A Time to Learn

It is also very important for others to learn. It is a way for me as the code reviewer to ask what a certain piece of code does. A good convention is that if it is something new that you haven't done before, then you can ask about it and learn about it. This is one of those points that is obvious yet it is passed on more than most.

178.5 A Time to Mentor

It is also a time to setup CODEOWNERS across the app, and specifically make junior developers code owners. Thereby mentoring them and bringing them up to speed on what needs to be done.

179 Pixel Grid Container

Now that we have created our Choose Size Form, let's go ahead and create our own route for our presentation container.

179.1 Create Pixel Grid Container Component

```
cd containers
ng g module pixel-grid-page
ng g component pixel-grid-page --export
```

179.2 Import Pixel Grid Container Component

In your app module, import the pixel-grid-page module. In the app routing module, create a route for the pixel-grid-page.

```
//app.module.ts
+ import { PixelGridPageModule } from './containers/pixel-grid-page/
    pixel-grid-page.module';

  ChooseSizePageModule,
+ PixelGridPageModule,
  NxModule.forRoot(),
```

179.3 Set up Router Link

In any situation wherein a page has been created with routing, we are going to need to create a routeLink, which will hook into the route which will be created. In particular, in our app, we will be creating a routerLink on the Create Pixel Grid button.

```
//choose-size.module.ts
+ import { RouterModule } from '@angular/router';

+ RouterModule,
```

```
//choose-size.component.html
+ <button routerLink="/pixel-grid"

+ RouterModule,
```

180 Pixel Grid Container Layout

Now that we have introduced Flex Layout and our designs call for three elements:

1. Code Viewer
2. Pixel Grid
3. Color Picker

180.1 Anticipate for Future Components

We know that we will have three components that will be set up side by side. On our main page component, which will contain all three, we will set the following three fxLayout directives:

```
fxLayout="row"
fxLayout.xs="column"
fxFlexFill
```

The above is pretty straight forward. On screens not extra small, the layout will be flex row. When the screen is extra small, the layout will be column. With regards to fxFlexFill, it will populate the host element with the following:

Key	Value
margin	0
width	100% [1]
height	100%
min-width	100%
min-height	100%

[1] Taken from documentation here: https://github.com/angular/flex-layout/wiki/fxFlexFill-API

180.2 Adding FxLayout to Pixel Grid Page

180.2.1 Add Flex Layout to App

```
npm i --save @angular/flex-layout
```

180.2.2 Add Flex Layout to Pixel Grid Page Module

```
1 +import { FlexLayoutModule } from '@angular/flex-layout';
2
3 +    FlexLayoutModule,
```

adding-a-route-to-our-container

181 Color Picker

Let's go through the steps again, being that we are creating our second component. Part of learning is not only discovery, but maintenance. As discussed in the preface, whenever we have the chance to repeat steps we have discussed once before, we will re-iterate them on a high level for memory sake.

181.1 Outline of what needs to be done

1. Create a UI dumb component for color picker in Lib folder
    a) Should be generated in app folder in UI folder.
    b) Use CII to generate module
    c) Use CII to generate component
2. Import module into pixel-grid page
3. Add component to pixel-grid page html
4. Add proper styling to the illColorPicker

181.2 CLI - Creation of Module and Component with Routing

First, let's create our component in the lib folder of our app.

```
ng g lib color-picker --routing --directory="dealworks/ui"
ng g component color-picker -a=color-picker --export
```

181.3 Add Color Picker to Pixel Grid Page Component

Let's import our ill color picker into our Pixel Grid page.

```
// pixel-grid-page.module.ts
+ import { IllColorPickerModule } from ``@ill/ill-color-picker';
// in imports array
+ IllColorPickerModule
```

In addition, let's add the ill-color-picker component to our html.

```
+ <ill-color-picker></ill-color-picker>
```

181.4 Add Styling Element to Create Basic

Let's add a basic width and height for out element.

```
.IllColorPicker {
  display: flex;
  flex-direction: column;
  width: 200px;
  height: 100%;
}
```

182 Constants

182.1 What is a Constant?

In Javascript the idea of having a constant would be assigning a variable, to a particular value. Whenever we would like that value, instead of typing out the value, we would use the variable. At first, however, it might seem counter-intuitive. Why use the constant value, if it is literally named the same thing as the actual value?

```
// Update last updated value to have latest payload data
const LAST_UPDATED = "LAST_UPDATED";
new updateValue(payload, LAST_UPDATED);
```

Listing 182.1: Example of a Constant

182.2 Benefits of a Constant

182.2.1 Creates a Table of Contents

When one creates a series of constants in particular file for a certain component, one can peruse through the constant file, being able to see all actionable items in one. For instance, the following:

```
// imagine these constants, are in a folder called ValueActionTypes
const UPDATE_VALUE = "UPDATE_VALUE";
const ADD_VALUE = "ADD_VALUE";
const DELETE_VALUE = "DELETE_VALUE";

//imagine the following code is in a new folder called valueActions
import * as types from "../ValueActionTypes";
import {BuyerValue} from './buyer-filter.interfaces';

export class UpdateValue implements Action {
  readonly type = UPDATE_VALUE;

  constructor(public payload?: BuyerValue, public keyName?) {};
}

export class AddValue implements Action {
  readonly type = ADD_VALUE;
```

```
18
19    constructor(public payload?: BuyerValue, public keyName?) {};
20  }
21
22  export class DeleteValue implements Action {
23    readonly type = DELETE_VALUE;
24
25    constructor(public payload?: BuyerValue, public keyName?) {};
26  }
```

Listing 182.2: Example of a Constant

182.2.2 Communicate to Maintainers

If using a constant value, it signifies to future maintainers of your code, that this is a value which is immutable. Further distinguishing intent of application/snippet of code.

182.2.3 Helps Secure Values

When typing in a string for a particular constant value, particular diligence must be applied in order to make sure it is type appropriately. Typing in the one place, allows the developer to type in one place, and simply copy and paste value, from a singular expected location(the const file) to 5 different places.

183 Enums as Constants

When working with Typescript, which if you are using Angular, then you most definitely are using Typescript [1]. Care must be taken to look into all the nuances that Typescript can offer.

183.1 In a non Typescript Setting

In order to define a constant in a non-Typescript setting, we use the const declaration to define variables:

```
1  const UP = "UP";
2  const DOWN = "DOWN";
3  const LEFT = "LEFT";
4  const RIGHT = "RIGHT";
```

183.2 Enums an Introduction

Simply put, Enums allow us to define a set of named constants [2].

```
1  enum PlaneActionTypes {
2      Up = "[Plane] Up",
3      Down = "[Plane] DOWN",
4      Left = "[Plane] LEFT",
5      Right = "[Plane] RIGHT",
6  }
```

183.3 Benefit of Enums over Constants

Enums allow us to organize a collection of related values. Think of them as a class for values, wherein the value can only be a string , or number.

[1] I look forward to working with Reason sometime soon for typechecking, but I digress.
[2] https://www.typescriptlang.org/docs/handbook/enums.html

183.4 Current quirk of String Enums

String Enums, as opposed to number Enums, have to be constant initialized with a string literal. To clarify, you might want expect the following to work:

```
  const prefix = '[Button]'
  enum Direction {
      Up = `${prefix} UP`,
      Down = `${prefix} DOWN`,
      Left = `${prefix} LEFT`,
      Right = `${prefix} RIGHT`,
  }
```

However, this does not work , because this is not a string literal, i.e. string only.

183.5 Convention as a Result of Quirk

As a result of quirk, we need a way of specifying that this action is happening in relation to a specific object. Even though we do have a set using enums, when identifying the string on it's own, from a state management (dev tool) perspective, or console perspective, it will be beneficial to have the string literal, be explicit on it's own. [Screenshot of a redux devtool would great]. Please reference above section, "Enums as an Introduction", for how this translates to code in principle.

183.6 Side Note - Why No All Caps in Enums?

A const in Javascript can actually be re-assigned to something else. For instance:

```
const PLANE = 'blackbird';
PLANE = 'thunderbird';
// barf
```

It is therefore a good convention when using a const, to put it in all caps, when the value is not attend to be re-assigned such as:

```
const PLANE = 'blackbird';
// woh, I was about to re-assign plane to thunderbird for some weird
    reason, but
// then I saw PLANE in all caps, so I didn't do it
```

However, this is not the convention with Enums, of course, because all enums are never re-assigned. It is therefore not necesary to to write in all caps.

184 Authorization

Authorization is a corner stone of any project. Many Greenfield projects will not have the ability to implement authorization right away, as backend will not have the capacity to do so. However, one of the cool things about authorization is that one has the ability to set it up ahead of time. When data from the backend comes in, the authorization service and directives will be ready to do.

TODO chapter on creating a centralized service goes here.

184.1 Creating directives for our service

As an example, let's say that we have html that we want to disable, or hide. We can do the following:

```
<div [myHideIfUnauthorized]="updatePermission"> <!-- a property set or
    passed into the component âĂŞ>
<div [myDisableIfUnauthorized]="updatePermission">
```

One is then going to want to create two different directives. One for disabling if unauthorized:

```
import { Directive, ElementRef, OnInit, Input } from '@angular/core';
import { AuthorizationService } from '../../services/authorization.
    service';
import { AuthGroup } from '../models/authorization.types';

@Directive({
    selector: '[myDisableIfUnauthorized]'
})
export class MyDisableIfUnauthorizedDirective implements OnInit {
    @Input('myDisableIfUnauthorized') permission: AuthGroup; // Required
         permission passed in
    constructor(private el: ElementRef, private authorizationService:
        AuthorizationService) { }
    ngOnInit() {
        if (!this.authorizationService.hasPermission(this.permission)) {
              this.el.nativeElement.disabled = true;
        }
    }
}
```

and another for hiding if unauthorized:

```
import { Directive, ElementRef, OnInit , Input } from '@angular/core';
import { AuthorizationService } from '../../services/authorization.service';
import { AuthGroup } from '../models/authorization.types';

@Directive({
    selector: '[myHideIfUnauthorized]'
})
export class MyHideIfUnauthorizedDirective implements OnInit {
    @Input('myHideIfUnauthorized') permission: AuthGroup; // Required permission passed in
    constructor(private el: ElementRef, private authorizationService: AuthorizationService) { }
    ngOnInit() {
        if (!this.authorizationService.hasPermission(this.permission)) {
            this.el.nativeElement.style.display = 'none';
        }
    }
}
```

184.2 Creating a Guard for unauthorized

As the last piece of our unauthorized trifecta, we will be wanting to create a guard. For reference on guards, please refer to the chapter on guards.

TODO - Go more into depth on what this guard is doing.

```
import { Injectable } from '@angular/core';
import { CanActivate, Router, ActivatedRouteSnapshot } from '@angular/router';
import { AuthorizationService } from './authorization.service';
import { AuthGroup } from '../models/authorization.types';

@Injectable()
export class AuthGuardService implements CanActivate {
    constructor(protected router: Router,
          protected authorizationService: AuthorizationService) { }
    canActivate(route: ActivatedRouteSnapshot): Promise<boolean> | boolean {
        return this.hasRequiredPermission(route.data['auth']);
    }
    protected hasRequiredPermission(authGroup: AuthGroup): Promise<boolean> | boolean {
        // If userâĂŹs permissions already retrieved from the API
        if (this.authorizationService.permissions) {
            if (authGroup) {
                return this.authorizationService.hasPermission(authGroup);
            } else {
```

```
                    return this.authorizationService.hasPermission(null)
                        ; }
          } else {
               // Otherwise, must request permissions from the API first
               const promise = new Promise<boolean>((resolve, reject) =>
                    {
                    this.authorizationService.initializePermissions()
                    .then(() => {
                       if (authGroup) {
                            resolve(this.authorizationService.
                               hasPermission(authGroup));
                       } else {
                            resolve(this.authorizationService.
                               hasPermission(null));
                       }

                    }).catch(() => {
                        resolve(false);
                    });
             });
             return promise;
        }
    }
}
```

184.3 Service Can Be Called Anywhere

We have the option to call the auth service anywhere in the app that we want to, in addition to using directives and the guard. For instance:

```
private showMenuItem(authGroup: AuthGroup) {
   return this.authorizationService.hasPermission(authGroup);
}
```

185 Building our Application

In order to go through the full gamut of Angular, we are going to focus on as lightweight of an application as possible. In order to go through entire Angular architecture, we obviously do not want to over due it, nor do too little. It goes without saying, that your classic todo app, will not suffice. Instead the following is the application that we will be building.

We are going to call it a pixel to coordinate illustrator. The idea behind the app, is that we should have a canvas, paint a pixel on that canvas. We then have a coordinate, that will appear based on where pixels are currently located.

In our application we have:

1. Form
    a) Pixel Size
    b) Number of Rows
    c) Number of Columns
2. Color Picker
    a) Background Color Picker
    b) Pixel Color Picker
3. Pixel Canvas
    a) Pixel Grid
    b) Ability to Remove Pixel
    c) Ability to Add Pixel
    d) Ability to Change Pixel Color
4. Coordinate Viewer

a) Show x Coordinate

b) Show y Coordinate

c) Show Pixel Size

d) Show Pixel Color

Application will be made responsive. Without further ado, let's begin our applcation.